This exercise provides examples to solve the following problems:

1) Spanning tree and greedy algorithm

2) Shortest-path, min-cost flow algorithm based on linear integer program

3) TSP/vehicle routing

As usual, we have to import the library at the top of the file. For this exercise, we need gurobipy library to solve the linear integer program.

You can refer to the following instructions to get gurobipy. http://www.gurobi.com/downloads/get-anaconda

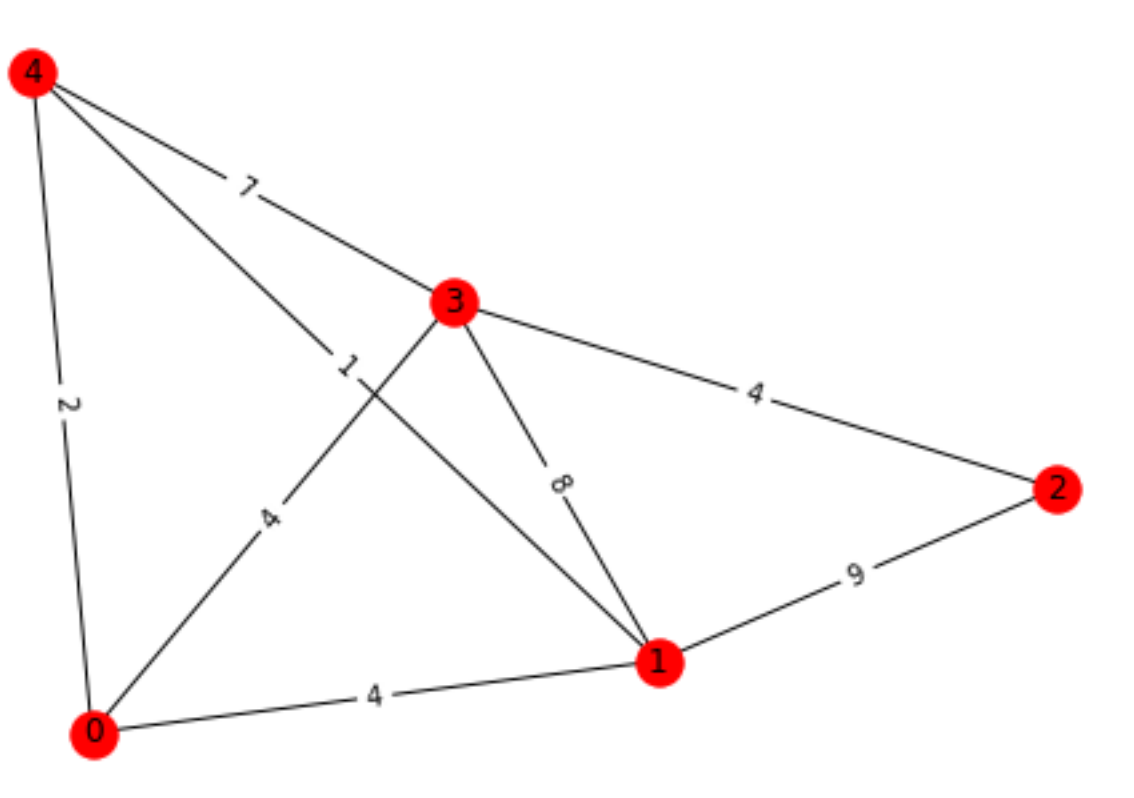
You will also need to get an academic license. https://user.gurobi.com/download/licenses/free-academic

In [1]:
```
# Import networkx library and rename it as nx.
import networkx as nx

# Adding support for large, multi-dimensional arrays and matrices.
import numpy as np

import sys
import matplotlib.pyplot as plt
from gurobipy import *
```

We use the input file "edgelist1.txt" for this example.

We express the graph using networkx library, and draw the graph on the screen with the node name and edge weight. This is achieved by getting the positions of every node in a particular layout (spring in this case). Then we use the function "draw_networkx_edge_labels" to plot the graph and the edge weights based on the obtained position. Note that this fragment of code is also used in exercise 2.

