Greedy Algorithm Introduction

"Greedy Method finds out of many options, but you have to choose the best option."

In this method, we have to find out the best method/option out of many present ways.

In this approach/method we focus on the first stage and decide the output, don't think about the future.

This method may or may not give the best output.

13.8M

275

Java Try Catch

Greedy Algorithm solves problems by making the best choice that seems best at the particular moment. Many optimization problems can be determined using a greedy algorithm. Some issues have no efficient solution, but a greedy algorithm may provide a solution that is close to optimal. A greedy algorithm works if a problem exhibits the following two properties:

1. **Greedy Choice Property:** A globally optimal solution can be reached at by creating a locally optimal solution. In other words, an optimal solution can be obtained by creating "greedy" choices.
2. **Optimal substructure:** Optimal solutions contain optimal subsolutions. In other words, answers to subproblems of an optimal solution are optimal.

Example: (red color r in syllabus)

1. machine scheduling
2. Fractional Knapsack Problem
3. Minimum Spanning Tree(PRIMS and KRUSKAL)
4. Huffman Code
5. Job Sequencing with deadline
6. Activity Selection Problem
7. Source to Destination using shortest path (dikjastra)

Steps for achieving a Greedy Algorithm are:

1. **Feasible:** Here we check whether it satisfies all possible constraints or not, to obtain at least one solution to our problems.
2. **Local Optimal Choice:** In this, the choice should be the optimum which is selected from the currently available
3. **Unalterable:** Once the decision is made, at any subsequence step that option is not altered.

Fractional Knapsack

Fractions of items can be taken rather than having to make binary (0-1) choices for each item.

Fractional Knapsack Problem can be solvable by greedy strategy whereas 0 - 1 problem is not.

Steps to solve the Fractional Problem:

1. Compute the value per pound Fractional Knapsack Problem for each item.
2. Obeying a Greedy Strategy, we take as possible of the item with the highest value per pound.
3. If the supply of that element is exhausted and we can still carry more, we take as much as possible of the element with the next value per pound.
4. If Sorting, the items by value per pound, the greedy algorithm run in O (n log n) time. Otherwise O(nxw)

Fractional Knapsack (Array v, Array w, int W)

1. for i= 1 to size (v)

2. do p [i] = v [i] / w [i]

3. Sort-Descending (p)

4. i ← 1

5. while (W>0)

6. do amount = min (W, w [i])

7. solution [i] = amount

8. W= W-amount

9. i ← i+1

10. return solution

**Example:** Consider 5 items along their respective weights and values: -

I = (I1,I2,I3,I4,I5)

15.8M

357

How to find Nth Highest Salary in SQL

w = (5, 10, 20, 30, 40)

v = (30, 20, 100, 90,160)

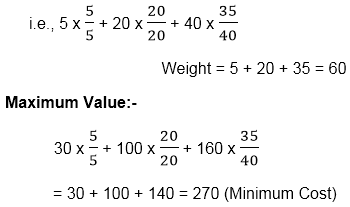
The capacity of knapsack W = 60

Now fill the knapsack according to the decreasing value of pi.

First, we choose the item Ii whose weight is 5.

Then choose item I3 whose weight is 20. Now,the total weight of knapsack is 20 + 5 = 25

Now the next item is I5, and its weight is 40, but we want only 35, so we chose the fractional part of it,



**Solution:**

|  |  |  |
| --- | --- | --- |
| **ITEM** | **wi** | **vi** |
| I1 | 5 | 30 |
| I2 | 10 | 20 |
| I3 | 20 | 100 |
| I4 | 30 | 90 |
| I5 | 40 | 160 |

Taking value per weight ratio i.e. pi=Fractional Knapsack Problem

|  |  |  |  |
| --- | --- | --- | --- |
| **ITEM** | **wi** | **vi** | **Pi=Fractional Knapsack Problem** |
| I1 | 5 | 30 | 6.0 |
| I2 | 10 | 20 | 2.0 |
| I3 | 20 | 100 | 5.0 |
| I4 | 30 | 90 | 3.0 |
| I5 | 40 | 160 | 4.0 |

**Now, arrange the value of pi in decreasing order.**

|  |  |  |  |
| --- | --- | --- | --- |
| **ITEM** | **wi** | **vi** | **pi=Fractional Knapsack Problem** |
| I1 | 5 | 30 | 6.0 |
| I3 | 20 | 100 | 5.0 |
| I5 | 40 | 160 | 4.0 |
| I4 | 30 | 90 | 3.0 |
| I2 | 10 | 20 | 2.0 |

**Job Sequencing With Deadlines-**

The sequencing of jobs on a single processor with deadline constraints is called as Job Sequencing with Deadlines.

Here-

* You are given a set of jobs.
* Each job has a defined deadline and some profit associated with it.
* The profit of a job is given only when that job is completed within its deadline.
* Only one processor is available for processing all the jobs.
* Processor takes one unit of time to complete a job.

**The problem states-**

“How can the total profit be maximized if only one job can be completed at a time?”

**Approach to Solution-**

* A feasible solution would be a subset of jobs where each job of the subset gets completed within its deadline.
* Value of the feasible solution would be the sum of profit of all the jobs contained in the subset.
* An optimal solution of the problem would be a feasible solution which gives the maximum profit.

**Greedy Algorithm-**

Greedy Algorithm is adopted to determine how the next job is selected for an optimal solution.

The greedy algorithm described below always gives an optimal solution to the job sequencing problem-

**Step-01:**

* Sort all the given jobs in decreasing order of their profit.

**Step-02:**

* Check the value of maximum deadline.
* Draw a Gantt chart where maximum time on Gantt chart is the value of maximum deadline.

**Step-03:**

* Pick up the jobs one by one.
* Put the job on Gantt chart as far as possible from 0 ensuring that the job gets completed before its deadline.

**PRACTICE PROBLEM BASED ON JOB SEQUENCING WITH DEADLINES-**

**Problem-**

Given the jobs, their deadlines and associated profits as shown-

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Jobs** | **J1** | **J2** | **J3** | **J4** | **J5** | **J6** |
| **Deadlines** | 5 | 3 | 3 | 2 | 4 | 2 |
| **Profits** | 200 | 180 | 190 | 300 | 120 | 100 |

Answer the following questions-

1. Write the optimal schedule that gives maximum profit.
2. Are all the jobs completed in the optimal schedule?
3. What is the maximum earned profit?

