CS 5112 Fall 2024 | HW3: *Randomized and Streaming Algorithms*
Name: Samy Lokanandi | **Due: November 26, 11:59pm ET**
NetID: sl3539
Collaborators: Konrad Kopko (kk2239) Grace Myers (gm586)

Problem 1. This problem involves working with graphs, specifically focusing on bipartite properties and matching.

(a) Test if a given graph is bipartite using the Breadth First Search (BFS) approach.

Given the function `is_bipartite(g_l) -> int`, the purpose is to determine if the graph represented by the adjacency list `g_l` is bipartite or not. A simple BFS algorithm can be employed where nodes are colored in a way that no two adjacent nodes share the same color. If it's possible to color the graph using two colors in this way, the graph is bipartite.

(*10 points*) Implement the `is_bipartite` function in `problem_1/p1_a.py`.

Code input and output format. The input for **Problem 1 (a)** is a graph represented as an **adjacency list**, and the output should be a Boolean (True if the graph is bipartite, False otherwise).

(b) Determine the maximal bipartite match of a given bipartite graph.

For the function `maximal_bipartite_match(g) -> int`, the objective is to find the maximum number of matches that can be achieved in the bipartite graph represented by the adjacency matrix `g`.

(*10 points*) Implement the `maximal_bipartite_match` function in `problem_1/p1_b.py`.

Code input and output format. The input for **Problem 1 (b)** is a bipartite graph represented as an **adjacency matrix**. The output should be an integer representing the number of maximal matches.

Answer: Problem 1: Graph Analysis and Bipartite Matching

(a) Test if a Graph is Bipartite

To determine if a graph is bipartite, the Breadth-First Search (BFS) approach is used. A graph is bipartite if its vertices can be colored using two colors such that no two adjacent vertices share the same color. The algorithm starts by initializing a color array to keep track of the colors assigned to each vertex, initially setting all vertices as uncolored. For each unvisited vertex, BFS is performed, alternately coloring vertices and their neighbors. If a conflict is detected (i.e., two adjacent vertices have the same color), the graph is not bipartite. Otherwise, after processing all components, the graph is determined to be bipartite.

Time Complexity: The BFS traversal visits each vertex and edge exactly once, resulting in a time complexity of $O(V+E)$, where V is the number of vertices and E is the number of edges.

—

(b) Find the Maximal Bipartite Match

The maximum matching in a bipartite graph is determined using a Depth-First Search (DFS)-based augmenting path algorithm. The graph is represented as an adjacency matrix. A matching array tracks the matched vertices in one set of the bipartite graph. For each unmatched vertex in the other set, a DFS attempts to find an augmenting path, which either matches an unmatched vertex or reassigns existing matches to create room for the new match. The process iteratively updates the matching until no more augmenting paths can be found, yielding the maximum matching.

Time Complexity: Each DFS explores the edges of the graph for a single vertex. With V vertices in one set and E edges, the time complexity is $O(V \cdot E)$.

Problem 2. Dr. Heinz Doofenshmirtz has just discovered a secret product studio idea in the Bloomberg Masters Studio and is making his way to the Tata Innovation Center to present it. Perry the Platypus from a rival start-up aims to intercept Dr. Heinz Doofenshmirtz by blocking the paths to the Tata Innovation Center. Determine the fewest number of paths that Perry the Platypus should block so that there's no path between Masters Studio and Tata.

`img/edited.png`

(a) (*30 points*) Compute the minimal number of paths to be blocked by Perry the Platypus.

Code input and output format. You are to design a function called `find_paths` with the following inputs:

- n, which correspond to the total number of junctions. The junctions are numbered from 1 to n, Dr. Heinz Doofenshmirtz is at the masters studio, which is at junction 1. Tata Innovation Center is at junction n. It is fine to assume $n \leq 500$ in this problem.
- paths: A list of tuples, where each tuple contains two integers—x and y—indicating a passage between junctions x and y. All paths are accessible in both directions, and there is at most one passage between two junctions.

Examples:

- For n = 5, paths = [(1,2), (3,4), (1,5), (2,5)], the paths (1,2) and (1,5) or (2,5) and (1,5) need to be blocked off to ensure product studio failure. The output should be 2.
- For n = 12, paths = [(1,4), (1,8), (1,9), (8, 9), (9, 12), (8, 11), (8, 12), (3,4), (3, 12), (1,5), (2,5)], three paths, such as (1, 4), (1, 8), (1, 9), need to be blocked off to ensure product studio failure. The output should be 3.

Answer: The `find_paths` function determines the minimal number of paths to block in an undirected graph to disrupt connectivity between junction 1 (source) and junction n (target). It uses the **Edmonds-Karp algorithm**, a BFS-based implementation of the **Ford-Fulkerson method**, to compute the maximum flow in the graph. The maximum flow corresponds to the minimal number of edges (or paths) that must be removed to break the connection. The graph is represented as an adjacency list with capacities initialized to 1 for all edges, signifying that each edge can be traversed once in either direction. A BFS is used to find augmenting paths, with a `route_map` tracing the path to adjust edge capacities and calculate the flow iteratively.

