CSC373 Fall'19
Assignment 2 Solutions

Due Date: Nov. 1, 2019, by 3pm

Q1 [25 Points] Fixing Pictures

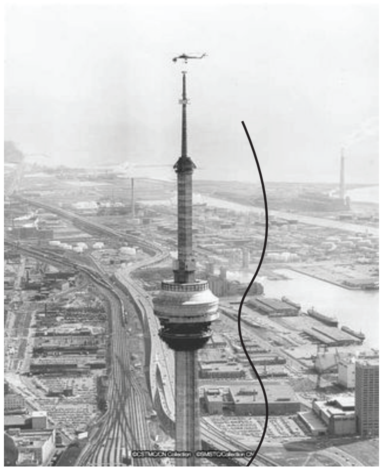

Figure 1: Scanned picture with a black dog hair.

You have scanned some old black-and-white photographs at home. Unfortunately a dog hair on the scanner glass has corrupted your pictures. You want to correct the picture by removing the dog hair. But first, you need to find which pixels (more accurately, chain of pixels) represent the dog hair.

Your input is a picture P of $m \times n$ pixels. You are given the intensity $P(i, j)$ of each pixel (i, j), which is a value of grey between 0 (black) and 1 (white). The hair is long, blackish, and at most one pixel wide. You may assume that the hair only ever passes through at most one pixel in any row of pixels. The figure above shows a hair on an old picture of the CN tower.

The likelihood that pixel (i, j) is part of the hair is given by

$$\ell(i,j) = \begin{cases} 0 & \text{if } P(i,j) \geq h, \\ 1 - \frac{P(i,j)}{h} & \text{if } P(i,j) < h, \end{cases}$$

where $h \in (0, 1)$ is a given threshold. The idea is that any pixel lighter than h is definitely not part of the hair, and pixels darker than h have more probability of being part of the hair the darker they are.

Your goal is to find a chain of pixels C. A chain is a set of pixels $\{(r, p_r) : r \in [i, j]\}$ containing one pixel from every row between row i and row j such that for $r \in (i, j)$, pixel (r, p_r) is a neighbour of pixels $(r-1, p_{r-1})$ and $(r+1, p_{r+1})$. Here, we are using the standard notion of "neighbourhood" under which each pixel has at most eight neighbours (up, down, left, right, northeast, northwest, southeast, southwest). The likelihood of a chain being the hair is the sum of likelihoods of its pixels being part of the hair: $\ell(C) = \sum_{(i,j)\in C} \ell(i,j)$.

(a) [17 Points] Suppose you want to compute the maximum likelihood of any chain being the hair. Use dynamic programming to design an algorithm for this.

- [3 Points] Clearly define the quantity that your DP is computing. (For example, in the shortest path DP that we did in class, we defined: "Let $OPT(t, i)$ be the length of the shortest path from s to t using at most i edges.")
- [5 Points] Write a Bellman equation for this quantity, and briefly reason why your equation is correct.
- [3 Points] Write the initial conditions.
- [3 Points] What is the running time and space complexity of your algorithm (say, in a top-down implementation)?
- [3 Points] In what order would you compute the quantities in a bottom-up implementation? Briefly reason why this order works.

Solution:

Assume the pixels are (r, c) for $r \in \{1, \ldots, m\}, c \in \{1, \ldots, n\}$, where we interpret the row as going down when r increases, and the column as going right when c increases.

Definition: Let $S(r, c)$ be the maximum likelihood of any chain starting from row 1 (any column) and ending at (r, c).