**Solution-**

**Step-01:**

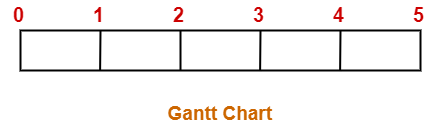
Sort all the given jobs in decreasing order of their profit-

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Jobs** | **J4** | **J1** | **J3** | **J2** | **J5** | **J6** |
| **Deadlines** | 2 | 5 | 3 | 3 | 4 | 2 |
| **Profits** | 300 | 200 | 190 | 180 | 120 | 100 |

**Step-02:**

Value of maximum deadline = 5.

So, draw a Gantt chart with maximum time on Gantt chart = 5 units as shown-

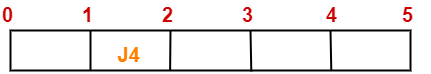


Now,

* We take each job one by one in the order they appear in Step-01.
* We place the job on Gantt chart as far as possible from 0.

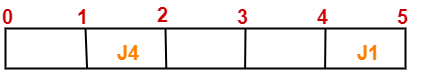
**Step-03:**

* We take job J4.
* Since its deadline is 2, so we place it in the first empty cell before deadline 2 as-



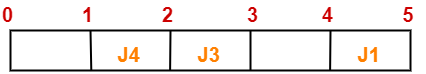
**Step-04:**

* We take job J1.
* Since its deadline is 5, so we place it in the first empty cell before deadline 5 as-



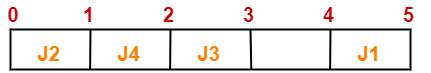
**Step-05:**

* We take job J3.
* Since its deadline is 3, so we place it in the first empty cell before deadline 3 as-



**Step-06:**

* We take job J2.
* Since its deadline is 3, so we place it in the first empty cell before deadline 3.
* Since the second and third cells are already filled, so we place job J2 in the first cell as-



**Step-07:**

* Now, we take job J5.
* Since its deadline is 4, so we place it in the first empty cell before deadline 4 as-



Now,

* The only job left is job J6 whose deadline is 2.
* All the slots before deadline 2 are already occupied.
* Thus, job J6 can not be completed.

Now, the given questions may be answered as-

**Part-01:**

The optimal schedule is-

**J2 , J4 , J3 , J5 , J1**

This is the required order in which the jobs must be completed in order to obtain the maximum profit.

**Part-02:**

* All the jobs are not completed in optimal schedule.
* This is because job J6 could not be completed within its deadline.

**Part-03:**

Maximum earned profit

= Sum of profit of all the jobs in optimal schedule

= Profit of job J2 + Profit of job J4 + Profit of job J3 + Profit of job J5 + Profit of job J1

= 180 + 300 + 190 + 120 + 200

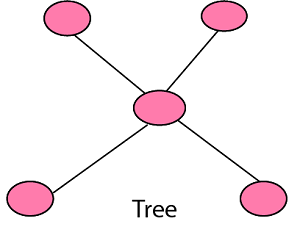
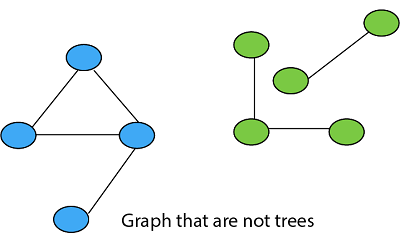
= 990 units

Introduction of Minimum Spanning Tree

Tree:

A tree is a graph with the following properties:

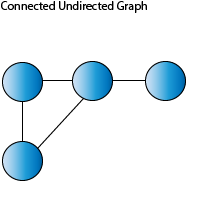
1. The graph is connected (can go from anywhere to anywhere)
2. There are no cyclic (Acyclic)

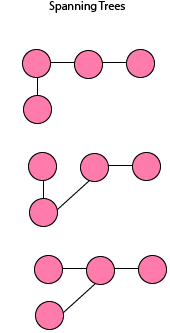
Spanning Tree:

Given a connected undirected graph, a spanning tree of that graph is a subgraph that is a tree and joined all vertices. A single graph can have many spanning trees.

**For Example:**



For the above-connected graph. There can be multiple spanning Trees like



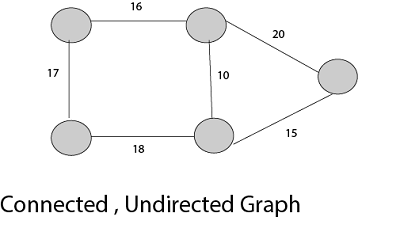
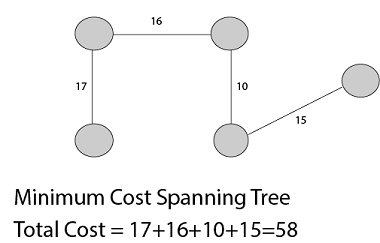
Properties of Spanning Tree:

1. There may be several minimum spanning trees of the same weight having the minimum number of edges.
2. If all the edge weights of a given graph are the same, then every spanning tree of that graph is minimum.
3. If each edge has a distinct weight, then there will be only one, unique minimum spanning tree.
4. A connected graph G can have more than one spanning trees.
5. A disconnected graph can't have to span the tree, or it can't span all the vertices.
6. Spanning Tree doesn't contain cycles.
7. Spanning Tree has **(n-1) edges** where n is the number of vertices.

Addition of even one single edge results in the spanning tree losing its property of **Acyclicity** and elimination of one single edge results in its losing the property of connectivity.

Minimum Spanning Tree:

Minimum Spanning Tree is a Spanning Tree which has minimum total cost. If we have a linked undirected graph with a weight (or cost) combine with each edge. Then the cost of spanning tree would be the sum of the cost of its edges.

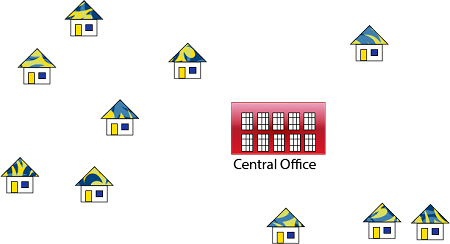
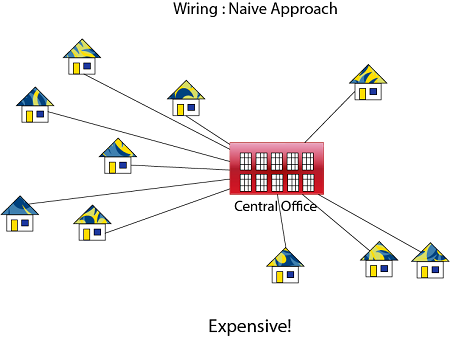
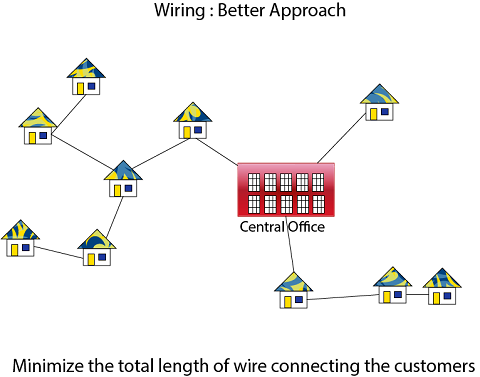
  


