Recitation 7 — Midterm debrief

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

September 21, 2011

Today's Agenda:

- Announcements
- Staging
- Exam debrief
- - Strongly connected components
  - Priority queue as sorted list + Merge
  - Graph array representation
  - Two recurrences, and what it means to be leaf/root-dominated or equal?

1 Announcements

- Assignment 5 is due Monday Oct 17, 2011 - 11:59pm
- Some feedback back from the survey.
- Comments about the exam?
- Questions about homework, class?

2 Staging

In homework 4, where we asked you to write algorithm for computing shortest paths, we asked you to write a *staged* function with type:

```
val query : thesaurus -> string -> string -> string Seq.seq option
```

This was the function for computing path from a word to another through the graph of thesaurus connections.

Now most of you got the actually shortest path computation right, but had not figured out how to make it *staged*.

Q: What does it mean for a function to be staged? When is staging useful?

A: Staging is useful when a computation contains a (complex) component that can be reused. For example, if you first compute shortest path tree from word "GOOD", you can use the same tree for computing paths to "BAD", "EVIL" and "CARROT". In this case, computing the tree is the hard part and the rest is just a table lookup.

Here is one way to write the stageable version of query:

```
fun queryStaged (oracle : oracleOfKevin) (actor1 : string)   =
let
    val tree = SP.makeTree oracle actor1
in
```

```
        SP.reportPath tree
    end
```

Q: What's the return type of this function? **A:** If we apply it to all its explicit arguments, the type is `string -> string Seq.seq option`.

We can stage this by bind this partial application, which does the work in `m`akeTree, and use that binding for fast repeated queries.

2.1 Another staging example

Let's now work quickly a simple staging example. Consider you want to implement function that returns the nth largest value from an unsorted int sequence. Here is the type:

```
val nthLargest :  int Seq.seq -> int -> int
```

Non-stageable implementation:

```
fun nthLargest s n =  Seq.nth (Seq.sort Int.compare s)  n
```

To see why this isn't stageable, let's move the second argumant to an inner function:

```
fun nthLargest s = fn n => Seq.nth (Seq.sort Int.compare s)  n
```

And consider what happens if we apply this function to some sequence e.g. `<4,5,3,7>`:

```
- val app = nthLargest <4,5,3,7>;
val app = fn n => Seq.nth (Seq.sort Int.compare <4,5,3,7>) n
: int -> int
```

Now every time we call `app` on some number, the `sort` function is invoked.

Instead, we can precompute this.

Q: How would we write a stageable version of `nthLargest`?

A:

```
fun nthLargestStaged  s =
let
    val presorted = Seq.sort String.compare s
in
    fn n => Seq.nth presorted n
end
```

Notice how the `sort` computation is no longer *guarded* by a function binding (`fn`). This means that when we apply it to a sequence e.g. `<4,5,3,7>`, we get

```
- val app = nthLargest <4,5,3,7>;
val app = fn n => Seq.nth <3,4,5,7> n : int -> int
```

Note that we can rewrite the last line of our staged function to

```
Seq.nth presorted
```

which is exactly equivalent, but the staging is clearer to see with the explicit binding.

Q: And how do we use the staged function?

A: We first call it without the n-argument:

```
val f = nthLargestStaged S;

val i = f 0;
val j = f 1;
val k = f 2;

etc...
```

Note: This can be called a high-order function, as it returns another function.

Q: Can you give examples of staged functions in our library?

A: map, scan, reduce, are *curried* functions. Staged functions are curried functions, but staging implies that the first stage performs some serious computation. In our library, there are not really staged functions, because it does not include complex algorithms.

3 Midterm debriefing

3.1 Strongly Connected Components

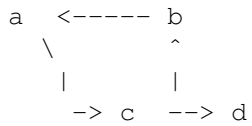
This is question 5 from the mid-term. So given a function to find reachable vertices from a vertex `v`, how can we find the strongly connected component that contains `v`

Q: Most of you got this, so please someone give the answer?

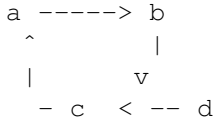
The idea was to find the set of vertices that are reachable from `v`in the original graph [let this be A] and the set of vertices reachable from `v` in the *reversed* graph [let this set be B].

Then the strongly connected component is $A \cap B$.

We can work this out on a simple graph (sorry about poor ascii graph):



and reversed graph is:



Let's find the SCC of "a". (Work this on the board).

Q: What is A? What is B?

A: A = a, c, b, d; B = b, c, a

And it is easy to see the intersection is "a,b,c" as expected.

In the heat of the exam, it might become mind-boggling to figure out whether it is union or intersection - this was a common mistake to use union.

I would like to give a good general way of how to figure out these kind of questions: make up the *simplest possible* problem and find out if your hypothesis works.

