Algorithmic Descriptions and Pseudo-Code for Phylogenetic Component Graphs

Ward C. Wheeler
Division of Invertebrate Zoology,
American Museum of Natural History,
Central Park West @ 79th Street,
New York, NY 10024-5192,
USA,
wheeler@amnh.org

November 24, 2016

Abstract

Algorithmic Descriptions and Pseudo-Code for Phylogenetic Component Graphs (PCG) code base.

1 Introduction

This document contains the descriptions and pseudo-code for core Phylogenetic Component Graphs (PCG) algorithms. This is designed to serve both as documentation for the code and a development tool. We have tried to keep the style as consistent as possible, Functional-type statements (*e.g.* map and fold) and structures (*e.g.* lists, tuples) are employed.

2 Labelling of a single graph

One of the fundamental procedures is in the labelling of individual graphs, This is the determination of various aspects (including preliminary, final, and single assignments) of vertices and edges.

The basic PCG graph is a forest of directed acyclic graphs (DAG). These DAGs may be trees in the sense of having vertices with (indegree, outdegree): (0, 1) the unique root, (1,0) leaves, and (1,2) "internal" vertices.; or networks adding (2,1) vertices as "network" vertices. Individual components may have more than one root vertex. All trees or networks are *components* of a forest. A forest is a list of components, $F = [N]$, where $N = (V, E)$ (conventionally V is a set of vertices and E a set of edges). One or more vertex in each component is a root. Vertices with indegree and outdegree 1 may be maintained for computational convenience, but would normally be contracted.

Following Moret et al. (2004), the above would be described as a "model" phylogenetic network. This has two additional constraints. First, that all edges must begin or end in a tree vertex (no network vertex to network vertex edges). Second, the parents of a network vertex must, at least potentially, be contemporaneous. [Insert path conditions here]

A less restrictive form of network ("reconstructable") is defined by Moret et al. (2004) with vertices (indegree, outdegree): $(0, \geq 1)$ the unique root, $(1, 0)$ leaves, and $(1, \geq 1)$ "internal" vertices, and networks adding $(\geq 2, 1)$ vertices as "network" vertices. For purposes of searching, outdegree of internal nodes is restricted to at most 2. Since edges may have zero length, networks may be output with vertices of outdegree > 2.

3 Optimization of a single graph

Optimization is a function that takes a graph (G), here a forest F, and data (D) as inputs and returns a numerical value. This value can be referred to as its score (S)

$$\begin{aligned} f &: \{\mathcal{F}, D\} \to \mathbb{R} \\ g &: \{\mathbb{R}, \mathbb{R}\} \to \mathbb{R} \end{aligned}$$

4 Forest Optimization/Labelling

The optimization is simply the optimization of each constituent component and their optimization values combined through f (summed for parisomy, if likelihood or such then add logs). Both local (vertex specific) and subtree costs (local cost of a vertex plus the subtree costs of its descendants) are stored. The overall component cost is stored at the traversal root.

$$\begin{aligned} \mathcal{F} &= [C] \\ \mathcal{F}_{labelled} &= \text{map } labelNetwork\ [C] \\ [S_{\mathcal{F}}] &= \text{map } f\ \mathcal{F}_{labelled} \\ \mathcal{F}_{optValue} &= \text{fold } g\ [S_{\mathcal{F}}] \end{aligned}$$

Define a 4-tuple $L = (a, b, c, d)$ with a a bitvector of the length of leaves, b a tree representation, c a character assignment (of appropriate character type), and d a real (likely double). Each vertex will have a list of these tuples, $[L]$, one element for each network node resolution, created during a postorder traversal of a component.

Define a function h that takes as input two labellings (state sets of a character) and character metadata, returning a labelling and a real denoting the cost of creating the returned labelling.

$$h : (l, l) \to (l, \mathbb{R})$$

Define a tree representation (e.g. Newick) and a function to "join" the tree representations of two descendant vertices into the representation of the subtree for that vertex. For a newick representation this would be the string operation joining representations A and B into (A, B).

