UNIVERSITY OF TORONTO
Faculty of Arts and Science

Midterm 2, Version 2
CSC263H1F

Friday November 11, 2016, 1:10-2:00pm (**50 min.**)
Examination Aids: No aids allowed

Name:

Student Number:

Please read the following guidelines carefully!

- Please write your name on the front **and back** of the exam.
- This examination has **3** questions. There are a total of **10 pages, DOUBLE-SIDED**.
- Answer questions clearly and completely. Give complete justifications for all answers unless explicitly asked not to. You may use any claim/result from class, unless you are being asked to prove that claim/result, or explicitly told not to.

Take a deep breath.

This is your chance to show us

How much you've learned.

We **WANT** to give you the credit

That you've earned.

A number does not define you.

Good luck!

1. **Hashing**. Part (a) should be done before (b) and (c); part (d) is independent of the others.

   (a) **[1 mark]** Suppose we have a hash table H of size m that resolves collisions using open addressing, with the linear probe sequence $h(k, i) = hash(k) + i \pmod m$, where $hash$ is a hash function.

   Suppose this hash table contains n keys, where n is divisible by 3, and the key locations in the array satisfy the following:

   - The keys are grouped into $n/3$ groups, each of size 3.
   - Each group is stored in a block of 3 consecutive locations, and each block is followed by an empty spot. "Consecutive" and "followed by" wrap around the end of the array. So if a block occupies indices $m-3$ to $m-1$, index 0 must be unoccupied.

   Draw an example of a hash table that satisfies this property for $m = 8$ and $n = 6$. Clearly identify which array spots are occupied and which are unoccupied.

Solution

Diagram omitted, but the array must have size 8, and one possible positioning of keys is at indices $\{0, 1, 2, 4, 5, 6\}$, leaving $\{3, 7\}$ unoccupied.

   (b) **[1 mark]** Suppose we randomly pick a new key k, and assume that for every index $0 \leq j < m$, the probability that $hash(k) = j$ is $\frac{1}{m}$.

   Find the maximum number of array spots visited when inserting k into the hash table H, and justify your answer. Here, "visited" includes the empty spot where the key is eventually inserted.

Solution

At most 4 spots are visited. This happens if a key is inserted at the front of one of these groups, and so must visit the whole group (size 3), but then reaches an unoccupied array spot.

   (c) **[2 marks] Find the probability** that exactly two array spots are visited when inserting k into the hash table, in terms of m and/or n.

Solution

This occurs when the key has the same hash value as one of the last elements in a group. There are exactly $n/3$ such spots, and since each of the m array spots is equally likely, the probability is $\frac{n}{3m}$, or $\frac{1}{3}\alpha$.

(d) **[3 marks]** Suppose we have a hash table of length m that resolves collisions using closed addressing with linked list chaining, and that currently stores n numeric keys.

Your task is to find an algorithm for finding the *smallest* key stored in the hash table.

In your solution, give both the **pseudocode** for your algorithm, as well as a **worst-case lower bound analysis** of your algorithm. Your lower bound should be tight, but you do not need to prove this.

Your solution will be graded on both correctness and efficiency.

Solution

This is pretty similar to an assignment question; loop through the different array elements, and the linked list at each one, keeping track of the smallest key.

```
1  def SmallestKey(H):
2      smallest_key = null
3      for i from 0 to H.length - 1:
4          # Loop through the linked list at H[i]
5          curr = H[i].head
6          while curr is not null:
7              if smallest_key == null or smallest_key > curr.key:
8                  smallest_key = curr.key
9              curr = curr.next
10
11     return smallest_key
```

The running time is $\Omega(m + n)$; the analysis is basically the same as the one from Assignment 4.

2. **Graph Searches, Randomization**. Part (a) is independent of the other three parts.

   (a) **[2 marks]** Consider this recursive version of Depth-First Search:

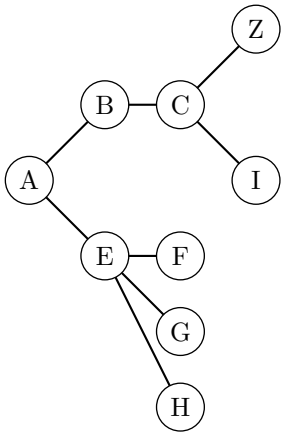
```
def DFS(graph, s):
    initialize all vertices in the graph to not started
    DFS_helper(graph, s)

def DFS_helper(graph, v):
    v.started = True
    Visit(v)  # do something with v, like print out its label
    for each neighbour u of v:
        if not u.started:
            DFS_helper(graph, u)
```

   Find an exact upper bound on the maximum possible *number* of calls to `DFS_helper` when we run DFS. Your answer should be in terms of the number of vertices, $|V|$, and/or edges, $|E|$, in the graph. You do not need to prove that this bound is tight.

Solution

The most important observation is that `DFS_helper` is not called on the same vertex twice. When this function is called on a vertex, its `started` attribute is set to True, and `DFS_helper` is only called on vertices that have not been started. So `DFS_helper` is called a maximum of $|V|$ times.

   (b) **[1 mark]** Consider the following graph.



   Suppose we *randomize* BFS and DFS so that for each vertex v, we randomly permute its neighbours before looping through them:

```
for each neighbour u of v, in a random order:
```

   Consider running randomized BFS on the above graph, starting at vertex A. **State one possible order** of vertex visits made by this BFS, and **the number of vertices** visited before Z is visited in this order. No justification is required.

Solution

Many possible answers, including A, B, E, C, F, G, H, Z, I. 7 vertices are visited before Z in this order.

