Algorithm Fifth Homework

Zhang Yuan

Student ID: 2015E8013261189

1 Problem 1: Problem Reduction

1.1 Basic idea and pseudo-code

We can construct a graph, link source and girls, set capacity=1, means a girl, link girls and boys they like, capacity=1, boys and sink, capacity=1, link boys and sink, capacity=1, means boy should be chosen at most one time. Find the maximum flow between source and sink.
Pseudo-Code:

```
Construct a graph, set nodes
Link the source and all girls, set edges' capacity=1
Link girls and boys they like, set edges' capacity=1
Link boys and the sink, set edges' capacity-1
Find the max flow between source and sink
```

1.2 Prove correctness

We can reduce the problem to the disjoint paths problem, we can not distribute the same boy for two girls, so it is same to the disjoint paths problem, then we can prove the correctness.

1.3 Complexity

Set m girls and n boys, so edge is mn at most, disjoint paths problem's complexity would be O(mn), so the complexity would be $O(m^2n)$.

2 Problem 2: Problem Reduction

2.1 Basic idea and pseudo-code

Construct a graph, if matrix has m columns and n rows, set mn nodes represents the value in ijth position in matrix, name these nodes as node set A, set m+n nodes represents the sum of every row and column, name these nodes as node set B, link node set B's nodes and source, set edges' capacity equal to the sum, link node A's nodes and sink, set edges' capacity equal to 1, means the value is 1 at most. Link node in set B and all nodes in corresponding column/row of set A, set edges' capacity equal to 1. Find the maximum flow between source and sink.
Pseudo-code:

```
Construct a graph, set source, sink, node set A and node set B
Link source and node set B, set edges' capacity equal to the sum
Link node in set B and all nodes in corresponding column/row of set A, set
Link sink and node set A, set edges' capacity equal to 1
Find the maximum flow between source and sink
If the maximum flow value is the sum of the sum of every rows and columns,
```

2.2 Prove correctness

This problem can be reduced to circulation problem with multiple sources and multiple sinks, the node represents the sum of every rows and columns equals to multiple sources in above problem. So if the maximum flow is the sum of the sum of every rows and columns, then we can find the suitable matrix. And the edge's flow between node of node set A and sink, is the matrix's ijth value.

2.3 Complexity

Set matrix contains m rows and n columns, nodes: O(mn+m+n), edges: O(mn), because the graph is bigraph, the complexity would be $O(m^2n^2)$.

3 Problem 3: Unique Cut

3.1 Basic idea and pseudo-code

Find max flow in Graph G, obtain a min-cut, and obtain all edges in the cut. We can iterate over each edge in the cut, add it's capacity, and rerun max flow, check whether the min-cut equals to the min-cut in last time.
Pseudo-Code:

```
Find the min cut in graph G
flag=False
FOR edge in min-cut:
increase the capacity, recalculate the min-cut
    IF min-cut=min-cut last time
     flag=True
        min-cut is not unique
    set edge back to the capacity before
IF flag==False:
min-cut is unique
```

3.2 Prove correctness

If this min-cut is not unique, then there exists some other minimum cut which has different cut-edge. If so, we can iterate over each edge in cut-edge, add it's capacity, recalculate the max flow algorithm, check if it doesn't increased, if so, the min cut is not unique, otherwise, it is unique.

3.3 Complexity

The max flow algorithm's complexity is $O(m^2n)$, the edge in min-cut is O(n), so the complexity is $O(m^2n^2)$.

4 Problem 5: Dogs and kennels

4.1 Basic idea and pseudo-code

Construct a graph, suppose there are n dogs, so for each dog, we set a node, for each kennels, we also set a node, link each dog and all kennels, set edges' capacity as the cost from dog's start point to the kennel. Then link all dogs node and the source, set edges' capacity equals to 1, and link all kennels and the sink, set edges' capacity equals to 1, means each kennel only contains one dog, and calculate the minimum cost flow when the flow is n.

```
Construct a graph, set n dogs node, n kennels node, link each dog and all k
Link source and dogs node, the edges' capacity equals to 1
Link sink and kennels node, the edges' capacity equals to 1
Calculate the minimum cost flow when the flow is n
```

4.2 Prove correctness

We can reduce the problem into a minimum cost flow problem, given the cost from dogs to kennels, we can construct a graph which satisfy the problem, the link between source and dogs node, we set the capacity equals to 1, means n dogs want to go to the kennels, the link between kennels node and sink, we set the capacity equals to 1, means a kennels only contains one dog, and given a n flow, we find the minimum cost flow, and we can compute the minimum amount of money you need to pay in order to send these n little dogs into those n different kennels.

