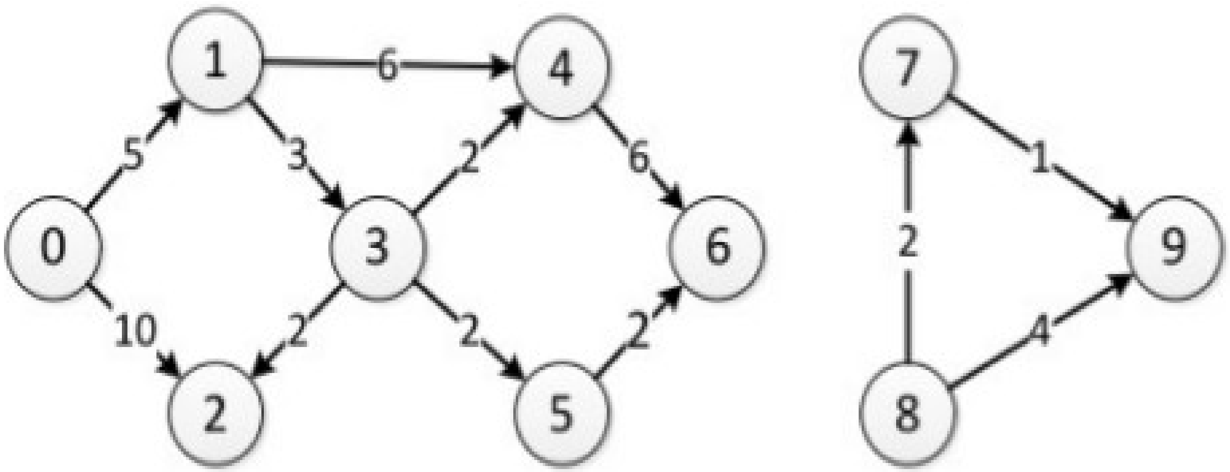
Project5 Weighted Graph

We will implement **Dijkstra's algorithm** for finding the shortest path from a source vertex to a destination vertex.