Application of Minimum Spanning Tree

1. Consider n stations are to be linked using a communication network & laying of communication links between any two stations involves a cost.  
   The ideal solution would be to extract a subgraph termed as minimum cost spanning tree.
2. Suppose you want to construct highways or railroads spanning several cities then we can use the concept of minimum spanning trees.
3. Designing Local Area Networks.
4. Laying pipelines connecting offshore drilling sites, refineries and consumer markets.
5. Suppose you want to apply a set of houses with
   * Electric Power
   * Water
   * Telephone lines
   * Sewage lines

To reduce cost, you can connect houses with minimum cost spanning trees.

**For Example, Problem laying Telephone Wire.**

Methods of Minimum Spanning Tree

There are two methods to find Minimum Spanning Tree

1. Kruskal's Algorithm
2. Prim's Algorithm

Kruskal's Algorithm:

An algorithm to construct a Minimum Spanning Tree for a connected weighted graph. It is a Greedy Algorithm. The Greedy Choice is to put the smallest weight edge that does not because a cycle in the MST constructed so far.

**If the graph is not linked, then it finds a Minimum Spanning Tree.**

**Steps for finding MST using Kruskal's Algorithm:**

18.6M

347

Difference between JDK, JRE, and JVM

1. Arrange the edge of G in order of increasing weight.
2. Starting only with the vertices of G and proceeding sequentially add each edge which does not result in a cycle, until (n - 1) edges are used.
3. EXIT.

**MST- KRUSKAL (G, w)**

1. A ← ∅

2. for each vertex v ∈ V [G]

3. do MAKE - SET (v)

4. sort the edges of E into non decreasing order by weight w

5. for each edge (u, v) ∈ E, taken in non decreasing order by weight

6. do if FIND-SET (μ) ≠ if FIND-SET (v)

7. then A ← A ∪ {(u, v)}

8. UNION (u, v)

9. return A

**If I have V no of vertices then total no of edge=vC2= v(v-1)/2**

**E**

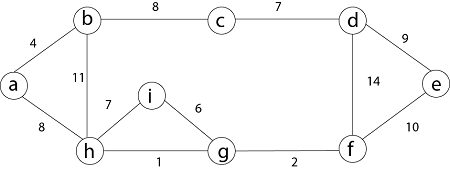
**Analysis:** Where E is the number of edges in the graph and V is the number of vertices, Kruskal's Algorithm can be shown to run in O (E log E) time, or simply, O (E log V) time, all with simple data structures. These running times are equivalent because:

* E is at most V2 and log V2= 2 x log V is O (log V).
* If we ignore isolated vertices, which will each their components of the minimum spanning tree, V ≤ 2 E, so log V is O (log E).

Thus the total time is

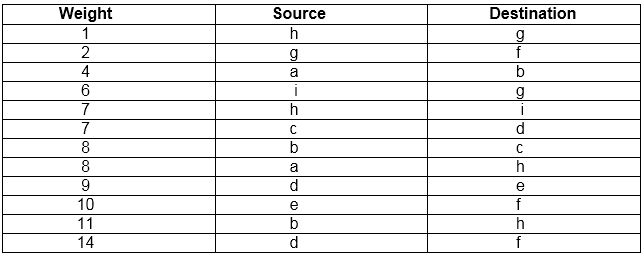
1. O (E log E) = O (E log V).

**For Example:** Find the Minimum Spanning Tree of the following graph using Kruskal's algorithm.



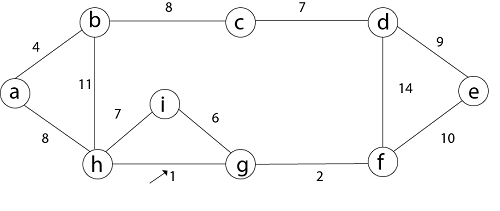
**Solution:** First we initialize the set A to the empty set and create |v| trees, one containing each vertex with MAKE-SET procedure. Then sort the edges in E into order by non-decreasing weight.

There are 9 vertices and 12 edges. So MST formed (9-1) = 8 edges

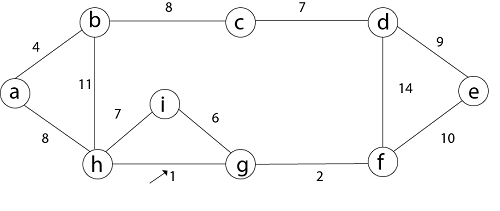


Now, check for each edge (u, v) whether the endpoints u and v belong to the same tree. If they do then the edge (u, v) cannot be supplementary. Otherwise, the two vertices belong to different trees, and the edge (u, v) is added to A, and the vertices in two trees are merged in by union procedure.

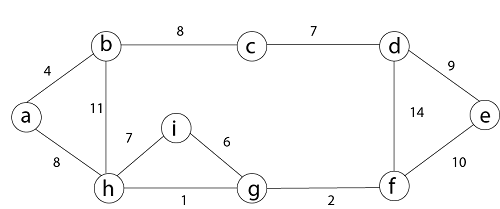
**Step1:** So, first take (h, g) edge



**Step 2:** then (g, f) edge.

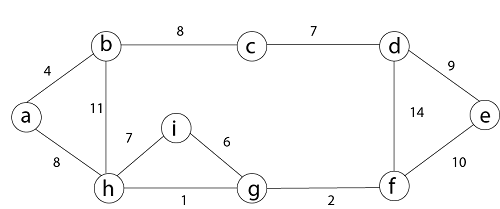


**Step 3:** then (a, b) and (i, g) edges are considered, and the forest becomes



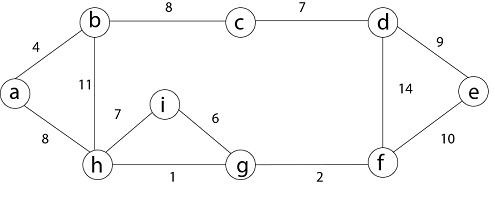
**Step 4:** Now, edge (h, i). Both h and i vertices are in the same set. Thus it creates a cycle. So this edge is discarded.

