Part 3

—

Graph Representations

CSCI 3110 Code

Summer 2015

1 Introduction

In these notes, I try not to put too much emphasis on the fact that Haskell is a purely functional language, and we will switch back and forth between pure functions and stateful computations as necessary to implement the algorithms from class. After all, this is an algorithms class and not a programming languages class.[1] However, programming in a functional language, even one like OCAML or Scheme where the two programming styles can be mixed at will without the need to switch in and out of monads, ultimately encourages us to program at least mostly functionally and keeping imperative code to a minimum. We did this successfully for the Gale-Shapley algorithm. For graphs, the challenge is greater. There are two reasons for that:

- Dynamic updates. We designed our adjacency list representation so it supports constant-time vertex and edge insertions and deletions. If this is really important in the application we are working on, then we're stuck in an inherently imperative world because the whole data structure needs to be implemented using many mutable pointers (in Haskell, *STRef*s). For the algorithms in class, we never used this dynamic update capability apart from arguing that, when building the graph, we can add the vertices and edges one by one at constant cost per insertion. We'll see that we can in fact build a static adjacency list representation in linear time purely functionally. So this problem goes away for us.

- Storing information "with vertices" (and edges). We described our algorithms as storing certain information with vertices and edges (explored or not, tree edge or not, etc.) If this information is stored directly in the graph structure, then again, we need to embed mutable pointers into our data structure, in order to support this. We get around this by associating a unique ID between 1 and n with every vertex and a unique ID between 1 and m with every edge; since the graph is

[1] Research into purely functional algorithms and data structures is a whole research area on its own because functional algorithms and data structures simply are a whole different beast from imperative ones. However, imperative algorithms are easier to design, so talking about the added challenges arising from trying to do things functionally would be a mistake at this point.

static, this is easy. Then whatever information we need to associate with the vertices and edges of a graph can be stored in an array indexed by their IDs. This array must often still be mutable, but the imperative nature of this part of the computation no longer pervades our whole graph representation.

2 Converting Between Graph Representations

To make our graph implementation easy to use, we would like to remove the burden of constructing an adjacency list representation from the user. Thus, normally, we expect the user to provide a very simple graph representation consisting of a list of vertices and a list of edges:

```
data Graph v vl el = Graph {graphVertices :: [Vertex v vl]
                           ,graphEdges    :: [Edge v el]
                           }

data Vertex v vl = Vertex {vertexId    :: v
                          ,vertexLabel :: vl
                          }

data Edge v el = Edge {edgeTail, edgeHead :: v
                      ,edgeLabel          :: el
                      }
```

Since our algorithm needs adjacency lists for fast operation, we need to convert this into an adjacency list representation defined as follows:

```
data AdjList v vl el = AdjList {adjGraph       :: GraphStructure
                               ,adjVertexId    :: V → v
                               ,adjVertexLabel :: V → vl
                               ,adjEdgeLabel   :: E → el
                               }
```

The *GraphStructure* stores the structure of the graph in the form of the adjacency list representation from class. Vertex IDs, vertex labels, and edge labels are associated with vertices and edges in this *GraphStructure* using appropriate functions. Thus, we can change the information associated with vertices and edges just by changing these functions. We do not need to touch the graph structure at all. Now let's define the representation of the graph structure:

```
data GraphStructure = G {gNumVertices :: Int
                        ,gNumEdges    :: Int
                        ,gVertices    :: [V]
                        ,gEdges       :: [E]
                        }
```

So, a graph structure stores the size of the graph and a list of vertices and edges. Every vertex stores its index and its adjacency list, split into in-edges and out-edges to make it easier to manipulate directed graphs.

```
data V = V {vIx       :: Int
           , vInEdges  :: [E]
           , vOutEdges :: [E]
           }
```

Every edge stores its index and its head and tail:

```
data E = E {eIx          :: Int
           , eTail, eHead :: V
           }
```

Now converting a *Graph v vl el* into an *AdjList v vl el* can take anywhere between $O(n+m)$ and $O(nm)$ time depending on what we know about the vertex ID type *v*. Since a vertex is identified by its ID, at the very least these IDs need to be comparable, that is, *v* must belong to the type class *Eq*. If that's all we know, the adjacency list construction, using *makeAdjListEq*, takes $O(nm)$ time. If in addition, we have an ordering relation on *v* (type class *Ord*), we can speed things up using a sorting algorithm and things take $O((n+m)\lg n)$ time, using *makeAdjListOrd*. Finally, if *v* belongs to the *Ix* class, we can use the elements in *v* as array indices and thus construct an adjacency list representation in $O(n+m)$ time, using *makeAdjListIx*.