Bellman equation with initial conditions:

$$S(r,c) = \begin{cases} -\infty, & \text{if } r < 1,\ r > m,\ c < 1, \text{ or } c > n, \#\text{ any undefined pixel} \\ \ell(1,c), & \text{if } r = 1, c \in \{1, \ldots, n\}, \\ \ell(r,c) + \max_{c' \in \{c-1,c,c+1\}} S(r-1, c'), & \text{otherwise.} \end{cases}$$

Justification: For $r = 1$, it is clear that $S(1, c) = \ell(1, c)$ for any c. Any chain ending at (r, c) (where $r > 1$) must pass through $(r-1, c')$ for some $c' \in \{c-1, c, c+1\}$. Further, by the optimal subsolution property, the chain up to $(r-1, c')$ must be the highest likelihood chain ending at $(r-1, c')$. This is precisely what the Bellman equation captures. We set $S(r, c) = -\infty$ when queried on any undefined pixel so that it will always get discarded by the maximum in the third condition. At the end, the algorithm can return $\max_c S(m, c)$, which is the highest likelihood of any chain ending in the last row. Since likelihoods never decrease by adding pixel, this should give the highest likelihood of any chain in the entire matrix.

Top-down running time & space: The runtime is $O(mn)$ since there are $O(mn)$ values to compute, and computing each value takes $O(1)$ operations. The space complexity will be $O(mn)$

to save all the values in a top-down implementation.

Bottom-up implementation: For r going from 1 to m, for c going from 1 to n, compute $S(r, c)$. Note that when computing row r, we only need row $r - 1$, which would already be computed in this order. Hence, this order works. In fact, by only storing one previous and one current row, this reduces the space complexity to $O(n)$.

(b) [5 Points] Suppose now that you actually want to find the chain with the maximum likelihood of being the hair. If there are multiple such chains, you want to return the maximum likelihood chain with the fewest number of pixels. Modify your DP so that it allows you to return the desired chain.

Solution:

At each pixel (r, c), along with $S(r, c)$ as above, we also want to store:

- $L(r, c)$ = the smallest length of any chain ending at (r, c) with likelihood $S(r, c)$.
- $P(r, c)$ = column in the previous row where this chain comes from (i.e. $c-1$, c, or $c+1$). We use $\perp$ when there is no previous row or the cell is invalid.

Bellman equation with initial conditions:

$$(S(r,c), L(r,c), P(r,c)) = \begin{cases} (-\infty, \infty, \perp), & \text{if } r < 1,\ r > m,\ c < 1, \text{ or } c > n, \#\text{ any undefined pixel} \\ (\ell(1,c), 1, \perp), & \text{if } r = 1, c \in \{1, \dots, n\}, \\ (\ell(r,c) + S(r-1,c'), 1 + L(r-1,c'), c'), & \text{otherwise, where } c' \in \{c-1, c, c+1\} \text{ maximizes } S(r-1,c'), \\ & \text{and among all such maximizers, minimizes } L(r-1,c'). \end{cases}$$

Justification and extraction of shortest maximum likelihood chain:

In the first case, it does not matter what we set as $L(r, c)$ and $P(r, c)$ because these are invalid cells with $-\infty$ likelihood and will never be used.

In the second case, we are in the first row, so the length of the chain so far is $L(r, c) = 1$ and there is no previous row, so $P(r, c) = \perp$.

In the final case, we choose the c' from the previous row which leads to highest likelihood (this is our first objective). If there are multiple such c', then we break ties by the one which achieves this highest likelihood with a shorter chain.

Finally, to extract the desired chain, we iterate across all (r, c) with the highest $S(r, c)$, and among them, choose one with the lowest $L(r, c)$. This is the last pixel in our chain. Then, we iteratively replace $c \leftarrow P(r, c)$ and $r \leftarrow r - 1$ to find previous pixels until we hit $P(r, c) = \perp$.

(c) [3 Points] (Open Ended Question) Is the likelihood function ($\ell(i, j)$ and $\ell(C)$) well-designed in your opinion? Describe a type of image on which this algorithm would perform poorly for the ultimate objective of finding the hair. Suggest a modification to the likelihood formula that you think might perform better.

