CPSC 320 Midterm 2
Friday, November 10th, 2006

[15] 1. The president of a large Game Development company has decided to invite as many of his employees as possible to a party at this house on Christmas Eve[1]. To make sure that the party is a success, he decides that none of the employees he will invite should ever have had a disagreement with any other one of the invited employees. He asks you for help in selecting the employees that will attend the party.

[8] a. Design a greedy algorithm that takes as input a set E of employee names, and a list L of the pairs of employees that have had a disagreement, and returns as large a subset S of the set of employees as possible, with the constraint that for all $x \in S$ and $y \in S$, $\{x, y\} \notin L$.

Solution : There might be several different solutions that will attempt to return the largest set of employees that have never disagreed with one another. Here is are two reasonably simple (and greedy) ones:

i. Sort employees by the number of other people they have disagreed with, and add employees to S in increasing order. Employees that have disagreed with an employee already in S are discarded.

ii. Use the same approach, but whenever an employee is added to S, all employees that disagree with him are eliminated, and then we decrement the value associated with each employee that disagreed with *them*.

The first approach is simpler, so I will describe the second one (it's easy to modify it to get the same behaviour as with the first approach).

Assume we have are given the graph $G = (E, L)$, represented by its adjacency list. We will add to each vertex a pointer to the corresponding node in a heap H. This will allow us (1) to update the heap when we decrement the value associated with a given vertex, and (2) to delete a selected vertex from the heap. Both operation will take time in $O(\log s)$ where s is the size of the heap. The algorithm then proceeds as follows:

```
Algorithm SelectEmployees(E, L)

  S ⟵ ∅

  for i ⟵ 0 to length[E] - 1 do
    Value(E[i]) ⟵ number of neighbours of E[i]

  Build min-heap RemainingEmployees from E

  while RemainingEmployees is not empty do
```

[1] He owns a *very* large house.

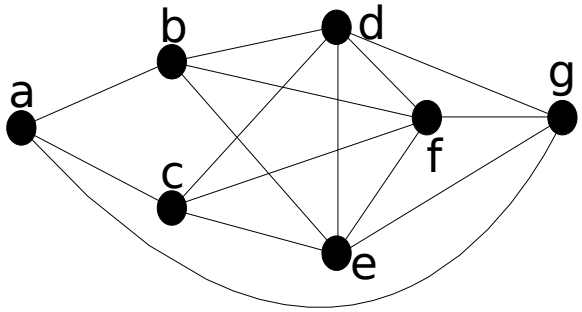
```
    employee ⟵ DeleteMin(RemainingEmployees)
    add employee to S
    for each neighbor notliked of employee do
      for each neighbor otherneighbor of notliked do
        decrement Value(otherneighbor) and update heap
      delete notliked from heap

  return S
```

[3] b. What is the running time of the algorithm you described in part (a), as a function of $|E|$ and $|L|$?

Solution : The total number of heap operations performed is: $|E|$ insertions, $|E|$ deletions, and at most $2|L|$ updates when decrementing values (since the total sum of the number of neighbours of all the vertices is $2|L|$. So the running time is in $O((|E| + |L|) \log |E|)$.

[4] c. Give one example where your algorithm from part (a) does not find the largest possible subset of employees satisfying the constraints (that is, say what $|E|$ and $|L|$ are, show the subset returned by your algorithm, and then show a bigger subset satisfying the constraints).

Solution : Here is a counter-example that works for both version of the algorithm. In both cases, vertex a has the fewest neighbours, and is chosen first. The algorithm then can not choose vertices b, c or g, and will pick exactly one of d, e, f (since they form a triangle in the graph). So it returns a set S with 2 employees only. However $\{b, c, g\}$ is a solution with 3 employees.



[18] 2. Short answers

[6] a. Algorithm DeterministicSelect uses as pivot the median of the medians of groups of 5 elements each. A student who finds it too complicated decides instead to use as pivot the median of a random group of $n/5$ elements. Will the student's algorithm run in $O(n)$ time? Explain why or why not.

Solution : No, it won't. If the random group of $n/5$ elements consists exactly of the $n/5$ largest elements, for instance, then the pivot chosen will be the $n/10$th largest

element, and DeterministicSelect might end up recursing on $9n/10$ elements. The worst-case running time is thus described by the recurrence

$$T(n) = T(n/5) + T(9n/10) + \Theta(n).$$

when n is large. Because $n/5 + 9n/10 = 11n/10$, the work done on each row of the recursion tree will be an increasing geometric function, and so $T(n) \notin O(n)$.