In fact, we have four versions of each, two for directed graphs, the other two for undirected graphs. The need for different functions for undirected and directed graphs arises because we want to avoid dealing with directed and undirected graphs differently in our graph traversal algorithms. To this end, we represent every undirected graph as its corresponding directed graph where every edge is replaced with two directed edges between its endpoints, one pointing in each direction. It may also be possible that the vertex IDs of the graph aren't themselves in the *Eq*, *Ord* or *Ix* class but there exists a unique mapping from vertex IDs to values of such types. In this case, we would still like to be able to take advantage of, for example, the greater efficiency of adjacency list construction for *Ix* types, so we offer functions that take the mapping from vertex IDs to the appropriate index types as an argument:

```
makeDirAdjListEq      :: Eq v  ⇒ Graph v vl el → AdjList v vl el
makeDirAdjListOrd     :: Ord v ⇒ Graph v vl el → AdjList v vl el
makeDirAdjListIx      :: Ix v  ⇒ Graph v vl el → AdjList v vl el
makeUndirAdjListEq    :: Eq v  ⇒ Graph v vl el → AdjList v vl el
makeUndirAdjListOrd   :: Ord v ⇒ Graph v vl el → AdjList v vl el
makeUndirAdjListIx    :: Ix v  ⇒ Graph v vl el → AdjList v vl el
```

```
makeDirAdjListIndexByEq      :: Eq t  ⇒ (v → t) → Graph v vl el → AdjList v vl el
makeDirAdjListIndexByOrd     :: Ord t ⇒ (v → t) → Graph v vl el → AdjList v vl el
makeDirAdjListIndexByIx      :: Ix t  ⇒ (v → t) → Graph v vl el → AdjList v vl el
makeUndirAdjListIndexByEq    :: Eq t  ⇒ (v → t) → Graph v vl el → AdjList v vl el
makeUndirAdjListIndexByOrd   :: Ord t ⇒ (v → t) → Graph v vl el → AdjList v vl el
makeUndirAdjListIndexByIx    :: Ix t  ⇒ (v → t) → Graph v vl el → AdjList v vl el
```

The first six are of course special case of the latter six:

```
makeDirAdjListEq    = makeDirAdjListIndexByEq    id
makeDirAdjListOrd   = makeDirAdjListIndexByOrd   id
makeDirAdjListIx    = makeDirAdjListIndexByIx    id
makeUndirAdjListEq  = makeUndirAdjListIndexByEq  id
makeUndirAdjListOrd = makeUndirAdjListIndexByOrd id
makeUndirAdjListIx  = makeUndirAdjListIndexByIx  id
```

The undirected versions are easily implemented using their directed counterparts:

```
makeUndirAdjListIndexByEq  f = makeDirAdjListIndexByEq  f ∘ dupEdges
makeUndirAdjListIndexByOrd f = makeDirAdjListIndexByOrd f ∘ dupEdges
makeUndirAdjListIndexByIx  f = makeDirAdjListIndexByIx  f ∘ dupEdges

dupEdges   :: Graph v vl el → Graph v vl el
dupEdges g = g {graphEdges = concatMap (λe@(Edge u v l) → [e, Edge v u l]) (graphEdges g)}
```

makeDirAdjListIndexByE, *makeDirAdjListIndexByO*, and *makeDirAdjListIndexByI* first turn the given graph into an intermediate representation:

```
data Indices = Indices {inIxs    :: [[Int]]
                       , outIxs  :: [[Int]]
                       , tailIxs :: [Int]
                       , headIxs :: [Int]
                       }
```

inIxs contains one list per vertex. This list stores the indices of the in-edges of the vertex. Similarly, each list in *outIxs* stores the indices of the out-edges of the vertex. *tailIxs* and *headIxs* have one entry for each edge in the graph, the indices of the tail and head of this edge. Given this intermediate representation, we can construct an adjacency list representation quite easily:

```
makeAdjList          :: Graph v vl el → Indices → AdjList v vl el
makeAdjList g ixs = AdjList {adjGraph          = gs
                                          , adjVertexId       = vid
                                          , adjVertexLabel = vlab
                                          , adjEdgeLabel    = elab
                                          }
  where n      = length (graphVertices g)
          m      = length (graphEdges g)
          vid   = makeVAttr $ zip [1 ..] (map vertexId       $ graphVertices g)
          vlab = makeVAttr $ zip [1 ..] (map vertexLabel $ graphVertices g)
          elab = makeEAttr $ zip [1 ..] (map edgeLabel    $ graphEdges g)

          gs = G {gNumVertices = n
                      , gNumEdges    = m
                      , gVertices       = vs
                      , gEdges           = es
                      }

          vs  = zipWith3 makeV [1 ..] (inIxs ixs) (outIxs ixs)
          es  = zipWith3 makeE  [1 ..] (tailIxs ixs) (headIxs ixs)
          va = listArray (1, n)  vs
          ea = listArray (1, m) es

          makeV i is os = V i (map (ea!) is) (map (ea!) os)
          makeE i t h    = E i (va ! t) (va ! h)
```

