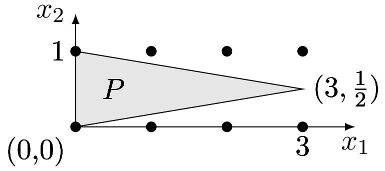
Caleb Logemann
MATH 566 Discrete Optimization
Homework 10

1. Solve the following problem using branch and bound. Draw the branching tree too.

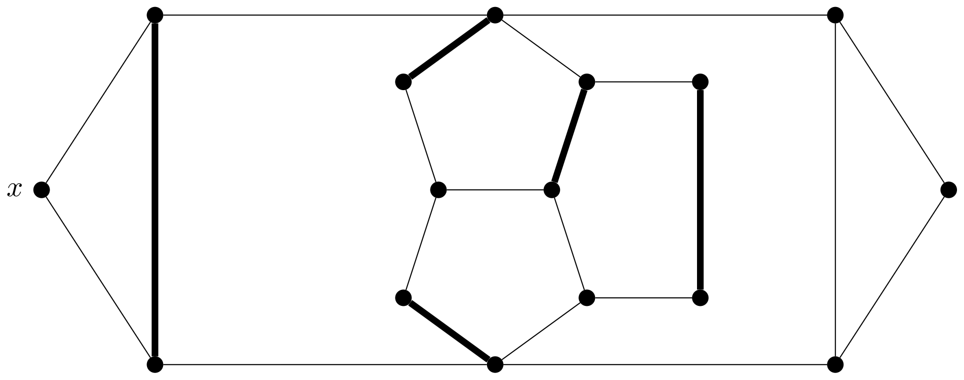
$$(P) = \begin{cases} \text{maximize} & -x_1 + 4x_2 \\ \text{subject to} & -10x_1 + 20x_2 \leq 22 \\ & 5x_1 + 10x_2 \leq 49 \\ & x_1 \leq 5 \\ & x_i \geq 0, x_i \in \mathbb{Z} \text{ for } i \in \{1, 2\} \end{cases}$$

You can use any linear programming solver for solving the relaxations.

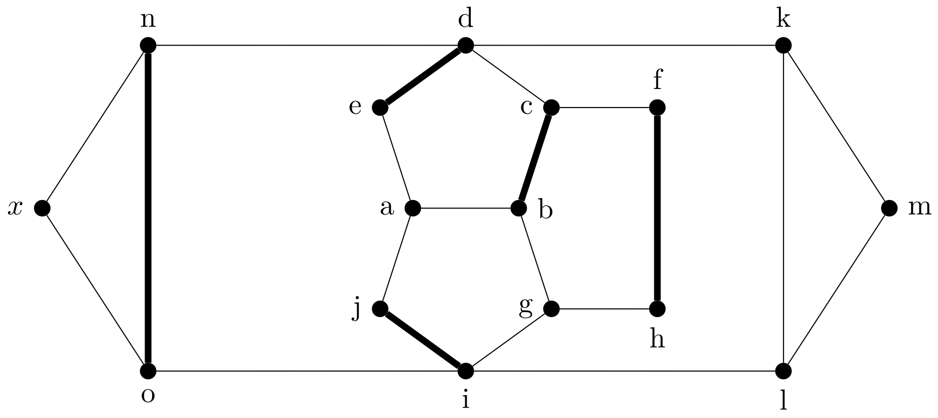
2. Let P be a convex hull of $(0,0), (0,1), (k, \frac{1}{2})$. Give an upper bound on Chvátal's rank of P. (Show it is at most $2k$, actually, it is exactly $2k$.)
Drawing of P for $k = 3$.



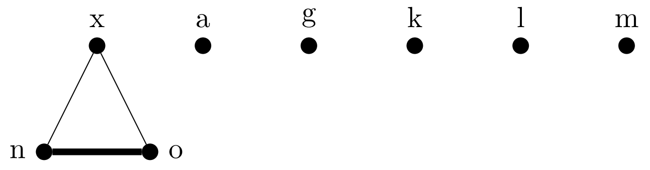
3. Run Edmond's Blossom algorithm on the following graph. Notice that somebody already found a partial matching. What is the largest possible matching? Try to start growing augmenting tree from x, use BFS algorithm for building the tree.



