Part 11

—

Bipartiteness Testing

CSCI 3110 Code

Summer 2015

Next up we use BFS to test whether a graph is bipartite. If we were just interested in a yes/no answer, the type signature of the function would be

$$isBipartite :: AdjList\ v\ vl\ el \to Bool$$

However, we would like our algorithm to provide proof for the correctness of its answer. In case of positive answer, we want the algorithm to report a correct bipartition. In case of a negative answer, we want the sequence of edges in an odd cycle. The common idiom to report different answers in case of different outcomes of the computation is to use the *Either* **type**:

$$\textbf{data}\ Either\ a\ b = Left\ a \mid Right\ b$$

It is also common to use *Right* to denote success and *Left* to denote failure. So, what we want here is

$$isBipartite :: AdjList\ v\ vl\ el \to Either\ [E]\ ([V],[V])$$

In case of a non-bipartite graph, we want the output to be *Left c*, where c is an odd cycle represented as a sequence of edges. If the graph is bipartite, we want the output to be *Right* (u,w), where (u,w) is the bipartition of the vertices of the graph.

$$\begin{aligned} &isBipartite\ g = maybe\ (Right\ bip)\ (Left \circ oddCycle\ f)\ odde \\ &\quad \textbf{where}\ f \quad\ = bfs\ g \\ &\qquad\quad bip \ = bipartition\ f \\ &\qquad\quad odde = oddEdge\ bip\ g \end{aligned}$$

isBipartite starts by computing a bipartition of the vertices of g that satisfies all the edges of a BFS forest f. To do this, it uses the function *bipartition* below. Then it checks whether g has an "odd edge", an edge which is part of an odd cycle. *oddEdge* returns *Just* such an edge if one exists and *Nothing* otherwise. Thus, if $odde \equiv Nothing$, we return the bipartition wrapped in *Right*. Otherwise, we run $oddCycle\ f$ on the odd edge and wrap the result in *Left*; $oddCycle\ f\ e$ returns the odd cycle induced by e under the assumption that the endpoints of e have the same depth in f, which is true because f is a BFS forest.

```
bipartition   :: Forest V E → ([V], [V])
bipartition f = (concat us, concat ws)
  where (us, ws)          = unzip $ map bip (trees f)
        bip (Tree v ts) = (v : concat ws', concat us')
          where (us', ws') = unzip $ map (λ(Subtree _ t) → bip t) ts
```

bipartition partitions each tree in f into alternating levels, thereby producing a bipartition that respects the edges of the tree. By concatenating the U- and W-sides of all these bipartitions, we obtain a bipartition that respects all the edges of f. To compute the bipartition of a single tree t, we call *bip t*. *bip t* computes a partition (us', ws') that respects the edges in t's subtrees and has all children of t in us'. Thus, by adding the root of t to ws' and then swapping the roles of us' and ws', we obtain a partition that satisfies all edges in t and has the root of t in us'.

Next let's find an odd edge, given a bipartition of f. First, in order to check in constant time whether a given edge is odd, we need a lookup table *bipa* that stores the bipartition each vertex belongs to. The side is either U or W. Given a bipartition (us, ws), we replace every vertex $u \in us$ with $(vIx\ u, U)$ and every vertex $w \in ws$ with $(vIx\ w, W)$ and then use the resulting list of index-side pairs to build the lookup table. Next we just need to scan the list of edges using *find* to locate an edge whose endpoints belong to the same side. *find* returns *Just* such an edge as soon as it finds one or *Nothing* if it reaches the end of the list without finding one:

```
data Side = U | W deriving Eq

oddEdge              :: ([V], [V]) → AdjList v vl el → Maybe E
oddEdge (us, ws) g = find (λ(E _ t h) → bipa ! (vIx t) ≡ bipa ! (vIx h)) (gEdges (adjGraph g))
  where bipa = array (1, gNumVertices (adjGraph g)) $
                 (zip (map vIx us) (repeat U)) ++ (zip (map vIx ws) (repeat W))
```

Exctracting an odd cycle requires us to traverse paths bottom-up, so we start by converting the given forest f into an *UpTree uf*. Then we look for the endpoints of the edge e in uf and extract their ancestor edges using *upPath*. We zip these two paths together; they must have the same length because e is an odd edge, that is, both its endpoints are on the same level in f. This implies that the zipped list *pairs* consists of a prefix of edge pairs where the two edges are distinct, followed by a suffix where the edges in each pair are the same. The edges in the prefix make up the paths from the endpoints of e to their lowest common ancestor. The suffix is the set of edges from this lowest common ancestor up to the root of the tree. Thus, by reversing the path from the tail to the LCA to obtain a path from the LCA to the tail, then adding e, and then adding all edges from the head back up to the LCA (flipping every edge so it points from child to parent, not the other way around), we obtain an odd cycle in g, which we report:

```
oddCycle      :: Forest V E → E → [E]
oddCycle f e = reverse tp' ++ [e] ++ map flipEdge hp'
  where uf          = toUpTree f
    tp              = upPath (eTail e)
    hp              = upPath (eHead e)
    upPath v        = pathEdges $ fromJust (find (λ(w, _) → vIx v ≡ vIx w) uf)
    pathEdges (_,p) = map fst p
    pairs           = zip tp hp
    (tp', hp')      = unzip (takeWhile (λ(e,f) → eIx e ≢ eIx f) pairs)
    flipEdge e      = e {eTail = eHead e, eHead = eTail e}
```