Shortest Path

Dijkstra- Greedy

https://www.youtube.com/watch?v=XB4MIexjvY0

On directed and undirected graphs

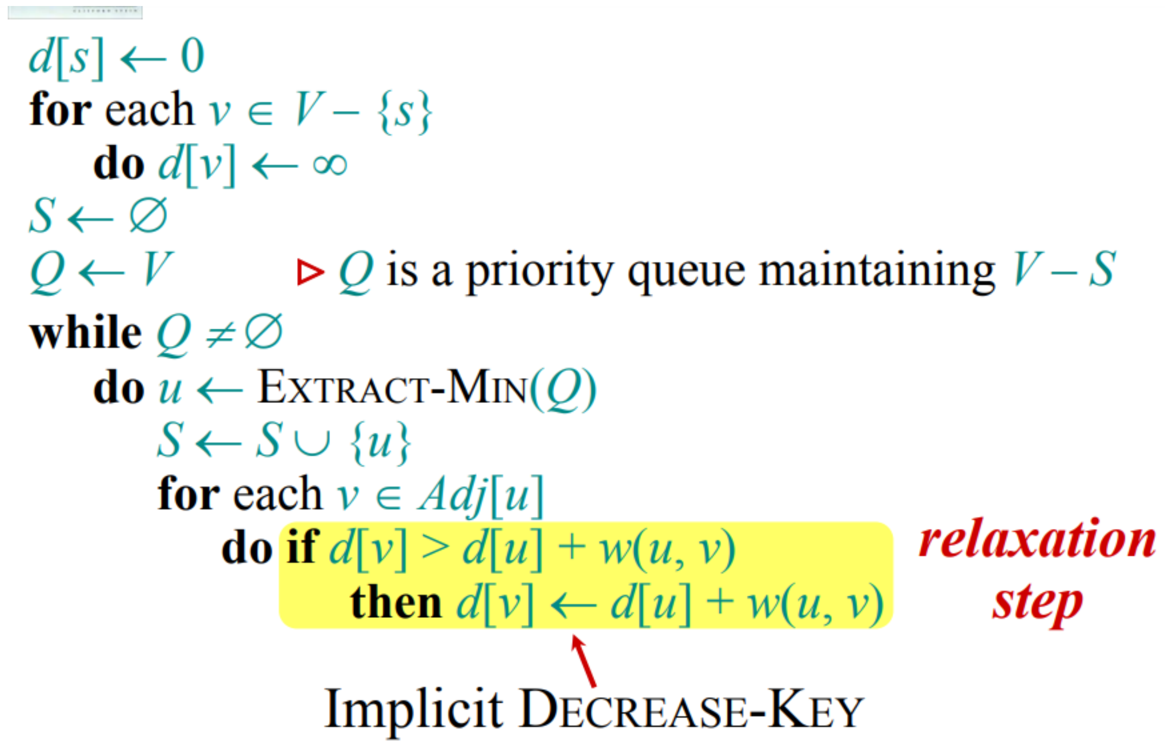
Worst case: O(V^2)

Gets the shortest path of each vertex from the source vertex.

- Gets the smallest weighted edge.
- looks at the connected vertices (connected directed if the graph is directed)
- checks which of the weighted edges are smaller
- Applies relaxation on the vertex of the small edge
- Keeps doing this until it reaches the last vertex in the graph (its gone through all the vertices)

Dijkstra doesn't take account for negative weight edges so if we try to use it on negative weight edges we might give a wrong answer; it may or may not work

Pseudo-code:

