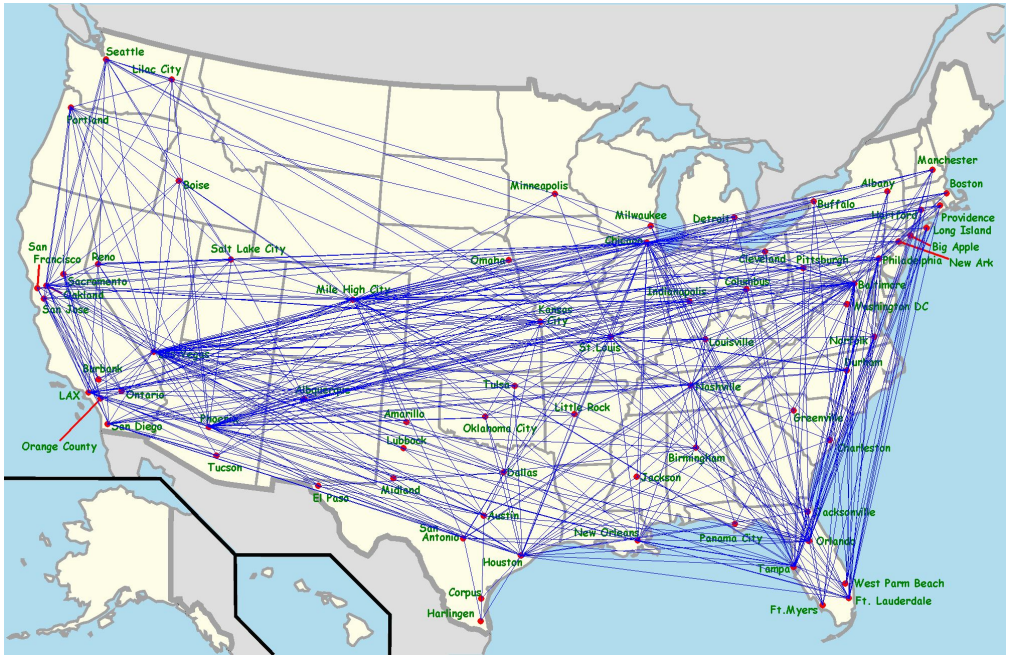
Dijkstra's Algorithm

UF Programming Team

Motivating Problem

Imagine that we have a graph of flights across the country, and each flight has a given cost and time.

We want to find a series of flights from city A to city B such that are cost (or time) is minimized.



Motivating Problem (cont.)

We can generalize this problem:

Given a graph **G** and a starting vertex **s**, what is the *shortest path* from **s** to any other vertex of **G**?

(*e.g.*, from the previous example, what is the shortest path from city A to any other city on the map?)

This problem is called the **Single-Source Shortest Paths** problem

Single-Source Shortest Path

In the flight example, the flights had costs and times from city to city, but that’s not always the case.

Imagine that we have a graph where the cost to go from one vertex to another vertex is 1, for all vertices.

We will look at how to find the SSSP of a graph with unweighted and weighted graphs.

SSSP on an *unweighted* graph

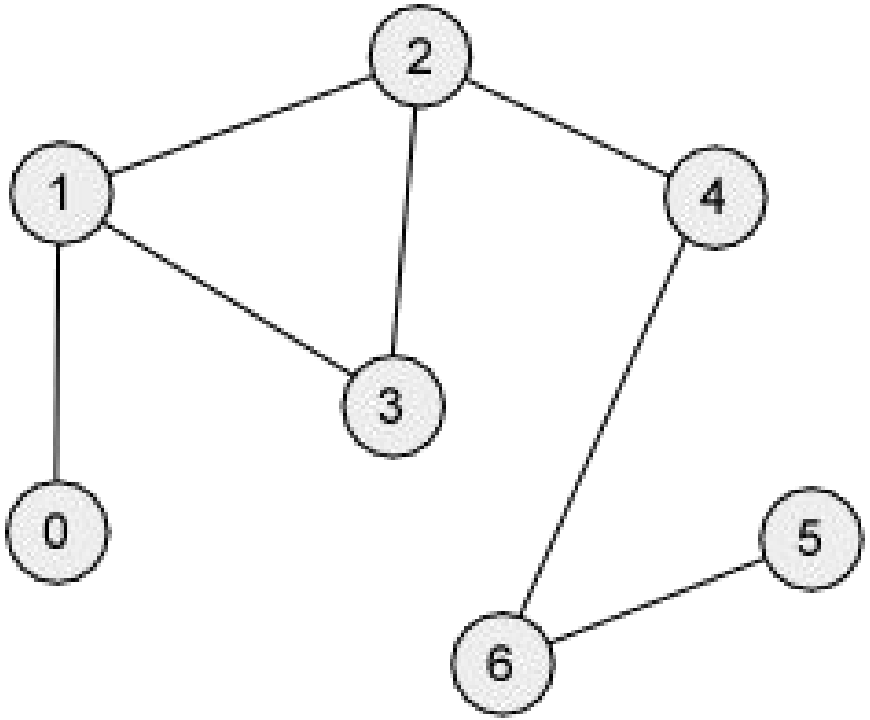
If the graph is unweighted (or if all of the edges have equal or constant weight), we can use the BFS algorithm in order to solve this problem.

BFS goes “level-by-level”, so it would visit all vertices that are 1 unit away first, and then 2 units away, and then 3 units away, etc.

Example: SSSP on an *unweighted* graph

Imagine that are starting vertex is 2

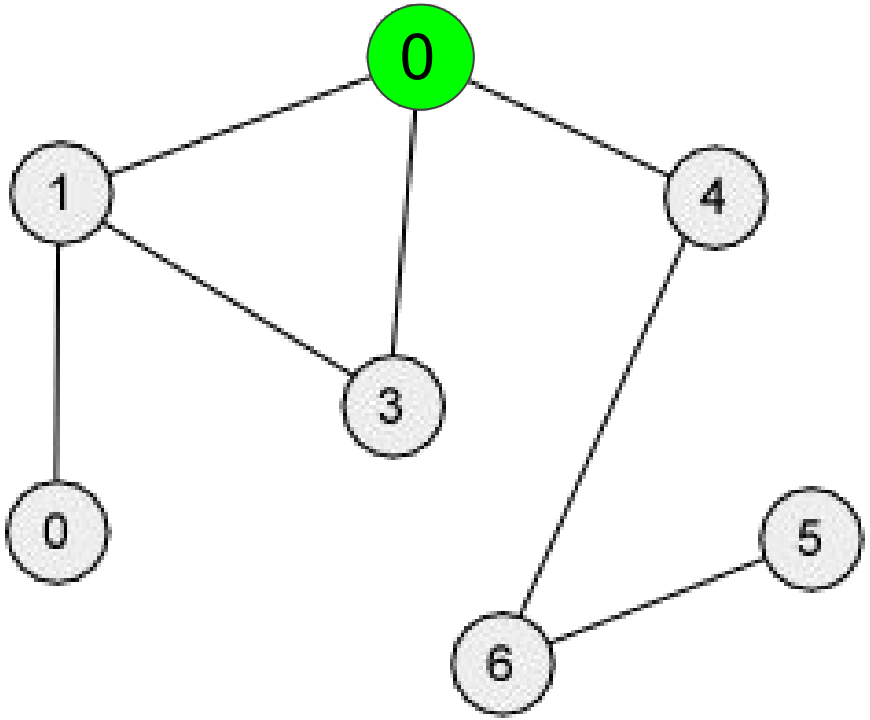
Because the graph is unweighted, we can use BFS to determine the distances to all other vertices



Example: SSSP on an *unweighted* graph (cont.)

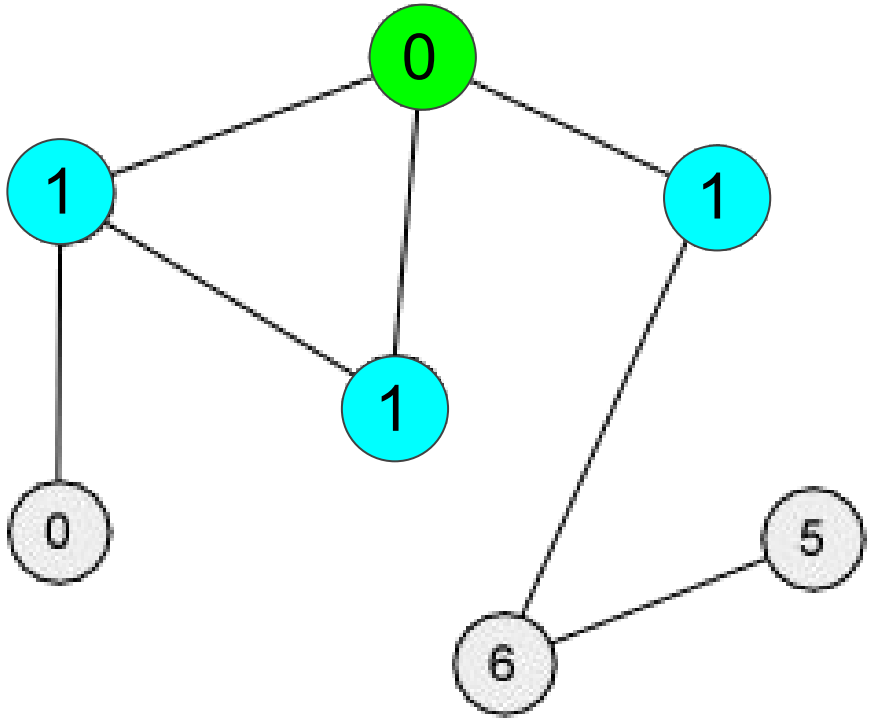
First we set our source vertex to 2

The values inside the colored vertices will indicate distance from the source



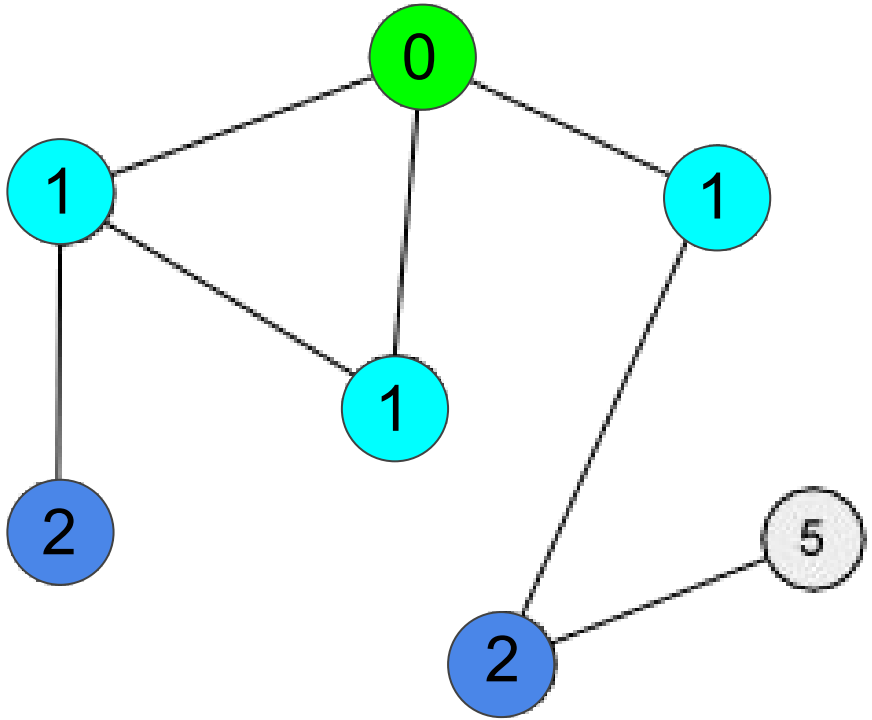
Example: SSSP on an *unweighted* graph (cont.)

Now we visit all of the neighbors of the source, which will be 1 unit away



Example: SSSP on an *unweighted* graph (cont.)

