23 Hopcroft-Tarjan, continued

Recall the (corrected) definition of $\phi(u,v)$:

$$\phi(u,v) = \begin{cases} 2 \times \texttt{pre\_rank}(v) & \text{if } (u,v) \text{ is a back-edge} \\ 2 \times \texttt{pre\_rank}(\texttt{highpt\_1}(v)) & \text{if } (u,v) \text{ is a tree edge and } \texttt{highpt\_2}(v) = v \\ 1 + 2 \times \texttt{pre\_rank}(\texttt{highpt\_1}(v)) & \text{if } (u,v) \text{ is a tree edge and } \texttt{highpt\_2}(v) \neq v \end{cases}$$

We have changed some names! Attempt layout has become can fill out...

In the palm tree, the out-edges (u,v) from every vertex u are sorted according to the ϕ-ordering.

Once the dfs has been completed and the palm tree constructed, the primitive object in the algorithm is the *path*, which will be a tree path followed by a single back edge. It is presented as a linked list of edges.

The algorithm is

```
G is now in palm-tree form.
extract_path ( root of dfs tree )
      Since G is biconnected, this is
      the only path extracted which is a simple cycle.
      See below for more about extract_path.

This cycle will always be alone in its block (see below), which
is created at this point.

Call can_fill_out_path with this cycle.  If it returns
1 then the graph is planar, else nonplanar.
```

(23.1) Definition 1. *A* path *is a sequence of edges, always consisting of a tree branch (possibly empty) followed by a back edge.*

2. *The first path developed during the algorithm is a simple cycle beginning and ending at the root. Otherwise the last vertex on the path is a proper ancestor of the first. The edges are directed from parent to child along the tree branch and from vertex to ancestor on the back edges.*

3. *Suppose that (a,b) and (c,d) are the first and last edges of a path, possibly the same. Then a is the* low attachment *and d is the* high attachment *of the path.*

4. *Any vertex on the path which is neither the low nor the high attachment is an* inner vertex.

5. *There is a function* `can_fill_out ( P )` *which visits the inner vertices of P from bottom to top. At each vertex u it calls* `extract_path(u)` *repeatedly to produce paths Q which are recursively processed in their turn. P is the* parent *of each such path Q. That is, the parent of Q is the unique path of which the low attachment of Q is an inner vertex.*

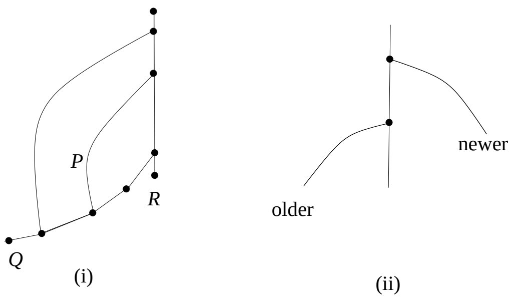

Figure 1: (i) P is tied to its parent Q; its high attachment is inner vertex of R, so R is the base path, and P must go on the same side of R as Q does. (ii) Interlacing test — if the newer has higher attachment than the older then the two paths interlace.

6. *The first path extracted will be a simple cycle containing the root of the DFS tree. It is called the* initial cycle*. It has no parent (or null parent).*

7. *The* base path *of a path P (not the initial cycle) is either the initial cycle if the high attachment of P is the root, or otherwise the unique path of which the high attachment is an inner vertex.*

8. *A path P is* tied *if its base path differs from its parent.*

With reference to Figure 1, the tree branch *leading from the low attachment of Q to its high attachment, an inner vertex of R, is contained in a union of several paths, its parent P, the parent of P (if different from R), and so on. Since the tree branch is connected, all of these paths must be on the same side (left or right) of R as the back-edge ending the path Q.*

The algorithm uses `extract_path` (see below) to decompose G into a set of paths.

(23.2) Lemma *Given that S is the set of paths extracted so far at some point in the algorithm, every vertex which occurs in a path in S, except the root, is an inner vertex in a unique path in S.*

(23.3) The routine `can_fill_out_path` first calls `can_add_path()` to see if the existing layout (of the earlier paths) can be extended to include this path.

If `can_add_path()` reports success, `can_fill_out_path` considers all the inner vertices of the path. If the path is a single back-edge, with no inner vertices, it does nothing more and reports success. Otherwise it processes the inner vertices from lowest (the 'from' vertex of the last edge on the path) to highest, ascending the dfs tree.

For each of these inner vertices u, it repeatedly calls `extract_path(u)`, halting when `null` is returned, and otherwise calling `can_fill_out_path` recursively on this path.

(23.4) The routine `extract_path ( u )` begins at u (low attachment) and ends with a back-edge. Edges added to the path are deleted from the graph, so eventually there will be no edges incident to u.

(23.5) Definition *Two paths P and Q,* where Q is the older path, so it has already been placed relative to other paths, interlace *if the high attachment for P is* above *that for Q, i.e., the high attachment for P has lower preorder rank.*

The crux of the algorithm is that P and Q interlace if and only if they belong to interlacing bridges relative to some cycle.

