Graph Algorithms

Minimum Spanning Tree (Prim's and Kruskal's algorithms)

Single Source Shortest Path

All Pair Shortest Path

Graph Representation: Adjacency Matrix

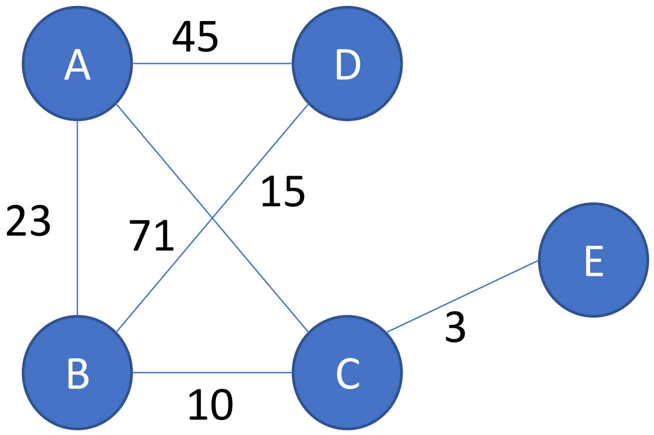
- Adjacency Matrix representation
- Adjacency List representation
- Adjacency Matrix representation
    - Represent each vertex in an nxn matrix where, n is the total number of nodes in the graph
    - The (i,j)th entry of the matrix is 1 if there exists an edge between vertex i and vertex j
    - If the edge between i and j is also weighted, then the (i,j)th entry is set to w, where w is the weight of the edge
    - If there is no edge between two vertices, the matrix entry is set to 0

Graph Representation: Adjacency List

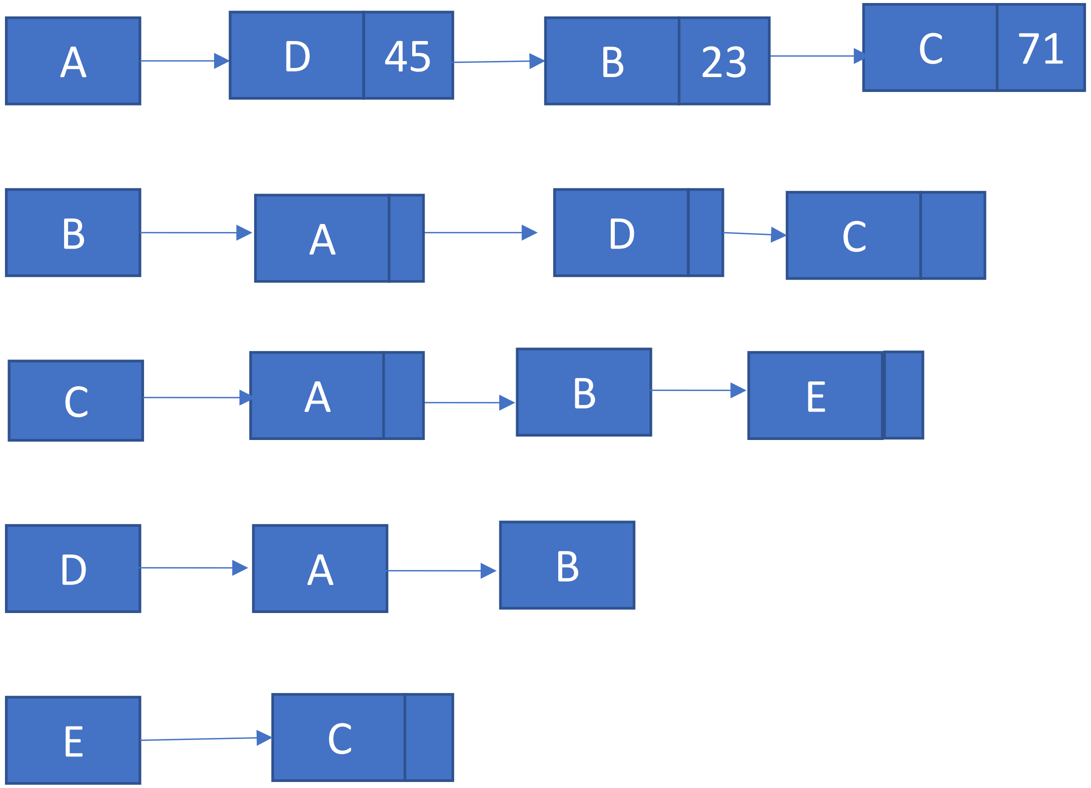
- For each vertex of the graph, a linked list is maintained which contains the edges originating from the graph
- Thus, n rows of the adjacency matrix essentially become n-linked lists
- Each element of the linked list contain weight of the edge (1 if there is no weight), name of the ending vertex and a pointer to the next edge originating from the vertex
- The vertex entries themselves may be stored in an array or a binary search tree or a hash table

