Design and Analysis of Algorithm [R1UC407B]

Module-IV: Greedy Technique

Dr. A K Yadav

GALGOTIAS UNIVERSITY

School of Computer Science and Engineering
Plat No 2, Sector 17A, Yamuna Expressway
Greater Noida, Uttar Pradesh - 203201

February 14, 2025

Contents

Greedy Algorithm 3
- Huffman Codes 7
- Fractional Knapsack problem 18
- An activity-selection problem 23
- Task-scheduling problem 30

Minimum Spanning Trees (MST) 33
- Kruskal's algorithm 35
- Prim's algorithm 42

Shortest-paths problem 46
- Dijkstra's algorithm 58
- Bellman-Ford algorithm 64

Greedy Algorithm

- Used for optimization
- Always makes the choice that looks best at the moment
- makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution
- Greedy algorithms do not always yield optimal solutions that means it can not be used for every problem
- If a problem can be solved using greedy that can also be solved using dynamic programming but not vice versa.
- Greedy algorithms are more faster than dynamic programming
- By nature dynamic programming follows bottom up and greedy follows top down approach

Elements of the greedy strategy

Steps involved in greedy strategy:

1. Determine the optimal substructure of the problem.
2. Develop a recursive solution.
3. Show that if we make the greedy choice, then only one subproblem remains.
4. Prove that it is always safe to make the greedy choice.
5. Develop a recursive algorithm that implements the greedy strategy.
6. Convert the recursive algorithm to an iterative algorithm.

More generally, we design greedy algorithms according to the following sequence of steps:

1. Design the optimization problem by selecting one as a choice and rest with the subproblem to solve
2. Prove that there is always an optimal solution to the original problem with greedy choice, and greedy choice is always safe.
3. Demonstrate optimal substructure by showing that if we combine an optimal solution of the the subproblem with main problem , we get an optimal solution to the original problem.

Ingredients of greedy approach

Two key ingredients of greedy approach:

1. Optimal substructure: an optimal solution to the problem contains within it optimal solutions to subproblems. This property is a key ingredient of assessing the applicability of dynamic programming as well as greedy algorithms
2. Greedy-choice property: we can assemble a globally optimal solution by making locally optimal (greedy) choices.

Huffman Codes

- Huffman Codes is used for data compression.
- Data is a sequence of characters.
- Each character is given with their frequency
- Each character is encoded into a codeword using some scheme.
- Suppose there are n characters in the set C.
- Each character $c \in C$ have frequency $freq_c$
- $bit(c)$ is the number of bits required to code c character whose frequency is $freq_c$.

- Our aim is to design a encoding scheme so that we can minimize the total length of codeword.

  minimize $\sum_{c \in C} freq_c \times bit(c)$

- We can use some standard fixed length formatting such as ASCII
- In this case the the total space requirement will be $8 \times \sum_{c \in C} freq_c$
- But we can save more space using non standard fixed length formatting scheme for n characters.
- The length of the code will be $\lceil \lg n \rceil$
- In this case the $\lceil \lg n \rceil$ number of bits are required to represent each character and total bits will be $\lceil \lg n \rceil \times \sum_{c \in C} freq_c$

- Huffman codes can be used even to save more space than both of the above and this uses variable length encoding scheme for each character.
- Huffman codes are prefix codes.
- **Prefix Codes:** Codes in which no codeword is a prefix of some other codeword are called **Prefix Codes**.
- The benefits of Prefix codes are simplified decoding, unambiguous encoding.
- But the disadvantages is that we can not start decoding in between the encoded codeword into the original character.
- Huffman code uses full binary tree for encoding, in which every non leaf node has two children
- All characters are used as leaf, so total leaf will be n
- There are $n - 1$ internal nodes as in full binary tree.

- Each left child is labelled as 0 and each right child is labelled as 1.
- Label value from root to leaf will be the encoding for that leaf character.
- All characters are kept in min priority queue according to their frequency that is least frequency character is at front of the queue.
- Every time we sum the least two frequency of the first two element of the queue and make one
- So after $n-1$ operation we will be having only one element in the min priority queue and that will be the root of the tree.
- Code length of the character c will be equal to the depth of the c $d_T(c)$ in tree T. So

$$\text{minimize } B(T) = \sum_{c \in C} freq_c \times d_T(c)$$

Huffman Coding algorithm

HUFFMAN(C)//C is the set of n characters

1. $n = |C|$
2. $Q = C$ //Q is Min-priority Queue
3. for $i = 1$ to $n - 1$
4. allocate a new node z
5. $z \rightarrow left = EXTRACT - MIN(Q)$
6. $z \rightarrow right = EXTRACT - MIN(Q)$
7. $freq_z = freq_{z \rightarrow left} + freq_{z \rightarrow right}$
8. $INSERT(Q, z)$
9. return $EXTRACT - MIN(Q)$

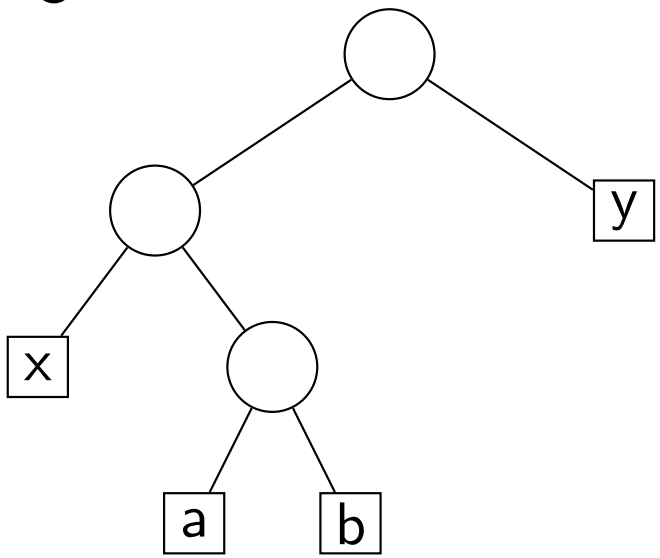
Line 2 will be require $O(n \lg n)$ time for building the min heap of n items. For loop executes $n - 1$ times and each time it rebuild the min heap in $O(\lg n)$ times. Total time complexity will be $O(n \lg n)$

