Recitation 5 — DFS, BFS, Topological Sort

Parallel and Sequential Data Structures and Algorithms, 15-210 (Spring 2012)

February 15, 2012

Today's Agenda:

- Announcements
- DFS vs BFS
- Topological Sort
- Staging

1 Announcements

- Assignment 3 is due **tomorrow**, Thursday the 16th.
- Assignment 4 was released yesterday and is due next Tuesday the 21st.
- Questions about homework, class, life, universe?

2 DFS vs. BFS

Recall the DFS algorithm:

```
fun DFS(G, v) = let
  fun DFS'(X, v) =
    if (v ∈ X) then X
    else iterate DFS' (X ∪ {v}) (N_G(v))
in DFS'({}, v) end
```

Conceptually, one way to view depth-first search is that it recursivelly calls DFS on each neighboring node, first searching as far as it can in one neighbor (the depth-first part), then iterating on to the second neighbor, etc.

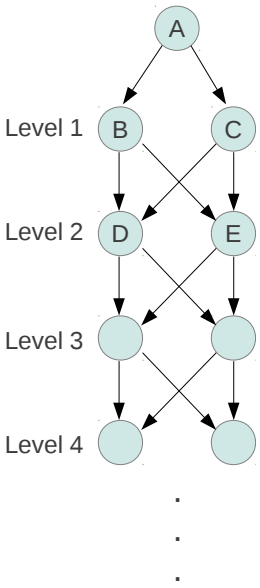
It might be tempting to then consider a similar version of BFS which instead of iteratively calling itself on each neighbor, runs BFS *in parallel* on each neighbor:

```
fun BFS(G, v) = let
  fun BFS'(X, v) =
    if (v ∈ X) then X
    else parallel BFS' (X ∪ {v}) (N_G(v))
in BFS'({}, v) end
```

Q: Unfortunately, this doesn't work at all. Why?

A: Because the calls are made in parallel, the visited sets are all independent among each parallel call now, and we can end up visiting the same node multiple times.

To see this, consider the graph:



Q: How many times will the parallel BFS visit node B or C?

A: Once.

Q: And node D or E?

A: Twice. Each node will be visited once as a neighbor of B and once as a neighbor of C.

Q: Now for some math practice. How many times would we visit a node in level i in this graph? It may be helpful to write this out as a recurrence.

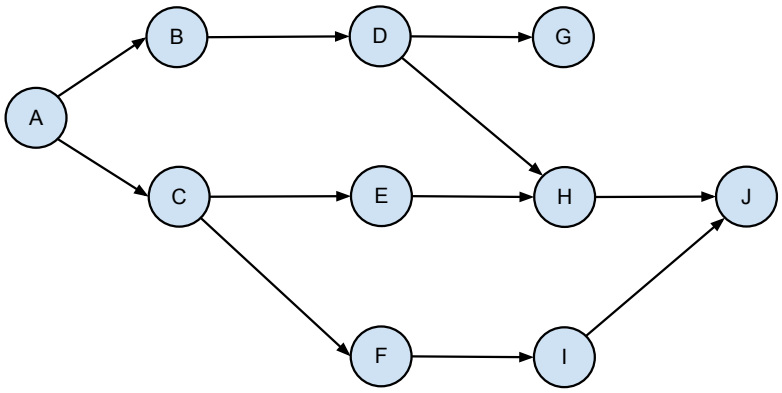
A: Let $V(i)$ be the number of times a node in level i is visited. Then $V(i) = 2V(i-1)$, with $V(1) = 1$. So, we get $V(i) = 2^i$. That's really bad!

Q: Can you think of a graph that would give even worse performance? What is the worst possible graph in terms of number of visits?

A: Consider a fully connected graph on n nodes. After starting at any node, we will end up generating every possible permutation of the rest of the vertices as a path that we search in parallel, giving $O(n^{(n-1)})$ visited nodes total. To see that this is also an upper bound and that we can't do worse, observe that we will never visit a given vertex twice in a particular chain of visits, and so any single chain of visits must correspond to a permutation of the remaining $(n-1)$ elements.

3 Topological Sort

Do example topological sort on sample graph:



4 Staging

In homework 4 you are asked to write algorithm for computing shortest paths using a *staged* function with type:

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

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

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

A: A staged function is one which takes multiple curried arguments, and **additionally** greedily performs as much computation as possible upon receiving each argument. 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.

4.1 A staging example

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

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

Non-stageable implementation:

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

In a perfect world, the SML compiler would figure out the inner expression can be evaluated without knowing the second argument, `n`. Unfortunately, this kind of analysis is difficult and the compiler doesn't do it. Let's take a look at what actually happens, if we re-write the curried function as two single argument functions:

```
fun kthElement 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 = kthElement <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 `kthElement`?

A:

```
fun kthElementStaged  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 = kthElementStaged <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 = kthElementStaged 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. For example, if we had a list of indices `K = <k1,k2,k3,...>` such that we wanted to find the kth element in a sequence `S` for each `ki`, we could simply do:

```
map (kthElementStaged S) K
```

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.

4.2 Staging Trade-offs

Q: If you remember from last week, we had a parallel version of `kthElement` that ran in $O(n)$ work and $O(\log^2(n))$ span. What would be the cost of finding m distinct kth elements from the same sequence?

A: $O(nm)$ work and $O(\log^2(n))$ span.

Q: Now, what if we use our staged version, `kthElementStaged`?

A: The one-time work of sorting the list is $O(n \log n)$, giving a total of $O(n \log n + m)$ work. The span is still $O(\log^2 n)$. Do you see why?

Q: When should we choose to use one algorithm or the other?

A: When $m > \log n$, the staged algorithm will due less work (ignoring constants).