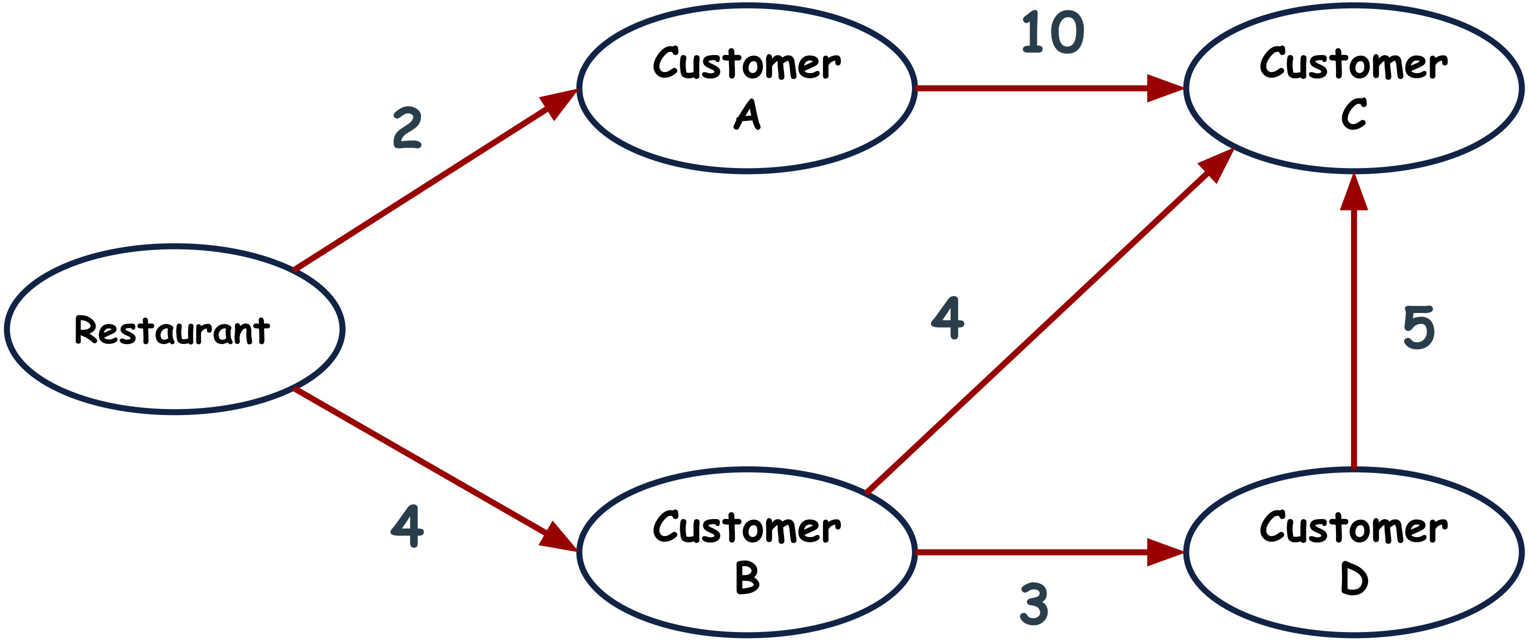
Dijkstra Algorithm

Graphs: Problem

Congratulations!!! You are now working in a 5 star restaurant. You are also providing the facility of food delivery. Since you are a Computer Scientist, you have converted the problem into graphs and now you have to find the shortest path (with minimum cost) from your restaurant to the customer's house to deliver the food.

Graphs: Problem



Graphs: Problem

Your Goal is to find the path from the Restaurant to each customer's house with minimum cost.

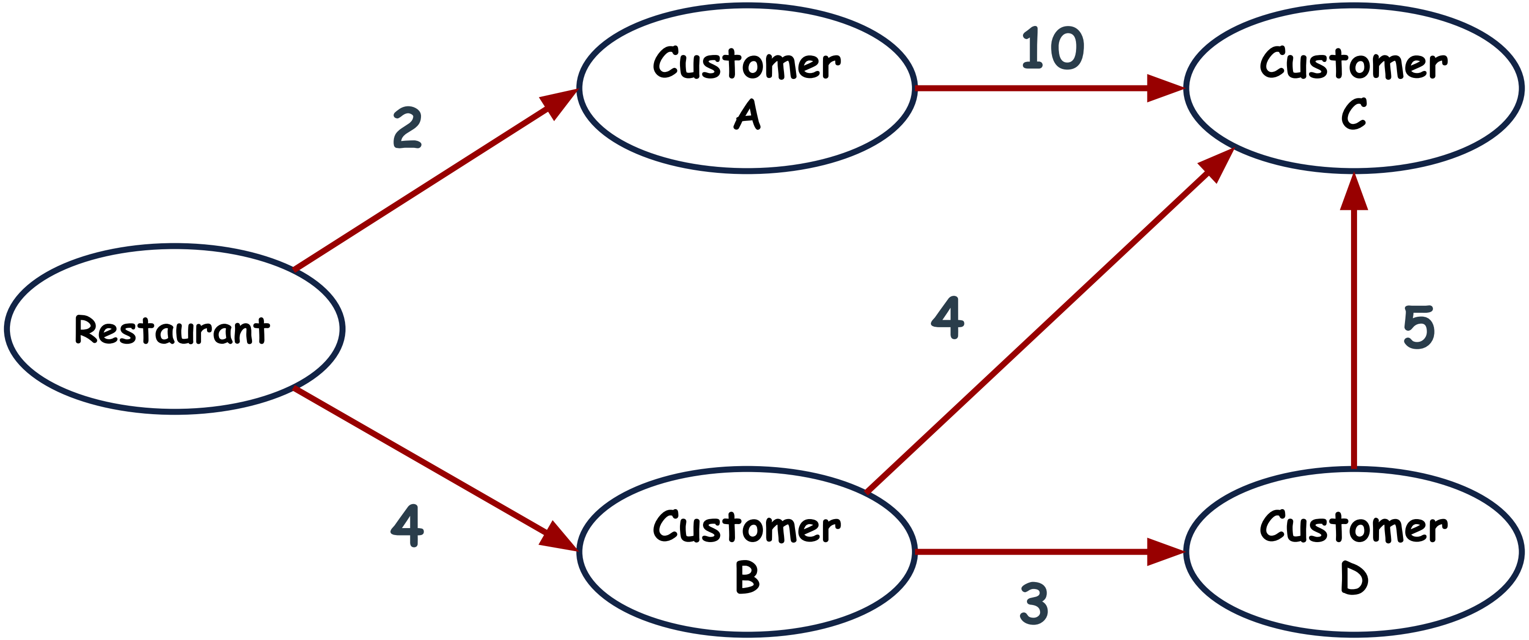
Graphs: Problem

Your Goal is to find the path from the Restaurant to each customer's house with minimum cost.

How can we do that?

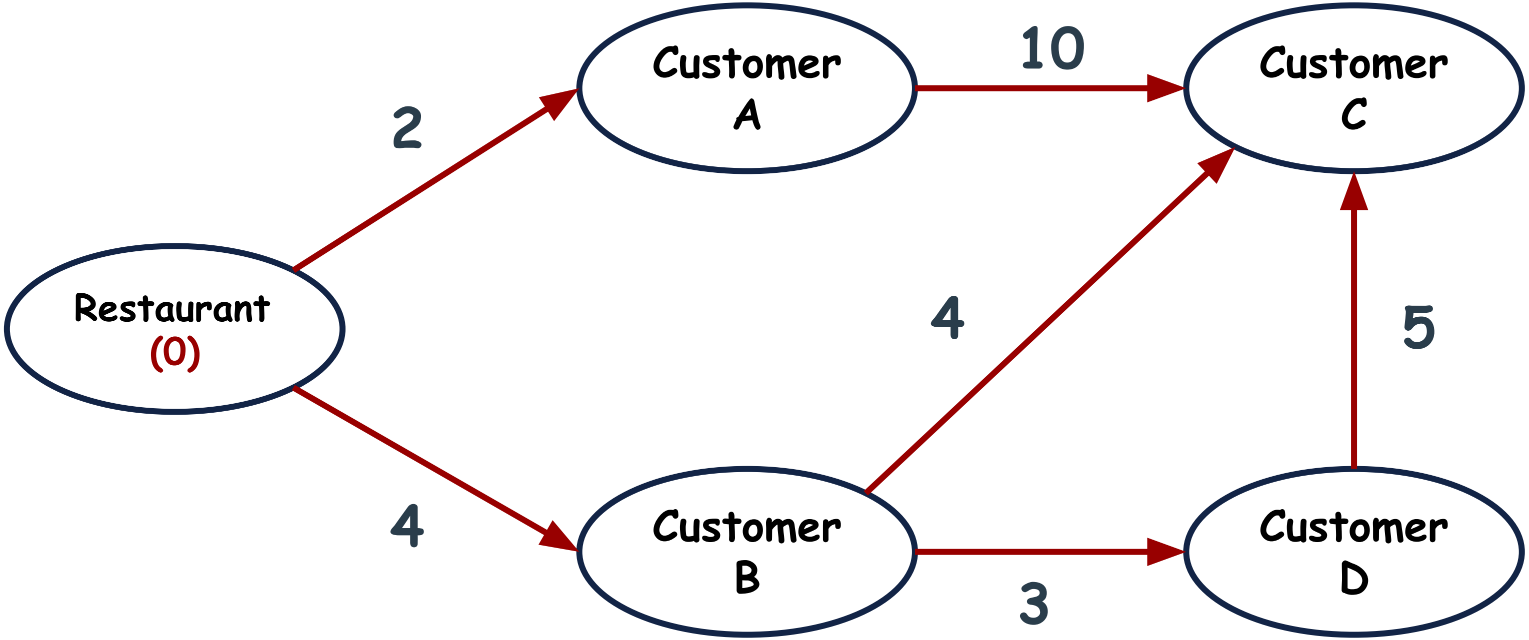
Graphs: Problem

We know that the starting point is the Restaurant.
Therefore, the cost to reach the Restaurant is 0.



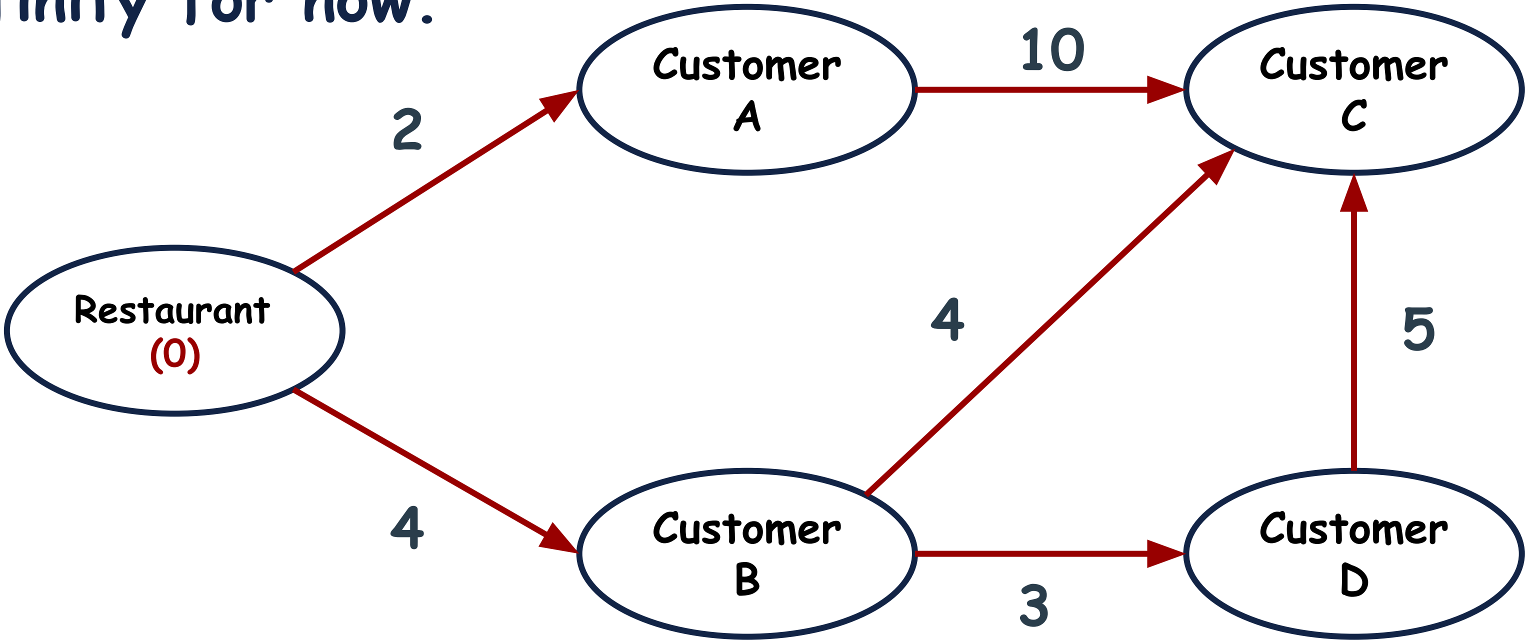
Graphs: Problem

We know that the starting point is the Restaurant. Therefore, the cost to reach the Restaurant is 0.



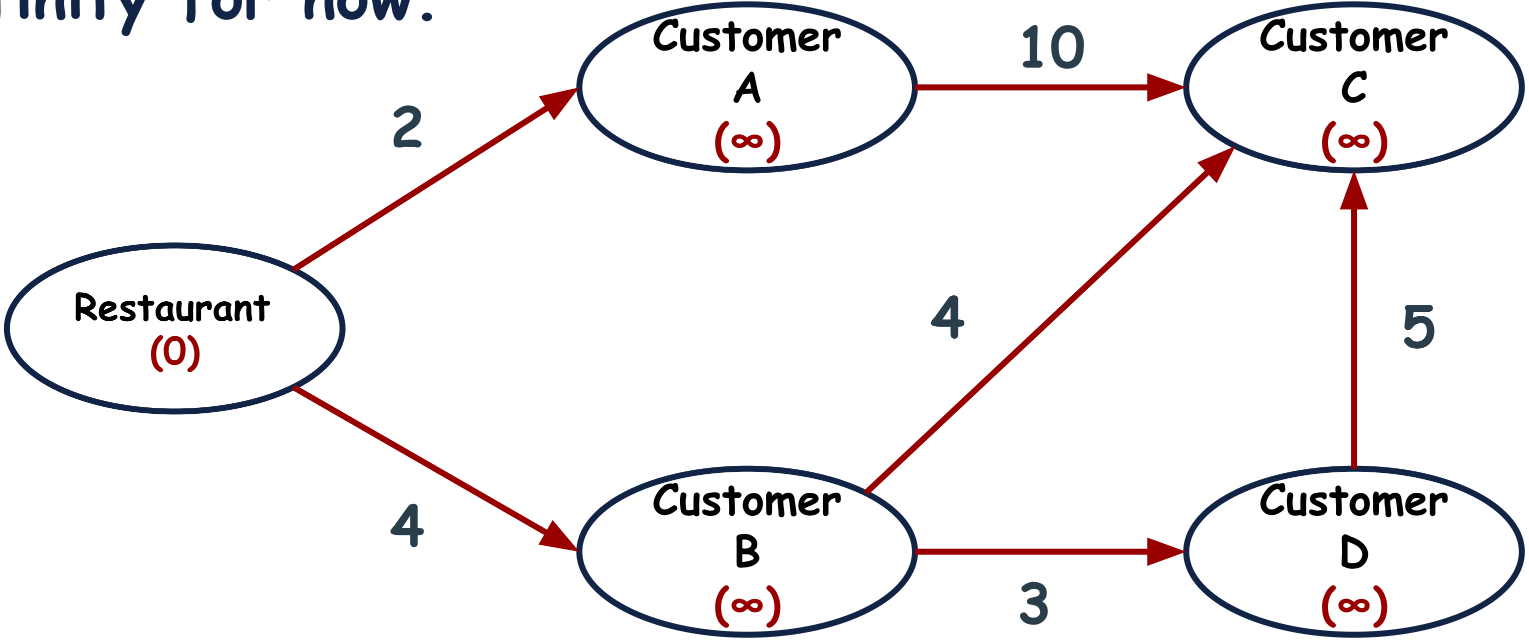
Graphs: Problem

Now, at this moment we don't know the cost to all the customers houses therefore, we'll assume these costs as infinity for now.



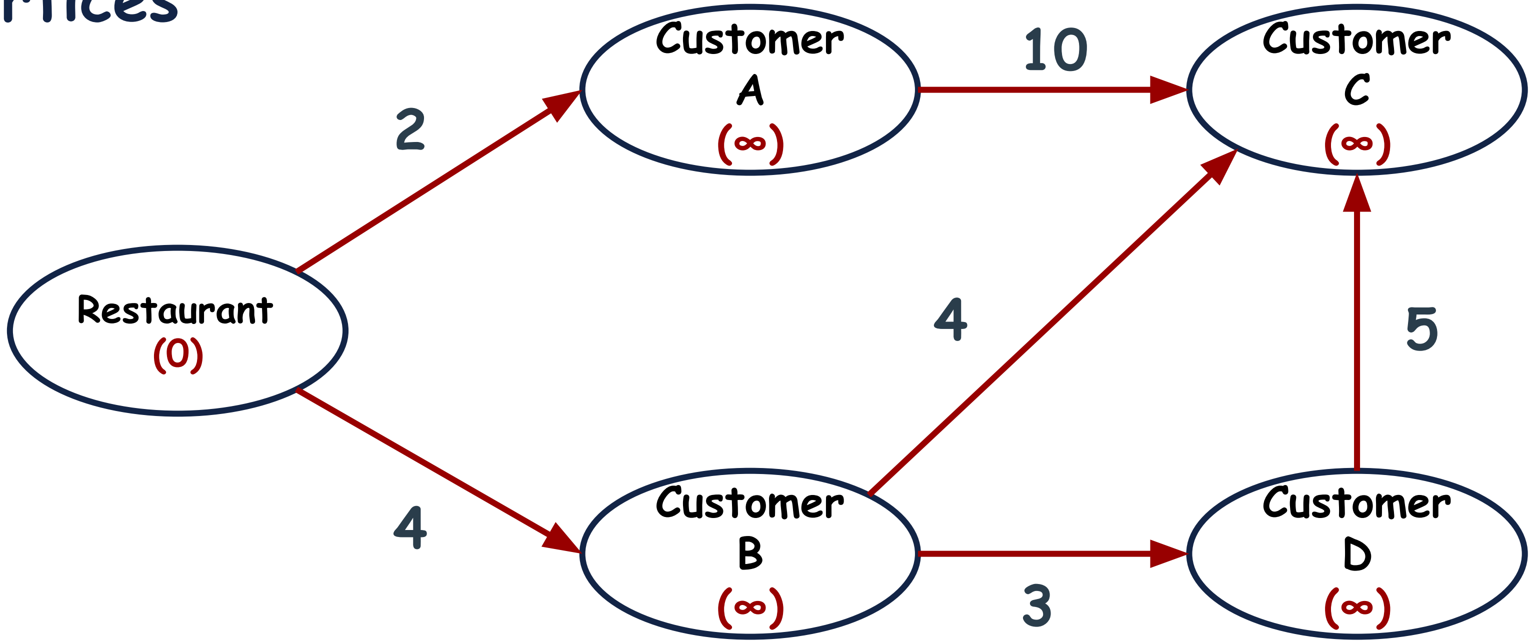
Graphs: Problem

Now, at this moment we don't know the cost to all the customers houses therefore, we'll assume these costs as infinity for now.



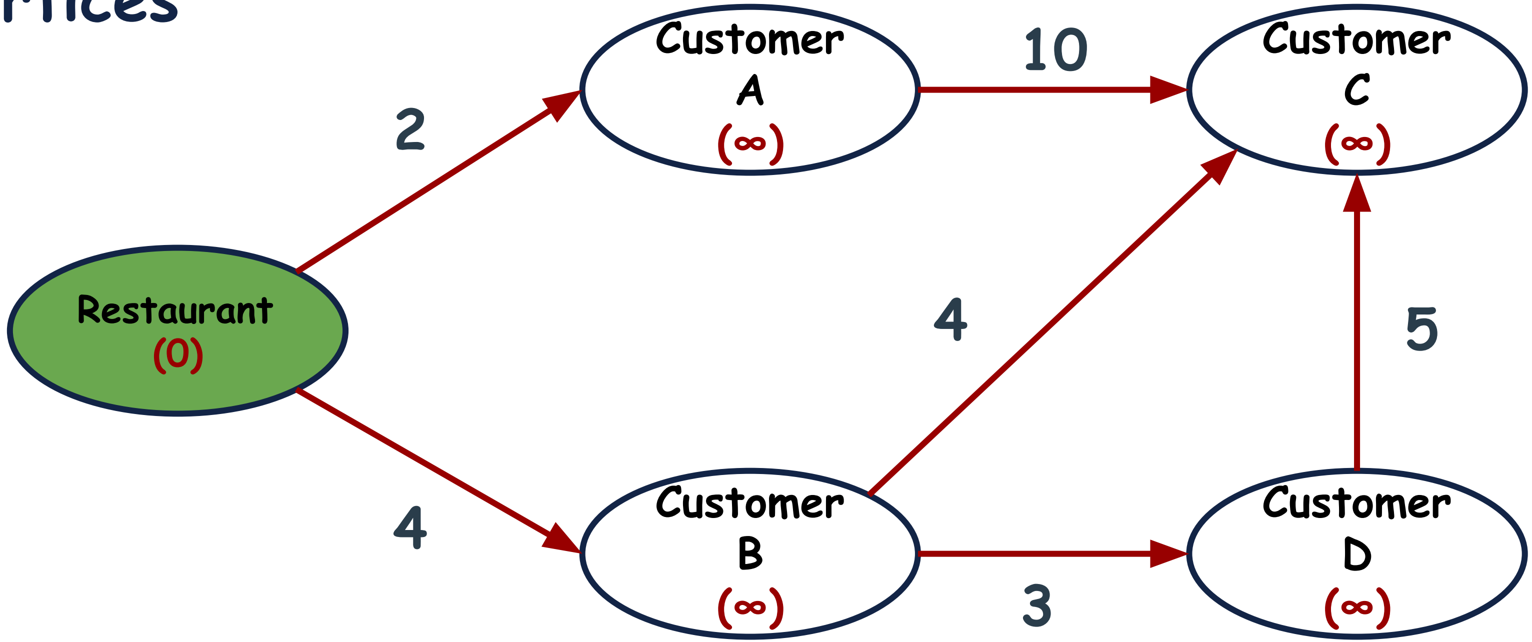
Graphs: Problem

We will start from the starting vertex (Restaurant), mark it as visited and check all of its unvisited adjacent vertices



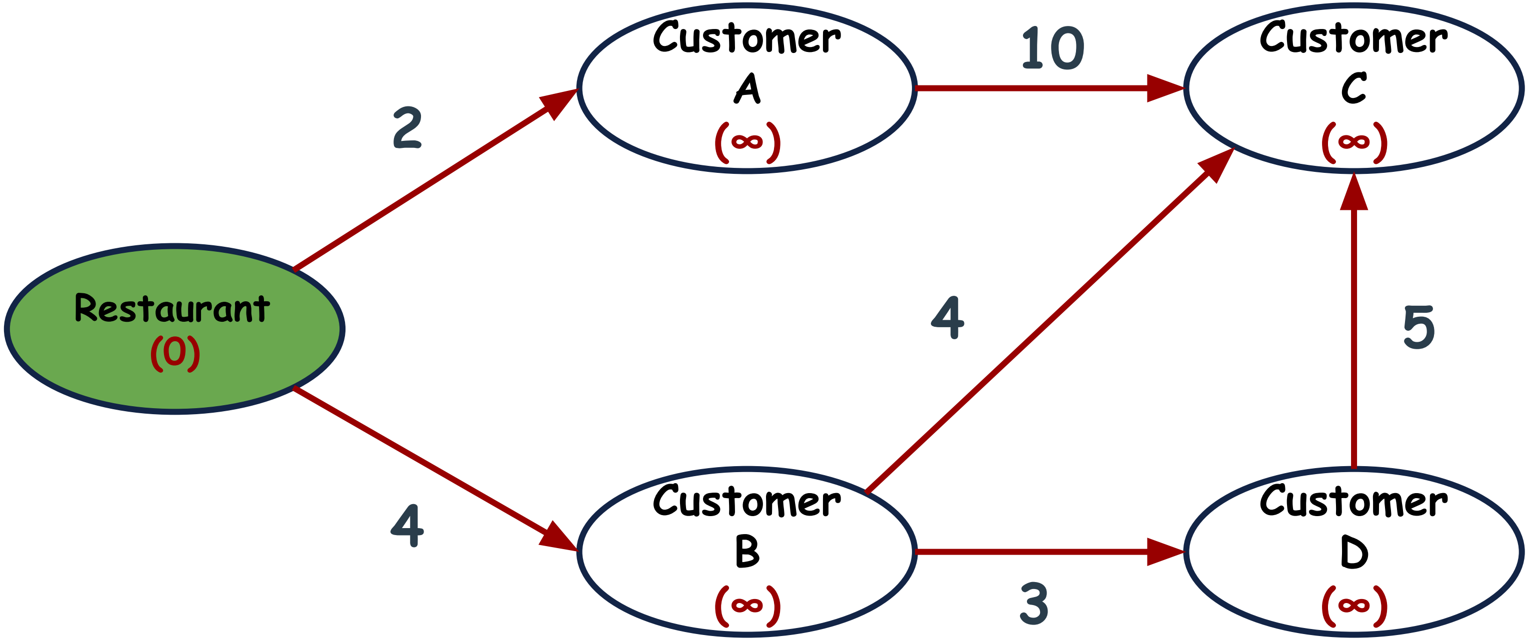
Graphs: Problem

We will start from the starting vertex (Restaurant), mark it as visited and check all of its unvisited adjacent vertices



