CS 161 **Problem Set 4**
Fall 2025 **Due:** Friday October 31 🎃, at 11:59pm on Gradescope.

Style guide and expectations: Please see the top of the "Homework" page on the course webpage for guidance on what we are looking for in homework solutions. We will grade according to these standards.

Make sure to look at the "**We are expecting**" blocks below each problem to see what we will be grading for in each problem!

Collaboration policy: You may do the HW in groups of size up to three. Please submit one HW for your whole group on Gradescope. (Note that there is an option to submit as a group). See the "Policies" section of the course website for more on the collaboration policy.

LLM policy: Check out the course webpage for best practices on how to productively use LLMs on homework, if you use them at all.

Exercises

We recommend you do the exercises on your own before collaborating with your group. The point is to check your understanding.

---

1. **(3 pt.)** In the IPython Notebook `HW4_E1.ipynb`, available on the website along with this problem set, you will get black-box access to two families of hash functions, A and B. One of these families is a universal hash family, and the other is not. The question for you is: which is which? You can play around with these families on the Jupyter notebook to answer your question.

   [**We are expecting:** *An answer to the question (is A or B the universal hash family?), along with an explanation. Your explanation should include relevant quantitative facts about A and B (a well-labeled graph would be okay too). You should explain what you computed/graphed, and why it convinces you that your answer is correct. Make sure that your answer references the definition of a universal hash family. You do not need to submit any code.*]

SOLUTION:

A is a universal hash family, and B is not. To see this, recall that the definition of a universal hash family (in this setting) is that for all $a, b \in \{0, 1, \ldots, 21\}$ with $a \neq b$, we should have

$$\Pr_{h \in \mathcal{H}}[h(a) = h(b)] \leq \frac{1}{3},$$

where $\mathcal{H}$ is our hash family (either A or B). Equivalently, we can multiply both sides by 506 (the number of functions in both A and B) to see that $\mathcal{H}$ (either A or B) is a universal hash family iff

$$\text{the number of } h \in \mathcal{H} \text{ so that } h(a) = h(b) \leq \frac{506}{3} = 168.666.$$

Thus, for each $a \neq b$ in $\{0, \dots, 21\}$, we should count how many h there are in A so that $h(a) = h(b)$, and we should do the same for B.

I wrote some code to do that, and I found that for A, for every pair of $a \neq b$, the number of $h \in A$ so that $h(a) = h(b)$ is 154, which indeed is less than 168. Thus, A is a universal hash family.

On the other hand, I found that there are lots of pairs a, b (for example, $a = 0$ and $b = 18$; or $a = 1$ and $b = 9$; or $a = 3$ and $b = 4$; etc) so that the number of $h \in B$ so that $h(a) = h(b)$ is 506, aka, all of them. Because there is even one such pair, we conclude that B is *not* a universal hash family.

2. **(2 pt.)** [**Filling in a gap from Lecture 8**] Let h be a uniformly random hash function that maps values u in a set U to the output set $\{1, ..., n\}$. Prove that if $u_i \neq u_j$, that the probability $\Pr[h(u_i) = h(u_j)] = \frac{1}{n}$.

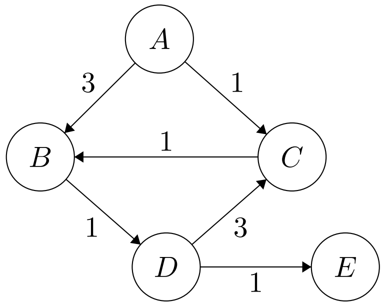
   [**We are expecting:** *A short formal proof. It doesn't need to be long, but make sure that you explicitly use the fact that $h(u_i)$ and $h(u_j)$ are independent for $i \neq j$.*]

SOLUTION: A uniformly random hash function can be produced with the following procedure: For each input value x, let $h(x)$ be a uniformly random value in $\{1, .., n\}$, independent from the choice of $h(y)$ for all $y \neq x$. Thus, for any $u_i \neq u_j \in U$, $h(u_i)$ and $h(u_j)$ are independent random variables. This means that

$$\begin{aligned}
\Pr[h(u_i) = h(u_j)] &= \sum_{a=1}^{n} \Pr[h(u_i) = a \text{ AND } h(u_j) = a] \\
&= \sum_{a=1}^{n} \Pr[h(u_i) = a] \cdot \Pr[h(u_j) = a] \\
&= \sum_{a=1}^{n} \frac{1}{n^2} \\
&= \frac{n}{n^2} = \frac{1}{n}.
\end{aligned}$$

Above, we have used independence of $h(u_i)$ and $h(u_j)$ in the second line.