Correctness of the Huffman algorithms

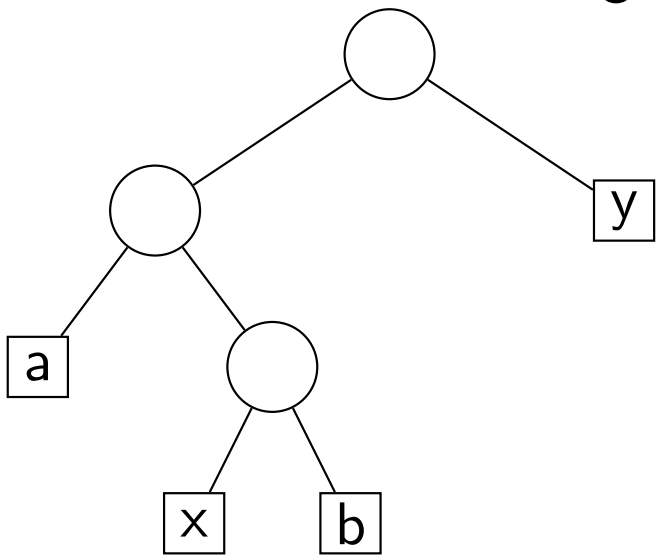
If character x and y having the least frequency then they will be at highest depth in the some optimal tree T for Huffman Code.

- Let character x and y having the least frequency $freq_x$ and $freq_y$ respectively. Also suppose $freq_x \leq freq_y$
- Suppose x and y are not at highest depth but a and b are at the highest depth in the tree T.
- Let $freq_a \leq freq_b$
- Since $freq_x, freq_y$ are lowest frequency and $freq_a, freq_b$ are any arbitrary frequency. Also $freq_x \leq freq_y$ and $freq_a \leq freq_b$. So $freq_x \leq freq_y \leq freq_a \leq freq_b$.
- if $freq_x = freq_b$ then $freq_x = freq_a = freq_y = freq_b$ and we can change the position of the x and y with a and b and hence proved.

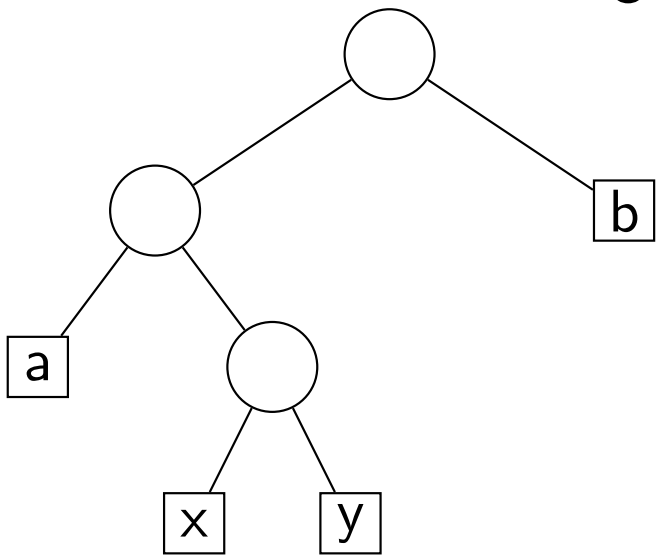
- So assume $freq_x \neq freq_b$ and take tree T as shown in below figure:



- Interchange the position of x with a in tree T and build tree T' as shown in below figure:



- Interchange the position of y with b in tree T' and build tree T'' as shown in below figure:



- Find $B(T) - B(T')$

$B(T) - B(T')$

$$= \sum_{c \in C} freq_c \times d_T(c) - \sum_{c \in C} freq_c \times d_{T'}(c)$$

$$= freq_x \times d_T(x) + freq_a \times d_T(a) - freq_x \times d_{T'}(x) - freq_a \times d_{T'}(a)$$

$$= freq_x \times d_T(x) + freq_a \times d_T(a) - freq_x \times d_T(a) - freq_a \times d_T(x)$$

$$= (freq_a - freq_x)(d_T(a) - d_T(x))$$

Now $(freq_a - freq_x) \geq 0$ because $freq_x$ is the least frequency and $(d_T(a) - d_T(x)) \geq 0$ because a is at the highest depth. So

$$(freq_a - freq_x)(d_T(a) - d_T(x)) \geq 0$$

$\Rightarrow B(T) - B(T^{'}) \geq 0$

$\Rightarrow B(T) \geq B(T^{'})$

But $B(T)$ can not be greater then $B(T^{'})$ because $B(T)$ is the optimal value. So the only possibility is $B(T) = B(T^{'})$
Same way we can prove $B(T^{'}) = B(T^{''})$
So $B(T) = B(T^{'}) = B(T^{''})$
So $B(T^{''})$ is an optimal Huffman code where least frequency characters x and y is at the highest depth.

Fractional Knapsack problem

- There are n item.
- Value of i^{th} item is v_i and weight is w_i.
- Knapsack capacity of the bag is W.
- We wish to maximize the total bag value subject to the constraint that total weight is less than or equal to W that is:

Maximize $\sum_{i=1}^{n} v_i x_i$

under the constraints $\sum_{i=1}^{n} x_i w_i \leq W$, and $x_i = [0, 1]$

Optimal-substructure property

- Consider that W is the optimal load of the bag.
- If we remove a weight w of one item j from the optimal load, the remaining load must be the most valuable load weighing at most $W - w$ from the $n - 1$.
- If this $W^1 = W - w$ is not optimal but W^2 is optimal then $W^2 + w$ will be optimal not W
- But W is optimal under assumption so W^1 will be optimal.
- So Fractional Knapsack problem satisfied optimal-substructure property

Greedy property

- Suppose items are arranged in descending order of $\frac{v_i}{w_i}$
- Take W is the sum of weights from first i weights.
- If we remove a weight w of one item i from the optimal load, the remaining load must be the most valuable load weighing at most $W - w$ from the $i - 1$.
- Because weights are arranged in descending order of value per unit weight
- So if we remove one item then remaining items also more valuable then the rest.
- Hence greedy-choice property is applicable.

