Part 5

—

Graph Traversal

CSCI 3110 Code

Summer 2015

1 Introduction

Using the graph representations from *Algos*.*Graphs*.*Graph* and *Algos*.*Graphs*.*Forest*, we now want to use them to implement a general graph traversal framework. As discussed in class, the goal of graph traversal is to build a spanning forest of the graph, which can subsequently be analyzed to extract different types of structural information about the graph.

2 Graph Traversal = Building A Forest

Most of the time, we'll care about top-down forests for further analysis, but it turns out that bottom-up forests are easier to compute, and we will need them in the context of computing shortest paths. So our goal is to develop two general-purpose graph traversal procedures; one produces a top-down forest, the other a bottom-up forest.

$$traverse :: VertexSet\ vs \Rightarrow (\forall\ s \circ Int \to ST\ s\ (vs\ s)) \to AdjList\ v\ vl\ el \to Forest\ V\ E$$
$$traverse' :: VertexSet\ vs \Rightarrow (\forall\ s \circ Int \to ST\ s\ (vs\ s)) \to AdjList\ v\ vl\ el \to UpTree\ V\ E$$

Both functions take a vertex set data structure to store candidate vertices to be explored as well as their ancestor paths in the tree.[1] Here's the interface such a data structure needs to support:

class *VertexSet vs* **where**
  $add :: vs\ s \to V \to [(E,V)] \to ST\ s\ ()$
  $remove :: vs\ s \to ST\ s\ (Maybe\ (V, [(E,V)]))$

Since we'll have to implement our traversal framework in the strict state monad, we declared our functions to be computations in this monad. We also ensure that *remove* only returns vertices that have not

[1] In class, we stored a queue or stack of edges to implement BFS and DFS and then worked our way from a priority queue of edges to a priority queue of vertices for Prim's and Dijkstra's algorithms, in order to make them more efficient. Since the latter crucially rely on having a priority queue of vertices, we work with a vertex set here and alter our BFS and DFS implementations so they also work with a queue or stack of vertices. The change is purely cosmetic for BFS and DFS.

been removed before, that is, that have not been explored before. As we will see, this will be useful to avoid duplication of work when implementing Dijkstra's and Prim's algorithms; without doing this, both the vertex set and the traversal algorithm would have to keep track of already explored vertices in these algorithms if we want to guarantee an $O(n \lg n + m)$ instead of an $O(m \lg n)$ running time.

The *traverse* function is easily implemented in terms of *traverse′* and *toForestIndexByIx*:

```
traverse makeSet = toForestIndexByIx vIx ∘ traverse′ makeSet
```

Here we make use of the fact that *toForestIndexByIx* preserves the order of the children of a node: If $v_1, v_2, \ldots, v_k$ are the children of u, then the produced *Forest* stores these vertices in the same order as they are listed in the input *UpTree*. This is important mostly for DFS because whether a forest is a DFS forest of a directed graph depends on the ordering of the children of each node. *traverse′* itself is implemented in terms of a computation *traverseM* in the *ST* monad:

```
traverse′ makeSet g = runST (traverseM makeSet (adjGraph g))

traverseM :: VertexSet vs ⇒ (Int → ST s (vs s)) → GraphStructure → ST s (UpTree V E)
```

traverseM now implements our traversal algorithm from class rather verbatim, with the exception that it returns an *UpTree*, not a *Forest*:

```
traverseM makeSet g = do s ← makeSet (gNumVertices g)
                         concat $ mapM (traverseFromVertexM s) (gVertices g)

traverseFromVertexM     :: VertexSet vs ⇒ vs s → V → ST s (UpTree V E)
traverseFromVertexM s v = add s v [ ] >> loop
  where loop = do nxt ← remove s
                  case nxt of
                    Nothing       → return [ ]
                    Just p@(v, as) → do mapM_ (λe → add s (eHead e) ((e, v) : as)) (vOutEdges v)
                                        (p :) $ loop
```

traverseFromVertexM is implemented in terms of a helper function *loop*. It first adds v to s, together with an empty ancestor path. *loop* then repeatedly removes the next unvisited vertex, adds all out-neighbours of the removed vertex to s and then reports the *UpTree* consisting of the removed vertex and the list of vertices produced by calling itself recursively.

3 Picking a Single Starting Vertex

In some algorithms, such as Dijkstra's algorithm, we may be interested in running the traversal from a single starting vertex and exploring only the part of the graph reachable from this starting vertex. To this end, we produce functions *traverseFrom* and *traverseFrom′* here:

```
traverseFrom                :: VertexSet vs ⇒ (∀ s ∘ Int → ST s (vs s)) → AdjList v vl el → V → Tree V E
traverseFrom makeSet g v    = head ∘ trees $ toForestIndexByIx vIx $ traverseFrom′ makeSet g v

traverseFrom′               :: VertexSet vs ⇒ (∀ s ∘ Int → ST s (vs s)) → AdjList v vl el → V → UpTree V E
traverseFrom′ makeSet g v = runST $ do s ← makeSet (gNumVertices $ adjGraph g)
                                       traverseFromVertexM s v
```