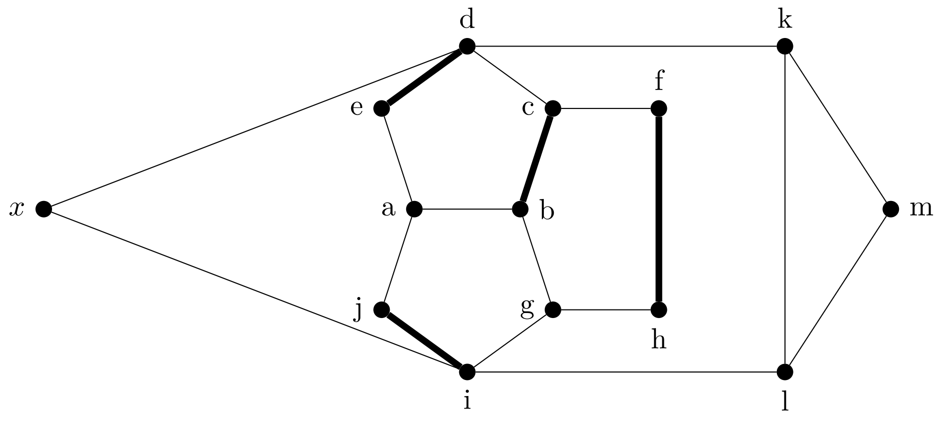
First I will label all of the vertices of the graph as follows.



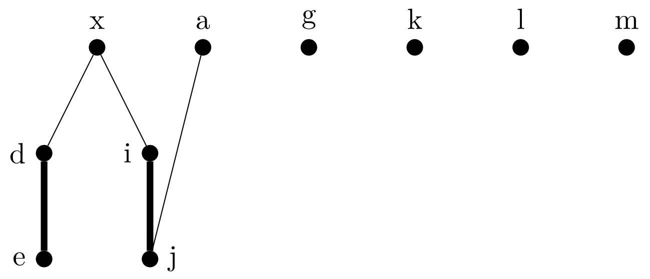
The first step in the algorithm is to grow forest from the exposed vertices. The following is the forest created by adding edges (x, n) and (x, o), which results in a blossom.



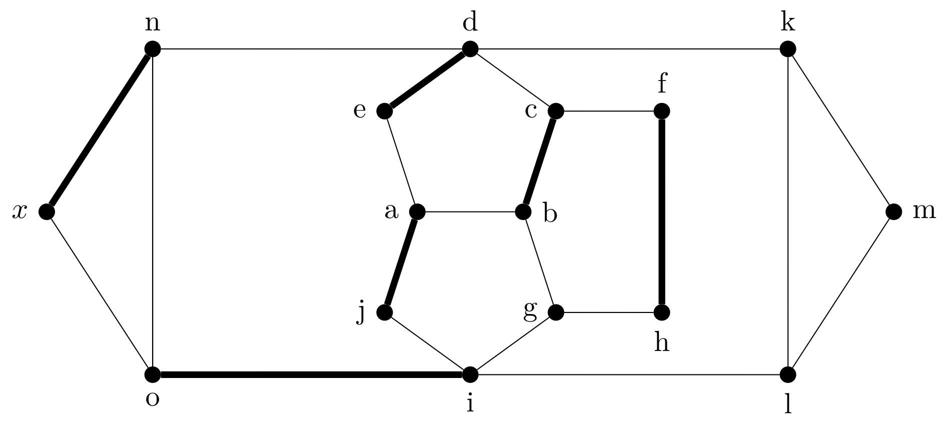
The blossom needs to be contracted in the forest and the graph. The contracted graph and forest are shown below.



This tree can now be continued to be expanded by adding (x, i) and (x, d) which causes (i, j) and (d, e) to be added respectively. Now adding the edge (j, a) creates an augmenting path. The alternating forest is now