Correctness of the algorithm

- Suppose items are arranged in descending order of $\frac{v_i}{w_i}$
- Take W is the sum of weights from first i weights with value V.
- If we remove a weight w of one item $j \leq i$ from the optimal load and replace w from the rest of item $k > i$
- The value of resultant load $V' = V - v_j + v_k$
- But $V' \leq V$ as $v_j \geq v_k$.
- So the the selection of the most valuable item is correct.

An iterative greedy algorithm

FKP(v, w, n, W)

1. Let items are arranged in descending order of $\frac{v_i}{w_i}$
2. J=0,V=0,i=1
3. while($i \leq n$ and $J < W$)
4. if $w_i \leq W - J$
5. $V = V + v_i$
6. $J = J + w_i$
7. else
8. $V = V + \frac{W-J}{w_i} \times v_i$
9. $J = W$
10. $i = i + 1$
11. return V

Complexity of the algorithm is $O(n \lg n)$ for sorting and $O(n)$ for selecting the weights. So it will be $O(n \lg n)$

An activity-selection problem

- Suppose $S = \{a_1, a_2, \ldots, a_n\}$ is a set of n activities
- Each activity a_i have start time s_i and finish time f_i such that $0 \leq s_i < f_i < \infty$
- Two activity a_i and a_j are compatible if one start after other finishes that is either $s_i \geq f_j$ or $s_j \geq f_i$ or we can say time interval $[s_i, f_i)$ of a_i and $[s_j, f_j)$ of a_j is non overlapping.
- a_0 finishes at 0 and a_{n+1} starts after f_n that is $[-\infty, 0)$ for a_0 and $[f_n, \infty)$ for a_{n+1}

Dynamic programming Solution

- S_{ij} is the set of activities that start after activity a_i finishes and that finish before activity a_j starts
- A_{ij} is the set of maximum mutually compatible activities that start after activity a_i finishes and that finish before activity a_j starts and $Aij \subseteq Sij$
- We want to find A_{0n+1}
- Suppose $a_k \in A_{ij}$
- There is two sub-problems A_{ik} and A_{kj} where $A_{ik} = A_{ij} \cap S_{ik}$ and $A_{kj} = A_{ij} \cap S_{kj}$
- $A_{ij} = Aik \cup \{a_k\} \cup A_{kj}$
- $|A_{ij}| = |Aik| + |A_{kj}| + 1$
- If $c[i,j]$ is the size of an optimal solution for the set S_{ij} then $c[i,j] = c[i,k] + c[k,j] + 1$

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \\ \max\{c[i,k] + c[k,j] + 1\} & \text{if } S_{ij} \neq \emptyset \end{cases}$$

Greedy choice Solution

- We should choose an activity that leaves the resource available for as many other activities as possible
- Choose the activity in S with the earliest finish time, so that maximum resource available for the following activities
- Since activities are in sorted increasing order of finishing time so select a_1
- After selecting a_1 only one sub-problem remains: selecting the activity from rest of the activities witch starts after a_1.
- Let $S_k = \{a_i \in S : s_i \geq f_k\}$ is the set of activities that start after activity a_k finishes
- So a_1 is finishing first, we have to find out S_1 only.

A recursive greedy algorithm

RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)

1. $m = k + 1$
2. while $m \leq n$ and $s_m < f_k$ // find the first activity in S_k to finish
3. $\quad m = m + 1$
4. if $m \leq n$
5. $\quad$ return $\{a_m\} \cup$RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
6. else
7. $\quad$ return $\emptyset$

Let a_0 is a activity which finishes at $f_0 = 0$. We want to find out optimal solution $S_0 = S$, So first call we be RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, n)

An iterative greedy algorithm

GREEDY-ACTIVITY-SELECTOR(s, f, n)

1. $A = \{a_1\}$
2. $k = 1$
3. for $m = 2$ to n
4. if $s_m \geq f_k$
5. $A = A \cup \{a_m\}$
6. $k = m$
7. return A

a_k is the most recent addition to A and activities are arranged in monotonically increasing finish time, so f_k is always the maximum finish time of any activity in A i.e. $f_k = \max\{f_i : a_i \in A\}$ and we are searching the next activity which starts after a_k finishes.

Correctness of the algorithm

Is the greedy choice in which we choose the first activity to finish is always part of some optimal solution?

- Statement: Consider any nonempty subproblem S_k. let a_m be an activity in S_k with the earliest finish time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k
- Let A_k is a maximum-size subset of mutually compatible activities in S_k so $A_k \subseteq S_k$.
- let a_j is the activity in A_k with the earliest finish time
- Now there are two cases : either $a_j = a_m$ or we can replace a_j with a_m in A_k without affecting the solution.
- if $a_j = a_m$ then nothing to do and Statement is correct.

- if $a_j \neq a_m$, then let another set $A_k^{'} = (A_k - \{a_j\}) \cup \{a_m\}$ that is substitute a_j with a_m.
- Now we have to check that is still $A_k^{'}$ is a maximum-size subset of mutually compatible activities.
- For that we have to check the new activity a_m is compatible with the rest of activities in $A_k^{'}$.
- As a_m is the first activity to finish in S_k and a_j is the first activity to finish in A_k and $A_k \subseteq S_k$ so either $a_j = a_m$ or $f_m \leq f_j$
- It means we can replace a_j with a_m and still set A_k will remain maximum sized non overlapping set of activities.
- So a_m will be the part of optimal solution.

Task-scheduling problem

The problem of scheduling unit-time tasks with deadlines and penalties/profits for a single processor has the following inputs:

- Set $S = \{a_1, a_2, \ldots, a_n\}$ is a set of n tasks.
- Each task requires unit time to complete
- Every task a_i having some deadline d_i such that $1 \leq d_i \leq n$
- There is n task and every task requires unit time so in n unit of time starting from 0 to n all tasks will be finished.
- If task a_i finishes before deadline d_i then there will be no penalty but if the task is late and finishes after deadline then there will be a penalty p_i
- Or if task a_i finishes before deadline d_i then there will be profit p_i but if the task is late and finishes after deadline then there will be no profit.