        Then edge (c, d), (b, c), (a, h), (d, e), (e, f) are considered, and the forest becomes.



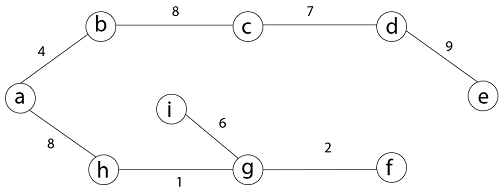
**Step 5:** In (e, f) edge both endpoints e and f exist in the same tree so discarded this edge. Then (b, h) edge, it also creates a cycle.

**Step 6:** After that edge (d, f) and the final spanning tree is shown as in dark lines.



**Step 7:** This step will be required Minimum Spanning Tree because it contains all the 9 vertices and (9 - 1) = 8 edges

1. e → f,  b → h,  d → f [cycle will be formed]



Minimum Cost MST

# Prim's Algorithm

It is a greedy algorithm. It starts with an empty spanning tree. The idea is to maintain two sets of vertices:

* Contain vertices already included in MST.
* Contain vertices not yet included.

At every step, it considers all the edges and picks the minimum weight edge. After picking the edge, it moves the other endpoint of edge to set containing MST.

### Steps for finding MST using Prim's Algorithm:

1. Create MST set that keeps track of vertices already included in MST.
2. Assign key values to all vertices in the input graph. Initialize all key values as INFINITE (∞). Assign key values like 0 for the first vertex so that it is picked first.
3. While MST set doesn't include all vertices.
   1. Pick vertex u which is not is MST set and has minimum key value. Include 'u'to MST set.
   2. Update the key value of all adjacent vertices of u. To update, iterate through all adjacent vertices. For every adjacent vertex v, if the weight of edge u.v less than the previous key value of v, update key value as a weight of u.v.

**MST-PRIM (G, w, r)**

1. for each u ∈ V [G]

2. do key [u] ← ∞

3. π [u] ← NIL

4. key [r] ← 0

5. Q ← V [G]

6. While Q ? ∅

7. do u ← EXTRACT - MIN (Q)

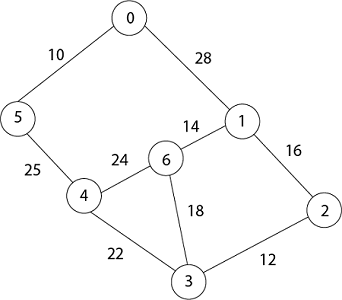
8. for each v ∈ Adj [u]

9. do if v ∈ Q and w (u, v) < key [v]

10. then π [v] ← u

11. key [v] ← w (u, v)

**Example:** Generate minimum cost spanning tree for the following graph using Prim's algorithm.

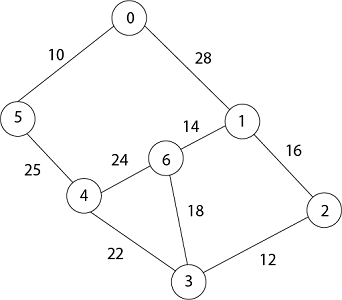


**Solution:** In Prim's algorithm, first we initialize the priority Queue Q. to contain all the vertices and the key of each vertex to ∞ except for the root, whose key is set to 0. Suppose 0 vertex is the root, i.e., r. By EXTRACT - MIN (Q) procure, now u = r and Adj [u] = {5, 1}.

Removing u from set Q and adds it to set V - Q of vertices in the tree. Now, update the key and π fields of every vertex v adjacent to u but not in a tree.

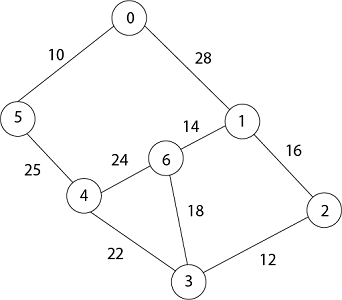


1. Taking 0 as starting vertex
2. Root = 0
3. Adj [0] = 5, 1
4. Parent, π [5] = 0 and π [1] = 0
5. Key [5] = ∞ and key [1] = ∞
6. w [0, 5) = 10  and w (0,1) = 28
7. w (u, v) < key [5] , w (u, v) < key [1]
8. Key [5] = 10 and key [1] = 28
9. So update key value of 5 and 1 is:

Now by EXTRACT\_MIN (Q) Removes 5 because key [5] = 10 which is minimum so u = 5.

1. Adj [5] = {0, 4} and 0 is already in heap
2. Taking 4, key [4] = ∞      π [4] = 5
3. (u, v) < key [v] then key [4] = 25
4. w (5,4) = 25
5. w (5,4) < key [4]
6. date key value and parent of 4.

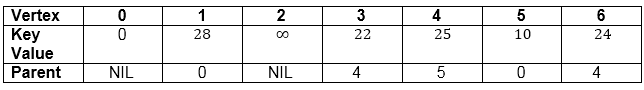
Now remove 4 because key [4] = 25 which is minimum, so u =4

1. Adj [4] = {6, 3}
2. Key [3] = ∞         key [6] = ∞
3. w (4,3) = 22        w (4,6) = 24
4. w (u, v) < key [v]    w (u, v) < key [v]
5. w (4,3) < key [3]      w (4,6) < key [6]

Update key value of key [3] as 22 and key [6] as 24.

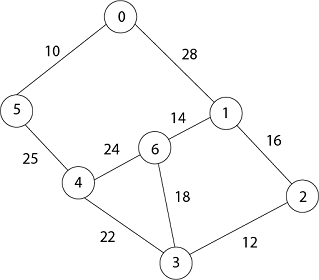
And the parent of 3, 6 as 4.

1. π[3]= 4       π[6]= 4



1. u = EXTRACT\_MIN (3, 6)            [key [3] < key [6]]
2. u = 3              i.e.  22 < 24

Now remove 3 because key [3] = 22 is minimum so u =3.

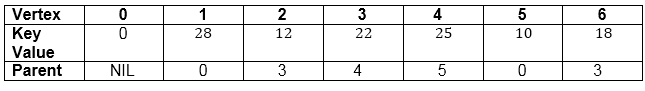


1. Adj [3] = {4, 6, 2}
2. 4 is already in heap
3. 4 ≠ Q key [6] = 24 now becomes key [6] = 18
4. Key [2] = ∞            key [6] = 24
5. w (3, 2) = 12          w (3, 6) = 18
6. w (3, 2) < key [2]         w (3, 6) < key [6]

Now in Q, key [2] = 12, key [6] = 18, key [1] = 28 and parent of 2 and 6 is 3.

