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

Note: *In this assignment ChatGPT was utilized to for latex formatting as well as for general explanations of code and debugging. For the written portion of the assignment, I used ChatGPT to help me understand concepts and double check my work.*

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).

Answer: In my implementation, the `is_bipartite` function checks if a graph, represented as an adjacency list, is bipartite using Breadth-First Search (BFS). A dictionary `color` stores node colors (0 or 1), ensuring adjacent nodes have different colors.

For each unvisited node, the function performs a BFS:

- Unvisited neighbors are assigned the opposite color and added to the queue.
- If a neighbor has the same color as the current node, the graph is not bipartite, and the function returns `0`.

If no conflicts are found, the graph is bipartite, and the function returns `1`.

(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`.

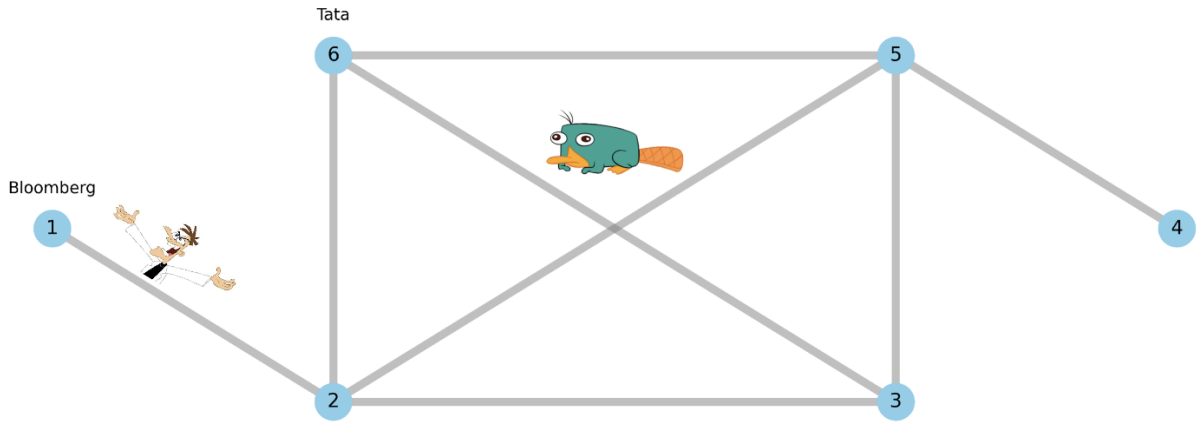
(*10 points*) Implement the `maximal_bipartite_match` function in `problem_1/p1_b.py`.

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:

The function `maximal_bipartite_match(g)` determines the maximum number of matches in a bipartite graph represented as an adjacency matrix `g`. It uses a recursive helper function `bpm` (bipartite matching) to perform a depth-first search and find augmenting paths. The algorithm iterates over each vertex on the left side of the bipartite graph, attempting to match it to an unmatched vertex on the right side, updating the `match_r` array accordingly. The `visited` list prevents revisiting nodes within the same search path. The function returns the total number of matches found, corresponding to the maximum bipartite matching. This implementation is based on the Hungarian algorithm for bipartite graph matching.

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.



(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 function `find_paths(n, paths)` computes the minimum number of paths that need to be blocked to ensure no path exists between junctions 1 and n in a graph represented by bidirectional

edges. The function builds a graph using an adjacency list and a capacity matrix to represent the edges and their capacities.

It implements the Edmonds-Karp algorithm, a variation of the Ford-Fulkerson method, to calculate the maximum flow between junctions 1 (source) and n (sink). The algorithm consists of the following steps:

- A Breadth-First Search (`bfs`) is used to find augmenting paths from the source to the sink.
- The residual capacities along the path are updated to reflect the flow, and the maximum flow is incremented by the path flow.

The result of the function corresponds to the minimum number of paths that must be blocked, which is equivalent to the maximum flow in the graph. This approach ensures an efficient and correct computation of the desired result.

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.

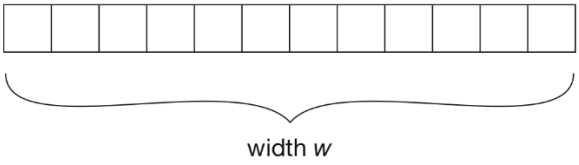


Figure 1: Counting with a single hash

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.

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

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.

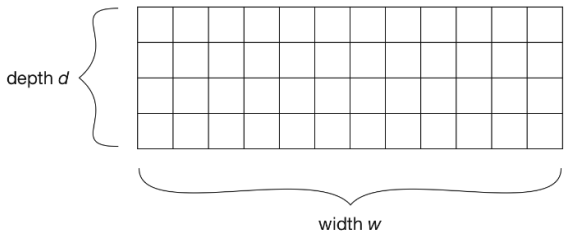


Figure 2: Counting with multiple hashes.

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

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.

Answer:

The hash function $h_i(x) = (x + a_i) \mod w$, where a_i is chosen randomly for each row, is a poor choice for the Count-Min Sketch because it leads to significant hash collisions. Consider the following counterexample:

Let $w = 4$ (number of memory cells), $d = 2$ (number of hash functions), $a_1 = 0$, and $a_2 = 2$. Assume the elements are $x_1 = 3$ and $x_2 = 7$. The hash functions evaluate as follows:

$$h_1(x) = (x + a_1) \mod w = x \mod 4, \quad h_2(x) = (x + a_2) \mod w = (x + 2) \mod 4.$$

Computing the hash indices:

$$h_1(3) = 3 \mod 4 = 3, \quad h_1(7) = 7 \mod 4 = 3,$$
$$h_2(3) = (3 + 2) \mod 4 = 5 \mod 4 = 1, \quad h_2(7) = (7 + 2) \mod 4 = 9 \mod 4 = 1.$$

Both x_1 and x_2 map to the same indices $(3, 1)$ in the sketch, causing their counts to fully overlap. This occurs because $x_2 - x_1 = 4$, a multiple of w. The hash functions fail to distribute values independently across the sketch, leading to significant collisions.

This counterexample demonstrates that the choice of hash functions is unsuitable for the Count-Min Sketch as it relies on modulo arithmetic with insufficient randomness, resulting in inaccurate and biased counts.

(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 function `count_min_sketch(a, b, p, w, stream)` implements the Count-Min Sketch algorithm, a space-efficient data structure for approximating frequency counts in a data stream. The sketch is represented as a $d \times w$ matrix, where d is the number of hash functions, and w is the width of each row. The hash functions are defined using coefficients `a` and offsets `b`, along with a large prime number `p`, to calculate hash values for each element in the stream.

For every element in the input stream, the function computes d hash values, each determining a position in one row of the sketch matrix. The corresponding cells are incremented to record the frequency. This approach ensures that the memory usage is fixed, and updates are fast, making the algorithm suitable for processing large-scale data. The function returns the sketch matrix as the output.

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

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:

The function `find_max_form(strs, m, n)` uses dynamic programming to find the maximum number of binary strings in a subset such that the total number of zeros and ones does not exceed m and n, respectively. It initializes a 2D DP table `dp` of size $(m+1) \times (n+1)$, where `dp[i][j]` represents the maximum number of strings that can be formed with i zeros and j ones.

For each string, the number of zeros and ones is calculated, and the DP table is updated in reverse order (starting from m and n) to avoid over-counting. The update rule ensures that the function

either includes the current string or skips it, based on the maximum possible subset size. Finally, the function returns `dp[m][n]`, which contains the largest subset size that meets the given constraints.