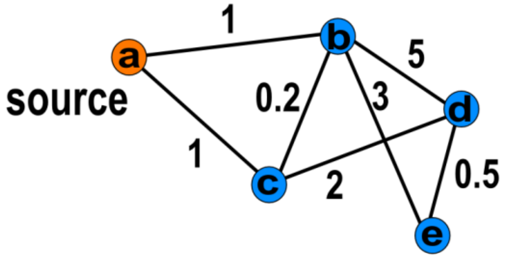
Graphs Algorithms

Mustafa Hajij



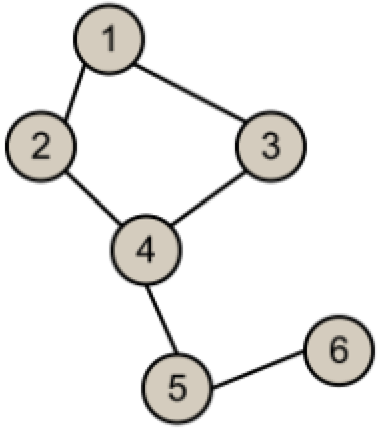
Graphs

A **graph** is an ordered pair (V,E) where,

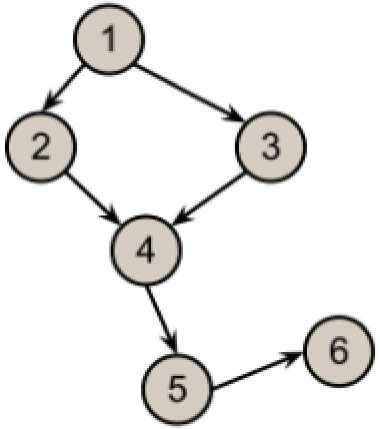
• V is the *vertex set (also node set)* whose elements are the vertices, or *nodes* of the graph.

• E is the *edge set* whose elements are the edges, or connections between vertices, of the graph. If the graph is undirected, individual edges are unordered pairs.

• If the graph is directed, edges are ordered pairs

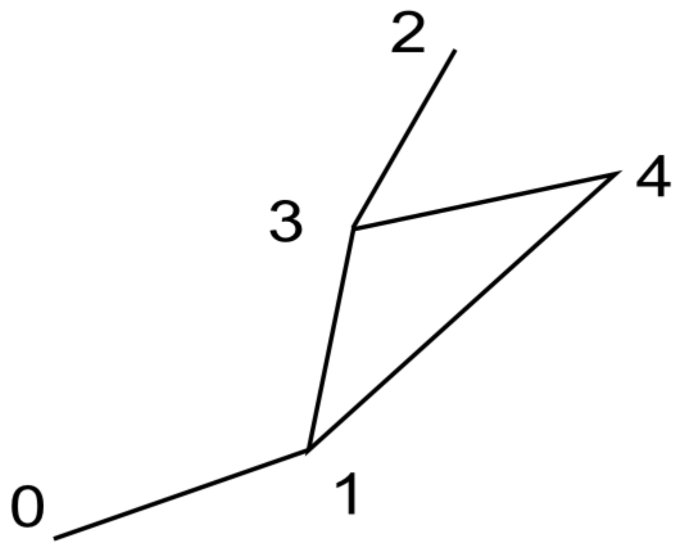


undirected



directed

Graphs representation: list of nodes and edges

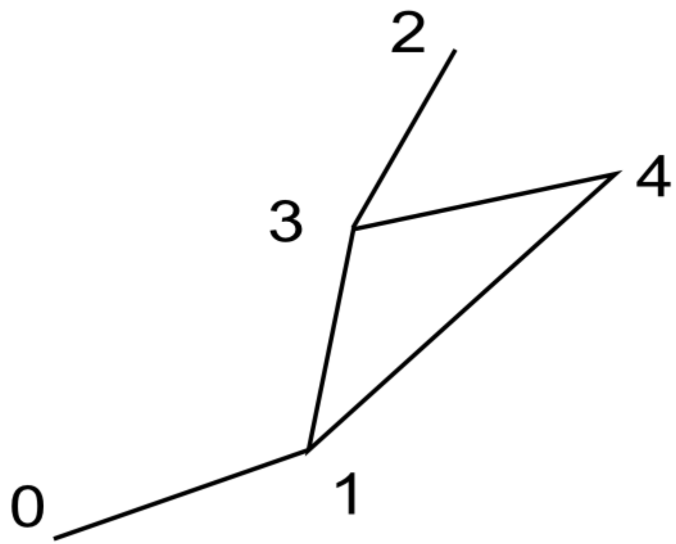


Nodes: [0,1,2,3,4]

Edges: [[0,1], [1,3],[1,4] [3,4], [3,2]]

Note that if the graph is connected, then the list of edges is enough to determine the graph completely.

Graphs representation: list of nodes and edges

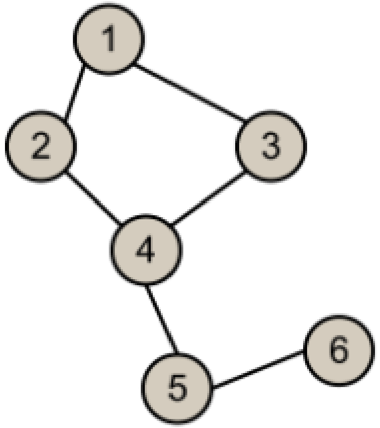


Nodes: [0,1,2,3,4]

Edges: [[0,1], [1,3],[1,4] [3,4], [3,2]]

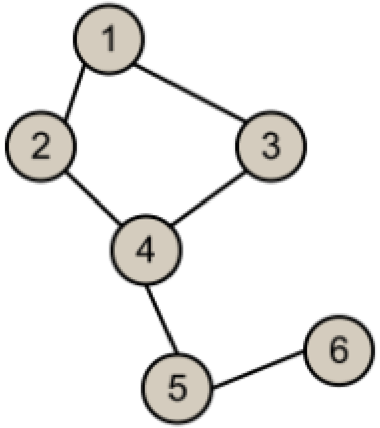
The order of the vertices is important only if the graph is directed.

Graphs representation: adjacency matrix

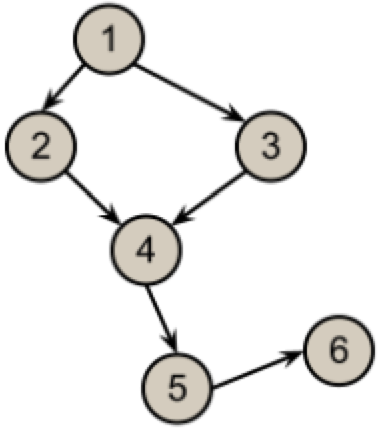


	1	2	3	4	5	6
1	0	1	1	0	0	0
2	1	0	0	1	0	0
3	1	0	0	1	0	0
4	0	1	1	0	1	0
5	0	0	0	1	0	1
6	0	0	0	0	1	0

Graphs representation: adjacency matrix



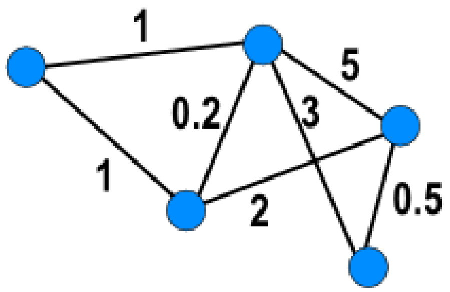
	1	2	3	4	5	6
1	0	1	1	0	0	0
2	1	0	0	1	0	0
3	1	0	0	1	0	0
4	0	1	1	0	1	0
5	0	0	0	1	0	1
6	0	0	0	0	1	0



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

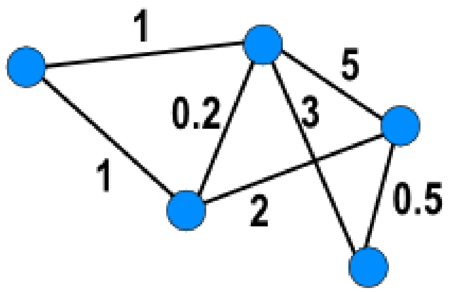
Weighted Graphs

A weighted graph is a graph in which every edge has a weight (non-negative real number)



Weighted Graphs

A weighted graph is a graph in which every edge has a weight (non-negative real number)



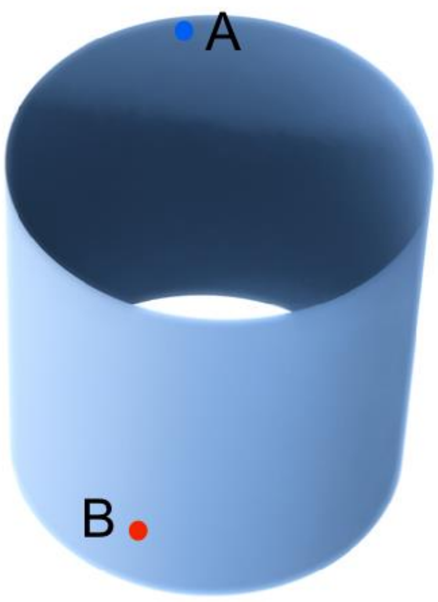
Formally speaking :

A weight function $w: E \rightarrow R^{+}$. In other words, the function w associates to every edge e a positive number (weight) $w(e)$

A weighted graph is a graph G=(V,E) with a weight function $w: E \rightarrow R^{+}$.

