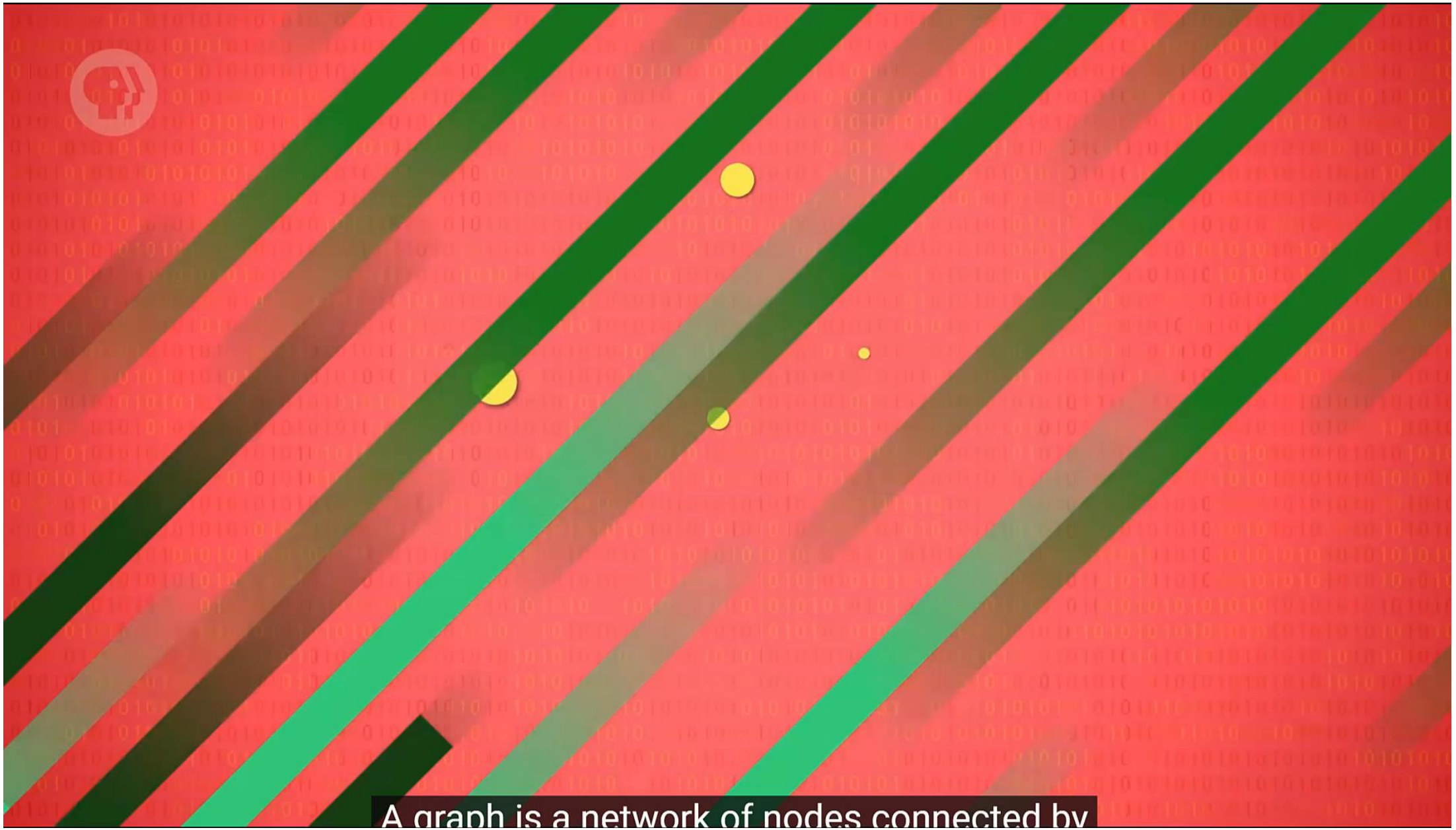
CSE 3318

Week of 07/15/2024

Instructor : Donna French



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)

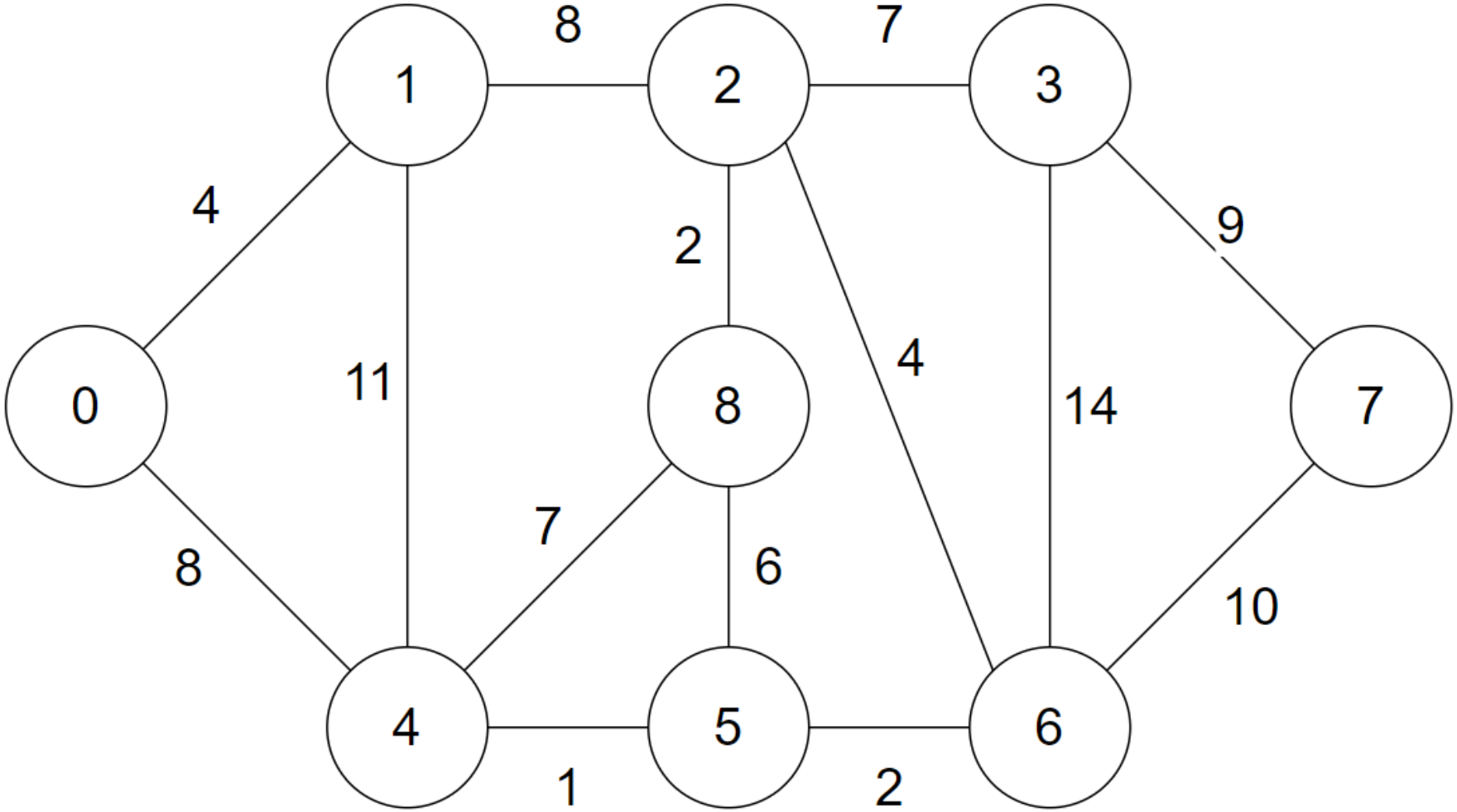
Works on a weighted graph.

Starts with an initial vertex and has a goal vertex.

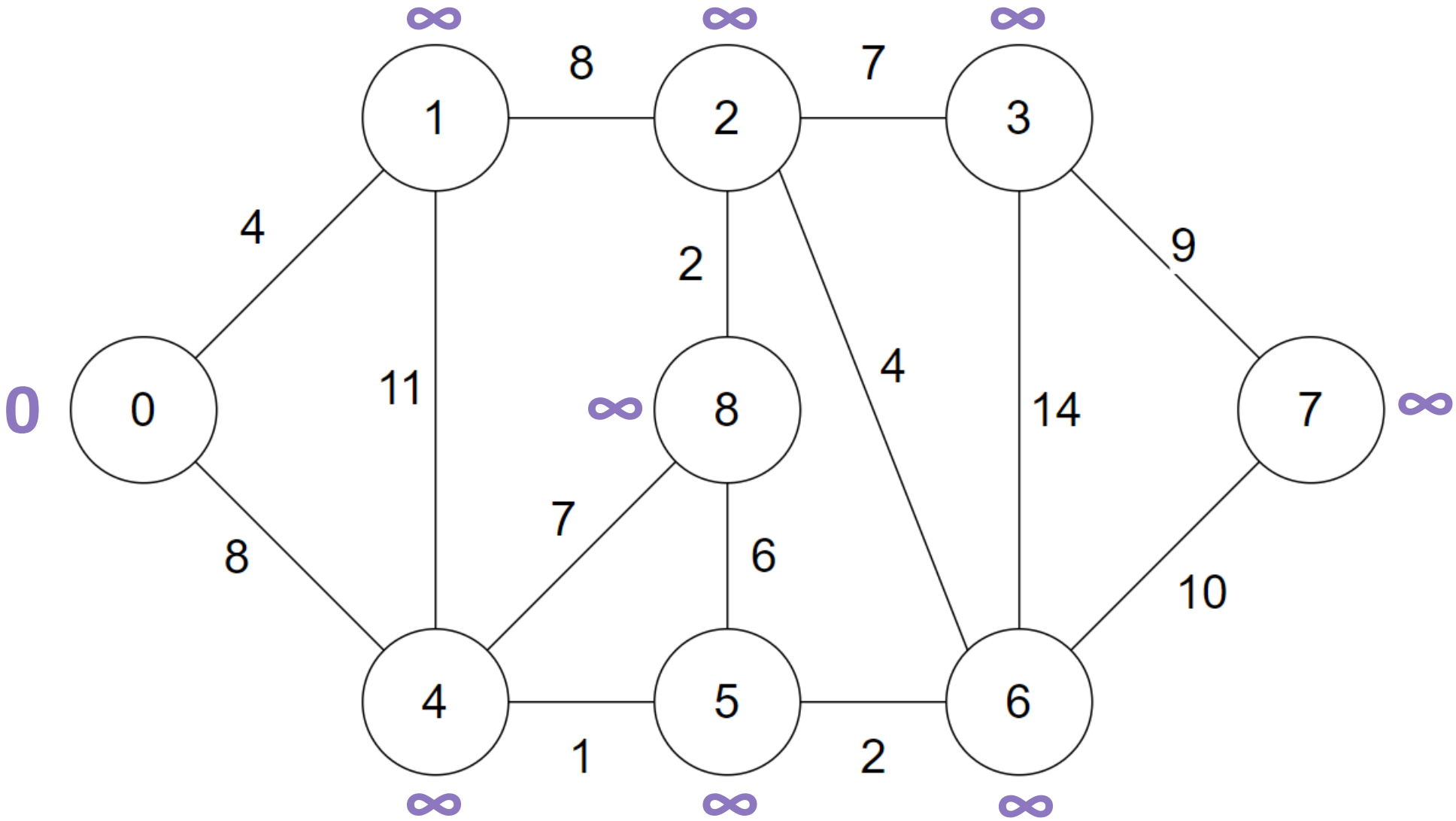
Finds the path with the lowest cost where the cost is based on the weights assigned to edges.

Path may not be unique.

Dijkstra's Algorithm



Dijkstra's Algorithm



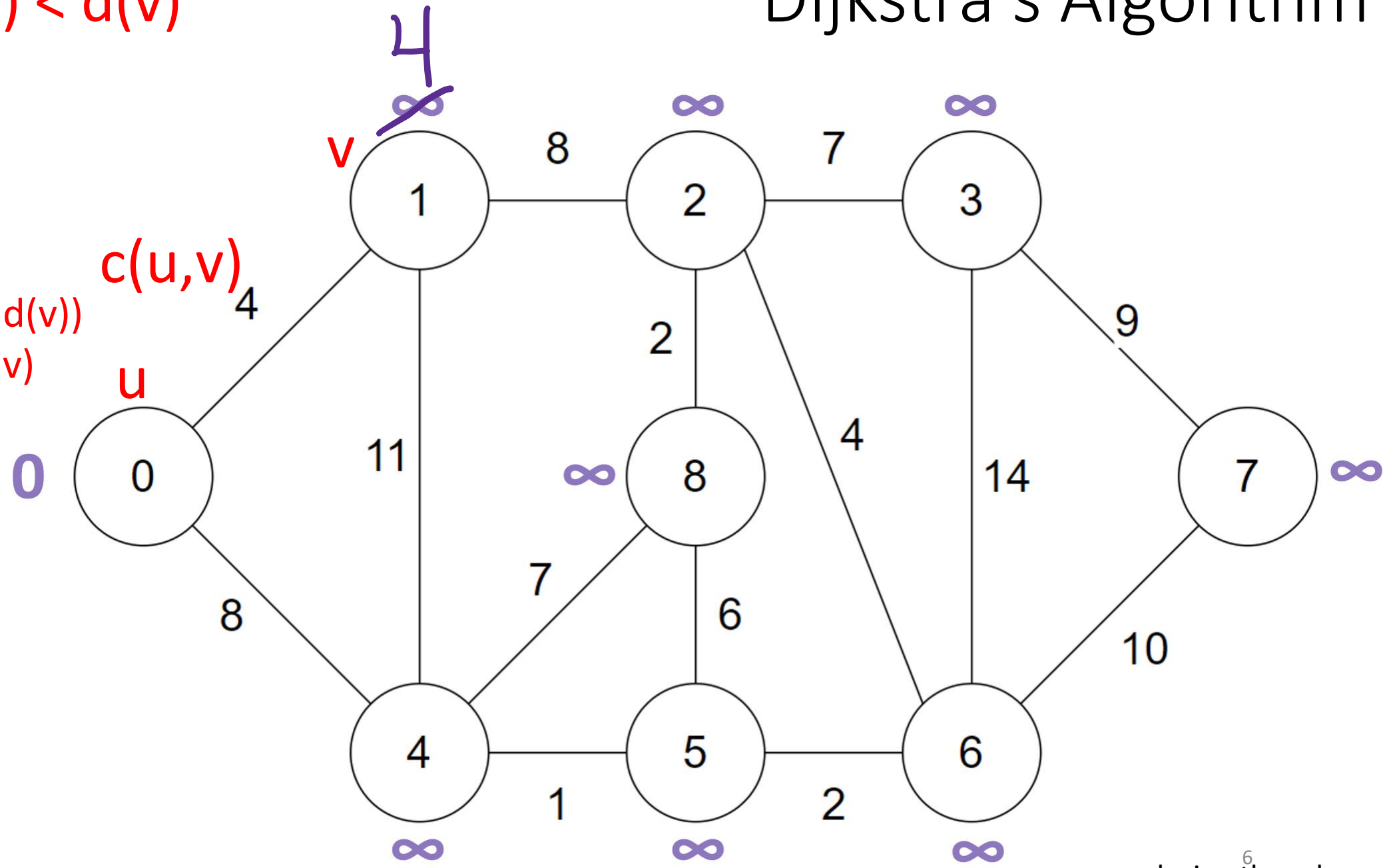
Dijkstra's Algorithm

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

$0 + 4 < \infty$

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

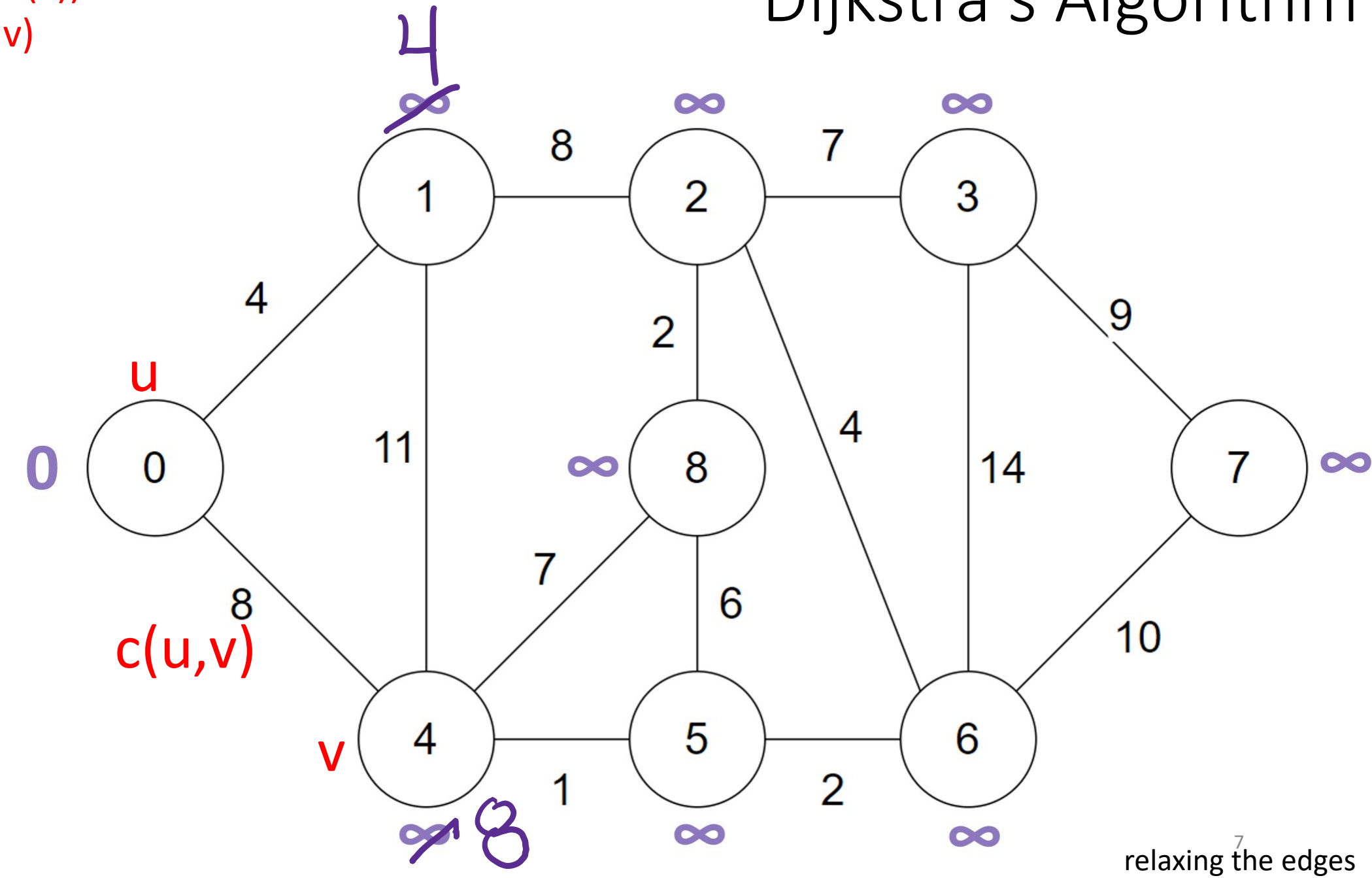


relaxing the edges

Dijkstra's Algorithm

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

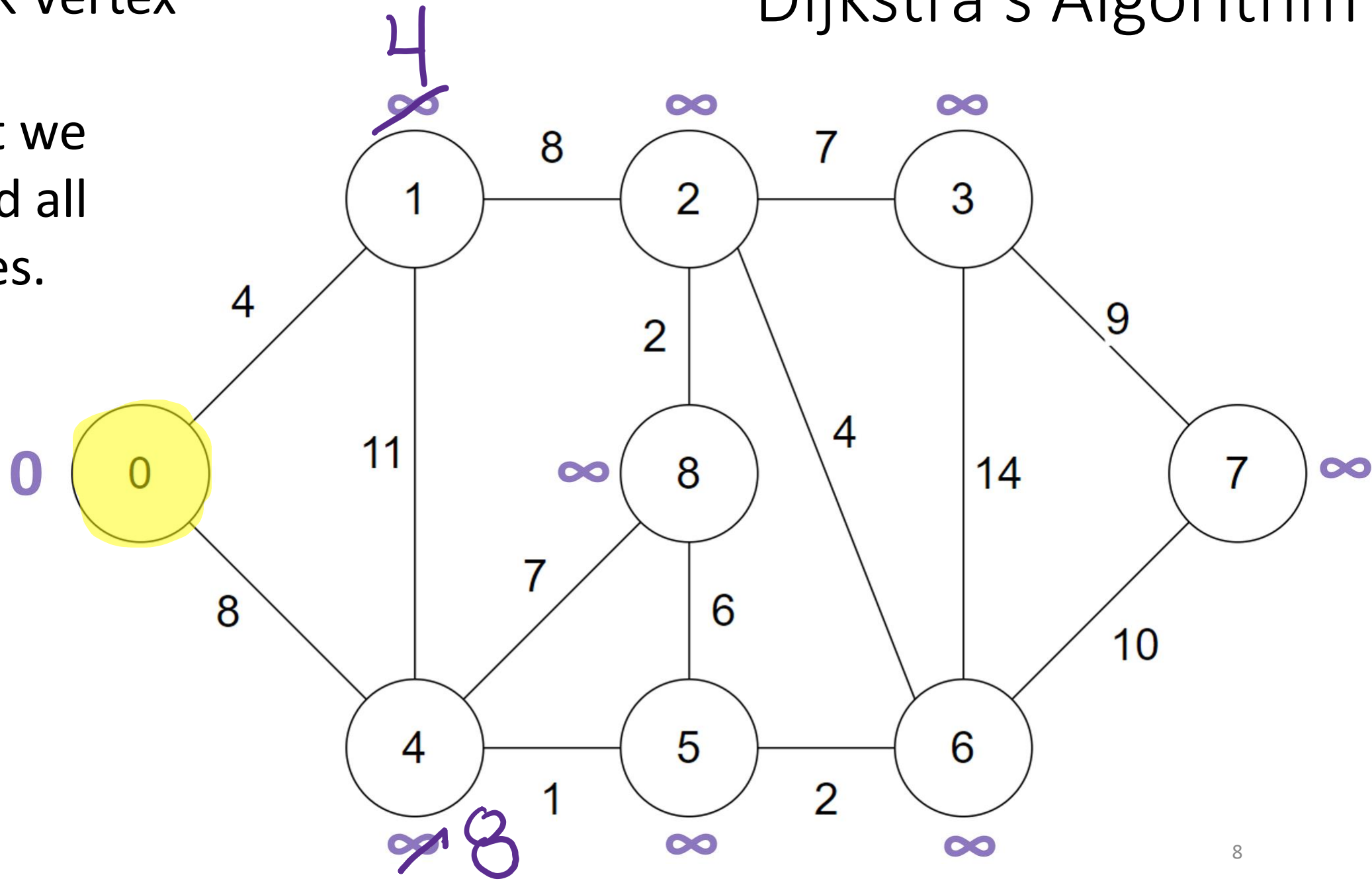
if (0 + 8) < ∞)
 d(v) = 0 + 8
 d(v) = 8



relaxing the edges

Dijkstra's Algorithm

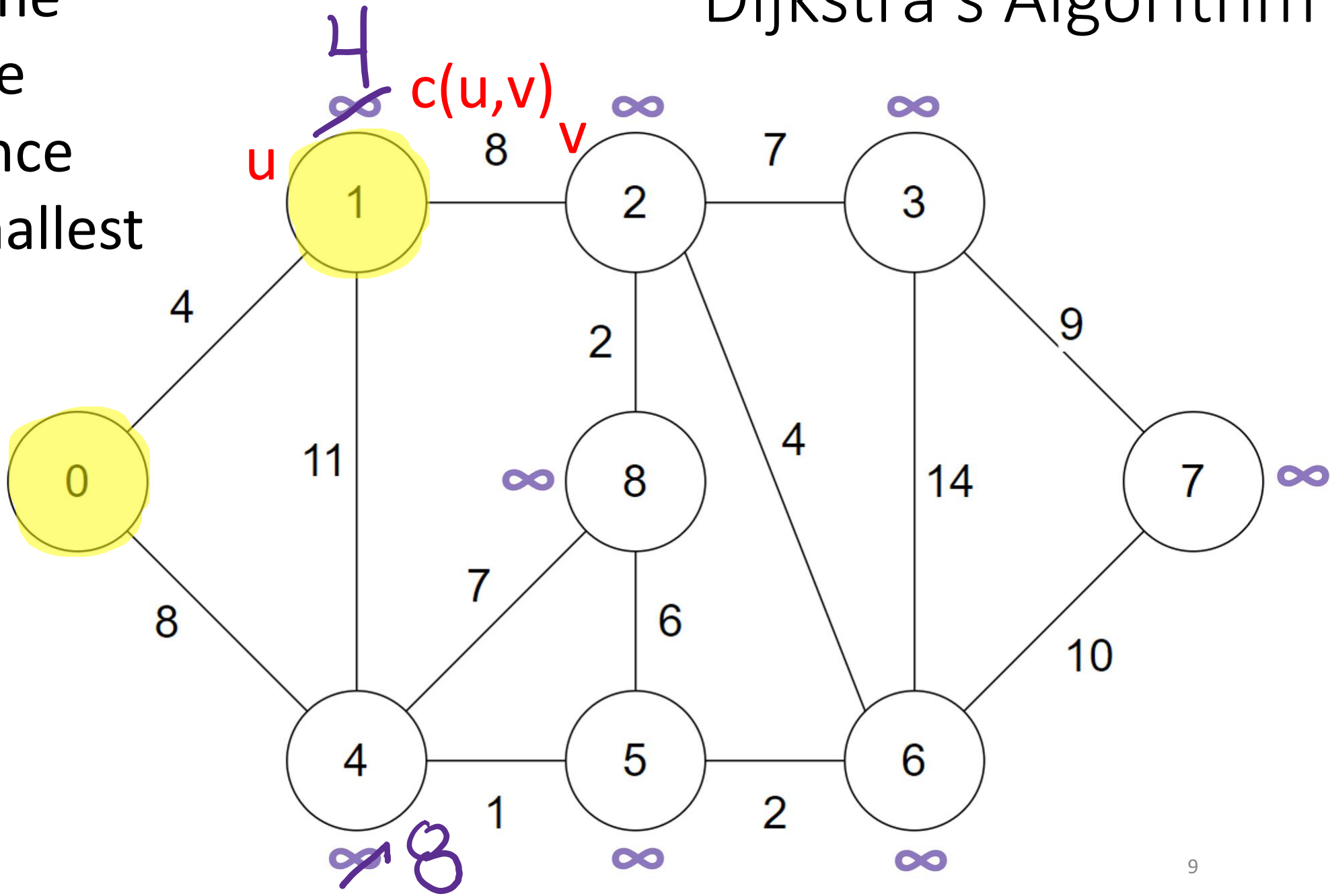
Now we mark Vertex 0 as "visited" meaning that we have explored all incident edges.



Dijkstra's Algorithm

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

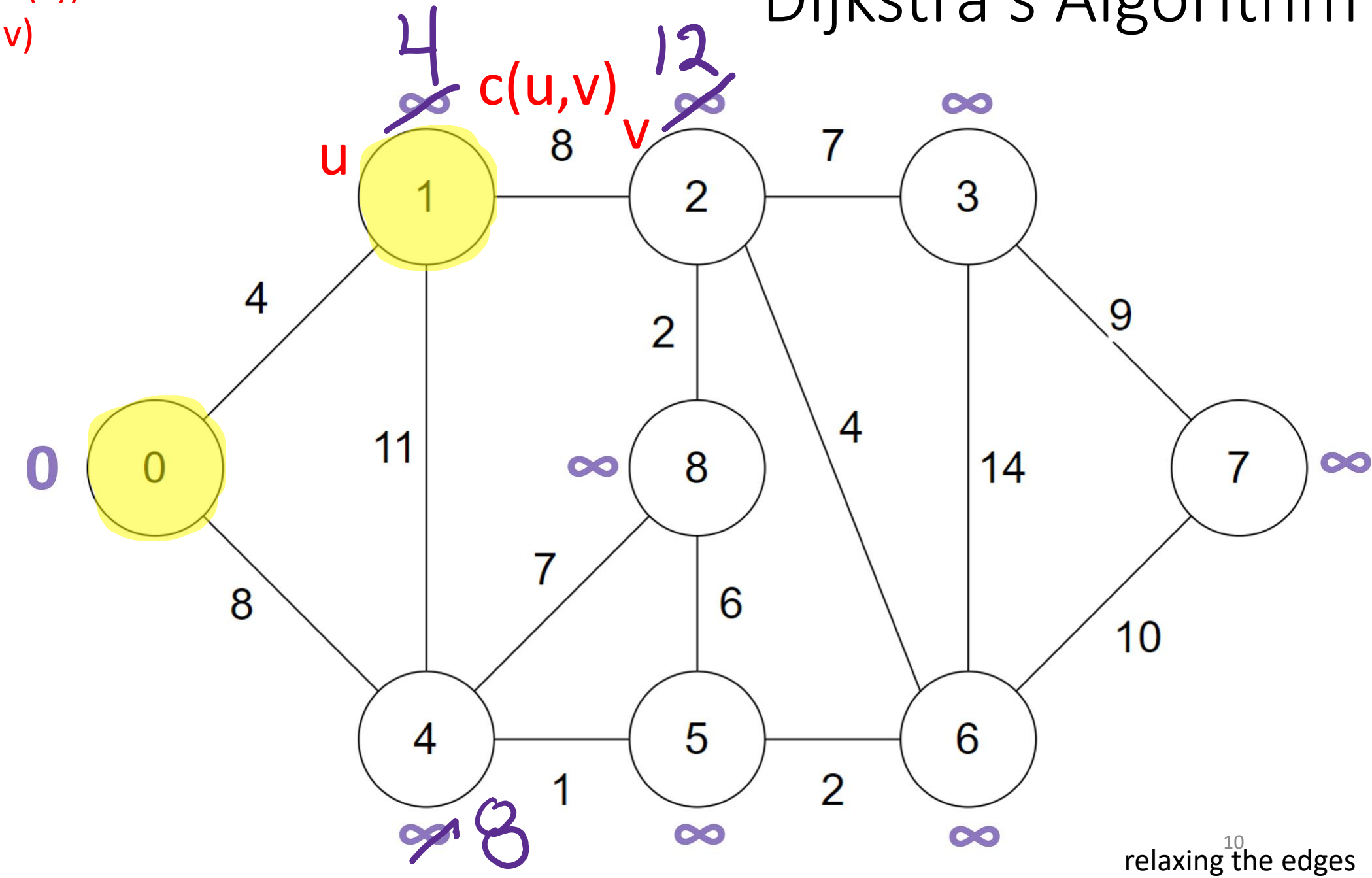
0 – already visited
1 – d(u) of 4
2 – d(u) of ∞
3 – d(u) of ∞
4 – d(u) of 8
5 – d(u) of ∞
6 – d(u) of ∞
7 – d(u) of ∞
8 – d(u) of ∞



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

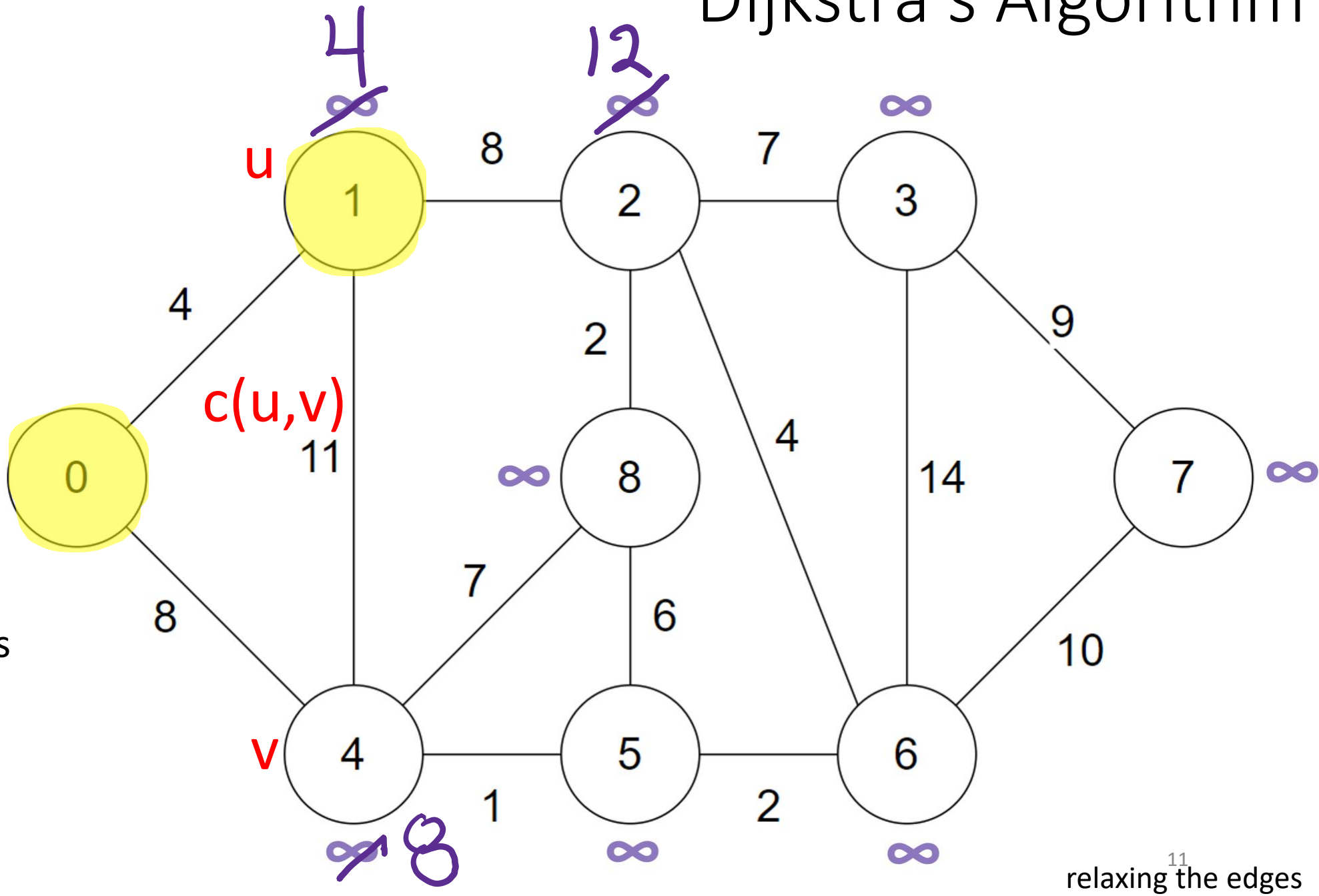


relaxing the edges

Dijkstra's Algorithm

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

if (4 + 11) < 8)



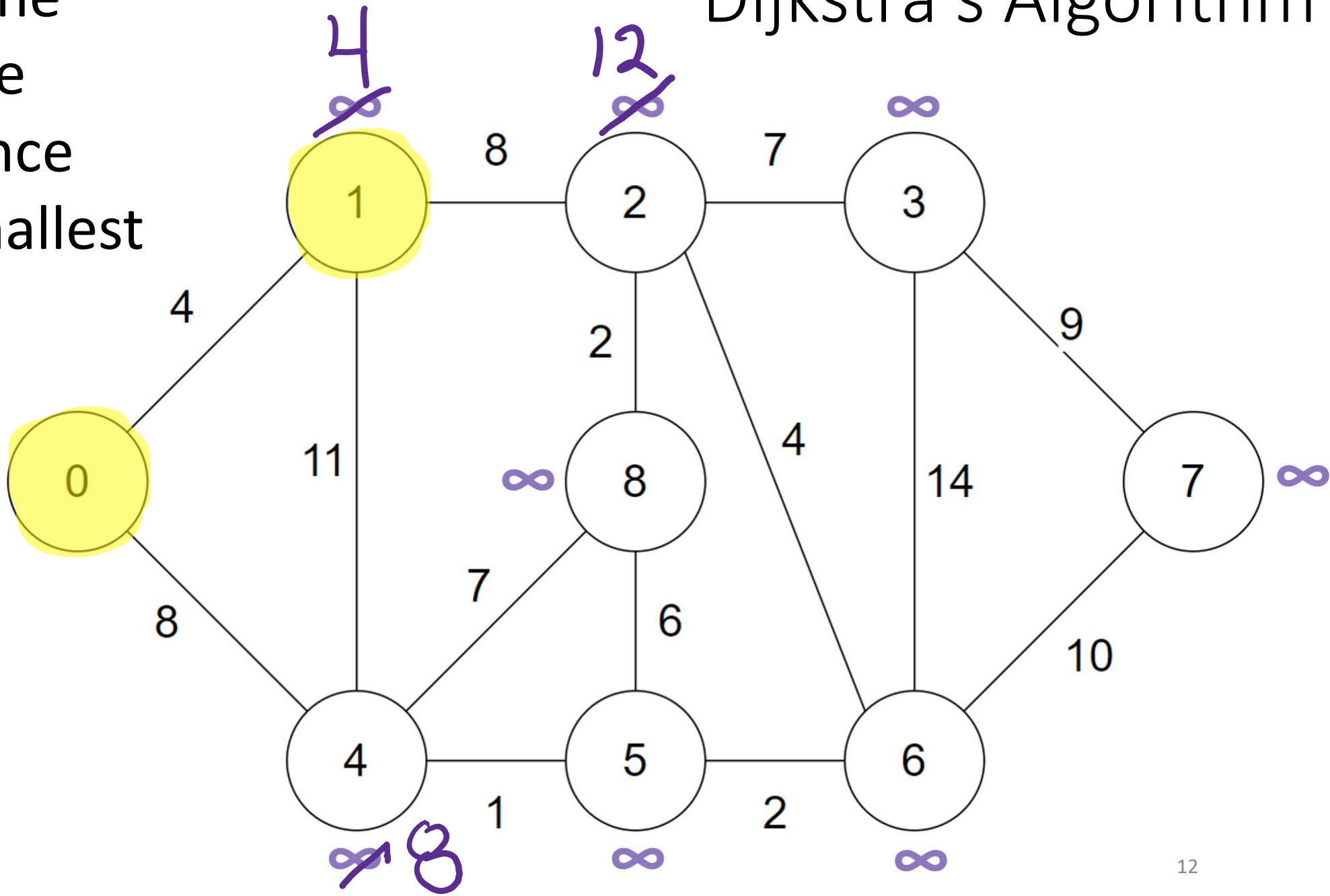
15 is greater than the existing value of Vertex 4 (which is 8). If the existing value is less than our new calculation, then we keep the original value.

relaxing the edges

Dijkstra's Algorithm

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

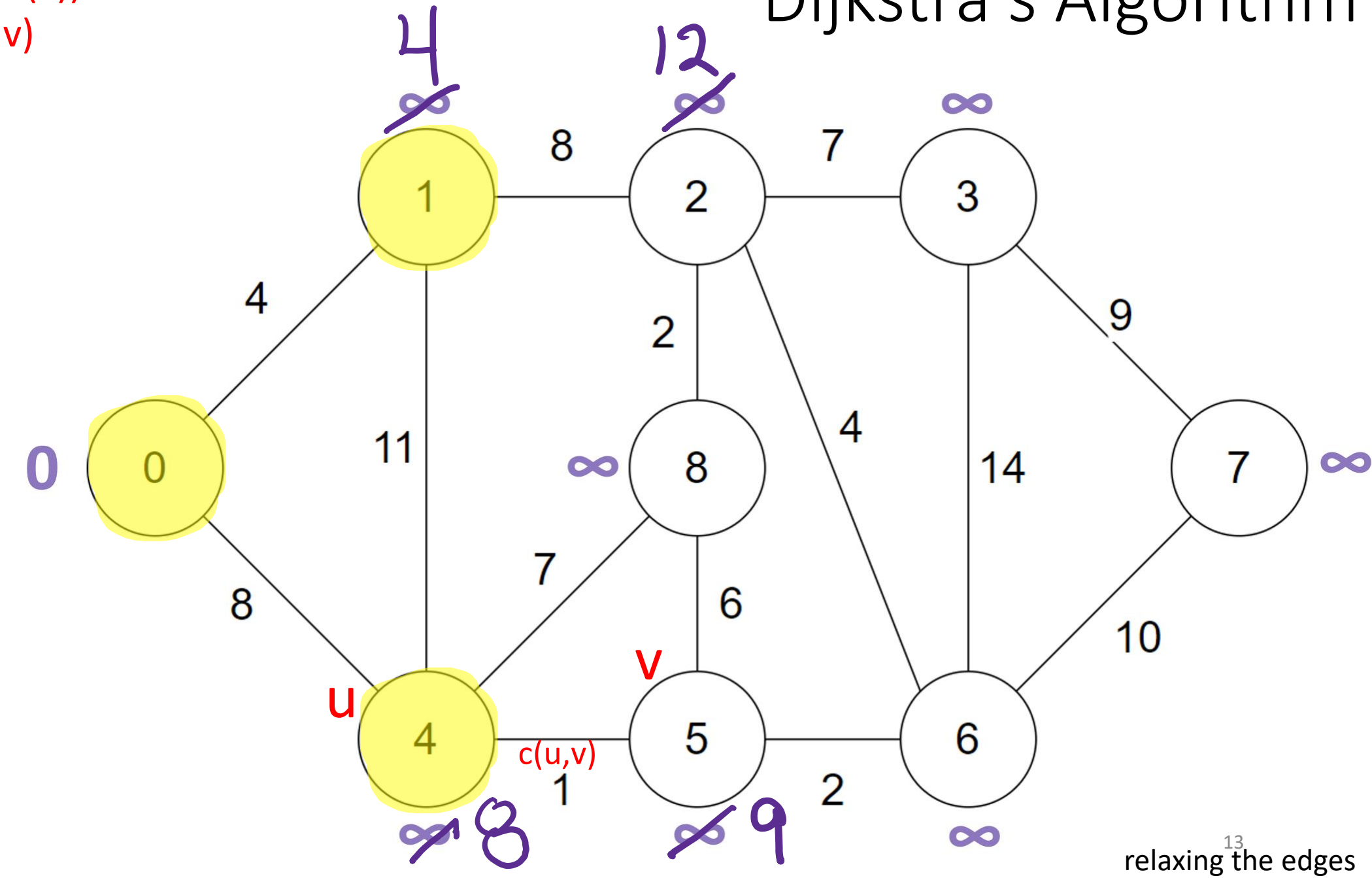
0 – already visited
1 – already visited
2 – d(u) of 12
3 – d(u) of ∞
4 – d(u) of 8
5 – d(u) of ∞
6 – d(u) of ∞
7 – d(u) of ∞
8 – d(u) of ∞



Dijkstra's Algorithm

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

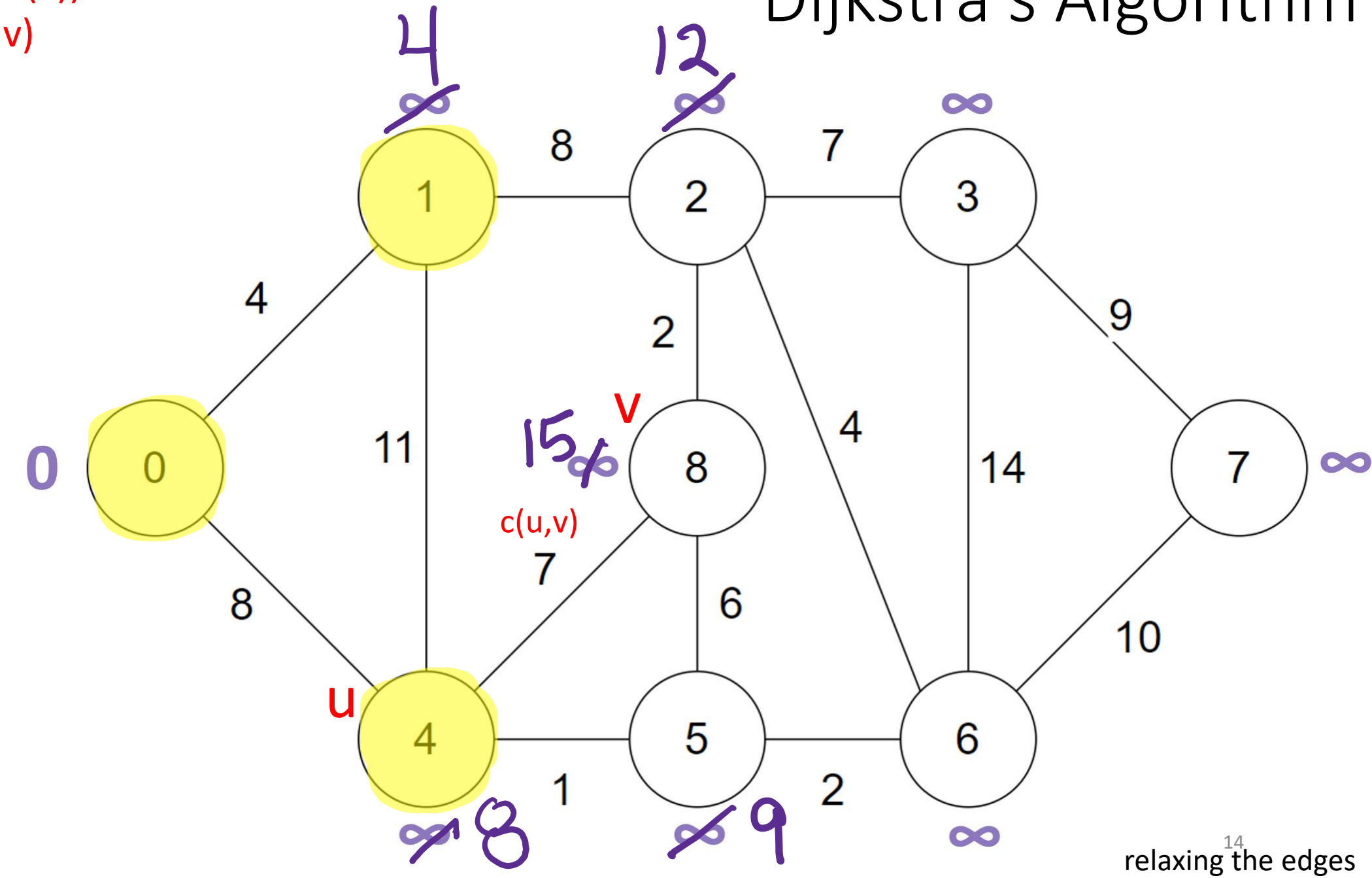


relaxing the edges

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

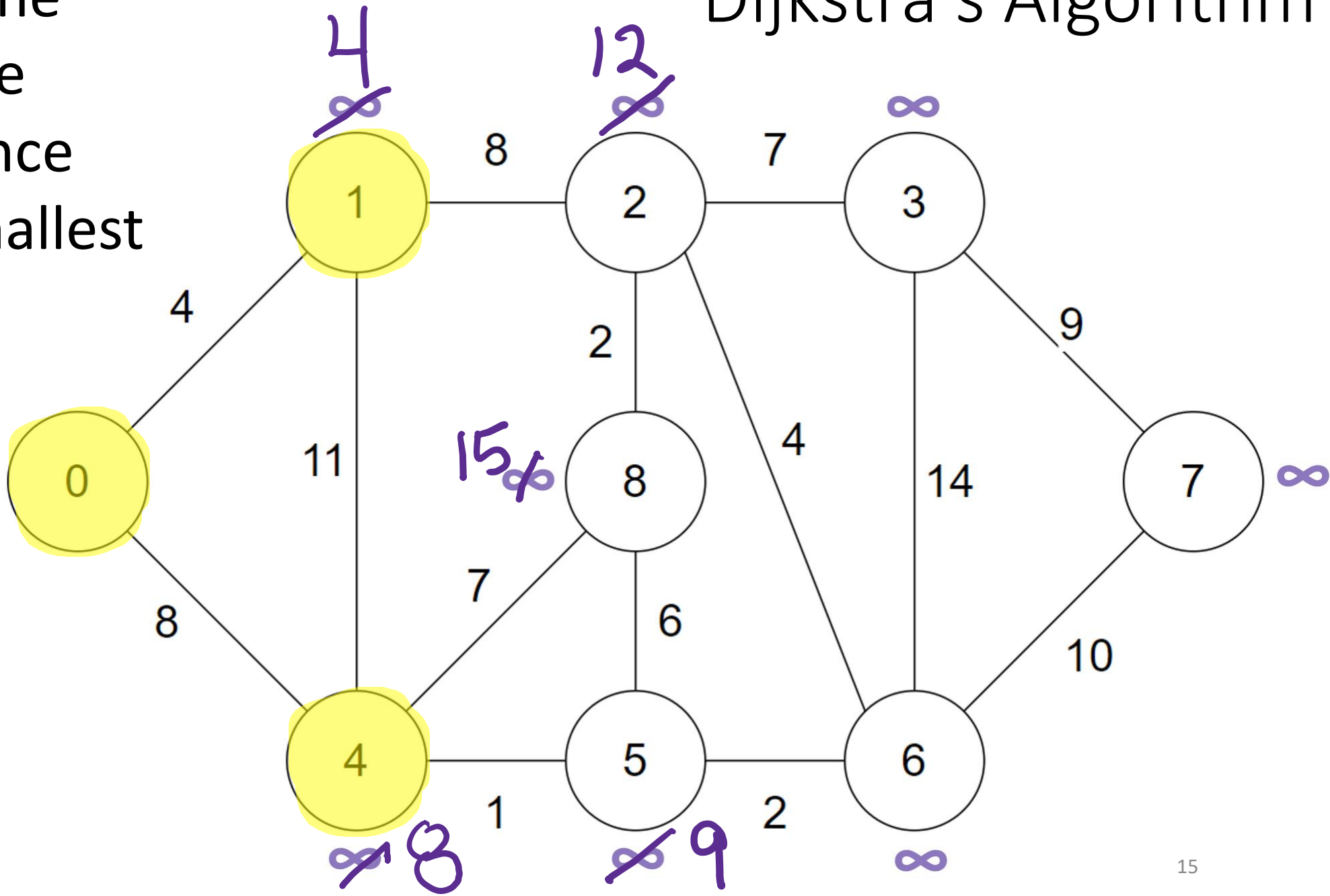


relaxing the edges

Dijkstra's Algorithm

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

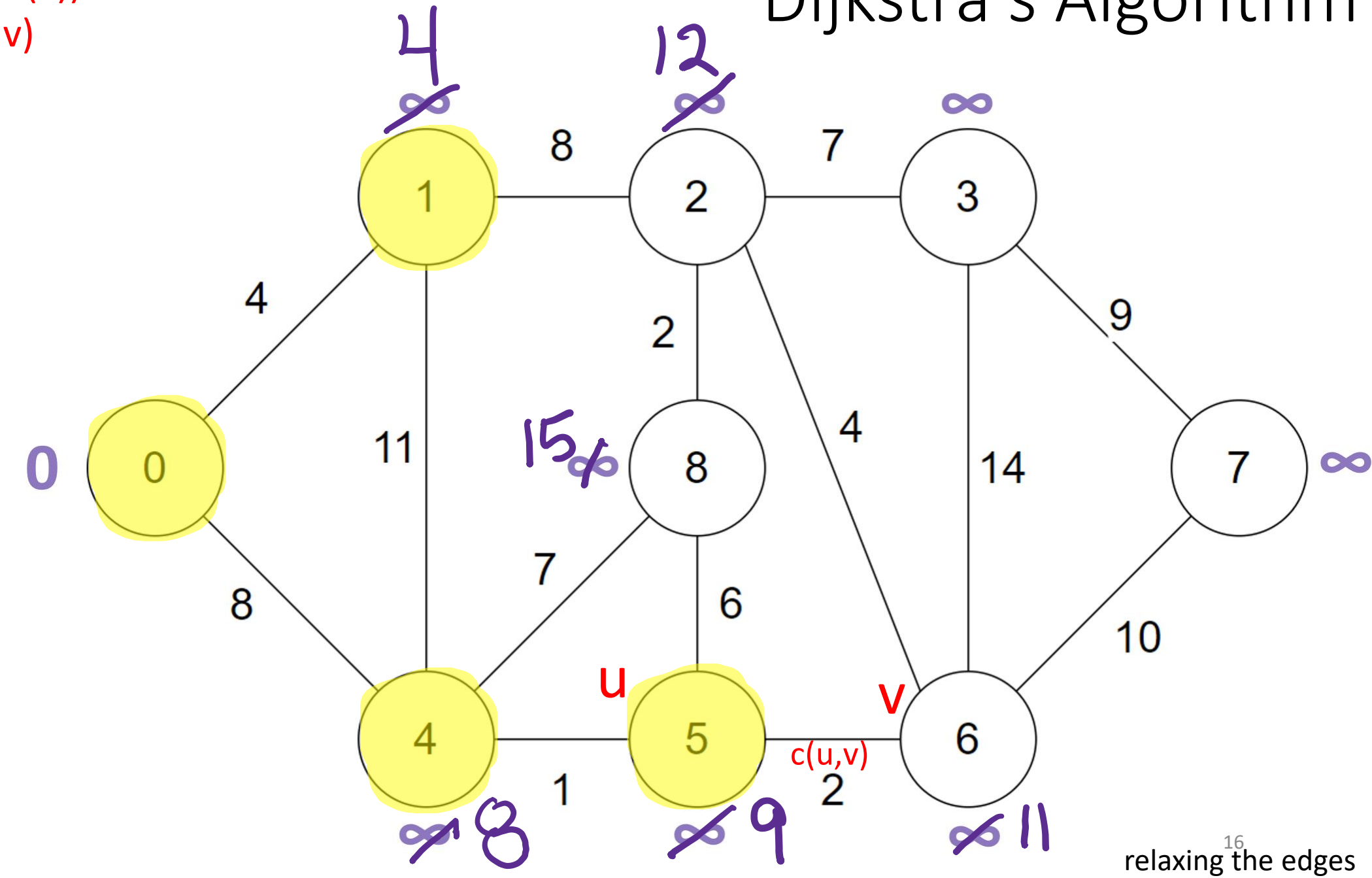
0 – already visited
1 – already visited
2 – d(u) of 12
3 – d(u) of ∞
4 – already visited
5 – d(u) of 9
6 – d(u) of ∞
7 – d(u) of ∞
8 – d(u) of 15



