Algorithms for minimum bipartite fill-in and the gene-duplication problem

by

Mukul Subodh Bansal

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
David Fernandez-Baca, Major Professor
Maria Axenovich
Oliver Eulenstein
Giora Slutzki

Iowa State University

Ames, Iowa

2006

UMI Number: 1439884

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

Finishing this thesis marks a milestone of sorts during my ongoing graduate career here at Iowa State University. I would certainly not have made it this far without the people who blessed me with their guidance, support and company.

First of all I would like to thank my advisor David Fernandez-Baca for his superlative guidance, patience, and immeasurable support. It is difficult to overstate my gratitude towards him. It is from him that I learn how to think, do research, and even how to teach. I have also been very fortunate to have had the opportunity to work with Oliver Eulenstein over the past several months. He has been like a second advisor to me. I am very grateful to him for sharing with me his invaluable ideas and his boundless enthusiasm for research.

I would like to thank Giora Slutzki for all his help and guidance. His active interest in the minimum bipartite fill-in problem led to many interesting and insightful discussions.

Special thanks to Maria Axenovich for her truly exceptional graph theory course and for being a member on my MS and PhD committees despite her busy schedule. I also thank Pavan Aduri, Srinivas Aluru, Soma Chaudhuri and Wallapak Tavanapong along with David Fernandez-Baca, Oliver Eulenstein and Giora Slutzki for their extraordinary courses.

I am indebted to Wen-Chieh Chang, Duhong Chen and Andre Wehe for being such excellent friends, colleagues and collaborators, and for providing a fun and stimulating environment in which to learn and grow. A very special thanks to my friend Jia Tao, for all the wonderful time I spent with her. I thank Raul Piaggio and Balaji Venkatachalam for their help and their company during the time they were part of our research group.

Finally, I thank Linda Dutton for always being so approachable and friendly and for going out of her way to make sure the thesis defense could be held without any hiccups.

CHAPTER 1. Introduction

This thesis deals with two different problems: 1) the minimum bipartite fill-in problem, and 2) the gene-duplication problem. We establish the fixed parameter tractability of the minimum bipartite fill-in problem[1] and develop a faster local search algorithm which speeds up heuristics for the gene-duplication problem[2].

The *minimum bipartite fill-in* problem is the problem of finding an edge set of minimum cardinality whose addition to a given bipartite graph makes it chordal bipartite. The parameterized version of this problem asks if a bipartite graph can be made chordal bipartite by adding at most k edges. We show that this problem is fixed parameter tractable by presenting a search tree based algorithm that solves it in $O(8^{2k}m \log n)$ time.

The *gene-duplication* problem is to infer a species supertree from a collection of gene trees that are confounded by complex histories of gene duplications. This problem is NP-hard and thus requires efficient and effective heuristics. Existing heuristics perform a stepwise search of the tree space, where each step is guided by an exact solution to an instance of a local search problem. We show that the local search problem can be solved in $O(n^2)$ time, where n is the number of species in the resulting supertree solution. This improves the running time of the current solution by a factor of n and makes the gene-duplication problem more tractable for large-scale phylogenetic analyses. We verify the exceptional performance of our solution in a comparison study using sets of large randomly generated gene trees. Furthermore, we demonstrate the utility of our solution by incorporating large genomic data sets from GenBank into a supertree analysis of plants.

This thesis is organized as follows: In the next chapter we present our algorithm for mini-

[1]This work was done in collaboration with David Fernandez-Baca

[2]This work was done in collaboration with Oliver Eulenstein, Andre Wehe and Gordon Burleigh

mum bipartite fill-in and show that it is fixed parameter tractable. In chapter 3, we present our improved local search algorithm for speeding up heuristics for the gene-duplication problem.

CHAPTER 2. Parameterized algorithm for minimum bipartite fill-in

2.1 Introduction

A formal study of fixed parameter tractability was initiated by Downey and Fellows (17; 1; 18). A parameterized problem is called *fixed parameter tractable* (FPT) if it has an algorithm that has complexity $O(f(k)n^{\alpha})$ where k is the parameter, n is the size of the problem and α is a constant. Fixed parameter tractability is an important concept because many NP-hard problems can be solved in polynomial time when a parameter in the problem is fixed. For example, consider the following problem: Given a graph, decide if it has a vertex cover of size at most k. This parameterized version of the well known *vertex cover* problem can be solved in linear time when k is fixed . Thus, the vertex cover problem is fixed parameter tractable. The way the complexity depends on k can vary dramatically for different problems. For example, the parameterized version of the well known *maximum independent set* problem can be stated as follows: Given a graph, decide if it has an independent set of size at least k. The best known algorithms for this problem have $\Omega(n^k)$ time complexity and hence it is not known to be fixed parameter tractable. Studying how the complexity depends on k for various parameterized problems is crucial for applications in which small, fixed parameter values are important.

Downey and Fellows defined a hierarchy of parameterized decision problem classes, $FPT \subseteq W[1] \subseteq W[2] \subseteq \cdots$, with appropriate reducibility and completeness notions. *Vertex cover* is in FPT, while *maximum independent set* is $W[1]$-complete. More details about this can be found in (17; 1; 18; 19).

A bipartite graph is called *chordal bipartite* if every cycle of length strictly greater than 4 has a chord (i.e. an edge between non-adjacent vertices on the cycle). Chordal bipartite graphs were first introduced by Golumbic and Goss (26; 25). The original motivation came from appli-

cations to non-symmetric matrices. Some of the applications include Gaussian elimination in sparce matrices (25; 4), integer linear programming (34), and matrix analysis (36; 35). Chordal bipartite graphs are very closely related to totally balanced matrices. More details about this relationship appear in the next chapter. The properties and characterizations of chordal bipartite graph, and their recognition algorithms have also been widely studied. Characterizations for chordal bipartite graphs exist in terms of perfect edge elimination orderings (39; 9; 4), minimal edge separators (26), vertex elimination orderings (31), totally balanced hypergraphs (10), among others.

The *minimum bipartite fill-in* problem is the problem of finding an edge set of minimum cardinality which when added to the input bipartite graph, makes it chordal bipartite. The parameterized version of this problem asks if the input bipartite graph can be made chordal bipartite by adding at most k edges. We present an $O(8^{2k}m \log n)$ time algorithm for the fixed parameter version of this problem, where the input bipartite graph has n vertices and m edges. This shows that the minimum bipartite fill-in problem is fixed parameter tractable.

The rest of this chapter is organized as follows: The next section introduces basic concepts, definitions and notation. The parameterized algorithm is presented in Section 3. We analyze the complexity of our algorithm in Section 4. Concluding remarks appear in section 5.

2.2 Basic definitions and preliminaries

A *chord* in a cycle is an edge between non-consecutive vertices on the cycle. Let $H = (X, Y, E)$ be a bipartite graph. The set V of vertices of H is $V = X \cup Y$. In a bipartite graph, a cycle is called *chordless* if it is of length greater than 4 and contains no chord. A bipartite graph is called *chordal* if every cycle of length strictly greater than 4 has a chord.

Given a bipartite graph $H = (X, Y, E)$ and a set of edges, F, such that $H' = (X, Y, E \cup F)$ is chordal bipartite, then F is called a *bipartite fill-in* of H. The *minimum bipartite fill-in* problem is the problem of finding a bipartite fill-in of smallest size for the given bipartite graph. A bipartite fill-in F of a bipartite graph H is called *minimal* if no proper subset of F is a bipartite fill-in of H. We shall say that chordal bipartite graphs are *bi-quadrangulated*.

The parameterized version of this problem asks if there exists a bipartite fill-in with at most k fill-in edges for the given bipartite graph H, for some given k. We will use *bipartite-fill-in(k)* to refer to this parameterized problem and *k-bi-quadrangulation* to refer to a bi-quadrangulation with at most k edges.

Observe that in a bipartite graph all cycles are of even length. Hence, when talking about bipartite graphs, chordless cycles refer to cycles which have length at least 6 and have no chords. Also, if C is a chordless cycle of even length and a and b are two vertices on C, then, we can add an edge from a to b if and only if a path from a to b on C is of odd length. A path with l edges is called an *l-path* and its *length* is l. If p is a path then $|p|$ denotes its length. A single vertex is considered a 0-path. A cycle with l edges is called an *l-cycle*.

2.3 Parameterized algorithm

In this section we present a $O(8^{2k}m \log n)$ algorithm for the parameterized bipartite fill-in problem *bipartite-fill-in(k)* on bipartite graph H with n vertices and m edges. It is based on a simple search-tree algorithm. This familiar technique has been used to prove the fixed parameter tractability of several problems. In particular, our algorithm is built along the same lines as the algorithm for strongly chordal graph completion presented in (37).

The proof of the following lemma follows easily by induction.

Lemma 2.3.1 *A minimal bipartite fill-in of a chordless l-cycle of even length consists of* $\frac{l-4}{2}$ *chords, which partition the cycle into* $\frac{l-2}{2}$ *4-cycles. Any two of these 4-cycles are either disjoint or share a chord. Every chord in a minimal bipartite fill-in is shared by exactly two 4-cycles.*

An *odd* chord in a cycle of even length is a chord such that the paths connecting its end points on the cycle contain an odd number of edges. Clearly, an odd chord in a cycle of even length partitions this cycle into two smaller cycles, say C_1 and C_2, of even length. Any odd chord in C_1 or C_2 is an odd chord of the original cycle as well. A *4-cycle decomposition* of a chordless cycle of even length is a minimal set, say S, of odd chords in C such that there are no induced chordless cycles of length six or greater in $C + S$.