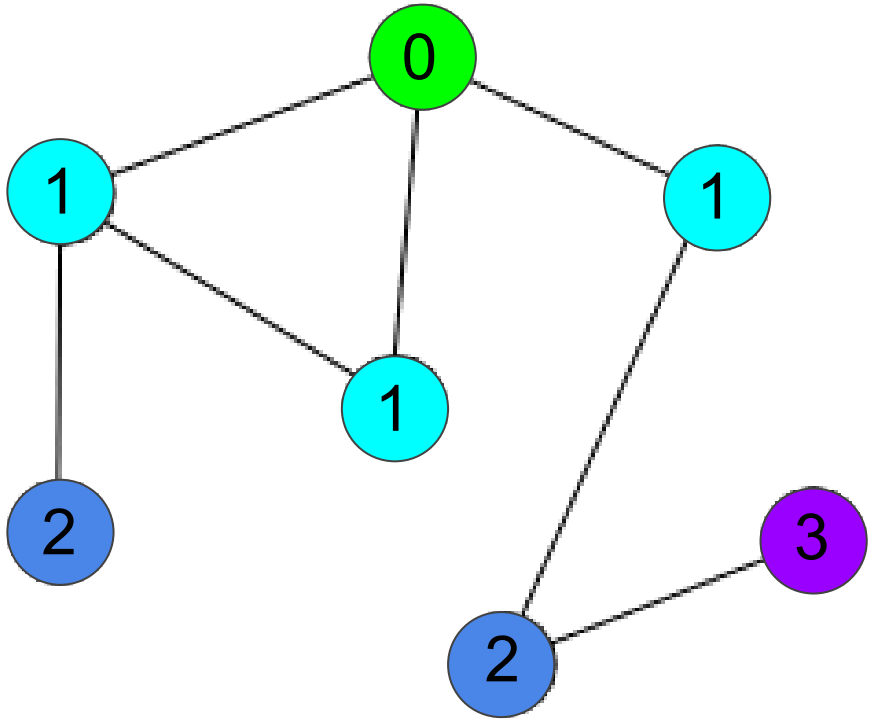
Now we visit all of the neighbors of the neighbors of the source, which will be 2 units away from the source



Example: SSSP on an *unweighted* graph (cont.)

Repeat the process until we are done

We now have all of the distances from the source to any other vertex!



SSSP on an *unweighted* graph

The fact that BFS visits vertices of a graph layer by layer from a source vertex turns BFS into a natural choice to solve the SSSP problem on unweighted graphs

Now what happens if the graph is *weighted*?

SSSP on a *weighted* graph

If the given graph is weighted, BFS no longer works

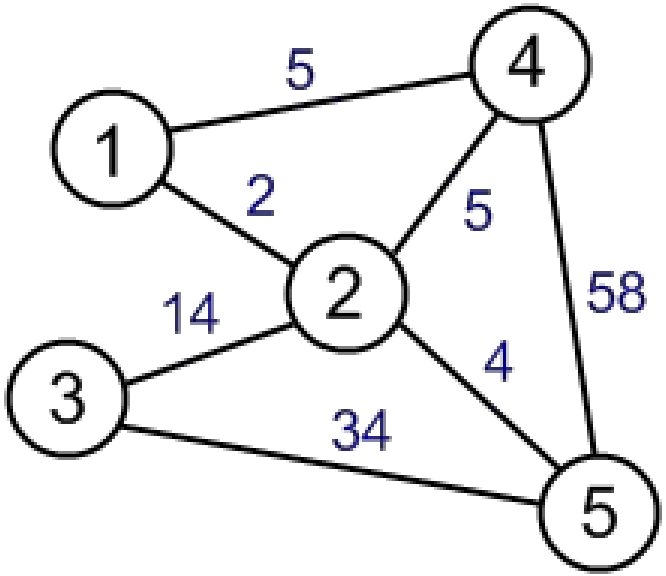
This is because there can be ‘longer’ paths from the source to a vertex, but has a smaller total cost than the ‘shorter’ path found by BFS.

SSSP on a *weighted* graph (cont.)

Imagine we have the following graph

What if we wanted to go from vertex 5 to vertex 4 using the BFS method?

It would immediately go to vertex 4 in 58 units, but there is a better path (5 → 2 → 4) that only costs 9 units



SSSP on a *weighted* graph (cont.)

To solve the SSSP problem on weighted graphs, we use a greedy algorithm: **Dijkstra's Algorithm**

There are several ways to implement this algorithm

→ **int[] distance**

→ **PriorityQueue<Vertex>**

Dijkstra's Algorithm

---

1) Give every vertex a tentative distance

→ the initial vertex will have a distance of zero

→ every other vertex will have a distance of infinity

2) Set the initial vertex as current; mark all other vertices unvisited

3) For the current vertex, consider all of its unvisited neighbors and calculated the tentative distance to each vertex; compare the newly calculated distance to the current assigned value and assign the smaller one

Dijkstra's Algorithm

---

4) When we are done considering all of the neighbors of the current vertex, mark the current vertex as visited

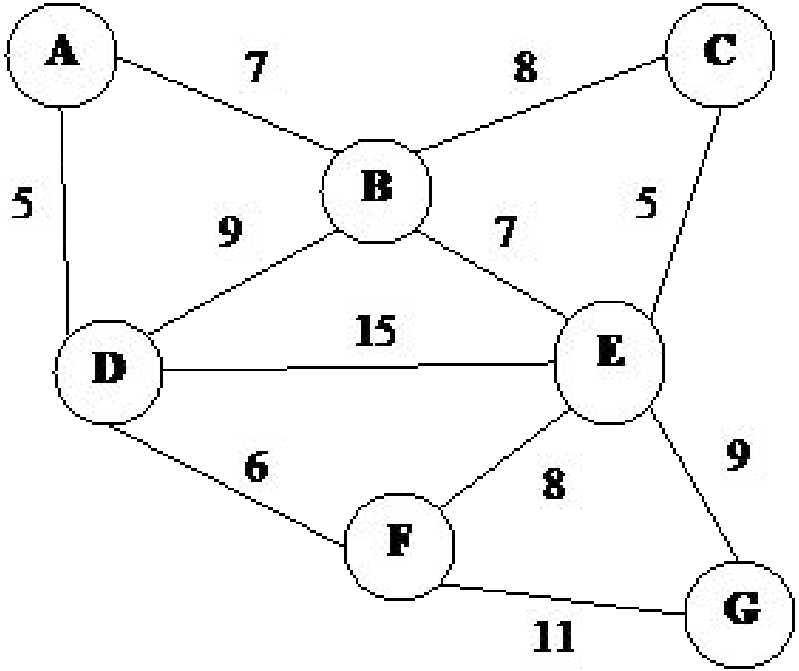
5) If the destination vertex has been marked visited, or if the smallest tentative distance among unvisited vertices is infinity, stop the algorithm

6) Select the unvisited vertex that is marked with the smallest tentative distance, and set it as the new current vertex, and then go back to step 3

Example: Dijkstra's Algorithm

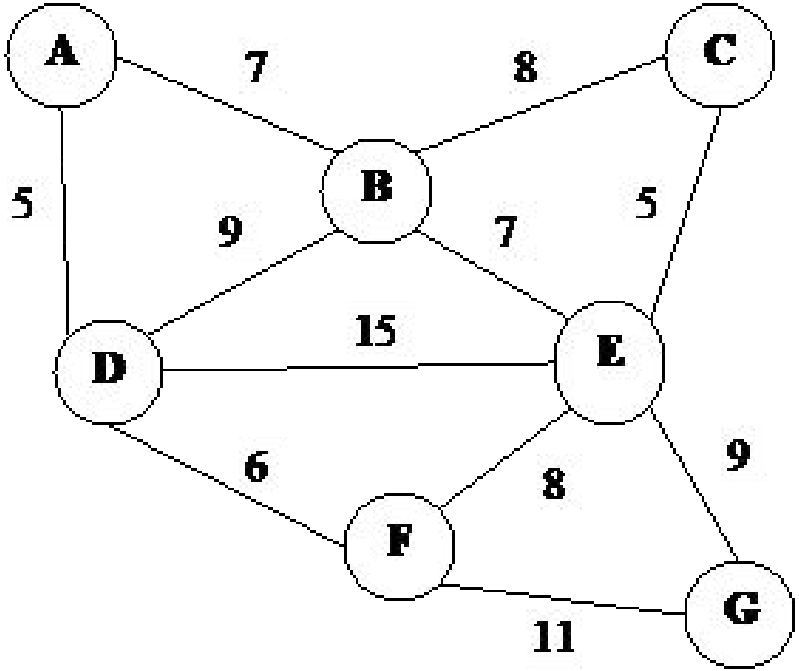
Imagine that we have the following graph, and we want to travel from vertex **A** to vertex **G**

We'll used a table to keep track of the tentative distances up to each vertex, and to see which vertices have been visited thus far



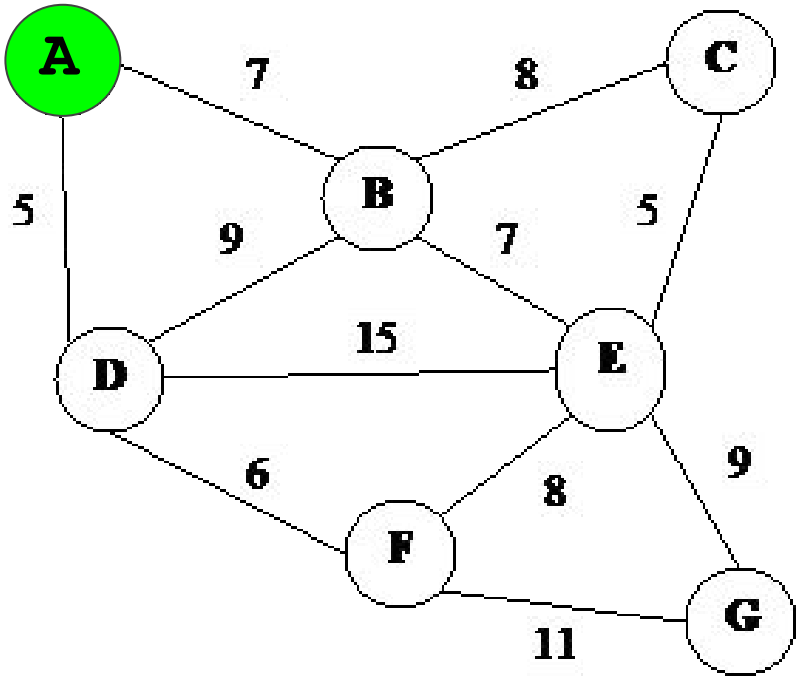
Example: Dijkstra's Algorithm

Vertex	*Distance*	*Visited*
A	INF	FALSE
B	INF	FALSE
C	INF	FALSE
D	INF	FALSE
E	INF	FALSE
F	INF	FALSE
G	INF	FALSE



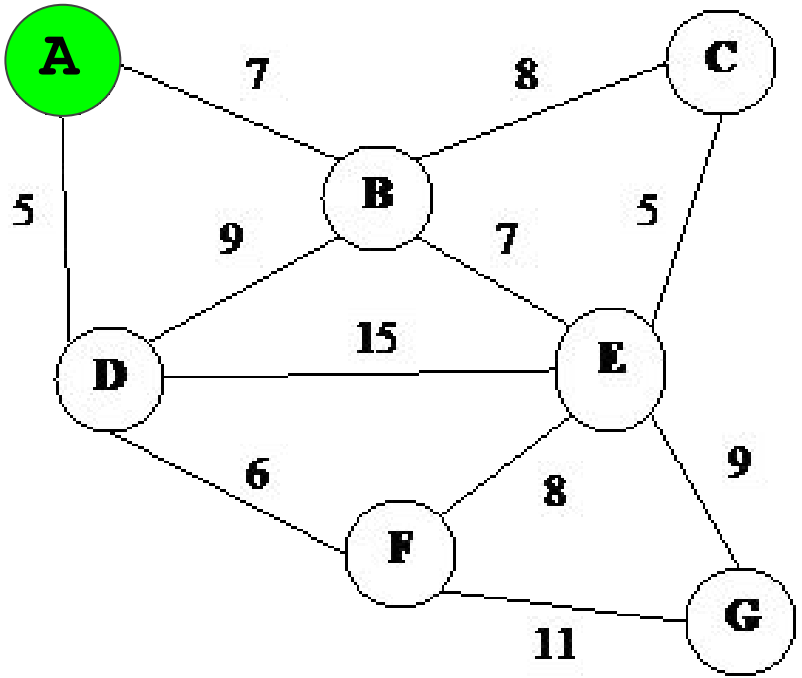
Example: Dijkstra's Algorithm

Vertex	*Distance*	*Visited*
A	**0**	**TRUE**
B	INF	FALSE
C	INF	FALSE
D	INF	FALSE
E	INF	FALSE
F	INF	FALSE
G	INF	FALSE