- Our aim is to minimize the penalty or maximize the profit.

minimize $Penalty(S) = \sum_{a_i \in S} p_i$ if a_i finishes after deadline d_i

maximize $Profit(S) = \sum_{a_i \in S} p_i$ if a_i finishes before deadline d_i

Solution

- Suppose task a_i having deadline $d_i = 5$ is late and task a_j having deadline $d_j = 6$ is early
- If we execute task a_i on or after 6, still it will be late
- If we execute task a_j on or before 5, still it will be early
- So a early task will be early if we execute that task on or before the deadline and a late task will be late if we execute that task anywhere after deadline.
- We try to execute the high penalty task on time and if any task is late then we can make that task maximum late.
- We try to execute the early task at the scheduled deadline or before.
- We can always transform an arbitrary schedule into early-first form, in which the early tasks precede the late tasks
- Complexity of the algorithm will be $O(n^2)$ using Greedy because n independence check takes $O(n)$ time.

Minimum Spanning Trees (MST)

- Tree is a connected undirected acyclic graph.
- Tree is a set of nodes and set of edges connected that nodes that is $T = (V, E)$ and each edge $(u, v) \in E$ having weight $w(u, v)$
- Minimum means the sum of all edges to connect all nodes is minimum
- Spanning means tree span over all the nodes.
- So Minimum Spanning Tree is a undirected weighted acyclic graphs that covers all the nodes with minimum sum of the weights.
- There will be $n - 1$ edges for n vertices in the MST.

- Let G is a undirected graph with set of vertices V and set of edges E and each edge $(u, v) \in E$ having weight $w(u, v)$. We have to find out an acyclic subset $T \subseteq G$ that connects all of the vertices and whose total weight is minimum that is

$$\text{minimize } W(T) = \sum_{(u,v)\in T} w(u, v)$$

- T in this case will be MST.

GENERIC-MST(G,w)

1. $A = \emptyset$
2. while A does not form a spanning tree
3. find an edge (u, v) that is safe for A
4. $A \cup \{(u, v)\}$
5. return A

Kruskal's algorithm

- Kruskal's algorithm uses the least weight edge to connect the two tree to make one.
- Kruskal's algorithm qualifies as a greedy algorithm because at each step it adds to the forest an edge of least possible weight
- It uses a disjoint-set data structure to maintain several disjoint sets of elements
- The operation $FIND - SET(u)$ returns address of the representative element from the set that contains u
- We check $FIND - SET(u)$ and $FIND - SET(v)$ for any edge (u, v). If they return the same address then it means the are already connected.

- If $FIND - SET(u)$ and $FIND - SET(v)$ returns different value that it means they are the members of two different tree and using $UNION(u, v)$ we make one tree connected by edge (u, v)

MST-KRUSKAL(G,w)

1. $A = \emptyset$
2. for each vertex $v \in V$
3. $\quad MAKE - SET(v)$
4. sort the edges of E into nondecreasing order by weight w
5. for each edge $(u, v) \in E$ taken in nondecreasing order by weight
6. $\quad$ if $FIND - SET(u) \neq FIND - SET(v)$
7. $\quad\quad A \cup \{(u, v)\}$
8. $\quad\quad UNION(u, v)$
9. return A

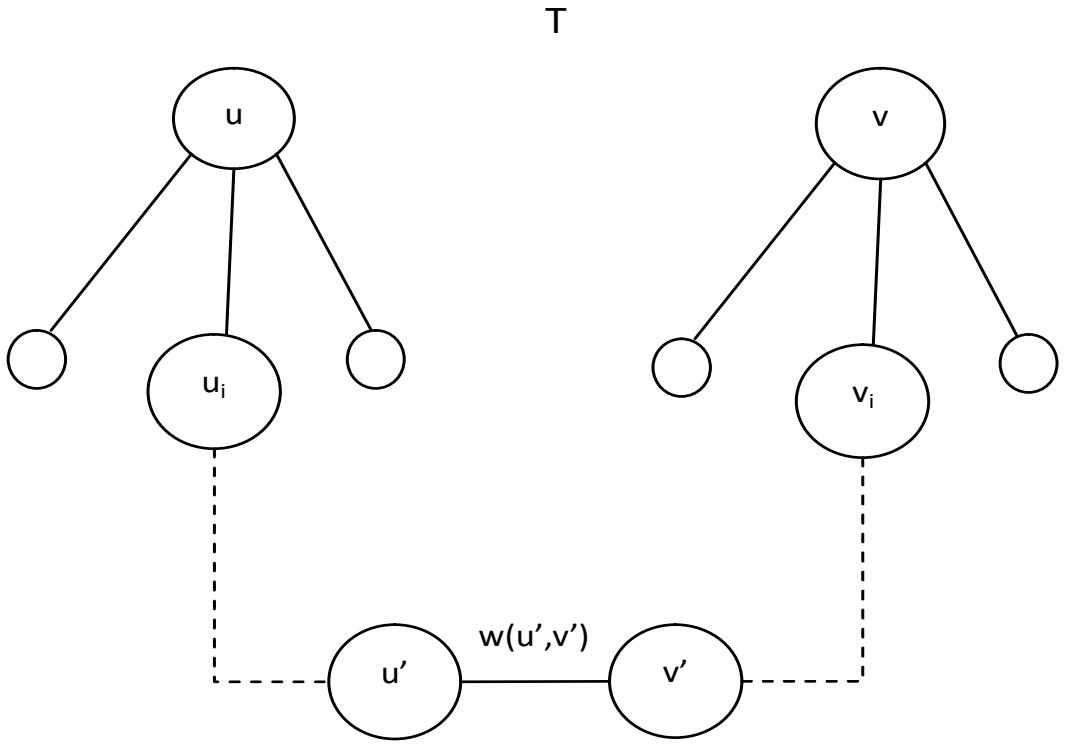
Time complexity of the algorithm is $O(E \lg E)$ or $O(E \lg V)$ as $|E| < |V|^2 \Rightarrow \lg E = O(\lg V)$

Correctness of Kruskal's Algorithm

Prove that if $w(u, v)$ is minimum weighted edge of the graph G then edge (u, v) will be part of the some MST T.