3. **(4 pt.)** Consider the following directed graph G:

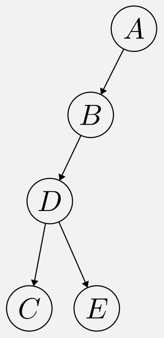


For the following parts you might want to use the website `http://madebyevan.com/fsm/`, which allows you to draw directed graphs in LaTeX. (Note: On a Mac, fn+Delete will delete nodes or edges). It is also fine to include an image created in your favorite drawing program, or a photo/scan of a hand-drawn graph.[1]

(a) **(2 pt.)** Draw the DFS tree for G, starting from node A. Assume that DFS traverses nodes in alphabetical order. (That is, if it could go to either B or C, it will always choose B first).

[**We are expecting:** *A picture of your tree. No further explanation is required.*]

SOLUTION:

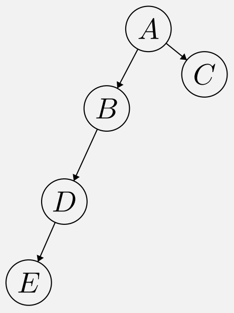


(b) **(2 pt.)** Draw the BFS tree for G, starting from node A. Assume that BFS traverses nodes in alphabetical order.

[**We are expecting:** *A picture of your tree. No further explanation is required.*]

SOLUTION:

[1] If you want to use an LLM to generate LaTeX code for a graph that you have come up with on your own, and which you describe to the LLM, that's fine; and it's okay to copy-and-paste that LLM's LaTeX output to generate your picture.



A
B
C
D
E

4. **(5 pt.)** Directed graphs are one of many ways to translate a messy real-world problem to one that admits to algorithmic analysis. Recall from the first Embedded Ethics recorded lecture, the case study from Bogotá's Ministry of Transport, where the Ministry attempted to measure the average speed of vehicles on roads using two spaced Bluetooth sensors. One way they might make use of such information is in the generation of a directed graph like the one in the previous exercise, where vertices represent locations of interest, and edge weights represent the expected amount of time it would take for someone to travel from one vertex to another.

   (a) **(3 pt.)** Suppose the Ministry of Transportation used such a directed graph to help plan routes for an expansion of their bus system.

   What are two features of transportation infrastructure relevant to the problem of planning bus routes that are **abstracted** or **idealized** away in representing transportation infrastructure as a directed graph?

   [**We are expecting:** *A short paragraph providing two relevant features of transportation infrastructure that are abstracted or idealized away. State the two features, explain why they are relevant to the problem of bus routing, and explain why the directed graph representation abstracts or idealizes away these features.*]

   SOLUTION:
   (Students can choose to present 2 abstractions, 2 idealizations, or 1 of each, so long as they present 2 total examples) Example: 2 relevant features of transportation that are abstracted away in the graph representation are *road infrastructure* and *the types of vehicles expected on the road.* They are abstracted away in the graph representation because they are present in the real-world situation the graph is meant to represent, but are absent in the representation. Road infrastructure might be relevant because some elements of infrastructure make bus transport easier or harder. For example, a pre-existing dedicated bus lane might make bus transportation substantially faster than the average speed would suggest, or curb extensions might make it harder for buses to turn at speed. The type of vehicle might be relevant because it may contribute to over- or under-estimated speeds for bus routes. A road with many bicycles on it might have a lower average speed than the expected bus speed, but a road with many motorcycles on it might have a higher average speed than the expected bus speed.

   (b) **(2 pt.)** Are there any harmful downstream consequences of deciding bus routes while abstracting or idealizing away the two features you stated in part (a)?

   [**We are expecting:** *1-3 sentences identifying a possible harm of abstracting or idealizing away the relevant feature stated in part (a)*]

   SOLUTION:
   (Any reasonable connection from the abstraction to a potential harm is good here, even if a little far-fetched.) Yes, ignoring road infrastructure might make a route more or less dangerous than expected. Curb extensions, for example, often have places close to the

corner for pedestrians to wait to cross, and it might make it more likely for an accident to happen if a bus takes the turn too sharply.