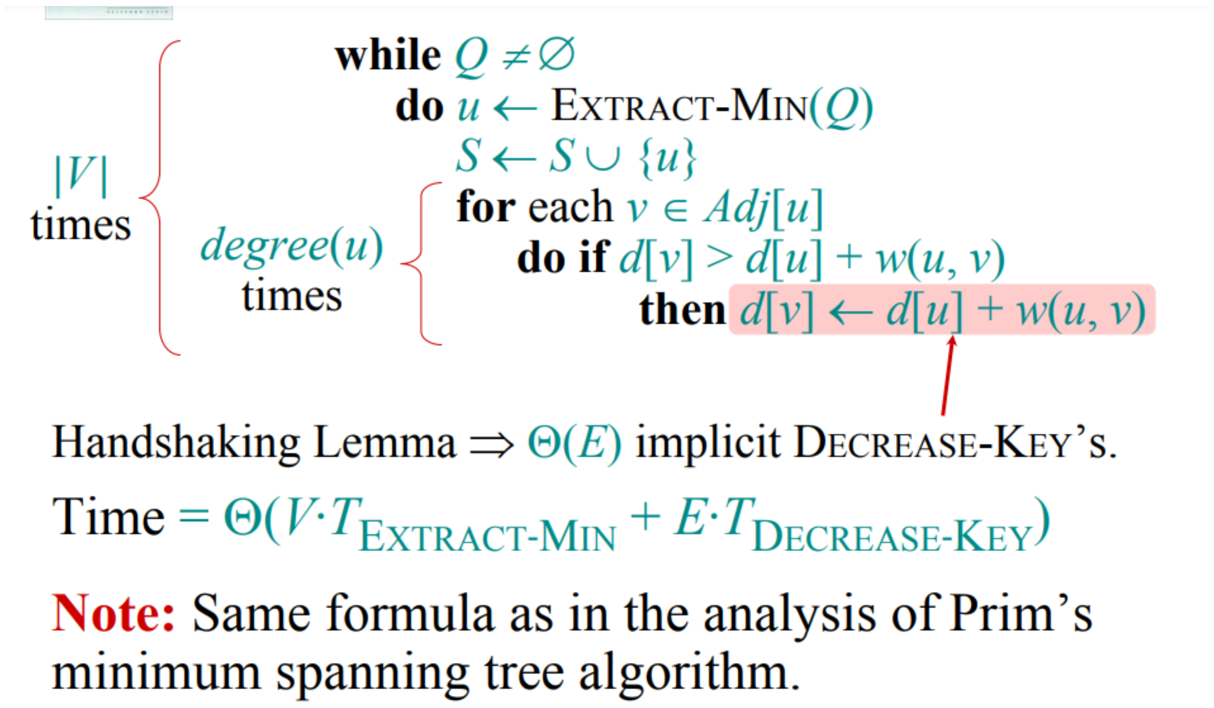


```
$d[s] \leftarrow 0$
for each $v \in V - \{s\}$
    do $d[v] \leftarrow \infty$
$S \leftarrow \emptyset$
$Q \leftarrow V$            $\triangleright$ $Q$ is a priority queue maintaining $V - S$
while $Q \neq \emptyset$
    do $u \leftarrow$ EXTRACT-MIN($Q$)
        $S \leftarrow S \cup \{u\}$
        for each $v \in Adj[u]$
            do if $d[v] > d[u] + w(u, v)$
                then $d[v] \leftarrow d[u] + w(u, v)$
```

relaxation step

Implicit DECREASE-KEY

Analysis of Pesudo-code:



```
            while $Q \neq \emptyset$
                do $u \leftarrow$ EXTRACT-MIN($Q$)
                    $S \leftarrow S \cup \{u\}$
                    for each $v \in Adj[u]$
                        do if $d[v] > d[u] + w(u, v)$
                            then $d[v] \leftarrow d[u] + w(u, v)$
```

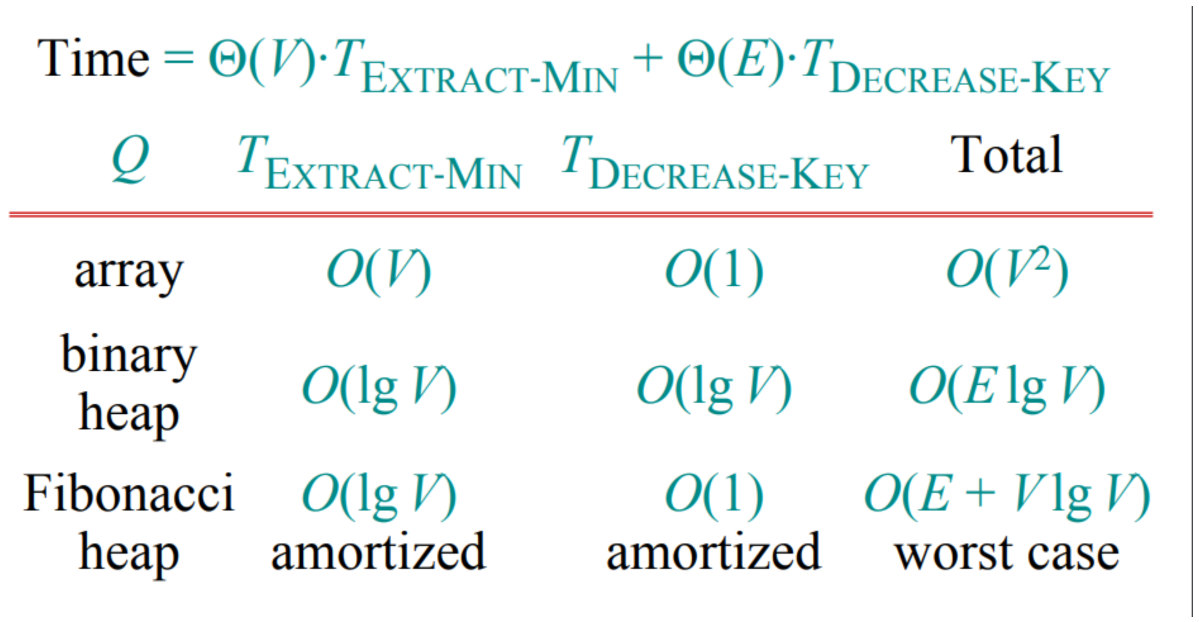
$|V|$ times

degree(u) times

Handshaking Lemma $\Rightarrow \Theta(E)$ implicit DECREASE-KEY's.

Time = $\Theta(V \cdot T_{\text{EXTRACT-MIN}} + E \cdot T_{\text{DECREASE-KEY}})$

Note: Same formula as in the analysis of Prim's minimum spanning tree algorithm.