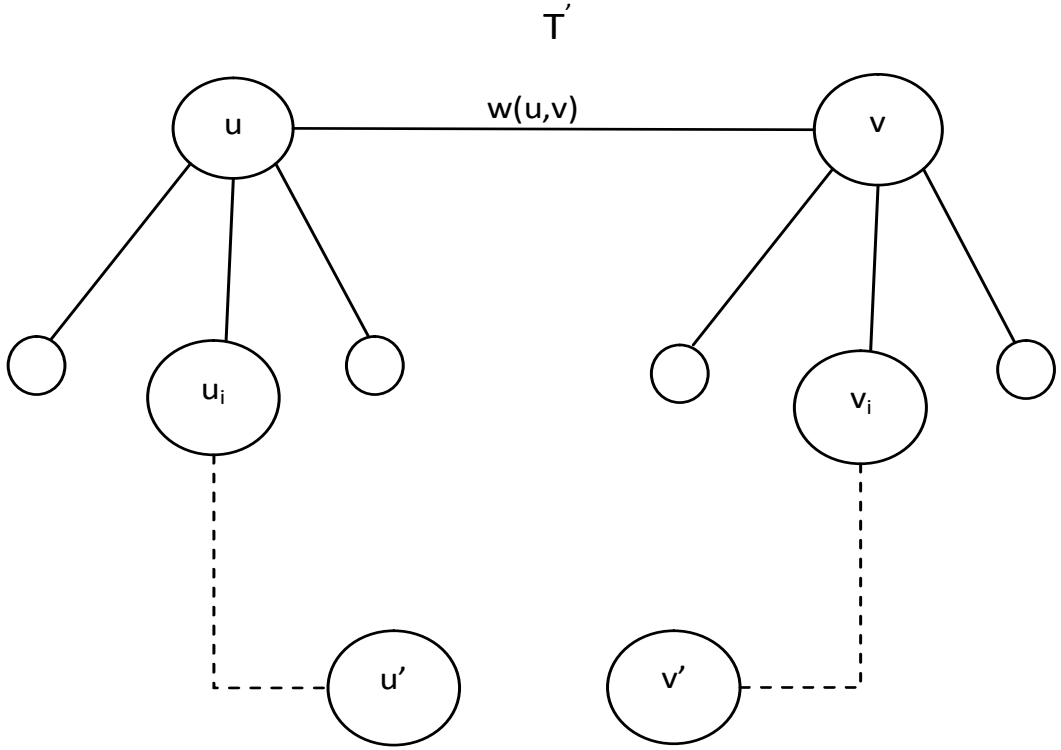
- Suppose T is a MST with weight $W(T)$ and edge (u, v) is not the part of the tree T.
- It means u and v are connected by some other edges being tree T is spanning tree.

- Suppose vertex u is connected by u_i vertices and v is connected by v_j vertices as shown in below figure:



- Now suppose $(u^{'}, v^{'})$ is a least weight edge from the path $(u, u_i, \ldots, v_j, v)$

- Remove $(u^{'}, v^{'})$ and add edge (u, v) to make the tree connected as shown in below figure:



- The weight of the tree $T^{'}$ will be: $W(T^{'}) = W(T) -$ weight of the edge removed $+$ weight of the edge added

- $\Rightarrow W(T^{'}) = W(T) - w(u^{'}, v^{'}) + w(u, v)$
- $\Rightarrow W(T^{'}) = W(T) - (w(u^{'}, v^{'}) - w(u, v))$
- $\Rightarrow W(T) - W(T^{'}) = w(u^{'}, v^{'}) - w(u, v)$
- $\Rightarrow W(T) - W(T^{'}) \geq 0$
- $\Rightarrow W(T) \geq W(T^{'})$
- But $W(T)$ is the MST as per assumption so $W(T)$ can't be greater than $W(T^{'})$
- So $W(T) = W(T^{'})$ and hence $T^{'}$ is also a MST and (u, v) will be the part of MST $T^{'}$

Prim's algorithm

- Prim's algorithm has the property that the edges in the set A always form a single tree unlike Kruskal.
- We starts taking any node r as root and grows till the tree spans all the vertices in V
- In each step a new minimum edge vertex is added into A
- The algorithm uses greediness, as each step it adds to the tree an edge that contributes the minimum weight to the tree.
- We use min-priority queue Q based on key to store all vertices that are not the part of the MST till now.
- For each vertex v, the value $key[v]$ will be the minimum weight of the edge connected v with all other nodes of the graph.

- if any node is not connected with node v then $key[v] = \infty$
- $key[r] = 0$ as r is the root of the tree
- As we are expending tree by taking the node with minimum weight edge at every step so the tree will be MST at every step.
- $\pi[v]$ is the predecessor of v. For example if (u, v) is the edge of the tree then $\pi[v] = u$
- $adj[u] = v$ if there is an edge from u to v that is (u, v)

Algorithm for Prim

MST-PRIM(G,w,r)

1. $A = \emptyset$
2. $Q = \emptyset$
3. for each $u \in V$
4. $\quad key[u] = \infty$
5. $\quad \pi[u] = NIL$
6. $\quad Insert(Q, u)$
7. $key[r] = 0$
8. while $Q \neq \emptyset$
9. $\quad u = Extract(Q)$
10. $\quad$ for each $v \in adj[u]$
11. $\quad\quad$ if $v \in Q$ and $w(u, v) < key[v]$

12. $\pi[v] = u$
13. $key[v] = w(u, v)$
14. $A \cup \{(u, v)\}$
15. return

Time complexity of the algorithm is $O(V \lg V + E \lg V)$ or $O(E \lg V)$ using Min-heap and $O(E + V \lg V)$ using Fibonacci heap.

Shortest-paths problem

- G is a given weighted and directed graph with set of vertices V and set of edges E such that $G = (V, E)$ with weight function $w : E \rightarrow \mathbb{R}$
- The weight $w(p)$ of path $p = \langle v_0, v_1, \ldots, v_k \rangle$ is the sum of the weights of edges of the path. That is

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

- The shortest-path weight $\delta(u, v)$ from u to v is defined as

$$\delta(u, v) = \begin{cases} min\{w(p) : u \rightsquigarrow v & \text{if there is a path from } u \text{ to } v \\ \infty & \text{otherwise} \end{cases}$$

- A shortest path from vertex u to vertex v is any path p with weight $w(p) = \delta(u.v)$

Variants of Shortest path

- Single-source shortest-paths problem : Find a shortest path from a given source vertex $s \in V$ to each vertex $v \in V$
- Single-destination shortest-paths problem: Find a shortest path from each vertex $u \in V$ to a given destination vertex $d \in V$
- Single-pair shortest-path problem: Find a shortest path from a given source $s \in V$ to a given destination $d \in V$.
- All-pairs shortest-paths problem: Find a shortest path from each vertex $u \in V$ to each vertex $v \in V$.