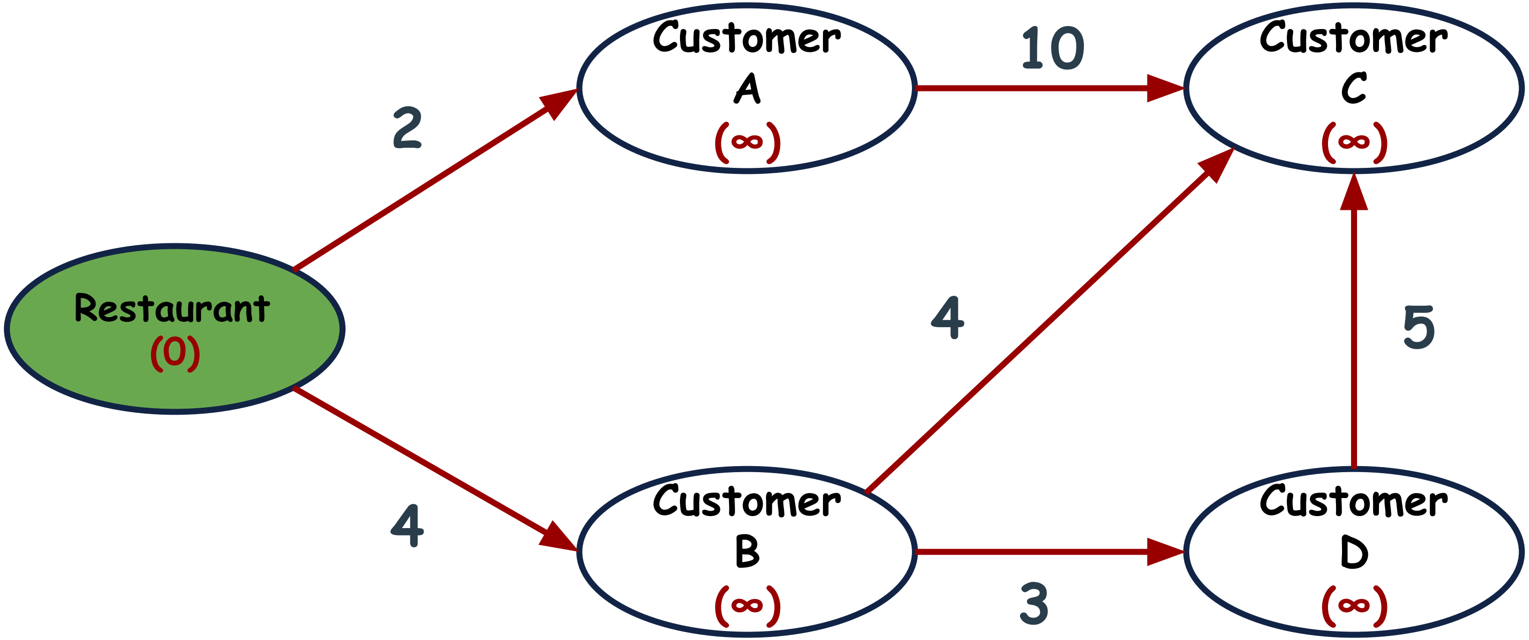
Graphs: Problem

Now, we will update (relax) the cost to reach the customers house using this formula.



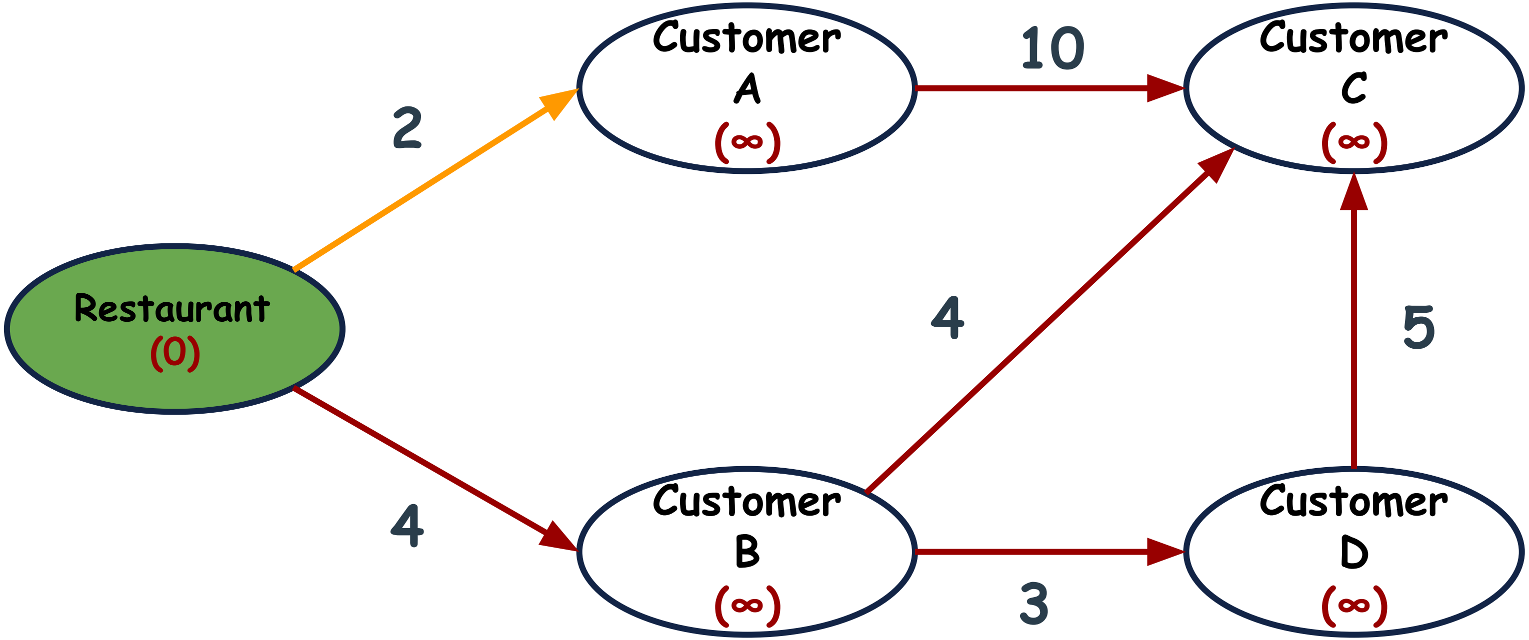
Graphs: Problem

```
if ((cost(curr) + weight) < cost(adj))
{   cost(adj) = cost(curr) + weight
}
```



Graphs: Problem

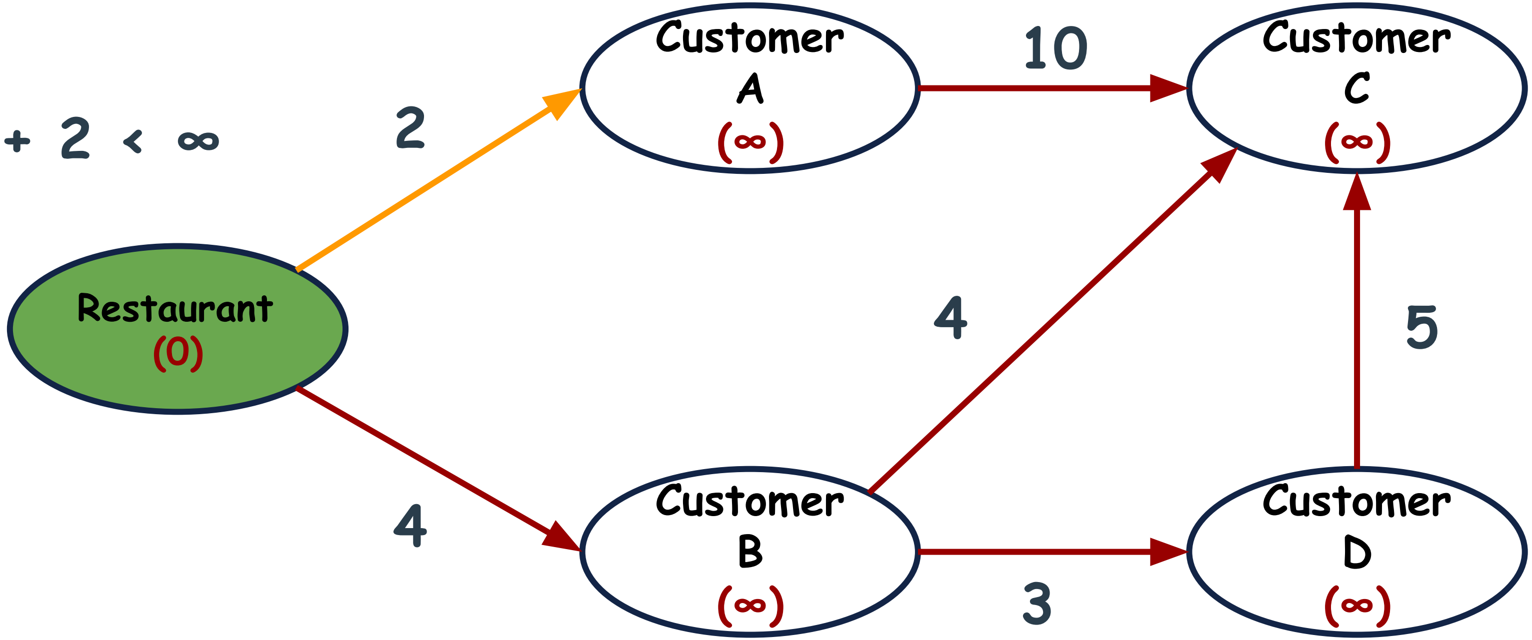
```
if ((cost(curr) + weight) < cost(adj))
{   cost(adj) = cost(curr) + weight
}
```



Graphs: Problem

```
if ((cost(curr) + weight) < cost(adj))
{   cost(adj) = cost(curr) + weight
}
```

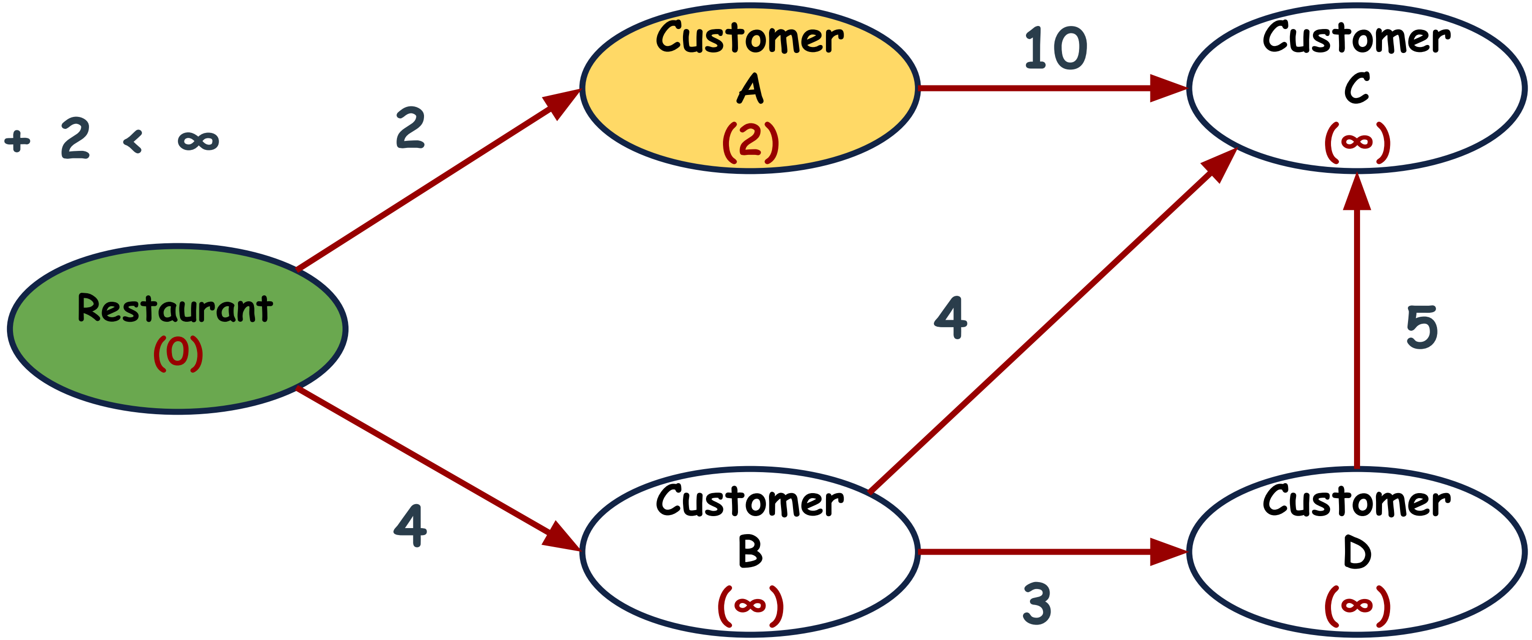
$0 + 2 < \infty$



Graphs: Problem

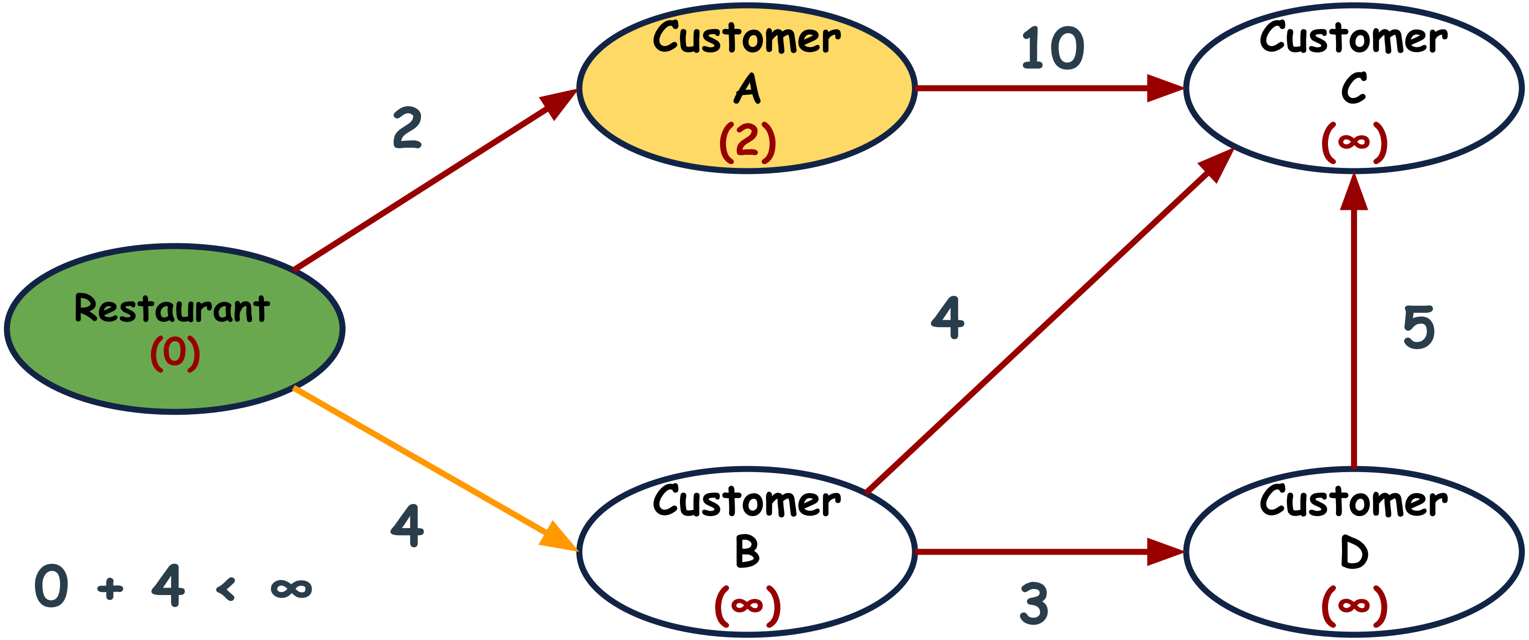
```
if ((cost(curr) + weight) < cost(adj))
{   cost(adj) = cost(curr) + weight
}
```

$0 + 2 < \infty$



Graphs: Problem

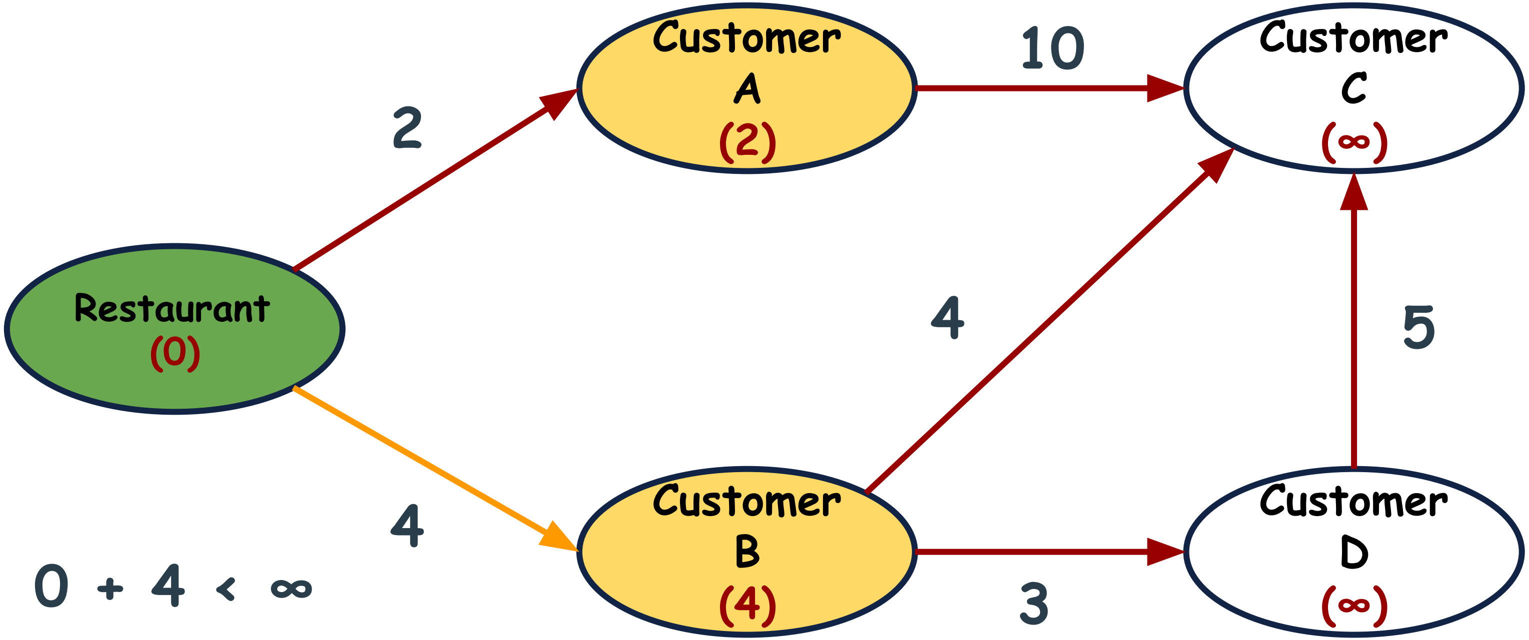
```
if ((cost(curr) + weight) < cost(adj))
{   cost(adj) = cost(curr) + weight
}
```



$0 + 4 < \infty$

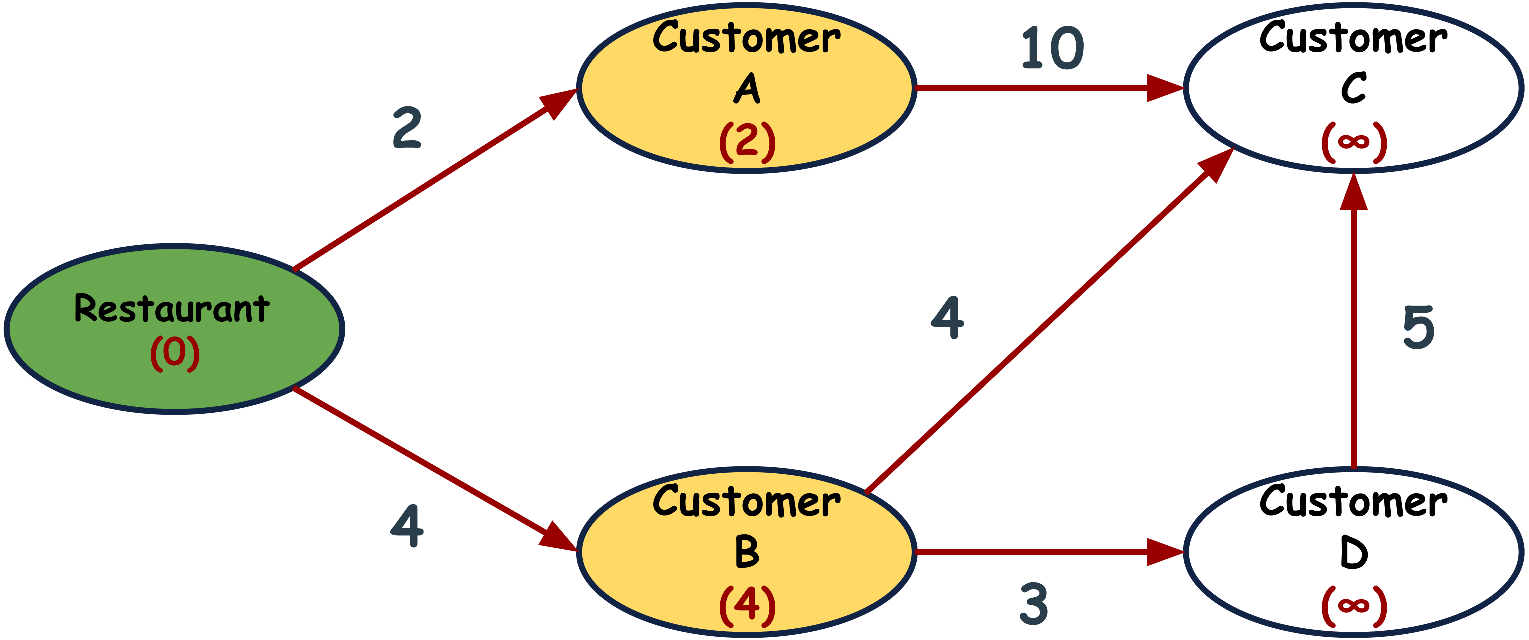
Graphs: Problem

```
if ((cost(curr) + weight) < cost(adj))
{   cost(adj) = cost(curr) + weight
}
```



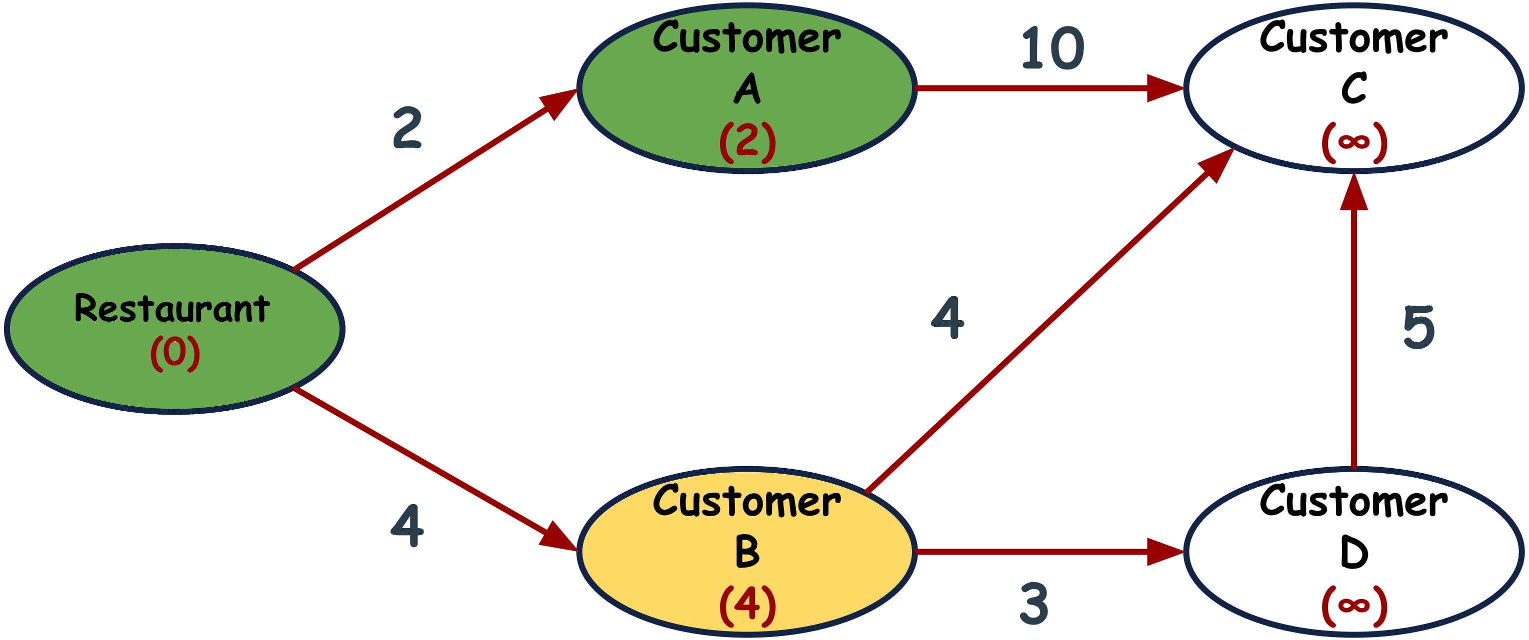
Graphs: Problem

Now, we will choose from the relaxed vertices, the one with the minimum cost.



