Assignment 5
Algorithm Design and Analysis

bitjoy.net

December 16, 2015

I choose problem 1,2,5,7,8,9.

1 Problem Reduction

1.1 Algorithm description

We first construct a bipartite graph by linking the girl and the boys who the girl likes. For example, girl 1 likes boy 5 and boy 6, then edge(1,5) and edge(1,6) are added. After that, add node s and t and connect s with girls and t with boys. Set each edge's capacity as 1, then we get a extended network. Figure 1 shows an example.

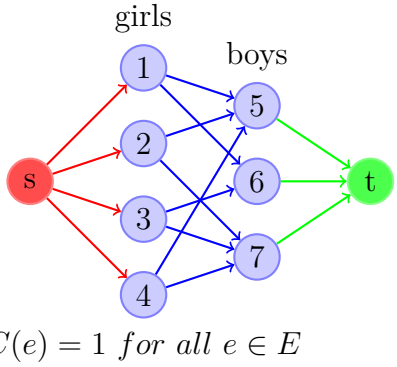

Figure 1: Extended network for maximal matching.

As a result, the maximum flow of the extended network is the maximum matching, we can run Ford-Fulkerson Algorithm to find the answer.

To summarize, we have the following algorithm, G is the original bipartite graph.

MAX-MATCHING(G)
1 adding node s and t to G
2 connecting s with girls and t with boys
3 setting the capacity of edges equals to 1 for all edges in G
4 running Ford-Fulkerson Algorithm to find the maximum flow in G
5 **return** the maximum flow as the maximum matching

1.2 Correctness of the algorithm

By setting the capacity of each edge as 1, we can assure that each girl appears at most in one pair, so does each boy. So, the maximum flow of the extended network

equals to the number of edges between girls and boys, which is the maximum matching between them.

1.3 Complexity of the algorithm

As we know, the time complexity of Ford-Fulkerson Algorithm is $O(E|f^*|)$ where E is the number of edges in G and $|f^*|$ is the maximum flow.

In algorithm MAX-MATCHING, we just add two nodes, say s and t, and connect some nodes to s, others to t, it takes $O(V)$ where V is the number of nodes in G. So, the total time complexity is $O(E|f^*| + V)$.

2 Problem Reduction

2.1 Algorithm description

Similar to Problem 1, we construct a extended network like Figure 2:

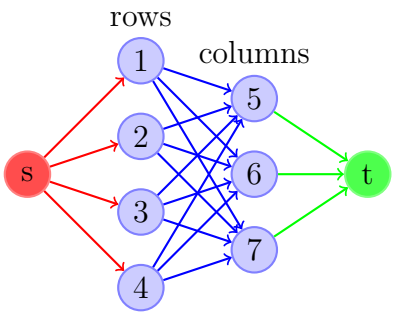

Figure 2: Extended network for matrix construction. $C(e) = 1$ for all edges between rows and columns, $C(< s, r >) = row\_sum[r]$, $C(< c, t >) = column\_sum[c]$.

As matrix can only be filled with 0 or 1, the capacity of edges between rows and columns equals to 1, the capacity out from s equals to each row sum and the capacity into t equals to each column sum. Given the 4*3 matrix, we have Figure 2.

We can get the maximum flow by running Ford-Fulkerson algorithm, if the maximum flow equals to the sum of all row sums(or all column sums), the final flow graph is the matrix we want.

To summarize, we have the following algorithm, R and C for the number of rows and columns, R_SUM and C_SUM for the list of sum of rows and columns.

MATRIX-CONSTRUCTION(R, C, R_SUM, C_SUM)

1 constructing a graph G with $R + C + 2$ nodes, node 0 for s, node $R + C + 1$ for t,
  node $1 \sim R$ for rows and node $R + 1 \sim R + C$ for columns
2 connecting s with rows, t with columns and all rows with all columns
3 setting $C(e) = 1$ for all edges between rows and columns,
  $C(< s, r >) = R\_SUM[r]$, $C(< c, t >) = C\_SUM[c]$
4 running Ford-Fulkerson Algorithm to find the maximum flow in G
5 **if** max-flow == sum of R_SUM
6 **return** the final flow graph
7 **else return** No matrix found!

2.2 Correctness of the algorithm

By setting the capacity of edges between rows and columns as 1, we can assure that only 0 or 1 will be filled in the matrix. So, the maximum flow of the extended network equals to the sum of all elements in the matrix. If the max flow equals to the sum of all row sums required, then the matrix is found, otherwise, no matrix meets the requirements.

2.3 Complexity of the algorithm

Similar to problem 1, the time complexity of MATRIX-CONSTRUCTION algorithm is $O(E|f^*| + V)$.

5 Dogs and kennels

5.1 Algorithm description

Suppose one dog stands at (x_1, y_1) and one kennel is placed at (x_2, y_2), then their distance is $(|x_1 - x_2|, |y_1 - y_2|)$. As 1 fee per step, once we get all distance between n dogs and n kennels, we get the cost between them.