Dijkstra's algorithm



g2.txt

0 1 5.0

0 2 10.0

1 3 3.0

1 4 6.0

3 2 2.0

3 4 2.0

3 5 2.0

4 6 6.0

5 6 2.0

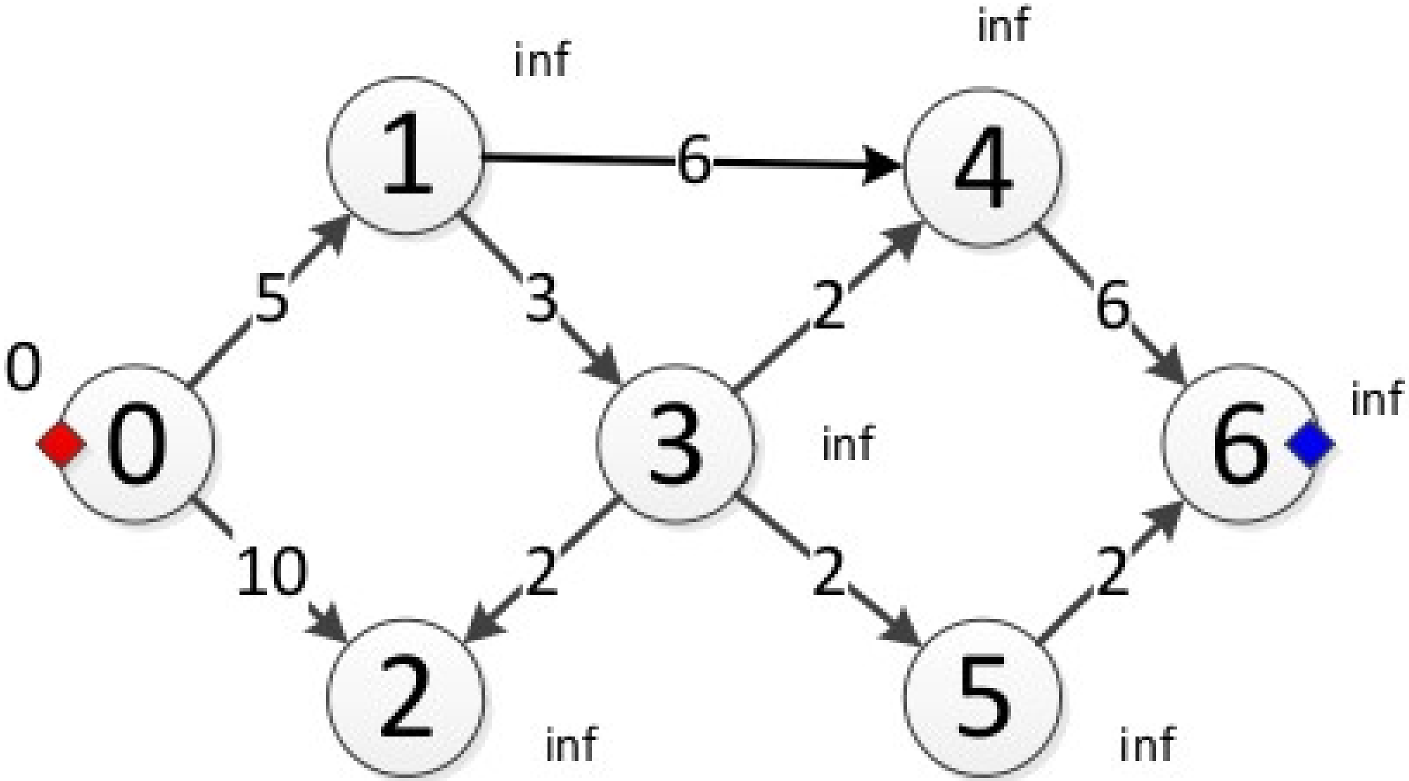
7 9 1.0

8 7 2.0