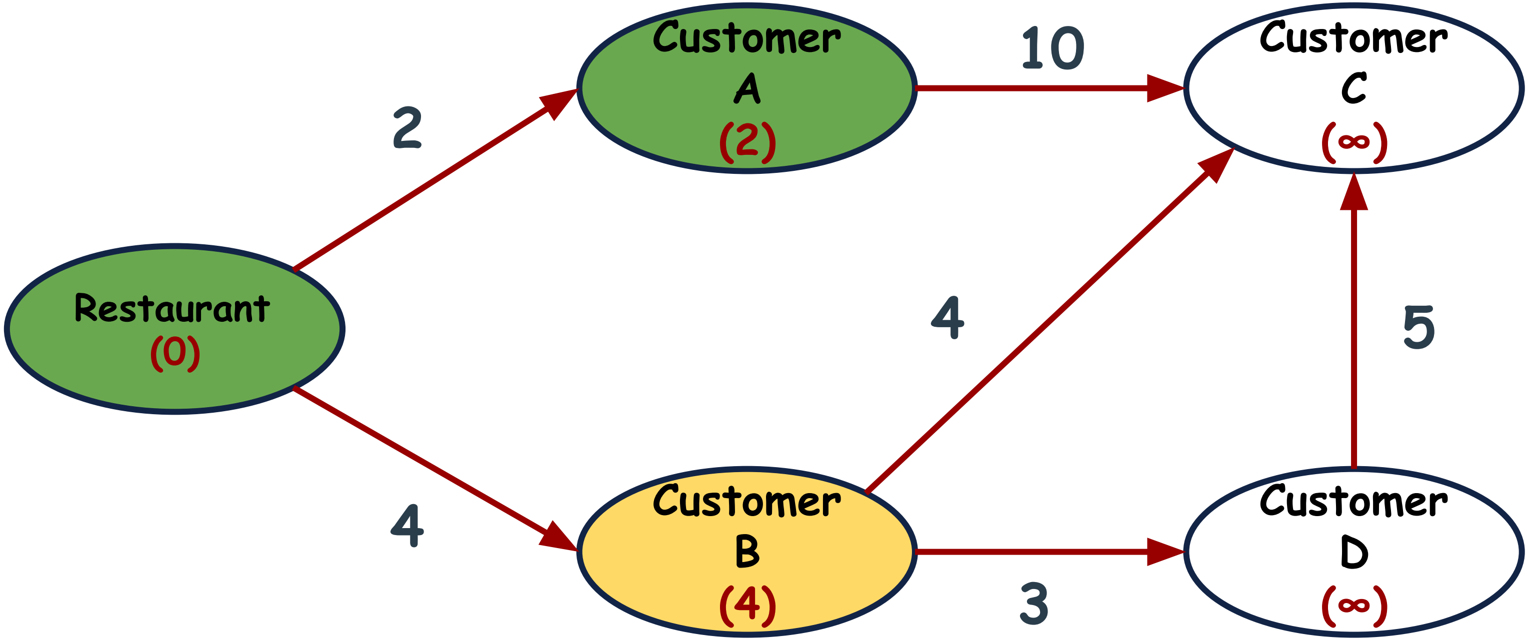
Graphs: Problem

We, will mark it as visited.



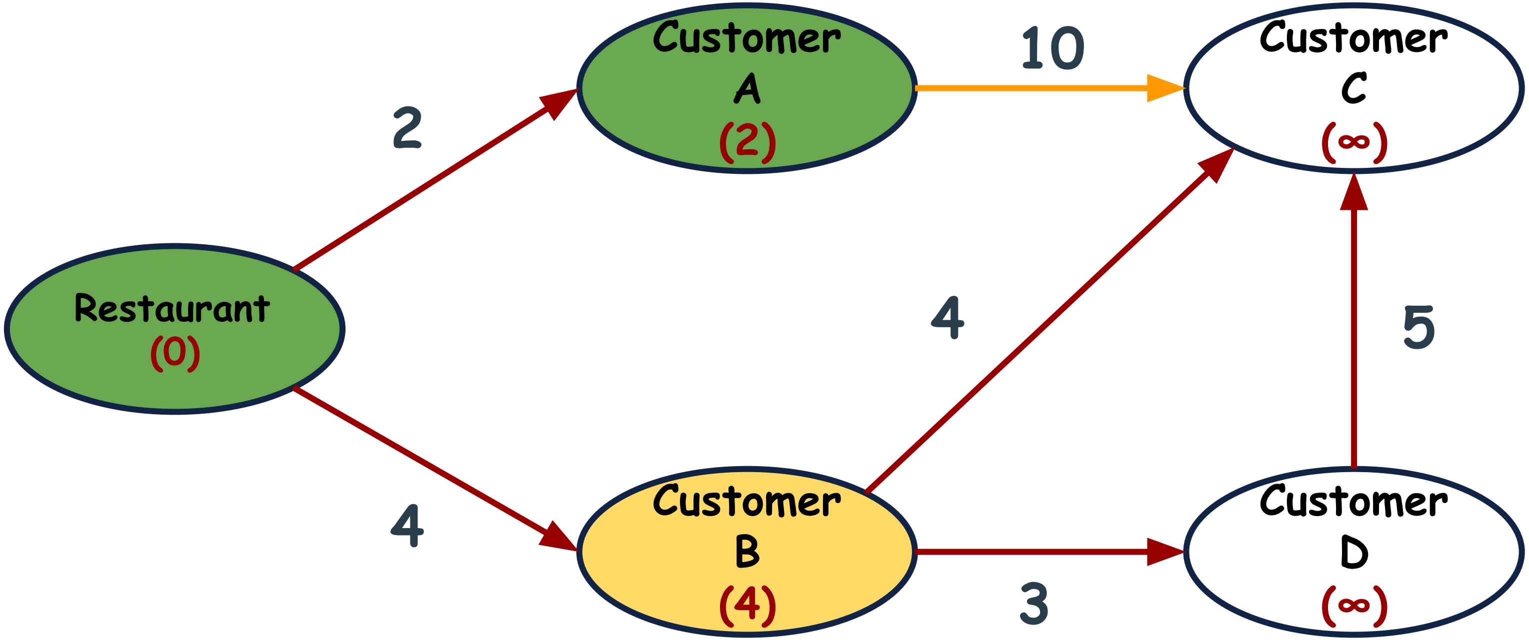
Graphs: Problem

Now, we will follow the same process for all vertices.



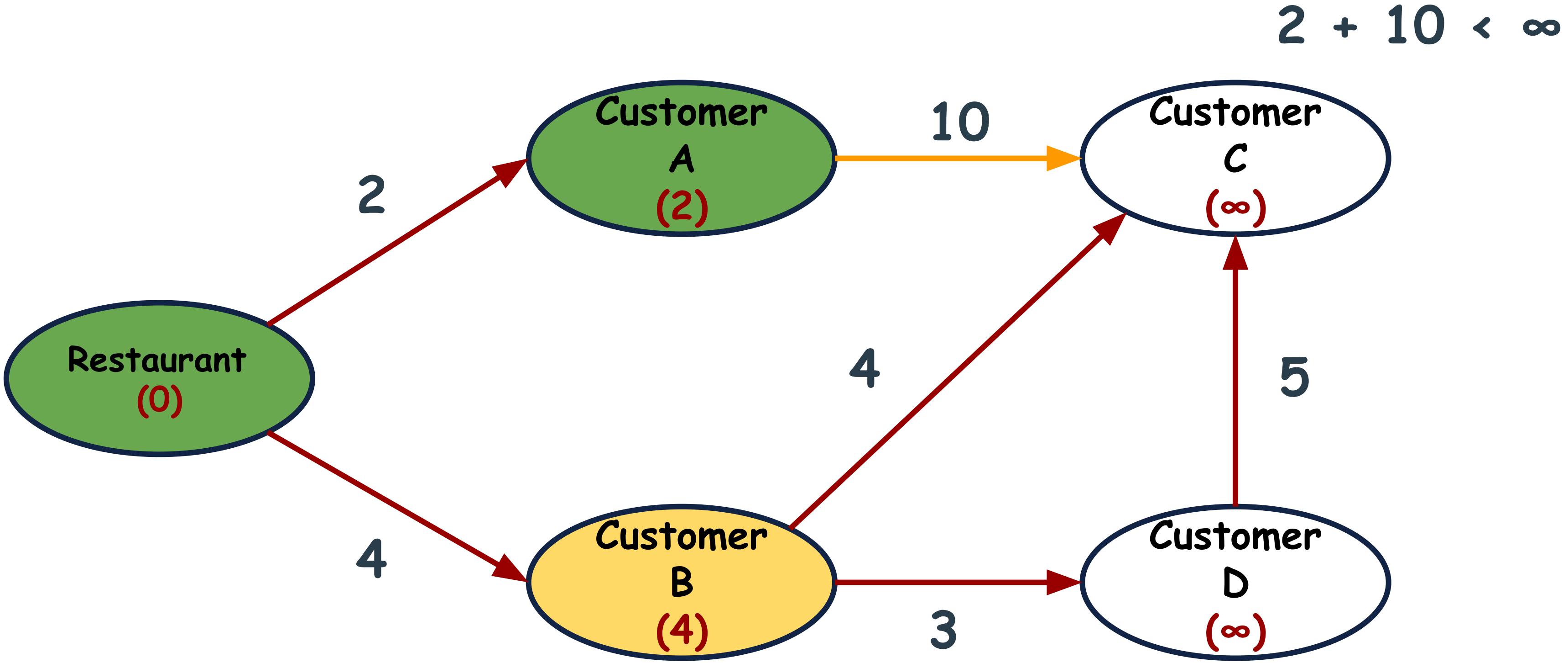
Graphs: Problem

Now, we will follow the same process for all vertices.



Graphs: Problem

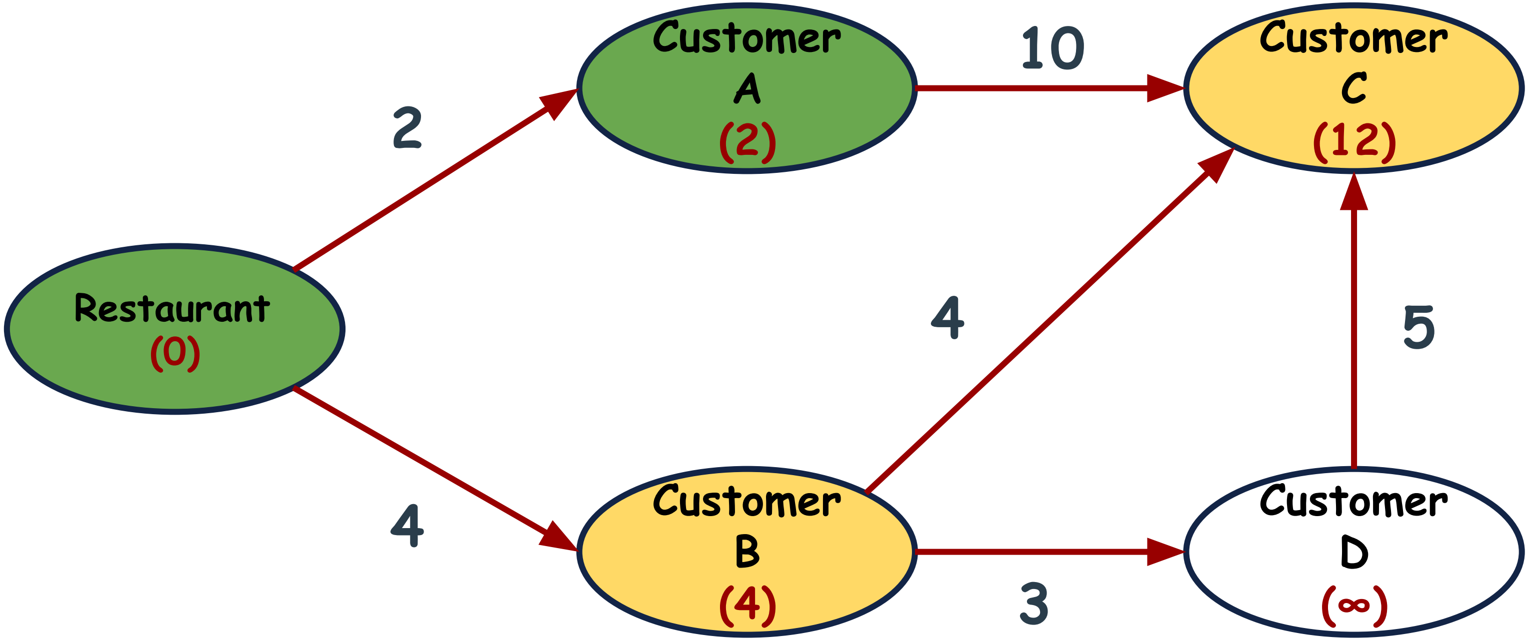
Now, we will follow the same process for all vertices.



Graphs: Problem

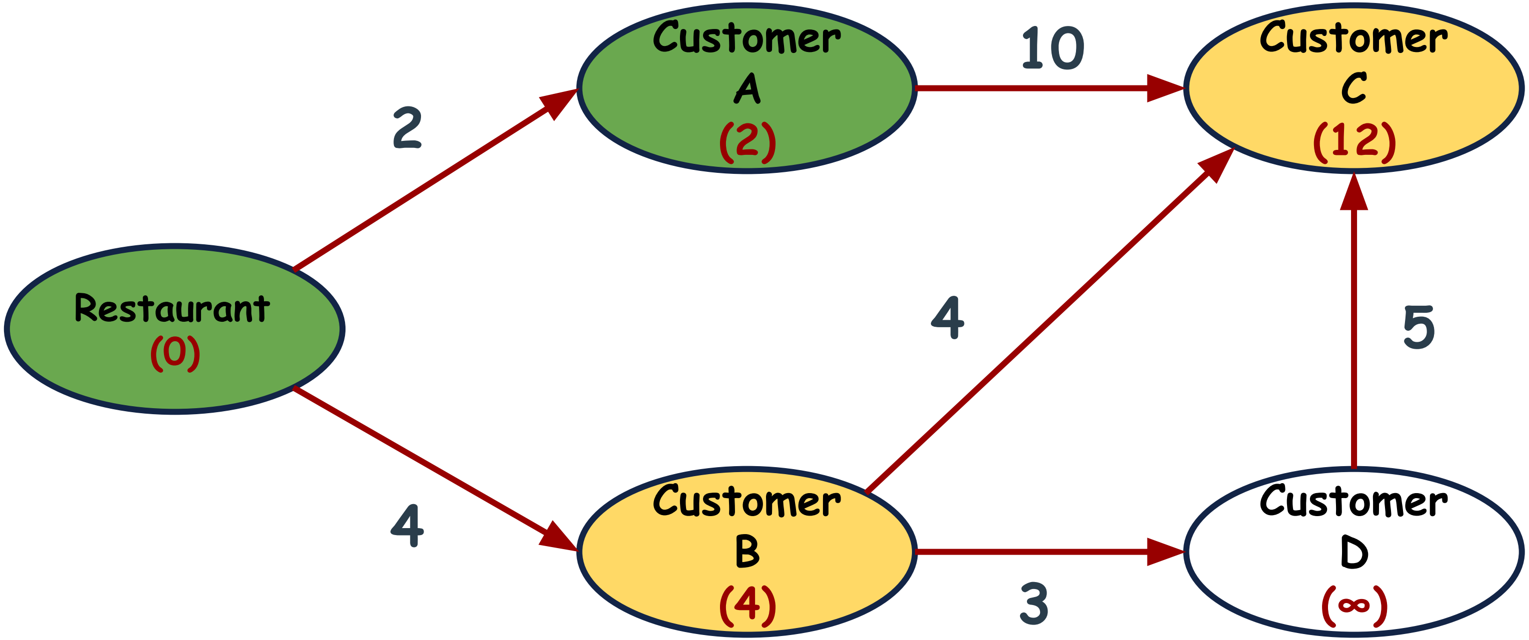
Now, we will follow the same process for all vertices.

$2 + 10 < \infty$



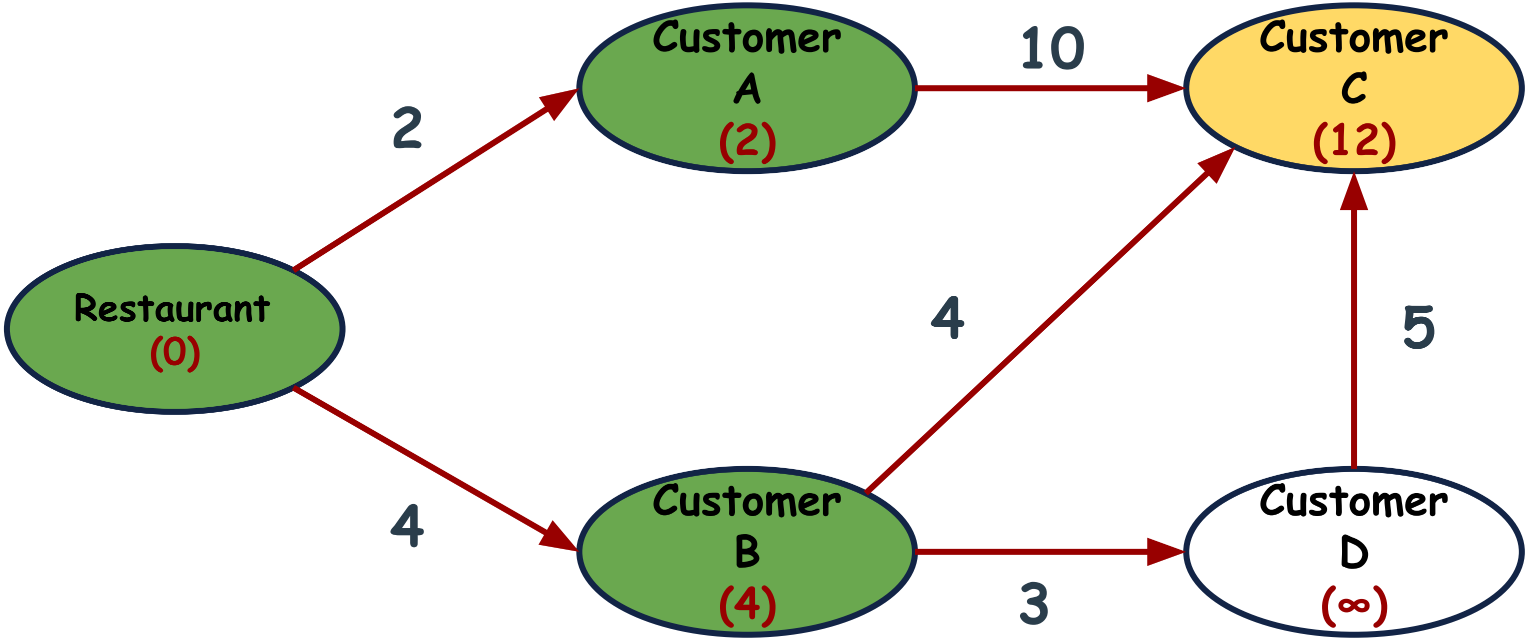
Graphs: Problem

Now, from the relaxed nodes, choose the one with minimum cost.



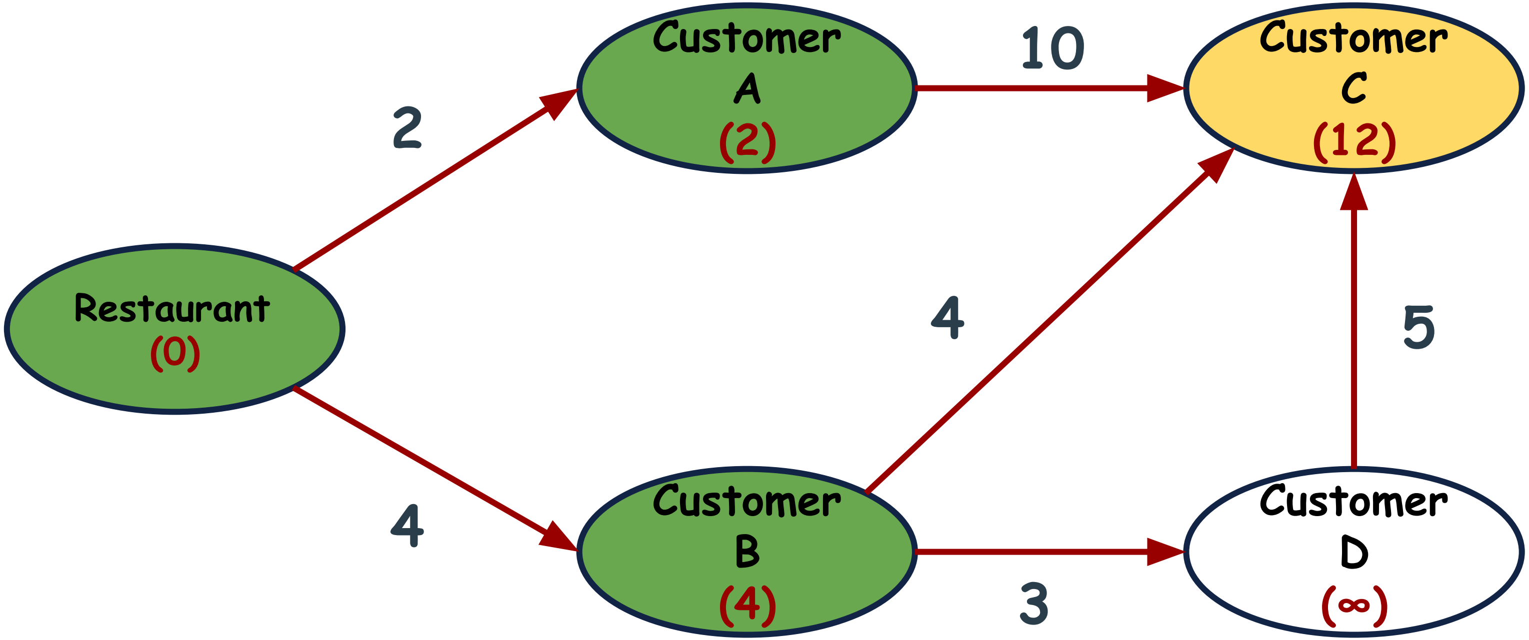
Graphs: Problem

Marked it as visited.