So the problem is transferred to minimum cost flow. We construct a extended network like Figure 3:

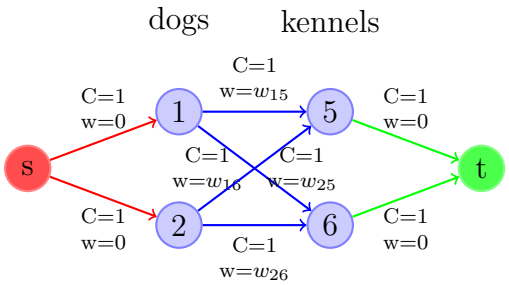

Figure 3: Extended network for dogs and kennels. $C(e) = 1$ for all edges, cost between dogs and kennels equals to their distance.

Set capacity as 1 for all edges, so that one dog can only enter one kennel and one kennel can only accommodate one dog. The objective of the network is to transfer $v_0 = n$ dogs to n kennels from s to t with minimal cost, which we can run minimum cost flow algorithm such as Klein's *Cycle canceling* algorithm to obtain it.

To summarize, we have the following algorithm, D and K are dogs' and kennels' position respectively, s and t are source and sink respectively.

```
SEND-DOGS-TO-KENNELS(D, K, s, t)
 1  for d ∈ D
 2        (s,d).C = 1
 3        (s,d).w = 0
 4  for k ∈ K
 5        (k,t).C = 1
 6        (k,t).w = 0
 7  for d ∈ D
 8        for k ∈ K
 9              (d,k).C = 1
10              (d,k).w = dist(d,k)
11  running minimum cost flow algorithm
        to find a circulation f with flow value v_0 = n and the cost is minimized
12  return the minimum cost
```

5.2 Correctness of the algorithm

By setting the capacity of all edges as 1, we can assure that one dog only enters one kennel and one kennel only accommodates one dog. By setting the objective flow as $v_0 = n$, we can assure all n dogs find their kennels. Once minimize the cost, we get the minimum money to send these n little dogs into those n different kennels.

Note since s and t are virtual nodes, their edges' weights equal to 0.

5.3 Complexity of the algorithm

As we know, the complexity of *Cycle canceling* algorithm is $O(E^2VCW)$ where E, V are the number of edges and vertexes respectively, C, W are the maximal capacity and cost respectively.

In SEND-DOGS-TO-KENNELS algorithm, $E = n^2 + 2n$, $V = 2n + 2$ and $C = 1$, so *Cycle canceling* costs $O(Wn^5)$. What's more, we have four extra for loops, which cost $O(n^2)$, so the total time complexity is $O(Wn^5 + n^2) = O(Wn^5)$ where W is the maximal distance between dogs and kennels.

7 Maximum flow

The dual of primal problem is:

$$\begin{array}{lrcll} \min & y & = & \sum u_e d_e & \\ s.t. & \sum_{e \in P} d_e & \geq & 1 & (\forall P) \\ & d_e & \geq & 0 & (\forall e) \end{array}$$

where y is the value that we optimize, u_e is the capacity of edge e and d_e's are variables.

This is LP formulation for minimum $s - t$ cut. Let the cut-set be $C = \{(u, v) \in E | u \in S, v \in T\}$, if $e \in C$ then $d_e = 1$, otherwise 0.

The first constraint says that there must be at least one edge in cut-set C from one path P, which is obvious. The second constraint says that each edge is nonnegative.

By strong duality, the minimum capacity over all $s-t$ cuts is equal to the maximum value of an $s - t$ flow[1].

[1] https://en.wikipedia.org/wiki/Max-flow_min-cut_theorem

8 Ford-Fulkerson algorithm

I implemented the Ford-Fulkerson algorithm in Python 3.

```
# -*- coding: utf-8 -*-
"""
Created on Mon Dec 14 21:21:27 2015

@author: czl
"""

# use numpy to handle data structure
import numpy as np
from collections import deque

# BFS finds a path and its bottleneck
def BFS(G, s, t):
    dq = deque([[[s], np.inf]])
    while len(dq) > 0:
        path, cf = dq.popleft()
        tmp = path[-1]
        if tmp == t:
            return True, path, cf
        else:
            for j in range(G.shape[1]):
                if j not in path and G[tmp][j] > 0:
                    dq.append([path + [j], min(cf, G[tmp][j])])
    return False, [], np.inf

# find max flow in directed graph G
def Ford_Fulkerson(G, s, t):
    flag, path, cf = BFS(G, s, t) # G is residual network Gf
    while flag:
        #print(G)
        #print(path)
        for i in range(len(path) - 1):
            u = path[i]
            v = path[i + 1]
            G[u][v] -= cf
            G[v][u] += cf
        flag, path, cf = BFS(G, s, t)
    max_flow = 0
    for j in range(G.shape[1]):
        max_flow += G[j][s]
    return max_flow

if __name__ == "__main__":
    f = open('./problem1.data')
    for i in range(3):
        f.readline()
    while True:
        line = f.readline()
        if not line:
            break
        girls, boys = line.split(' ')
        girls = int(girls)
        boys = int(boys)
        G = np.zeros([girls + boys + 2] * 2, dtype=np.int)
        s = 0
        t = girls + boys + 1
        for girl in range(girls):
```

```
            G[s][girl + 1] = 1
            line = f.readline().rstrip('\n')
            b = line.split(' ')
            b = list(map(int, b))
            for boy in range(b[0]):
                G[girl + 1][girls + b[boy + 1]] = 1
        for boy in range(boys):
            G[girls + boy + 1][t] = 1
        print(Ford_Fulkerson(G, s, t))
    f.close()
```