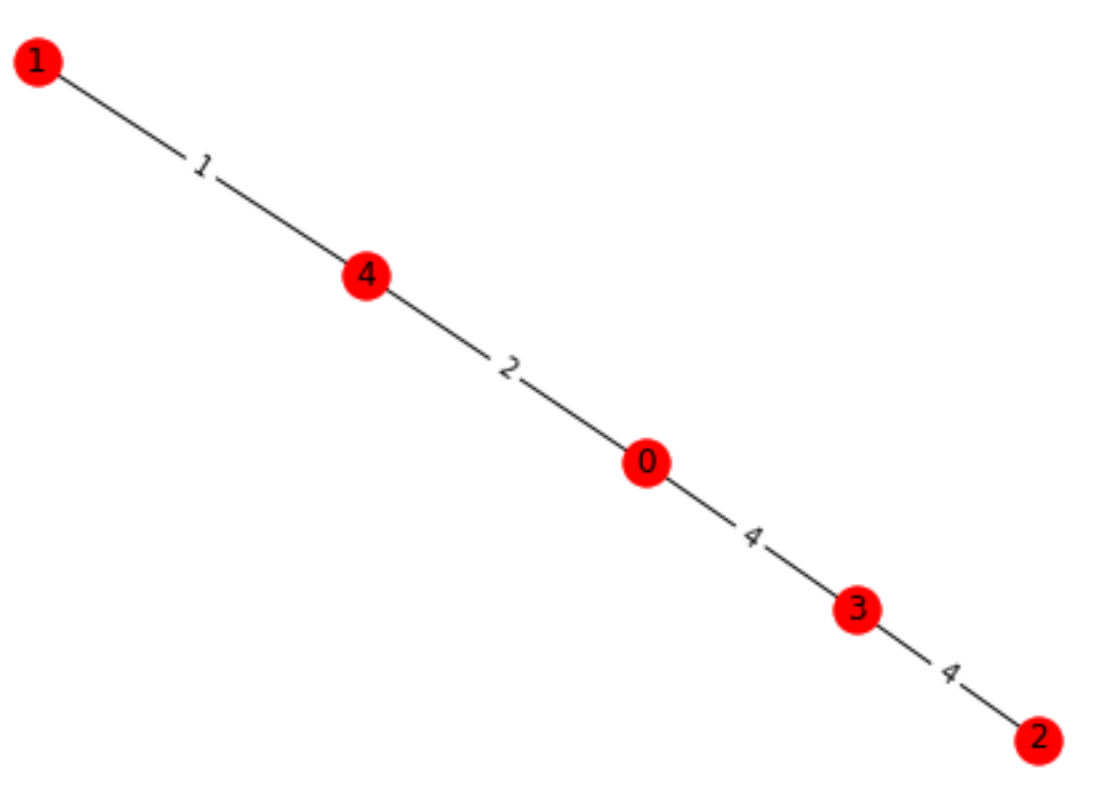
In [15]:
```
place_holder = "./"

# Read an un-directed graph from a list of edges
G0 = nx.read_edgelist(place_holder + "edgelist1.txt", create_using = nx.Graph(), nodetype = str,
                      data = (('weight', int),))


"""
A display function that shows the graph and the edge weight. The codes are adapted from the above.

Args:
    graph_w: The undirected graph with edge weight

Returns:
    Null
"""

def showGraphwithWeight(graph_w):
    pos = nx.spring_layout(graph_w)

    # draw the graph with the node position based on pos
    nx.draw(graph_w, pos, with_labels = True)

    labels = nx.get_edge_attributes(graph_w,'weight')

    # draw the edge weight
    nx.draw_networkx_edge_labels(graph_w, pos, edge_labels = labels)


showGraphwithWeight(G0)
```



Example 1: Spanning tree algorithm

We apply the function 'minimum_spanning_tree()' from Networkx library to get the minimum spanning tree for the graph G. Basically, this function returns a minimum spanning tree or forest of an undirected weighted graph. It uses a greedy algorithm which is a general strategy of picking the next-best while maintaining feasibility.