Adjacency Matrix

	A	B	C	D	E
A	0	23	71	45	0
B	23	0	10	15	0
C	71	10	0	0	3
D	45	15	0	0	0
E	0	0	3	0	0



Adjacency List



A → D 45 → B 23 → C 71

B → A → D → C

C → A → B → E

D → A → B

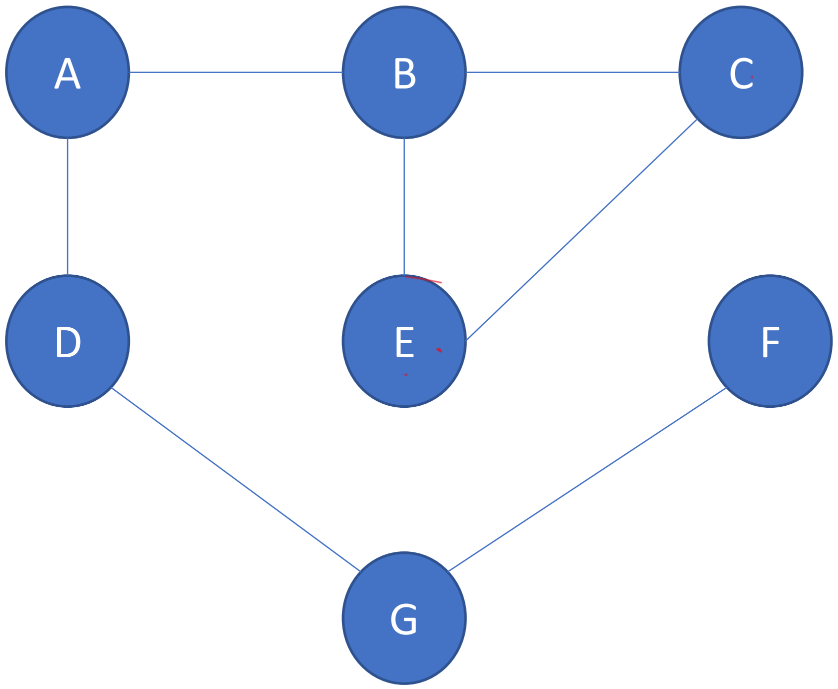
E → C

Graph Traversal

- Traversal means visiting each node of the graph
- Graph traversals are used for searching an element in the graph
- Two commonly used techniques for traversal are:
  - Breadth First Traversal (Search)
  - Depth First Traversal (Search)
- Two concepts are needed for implementing the traversals:
  - When we visit a vertex we call it as visited and store it in a queue
  - When we visit a vertex as well as all the vertices adjacent to it we call it explored

Breadth First Traversal

- We first visit a node v
- visited v is set to 1
- We find out what all its adjacent vertices are and put them all in the queue
- We then, remove the first node from the queue and add all its adjacent vertices to the queue
- Now, this vertex is also explored
- We move in this manner until queue is empty at which point all the nodes have been explored



BFS: ABDCEGF

DFS: ABCEDGF

Algorithm: Breadth First Traversal

```
BFS(v)
{
visited(v) = 1
u <- v
Q <- empty
Repeat{
for all vertices w adjacent from u do
{    if(visited(w)==0)
          { add w to Q
            visited(w) = 1
          }
}
if Q is empty then return;
delete u from Q
} until(false)
} END BFS
```

Q <- queue

visited(1:n) <- 0

visited(i) = 0 implies that the vertex is not visited

visited(i) = 1 implies that the vertex is visited

u contains the first element of the queue

Depth First Traversal

- Here, the exploration of a node is suspended just when a new node is encountered
- At this point, the exploration of this new node starts and it again suspended just when the new adjacent node is found
- This process continues and at each point where we suspend the exploration of a node, it is stored in a queue
- When no new node is found the method backtracks to find the last unexplored node and starts exploring it again
- This process continues until all the vertices of the graph are explored
- This algorithm is based on Greedy technique

Algorithm: Depth First Traversal

```
Algorithm DFS(v)
{
    visited(v) <- 1
    for each vertex w adjacent from v do
    {
        if(visited(w)==0)
            call DFS(w)
    }
} END DFS
```