Solution:

Note: Almost any solution that proposes some improvement is a valid solution to this. Here is a sample solution.
Suppose the hair is all exactly black, and the rest of the picture is all exactly white, except for one stray pixel far away from the hair with likelihood 0.001. Then the algorithm might output a chain that includes the real hair, but might extend the hair all the way to that remote pixel.

To fix this, allow ℓ to take on negative values, e.g. define

$$\ell(i,j) = \begin{cases} -0.01 & \text{if } P(i,j) \geq h, \\ 1 - \frac{1.01 \cdot P(i,j)}{h} & \text{if } P(i,j) < h. \end{cases}$$

Q2 [20 Points] Approximating Functions

You are given $n+1$ points $\{p_0, p_1, .., p_n\}$, where $p_i = (x_i, y_i)$ for $0 \leq i \leq n$. The points are sorted in the increasing order of (unique) x_i's: that is, $x_0 < x_1 < \ldots < x_n$.

In reality, these are sample points on the curve of some unknown function $f : \mathbb{R} \to \mathbb{R}$ with $y_i = f(x_i)$ for each $0 \leq i \leq n$. Your goal is to approximate this function using a sequence of line segments.

If you use the sequence of n line segments joining all adjacent points (i.e. joining p_i and p_{i+1} for $0 \leq i < n$), then this provides an excellent approximation since it passes through (x_i, y_i) for each $0 \leq i \leq n$. However, it uses too many line segments.

If you replace the line segments between p_i and p_j (where $j > i$) with a single straight line connecting p_i and p_j, then the error introduced is given by

$$E[i,j] = \sum_{\ell=i+1}^{j-1} |y_\ell - \hat{y}_\ell|,$$

where $(x_\ell, \hat{y}_\ell)$ is point through which the line segment joining p_i and p_j passes. Note that $E[i, i+1] = 0$ for all i.

Suppose every line segments costs C. If you select a subset of points (without changing their order) $(p_{i_0}, \ldots, p_{i_k})$ and use the k line segments connecting adjacent points p_{i_m} and $p_{i_{m+1}}$ for $0 \leq m < k$, the total error of this approximation is $\sum_{m=0}^{k-1} E[i_m, i_{m+1}] + k \cdot C$. Your goal is to find the optimal approximation which minimizes this error. Note that k is not given to you. You need to optimize the number of line segments to use too. See Figure for an illustration of the problem.

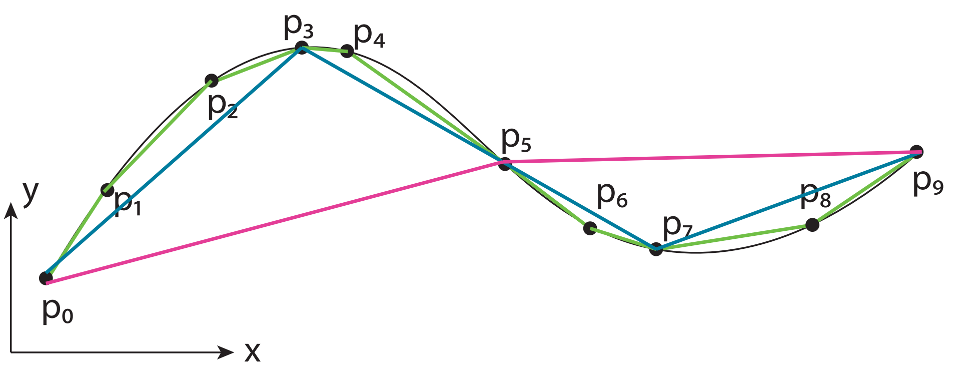

Figure 2: The function (black curve) with samples $p_0, \ldots, p_9$ is shown approximated using 9 (green), 3 (blue), and 2 (pink) line segments. The optimal approximation will try to use fewer line segments but also try to achieve a low approximation error.

(a) [3 Points] Express the total error of an approximation $\{p_{i_0}, \ldots, p_{i_k}\}$ explicitly in terms of the inputs to the problem $((x_i, y_i)$ for $0 \le i \le n)$.