The algorithm terminates when no more augmenting paths exist, yielding the total flow as the minimum number of paths to block. The BFS step ensures efficient pathfinding, while capacity updates reflect path usage dynamically. The time complexity of the function is $O(E \cdot (V + E))$, where V is the number of vertices and E is the number of edges. This accounts for $O(E)$ augmenting paths in the worst case, each requiring a BFS traversal of $O(V + E)$. This approach efficiently combines graph traversal and flow network principles to solve the problem.

Problem 3. Let's say we want to count the number of times elements appear in a stream of data, $x_1, \ldots, x_q$. A simple solution is to maintain a hash table that maps elements to their frequencies.

This approach does not scale: Imagine having an extremely large stream consisting of mostly unique elements. For example, consider network monitoring (either for large network flows or anomalies), large service analytics (e.g. Amazon view/buy counts, Google search popularity), database analytics, etc. Even if we are only interested in the most important ones, this method has huge space requirements. Since we do not know for which items to store counts, our hash table will grow to contain billions of elements.

The Count-Min Sketch, or CMS for short, is a data structure that solves this problem in an approximate way.

Approximate Counting with Hashing. Given that we only have limited space availability, it would help if we could get away with not storing elements themselves but just their counts. To this end, let's try to use only an array, with w memory cells, for storing counts as shown below in Figure 1.

`img/single_hash.png`

Figure 1: Counting with a single hash

With the help of a hash function h, we can implement a counting mechanism based on this array. To increment the count for element x, we hash it to get an index into the array. The cell at the respective index $h(x)$ is then incremented by 1.

Concretely, this data structure has the following operations:

- Initialization: $\forall i \in \{1, \ldots, w\}$: count[$i$]=0.
- Increment count (of element x): count[h(x)]+=1
- Retrieve count (of element x): count[h(x)]

This approach has the obvious drawback of hash conflicts, which result in over-counting. We would need a lot of space to make collisions unlikely enough to get accurate counts. However, we at least do not need to explicitly store keys anymore.

More hash functions

Instead of just using one hash function, we could use d different ones but with the sample memory array. These hash functions should be pairwise independent. To update a count, we hash an item with all d hash functions and increment each resulting index. If two hash functions map to the same index, we increment the cell only once.

Unless we increase the available space, of course all this does for now is to just increase the number of hash conflicts. We will deal with that in the next section. For now let's continue with this thought for a moment.

If we now want to retrieve a count, there are up to d different cells to look at. The natural solution is to take the minimum value of all of these. This is going to be the cell which had the fewest hash conflicts with other cells.

$$\text{min}_{i=1}^{d}\text{count}[\text{h}_i(x)]$$

While we are not fully there yet, this is the fundamental idea of the Count-Min Sketch. Its name stems from the process of retrieving counts by taking the minimum value.

Fewer hash conflicts

We added more hash functions but it is not evident whether this helps in any way. If we use the same amount of space, we definitely increase hash conflicts. In fact, this implies an undesirable property of our solution: Adding more hash functions increases potential errors in the counts.

Instead of trying to reason about how these hash functions influence each other, we can design our data structure in a better way. To this end, we use a matrix of size $w \times d$. Rather than working on an array of length w, we add another dimension based on the number of hash functions as shown below in fig. 2.

Next, we change our update logic so that each function operates on its own row. This way, hash functions cannot conflict with another anymore. To increment the count of element a, we now hash it with a different function once for each row. The count is then incremented in exactly one cell per row.

- Initialization: $\forall i \in \{1, \ldots, d\}, j \in \{1, \ldots, w\}$: count$[i, j] = 0$
- Increment count (of element x): $\forall i \in \{1, \ldots, d\}$: count$[i, \text{h}_i(x)]$ += 1
- Retrieve count (of element x): $\text{min}_{i=1}^{d}\text{count}[i, \text{h}_i(x)]$

This is the full CMS data structure. We call the underlying matrix a sketch.

(a) **(10 points)** Your friend implemented the following hash functions for each row in the sketch: $\text{h}_i(x) = (x + a_i) \mod w$ for $0 < i < d$ where a_i is chosen randomly for each row, and mod is the mod operation (sometimes written as $\%$). Is the choice of hash functions good or bad? Please justify your answer in 1 – 2 sentences or provide a counter example.

img/multiple_hashes.png

Figure 2: Counting with multiple hashes.

Answer: The choice of hash functions $h_i(x) = (x + a_i) \mod w$ is **bad** because these functions are not pairwise independent, which is a crucial property for reducing hash conflicts in the Count-Min Sketch. Each function is a simple linear transformation of x modulo w, meaning they are highly correlated. If two elements x_1 and x_2 hash to the same index under one hash function $h_1(x)$, they will hash to the same index under all $h_i(x)$ for $i = 1, 2, \ldots, d$. This correlation undermines the purpose of having multiple hash functions, as it increases the likelihood of systematic hash conflicts.

