Homework 7:

Due: October 28, 2025 at 2:30p.m.

This homework must be typed in LaTeX and submitted via Gradescope.

Please ensure that your solutions are complete, concise, and communicated clearly. Use full sentences and plan your presentation before your write. Except where indicated, consider every problem as asking for a proof.

Problem 1. Given a flow network graph G with n nodes and m edges, assume you are given a maximum flow assigment. Propose an algorithm that finds a minimum capacity cut in time $O(m)$.

Solution. **Algorithm** Starting from the source node, run a modified version DFS/BFS marking every node that is seen. Traverse only those edges with positive residual capacity. If traversing an edge in the forward direction then $RC = c(e) - f(e)$. If traversing an edge in the backwards direction then $RC = f(e)$. We stop a DFS/BFS branch upon reaching a node whose forward edges all have residual capacity zero. Let V_s denote the set of all marked vertices. The algorithm returns $(V_s, V \setminus V_s)$.
Correctness To show that the algorithm is correct we note the following fact:
Let $G = (V, E)$ be a network with max flow. Let V_s and V_t be a partition of V such that for every edge e going from V_s to V_t we have that $RC(e) = 0$. Then, (V_s, V_t) is a min-cut of G.
Since (V_s, V_t) is a cut we know that the flow through (V_s, V_t) is equal to the flow through the network, and thus is the max-flow f^* of the network. For every edge e between from V_s to V_t we have that

$$RC(e) = 0 = c(e) - f(e) \implies c(e) = f(e)$$

Let $\mathcal{X}$ denote the set of all edges from V_s to V_t. We have that

$$c(\mathcal{X}) = \sum_{e \in \mathcal{X}} c(e) = \sum_{e \in \mathcal{X}} f(e) = f^*$$

By the min-cut max-flow theorem, since the capacity of $\mathcal{X}$ is equal to the max-flow of the network, $\mathcal{X}$ is a min-cut of the network.
Our proposed algorithm returns two sets V_s and $V \setminus V_s$ such that all forward edges from V_s to $V \setminus V_s$ have residual capacity 0. By the above statement, we have that $(V, V \setminus V_s)$ is a min cut.
Runtime The runtime of our algorithm is given simply by the running time of DFS/BFS since we can create our two output sets by initially starting with two sets, an empty one and the set of a vertices, and moving elements from the latter into the former. This can be done in constant time for a given iteration. The running time of DFS/BFS is $O(n + m)$ where n is the number of nodes and m is the number of edges. Note, however, our since our network is connected that the number of vertices in the graph is $O(m)$. Thus, we can reduce our overall running time to $O(m)$. □

Problem 2. Karp-Rabin patern matching algorithm is particularly useful in multidimensional pattern matching.
Let T be an array of $n \times m$ characters, and let P be a pattern of $p \times q$ characters, $p << n$ and $q << m$.
We hash a pattern of $p \times q$ characters as follows: We first hash each row $1 \leq i \leq p$ (which is a string of q characters) to a value $h(i)$. We then hash the string of p values $(h(1), \ldots, h(p))$ to one value $h_2(h(1), \ldots, h(p))$.
We use a "rolling hash" function for both the row and column hashing.
To search for pattern P in T we compute the h_2 value of each $p \times q$ block by applying the rolling hash function h to each row, and then the rolling hash function h_2 to compute the h_2 value of the block.

- Write a (pseudo code) program that implements this search with no more than $O(n)$ extra memory.
- Show that the time complexity of your program is $O(nm)$. (You can ignore the cost of verifying patterns found by the hashing process.)

Solution. The main idea of the algorithm is that we'll use two rolling hashes: h for rows (horizontal) and h_2 for stacks of rows (vertical). After hashing our pattern, we will compute each $p \times q$ block of T by iterating vertically in the inner loop and iterating horizontally in the outer loop. The pseudocode is provided below:

Algorithm 1 2DKarpRabin(T, P, h, h_2)

```
Compute row hashes $h_P[r] \leftarrow h(P[r][0, \ldots, q-1])$ for $r = 0, \ldots, p-1$
targetH2 $\leftarrow h_2(h_P[0], \ldots, h_P[p-1])$
Compute $n$ row hashes h_T$[k] \leftarrow h(T[k][0, \ldots, q-1])$ for $k = 0, \ldots, n-1$
for $i = 0$ to $m - q$ do
    curH2 $\leftarrow h_2($h_T$[0], \ldots,$ h_T$[p-1])$
    for $j = 0$ to $n - p$ do
        if curH2 $=$ targetH2 then
            verify and report match at $(j, i)$
        if $j < n - p$ then
            slide curH2 vertically by rolling on h_T$[j+p]$
    if $i < n - p$ then
        slide all $n$ row hashes in $h_T$ horizontally
return no match
```

The idea behind storing the leftmost n hashes is so that we can roll them all horizontally after finishing our inner loop of vertical rolling. This is $O(n)$ extra memory.

Time complexity

During preprocessing, we use $O(pq)$ time to `targetH2`, our pattern's hash, and we use $O(nq)$ time to calculating `h_T`, our first n row hashes.

Then, we iterate through $O(m)$ columns in the outer loop. We first calculate the second hash, which is $O(p)$, then we iterate through the innermost loop $O(n)$ times and do a constant number of operations (since rolling is constant). This gives us $O(mp + nm)$.

