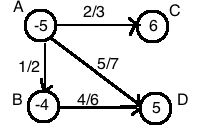
CIRCULATION PROBLEM WITH LOWER BOUND

1. Problem description:

Given a directed graph with demand value on each node, lower bound and capacity on each edge, we want to check if those demands on each vertex can be satisfied by meeting those requirements on edges. For example:



Figure

We will say it’s a feasible circulation if node A can flow out 5 unit flow, node B can flow out 4, node C can receive 6, and node D can receive 6 and flows obey the edge restrictions. (Notation on edges: a/b means lower bound is a, and capacity is b)

1. Data Structure:

We use adjacent lists to store the whole graph information, including the node name, node demand, edge lower bound and capacity.

Graph:

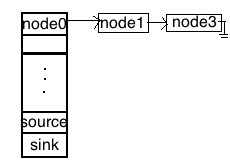


Figure 2

Firstly, we initiate a node structure to describe a node (each single square in figure 2.)

typedef struct Node{

int name;

int demand;

int capacity;

int lowerbound;

struct Node\* next;//point to next node

}Node;

Secondly, we create a structure to describe the graph (the array in figure 2).

typedef struct Adjlist{

int name;

int demand;

int capacity;

int lowerbound;

Node \*head;

}Adjlist;

Then the graph will be represented by Adjlist array[number of vertices +2](Plus 2 due to the souce and sink node)

The name of nodes should be 0,1,2,….and they will be stored in the array in order.(The index in the array indicates their name.) The capacity and lower bound of edge(i,j) will be stored to node j who appears in the linked list of array[i]. In part 4, we will explain how to input the graph more.

Additionally, we use 2 nxn adjacent matrix to store the residual graph and capacity information because it’s easier to be used in computing maximal flow.

Matrix[i][j] indicates the edge(i,j).

1. Algorithm and Time complexity:

The following is the flow chart of our algorithm:

Figure 3

Step1: Eliminate lower bound on every edge

We iterate on every edge, if there is a lower bound on edge (i,j), we set the new capacity on to original capacity minus lower bound, decrease the demand of node i by the value of lower bound, increase the demand of node j by the value of lower bound like following:

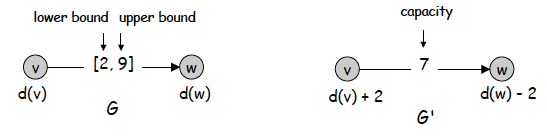


Figure 4

*Time complexity: O(nxm)* (n: number of vertices, m: number of edges)

Step2: Check if Σvdv = 0. If not, there is no solution.

To sum up all the demand value on each node, we only go through the array from array[0] to array[number of vertices -1]. There is no need to go through the linked list on each node. In this step, we also sum up the positive demand value, it will be used in step 5.

*Time complexity: O(n)*

Step3: Add source(S\*) and sink node(t\*)

When we initialized the array for the graph, we have already get the space for source (represented by array[number of vertices]) and sink node(represented by array[number of vertices]). In this step, we try to create edges to let source node point to the nodes with negative demand, and let the nodes with positive demand point to the sink node. All the capacity of these new edges depend on the absolute value of demand of those nodes. We go through the nodes, if its demand value is negative, we add one new node which has the same information of the negative demand node to the linked list of array[number of vertices]. If its demand is positive, we add one new node whose name is number of vertices+1(the name of sink node) to the linked list of array[the name of positive demand node]. For example:

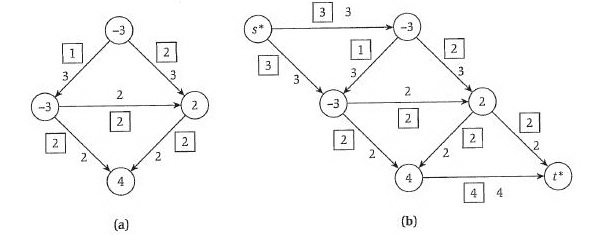


Figure 5

*Time complexity: O(n) since we only need to go through every node.*

Step4: Find maximum flow by Ford-Fulkerson algorithm

To compute a max flow, we use 2 adjacent matrix as a new data structure to store the residual graph and capacity like we mentioned before.

The two main functions in our implement:

(1) find a path in a residual graph, and (2) compute max flow.

(1)We name the function of finding a path “DFS” because we use depth first search to find paths. First, we define “ADDED” as “true” and “UNADDED” as “false”, which is for checking whether a node has been added when finding a path. In this function we search from the source node. If there is any adjacent node that has not been visited, we call DFS again and search the adjacent node. We call DFS recursively until we reach the sink (return true) or no node can be visited (return false). We record the path when running this function, so that we can have a stored path when returning true.

