Notes 3 : Maximum Parsimony

MATH 833 - Fall 2012

Lecturer: Sebastien Roch

References: [SS03, Chapter 5], [DPV06, Chapter 8]

1 Beyond Perfect Phylogenies

Viewing (full, i.e., defined on all of \(X\)) binary characters as \(X\)-splits, the Splits-Equivalence Theorem and its proof via the Tree Popping procedure provide an algorithmic solution to the problem of checking whether a collection of binary characters are compatible and, if so, of constructing a minimal \(X\)-tree on which they are convex. (For a discussion of the more general non-binary, non-full problem, see [SS03, Chapters 4, 6].)

However, typical data may not be compatible and a more flexible approach is needed.

**DEF 3.1 (Parsimony Score)** Let \(C\) be a character state space with \(|C| \geq 2\), let \(\chi\) be a (full) character on \(X\) and let \(T = (T, \phi)\) be an \(X\)-tree with \(T = (V, E)\). Let \(\bar{\chi}\) be an extension of \(\chi\) to \(V\). The changing number \(\text{ch}(\bar{\chi})\) is

\[
\text{ch}(\bar{\chi}) = |\{\{u, v\} \in E : \bar{\chi}(u) \neq \bar{\chi}(v)\}|
\]

The parsimony score \(\ell(\chi, T)\) of \(\chi\) on \(T\) is the minimum value of \(\text{ch}(\bar{\chi})\) over all extensions of \(\chi\) on \(T\). For a collection \(C = \{\chi_1, \ldots, \chi_k\}\) of characters, the parsimony score of \(C\) on \(T\) is

\[
\ell(C, T) = \sum_{i=1}^{k} \ell(\chi_i, T).
\]

A maximum parsimony tree \(T^*\) for \(C\) is an \(X\)-tree which minimizes \(\ell(C, T)\) over all \(X\)-trees. The corresponding parsimony score is denoted by \(\ell(C)\). A natural generalization of the parsimony score is obtained by considering a metric \(\delta\) on \(C\) and replacing \(\text{ch}(\bar{\chi})\) with

\[
\sum_{e = \{u, v\} \in E} \delta(\bar{\chi}(u), \bar{\chi}(v)).
\]
We then use the notation $\ell_\delta$.

Given a character $\chi$ on $X$, an $X$-tree $T$ and a metric $\delta$ on $C$, one can compute the parsimony score $\ell_\delta(\chi, T)$ using a technique known as *dynamic programming*. Choose an arbitrary root $\rho$ on $T$. If $v = \phi(x)$ for some $x \in X$, for each $\alpha \in C$ let

$$l(v, \alpha) = \begin{cases} 
0, & \text{if } \chi(x) = \alpha, \\
+\infty, & \text{otherwise.}
\end{cases}$$

(By convention, we assume that the parsimony score is $+\infty$ if two different states are assigned to the same node of $T$.) For all $v \notin \phi(X)$, let $v_1, \ldots, v_m$ be the children of $v$ (i.e., the immediate descendants of $v$ in the partial order defined under the above rooting of $T$) and for each $\alpha \in C$ define

$$l(v, \alpha) = \sum_{i=1}^{m} \min_{\beta \in C} \{\delta(\alpha, \beta) + l(v_i, \beta)\}.$$

Then, it is straightforward to check by induction that

$$\ell_\delta(\chi, T) = \min_{\alpha \in C} l(\rho, \alpha),$$

which can be computed recursively from the leaves up to the root. For a collection of characters $C$, one can compute $\ell_\delta(C, T)$ by computing the parsimony scores of each character separately. Computing $\ell_\delta(C, T)$ is known as the *Fixed Tree Problem*.

As it turns out, computing $\ell_\delta(C)$ is much harder and, as we now explain, no efficient procedure is likely to exist for it.

## 2 Computational Complexity: A Brief Overview

We will use the notation $g(n) = O(f(n))$ to indicate that there is $K > 0$ such that $g(n) \leq K f(n)$ for all $n \geq 1$. The following definitions are intentionally informal. For more details, see [Pap94].

