NATIONAL UNIVERSITY OF SINGAPORE

CS5234 - Combinatorial and Graph Algorithms

(Semester 1 AY2016/17)

Time Allowed: 2 Hour

Instructions

a. Write your Student Number below, and on every page. Do not write your name.

b. The assessment contains **FIVE** multi-part problems (and one additional sixth problem just for fun). You have **120** minutes to earn **100 points**.

c. The assessment contains 18 pages, including the cover page and 4 pages of scratch paper.

d. The assessment is closed book. You may bring two double-sided sheet of A4 paper to the assessment. You may not use a calculator, your mobile phone, or any other electronic device.

e. Write your solutions in the space provided. If you need more space, please use the scratch paper at the end of the assessment. Do not put part of the answer to one problem on a page for another problem.

f. Show your work. Partial credit will be given. Please be neat.

g. You may use (unchanged) any algorithm or data structure that we have studied in class, without restating it. If you want to modify the algorithm, you must explain exactly how your version works.

h. Draw pictures frequently to illustrate your ideas.

i. Good luck!

Student Number: _____

EXAMINER'S USE ONLY			
Question		Mark	Score
1	True, False, Explain	20	
2	Search, Search, Search	18	
3	Parallel Search	18	
4	B-linked Stack	22	
5	Interstellar Friendship	22	
	TOTAL	100	

Problem 1. True, False, and Explain. [20 points]

This problem consists of a set of true/false questions. For each question, circle true or false and briefly (in *at most* one sentence) explain your answer.

Let A be a randomized algorithm that returns an integer between 1 and N. On each execution, it returns the correct answer for your problem with probability $> 2/3$.

Assume Algorithm B consists of the following:

> Execute algorithm A for $(\log(1/\epsilon)/\log(3))$ times, where $0 < \epsilon < 1$; return the maximum value produced by algorithm A during any of the executions.

Then Algorithm B is guaranteed to return the correct answer with probability at least $1 - \epsilon$.

TRUE **FALSE**

Solution: **False**: the maximum value returned by A is meaningless. There is no reason to believe that the correct answer is the maximum.

In MapReduce, the Map function takes as input only one key/value pair.

TRUE **FALSE**

Solution: **True**: this is a restriction in the MapReduce model.

Recall the streaming algorithm (that we studied in class) which uses graph sketches to determine the number of connected components in a graph. This algorithm is a good design choice for graphs where the maximum degree $d = O(1)$.

TRUE **FALSE**

Solution: **False**: the streaming algorithm uses $\Omega(n \log^c n)$ space, while simply storing all the edges would take only $O(n)$ space.

Recall the sampling algorithm (that we studied in class) for determining the approximate number of connected components in a graph. This algorithm is a good design choice for graphs where the average degree $\bar{d} = \Theta(n)$.

TRUE **FALSE**

Solution: **False**: the running time is $O(d/\epsilon^3)$ and d is large in this case.

Problem 2. Searching, Searching, Searching. [18 points]

For each of the following algorithms, specify the proper asymptotic running time from the following list. Write the letter in the box. For each algorithm, assume we are in the external memory model with memory size M, block size B, and the cost metric is the number of block transfers.

Note: $\log_B(n/B) = \Theta(\log_B n)$. *When no base of the log is specified, we always mean* $\log_2$.

A. $\Theta(\log n)$	B. $\Theta(\log(n/B))$	C. $\Theta(\log_B n)$
D. $\Theta((1/B)\log n)$	E. $\Theta((1/B)\log(n/B))$	F. $\Theta((1/B)\log_B n)$
G. $\Theta(\log_B n + (\log^2 n)/B)$	H. None of the above.	

B	Binary search in an array.
A	Searching for an element in a balanced binary search tree, e.g., a red-black tree or an AVL tree.
B	Searching for an element in a standard buffer tree (with buffers of size B, node degree 2 or 3, and leaves of size B).
C	Searching for an element in a buffer tree with buffers of size B, node degree $\Theta(\sqrt{B})$, and leaves of size B.
C	Searching for an element in a B-tree.
C	Search for an element in a (static) cache-oblivious search tree.

