

Name: Solution

E-mail: virgi@cs.cmu.edu, chengwen@cs.cmu.edu

Instructions:

- This is a closed book exam. No calculators or laptops are allowed. Absolutely no collaboration.
- Do all your work including scratch work on the pages of this exam.
- There are 16 pages in total, including this page.
- The exam is graded out of 100 points.
- If you use anything not proven in class, please provide a proof. Briefly explain any pseudocode you provide. Always state the running time of your algorithms.
- For this exam assume all mentioned graphs are undirected, simple (no duplicate edges and no self loops), connected and (unless stated otherwise) m and n are the number of edges and vertices respectively.

1 “Old” Problems

1. (5 pts) In approx. 100 words or less, describe how to find a $(1/2, 1/2)$ planar (vertex) separator of size $O(\sqrt{n})$ in linear time. You may assume the existence of a linear time $(1/3, 2/3)$ planar separator algorithm.

Answer: Shown in homework solution.

2. (5 pts) In approx. 100 words or less, prove that the expected fraction of time spent on every edge in a random walk on any (undirected simple connected) graph is $1/m$.

Answer: Shown in class.

2 Data Structures

1. (5 pts) For n elements x_i drawn from a totally ordered set, let $rank(x_i)$ be the position of element x_i in a sorted list of the n elements.

Design an efficient (randomized) data structure, S , to support the following operations.

- $Insert(S, x)$: inserts x into S .
- $Delete(S, x)$: deletes x from S . (You may assume a pointer to x is given.)
- $FindAny(S)$: return any element x in S together with its $rank(x)$ in S .

For example, the elements in S might be English words, and their rank is determined by the lexicographical ordering: if we have *car*, *cat* and *ant*, then *car* will have rank 2, *cat* will have rank 3 and *ant* will have rank 1.

An example sequence of operations on S , starting with $S = \emptyset$:

- $Insert(S, cat)$
- $Insert(S, car)$
- $Insert(S, ant)$
- $FindAny(S) \rightarrow (cat, 3)$ [or $(car, 2)$, or $(ant, 1)$]
- $Delete(S, ant)$
- $FindAny(S) \rightarrow (cat, 2)$ [or $(car, 1)$]
- $Delete(S, cat)$
- $Insert(S, bag)$
- $FindAny(S) \rightarrow (car, 2)$ [or $(bag, 1)$]
- $FindAny(S) \rightarrow (bag, 1)$ [or $(car, 2)$]
- $FindAny(S) \rightarrow (bag, 1)$ [or $(car, 2)$]

Answer: Use a binomial heap (or binary heap, balanced binary search tree, treap, splay tree) to obtain $O(\log n)$ for Insert and Delete, and $O(1)$ for FindAny (returns the minimum and its rank 1). (If you use splay trees, then you need to maintain the size of subtree to find the rank of a node efficiently.)

2. (15 pts) Prove that either $\text{Insert}(S, x)$, $\text{Delete}(S, x)$, or $\text{FindAny}(S)$ must spend $\Omega(\log n)$ in the worst case (in a comparison based model).

Answer: Proof by contradiction. We can sort n elements, $x_1, x_2 \dots x_n$ in the following manner,

- Create an array of size n , calls SORTED
- For $i = 1$ to n , $\text{Insert}(S, x_i)$
- For $j = 1$ to n
 - $\text{FindAny}(S) \rightarrow (x, k)$
 - Put x in the k th empty location in SORTED array.
(Note: this step takes $O(n)$)
 - $\text{Delete}(S, x)$
- return SORTED

This above algorithm actually use $O(n^2)$ time. However, the $O(n^2)$ time comes from comparing the ranks, not the elements of x . All the comparison between elements of x comes from the data structure. Since the data structure only spend little $o(n \log n)$. This violates the sorting lower bound.

3. (20 pts) A method to defeat the lower bound is to restrict the power of the adversary. A *weak* adversary must provide the sequence of operations in advance (before any of them are to be performed by the data structure). The data structure however does *not* know the *entire* sequence of operations, but performs the operations one by one, following the sequence. The difference here is that the sequence of operations is independent of what the data structure returns - the adversary cannot adapt.

Design an efficient (randomized) data structure to support the operations above assuming a weak adversary. (For a sequence of n Insert, n Delete and n FindAny, your data structure must run in expected or amortized little $o(n \log n)$).

Answer: Main idea: the data structure maintains a single uniformly random element, y and $r = \text{rank}(y)$. And a linklist of all the elements in S .

- $\text{Insert}(S, x)$: Append x into the linklist. If $x < y$, then $r = r + 1$. Moreover, we probability $1/|S|$, change the element we maintain by setting $y = x$, and compute the rank of y .
- $\text{Delete}(S, x)$: If $x < y$, then $r = r + 1$. If $x = y$, then randomly pick another element to be the new y , and compute the new r .
- $\text{FindAny}(S)$: returns (y, r) .

