Improvements to propinquity

October 19, 2016

1 Handling *incertae sedis* taxa

In order to handle *incertae sedis* taxa we must modify three or four stages of the propinquity pipeline. First, we must modify the decomposition into subproblems to place *incertae sedis* taxa in the correct subproblem and handle conflict between subproblems in the placement of such taxa. Second, we must extend the subproblem solver to correctly solve subproblems that contain *incertae sedis* taxa. Third, we *might* need to modify how unpruning works. Lastly, we must modify the the naming of nodes to take into account the fact that OTT taxa might have had *incertae sedis* taxa placed inside them, so that OTT taxa cannot be strictly identified with their set of descendants anymore.

1.1 Decomposing into subproblems

Decomposition involves subdividing the tree into subproblems that can (usually) be solved independently. When *incertae sedis* taxa are involved, we must place *incertae sedis* taxa into subproblems in such a way that the subproblem solver can perform the placement inside the subproblem. However, we must also handle conflicts of *incertae sedis* taxon placement.

I think we can solve this in a logically coherent way by recognizing that any taxonomy edges leading to conflicting placement on an *incertae sedis* taxon must end up in the same subproblem. Thus, if one input phylogeny places the *incertae sedis* taxon X in $((X)B)A$ and another input phylogeny places *incertae sedis* taxon X in $((X)C)A$ then we must mark the edges B and C as contested edges, even if these edges would *not* be contested where taxon X to be removed. This results in a new way to contest edges that involves the interaction of two input trees, and not just the interaction of each input tree with the taxonomy.

The result of such a decomposition procedure would be that the subproblem solve will handle any conflicting placement information.

1.1.1 Possible approach

One approach would be to modify the taxonomy to place *incertae sedis* taxa when scanning input phylogenies. A series of input phylogenies containing taxon X might not disagree, but might each place the taxon X successively more tipward. However, when reading an input phylogeny that conflicts with the placement of X resulting from previously processed input phylogenies, we would end up marking the branches connecting the two attachment points as conflicting with the most recently processed input phylogeny. These branches would then not be used to separate subproblems from each other, and would thus end up in the same subproblem. *This is not proved - expand and check.*

If this does work, then it is hopeful that this would decrease the number of subproblems too much. It seems like it might be OK, since we already contest edges result from alternative placement of non-*incertae sedis* taxa.

1.2 Subproblem solution

Subproblem solution does not need to be modified very much, since the BUILD algorithm already natively supports building trees from partial splits. Thus, we need to modify the subproblem solver to create the partial splits from the incertae sedis information.

1.2.1 Reading incertae sedis information

Currently, we read the *incertae sedis* information as a list of OTT ids for *incertae sedis* taxa. This does not require adding further annotations to the node names. Only taxonomy nodes can be *incertae sedis* at the moment, and only the taxonomy tree for the subproblem contains OTT ids. Therefore we handle *incertae sedis* information by constructing modified split sets for the lowest-ranked tree when the list of *incertae sedis* nodes is not empty.

1.2.2 Exclude sets modified by *incertae sedis* marks

Incertae sedis information changes only the exclude set of splits, and only reduces this set. The exclude set for a node is is the union of the exclude set of the parent node and the include sets of non-incertae sedis siblings. If we use the terminology that the include and exclude sets for a node n are $\mathcal{I}(n)$ and $\mathcal{E}(n)$, then we have

$$\mathcal{E}(n) = \mathcal{E}(parent(n)) \cup \left[\mathcal{I}(m) \middle| m \in siblings(n), m \text{ not } incertae\ sedis\right].$$

This leads to the algorithm to compute the exclude set for all nodes in a tree.

1. Set the exclude set of the root node to be empty
2. For each *node* (except the root) in preorder
   - combine the *exclude* set of the parent node with the *include* set of non-*incertae-sedis* siblings.
   - store this set in a hash, with key *node*

This is currently implemented in *otc-solve-subproblem*. We do not use bitmaps, but sets to store the sets right now.

1.3 Naming

Work on this next, but just for the output of *otc-solve-subproblem*.

1.4 Unpruning

Currently the unpruner might require that the OTT ids are named in the grafted solution before unpruning starts.

2 Unified handling of incompatible splits and broken taxa

Currently when an input tree split conflicts with a previously incorporated split during subproblem solution we discard it altogether. However, when handling a broken taxa during unpruning we do not discard the broken taxon altogether, but instead attach its children at the MRCA of included children. This unpruning can be seen as the solution of a special kind of subproblem that contains the grafted solution (which is pruned) and the full taxonomy (which is not). If the two procedures are not consistent, then pruning will lead to different results. Specifically, lack of pruning would lead not just to slower computation, but to a tree which does not attach the children of broken taxa at the MRCA. Thus, we desire to achieve consistency between handling of broken taxa during pruning and incompatible splits during subproblem solution.

2.0.1 Hypothetical/Platonic: Make the subproblem solver perform unpruning?

Would it be theoretically possible to make the subproblem solver able to perform unpruning by simply feeding the special subproblem

```
grafted solution (pruned)
cleaned ott (unpruned)
```

to the subproblem solver? If so, that would simplify a lot of things conceptually. However, this seems computationally challenging.