Solution:

Denote $p_{i_m} = (x_{i_m}, y_{i_m})$. Then, the total error of approximation $(p_{i_0}, \ldots, p_{i_k})$ is

$$\begin{aligned}
&k \cdot C + \sum_{m=0}^{k-1} E[i_m, i_{m+1}] \\
&= k \cdot C + \sum_{m=0}^{k-1} \sum_{\ell=i_m}^{i_{m+1}-1} |y_\ell - \hat{y}_\ell| \\
&= k \cdot C + \sum_{m=0}^{k-1} \sum_{\ell=i_m}^{i_{m+1}-1} \left| y_\ell - \frac{y_{i_{m+1}} - y_{i_m}}{x_{i_{m+1}} - x_{i_m}} \cdot (x_\ell - x_{i_m}) \right|
\end{aligned}$$

(b) [17 Points] Describe a dynamic programming solution that finds the optimal subset of points $\{p_{i_0}, p_{i_1}, .., p_{i_k}\}$ minimizing the total error.

- [3 Points] Clearly define the quantity or quantities that your DP is computing.
- [5 Points] Write a Bellman equation for these quantities, and briefly reason why your equation is correct.
- [3 Points] Write the initial conditions.
- [3 Points] What is the running time and space complexity of your algorithm (say, in a top-down implementation)? Assume that all $E[i, j]$ are pre-computed and you have constant-time access to them.

- [3 Points] In what order would you compute the quantities in a bottom-up implementation? Briefly reason why this order works.

Solution:

Definition: Let $F[\ell]$ denote the minimum error in approximating $p_0, \ldots, p_\ell$. Let $L[\ell]$ denote the starting point of the last line in this approximation (the last line must end at p_ℓ).
Bellman equation & justification: Due to the optimal substructure property, we know that $F[\ell]$ must equal to $F[j] + E[j, \ell] + C$ for $j \in \{0, \ldots, \ell - 1\}$ that minimizes this quantity; further, this is the j that we want to set as $L[\ell]$ by definition. The base case is $\ell = 0$, in which case $F[\ell] = 0$ and we set $L[\ell] = \perp$ (there is no previous point). The following Bellman equation achieves this.

$$(F[\ell], L[\ell]) =$$
$$\begin{cases} (0, \perp), & \text{if } \ell = 0, \\ (F[j] + E[j, \ell] + C, j), & \text{if } \ell > 0, \text{ where } j \in \{0, \ldots, \ell - 1\} \text{ minimizes } F[j] + E[j, \ell] + C. \end{cases}$$

The minimum error is given by $F[n]$. To construct the optimal solution, we know that the last point must be p_n. We start with $\ell = n$, and iteratively find the previous point in the optimal approximation $p_{L[\ell]}$ and set $\ell \leftarrow L[\ell]$ until we hit $L[\ell] = \perp$.
Top-down running time and space complexity: There are $O(n)$ values of F and L to be computed. Hence, the space complexity is $O(n)$. Assuming other F and L values are available, each such value can be computed in $O(n)$ time by iterating over possible j's. Hence, the running time is $O(n^2)$.
Bottom-up implementation: We should compute this in the increasing order of ℓ since the computation for ℓ requires knowing answers for all $j < \ell$. This has the same space and runtime complexity as top-down implementation.

Q3 [30 Points] Maximum Flow

Suppose you are given a network N, a *maximum* flow f on N, and one edge $e_0 \in N$ such that $f(e_0) = c(e_0)$. You are also given that f has *integer* flow values $f(e)$ for all edges e.

(a) [5 Points] If we decrease $c(e_0)$ by 1 (i.e., let $c'(e_0) = c(e_0) - 1$ but other capacities remain the same), then one might expect that the maximum flow on N might also decrease by one unit. But does this always happen?