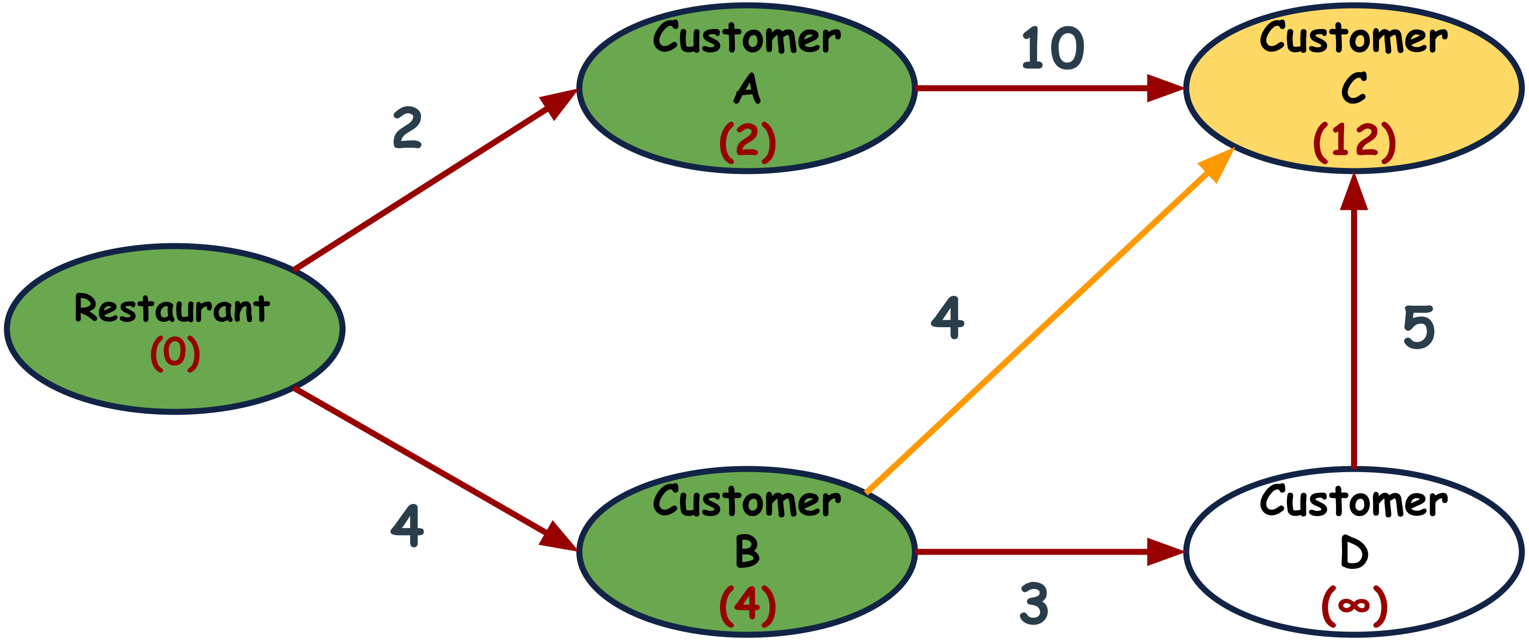
Graphs: Problem

Check all the adjacent unvisited nodes.



Graphs: Problem

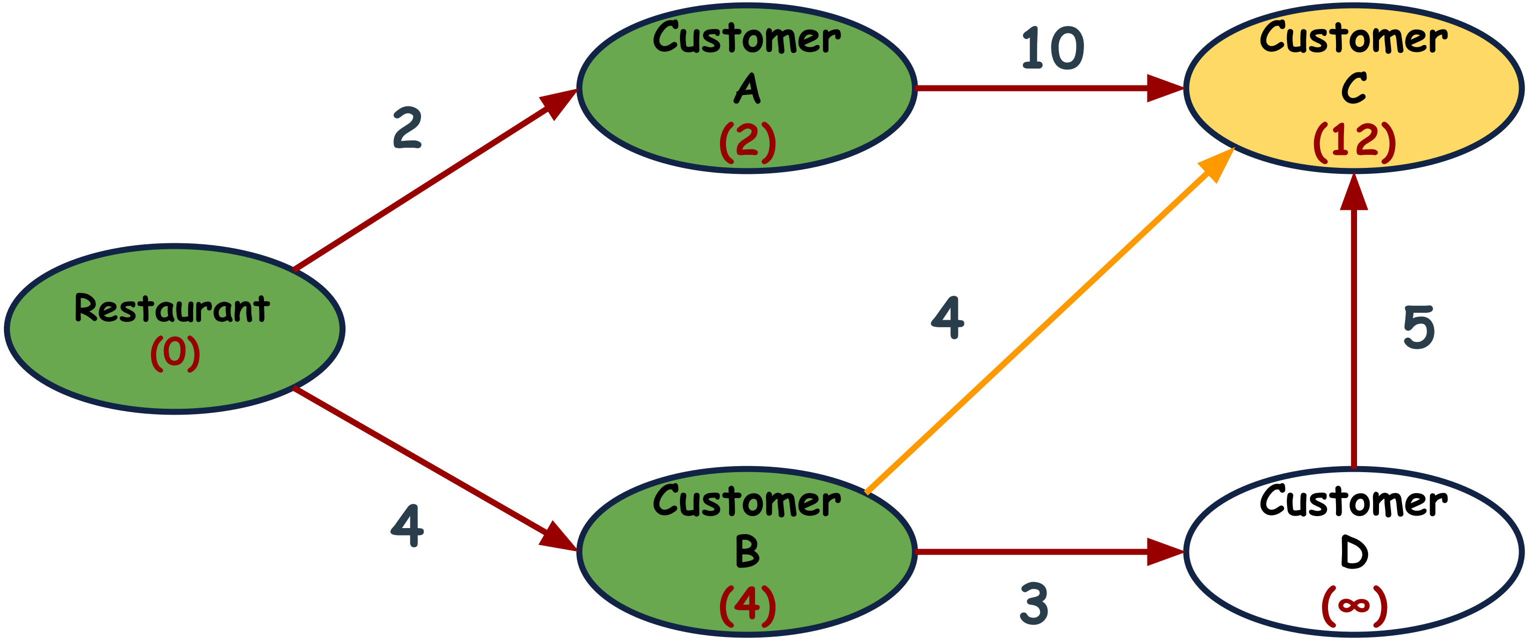
Check all the adjacent unvisited nodes.



Graphs: Problem

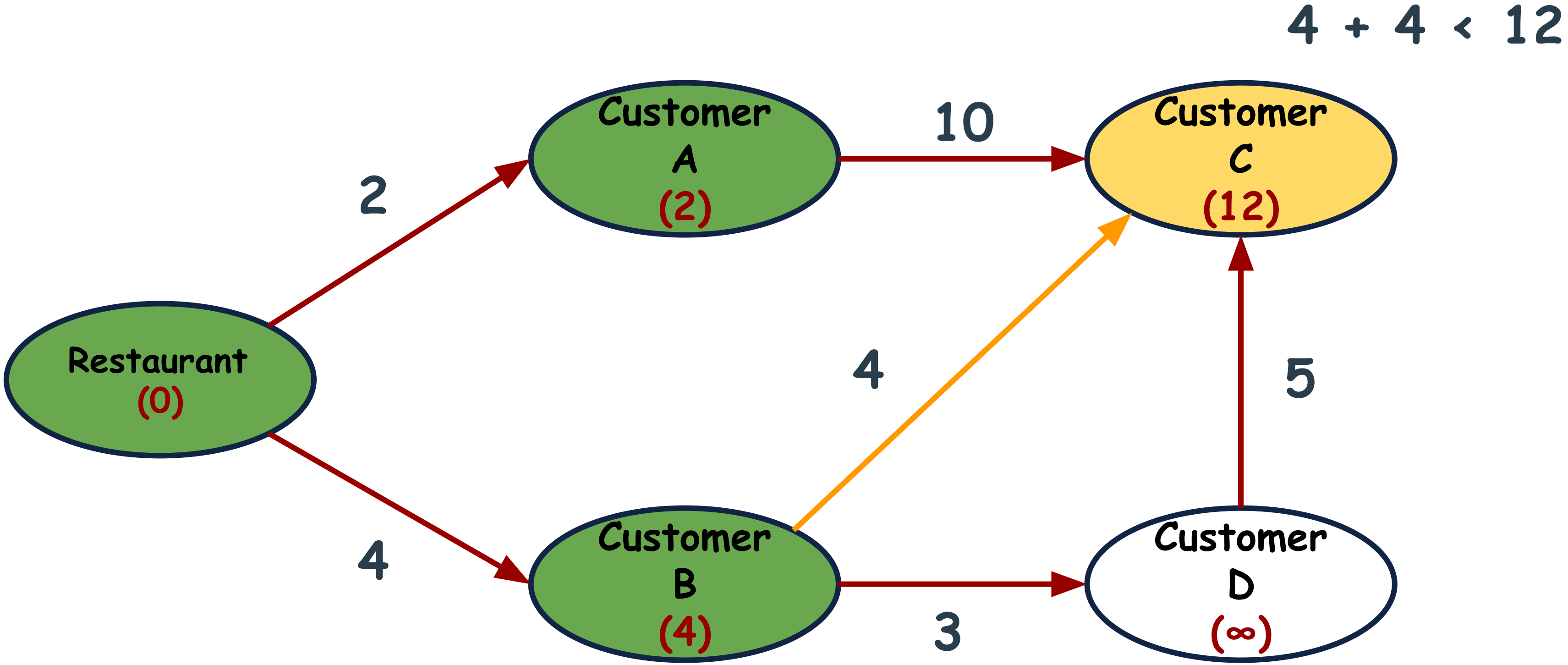
Check all the adjacent unvisited nodes.

$4 + 4 < 12$



Graphs: Problem

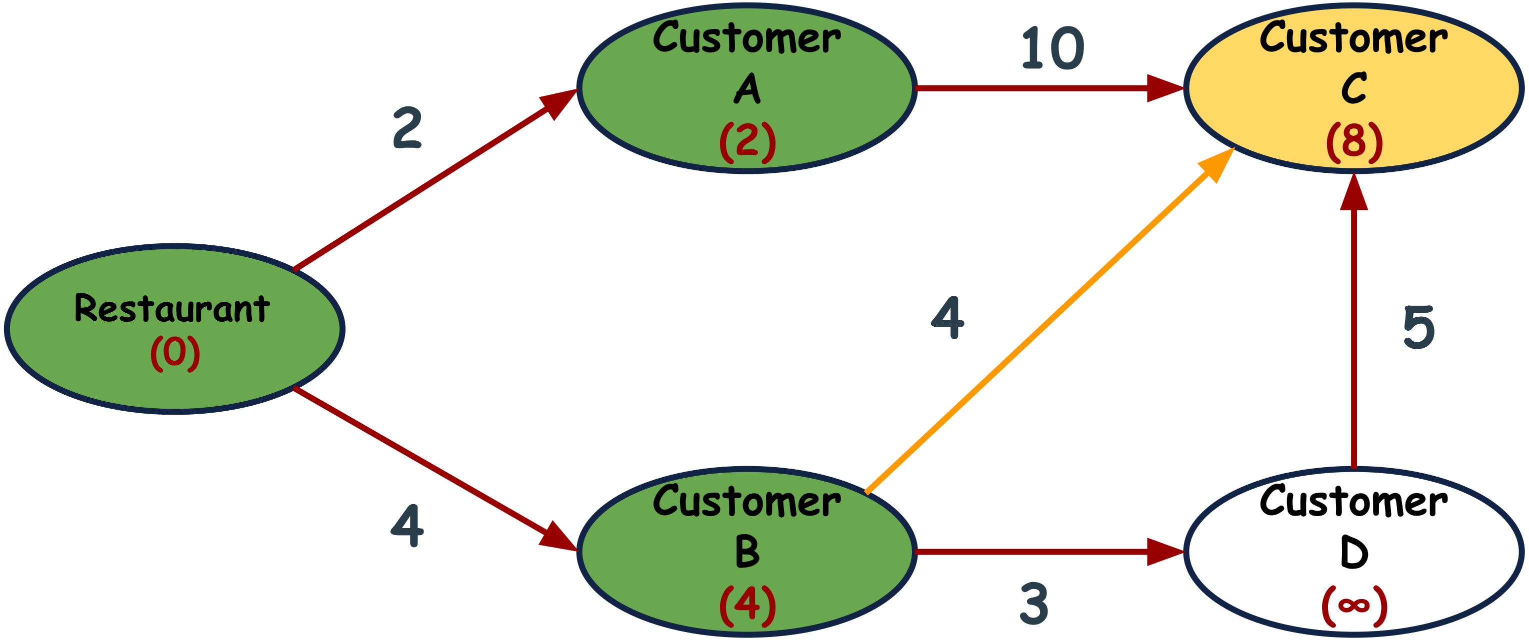
Relax the node.



Graphs: Problem

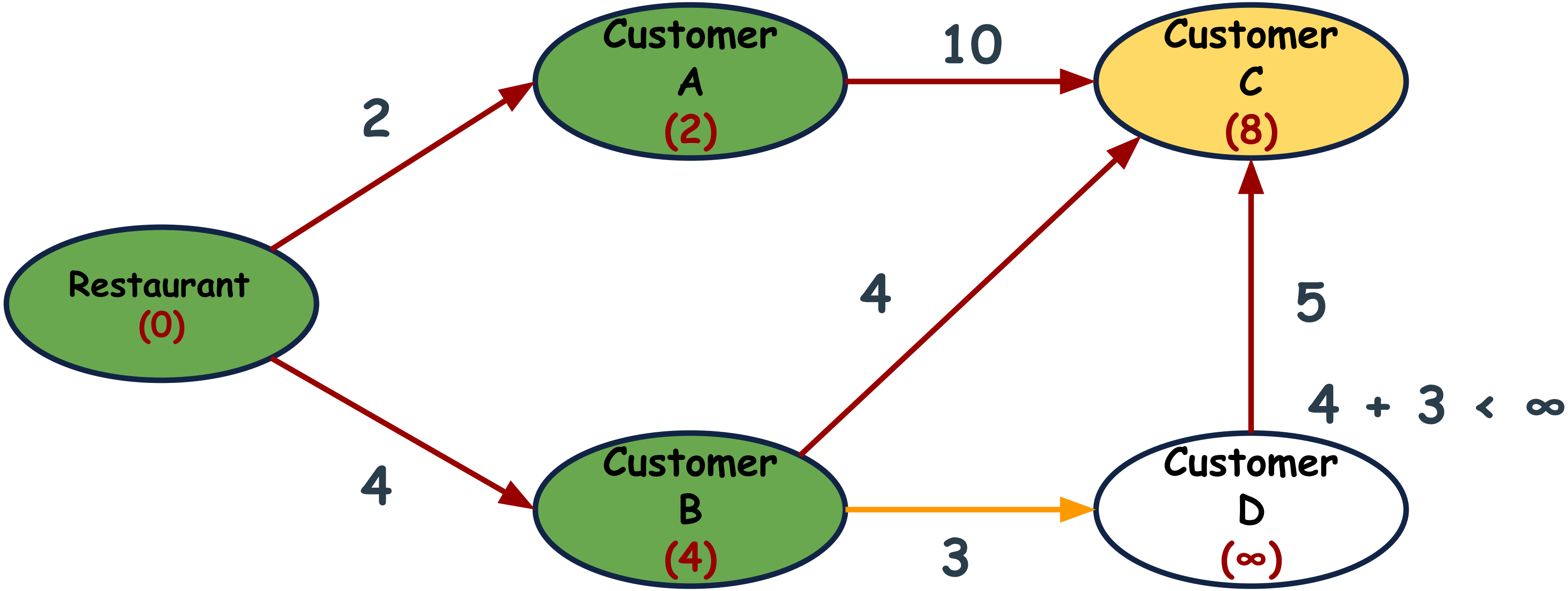
Relax the node.

$4 + 4 < 12$



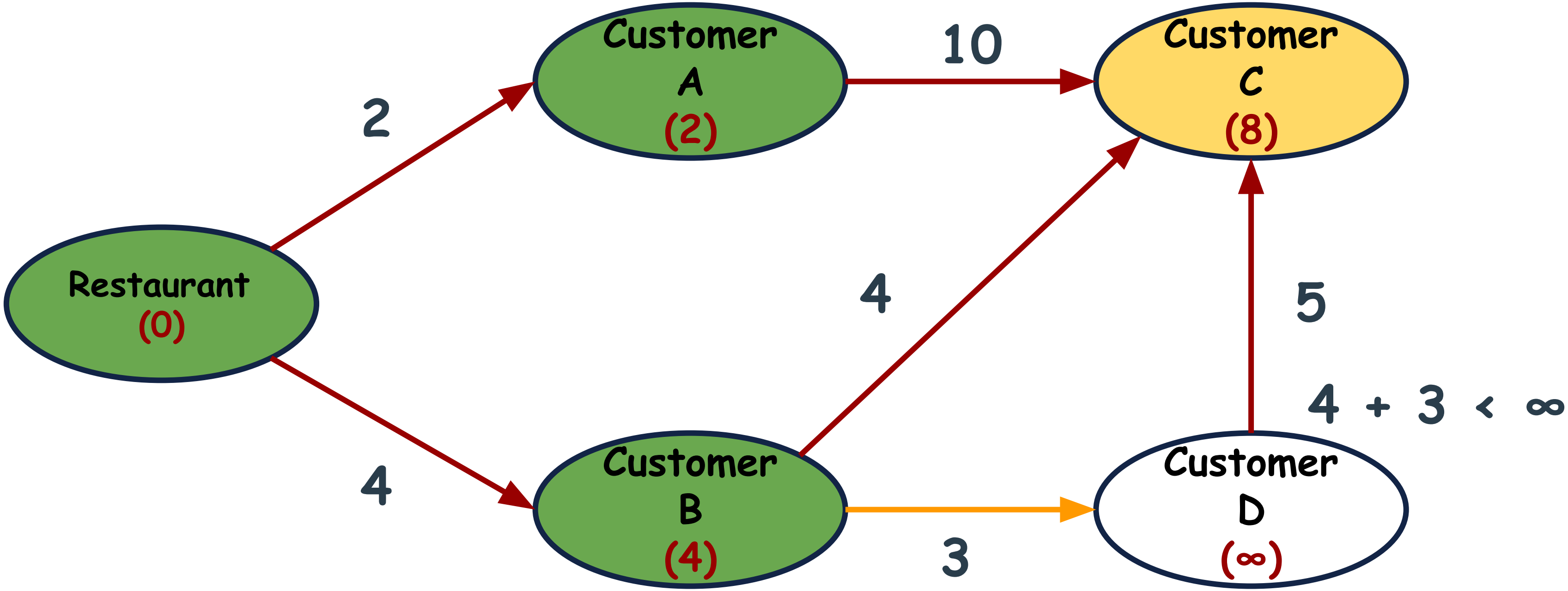
Graphs: Problem

Check all the adjacent unvisited nodes.



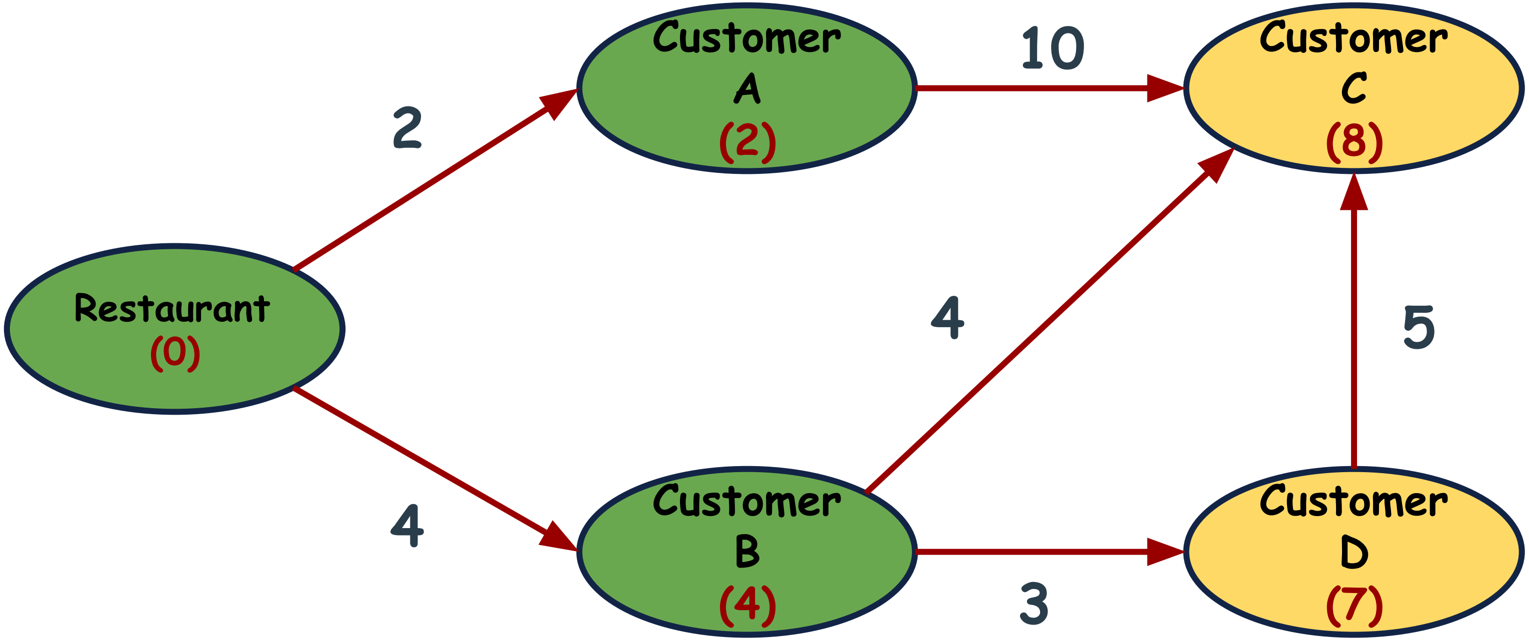
Graphs: Problem

Relax the node.



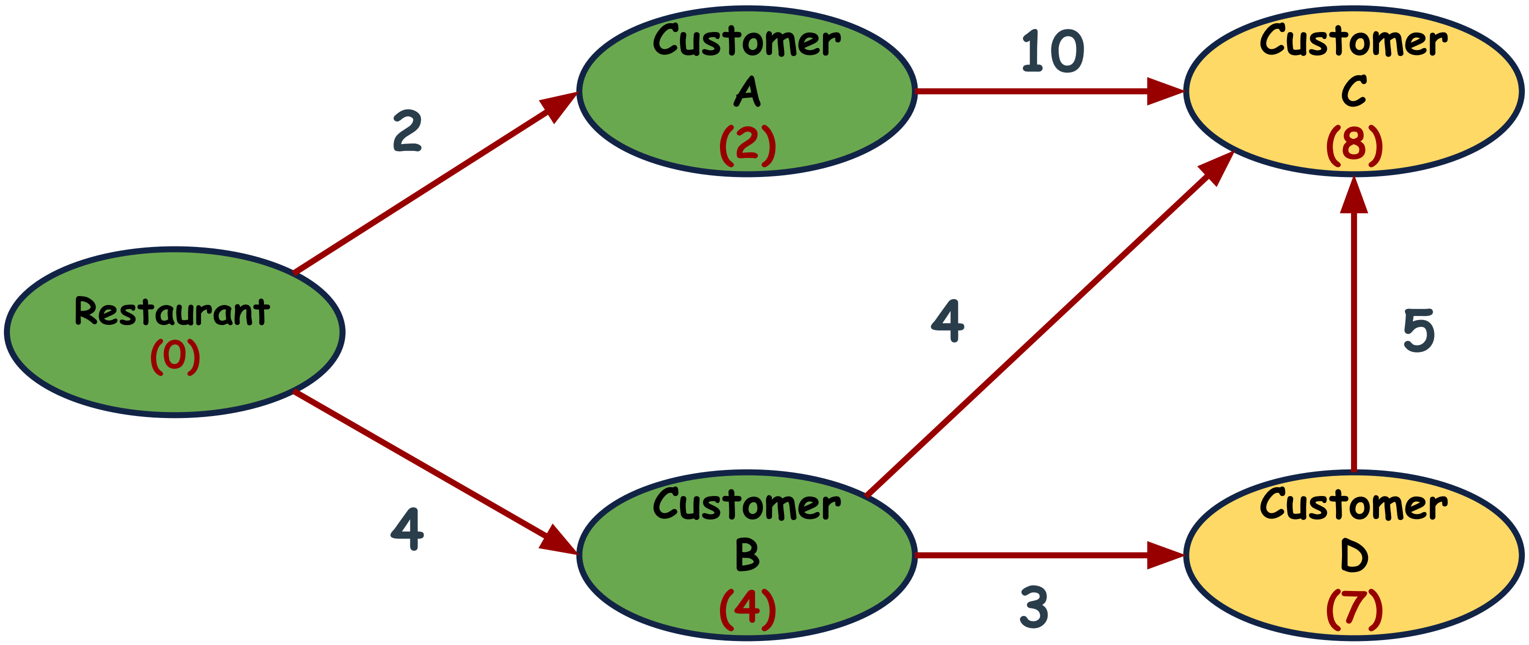
Graphs: Problem

Relax the node.



