Construction, enumeration, and optimization of perfect phylogenies on multi-state data.

Michael Coulombe
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
Email: mcoulomb@mit.edu

Kristian Stevens and Dan Gusfield
Department of Computer Science
University of California, Davis
Davis, CA 95616
Email: {kastevens, gusfield }@ucdavis.edu

Abstract

Perfect phylogenies are central to both evolutionary biology and population genetics. We implemented and evaluated algorithms for constructing, counting, and enumerating perfect phylogenies on data with an arbitrary number of states. Ours is the first program to implement the efficient algorithm of Agarwala and Fernández-Baca (1994) with the speedups and enumeration extensions by Kannan and Warnow (1995). It is written in the C++ language and uses specialized algorithms and datastructures for faster and more compact execution.

We have included new extensions to the previously described algorithms. Our software can efficiently construct a phylogeny, determine it's uniqueness, or determine that no phylogeny exists. It can handle input data with missing values and find a largest subset of compatible characters. It can count and enumerate the potentially exponential number of trees that may explain an input dataset. Using dynamic programming, it can find a smallest tree or a tree with maximum edge support. While many of these problems have been shown to be NP hard, our implementations are demonstrably practical for many datasets.

1. Introduction

In population genetics, the perfect phylogeny is closely related to the *coalescent* [7]. A perfect phylogeny on binary characters results under the commonly used *infinite-sites* model, where characters are binary and only change state once in the phylogeny. Perfect phylogenies where characters have more than two states occur under the *infinite-alleles* model. The non-root states may arise only once by mutation in the phylogeny. In population genetics perfect phylogenies model ancestry in situations where recombination can be ignored. Such instances arise frequently in analyses of current whole genome data. They have been used in case-control association mapping to find disease loci [3]. They are used extensively to obtain the haplotypes of diploid individuals from genotype information [5] [7].

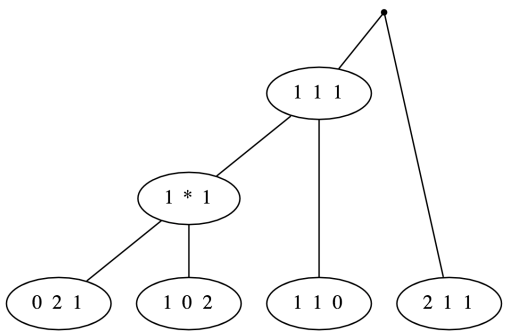
Given an input dataset, the perfect phylogeny problem is to build a tree consistent with the widely used perfect phylogeny model [1], [4], [7]–[9], or determine that no such tree exists. Our software (PerfectPhy) implements the generalized algorithm of Agarwala and Fernández-Baca, who were first to show that the problem is efficiently solvable when the number of states per character is a fixed parameter [1] and incorporates the speedups and the enumeration algorithm described by Kannan and Warnow [8].

Our implementation achieves the running time claimed possible in [8]. We also developed and implemented extensions to the algorithms given in [8]. PerfectPhy handles situations where the input contains missing entries that should be completed from the set of states observed for each character. In situations where a perfect phylogeny exists for only a subset of the characters, it will find a maximum subset. Many of these extensions have been shown to be hard [8], [9], yet we demonstrate that a solution is practical over a large range of the input parameters.

Using extensions to the algorithm and data struc-

978-1-4673-9963-9/15/$31.00

a)



b)

```
(('0 2 1','1 0 2')'1 * 1','1 1 0')'1 1 1','2 1 1')
```

Figure 1. (a) A perfect phylogeny on three characters produced by PerfectPhy. In this example, the input data consisted of the four taxa at the leaves of the tree with '211' as the outgroup. The two ancestral taxa at the internal nodes were inferred by the algorithm. In a perfect phylogeny, the taxa labeled by a the same character state must form a connected subtree. When multiple labelings of imputed ancestral taxon are possible, PerfectPhy uses a wildcard character to indicate when a taxon can be labeled by an adjacent character state. The graphical representation uses the GraphViz *dot* output format. (b) The equivalent output in Newick format.

tures algorithm described in [8], PerfectPhy can count and enumerate the potentially exponential number of trees that are compatible with an input dataset. It also implements novel optimization extensions. It can find a smallest tree, or it can find a tree with maximum support, where support is defined for each edge as the fraction of trees in which it occurs.

