Simplifying Failure to Diverge Events by Disallowing Some Host Switches

Kevin Black

August 12, 2011

1 Introduction

The addition of a failure to diverge (FTD) event to the cophylogeny reconstruction problem permits parasites to reside on multiple hosts, broadening the range of problems that can be solved by software such as Jane. Unfortunately, when computing a reconstruction it may be difficult to guarantee that a multihost parasite will infect all necessary hosts. To facilitate polynomial time reconstruction we can impose a restriction on host switches.

2 Restrictions

Let e_P be a parasite edge with children e_{P_1} and e_{P_2} and let $e_{P_{tip}}$ be any parasite tip descending from e_{P_1} (note that an edge is considered a descendant of itself). Then a host switch where e_{P_1} is the edge which switches hosts is not allowed if $e_{P_{tip}}$ failed to diverge at any ancestor of e_P. That is, parasites and their ancestors are disallowed from switching hosts if they have previously undergone a failure to diverge event. Such a parasite or ancestor *is* allowed to take part in a host switch, so long as it is not the lineage switching hosts. For the purposes of this writeup, when I say that a host switch is now allowed after an FTD event, I mean specifically the host switches meeting these restrictions.

In the context of a dynamic programming algorithm such as Jane's that evaluates from the tip associations back to the root of the host tree, a parasite node has no knowledge of where FTD events occurred among its ancestors. In contrast, because FTD events are the only allowed method of obtaining multihost parasites, the locations of FTD events on the host tree for each parasite are known ahead of time. Moreover, to reach the correct number of hosts, a parasite must experience each FTD event. To ensure this, I will assume for this writeup that the edge occurring before the parasite root is always placed before the first FTD event.

3 Proposed Solution

An appropriate solution must be incorporable in Jane and allow exactly those host switches that are legal according to the restriction presented above. The following is a tractable algorithm that satisfies these requirements:

Algorithm 1. Suppose at time t we wish to examine how to best place the parasite e_P onto the alive, non-root host e_H. Let e_{P_1} be the child of e_P's considering switching hosts (if e_P has no children, a host switch cannot occur). Then we do not allow a host switch if and only if there exists a descendant $e_{P_{tip}}$ of e_{P_1} and a host node $v_{H'}$ occurring at a time $t' \leq t$ such that $e_{P_{tip}}$ is a parasite tip needing to undergo FTD at $v_{H'}$.

Lemma 1 *Algorithm 1 prevents host switches of a parasite's ancestor from occurring after FTD events involving that same parasite while allowing all other host switches. That is, it obeys the given restrictions.*

I use the same variables from Algorithm 1. Suppose first that a host switch should not be permitted. Then there is some parasite tip $e_{P_{tip}}$ descending from e_{P_1} that has already failed to diverge at an ancestor $e_{P'}$ of e_P. The FTD event at $e_{P'}$ occurs at or before time t and so the host switch would not be allowed by Algorithm 1.

Suppose instead that the algorithm disallows a host switch. Then there exists a descendant $e_{P_{tip}}$ of e_{P_1} and a host node $v_{H'}$ occurring at a time $t' \leq t$ such that $e_{P_{tip}}$ fails to diverge along the parasite edge $e_{P'}$ at $v_{H'}$. If $e_{P'}$ is an ancestor of e_P then the host switch should be disallowed and so the algorithm chose correctly. If instead $e_{P'}$ is not an ancestor of e_P then they must have previously undergone an FTD of $e_{P_{tip}}$ at a common ancestor (as both have $e_{P_{tip}}$ as a descendant), again validating the algorithm. So in either case a host switch should not have been allowed.

Hence Algorithm 1 correctly identifies the permissibility of all host switches.