What would be the simplest case here? Perhaps a chain, and you try to find the SCC of the middle vertex:

```
a -> b -> c
```

Clearly you see the SCC of "b" is just $\{b\}$. If you take union, you will get $\{a, b, c, \}$, so now you see it cannot be union.

Similar simple trick works often, if you cannot memorize if something was something or another.

3.2 Priority queue as sorted list + Merge

Priority queue question in the exam turned to be quite easy for most of you. Some of you seem to confuse *scan* with *reduce*, though. Remember that scan generates all the interim answers! If you only need the last answer, you don't want to scan.

Last question of priority queues was whether you could match the cost specifications of our meldable priority queue by representing the queue as a sorted array-sequence.

Q: Can someone quote the answer?

Let's look at the operations in turn:

- empty: trivial.
- deleteMin: This could be basically done in O(1) time using an sorted array-sequence. **Q:** how? **A:** By just updating a pointer of the top element.
- insert: this take O(n). **Q:** why? **A:** This has two parts. First you need to look for the position to insert the element. **Q':** How to do it in log(n)? **A':** binary search; Then you need to insert the element to its position. This takes O(n) because you need to create a new sequence. Even with a single-threaded array, you would need to *shift* the values, with average cost of n. Many of the students also incorrectly claimed you need to resort the array after insert!
- meld: We got very large incorrect answers for this one. It was correct to say that you cannot match the $\log^2 n$ cost of meldable queue. But many of you claimed it takes $O(n \lg n)$ to meld two sorted array sequences, because you need to sort it. **Q:** how to combine two sorted array sequences in *linear* time? (Note: our lib may still have merge implemented as sort, which is functionally correct but not cost-wise correct. This might be source of some confusion).

Important take-home message: **use merge to combine two sorted sequences in linear time**.

3.3 Graph array representation

One of the short answer questions was whether the *asymptotic* cost of Djikstra's algorithm would be improved if we would use array-representation of graph instead of a table-representation.

Most were correct to say that the asymptotic cost was not altered because the big cost comes from the use of the priority queue.

Q: What about in practice, would array-representation be faster?

A: Yes! Quite a lot actually. This is because looking up array index takes O(1). You get good performance also by using a hash table, where array-lookups are also O(1), but with a larger constant.

Some had clearly misunderstood the idea of array-representation and thought that looking up vertices would take linear time because you would need to search through the array. (**Q:** What if the array was sorted? **A:** Could use binary search.).

But the idea is that the vertex identifier (id) *equals* the array-index. I.e if we want to look for vertex 42, we would just `Seq.nth arr 42`.

The array representation of graphs is a very important topic and most large graphs are represented as such.

Q: How does this relate to the matrix representation of graphs?

A: Vertex id is just the row/column of the matrix.

3.4 Recurrences

Many students had difficulty with the recurrences, and made the same mistakes. Let's consider common form of recurrence that occur with many algorithms:

$$W(n) = aW(n/b) + \Theta(f(n))$$
$$W(1) = \Theta(1)$$

where $f(n) = n^k$, and a, b, and k are positive constants.

Three of the recurrence questions on the midterm were of this form.

need picture of the recurrence tree

A common error in expanding the recurrence tree was calculating the cost at each level of the tree.

level 0: $f(n) = n^k$
level 1: a copies of $f(n/b)$ for a cost of $a(n/b)^k = (a/b^k)n^k = (a/b^k)f(n)$.
level 2: a^2 copies of $f(n/b^2)$ for a total of $a^2(n/b^2)^k = (a/b^k)^2 n^k = (a/b^k)^2 f(n)$.
...
level i: a^i copies of $f(n/b^i)$ for a total of $a^k(n/b^i)^k = (a/b^k)^i f(n)$.
...
level d: some number of copies (that is, the number of leaves) of $f(1) = c$, where c is a constant.

(We'll discuss what the value of d is later.)

Common error: Many students dropped the k in term $(a/b^k)^i$.

Q. Why is that k important?

The term a/b^k is used to determine whether the cost at each level is *geometrically* decreasing, stays the same, or *geometrically* increasing. It is these three cases that determines whether the work is dominated at the root of the recurrence tree, is equal at each level, or is dominated by the leaves of the recurrence tree, respectively. Students who dropped the k reversed these cases.

Q. Case 1: When is the cost dominated at the root?

If $a/b^k < 1$ then the cost at each level is **geometrically decreasing**. That is, the cost at level i is a *constant factor* less than the cost at level $i+1$ for every pair of adjacent level. In particular,

$$W(n) = \sum_{i=0}^{d} = (a/b^k)^i f(n) = f(n) \sum_{i=0}^{d} (a/b^k)^i$$

Because the latter series is a *geometric* decreasing series, this sum evaluates to a **constant**. (Even the infinite series sums to a constant, $1/(1-(a/b^k))$.) Therefore, the cost is dominated at the root and $W(n) \in \Theta(f(n))$. The cost is *all* at the root, within a constant factor. (A few students made the mistake of answering $f(n) \log n$, which is reserved only for case 2 below.)