Optimal substructure of a shortest path

Statement: If p is a optimal path from v_0 to v_k then p_{ij} will be the optimal path from p_i to p_j for any value of i and j such that $0 \leq i \leq j \leq k$.

Proof:

- Given p is the optimal path from v_0 to v_k that is $w(p)$ is minimum over $v_0 \rightsquigarrow v_k$
- Let's break path p into 3 parts :
- p_{0i} is the path from v_0 to v_i with weight $w(p_{0i})$
- p_{ij} is the path from v_i to v_j with weight $w(p_{ij})$
- p_{jk} is the path from v_j to v_k with weight $w(p_{jk})$
- That is $v_0 \rightsquigarrow v_i \rightsquigarrow v_j \rightsquigarrow v_k$
- Now $w(p) = w(p_{0i}) + w(p_{ij}) + w(p_{jk})$

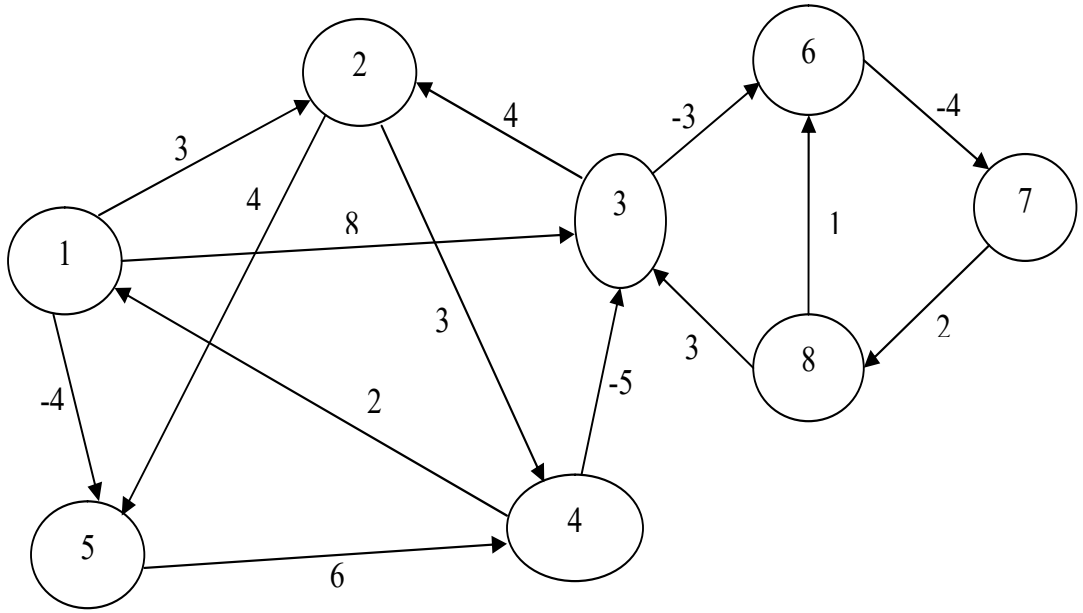
- Let $w(p_{ij}^{'})$ is the minimum weight path from v_i to v_j through other vertices that is $w(p_{ij}^{'}) \leq w(p_{ij})$
- So $w(p^{'}) = w(p_{0i}) + w(p'_{ij}) + w(p_{jk})$
- As $w(p'_{ij}) \leq w(p_{ij})$ so $w(p^{'}) \leq w(p)$
- But $w(p)$ is the optimal path so $w(p^{'}) \not< w(p)$
- The only possible case is $w(p^{'}) = w(p)$
- It means $w(p'_{ij}) = w(p_{ij})$

So Subpaths of shortest paths are shortest paths.

Negative-weight cycle

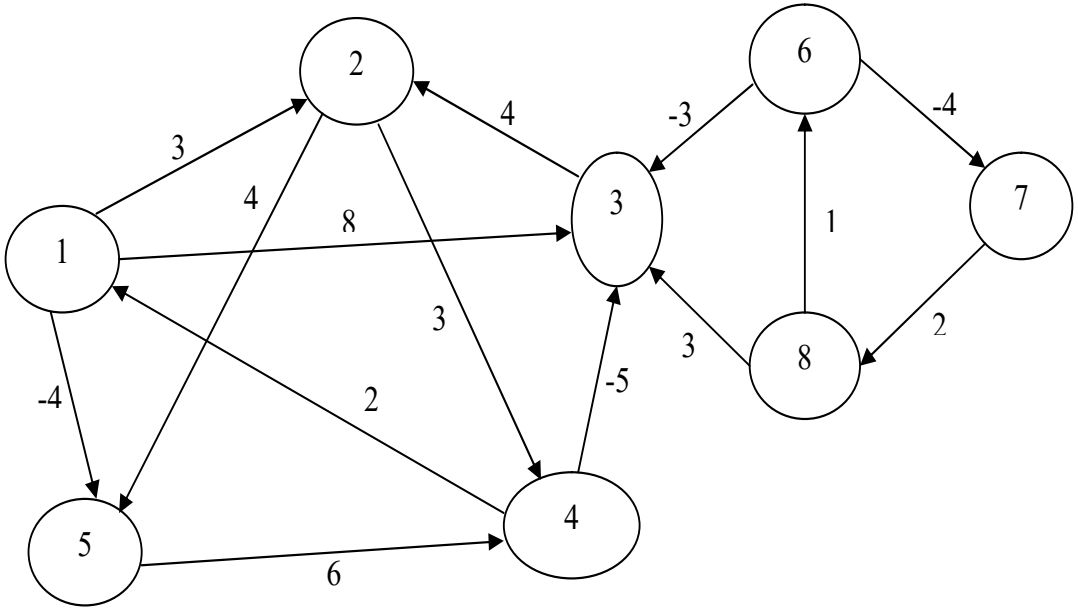
- An edge with with negative weight is called negative-weight edge.
- If $w(p)$ is less than 0 ($w(p) < 0$) then it is known as negative-weight cycle
- If no negative-weight cycles reachable from the source s then shortest path said to well defined
- If negative-weight cycles reachable from the source s then shortest path said to not well defined
- If shortest path is well defined then $\delta(u, v) =$"some finite value"
- If shortest path is not well defined then $\delta(u, v) = -\infty$
- It means no path from s to a vertex on the cycle can be a shortest path

- That is we can always find a path with lower weight by just traversing the negative-weight cycle once more.
- See below figure for negative-weight cycle reachable from the source $s = 1$



- But if negative-weight cycles unreachable from the source $s = 1$ then also we can find the shortest path

- Please see below figure for negative-weight cycle cycles unreachable from the source $s = 1$



- There is no path from source $s = 1$ to the unreachable nodes. so $w(p) = \infty$ in this case
- Any acyclic path can have maximum $|V| - 1$ edges and $|V|$ vertices.