4 Going Further

With Algorithm 1 implemented, host switches will not occur after failure to diverge events. Recalling the assumption that the parasite root edge must begin before the first FTD event, we present the following lemma:

Lemma 2 *Assume that there exists a finite-cost reconciliation of the host and parasite trees. Let $e_{P_{tip}}$ be a parasite that must undergo an FTD event at the host node v_H and suppose Algorithm 1 is used to check host switches. Then, in the final mapping of the parasite tree onto the host tree, $e_{P_{tip}}$ will undergo FTD at v_H as necessary. In other words, all FTD events occur (none are skipped).*

Suppose, by way of contradiction, that some FTD event is skipped and consider the first such skipped event. Let v_H be the host at time t and $e_{P_{tip}}$ be the parasite that should have failed to diverge. Note that FTD events, when available, are forced and so no ancestor of $e_{P_{tip}}$ appears on v_H.

Suppose first that there is an earlier host $v_{H'}$ at time $t' < t$ at which $e_{P_{tip}}$ fails to diverge. Then we know it must have been successful (as the first skipped FTD event occurs later, at time t). Moreover, by Lemma 1, no ancestors of $e_{P_{tip}}$ may have switched hosts after time t'. Since no ancestors of $e_{P_{tip}}$ appear on v_H, the ancestry of $e_{P_{tip}}$ must have undergone a loss event somewhere between t' and t.

Let $v_{H''}$ be the location of that loss with children $e_{H''_1}$ and $e_{H''_2}$ and suppose, without loss of generality, that $e_{H''_1}$ is the ancestor of v_H, giving that $e_{H''_2}$ is the path taken by the loss event. Recall that a loss event can only occur when an FTD event is not required. Thus, since $e_{P_{tip}}$ infects a host descending from $e_{H''_1}$, it does not infect a host descending from $e_{H''_2}$. As the ancestor of $e_{P_{tip}}$ is prohibited from switching hosts off of the subtree rooted at $e_{H''_2}$, this parasite tip has nowhere to go and so the solution will have infinite cost, a contradiction.

Suppose instead that v_H is the first host at which $e_{P_{tip}}$ must fail to diverge. Then all hosts infected by $e_{P_{tip}}$ appear in v_H's subtree. But since the FTD event was skipped and no host switches are allowed afterwards by Algorithm 1, the trees cannot be reconciled (as $e_{P_{tip}}$ has no host on which to appear), a contradiction. In both cases a contradiction is reached and so no FTD events are skipped.

Theorem 1 *Let p be a parasite that infects multiple hosts and suppose Algorithm 1 is used to check host switches. Then, in the final mapping of the parasite tree onto the host tree, p will be mapped to each of its hosts.*

Suppose, by way of contradiction, that the final mapping leaves some host e_H unoccupied by its parasite p. Recall the assumption that the edge before parasite tip starts before the first FTD event. Then by Lemma 2 and the fact that each failure to diverge event results in one additional copy of the parasite, there must exist some other host $e_{H'}$ with multiple copies of p. This implies the existence of a host switch that brings an ancestor of p onto a host edge already containing an ancestor of p. But then there would have been two ancestors of p alive at the same time, indicating that p underwent an earlier FTD event. This contradicts Lemma 1. Hence in the final solution each multihost parasite is mapped to all of its necessary hosts, as desired.

5 Running Time

We have seen that Algorithm 1 is sufficient to guarantee that multihost parasites will infect each of their hosts. Next we must ask about efficiency. Let p represent the number of multihost parasites. A naïve implementation would, when checking host switches, first loop over all previous host nodes. It would then check each element in that host's list of parasites needing to FTD there to see if it is a descendant of e_P. This requires $O(np)$ operations to compute a cell in the DP table.

However, we can do better. For each edge e_P, store the earliest time of an FTD event undergone by a descendant of e_P (or, if no descendants undergo FTD, store the largest time possible). Assuming a balanced parasite tree, this array requires $O(p \log n)$ to fill.

To check if a host switch is allowed at a time t for an edge e_P we check if the value stored at e_{P_1} or at e_{P_2} is less than t, requiring only one operation. So the algorithm's total running time is not slowed.