CS 4800: Algorithms Take-home Final

Karl Lieberherr

April 25^{th} 2010

```
Out April 26, 9 am
In  April 27, 5 pm (by email to me)

Take home final may be done in pairs of 2 students:
the same pairs you did the homework with.
You can also do it individually.

No other communication about the exam is allowed, except
with your partner or with me.

If something is not clear in the exam, make a reasonable assumption,
state the assumption you made and solve the problem under
that assumption.

Use of web for browsing is allowed.

Sign the following statement at the beginning of the exam:
I have followed the rules of this exam.

Please bring a copy of the acknowledgement page that you have done
the TRACE survey to the oral final.

The Oral Final Schedule:
http://www.ccs.neu.edu/home/lieber/courses/algorithms/cs4800/sp10/final-schedule
```

1 DAG question, 20 points

Develop an efficient algorithm for the following problem: Does a DAG contain a directed path that goes through each node once?

Argue for the correctness of your algorithm and analyze the running-time of your algorithm.

2 Leaf Covering: Decision ⇒ Search (40 points)

2.1 Explanation

Remember the leaf covering problem? Here's a reminder, with some background, though it may be a little different than you remember it. We are given n trees, $(T_1, \ldots, T_n)$ over the alphabet Σ, and a set of n-tuples, $M \subseteq (\Sigma \times \cdots \times \Sigma)$. Each of the trees is defined with a successor function, that returns the set of symbols that are *children* of a given node:

$$succ : \Sigma \to \mathcal{P}(\Sigma)$$

And a function *root*, that returns the root symbol of a tree:

$$root : Tree \to \Sigma$$

We will also use $succ^*$ to represent the *transitive closure* of *succ*. Each *succ* (and $succ^*$) is technically a partial function.

We define the function *leaves* that returns the set of symbols (nodes) in a tree that have no successors:

$$leaves(T) \equiv \{ \sigma \in \Sigma \mid succ(\sigma) = \emptyset \}$$

Decision Problem: *DLC* Given the n trees, define the set of n-tuples, *Leaves*, as the cartesian product of the leaves of our trees:

$$Leaves = leaves(T_1) \times \cdots \times leaves(T_n)$$

determine whether each n-tuple in *Leaves* has an n-tuple in M whose elements are ancestors of the corresponding element in the leaf. More formally:

$$\begin{array}{l} DLC(T_1, \ldots, T_n, M) \equiv \\ \quad \forall (\sigma_1, \ldots, \sigma_n) \in Leaves \; . \\ \quad\quad \exists (m_1, \ldots, m_n) \in M \; . \forall i \in [1..n] \; . \; \sigma_i \in succ^*(m_i) \end{array}$$

In other words, for each n-tuple in *Leaves*, there's an n-tuple in M with elements that *cover* the leaf elements.

Search Problem: *SLC* Previously it wasn't known whether or not a solution to the decision problem (DLC) could be used to find an n-tuple in *Leaves* that is uncovered (a witness). We've been working at it for some time, and have developed an algorithm that might work:

```
SLC(T_1, ..., T_n, M) ≡
    m[] := (root(T_1), ..., root(T_n))
    for i = 1 to n; do
        foreach ℓ ∈ leaves(T_i); do
            m[i] := ℓ
            if ( ¬DLC(T_1, ..., T_n, M ∪ {m}) ) then
                next i
    return m
```

We use the decision procedure as a *black-box*, and attempt to find a witness that shows not all leaves are covered. First, we initialize a local variable m starting at the *root*-tuple. Note that by definition, initially $M \cup m$ covers all leaves. Then, for each tree, we loop through its leaves, and try each one in the i^{th} position by calling DLC adding the updated tuple to M. If DLC returns *false* then we keep that element, and move to the next position (written "`next` i"). Of course, if there is no next element, we fall out from the for loop.

2.2 Questions

1. What is the running time of our SLC algorithm, assuming the running time of DLC is $O(1)$, (constant time)? Use t as a bound on the size of each tree (and so the *leaves* of each tree), n for the number of trees, and $|M|$ for the size of M.

2. Though at first glance this algorithm seems promising, it is not clear whether or not it works correctly. Argue that the algorithm **is** or **is not** correct. That is, when (and if) $DLC(T_1, \ldots, T_n, M)$ returns *false* (meaning an uncovered leaf exists), then $SLC(T_1, \ldots, T_n, M)$ returns an uncovered leaf (*correct* behavior) or a covered leaf (*incorrect* behavior).

   If you believe it is correct, be convincing. If not, then you should be able to provide a simple counter example where the algorithm gives the wrong answer. It may help to visualize or discuss the idea of a *graph cartesian product*, that was mentioned earlier in the semester, though not required. And, remember that `foreach` can select elements in *any* order.

3 k-th Largest Element (30 points)

In class we discussed several deterministic algorithms for selecting the k-largest element of a set of numbers, including the median of the median algorithm. In section 13.5 of the text book, immediately following the section on Randomized Approximation, that we covered, there is a randomized algorithm for solving the k-th largest element problem. You only need to read the first 5 pages of the section, up to the middle of page 731.

Answer the following questions:

1. What are the advantages and disadvantages of the randomized algorithm over a deterministic algorithm for the same problem?

2. Why is choosing a random splitter working well?

3. Where is linearity of expectation used in the analysis of the algorithm? What is the definition of linearity of expectation?

4. Describe a situation where the randomized algorithm would run slower than O(n).

4 Highest Safe Rung Recurrence (20 points)

Highest Safe Rung: Chapter 2, exercise 8.

Develop a recurrence involving a minimization step for the minimum number of drops $HSR(n, k)$ as a function of k (the number of jars allowed to break) and n, the number of rungs.

$HSR(n, 1)$ is given directly by a simple function. Express $HSR(n, 2)$ in terms of minimizing a function involving $HSR(n, 1)$. Express $HSR(n, 3)$ in terms of minimizing a function involving $HSR(n, 2)$.

It is enough to give the recurrence only for these two special cases with an explicit derivation of the solution of the recurrence.