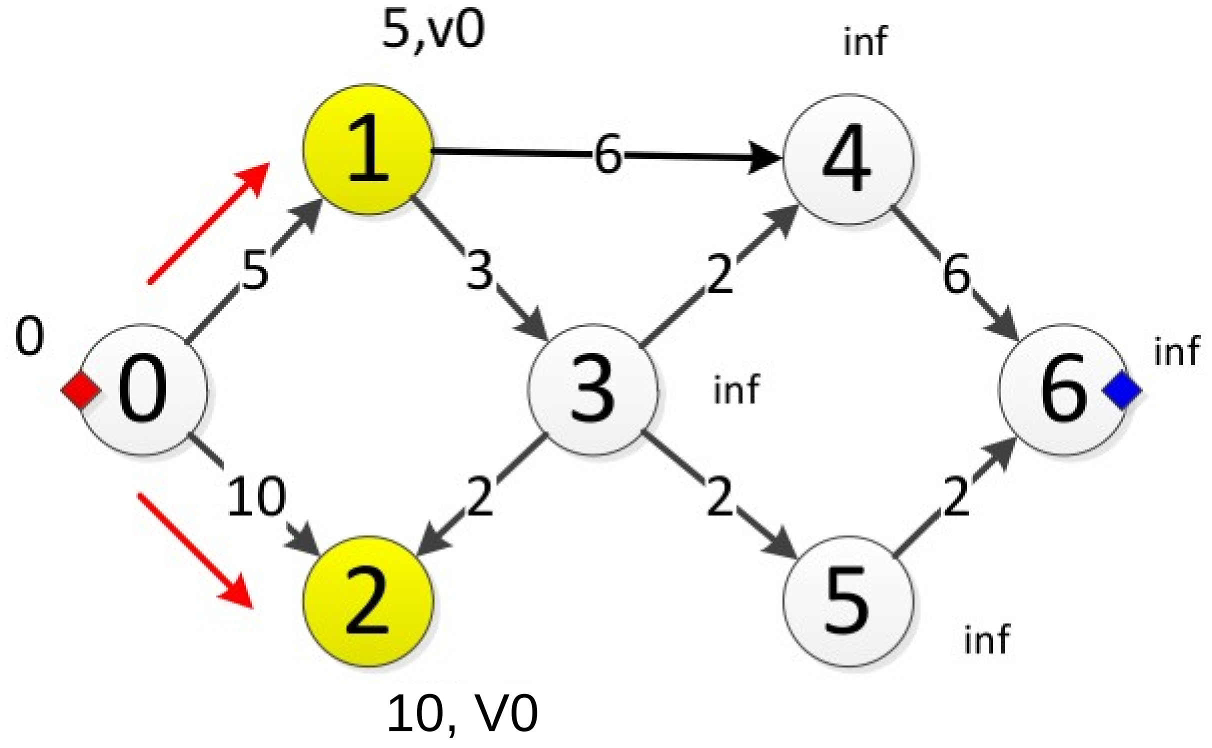
8 9 4.0

Find a shortest path from 0 to 6

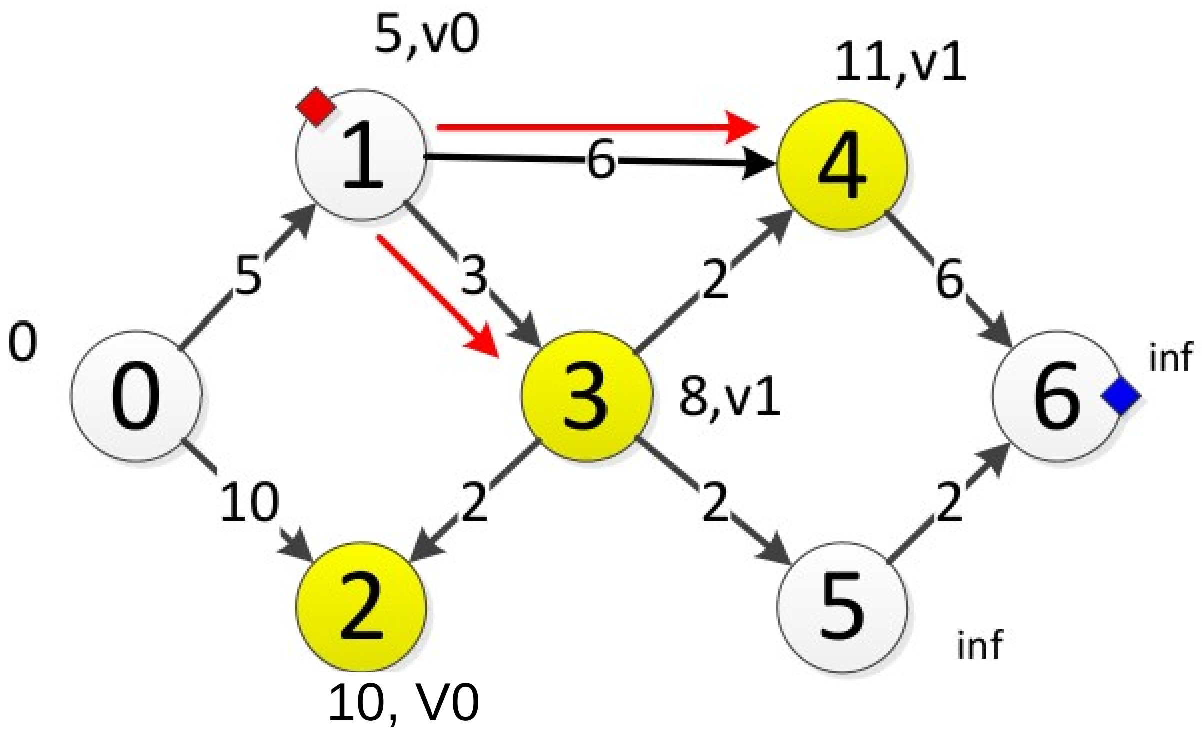
- 1 initial the cost 0 to source, and *inf* for other ve