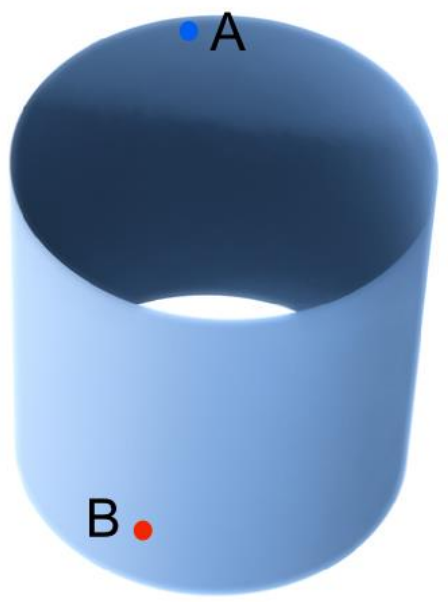
Shortest distance

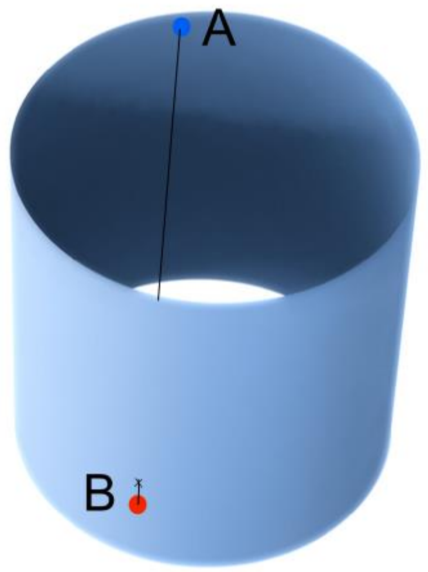
What is the shortest distance between A and B?

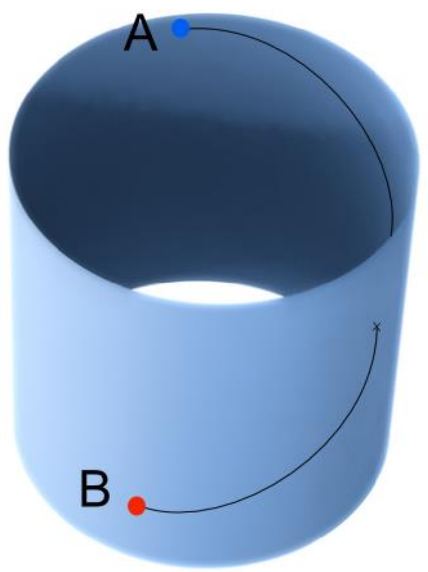


Shortest distance

What is the shortest distance between A and B?





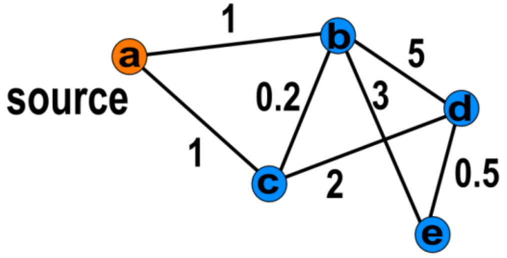


Dijkstra algorithm

The basic Dijkstra algorithm operates on connected, undirected, weighted graph.

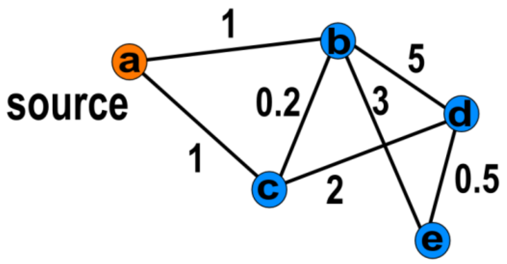
```
Dijkstra(Graph, source):
    for each vertex v in Graph:
        distance[v] := infinity          // the initial distance from source to any other vertex v is infinity
        previous[v] := undefined
    distance[source] := 0                // Distance from the source to itself is zero
    Q := the set of all nodes in the Graph   // all nodes are going in this container
    while Q is not empty:                // main loop
        u := the node in Q with smallest distance from the source   (what kind of queue you use here?)
        remove u from Q                  //the source will be removed first
        for each neighbor v of u:        // v is still in the container Q
            alt := distance[u] + length(u, v)
            if alt < distance[v]                 //A shorter path from v to the source has been found
                distance[v] := alt
                previous[v] := u
    return distance[], previous[ ]
```

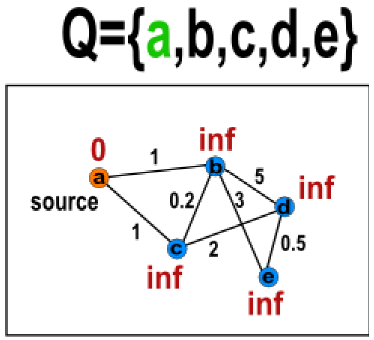
Dijkstra algorithm : Example



Input : weighted graph
with a source vertex

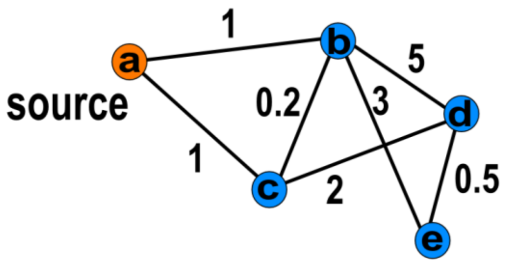
Dijkstra algorithm : Example



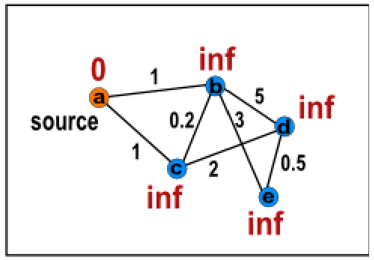


Algorithm starts by initializing the distance to every vertex other than the source to infinity. We also create a queue Q and put in it all vertices of G.

Dijkstra algorithm : Example

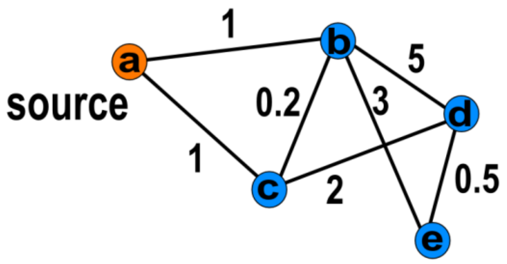


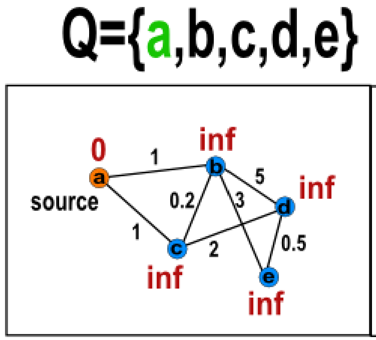
Q={a,b,c,d,e}

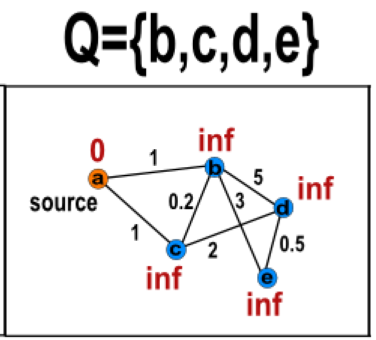


When we enter the while loop we dequeue the element in Q with shortest distance to source. In this case it is a.

Dijkstra algorithm : Example

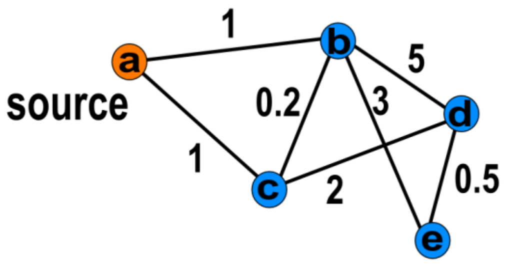


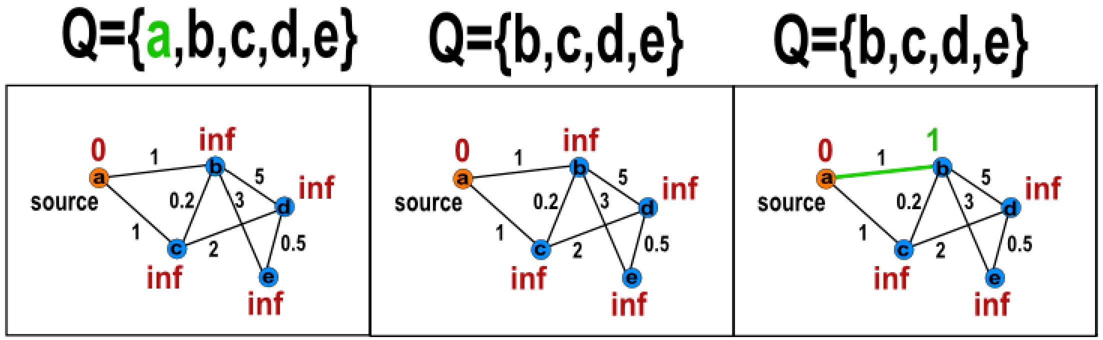




When we enter the while loop we dequeue the element in Q with shortest distance to source. In this case it is a.

