Caleb Logemann
MATH 566 Discrete Optimization
Final

1. Emperor has decided to build Death Star II. He needs money to pay for it. He sent you to collect taxes from several planets. Here are the coordinates of the planets assigned to you. You are starting at Corusant (0,0) and want to return there. The time of travel between the planets corresponds to their Euclidean distance. You have to visit all planets. Minimize the travel time.

```
X = [  [0,0], [9,0], [1,1], [8,1], [0,2], [9,2], [4,3], [5,3], \
[0,4], [1,4], [2,4], [3,4], [6,4], [7,4], [8,4], [9,4], \
[0,5], [1,5], [3,5], [2,5], [6,5], [7,5], [8,5], [9,5], \
[4,6], [5,6], [0,7], [9,7], [1,8], [8,8], [0,9], [9,9] ];
```

This is a traveling salesman problem. There are several approximation algorithms that can find good solutions. I have programmed the nearest neighbor algorithm, the cheapest insertion algorithm, and the furthest insertion algorithm in the following script. This script also has a function to compute a lower bound on the traveling salesman tour.

```
import itertools as it
load('kruskal.sage')
def nearestNeighbor(points):
    pointList = list(points)
    n = len(pointList)
    cycle = [pointList.pop()]
    total_distance = 0
    for i in range(n-1):
        j = min(enumerate(pointList), key=lambda x:distance(cycle[-1], x
            ↪ [1]))[0]
        cycle.append(pointList.pop(j))
        total_distance += distance(cycle[-1], cycle[-2])

    total_distance+=distance(cycle[0], cycle[-1])
    return (cycle, N(total_distance))

def cheapestInsertion(points):
    pointList = list(points)
    n = len(pointList)
    cycle = [pointList.pop(), pointList.pop()]
    total_distance = distance(cycle[0], cycle[1])
    for i in range(n-2):
        minVertexIndex = None
        minDistance = None
        # find vertex not in cycle closest to vertex in cycle
        for j in range(len(pointList)):
            d = min([distance(pointList[j], x) for x in cycle])
            if d < minDistance or minDistance == None:
                minVertexIndex = j
                minDistance = d
```

```
        # find place to insert in cycle that minimizes increase in total
            ↪ length
        minCycleIndex = None
        minDistanceChange = None
        for j in range(len(cycle)):
            if j == len(cycle) - 1:
                jPlus = 0
            else:
                jPlus = j + 1
            d = distance(cycle[j], pointList[minVertexIndex]) \
                + distance(cycle[jPlus], pointList[minVertexIndex]) \
                - distance(cycle[j], cycle[jPlus])
            if d < minDistanceChange or minDistanceChange == None:
                minDistanceChange = d
                minCycleIndex = j
        total_distance += minDistanceChange
        cycle.insert(minCycleIndex+1, pointList.pop(minVertexIndex))

    total_distance += distance(cycle[0], cycle[-1])
    return (cycle, N(total_distance))

def furthestInsertion(points):
    pointList = list(points)
    n = len(pointList)
    # find initial max distance
    maxDistance = None
    maxIndexA = None
    maxIndexB = None
    for a in range(n-1):
        b = max(enumerate(pointList[a+1:]), key=lambda x:distance(
            ↪ pointList[a], x[1]))[0] + a + 1
        d = distance(pointList[a], pointList[b])
        if d > maxDistance or maxDistance == None:
            maxDistance = d
            maxIndexA = a
            maxIndexB = b

    pointA = pointList[maxIndexA]
    pointB = pointList[maxIndexB]
    cycle = [pointA, pointB]
    pointList.remove(pointA)
    pointList.remove(pointB)
    total_distance = 2*distance(cycle[0], cycle[1])
    for i in range(n-2):
        maxVertexIndex = None
        maxDistance = None
        # find vertex not in cycle farthest from vertex in cycle
        for j in range(len(pointList)):
            d = max([distance(pointList[j], x) for x in cycle])
            if d > maxDistance or maxDistance == None:
```

```
                maxVertexIndex = j
                maxDistance = d

        minCycleIndex = None
        minDistanceChange = None
        for j in range(len(cycle)):
            if j == len(cycle) - 1:
                jPlus = 0
            else:
                jPlus = j + 1
            d = distance(cycle[j], pointList[maxVertexIndex]) \
                + distance(cycle[jPlus], pointList[maxVertexIndex]) \
                - distance(cycle[j], cycle[jPlus])
            if d < minDistanceChange or minDistanceChange == None:
                minDistanceChange = d
                minCycleIndex = j
        total_distance += minDistanceChange
        cycle.insert(minCycleIndex+1, pointList.pop(maxVertexIndex))
    return (cycle, N(total_distance))

def lowerBound(points):
    LB = 0
    for v in points:
        vertexList = list(points)
        vertexList.remove(v)
        n = len(vertexList)
        edgeList = list(it.combinations(range(n), 2))
        costList = []
        for edge in edgeList:
            u = vertexList[edge[0]]
            v = vertexList[edge[1]]
            costList.append(distance(u, v))
        treeEdges = kruskal(vertexList, edgeList, costList)
        bound = sum([distance(vertexList[e[0]], vertexList[e[1]]) for e
          ↪ in treeEdges])
        w = min(vertexList, key=lambda x:distance(x,v))
        bound += distance(v,w)
        vertexList.remove(w)
        u = min(vertexList, key=lambda x:distance(x,v))
        bound += distance(v,u)
        if bound > LB:
            LB = bound
    return LB

def plotCycle(cycle):
    plot = line([])
    for i in range(len(cycle)):
        plot += disk(cycle[i], 0.1, (0,2*pi), color='red')
        if i < len(cycle)-1:
            plot += line([cycle[i], cycle[i+1]])
```

```
        else:
            plot += line([cycle[i], cycle[0]])
    plot.show()

def distance(a, b):
    diff = [i - j for i,j in zip(a, b)]
    squares = [x^2 for x in diff]
    return sqrt(sum(squares))

def computeTotalDistance(cycle):
    total = 0
    for i in range(len(cycle) - 1):
        total += distance(cycle[i], cycle[i+1])
    total += distance(cycle[0], cycle[-1])
    return N(total)
```