Hence, the total time is $O(pq+nq+mp+nm) = O(nm)$ ignoring verification of candidate matches.

$\square$

Problem 3.
Let T be a set of teams in a sports league, which, for historical reasons, we assume is baseball. At any point during the season, each team, i, in T, will have some number, w_i, of wins, and will have some number, g_i, of games left to play. The baseball elimination problem is to determine whether it is possible for team k to finish the season first, given the games it has already won and the games it has left to play. Note that this depends on more than just the number of games left for team k, however; it also depends on the respective schedules of team k and the other teams. (See example in Table 1.)

We can solve this problem by reduction to a network flow problem:

Let $g_{i,j}$ denote the number of games remaining between team i and team j.
Let $g_i = \sum_{j \in T \setminus \{k\}} g_{i,j}$.
w_i is the current number of wins for team i.
Let $T' = T \setminus \{k\}$ be the rival teams of k.
Let L be the set of matches left to play between teams in T'.

$$L = \{\{i,j\} : i,j \in T' \text{ and } g_{i,j} > 0\}$$

Let

$$W = w_k + g_k$$

be the aximum number of possible wins for k, and assume

$$W \geq \max_i w_i.$$

To consider how a combination of teams and game outcomes might eliminate team k, we create a graph G:

Vertices: $\{s,t\} \cup L \cup T'$

Edges:

- For each $\{i,j\} \in L$, add edge $(s, \{i,j\})$ with capacity $g_{i,j}$.
- For each $\{i,j\} \in L$, add edges $(\{i,j\}, i)$ and $(\{i,j\}, j)$ with capacity ∞.
- For each team i, add edge (i,t) with weight $W - w_i > 0$.

1. Let f be the maximum flow in this graph. Prove that there exists a combination of results for which team k wins the championship if and only if $f = \sum_{i,j \neq k} g_{i,j}$.
2. What is the time complexity of running the Ford Fulkerson flow algorithm on this graph?

Solution. **1. Correctness:** Let

$$G(T') = \sum_{\{i,j\} \subseteq T',\, i<j} g_{i,j}$$

be the total number of games left to be played between all teams other than team k.

Forward Direction. Assume team k wins all its remaining games (reaching $W = w_k + g_k$), and no other team exceeds W wins.
From this feasible outcome, construct a flow:

- For each game $\{i, j\}$ between teams in T', send g_{ij} units from s to game node (i, j),
- Send each unit from (i, j) to either i or j depending on who wins in the assumed outcome,
- Each team i sends at most $W - w_i$ units to t, since it never exceeds W total wins.

This flow respects all capacities and has total value $\sum_{\{i,j\}\subseteq T'} g_{ij} = G(T')$. Thus the maximum flow satisfies $f = G(T')$.

Backward Direction. Assume $f = G(T')$. Since capacities are integers, there exists an integral max flow.
Then:

- All edges $s \to (i, j)$ are saturated: every game $\{i, j\}$ is assigned.
- Flow from (i, j) to i or j gives how many wins each team receives in that matchup.
- For each team $i \in T'$, flow into i is at most $W - w_i$ (capacity of edge $i \to t$), so its total wins $\leq W$.

So if team k wins all its games, it reaches W wins and no other team exceeds W. Thus team k can still win the championship.

2. Time Complexity of Ford–Fulkerson: The network contains:

- One source s and one sink t,
- One node for each team $i \neq k$ (there are $n - 1$ such nodes),
- One node for each remaining game $\{i, j\}$ between teams $i, j \neq k$; there are $O(n^2)$ such nodes.

Thus, the graph has $O(n^2)$ vertices and $O(n^2)$ edges.

The Ford–Fulkerson method repeatedly:

1. searches the residual graph to find a path from s to t (this takes $O(E)$ time), and
2. pushes as much flow as possible along that path.

Therefore, in the worst casee, Ford–Fulkerson may find an augmenting path up to $G(T')$ times and each path search takes $O(E)$ time, so the total time is:

$$O(E \cdot f_{\max}) = O(n^2 \cdot G(T')).$$

If we assume each pair of teams has at most one game remaining, then $G(T') = O(n^2)$ and

$$O(n^2 \cdot n^2) = O(n^4).$$

Edmonds–Karp (Guaranteed Polynomial Time).

$$O(|V| \cdot |E|^2) = O(n^2 \cdot (n^2)^2) = O(n^6),$$

which holds regardless of capacities. □

Team i	**Wins** w_i	**Games Left** g_i	**Schedule** $(g_{i,j})$			
			LA	Oak	Sea	Tex
Los Angeles	81	8	–	1	6	1
Oakland	77	4	1	–	0	3
Seattle	76	7	6	0	–	1
Texas	74	5	1	3	1	–

Table 1: A set of teams, their standings, and their remaining schedule. Clearly, Texas is eliminated from finishing in first place, since it can win at most 79 games. In addition, even though it is currently in second place, Oakland is also eliminated, because it can win at most 81 games, but in the remaining games between LA and Seattle, either LA wins at least 1 game and finishes with at least 82 wins or Seattle wins 6 games and finishes with at least 82 wins.