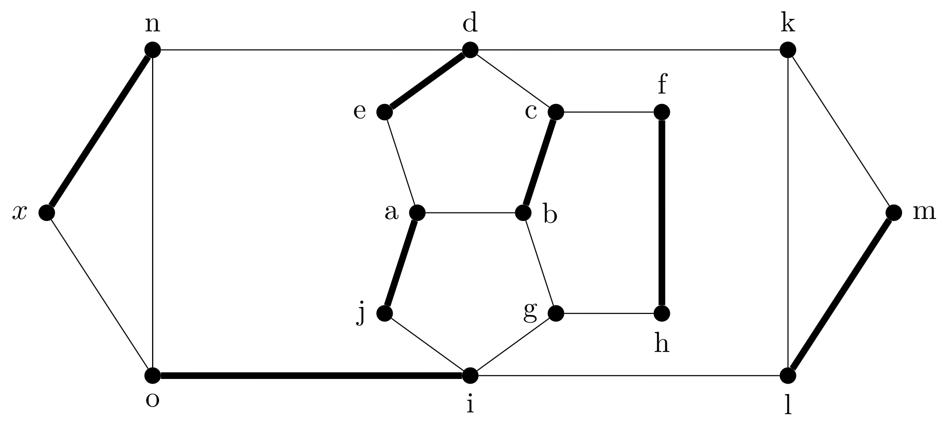


This augmenting path going around the blossom is $x \to n \to o \to i \to j \to a$. Doing the augmentation addes (x, n),(o, i), and (j, a) to the matching, while removing (n, o) and (i, j). The new matching is shown in the following graph.

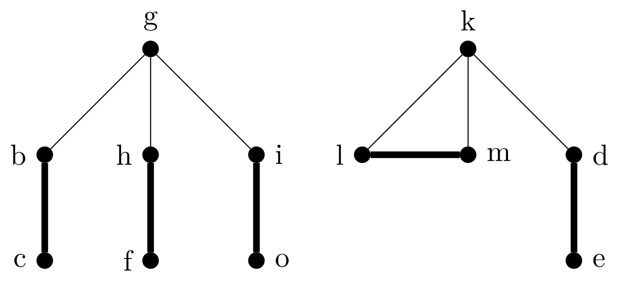


Again in order to find an augmeting path we start growing the alternating forest from the exposed vertices. Note that an augmenting path can be found directly by adding (l, m).

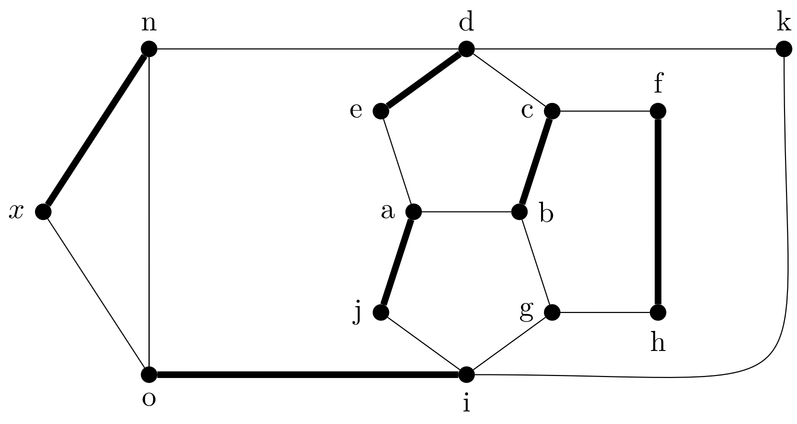
Augmenting on this path adds (l, m) to the matching, so the new matching is

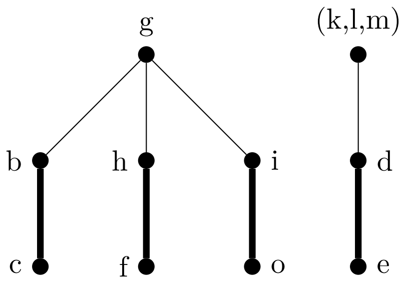


We now have only two exposed vertices, so the alternating forest needs to be grown to check if there is an augmenting path from g to k. Adding the edges incident to g and k results in the following.

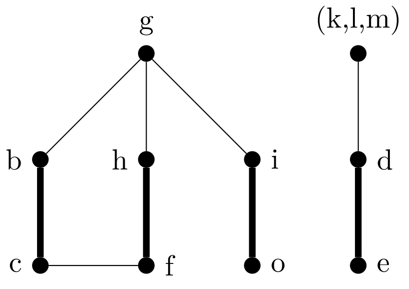


There is a blossom containing k, l, and m. Contracting this blossom in the forest and the graph results in

