Malay Kumar (6316038)

Gurdeep (6316022)

Amritpal(6316076)

Kruskal’s Algorithm

**Kruskal's algorithm** is a [minimum-spanning-tree algorithm](https://en.wikipedia.org/wiki/Minimum_spanning_tree#Algorithms) which finds an edge of the least possible weight that connects any two trees in the forest. It is a [greedy algorithm](https://en.wikipedia.org/wiki/Greedy_algorithm) in [graph theory](https://en.wikipedia.org/wiki/Graph_theory) as it finds a [minimum spanning tree](https://en.wikipedia.org/wiki/Minimum_spanning_tree) for a [connected](https://en.wikipedia.org/wiki/Connectivity_(graph_theory)) [weighted graph](https://en.wikipedia.org/wiki/Glossary_of_graph_theory#Weighted_graphs_and_networks) adding increasing cost arcs at each step.

A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected..

By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one spanning tree.

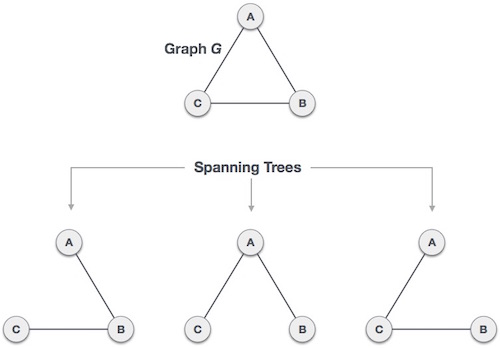


Figure: spanning tree

**SOURCE CODE:**

*# Python program for Kruskal's algorithm to find*

*# Minimum Spanning Tree of a given connected,*

*# undirected and weighted graph*

*from collections import defaultdict*

*#Class to represent a graph*

*class Graph:*

*def \_\_init\_\_(self,vertices):*

*self.V= vertices #No. of vertices*

*self.graph = [] # default dictionary*

*# to store graph*

*# function to add an edge to graph*

*def addEdge(self,u,v,w):*

*self.graph.append([u,v,w])*

*# A utility function to find set of an element i*

*# (uses path compression technique)*

*def find(self, parent, i):*

*if parent[i] == i:*

*return i*

*return self.find(parent, parent[i])*

*# A function that does union of two sets of x and y*

*# (uses union by rank)*

*def union(self, parent, rank, x, y):*

*xroot = self.find(parent, x)*

*yroot = self.find(parent, y)*

*# Attach smaller rank tree under root of*

*# high rank tree (Union by Rank)*

*if rank[xroot] < rank[yroot]:*

*parent[xroot] = yroot*

*elif rank[xroot] > rank[yroot]:*

*parent[yroot] = xroot*

*# If ranks are same, then make one as root*

*# and increment its rank by one*

*else :*

*parent[yroot] = xroot*

*rank[xroot] += 1*

*# The main function to construct MST using Kruskal's*

*# algorithm*

*def KruskalMST(self):*

*result =[] #This will store the resultant MST*

*i = 0 # An index variable, used for sorted edges*

*e = 0 # An index variable, used for result[]*