A full optimization of a forest results in a fully "decorated" graph (all vertex and edge information present). This normally proceeds via a map of the optimization operations over the components of a forest. The forest components are optimized in turn by a series of traversals over the vertex and edge sets.

Algorithm 1: forestFullOptimization

```
Data: Input forest $F$ (list of DAGs including components, networks,
      and trees) and data $D$
Result: Optimality value of Forest and and list of optimality value
        and fully labelled (vertex, edge features etc) forest
        components
▷ map optimization over list of components
$optimizedComponentList$ = map (componentFullOptimization $D$) $F$
▷ extract and combine optimality values of components
$optimalityValue$ = fold $g$ (map first $optimizedComponentList$)
return ($optimalityValue$, $optimizedComponentList$)
```

5 Forest Rearrangement

There are two operations in forest rearrangements. The first is the operation which splits network components into multiple components and the second joins separate components into network components.

Algorithm 2: componentFullOptimization

```
Data: Input component $C$ (DAGs multi-rooted component, network,
      or tree) and data $D$
Result: Decorated, fully labelled (vertex, edge features etc)
        component with optimality value
▷ Post-order traversal, beginning with root as fold right
$preliminaryComponent$ = fold (makePreliminaryVertex $D$) $C$
▷ Pre-order traversal beginning at root
$finalComponent$ = fold (makeFinalVertex $D$)
$preliminaryComponent$
return (optimalityValue, optimizedComponent)
```

Algorithm 3: makePreliminaryVertex

```
Data: Input vertex $V$, descendent lists of left $L$ and right $R$
      descendants of $V$ (if not a leaf), and data $D$
Result: List of label 4-tuple $[(a, b, c, d)]$ as defined above for $V$
▷ If leaf take observed states and subtree is just name,
  bitvector has ‘1’ for that leaf and ‘0’ for all others
if $V$ is leaf then
    $labelList$ = $\big[\big(V_{bv}, V_{subtreeRep}, \text{observed state}, 0\big)\big]$
end
▷ Otherwise do all combinations of two lists
▷ Check if previous in list $L$ and $R$ are same as
  current--if so then just return previous label
else
    $labelList$ = map2 joinTwoLabelings $L$ $R$
end
return labelList
```

Algorithm 4: joinTwoLabelings

```
Data: Input two $L$ and $R$ 4-tuple labelings
Result: 4-tuple labelling, could be null ‘()’
▷ If bitvectors have non-empty intersection then
  incompatible resolution and return null
if $L_{bv}$ AND $R_{bv}$ <> 0 then
    return ()
end
▷ Otherwise combine elements
else
    return $\left(L_{bv} \text{ OR } R_{bv}, \text{join } L_{subtreeRep} R_{subtreeRep}, h \ (L_{assignment} R_{assignment})\right)$
end
```

5.1 Break

Each tree edge is deleted in turn. There are two possible outcomes. The resulting two components may or may not be connected (have vertices in common). The neighborhood can be generated and evaluated for overall minimum or breaks evaluated seriatim. In either case, the new forest elements are diagnosed and compared to the current best. If better cost, then kept, else discarded.

Algorithm 5: forestBreak

```
Data: Input forest $F$ (list of DAGs including networks and trees)
      and data $D$
Result: Forest break neighborhood as list
▷ Break each forest component at each tree edge.  Head of
  deleted edge becomes new root, tail becomes (1,1)
  vertex
$[F']$ ←all combinations of tree edge deletions
return $[F']$
```

5.2 Join

Pairs of components are joined by creating a new edge between each pair of edges in the input networks, subject to the time consistency constraint of network edges. This basically proceeds as SPR/TBR joins with added complexity of network edge issues (ie. time consistency.

```
Algorithm 6: forestJoin
  Data: Input forest $F$ (list of DAGs including networks and trees)
        and data $D$
  Result: Forest join neighborhood as list
  ▷ For each pair of forest components, add edge between
    each pair of edges (TBR), or edges and rot vertex (SPR)
  $[F']$ ←all combinations of tree edge joins
  return $[F']$
```

6 Network Rearrangement

As with forest rearrangement, network edges can be added or subtracted, and underlying (displayed) trees rearranged vis NNI, SPR, TBR etc.