1. π [2] = 3      π[6]=3

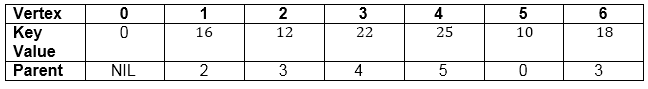
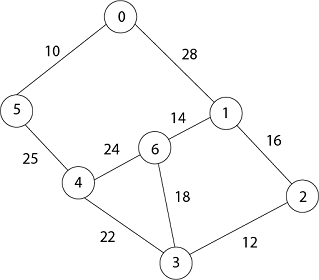
Now by EXTRACT\_MIN (Q) Removes 2, because key [2] = 12 is minimum.



1. u = EXTRACT\_MIN (2, 6)
2. u = 2          [key [2] < key [6]]
3. 12 < 18
4. Now the root is 2
5. Adj [2] = {3, 1}
6. 3 is already in a heap
7. Taking 1, key [1] = 28
8. w (2,1) = 16
9. w (2,1) < key [1]

So update key value of key [1] as 16 and its parent as 2.

1. π[1]= 2

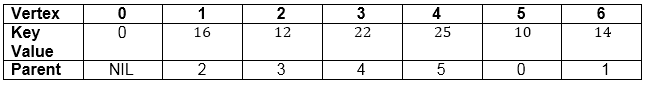
  


Now by EXTRACT\_MIN (Q) Removes 1 because key [1] = 16 is minimum.

1. Adj [1] = {0, 6, 2}
2. 0 and 2 are already in heap.
3. Taking 6, key [6] = 18
4. w [1, 6] = 14
5. w [1, 6] < key [6]

Update key value of 6 as 14 and its parent as 1.

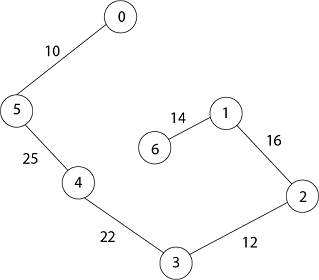
1. Π [6] = 1



Now all the vertices have been spanned, Using above the table we get Minimum Spanning Tree.

1. 0 → 5 → 4 → 3 → 2 → 1 → 6
2. [Because Π [5] = 0, Π [4] = 5, Π [3] = 4, Π [2] = 3, Π [1] =2, Π [6] =1]

**Thus the final spanning Tree is**



**Total Cost = 10 + 25 + 22 + 12 + 16 + 14 = 99**

Single Source Shortest Paths

Introduction:

In a **shortest- paths problem**, we are given a weighted, directed graphs G = (V, E), with weight function **w: E → R** mapping edges to real-valued weights. The weight of path p = (v0,v1,..... vk) is the total of the weights of its constituent edges:

Single Source Shortest Paths

We define the shortest - path weight from u to v by δ(u,v) = min (w (p): u→v), if there is a path from u to v, and δ(u,v)= ∞, otherwise.

The **shortest path** from vertex s to vertex t is then defined as any path p with weight w (p) = δ(s,t).

The **breadth-first- search algorithm** is the shortest path algorithm that works on unweighted graphs, that is, graphs in which each edge can be considered to have unit weight.

13.8M

275

Java Try Catch

In a **Single Source Shortest Paths Problem**, we are given a Graph G = (V, E), we want to find the shortest path from a given source vertex s ∈ V to every vertex v ∈ V.

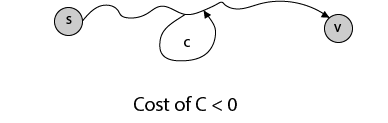
Variants:

There are some variants of the shortest path problem.

* **Single- destination shortest - paths problem:** Find the shortest path to a given destination vertex t from every vertex v. By shift the direction of each edge in the graph, we can shorten this problem to a single - source problem (DIJKASTRA OR BELLMENFORD).
* **Single - pair shortest - path problem:** Find the shortest path from u to v for given vertices u and v. If we determine the single - source problem with source vertex u, we clarify this problem also. Furthermore, no algorithms for this problem are known that run asymptotically faster than the best single - source algorithms in the worst case.
* **All - pairs shortest - paths problem:** Find the shortest path from u to v for every pair of vertices u and v. Running a single - source algorithm once from each vertex can clarify this problem; but it can generally be solved faster, and its structure is of interest in the own right (FLOYD WARSHALL).

Shortest Path: Existence:

If some path from s to v contains a negative cost cycle then, there does not exist the shortest path. Otherwise, there exists a shortest s - v that is simple.



# Negative Weight Edges

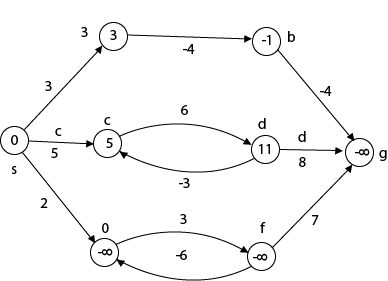
It is a weighted graph in which the total weight of an edge is negative. If a graph has a negative edge, then it produces a chain. After executing the chain if the output is negative then it will give - ∞ weight and condition get discarded. If weight is less than negative and - ∞ then we can't have the shortest path in it.

Briefly, if the output is -ve, then both condition get discarded.

1. - ∞
2. Not less than 0.

And we cannot have the shortest Path.

### Example:



1. Beginning from s
2. Adj [s] = [a, c, e]
3. Weight from s to a is 3

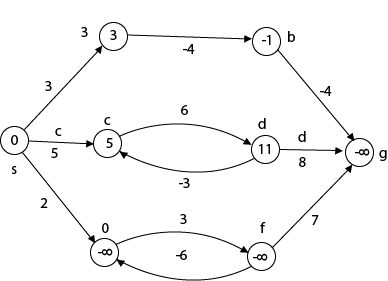
Suppose we want to calculate a path from s→c. So We have 2 paths /weight

Difference between JDK, JRE, and JVM

1. s to c = 5, s→c→d→c=8
2. But s→c is minimum
3. So s→c = 5

Suppose we want to calculate a path from s→e. So we have two paths again

1. s→e = 2,    s→e→f→e=-1
2. As -1 < 0 ∴ Condition gets discarded. If we execute **this** chain, we will get - ∞. So we can't get the shortest path ∴ e = ∞.



This figure illustrates the effects of negative weights and negative weight cycle on the shortest path weights.

Because there is only one path from "s to a" (the path <s, a>), δ (s, a) = w (s, a) = 3.