The following script runs each of these algorithms on the given set of points and computes a lower bound.

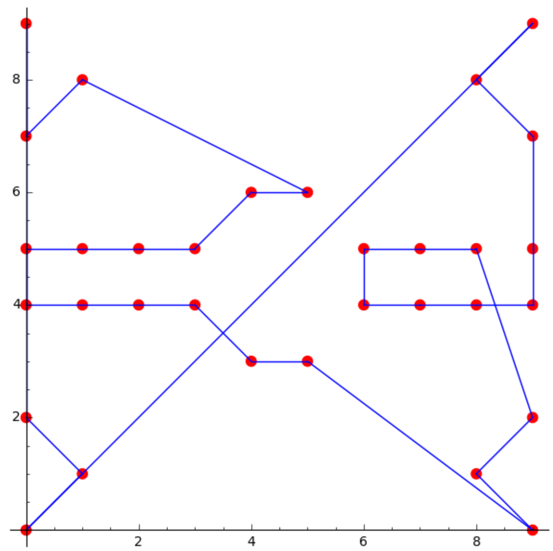
```
load('travelingSalesman.sage')

pointList = [[0,0], [9,0], [1,1], [8,1], [0,2], [9,2], [4,3], [5,3], \
             [0,4], [1,4], [2,4], [3,4], [6,4], [7,4], [8,4], [9,4], \
             [0,5], [1,5], [3,5], [2,5], [6,5], [7,5], [8,5], [9,5], \
             [4,6], [5,6], [0,7], [9,7], [1,8], [8,8], [0,9], [9,9]]


cycle, total_distance = nearestNeighbor(pointList)
print 'Nearest Neighbor'
print('Distance: %s' % total_distance)
plotCycle(cycle)

cycle, total_distance = cheapestInsertion(pointList)
print 'Cheapest Insertion'
print('Distance: %s' % total_distance)
plotCycle(cycle)

cycle, total_distance = furthestInsertion(pointList)
print 'Furthest Insertion'
print('Distance: %s' % total_distance)
plotCycle(cycle)

LB = lowerBound(pointList)
print('Lower Bound: %s' % LB)
```

The output of this script is shown below.