Lemma 2.3.2 *Given a chordless cycle C of even length, a set of edges S is a 4-cycle decomposition of C if and only if S is a minimal bipartite fill-in of C.*

Proof: Let A be a minimal bipartite fill-in of C. By definition, $C + A$ contains no induced chordless cycles of length greater than six. Also, since $C+A$ is bipartite all the edges in A must be odd chords in C. Hence, every minimal bipartite fill-in of C is also a 4-cycle decomposition of C.

To prove the converse, let B be a 4-cycle decomposition of C. By definition, $C+B$ contains no induced chordless cycles of length six or greater. Also, all chords in B are odd chords in C. Hence, $C + B$ is chordal bipartite. All 4-cycle decompositions are also minimal by definition. Hence, B is in fact a minimal bipartite fill-in of C. ■

A *ternary tree* is a tree in which every internal node has exactly three children. In light of Lemma 2.3.2, the following two lemmas are implicit in (37).

Lemma 2.3.3 *The number of distinct minimal bipartite fill-ins of a chordless l-cycle of even length is equal to the number of ternary trees with $\frac{l-2}{2}$ internal nodes.*

Lemma 2.3.4 *The number of distinct minimal bipartite fill-ins of a chordless l-cycle of even length is no greater than $8^{\frac{l-2}{2}}$.*

The algorithm will traverse part of a search tree in which each node represents a supergraph of the original input bipartite graph $H = (X, Y, E)$. The search tree is defined as follow:

- The root of the search tree corresponds to the input graph H.
- The children of an internal node, say x, are generated as follows. Let H' be the bipartite graph corresponding to node x. Find a chordless cycle, C, in H'. Let S be a minimal bipartite fill-in of this cycle C. Then the graph $H'' = (X, Y, E \cup S)$ corresponds to one of the children of node x. Each child of x corresponds to the bipartite graph obtained from different minimal bipartite fill-ins of cycle C.

Observation 1 *Each leaf of this tree corresponds to a chordal bipartite supergraph of H. And each minimal bi-quadrangulation of H is represented by at least one leaf in this tree.*

The algorithm will only visit those nodes in the search tree which correspond to supergraphs of H with no more than k additional edges. If one of these nodes is a leaf node then we have found a k-bi-quadrangulation, otherwise no such bi-quadrangulation exists.

2.4 Complexity analysis

A *bipartite adjacency matrix* of a bipartite graph $H = (X, Y, E)$ is the $|X| \times |Y|$ (0,1)-matrix $M = m_{ij}$ where $m_{ij} = 1$ if and only if $a_i b_j \in E$. A *cycle* matrix is a (0,1) $k \times k$ matrix with $k \geq 3$ in which each row and each column has exactly two non-zero entries, and which is minimal for this property. A (0,1)-matrix is *totally balanced* if it contains no cycle submatrices. It is known (3; 34; 31; 42) that a bipartite graph H is chordal bipartite if and only if the corresponding bipartite adjacency matrix is totally balanced. A *double lexical ordering* of a matrix is an ordering so that the rows and columns, as vectors, are lexicographically ordered from top to bottom and from left to right. Γ is defined to be an ordered (0,1)-matrix as follows:

$$\mathbf{\Gamma} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

From (3; 34; 40) we have the following two theorems.

Theorem 2.4.1 *A (0,1)-matrix has a Γ-free ordering if and only if it is totally balanced. Moreover, a doubly lexical ordering of a totally balanced matrix is Γ-free.*

Theorem 2.4.2 *In a doubly lexical (0,1)-matrix any Γ sub-matrix is embedded in a cycle submatrix.*

In a graph $G = (V, E)$, if $S \subseteq V$, let $G[S]$ denote the subgraph of G induced by vertices in S.

Lemma 2.4.1 *Let M be a bipartite adjacency matrix of a bipartite graph H, and let N be a $k \times k$ cycle submatrix of M with rows $r_1, \ldots, r_k$ and columns $c_1, \ldots, c_k$. Let $V_N = \{v_l \mid l = r_i$ or $l = c_j,\ i \leq i, j \leq k\}$. Then, the vertices in V_N induce a chordless cycle of even length in H.*

Proof: The vertices of V_N are the vertices that correspond to the rows and columns of the cycle submatrix. Observe that the vertices corresponding to the rows and columns in this cycle

submatrix are distinct. By the definition of a cycle submatrix, all the vertices in V_N in the graph $H[V_N]$ have degree 2. Moreover, by definition each cycle submatrix is minimal for its property. Hence, $H[V_N]$ only contains one cycle. Also, $|V_N| \geq 6$. Thus, the vertices in V_N induce a chordless cycle of even length in H. ■

In (40) Lubiw presents an algorithm that finds a double lexical ordering of any $p \times q$ matrix in $O(s \log^2 s)$ time, where $s = p + q + e$, and e is the number of non-zero entries in the matrix. Note that in a bipartite adjacency matrix, s is the sum of the number of vertices and edges in the bipartite graph. Paige and Tarjan (47) improve this running time to $O(s \log s)$. This implies that if n is the number of vertices and m the number of edges in the bipartite graph then we can obtain a doubly lexical ordering of the corresponding bipartite adjacency matrix in $O(m \log n)$ time. Lubiw (40) also shows how to search for a Γ submatrix in this doubly ordered matrix in $O(m)$ time. Once a Γ submatrix is found, a cycle submatrix that contains it can also be found in $O(m)$ time (40). By Lemma 2.4.1 we can extract a chordless cycle from the cycle submatrix in $O(n)$ time.

Thus, we can find a chordless cycle (if one exists) in a bipartite graph with n vertices and m edges in $O(m \log n)$ time.

The algorithm described in (50) can be easily extended to enumerate all ternary trees with n nodes, spending $O(n)$ time for each node. Hence, by applying Lemma 2.3.3 we can obtain an algorithm that enumerates all minimal bipartite fill-ins of any l-cycle of even length in $O(l)$ time per fill-in. Based on these observations, we have the following theorem.

Theorem 2.4.3 *All minimal k-bi-quadrangulations of a bipartite graph H with n vertices and m edges can be found in $O(8^{2k} m \log n)$ time.*

Proof: Let T denote the sub-tree of the search tree traversed by the algorithm. For any node $x \in T$, let $H_x = (V, E_x)$ denote the supergraph of H corresponding to node x, and d_x denote the maximum length of a path from node x to a leaf of T. Let $a_x = \max(\{|E_l| - |E_x| \mid \text{l is a leaf of the subtree of } T \text{ rooted at } x\})$ and let l_x denote the total number of leaf nodes in the subtree of T rooted at x.

We claim that $l_x \leq 8^{d_x+a_x}$. If this is true then it implies that the number of leaf nodes in T is bounded by 8^{2k}, and hence the total number of nodes in T must be less than $2 \cdot 8^{2k}$ i.e. $O(8^{2k})$. Since we spend $O(m \log n)$ time on each such node, the proof is complete.

We will use induction to prove our claim. Assume that the claim is true for all the children of a node x. Observe that the claim is indeed trivially true for all leaf nodes in T. Let l be the length of the cycle that is detected at x. Let $d_{max} = \max\{d_y |\ y \text{ is a child of } x\}$, and let $a_{max} = \max\{a_y |\ y \text{ is a child of } x\}$. Observe that $d_x = d_{max} + 1$, and $a_x = a_{max} + \frac{l-4}{2}$. By the induction hypothesis, the number of leaves in the subtree of T rooted at any child of x is bounded by $8^{d_x+a_x}$. By Lemma 2.3.4 we know that the number of children of x is bounded by $\frac{l-2}{2}$. Hence, the total number of leaves in the subtree of T rooted at x must be bounded by $8^{\frac{l-2}{2}} \cdot 8^{d_{max}+a_{max}} = 8^{d_{max}+1+a_{max}+\frac{l-4}{2}} = 8^{d_x+a_x}$. ■

Note that this algorithm may list the same bi-quadrangulation several times. If required, this issue can be easily remedied by storing the solutions in a suitable data structure and checking each new solution to see if it has already been found. Among these, a minimum bipartite fill-in is one which has the least number of fill-in edges.

2.5 Conclusion

We have presented an algorithm with $O(8^{2k}m \log n)$ time complexity for the fixed parameter version of the minimum bipartite fill-in problem. This algorithm shows that minimum bipartite fill-in is fixed parameter tractable. It also shows that when k is restricted to be at most logarithmic in the size of the graph, we can compute the minimum bipartite fill-in in polynomial time.

The minimum fill-in problem is known to be NP-complete (54) but the complexity status of problems of computing a minimum bipartite fill-in remains unknown. There is a wealth of information known about chordal graph completion problems. Several characterizations and algorithms are known for computing minimal fill-ins. Not much is known about minimal bipartite fill-ins. Chordal bipartite graphs have nice structural properties and several NP-complete problems can be solved in polynomial time when restricted to this graph class. We

believe that studying the structure of various fill-in problems related to this and related graph classes is an interesting and useful direction for future research.