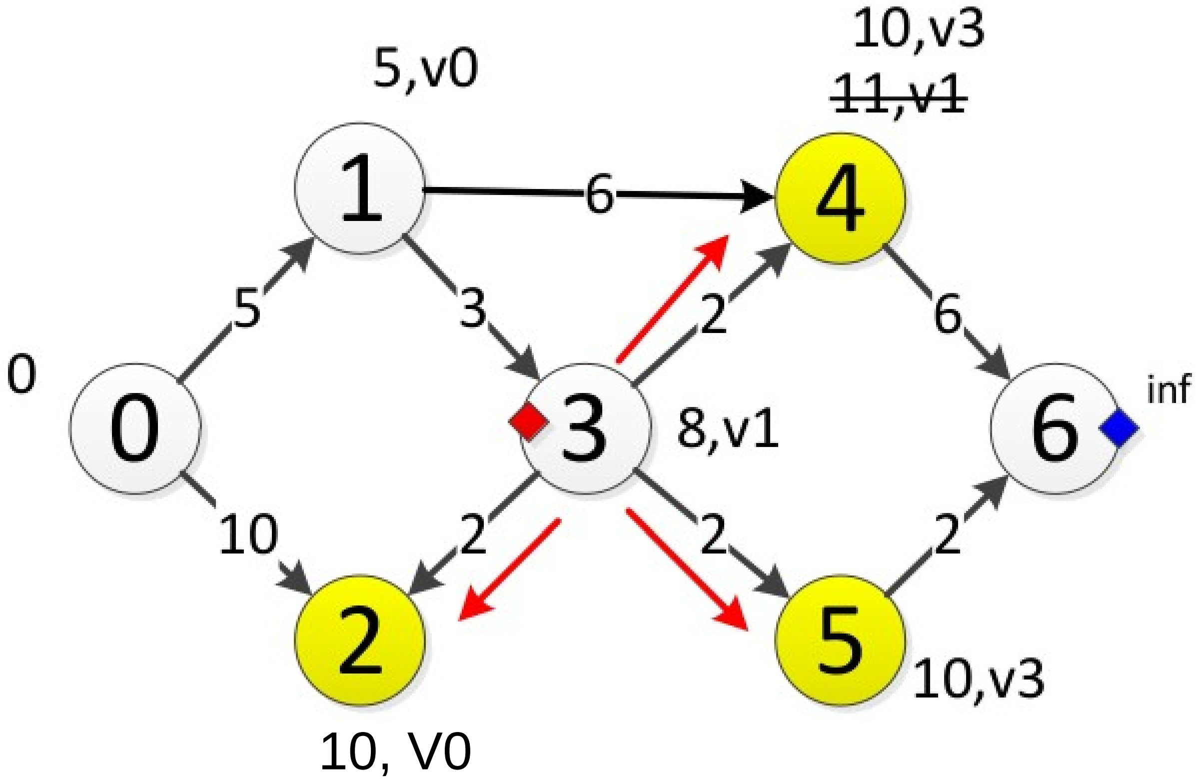
- 2. At vertex 0, update cost of its adjacent and record the previous vertex
- Traversing list: v1, v2



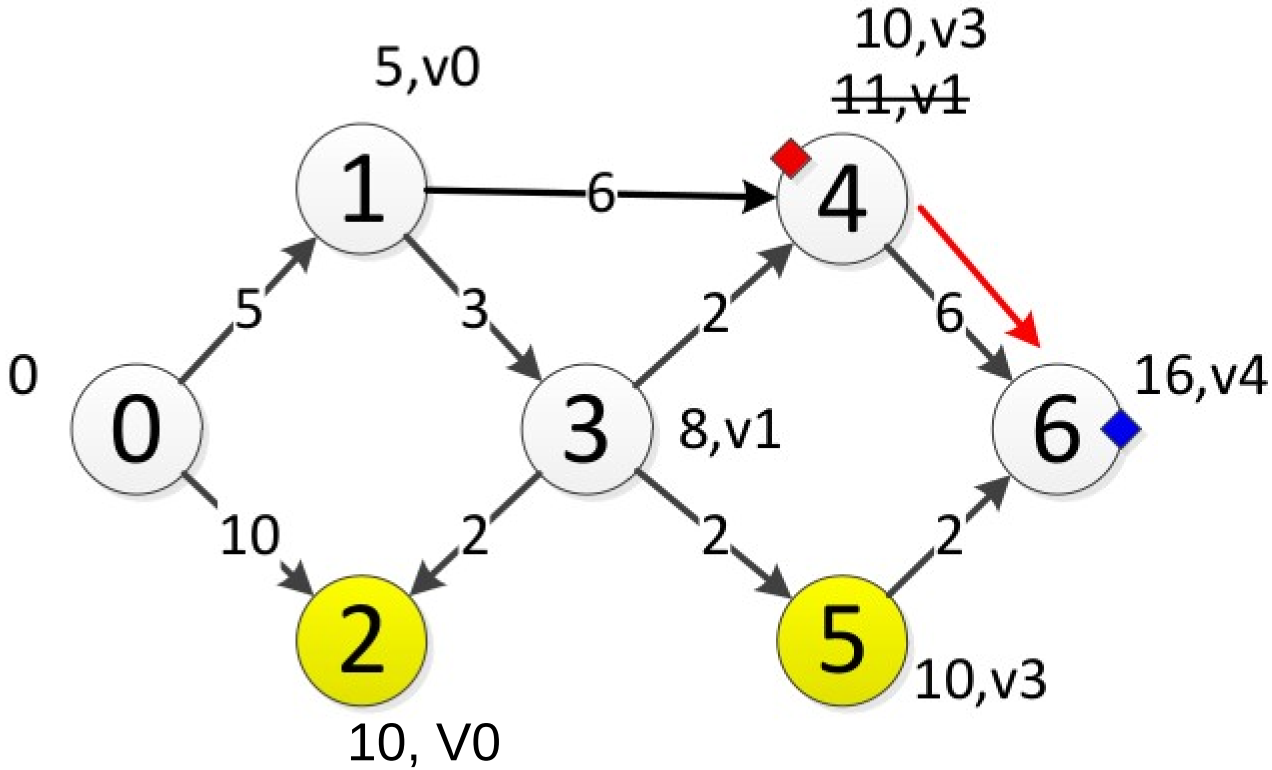
- 3. At vertex 1, update cost of its adjacent and record the previous vertex.
- Traversing list: v2, v3, v4