Spanning Tree

- Given a graph G(V, E), any tree consisting of solely the edges in G and all the vertices in G is called a spanning tree
- In other words, the spanning tree is a subgraph of G that consists of all the vertices of G but not necessarily all the edges of G
- Definition: if G consisting of (V,E) be an undirected connected graph then, a subgraph T (V, E') of G is a spanning tree of G iff T is a tree
- The traversal techniques we had seen earlier both result into a spanning tree i.e. the generated traversals make a spanning tree
- The spanning tree formed using breadth first traversal is called breadth first spanning tree
- Similarly, depth first spanning tree is also defined

Minimum Cost Spanning Tree

- Also known as Minimum Spanning Tree or MST
- Definition: Consider a weighted graph G(V,E), we define the cost of spanning tree as the sum of weights of the edges of the spanning tree. Minimum cost spanning tree is the spanning tree having minimum cost among all possible spanning trees of G
- Thus, in finding MST our aim is to use only minimal cost edges to build the spanning tree
- A greedy technique to solve this problem would be to each time add a minimum weight edge to the graph until all the vertices are included and n-1 edges are added
- For a given graph, there could be more than one MST

Finding MST

- The selection of minimal weight edge is based on some criteria
- Based on the criteria for choosing next edge two types of algorithms are proposed:
    - Kruskal’s Algorithm and
    - Prim’s Algorithm

Prim’s Algorithm

- This algorithm starts with any vertex as the first vertex and the tree is built one vertex at a time.
- That is, each time one new vertex is added to the tree until all the vertices are added
- At each iteration we check the adjacent vertices of the current vertex
- The vertex with minimum weight edge is added to the tree if it is not already present in the tree
- The cost of the tree now becomes original cost + weight of the newly added edge
- We keep adding vertices in this manner until all the vertices are added to the tree
- At this point the tree has exactly n-1 edges and minimum cost

Prim’s Algorithm

- E is the set of edges in G
- cost[1:n, 1:n] is the cost adjacency matrix of an n vertex graph such that cost[i,j] is either a positive real number or ∞ if no edge (i,j) exists
- A minimum spanning tree is constructed and stored as set of edges in the array t[1:n-1, 1:2]
- t[i, 1], t[i, 2] is an edge in the minimum cost spanning tree.
- For every vertex of the graph, near[j] is a vertex in the tree such that cost[j, near[j]] is minimum among all choices for near[j]
- The final cost is returned

- Prim’s(E, cost, n, t)

```
{   let (k,l) be an edge of min. cost in E
     min_cost = cost(k,l)
     t(1,1)=k,      t(1,2)=l
     for i in 1 to n do
     {
            if cost(i,l) < cost(i,k)
                 then near(i) <- l
            else
                 near(i) <- k
     }
     near(k)=near(l) <- 0
for i in 2 to n-2 do
{
    let j be an index s.t. near(j) != 0 and cost(j,near(j)) is minimum then set
    t(i,1) <- j,    t(i,2) <- near(j)
    min_cost <- min_cost + cost(j, near(j))
    near(j) <- 0
    for k in 1 to n do
    {      if near(k) != 0  and cost(k, near(k)) > cost(k,j)
            then near(k) <- j
    }
    return min_cost
 }
} End Prim's
```