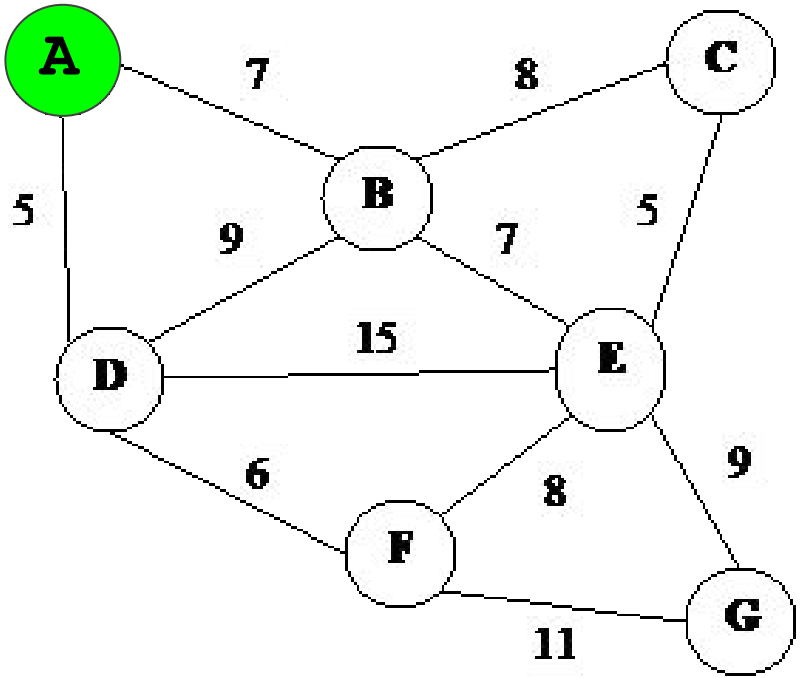
Example: Dijkstra's Algorithm

Vertex	*Distance*	*Visited*
A	**0**	**TRUE**
B	**0 + 7 <** INF	FALSE
C	INF	FALSE
D	**0 + 5 <** INF	FALSE
E	INF	FALSE
F	INF	FALSE
G	INF	FALSE



Example: Dijkstra's Algorithm

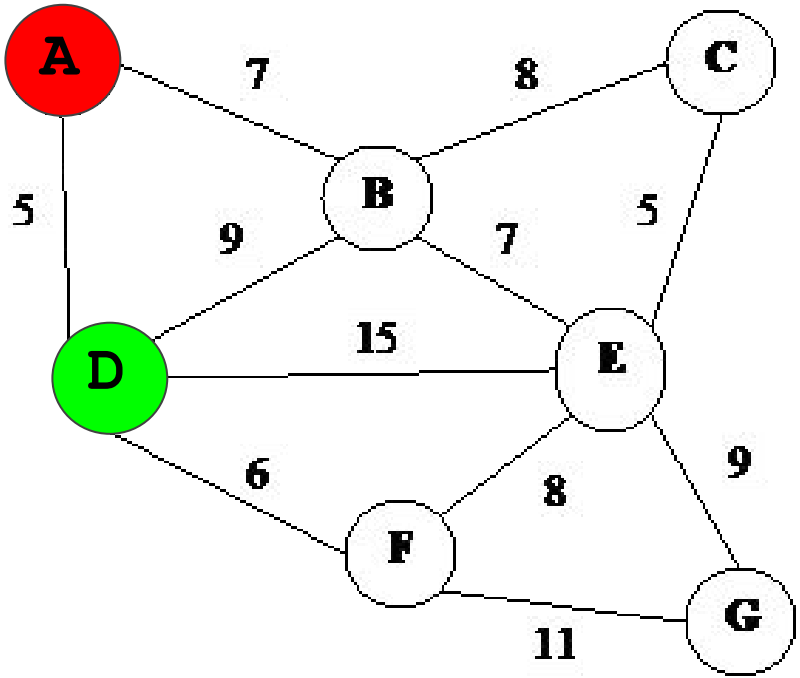
Vertex	*Distance*	*Visited*
A	**0**	**TRUE**
B	7	FALSE
C	INF	FALSE
D	5	FALSE
E	INF	FALSE
F	INF	FALSE
G	INF	FALSE



Example: Dijkstra's Algorithm

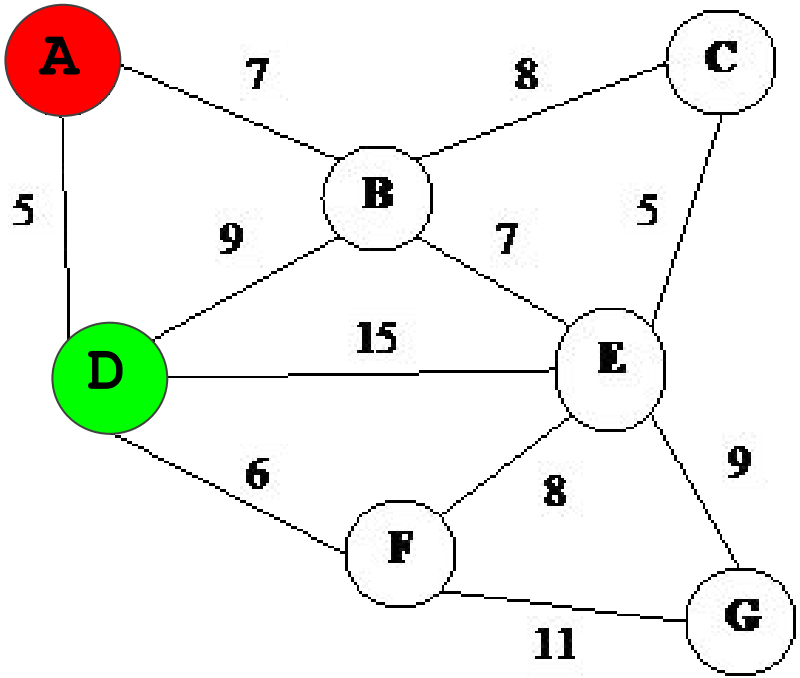
The distance up to vertex D is the smallest out of all **non-visited** vertices, so that's our new current

Vertex	*Distance*	*Visited*
A	0	TRUE
B	7	FALSE
C	INF	FALSE
D	**5**	**TRUE**
E	INF	FALSE
F	INF	FALSE
G	INF	FALSE



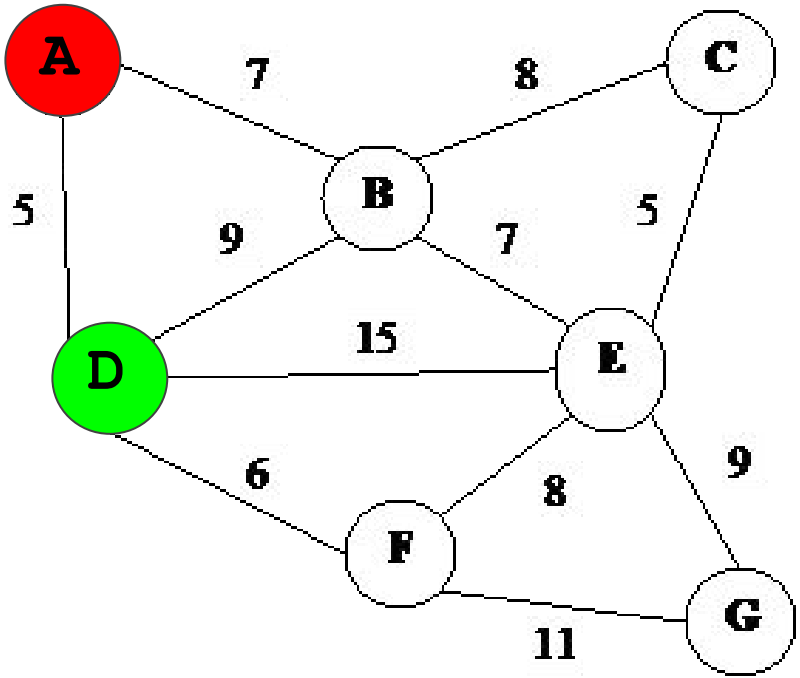
Example: Dijkstra's Algorithm

Vertex	*Distance*	*Visited*
A	0	TRUE
B	**5 + 9 <** 7	FALSE
C	INF	FALSE
D	**5**	**TRUE**
E	**5 + 15 <** INF	FALSE
F	**5 + 6 <** INF	FALSE
G	INF	FALSE



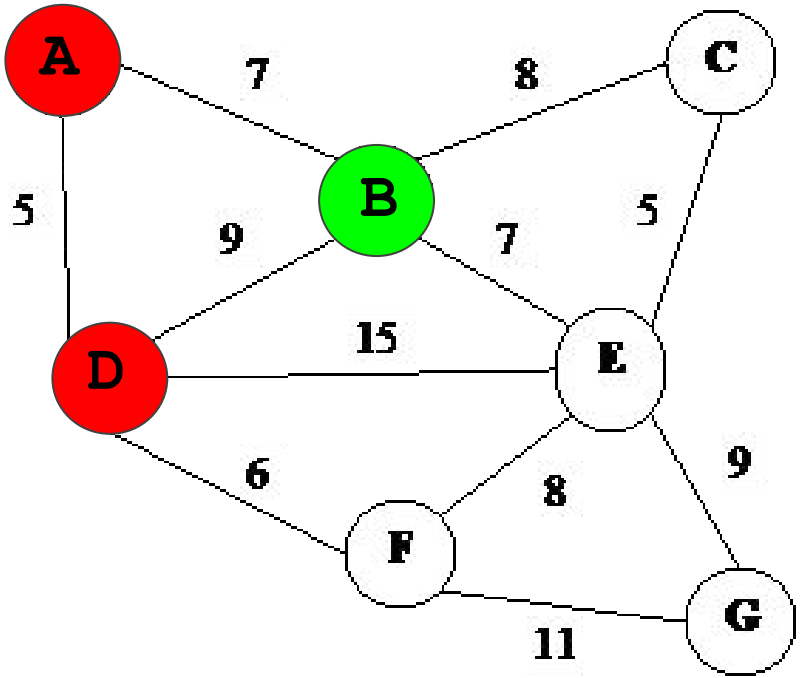
Example: Dijkstra's Algorithm

Vertex	*Distance*	*Visited*
A	0	TRUE
B	7	FALSE
C	INF	FALSE
D	**5**	**TRUE**
E	20	FALSE
F	11	FALSE
G	INF	FALSE



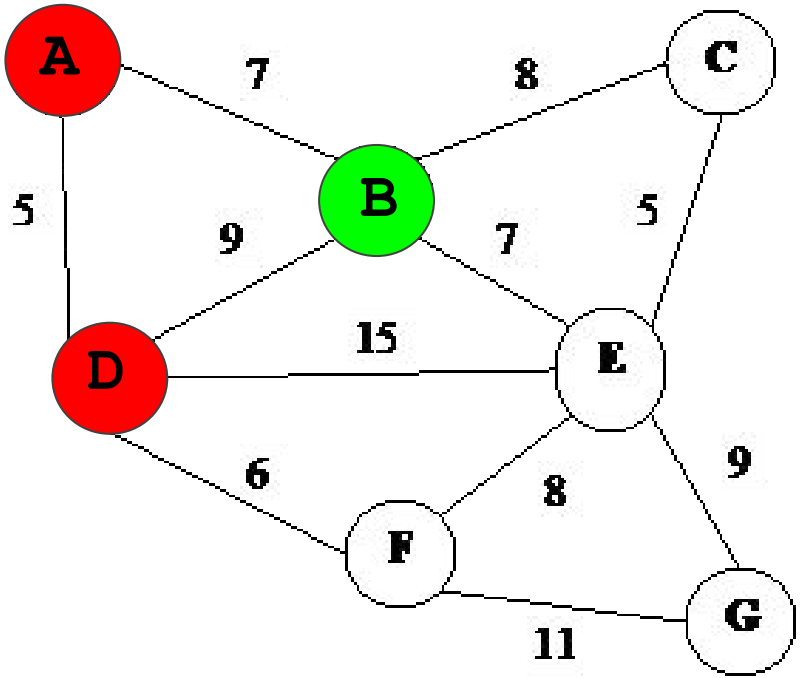
Example: Dijkstra's Algorithm

Vertex	*Distance*	*Visited*
A	0	TRUE
B	**7**	**TRUE**
C	INF	FALSE
D	5	TRUE
E	20	FALSE
F	11	FALSE
G	INF	FALSE