2. Background and Implementation

Formally, the input to our problem consists of a set S of n taxa for which we have collected information about observed traits in the form of m characters. Each character can take on up to k discrete states. The most common input data are molecular sequences, where the states are either nucleotides or amino acids. The resulting perfect phylogeny, if it exists, is a rooted tree with the input taxa labeling the leaves and any imputed ancestral taxa labeling the internal nodes (see Fig. 1). Under the perfect phylogeny model, all character/state pairs will label subtrees, a property known as *convexity*. See [7], [8] for alternative definitions.

Construction Algorithm

The algorithms of [1] and [8] are based on the following observation about the edges of a perfect phylogeny. For any perfect phylogeny T, every edge of T defines a bi-partition of S into subsets G and $S - G$. For every character labeling T, at most one state labels a taxon in both G and $S - G$. This follows from convexity. We further confine ourselves to the solution space of parsimonious trees on which each edge corresponds to some change of a character state. So for every edge of T, there is a character where no state labels a taxon in both G and $S - G$.

A key idea [1] is that all partitions consistent with these observations can be enumerated efficiently for a bounded number of states per character. Given a k-state character c over S, there are 2^{k-1} bi-partitions of S into the sets G and $S - G$ such that no state of c labels a taxon in both sets. For each partition, we can also check in linear time if at most one state is shared between G and $S - G$ for all other characters. If this is the case, then there is potentially an edge in some perfect phylogeny that corresponds to the bi-partition. The subsets G and $S - G$, that correspond to potential edges, are called *proper clusters*. For a proper cluster G, the set of character states shared between G and $S - G$ is the *splitting vector* of G, $sv(G)$. Observe that $sv(G)$ is the same as $sv(S - G)$. It defines the set of forced state labels on the ends of the potential edge.

Primarily, we are interested in determining when a rooted perfect phylogeny can be constructed for a proper cluster. A proper cluster G is said to have a *subphylogeny* if there exists some perfect phylogeny for G with a root labeled by $sv(G)$. Figure 2 gives the outline of our implementation of the decision algorithm of [8]. The recursive subroutine SUBPHYLOGENY(G) attempts to build a rooted tree for a proper cluster G. The base case is when G is only a single taxon. Otherwise, to determine if a rooted subphylogeny for G exists, we sufficiently examine possible partitions of G into proper clusters with subphylogenies that can be attached to a common root. First an outgroup t is specified, this is a leaf taxon adjacent to the root which can be arbitrarily chosen from S (e.g. `'211'` in Figure 1). Then the subroutine PERFECTPHYLOGENY(S) calls SUBPHYLOGENY(S-t) at the top level of the re-

```
PERFECTPHYLOGENY(S) {Return a phylogeny for S or fail}
1 Choose a taxon t to be an outgroup
2 if T ← SUBPHYLOGENY(S-{t}) returns failure
4  return failure
5 else
6  return the tree created by attaching t to the root of T

SUBPHYLOGENY(G) {Return a rooted phylogeny for G or fail}
1 initialize root r labeled with sv(G)
2 if G is a single taxon t
3  return the taxon t attached to r
4 else
5  foreach subset H_1 of G where
     T_{H_1} ← SUBPHYLOGENY(H_1) exists and can be attached to r
6   if H_2 ← G − H_1 is a proper cluster
7    if T_{H_2} ← SUBPHYLOGENY(H_2) exists and can be attached to r
8     return the tree created by attaching T_{H_1} and T_{H_2} to r
9   elsif G can be partitioned into l > 2 proper clusters H_1, ..., H_l
      with subphylogenies T_{H_1}, ..., T_{H_l} that can be attached to r
10    return the tree created by attaching T_{H_1}, ..., T_{H_l} to r
11  return failure if no H_1 worked
```

Figure 2. Pseudocode for a memoized recursive formulation of the efficient construction algorithm [8]. The subroutine PERFECTPHYLOGENY() is called at the top level. The recursive subroutine SUBPHYLOGENY() builds a rooted tree for a subset of the taxa from subphylogenies of smaller subsets.

cursion. We implemented the algorithm using dynamic programming.

