Week1:

import networkx as nx

def cycle\_length(g, cycle):

# Checking that the number of vertices in the graph equals the number of vertices in the cycle.

assert len(cycle) == g.number\_of\_nodes()

# Write your code here.

w = 0

cycle.append(cycle[0])

for i in range(len(cycle)-1):

w += sum(g[cycle[i]][cycle[i+1]].values())

return w

# Here is a test case:

# Create an empty graph.

g = nx.Graph()

# Now we will add 6 edges between 4 vertices

g.add\_edge(0, 1, weight = 2)

# We work with undirected graphs, so once we add an edge from 0 to 1, it automatically creates an edge of the same weight from 1 to 0.

g.add\_edge(1, 2, weight = 2)

g.add\_edge(2, 3, weight = 2)

g.add\_edge(3, 0, weight = 2)

g.add\_edge(0, 2, weight = 1)

g.add\_edge(1, 3, weight = 1)

# Now we want to compute the lengths of two cycles:

cycle1 = [0, 1, 2, 3]

cycle2 = [0, 2, 1, 3]

# assert(cycle\_length(g, cycle1) == 8)

# assert(cycle\_length(g, cycle2) == 6)

1.

Question 1

Implement the brute force algorithm for the Traveling Salesman Problem. The algorithm should check all the permutations of the vertices and return the minimum weight of a cycle visiting each vertex exactly once.

**1 / 1 point**

import time

import networkx as nx

from itertools import permutations

def all\_permutations(g):

n = g.number\_of\_nodes()

shortest\_tour = None

for tour in permutations(range(n)) :

tour\_len = 0

valid = True

# length of the tour

for i in range(len(tour)) :

if not g.has\_edge(tour[i], tour[i-1]) :

valid = False

break

else :

tour\_len += g[tour[i-1]][tour[i]]['weight']

# may be a valid cycle with the shortest length

if valid and (shortest\_tour is None or tour\_len < shortest\_tour) :

shortest\_tour = tour\_len

return shortest\_tour

1.

Question 1

Compute the average weight of a Hamiltonian cycle in the given graph.

import networkx as nx

from itertools import permutations

def cycle\_length2(g, cycle):

# Checking that the number of vertices in the graph equals the number of vertices in the cycle.

assert len(cycle) == g.number\_of\_nodes()

w = 0

cycle += (cycle[0],)

for i in range(len(cycle)-1):

#print(cycle1[i], cycle1[i+1])

w += sum(g[cycle[i]][cycle[i+1]].values())

return w

import math

def average(g):

n = g.number\_of\_nodes()

sum\_of\_weights = sum(g[i][j]['weight'] for i in range(n) for j in range(i))

return 2\*sum\_of\_weights/(n-1)

1.

Question 1

Implement the Nearest Neighbors Heuristic for the Traveling Salesman Problem. Your algorithm should start with the vertex number 0, and then each time select the closest vertex among the ones which don't yet belong to the cycle.

import networkx as nx

# This function takes as input a graph g.

# The graph is complete (i.e., each pair of distinct vertices is connected by an edge),

# undirected (i.e., the edge from u to v has the same weight as the edge from v to u),

# and has no self-loops (i.e., there are no edges from i to i).

#

# The function should return the weight of the nearest neighbor heuristic, which starts at the vertex number 0,

# and then each time selects a closest vertex.

def nearest\_neighbors(g):

current\_node = 0

path = [current\_node]

n = g.number\_of\_nodes()

# We'll repeat the same routine (n-1) times

for \_ in range(n - 1):

next\_node = None

# The distance to the closest vertex. Initialized with infinity.

min\_edge = float("inf")

for v in g.nodes():

if g[current\_node][v]['weight'] < min\_edge and v not in path:

# Write your code here: decide if v is a better candidate than next\_node.

# If it is, then update the values of next\_node and min\_edge

min\_edge = g[current\_node][v]['weight']

next\_node = v

assert next\_node is not None

path.append(next\_node)

current\_node = next\_node

weight = sum(g[path[i]][path[i + 1]]['weight'] for i in range(g.number\_of\_nodes() - 1))

weight += g[path[-1]][path[0]]['weight']

return weight

# You might want to copy your solution to your Jupiter Notebook to see how close this heuristic is to the optimal solution.

We2:

**Branch and Bound**

**LATEST SUBMISSION GRADE**

100%

1.

Question 1

Implement the Branch-and-Bound algorithm for the Traveling Salesman problem.

**1 / 1 point**



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

76

77

78

79

80

81

82

83

84

85

86

87

88

89

90

91

92

93

94

95

96

97

98

99

100

101

102

103

104

105

106

107

108

109

import networkx as nx

import time

import math

from itertools import permutations

# Lower bound used as a hueristic in this searching problem where there are many

  extensions of

# the current subpth but we need to invade only the subgraphs that may have a

  solution (cycle)

# better than the one we have Lower Bound Here is the MST cost plus the cost of

  the subpth

# if this Low\_bound >= the current solution we ignore this branch of extension

def lower\_bound(g, sub\_cycle):

# The weight of the current path.

current\_weight = sum([g[sub\_cycle[i]][sub\_cycle[i + 1]]['weight'] for i in

      range(len(sub\_cycle) - 1)])

# For convenience we create a new graph which only contains vertices not

      used by g.

unused = [v for v in g.nodes() if v not in sub\_cycle]

h = g.subgraph(unused)

# Compute the weight of a minimum spanning tree.

t = list(nx.minimum\_spanning\_edges(h))

print(t)

mst\_weight = sum([h.get\_edge\_data(e[0], e[1])['weight'] for e in t])