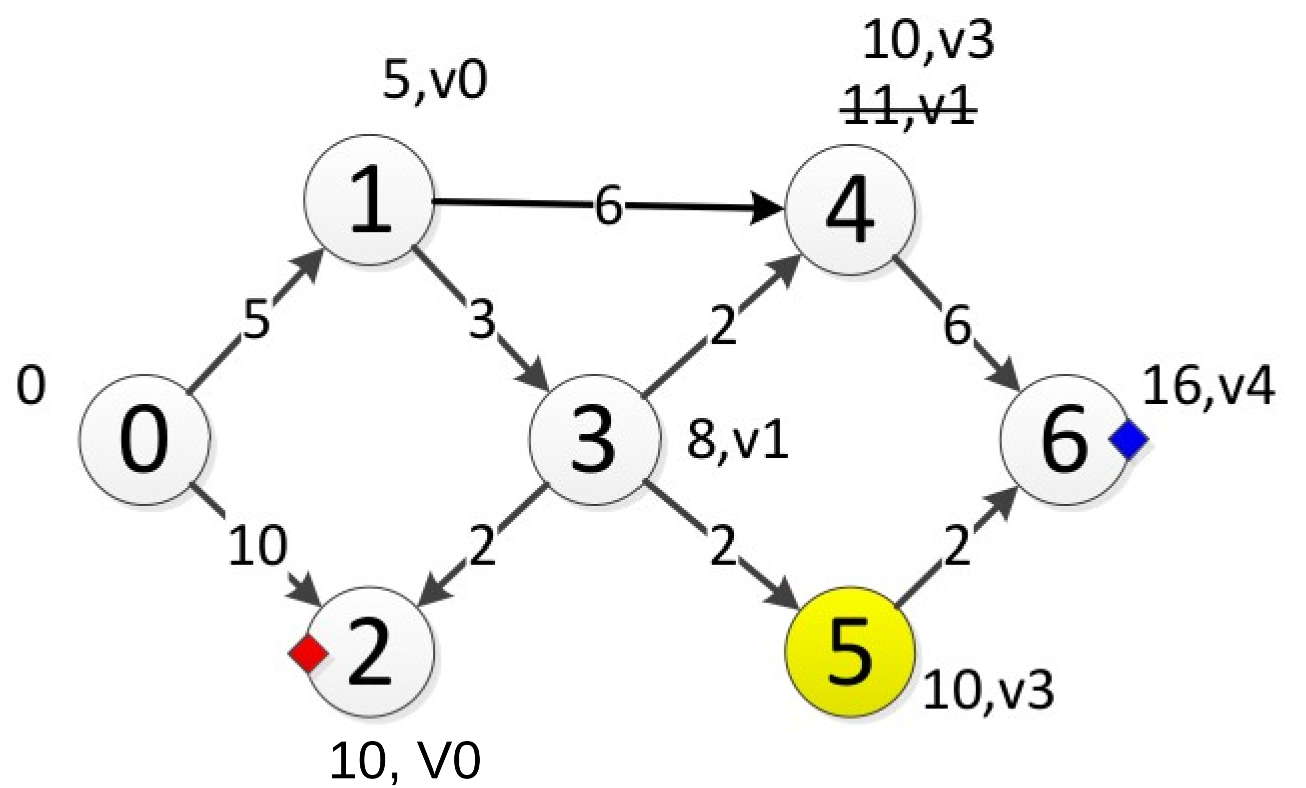
- 4. At vertex 3, update cost of its adjacent and record the previous vertex.
- Traversing list: v2, v4, v5



