CS2420 Program 8

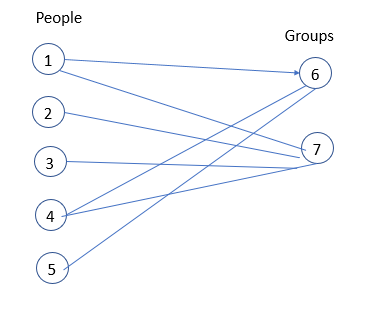
Min Cost Max Flow

After the semester ended, a student wrote:

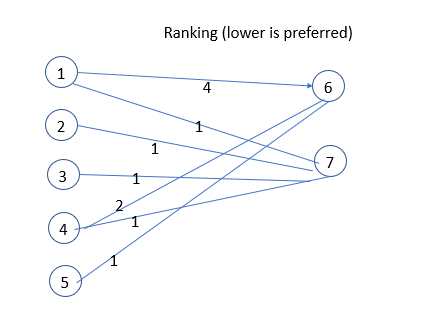
  I am working for a professor to develop a program to sort students into their desired project groups with the highest level of satisfaction. Each student will pick 3-5 groups ranked from 1st preference to last preference.  Groups are limited in size. I think it is a max flow problem, but I’m not sure how to make it work.

This can be solved by a max flow min cost algorithm.

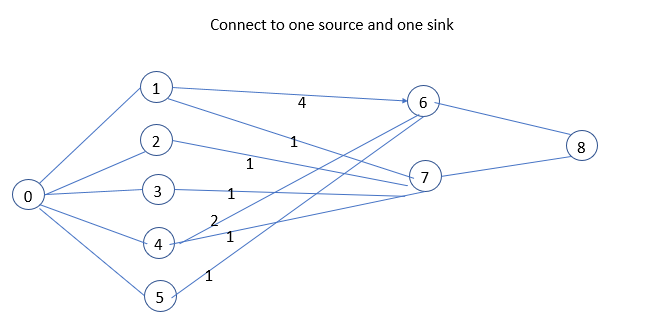
You set it up as a bipartite graph, with people and groups. For simplicity, I’ll assume 5 people and two groups.



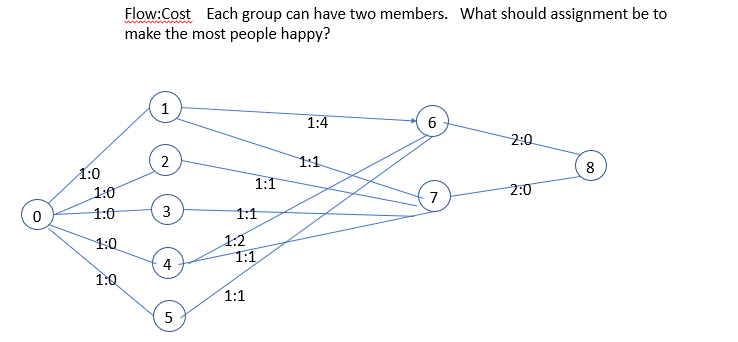
In the figure below, each edge is labeled with a ranking (1 is best) which functions as a cost. You will notice that some people didn’t give a second choice. Person1 gave group six a rank of 4 to indicate his low preference.



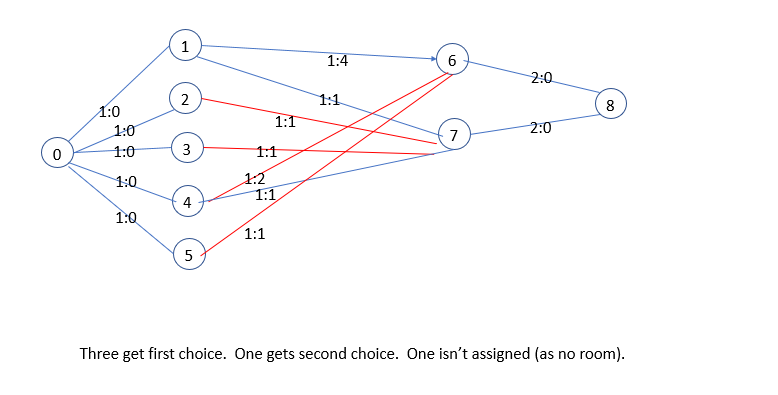
To turn it into a max flow problem, we need to have one source and one sink.



We need to add capacity (not just cost). Capacity is 1 for all arcs, except for the groups. The capacity of each group is shown below as 2. If you wanted each person to belong to two groups, how would that be done?