In a *search problem*, we are given an instance $I$ and we are asked to find a *solution* $S$, that is, an object that meets certain requirements (or indicate that no such solution exists). For example, in the SAT problem, we are given a formula $f$ over a Boolean vector $x = (x_1, \ldots, x_n)$ and we are asked to find an assignment for $x$ such that $f(x)$ is TRUE—if such an assignment exists.
An algorithm \( A \) for a search problem is said to be efficient if the number of elementary operations it performs on any instance \( I \) is bounded by a polynomial in the size of the input, that is, there is a constant \( K > 0 \) such that the running time of \( A \) on an input of size \( n \) is \( O(n^K) \).

**EX 3.2 (Fixed Tree Problem: Dynamic Programming)** Consider again the dynamic programming algorithm for solving the Fixed Tree Problem. For each vertex and each character state, we must perform a calculation which takes \( O(m|C|) \) where \( m \) is the number of children of that particular vertex. Summing over all vertices and character states, we get a running time of \( O(|V| \times |C|^2) \). The input here is a character, a tree and a metric, the size of which is \( O(|X| + |V| + |C|^2) \). Hence, the dynamic programming procedure is efficient.

**EX 3.3 (Maximum Parsimony: Exhaustive Search)** Suppose we are given a collection of characters \( C = \{\chi_1, \ldots, \chi_k\} \) on \( X \) and we seek to compute \( \ell_\delta^{(2)}(C) \) for a metric \( \delta \), where \( \ell_\delta^{(2)} \) indicates the maximum parsimony score restricted to binary phylogenetic trees on \( X \). The input size is \( O(k|X| + |C|^2) \). If we perform an exhaustive search over all binary phylogenetic trees and use dynamic programming on each of them to compute its parsimony score, the running time is \( O(b(|X|) \times |V| \times |C|^2) \), which is not polynomial in the size of the input.

**EXER 3.4 (Tree Popping)** Show that the Tree Popping algorithm is efficient. What is its running time?

The class of all search problems for which there exists an efficient algorithm is called \( P \). Another important class of search problems is \( NP \), which is defined as those problems for which a solution can be verified efficiently. For example, \( SAT \) is in \( NP \) as, given a solution \( x \), it is easy to check whether \( f(x) \) is \( TRUE \). (The standard definition involves decision problems which we will not discuss here.) An important conjecture is that \( P \neq NP \), that is, there exist problems in \( NP \) for which there is no efficient algorithm. In particular, it is possible a sub-class of \( NP \) consisting of the “hardest” problems within \( NP \) in the sense that the existence of an efficient algorithm for any such problem would lead to an efficient algorithm for any problem in \( NP \). Such problems are called \( NP \)-complete and require the notion of a reduction to be defined. A reduction from a search problem \( A \) to a search problem \( B \) is:

An efficient algorithm \( f \) that transforms any instance \( I \) of \( A \) into an instance \( f(I) \) of \( B \), together with another efficient algorithm \( h \) that maps any solution \( S \) of \( f(I) \) back into a solution \( h(S) \) of \( I \).
See [DPV06] for several examples of reductions. Then, the class of \textbf{NP}-complete problems is defined as follows:

A search problem is \textbf{NP}-complete if all other search problems in \textbf{NP} reduce to it.

It is well-known that \texttt{SAT} is \textbf{NP}-complete.

### 3 Maximum Parsimony is \textbf{NP}-complete

A natural way to transform an \textit{minimization problem} such as Maximum Parsimony into a search problem is to add to the input a threshold \( g \) and ask for a solution with objective function below \( g \). Then, the following was shown by Graham and Foulds:

**THM 3.5 (Complexity of Maximum Parsimony)** *The search problem corresponding to Maximum Parsimony is \textbf{NP}-complete.*

In other words, it is unlikely that an efficient algorithm exists for Maximum Parsimony.

### Further reading

The definitions and results discussed here were taken from Chapter 5 of [SS03] and Chapter 8 of [DPV06]. The rigorous theory of computational complexity is described at length in [Pap94]. The proof that Maximum Parsimony is \textbf{NP}-complete can be found in [GF82].

### References


Notes 3: Maximum Parsimony