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



relaxing the edges

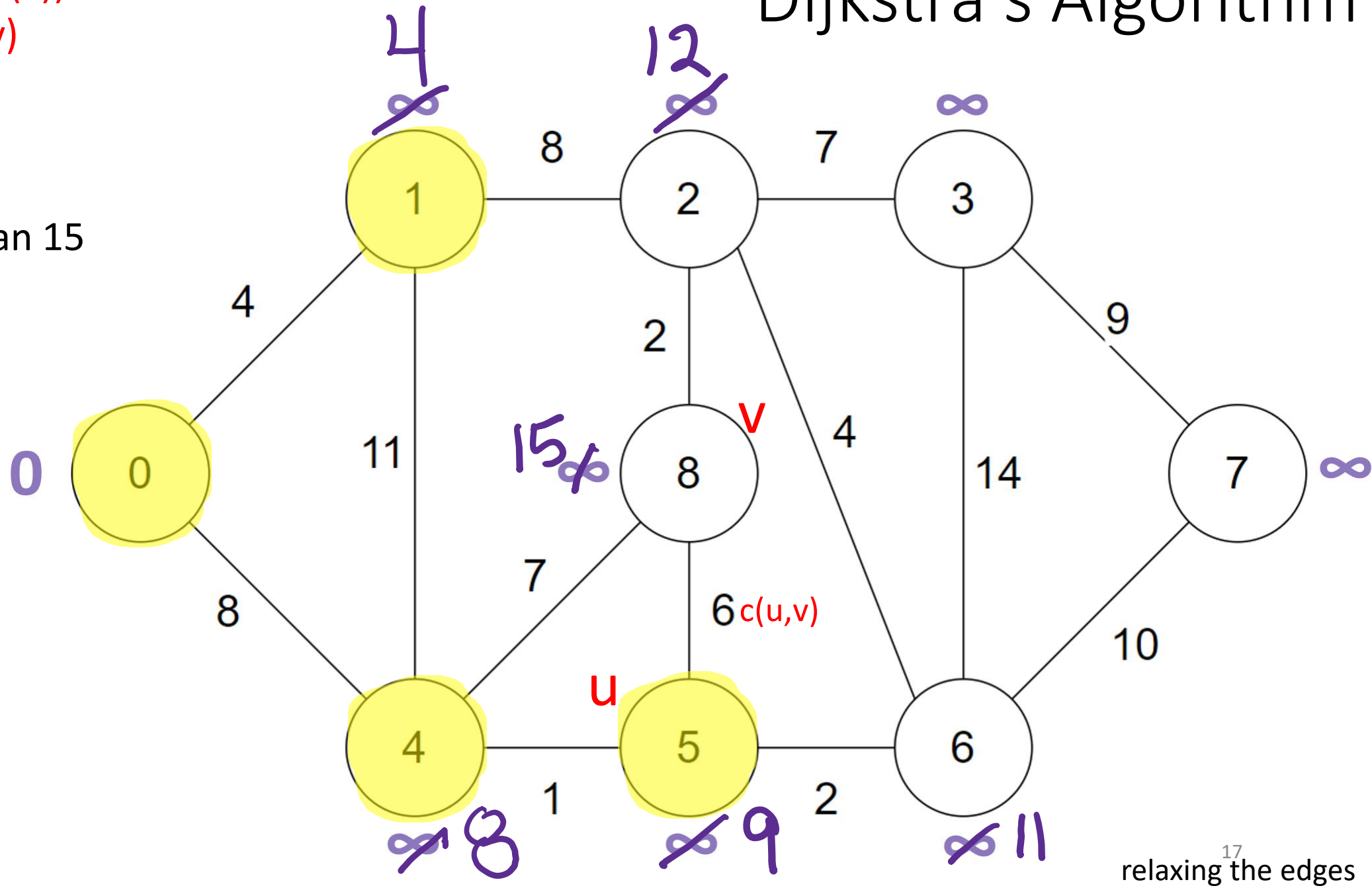
Dijkstra's Algorithm

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

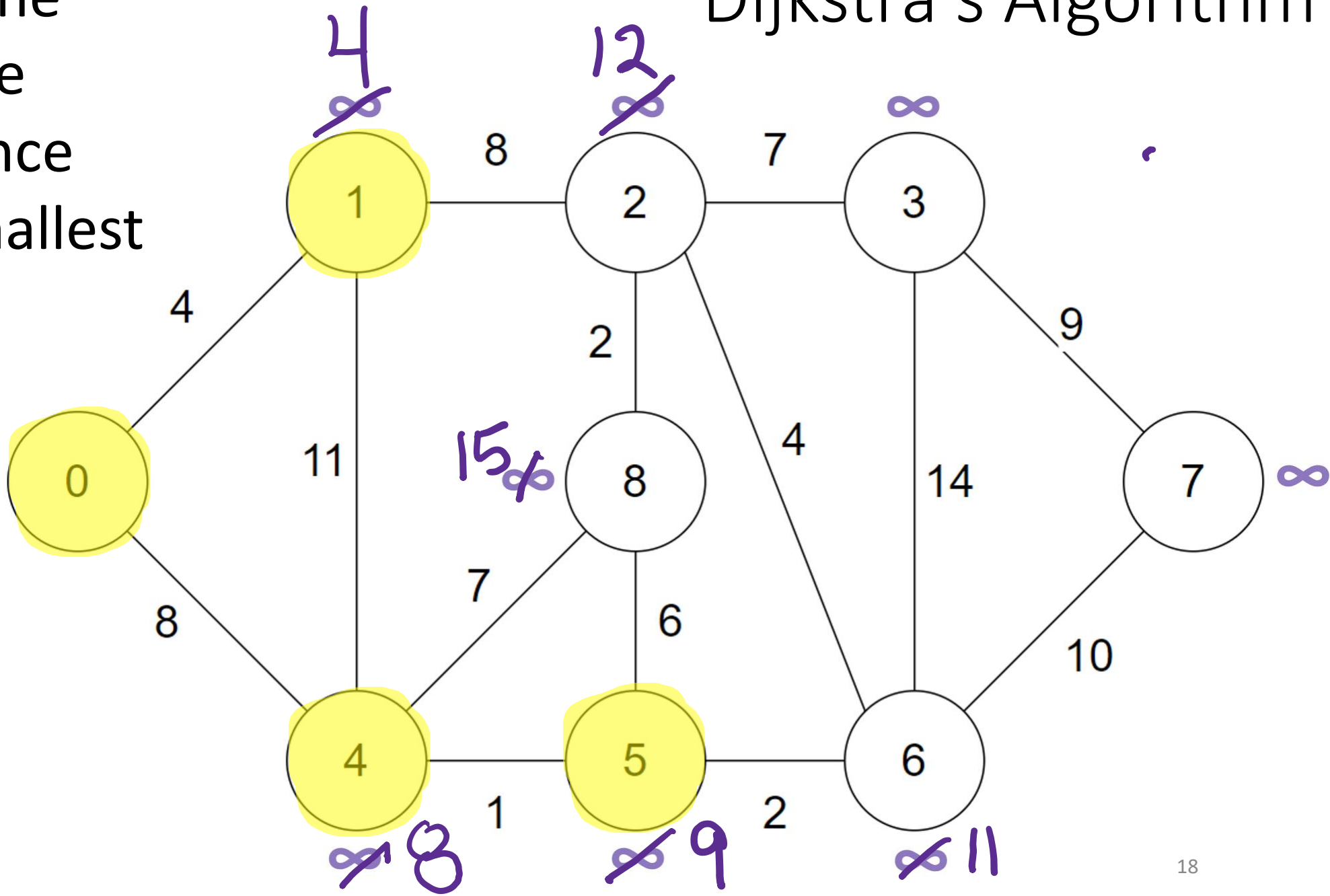


relaxing the edges

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



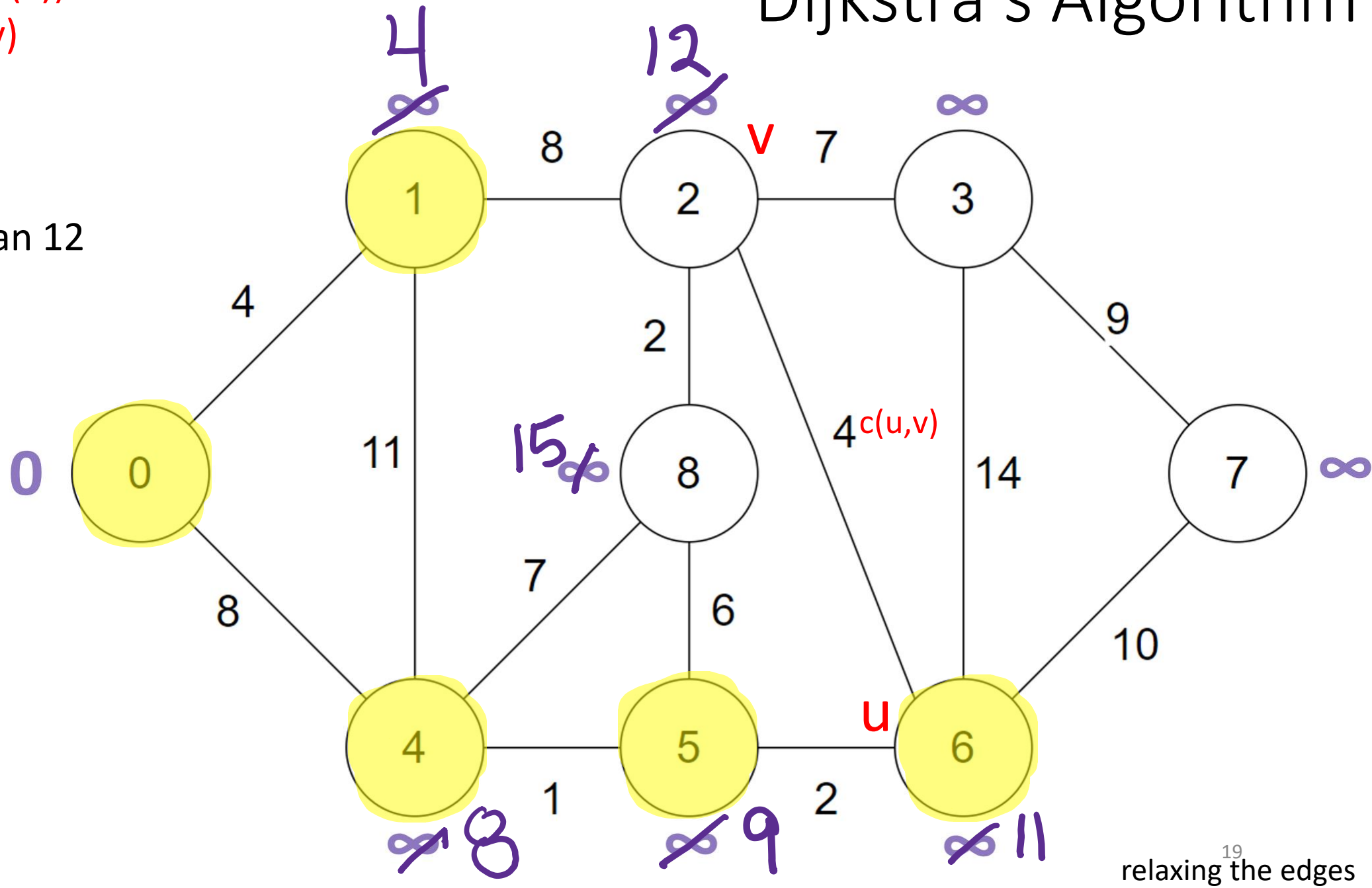
Dijkstra's Algorithm

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

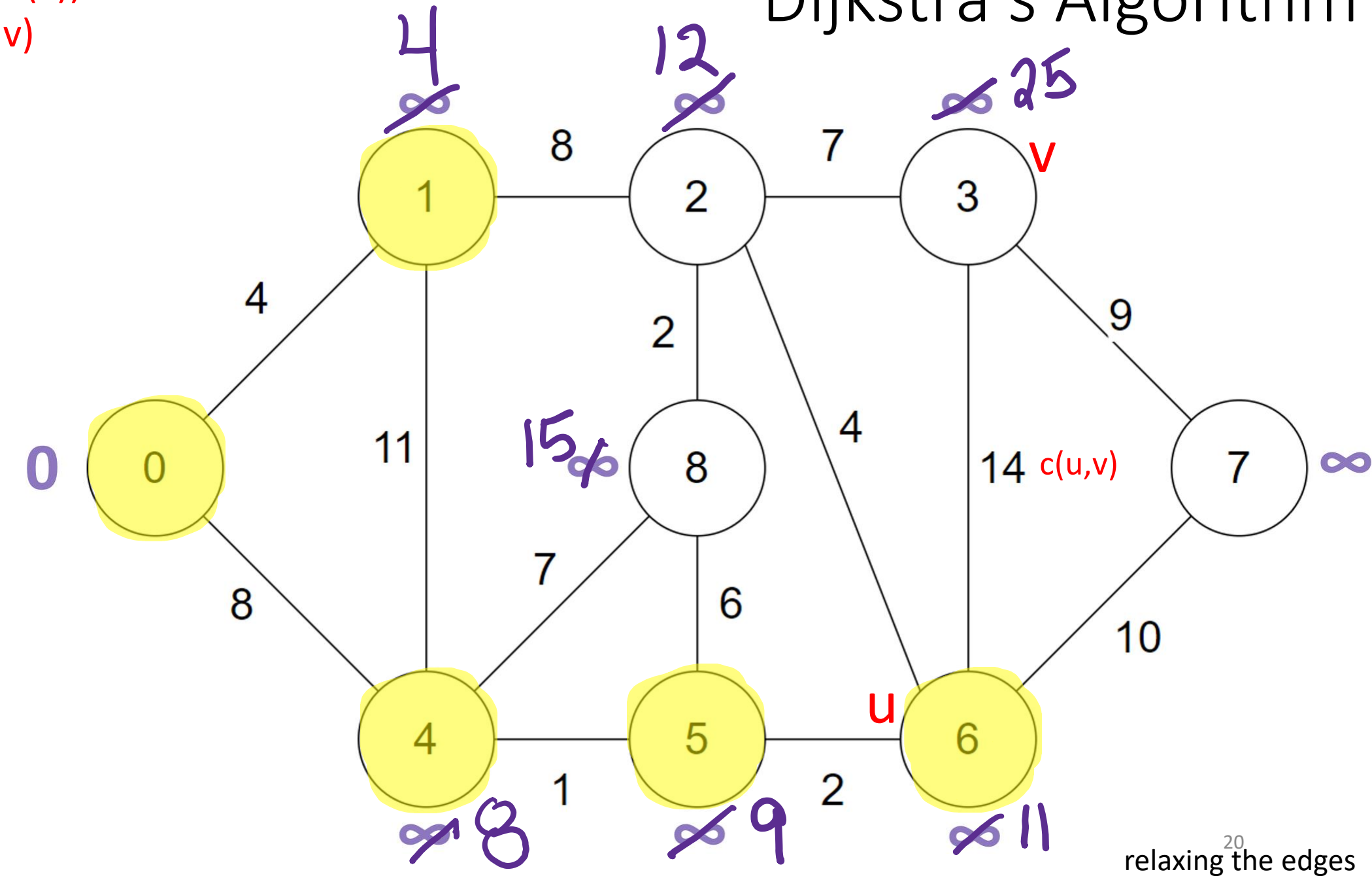


relaxing the edges

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

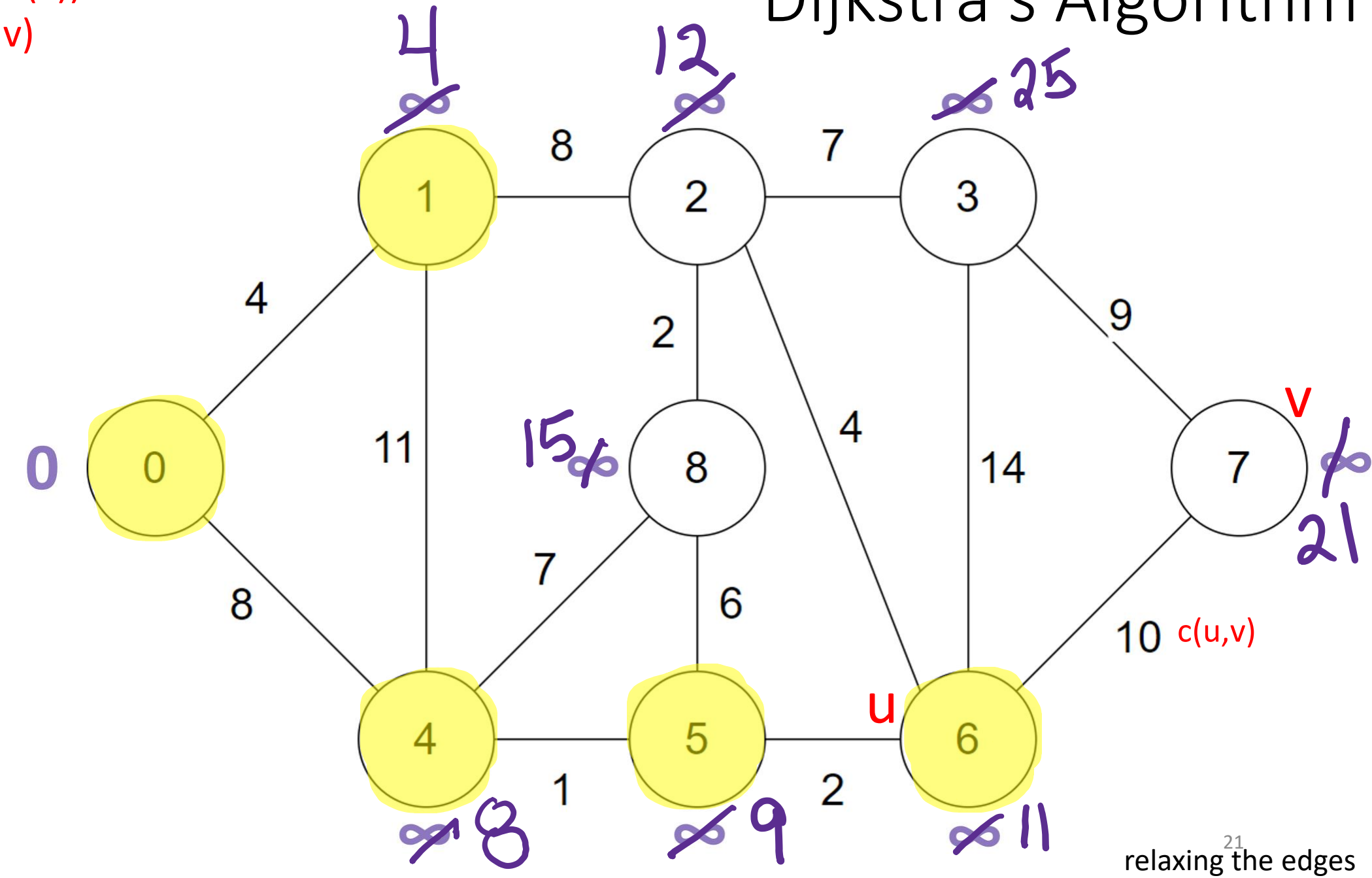


relaxing the edges

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

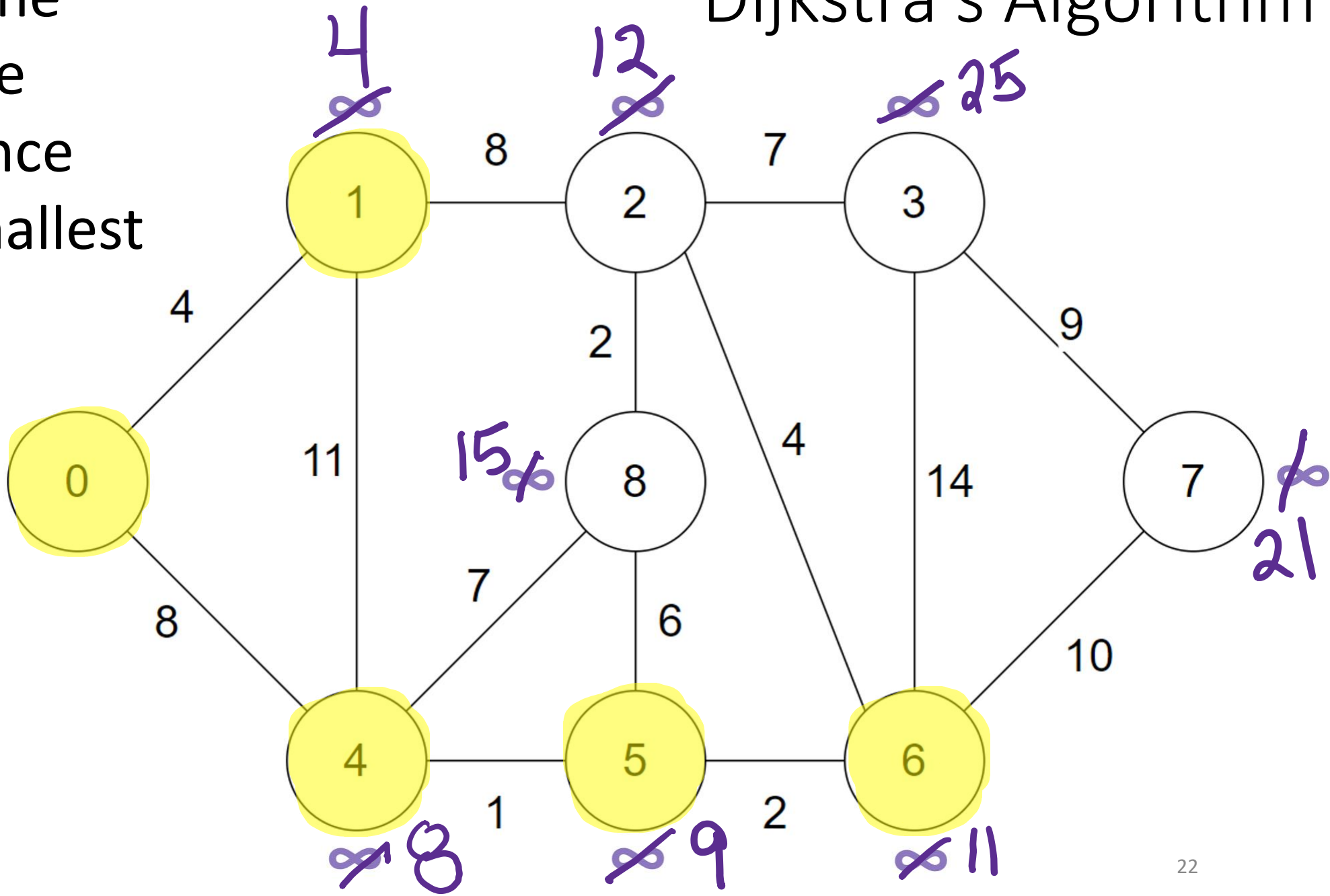


relaxing the edges

Dijkstra's Algorithm

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

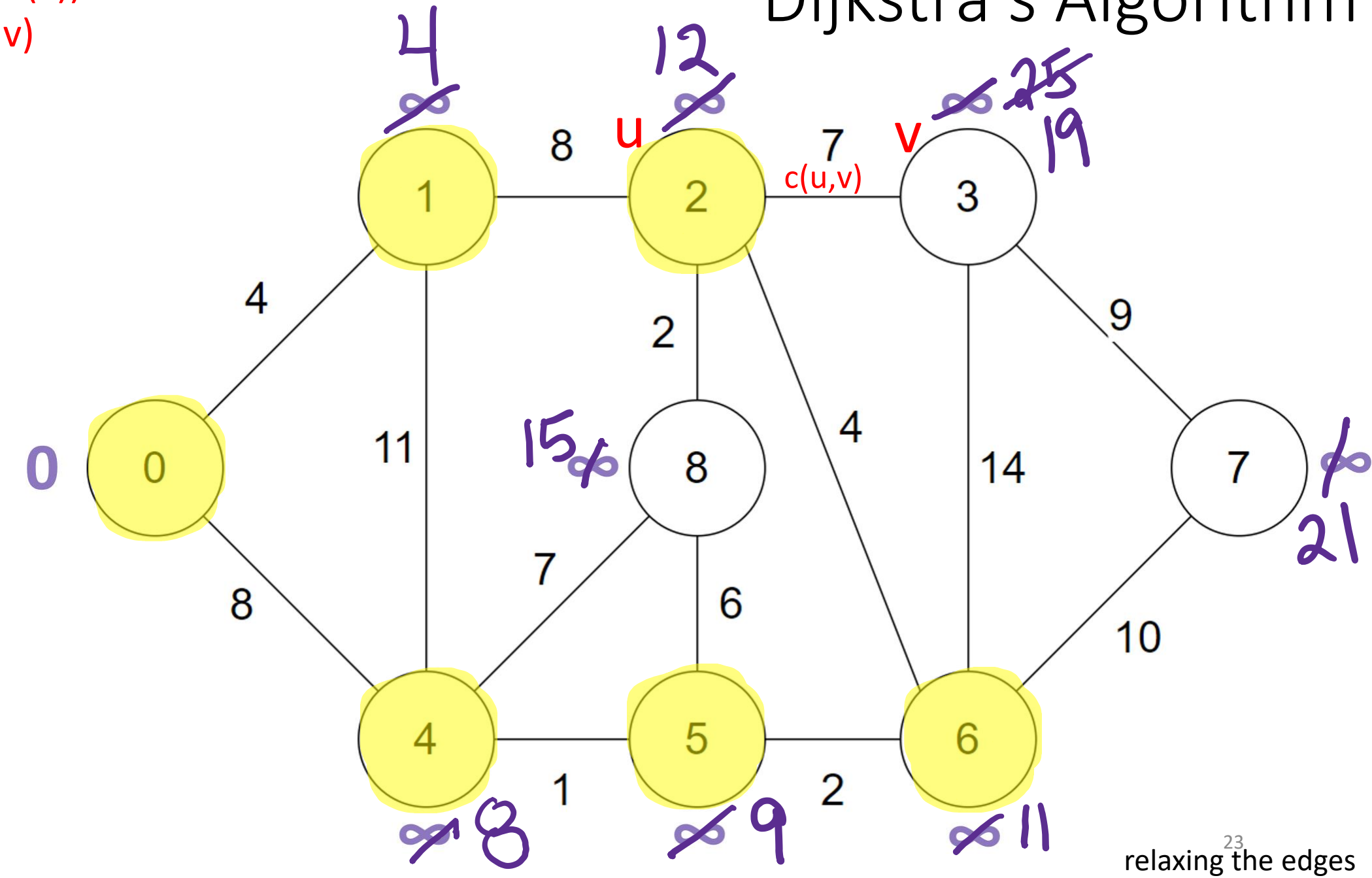
0 – already visited
1 – already visited
2 – d(u) of 12
3 – d(u) of 25
4 – already visited
5 – already visited
6 – already visited
7 – d(u) of 21
8 – d(u) of 15



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

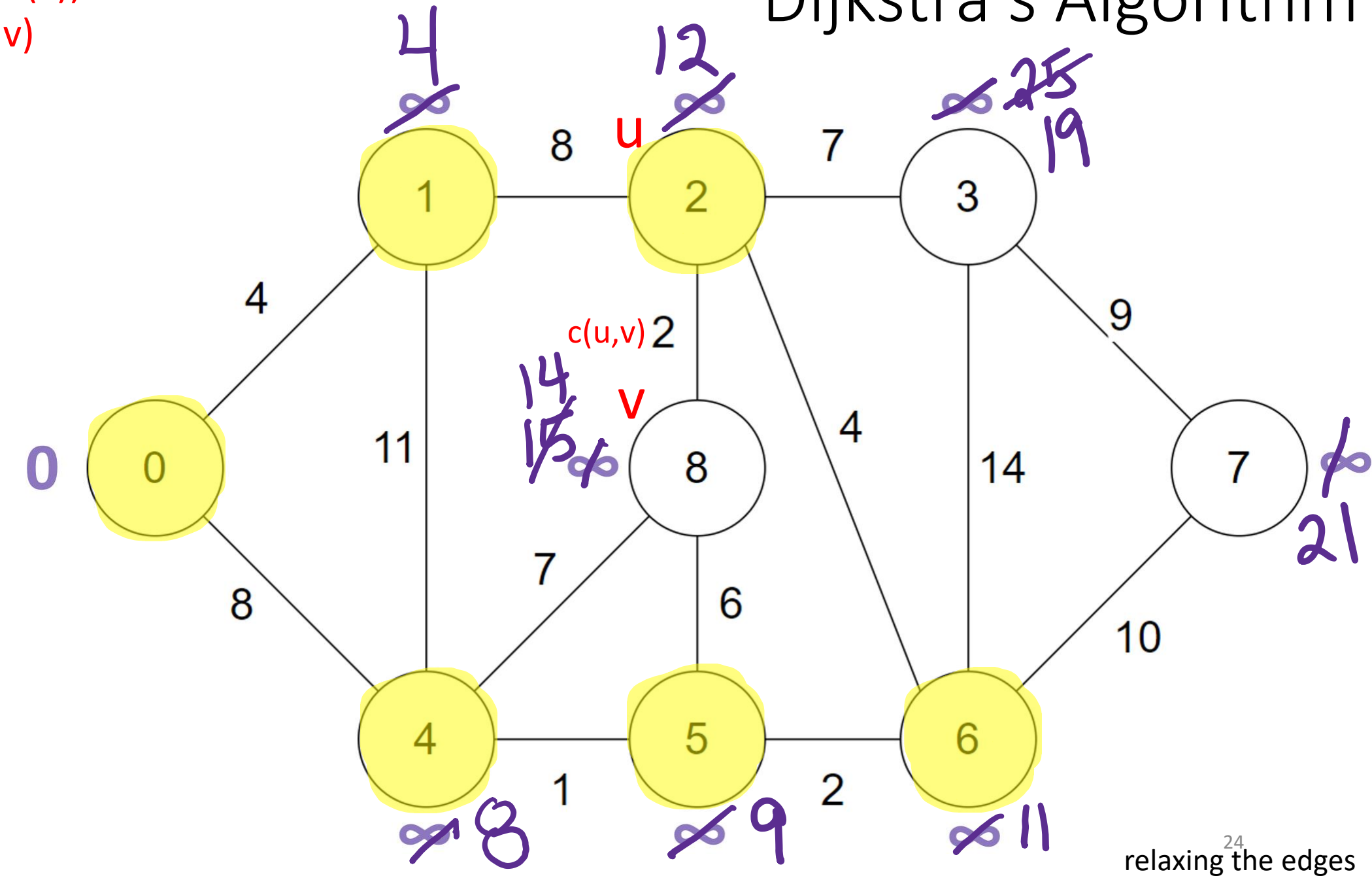


relaxing the edges

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

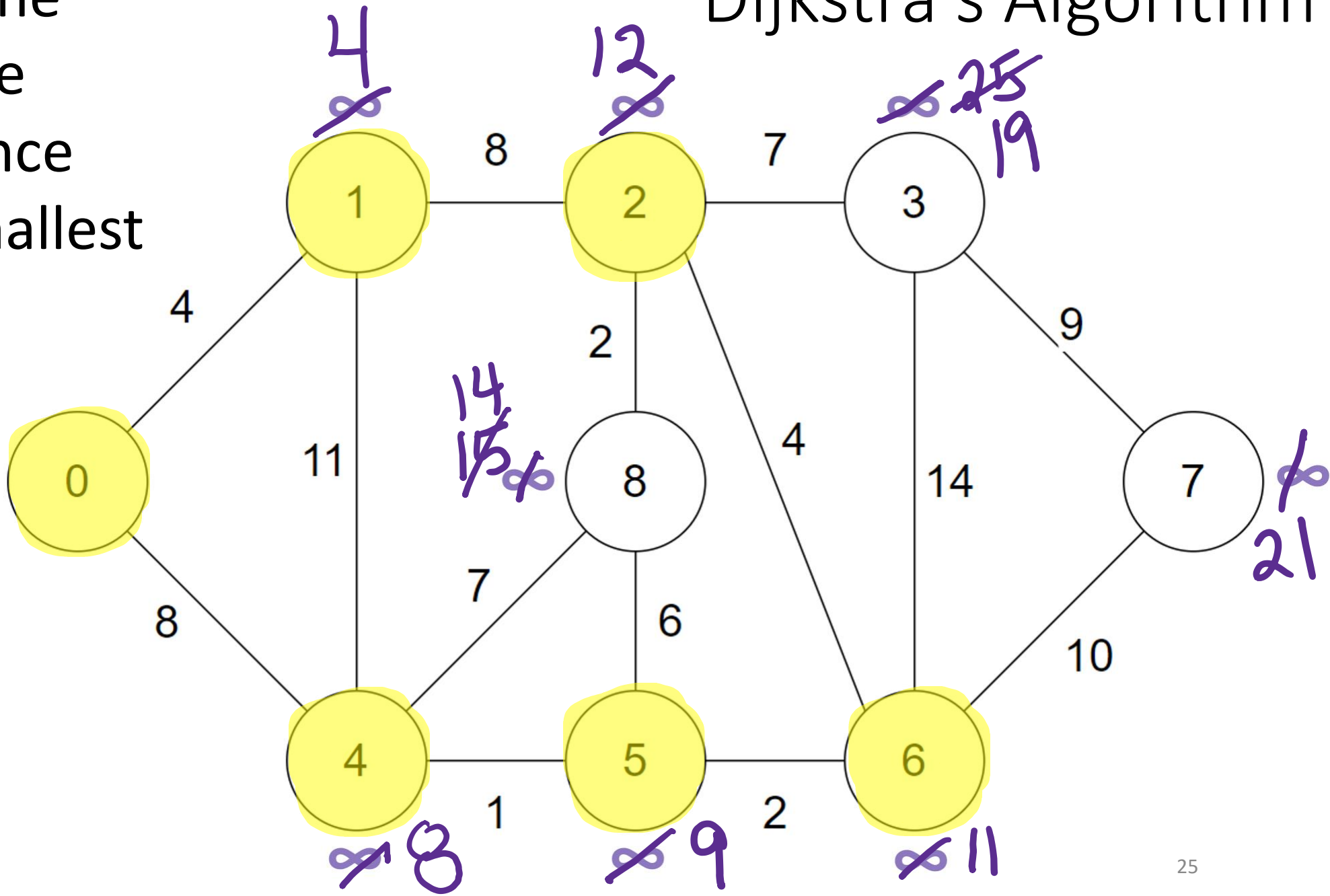


relaxing the edges

Dijkstra's Algorithm

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

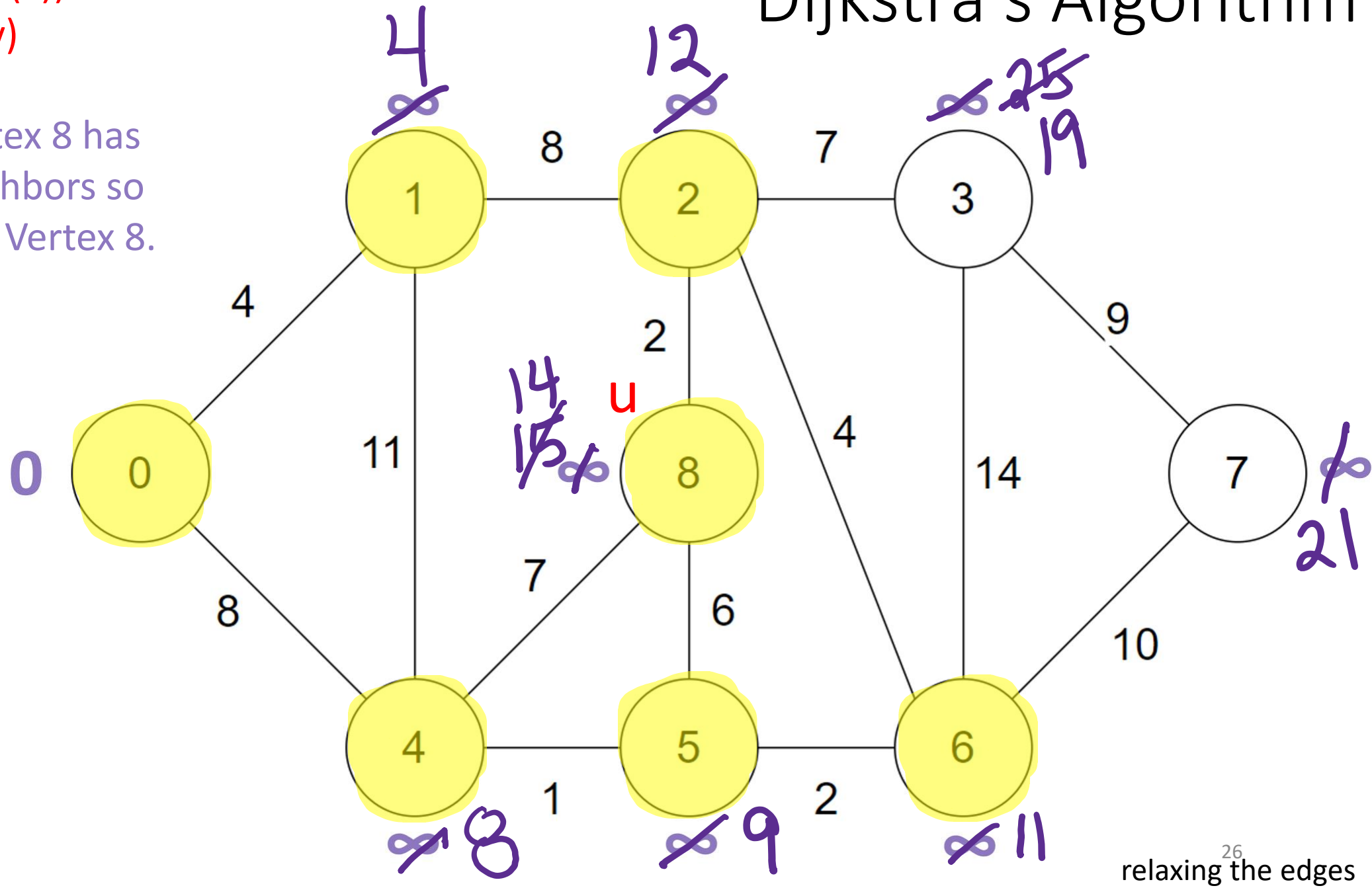
- 0 – already visited
- 1 – already visited
- 2 – already visited
- 3 – d(u) of 19
- 4 – already visited
- 5 – already visited
- 6 – already visited
- 7 – d(u) of 21
- 8 – d(u) of 14



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.

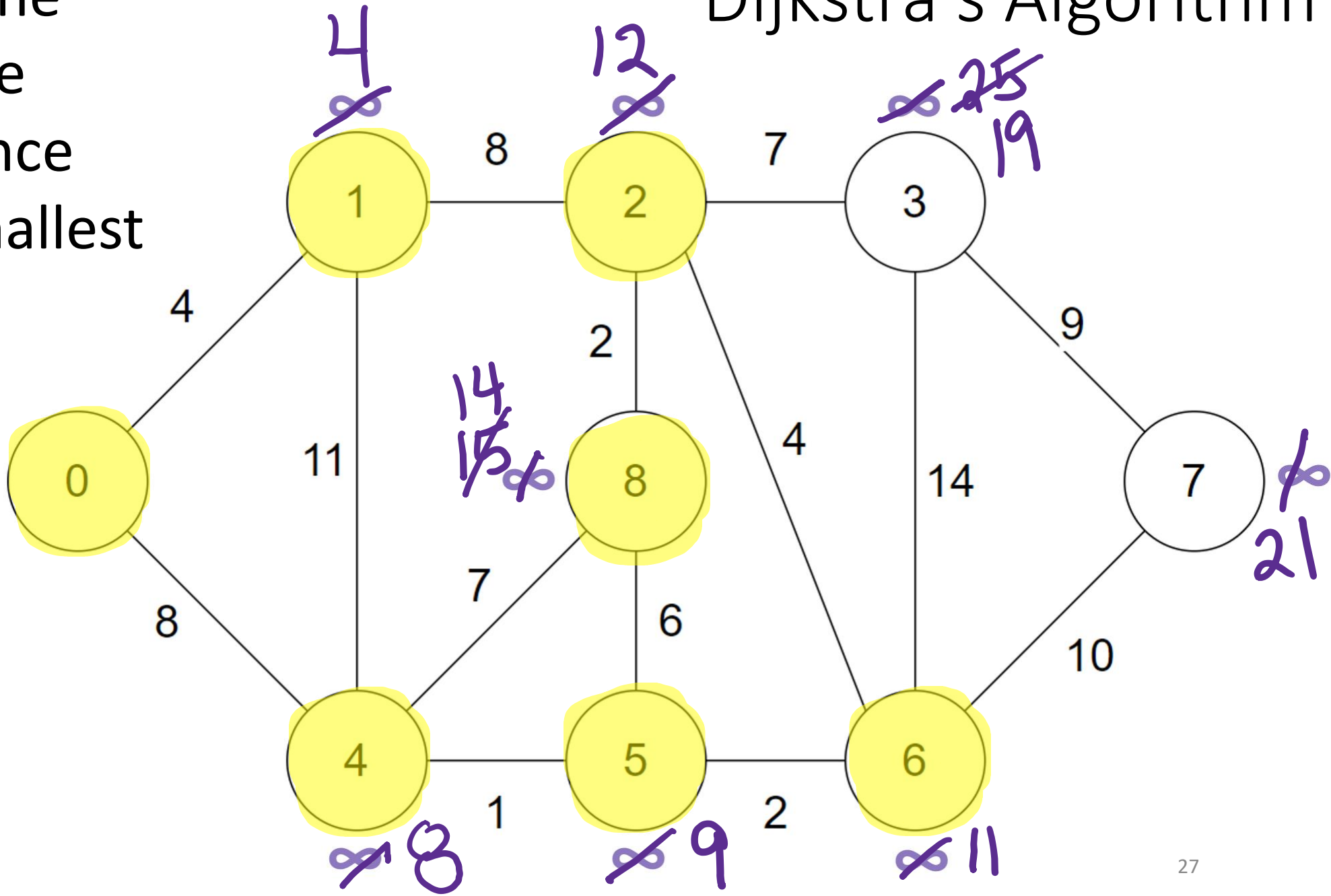


relaxing the edges

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



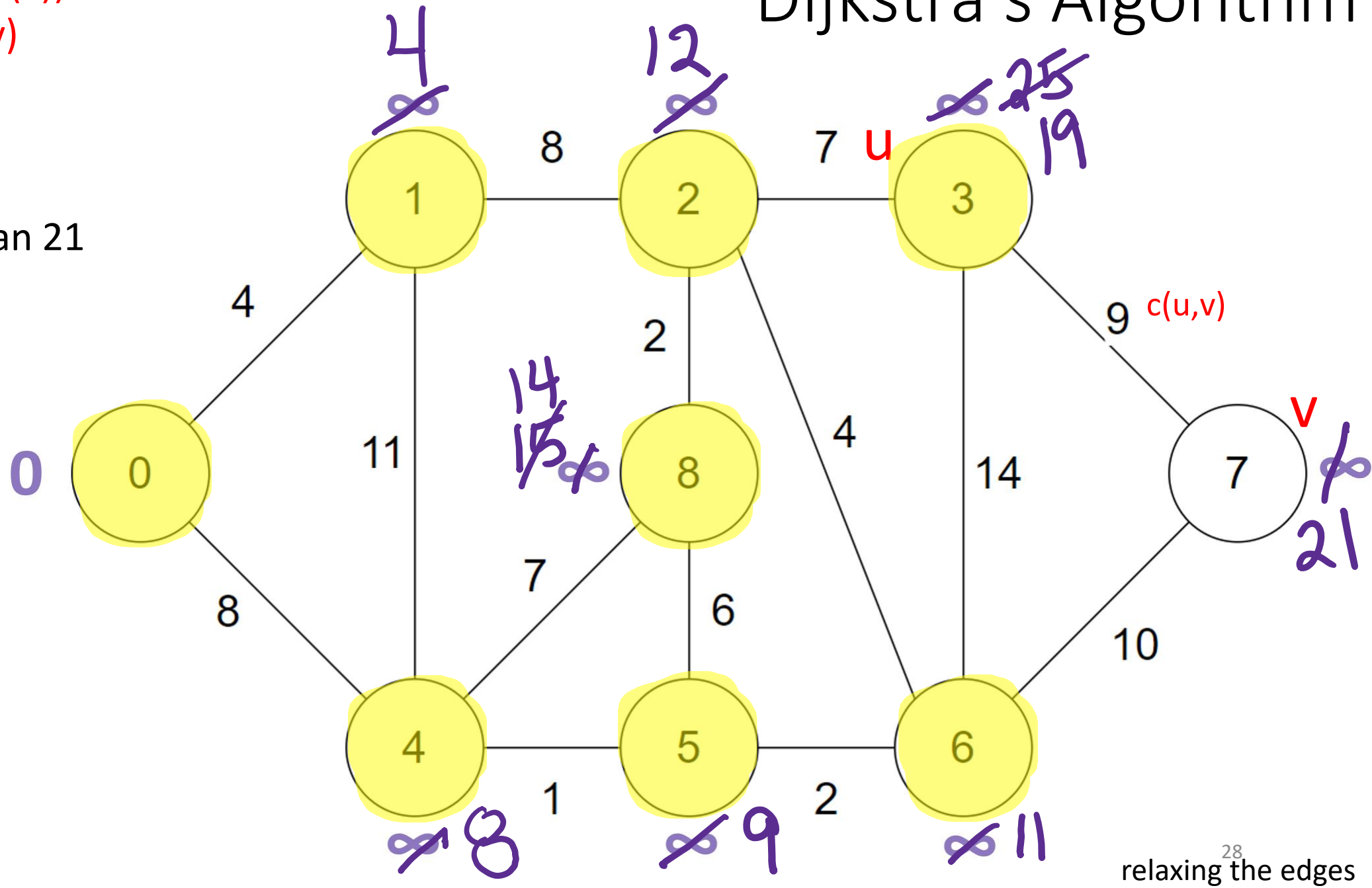
Dijkstra's Algorithm

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

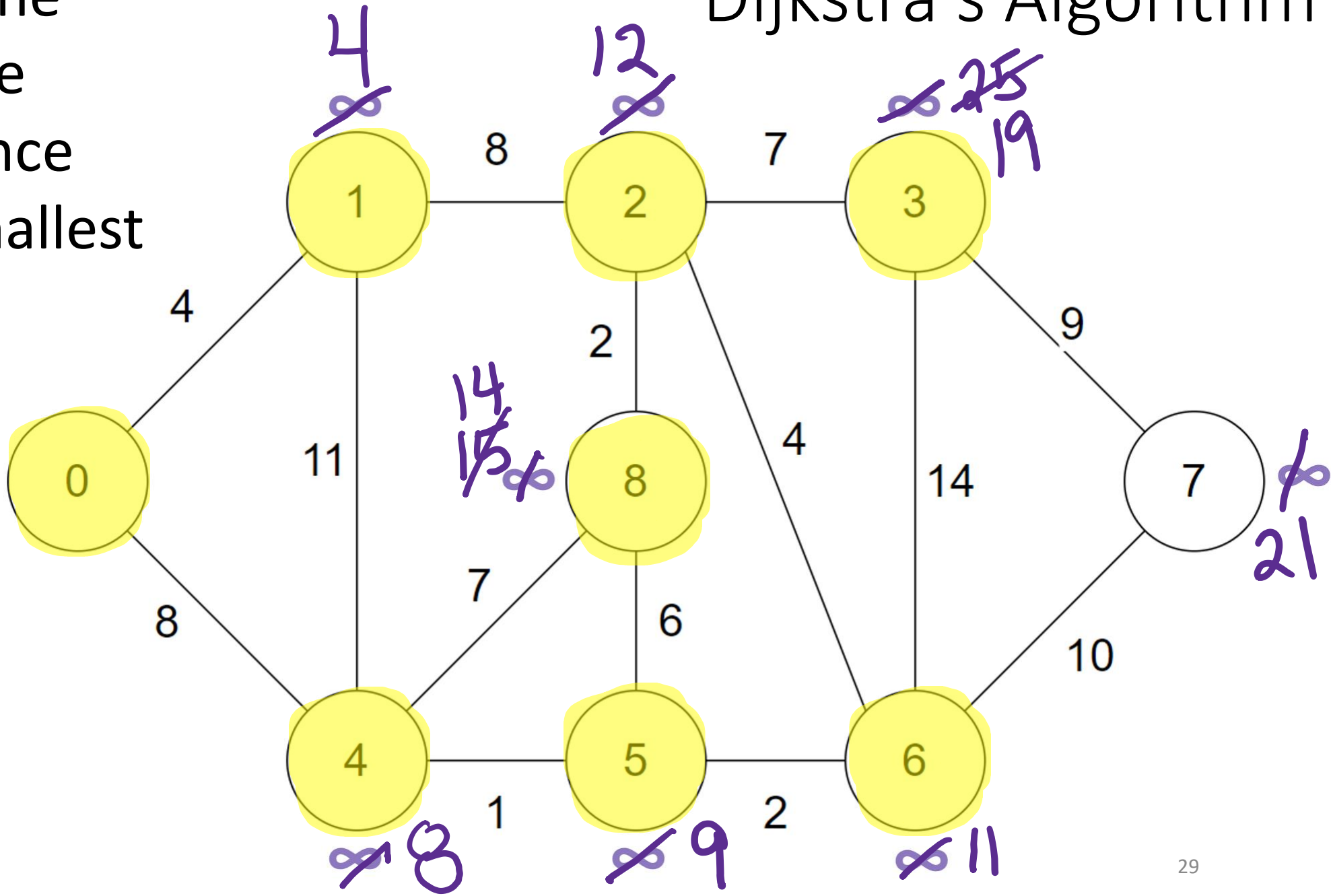


relaxing the edges

Dijkstra's Algorithm

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

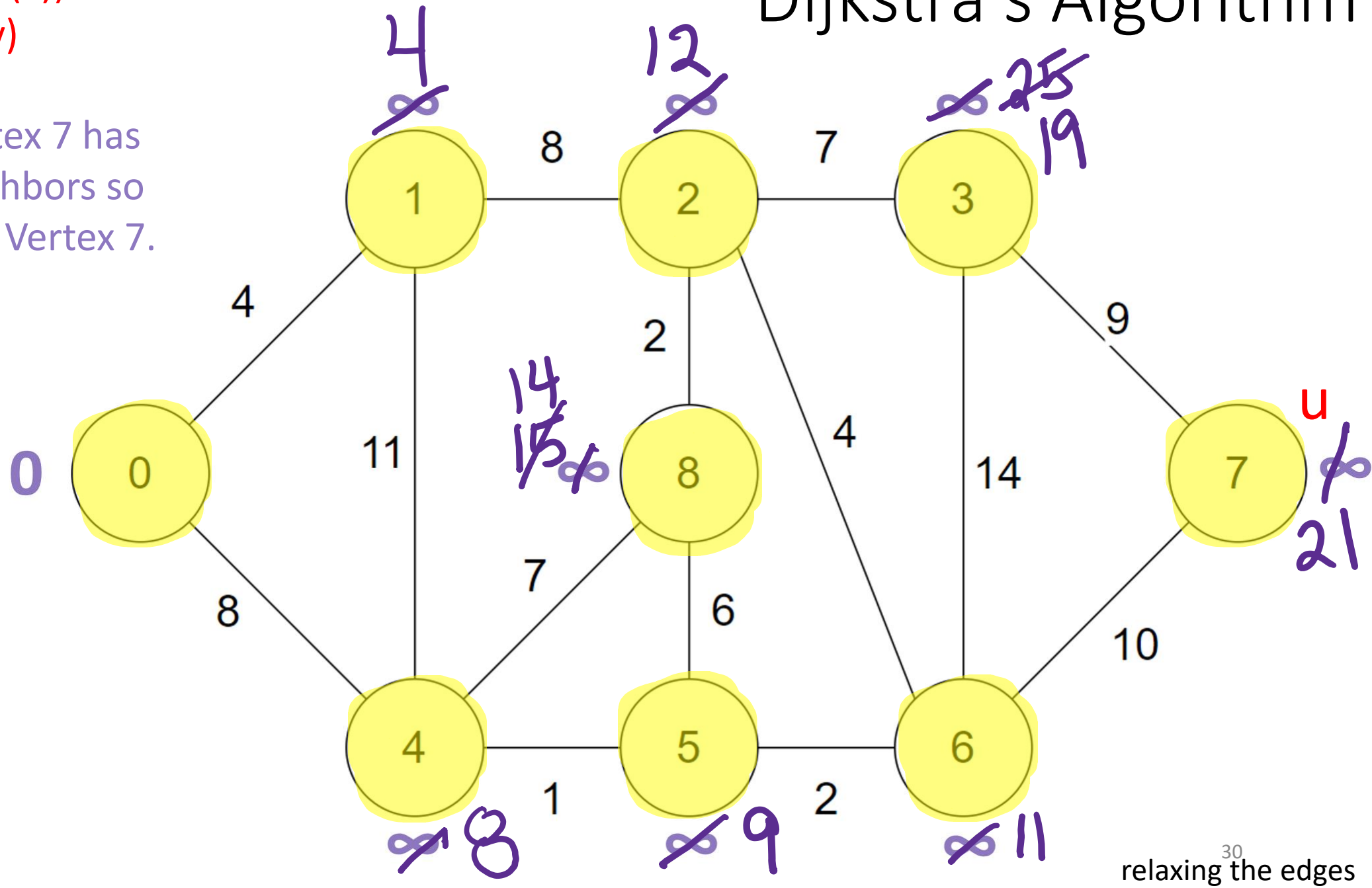
0 – already visited
1 – already visited
2 – already visited
3 – already visited
4 – already visited
5 – already visited
6 – already visited
7 – d(u) of 21
8 – already visited