Most of this construction is pretty straightforward: An adjacency list consists of a graph structure and three functions associating vertex IDs, vertex labels, and edge labels with the vertices and edges of the graph. These functions are easily constructed from the vertex and edge lists of *g* using the first block in the **where** clause. We use helper functions *makeVAttr* and *makeEAttr* here, which translate an array into a function by implementing the function using array lookup:

```
makeVAttr      :: [(Int, t)] → V → t
makeVAttr as = vAttr
  where a          = array (1, length as) as
          vAttr v = a ! vIx v

makeEAttr      :: [(Int, t)] → E → t
makeEAttr as = eAttr
  where a          = array (1, length as) as
          eAttr e = a ! eIx e
```

The graph structure is a bit trickier to build. It stores the number of vertices and edges; that's easy.

It also stores the vertex and edge lists of the graph. They are constructed using the third block of the **where** clause, and we also turn them into arrays for fast lookups. Now, every vertex in *vs* is constructed from its index *i*, the list of indices of its in-edges, and the list of indices of its out-edges. Since the vertex needs to store the actual edges, we look them up in the edge array *ea*. An edge is constructed similarly by looking up its tail and head in the vertex array *va*. But this seems to create a chicken-and-egg situation. To construct a vertex, we need to look up its incident edges in the edge array. To construct an edge, we need to look up its endpoints in the vertex array. The answer to the puzzle is laziness, which allows us to store a vertex in the vertex array before having fully computed it, and the same for edges.[2]

All we need to figure out now is how to build an *Indices* structure from the given graph. This is where the three graph construction functions differ because the tricks we can apply to make this construction fast depend on the capabilities of the vertex ID type *v*. Again, we can in fact factor out a lot of common code into a function *makeIndices*. This function needs a dictionary that can be built from an association list and allows us to look up things by vertex ID. For this, we can use the dictionary implementations provided by *Algos.DS.Dict*. The different construction methods for the *Indices* structure then differ in the type of dictionary we provide to *makeIndices*.

```
makeIndices                    :: (v → k) → ([k] → [(k,Int)] → d) → (d → k → [Int]) →
                                  Graph v vl el → Indices
makeIndices f makeDict findDict g = Indices is os ts hs

  where vertexKeys = map (f ∘ vertexId)   (graphVertices g)
        tailKeys   = map (f ∘ edgeTail)   (graphEdges g)
        headKeys   = map (f ∘ edgeHead)   (graphEdges g)

        indexedVs     = makeDict vertexKeys (zip vertexKeys [1..])
        tailIndexedEs = makeDict vertexKeys (zip tailKeys   [1..])
        headIndexedEs = makeDict vertexKeys (zip headKeys   [1..])

        is = map (       findDict headIndexedEs ∘ f ∘ vertexId)  (graphVertices g)
        os = map (       findDict tailIndexedEs ∘ f ∘ vertexId)  (graphVertices g)
        ts = map (head ∘ findDict indexedVs     ∘ f ∘ edgeTail)  (graphEdges g)
        hs = map (head ∘ findDict indexedVs     ∘ f ∘ edgeHead)  (graphEdges g)
```

Now, if all we can do with elements of type *k* is to test them for equality, then we need to use an *EqDict*:

```
makeDirAdjListIndexByEq f g = makeAdjList g (makeIndices f makeEqDict eqDictFind g)
```

Since *eqDictFind* takes time linear in the size of the, the construction of *is* and *os* applies *eqDictFind* to each vertex, and each such *eqDictFind* operation searches a dictionary of size *m*, the cost is $O(nm)$. Conversely, the construction of *ts* and *hs* applies *eqDictFind* to each edge and each such operation searches a dictionary of size *n*, again a cost of $O(nm)$. Thus, *makeDirAdjListIndexByE* takes $O(nm)$ time:

[2] In Haskell-speak, what the array stores for each vertex is a *thunk* that computes the vertex when needed.

If we have an ordering on elements of type k, we can use an *OrdDict* instead of an *EqDict*:

```
makeDirAdjListIndexByOrd f g = makeAdjList g (makeIndices f makeOrdDict ordDictFind g)
```

As before, the cost of *makeAdjList* is linear once we are given the *Indices* structure. *makeIndices* calls *makeOrdDict* three times, once on a set of size n and twice on a set of size m. *makeOrdDict* on an input of size s takes $O(s \lg s)$ time. Thus, all the calls to *makeOrdDict* made by *makeIndices* take $O(n \lg n + m \lg n) = O((n+m) \lg n)$ time. Finally, the construction of *is*, *os*, *ts*, and *hs* in *makeIndices* requires $O(n)$ lookups on dictionaries of size m and $O(m)$ lookups on dictionaries of size n. This takes $O(n \lg m + m \lg n) = O((n+m) \lg n)$ time. Thus, in total, the cost of *makeDirAdjListIndexByOrd* is $O((n+m) \lg n)$.