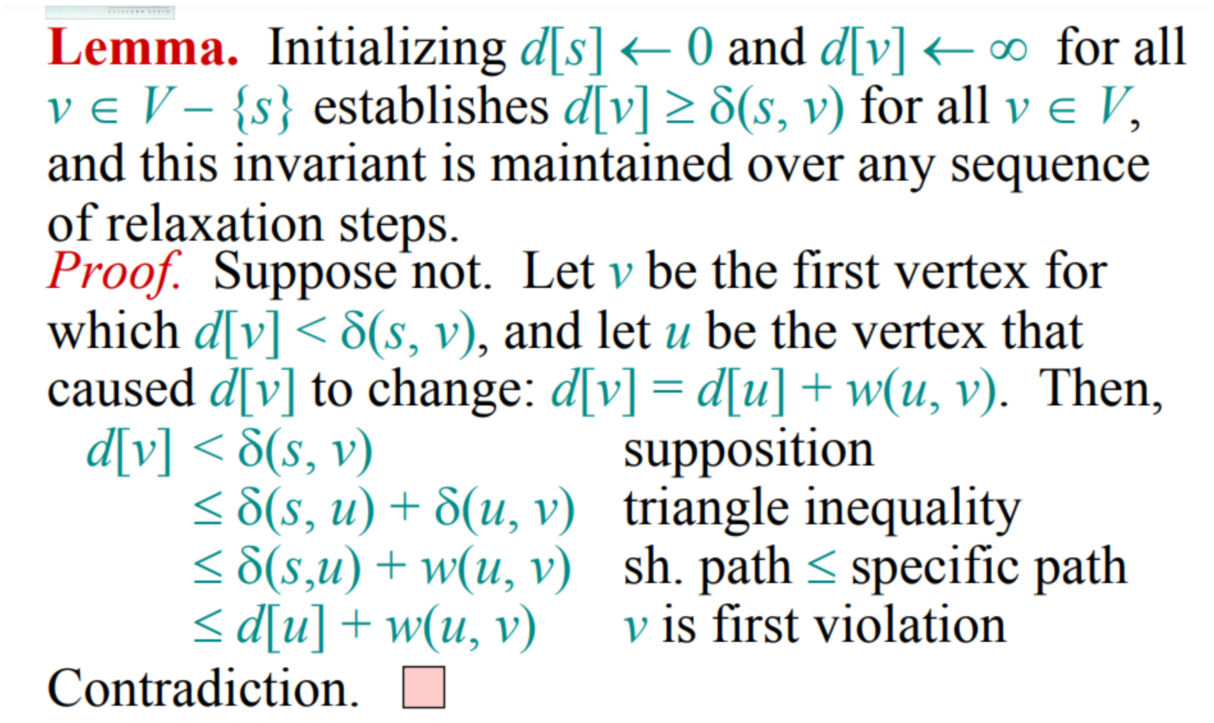


$$\text{Time} = \Theta(V)\cdot T_{\text{EXTRACT-MIN}} + \Theta(E)\cdot T_{\text{DECREASE-KEY}}$$

Q	$T_{\text{EXTRACT-MIN}}$	$T_{\text{DECREASE-KEY}}$	Total
array	$O(V)$	$O(1)$	$O(V^2)$
binary heap	$O(\lg V)$	$O(\lg V)$	$O(E \lg V)$
Fibonacci heap	$O(\lg V)$ amortized	$O(1)$ amortized	$O(E + V \lg V)$ worst case

THEOREMES:

Theorem 1:



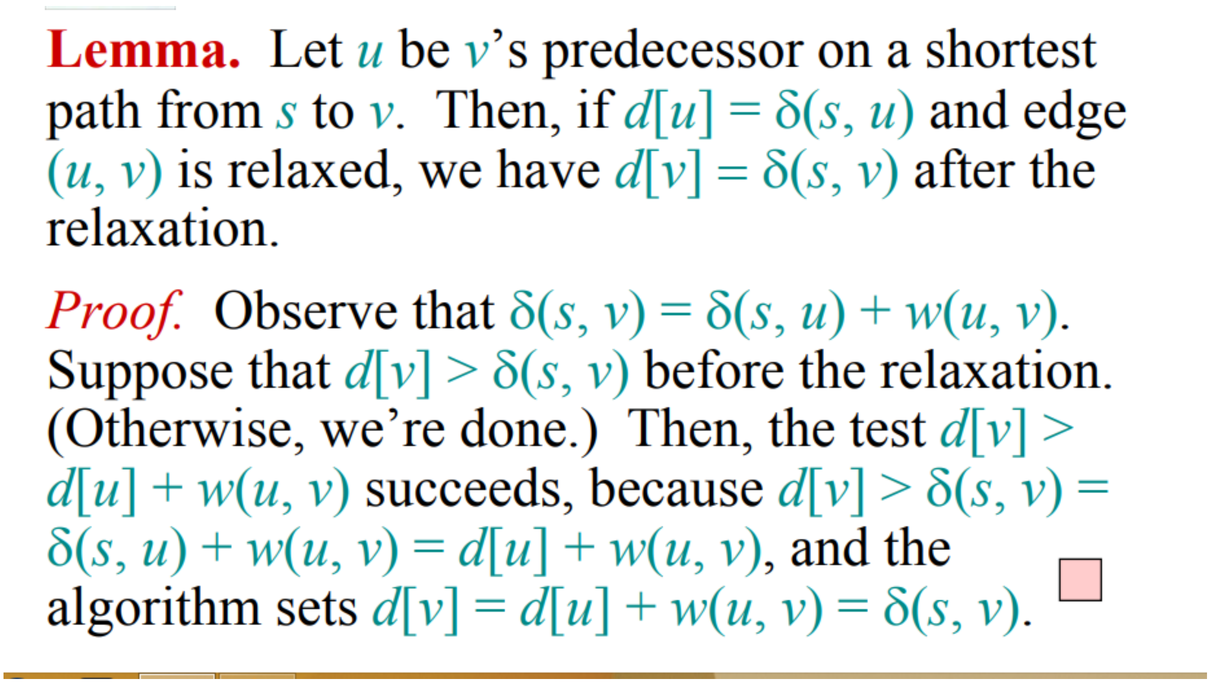
Lemma. Initializing $d[s] \leftarrow 0$ and $d[v] \leftarrow \infty$ for all $v \in V - \{s\}$ establishes $d[v] \geq \delta(s, v)$ for all $v \in V$, and this invariant is maintained over any sequence of relaxation steps.