If you want to list the intermediate steps, just uncomment line 30 and line 31.The answers to problem 1 gotten by my implementation show below:

2
9
11
16
18
116

9 Push-relabel

I implemented the Push-relabel algorithm in Python 3.

```
# -*- coding: utf-8 -*-
"""
Created on Tue Dec 15 14:51:33 2015

@author: czl
"""

# use numpy to handle data structure
import numpy as np

def push(Gf, height, excess_flow, u):
    if excess_flow[u] <= 0:
        return False
    for v in range(len(Gf)):
        if v != u and Gf[u][v] > 0 and height[u] == height[v] + 1:
            df = min(excess_flow[u], Gf[u][v])
            Gf[u][v] -= df
            Gf[v][u] += df
            excess_flow[u] -= df
            excess_flow[v] += df
            return True
    return False

def relabel(Gf, height, excess_flow, u):
    if excess_flow[u] <= 0:
        return False
    min_h = np.inf
    for v in range(len(Gf)):
        if v != u and Gf[u][v] > 0:
            if height[u] > height[v]:
                return False
            else:
                min_h = min(min_h, height[v])
    height[u] = min_h + 1
    return True
```

```
36
37
38 def initialize_preflow(G, s):
39     n = len(G)
40     height = [0] * n
41     excess_flow = [0] * n
42     height[s] = n
43     for v in range(n):
44         if v != s and G[s][v] != 0:
45             excess_flow[v] = G[s][v]
46             excess_flow[s] -= G[s][v]
47             G[v][s] = G[s][v]
48             G[s][v] = 0
49     return G, height, excess_flow
50
51
52 def generic_push_relabel(G, s, t):
53     G_original = np.copy(G)
54     n = len(G)
55     Gf, height, excess_flow = initialize_preflow(G, s)
56     while True:
57         push_or_relabel = False
58         for u in range(n):
59             if u != s and u != t:
60                 if push(Gf, height, excess_flow, u):
61                     push_or_relabel = True
62                     break
63                 if relabel(Gf, height, excess_flow, u):
64                     push_or_relabel = True
65                     break
66         if not push_or_relabel:
67             break
68 #         else:
69 #             print(Gf)
70     max_flow = 0
71     for j in range(n):
72         max_flow += Gf[j][s]
73     G_flow = np.zeros(G.shape, dtype=np.int)
74     for row in range(G.shape[0]):
75         for col in range(G.shape[1]):
76             if G_original[row][col] != 0:
77                 G_flow[row][col] = G_original[row][col] - Gf[row][col]
78     return max_flow, G_flow
79
80
81 if __name__ == "__main__":
82     f = open('./problem2.data')
83     ans = open('./problem2.answer', 'ab')
84     for i in range(3):
85         f.readline()
86     while True:
87         line = f.readline()
88         if not line:
89             break
90         R, C = line.split(' ')
91         R = int(R)
92         C = int(C)
93         G = np.zeros([R + C + 2] * 2, dtype=np.int)
94         line = f.readline().rstrip(' \n')
95         R_SUM = list(map(int, line.split(' ')))
96         line = f.readline().rstrip(' \n')
```

```
97          C_SUM = list(map(int, line.split(' ')))
98          s = 0
99          t = R + C + 1
100         for row in range(R):
101             G[s][row + 1] = R_SUM[row]
102         for col in range(C):
103             G[R + col + 1][t] = C_SUM[col]
104         for row in range(1, R + 1):
105             for col in range(R + 1, R + C + 1):
106                 G[row][col] = 1
107         max_flow, G_flow = generic_push_relabel(G, s, t)
108         matrix = G_flow[1 : R + 1, R + 1 : R + C + 1]
109         #print(matrix)
110         np.savetxt(ans, matrix, delimiter=',', fmt='%d')
111         ans.write(bytes('----------\n', 'UTF-8'))
112         if max_flow == sum(R_SUM):
113             print('Answer is right!')
114         else:
115             print('Answer is wrong!')
116     f.close()
117     ans.close()
```

If you want to list the intermediate steps, just uncomment line 68 and line 69.The first answer to problem 2 gotten by my implementation shows below:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

For more details, please see file *problem2.answer*. According to the pseudo code of problem 2, if the maximum flow of the extended network equals to the sum of all row sums, the answer is right. I have checked my answers, all of them are right.