Representing shortest paths

- G is a graph with $G = (V, E)$. $\pi[v]$ is a predecessor of node v. It means there is edge between $(\pi[v], v)$
- **Predecessor subgraph** $G_\pi = (V_\pi, E_\pi)$ is subgraph of G taking s as source where:.
- The vertex set V_π to be the set of vertices of G with non-NIL predecessors, plus the source s:
  $V_\pi = \{v \in V : \pi[v] \neq NIL\} \cup \{s\}$
- The directed edge set E_π is the set of edges induced by the π values for vertices in V_π:
  $E_\pi = \{(\pi[v], v) \in E : v \in V_\pi - \{s\}\}$

- **Shortest-paths tree** is a directed subgraph $G' = (V', E')$ with s as root where $V' \subseteq V$ and $E' \subseteq E$ such that :
  1. G' forms a rooted tree with root s
  2. V' is the set of vertices reachable from s in G
  3. $\forall v \in V'$, the unique simple path from s to v in G' is a shortest path from s to v in G
- Shortest paths are not necessarily unique and neither are shortest-paths trees.
- $d[v]$ is the shortest-path estimate between the source s and each vertex $v \in V$
- We initialize the shortest-path estimates and predecessors as : **INITIALIZE-SINGLE-SOURCE(G,s)**
  1. for each vertex $v \in V$
  2. $\quad d[v] = \infty$
  3. $\quad \pi[v] = NIL$

4. $d[s] = 0$

- **Relaxation**: Relaxing an edge (u, v) means testing whether we can improve the shortest path to v found so far by going through u, if yes then update $d[v]$ and $\pi[v]$
- The following code performs a relaxation step on edge (u, v)

RELAX(u,v,w)

1. if $d[v] > d[u] + w(u, v)$
2. $\quad d[v] = d[u] + w(u, v)$
3. $\quad \pi[v] = u$

Properties of shortest paths

- **Triangle inequality:** For any edge $(u, v) \in E$, we have $\delta(s, v) \leq \delta(s, u) + w(u, v)$
- **Upper-bound property:** $\forall v \in V, d[v] \geq \delta(s, v)$ and once $d[v] = \delta(s, v)$ then it never changes
- **No-path property:** If there is no path from s to v, then we always have $d[v] = \delta(s, v) = \infty$
- **Convergence property:** If $s \rightsquigarrow u \rightarrow v$ is a shortest path in G for some $u, v \in V$ and if $d[u] = \delta(s, u)$ at any time prior to relaxing edge (u, v), then $d[v] = \delta(s, v)$ at all times afterward.

- **Path-relaxation property:** If $p = \langle v_0, v_1, \ldots, v_k \rangle$ is a shortest path from $s = v_0$ to v_k, and we relax the edges of p in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, then $d[v_k] = \delta(s, v_k)$. This property holds regardless of any other relaxation steps that occur, even if they are intermixed with relaxations of the edges of p.
- **Predecessor-subgraph property:** Once $f[v] = \delta(s, v)$ for all $v \in V$, the predecessor subgraph is a shortest-paths tree rooted at s.

Dijkstra's algorithm

- It is used to find single-source shortest-paths problem on a non-negative weighted,directed graph
- S is the set of vertices whose final shortest-path weights from the source s have already been determined
- The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate
- Adds u to S, and relaxes all edges leaving u.
- We use a min-priority queue Q of vertices based on their d values

Algorithm: DIJKSTRA(G,w,s)

1. INITIALIZE-SINGLE-SOURCE(G,s)
2. $S = \emptyset$

3. $Q = V$
4. while $Q \neq \emptyset$
5. $\quad u =$EXTRACT-MIN(Q)
6. $\quad S = S \cup \{u\}$
7. $\quad$ for each vertex $v \in adj[u]$
8. $\quad\quad RELAX(u, v, w)$

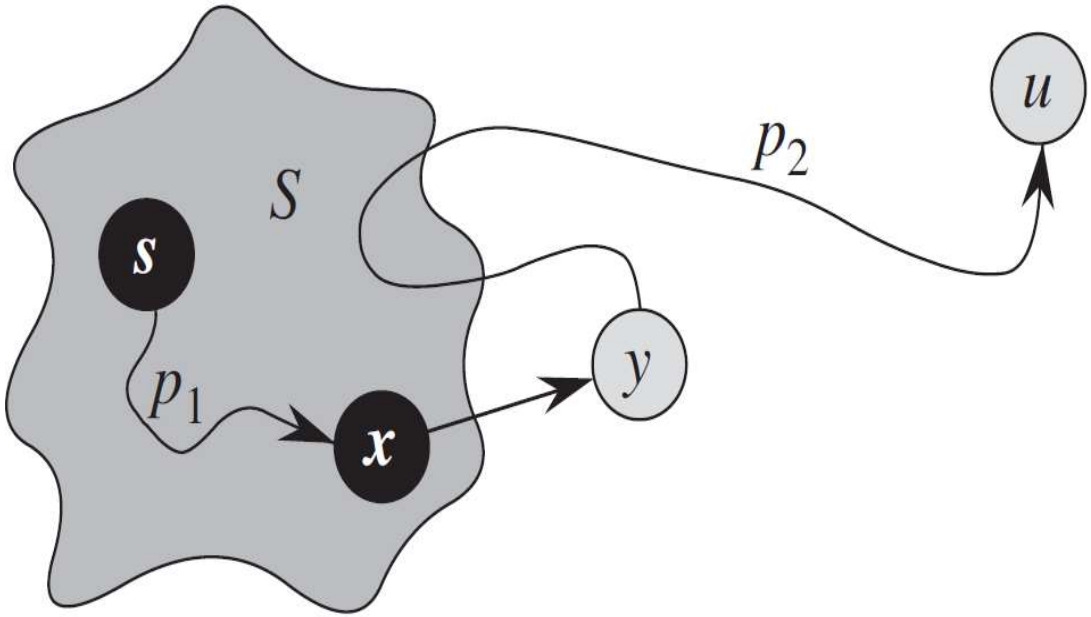
Time complexity of the algorithm is $O(V \lg V + E \lg V)$ or $O(E \lg V)$ using Min-heap and $O(E + V \lg V)$ using Fibonacci heap.