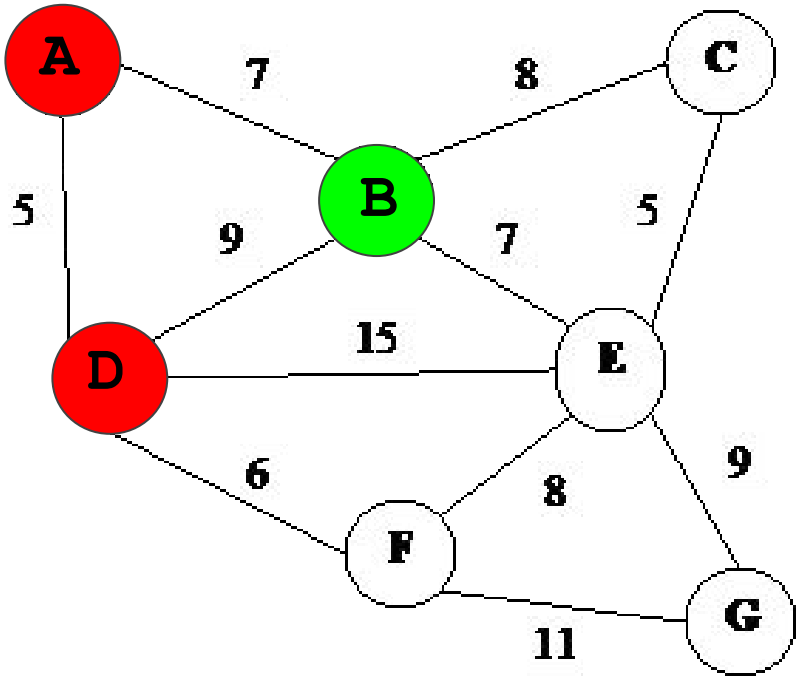
Example: Dijkstra's Algorithm

Vertex	*Distance*	*Visited*
A	0	TRUE
B	**7**	**TRUE**
C	**7 + 8 <** INF	FALSE
D	5	TRUE
E	**7 + 7 <** 20	FALSE
F	11	FALSE
G	INF	FALSE



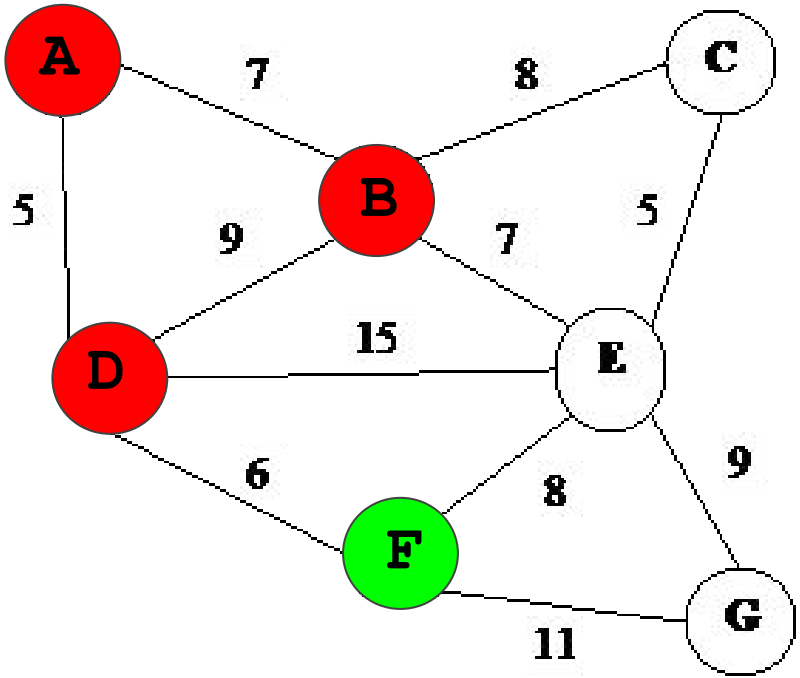
Example: Dijkstra's Algorithm

Vertex	*Distance*	*Visited*
A	0	TRUE
B	**7**	**TRUE**
C	15	FALSE
D	5	TRUE
E	14	FALSE
F	11	FALSE
G	INF	FALSE



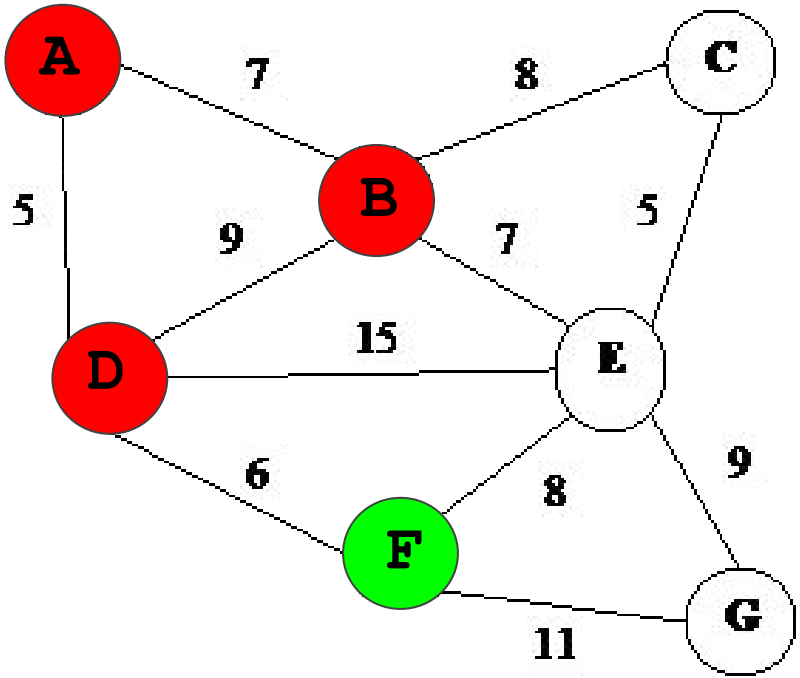
Example: Dijkstra's Algorithm

Vertex	*Distance*	*Visited*
A	0	TRUE
B	7	TRUE
C	15	FALSE
D	5	TRUE
E	14	FALSE
F	**11**	**TRUE**
G	INF	FALSE



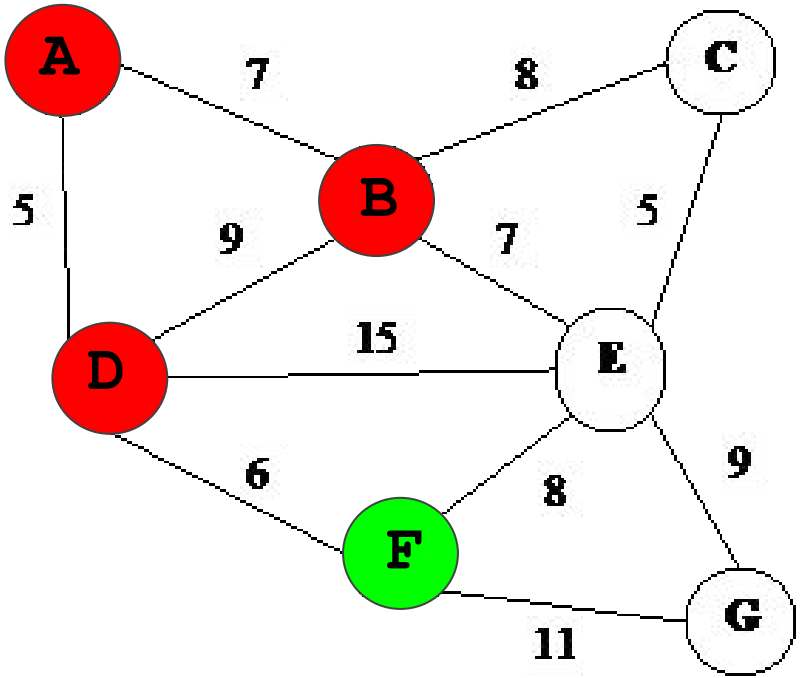
Example: Dijkstra's Algorithm

Vertex	*Distance*	*Visited*
A	0	TRUE
B	7	TRUE
C	15	FALSE
D	5	TRUE
E	**11 + 8 <** 14	FALSE
F	**11**	**TRUE**
G	**11 + 11 <**INF	FALSE



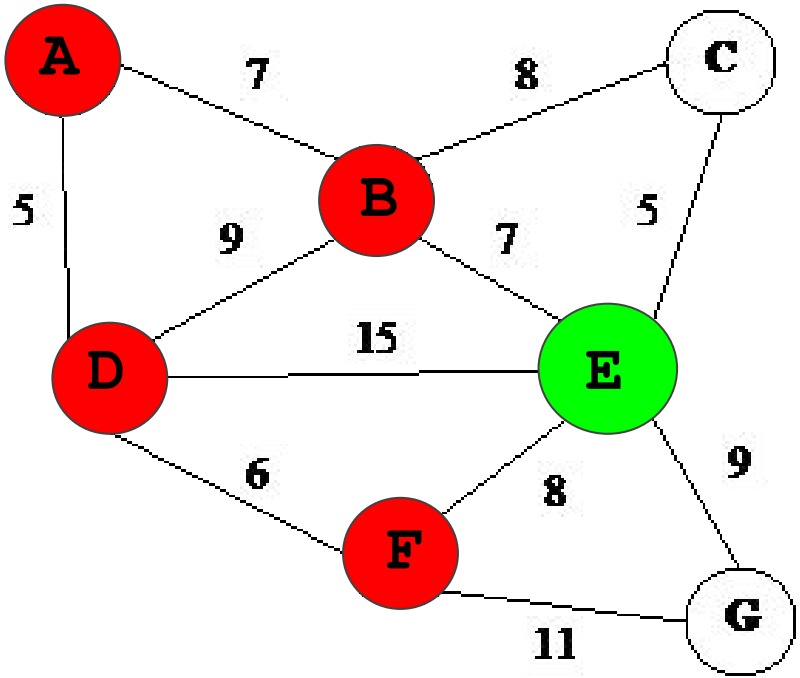
Example: Dijkstra's Algorithm

Vertex	*Distance*	*Visited*
A	0	TRUE
B	7	TRUE
C	15	FALSE
D	5	TRUE
E	14	FALSE
F	**11**	**TRUE**
G	22	FALSE



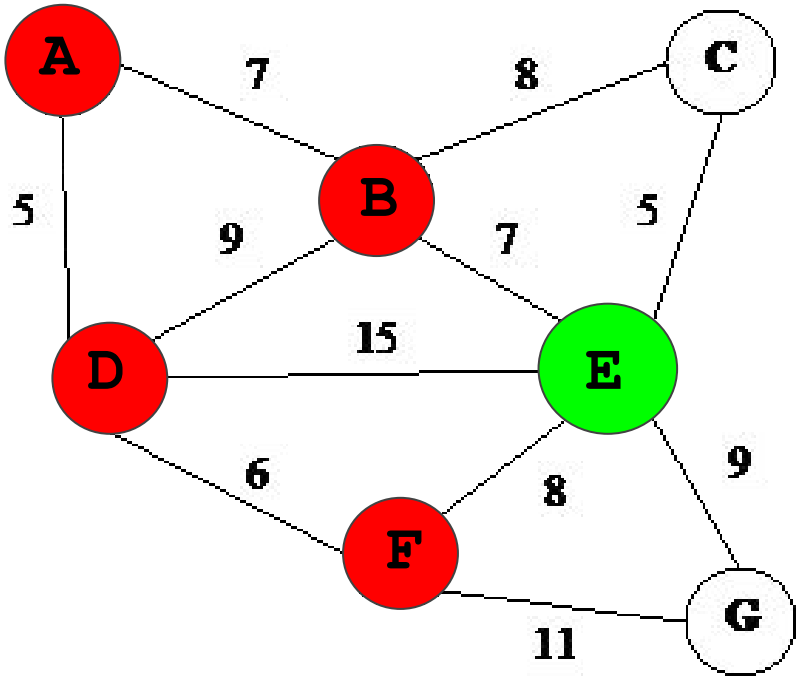
Example: Dijkstra's Algorithm

Vertex	*Distance*	*Visited*
A	0	TRUE
B	7	TRUE
C	15	FALSE
D	5	TRUE
E	**14**	**TRUE**
F	11	TRUE
G	22	FALSE