*# Step 1: Sort all the edges in non-decreasing*

*# order of their*

*# weight. If we are not allowed to change the*

*# given graph, we can create a copy of graph*

*self.graph = sorted(self.graph,key=lambda item: item[2])*

*parent = [] ; rank = []*

*# Create V subsets with single elements*

*for node in range(self.V):*

*parent.append(node)*

*rank.append(0)*

*# Number of edges to be taken is equal to V-1*

*while e < self.V -1 :*

*# Step 2: Pick the smallest edge and increment*

*# the index for next iteration*

*u,v,w = self.graph[i]*

*i = i + 1*

*x = self.find(parent, u)*

*y = self.find(parent ,v)*

*# If including this edge does't cause cycle,*

*# include it in result and increment the index*

*# of result for next edge*

*if x != y:*

*e = e + 1*

*result.append([u,v,w])*

*self.union(parent, rank, x, y)*

*# Else discard the edge*

*# print the contents of result[] to display the built MST*

*print ("Following are the edges in the constructed MST")*

*for u,v,weight in result:*

*#print str(u) + " -- " + str(v) + " == " + str(weight)*

*print ("%d -- %d == %d" % (u,v,weight))*

*# Driver code*

*g = Graph(4)*

*g.addEdge(0, 1, 10)*

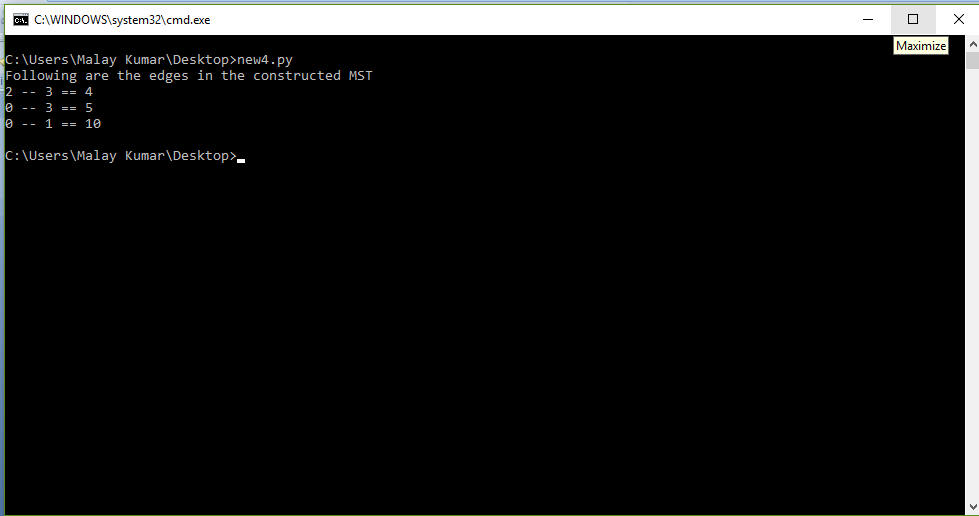
*g.addEdge(0, 2, 6)*

*g.addEdge(0, 3, 5)*

*g.addEdge(1, 3, 15)*

*g.addEdge(2, 3, 4)*

*g.KruskalMST()*

**

*Fig:output*

Github link: <https://github.com/malay190/DAAAssignment/blob/master/kruskalalgo.py>

Hamiltonian Cycle | Backtracking-6

[Hamiltonian Path](http://en.wikipedia.org/wiki/Hamiltonian_path) in an undirected graph is a path that visits each vertex exactly once. A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian Path such that there is an edge (in graph) from the last vertex to the first vertex of the Hamiltonian Path. Determine whether a given graph contains Hamiltonian Cycle or not. If it contains, then print the path. Following are the input and output of the required function.

Output:

An array path[V] that should contain the Hamiltonian Path. path[i] should represent the ith vertex in the Hamiltonian Path. The code should also return false if there is no Hamiltonian Cycle in the graph.

For example, a Hamiltonian Cycle in the following graph is {0, 1, 2, 4, 3, 0}. There are more Hamiltonian Cycles in the graph like {0, 3, 4, 2, 1, 0}

(0)--(1)--(2)

| / \ |

| / \ |

| / \ |

(3)-------(4)