CHAPTER 3. Heuristics for the gene-duplication problem: An $\Omega(n)$ speed-up for the local search

3.1 Introduction

The rapidly increasing amount of available genomic sequence data provides an abundance of potential information for phylogenetic analyses. Most phylogenetic analyses combine genes from presumably orthologous loci, or loci whose homology is the result of speciation. These analyses largely neglect the vast amounts of sequence data from gene families, in which complex evolutionary processes such as gene duplication and loss, recombination, and horizontal transfer generate gene trees that differ from species trees. One approach to utilize the data from gene trees (gene families) in phylogenetics is to reconcile the gene trees with species trees based on an optimality criterion, such as the Gene Duplication model introduced by Goodman et al. (27). This problem is type of a *supertree problem*, that is, assembling from a set of input trees (the gene trees) a species supertree that contains all species found in at least one of the input trees. The decision version of the gene-duplication problem is NP-complete (41). Existing heuristics aimed at solving the gene-duplication problem search the space of all possible supertrees guided by a series of exact solutions to instances of a local search problem (45). The gene-duplication problem has shown much potential for building phylogenetic trees for snakes (49), vertebrates (46); (15), *Drosophia* (14), and plants. Yet, the runtime performance of existing heuristics has limited the size of such studies. We improve on the best existing solution for the local search problem asymptotically by a factor of n, where n is the number of species from which sequences in the gene trees were sampled (that is the number of nodes in a resulting supertree). To show the applicability of our improved solution for the local search problem, we implemented it as part of standard heuristics for the gene-duplication problem. We demonstrate that the

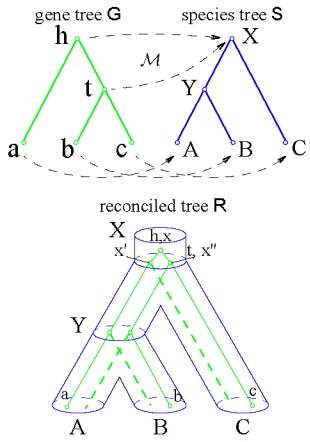

Figure 3.1 Upper part: Trees G and S are comparable, as the mapping from the leaf-genes to the leaf-species indicates. $\mathcal{M}$ is the lca-mapping from G to S. Lower part: R (green) is the reconciled tree where gene-duplication x copies into the genes x' and x'' in species X. The solid green lines in the reconciled tree R represent the embedding of gene tree G into R.

implementation of our method greatly improves the speed of heuristics for the gene-duplication problem and makes it possible to infer large supertrees that were previously difficult, if not impossible, to compute.

For convenience, we use the term "tree" to refer to a rooted and full-binary tree. The terms "leaf-gene" and "leaf-species" refer to a gene or species that is represented by a leaf of a gene or species tree respectively throughout this work unless otherwise stated.

Previous Results: The gene-duplication problem is based on the Gene Duplication model from Goodman et al. In the following, we (i) describe the Gene Duplication model, (ii) formulate the gene-duplication problem, and (iii) describe a heuristic approach of choice (45) to solve the gene-duplication problem.

Gene Duplication model: The Gene Duplication (GD) model (44; 29; 43; 21; 55; 12; 7; 28) explains incompatibilities between a pair of "comparable" gene and species trees through gene duplications. A gene and a species tree are *comparable*, if a sample mapping, called *s-mapping*, exists that maps every leaf-gene to the leaf-species from which it was sampled. Figure 3.1 depicts an example. Gene tree G is inferred from the leave-genes that were sampled

from the leaf species of the species tree described by the s-mapping. However, both trees describe incompatible evolutionary histories. The GD model explains such incompatibilities by reconciling the gene tree with postulated gene duplications. For example, in Figure 3.1 a reconciled gene tree R can be theoretically inferred from the species tree S by duplicating a gene x in species X into the copies x' and x'' and letting both copies speciate according to the topology of S. In this case, the gene tree can be embedded into the reconciled tree. Thus, the gene tree can be reconciled by the gene duplication x to explain the incompatibility. The gene duplications that are necessary under the GD model to reconcile the gene tree can be described by the *lca-mapping* $\mathcal{M}$, which is an extension of the given s-mapping. $\mathcal{M}$ maps every gene in the gene tree to the most recent species in the species tree that could have contained the gene. To make the definition precise, $\mathcal{M}$ maps each gene to the least common ancestor of the species from which the leaf-genes of the subtree rooted at the gene were sampled (given by the s-mapping). A gene in the gene tree is a *gene duplication* if it has a child with the same lca-function. In Figure 1 gene h and its child t map under the lca-function to the same species X. The *reconciliation cost* for gene tree and a comparable species tree is measured in the number of gene duplications in the gene tree induced by the species tree. [1] The *reconciliation cost* for a given set of gene trees and a species tree is the sum of the reconciliation cost for every gene tree in the set and the species tree. The lca-function is linear time computable on a PRAM (55) through a reduction from the least common ancestor problem (5; 32). Hence, the reconciliation cost for a set of gene trees and a species tree is computable in linear time.

Gene-duplication problem and heuristic: The *gene-duplication problem* is to find for a given set of gene trees a comparable species tree with the minimum reconciliation cost. The decision variant of this problem and some of its characterizations are NP-complete (41; 23) while some parameterizations are fixed parameter tractable (51; 30). Therefore, in practice, heuristics are commonly used for the gene-duplication problem even if they are unable to guarantee an optimal solution. However, GeneTree (Page), an implementation of a standard local

[1] Alternatively, the reconciliation cost could be defined in the number of gene duplications and losses. However, it is extremely difficult to accurately infer gene losses if there is missing data. Thus, for this study we only consider gene duplications.

search heuristic for the gene-duplication problem, was used to show that the gene-duplication problem can be an effective approach. While the local search heuristic for the gene-duplication problem performs reasonably well in computing smaller sized instances, it does not allow the computation of larger species supertrees. In this heuristic, a *tree graph* is defined for the given set of gene trees and some fixed tree edit operation. The nodes in the tree graph are the species trees which are comparable with every given gene tree. An edge is drawn between two nodes exactly if the corresponding trees can be transformed into each other by the tree edit operation. The *reconciliation cost* of a node in the graph is the reconciliation cost of the species tree represented by that node and the given gene trees. Given a starting node in the tree graph, the heuristic's task is to find a maximal-length path of steepest descent in the reconciliation cost of its nodes and to return the last node on such a path. This path is found by solving the *local search problem* for every node along the path. The local search problem is to find a node with the minimum reconciliation cost in the neighborhood of a given node. The time complexity of the local search problem depends on the tree edit operation used. An edit operation of interest is the rooted subtree pruning and regrafting (rSPR) operation (2; 8). Given a tree S, an rSPR operation can be performed in three steps: (i) prune some subtree P from S, (ii) add a root edge to the remaining tree S, (iii) regraft P into an edge of the remaining tree S. An example is depicted in Figure 3.2. The resulting tree graph is connected and every node has a degree of $\Theta(n^2)$, where n is the size of a species tree comparable to the given gene trees. Thus the local search problem for the rSPR edit operation can be solved naively in $\Theta(n^3)$ time (assuming the mapping from a gene tree to the species tree can be constructed in $\Theta(n)$ time). This is the best-known algorithm to solve the local search problem for rSPR operations. In practice, the cubic running time typically allows only the computation of smaller supertrees (45). A common approach to overcome this limitation is to consider only an $O(n)$ cardinality subset of the rSPR neighborhood at each node by using the rooted nearest neighbor interchange (rNNI) edit operation. The local search problem for the rNNI edit operation can be solved in $O(n^2)$ time. We show how to solve the local search problem for the rSPR edit operations within the same $O(n^2)$ time bound.

Contribution of this manuscript: First we introduce an algorithm that solves the local search problem for the rSPR tree edit operation in $O(n^2)$ time, where n is the size of any species tree resulting from the given gene trees. Our algorithm was implemented as part of a standard heuristic for the gene-duplication problem, and we compared the running times of our implementation and the program GeneTree, which can infer species trees using the same gene-duplication heuristic. Finally, we demonstrate the ability of our heuristic to utilize gene-family sequences to construct large subtrees of the Tree of Life.

The rest of this chapter is organized as follows: Section 2 introduces basic terminology and problem definitions. In Section 3 we formally introduce the local search problem for the rSPR tree edit operation and our approach for solving it. To solve this refined local search problem we study gene duplication properties when a tree is modified using rSPR operations in Section 4. In Section 5 we introduce our algorithm for the (refined) local search problem, show its correctness and analyze its running time. Experimental results are presented in Section 6 and concluding remarks appear in Section 7.

3.2 Basic notation and preliminaries

We borrow some the following notation from (30). Given a rooted tree T, let $V(T)$ and $E(T)$ denote the node (vertex) set and edge set of T respectively. $\mathsf{Root}(T)$ represents the root node of tree T and $\mathsf{Le}(T)$ denotes the leaf set of T. The set of internal nodes of T is given by $V(T) \setminus \mathsf{Le}(T)$. T_v denotes the complete subtree of T rooted at node $v \in V(T)$. Given a node $v \in V(T)$, if $u \in V(T_v)$ and $u \neq v$, then u is called a *(proper) descendant* of $v \in V(T)$. A descendant u of node $v \in V(T)$ is called a *child* of v if $(v, u) \in E(T)$. Let $\mathsf{Ch}_T(v)$ denote the set of children of node $v \in V(T)$. If u is a child of node v then v is called the *parent* of u, denoted by $\mathsf{Pa}_T(u)$. Two nodes that have the same parent are called *siblings* of each other. Given a set $L \subseteq \mathsf{Le}(T)$, the *least common ancestor* of L in T is defined to be the node $v \in V(T)$ such that $L \subseteq \mathsf{Le}(T_v)$ but $L \not\subseteq \mathsf{Le}(T_u)$ for any descendant u of $v \in V(T)$. In the following we motivate, explain and formally define the terms comparable, lca-mapping, gene duplication, and the gene-duplication problem that were introduced in the introduction.