The speedups in [8] to the algorithm in [1] target the inner loop of the recursive subroutine SUBPHYLOGENY() (lines 5-10). Using specialized data structures and graph algorithms, once a subphylogeny for H_1 is specified and attached to the root, $G-H_1$ can be partitioned into one or more subphylogenies in $O(n)$ time.

Proper cluster indexing. A central data structure in [8] is a proper cluster index, a lexicographical *trie*, that allows for lookup with a time complexity that is linear in the number of elements. Implementing the index this way, we can quickly look up if $G - H_1$ has a subphylogeny that can be attached to the root. The index gives an amortized complexity of $O(n)$ for determining if a given partition of $G-H_1$ corresponds to a set of subphylogenies that can be attached to the root. To support these time bounds, each node of the trie must contain a random access data structure pointing to its $O(n)$ children. The trie takes up the largest portion of the memory required by the program. With $O(2^k m)$ proper clusters, the size of the trie is $O(2^k mn^2)$. Figure 3(a) illustrates the trie and highlights, in grey, space inefficiencies of the data structure:

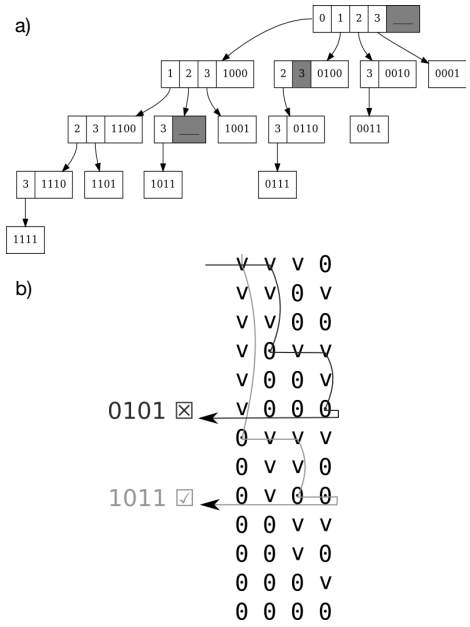


Figure 3. a) The trie datastructure was recommended in [8] to index proper clusters. This example indexes a set of 13 proper clusters b) The corresponding pointer table datastructure is more space efficient index for proper clusters. Here $0 \rightarrow \vee$ and $1 \rightarrow \odot$

nodes which do not correspond to a proper clusters and unused edge pointers.

We developed a novel *pointer table* index which uses less space than the trie. The space used is $O(2^k mn)$, which is the space bound claimed in [8] for the trie, but not verified. Our improved datastructure is a pointer table over the lexicographically-sorted table T of proper clusters. Given T, let Q be a table such that:

$$Q[p][i] = \text{the smallest } p' \geq p \text{ such that } T[p'][i] = 1.$$

A scan of T in descending order can construct Q in constant time per cell, and $|Q| = |T|$, thus preprocessing of Q takes $O(2^k mn)$ time and space. All needed trie operations are supported by algorithms over Q with the same $O(n)$ time complexity.

We use the following function to lookup the index p of a known proper cluster G using Q or return the result that G is not a proper cluster:

```
function LOOKUP(Q, G)
    p ← 0
    for all taxa t ∈ G in order do
        p ← Q[p][t]
```

end for
if $|G| = |G_p|$ and $\forall t \in G\ (p = Q[p][t])$ **then return** p **else return** $NULL$

The lookup function takes time $O(n)$ when G is represented as a bitset, but looking up a disjoint set D of known proper clusters can be done in $O(|G|)$ per $G \in D$ thus $O(n)$ total time is needed when the set is compactly represented. When it is not known whether G is a proper cluster, additional checks can be made within $O(n)$ time to determine it. Given the proper cluster G_p indexed by p, a size check and subset check can determine whether $G = G_p$ in $O(n)$ time. Searching for a disjoint set of proper clusters can be done in $O(n)$ total time given that $|G|$ is precomputed for all G and the set is compactly represented.