Dijkstra's Algorithm

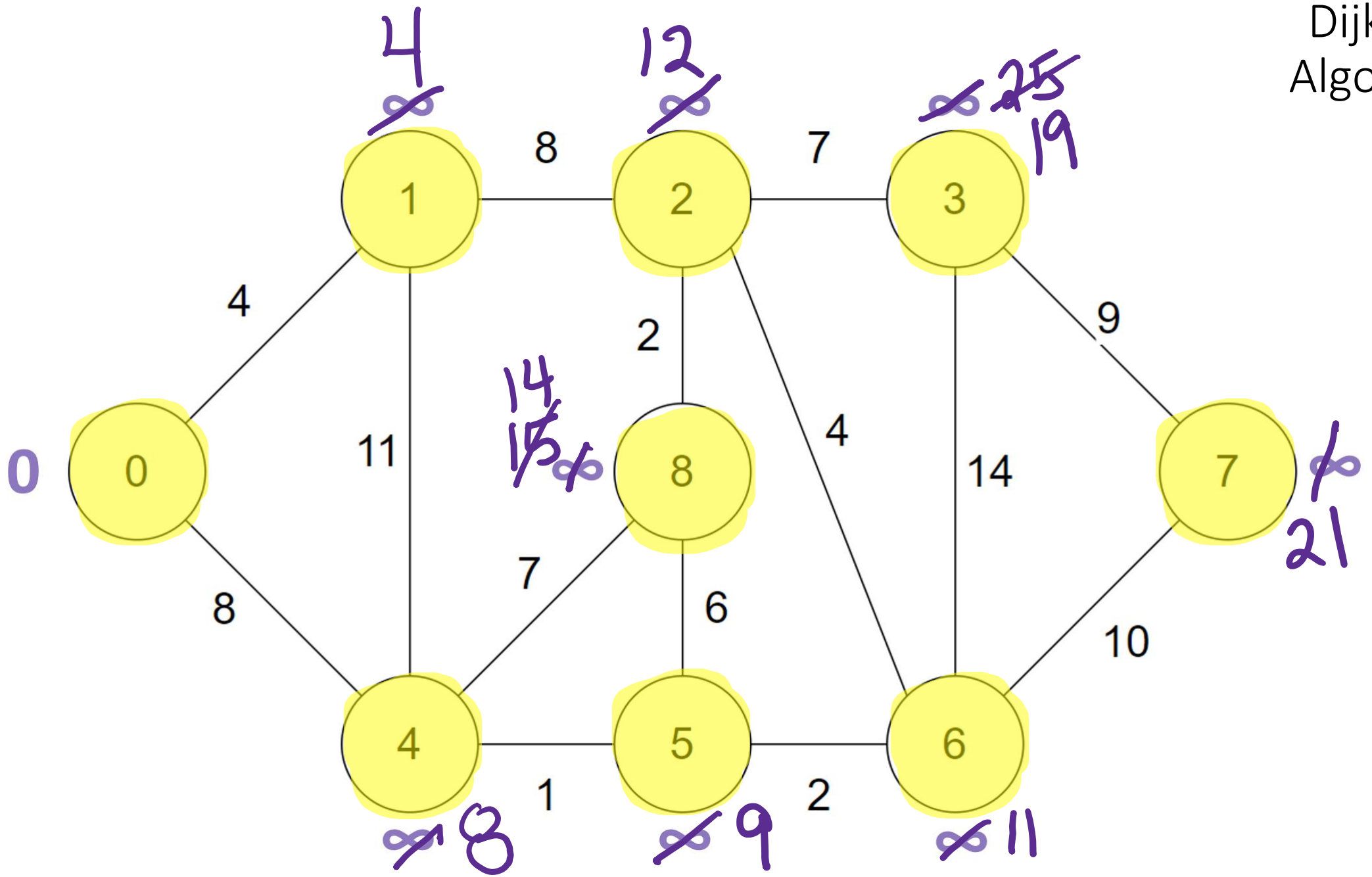
if (d(u) + c(u,v) < d(v))
 d(v) = d(u) + c(u,v)

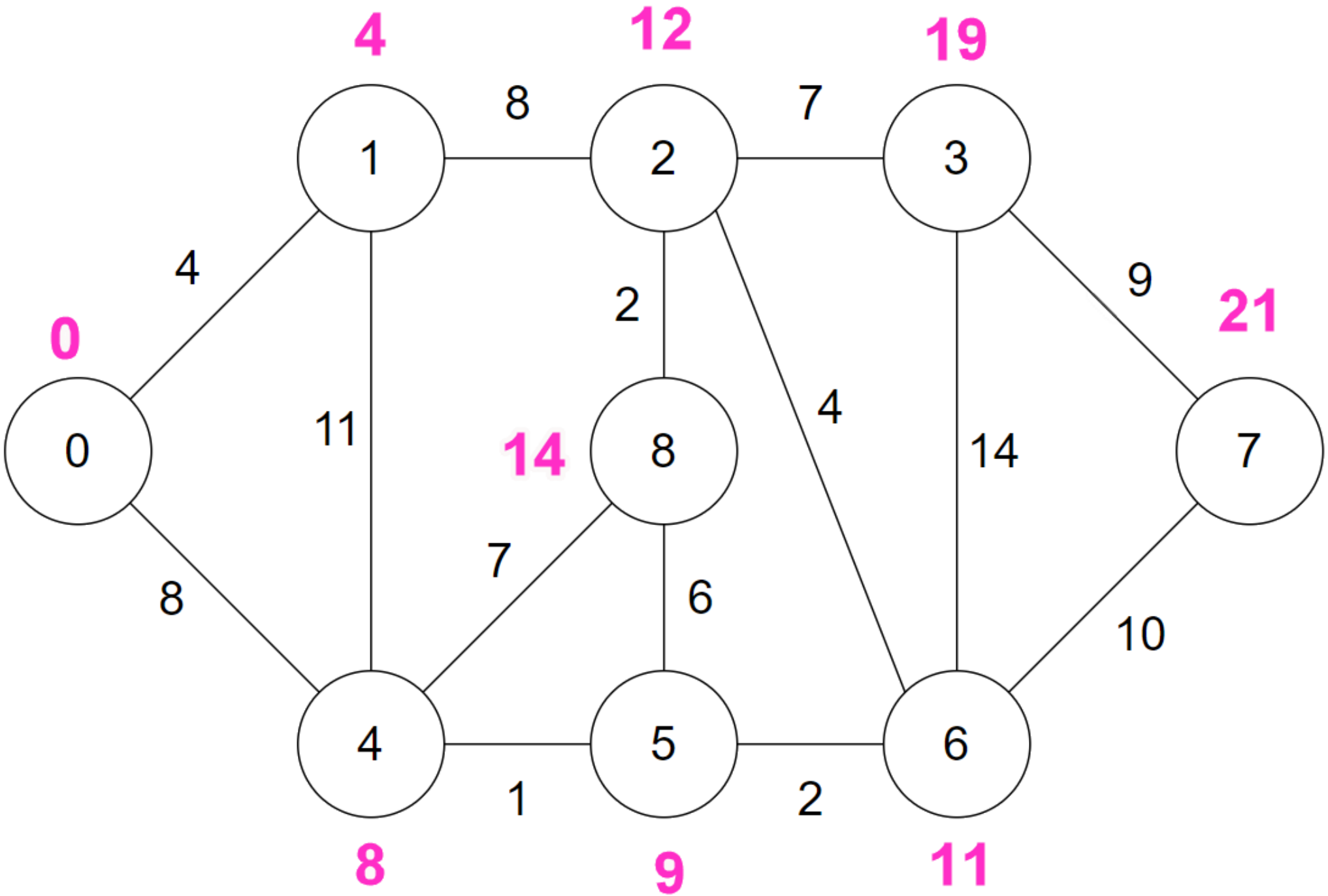
d(u) is 21 but Vertex 7 has NO unvisited neighbors so we are done with Vertex 7.



relaxing the edges

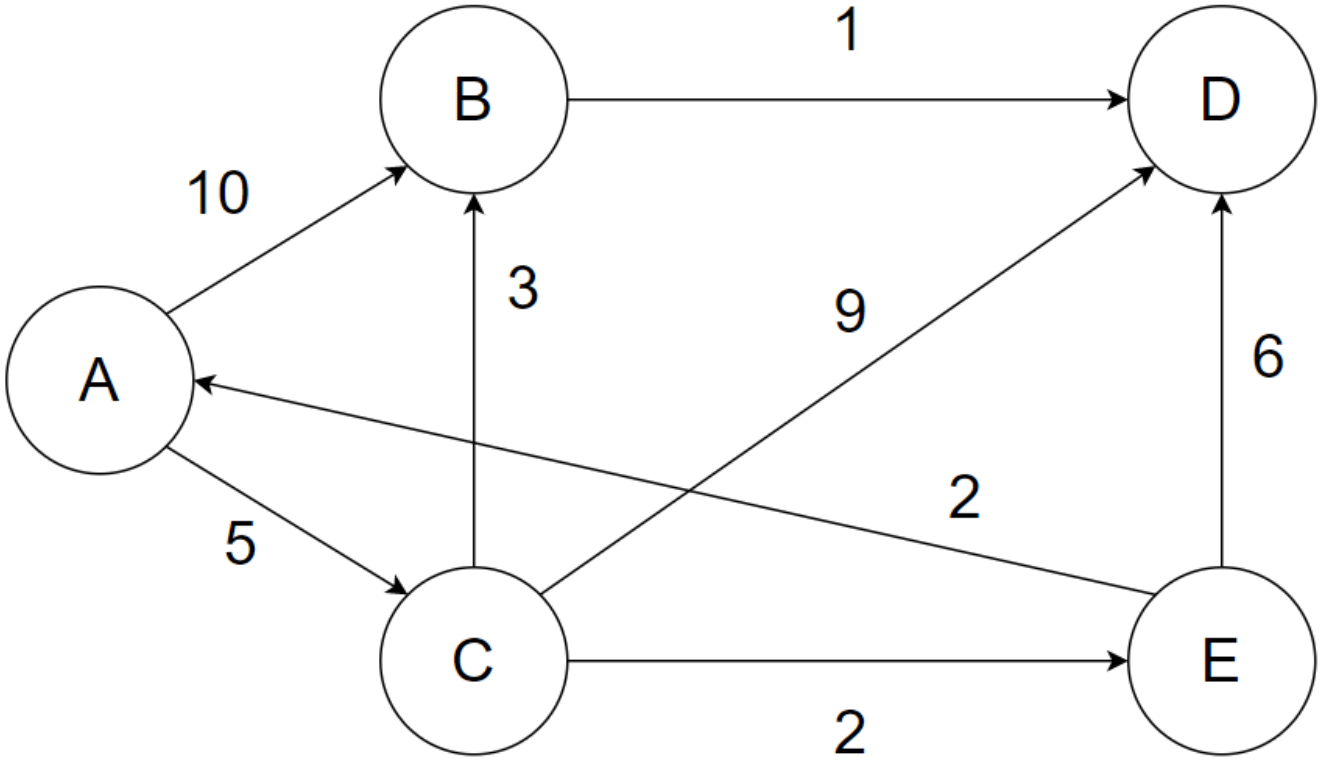
Dijkstra's Algorithm




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

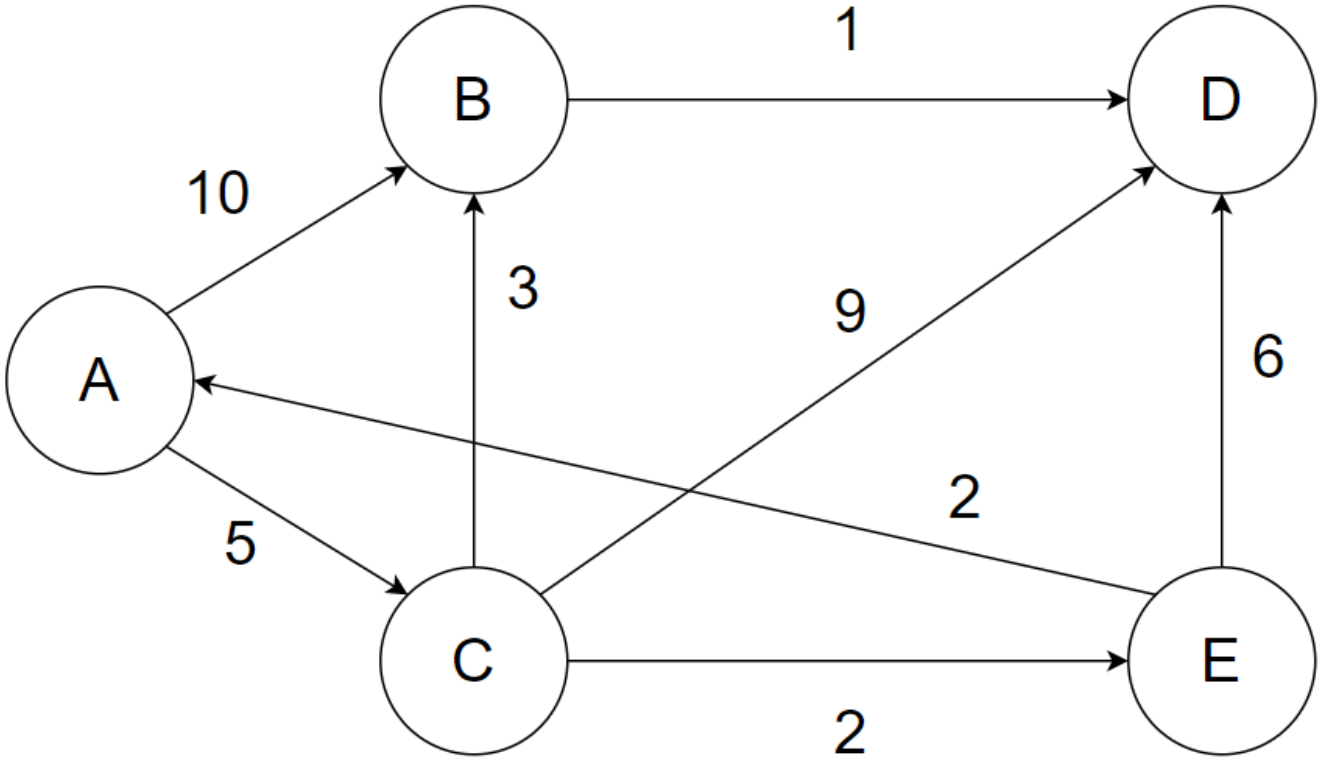
Source vertex MUST be listed first.

	A	B	C	D	E
	0	∞	∞	∞	∞



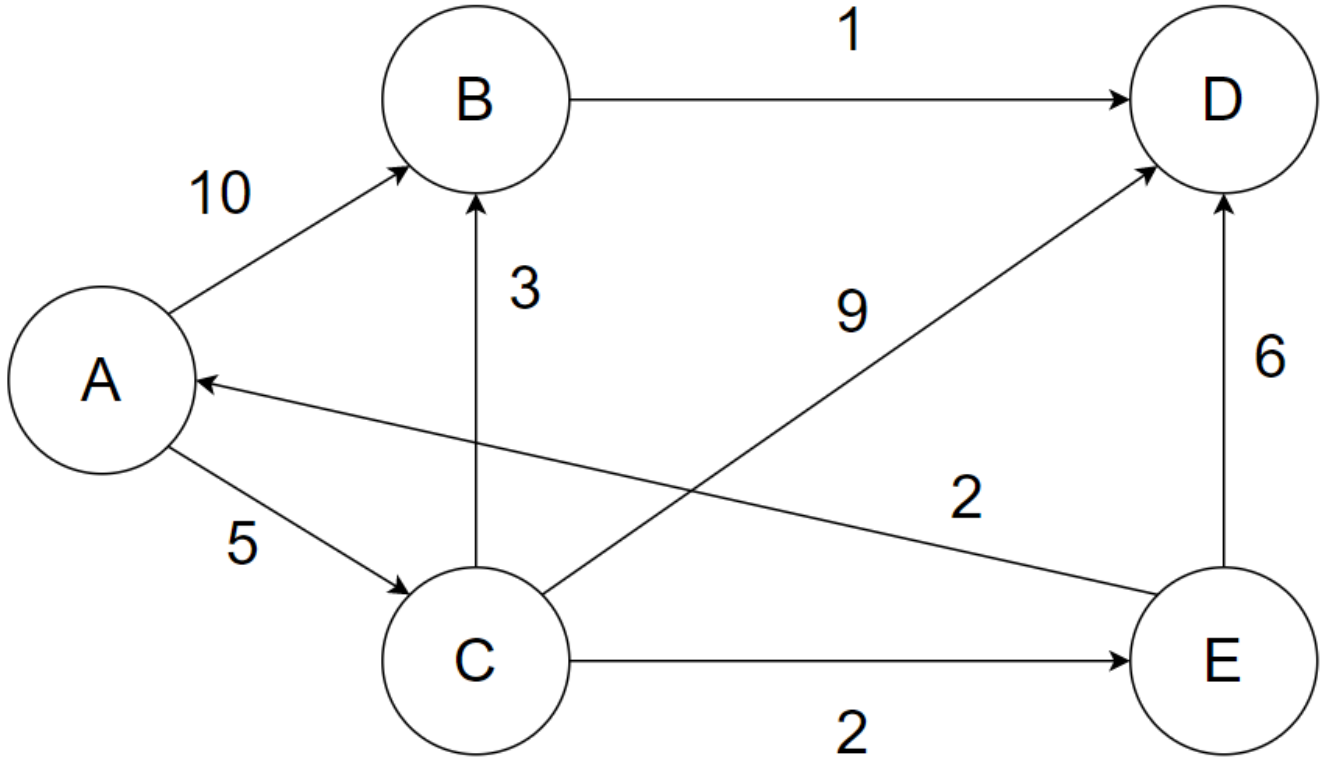
Which vertex has the smallest d(u)?

	A	B	C	D	E
A	0	∞	∞	∞	∞



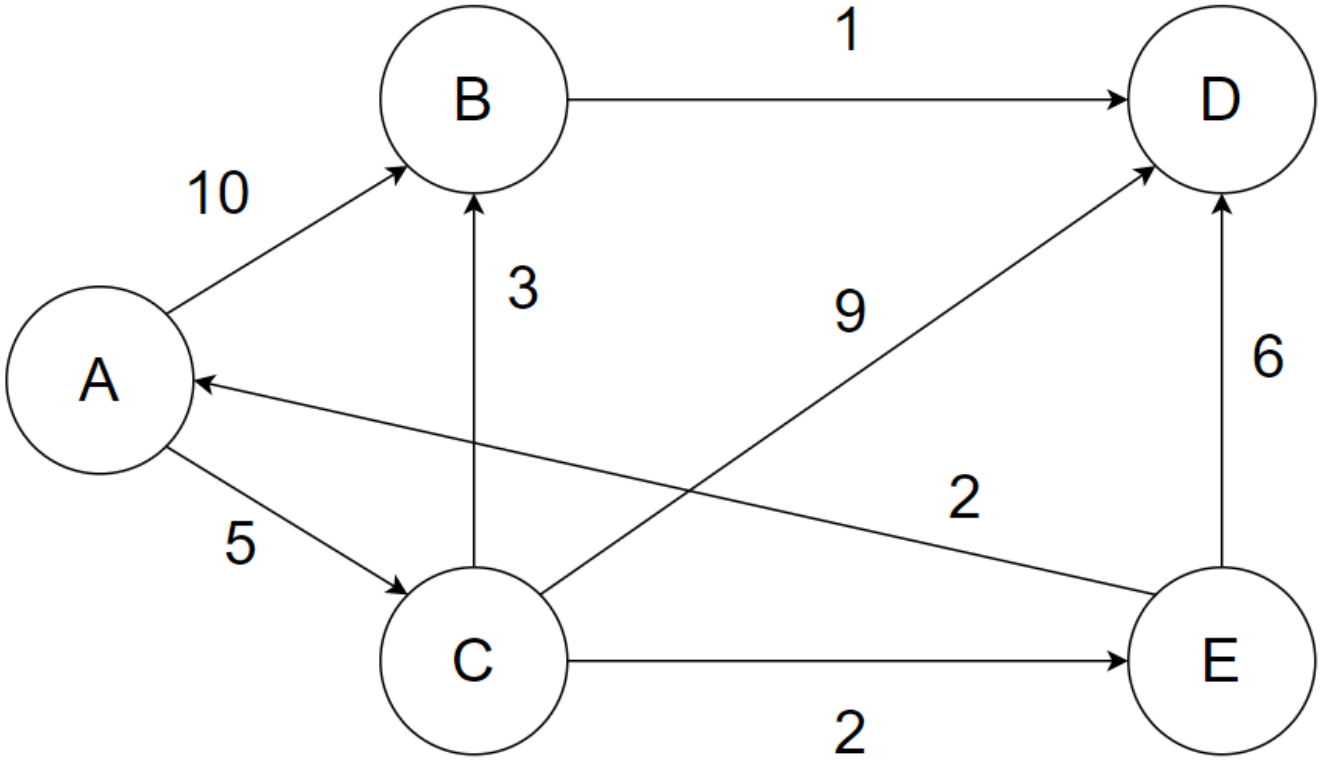
A -> B and A -> C

	A	B	C	D	E
A	0	∞	∞	∞	∞
		10	5	∞	∞



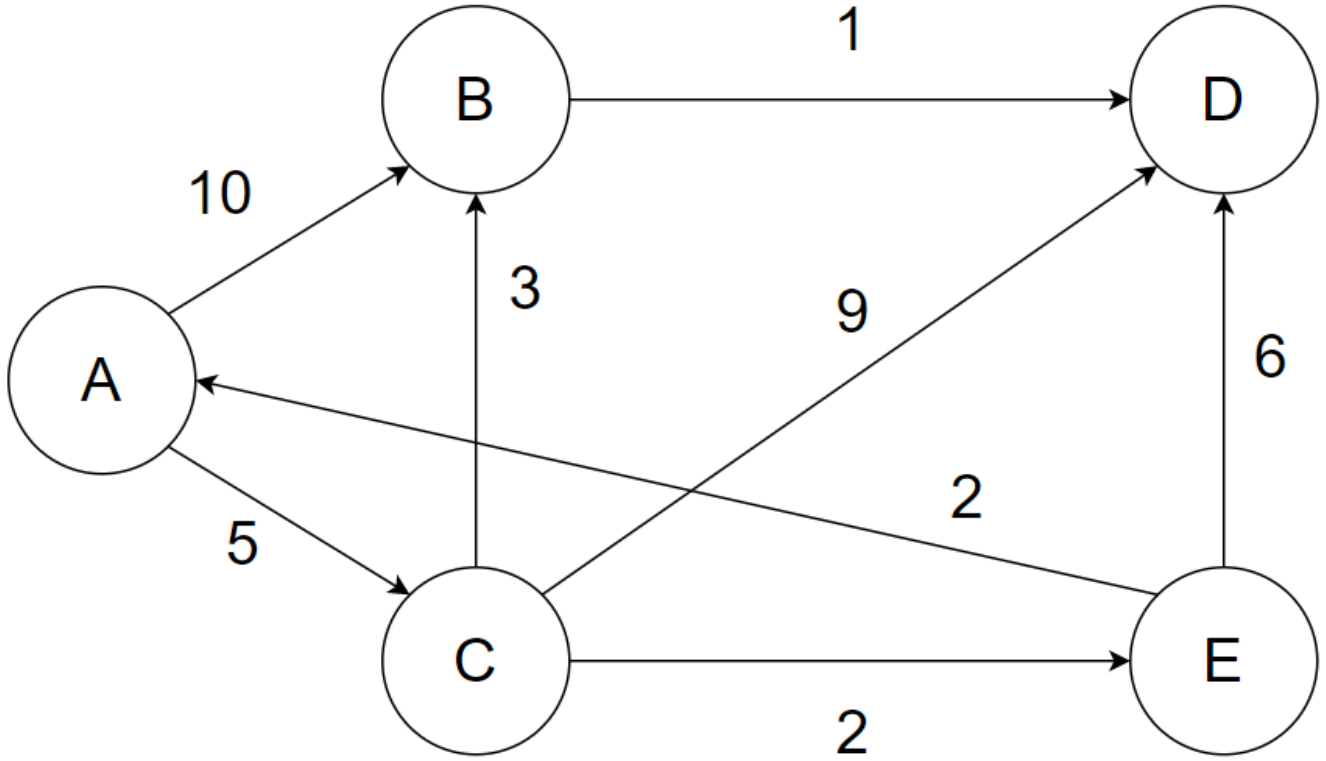
Which vertex has the smallest d(u)?

	A	B	C	D	E
A	0	∞	∞	∞	∞
C		10	5	∞	∞



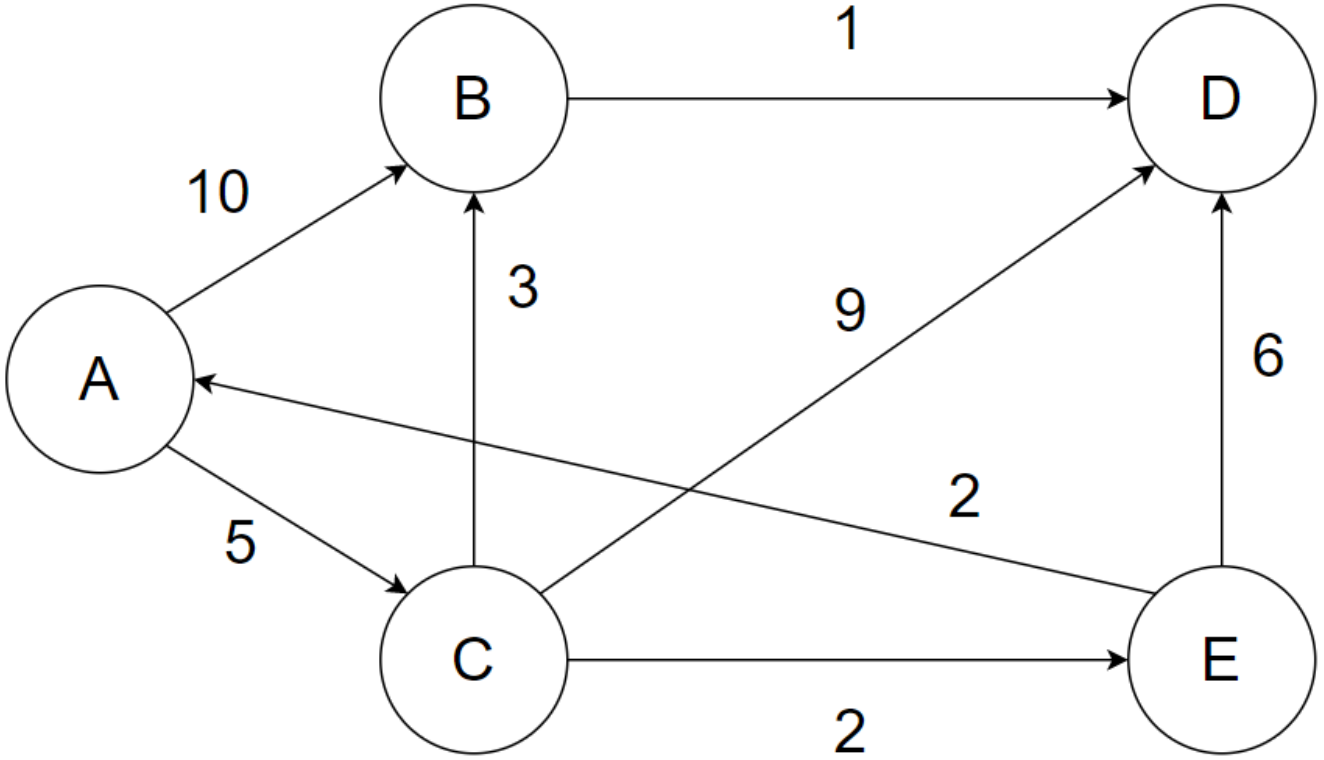
C -> B and C -> D and C->E

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



Which vertex has the smallest d(u)?

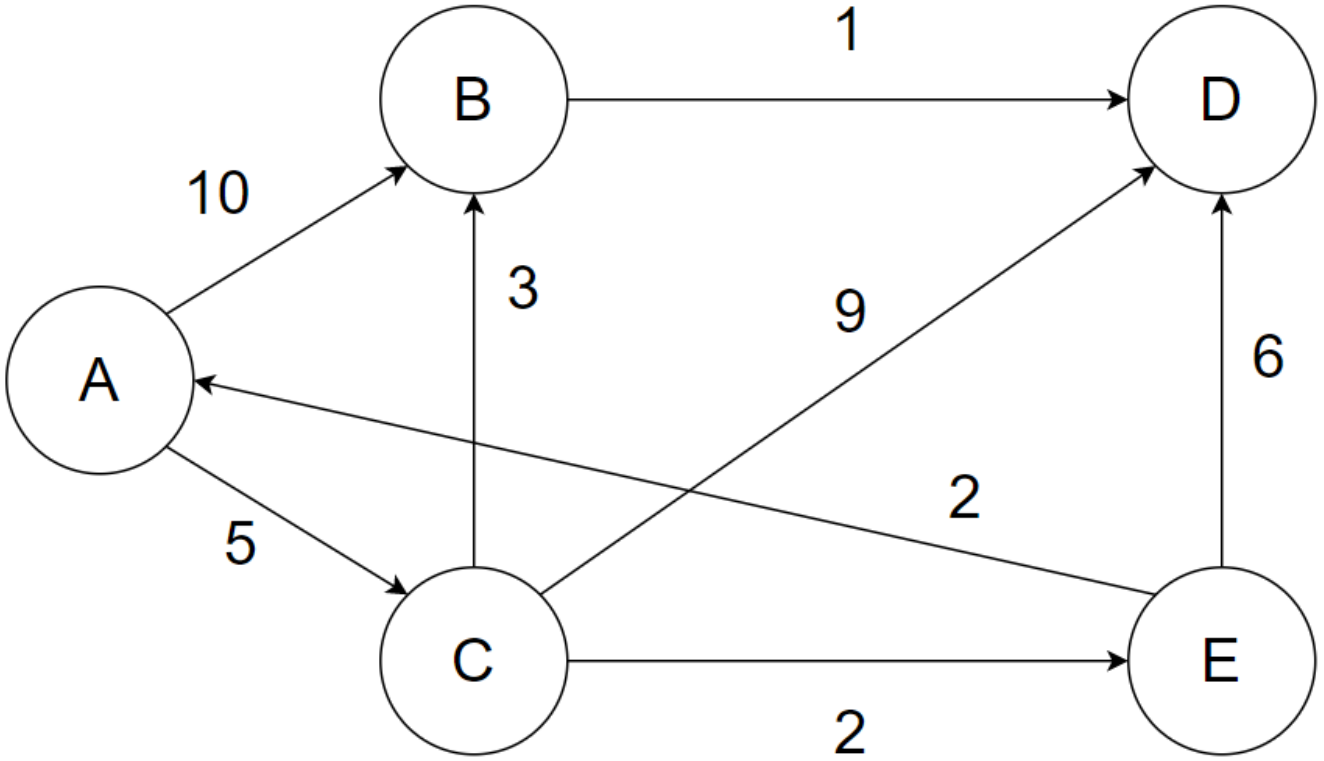
	A	B	C	D	E
A	0	∞	∞	∞	∞
C		10	5	∞	∞
E		8		14	7



E -> A and E -> D

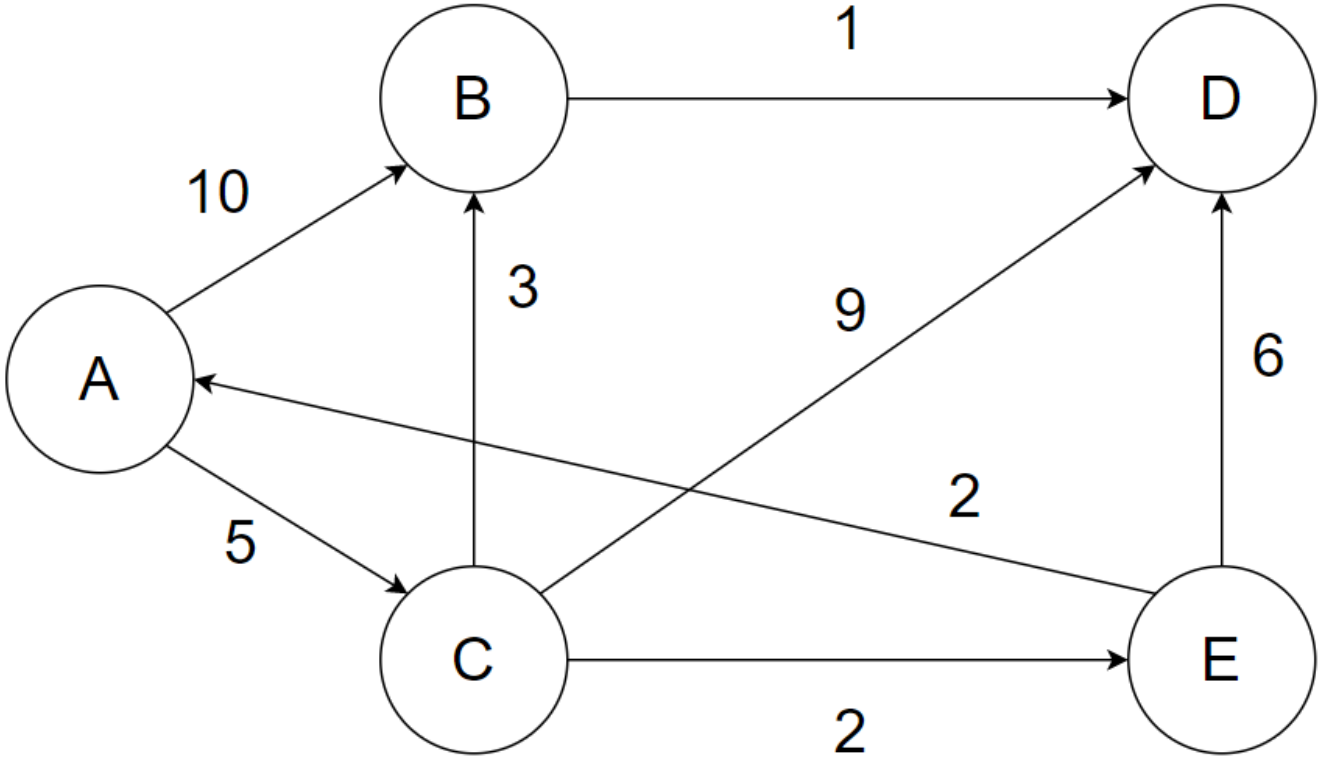
7 + 6

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



Which vertex has the smallest d(u)?

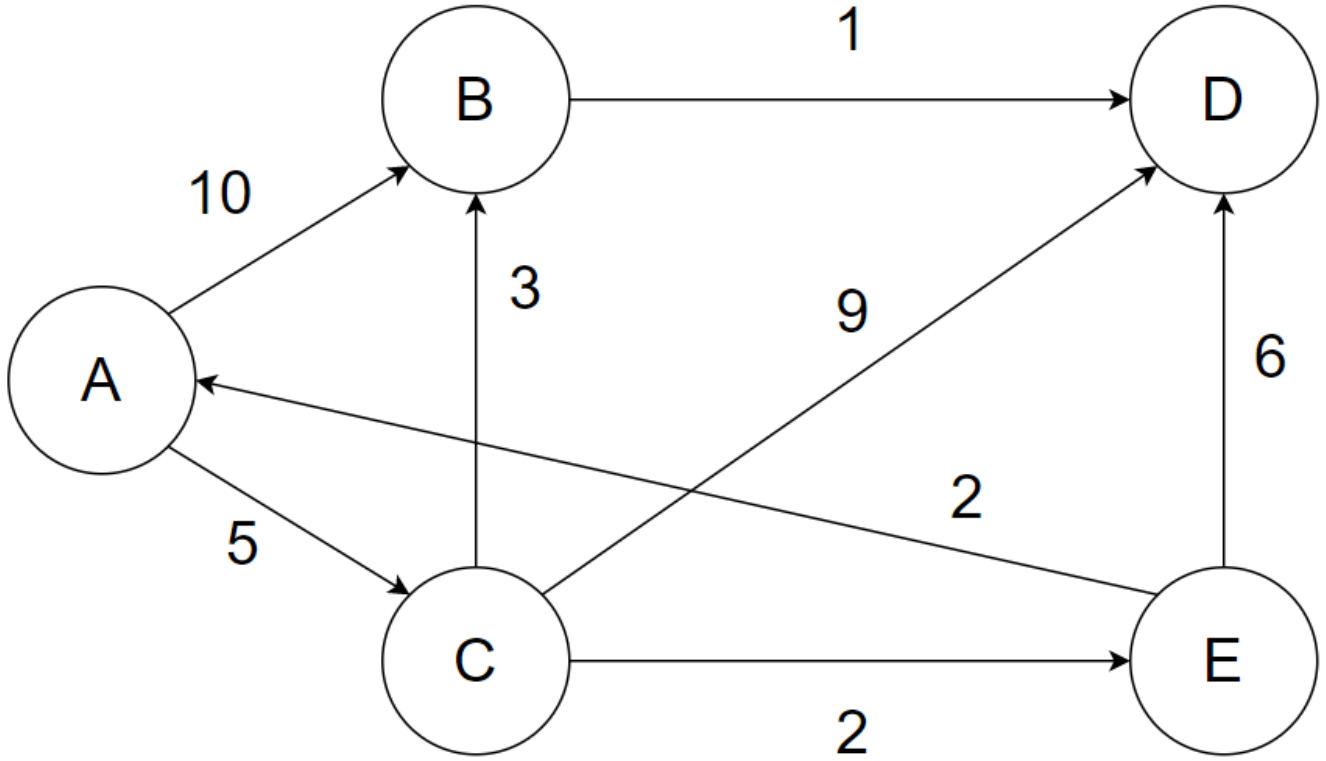
	A	B	C	D	E
A	0	∞	∞	∞	∞
C		10	5	∞	∞
E		8		14	7
B		8		13	



B -> D

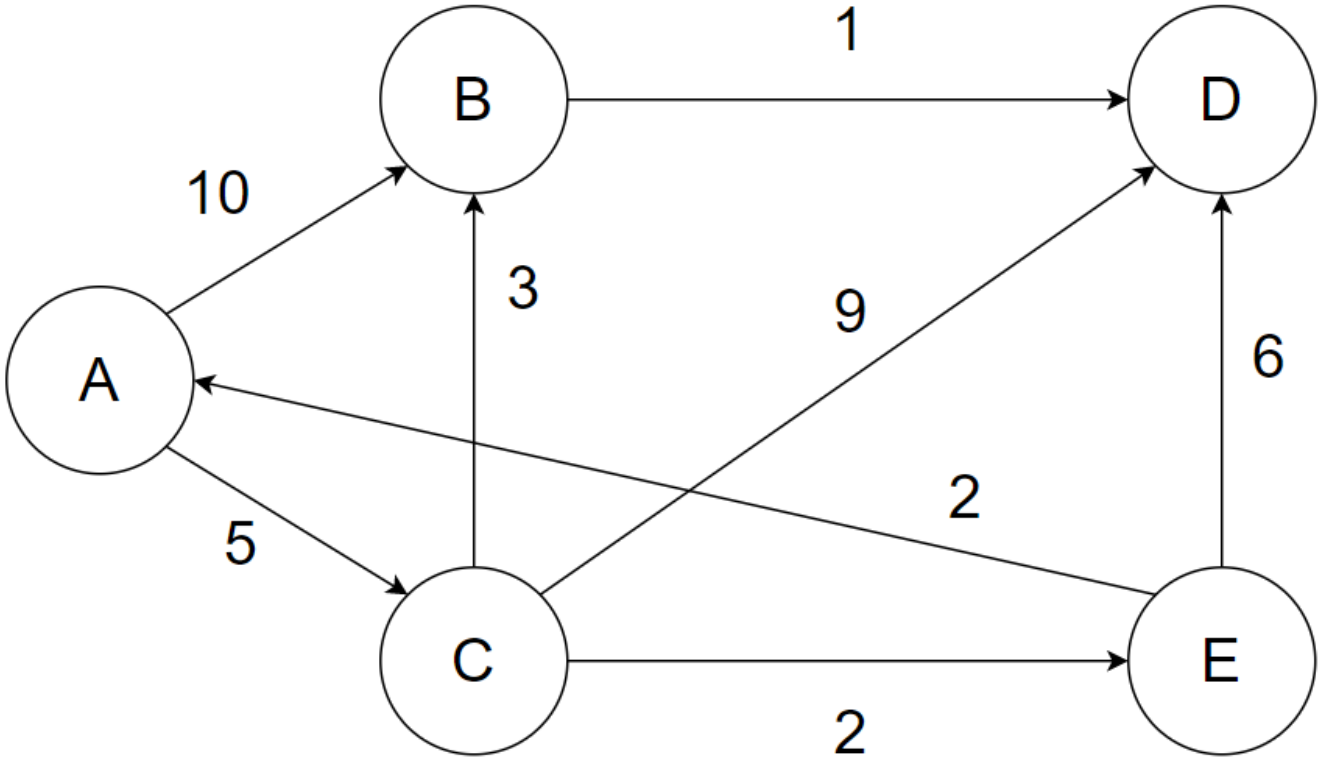
8 + 1

	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)?

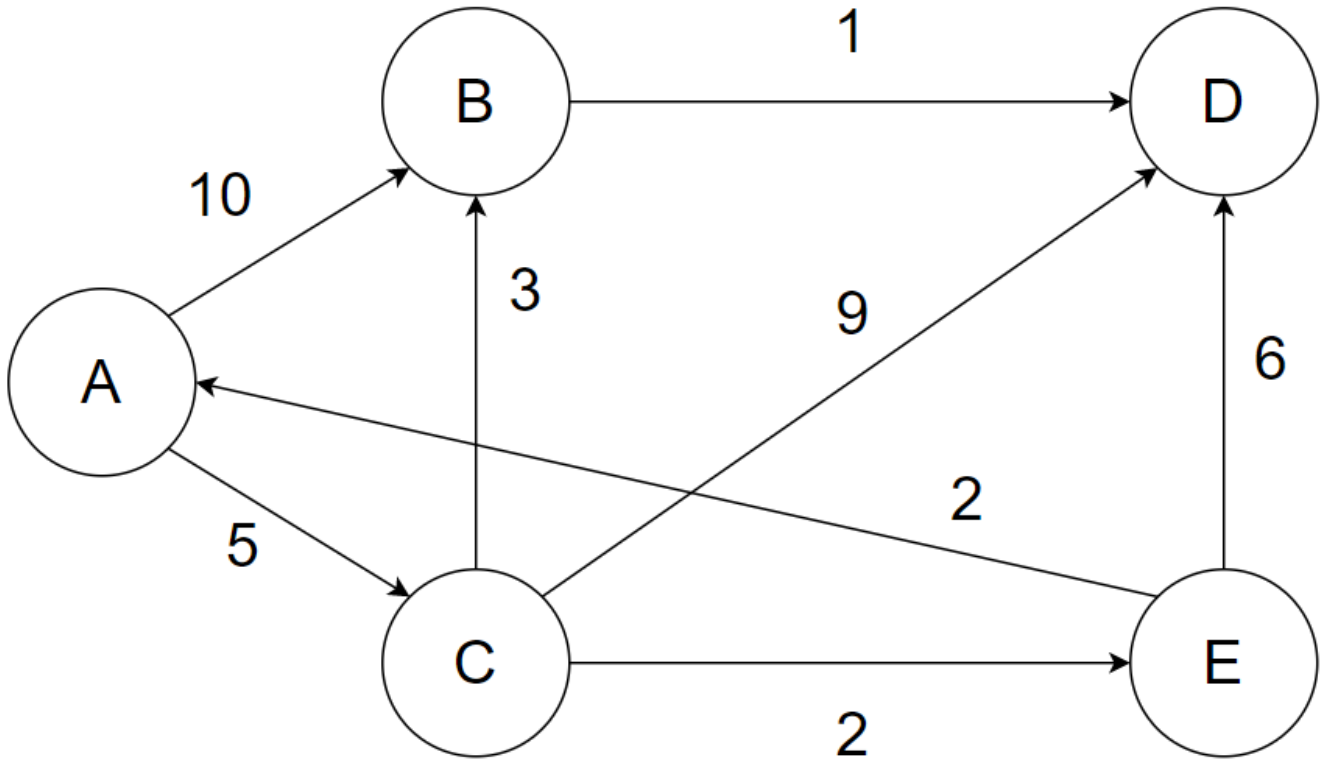
	A	B	C	D	E
A	0	∞	∞	∞	∞
C		10	5	∞	∞
E		8		14	7
B		8		13	
D				9	



What is the shortest distance from A to D?
9

What is the shortest distance from A to B?
8

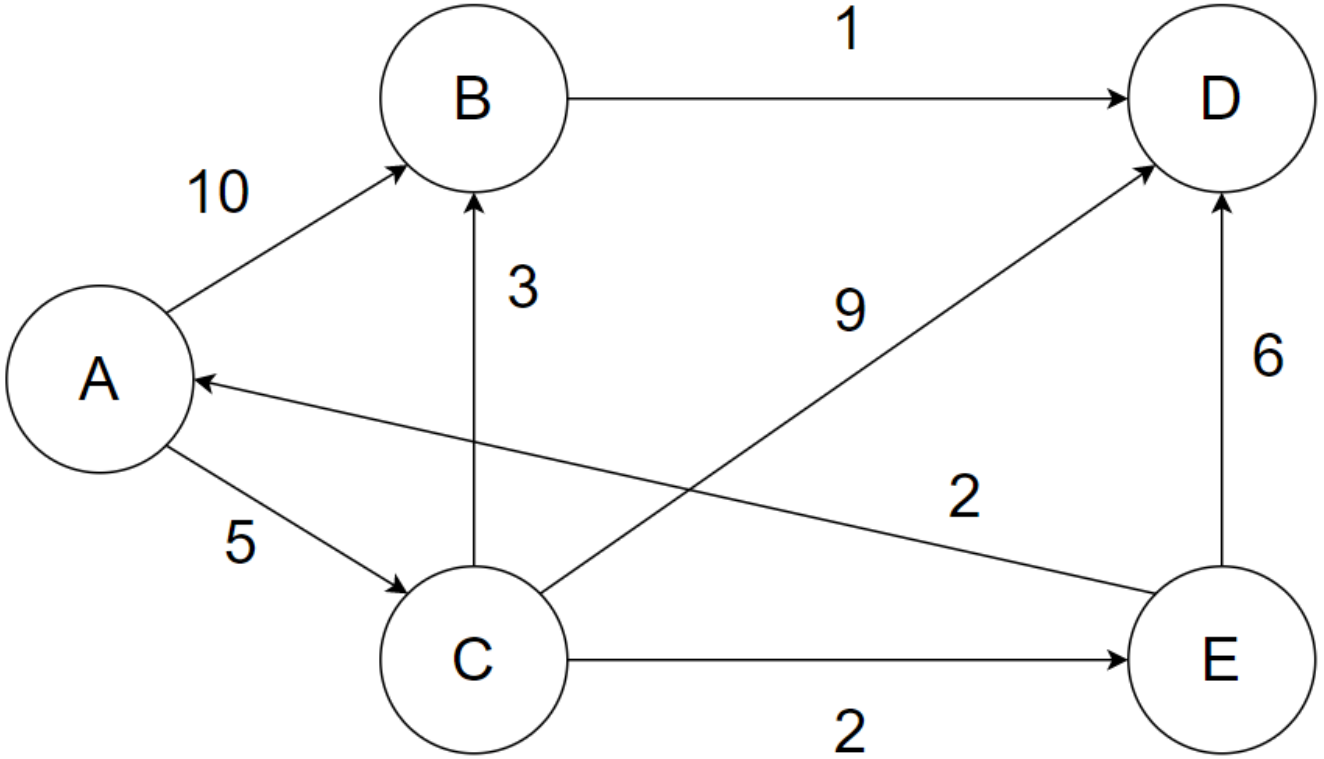
	A	B	C	D	E
A	0	∞	∞	∞	∞
C		10	5	∞	∞
E		8		14	7
B		8		13	
D				9	



What is the path from A to D?

