Luiss
Libera Università Internazionale degli Studi Sociali Guido Carli

Algorithms A.Y. 2022/2023

Lab – Graphs and Shortest Path

Irene Finocchi, Flavio Giorgi, Bardh Prenkaj
finocchi@luiss.it, fgiorgi@luiss.it, bprenkaj@luiss.it©

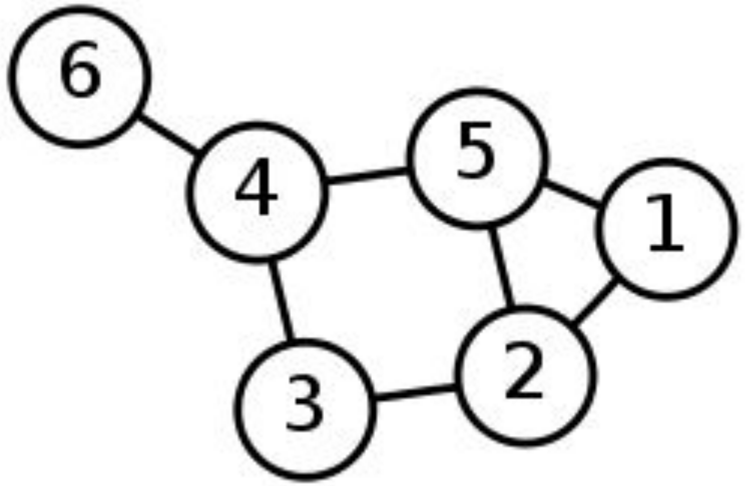
19 April 2023

LUISS

Dipartimento di Impresa e Management

Graphs

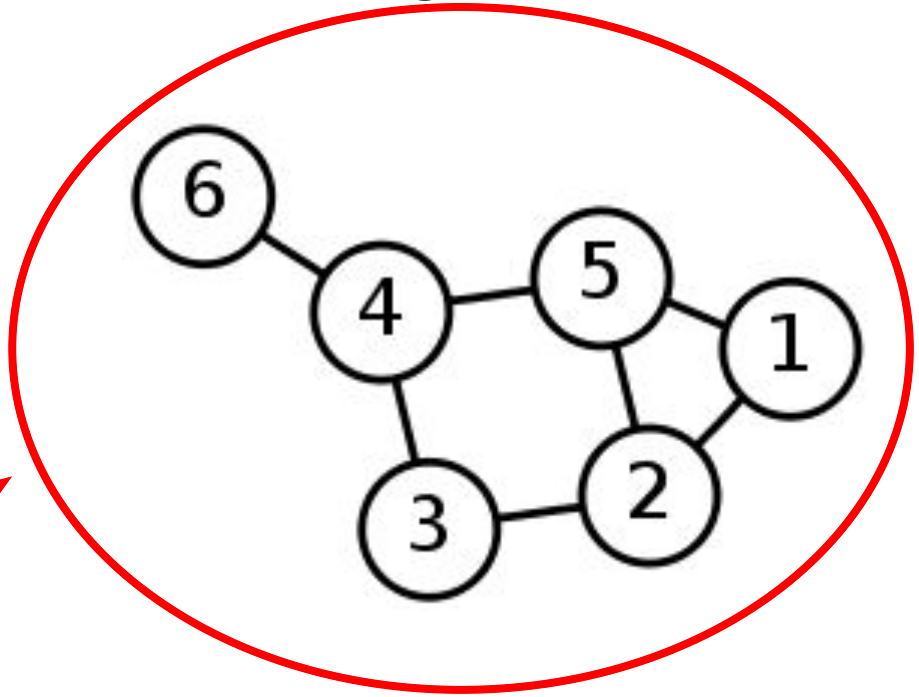
A **Graph** is a pair *G=(V, E)* where ***V*** is the set containing all the vertices, ***E*** instead is the set of all the edges.



Graphs

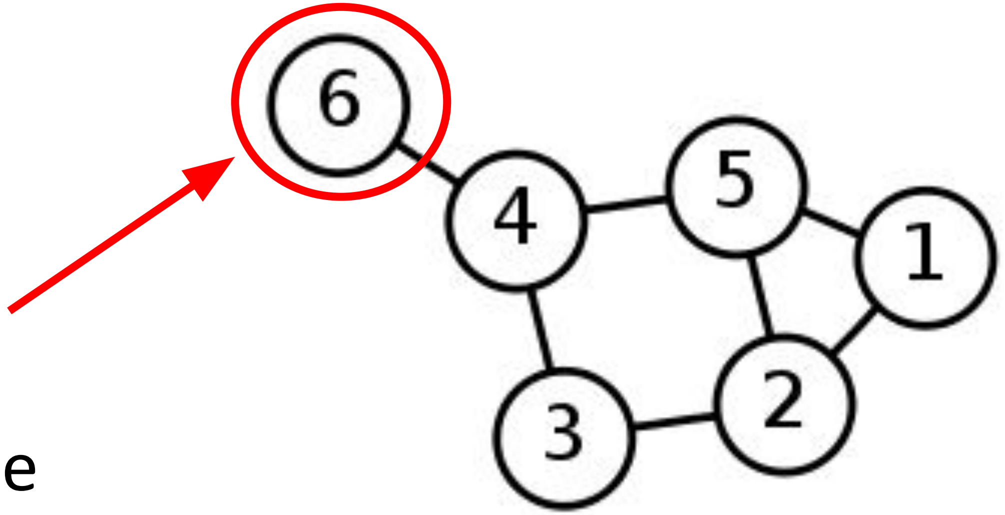
A **Graph** is a pair *G=(V, E)* where ***V*** is the set containing all the vertices, ***E*** instead is the set of all the edges.

Graph



Graphs

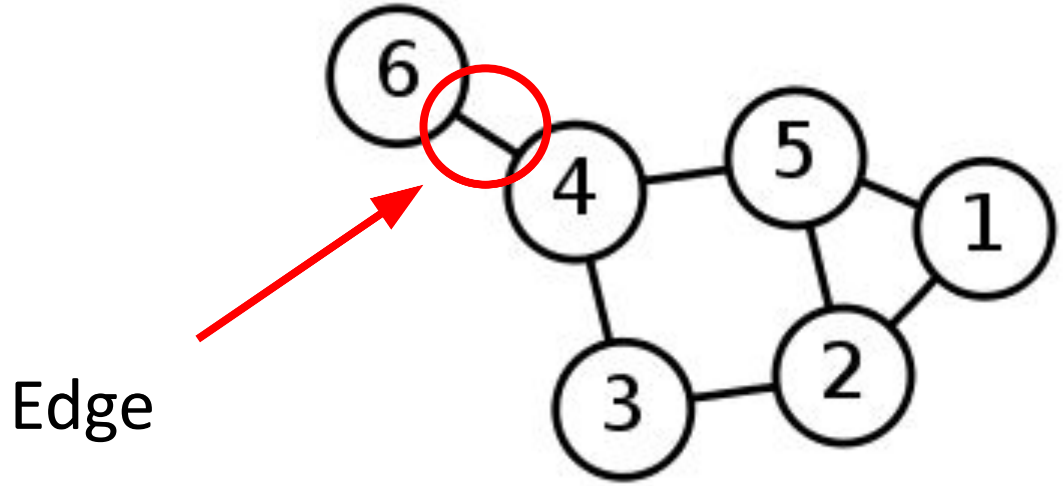
A **Graph** is a pair *G=(V, E)* where ***V*** is the set containing all the vertices, ***E*** instead is the set of all the edges.



Vertex
or node

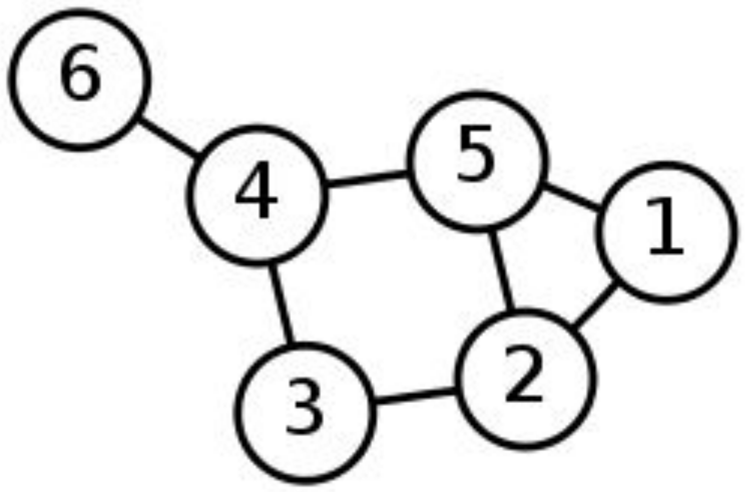
Graphs

A **Graph** is a pair *G=(V, E)* where ***V*** is the set containing all the vertices, ***E*** instead is the set of all the edges.



Graphs - Non Direct

A **Graph** is a pair *G=(V, E)* where ***V*** is the set containing all the vertices, ***E*** instead is the set of all the edges.



G =

V = ?

E = ?

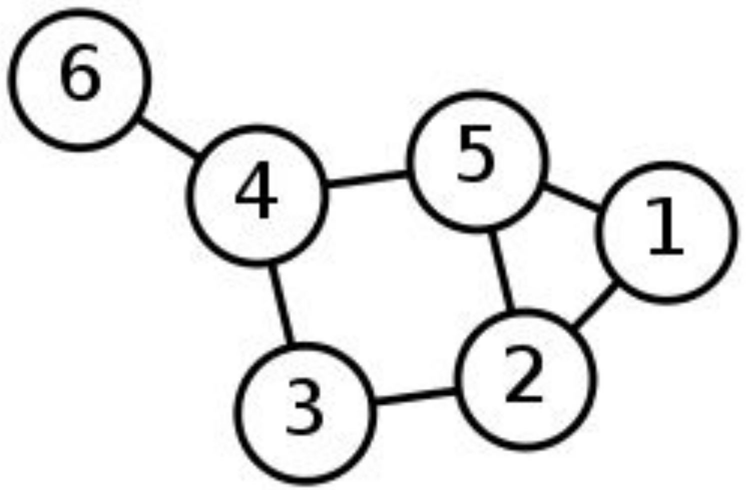
Graphs - Non Direct

A **Graph** is a pair *G=(V, E)* where ***V*** is the set containing all the vertices, ***E*** instead is the set of all the edges.

G =

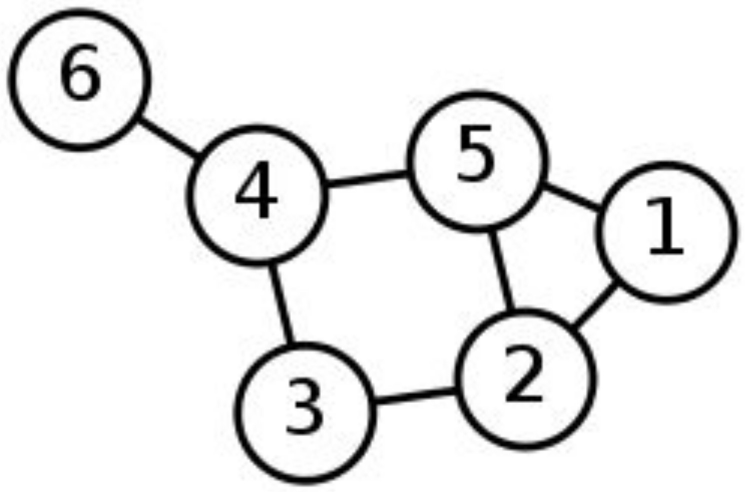
V = {6, 4, 5, 1, 2, 3}

E = ?



Graphs - Non Direct

A **Graph** is a pair *G=(V, E)* where ***V*** is the set containing all the vertices, ***E*** instead is the set of all the edges.



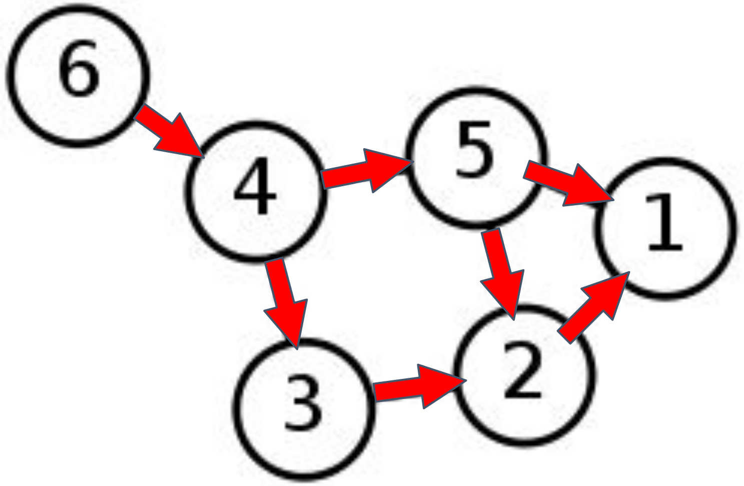
G =

V = {6, 4, 5, 1, 2, 3}

E = {(6, 4), (4, 5), (5, 1), (5, 2),
(2, 1), (4, 3), (3, 2)}

Graphs - Direct

A **Graph** is a pair *G=(V, E)* where ***V*** is the set containing all the vertices, ***E*** instead is the set of all the edges.



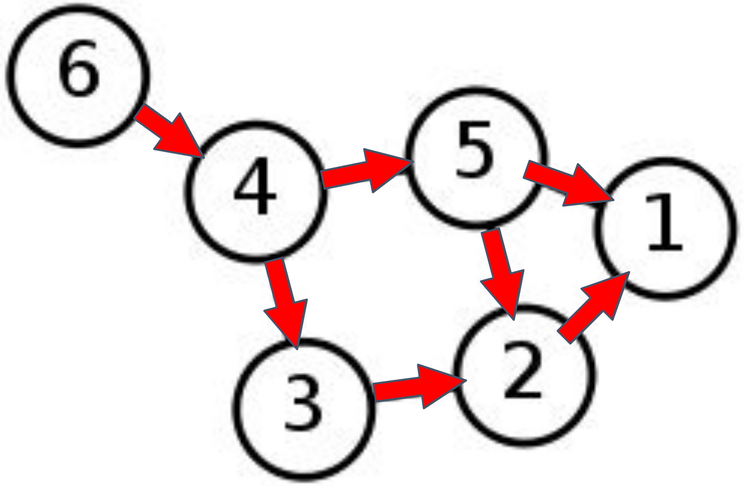
G =

V = ?

E = ?

Graphs - Direct

A **Graph** is a pair *G=(V, E)* where ***V*** is the set containing all the vertices, ***E*** instead is the set of all the edges.



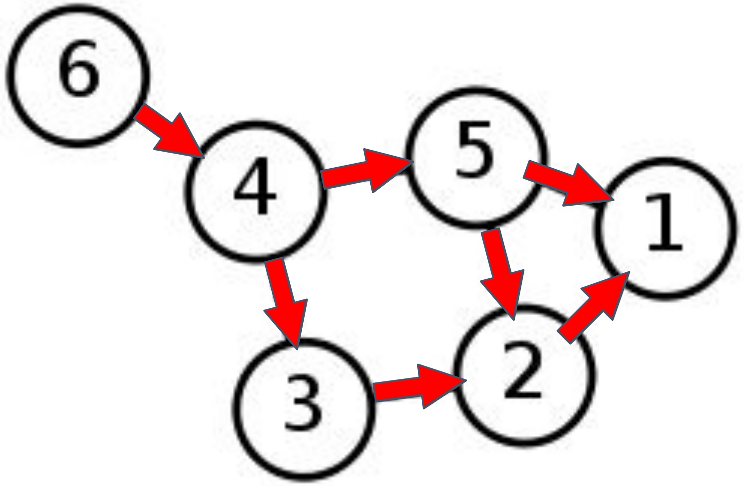
G =

V = {6, 4, 5, 1, 2, 3}

E = ?

Graphs - Direct

A **Graph** is a pair *G=(V, E)* where ***V*** is the set containing all the vertices, ***E*** instead is the set of all the edges.



G =

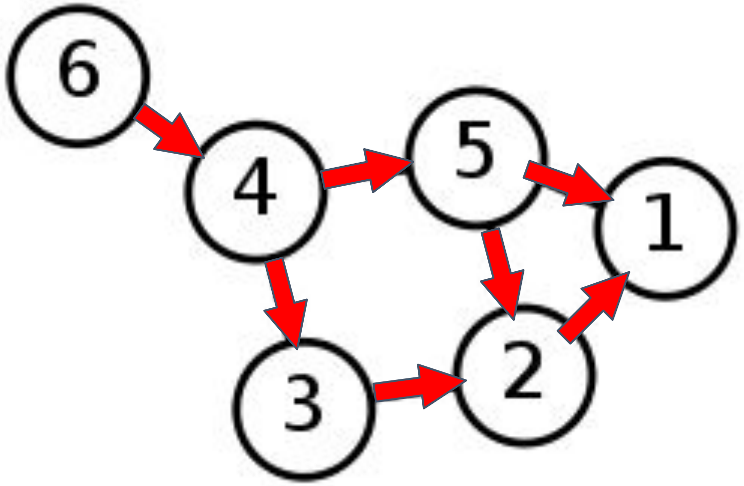
V = {6, 4, 5, 1, 2, 3}

E = {(6, 4), (4, 3), (4, 5), (3, 2), (5, 2), (2, 1), (5, 1)}

REMEMBER: if the graph is ***direct*** it means that for any node $u, v \in V$, *if* $(u, v) \in E$ ***it is possible that (v, u)*** $\notin E$

Graphs - Direct

A **Graph** is a pair *G=(V, E)* where ***V*** is the set containing all the vertices, ***E*** instead is the set of all the edges.



G =

V = {6, 4, 5, 1, 2, 3}

E = {(6, 4), (4, 3), (4, 5), (3, 2), (5, 2), (2, 1), (5, 1)}

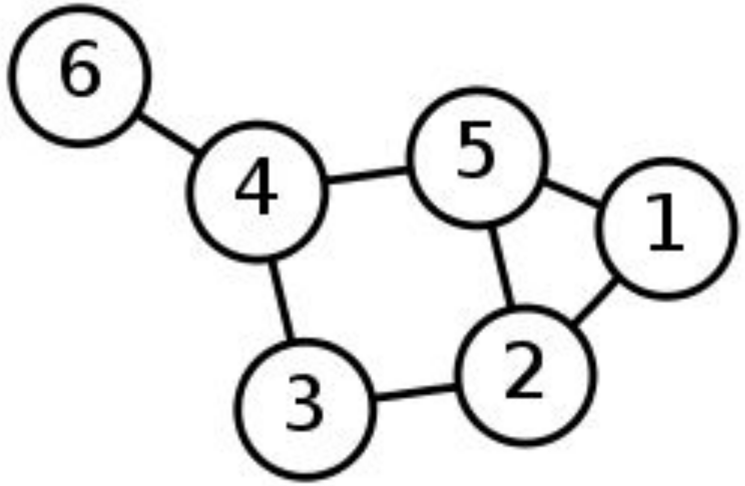
REMEMBER: Thus (4, 6) ≠ (6, 4)

Graphs - How to represent a graph

There are many ways to represent a graph:

Adjacency matrix

	1	2	3	4	5	6
1						
2						
3						
4						
5						
6						



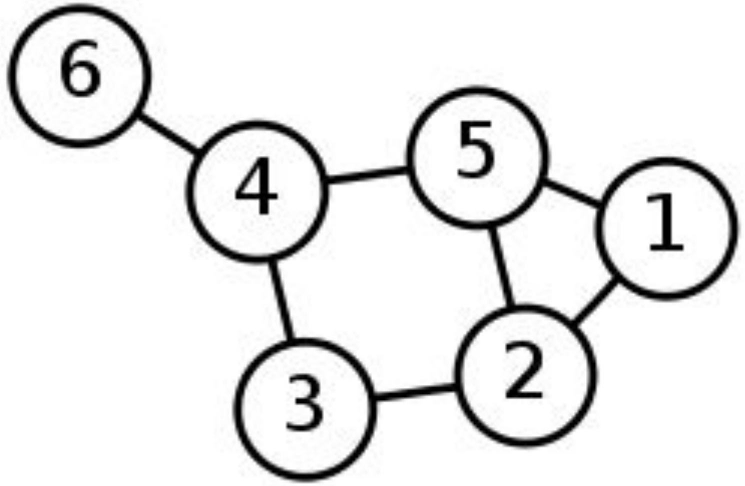
LUISS

Graphs - How to represent a graph

There are many ways to represent a graph:

Adjacency matrix

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



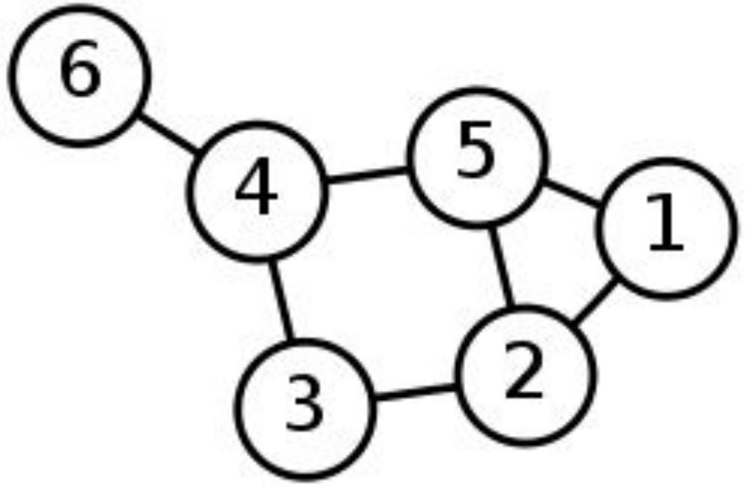
Graphs - How to represent a graph

There are many ways to represent a graph:

Adjacency matrix

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

?

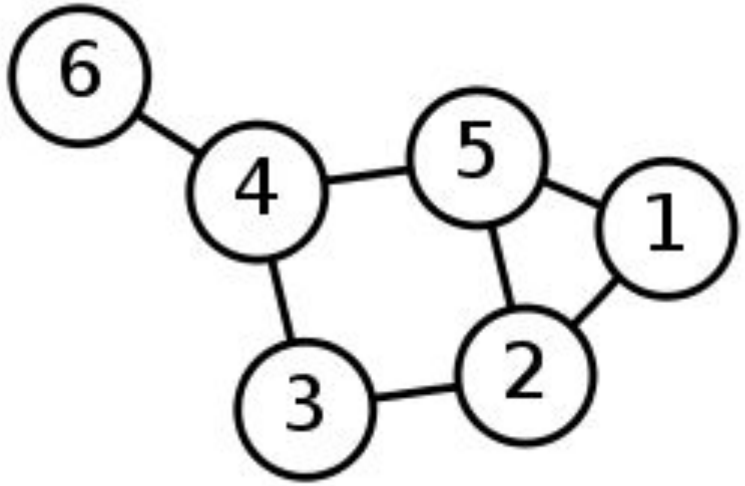


Graphs - How to represent a graph

There are many ways to represent a graph:

Adjacency matrix

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



The diagonal represents self-loops

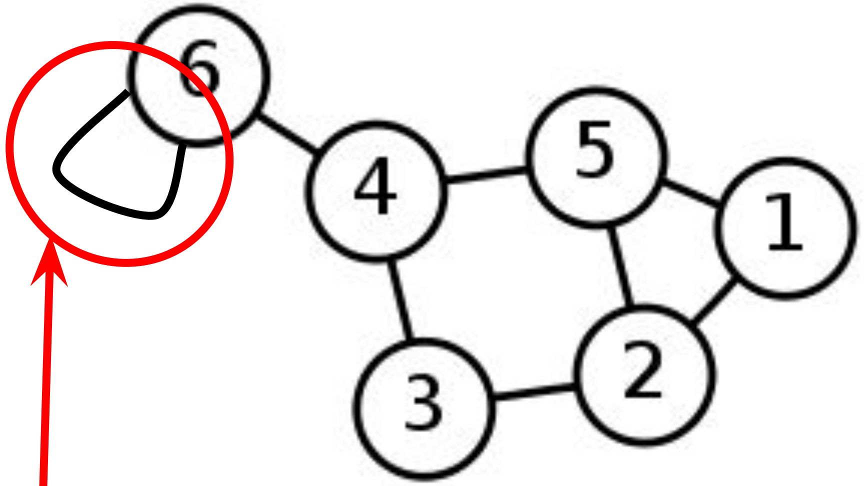
LUISS

Graphs - How to represent a graph

There are many ways to represent a graph:

Adjacency matrix

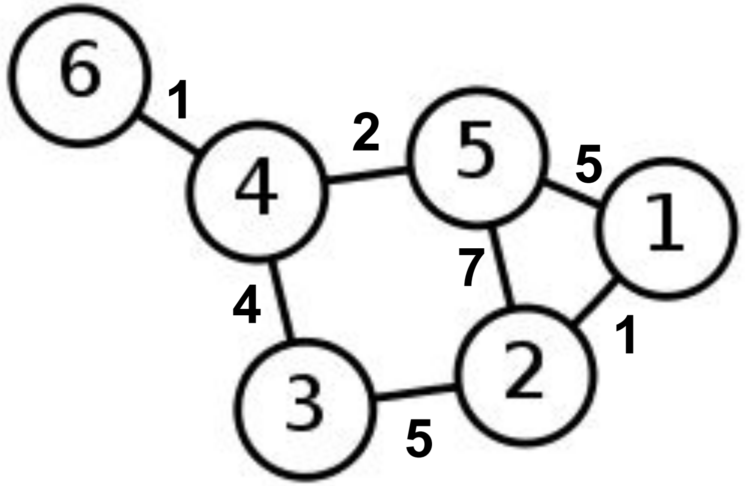
	1	2	3	4	5	6
1	0					
2	1	0				
3	0	1	0			
4	0	0	1	0		
5	1	1	0	1	0	
6	0	0	0	1	0	1



Self-loop

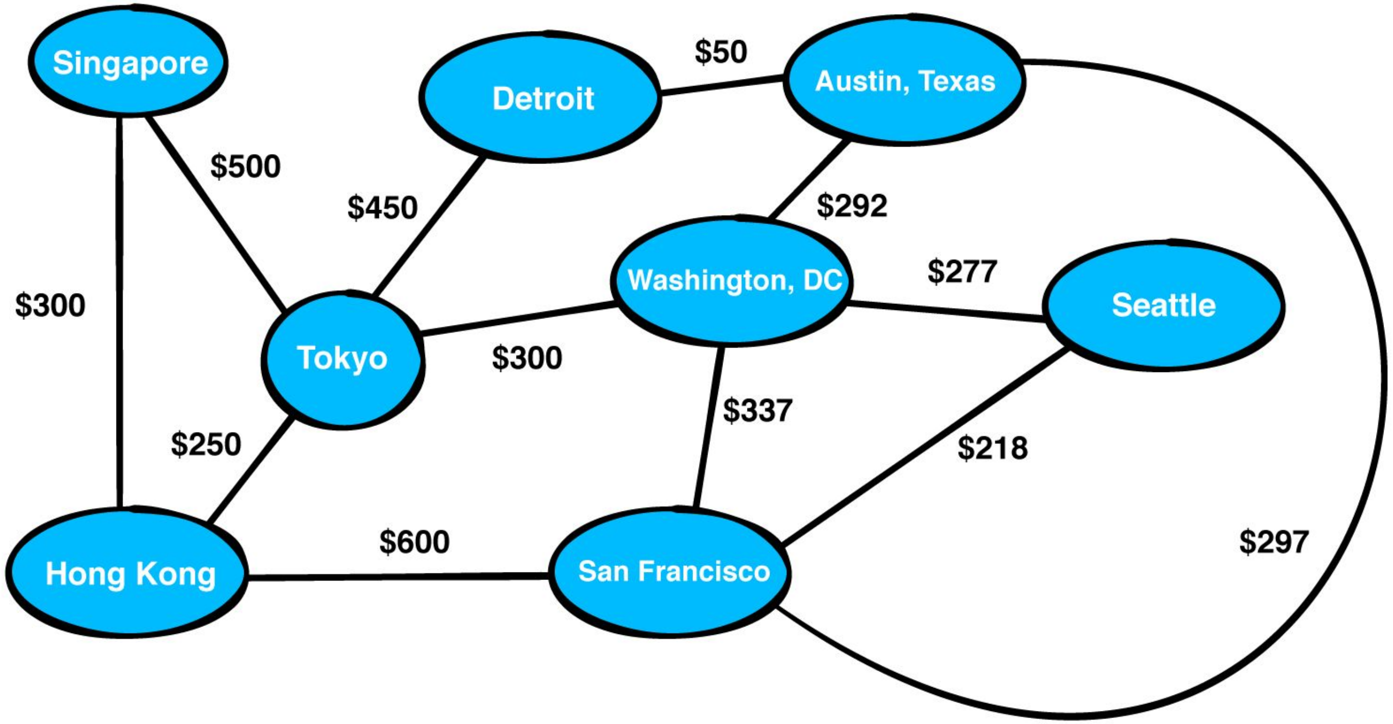
Graphs - Weights

Given an undirected graph G(V, E) we can add weights on the edges



Graphs - Path

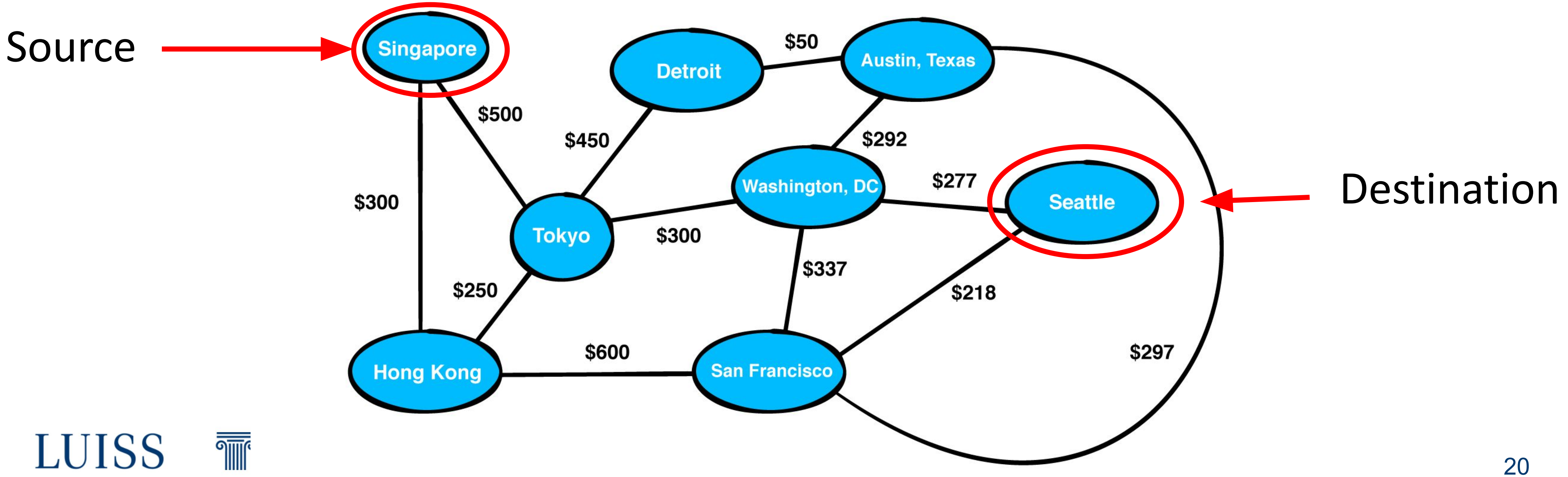
Let's suppose that we have multiple destination and that we want to know which are the paths **from a source location** toward a **destination** that **cost the least amount of money.**



LUISS

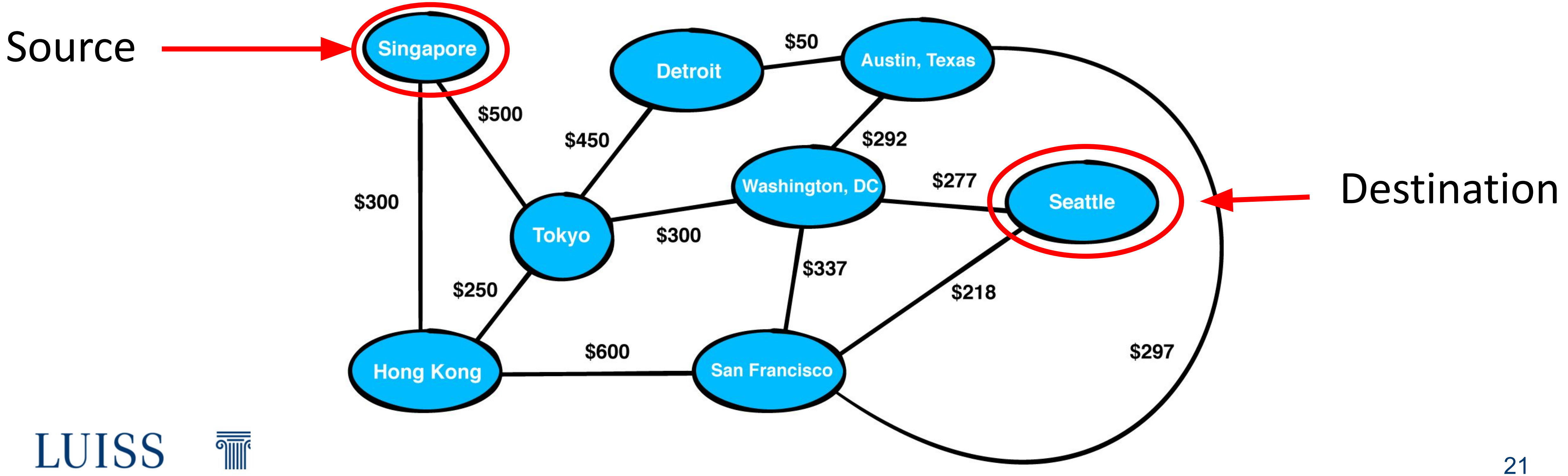
Graphs - Path

Let's suppose that we have multiple destination and that we want to know which are the paths **from a source location** toward a **destination** that **cost the least amount of money.**



Graphs - Path

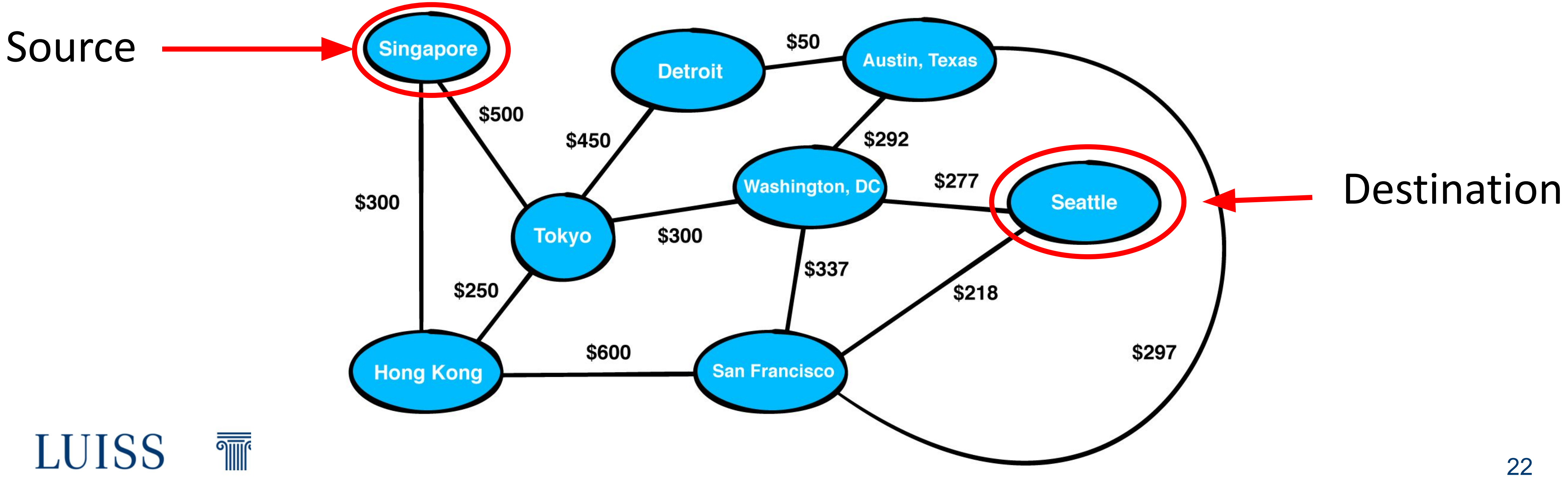
How can we do that?