Figure 3(b) shows a successful (grey) and failed (black) index query. For each taxon in the proper cluster, the path through the table follows the pointer in the current row and corresponding column, and once every taxon is processed the row can be checked for equality.

3. Enumeration and optimization

We implemented the algorithm introduced in [8] for enumerating all *minimal* perfect phylogenies. A labeled perfect phylogeny is *minimal* if every edge corresponds to a state change of one or more characters. This characterization is appropriately named because the labeled tree will no longer be a perfect phylogeny for the data if any of the edges are contracted. However, details in [8] were insufficient to achieve the claimed time and space complexities. Ultimately the implementation required additional algorithmic detail and novel non-trivial data structures described here. The achievement of an effective implementation of the enumerative perfect phylogeny algorithm, allowes us to add a number of biologically motivated and practical extensions described below.

Faster DAG Construction. The directed acyclic graph (DAG), as defined in [8], is used for all enumeration and optimization algorithms.

Let $G \subset S$ be a proper cluster where SUBPHYLOGENY(G) in some execution of PERFECTPHYLOGENY(S) did not return failure. The critical first step of DAG construction is to enumerate all the valid roots of SUBPHYLOGENY(G). Indexed by each connected $H_i \subset G$, these form the set $Ext(G, H_i)$ (see [8]). The description in [8] did not give sufficient detail, but implies brute force. We evaluated our brute force method, and two novel alternatives.

Our alternative methods utilize an incompatibility graph $I(G) = (V, E)$ where the vertex set V consists of all 'very good' pairs (G, H_i) of G as defined above. For each pair (G, H_u) and (G, H_v), if H_u and H_v have a non-empty intersection, then there is an edge (u, v) in E. For each pair (G, H_u) and (G, H_v) if $sv(H_u)$ is incompatible with $sv(H_v)$ then there is an edge in E. The following is true for $I(G)$.

Lemma 1: Every valid decomposition of G corresponds to a maximal independent set in $I(G)$.

Next, we state a useful bound.

Lemma 2: The canonically labeled root $cl(r_G)$ (Def 9 of [8]) of P_H is completely specified by examining any $r-1$ cardinality subset of proper clusters in $\{H_1\ \ldots\ H_t\}$ when $t > r - 1$.

Lemmas 2, 1, and a bound on k-indepepent sets [2] give a lower asymptotic bound than Lemma 6 of [8] for the size of $Ext(G, H)$ and suggest alternative methods of generation.

Lemma 3: There are at most $\lfloor n/k \rfloor^{k-(n \bmod k)} \lfloor n/k + 1 \rfloor^{(n \bmod k)}$ roots of H. [2]

To compute $Ext(G, H)$ we evaluated three implementations. We evaluated the brute force, up to n choose k, method of [8] against two methods that utilize $I(G)$ and the aforementioned lemmas. $MaxIS$ enumerates all maximal independent sets of $I(H)$ using the algorithm of [10]. k-MaxIS implements the recursive function proposed in [2] for enumerating all independent sets of $I(G)$ that are either maximal or size k but none larger than k.

DAG Algorithms. Utilizing the DAG, PerfectPhy will list all of the trees compatible with the input characters. PerfectPhy will also count the number of trees by dynamic programming over the DAG. In addition to these previously characterized algorithms, PerfectPhy implements other biologically motivated algorithms utilizing the DAG. As mentioned previously, PerfectPhy will find the most parsimonious perfect phylogeny (the tree of smallest size). It can also find a tree with maximum support, where support is defined for each edge as the fraction of trees in which it occurs. While

Table 1. Average execution times for significant problem instances. (30 trials)

n, m	4 state (nucleotide)	10 state	20 state (amino acid)
50,50	0.005s	0.024s	0.303s
100,100	0.028s	0.113s	1.55s
500,500	3.21s	17.6s	239s
1000,1000	51.9s	271s	2,320s
2000,2000	529s	2,590s	19,300s

Table 2. Average execution times for tree construction on nucleotide input data with missing values occurring with rate r for 100 characters and n taxa. (30 trials)

	$n = 20$	$n = 40$	$n = 60$	$n = 80$	$n = 100$
$r = 0$	0.008s	0.013s	0.019s	0.025s	0.028s
$r = 0.1$	0.015s	0.066s	0.197s	0.710s	3.13s
$r = 0.2$	0.030s	0.397s	9.20s	231s	20025s

these problems are NP hard, in practice PerfectPhy can effectively solve them for data of practical size.