4.3 Complexity

Assume n dogs and n kennels, we have 2n nodes, and $O(n^2)$ edges, the minimum cost flow's time complexity is $O(n^3)$, so the complexity would be $O(n^3)$.

5 Problem 8: Ford-Fulkerson Algorithm

I use Edmonds-Karp Algorithm to get max flow:

```
#coding=utf-8

class Graph(object):
    def __init__(self,*args,**kwargs):
        self.node_neighbors={}
        self.visited={}
        self.length=0

    def clear(self):
        self.visited={}

    def add_nodes(self,nodelist):
        for node in nodelist:
            self.add_node(node)

    def add_node(self,node):
```

```
    if  not node in self.nodes():
        self.node_neighbors[node]=[]
        self.length+=1

def add_edge(self,edge):
    u,v=edge
    if v not in self.node_neighbors[u]:
        self.node_neighbors[u].append(v)

def remove_edge(self,edge):
    u,v=edge
    self.node_neighbors[u].remove(v)

def nodes(self):
    return self.node_neighbors.keys()

def breadth_first_search(self,root=-1):
    queue=[]
    order=[]
    pre=[]
    for i in range(self.length):
        pre.append(-1)

    def bfs():
        while len(queue)>0:
            node=queue.pop(0)
            self.visited[node]=True

            for n in self.node_neighbors[node]:
                if (not n in self.visited) and (not n in queue):
                    pre[n]=node
                    queue.append(n)
                    order.append(n)

    if root!=-1:
        queue.append(root)
        order.append(root)
        bfs()


#    print "order:",order
#    print "pre:",pre
    return order,pre

def LoadData(self):
    #       graph=[[0,65,32,-1],[-1,0,1,64],[-1,-1,0,33],[-1,-1,-1,0]]
    # graph=[[0,4,6,-1,-1,-1,-1],[-1,0,4,7,4,-1,-1],[-1,-1,0,1,12,-1,-
    #Girls And Boys m and n
```

```
        #Numbers=[3,2]
        Numbers=[10,10]
        #Pairs: the boys girls like
        #Pairs=[[1,[2]],[1,[1]],[0]]
        Pairs=[[2,[5,9]],[3,[1,6,10]],[2,[3,8]],[3,[1,8,10]],[2,[4,10]],[7,
        Girls=Numbers[0]
        Boys=Numbers[1]
        Graph=[]
        for i in range(Girls+Boys+2):
            Graph_Son=[]
            for j in range(Girls+Boys+2):
                if i==0:
                    if j==0:
                        Graph_Son.append(0)
                    elif j<=Girls:
                        Graph_Son.append(1)
                    else:
                        Graph_Son.append(-1)
                elif i<Girls+Boys+1 and i>Girls:
                    if j==Girls+Boys+1:
                        Graph_Son.append(1)
                    elif i==j:
                        Graph_Son.append(0)
                    else:
                        Graph_Son.append(-1)
                elif i==j:
                    Graph_Son.append(0)
                else:
                    Graph_Son.append(-1)
            Graph.append(Graph_Son)

        Base=Girls
        for i in range(Girls):
            Key=Pairs[i][0]
            for j in range(Key):
                Value=Pairs[i][1][j]
                Graph[i+1][Base+Value]=1

        #print "Graph:",Graph
        #        graph=[[0,1,1,1,-1,-1,-1],[-1,0,-1,-1,1,-1,-1],[-1,-1,0,-
        return Graph

if __name__=='__main__':
    g=Graph()
    graph=g.LoadData()
    g.add_nodes([i for i in range(len(graph))])
    for i in range(len(graph)):
        for j in range(len(graph[0])):
            if graph[i][j]==0 or graph[i][j]==-1:
                pass
            else:
                g.add_edge((i,j))
#   print "nodes:",g.nodes()
    source=0
    sink=len(g.nodes())-1

    maxFlow=0
    while True:
        g.clear()
        order,pre=g.breadth_first_search(0)
        if not sink in order:
            break
        Key=len(g.nodes())-1
        d=graph[pre[Key]][Key]
        while True:
            pivot=pre[Key]
            d=min(d,graph[pre[Key]][Key])
            Key=pivot
            if Key==source:
                break

        if d==0:
            break
        maxFlow+=d

        print "maxFlow:", maxFlow
        Key=len(g.nodes())-1
        while True:
            pivot=pre[Key]
            if graph[Key][pivot]==-1:
                graph[Key][pivot]=d
            else:
                graph[Key][pivot]+=d
            print "Add:",Key,pivot
            g.add_edge((Key,pivot))
            if graph[pivot][Key]==d:
                graph[pivot][Key]=-1
                print "Remove:",pivot,Key
                g.remove_edge((pivot,Key))

            else:
                graph[pivot][Key]-=d
            Key=pivot
            if Key==0:
                break
#   print "Graph:",graph
```

```
    print "Max Flow:", maxFlow
#order=g.breadth_first_search(0)
```