Pseudocode:

bool DFS(int path[], int start, int target, int path\_index){

set start as “ADDED”;

set path\_index as start;

if (start==target)

we find a path, return true;

while (we haven’t reached the target node)

if any adjacent node i of start node has not been visited

set i as new start node;

call DFS recursively with i=new start;

}//end while

}

Because this function runs DFS, the *time complexity* is O(n+m).

(2)For computing maxflow, we implement Ford-Fulkerson Algorithm as the function. We repeatedly check whether there is a path in the residual graph. If yes, we update flow in the residual graph and check again until there is no path in the residual graph. When we finish checking, the final flow should be the max flow. Because it is not possible to update flow in the path recursively with adjacent lists we define, we use matrix “flow” for updating. After we find a path, we trace it and update the flow matrix.

Pseudocode:

void max\_flow(int path[],int start, int target){

while there is a path in the residual graph

calculate the bottleneck b, and update the flow with b

update flow of edges in the path:

if e is a forward edge, decrease f(e) by b in the residual graph

else if e is a backward edge, increase f(e) by b in the residual graph

}

In the while loop, we update the flow of a path, whose time complexity is O(n). However, every time we find a path takes O(n+m), so the total *time complexity is O(n+m).*

Step5: If Σv:dv>0dv = value of max flow, this circulation is feasible

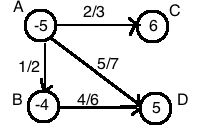
By checking the two values, we will know if this circulation is feasible.

If yes, we will print out the flow paths.

If no, we will say it’s not feasible.

1. Input method:

For example: (node A=node 0, B=1, C=2, D=3)



Figure

1. Enter the number of vertices in the graph:4
2. Enter demand of each node
3. Enter number of neighbors of vertex0,1,…..

-Enter name, demand, lower bound, capacity about the neighbors(Those numbers are separated by a space.)

For example, we want to enter the information for node 0.

The number of its neighbors is 3. For neighbor node1 with demand -4, lower bound 1, capacity 2, we enter “1 -4 1 2”. In the same way, the input for neighbor node2 will be “2 6 2 3”…etc

1. Examples and outputs:

We will output the new graph which is after step 3 if the sum of demands equals zero first, then tell user the result. (The demand of source and sink node won’t be used, so we didn’t initialize it. It’s normal to be a weird value)

Example 1: (Figure 6)

The result is “Circulation for this graph is not feasible because the sum of demands is not zero”.

Example 2:

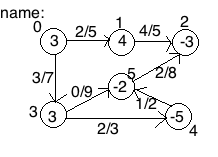


Figure 7

In this case, result will be “Positive demand value is not equal to maxflow, this graph is not feasible”.

Example 3: (Changed from lecture notes)

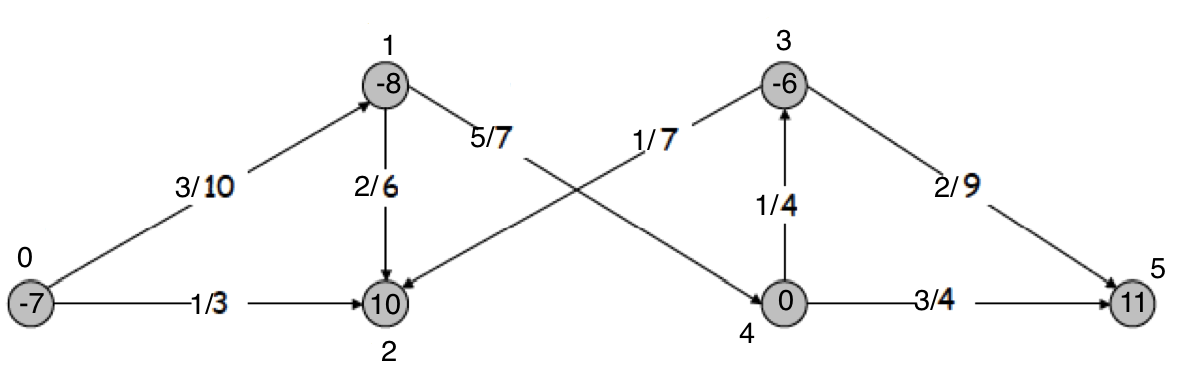


Figure 8

The result will be “Positive demand value is equal to maxflow, this graph is feasible”. We will also print out the process of finding maximal flow and how the flow goes corresponding to the new graph after step3. We put the input and output detail in the appendix.

1. Reference:
   1. [B609 Lecture notes] Lecture 4 flow by professor Haixu Tang.
   2. [Algorithm Design] Chapter 7 Network Flow by Jon KleinBerg and Eva Tardos.