Graphs - Path

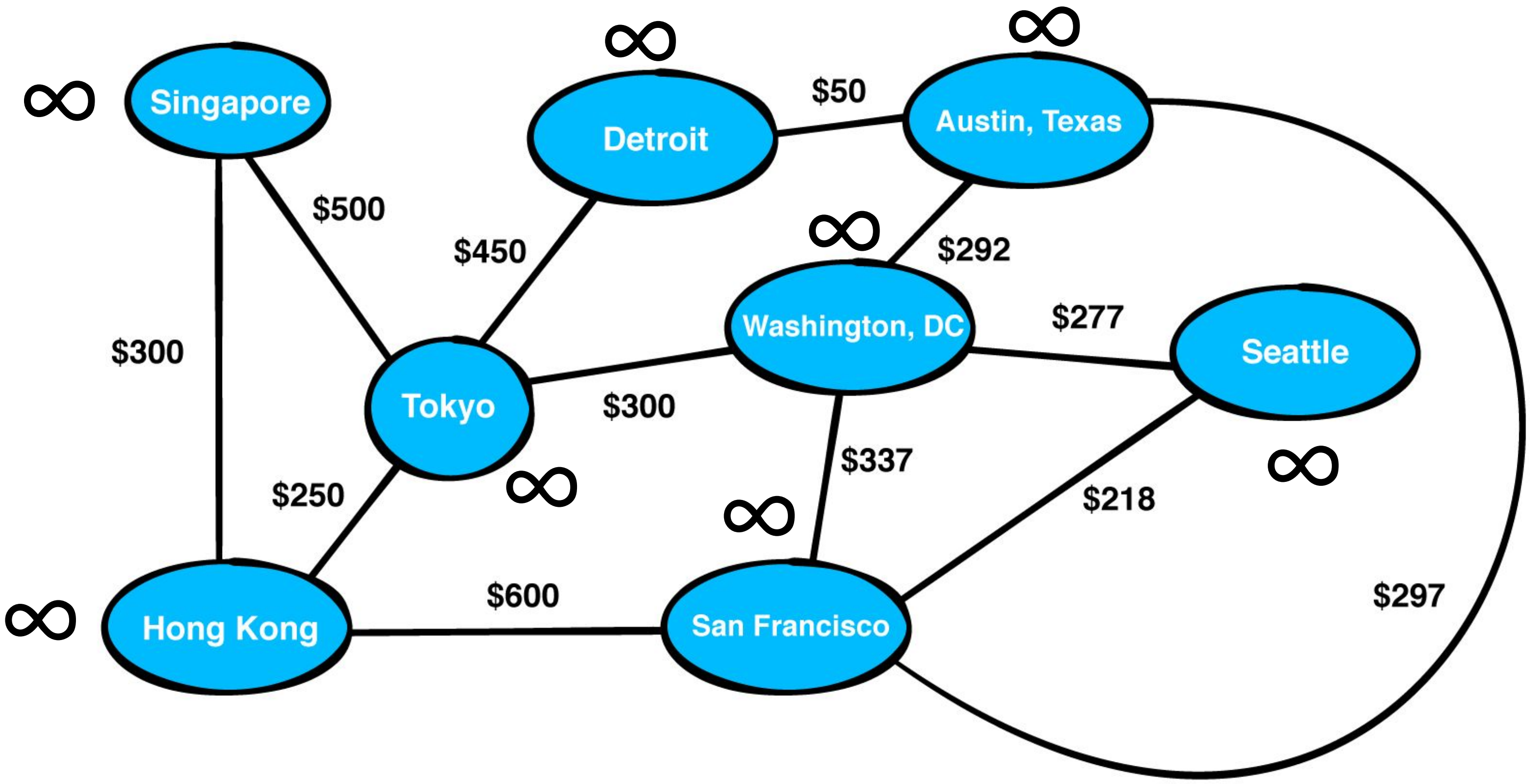
How can we do that?

Dijkstra algorithm!



Graphs - Dijkstra Algorithm

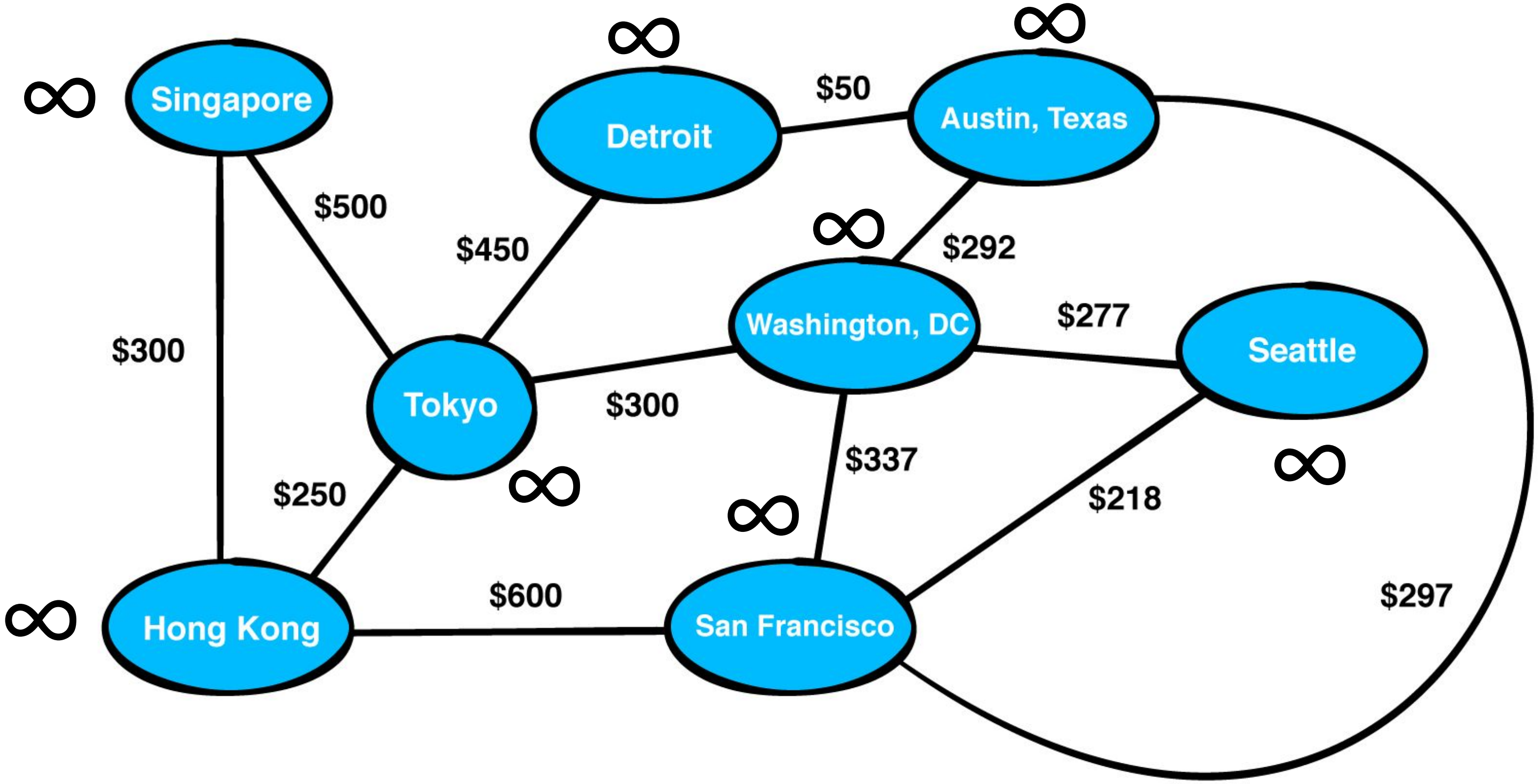
First of all we set all the distances to infinity for every node



LUISS

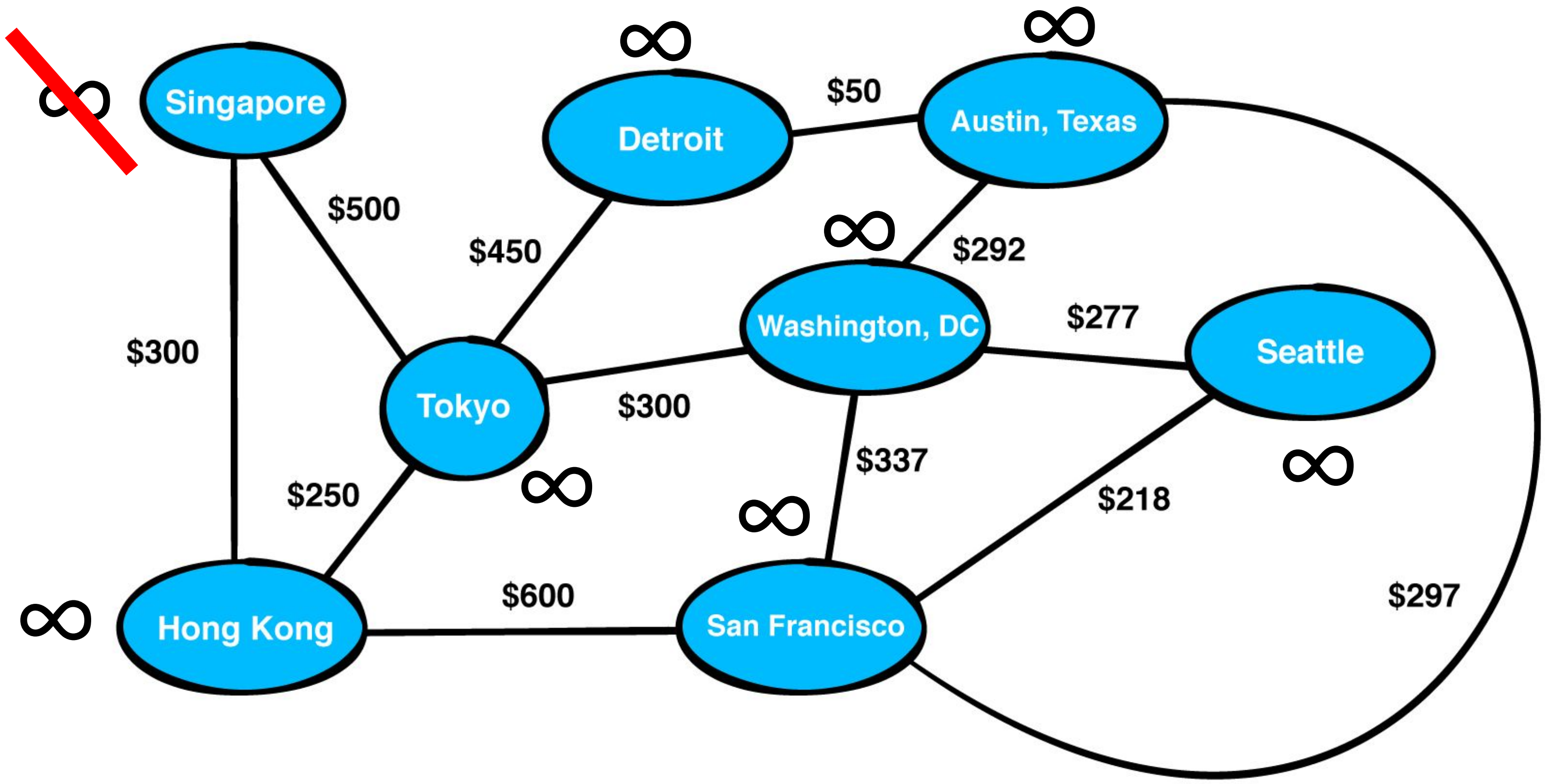
Graphs - Dijkstra Algorithm

Starting from the source node (Here Singapore) we start changing the cost to reach any given node



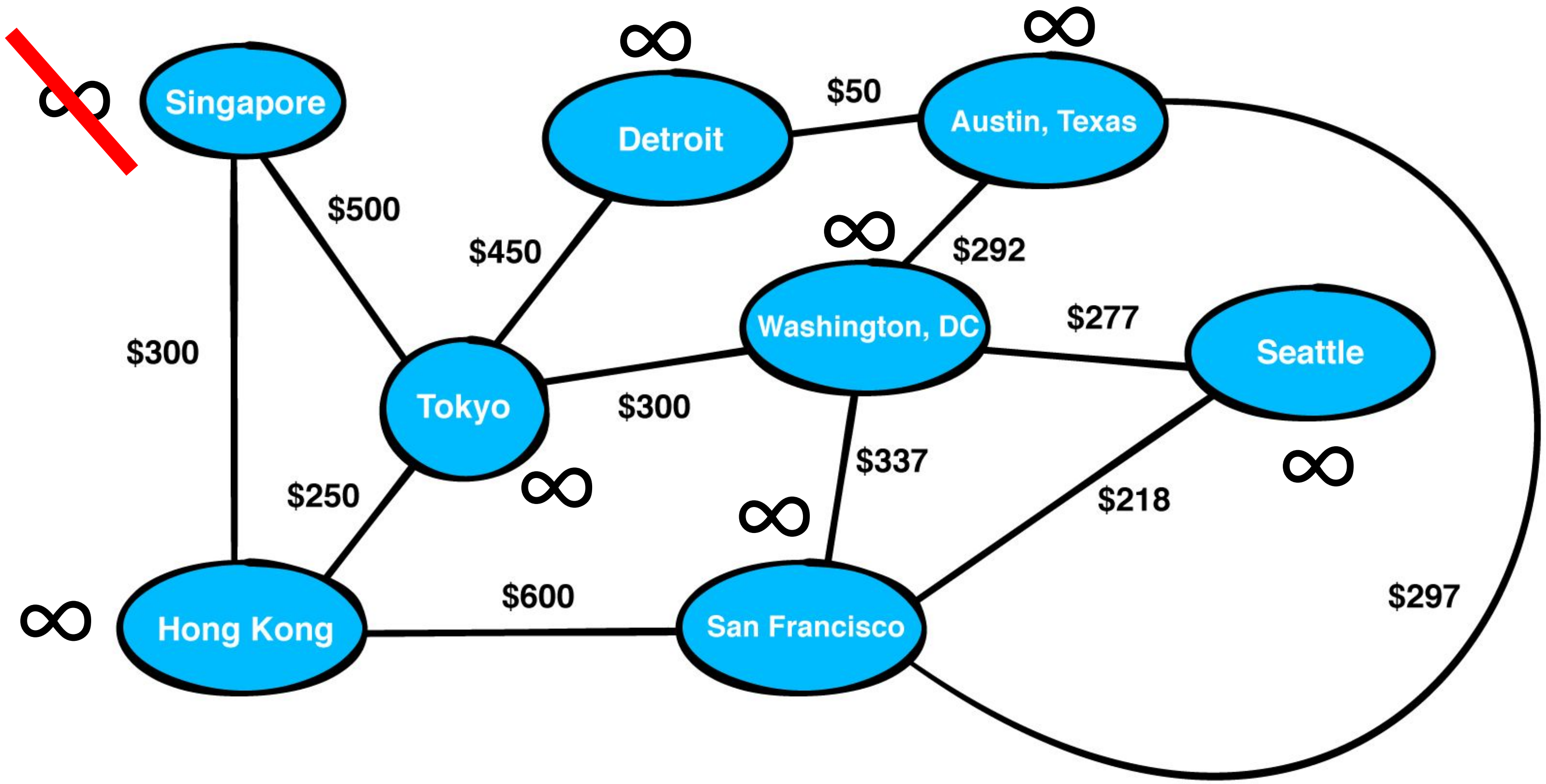
Graphs - Dijkstra Algorithm

Starting from the source node (Here Singapore) we start changing the cost to reach any given node



Graphs - Dijkstra Algorithm

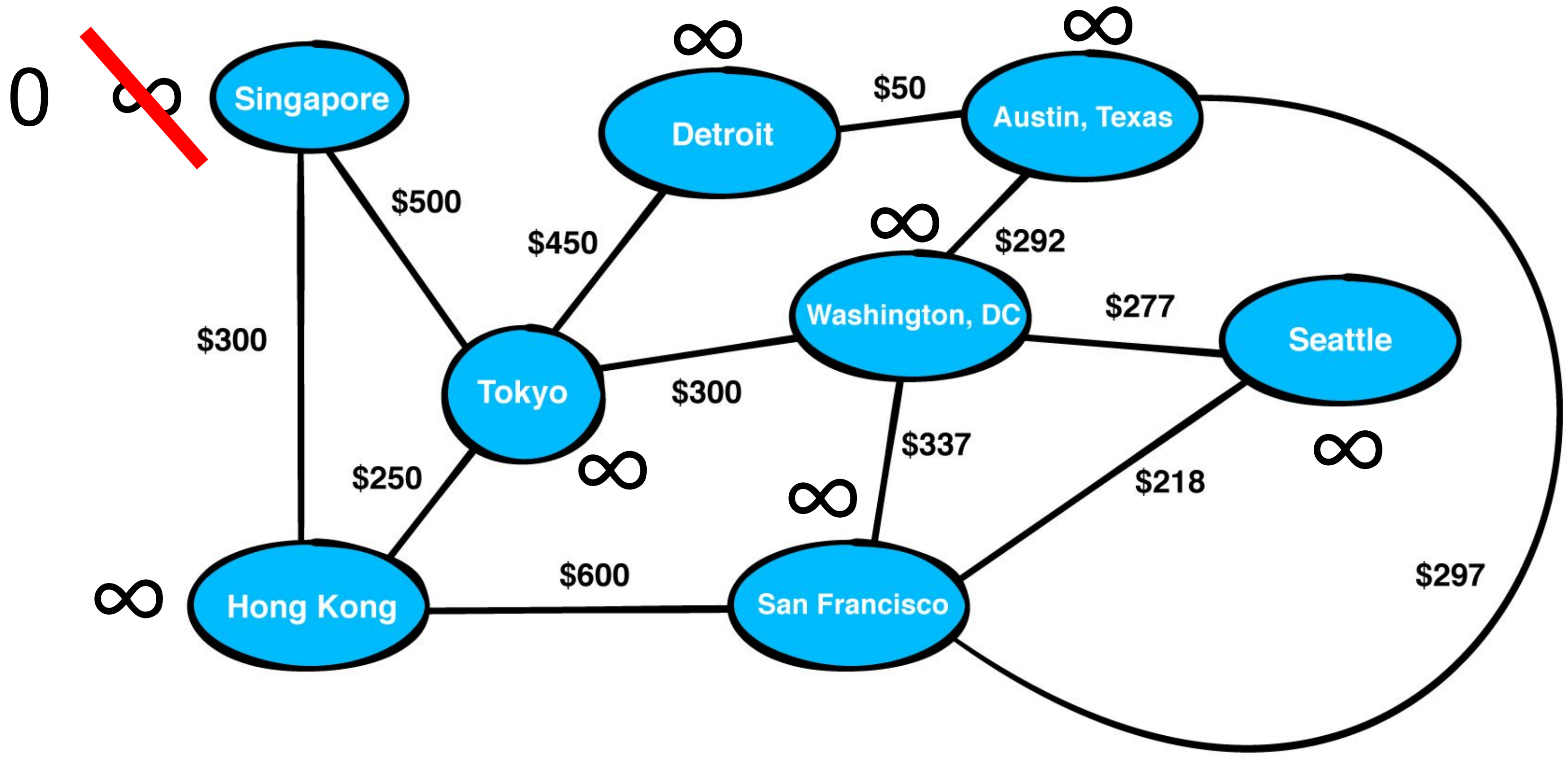
Reach Singapore from Singapore costs 0 Dollars!



LUISS

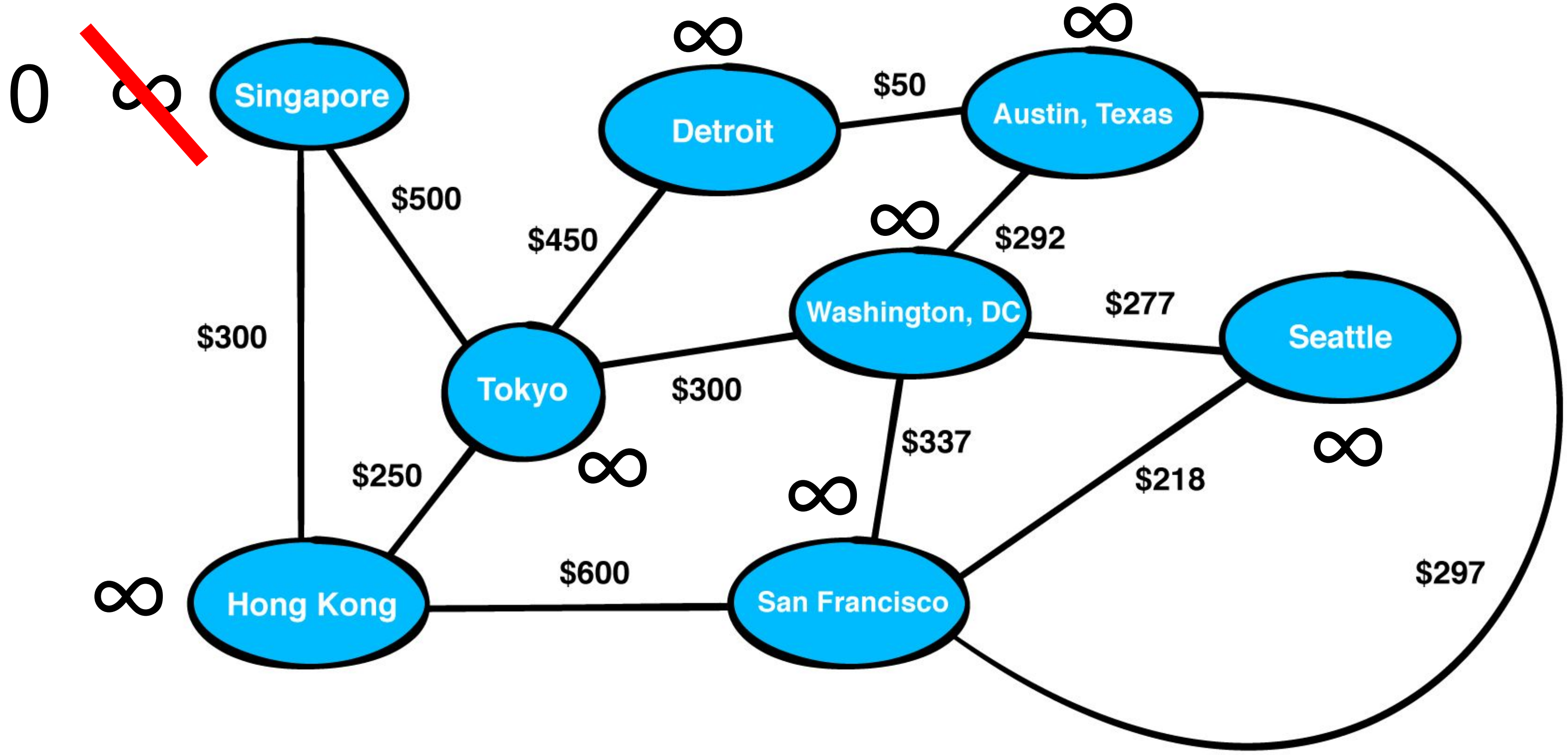
Graphs - Dijkstra Algorithm

Reach Singapore from Singapore costs 0 Dollars! So we can change the cost to zero.



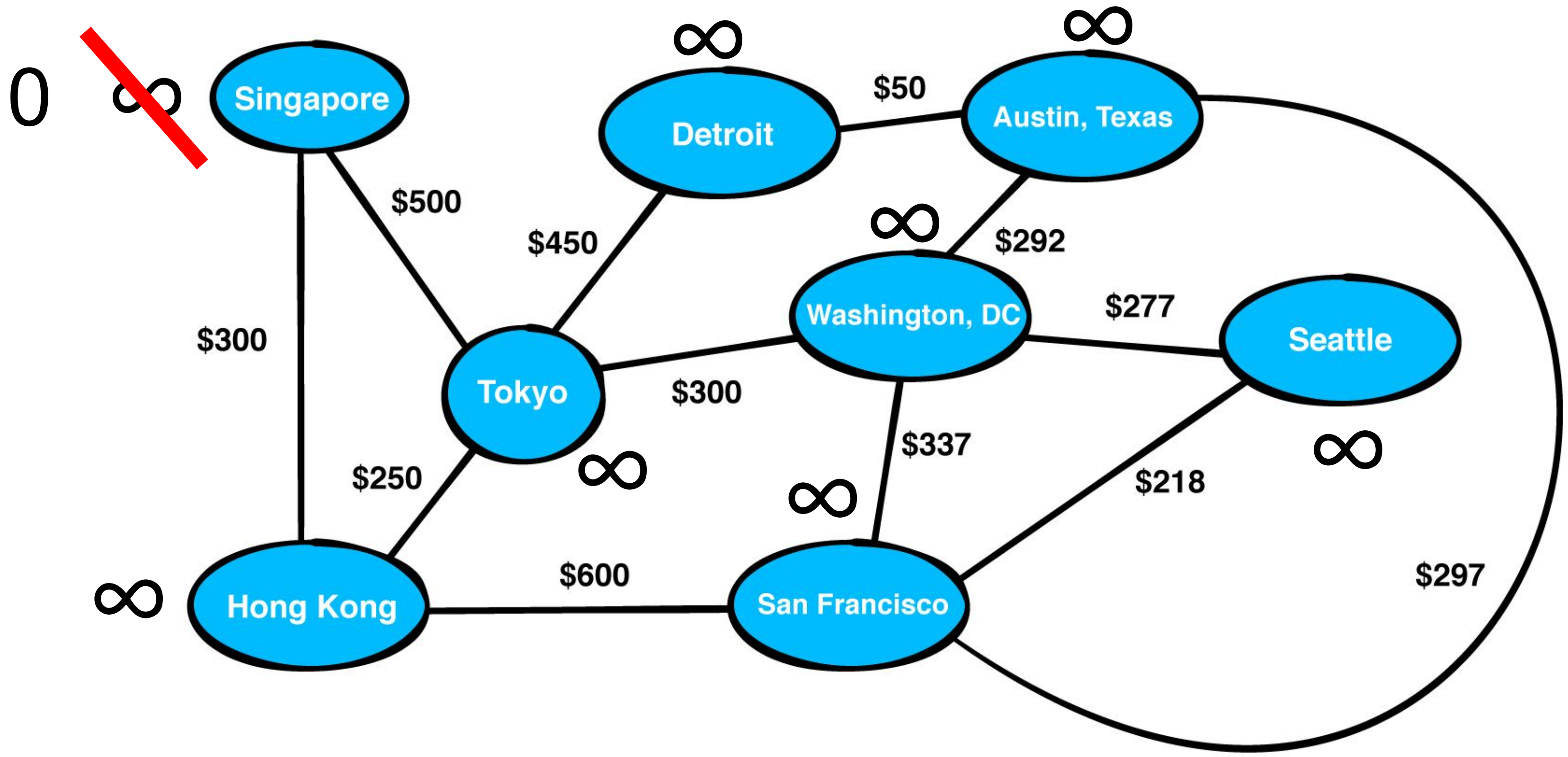
Graphs - Dijkstra Algorithm

Now from here we have to explore the outgoing links from the current node.