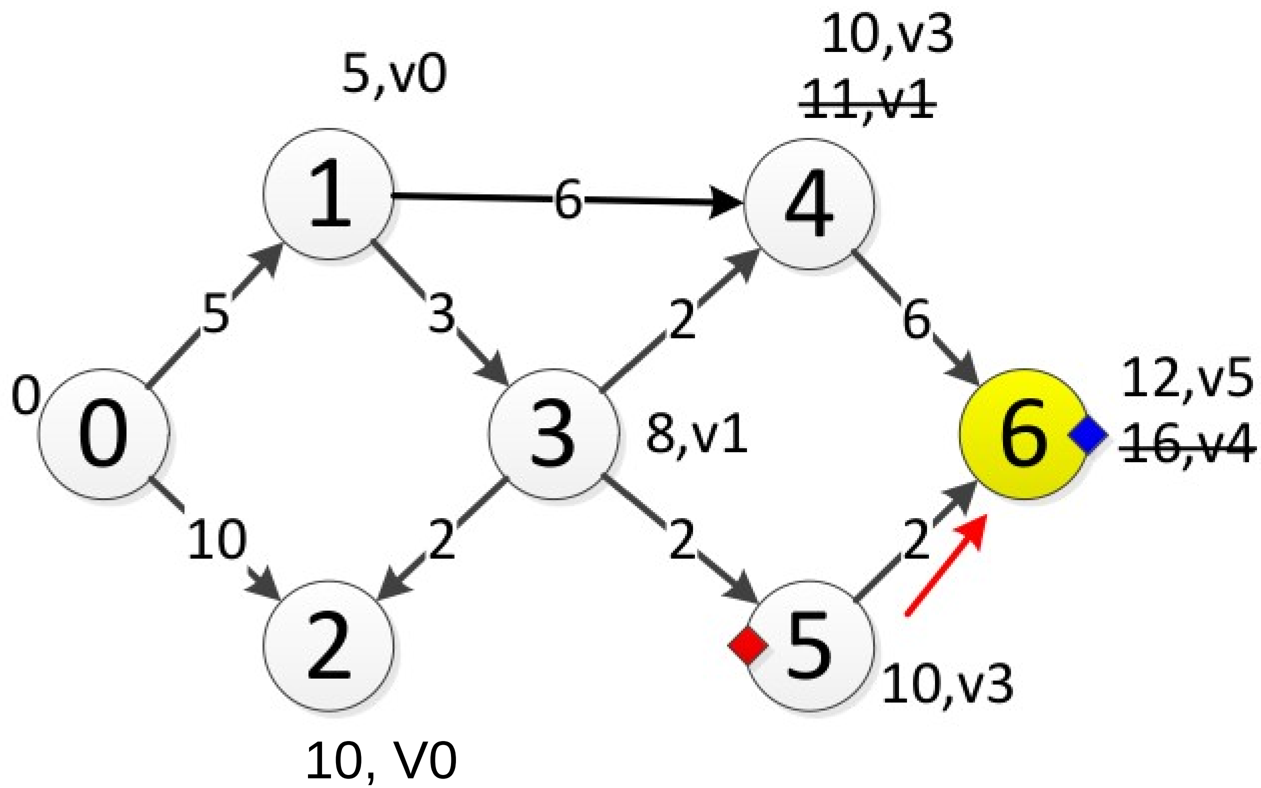
- 5. At vertex 4, update cost of its adjacent and record the previous vertex.
- Traversing list: v2, v5, v6