In [3]:
```
# calculate minimum spanning tree for G
min_span_G0 = nx.minimum_spanning_tree(G0)
showGraphwithWeight(min_span_G0)
```



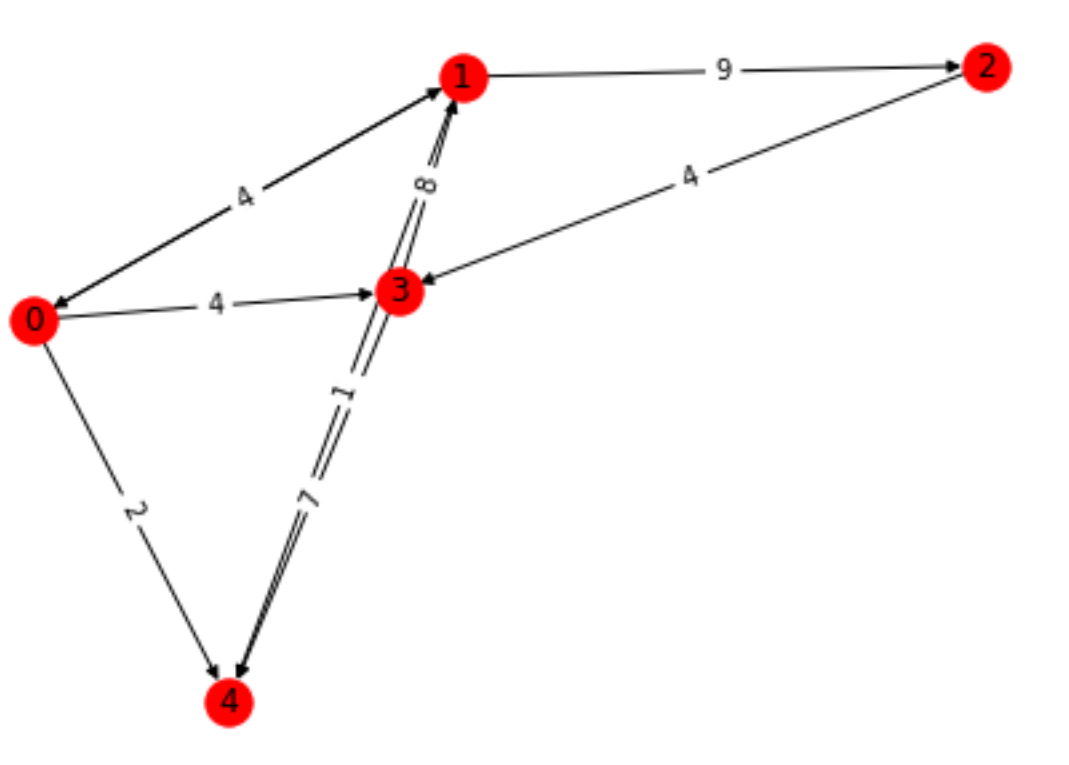
Example 2a: Shortest-path, min-cost flow algorithm based on linear integer program

We also use the same input file "edgelist1.txt" in this example. However, we treat it as a directed graph when we read the edge list.

Let's say we want to get the shortest path from node '2' to '0'. To calculate the shortest-path, we need to formulate the graph as a non-edge incidence_matrix A, a cost vector c and a column vector b such that transpose(c).x is minimized for Ax=b.

In [17]:
```
# Read an un-directed graph from a list of edges
G1 = nx.read_edgelist(place_holder + "edgelist1.txt", create_using = nx.DiGraph(), nodetype = str,
                      data = (('weight', int),))

showGraphwithWeight(G1)

# We use the function incidence_matrix() to get the incidence matrix of G
A1 = nx.incidence_matrix(G1, oriented = True)
A1_array = A1.toarray().astype(np.int)

print("The node-edge incidence matrix")
print(A1_array)
```

```
The node-edge incidence matrix
[[-1 -1 -1  1  0  0  0  0  0]
 [ 1  0  0 -1 -1 -1  0  1  0]
 [ 0  0  0  0  1  0 -1  0  0]
 [ 0  1  0  0  0  0  1 -1 -1]
 [ 0  0  1  0  0  1  0  0  1]]
```



We then formulate the cost vector c and vector b.