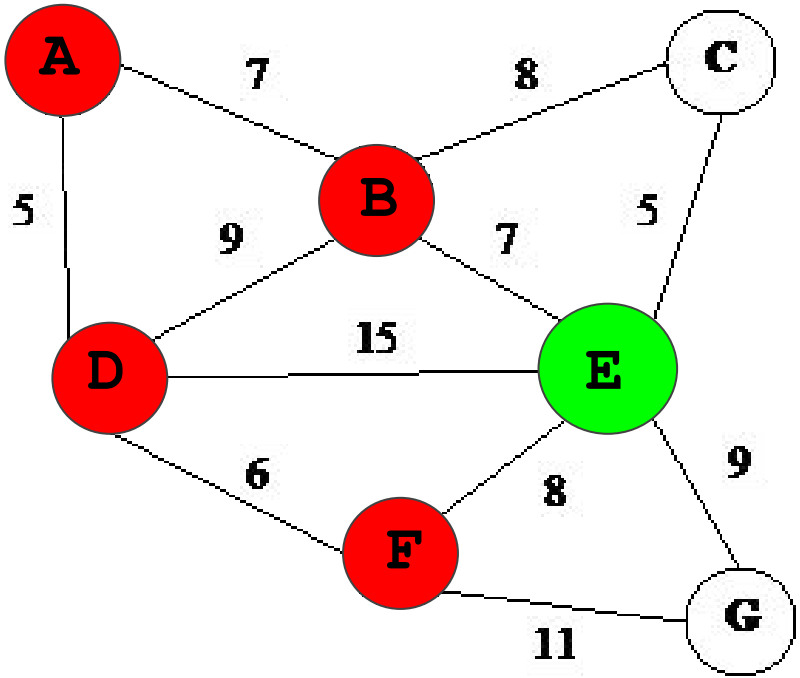
Example: Dijkstra's Algorithm

Vertex	*Distance*	*Visited*
A	0	TRUE
B	7	TRUE
C	**14 + 5 <** 15	FALSE
D	5	TRUE
E	**14**	**TRUE**
F	11	TRUE
G	**14 + 9 <** 22	FALSE



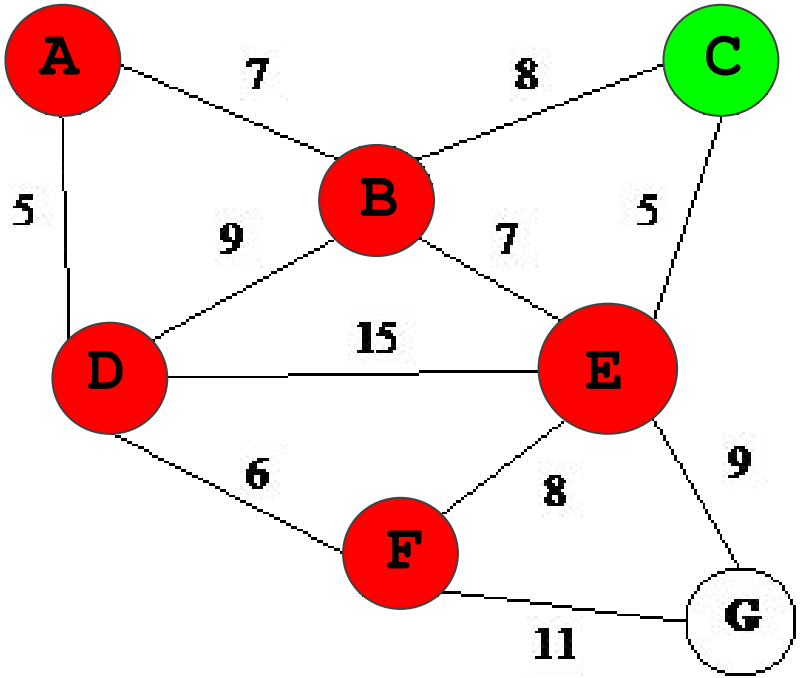
Example: Dijkstra's Algorithm

Vertex	*Distance*	*Visited*
A	0	TRUE
B	7	TRUE
C	15	FALSE
D	5	TRUE
E	**14**	**TRUE**
F	11	TRUE
G	22	FALSE



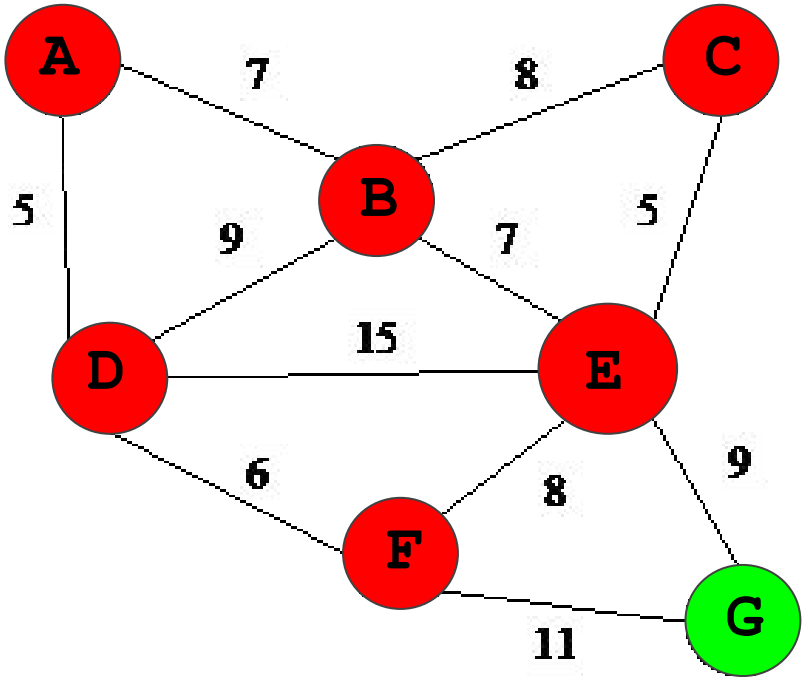
Example: Dijkstra's Algorithm

Vertex	*Distance*	*Visited*
A	0	TRUE
B	7	TRUE
C	**15**	**TRUE**
D	5	TRUE
E	14	TRUE
F	11	TRUE
G	22	FALSE



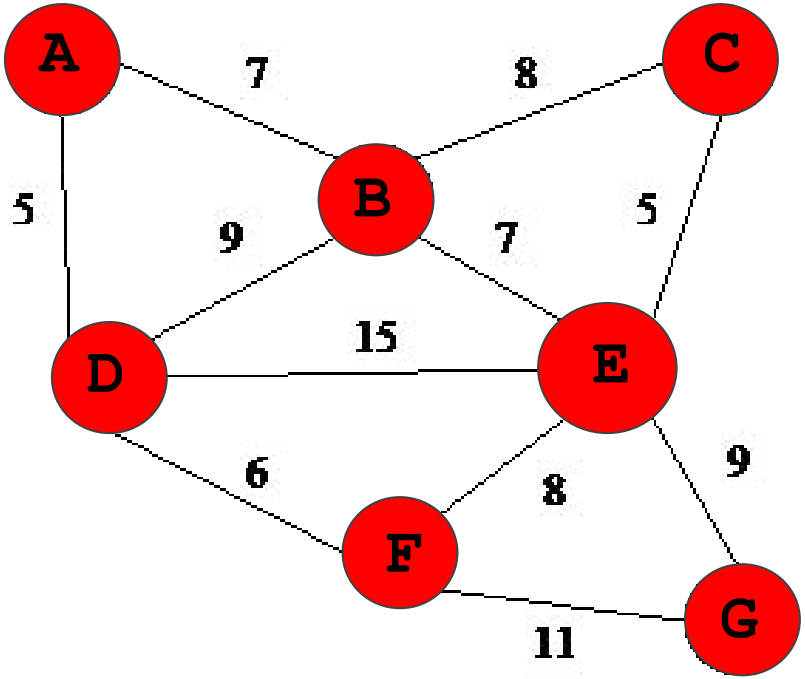
Example: Dijkstra's Algorithm

Vertex	*Distance*	*Visited*
A	0	TRUE
B	7	TRUE
C	15	TRUE
D	5	TRUE
E	14	TRUE
F	11	TRUE
G	**22**	**TRUE**



Example: Dijkstra's Algorithm

Vertex	*Distance*	*Visited*
A	0	TRUE
B	7	TRUE
C	15	TRUE
D	5	TRUE
E	14	TRUE
F	11	TRUE
G	22	TRUE



When we mark vertex **G** as visited, we can stop the algorithm since we have reached our destination.

We now know the minimum distance it takes to get there from vertex **A**!

Dijkstra's Algorithm (cont.)

The previous example implemented Dijkstra's Algorithm using the ***dist[]*** method, but what if we want to go about using the ***PriorityQueue<Vertex>*** method?

Remember that as you add items to a *PriorityQueue*, they are automatically sorted within the queue, so that you are given the "smallest" item when you call **priorityQueue.remove()**

First we need to define our class *Vertex* and what attributes it will contain

Dijkstra's Algorithm using a PriorityQueue

```
class Vertex implements Comparable<Vertex> {
    int id, dist;
    List<Integer> neighbors;

    public Vertex( int id ) {
        this.id = id;
        this.dist = INF;
        this.neighbors = new ArrayList<Integer>();
    }

// more code on next slide
```

Dijkstra's Algorithm using a PriorityQueue (cont.)

```
    public int compareTo( Vertex other ) {
        if ( this.dist < other.dist) { // if this vetex's dist is smaller than
            return -1;                 // the other vertex's dist, return -1
        } else {
            return 1;
        }
    }
} // end of class Vertex
```

Dijkstra's Algorithm using a PriorityQueue (cont.)

Why does *Vertex* implement *Comparable<Vertex>* and what does this mean?

We want to be able to sort the vertices by their tentative distance so that it will be easy to find the next vertex to visit

Remember that when we used ***dist[]*** to keep track of distances, we had to look through **all** tentative distances to see which vertex to visit; with this method, we just remove the next vertex from the *PriorityQueue*, and it will be guaranteed to have the smallest distance!

Dijkstra's Algorithm using a PriorityQueue (cont.)

Let's break down the attributes of *Vertex*

- **int id**
  - this is a quick and easy way to determine the location of a vertex
- **List<Integer> neighbors**
  - gives us a list of **id**s that **this vertex** has an edge to
- **int dist**
  - the most important attribute (remember that we are using this attribute to sort the vertices!)
  - gives us the tentative distance from the source to **this vertex**

Implementing Dijkstra's Algorithm

Now that we have setup our *Vertex* class, how do we use it to implement Dijkstra's Algorithm?

Let's start by creating all of the vertices of a graph and storing them into an array, **graph.**

```
Vertex[] graph = new Vertex[numVertices];

for ( int i = 0; i < numVertices; i++ ) {

    graph[i] = new Vertex( i );

}
```

Implementing Dijkstra's Algorithm (cont.)

How do we add an edge from **vertex i** to **vertex j**?

```
graph[i].neighbors.add( j );
```

We get the vertex with an **id** of **i**, get the list of neighbors of that vertex **i**, and finally add the vertex **j** to that list

Implementing Dijkstra's Algorithm (cont.)

We have set up our vertices, added the edges, so now let's implement Dijkstra's

We know the **dist** of our source (starting) vertex is 0, so we can assign that value and add the *Vertex* to the *PriorityQueue*

```
vertex[source].dist = 0;

priorityQueue.add( vertex[source] );
```

Implementing Dijkstra's Algorithm (cont.)

Just like BFS, we are repeating steps of an algorithm until the queue is empty. This makes sense for Dijkstra's since we want to keep repeating the process until we either

1) run out of vertices (queue is empty), or
2) reach our destination vertex

Now that we have our stopping condition, we just have to worry about the inside of the loop

Implementing Dijkstra's Algorithm (cont.)

```
while ( !priorityQueue.isEmpty() ) { // while we still have vertices to visit

    Vertex current = priorityQueue.remove(); // get the Vertex with the smallest
                                             // tentative distance

    if ( current.id == destination ) break; // break if we are done!

    List<Integer> neighbors = current.neighbors; // get the list of ids of the
                                                 // current vertex's neighbors

    // more code on the next slide
```

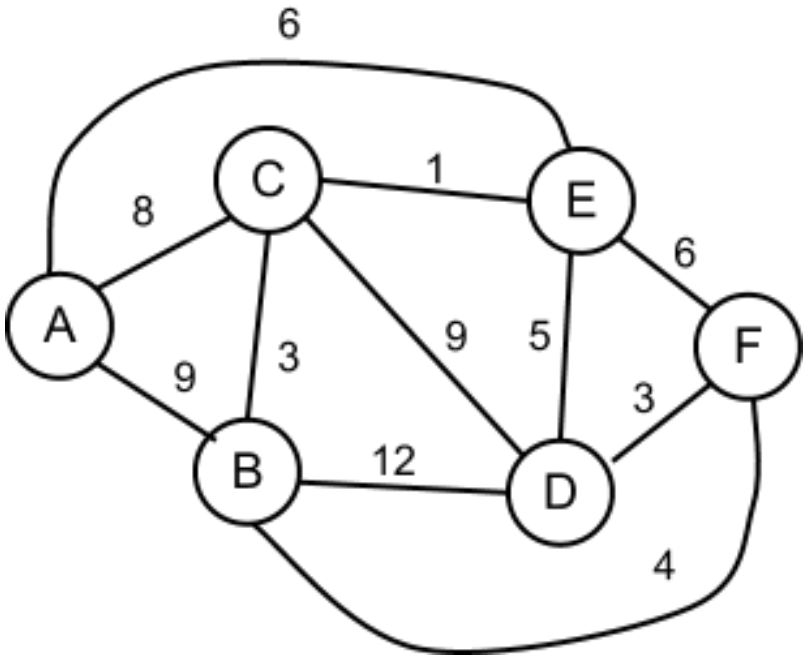
Implementing Dijkstra's Algorithm (cont.)

```
    for ( Integer j : neighbors ) { // for each neighbor j connected to our current vertex
         Vertex other = graph[j]; // get the neighboring Vertex

         int newDistance = current.dist + cost[current.id][j]; // calculate the distance from
                                                               // the current vertex to j
         if ( newDistance < graph[j].dist ) {   // if the new distance is smaller than j's dist
              priorityQueue.remove( graph[j] ); // remove the vertex from the PriorityQueue
              graph[j].dist = newDistance;      // set the vertex's new dist
              priorityQueue.add( graph[j] );    // add it back into the PriorityQueue so that it
         }                                      // will be "resorted"
    }
} // done with the loop
```

Example: Dijkstra's Algorithm

Imagine in the following graph, we are trying to travel from vertex **A** to vertex **F**.