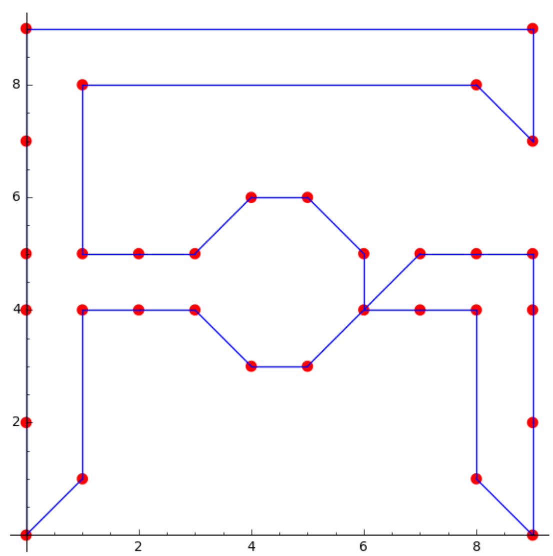
```
Nearest Neighbor
Distance: 65.0902577378837
Cheapest Insertion
Distance: 63.3137084989848
```

```
Furthest Insertion
Distance: 55.0364766805865
Lower Bound: 40.7279220613579
```

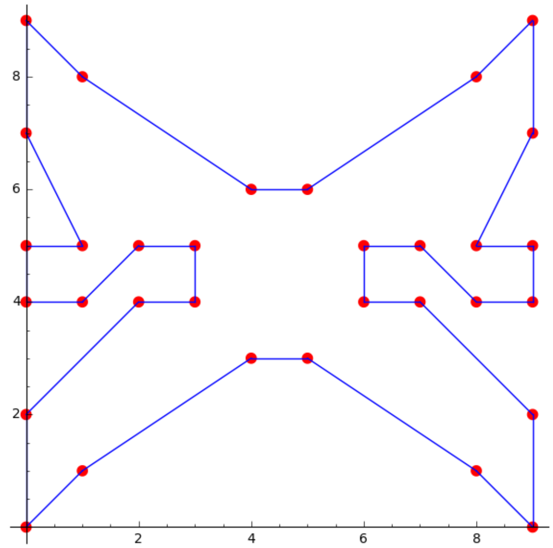
The following plots are also generated. For the nearest neighbor algorithm the following tour was found.



For the cheapest insertion algorithm the following tour was found.



For the furthest insertion algorithm the following tour was found.



The furthest insertion algorithm found the best tour out of all three algorithms with a total distance of 55.036. The lower bound that was found was 40.728.

2. You are leading a group 4 of imperial fighters that are protecting an imperial shuttle transporting secret plans to Death Star II. You were ambushed by 3 rebel ships. You must protect the secret plans! Do not repeat the same mistake that happened with Death Star.

   Choose one fighter from your group to accompany the transport shuttle cover their retreat by fighting the rebel ship. The remaining three ships will each try to fight one rebel ship. You need to fight against each of the rebel ships to prevent pursuit of the transport shuttle and you do not want to leave the transport ship unguarded.

   For the remaining 3 ships that fight the rebel ships, maximize the total damage (sum of damages) caused to the rebel ships. For every pair of imperial and rebel ship, the damage caused by imperial ships to rebel ships is described in the following table.

	r_1	r_2	r_3
i_1	3	2	3
i_2	2	3	1
i_3	4	2	2
i_4	1	5	1

   This problem can be solved using an integer program. Let $x_{ir} \in \{0, 1\}$ be a variable describing whether imperial ship i is going to attach rebel ship r. The damage that imperial ship i can cause to rebel ship r will be given by d_{ir}. These values are constant and are given in the table above. Therefore the objective function for this integer program will be

$$\text{maximize } \sum_{i=1}^{4} \left(\sum_{r=1}^{3} (d_{ir} x_{ir}) \right)$$

   There are several constraints on the variables x_{ir}. The first constraint is each imperial ship can attach at most one rebel ship, that is

$$\sum_{r=1}^{3} (x_{ir}) \leq 1 \text{ for } 1 \leq i \leq 4$$

Also every rebel ship must be attacked by at least one imperial ship.

$$\sum_{i=1}^{4} (x_{ir}) \geq 1 \text{ for } 1 \leq r \leq 3$$

Lastly one ship must guard the transport ship, or in other words at most three imperial ships can attack the rebel ships.