Either give a specific example where the maximum flow on N does not change (and show that this is the case), or give a general argument that the maximum flow on N always changes.
Solution:

Consider the network N below with the flow f indicated in the diagram, and let $e_0 = (s, b)$. Reducing the capacity of e_0 to 1 does not decrease the maximum flow because additional flow can be routed through (s, a) and (a, b).

$$\begin{tikzcd}
s \arrow[rr, "2/2"] \arrow[dr, "0/2"'] & & b \arrow[r, "2/2"] & t \\
& a \arrow[ur, "0/2"'] & &
\end{tikzcd}$$

(b) [10 Points] Irrespective of your answer on the previous part, there are cases when this change in capacity causes the maximum flow to decrease.

Give an efficient algorithm that takes a network N, a maximum integral flow f on N, and one edge $e_0 \in N$ such that $f(e_0) = c(e_0)$ and that determines a new maximum flow in the network N', where $N' = N$ except for $c'(e_0) = c(e_0) - 1$.

Include a brief justification that your algorithm is correct and a brief analysis of your algorithm's worst-case runtime (which should be as small as possible).

Solution:

Let $e_0 = (a, b)$. Intuitively, we want to find a "reducing path" from s to a and another "reducing path" from b to t, to rebalance the flow in the network after it has been reduced by 1 unit through edge e_0. Then, we can search for an augmenting path in N', in case there is a way to re-route the flow.

There is one technical twist we must deal with. In general, all flow coming into a node a can be traced back to the source s, and similarly, all flow coming out of a node b can be followed forward to the target t. However, it is possible for a network to contain a directed cycle of edges where "phantom flow" is circulating: a certain amount of flow that simply goes around the cycle, without contributing anything to the overall flow value. For example, $a \to b \to c$ is such a cycle in the network below, with 3 units of "phantom flow:"

$$\begin{tikzcd}
s \arrow[r, "2/2"] & a \arrow[rr, "5/5"] & & b \arrow[r, "2/2"] \arrow[dl, "3/3"] & t \\
& & c \arrow[ul, "3/3"] & &
\end{tikzcd}$$

This is the reason for the first if-statement in the algorithm below.

```
Given $N$, $f$, and $e_0 = (a, b)$:
    Set $f'(a, b) = f(a, b) - 1$.
    Run BFS starting from $a$, using only backward edges $(x, y)$ with $f(x, y) > 0$ or
        forward edges $(y, x)$ with $f(y, x) < c(y, x)$ — (looking for a "reducing path").
    If $b$ is reached:   # we have found a cycle that contains $(a, b)$
        Adjust the flow on every edge from $a$ to $b$:
            Set $f'(x, y) = f(x, y) - 1$ for every backward edge $(x, y)$ with $f(x, y) > 0$.
            Set $f'(y, x) = f(y, x) + 1$ for every forward edge $(y, x)$ with $f(y, x) < c(y, x)$.
    Else:   # there is no cycle that contains $(a, b)$
        # There must be a path from $a$ to $s$.
        Adjust the flow on every edge from $a$ back to $s$:
            Set $f'(x, y) = f(x, y) - 1$ for every backward edge $(x, y)$ with $f(x, y) > 0$.
            Set $f'(y, x) = f(y, x) + 1$ for every forward edge $(y, x)$ with $f(y, x) < c(y, x)$.
        # There must also be a "reducing path" from $b$ forward to $t$.
        Run BFS starting from $b$, using only forward edges $(x, y)$ with $f(x, y) > 0$ or
            backward edges $(y, x)$ with $f(y, x) < c(y, x)$ — (looking for a "reducing path").
```

```
Adjust the flow on every edge from $b$ to $t$:
    Set $f'(x,y) = f(x,y) - 1$ for every forward edge $(x,y)$ with $f(x,y) > 0$.
    Set $f'(y,x) = f(y,x) + 1$ for every backward edge $(y,x)$ with $f(y,x) < c(y,x)$.
# Now, check if the flow can be re-routed.
Run BFS on $N'$ to find an augmenting path, and if one is found, do the augmentation.
```

The correctness of the algorithm follows from the reasoning that precedes the algorithm.

