

UNIVERSITY OF TORONTO  
Faculty of Arts and Science

Midterm 2, Version 1  
CSC263H1F

Friday November 11, 2016, 10:10-11:00am (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.** The three parts can be done in any order.

- (a) **[2 marks]** Suppose we have an empty hash table of length  $m > 4$  that resolves collisions using closed addressing (linked list chaining), and we randomly choose **four** distinct keys to insert into the hash table. Assume that for *each* of the four choices, and every index  $0 \leq i < m$ , the probability that the chosen key hashes to  $i$  is  $\frac{1}{m}$ . **Find the exact probability** that all linked lists in the hash table are length 0 or 1 after four keys are inserted.

**Solution**

The four keys must have four distinct hash values.  $m$  choices for the first key,  $m - 1$  for the second, etc:

$$\frac{m}{m} \cdot \frac{m-1}{m} \cdot \frac{m-2}{m} \cdot \frac{m-3}{m} = \frac{m(m-1)(m-2)(m-3)}{m^4}.$$

- (b) **[2 marks]** David says that for a hash table of size  $m$  using chaining, “the worst-case running time for SEARCH when there are  $n$  keys is  $\mathcal{O}(1 + \alpha) = \mathcal{O}(1 + \frac{n}{m})$ .” If you believe he is correct, perform a *upper bound* analysis on the worst-case running time of this operation. If not, perform a *lower bound* analysis on the worst-case running time.

**Solution**

This is incorrect (that’s the bound for an average-case analysis, under the simple uniform hashing assumption).

For a lower bound, consider a hash table with  $n$  keys that all have the same hash value, and so are all in the same linked list. Searching for the last key in the chain takes  $\Omega(n)$  time, as the entire linked list (of length  $n$ ) must be traversed.

ItemID 19473

- (c) [3 marks] Suppose we have a hash table of length  $m$  that resolves collisions using open addressing, and that currently stores  $n$  key-value pairs.

Suppose we want to support the operation `INVERSESEARCH`, which is given a hash table and value, and returns a list of all keys in the hash table which correspond to that value.

Give both the **pseudocode** for an implementation of this algorithm, as well as a **worst-case upper bound analysis** for your algorithm. Your upper bound should be tight, but you do not need to prove this.

Your solution will be graded on both correctness and efficiency.

### Solution

This algorithm relies on two ideas: looping through array elements, and the fact that hash tables store both the key and value when pairs are inserted.

```
1 def InverseSearch(H, value):  
2     keys = []  
3     for i from 0 to H.length - 1:  
4         if H[i].value == value:  
5             add H[i].key to keys  
6     return keys
```

For running time, the loop runs  $m$  times, and each iteration takes constant time, for a total running time of  $\mathcal{O}(m)$ . [Comment: in fact, the loop always runs  $m$  times, regardless of the contents of the hash table or value being searched for. So the running time of this algorithm is always  $\Theta(m)$ .]

ItemID 19473

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

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

```

1 def DFS(graph, s):
2     initialize all vertices in the graph to not started
3     DFS_helper(graph, s)
4
5 def DFS_helper(graph, v):
6     v.started = True
7     Visit(v) # do something with v, like print out its label
8     for each neighbour u of v:
9         if not u.started:
10             DFS_helper(graph, u)

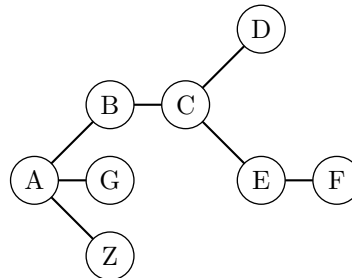
```

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:

```

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

```

Consider running this 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, G, Z, C, D, E, F. 3 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 B, G, and Z are visited matters; Z is guaranteed to be visited before all other vertices.

There is a  $\frac{1}{3}$  chance that Z is picked first among these three, a  $\frac{1}{3}$  chance Z is picked second, and a  $\frac{1}{3}$  chance Z is picked third. The expected number of vertices visited before Z is

$$\mathbb{E}[T_{BFS}] = \frac{1}{3} \cdot 1 + \frac{1}{3} \cdot 2 + \frac{1}{3} \cdot 3 = 2.$$

- (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 again depends only on the order in which B, G, and Z are chosen, with the difference being that when B is visited, all of {C, D, E, F} are also visited before the next neighbour of A. Skipping some justification:

$$\begin{aligned} \mathbb{E}[T_{DFS}] &= 1 \cdot \Pr[T_{DFS} = 1] + 2 \cdot \Pr[T_{DFS} = 2] + 6 \cdot \Pr[T_{DFS} = 6] + 7 \cdot \Pr[T_{DFS} = 7] \\ &= 1 \cdot \frac{1}{3} + 2 \cdot \frac{1}{6} + 6 \cdot \frac{1}{6} + 7 \cdot \frac{1}{3}. \end{aligned}$$