$$\sum_{i=1}^{4} \left(\sum_{r=1}^{3} (x_{ir}) \right) \leq 3$$

The full integer program is thus

$$(IP) \begin{cases} \text{maximize} & \sum_{i=1}^{4} \left(\sum_{r=1}^{3} (d_{ir} x_{ir}) \right) \\ \text{subject to} & \sum_{r=1}^{3} (x_{ir}) \leq 1 \text{ for } 1 \leq i \leq 4 \\ & \sum_{i=1}^{4} (x_{ir}) \geq 1 \text{ for } 1 \leq r \leq 3 \\ & \sum_{i=1}^{4} \left(\sum_{r=1}^{3} (x_{ir}) \right) \leq 3 \\ & x_{ir} \in \{0,1\} \quad \forall i, r \end{cases}$$

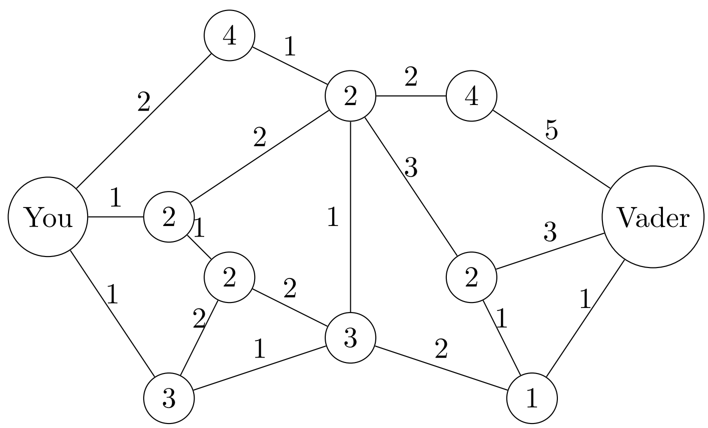
The following Sage script implements this integer program.

```
d = Matrix([(3, 2, 3),
            (2, 3, 1),
            (4, 2, 2),
            (1, 5, 1)])

nI = d.nrows()
nR = d.ncols()

milp = MixedIntegerLinearProgram(maximization=True)
x = milp.new_variable(binary=True)
obj = 0
for i in range(nI):
    obj += sum([d[i,r]*x[i+1,r+1] for r in range(nR)])
milp.set_objective(obj)

for i in range(nI):
    milp.add_constraint(sum([x[i+1,r+1] for r in range(nR)]) <= 1)
for r in range(nR):
    milp.add_constraint(sum([x[i+1,r+1] for i in range(nI)]) >= 1)

con = 0
for i in range(nI):
    con += sum([x[i+1,r+1] for r in range(nR)])
milp.add_constraint(con <= 3)

print('Objective Value: {}'.format(milp.solve()))
sol = milp.get_values(x)
for i, v in sol.items():
    print('x[%s] = %s' % (i, v))
```

The output of this script is shown below.

```
Objective Value: 12.0
x[(1, 2)] = 0.0
x[(3, 2)] = 0.0
x[(1, 3)] = 1.0
x[(3, 3)] = 0.0
x[(3, 1)] = 1.0
x[(2, 1)] = 0.0
x[(2, 3)] = 0.0
x[(4, 3)] = 0.0
x[(2, 2)] = 0.0
x[(4, 2)] = 1.0
x[(4, 1)] = 0.0
x[(1, 1)] = 0.0
```

3. Darth Vader forgot his lightsaber. Bring it to him as fast as you can so the Emperor can watch a lightsaber fight between Darth Vader and Luke Skywalker. You have only a small ship and hence you need refueling. Here is a map of the space. Every planet has a number that corresponds to the time needed for refueling and every connection has a travel time associated to it.



This is a shortest path problem with the addition of weights on the vertices. This problem can be solved with a slightly modified version of Djikstra's algorithm. Note that Dijkstras algorithm will work because all edge and vertex weights are positive.