Kruskal’s Method

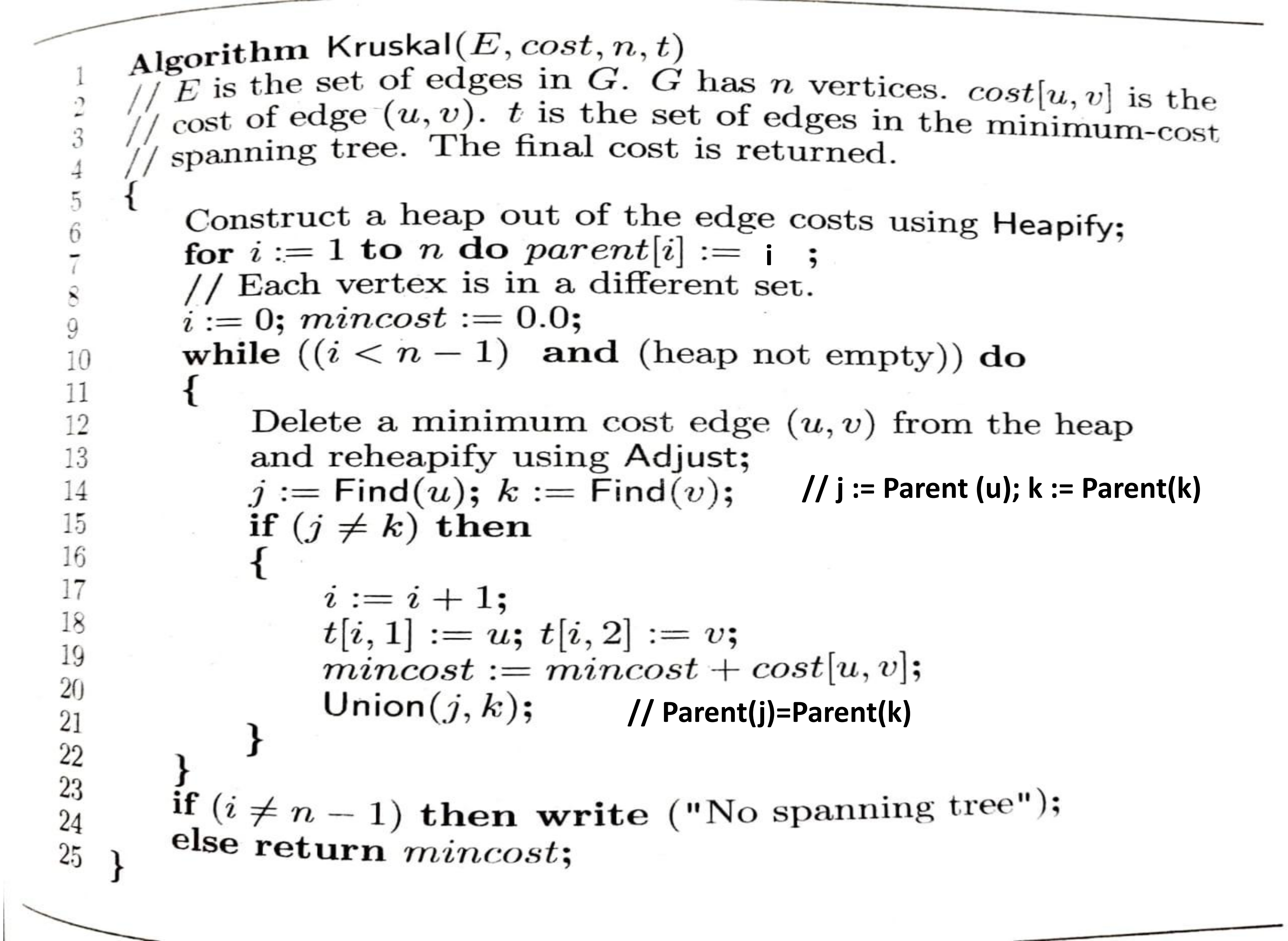
- In Kruskal’s method, edges are added in non-decreasing order of their weights provided that each time an edge is added, at least one new vertex gets added to the tree
- Thus, we start with the minimum weight edge and add the two end vertices to the spanning tree along with the edge
- Next, another minimum weight edge is chosen provided that it creates no cycle in the tree
- We continue in this manner until all the vertices are added to the tree at which point the algorithm stops and there are n-1 edges present in the tree

- Kruskal’s (G)

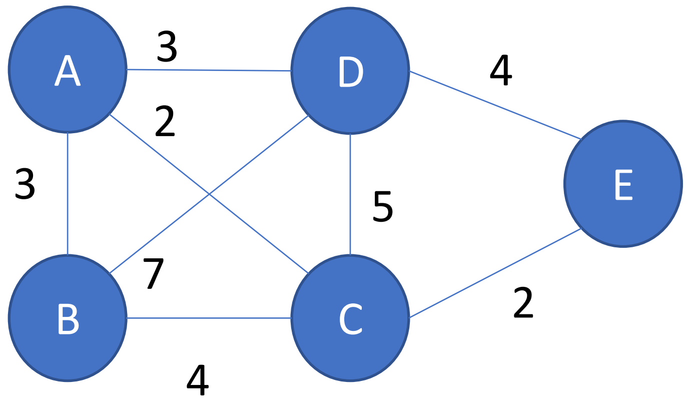
```
{     T <- $\phi$
      while T contains fewer than (n-1) edges and $E \neq \phi$ do
      {   choose an edge (u,w) from E
          if   (u,w) doesn’t create a cycle in T then add (u,w) to T
          else  discard (u,w)
      }
      End while
}
```

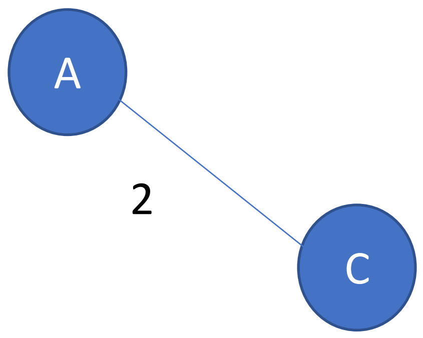
Edges are assumed to occur in sorted order