(c) **[2 marks]** Let T_{BFS} be a random variable representing the number of vertices visited before vertex Z when we run a randomized BFS on this graph starting at A.

Find the expected value of T_{BFS}. Make sure to explain your work here.

Hint: There are only a few possible values for T_{BFS}; you don't need to enumerate all possible choices made by BFS to answer this question.

Solution

The key insight is that only the order in which Z and I are visited matters; Z is guaranteed to be visited after all the other vertices.

There is a $\frac{1}{2}$ chance that Z is picked before I and a $\frac{1}{2}$ chance Z is picked after I. The expected number of vertices visited before Z is

$$\mathbb{E}[T_{BFS}] = \frac{1}{2} \cdot 7 + \frac{1}{2} \cdot 8 = \frac{15}{2}.$$

(d) **[2 marks]** Define T_{DFS} as analogous to T_{BFS}, except that we run a randomized DFS instead of BFS, but still starting at A.

Find the expected value of T_{DFS}. Make sure to explain your work here.

Solution

This now depends on whether B is visited before E, and whether Z is visited before I; these choices are independent of each other. There are four choices total, and each results in a different number of vertices visited before Z. Skipping some justification:

$$\begin{aligned}\mathbb{E}[T_{DFS}] &= 3 \cdot \Pr[T_{DFS} = 3] + 4 \cdot \Pr[T_{DFS} = 4] + 7 \cdot \Pr[T_{DFS} = 7] + 8 \Pr[T_{DFS} = 8] \\ &= 3 \cdot \frac{1}{4} + 4 \cdot \frac{1}{4} + 7 \cdot \frac{1}{4} + 8 \cdot \frac{1}{4}\end{aligned}$$

3. **Graph Search Applications**. Part (a) is independent of (b) and (c).

   Suppose we are given a graph $G = (V, E)$, and vertex s. We want to support the operation PrintClose, which prints out a list of all vertices which are at distance at most 2 from s, i.e., s itself, its neighbours, and the neighbours of its neighbours. Each such vertex should be printed out once. Order doesn't matter.

   Here is one algorithm that does this, basically using BFS.

```
def PrintClose(graph, s):
    # Run BFS with distances
    queue = new empty queue
    initialize all vertices in the graph to not enqueued

    queue.enqueue(s)
    s.enqueued = True
    s.distance = 0

    while queue is not empty:
        v = queue.dequeue()

        for each neighbour u of v:
            if not u.enqueued:
                queue.enqueue(u)
                u.enqueued = True
                u.distance = v.distance + 1

    # Print the vertices at distance 2 or less from s
    for each vertex v in G:
        if v.distance <= 2:
            print(v)
```

   (a) **[3 marks]** Prove that this algorithm takes $\Omega(|V| + |E|)$ in the worst case.
   You may **not** assume that BFS runs in $\Omega(|V| + |E|)$ time. You should analyse this algorithm directly, although you may repeat the ideas in our BFS analysis from the course.

Solution

Consider a graph that is connected, i.e., there is a path from any vertex to any other vertex. The initialization of the `enqueued` attribute takes $\Omega(|V|)$ time. We know that every vertex will be enqueued, because BFS visits every vertex that is connected to the starting vertex. For a current vertex v in the body of the outer loop, the inner loop runs for d_v iterations (the degree of v), and so the total number of iterations here is $\Omega(|E|)$, leading to a running time of $\Omega(|V| + |E|)$.

(b) **[3 marks]** Let n_s be the number of vertices in G that are within distance 2 of vertex s. Show how to implement PrintClose with worst-case running time $\mathcal{O}(n_s^2)$ (i.e., runtime depends only on the number of vertices within distance 2 of s). Remember that you should not print out the same vertex more than once.

Give both **pseudocode** as well as **brief English justification** about why your algorithm is correct.

You may not use any auxiliary attributes other than the ones used in the previous algorithm, and/or one to keep track of whether a vertex has already been printed. Clearly state how these attributes must be initialized, but do **not** include the cost of initializing these attributes in your algorithm analysis.

Hint: you can use the same basic idea as BFS, but don't visit all vertices.

Solution

We can use the exact same algorithm, except stop enqueuing vertices at distance 2. This algorithm visits only the vertices at distance 2 or less than from the starting vertex.

```
1  def PrintClose(graph, s):
2      # Run BFS with distances
3      queue = new empty queue
4      initialize all vertices in the graph to not enqueued
5
6      queue.enqueue(s)
7      s.enqueued = True
8      s.distance = 0
9
10     while queue is not empty:
11         v = queue.dequeue()
12         print(v)
13
14         # Only enqueue neighbours if at distance 0 or 1.
15         if v.distance < 2:
16           for each neighbour u of v:
17               if not u.enqueued:
18                   queue.enqueue(u)
19                   u.enqueued = True
20                   u.distance = v.distance + 1
```

[Comment: another common approach is to explicitly loop through the neighbours of the neighbours of s, using an attribute to keep track of which ones had already been printed.]

(c) **[2 marks]** Prove that your algorithm from the previous part always runs in time $\mathcal{O}(n_s^2)$.

Solution

The first key idea is that only n_s vertices are enqueued. The inner loop only runs when v is either s itself, or a neighbour of s. In both cases, v's neighbours are all at distance at most 2 from s, and so the number of neighbours is at most n_s. So the number of inner loop iterations is at most n_s, and the outer loop happens (exactly) n_s times, for a total running time of $\mathcal{O}(n_s^2)$.

Use this page for rough work.

Use this page for rough work.

Name:

	Q1	Q2	Q3	Total
Grade				
Out Of	7	7	8	22