In order to handle the vertex weights as well a couple of modifications are necessary. First the distance from the source to itself should be the weight of the source vertex instead of zero. In our case the weight of the source is zero anyways. Second when modifying the distance from the source to any other vertex the weight of the target vertex must also be included. For example if attempting to add edge (v, w), normally the following occurs

$$l(w) = \min\{l(w), l(v) + c(v, w)\}$$

where $c(v, w)$ is the cost of the (v, w) edge. With the addition of vertex weights the following update will occur.

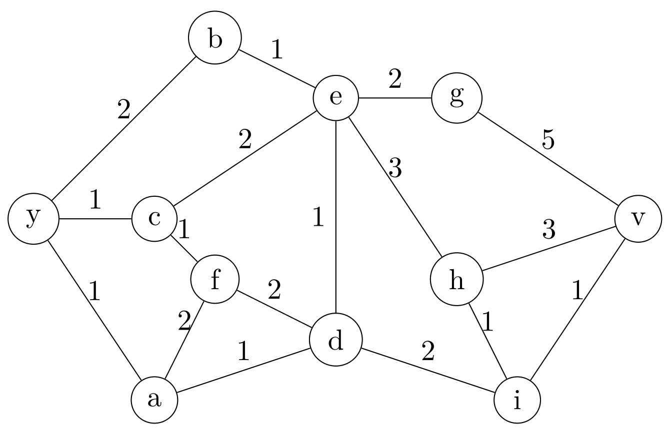
$$l(w) = \min\{l(w), l(v) + c(v, w) + c(w)\}$$

where $c(w)$ is the cost of the vertex w. Note that $l(v)$ will already have the cost of v accounted for. These two modifications will allows Dijkstra's Algorithm to solve this problem.

The following function runs Dijkstra's Algorithm with the forementioned modifications.

```
def dijkstras(graph, source):
    reached = set()
    vertexSet = set(graph.vertices())
    length = {v:Infinity for v in vertexSet}
    length[source] = graph.get_vertex(source)
    parent = {v:None for v in vertexSet}
    while reached != vertexSet:
        v = min(vertexSet - reached, key=lambda v:length[v])
        reached.add(v)
        for w in graph.neighbors(v):
            d = length[v] + graph.edge_label(v, w) + graph.get_vertex(w)
            if d < length[w]:
                length[w] = d
                parent[w] = v

    return (length, parent)
```

Now I will label the vertices of the graph as follows, while keeeping in mind the weights on the vertices



Now the following script will run the algorithm on this graph.

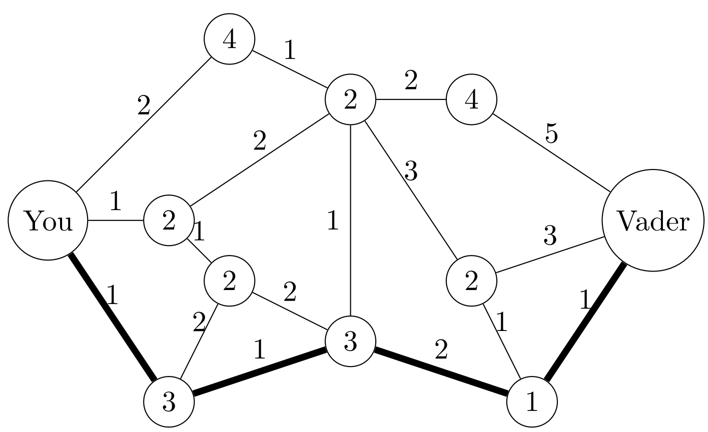
```
load('dijkstras.sage')
#                a  b  c  d  e  f  g  h  i  v  y
M = Matrix([(0, 0, 0, 1, 0, 2, 0, 0, 0, 0, 1), #a
            (0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 2), #b
            (0, 0, 0, 0, 2, 1, 0, 0, 0, 0, 1), #c
            (1, 0, 0, 0, 1, 2, 0, 0, 2, 0, 0), #d
            (0, 1, 2, 1, 0, 0, 2, 3, 0, 0, 0), #e
            (2, 0, 1, 2, 0, 0, 0, 0, 0, 0, 0), #f
            (0, 0, 0, 0, 2, 0, 0, 0, 0, 5, 0), #g
            (0, 0, 0, 0, 3, 0, 0, 0, 1, 3, 0), #h
            (0, 0, 0, 2, 0, 0, 0, 1, 0, 1, 0), #i
            (0, 0, 0, 0, 0, 0, 5, 3, 1, 0, 0), #v
            (1, 2, 1, 0, 0, 0, 0, 0, 0, 0, 0)]) #y

graph = Graph(M, weighted=True)
```

```
graph.relabel({0:'a', 1:'b', 2:'c', 3:'d', 4:'e', 5:'f', 6:'g', 7:'h',
    ↪ 8:'i', 9:'v', 10:'y'})
graph.set_vertices({'a':3, 'b':4, 'c':2, 'd':3, 'e':2, 'f':2, 'g':4, 'h'
    ↪ :2, 'i':1, 'v':0, 'y':0})
source = 'y'
target = 'v'

(length, parent) = dijkstras(graph, source)

path = [target]
while parent[path[-1]] != None:
    path.append(parent[path[-1]])

path.reverse()
print("Distance: %s" % length[target])
print("Shortest path: %s" % path)
```

