Dijkstra’s Single Shortest pair Algorithm:

Adjacency matrix stores cost and distance.

Distance[w] = shortest path known between sources (s) and w

If there is no edge to a vertex the cost is ∞

Pred[w]: the vertex immediately before w on the path from the source to W

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph{     // Initialization
      dist[v] ← INFINITY         // Unknown distance from source to v
      prev[v] ← UNDEFINED // Previous node in optimal path from source
      add v to Q       // All nodes initially in Q (unvisited nodes)
       }

    dist[source] ← 0   // Distance from source to source

    while Q is not empty{
        u ← vertex in Q with min dist[u] // Source node will be selected first
        remove u from Q

        for each neighbor v of u: {          // where v is still in Q.
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:    // A shorter path to v has been found
                dist[v] ← alt
                prev[v] ← u
           }
   }

    return dist[], prev[]
```

Pseudo code using Min-Priority Queue.

```
For vertices w ε V {
    Distance[w] = C[s, w]
Pred [w] = S
}
While q is not empty
{
    U = get Min(q)
    For each neighbor v of u
        If distance [v] + C[u, v] < distance [v]
        {
        Distance[v] = distance[u] + C[u,v]
        Pred[v] = u
        Update q -> priority of v
        }
}
```

The time complexity is O(|E|log|v|+|v|

The time complexity

Fibonacci Heap is used to implement a min-priority queue which allows the algorithm to run in O(|E| + |V|log|V|)

Fibonacci Heap properties:

Better amortized running time than many other priority queue data structures including binary heap and binomial heap .

Finding minimum operation takes constant O(1)

Insert and decrease key operation in constant amortized time .

Deleting an element works in O(log n) amortized time.

Calculating all of the functions together yields a worst case scenario of O(a + b log n) compared to a binary or binomial heap which yields O((a + b)log n) time.

The following shows Fibonacci time complexity broken down and compared to other heaps:

Operation	**Binary**[6]	**Binomial**[6]	**Fibonacci**[6]	**Pairing**[7]	**Brodal**[8][a]	**Rank-pairing**[10]	**Strict Fibonacci**[11]
find-min	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
delete-min	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$[b]	$O(\log n)$[b]	$O(\log n)$	$O(\log n)$[b]	$O(\log n)$
insert	$\Theta(\log n)$	$\Theta(1)$[b]	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
decrease-key	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$[b]	$o(\log n)$[b][c]	$\Theta(1)$	$\Theta(1)$[b]	$\Theta(1)$
merge	$\Theta(n)$	$O(\log n)$[d]	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

Pseudo code of Dijkstra's Algorithm implementing a priority queue:

```
function Dijkstra(Graph, source):
        dist[source] ← 0                                  // Initialization

        create vertex set Q
```

```
    for each vertex v in Graph:
        if v ≠ source
            dist[v] ← INFINITY                          // Unknown distance from source to v
            prev[v] ← UNDEFINED                         // Predecessor of v

        Q.add_with_priority(v, dist[v])


    while Q is not empty:                              // The main loop
        u ← Q.extract_min()                            // Remove and return best vertex
        for each neighbor v of u:                       // only v that is still in Q
            alt = dist[u] + length(u, v)
            if alt < dist[v]
                dist[v] ← alt
                prev[v] ← u
                Q.decrease_priority(v, alt)

    return dist[], prev[]
```

All Pairs shortest path

Dijakstra's Algorithm with V

Dynamic Programming is a solution to the all pairs shortest path.

Think fibocci recursive but there is a problem of a stack over flow. Recomputing the same value many times.

F[0] = 0 and F[1] = 1

For (I = 1 to 100) F[i] = F[I -1] + F[I -2] //working bottom up

This is called memorization: storing answers in memory.

ShortestPsath(I, j, K+1) = min(shortestPath(I,j,k), shortestPath(I,K+1,k) + shortestPath (k+1,j,k))

The wrong way to tackle this problem is

```
For I = 1 to n
        For j = 1 to n
                For k = 1 to n //intermediate edge.
```

The correct methods:

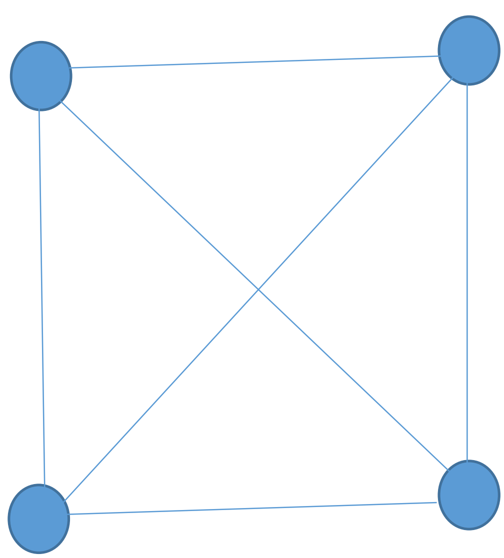
```
For I = 1 to n {                            //initialization
        For j = 1 to n {
                D[I,j] =  c[I,j]
                P[I;j]= i
        }
}
For  k = 1 to n
        For I = 1 to n
                For j = 1 to n
                        { if (d[I,k] + d[k,j] < d[I,j])) {
                                        D[I,j] = d[I,k] + d[k,j]
                                        Pred[I,j] = p[k,j]
                                }
                        }
```