Input:

```
10 10
2 5 9
3 1 6 10
2 3 8
3 1 8 10
2 4 10
7 1 4 6 7 8 9 10
2 1 8
1 10
1 8
2 4 7
```

Output:

```
maxFlow: 1
Add: 21 15
Remove: 15 21
Add: 15 1
Remove: 1 15
Add: 1 0
Remove: 0 1
maxFlow: 2
Add: 21 11
Remove: 11 21
Add: 11 2
Remove: 2 11
Add: 2 0
Remove: 0 2
maxFlow: 3
Add: 21 13
Remove: 13 21
Add: 13 3
Remove: 3 13
Add: 3 0
Remove: 0 3
maxFlow: 4
Add: 21 18
Remove: 18 21
Add: 18 4
Remove: 4 18
Add: 4 0
Remove: 0 4
```

```
maxFlow: 5
Add: 21 14
Remove: 14 21
Add: 14 5
Remove: 5 14
Add: 5 0
Remove: 0 5
maxFlow: 6
Add: 21 16
Remove: 16 21
Add: 16 6
Remove: 6 16
Add: 6 0
Remove: 0 6
maxFlow: 7
Add: 21 20
Remove: 20 21
Add: 20 8
Remove: 8 20
Add: 8 0
Remove: 0 8
maxFlow: 8
Add: 21 17
Remove: 17 21
Add: 17 10
Remove: 10 17
Add: 10 0
Remove: 0 10
maxFlow: 9
Add: 21 19
Remove: 19 21
Add: 19 6
Remove: 6 19
Add: 6 16
Remove: 16 6
Add: 16 2
Remove: 2 16
Add: 2 11
Remove: 11 2
Add: 11 7
Remove: 7 11
Add: 7 0
Remove: 0 7
Max Flow: 9
```

