Problem 1: Implement Ford-Fulkerson on Bipartite Matching

Screenshots taken from Jupiter Notebook, I have all necessary constructions and explanations written in-between. Will attach a .py version of my code converted automatically from Jupyter and the .ipnb version as well.

For bipartite matching,

the construction is to introduce source- and sink nodes to the bipartite graph G(V,E) call ford-fulkerson on a new graph G'(V', E') where we mark source node by index 0 and sink (V+1). Every internal node on the left, X, has an edge of weight 1 to the sink s; every internal node on the right, Y, has an edge of weight 1 to the sink t; the original edges E remains, each assigned with weight 1. We will have the perfect matching if ford-fulkerson returns a valid max flow value and every vertex is visited if not we will determine the size of the largest bipartite matching.

In [18]:
```
# Deal with the data storage

V = 200
src = 0
sink = V+1
Vp = V+2 # V' as V prime

graph = [[0 for col in range(Vp)] for row in range(Vp)]

import string
def readFileToArray( filename ):
    arr = [] # List of rankings
    with open( filename ) as file:
        for List in file:
            tmp = List.strip(string.whitespace)
            arr.append(tmp)
    file.close()
    return arr
# assign the correspding edge weight to 1 in our adjacency list
for i in range(len(edge)):
    idx0 = int(edge[i].split(' ')[0])
    idx1 = int(edge[i].split(' ')[1])
    graph[idx0][idx1] = 1

for i in range(1,101):
    graph[0][i] = 1 # each node in X are connected to source
    graph[i+100][201] = 1
```

In [19]:
```
graph[0][0:10]
```

Out[19]:
```
[0, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

Ford-Fulkerson algorithm using BFS

For this part, I collaborated with Jin Yue, Xiang Li, Jingjing Zhu and Yanjia Zhang since this is the main algorithm of our course project- Image Segmentation- for which we have submitted earlier a similar version of this implementation. The following is my implementation for this homework 5.1

In [20]:
```
def ford_fulkerson(graph, s, t, V):
    # Input graph stored in adjacency list, for example,
    # graph = [[0, 1, 1, 0],
    #          [0, 0, 1, 1],
    #          [0, 0, 0, 1],
    #          [0, 0, 0, 0]]
    # in this case, we have four vertices: source, two internal nodes and sink.
    # The first row corresponds to the edges starting from the source node s, e.g. there are t
wo edges of weight 1:
    # from s to node 1 and from s to node 2
    # both node 1 and 2 has an edge to the sink node t

    # V is the dimension of our adjacency list

    max_flow = 0
```

```
    rgraph = [[0 for col in range(V)] for row in range(V)]
    # initialize residual graph
    for i in range(V):
        for j in range(V):
            rgraph[i][j] = graph[i][j]

    parent = [0 for i in range(V)]

    # finding s-t augmenting paths in the residual graph

    while bfs(rgraph, s, t, parent, V):
        # initial flow we now recursively search from sink
        # and calculate bottleneck
        flow = float("inf")
        v = t
        while v != s:
            u = parent[v]
            flow = min(flow, rgraph[u][v])
            v = parent[v]
        bottleneck = flow
        v = t
        while v != s:
            u = parent[v]

            rgraph[v][u] += bottleneck
            # forward edge
            rgraph[u][v] -= bottleneck
            # backward edge
            v = parent[v]

        max_flow += bottleneck  # get the total values of flows

    # get all nodes that are visited
    parent = [0 for i in range(V)]
    visited = bfs2(rgraph, s, parent, V)
    print "The total values of flows is: %d" % max_flow
    return visited
```

```
# determine if there is a path using BFS
# return True or False, boolen value
def bfs(rgraph, s, t, parent, V):
    # create a list to store visited nodes
    # initialize to False
    visited = [False for i in range(V)]
    visited[s] = True
    # parent remembers the start node of each edge
    parent[s] = -1

    q = list()
    q.insert(0, s)

    # standard bfs
    while len(q) != 0:
        row = q.pop()
        for i in range(V):
            if not visited[i] and rgraph[row][i] > 0:
                q.insert(0, i)
                parent[i] = row
                # mark its parent as the current row node
                visited[i] = True
                # mark the node as visited
    return visited[t]
```

```
# bfs2: exactly the same as bfs
# except that we now return
# the complete array of the visited nodes
def bfs2(rgraph, s, parent, V):

    visited = [False for i in range(V)]
    visited[s] = True
    parent[s] = -1
    q = list()
    q.insert(0, s)
    while len(q) != 0:
        row = q.pop()
        for i in range(V):

            if not visited[i] and rgraph[row][i] > 0:
                q.insert(0, i)
                parent[i] = row
                visited[i] = True
    return visited
```

In [21]:

```
# sample small dataset borrowed from CLRS book 286.8(C) to validate my algorithm
sample_V = 11
sample_src = 0
sample_sink = 10
sample_graph = [[0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0],
         [0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
         [0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0],
         [0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0],
         [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
         [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
         [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
         [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
         [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
         [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
         [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]

match = ford_fulkerson(sample_graph, sample_src, sample_sink, sample_V) # test out to be 3, as
 was shown in the book
```

```
The total values of flows is: 3
```

In [22]:

```
match = ford_fulkerson(graph, src, sink, Vp)
# print those vertices who are not being visited
print "Vertices not being visited:"
for i in range(1,V+1):
    if not match[i]:
        print i
```

```
The total values of flows is: 99
Vertices not being visited:
1
4
8
13
21
23
38
39
41
47
75
78
92
99
119
123
124
130
133
137
140
158
160
170
171
174
176
185
197
```

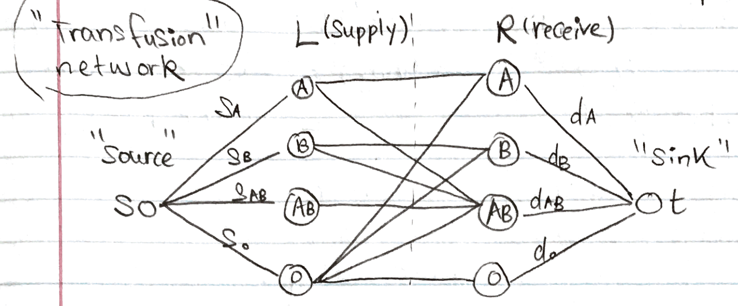
Problem 2: Blood Supply and Demand

EC504 HW5 Qiuxuan Lin U09077467

(Collaborated with Yanjia Zhang)

Problem 2: Blood Supply

(a) One can think of this problem as a bipartite matching problem, where L and R both consist of four blood types, e.g.



$G(V, E)$:
$V = L \cup R \cup \{s, t\}$

E: { $e(s,u), \forall u \in L$, capacity given by supply units, there is an edge if $v \in R$ can receive the blood type from $u \in L$, $e(v,t), \forall v \in R$ capacity given by demand units. }

This completes the construction of our flow network.
Note that by running Ford-Fulkerson does not solve the problem completely. We can only say that the blood on hand would suffice iff $e(r,t), \forall r \in R$, are fully "saturated", e.g. actual flow meets capacity limit.

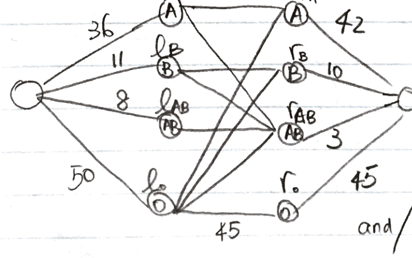
Running time: $|V| = 2\times4+2$ $|E| = 2\times4+7$ ~ constant C_0

the running time would be bounded by the maximum flow $|f^*|$ through this transfusion network.

$|f^*| \leq S_A + S_B + S_{AB} + S_O = \sum_{\forall \text{all blood types}} S_K$

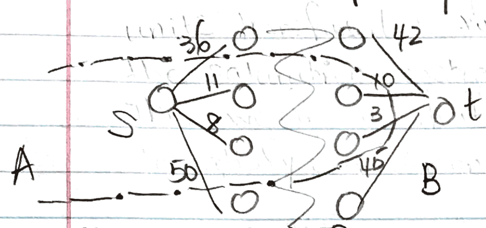
$\therefore \sim O(|f^*|)$

(b) Translate the given table into our transfusion network,



"Simple observation":
$c(r_O, t) = 45 = d_O$, only 5 units of type O left.
$c(r_A, t) = 42$, for r_A, in-coming flow has two possibilities, $l_A \to r_A$ and/or $l_O \to r_A$, $36+5=41 < 42$.

(1) Min-cut capacity, consider a cut to the source, the minimum cut would be, (illustrated)



$$Cut(A,B) = \sum f_{out} - \cancel{\sum f_{in}} \rightarrow 0$$

$$= 36+50+10+3 = 99 < 100$$

(2) Run ford-fulkerson (see screen shots).

Total number of flow = 99 < 100

can't satisfy!

For clinic admin.'s:

refer to the "simple observation" sketch, on previous page

type O demands 45 units.

type A demands 42 units

even if we have all 5 units of type O (50−45 = 5)

transfused to type A receivers, 36 + 5 = 41, we'd only be able to supply 41 units of type A, which is short of our demand 42 units.

In [2]:

```
V = 10
src = 0
sink = 9
# I take the liberty to set "transfusion" edges to 200
# (instead of infinity, e.g. there is no much constraint for one
# to send flows from supply side to demand side)
# as it would be the easier way
graph = [[0, 36, 11, 8, 50, 0, 0, 0, 0, 0],
         [0, 0, 0, 0, 0, 200, 0, 200, 0, 0],
         [0, 0, 0, 0, 0, 0, 200, 200, 0, 0],
         [0, 0, 0, 0, 0, 0, 0, 200, 0, 0],
         [0, 0, 0, 0, 0, 200, 200, 200, 200, 0],
         [0, 0, 0, 0, 0, 0, 0, 0, 0, 42],
         [0, 0, 0, 0, 0, 0, 0, 0, 0, 10],
         [0, 0, 0, 0, 0, 0, 0, 0, 0, 3],
         [0, 0, 0, 0, 0, 0, 0, 0, 0, 45],
         [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
flow = ford_fulkerson(graph, src, sink, V)
```

```
The total values of flows is: 99
```

Problem 3: The escape problem

EC504 HW5
Qiuxuan Lin (Shane)
U09077467

26-1 Escape problem

1a) Let us denote the original undirected graph as $G(\bar{V}, E)$ with vertex and edge capacity constraints.

The goal, in essence, is to transform G into a classic flow network G' such that the new network only has edge capacities. Construct $G'(\bar{V}', E')$ in the following way,

① split every $v \in \bar{V}$ into two vertices V_{in} and V_{out}, the directed edge $V_{in} \rightarrow V_{out}$ has a capacity corresponds to v. Formulate this splitting formally,

$$\bar{V}': \text{edge capacity } c'(V_{in}, V_{out}) = c(v), \quad \forall v \in \bar{V}$$

② Consider edges in G, e.g. $e(u,v)$, would now be represented as a flow from U_{out} to V_{in},

$$\bar{V}': \text{edge capacity } c'(U_{out}, V_{in}) = c(u,v), \quad \forall e(u,v) \in \bar{V}$$

To sum up, we have now reduced G into an ordinary max-flow problem with,

$$G': \quad V' = \{ V_{in}, V_{out} \mid v \in \bar{V} \}$$
$$E' = \{ U_{out}, V_{in} \mid (u,v) \in E \} \cup \{ V_{in}, V_{out} \mid v \in \bar{V} \}$$

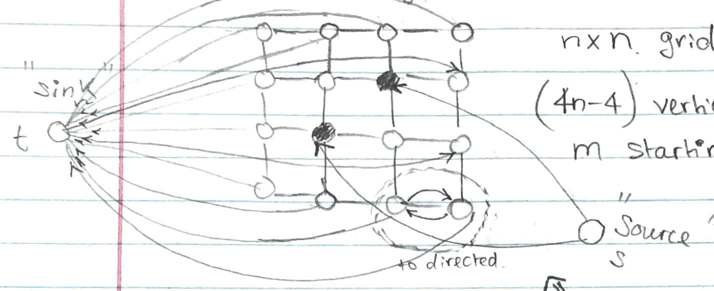
and edge capacities assigned as,

$$c' = \begin{cases} c'(V_{in}, V_{out}), \text{ if } v \in \bar{V} \\ c'(U_{out}, V_{in}), \text{ if } (u,v) \in \bar{V} \\ c'(u,v) = 0, \text{ else} \end{cases}$$

Although not specified in the problem statement, if the original graph G has a source and sink node, s and t, they would have become s_{in} and t_{out} respectively.

New graph G': $|V'| = 2 \cdot |V|$, $|E'| = |V| + |E|$.

(b) Describe an efficient algorithm to solve this escape problem and analyze its running time.



$n \times n$ grid, $(2n^2-2n)$ undirected path

$(4n-4)$ vertices on the boundary.

m starting points

The construction of the illustrated graph $G(\bar{V}, E)$:

① Vertices $\bar{V}$: $\{s, t\} \cup \{\text{all vertices in the grid}\}$

introducing artificial source and sink.

② Edges E:
- $e(s, v_k)$, $k = 1, 2, \dots m$

  from source to each starting point
- $e(v_b, t)$, for $b \in$ Boundary.

  from every boundary point to the sink t
- $e(u, v)$, if u, v are neighbors

  make undirected edge into two directed edges

③ To satisfy the constraint that escape paths must be disjoint to one another, we assign edge capacity as 1 to each edge in G and vertex capacity as 1 to each vertex.

$\Rightarrow$ If the max flow in G is m, then there must be m disjoint paths in the flow network. This is by construction, since each flow, originated at source, would go through one starting point and escape through one boundary point. Suppose the flow is less than m, there must be one vertex (out of the starting points) not visited.

Running time: $|V| = n^2+2$, $|E| = m + 4n-4 + 2(2n^2-2n) \sim n^2$

$\therefore$ The total running time, $O(|E||f^*|) \sim \boxed{O(n^3)}$ $|f^*| \leq 4n-4$, max flow