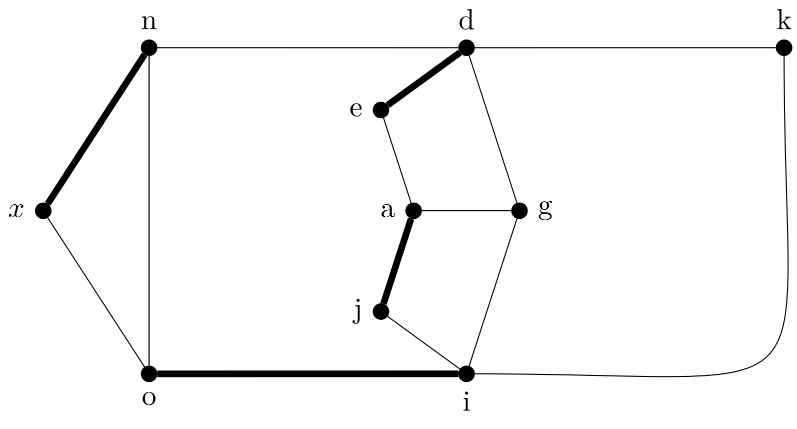


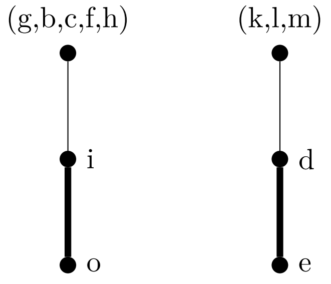


Now let us add (c, f), this creates a blossom of length 5.

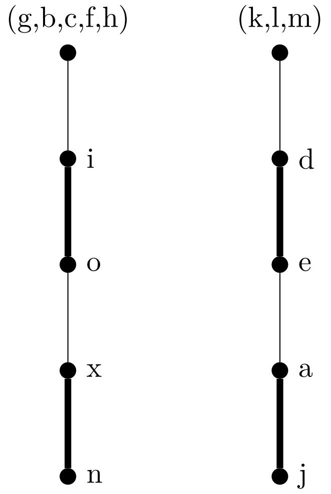


Contracting this blossom in the forest and graph results in

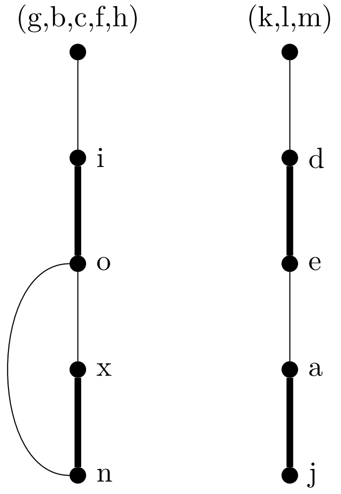


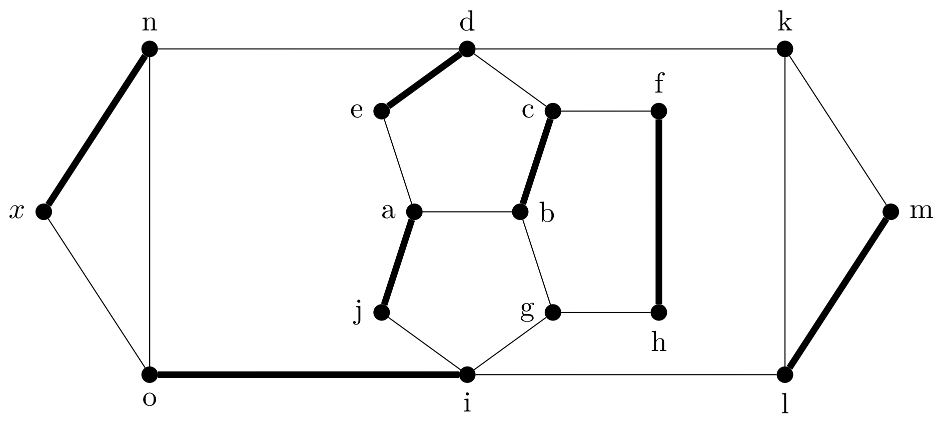


We can now continue growing the forest, consider adding (e, a) and (o, x).



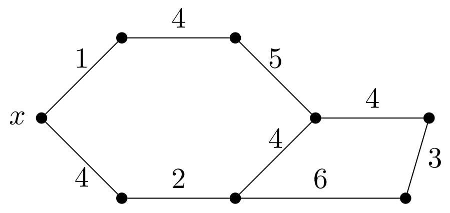
The edges (g, a), (k, i), (j, i) and (n, d) can't be added because they connect to vertices that are an odd distance from their root. The last edge that can be added is (n, o). This creates a blossom, but as it is the last edge that can be added and no augmenting path has been found, we can conclude that we have found the maximal matching. The final forest and maximal matching are





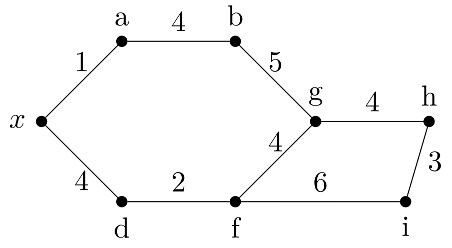
Therefore the largest possible matching contains 7 edges.

