CS 630, Fall 2024, Homework 1 Solutions
Due Wednesday, September 18, 2024, 11:59 pm EST, via Gradescope

Homework Guidelines

Collaboration policy Collaboration on homework problems, with the exception of programming assignments and reading quizzes, is permitted, but not encouraged. If you choose to collaborate on some problems, you are allowed to discuss each problem with at most 5 other students currently enrolled in the class. Before working with others on a problem, you should think about it yourself for at least 45 minutes. Finding answers to problems on the Web or from other outside sources (including generative AI tools or anyone not enrolled in the class) is strictly forbidden.

You must write up each problem solution by yourself without assistance, even if you collaborate with others to solve the problem. You must also identify your collaborators. If you did not work with anyone, you should write "Collaborators: none." It is a violation of this policy to submit a problem solution that you cannot orally explain to an instructor or TA.

Typesetting Solutions should be typed and submitted as a PDF file on Gradescope. You may use any program you like to type your solutions. LaTeX, or "Latex", is commonly used for technical writing (`overleaf.com` is a free web-based platform for writing in Latex) since it handles math very well. Word, Google Docs, Markdown or other software are also fine.

Solution guidelines For problems that require you to provide an algorithm, you must provide:

1. pseudocode and, if helpful, a precise description of the algorithm in English. As always, pseudocode should include
   - A clear description of the inputs and outputs
   - Any assumptions you are making about the input (format, for example)
   - Instructions that are clear enough that a classmate who hasn't thought about the problem yet would understand how to turn them into working code. Inputs and outputs of any subroutines should be clear, data structures should be explained, etc.

   If the algorithm is not clear enough for graders to understand easily, it may not be graded.
2. a proof of correctness
3. an analysis of running time and space

You may use algorithms from class as subroutines. You may also use facts that we proved in class.

You should be as clear and concise as possible in your write-up of solutions. A simple, direct analysis is worth more points than a convoluted one, both because it is simpler and less prone to error and because it is easier to read and understand.

Problem 1 ***Database restore*** *(10 points)*

In the heart of a bustling metropolis, you find yourself working as a data analyst at one of the city's most prestigious banks. Each day, hundreds of clients make transactions, and you are tasked with recording every detail a meticulous job to keep the financial wheels turning. But one fateful evening, disaster strikes. A technical glitch sweeps through the bank's system, erasing every last bit of transactional data for the past m days from your records. The once carefully organized files of n clients now lay in digital ruin. Fortunately, despite the chaos, a glimmer of hope emerges. Thanks to a clever data-preserving mechanism, not all is lost. For each client, youve managed to salvage the sum total of their transactions over the entire period. Additionally, you have one more crucial piece of information: for each of the m days, you possess the grand total of all transactions that took place across all clients. Armed with these fragments, your task is clear but challenging: to restore the detailed transaction data for each client on every day, breathing life back into the banks records and restoring order from the brink of disaster. Can you untangle this web and retrieve the missing data, client by client, day by day?

As *input* you are given a table C for the clients, such that $C[i]$ contains the total Dollar amount of transactions for client i. We also get a table D, such that $D[j]$ is the total amount on day j.

As *output* your algorithm should return a table M, such that $M[i][j]$ contains the transaction amount of client i on day j or return that there doesn't exist such a database.

Solution.

We model the problem as following. The database is represented as a two dimensional array $M[\cdot,\cdot]$ where in the i,j entry we store the data of the *i-th* client at the *j-th*. The sum of rows (total transactions of a client) are stored in a table C, such that $C[i]$ is the sum of values in row i and the sum of columns (total transactions of a given day) at the table D where $D[j]$ is the sum of values in column j.

We create a network flow graph to model this problem. G has a source s and sink t. Further we have a node c_i for each row and a node d_j for each column. We add edges from s to the rows. The edge (s, c_i) has capacity $C[i]$. We also add edges from d_j to t, the capacity of (d_j, t) is $D[j]$. Finally we add a directed edge with infinite capacity from every row to every column. (We can use a large number, such as $\sum C[i]$ instead.)

The amount of flow on the edges between (c_i, d_j) correspond to the values in M. Due to the edge capacity $c(s, c_i) = C[i]$ we know that if (s, c_i) is saturated by the max flow, then $C[i]$ amount of flow is distributed to the columns that c_i is connected to. Same way, due to $c(d_j, t) = D[j]$ we know that if all edges into t are saturated then the amount of flow associated with each column is also the desired value.

After finding the maximum flow, we create table M. The value $M[i][j]$ is equal to the amount of flow on edge (c_i, d_j).

Algorithm 1: Table(C, D)

```
   /* create graph                                                          */
 1 if ∑_i C[i] ≠ ∑_j D[j] then
 2     return row and column sums not equal./* impossible valid assignment */
 3 G ← empty (nested) hash table;
 4 G[s] ←empty hash table;
 5 G[t] ← empty hash table;
 6 for j in D do
 7     G[j][t] ← D[j];
 8 for i in C do
 9     G[s][i] ← C[i];
10     for j in D do
11         G[i][j] ← ∞;
   /* find max flow                                                         */
12 F ← FordFulkerson(G, s, t)/* F[u][v] = max flow on edge (u, v)          */
   /* decide whether solution is valid based on value of ma flow           */
13 if sum_i F[s][i] ≠ ∑_i C[i] then
14     return no valid assignment/* edges out of s not saturated           */
15 M ← (n × m)table;
16 for i in C do
17     for j in D do
18         M[i][j] ← F[i][j];
19 return M
```

runtime and space: Graph G has $n + m + 2 = O(n + m)$ nodes. and $n + m + nm = O(nm)$ edges.