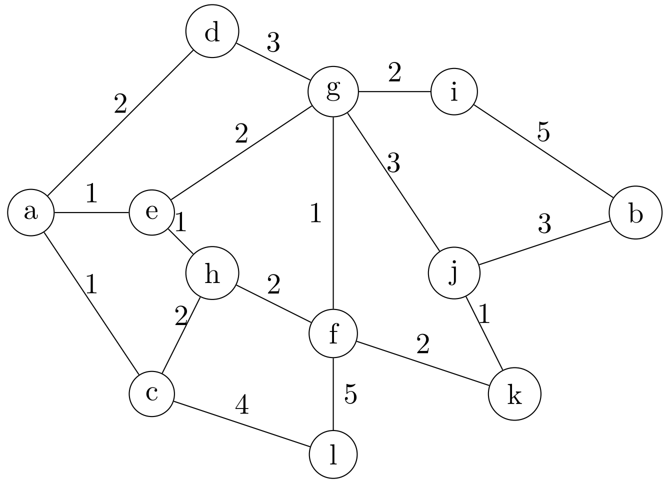
The output of this script is as follows.

```
Distance: 12
Shortest path: ['y', 'a', 'd', 'i', 'v']
```

So the shortest path from me to Vader is of length 12 and is marked in the graph below.



4. The Emperor, as any other good emperor, knows that it is best if his enemies are fighting among each other. Here is a map of Emperor's enemies and how much does it cost for each pair to start fighting each other. Find for the Emperor which pairs to trick into fighting each other such that every enemy is in exactly one fight. Provide also a certificate verifying the optimality of your solution.



This is a minimum matching problem in a weighted bipartite graph. To see that this graph is bipartite note that the vertices can be partitioned into sets $\{a, b, g, h, k, l\}$ and $\{c, d, e, f, i, j\}$, that are not internally connected. The minimum matching problem in a weighted bipartite graph can be solved with the following integer program.

$$(IP) \begin{cases} \text{maximize} & \sum_{e \in E} (c(e) x_e) \\ \text{subject to} & \sum_{e \in \delta(v)} (x_e) = 1 \quad \forall v \in V \\ & x_e \in \{0, 1\} \quad \forall e \in E \end{cases}$$

where x_e represent whether edge e is in the matching.

The following script implements this integer program for the given graph in Sage.