Problem 3. Parallel Search. [18 points]

Consider the following parallel algorithm written in the fork-join model:

Algorithm 1: ParallelSearch(A, i, j, key)

```
1 if i = j then
2     if A[i] = key then return true
3     else return false
4 else
5     mid = ⌊(i + j)/2⌋
6     in parallel
7         a_1 = ParallelSearch(A, i, mid, key)
8         a_2 = ParallelSearch(A, mid + 1, j, key)
9     return (a_1 or a_2)
```

Assume the array A has size n and we execute ParallelSearch$(A, 1, n, key)$ for some key. All answers for this question can be given with asymptotic notation, with bounds as tight as possible.

Problem 3.a. What is the **work** of the algorithm, as a function of n? $O(n)$

Briefly explain your answer with a recurrence:

Solution: $W(n) = 2W(n/2) + O(1) = O(n)$

Problem 3.b. What is the **span** of the algorithm, as a function of n? $O(\log n)$

Briefly explain your answer with a recurrence:

Solution: $s(n) = S(n/2) + O(1) = O(\log n)$

Problem 3.c. What is the **parallelism** of the algorithm, as a function of n? $O(n/\log n)$

Briefly explain your answer:

Solution: $P(n) = W(n)/S(n) = O(n/\log n)$

Problem 4. B-Linked Stack. [22 points]

In this problem, we are going to analyze a cache-efficient stack built using a linked list. The linked list is singly-linked (i.e., each node has a *next* pointer), and each node stores a collection of keys. More specifically, each node contains the following fields:

next	a pointer, which indicates the next node in the linked list;
num	an integer, which indicates the number of keys stored in the node;
keys	a fixed size array of size $2B$ that is used for storing keys.

To ensure that the linked list is cache-efficient, we will maintain the following properties:

a. The first node in the linked-list has at most $2B$ keys.
b. Every other node in the linked-list has exactly B keys.

We will refer to the first node in the list as the head. Both PUSH and POP have access to the *head*. Assume the *head* node exists but is not loaded in memory when the execution begins.

For this problem, consider a model with blocks of size B and memory size $M \geq 32B$. All costs are with respect to the number of block transfers.

The pseudocode for push and pop are described on the next page.

The problem continues on the next page.

Algorithm 2: push(key)

```
1  // Increment the count and add the key to the head node:
2  head.num = head.num + 1
3  head.keys[head.num − 1] = key
4
5  // Check if the node is too big:
6  if (head.num = 2B) then
7      // Split the head node:
8      snode = new node // Create a new node
9      snode.next = head.next
10     head.next = snode
11     snode.num = B
12     head.num = B
13     // Divide the keys between the old and new node, preserving stack order
14     for i = 0 to B − 1
15         snode.keys[i] = node.keys[i] // Copy key to new node
16         node.keys[i] = node.keys[i + B] // Move key forward
```

Algorithm 3: pop()

```
1  // Check if there is anything in the stack:
2  if (head.num = 0) then return EMPTY
3  // Choose an item to return from the top of the stack:
4  key = head.keys[head.num − 1]
5  head.num = head.num − 1
6
7  // If the current node is empty:
8  if (head.num = 0) then
9      // Delete the head node and replace it with the next node (if there is a next node):
10     if head.next exists then head = head.next
11 return key
```

Problem 4.a. [6 points] Your goal here is to analyze the cost of performing k operations.

Let $k \geq 1$ be an arbitrary integer, and let $S = \langle s_1, s_2, \ldots, s_k \rangle$ be an arbitrary sequence of k push and pop operations.[1]

What is the cost for executing the sequence S as a function of k, M, and B? (Recall, the cost is measured in terms of block transfers, and you may use big-O notation. When using big-O notation, give bounds as tight as possible and do not treat B or M as a constant: for example, $O(B) \neq O(1)$.)

$O(k/B + 1)$

Problem 4.b. [10 points] Prove that your answer from Part (a) is correct:

Solution: First, we recall the invariant stated above: the head node always has at most $2B$ keys, while every other node has exactly B keys. This can be proved by induction.