A *species tree* is a tree that depicts the evolutionary relationships of a set of species. Given a gene (or gene family) for a set of species, a *gene tree* is a tree that depicts the evolutionary relationships among the sequences encoding only that gene (or gene family) in the given species. By definition, species trees and gene trees are leaf labeled. In addition, while a given leaf label may occur only once in a species tree, it can occur multiple times in a gene tree. We limit our attention to gene trees and species trees which are rooted and fully binary. Note that the leaves of species trees as well as gene trees correspond to biological species. Based on this notion of correspondence between the leaves of gene trees and species trees, we have the following definition.

Definition 3.2.1 (Comparable) *A gene tree G and a species tree S are said to be* comparable *if* $\mathsf{Le}(G) \subseteq \mathsf{Le}(S)$. *Given a set $\mathcal{G}$ of gene trees and a species tree S, S is said to be* comparable *with $\mathcal{G}$ if* $\mathsf{Le}(S) = \bigcup_{G \in \mathcal{G}} \mathsf{Le}(G)$.

A common strategy for constructing a species tree is to construct gene trees for a set of distinct gene families and then inferring the species tree from these gene trees. The initial assumption is that the genes evolve in the same way as species, however, it is observed that gene trees can differ from the actual species tree. This is the reason why several different gene trees are used to construct the species tree. Several evolutionary phenomena have been used to explain this difference. In particular, the *gene duplication* model seeks to explain this difference through *gene duplications*. Gene duplications are common evolutionary events which result in multiple copies of a gene located along a DNA strand. These copies then evolve independently of each other. In order to relate/reconcile a gene tree to a (comparable) species tree, the gene duplication model introduces a mapping from the nodes of the gene tree to the nodes of the species tree. Each internal node, say a, in the gene tree represents an ancestral node, and it is associated by the mapping to the most recent ancestral species in the species tree that contains all contemporary genes descending from a. This mapping can be computationally modeled as a least common ancestor mapping in the species tree.

Definition 3.2.2 ((lca-)mapping) *Let G be a gene tree and S a comparable species tree. The*

mapping $\mathcal{M}_{G,S}\colon V(G) \to V(S)$ *is defined such that* $\mathcal{M}_{G,S}(v)$ *is the least common ancestor of* $\mathrm{Le}(G_v)$ *in* S.

Definition 3.2.3 ((Gene) duplication) *Given the mapping* $\mathcal{M}_{G,S}\colon V(G) \to V(S)$, *a node* $v \in V(G)$ *is a* duplication *if there exists a child* u *of* $v \in V(G)$ *such that* $\mathcal{M}_{G,S}(v) = \mathcal{M}_{G,S}(u)$.

Definition 3.2.4 (Reconciliation cost) *The* reconciliation cost, $\Delta(G,S)$, *of* G *and* S *is the number of duplications needed to explain the gene tree* G *under the species tree* S. *Formally,* $\Delta(G,S) = |\{v\colon v \in V(G) \text{ and } v \text{ is a duplication}\}|$. *If* $\mathcal{G}$ *denotes a set of gene trees, then* $\Delta(\mathcal{G},S) = \sum_{G\in\mathcal{G}} \Delta(G,S)$. *If* $\mathcal{S}$ *is the set of all species trees comparable with* $\mathcal{G}$, *then* $\Delta(\mathcal{G}) = \min_{S\in\mathcal{S}} \Delta(\mathcal{G},S)$.

The gene duplication problem is the problem of finding a species tree that requires the minimum number of postulated duplications. More formally:

Definition 3.2.5 (DUPLICATION problem) *Given a set* $\mathcal{G}$ *of gene trees, the* DUPLICATION *problem is the problem of finding a species tree* S_{OPT} *comparable with* $\mathcal{G}$ *and which has reconciliation cost* $\Delta(\mathcal{G})$.

3.3 Refining the local search problem

Local search heuristics for the DUPLICATION problem search the tree space guided by a series of stepwise solutions to a local search problem. The main idea is to repeatedly try to find a species tree which has a lower reconciliation cost than the previous tree. The heuristic terminates when a better solution cannot be found in the neighboring search space. The LOCAL SEARCH problem is to find a tree with minimum reconciliation cost in the neighborhood of a given species tree. The *neighborhood* of a species tree is the set of trees into which the species tree can be transformed by one tree edit operation. Rooted nearest neighbor interchange (rNNI) and rooted subtree pruning and regrafting (rSPR) are important examples of tree perturbation operations. As seen in the Introduction, the neighboring search space defined using rSPR operations is much larger than, and a superset of, the neighboring search space

defined using rNNI operations. In this work we use the rooted subtree pruning and regrafting (rSPR) edit operation. We will solve the LOCAL SEARCH problem by finding the reconciliation cost for every tree in the neighborhood efficiently. Therefore, we first formally define the rSPR operation. Then we formulate the neighborhood-search problem as follows: Given a set of gene trees $\mathcal{G}$ and a comparable species tree S, finds the reconciliation cost of every tree in the rSPR neighborhood of S. Finally, to efficiently solve the neighborhood-search problem we divide it into sub-problems, each of which is a restricted-neighborhood-search problem. In Sections 4 and 5 we provide an efficient solution for this restricted problem.

Informally, the basic idea of an rSPR operation is as follows (see Figure 3.2). The subtree to be pruned along with its root edge is first detached from the original tree. This root edge is then merged back into some edge of the remaining tree. This forms a new tree whose leaf set is the same as the original tree. More formally, we have the following:

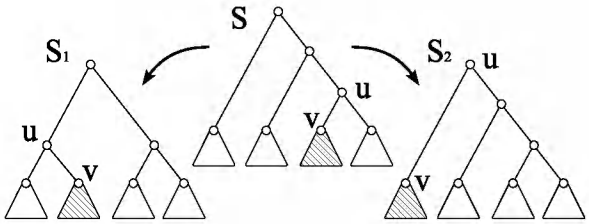

Figure 3.2 S_1 and S_2 are obtained from S by pruning the subtree rooted at v and regrafting it into the remaining tree S.

Definition 3.3.1 (rSPR operation) *A* rooted subtree prune and regraft (rSPR) *operation on a rooted binary tree S is defined as cutting any edge, say (u, v), where $u = \mathsf{Pa}_S(v)$, and thereby pruning a subtree, S_v, and then regrafting the subtree by the same cut edge in one of the following two ways:*

1. *Creating a new node u' which subdivides an edge in $S \backslash S_v$, and regrafting the subtree by the cut edge at node u'. Then, either suppressing the degree-two node u or, if u is the root of S, deleting u and the edge incident with u, making the other end-node of this edge the new root.*

2. *Creating a new root node u' and a new edge between u' and the original root. Then regrafting the subtree by the cut edge at node u' and suppressing the degree-two node u.*

Note that throughout this text we assume that the new node created, u', is given the same label as the node removed, u. Hence, after the rSPR operation is performed according to the scenario described above, the parent of node v will still be named u.

Definition 3.3.2

1. *Define $\mathfrak{rSPR}(S, v, u)$ to be the tree obtained by performing an rSPR operation on the rooted binary tree S, such that the subtree rooted at $v \in V(S)$ is pruned and appears regrafted on the edge $(u, \mathsf{Pa}_{S'}(u))$ where $S' = \mathfrak{rSPR}(S, v, u)$.*

2. $\mathfrak{rSPR}(S, v) = \bigcup_{u \in V(S) \setminus V(S_v)} \mathfrak{rSPR}(S, v, u)$.

3. $\mathfrak{rSPR}(S) = \bigcup_{v \in V(S) \setminus \{\mathsf{Root}(S)\}} \mathfrak{rSPR}(S, v)$.

In other words, $\mathfrak{rSPR}(S)$ is defined to be the set of all rooted binary trees that can be obtained by performing one rSPR operation on the rooted binary tree S, and $\mathfrak{rSPR}(S, v)$ is defined to be the set of all rooted binary trees that can be obtained by performing rSPR operations on the rooted binary tree S and only pruning the subtree rooted at node $v \in V(S)$.

Definition 3.3.3 (NS problem) *The* NEIGHBORHOOD-SEARCH (NS) *problem is defined to be the problem of finding the reconciliation costs of all species trees in $\mathfrak{rSPR}(S)$, where S is a given species tree.*

We will show how to solve the NS problem without having to separately compute the reconciliation cost for each tree in the neighboring search space. This gives us an algorithm that is $\Omega(n)$ times faster than existing algorithms for the NS problem, where n is the number of leaves in the species tree. This in turn also speeds up heuristics for the DUPLICATION problem by a factor of at least n.

Our faster algorithm is based on the observation that if we traverse through $\mathfrak{rSPR}(S)$ in a particular way, we can reuse some of the previously computed information to obtain the reconciliation costs efficiently. We introduce this special traversal strategy in the next subsection.

