Code:

```
import numpy as np
import heapq


def update_lists(list,item):
  print str(list) + str(item)
  list.extend([item])
  print "call"
  print list
  return list


#This class is implemented for function of priority queue

class PriorityQueue:
  def __init__(self):
    self._State_queue = []
    self._State_index = 0

  def push(self, item, priority):
    heapq.heappush(self._State_queue, (priority, self._State_index, item))
    print "item pushed with priority"
    print priority
    self._State_index += 1

  def pop(self):
    return heapq.heappop(self._State_queue)[-1]


# This class is for representing graph
class garph_class:
  #Declaration of members of Class graph_class
  def __init__(self, data):
    self.vertex_count = int(data)
    self.graph=np.zeros((self.vertex_count,self.vertex_count),dtype=int)
    self.opt_cycle=[]

  #This function collects
  def generate_edges(self):
    edge=0
    for i in range(0,self.vertex_count):
      for j in range(0,self.vertex_count):
        if i is not j:
          if self.graph[i][j] == 0:
            print "i, j=" + str(i) + " " + str(j)
            while(int(edge) == 0):
              edge = input("enter weight of edge between " + str(i) + " and " + str(j) + ":")
            self.graph[i][j]= int(edge)
            self.graph[j][i]= int(edge)
            edge=0
        else:
          self.graph[i][j] = 922337
```

```
                self.graph[j][i] = 922337

        #self.graph = [[922337,2,5,4],[2,922337,3,9],[5,3,922337,1],[4,9,1,922337]]

    #This function just displays the graph in adjacency graph
    def print_graph(self):
        print self.graph

    def update_opt_cycle(self,list):
        self.opt_cycle=list




#This is class represents
class State():
    #Declaration of members of class
    def __init__(self,g,edge_seq,n,id,l1,l2,r,s):
        self.cost=0
        self.include_list = l1
        self.exclude_list = l2
        self.graph = np.zeros((n,n), dtype=int)
        self.graph = g
        self.edge_seq=edge_seq
        self.sequence_id=id
        self.saturated_vertex = s
        self.record=r
        self.degree= int(n-1)

    #This function checks for vertex if it has already incident two edges then it marks other to exclude list or vice
vera
    def update_graph(self,vertex,flag):
        index= []
        print str(vertex) + " Saturated"
        if flag==1:
            for i in self.include_list:
                if self.edge_seq[i][0] == vertex:
                    update_lists(index,self.edge_seq[i][1])
                if self.edge_seq[i][1]== vertex:
                    update_lists(index,self.edge_seq[i][0])
            for j in range(0,self.degree+1):
                if j not in index:
                    self.graph[vertex][j]=922337
                    update_lists(self.exclude_list,j)
                    print "in update graph appended vertex" + str(j)
                    print "excluded appended " + str(index)
        if flag==0:
            for i in self.exclude_list:
                if self.edge_seq[i][0] == vertex:
                    update_lists(index,self.edge_seq[i][1])
                if self.edge_seq[i][1]== vertex:
                    update_lists(index,self.edge_seq[i][0])
            for j in range(0,self.degree+1):
```

```
            if j not in index:
                update_lists(self.include_list,j)
                print "in update graph appended vertex" + str(j)

    #Cost calculation
    def calculate_cost(self):
        cost = 0
        print self.graph
        itr = self.graph.argsort()
        for i in range(0, self.degree + 1):
            x= itr[i][0]
            y= itr[i][1]
            cost = cost + self.graph[i][x] + self.graph[i][y]
            print str(self.graph[i][x]) +" "+ str(self.graph[i][y])

        self.cost = cost/2
        return self.cost


    def check_incidence(self):
        #In case of root
        self.calculate_cost()
        return True

    # This is the function checks if selected edge can be excluded or included and for possible states it calculates
cost
    def check_exclude(self,index):
        i = self.edge_seq[index][0]
        j = self.edge_seq[index][1]
        # If vertex already have two incedent edges
        if i in self.saturated_vertex or j in self.saturated_vertex:
            return False
        print "check incidence Exclude section"
        print "include-" + str(self.include_list)
        print "exclude-" + str(self.exclude_list)
        update_lists(self.exclude_list,index)
        print "in check incidence excluded appended " + str(index)
        print "Lists updated immideatly E:"
        print "include-" + str(self.include_list)
        print "exclude-" + str(self.exclude_list)

        self.record[i][1] = self.record[i][1] - 1
        self.record[j][1] = self.record[j][1] - 1
        if self.degree + self.record[i][1] == 2:
            print "vertex saturated " + str(i)
            update_lists(self.saturated_vertex,i)
            # self.update_graph(self, int(i), 0)
        if self.degree + self.record[j][1] == 2:
            print "vertex saturated " + str(j)
            update_lists(self.saturated_vertex,j)
            # self.update_graph(self, int(i), 0)
        self.graph[i][j] = 922337
        self.graph[j][i] = 922337
        self.calculate_cost()
```

```
        print "Updated incidence: "
        print str(self.record)
        return True

    # This is the function checks if selected edge can be excluded or included and for possible states it calculates cost
    def check_include(self,index):
        # When edge is to be included
        i = self.edge_seq[index][0]
        j = self.edge_seq[index][1]
        # If vertex already have two incedent edges
        if i in self.saturated_vertex or j in self.saturated_vertex:
            return False
        print "check incidence Include section"
        f = 0
        v = -1
        print "include-" + str(self.include_list)
        print "exclude-" + str(self.exclude_list)
        update_lists(self.include_list,index)
        print "in check incidence included appended " + str(index)
        print "Lists updated immideatly I:"
        print "include-" + str(self.include_list)