The algorithm runs in worst-case time $\Theta(n+m)$ (linear in the size of the network): BFS takes linear time in the size of the input graph and is executed at most three times, and doing each reduction and augmentation also takes linear time (traversing a single path).

(c) [5 Points] If we increase $c(e_0)$ by 1 (i.e., let $c'(e_0) = c(e_0) + 1$ but other capacities remain the same), then one might expect that the maximum flow on N might also increase by one unit. But does this always happen?

Either give a specific example where the maximum flow on N does not change (and show that this is the case), or give a general argument that the maximum flow on N always changes.

Solution:

Consider the network $N = s \xrightarrow{2/2} a \xrightarrow{2/2} t$ with the flow f indicated in the diagram, and let $e_0 = (s,a)$. Increasing the capacity of e_0 to 3 does not increase the maximum flow because no additional flow can be pushed on edge (a,t).

(d) [10 Points] Irrespective of your answer on the previous part, there are cases when this change in capacity causes the maximum flow to increase.

Give an efficient algorithm that takes a network N, a maximum integral flow f on N, and one edge $e_0 \in N$ such that $f(e_0) = c(e_0)$ and that determines a new maximum flow in the network N', where $N' = N$ except for $c'(e_0) = c(e_0) + 1$.

Include a brief justification that your algorithm is correct and a brief analysis of your algorithm's worst-case runtime (which should be as small as possible).

Solution:

Let us state the facts. We have a maximum flow in network N. This means that N contains no augmenting path (by the Ford-Fulkerson Theorem). The only way for the maximum flow to increase in N' is to have an augmenting path in N'. And the only difference between N' and N is the increase in the capacity of edge e_0.

So we can find a new maximum flow by looking for an augmenting path that uses edge e_0. If there are no such paths, then we know that there is no augmenting path in N' and flow f is already maximum. If there is one augmenting path, we can augment the flow along this path. But then we are done: there can be no other augmenting path because edge e_0 will be back at full capacity and everything else is the same as in N.

```
Given $N$, $f$, and $e_0$:
    Run BFS on $N'$ to find an augmenting path.
    If we find one augmenting path $P$:   # $P$ will contain $e_0$
        Augment $f$ along $P$.
    # Else: do nothing—there is no other augmenting path in $N$.
```

The correctness of the algorithm follows from the reasoning that precedes the algorithm.

The algorithm runs in worst-case time $\Theta(n + m)$ (linear in the size of the network): BFS takes linear time in the size of the input graph, and doing one augmentation also takes linear time.

BONUS QUESTION

Q4 [15 Points] Bloober (Remember: 20% rule does not apply to this bonus question)

A start-up in the competitive self-driving bus ride sharing space, Bloober, would like to figure out the minimum number of buses it needs in its fleet to service the n routes: each route r_i departs from bus stop $s(r_i)$ at time $d(r_i)$ and ends at stop $e(r_i)$. The travel time between any two bus stops a and b is $t(a, b)$.

Note that we only care about the start stop, the end stop, and the travel time $t(s(r_i), e(r_i))$ of going from the start stop to the end stop. We do not care about any stops in between.

A bus that is not in use (after it has reached the end stop of its current route) can be used to service another route if it can reach the start stop of that route by the scheduled departure time of that route.

(a) [8 Points] Design an algorithm that can test if Bloober can service the routes using exactly k buses. You get 4 marks for describing the algorithm, and 4 marks for proving its correctness.

You can assume that each bus can reach the start stop of the first route that it services by the departure time of that route, and once a bus reaches a stop, it is allowed to just stay there for a while. Note thus that you can trivially service n routes with n buses. The goal is clearly to service them with a smaller number of buses by reusing buses to service multiple routes where the travel times are compatible.