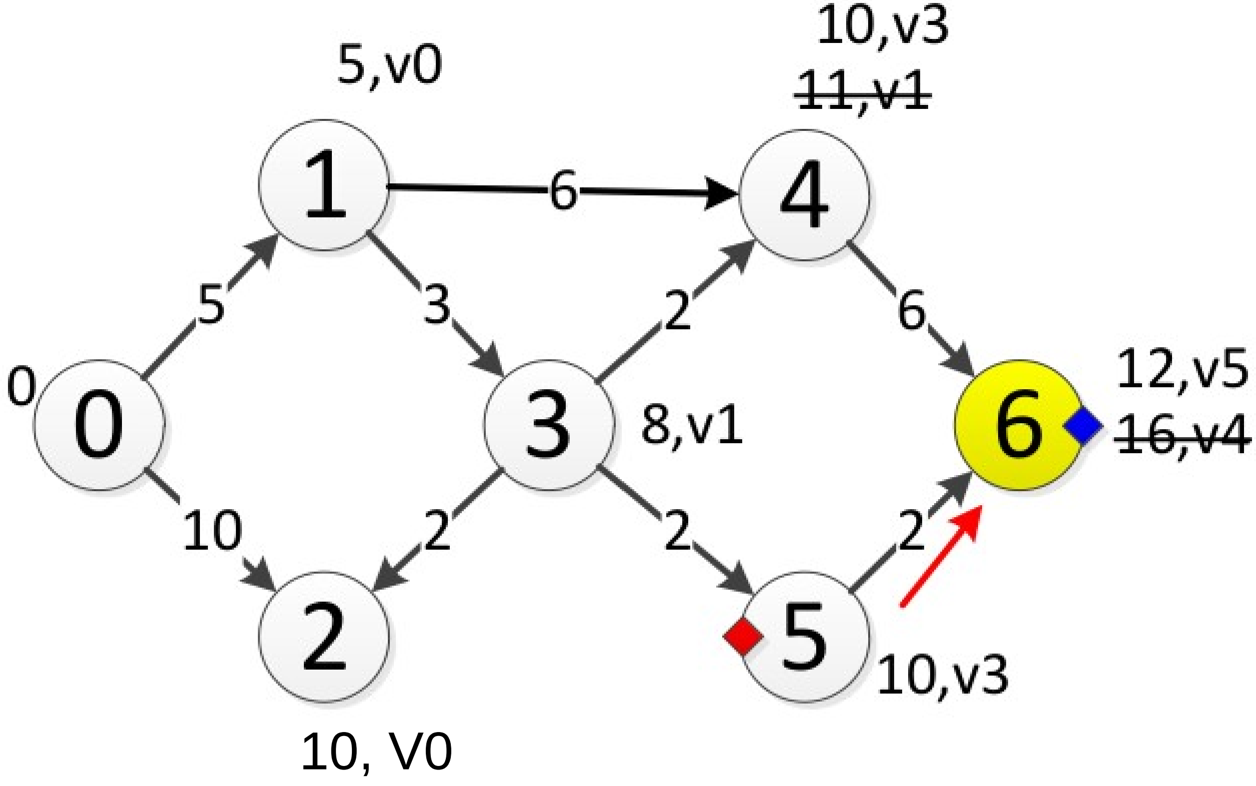
- 6. At vertex 2, update cost of its adjacent and record the previous vertex.
- Traversing list: v5, v6



- 7. At vertex 5, update cost of its adjacent and record the previous vertex.
- Traversing list: v6