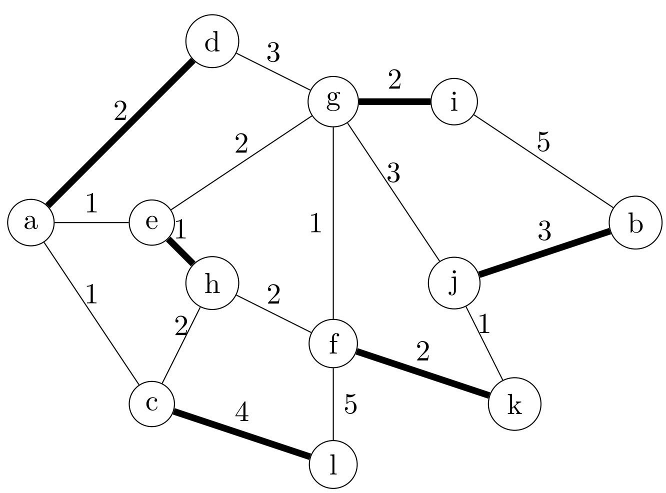
```
load('contractVertices.sage')
load('edmondsBlossom.sage')
#             a  b  c  d  e  f  g  h  i  j  k  l
M = Matrix([(0, 0, 1, 2, 1, 0, 0, 0, 0, 0, 0, 0), #a
            (0, 0, 0, 0, 0, 0, 0, 0, 5, 3, 0, 0), #b
            (1, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 4), #c
            (2, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0), #d
            (1, 0, 0, 0, 0, 0, 2, 1, 0, 0, 0, 0), #e
            (0, 0, 0, 0, 0, 0, 1, 2, 0, 0, 2, 5), #f
            (0, 0, 0, 3, 2, 1, 0, 0, 2, 3, 0, 0), #g
            (0, 0, 2, 0, 1, 2, 0, 0, 0, 0, 0, 0), #h
            (0, 5, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0), #i
            (0, 3, 0, 0, 0, 0, 3, 0, 0, 0, 1, 0), #j
            (0, 0, 0, 0, 0, 2, 0, 0, 0, 1, 0, 0), #k
            (0, 0, 4, 0, 0, 5, 0, 0, 0, 0, 0, 0)])

graph = Graph(M, weighted=True)
graph.relabel({0:'a',1:'b',2:'c',3:'d',4:'e',5:'f',6:'g',7:'h',8:'i',9:'j',10:'k',11:'l'})

milp = MixedIntegerLinearProgram(maximization=False)
x = milp.new_variable(binary=True)

milp.set_objective(sum([e[2]*x[e] for e in graph.edges()]))
```

```
for v in graph.vertices():
    milp.add_constraint(sum([x[e] for e in graph.edges_incident(v)]) ==
        ↪ 1)

print('Objective Value: {}'.format(milp.solve()))
sol = milp.get_values(x)
for i, v in sol.items():
    print('x[%s] = %s' % (i, v))
```

The output of the script is shown below.

```
Objective Value: 14.0
x[('j', 'k', 1)] = 0.0
x[('a', 'd', 2)] = 1.0
x[('g', 'j', 3)] = 0.0
x[('a', 'e', 1)] = 0.0
x[('d', 'g', 3)] = 0.0
x[('c', 'l', 4)] = 1.0
x[('e', 'g', 2)] = 0.0
x[('f', 'k', 2)] = 1.0
x[('b', 'i', 5)] = 0.0
x[('f', 'l', 5)] = 0.0
x[('c', 'h', 2)] = 0.0
x[('b', 'j', 3)] = 1.0
x[('f', 'h', 2)] = 0.0
x[('a', 'c', 1)] = 0.0
x[('g', 'i', 2)] = 1.0
x[('e', 'h', 1)] = 1.0
x[('f', 'g', 1)] = 0.0
```

The matching can be viewed in the graph as follows.



In order to check that this is in fact the optimal solution the dual linear program can be solved as well. If the provide the same optimal value, then we know that we have found an optimal solution.

The dual integer program is

$$(D)\begin{cases} \text{maximize} & \sum_{v\in V}(y_v) \\ \text{subject to} & y_u + y_v \leq c(u,v) \quad \forall (u,v)\in E \\ & y_v \in \mathbb{R} \quad \forall v \in V \end{cases}$$

The following script implements this linear program for the given graph.

```
milp = MixedIntegerLinearProgram(maximization=True)
y = milp.new_variable()
milp.set_objective(sum({y[v] for v in graph.vertices()}))
for e in graph.edges():
    milp.add_constraint(y[e[0]] + y[e[1]] <= e[2])

print('Objective Value: {}'.format(milp.solve()))
sol = milp.get_values(y)
for i, v in sol.items():
    print('y[%s] = %s' % (i, v))
```

The output of this script is shown below.

```
Objective Value: 14.0
y[a] = -0.0
y[c] = 1.0
y[b] = 3.0
y[e] = 1.0
y[d] = 2.0
y[g] = 0.0
y[f] = 1.0
y[i] = 2.0
y[h] = 0.0
y[k] = 1.0
y[j] = 0.0
y[l] = 3.0
```

As shown the optimal solution has value 14 just as the original program did. This shows that we have in fact found an optimal solution.