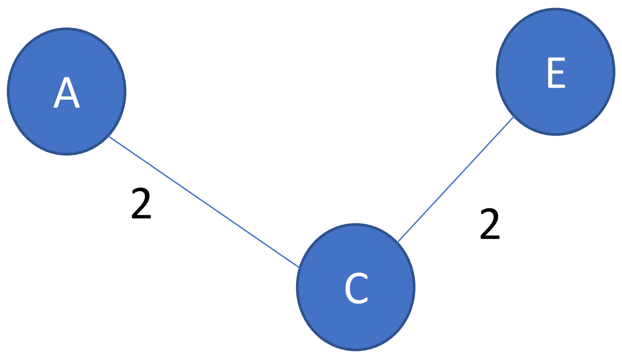
Use a function named ID to assign vertex set header to check if two edges are in the same set

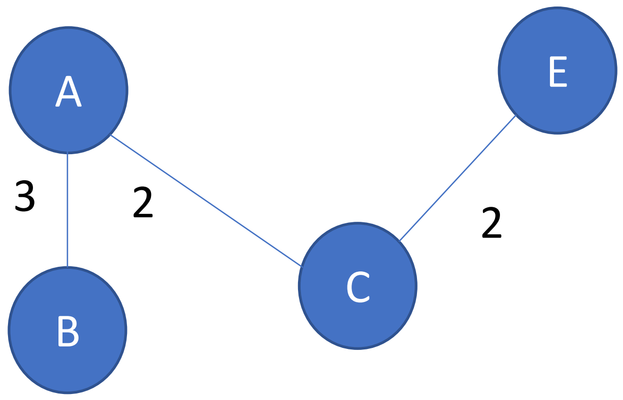


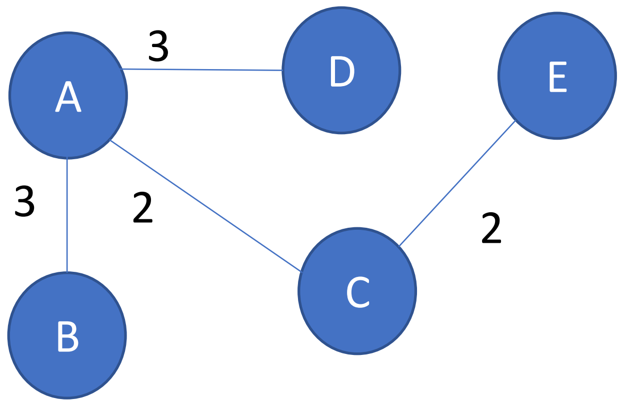
```
Algorithm Kruskal($E, cost, n, t$)
// $E$ is the set of edges in $G$. $G$ has $n$ vertices. $cost[u, v]$ is the
// cost of edge $(u, v)$. $t$ is the set of edges in the minimum-cost
// spanning tree. The final cost is returned.
{
    Construct a heap out of the edge costs using Heapify;
    for $i := 1$ to $n$ do $parent[i] :=$ i ;
    // Each vertex is in a different set.
    $i := 0$; $mincost := 0.0$;
    while (($i < n - 1$) and (heap not empty)) do
    {
        Delete a minimum cost edge $(u, v)$ from the heap
        and reheapify using Adjust;
        $j :=$ Find($u$); $k :=$ Find($v$);      // j := Parent (u); k := Parent(k)
        if ($j \neq k$) then
        {
            $i := i + 1$;
            $t[i, 1] := u$; $t[i, 2] := v$;
            $mincost := mincost + cost[u, v]$;
            Union($j, k$);      // Parent(j)=Parent(k)
        }
    }
    if ($i \neq n - 1$) then write ("No spanning tree");
    else return $mincost$;
}
```


A
3
D
4
2
3
5
E
7
B
C
2
4


A
2
C


A
E
2
2
C


A
E
3
2
2
C
B


3
D
E
A
3
2
2
C
B

Shortest Path

- The graph is a weighted graph and our aim is to find out a path with shortest minimum weight
- Also known as least-cost path
- Weight could depict anything like distance, ease of movement or any other special constraint
- Two types of algorithms are there:
  - Single Source Shortest Path (shortest path between a given node to all other nodes)
  - All Pair Shortest Path (shortest path between any pair of nodes)

Shortest Path

- The graph is a weighted graph and out aim is to find out a path with shortest minimum weight
- Also known as least-cost path
- Weight could depict anything like distance, ease of movement or any other special constraint
- Two types of algorithms are there:
    - Single Source Shortest Path (shortest path between a given node to all other nodes)
    - All Pair Shortest Path (shortest path between any pair of nodes)