4. Find minimum-weight perfect matching in the following graph:



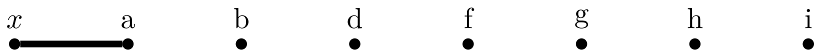
(a) By using algorithm from class that grows augmenting tree (and keep primal/dual solutions). Start growing x.

First I will relabel the vertices of $G = (V, E)$ as follows.

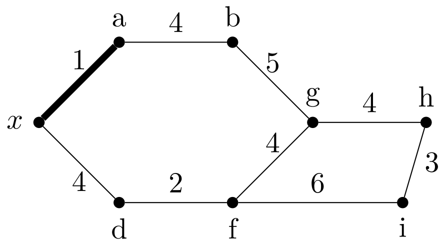


Initially we will start with initial solutions $y_v = 0$ for all $v \in V$ and $x_e = 0$ for all $e \in E$. The initial solution $\mathbf{y}$ is a feasible solution to the dual problem as $y_u + y_v <= c(e_{uv})$ for all edges. In this case $E_= = \{\}$ because there are no edges $e = (u, v)$ such that $y_u + y_v = c(e)$. Now we will construct the initial alternating forest using $E_=$. Since $E_=$ is empty the alternating forest will contain all vertices as single node trees.

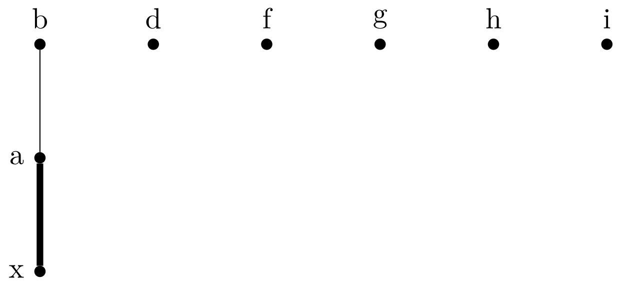
Since a perfect matching is not possible, we must modify $\mathbf{y}$ for one of the connected components of F, the alternating forest. Therefore we will change y_x. The smallest possible increase in y_x that maintains feasibility of the dual is one, so let $y_x = 1$. Now $E_= = \{(x, a)\}$. We can now start growing the alternating forest again with edges from $E_=$.



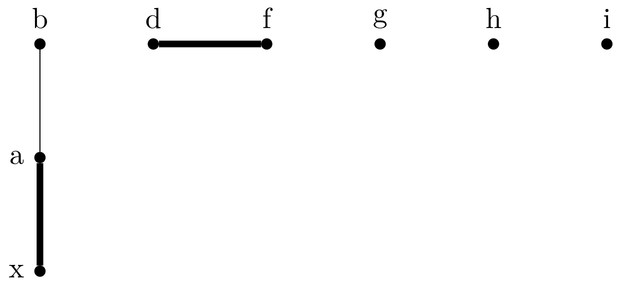
The only augmenting path is from x to a. The matching that is created is shown below with bold edges.



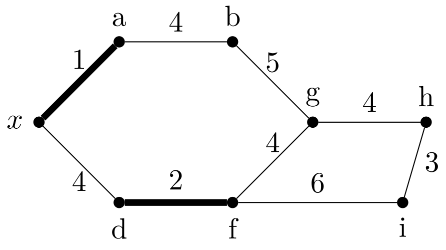
This does not creat a perfect matching so we must change a value of $\mathbf{y}$ to allow one of the connected components of F to be expanded. Consider the vertex b, y_b can be increased by 4 so that $y_b = 4$ and $E_= = \{(x,a),(a,b)\}$. Now the alternating forest can be grown again into a matching. If we first match x and a as before, then the final forest would look like



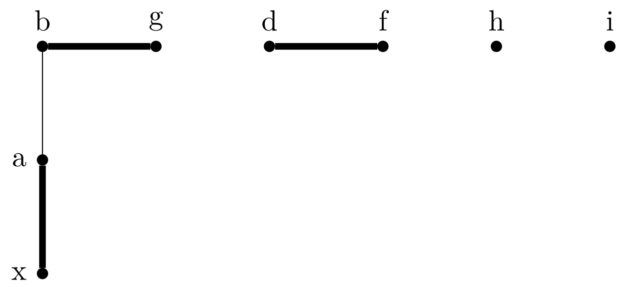
This does not find a perfect matching. In fact this is the same matching as before. Therefore $\mathbf{y}$ must be modified for one of the connected components of F. Consider modifying y_d. Let $y_d = 2$ so that (d,f) can be added to $E_=$. Now $E_= = \{(x,a),(a,b),(d,f)\}$. Building on the previous alternating forest we can find an alternating path from d to f