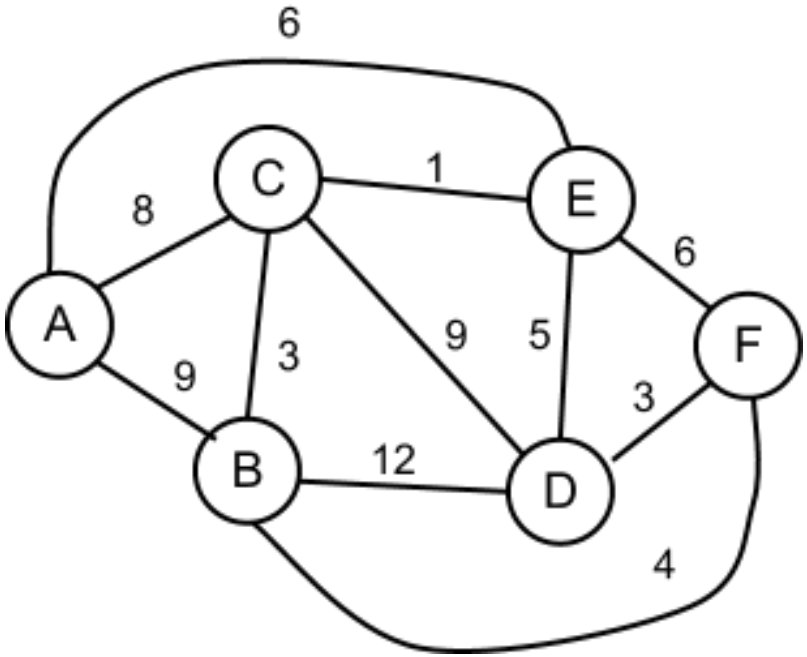


Example: Dijkstra's Algorithm

`current = null`

`priorityQueue = { }`

Vertex	*dist*
A	INF
B	INF
C	INF
D	INF
E	INF
F	INF

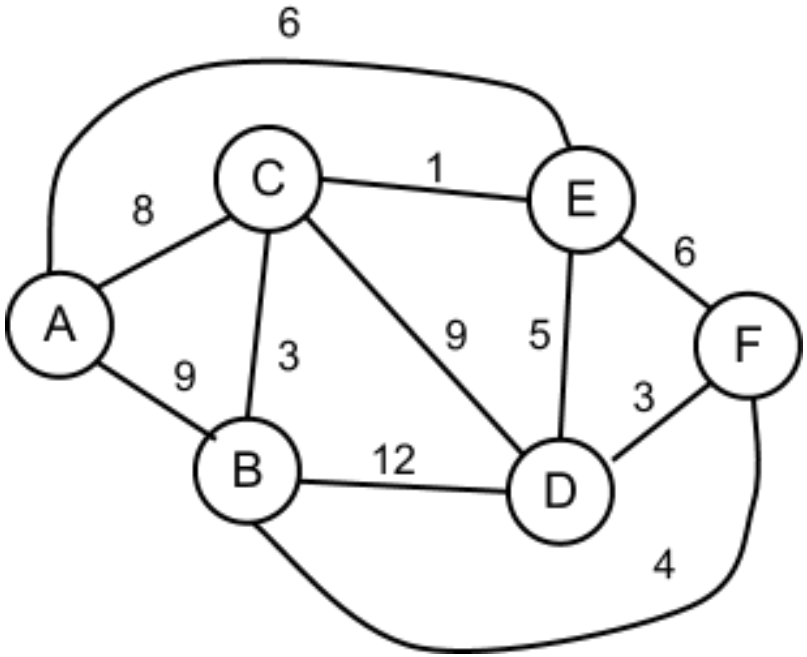


Example: Dijkstra's Algorithm

`current = null`

`priorityQueue = { A }`

Vertex	*dist*
A	0
B	INF
C	INF
D	INF
E	INF
F	INF

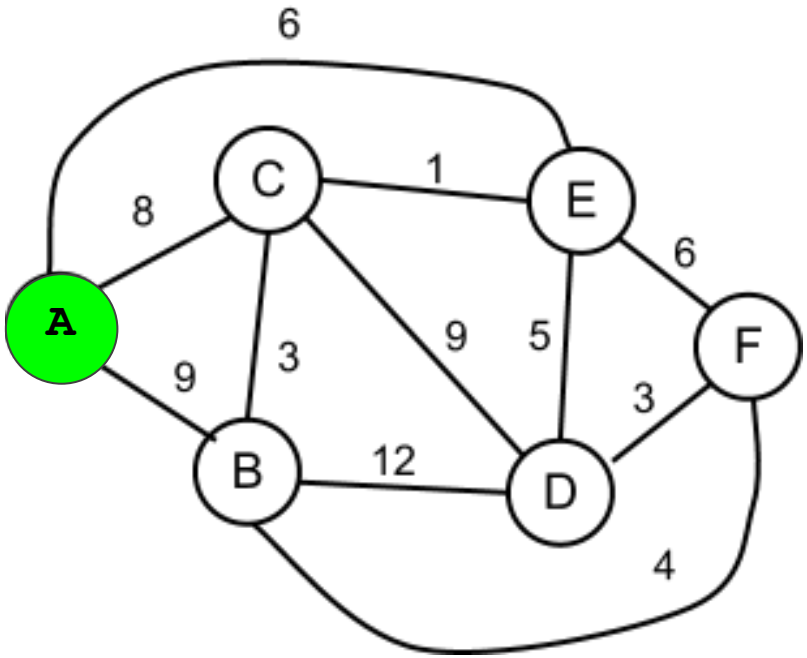


Example: Dijkstra's Algorithm

`current = A`

`priorityQueue = { }`

Vertex	*dist*
A	**0**
B	INF
C	INF
D	INF
E	INF
F	INF

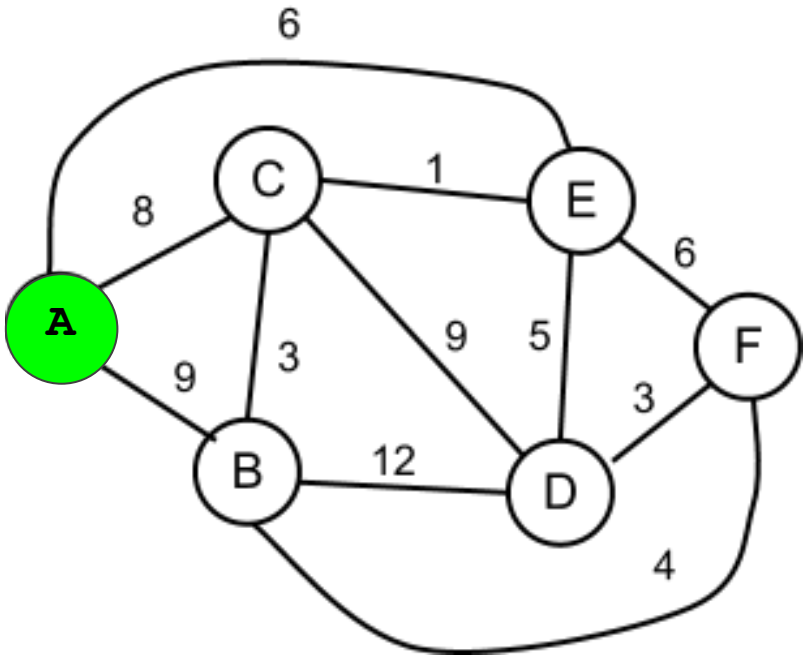


Example: Dijkstra's Algorithm

`current = A`

`priorityQueue = { }`

Vertex	*dist*
A	**0**
B	**0 + 9** < INF
C	**0 + 8** < INF
D	INF
E	**0 + 6** < INF
F	INF