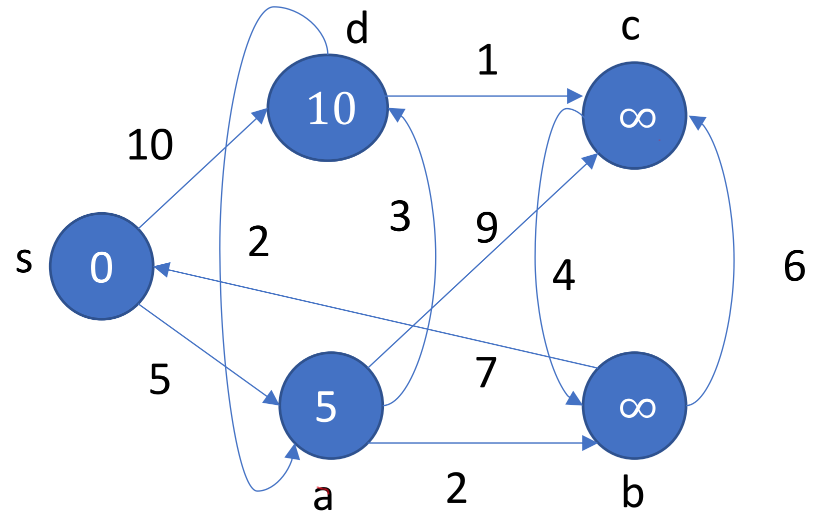
Dijkstra’s Algorithm

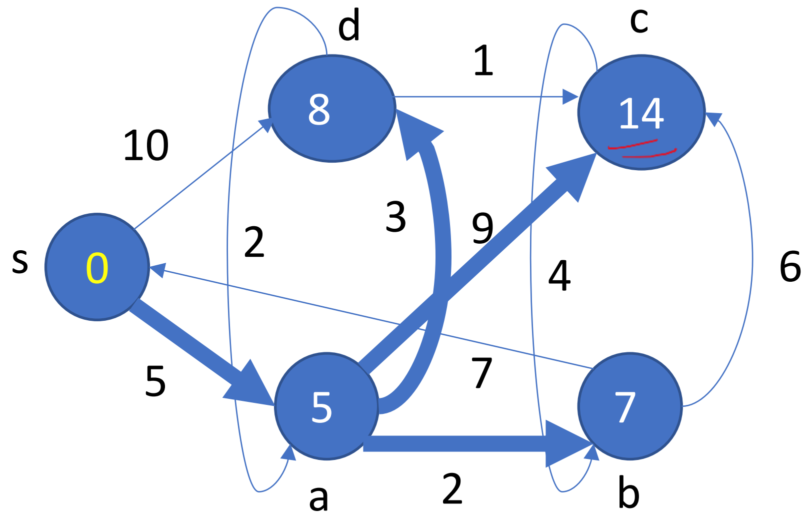
- This algorithm will compute the shortest path from source vertex (v) to all other vertices
- A cost matrix A will be designed such that
  - If an edge of weight w exists between vertices i and j then A[i,j] = w
  - otherwise, A[i,j] = ∞, this helps in making sure that this path is never taken
- Greedy Algorithm
- Suppose S denotes the set of vertices to which shortest path has already been found
- For w ∉ S, Let DIST(w) be the length of the shortest path starting from v going only through vertices that are in S and ending at w
- Here, the graph has n vertices numbered from 1 to n and S is an array such that S(i) = 0 means vertex i is not in S and S(i)=1 means vertex i is in S

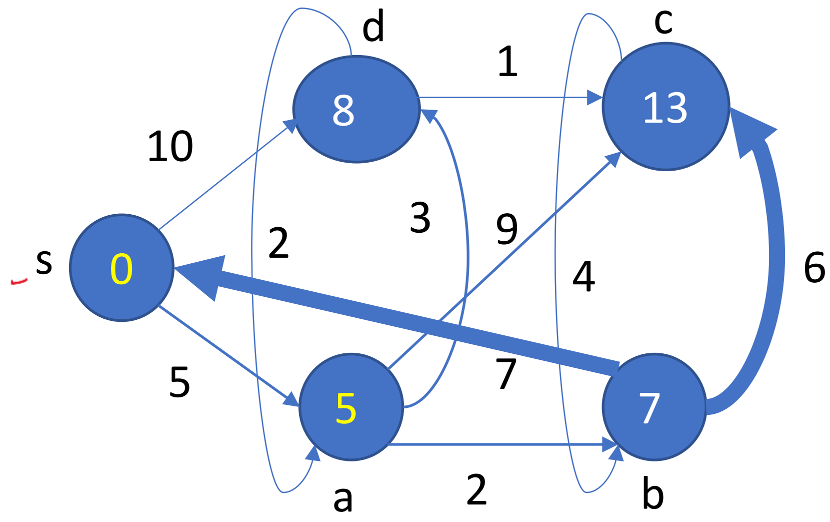
```
Dijkstra's (v, cost, DIST, n)
{
    for i <- 1 to n     do
    s(i) <- 0 ,   DIST(i) = cost(v,i),

      DIST(v) <- 0
      S(v) <- 1
      for i <- 1 to n
      {
              choose u such that  DIST(u) = min(DIST(u)) & s(u) = 0

              s(u) <- 1;

              for( all w adjacent to u with S(w) =0) do
              {
                  DIST(w) <- min(DIST(w), DIST(u) + cost(u,w))
              }
       }
}
```