# If the current sub\_cycle is "trivial" (i.e., it contains no vertices or

      all vertices), then our lower bound is

# just the sum of the weight of a minimum spanning tree and the current

      weight.

if len(sub\_cycle) == 0 or len(sub\_cycle) == g.number\_of\_nodes():

return mst\_weight + current\_weight

# If the current sub\_cycle is not trivial, then we can also add the weight

      of two edges connecting the vertices

# from sub\_cycle and the remaining part of the graph.

# s is the first vertex of the sub\_cycle

s = sub\_cycle[0]

# t is the last vertex of the sub\_cycle

t = sub\_cycle[-1]

# The minimum weight of an edge connecting a vertex from outside of

      sub\_sycle to s.

min\_to\_s\_weight = min([g[v][s]['weight'] for v in g.nodes() if v not in

      sub\_cycle])

# The minimum weight of an edge connecting the vertex t to a vertex from

      outside of sub\_cycle.

min\_from\_t\_weight = min([g[t][v]['weight'] for v in g.nodes() if v not in

      sub\_cycle])

# Any cycle which starts with sub\_cycle must be of length:

# the weight of the edges from sub\_cycle +

# the minimum weight of an edge connecting sub\_cycle and the remaining

      vertices +

# the minimum weight of a spanning tree on the remaining vertices +

# the minimum weight of an edge connecting the remaining vertices to

      sub\_cycle.

return current\_weight + min\_from\_t\_weight + mst\_weight + min\_to\_s\_weight

# The branch and bound procedure takes

# 1. a graph g;

# 2. the current sub\_cycle, i.e. several first vertices of cycle under

  consideration.

# Initially sub\_cycle is empty;

# 3. currently best solution current\_min, so that we don't even consider paths

  of greater weight.

# Initially the min weight is infinite

def branch\_and\_bound(g, sub\_cycle=None, current\_min=float("inf")):

# If the current path is empty, then we can safely assume that it starts

      with the vertex 0.

if sub\_cycle is None:

sub\_cycle = [0]

# If we already have all vertices in the cycle, then we just compute the

      weight of this cycle and return it.

if len(sub\_cycle) == g.number\_of\_nodes():

weight = sum([g[sub\_cycle[i]][sub\_cycle[i + 1]]['weight'] for i in range

          (len(sub\_cycle) - 1)])

weight = weight + g[sub\_cycle[-1]][sub\_cycle[0]]['weight']

return weight

# Now we look at all nodes which aren't yet used in sub\_cycle.

unused\_nodes = list()

for v in g.nodes():

if v not in sub\_cycle:

unused\_nodes.append((g[sub\_cycle[-1]][v]['weight'], v))

# We sort them by the distance from the "current node" -- the last node in

      sub\_cycle.

unused\_nodes = sorted(unused\_nodes)

for (d, v) in unused\_nodes:

assert v not in sub\_cycle

extended\_subcycle = list(sub\_cycle)

extended\_subcycle.append(v)

# For each unused vertex, we check if there is any chance to find a

          shorter cycle if we add it now.

if lower\_bound(g, extended\_subcycle) < current\_min:

# If there is such a chance, we add the vertex to the current cycle,

              and proceed recursively.

temp\_min = branch\_and\_bound(g, extended\_subcycle, current\_min)

# If we found a short cycle, then we update the current\_min value.

current\_min = temp\_min if temp\_min < current\_min else current\_min

# The procedure returns the shortest cycle length.

return current\_min

def dist(point1, point2) :

return math.sqrt((point1[0] - point2[0])\*\*2 + (point1[1] - point2[1])\*\*2)

def main():

G = nx.Graph()

points = [(199, 59), (152, 117), (68, 87), (281, 161), (11, 53), (254, 227)]

for i in range(len(points)) :

for j in range(i+1, len(points)) :

G.add\_edge(i, j, weight=dist(points[i], points[j]))

print(list(nx.minimum\_spanning\_tree(G)))

# start\_time = time.time()

# print(branch\_and\_bound(G))

# end\_time = time.time()

# print(end\_time - start\_time)

if \_\_name\_\_ == '\_\_main\_\_':

main()

Run

Week3:

import networkx as nx

import math

# This function takes as input a graph g.

# The graph is complete (i.e., each pair of distinct vertices is connected by an

  edge),

# undirected (i.e., the edge from u to v has the same weight as the edge from v

  to u),

# and has no self-loops (i.e., there are no edges from i to i).

#

# The function should return a 2-approximation of an optimal Hamiltonian cycle.

def approximation(g):

# n is the number of vertices.

n = g.number\_of\_nodes()

# You might want to use the function "nx.minimum\_spanning\_tree(g)"

# MST of the graph

mst = nx.minimum\_spanning\_tree(g)

order = list(nx.dfs\_preorder\_nodes(mst, 0))

# weight of the cycle

weight = sum(g[order[i-1]][order[i]]['weight'] for i in range(len(order)))

return weight

def dist(point1, point2) :

return math.sqrt((point1[0] - point2[0])\*\*2 + (point1[1] - point2[1])\*\*2)

def main():

G = nx.Graph()

points = [(199, 59), (152, 117), (68, 87), (281, 161), (11, 53), (254, 227)]

for i in range(len(points)) :

for j in range(i+1, len(points)) :

G.add\_edge(i, j, weight=dist(points[i], points[j]))

print(approximation(G))

if \_\_name\_\_ == '\_\_main\_\_':

main()