4. Results

We benchmarked the construction algorithm on a range of datasets including nucleotide and amino acid data to empirically demonstrate its effectiveness. We used a population genetics simulator to generate binary data consistent with the infinite sites model. To convert a binary matrix to a k-state matrix, k-1 adjacent characters are combined. In each set of characters, the binary states labeling each taxon will map to up to k multi-state characters. Finally, missing data may be randomly removed from the matrix. A complete description of this simulator appears in [6]. Results for the construction algorithm are shown in Table 1.

In Table 2 we assessed a practical extension to the basic construction algorithm, datasets with missing values. We evaluated datasets of 100 nucleotide characters while varying the rate of missing data (r) and the number of taxa (n). For each character missing values are replaced with states not seen in the data [9]. While this implementation has a worst case complexity of $O(2^{2n}nm^2)$, the execution time is practical for lower missing data rates.

Implementation

In Table 3 we evaluate three different algorithms for DAG construction. Implementations that utilized the incompatibility graph, MaxIS and k-MaxIS, were faster than the brute force method suggested in [8] of choosing up to k proper clusters, epecially for larger k. We did not see a large speedup when comparing k-MaxIS to MaxIS.

Table 3. DAG construction time (k-MaxIS / MaxIS / n choose k enumeration; 60 trials)

n, m	$k = 4$	$k = 10$
50,50	14ms / 12ms / 49ms	41ms / 48ms / 421s*
100,100	48ms / 46ms / 1.11s	160ms / 176ms / 261s
500,500	3.99s / 3.95s / 4.38s	13.3s / 13.4s / 44.7s*
1000,1000	30.8s / 31.4s / 33.6s	127s / 124s / 142s*

*Note that actual average execution time is higher for these cases because one or more instances timed out at 20min.

We compared the execution times of the pointer table to the trie, and found them to be very close with a slight advantage going to the pointer table. The primary advantage of the pointer table is its asymptotically smaller space complexity, which in practice also yields a smaller datastructure.

Table 4. Average percent increase in [time / space] using trie over pointer table (80 trials)

n,m	$k = 4$	$k = 10$	$k = 20$
50,50	3.07% / 165%	2.75% / 189%	1.55% / 174%
100,100	2.80% / 386%	2.95% / 516%	1.60% / 490%
500,500	1.27% / 1886%	2.67% / 2749%	1.18% / 2957%
1000,1000	1.15% / 3775%	2.89% / 5522%	1.12% / 6525%

We timed the DAG construction algorithms, it can be seen in Table 4 that the trie was more often slower than the pointer table, though the differences in execution times were minor. We observed a much larger difference in the amount of memory used for the large problem instances.

To investigate space requirements of the implementation, we measured the DAG size as saved to disk (Table 5). While the worst case bound for the size of the DAG suggests space may be an issue, we utilized a relatively modest amount of memory for even the largest problems we found practical to solve in a timely manner. This led us to conclude that execution time was the dominant factor determining solvability.

Table 5. Average DAG Size in KiB (50 trials)

n,m	$k = 4$	$k = 10$	$k = 20$
50,50	6.13	11.2	33.6
100,100	19.2	32.6	89.5
500,500	355	572	881
1000,1000	1,310	2,170	2,980

Tree enumeration and optimization

For multi-state data, Table 6 shows the average number of minimal perfect phylogenies compatible with a problem instance. We observed that many trees can be compatible with an input dataset. This is particularly true as k increases, either from the number of native states observed in the data or because of missing data converted to new unobserved states.