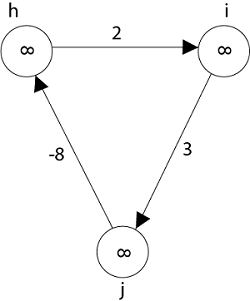
Furthermore, there is only one path from "s to b", so δ (s, b) = w (s, a) + w (a, b) = 3 + (-4) = - 1.

There are infinite many path from "s to c" : <s, c> : <s, c, d, c>, <s, c, d, c, d, c> and so on. Because the cycle <c, d, c> has weight δ (c, d) = w (c, d) + w (d, c) = 6 + (-3) = 3, which is greater than 0, the shortest path from s to c is <s, c> with weight δ (s, c) = 5.

Similarly, the shortest path from "s to d" is <s, c, d> with weight δ (s, d) = w (s, c) + w (s, d) = 11.

Analogously, there are infinite many paths from s to e: <s, e>, <s, e, f, e>, <s, e, f, e, f, e> and so on. Since the cycle <e, f, e> has weight δ (e, f) = w (e, f) + w (f, e) = 3 + (-6) = -3. So - 3 < 0, however there is no shortest path from s to e. B8y traversing the negative weight cycle <e, f, e>. This means path from s to e has arbitrary large negative weights and so δ (s, e) = - ∞.

Similarly δ (s, f) = - ∞ because g is reachable from f, we can also find a path with arbitrary large negative weight from s to g and δ (s, g) = - ∞



Vertices h, i, g also from negative weight cycle. They are also not reachable from the source node, so distance from the source is - ∞ to three of nodes (h, i, j).

Representing: Shortest Path

Given a graph G = (V, E), we maintain for each vertex v ∈ V a **predecessor** π [v] that is either another vertex or NIL. During the execution of shortest paths algorithms, however, the π values need not indicate shortest paths. As in breadth-first search, we shall be interested in the **predecessor subgraph** Gn= (Vn,En) induced by the value π. Here again, we define the vertex set Vπ, to be the set of vertices of G with non - NIL predecessors, plus the source s:

Vπ= {v ∈ V: π [v] ≠ NIL} ∪ {s} }

The directed edge set EΠ is the set of edges induced by the Π values for vertices in VΠ:

EΠ= {(Π[v], v) ∈ E: v ∈ VΠ - {s}}

A **shortest - paths tree** rooted at s is a directed subgraph G = (V' E'), where V'∈ V andE'∈E, such that

1. V' is the set of vertices reachable from s in G
2. G' forms a rooted tree with root s, and
3. For all v ∈ V', the unique, simple path from s to v in G' is the shortest path from s to v in G.

Shortest paths are not naturally unique, and neither is shortest - paths trees.

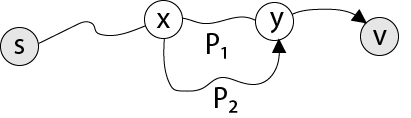
14.6M

293

HTML Tutorial

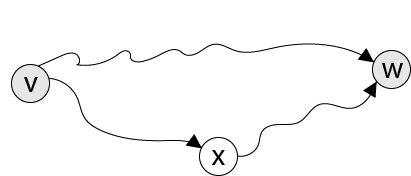
Properties of Shortest Path:

**1. Optimal substructure property:** All subpaths of shortest paths are shortest paths.



Let P1 be x - y sub path of shortest s - v path. Let P2 be any x - y path. Then cost of P1≤ cost of P2,otherwise P not shortest s - v path.

**2. Triangle inequality:** Let d (v, w) be the length of shortest path from v to w. Then,  
d (v, w) ≤ d (v, x) + d (x, w)



**3. Upper-bound property:** We always have d[v] ≥ δ(s, v) for all vertices v ∈ V, and once d[v] conclude the value δ(s, v), it never changes.

**4. No-path property:** If there is no path from s to v, then we regularly have d[v] = δ(s, v) = ∞.

**5. Convergence property:** If s->u->v is a shortest path in G for some u, v ∈ V, and if d[u] = δ(s, u) at any time prior to relaxing edge (u, v), then d[v] = δ(s, v) at all times thereafter.

Relaxation

The single - source shortest paths are based on a technique known as **relaxation**, a method that repeatedly decreases an upper bound on the actual shortest path weight of each vertex until the upper bound equivalent the shortest - path weight. For each vertex v ∈ V, we maintain an attribute d [v], which is an upper bound on the weight of the shortest path from source s to v. We call d [v] the **shortest path estimate**.

**INITIALIZE - SINGLE - SOURCE (G, s)**

1. for each vertex v ∈ V [G]

2. do d [v] ← ∞

3. π [v] ← NIL

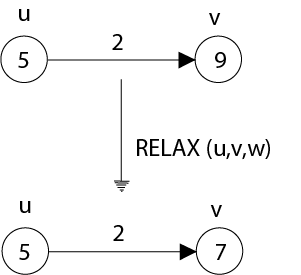
4. d [s] ← 0

After initialization, π [v] = NIL for all v ∈ V, d [v] = 0 for v = s, and d [v] = ∞ for v ∈ V - {s}.

The development of relaxing an edge (u, v) consists of testing whether we can improve the shortest path to v found so far by going through u and if so, updating d [v] and π [v]. A relaxation step may decrease the value of the shortest - path estimate d [v] and updated v's predecessor field π [v].

Fig: Relaxing an edge (u, v) with weight w (u, v) = 2. The shortest-path estimate of each vertex appears within the vertex.

History of Java

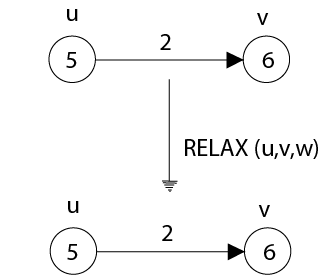
 d(u)=0, w(u,v) or c(u,v)=2, d(v)=infinity

if d(u)+c(u,v)<d(v) then

0+2<infinite

d(v)= d(u)+c(u,v)=2

**(a) Because v. d > u. d + w (u, v) prior to relaxation, the value of v. d decreases**



**(b) Here, v. d < u. d + w (u, v) before relaxing the edge, and so the relaxation step leaves v. d unchanged.**

The subsequent code performs a relaxation step on edge (u, v)

**RELAX (u, v, w)**

1. If d [v] > d [u] + w (u, v)

2. then d [v] ← d [u] + w (u, v)

3. π [v] ← u

Dijkstra's Algorithm

It is a greedy algorithm that solves the single-source shortest path problem for a directed graph G = (V, E) with nonnegative edge weights, i.e., w (u, v) ≥ 0 for each edge (u, v) ∈ E.