FindAny always takes $O(1)$ time. With probability $1/|S|$, Insert or Delete takes $O(|S|)$ time. Otherwise, Insert or Delete takes $O(1)$. So it is expected constant time per operation.

4. **[Bonus]** (30pts) Prove that for any deterministic data structure supporting Insert, Delete and FindAny, if Insert and Delete take little $o(\log n)$, then FindAny must take $\Omega(n^{1-\epsilon})$, for constant $\epsilon > 0$.

Answer:

As appeared in homework 2 solution, "The proof of $\Omega(n^{1-\epsilon})$ lower bound ... can be found in the paper 'The Randomized Complexity of Maintaining the Minimum' by Gerth Stolting Brodal. (Section 4: Deterministic lower bound for FindAny)". In fact, the whole question comes directly and *intentionally* from that paper.

3 Proof or “Spoof”

In this section you are required to check the correctness of a proof or an algorithm.

1. (5 pts) Here is an algorithm to find all the connected components of a graph $G = (V, E)$. If the algorithm is correct, give a short explanation, and if it is not correct, pinpoint the errors and suggest a way to correct them.

Input: an array with all the vertices V , and edges E , represented as adjacency lists.

Output: each vertex in the same component gets the same label, and vertices in different components gets different labels.

```
Connected_Com
  for i = 1 to |V|
    visited[V[i]] = false
  for i = 1 to |V|
    label[V[i]] = i
    DFS(V[i])
```

```
DFS(v)
  if (visited[v]) return
  visited[v] = true
  for all neighbors u of v
    if (!visited[u])
      label[u] = label[v]
      DFS(u)
```

2. (5 pts) Here is an algorithm to compute the shortest paths from a given vertex s to all other vertices in the graph, assuming all edge weights are 1. If the algorithm is correct, give a short explanation, and if it is not correct, pinpoint the errors and suggest a way to correct them.

Input: an array with all the vertices V , and E represented as adjacency lists, and a starting vertex s .

Output: the shortest distance from s to every vertices.

ShortestPath(s)

```
for i = 1 to |V|
    dist[V[i]] = -1

dist[s] = 0
Stack.push(s)

While(!Stack.empty())
    v = Stack.pop()
    for all neighbors u of v
        if (dist[u] == -1)
            dist[u] = dist[v] + 1
            Stack.push(u)

Loop

return dist
```

3. **[Bonus]** (10 pts) Consider the following question: Let T be the cover time of a graph G (the maximum, over all start vertices, expected time to visit all vertices of G using a random walk.) Suppose we add an edge to the graph without increasing the number of vertices. Can T increase by such an operation?

Here is an argument that T cannot increase. If the argument is correct, complete the proof with details, and if it is not correct, either suggest a way to correct it, or show that the statement is false and T can increase.

Argument: Adding more edges increases the connectivity of the graph so the random walk will visit more and more vertices faster. Hence T can only decrease.

4 Easy or Hard

1. Here we investigate two problems on Vertex Cover and triangles. (A triangle in a graph is a cycle of length 3.)

- (a) (15 pts) Consider the following **Triangle Free Vertex Cover** decision problem (TFVC): Let G be a graph containing no triangles. Let $K > 0$. Does there exist a vertex cover of G of size $\leq K$?

Show TFVC is NP-complete.

(Hint: try a local transformation of each edge.)

- (b) (15 pts) A graph is *triangulated* if it is simple, connected and *every* cycle $C = c_1, \dots, c_k, c_1$ of length at least 4 has an edge (c_i, c_j) with c_i and c_j **not** adjacent in the cycle.

Consider the following **Triangle Vertex Cover (TVC)** *optimization* problem: Let G be triangulated. Find a minimum vertex cover of G .

Give an efficient 1.5-approximation algorithm.

2. (10 pts) Here is an interesting property of SAT:

*For any CNF formula, at least half of the assignments satisfy at least half of the clauses.
(We assume that each clause contain at least 1 literal.)*

Give a proof of the property.

(Hint: The problem and solution (a one-liner) were discovered by Ryan Williams, while he was a freshman at Cornell.)

3. **[Bonus]** Consider the following **Faculty Hire Problem** (FHP):

The school of CS has a pool S of n job candidates. The school has m departments. Each department i is considering a subset $S_i \subseteq S$ of candidates (these subsets may overlap), and needs to hire exactly n_i of the candidates it is considering. Given S , S_i and n_i , does there exist an assignment of candidates to departments so that a candidate is employed by at most one department and for each $1 \leq i \leq m$, department i gets n_i employees from its interest pool S_i ?

(a) (5pts) Is FHP NP-complete, or is there a polynomial time algorithm for it?

- (b) (25pts) If you think the problem is NP-complete, give a high level proof; If you think the problem is in P, give a high level description of your algorithm. No rigorous proof required.

Extra Page, intentionally left blank.