Proof. Suppose not. Let v be the first vertex for which $d[v] < \delta(s, v)$, and let u be the vertex that caused $d[v]$ to change: $d[v] = d[u] + w(u, v)$. Then,

$d[v] < \delta(s, v)$		supposition
	$\leq \delta(s, u) + \delta(u, v)$	triangle inequality
	$\leq \delta(s,u) + w(u, v)$	sh. path $\leq$ specific path
	$\leq d[u] + w(u, v)$	v is first violation

Contradiction. $\square$

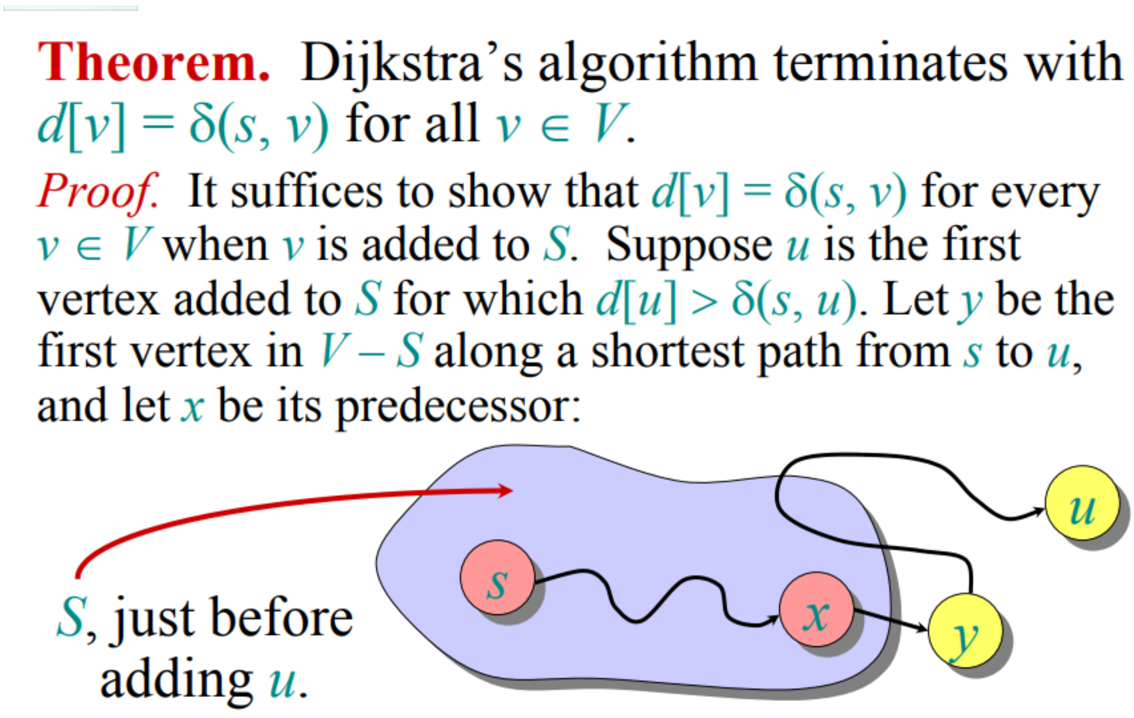
Theorem 2:



Lemma. Let u be v's predecessor on a shortest path from s to v. Then, if $d[u] = \delta(s, u)$ and edge (u, v) is relaxed, we have $d[v] = \delta(s, v)$ after the relaxation.