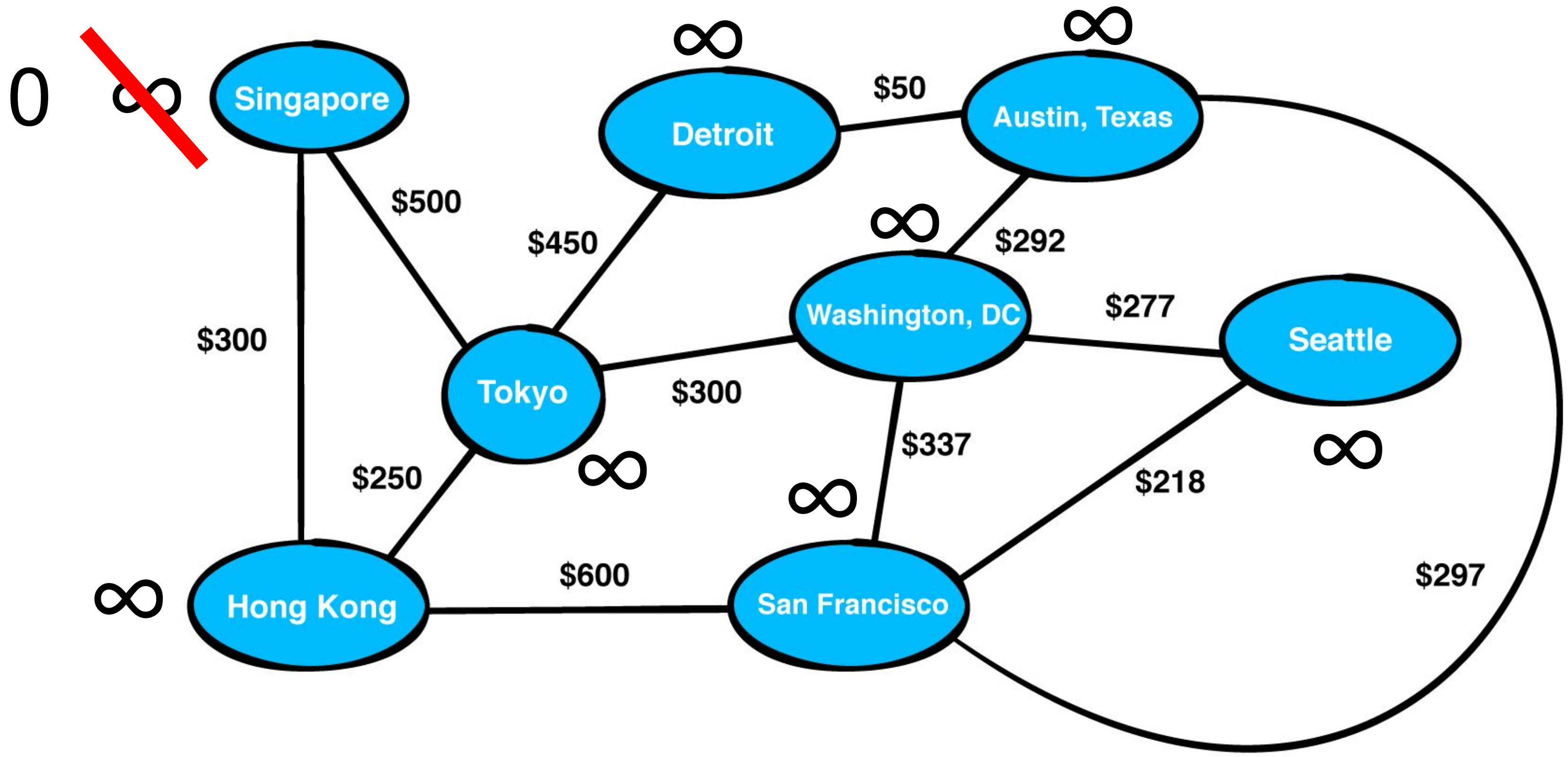
Graphs - Dijkstra Algorithm

For each node adjacent to the current node we have to ask 2 things: **it has been visited?**



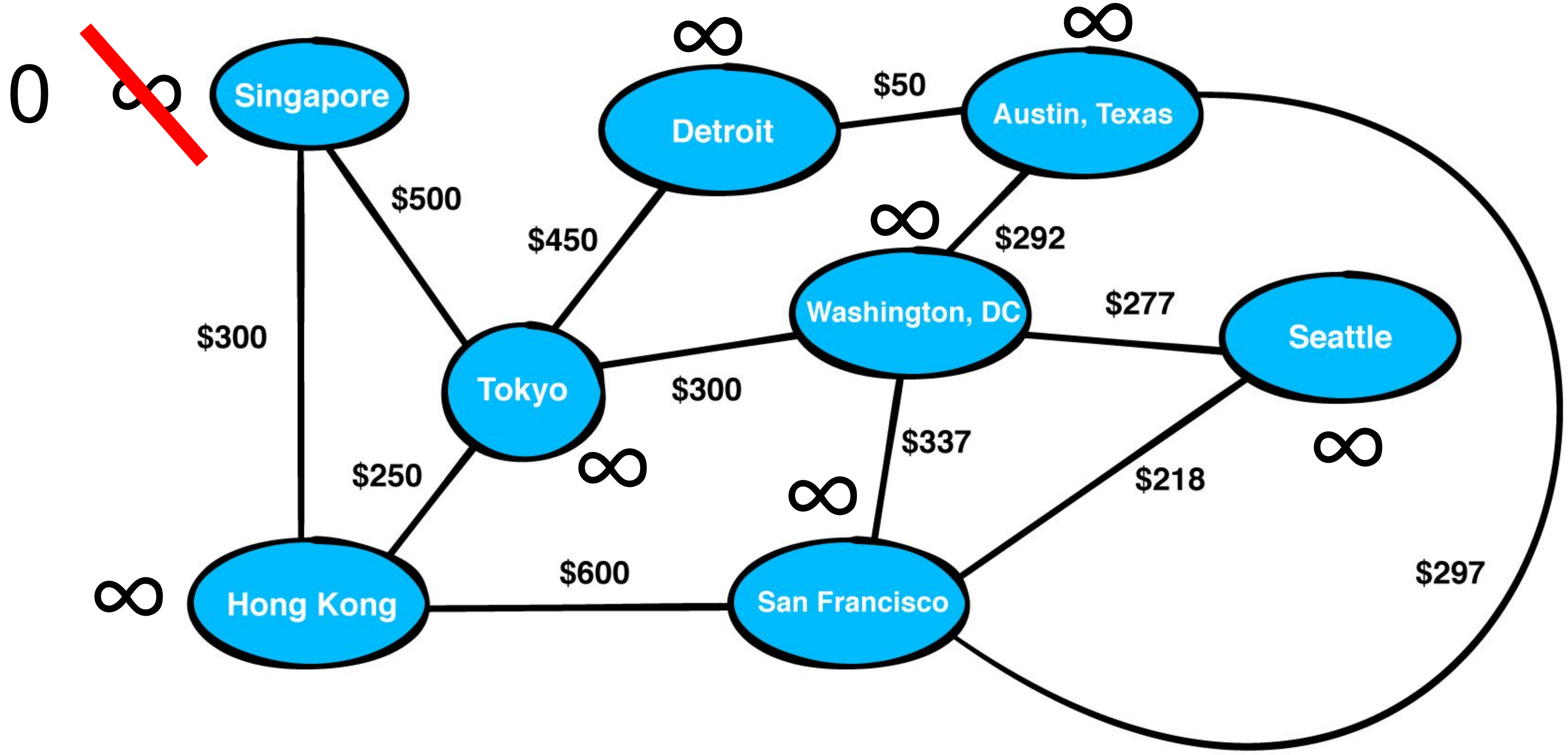
Graphs - Dijkstra Algorithm

For each node adjacent to the current node we have to ask 2 things: **Is the cost to reach that node from the current node lower than the current cost?**



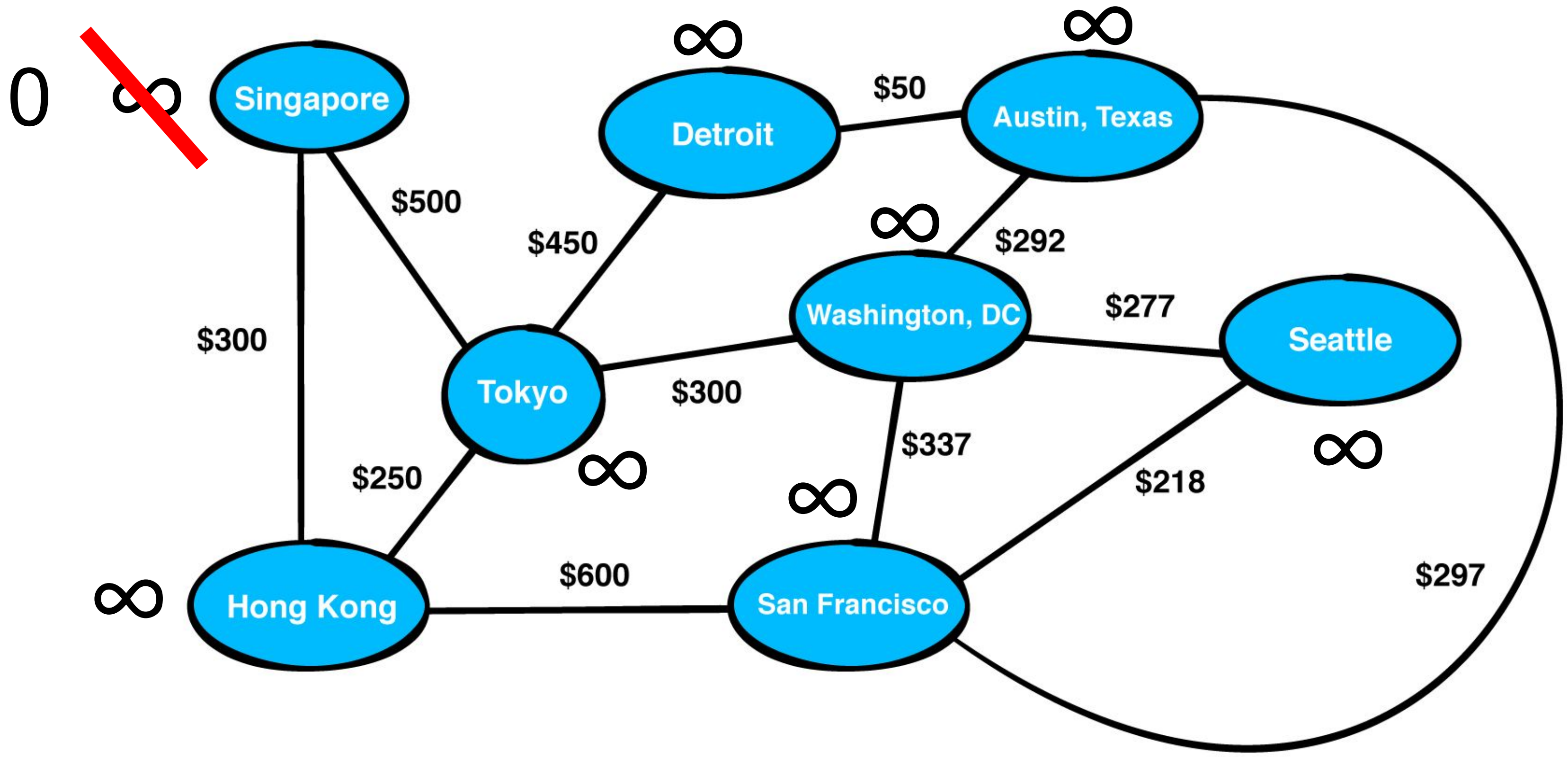
Graphs - Dijkstra Algorithm

First of all we go to the **Hong Kong** node. The current cost of this node is infinity.



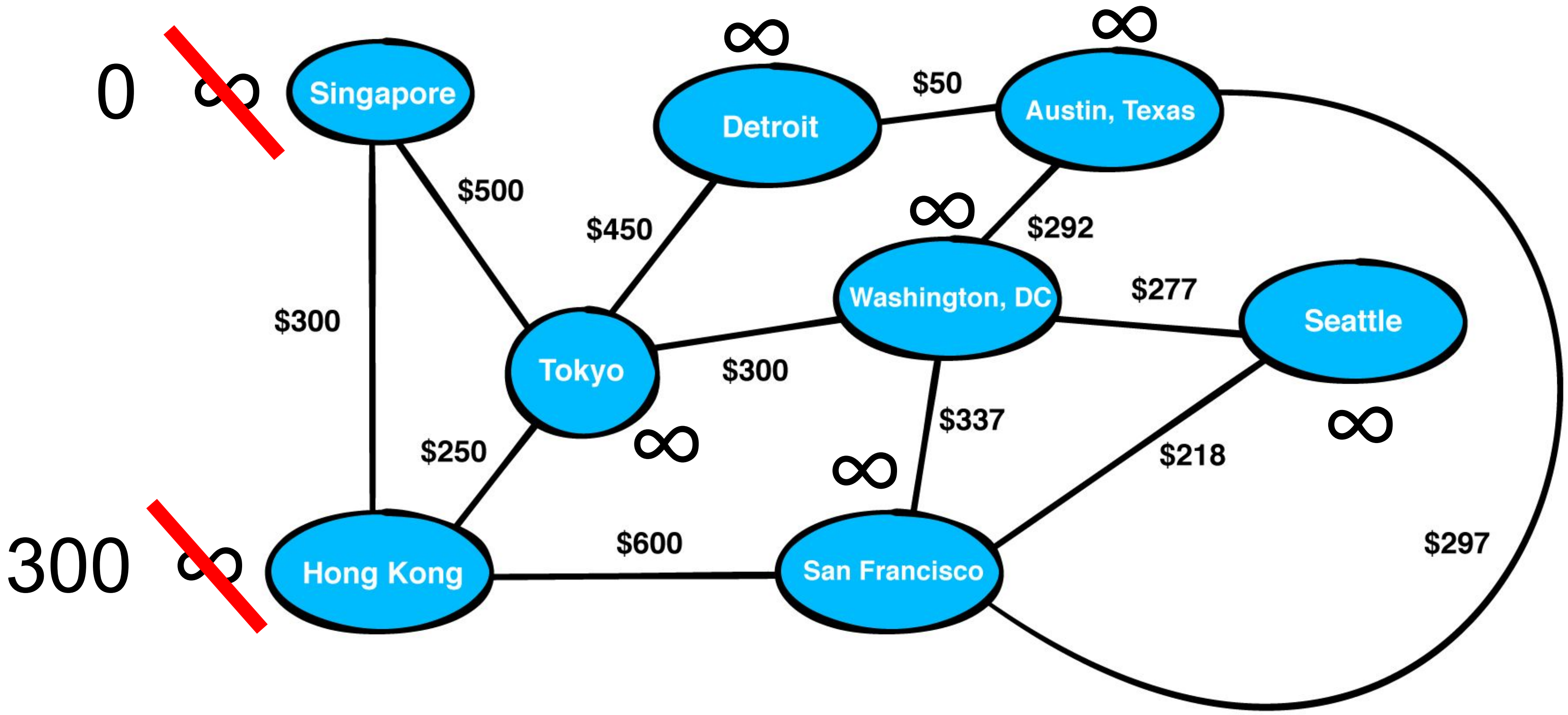
Graphs - Dijkstra Algorithm

To reach that node from **Singapore** we have to add the cost to reach Singapore (0) and the cost of the link (300)



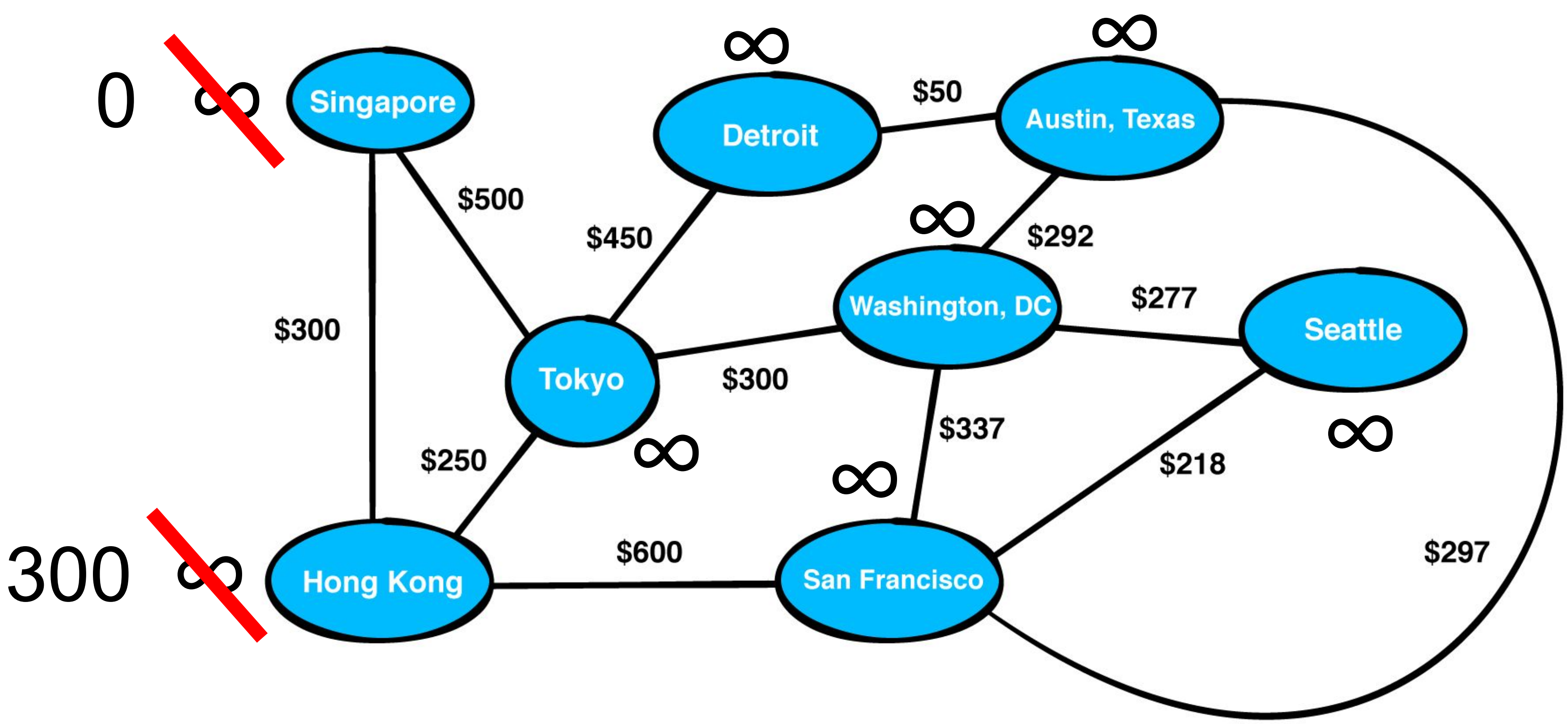
Graphs - Dijkstra Algorithm

Since 0 + 300 = 300 < ∞ then we change the cost of **Hong Kong**



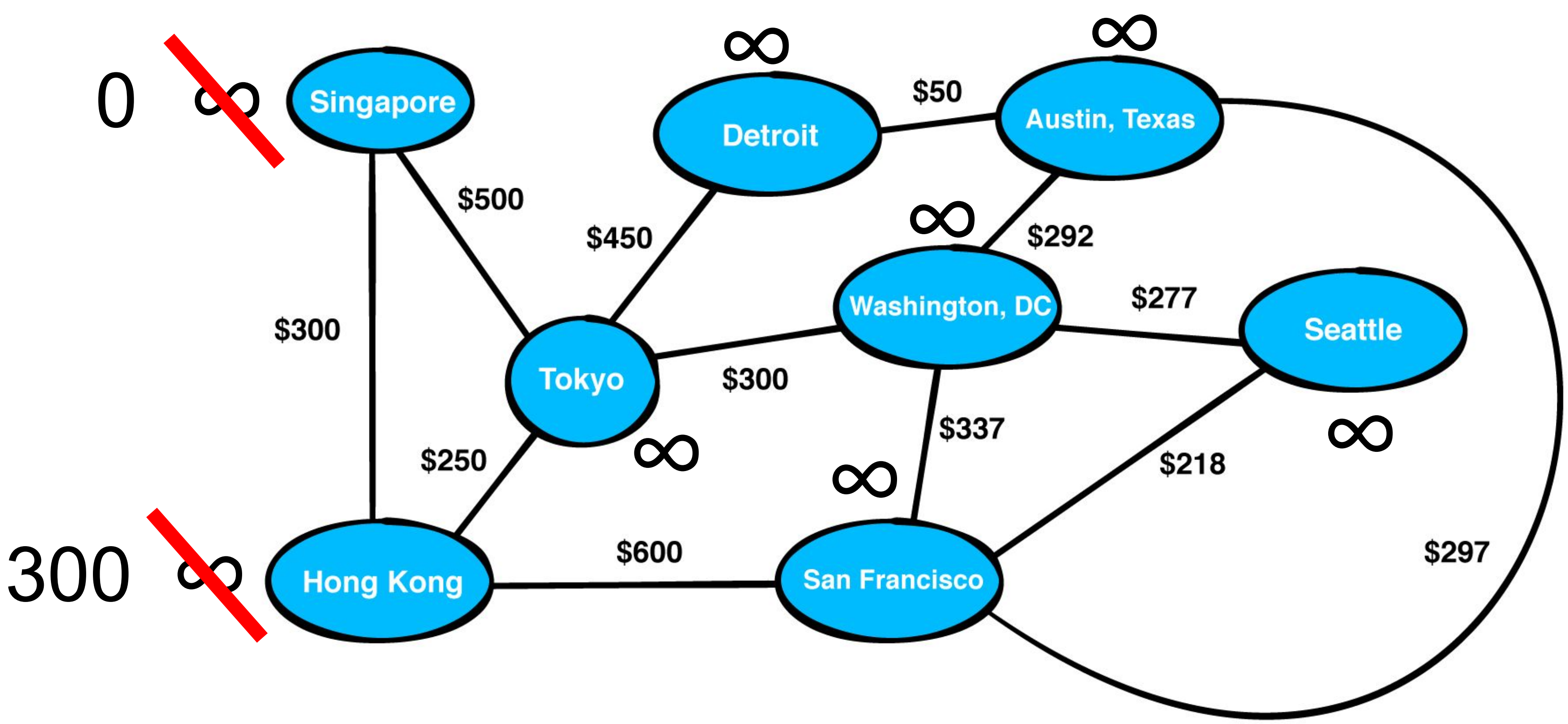
Graphs - Dijkstra Algorithm

Same thing happens for **Tokyo.**



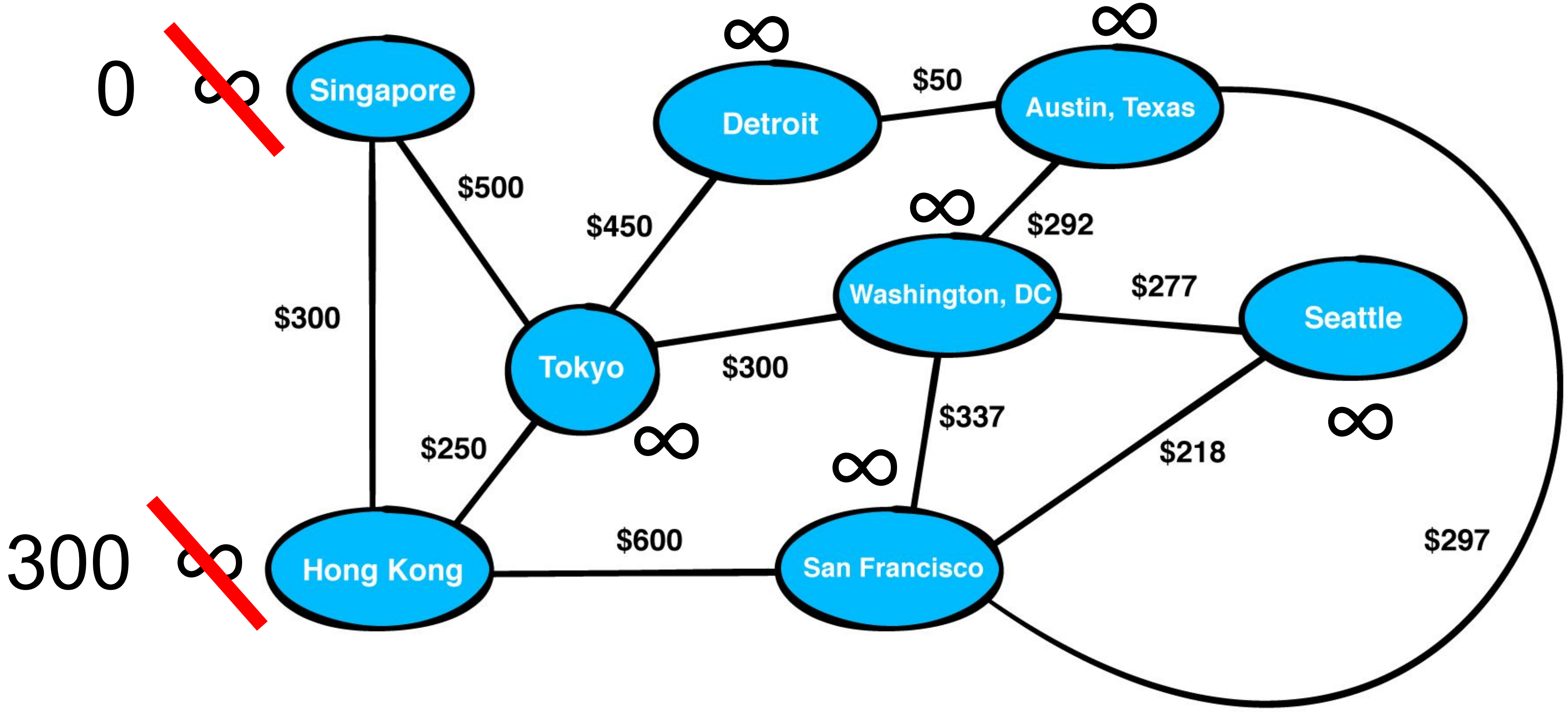
Graphs - Dijkstra Algorithm

We ask has **Tokyo** been visited? **NO**



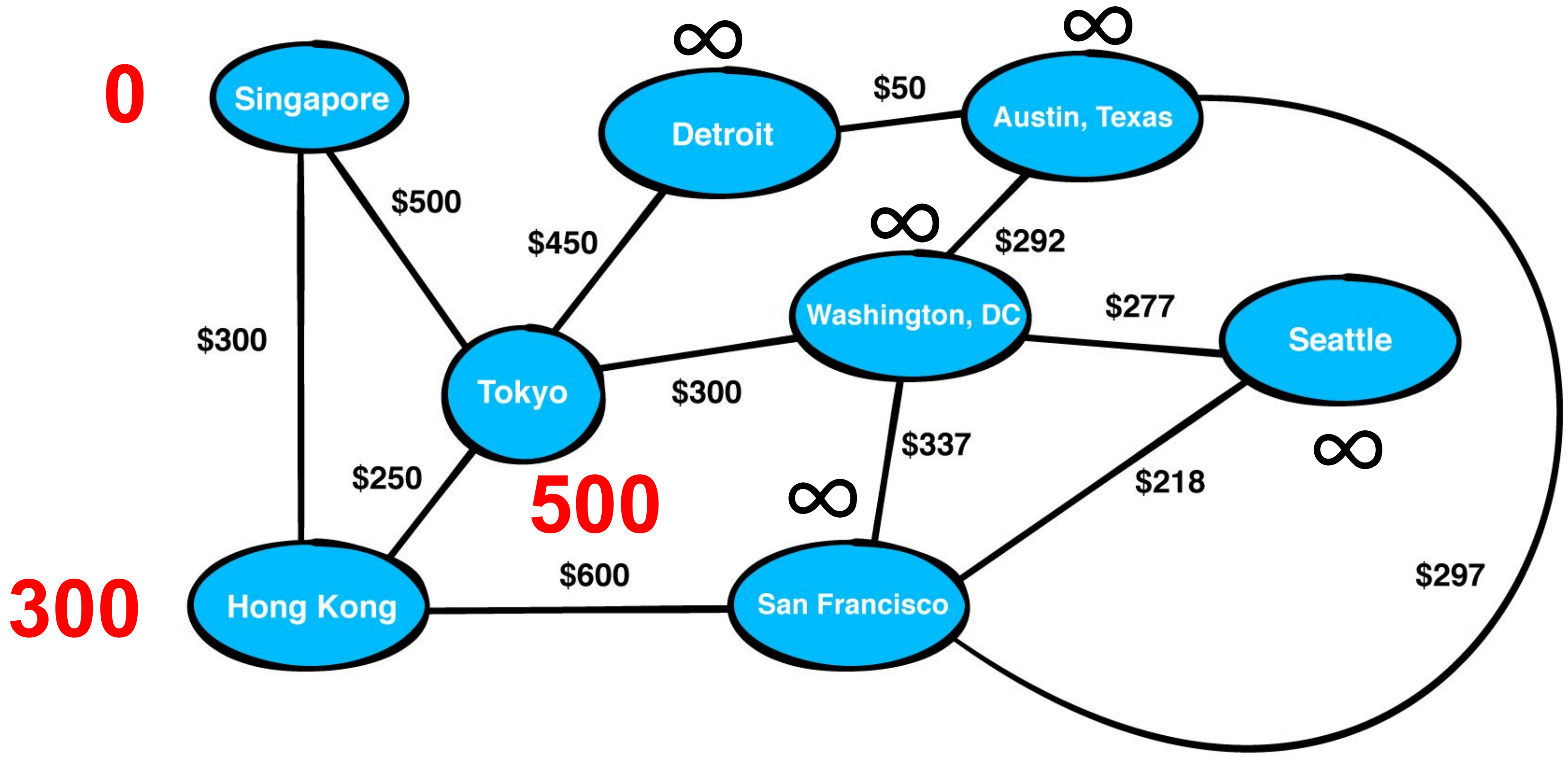
Graphs - Dijkstra Algorithm

Is the cost of **Tokyo** smaller than the cost of **Singapore** plus the cost of the link between the two nodes? **NO!** Because 0 + 500 < ∞



