

# Week 9 Applied Sheet

## (Solutions)

**Useful advice:** The following solutions pertain to the theoretical problems given in the applied classes. You are strongly advised to attempt the problems thoroughly before looking at these solutions. Simply reading the solutions without thinking about the problems will rob you of the practice required to be able to solve complicated problems on your own. You will perform poorly on the exam if you simply attempt to memorise solutions to these problems. Thinking about a problem, even if you do not solve it will greatly increase your understanding of the underlying concepts. Solutions are typically not provided for Python implementation questions. In some cases, pseudocode may be provided where it illustrates a particular useful concept.

## Problems

**Problem 1.** Let  $G$  be a flow network and let  $f$  be a valid flow on  $G$ . Prove that the net outflow out of  $s$  is equal to the net inflow into  $t$ .

### Solution

Suppose the set of all vertices is  $V$ . We know that the flow of every cut is equal to the flow of the network. Therefore the flow of the cut  $(\{s\}, V \setminus \{s\})$  - which is the flow out of  $s$  - must equal the flow of the cut  $(V \setminus \{t\}, \{t\})$  - which is the flow into  $t$ .

**Problem 2.** Consider a variant of the maximum network flow problem in which we allow for multiple source vertices and multiple sink vertices. We retain all of the capacity and flow conservation constraints of the original maximum flow problem. As in the original problem, all of the sources and sinks are excluded from the flow conservation constraint. Describe a simple method for solving this problem.

### Solution

We create a new graph by adding a “super-source” and “super-sink” vertex. The super source is connected to each source by an edge with capacity equal to the total capacities of all the outgoing edges from that source. Similarly, each sink is connected by an edge to the super sink, and the capacity of that edge is equal to the total capacities of all incoming edges to that sink. We then solve the flow problem on this new graph as normal.

**Problem 3.** Given a list of  $n$  integers  $r_1, r_2, \dots, r_n$  and  $m$  integers  $c_1, c_2, \dots, c_m$ , we wish to determine whether there exists an  $n \times m$  matrix  $X$  consisting of zeros and ones whose row sums are  $r_1, r_2, \dots, r_n$  respectively and whose column sums are  $c_1, c_2, \dots, c_m$  respectively. It is assumed that the sum of the  $r$ 's is equal to the sum of the  $c$ 's (otherwise it is trivially impossible to satisfy the constraints). Describe an algorithm for solving this problem by using a flow network.

### Solution

Construct a flow network as follows: Create two sets of vertices,  $A$  and  $B$ . Each vertex in  $A$  ( $a_1, a_2, \dots, a_n$ ) corresponds to one of the  $n$  rows, and each vertex in  $B$  ( $b_1, b_2, \dots, b_m$ ) corresponds to one of the  $m$  columns. Add a source and a sink. For  $1 \leq i \leq n$ , add an edge from the source to  $a_i$  with capacity  $r_i$ . For each  $1 \leq j \leq m$ , add an edge from vertex  $b_j$  to the sink with capacity  $c_j$ . Add edges  $e_{i,j}$  between every pair of vertices  $(a_i, b_j)$  with capacity 1. Determine the maximum flow in this network. If all the edges from the source are full, then we can construct such a matrix, and moreover, the elements  $X_{i,j}$  of the matrix are equal to the flows along  $e_{i,j}$ . If any edge from the source is not full, such a matrix does not exist.

To see why this method works, consider the flows into and out of the  $a_i$  vertices. We notice that a row has

$r_i$  is if, and only if, the vertex  $a_i$  has a total outflow of  $r_i$  (because each edge out of  $a_i$  corresponds to one of the values in column  $i$ , and all the edges from  $a_i$  have capacity 1), but this is only possible if the flow from the source to  $a_i$  is exactly  $r_i$  because the only edge that flows into  $a_i$  is from the source. These edges have capacity  $r_i$ , so they all need to be full.

After you have studied circulation with demands in the second seminar about network flow, please note that they can be used to model this problem in much simpler terms as there is no need to include additional source and sink nodes (and their outgoing/incoming edges): you simply put the nodes  $a_i$  with demand  $-r_i$ , the nodes  $b_j$  with demand  $c_j$  and solve the circulation with demands to check if there is a feasible solution. The additional nodes, edges and checks in the solution in the previous paragraphs are just implicitly doing the reduction of the circulation with demands problem to the max-flow one.

**Problem 4.** Consider the problem of allocating a set of jobs to one of two supercomputers. Each job must be allocated to exactly one of the two computers. The two computers are configured slightly differently, so for each job, you know how much it will cost on each of the computers. There is an additional constraint. Some of the jobs are related, and it would be preferable, but not required, to run them on the same computer. For each pair of related jobs, you know how much more it will cost if they are run on separate computers. Give an efficient algorithm for determining the optimal way to allocate the jobs to the computers, where your goal is to minimise the total cost.