Notice the assignment below maximized the flow (as both groups are full), but the cost is only 5. That is the goal of a min-cost max flow: find the cheapest way to maximize flow.



So what is the algorithm? It is like the max flow we discussed in class.

Let U(i,j) be the capacity of an edge if this edge exists. And let C(i,j) be the cost per unit of flow along this edge.  And finally let F(I,j) be the flow along the edge. Initially all flow values are zero.

We **modify** the network as follows: for each edge (i,j) add the **reverse edge** (j,i) to the network with the capacity of 0  and the cost C(j,i) = -C(i,j). This indicates that going backwards along an edge removes the cost incurred by going forward. In addition we will always keep the condition F(j,i) = -F(i,j) true during the steps of the algorithm. This means that if you push flow forward on an edge, you can push the same amount backwards on the reverse edge. Since the capacity of the reverse edge is 0, having a negative flow slows flow (as the residual is capacity – flow). So if the edge (I,j) has capacity 1 and flow 1, there is no more flow that can go forward, but on the reverse edge (having capacity 0 and flow -1), the residual is 0-(-1) or 1.

We define the **residual network** just like in Ford Fulkerson. The residual network contains only unsaturated edges (flow < capacity) and the capacity is U(i,j)-F(i,j).

Now we can talk about the **algorithms** to compute the minimum-cost flow. At each iteration of the algorithm we find the shortest path in the residual graph from source to sink. However, we look for the shortest path in terms of the cost of the path, instead of the number of edges. If there doesn't exists a path anymore, then the algorithm terminates. If a path was found, we increase the flow along it as much as possible (i.e. we find the minimal residual capacity  of the path, and increase the flow by it, and reduce the back edges by the same amount). We repeat until there is no more flow possible.

Because we have negative edge weights (negative costs), we can’t use Dijkstra’s algorithm, but need to use the Bellman-Ford or the modified Dijkstra’s algorithm instead. The following is intended to be an algorithm (not java code). I modified the algorithm in your notes slightly to always reset the predecessors. Also, I returned a Boolean to indicate if the algorithm had found more flow. It finds the shortest path to every node (from the source), but the only one I care about is the path to the sink.

**boolean** DijkstraNeg() {  
 Queue<Integer> q = new Queue;  
 **for** each vertex v {  
 v.distance = INFINITY;  
 v.pred = -1; // Each node stored its predecessor in the shortest path  
 }  
 source.distance = 0;  
 q.enqueue(source);  
 **while** (!q.isEmpty()) { // while anything has changed, keep updating  
 vertex v = q.dequeue();  
 **for** each vertex w adjacent to v having cost edgeCost {  
 **if** v.distance + edgeCost < w.distance){  
 w.distance = v.distance + edgeCost;  
 w.pred = v;  
 q.enqueue(w); // the distance to w has changed so its successors are updated  
 }  
 }  
 }  
 **return** sink.pred >= 0; // Did you find a path to the sink?  
}

My output for group1.txt is as follows. The specific assignment may vary, but the best assignment should have a cost of 5.

**found flow 1: 0 5 6 8**

**found flow 1: 0 4 7 8**

**found flow 1: 0 3 7 8**

**found flow 1: 0 2 7 4 6 8 // Since reverse edges are used, you undo a previous assignment**

**group1.txt Max Flow SPACE 4 assigned 4**

**Edge (2,7) assigned 1(1)**

**Edge (3,7) assigned 1(1)**

**Edge (4,6) assigned 1(2)**

**Edge (5,6) assigned 1(1)**

**TotalCost = 5**

**Hints:**

**I’ve given you starter code, which may be useful. Don’t feel you need to use it if you don’t want.**

**Test your shortest path separately (before trying to use it as part of the complete algorithm). The file bellman0.txt may be of use to test this. While this is NOT required, you are going to be so much happier if you test chunks separately.**

**I found it simpler to have a residual graph that just showed how much flow was currently allowed on each edge. Costs never change in the graph. I didn’t have to remember whether I was going forward or backward along an edge if I only looked at the residual graph. You can see remnants of how I used it in the starter code. In particular, the flow on an edge was the difference between the capacity and the residual (what more could be sent on the edge).**

**group7.txt and graph8.txt are graphs which model a different application, but still wants a minCost, max flow solution. The cost is the cost per unit of flow. You find the minimum cost to push one unit of flow through the graph, BUT if you can push more than one unit, you do so as you won’t find a better path.**