        print "exclude-" + str(self.exclude_list)

        self.record[i][0] = self.record[i][0] + 1
        self.record[j][0] = self.record[j][0] + 1
        if self.record[i][0] == 2:
            f = f + 1
            print "vertex saturated " + str(i)
            update_lists(self.saturated_vertex,i)
            #self.update_graph(int(i), 1)
            v = i
        if self.record[j][0] == 2:
            f = f + 1
            print "vertex saturated " + str(i)
            update_lists(self.saturated_vertex,j)
            #self.update_graph(int(j), 1)
            v = j
        if f == 0:
            cost = 0
            print self.graph
            itr = self.graph.argsort()
            for x in range(0, self.degree + 1):
                if x is i:
                    cost = cost + self.graph[x][itr[x][0]] + self.graph[i][j]
                    print  str(self.graph[x][itr[x][0]]) + " " + str(self.graph[i][j])
                elif x is j:
                    cost = cost + self.graph[x][itr[x][0]] + self.graph[j][i]
                    print str(self.graph[x][itr[x][0]]) + " " + str(self.graph[j][i])
                else:
                    cost = cost + self.graph[x][itr[x][0]] + self.graph[x][itr[x][1]]
                    print str(self.graph[x][itr[x][0]]) + " " + str(self.graph[x][itr[x][1]])
            self.cost = cost / 2
        if f == 1:
```

```
            cost = 0
            print self.graph
            itr = self.graph.argsort()
            for x in range(0, self.degree + 1):
                if x == v:
                    if v == i:
                        w = j
                    else:
                        w = i
                    cost = cost + self.graph[x][itr[x][0]] + self.graph[v][w]
                    print str(self.graph[x][itr[x][0]]) + " " + str(self.graph[v][w])

                else:
                    cost = cost + self.graph[x][itr[x][0]] + self.graph[x][itr[x][1]]
                    print str(self.graph[x][itr[x][0]]) + " " + str(self.graph[x][itr[x][1]])
            self.cost = cost / 2
        if f == 2:
            self.calculate_cost()
        print "Updated incidence: "
        print str(self.record)
        print "Lists updated"
        print "include-" + str(self.include_list)
        print "exclude-" + str(self.exclude_list)
        return True

    #This function checks if the
    def is_valid_cycle(self):
        if len(self.saturated_vertex) == self.degree +1:
            list = []
            for i in range(0,self.degree+1):
                for j in range(0,self.degree+1):
                    if self.graph[i][j] is not 922337:
                        update_lists(list,j)
            if len(list)== (self.degree + 1 ) * 2:
                for i in range(0,self.degree+1):
                    if list.count(i) is not 2:
                        print "Not a vaid cycle!!"
                        return False
                print "Woho!! Vallid cycle!"
                return True

        else:
            print "Not a vaid cycle!!"
            return False


def generate_edge_seq(g):
    n =g.vertex_count
    n= n*(n - 1) / 2
    index=0
    edge_seq = np.zeros((n,2),dtype=int)
    for i in range(0, g.vertex_count):
        for j in range(0, g.vertex_count):
            if i is not j:
```

```
            if i<j:
                edge_seq[index][0]= int(i)
                edge_seq[index][1]= int(j)
                index = index + 1
    return edge_seq

#generation of Root state
def generate_root(edge_seq):
    l1=[]
    l2=[]
    l=[]
    r = np.zeros((g.vertex_count,2), dtype=int)
    sroot= State(g.graph,edge_seq,g.vertex_count,-1,l1,l2,r,l)
    cost = sroot.check_incidence()
    print "Root generated with cost:"
    print cost
    return sroot

#Thiis function expands the tree after root
def expand_tree(q,lowerbound):
    local = q.pop()
    print "node poped -" + str(local.cost)
    print "include-"+ str(local.include_list)
    print "exclude-" + str(local.exclude_list)
    print lowerbound
    #Check if cycle with less cost is already found
    #pruning
    if local.cost < lowerbound:
        id = local.sequence_id
        id = id + 1
        print "id :" + str(id)
        print "lesser lower bound"

        #Node generated with the edge
        along = State(local.graph,local.edge_seq,local.degree +
1,id,local.include_list,local.exclude_list,local.record,local.saturated_vertex)
        without = State(local.graph, local.edge_seq, local.degree + 1, id, local.include_list,
local.exclude_list,local.record, local.saturated_vertex)
        print "Lists Generated"
        print "include-" + str(along.include_list)
        print "exclude-" + str(along.exclude_list)
        if along.check_include(id):
            print "Node generated along edge " + str(along.edge_seq[id]) + " with cost: "
            print along.cost
            if along.is_valid_cycle():
                lowerbound=along.cost
            else:
                q.push(along,along.cost)
        # Node generated with the edge


        print "Lists Generated:"
        print "include-" + str(without.include_list)
        print "exclude-" + str(without.exclude_list)
```

```
        if without.check_exclude(id):
            print "Node generated without edge " + str(without.edge_seq[id]) + " with cost: "
            print without.cost
            if without.is_valid_cycle():
                lowerbound = without.cost
            else:
                q.push(without, without.cost)
    else:
        return lowerbound


def completion_check(g,q):
    if len(g.opt_cycle) is not 0:
        if q:
            return False
    return True

#This function generates the root and expand the logical tree till optimal cycle is found
def tsp(g):
    lower_bound =999999
    edge_seq = generate_edge_seq(g)
    q = PriorityQueue()
    if q:
        root = generate_root(edge_seq)
        q.push(root,root.cost)
    else:
        print "Error!!"
    while completion_check(g,q):
        expand_tree(q,lower_bound)
    return


#main function contring entire flow
if __name__ == "__main__":
    g= garph_class(input("Number of vertices:"))
    g.generate_edges()
    g.print_graph()
    tsp(g)
    print "The optimal cycle is:"
    print g.opt_cycle
```