Dijkstra algorithm : Example





Now we visit all neighbors of a and update the distance to them:

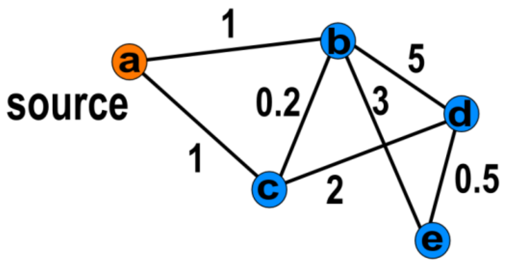
for each neighbor v of u:

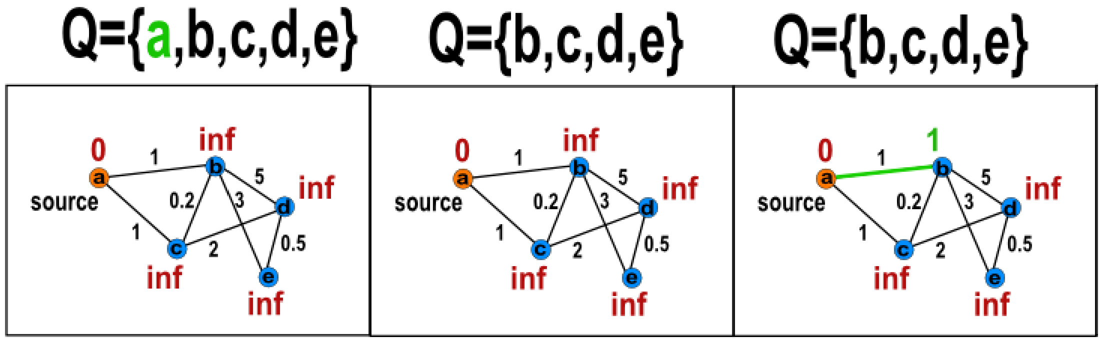
alt := distance[u] + length(u, v)

if alt < distance[v]

distance[v] := alt

Dijkstra algorithm : Example





In this case we update the distance to b to 1.

Now we visit all neighbors of a and update the distance to them:

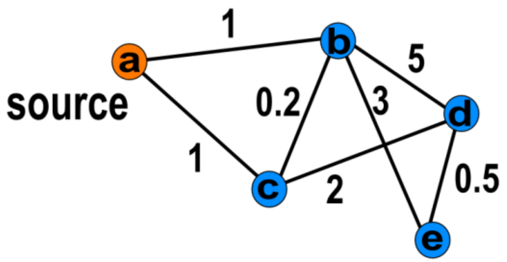
for each neighbor v of u:

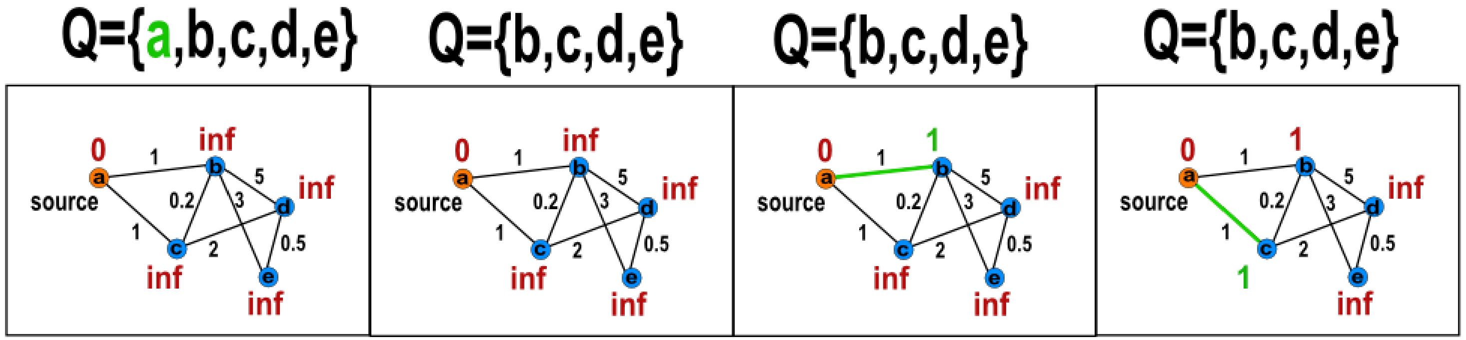
alt := distance[u] + length(u, v)

if alt < distance[v]

distance[v] := alt

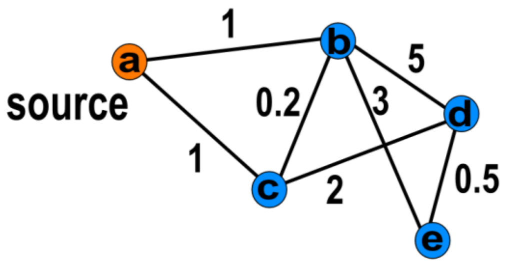
Dijkstra algorithm : Example

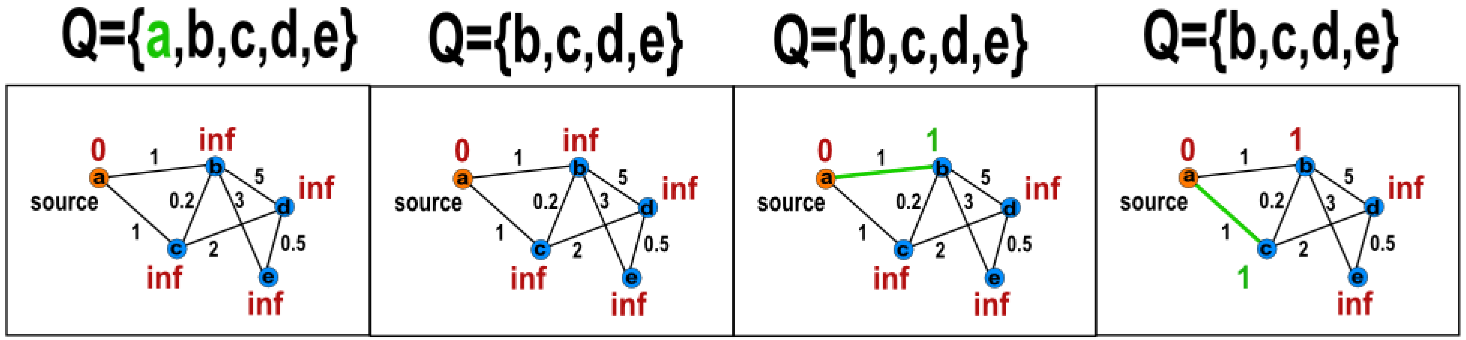




Here we update the distance to c to be 1 as well

Dijkstra algorithm : Example

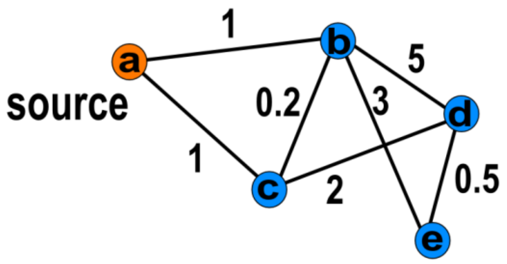




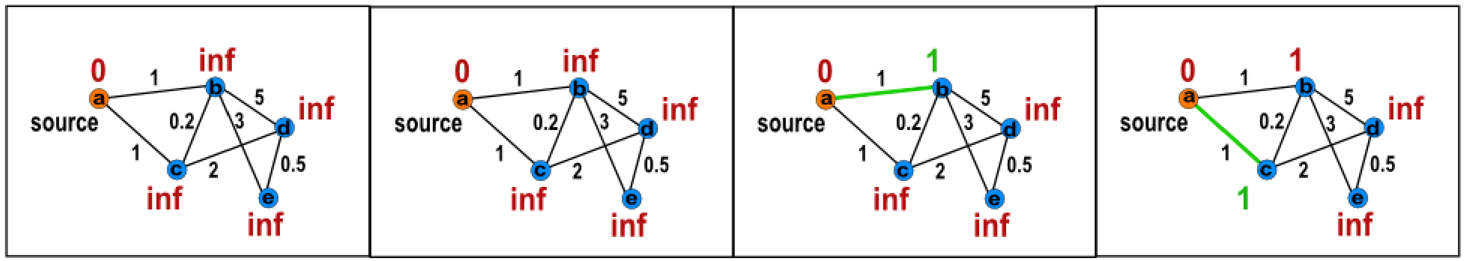
Here we update the distance to c to be 1 as well

At this stage all neighbors of a have been visited so we check the queue again: if is not empty we start the process again.