Solution:

The problem can be reduced to circulation with lower bounds. Construct a directed graph $G = (V, E)$ as follows.

- For each route r_i, add two vertices $s(r_i)$ and $e(r_i)$ corresponding to the start and end bus stop of the route. If a bus stop shows up multiple times as start/end stop of multiple routes, use fresh vertex for it every time.

- Add edge $(s(r_i), e(r_i))$ with lower bound and capacity both 1 (in other words, exactly one bus is needed on this route).
- For each pair of routes r_i and r_j, if $d(r_i) + t(s(r_i), e(r_i)) + t(e(r_i), s(r_j)) \leq d(r_j)$, then add an edge $(e(r_i), s(r_j))$, with lower bound 0 and capacity 1. This allows (but does not require) a bus to travel from the end of route r_i in time to service route r_j.
- Create a source b_s and sink node b_t, to serve and collect the buses circulating in the network. For every route r_i, add edges $(b_s, s(r_i))$ and $(e(r_i), b_t)$ with lower bound 0 and capacity 1. This means every route can get a new bus directly from source, or send the bus servicing it directly to the sink afterwards.
- The source and sink have demand $-k$ and k respectively (all other nodes have demand 0). This imposes the condition that *exactly* k buses must leave the source and be collected at the sink.

Correctness: We now show that the routes can be serviced using exactly k buses if and only if there is a valid circulation in the graph constructed above.

Feasible schedule $\Rightarrow$ Valid circulation: Suppose there is a schedule for servicing the routes using exactly k buses. Construct a flow as follows: for each bus j servicing routes $q_1^j, q_2^j, \ldots, q_l^j$ in that order, send 1 unit of flow along the path $b_s \to s(q_1^j) \to e(q_1^j) \to \ldots \to s(q_l^j) \to e(q_l^j) \to b_t$. This results in a supply and demand of k units (one for each of the k buses) at the source and sink, respectively, and 0 demand at all other nodes. The flow on each route edge is exactly 1, as each route is serviced by exactly one bus in the given schedule. We thus have a valid circulation.

Valid circulation $\Rightarrow$ Feasible schedule: If there is a valid circulation, then there is an integer valid circulation. In this circulation, exactly k units of flow leaves b_s and enters b_t, and each edge carries either 0 or 1 unit of flow. Using the trick learned in class for edge-disjoint paths application of network flow, we can "follow" each unit of flow and create k edge-disjoint paths from b_s to b_t. Each path must have a start node right after source, the end node of a route right after the start node of that route, and the sink only after an end node due to our graph construction. Hence, this gives a valid schedule for a bus. Thus, from a valid integral circulation, we can generate a valid schedule for servicing all routes using exactly k buses.

(b) [4 Points] What is the time complexity of your algorithm, assuming the naïve implementation of the Ford Fulkerson algorithm that runs in $O(mnC)$ time, where m, n, and C are the number of edges, the number of vertices, and the sum of capacities of edges from the source vertex, respectively.

Solution:

Our graph has $O(n)$ vertices (there are n routes) and $O(n^2)$ edges (there is possibly an edge for every pair of routes). The capacity C is k, where $k \leq n$ (since the routes can clearly be serviced using n or more buses). By following our reduction from class (circulation with lower bounds $\to$ circulation $\to$ network flow), we see that this amounts to solving a network flow problem on $O(n)$ vertices and $O(n^2)$ edges, where the sum of capacities of outgoing edges from source is $O(n)$. Using

Ford-Fulkerson, this can be solved in time $O(n^2 \cdot n \cdot n) = O(n^4)$.

(c) [3 Points] Use the solution to part (a) to determine the minimum number of buses needed in the fleet. What is the time complexity of the overall algorithm?

Solution:

To find the minimum k, we can perform a binary search over $k \in \{1, \dots, n\}$ in the algorithm from part (a). This takes $O(n^4 \log n)$ time.