6.1 Network edge removal

The optimization of networks identifies unused edges, so these can be removed during optimization. Standardly, networks with unused edges are attached an infinite cost. Another option would be to output a new network with the offending edge deleted. Remaining network edges can be removed in turn.

6.2 Network edge addition

Subject to edge time constraints and that each edge must have at least one tree vertex, new edges are added between each pair of edges (except for those where both are network edges).

```
Algorithm 7: networkEdgeAdd
  Data: Input network $N$
  Result: Network neighborhood as list
  ▷ For each pair of edges in input network, add edge
    between each pair of edges, subject to time and edge
    tree vertex constraints
  $[N']$ ←all combinations of available edge joins
  return $[N']$
```

6.3 Display tree rearangement

NNI, SPR, TBR etc.

7 *Ab initio* build

1. Build trees and create EUN
2. fold over list of leaves = wagner build
3. Build with forest and network rearrangements based on sequential addition of leaves

8 Forest Refinement

Various combinations of forest, network, tree rearrangements.

9 Naive Tree Optimization

10 Tree ReRoot Optimization

11 Naive Soft-Wired Network Optimization

12 Less-Naive Network Optimization

Algorithm 8: Post-Order Soft-Wired Network Traversal and Optimization for arbitrary in-degree and out-degree vertices

```
Data: Input network $N$ and data $D$
Result: The minimum cost and tree displayed by network for each
        input character in $D$
For all characters $\in Dcost_{character} \leftarrow 0$;
while $nodesToDo = True$ do
    if $node_{outdegree} = 0$ then
        $node_{prelim}$ =observed
    end
    else if $node_{outdegree} = 1$ then
        $node_{prelim} = node^{desc}_{prelim}$
    end
    else if $node_{outdegree} = 2$ then
        $node_{prelim}$ = standard if not done and not desc of network edge
    end
    else
        $node_{prelim}$ = ladderize
    end
    if $node_{in-degree} = 0$ then
        $nodesToDo = False$
    end
    else if $node_{in-degree} = 1$ then
        recurse to parent
    end
    else
        split tree then recurse each parent
    end
end
return $[(tree, [cost])]$ can choose best displayed tree for each
character, later do reroot thing (3d-Varon and Wheeler (or Malign
too) for dynamic characters
```

Algorithm 9: FullComponentOptimization

```
Data: Input component $C$ and data $D$
Result: Pair of component optimality value and decorated (edge
        labels, etc) component
▷ Get optimality value of component and postorder
  labelling of component
$(optimalityValue, postOrderAssignment)$ = PostOrderComponent $C$
$D$
▷ Get preorder ‘final’ labelling of component (vertex and
  edges costs assignments etc)
$finalLabelling$ = PreOrderOptimization $postOrderAssignment$ $D$
return (optimalityValue, finalLabelling)
```

Algorithm 10: PostOrderOptimization

```
Data: Input data $D$
Result: Blah
Comment return (Optimization value, labelled component)
```

Algorithm 11: PreOrderOptimization

```
Data: Input data $D$
Result: Blah
Comment return labelled component
```

References

Moret, B. M., Nakhleh, L., Warnow, T., Linder, C. R., Tholse, A., Padolina, A., Sun, J., and Timme, R. 2004. Phylogenetic networks: Modeling, reconstructibility, and accuracy. *IEEE Transactions on Computational Biology and Bioinformatics* 1:13–23.