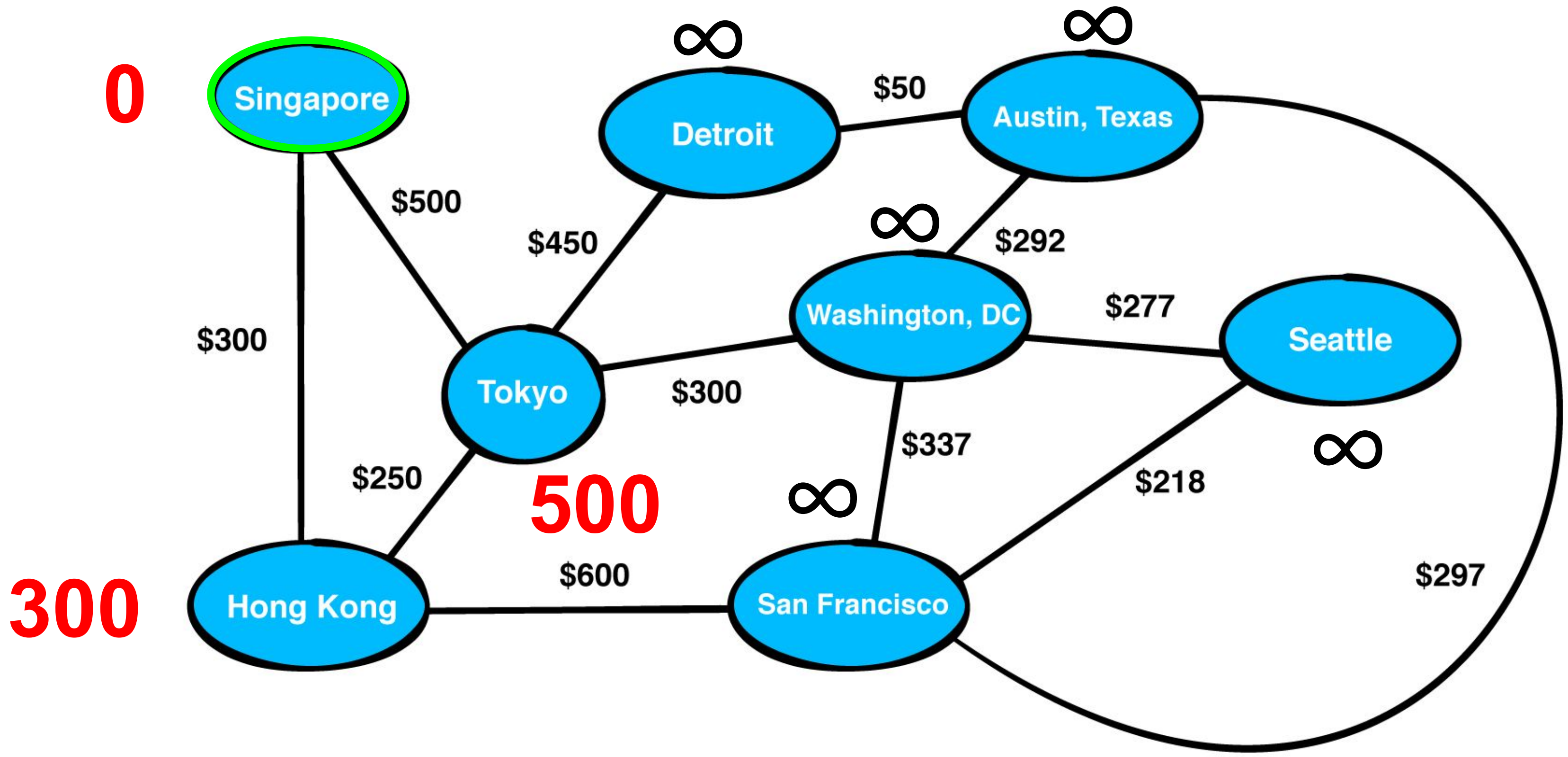
Graphs - Dijkstra Algorithm

We can change the cost of **Tokyo** to 500



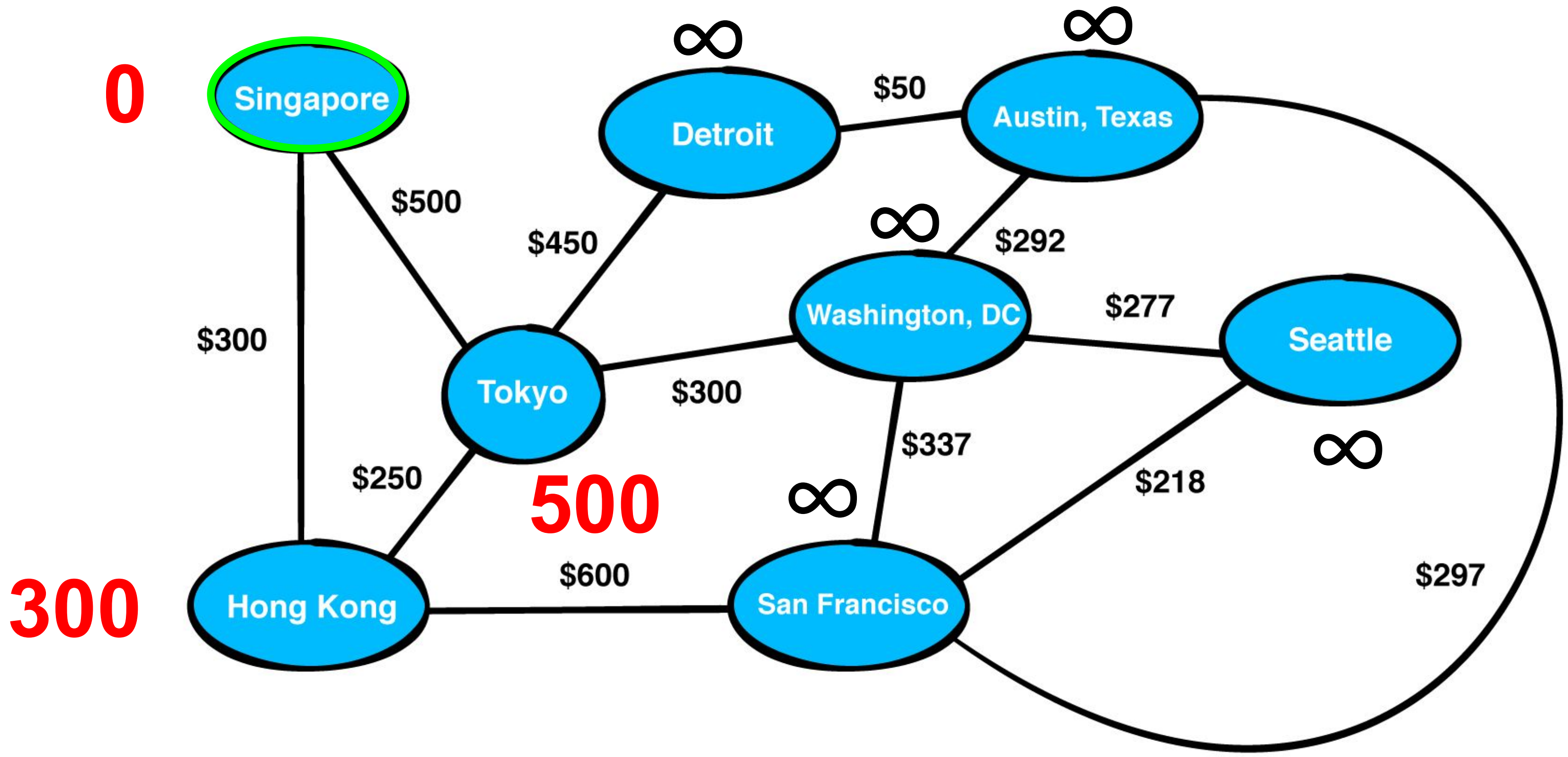
Graphs - Dijkstra Algorithm

Singapore has no other adjacent nodes so we can mark it as visited



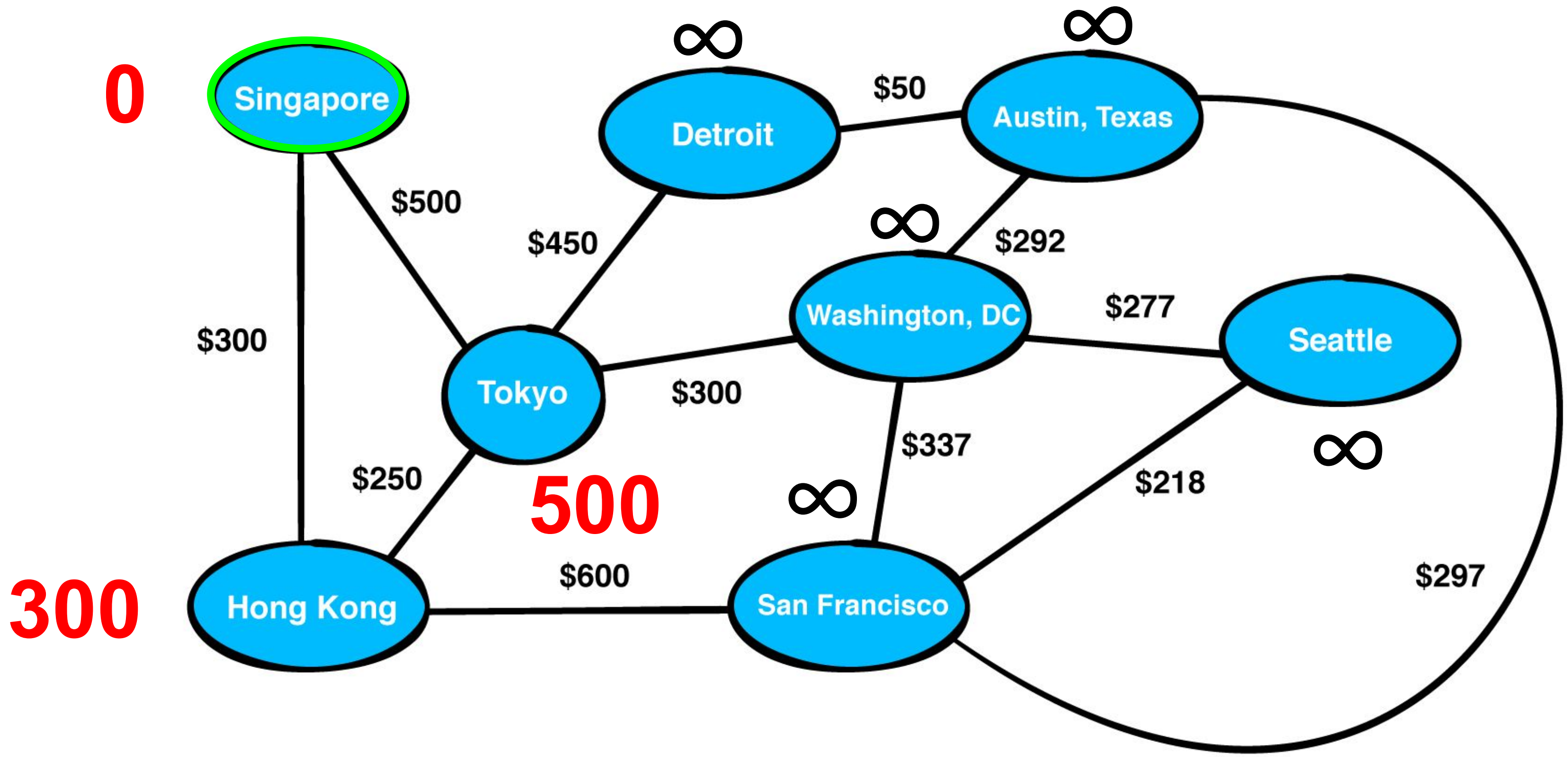
Graphs - Dijkstra Algorithm

Now we have to select the node among the adjacent nodes of **Singapore** that has the smallest cost and it has not been visited yet



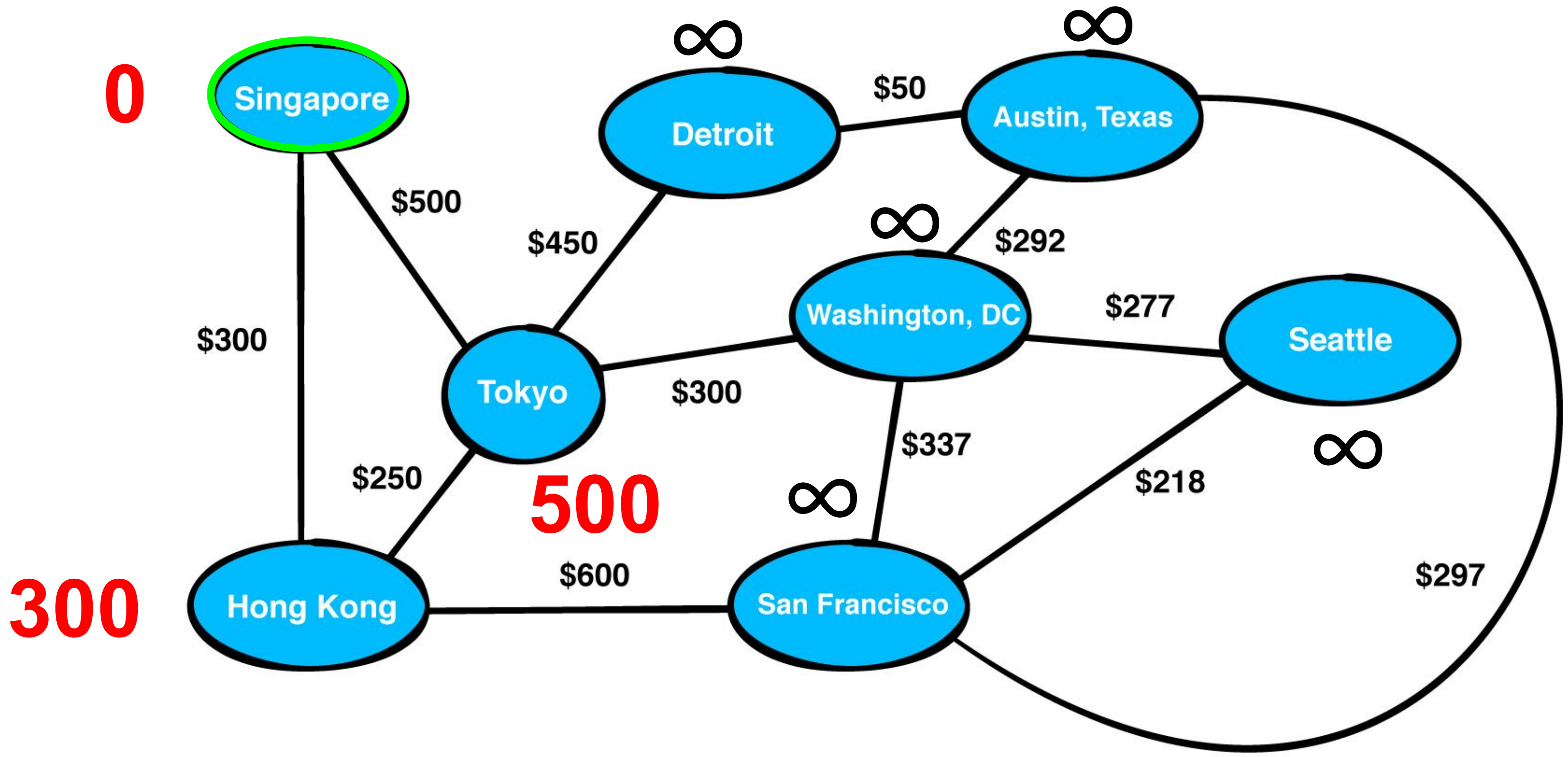
Graphs - Dijkstra Algorithm

So neither **Hong Kong nor Tokyo** has been visited. We select **Hong Kong** as next node to explore because of the cost.



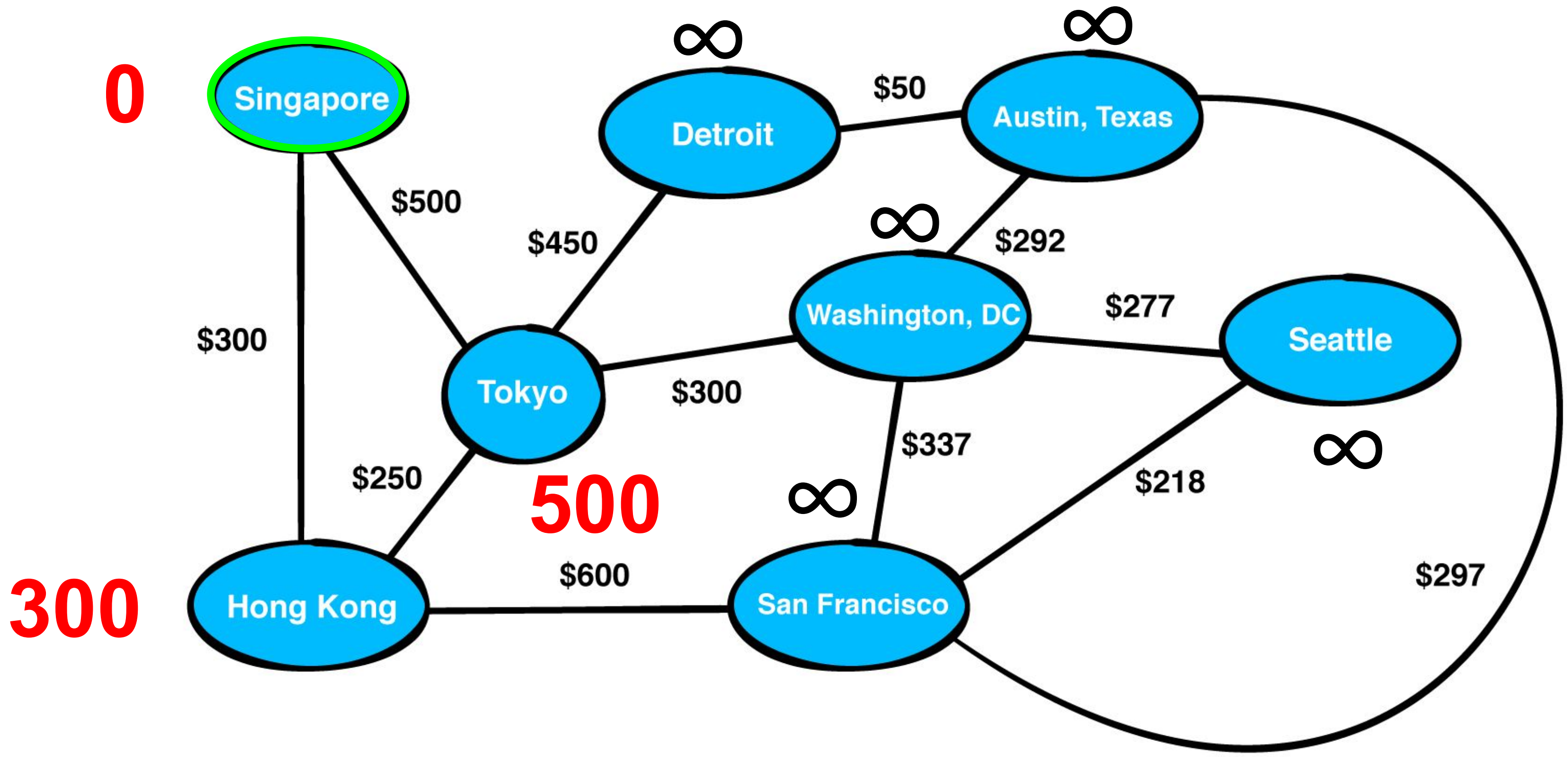
Graphs - Dijkstra Algorithm

We start exploring the adjacent nodes of Hong Kong (Tokyo and San Francisco)



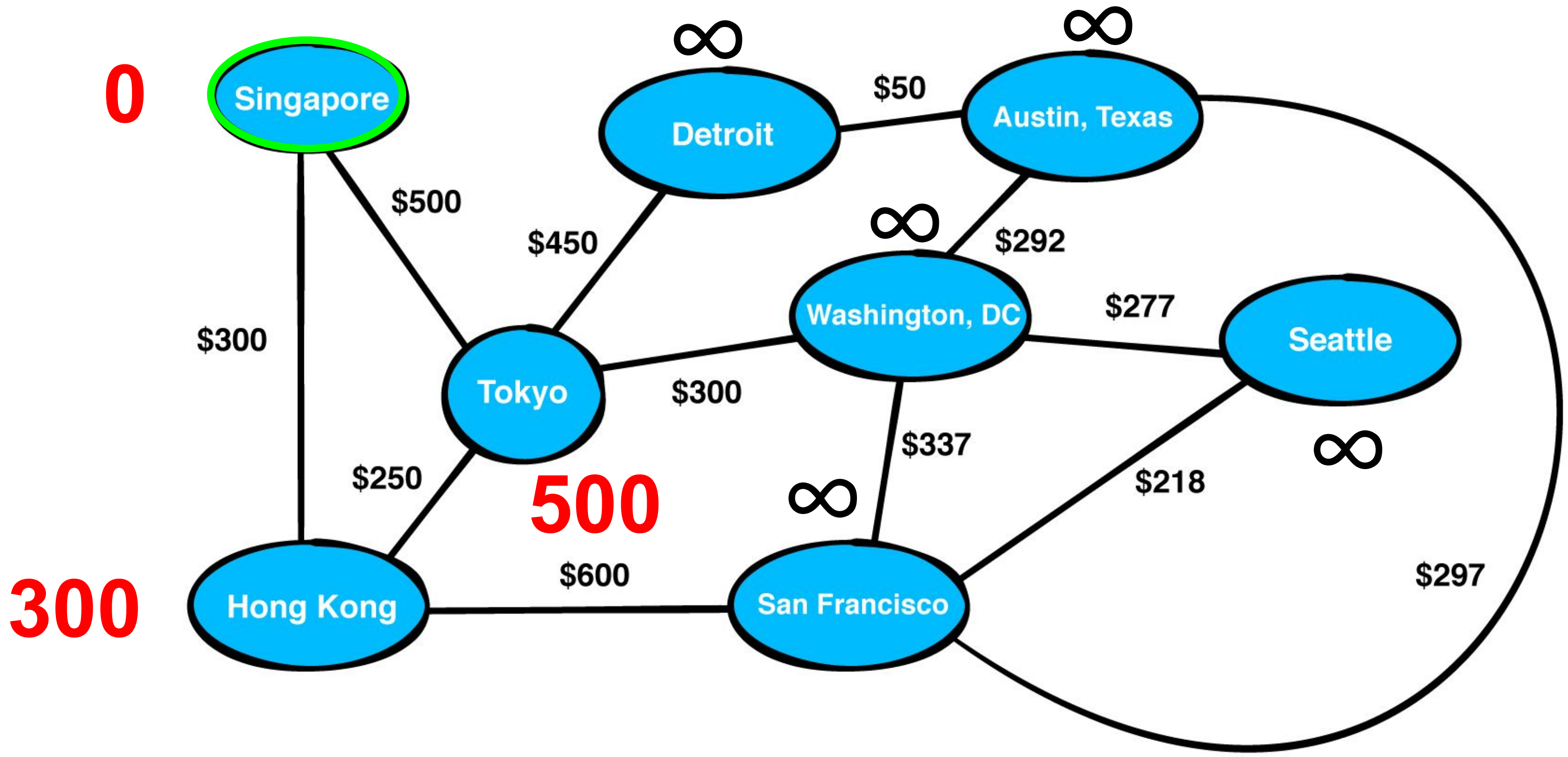
Graphs - Dijkstra Algorithm

Starting with Tokyo we always ask the two questions.



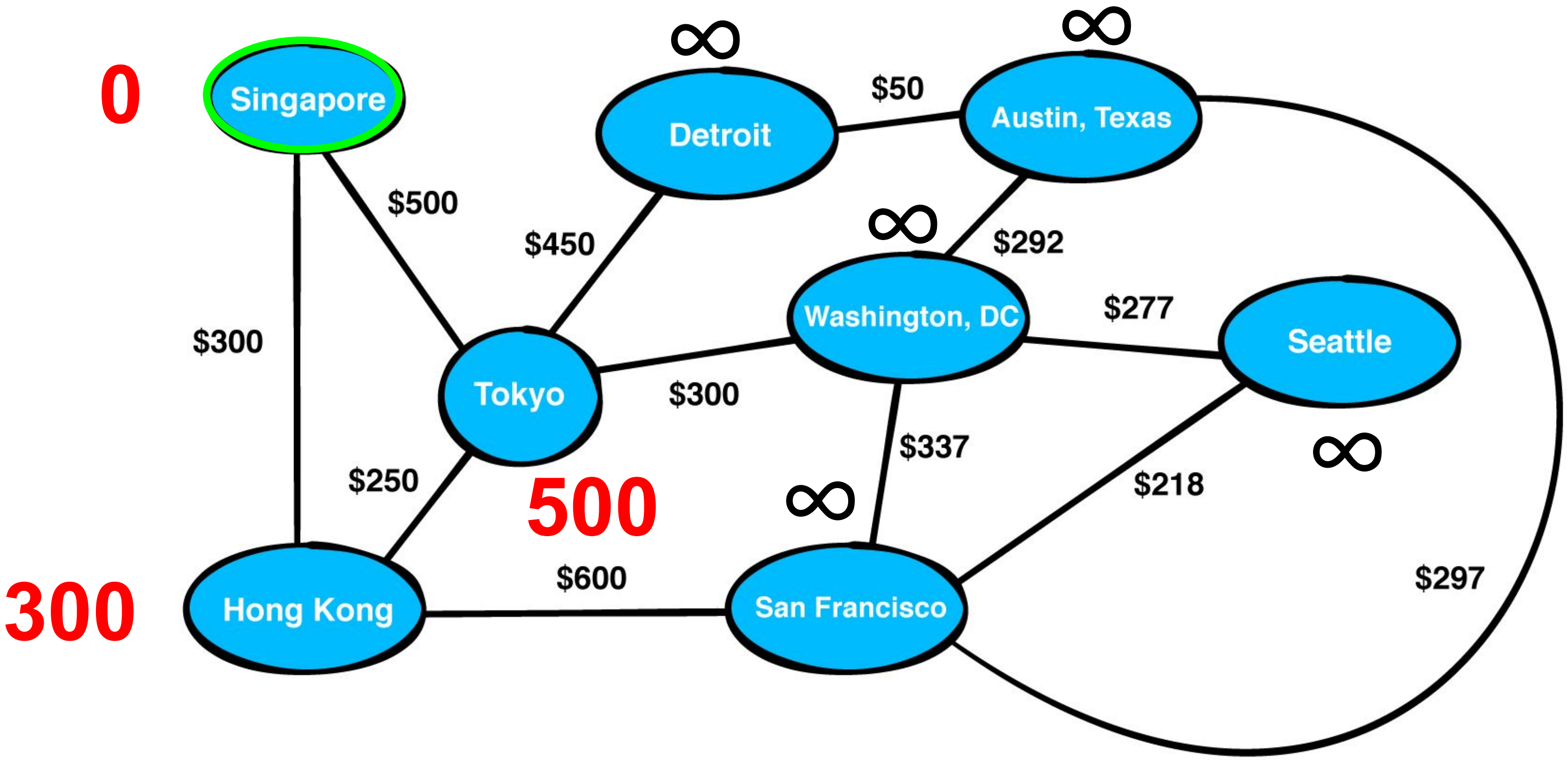
Graphs - Dijkstra Algorithm

Has Tokyo been visited? **NO**, Is the cost of Tokyo (500) smaller than the cost of Hong Kong plus the cost from Hong Kong to Tokyo? **NO! 500 < 300 + 250**



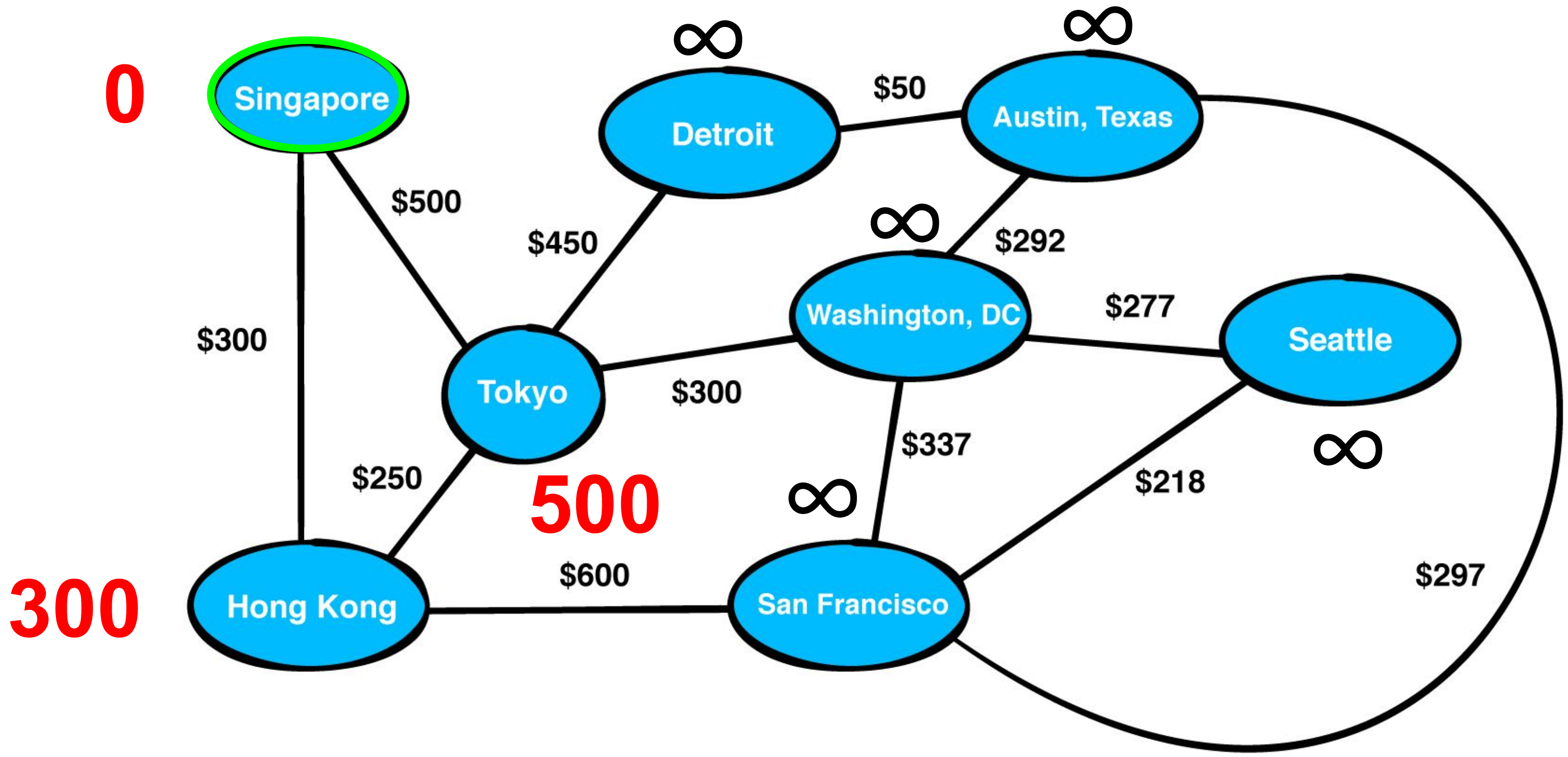
Graphs - Dijkstra Algorithm

So we do not modify anything. We explore **San Francisco** and ask alway the questions: has San Francisco been visited? **NO!**



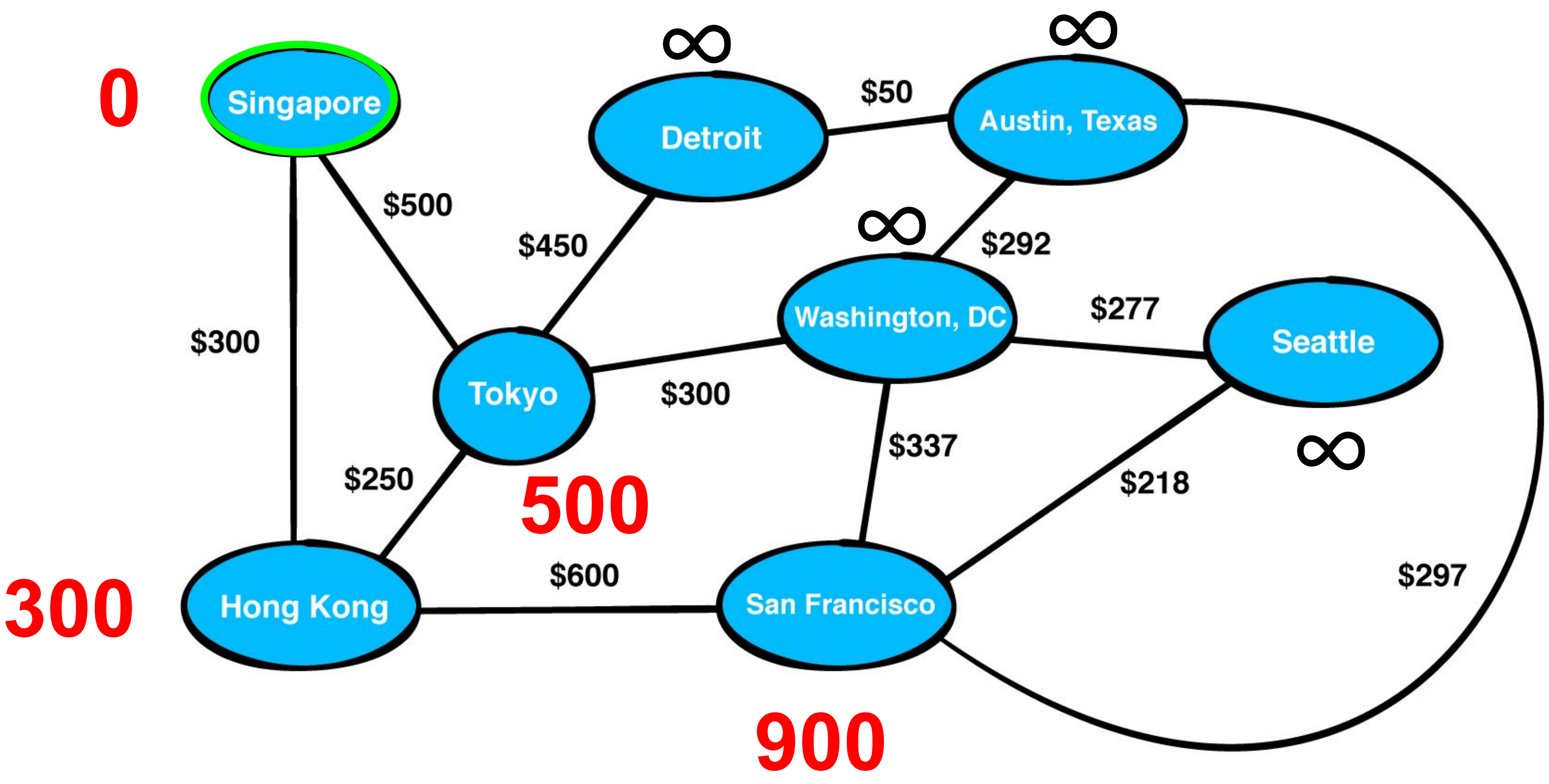
Graphs - Dijkstra Algorithm

Is the cost of San Francisco smaller than the cost of Hong Kong (300) plus the cost from Hong Kong to San Francisco (600)? **Yes!** 300 + 600 < ∞



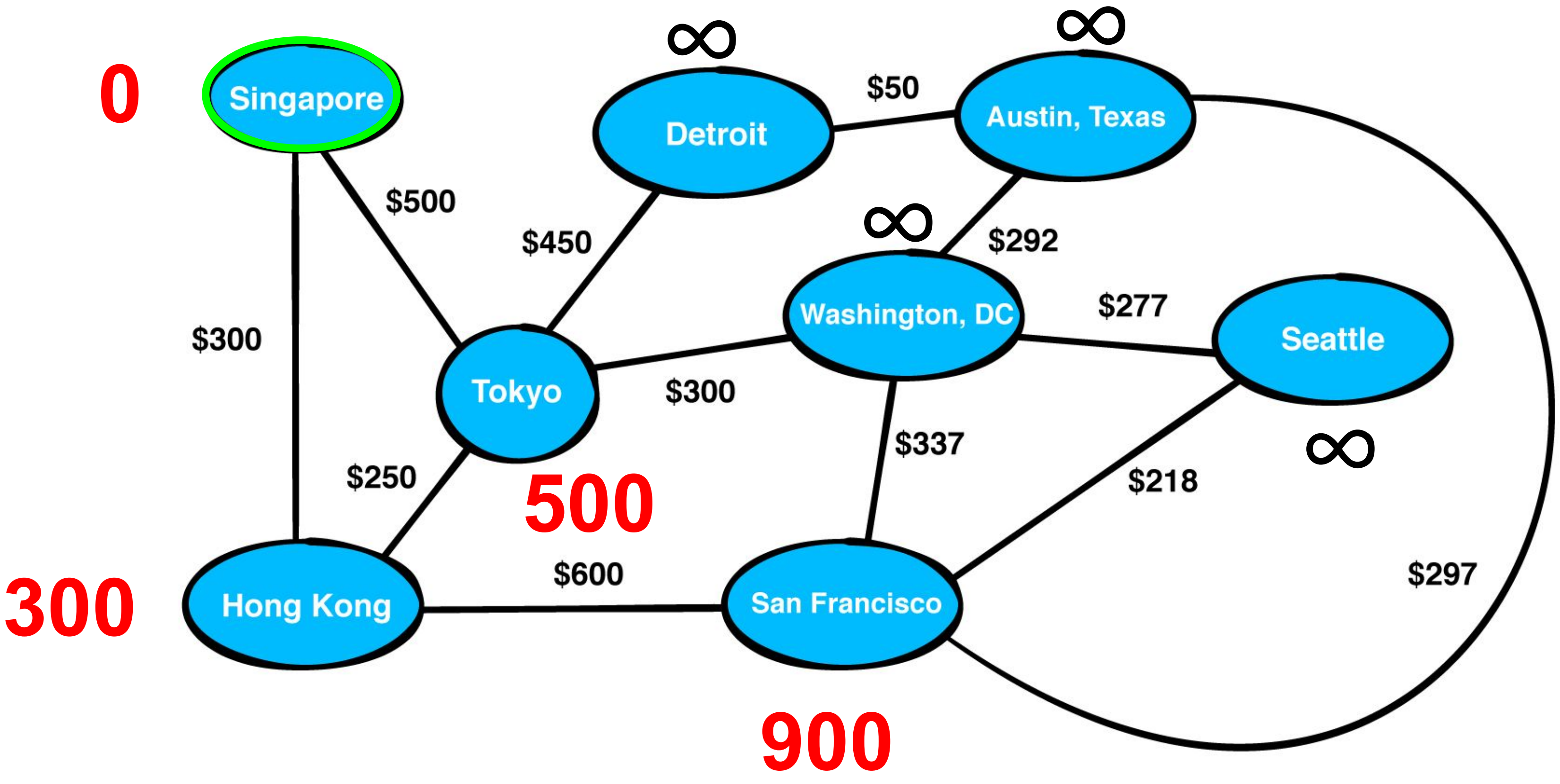
Graphs - Dijkstra Algorithm

So we change the cost of San Francisco to 600 + 300 = 900



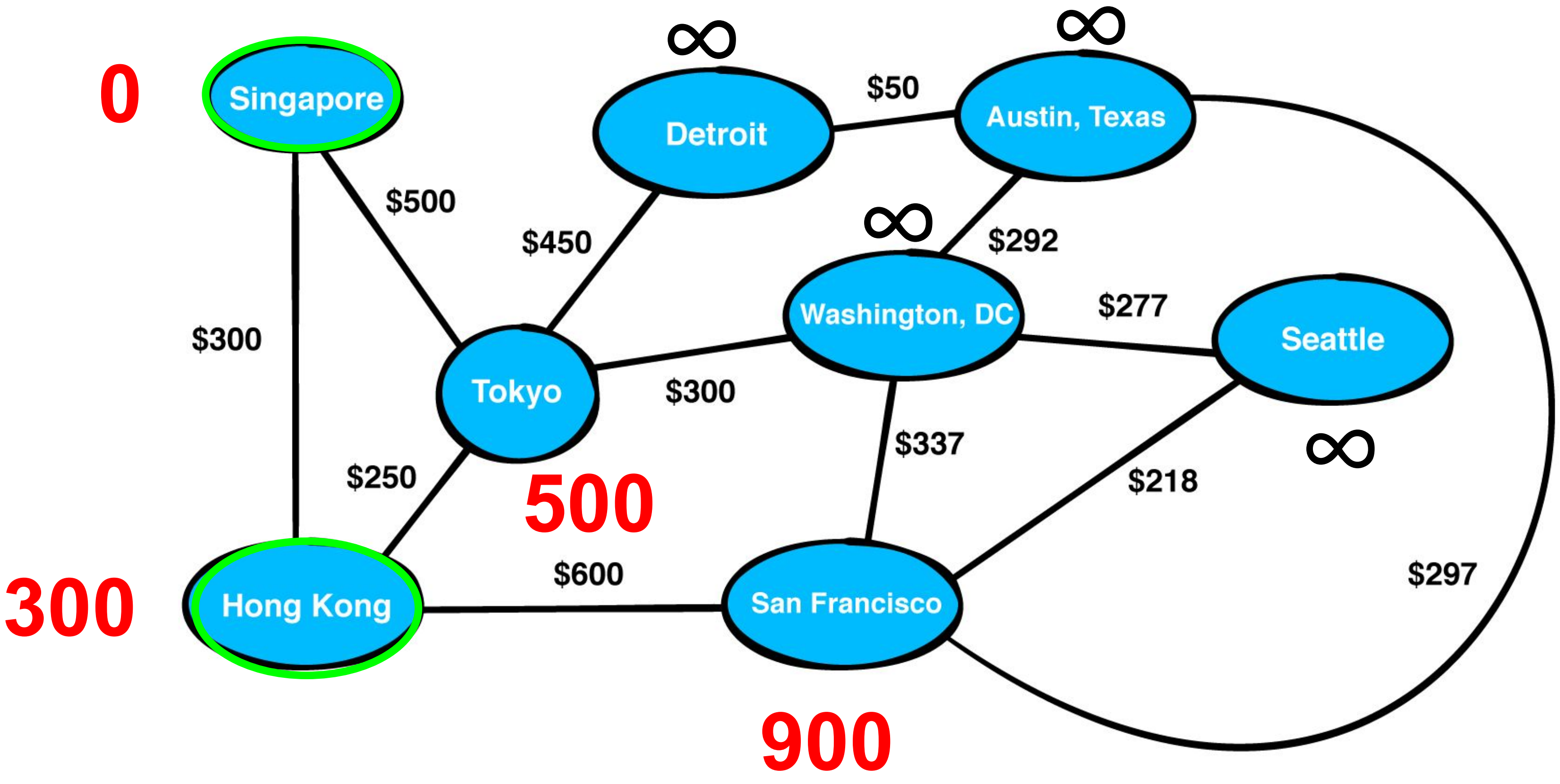
Graphs - Dijkstra Algorithm

Now Hong Kong has no more adjacent nodes, so we can set it as visited and we go to the next adjacent nodes that has the lowest cost.