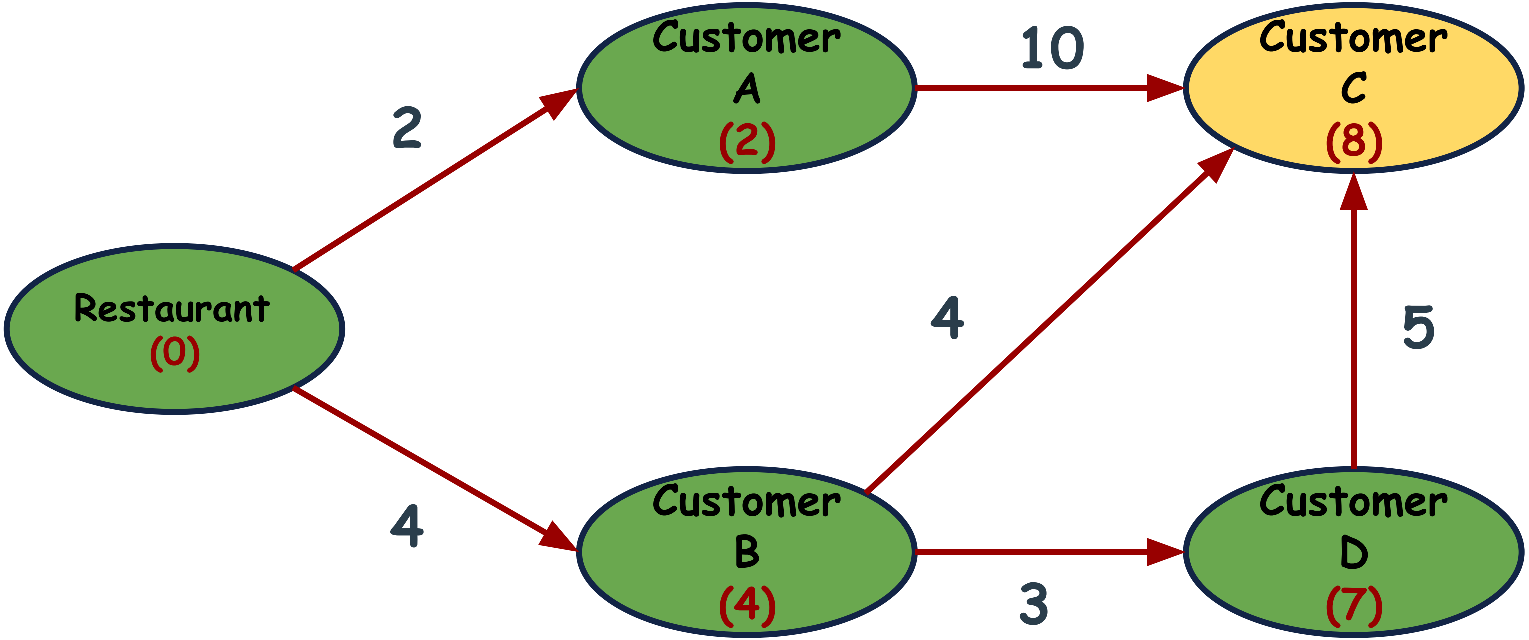
Graphs: Problem

From the relaxed nodes, choose the one with minimum cost.



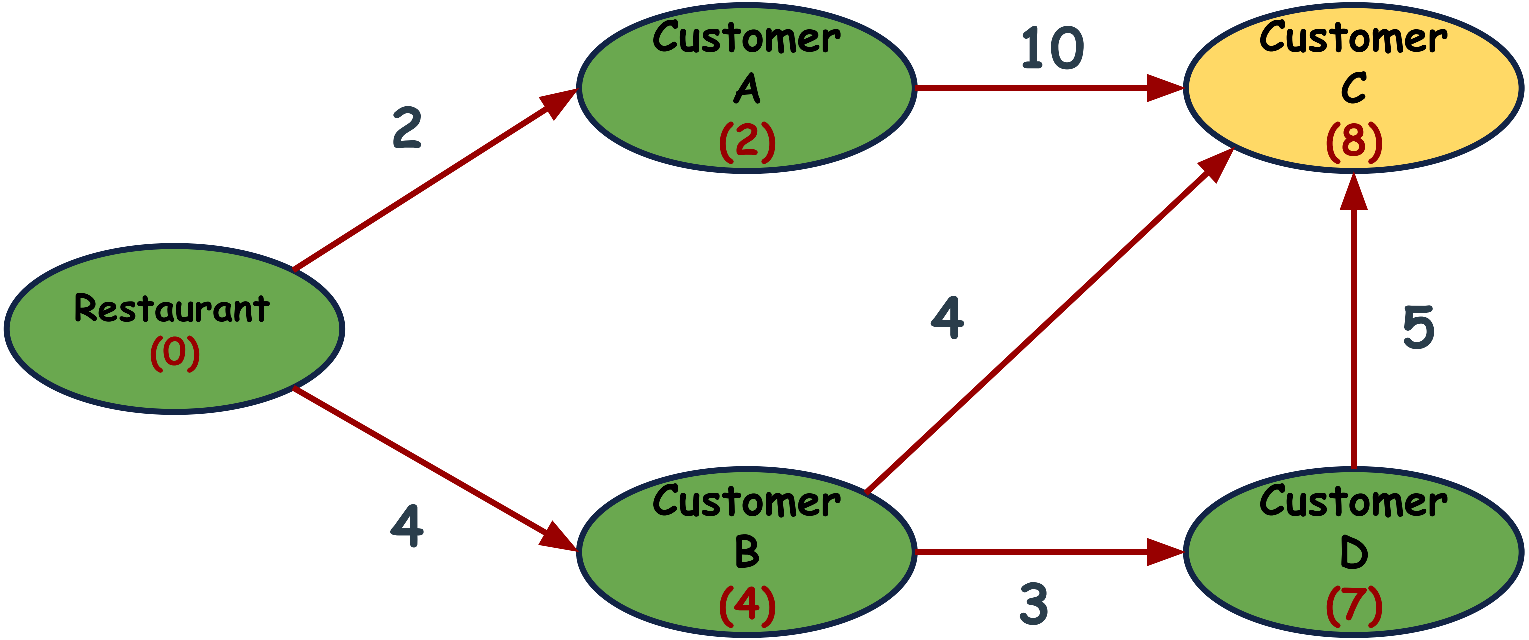
Graphs: Problem

Mark it as visited.



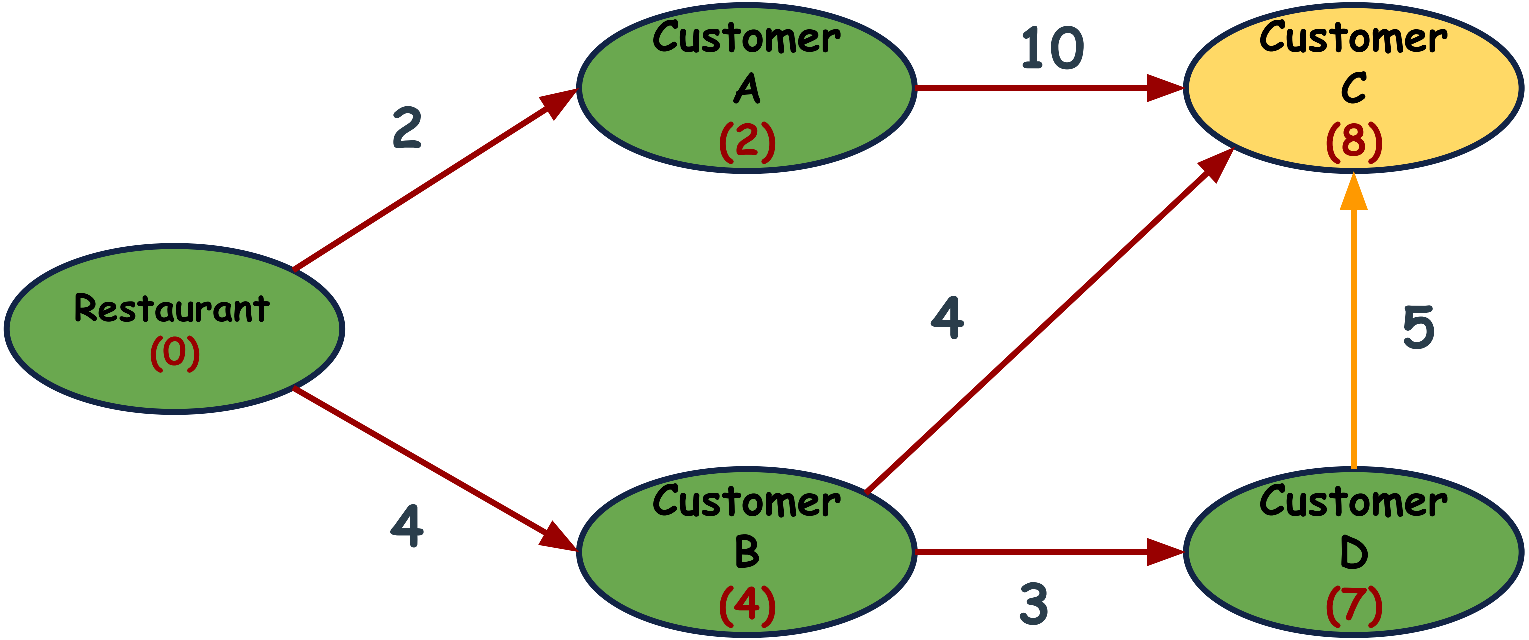
Graphs: Problem

Repeat the same process.



Graphs: Problem

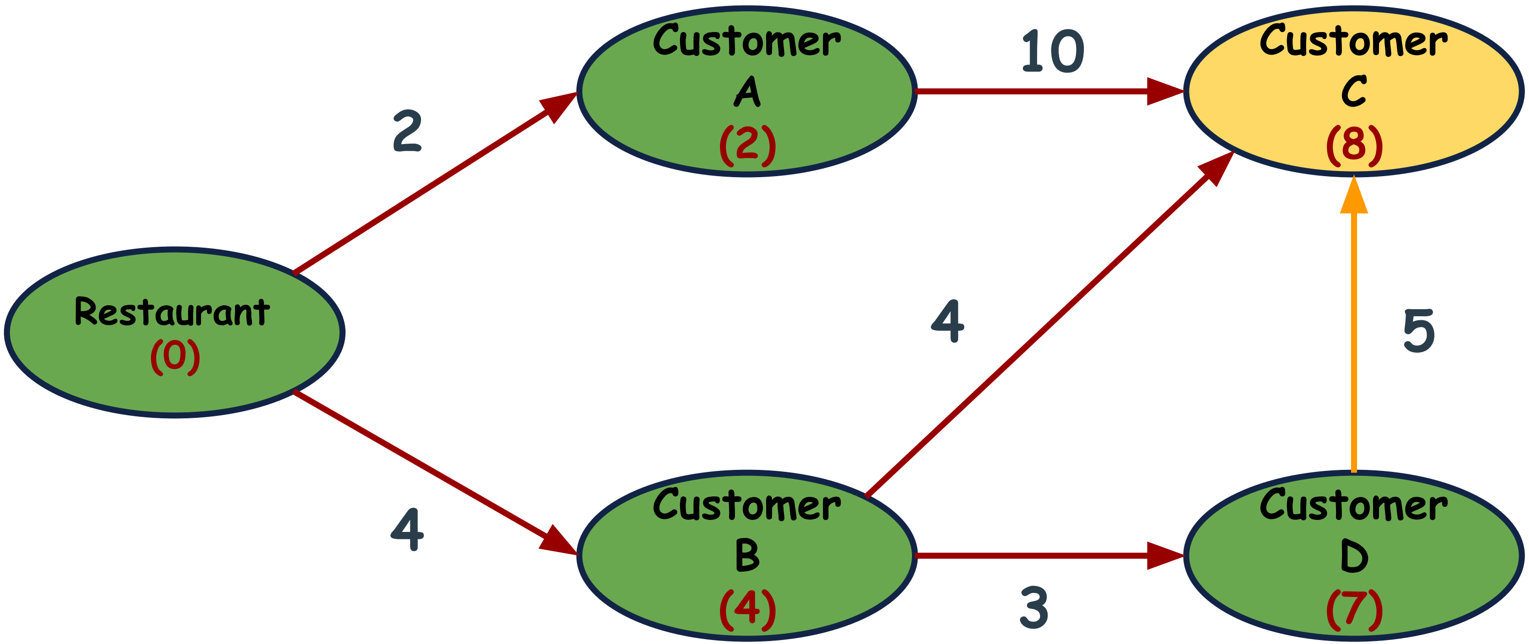
Repeat the same process.



Graphs: Problem

Repeat the same process.

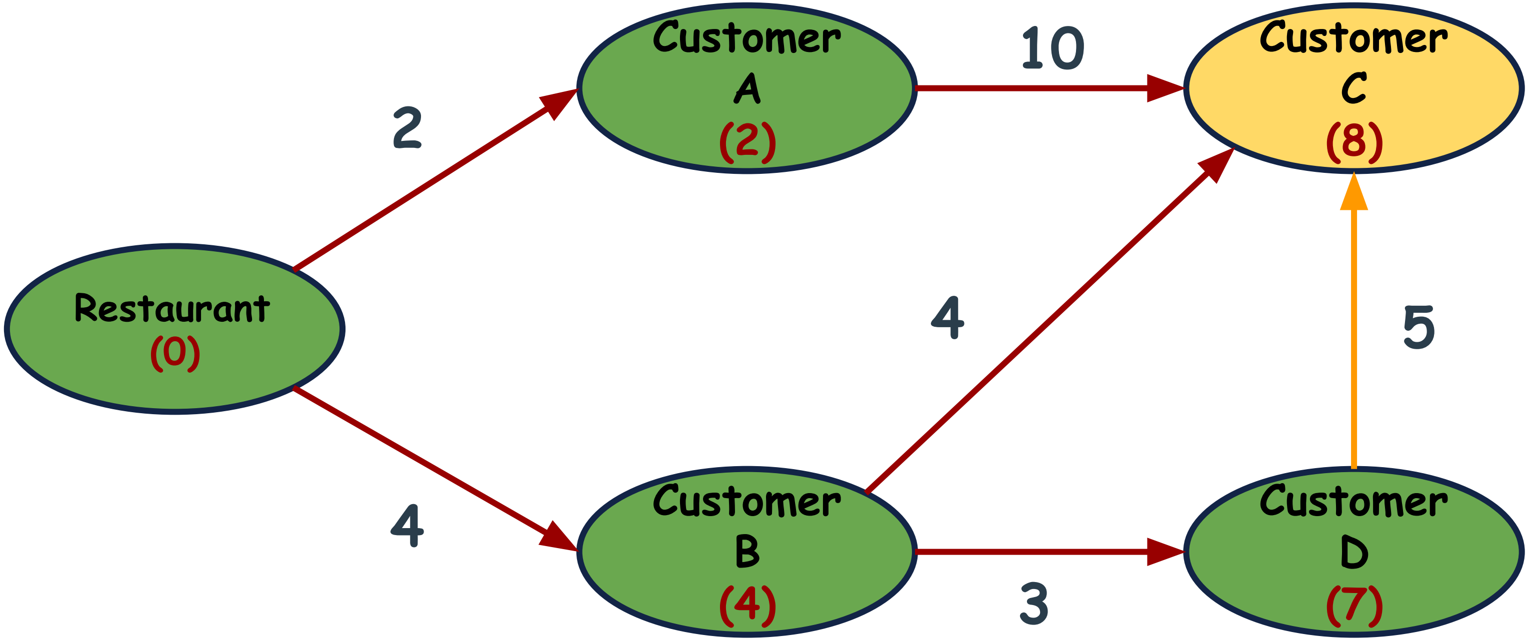
7 + 5 < 8



Graphs: Problem

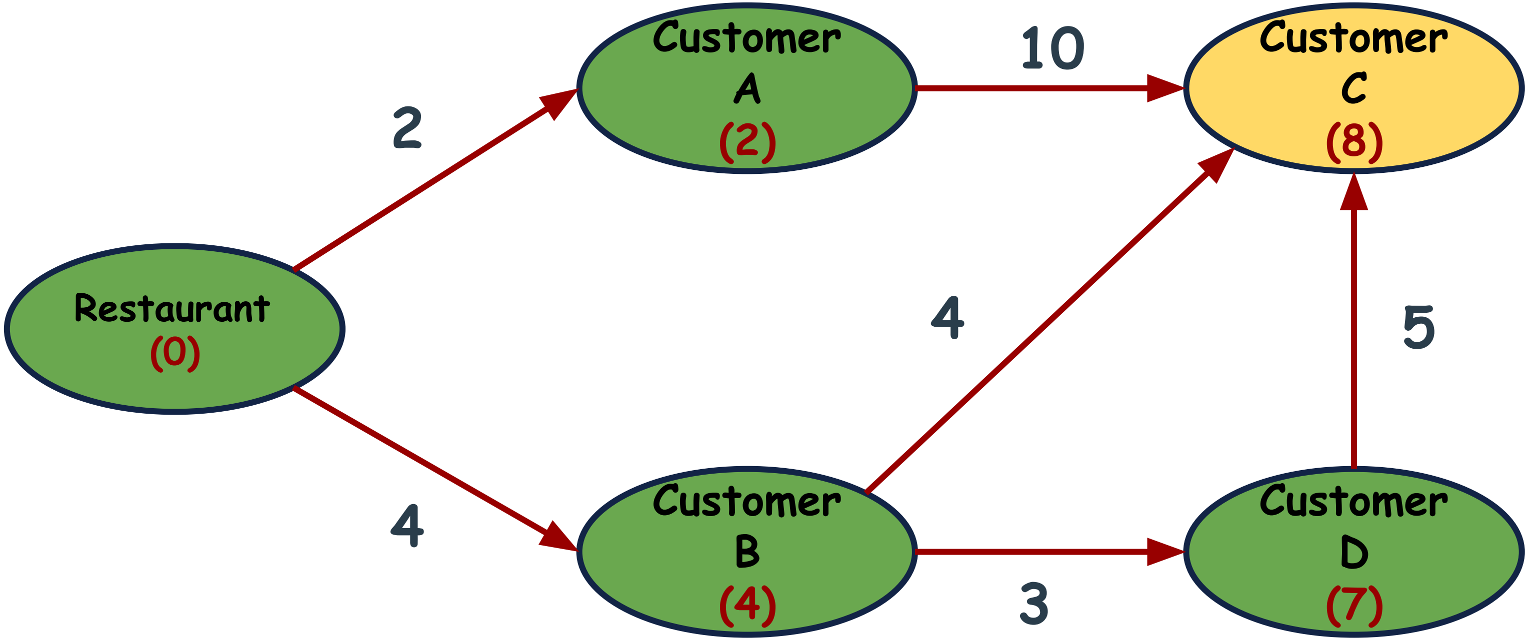
Do not relax the node.

7 + 5 < 8



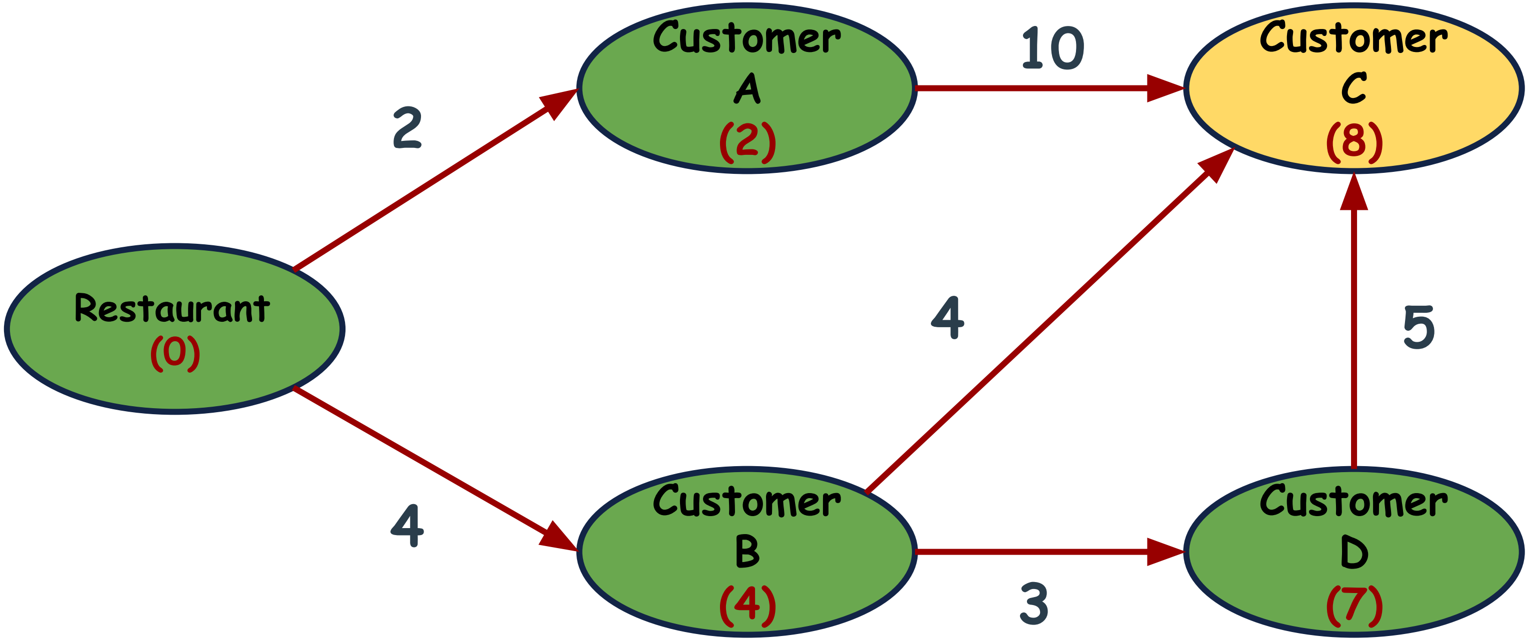
Graphs: Problem

Do not relax the node.