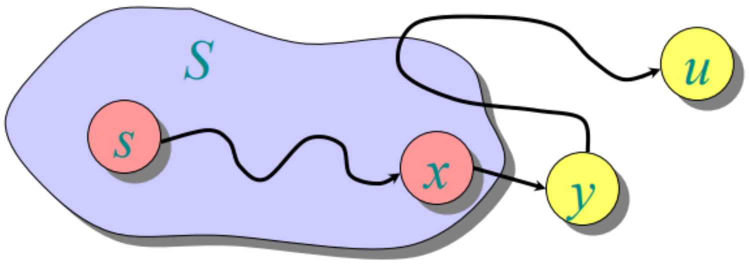
Proof. Observe that $\delta(s, v) = \delta(s, u) + w(u, v)$. Suppose that $d[v] > \delta(s, v)$ before the relaxation. (Otherwise, we're done.) Then, the test $d[v] > d[u] + w(u, v)$ succeeds, because $d[v] > \delta(s, v) = \delta(s, u) + w(u, v) = d[u] + w(u, v)$, and the algorithm sets $d[v] = d[u] + w(u, v) = \delta(s, v)$. $\square$

Theorem 3:



Theorem. Dijkstra's algorithm terminates with $d[v] = \delta(s, v)$ for all $v \in V$.

Proof. It suffices to show that $d[v] = \delta(s, v)$ for every $v \in V$ when v is added to S. Suppose u is the first vertex added to S for which $d[u] > \delta(s, u)$. Let y be the first vertex in $V - S$ along a shortest path from s to u, and let x be its predecessor:



Since u is the first vertex violating the claimed invariant, we have $d[x] = \delta(s, x)$. When x was added to S, the edge (x, y) was relaxed, which implies that $d[y] = \delta(s, y) \le \delta(s, u) < d[u]$. But, $d[u] \le d[y]$ by our choice of u. Contradiction. □

Improving Dikstra:

Suppose that $w(u, v) = 1$ for all $(u, v) \in E$. Can Dijkstra's algorithm be improved?

- Use a simple FIFO queue instead of a priority queue.

Breadth-first search

```
while $Q \neq \emptyset$
    do $u \leftarrow$ DEQUEUE($Q$)
        for each $v \in Adj[u]$
            do if $d[v] = \infty$
                then $d[v] \leftarrow d[u] + 1$
                     ENQUEUE($Q, v$)
```

Analysis: Time = $O(V + E)$.

Key idea: The FIFO Q in breadth-first search mimics the priority queue Q in Dijkstra.

- **Invariant:** v comes after u in Q implies that d[v] = d[u] or d[v] = d[u] + 1.

CODE:

```
public class Dijkstra {
static class Node implements Comparator<Node>{
int node, cost;
Node() {}
Node(int node, int cost){
this.node = node;
this.cost = cost;
}
public int compare(Node o1, Node o2) {
    if (o1.cost < o2.cost)
      return -1;
    if (o1.cost > o2.cost)
      return 1;
    return 0;
    }
  }
private int [] distance;
private Set<Integer> settled;
private PriorityQueue<Dijkstra.Node> q;
private int v;
List<List<Dijkstra.Node>> adj;
public Dijkstra(int v) {
  this.v = v;
  distance = new int [v];
  settled = new HashSet<Integer>();
  q = new PriorityQueue<Node>(v, new Node());
}

void dij(List<List<Node>> adj, int source) {
  this.adj = adj;
  for(int i = 0 ; i < v; i++)
    distance[i] = Integer.MAX_VALUE;
  q.add(new Node(source, 0));
  distance[source] = 0 ;
  while(settled.size() != v ) {
    if(q.isEmpty())
      return;
    int minimum = q.remove().node;
    if(settled.contains(minimum))
      continue;
    settled.add(minimum);
    neigh(minimum);
    }
  }

private void neigh(int minimum) {
  int Edist = -1;
  int Ndist = -1;
  for(int i = 0 ;i< adj.get(minimum).size(); i++) {
    Node n = adj.get(minimum).get(i);
    if(!settled.contains(n.node)) {
      Edist = n.cost;
      Ndist = distance[minimum] + Edist;
      if(Ndist < distance[n.node]) {
        distance[n.node] = Ndist;
        q.add(new Node(n.node, distance[n.node]));
```

```
          }
        }
      }
    }

  public static void main(String [] args) {

    Scanner sc=new Scanner(System.in);
    int v = sc.nextInt();
    int source = sc.nextInt() ;
    List<List<Node>> adj = new ArrayList<List<Node>>();
    for(int i = 0 ; i <v ; i++) {
      List<Node> item = new ArrayList<Node>();
      adj.add(item);
      }

    for(int i=0;i<v+1;i++) {
      int x=sc.nextInt();
      int y=sc.nextInt();
      adj.get(source).add(new Node(x,y));
      }

    Dijkstra q = new Dijkstra(v);
    q.dij(adj, source);
    System.out.println("The shortest path from node: ");
    for(int i =0; i<q.distance.length;  i++)
      System.out.println(source+" to "+i+" is "+q.distance[i]);
    sc.close();
  }
  }
```

Bellmond Ford- DP

works for negative weights unlike dijkstra

we apply relaxation on all vertices v-1 times

- mark the source distance 0 and all the other vertex distances as infinity.
- start relaxing the edges
  - select the nearest edge
  - 0 (source vertex)+ weighted edge<infinity so change the distance of the vertex to 0+wieghted edge
  - then after all those around the source vertex are done do the nearest to the relaxed vertex
- apply relaxation again
- apply relaxation v-1 times