3.3.1 The restricted-neighborhood-search problem

To develop a faster algorithm for the NS problem we first define a restricted version of this problem.

Definition 3.3.4 (RNS problem) *Given the input tree S and a subtree P of S, we define the* RESTRICTED-NEIGHBORHOOD-SEARCH *(RNS) problem as the problem of finding the reconciliation costs of all trees in $\mathfrak{rSPR}(S, \mathsf{Root}(P))$.*

Observation 2 *The NS problem on S can be solved by solving the RNS problem for each subtree of S. More formally, $\mathfrak{rSPR}(S) = \bigcup_{v \in V(S) \setminus \{\mathsf{Root}(S)\}} \mathfrak{rSPR}(S, v)$.*

The main idea of our algorithm is that if we find the reconciliation cost of a particular tree in $\mathfrak{rSPR}(S, \mathsf{Root}(P))$ for a given species tree S and its subtree P, then we can find the cost of all the remaining trees in $O(1)$ time per tree.

Let S_{root} denote the tree after P is pruned and regrafted to the root in S, as shown in Figure 3.3. Clearly, $S_{root} \in \mathfrak{rSPR}(S, \mathsf{Root}(P))$. Note that any tree in $\mathfrak{rSPR}(S, \mathsf{Root}(P))$ can be obtained by pruning P from S_{root} and regrafting the pruned subtree to the proper edge below. This regrafting procedure can also be performed step by step by regrafting into the edges along the path from the root node to the required node in S_{root}. During each of these steps we regraft the pruned subtree into the next edge along the path. This involves regrafting the pruned subtree into an edge immediately below the current position. Therefore it makes sense to define the following.

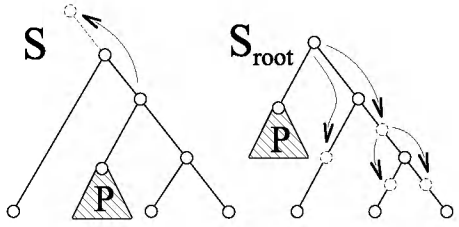

Figure 3.3 This figure depicts how each tree in $\mathfrak{rSPR}(S, \mathsf{Root}(P))$ can be obtained by starting from S_{root} and successively performing move–down operations.

Definition 3.3.5 (Move-down operation) *Given a fully binary tree S, and its subtree P to be pruned, we define the* move-down *operation as the rSPR operation which produces the tree $\mathsf{rSPR}(S, \mathsf{Root}(P), c)$, where u is the sibling of $\mathsf{Root}(P)$, and $c \in \mathsf{Ch}_S(u)$. In particular, if c is the left child of u, then the operation is called* move-down-left*, and* move-down-right *otherwise.*

Definition 3.3.6 (*movedown*(S, P)) *movedown(S, P) is defined to be the set of all species trees that can be obtained by performing successive move-down operations starting from S_{root} with a fixed pruned subtree P.*

Observation 3 *Each tree in $\mathsf{rSPR}(S, \mathsf{Root}(P))$ can be obtained by starting from S_{root} and successively performing move-down operations. Formally, $\mathsf{rSPR}(S, \mathsf{Root}(P)) = movedown(S, P) \bigcup \{S_{root}\}$. See Figure 3.3.*

Therefore, it is of interest to study how the reconciliation cost is affected when a move-down operation is performed. In the next section we study some of the structural properties of trees obtained by performing rSPR operations. These properties allow the design of an efficient algorithm for the RNS problem. In particular, we describe how the tree and its reconciliation cost changes when rSPR operations are performed in a predefined order.

3.4 Structural properties

In this section we study the change in the duplication status of nodes in a gene tree when mode-down operations are performed on the species tree. To do this we will first look at the changes in the mapping from a gene tree to the species tree when the species tree is modified using move-down operations.

Given a gene tree G and a comparable species tree S, let P be the subtree of S to be pruned. Let p denote the root node of P, let $x = \mathsf{Pa}_S(p)$ and y the sibling of p. Let $Q = S_y$. Given $\mathcal{M}_{G,S}$, let $\mathcal{M}^{-1}_{G,S}(v)$ denote the set of nodes in G that map to node v in S under the mapping $\mathcal{M}_{G,S}$. Let S' be the tree obtained from S by regrafting P according to a move-down-right operation. This situation is depicted in Figure 3.4. In other words, S' is the species tree obtained by moving the node x along with subtree P such that x now becomes a child of y.

$\mathcal{M}_{G,S'}$ denotes the mapping from G to S'. Lemmas 3.4.1 and 3.4.2 provide more insight into the manner in which the mapping from G is affected in this situation.

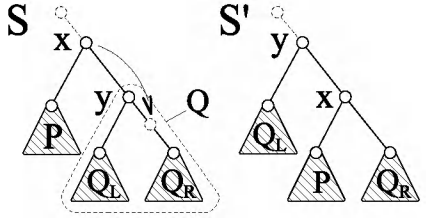

Figure 3.4 The subtree on the right, S', is obtained from S by moving x and P to the right subtree of y.

Lemma 3.4.1 $\mathcal{M}^{-1}_{G,S}(v) = \mathcal{M}^{-1}_{G,S'}(v)$, $\forall v \in V(S)\backslash\{x, y\}$.

Proof: Suppose node $\alpha \in V(G)$ maps to node $\beta \in V(S)$ under the mapping $\mathcal{M}_{G,S}$. If $\mathsf{Le}(S_\beta) = \mathsf{Le}(S'_\beta)$, then by the definition of a mapping, node α must map to node $\beta \in V(S')$ under mapping $\mathcal{M}_{G,S'}$. Observe that $\forall v \in V(S)\backslash\{x, y\}$, $\mathsf{Le}(S_v) = \mathsf{Le}(S'_v)$. The correctness of the lemma follows. ■

Lemma 3.4.2 $\mathcal{M}^{-1}_{G,S'}(x) \subseteq \mathcal{M}^{-1}_{G,S}(x)$ *and* $\mathcal{M}^{-1}_{G,S}(y) \subseteq \mathcal{M}^{-1}_{G,S'}(y)$.

Proof: Observe that $\mathsf{Le}(S'_x) \subset \mathsf{Le}(S_x)$ and $\mathsf{Le}(S_y) \subset \mathsf{Le}(S'_y)$. Based on this observation and the definition of LCA mapping, the lemma follows easily. ■

Some of the nodes from G that were duplications under the mapping $M_{G,S}$ may no longer be duplications under the mapping $M_{G,S'}$, and vice versa. This change in duplication status is directly linked to the change in mappings. Based on the observations from Lemmas 3.4.1 and 3.4.2, the following three lemmas characterize the possible change in the duplication status of nodes in G.

Lemma 3.4.3 *The duplication status for any node in G that does not map to x under mapping* $\mathcal{M}_{G,S}$ *remains unchanged.*

Proof: From Lemma 3.4.1 we know that the mappings from G into all nodes except x and y remain unchanged when S is changed into S'. This trivially implies that the duplication status

of any node that does not map to x or y under mapping $\mathcal{M}_{G,S}$ remains unchanged. To prove the lemma it is enough to show that the duplication status of any node mapping to node y under mapping $\mathcal{M}_{G,S}$ also remains unchanged.

Consider some node $v \in \mathcal{M}^{-1}_{G,S}(y)$. There are two possible cases:

Case 1: Node v is a duplication: By Lemma 3.4.2 we know that $\mathcal{M}^{-1}_{G,S}(y) \subseteq \mathcal{M}^{-1}_{G,S'}(y)$. This means that v must continue to be a duplication under mapping $\mathcal{M}_{G,S'}$ as well.

Case 2: Node v is not a duplication: This means that none of the children of v in G map to y under mapping $\mathcal{M}_{G,S}$. Consider the node $\alpha \in \mathsf{Ch}_G(v)$. Under mapping $\mathcal{M}_{G,S}$, α must map to a node which is a descendant of y in S. From Lemma 3.4.1 we know that node α will also map to the same node in S' under the new mapping $\mathcal{M}_{G,S'}$. Since this is true for both children of node v, it is not possible for v to become a duplication under mapping $\mathcal{M}_{G,S'}$.

Thus, the duplication status of v does not change in either of these cases. ■

Lemma 3.4.4 *If a node $v \in \mathcal{M}^{-1}_{G,S}(x)$ is not a duplication under mapping $\mathcal{M}_{G,S}$ then it becomes a duplication under mapping $\mathcal{M}_{G,S'}$ if and only if one of the children of v maps to node y in S.*

Proof: If node v is not a duplication then it means that none of its children in G map to node x under the mapping $\mathcal{M}_{G,S}$. Suppose $\alpha \in \mathsf{Ch}_G(v)$ and $\mathcal{M}_{G,S}(\alpha) = y$. Clearly the other child must map to a node in P. Observe that $\mathcal{M}_{G,S'}(v) = y$, and by Lemma 3.4.2 we know that $\mathcal{M}_{G,S'}(\alpha) = y$. Thus, v must be a duplication under the mapping $\mathcal{M}_{G,S'}$.