In [5]:
```
# Obtain the incidence matrix again, but weighted
A1_w = nx.incidence_matrix(G1, oriented = True, weight='weight')

# We then get the maximum value of each column, which is equivalent to the cost vector
c1 = A1_w.toarray().astype(np.int).max(0)

# Since we know where the start and end nodes are, we can formulate the vector b directly.
b1 = [1, 0 ,-1, 0, 0]

print("The cost vector:")
print(c1)

print("The vector b")
print(b1)
```

```
The cost vector:
[4 4 2 4 9 1 4 8 7]
The vector b
[1, 0, -1, 0, 0]
```

We can then apply gurobipy, a python wrapper to the optimization tool Gurobi, to solve the shortest path problem based on linear integer program.

The first step is to create a model that holds a single optimization problem. Then we add the decision variables in the same model.

In [6]:
```
m1 = Model("shortest_path")

# Get the start and end node of an edge, and make them as edge name
start_nodes = A1_array.argmin(axis=0)
end_nodes = A1_array.argmax(axis=0)

# Get the number of edges
num_edges = G1.number_of_edges()
# Get the number of nodes
num_nodes = G1.number_of_nodes()

# Define an empty variable list for future use
varlist = []

for i in range(0, num_edges):

    # Name the edge
    edge_str = 'edge'+str(start_nodes[i])+'_'+str(end_nodes[i])

    # Add the variables in the model
    var = m1.addVar(vtype = GRB.BINARY, name=edge_str)
    varlist.append(var)

m1.update()
```

```
Academic license - for non-commercial use only
```

After that, we are going to provide an objective function to the model. The objective function is:

4 *edge0-1* + 4 edge0-3 + 4 *edge1-0* + 9 edge1-2 + 1 *edge1-4* + 4 edge2-3 + 8 *edge3-1* + 7 edge3-4

In [7]:
```
objValue = quicksum(c1[i] * varlist[i] for i in range(num_edges))

# We have to minimize the objective function.
m1.setObjective(objValue, GRB.MINIMIZE)

m1.update()
```

We also have to write up all the constraints for the model. Each constraint is basically the addition of one row of the incidence matrix. The corresponding result has to equal to the same row of the vector b.

In [8]:
```
# iterate the incidence matrix to get the constraint
for j in range(num_nodes):
    m1.addConstr(quicksum(A1_array[j, k] * varlist[k] for k in range(num_edges)) == b1[j])


for k in range(num_edges):
    m1.addConstr(0<= varlist[k]<=1)

m1.update()
```

We can call the optimize() function to perform optimization, which in turn solve the shortest path problem.

In [9]:
```
m1.optimize()

# print the final value for x
print('The final value for x:')
for v in m1.getVars():
    print(v.varName, v.x)
```

```
Optimize a model with 14 rows, 9 columns and 27 nonzeros
Variable types: 0 continuous, 9 integer (9 binary)
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
  Objective range  [1e+00, 9e+00]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 1e+00]
Presolve removed 14 rows and 9 columns
Presolve time: 0.00s
Presolve: All rows and columns removed

Explored 0 nodes (0 simplex iterations) in 0.02 seconds
Thread count was 1 (of 4 available processors)

Solution count 1: 16

Optimal solution found (tolerance 1.00e-04)
Best objective 1.600000000000e+01, best bound 1.600000000000e+01, gap 0.0000%
The final value for x:
edge0_1 0.0
edge0_3 0.0
edge0_4 0.0
edge1_0 1.0
edge1_2 0.0
edge1_4 0.0
edge2_3 1.0
edge3_1 1.0
edge3_4 0.0
```

The edge with a number '1' in the solution indicates that edge is on the shortest path.

