CORE EXAMINATION
Department of Computer Science
New York University
February 8, 2008

This is the common examination for the M.S. program in CS. It covers core computer science topics: Languages and Compilers, Operating Systems, and Algorithms. The exam has two parts. The first part lasts three hours and covers the first two topics. The second part, given this afternoon, lasts one and one-half hours, and covers algorithms.

Use the proper booklet for each question. Each booklet is marked with the Area and Question number in the form PL&C1, PL&C2, PLC&C3, OS1, OS2, ALGS1, ALGS2, ALGS3. Use the appropriate booklet for each question. DO NOT put your name on the exam booklet. Instead, your exam number must be on every booklet.

You will be graded according to your exam number, shown on the envelope containing the booklets. Remember your exam number: when grades are given out, they will be published according to this number, not by name.

Make sure your name and signature are on the envelope. This is the only place where your name appears. Please include all the booklets inside the envelope. You can keep the exam.

Good luck!

Algorithms

Question 1

This question concerns the enhancement of search trees to compute specialized queries about the stored data.

a. [4 points] Briefly explain how to enhance a 2-3 Tree or some other efficient search tree so that the following operations all run in $O(\log n)$ time for a structure with n records:

i. $Insert(x)$: inserts a record with search key x.

ii. $Find(x)$: returns a pointer to a record with search key x if there is such a record.

iii. $Delete(x)$: deletes a record with search key x from the data structure.

iv. $\#LEQ(x)$: returns the number of records with search keys less than or equal to x.

It is not necessary to explain why the first three operations still run in $O(\log n)$ time, but you do have to explain what is needed to ensure that the additional operation iv also runs in $O(\log n)$ time.

Solution

Enhance a 2-3 Tree so that each internal node v maintains a count of the leaves in the subtree rooted by v. For expositional completeness, let each leaf have a count of one. Then $\#LEQ(x)$ is the sum of the counts in the roots of all leftward siblings of vertices on the path from the root of the tree to the virtual leaf x if x is not in the structure, and one more than this count to the actual leaf x if x is in the structure. The wording leftward sibling is intended to denote the zero, one, or possibly two subtree roots (or possibly leaves) that are siblings of a 2-3 Tree node v and that hold keys that are smaller than the keys in the subtree rooted by v.

These counts are easy to maintain during an insertion or deletion operation for 2-3 Trees. In general, an insertion or deletion causes an increment or decrement of the count for each non-leaf node on the path from the root to the affected leaf. The new counts for a node that is a merge of two nodes or the two nodes that result from a split of one node can be computed as the sum of the counts of the children nodes.

It is clear that the operations for the above two paragraphs take constant time per node, and so each of the four operations in part (a) requires $O(\log n)$ time.

---

b. [6 points] Now let S be a set of intervals where an interval $[u, w)$ represents the points v in the x-axis for which $u \leq v < w$. Note that two overlapping intervals are just that: a pair of intervals and not their union. So for example if S contains $[1, 7)$ and $[3, 9)$ then S is different from a set T that just contains $[1, 9)$. In particular, S has two intervals that contain $[3, 7)$, whereas T has just one. You are to explain how to maintain a set S of (possibly overlapping) intervals as detailed below.

Explain how to implement a data structure to support the following operations in $O(\log n)$ time for a set of n intervals:

i. $Insert[u, w)$: inserts the interval $[u, w)$.

ii. $Delete[u, w)$: deletes the interval $[u, w)$, if present.

iii. $StabbingNumber(v)$: returns the number of intervals $[u, w)$ in the structure that contain v (i.e., the number of intervals where $u \leq v < w$).

Simplifications: You can assume that the data structure never contains more than one interval $[u, w)$ with the left endpoint value u.

Once again, it is not necessary to explain why the first two operations still run in $O(\log n)$ time, but you do have to explain how to implement operation iii in a way that is equally efficient.

Hint for computing $StabbingNumber(v)$: If you imagine scanning your eyes from left to right over the intervals in the data structure, then $StabbingNumber(v)$ will equal the number of intervals $[u, w)$ where you have seen the starting values u where $u \leq v$ but not the termination values w where $v < w$.

Hint: The data structure and data management for part a) is useful for part b).

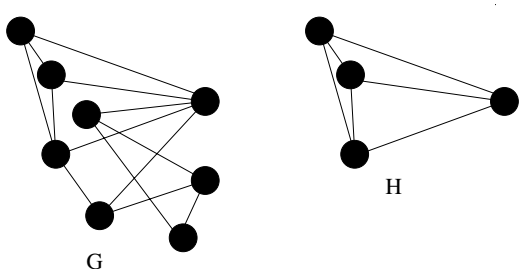
Solution

Use the structure of part (a). An interval $[u, w]$ is inserted into the 2-3 Tree by inserting a start record with the key u, and an end record with the key w. The u key must have some accompanying information to record its endpoint w. For the end record with key w, it suffices to keep a count of the number of end records with the key w, and the start record for u might include a field for w. [If multiple intervals with the same start value are permitted, then each start record should be stored as a pair (u, w) with u as the major key and w as the minor key.]

A natural solution for computing $StabbingNumber(v)$ is to maintain two counts in each internal node: the number of records with start key values in the subtree, and the number of records with termination key values in the subtree. $StabbingNumber(v)$ is the difference between the number of start values less than or equal to v, and the number of termination values less than or equal to v.

Question 2

Let $G = (V, E)$ be an undirected graph, and k an input value that is a positive integer. You are asked, in the three parts below, to present **in pseudocode** an algorithm that constructs the subgraph $H = (W, F)$, where the vertex set W is a subset of V, and the edge set F is a subset of E, each vertex in H has at least k neighbors in H, and H is as large as possible. An example of G and the solution subgraph H for $k = 3$ is shown on the right.