ACBD

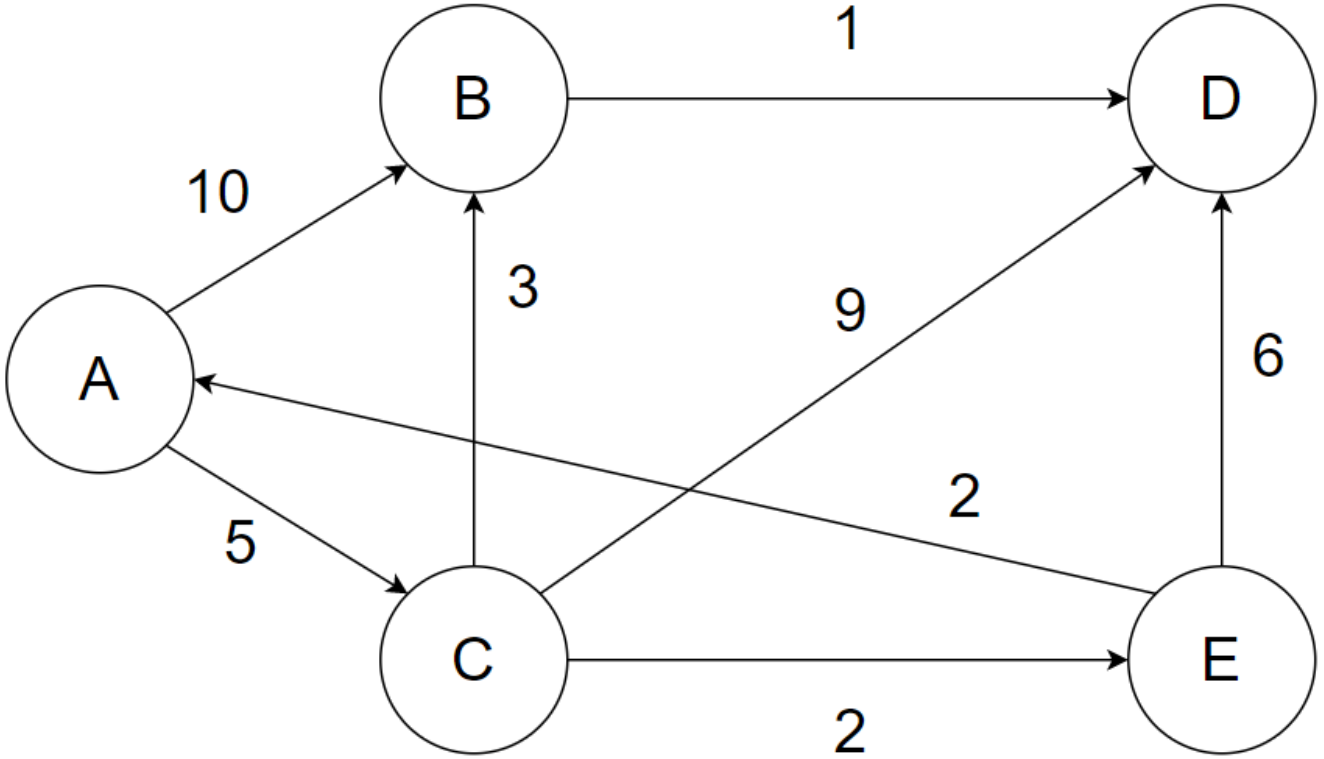
	A	B	C	D	E
A	0	∞	∞	4	∞
C		10	5	∞	∞
E		8	3	14	7
B		8		13	2
D				9	1



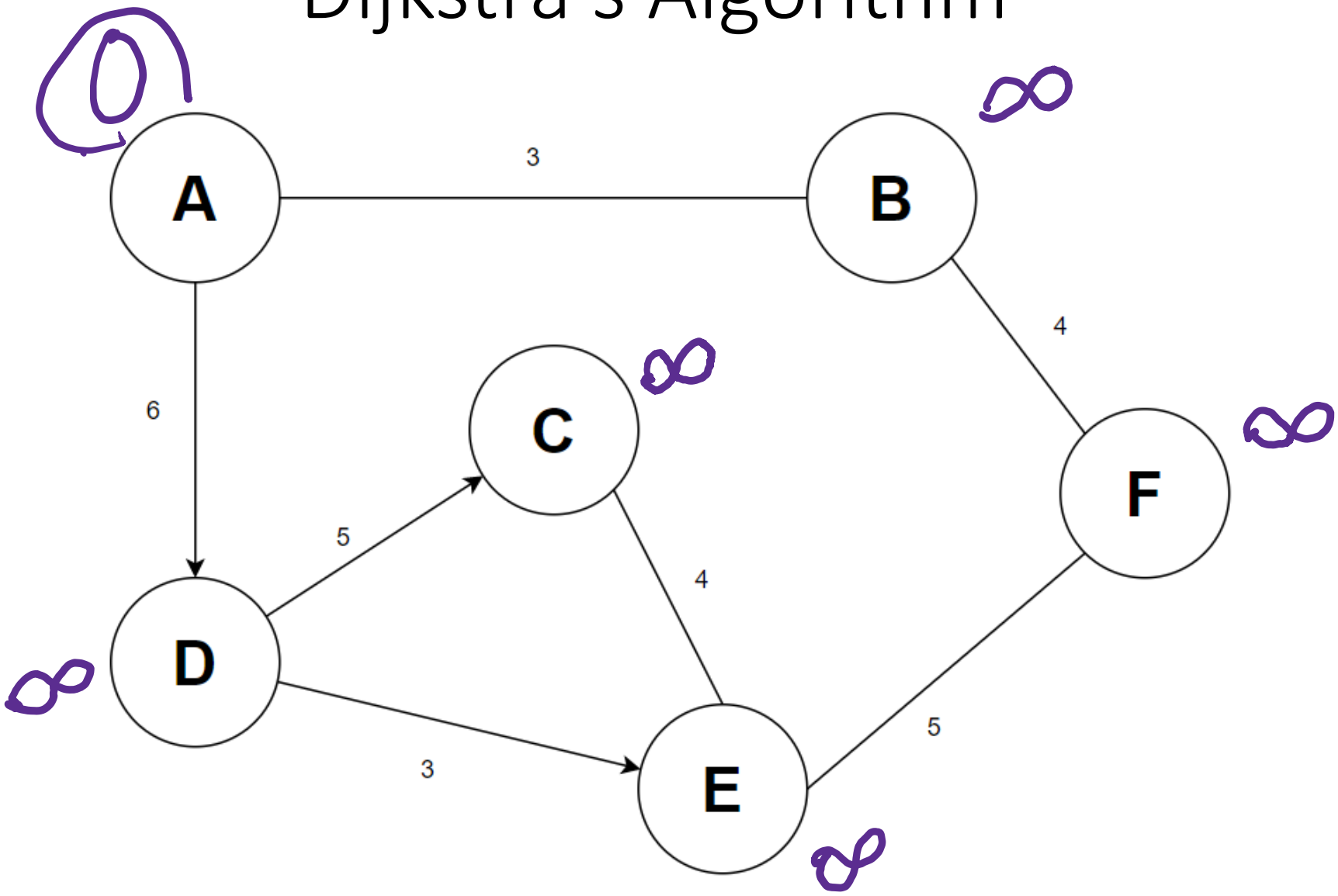
What is the path from A to B?

ACB

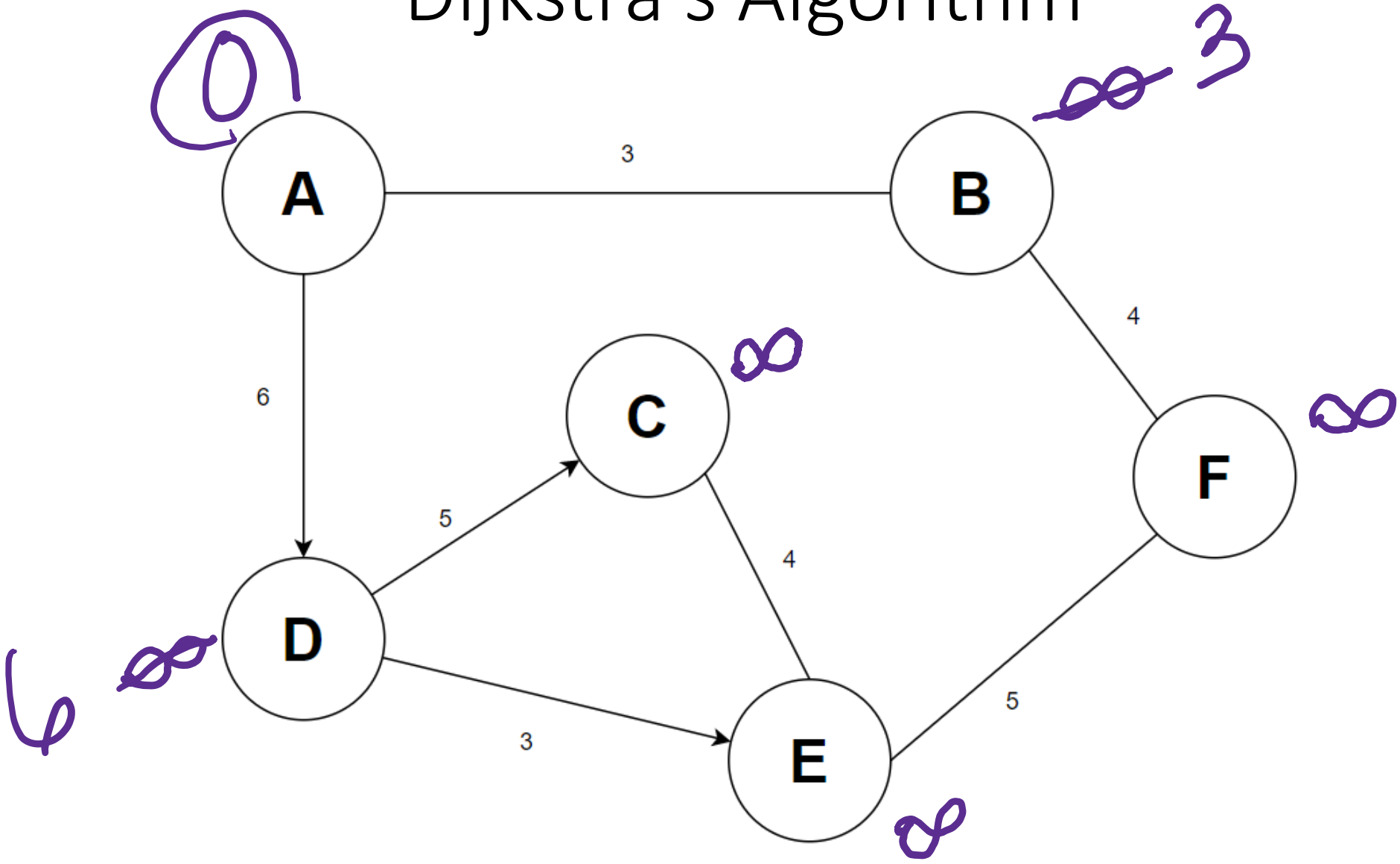
	A	B	C	D	E
A	0	∞	∞	3	∞
C		10	5	∞	∞
E		8	2	14	7
B		8	1	13	
D				9	



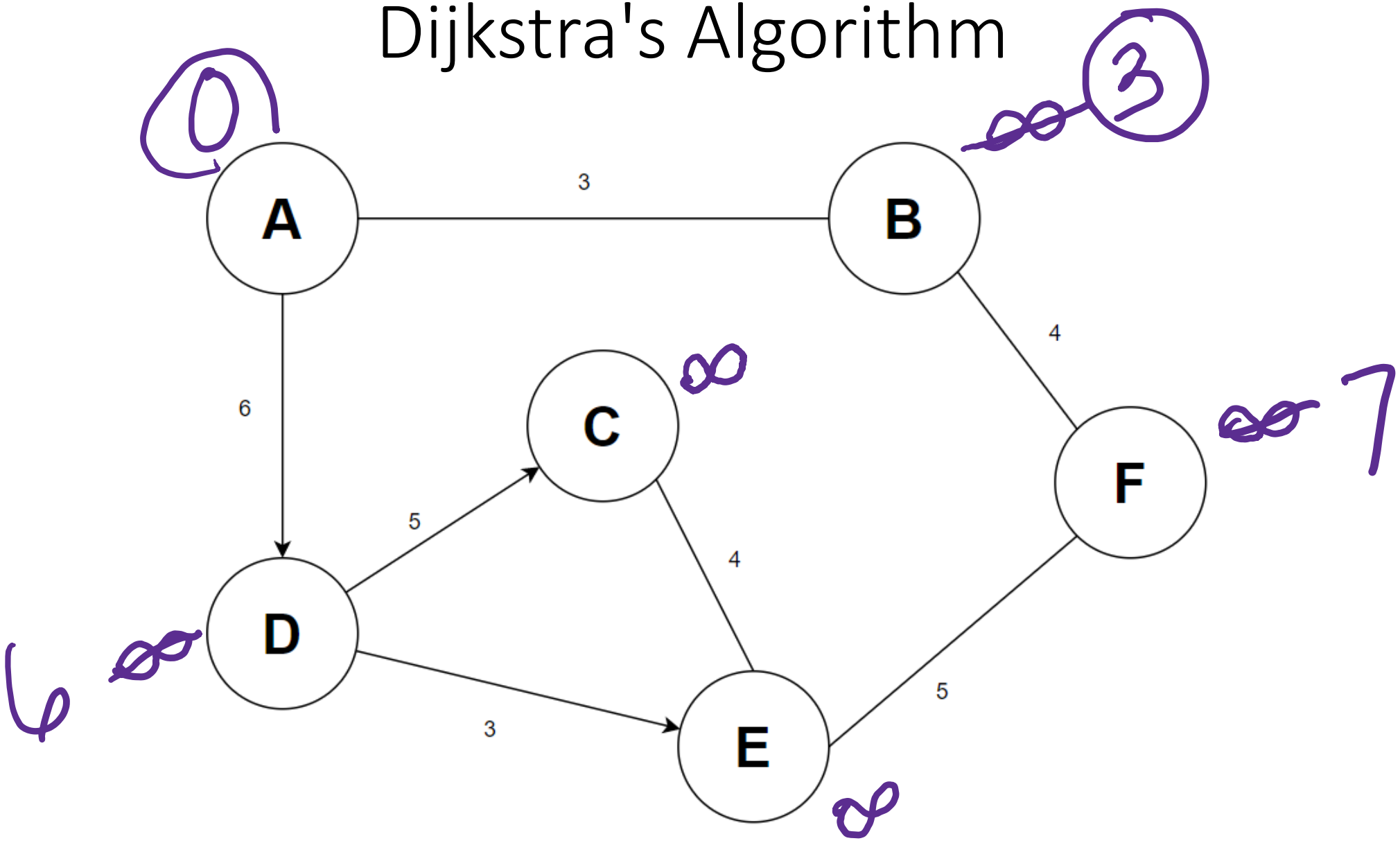
Dijkstra's Algorithm



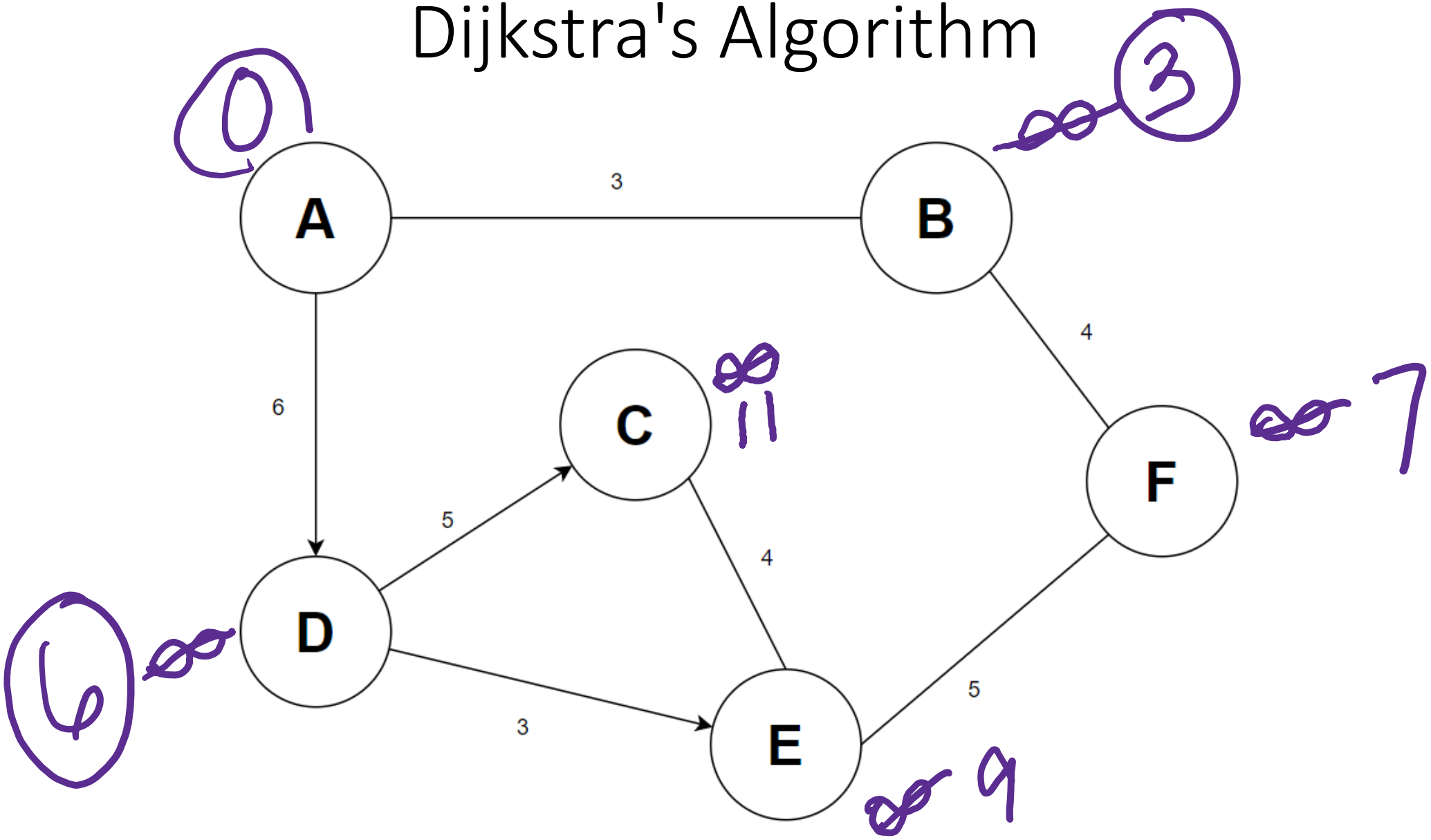
Dijkstra's Algorithm



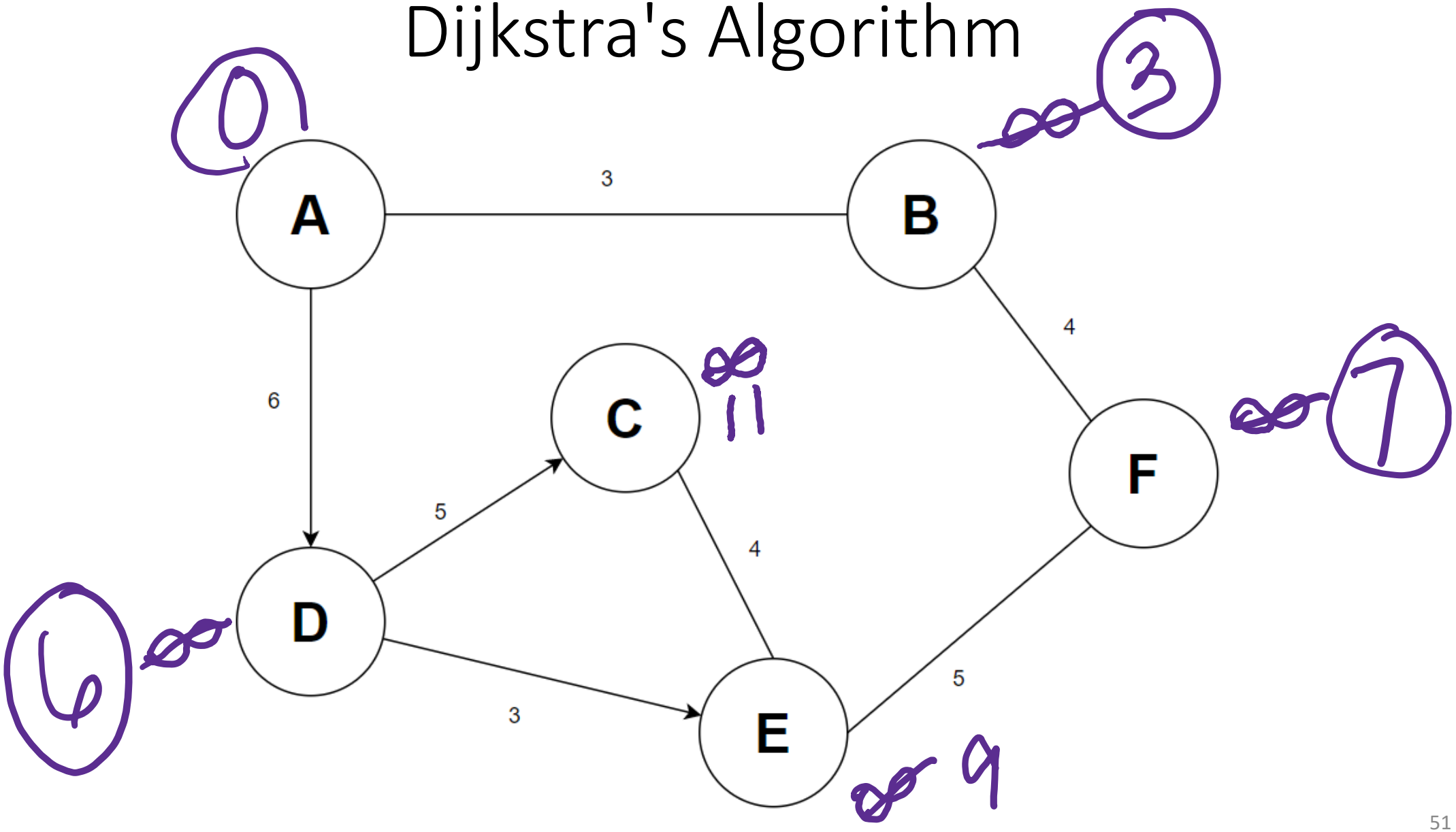
Dijkstra's Algorithm



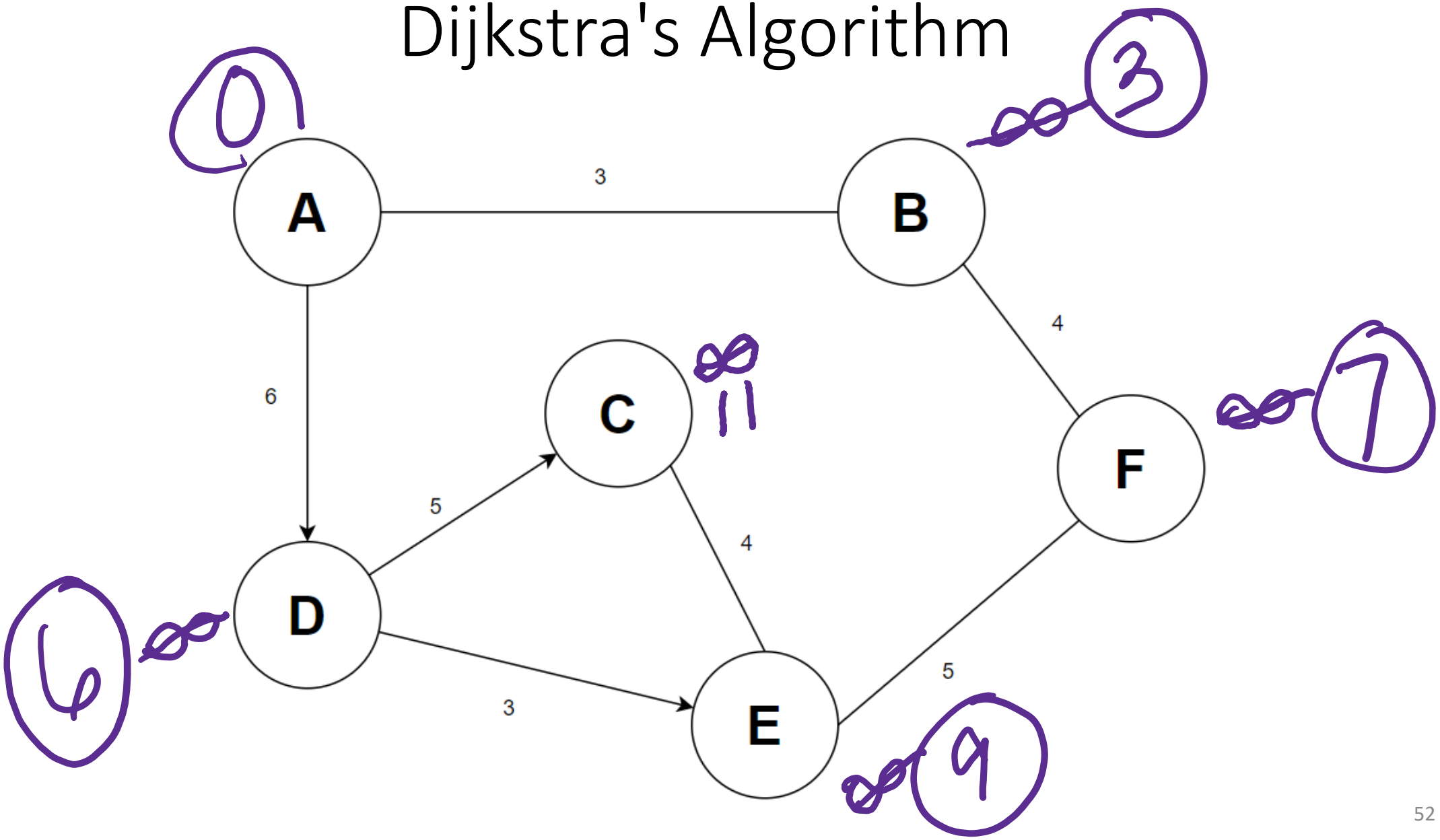
Dijkstra's Algorithm



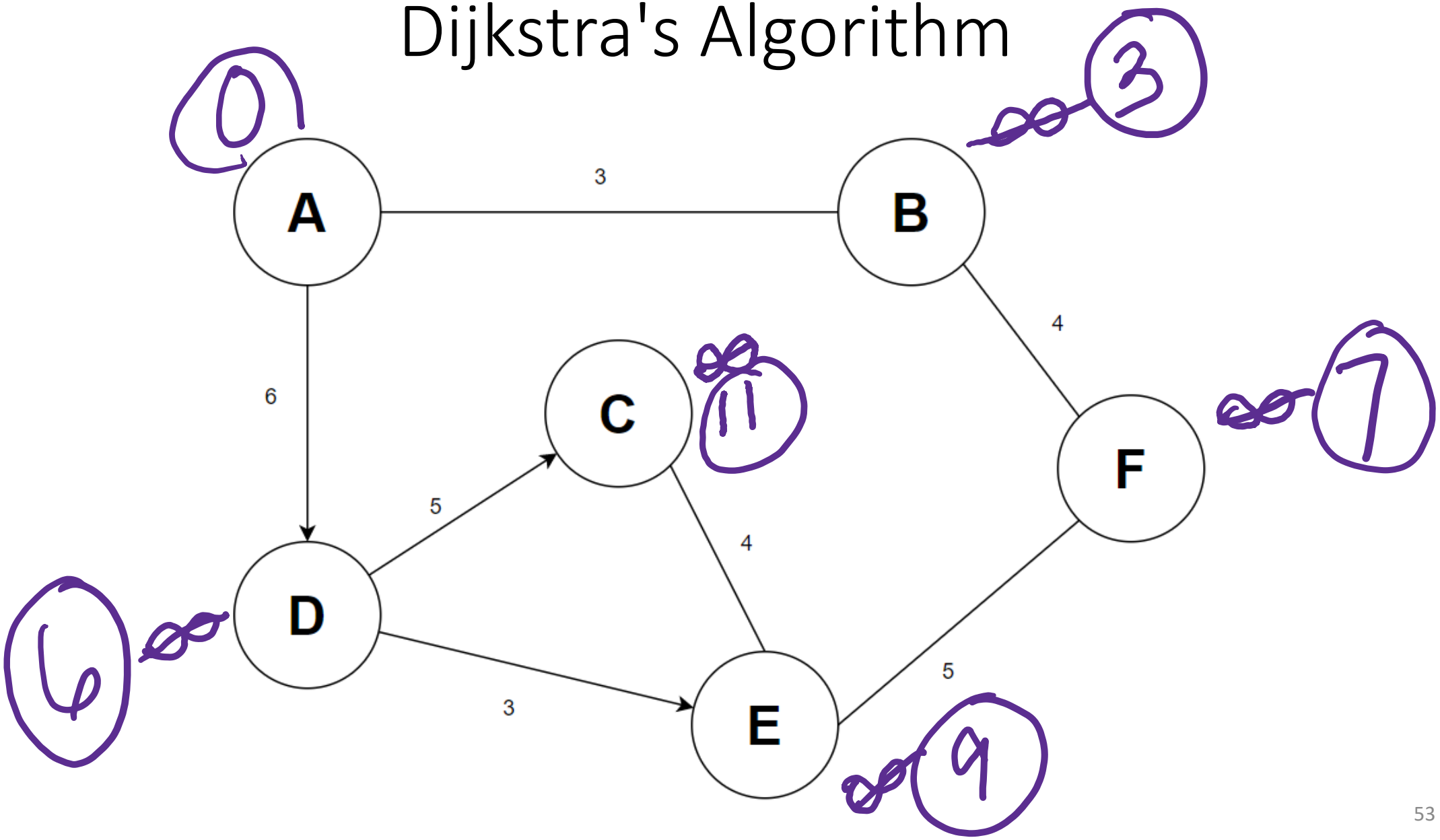
Dijkstra's Algorithm



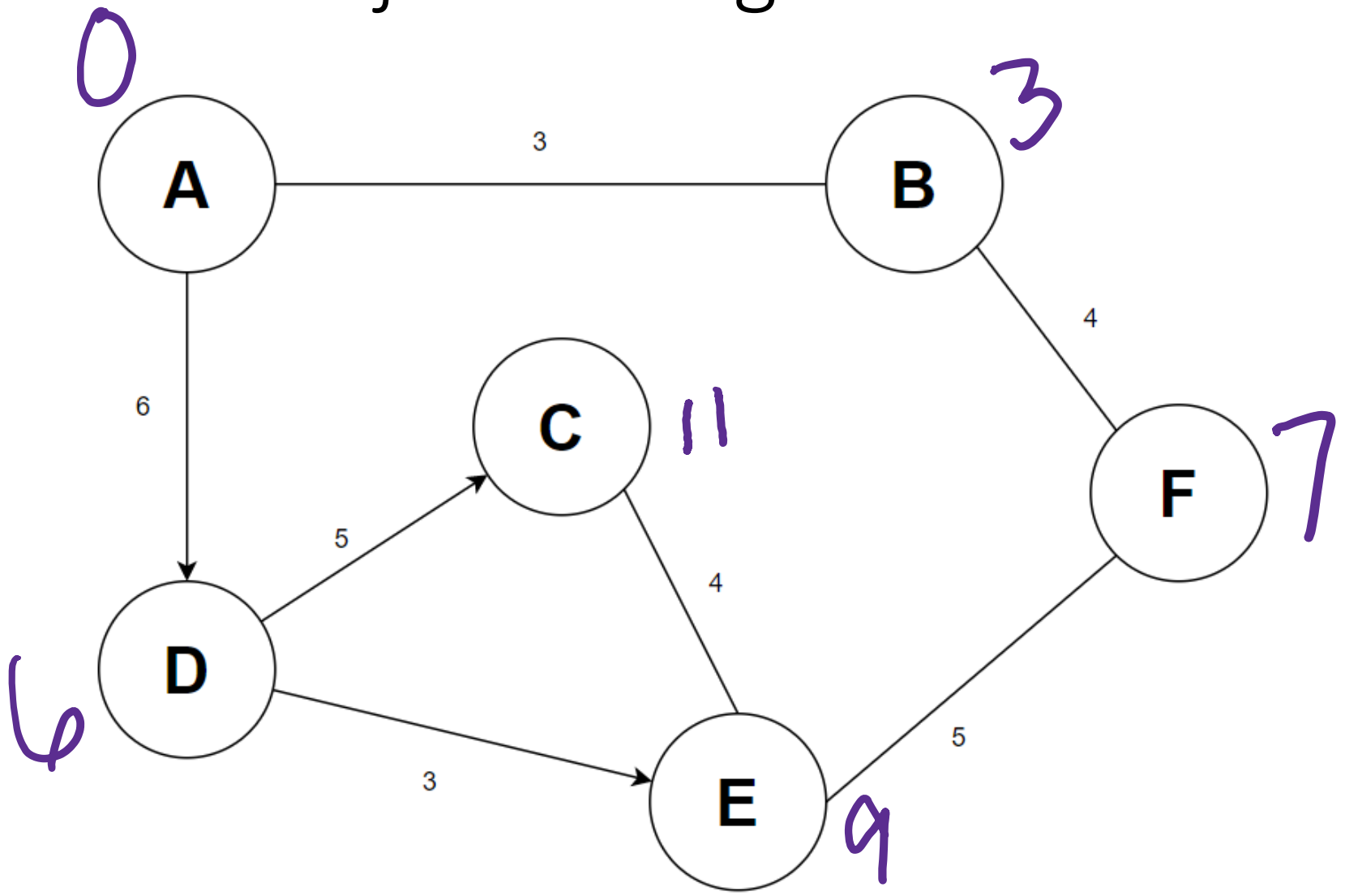
Dijkstra's Algorithm



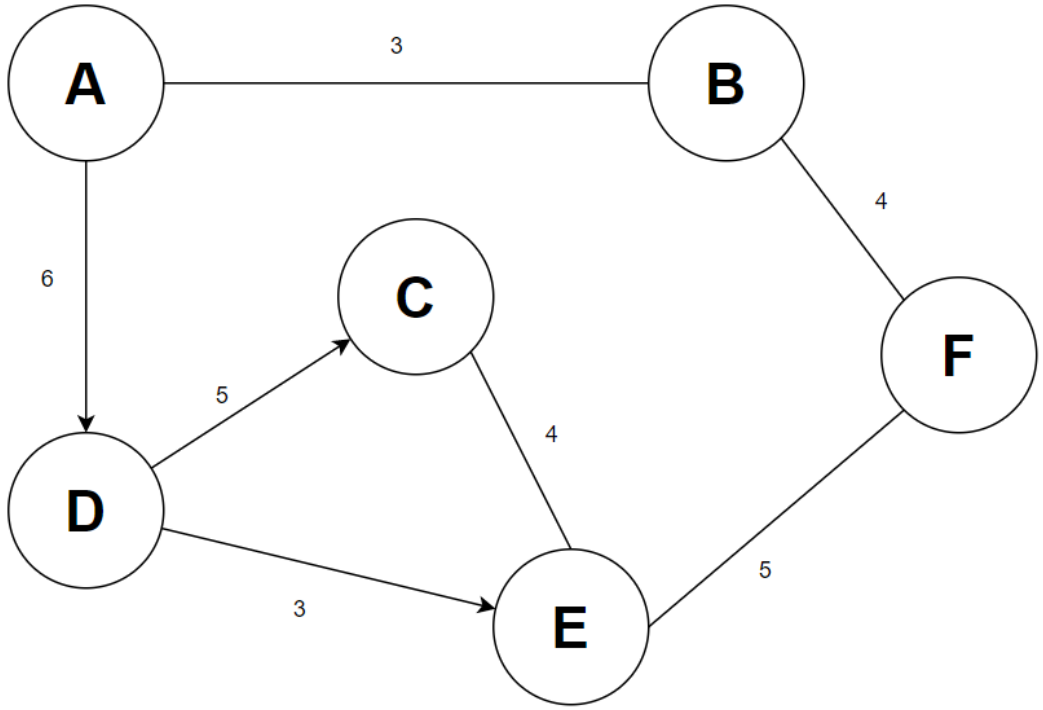
Dijkstra's Algorithm



Dijkstra's Algorithm

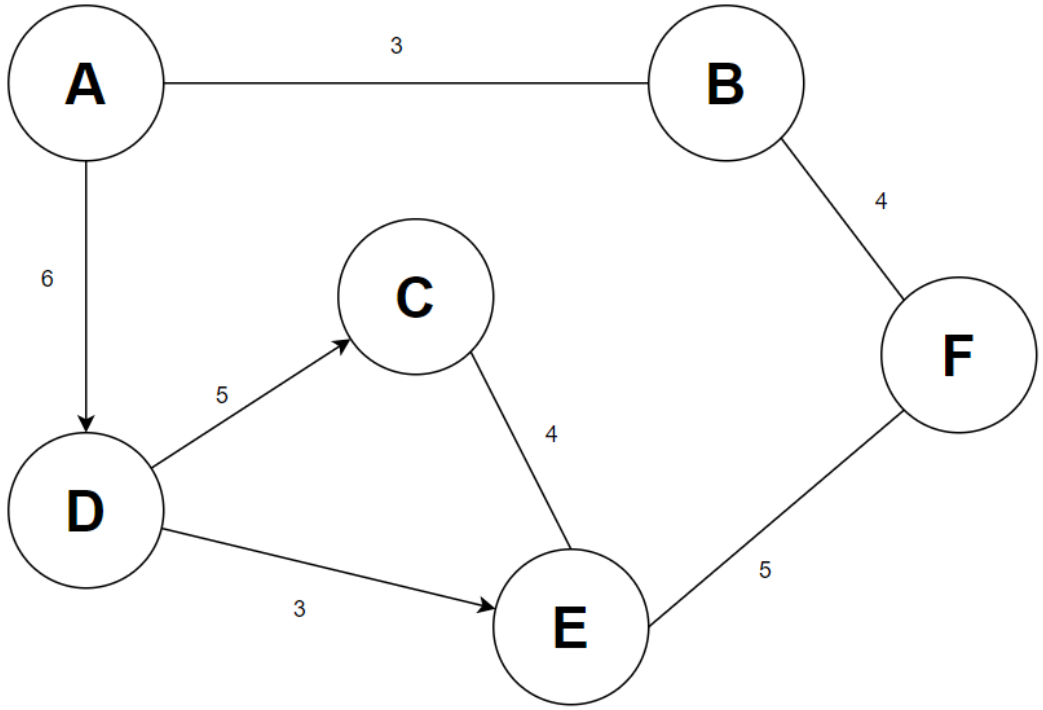


Dijkstra's Algorithm



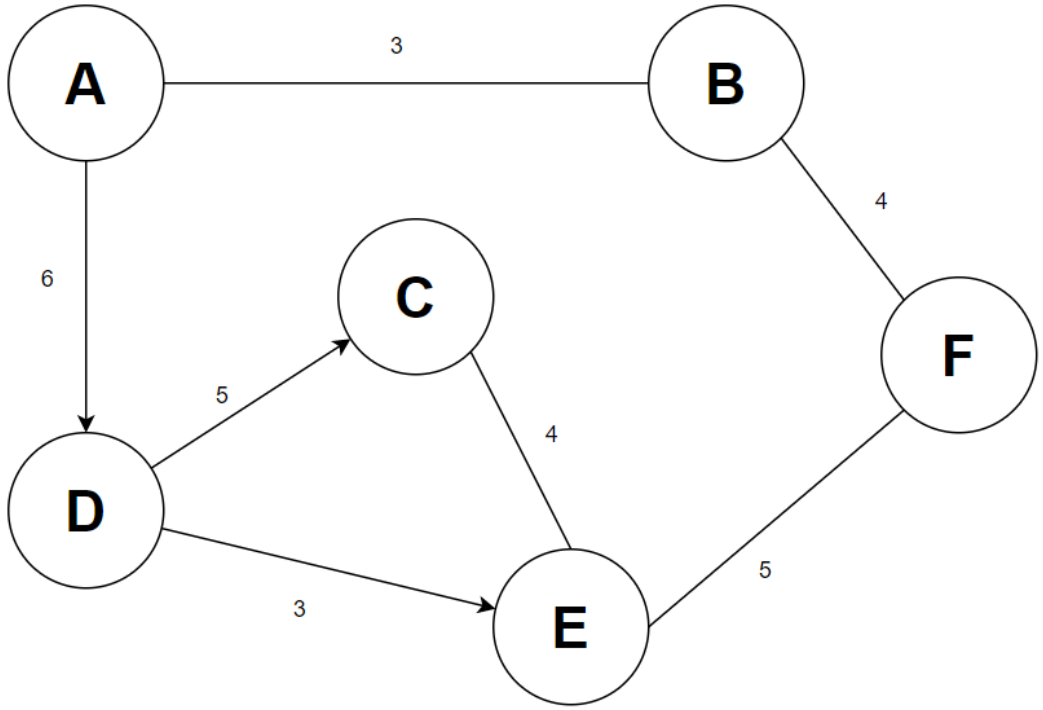
	A	B	C	D	E	F
A	0	∞	∞	∞	∞	∞

Dijkstra's Algorithm



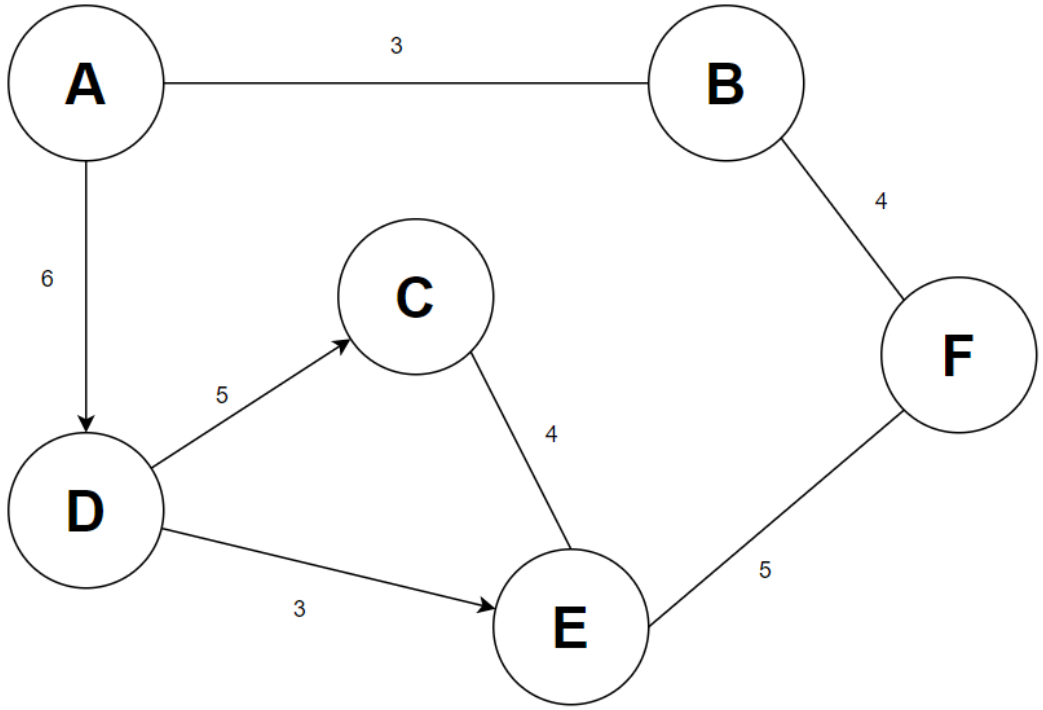
	A	B	C	D	E	F
A	0	∞	∞	∞	∞	∞
B		3	∞	6	∞	∞

Dijkstra's Algorithm



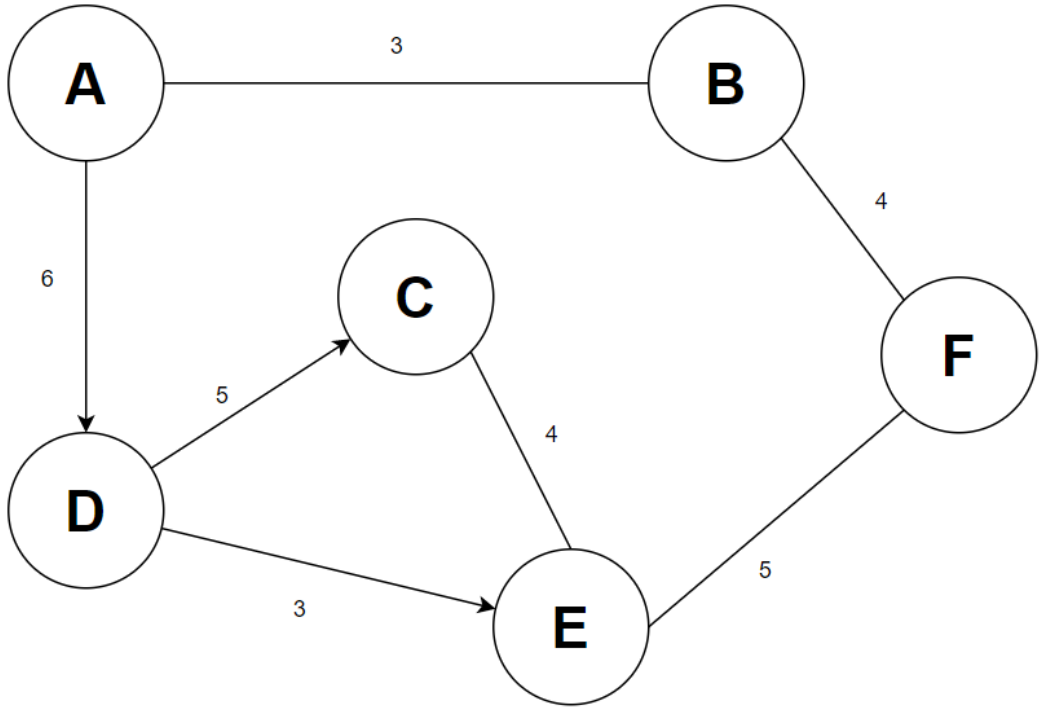
	A	B	C	D	E	F
A	0	∞	∞	∞	∞	∞
B		3	∞	6	∞	∞
D			∞	6	∞	7

Dijkstra's Algorithm



	A	B	C	D	E	F
A	0	∞	∞	∞	∞	∞
B		3	∞	6	∞	∞
D			∞	6	∞	7
F			11		9	7

Dijkstra's Algorithm



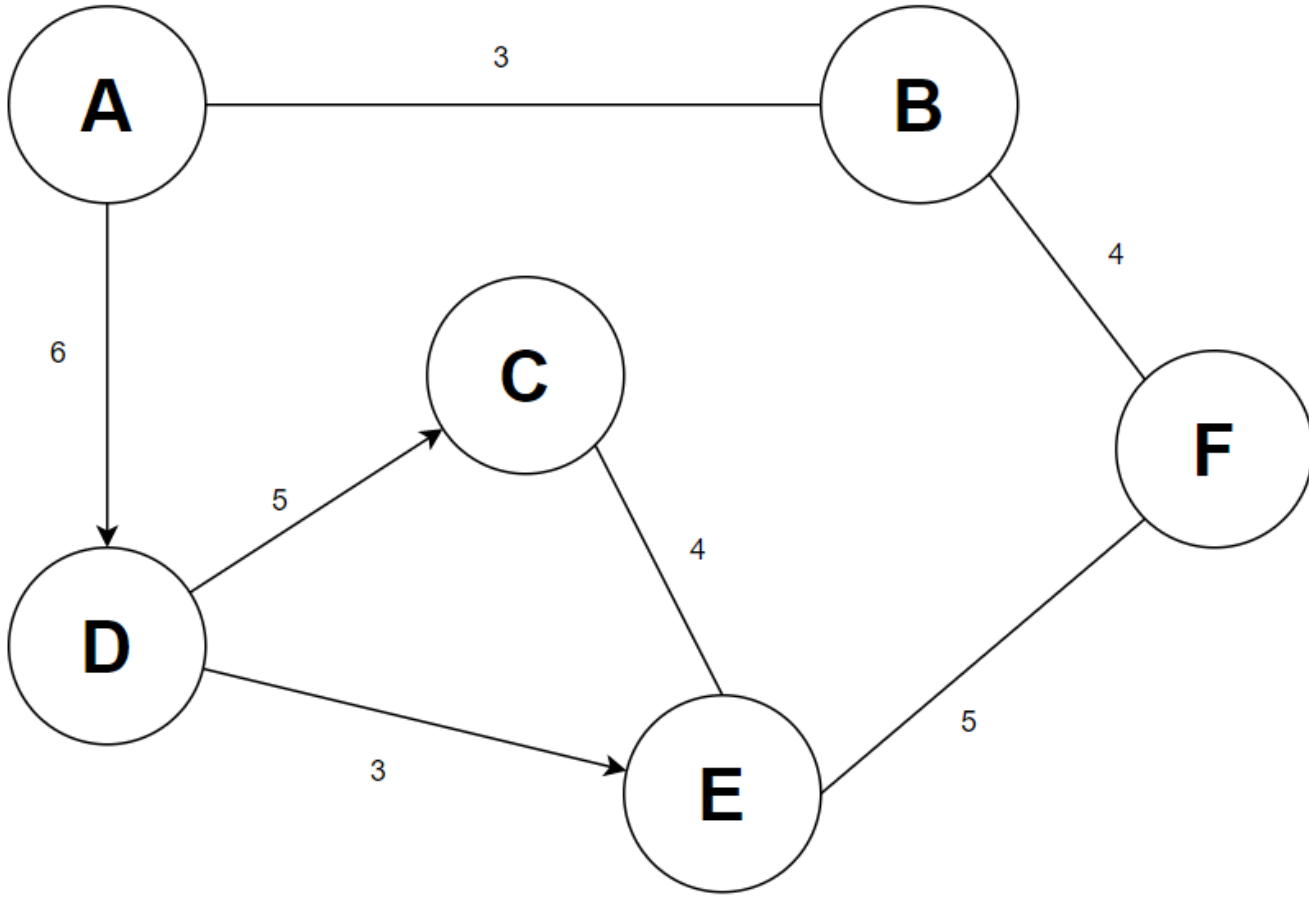
	A	B	C	D	E	F
A	0	∞	∞	∞	∞	∞
B		3	∞	6	∞	∞
D			∞	6	∞	7
F			11		9	7
E			11		9	
C			11			

Vertex Array

I	L	D	P	V
0	A	2147483647	-1	0
1	B	2147483647	-1	0
2	C	2147483647	-1	0
3	D	2147483647	-1	0
4	E	2147483647	-1	0
5	F	2147483647	-1	0

Adjacency Matrix

-1	3	-1	6	-1	-1
3	-1	-1	-1	-1	4
-1	-1	-1	-1	4	-1
-1	-1	5	-1	3	-1
-1	-1	4	-1	-1	5
-1	4	-1	-1	5	-1



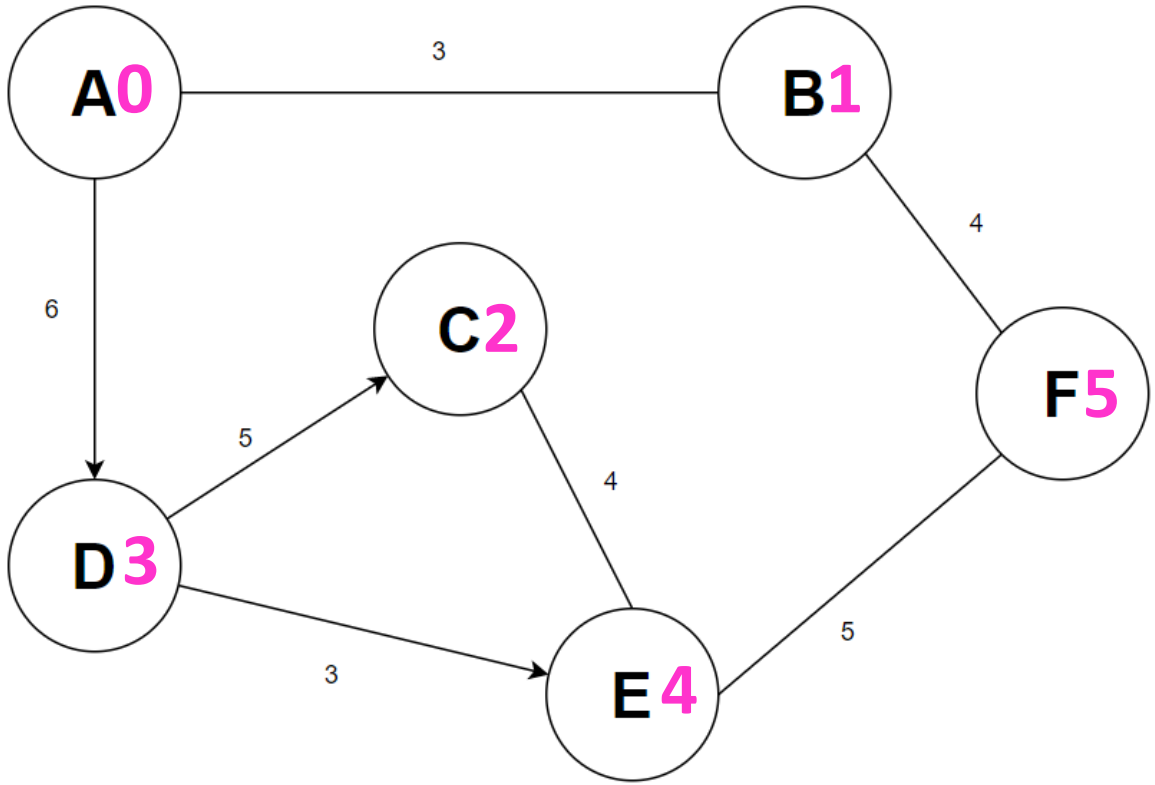
What is the starting vertex? **A**
What is the destination vertex? **E**
The path from A to E is A->D->E
and has a length of 9

Vertex Array

I	L	D	P	V
0	A	0	-1	1
1	B	3	0	1
2	C	11	3	1
3	D	6	0	1
4	E	9	3	1
5	F	7	1	1

Adjacency Matrix

	0	1	2	3	4	5
0	-1	3	-1	6	-1	-1
1	3	-1	-1	-1	-1	4
2	-1	-1	-1	-1	4	-1
3	-1	-1	5	-1	3	-1
4	-1	-1	4	-1	-1	5
5	-1	4	-1	-1	5	-1

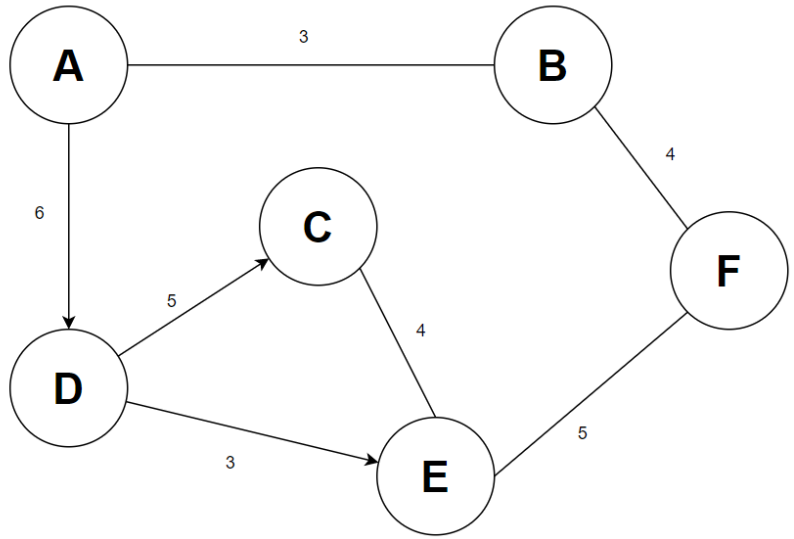


What is the starting vertex? **A**
What is the destination vertex? **E**
The path from A to E is A->D->E
and has a length of 9

```
int CurrentVertex = StartVertex;
VertexArray[StartVertex].distance = 0;
VertexArray[StartVertex].previous = -1;
VertexArray[StartVertex].visited = 1;
```