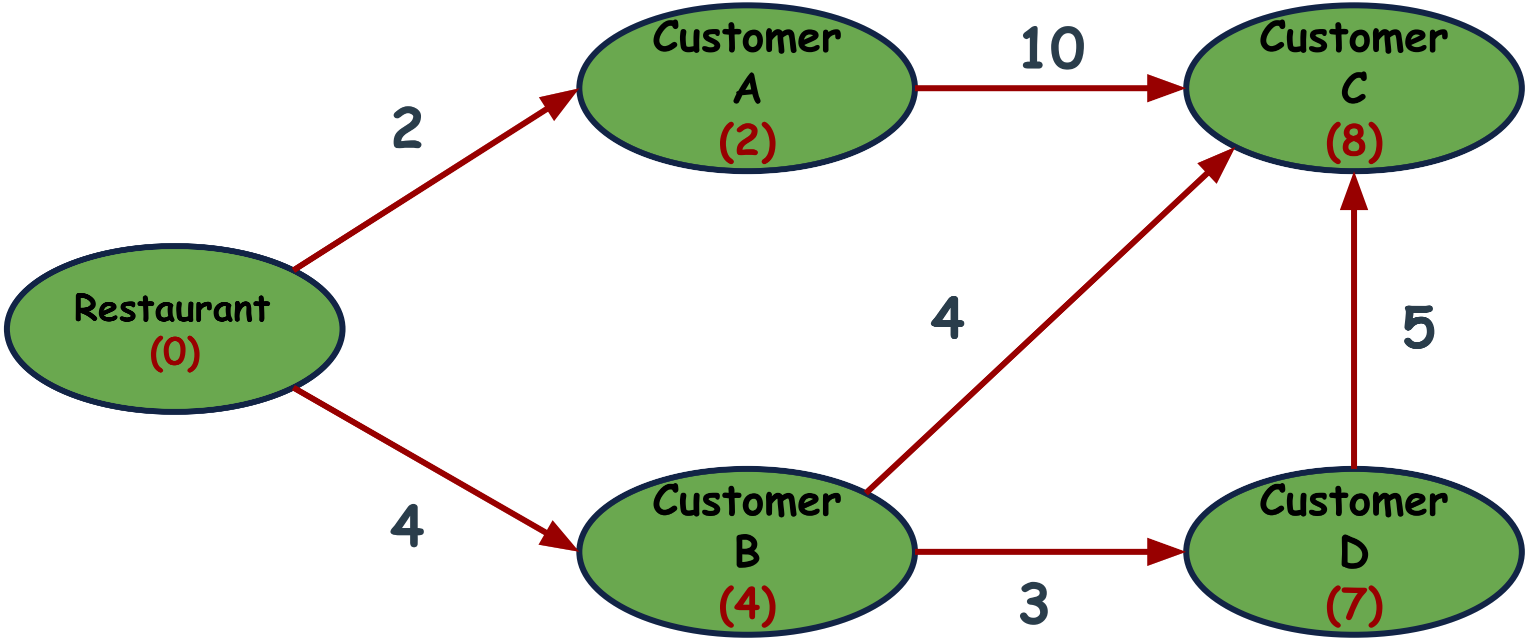
Graphs: Problem

Choose from the relaxed nodes, the one with minimum cost



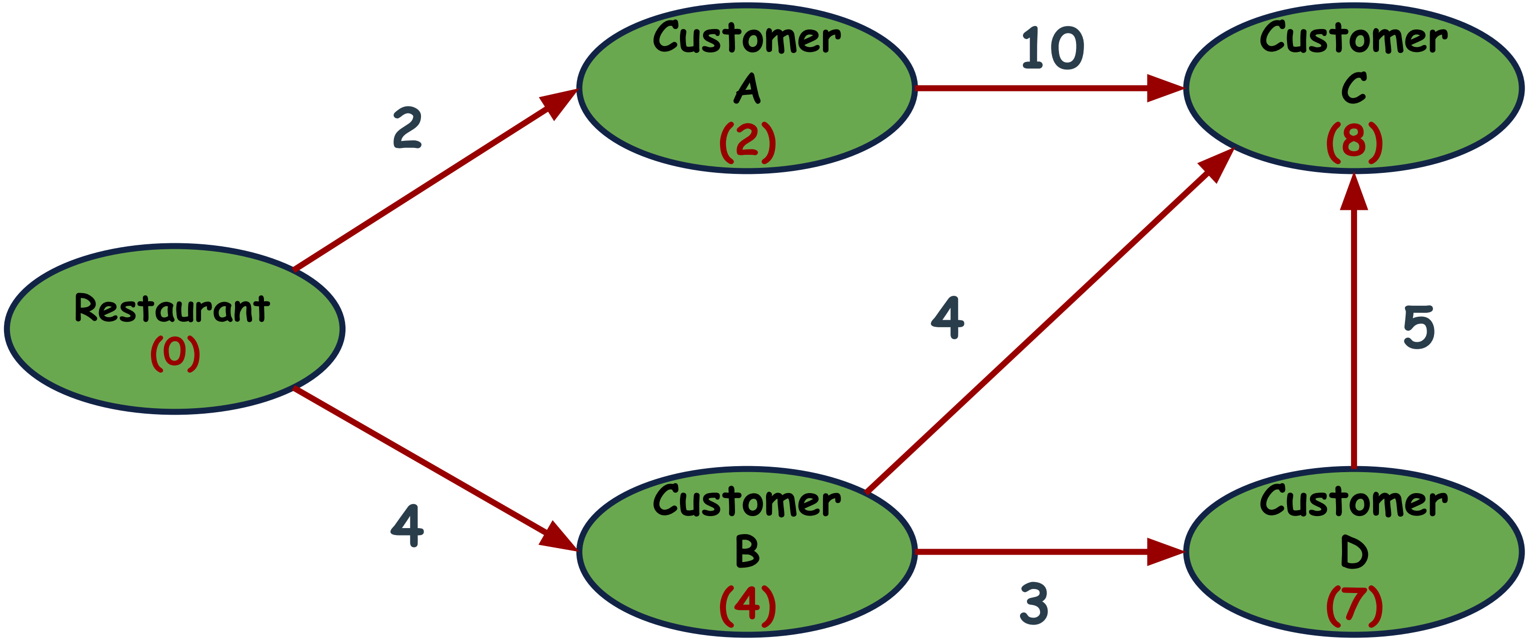
Graphs: Problem

Mark it as visited.



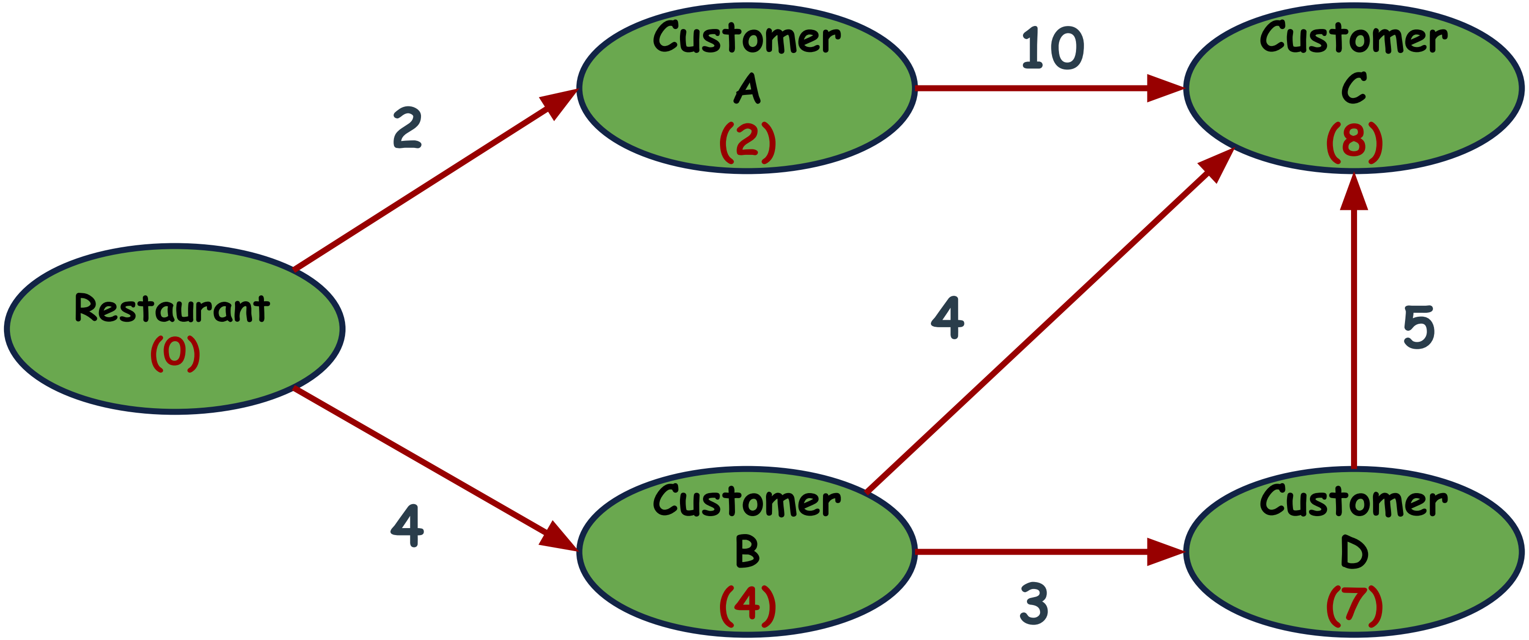
Graphs: Problem

One visited then it means that these nodes can not be further relaxed.



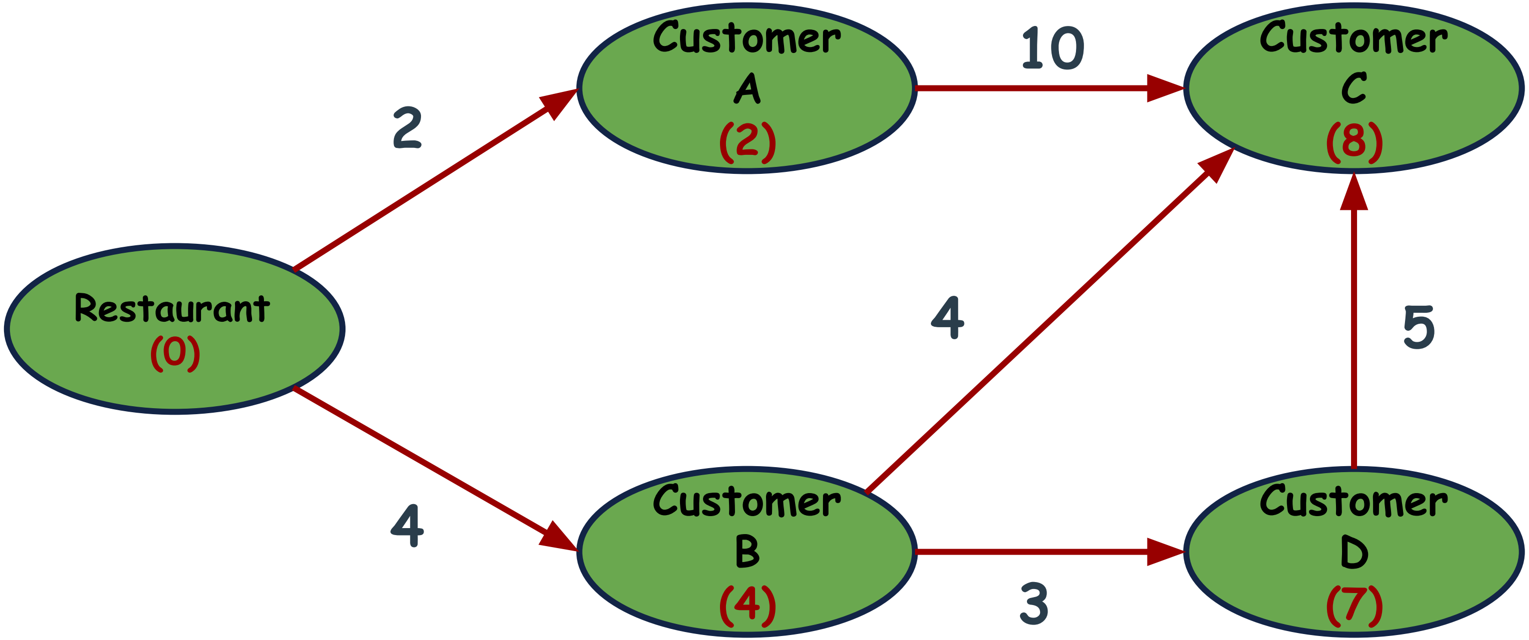
Graphs: Problem

This algorithm is called **Dijkstra Algorithm** (single source shortest Path).



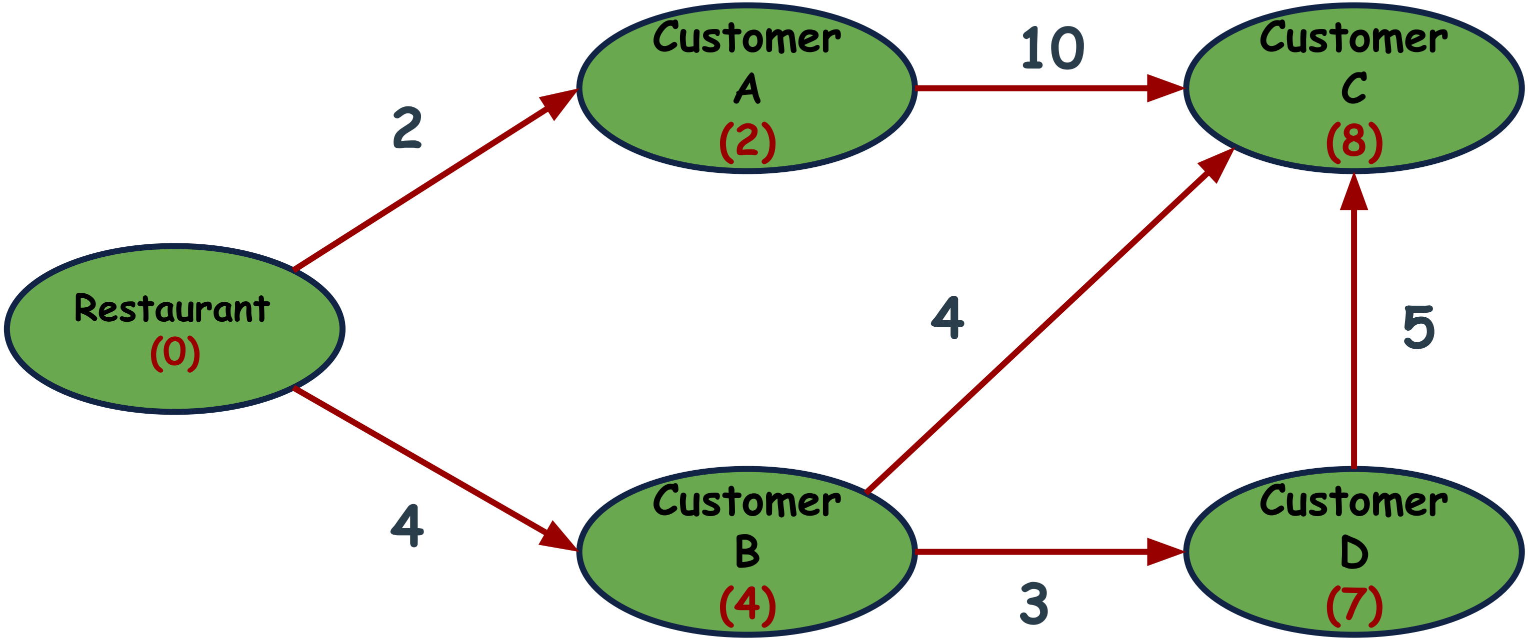
Graphs: Dijkstra Algorithm

This algorithm is called **Dijkstra Algorithm** (single source shortest Path).



Graphs: Dijkstra Algorithm

Let's Implement the Solution now.



Graphs: Dijkstra Algorithm

Let's Implement the Solution now.

```
class Graph
{
    typedef pair<int, string> edgeCost;
    unordered_map<string, vector<edgeCost>> g;
    int maxValue = 2147483647;

public:
    addEdge(string source, string destination, int weight)
    {
        g[source].push_back({weight, destination});
    }
```

Graphs: Dijkstra Algorithm

Let's Implement the Solution now.

```
main()
{
    Graph g;

    g.addEdge("Res", "A", 2);
    g.addEdge("Res", "B", 4);

    g.addEdge("A", "C", 10);

    g.addEdge("B", "C", 4);
    g.addEdge("B", "D", 3);

    g.addEdge("D", "C", 5);

    cout << g.dijkstraAlgorithm("Res", "C");
}
```

```cpp
int dijkstraAlgorithm(string source, string destination){
        unordered_map<string, bool> visited;
        priority_queue<edgeCost, vector<edgeCost>, greater<edgeCost>> pq;
        unordered_map<string, int> costs;
        initializeCosts(costs, source);

        pq.push({costs[source], source});
        while(!pq.empty())
        {
            string current = pq.top().second;
            pq.pop();
            visited[current] = true;
            for(auto edge: g[current])
            {
                if(visited.find(edge.second) == visited.end())
                {
                    if(costs[current] + edge.first < costs[edge.second])
                    {
                        costs[edge.second] = costs[current] + edge.first;
                        pq.push({edge.first, edge.second});
                    }
                }
            }
        }
        return costs[destination];
    }
```

Graphs: Dijkstra Algorithm

```
void initializeCosts(unordered_map<string, int> &costs, string source)
    {
        for (auto vertex : g)
        {
            if(vertex.first == source)
                costs[vertex.first] = 0;
            else
                costs[vertex.first] = maxValue;
            for (auto edge : vertex.second)
            {
                if(edge.second == source)
                    costs[edge.second] = 0;
                else if (costs.find(edge.second) == costs.end())
                    costs[edge.second] = maxValue;
            }
        }
    }
```

Dijkstra Algorithm: Implementation

What is the Time Complexity of Dijkstra Algorithm?

Dijkstra Algorithm: Implementation

- We are traversing the complete graph.
  $O(|V+E|)$ is the time complexity to traverse the graph.

- We are maintaining the min heap for finding the vertex with minimum cost.
  Height of the min heap would be $\log(V)$.

- Therefore, time complexity is $O(|V+E| * \log(|V|))$.

Dijkstra Algorithm: Implementation

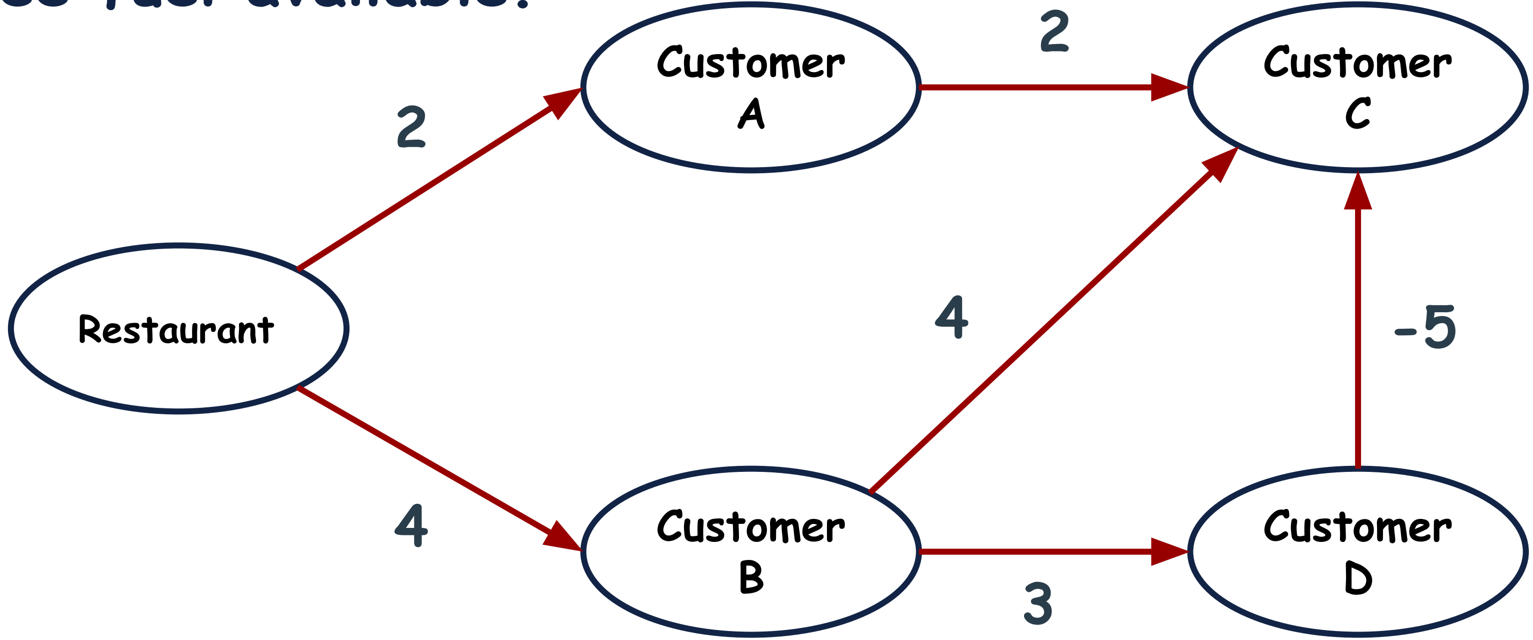
Time Complexity is $O(|E| * \log(|V|))$

Dijkstra Algorithm: Implementation

Single Source Shortest Path	Time Complexity	Space Complexity
	Worst Case	Worst Case
Dijkstra Algorithm	$O(\|E\| * \log(\|V\|))$	$O(\|E\| + \|V\|)$

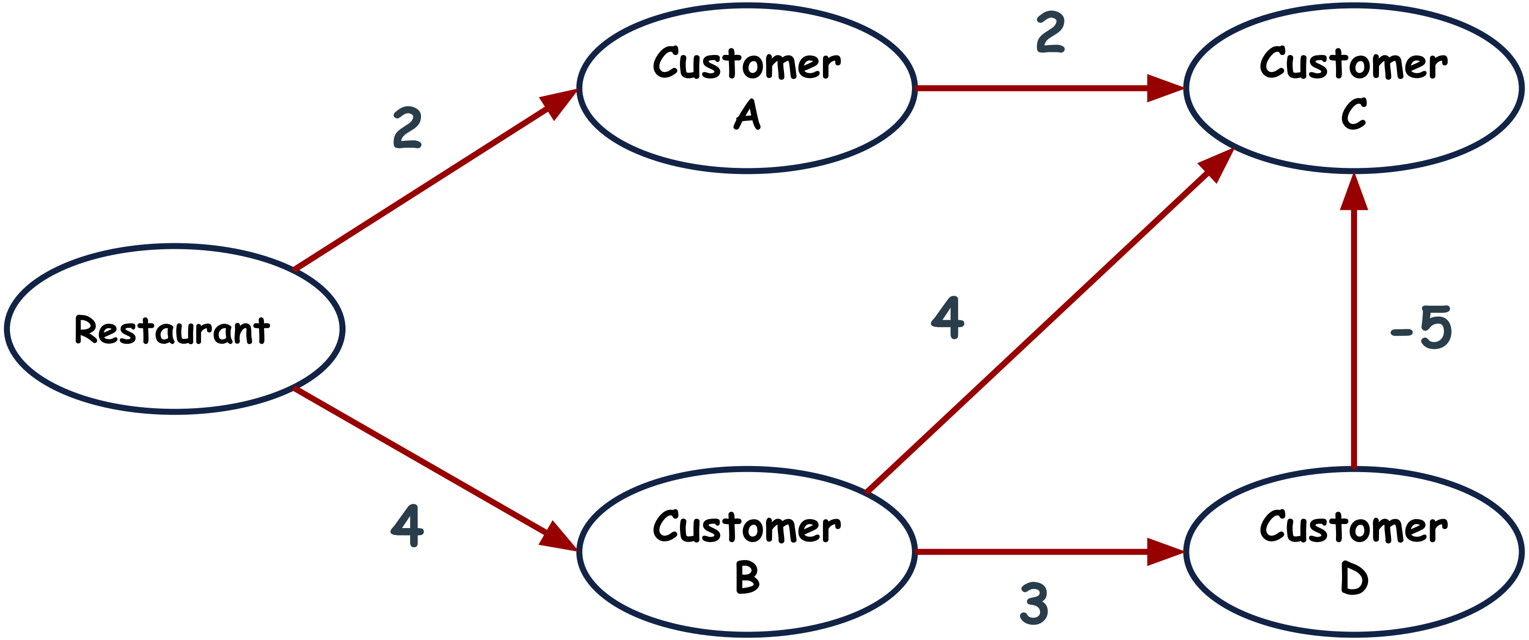
Graphs: Dijkstra Algorithm

Suppose that in the path from Customer D to C there is free fuel available.



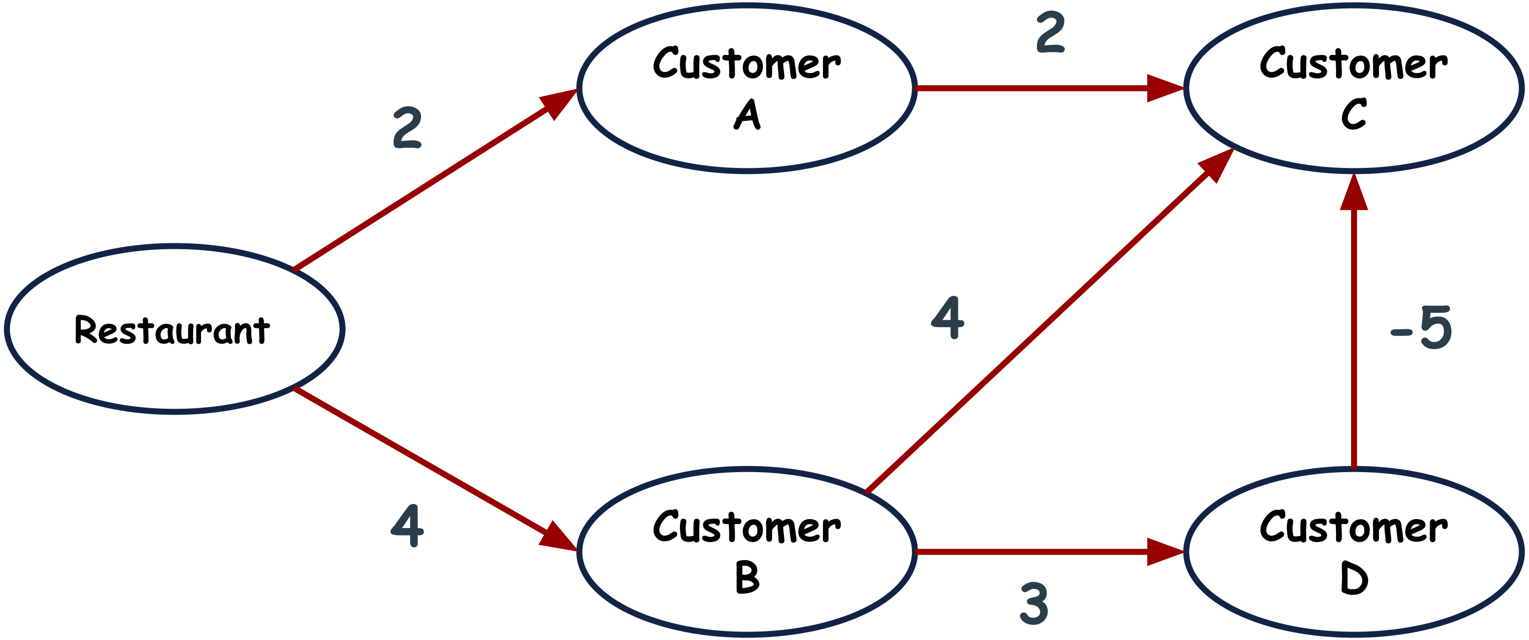
Graphs: Dijkstra Algorithm

Positive Weight is replaced with negative weight.