Next, consider the sequence S. We can assume that the memory is large enough (i.e., M is sufficiently big) to store the entire head node. The head node is loaded for the first time on the very first operation. Once the head node has been loaded, it will remain in memory until it is deleted, which occurs only when the head node becomes empty. The other cost we have to account for is inserting a new node, when the head node is split.

Let S' be the subsequence of operations $s_{i_1}, s_{i_2}, \ldots$ where each s_{i_j} is either the first operation (i.e., s_1), an operation in which the head node is split, or an operation in which the head node is deleted and reset to a non-null next pointer.

Notice that these operations in S' are the only ones that we need to pay for. All the rest of the operations are free, since they only access the head node which is already in memory. Moreover, each of these operations has cost $O(1)$, as it requires transferring $O(1)$ blocks to load the head node or to create a new node. (Each block consists of $O(1)$ blocks, and is of total size $O(B)$.) Thus the total cost of the sequence S is $O(|S'|)$.

Next, notice that between any pair of operations s_{i_j} and s_{i_j+1}, there must be at least $B - 2$ more operations. Consider the three possible cases:

- s_{i_j} is the first operation in the sequence: There are no other nodes in the list yet, so operation s_{i_j+1} must be a split. After operation s_{i_j}, there is only one key in the head

[1] Typically, we would just be interested in amortized performance. For technical reasons, here we ask about the cost of a sequence of k operations, for any integer k.

node. Thus there cannot be a split until there are at least $B-1$ more push operations, hence there must be at least $B-2$ intervening operations.

- s_{i_j} is a split operation: A split operation only occurs when there are exactly $2B$ keys in the head node, so after the split, there are exactly B keys in the head node. The next split will not occur until there are $2B$ keys in the head node, i.e., until at least B more push operations. The next operation deleting the head will not occur until there are 0 keys in the head node, i.e., until at least B more pop operations. Hence operation s_{i_j+1} cannot occur until after at least $B-1$ more intervening operations.
- s_{i_j} is an empty operation: An empty operation sets the head equal to the next node in the list. By the invariant discussed above, that next node has exactly B keys, and so the head has exactly B nodes after the empty operation. As in the previous case, the next split will not occur until there are $2B$ keys in the head node, i.e., until at least B more push operations. The next operation deleting the head node will not occur until there are 0 keys in the head node, i.e., until at least B more pop operations. Hence operation s_{i_j+1} cannot occur until after at least $B-1$ more intervening operations.

We conclude that between each of the events in S', there are at least $B-2$ operations. From this we conclude that $|S'| \leq k/(B-2)+1 = O(k/B+1)$.

Notice that in this case, the $+1$ is important. It is incorrect to say that the running time is $O(1/B)$. For example, if $k=1$, then $O(k/B) = O(1/B)$. Notice that $O(1/B)$ is smaller than $\Theta(1)$, and yet the cost of the sequence is ≥ 1.

Problem 4.c. [6 points] Joe Programmer decides to use this stack to implement depth-first-search (DFS) on a graph $G = (V, E)$ starting from source s as follows:

Algorithm 4: DFS$(G = (V, E), s)$

```
1 Let stack be a B-linked-stack, as described above.
2 stack.push(s)
3 while stack is not empty
4     u = stack.pop()
5     visit(u) // Do whatever you need to do at u
6     for each neighbor v of u
7         stack.push(u)
```

Assume that the stack supports checking whether it is empty in zero (0) time, because that information is always stored in main memory.[2] Assume that $visit(u)$ requires only $O(1)$ cost. Let $COST(k)$ be the cost for executing a sequence of k push and pop operations, as per Part (b).

Joe claims that the DFS algorithm presented above correctly visits every node in DFS order in time $O(n + COST(m))$ on a graph with n nodes and m edges. What is wrong with Joe's claim?

Solution: It never checks whether a node is visited before adding it to the stack. Hence it never terminates (and visits nodes too many times).