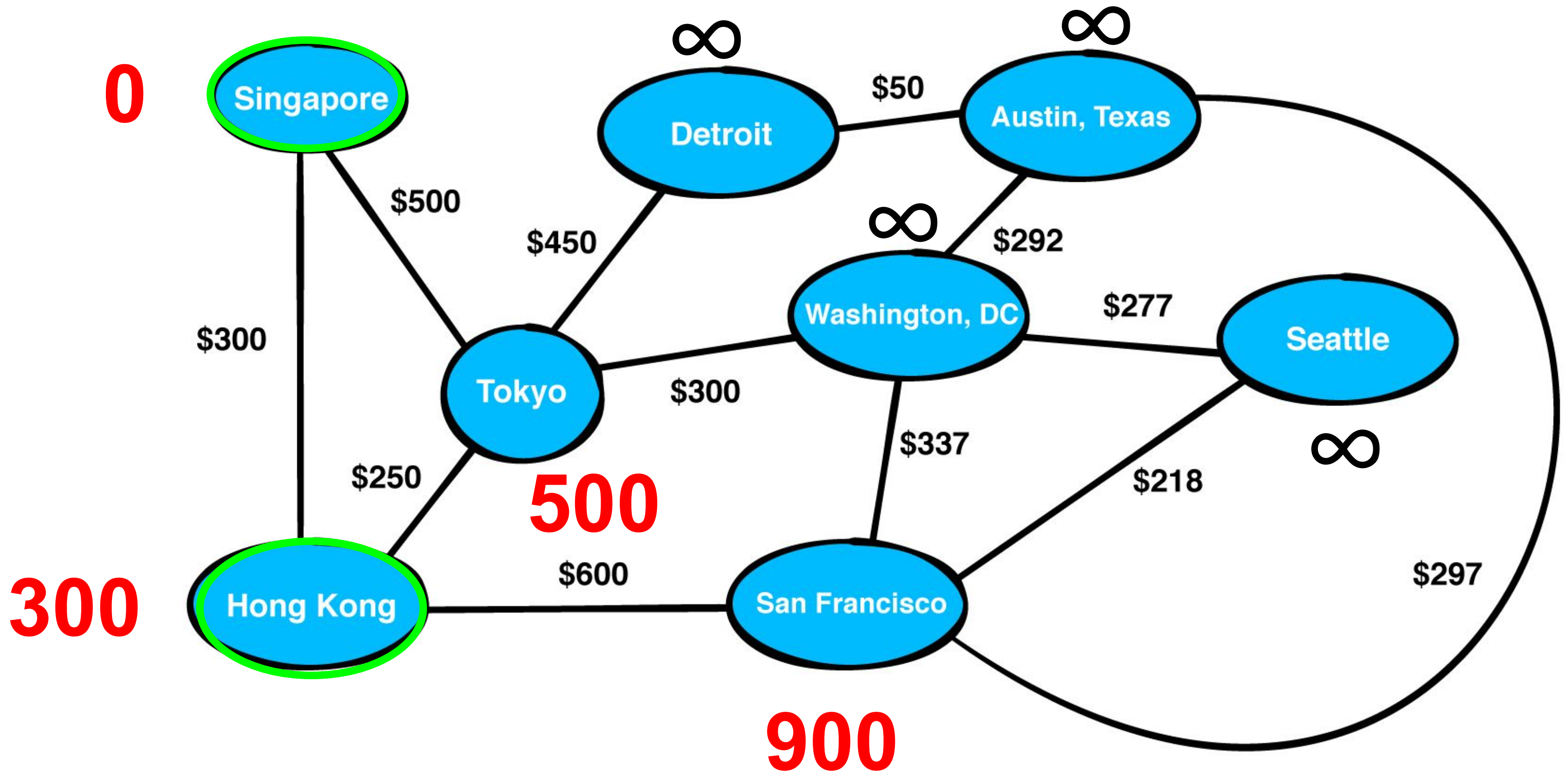
Graphs - Dijkstra Algorithm

This node is **Tokyo.** Now we start exploring the nodes directly linked with Tokyo.



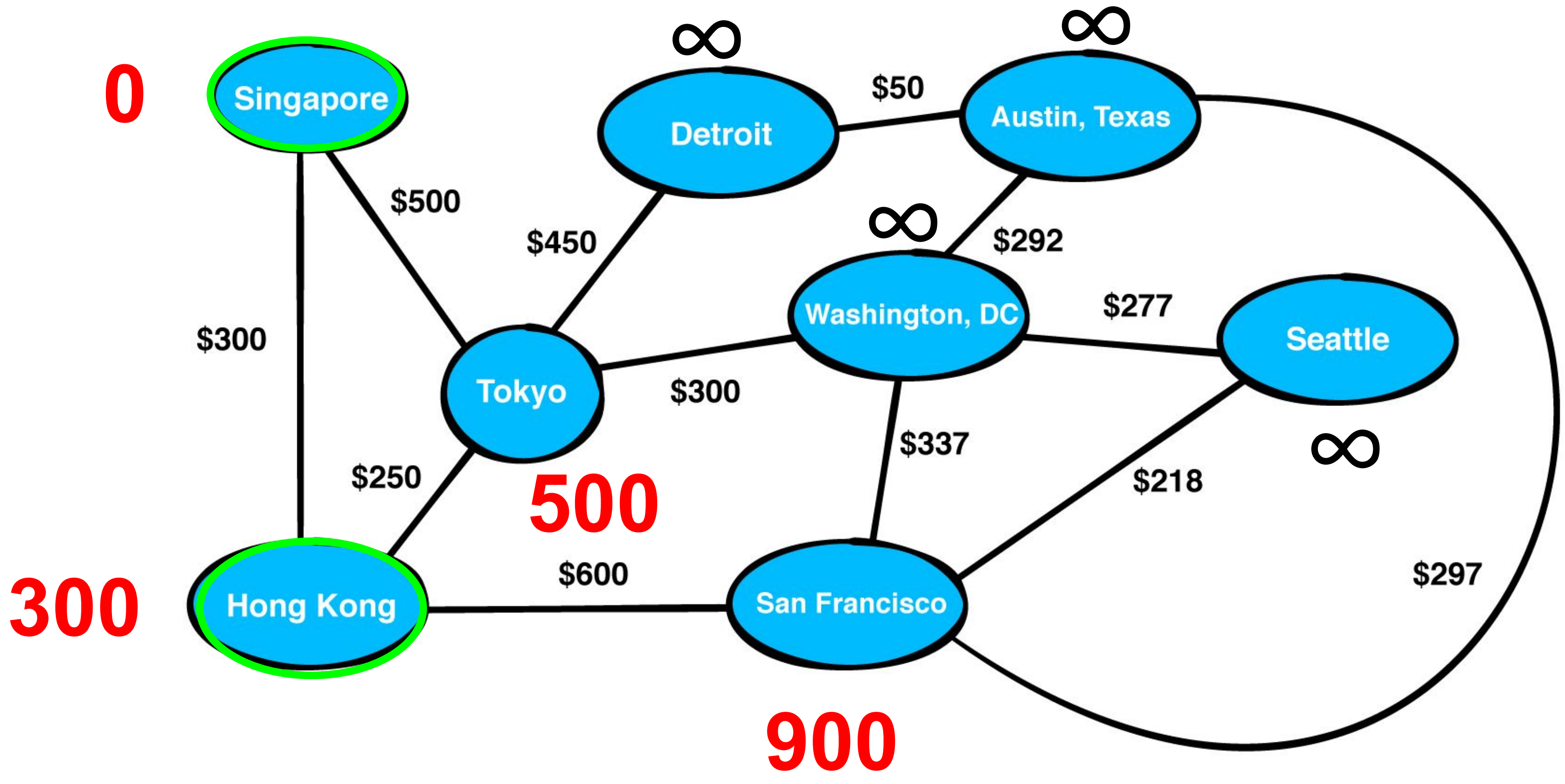
Graphs - Dijkstra Algorithm

We start with **Detroit**. Again we ask always the same questions: has Detroit been visited? **NO!**



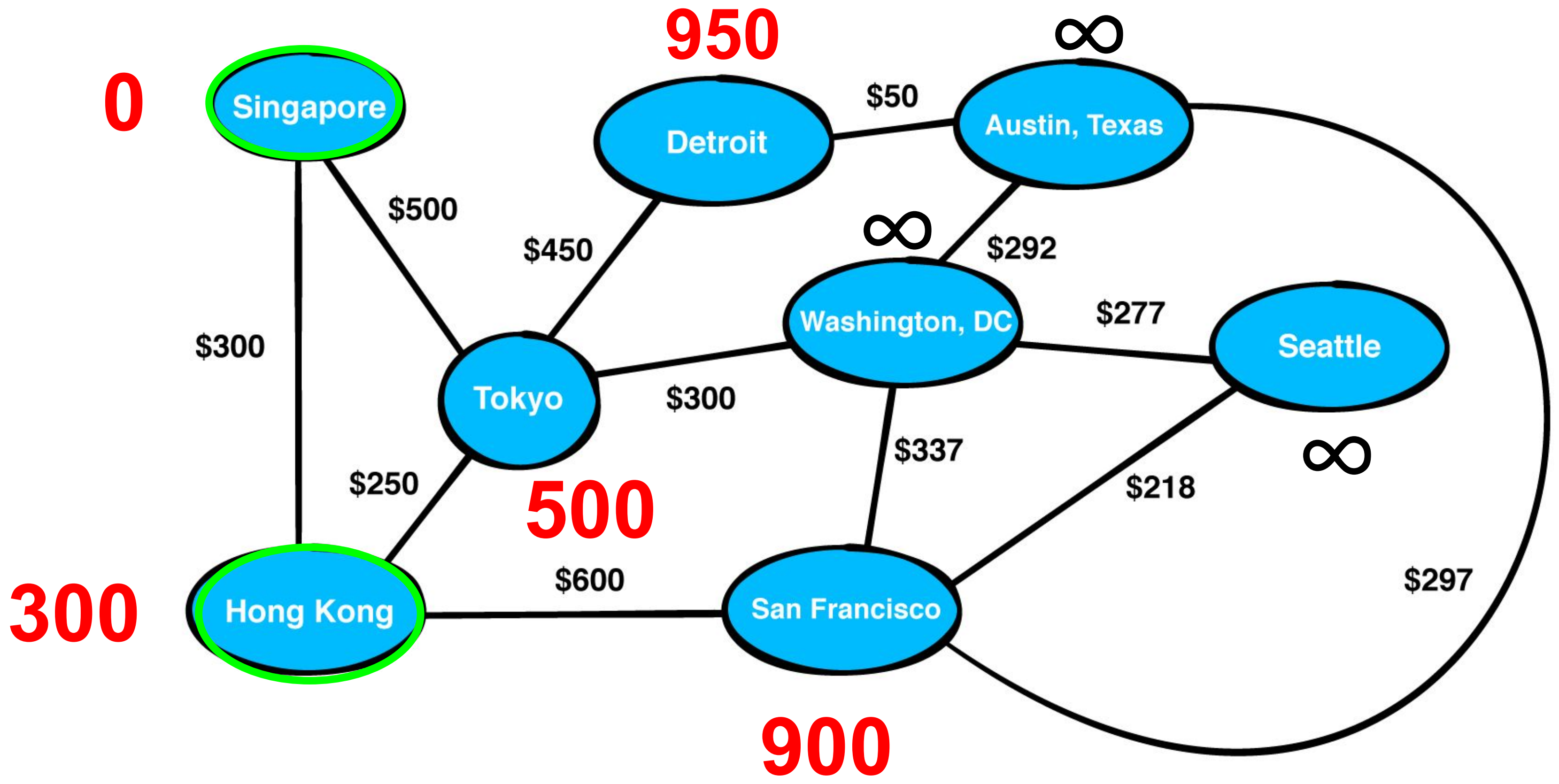
Graphs - Dijkstra Algorithm

Is the cost of **Detroit** smaller than the cost to reach **Tokyo** plus the cost to go from Tokyo to Detroit? **YES** because 500 + 450 < ∞ so we change the cost



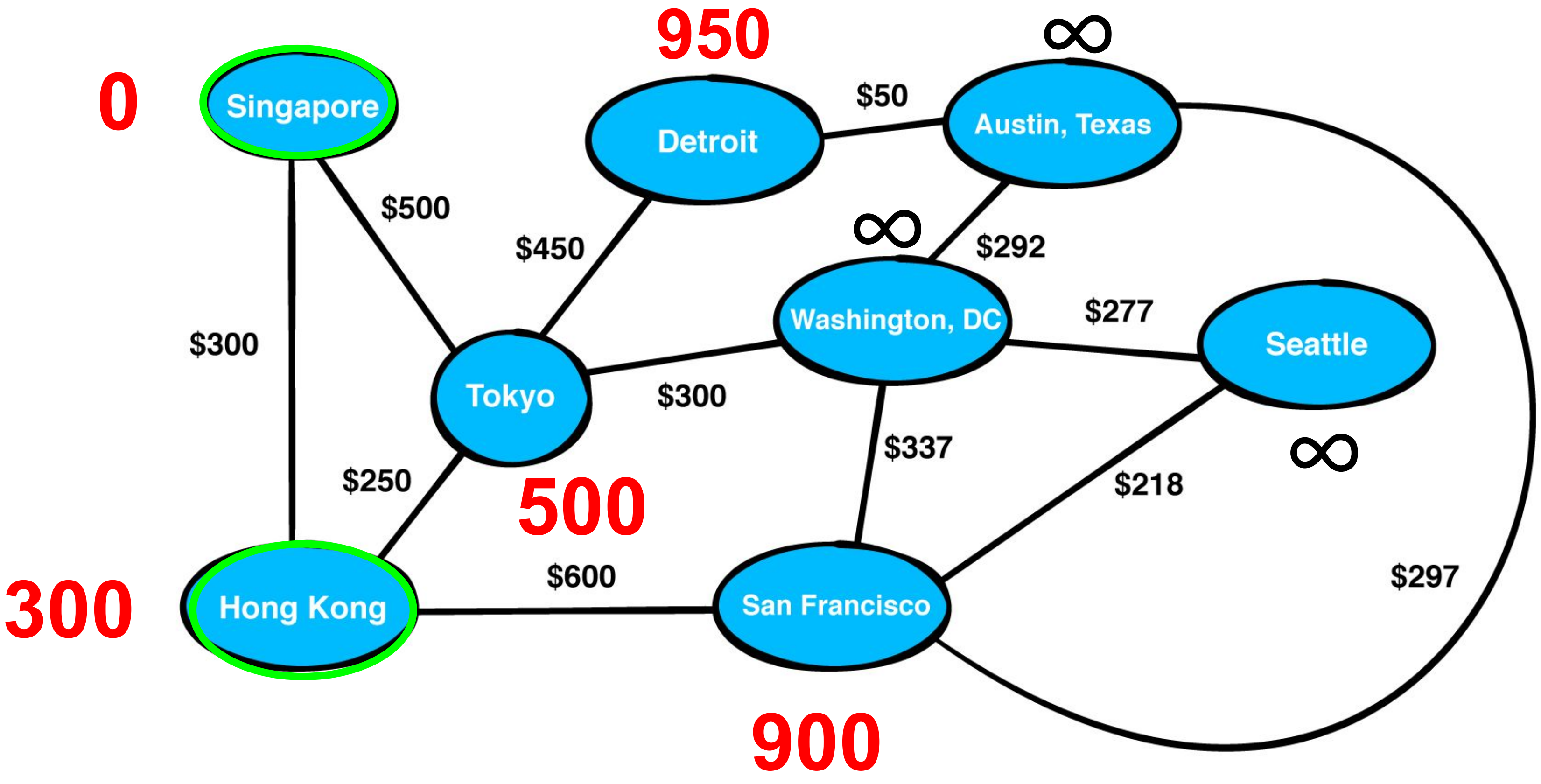
Graphs - Dijkstra Algorithm

Is the cost of **Detroit** smaller than the cost to reach **Tokyo** plus the cost to go from Tokyo to Detroit? **YES** because 500 + 450 < ∞ so we change the cost



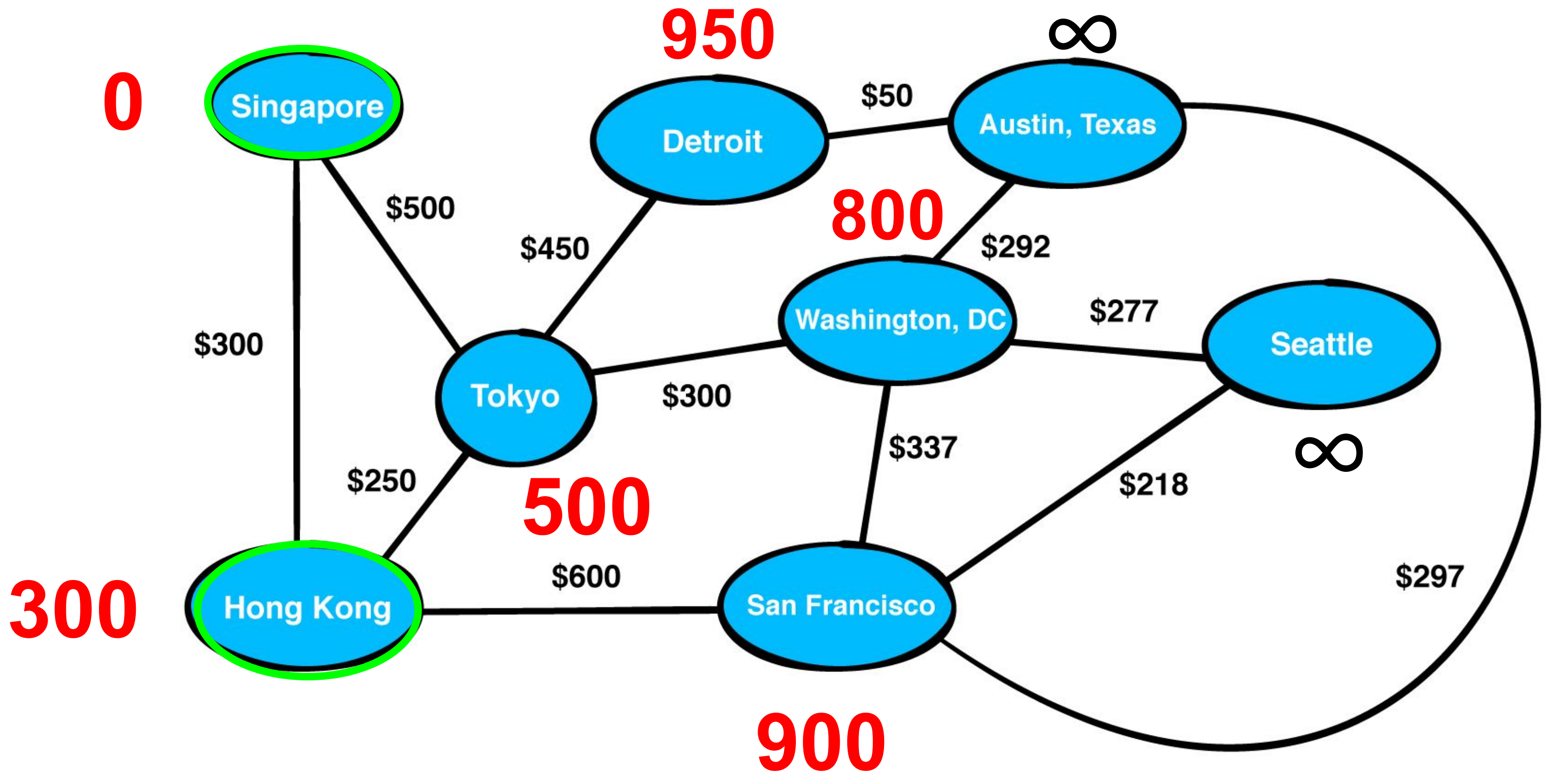
Graphs - Dijkstra Algorithm

Now again we ask the same question for **Washington** and we can see that the cost of **Washington** becomes 500 + 300 = 800



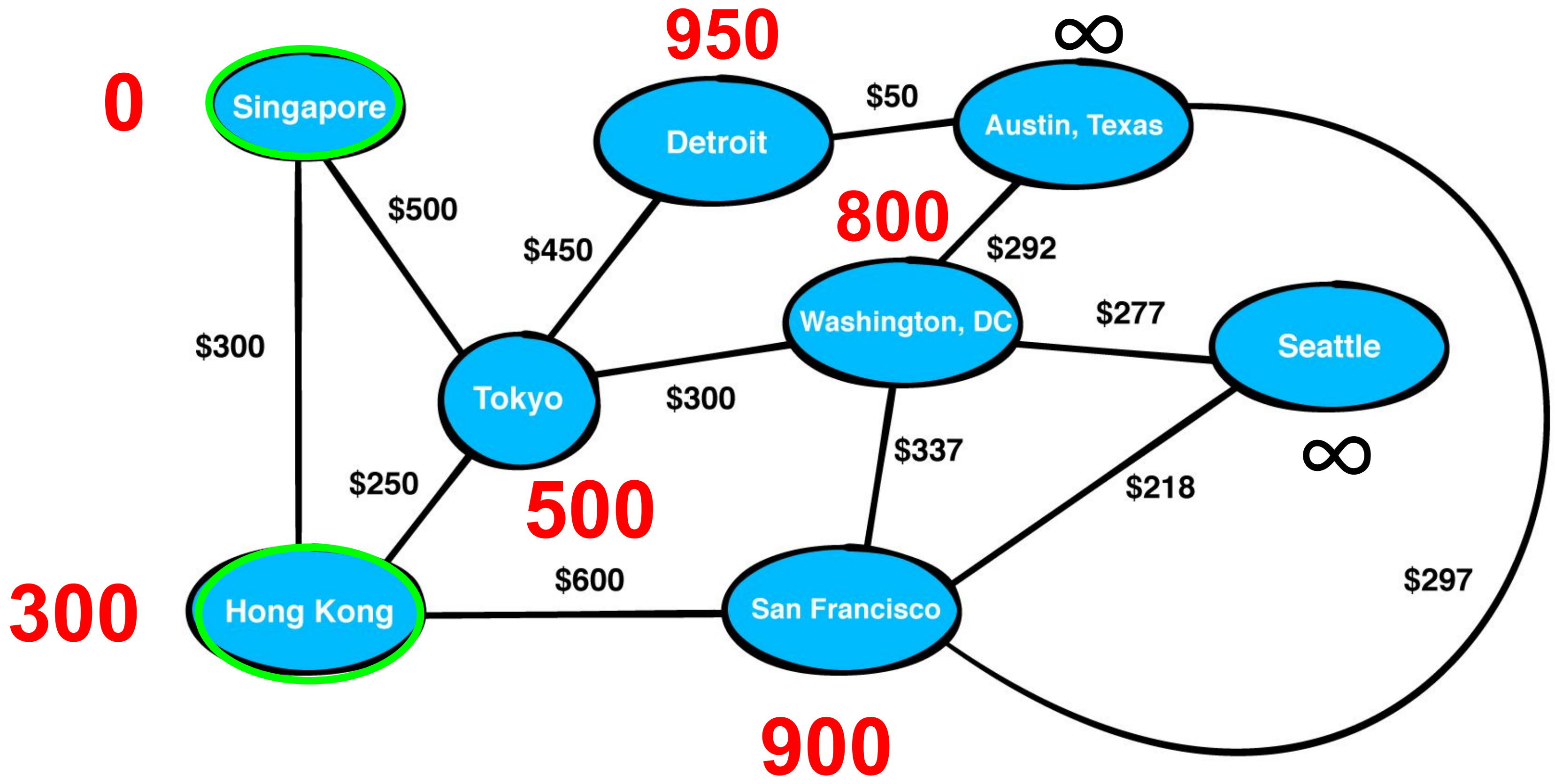
Graphs - Dijkstra Algorithm

Now again we ask the same question for **Washington** and we can see that the cost of **Washington** becomes 500 + 300 = 800



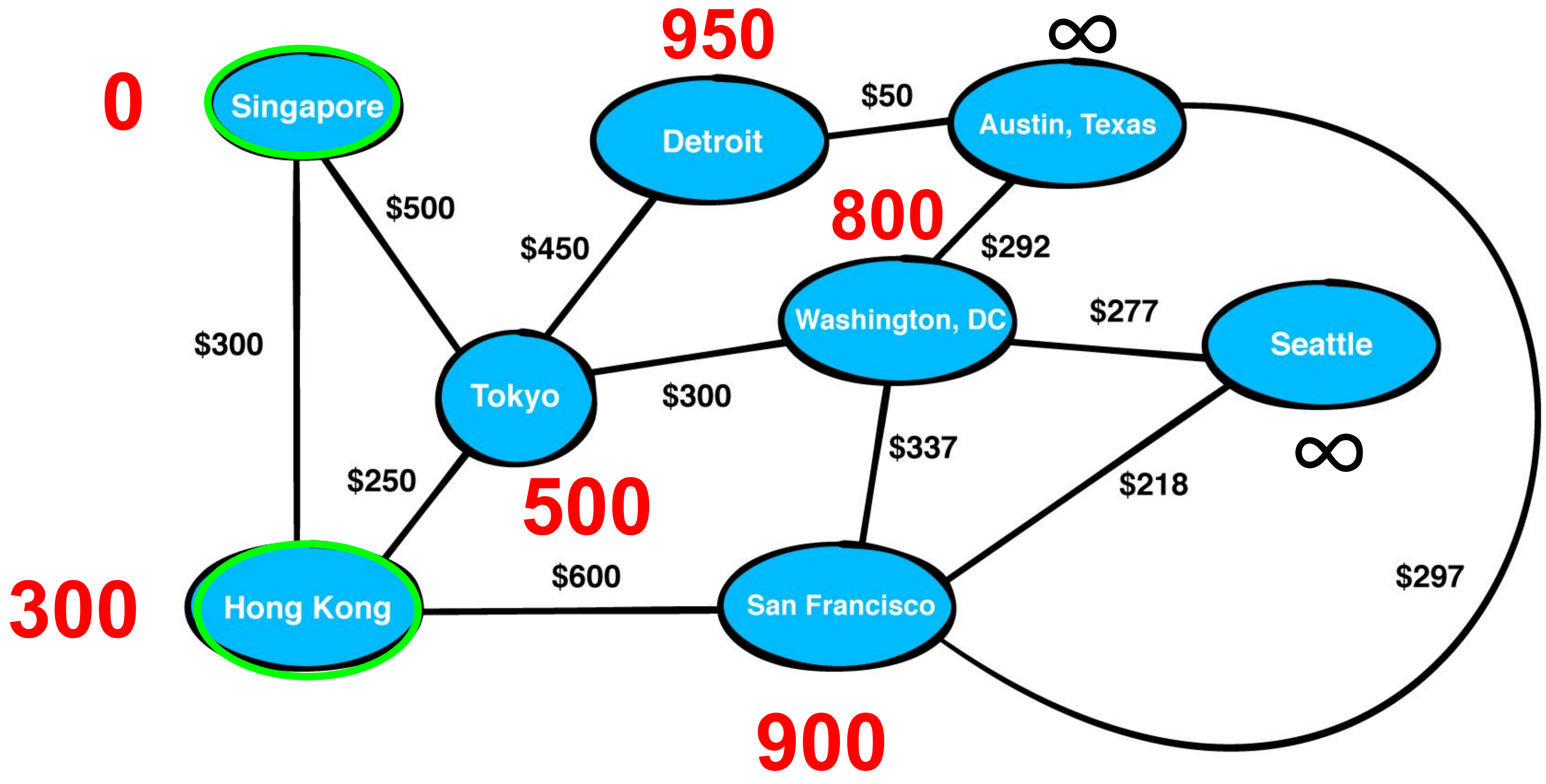
Graphs - Dijkstra Algorithm

REMARK: as you can see Tokyo is linked with Singapore and Hong Kong but we skip these nodes because they have been visited!



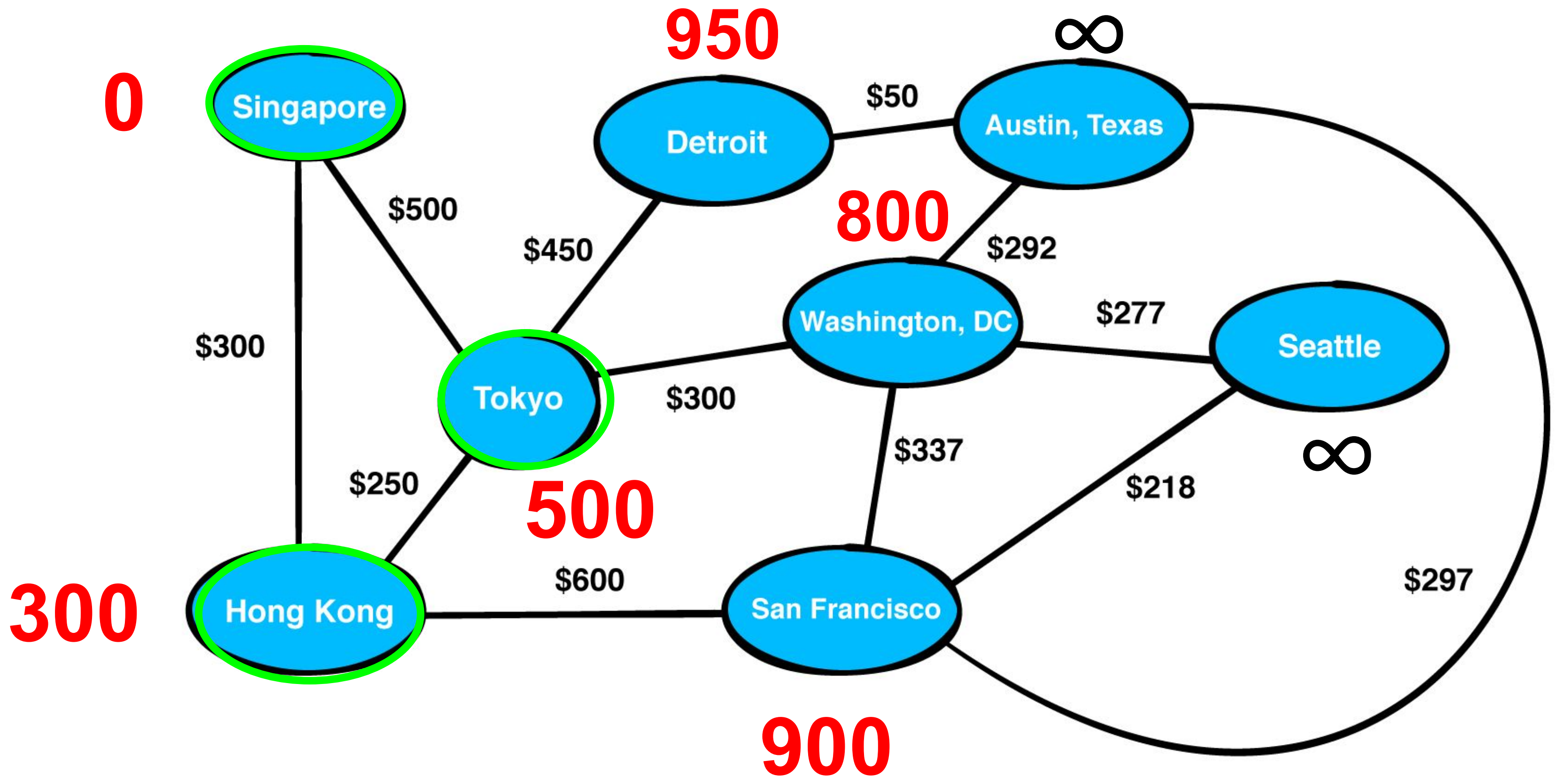
Graphs - Dijkstra Algorithm

Now since **Tokyo** has no other adjacent nodes we can set it as visited and we can select the adjacent node with the smallest cost.