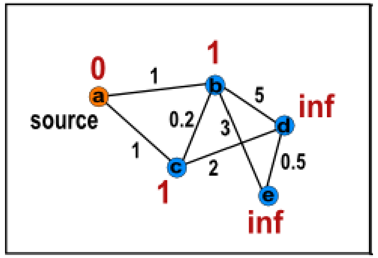
Dijkstra algorithm : Example



Q={a,b,c,d,e} Q={b,c,d,e} Q={b,c,d,e} Q={b,c,d,e}

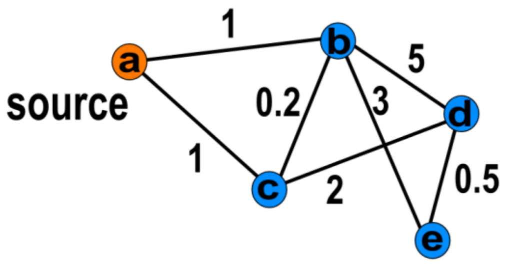


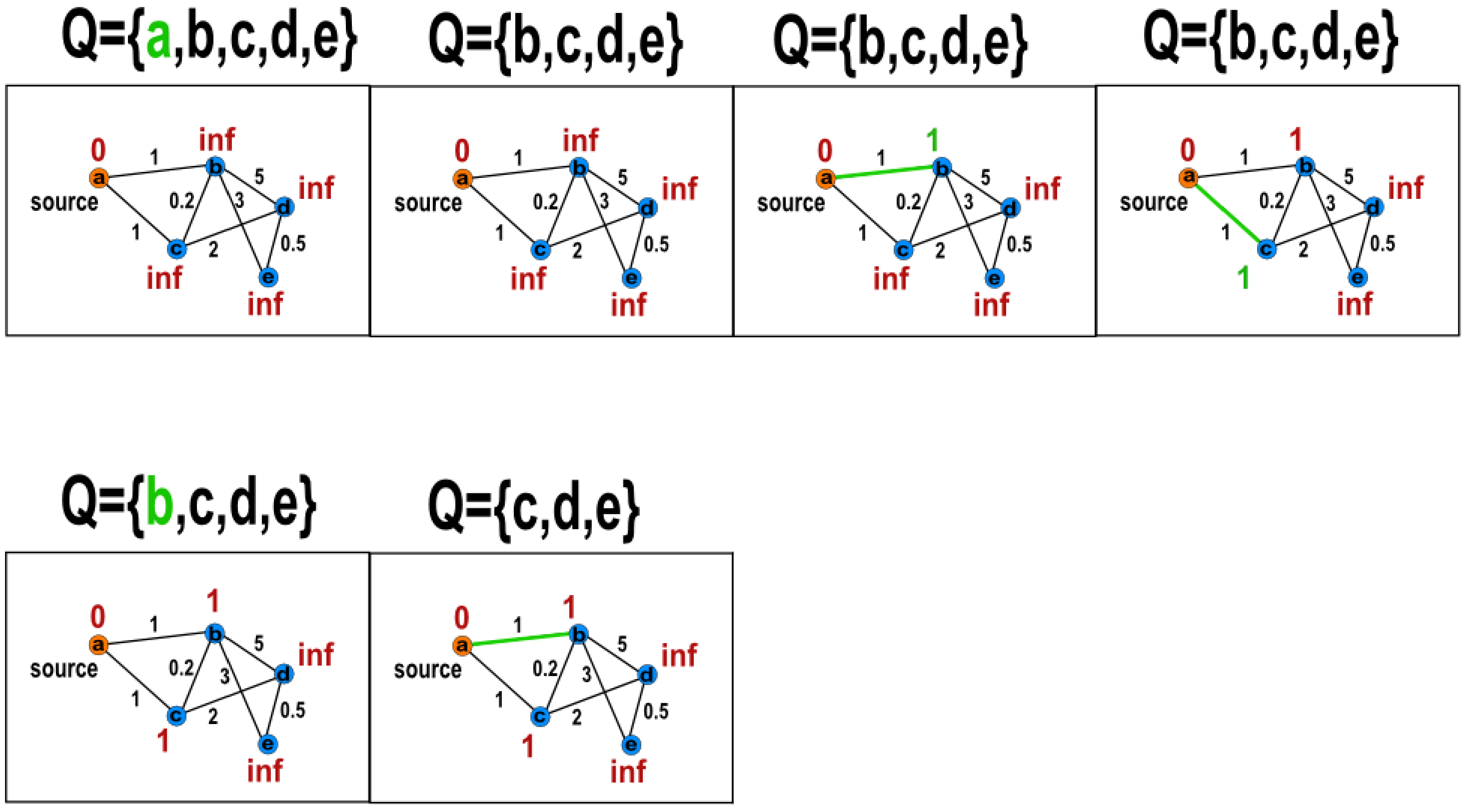
Q={b,c,d,e}



We select b (the closest element to a so far)

Dijkstra algorithm : Example

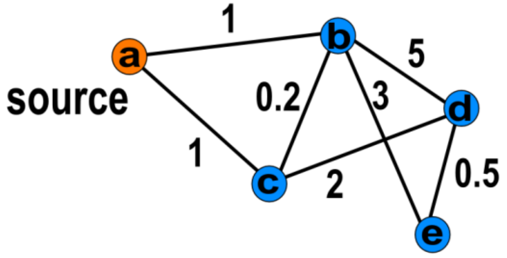


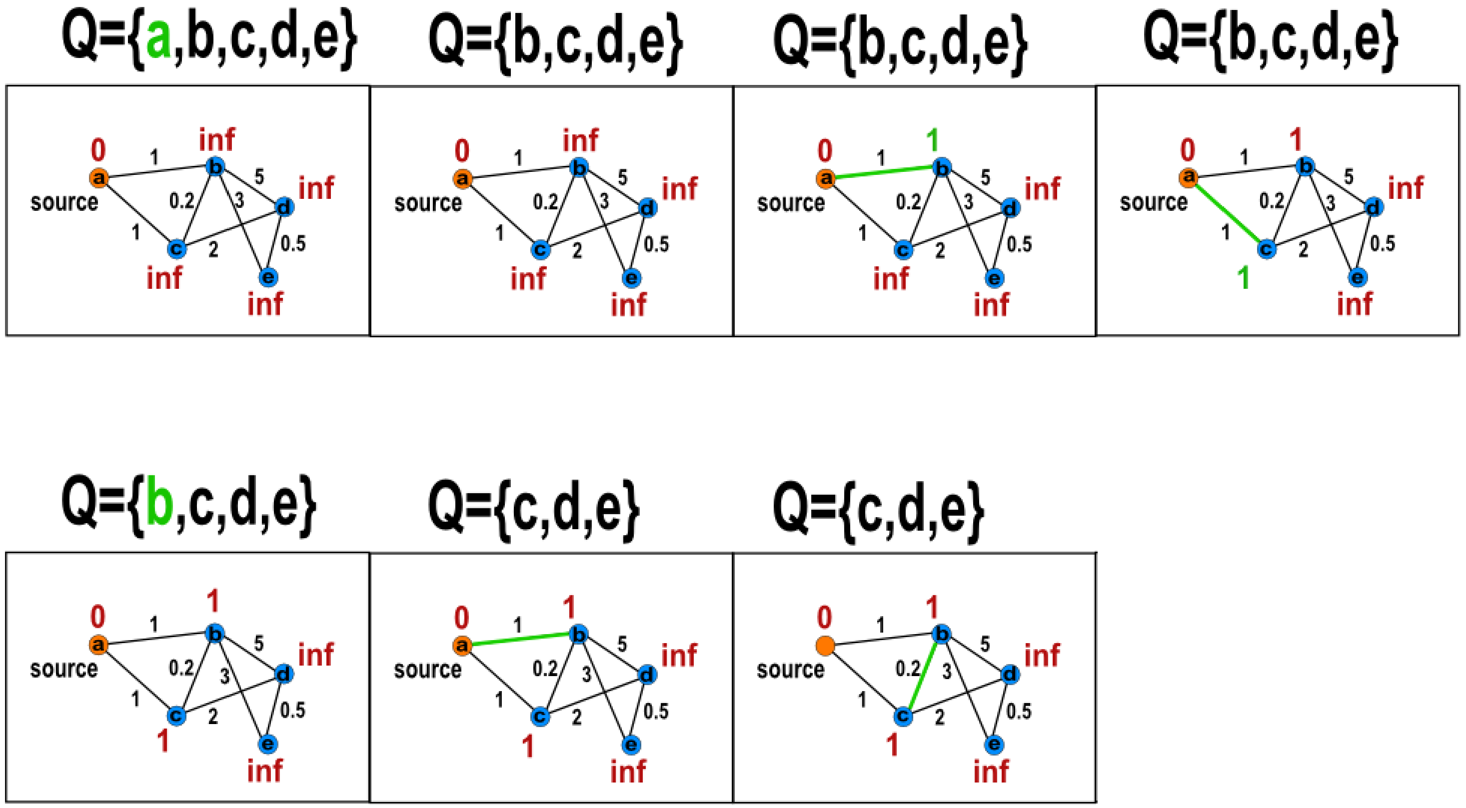


Remove b from the queue and start visiting all its neighbors and update the distance.

In this case the distance does not update—why ?