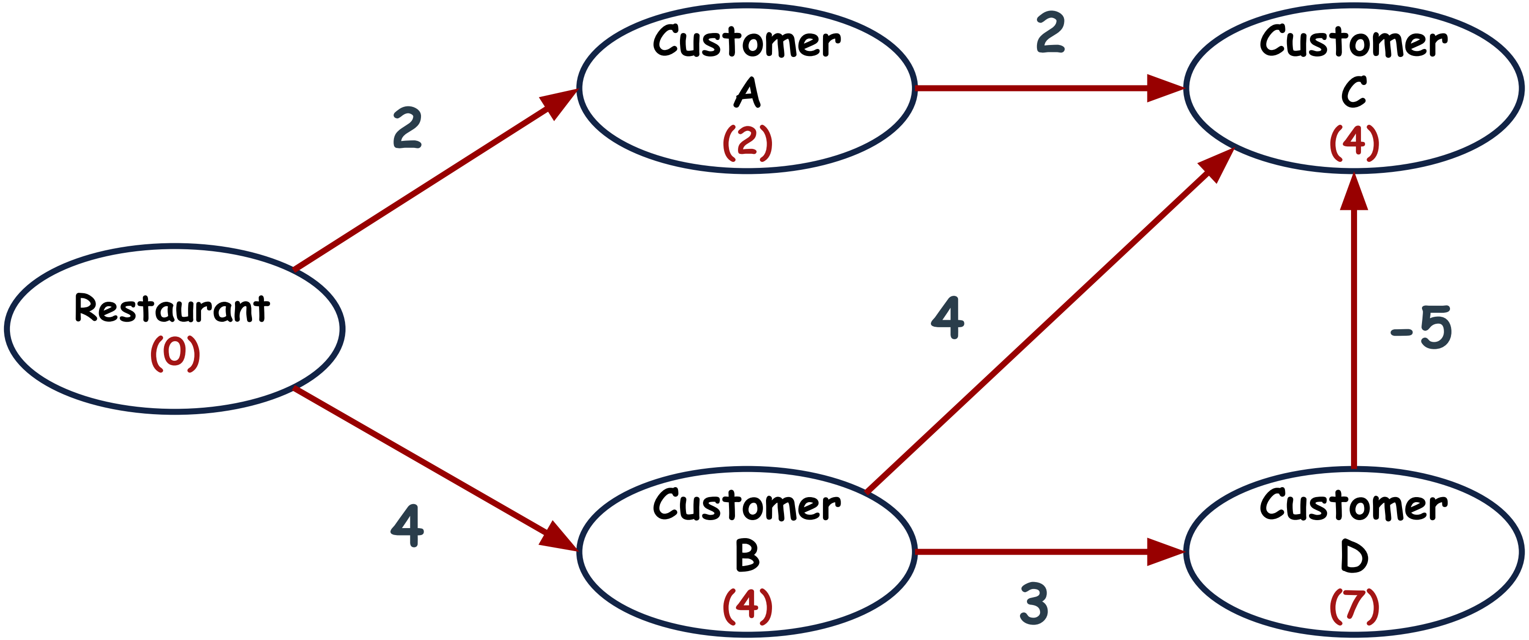
Graphs: Dijkstra Algorithm

Now, what are the costs for each vertex?



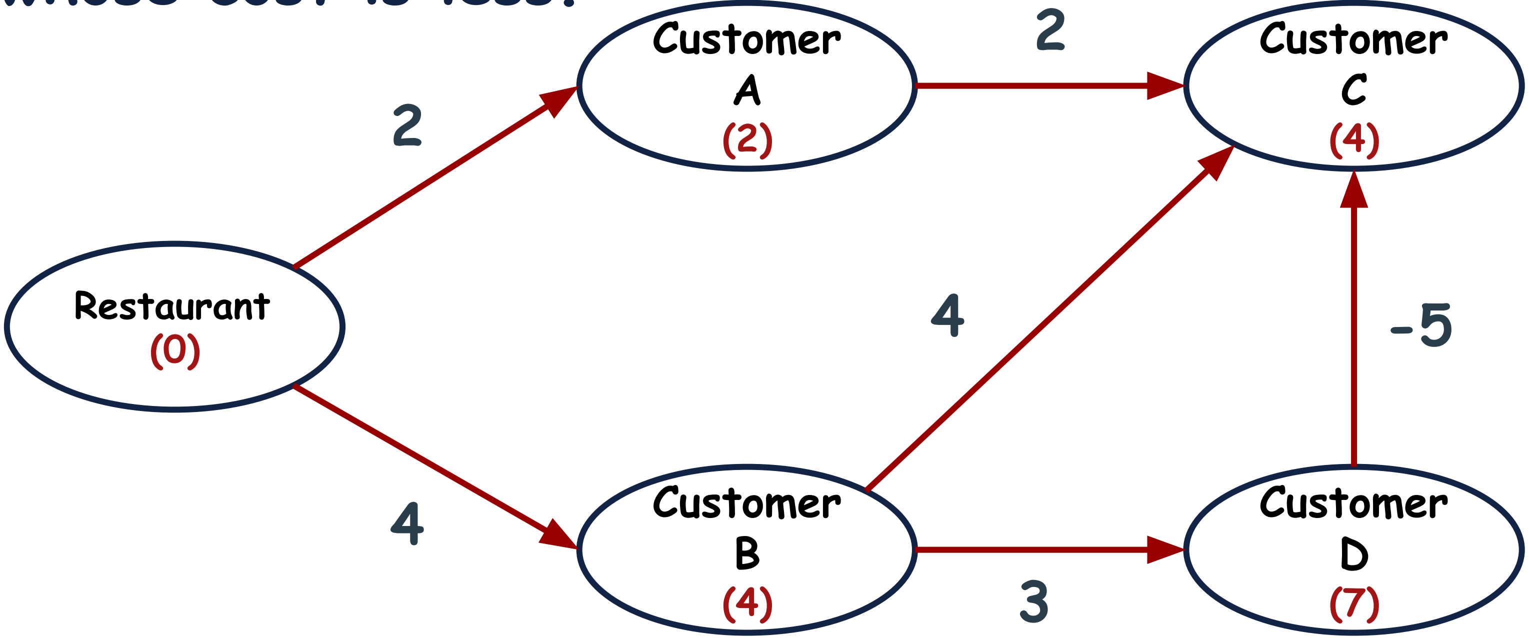
Graphs: Dijkstra Algorithm

Now, what are the costs for each vertex?



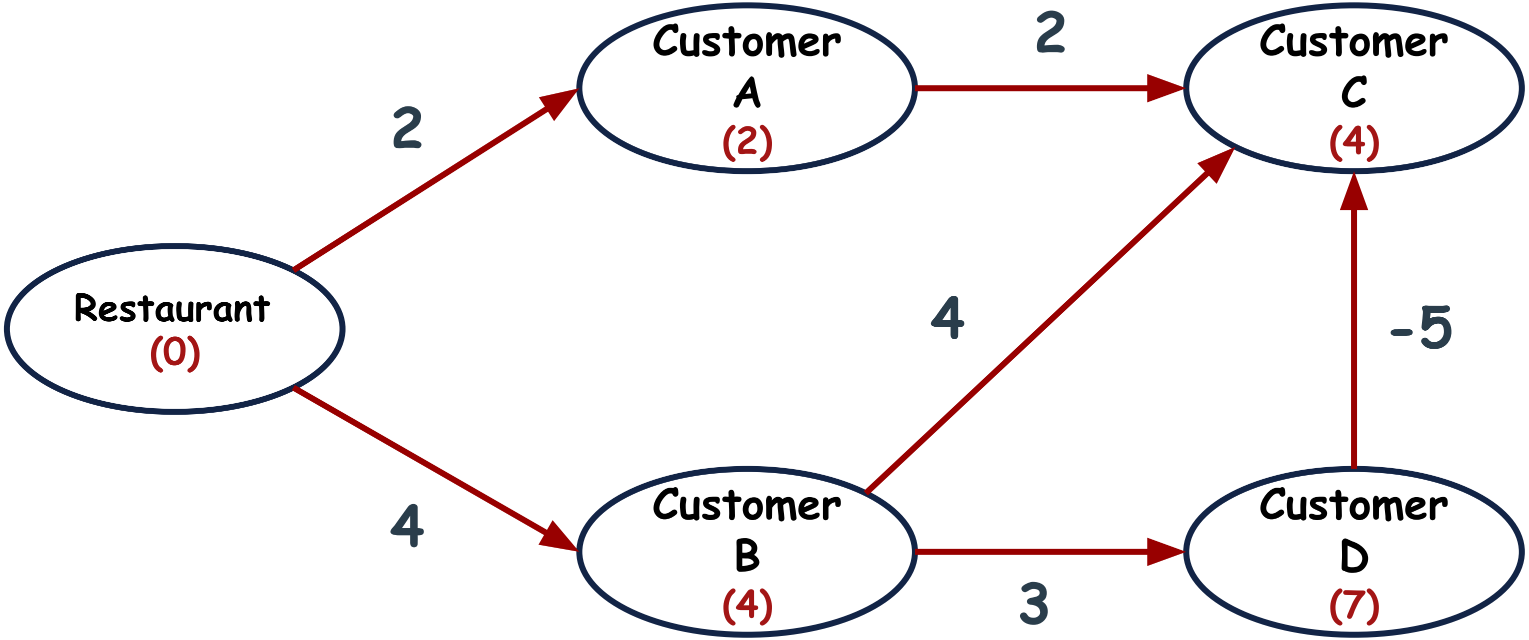
Graphs: Dijkstra Algorithm

Although there is a path from Customer D to Customer C whose cost is less.



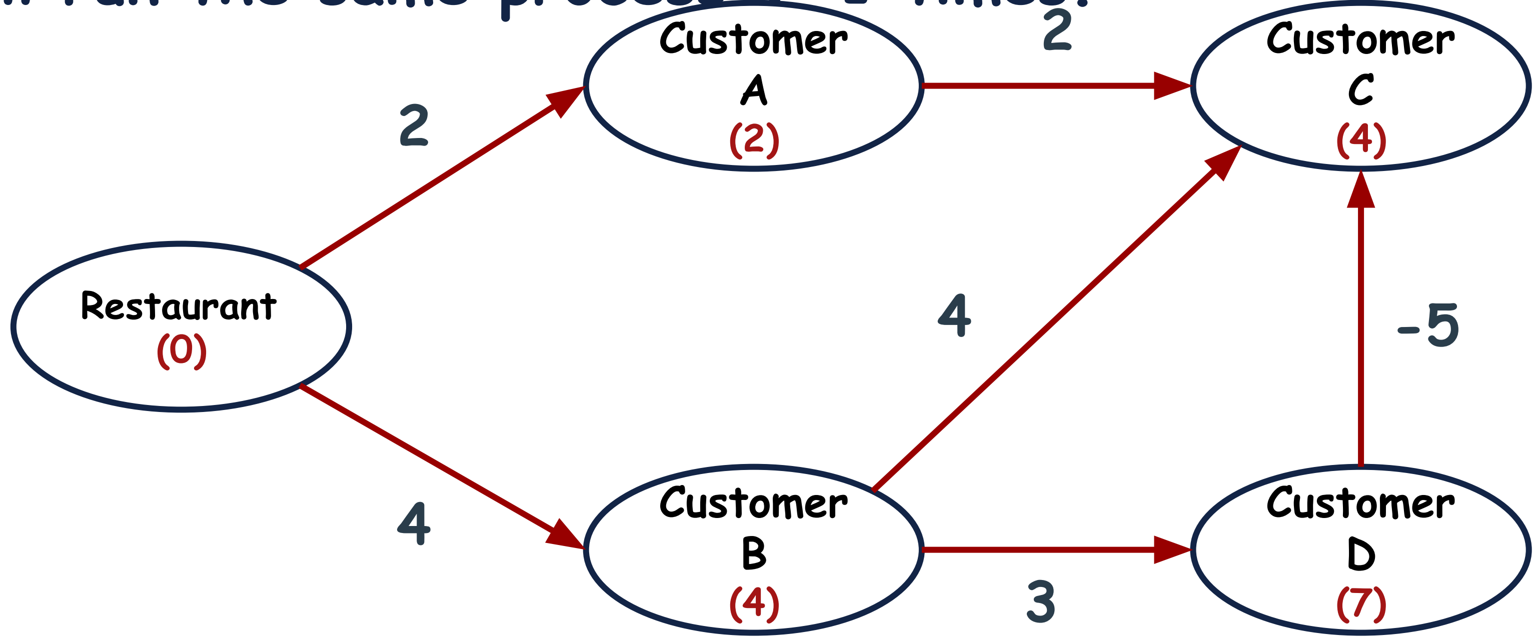
Dijkstra Algorithm: Negative Weight issue

Now, how to resolve this issue?



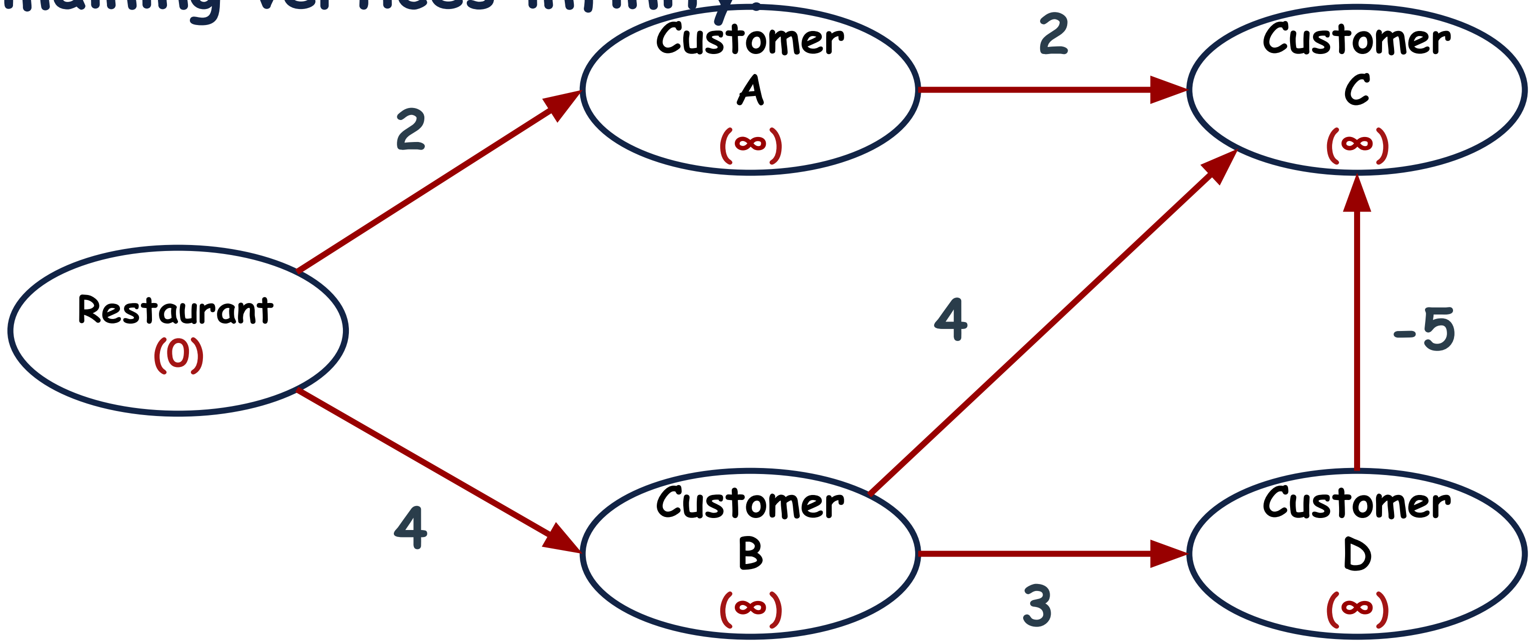
Dijkstra Algorithm: Negative Weight issue

Now, instead of just iterating the solution one Time. We will run the same process V-1 times.



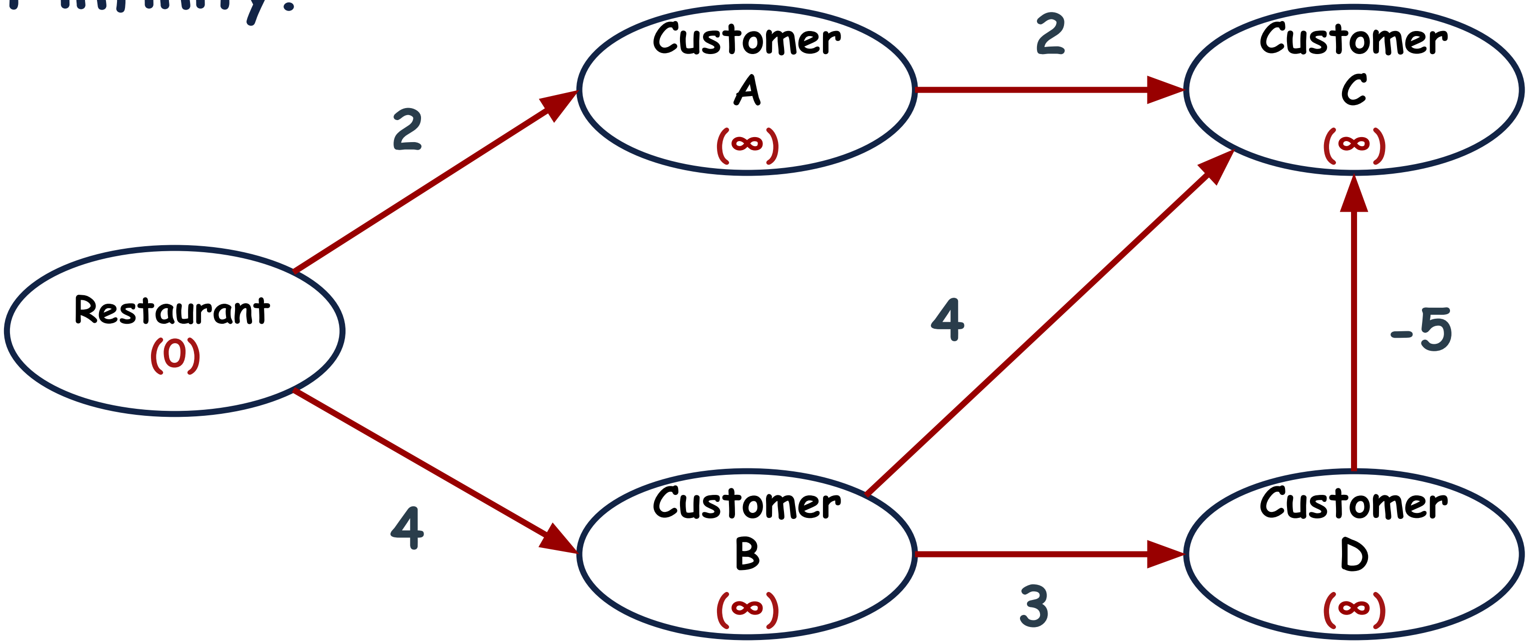
Dijkstra Algorithm: Negative Weight issue

Make the cost of the starting vertex 0 and all the remaining vertices infinity.



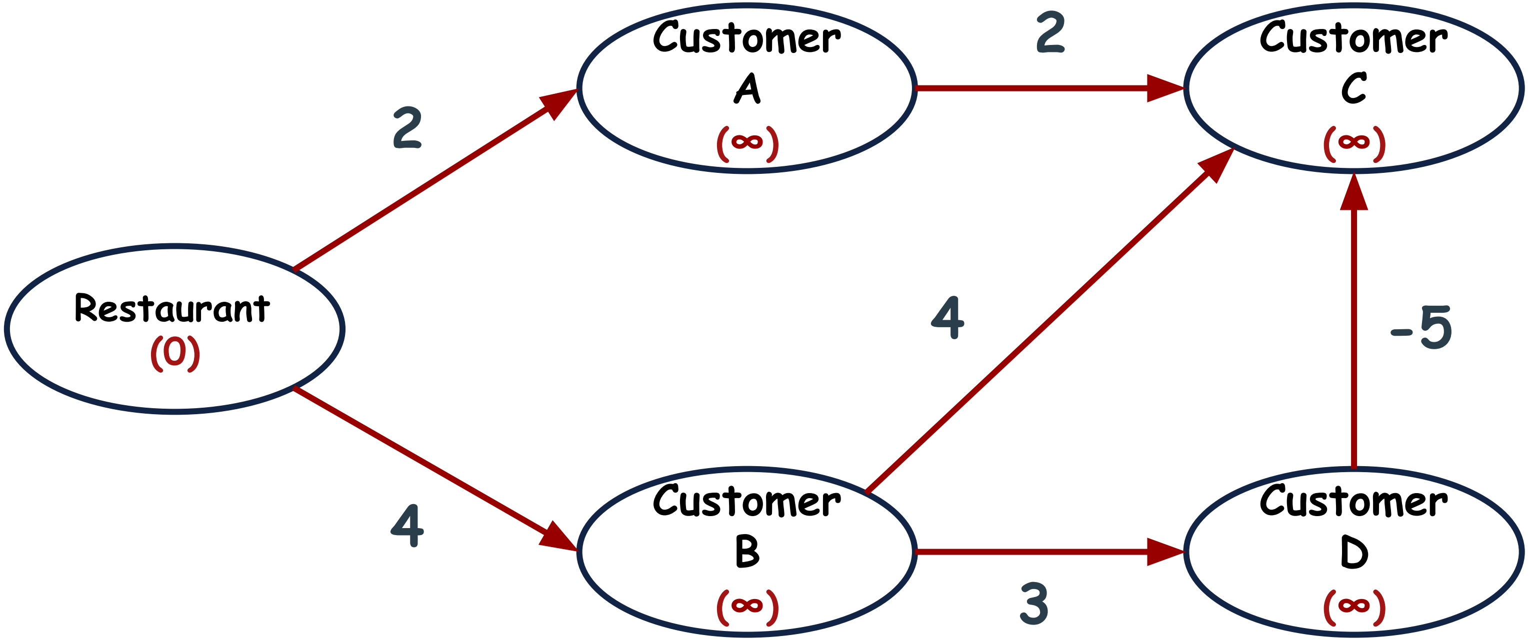
Dijkstra Algorithm: Negative Weight issue

Relax the adjacent vertices of the vertex whose cost is not infinity.



