CS120: Intro. to Algorithms and their Limitations Hesterberg & Vadhan

Section 5: Randomization and Graphs

Harvard SEAS - Fall 2022 *Oct. 6, 2022*

1 Randomized Algorithms

There are two different flavors of randomized algorithms:

- *Las Vegas Algorithms:* Always output a correct answer, but their running time depends on their random choices. In the worst case, Las Vegas algorithms can have infinite running time. Typically, we try to bound their *expected* running time. That is, we say the *(worst-case) expected running time* of A is

$$T(n) = \max_{x:\text{size}(x)\leq n} \mathrm{E}[\text{Time}_A(x)],$$

  where $\text{Time}_A(x)$ is the random variable denoting the runtime of A on x.

- *Monte Carlo Algorithms:* Always run within a desired time bound $T(n)$, but may err with some small probability (if they are unlucky in their random choices), i.e. we say that A *solves* computational problem $\Pi = (\mathcal{I}, f)$ *with error probability* p if

$$\text{for every } x \in \mathcal{I},\ \Pr[A(x) \in f(x)] \geq 1 - p$$

  Think of the error probability as a small constant, like $p = .01$. Typically this constant can be reduced to an astronomically small value by running the algorithm several times independently.

Prevailing conjecture (based on the theory of pseudorandom number generators) is: **randomness is not necessary for polynomial-time computation.** More precisely, we believe a $T(n)$ time Monte-Carlo algorithm *implies* a deterministic algorithm in time $O(n \cdot T(n))$, so we can convert any randomized algorithm into a deterministic one. (You do not need to know details about this conjecture beyond its statement for the purposes of this class.)

1.1 QuickSelect

Input	: An array A of key-value pairs $((K_0, V_0), \ldots, (K_{n-1}, V_{n-1}))$, where each key $K_j \in \mathbb{N}$, and a *rank* $i \in [n]$
Output	: A key-value pair (K_j, V_j) such that K_j is an i'th smallest key. That is, there are at most i values of k such that $K_k < K_j$ and there are at most $n - i - 1$ values of k such that $K_k > K_j$.

Computational Problem Selection

In particular, when $i = (n-1)/2$, we need to find the *median* key in the dataset.

Theorem 1.1. *There is a randomized algorithm QuickSelect that always solves Selection correctly, and has (worst-case) expected running time $O(n)$.*

Question 1.2. Perform the QuickSelect algorithm on the following array:

$$A = [(8, a), (13, b), (2, c), (25, d), (4, e), (6, f), (1, g), (21, h), (7, i), (40, j), (32, k)]$$

and $i = 3$. During the algorithm, the first pivot happens to be 21, then 8, then 1, and then 4.

Question 1.3. What is the worst case runtime of QuickSelect? Provide an example of such a scenario, and compute the probability that the choice of pivots is "as bad as possible" in your example. What is the best runtime of QuickSelect, and what is that scenario look like? What is the probability that you pick this "lucky pivot"? (For simplicity, assume no keys are repeated in this problem.)

2 Randomized Data Structures

A Monte Carlo data structure:

- Preprocess(U, m): Initialize an array A of size m. In addition, choose a random hash function $h : [U] \to [m]$ from the universe $[U]$ to $[m]$.
- Insert(K, V): Place (K, V) into the linked list at slot $A[h(K)]$.
- Delete(K): Remove an element from the linked list at slot $A[h(K)]$.
- Search(K): Return the head of the linked list at $A[h(K)]$.

If there are two distinct keys K_1, K_2 with $h(K_1) = h(K_2)$, on the query Search(K_1) we could return an key-value pair (K_2, V). To bound this error probability, consider a query Search(K). In order for us to return the wrong value, we must have inserted a distinct key that hashes to the same value as $h(K)$. If we've inserted items with keys $K_0, \ldots, K_{n-1}$ that are different than K, we have:

$$\begin{aligned}\Pr[\text{Search}(K) \text{ returns incorrect value}] &\leq \Pr[h(K) \in \{h(K_0), \ldots, h(K_{n-1})\}] \\ &\leq \sum_{i=0}^{n-1} \Pr[h(K) = h(K_i)] \\ &= n \cdot \frac{1}{m}\end{aligned}$$

where in the final line we used that h was a random mapping from $[U]$ to $[m]$.

A Las Vegas data structure ("Hash Table"): The same data structure as above, except the linked list at every array index stores (K, V) pairs. Then when we query Search(K), go to the linked list $A[h(K)]$ and return the first element of the list that has the correct key K (and if none do, return $\perp$).

Here, we bound the *expected runtime* via the same analysis as before, because if no elements collide (the event we bound above) the additional linked list checking will only be an $O(1)$ slowdown. Quantitatively, we can show an expected runtime of $O(1 + n/m)$. We call $\alpha = n/m$ the *load* of the table, so the runtime is $O(1 + \alpha)$. Notice that here we get $O(1)$ expected time even if $n > m$, provided that $m = \Omega(n)$.

Question 2.1. Let $U = \{1, \ldots, 100\}$. Let the Hash Table have 9 slots, and let the hash function be $h(k) = k \mod 9$. Draw the Hash Table that results when we insert the keys 5, 28, 19, 15, 20, 33, 12, 17, 10. What is the value of the load factor α in this case? What would α be if the hash function h was defined as $h(x) = x$ for all $x \in U$? In general, do we want α to be large or small?

Question 2.2. Why do we say that a Hash Table is a randomized data structure? Where is the randomization coming from? For a hash function h, is the value $h(x)$ for any x deterministic?

3 Graphs

Definition 3.1. A *directed graph* $G = (V, E)$ consists of a finite set of *vertices* V (sometime called *nodes*), and a set E of ordered pairs (u, v) where $u, v \in V$ and $u \neq v$.