Now, suppose α does not map to y under mapping $\mathcal{M}_{G,S}$. Then α must map to a descendant of y in S. By Lemma 3.4.2, under the mapping $\mathcal{M}_{G,S'}$, α will still map to the same node in S'. For the same reason, the other child of v also keeps mapping to a node in P under the mapping $\mathcal{M}_{G,S'}$. In this situation, we have $\mathcal{M}_{G,S'}(v) = y$. However, none of the children of node v map to y under this mapping. Hence, v cannot become a duplication in this case. ■

Lemma 3.4.5 *Suppose $v \in \mathcal{M}_{G,S}^{-1}(x)$ is a duplication under mapping $\mathcal{M}_{G,S}$, and let b, c be its two children in G. Then node v will lose its duplication status under mapping $\mathcal{M}_{G,S'}$ if and only if:*

1. *Exactly one of b or c, say b, maps to x under $\mathcal{M}_{G,S}$,*
2. *The other node ,c, maps to a node in the left subtree of node y under $\mathcal{M}_{G,S}$ and,*
3. *Node b maps to x in S' under mapping $\mathcal{M}_{G,S'}$.*

Proof: Node v loses its duplication status under mapping $\mathcal{M}_{G,S'}$. This implies that $\mathcal{M}_{G,S'}(v) = y$, because if v keeps mapping to x, then its child b must also map to x under mapping $\mathcal{M}_{G,S'}$, which would make v a duplication. By the precondition we know that node c maps to a node in the left subtree of y under mapping $\mathcal{M}_{G,S}$. By Lemma 3.4.1 we can say that c must map to the same node under mapping $\mathcal{M}_{G,S}$ as well. Also, b maps to node x under the new mapping. Hence v which maps to y under the new mapping can not be a duplication in this case. This proves one direction in the proof.

Now, we know that $\mathcal{M}_{G,S'}(v) = y$. If node v is not a duplication under this mapping then neither b nor c must map to node y in S'. Node b may map to either x or y under mapping $\mathcal{M}_{G,S}$, but if it maps to y, then node v would be a duplication in this case. Hence, $\mathcal{M}_{G,S}(b) = x$. Now, if $\mathcal{M}_{G,S}(c) = y$ then node v would be a duplication under the new mapping. If $\mathcal{M}_{G,S}(c) = x$ then node c must map to x under the new mapping (otherwise node v would be a duplication). But in this case node v would also map to x because both its children map to x. If node c maps to a node in the right subtree of y under mapping $\mathcal{M}_{G,S}$, then again v would map to node x under the new mapping. Thus, c cannot map to any of these nodes. Hence, node c must map to a node in the left subtree of y under mapping $\mathcal{M}_{G,S}$.

■

Note that in Lemma 3.4.5, S' was obtained by moving node x along with the pruned subtree to the right subtree of node y. The case when S' is obtained by moving x and the pruned subtree to the left subtree of y is symmetric.

Lemmas 3.4.1 and 3.4.2 showed how the mappings are affected when a move-down operation is performed. Recall that S_{root} denotes the tree after the given subtree, P, is pruned and regrafted to the root in species tree S. In particular, we always start our sequence of move-down operations starting from the tree S_{root}. In the next subsection we will study the exact effect on the mapping from G when move-down operations are performed on S_{root} and the given subtree P.

3.4.1 Partial gene tree

Based on Lemmas 3.4.1 and 3.4.2 we will now show how the mapping from gene tree G behaves when move-down operations are successively performed starting from S_{root} to obtain a tree in $movedown(S, P)$.

Consider the situation as shown in Figure 3.3. The pruned subtree P of species tree S has been regrafted to the root, resulting in the tree S_{root}. Let G be a gene tree. Note that $x = \mathsf{Root}(S_{root})$. Let q denote the other child of x. And let Q be the subtree of S_{root} rooted at q. We would like to know how the mapping from G changes as we perform move-down operations starting from the tree S_{root}.

Lemma 3.4.6 *Given the mapping $\mathcal{M}_{G,S_{root}}$, among all nodes in G, only the mappings from nodes in the node set $\mathcal{M}^{-1}_{G,S_{root}}(x)$ may change when we successively perform move-down operations starting from tree S_{root}.*

Proof: This follows easily from Lemmas 3.4.1 and 3.4.2. ■

Definition 3.4.1 (Supporting nodes) Supporting nodes *are exactly those nodes of G, whose parent is in $M^{-1}_{G,S_{root}}(x)$, and which map to some node in Q under mapping $\mathcal{M}_{G,S_{root}}$.*

Definition 3.4.2 (Γ) Γ *is defined as the graph induced by G on the node set $\mathcal{M}^{-1}_{G,S_{root}}(x)$ and the supporting nodes in G.*

Note that the nodes from G whose mapping may change according to Lemma 3.4.6 are exactly the non-supporting nodes in Γ.

Lemma 3.4.7 Γ *is a tree.*

Proof: Γ is a subgraph of G by construction. Clearly, Γ is connected if the nodes in $\mathcal{M}^{-1}_{G,S_{root}}(x)$ are connected in G. For any node $u \in \mathcal{M}^{-1}_{G,S_{root}}(x)$ its parent $\mathtt{Pa}_G(u)$ must be in $\mathcal{M}^{-1}_{G,S_{root}}(x)$, because the leaf set of G_u is a subset of the leaf set of $G_{P_G(u)}$. Hence, Γ must be a connected subgraph of G, i.e. Γ must be a tree. ■

Lemma 3.4.8 *All the leaf nodes in* Γ *are supporting nodes.*

Proof: Suppose one of the leaf nodes of Γ, say v, is not a supporting node. Hence, $\mathcal{M}_{G,S_{root}}(v) = x$. This is only possible in two cases:

Case 1: A child of node $v \in \Gamma$ maps to x in S_{root}: This is not possible because by our assumption $v \in \mathtt{Le}(\Gamma)$.

Case 2: One child of node v maps to a node in P and the other child to a node in Q: This is not possible because the child node which maps to Q would be a part of Γ by definition, and hence v could not be a leaf node in Γ.

Thus, we have a contradiction. ■

Thus, Γ is a *partial gene tree* obtained from G. Note that Γ may not be a fully binary tree. The supporting nodes in Γ map to nodes in Q under mapping $\mathcal{M}_{G,S_{root}}$. Let this define an initial mapping from the leaves of Γ to the nodes in Q. Based on this initial mapping we can now construct the mapping $\mathcal{M}_{\Gamma,Q}$.

Recall that according to our strategy each tree in $movedown(S, P)$ is obtained by performing successive move-down operations starting from the tree S_{root}. Also recall that according to our naming convention, the root node of S_{root}, x, will always remain the parent node of $\mathtt{Root}(P)$. Thus, in all trees in $movedown(S, P)\backslash\{S_{root}\}$ the root node is $\mathtt{Root}(Q)$. The trees in $movedown(S, P)\backslash\{S_{root}\}$ are exactly the trees that are obtained by regrafting P into an edge in Q.

Based on the mapping $\mathcal{M}_{\Gamma,Q}$, we have the following lemma.

Lemma 3.4.9 *Let a be an internal (i.e. non-supporting) node in Γ which maps to some node, say j, in Q under the mapping $\mathcal{M}_{\Gamma,Q}$. Then we have the following:*

1. *If the pruned subtree is regrafted on an edge along the path from $\mathtt{Root}(Q)$ to node j in S_{root}, node $a \in G$ will keep mapping to node x.*
2. *If the pruned subtree is regrafted into an edge in the subtree rooted at j in S_{root}, then the node a in G maps to node j in the new tree.*
3. *For all other regraft locations, node a in G will map to some node along the path from $\mathtt{Root}(S_{root})$ to j, but not x and j.*

Proof: We have, $\mathcal{M}_{\Gamma,S_{root}}(a) = j$ and $\mathcal{M}_{G,S_{root}}(a) = x$. This means that:

- At least one descendant of $a \in G$ maps to a node in P under mapping $\mathcal{M}_{G,S_{root}}$, and at least one descendant maps to a node in Q_j.
- Under mapping $\mathcal{M}_{G,S_{root}}$, all the descendants of $a \in G$ map to either node x, or a node in P, or to a node in the subtree Q_j. This follows from the definition of Γ.
- No matter where P is regrafted, the mapping from node $a \in G$ can never go below the node j in the resulting tree.

Observe that these three properties remain independent of the location where P is regrafted. Based on this, all three parts of the lemma follow easily. ■

The end goal for the lemmas and observations seen so far is to be able to solve the NS problem efficiently. In the next section we describe an algorithm that can be used to efficiently solve the RNS problem. As seen in Observation 2, this lets us solve the NS problem efficiently as well.

3.5 Description of the algorithm

Based on the results obtained in the previous section we will now show how the RNS problem can be solved efficiently. Algorithm 1 describes this efficient algorithm to solve the RNS problem.

The input for Algorithm 1 is a gene tree G, a species tree S and a subtree P of S to be pruned. The first step in the algorithm is to obtain the tree S_{root}. Let $x = \mathsf{Root}(S_{root})$. The root of P is one child of x and let q denote the other child. And let Q be the subtree of S_{root} rooted at q.

The output Ω is a node weighted version of tree Q. The weight of a node u in tree Ω, represented by $\Omega(u)$ gives the reconciliation cost for G when the pruned subtree is regrafted onto the edge $(u, \mathsf{Pa}(u))$ in Q.