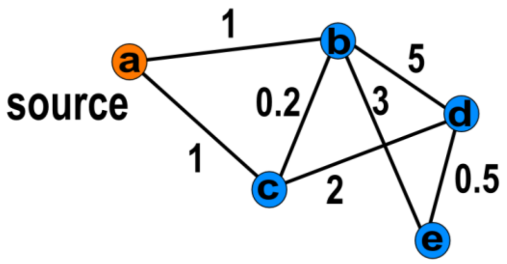
Dijkstra algorithm : Example

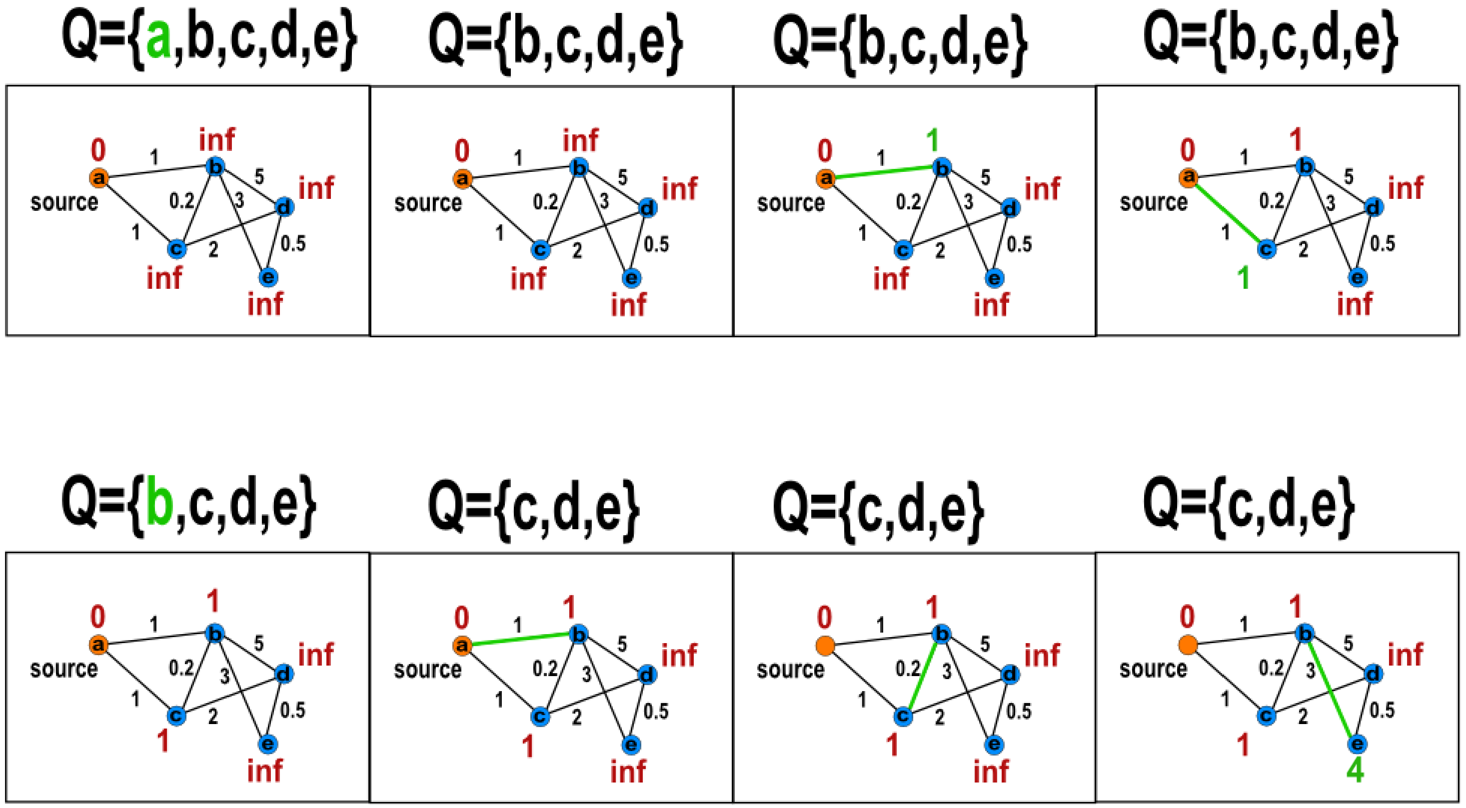




The distance here also does not update

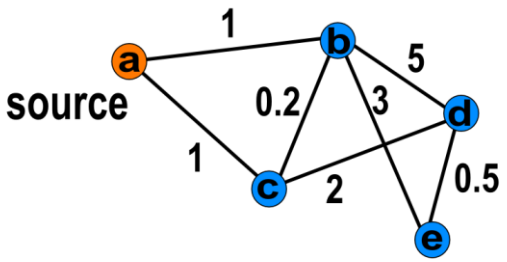
Dijkstra algorithm : Example



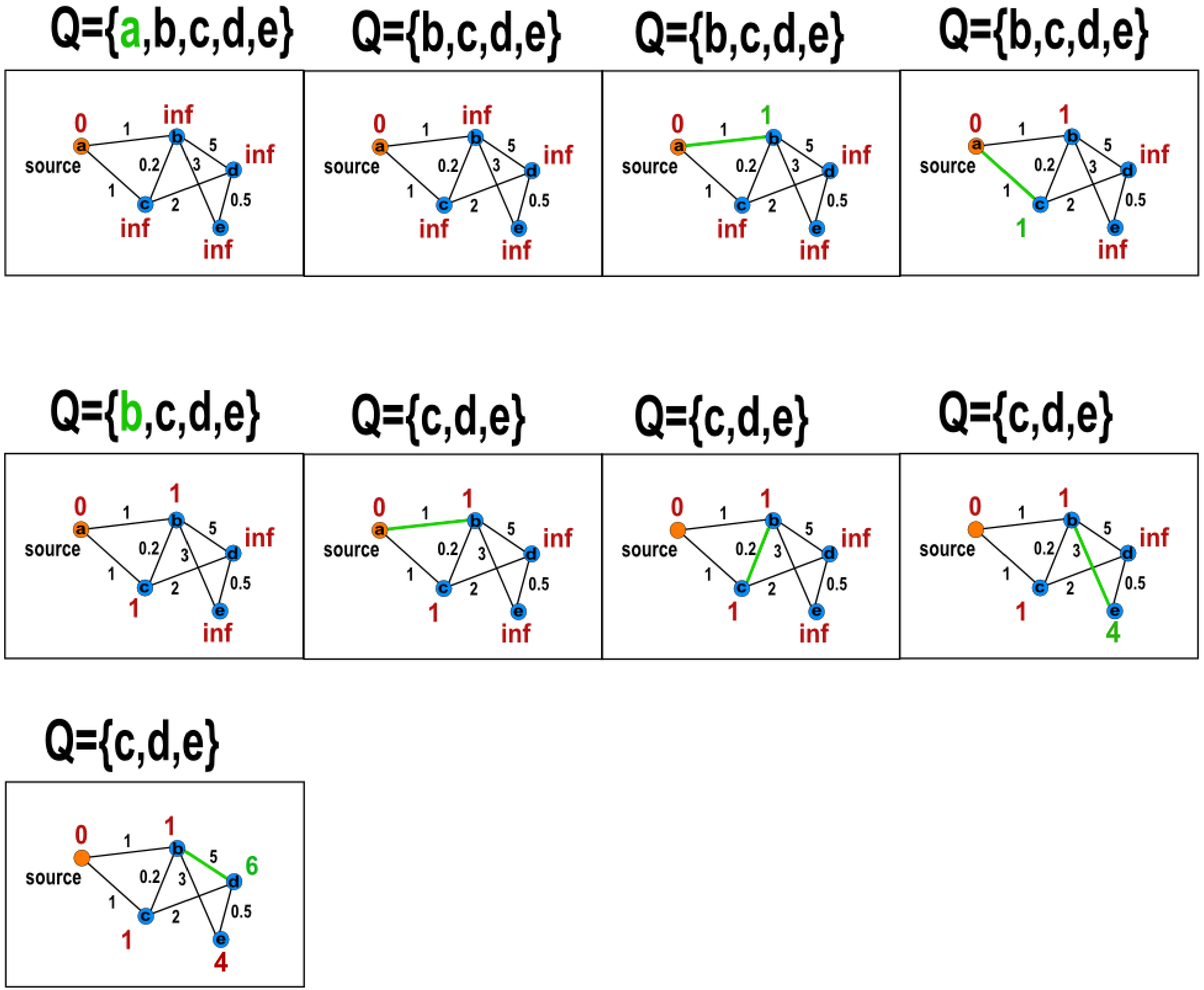


Here we update the distance from a to e to be 4