Problems

5. **(12 pt.)** **[Painted Penguins.]** A large flock of T painted penguins will be waddling past the Stanford campus next week as part of their annual migration from Monterey Bay Aquarium to the Sausalito Cetacean Institute. Painted Penguins (not to be confused with pedantic penguins) are an interesting species. They can come in a huge number of colors—say, M colors—but each flock of T penguins only has m colors represented, where $m < T$. The penguins will waddle by one at a time, and after they have waddled by they won't come back again.

   For example, if $T = 7$, $M = 100000$ and $m = 3$, then a flock of T painted penguins might look like:

   `seabreeze, seabreeze, indigo, ultraviolet, indigo, ultraviolet, seabreeze`

   You'll see this sequence in order, and only once. After the penguins have gone, you'll be asked questions like "How many `indigo` penguins were there?" (Answer: 2), or "How many `neon orange` penguins were there?" (Answer: 0).

   You know m, M and T in advance (and you know the set of M possible colors), and you have access to a universal hash family $\mathcal{H}$, so that each function $h \in \mathcal{H}$ maps the set of M possible colors into the set $\{0, \ldots, n-1\}$, for some integer n. For example, one function $h \in \mathcal{H}$ might have $h(\texttt{seabreeze}) = 5$.

   (a) **(6 pt.)** Suppose that $n = 10m$. Suppose also that you only have space to store:

   - An array B of length n, which stores numbers in the set $\{0, \ldots, T\}$, and
   - one function h from $\mathcal{H}$.

   Use the universal hash family $\mathcal{H}$ to create a randomized data structure that fits in this space and that supports the following operations in time $O(1)$ in the worst case (assuming that you can evaluate $h \in \mathcal{H}$ in time $O(1)$):

   - `Update(color)`: Update the data structure when you see a penguin with color `color` waddle by.
   - `Query(color)`: Return the number of penguins of color `color` that you have seen so far. For each query, your query should be correct with probability at least 9/10. That is, for all colors `color`,

   $$\mathbb{P}\{\texttt{Query(color)} = \text{ the true number of penguins with color } \texttt{color}\ \} \geq \frac{9}{10}.$$

   To describe your data structure:

   i. Describe how the array B and the function h are initialized.

ii. Give pseudocode for `Query`.

iii. Give pseudocode for `Update`.

[**We are expecting:** *A description following the outline above (including pseudocode), and a short but rigorous proof that your data structure meets the requirements. Make sure you clearly indicate where you are using the property of universal hash families.*]

(b) **(6 pt.)** Suppose that you now have k times the space you had in part (a). That is, you can store k arrays $B_1, \dots, B_k$ and k functions $h_1, \dots, h_k$ from $\mathcal{H}$. Adapt your data structure from part (a) so that all operations run in time $O(k)$, and the `Query` operation is correct with probability at least $1 - \frac{1}{10^k}$.

[**We are expecting:** *A description following the outline above (except say how all arrays B_i and functions h_i are initialized), and a short but rigorous proof that your data structure meets the requirements. Make sure you clearly indicate where you are using the property of universal hash families.*]

SOLUTION:

(a) Here is the description of our data structure:

- Our data structure stores an array B of length n, where each bucket stores a number in $\{0, \dots, T\}$ and is initialized to zero. Before the flock waddles by, we choose a random $h \in \mathcal{H}$ and store that too.
- `Update(color):  B[h(color)] ++`
- `Query(color):  Return B[h(color)].`

Each of these operations takes time $O(1)$. The probability that a single `Query` option fails is the probability that any of the m (or $m-1$ other) colors which did appear collided with the color that was queried. That is, we want

$$\mathbb{P}\{\text{there is a color } x \text{ which appeared, not the same as } \texttt{color}\text{, so that } h(x) = h(\texttt{color})\}$$

to be small. By the universal hash family property, we have for each color x,

$$\mathbb{P}\{h(x) = h(\texttt{color})\} \le \frac{1}{n}.$$

Thus, by the union bound, the probability that there exists an x which appeared that collides with `color` is at most

$$\mathbb{P}\{\text{there is a color } x \text{ which appeared, not the same as } \texttt{color}\text{, so that } h(x) = h(\texttt{color})\}$$
$$\le m \cdot \mathbb{P}\{h(x) = h(\texttt{color})\} \le \frac{m}{n} = \frac{1}{10}.$$

(b) We will basically just keep k copies of our data structure from part (a). More precisely, our data structure stores:

- k arrays $B_1, \dots, B_k$, initialized to zero.
- k hash functions $h_1, \dots h_k \in \mathcal{H}$, chosen uniformly at random and independently. (With replacement).