#### Solution

Create a graph with (bidirectional) edges between related jobs, whose capacities are the cost to separate them. Add edges from  $s$  to all jobs with a capacity equal to the *cost to run on computer 1*, and edges from all jobs to  $t$  with capacity equal to the *cost to run on computer 2*. The minimum cut will be the best total cost since you are paying to either put the job on computer 1 or 2, and paying for related jobs that are cut from each other when assigned to different computers. But be careful: when considering the minimum cut, all the jobs that end up in the  $S$  side of the cut are executed on *computer 2*, while all the jobs that end up in the  $T$  side of the cut are executed on *computer 1*.

**Problem 5.** Consider a variant of network flows in which vertices also have capacities. That is for each vertex except  $s$  and  $t$ , there is a maximum amount of flow that can enter and leave it. Describe a simple transformation that can be made to this problem so that its maximum flow can be solved using an ordinary maximum flow algorithm<sup>1</sup>.

#### Solution

For each non-source non-sink vertex  $v$ , do the following: Create two new vertices  $v_{\text{in}}$  and  $v_{\text{out}}$ . Take all the inflowing edges to  $v$  and replace  $v$  with  $v_{\text{in}}$ . Take all the outflowing edges from  $v$  and replace  $v$  with  $v_{\text{out}}$ . Create an edge from  $v_{\text{in}}$  to  $v_{\text{out}}$  with capacity equal to the capacity of  $v$ . Delete  $v$  from the network.

Now we can solve for the maximum flow in this graph, and the solution will be the maximum flow in the original graph as well.

**Problem 6.** Two paths in a graph are *edge disjoint* if they have no edges in common. Given a directed graph, we would like to determine the maximum number of edge-disjoint paths from vertex  $s$  to vertex  $t$ .

- Describe how to determine the maximum number of edge-disjoint  $s - t$  paths.
- What is the time complexity of this approach?
- How could we modify this approach to find *vertex-disjoint paths* (i.e. paths with no vertices in common)?

<sup>1</sup>Do not try to modify the Ford-Fulkerson algorithm. In general, it is always safer when solving a problem to reduce the problem to another known problem by transforming the input, rather than modifying the algorithm for the related problem.

### Solution

- (a) Given a directed graph  $G$ , create a flow network  $G'$  from  $G$  as follows: Let  $s$  be the source, and  $t$  be the sink. Give every edge a capacity of 1. Now we can run the maximum flow algorithm, and the resulting flow is the number of *edge-disjoint* paths.

This works because the number of *edge-disjoint* paths that pass through a vertex  $v$  is at most  $\min(\text{indegree}(v), \text{outdegree}(v))$ . The maximum flow through a vertex  $v$  is the same value, because all edges are capacity 1.

- (b) The time complexity of Ford-Fulkerson is bounded by  $O(Ef)$  where  $E$  is the number of edges and  $f$  is the maximum flow. Here, the maximum flow is upper-bounded by the outdegree of  $s$ , so the complexity is  $O(E \times \text{outdegree}(s))$  which is bounded by  $O(EV)$ .
- (c) Create the flow network as specified in part a), but add capacities of 1 for each vertex, and then perform the transform described in the solution to problem 5. This ensures that the flow through each vertex is at most 1, whereas before the flow through a vertex was equal to the number of paths through it.

## Supplementary Problems

**Problem 7.** You are a radio station host and are in charge of scheduling the songs for today's program. Every song can be classified as being from a particular era, and a particular genre. Your boss has given you a strict set of requirements that the songs must conform to. No song can be played more than once during the program. Among the songs chosen, for each of the  $n$  eras  $1 \leq i \leq n$ , you must play exactly  $n_i$  songs of that era, and for each of the  $m$  genres  $1 \leq j \leq m$ , you must play exactly  $m_j$  songs from that genre. Of course,  $\sum n_i = \sum m_j$ . Devise an algorithm that given a list of all of the songs you could choose from, their era and their genre, determines a subset of them that satisfies the requirements, or determines that it is not possible.

### Solution

Make a bipartite graph with vertices  $u_1, \dots, u_n$  corresponding to eras, and vertices  $v_1, \dots, v_m$  corresponding to genres. For each combination of era  $i$  and genre  $j$ , add an edge from  $u_i$  to  $v_j$  with a capacity equal to the number of distinct songs that belong to that combination of era and genre (think about how this would change if a certain amount of repetitions of each song were possible). Add a source with outgoing edges to each genre vertex  $u_i$  with capacity  $n_i$  (where  $1 \leq i \leq n$ ). Add a sink vertex, with an edge from each vertex  $v_j$  to the sink with capacity  $m_j$  (where  $1 \leq j \leq m$ ). Run Ford-Fulkerson on this graph, and if a max-flow has value  $\sum n_i = \sum m_j$ , then it is possible to play songs in the required combination. Otherwise it is not. The flow along the edges from left side to the right side will tell you how many songs from a particular combination of era and genre should be played (which you can then easily select without repetition by looking at your input list).

**Problem 8.** Implement the Ford-Fulkerson method using:

- DFS for finding augmenting paths.
- BFS for finding augmenting paths.