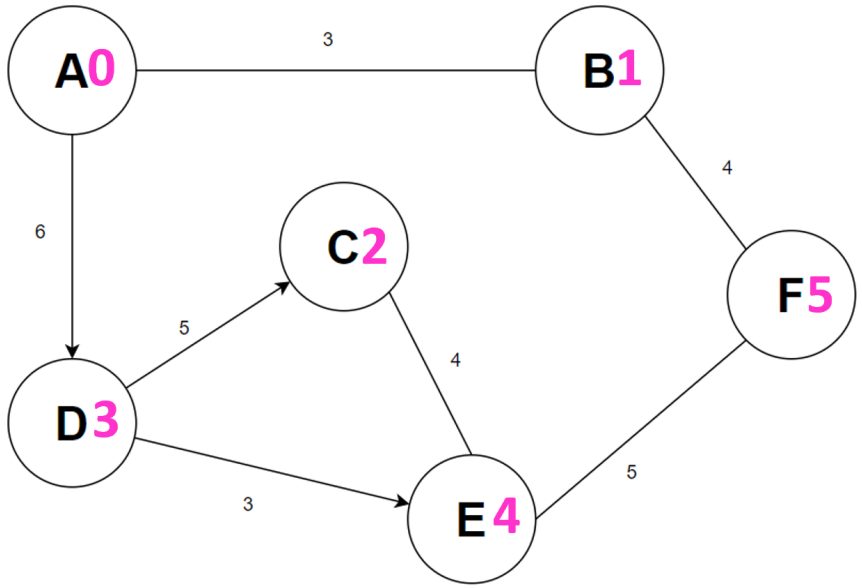
Setup starting Vertex

```
for (x = 0; x < VertexCount-1; x++)
{
    // relaxing edges
    // looking for smallest vertex
}
```



Starting at the next vertex in the Vertex Array, we need to visit every other vertex.

```
for(i = 0; i < VertexCount; i++)   // relaxing edges
{
   if (AdjMatrix[CurrentVertex][i] != -1 && !VertexArray[i].visited)
   {
      dofu = VertexArray[CurrentVertex].distance;
      cofuv = AdjMatrix[CurrentVertex][i];
      dofv = VertexArray[i].distance;

      if (dofu + cofuv < dofv)
      {
         dofv = dofu + cofuv;
         VertexArray[i].distance = dofv;
         VertexArray[i].previous = CurrentVertex;
      }
   }
```

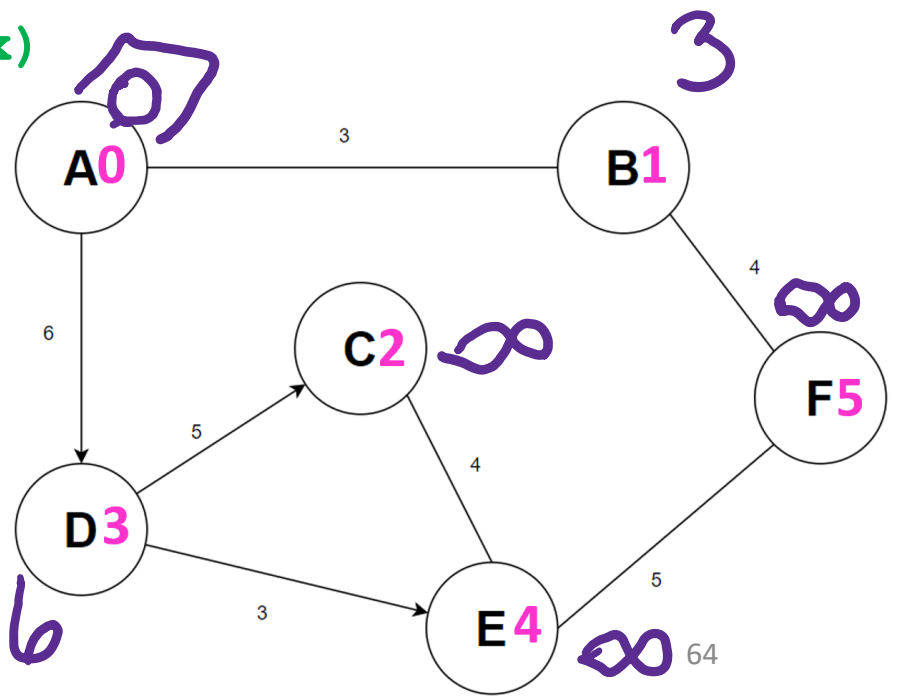


I	L	D	P	V
0	A	0	-1	1
1	B	2147483647	-1	0
2	C	2147483647	-1	0
3	D	2147483647	-1	0
4	E	2147483647	-1	0
5	F	2147483647	-1	0

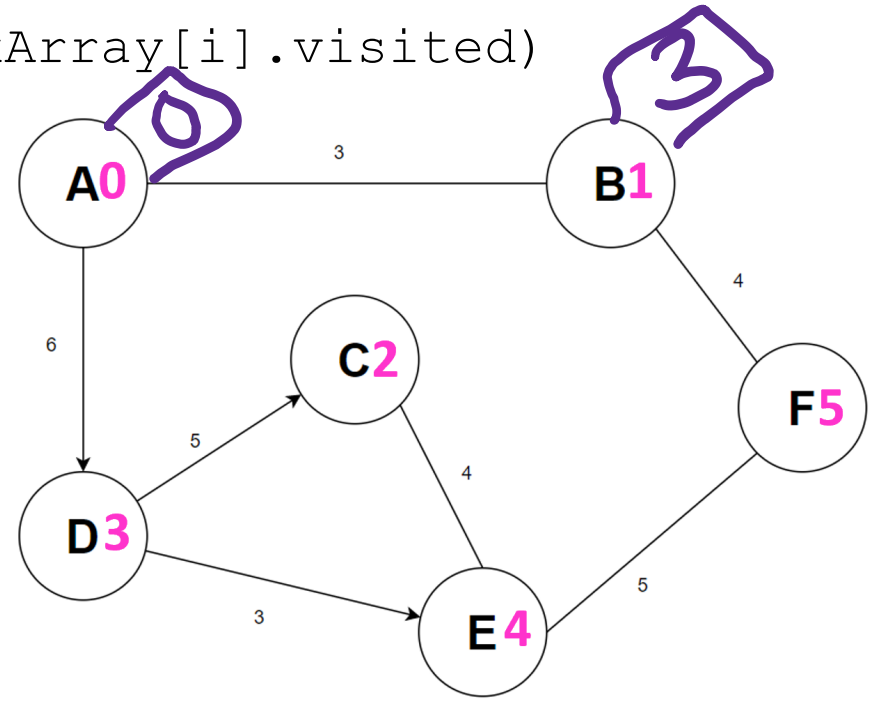
	0	1	2	3	4	5
0	-1	3	-1	6	-1	-1
1	3	-1	-1	-1	-1	4
2	-1	-1	-1	-1	4	-1
3	-1	-1	5	-1	3	-1
4	-1	-1	4	-1	-1	5
5	-1	4	-1	-1	5	-1

```
for(x = 0; x < VertexCount-1; x++)
{
   // relaxing the edges for loop

   SmallestVertexIndex = -1;
   int SmallestVertex = INT_MAX;

   for(i = 0; i < VertexCount; i++) // looking for smallest vertex
   {
      if (!VertexArray[i].visited)
      {
         if (VertexArray[i].distance < SmallestVertex)
         {
            SmallestVertex = VertexArray[i].distance;
            SmallestVertexIndex = i;
         }
      }
   }
   CurrentVertex = SmallestVertexIndex;
   VertexArray[CurrentVertex].visited = 1;
}
```

INT_MAX is a define in limits.h



```
for(i = 0; i < VertexCount; i++) // relaxing edges
{
   if (AdjMatrix[CurrentVertex][i] != -1 && !VertexArray[i].visited)
   {
      dofu = VertexArray[CurrentVertex].distance;
      cofuv = AdjMatrix[CurrentVertex][i];
      dofv = VertexArray[i].distance;

      if (dofu + cofuv < dofv)
      {
         dofv = dofu + cofuv;
         VertexArray[i].distance = dofv;
         VertexArray[i].previous = CurrentVertex;
      }
   }
}
```



I	L	D	P	V
0	A	0	-1	1
1	B	3	0	1
2	C	2147483647	-1	0
3	D	6	0	0
4	E	2147483647	-1	0
5	F	2147483647	-1	0

	0	1	2	3	4	5
0	-1	3	-1	6	-1	-1
1	3	-1	-1	-1	-1	4
2	-1	-1	-1	-1	4	-1
3	-1	-1	5	-1	3	-1
4	-1	-1	4	-1	-1	5
5	-1	4	-1	-1	5	-1

```
for(i = 0; i < VertexCount; i++) // relaxing edges
{
   if (AdjMatrix[CurrentVertex][i] != -1 && !VertexArray[i].visited)
   {
      dofu = VertexArray[CurrentVertex].distance;
      cofuv = AdjMatrix[CurrentVertex][i];
      dofv = VertexArray[i].distance;
      if (dofu + cofuv < dofv)
      {
         dofv = dofu + cofuv;
         VertexArray[i].distance = dofv;
         VertexArray[i].previous = CurrentVertex;
      }
   }
}
```


```
dofu = VertexArray[CurrentVertex].distance;
cofuv = AdjMatrix[CurrentVertex][i];
dofv = VertexArray[i].distance;
if (dofu + cofuv < dofv)
dofv = dofu + cofuv;
VertexArray[i].distance = dofv;
VertexArray[i].previous = CurrentVertex;
```



```
if (VertexArray[CurrentVertex].distance +
AdjMatrix[CurrentVertex][i] < VertexArray[i].distance)
{
    VertexArray[i].distance = VertexArray[CurrentVertex].distance +
                              AdjMatrix[CurrentVertex][i];
    VertexArray[i].previous = CurrentVertex;
}
```

Dijkstra's Algorithm

What is the run time of Dijkstra's Algorithm?

The big O of Dijkstra's Algorithm will depend on how the algorithm is implemented

Which data structures are used

and how those data structures are used.

Dijkstra's Algorithm

What is the run time of our implementation of Dijkstra's Algorithm?

```
for (x = 0; x < VertexCount-1; x++)
{
        for(i = 0; i < VertexCount; i++)
        {
                relaxing edges
        }
        for(i = 0; i < VertexCount; i++)
        {
                looking for smallest vertex
        }
}
```

$|V| - 1$

$|V|$

$|V|$

$(|V| - 1) * 2|V|$

$2|V^2| - 2|V|$

$O(V^2)$

Dijkstra's Algorithm

So our implementation using Vertex Array and Adjacency Matrix $O(V^2)$

Implementing Dijkstra using a min-priority queue $O(V^2)$

Implementing Dijkstra using a min-priority queue with binary min-heap $O(E\log_2 V)$

Implementing Dijkstra using a min-priority queue with Fibonacci heap $O(V\log_2 V+E)$

Coding Assignment 5

The Program

You are creating a program that can read a file of graph information and run Dijkstra's Algorithm on the graph and produce the path and the length/weight of the path between the starting Vertex and any other Vertex in the graph.

Let's say we start with this graph.

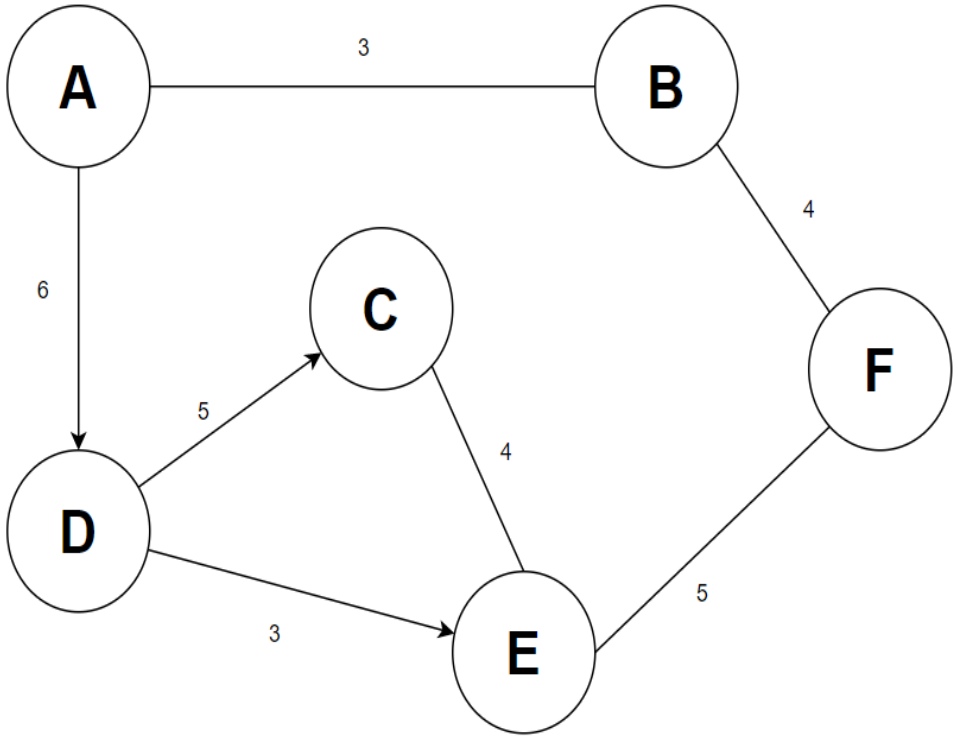
Coding Assignment 5

To translate graph to file, we'll first start by numbering the vertices.

The number we assign to a vertex will be that vertex's index number in the Vertex Array.

Just for convenience, I am going to start with A and number it 0 and go alphabetically from there. This is not a necessary order. You could start with Vertex E and 0 and mark Vertex C as 1, etc.

Just pick one to be index 0 and number from there.



Coding Assignment 5

Now we can translate this graph to our file format.

Our file format assumes that each line in the file represents a vertex and all of its edges and weights.

If the file has 5 lines, then the graph has 5 vertices.

The format of a line in the file is

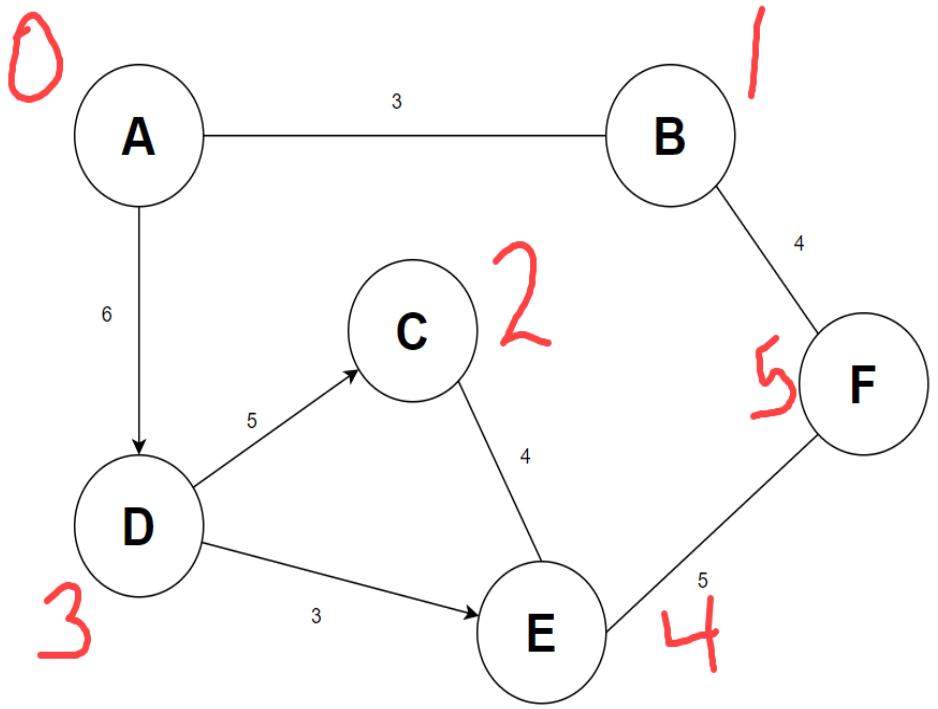
VertexLabel, Adjacent Vertex Index, Weight of Edge

where “Adjacent Vertex Index, Weight of Edge” can be repeated.

The first line in our file would be

```
A,1,3,3,6
```

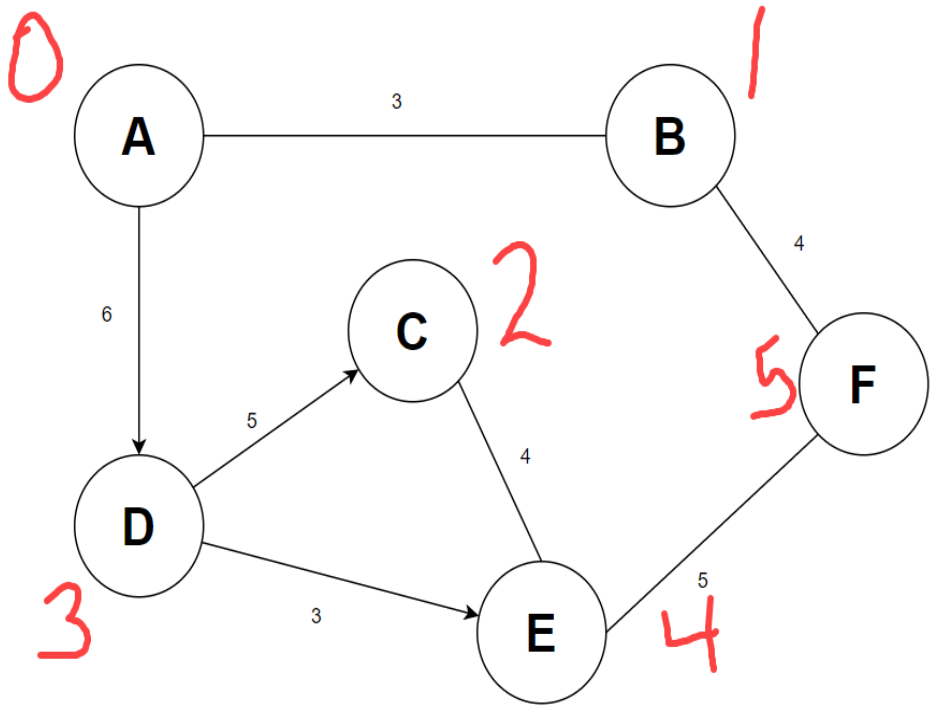
This shows that Vertex A (which is at index 0) has an edge between itself and Vertex B (which is at index 1) and that edge has a weight of 3. Vertex A also has an edge between itself and Vertex D (index 3) and that edge has a weight of 6.



Coding Assignment 5

The file for this graph will be

```
A,1,3,3,6
B,0,3,5,4
C,4,4
D,2,5,4,3
E,2,4,5,5
F,1,4,4,5
```



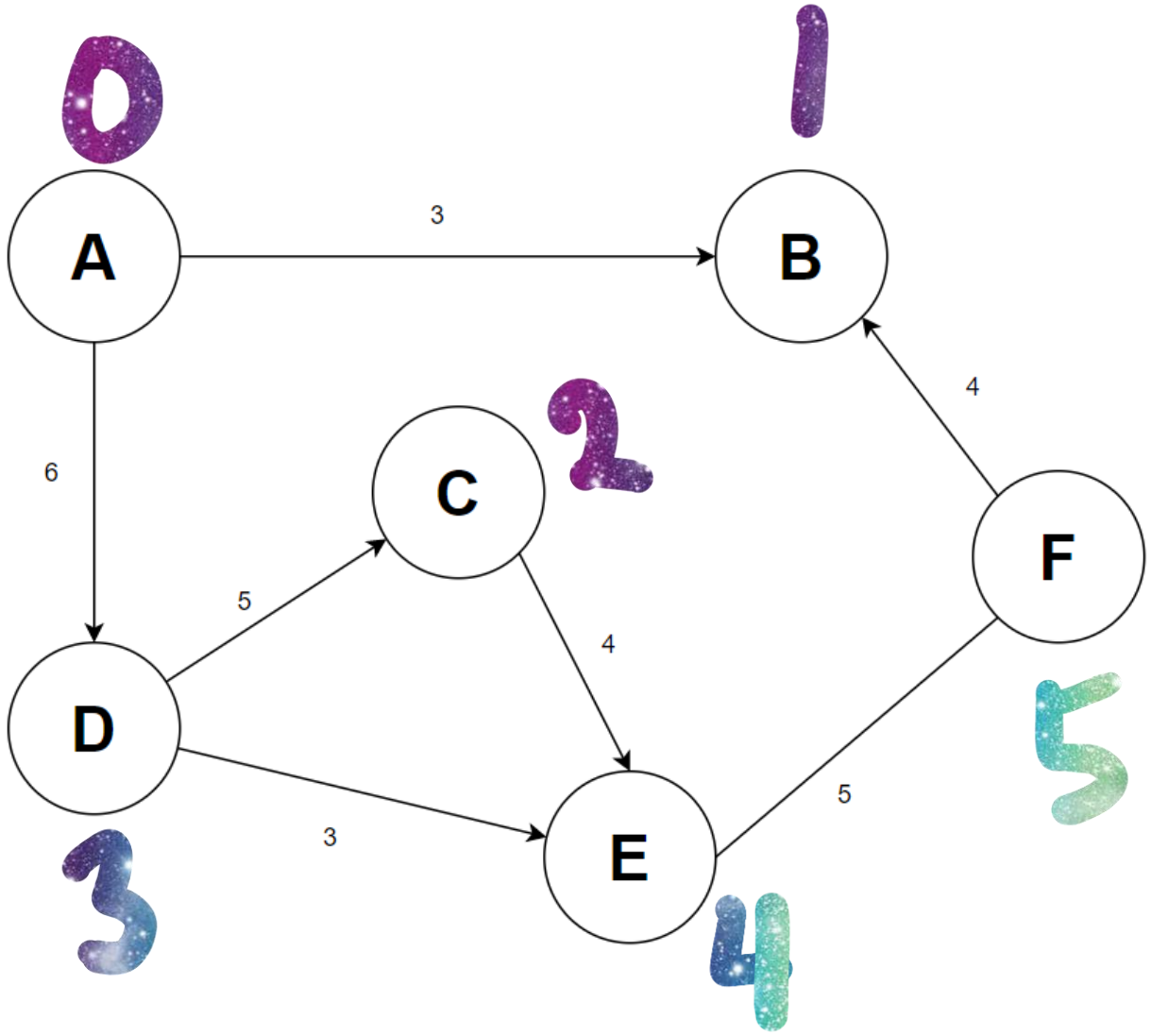
The ordering of the vertex,edge pairs does not matter – `0,3,5,4` is the same as `5,4,0,3`.

You need to create the file that represents your graph.
Your program will be tested with various files/graphs.

Coding Assignment 5

```
A,1,3,3,6
B,0,3,5,4
C,4,4
D,2,5,4,3
E,2,4,5,5
F,1,4,4,5
```

```
A,1,3,3,6
B
C,4,4
D,2,5,4,3
E,5,5
F,1,4,4,5
```



Coding Assignment 5

Getting Started

Create a define that will be set the to maximum size of your graph – the maximum number of vertices that your program will load and process.

Create your Vertex Array and your Adjacency Matrix using that define.

Initialize the Adjacency Matrix to -1.

See **File Handling** for details on how to populate the Vertex array and the Adjacency Matrix.

Prompt for the starting vertex. Your program should be able to use any vertex in the graph as the starting vertex.

Coding Assignment 5

```
typedef struct
{
    char label[6];
    int distance;
    int previous;
    int visited;
}
Vertex;
```

File Handling

As in the previous assignments, open the file from the command line using `argv[1]`.

While using `fgets`() to read through the file, use `strtok()` to parse each line into its parts.

Use the parts to add vertices to the Vertex Array and edges to the Adjacency Matrix.

Coding Assignment 5

```
A,1,3,3,6
B,0,3,5,4
C,4,4
D,2,5,4,3
E,2,4,5,5
F,1,4,4,5
```

You are required to use `strtok()` to parse your file. I have posted review materials on `strtok()`.

Inside your `fgets` `while` loop that is reading the file, you will do the first `strtok()` to get the label for the vertex.

```
A,1,3,3,6
B
C,4,4
D,2,5,4,3
E,5,5
F,1,4,4,5
```

Then, you will need a `while` loop to get as many of Adjacent Vertex Index, Weight of Edge pairs that are in a line in the file.

Please note the restrictions on `strtok()` in the rubric.

Coding Assignment 5

Conditional Compile

Add a conditional compile statement `PRINTIT` to print out the Adjacency Matrix after you have populated it (using values from the file). So, if your code is compiled as

```
gcc Code5_xxxxxxxxxx.c -D PRINTIT
```

then your adjacency matrix will be printed to the screen.

```
-1    3   -1    6   -1
 3   -1   -1   -1   -1
-1   -1   -1   -1    4
-1   -1    5   -1    3
-1   -1    4   -1   -1
```

```
#ifdef PRINTIT
printf("\n");
for(i = 0; i < VertexCount; i++)
{
    for(j = 0; j < VertexCount; j++)
        printf("%5d\t", AdjMatrix[i][j]);
    printf("\n");
}
#endif
```

Coding Assignment 5

Conditional Compile

The `PRINTIT` conditional compile should also be used to print out your Vertex Array after running Dijkstra.

```
#ifdef PRINTIT
printf("\n\nI\tL\tD\tP\tV\n");
for (i = 0; i < VertexCount; i++)
{
    printf("%d\t%s\t%d\t%d\t%d\n", i,
        VertexArray[i].label, VertexArray[i].distance,
        VertexArray[i].previous, VertexArray[i].visited);
}
printf("\n");
#endif
```

```
gcc Code5_xxxxxxxxxx.c -D PRINTIT
```

then your Vertex Array will be printed to the screen.

```
I       L       D       P       V
0       A       0       -1      1
1       B       3       0       1
2       C       11      3       1
3       D       6       0       1
4       E       9       3       1
```

I = **I**ndex, L = **L**abel, D = **D**istance, **P** = Previous, **V** = Visited

This feature will be used to grade your program. The printing should be controlled with the compiler directive (not by commenting or uncommenting code). The output of your adjacency matrix and vertex array **must** have these formats.

Coding Assignment 5

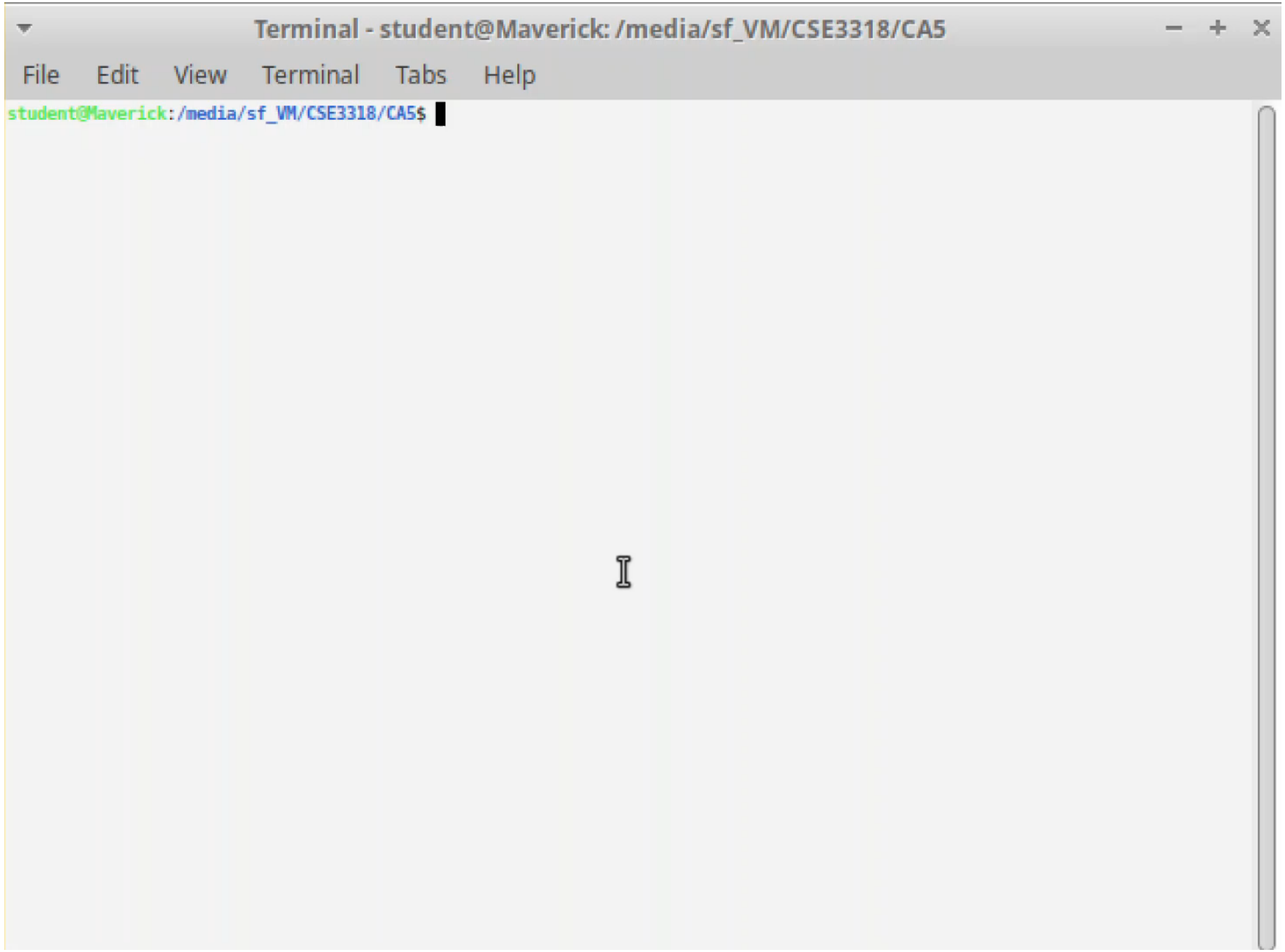
Dijkstra's Algorithm

I suggest you use the Dijkstra code shown in class.

It closely mirrors the manual method we learned.

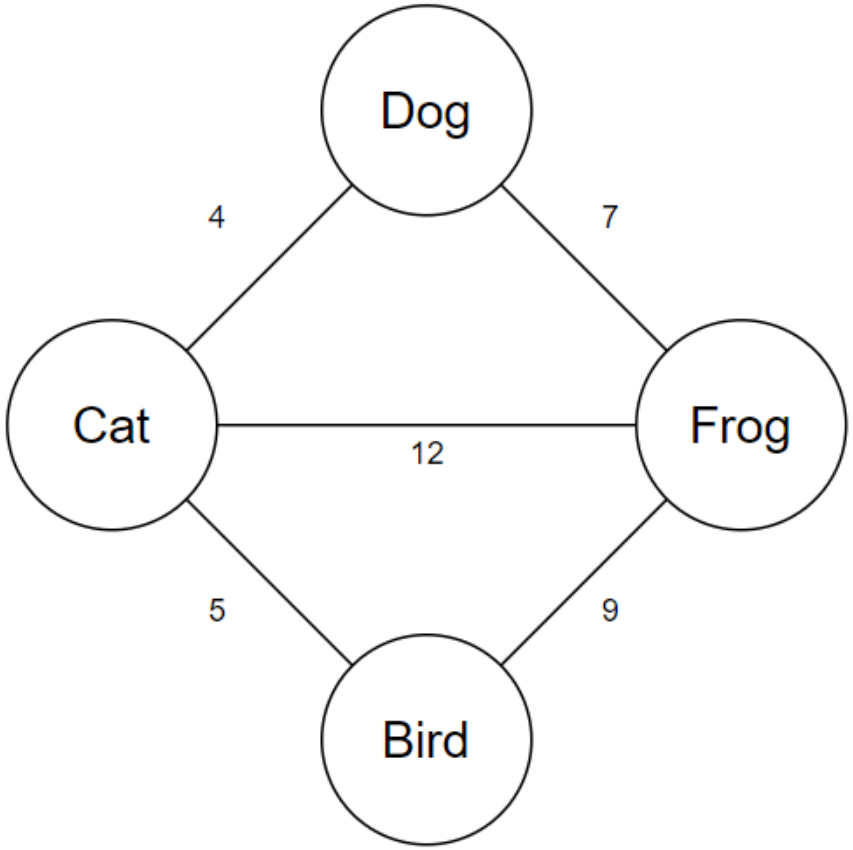
If you so choose to use the Internet as your source, be sure you are getting the C version of Dijkstra (and not some other graph traversal/variation of Dijkstra).

Your Dijkstra code, regardless of where you source it, **MUST** use an adjacency matrix and a vertex array so that you can create the required output for grading.


Terminal - student@Maverick: /media/sf_VM/CSE3318/CA5
File Edit View Terminal Tabs Help
student@Maverick:/media/sf_VM/CSE3318/CA5$

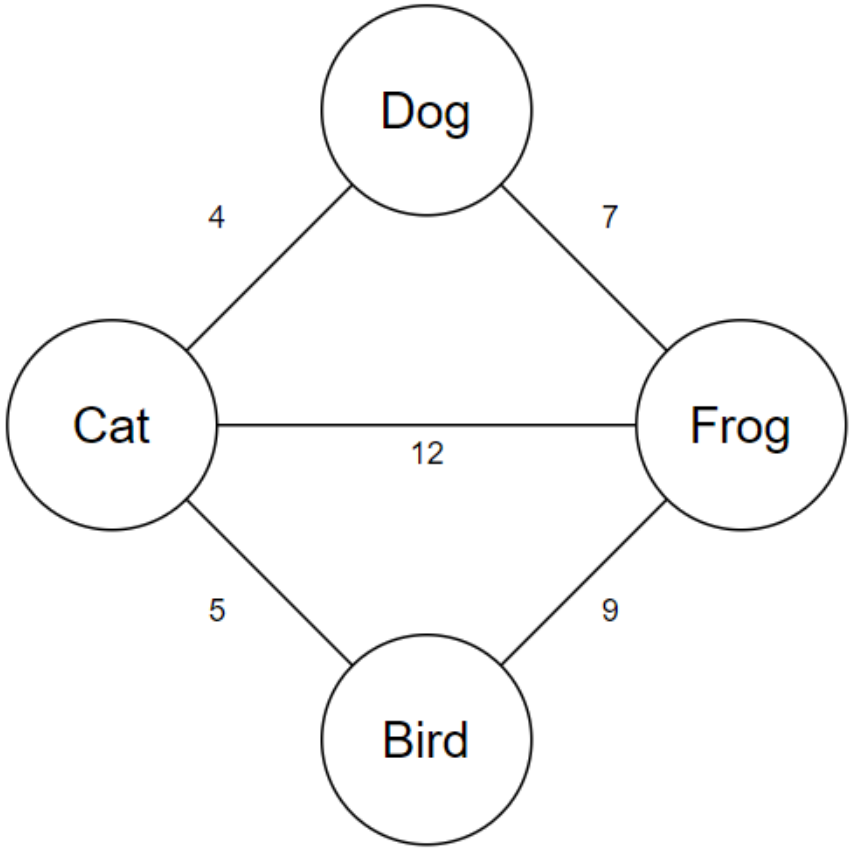
Coding Assignment 5

```
typedef struct
{
    char label[6];
    int distance;
    int previous;
    int visited;
}
Vertex;
```



Coding Assignment 5

```
Dog,1,4,2,7
Cat,0,4,3,5,2,12
Frog,0,7,3,9,1,12
Bird,1,5,2,9
```



Coding Assignment 5

What is the starting vertex? Cat

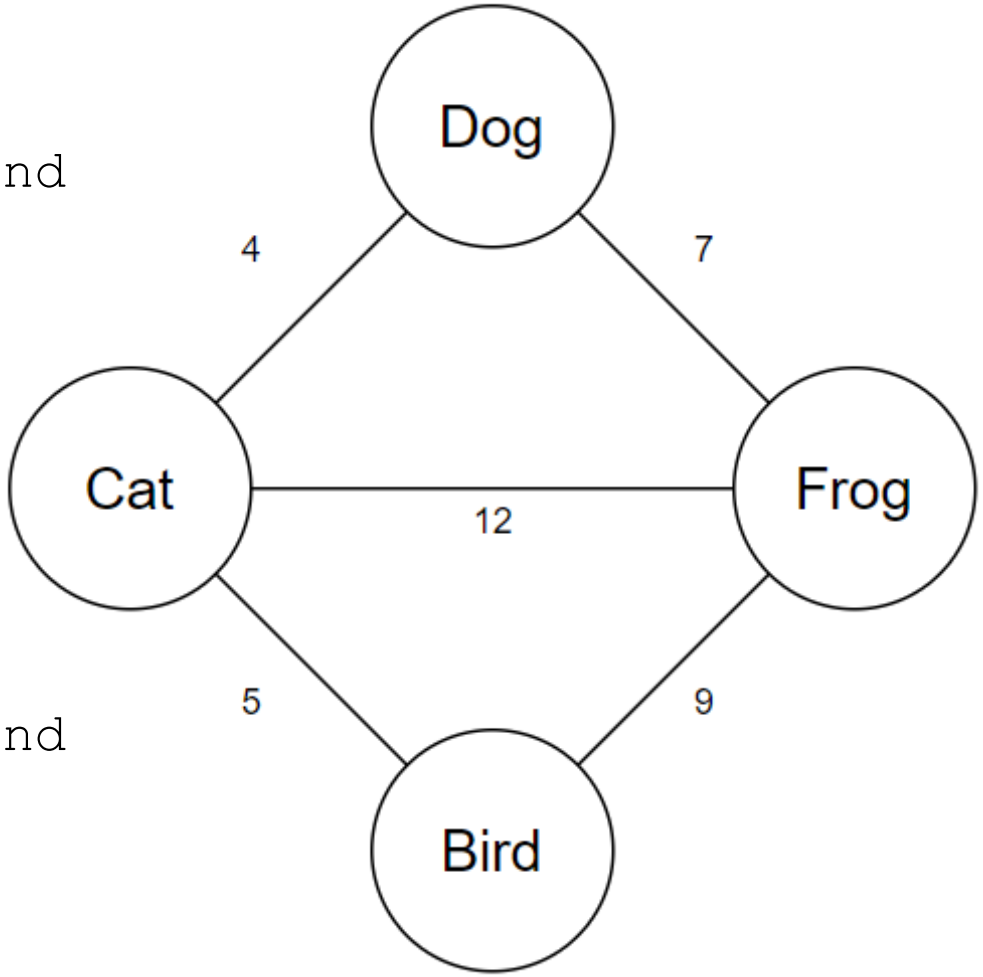
What is the destination vertex? Frog

The path from Cat to Frog is Cat->Dog->Frog and has a length of 11

What is the starting vertex? Dog

What is the destination vertex? Bird

The path from Dog to Bird is Dog->Cat->Bird and has a length of 9



Coding Assignment 5

IMPORTANT

Make NO assumptions about the ordering or the labeling of the vertices in the graph.

The `VertexLabel` could be up to 5 characters and the vertices can be listed in any order in the file.

Coding Assignment 5

You will also need to submit a drawing of your graph.

Draw the graph by hand or use a program.

However you create the graph, you must submit an electronic picture version of it. Make a PDF version, take a picture with your phone or take a screenshot – doesn’t matter how you do it.

Your submission must be something that can be viewed without a specialized app – use PDF, JPEG or PNG.

Coding Assignment 5

Your graph should have a minimum of 5 vertexes.

Your graph should have a mix of directed and undirected edges.

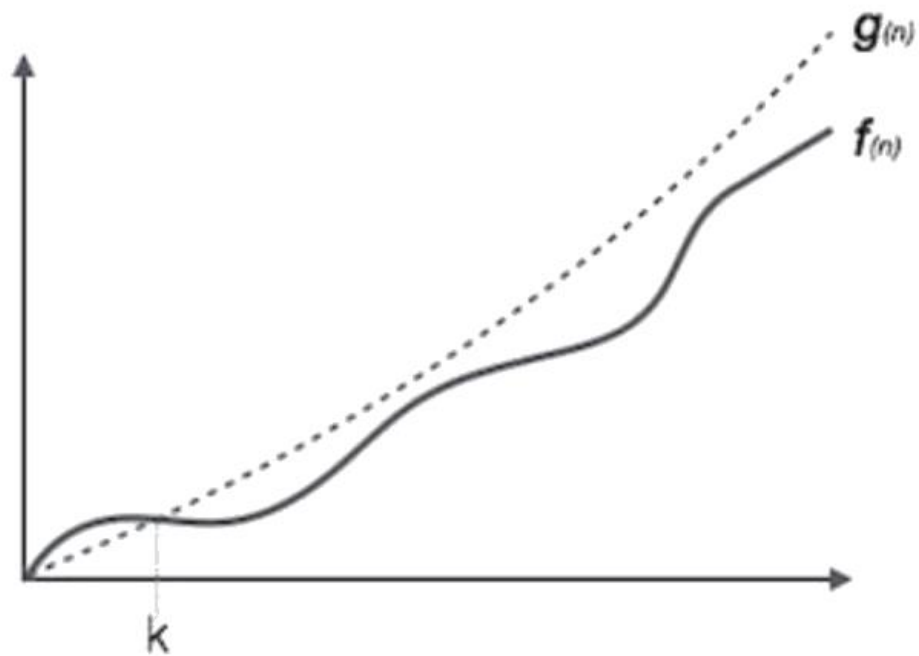
Your edges must ALL be weighted.

Make sure every vertex can be reached from every other vertex.

```
A,1,3,3,6
B,0,3,5,4
```

Big Oh Notation, O

The notation O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.

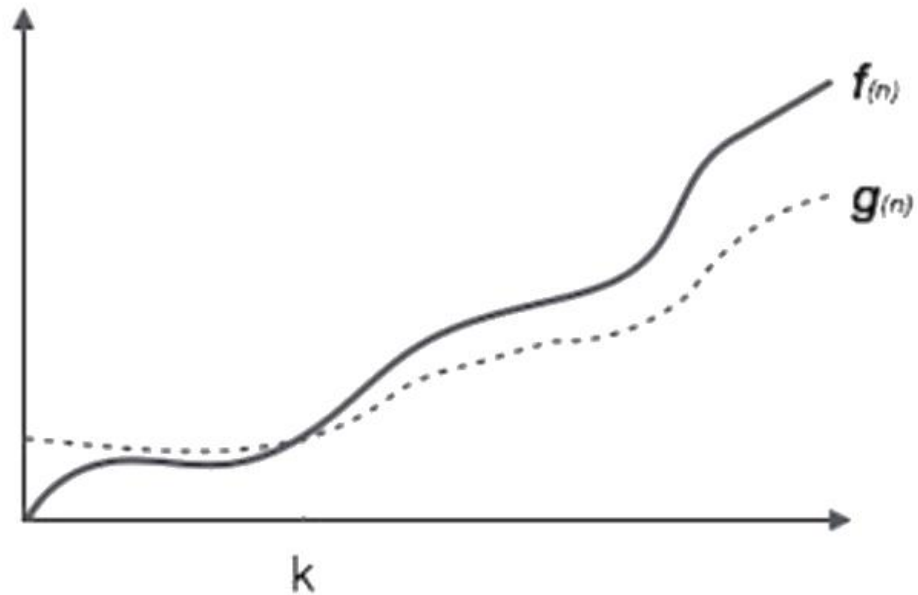


For example, for a function ***f*(n)**

$O(f(n)) = \{ g(n) : \text{there exists } c > 0 \text{ and } n_0 \text{ such that } f(n) \leq c.g(n) \text{ for all } n > n_0. \}$

Omega Notation, Ω

The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

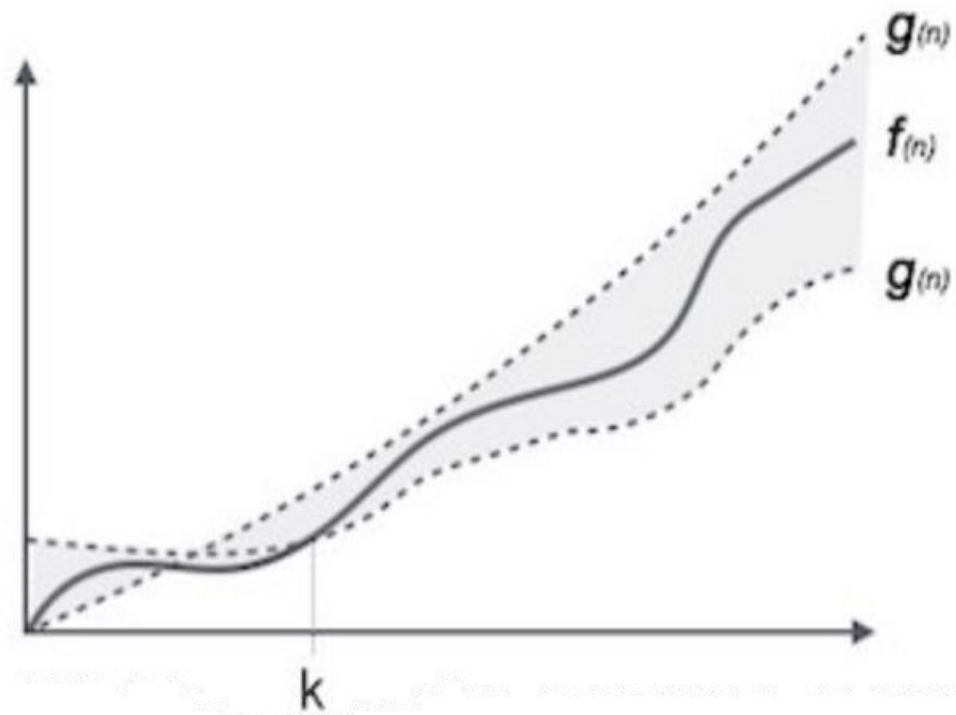


For example, for a function ***f*(n)**

$\Omega(f(n)) \geq \{ g(n) : \text{there exists } c > 0 \text{ and } n_0 \text{ such that } g(n) \leq c.f(n) \text{ for all } n > n_0. \}$

Theta Notation, θ

The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time. It is represented as follows −



$\theta(f(n)) = \{ g(n)$ if and only if $g(n) = O(f(n))$ and $g(n) = \Omega(f(n))$ for all $n > n_0$. $\}$

Common Asymptotic Notations

Following is a list of some common asymptotic notations −

constant	–	O(1)
logarithmic	–	O(log n)
linear	–	O(n)
n log n	–	O(n log n)
quadratic	–	$O(n^2)$
cubic	–	$O(n^3)$

What does it mean when we say that an algorithm X is asymptotically more efficient than Y? Which of these statements is true?

What is the time complexity of this code segment?

```
int i, j, k = 0;
for (i = n / 2; i < n; i++)
{
      for (j = 2; j <= n; j = j * 2)
      {
            k = k + n / 2;
      }
}
```

If *n* is 16,

i = n/2 = 16/2 = 8 i < 16	j = 2 j <= 16 j = j*2	k = k + n/2
8	2, 4, 8, 16	k = doesn't matter
9	2, 4, 8, 16	
10	2, 4, 8, 16	
11	2, 4, 8, 16	
12	2, 4, 8, 16	
13	2, 4, 8, 16	
14	2, 4, 8, 16	
15	2, 4, 8, 16	

```
int i, j, k = 0;
for (i = n / 2; i < n; i++)
{
        for (j = 2; j <= n; j = j * 2)
        {
                k = k + n / 2;
        }
}
```

then 8 passes ($\frac{n}{2}$) through i loop and 4 passes through j loop for each pass through i loop (n = 16 gives 4 passes).

What if n is 32?

i = n/2 = 32/2 = 16 i < 32	j = 2 j <= 32 j = j*2	k = k + n/2
16	2, 4, 8, 16, 32	k = doesn't matter
17	2, 4, 8, 16, 32	
18	2, 4, 8, 16, 32	
19	2, 4, 8, 16, 32	
20	2, 4, 8, 16, 32	
21	2, 4, 8, 16, 32	
22	2, 4, 8, 16, 32	
23	2, 4, 8, 16, 32	
24	2, 4, 8, 16, 32	
25	2, 4, 8, 16, 32	
26	2, 4, 8, 16, 32	
27	2, 4, 8, 16, 32	
28	2, 4, 8, 16, 32	
29	2, 4, 8, 16, 32	
30	2, 4, 8, 16, 32	
31	2, 4, 8, 16, 32	

```
int i, j, k = 0;
for (i = n / 2; i < n; i++)
{
        for (j = 2; j <= n; j = j * 2)
        {
                k = k + n / 2;
        }
}
```

then 16 passes ($\frac{n}{2}$) through i loop and 5 passes through j loop for each pass through i loop (n = 32 gives 5 passes).

What if n is 8?

i = n/2 = 8/2 = 4 i < 8	j = 2 j <= 8 j = j*2	k = k + n/2
4	2, 4, 8	k = doesn't matter
5	2, 4, 8	
6	2, 4, 8	
7	2, 4, 8	

```
int i, j, k = 0;
for (i = n / 2; i < n; i++)
{
    for (j = 2; j <= n; j = j * 2)
    {
        k = k + n / 2;
    }
}
```

then 4 passes ($\frac{n}{2}$) through i loop and 3 passes through j loop for each pass through i loop (n = 8 gives 3 passes).

For the outer i loop, we always had $\frac{n}{2}$ passes.

For the inner j loop, we had

n of 8 gave 3 passes

n of 16 gave 4 passes

n of 32 gave 5 passes

n	$\log_2 n$
1	0
2	1
4	2
8	3
16	4
32	5
64	6

This tell us that the inner j loop will have $\log_2$n passes.

For the complete run time, we have to multiple the run time of the inner loop by the iterations of the outer loop

$\frac{n}{2}$ * $\log_2$n = $\frac{1}{2}n$ * $\log_2 n$ = $O(n\log_2 n)$

Using big *O* notation allows us to drop the $\frac{1}{2}$ from the function.

What is the time complexity of this code segment?

```
int a = 0, i = n;  // n is the number of items in the array
while (i > 0)
{
   a += i;
   i /= 2;
}
```

Pick one

1. $O(n)$
2. $O(\sqrt{n})$
3. $O(\frac{n}{2})$
4. $O(\log_2 n)$

```
int a = 0, i = n;  // n is the number of items in the array
while (i > 0)
{
    a += i;
    i /= 2;
}
```

The pattern here is

32 -> 6

16 -> 5

8 -> 4

4 -> 3

which looks almost familiar.

If we think of this pattern like this…

32 -> 6 -> 5+1

16 -> 5 -> 4+1

8 -> 4 -> 3+1

4 -> 3 -> 2+1

This looks like $\log_2 n + 1$

We can drop the +1 for asymptotic notation so we can call this $O(\log_2 n)$.

n	$\log_2 n$
1	0
2	1
4	2
8	3
16	4
32	5
64	6

What is the time complexity of this code segment?

```
int a = 0;
for (i = 0; i < n; i++)     // n is the number of elements in the array
{
   for (j = n; j > i; j--)
   {
      a = a + i + j;
   }
}
```

Pick one

1. $O(n)$
2. $O(n\log(n))$
3. $O(n\sqrt{n})$
4. $O(n^2)$

n = 32

i = 0; i < 32; i++	j = 32; j > i; j--
0	j loops from 32 to 0 => 32 times
1	j loops from 32 to 1 => 31 times
2	j loops from 32 to 2 => 30 times
3	j loops from 32 to 3 => 29 times
…	…
29	j loops from 32 to 29 => 3 times
30	j loops from 32 to 30 => 2 times
31	j loops from 32 to 31 => 1 time
loop condition fails at i = 32	

```
int a = 0;
for (i = 0; i < n; i++)
{
    for (j = n; j > i; j--)
    {
        a = a + i + j;
    }
}
```

$\frac{n^2+n}{2}$ describes the inner loop

The outer loop loops n times

$$\frac{n^2+n}{2} + n =$$

$$\frac{1}{2}n^2 + \frac{1}{2}n + n =$$

We could do n = 16 or other values but we'll see the same pattern.

