CSE 101 Class Notes Today: Graphs, Recursion vs. Iteration

April 27, 2004

Breadth- and Depth-First Search

Problem: Given a directed graph $G = (V, E)$, find the set of nodes reachable from some node x.

In doing this, we want to avoid visiting a node more than once, so we need to keep track of these visited nodes. Initially, `Visited[x]` is `false` for all x. We also need to track the set F of "frontier" nodes, which we have seen (as neighbors of a processed node) but have not yet processed.

Algorithm

```
1     Frontier <- { x }
2     Visited[x] = T
3     forall y != x
4         Visited[y] = F
5     while Frontier not empty
6         y <- Choose(Frontier)
7         for z in Adj(y)
8             if !Visited[z]
9                 Insert(Frontier, z)
10                Visited[z] = T
11        Remove(Frontier, y)
```

What data structure should we use? It must support (1) inserting an element, (2) choosing an arbitrary element, and (3) removing that element. Both stacks (push, top, pop) and queues (enqueue, head, dequeue (or "decapitate"?)) support these operations in constant time.

Running Time

After a node is added to F, it is marked `Visited`. No visited node is added to F. Since a `Visited` mark is never removed, this implies that each node is added to F at most once. Therefore the loop at line 7 is done at most once per node y.

This loop takes $O(\deg(y))$, since it is executed once for each edge from y. The sum of all nodes' out-degrees in a directed graph is $|E|$. Therefore the total running time for lines 7-10 is $O(\sum_{v \in V} \deg(v)) = O(|E|)$.

The remaining lines are executed $|V|$ times, and take constant time. Therefore the total running time is $O(|V| + |E|)$.

Note that this running time is the same whether we use a stack or a queue. However, which we use will determine whether we will execute a depth-first (stack) or breadth-first (queue) search. Using a stack, however, we can gather additional information by remembering the order in which we insert and remove items. In particular, no node is removed from the stack until all its children have been removed. This order of removal (children before parents) is referred to as *topological order*, and the algorithm reporting the depth-first search finish order is called *topological sort*.

Recursion vs. Iteration

On a different subject, here we explore the relationship between recursion and iteration, using as an example the simple problem of finding the maximum of an array. This case is a particular example of how, in general, correctness easier to show for recursive algorithms, while running time is easier to find for iterative ones.

Iterative

```
1     max <- A[1]
2     for i in 2..size(A)
3         if A[i] > max
4             max <- A[i]
5     return max
```

Time: Clearly $O(n)$ – we can verify this by counting the number of loop iterations, and by noting that each operation within the loop takes constant time.

Correctness: First, we need to find a helpful loop invariant. Applying the "cleverness" technique, we find that after loop k, `max` is the maximum of `A[1..k]`. We then need to prove (by induction) that this invariant is satisfied on every iteration of the loop. We also must show that the invariant's being satisfied on the last iteration implies that our algorithm is correct, i.e. that `max` is the maximum element in `A[1..n]`.

Recursive

```
1     max(A[1..n]) =
2         if n == 1 return A[1]
3         else
4             m <- max(A[1..n-1])
5             if   m > A[n]  return m
```

```
6            else          return A[n]
```

Correctness: Easy – because we don't have to come up with a loop invariant. The shape of the inductive proof is present in the algorithm: line 2 is our base case, and lines 3-6 are our inductive case. We just need to run through this proof to show that the function `max` works.

Time: Harder – while all the basic steps (except line 4) take constant time, it's not nearly as obvious how many times the body executes (i.e. how many total recursive calls are made). Usually, to find a recursive algorithm's running time we have to formulate and solve a recurrence. In the example above:

- $T(1) = c$ (i.e. line 2)
- $T(n) = T(n-1) + d$ (i.e. line 4 plus lines 5-6)

Using induction, we can show that

$$T(n) = c(n-1) + d$$

for constants c and d, and therefore that the algorithm is $O(n)$.