(23.6) Definition *A* block *is a maximal set L, R of paths with the property that every path on one side is either tied to another on that side or it interlaces a path on the other side (and no two paths on the same side interlace).*

```
  can_layout (path) does the following.
    It calls can_add_path (path) to check that
      the path itself can be placed without
      violating planarity.

    It processes the _inner_ vertices of the path in
      ascending order (ascending the dfs tree).
      The current vertex in this traversal is u.
      The routine extract_path() constructs a path,
      deleting the edges of the path from the graph as
      it proceeds.

        repeatedly, until there are no edges left out of u,
        calls remove_paths_notabove() to remove all
        existing paths whose high attachment is at or below u,
        calls extract_path(u),
        and calls can_add_path with the new path.

        can_add_path can return 0 at any point in which
        case the graph is nonplanar.
```

23.1 Can add path

(23.7) Definition *A (new) path P* interlaces *a block B* low left *if B contains an active path on its left and P interlaces the lowest of these paths. Similarly, one can define when P interlaces B low right.*

```
  int low_interlacing_check ( PATH * path, BLOCK * block )
returns
  (i) LEFT (ii) RIGHT (iii) BOTH or (iv) NEITHER
```

(abbreviated ℓ, r, b, n) according as the path interlaces the block's low active paths (i) left only, (ii) right only, (iii) left and right, or (iv) neither left nor right.

The blocks are stored in a linked list. There is a *low block* which is the first in the linked list; the last in the list is the block containing the initial cycle, with the initial cycle on the left; it remains the only path in that block because it is tied to nothing and interlaces nothing.

The blocks have associated depths with the low block deepest and the block containing the initial cycle at depth 0.

The routine `can_add_path (P)` begins with `can_collect (P)` which

- Finds the base path for P, hence determining if P is tied. If P is tied then the 'top path' is the last path following parent links, beginning at P and not including the base path. The 'top block' is that containing the 'top path'
- If P is not tied then the 'top block' is that below the block for which `low_interlacing_check` produces `NEITHER`. (This exists since the initial cycle never interlaces).
- Then a side is chosen for P. If P is tied it is the side containing the top path in the top block; otherwise the choice is arbitrary.
- Then go through the blocks from low block to top block, at each time invoking `low interlacing check` to see whether P interlaces the lowest path on one or other side of the block.
    - If the check returns 'both' then the path interlaces paths on both sides and the graph is not planar.
    - If the check returns the side chosen and P is tied then the P interlaces a path on the side it is tied to and the graph is not planar.
    - If the check returns the side chosen and P is not tied then the sides of the block are switched.
- Having gone through this process without discovering that the graph is not planar, all the blocks visited are coalesced into one. All blocks up to top block are removed.
- Then `can_collect` returns to the calling function, `can_add_path`. This routine transfers the paths accumulated in `can_collect` into a new low block to which P is added.

Back to `can_fill_out_path`. Having placed P in the low block, it visits all the inner vertices (if any) of P, lowest (with respect to the DFS tree) vertex first. At each vertex u, it repeatedly calls `remove_paths_notabove`(u) which removes from the blocks all paths whose high attachment is at or below u (in the DFS tree), and then `extract_path`(u) which produces a new path from u each time, until it returns null. Recursively, `can_fill_out_path` is applied to each of these paths in turn; if it returns 0 then the graph is not planar. Ultimately, if all of these functions `can_fill_out_path` return 1, then it is reported that the graph is planar.

23.2 Example

Figure 2 shows the decomposition of a graph into paths following the palm tree construction. The vertices are ordered downward according to preorder rank. Paths are indexed from 0 to 5.

Path 0: initial cycle $0, 1, 2, 0$. Block depth 0, which never changes, contains just this cycle. The initial cycle is placed on the left, an arbitrary choice.

Path 1: $1, 2, 0$. Its base path is its parent so it is not tied nor interlaces. There are now two blocks. Block 0: path 0 on left. Block depth 1: path 1 on left (arbitrary).

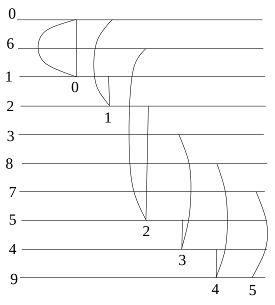

Figure 2: A planar graph

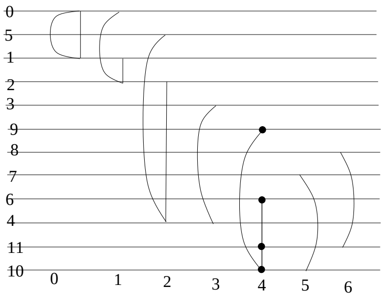

Figure 3: A nonplanar graph

Path 2: $2, 3, 8, 7, 5, 6$. Its base path is the initial cycle and it is tied to Path 1. Block depth 1 now contains $2, 1$ on left.

Path 3: $5, 4, 3$. Its base path is 2 (parent, not tied) and it does not interlace. It is put in its own block: Depth 2: path 3 on left. Depth 1: paths 2,1 on left Depth 0: path 0 on left

Path 4: $4, 9, 8$. It has base path 2 and is tied to path 3. Now the blocks are: Depth 2: left 4,3 Depth 1: left 2,1. Depth 0: left 0.

Path 5: $9, 7$. It does not interlace any path and its base path is 2, so it is tied to 3 and 4. Blocks: depth 2: $5, 4, 3$ on left; depth 1: $2, 1$ on left; depth 0: 0 on left.

All the paths have been successfully 'filled out' and the graph is planar.

23.3 Example

From this data (Figure 3) there are 7 paths labelled 0-6.

Path 0: $0, 5, 1, 0$ in block 0 on left.
Path 1: $1, 2, 0$ not tied, not interlacing, block 1 on left.
Path 2: $2, 3, 4, 9, 8, 7, 6, 4, 5$. Base path 0, tied to path 1. Block 1: paths $2, 1$ on left.
Path 3: base = parent, no interlacing. New block 2, path 3 on left.
Path 4: base = parent, no interlacing. New block 3, path 4 on left.
Path 5: base = 2, tied to 4, no interlacing. Block 3: 5,4.
Path 6: base = 2, tied to 4 in block 3, on left. But path 5 is also on left of block 3, and 6 interlaces 5, so the graph is nonplanar.