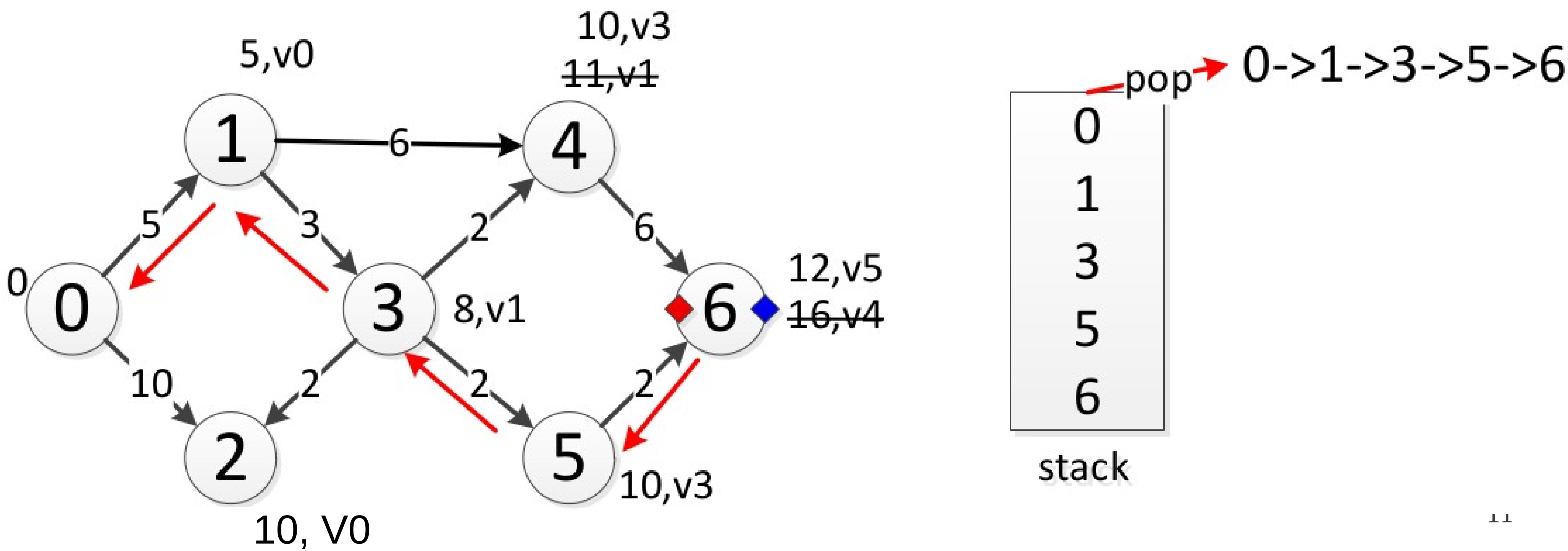


- 8. At vertex 6, once we visit the destination vertex, we have found the shortest path :)
- Traversing list: empty



The shortest path

- We know that the shortest path from 0 to 6 cost 12.
- Let's track back to get a traversing path.



How to store a graph

Method1:

- 2d-array
  - double weight[V-1][V-1]
  - where V is the number of vertices (you can assume that vertex id is always start from 0 to V-1)

- weight[0][1] stores the weight from vertex 0 to vertex 1
- weight[1][0] stores the weight from vertex 1 to vertex 0
- weight[1][2] stores the weight from vertex 1 to vertex 2 and so on

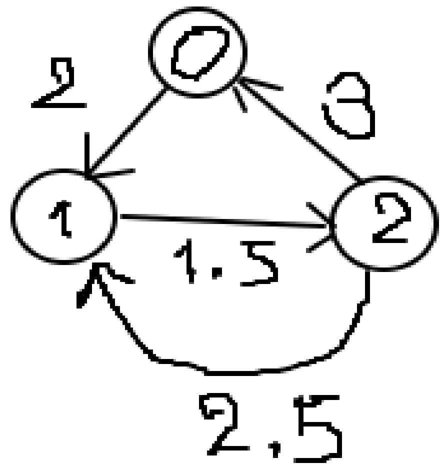
Not effective, because we have to read through an input file to know a number of vertex.

Also, access adjacent of a vertex take a linear time instead of a constant time.

It's OK, it at least works (keep this as your very last option)

*** To speed up searching time, you might create a map <int,int> store edges between each vertex, and look up the weight in the array weight.*

Example



weight

	[0]	[1]	[2]
[0]		2.0	
[1]			1.5
[2]	3.0	2.5	

Method2:

```
class Vertex {
   int id;
   map <int, double> edgesToAdjacent;
   // override operations...
}
class Graph {
   vector<Vertex> vertices;
}
```

more effective, a bit more complicate

Think about how you could store cost/path of vertices during traverse a graph

- A traversing list: stores a list of vertex that we will visit. It's like a next visited list.
- We always visit a vertex with the minimum cost first.
  - Min-heap / priority queue

Traverse a graph finding a shortest path

```
- Initial the cost of a source vertex to 0 and other vertex to inf
- add adjacent of a source vertex to a traverse list
while (a traverse list is not empty) {
     u = a minimum cost vertex in a traverse list
     if (u is a destination vertex) {
          // we have found a shortest path, break a while loop
     } else {
       for (all adjacent of u) {
         if (cost of of an adjacent is inf) {
            // assign a cost to this adjacent
         } else {
            // check if the current cost is less than the existing cost.
            // If so, update cost of this adjacent to the lower cost
            // and update a path to this adjacent
         }
       }
     }
}
```

Trace back to get a traversing path

- We can use **stack** to store vertices

```
//start from a destination vertex, put a destination vertex in a stack
while (we have not reached a source vertex) {
    u = look at the top of a stack
    v = a vertex that u comes from
    put v in a stack
}
// put a source vertex in a stack
// iterate pop thing out from a stack → we get a traversing path :)
```

If you want to watch the animation of Dijstra's algorithm, check this out
https://www.youtube.com/watch?v=zXfDYaahsNA