Recitation 6 — BFS, DFS and Staging

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2013)

October 2, 2013

1 Announcements

- How did Homework 4 go?
- Homework 5 has been released.
- Questions?

2 DFS vs. BFS

Recall the algorithm for DFS starting at a node v:

1. Add v to the visited set.
2. Recurse on all out-neighbors of v that have not yet been visited.

Recall that in DFS, we want to finish searching as far as we can in one neighbor (why it's called *depth-first*) before visiting the next. Therefore, the recursive calls can't be performed in parallel. Rather, we want to call `DFS` on each out-neighbor in sequence, passing the visited set returned by one call to the next call. We can achieve this using the function `iter`, which gives the following pseudocode.

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

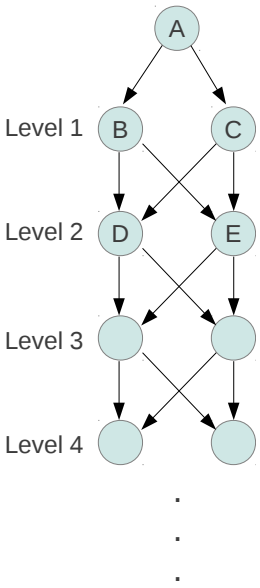
The discussion above might imply that performing the recursive calls in parallel will give an implementation of BFS, and might tempt you to write the following code to run BFS *in parallel* on each neighbor:

```
fun BFS-WRONG(G, v) = let
  fun BFS'(X, v) =
    if (v ∈ X) then X
    else reduce ∪ (map 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.

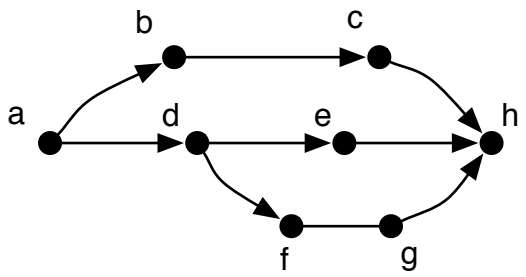
A: Let $V(i)$ be the number of times a node in level i is visited. Then $V(i) = 2V(i-1)$, with $V(1) = 1$. So, we get $V(i) = 2^i$. That's really bad!

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 Example

Below is an example of a DAG that we might want to do a topological sort over:



A possible call order would be to start the DFS on the top path. This hits all the vertices down to *h* with no branch-offs. When we exit each vertex we add it to our list of vertices, meaning that when we get back to *a*, our list is `[b, c, h]`. Before exiting *a* we need to search each of its children, so we move down to *d*. If the next path that we search down is the middle path, only *e* will be added to our list because our DFS has already touched *h*. We do the same process for the bottom path, so right before *d* exits, our list is `[f, g, e, b, c, h]`. Next we exit *d* and add it to the list, then we exit *a* and add it to the list, too. Our final list is `[a, d, f, g, e, b, c, h]`

4 Staging

Homework 5 asks you to use staging. Let's now quickly work a simple staging example. Suppose 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 argument to an inner function:

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

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:

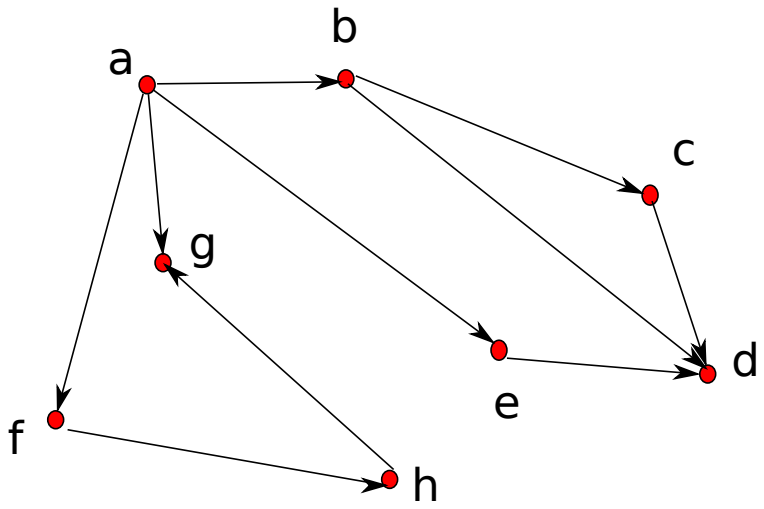
```
val f = nthLargestStaged S;

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

etc...
```

Note: This can be called a higher-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.

5 Some preparation for Homework 05



Q: What are the out-neighbors of vertex a?

A: b, e, f, g

Q: What edges would never appear in a shortest path from vertex a to another vertex?

A: Edges (c, d) and (h, g) would never appear in any shortest path, since there are shortcuts from a to d and h that avoid these edges.

Q: What is/are the shortest path(s) from vertex a to d?

A: $\langle\langle a, b, d\rangle, \langle a, e, d\rangle\rangle$

Q: What is/are the shortest path(s) from vertex a to a?

A: $\langle\langle a\rangle\rangle$