Then our update strategy is:

```
Update(color):
    for i = 1,...,k:
        B_i[ h_i(color) ] ++



Query(color):
    return min_{i = 1,...,k} B_i[ h_i(color) ]
```

Both of these operations take time $O(k)$, since they both loop over k things.

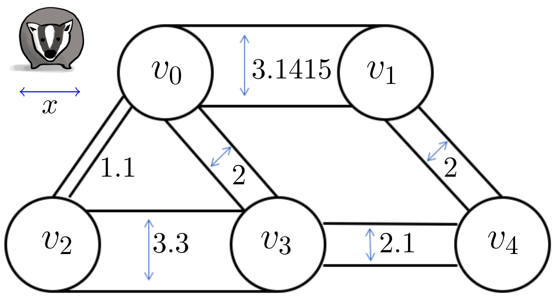
To compute the success probability of `Query`, notice that this returns the correct value as long as the color `color` is isolated in *any* of the k tables. Since each of these k hash functions are independent, we have:

$$\begin{aligned}
&\mathbb{P}\{\text{for all } i\text{, there is a color } x \text{ which appeared, not the same as } \texttt{color}\text{, so that } h_i(x) = h_i(\texttt{color})\}\\
&= \left(\mathbb{P}\{\text{there is a color } x \text{ which appeared, not the same as } \texttt{color}\text{, so that } h_i(x) = h_i(\texttt{color})\}\right)^k\\
&\le \left(m \cdot \mathbb{P}\{h(x) = h(\texttt{color})\}\right)^k\\
&\le \left(\frac{m}{n}\right)^k\\
&= \frac{1}{10^k}.
\end{aligned}$$

Thus, with probability at least $1 - 1/10^k$, there is at least one i so that $B_i[h_i(\texttt{color})]$ is equal to the number of times that that `color` appeared, and `Query(color)` returns the right thing.

6. **(6 pt.)** **[Badger badger badger.]** A family of badgers lives in a network of tunnels; the network is modeled by a connected, undirected graph G with n vertices and m edges (see below). Each of the tunnels have different widths, and a badger of width x can only pass through tunnels of width $\geq x$.

For example, in the graph below, a badger with width $x = 2$ could get from v_0 to v_4 (either by $v_0 \to v_1 \to v_4$ or by $v_0 \to v_3 \to v_4$). However, a badger of width 3 could not get from v_0 to v_4.



The graph is stored in the adjacency-list format we discussed in class. More precisely, G has vertices stored as an array V of length n, and edges stored in an array E of length m. For each $i = 0, \ldots, n-1$, $V[i]$ stores a pointer to the head of a linked list N_i, which stores integers that index E. If e is in N_i, that means that the edge represented by $E[e]$ touches the i'th vertex. For each e, $E[e]$ stores two integers (say, $E[e][0]$ and $E[e][1]$) so that if i is in $E[e]$, then the i'th vertex is an endpoint of that edge.

You have access to a function `tunnelWidth`, which runs in time $O(1)$, so that if e is an edge in G, (that is, an integer between 0 and $m-1$ that indexes E), then `tunnelWidth(e)` returns the width of the corresponding tunnel.

If it is helpful, you may assume access to a function `BFS`. Given a corresponding vertex array V of pointers to linked lists, edge array E, and source vertex $s \in \{0, ..., n-1\}$, you may assume the function `BFS`(V, E, s) returns an array `distance` where `distance`$[i]$ is the number of edges on the shortest path from s to node i (or -1 if i is unreachable from s). The BFS function runs in $O(n+m)$ time.

[Actual questions on next page.]

(a) **(6 pt.)** Design a deterministic algorithm that takes as input G in the format above, integers $s, t \in \{0, \ldots, n-1\}$, and a desired badger width $x > 0$; the algorithm should return **True** if there is a path from v_s to v_t that a badger of width x could fit through, or **False** if no such path exists.

(For example, in the example above we have $s = 0$ and $t = 4$. Your algorithm should return **True** if $0 < x \leq 2$ and **False** if $x > 2$).

Your algorithm should run in time $O(n + m)$. You may use any algorithm we have seen in class as a subroutine.