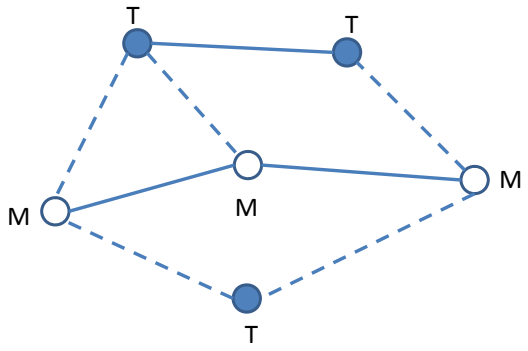
[2] This is not necessarily exactly true, but assume it for now.

Problem 5. Interstellar Friendship. [22 points]

Assume you have a graph $G = (V, E)$ representing a social network: each node $u \in V$ represents a person and each edge $(u, v) \in E$ represents a friendship. Each user has *exactly* k friends. Let $n = |V|$ be the number of people in the network, and let $m = |E|$ be the number of friendships in the network. (Assume that you know n and m.) Each person in this network is also either a Martian (from Mars) or a Terran (from Earth).

Assume you are given the graph as an adjacency list: for each user u you have access to the list $friends[u]$ specifying the friends of u. Also, you have an array P that specifies which planet each user comes from, e.g., $P[u] = T$ if user u is Terran, and $P[u] = M$ if user u is Martian.

Design an algorithm to estimate how many interplanetary friendships there are, i.e., how many pairs $(u, v) \in E$ are there where u is Terran and v is Martian or vice versa. For example, in the following graph (where T represents a Terran and M represents a Martian) there are five interplanetary friendships (represented by the dashed lines):



Design your algorithm to be as fast as possible, asymptotically, while guaranteeing that with probability at least 2/3, it returns an estimate that satisfies the following guarantee: Let C be the actual number of interplanetary friendships, and let D be the output of your algorithm, then: $|D - C| \leq \epsilon m$.

For this problem, please give the complete solution/analysis, i.e., do not just reduce the problem to one that we solved in class. (You may of course use theorems that we used or proved in class, e.g., general theorems regarding probability.)

Problem 5.a. [7 points] Describe your algorithm. (If you use random sampling, be sure to specify how algorithmically the random item is selected.)

Solution: Let S be a randomly chosen set of $\Theta(1/\epsilon^2)$ edges in the adjacency list. Each edge is chosen by first choosing a random user with probability $1/n$, and then a random friend of that user with probability $1/k$. Let $S' \subseteq S$ be the subset of edges that are interplanetary. Return $(m \cdot |S'|/|S|)$.

Problem 5.b. [3 points] What is the running time of your algorithm as a function of the parameters n, m, k, and ϵ? (You may use big-O notation.)

$O(1/\epsilon^2)$

Problem 5.c. [10 points] Show that your algorithm guarantees the requisite property, e.g., with probability at least 2/3, $|D - C| \leq \epsilon m$, where D is the output of your algorithm and C is the actual number of interplanetary friendships.

Solution: Let x_i be the random variable representing whether sample i is interplanetary. Notice that $\mathrm{E}[x_i] = C/m$, and hence $\mathrm{E}[\sum(x_i)] = |S|(C/m)$ by linearity of expectation. Since $|S'| = \sum(x_i) = |S|(C/m)$, the expected return value is exactly $D = |S|(C/m)(m/|S|) = C$.

Similarly, if $\sum(x_i) \leq |S|(C/m) + (\epsilon|S|)$, then the return value is $\leq C + \epsilon m$, and if $\sum(x_i) \geq |S|(C/m) - (\epsilon|S|)$, then the return value is $\geq C - \epsilon m$. Thus we want to find the probability that $|\sum(x_i) - |S|(C/m)| \leq \epsilon|S|$, which indicates the probability that the algorithm is correct.