6 Problem 10: Cycle Canceling

I use Edmonds-Karp algorithm to find suitable flow and Bellman-Ford algorithm to find negative cycle.

```
#coding=utf-8

class Graph(object):
    def __init__(self,*args,**kwargs):
        self.node_neighbors={}
        self.visited={}
        self.length=0

    def clear(self):
        self.visited={}

    def add_nodes(self,nodelist):
        for node in nodelist:
            self.add_node(node)

    def add_node(self,node):
        if  not node in self.nodes():
            self.node_neighbors[node]=[]
            self.length+=1

    def add_edge(self,edge):
        u,v=edge
        if v not in self.node_neighbors[u]:
            self.node_neighbors[u].append(v)

    def remove_edge(self,edge):
        u,v=edge
        self.node_neighbors[u].remove(v)

    def nodes(self):
        return self.node_neighbors.keys()

    def breadth_first_search(self,root=-1):
        queue=[]
        order=[]
        pre=[]
        for i in range(self.length):
            pre.append(-1)

        def bfs():
            while len(queue)>0:
                node=queue.pop(0)
                self.visited[node]=True
                for n in self.node_neighbors[node]:
                    if (not n in self.visited) and (not n in queue):
                        pre[n]=node
                        queue.append(n)
                        order.append(n)

        if root!=-1:
            queue.append(root)
            order.append(root)
            bfs()


        #    print "order:",order
        #    print "pre:",pre
        return order,pre

    def LoadData(self,Graph):
        #Graph=[[0,2,2,-1],[-1,0,1,1],[-1,1,0,1],[-1,-1,-1,0]]
        return Graph

    def ReadData(self,inputFile):
        keyword=['0','1','2','3','4','5','6','7','8','9']
        file=open(inputFile,'rw+')
        line=file.readlines()
        node_len=0
        arc_len=0
        key_file=0
        for i in range(len(line)):
            if line[i][0]=='c':
                pass
            elif line[i][0]=='p':
                length=len(line[i])
                j=0
                while True:
                    t=line[i][-2-j]
                    if t in keyword:
                        arc_len+=int(t)*10**j
                        j+=1
                    else:
                        j+=1
                        break
                print "Arc Length:",arc_len
                a=0
                while True:
                    t=line[i][-2-j-a]
                    if t in keyword:
                        node_len+=int(t)*10**a
                        a+=1
                    else:
                        break
                print "Node Length:",node_len
            elif line[i][0]=='n':
                if line[i][-1]=='s':
                    pass
                elif line[i][-1]=='t':
                    pass
            else:
                key_file=i
                break

        Graph=[]
        for u in range(node_len):
            Graph_son=[]
            for v in range(node_len):
                if u==v:
                    Graph_son.append(0)
                else:
                    Graph_son.append(-1)
            Graph.append(Graph_son)

        cost=[]
        for u in range(node_len):
            cost_son=[]
            for v in range(node_len):
                if u==v:
                    cost_son.append(0)
                else:
                    cost_son.append(-1)
            cost.append(cost_son)
#        print "Graph:",Graph

        for i in range(arc_len):
            #print line[key_file]
            row=int(line[key_file][2])
            #print "row:",row
            col=int(line[key_file][4])
            #print "col:",col
            #print "why:",line[key_file][6]
            Graph[row][col]=int(line[key_file][6])
            cost[row][col]=int(line[key_file][8])
            key_file+=1

        print "Graph:",Graph
        print "Cost:",cost
        return Graph,cost
```

```
def Bellman_Ford(self,Graph,cost):
    n=len(Graph)
    check_Graph=[]
    for i in range(n):
        check_Graph_son=[]
        for j in range(n):
            if i==j:
                check_Graph_son.append(0)
            else:
                check_Graph_son.append(999)
        check_Graph.append(check_Graph_son)
    for i in range(n):
        for j in range(n):
            if Graph[i][j]>=1:
                if cost[i][j]==-1 and cost[j][i]!=-1:
                    check_Graph[i][j]=-cost[j][i]
                elif cost[i][j]!=-1:
                    check_Graph[i][j]=cost[i][j]
                else:
                    pass

    print "Check Graph:",check_Graph

    ecost=[]
    for i in range(n):
        ecost.append(999)
    ecost[0]=0
    path=[]
    for i in range(n):
        path.append(-1)

    for i in range(n):
        for j in range(n):
            if check_Graph[i][j]!=999 and ecost[i]+check_Graph[i][j]<ec
                ecost[j]=ecost[i]+check_Graph[i][j]
                path[j]=i
    print "ecost before:",ecost
    Verify=False
    for i in range(n):
        for j in range(n):
            if check_Graph[i][j]!=999 and ecost[i]+check_Graph[i][j]<ec
                print "Detect Negative Graph, a graph without negative
                path[j]=i
                Verify=True

    print "ecost:",ecost

    print "Path:",path
```

```
        return Verify,path,check_Graph



if __name__=='__main__':
    g=Graph()
    Gra,cost=g.ReadData("/Users/zhangyuan/Desktop/learn/inputFlow.txt")
    graph=g.LoadData(Gra)
    g.add_nodes([i for i in range(len(graph))])
    for i in range(len(graph)):
        for j in range(len(graph[0])):
            if graph[i][j]==0 or graph[i][j]==-1:
                pass
            else:
                g.add_edge((i,j))
    # print "nodes:",g.nodes()
    source=0
    sink=len(g.nodes())-1

    maxFlow=0
    while True:
        g.clear()
        order,pre=g.breadth_first_search(0)
        if not sink in order:
            break
        Key=len(g.nodes())-1
        d=graph[pre[Key]][Key]
        while True:
            pivot=pre[Key]
            d=min(d,graph[pre[Key]][Key])
            Key=pivot
            if Key==source:
                break

        if d==0:
            break
        maxFlow+=d

        print "maxFlow:", maxFlow
        Key=len(g.nodes())-1
        while True:
            pivot=pre[Key]
            if graph[Key][pivot]==-1:
                graph[Key][pivot]=d
            else:
                graph[Key][pivot]+=d
            print "Add:",Key,pivot
            g.add_edge((Key,pivot))
            if graph[pivot][Key]==d:
                graph[pivot][Key]=-1
                print "Remove:",pivot,Key
                g.remove_edge((pivot,Key))

            else:
                graph[pivot][Key]-=d
            Key=pivot
            if Key==0:
                break
#   print "Graph:",graph

    print "Final Graph:",graph
    print "Max Flow:", maxFlow

#order=g.breadth_first_search(0)
    minCost=0
    check_num=0
    while True:
        verify,path,check_Graph=g.Bellman_Ford(graph,cost)
        if verify==False:
            if check_num==0:
                for i in range(len(graph)):
                    for j in range(len(graph)):
                        if graph[i][j]>0 and check_Graph[i][j]<=0:
                            minCost+=-1*graph[i][j]*check_Graph[i][j]
                print "Min Cost:", minCost
                print "Current flow is:",graph
                break
            else:
                print "Min Cost:",minCost
                print "Current flow is:",graph
                break
        check_num=1
        for i in range(len(graph)):
            for j in range(len(graph)):
                if graph[i][j]>0 and check_Graph[i][j]<=0:
                    minCost+=-1*graph[i][j]*check_Graph[i][j]
        print "Cost:",minCost

        min_flow=0
        cycle=[]
        unit_cost=0
        for i in range(len(path)):
            if path[i]!=-1:
                cycle.append(graph[path[i]][i])
                unit_cost+=check_Graph[path[i]][i]
                print "unit_cost:",unit_cost
        minCost+=unit_cost*min(cycle)
        print "New Cost:",minCost
        flow_transfer=min(cycle)
        for i in range(len(path)):
            graph[path[i]][i]-=flow_transfer
            if graph[i][path[i]]==-1:
                graph[i][path[i]]=flow_transfer
            else:
                graph[i][path[i]]+=flow_transfer
            if graph[path[i]][i]==0:
                    graph[path[i]][i]=-1
        print "New Graph:",graph
```