Q. Case 2: When is the cost equally divided among the levels?

When $a/b^k = 1$. Since the cost is $f(n)$ at all levels of the recurrence tree, the important question here is how many levels are there?

Q. What part of the recurrence determines the number of levels?

$W(n/b)$ only. The number of levels is $1 + \log_b n$. Thus, the $W(n) \in \Theta(f(n) \log n)$. Note that it is not important what base you use for $\log n$, since

$$\log_b n = \log_c n / \log_c b,$$

where c is a positive constant and hence $\log_c b$ is a constant. (Note that in case 3 below, the base b is required.)

Q. Case 3: When is the cost dominated by the number of leaves?

When $a/b^k > 1$. This time we have a **geometrically increasing** sequence, and it is the last term in the geometric series that dominates. (The other terms in the series sum to a constant multiple of the last term).

There are two ways to find the cost for $W(n)$. One is to compute this series explicitly, which many students laboriously did. The other is to consider the cost only at the leaves. In particular, since the cost at each leaf is $f(1) = c$, it is the *number* of leaves in the recurrence tree that is important.

Q. What part of the recurrence determines the number of leaves?

$aW(n/b)$. Since a is the branching factor at each level, and $f(n/b^k) = f(1)$ when $k = \log_b n$, the number of leaves is

$$a^{\log_b n}.$$

Since, $\log_a n = \log_b n / \log_b a$ we can simplify the above:

$$\begin{aligned} a^{\log_b n} &= a^{(\log_a n)(\log_b a)} \\ &= (a^{\log_a n})^{\log_b a} \\ &= n^{\log_b a} \end{aligned}$$

Therefore, $W(n) = \Theta(n^{\log_b a})$. Notice that here the base b in the logorithm is critical, as it is in the exponent of n and not a constant factor.

The above tree expansion approach can be used for many other recurrences. If you are able to show that the cost at each level is a constant multiple c of the previous level, you can determine whether total cost is dominated at the root or the leaves. A good strategy is to look at the ratio of the costs at adjacent levels and see if the ratio can be bounded by a constant. In particular, as we observed already, if $c < 1$, it is dominated at the root; if $c = 1$, it's equal at all levels; and if $c > 1$, it's dominated at the leaves.

3.5 Recurrences on the midterm

Let's try using this approach on two of the more difficult recurrences on the.

Try to work these out with students/

- $f(n) = 5f(n/8) + \Theta(n^{2/3})$.
  Solution: Note that $5f(n/8) = c5(n/8)^{2/3} = c(5/4)n^{2/3}$. Since $(5/4) > 1$ the recurrence is dominated at the leaves. $\Theta(n^{\lg_8 5})$ (roughly $\Theta(n^{0.77})$). Note: many ways to write the answer correctly.

- $f(n) = f(n/2) + \Theta(\lg n)$. Solution: $\Theta(\lg^2 n)$, approximately equal. Common mistake: $O(\lg n)$. It is tempting to assume that the cost of the algorithm is dominated at the top of the recurrence tree, as the cost decreases at each level. But it does not decrease *geometrically*. In this case, it is wise to solve the summation explicitly.

3.6 Leaf/root/equal

Many have trouble with the question of whether a recurrence is leaf or root dominated or the work equally distributed.

Q: First of all, why is it good to learn to identify this?

A: If you know it is leaf-dominated, you can just compute the last level of the tree; if you know it is root-dominated, just check the root?

How to identify?

This is actually related to what is known as the **master theorem**, which you may have seen before. It gives you the three cases of domination and cook-book to solve them. Instead of making you memorize the master theorem (though, you're welcome to study it on your own if you like), we suggest that you use the tree method: after writing out the tree, you will learn to see the pattern. With some practice, you'll be able to figure out the answer in no time and feel more comfortable solving unfamiliar recurrences.

- When you work out the expression of work for a level in the tree, if it is *geometrically* decreasing, it means that the work decreases *very quickly*. **Q:** Implication? **A:** Then it is root-dominated.
- When you work out the expression of work for a level in the tree, if it is *geometrically* increasing, it means that the work increases *very quickly*. Most of the work is at the bottom of the tree and it is the number of leaves that dominates the work. It is leaf-dominated.

Note: if you use the master theorem, you should check your answer using the substitution / induction method! If you compute the recurrence directly by evaluating the sum from the tree method - you are done!