Now, let's check the solution against the traditional algorithm. This can be done by calling the function 'shortest_path' from Networkx library.

In [10]:
```
path_2_0 = nx.shortest_path(G1, source ='2',target = '0')
print("The shortest path from 2 to 0: ", path_2_0)
```

```
The shortest path from 2 to 0:  ['2', '3', '1', '0']
```

Example 3: TSP/vehicle routing

In this example, we will use another optimization tool, Google OR-Tools, to solve the travelling salesman problem. This example makes reference to the TSP on OR-Tools webpage. https://developers.google.com/optimization/routing/tsp

You can find the installation instruction here: https://developers.google.com/optimization/install/

Mathematically, traveling salesman problems can be represented as a graph, where the locations are the nodes and the edges (or arcs) represent direct routes between the nodes. We use the same undirected graph G0 and treat that as a map for the TSP.

In [11]:
```
# Import Google OR-Tools
from ortools.constraint_solver import routing_enums_pb2
from ortools.constraint_solver import pywrapcp


# Get the adjacency matrix for the undirected graph G0
A0 = nx.adjacency_matrix(G0, weight='weight')
A0_amat = A0.toarray().astype(np.int)

print("The adjacency matrix:")
print(A0_amat);

# Hard code the node name
node_name = [0, 1, 2, 3, 4]

print("The node name:")
print(node_name)
```

```
The adjacency matrix:
[[0 4 0 4 2]
 [4 0 9 8 1]
 [0 9 0 4 0]
 [4 8 4 0 7]
 [2 1 0 7 0]]
The node name:
[0, 1, 2, 3, 4]
```

In [12]:
```
"""
To use the routing solver, you need to create a distance (or transit) callback: a function
that takes any pair of locations and returns the distance between them. The easiest way
to do this is using the distance matrix.

If it is too complicated, please feel free to skip it. You just have to copy this function
when you use OR-Tools.

Args:
    from_index, to_index

Returns:
    The callback function that calculate the distance.
"""

def distance_callback(from_index, to_index):
    """Returns the distance between the two nodes."""
    # Convert from routing variable Index to distance matrix NodeIndex.
    from_node = manager.IndexToNode(from_index)
    to_node = manager.IndexToNode(to_index)
    return A0_amat[from_node][to_node]


tsp_size = len(node_name)

num_routes = 1
depot = 0

# Create routing model
if tsp_size > 0:
    # The solver instance is declared by:
    # The parameters are: tsp_size - The number of cities,
    #                     num_routes — The number of routes,
    #                     depot — The start and end node of the route
    manager = pywrapcp.RoutingIndexManager(tsp_size, num_routes, depot)
    routing = pywrapcp.RoutingModel(manager)

    transit_callback_index = routing.RegisterTransitCallback(distance_callback)
    # Set the cost of travel
    routing.SetArcCostEvaluatorOfAllVehicles(transit_callback_index)

    # sets the default search parameters and a heuristic method for
    # finding the first solution
    search_parameters = pywrapcp.DefaultRoutingSearchParameters()
    search_parameters.first_solution_strategy = (
    routing_enums_pb2.FirstSolutionStrategy.PATH_CHEAPEST_ARC)
```

After initalization, we can invoke the solver and display the results.

In [13]:
```
solution = routing.SolveWithParameters(search_parameters)

"""Prints solution on console."""
print('Objective: {}'.format(solution.ObjectiveValue()))
index = routing.Start(0)
plan_output = 'Route for vehicle 0:\n'
route_distance = 0
while not routing.IsEnd(index):
    plan_output += ' {} ->'.format(manager.IndexToNode(index))
    previous_index = index
    index = solution.Value(routing.NextVar(index))
    route_distance += routing.GetArcCostForVehicle(previous_index, index, 0)
plan_output += ' {}\n'.format(manager.IndexToNode(index))
print(plan_output)
plan_output += 'Route distance: {}miles\n'.format(route_distance)
```

```
Objective: 13
Route for vehicle 0:
 0 -> 2 -> 4 -> 1 -> 3 -> 0
```

In [ ]: