EFFICIENT CONSTRUCTION OF ACCURATE MULTIPLE ALIGNMENTS AND LARGE-SCALE PHYLOGENIES

by

Travis John Wheeler

A Dissertation Submitted to the Faculty of the

DEPARTMENT OF COMPUTER SCIENCE

In Partial Fulfillment of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY

In the Graduate College

THE UNIVERSITY OF ARIZONA

2 0 0 9

THE UNIVERSITY OF ARIZONA
GRADUATE COLLEGE

As members of the Dissertation Committee, we certify that we have read the dissertation prepared by Travis John Wheeler
entitled Efficient Construction of Accurate Multiple Alignments and Large-Scale Phylogenies
and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy.

_____ Date: 08/18/09
John D. Kececioglu

_____ Date: 08/18/09
Michael J. Sanderson

_____ Date: 08/18/09
Alon Efrat

_____ Date: 08/18/09
David R. Maddison

_____ Date: 08/18/09
Bongki Moon

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copies of the dissertation to the Graduate College.
We hereby certify that we have read this dissertation prepared under our direction and recommend that it be accepted as fulfilling the dissertation requirement.

_____ Date: 08/18/09
Dissertation Director: John D. Kececioglu

_____ Date: 08/18/09
Dissertation Director: Michael J. Sanderson

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at the University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED: Travis John Wheeler

ACKNOWLEDGEMENTS

I owe a debt of gratitude to the many inspirational and supportive people who have enabled this work, a few of whom are mentioned here.

I grew up surrounded by books, and by parents who not only preached life-long learning, but live it. Both entered teaching roles in academics late in life, and probably won't leave until a visit from the Grim Reaper. That dedication is in my blood, and I'm thankful for it. None of this would have been possible without their nurturing support and encouragement.

I also had the good fortune to have what you might call a second tier of parental figures, in the parents of my close friends Sam Mordka and Ephron Rosenzweig. I spent a great deal of time at their homes in my formative years, and Maurice Mordka and Michael Rosenzweig both provided significant encouragement and important guidance.

While an undergraduate English major, I had the good fortune of taking a life-changing course from Stephen Zegura. He certainly doesn't remember me, but his enthusiastic and brilliant introduction to genetics in that class, and the two others I later took with him, paved the way for the path I have followed since.

I worked for a couple years as an undergraduate research assistant with Yaron Ziv, then a PhD student in Michael Rosenzweig's lab. Like many in Michael's lab, Yaron had entered graduate studies after several years out of the academic system. Together, they taught me a fair amount about the science of ecology, but a lot more about life - especially life as an older PhD student (which is what I became, after going almost 10 years between earning my bachelor's and starting my PhD).

After my undergraduate degree, I went to work at Intuit. Nicki Graham and Stephanie Yablonski were instrumental in allowing me to teach myself computer programming on the company's dime (sure, I was producing results . . . but computer programming felt more like fun than work). While at Intuit, I met my peerless wife, Karen Tempkin, whose love and support have made me a better person. Her return to school for her MBA was the final reminder that it was time to go back to academia, and it was clear that the field should be at the intersection of old and new joys, genetics (now genomics) and computer programming.

David Maddison was kind enough to hire me to design and develop a database-driven upgrade to the Tree of Life Web Project (ToL, http://tolweb.org) while I took undergraduate courses in computer science to pave the way for my eventual graduate college admission. He and Katja Schultz introduced me to much of what I know about phylogenetics. David has continued his encouragement and support since I left the ToL, now serving on my PhD committee.

Before entering the CS PhD program, I was lucky enough to sit in on a course on computational biology, taught by John Kececioglu. The topic was enthralling, and John's excellent lectures made it accessible to a newcomer to the field. I joined his research group's weekly discussions, and knew I'd found a home. Since then, John has always been available whenever I've needed him, and he's been a constant source of guidance, encouragement, and information. He is the deepest thinker I've known, and I can only hope that some of that has rubbed off on me. The first part of this dissertation describes work done under his advisement, and it's good qualities are a testament to his excellent guidance.

During the course of my studies, I became increasingly interested in methods of statistical inference and machine learning, and was welcomed into the discussions of Kobus Barnard's research group. I am thankful for the inviting attitude of the group, and for the insight shared.

About three years ago, Michael Sanderson became my co-advisor. We've had innumerable enlightening conversations, covering a wide range of topics. One of those topics led to the work described in the second part of the dissertation. Mike has been an invigorating presence in my life, and I thank him for that. In a more mundane matter, I also thank him for allowing me access to his computing cluster; the experiments given in the paper are computationally demanding, and were to a great extent only possible because of his cluster.

I've made a great many friends in John's and Mike's research groups, the CS department, and also through the IGERT-in-genomics fellowships that have funded almost all of my graduate studies. I'm sure I'll unfairly leave someone off this list, but I'll always appreciate my stimulating, enlightening, and entertaining discussions with Dean Starrett, Eagu Kim, Somu Perianayagam, Jeff Good, [SC]eline Hayden, Matt Dean, Joel Wertheim, Patrick Degnan, David Hearn, Karen Cranston, and Darren Boss.

Finally, I thank Misawa Katoh for assistance with re-engineering `MAFFT`, specifically for the experiments described in Section 4.6, and Morgan Price for many recent discussions involving fast phylogeny inference in `NINJA` and `FastTree`.

During the course of my studies, I have been supported by the NSF through three separate grants: (1) a PhD Fellowship from the University of Arizona NSF IGERT Genomics Initiative Grant DGE-0114420 (3.5 years), (2) Grant DBI-0317498 (1 year), and (3) a PhD Fellowship from the University of Arizona NSF IGERT Comparative Genomics Initiative Grant DGE-0654435 (1 year).

This work is dedicated to three generations of family:

To my mother, Kathy Abramowitz, and father, Norty Wheeler: I didn't need to look very far for examples of commitment to lifelong learning and contribution to the greater good. Their love and nurturing were everything I could have hoped for.

To my wife, Karen Tempkin: it's possible that I could have made this academic journey without her in my life, but I'd have been a much lonelier, poorer, and grumpier person without her love, support, and supreme patience. I also wouldn't have a third generation to include in this dedication.

To my children, Elliot and Allison Wheeler: they didn't exactly improve the quality of my research . . . but that's not the point, is it? It's pretty hard to take yourself too seriously when your kids think your greatest talents are tickling and making fart noises.

TABLE OF CONTENTS

TABLE OF CONTENTS — *Continued*

TABLE OF CONTENTS — *Continued*

TABLE OF CONTENTS — *Continued*

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES — *Continued*

ABSTRACT

A central focus of computational biology is to organize and make use of vast stores of molecular sequence data. Two of the most studied and fundamental problems in the field are *sequence alignment* and *phylogeny inference*. The problem of multiple sequence alignment is to take a set of DNA, RNA, or protein sequences and identify related segments of these sequences. Perhaps the most common use of alignments of multiple sequences is as input for methods designed to infer a phylogeny, or tree describing the evolutionary history of the sequences. The two problems are circularly related: standard phylogeny inference methods take a multiple sequence alignment as input, while computation of a rudimentary phylogeny is a step in the standard multiple sequence alignment method.

Efficient computation of high-quality alignments, and of high-quality phylogenies based on those alignments, are both open problems in the field of computational biology. The first part of the dissertation gives details of my efforts to identify a best-of-breed method for each stage of the standard form-and-polish heuristic for aligning multiple sequences; the result of these efforts is a tool, called `Opal`, that achieves state-of-the-art 84.7% accuracy on the `BAliBASE` alignment benchmark. The second part of the dissertation describes a new algorithm that dramatically increases the speed and scalability of a common method for phylogeny inference called neighbor-joining; this algorithm is implemented in a new tool, called `NINJA`, which is more than an order of magnitude faster than a very fast implementation of the canonical algorithm, for example building a tree on 218,000 sequences in under 6 days using a single processor computer.

CHAPTER 1

INTRODUCTION

Molecular sequencing technology has ushered in a new era in biology, one in which gene function and structure can be predicted based only on sequence (though confirmation of predictions still depends on experimentation), and in which historical relationship of sequences and species can be inferred based on fairly objective sequence-based methods rather than subjective classification schemes.

The task of sequence alignment is to take a set of molecular sequences (DNA, RNA, or protein) and identify regions with functional, structural, or evolutionary relationship. This is typically done by organizing the sequences into a matrix in which each row corresponds to one sequence, and the sequences are spread out so that related residues (amino acids or nucleotides) share a column. Alignments can be used, for example, to:

- identify conserved regions (conservation suggests that the region encodes some function);
- estimate molecular structure (much better predictions of structure are possible from a high-quality alignment of multiple sequences than from a single sequence);
- find signals of natural selection (for example based on Ka/Ks ratio, which may indicate positive or purifying selection [57]);
- make statements of evolutionary ancestry, where the statement may be something like "sequence A shares common ancestry with sequence B", or "the phylogeny (family tree) of this set of sequences is ...".

Take as an example a single long-dead organism that is ancestor to a collection of extant organisms, and consider a single gene in that organism. If we could track the mutations that were endured by copies of that gene in the course of the tree-like process of descent leading to present-day sequences, we would find that substitutions, insertions, and deletions occurring in organisms at various historical points would be passed on to their descendants. This evolutionary process can be seen to induce a multiple sequence alignment in which each column contains characters that share descent based on (perhaps faulty) replication from the same ancestral character, while positions in sequences are shifted one way or another to account for historical insertions and deletions (see Figure 1.1). Of course,

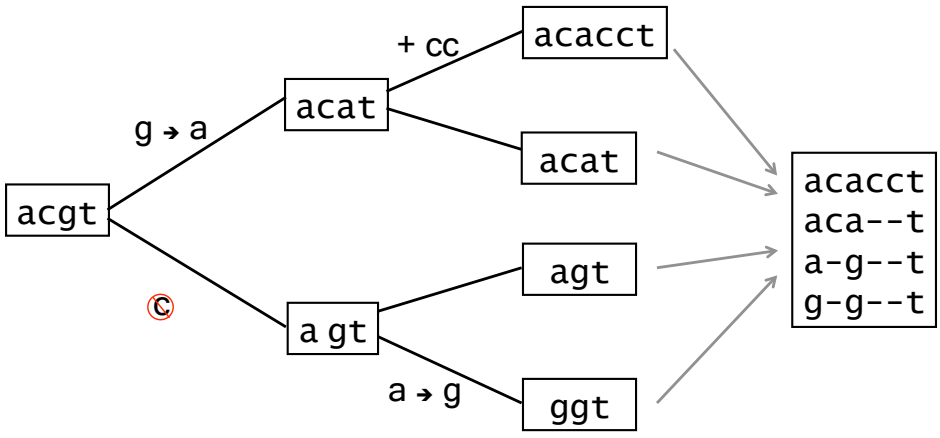


Figure 1.1: Descent with modification, in the form of substitutions, insertions, and deletions. The evolutionary history of the sequences induces an alignment of the extant sequences.

---

such a history is generally unknowable, but is the aim of much of computational biology: recovering the alignment without knowing the ancestral sequences or true phylogeny, and recovering the true phylogeny without knowing the true alignment.

This dissertation covers these two closely related topics, exploring a wide range of methods applied to aligning multiple sequences, as well as an approach that dramatically increases the speed and scalability of a common method for phylogeny inference. The majority of methods applied to phylogeny and alignment treat the two problems as essentially independent. Phylogeny methods typically take a single alignment as input, and treat it as if it were a true statement of homology of characters, despite clear evidence that uncertainty in alignment correlates with problematic tree inference [85]. Meanwhile, the standard method of aligning multiple sequences makes use of a quickly estimated phylogeny (guide tree) in an internal step, despite concerns that choice of guide tree may bias analyses made based on the alignment [70]. Though groundbreaking progress has been made in the area of simultaneously inferring both alignment and phylogeny [91], the computational complexity of such methods is such that only several sequences can be compared. In the genomics era, in which alignments of hundreds, or even tens of thousands, of sequences may be the subject of analysis, it is still of value to consider alignment and phylogeny as stages in a process, as is done in this dissertation.

1.1. Overview

The remainder of this chapter will provide a very basic introduction to the topics of sequence alignment and phylogeny inference. The following chapters are broken into two parts.

Part 1 (Chapters 2 through 4) will relate to the problem of multiple sequence alignment. Chapter 2 gives an overview of the standard multiple sequence alignment framework, describing the stages used in what we call the *form-and-polish* method. It also presents an investigation of new and established methods at each stage of the form-and-polish method, and identifies best-of-breed methods for each. Chapter 3 gives details on a promising new method for one of the stages, involving the *weighting of sequence pairs* used to overcome the dominance by over-represented groups of the standard sum-of-pairs score. Finally, Chapter 4 provides in-depth discussion of a new method for using *alignment consistency* to avoid errors that are a common by-product of the standard method of alignment construction.

Part 2 (Chapters 5 through 7) relates to a new algorithm that dramatically increases the speed and scalability of a common method for phylogeny inference, called neighbor-joining. Chapter 5 describes the canonical neighbor-joining algorithm, and places it in context of related work. Chapter 6 give details of the new algorithm, which reduces run time via a two-staged filtering approach, and overcomes memory limitations by employing data structures that are external-memory-efficient. Chapter 7 presents experimental results supporting the success of this new algorithm, highlighted by the result of producing a tree for an input of 218,000 sequences in under 6 days on a single computer.

Finally, we conclude with summary remarks and possible future directions of research.

1.2. Introduction to sequence alignment

When aligning sequences, it is common to consider three kinds of mutation: *substitution* (in which one nucleotide is replaced with another, perhaps as the result of a faulty replication), *insertion* (in which a stretch of characters is added into a sequence), and *deletion* (in which a stretch of characters is removed). Other mutations, such as reversal or recombination, are much more rare, require more complex representative techniques, and add computational complexity, and are thus not considered in standard sequence alignment methods. Mutations are subject to selection, with the result that they are relatively less likely to be observed in regions of functional importance, and of the those that are observed in functional regions, most have minimal functional impact, for example replacing one amino acid with another one bearing similar biochemical properties.

The task of sequence alignment is to take a set of molecular sequences and identify related regions. A sequence alignment is typically represented as a matrix, in which each row corresponds to one sequence, and the characters of each sequence

are shifted left and right so that related residues share a column. Characters in a column need not be identical, just homologous (i.e. the result of a substitution). The way sequences are spread out is by placing one or more spacer characters (typically '-', which is effectively the null character) between characters in the sequence, as in the induced alignment in Figure 1.1. These spacer characters are often called *gap characters*. A maximal run of consecutive spacer characters in a row is called a *gap* (alternatively an *indel*), and is presumed to be the result of a single mutational event, either an insertion or deletion (or possibly an artifact of incomplete sequencing, in the case of terminal gaps).

Finding a biologically meaningful alignment depends on assigning scores to substitutions, and costs to gaps. In protein sequence alignment, the substitution scores typically come from a 20×20 matrix (e.g. BLOSUM [51] or VTML [79]), in which the score of a substitution from residue a to residue b may be viewed as the logarithm of the odds ratio ($\sigma(a,b) = \log \frac{p(ab)}{p(a)p(b)}$), based on frequencies in an alignment database of observed alignments (where $p(a)$ is the observed frequency of character a among all sequences in the database, and $p(ab)$ is the frequency of observing an a aligned with a b in the database). In these matrices, identities have highest (positive) score, while substitutions between amino acids with very different biochemical properties have lowest (and negative) score. The substitution matrix for nucleotides is a simpler 4×4 matrix. Gaps are typically assigned an affine cost, in which a gap incurs a length-dependent cost, with cost of λ for each character in the gap, and a per-gap cost γ (so the total cost of a gap of length ℓ is $\gamma + \ell\lambda$). In this scoring scheme, the goal is to line characters up into an alignment A so as to maximize the sum of the induced substitution scores, minus the induced gap costs. This is the scoring scheme used in most related work.

It is a trivial matter to convert substitution scores to costs, so that identities have low cost and substitutions have high cost, thus turning the task into an equivalent cost-minimization problem. This is the scoring scheme used in this work, so that the cost of a pairwise alignment is

$$\text{cost}(A) \;:=\; c\gamma + \ell\lambda + \sum_{a \to b} \sigma(a,b), \tag{1.1}$$

where the number of gaps is c, the total length of those gaps is ℓ, and the summation is over the cost of all substitutions ($a \sim b$). The matter of maximizing scores or minimizing costs is mostly a semantic issue, but one that informs the terms used in describing other tools (score) and our models (cost).

The technique of dynamic programming can be used to find a pairwise alignment with optimal cost. The key to using this technique is the observation that an optimal alignment of a pair of prefixes ends in a column (involving either a substitution or a gap), and if that final column is removed, the remaining alignment must be an optimal alignment of the shortened prefixes. Under the condition that

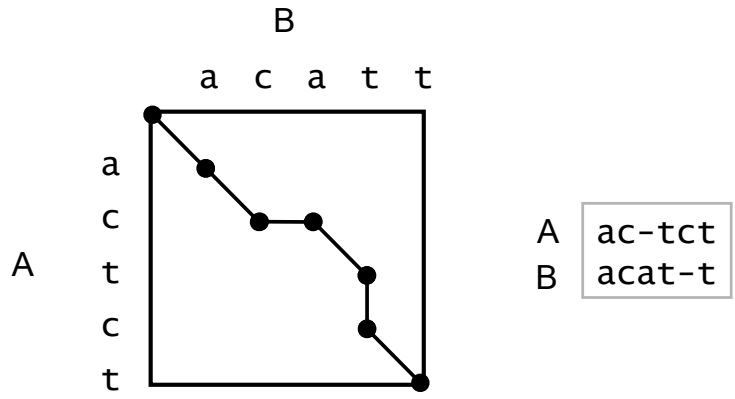


Figure 1.2: Path in a pairwise alignment dynamic programming table. A diagonal edge leading into cell (i, j) corresponds to a substitution column containing a_i and b_j. A vertical edge leading into cell (i, j) corresponds to a column containing a_i in a gap between b_j and b_{j+1}. A horizontal edge leading into cell (i, j) corresponds to a column containing b_j in a gap between a_i and a_{i+1}.

---

the per-gap cost γ is 0, this leads to the following recurrence for computing the optimal cost, $C(i, j)$ of an alignment of prefixes $(a_1 \ldots a_i)$ and $(b_1 \ldots b_j)$ of sequences A and B:

$$C(i,j) = \min \left\{ \begin{array}{l} C(i-1, j-1) \ + \ \sigma(a_i, b_j), \\ C(i-1, j) \ + \ \lambda, \\ C(i, j-1) \ + \ \lambda \end{array} \right\}. \tag{1.2}$$

(The recurrence in the case $\gamma > 0$ is a bit more complex, requiring that optimal costs must be kept at each cell for each of the 3 possible columns that might end a prefix alignment. Details are unnecessary for the purposes of this introduction, but can be found in [50].)

The optimal cost of aligning the full sequences A and B, of length m and n respectively, is then the value $C(m, n)$. This value may be found by filling in an $m \times n$ table, where the i^{th} row corresponds to the i^{th} position of A, and the j^{th} column to the j^{th} position of B, so that the cost at a cell (i, j) corresponds to the cost of aligning prefixes $(a_1 \ldots a_i)$ and $(b_1 \ldots b_j)$. This dynamic programming table is filled out in row-major order (row by row, from upper left to lower right), so that the three values required by the recurrence in equation 1.2 are always available when the value at a new cell is computed. Thus, a single pass through the table is

sufficient to find the cost of the optimal alignment. Since constant time is required to compare the 3 values at each cell, this gives a $O(nm)$ algorithm for finding the optimal cost. An alignment corresponds to a path through the alignment graph, as in Figure 1.2. A path (and corresponding alignment) with the optimal cost can be found by the standard dynamic programming method of backtracking through this table [50].

Generalization of this approach to finding good alignments of multiple sequences requires generalization of the scoring method. In principle, if we knew the phylogeny relating sequences, and the ancestral sequences labelling internal nodes, it would be reasonable to seek an alignment of extant and ancestral sequences that minimize the sum of pairwise alignment costs over all edges in the phylogeny. However, the phylogeny and internal labellings are in general unknown. Even given a fixed phylogeny, the problem of labeling internal nodes so as to produce a minimal cost summing over all edges (often called Tree alignment [96]) is NP complete [114]. For this reason, the standard practice is to optimize the sum of the alignment costs over all pairwise alignments induced by the multiple alignment. This scoring scheme has the odd trait of effectively treating each sequence as if it evolved from all the others, but has proven to be useful.

With this sum-of-pairs scoring scheme in place, the dynamic programming method for two-sequence alignment can be generalized to aligning multiple sequences by filling in a k-dimensional table for k sequences. The exponential size of this table makes exact alignment infeasible for more than several sequences, even with clever methods for limiting the visited space in that table [71, 44], so heuristics are used in practice. An overview of the standard heuristic for aligning multiple sequences, which we call the *form-and-polish* approach, are given in Chapter 2, with detailed investigation of the stages of that method described in Chapters 2 through 4.

1.3. Introduction to phylogeny inference

The diversity of life, and the DNA that encodes its function, arose through a branching pattern of evolution: a copy of the DNA of an organism is passed on to that organism's offspring, possibly with mutation. Over time, this descent with modification [21] can be seen to induce a tree-like relationship on a family of present-day sequences.

The task of molecular phylogenetic inference is to take as input a set of sequences, usually already aligned, and create a tree that recovers their ancestral history. This tree may be what is called a *gene tree*, which defines the ancestral history of descendants of a single gene (or perhaps a concatenated set of genes), or a *species tree*, which defines the history of the species containing those genes. Due to incomplete lineage sorting [73, 74], which may cause incongruence between the trees of different genes sequenced from the same individuals, these may not agree.

In this work, discussion of phylogeny inference relates to gene tree construction.

Such a tree is valuable both in its own right as a statement of the relationship of entities, and also as a framework for understanding a wide range biological processes (e.g. rates of evolution, selective pressures) and for addressing biodiversity issues.

Inferred trees have two components: the topology (branching order) and edge lengths. Trees are typically binary, and most commonly-used inference methods produce unrooted trees. A standard method of establishing a root for an unrooted tree is to include a known outgroup (a sequence that is known not to break the monophyly of the remaining sequences): the root is then placed along the edge connecting that group to the rest of the tree.

A variety of methods are applied to the problem of phylogeny inference, with the notable ones including Parsimony [33], Maximum-likelihood [34], Bayesian [117, 56], and distance based methods Minimum Evolution [94, 22] and Neighbor-joining [95].

The first three in this list all assign a value to a tree that depends on possible (unknown) ancestral sequences, then seek trees with optimal value. The number of possible topologies grows super-exponentially in the number of sequences [35], leading parsimony and maximum-likelihood (ML) to use local-search heuristics to try to find good trees, while Bayesian phylogeny inference employs a Markov-Chain Monte-Carlo approach to sample from the probability landscape. In all three methods, positions are usually assumed to be independent, and gap characters are treated either as an extra character or as missing data. Parsimony aims to find a tree that minimizes the number of changes over the tree that are required to attain the observed sequences; an efficient dynamic programming approach (linear in the size of the input alignment) allows computation of optimal labellings for internal nodes for a fixed tree. ML seeks to find a tree that has highest likelihood under a probabilistic model of sequence evolution; the likelihood for a fixed tree is computed by marginalizing over all possible internal node labellings, and is computed with a linear time dynamic-programming algorithm. Bayesian inference is also model based, but samples from the space of possible trees, rather than seeking a single peak in the probability space.

Distance-based methods do not consider internal node labels. Rather, they begin by computing distances between all pairs of sequences, and build trees based only on these pairwise distances. Under the minimum evolution (ME) framework, edge lengths for a fixed topology are those that minimize the sum of the squares of the differences between the input pairwise distances and those induced by the tree's edge lengths. The minimum-evolution tree is the tree with topology minimizing the sum of these edge lengths, and ME methods use local search heuristics to try to find a tree near this minimum. Neighbor-joining (NJ) is a greedy heuristic for finding the balanced minimum evolution tree [40], is proven statistically consistent (guarantees the correct tree given a sufficiently long alignment), and is guaranteed to produce the correct tree even when the input pairwise distances are somewhat

noisy (see Chapter 5 for details).

The largest published tree inferred by the Parsimony method contains 73,060 taxa [42], while ML has currently achieved limits of around 50,000 sequences in testing (pers. comm. Mike Sanderson, Karen Cranston). Both methods require weeks of computation time to build such trees. Bayesian inference can only handle inputs on the order of hundreds of sequences. There is little hope that these methods will scale to the point of rapidly building trees for inputs of more than 200,000 sequences in the near future. Since trees of this size are now possible given the available sequence data, methods that can scale up are desirable. The canonical NJ algorithm is faster than Parsimony or ML, but its $O(k^3)$ run time and $O(k^2)$ space requirements for k sequences, make it unfit for such large inputs. In Part 2, we describe a new algorithm for computing a neighbor-joining tree that improves over the run time of the canonical algorithm by more than an order of magnitude. This new method reduces run time by using a filtering mechanism to dramatically reduce the number of computations done for each iteration of the tree formation process, and uses external-memory efficient data structures to overcome space constraints. As an example, it produces a tree with 50,000 sequences in about 9 hours using a single processor on a system with 4GB of RAM, and a tree with 218,000 sequences in fewer than 6 days on a similar system with 12 GB RAM.

PART 1 ACCURATE MULTIPLE ALIGNMENT WITH THE FORM-AND-POLISH STRATEGY

CHAPTER 2

BEST-OF-BREED METHODS FOR FORM-AND-POLISH ALIGNMENT

All widely-used tools for multiple sequence alignment at essence seek an alignment that minimizes the *sum-of-pairs cost*: the weighted sum of the costs of all pairwise alignments induced by the multiple alignment. Optimal multiple alignment with sum-of-pairs scoring is NP-complete [114] which motivates the use of good heuristics.

Most commonly-used tools use a heuristic called progressive alignment [36] which has two steps: (1) construct a binary *merge tree* whose leaves are the input sequences and whose internal nodes arrange the sequences into groups, and (2) merge these groups leaf-to-root over the tree by combining the alignments at the two children of a node into one alignment at their parent to form successively larger alignments at internal nodes. When merging the groups at the two children of a node into one group at their parent, the two sets of sequences in the groups are usually combined by aligning alignments [45, 63] or aligning profiles that compactly represent alignments [46, 64]. When this merging process reaches the root, the last pair of clusters is merged at the root of the tree, and it has formed an alignment of all the input sequences.

This hierarchical approach is greedy: the alignment between two sequences in a group is not altered when that group is merged with another group. Consequently, errors made in early merges remain in the final alignment, and may lead to further misalignment in later merges. One approach to correcting such errors is to apply a second stage we call *polishing* [9], which refines the alignment by repeatedly splitting its sequences into subsets and realigning their induced subalignments. Another strategy is to reduce early errors using an approach called *consistency* [84, 25], which tries to avoid making such errors in the first place. Consistency approaches assign position-specific substitution scores for a pair of sequences A and B and a pair of positions i and j that depend on the support for the substitution between i and j from the pairwise alignments of sequences A and B to all other sequences C.

The combination of all these steps makes up the core of what we call the *form-and-polish* strategy for multiple alignment. In total, this strategy consists of seven stages:

1. Choose alignment parameters.

2. Estimate distances between pairs of sequences.

3. Construct a merge tree.

4. Compute weights for sequence pairs.

5. Compute consistency-modified scores.

6. Merge alignments over the merge tree from stage 3.

7. Polish the alignment formed at the root of the merge tree in stage 6.

Form-and-polish alignment methods make use of these stages, with stages 1, 2, 3, and 6 being obligatory, while stages 4, 5, and 7 are optional. The stages will be described in detail below. The order of presentation will differ to simplify exposition.

2.1. Survey of methods and tools

Current tools use a profusion of methods for each stage. The most widely-used multiple alignment tool, `ClustalW` [110], relies on a complex scoring scheme in which substitution scores and gap penalties are adjusted according to features of the aligned sequences, including divergence, length, hydrophobicity of amino acids, and proximity of neighboring gaps.

`PRRP/N` [48] improved accuracy using polishing, in which an alignment is repeatedly split into subalignments, which are then realigned.

`T-Coffee` [83] and its predecessor `Coffee` [84] introduced alignment consistency, with resulting increase in accuracy. `T-Coffee` assigns position-specific substitution scores based on a mixture of support provided by optimal global pairwise alignments and a small set of high scoring local pairwise alignments.

Early versions of `MAFFT` [60] substantially increased alignment speed without sacrificing `ClustalW`-like accuracy through a host of ideas including scoring system modifications, use of the Fast Fourier Transform to speed up profile alignment and a two-stage strategy for constructing an alignment that first builds a draft alignment using a merge tree based on distance estimates from k-mer frequencies, and then obtains a pre-polished alignment based on a merge tree built using distances derived from the pairwise alignments induced by the draft alignment.

`Muscle` [29] further improved speed with a variety of algorithm-engineering approaches, and matched `T-Coffee`'s accuracy by employing reduced terminal gap costs and a measure called log-expectation to score alignments of profiles.

`ProbCons` [25] introduced probabilistic consistency, which assigns position-specific substitution scores based on a measure of expected accuracy derived from a hidden Markov model using a three-way consistency transform. It also offers a column reliability score that estimates the likelihood that each column represents a correct alignment of residues.

Recent versions of `MAFFT` [59] incorporate a heuristic for `T-Coffee`-like consistency, resulting in a substantial improvement in accuracy. The results are slower than the fastest version of `MAFFT`, but still quite fast.

Details of the consistency methods of T-Coffee, MAFFT, and ProbCons are provided in Chapter 4.

2.2. Overview

It can be difficult to determine which methods are contributing to the accuracy of these various alignment tools, and should be in a best-of-breed tool.

In this chapter, we carefully study the impact of the standard methods for each stage of the form-and-polish method of multiple sequence alignment, and offer several new methods that substantially improve accuracy. The greatest gains in accuracy come from two simple new ideas: (i) estimating distances between sequences for merge tree construction by normalized alignment costs, and (ii) polishing the final alignment using 3-partitions of the input sequences induced by cutting pairs of edges in the merge tree.

By combining the best methods for each of the stages of the form-and-polish strategy, we obtain a new tool, which we call Opal, whose accuracy matches the state-of-the-art as measured on the standard benchmark datasets.

In the next section, we describe our experimental methods. In Section 2.4, we study methods for constructing the merge tree. Section 2.5 considers methods for merging alignments. Section 2.6 assesses the effect of weighted sum-of-pairs. Section 2.7 explores methods for polishing alignments. Section 2.8 inspects the value of alignment consistency. Section 2.9 examines the impact of parameter choice, and finally Section 2.10 compares the combined approach to current tools.

It is reasonable to ask if the order in which stages are addressed might impact the results. We considered many combinations of the methods for each stage, and the best-of-breed choices are consistent under any ordering. The only impact of the order appears to be on the scale of improvements attributed to a method. For example, if polishing were presented earlier, its gains would be larger, while the gains seen in other methods would be smaller.

2.3. Methodology

The standard practice for evaluating multiple alignment tools is to use benchmark datasets of reference alignments that are usually based on structural alignment of proteins. When comparing methods for a stage, and comparing alignment tools, we evaluate accuracy by measuring the recovery of reference alignments from three standard suites of protein alignment benchmarks: BAliBASE3.0 [109], SABmark1.65 [113], and PALI2.5 [7]. These suites, which are used by many comparative studies, of course represent only a sample of the types of inputs biologists face.

BAliBASE is a collection of 218 reference alignments based on structural alignments with manually-arranged gaps, exhibiting a variety of phylogenetic and structural characteristics. We limited our tests to the 163 alignments with no more than 40 sequences, as our focus is measuring accuracy and not speed.

The SABmark benchmark contains 627 alignments, each containing at most 25 sequences, that cover the entire known fold space found in the SCOP[80] classification of protein families. Each benchmark is a collection of pairwise structural alignments that are not necessarily consistent with one multiple alignment.

PALI contains 1655 alignments of all SCOP families constructed by structural multiple alignment without hand curation. We used a subset of 102 alignments consisting of all reference alignments with at least 7 sequences that have nontrivial gap structure.

In our experiments the measure of accuracy is mainly the so-called *SPS score* [6]: the percentage of pairs of aligned positions from the reference alignment that were correctly recovered by the computed alignment. In some cases we also report the so-called *TC score* [6]: the percentage of columns from the reference alignment that were completely recovered; this score is appropriate for BAliBASE and PALI, but not SABmark, since there is no column to be recovered (only possibly inconsistent pairwise alignments).

For BAliBASE and PALI, both scores are measured on their *core blocks*: those columns in the reference alignment that are deemed reliable by the benchmark (typically through strong support from a structural alignment). SABmark does not provide core blocks, so accuracy is measured against all columns of the benchmark alignment pairs, regardless of their true reliability. This fact partially accounts for the much lower recovery rates seen for SABmark than for BAliBASE and PALI, though the divergent nature of the SABmark alignments is certainly responsible for much of the difference.

We use the above suites of benchmarks in experiments to determine the best method for each of the eight identified stages of the form-and-polish strategy for multiple alignment. While this may appear to ignore interactions between methods for different stages, results not shown here show the best method for a given stage is independent of the choices at other stages. The best method is also generally independent of which suite of benchmarks is considered.

2.4. Merge tree

Constructing a merge tree involves (1) grouping sequences hierarchically, based on (2) a measure of distance between sequences. We study these two aspects in turn.

2.4.1. Grouping sequences

Progressive alignment tools construct the merge tree using one of a number of similar sequential clustering algorithms, in which the tree is built in a stepwise manner. In the general step, a distance (see Section 2.4.2) is known for each pair of sequence clusters, and the two closest clusters are merged into a new cluster, which defines an internal node of a binary tree. For a cluster *ab* that is the merge

Table 2.1: Clustering methods for constructing the merge tree.

Method	Cluster ab minimization criterion	Updated distance from ab to c
NJ	$(d_{ab} - \sum_c (d_{ac} + d_{bc}))/(n-2)$	$(d_{ac} + d_{bc} - d_{ab})/2$
UPGMA	d_{ab}	$(d_{ac} + d_{bc})/2$
MST	d_{ab}	$\min\{d_{ac}, d_{bc}\}$
MST+UPGMA	d_{ab}	$\alpha \min\{d_{ac}, d_{bc}\}+$ $(1-\alpha)(d_{ac}+d_{bc})/2$
DAD	d_{ab}	Cost of aligning ab to c

of clusters a and b, new distances $d_{ab|c}$ to all other clusters c are calculated as a simple combination of d_{ac}, d_{bc}, and possibly d_{ab}. Merging continues until one group remains, which is the root of the tree. The tree may be rerooted at a different node, for instance to balance root-to-leaf path lengths, or based on integration of a known outgroup.

Beginning with distance $d_{a|b}$ (see Section 2.4.2) between each pair a,b, the various methods discussed choose the pair to merge, then calculate a distance for the new cluster to all others. We study five clustering schemes, one new, as shown in Table 2.1.

Neighbor-joining Neighbor-joining [95, 105] is the method used by `ClustalW` and `T-Coffee`. It is a heuristic for construction of a minimum evolution tree [40], and is generally regarded as among the best of the fast distance-based methods at producing the true evolutionary tree for the sequences.

Let G be the current set of groups during the merging process, and $d_{a|b}$ be the current distance between a pair a, b of groups. neighbor-joining merges the groups a and b that minimize

$$(|G| - 2)\ d_{a|b}\ -\ \sum_{c \in G - \{a,b\}} \Big(d_{a|c}\ +\ d_{b|c} \Big). \tag{2.1}$$

The new distance $d_{ab|c}$ between the merged group ab and all other groups c is

$$d_{ab|c}\ :=\ \frac{d_{a|c}\ +\ d_{b|c}\ -\ d_{a|b}}{2}. \tag{2.2}$$

The canonical algorithm for neighbor-joining takes $O(k^3)$ time for k sequences. See Part 2 for significant discussion of this method.

UPGMA and MST The unweighted-pair group-method with arithmetic-mean or UPGMA [101], and minimum spanning tree or MST [20], are simpler approaches that run in $O(k^2)$ time. Both merge the pair a, b of groups with minimum distance $d_{a|b}$, but differ in how they define the distance $d_{ab|c}$ from the merged group ab to all other groups c. UPGMA sets $d_{ab|c}$ to the average $\frac{1}{2}(d_{a|c}+d_{b|c})$, while MST uses $\min\{d_{a|c}, d_{b|c}\}$.

We also considered the mixture UPGMA + MST used by MAFFT, which for $0 \leq \alpha \leq 1$, sets $d_{ab|c}$ to the convex combination

$$d_{ab|c} \;=\; \alpha \, \min\big\{d_{a|c}, d_{b|c}\big\} \;+\; (1-\alpha)\,\frac{1}{2}\,\big(d_{a|c} \;+\; d_{b|c}\big). \tag{2.3}$$

DAD A shortcoming of the above methods is that distances are derived from sequences only at initialization. When group ab is formed, the new distances $d_{ab|c}$ are calculated from original sequence distances, which ignores the constraints on sequence pairs across groups ab and c imposed by the alignments for these groups.

We evaluated several new methods that take such constraints into account. The best of these, which we call dynamic alignment distance or DAD, computes $d_{ab|c}$ by aligning the current alignments for ab and c to obtain an alignment $\mathcal{A}$ for group abc, and then taking for $d_{ab|c}$ the *minimum* of the distances measured on the pairwise alignments in $\mathcal{A}$ induced by sequence pairs across groups ab and c (analogous to MST). The distance measure on pairwise alignments can be any of those from Section 2.4.2.

We also considered variants that use the *average* of the distances of the induced pairwise alignments between ab and c (analogous to UPGMA), and that add in the average distances between abc and all *other* groups d, but the above method performed the best.

DAD does not perform as well as the seemingly less-informed methods neighbor-joining, UPGMA, and MST. While there is more information in the aligned sequences, alignments against large groups are more constrained than those against small groups, so larger groups tend to have higher distances. This causes smaller groups to be preferentially merged next, which leads to a balanced merge tree even when this is undesirable.

Comparison Table 2.2 shows a comparison of the *accuracy* of these methods on three suites of benchmarks in terms of their SPS score. As our baseline for the other stages, we measure the initial sequence distances using percent identity over a compressed alphabet (Section 2.4.2), merge alignments using pessimistic gap counts (Section 2.5), and use unweighted sum-of-pairs (Section 2.6), no polishing (Section 2.7), no consistency (Section 2.8), and default parameters (Section 2.9). For UPGMA + MST we use $\alpha = 0.9$, as in MAFFT. The best accuracies are in bold.

The multiple alignment literature [29, 25] suggests that using a merge tree that is similar to the correct evolutionary tree is less important for alignment accuracy

Table 2.2: Comparison of clustering methods.

Tree method	BAliBASE	SABmark	PALI	average
MST	**79.4**	**44.1**	79.8	**67.8**
UPGMA + MST	79.2	44.0	80.2	**67.8**
UPGMA	78.0	42.7	**80.5**	67.1
neighbor-joining	77.4	42.1	77.2	65.6
DAD	78.2	43.5	73.0	64.9

than using a tree that groups similar sequences first. Our results agree with this suggestion: generally the methods based on minimum spanning trees outperform the others. For the merge tree method in the rest of the chapter, we chose the simpler method MST.

2.4.2. Measuring distances

The standard measure of distance between two sequences is based on *percent identity*: the percentage of matched positions in an optimal pairwise alignment that are identities. (The actual distance used is the complement of percent identity.) Many tools modify percent identity by the Kimura correction for multiple substitutions at a locus [65], and measure it over a compressed alphabet that groups amino acids with similar characteristics into equivalence classes, which we call *compressed identity*. MAFFT and Muscle compute a draft alignment from distances based on k-mer frequencies on the way to their initial alignment based on percent identity.

Percent identity is a coarse measure of similarity, while alignment cost is a more refined measure that can be obtained with no overhead. We therefore tested a new distance measure, which we call *normalized alignment cost*, that simply normalizes the cost of an optimal pairwise alignment by dividing by the average length of the two sequences (where alignment cost uses affine gap penalties, as discussed in Section 2.5). In a sense this generalizes percent identity and compressed identity to the full spectrum of substitutions while also taking into account gaps. The distance measure used in ProbCons, expected accuracy, is similar in concept to normalized cost. It cannot be tested here, because it depends on the hidden Markov model foundation of ProbCons.

Comparison Table 2.3 shows a comparison the accuracy of these three distance methods. We use an MST merge tree, pessimistic gap counts, no weighting, no

Table 2.3: Comparison of distance methods.

Distance method	BAliBASE	SABmark	PALI	average
normalized cost	**81.6**	**48.2**	**83.0**	**70.9**
compressed identity	79.4	44.1	79.8	67.8
percent identity	78.5	43.5	79.9	67.3

polishing, no consistency, and default parameters. Identity measures have Kimura correction.

Normalized costs give a striking boost in accuracy, that persists even after polishing (not shown). Results in the rest of the chapter depend on normalized costs as the distance measure.

2.5. Merging alignments

Merging two multiple alignments of disjoint groups of sequences into one alignment of all the sequences, which has been called group-to-group alignment [45, 46] and aligning alignments [64, 63], is central to both forming an initial multiple alignment and polishing a final alignment.

When forming an initial multiple alignment, the merge tree T is processed from the leaves to the root. Each leaf is labeled by an input sequence, which may be viewed as a trivial alignment. As each internal node v is processed, the alignments $\mathcal{A}$ and $\mathcal{B}$ labeling the children of v are combined into one alignment $\mathcal{C}$ of all the sequences at the leaves of the subtree for v. During the polishing stage, the final alignment is repeatedly split into subalignments $\mathcal{A}$ and $\mathcal{B}$ that are recombined into an updated alignment $\mathcal{C}$. Clearly the quality of the final alignment strongly depends on how subalignments are merged in both stages.

When merging alignments $\mathcal{A}$ and $\mathcal{B}$ to form $\mathcal{C}$, we take as our objective to optimize the *sum-of-pairs score* [17] of alignment $\mathcal{C}$, which is the sum of the scores of all induced pairwise alignments in $\mathcal{C}$. This is the objective used in all commonly-used tools today. The sum-of-pairs score may be weighted, as discussed in Section 2.6. When merging $\mathcal{A}$ and $\mathcal{B}$ to form $\mathcal{C}$, the columns within subalignments $\mathcal{A}$ and $\mathcal{B}$ are preserved within $\mathcal{C}$, as pictured in Figure 2.1.

Computing an optimal $\mathcal{C}$ given $\mathcal{A}$ and $\mathcal{B}$ using sum-of-pairs scoring has been called the problem of *aligning alignments* [64], and was first considered by Gotoh [45]. Scores of the alignments are usually evaluated using *affine gap penalties*, where a gap of length ℓ in a pairwise alignment has cost $\gamma + \ell\lambda$, for constants γ

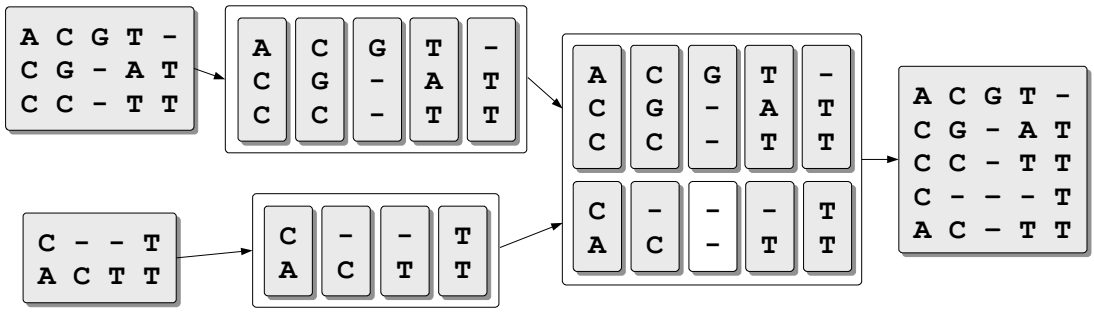


Figure 2.1: Aligning alignments is a merging of two alignments where the columns of one alignment are aligned as units with the columns of the other, resulting in a new multiple alignment. (Reproduced with permission from [103])

---

and λ. (A gap of length ℓ is a maximal run of either ℓ insertions or deletions.) The constants γ and λ are called the gap open and extension penalties, and may have different values for terminal or internal gaps. When computing an optimal merge $\mathcal{C}$ by dynamic programming, the essential difficulty is in correctly counting the total number of gaps incurred in the induced pairwise alignments of $\mathcal{C}$.

We study two basic ways of computing the merge $\mathcal{C}$ by aligning alignments: using *exact* gap counts, which yields a merge that has optimal sum-of-pairs score; and using *pessimistic* gap counts, which is a fast heuristic that may yield a suboptimal merge.

Exact counts Surprisingly, computing an optimal merge $\mathcal{C}$ of two alignments $\mathcal{A}$ and $\mathcal{B}$ with affine gap penalties is NP-complete [72]. While this shows there is likely *no* algorithm that computes an optimal merge and is fast in the worst case, nevertheless Kececioglu and Starrett developed an exact algorithm [63] that computes an optimal merge and is remarkably fast in practice. To optimally align two alignments, each having k sequences and n columns, their algorithm takes worst-case time $O(5^k n^2)$. Extensive experiments, however, show empirically that it runs in $O(k^2 n^2)$ time on biological data [63].

In this study, we compute an optimal merge $\mathcal{C}$ with exact gap counts, using an implementation of the algorithm of [63].

Pessimistic counts Another approach to computing the merge $\mathcal{C}$ is to avoid the difficulty of determining exact gap counts by instead using an approximation called pessimistic gap counts [64]. Pessimistic counts were introduced under the name quasi-natural gap costs by Altschul [2]. This approximation overestimates the true number of gaps by assuming, in cases where the number of gaps started by a multiple alignment column is not determined by the preceding column, that the

Table 2.4: Comparison of alignment methods.

Merge method	BAliBASE	SABmark	PALI	average
exact	**82.4**	**48.4**	**84.0**	**71.6**
pessimistic	81.6	48.2	83.0	70.9

number of gaps started attains its largest possible value. The benefit of pessimistic gap counts is that the merge of two alignments that is best under this estimate can always be found efficiently: with an alphabet of size $|\Sigma|$, computing the best pessimistic merge of two alignments, each having k sequences and n columns, takes worst-case time $O(n^2 \cdot \min(|\Sigma|, k))$ with $O(nk)$ preprocessing (see Chapter 8 in [103]).

Most multiple alignment tools use some version of profile alignment [46, 64, 60, 29], to merge alignments. It is entirely possible that the pessimistic heuristic, which is also a profile method, is outperforming other profile methods, though we do not study that here.

Comparison Table 2.4 compares the accuracy (SPS score) of exact and pessimistic gap counts when merging alignments leaf-to-root on the tree. We use an MST merge tree with normalized costs, no weighting, no polishing, no consistency, and default parameters.

Exact gap counts are consistently superior to pessimistic gap counts, though only slightly. Our experiments (not shown) indicate that exact gap counts continue to slightly outperform pessimistic counts after polishing. We use exact counts in the rest of the chapter, but note that the performance of the pessimistic heuristic is encouraging, and suggests that it is a reasonable choice for inputs so large that even the surprisingly fast exact-counts algorithm is too slow to use in practice.

2.6. Sequence-pair weights

Sum-of-pairs scoring of multiple alignments is a potentially biased scoring measure: if the input sequences are not independent but instead over-sample some groups compared to others, the higher number of pairwise alignments to an over-sampled group can dominate the alignment score. This greater contribution of an over-sampled group to the score will tend to drive the multiple alignment toward improving the pairwise alignments to such groups at the price of worsening the pairwise alignments to under-sampled groups, thus degrading the overall quality of the alignment.

Several schemes have been proposed to correct for this bias by nonuniformly weighting the pairwise alignment scores in the sum-of-pairs measure. We study three such schemes. The first is new, and the other two are the schemes most widely used by current alignment software. Brief descriptions are given here; full details of the methods are provided in Chapter 3.

All these schemes assign weights to pairs of input sequences on the basis of a tree T whose leaves correspond to the sequences, and that has edge lengths ℓ_e for all edges $e \in T$. Each scheme assigns a weight w_{ij} to a pair i, j of leaves in T that depends on both the topology of T and its edge lengths. Suppose that in a multiple alignment $\mathcal{A}$ of the sequences, the score of the induced pairwise alignment of the sequences associated with leaves i and j is s_{ij}. The *weighted sum-of-pairs score* of alignment $\mathcal{A}$ using these weights is

$$\sum_{i,j} w_{ij}\, s_{ij} \ . \tag{2.4}$$

Influence weights Our new method of assigning a weight w_{ij} to a pair i, j of leaves depends on quantifying the *influence* of one leaf on another. Informally, the influence of j on i, $\omega(i, j)$, is computed by rerooting the tree at i, then assigning a weight of 1 to i and distributing that weight among the remaining leaves in a manner that depends on shape of T and edge lengths.

The function ω is not necessarily symmetric, but we can define a symmetric weight on a pair of leaves by the geometric mean of their influences:

$$w_{ij} \ := \ \sqrt{\omega(i, j)\, \omega(j, i)} \ . \tag{2.5}$$

We call the w_{ij} obtained in this way, *influence weights*.

Influence weights have several nice properties. For k sequences, the weights w_{ij} can be computed in time $O(k^2)$, given tree T and its edge lengths, which is optimal. They also avoid the anomalous behavior exhibited by the other two methods described below, as discussed in chapter 3.

Covariance weights Perhaps the best-known weights for sum-of-pairs multiple alignment are those of Altschul et al. [3]. Of the two weighting schemes they suggest, their second scheme, which they call rationale 2 weights, has the more rigorous basis and is the one that has been more widely adopted. We call their second scheme, *covariance weights*.

Covariance weights depend on the extent to which paths between two different pairs of sequences share internal edges (their covariance). In principle, they are computed with a formula that inverts a matrix of these $O(k^4)$ covariances, but a run time of $O(k^2)$ is achieved using an involved algorithm described in [4]. This scheme is not directly used in common software; instead Gotoh's 3-way method [47] of approximating covariance weights is normally used, for example in weighting sequences during the polishing stages of `MAFFT` and `Muscle`.

Table 2.5: Comparison of weighting methods.

Weighting method	BAliBASE	SABmark	PALI	average
uniform	**82.4** / **53.6**	**48.4**	84.0 / 57.3	**71.6** / **55.5**
influence	82.2 / 53.3	**48.4**	84.1 / **57.7**	**71.6** / **55.5**
division	82.2 / 53.4	48.2	**84.3** / 57.3	**71.6** / 55.4
covariance	82.1 / 53.2	**48.4**	83.5 / 57.5	71.3 / 55.4

Division weights The program ClustalW introduced a simple weighting scheme that has been incorporated into other alignment software as well, such as in the construction stages of MAFFT and Muscle. We call ClustalW's scheme, *division weights*.

Briefly, division weights are computed by rooting the tree, then dividing the length of each edge in the tree among the leaves under that edge, finally computing leaf-pair weights as the product of the collected leaf totals. The w_{ij} for k sequences can be computed in time $O(k^2)$.

Comparison Table 2.5 compares the accuracy of the weighting methods across the benchmarks. We use an MST merge tree with normalized costs, exact gap counts, no polishing, no consistency, and default parameters. (Using pessimistic gap counts and polishing we find the same ranking of weighting methods.)

Covariance weights are computed exactly using matrix inversion, based on the original method of [3] (i.e. not Gotoh's approximation). The edge lengths ℓ_e on the MST tree are computed from normalized costs d_{ij} by fitting edge lengths to T so as to minimize the sum of the differences between path lengths ℓ_{ij} and the distances d_{ij}. These optimally-fitted edge lengths are efficiently computed by solving a linear program. Details of this linear program are given in Chapter 3. On BAliBASE and PALI, we also report the TC score as the second measure of quality, since it is less distorted by overrepresentation of groups than the SPS score.

Surprisingly, and in contrast to what has generally been suggested in the literature [47, 29], *unweighted* sum-of-pairs (the uniform row of the table) performs as well as all three weighting schemes. Even for inputs with the largest numbers of sequences, where overrepresentation is more likely, weighting continues to give no benefit under both measures of quality.

This behavior might be due to the fact that most benchmark alignments do *not* contain strongly overrepresented groups. BAliBASE references 2 and 3 do provide a set of inputs that are designed such that they contain somewhat oversampled

Table 2.6: Comparison of weighting methods on BAliBase refs 2 and 3.

Weighting method	BAliBASE refs 2 and 3 (SPS/TC)
influence	**83.3** / 30.7
uniform	82.8 / 31.4
division	82.5 / 30.8
covariance	81.5 / **31.5**

groups. Table 2.6 compares recovery of the weighting methods on the 28 inputs we used from these two references. Here, the SPS and TC numbers show disagreement regarding the best performing method. Since uniform weighting is most consistently successful, even on these inputs manifesting overrepresented groups, we use uniform weighting for the remaining tests in this chapter.

2.7. Polishing

The progressive alignment method forms an alignment at each internal node by aligning the columns of the alignments at that node's children, keeping those columns intact. The result is that errors made in early stages can disrupt the quality of the final alignment. Responses to this problem in the literature come in two forms: (1) avoid errors in the first place (consistency-based schemes [84, 25]), and (2) fix errors after they've been made (polishing, aka refinement [9]; see [52] for review of methods). We consider polishing methods in this section, and consistency in the next.

To our knowledge, all previously implemented techniques perform polishing by splitting sequences into two groups, resulting in two induced alignments of subsets $\mathcal{A}$ and $\mathcal{B}$ of the sequences. The subalignments $\mathcal{A}$ and $\mathcal{B}$ induced on these groups are realigned, without altering the columns of $\mathcal{A}$ or $\mathcal{B}$. Realignment of $\mathcal{A}$ and $\mathcal{B}$ is done as when merging alignments in Section 2.5 by aligning profiles [46, 64] or aligning alignments [45, 63]. The resulting alignment is retained if its score improves. This process is a form of local search, and is repeated for a fixed number of iterations or until there is no improvement.

Though 2-partitions may be formed in a number of ways, the methods in common tools either randomly split sequences into two groups [25], or perform *tree dependent restricted partitioning* [52], which considers only those partitions that can be formed by cutting an edge on the merge tree. Tree-based partitioning may be iterated exhaustively over the edges of the tree [29], or edges may be repeatedly cut

at random [59]. Tools using random choices tend to do many iterations: `ProbCons` by default does 100 iterations, and `MAFFT` does 1000.

We considered several alternatives, and present results of the three best performers (with combinations) here:

Exhaustive 2-cut We implemented a tree-based method we call exhaustive 2-cut that cuts tree edges and realigns until there is no improvement. Since a tree with k leaves has $O(k)$ edges, if there is an edge whose cut gives improvement this finds it within a linear number of realignments.

Rather than scanning tree edges in a fixed order [29], we dynamically order the edges e by a measure $\Phi(e)$ of their potential for improvement. For edge e, let $P(e)$ be the set of pairs of sequences that are on opposite sides of the partition given by cutting e, let c_{ij} be the cost of the pairwise alignment induced on sequences i and j in the current multiple alignment, and let d_{ij} be the cost of their optimal pairwise alignment. We use the potential

$$\Phi(e) \;:=\; \frac{\sum_{(i,j)\,\in\,P(e)} \left(c_{ij} - d_{ij}\right)}{|P(e)|}\;. \tag{2.6}$$

These potentials are updated after several cuts alter the alignment (a default of five). This approach yields a slight speedup in convergence over the exhaustive 2-cut method used in `Muscle` [29] while attaining the same accuracy.

Random 3-cut Consider the situation where two sequences in a large alignment are misaligned. The 2-cut method cannot separate these two sequences from the rest of the input and realign them without interference from all other sequences. This can be easily accomplished, however, by instead partitioning into *three* groups. We examined a variety of methods for three-way partitioning, both tree-based and not. We report the best one here, which we call *random 3-cut*. To our knowledge, this is the first time 3-cut polishing has been implemented.

This method partitions the sequences by cutting *two* tree edges selected at random. (A tree with k leaves has $O(k^2)$ such cuts, so an exhaustive approach would be slow.) The resulting groups a, b, and c are merged in two steps by realigning a and b to form group ab, and then realigning c with ab. We consider the three merge orders $ab|c$, $ac|b$, $bc|a$, and retain the best of the three if it gives improvement. This process is repeated until a time or iteration limit is reached. Most edges are near the leaves, so this method tends to split off two relatively small groups of sequences, enabling repair of errors between small groups followed by integration into the rest of the alignment.

In contrast to 2-cut, this method in essence alters the merge tree. We also considered 3-cut variants that reattach tree edges to reflect the merges of the three

Table 2.7: Comparison of polishing methods.

Polishing method	BAliBASE	SABmark	PALI	average
3-cut + on-the-fly	84.3	**50.2**	84.6	**73.1**
3-cut	84.2	49.7	**84.8**	72.9
2-cut	**84.4**	49.8	84.7	72.9
2-cut + on-the-fly	83.6	50.0	84.5	72.7
on-the-fly	83.3	49.6	84.4	72.4
none	82.4	48.4	84.0	71.6

groups, or that rebuild the tree on the affected paths up to the root, but surprisingly none gave better quality than the above method that does not change the tree.

On-the-fly An attractive idea is to polish subalignments as they are formed [106], rather than waiting for a complete alignment. This allows errors to be fixed before causing further misalignment. We implemented a version we call *on-the-fly* polishing: when an internal node v is created during merge tree construction, iterate over edges that are grandchildren or children of v, at each iteration cutting and realigning. This is repeated until no improvement is seen in one sweep through nearby offspring edges. This is similar to 2-cut, but scans a limited set of edges and operates while forming the initial alignment.

Combined We also consider the method of polishing both during and after alignment construction. Our method is similar to the *tree-based iterative* algorithm of [52], but with polishing at internal nodes restricted in depth, and post-polishing possibly using tree-based 3-cuts, rather than 2-cuts. We call this method *k-cut + on-the-fly*.

Comparison Table 2.7 compares the accuracy of polishing methods using MST trees with normalized costs, exact gap counts, no weighting, no consistency, and default parameters. For 3-cut we did 60 iterations; 2-cut and on-the-fly iterated until no improvement.

These results suggest that the post-polishing methods tend to converge to roughly the same alignments, and that depth-restricted on-the-fly polishing, though a much faster (not shown) way to gain improvement, is not capable of reaching this same convergence point on its own. It is worth noting that, for inputs with 30-40

sequences, the run time for exhaustive 2-cut is generally twice that of the 3-cut method. Also, using on-the-fly in conjunction with with exhaustive 2-cut slightly speeds up convergence, since it performs much of the same polishing work before the full compliment of sequences has been added.

In the rest of the chapter we use 3-cut + on-the-fly polishing.

2.8. Alignment consistency

Polishing, described in the last section, attempts to overcome the inherent error propagation of the progressive alignment method by cleaning errors after-the-fact. An alternate approach is to try to avoid errors in the first place, using a method known as *alignment consistency* [84, 25].

In the standard alignment scheme, two sequences are aligned such that the alignment optimizes, over all possible alignments, the sum of the substitution and gap scores. These scores are position-independent: the substitution score is based solely on the pair of characters being aligned (as from BLOSUM substitution score matrices [51]), and the per-gap and gap-extension costs are fixed (the γ and λ values of Section 2.5, possibly with different such fixed costs for internal and terminal gaps). Two *alignments* are aligned in an attempt to optimize the sum of the scores of induced pairwise alignments, with the constraint that the columns of each alignment remain intact.

In the alignment consistency framework, costs are modified in a per-position manner. Typically [84, 59, 25], this involves computing a consistency-modified substitution score for each position-pair (a_i, b_j). These modified position-pair scores are based on the amount of support found in other sequences C for the alignment of position a_i with position b_j. Details of how this support is calculated for position pairs differ from tool to tool, and are given detailed treatment in Chapter 4.

In the consistency framework, gap-related costs in these methods are usually ignored, so that only modified substitution costs are used. The argument for doing this is that gap costs are implicitly a part of the process of computing the modified position-pair costs, but this approach may lead to less reasonable gap structure than is typically recovered with an affine gap penalty. Our new method depends not only on modified substitution scores, but also on modified per-gap and and gap-extension costs.

As before, two alignments are aligned with the goal of optimizing the sum of scores of induced pairwise alignments, where the scores are now based on modified substitution scores.

Comparison We considered a wide variety of novel approaches for calculating modified alignment feature costs. Details are provided in Chapter 4. Results of the best-performing method are given in Table 2.8. We use an `MST` merge tree with normalized costs, exact gap counts, no weighting, and default baseline parameters.

Table 2.8: Comparison of consistency methods.

Consistency method	BAliBASE	SABmark	PALI	average
consistency, no polish	**84.0** / 56.1	49.6	**84.5** / **59.8**	72.7 / **58.0**
consistency, polish	83.6 / 56.2	49.9	83.7 / 57.2	72.4 / 56.7
no consistency, no polish	82.5 / 52.7	48.5	83.8 / 57.1	71.6 / 54.9
no consistency, polish	83.9 / **56.8**	**50.1**	84.3 / 57.9	**72.8** / 57.4

Results are given both with and without polishing, since consistency and polishing are alternate approaches for solving the same problem, and the results of one impact the efficacy of the other.

(The astute reader will notice that non-consistency numbers disagree slightly with those of the prior section. This is due to a minor change in the set of alignments used as input to the alternatives in this section).

These results show that the alignment consistency approach (without polishing) overcomes early errors of the progressive alignment method about as well as post-polishing does. Intriguingly, however, a mixture of the two approaches results in a worsening of the average alignment recovery. Reasons for this worsening of quality are discussed in Chapter 4.

Because consistency does not clearly improve results, and runs slower than no-consistency with polishing (even with speedup methods described in Chapter 4), the no-consistency option is used for the rest of the chapter.

2.9. Parameter advisor

Selecting gap penalties involves a good deal of guesswork [111] and most practitioners simply use the default values provided by modern software. Fortunately there are now well-designed suites of protein alignment benchmarks, and the sequences in these benchmarks presumably represent the kinds of protein inputs a tool will see in the wild, which suggests that parameters optimized on those benchmarks should be reasonably good choices (though this is likely not the case for DNA, where only coding-DNA- [18] and -RNA- [38] benchmarks are available). We look at the effect of parameter choice on accuracy from three perspectives: finding the best *default* input-independent values, determining how well a perfect input-dependent choice given by an *oracle* can perform, and designing an *advisor* that makes a good choice.

Default parameters

To select default parameters for aligning proteins with `Opal`, we trained primarily on `BAliBASE`. Based on results from doing inverse parametric sequence alignment using the tool `InverseAlign` [62], we fixed the substitution matrix at `BLOSUM62` [51] and identified a reasonable seed value for the gap open and extension penalties. We then enumerated a range of parameters around this seed, including variants with reduced terminal gap costs. In total we examined 774 parameter choices.

To quickly filter out poor parameter choices, we aligned all `BAliBASE` inputs with a fast version of `Opal` (`MST` with normalized cost, pessimistic gap counts, no weights, and no polishing), and identified the 100 choices with the best average recovery. We then selected a parameter choice from this set that had high recovery on all three benchmark suites using a more accurate version of `Opal` (same as above, but using exact gap counts and on-the-fly polishing). From this set of parameters we selected as our default the choice that had the highest recovery over all three suites of benchmarks. With `BLOSUM62` transformed to a cost matrix in the range [0,88], our *default* parameter choice has internal-gap open and extension penalties γ_I and λ_I, and terminal-gap open and extension penalties γ_T and λ_T, of

$$(\gamma_I, \lambda_I, \gamma_T, \lambda_T) \;=\; (60, 38, 15, 36).$$

A reduced terminal-gap open penalty is common, typically half the internal-gap open penalty [29], though a reduced terminal-gap extension penalty is not. This parameter choice was used in all results described in prior sections.

Parameter oracle

While the default performs well overall, it severely underperforms other choices on many benchmark alignments. (The default has accuracy more than 10% worse than the optimal choice on about 15% of the inputs). This leads us to ask what accuracy could be achieved if we had an *oracle* that could identify the best parameter choice for each input. We consider results with an oracle for purposes of comparison.

Parameter advisor

Though a true oracle is unattainable, it is possible to design an *advisor* that can choose an input-dependent parameter value that improves alignment quality. We are aware of only one other attempt at implementing a method to automatically choose parameters, called `MULTICLUSTAL` [118]. Our tests show that, for combinations of 3 parameters, Multiclustal's advisor performs poorly, typically choosing the worst of the 3 on nearly 50% of inputs.

We considered a variety of novel advisor methods, and describe two of them here.

Naive Bayes advisor Multiple sequence alignments present a broad range of features that may be useful in picking parameters for an input. `ClustalW` uses one of these, percent identity in optimal pairwise alignments, as part of its score-modification strategy. In addition to percent identity, we considered features related to spacer density (number of gap characters) and gap-start density in computed multiple alignments.

Our approach is to ask, for each parameter x among a small set of alternative parameter choices $\mathcal{P}$, "what is the range of observed feature values in alignments generated by x that have high accuracy?". The idea is that some parameters may work better with alignments that are, for example, more gappy than average, and that observing such traits may give support for using one parameter over another. The approach is then to identify, for a new input, which parameter results in an alignment with feature values most like those for which the parameter has worked well on training data.

We fit a Gaussian, G_{xj}, for each feature j to the set of observed values of that feature over all inputs $\mathcal{S}$ for which x is *good* (i.e. x results in an alignment with accuracy no more than 2% worse than the best parameter choice). Similar Gaussians, B_{xj}, are determined for *bad* inputs (x results in alignments at least 10% worse than the best parameter). Distributions were trained using alignments from `BAliBASE` for parameter choices in $\mathcal{P}$.

After determining these distributions, a quality value is computed for parameter choice x on input $\mathcal{S}$ as follows. Let $\mathcal{A}_x$ be the alignment of $\mathcal{S}$ under x, and let $f(j, \mathcal{A}_x)$ be the value for feature j of $\mathcal{A}_x$. Let $G(x, j, \mathcal{S})$ be the area under the curve G_{xj} of an interval sized ϵ around $f(j, \mathcal{A}_x)$, and likewise $B(x, j, \mathcal{S})$ under the bad Gaussian B_{xj}. Let g be the number of training inputs for which the parameter is good, and likewise b for for bad, so g/b is a prior belief of the quality of an alignment. Then the quality of a parameter x on an input $\mathcal{S}$ is

$$Q(x, \mathcal{S}) \;=\; \frac{g}{b} \frac{\prod_j G(x, j, \mathcal{S})}{\prod_j B(x, j, \mathcal{S})}. \tag{2.7}$$

Given a new input $\mathcal{S}$, the parameter choice that this advisor selects is

$$p \;:=\; \operatorname*{argmax}_{x \in \mathcal{P}} Q(x, \mathcal{S}).$$

This approach bears a strong resemblance to a common discrimination technique called a *naive Bayes classifier* [26].

Note that this approach makes an assumption that the various feature ranges are independent. Modification to this approach to account for possible covariance led to no improvement in advisor success.

Core column count advisor A simpler approach works as follows. Define a *core column* to be a column in a multiple alignment where at least a fraction α of its

rows have letters from the same character class in the compressed alphabet. (In the experiments, we use $\alpha = 0.9$, and used the compressed alphabet of [59].) Given a set $\mathcal{S}$ of input sequences to align and a candidate parameter choice x, let $\mathcal{A}_x$ be the alignment of $\mathcal{S}$ that results from using parameter choice x, and let $f(\mathcal{A}_x)$ be the number of core columns in $\mathcal{A}_x$.

The parameter choice that this advisor selects based on core columns is

$$p \;:=\; \operatorname*{argmax}_{x \in \mathcal{P}} f(\mathcal{A}_x).$$

In other words, it selects the parameter choice that gives an alignment with the most core columns (ties are broken in favor of shorter alignments).

Both of the above advisors perform similarly. We present results for the core column approach. The output alignment using either advisor is $\mathcal{A}_p$.

Comparison Table 2.9 compares the hypothetical accuracy of the oracle to that achieved using default parameters and the advisor. We picked a set $\mathcal{U}$ of twelve parameter choices that performed well on average and cover the domain of reasonable gap open and extension values, and applied the oracle to set $\mathcal{U}$. The advisor was applied to a smaller set $\mathcal{P} \subseteq \mathcal{U}$ of four parameter choices, choosing $p = (\gamma_I, \lambda_I, \gamma_T, \lambda_T)$ from the set

$$\mathcal{P} \;=\; \left\{ \begin{array}{l} (56, 38, 7, 36), \\ (58, 37, 7, 35), \\ (64, 37, 8, 37), \\ (64, 38, 32, 36) \end{array} \right\}.$$

(An alignment $\mathcal{A}_x$ must be computed by the advisor for each $x \in \mathcal{P}$, so a small set $\mathcal{P}$ is preferable.) We also considered running the oracle on $\mathcal{P}$.

Given a parameter choice, alignments were computed using the best methods from prior stages: `MST` trees with normalized costs, exact gap counts, no weighting, no consistency, and on-the-fly + 3-cut polishing.

The oracle clearly provides a large boost in recovery, and offers an intriguing target for further research. The improvement of the advisor over the default is modest, and might conceivably be the result of just fortuitous choices that exploit the variation in accuracy within set $\mathcal{P}$. A closer look, however, reveals that the advisor's performance is much better than random. For a given subset $\mathcal{S} \subseteq \mathcal{U}$, we can compare the accuracy of the advisor on $\mathcal{S}$ to that of the single best choice from $\mathcal{S}$. Our advisor outperforms the best choice for 60% of all subsets $\mathcal{S} \subseteq \mathcal{U}$ tested. It also outperforms the mean accuracy of the $x \in \mathcal{S}$ for 94% of all subsets.

Table 2.9: Comparison of parameter advisor methods.

Parameter method	BAli	SAB	PALI	average
oracle on $\mathcal{U}$	**87.0**	**54.4**	**87.1**	**76.1**
oracle on $\mathcal{P}$	86.2	52.9	86.2	75.1
advisor on $\mathcal{P}$	84.7	50.5	84.9	73.4
default	84.4	50.0	84.5	73.0

(If the advisor's parameter selection were random, we would expect it to beat the average accuracy 50% of the time.) By contrast the method in [118] outperforms the best choice from $\mathcal{S}$ for only 1% of the subsets, and the mean of $\mathcal{S}$ for less than 3%.

When comparing against other tools in the next section, we consider performance both with and without an advisor.

2.10. Discussion

The prior sections have examined seven stages in the form-and-polish strategy, and identified the best method for each stage. In Table 2.10 we summarize for all stages the net improvement in alignment quality gained by the best method over a standard method.

The baseline methods in the table are a UPGMA merge tree, percent identity for distances, pessimistic gap counts to merge subalignments, unweighted sum-of-pairs, no polishing, no consistency, and default parameters. We omit the stages of choosing weights and consistency which gave no improvement.

These results represent the outcome of a careful study of methods for each stage of the form-and-polish strategy for multiple alignment. This includes new methods for estimating distances, merging alignments, weighting pairs of sequences, polishing the alignment, employing alignment consistency, and choosing parameters.

A new merging method that optimally aligns alignments yields only a minor improvement over an approximate merging heuristic. The largest gains in quality come from new methods for estimating distances by normalized alignment costs, and polishing by 3-cuts on the merge tree; together these two boost recovery by more than 4%. The weighting and consistency methods did not result in improved accuracy as implemented, but are promising; details are given in the next two chapters. The best method for a stage is generally the same across all suites

Table 2.10: Assessing the accuracy gains by selecting best-of-breed methods selected at each stage.

Stage	BAli	SAB	PALI	average
(baseline)	78.0	42.7	80.5	67.1
tree	+1.4	+1.4	-0.7	+0.7
distance	+2.2	+4.1	+3.2	+3.1
merge	+0.8	+0.2	+1.0	+0.7
polish	+1.9	+1.8	+0.6	+1.5
parameters	+0.4	+0.3	+0.3	+0.3
(combined)	**84.7**	**50.5**	**84.9**	**73.4**

of benchmarks, suggesting that what we have singled out as best has not been overfitted to the data.

This best combination of methods yields a new tool we call Opal. In brief, the best-performing variant of Opal uses an MST merge tree, normalized costs for distances, exact gap counts to merge subalignments, unweighted sum-of-pairs, no consistency, and both on-the-fly and 3-cut polishing. A slight boost in accuracy can be gained using the parameter advisor.

In Table 2.11, we compare the accuracy of Opal to other commonly-used tools: ProbCons [25], MAFFT [59], Muscle [29], T-Coffee [83], and ClustalW [110]. On BAliBASE and PALI, the second quality measure we report is the percentage of reference columns completely recovered, the TC score [6].

Both the most accurate and baseline versions of Opal are shown. All other tools were run at their highest accuracy (for MAFFT this was their L-INS-i variant). We highlight two observations: (1) The results show that even the baseline version of Opal strongly outperforms ClustalW, which uses essentially identical choices at each stage. This suggests that the pessimistic heuristic for aligning alignments works much better than the heuristic used in ClustalW. (2) By combining the best methods at each stage, Opal attains accuracy on par with the state-of-the-art (namely ProbCons and MAFFT), even without using the alignment consistency method that is responsible for much of the success of those tools.

Though the experiments presented in this chapter were performed on protein benchmarks, Opal parameters have been learned for DNA and RNA alignments in a manner similar to that in Section 2.9. Table 2.12 compares the accuracy of Opal

Tool	BAliBASE		SABmark	PALI		Average	
	SPS	TC	SPS	SPS	TC	SPS	TC
MAFFT	**85.1**	**60.4**	49.2	84.3	**60.3**	72.9	**60.4**
ProbCons	84.5	57.9	50.1	84.8	60.0	73.1	59.0
Opal with advisor	84.7	57.9	**50.5**	**84.9**	59.6	**73.4**	58.7
Opal with default parameters	84.3	58.2	50.2	84.6	58.5	73.1	58.4
T-Coffee	80.1	54.3	46.7	81.4	55.0	69.4	54.7
Muscle	80.2	52.1	45.6	81.2	55.5	69.0	53.8
Opal baseline	78.0	48.0	42.7	80.5	50.2	67.1	49.1
ClustalW	73.2	41.6	44.0	74.5	44.3	63.9	43.0

Table 2.11: Comparison of the accuracy of Opal to commonly-used tools on protein benchmarks. For each tool and for each suite of benchmarks, the table reports the average accuracy of the tool across all benchmarks in the suite. The last columns report the average accuracy of a tool across all suites. Accuracy is measured by the *SPS score*, and where applicable, the *TC score* [6]. The SPS score is the percentage of pairs of aligned positions from the reference alignment that were correctly recovered; the TC score is the percentage of columns from the reference alignment that were completely recovered. The performance of Opal is shown both using an *advisor* method for choosing input-dependent values for gap penalties for scoring alignments, and using input-independent *default* values. The highest accuracies for a suite are in bold.

Table 2.12: Comparison of the accuracy of Opal to commonly-used tools on the RNA benchmark, BRAliBASE.

Tool	BRAliBASE (SPS/TC)
ProbCons	**86.9** / **76.8**
Opal	86.8 / 76.7
MAFFT	85.8 / 74.8
Muscle	85.1 / 74.3
ClustalW	82.2 / 71.0

to that of other tools on the BRAliBASE benchmark, which consists of mostly RNA alignments.

Opal was developed as a testbed for methods, with a focus on accuracy, but has been somewhat optimized for speed. On inputs of 40 sequences, its running time when using the exact algorithm for aligning alignments is about two orders of magnitude slower than ClustalW and Muscle, and about the same as the slowest tool, T-Coffee; the pessimistic heuristic for aligning alignments gives about an order of magnitude speedup. Over all benchmarks, the median run time for Opal with the exact algorithm, was less than 10 seconds, which was on an input of 20 sequences of length near 250.

CHAPTER 3

WEIGHTS FOR SEQUENCE-PAIRS RELATED BY A TREE

Sum-of-pairs scoring of multiple alignments is problematic when applied to sequences that have evolved on an evolutionary tree. The implicit assumption of sum-of-pairs is independence of sequences, but biological sequences are related under a tree-like evolutionary process, so the common case is that one sequence will tend to be more similar to some sequences than to others, and sequences often cluster into groups within the evolutionary tree describing their relationship. If the input sequences oversample some groups, the result of optimizing sum-of-pairs scoring will be to improve the pairwise alignments to such groups at the price of worsening the pairwise alignments to other groups. An example of this is seen in Figure 3.1.

Overview

Several schemes have been proposed to correct for this bias by nonuniformly weighting the pairwise alignment scores in the sum-of-pairs measure. The two schemes that are most widely used by current alignment software are described in Section 3.1. A new scheme, which we call *influence weights*, is easy to implement, overcomes the anomalies observed for the first two, and is the subject Section 3.2.

All these schemes assign weights to pairs of input sequences on the basis of a tree T whose leaves correspond to the sequences, and that has edge lengths ℓ_e for all edges $e \in T$. Each scheme assigns a weight w_{ij} to a pair i, j of leaves in T. We now discuss details of the calculation of these weights.

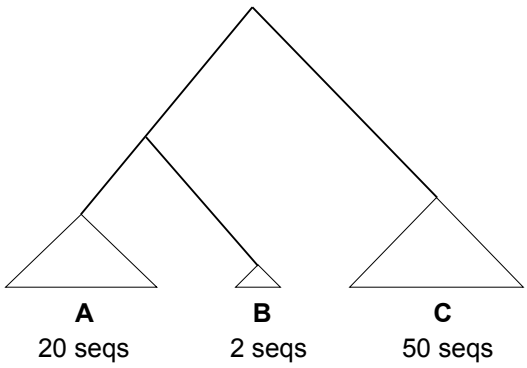
3.1. Prior approaches

Covariance weights

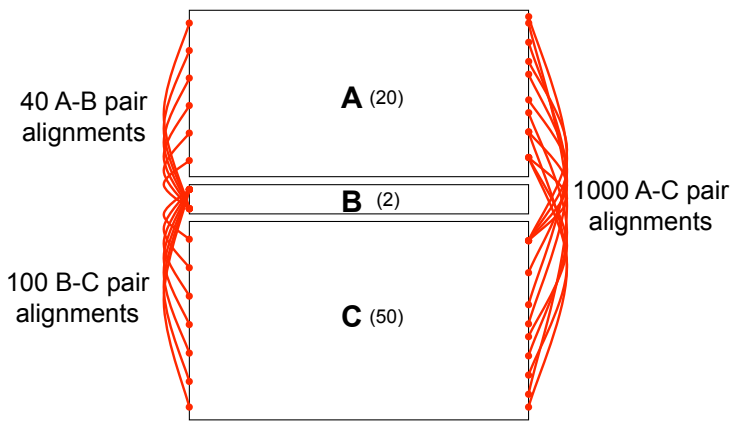
Perhaps the best-known weights for sum-of-pairs multiple alignment are those of Altschul et al. [3]. Of the two weighting schemes they suggest, their second scheme, which they call rationale 2 weights, has the more rigorous basis. We call this scheme *covariance weights*.

Let $\ell(p)$ be the length of path p between two leaves in a tree, and let $\ell(pq)$ be the length of the overlapping portion of two paths p and q. Altschul et al. argue that a correlation coefficient between two pairs of sequences should be defined as

$$\rho_{pq} \;=\; \frac{(\ell(pq))^2}{\ell(p)\ell(q)}. \tag{3.1}$$



(a)



(b)

Figure 3.1: Overrepresented groups can dominate sum-of-pairs scoring function. a) Sequences related by a tree, with two overrepresented groups. b) Sum-of-pairs scoring will drive towards small improvements in alignments between sequences in groups A and C, at the expense of possibly large reduction in quality in alignments between sequences in group B and all others.

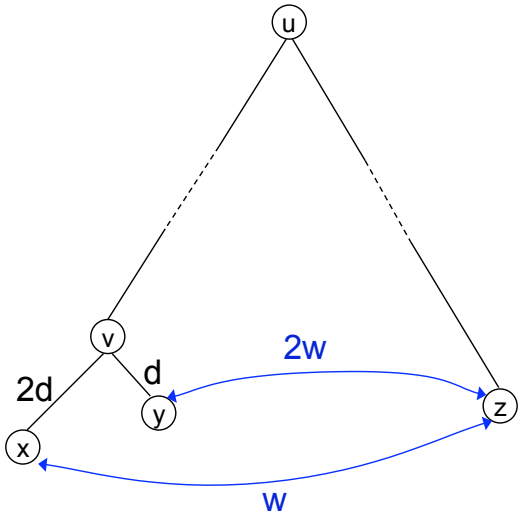


Figure 3.2: The covariance weight method gives the anomalous result that the ratio of weights $w_{xz} : w_{yz}$ depends entirely on the ratio of edge lengths $\ell_{vx} : \ell_{vy}$, even when lengths ℓ_{vx} and ℓ_{vy} are very small compared to lengths $\ell_{uv} = \ell_{uz}$ See text for validation.

For k sequences, if these $O(k^4)$ covariance are stored in a matrix M, weights for the pairs can then be computed by taking row sums from M^{-1}. This would take $O(k^4)$ time just to create such a matrix, but Altschul described an involved algorithm [4] that avoids matrix inversion, and can compute weights in $O(k^2)$ time.

Covariance weights have the nice property of being independent of root placement, but can exhibit counterintuitive behavior. For example, consider the tree in Figure 3.2. Suppose edge lengths ℓ_{vx} and ℓ_{vy} are very small compared to ℓ_{uv} and ℓ_{uz}. Under this condition, nodes x and y are essentially identical from the perspective of z, so it is reasonable that w_{xz} and w_{yz} should be roughly equal. But under covariance weights, if $\ell_{vx} = c\,\ell_{vy}$, then $w_{yz} \approx c\,w_{xz}$ even for arbitrarily long $\ell_{uv} + \ell_{uz}$. An example is provided here: let $\ell_{uv} = 998$, $\ell_{uz} = 1000$, $\ell_{vx} = 2$, and $\ell_{vy} = 1$. Then

$$M = \begin{bmatrix} 1 & \frac{2^2}{3 \cdot 2000} & \frac{1^2}{3 \cdot 1999} \\ \frac{2^2}{2000 \cdot 3} & 1 & \frac{1998^2}{1999 \cdot 2000} \\ \frac{1^2}{1999 \cdot 3} & \frac{1998^2}{2000 \cdot 1999} & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0.00067 & 0.00017 \\ 0.00067 & 1 & 0.99850 \\ 0.00017 & 0.99850 & 1 \end{bmatrix},$$

$$M^{-1} = \begin{bmatrix} 1.00008 & -0.16689 & 0.16647 \\ -0.16689 & 333.61137 & -333.11093 \\ 0.16647 & -333.11093 & 333.61123 \end{bmatrix}.$$

And the resulting pair weights are:

$$w_{xy} = 0.99967 \ ,$$

$$w_{xz} = 0.33356 \ ,$$

$$w_{yz} = 0.66678 \ .$$

Covariance weights are not used in practice for sequence alignment, but an approximate method for computing them, called *3-way weights* is. See below for details.

3-way weights

Aligning two length-n alignments of k sequences takes time $O(n^2k^2)$ when the substitution costs for aligning a column of one alignment with a column of another alignment are computed naively. Gotoh described significant speedups using what he called *generalized profile* operations [46, 103], which give an $\Omega(k)$ reduction in run time, to $O(n^2k)$. This speedup is employed by the fastest alignment tools, `Muscle` [29] and `MAFFT` [59], and in `Opal` [115]. However, these operations can be shown [47] to work for the weighted sum-of-pairs scoring function only if each pairwise weight w_{xy} for sequences x and y is of the form

$$w_{xy} \ = \ w_x \cdot w_y \tag{3.2}$$

(i.e. pair weights are the product of constant per-sequence weights).

Covariance weights do not obey this property, but Gotoh described a method [47] of approximating covariance weights that does. This method is often called *3-way weights*. Weights for each *edge* are computed based on a decomposition of the tree into overlapping three-way trees. When aligning profiles formed from two

subtrees (as is done in both progressive alignment [36] or tree-based polishing [52]) a weight w_x can be efficiently computed for each sequence x as a product of the weights of the edges from that sequence to the the root of its subtree, satisfying equation 3.2.

The edge weights are computed as follows. For a three-way (unrooted) tree with edges A, B, and C, with respective edge lengths a, b, and c, edge weights are assigned values

$$w_A = \sqrt{\frac{bc(a+b)(a+c)}{a(b+c)FS}}$$

$$w_B = \sqrt{\frac{ac(b+a)(b+c)}{b(a+c)FS}}$$

$$w_C = \sqrt{\frac{ab(c+a)(c+b)}{c(a+b)FS}}$$

where $S := (ab + bc + ac)$, and F is a hand-tunable parameter. In the full tree, a leaf edge takes its weight directly from the single 3-way tree that contains the edge. The weight of an internal edge is computed as the product of the weights for that edge found in the two 3-way subtrees containing that edge.

This method depends on a hand-tuned parameter, and provides no guarantee that it approximates covariance weights, though it found similar pairwise weights in experiments on a very limited range of topologies [47]. To the extent that the 3-way method does approximate covariance weights, it suffers from the same anomalous behavior as covariance weights do. 3-way weights are used in the polishing stages of `MAFFT` and `Muscle`.

Division weights

The program `ClustalW` introduced a simple weighting scheme that has been incorporated into other alignment software as well, such as in the construction stages of `MAFFT` and `Muscle`. We call `ClustalW`'s scheme, *division weights*.

In this scheme, each leaf i of a rooted tree T is first assigned a weight w_i as follows. The length ℓ_v of an edge from child v to parent u is equally divided among the k_v leaves in the subtree rooted at v, and the portion ℓ_v/k_v that leaf i gets is accumulated in w_i. Letting p_i be the path from leaf i to the root,

$$w_i \;=\; \sum_{v \in p_i} \ell_v / k_v. \tag{3.3}$$

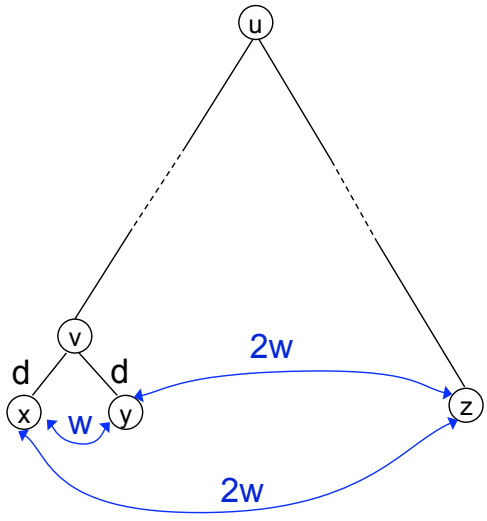


Figure 3.3: The division weight method gives the anomalous result that both w_{xz} and w_{yz} converge to a value twice as large as w_{xy} when lengths ℓ_{vx} and ℓ_{vy} are very small compared to lengths $\ell_{uv} = \ell_{uz}$. See text for derivation.

The weight for pair i, j is defined to be $w_{ij} := w_i\, w_j$. It is easy to see that the w_{ij} for k sequences can be computed in time $O(k^2)$, and that the weights satisfy the requirements for equation 3.2.

Division weights are dependent on the placement of the root, which may be problematic, as correct root placement is often not known for a set of sequences to be aligned, and the tree construction methods used by alignment tools in the guide tree construction stage (Section 2.4) compute unrooted trees. Division weights can also exhibit counterintuitive behavior. Consider the tree in Figure 3.3. Suppose lengths ℓ_{vx} and ℓ_{vy} are very small compared to lengths $\ell_{uv} \approx \ell_{uz}$. The similarity of x and y is high, so pair weight w_{xy} should be much higher than pair weights w_{xz} and w_{yz}, which correspond to unreliable alignments. But under division weights, leaf weight w_z will be roughly twice w_x and w_y since ℓ_{uv} is split between w_x and w_y while ℓ_{uz} is given entirely to w_z. The result is that $w_{xz} = w_{yz} \approx 2w_{xy}$. By using division weights only in the construction stage (not in polishing), `Muscle` and `MAFFT` avoid the trouble this causes, as the weights between groups only come into play after the weights within groups have already played their role in the alignment of those groups. This sort of error would wreak havoc with alignments if, for example, the edge between x and v were cut in a tree polishing step (see Section 2.7). Note that `ClustalW` does not perform polishing.

For another anomaly, see Figure 3.4. When $\ell_{vx} > \ell_{vy}$, weight w_{yz} should be

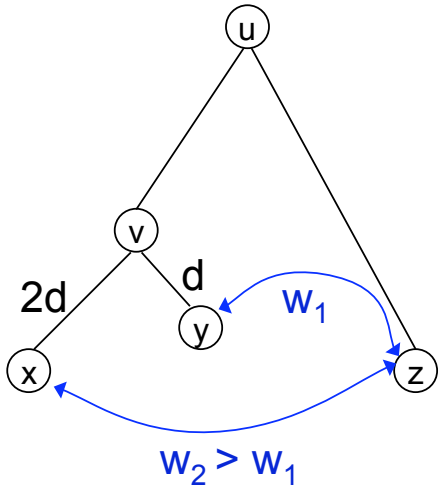


Figure 3.4: The division weight method gives more weight to more divergent sequences. When sibling edges vx and vy have different lengths, the alignment of the less divergent pair of sequences, z and y is more reliable, but receives less weight under this scheme.

---

greater than w_{xz}, since sequence y is more similar to z, and the alignment of y to z is therefore more reliable than the alignment of x to z. But division weight method sets $w_{xz} > w_{yz}$.

3.2. Influence weights

Here we describe a new weighting method that is easy to compute and overcomes the anomalous behavior observed for the other methods.

One way to assign a weight w_{ij} to a pair i, j of leaves is to quantify the *influence* of one leaf on another on the basis of the shape of T and its edge lengths. Suppose we have a measure $\omega(i, j)$ of the influence of leaf j on leaf i in T where function ω is not necessarily symmetric. Then we can define a symmetric weight on a pair of leaves by the geometric mean of their influences:

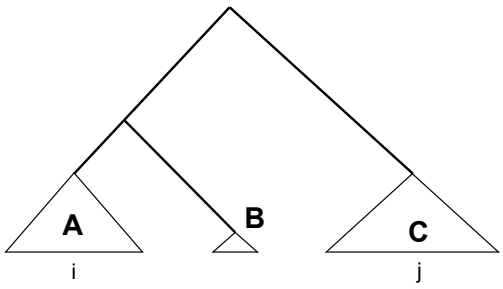
$$w_{ij} := \sqrt{\omega(i, j)\, \omega(j, i)}. \tag{3.4}$$

We call the w_{ij} obtained in this way *influence weights*.

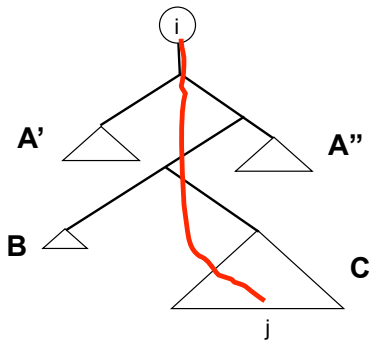
Our influence function ω is nonnegative, and for every leaf i it satisfies

$$\sum_{j\,:\,j \neq i} \omega(i, j) = 1. \tag{3.5}$$

As a consequence, the resulting weights satisfy $0 \leq w_{ij} \leq 1$.



(a)



(b)

Figure 3.5: Calculating influence weights. (a) Sequences i and j in the context of a rooted tree. (b) The tree is re-rooted at i, and the influence of j on i is computed.

Influence definition

To determine the influence $\omega(i, j)$ of leaf j on leaf i, we carry out the following recursive process. Imagine making i the new root of T, and letting T hang from root i. We denote this *rerooted tree* with root i by T_i. Starting from root i, we process T_i top-down, splitting the total mass of weight 1 from equation 3.5 among the descendants of i. The new root i has exactly one child (which was originally the parent of i in T), and this child receives the entire mass of weight 1 passed down from its parent i. In general, if a descendant x receives mass w_x from its parent, this mass is split among its two children y and z (possibly unequally) so that

$$w_x \;=\; w_y + w_z. \tag{3.6}$$

After completing this top-down splitting process, the amount of the original mass 1 at the root i that ends up at leaf j is taken as the influence of leaf j on root i, (see figure 3.5b):

$$\omega(i, j) \;:=\; w_j. \tag{3.7}$$

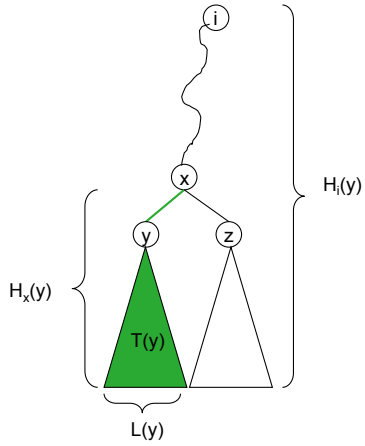


Figure 3.6: Graphical description of values used in influence calculation

---

The key is determining how to split the mass at a parent among its two children on the basis of the edge lengths of T. We do such a split as follows.

For nodes v and w of T, denote the path in T between v and w by P_{vw}. For a node y, let $T(y)$ be the subtree of T rooted at node y, and let $T_x(y)$ be $T(y)$ together with the path P_{xy}. Denote the length of path P_{xy} by ℓ_{xy} and the *set of leaves* in $T(y)$ by $L(y)$. The total *size* of $T_x(y)$ is

$$S_x(y) \;:=\; \ell_{xy} \;+\; \sum_{e \,\in\, T(v)} \ell_e \;. \tag{3.8}$$

The average *height* (see Figure 3.6) of $T_x(y)$ is

$$H_x(y) \;:=\; \ell_{xy} \;+\; \frac{1}{\big|L(y)\big|} \sum_{i \,\in\, L(v)} \ell_{iy} \;. \tag{3.9}$$

We call the effective *number of leaves* in $T_x(y)$,

$$N_x(y) \;:=\; \begin{cases} \dfrac{S_x(y)}{H_x(y)}, & H_x(y) \neq 0; \\[2ex] 1, & \text{otherwise.} \end{cases} \tag{3.10}$$

The effective number of leaves satisfies $1 \leq N_x(y) \leq \big|L(v)\big|$. See Figure 3.7 for extremal examples of the effective number of leaves.

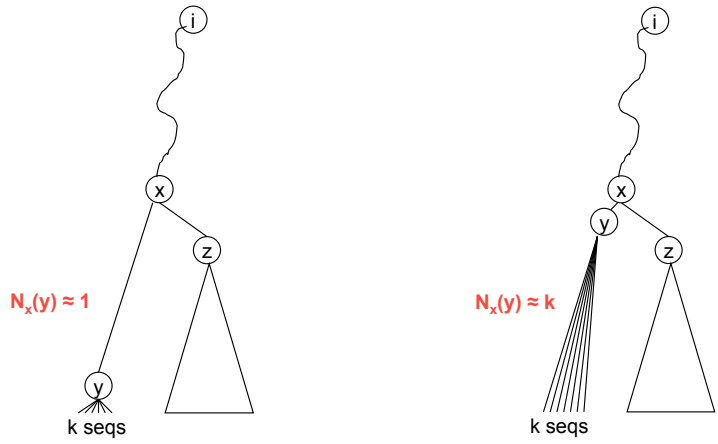


Figure 3.7: Effective number of sequences in a subtree depends on the apparent independence of sequences from the perspective of the root of the subtree. a) When k sequences appear to be nearly identical, $N_x(z) \approx 1$. b) When k sequences appear to be independent, $N_x(z) \approx k$.

In tree T_i, we split weight $w_x = w_y + w_z$ between the two children y and z of x according to the ratio,

$$\frac{w_y}{w_z} = \frac{N_x(y)}{N_x(z)} \frac{H_i(z)}{H_i(y)} \tag{3.11}$$

(where when $H_i(y) = 0$, we assign node y all the weight). For example, with children that have identical average heights and effective numbers of leaves $N_x(y) = 1$ and $N_x(z) = 3$, child y gets $1/4$ of w_x and child z gets $3/4$. See Figure 3.8 for further examples.

Splitting the weight $w_i = 1$ top-down over T_i according to these ratios fully specifies the leaf weights w_j, and hence the influence function $\omega(i, j)$ and the weights w_{ij}.

For k sequences, influence weights w_{ij} can be computed in time $O(k^2)$, which is optimal. They produce reasonable weights in the cases that division and covariance weights produce anomalous results. For example, in the covariance anomaly of Figure 3.2, influence weights converge to $w_{yz} \approx w_{xz}$ as ℓ_{vz} grows. In the division weight anomaly of Figure 3.3, influence weights converge to $w_{xz} = w_{yz} \approx w_{xy}/\sqrt{2\ell_{vz}}$ as ℓ_{vz} grows, which matches the expectation that the alignment of similar sequences x and y should be more reliable than the other alignments. And in the division weight anomaly of Figure 3.4, influence weights appropriately assign greater weight

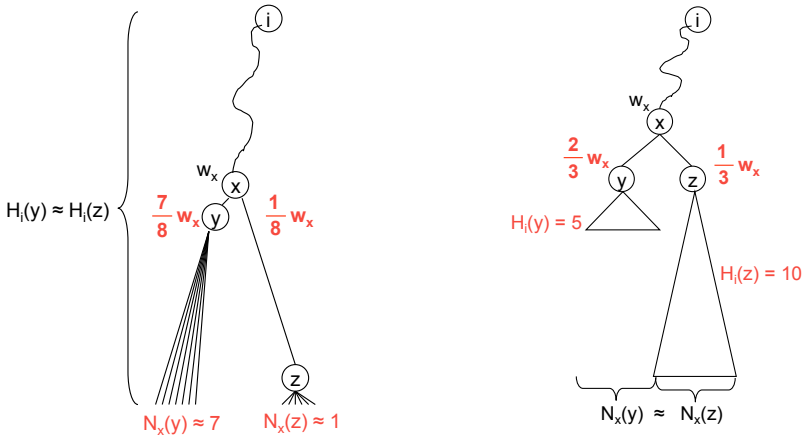


Figure 3.8: Splitting w_x, the the portion of i's weight that has been distributed to x, between the subtrees of x. a) When heights are equal, the subtree with more effective sequences gets a larger portion of the weight. b) When effective sequence counts are equal, the subtree with lower average height to the root gets a larger portion of the weight

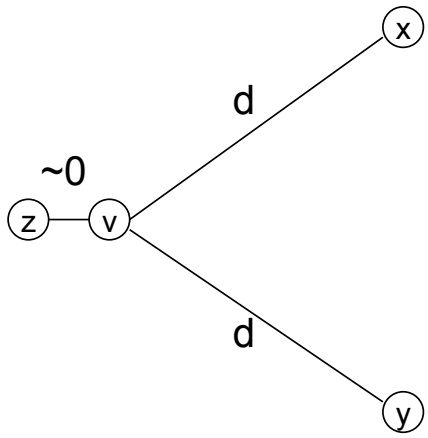


Figure 3.9: Altschul [3] suggests that the weight w_{xy} should be 0 as length ℓ_{yz} goes to 0. This may not be appropriate, as it will fail to sensibly align regions resulting from convergent evolution.

to a pair of sequences with higher similarity.

Influence weights disagree with both division and covariance weights in another situation. Consider the unrooted tree in Figure 3.9 . Altschul suggests [3] that, as ℓ_{vz} approaches 0, w_{xy} should go to 0. At first glance this seems reasonable, since in this extreme case sequence z is effectively the ancestral sequence relating x and y, so those two sequences should align to each other *through* z. The covariance weighting scheme achieves this goal, as do division weights (so long as the root is placed approximately at z).

After further consideration, however, it appears that a non-zero w_{xy} weight is more appropriate. Consider the case where the sequences are:

$$x := \text{ACTG} \ ,$$
$$y := \text{ACTG} \ ,$$
$$z := \text{ACG} \ .$$

In other words, there were independent insertions of a T between the C and G of the ancestral sequence. This is a simple form of convergent mutation, and though the Ts are not homologous in terms of being replicated from the same ancestral character, they are at the very least similar, and may be a sort of latent homology (e.g. sequence x was primed for insertion of the T). If the w_{xy} is zero, then the weighted score of the multiple alignment of all three sequences won't be affected by the way those Ts are aligned (in the same column or two separate columns), so no alignment will be preferred. In contrast, if w_{xy} is non-zero, then the weighted score will favor an alignment placing the insertions in the same column.

Influence weights assign a non-negligible value is assigned to w_{xy}. This is due to the $\frac{H_i(z)}{H_i(y)}$ component of the recursive distribution of influences. The concrete values are:

$$\omega(z, x) = 0.5 \ ,$$
$$\omega(z, y) = 0.5 \ ,$$
$$\omega(x, y) = 0.33 \ ,$$
$$\omega(x, z) = 0.67 \ ,$$
$$\omega(y, x) = 0.33 \ ,$$
$$\omega(y, z) = 0.67 \ ,$$

so

$$w_{xz} = \sqrt{0.5 \cdot 0.67} = 0.58 \ ,$$
$$w_{yz} = \sqrt{0.5 \cdot 0.67} = 0.58 \ ,$$
$$w_{xy} = \sqrt{0.5 \cdot 0.33} = 0.41 \ .$$

These weights seem reasonable, as they assign most of the weight to good alignments of x through z to y, but will still reasonably resolve instances of convergent evolution.

Efficiently computing weights

Pairwise weights can be computed by hanging the tree by each leaf i in order, and for each such rehanging, computing the influence of each other leaf j on i by distributing i weight across the tree as described in the previous section. If the average height, $H_x(y)$, and effective number of leaves, $N_x(y)$, are pre-computed then distribution of weights for each root will take $O(k)$ time for k sequences, resulting in an overall $\Theta(k^2)$ computation.

In an unrooted binary tree T, every internal node has three relatives, which map in some way to the parent and two children in a rooted tree, with the mapping dependent on how the tree is rooted. We aim to store for each internal node x the $H_x(y)$ and $S_x(y)$ values for each relative y as thought x were the root of T and those relatives were all its children. Then, for any arbitrary hanging of T, we will have the $H_x(y)$ and $S_x(y)$ values for the two children y of x induced by that hanging.

These values can be computed in an $O(k)$ preprocessing step, in which we begin by hanging T by an arbitrary leaf. A simple bottom-up pass over T allows computation of the $H_x(y)$ and $S_z(y)$ values for the two children y of every node x induced by that hanging, but does not assign the values for the third relative (currently the parent). The $S_z(y)$ values of the parent for a fixed node is simply the total branch-length of the tree minus the sizes of the two other relatives. $H_z(y)$ values can be gathered via a second top-down sweep over T, in which weighted heights to the remainder of the tree above that node are collected based on values computed in the first (bottom-up) pass.

It appears to be not possible to apply these weights to Gotoh's generalized profile operations, since the weights are not of the form $w_{ij} = \omega_i \cdot \omega_j$, and there is no clear way to make them behave in a way analogous to the method of 3-way weights for profile operations [47]. This is because of the $H_i(x)$ factors in formula 3.11. This suggests an avenue of future work on the problem.

3.3. Estimation of edge lengths

All the methods described in this chapter for computing pair weights depend on a tree with known edge lengths, a requirement that is largely ignored in papers describing the application of weights to sequence alignment. The neighbor-joining method, used for guide tree construction (Section 2.4) in `ClustalW`, generates a tree with accompanying edge lengths, but the guide tree methods that have been shown to produce better alignment results, `UPGMA` and `MST` [30, 59, 115], generate a topology only, with no edge lengths. When guide trees are so constructed, a pair weight method requires that edge lengths be computed.

In the tests described in Section 2.6, we estimated edge lengths by minimizing the L_1-norm between the observed pairwise distances and the path lengths induced by the tree's edge lengths. The L_1-norm is a measure of distance between vectors that is often called *manhattan distance*, specifically the sum of differences in vector values. In this case it is the sum over all leaf-pair paths of the differences between observed and tree-induced path lengths. Formally, given a tree topology $\mathcal{T}$ and pairwise distances d_{ij}, let $\ell(e) \geq 0$ be the estimated length of edge e, and let ℓ_{ij} be the length of the path p_{ij} from leaf i to leaf j: $\ell_{ij} = \sum_{e \in p_{ij}} \ell(e)$. The L_1-norm can be found using a simple linear program, with the objective function minimizing $\sum_{i,j} |d_{ij} - \ell_{ij}|$.

The L_1-norm is not, however, the only method for estimating edge lengths. For example, the Minimum Evolution method of phylogeny inference is based on least-squares inference of edge lengths, which is essentially the L_2-norm (often called *Euclidean distance*, the square root of the sum of squared differences). Under least-squares, edge lengths are found that minimize $\sum_{i,j}(d_{ij} - \ell_{ij})^2$. This method for fitting a linear function to data is known as ordinary least squares (OLS); its application to edge length estimation was suggested by Cavalli-Sforza and Edwards [19], and followed by Altschul [3] and Gotoh [47]. It works under the implicit assumption that errors in observed values are uncorrelated and have equal variance. OLS is known to be a maximum-likelihood estimator of a function when these conditions hold and errors are normally distributed. Furthermore, the Gauss-Markov theorem indicates that OLS has the minimum variance among all linear estimators when errors have expected value of zero, regardless of the distribution [1].

Since variance of the distance estimate for a pair of sequences is known to rise as sequence distance rises [39], the assumption of equal variance among pairwise distances is unfounded; lengths can be found under weighted least squares (WLS), which accounts for the per-pair variances. Also, pairwise distances are, of course, correlated since the sequences are related by a tree; generalized least squares (GLS) accounts for this by incorporating a covariance matrix. Bryant and Waddell [14] described algorithms for computation of edge lengths under each of these criteria, OLS, WLS, and GLS with respective run times of $O(k^2)$, $O(k^3)$, and $O(k^4)$ for k sequences. Variances and covariances for pairwise distances under the Jukes Cantor model of sequence evolution [58] can be estimated using the method of Bulmer [16, 39].

3.4. Discussion

We have presented a new method for computing non-uniform weights for sequences related by a tree. It is easy to implement, fast to evaluate, and avoids the anomalies of current approaches. Surprisingly, and in contrast to what has generally been suggested in the literature [47, 29], *unweighted* sum-of-pairs performs as well as all weighting schemes that we tested in our experiments on using weighted sum-of-

pairs scores for multiple sequence alignment (see Section 2.6). Even for the small number of inputs from `BAliBASE` that do present some level of overrepresentation, the accuracy gains from using these weights are limited at best.

An important point, which to our knowledge has not been mentioned in prior studies of multiple alignment methods, is that the standard measure of recovery is itself *inherently biased*. The SPS recovery score is measured uniformly over all induced pairwise alignments. So if a weighting method alters an alignment by correcting for over-represented groups, it is entirely possible that the corrected alignment worsens between a few highly over-represented groups while improving between several under-represented groups, and in the end has decreased percent recovery under SPS. Most other measures of alignment accuracy also sum a measure of pairwise accuracy over all pairs, and thus suffer the bias. (The TC score measures accuracy as the percent of reference columns that are exactly recovered by the computed alignment; thus, it doesn't suffer the same problem, but is highly sensitive to small errors, and thus seems not applicable to alignments of a large number of sequences.) While weighting methods attempt to correct for precisely this bias, it is far from clear how to measure recovery more objectively, since measuring recovery using a weighting method's own weights is not objective either.

It is possible that the choice of L_1-norm edge lengths had a negative impact on the quality of pair weights, leading to the observation (Section 2.6) that pair weights did not impact alignment quality. This is worth further consideration, since other work ([29, 59]) suggests a slight positive impact from weights, though they do not describe how edge lengths were computed for their trees.

Finally, we note that this weighting scheme is general, and may be applied to other fields in which entities are related by a tree. For example, they could be used in estimating an average character trait value for a family of organisms, or in improving homology search against sequence families, or in other tree-based measures such as a "balanced" average subtree distance with application to the least-squares framework for edge length estimation [24]) (see Section 8.2 for more details of this idea).

CHAPTER 4

IMPROVING MULTIPLE ALIGNMENT THROUGH PAIRWISE ALIGNMENT CONSISTENCY

The standard heuristic for building multiple sequence alignments, called *progressive alignment* [36] and described in detail in Chapter 2, merges sequences into increasingly large alignments in an order determined by a binary tree relating the input sequences. Each leaf of the tree stores a single sequence from the input (this sequence can be thought of as a degenerate alignment), and each internal node stores an alignment of the sequences belonging to the leaves in the subtree under that node. The alignment at an internal node is generated by merging (aligning) the alignments of the node's two children. When merging alignments, columns of one alignment are aligned with columns of the other; the columns of each child alignment remain unaltered in the resulting alignment. In this process, an alignment at an internal node is formed based only on information contained in the sequences being aligned, without regard for how well the resulting alignment will align with other sequences in the final alignment formed at the root of the tree.

The progressive alignment heuristic can be contrasted with computing an optimal alignment of multiple sequences, e.g. [71]. By definition, optimal alignment of multiple sequences under the standard sum-of-pairs scoring function (which is NP-hard [114]) induces pairwise alignments for each pair of sequences A and B that are optimal with respect to how the alignment ($A \sim B$) constrains alignments of A and B with all other sequences C - no other alignment of A and B could improve the sum of pairwise scores over all pairs in the alignment. In the standard progressive strategy, an optimal alignment of sequences in a subtree is computed based on local information contained in those sequences, and may prove to be suboptimal in the larger context of the full alignment - resulting in a final alignment in which some (or all) induced pairwise alignments ($A \sim B$) could be realigned in a way that would improve the overall alignment score.

As mentioned in Chapter 2, one approach to resolving this suboptimality is to modify the alignment after the fact, in a method known as polishing (see Section 2.7). Another approach is to avoid making the errors in the first place using a technique based on consistency of alignments.

The idea of using alignment consistency is to bridge the divide between the progressive and exact approaches. The core idea is to make alignment decisions at each internal node of the progressive alignment guide tree based on global information, specifically information gleaned from other sequences in the input. Practically, this means modifying the function used to score an alignment generated

for the sequences belonging to a subtree in a way that reflects the support given by sequences outside that subtree for particular alignment decisions. Use of consistency has been shown to improve recovery of reference alignments by several percent on average [83, 25].

All the consistency-based methods described below share the common strategy of modifying scores for aligning pairs of sequences, then aligning alignments so as to optimize the sum of (modified) pairwise alignment scores. For this reason, description of methods below will focus on how scores are modified for alignments of a pair of sequences A and B.

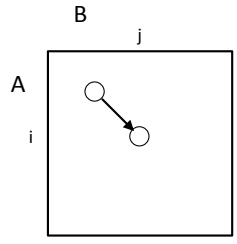
Under standard scoring conventions, the score for aligning two sequences is comprised of substitution scores and gap penalties. Substitution scores are typically derived from a matrix (e.g. BLOSUM [51]) that gives a position-independent score for substituting one letter with another, while affine gap penalties (see Section 2.9) are composed of a per-gap cost, γ, and a length-dependent cost, $\ell\lambda$ for a gap of length ℓ. The gap costs are position independent, and chosen so that the tool performs well on benchmarks.

In the consistency framework, position-dependent scores are computed. A substitution of the i^{th} position of A with the j^{th} position of B can be represented as $(a_i \sim b_j)$. The position-dependent score $D(a_i, b_j)$ of $(a_i \sim b_j)$ depends on the support given by other sequences for that particular substitution. Gap penalties are not used in prior implementations of this framework, so that the score of an optimal pairwise alignment of prefixes $(a_1 \ldots a_i)$ of A and $(b_1 \ldots b_j)$ of B can be computed by the recurrence in equation 1.2, but with $D(a_i, b_j)$ replacing $\sigma(a_i, b_i)$, and $\lambda = 0$.

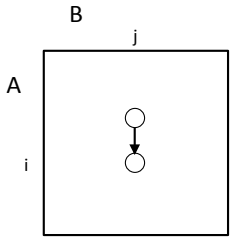
This recurrence can be generalized to aligning two alignments, $\mathcal{A}$ and $\mathcal{B}$, by replacing the $D(a_i, b_j)$ score with a function that sums over the scores of all pairs of characters between the two columns i and j. The two-sequence variant finds a maximum weight trace [61] on two sequences, and the two-alignment variant finds a constrained maximum weight trace of all the sequences in the two sets, in which the columns of the two merged alignments are required to remain intact.

As described in Section 1.2, an alignment of two sequences can be viewed as a path in a 2-dimensional dynamic programming table. It is useful to consider features of such paths, as they will play a role in devising a more complete model of alignment consistency. A feature can be thought of as a subpath in the graph. A diagonal edge in a path (as in Figure 4.1a) corresponds to a substitution feature. A vertical or horizontal edge in a path (as in Figures 4.1b and c) corresponds to a gap extension feature, while the subpaths in Figures 4.1d through k correspond to gap-open and gap-close features.

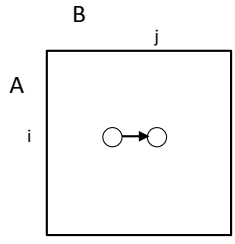
Note that in prior alignment consistency methods, a substitution score (the diagonal edge in Figure 4.1a) is incorporated, but all other features (gap-related



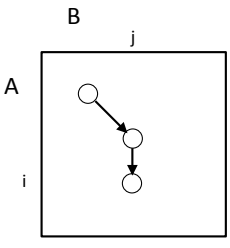
(a) substitution



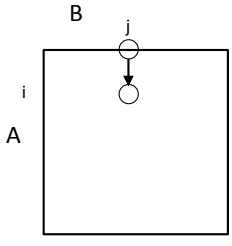
(b) vertical gap extension



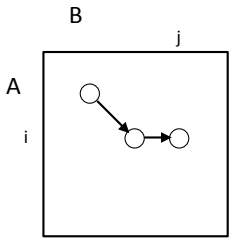
(c) horizontal gap extension



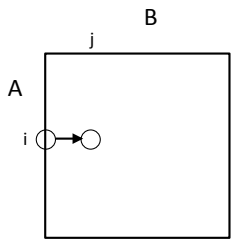
(d) vertical gap open, general case



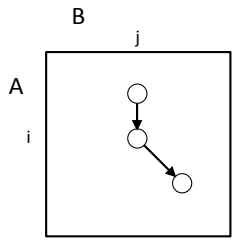
(e) vertical gap open, special case



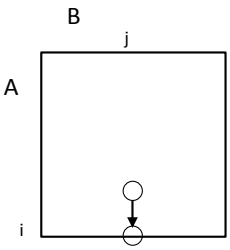
(f) horizontal gap open, general case



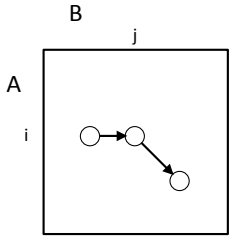
(g) horizontal gap open, special case



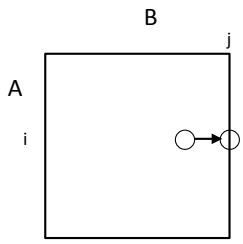
(h) vertical gap close, general case



(i) vertical gap close, special case



(j) horizontal gap close, general case



(k) horizontal gap close, special case

Figure 4.1: Features in dynamic programming table for alignment ($A \sim B$). (a) Substitution of a_i with b_j. (b) Placing a_i somewhere in a gap between b_j and b_{j+1}. (c) Placing b_j somewhere in a gap between a_i and a_{i+1}. (d and e) The general (d) and special (e) cases of opening a gap with a_i immediately following b_j. (f and g) The general (f) and special (g) cases of opening a gap with b_j immediately following a_i. (h and i) The general (h) and special (i) cases of a_i ending a gap that immediately follows b_j. (j and k) The general (j) and special (k) cases of b_j ending a gap that immediately follows a_i.

costs, Figures 4.1 b through k) have zero cost. The argument supporting this gap-cost-free approach is that gap costs are considered in computing the modified substitution scores, so they need not be considered in the final alignment procedure based on those modified scores. However, it is possible that conflicting signals from a set of different sequences could lead to an alignment ($A \sim B$) that opens gaps in a way that is universally poor, for example opening many small gaps rather than a few large gaps. Such gap patterns are restricted in standard alignment methods by use of the common affine gap cost function, in which each run of gap characters incurs a per-gap penalty (γ, which corresponds to the open and close costs pictured in Figures 4.1d through k) and a per-unit-length penalty (λ, Figures 4.1b and c). For this reason, it seems worthwhile to consider a consistency-based score modification method that includes modified gap costs. Our new method does this.

4.1. Prior approaches

The idea of using consistency to improve alignments has a long history. Gotoh [44] described a method that identifies regions in optimal pairwise alignments that are *consistent* with each other, and uses them as anchor points in an algorithm for exact alignment of multiple sequences. It is noteworthy that the method does not simply consider a single optimal alignment for each pair of sequences, but rather the set of alignments for each pair that have optimal score. Unfortunately, as the number of sequences grows, it becomes increasingly unlikely that all sequence pairs will agree with any anchor point, severely limiting the scale of inputs the method can handle.

The methods described below employ consistency in a progressive alignment framework. The primary difference between these methods is how each measures support for an aligned pair.

`T-Coffee` Notredame et al. merged the ideas of consistency and progressive alignment in their landmark work on tools `Coffee` [84] and `T-Coffee` [83]. This amounts to a redefining of the term consistency as applied to alignment: rather than using consistent segments of pairwise alignments as mechanisms for speeding exact alignment, these tools use consistency of pairwise alignments to improve the quality of alignments built with the progressive framework. They take as input a set of sequences and a set of pairwise alignments of those sequences, and define the quality of a multiple alignment of those sequences as the extent to which that multiple alignment is *consistent* with the set of pairwise alignments. There is no requirement that all pairwise alignments be consistent with any part of the resulting multiple alignment, just that the multiple alignment try to agree as much as possible with those pairwise alignments. Where `T-Coffee` and other consistency methods differ is in their definitions of such agreement.

The approach of `T-Coffee` is to compute substitution scores for all pairs of positions in all pairs of input sequences, then perform alignments based on

those position-pair substitution scores. The position-pair scores of `T-Coffee` are calculated as follows. For each pair of sequences (A, B), an optimal global alignment is calculated using `ClustalW` [110], and the 10 top-scoring non-overlapping local alignments from `Lalign` [55] are gathered. Every aligned position pair $(a_i \sim b_j)$ found in one of these alignments contributes a value to the position-pair score $S(a_i, b_j)$, where the value contributed is the percent sequence identity of the alignment containing that pair. The motivation behind using percent identity as a weight is that more similar sequences should contribute more information to the final alignment. Scores are contributed additively, so that if a pair is found in more than one pairwise alignment (i.e. both a global and local alignment), it will have a score that is the sum of the percent identities of the containing alignments. If a pair is not found in any alignment, then its score is 0. This describes how to calculate an initial score for each pair. The set of initial scores is called the *primary library*.

The notion of consistency is employed in developing what is termed the *extended library*. The approach is to quantify the support from other sequences C for aligning a position pair $(a_i \sim b_j)$ by tallying the positions c_h that are found to align to both a_i and b_j in the input alignments. These can be thought of positions such that a_i aligns *through* c_h to b_j, $(a_i \sim c_h \sim b_j)$. Specifically, the consistency-based position-pair score is

$$D(a_i, b_j) \;=\; S(a_i, b_j) + \sum_{C} \min \{S(a_i, c_h), S(b_j, c_h)\}. \tag{4.1}$$

For k sequences of length n, `T-Coffee` requires time $O(k^2n^2)$ and space $O(k^2n)$ to construct and store the primary library, and worst case time $O(k^3n^2)$ to complete the extended library (though in practice runtime is reported to be roughly $O(k^3n)$ because of the sparseness of the score matrices in the primary library). Results given in the `T-Coffee` paper (and reiterated in Sections 2.10 and 4.6) show that `T-Coffee` gives a several percent average increase in recovery over `ClustalW`, affirming the benefits of incorporating local alignment and triple-based consistency.

As mentioned earlier, one concern about this approach is that the alignment method depends only on substitution scores, with no accounting for gaps. Another significant drawback of this approach is that only a single optimal global alignment for each pair is used to establish pair scores. This is despite the fact that when sequence C is eventually merged with the alignment $(A \sim B)$, it will likely do so with an alignment that disagrees with the both optimal alignments $(A \sim C)$ and $(B \sim C)$. It seems preferable to allow C to contribute to the modified $(A \sim B)$ scores by expressing the extent of suboptimality required of alignments $(A \sim C)$ and $(B \sim C)$ in order to support the alignment of position-pair $(a_i \sim b_j)$.

`MAFFT` Katoh et al. developed a method [59] that allows `MAFFT` to compute its analog to the extended library in time quadratic in the number of sequence,

avoiding the cubic overhead of `T-Coffee`. Rather than explicitly measure support for aligning positions ($a_i \sim b_j$) based on triples ($a_i \sim c_h \sim b_j$), `MAFFT`'s approach measures support based on how often a_i and b_j are part of *some* optimal alignment. The *importance score* of a pair of positions is computed such that high scores are assigned to pairs in which both positions are frequently involved in (possibly independent) high-scoring gap-free segments in pairwise alignment with other sequences. Position-pair scores are calculated based on a linear mixture of BLOSUM-derived substitution scores and these importance scores.

This approach avoids explicitly considering triples of strings, and results in a runtime of $O(k^2n^2)$ for calculation of consistency-modified position-pair scores. While this method gives a significant speedup in practice, the role of consistency has been changed to an indirect one: rather than finding an alignment that will be consistent with respect to other sequences, it finds an alignment that aligns positions that are consistently (read: frequently) paired off in other alignments. Even with this indirect approach, results given in the paper show a several percent improvement in recovery, across a variety of benchmarks, over not using this consistency approach (see Section 4.6).

Aside from the indirect method of computing consistency modified scores `MAFFT`'s approach is essentially the same as that of `T-Coffee`. It bases its consistency-modified scores on global alignments computed with its FFT alignment algorithm, and optionally includes local alignments from the tool `fasta34`. Like `T-Coffee`, `MAFFT` does not account for sub-optimal global alignments in assigning scores, and does not incorporate gap costs in the final alignment cost scheme.

`ProbCons` Do et al. introduced an idea they call *probabilistic consistency*, in a tool named `ProbCons` [25]. Their approach depends on an established method [28] of computing, for a pair of sequences A and B, the probability that each position a_i aligns with each position b_j. Letting z^* be the true alignment, and $\mathcal{Z}$ the set of all possible alignments, their method defines the posterior probability of residue a_i aligning to residue b_j as

$$p(a_i \sim b_j \in z^* | A, B) \;=\; \sum_{z \in \mathcal{Z}} P(z|A,B) \;\; 1(a_i \sim b_j \in z) \tag{4.2}$$

where $\mathbf{1}(expr)$ is the indicator function that returns 1 if *expr* is true, and 0 otherwise. $P(z|A,B)$ represents the probability of emitting alignment z under a pairwise HMM (see [28]), and given sequences A and B.

Two sequences can be aligned in a way that maximizes the sum of these probabilities, in effect finding an alignment of maximum *expected accuracy* [78, 28]. (It should be noted that this may disagree with the most probable alignment under the pair-HMM, found using the so-called Viterbi algorithm, which is equivalent to the standard dynamic programming algorithm.) This pairwise alignment

method can be trivially extended to aligning multiple sequences: the position-pair probabilities are simply another form of position-pair score, so progressive alignment can be applied with sum-of-pairwise scores maximized at each internal node.

The notion of consistency is applied to this framework via a consistency transformation based on the probability, for each other sequence C in input $\mathbb{C}$, of the pairs that induce an aligned triple $(a_i \sim c_h \sim b_j)$. This is computed as

$$p'(a_i \sim b_j) \;=\; \frac{1}{|\mathcal{C}|} \sum_{C \in \mathbb{C}} \sum_{c_h} p(a_i \sim c_h)\; p(b_j \sim c_h). \tag{4.3}$$

These consistency-transformed probabilities replace the initial ones in computing an alignment with maximum expected accuracy.

This approach represents a key advance in sequence alignment. In the other methods described, only a single optimal global alignment is used for each pair of characters, along with a small number of local alignments. These do not allow any input from pairwise alignments that are suboptimal under the scoring function, or for that matter, any alternate alignments with optimal score. In the scheme of `ProbCons`, alignment probabilities for each pair of positions (a_i, b_j) are summed over all ways of aligning A and B. Thus, in the consistency transform, all pairwise alignments, optimal and suboptimal, contribute to the support (score) of pair $(a_i \sim b_j)$, with their level of support dependent precisely on the probability of that alignment under the model.

The time to compute the initial $p(a_i \sim b_j)$ values over all pairs of sequences is $O(k^2n^2)$. The worst-case time to then calculate the transformed probabilities is $O(k^3n^3)$: for each triple of sequences, each pair (a_i, b_j) requires consideration of all positions c_h. But since most values in the probability matrices are typically near zero, the transformation is computed efficiently using sparse matrix multiplication by ignoring all entries smaller than a threshold ω. Thus, the runtime is cubic in k with a smaller practical factor of n. The paper reports a 5% improvement in benchmark recovery over using pair probabilities without consistency, in agreement with results in Section 4.6.

An additional benefit of the method is that it also offers a column reliability score that estimates the probability that each column represents a correct alignment of residues.

A variety of tools based on the `ProbCons`' formulation of probabilistic consistency have been developed in the past few years [88, 89, 93, 97, 86, 11], with various aims of improving speed, accuracy (sensitivity), or specificity. All follow the same central dogma of determining consistency-based scores using only information about aligned pairs, with no regard for gap costs. The two recent tools out of Pachter's group, `AMAP` [97] and `FSA` [11], *do* somewhat account for gaps in the progressive alignment step, but this is only after gap-ignorant consistency modification is

performed, and still ignores the per-gap penalties that allow affine gap scoring to prefer long gap stretches over many short gaps.

4.2. Suboptimality as a measure of support for alignment features

In the equations above, a position c_h provides support for the alignment of a_i with b_j only if it is aligned with both. In a sense, this doesn't so much consider the *consistency* of alignments of C with A and B as it does the *constraint* they impose. For example, the following pairwise alignments of C with A and B are consistent with $(a_i \sim b_j)$, in the sense that they do not restrict that alignment from occurring:

$$\begin{array}{ccccc} \cdots & \mathtt{a_{i-1}} & \mathtt{a_i} & \cdots \\ & \mathtt{c_h} & - & \end{array} \qquad \text{and} \qquad \begin{array}{ccccc} \cdots & \mathtt{c_h} & - & \cdots \\ & \mathtt{b_{j-1}} & \mathtt{b_j} & \end{array}$$

Consistent, but non-constraining, alignments may provide more flexibility for identifying support of one sequence for alignment of another. But because of the high level of freedom such alignments have, they incur extra computational burden, so we follow in the footsteps of the methods above, and consider only alignments involving other sequences C that *constrain* the alignment of A and B to have particular features, though we continue to describe the approach as consistency-based, in agreement with standard nomenclature.

Our approach departs from other methods in two ways:

1. It explicitly accounts for gaps. Rather than compute only consistency-modified substitution scores and align with no-cost gap model, we compute consistency-modified costs for all features of an alignment under affine gap costs: substitution, gap-open, gap-extension, and gap-close costs.

2. Rather than ask "which optimal alignments support this alignment feature?", our method asks "what is the minimum suboptimality required of other alignments in order for those alignments to support this feature?". The first question is the one asked by the consistency methods of `T-Coffee` and `MAFFT`. The latter is similar to that asked by `ProbCons`, though in fact, `ProbCons` goes further by effectively soliciting support from all alignments, not just the least suboptimal.

4.2.1. Modified substitution score

We begin by describing a simple method of computing the support by a sequence C for a pairwise substitution, $(a_i \sim b_j)$, a diagonal edge in the dynamic programing matrix. Similar modified costs for other edges are described later, and a more complete discussion of modification equations follows in the next section.

The extent of agreement that position c_h has with substitution $(a_i \sim b_j)$ (Figure 4.1a), can be determined by computing the level of suboptimality required

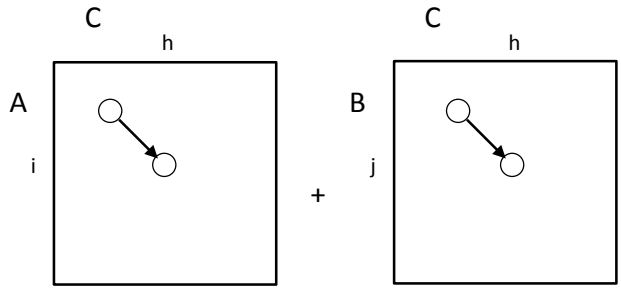


Figure 4.2: Subpaths of alignment graphs of $(A \sim C)$ and $(B \sim C)$ that are consistent with $(a_i \sim b_j)$ (figure 4.1a). The alignment that agrees with these subpaths is:

$$\begin{array}{ccc} & \mathtt{a_i} & \\ \cdots & \mathtt{c_h} & \cdots \\ & \mathtt{b_j} & \end{array}$$

---

of alignments $(A \sim C)$ and $(B \sim C)$ to be consistent with the substitution. Since our definition of consistency requires constraint, the only way c_h can be consistent with $(a_i \sim b_j)$ is if c_h is aligned with both: $(a_i \sim c_h \sim b_j)$. This can be represented as a pair of alignment subpaths, one for $(A \sim C)$ and one for $(B \sim C)$, as in Figure 4.2.

Let $T(A, C)$ be the optimal (minimum) cost of aligning A and C, and let $T_{ih}(A, C)$ be the cost of the optimal alignment of those sequences, under the constraint that the alignment is forced to go through the diagonal edge $(a_i \sim c_h)$. Then the suboptimality of using that edge is defined as

$$S_{ih}(A, C) \;:=\; \frac{T_{ih}(A, C) - T(A, C)}{T(A, C)}. \tag{4.4}$$

$S_{jh}(B, C)$ is defined similarly. The suboptimality required of C in order to support $(a_i \sim b_j)$ can be defined by picking the best position c_h :

$$\mathrm{sub}(a_i, b_j, C) \;:=\; \min_h \Big\{ \max \{ S_{ih}(A, C) \;,\; S_{jh}(B, C) \} \Big\}. \tag{4.5}$$

Suppose the values of $S_{ih}(A, C)$ and $S_{jh}(B, C)$ are capped at Δ (this is done for reasons of computational efficiency, as described later). Then one way of modifying

the cost of substitution $(a_i \sim b_j)$ given a single other sequence C is to define the modified cost as:

$$D(a_i, b_j) \;:=\; \sigma(a_i, b_j)\left(1 + F \cdot \frac{\mathrm{sub}(a_i, b_j, C)}{\Delta}\right), \tag{4.6}$$

where $\sigma(x, y)$ is the cost of substitution x for y, as from a substitution cost matrix (e.g. BLOSUM62 transformed to a cost matrix). Under this definition, an aligned pair $(a_i \sim b_j)$ with consistent subpaths that are optimal ($\mathrm{sub}(a_i, b_j, C) = 0$) will have cost $\sigma(a_i, b_j)$, while an aligned pair $(a_{i'} \sim b_{j'})$ with maximally suboptimal consistent subpaths will have that cost multiplied by $(F + 1)$.

Equation 4.6 can be extended to computing modifications based on a *set* of other sequences $\mathbb{C}$ by taking the average suboptimality over the sequences $C \in \mathbb{C}$:

$$D(a_i, b_j) \;:=\; \sigma(a_i, b_j)\left(1 + F \cdot \frac{\sum_{C \in \mathbb{C}} \mathrm{sub}(a_i, b_j, C)}{|\mathbb{C}| \cdot \Delta}\right). \tag{4.7}$$

The consistency definitions provided here are fairly simple. Somewhat more complicated (and more successful) methods are discussed in Section 4.3.

4.2.2. Modified gap extension scores

The modified cost described above is a substitution cost, which corresponds to a diagonal edge in the dynamic programming table of alignment $(A \sim B)$, as in Figure 4.1a. Computing modified costs for other features is done in a similar fashion. For example, a vertical gap extension is shown in Figure 4.1b, and corresponds to character a_i being part of a run of characters from A that align to a gap in B between b_j and b_{j+1}. A modified vertical gap extension cost can be based on the level of suboptimality required for alignments $(A \sim C)$ and $(B \sim C)$ to be consistent with the edge of $(A \sim B)$ shown in Figure 4.1b. One such consistent pair of alignment subpaths involving C is shown in Figure 4.3. An exhaustive list of such subpath-pairs, consistent with both vertical and horizontal (Figure 4.1c) gap extensions, is given in appendix Section A.0.2.

Let $T(B, C)$ be defined as before, and let $T_{j \prec h}(B, C)$ be the cost of the optimal alignment of $(B \sim C)$ when that alignment is forced to go through the horizontal edge leading to cell (j, h) of the dynamic programming table (the right half of Figure 4.3). Then the suboptimality of that edge is defined as

$$S_{j \prec h}(B, C) \;:=\; \frac{T_{j \prec h}(B, C) - T(B, C)}{T(B, C)}, \tag{4.8}$$

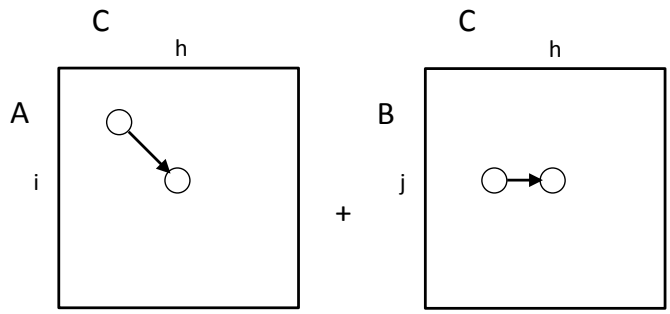


Figure 4.3: Subpaths of alignment graphs of $(A \sim C)$ and $(B \sim C)$ that are consistent with a_i extending a gap after b_j (Figure 4.1b). The alignment that agrees with these graphs is:

$$\begin{array}{ccc} & \circ & \mathtt{a_i} \\ \cdots & \circ & \mathtt{c_h} \quad \cdots \\ & \mathtt{b_j} & - \end{array}$$

(where $\circ$ represents freedom to either include or not include a character from the associated sequence)

---

and the suboptimality required of C in order to support the edge of $(A \sim B)$ shown in Figure 4.1b can be defined by picking the best position c_h, for the best subpath pair p among all consistent subpath pairs $\mathcal{P}$:

$$\text{ext}(a_i, b_j, C) \;:=\; \min_{p\mathcal{P}}\left\{\min_{h}\left\{\max\left\{S_{ih}(A,C)\;,\; S_{j\prec h}(B,C))\right\}\right\}\right\}. \tag{4.9}$$

With the cap Δ on the value of $S(\cdot)$ suboptimality values, the modified vertical gap extension cost due to a set of other sequences $\mathbb{C}$ can be defined as:

$$V(a_i, b_j) \;=\; \lambda\left(1 + F \cdot \frac{\text{avg}_{C\in\mathbb{C}}\{\text{ext}(a_i, b_j, C)\}}{\Delta}\right), \tag{4.10}$$

where λ is the unmodified cost of a gap extension.

4.2.3. Modified per-gap scores

Per-gap costs can be modified in a similar way. In our model, the typical per-gap cost (γ) is divided into two halves, the gap-open cost and the gap-close cost, each with value $\gamma/2$. This is done to ensure that an alignment under the modified

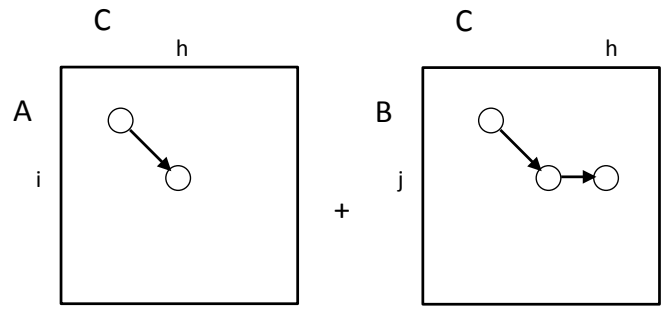


Figure 4.4: Subpaths of alignment graphs of $(A \sim C)$ and $(B \sim C)$ that are consistent with a_i opening a gap immediately after b_j (figure 4.1d). The alignment that agrees with these subpaths is:

$$\begin{array}{ccccc} & & \circ & \mathtt{a_i} & \\ \cdots & \mathtt{c_{h-1}} & \mathtt{c_h} & \cdots \\ & & \mathtt{b_j} & - & \end{array}$$

(where $\circ$ represents freedom to either include or not include a character from the associated sequence)

---

cost scheme will have the same score if its columns are reversed. Here, we give an example of computing modified vertical gap-open costs (Figure 4.1d). A vertical gap-open corresponds to a_i being the first character in a run of characters from A that are aligned to a gap between positions b_j and b_{j+1}. Horizontal gap open costs (Figure 4.1f), and both vertical (Figure 4.1h) and horizontal (Figure 4.1j) gap-close costs are computed similarly. One pair of subpaths involving C that is consistent with the horizontal gap open of Figure 4.1f is shown in Figure 4.4. An exhaustive list of such graph-pairs, consistent with both horizontal and vertical gap boundaries, is given in appendix Section A.0.3.

Let $T(B, C)$ be defined as before, and let $T_{j|\prec h}(B, C)$ be the cost of the optimal alignment $(B \sim C)$ when that alignment is forced to open a gap with c_h being the first entry of the gap following b_j (the right half of Figure 4.4). Then the suboptimality of that path is

$$S_{j|\prec h}(B, C) \;=\; \frac{T_{j|\prec h}(B, C) - T(B, C)}{T(B, C)}, \tag{4.11}$$

and the suboptimality required of C in order to support the gap open in $(A \sim B)$ shown in Figure 4.1f can be defined by picking the best position c_h, for the best

subpath pair p among all consistent subpath pairs $\mathcal{P}$:

$$\text{open}(a_i, b_j, C) \;=\; \min_{p \in \mathcal{P}} \left\{ \min_{h} \left\{ \max \left\{ S_{ih}(A, C) + S_{j|\prec h}(B, C) \right\} \right\} \right\}. \tag{4.12}$$

With the cap Δ on the value of $S(\cdot)$, the modified vertical gap open cost due to a set of other sequences $\mathbb{C}$ can be defined as:

$$V_o(a_i, b_j) \;=\; \frac{\gamma}{2} \left(1 + F \cdot \frac{\text{avg}_{C \in \mathbb{C}} \{ \text{open}(a_i, b_j, C) \}}{\Delta} \right), \tag{4.13}$$

where $\gamma/2$ is the unmodified gap open cost.

4.2.4. Support from multiple parameter choices

This section has described a method for computing consistency-modified costs based on the suboptimality information found in a single alignment for each pair of sequences. In their `MCoffee` paper, Wallace et al. [112] described a method that develops consistency-modified scores based on multiple alignments generated by multiple software tools. A similar idea can be harnessed internally, generating pairwise alignments with multiple parameter choices, and computing alignment feature support from that set of pairwise alignments. The idea is based on the fact that we don't know which parameters are optimal for a given input, but hope that the relatively good parameters will tend to agree, and overwhelm the input of the few poor parameters. This is related to the underlying motivation of the elision method [116].

We tested this idea with a wide range of parameter combinations, using up to 10 parameter choices, but found slight degradation in average alignment accuracy relative to simply using the single parameter choice that performs best.

4.3. Alternate definitions for consistency-modified costs

In the previous section, simple equations (4.7, 4.10, and 4.13) were given for computing modified alignment costs based on graph-derived suboptimalities. The equations have explanatory value, but did not perform well in our tests. Alternate equations are given in this section. The focus of this section will be on computing modified substitution costs. Equations for computing modified gap costs are not given, but are easy to derive, essentially by replacing the $\sigma(a_i, b_j)$ values with either λ or $\gamma/2$ values, as in the previous section.

We call the various equations *consistency blends*, because they all aim to blend unmodified costs for features in an $(A \sim B)$ alignment with the extent of suboptimality required of pairwise alignments involving other sequences C in order to be consistent with those features.

4.3.1. Imbalanced suboptimality blend variants

The equations from the previous section are what we call the *imbalanced maximum suboptimality blend*. For each sequence $C \in \mathbb{C}$, the suboptimality $\text{sub}(a_i, b_j, C)$ comes from the position c_h giving the smallest *maximum* of $S_{ih}(A, C)$ and $S_{jh}(B, C)$ (see equation 4.5). These suboptimalities are averaged over all sequences C, normalized by the maximum suboptimality allowed (Δ), and blended with the *unmodified* cost of the $(A \sim B)$ feature to give the modified feature cost.

A similar modification can be called the *imbalanced average suboptimality blend*. Under this method, for each sequence $C \in \mathbb{C}$, the suboptimality comes from the position c_h giving the smallest *average* of $S_{ih}(A, C)$ and $S_{jh}(B, C)$.

$$\text{sub}(a_i, b_j, C) \;:=\; \min_h \left\{ \frac{S_{ih}(A, C) \;+\; S_{jh}(B, C)}{2} \right\}. \tag{4.14}$$

The modified feature cost is computed as before.

One problem with these methods is that, when computing a modified cost for a feature in the alignment graph of $(A \sim B)$, they fail to account for how suboptimal that feature of the $(A \sim B)$ alignment is in its own (unmodified) alignment graph. This represents an imbalance, where suboptimality is considered only for features in (A, C) and (B, C) alignments, but not (A, B) alignments. This can be resolved with a slightly more complicated equation, which imposes a balance between internal and external suboptimality.

4.3.2. Balanced suboptimality blend variants

Let $S_{ij}(A, B)$ be defined as $S_{ih}(A, C)$ was defined in the previous section, and capped at a maximum value of Δ. One natural way to insert self-suboptimality into the equation is by blending it into the numerator of equation 4.7:

$$D(a_i, b_j) \;=\; \sigma(a_i, b_j) \left(1 + F \cdot \frac{(1-\alpha) \cdot S_{ij}(A, B) + \alpha \cdot \text{avg}_{C \in \mathbb{C}}\{\text{sub}(a_i, b_j, C)\}}{\Delta} \right), \tag{4.15}$$

where $0 \leq \alpha \leq 1$ is a blending coefficient.

Note that this equation now features two tuning coefficients, F and α. A single-parameter variant that works equally well in our tests is

$$D(a_i, b_j) \;=\; \sigma(a_i, b_j) \Big(1 + \frac{S_{ij}(A, B) + F \cdot \text{avg}_{C \in \mathbb{C}}\{\text{sub}(a_i, b_j, C)\}}{\Delta \cdot (1 + F)} \Big). \tag{4.16}$$

If all suboptimalities are zero, the modified cost of the feature will be $\sigma(a_i, b_j)$, while maximal suboptimality will cause the modified feature cost to be doubled. We call this equation the *balanced suboptimality blend*, and it can be based on either the maximum or average suboptimality for each pair $(A \sim C)$ and $(B \sim C)$ (Equations 4.5 and 4.14, respectively).

4.4. Easing the computational burden

The methods described above for computing consistency-modified feature costs are computationally demanding. In the default form, run time to compute consistency costs will be an impractical $O(k^3n^3)$: for each pair of sequences, (A, B), every other sequence C is considered, and for each such (A, C, B) triple, all (a_i, c_h, b_j) position-triples are viewed. This is much more than the practical run times of the related tools: `T-Coffee` and `ProbCons` require time more like $O(k^3n^2)$, and `MAFFT`'s takes time $O(k^2n^2)$, as discussed in Section 4.1.

Also, in the simple method described in the previous sections, the entire alignment matrix is stored for each pairwise alignment, in order to compute suboptimality values. This consumes default space $O(k^2n^2)$, which is greater than the $O(k^2n)$ space used by `MAFFT` and `T-Coffee` to store a single optimal alignment of each pair. It is also greater than that used by `ProbCons`, which in principle could require $O(n^2)$ space per alignment to store posterior probabilities for all position pairs, but in practice consumes less since it maintains a sparse matrix holding only non-negligible values.

The practical run time for computing consistency costs can be reduced to $O(k^2n^2)$ by using two orthogonal ideas: (1) compute consistency-modified scores based on a fixed-sized set of sequences $\mathbb{C}$, and (2) limit the number of positions c_h that are viewed in searching for c_h with minimum suboptimality. Further speed up is possible if consistency-modified costs are only used for the few alignments performed near the tips of the guide tree (see Section 2.4) in the progressive alignment method. As implemented for our experiments, alignment with consistency is quite slow even with these speedups, taking slightly longer to complete (with no polishing) than standard alignment with polishing takes.

4.4.1. Computing consistency from $\mathbb{C}$ of fixed size

In the methods of `T-Coffee` and `ProbCons`, per-position scores for sequence-pair (A, B) are derived by looking at all other sequences C. This results in a time bound of $O(k^3)$ for constant n. By restricting the number of sequences C that are used to compute consistency-based costs for $(A \sim B)$, this bound is reduced. We do this by creating a subset of $\mathbb{C}$ for each pair (A, B), $\mathbb{C}_{AB}$. Then, for example, equation 4.7 can be altered to be

$$D(a_i, b_j) \;=\; \sigma(a_i, b_j)\left(1 + F \cdot \frac{\text{avg}_{C \in \mathbb{C}_{AB}}\{\text{sub}(a_i, b_j, C)\}}{\Delta}\right), \qquad (4.17)$$

with other equations altered similarly.

This is a seemingly reasonable thing to do, since the sequences most distant from A and B can probably not contribute much information about the alignment of $(A \sim B)$ that isn't found in more similar sequences. In other words, sequences C

that are similar to sequences A and B are more likely to provide accurate support for features in the alignment $(A \sim B)$. Based on this observation, our approach is to compute pairwise distances for all sequences, with $d(A, B)$ equal to the normalized cost from Section 2.4. Then for each sequence-pair (A, B), the X sequences from $\{\mathbb{C} - \{A, B\}\}$ with smallest $d(A, C) + d(B, C)$ are assigned to $\mathbb{C}_{AB}$. In our tests, we found $X = 5$ to result in accuracy as high as larger values of X.

Note that it is also possible to assign weights to the contributions of various sequences to the consistency equations. Since sequences that are more similar to A and B are more likely to provide accurate signal to their alignment, we could apply higher weights to the suboptimality values of more similar sequences. This was not found to improve accuracy. Creating a strict limit to the number of utilized sequences amounts to a step function for weighting sequences: the closest X sequences all get weight 1, and others get weight 0.

4.4.2. Computing consistency using a band around optimal alignments

For a pair of sequences (A, B) and a third sequence C, the default time to compute consistency-modified feature costs is $O(n^3)$ for sequences of length n: for each position-pair (a_i, b_j), all positions c_h are considered. But the practical runtime can be reduced by considering a restricted set of positions in C for alignments $(A \sim C)$ and $(B \sim C)$, specifically the positions that are likely to be involved with the least suboptimal subpath pairs. One way to achieve this is to consider, for a position-pair (a_i, b_j), only those positions c_h that fall inside a band around optimal alignments of $(A \sim C)$ and $(B \sim C)$. This is sensible, since the support, as in equation 4.5, comes from the position c_h with lowest suboptimality, and positions (a_i, c_h) that are far from an optimal alignment $(A \sim C)$ are unlikely to provide the lowest suboptimality support levels.

Because there can be more than one optimal scoring alignment $(A \sim C)$, the leftmost and rightmost optimal paths in the dynamic programming table are determined. Then, for each row i in that table (corresponding to position a_i), a left-to-right window is established that extends a few cells to the left of the leftmost optimal alignment and a few cells to the right of the rightmost optimal alignment. Note that the window width may be different for each row, depending on the flexibility of optimal alignments involving each position a_i (see Figure 4.5).

One approach to specifying what "a few cells to the left" means is to extend the window a fixed number of cells w outside the left- and right-most optimal alignments. An alternative, which we found to be more successful, is to extend the window as far as possible to include all cells (a_i, c_h) such that the optimal alignment constrained to align $(a_i \sim c_h)$ has cost no more than a fixed value Δ higher than

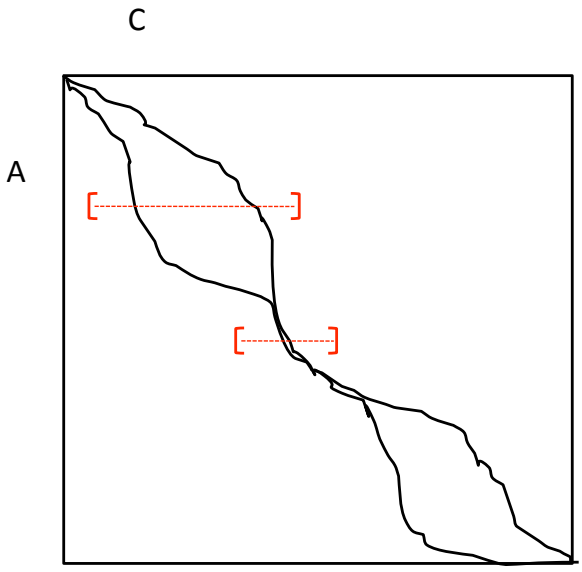


Figure 4.5: Size of windows in alignment $(A \sim C)$ that will be used to compute suboptimality measures to alignment $(A \sim B)$. Window width may vary for positions a_i, depending on the horizontal range of left-most and right-most optimal alignments.

---

the optimal alignment of $(A \sim C)$. Note that this is the Δ used as a suboptimality limit in previous sections. In our experiments, we found $\Delta = 150$ to result in high accuracy while limiting computation. With substitution costs ranging between 0 and 88, and the cost of 98 for a gap of length one (see Section 2.9), this allows for a little flexibility in alignment suboptimality, but does not include alignments that require multiple clearly erroneous features.

Note that the range of good h-values for the i^{th} row of $(A \sim C)$ may disagree with the range of good h-values for the j^{th} row of $(B \sim C)$. We found that it was sufficient to search for the lowest suboptimality only the intersection of those ranges (see Figure 4.6); if the ranges did not intersect, then the suboptimality was defined as Δ.

In addition to reducing run time, space utilization is reduced as well, based on the combination of restricting $\mathbb{C}_{AB}$ and using suboptimality windows. The default method would require (possibly infeasible) space $O(k^2n^2)$ to store the full dynamic programming table for all pairwise alignments. This can be dramatically reduced by recomputing alignments $(A \sim C)$ as they are needed for calculating consistency-based costs for other alignments $(A \sim B)$. For each pair (A, B), compute and store

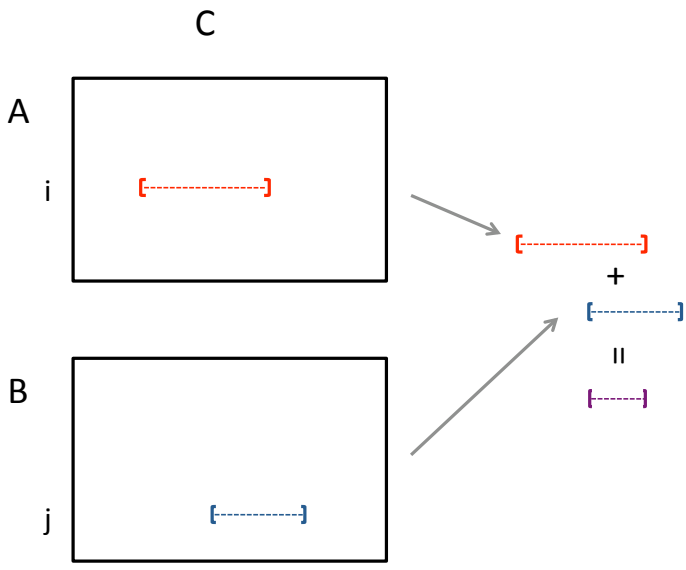


Figure 4.6: The range used in consistency is the intersection of $(A \sim C)$ and $(B \sim C)$ windows.

---

the alignments $(A \sim B)$ and for a small set of C, $(A \sim C)$ and $(B \sim C)$, storing only the portions of those tables falling inside the suboptimality windows. When X neighboring sequences are used to compute consistency, this will require an X-fold increase in the number of pairwise alignments performed over the default, but result in a large storage savings, requiring $O(n)$ space (for fixed X) in practice.

4.4.3. Consistency only for small subtrees

Even with the speedups described above, this method is quite slow for inputs with more than a dozen sequences. Run time can be reduced substantially if modified costs are computed for only a fraction of the pairs of sequences. We achieve this by restricting consistency-based alignment to only those nodes of the progressive alignment merge tree (see Section 2.4) that involve a small number of sequences. In other words, consistency-modified costs are used for nodes close to the leaves, and standard costs for deeper internal nodes. See Figure 4.7 for an example of the resulting reduction in computation.

We should also note that the fast profile alignment methods described by Gotoh and Starrett [45, 103] can not be applied when position-pair costs are used, so the aligning alignments step will tend to grow quadratically with number of sequences,

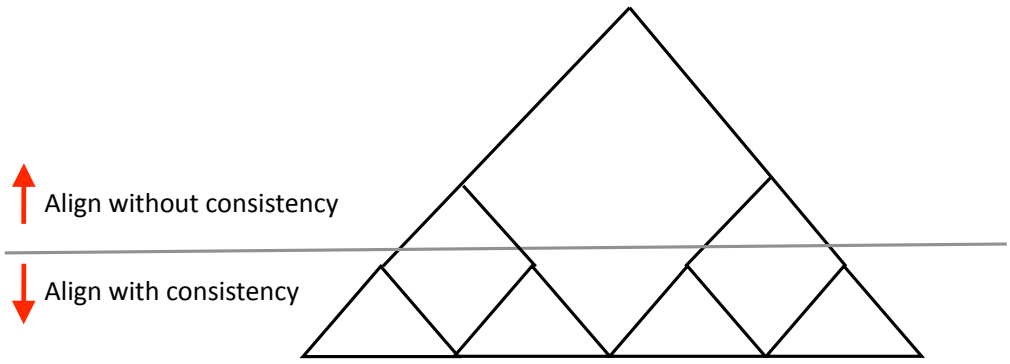


Figure 4.7: Number of pairwise alignments performed is reduced by limiting consistency to nodes near leaves. For example, suppose each pictured subtree contains 6 sequences. Then if all pairs are computed, the total number of pairs is $24 * 23/2 = 276$, but if pairs are only computed within subtrees, then $6 \cdot 5/2 = 15$ pairs are computed for each subtree, for a total of 60. If the number of sequences is doubled to 48, but the same threshold retained (thus doubling the number of size-6 subtrees), then the number of computed pairwise alignments will be reduced from 1128 to 120.

---

rather than linearly. By limiting the size of alignments in which consistency is used, the fast profile methods can be used for the large alignments formed at deep internal nodes.

This form of tree-restricted consistency is necessary to achieve acceptable speed, but also seems well-motivated. The purpose of using consistency modified costs is to avoid making errors early in the progressive alignment process. Once alignments being merged have reached sufficient size, there should be enough signal in those alignments to overcome the need for external influence on alignment costs. When no polishing is used, we found that there was effectively no benefit in our scheme to using consistency on subtrees larger than 6 sequences. However, our experiments (Section 4.6) show that restricting consistency to subtrees results in conflicts with polishing.

4.5. Incorporating consistency-modified costs into the algorithm for aligning alignments

It is a straightforward exercise to incorporate consistency-based costs into the algorithm for aligning two sequences, and into the pessimistic heuristic for aligning alignments (see Section 2.5 and [64]), replacing the matrix-derived substitution

(a) Flat: substitution

(b) Overhang: vertical gap

(c) Underhang: horizontal gap

Figure 4.8: Shapes in a two-sequence alignment

costs and fixed gap costs λ and γ with the modified variants described in Sections 4.2 and 4.3. Modifications of the algorithm for aligning alignments optimally (with exact gap counts, see Section 2.5 and [63, 103]) are only slightly more complicated. Here we discuss details of that algorithm only in the depth required to see the modifications necessary to incorporate consistency. See Section 2.5 and [64, 63, 103] for more details.

The algorithm for exactly aligning alignments follows a standard dynamic programming approach, and can be described as it relates to aligning two sequences. In aligning two sequences under affine gap costs, a subproblem (i, j) corresponds to prefixes $(a_1 \dots a_i)$ and $(b_1 \dots b_j)$ of sequences A and B, and a shape that represents the relative ordering of the final characters of the two prefixes. The flat shape (Figure 4.8a) indicates that a_i is aligned to b_j (a substitution). The overhanging shape (Figure 4.8b) corresponds to a run of characters from A ending in a_i aligning to a gap between positions b_j and b_{j+1}; an overhang is a run of vertical edges in the dynamic programming table. The underhanging shape (Figure 4.8c) similarly corresponds to a run of characters from B in a (horizontal) gap. A direct result of the dynamic programming recurrence is that, when filling in a matrix with the costs of solutions to these subproblems from the upper-left portion of the matrix to lower-right portion is that, for each of the three shapes, only the cost of the cheapest path (alignment of prefixes) ending in that shape in that cell need be retained. This is effectively a pruning method: rather than keep track of all alignments of prefixes $(a_1 \dots a_i)$ and $(b_1 \dots b_j)$, only one for each shape is tracked.

In the problem of aligning two **alignments**, the alignments $\mathcal{A}$ and $\mathcal{B}$ are viewed as sequences of columns, and a subproblem consists of prefixes of the two alignments, along with the shape leading to those prefixes. As in pairwise alignment, shapes describe the relative order of occurrence of the most-recent letter in each sequence. Multi-sequence shapes are more complicated, as sequences within an alignment may have gaps relative to other sequences in the alignment. See Figure 4.9 for an example.

The difficulty in aligning alignments optimally is in correctly computing the

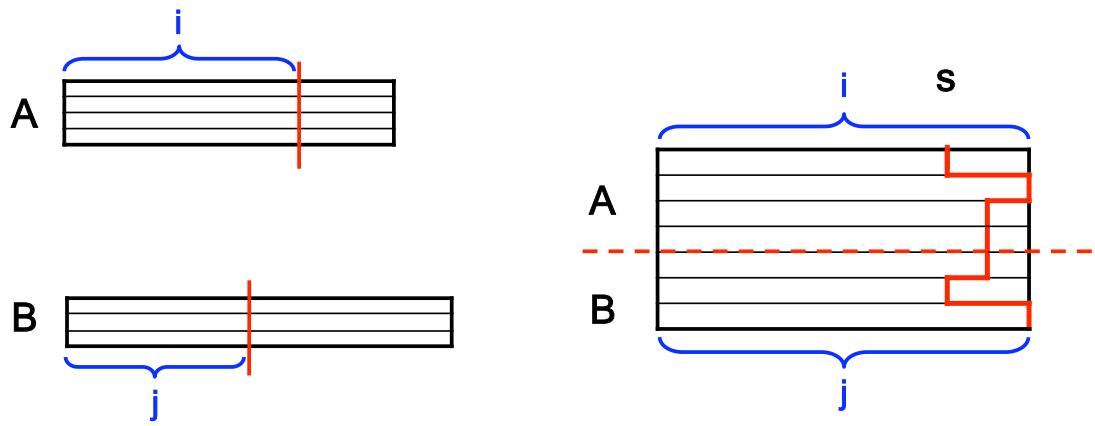


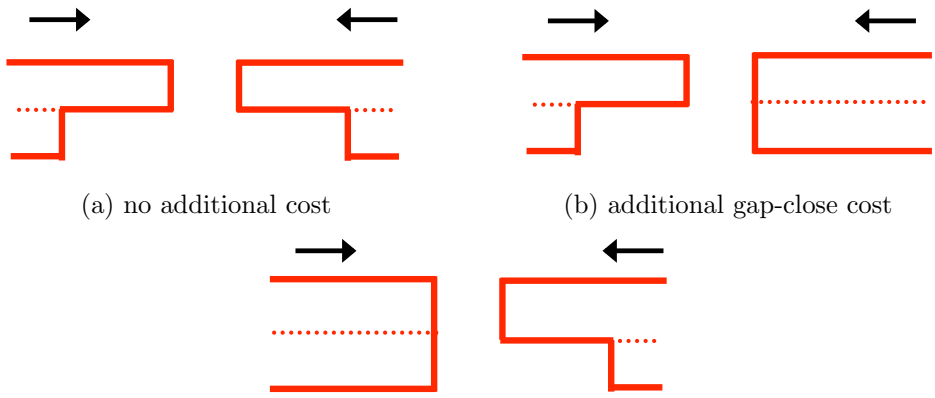
Figure 4.9: Multiple sequence shape.

per-gap cost of an alignment as it is extended in the dynamic programming matrix. Using standard costs, this amounts to correctly counting the number of gaps opened by a new column in the alignment. Shapes are the key to determining whether a column starts a gap in a pair of rows: for example, if a new column appends a letter from A against a gap in B, the effect is to create an overhang. If that column is appended to a shape that is not already an overhang (Figure 4.8b), then a new gap $(a, -)$ has been opened.

As in the two-sequence method, a sort of rudimentary pruning is possible: many paths may result in the same shape at a given cell in the dynamic programming matrix, but it is simple to ensure that each shape retained in that cell represents the cheapest path leading to that cell and ending at that shape. But more effective pruning of shapes is possible, as well; if it can be shown that a shape s can not possibly lead to an optimal alignment, then that shape may be pruned immediately. Two such improved pruning techniques are described in the work of Kececioglu and Starrett [63, 103], *bound pruning* and *dominance pruning*. These require special attention in terms of how they incorporate consistency-modified costs. The descriptions below present the issues in terms of two-sequence shapes induced by a multi-sequence shape; this is natural, as alignment costs are based on sum-of-pairwise costs.

4.5.1. Bound pruning

The bound pruning method uses heuristic alignments to establish bounds to support shape pruning. First, a heuristic alignment is performed on the reversed alignments



(a) no additional cost

(b) additional gap-close cost

(c) additional gap-open cost

Figure 4.10: Stitching shapes for bound pruning. When considering cost of extending a shape from the left with a shape from the right, additional costs may be incurred.

$\mathcal{A}$ and $\mathcal{B}$. Think of this as building an alignment from the bottom-right up to the top-left on the un-reversed alignments. The heuristic used is the optimistic gap-count heuristic [63], in which per-gap costs are counted only when it is clear that a new gap has been formed. At each cell (a_i, b_j) in the dynamic programming table, these (optimistic) costs provide a lower bound on the cost of aligning the suffixes $(a_i \dots a_m)$ and $(b_j \dots b_n)$. These suffix-alignment lower bounds can be used to establish cost bounds on extending alignments of prefixes (in the forward direction).

The true cost of this heuristic alignment is then computed. Since it is a feasible alignment, this represents an upper bound on the optimal cost of an alignment (in practice, a pessimistic heuristic alignment is also performed, since this tends to have lower cost).

In the optimal alignment algorithm, the general step is to consider a shape s in cell (i, j), where the cost of the cheapest alignment at that cell ending in that shape is $c(i, j, s)$. The idea is to find a lower bound on the cost of extending s into a complete alignment; if that lower bound exceeds the previously established upper bound (from the optimistic or pessimistic heuristic), then s may be discarded.

At cell (i, j), there are three possible ways to begin the extension: right, down, and diagonally down/right. The reverse alignment described above establishes a lower-bound on the cost of extension in each direction, so a lower-bound on total cost can be established by adding the cost of s (which is the exact cost of the optimal alignment of prefixes leading to that cell with that shape) to the lower bound on cost of extension in each direction.

In the normal (non-consistency) scoring scheme, the per-gap cost was implemented as a single gap-open cost γ. Thus, when an overhang in shape s is extended into a similarly overhanging shape from the reverse alignment (Figure 4.10a), both shapes had already incurred a per-gap cost. This required that the bound be corrected by reducing the summed costs based on possible gap-open overcharges. This issue it avoided by splitting the per-gap cost into gap-open and gap-close costs, each $\gamma/2$. With these split-gap costs, no overcounting is possible, either in the standard or consistency-modified scoring schemes. With these split costs comes a new requirement: to keep the bound as tight as possible, all possible gap-close (Figure 4.10b) and gap-open (Figure 4.10c) costs are now added to the lower-bound on the cost of extending a shape.

4.5.2. Dominance pruning

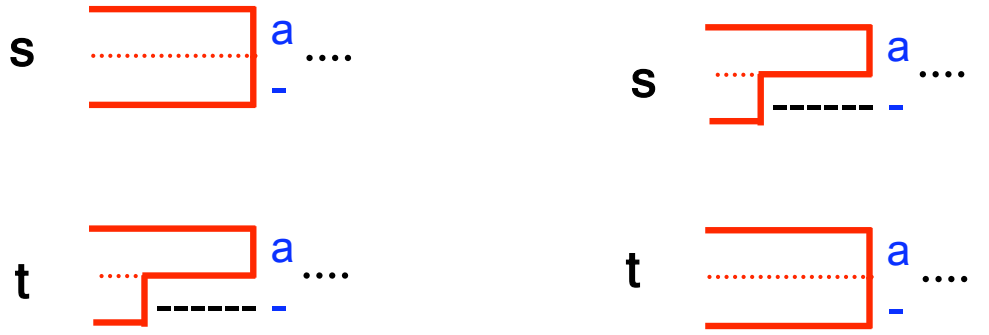
If two paths to the same cell end in the same shape, then the one with lower cost can be said to *dominate* the other: because the two shapes will incur identical costs for any possible extension, there is no way that the more expensive path to that shape can lead to an alignment with a cheaper cost. A dominated path can then be discarded while still guaranteeing that an alignment with optimal cost will be found.

This notion can be generalized to comparing different shapes. For two shapes in a cell, s and t, if there is no possible path through the remainder of the alignment graph that will cause the cost of t extended through that path to be less than the cost of s extended through that path, then s can be said to dominate t, and t can be discarded. It is not enough that $c(i, j, s) < c(i, j, t)$, because it is possible that some path will force s to incur a new cost that t does not incur (s opens a gap that t already has open, or s closes a gap that isn't open in t), thus causing the cost of extending s to rise relative to the cost of extending t. A bound on potential excess gap-open costs for s can be computed by counting the number of pairs for which t has an open gap that s does not have (Figure 4.11a). A similar accounting for excess gap-close costs can be made, by counting the pairs for which s has an open gap where t does not (Figure 4.11b).

Let B be the sum, over all pairs of sequences, of all such possible gap-open or gap-close costs incurred by s and not t (Figures 4.11a and 4.11b). Then s dominates t if $c(i, j, s) + B \leq c(i, j, t)$. Thus, dominance is an easily-tested sufficient condition for t to be no better than s.

4.6. Experimental results

The accuracies of the blend methods described in the Section 4.3 are compared in Table 4.1. The methods were implemented in `Opal`, and experiments were performed as in Chapter 2 (using benchmarks `BAliBASE`, `PALI`, and `SABmark`;



(a) Extra gap-open cost.

(b) Extra gap-close cost.

Figure 4.11: Extra gap-open and gap-close costs in dominance pruning. For two shapes s and t, each pair of sequences with induced pairwise shapes shown here cause the cost of s to grow, relative to the cost of t. In (a) the cost growth is the cost of opening a gap. In (b) the cost growth is the cost of closing a gap.

---

measuring accuracy with the SPS and TC scores). Inputs with no more than 50 sequences were used, due to excessive run time for the algorithm as implemented. A range of blend factors (the F value) were tested, and results shown are under the best F for each method. Consistency was computed only on subtrees of 6 or fewer sequences, with larger subtrees aligned under standard scoring. Consistency support was gathered from 5 nearest neighbors for each pair (A, B), and $\Delta = 150$ was used in all cases. The methods are compared to default alignment in `Opal`, with no polishing. The balanced maximum suboptimality method was found to perform best. Results presented in Section 2.8 indicate that the gains were roughly the same as gains from polishing, and consistency and polishing did not work well together.

One valid concern about these tests is that the observed success of basing consistency-modification on 5 related sequences may be due to the constraint placed on the number of sequences in tested inputs. We addressed this concern by using a similar methodology with `MAFFT` at the core. Because `MAFFT`'s method is much faster, all inputs from `PALI` and `BAliBASE` with more than 60 sequences could be tested (`SABmark` has no inputs of that size). After computing a guide tree, all subtrees with no more than X sequences were aligned with `MAFFT`'s local-alignment-based consistency method (L-INS-1), while all internal nodes with subtrees containing more than X sequences were aligned using `MAFFT`'s profile alignment with unmodified parameters. Table 4.2 shows accuracy results over a range of values for X. Recovery as measured by SPS leveled off at $X = 15$, though it is interesting to note that the TC recovery values continued to grow with X, and

Table 4.1: Comparison of consistency methods. All results are without polishing.

Consistency method	BAliBASE	SABmark	PALI	average
balanced maximum (F=1.9)	**84.3** / **56.7**	**49.9**	84.7 / 59.6	**73.0** / **58.2**
balanced average (F=3.0)	83.9 / 56.4	49.2	**84.8** / **60.0**	72.6 / **58.2**
imbalanced maximum (F=1.0)	83.1 / 55.3	45.7	84.3 / 58.7	71.0 / 57.0
imbalanced average (F=1.0)	82.9 / 55.1	45.1	84.2 / 59.1	70.7 / 57.1
no consistency	82.5 / 52.7	48.5	83.8 / 57.1	71.6 / 54.9

even $X = 40$ didn't attain TC values as high as using consistency for all inputs.

The results described so far are for consistency without polishing. In Section 2.8, it was reported that accuracy was reduced when polishing was applied to a consistency-based alignment. It is instructive to understand if this is true for other methods as well. Tables 4.3 and 4.4 show the recovery gains due to consistency and polishing on the same inputs as in Table 4.1, for MAFFT and ProbCons respectively. Both tools experience additive gains in accuracy by combining consistency and polishing, with ProbCons getting a small ($< 1\%$) gain in both SPS and TC score over just consistency, while MAFFT's gains were 2% and 3% in SPS and TC respectively.

To understand why these tools see gains in combining the methods, while Opal experiences a reduction in quality, it is important to remember how the Opal model works: consistency-modified costs are used when building alignments in small subtrees of the full guide tree, but standard (unmodified costs) are used for both deep internal nodes and for polishing. This partitioning of consistency was used because the computational burden of Opal's model was excessive for larger trees. The conjecture that this approach would still work well was rooted in the idea that using consistency on small trees would help avoid errors when information was sparse, but that information contained by larger alignments would be sufficient to overcome the need for consistency. However, it appears that this is likely the cause of the poor relationship between consistency and polishing. An analogous idea is to build a full alignment with MAFFT using consistency, then to apply MAFFT's standard-parameter (non-consistency) polishing to that alignment. When that was done, accuracy was worse than polish-free consistency-based alignment (results not shown). This, along with similar results in our model, suggests that polishing and consistency can cooperate effectively only if the parameters used to polish are the same ones used in computing an initial consistency-based alignment. When

Table 4.2: Accuracy of method using MAFFT's consistency approach (linsi) on small subtrees of the guide tree, and unmodified profile alignment on deep internal nodes. No polishing was used. Results shown for the 26 inputs from BAliBASE and PALI containing at least 60 sequences.

which internal-node alignments performed with consistency	SPS	TC
no consistency (fftns1)	90.8	40.6
subtrees $\leq$ 5 sequences	90.5	39.5
subtrees $\leq$ 6 sequences	91.4	41.2
subtrees $\leq$ 7 sequences	91.4	41.2
subtrees $\leq$ 8 sequences	90.8	40.8
subtrees $\leq$ 9 sequences	91.0	42.6
subtrees $\leq$ 10 sequences	91.0	43.3
subtrees $\leq$ 11 sequences	91.5	41.9
subtrees $\leq$ 12 sequences	91.6	43.5
subtrees $\leq$ 13 sequences	91.7	44.5
subtrees $\leq$ 14 sequences	92.1	42.4
subtrees $\leq$ 15 sequences	92.1	44.4
subtrees $\leq$ 16 sequences	91.9	44.9
subtrees $\leq$ 17 sequences	91.9	45.0
subtrees $\leq$ 18 sequences	91.9	45.1
subtrees $\leq$ 19 sequences	91.9	44.5
subtrees $\leq$ 20 sequences	92.1	43.5
subtrees $\leq$ 25 sequences	92.5	48.5
subtrees $\leq$ 30 sequences	91.2	49.7
subtrees $\leq$ 35 sequences	91.5	51.3
subtrees $\leq$ 40 sequences	91.3	49.2
all (lins1)	92.1	54.4

Table 4.3: Impact of using global- and local-sequence alignments in computing consistency-modified scores in MAFFT.

MAFFT variant	BAliBASE	SABmark	PALI	average
no consistency, no polish	77.8 / 47.4	42.0	79.2 / 50.2	66.3 / 48.8
with consistency, no polish	82.8 / 55.4	44.1	82.8 / 56.1	69.9 / 55.8
with consistency, with polish	84.6 / 59.2	46.2	84.0 / 59.2	**71.6** / **59.2**

Table 4.4: Impact of using consistency-modified scores in ProbCons.

ProbCons variant	BAliBASE	SABmark	PALI	average
no consistency, no polish	79.7 / 49.1	44.3	81.7 / 52.8	68.6 / 50.1
with consistency, no polish	83.0 / 56.4	46.7	84.3 / 59.2	71.3 / 57.8
with consistency, with polish	83.7 / 57.1	47.1	84.5 / 59.3	**71.8** / **58.2**

the polishing parameters differ from the initial-alignment parameters, the gains of consistency may be broken by polishing without getting sufficient return in the form of polishing-based improvement.

PART 2 LARGE-SCALE NEIGHBOR-JOINING PHYLOGENIES

CHAPTER 5

BACKGROUND ON NEIGHBOR-JOINING

The neighbor-joining method of Saitou and Nei [95] is a widely-used method for constructing phylogenetic trees, owing its popularity to good speed, generally good accuracy [81], and proven statistical consistency (informally: neighbor-joining reconstructs the correct tree given a sufficiently long sequence alignment) [35]. In particular, it has been highlighted as a valuable method for inferring very large phylogenies [108], due to its combination of speed and accuracy.

Neighbor-joining is a hierarchical clustering algorithm. It begins with a distance matrix, where d_{ij} is the observed distance between clusters i and j, and initially each of the n input sequences forms its own cluster. Neighbor-joining repeatedly joins a pair of clusters that are closest under a measure, q_{ij}, that is related to the d_{ij} values. The canonical algorithm [105] finds the minimum q_{ij} at each iteration by scanning through the entire current distance matrix, requiring $\Theta(r^2)$ work per iteration, where r is the number of remaining clusters. The result is a $\Theta(n^3)$ run time, using $\Theta(n^2)$ space. Thus, while neighbor-joining is quite fast for n in the hundreds or thousands, both time and space balloon for inputs of tens of thousands of sequences.

As a frame of reference, there are 8 families in Pfam [37] containing more than 50,000 sequences, and 3 families in Rfam [49] with more than 100,000 sequences, and since the number of sequences in GenBank is growing exponentially [41], these numbers will certainly increase. Phylogenies of such size are applicable, for example, to large-scale questions in comparative biology (e.g. [100]).

In this part of the dissertation, a new algorithm is described that produces a correct neighbor-joining phylogeny, and achieves increased speed by restricting its search for the smallest q_{ij} at each iteration to a small portion of the quadratic-sized distance matrix. The key innovations of this algorithm are (1) introduction of a search-space filtering scheme that is shown to be consistently effective even in the face of difficult inputs, and (2) inclusion of data structures that efficiently use disk storage as external memory in order to overcome input size limits.

The result is a statistically consistent phylogeny inference tool that is roughly an order of magnitude faster than a very fast implementation of the canonical algorithm, `QuickTree` (for example, calculating a neighbor-joining tree for 60,000 sequences in less than a day on a desktop computer), and is scalable to hundreds of thousands of sequences. The algorithm is implemented in a tool called `NINJA`, freely available at http://nimbletwist.com/software/ninja.

5.1. Canonical neighbor-joining algorithm

Neighbor-joining [95, 105] is a hierarchical clustering algorithm. It begins with a distance matrix, D, where d_{ij} is the observed distance between clusters i and j, and initially each sequence forms its own cluster. Neighbor-joining forms an unrooted tree by repeatedly joining pairs of clusters until a single cluster remains. At each iteration, the pair of clusters merged are those that are closest under a transformed distance measure

$$q_{ij} \;=\; (r-2)\, d_{ij} \;-\; t_i \;-\; t_j \;, \tag{5.1}$$

where r is the number of clusters remaining at the time of the merge, and t_i is the total distance of cluster i to all other clusters:

$$t_i \;=\; \sum_k d_{ik} \;. \tag{5.2}$$

When the $\{i, j\}$ pair with minimum q_{ij} is found, clusters i and j are joined by a new parental node, which now represents cluster ij, and the length of the edge from i to the parent node is

$$b_i \;=\; \frac{1}{2}\left(d_{ij} \;+\; \frac{t_i - t_j}{(r-2)} \right), \tag{5.3}$$

while the length of the edge from j to the parent node

$$b_i \;=\; \frac{1}{2}\left(d_{ij} \;+\; \frac{t_j - t_i}{(r-2)} \right). \tag{5.4}$$

D is updated by inactivating both the rows and columns corresponding to clusters i and j, then adding a new row and column containing the distances to all remaining clusters for the newly formed cluster ij. The new distance $d_{ij|k}$ between the cluster ij and each other cluster k is

$$d_{ij|k} \;=\; \frac{d_{i|k} \;+\; d_{j|k} \;-\; d_{i|j}}{2} \;. \tag{5.5}$$

There are n-1 merges, and in the canonical algorithm each iteration takes time $\Theta(r^2)$ to scan all of D. This results in an overall running time of $\Theta(n^3)$.

Note that it is possible for one of each pair of branch lengths to gain a negative length based on formulas 5.3 and 5.4; this is problematic since the length of an edge represents the extent of sequence evolution occurring along that edge, which should be non-negative. It has been suggested that the branch with negative length should be set to length zero, and the sibling branch shortened an equal amount. This does not alter the topology of the tree (Kuhner and Felsenstein, 1994).

Also note that it is possible that more than one pair will share the same minimum q value. It appears to be common to make an arbitrary choice, though it is certainly possible to choose randomly (an option in [98]), and repeat analysis of the input multiple times to measure robustness of resulting trees.

5.2. Properties of neighbor-joining

Numerous studies have demonstrated neighbor-joining's efficacy in recovering accurate phylogenies [67, 81, 104, 69], and in particular, Tamura et al. [108] observed very little reduction in accuracy as the number of sequences grows into the thousands. (It should be noted that accuracy results in phylogenetics depend on simulation studies, since the true evolutionary history or real sequences is unknowable.)

Neighbor-joining takes as input a matrix of estimated pairwise distances, and constructs a tree such that the pairwise distances induced by that tree are close to the input distances. Input distances are estimated from observed sequence data, usually based on a probabilistic model of sequence evolution. These models are statistically consistent in the sense that they converge on the true distance with sufficiently long sequences and a correct model of sequence evolution [35]. Neighbor-joining, which is known to correctly recover trees when pairwise distances are in agreement with the true tree (these are known as additive, or tree-like, distances) [39] is statistically consistent as well.

In practice, the model of sequence evolution may be wrong, and distances are calculated on finite-length sequence, meaning computed distances are subject to statistical error. It is therefore important to understand how neighbor-joining deals with erroneous estimated pairwise distances. Consider the true tree under which the sequences evolved. The edge lengths of that tree induce a distance between each pair of sequences: the sum of the length of edges on the path between the leaves corresponding to those two sequences. Call these the true-tree-induced distances. The estimated pairwise distances may differ from these true distances, so we desire a method that is robust to large errors. Atteson [5] showed that neighbor-joining will recover the correct tree if the largest error among all pairwise distance estimates is at most $1/2$ the length of the shortest edge in the true tree. Distances that are close in this way to the true-tree-induced distances are called "nearly-additive". The ratio of allowed disagreement from true-tree-induced distances to minimum edge length is called the reconstruction radius; neighbor-joining has a radius of $1/2$. This reconstruction radius has been shown to be optimal for distance methods [31].

The reconstruction radius indicates robustness in the face of distance estimation error, but seems to be of little practical value for moderate size real-world problems, where a near-zero-length branch is quite likely to be found somewhere in the true tree. A more valuable result from Mihaescu et al. [77] is the related notion of an edge radius, wherein neighbor-joining is guaranteed to correctly recover every edge with length at least 4x as large as the largest distance error. This is shown in the same work to be optimal. Thus, even though distance estimation error may cause neighbor-joining to fail in finding the correct tree, only short edges will be missed. (It should be noted that these short edges are not guaranteed to be missed, and are often correctly recovered despite the lack of guarantee [77]).

Gascuel [39] showed that formula 5.5 can be replaced with a formula that differentially weights the contributions of $d_{i|k}$ and $d_{j|k}$ to $d_{ij|k}$, while still retaining neighbor-joining's desirable properties. His approach is to assign weights that are dependent on sampling variance of distance estimates, such that the distance with higher variance (which is thus a less reliable estimate of the true distance) is down-weighted. He also shows that sampling variance can be reasonably approximated when distances are estimated from aligned sequences. This approach was implemented in a tool called `BioNJ`, which reportedly improved the accuracy of tree estimation from sequence data, on simulated data.

Bryant [13] showed that the selection criterion (formula 5.1) is the unique such criterion satisfying the requirements of being linear, unaffected by order of input, statistically consistent, and based solely on distance data.

5.3. Prior methods

`QuickTree` [54] is a very efficient implementation of the canonical neighbor-joining algorithm. Due to low data-structure overhead, it is able to compute trees up to nearly 40,000 sequences before running out of memory on a system with 4GB of RAM. `QuickJoin` [76, 75], `RapidNJ` [99], and the bucket-based method of [119] all produce correct neighbor-joining trees, reducing run time by finding the globally smallest q_{ij} without looking at the entire matrix in each iteration. While all methods still suffer from worst-case running time of $O(n^3)$, they offer substantial speed improvements in practice. Unfortunately, the memory overhead of the employed data structures reduces the number of sequences for which a tree can be computed. For example, on a system with 4GB RAM, `RapidNJ` scales to 13,000 sequences, and `QuickJoin` scales to 8000. No implementation of [119] was found.

The focus of this work is on exact neighbor-joining tools, but we briefly mention other methods for completeness. Alternate methods for phylogeny inference include parsimony, maximum-likelihood, Bayesian, and minimum evolution. Parsimony and maximum-likelihood (ML) use local-search heuristics to seek good trees, while the Bayesian method samples from the probability landscape. Bayesian methods (e.g.[92, 27, 107]) are unable to handle inputs on the scale of thousands of sequences. Recent advances in parsimony [43] and ML [102] methods have greatly improved the size of problems that can be handled by these methods, and the speed with which trees can be inferred. The largest published tree inferred by either of those methods contains 73,060 taxa [42], which reportedly took 2.5 months of processor time.

Relaxed [32] and fast [31] neighbor-joining are neighbor-joining heuristics that improve speed by choosing the pair to merge from an incomplete subset of all pairs. Both will return the same tree as neighbor-joining when distances are additive. Fast neighbor-joining shares the same reconstruction radius with exact neighbor-joining [77], and runs in time $O(n^2)$, but was observed to be slightly

less accurate than neighbor-joining [31], suggesting a lack of robustness when that radius is violated. An implementation of the relaxed neighbor joining heuristic, `ClearCut` [98], is nearly an order of magnitude faster than `NINJA` on very large inputs, but also shows reduced accuracy.

Minimum-evolution (ME) methods offer an alternate fast approach. Under the ME framework, edge lengths for a fixed topology are those that minimize the sum of the squares of the differences between the input pairwise distances and those induced by the tree's edge lengths. The minimum-evolution tree is the tree with topology minimizing the sum of edge lengths. Saitou and Nei [95] described their neighbor-joining method as working "under the principle of minimum evolution", and it has been proven to be a greedy heuristic for finding the balanced minimum evolution tree [40].

Minimum-evolution with the SPR [53] local-search improvement method has proven reconstruction radius of 1/3 [10], and consistency (though not a reconstruction radius > 0) has been conjectured for ME with the NNI [22] local-search improvement method. A recent implementation of a minimum-evolution heuristic with NNI, `FastTree` [90], is notable for constructing accurate trees on datasets of the scale discussed here, with speed more than 10-fold greater than that achieved by `NINJA` on very large inputs.

CHAPTER 6

DATA STRUCTURES AND ALGORITHMS FOR SCALING UP NEIGHBOR-JOINING

In this chapter, we describe a two-tiered filtering regime that dramatically reduces the number of distance values that are viewed during the search for the minimum q_{ij} value at each iteration, relative to a complete scan of the distance matrix. In addition, the method overcomes memory constraints seen in earlier filtering-based work by incorporating external-memory-efficient data structures into the algorithm.

6.1. Restricting search of the distance matrix

6.1.1. The d-filter

A valid filter must retain the standard neighbor-joining optimization criterion at each iteration: merge a pair $\{i, j\}$ with smallest q_{ij}. To avoid scanning the entire distance matrix D, the pairs can be organized in a way that makes it possible to view only a few values before reaching a bound that ensures that the smallest q_{ij} has been found.

To achieve this we use a bound that represents a slight improvement to that used in `RapidNJ` [99]. In that work, $(\{i, j\}, d_{ij})$ triples are grouped into sets, sorted in order of increasing d_{ij}, with one set for each cluster. Thus, when there are r remaining clusters, each cluster i has a related set S_i containing $r-1$ triples, storing the distances of i to all other clusters j, sorted by d_{ij}. Then, for each cluster i, S_i is scanned in order of increasing d_{ij}. The value of q_{ij} is calculated (equation 5.1) for each visited entry, and kept as $q_{\min}$ if it is the smallest yet seen.

To limit the number of triples viewed in each set, a second value is calculated for each visited triple, a lower bound on q-values among the unvisited triples in the current set S_i: $q_{\text{bound}} = (r-2)\, d_{ij} - t_i - t_{\max}$, where $t_{\max} = \max_k\{t_k\}$. In a single iteration, $t_{\max}$ is constant, and for a fixed set S_i, t_i is constant and the sorted d_{ij} values are by construction non-decreasing. Thus, if $q_{\text{bound}} \geq q_{\min}$, no unvisited entries in S_i can improve $q_{\min}$, and the scan is stopped. After this bounded scan of all sets, it is guaranteed that the correct $q_{\min}$ has been found. This is the approach of `RapidNJ`.

Improving the d-filter While this method is correct, and provides dramatic speed gains [99], it can be improved. First, observe that the bound is dependent on $t_{\max}$, which may be very loose (see fig. 7.2a). One way to provide tighter

bounds is to abandon the idea of creating one list per cluster. Instead, the interval $(t_{\min}, t_{\max})$ is divided into evenly spaced disjoint bins, where each bin B_x covers the interval $[T_x^{\min}, T_x^{\max})$. For X bins, then, the size of the interval between min and max values will be $(t_{\max} - t_{\min})/X$ (the default number of bins is 30). Each cluster i is associated with the bin B_x for which $T_x^{\min} \leq t_i < T_x^{\max}$. Adopt the notation that cluster i's bin $B(i) = x$. Note that bins may contain differing numbers of clusters. Then create a set $S_{\{x,y\}}$ for each bin-pair $\{B_x, B_y\}$.

Now, instead of placing $(\{i, j\}, d_{ij})$ triples into per-cluster sets as before, place them in per-bin-pair sets $S_{\{B(i),B(j)\}}$, still sorting triples within a set by increasing d_{ij}. To find $q_{\min}$, traverse the sets, scanning through each as before, but now calculating the bound based on current triple $(\{i, j\}, d_{ij})$, taken from set $S_{\{x,y\}}$, as

$$q_{\text{bound}} = (r - 2)\, d_{ij} - T_x^{\max} - T_y^{\max}. \tag{6.1}$$

This improves the filter because, for an unvisited pair $\{i', j'\}$ from the same set $S_{\{x,y\}}$, setting $\rho = (r - 2)\, d_{i'j'}$, $\rho - T_x^{\max} - T_y^{\max}$ will usually be a tighter bound on $q_{i'j'}$ than is $\rho - t_{i'} - t_{\max}$.

Updating data structures After merging clusters i and j, the rows and columns associated with those clusters are inactivated in D, and a new row and column are added for the merged cluster ij. Entries in the sets also require update. The new cluster, ij, is associated with the bin $B(ij) = \text{argmin}_x\{T_x^{\max} > t_{ij})\}$. Triples $(\{ij, k\}, d_{ij|k})$ for distances to each remaining cluster are added to the appropriate sets, $S_{\{B(ij),B(k)\}}$. Triples for the removed clusters i and j are removed from sets in a lazy fashion: i and j are marked as retired, and when triples involving either i or j are encountered while scanning sets in future iterations, they are removed.

While this method provides tighter q_{bound} values than the method of keeping one set per cluster, these bounds will tend to loosen over time. Before any merges are performed, the intervals of these sets are non-overlapping, but because the change in t_k after a merge may be different for each cluster k, this non-overlapping property is no longer guaranteed to hold after a merge is performed. The result is a loosening of the value $T_x^{\max}$ as a bound for t_i for an arbitrary cluster (i.e. the difference between the true value and the bound may be greater than $(t_{\max} - t_{\min})/X$). The loosening of the bound grows as iterations pass, though it is still tighter than the per-cluster bound until the set ranges overlap almost completely.

It may seem appealing to move a cluster to a new bin when that bin could provide a tighter bound, but doing so would incur substantial work to take all corresponding triples out of the various bin-pair sets. The strategy taken by `NINJA` is to occasionally rebuild the sets from scratch. For a constant $K > 1$ (the default is $K = 2$), the sets are rebuilt after r/K merges have been performed since the last rebuild, where r is the number of clusters remaining at the time of that prior rebuild. Overall runtime for these set constructions is dominated by the time of the

first construction, $O(n^2 \log n)$. This process of regularly rebuilding data structures resembles that used in [76].

6.1.2. Overcoming memory limits

The size of D is quadratic in the number of sequences, as is the size of the sets of triples described above. If these structures grow to exceed available RAM, an application may either abort or store the structures to *external storage* (i.e. disk). Data in RAM may be accessed in random patterns without concerns about speed, but reading and writing data on disks is done in large chunks (called disk blocks) of several kilobytes. The dramatic difference in latency between disk and RAM access (on the order of 10^6-fold difference [87]) means that it takes effectively the same amount of time to access a single value from disk as it takes to access all the values stored in a disk block. This necessitates I/O-efficient algorithms if external storage is to be used. We describe methods for efficiently handling both the sets $S_{\{x,y\}}$ and the distance matrix.

Bin-pair sets in external memory The set of triples associated with each bin pair set $S_{\{x,y\}}$ has been described as a sorted list. In fact, in order to allow fast insertion of triples for new clusters, such a list would likely be implemented as a data structure such as a binary search tree [66]. Binary search trees have poor I/O behavior when stored to disk, but could be easily replaced by a B-tree [8] or B+ tree, which allow for logarithmic number of disk I/Os for both insertions and reads.

However, since only a small portion of the entries in a set are accessed, the effort of keeping a totally ordered data structure is unnecessary. A min-heap [20] provides the tools necessary to scan through increasing d_{ij}, with less overhead since it only need keep a partial order. `NINJA` implements an *external memory array heap* [12], keyed on d_{ij}. This heap structure can store more triples than would fill a 1TB hard drive while maintaining a memory footprint smaller than 2MB, and guarantees an amortized number of I/O operations for *insert* and *extract-min* operations that is logarithmic in the number of inserted triples. One heap is used for each set $S_{\{x,y\}}$.

Distance matrix in external memory Though the heaps are used to identify the cluster pair $\{i, j\}$ to merge, the distance matrix D should still be maintained. After a merge, $d_{ij|k}$ is calculated for every cluster k. From equation 5.5, we see that we must view d_{ik} and d_{jk} for every k, which is more efficiently done by traversing the rows and columns for i and j in D than by scanning through the heaps.

Since neighbor-joining assumes a symmetric D, an efficient way to store D for in-memory use is to keep only its upper-right triangle: distances for cluster i are spread across row i and column i, such that all reside in the upper triangle. When a pair of clusters $\{i, j\}$ is merged, a new row and column are said to be added, but no additional space is actually required: the distances of the new cluster ij to all

remaining clusters k can be stored in the cells previously belonging to one of the retired clusters, say i, so $d_{ij|k}$ is stored in the cell where d_{ik} was stored. Clusters i and j are noted as retired, and the mapping of cluster ij's stored location is simple.

However, when D is stored to disk, this approach will lead to poor disk paging behavior, because values for cluster i are split between row i (which can be accessed efficiently from disk, with many consecutive values per disk block), and column i (which will be spread across the disk, with typically one value per disk block). Therefore, a modification is required. For an input of n sequences, a file F stores a matrix with $2n$ columns and n rows. The full initial D (i.e. both the upper and lower triangles) is stored to F, filling the first n columns of each row. When a merge is performed, and new distances are calculated, the values d_{ik} and d_{jk} can be gathered by sweeping through rows i and j, allowing the number of distance values that fit in a disk page to be gathered at the cost of a single disk access. The mapping for the storage location of the new $d_{ij|k}$ values will be different for rows and columns: if ij is formed as the result of the pth merge, then it will map to the row in F where i was stored, but will fill a new column $n+p-1$. Newly calculated distances are not immediately stored to disk, instead waiting until enough values have been calculated to allow for efficient disk I/O. Suppose b distance values fit in a disk block: then d_{ij}s for new clusters are appended to a b x n in-memory matrix M until all b columns of that matrix are full. At that time, each row of M is appended to the same row in F (requiring one disk I/O per row), and each column is translated and written into the mapped row in F (requiring up to $\lceil n/b \rceil$ I/Os).

6.2. Candidate handling

Due to the nature of heaps, all viewed $(\{i, j\}, d_{ij})$ triples are removed from their containing heaps during the search for $q_{\min}$; call these the *candidates*. The d-filter method described in Section 6.1 dramatically reduces the number of candidates viewed in most cases, but inputs with relationships like those seen in Figure 7.2a reduce the efficacy of d-filtering, for reasons described in Section 7.1. Examples of the impact on run time are given in Table 7.2c.

Below, a second level of filtering, called the *q-filter*, is described. It works by sequestering candidates passing the d-filter, and organizing them in a way that allows a new bound to limit the number of those candidates that are viewed in each iteration.

q-filter on a candidate heap Let $q_{ij}(p)$, $r(p)$, and $t_i(p)$ correspond to the values of q_{ij}, r, and t_i at a fixed previous iteration p. And let $\delta_i(p) = (r-2)t_i(p) - (r(p)-2)t_i$. Then it is easy to show that, for the current iteration,

$$q_{ij} \;=\; \frac{(r-2)\, q_{ij}(p) \;+ \delta_i(p) + \delta_j(p)}{r(p)-2}. \tag{6.2}$$

Suppose all candidates on hand at iteration p are stored as $(\{i,j\}, q_{ij}(p))$ triples in a candidate set, sorted according to their $q_{ij}(p)$ values. Assign the current r and t_i as $r(p)$ and $t_i(p)$ for that set. Since relative q-values change by small amounts from one iteration to the next, the $\{i,j\}$ pair with the smallest q_{ij} at a future iteration is likely to be near the front of this sorted list. It can be found by initializing $q_{\min}$ to ∞, then scanning candidates in order of increasing $q_{ij}(p)$, updating $q_{\min}$ when an entry with a smaller q_{ij} is found.

Let S be the set of all clusters with at least one representative in the candidate set, and define

$$\Delta_{\max}(p) := \max_{\substack{i,j \in S \\ i \neq j}} \{\delta_i(p) + \delta_j(p)\} \ . \tag{6.3}$$

Then scanning of this sorted list may be stopped when an element is found with

$$\frac{(r-2)\, q_{ij}(p) \ + \Delta_{\max}(p)}{r(p) - 2} \ \geq \ q_{\min} \ . \tag{6.4}$$

The candidate set can be large enough to exceed memory for very large inputs, and because only a partial order is required, `NINJA` stores the contents of the candidate set in an external memory heap array, as described for the d-bound bin-pairs in Section 6.1. The heap formed from such candidates is called a *candidate heap.*

Candidate heap chain Adding a new candidate to a candidate heap created in a previous iteration p_a (with associated $r(p_a)$ and $t_i(p_a)$ values) is problematic: (1) if the candidate involves a cluster j that was formed after p_a, then $q_{ij}(p_a)$ and $t_j(p_a)$ are undefined, and (2) even if both clusters existed before p_a, the candidate would need to be stored on the heap with a back-calculated $q_{ij}(p_a)$ (and thus looser than necessary bounds) to retain sensible δ-values.

`NINJA` overcomes this problem by keeping a chain of candidate heaps. At initiation, there are no candidates. In each iteration, newly gathered candidates from the d-filter are placed in a single candidate pool. When the size of that pool exceeds a threshold (default is 50,000; it should be fairly large because of the overhead required to form an external memory array heap), a candidate heap is created and populated with the triples in the pool, and the pool is then emptied. As more candidates are gathered, they are again stored in the pool, until it exceeds threshold, at which time a second candidate heap is formed, filled from the candidate pool, and linked to the first. This is repeated until the tree is complete. This results in a chain of candidate heaps. The chain is destroyed when bin-pair heaps are rebuilt (Section 6.1.1).

At each iteration, these heaps are scanned for elements with small q_{ij} by removing triples until the bound (6.4) is reached. Those viewed triples with $q_{ij} > q_{\min}$ are placed in the candidate pool, rather than being returned to their

source candidate heap, because the δ-bound usually gets looser, so they'd almost always just be pulled back off their original heap on the next iteration. When a candidate heap drops below a certain size (default = 60% of original size), it is liquidated, and all triples placed in the candidate pool.

6.3. Algorithm overview

At each iteration p_a, `NINJA` follows this process, tracking $q_{\min}$ at each step:

1. Scan all candidates in the pool, keeping the one with smallest q_{ij}.
2. Sweep through the candidate heap chain, for each heap removing triples until reaching the bound (6.4), and placing those triples in the candidate pool. Possibly liquidate heaps in the chain if they become too empty.
3. Sweep through the bin-pair heaps, for each heap removing triples until reaching bound (6.1), and placing those triples in the candidate pool.
4. If the size of the candidate pool exceeds threshold, move all candidates into a new heap, storing $q_{ij}(p_a)$ for each candidate, and $t_i(p_a)$-values and $r(p_a)$ for the heap. Append this heap to the candidate heap chain.
5. Having found the q_{ij} with minimum value, merge clusters i and j, update the bin-pair heaps and the in-memory part of the distance matrix M with entries for new cluster ij, and possibly write out to the on-disk distance matrix D. Also occasionally liquidate the candidate heap chain and rebuild the bin-pair heaps (see Section 6.1.1).

Steps 1 and 2 typically identify a pair with good q_{ij} value, because they start with a set of previously filtered candidates. This improves the efficiency of the bin-heap search, since the d-filter bound will be more effective if the starting $q_{\min}$ is close to the true $q_{\min}$.

CHAPTER 7

EXPERIMENTAL VERIFICATION OF METHODS FOR SCALING UP NEIGHBOR-JOINING

To assess the effectiveness of the two-tiered filtering algorithm, it has been implemented in an application called `NINJA`. Three variants were used in various tests in the results shown below. The default variant, `NINJA`, stores the distance matrix on disk, and uses both the d-filter described in Section 6.1 and the q-filter described in Section 6.2, both implemented with external-memory array heaps [12]. The variant labeled `NINJA-d-filter` is identical to `NINJA`, except that it implements only the d-filter, not the q-filter. The variant labeled `NINJA-InMem` also uses only the d-filter, but does so with in-memory data structures - keeping the distance matrix entirely in memory, and using a binary heap in place of the external-memory array heap. `NINJA-InMem` makes it possible to directly assess the impact of external-memory components of the algorithm. On a machine with 4GB RAM, `NINJA-InMem` is only able to compute neighbor-joining trees on inputs of fewer than about 7000 sequences, due to overhead memory use.

For comparison purposes, we tested two tools that similarly avoid viewing the entire distance matrix at each iteration, `QuickJoin` and `RapidNJ`, and a very fast implementation of the canonical algorithm, `QuickTree`. To our knowledge, these are the fastest available tools that implement exact neighbor-joining. Both of the former tools are unable to handle inputs of more than 13,000 sequences on a machine with 4GB of RAM, but an experimental external-memory version of `RapidNJ`, called `RapidDiskNJ`, has been released. A version of `RapidDiskNJ` downloaded on 04/24/09 was used as a reference for large inputs. Note that the filter used in `RapidNJ` and `RapidDiskNJ` is almost equivalent to that used in `NINJA-d-filter` and `NINJA-InMem`. This analysis considers only tools that produce exact neighbor-joining trees. It is worth noting that `ClearCut` and `FastTree`, both of which implement neighbor-joining heuristics, are both faster than `NINJA`.

`QuickTree` was implemented in *C*, `QuickJoin` and both `RapidNJ` variants were implemented in *C++*, and the `NINJA` variants were implemented in *Java* (see appendix B).

Environment Experiments were run on a bank of 8 identical dedicated systems running CentOS 4.5 (kernel 2.6.9-55), with 64 bit 2.33 GHz Xeon processors, 4 GB allocated RAM, and 500 GB 7200 RPM SATA hard drives. `NINJA` used roughly 60 GB of disk space for the largest inputs. The "real time" output from the standard `time` tool was used to measure run time.

Data `Pfam` [37] families were used as sample input for the tools. Each protein domain family was preprocessed to remove duplicate sequences, and all 415 families with more than 2000 unique sequences were used. Distances calculated with `QuickTree`, with Kimura's correction [65] for multiple substitutions at a locus, were used as input to all tools.

7.1. Experiments

Effect of filters At each iteration, the canonical algorithm scans all $r(r-1)/2$ cells in the distance matrix, where r is the number of remaining clusters. It thus views $\Theta(n^3)$ cells (candidates) over the course of building a complete neighbor-joining tree on n sequences. Figure 7.1 shows the often dramatic reduction in number of candidates passing the d-filter, relative to this total count of cells. It also highlights instances where the d-filter is mostly ineffective, and shows that more consistent success is achieved when the q-filter is used in conjunction with the d-filter. It is important to note that Figures 7.1, 7.3, and 7.4 are all log-log plots. Thus, the roughly linear growth observed in all plots corresponds to polynomial growth of both candidates and run time, with the polynomial exponent visible in the log-log slope.

Inputs causing bad d-filtration Figure 7.2a shows an example of the kind of input that makes the d-filter fairly ineffective. It contains large clusters of very closely related sequences, and a few relatively long branches. Contrast this to the more evenly-distributed sequences seen in Figure 7.2b, for which the d-filter is quite effective.

The reason for the computational difficulty of trees like the one in Figure 7.2a is that the clusters on the very long branches have very large t values relative to the t values for most clusters, while the clusters for the tree in Figure 7.2b will all have fairly similar t values. With `RapidNJ`'s bound, which depends on $t_{\max}$, d-filtering on 7.2a is immediately inefficient because of this t-value discrepancy. `NINJA`'s bound (equation 6.1) starts off relatively tight, but the d-filtering becomes inefficient as the range of t-values within a bin grows. This can happen dramatically when clusters along one long branch, which thus begin in a high t-value bin, are merged (with corresponding relative reduction in t values), while other clusters sharing the same bin are not merged, and thus retain relatively high t values.

Table 7.2c shows the effect that these differing tree forms have on both number of viewed candidates and run time. Focus on the results for family Cytochrom_B_N, an input with structure like that shown in Figure 7.2a: d-filtering only reduces the number of candidates by a factor of 10, much less effective filtering than the 10,000-fold reduction seen in family WD40, an input with structure much like that seen in Figure 7.2b. Because of the extra overhead of their algorithms, the resulting

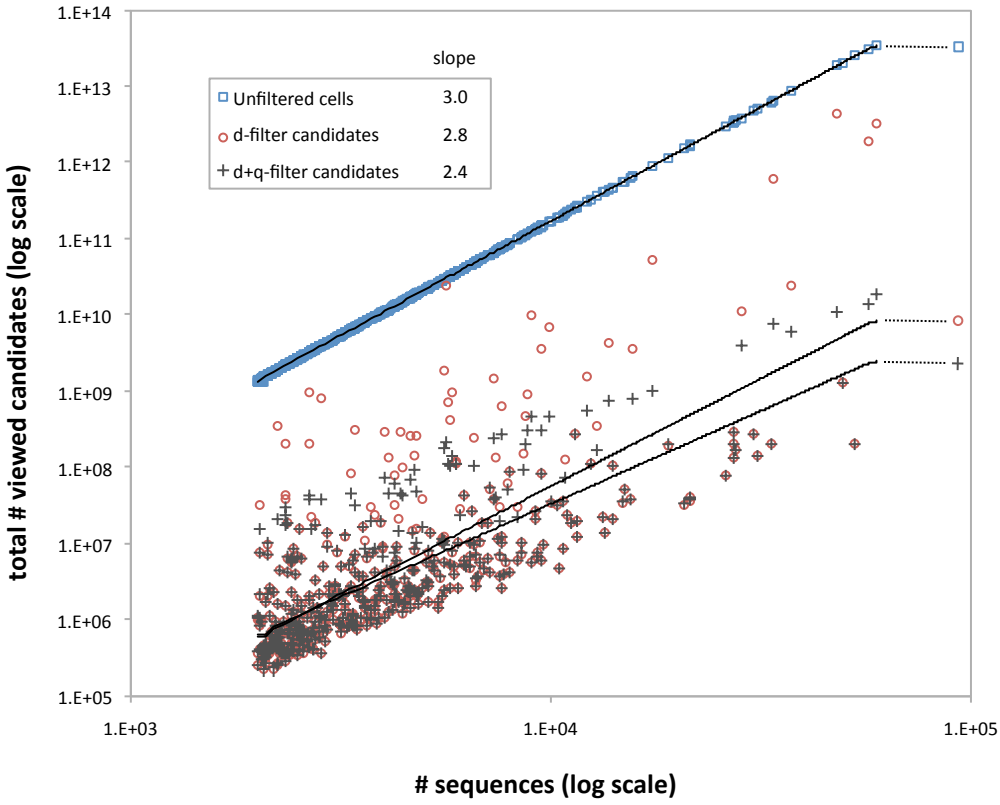
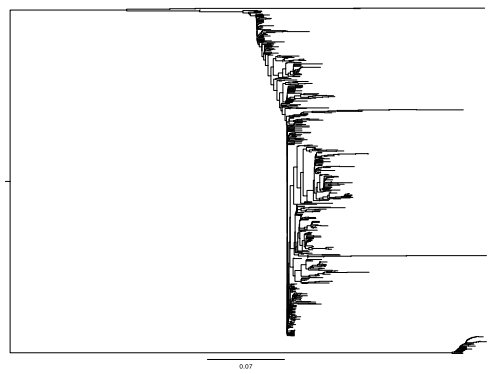
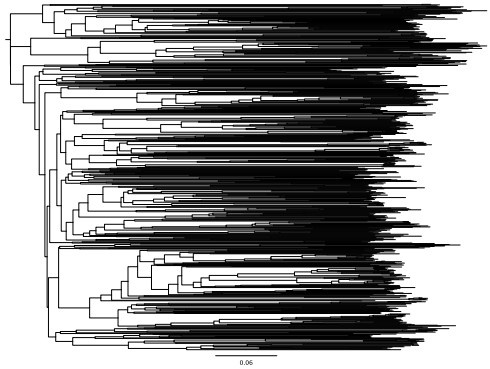


Figure 7.1: Number of candidates viewed during tree-building with and without filters, for all 415 Pfam alignments with more than 2000 non-duplicate sequences. Data points are placed on a log-log plot: the slope on such a plot gives the exponent of growth. The canonical algorithm treats all cells as candidates, and the corresponding count of unfiltered cells shows the expected slope of 3 for $\Theta(n^3)$ number of cells. The d-filter often reduces the number of viewed candidates by more than 3 orders of magnitude, but is less effective for some inputs. Addition of the q-filter results in more consistent filtering success across all inputs, and an observed growth rate in number of viewed cells of roughly $O(n^{2.4})$.



(a) Flu_M1, 707 sequences. Example of a topology for which filtering is ineffective.



(b) QRPTase_N, 707 sequences. A topology for which filtering is effective.

		Number candidates			Run time (minutes)			
Pfam ID	Sequence number	All cells	d-filter only	d+q filters	QuickTree	RapidDiskNJ	NINJA d-filter	NINJA d+q filters
RuBisCO_large	17,490	9E+11	5E+10	1E+09	64	124	331	25
PPR	18,961	1E+12	2E+08	2E+08	155	20	21	20
Cytochrom_B_N	33,789	6E+12	6E+11	7E+09	539	1,678	6,092	146
WD40	33,327	6E+12	2E+08	2E+08	756	52	121	110
RVT_1	56,822	3E+13	2E+12	1E+10	n/a	>18,500	>18,500	717
ABC_tran	53,116	2E+13	2E+08	2E+08	n/a	159	554	530

(c) Impact of d- and q-filtering at various input sizes.

Figure 7.2: Trees (a) and (b) are both of 707 sequences, and represent approximately equal evolutionary distance between the most divergent pair of sequences. They are mid-point-rooted, and images were created using `FigTree` (http://tree.bio.ed.ac.uk/software/figtree/). `Pfam` datasets with relationships like those shown in (a), with many closely related sequences and a few relatively long branches, cause d-filtering to be ineffective. In Table (c), the first of each pair has topology similar to (a), and shows poor d-filtering; the second has topology similar to (b), and shows good d-filtering. Run times for `RapidNJ` and `NINJA-d-filter` are very slow when d-filtering is ineffective, but the additional q-filter used by `NINJA` results in much better filtering, and therefore improves runtimes even for these hard cases. On a system with 4GB RAM, `QuickTree` crashes on all `Pfam` families with more than 37,000 sequences. `RapidDiskNJ` and `NINJA-d-filter` both took longer than 13 days to compute a tree for RVT_1.

run times for both RapidDiskNJ and NINJA-d-filter are much worse than that of QuickTree. By applying the q-filter, NINJA achieves a further 100-fold reduction in candidates viewed for Cytochrom_B_N, along with a large reduction in run time.

Comparison to other tools Figures 7.3 and 7.4 compare NINJA variants to other neighbor-joining tools. They focus on inputs of more than 2000 sequences, since smaller inputs are solved by the canonical algorithm (implemented in QuickTree) in under 10 seconds.

The orders-of-magnitude reduction in viewed candidates seen in Figure 7.1 does not translate to a similar reduction in run-time because the underlying data structures required to gain this filtering advantage incur a great deal of overhead relative to the simple scanning of a matrix. In addition, the large-scale applications (NINJA and RapidDiskNJ) incur a constant-factor overhead from disk accesses. Those factors are mitigated by using algorithms with good disk-paging behavior, but are nevertheless present.

Figure 7.3 shows run times for a random sample of 113 medium-sized (2000-7000 sequences) inputs from Pfam. A sample is shown, rather than the entire dataset, to improve visibility of the chart, and agrees with trends for the full set of similarly-sized inputs. Note that QuickTree's run time grows with a slope of 2.9 on a log-log plot, essentially what is expected of a $\Theta(n^3)$ algorithm. QuickJoin and RapidNJ are in-memory versions of competitor algorithms - both show a reduction in run-time, and a growth rate that is slightly more than quadratic. This is in agreement with results from [99]. Results for NINJA-InMem and NINJA are presented to show their relative performance to each other and the other tools. Both show a roughly quadratic run-time growth on this data set. NINJA-InMem is slightly faster than the fastest other tool, RapidNJ. Since the two tools use essentially the same bounding method for their d-filter methods, this difference is likely explained by the tighter bounds generated by the bin-pair approach of NINJA.

Figure 7.4 shows run times for all inputs from Pfam with more than 7000 sequences. Results are given for the variant of each tool that best handles these large inputs: QuickTree, RapidDiskNJ, and NINJA. Only NINJA successfully computed neighbor-joining trees for all inputs; QuickTree crashed on all inputs with more than 37,000 sequences, while RapidDiskNJ failed to complete within 13 days on the two largest inputs. QuickTree continues to exhibit the expected slope (3.0) on a log-log plot for a $O(n^3)$ algorithm. Interestingly, both RapidDiskNJ and NINJA also show a similar cubic slope for these larger inputs, in conflict with the lower rate of growth observed for smaller inputs in Figure 7.3 and [99]. Inspection of the data suggests that this is due to an increased frequency in these larger datasets of the sort of difficult inputs characterized by Figure 7.2a. Note that the number of viewed candidates was observed in Figure 7.1 as growing with a power of 2.4. The

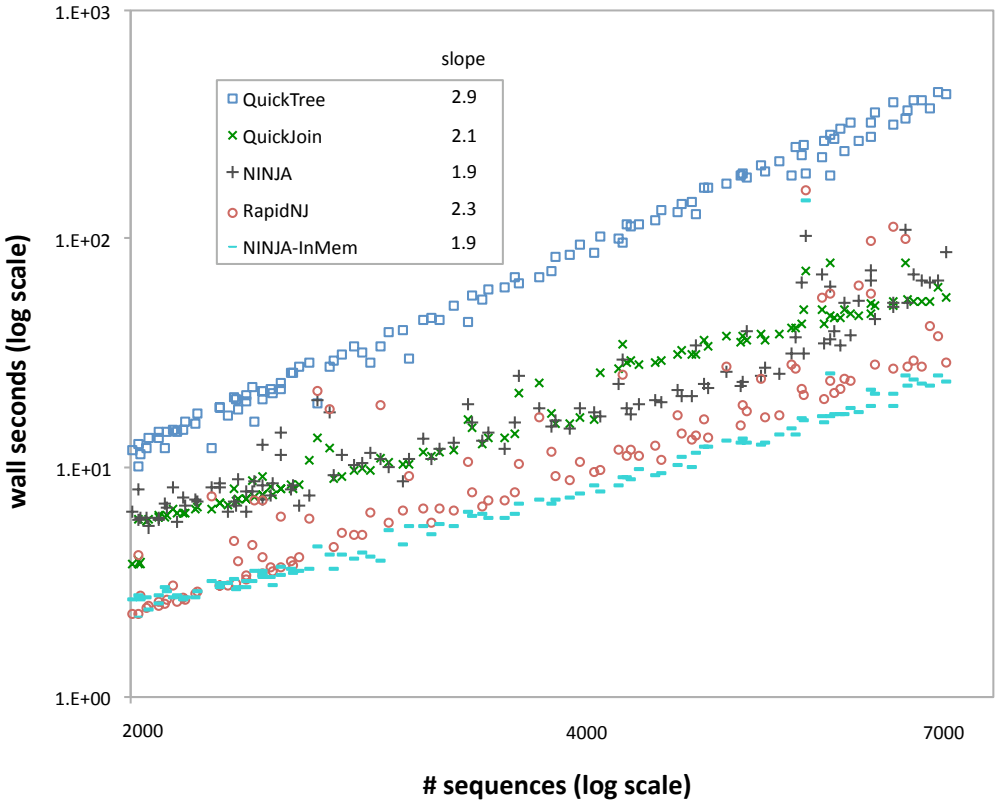


Figure 7.3: Performance of `NINJA` and `NINJA-InMem` compared to that of `QuickTree`, `QuickJoin`, and `RapidNJ` on a random sample of 113 medium-sized (2000 to 7000 sequences) `Pfam` inputs.

---

logarithmic overhead of heap data structures is responsible for the observation that run time grows faster than the number of candidates.

While the inputs from `Pfam` are large, the largest contains fewer than 60,000 unique sequences. To assess `NINJA`'s ability to build trees on truly large datasets, the GreenGenes collection of 218,348 16S ribosomal RNAs was used as input (http://www.microbesonline.org/fasttree/downloads/Large16S.tar.gz). On a system with the same specs as earlier, but with 12GB allocated RAM, `NINJA` computed the tree in 134 hours (5.6 days). Interestingly, this represents a roughly quadratic growth in run time over the 10 hours required to build a tree for almost 60,000 sequences. It is estimated that `QuickTree` would take more than 80 days to compute the same tree, if a machine with sufficient memory (more than 500GB RAM) were available.

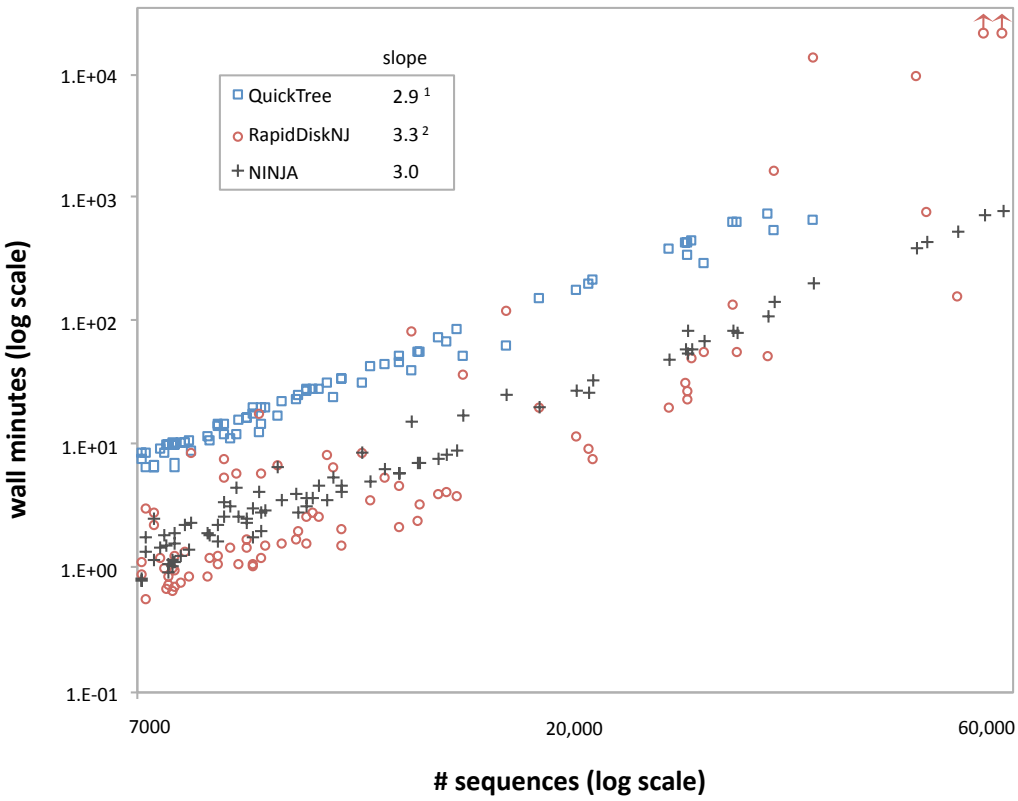


Figure 7.4: Performance of `NINJA` compared to that of `QuickTree` and `RapidDiskNJ` on all large (7000 to 60,000 sequences) `Pfam` inputs. (1) On a system with 4GB RAM, `QuickTree` crashes on inputs with more than 37,000 sequences. (2) `RapidDiskNJ` failed to complete within 13 days for the two largest inputs; the uncertain times-to-completion are represented with the arrowed circles in the upper right corner. The slope for `RapidDiskNJ`, which shows that its run time is growing faster than n^3, does not include these two points.

7.2. Discussion

We have presented a new tool, NINJA, that builds a tree under the traditional optimization criteria of neighbor-joining, with the associated guarantee of statistical consistency and optimal edge radius. NINJA speeds up neighbor-joining by employing a two-tiered filtering regime, which greatly reduces the number of viewed candidates in each iteration relative to the complete scan of the distance matrix that is employed in the canonical algorithm. NINJA also overcomes memory constraints seen in earlier filtering-based work by incorporating external-memory-efficient data structures into the algorithm, specifically the external memory array heap [12] and simple on-disk storage of the distance matrix. The latter structure can be trivially co-opted by any neighbor-joining tool to overcome memory constraints due to the size of the distance matrix.

Run time growth rate is somewhat unclear. The results of Figure 7.4 suggest a cubic rate of growth, which represents no asymptotic improvement over the canonical algorithm. However, the single huge 16S RNA tree does not follow this trend. Further examination, perhaps on simulated alignments, might clear this up. In any case, NINJA represents a major advance in the scalability of neighbor-joining, and makes it feasible to construct trees for inputs with well over 100,000 sequences in a matter of a small number of days of computation on a modern desktop.

The accuracy of NINJA is not discussed in this paper, as accuracy of any exact neighbor-joining tool is expected to be the same. That said, there is a clear dependency on correctly estimated distances, motivating more attention to this topic. BioNJ [39] implements a method of minimizing the variance of distances as the neighbor-joining tree is formed, with demonstrated accuracy benefits relative to canonical neighbor-joining; it is a straightforward exercise to incorporate these methods into NINJAś algorithm. In addition, a number of methods have been suggested for improving distance estimation, including Bayesian inference of evolutionary rates [82], and simultaneous estimation of all pairwise distances under a likelihood framework [108].

CHAPTER 8

FUTURE WORK AND CONCLUSIONS

The focus of this dissertation has been spread over two classic, and closely related, topics in molecular sequence analysis: multiple sequence alignment and phylogeny inference. I have explored a variety of approaches applied to the various stages of the form-and-polish heuristic for aligning multiple sequences, and described an approach that dramatically increases the speed and scalability of the neighbor-joining method for phylogeny inference.

Next I summarize my work and end by suggesting a few related research questions.

8.1. Summary

Part 1 (Chapters 2 through 4) of this work investigated methods for the phases of the fom-and-polish method of sequence alignment, while Part 2 (Chapters 6 through 8) described a new algorithm for the neighbor-joining method of phylogeny inference.

Chapter 2 introduced the standard form-and-polish heuristic for multiple sequence alignment, delineating what we call the stages of that method. A profusion of approaches have been applied in the literature to those various stages, with little discussion of their relative efficacy, and that chapter presented a careful study of methods for each stage, identifying best-of-breed methods for each. The methods investigated included both established and novel ones, including new methods for estimating distances, weighting pairs of sequences, polishing the alignment, employing alignment consistency, and choosing parameters. We showed that the largest gains in quality come from new methods for estimating distances by normalized alignment costs, and polishing by 3-cuts on the merge tree. We also found that the exact gap-count algorithm for aligning alignments [63] yields only a minor improvement over an approximate merging heuristic. The outcome of this investigation is an alignment tool called `Opal` which, with a careful choice of methods for each stage, achieves the same accuracy as state-of-the-art tools like `ProbCons` and `MAFFT` that in addition use various forms of consistency.

Chapter 3 presented details of a new method for assigning weights to pairs of sequences. The purpose of devising sequence pair weights is to reduce the undue influence of overrepresented groups on the final outcome of an alignment; without weighting, the sum-of-pairs scoring function may lead to small improvements to the quality of alignments involving overrepresented groups while greatly reducing the quality of alignments involving underrepresented groups. This new method

avoids anomalous behavior exhibited by two other standard methods. Also, despite conflicting results in the literature, we found that the tested methods of pair-weighting provided no significant benefit to alignment accuracy across benchmarks (though our new method gave encouraging results on a small dataset expected to suffer from overrepresented groups).

Chapter 4 gave details of a new method for using alignment consistency to avoid the errors that often occur in the early stages of the progressive alignment method. The approach depends on modifying the function used to score an alignment generated at a node of the alignment merge tree. The function is modified in a way that reflects the support given for particular alignment features in a pair of sequences A and B by other sequences C. Our results show that our new consistency model works well, but does not provide accuracy gains when used in conjunction with polishing, because of conflicts between the local application of consistency-based modified costs and polishing based on un-modified costs.

The canonical algorithm for neighbor-joining was given in Chapter 5, and Chapter 6 developed a new algorithm that finds a exact neighbor-joining tree in much less time, while overcoming previous memory-requirements. The method depends on a two-tiered filtering process, which dramatically restricts the number of distance values that are viewed at each iteration of the neighbor-joining method. The filters depend on external-memory-efficient priority queues, which, in conjunction with efficient on-disk storage of the distance matrix, allow the algorithm to handle inputs of effectively unlimited size (bounded only by disk size). Chapter 7 presents the results of experiments showing that an implementation of this algorithm reduces run time by roughly an order of magnitude on benchmark protein alignments. Specifically, the implementation is able to compute a tree for over 200,000 sequences in less than 6 days, while the canonical algorithm is predicted to take more than 80 days if a machine with sufficient RAM (roughly 500GB) were available.

8.2. Future directions

The work described here suggests a number of attractive directions of further research.

Sequence pair weights that permit fast profile alignment

In Section 2.6 and Chapter 3, we described a new method for weighting sequences pairs to overcome the bias induced by overrepresented groups. While the method avoids the anomalous behavior of both division- and covariance-weights, and gives encouraging results on a small sample of alignments likely experiencing overrepresented groups, it can not be used by optimized profile alignment subroutines. It would be of value to modify the weighting method to allow application to fast profile alignment methods.

Unbiased measure of alignment accuracy

We noted in Section 2.6 that assessment of sequence pair weighting methods is inherently difficult, as the unweighted recovery measure is itself biased. Devising an *unbiased recovery measure* that takes overrepresentation into account appears to be a challenging but valuable challenge.

Alignment parameter advisor

Our experiments (Section 2.9) with an oracle for choosing alignment parameter values suggest that large gains in recovery may be possible by an input-dependent choice of parameters. We reported modest success with both our naive Bayes classifier and core-column-count advisors, but a great deal of room for advancement remains. One interesting direction to take this work is into the realm of probabilistic alignment (such as `ProbCons`). A full-blown expectation-maximization approach would be too slow in practice, but an advisor that selects from a finite number of candidate parameterizations might offer improved average alignment quality.

Local alignments in suboptimal-alignment-based consistency

Much of the success of the consistency methods of `T-Coffee` and `MAFFT` comes from their inclusion of local alignment information into the consistency framework. In both tools, if local alignment support is removed from the consistency modification method, accuracy values are several percent worse (result not shown). `ProbCons` and `Opal` do not incorporate local alignments, but perform very well because of their use of suboptimal alignments in estimating support for alignment features. Merging the use of local alignments and posterior alignment probabilities seems like a natural step in the advancement of consistency-based sequence alignment.

Speeding up suboptimality-based consistency to permit polishing

Chapter 4 showed that polishing and consistency failed to work well together in the scheme tested in `Opal`, because polishing using a different scoring scheme than consistency, altering the alignment in ways that eliminated the gains made by consistency. Consistency-modifed costs weren't used in that scheme because computing those costs was too slow to be used on more than tens of sequences. Algorithm engineering to substantially speed up suboptimality-based consistency might allow it to be applied along the full tree, and in conjunction with polishing. Results in `MAFFT` and `ProbCons` suggest that this will produce improved alignment accuracy.

Probabilistic consistency with posterior probabilities on gaps

The probabilistic consistency method of `ProbCons` depends on posterior probabilities for only substitutions, but there is no reason posteriors can't be computed for other transitions in a pair-HMM. By computing these posteriors, and applying the constrained subpaths in Chapter 4 and appendix A, it may be possible to gain improved results from the probabilistic consistency framework.

Faster methods for optimizing branch length for minimum evolution

The minimum evolution (ME) framework (described in Chapter 5) assigns edge lengths to a tree that minimize the squared difference between observed pairwise sequence distances and those induced by the tree. While a fair bit of work has gone into developing fast methods for computing least-squares edge lengths [14, 68], there may be room for improvement. An algorithm is developed [14] for computing ordinary least squares (OLS) in $O(k^2)$ time for k leaves, but other variants WLS and GLS (which weight the components of the sun-of-squares based on variance and covariance of distance estimates) run in time $O(k^3)$ and $O(k^4)$ respectively. Speeding up WLS and GLS branch length computation may be value since it appears that ME local search may improve phylogeny recovery, relative to an initial neighbor-joining tree (Morgan Price, personal communication).

Sequence pair weighting applied to WLS branch lengths

Desper and Gascuel have presented very good phylogeny inference accuracy in results based on a variant of minimum evolution called *balanced minimum evolution* (BME) [22, 23]. At its heart, the approach depends on a computation of the average distance between two subtrees in which, for two subtrees A and B, where $B = B_1 \bigcup B_2$, the average distance between A and B is defined recursively, in the general case being

$$D(A, B) = \frac{1}{2} \left(D(A, B_1) + D(A, B_2) \right).$$

The result is to avoid undue influence on weights by overrepresented groups. The equation is dependent only on topology, and is completely ignorant of branch lengths. As was shown in Chapter 3, specifically in Figure 3.7, identical topologies can correspond to wildly different levels of sequence independence. Replacing the topology-only equation with one based on our sequence weighting scheme might result in an improvement in the accuracy of phylogenies computed in the BME framework.

Clearly there are many potentially fruitful directions in which the work of this dissertation can be extended. I look forward to exploring them.

APPENDIX A

CONSISTENCY GRAPHS

In this appendix, an exhaustive listing is given of the sub-paths in alignment graphs for $(A \sim C)$ and $(B \sim C)$ that are consistent with each feature in the alignment graph of $(A \sim B)$ (see Figure 4.1). This appendix extends the description in Section 4.2, and agrees with that definition of consistency, in that it lists alignment subpaths that *constrain* particular features in the alignment of sequences A and B.

A.0.1. Subpath pairs consistent with substitution

Only one subpath pair is consistent with a substitution column $(a_i \sim b_j)$ (Figure 4.1a). That subpath pair is shown in Figure 4.2, and given again here.

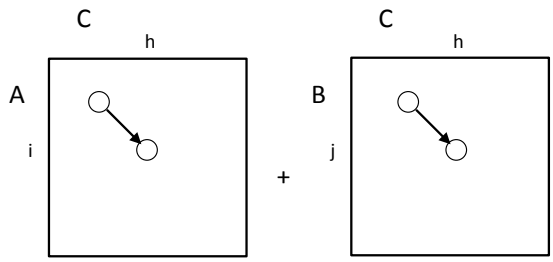


Figure A.1: Alignment subpaths of $A \sim C$ and $B \sim C$ that are consistent with $(a_i \sim b_j)$. The alignment that agrees with these subpaths is:

$$\begin{array}{ccc} & a_i & \\ \cdots & c_h & \cdots \\ & b_j & \end{array}$$

A.0.2. Subpath pairs consistent with gap extension

Figures A.2 through A.5 present four subpath pairs that are consistent with a_i extending a gap between b_j and b_{j+1}. This is a vertical edge in the pairwise alignment graph of $(A \sim B)$ (Figure 4.1b). Creating subpath pairs for horizontal gap extension is a trivial matter of flipping the graphs on a diagonal axis.

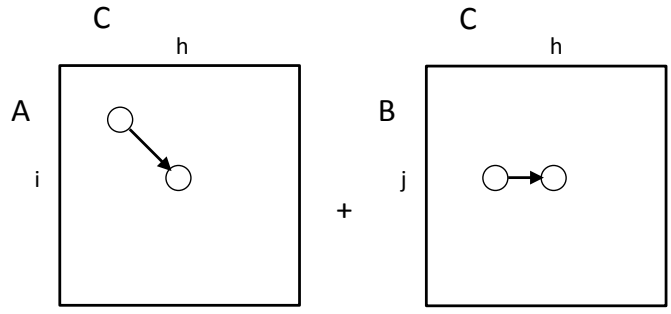


Figure A.2: Alignment subpaths of $(A \sim C)$ and $(B \sim C)$ that are consistent with a_i extending a gap between b_j and b_{j+1}. The alignment that agrees with these subpaths is:

$$\begin{array}{ccc} & \circ\ \mathtt{a_i} & \\ \cdots\ \circ\ \mathtt{c_h} & \cdots & \\ \mathtt{b_j}\ - & & \end{array}$$

(where $\circ$ represents freedom to either include or not include character from the associated sequence)

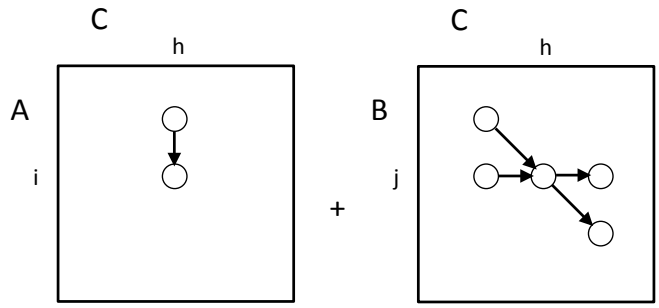


Figure A.3: Alignment subpaths of $A \sim C$ and $B \sim C$ that are consistent with a_i extending a gap between b_j and b_{j+1}. The alignment that agrees with these subpaths is:

$$\begin{array}{ccccc} & & \circ & \mathtt{a_i} & \circ & & \\ \cdots & \mathtt{c_h} & - & \mathtt{c_{h+1}} & \cdots \\ & & \circ & & \circ & & \\ & \nwarrow & \uparrow & & \uparrow & \nearrow & \\ & & \mathtt{b_j} & & \mathtt{b_{j+1}} & & \end{array}$$

The arrows indicate that the subpaths correspond to alignments of B with C such that b_i aligns either with or before c_h, and b_{i+1} aligns either with or after c_{h+1}.

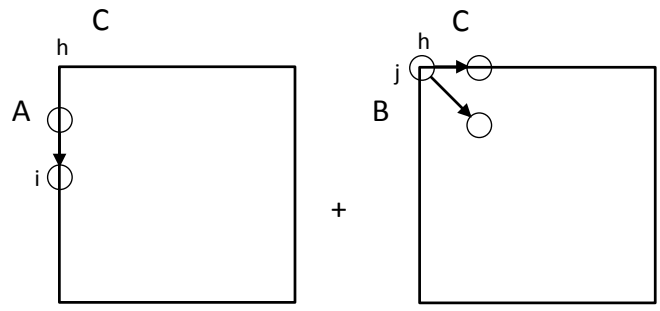


Figure A.4: Alignment subpaths of $(A \sim C)$ and $(B \sim C)$ that are consistent with a_i extending a terminal gap before the first character of B. The alignment that agrees with these subpaths is:

$$\begin{array}{ccccc} & \mathtt{a_i} & & \circ & \\ \cdots & - & \cdots & \mathtt{c_1} & \cdots \\ & & & \circ & \\ & & & \uparrow & \nearrow \\ & & & \mathtt{b_1} & \end{array}$$

The arrows indicate that the subpaths correspond to alignments of B with C such that b_1 aligns either with or after c_1, and c_1 aligns after a_i.

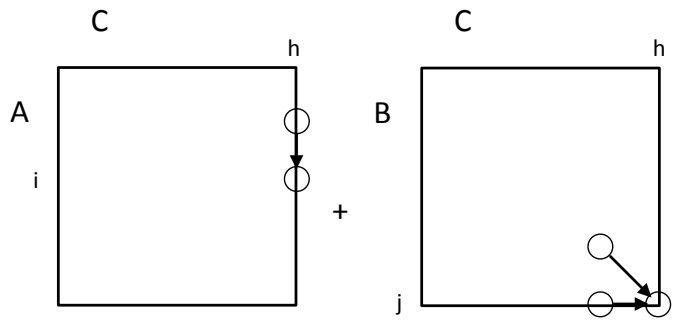


Figure A.5: Alignment subpaths of $(A \sim C)$ and $(B \sim C)$ that are consistent with a_i extending a terminal gap after the last character of B. The alignment that agrees with these subpaths is:

$$\begin{array}{ccccc} & \circ & & \mathtt{a_i} & \\ \cdots & \mathtt{c_p} & \cdots & - & \cdots \\ & \circ & & & \\ \nwarrow & \uparrow & & & \\ & \mathtt{b_n} & & & \end{array}$$

The arrows indicate that the subpaths correspond to alignments of B with C such that b_n aligns either with or before c_p, and c_p aligns before a_i.

A.0.3. Subpath pairs consistent with gap-open and gap-close.

Figures A.6 through A.13 present eight subpath pairs that are consistent with a_i opening a gap immediately after b_j. This is a vertical gap open in the pairwise alignment graph of $(A \sim B)$. Figures A.14 through A.21 presnt another eight subpath pairs that are consistent with vertical gap closure. Creating subpath pairs for horizontal gap boundaries is a trivial matter of flipping the graphs on a diagonal axis.

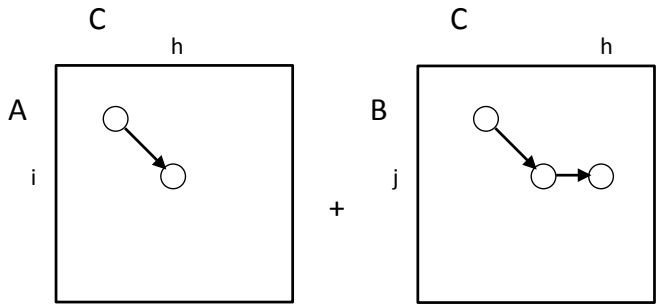


Figure A.6: Alignment subpaths of $(A \sim C)$ and $(B \sim C)$ that are consistent with a_i **opening** a gap immediately after b_j (general case). The alignment that agrees with these subpaths is:

$$\begin{array}{ccccc} & & \circ & a_i & \\ \cdots & c_{h-1} & c_h & \cdots \\ & & b_j & - & \end{array}$$

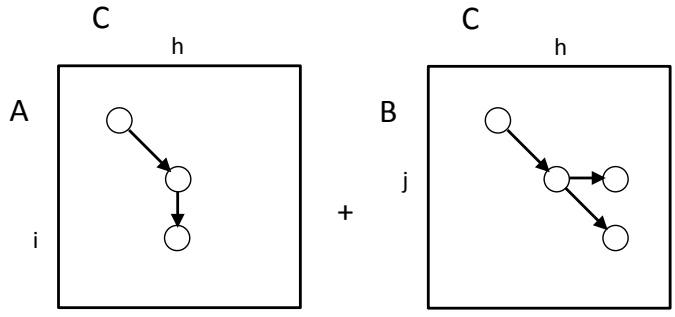


Figure A.7: Alignment subpaths of $(A \sim C)$ and $(B \sim C)$ that are consistent with a_i **opening** a gap immediately after b_j (case: $i > 1$, $0 < j < n$, $0 < h < p$). The alignment that agrees with these subpaths is:

$$\begin{array}{ccccccc} & & \mathtt{a_{i-1}} & \mathtt{a_i} & & & \\ \cdots & & \mathtt{c_h} & - & \cdots & \mathtt{c_{h+1}} & \cdots \\ & & \mathtt{b_j} & & & \circ & \\ & & & & & \uparrow & \nearrow \\ & & & & & \mathtt{b_{j+1}} & \end{array}$$

The arrows indicate that the subpaths correspond to alignments of B with C such that b_{j+1} aligns either with or after c_{h+1}.

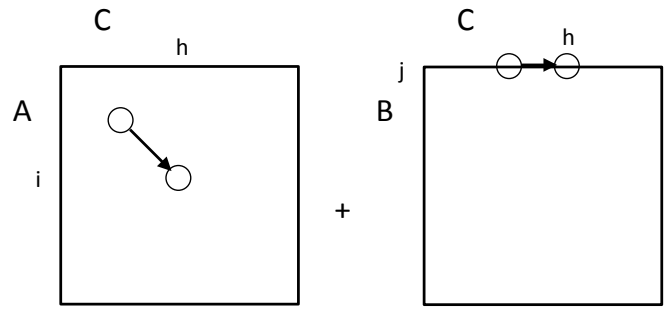


Figure A.8: Alignment subpaths of $(A \sim C)$ and $(B \sim C)$ that are consistent with a_1 **opening** a gap before the first character of B (case: $j = 0$, $i = 1$). The alignment that agrees with these subpaths is:

$$\begin{array}{ccccc} & & \mathtt{a_i} & & \\ \cdots & \mathtt{c_h} & \cdots & \circ & \cdots \\ & & & \mathtt{b_1} & \end{array}$$

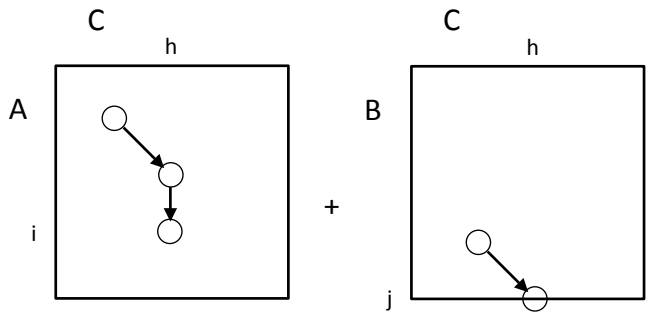


Figure A.9: Alignment subpaths of $(A \sim C)$ and $(B \sim C)$ that are consistent with a_i **opening** a gap immediately after b_n (case: $j = n$, $0 < h \leq p$). The alignment that agrees with these subpaths is:

$$\begin{array}{ccccc} & & \mathtt{a_{i-1}} & \mathtt{a_i} & \\ \cdots & & \mathtt{c_h} & - & \cdots \\ & & \mathtt{b_n} & & \end{array}$$

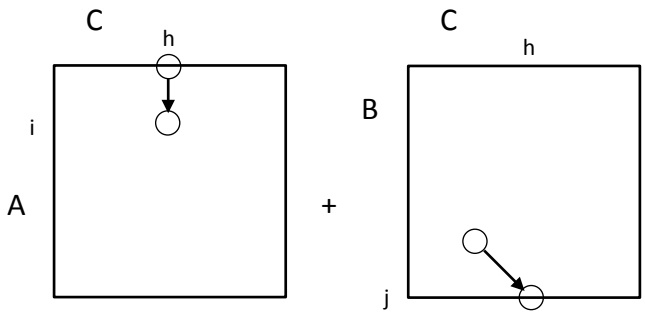


Figure A.10: Alignment subpaths of $(A \sim C)$ and $(B \sim C)$ that are consistent with a_1 **opening** a gap immediately after b_n (case: $i = 1$, $j = n$, $0 < h \leq p$). The alignment that agrees with these subpaths is:

$$\begin{array}{ccccc} & & \mathtt{a_1} & & \\ \cdots & \mathtt{c_h} & - & \cdots & \\ & \mathtt{b_n} & & & \end{array}$$

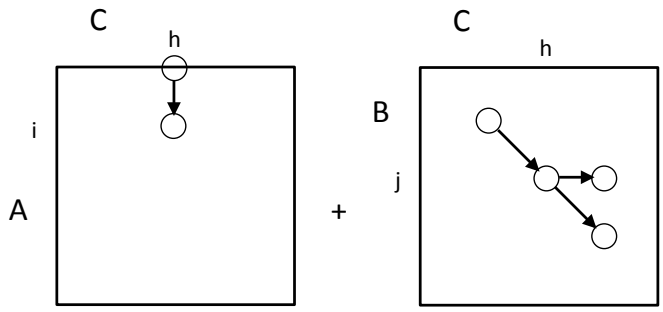


Figure A.11: Alignment subpaths of $(A \sim C)$ and $(B \sim C)$ that are consistent with a_1 **opening** a gap immediately after b_j (case: $i = 1$, $j > 0$, $0 < h < p$). The alignment that agrees with these subpaths is:

$$\begin{array}{ccccccc} & & & a_1 & & & \\ \cdots & c_h & - & \cdots & c_{h+1} & \cdots & \\ & b_j & & & \circ & & \\ & & & & \uparrow & \nearrow & \\ & & & & b_{j+1} & & \end{array}$$

The arrows indicate that the subpaths correspond to alignments of B with C such that b_{j+1} aligns either with or after c_{h+1}.

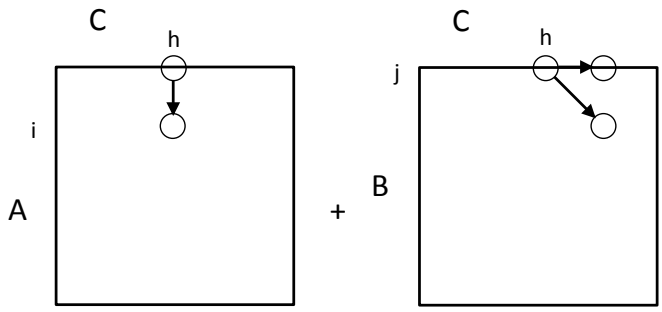


Figure A.12: Alignment subpaths of $(A \sim C)$ and $(B \sim C)$ that are consistent with a_1 **opening** a gap before the first character of B (case: $i = 1$, $j = 0$, $0 \leq h < p$). The alignment that agrees with these subpaths is:

$$\begin{array}{ccccccc} & & & & \mathtt{a}_1 & & \\ \cdots & \mathtt{c}_h & - & \cdots & \mathtt{c}_{h+1} & \cdots & \\ & & & & \circ & & \\ & & & & \uparrow & \nearrow & \\ & & & & \mathtt{b}_1 & & \end{array}$$

The arrows indicate that the subpaths correspond to alignments of B with C such that b_1 aligns either with or after c_{h+1}.

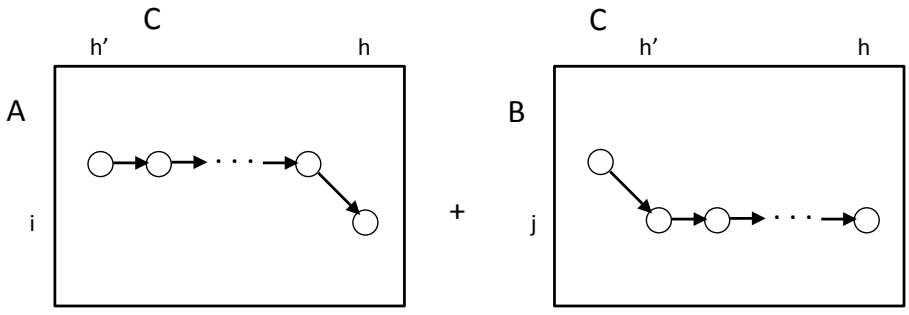


Figure A.13: Complex alignment subpaths of $(A \sim C)$ and $(B \sim C)$ that are consistent with a_i **opening** a gap between b_j and b_{j+1}. The alignment that agrees with these subpaths is:

$$\begin{array}{ccccccc} & \circ & - & - & - & \mathtt{a_i} & \\ \cdots & \mathtt{c_{h'}} & \cdot & \cdot & \cdot & \mathtt{c_h} & \cdots \\ & \mathtt{b_j} & - & - & - & - & \end{array}$$

In this complex graph pair, a_i is the first character of A to appear after a gap that goes back to at least $c_{h'}$, and b_j is the last character of B to appear before a gap that goes until at least c_h. This special case is valid only for $h' \leq h - 2$. See text for proof that an optimal pair (h', h) can be found in linear time for fixed i and j.

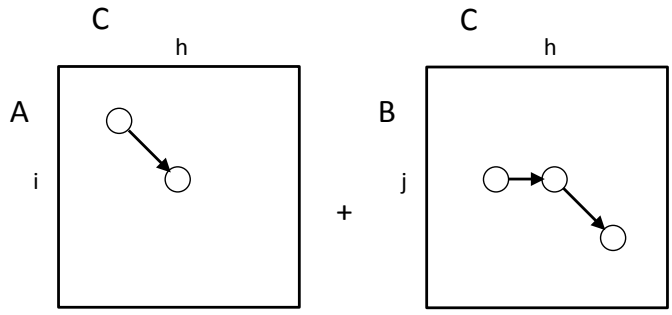


Figure A.14: Alignment subpaths of $(A \sim C)$ and $(B \sim C)$ that are consistent with a_i **closing** a gap between b_j and b_{j+1} (general case). The alignment that agrees with these subpaths is:

$$\begin{array}{ccccccc} & & & & \mathtt{a_i} & & \\ \cdots & \circ & \cdots & \mathtt{c_h} & \mathtt{c_{h+1}} & \cdots & \\ & \mathtt{b_j} & & - & \mathtt{b_{j+1}} & & \end{array}$$

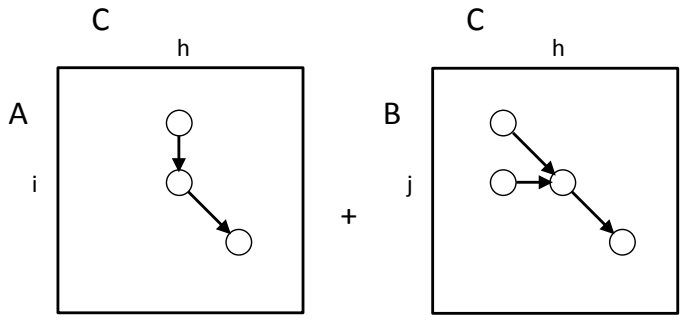


Figure A.15: Alignment subpaths of $(A \sim C)$ and $(B \sim C)$ that are consistent with a_i **closing** a gap between b_j and b_{j+1} (case: $i > 1$, $j > 0$, $0 < h < p$). The alignment that agrees with these subpaths is:

$$\begin{array}{ccccccc} & & & & \mathtt{a_i} & \mathtt{a_{i+1}} & \\ \cdots & \mathtt{c_h} & \cdots & - & \mathtt{c_{h+1}} & \cdots & \\ & \circ & & & \mathtt{b_{j+1}} & & \\ \nwarrow & \uparrow & & & & & \\ & \mathtt{b_j} & & & & & \end{array}$$

The arrows indicate that the subpaths correspond to alignments of B with C such that b_j aligns either with or before c_h.

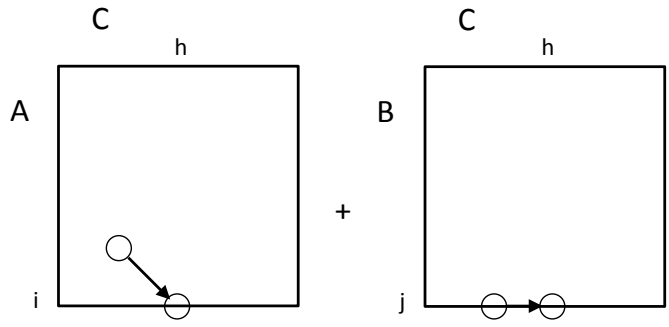


Figure A.16: Alignment subpaths of $(A \sim C)$ and $(B \sim C)$ that are consistent with a_m **closing** a terminal gap after the last character of B (case: $i = m$, $j = n$). The alignment that agrees with these subpaths is:

$$\begin{array}{ccccc} & & & & \mathtt{a_m} \\ \cdots & \circ & \cdots & \mathtt{c_h} & \cdots \\ & & \mathtt{b_n} & & \end{array}$$

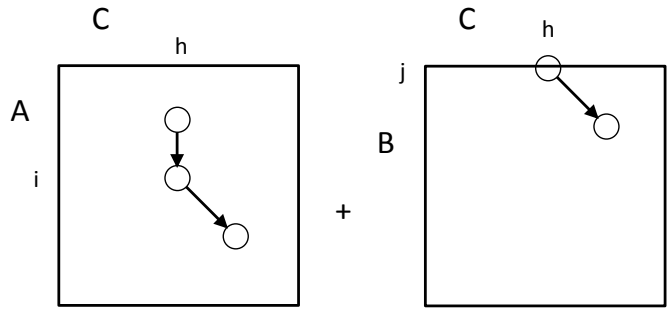


Figure A.17: Alignment subpaths of $(A \sim C)$ and $(B \sim C)$ that are consistent with a_i **closing** a terminal gap before b_1 (case: $j = 0$, $0 < i < m$, $0 \leq h < p$). The alignment that agrees with these subpaths is:

$$\begin{array}{ccccc} & & \mathtt{a_i} & \mathtt{a_{i+1}} & \\ \cdots & - & \mathtt{c_{h+1}} & \cdots & \\ & & \mathtt{b_1} & & \end{array}$$

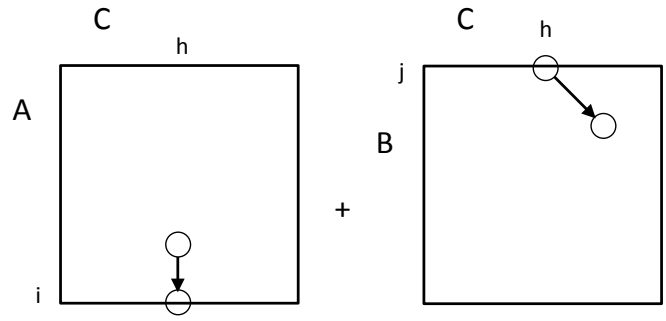


Figure A.18: Alignment subpaths of $(A \sim C)$ and $(B \sim C)$ that are consistent with a_m **closing** a terminal gap before the first character of B (case: $i = m$, $j = 0$, $0 \leq h < p$). The alignment that agrees with these subpaths is:

$$\begin{array}{ccccc} & & \mathtt{a_m} & & \\ \cdots & - & \mathtt{c_{h+1}} & \cdots & \\ & & \mathtt{b_1} & & \end{array}$$

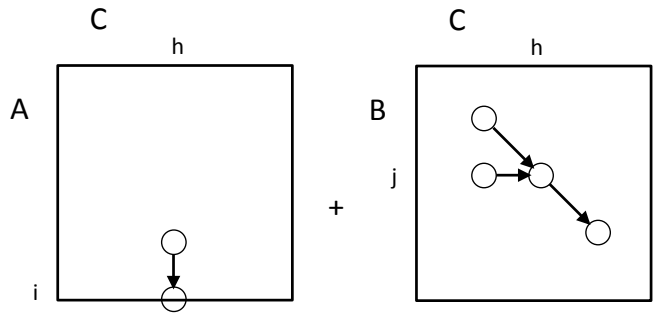


Figure A.19: Alignment subpaths of $(A \sim C)$ and $(B \sim C)$ that are consistent with a_m **closing** a gap between b_j and b_{j+1} (case: $i = m$, $0 < j < n$, $0 < h < p$). The alignment that agrees with these subpaths is:

$$\begin{array}{ccccccc} & & & & a_m & & \\ \cdots & c_h & \cdots & - & c_{h+1} & \cdots & \\ & \circ & & & b_{j+1} & & \\ \nwarrow & \uparrow & & & & & \\ & b_j & & & & & \end{array}$$

The arrows indicate that the subpaths correspond to alignments of B with C such that b_j aligns either with or before c_h.

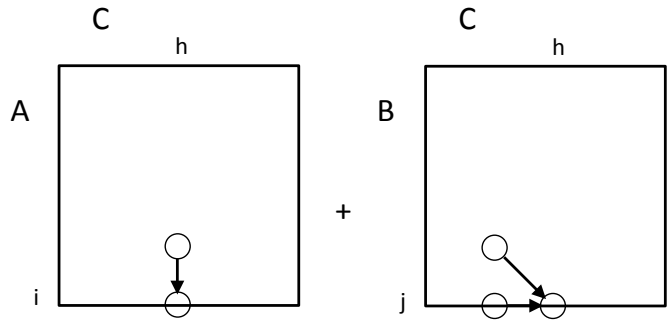


Figure A.20: Alignment subpaths of $(A \sim C)$ and $(B \sim C)$ that are consistent with a_m **closing** a terminal gap after b_n (case: $i = m$, $j = n$, $0 < h \leq p$). The alignment that agrees with these subpaths is:

$$\begin{array}{ccccc} & & & a_m & \\ \cdots & c_h & \cdots & - & \cdots \\ & \circ & & & \\ \nwarrow & \uparrow & & & \\ & b_n & & & \end{array}$$

The arrows indicate that the subpaths correspond to alignments of B with C such that b_n aligns either with or before c_h.

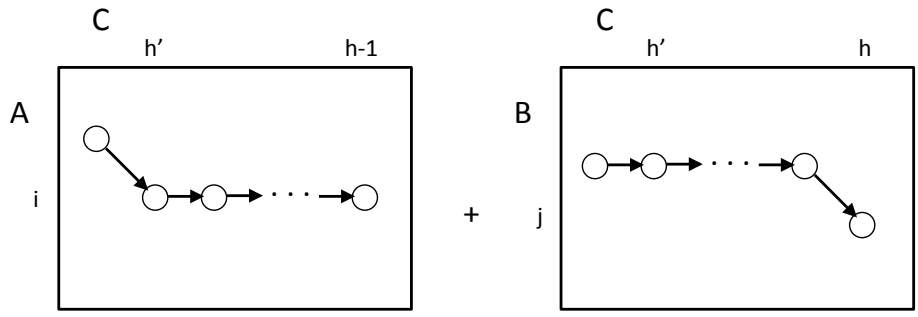


Figure A.21: Complex alignment subpaths of $(A \sim C)$ and $(B \sim C)$ that are consistent with a_i **closing** a gap between b_j and b_{j+1}. The alignment that agrees with these subpaths is:

$$\begin{array}{cccccccc} & \mathtt{a_i} & - & - & - & - & \circ & \\ \cdots & \mathtt{c_{h'}} & \cdot & \cdot & \cdot & \mathtt{c_{h-1}} & \mathtt{c_h} & \cdots \\ & - & - & - & - & - & \mathtt{b_j} & \end{array}$$

In this complex graph pair, a_i is the last character of A to appear before a gap starting at $c_{h'+1}$ and going to c_{h-1}, and b_j is the first character of B to appear after a gap going back to at least $c_{h'}$. This special case is valid only for that $h' \leq h-2$. See text for proof that an optimal pair (h', h) can be found in linear time for fixed i and j.

The reader should remember that these graphs are used in the computation of modified costs by finding, for each position-pair (i, j), the values of h' and h that result in minimum suboptimality. Most of the subpaths above involve only a single value, h, so scanning for the best h requires linear time for each (i, j). This leads to $O(n^3)$ run time for computing suboptimality modifiers from one sequence C for alignment $(A \sim B)$, when all three sequences have length n. However, two of the subpaths (Figures A.13 and A.21) involve a pair of values, h' and h. For a fixed pair (i, j), naively finding the pair (h', h) that gives smallest suboptimality would require quadratic time, leading to an overall $O(n^4)$ algorithm.

Fortunately, it is possible to find the optimal pair (h', h) for a fixed (i, j) in a single linear-time scan through C. This is achieved by keeping two pointers to positions in sequence C. Call these the leading and trailing pointers, which point to positions f and g, and serve as candidates for h' and h respectively. The leading pointer is advanced one position at a time, and the trailing pointer makes occasional (possibly multi-step) advances. When the leading pointer reaches the end of C, the process is complete; this is a linear process since the leading pointer moves one step at a time and neither pointer ever backtracks.

We now provide details to show how the optimal pair (h', h) is found. The approach will be described in the context of the average suboptimality described in 4.3. In that context, the pair of constrained alignments with lowest cost will also be the pair with lowest suboptimality. Modifying this approach to use the maximum suboptimality criterion is straightforward.

For a fixed i and j, we will maintain the invariant that, for a pointer value g, f is the position in C that results in the lowest average cost of constrained alignments $(A \sim C)$ and $(B \sim C)$ among values $\leq g - 2$. Initialize $f = 1$ and $g = 3$; f satisfies the above condition trivially. In the general case, we seek to advance g to g', and find the value f' that gives lowest cost among all values $\leq g' - 2$. It will be shown (below) that for $g' = g + 1$, f gives lowest cost among positions $\leq g' - 3$. Thus, for g', the optimal f' will be either f or $g - 2$. This means that a fixed amount of work is performed for each advancement of the leading pointer g. For fixed i and j, when all values g (each with corresponding f) have been tested, the pair (f, g) with lowest cost gives the (h', h) with lowest suboptimality. This is a linear scan per (i, j) pair, resulting in cubic run time to compute suboptimalities over all (i, j).

Theorem A.1 (Linear time to compute best (h', h) pair for suboptimality graph in Figure A.13)**.** *For fixed position g, let f be the position giving lowest cost constrained alignments $(A \sim C)$ and $(B \sim C)$ among values $\leq g - 2$. Then for position $g' = g + 1$, f is the position giving lowest constrained cost among values $\leq g' - 3$.*

Proof. For position pair (f, g), the two parts of the path-pair in Figure A.13 can be decomposed into subpaths that will be used in this proof. Specifically, the cost of the optimal alignment $(A \sim C)$ forced to go through positions f and g can be

broken into the sum $L + M + N + P$, as shown in Figure A.22. The cost for the similar alignment for $(B \sim C)$ can be broken into into the sum $W + X + Y + Z$, as in Figure A.24. Similar decompositions can be made for position pair (f, g'), as in Figures A.23 and A.25.

Let $C(f, g)$ be the sum of $(A \sim C)$ and $(B \sim C)$ costs going through f and g of Figures A.22 and A.24, and let $C(f, g')$ be the similar sum going through f and g'. It is clear that

$$C(f, g') = C(f, g) - N - P - Z + N' + P' + Z' + 2\lambda$$

for any $f \leq g - 2$. Thus, when g' replaces g, all positions $f \leq g - 2$ will be subject to constant change in cost, so the relative order of costs for positions $\leq g - 2$ is unchanged when g is replaced by g'. This ensures that the $f \leq g - 2$ that gives lowest cost alignments for subpaths ending at g will also give the lowest cost alignments for subpaths ending at $g' = g + 1$, among values $\leq g' - 3$.

□

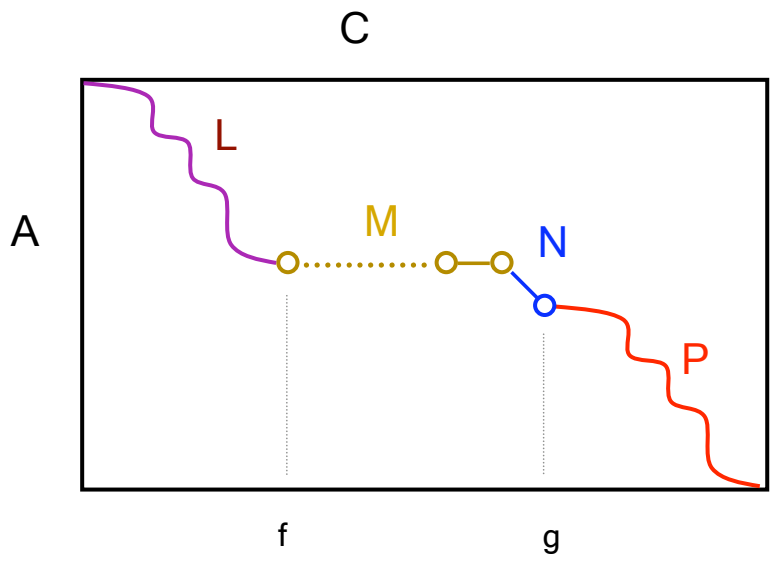


Figure A.22: Decomposing the $(A \sim C)$ part of figure A.13 (1). Used in proof of linear-time computation of optimal (f, g) pair.

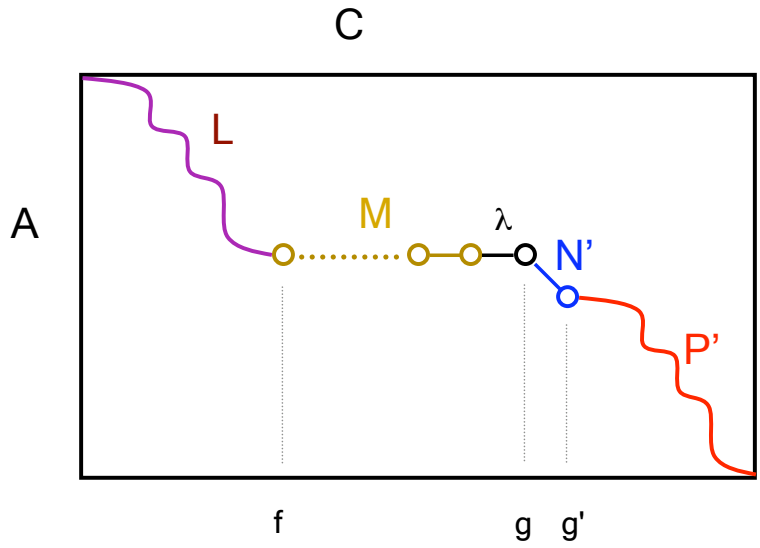


Figure A.23: Decomposing the $(A \sim C)$ part of figure A.13 (1). Used in proof of linear-time computation of optimal (f, g) pair.

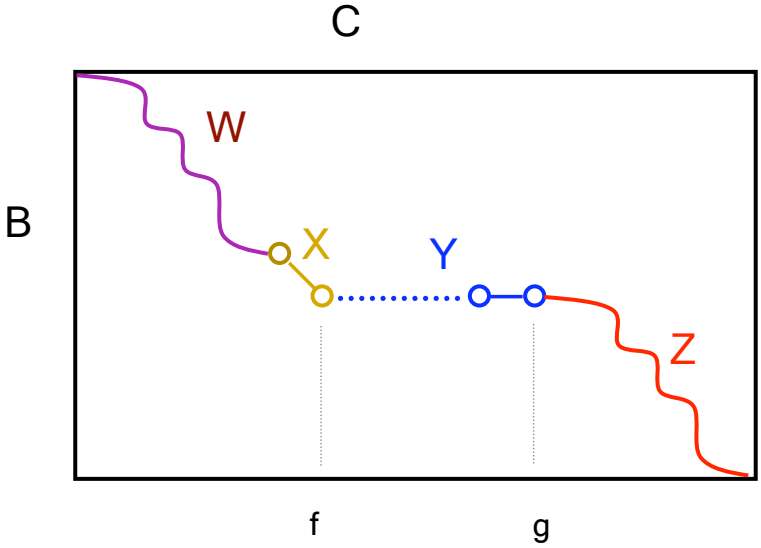


Figure A.24: Decomposing the $(B \sim C)$ part of figure A.13 (1). Used in proof of linear-time computation of optimal (f, g) pair.

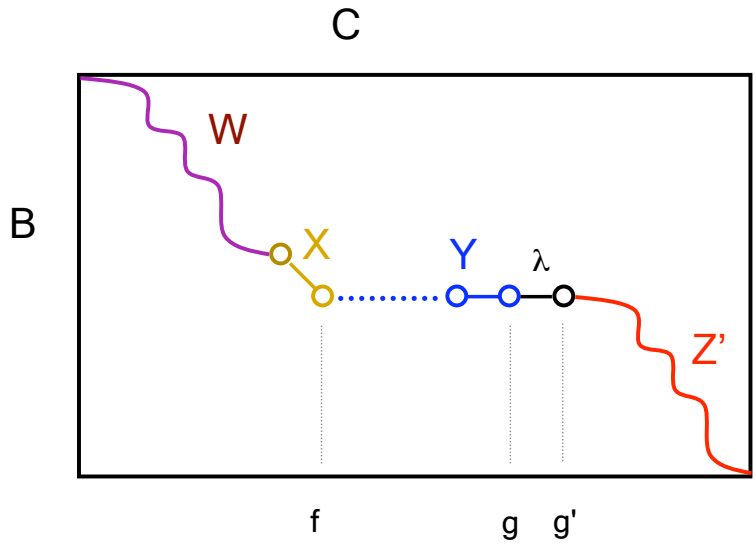


Figure A.25: Decomposing the $(B \sim C)$ part of figure A.13 (1). Used in proof of linear-time computation of optimal (f, g) pair.

APPENDIX B

ON JAVA VERSUS C++ FOR SCIENTIFIC COMPUTING

In this appendix, I briefly describe my personal experience with developing complex scientific applications in Java. I start by giving my motivations for moving from C++ to Java in the first place. I then give a sense of how Java has performed in my two scientific computing applications, and reach the conclusion that (1) Java is an excellent language for prototyping scientific algorithms and gives acceptable speed if used carefully, but (2) C++ appears to give better (faster) performance, especially in I/O-intensive applications, and is thus preferable for polished software release.

Based on personal observation, the implementation languages of choice in computational biology appear to be C and C++. Java seems to have much less penetrance in the market of serious scientific computing, likely because it is viewed as being too slow or restrictive for such use.

The original implementation of `Opal` [115] was in C, and depended on an existing C implementation of the algorithms mentioned in Section 2.5 for aligning alignments [63] . As I began development related to the consistency methods described in Chapter 4, it was clear that a great deal of refactoring of code would be required to incorporate the modified cost schemes, and the flexibility of an object-oriented programming language would make it much easier to test the wide range of ideas we had in mind. Because I preferred the ease of development, debugging, and portability that Java offer, I decided to investigate reimplementing `Opal` with Java. The results of Bull et al. [15] and a variety of benchmark tests posted to the web (e.g. http://www.idiom.com/∼zilla/Computer/javaCbenchmark.html and http://kano.net/javabench/) were compelling enough to convince me to reimplement the code for aligning alignments (`AlignAlign`) in Java to compare performance.

`AlignAlign` is an interesting test case, in that the C implementation requires a fairly complex memory pool management scheme in order to efficiently deal with creating and discarding the shapes described in Section 4.5. It is also likely an ideal candidate for porting to Java, since input files can be handled much more easily with Java's regular expression engine than with C, and its core computation effectively consists of a big sweep through a 2-dimensional array. In the Java implementation, I was able to do away with pool management, leaving the job to the Java virtual machine (JVM) garbage collector. With otherwise essentially identical algorithms, the Java implementation ran about 10% slower than a heavily optimized C implementation, on average over a wide range of inputs and several

computers. It should be noted that this required careful avoidance of creating unnecessary objects inside core loops of the program, as object creation in Java is slow. It seemed worthwhile to pay a 10% performance penalty for improved ease of implementation and debugging. A side benefit of implementing in pure Java is that it was a simple matter to plug `Opal` into `Mesquite` (http://mesquiteproject.org), a well-regarded software suite for evolutionary biology.

`NINJA` was also developed in Java, and requires quite a bit of low-level disk-I/O code. In this case, it appears that a C implementation might provide a substantial speed improvement. `NINJA` is roughly as fast as `RapidNJ` (written in C++) on large inputs of more than 1000 sequences, but `RapidNJ` is much faster for small inputs. In fact, on inputs of a few hundred sequences `RapidNJ` completes in less time than `NINJA` requires just to read in the quadratic-sized distance matrix, despite significant effort invested in optimizing `NINJA`'s disk-reading functions. This suggests that `NINJA`'s success on larger inputs is due to algorithmic success, and that porting `NINJA` to C might result in a modest speed improvement.

In addition to the slow disk-I/O in java, other issues have come to light that suggest limitations to the value of Java in scientific computing. First, on a 32-bit machine, JVMs will allow allocation of only 2GB RAM, even if more is available on the machine; this, for example, may not be sufficient to store the dynamic programming tables for aligning very long sequences. Second, even on 64-bit machines, memory allocation appears to be a problem on machines with huge RAM; Morgan Price (pers. comm.) reports that his JVM crashed repeatedly when attempting to allocate more than 60GB on a machine that has 128GB available - this was not a `NINJA` error. In addition, Java appears to be prone to dropping buffered data when processing data from a Unix pipe, though I've not seen this documented elsewhere. A final problem is that the portability of Java applications depends on the availability of a JVM; while appropriate JVMs are available for download, end users may find themselves unable to install the JVM on a server over which they have no control. Furthermore, some more recent benchmarking results released to the web have been less supportive of Java's asserted near-equality to the C variants (e.g. http://www.stefankrause.net/wp/?p=9)

Having developed two serious scientific computing tools in Java, I believe that it is an excellent test-bed for algorithms, providing a relatively simple programming environment with a reasonable speed. However, any tool that is likely to be used heavily by the public, and for which a premium will be placed on performance, should probably be ported to C++ in the end. This should be a fairly simple port, given the similarities of the languages.

REFERENCES

[1] A. Aitken, *On least squares and linear combination of observations*, Proceedings of the Royal Society of Edinburgh. Section A. Mathematics, 55 (1934), p. 42.

[2] S. Altschul, *Gap costs for multiple sequence alignment*, Journal of theoretical biology, 138 (1989), pp. 297–309.

[3] S. Altschul, R. Carroll, and D. Lipman, *Weights for data related by a tree.*, Journal of Molecular Biology, 207 (1989), pp. 647–653.

[4] S. F. Altschul, *Leaf pairs and tree dissections*, SIAM Journal on Discrete Mathematics, 2 (1989), pp. 293–299.

[5] K. Atteson, *The performance of neighbor-joining methods of phylogenetic reconstruction*, Algorithmica, 25 (1999), pp. 251–278.

[6] A. Bahr, J. Thompson, J. Thierry, and O. Poch, *BAliBASE (Benchmark Alignment dataBASE): enhancements for repeats, transmembrane sequences and circular permutations*, Nucleic Acids Research, 29 (2001), pp. 323–326.

[7] S. Balaji, S. Sujatha, S. Kumar, and N. Srinivasan, *Pali—a database of phylogeny and alignment of homologous protein structures*, Nucleic Acids Research, 29 (2001), pp. 61–65.

[8] R. Bayer and E. McCreight, *Organization and maintenance of large ordered indexes*, Acta informatica, 1 (1972), pp. 173–189.

[9] M. Berger and P. Munson, *A novel randomized iterative strategy for aligning multiple protein sequences*, CABIOS, 7 (1991), pp. 479–484.

[10] M. Bordewich, O. Gascuel, K. Huber, and V. Moulton, *Consistency of topological moves based on the balanced minimum evolution principle of phylogenetic inference*, IEEE/ACM Transactions on Computational Biology and Bioinformatics, 6 (2009), pp. 110–117.

[11] R. Bradley, A. Roberts, M. Smoot, S. Juvekar, J. Do, C. Dewey, I. Holmes, and L. Pachter, *Fast statistical alignment*, PLoS Computational Biology, 5 (2009), p. e1000392.

[12] K. Brengel, A. Crauser, P. Ferragina, and U. Meyer, *An experimental study of priority queues in external memory*, Proceedings of the 3rd International Workshop on Algorithm Engineering, (1999), pp. 345–359.

[13] D. Bryant, *On the uniqueness of the selection criterion in neighbor-joining*, Journal of Classification, 22 (2005), pp. 3–15.

[14] D. Bryant and P. Waddell, *Rapid evaluation of least-squares and minimum-evolution criteria on phylogenetic trees*, Molecular Biology and Evolution, 15 (1998), pp. 1346–1359.

[15] J. Bull, L. Smith, L. Pottage, and R. Freeman, *Benchmarking java against c and fortran for scientific applications*, Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande, (2001), pp. 97–105.

[16] M. Bulmer, *Use of the method of generalized least squares in reconstructing phylogenies from sequence data*, Molecular Biology and Evolution, 8 (1991), pp. 868–883.

[17] H. Carrillo and D. Lipman, *The multiple sequence alignment problem in biology*, SIAM Journal on Applied Mathematics, 48 (1988), pp. 1073–1082.

[18] H. Carroll, W. Beckstead, T. O'Connor, and M. Ebbert, *Dna reference alignment benchmarks based on tertiary structure of encoded proteins*, Bioinformatics, 23 (2007), pp. 2648–2649.

[19] L. Cavalli-Sforza and A. Edwards, *Phylogenetic analysis models and estimation procedures*, American Journal of Human Genetics, 19 (1967), pp. 233–257.

[20] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to algorithms*, 2001.

[21] C. Darwin, *On the origin of species by means of natural selection, or the preservation of favoured races in the struggle for life*, New York: D. Appleton, (1859).

[22] R. Desper and O. Gascuel, *Fast and accurate phylogeny reconstruction algorithms based on the minimum-evolution principle*, J Comput Biol, 9 (2002), pp. 687–705.

[23] ——, *Theoretical foundation of the balanced minimum evolution method of phylogenetic inference and its relationship to weighted least-squares tree fitting*, Molecular Biology and Evolution, 21 (2004), pp. 587–598.

[24] ——, *The minimum evolution distance-based approach to phylogenetic inference*, In: Gascuel O, editor. Mathematics of evolution & phylogeny. Oxford, UK: Oxford University Press, (2005), pp. 1–32.

[25] C. Do, M. Mahabhashyam, M. Brudno, and S. Batzoglou, *Probcons: Probabilistic consistency-based multiple sequence alignment*, Genome Research, 15 (2005), pp. 330–340.

[26] P. Domingos and M. Pazzani, *On the optimality of the simple Bayesian classifier under zero-one loss*, Machine learning, (1997).

[27] A. Drummond and A. Rambaut, *Beast: Bayesian evolutionary analysis by sampling trees*, BMC Evol Biol, (2007).

[28] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison, *Biological sequence analysis: Probabilistic models of proteins and nucleic acids*, books.google.com, (1998).

[29] R. Edgar, *Muscle: a multiple sequence alignment method with reduced time and space complexity*, BMC Bioinformatics, 5 (2004), p. 113.

[30] R. Edgar, *Muscle: multiple sequence alignment with high accuracy and high throughput*, Nucleic Acids Research, (2004).

[31] I. Elias and J. Lagergren, *Fast neighbor joining*, Proc. of the 32nd International Colloquium on Automata, Languages and Programming (ICALP'05, (2005), pp. 1263–1274.

[32] J. Evans, L. Sheneman, and J. Foster, *Relaxed neighbor joining: a fast distance-based phylogenetic tree construction method*, Journal of Molecular Evolution, (2006).

[33] J. Farris, V. Albert, M. Kallersjo, and D. Lipscomb, *Parsimony jackknifing outperforms neighbor-joining*, Cladistics, (1996).

[34] J. Felsenstein, *Numerical methods for inferring evolutionary trees*, Quarterly Review of Biology, (1982), pp. 379–404.

[35] J. Felsenstein, *Inferring phylogenies*, (2004).

[36] D. Feng and R. Doolittle, *Progressive sequence alignment as a prerequisite to correct phylogenetic trees*, Journal of Molecular Evolution, 25 (1987), pp. 351–360.

[37] R. D. Finn, J. Tate, J. Mistry, P. C. Coggill, S. J. Sammut, H. Hotz, G. Ceric, K. Forslund, S. R. Eddy, E. L. L. Sonnhammer, and A. Bateman, *The pfam protein families database*, Nucleic Acids Research, 36 (2007), pp. D281–D288.

[38] P. P. Gardner, *A benchmark of multiple sequence alignment programs upon structural rnas*, Nucleic Acids Research, 33 (2005), pp. 2433–2439.

[39] O. Gascuel, *BIONJ: an improved version of the NJ algorithm based on a simple model of sequence data*, Molecular Biology and Evolution, (1997).

[40] O. Gascuel and M. Steel, *Neighbor-joining revealed*, Molecular Biology and Evolution, 23 (2006), pp. 1997–2000.

[41] N. Goldman and Z. Yang, *Introduction. statistical and computational challenges in molecular phylogenetics and evolution*, Philosophical transactions of the Royal Society of London. Series B, Biological sciences, 363 (2008), pp. 3889–3892.

[42] P. A. Goloboff, S. A. Catalano, J. M. Mirande, C. A. Szumik, J. S. Arias, M. Kallersjo, and J. S. Farris, *Phylogenetic analysis of 73 060 taxa corroborates major eukaryotic groups*, Cladistics, 25 (2009), pp. 211–230.

[43] P. A. Goloboff, J. S. Farris, and K. C. Nixon, *TNT, a free program for phylogenetic analysis*, Cladistics, 24 (2008), pp. 774–786.

[44] O. Gotoh, *Consistency of optimal sequence alignments*, Bulletin of Mathematical Biology, 52 (1990), pp. 509–525.

[45] ——, *Optimal alignment between groups of sequences and its application to multiple sequence alignment*, CABIOS, 9 (1993), pp. 361–370.

[46] ——, *Further improvement in methods of group-to-group sequence alignment with generalized profile ...*, CABIOS, 10 (1994), pp. 379–387.

[47] ——, *A weighting system and aigorithm for aligning many phylogenetically related sequences*, CABIOS, 11 (1995), pp. 543–551.

[48] ——, *Significant improvement in accuracy of multiple protein sequence alignments by iterative refinement as assessed by reference to structural alignments*, Journal of Molecular Biology, 264 (1996), pp. 823–838.

[49] S. Griffiths-Jones, S. Moxon, M. Marshall, A. Khanna, S. R. Eddy, and A. Bateman, *Rfam: annotating non-coding RNAs in complete genomes*, Nucleic Acids Research, 33 (2005), pp. D121–D124.

[50] D. Gusfield, *Algorithms on strings, trees, and sequences: computer science and computational biology*, (1997).

[51] S. Henikoff and J. Henikoff, *Amino acid substitution matrices from protein blocks*, Proceedings of the National Academy of Sciences, 89 (1992), pp. 10915–10919.

[52] M. Hirosawa, Y. Totoki, M. Hoshida, and M. Ishikawa, *Comprehensive study on iterative algorithms of multiple sequence alignment*, CABIOS, 11 (1995), pp. 13–18.

[53] W. Hordijk and O. Gascuel, *Improving the efficiency of SPR moves in phylogenetic tree search methods based on maximum likelihood*, Bioinformatics, 21 (2005), pp. 4338–4347.

[54] K. Howe, A. Bateman, and R. Durbin, *QuickTree: building huge neighbour-joining trees of protein sequences*, Bioinformatics, 18 (2002), pp. 1546–1547.

[55] X. Huang and W. Miller, *A time-efficient linear-space local similarity algorithm*, Advances in Applied Mathematics, 12 (1991), pp. 337–357.

[56] J. Huelsenbeck and F. Ronquist, *MRBAYES: Bayesian inference of phylogenetic trees*, Bioinformatics, 17 (2001), pp. 754–755.

[57] L. Hurst, *The Ka/Ks ratio: diagnosing the form of sequence evolution*, TRENDS in Genetics, 18 (2002), pp. 486–487.

[58] T. Jukes and C. Cantor, *Evolution of protein molecules*, Mammalian protein metabolism, 3 (1969), pp. 21–132.

[59] K. Katoh, K. Kuma, H. Toh, and T. Miyata, *MAFFT version 5: improvement in accuracy of multiple sequence alignment*, Nucleic Acids Research, 33 (2005), pp. 511–518.

[60] K. Katoh, K. Misawa, K. Kuma, and T. Miyata, *MAFFT: a novel method for rapid multiple sequence alignment based on fast fourier transform*, Nucleic Acids Research, 30 (2002), p. 3059.

[61] J. Kececioglu, *The maximum weight trace problem in multiple sequence alignment*, Proceedings of the 4th Annual Symposium on Combinatorial Pattern Matching, (1993), pp. 106–119.

[62] J. Kececioglu and E. Kim, *Simple and fast inverse alignment*, Proceedings of the 10th ACM Conference on Research in Computational Molecular Biology (RECOMB), 3909 (2006), pp. 441–455.

[63] J. Kececioglu and D. Starrett, *Aligning alignments exactly*, Proceedings of the 8th ACM Conference on Research in Computational Molecular Biology (RECOMB 2004), (2004), pp. 85–96.

[64] J. Kececioglu and W. Zhang, *Aligning alignments*, Combinatorial Pattern Matching: 9th Annual Symposium, (1998), pp. 189–208.

[65] M. Kimura, *The neutral theory of molecular evolution*, Cambridge University Press, (1983).

[66] D. Knuth, *The art of computer programming, vol. 3: Sorting and searching*, Addison-Wesley, (1973).

[67] M. Kuhner and J. Felsenstein, *A simulation comparison of phylogeny algorithms under equal and unequal evolutionary rates*, Molecular Biology and Evolution, 11 (1994), pp. 459–468.

[68] S. Kumar, *A stepwise algorithm for finding minimum evolution trees*, Molecular Biology and Evolution, 13 (1996), pp. 584–593.

[69] S. Kumar and S. Gadagkar, *Efficiency of the neighbor-joining method in reconstructing deep and shallow evolutionary relationships in large phylogenies*, Journal of Molecular Evolution, 51 (2000), pp. 544–553.

[70] J. Lake, *The order of sequence alignment can bias the selection of tree topology*, Molecular Biology and Evolution, 8 (1991), pp. 378–385.

[71] D. Lipman, S. Altschul, and J. Kececioglu, *A tool for multiple sequence alignment*, Proceedings of the National Academy of Sciences, 86 (1989), pp. 4412–4415.

[72] B. Ma, Z. Wang, and K. Zhang, *Alignment between two multiple alignments*, Combinatorial pattern matching: 14th annual symposium, CPM, (2003), pp. 254–265.

[73] W. Maddison, *Gene trees in species trees*, Systematic Biology, 46 (1997), pp. 523–536.

[74] W. Maddison and L. Knowles, *Inferring phylogeny despite incomplete lineage sorting*, Systematic Biology, 55 (2006), pp. 21–30.

[75] T. Mailund, G. Brodal, R. Fagerberg, C. Pedersen, and D. Phillips, *Recrafting the neighbor-joining method*, BMC Bioinformatics, 7 (2006).

[76] T. Mailund and C. Pedersen, *QuickJoin-fast neighbour-joining tree reconstruction*, Bioinformatics, 20 (2004), pp. 3261–3262.

[77] R. Mihaescu, D. Levy, and L. Pachter, *Why neighbor-joining works*, Algorithmica, 54 (2009), pp. 1–24.

[78] S. Miyazawa, *A reliable sequence alignment method based on probabilities of residue correspondences*, Protein Engineering, 8 (1995), pp. 999–1009.

[79] T. Muller, R. Spang, and M. Vingron, *Estimating amino acid substitution models: a comparison of Dayhoff's estimator, the resolvent approach and a maximum likelihood method*, Molecular Biology and Evolution, 19 (2002), pp. 8–13.

[80] A. Murzin, S. Brenner, T. Hubbard, and C. Chothia, *SCOP: a structural classification of proteins database for the investigation of sequences and structures*, Journal of Molecular Biology, 247 (1995), pp. 536–540.

[81] L. Nakhleh, B. Moret, U. Roshan, K. John, and T. Warnow, *The accuracy of fast phylogenetic methods for large datasets*, Proc. 7th Pacific Symp. Biocomputing PSB 2002, (2002), pp. 211–222.

[82] M. Ninio, E. Privman, T. Pupko, and N. Friedman, *Phylogeny reconstruction: increasing the accuracy of pairwise distance estimation using Bayesian inference of evolutionary rates*, Bioinformatics, 23 (2007), pp. e136–e141.

[83] C. Notredame, D. Higgins, and J. Heringa, *T-Coffee: A novel method for fast and accurate multiple sequence alignment*, Journal of Molecular Biology, 302 (2000), pp. 205–217.

[84] C. Notredame, L. Holm, and D. Higgins, *Coffee: an objective function for multiple sequence alignments*, Bioinformatics, 14 (1998), pp. 407–422.

[85] T. Ogden and M. Rosenberg, *Multiple sequence alignment accuracy and phylogenetic inference*, Systematic Biology, 55 (2006), pp. 314–328.

[86] B. Paten, J. Herrero, K. Beal, and E. Birney, *Sequence progressive alignment, a framework for practical large-scale probabilistic consistency alignment*, Bioinformatics, 25 (2008), pp. 295–301.

[87] D. Patterson, *Latency lags bandwith*, Commun Acm, 47 (2004), pp. 71–75.

[88] J. Pei and N. Grishin, *MUMMALS: multiple sequence alignment improved by using hidden markov models with local structural information*, Nucleic Acids Research, 34 (2006), pp. 4364–4374.

[89] J. Pei and N. V. Grishin, *PROMALS: towards accurate multiple sequence alignments of distantly related proteins*, Bioinformatics, 23 (2007), pp. 802–808.

[90] M. N. Price, P. S. Dehal, and A. P. Arkin, *FastTree: Computing large minimum evolution trees with profiles instead of a distance matrix*, Molecular Biology and Evolution, 26 (2009), pp. 1641–1650.

[91] B. Redelings and M. Suchard, *Joint bayesian estimation of alignment and phylogeny*, Systematic Biology, 54 (2005), pp. 401–418.

[92] F. Ronquist and J. Huelsenbeck, *MrBayes 3: Bayesian phylogenetic inference under mixed models*, Bioinformatics, 19 (2003), pp. 1572–1574.

[93] U. Roshan and D. R. Livesay, *Probalign: multiple sequence alignment using partition function posterior probabilities*, Bioinformatics, 22 (2006), pp. 2715–2721.

[94] A. Rzhetsky and M. Nei, *Theoretical foundation of the minimum-evolution method of phylogenetic inference*, Molecular Biology and Evolution, 10 (1993), pp. 1073–1095.

[95] N. Saitou and M. Nei, *The neighbor-joining method: a new method for reconstructing phylogenetic trees*, Molecular Biology and Evolution, 4 (1987), pp. 406–425.

[96] D. Sankoff, *Minimal mutation trees of sequences*, SIAM Journal on Applied Mathematics, 28 (1975), pp. 35–42.

[97] A. S. Schwartz and L. Pachter, *Multiple alignment by sequence annealing*, Bioinformatics, 23 (2007), pp. e24–e29.

[98] L. Sheneman, J. Evans, and J. Foster, *Clearcut: a fast implementation of relaxed neighbor joining*, Bioinformatics, 22 (2006), pp. 2823–2824.

[99] M. Simonsen, T. Mailund, and C. Pedersen, *Rapid neighbour-joining*, Proceedings of the 8th international Workshop on Algorithms in Bioinformatics, (2008), pp. 113–122.

[100] S. A. Smith, J. M. Beaulieu, and M. J. Donoghue, *Mega-phylogeny approach for comparative biology: an alternative to supertree and supermatrix approaches*, BMC Evol Biol, 9 (2009), pp. 1–12.

[101] P. Sneath and R. Sokal, *Numerical taxonomy: the principles and practice of numerical classification*, Springer, (1973).

[102] A. Stamatakis, *RAxML-VI-HPC: maximum likelihood-based phylogenetic analyses with thousands of taxa and mixed models*, Bioinformatics, 22 (2006), pp. 2688–2690.

[103] D. Starrett, *Optimal alignment of multiple sequences*, Dissertation, (2008).

[104] K. St.John, T. Warnow, B. Moret, and L. Vawter, *Performance study of phylogenetic methods: (unweighted) quartet methods and neighbor-joining*, J Algorithm, 48 (2003), pp. 173–193.

[105] J. Studier and K. Keppler, *A note on the neighbor-joining algorithm of Saitou and Nei*, Molecular Biology and Evolution, 5 (1988), pp. 729–731.

[106] S. Subbiah and S. Harrison, *A method for multiple sequence alignment with gaps*, Journal of Molecular Biology, 209 (1989), pp. 539–548.

[107] M. Suchard and B. Redelings, *BAli-Phy: simultaneous Bayesian inference of alignment and phylogeny*, Bioinformatics, 22 (2006), pp. 2047–2048.

[108] K. Tamura, M. Nei, and S. Kumar, *Prospects for inferring very large phylogenies by using the neighbor-joining method*, Proceedings of the National Academy of Sciences, 101 (2004), pp. 11030–11035.

[109] J. Thompson, F. Plewniak, and O. Poch, *A comprehensive comparison of multiple sequence alignment programs*, Nucleic Acids Research, 27 (1999), pp. 2682–2690.

[110] J. D. Thompson, D. G. Higgins, and T. J. Gibson, *CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice*, Nucleic Acids Research, 22 (1994), pp. 4673–4680.

[111] M. Vingron and M. Waterman, *Sequence alignment and penalty choice. review of concepts, case studies and implications.*, Journal of Molecular Biology, 235 (1994), pp. 1–12.

[112] I. Wallace, OO'Sullivan, D. Higgins, and C. Notredame, *M-Coffee: combining multiple sequence alignment methods with T-Coffee*, Nucleic Acids Research, 34 (2006), pp. 1692–1699.

[113] I. V. Walle, I. Lasters, and L. Wyns, *SABmark–a benchmark for sequence alignment that covers the entire known fold space*, Bioinformatics, 21 (2005), pp. 1267–1268.

[114] L. Wang and T. Jiang, *On the complexity of multiple sequence alignment.*, J Comput Biol, 1 (1994), pp. 337–348.

[115] T. Wheeler and J. Kececioglu, *Multiple alignment by aligning alignments*, Bioinformatics, 23 (2007), pp. i559–i568.

[116] W. C. Wheeler, J. Gatesy, and R. DeSalle, *Elision: a method for accommodating multiple molecular sequence alignments with alignment-ambiguous sites*, Molecular Phylogenetics and Evolution, 4 (1995), pp. 1–9.

[117] Z. Yang and B. Rannala, *Bayesian phylogenetic inference using DNA sequences: A Markov chain monte carlo method*, Molecular Biology and Evolution, 14 (1997), pp. 717–724.

[118] J. Yuan, A. Amend, J. Borkowski, R. DeMarco, W. Bailey, Y. Liu, G. Xie, and R. Blevins, *MULTICLUSTAL: a systematic method for surveying Clustal W alignment parameters*, Bioinformatics, 15 (1999), pp. 862–863.

[119] L. Zaslavsky and T. A. Tatusova, *Accelerating the neighbor-joining algorithm using the adaptive bucket data structure*, Lecture Notes in Computer Science, 4983 (2008), pp. 122–133.