Appendix: Whole input and output for figure 8

Please enter how many vertices in your graph:

6

You have 6 vertices

Please enter demand of vertex0

-7

Please enter demand of vertex1

-8

Please enter demand of vertex2

10

Please enter demand of vertex3

-6

Please enter demand of vertex4

0

Please enter demand of vertex5

11

Please enter the number of neighbors of vertex 0

2

Please enter following information about the neighbor of vertext0

Enter name, demand, lowerbound, capacity in order and separeated by one space

1 -8 3 10

name:1, demand:-8, lowerbound:3, capacity:10

Please enter following information about the neighbor of vertext0

Enter name, demand, lowerbound, capacity in order and separeated by one space

2 10 1 3

name:2, demand:10, lowerbound:1, capacity:3

Please enter the number of neighbors of vertex 1

2

Please enter following information about the neighbor of vertext1

Enter name, demand, lowerbound, capacity in order and separeated by one space

4 0 5 7

name:4, demand:0, lowerbound:5, capacity:7

Please enter following information about the neighbor of vertext1

Enter name, demand, lowerbound, capacity in order and separeated by one space

2 10 2 6

name:2, demand:10, lowerbound:2, capacity:6

Please enter the number of neighbors of vertex 2

0

Please enter the number of neighbors of vertex 3

2

Please enter following information about the neighbor of vertext3

Enter name, demand, lowerbound, capacity in order and separeated by one space

2 10 1 7

name:2, demand:10, lowerbound:1, capacity:7

Please enter following information about the neighbor of vertext3

Enter name, demand, lowerbound, capacity in order and separeated by one space

5 11 2 9

name:5, demand:11, lowerbound:2, capacity:9

Please enter the number of neighbors of vertex 4

2

Please enter following information about the neighbor of vertext4

Enter name, demand, lowerbound, capacity in order and separeated by one space

3 -6 1 4

name:3, demand:-6, lowerbound:1, capacity:4

Please enter following information about the neighbor of vertext4

Enter name, demand, lowerbound, capacity in order and separeated by one space

5 11 3 4

name:5, demand:11, lowerbound:3, capacity:4

Please enter the number of neighbors of vertex 5

0

Sum of positive demand:12

0: demand=-3

i:0,name:1,demand:-11,lowerbound:3,capacity:7

i:0,name:2,demand:9,lowerbound:1,capacity:2

1: demand=-4

i:1,name:4,demand:-5,lowerbound:5,capacity:2

i:1,name:2,demand:8,lowerbound:2,capacity:4

2: demand=6

i:2,name:7,demand:0,lowerbound:0,capacity:6

3: demand=-4

i:3,name:2,demand:9,lowerbound:1,capacity:6

i:3,name:5,demand:9,lowerbound:2,capacity:7

4: demand=-1

i:4,name:3,demand:-7,lowerbound:1,capacity:3

i:4,name:5,demand:8,lowerbound:3,capacity:1

5: demand=6

i:5,name:7,demand:0,lowerbound:0,capacity:6

6: demand=-774778415

i:6,name:0,demand:-3,lowerbound:0,capacity:3

i:6,name:1,demand:-4,lowerbound:0,capacity:4

i:6,name:3,demand:-4,lowerbound:0,capacity:4

i:6,name:4,demand:-1,lowerbound:0,capacity:1

7: demand=-774778415

Begin to calculate maxflow:

Find a path: Source -> 0 -> 1 -> 2 -> Sink

Bottleneck of this path: 3

Find a path: Source -> 1 -> 0 -> 2 -> Sink

Bottleneck of this path: 2

Find a path: Source -> 1 -> 2 -> Sink

Bottleneck of this path: 1

Find a path: Source -> 1 -> 4 -> 3 -> 5 -> Sink

Bottleneck of this path: 1

Find a path: Source -> 3 -> 2 -> 0 -> 1 -> 4 -> 5 -> Sink

Bottleneck of this path: 1

Find a path: Source -> 3 -> 5 -> Sink

Bottleneck of this path: 3

Find a path: Source -> 4 -> 1 -> 0 -> 2 -> 3 -> 5 -> Sink

Bottleneck of this path: 1

maxflow of this circulation is: 12

Positive demand value is equal to maxflow, this graph is feasible.

The new flow is as following:

Node 0 to node1: 1

Node 0 to node2: 2

Node 1 to node4: 1

Node 1 to node2: 4

Node 2 to node7: 6

Node 3 to node2: 0

Node 3 to node5: 5

Node 4 to node3: 1

Node 4 to node5: 1

Node 5 to node7: 6

Source node to node0: 3

Source node to node1: 4

Source node to node3: 4

Source node to node4: 1