Note: In your pseudocode, make sure you use the adjacency-list format for G described above. For example, your pseudocode should *not* say something like "iterate over all edges in the graph." Instead it should more explicitly show how to do that with the format described. (We will not be so pedantic about this in the future, but one point of this problem is to make sure you understand how the adjacency-list format works).

[**We are expecting:** *Pseudocode **AND** an English description of your algorithm, and a short justification of the running time. You should make sure to use the adjacency-list representation of G described above in your pseudocode. You can use any algorithms we have seen from class as a subroutine, but if you significantly modify them, make sure to be precise about how this interacts with the adjacency-list representation.*]

(b) **(0 pt.)** [**This part is OPTIONAL since this PSET is long enough. It won't be graded, but it's good practice!**] Design a deterministic algorithm which takes as input G in the format above and integers $s, t \in \{0, \ldots, n-1\}$; the algorithm should return the largest real number x so that there exists a path from v_s to v_t which accomodates a badger of width x. Your algorithm should run in time $O((n + m)\log(m))$. You may use any algorithm we have seen in class as a subroutine. (Hint, use part (a)).

Note: Don't assume that you know anything about the tunnel widths ahead of time. (e.g., they are not necessarily bounded integers).

[**We are expecting:** *Nothing, this part is optional! But if we were expecting something, it would be: Pseudocode **AND** and English description of your algorithm, and a short justification of the running time.*]

SOLUTION:

(a) The idea is to remove the too-narrow edges from G, and then run BFS (or DFS works too) to see if there is a path between s and t.

```
def widePath( G, s, t, x ):
     Say that G = V,E
     # first, remove all of the edges that are too narrow
     for i in range(n):
         Ni = V[i]
         for e in Ni:
             if tunnelWidth(e) < x:
                 remove e from Ni
                 \\ time O(1) to remove e since we are already looking at it
```

```
                \\ (just manipulating pointers)
                E[e] = None
        distance = BFS(V,E,s)
        if distance[t] != -1:
            \\ t was reached during the BFS
            return True
        Else:
            return False
```

This algorithm runs in time $O(n+m)$. First, it loops over the edges and removes the ones that are too narrow. This takes time $O(m+n)$, since for each vertex i we do $O(\deg(i)) + O(1)$ work: for each neighbor j of i we check if the edge $e = \{i, j\}$ is too narrow, and potentially remove it.

$$O\left(\sum_i \deg(i) + \sum_i 1\right) = O(m+n)$$

work. Next, it runs BFS, which takes time $O(n+m)$.

You might be worried that if $e = \{i, j\}$, then we remove e from N_i, but we don't remove e from N_j when we do that. However, we will remove it when we hit j in the outer loop.

Note: It is also okay to write down a modified version of BFS/DFS that just ignores the edges which are too narrow for the badger, as long as the modified version is sufficiently pedantic about the adjacency-list format.

Note: It is also okay to say that the algorithm actually runs in time $O(m)$. This is because we only need to explore the connected component of the graph that s is in, so without loss of generality we may assume that G is connected. In that case, $n = O(m)$, so $O(n+m) = O(m)$.

(b) The high-level idea is to use binary search and then use part (a) to find the largest x that works. Notice that the maximum x must be equal to one of the edge weights.

```
def widestPath( G, s, t ):
    if s == t:
        return Infinity  # any badger can fit in a path from s to t, since s=t

    # first, generate a list of the weights that appear, in time O(m + n)
    W = []
    for i in range(n):
        Ni = V[i]
        for e in V[i]:
            if i == min(E[e][0],E[e][1]):  # so that we don't add the tunnel twice
                 add tunnelWidth(e) to W
     sort W in time O(m log(m)) using MergeSort.

     # now run binary search to see what the largest width we can accomodate.
     a=0, b=n-1 # we are searching in {a,a+1,...,b}
     while b > a:
        mid = ceiling( (a+b)/2 )
        isThereAPath = widePath( G, s, t, W[mid] )
        if isThereAPath:
              # then a badger of size W[mid] could get from s to t;
              # next check a wider badger
              a = mid
        else:
              # then a badger of size W[mid] was too wide to get from s to t;
              # next check a narrower badger
              b = mid - 1

      # now we should have a=b = index of widest possible value in W.
      return W[a]
```

The running time is $O((n+m)log(m))$. This is because it takes time $O(mlog(m))$ to sort the list. Then we call `widePath` $O(\log(m))$ times during the binary search, and each time it takes time $O(n+m)$.