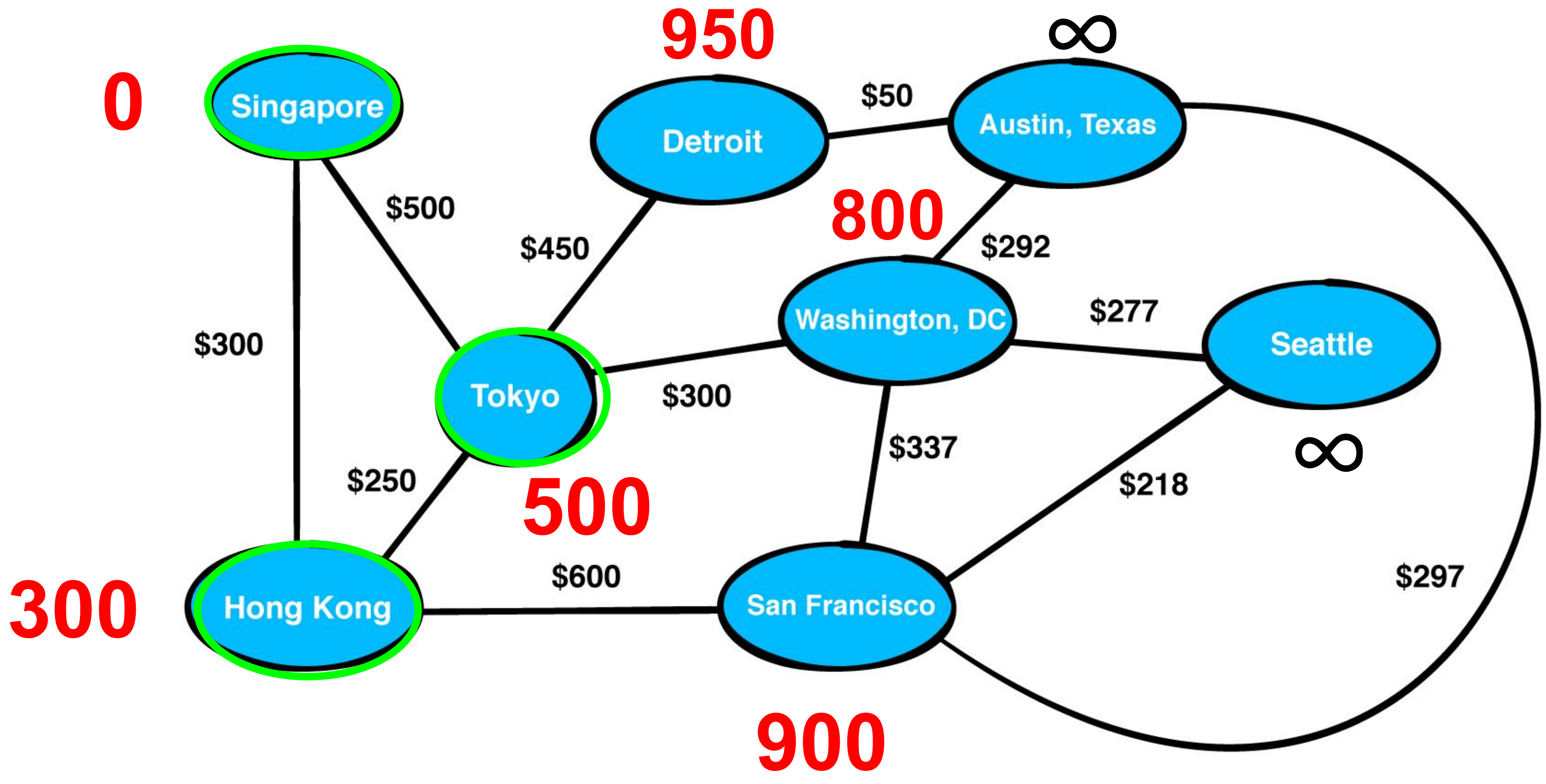
Graphs - Dijkstra Algorithm

Now since **Tokyo** has no other adjacent nodes we can set it as visited and we can select the adjacent node with the smallest cost.



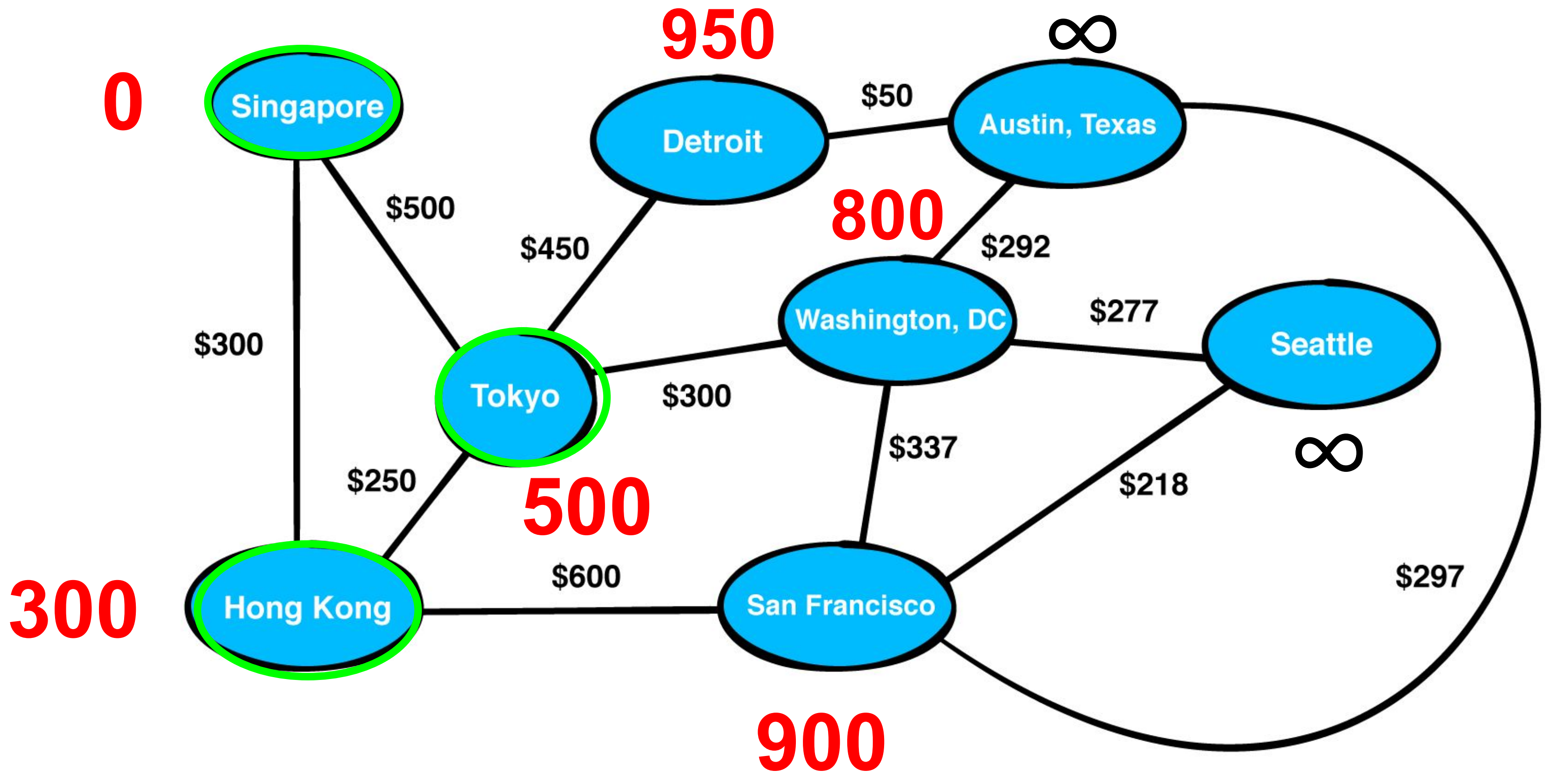
Graphs - Dijkstra Algorithm

The node is **Washington**, and we start exploring the adjacent nodes. We start from **Austin**. Again we ask the questions.



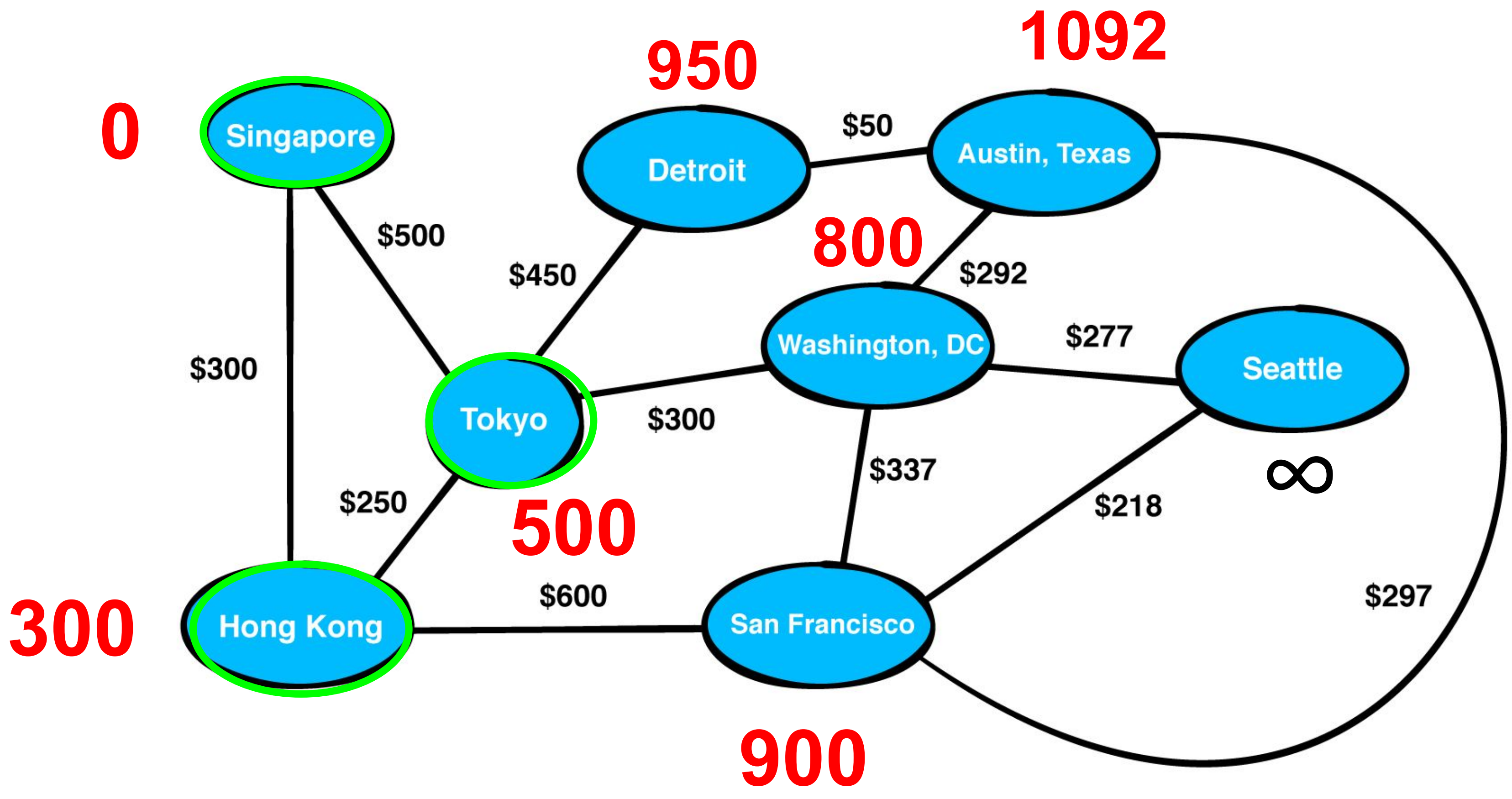
Graphs - Dijkstra Algorithm

Has Austin been visited? **NO**!, is the cost of Austin lower than the cost of Washington plus the cost of the link between Washington and Austin? **YES**



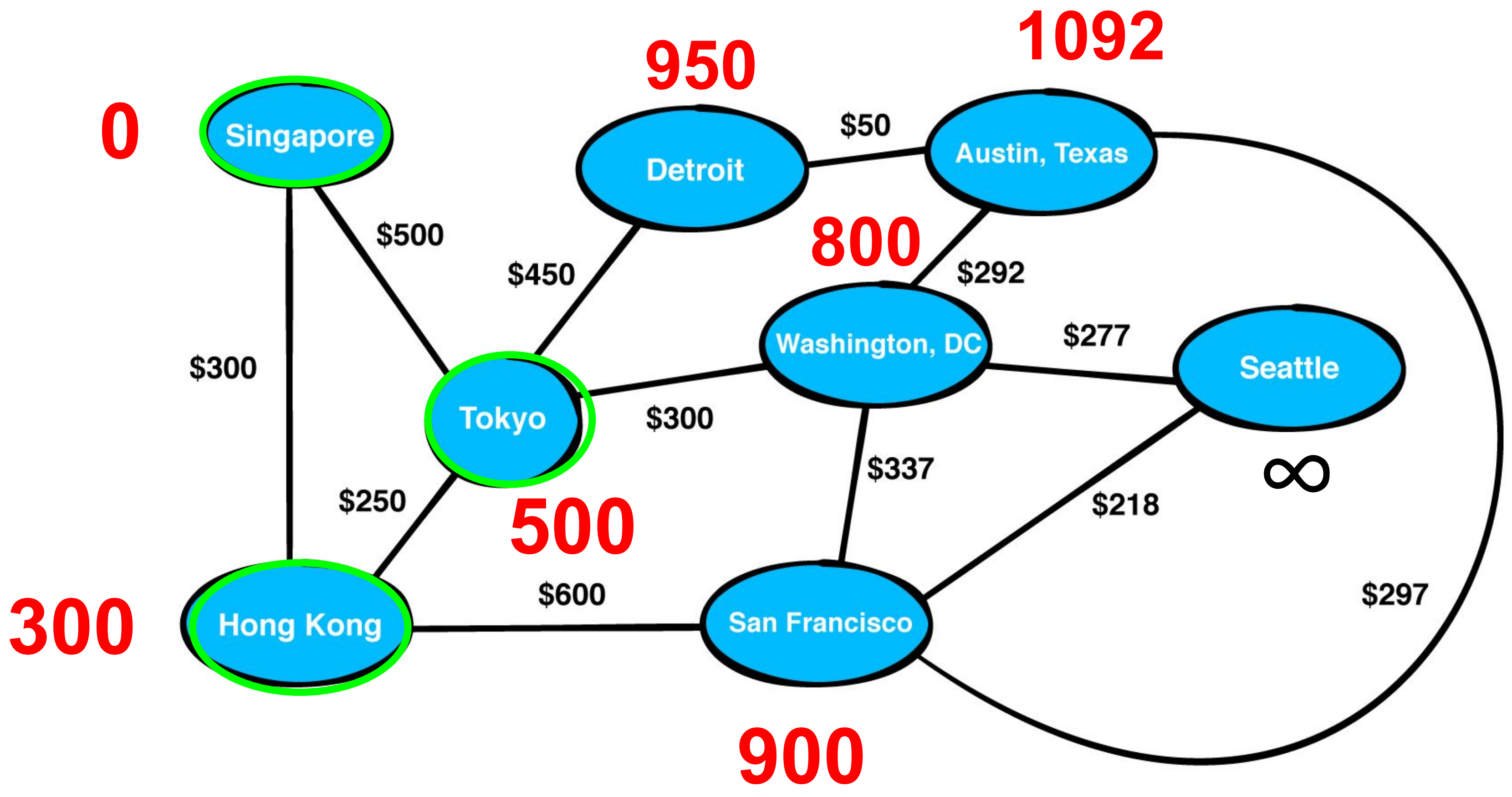
Graphs - Dijkstra Algorithm

So we change the cost of Austin in 800 + 292 = 1092



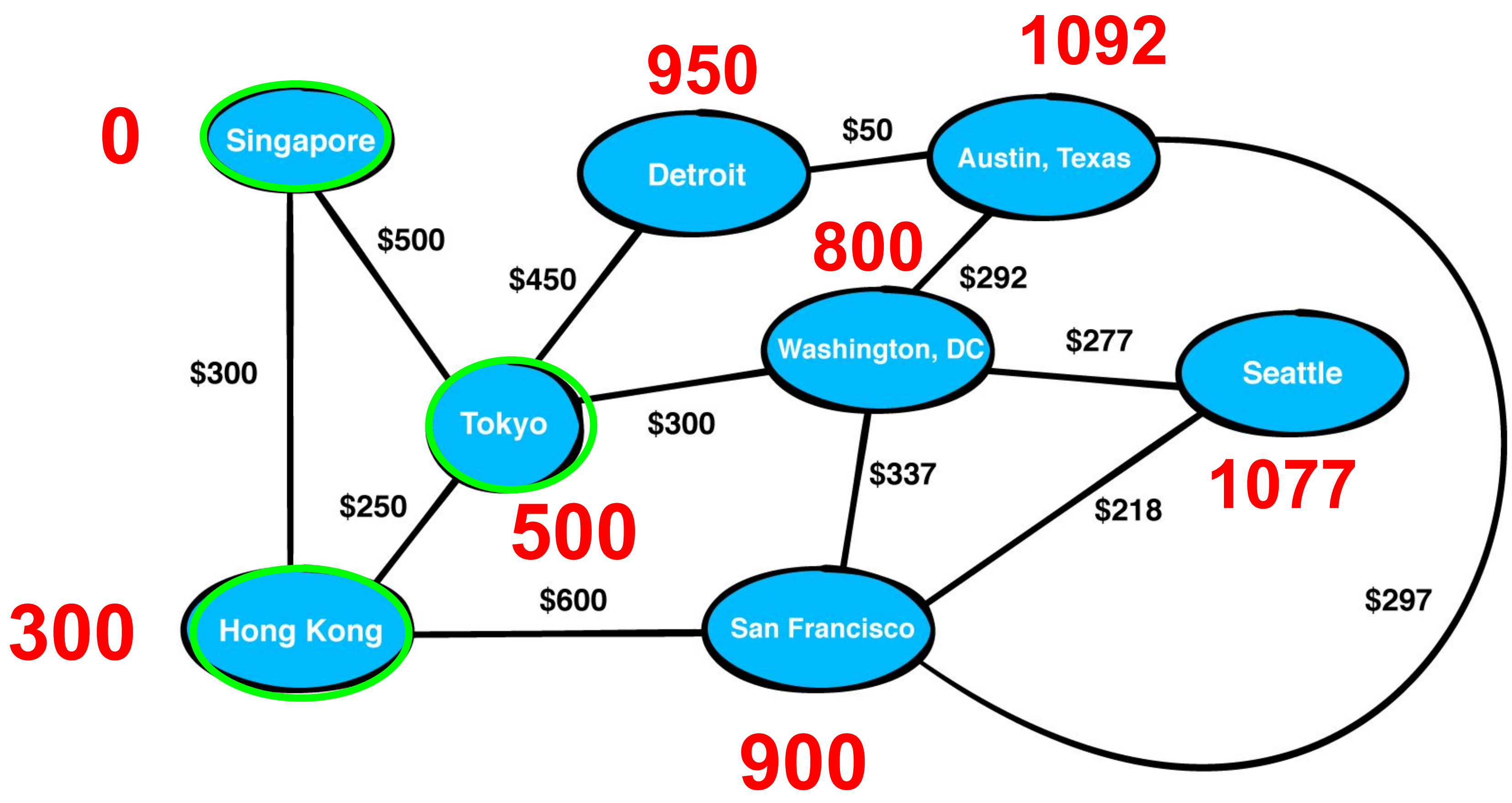
Graphs - Dijkstra Algorithm

Same thing happen to **Seattle**. **Please tell which is the cost of Seattle**



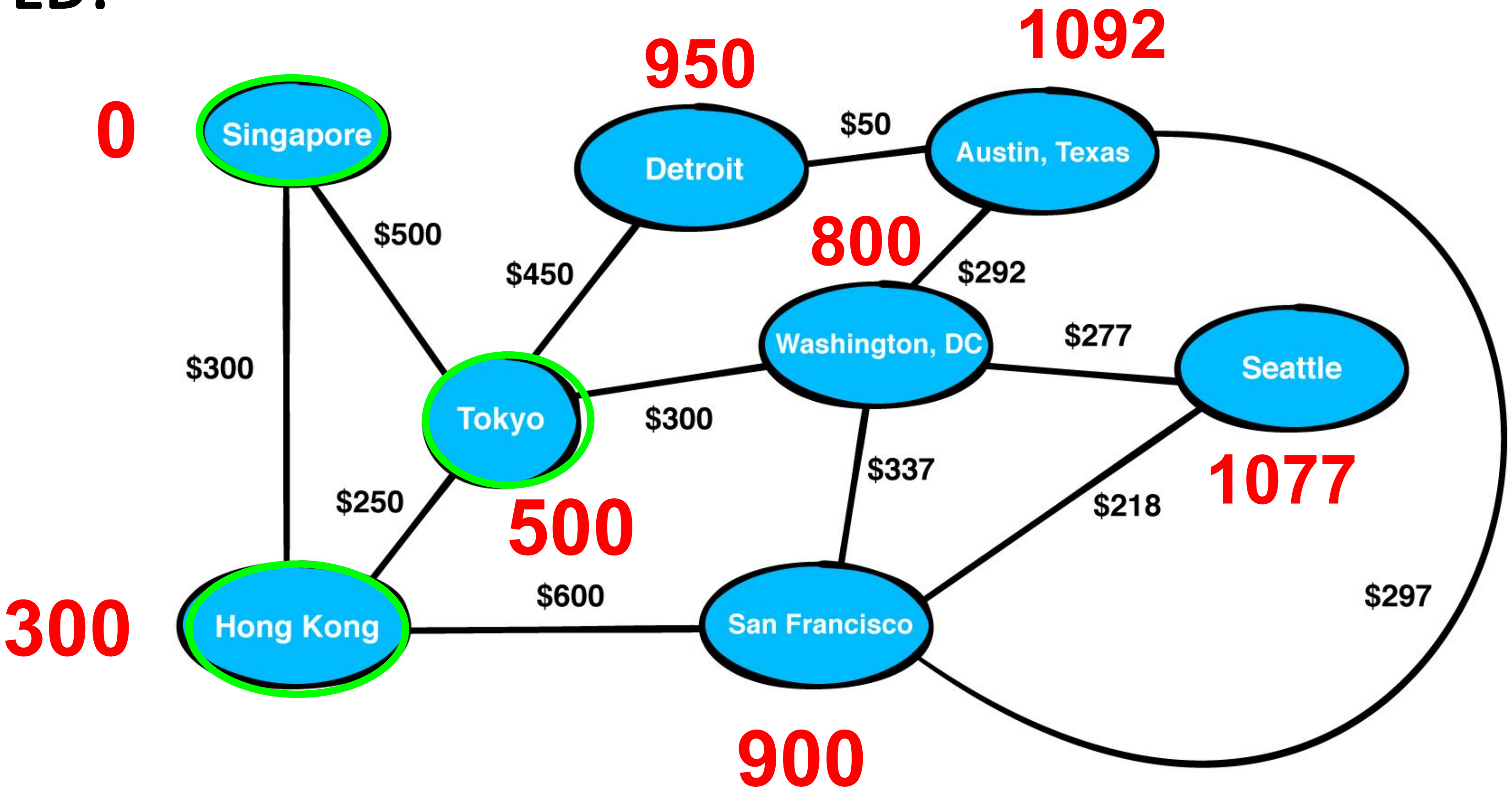
Graphs - Dijkstra Algorithm

The cost of Seattle is 800 + 277 = 1077



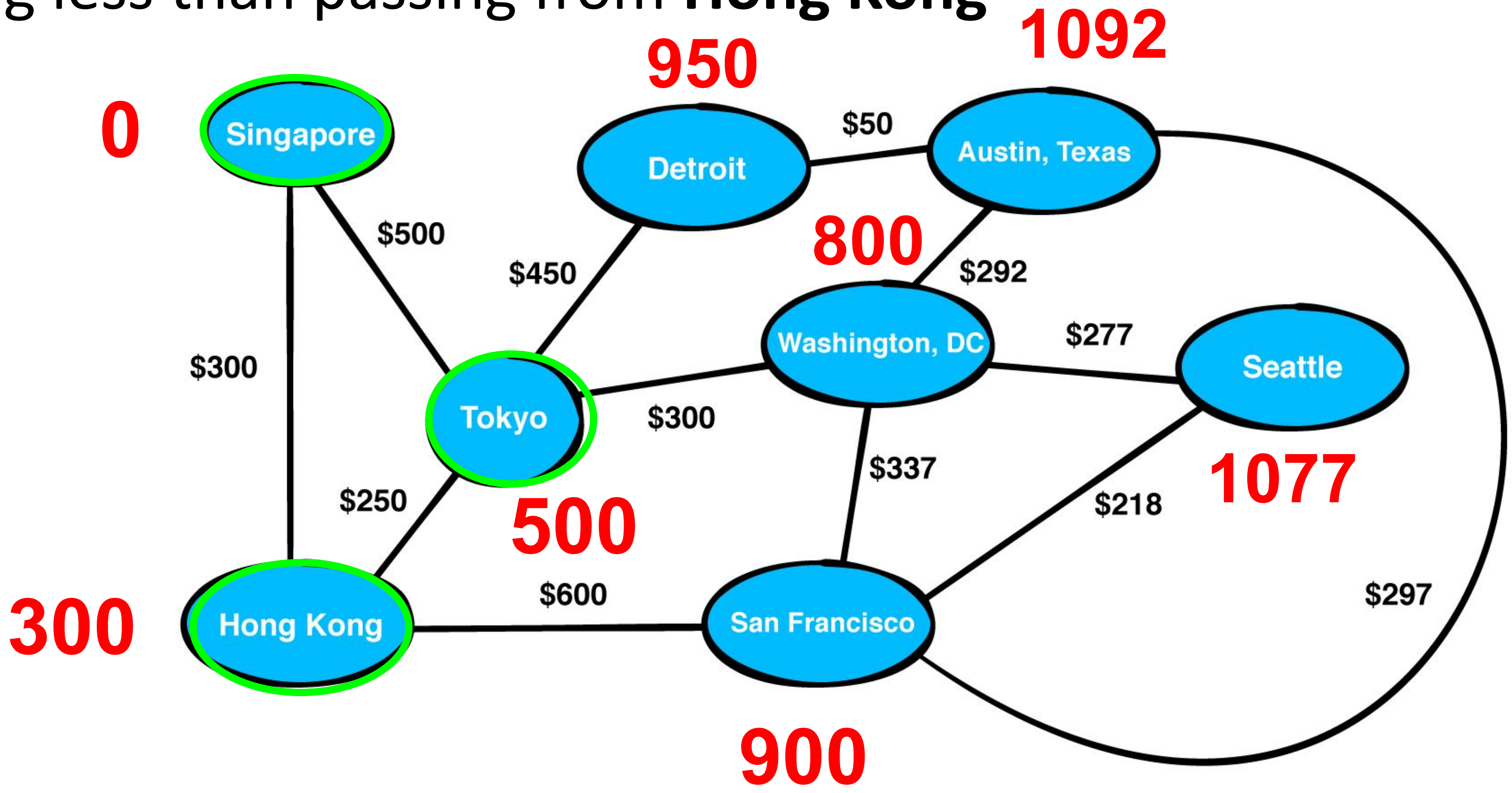
Graphs - Dijkstra Algorithm

We again explore **San Francisco** that has been explored **BUT NOT MARKED AS VISITED!**



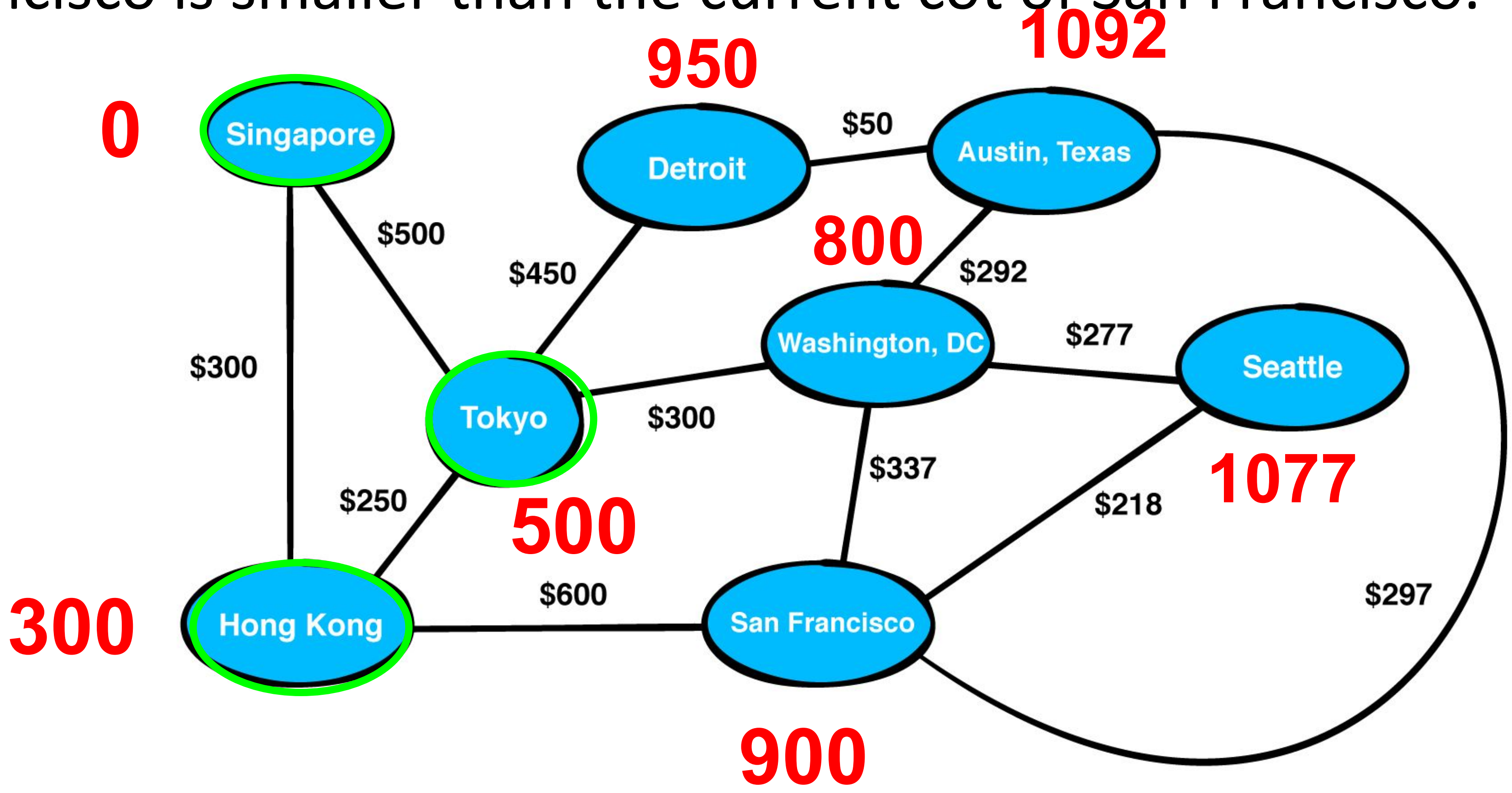
Graphs - Dijkstra Algorithm

Now we try to see if it is possible to reach **San Francisco** though **Washington** spending less than passing from **Hong Kong**



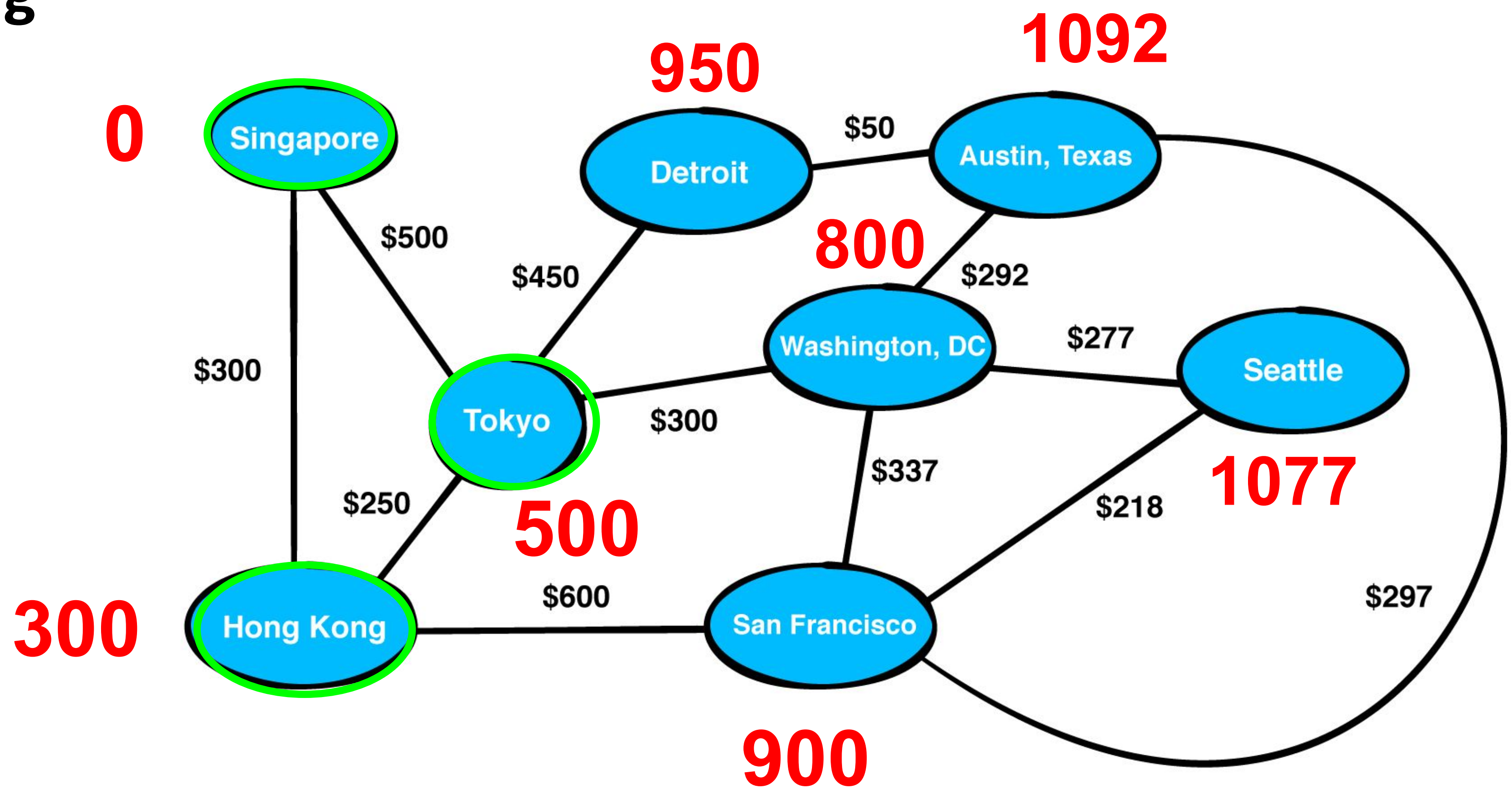
Graphs - Dijkstra Algorithm

We check if the cost of Washington plus the cost to go from Washington to San Francisco is smaller than the current cot of San Francisco.