This pattern is an arithmetic series which is described by $\frac{n^2+n}{2}$

$O(n^2)$

$$\frac{n^2 + n}{2}$$

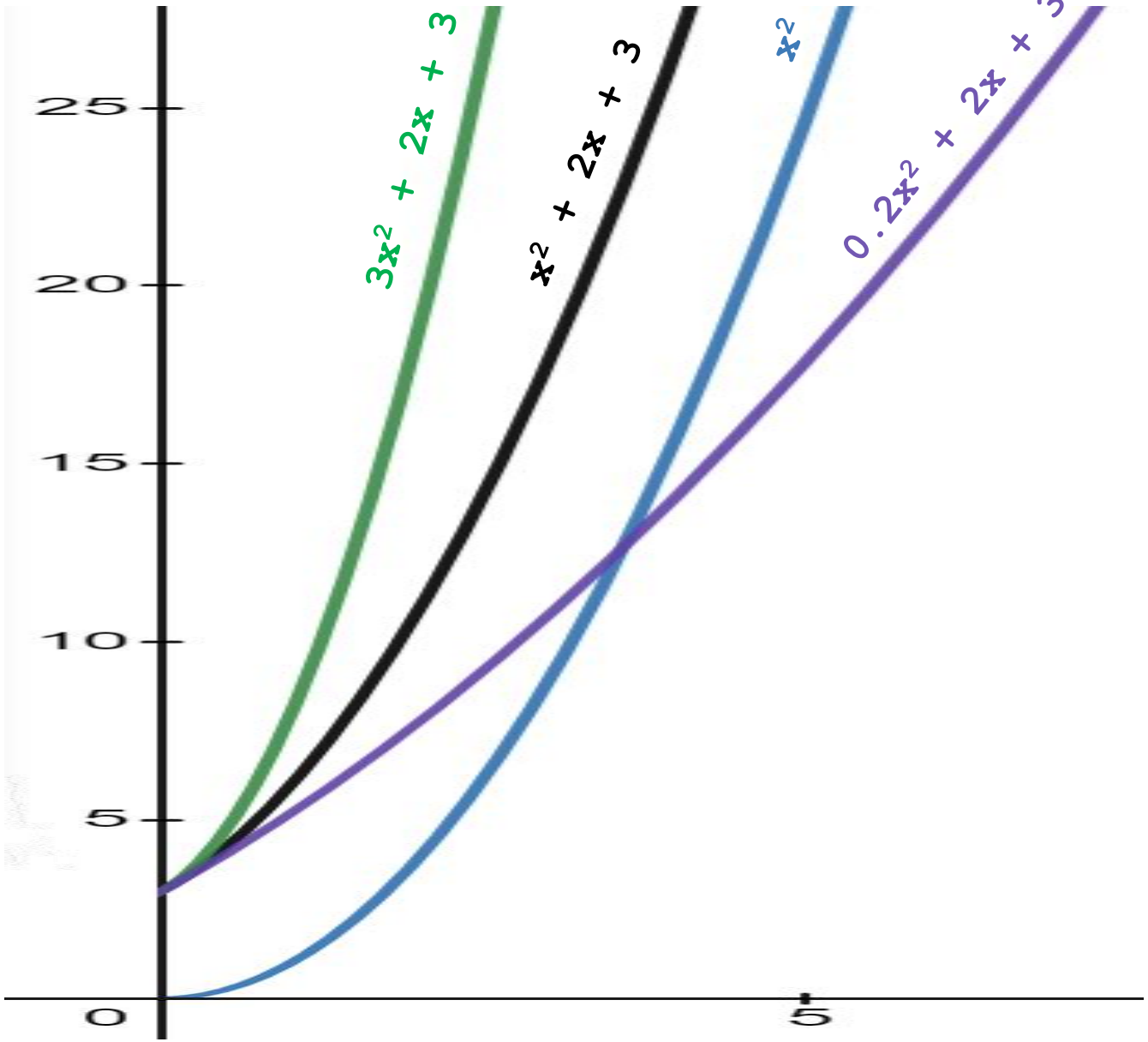
```
./a.out
Enter a value for n 32
loop counter = 528

./a.out
Enter a value for n 16
loop counter = 136

./a.out
Enter a value for n 8
loop counter = 36

./a.out
Enter a value for n 4
loop counter = 10
```


```
int a = 0, counter = 0;
for (i = 0; i < n; i++)
{
    for (j = n; j > i; j--)
    {
        a = a + i + j;
        ++counter;
    }
}

printf("loop counter =
%d", counter);
```

Coding Assignment 5

- The prompt for starting vertex is for the label – not for the index.
- You will need to add code to translate the label to an index.
  - Go back to the BFS code.

- The BFS code shows how to create the path from start to end by going backwards through the path – starting at the end and tracing the path back to the start.
- You will need something **similar** for this coding assignment.
  - The labels in the BFS were single chars – in your assignment, the labels are strings



$3x^2 + 2x + 3$

$x^2 + 2x + 3$

x^2

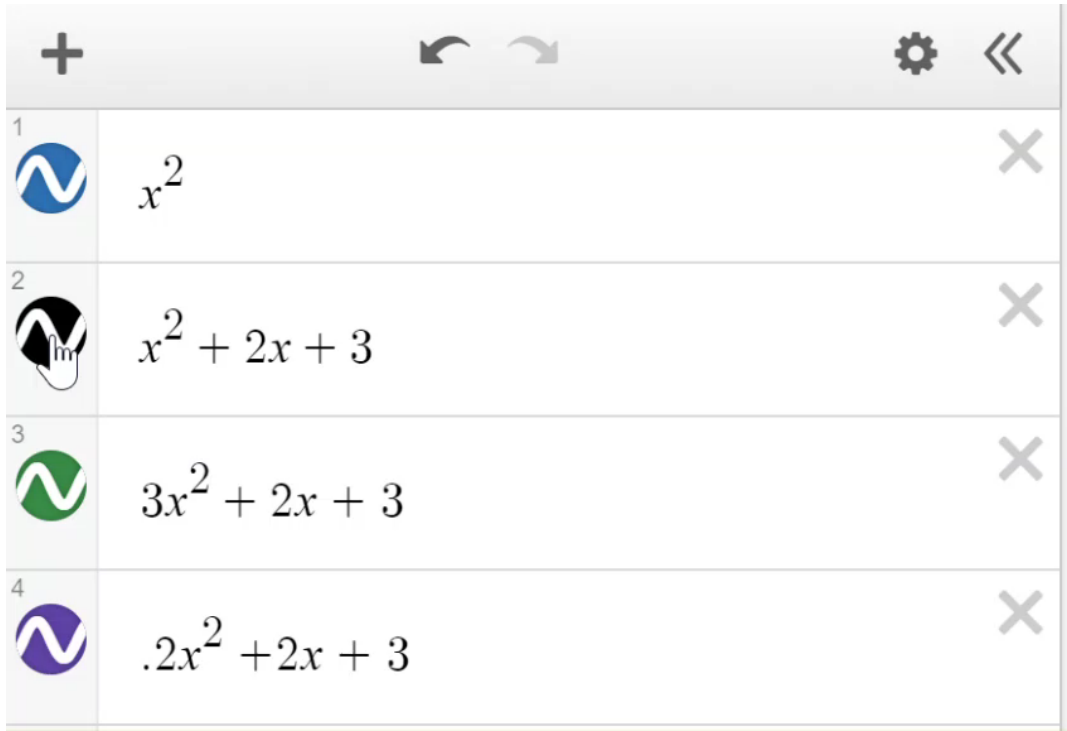
$0.2x^2 + 2x + 3$

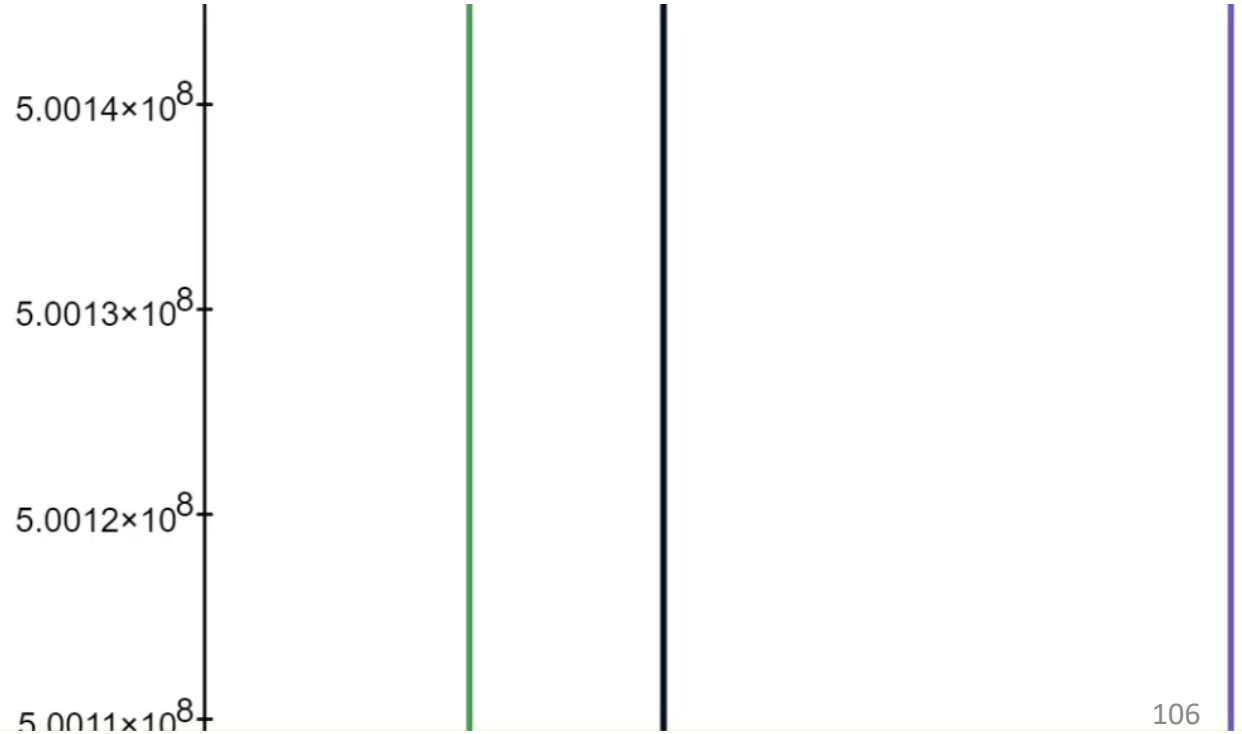
$3x^2 + 2x + 3$

$x^2 + 2x + 3$

$0.2x^2 + 2x + 3$

x^2



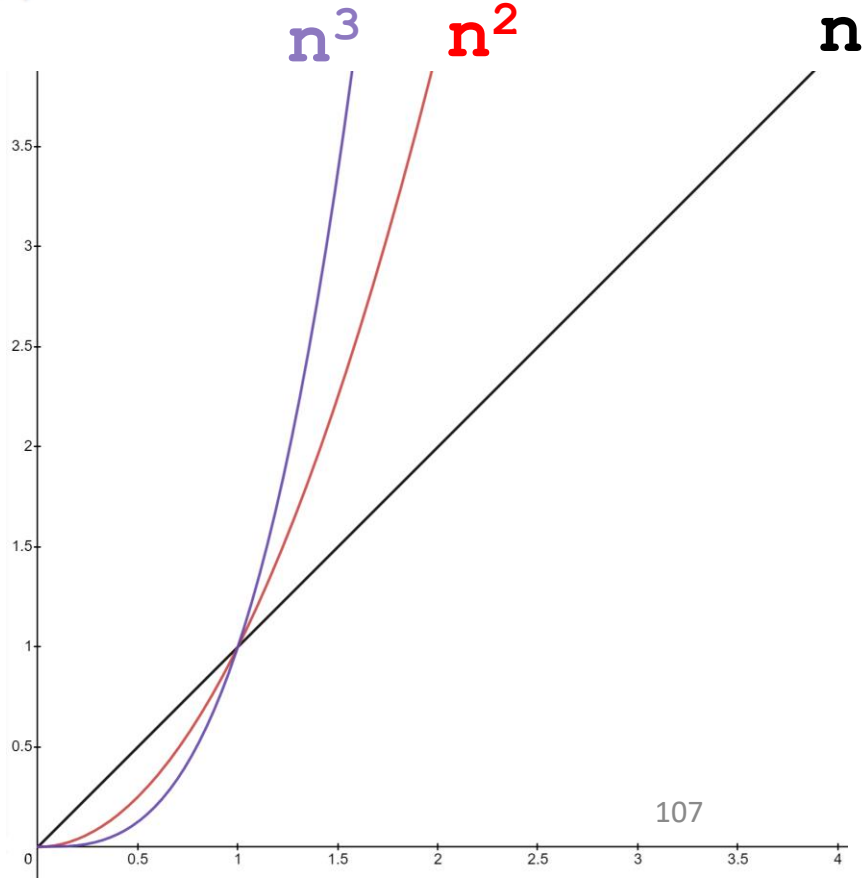


If the running time is $\Theta(f(n))$ in a particular situation, then it's also O(f(n))

Let $T(n) = n^2 + 100n + 300000$

Which of the following statements are true?

- ☑ $T(n) = \Omega(n)$
- ☑ $T(n) = O(n^3)$
- ☑ $T(n) = O(n^2)$
- ☑ $T(n) = \Theta(n^2)$



$\Theta(n\log_2 n)$ describes the run time of (check all that apply)

- [ ] Quick Sort worst case
- [x] Merge Sort
- [ ] Insertion Sort
- [x] Quick Sort best case
- [ ] Selection Sort

$\Theta(n\log_2 n)$ describes the run time of (check all that apply)

- [x] Quick Sort best case
- [x] Quick Sort average case
- [ ] Quick Sort worst case

$\Theta(n^2)$ describes the run time of (check all that apply)

- [x] Quick Sort worst case
- [x] Selection Sort
- [ ] Quick Sort best case
- [ ] Merge Sort
- [x] Insertion Sort

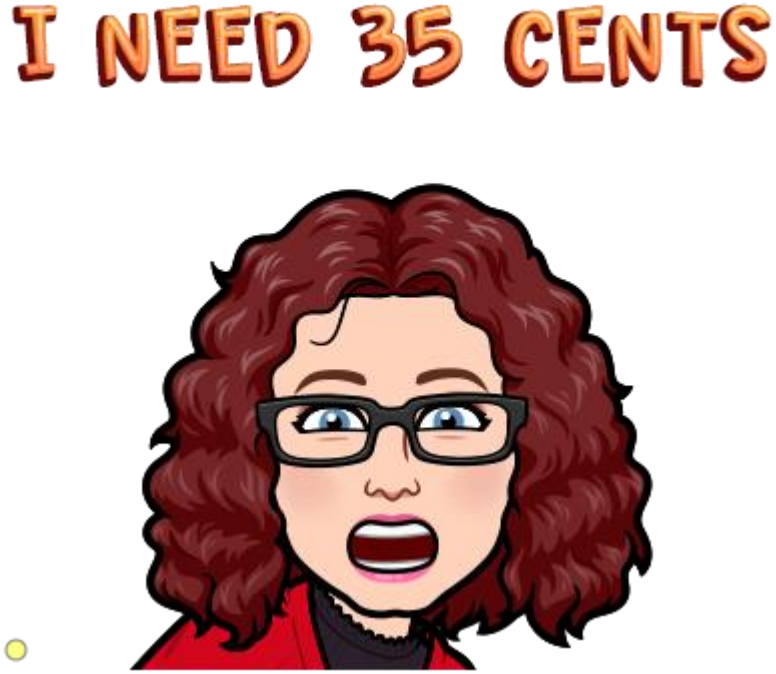
Greedy Algorithms

Let's start with a pile of US pennies (1¢) and a pile of US nickels (5¢) and a pile of US dimes (10¢)

Greedy Algorithms

If I need 35¢, how many different ways could you give me 35¢ from these piles?

35 pennies
30 pennies and 1 nickel
25 pennies and 2 nickels
25 pennies and 1 dime
20 pennies and 3 nickels
20 pennies and 1 nickel and 1 dime
15 pennies and 4 nickels
15 pennies and 2 dimes



Greedy Algorithms

What if I specifically asked for the least number of coins that will sum up to 35¢?

Your biggest coin is 10¢, so you would give me a dime first.

35 – 10 means I still need 25¢ so you would give me another dime

25 – 10 means I still need 15¢ so you would give me another dime

15 – 10 means I still need 5¢ so you would me a nickel


Greedy Algorithms


How would the coins you give me differ if you also had quarters (25¢)?

You would give me the largest coin first – a quarter.

35 – 25 means I still need 10¢ so you would give me dime

Our final set of coins is different for the same 35¢ but we were using a different set of coins at the start.


Greedy Algorithms


Let's say that you also had half dollars (50¢ coins).

Would that change your answer of

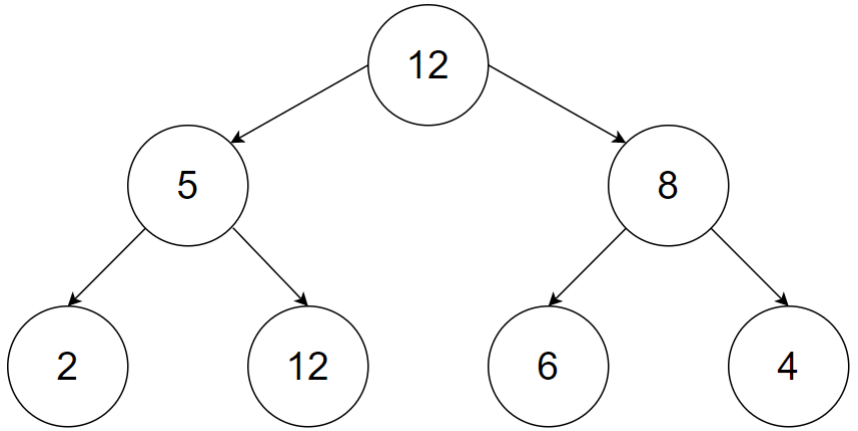
No, because 50¢ is larger than my goal of 35¢.

Greedy Algorithms

In this example, we were able to arrive at the least number of coins by using the algorithm of always choosing the largest denomination that was less than our goal.

Let's apply this same algorithm to another situation.

I want to find the path through the tree that gives me largest final sum.



Greedy Algorithms

"always choosing the largest denomination that was less than our goal"

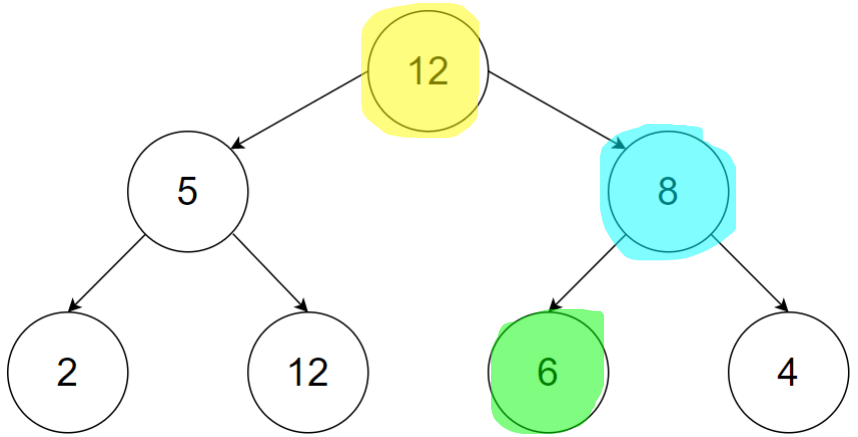
In this situation, we are going to choose the node with the largest value every time we have a choice. Our goal is to get to the largest sum.

Starting from 12, we would choose

8 over 5 because 8 > 5.

From 8, we would pick 6 over 4

because 6 > 4



Greedy Algorithms

Did we succeed at our goal?

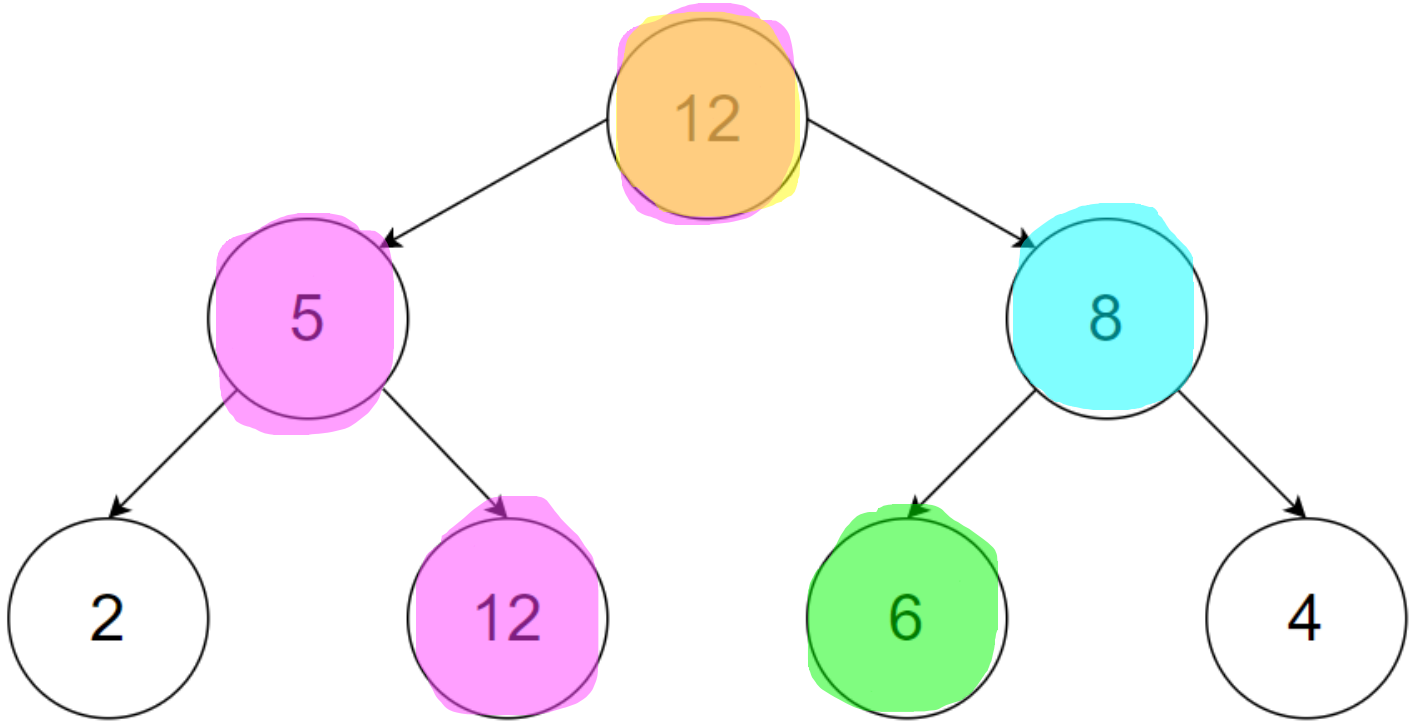
Did we find the largest sum possible?

No. We should have picked

12 +

5 +

12



Greedy Algorithms

A greedy algorithm is an algorithmic paradigm that follows the problem solving approach of making the locally optimal choice at each stage with the hope of finding a global optimum.

Pros

simple, easy to implement, runs fast

Cons

very often they don't provide a globally optimum solution

Greedy Algorithms

So when do we use a greedy algorithm since they don't always work?

Greedy algorithms succeed when the problem has two properties

1. Greedy choice property

    A global optimum can be arrived at by selecting a local optimum

2. Optimal Substructure

    An optimal solution to the problem contains an optimal solution to subproblems.

Greedy Algorithms

Greedy choice property

We can make whatever choice seems best at the moment and then solve the subproblems that arise later.

The choice made by a greedy algorithm may depend on choices made so far, but not on future choices or all the solutions to the subproblem.

It iteratively makes one greedy choice after another, reducing each given problem into a smaller one.

Greedy Algorithms

Greedy choice property

A greedy algorithm never reconsiders its choices.

This is the main difference from dynamic programming, which is exhaustive and is guaranteed to find the solution.

After every stage, dynamic programming makes decisions based on all the decisions made in the previous stage and may reconsider the previous stage's algorithmic path to solution.

Greedy Algorithms

For many other problems, greedy algorithms fail to produce the optimal solution and may even produce the unique worst possible solution.

One example is the traveling salesman problem..

The traveling salesman problem (TSP) asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?"

Greedy Algorithms

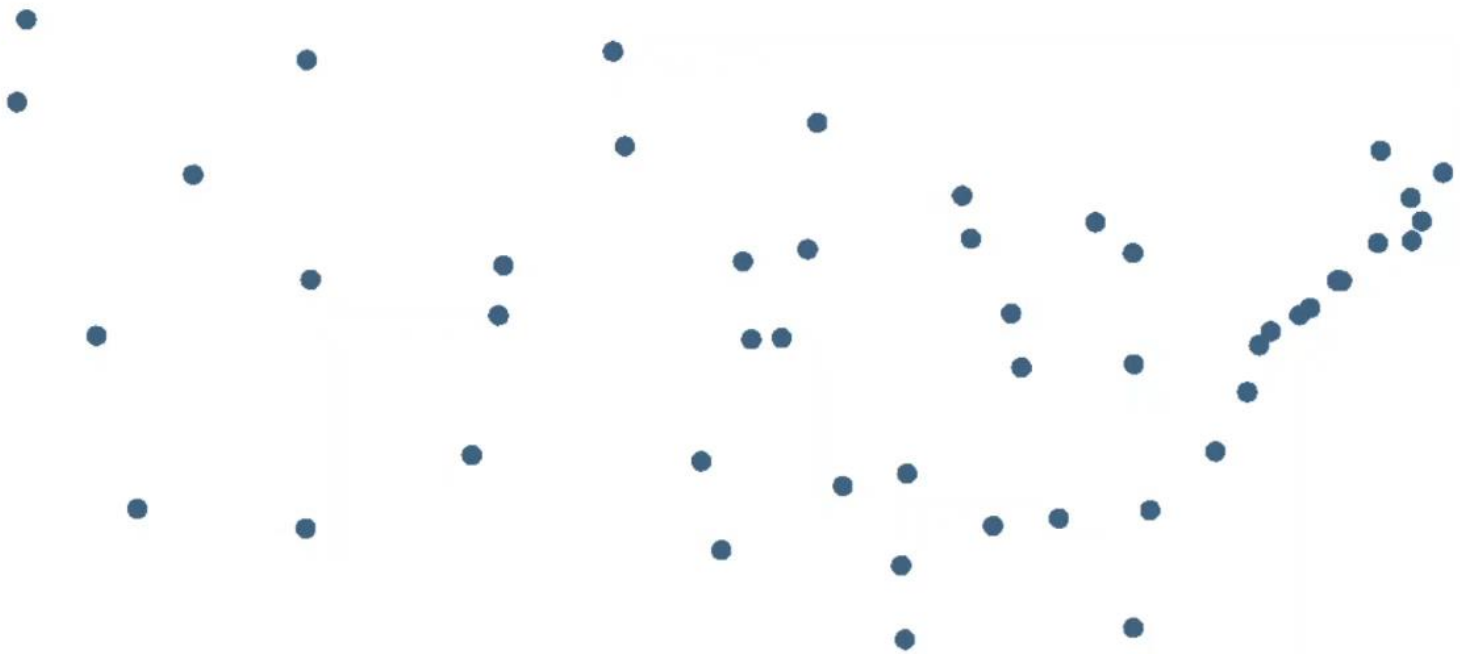
TSP can be modeled as an undirected weighted graph, such that cities are the graph's vertices, paths are the graph's edges, and a path's distance is the edge's weight.

It is a minimization problem starting and finishing at a specified vertex after having visited each other vertex exactly once.

Often, the model is a complete graph (i.e., each pair of vertices is connected by an edge). If no path exists between two cities, adding a sufficiently long edge will complete the graph without affecting the optimal tour.

A greedy algorithm is a general term for algorithms that try to add the lowest cost possible in each iteration, even if they result in sub-optimal combinations.

In this example, all possible edges are sorted by distance, shortest to longest. Then the shortest edge that will neither create a vertex with more than 2 edges, nor a cycle with less than the total number of cities is added. This is repeated until we have a cycle containing all of the cities.



Greedy Algorithms

Greedy algorithms can be characterized as being 'short sighted', and also as 'non-recoverable'.

They are ideal only for problems which have 'optimal substructure'.

Despite this, for many simple problems, the best suited algorithms are greedy algorithms.

Greedy algorithms mostly (but not always) fail to find the globally optimal solution because they usually do not operate exhaustively on all the data.

They can make commitments to certain choices too early which prevent them from finding the best overall solution later.

Greedy Algorithms

So why use a greedy algorithm?

The greedy method is quite powerful and work well for a large range of problems.

They are useful because they are quick to think up and often give good approximations to the optimum.

If a greedy algorithm can be proven to yield the global optimum for a given problem class, it typically becomes the method of choice because it is faster than other optimization methods like dynamic programming.

Greedy Algorithms

Greedy algorithms appear in network routing.

Using greedy routing, a message is forwarded to the neighboring node which is "closest" to the destination.

The notion of a node's location (and hence "closeness") may be determined by its physical location, as in geographic routing used by ad hoc networks.

Is Dijkstra's Algorithm greedy?

Yes!

Greedy Algorithms

Other examples of greedy algorithms are

Prim's Algorithm for Minimum Spanning Trees
Kruskal's Algorithm for Minimum Spanning Trees

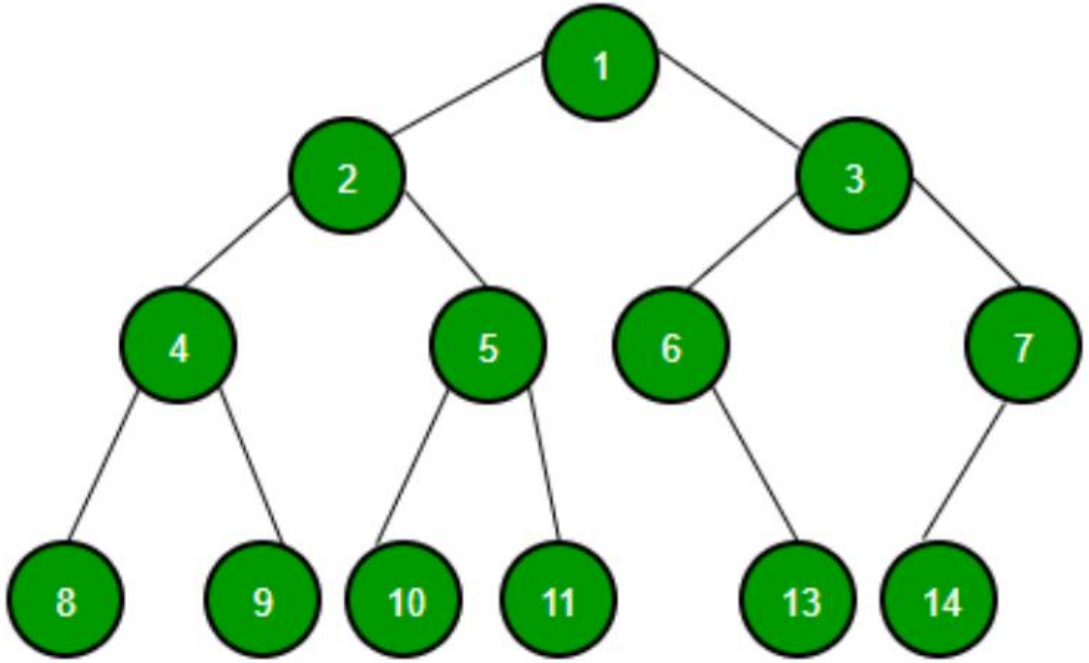
So, first, let's talk about what is a

Minimum Spanning Tree

How about we start with what is a spanning tree??

Spanning Tree

A **spanning tree** of an undirected graph is a subgraph that includes all of the vertices of the graph and is also a **tree**.



Graph vs Tree

Graph	Tree
Non-linear data structure	Non-linear data structure
Collection of vertices and edges	Collection of nodes and edges
Each vertex can have any number of edges	General trees can have any number of child nodes. Binary trees have at most two child nodes.
No unique vertex called root	Unique node called root
A cycle can be formed	No cycles are allowed

Spanning Tree

A tree is a connected, acyclic graph.

A **minimum spanning tree** (MST) of an undirected graph is a subgraph that includes all of the vertices of the graph **with a minimum number of edges** and is also a tree.

The primary goal of a spanning tree is to connect all vertices with the minimum number of edges.

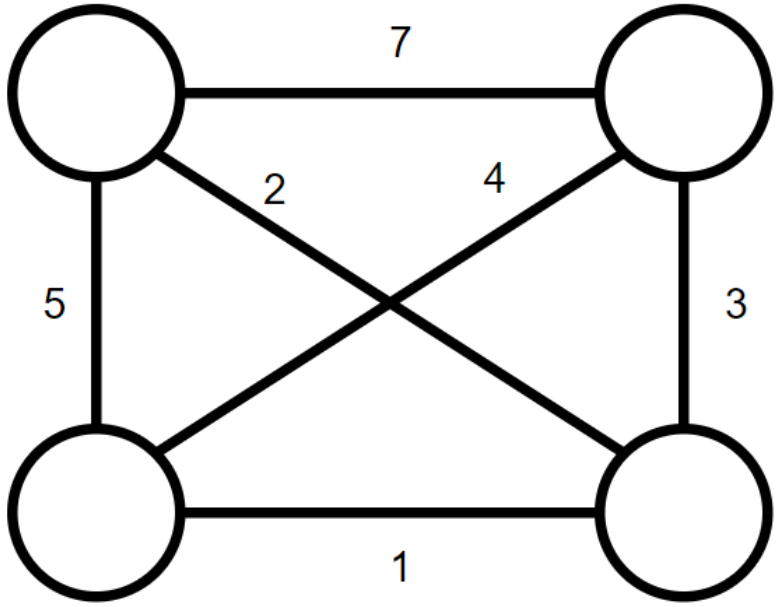
Every connected graph contains at least one spanning tree - at least one connected acyclic spanning subgraph.

A connected graph will usually have many spanning trees

More generally, every graph contains at least one spanning forest and usually several spanning forests

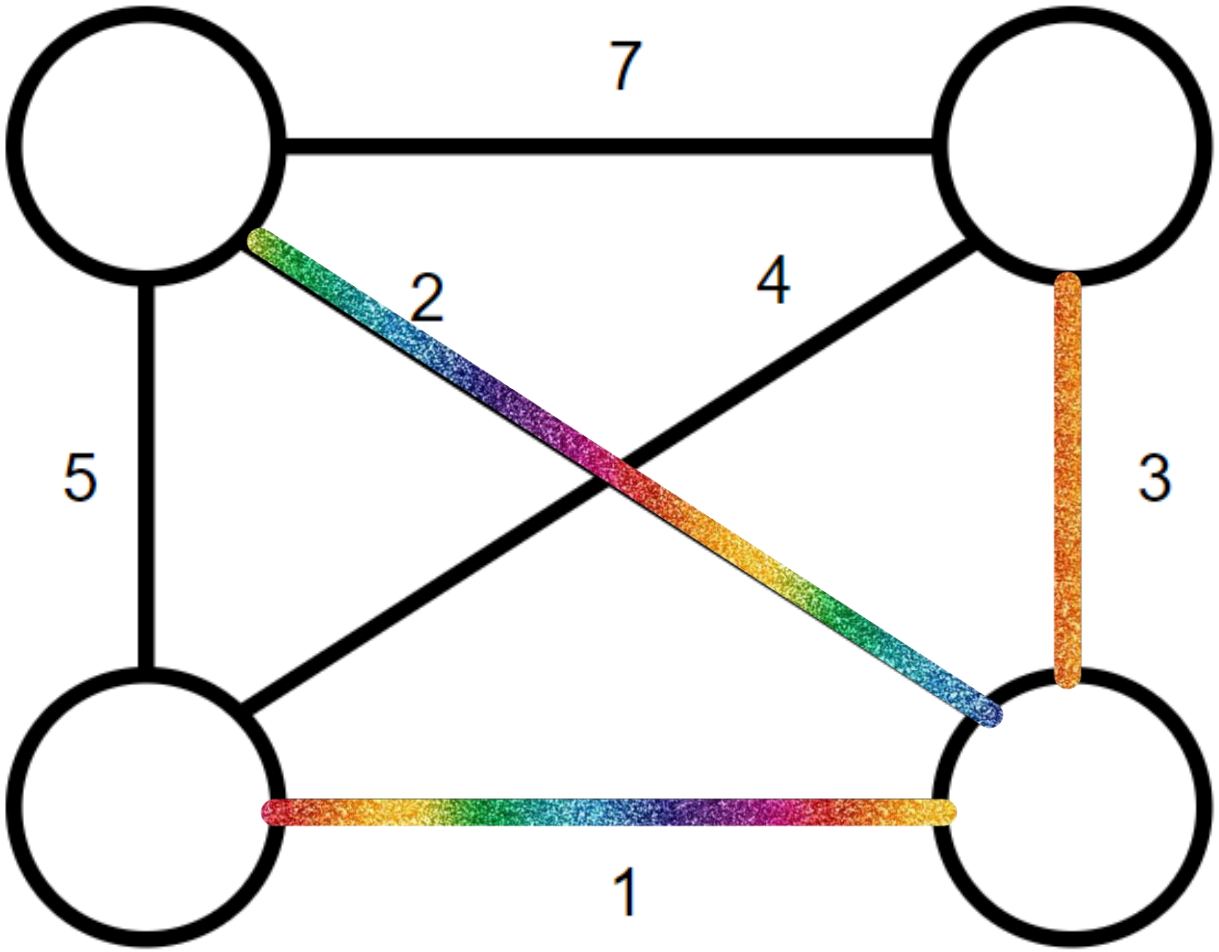
Minimum Spanning Tree

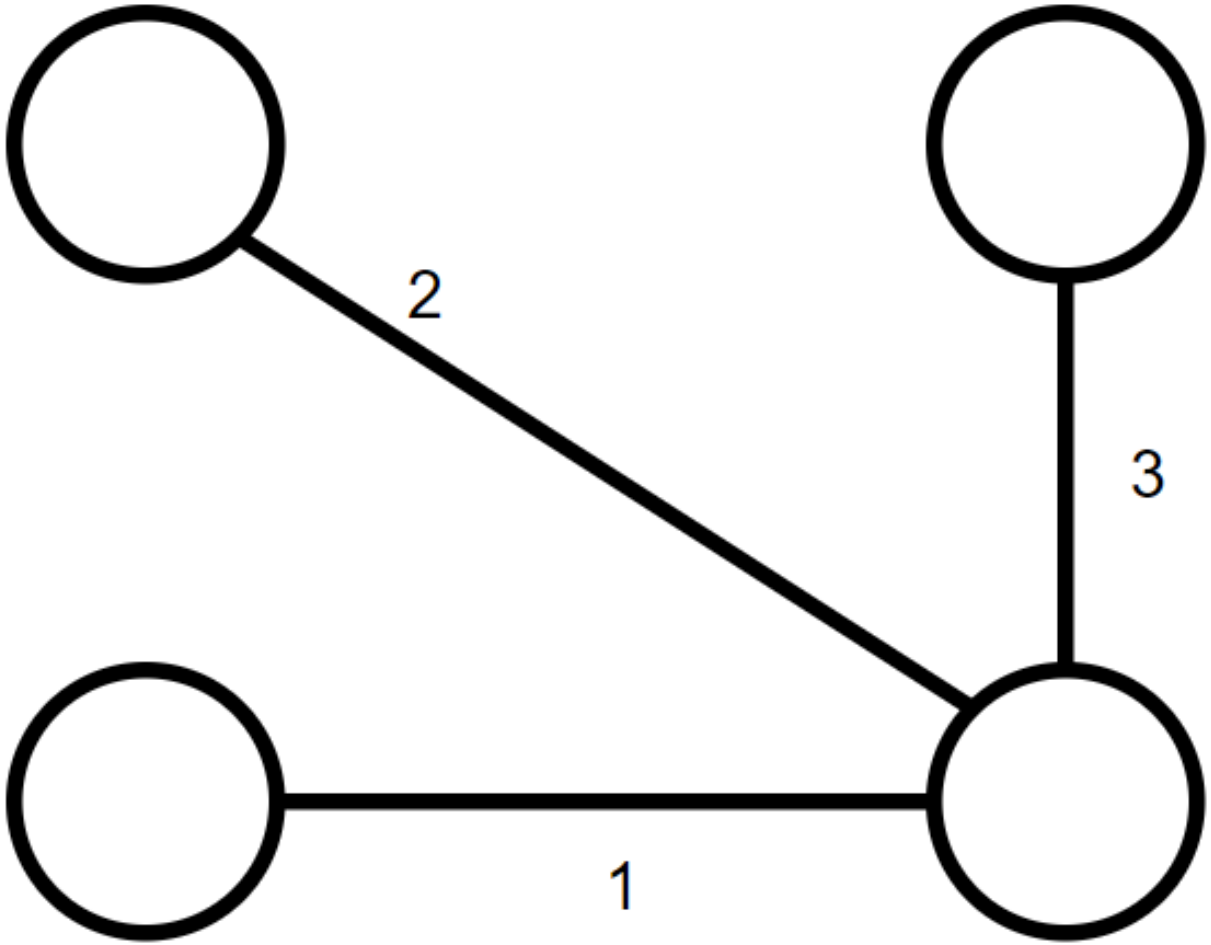
A MST is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible edge weight.



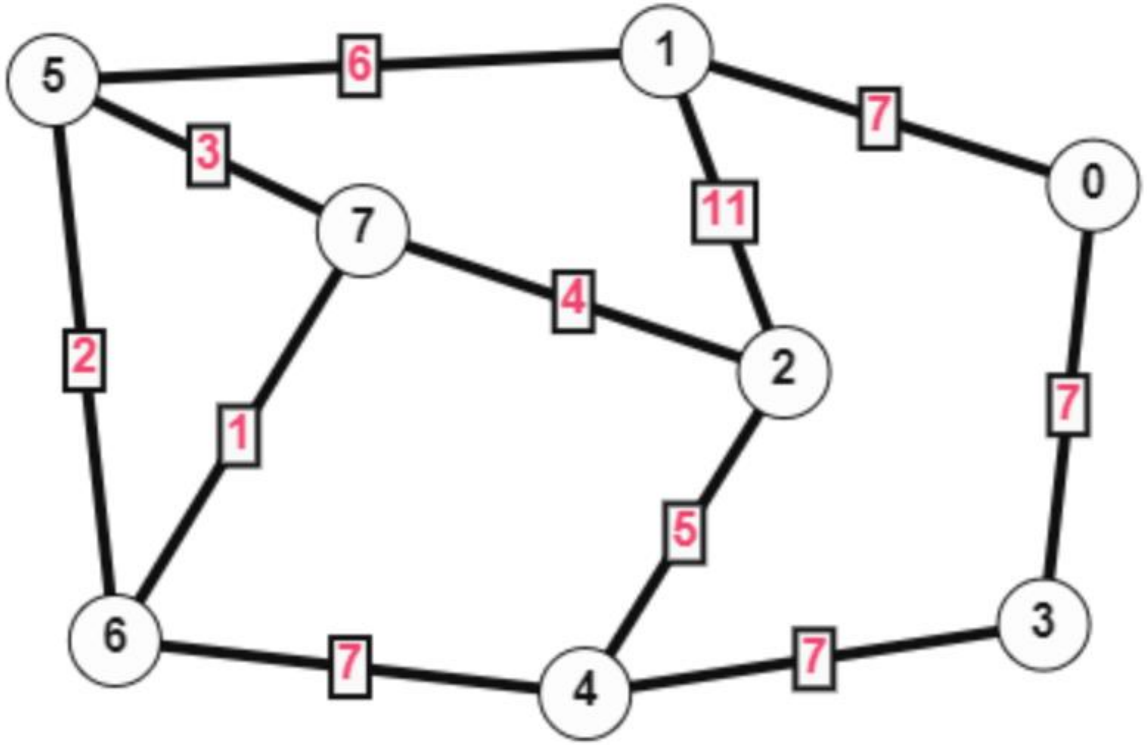
What is the MST of this graph?

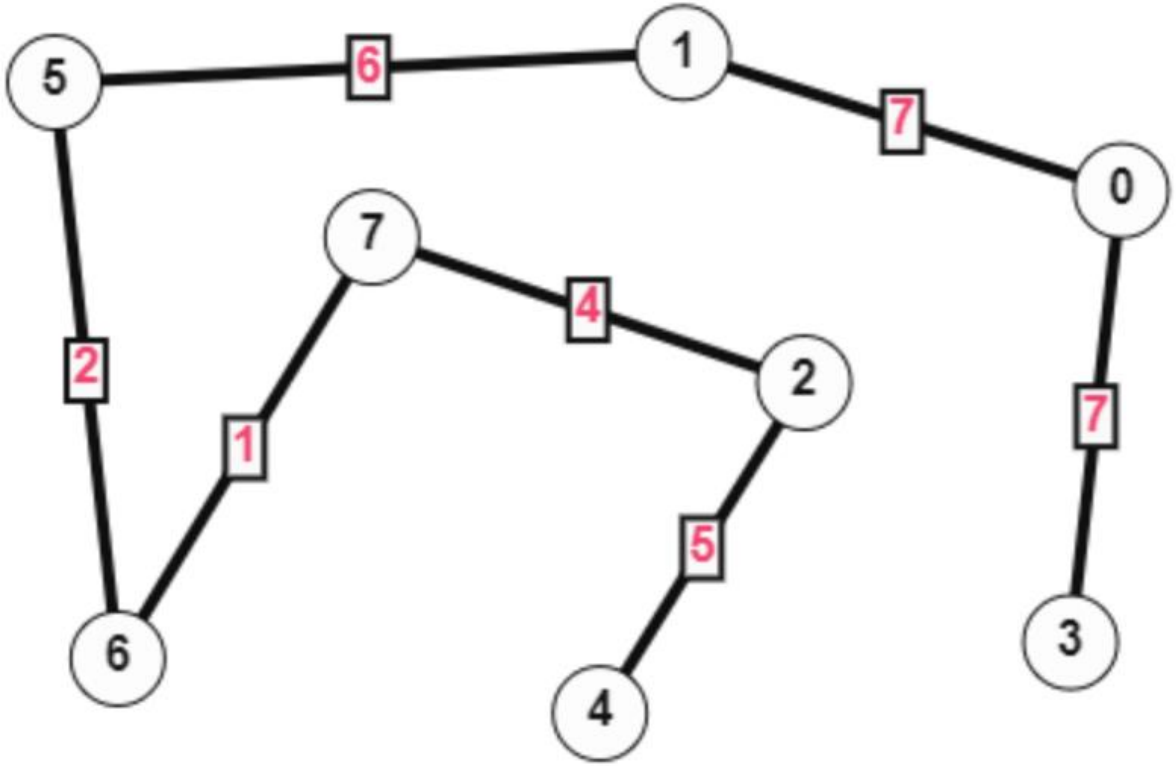
Minimum Spanning Tree

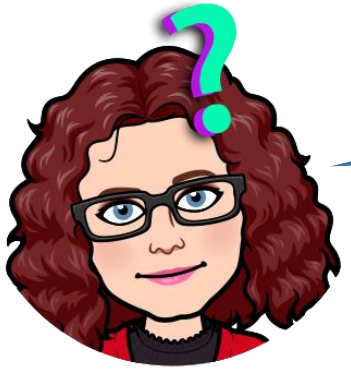




Minimum Spanning Tree



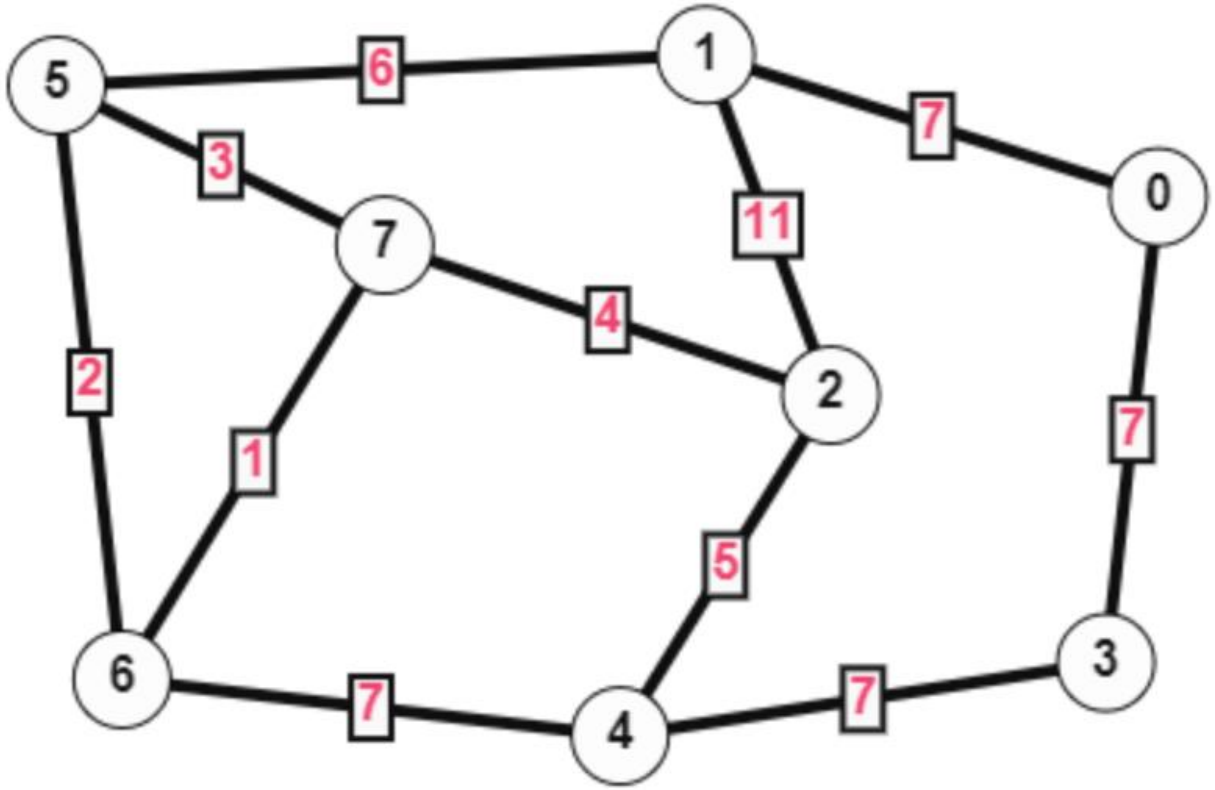


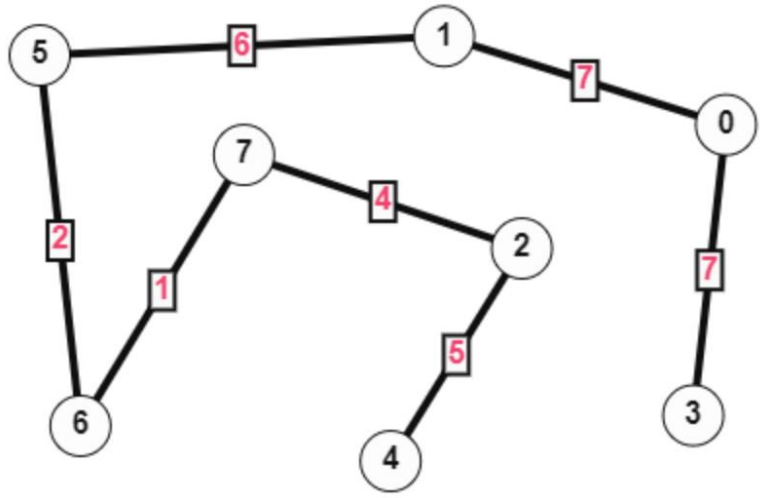


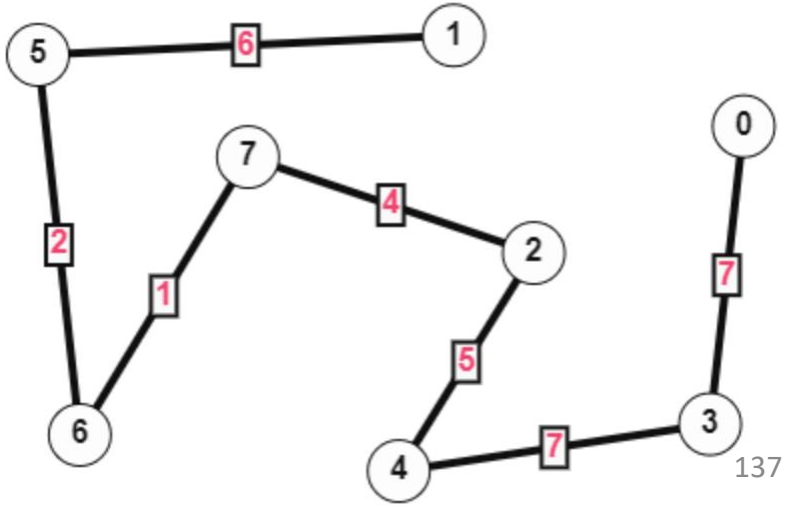
How many vertices are in the graph?
How many edges are in the MST?
Hmmmm…

Minimum Spanning Tree

Is the MST unique?







Minimum Spanning Tree

Before we talk about forests, let's define **connected component**.

A **component**, sometimes called a **connected component**, of an undirected graph is a subgraph in which any two vertices are connected to each other by paths and which is connected to no additional vertices in the supergraph.

A vertex with no incident edges is itself a component.

A connected graph has exactly one component - the whole graph.