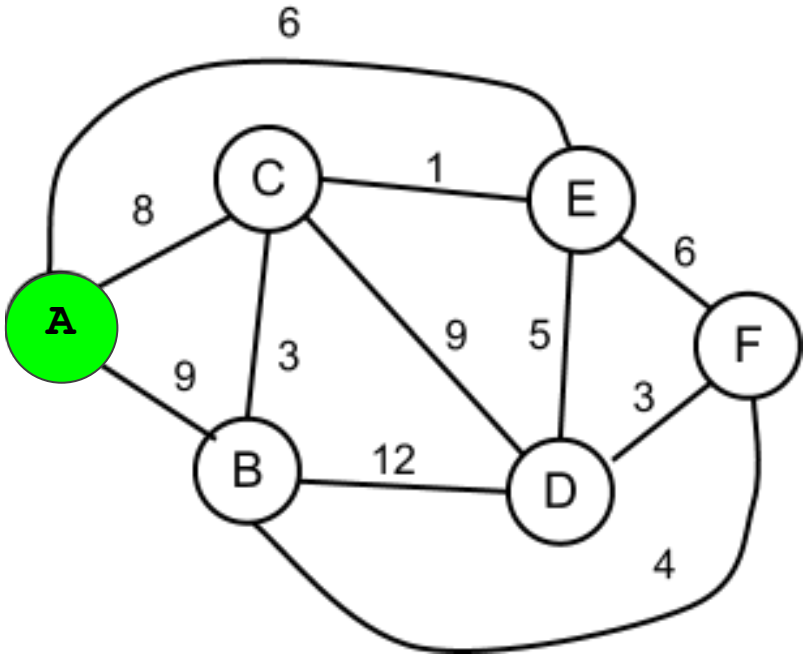


Example: Dijkstra's Algorithm

`current = A`

`priorityQueue = { E, C, B }`

Vertex	*dist*
A	**0**
B	9
C	8
D	INF
E	6
F	INF

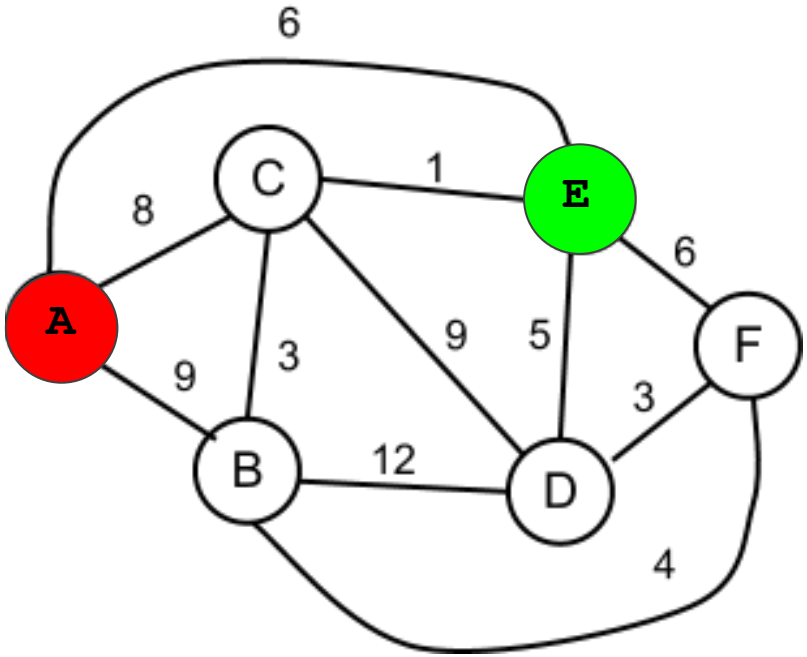


Example: Dijkstra's Algorithm

`current = E`

`priorityQueue = { C, B }`

Vertex	*dist*
A	0
B	9
C	8
D	INF
E	**6**
F	INF

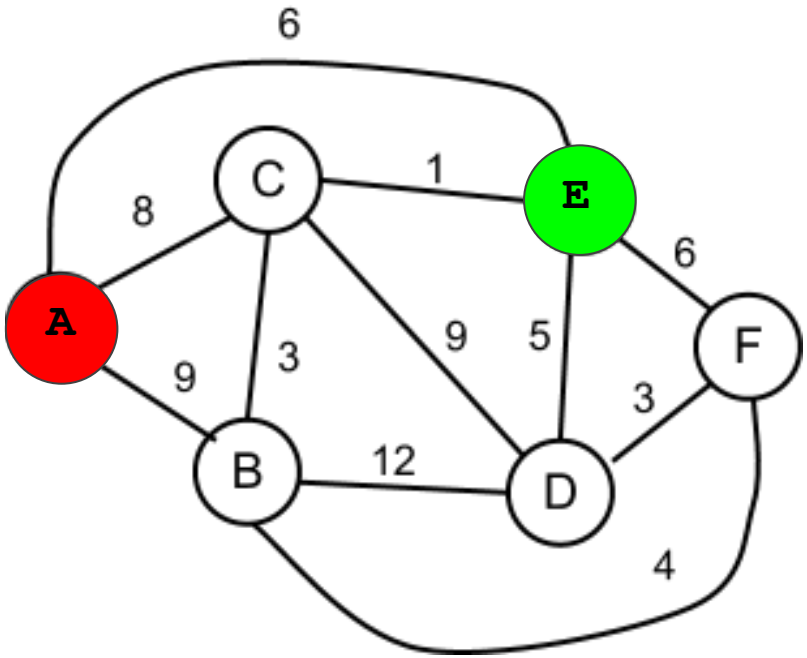


Example: Dijkstra's Algorithm

`current = E`

`priorityQueue = { C, B }`

Vertex	*dist*
A	0
B	9
C	**6 + 1** < 8
D	**6 + 5** < INF
E	**6**
F	**6 + 6** < INF