Types of Graphs:

- Directed vs. Undirected
- Multigraphs (more than one edge between u and v) vs. Simple Graphs
- Weighted vs. Unweighted
- Cyclic vs. Acyclic
- In this class, assume *graph* means a **simple, unweighted, undirected** graph.
- In this class, assume *digraph* means a **simple, unweighted, directed** graph.

Graph Representation:

Graphs are a common, general abstraction for a wide variety of more specific scenarios; many of our abstract graph problems can be motivated by real-world situations. For instance, let's think about a One-Player Game $G = (S, M, T)$, represented by a set of game States, S, a set of Moves between states, M, and a set of target states $T \subset S$. Our game will have some initial state s_0, like a freshly-set chess board, and many potential "target states" $t_i \in T$ where the player wins.

One such one-player game is the Tower of Hanoi game. Your goal is to move a stack of disks from one peg to another. There are two constraints to your moves: 1. You may only move 1 top disk at a time, and 2. a bigger disk may never be placed on top of a smaller disk.

Question 3.2. How might you represent the states (elements of S) of this game? How about the moves (elements of M)? Now, interpreting S as a set of vertices, and M as a set of directed edges, draw out the first couple ($d \leq 2$) layers of this graph.

4 BFS

Motivating problem: Shortest walk from s to t.

Input	: A digraph $G = (V, E)$ and two vertices $s, t \in V$
Output	: A *shortest walk* from s to t in G, if any walk from s to t exists

Computational Problem ShortestWalk

Definition 4.1. Let $G = (V, E)$ be a directed graph, and $s, t \in V$.

- A *walk* w from s to t in G is a sequence $v_0, v_1, \dots, v_\ell$ of vertices such that $v_0 = s$, $v_\ell = t$, and $(v_{i-1}, v_i) \in E$ for $i = 1, \dots, \ell$.
- The *length* of a walk w is length(w) = the number of edges in w (the number ℓ above).
- The *distance* from s to t in G is

$$\mathrm{dist}_G(s,t) = \begin{cases} \min\{\mathrm{length}(w) : w \text{ is a walk from } s \text{ to } t\} & \text{if a walk exists} \\ \infty & \text{otherwise} \end{cases}$$

- A *shortest walk* from s to t in G is a walk w from s to t with length$(w) = \mathrm{dist}_G(s,t)$

BFS Algorithm:

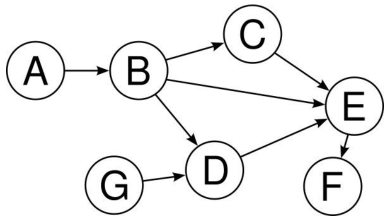
```
1 BFS($G, s, t$)
  Input  : A directed graph $G = (V, E)$ and two vertices $s, t \in V$ (in adjacency list
           representation)
  Output : The distance from $s$ to $t$ in $G$
2 $S = \{s\}$;
3 $F = \{s\}$ $d = 0$;
4 while $F \neq \emptyset$ do
5     if $t \in F$ then return $d$;
6     $F = \{v \in V - S : \exists u \in F$ s.t. $(u, v) \in E\}$;
7     $S = S \cup F$;
8     $d = d + 1$;
9 return $\infty$
```

BFS Applications

Breadth-first search (BFS) is an algorithm that is used to traverse graphs or searching trees. It is one of the many methods of graph traversal that allows us to easily find the shortest path.

- Web Crawling
- Social Networking
- GPS Navigation
- Solving puzzles and games (Check Section 6 notes for an example)

Question 4.2. Consider the following graph:



1. Write down the adjacency list representation of this graph.
2. Run BFS on the above graph with $s = A$ and $t = F$, writing down the sets S_i and F_i at each iteration of the algorithm.
3. What can you say about the sets F_i at each iteration of the algorithm?
4. If besides the distance between $s = A$ and $t = F$ we wanted to find *the exact shortest path* between them, what do we add to the BFS algorithm? Run it again with this extension. (And notice that at the end we have the shortest paths from s to any vertex reached by s during BFS.)
5. Did the runtime change? What is the runtime in the case of part (b) and what is the runtime in the case of part (c)?

BFS as a Proof Strategy

Let's re-examine some of the most helpful features of BFS:

1. S: This set keeps track of every vertex we've already visited, to make sure we don't explore vertices more than once. In BFS, whenever we visit a point, we're sure we've visited it through the shortest path possible, so there's no need to explore vertices multiple times!
2. F: The Frontier set; F contains every vertex up next on our roster of vertices to explore. At each iteration of our While loop, F is updated to contain all the "children" of the current vertices in F, excluding those in S which we've already explored. Examining $|F|$, the size of the frontier, can provide a useful bound in proofs about certain graphs.
3. d: This represents a distance away from our source node s. For every step in our BFS loop, we increment this counter by 1; accordingly, at every step in our BFS loop, we can be sure that F contains vertices at exactly distance d away from s. We call this the *loop invariant*, and we can exploit this fact to induct on d (or equivalently, loop iteration) in our proofs.

It is often helpful to use BFS as a tool for certain graph proofs. Framing proofs in terms of these variables in BFS is a good proof strategy. For example, try this one out:

Question 4.3. Prove that an undirected connected graph where two vertices have degree 1 and all other vertices have degree 2 must form a path of length $n - 1$, where n is the number of vertices.

- A graph G forms a path if we can order all vertices into $v_0, v_1, \ldots, v_\ell$ such that for $i = 1, ..., \ell$, $(v_{i-1}, v_i) \in E$ and $(v_i, v_j) \notin E$ otherwise. (This means that all pairs of consecutive vertices are connected by an edge, and there are no other edges.)
- The degree of a node in an undirected graph is the number of edges connected to it.