Dijkstra's Algorithm maintains a set S of vertices whose final shortest - path weights from the source s have already been determined. That's for all vertices v ∈ S; we have d [v] = δ (s, v). The algorithm repeatedly selects the vertex u ∈ V - S with the minimum shortest - path estimate, insert u into S and relaxes all edges leaving u.

Because it always chooses the "lightest" or "closest" vertex in V - S to insert into set S, it is called as the **greedy strategy**.

**Dijkstra's Algorithm (G, w, s)**

1. INITIALIZE - SINGLE - SOURCE (G, s)

2. S←∅

3. Q←V [G]

4. while Q ≠ ∅

5. do u ← EXTRACT - MIN (Q)

6. S ← S ∪ {u}

7. for each vertex v ∈ Adj [u]

8. do RELAX (u, v, w)

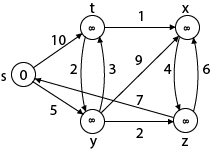
**Analysis:** The running time of Dijkstra's algorithm on a graph with edges E and vertices V can be expressed as a function of |E| and |V| using the Big - O notation. The simplest implementation of the Dijkstra's algorithm stores vertices of set Q in an ordinary linked list or array, and operation Extract - Min (Q) is simply a linear search through all vertices in Q. In this case, the running time is O (|V2 |+|E|=O(V2 ). Final is O(E logV)

14.6M

293

HTML Tutorial

**Example:**



**U-> starting vertex**

**V-> to reach**

**C(u,v) (current) or w(u,v) or c(s,t)=10**

**D(s,t)=infinity.**

**Solution:**

**Step1:** Q =[s, t, x, y, z]

We scanned vertices one by one and find out its adjacent. Calculate the distance of each adjacent to the source vertices.

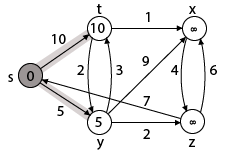
We make a stack, which contains those vertices which are selected after computation of shortest distance.

Firstly we take's' in stack M (which is a source)

1. M = [S]       Q = [t, x, y, z]

**Step 2:** Now find the adjacent of s that are t and y.

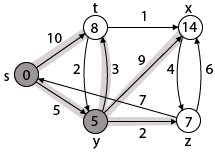
1. Adj [s] → t, y      [Here s is u and t and y are v]



**Case v=s, v=t - (i)**s → t  
                d [v] < d [u] + w [u, v]  
                d [t] < d [s] + w [s, t]  
                ∞ > 0 + 10                [false condition]  
Then       **d [t] ← 10**  
                **π [t] ← 5**  
Adj [s] ← t, y

**Case - (ii)**s→ y  
                d [v] < d [u] + w [u, v]  
                d [y] < d [s] + w [s, y]  
                ∞ > 0 + 5                [false condition]  
                ∞ > 5  
Then       **d [y] ← 5**  
              **π [y] ← 5**

By comparing case (i) and case (ii)  
     Adj [s] → t = 10, y = 5  
     y is shortest  
**y is assigned in 5 = [s, y]**



**Step 3:** Now find the adjacent of y that is t, x, z.

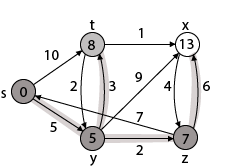
1. Adj [y] → t, x, z   [Here y is u and t, x, z are v]

**Case - (i)** y →t  
              d [v] > d [u] + w [u, v]  
              d [t] > d [y] + w [y, t]  
              10 > 5 + 3  
              10 > 8  
Then     d [t] ← 8  
              π [t] ← y

**Case - (ii)** y → x  
              d [v] > d [u] + w [u, v]  
              d [x] > d [y] + w [y, x]  
              ∞ > 5 + 9  
              ∞ > 14  
Then      d [x] ← 14  
             π [x] ← 14

**Case - (iii)** y → z  
             d [v] > d [u] + w [u, v]  
             d [z] > d [y] + w [y, z]  
             ∞ > 5 + 2  
             ∞ > 7  
Then      d [z] ← 7  
             π [z] ← y

By comparing case (i), case (ii) and case (iii)  
           Adj [y] → x = 14, t = 8, z =7  
z is shortest  
**z is assigned in 7 = [s, z]**

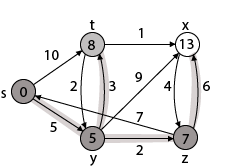


**Step - 4 Now** we will find adj [z] that are s, x

1. Adj [z] → [x, s]    [Here z is u and s and x are v]

**Case - (i)** z → x  
              d [v] > d [u] + w [u, v]  
              d [x] > d [z] + w [z, x]  
              14 > 7 + 6  
              14 > 13  
Then       d [x] ← 13  
              π [x] ← z

**Case - (ii)** z → s  
              d [v] > d [u] + w [u, v]  
              d [s] > d [z] + w [z, s]  
              0 > 7 + 7  
              0 > 14  
∴ This condition does not satisfy so it will be discarded.  
Now we have x = 13.

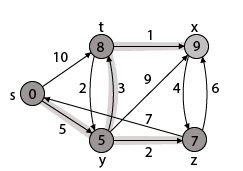


**Step 5:** Now we will find Adj [t]

Adj [t] → [x, y] [Here t is u and x and y are v]

**Case - (i)** t → x  
              d [v] > d [u] + w [u, v]  
              d [x] > d [t] + w [t, x]  
              13 > 8 + 1  
              13 > 9  
**Then       d [x] ← 9**  
              **π [x] ← t**

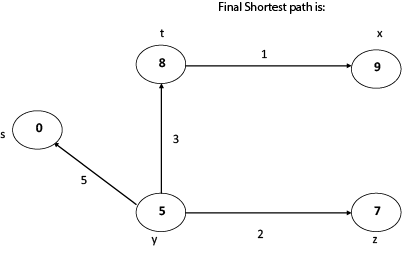
**Case - (ii)** t → y  
              d [v] > d [u] + w [u, v]  
              d [y] > d [t] + w [t, y]  
              5 > 10  
∴ This condition does not satisfy so it will be discarded.



Thus we get all shortest path vertex as

Weight from s to y is 5  
Weight from s to z is 7  
Weight from s to t is 8  
Weight from s to x is 9

These are the shortest distance from the source's' in the given graph.



Disadvantage of Dijkstra's Algorithm:

1. It does a blind search, so wastes a lot of time while processing.
2. It can't handle negative edges.
3. It leads to the acyclic graph and most often cannot obtain the right shortest path.
4. We need to keep track of vertices that have been visited.