Hint: a first step is to remove from G each vertex v that has fewer than k neighbors, and to account for each neighbor's loss of an edge to v.

For full credit, your algorithm should have an operation count of $O(|V| + |E|)$.

a. [5 points] Present, in the pseudocode of your choice, an algorithm to construct the vertex set W. You can assume that $V = \{1, 2, \ldots, n\}$.

Solution

Let $Degree[i]$ store the number of neighbors of vertex i.
$Degree[1..n]$ can be trivially set to the correct counts for G in $O(|V|+|E|)$ operations by initializing all entries to zero, sequencing through all edges $\{a, b\}$ in E, and incrementing $Degree[a]$ and $Degree[b]$ for each such edge. Then the algorithm is:

```
create a set ToBeRemoved that is initially empty;
for all vertices i do
    set i.belongs to true;
    if Degree[i] < k then insert i into the set ToBeRemoved endif
endfor;
while ToBeRemoved not empty do
    v ← remove from ToBeRemoved;
```

```
        v.belongs ← false;
        for each neighbor w in Adj[v] do
            Degree[w] ← Degree[w] − 1;
            if Degree[w] = k − 1 then insert w into ToBeRemoved endif
        endfor
    endwhile;
    create the initially empty set W;
    for i ← 1 to n do
        if i.belongs then insert i into W endif
    endfor;
```

b. [2 points] Briefly explain why your algorithm is correct.

Solution

This greedy algorithm removes only vertices that must be removed from G. That is, at each iteration, the vertices inserted into $ToBeRemoved$ must be removed from G if the final solution is to have no vertex with fewer than k neighbors. So each removal is a correct step. The algorithm terminates only when all remaining vertices have degree k or more, so at termination the solution must be correct. Finally, the algorithm terminates because no more than $|V|$ removals can occur, which ensures that the while loop has no more than $|V|$ iterations, and all other loops have a fixed number of iterations.

c. [3 points] Assume that W has been constructed correctly. Present, in the pseudocode of your choice, an algorithm to construct the set of edges F for H.

Solution

To recover the edges, a sufficient processing procedure is:

```
create a set F that is initially empty;
mark the .belongs fields for all x in W as true, and set the field to false for all other vertices.
foreach edge {a, b} in E do
    if a.belongs and b.belongs then insert {a, b} into F endif
endfor;
```

Question 3

Let $M_1, M_2, \ldots, M_n$ be a sequence of n matrices, and let $D[1..n+1]$ be an array of $n+1$ values where M_j has the dimensions $D[j] \times D[j+1]$. The following high-level recursive specification computes the least number of scalar multiplications needed to compute the matrix product $M_i \times M_{i+1} \times \cdots \times M_j$:

$$Least(i,j) = \begin{cases} 0 & if\ i = j; \\ \min\Big\{ Least(i,i) + Least(i+1,j) + D[i] \times D[i+1] \times D[j+1], & \\ \quad Least(i,i+1) + Least(i+2,j) + D[i] \times D[i+2] \times D[j+1], & \\ \quad Least(i,i+2) + Least(i+3,j) + D[i] \times D[i+3] \times D[j+1], & \\ \quad \vdots & \\ \quad Least(i,j-1) + Least(j,j) + D[i] \times D[j] \times D[j+1] \Big\} & if\ i < j. \end{cases}$$

a. [4 points] Use the pseudocode of your choice to present an algorithm that computes $Least(i,j)$ in an efficient manner.
Suggestion: present the algorithm in a way that is easily (or already) enhanced to help solve part b). See the hint below.

Solution

```
global D[1..n+1], Look[1..n, 1..n], Split[1..n, 1..n];
for i ← 1 to n − 1 do
    for j ← i + 1 to n do
        Look[i, j] ← nil
    endfor
endfor;
function best(i, j);
    if i = j then temp ← 0
    elseif Look[i, j] = nil then
        temp ← ∞;
        for k ← i to j − 1 do
            if temp > best(i, k) + best(k + 1, j) + D[i] ∗ D[k + 1] ∗ D[j + 1] then
                temp ← best(i, k) + best(k + 1, j) + D[i] ∗ D[k + 1] ∗ D[j + 1];
                split[i, j] ← k    //Useful for part b) //
            endif
        endfor;
        Look[i, j] ← temp
    else
        temp ← Look[i, j];
    endif;
    return(temp)
end_best;
```

---

b. [6 points] Use the pseudocode of your choice to present an algorithm that builds and returns the binary parse tree for the product $M_1 \times M_2 \times \cdots \times M_n$ that corresponds to the solution $Least(1, n)$ from part (a). Each leaf in the tree is a matrix M_i, and each internal node represents the binary operator "×," and has left and right subtrees. If you wish, you can use the pseudocode $p \leftarrow NewNode$ to create these internal nodes. Acceptable assignments might be written as:

$$p.left \leftarrow q \quad \text{and} \quad q.left \leftarrow \text{pointer to } M[j], \quad \text{etc.}$$

Hint: Augment your solution in part (a) to record the information needed to determine the tree structure (i.e., the split of interval (i, j) that is used to compute $Least(i, j)$, for each pair (i, j)).

Solution

```
best(1, n);
ParseTree ← buildtree(1, n);
procedure buildtree(i, j);
    if i = j then p ← pointer to M_i
    else
        p ← NewNode;
        p.left ← buildtree(i, split[i, j]);
        p.right ← buildtree(split[i, j] + 1, j)
    endif;
    return(p)
end_buildtree;
```