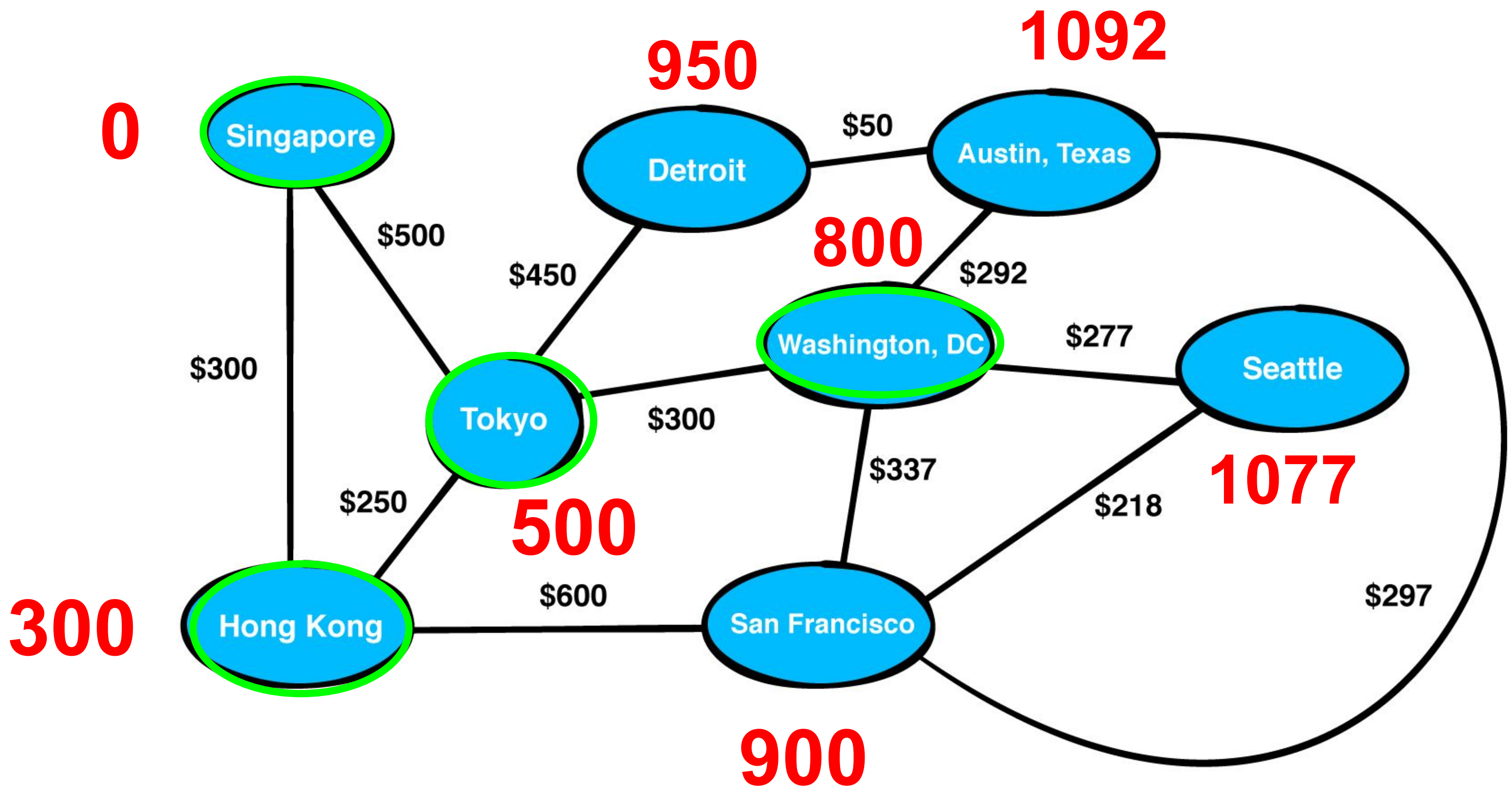
Graphs - Dijkstra Algorithm

800 + 337 = 1137 > 900 so it is not convenient and we **do not change anything**



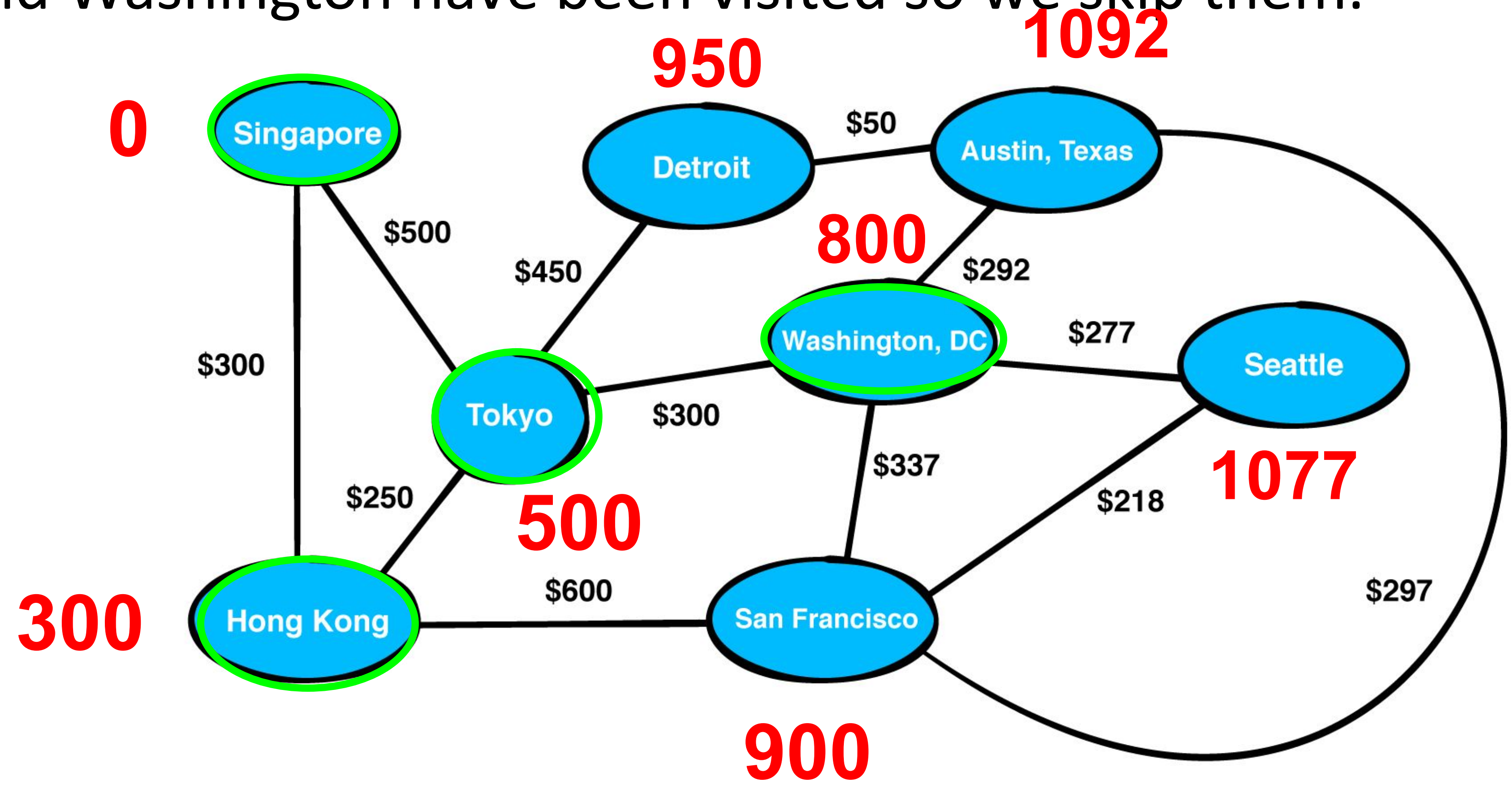
Graphs - Dijkstra Algorithm

We set Washington as visited



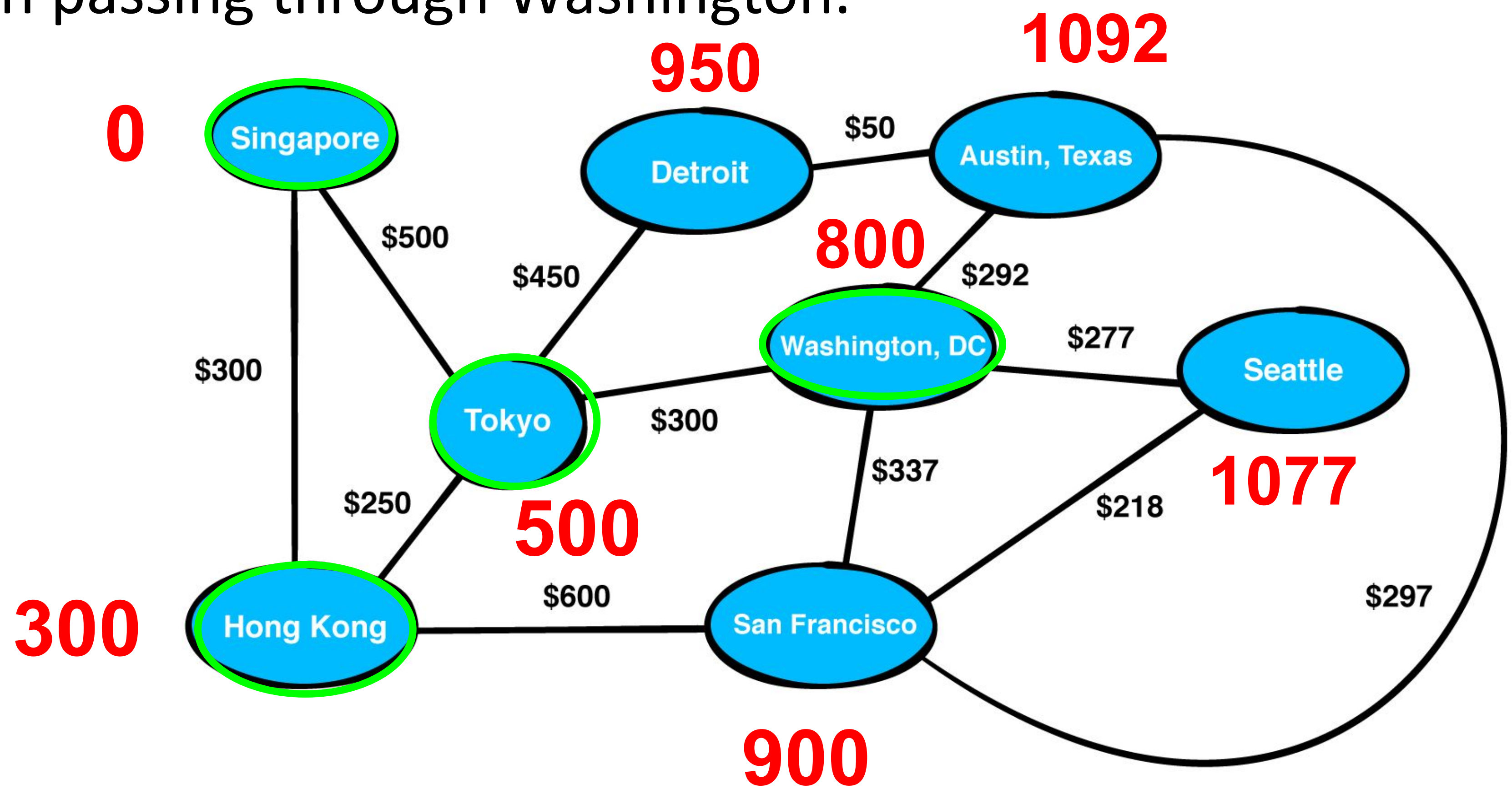
Graphs - Dijkstra Algorithm

We choose the node with the smallest cost that is San Francisco. Both Hong Kong and Washington have been visited so we skip them.



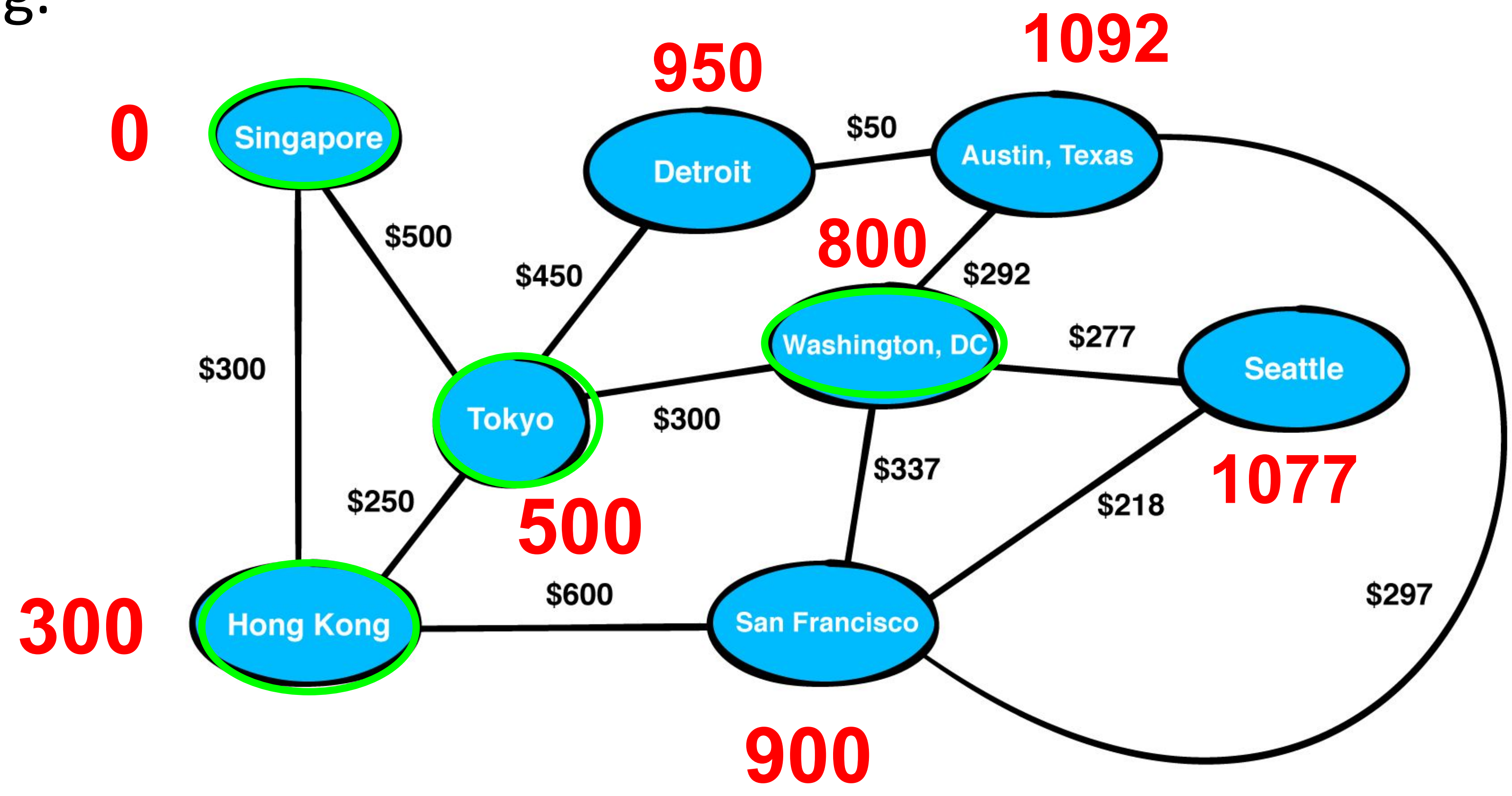
Graphs - Dijkstra Algorithm

We check is it is possible to reach Seattle through San Francisco spending less than passing through Washington.



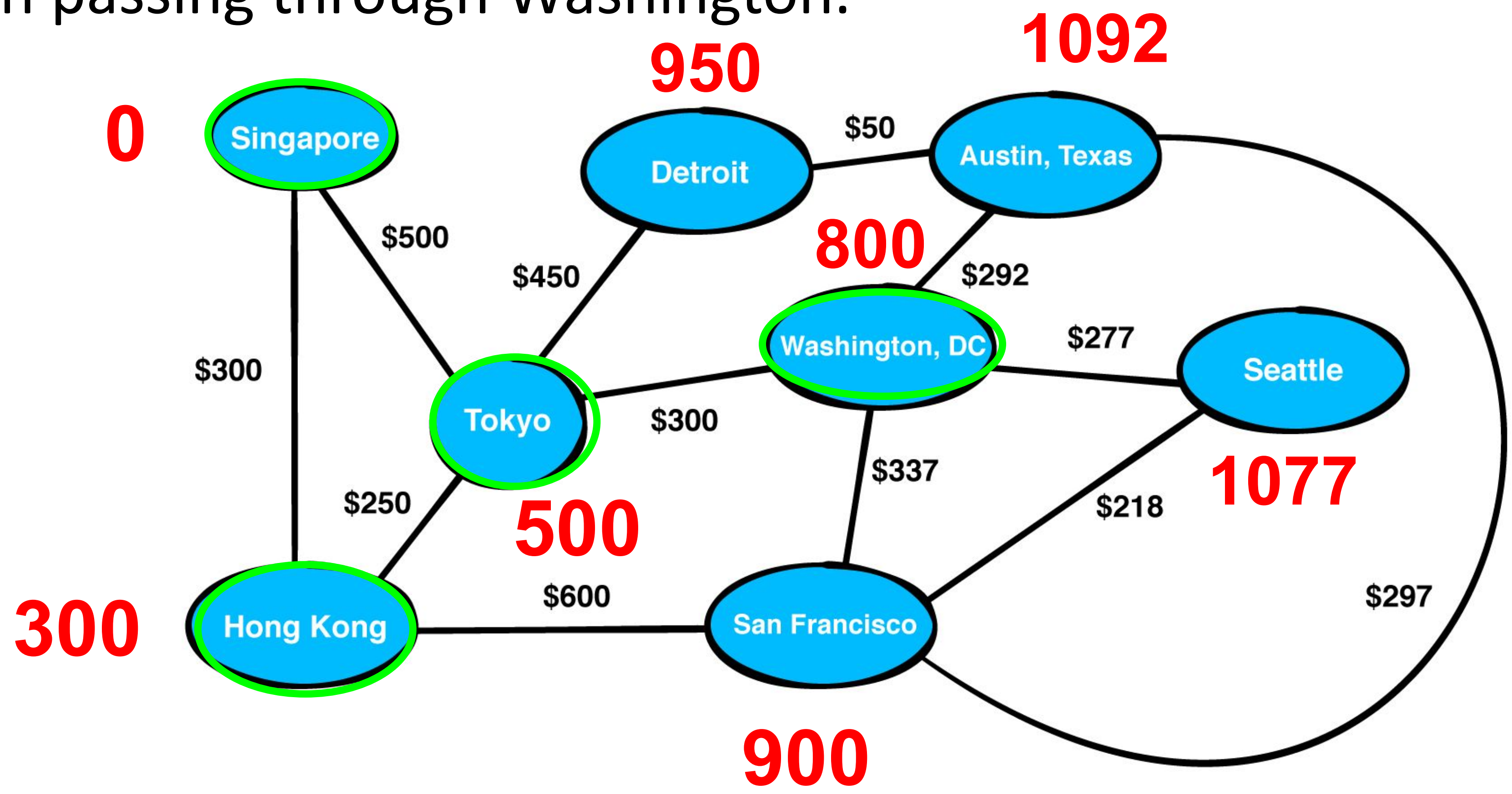
Graphs - Dijkstra Algorithm

The cost of Seattle is 1077 < 900 + 218 = 1118 so we do not change anything.



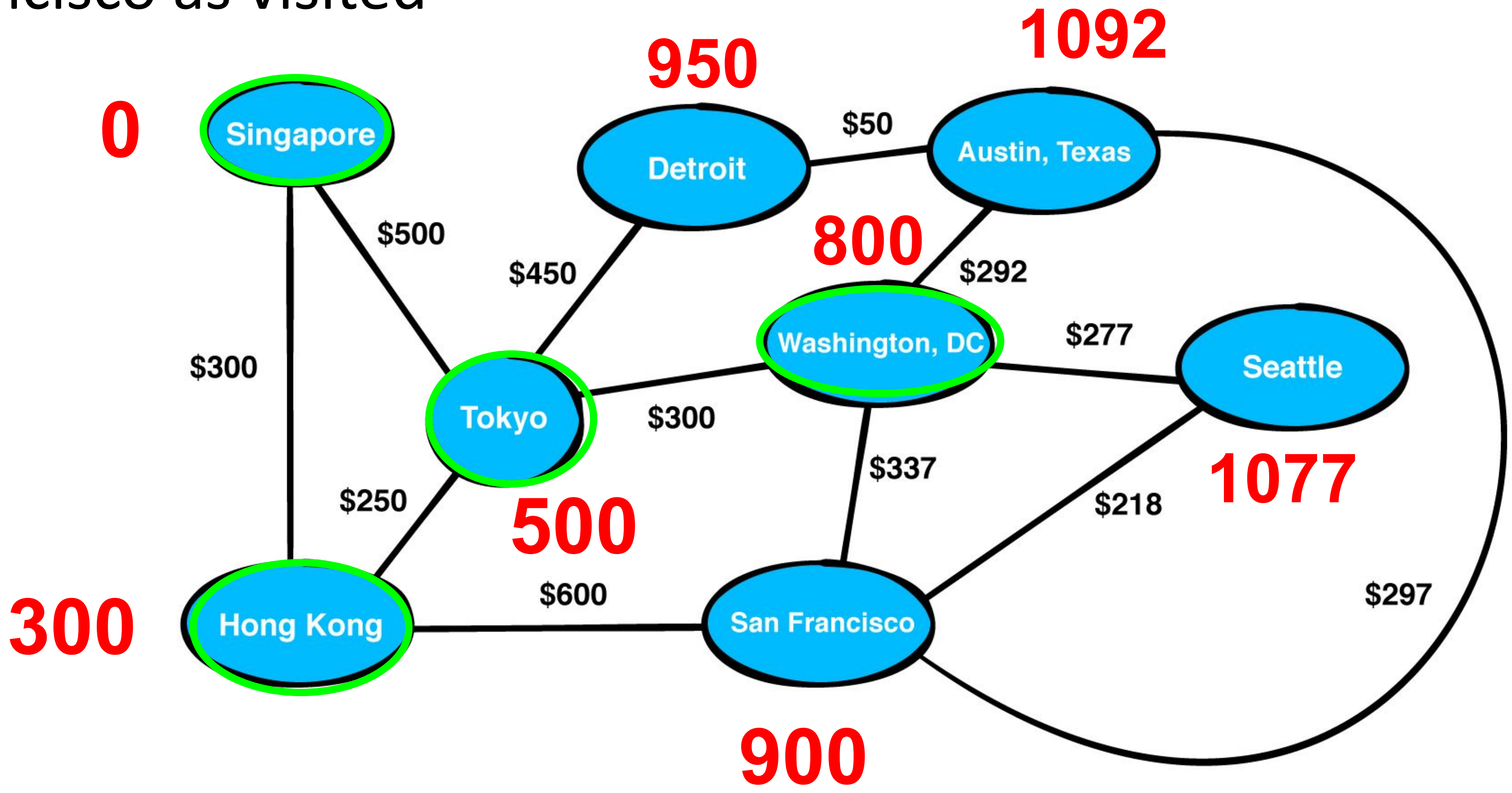
Graphs - Dijkstra Algorithm

Now we check if it is possible to reach Austin from San Francisco spending less than passing through Washington.



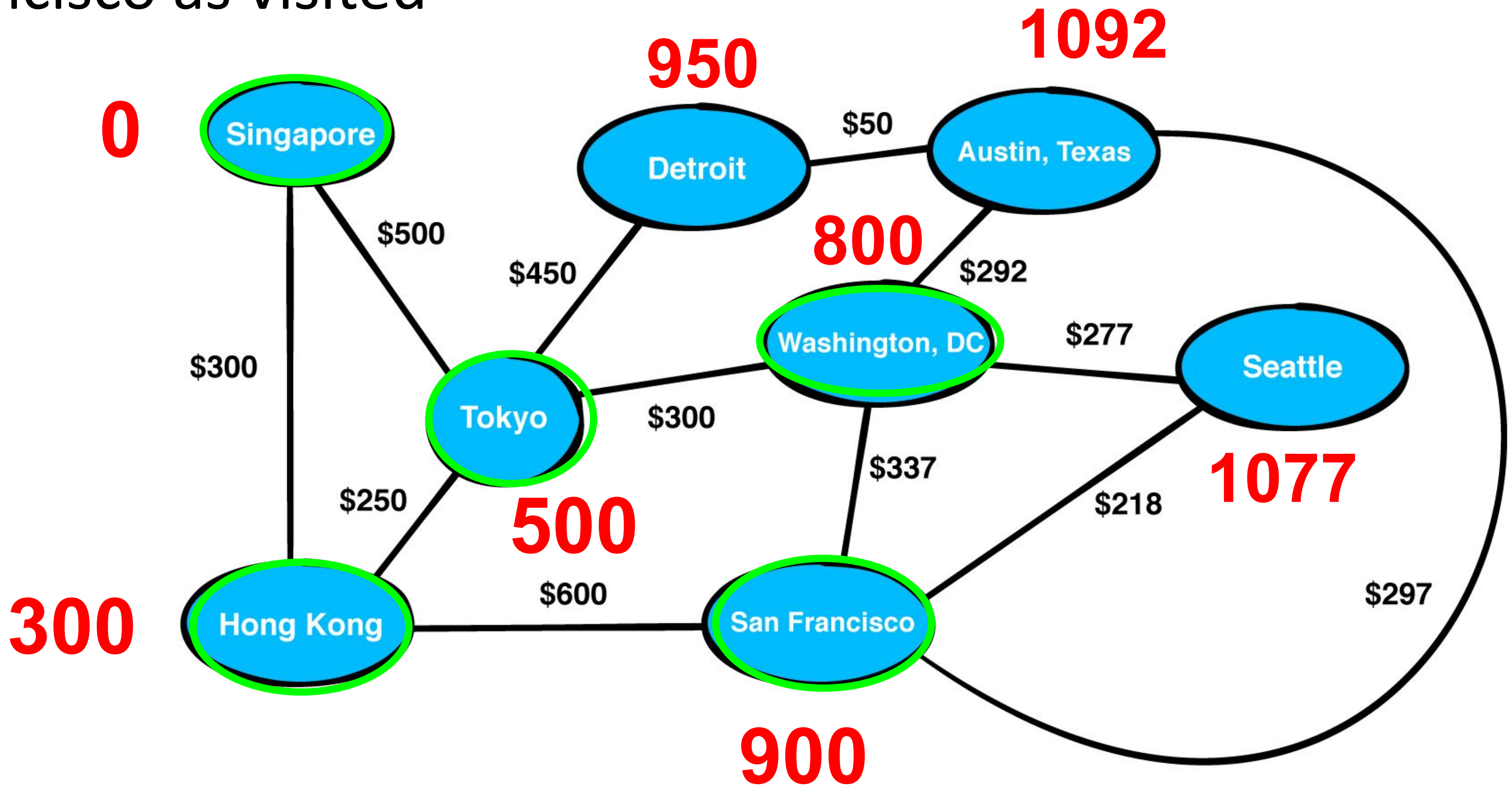
Graphs - Dijkstra Algorithm

1092 < 900 + 297 = 1197 So even here we do not change anything. We set San Francisco as visited



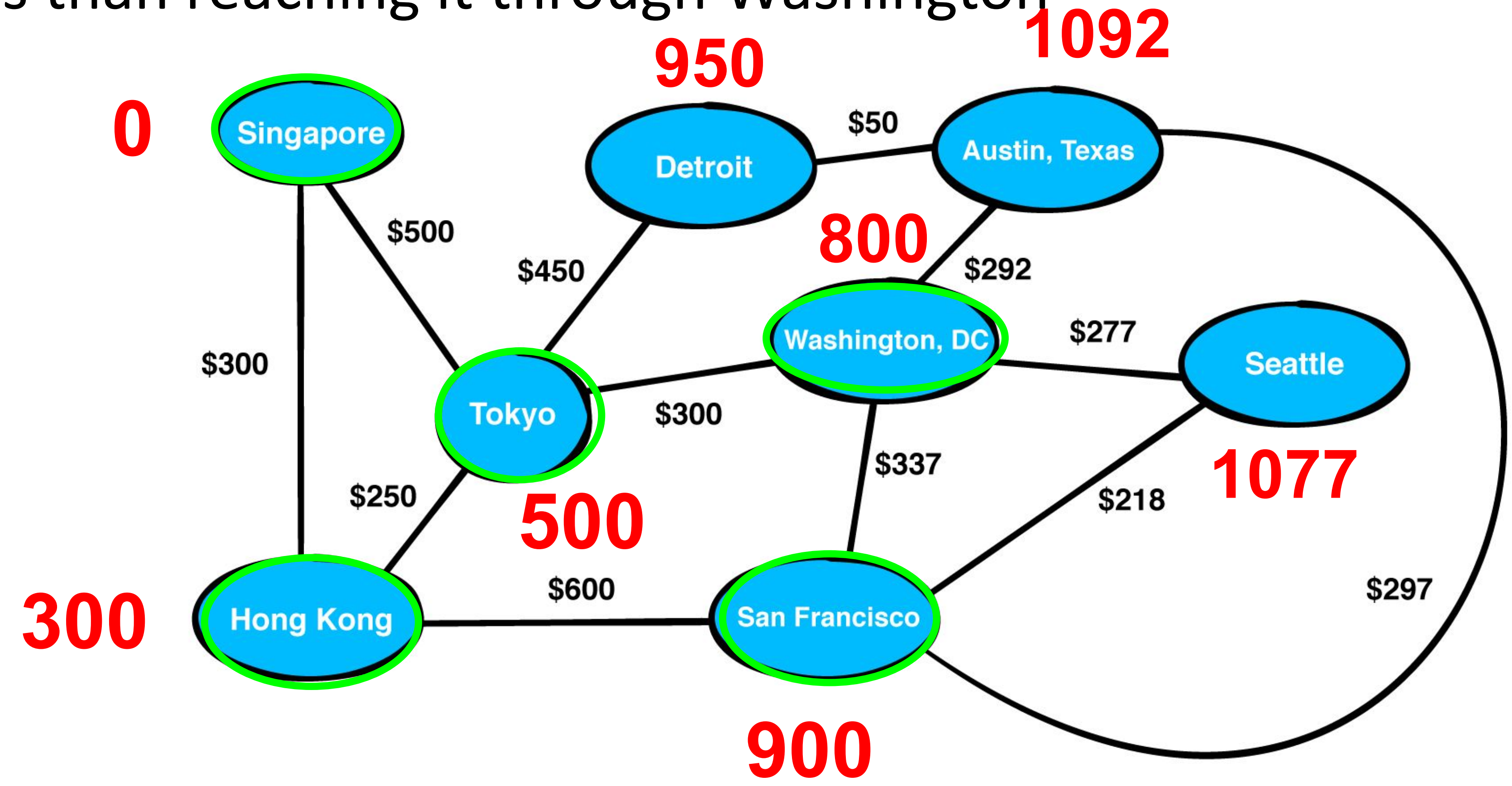
Graphs - Dijkstra Algorithm

1092 < 900 + 297 = 1197 So even here we do not change anything. We set San Francisco as visited