Algorithm 1 RECONCILIATION-COST-TREE

Input: G, S, P
Output: Ω
Construct the tree S_{root}. Let $x = \mathsf{Root}(S_{root})$. The root of P is a child of x. Let q denote the other child of x. And let Q be the subtree of S_{root} rooted at q.
Create three counters g, l_l, and l_r at each node in Q. They are all set to 0 initially.
Calculate $\mathcal{M}_{G,S_{root}}$, and construct the tree Γ
Calculate $\mathcal{M}_{\Gamma,Q}$
for all $\sigma \in \mathsf{Le}(\Gamma)$ **do**
    Let $u = \mathsf{Pa}_\Gamma(\sigma)$
    if u has only one child in Γ **then**
        $g(\mathcal{M}_{\Gamma,Q}(\sigma)) \Leftarrow g(\mathcal{M}_{\Gamma,Q}(\sigma)) + 1$
    else
        Let v denote the other child of u
        if $\mathcal{M}_{\Gamma,Q}(v)$ is in the left subtree of $\mathcal{M}_{\Gamma,Q}(u)$ **and** $\mathcal{M}_{\Gamma,Q}(\sigma)$ is in the right subtree of $\mathcal{M}_{\Gamma,Q}(u)$ in Q **then**
            $l_l(\mathcal{M}_{\Gamma,Q}(u)) \Leftarrow l_l(\mathcal{M}_{\Gamma,Q}(u)) + 1$
        if $\mathcal{M}_{\Gamma,Q}(v)$ is in the right subtree of $\mathcal{M}_{\Gamma,Q}(u)$ **and** $\mathcal{M}_{\Gamma,Q}(\sigma)$ is in the left subtree of $\mathcal{M}_{\Gamma,Q}(u)$ in Q **then**
            $l_r(\mathcal{M}_{\Gamma,Q}(u)) \Leftarrow l_r(\mathcal{M}_{\Gamma,Q}(u)) + 1$
Initialize Ω with the tree Q and weight 0 for each node
$d \Leftarrow \Delta(G, S_{root})$
for each node u in a DFS traversal of Ω **do**
    If $u = \mathsf{Root}(Q)$, then $\Omega(u) \Leftarrow d$
    If $u \neq \mathsf{Root}(Q)$, then let $v = \mathsf{Pa}_Q(u)$
    if u is the left child of v **then**
        $d \Leftarrow d + g(v) - l_l(v)$
    else
        $d \Leftarrow d + g(v) - l_r(v)$
    $\Omega(u) \Leftarrow d$

First we create S_{root} and initialize three counters g, l_l and l_r for each node in Q. We then

compute the mapping $\mathcal{M}_{G,S_{root}}$, the tree Γ and the mapping $\mathcal{M}_{\Gamma,Q}$ as defined in the previous section. Next, we traverse through the tree Γ, and for each leaf node σ that has no siblings we do the following. Let u denote its parent, then if u and σ map to the same node in Q under the mapping $\mathcal{M}_{\Gamma,Q}$, we increment the counter g corresponding to the node $\mathcal{M}_{\Gamma,Q}(\sigma)$ in Q.

Similarly, for each leaf node $\sigma \in \Gamma$ that has a sibling, do the following. Let u denote its parent, v its sibling, and let α denote the node in Q that u maps to under mapping $\mathcal{M}_{\Gamma,Q}$. If v maps into the left subtree of α in and u into the right subtree of α under mapping $\mathcal{M}_{\Gamma,Q}$, then increment the counter l_l associated with node α by 1. And, if v maps into the right subtree of α in and u into the left subtree of α under mapping $\mathcal{M}_{\Gamma,Q}$, then increment the counter l_r associated with node α by 1.

Finally, we compute the node weighted tree Ω. Ω is initialized to Q and all the node weights are 0. We also compute the reconciliation cost of G to the original input tree S and set this value to a variable d. We traverse the tree Ω in depth first search order and for every node encountered, we calculate the weight of that node as shown in the pseudocode in Algorithm 1. Note that the weight at the root node of Q represents the value $\Delta(G, S_{root})$.

Observation 4 *Each tree in* $\mathsf{rSPR}(S, \mathsf{Root}(P))$ *can be obtained by regrafting* P *on the subtree* Q*.*

From Observation 4, it is clear that this is the information we need to solve the RNS problem for one input gene tree. Typically, the RNS problem needs to be solved for multiple input gene trees. In this case we simple follow Algorithm 1 for each gene tree separately. Algorithm 2 shows in detail how this is handled. Note that the tree Ω obtained for each gene tree is identical except for the weights on the nodes. $\Phi(u)$ denotes the weight of node u in tree Φ.

From Observation 2 we know that the NEIGHBORHOOD-SEARCH problem can be solved using the RNS problem (Algorithm 2). Each edge in the given species tree S, defines a subtree that can be pruned. To solve the NS problem we simple keep calling Algorithm 2 for each of these subtrees that can be pruned in S. This produces a node weighted tree Φ for each pruned subtree. It is straightforward to see that these form the required solution for the NS problem.

Algorithm 2 FAST-RNS

1: **Input:** Set of gene trees $\mathcal{G}$, a species tree S, and the pruned subtree P.
2: **Output:** Φ
3: **for** each $G \in \mathcal{G}$ **do**
4: Call RECONCILIATION-COST-TREE(G, S, P) to obtain the tree Ω_G.
5: Initialise Φ to the tree Ω_G for some $G \in \mathcal{G}$.
6: **for** each node u in a DFS traversal of Φ **do**
7: $\Phi(u) \Leftarrow \sum_{G \in \mathcal{G}} \Omega_G(u)$

3.5.1 Proof of correctness

We will show that our algorithm to solve the NS problem is indeed correct. To do this it is sufficient to show that the RNS problem is correctly solved by algorithm FAST-RNS.

Recall that the input for Algorithm 1 is a gene tree G, a species tree S and the pruned subtree P. We have $x = \mathsf{Root}(S_{root})$, the root of P is one child of x and q denotes the other child. Q is the subtree of S_{root} rooted at q. The output Ω is a node weighted version of tree Q.

Based on this we have the following lemma.

Lemma 3.5.1 *After the execution of Algorithm 1 the weight of a node u in tree Ω gives the reconciliation cost for G when the pruned subtree is regrafted onto the edge $(u, \mathsf{Pa}(u))$ in Q.*

Proof: In Algorithm 1, to obtain Ω, we first calculate the three values l_l, l_r and g for each node in Q. Each of these three counters is initially set to 0 at each node. The value of g at a node u in Q, represents the number of additional nodes from G that will become duplications when P is regrafted onto the edge (u, v), $v \in \mathsf{Ch}(u)$, from the edge $(\mathsf{Pa}(u), u)$. The value l_l represents the number of nodes from G that will lose their duplication status when P is regrafted onto the edge (u, v) where v is the left child of u, from the edge $(\mathsf{Pa}(u), u)$. Similarly, the value of l_r represents the number of nodes from G that will lose their duplication status when P is regrafted onto the edge (u, v) where v is the right child of u, from the edge $(\mathsf{Pa}(u), u)$.

Suppose these values are computed correctly at each node in Q. Then, in Algorithm 1 we compute the value $\Delta(G, S_{root})$. This is the weight at the root node of Ω. We calculate the other node weights by performing a DFS traversal of Ω. This ensures that when we reach a

node in Ω, the weight of its parent has already been computed. We then compute the weight of that node based on the values g, l_l and l_r at that node (in Q) and the weight of the parent node.

Consider those internal nodes in tree Γ which do not have any children which are supporting nodes. Under the mapping $\mathcal{M}_{G,S_{root}}$, all such nodes are clearly duplication nodes. From Lemma 3.4.9 it follows that these nodes will continue to be duplication nodes irrespective of the regraft location of P in Ω. From Lemma 3.4.6 it also follows that only the internal nodes of Γ may change their duplication status when subtree P is regrafted to different positions in Ω. Thus, to calculate the values of g, l_l and l_r, we may limit our attention to only those internal nodes of Γ that have a supporting node as a child. Note that, by the definition of Γ, exactly one of the children of each such internal node may be a supporting node. Let us call this set of nodes the *feasible* set. Consider a node a in this feasible set. If it has two children in Γ then one of the children must be an internal node. This means that node a is a duplication node under mapping $\mathcal{M}_{G,S_{root}}$.

The following observation is crucial and follows easily from Lemma 3.4.9.

Observation 5 *If a node u in the feasible set becomes a duplication node when subtree P is regrafted onto an edge, say $(u, \mathsf{Pa}(u))$, in Q, then it remains a duplication node when P is regrafted at any edge in the subtree rooted at u in Q. Similarly, if u loses its duplication status when subtree P is regrafted onto an edge, say $(u, \mathsf{Pa}(u))$, in Q, then it remains a non-duplication node when P is regrafted at any edge in the subtree rooted at u in Q.*

Based on Observation 5, to prove the correctness of Algorithm 1it is enough to show that the values g, l_l and l_r at each node in Q are computed correctly.

Consider a node u from the feasible set that has only one child. In the mapping $\mathcal{M}_{\Gamma,Q}$, u must map to the same node, say a, in Q as its child. It is also clear that u is not a duplication node to begin with i.e. under mapping $\mathcal{M}_{G,S_{root}}$. It is clear from Lemmas 3.4.4 and 3.4.9 that u will be a duplication node if P is regrafted into the subtree rooted at a in Q. Also, note that this is the only scenario in which u may become a duplication node. If a node u from the feasible set has more than one child, then one of these children must be a non-supporting

node, and hence it must be a duplication node to begin with. From Observation 5 it follows that such a node need not be considered while calculating values of g. Consider Lines 7 to 10 in Algorithm 1. They capture exactly the scenario described above and correctly increment the value of g at nodes in Q.