Time complexity: O(EV)=O(n^2)

In worst case (complete graphs): O(n^3)

DRAWBACK: IF THE SUM OF A CYCLE IN A NEGATIVE WEIGHTED GRAPH IS NEGATIVE THEN BELLMOND FORD DOESN'T WORK

Dijkstra doesn't work for Graphs with negative weight cycle*, Bellman-Ford works for such graphs. Bellman-Ford is also simpler than Dijkstra and suites well for distributed systems. But time complexity of Bellman-Ford is O(VE), which is more than Dijkstra.*

Input: Graph and a source vertex *src*

Output: Shortest distance to all vertices from *src*

If there is a negative weight cycle, then shortest distances are not calculated, negative weight cycle is reported.

How the code should work:

1. This step initializes distances from the source to all vertices as infinite and distance to the source itself as 0. Create an array dist[] of size |V| with all values as infinite except dist[src] where src is source vertex.
2. This step calculates shortest distances. Do following |V|-1 times where |V| is the number of vertices in given graph……
    - Do following for each edge u-v
        - If dist[v] > dist[u] + weight of edge uv, then update dist[v]
        - dist[v] = dist[u] + weight of edge uv
3. This step reports if there is a negative weight cycle in graph. Do following for each edge u-v
    - If dist[v] > dist[u] + weight of edge uv, then "Graph contains negative weight cycle"

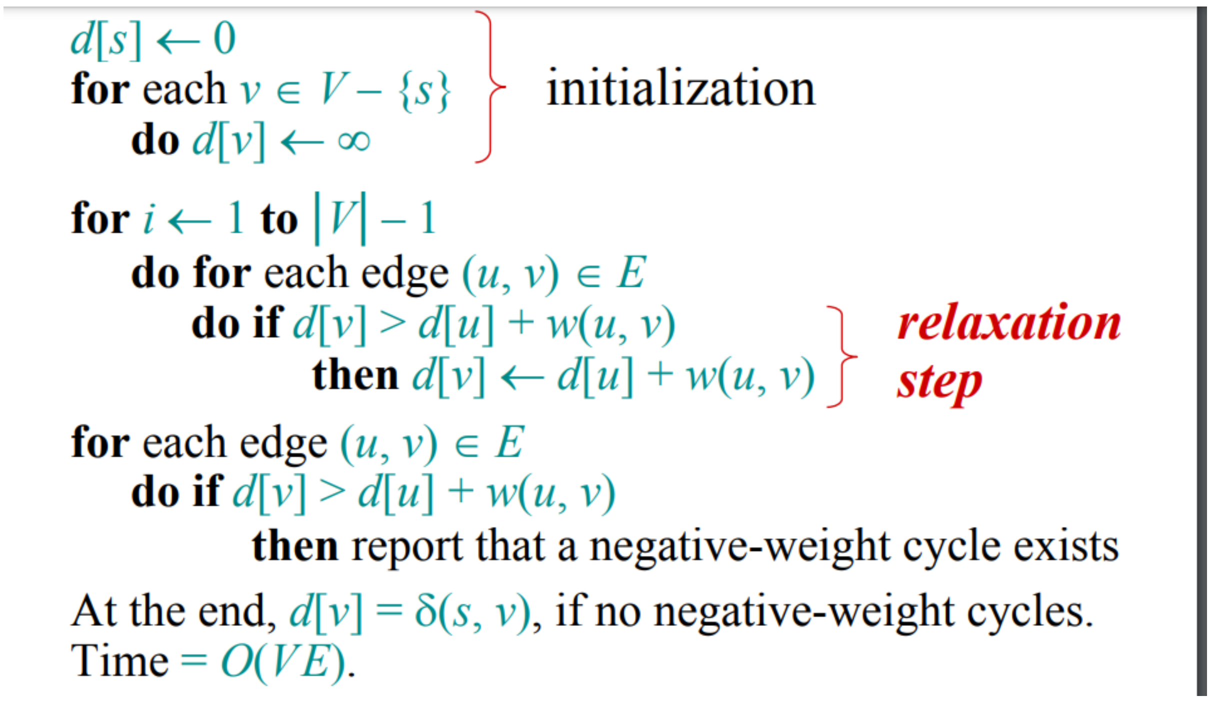
The idea of step 3 is, step 2 guarantees the shortest distances if the graph doesn't contain a negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle

How does this work?

Like other Dynamic Programming Problems, the algorithm calculates shortest paths in a bottom-up manner. It first calculates the shortest distances which have at-most one edge in the path. Then, it calculates the shortest paths with at-most 2 edges, and so on. After the i-th iteration of the outer loop, the shortest paths with at most i edges are calculated. There can be maximum |V| – 1 edges in any simple path, that is why the outer loop runs |v| – 1 times. The idea is, assuming that there is no negative weight cycle, if we have calculated shortest paths with at most i edges, then an iteration over all edges guarantees to give shortest path with at-most (i+1) edges.

Bellman-Ford algorithm: Finds all shortest-path lengths from a source s ∈ V to all v ∈ V or determines that a negative-weight cycle exists.