Minimum Spanning Tree

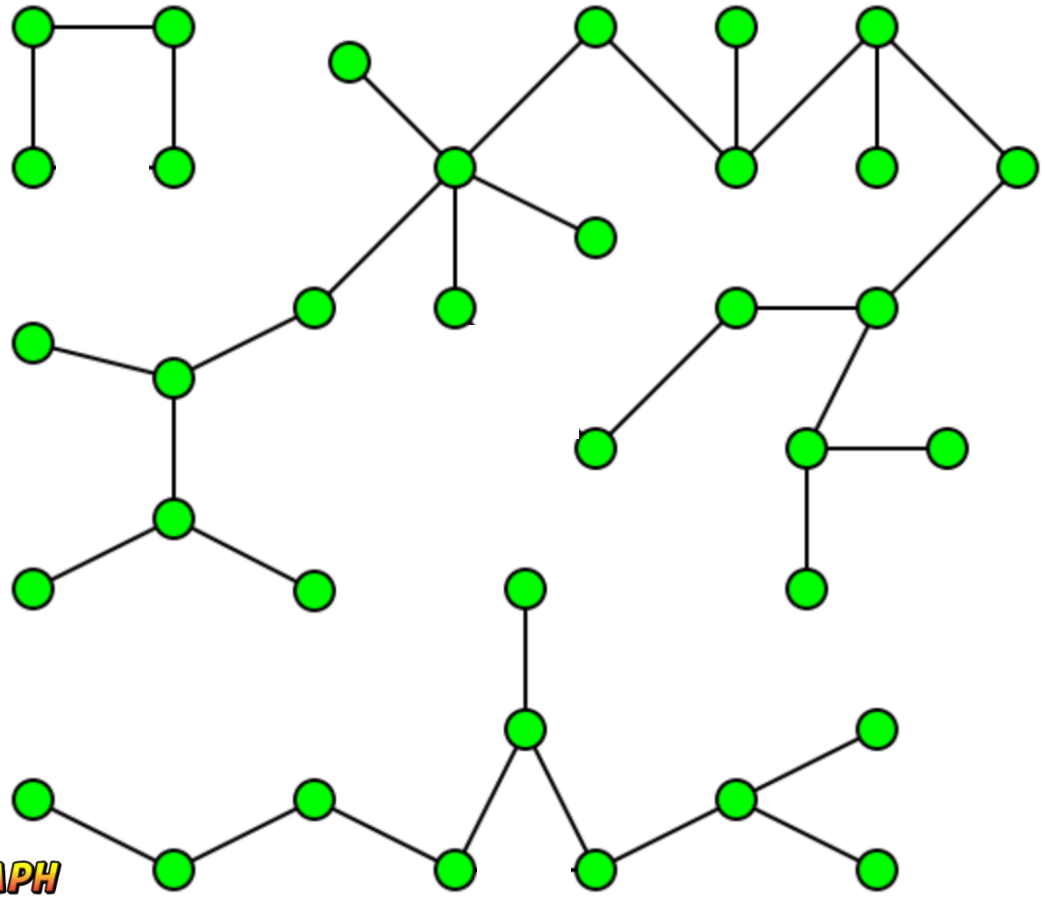
When there is only one connected component in your graph,

spanning tree = spanning forest.

When there are multiple connected components in your graph,

all spanning trees together = spanning forest

Minimum Spanning Tree



So for each connected component, we will have a spanning tree and all **3** spanning trees will constitute a spanning forest

Minimum Spanning Tree

What are MSTs used for?

- Network design
  - Telephone
  - Cable
- Finding approximate solutions for complex mathematical problems like the Traveling Salesman Problem
- Cluster Analysis.
- Real-time face tracking and verification (i.e. locating human faces in a video stream).
- Autoconfig protocol for Ethernet bridging to avoid cycles in a network
- Entropy based image registration.
- Max bottleneck paths.
- Dithering (adding white noise to a digital recording in order to reduce distortion).

Minimum Spanning Tree

A telecommunications company trying to lay cable in a new neighborhood.

They can only bury the cable only along certain paths/roads.

A graph could be constructed that contains the points/houses connected by those paths.

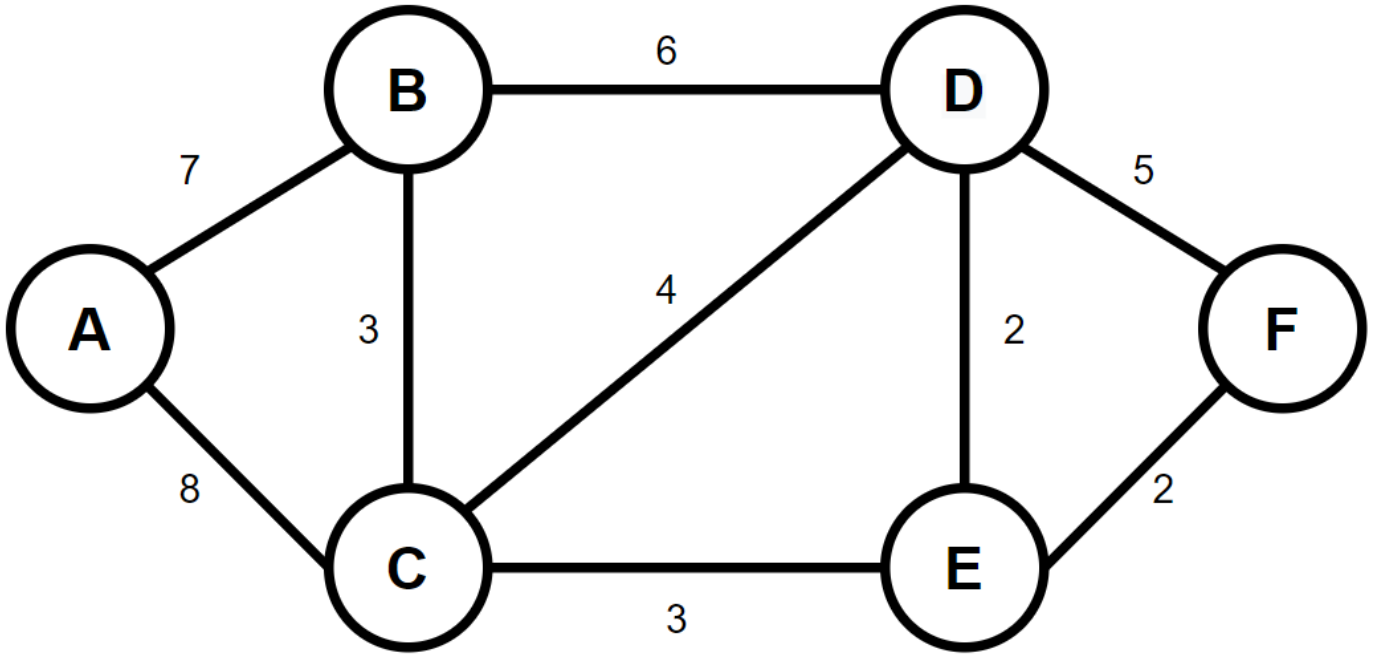
Some of the paths might be more "expensive" – longer or require the cable to be buried deeper – these differences would be represented by the weights assigned to edges.

A *spanning tree* for that graph would be a subset of those paths that has no cycles but still connects every house; there might be several spanning trees possible.

A *minimum spanning tree* would be one with the lowest total cost/weight which would represent the least expensive path for laying the cable.

Minimum Spanning Tree

What is the
Minimum Spanning Tree
for this graph?



MST – Prim's Algorithm

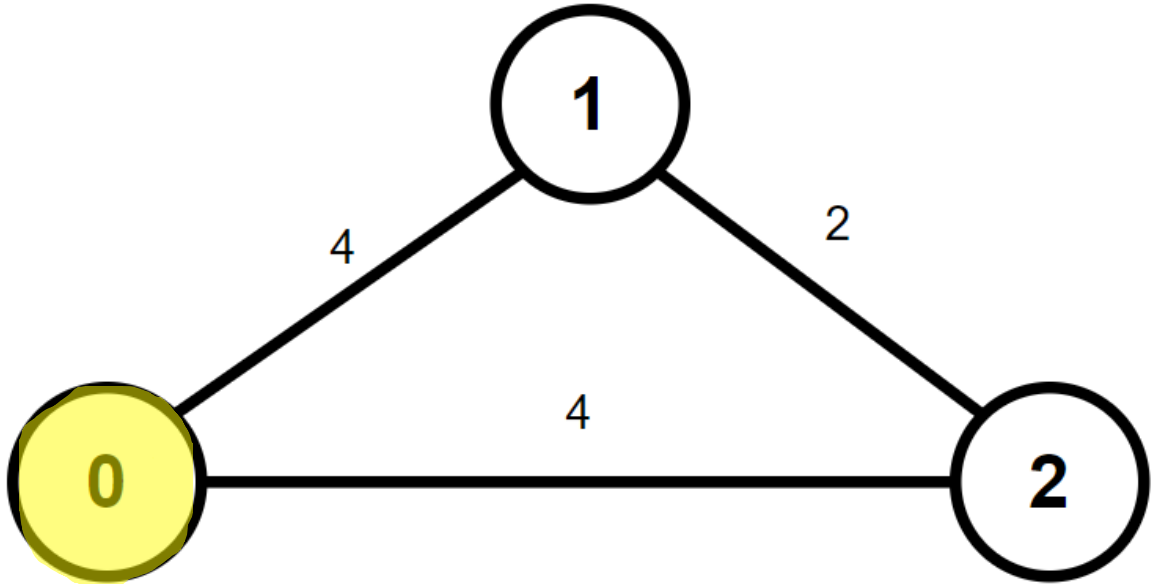
Let's start with something a little smaller….

Prim's Algorithm tell us to pick a starting vertex.

Let's pick 0 and add it to our MST.

The edges incident to 0 are

the edge to Vertex 1 with a weight of 4 – (0,1,4)
the edge to Vertex 2 with a weight of 4 – (0,2,4)



Notice that I am denoting the edge using (from-vertex, to-vertex, weight)

(0,1,4)
(0,2,4)

Now I need to sort these by weight and then by to-vertex…

MST – Prim's Algorithm

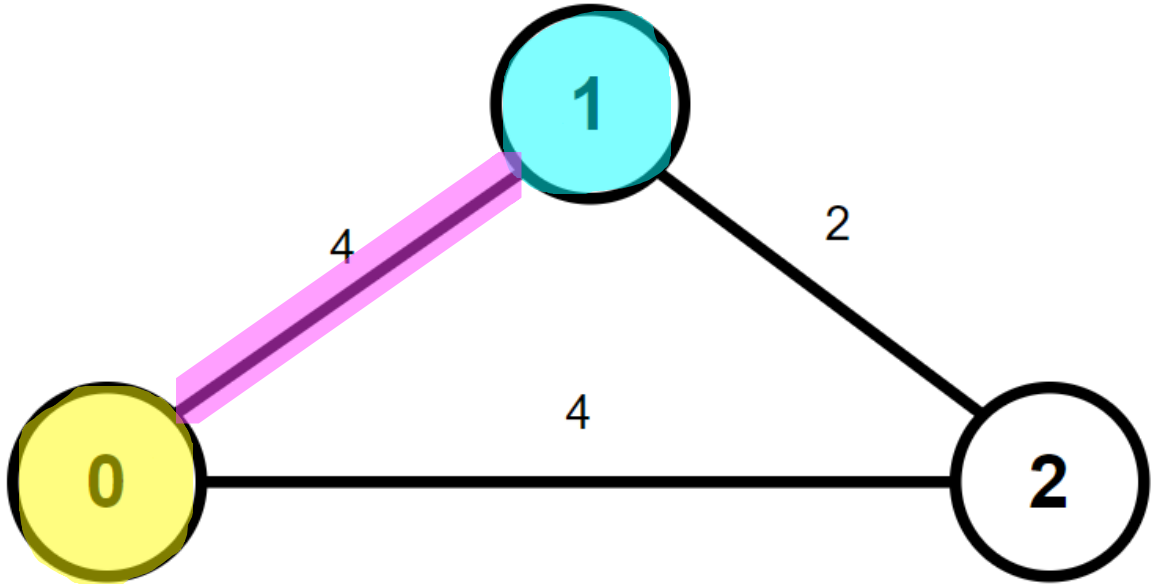
We pick the edge with the lowest/minimum weight/cost.

In this case, both edges have a weight of 4 so we pick (0,1,4) before (0,2,4) because of our sorting.

We remove (0,1,4) from our list.

So now we add Vertex 1 to our MST.

Which edges are incident to Vertex 1?



We don't consider Vertex 0 because Vertex 0 is already part of our MST.

We add (1,2,2) – the edge to Vertex 2 with a weight of 2 – we add it to our sorted list.

~~(0,1,4)~~

(1,2,2)

(0,2,4)

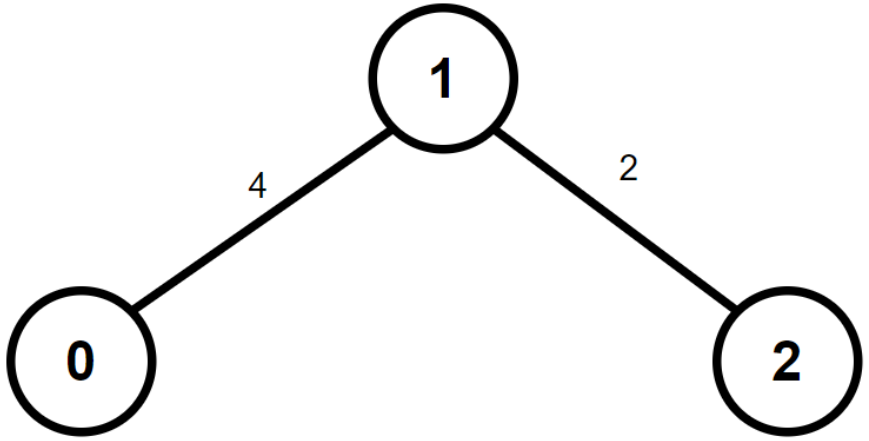
MST – Prim's Algorithm

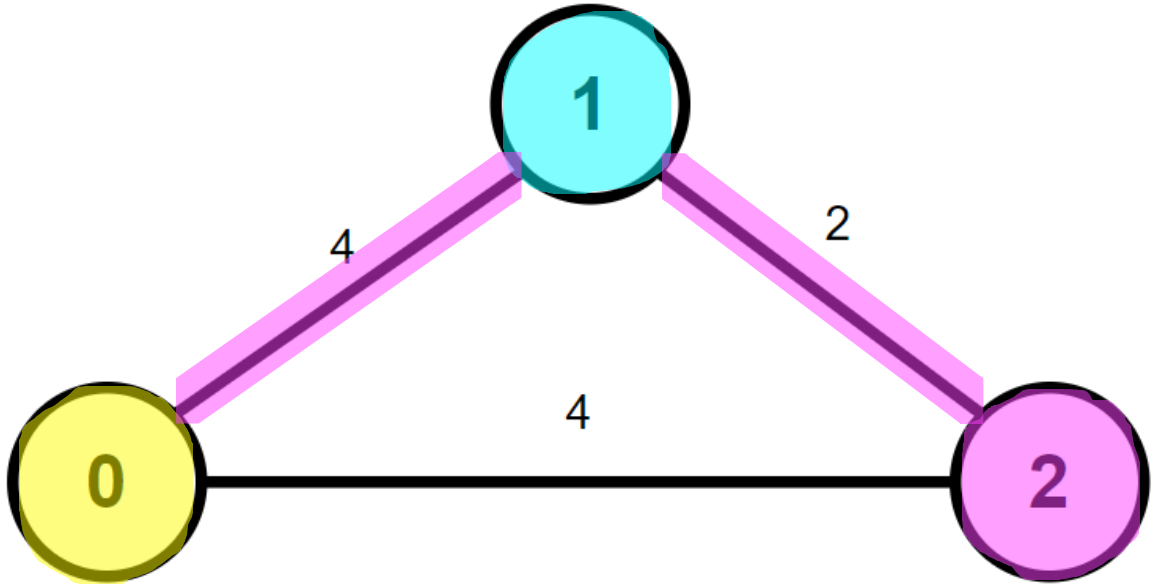
We pick the edge with the lowest/minimum weight/cost.

Now that is (1,2,2)

We add Vertex 2 to our MST.

Our graph has 3 vertices and we have 3 vertices in our MST and 2 edges.



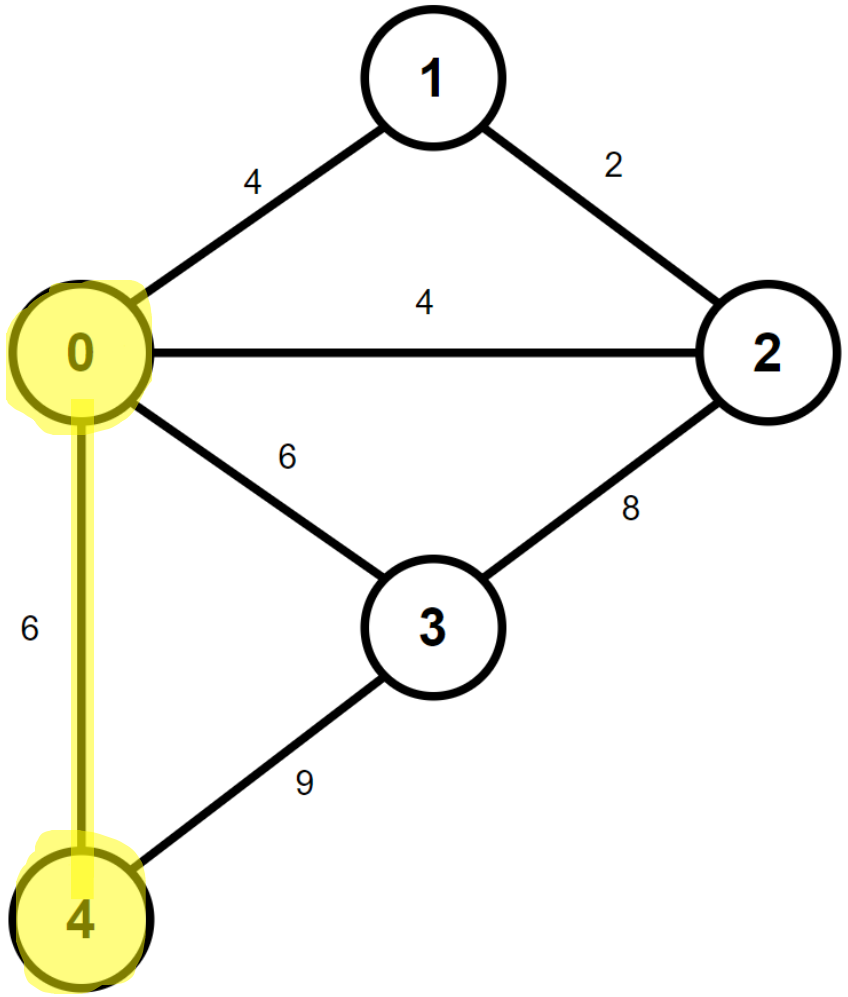


~~(0,1,4)~~

~~(1,2,2)~~

(0,2,4)

MST – Prim's Algorithm



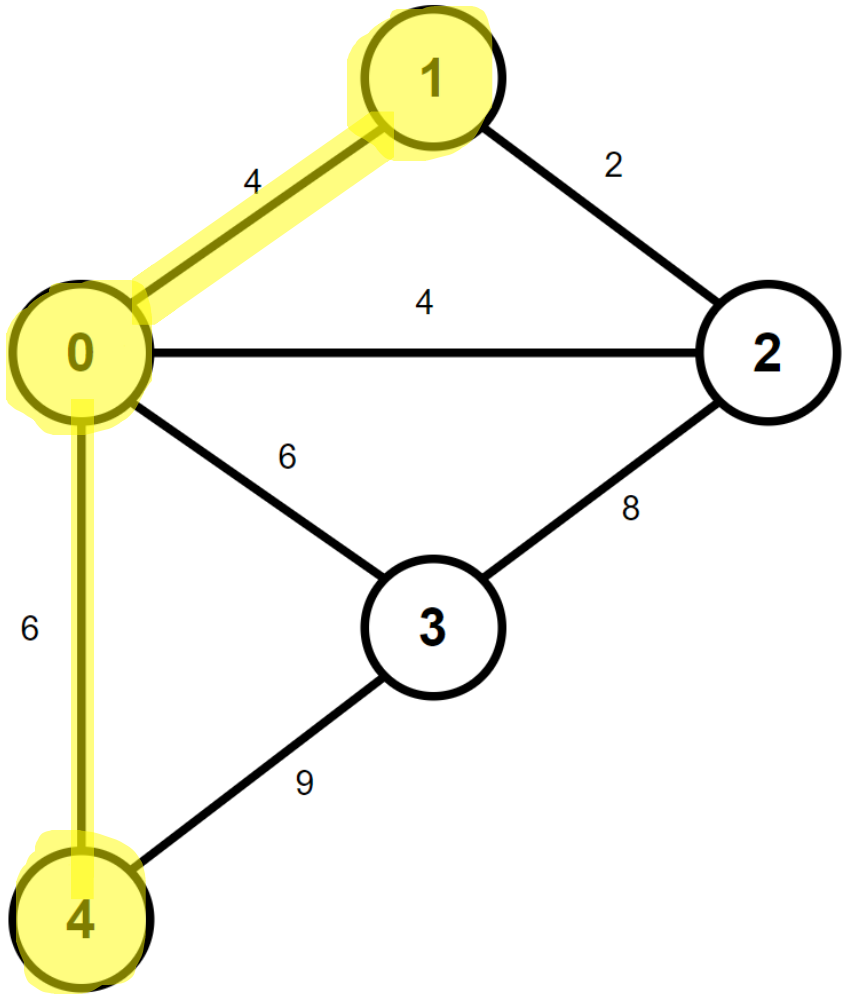
Pick a starting vertex.

Let's pick 4.

~~(4,0,6)~~

(4,3,9)

MST – Prim's Algorithm



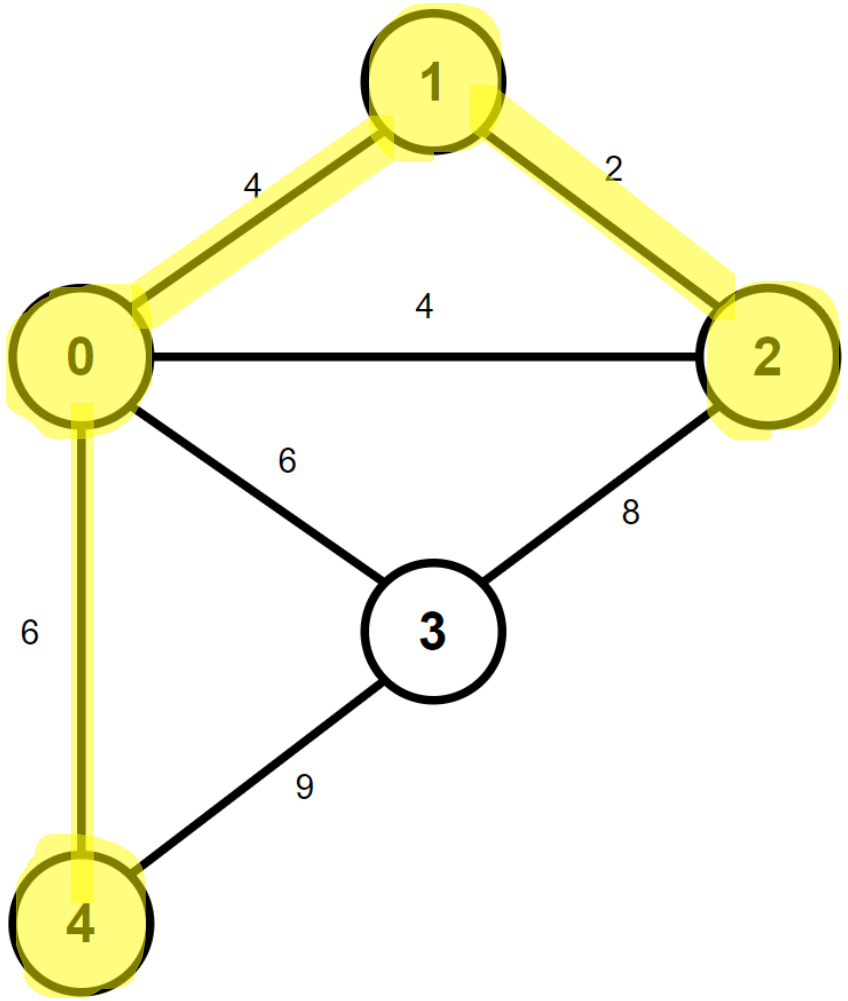
~~(4,0,6)~~
~~(0,1,4)~~
(0,2,4)
(0,3,6)
(4,3,9)

Now we look at the edges from Vertex 0.

(0,3,6)
(0,2,4)
(0,1,4)

We add them in order to our list.

MST – Prim's Algorithm



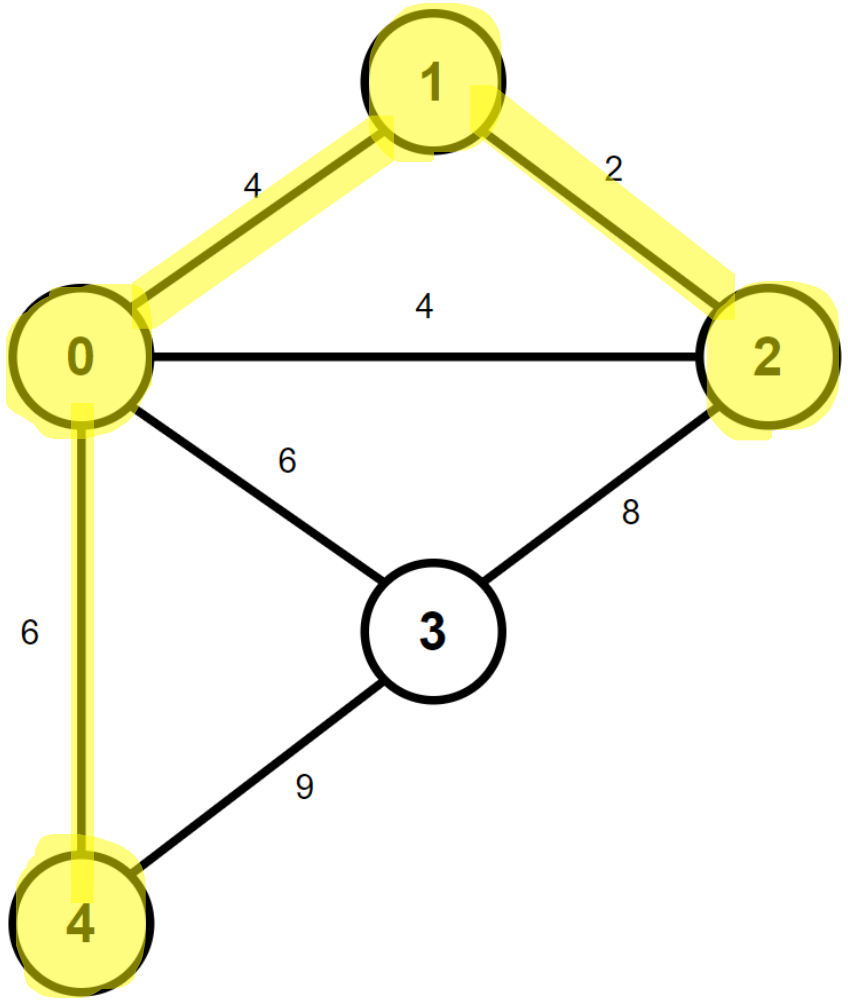
~~(4,0,6)~~
~~(0,1,4)~~
~~(1,2,2)~~
(0,2,4)
(0,3,6)
(4,3,9)

Now we look at the edges from Vertex 1.

(1,2,2)

We add to our ordered list.

MST – Prim's Algorithm



~~(4,0,6)~~
~~(0,1,4)~~
~~(1,2,2)~~
(0,2,4)
(0,3,6)
(2,3,8)
(4,3,9)

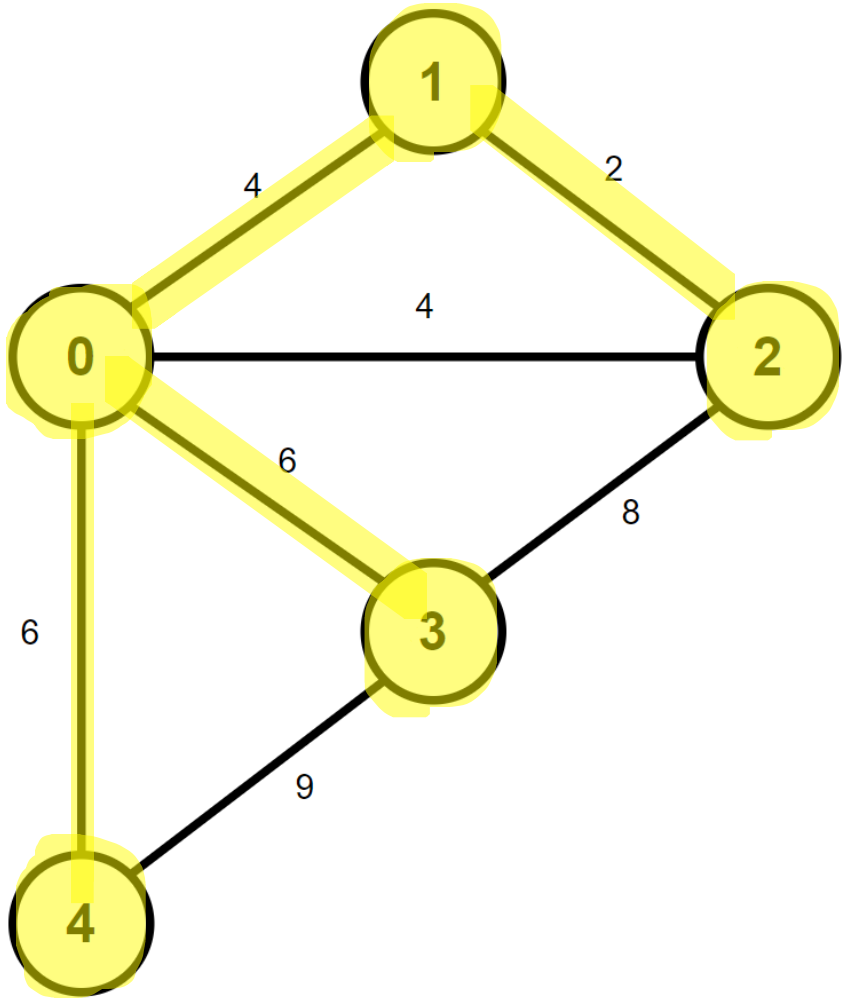
Vertex 2 has an edge with a weight of 4 going to Vertex 0.

Vertex 0 is already in our MST so we don't add that edge.

Vertex 2 also has an edge of weight 8 going to Vertex 3.

We add (2,3,8) to the list.

MST – Prim's Algorithm



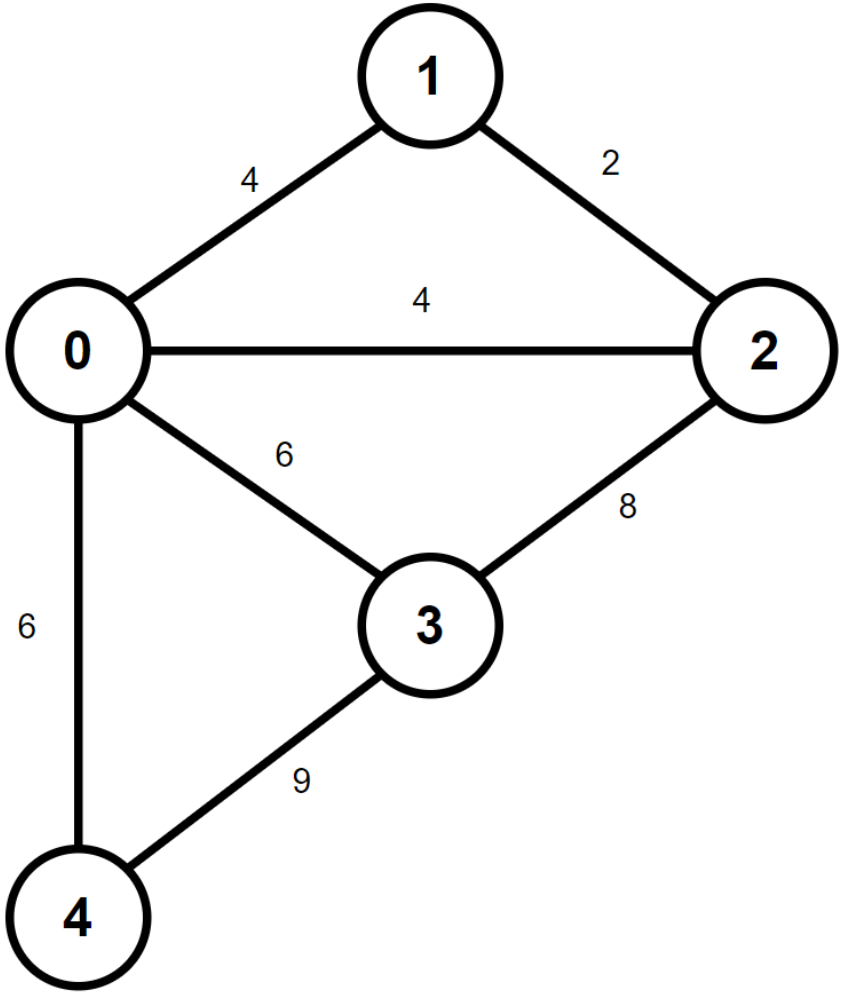
~~(4,0,6)~~
~~(0,1,4)~~
~~(1,2,2)~~
~~(0,2,4)~~
~~(0,3,6)~~
(2,3,8)
(4,3,9)

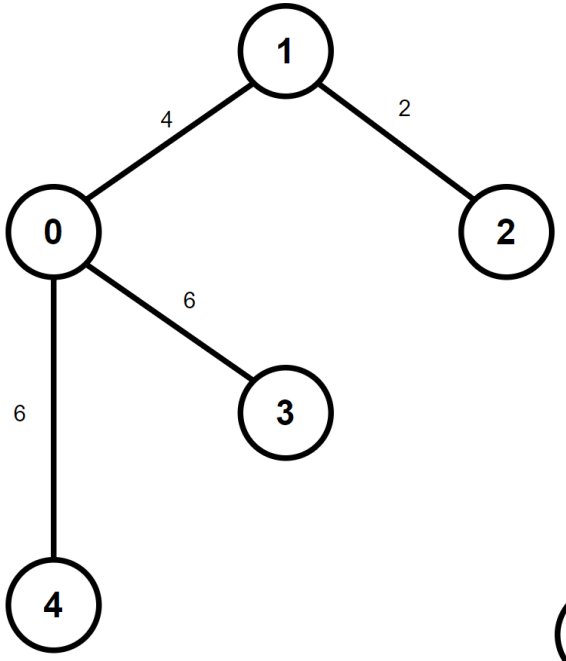
The next item in our list is (0,2,4).

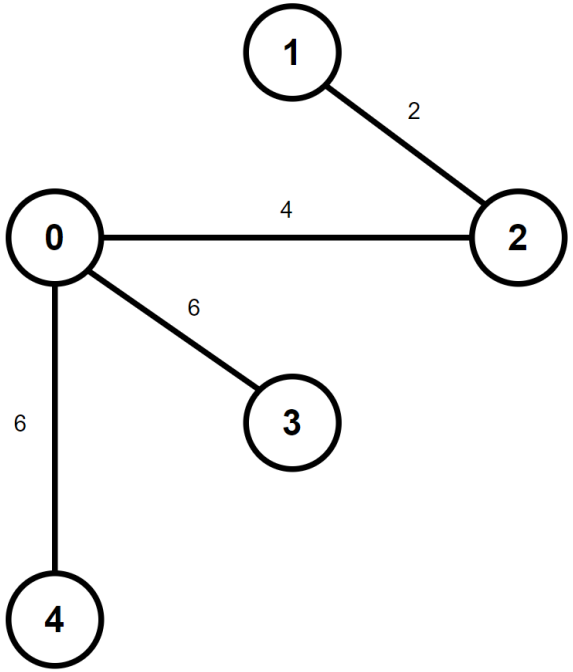
Since Vertex 2 has been added to our MST, we eliminate that choice from our list.

(0,3,6) is next and we have not been to Vertex 3 so we use (0,3,6) to add Vertex 3 to our MST.

MST – Prim's Algorithm



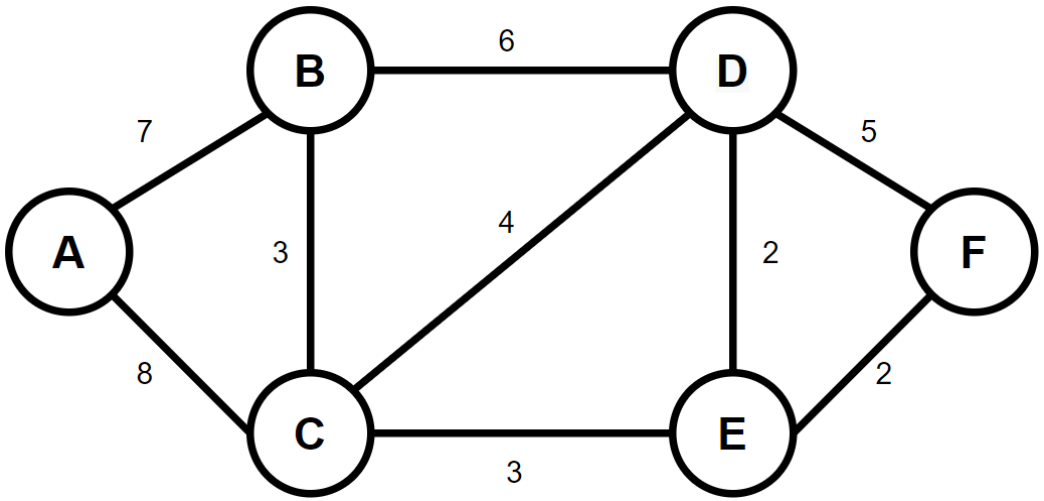


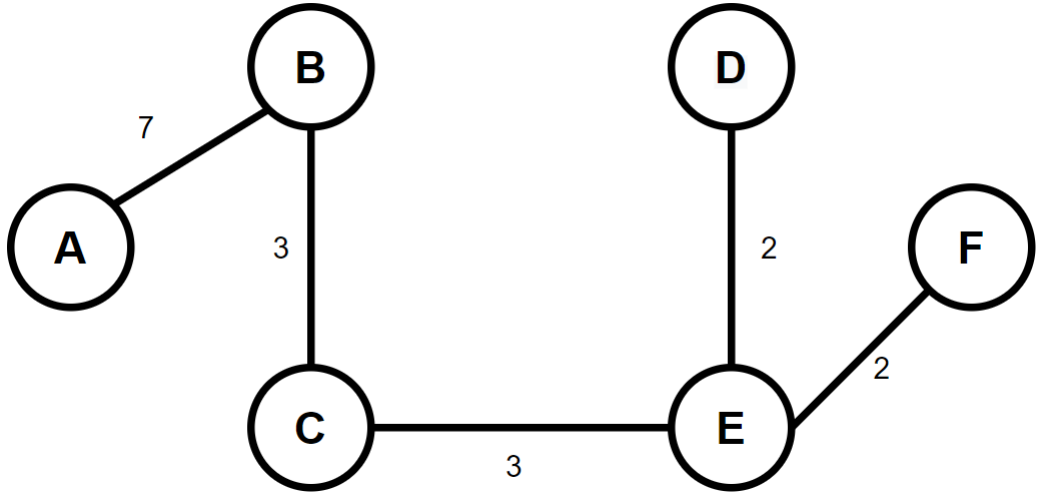


MST – Prim's Algorithm

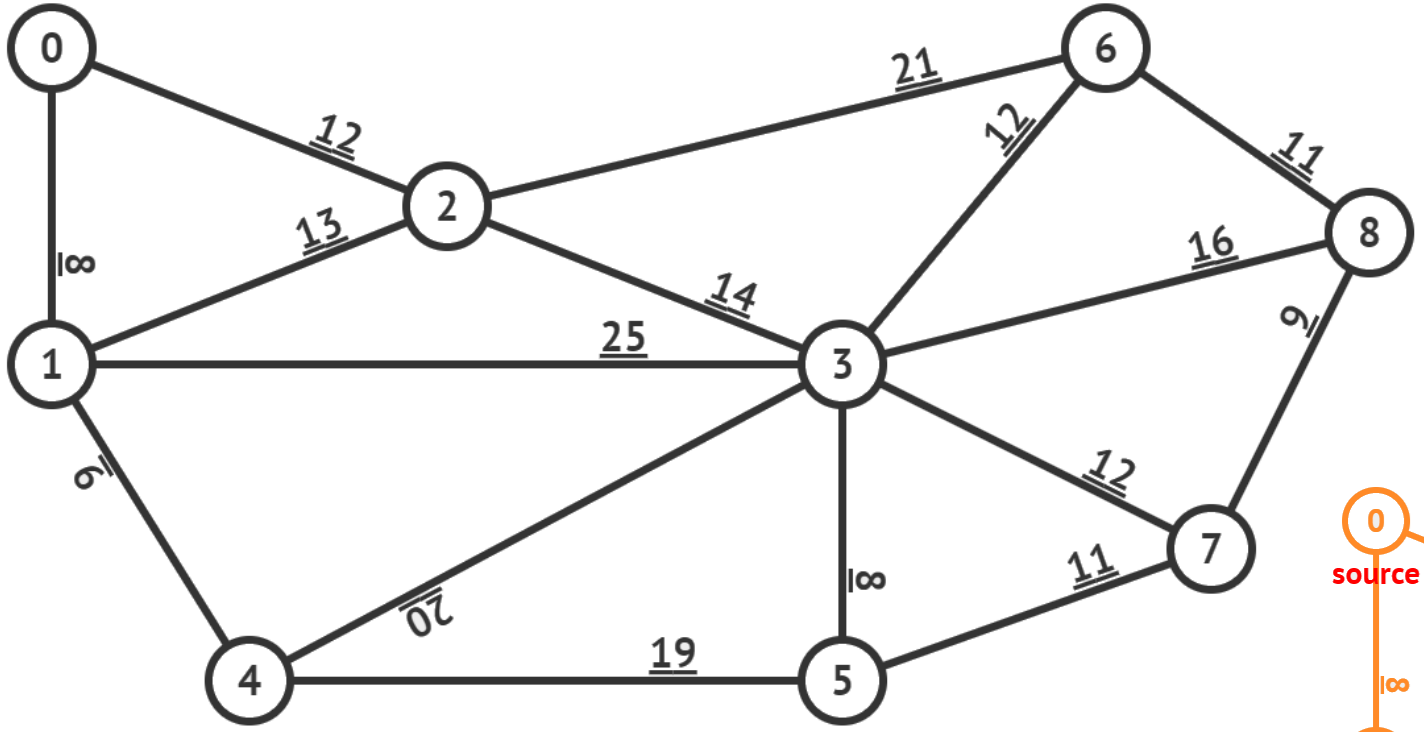
Start with vertex A

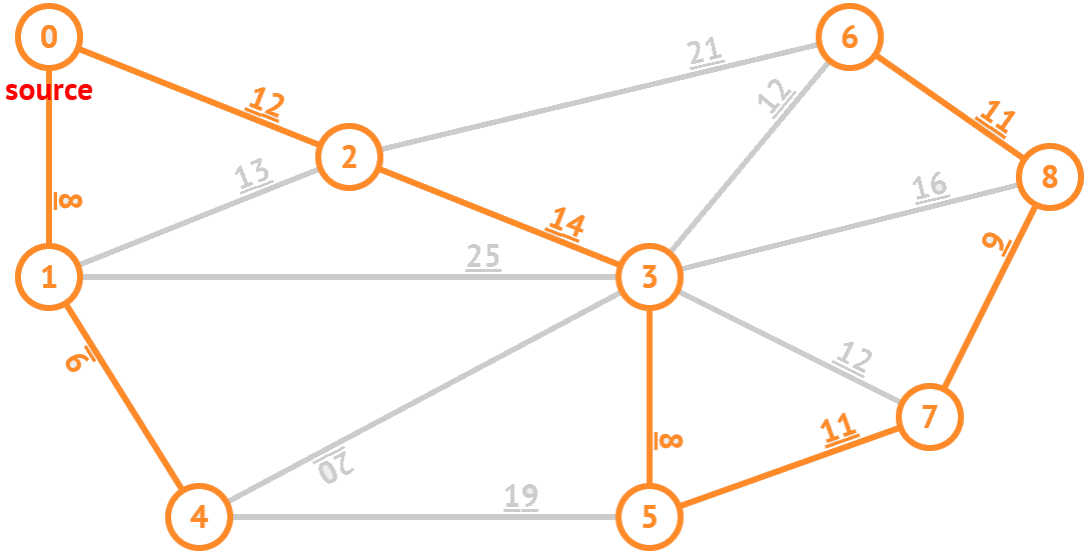
What is the Minimum Spanning Tree for this graph?



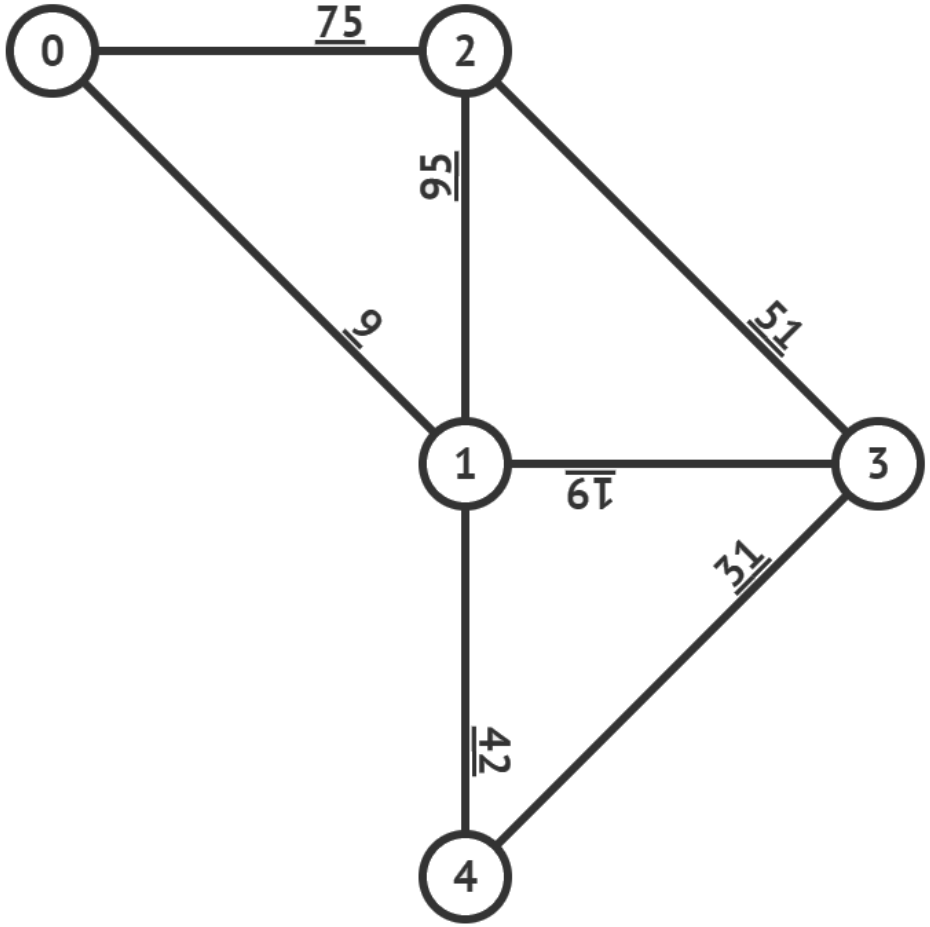


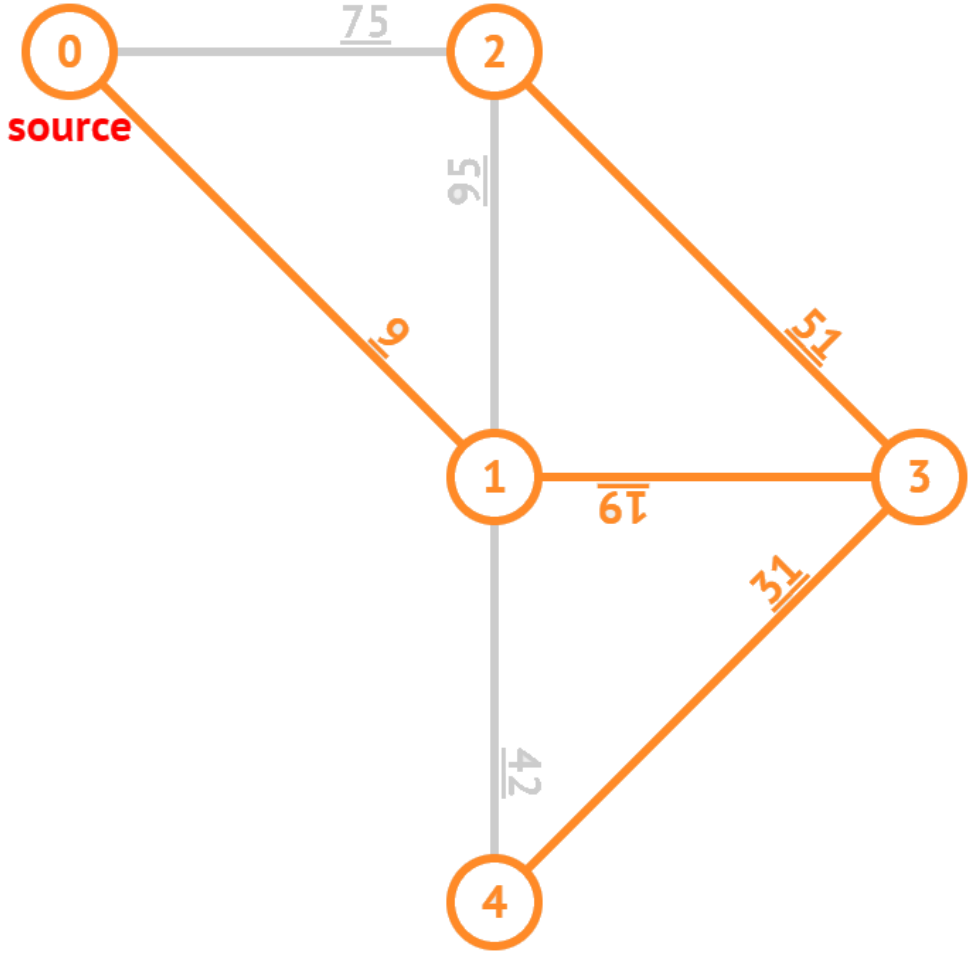
MST – Prim's Algorithm





MST – Prim's Algorithm



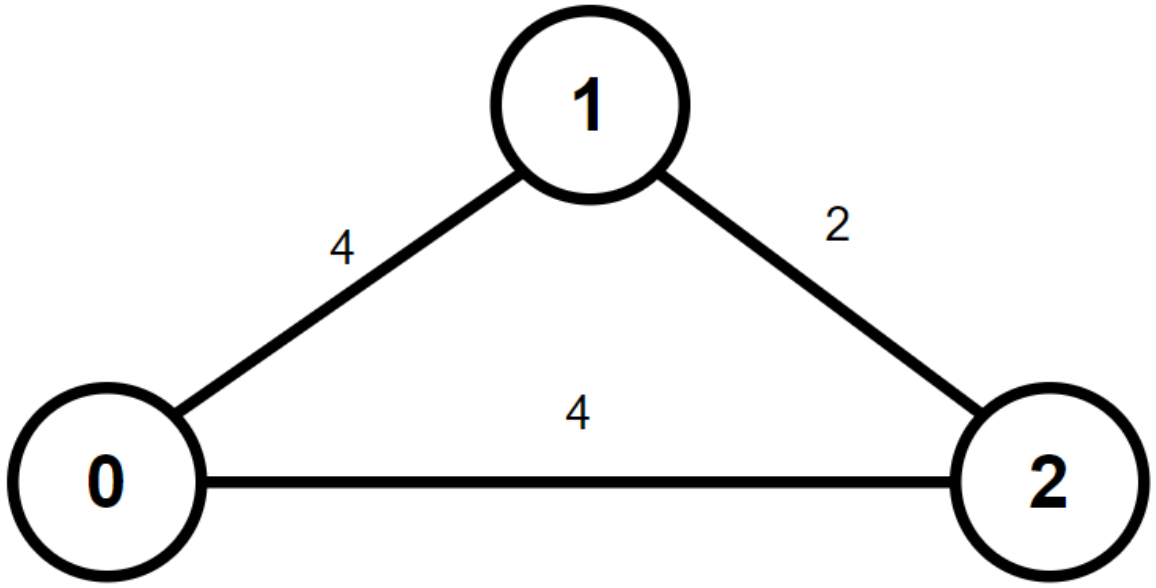


MST – Kruskal's Algorithm

Kruskal's Algorithm is another Greedy Algorithm for MST.

Kruskal's Algorithm tell us to order all the vertices by their edge weights.

(1,2,2)
(0,1,4)
(0,2,4)

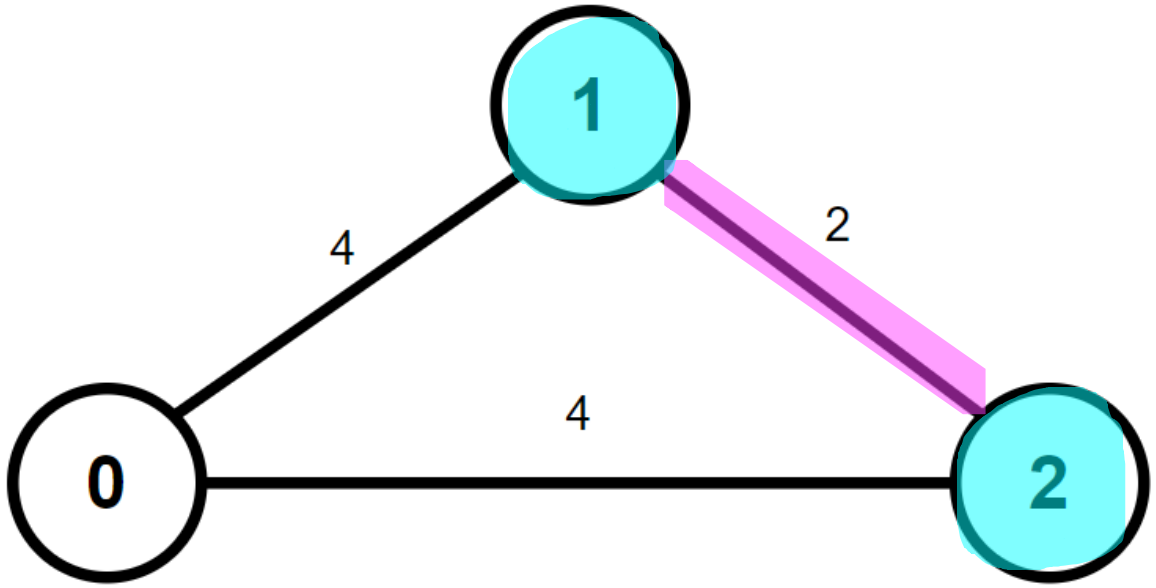


MST – Kruskal's Algorithm

(1,2,2)
(0,1,4)
(0,2,4)

Kruskal says to pick the edge with the lowest value/weight.