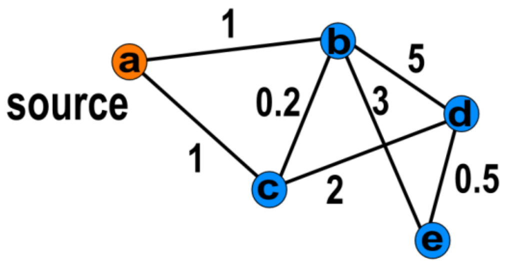
Dijkstra algorithm : Example

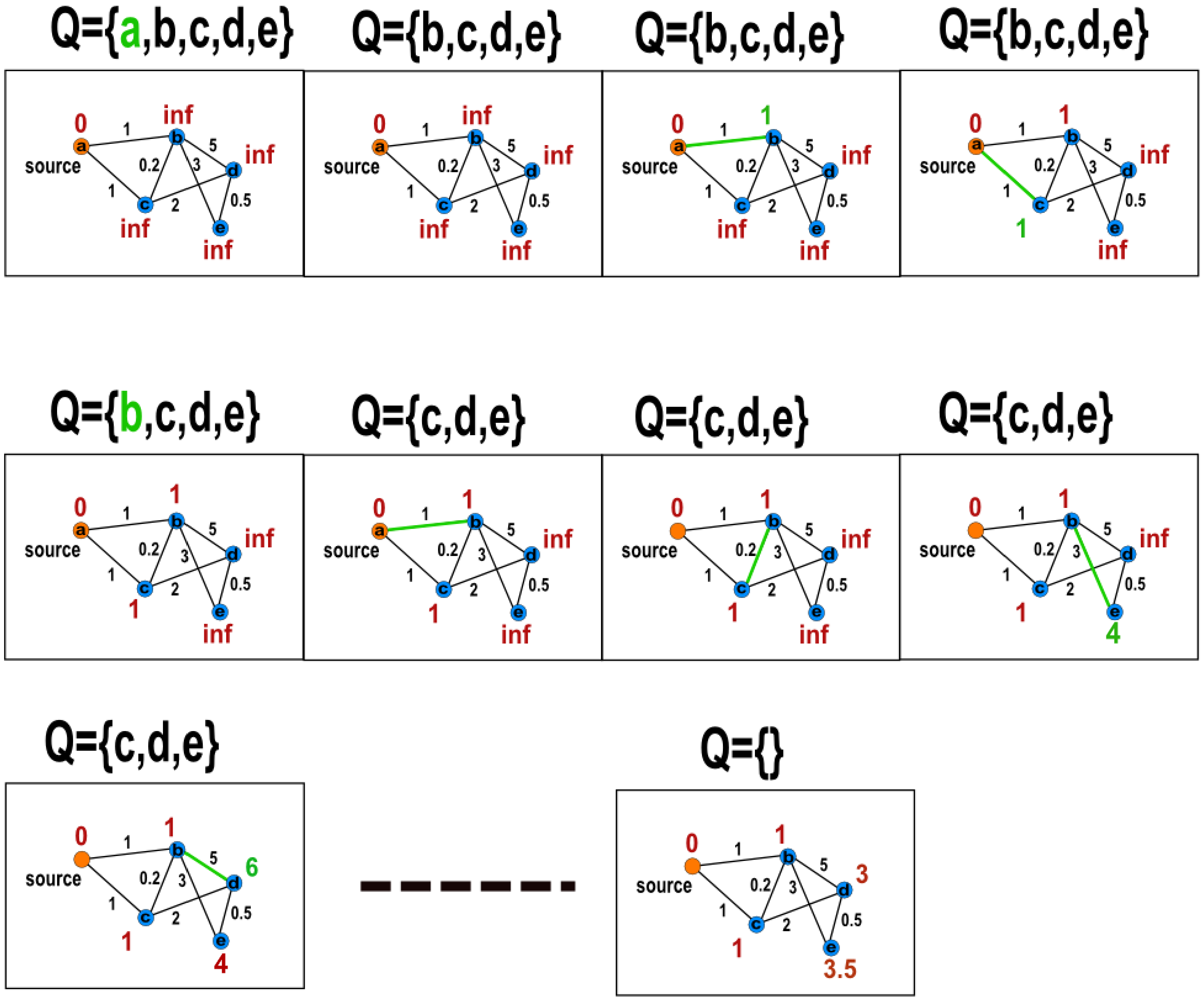


Update the distance from a to d



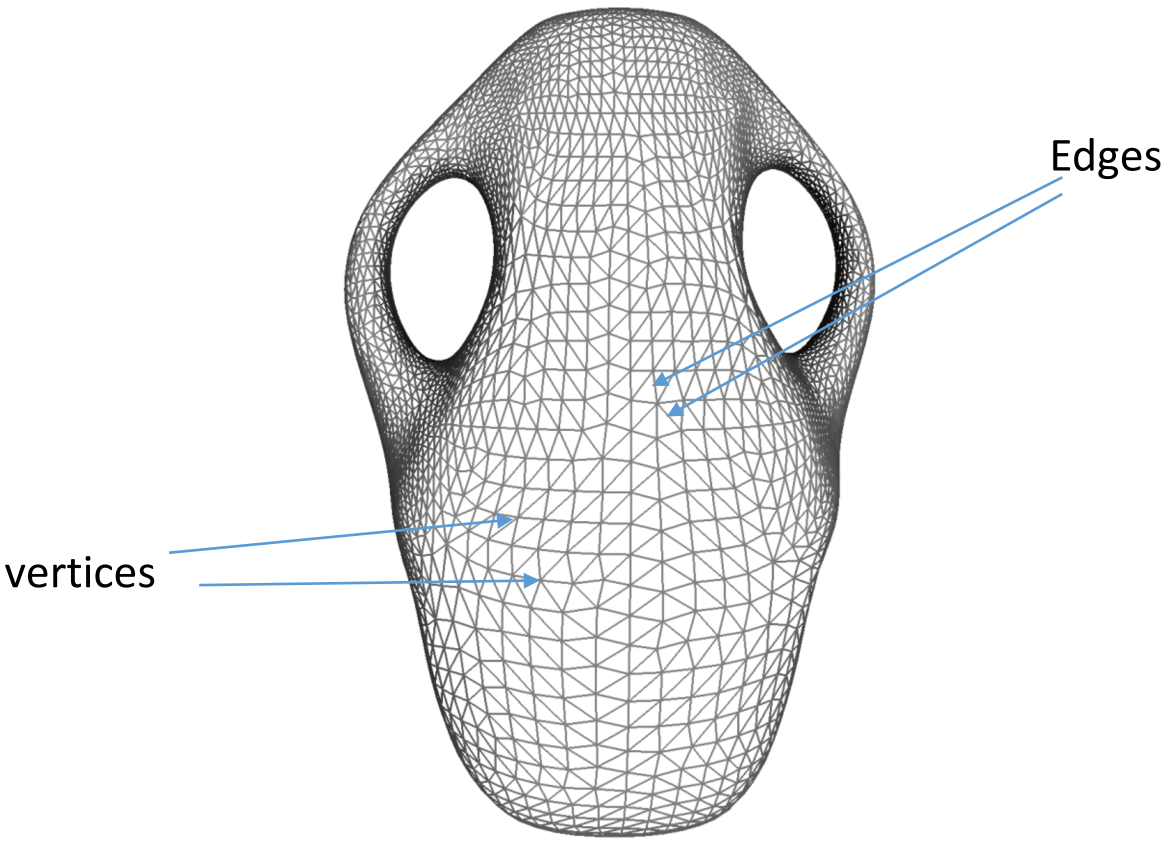
Dijkstra algorithm : Example





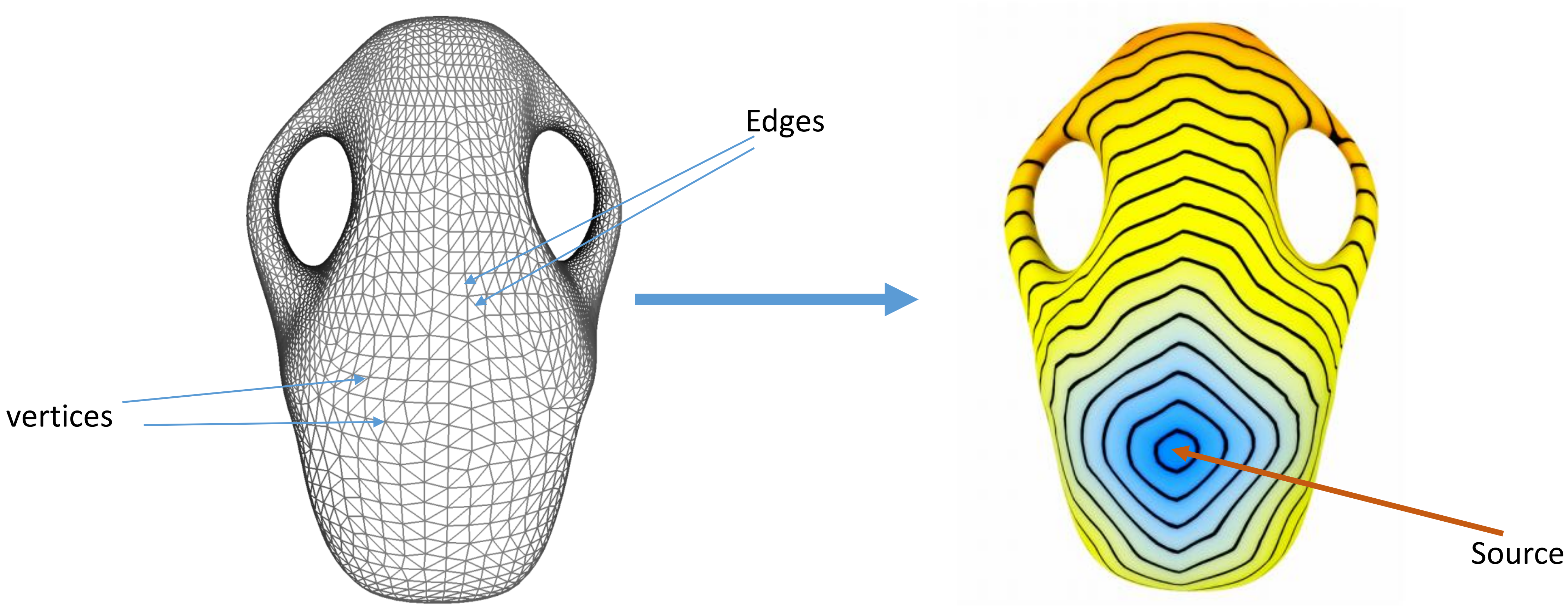
And so on

Example



We can view a mesh as a graph and apply Dijkstra algorithm on it.