**SOURCE- CODE**

*# Python program for solution of*

*# hamiltonian cycle problem*

*class Graph():*

*def \_\_init\_\_(self, vertices):*

*self.graph = [[0 for column in range(vertices)]\*

*for row in range(vertices)]*

*self.V = vertices*

*''' Check if this vertex is an adjacent vertex*

*of the previously added vertex and is not*

*included in the path earlier '''*

*def isSafe(self, v, pos, path):*

*# Check if current vertex and last vertex*

*# in path are adjacent*

*if self.graph[ path[pos-1] ][v] == 0:*

*return False*

*# Check if current vertex not already in path*

*for vertex in path:*

*if vertex == v:*

*return False*

*return True*

*# A recursive utility function to solve*

*# hamiltonian cycle problem*

*def hamCycleUtil(self, path, pos):*

*# base case: if all vertices are*

*# included in the path*

*if pos == self.V:*

*# Last vertex must be adjacent to the*

*# first vertex in path to make a cyle*

*if self.graph[ path[pos-1] ][ path[0] ] == 1:*

*return True*

*else:*

*return False*

*# Try different vertices as a next candidate*

*# in Hamiltonian Cycle. We don't try for 0 as*

*# we included 0 as starting point in in hamCycle()*

*for v in range(1,self.V):*

*if self.isSafe(v, pos, path) == True:*

*path[pos] = v*

*if self.hamCycleUtil(path, pos+1) == True:*

*return True*

*# Remove current vertex if it doesn't*

*# lead to a solution*

*path[pos] = -1*

*return False*

*def hamCycle(self):*

*path = [-1] \* self.V*

*''' Let us put vertex 0 as the first vertex*

*in the path. If there is a Hamiltonian Cycle,*

*then the path can be started from any point*

*of the cycle as the graph is undirected '''*

*path[0] = 0*

*if self.hamCycleUtil(path,1) == False:*

*print "Solution does not exist\n"*

*return False*

*self.printSolution(path)*

*return True*

*def printSolution(self, path):*

*print "Solution Exists: Following is one Hamiltonian Cycle"*

*for vertex in path:*

*print vertex,*

*print path[0], "\n"*

*# Driver Code*

*''' Let us create the following graph*

*(0)--(1)--(2)*

*|   / \   |*

*|  /   \  |*

*| /     \ |*

*(3)-------(4)    '''*

*g1 = Graph(5)*

*g1.graph = [ [0, 1, 0, 1, 0], [1, 0, 1, 1, 1],*

*[0, 1, 0, 0, 1,],[1, 1, 0, 0, 1],*

*[0, 1, 1, 1, 0], ]*

*# Print the solution*

*g1.hamCycle();*

*''' Let us create the following graph*

*(0)--(1)--(2)*

*|   / \   |*

*|  /   \  |*

*| /     \ |*

*(3)       (4)    '''*

*g2 = Graph(5)*

*g2.graph = [ [0, 1, 0, 1, 0], [1, 0, 1, 1, 1],*

*[0, 1, 0, 0, 1,], [1, 1, 0, 0, 0],*

*[0, 1, 1, 0, 0], ]*

*# Print the solution*

*g2.hamCycle();*

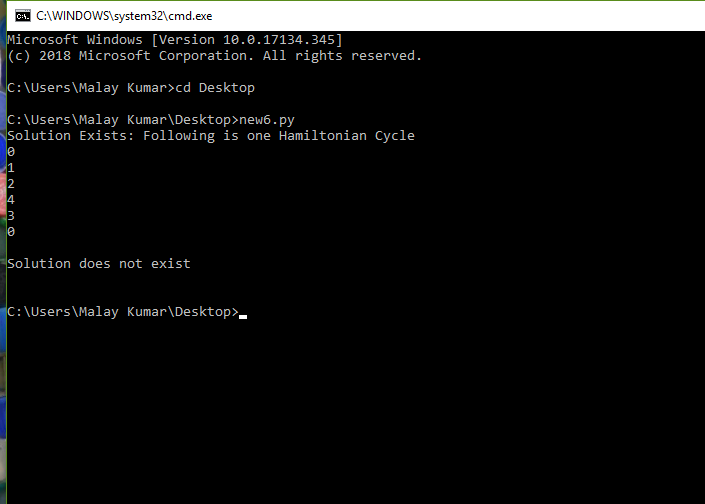


Fig:output

Github link:<https://github.com/malay190/DAAAssignment/blob/master/hamiltonion.py>