(1,2,2)



MST – Kruskal's Algorithm

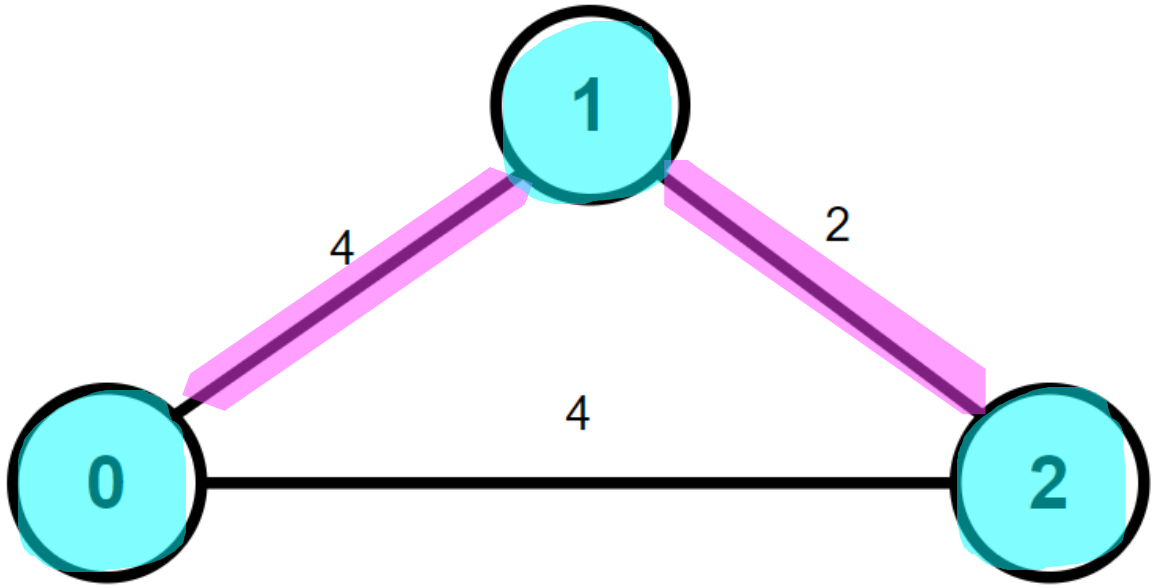
~~(1,2,2)~~

~~(0,1,4)~~

(0,2,4)

Kruskal says to pick the pair with the lowest value.

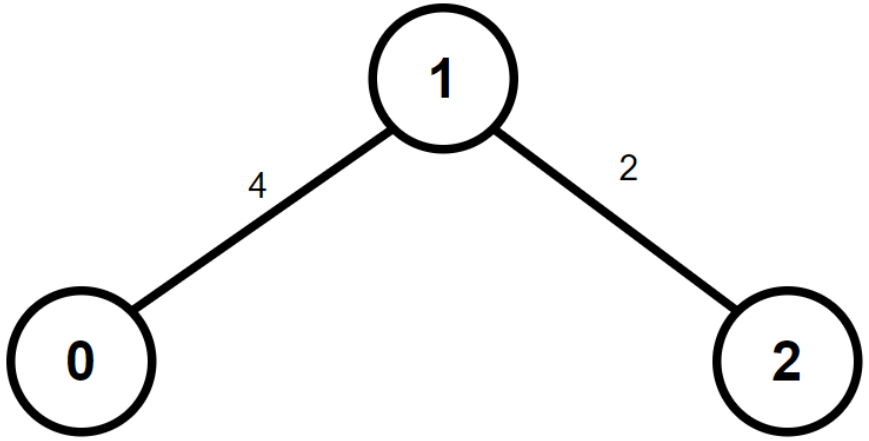
(0,1,4)

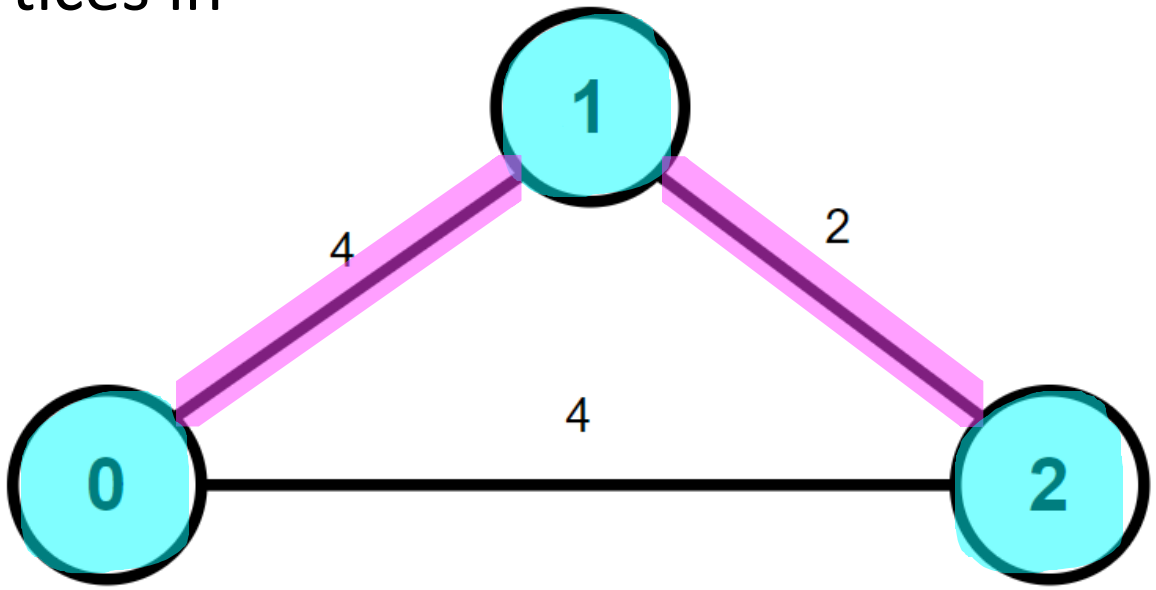


MST – Kruskal's Algorithm

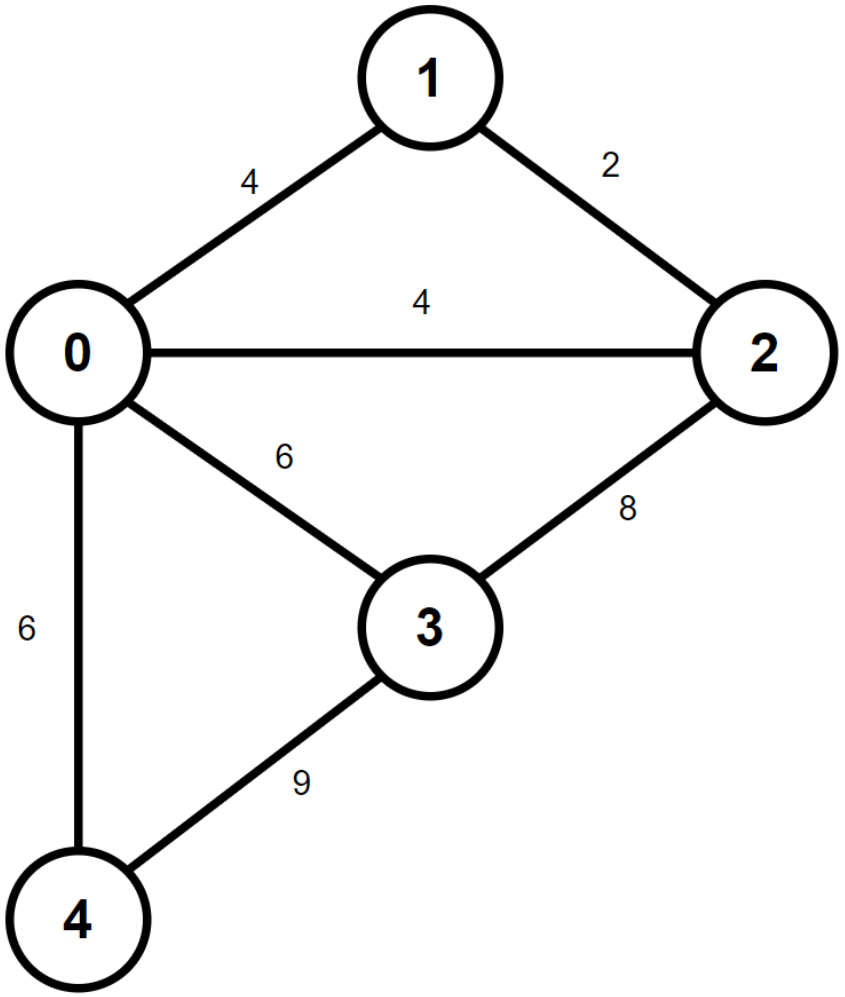
Our graph has 3 vertices and we have 3 vertices in our MST and 2 edges.

We are done.





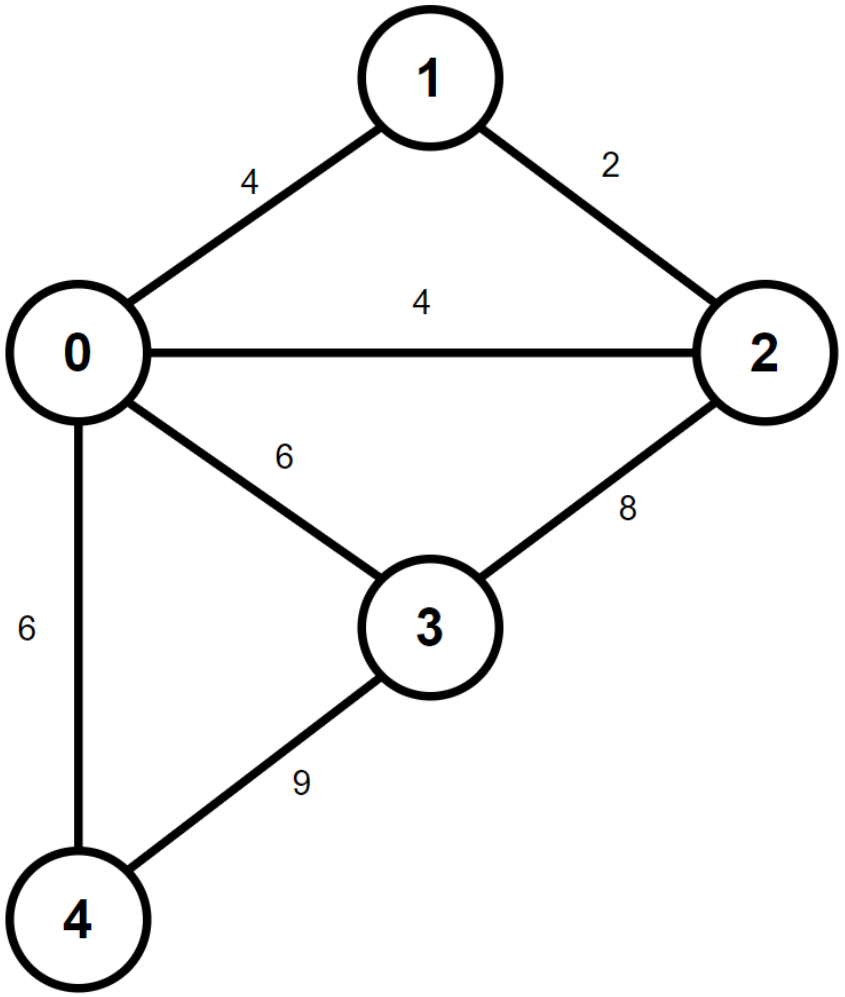
MST – Kruskal's Algorithm



Order all of the vertex,edge pairs by edge weight.

(0,1,4)
(0,2,4)
(0,3,6)
(0,4,6)
(1,2,2)
(2,3,8)
(3,4,9)

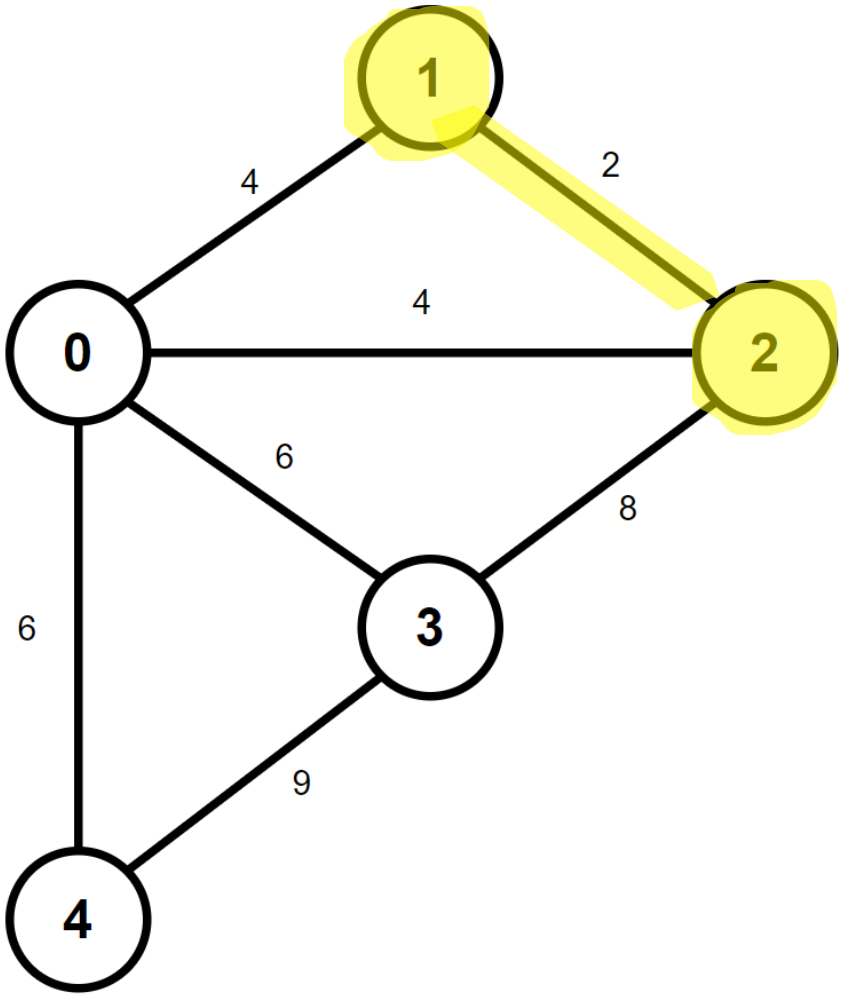
MST – Kruskal's Algorithm



Order all of the vertex,edge pairs by edge weight.

(1,2,2)
(0,1,4)
(0,2,4)
(0,3,6)
(0,4,6)
(2,3,8)
(3,4,9)

MST – Kruskal's Algorithm



Choose the minimum…

(1,2,2)

(0,1,4)

(0,2,4)

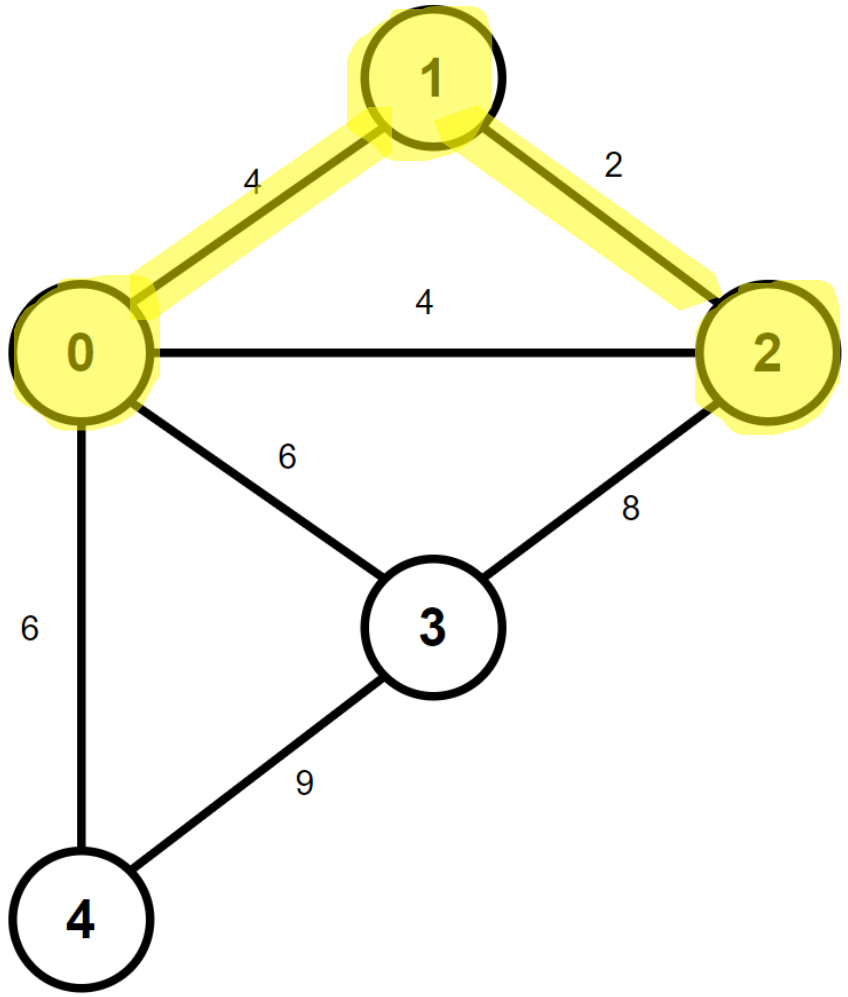
(0,3,6)

(0,4,6)

(2,3,8)

(3,4,9)

MST – Kruskal's Algorithm



Choose the minimum…

~~(1,2,2)~~

~~(0,1,4)~~

(0,2,4)

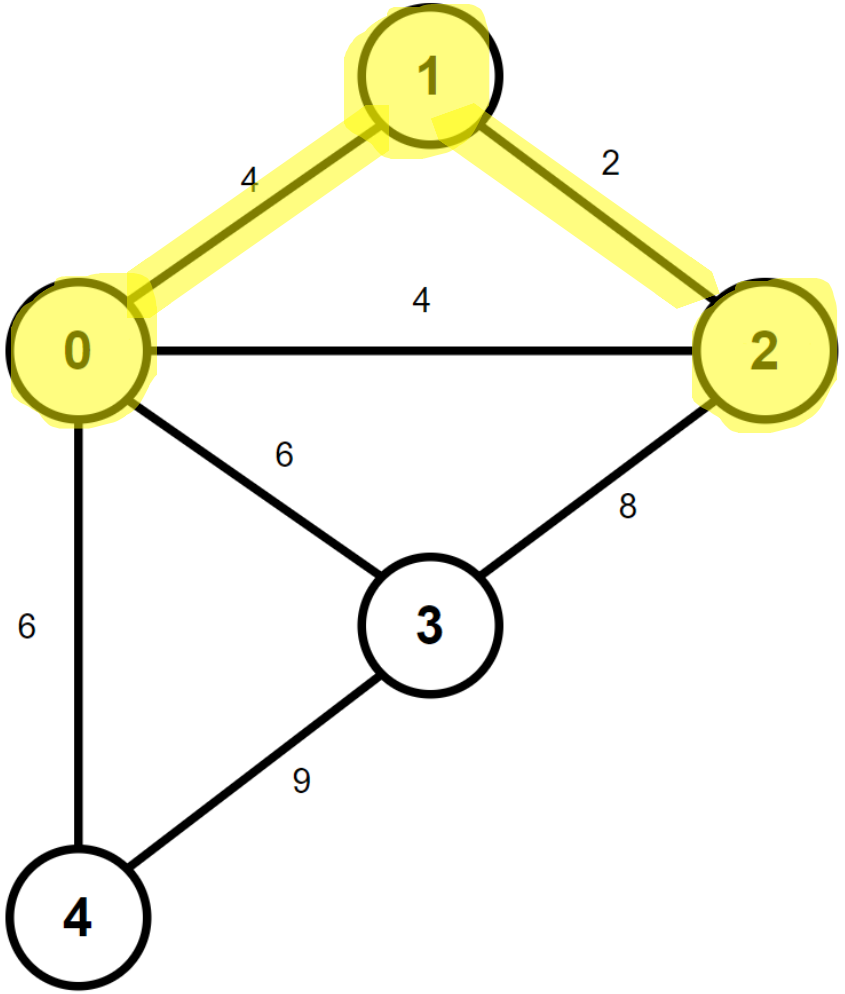
(0,3,6)

(0,4,6)

(2,3,8)

(3,4,9)

MST – Kruskal's Algorithm



Choose the minimum…

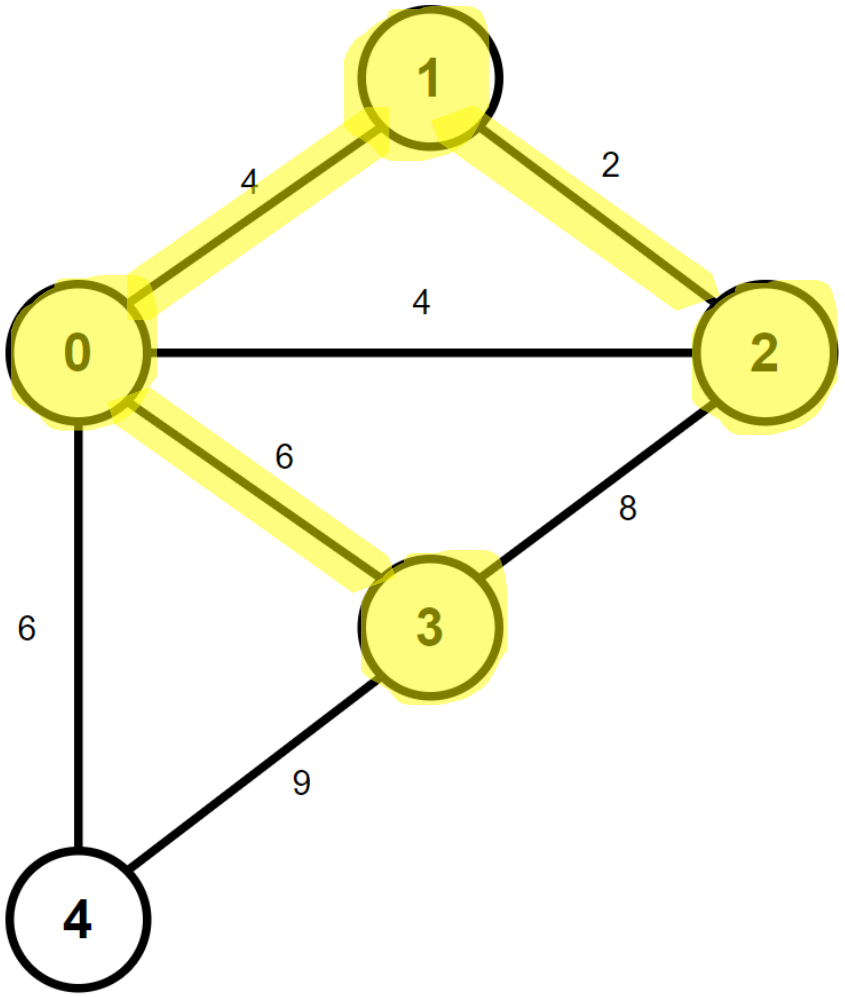
~~(1,2,2)~~
~~(0,1,4)~~
~~(0,2,4)~~
(0,3,6)
(0,4,6)
(2,3,8)
(3,4,9)

The next choice is

(0,2,4)

Vertex 0 and Vertex 2 are both already in our MST and adding an edge between them would create a cycle which is not allowed.

MST – Kruskal's Algorithm



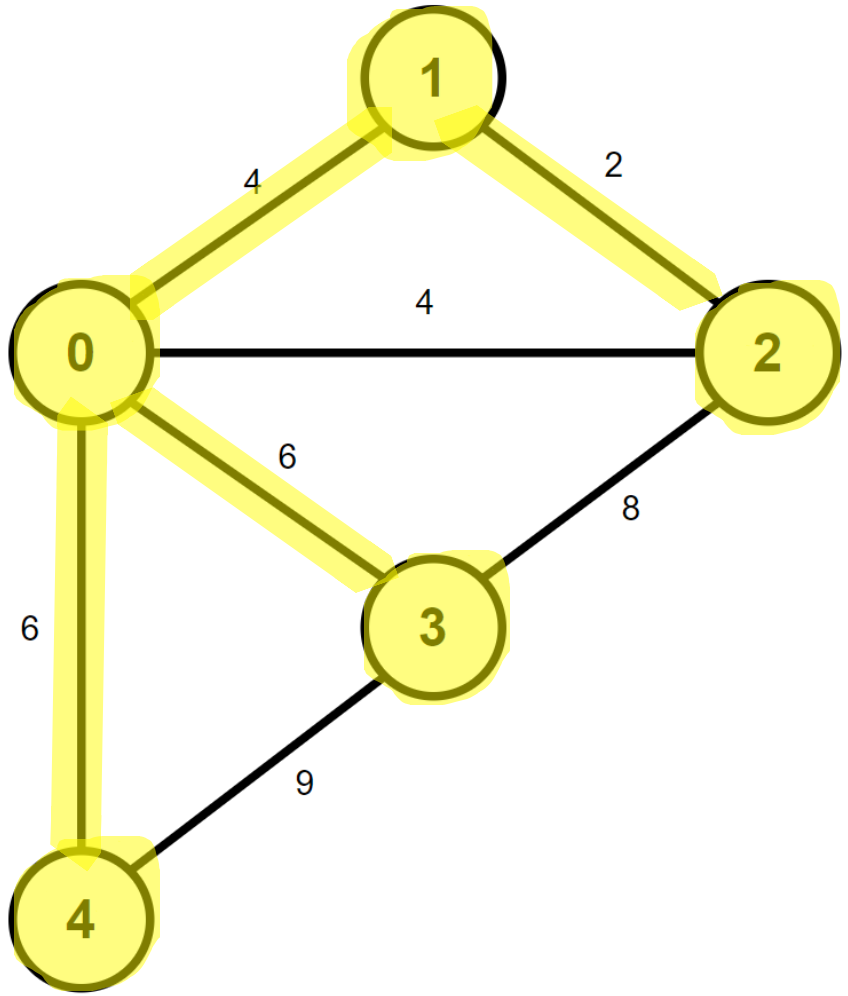
Choose the minimum…

~~(1,2,2)~~
~~(0,1,4)~~
~~(0,2,4)~~
~~(0,3,6)~~
(0,4,6)
(2,3,8)
(3,4,9)

The next choice is

(0,3,6)

MST – Kruskal's Algorithm



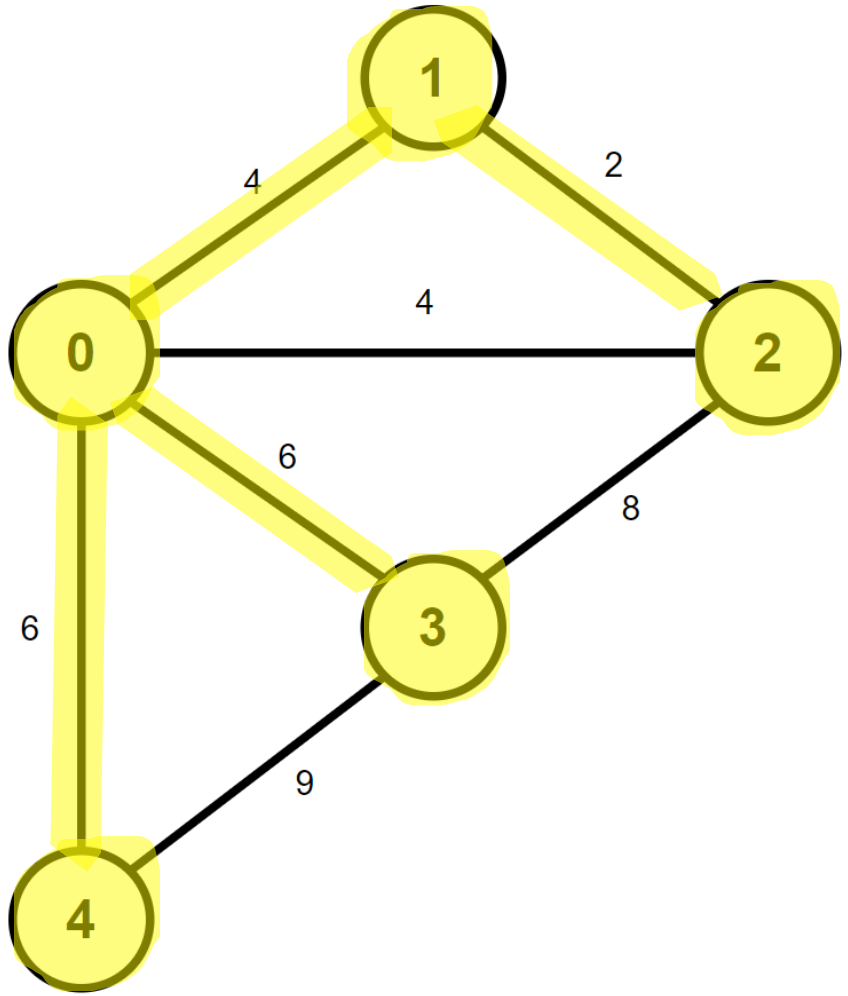
Choose the minimum…

~~(1,2,2)~~
~~(0,1,4)~~
~~(0,2,4)~~
~~(0,3,6)~~
~~(0,4,6)~~
(2,3,8)
(3,4,9)

The next choice is

(0,4,6)

MST – Kruskal's Algorithm



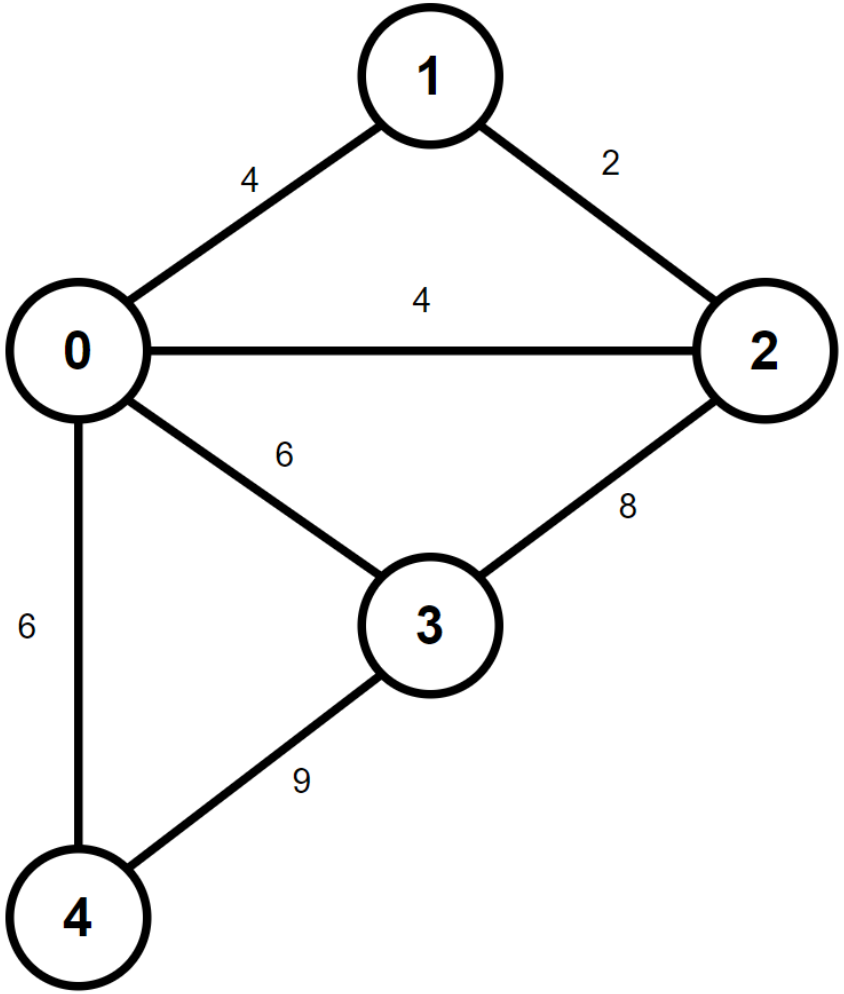
Choose the minimum…

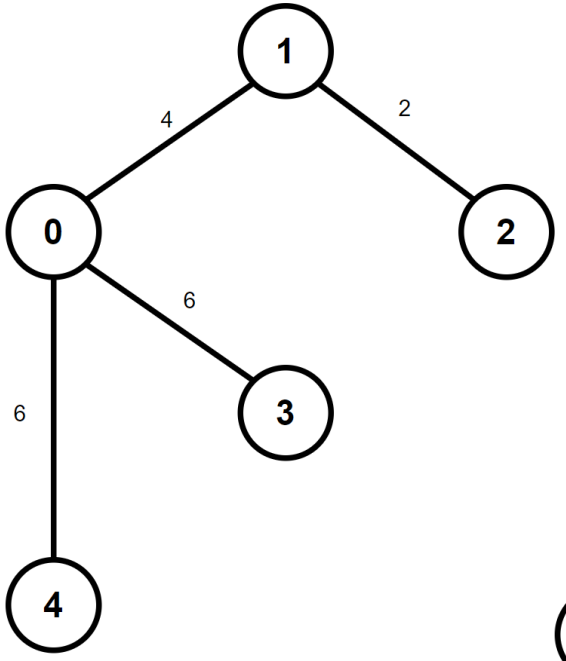
~~(1,2,2)~~
~~(0,1,4)~~
~~(0,2,4)~~
~~(0,3,6)~~
~~(0,4,6)~~
(2,3,8)
(3,4,9)

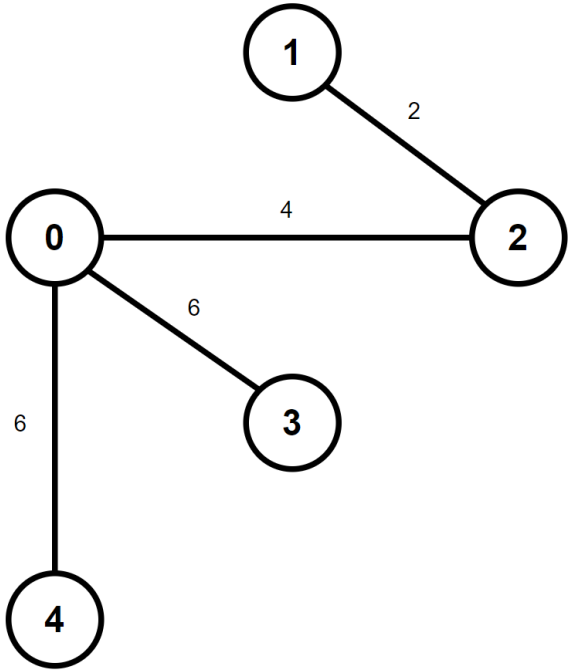
Now we could either recognize that we have 5 vertices in our graph and 5 vertices (and 4 edges) in our MST so we must be done.

Or we could pick (2,3,8) and discard it because Vertex 2 and 3 are already in the MST. Same for Vertex 3 and Vertex 4 in (3,4,9).

MST – Kruskal's Algorithm

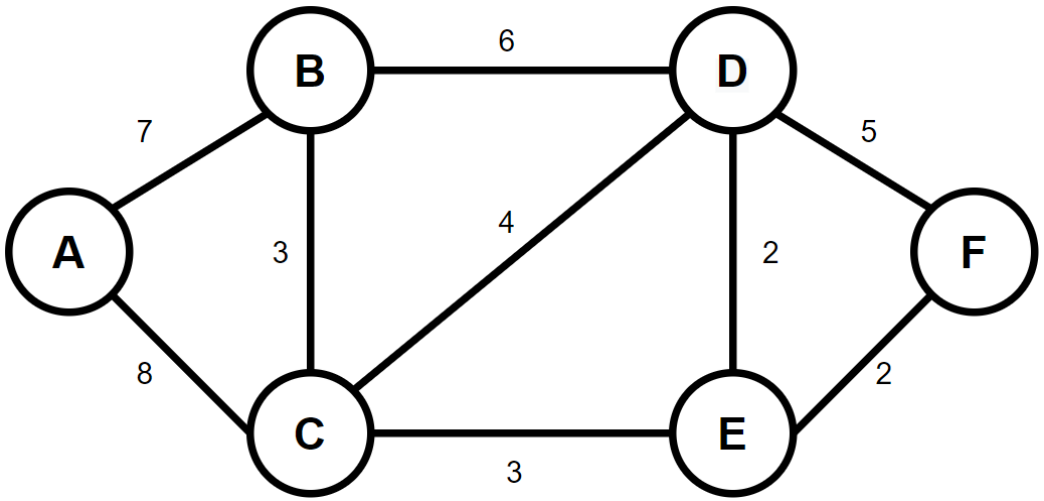


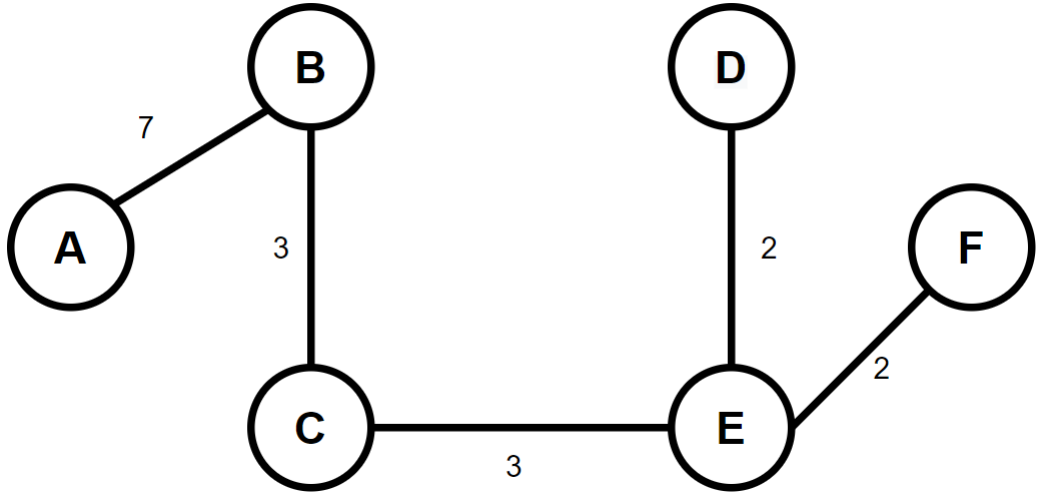




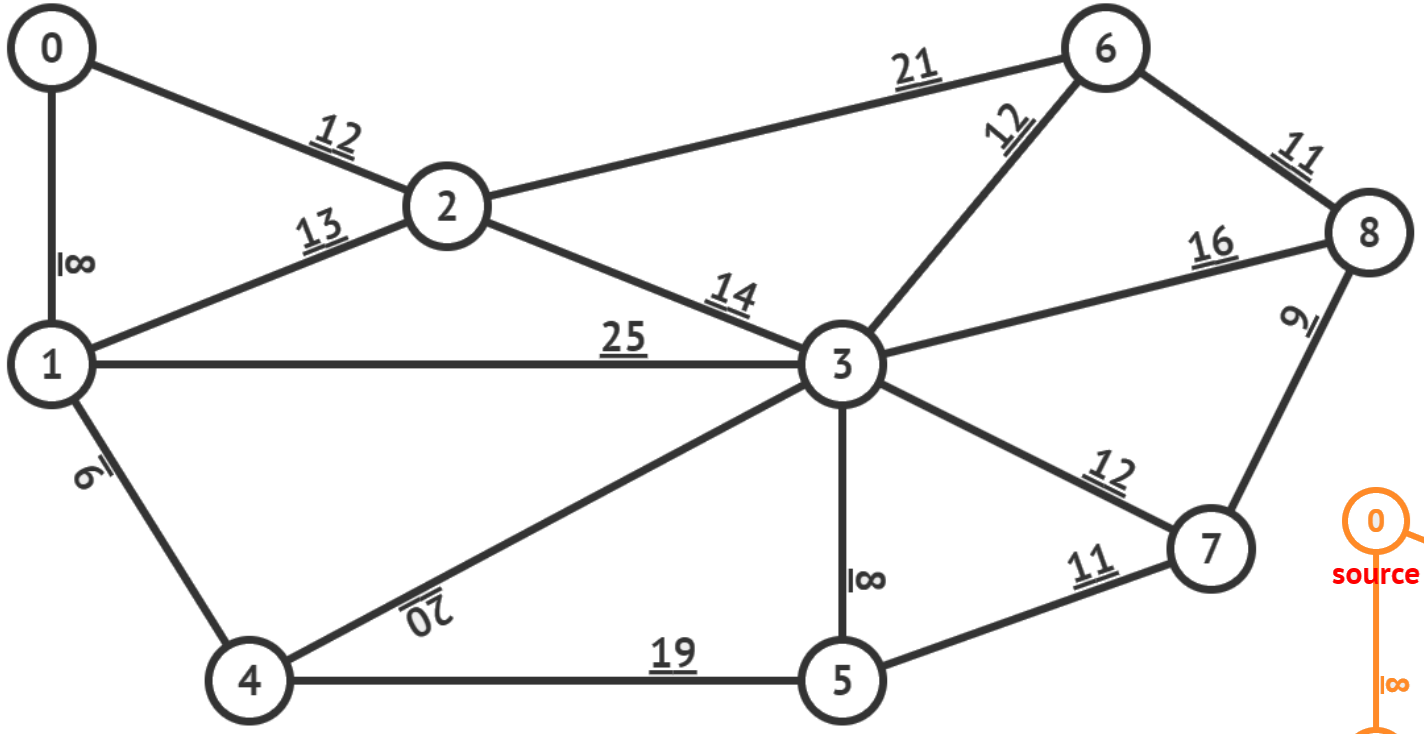
MST – Kruskal's Algorithm

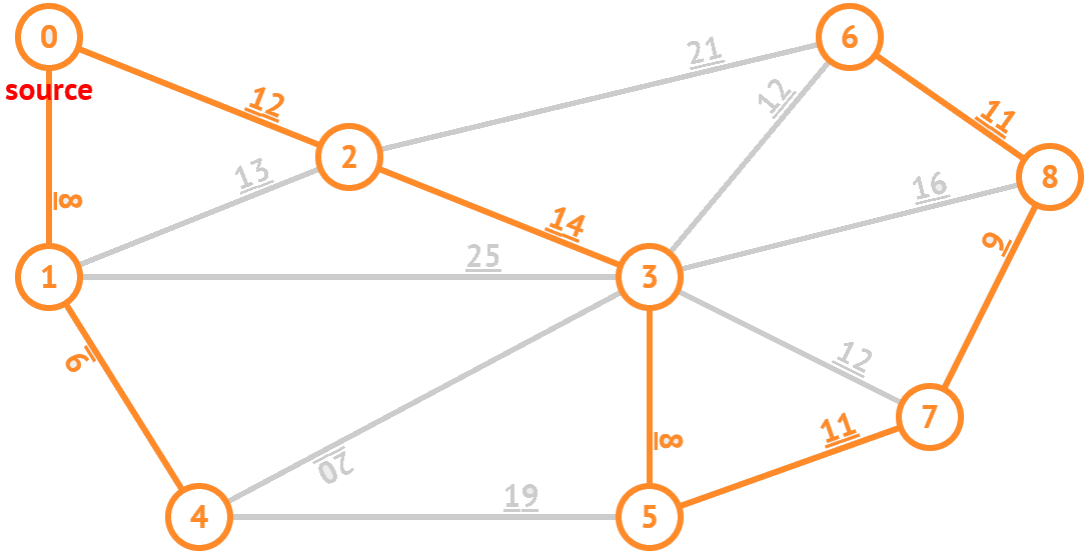
What is the Minimum Spanning Tree for this graph?



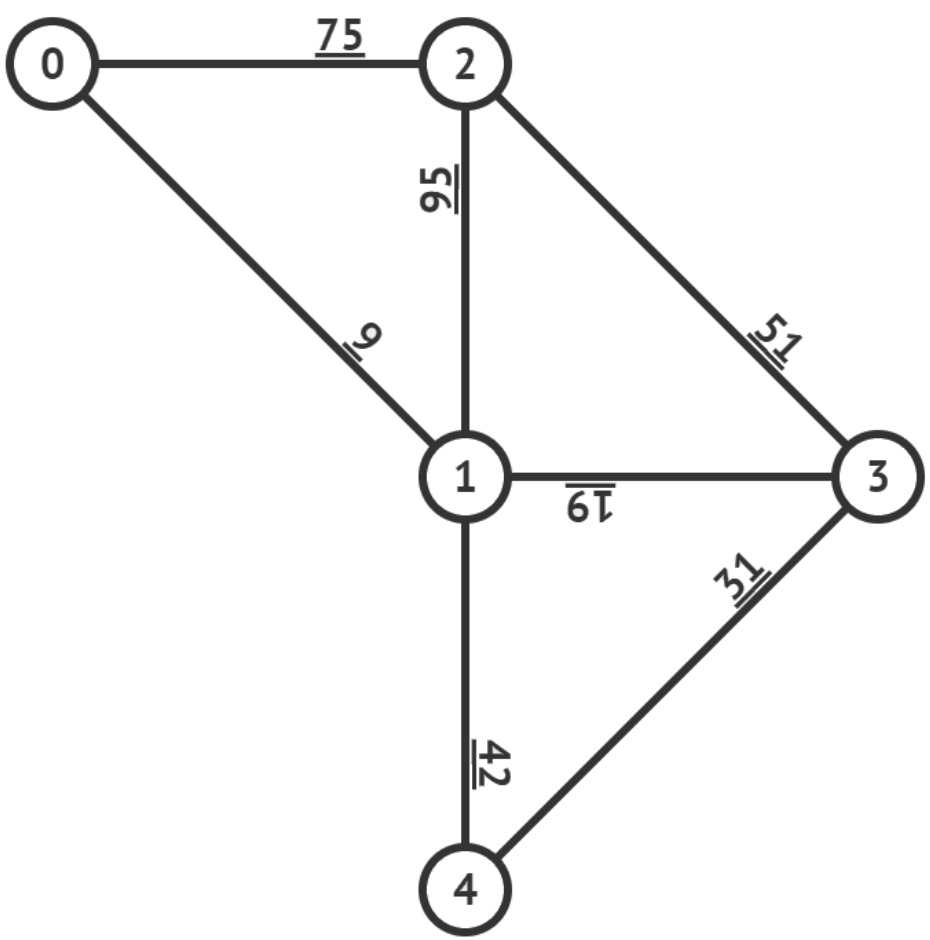


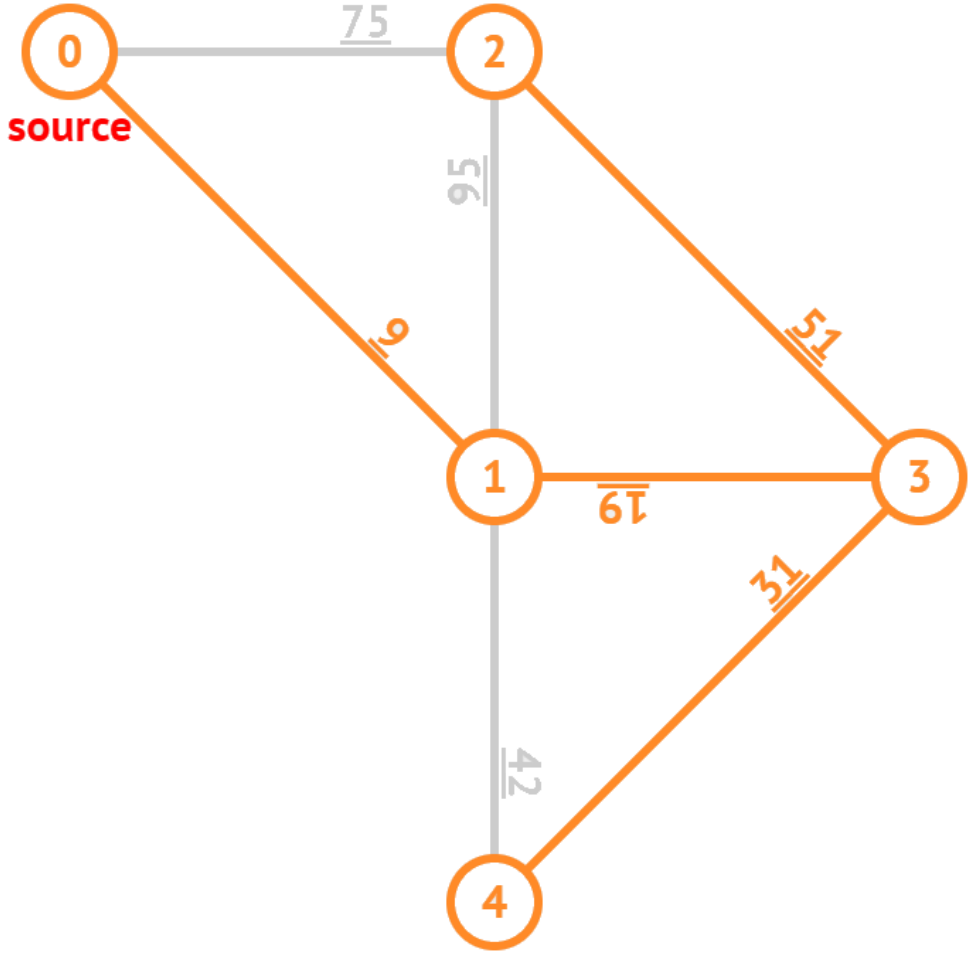
MST – Kruskal's Algorithm





MST – Kruskal's Algorithm





Prim vs Kruskal

You can’t really ask which algorithm is better in a given case without considering specific details of problem you are trying to solve.

Use Prim’s Algorithm when you have a graph with lots of edges.

Kruskal's Algorithm is general faster when you have a graph with fewer edges.

Prim vs Kruskal

Kruskal

builds a minimum spanning tree by adding one edge at a time. The next line is always the shortest (minimum weight) ONLY if it does NOT create a cycle.

Prims

builds a minimum spanning tree by adding one vertex at a time. The next vertex to be added is always the one nearest to a vertex already on the graph.

Prim vs Kruskal

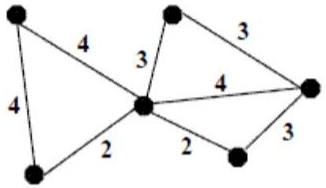
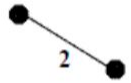
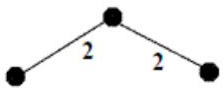
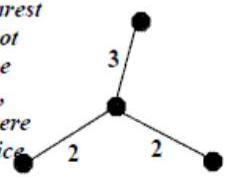
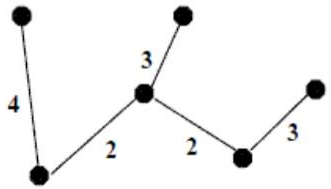
Prim always joins a "new" vertex to an "old" vertex, so that every stage is a tree.

Kruskal allows both "new" to "new" and "old" to "old" to get connected, so it risks creating a circuit and must check for them every time.

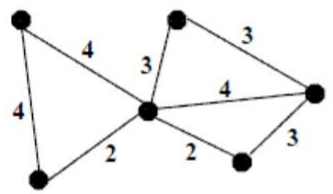
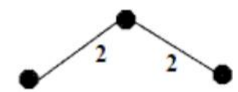
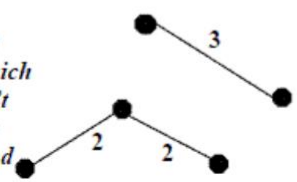
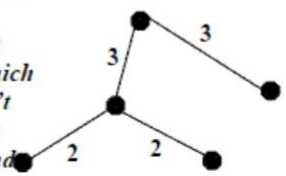
Kruskal has a larger complexity than Prim and could result in an incomplete/unconnected result if interrupted.

Prim vs Kruskal

Prim's Algorithm

1 *Given a network……* 	2 *Choose a vertex* 	3 *Choose the shortest edge from this vertex.* 
4 *Choose the nearest vertex not yet in the solution.* 	5 *Choose the next nearest vertex not yet in the solution, when there is a choice choose either.* 	6 *Repeat until you have a minimal spanning tree.* 

Kruskal's Algorithm

1 *Given a network……* 	2 *Choose the shortest edge (if there is more than one, choose any of the shortest)……..* 	3 *Choose the next shortest edge and add it……..* 
4 *Choose the next shortest edge which wouldn't create a cycle and add it.* 	5 *Choose the next shortest edge which wouldn't create a cycle and add it.* 	6 *Repeat until you have a minimal spanning tree.*