Finally, if the type k belongs to the type class *Ix*, we can use an *IxDict*. This reduces the construction time of the dictionaries used by *makeIndices* to $O(n+m)$ and allows constant-time lookups. Thus, the cost of *makeDirAdjListIndexByIx* becomes $O(n+m)$:

```
makeDirAdjListIndexByIx f g = makeAdjList g (makeIndices f makeIxDict ixDictFind g)
```

Finally, we'd also like to be able to convert an adjacency list representation back to a graph if necessary. We need to distinguish whether the adjacency list we're converting represents a directed or undirected graph. In a directed graph, we can add every edge in the adjacency list to the edge list of the graph we construct. For an undirected graph, remember that we duplicated every edge so we can treat directed and undirected graphs uniformly in our algorithms. This means that we now need to ensure we report only one of these two copies of each edge. The criterion is easy: We only report edges leading from vertices with lower indices to vertices with higher indices.

```
makeDirGraph :: AdjList v vl el → Graph v vl el
makeDirGraph = makeGraph (const True)

makeUndirGraph :: AdjList v vl el → Graph v vl el
makeUndirGraph = makeGraph (λ(E _ t h) → vIx t < vIx h)

makeGraph                                     :: (E → Bool) → AdjList v vl el → Graph v vl el
makeGraph edgeFilt (AdjList g vid vlab elab) = Graph vs es
  where vs = map (λv             → Vertex (vid v)         (vlab v)) (gVertices g)
        es = map (λe@(E _ t h) → Edge   (vid t) (vid h) (elab e)) (filter edgeFilt $ gEdges g)
```

3 Some Basic Operation

... can now be implemented quite easily. These include degree queries, listing neighbours, and listing incident edges.

```
vIncidentEdges   :: V → [E]
vIncidentEdges v = vInEdges v ++ vOutEdges v
```

```
vInNeighbours, vOutNeighbours, vNeighbours :: V → [V]
vInNeighbours   = map eTail ∘ vInEdges
vOutNeighbours  = map eHead ∘ vOutEdges
vNeighbours v   = vInNeighbours v ++ vOutNeighbours v
```

```
vInDegree, vOutDegree, vDegree :: V → Int
vInDegree   = length ∘ vInEdges
vOutDegree = length ∘ vOutEdges
vDegree v    = vInDegree v + vOutDegree v
```

4 “Flipping” a Graph

An operation that is useful in the context of an algorithms for finding the strongly connected components of a graph and may be useful elsewhere is that of “flipping” a graph: computing a graph with the same vertex set but where every edge points in the opposite direction. Note that the transformation we apply here means that the flipped graph doesn’t in fact have the same vertex set but rather every vertex in the flipped graph is indistringuishable from its corresponding vertex in the original graph in terms of vertex index, vertex ID, and vertex label; since the two graphs have different edge sets and every vertex points to its incident edges, they cannot in fact share the same vertex set. Here’s how we flip a graph in linear time:

```
flipGraph    :: AdjList v vl el → AdjList v vl el
flipGraph g = g {adjGraph = flippedGS}
  where gs           = adjGraph g
        flippedGS = gs {gVertices = map ((!) vs ∘ vIx) (gVertices gs)
                             , gEdges    = map ((!) es ∘ eIx) (gEdges gs)
                             }
        vs           = array (1, gNumVertices gs) $ map makeV (gVertices gs)
        es           = array (1, gNumEdges gs)    $ map makeE  (gEdges gs)
        makeV v   = (vIx v, v {vInEdges    = map (λe → es ! eIx e) (vOutEdges v)
                                    , vOutEdges = map (λe → es ! eIx e) (vInEdges v)
                                    })
        makeE e   = (eIx e, e {eTail   = vs ! (vIx $ eHead e)
                                    , eHead = vs ! (vIx $ eTail e)
                                    })
```

Note that we could have constructed the vertices and edges of *flippedGS* a bit easier, as *elems vs* and *elems es*, respectively. This, however, would have resulted in vertex and edge lists sorted by vertex indices. In the context of Kosaraju’s strongly connected components algorithm, it is imperative that the ordering of the vertices is not disturbed. So we ensure more generally here that all vertices and edges in the flipped graph appear in the same order as their counterparts in the original graph.