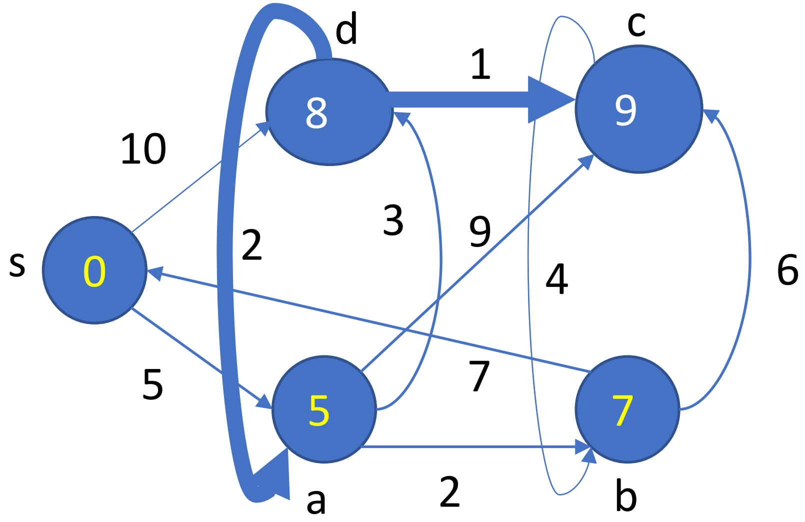
DIST -> array , 1:n
where DIST(1) = distance
from source to 1 and so on