Pseudo-code:



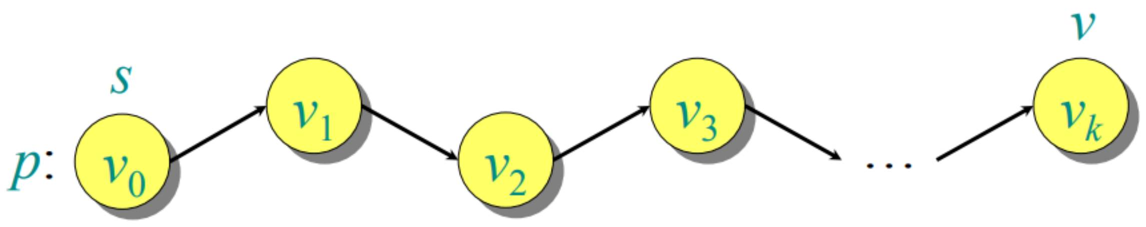
```
$d[s] \leftarrow 0$                                   } initialization
for each $v \in V - \{s\}$                            }
    do $d[v] \leftarrow \infty$                       }

for $i \leftarrow 1$ to $|V| - 1$
    do for each edge $(u, v) \in E$
        do if $d[v] > d[u] + w(u, v)$                          } relaxation
                then $d[v] \leftarrow d[u] + w(u, v)$          } step
for each edge $(u, v) \in E$
    do if $d[v] > d[u] + w(u, v)$
            then report that a negative-weight cycle exists
At the end, $d[v] = \delta(s, v)$, if no negative-weight cycles.
Time = $O(VE)$.
```

Theorem:

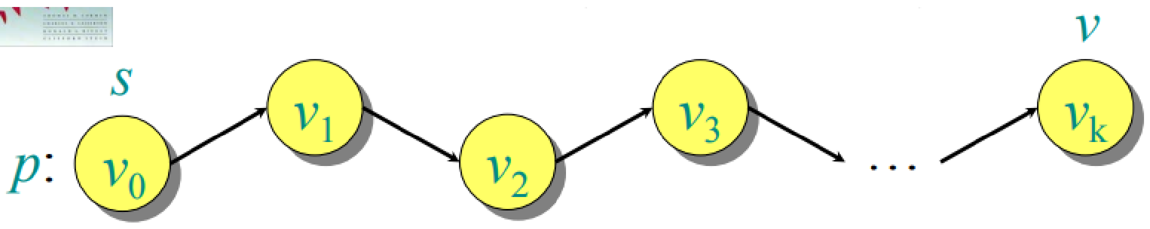
If G = (V, E) contains no negative weight cycles, then after the Bellman-Ford algorithm executes, d[v] = δ(s, v) for all v ∈ V.

Proof. Let $v \in V$ be any vertex, and consider a shortest path p from s to v with the minimum number of edges.



Since p is a shortest path, we have

$$\delta(s, v_i) = \delta(s, v_{i-1}) + w(v_{i-1}, v_i) .$$



Initially, $d[v_0] = 0 = \delta(s, v_0)$, and $d[v_0]$ is unchanged by subsequent relaxations (because of the lemma from Lecture 14 that $d[v] \geq \delta(s, v)$).

- After 1 pass through E, we have $d[v_1] = \delta(s, v_1)$.
- After 2 passes through E, we have $d[v_2] = \delta(s, v_2)$.

  ⋮
- After k passes through E, we have $d[v_k] = \delta(s, v_k)$.

Since G contains no negative-weight cycles, p is simple. Longest simple path has $\leq |V| - 1$ edges. □

Detection of Negative-Weights Cycles

Corollary. If a value d[v] fails to converge after |V| – 1 passes, there exists a negative-weight cycle in G reachable from s.

Unsatisfiable constraints

Theorem. If the constraint graph contains a negative-weight cycle, then the system of differences is unsatisfiable.

Proof. Suppose that the negative-weight cycle is $v_1 \to v_2 \to \cdots \to v_k \to v_1$. Then, we have

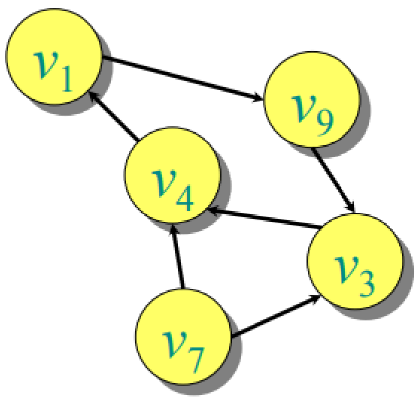
$$\begin{aligned} x_2 - x_1 &\le w_{12} \\ x_3 - x_2 &\le w_{23} \\ &\vdots \\ x_k - x_{k-1} &\le w_{k-1,k} \\ x_1 - x_k &\le w_{k1} \\ \hline 0 &\le \text{weight of cycle} \\ &< 0 \end{aligned}$$