Next

Bellman-Ford Algorithm

Solves single shortest path problem in which edge weight may be negative but no negative cycle exists.

This algorithm works correctly when some of the edges of the directed graph G may have negative weight. When there are no cycles of negative weight, then we can find out the shortest path between source and destination.

It is slower than Dijkstra's Algorithm but more versatile, as it capable of handling some of the negative weight edges.

This algorithm detects the negative cycle in a graph and reports their existence.

11.9M

240

History of Java

Based on the "**Principle of Relaxation**" in which more accurate values gradually recovered an approximation to the proper distance by until eventually reaching the optimum solution.

Given a weighted directed graph G = (V, E) with source s and weight function w: E → R, the Bellman-Ford algorithm returns a Boolean value indicating whether or not there is a negative weight cycle that is attainable from the source. If there is such a cycle, the algorithm produces the shortest paths and their weights. The algorithm returns TRUE if and only if a graph contains no negative - weight cycles that are reachable from the source.

Recurrence Relation(NOT in Syllabuss)

**distk [u] = [min[distk-1 [u],min[ distk-1 [i]+cost [i,u]]] as i except u.**

k → k is the source vertex  
u → u is the destination vertex  
i → no of edges to be scanned concerning a vertex.

**BELLMAN -FORD (G, w, s)**

1. INITIALIZE - SINGLE - SOURCE (G, s)

2. for i ← 1 to |V[G]| - 1

3. do for each edge (u, v) ∈ E [G]

4. do RELAX (u, v, w)

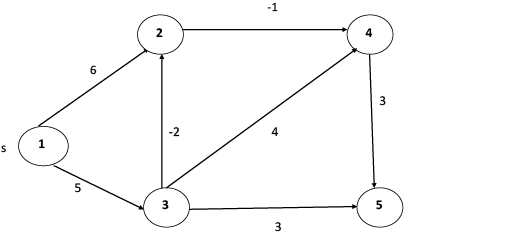
5. for each edge (u, v) ∈ E [G]

6. do if d [v] > d [u] + w (u, v)

7. then return FALSE.

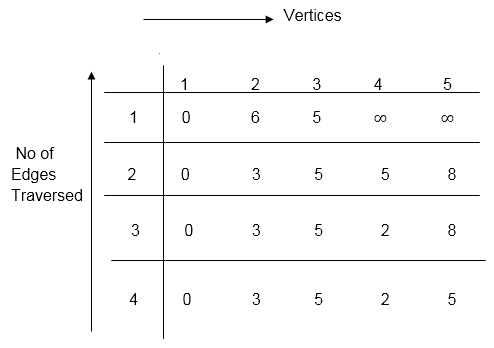
8. return TRUE.

**Example:** Here first we list all the edges and their weights.



**`Solution:**

**distk [u] = [min[distk-1 [u],min[distk-1 [i]+cost [i,u]]] as i ≠ u.**



dist2 [2]=min[dist1 [2],min[dist1 [1]+cost[1,2],dist1 [3]+cost[3,2],dist1 [4]+cost[4,2],dist1 [5]+cost[5,2]]]

Min = [6, 0 + 6, 5 + (-2), ∞ + ∞ , ∞ +∞] = 3

dist2 [3]=min[dist1 [3],min[dist1 [1]+cost[1,3],dist1 [2]+cost[2,3],dist1 [4]+cost[4,3],dist1 [5]+cost[5,3]]]

Min = [5, 0 +∞, 6 +∞, ∞ + ∞ , ∞ + ∞] = 5

dist2 [4]=min[dist1 [4],min[dist1 [1]+cost[1,4],dist1 [2]+cost[2,4],dist1 [3]+cost[3,4],dist1 [5]+cost[5,4]]]

Min = [∞, 0 +∞, 6 + (-1), 5 + 4, ∞ +∞] = 5

dist2 [5]=min[dist1 [5],min[dist1 [1]+cost[1,5],dist1 [2]+cost[2,5],dist1 [3]+cost[3,5],dist1 [4]+cost[4,5]]]

Min = [∞, 0 + ∞,6 + ∞,5 + 3, ∞ + 3] = 8

dist3 [2]=min[dist2 [2],min[dist2 [1]+cost[1,2],dist2 [3]+cost[3,2],dist2 [4]+cost[4,2],dist2 [5]+cost[5,2]]]

Min = [3, 0 + 6, 5 + (-2), 5 + ∞ , 8 + ∞ ] = 3

dist3 [3]=min[dist2 [3],min[dist2 [1]+cost[1,3],dist2 [2]+cost[2,3],dist2 [4]+cost[4,3],dist2 [5]+cost[5,3]]]

Min = [5, 0 + ∞, 3 + ∞, 5 + ∞,8 + ∞ ] = 5

dist3 [4]=min[dist2 [4],min[dist2 [1]+cost[1,4],dist2 [2]+cost[2,4],dist2 [3]+cost[3,4],dist2 [5]+cost[5,4]]]

Min = [5, 0 + ∞, 3 + (-1), 5 + 4, 8 + ∞ ] = 2

dist3 [5]=min[dist2 [5],min[dist2 [1]+cost[1,5],dist2 [2]+cost[2,5],dist2 [3]+cost[3,5],dist2 [4]+cost[4,5]]]

Min = [8, 0 + ∞, 3 + ∞, 5 + 3, 5 + 3] = 8

dist4 [2]=min[dist3 [2],min[dist3 [1]+cost[1,2],dist3 [3]+cost[3,2],dist3 [4]+cost[4,2],dist3 [5]+cost[5,2]]]

Min = [3, 0 + 6, 5 + (-2), 2 + ∞, 8 + ∞ ] =3

dist4 [3]=min[dist3 [3],min[dist3 [1]+cost[1,3],dist3 [2]+cost[2,3],dist3 [4]+cost[4,3],dist3 [5]+cost[5,3]]]

Min = 5, 0 + ∞, 3 + ∞, 2 + ∞, 8 + ∞ ] =5

dist4 [4]=min[dist3 [4],min[dist3 [1]+cost[1,4],dist3 [2]+cost[2,4],dist3 [3]+cost[3,4],dist3 [5]+cost[5,4]]]

Min = [2, 0 + ∞, 3 + (-1), 5 + 4, 8 + ∞ ] = 2

dist4 [5]=min[dist3 [5],min[dist3 [1]+cost[1,5],dist3 [2]+cost[2,5],dist3 [3]+cost[3,5],dist3 [5]+cost[4,5]]]

Min = [8, 0 +∞, 3 + ∞, 8, 5] = 5