Table 6. Average number minimal perfect phylogenies and compute time. (80 trials)

n,m	$k = 4$	$k = 10$	$k = 20$
50,50	3.40 (0.00998 s)	237 (0.0539s)	120,000 (3.68s)
100,100	2.48 (0.0473s)	495 (0.195s)	1,710,000 (3.30s)
500,500	1.66 (4.38s)	118 (15s)	292,000 (184s)
1000,1000	1.91 (33.5s)	11.0 (124s)	207,000 (1,000s)

The observation that many trees are compatible with a particular input dataset motivates the selection of a specific "best" tree. Appealing to parsimony arguments, we computed the smallest tree compatible with an input dataset. To quantify improvement and motivate the approach, the average difference in size between the smallest phylogeny and the phylogeny specified by an execution of the efficient algorithm given in [8] is shown in Table 7. We also calculated the rate at which the algorithm in [8] returns a tree that is the smallest size. For most of the datasets tested, a more parsimonious phylogeny was obtainable. For a faster result, we also implemented efficient algorithms that could do better than the algorithm in [8] but do not use the DAG. These do better, but lack the smallest tree guarantee. We also computed the tree of maximum edge support, where the score of an individual edge is the number of compatible trees it occurs in, and the score of a tree is the sum over all edges. We observed that these trees were usually distinct objects, but often close to the minimum size.

Software availability. The software can be downloaded at http://wwwcsif.cs.ucdavis.edu/˜gusfield.

Table 7. (top) Average increase in size and number of additional nodes when comparing the phylogeny given by the algorithm in [8] to the phylogeny of minimum size (bottom) The rate that a phylogeny given by the algorithm in [8] is of minimum size.

n,m	$k = 4$	$k = 10$	$k = 20$
50,50	2.74%, 1.34	7.14%, 4.81	11.3%, 8.30
100,100	1.28%, 1.20	3.16%, 4.28	5.43%, 8.49
500,500	0.328%, 1.387	0.675%, 4.25	1.08%, 8.34
1000,1000	0.145%, 1.18	0.350%, 4.40	0.552%, 8.38

n,m	$k = 4$	$k = 10$	$k = 20$
10, 10	15/80	1/80	0/80
50, 50	17/80	0/80	0/80
100, 100	19/80	1/80	0/80
500, 500	22/80	0/80	0/80
1000, 1000	24/80	0/80	0/80

References

[1] R. Agarwala and D. Fernández-Baca. A polynomial-time algorithm for the perfect phylogeny problem when the number of character states is fixed. *SIAM Journal of Computing*, 23(6):1216–1224, 1994.

[2] Jesper Makholm Byskov and Martin Farach-Colton. Algorithms for k-colouring and finding maximal independent sets. *SODA 2003*, pages 456–457, 2003.

[3] Z. Ding, T. Mailund, and Y. S. Song. Efficient whole-genome association mapping using local phylogenies for unphased genotype data. *Bioinformatics*, 24(19):2215–2221, October 2008.

[4] D. Gusfield. Efficient algorithms for inferring evolutionary history. *Networks*, 21:19–28, 1991.

[5] D. Gusfield. Haplotyping as perfect phylogeny: conceptual framework and efficient solutions. In *Proceedings of the sixth annual international conference on Computational biology*, pages 166–175. ACM, 2002.

[6] D. Gusfield. The multi-state perfect phylogeny problem with missing and removable data: Solutions via integer-programming and chordal graph theory. *Journal of Computational Biology*, 17(3):383–399, 2010.

[7] D. Gusfield. *ReCombinatorics: The Algorithmics of Ancestral Recombination Graphs and Explicit Phylogenetic Networks*. MIT Press, 2014.

[8] S. Kannan and T. Warnow. A fast algorithm for the computation and enumeration of perfect phylogenies when the number of character states is fixed. In *Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 595–603, Philadelphia, PA, USA, 1995. Society for Industrial and Applied Mathematics.

[9] C Semple and M Steel. *Phylogenetics*. Oxford University Press, 2003.

[10] Shuji Tsukiyama, Mikio Ide, Hiromu Ariyoshi, and Isao Shirakawa. A new algorithm for generating all the maximal independent sets. *SIAM Journal on Computing*, 6(3):505–517, 1977.