For example, consider $w = 10$, $x_1 = 3$, and $x_2 = 13$. If $a_1 = 2$, then $h_1(x_1) = (3 + 2) \mod 10 = 5$ and $h_1(x_2) = (13 + 2) \mod 10 = 5$. This conflict will persist across all $h_i(x)$ because the structure of the hash functions does not vary enough to prevent such collisions. This limitation can result in significant over-counting, especially when x values follow specific patterns or when w is small, and the sketch loses its accuracy advantage.

(b) **(10 points)** Implement CMS in the function `count_min_sketch` in `problem_3/p3_b.py`. Given the two vectors $\mathbf{a} = [a_1, \ldots, a_d]$, $\mathbf{b} = [b_1, \ldots, b_d]$, a scalar p implement the hash function $\mathrm{h}_i(x) = ((a_i x + b_i) \mod p) \mod w$ where d is the number of hash functions (or depth). Then, use this hash function on a stream of data to create the sketch matrix.

Code input and output format. The function `count_min_sketch` takes in the following arguments: $\mathbf{a} = [a_1, \ldots, a_d]$, $\mathbf{b} = [b_1, \ldots, b_d]$ as vectors with positive entries, `w` and `p` as scalars, and a python generator function, `stream`, that produces a stream of data. The output of the function is the sketch matrix, of size $d \times w$.

Example

Given the vectors $\mathbf{a} = [1, 2]$, $\mathbf{b} = [3, 5]$, $p = 100$, $w = 3$, we get the following two hash

functions: $\mathrm{h}_1(x) = ((x + 3)\ \%\ 100)\ \%\ 3$, $\mathrm{h}_2(x) = ((2x + 5)\ \%\ 100)\ \%\ 3$. For the stream of data $[10, 11, 10]$, the function `count_min_sketch` should return the following:
`[[0, 2, 1], [1, 2, 0]]` which corresponds to the following sketch matrix:

```
0, 2, 1
1, 2, 0
```

Answer: The provided code implements a **Count-Min Sketch (CMS)** data structure for approximate frequency counting in data streams. It uses multiple hash functions $h_i(x) = ((a_i \cdot x + b_i) \mod p) \mod w$, where a and b are vectors defining the hash functions, p is a large prime number, and w is the width of the sketch matrix. The `count_min_sketch` function initializes a $d \times w$ sketch matrix (where d is the number of hash functions) and updates the counts in the matrix as elements from the input stream are processed. Each element in the stream is hashed to a specific index in each row of the sketch matrix, and the corresponding cell is incremented, effectively updating the frequency approximation.

The code also includes helper functions for interacting with the CMS. The `increment` function updates the sketch matrix for a specific element, applying all hash functions to determine which cells to update. The `query` function estimates the frequency of a given element by hashing it across all rows of the sketch matrix and returning the minimum value among the corresponding cells. This approach ensures a conservative estimate of the true frequency, reducing overestimation caused by hash collisions. Overall, the implementation efficiently supports frequency estimation in limited space while providing mechanisms to process and query data streams.

CS 5112 Fall 2023	HW3: Challenge Problem
Name: Samy Lokanandi	**Due: November 26, 11:59pm ET**
NetID: sl3539	
Collaborators: Konrad Kopko (kk2239) Grace Myers (gm586)	

In the realm of binary codes, there exists an array of binary strings called `strs`. Each string in `strs` contains only 0s and 1s, forming the key to unlock certain secrets. Alongside this array, two integers `m` and `n` stand as guides, setting constraints on how these binary strings can be combined.

The task at hand is to determine the maximum size of a subset of binary strings from `strs` that meets the conditions defined by `m` and `n`. Specifically, this subset can contain no more than `m` 0s and `n` 1s in total.

A subset `x` is considered valid only if all elements in `x` are also found in the larger set `strs`. The goal is to maximize the number of strings in this subset while adhering to the restrictions on the counts of 0s and 1s.

Implement the function `find_max_form` in `challenge/challenge.py`.

Code input and output format. `strs` is an array of binary strings, and `m` and `n` are integers. The function should return an integer representing the maximum number of strings in the largest subset of `strs` that has no more than `m` 0s and `n` 1s.

Example 1:

- Input: `strs = ["10", "0001", "111001", "1", "0"]`, `m = 5, n = 3`
- Output: 4
- The largest subset with at most 5 0's and 3 1's is `{"10", "0001", "1", "0"}`, so the answer is 4. Other valid but smaller subsets include `{"0001", "1"}` and `{"10", "1", "0"}`. `{"111001"}` is an invalid subset because it contains 4 1's, greater than the maximum of 3.

Example 2:3

- Input: `strs = ["10","0","1"]`, `m = 1, n = 1`
- Output: 2
- The largest subset is `{"0", "1"}`, so the answer is 2.

Answer: Please answer here.