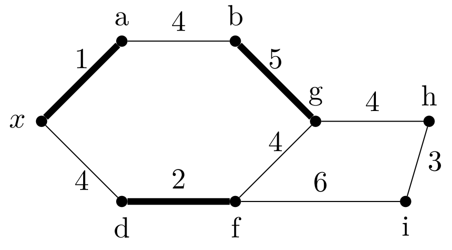
This forest creates the matching shown below.



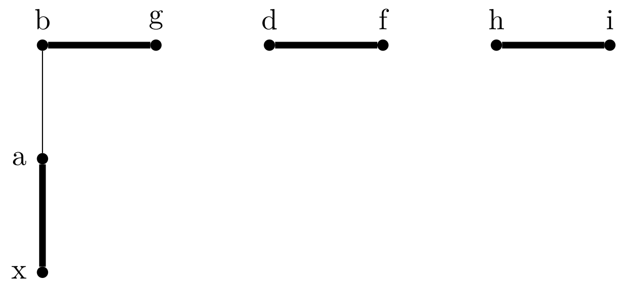
This matching isn't perfect, therefore $\mathbf{y}$ needs to be updated to allow for the expansion of F. The value of y_g can be set to 1, to allow (b,g) to be added to $E_=$. This allows for the alternating forest to be expanded as follows.



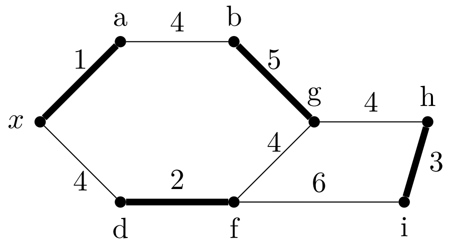
The matching from this alternating forest is shown below.



Again this is not a perfect matching, therefore y_h must be set to 3. This adds the edges (g, h) and (h, i) to $E_=$. The set $E_=$ is now $\{(x, a), (a, b), (d, f), (b, g), (g, h), (h, i)\}$. Growing an alternating forest from this set of edges allows us to find an alternating path from h to i. The final alternating forest is shown below, with no exposed vertices.



Since all edges are covered this is a perfect matching. Now since we have a feasible solution to both the primal and the dual problem, this is the optimal solution to the minimal matching problem. This matching is shown below.



Also to recap here are the values of $\mathbf{y}$.

$$
\begin{aligned}
y_x &= 1\\
y_a &= 0\\
y_b &= 4\\
y_d &= 2\\
y_f &= 0\\
y_g &= 1\\
y_h &= 3\\
y_i &= 0
\end{aligned}
$$

(b) Formulate the problem using Integer/Linear programming and solve it with your favorite solver. The integer program for solving the maximal perfect matching problem for a graph $G = (V, E)$. Uses a variable x_e for each edge, $e \in E$.

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

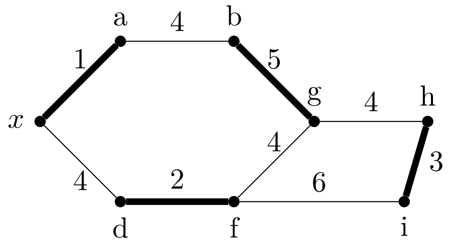
The following sage script solves this linear program for the given graph.

```
#d = {'x': ['a', 'd'], 'a': ['b'], 'b': ['g'], 'g': ['f', 'h'], 'h':
    ↪ ['i'], 'd': ['f']}
#                x, a, b, d, f, g, h, i
m = Matrix([(0, 1, 0, 4, 0, 0, 0, 0),
            (1, 0, 4, 0, 0, 0, 0, 0),
            (0, 4, 0, 0, 0, 5, 0, 0),
            (4, 0, 0, 0, 2, 0, 0, 0),
            (0, 0, 0, 2, 0, 4, 0, 6),
            (0, 0, 5, 0, 4, 0, 4, 0),
            (0, 0, 0, 0, 0, 4, 0, 3),
            (0, 0, 0, 0, 6, 0, 3, 0)])

graph = Graph(m, weighted=True)
graph.relabel({0:'x', 1:'a', 2:'b', 3:'d', 4:'f', 5:'g', 6:'h', 7:'i
    ↪ '})
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 this script is shown below.