### 3. Graph Search Applications. Part (a) can be after (b) and (c).

A **triangle** in a graph is a set of three vertices that are all neighbours of each other (i.e., a cycle of length 3). Suppose we are given a graph  $G = (V, E)$  and vertex  $v \in V$ , and want to determine whether  $v$  is part of a triangle. Here is a brute force way of doing this:

---

```

1 def FindTriangle(graph, v):
2     for each neighbour u of v:
3         for each neighbour w of v:
4             if u != w and (u, w) is an edge:
5                 return True
6     return False

```

---

(a) [3 marks] Prove that FINDTRIANGLE takes  $\Omega(|V|^2)$  time in the worst case, assuming  $|E| = \Omega(|V|)$ .

Assume that checking whether  $(u, w)$  is an edge always takes constant time (so this isn't the slow part).

#### Solution

For an input family, consider a graph with vertex  $v$  that is connected to every other vertex, and there are no other edges. So  $v$  is not part of a triangle, but running FINDTRIANGLE on this graph and  $v$  will require  $|V| - 1$  iterations of outer loop, and  $|V| - 1$  iterations of the inner loop, for a total running time of  $\Omega(|V|^2)$ . [Comment: this is where we use the fact that there are  $\Omega(|V|)$  edges. If there were fewer, e.g., 10, then this algorithm would take  $\Omega(|E|^2)$  time instead.]

**BFS Algorithm** (reference for part (b)).

---

```

1 def BFS(graph, s):
2     queue = new empty queue
3     initialize all vertices in the graph to not enqueued
4
5     queue.enqueue(s)
6     s.enqueued = True
7
8     while queue is not empty:
9         v = queue.dequeue()
10        Visit(v) # do something with v, like print out its label
11
12        for each neighbour u of v:
13            if not u.enqueued:
14                queue.enqueue(u)
15                u.enqueued = True

```

---

- (b) [3 marks] Show how to implement FINDTRIANGLE in worst-case time  $\mathcal{O}(|V| + |E|)$ . Give both **pseudocode** as well as **brief English justification** about why your algorithm is correct. You may again assume that it is always possible to check whether a given tuple  $(v_1, v_2)$  is an edge in constant time.

**Hint:** Recall how we analysed BFS in lecture. You can use the main idea from the previous algorithm, and/or from BFS itself.

### Solution

One solution would be to take the idea of BFS, except explicitly only explore up to distance 2. In this case, if a vertex has an “enqueued” neighbour not equal to  $v$ , this must form a triangle.

```

1 def FindTriangle(graph, v):
2     queue = new empty queue
3     initialize all vertices in the graph to not enqueued
4
5     for each neighbour u of v:
6         queue.enqueue(u)
7         u.enqueued = True
8
9     # Run BFS on the neighbours
10    while queue is not empty:
11        w = queue.dequeue()
12
13        for each neighbour u of w:
14            # Rather than enqueue more vertices,
15            # simply check if it's been enqueued.
16            if u.enqueued:
17                return True
18
19    return False

```

A cute simplification of this idea makes it obvious that it is a rather basic variation of the given algorithm:

```

1 def FindTriangle(graph, v):
2     for each neighbour u of v:
3         for each neighbour w of u:
4             if v != w and (v, w) is an edge:
5                 return True
6     return False

```

- (c) [2 marks] Show that your algorithm runs in  $\mathcal{O}(|V| + |E|)$  time in the worst case.

You may **not** assume that BFS runs in  $\mathcal{O}(|V| + |E|)$  time. You should analyse your algorithm directly, although you may repeat the ideas in our BFS analysis.

### Solution

Since we can't assume the running time of BFS, we have to remember how we analysed it. Initializing attributes takes  $\mathcal{O}(|V|)$  time. The inner loop runs in time  $d_u$ , the degree of the current vertex  $u$ . So the total number of iterations of the inner loop is less than or equal to the sum of all degrees in the graph, which is  $\mathcal{O}(|E|)$ . Putting these together lead to a  $\mathcal{O}(|V| + |E|)$  running time.

Use this page for rough work.

www.oxdia.com

DownloaderID 30768

ItemID 19473



Use this page for rough work.

www.oxdia.com

DownloaderID 30768

ItemID 19473

Name:

www.oxdia.com

DownloaderID 30768

	Q1	Q2	Q3	Total
Grade				
Out Of	7	7	8	22

ItemID 19473