Example



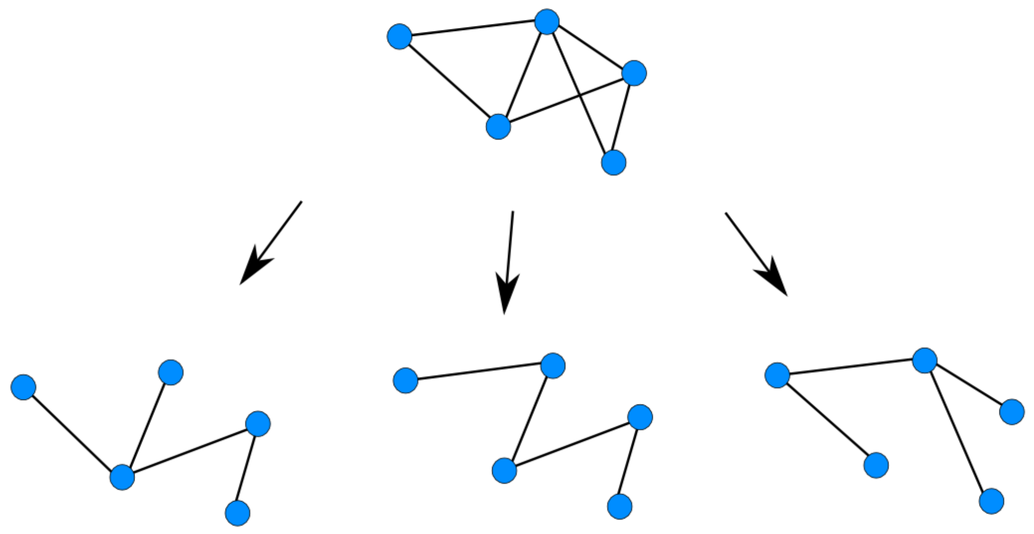
We can view a mesh as a graph and apply Dijkstra algorithm on it.

Blue indicates the regions closest to the source

Spanning Tree

Let $G = (V, E)$ be a connected weighted graph. A spanning tree for G is a subgraph of G which includes all of the vertices of G and is a tree.

A graph might have more than one spanning tree

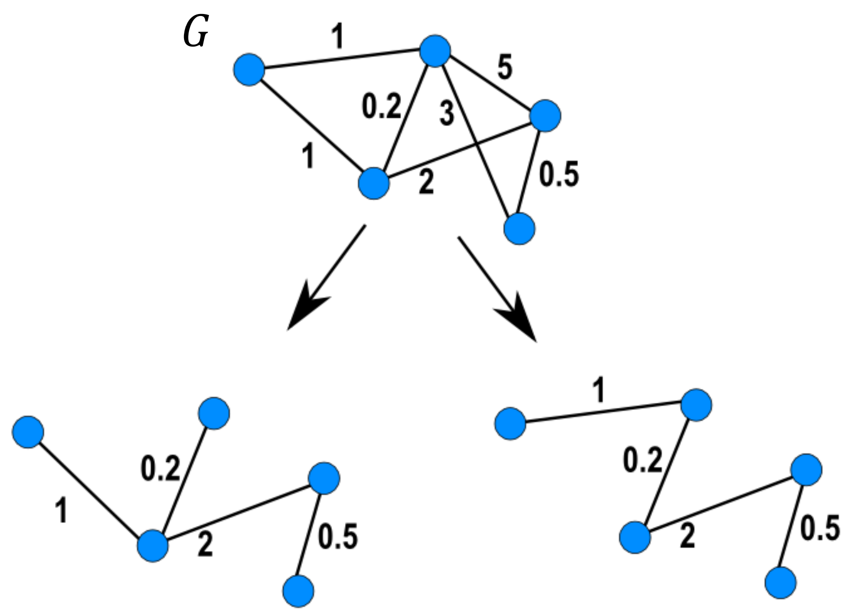


Spanning trees for G

Minimal Spanning Tree

Let $G = (V, E, w)$ be a connected weighted graph. A minimal spanning tree for G is a spanning tree whose sum of edge weights is as small as possible.

A graph might have more than one minimal spanning tree. However, if all edges in the graph have unique weights then the minimal spanning tree is unique.



Minimal spanning trees for G

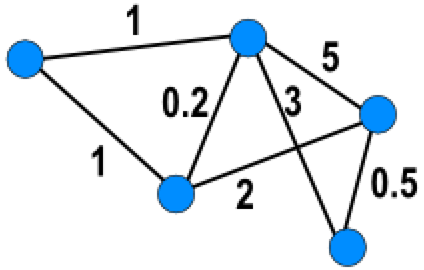
Kruskal's Algorithm

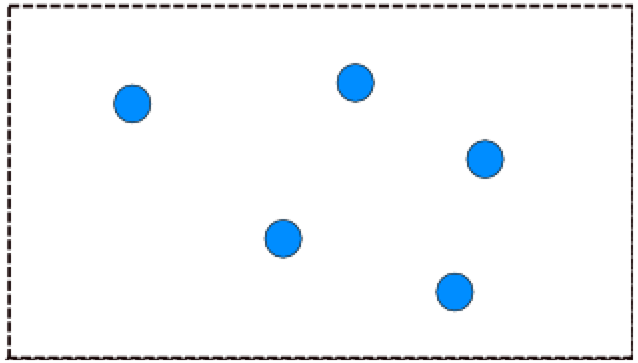
Let $G = (V, E, w)$ be a connected weighted graph. The Kruskal's algorithm is a greedy algorithm.

Informally, the algorithm can be given by the following three steps :

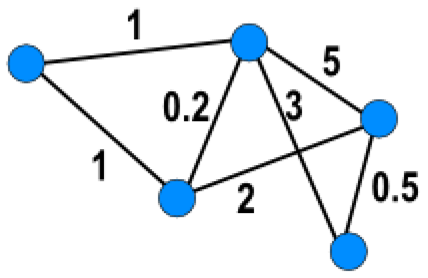
1. Set V_T $to\ be\ V, Set\ E_T = \{\}$. Let $S = E$
2. While S is not empty and T is not a spanning tree
    1. Select an edge e from S with the minimum weight and delete e from S.
    2. If e connects two separate trees of T then add e to E_T