Since $\mathrm{E}[\sum(x_i)] = |S|(C/m)$, we apply a Hoeffding bound to calculate the probability that this does not occur:

$$\begin{aligned}
\Pr[|\sum(x_i) - |S|(C/m)| \geq \epsilon|S|] &\leq 2e^{-2(\epsilon^2|S|^2)/|S|)} \\
&\leq 2e^{-2\epsilon^2|S|} \\
&\leq 2e^{-2\epsilon^2(2/\epsilon^2)} \\
&\leq 2(1/2^4) \\
&\leq 1/8 \\
&\leq 1/3
\end{aligned}$$

Hence we have shown that with probability at least 2/3, $\Pr[|\sum(x_i) - |S|(C/m)| \leq \epsilon|S|]$, and hence $|D - C| \leq \epsilon m$.

Problem 5.d. [2 points] Assume, instead, that each user has at least 1 friend and at most k friends, where k is relatively small compared to n. (Recall in the previous parts we had assumed that each user had exactly k friends.) Assume that you are also given an array $numFriends$ that indicates the number of friends of each user, i.e., $numFriends[u]$ is the number of friends that user u has.

Explain briefly how you would modify your algorithm to cope with this change, and why your modification works. (You only need to state the part that changes.) What is the expected running time of the revised algorithm as a function of the parameters n, m, k, ϵ?

Solution: We still want to sample edges randomly from the graph. Unfortunately, since users no longer have a uniform number of friends, it is not as easy to find a random edge.

We use the following idea to find a random edge: choose a random user u and a random integer i in the range $[1, k]$. If user u has at least i friends, then return the ith friend in the adjacency list for u. Otherwise, return nothing.

Notice that in this case, each edge is selected with probability $2/(nk)$: for edge (u, v), with probability $1/n$ we select u and with probability $1/k$ we select the correct index for edge (u, v); with probability $1/n$ we select v and with probability $1/k$ we select the correct index for edge (u, v). Thus each edge is chosen with equal probability.

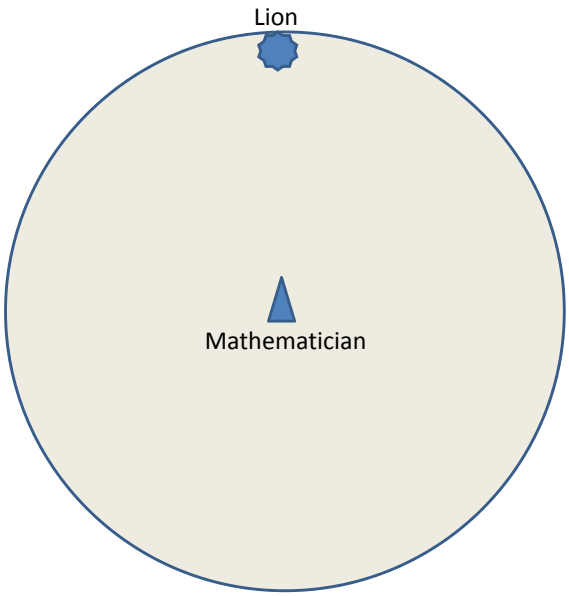
With some probability, however, the sampling procedure returns nothing. Specifically, with probability $m/(nk)$ it returns an edge and with probability $1 - m/(nk)$ it returns nothing.

We repeat this sampling procedure until we find $2/\epsilon^2$ edges, and use this as the random sample in the algorithm from the previous section. Since each edge is chosen with equal probability, this sample satisfies the required randomness.

For each edge, we can expect to perform nk/m samples in order to find one edge. Thus to find $2/\epsilon^2$ edges will take, in expectation, $O((nk/m)(1/\epsilon^2) \leq O(k/\epsilon^2)$ time.

Problem 6. The Mathematician and the Lion (*Just for Fun*) [0 points]

A mathematician is trapped in a circular arena with a lion. She is fast: she can run just as fast as the lion. And she has the stamina to continue fleeing for infinite time. She starts in the middle of the arena, while the lion begins at the gate on the north edge. Can she escape the lion forever? She is successful as long as she can maintain a distance > 0 from the lion forever.



Scratch Paper

Scratch Paper

Scratch Paper

Scratch Paper

End of Paper