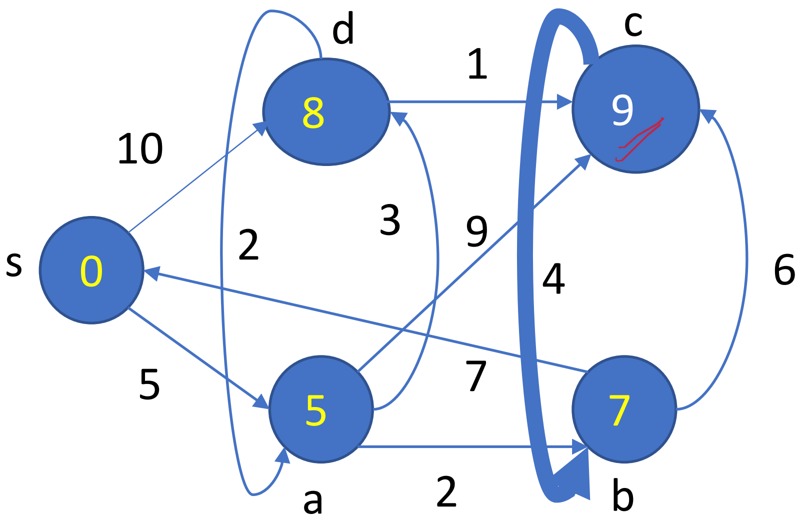
cost: edge, cost n: n


d
c
1
10
10
∞
s
0
2
3
9
4
6
5
7
5
∞
a
2
b

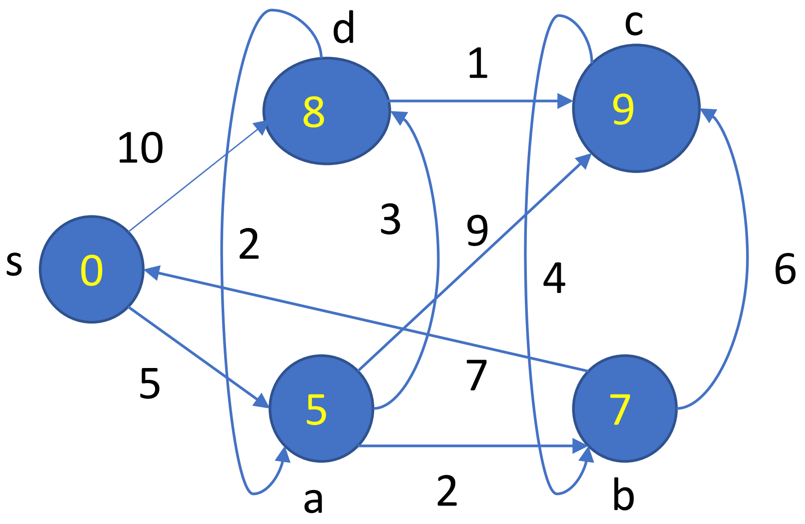

d
c
1
8
14
10
s
0
2
3
9
4
6
5
7
5
7
a
2
b


d
c
1
8
13
10
s
0
2
3
9
4
6
5
7
5
7
a
2
b


d
c
1
8
9
10
s
0
2
3
9
4
6
5
7
5
7
a
2
b



s
0
d
8
c
9
a
5
b
7
10
5
2
3
9
1
4
7
6
2



s
0
d
8
c
9
a
5
b
7
10
5
2
3
9
1
4
7
6
2

Few Important Points

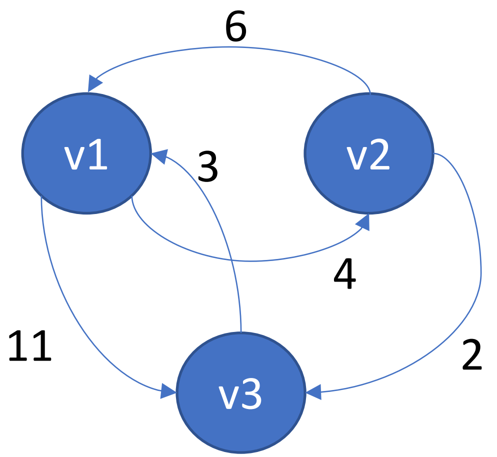
- Dijkstra’s algorithm takes order of n^2 time for a graph with n vertices
- This algorithm doesn’t work for negative weight assignments
- For dealing with negative weight edges another algorithm called Bellman Ford algorithm is used. This algorithm also supposes that there are no negative weight cycles (wherein the problem is unsolvable)
- Time complexity of Dijkstra’s algorithm can be reduced with the use of better data and storage structures
- Dijkstra’s algorithm is based on Greedy technique

Floyd’s Algorithm: All Pair Shortest Path

- Finds shortest path between all pairs of vertices in a graph
- Useful in applications such as writing the shortest distance between any two cities in an Atlas
- Based on Dynamic Programming technique
- It follows a bottom-up approach
- Requires an nxn cost matrix as input for calculating the shortest distance
- Allows negative weight edges

- Dijikstra’s method can be applied n times in order to find all pair shortest path
- Another method is to use dynamic programming, here restriction is less as negative weights on edges is allowed
- However, we assume that there is no negative weight cycle
- Let us examine a shortest i to j path in $G, i \neq j$
- This path originates at vertex i goes through some intermediate vertices and terminates at vertex j.
- If k is an intermediate vertex on this path then the subpaths from $i\ to\ k$ and $k\ to\ j$ must be shortest paths from $i\ to\ k$ and $k\ to\ j$
- Hence, the principal of optimality holds and therefore dynamic programming can be applied

```
procedure
{   for i <- 1:n  do
      {    for j <- 1:n do
               A(i,j) <- cost(i,j)
       }
    for k <- 1:n do
    {     for i <- 1:n do
          {    for j <- 1:n do
                    A(i,j) <- min(A(i,j), {A(i,k)+A(k,j)})
           }
     }
}
```



$$\begin{array}{c|ccc} & v1 & v2 & v3 \\ \hline v1 & 0 & 4 & 11 \\ v2 & 6 & 0 & 2 \\ v3 & 3 & \infty & 0 \end{array} \qquad A[0]$$

$$\begin{array}{c|ccc} & v1 & v2 & v3 \\ \hline v1 & 0 & 4 & 11 \\ v2 & 6 & 0 & 2 \\ v3 & 3 & 7 & 0 \end{array} \qquad A[1]$$

$$\begin{array}{c|ccc} & v1 & v2 & v3 \\ \hline v1 & 0 & 4 & 6 \\ v2 & 6 & 0 & 2 \\ v3 & 3 & 7 & 0 \end{array} \qquad A[2]$$

$$\begin{array}{c|ccc} & v1 & v2 & v3 \\ \hline v1 & 0 & 4 & 6 \\ v2 & 5 & 0 & 2 \\ v3 & 3 & 7 & 0 \end{array}$$

A[3]

$$A^{k}(i,j) = \min\{A^{k-1}(i,j), A^{k-1}(i,k) + A^{k-1}(k,j)\}, \qquad \text{k} \geq 1$$