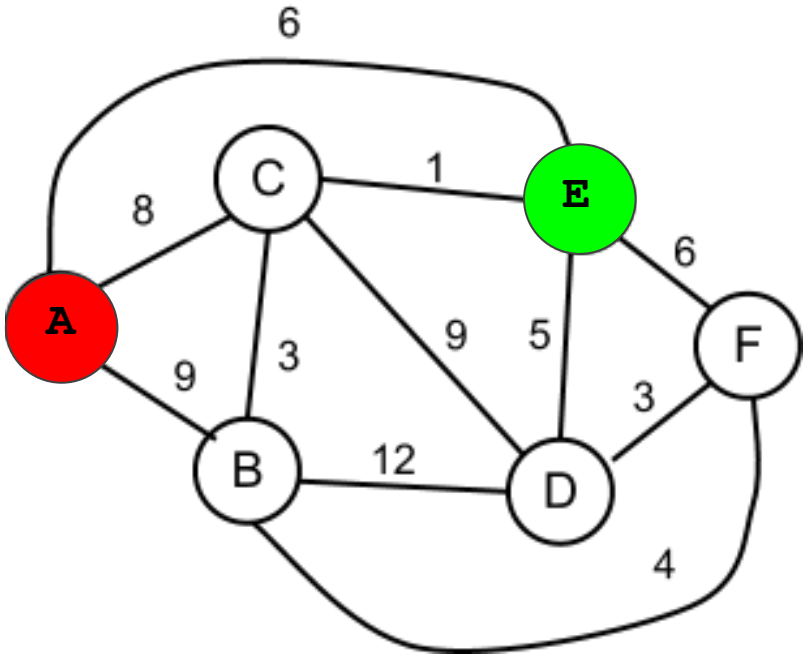


Example: Dijkstra's Algorithm

`current = E`

`priorityQueue = { C, B, D, F }`

Vertex	*dist*
A	0
B	9
C	7
D	11
E	**6**
F	12

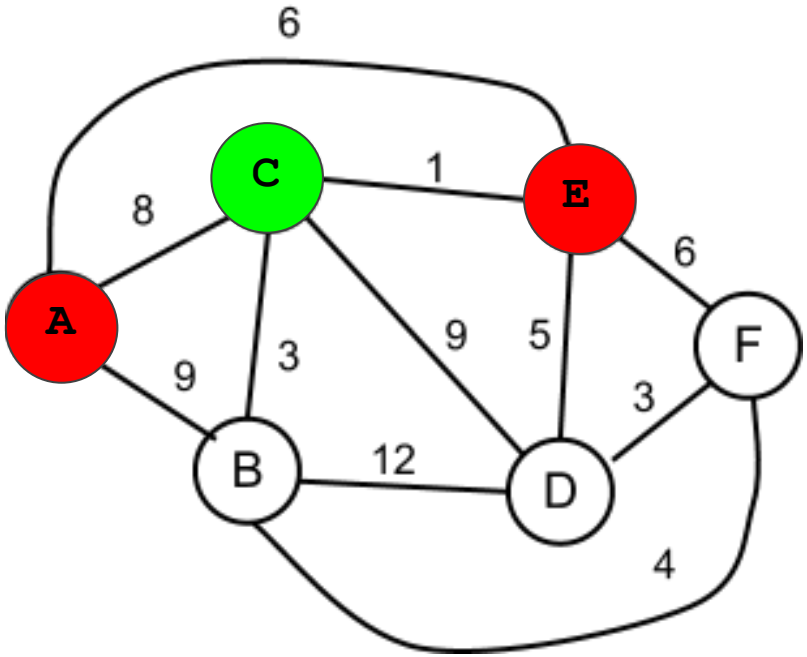


Example: Dijkstra's Algorithm

`current = C`

`priorityQueue = { B, D, F }`

Vertex	*dist*
A	0
B	9
C	**7**
D	11
E	6
F	12

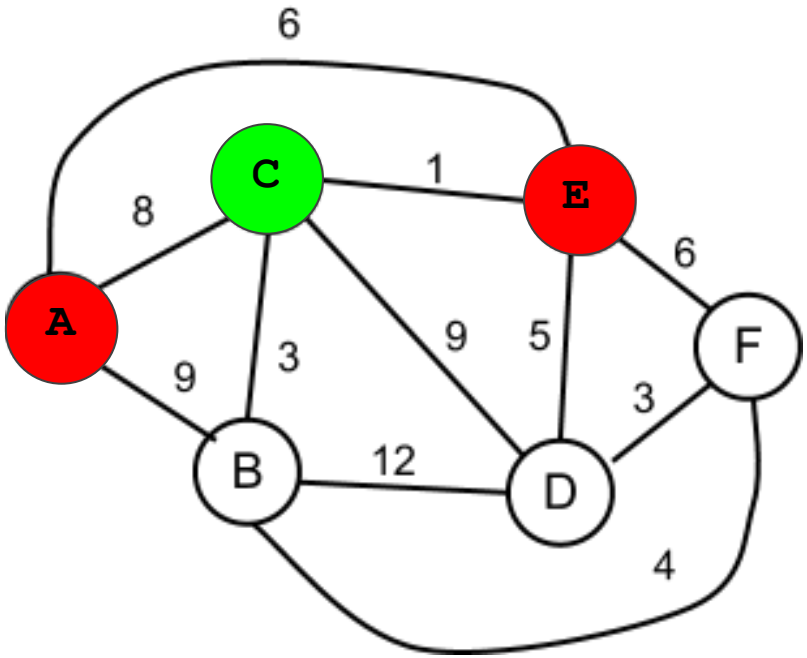


Example: Dijkstra's Algorithm

`current = C`

`priorityQueue = { B, D, F }`

Vertex	*dist*
A	0
B	**7 + 3** < 9
C	**7**
D	**7 + 9** < 11
E	6
F	12

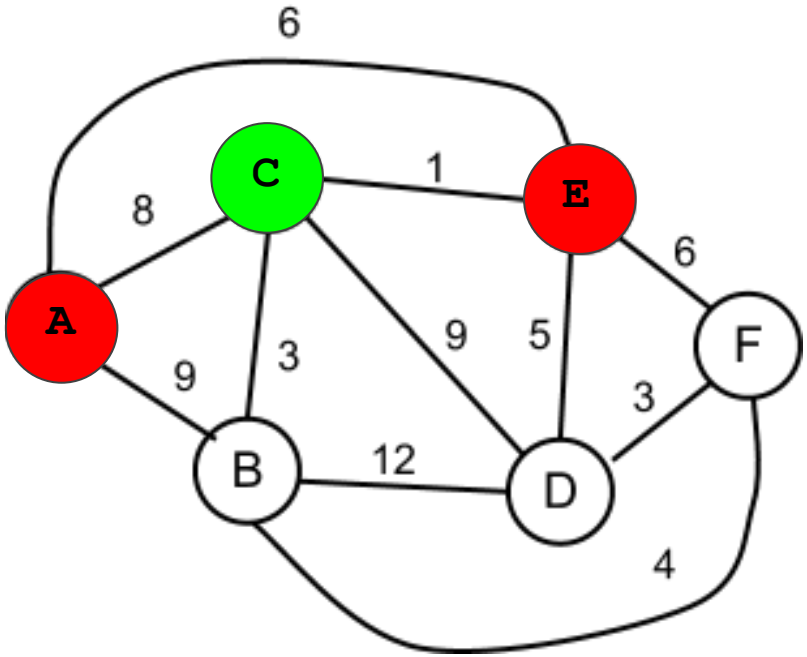


Example: Dijkstra's Algorithm

`current = C`

`priorityQueue = { B, D, F }`

Vertex	*dist*
A	0
B	9
C	**7**
D	11
E	6
F	12

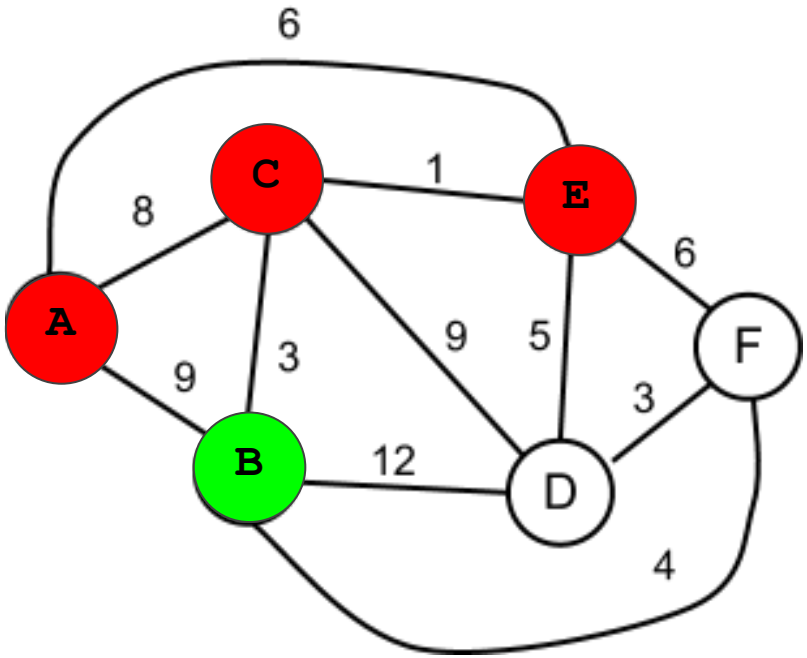


Example: Dijkstra's Algorithm

`current = B`

`priorityQueue = { D, F }`

Vertex	*dist*
A	0
B	**9**
C	7
D	11
E	6
F	12

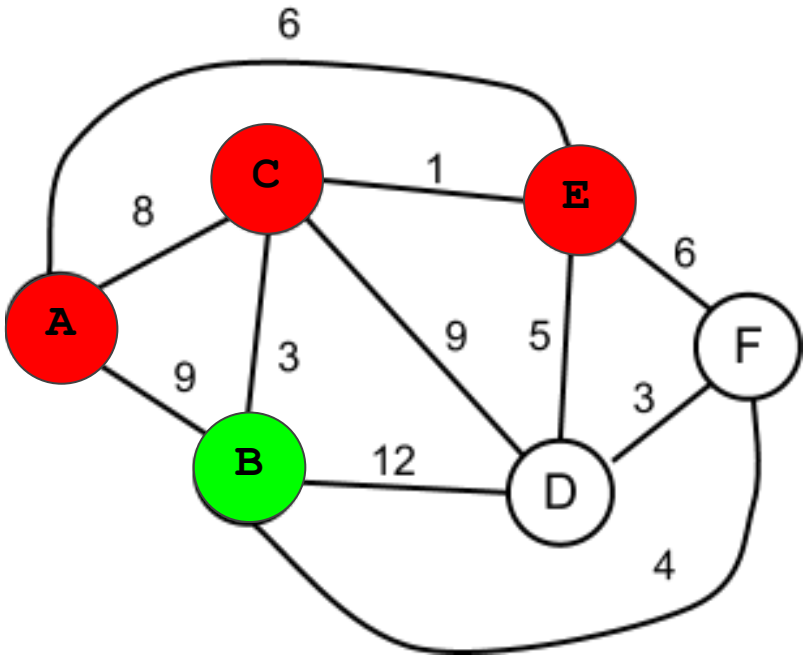


Example: Dijkstra's Algorithm

`current = B`

`priorityQueue = { D, F }`

Vertex	*dist*
A	0
B	**9**
C	7
D	**9 + 12 < 11**
E	6
F	12

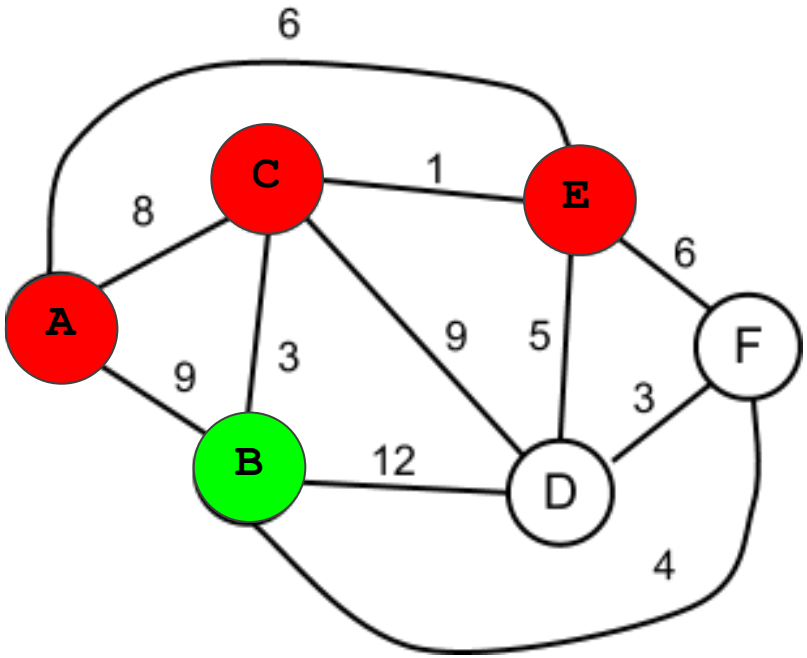


Example: Dijkstra's Algorithm

`current = B`

`priorityQueue = { D, F }`

Vertex	*dist*
A	0
B	**9**
C	7
D	11
E	6
F	12

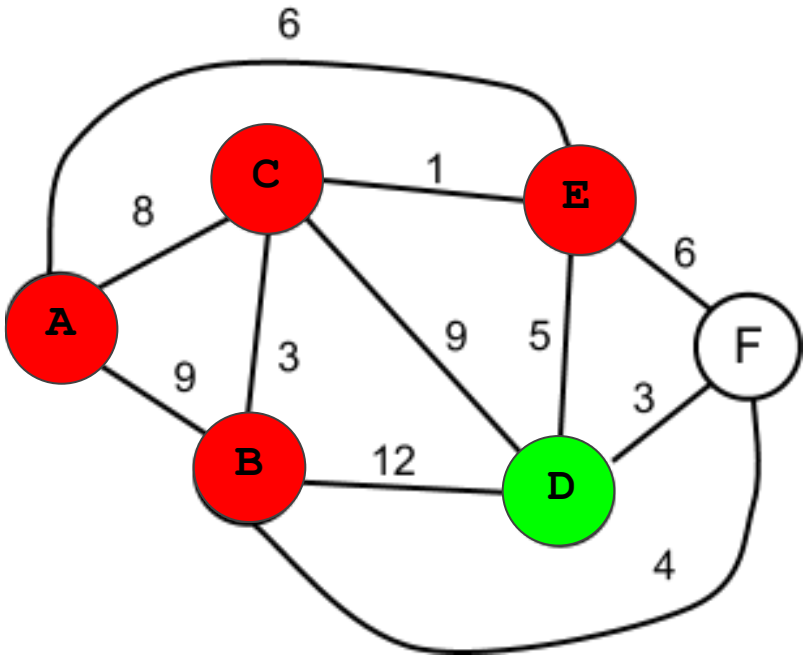


Example: Dijkstra's Algorithm

`current = D`

`priorityQueue = { F }`

Vertex	*dist*
A	0
B	9
C	7
D	**11**
E	6
F	12

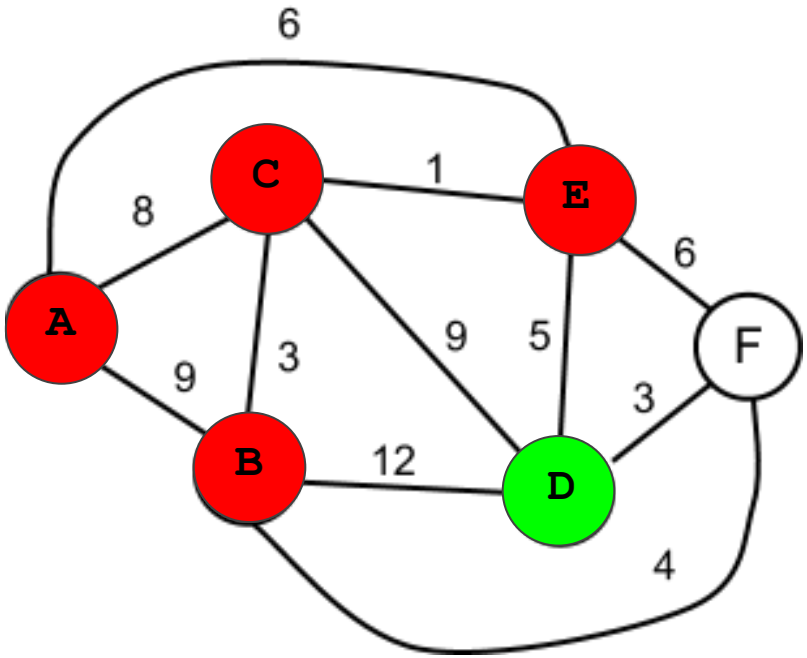


Example: Dijkstra's Algorithm

`current = D`

`priorityQueue = { F }`

Vertex	*dist*
A	0
B	9
C	7
D	**11**
E	6
F	**11 + 3** < 12

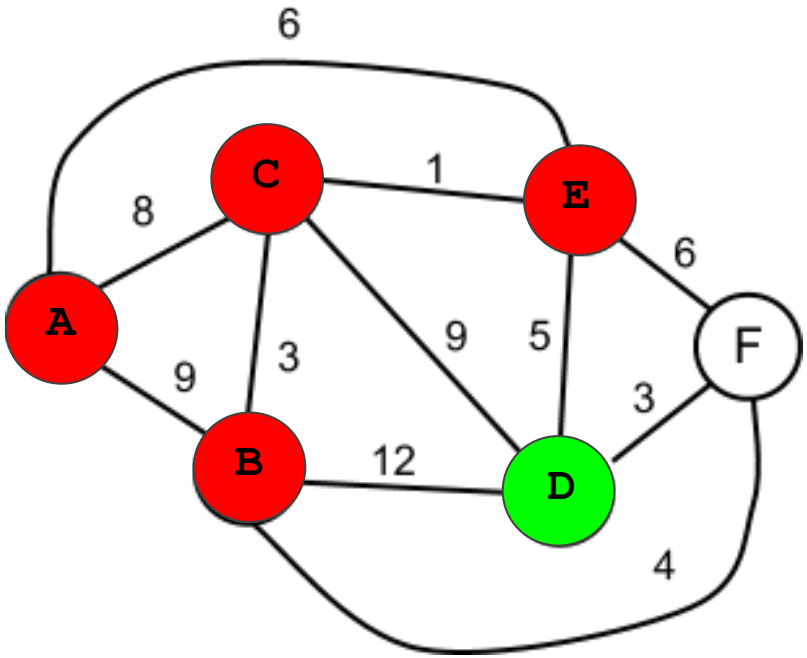


Example: Dijkstra's Algorithm

`current = D`

`priorityQueue = { F }`

Vertex	*dist*
A	0
B	9
C	7
D	**11**
E	6
F	12

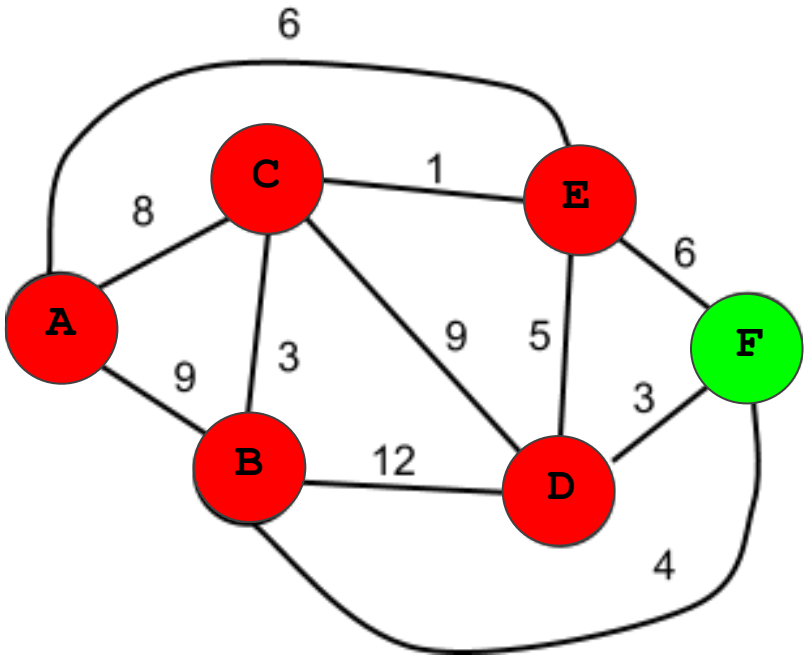


Example: Dijkstra's Algorithm

`current = F`

`priorityQueue = { }`

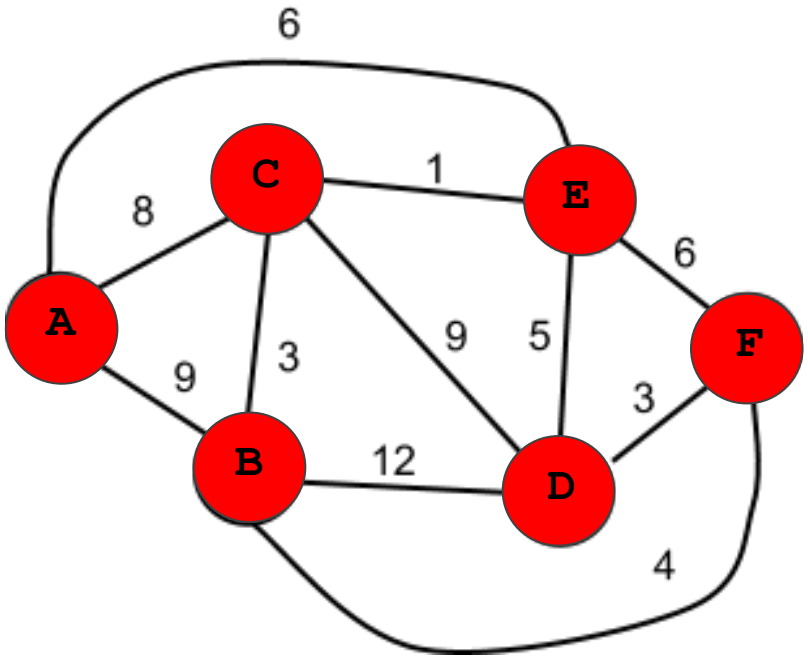
Vertex	*dist*
A	0
B	9
C	7
D	11
E	6
F	**12**



Example: Dijkstra's Algorithm

We have now reached our destination, vertex **F**

The shortest path from **A** to **F** is 12 units (**A** → **E** → **F**)



Problems

Break up into groups of 2-3 and work on the following problem:

spoj.com/problems/EZDIJKST

IMPORTANT: *the graph in EZDIJKST is* ***not*** *bidirectional*

More problems:

→ spoj.com/problems/CHICAGO

→ spoj.com/problems/MICEMAZE

→ spoj.com/problems/SHOP

→ spoj.com/problems/CCHESS