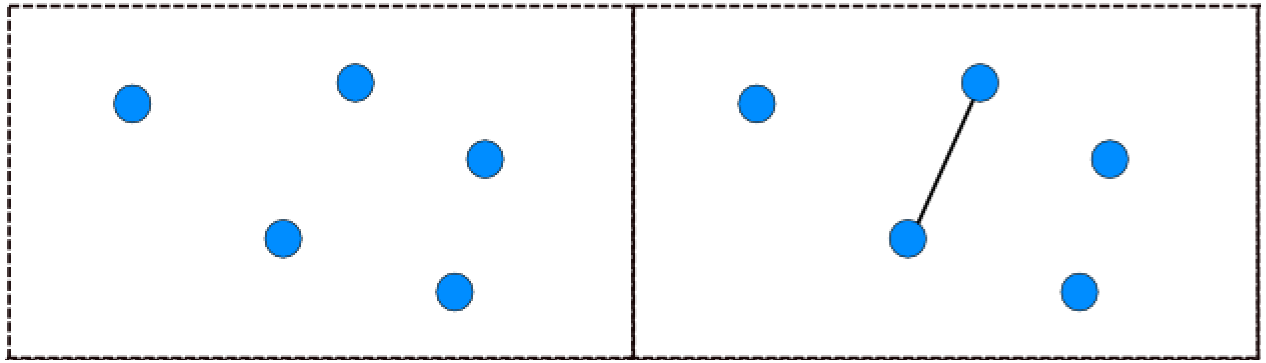
Kruskal’s Algorithm Example



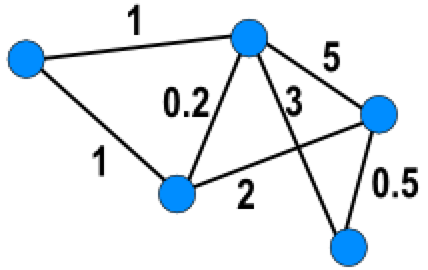


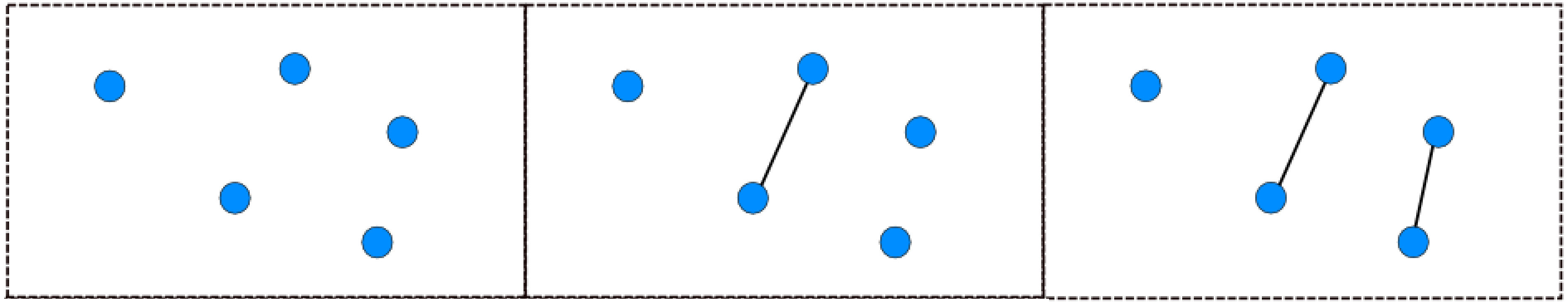
Kruskal’s Algorithm Example



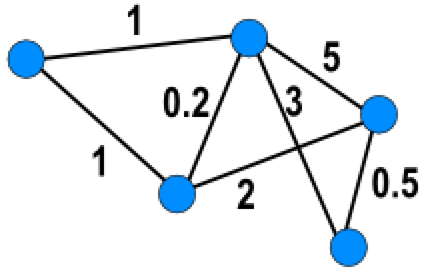


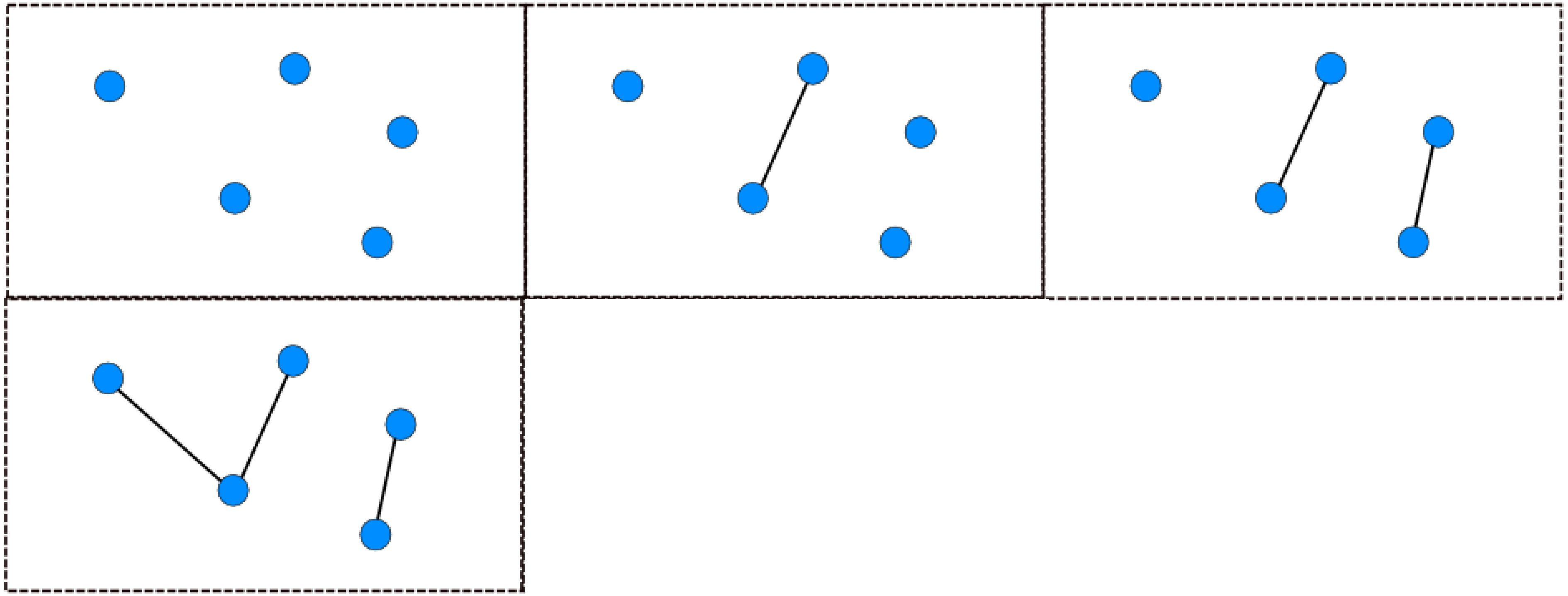
Kruskal’s Algorithm Example



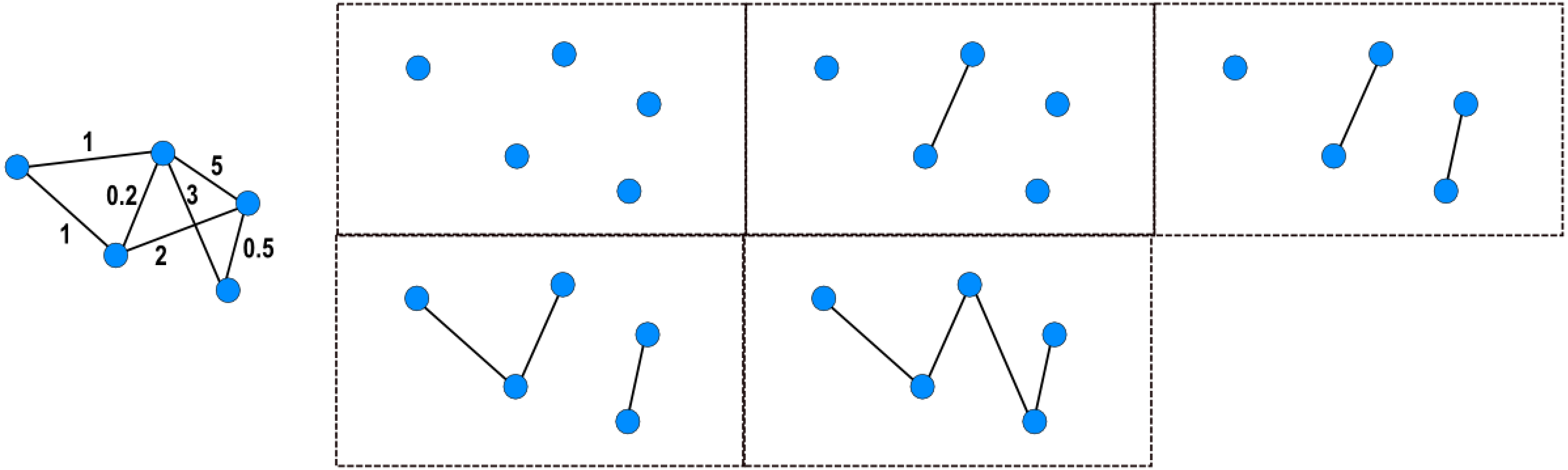


Kruskal’s Algorithm Example





Kruskal's Algorithm Example



Interestingly : this simple algorithm (with very simple greedy strategy) yields a minimal spanning tree!

Kruskal's Algorithm

Let $G = (V, E, w)$ be a connected weighted graph. The Kruskal's algorithm is a greedy algorithm

This can be implemented using union-find data-structure. Union-find is a data structure that can be used to create, merge and track connected components.

In short a union find data structure allows for three operations :

(1) MAKE-SET(v) : Make a connected component from the node. O(1)

(2) FIND-SET(u) :given a node u, returns a pointer to the connected component it belongs to. O(log(n))*

(3) UNION(u, v): Given two nodes u, v that may belong to two separate connected components, merge these two separate connected components into a single one. O(log(n))*

Kruskal's Algorithm

Let $G = (V, E, w)$ be a connected weighted graph. The Kruskal's algorithm is a greedy algorithm

This can be implemented using union-find data-structure. Union-find is a data structure that can be used to create, merge and track connected components.

In short a union find data structure allows for three operations :

(1) MAKE-SET(v) : Make a connected component from the node. O(1)
(2) FIND-SET(u) :given a node u, returns a pointer to the connected component it belongs to. O(log(n))*
(3) UNION(u, v): Given two nodes u, v that may belong to two separate connected components, merge these two separate connected components into a single one. O(log(n))*

* The optimal complexity is in fact O($\alpha(n)$), $\alpha(n)$ is the extremely slow-growing inverse Ackermann function.

Kruskal's Algorithm

Let $G = (V, E, w)$ be a connected weighted graph. The Kruskal's algorithm is a greedy algorithm

This can be implemented using union-find data-structure. Union-find is a data structure that can be used to create, merge and track connected components.

In short a union find data structure allows for three operations :

(1) MAKE-SET(v) : Make a connected component from the node. O(1)
(2) FIND-SET(u) :given a node u, returns a pointer to the connected component it belongs to. O(log(n))*
(3) UNION(u, v): Given two nodes u, v that may belong to two separate connected components, merge these two separate connected components into a single one. O(log(n))*

Kruskal's algorithm

1- A= {}
2-foreach $v \in V$:
3- MAKE-SET(v)
4-foreach (u, v) in E ordered by weight(u, v), increasing:
5- if FIND-SET(u) $\neq$ FIND-SET(v):
6- $A = A \cup \{(u, v)\}$
7- UNION(u, v)
8-return A

Prim's Algorithm

Let $G = (V, E, w)$ be a connected weighted graph. The Prim's algorithm is a greedy algorithm

Informally, the algorithm can be given by the following three steps :

1. Select an arbitrary vertex v from V. Set $V_T = \{v\}\ and\ E_T = \{\quad\}$
2. Grow the tree by one edge : choose an edge e(u,v) from the set E with the lowest cost such that u in V_T and v is in $V \backslash V_T$ then add v to V_T and add e to E_T
3. If $V_T = V$ break, otherwise go to step 2.

Application to Clustering : Zahn's algorithm

Zahn's algorithm that we used to obtain a clustering algorithm on point cloud can be simply used to obtain a clustering algorithm on graphs as follows.

Suppose that we are given a set of a weighted graph G.

1. Construct the MST of G (using say Kruskal's algorithm).
2. Remove the *inconsistent edges* to obtain a collection of connected components (clusters).
3. Repeat step (2) as long as the termination condition is not satisfied.

The connected components of the remaining forest are the clusters of the graph

In this case, an edge in the tree is called inconsistent if it has a length more than a certain given length L

Question : how can you apply this algorithm to point cloud ?