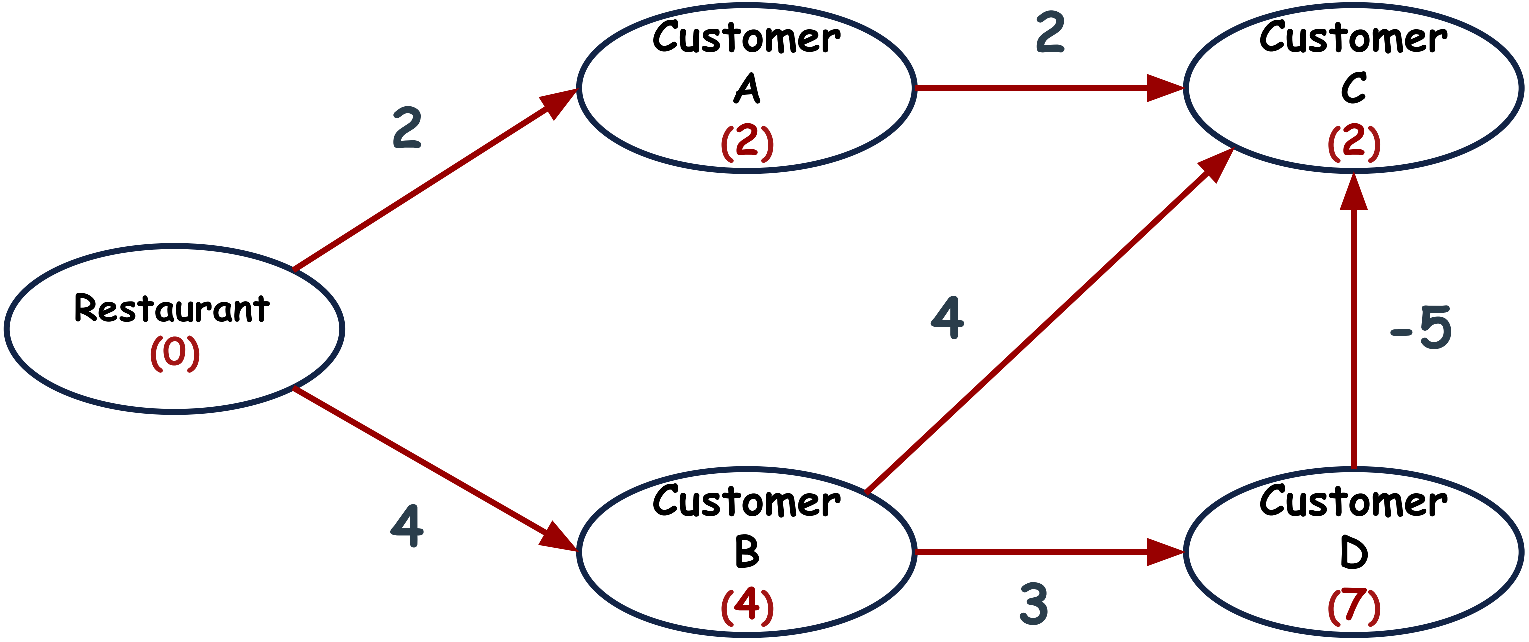
Dijkstra Algorithm: Negative Weight issue

You have to relax each edge V-1 times.



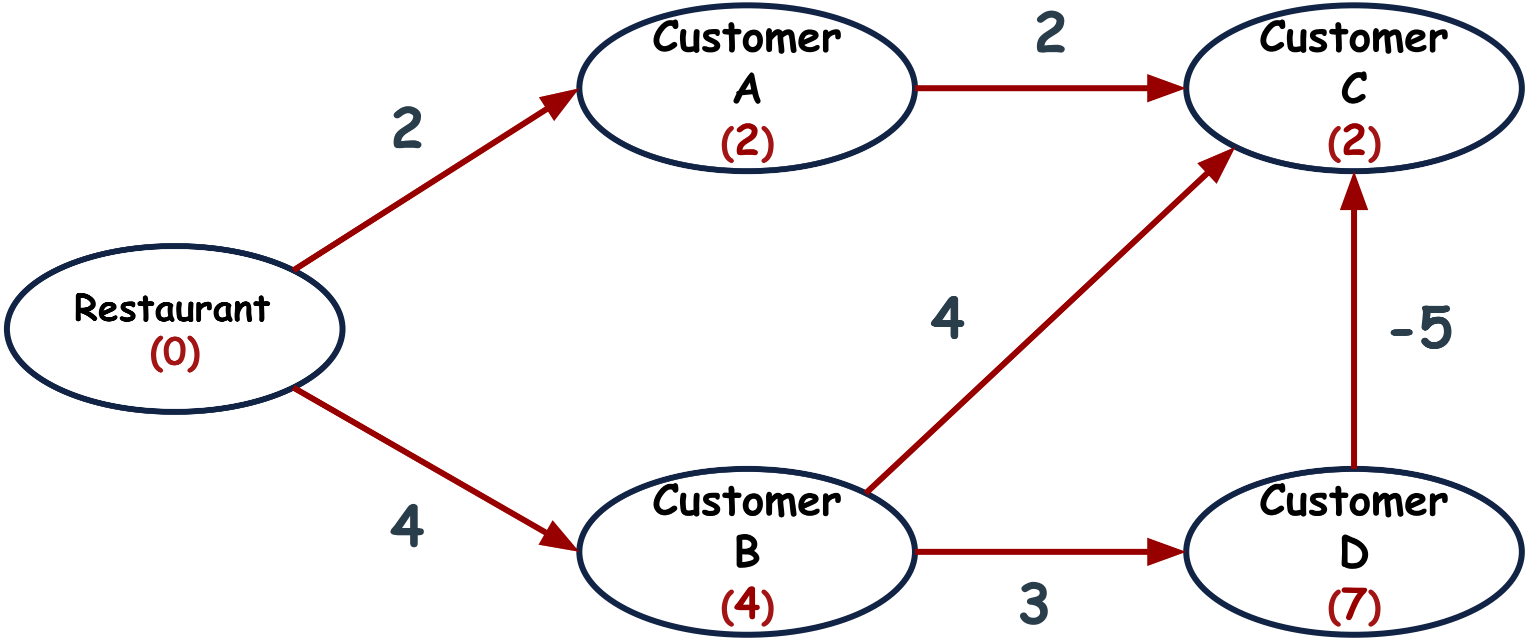
Dijkstra Algorithm: Negative Weight issue

In the end, the cost for each vertex will become minimum.



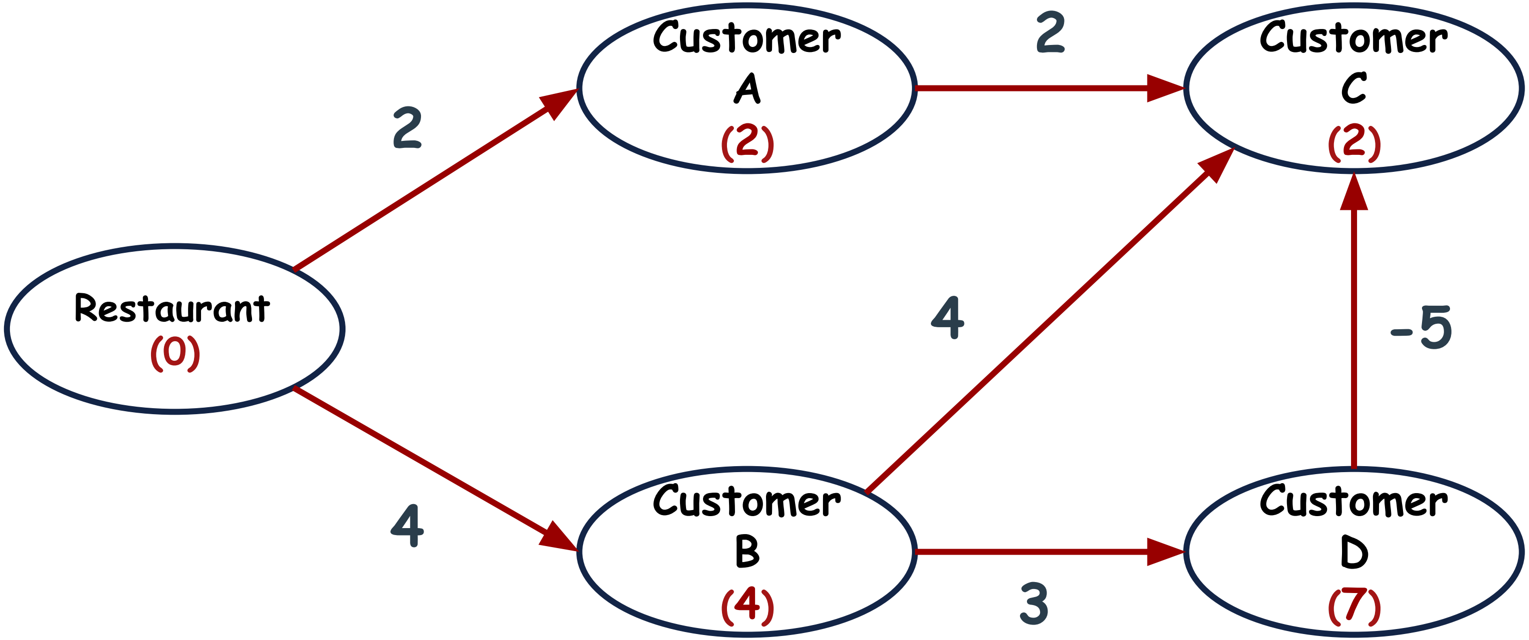
Bellman-Ford Algorithm

This Algorithm is called as Bellman-Ford Algorithm.



Bellman-Ford Algorithm

Lets implement the solution.



Bellman-Ford Algorithm

Lets implement the solution.

```
class Graph
{
    typedef pair<int, string> edgeCost;
    unordered_map<string, vector<edgeCost>> g;
    int maxValue = 2147483647;

public:
    addEdge(string source, string destination, int weight)
    {
        g[source].push_back({weight, destination});
    }
```

Bellman-Ford Algorithm

Lets implement the solution.

```
main()
{
    Graph g;

    g.addEdge("Res", "A", 2);
    g.addEdge("Res", "B", 4);

    g.addEdge("A", "C", 10);

    g.addEdge("B", "C", 4);
    g.addEdge("B", "D", 3);

    g.addEdge("D", "C", 5);

    cout << g.bellmanFord("Res", "C");
}
```

Bellman-Ford Algorithm

```
int bellmanFord(string source, string destination){
        unordered_map<string, int> costs;
        initializeCosts(costs, source);
        for (int x = 0; x < costs.size() - 1; x++)
        {
            for (auto vertex : g)
            {
                for (auto edge : vertex.second)
                {
                    if (costs[vertex.first] != maxValue && costs[vertex.first] + edge.first <
costs[edge.second])
                        costs[edge.second] = costs[vertex.first] + edge.first;
                }
            }
        }
        return costs[destination];
    }
```

Bellman-Ford Algorithm

```
void initializeCosts(unordered_map<string, int> &costs, string source)
    {
        for (auto vertex : g)
        {
            if(vertex.first == source)
                costs[vertex.first] = 0;
            else
                costs[vertex.first] = maxValue;
            for (auto edge : vertex.second)
            {
                if(edge.second == source)
                    costs[edge.second] = 0;
                else if (costs.find(edge.second) == costs.end())
                    costs[edge.second] = maxValue;
            }
        }
    }
```

Bellman-Ford: Implementation

What is the Time Complexity of Bellman-Ford Algorithm?

Bellman-Ford: Implementation

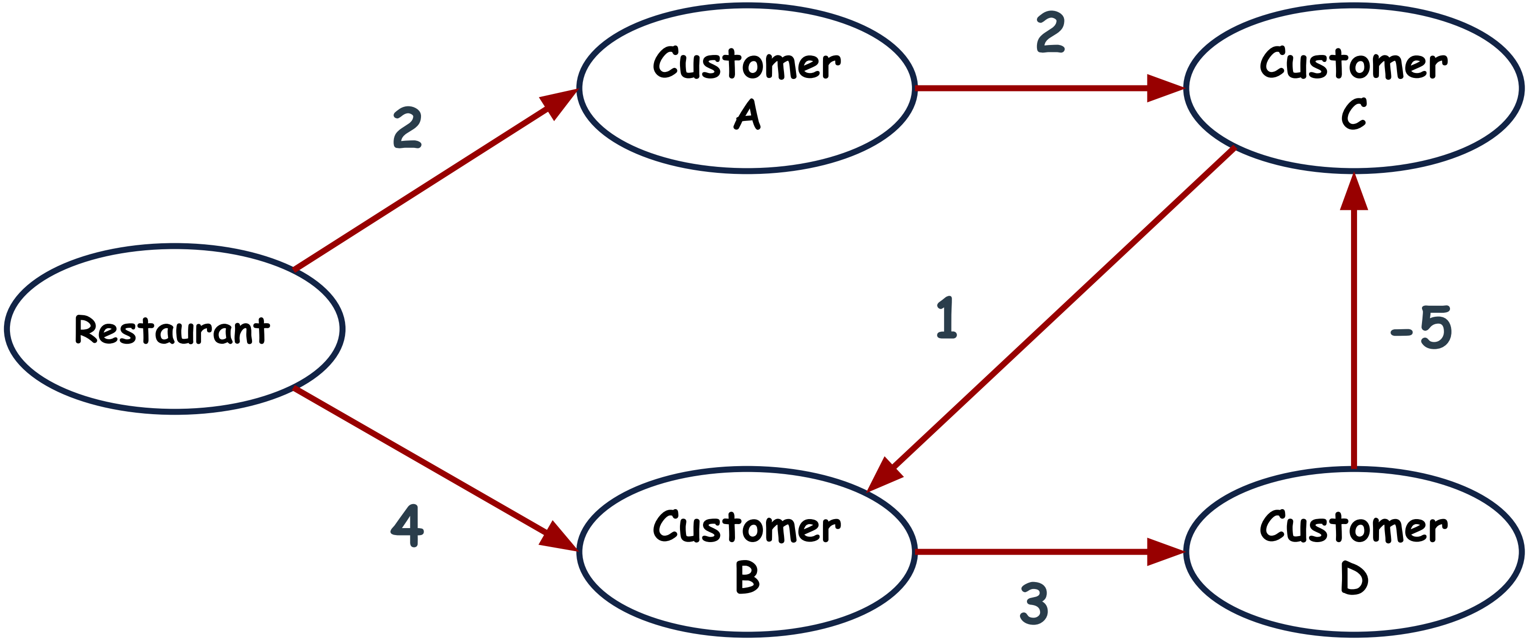
Time Complexity is $O(|V| * |E|)$

Single Source Shortest Path Algorithms

Single Source Shortest Path	Time Complexity	Space Complexity
	Worst Case	Worst Case
Dijkstra Algorithm	$O(\|E\| * \log(\|V\|))$	$O(\|E\| + \|V\|)$
Bellman-Ford Algorithm	$O(\|V\| * \|E\|)$	$O(\|V\|)$

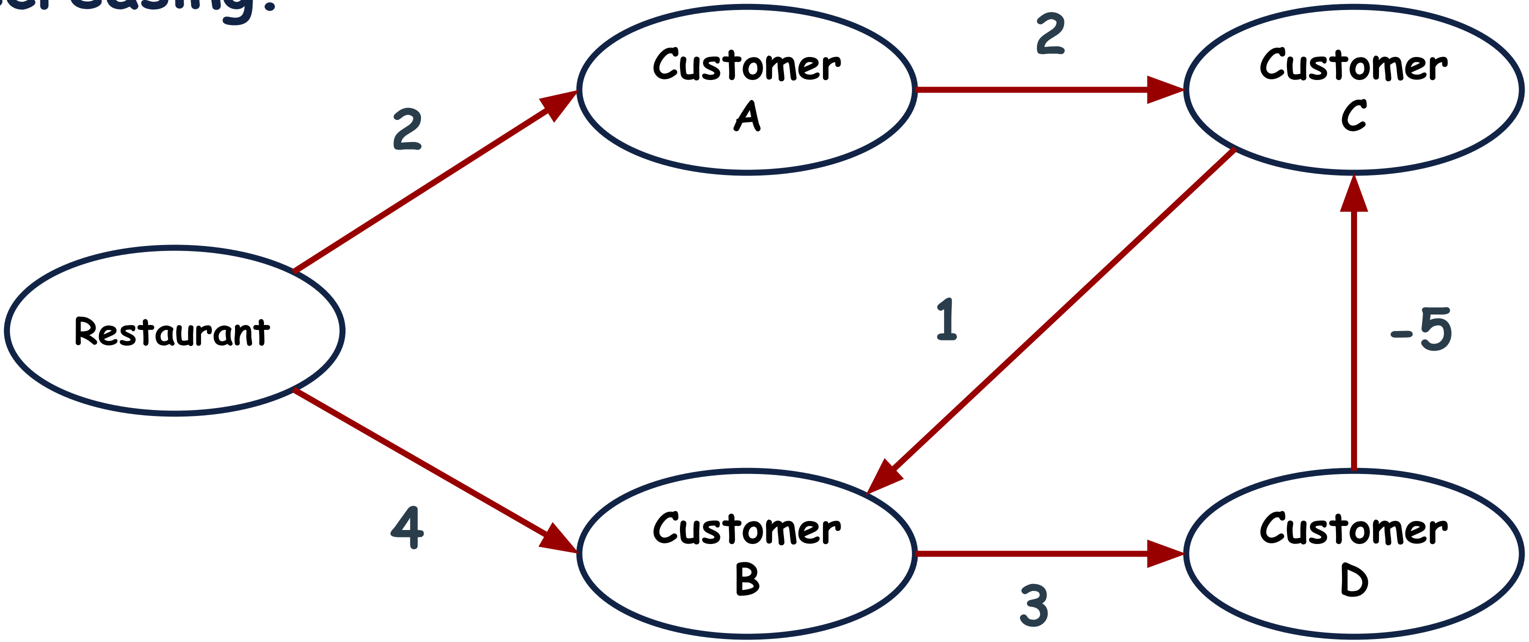
Graphs: Bellman-Ford Algorithm

Now, what if there is a cycle with negative weight?



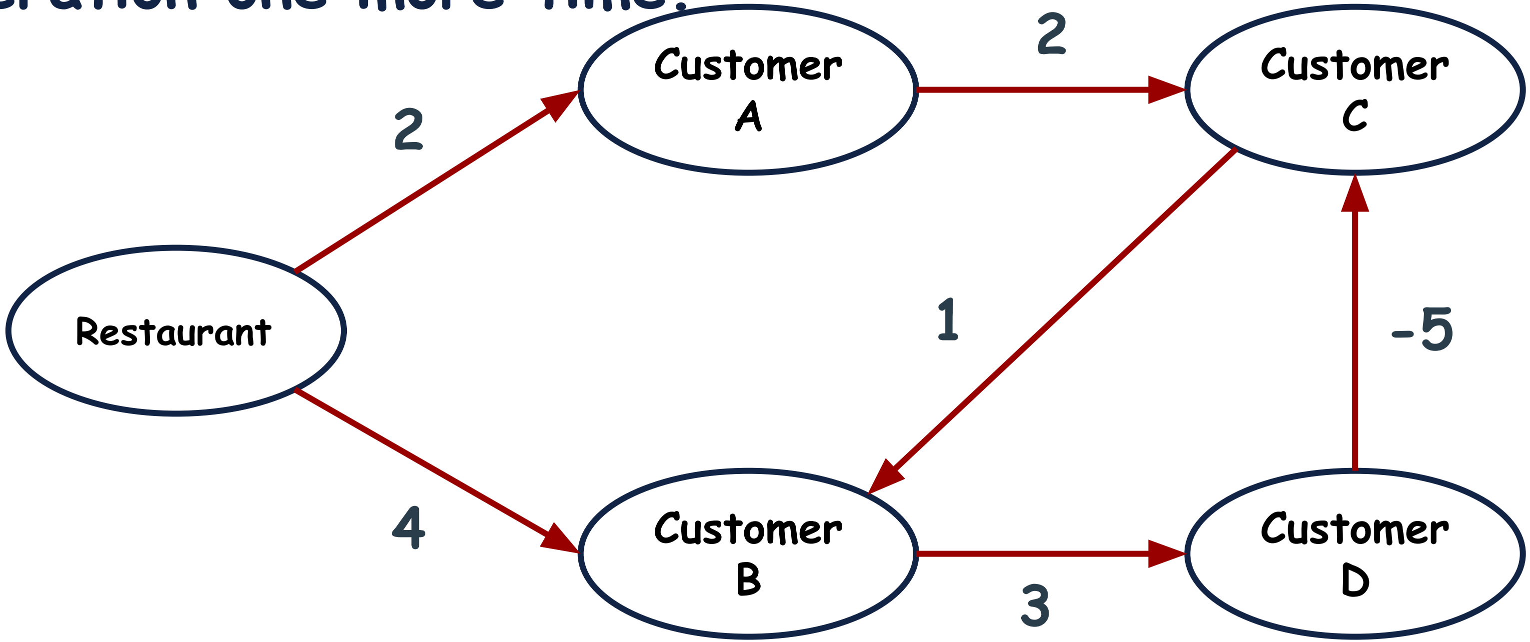
Graphs: Bellman-Ford Algorithm

Then the cost of Customer B, C and D will keep on decreasing.



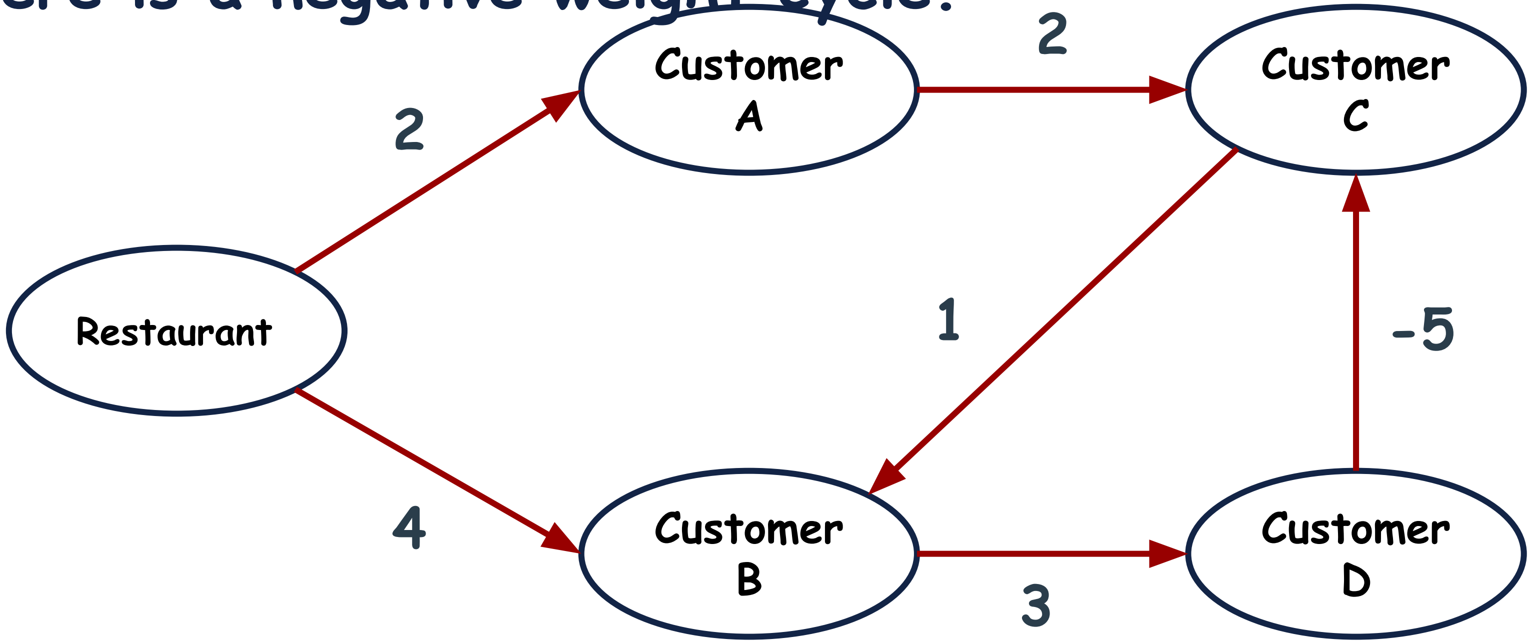
Graphs: Bellman-Ford Algorithm

We can detect the negative weight cycle, by running the iteration one more time.



Graphs: Bellman-Ford Algorithm

If the cost of any vertex is updated then it means there is a negative weight cycle.



```
int bellmanFord(string source, string destination){
        unordered_map<string, int> costs;
        initializeCosts(costs, source);
        for (int x = 0; x < costs.size() - 1; x++){
            for (auto vertex : g)
            {
                for (auto edge : vertex.second)
                {
                    if (costs[vertex.first] != maxValue && costs[vertex.first] + edge.first <
costs[edge.second])
                        costs[edge.second] = costs[vertex.first] + edge.first;
                }
            }
        }
        for (auto vertex : g){
            for (auto edge : vertex.second)
            {
                if (costs[vertex.first] != maxValue && costs[vertex.first] + edge.first <
costs[edge.second])
                {
                    costs[edge.second] = costs[vertex.first] + edge.first;
                    return 0;
                }
            }
        }
        return costs[destination];
    }
```

Learning Objective

Students should be able to find shortest path to solve real life problems.