```
Objective Value: 11.0
x[('f', 'i', 6)] = 0.0
x[('d', 'x', 4)] = 0.0
x[('a', 'b', 4)] = 0.0
x[('b', 'g', 5)] = 1.0
x[('h', 'i', 3)] = 1.0
x[('f', 'g', 4)] = 0.0
x[('a', 'x', 1)] = 1.0
x[('g', 'h', 4)] = 0.0
x[('d', 'f', 2)] = 1.0
```

This matching is the same as the matching found in part (a) and can be shown on the graph as shown below.



5. Slither is a two-person game played on a graph $G = (V, E)$. The players, called First and Second, play alternatively, with First playing first. At each step the player whose turn it is chooses a previously unchosen edge. The only rule is that at every step the set of chosen edges forms a path. The loser is the first player unable to make a legal move at his or her turn. Prove that if G has a perfect matching, then First can force a win.

   Proof. Suppose $G = (V, E)$ has a perfect matching, $M \subset E$. Let First select an edge $e \in M$ for their first move. Now on Second's turn, they must select an edge adjacent to e. Since $e \in M$ and edge adjacent to e will not be in M, otherwise a vertex would be incident to two edges in the matching. After Second selects an edge there will be three vertices in the path, two vertices incident to First's first selection and one additional vertex incident to Second's selection. First must now select the edge in M that is incident to this third vertex. This edge exists because M is a perfect matching so it covers all vertices. Also this edge could not have been previously selected as it was just made adjacent to the path in the last turn. First should continue to select the unique edge in M that is a possible selection. When First selects the last remaining edge in M, Second will be unable to make a legal move. At this point all edges in the perfect matching have been selected, so that all vertices must be a part of the path. Therefore edge that is selected by Second will join two vertices already in the path and create a cycle. This implies that there is no legal move for Second. □

6. Implement algorithm for finding maximum matching in bipartite graphs. Test it on the 3D-cube.

   See problem 7

7. Implement algorithm for finding maximum matching in any graph.
   Test it on the 3D-cube and the graph from question 3.
   (Doing this will also solve the previous question - 2 for 1.)

   The following sage script implements Edmonds Blossom algorithm which finds maximum matchings for any graph.

```
def edmondsBlossom(graph):
    matching = set()
    path = find_augmenting_path(graph, matching)
    while path != None:
        matching = augment_on_path(graph, matching, path)
        path = find_augmenting_path(graph, matching)
    return matching

def find_augmenting_path(graph, matching):
    forest = Graph()
    covered_vertices = set([e[0] for e in matching] + [e[1] for e in
       ↪ matching])
    # exposed vertices
    nodes_to_check = list(set(graph.vertices()) - covered_vertices)
    forest.add_vertices(nodes_to_check)
    node_to_root = {v:v for v in nodes_to_check}
    marked_edges = matching.copy()
    for v in nodes_to_check:
        unmarked_incident_edges = set(graph.edges_incident(v)) -
           ↪ marked_edges
        for e in unmarked_incident_edges:
            if e[0] == v:
                w = e[1]
            else:
                w = e[0]

            if w not in forest.vertices():
                # grow forest
                matched_edge_w = [edge for edge in matching if w in edge
                   ↪ ][0]
                if matched_edge_w[0] == w:
                    x = matched_edge_w[1]
                else:
                    x = matched_edge_w[0]

                forest.add_vertices([w, x])
                forest.add_edges([e, matched_edge_w])
                nodes_to_check.append(x)
                node_to_root[w] = node_to_root[v]
                node_to_root[x] = node_to_root[v]

            else:
                if forest.distance(w,node_to_root[w]) % 2 == 0:
                    if node_to_root[v] != node_to_root[w]:
                        # found augmenting path
                        path1 = forest.shortest_path(node_to_root[v], v)
                        path2 = forest.shortest_path(w, node_to_root[w])
                        path = path1 + path2
                    else:
                        # blossom
                        path = contract_blossom(graph, matching, forest,
                            ↪ v, w)
                    return path
            marked_edges.add(e)
    return None

def augment_on_path(graph, matching, path):
    edges = {(path[i],path[i+1],graph.edge_label(path[i],path[i+1])) for
        ↪ i in range(len(path)-1)}