Correctness of Dijkstra's algorithm

We have to prove that $d[u] = \delta(s, u)$ when u is added to set S for each vertex $u \in E$ if u is reachable from s.

- We will prove it by contradict.
- Let u is the first vertex such that $\delta(s, u) < d[u] < \infty$ when add to S.
- $u \neq s$ because s is the first vertex added to the S and $d[s] = \delta(s, s) = 0$
- $S \neq \emptyset$ because at least s was there in S before u is added to S

- There must be at least one path from s to u because our assumption is that $\delta(s, u) < d[u] < \infty$, There must be at least one shortest path p from s to u.



- Let p is a path from s to u before adding u into S such that $s \rightsquigarrow x \rightsquigarrow y \rightsquigarrow u$ and $s, x \in S$ and $y, u \notin S$

- $d[y] = \delta(s, y)$ because y is predecessor of u so it will be relax before u and as per assumption u is the first vertex where $d[u] \neq \delta(s, u)$
- $\delta(s, y) \leq \delta(s, u)$ because y is the predecessor of u and at the the shortest path from s also there is no negative weights so $w(y, u) \geq 0$

$$d[y] = \delta(s, y)$$

$$\leq \delta(s, u)$$

$$\leq d[u]$$

- But because both vertices $y, u \notin S$ and when we extract u from Q then $d[u] \leq d[y]$
- $d[y] \leq d[u]$ and $d[u] \leq d[y]$ is only possible when $d[u] = d[y]$

- So $d[y] = \delta(s, y) = \delta(s, u) = d[u]$ means out assumption is wrong.
- Hence proved that $d[u] = \delta(s, u)$ when u is added to set S

Bellman-Ford algorithm

- A graph with $|V|$ vertices can have maximum $|V| - 1$ edges and $|V|$ vertices in any simple path.
- If number of edges in the path is greater than $|V| - 1$ or vertices on the path is greater than $|V|$ including source and destination then there is cycle in path.
- Bellman works upon this fundamental and relaxes all edges $|V| - 1$ times to find the shortest path from single source.
- After $|V| - 1$ passes, there will be shortest path among nodes form source s, if exists.
- One more cycle of relax, if decrease the shortest path from source then it means there will be $|V|$ edges and $|V| + 1$ vertices and hence a negative weight cycle.
- The algorithm returns either TRUE if there is no negative weight cycle or FALSE if negative weight cycle

Algorithm

BELLMAN-FORD(G,w,s)

1. INITIALIZE-SINGLE-SOURCE(G, s)
2. for $i = 1$ to $|V| - 1$
3. for each edge $(u, v) \in E$
4. RELAX(u, v, w)
5. for each edge $(u, v) \in E$
6. if $d[v] > d[u] + w(u, v)$
7. return FALSE
8. return TRUE

Time complexity of the algorithm is $O(VE)$

Correctness of the Bellman-Ford algorithm

Statement: If G contains no negative-weight cycles that are reachable from s, then the algorithm returns TRUE, we have $d[v] = \delta(s, v)$ for all vertices $v \in V$, and the predecessor subgraph G_π is a shortest-paths tree rooted at s. If G does contain a negative-weight cycle reachable from s, then the algorithm returns FALSE.

- Suppose that graph G contains no negative-weight cycles that are reachable from the source s.
- Now we have to prove that the algorithms will return TRUE.
- If vertex v is reachable from s, then at termination after $|V| - 1$ iterations $d[v] = \delta(s, v)$
- If vertex v is unreachable from s, then at termination after $|V| - 1$ iterations $d[v] = \delta(s, v) = \infty$

- After termination of $|V| - 1$ iterations, for all edge $(u, v) \in E, d[v] = \delta(s, v)$

  $d[v] = \delta(s, v)$

  $d[u] = \delta(s, u)$

  $d[v] = \delta(s, v) \leq \delta(s, u) + w(u, v)$-by the triangle inequality

  $d[v] \leq d[u] + w(u, v)$

- So $d[v] \leq d[u] + w(u, v)$ for all edge $(u, v) \in E$ in the predecessor subgraph G_π is a shortest path from source s
- So $d[v] > d[u] + w(u, v)$ of line no 6 of the algorithm will never returns TRUE and hence algorithm will return TRUE.
- Suppose that graph G contains negative-weight cycles that are reachable from the source s.

- Now we have to prove that the algorithms will return FALSE.
- Let $c = \langle v_0, v_1, \ldots, v_k \rangle$ is a negative weight cycle in the path p reachable from source s in graph G where $v_0 = v_k$
- $\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0 \leftarrow$ because of negative weight cycle
- Now let contradiction that there is a negative weight cycle but still algorithm returns TRUE.
- So $d[v_i] \leq d[v_{i-1}] + w(v_{i-1}, v_i)$ for $i = 1, 2, \ldots, k$.
- Taking summation over all the k values

$$\sum_{i=1}^{k} d[v_i] \leq \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

- $d[v_1] + d[v_2] + \cdots + d[v_{k-1}] + d[v_k]$

  $\leq d[v_0] + d[v_1] + \cdots + d[v_{k-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i)$

- $0 \leq \sum_{i=1}^{k} w(v_{i-1}, v_i), \because v_0 = v_k$
- But $\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$ because of negative weight cycle
- So our assumption of contradiction is wrong and the algorithm will return FALSE.

Thank you

Please send your feedback or any queries to
ashokyadav@galgotiasuniversity.edu.in