Therefore, no values for the x_i can satisfy the constraints. $\square$

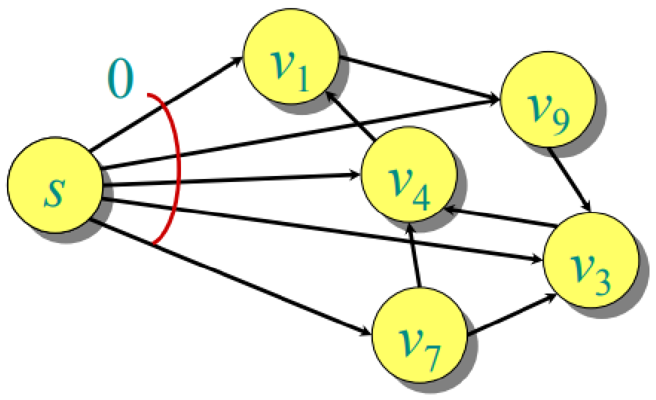
Satisfying the constraints

Theorem. Suppose no negative-weight cycle exists in the constraint graph. Then, the constraints are satisfiable.

Proof. Add a new vertex s to V with a 0-weight edge to each vertex $v_i \in V$.



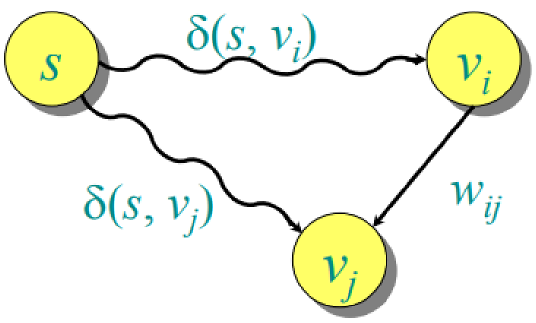
Proof. Add a new vertex s to V with a 0-weight edge to each vertex $v_i \in V$.



Note:
No negative-weight cycles introduced $\Rightarrow$ shortest paths exist.

Claim: The assignment $x_i = \delta(s, v_i)$ solves the constraints.

Consider any constraint $x_j - x_i \le w_{ij}$, and consider the shortest paths from s to v_j and v_i:



The triangle inequality gives us $\delta(s,v_j) \le \delta(s, v_i) + w_{ij}$. Since $x_i = \delta(s, v_i)$ and $x_j = \delta(s, v_j)$, the constraint $x_j - x_i \le w_{ij}$ is satisfied. □

CODE:

```
public class BellmanFord {
       static void BF(int graph[][], int V, int E,int src) {
            int []dis = new int[V];
            for (int i = 0; i < V; i++)
               dis[i] = Integer.MAX_VALUE;
            dis[src] = 0;
            for (int i = 0; i < V - 1; i++)
              for (int j = 0; j < E; j++)
                 if (dis[graph[j][0]] != Integer.MAX_VALUE &&
                     dis[graph[j][0]] + graph[j][2] < dis[graph[j][1]])
```

```
                        dis[graph[j][1]] = dis[graph[j][0]] + graph[j][2];

            for (int i = 0; i < E; i++){
                int x = graph[i][0];
                int y = graph[i][1];
                int weight = graph[i][2];
                if (dis[x] != Integer.MAX_VALUE && dis[x] + weight < dis[y])
                System.out.println("Graph contains negative weight cycle");
             }

          System.out.println("Vertex Distance from Source");
          for (int i = 0; i < V; i++)
           System.out.println(i + "\\t\\t" + dis[i]);
    }
```

Floyd-Warshall: DP

The Floyd Warshall algorthim is for solving the All Pairs Shortest Path problem. The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph.

Gets the distance for every vertex form the source to every other vertex and apply this to all the vertexes.

Pseodu-code:

```
initialize  A[i][j] to E[i][j]
for k = 1 to n do
   for i = 1 to n do
      for j = 1 to n do
        if A[i][j]>A[i][k]+A[k][v]
            A[i][j]=A[i][k]+A[k][j];
```

CODE:

```
import java.util.*;
public class AllPairsShortestPath {
static int v, INF = 99999;
AllPairsShortestPath(int v) {
    AllPairsShortestPath.v = v;
}
void FW(int M[][]) {
    int[][] distance= new int [v][v];
    for (int i = 0 ; i < v; i++)
    for (int j = 0 ; j < v; j++)
    distance[i][j] = M[i][j];
    for(int k = 0 ; k < v ; k++)
       for(int i = 0 ; i < v ; i++)
          for(int j = 0 ; j < v; j++)
               if(distance[i][k] + distance[k][j] < distance[i][j])
                    print(distance);
}
void print(int [][] distance) {
   for(int i = 0; i < v; i++) {
        for(int j = 0; j< v; j++) {
            if(distance[i][j]== INF)
               System.out.print("INF ");
               else
                   System.out.print(distance[i][j] + "   ");
}
System.out.println();
}
}

public static void main (String [] args) {
  Scanner s= new Scanner(System.in);
  AllPairsShortestPath a = new AllPairsShortestPath(s.nextInt());
  int M[][] = new int[v][v];
  for(int i =0; i<v; i++)
      for(int j = 0 ; j < v; j++)
         M[i][j] = s.nextInt();
         a.FW(M);
}
}
```