[3] b. Randomized Quicksort is an algorithm similar to Quicksort, that uses a random element as a pivot instead of the element at a specific position in the array. These two algorithms have the same worst-case running time ($\Theta(n^2)$) and the same average-case running time ($\Theta(n \log n)$). Why, then, is Randomized Quicksort better than Quicksort?

Solution : Quicksort has bad inputs: inputs where it will *always* run in $\Theta(n^2)$ time (for instance, a sorted array). There are no bad inputs for Randomized Quicksort: its average-case running time (for every possible inputs) is always $\Theta(n \log n)$.

[3] c. Under which circumstances, if any, do we change the "level" of an already existing node in a skip list?

Solution : Never: we set the level of the node when it is created (that is, when the element it contains is inserted into the skip list), and it remains fixed until the node is deleted.

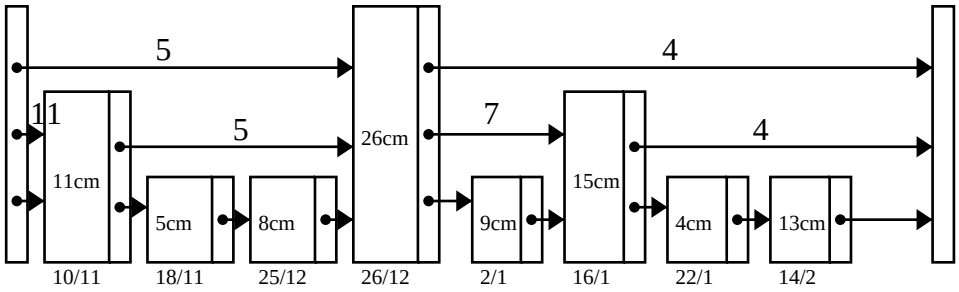
[3] d. A computer scientist needing to perform efficient queries on a property of the elements in a skip list decides to augment this skip list by adding an extra piece of information $\mathcal{X}$ to each pointer in the skip list. What two properties does $\mathcal{X}$ need to possess?

Solution : $\mathcal{X}$ needs to be useful in answering the queries efficiently, and we also need to be able to update it when new elements are inserted in or deleted from the skip list, without affecting the running time of these two operations.

[3] e. Why is an adjacency list data structure better suited to the Prim-Jarník algorithm than an adjacency matrix? Hint: think of an operation that is more efficient for adjacency lists.

Solution : The Prim-Jarník algorithm needs to go through all of the neighbors of the vertex that is being added to the tree. This operation is much more efficient with an adjacency list data structure ($\Theta(1)$ per neighbor) than with an adjacency matrix ($\Theta(n)$ even if there aren't any neighbors).

[12] 3. A meteorologist studying snow storms in Winnipeg is using a skip list to store pairs of the form (`day`, `amount`), as shown in the figure below. `Day` is the day of the year a snow storm occurred (written below each node in the figure), and `amount` is the quantity of snow that fell during the snow storm (written inside each node). The elements of the skip list are sorted by day.



The meteorologist then decides to augment the skip list by storing with the pointer going from node N_1 to node N_2 the smallest amount of snow that occurred in a snow storm stored in any one of the nodes that comes after N_1, up to and including node N_2.

[4] a. Add to the figure the values stored on all pointers at levels 2 and 3 in the augmented skip list (just write them on top of the arrows).

[8] b. Describe how you would delete the pair for a date `day` from the augmented skip list.

Solution : We start by deleting the node and updating the skip list the same way we would do it for an un-augmented skip list. Then we have to find the new value that goes on each pointer. We do this as follows: first, observe that the only pointers whose value needs to be updated are those starting at nodes in the `Pointer` array. More specifically, we need to update the pointer at level `lev` that starts at node `Pointer[lev]`.

So, for each level `lev` from 1 to `maxLevel(S)`, we will recompute the value `minSnow(Pointer[lev], lev)` by traversing the list at level `lev - 1`, from node `Pointer[lev]` to the node its pointer at level `lev` points to, and finding the smallest value stored in one of the level `lev - 1` pointers.

In other words, this is the same as dealing with deletions with the `maxGap` problem from assignment #6.