This algorithm has multiple nested for loops with $n \times m$ iterations. Within each loop we have $O(1)$ operations. This yields $O(nm)$. The value of the max flow in this graph is $W = \sum_i C[i]$, which is also the sum of capacities out of s. Hence the runtime of FF is $O(Wnm)$. Overall, the runtime is $O(Wnm)$.

correctness: the edge capacities between s and the rows, and t and the columns ensure that the flow through each node doesn't exceed the prescribed row/col sum. The flow on the edges (c_i, d_j) correspond to the number in $M[i][j]$ that contributes to the row sum $R[i]$ and col sum $D[i]$. If the max flow saturates the edges from s or to t that means that we are able to distribute the row/col sum amongst the table entries corresponding to this index. If the max flow doesn't saturate one of these edges that implies that the sums can't be distributed.

Problem 2 ***Essential edges*** *(10 points)*

We are given a network flow graph $G(V, E, s, t, c)$ and a maximum flow f on this network. That is, for each edge (u, v) we are given the amount of flow $f(u, v)$.

1. (*not to be handed in*) As a warm up, think of it how we could verify that the given flow is indeed a max flow. That is, it can't be increased anymore. One way is to run Ford-Fulkerson from scratch and compare the value of the flow. We can do it in fact much faster, in $O(|V| + |E|)$ time. What is this faster approach?

   Solution. If the given flow f is not maximum, that means that there is an augmenting path along which the flow can be increased. We can find this path by running one more iteration of FF. That is,

run BFS/DFS from s on the residual graph. If t is not reachable, then we can conclude that the flow is indeed maximal.

2. Now we know for a fact that f is indeed a maximum flow. Let (x, y) be a specific edge, and suppose that we decrease the capacity of this edge by one. That is, if the current capacity is $c(x, y)$, then the new capacity is $c'(x, y) = c(x, y) - 1$. Given $value(f)$ in the original flow network, describe in words what values the maximum flow can take in the decreased graph. Also explain what property the edge (x, y) has for the different max flow values.

   Solution. The max flow in the decreased graph is either $value_{new}(f) = value(f)$ or $value_{new}(f) = value(f) - 1$. The new flow is decreased iff (x, y) is an edge in a min-cut.

3. Design an algorithm that given $G(V, E, s, t, c)$, the max flow f and the edge (x, y) as input computes the max flow f' in the decreased graph. (Don't confuse the max flow f' with the $value(f')$, the former is a function assigning values to each edge, the latter is a single number.) Your algorithm should run faster than rerunning Ford-Fulkerson from scratch. For full credit it should run in $O(|V| + |E|)$.

   Solution. First note that the flow only needs to be updated if $f(x, y) = c(x, y)$, otherwise the decreased capacity won't affect the original flow.

   algorithm: The new max flow f' will carry one less flow on edge (x, y). The key insight is that we can't just reduce $f(x, y)$ in isolation as that would violate flow conservation; there would be excess flow in x and not enough in y. To fix this, in our implementation this we need to update the flow along an entire path with flow that contains edge (x, y). This is done in lines (lines 3-7). We reduce the flow along these edges (lines 8-9). Finally, we run an additional iteration of FF to see whether we can find an alternate route to send flow (line 10).

Algorithm 2: DecreaseFlow($G(V, E, s, t, c), f, (x, y)$)

```
1  if f(x,y) < c(x,y) then
2      return f
3  G_f ← residual graph for G and f;
4  f' ← f /* initialize the new flow equal to the old flow.                    */
5  p_1 ← modBFS(s, x, G_f, f) /* Path from s to x with non-zero edgeges.  Modify BFS/DFS by
       only considering edges where f(u,v) > 0.                                */
6  p_2 ← modBFS(y, t, G_f, f) /* path from y to t only considering edges f(u,v) > 0.   */
7  p ← p_1 + (x,y) + p_2 /* create path p by appending path p1, edge (x,y) and p2   */
8  for (u,v) in p do
9      f'(u,v) = f(u,v) − 1 /* decrease flow along path                          */
10 f' ← partialFF(G, s, t, c, f') /* run FF starting from the residual defined by f'   */
11 return f'
```

runtime: The running time of this algorithm comes from multiple runs of BFS, which we know is $O(|V| + |E|)$ for $G(_f)$ which is the same as for G. The fact that we ignore non-zero edges doesn't increase or decrease the runtime. In the call to partialFF( ) we know that there is at most one iteration left as the max flow in the reduced graph can be increased by at most 1. Hence, this amounts to an additional call to BFS/DFS and also takes $O(|V| + |E|)$.

correctness: We have to prove two things (1.) the returned f' is a valid flow. (2.) f' is a max flow.

(1.) The fact that we reduced the flow by 1 along a path means that for any node v both the incoming and outgoing flow was reduced by 1. Since initially f was a valid flow and the values were changed by the same amount, flow conservation is maintained.

(2.) If f' (directly after the decrease) is not maximum, that means that there is an augmenting path in $G_{f'}$, which we have proven in class that FF can find. We know that f is a max flow. Since we didn't increase any capacity in G we know that the max flow cannot have higher value than $value(f)$ after decreasing $c(x, y)$. As a result running one iteration of FF is enough as f' can increased by at most 1 unit.