```
let dist be a |V| × |V| array of minimum distances initialized to ∞ (infinity)
for each vertex v
   dist[v][v] ← 0
```

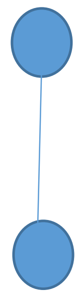
```
for each edge (u,v)
   dist[u][v] ← w(u,v)  // the weight of the edge (u,v)
for k from 1 to |V|
   for i from 1 to |V|
      for j from 1 to |V|
         if dist[i][j] > dist[i][k] + dist[k][j]
             dist[i][j] ← dist[i][k] + dist[k][j]
         end if
```

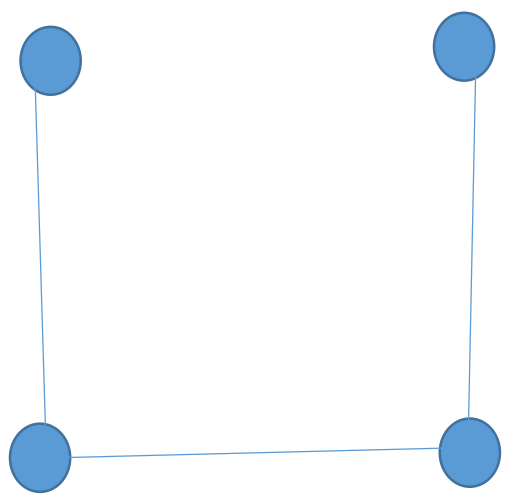
Minium Spanning Trees: Connecting each vertices in a graph using the least edges possible. If weighted using the least cost effect edge (or weight) to connect all vertices.

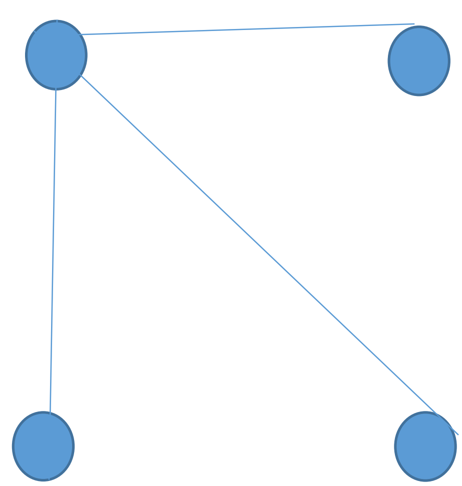
k- 4 complete graph



a tree but not a spanning tree

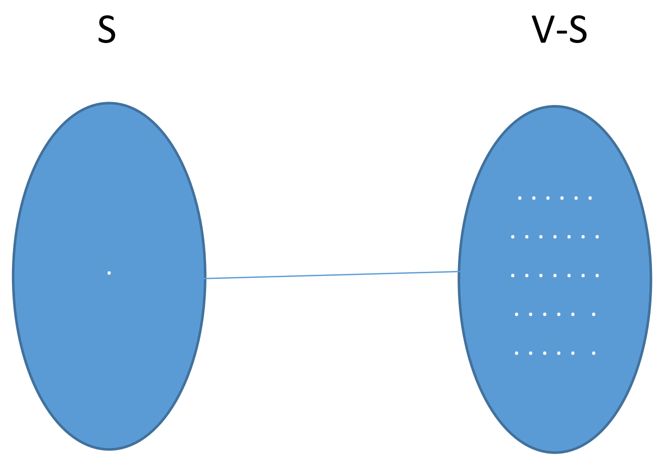






Prim’s Algorithm

Separate verticies into sets



For I = 1 to n

minCost[i] = C[s,i]

closest[i] = s

kruskal’s algorithm

for I = 2 to n

find closest minimum of minCost for vertices still in vis

j= Find closest minimum of mun Cost for vertices still in v-s

for each neighbors k of j in v –s

if monCost[k] > c[j,k]{

minCost[k] = c[j,k]

closest [k] = j

Complextiy time O(|v|^2 = |v|^2)

|V|log|v| + |E|log|v|