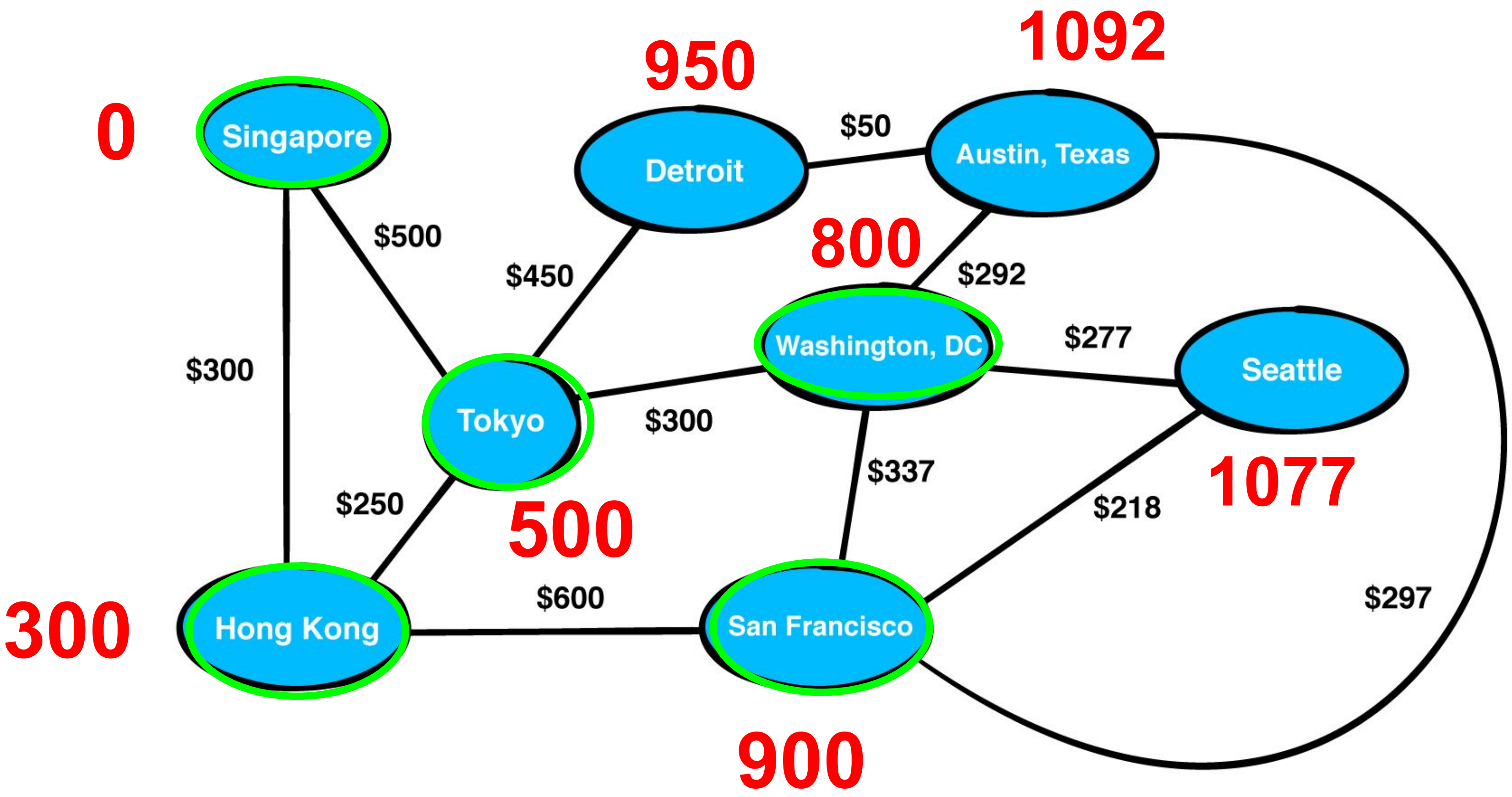
Graphs - Dijkstra Algorithm

Now we go to Detroit and we try to see if reaching Austin through Detroit cost less than reaching it through Washington



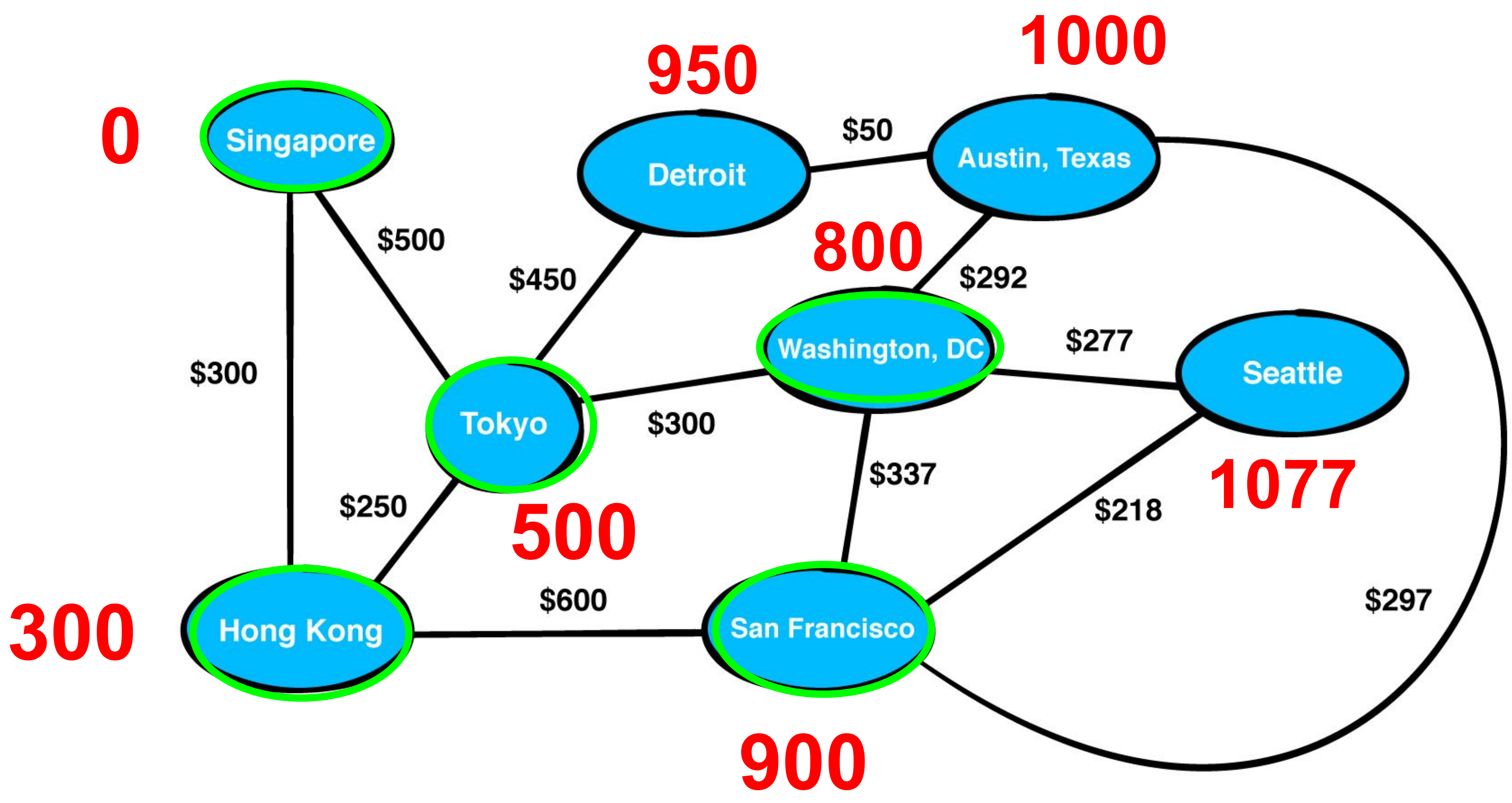
Graphs - Dijkstra Algorithm

1092 > 950 + 50 = 1000 **So YES it cost less to reach Austin from Detroit!!**



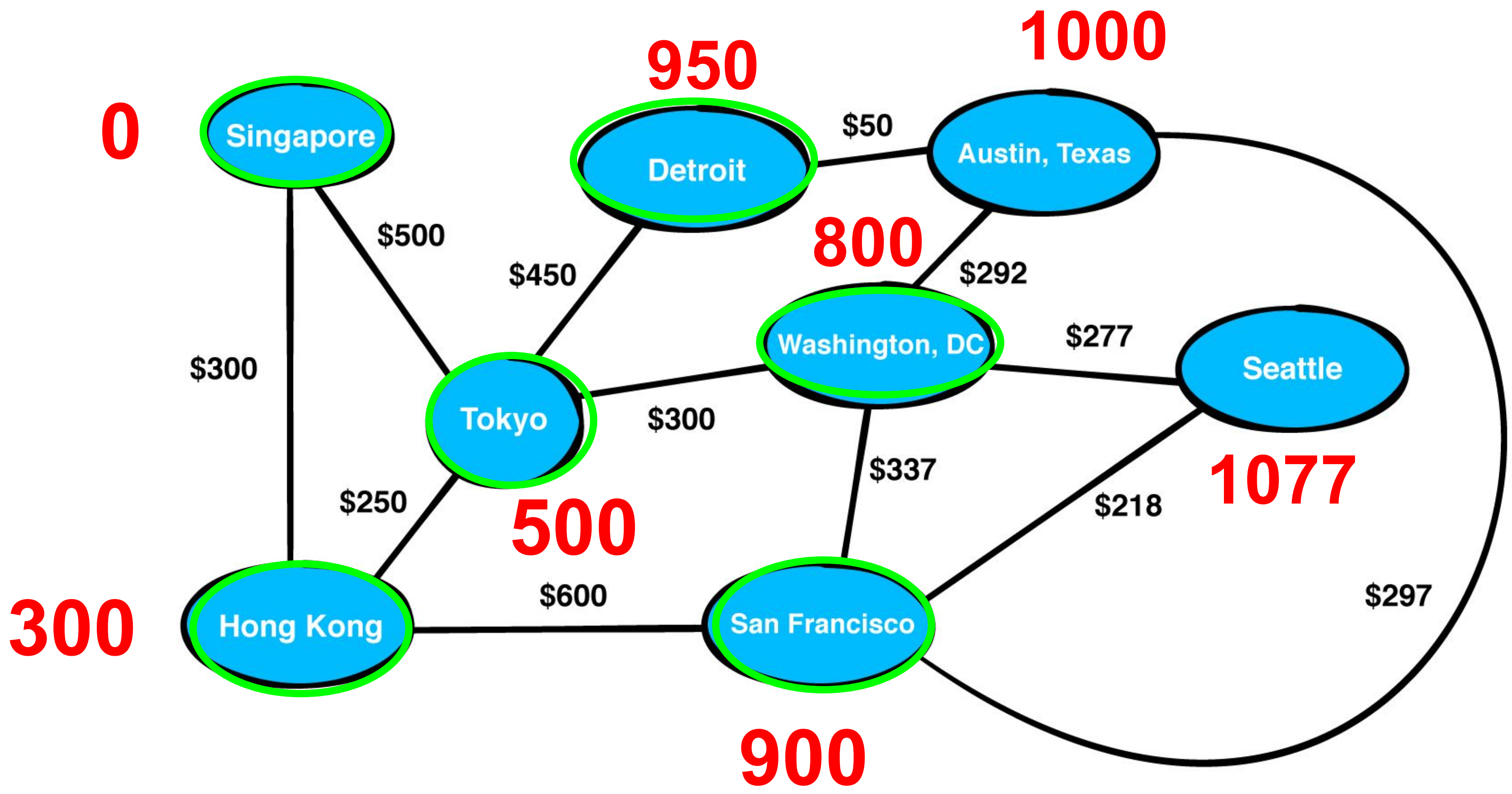
Graphs - Dijkstra Algorithm

Now we can change the cost of Austin to 1000



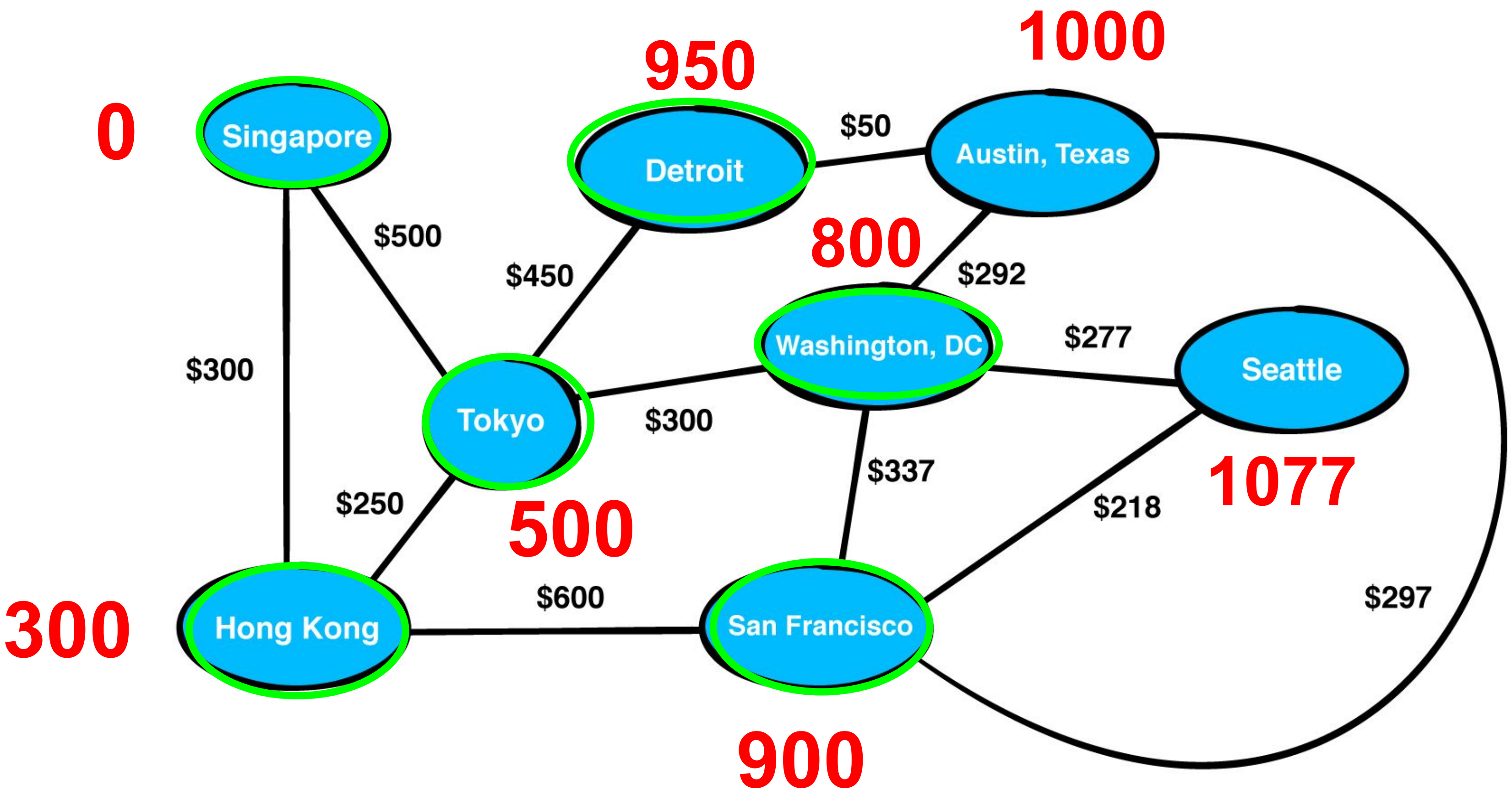
Graphs - Dijkstra Algorithm

Since Tokyo has been visited we set Detroit has visited and we go to Austin



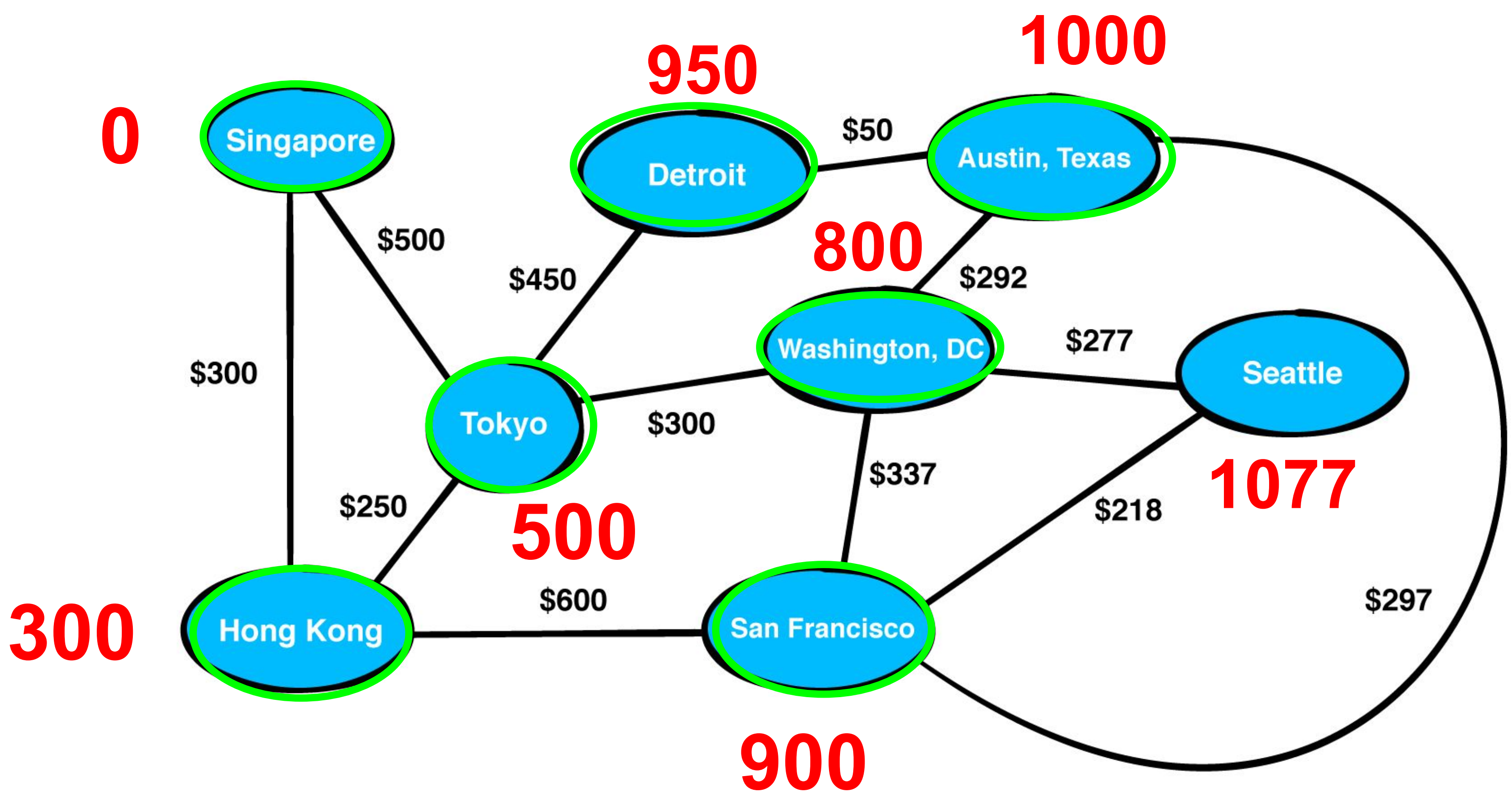
Graphs - Dijkstra Algorithm

Since every adjacent nodes of Austin has been visited we set it has visited!



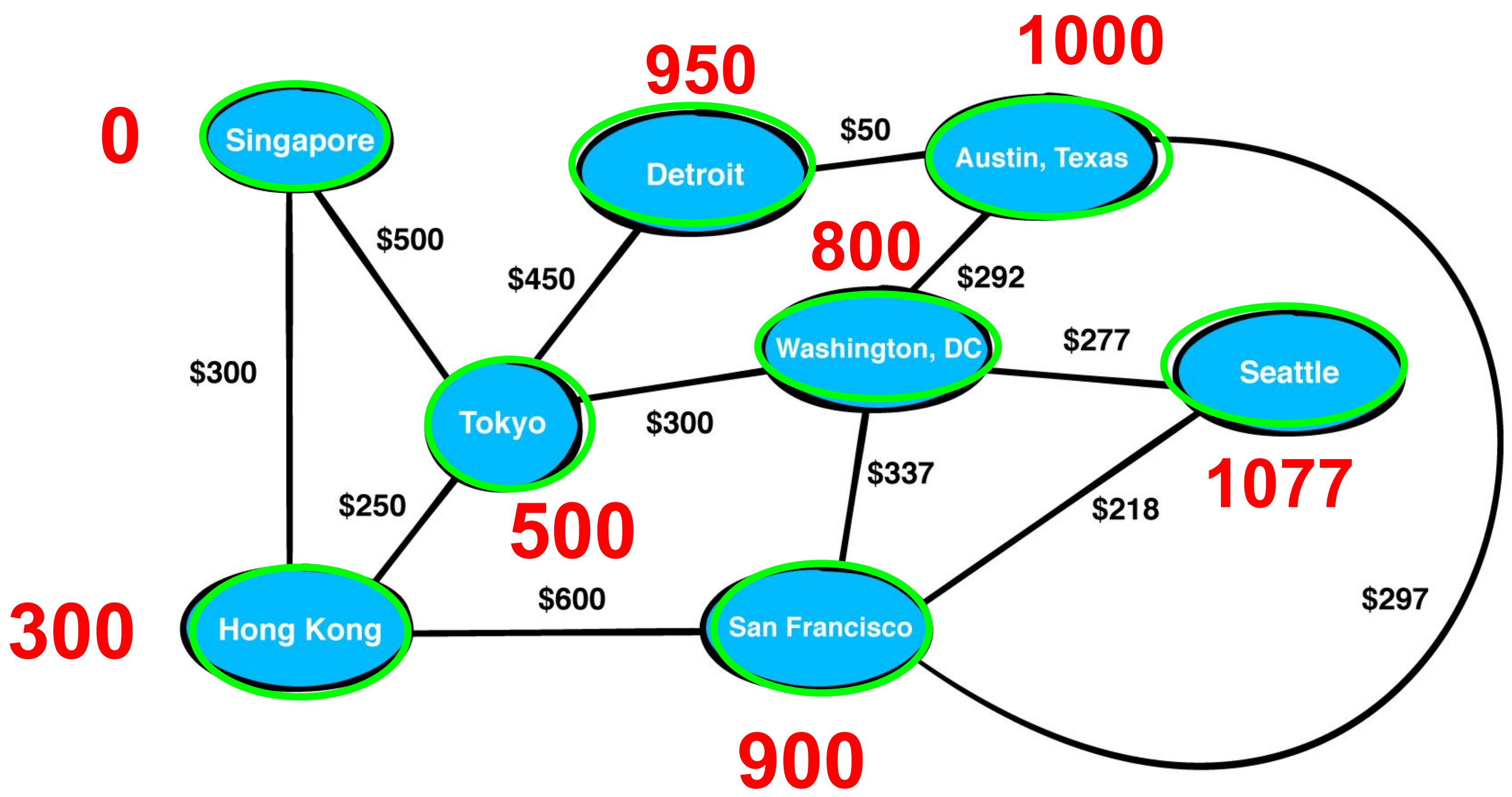
Graphs - Dijkstra Algorithm

Since every adjacent nodes of Austin has been visited we set it has visited!



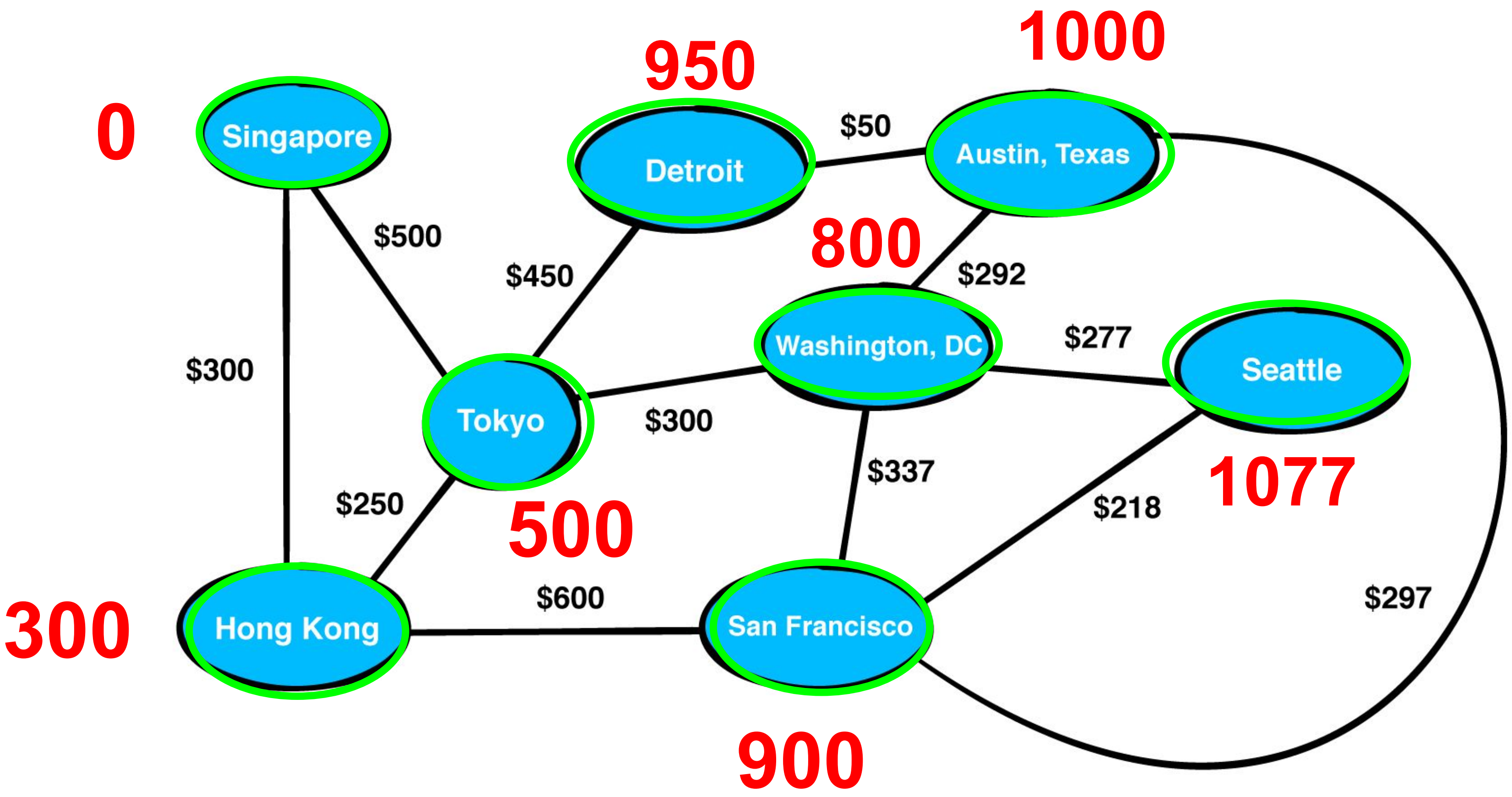
Graphs - Dijkstra Algorithm

Same thing happen to Seattle!



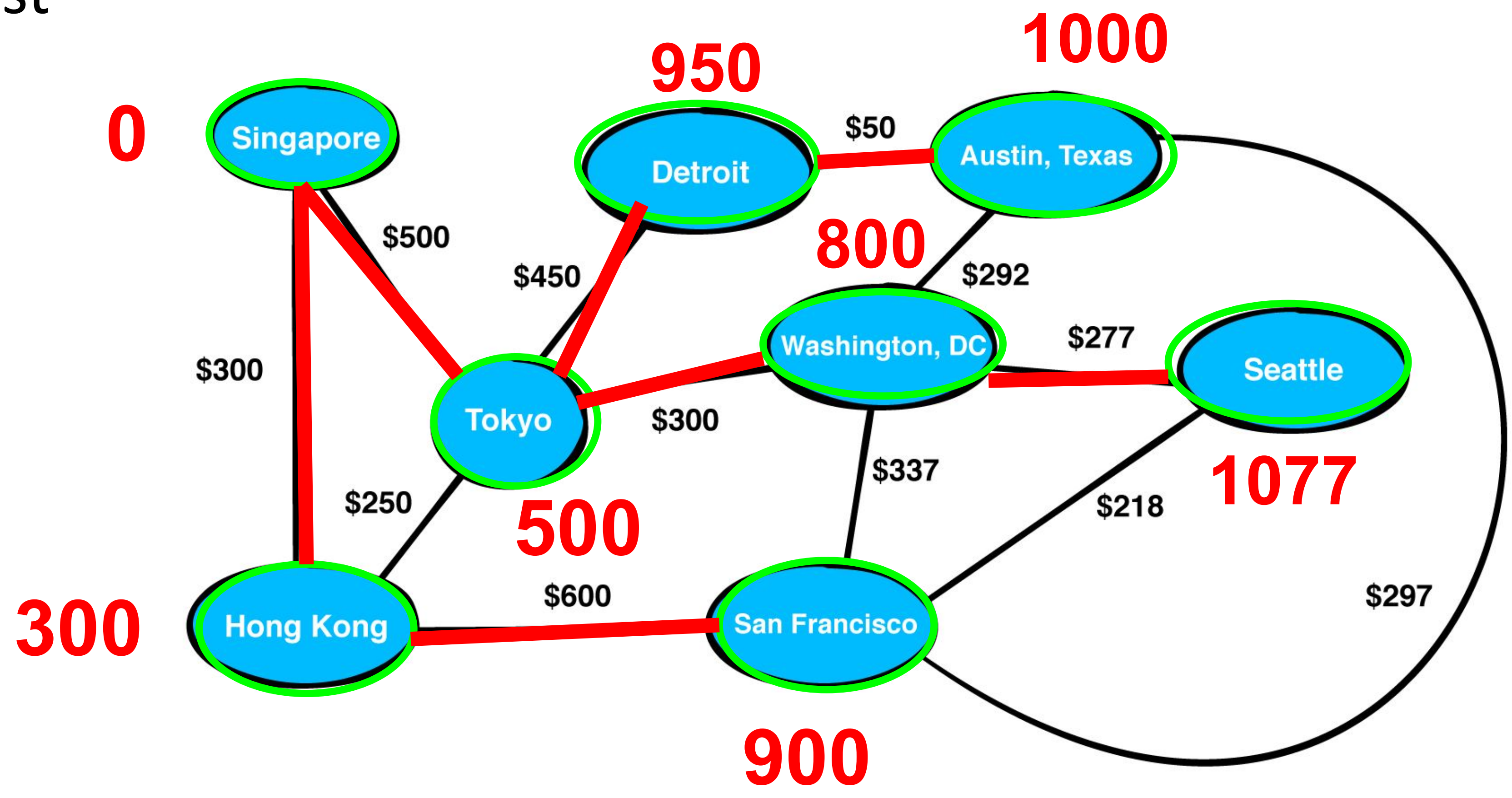
Graphs - Dijkstra Algorithm

Finished!



Graphs - Dijkstra Algorithm

We can draw the paths from Singapore to any other country that have the least cost



Graphs - Dijkstra Algorithm

Pseudocode

Initialization

```
function Dijkstra(Graph, source):

    for each vertex v in Graph.Vertices:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]
        remove u from Q

        for each neighbor v of u still in Q:
            alt ← dist[u] + Graph.Edges(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

Graphs - Dijkstra Algorithm

Pseudocode

```
function Dijkstra(Graph, source):

    for each vertex v in Graph.Vertices:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]
        remove u from Q

        for each neighbor v of u still in Q:
            alt ← dist[u] + Graph.Edges(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

the process we have done in the example

Graphs - Dijkstra Algorithm

Complexity:

If the queue is implemented using **binary heaps** the complexity is: $\Theta(|E| + |V| \log |V|)$

If the queue is implemented using **fibonacci heaps** the complexity is: $O(|E| + |V| \log |V|)$