My Input as follows:

```
c This is a simple example file to verify the minimum cost flow program.
c Problem line (nodes, links).
p max 4 5
c source
n 1 s
c sink
n 4 t
c Arc descriptor lines(from, to, capacity)
a 0 1 2 4
a 0 2 2 1
a 2 1 1 2
a 1 3 1 5
a 2 3 1 2
```

My output as follows:

```
Arc Length: 5
Node Length: 4
Graph: [[0, 2, 2, -1], [-1, 0, -1, 1], [-1, 1, 0, 1], [-1, -1, -1, 0]]
Cost: [[0, 4, 1, -1], [-1, 0, -1, 5], [-1, 2, 0, 2], [-1, -1, -1, 0]]
maxFlow: 1
Add: 3 1
Remove: 1 3
Add: 1 0
maxFlow: 2
Add: 3 2
Remove: 2 3
Add: 2 0
Final Graph: [[0, 1, 1, -1], [1, 0, -1, -1], [1, 1, 0, -1], [-1, 1, 1, 0]]
Max Flow: 2
Check Graph: [[0, 4, 1, 999], [-4, 0, 999, 999], [-1, 2, 0, 999], [999, -5,
ecost before: [0, 3, 1, 999]
Detect Negative Graph, a graph without negative cycle would not decrease a
```

```
ecost: [0, 3, 1, 999]
Path: [1, 2, 0, -1]
Cost: 12
unit_cost: -4
unit_cost: -2
unit_cost: -1
New Cost: 11
New Graph: [[0, 2, -1, -1], [-1, 0, 1, -1], [2, -1, 0, -1], [-1, 1, 1, 1]]
Check Graph: [[0, 4, 999, 999], [999, 0, -2, 999], [-1, 999, 0, 999], [999,
ecost before: [0, 4, 2, 999]
ecost: [0, 4, 2, 999]
Path: [-1, 0, 1, -1]
Min Cost: 11
Current flow is: [[0, 2, -1, -1], [-1, 0, 1, -1], [2, -1, 0, -1], [-1, 1,
```