    for e in edges:
        if e in matching:
            matching.remove(e)
        elif (e[1], e[0], e[2]) in matching:
            matching.remove((e[1], e[0], e[2]))
        else:
            matching.add(e)
    return matching

def contract_blossom(graph, matching, forest, v, w):
    blossom = forest.shortest_path(v,w)
    matched_edges_in_blossom = set()
    for e in matching:
        if e[0] in blossom and e[1] in blossom:
            matched_edges_in_blossom.add(e)
    base_vertex = list(set(blossom) - set([e[0] for e in
        ↪ matched_edges_in_blossom] + [e[1] for e in
        ↪ matched_edges_in_blossom]))[0]
    #print base_vertex
    contracted_graph = graph.copy()
    for x in blossom:
        if x != base_vertex:
            contractVertices(contracted_graph, base_vertex, x)
    contracted_matching = set()
    for e in matching:
        if e[0] in blossom and e[1] not in blossom:
            contracted_matching.add((base_vertex, e[1], e[2]))
        elif e[1] in blossom and e[0] not in blossom:
            contracted_matching.add((e[0], base_vertex, e[2]))
        elif e[0] not in blossom and e[1] not in blossom:
            contracted_matching.add(e)
    #contracted_graph.plot().show()
    path = find_augmenting_path(contracted_graph, contracted_matching)
    if path != None and base_vertex in path:
        i = path.index(base_vertex)
        path1 = path[:i]
        path2 = path[i+1:]
        # need alternating path from base_vertex to path2[0]
        for x in blossom:
            if path2[0] in graph.neighbors(x):
                starting_vertex = x

        blossom_path = [starting_vertex]
        while blossom_path[-1] != base_vertex:
            matched_edge = [edge for edge in matching if blossom_path
                ↪ [-1] in edge][0]
            # add vertex incident with matched edge
            if matched_edge[0] == blossom_path[-1]:
                blossom_path.append(matched_edge[1])
            else:
                blossom_path.append(matched_edge[0])

            # add next vertex in blossom
            j = blossom.index(blossom_path[-1])
            if j == 0:
                jminus = len(blossom) - 1
            else:
                jminus = j - 1

            if j== len(blossom)-1:
                jplus = 0
            else:
                jplus = j + 1

            if blossom[jminus] == blossom_path[-2]:
                blossom_path.append(blossom[jplus])
            else:
                blossom_path.append(blossom[jminus])
        blossom_path.reverse()
        path = path1 + blossom_path + path2
    return path
```

The following function is used in these functions to do contraction in graphs.

```
def contractVertices(graph, v, w):
    if graph.get_vertex(v) != None:
        graph.set_vertex(v, graph.get_vertex(v).union(graph.get_vertex(w
            ↪ )))
    for x in graph.neighbors(w):
        if x != v:
            weight = graph.edge_label(w, x)
            if x not in graph.neighbors(v):
                graph.add_edge(v, x, weight)
            elif weight != None and graph.edge_label(x, v) != None:
                graph.set_edge_label(x, v, graph.edge_label(x, v) +
                    ↪ weight)
    graph.delete_vertex(w)

    return None
```

The following script now tests this function on the 3D cube and the graph from problem 3.

```
load('contractVertices.sage')
load('edmondsBlossom.sage')

# 3D cube
graph = graphs.CubeGraph(3)
matching = edmondsBlossom(graph)
print 'Edges␣in␣matching␣for␣3D␣cube'
for e in matching:
    print e

# graph from problem 3
d = {'a':['b', 'e', 'j'], 'b':['c', 'g'], 'c':['d', 'f'], 'd':['e','k','
   ↪ n'],
        'f':['h'], 'g':['h', 'i'], 'i':['j','l','o'], 'k':['l', 'm'], 'l
           ↪ ':['m'],
        'n':['o', 'x'], 'o':['x']}
graph = Graph(d)
matching = edmondsBlossom(graph)

print 'Edges␣in␣matching␣for␣graph␣in␣problem␣3'
for e in matching:
    print e
```

The output of this script is shown below.

```
Edges in matching for 3D cube
('010', '110', None)
('100', '000', None)
('011', '001', None)
('111', '101', None)
Edges in matching for graph in problem 3
('a', 'b', None)
('i', 'l', None)
('d', 'e', None)
('f', 'c', None)
('x', 'o', None)
('k', 'm', None)
('g', 'h', None)
```

As can be seen the function finds a perfect matching for the 3D cube with is bipartite, and finds a matching with 7 edges for problem 3 which we have previously shown to be the maximum size of a matching.