Now consider a node, u, in the rest of the feasible set. u must have two children, one of which must be a non-supporting node. Thus, u is a duplication node. Let α denote the supporting node child and β the other child. Suppose α maps to the left subtree of $\mathcal{M}_{\Gamma,Q}(u)$ in Q, and β to the right subtree. Then, if P is regrafted into the left subtree of $\mathcal{M}_{\Gamma,Q}(u)$, node u loses its duplication status. This follows from Lemmas 3.4.5 and 3.4.9. The case when α maps to the right subtree of $\mathcal{M}_{\Gamma,Q}(u)$ in Q, and β to the left subtree, is symmetric. Observe that it is not possible for both α and β to both map into the left or the right subtree of $\mathcal{M}_{\Gamma,Q}(u)$ in Q. Also, if β maps to the same node as u under mapping $\mathcal{M}_{\Gamma,Q}(u)$, then u can never lose its duplication status. This covers all possible situations for mappings of α and β. Consider Lines 11 to 16 in Algorithm 1. They capture exactly the scenario described above and correctly increment the value of l_l and l_r at node $\mathcal{M}_{\Gamma,Q}(u)$ in Q. ■

Based on the above lemma and definition of reconciliation cost, the following lemma follows easily.

Lemma 3.5.2 *The weight of a node u in tree Φ gives the reconciliation cost for $\mathcal{G}$ when the pruned subtree is regrafted onto the edge $(u, \mathtt{Pa}(u))$ in Q.*

Thus, we have the following theorem.

Theorem 3.5.1 *The* RNS *problem is correctly solved by Algorithm 2*

3.5.2 Complexity analysis

We now analyse the complexity of our algorithm to solve the NS problem. The major component of this algorithm is our algorithm that solves the RNS problem (Algorithm 2). Therefore we first analyse the complexity of algorithm FAST-RNS (i.e. Algorithm 2). Note: for

simplicity of analysis we will assume that all $G \in \mathcal{G}$ have approximately the same size. Even if this does not hold true, our algorithm still improves the current solution by at least $\Omega(n)$.

The input for algorithm FAST-RNS is a set $\mathcal{G}$ of gene trees, a species tree S, and the pruned subtree P of S. Let $n = |\,\mathrm{Le}(S)|$, and $k = |\mathcal{G}|$ i.e. there are k gene trees in the input. Clearly, the size of S and of each $G \in \mathcal{G}$ is $O(n)$.

The input for algorithm RECONCILIATION-COST-TREE is a gene tree G along with S and P. Let $m = |\,\mathrm{Le}(S)| + |\,\mathrm{Le}(G)|$. In this algorithm, we first create the tree S_{root} and the counters g, l_l and l_r for each node in Q. This takes $O(|V(Q)|)$ time. The mapping $\mathcal{M}_{G,S_{root}}$ can be constructed in $O(|V(S_{root})| + |V(G)|)$ time, and the tree Γ can then be easily constructed in $O(|V(\Gamma)|)$ time. Note that $|V(\Gamma)|$ is bounded by $O(|V(G)|)$. $\mathcal{M}_{\Gamma,Q}$ can also be constructed in $O(|V(\Gamma)| + |V(S_{root})|)$ time. All the g , l_l and l_r values can be computed by simply traversing through the tree Γ once. However, while updating the l_l and l_r values we have to check whether a given descendant of some node in Q is in the left subtree or the right subtree of that node. This can be done in constant time as follows: Initially, we perform an inorder traversal of the tree Q and label the nodes with increasing integer values in the order in which they are traversed. This preprocessing step takes $O(|V(Q)|)$ time. Now, given a node a and its descendant b in Q we can tell if b is in the left subtree of a if the label of b is less than the label of a. Otherwise b is in the right subtree of a. This can be checked in constant time.

Once all the g, l_l and l_r values have been set, computing the tree Ω involves first computing the value $\Delta(G, S_{root})$, then traversing the tree Q in depth first search order and spending $O(1)$ time at each node. Thus, the time complexity of this step is $O(|V(G)|) + O(|V(Q)|)$. Hence, the overall complexity of algorithm RECONCILIATION-COST-TREE is bounded by $O(m)$.

Algorithm FAST-RNS calls Algorithm RECONCILIATION-COST-TREE k times. Hence the complexity of this part is $O(km)$. Computing the final tree Φ involves traversing each of the Ω trees produced in a depth first search order. This step takes $O(n)$ time per tree and hence $O(kn)$ time overall. Thus, the time complexity of the FAST-RNS algorithm is bounded by $O(km)$.

The time complexity of our algorithm for the NS problem is thus $O(n) \times O(km) \equiv O(knm)$

(based on Observation 2). The naive brute force algorithm to solve the NS problem requires $O(knm^2)$ time. Our algorithm for the NS problem improves on this by a factor of m i.e. by a factor of at least n . Also observe that this speed up does not come at the expense of higher space complexity.

3.6 Experimental results

In order to study the performance of our algorithm we implemented it as part of a standard local search heuristic for the DUPLICATION problem. We call this program FASTGENEDUP and it implements our FAST-RNS algorithm. In our experiments, we use the local search heuristic and build our starting species tree randomly based on the leaf set in the input gene trees. In particular, we performed two different types of experiments. One analyzed the performance and scalability of FASTGENEDUP using simulated input data and the second focused on an analysis of large empirical data sets.

3.6.1 Performance and scalability

We compared the runtime performance of our program FASTGENEDUP against the program GENETREE (45) that can infer species trees using the same gene duplication heuristic. GENETREE is a well established software and, to the best of our knowledge, the only one that can build species supertrees based on gene duplication heuristics. We measured the time used by each program to compute and output its final species supertree using the same set of input gene trees and the same randomly generated starting species tree. In particular, the input for each run consisted of a set of 20 randomly generated gene trees and a randomly generated species tree, all with the same number of taxa. We conducted six such sets of runs, each with a different number of taxa (50, 100, 200, 400, 1000, and 2000) in the input trees. These experiments were performed on a 3 Ghz Intel Pentium 4 CPU based personal computer with Windows XP operating system. The results of these experiments are shown in Table 3.1. FASTGENEDUP shows a vast improvement in runtime and scalability. Consequently, FASTGENEDUP can compute much larger supertrees within a reasonable time. This also allows our

algorithm to be used with more thorough versions of the heuristic to obtain super trees with lower reconciliation costs. We could not run GeneTree on input trees of size more than 200. The memory consumption of our program also was lower than the memory consumption of GeneTree.

Note that even though both FastGeneDup and GeneTree implement the same local search heuristic, they may produce different supertrees which may even have different reconciliation costs. This happens because during a local search step, more than one neighboring node may have the smallest reconciliation cost. In this case the node to follow is chosen arbitrarily among such nodes, and this may cause the programs to follow different paths in the search space. In practice we noticed little or no difference in the final reconciliation costs. In fact, during the experiments, FastGeneDup inferred supertrees with smaller reconciliation cost more often than GeneTree.

Table 3.1 GeneTree vs. FastGeneDup

Taxa size	GeneTree	FastGeneDup
50	9m:23s	1s
100	3h:25m	6s
200	108h:33m	58s
400	–	9m:19s
1000	–	3h:20m
2000	–	38h:25m

3.6.2 Empirical example

The rapid increase in the amount of available protein sequence data from many taxa makes it possible to perform large-scale analyses of the duplication problem that require fast heuristics. We demonstrated the feasibility of such phylo-genomic analyses using FastGeneDup on plant gene trees. The gene trees were derived from the set of all plant (Viridiplantae) sequences in GenBank (http://www.ncbi.nlm.nih.gov) downloaded on April 13, 2006. In total, this included 390,230 amino acid sequences. The amino acid sequences were clustered into sets of homologs, representing gene families, using the NCBI BLASTCLUST program (16), which performs single linkage clustering of the sequences based on pairwise BLAST scores. We used

a 60% identity cutoff value for the single-linkage clustering and the BLASTCLUST default alignment length. We then pruned the set of all clusters to identify a set of clusters containing at least 4 sequences from at least 3 taxa and containing only sequences from taxa that are found in 10 or more such clusters. We found 3,978 clusters containing sequences from 624 taxa (or technically 624 Genbank taxon ids, most of which represent distinct taxa) that met this criterion. From this set of clusters, we made three data sets that were used to produce the input trees for gene duplication analysis. The first set, the small data set, consisted of the 97 clusters (or gene families) that each included sequences from at least 40 different taxa. This set contained a total of 18,402 protein sequences. The second set, the medium data set, consisted of the 599 clusters that each included sequences from at least 10 different taxa. This data set contained a total of 48,156 sequences, more than 12% of the available plant protein sequences. Finally, the large data set consisted of all 3,978 clusters and contained a total of 100,914 sequences, over 25% of the available plant protein sequences. To our knowledge, the large data set contains by far the most sequences ever incorporated into a single phylogenetic analysis of plants.

The sequences from each of the chosen clusters were aligned using the default options in ClustalW (53). To obtain the gene trees from our data set, we built neighbor-joining trees (48) using PAUP* (52). Since the DUPLICATION problem requires binary, rooted gene trees, zero length branches were randomly resolved, and the trees were rooted with midpoint rooting. We tested the performance of FASTGENEDUP using the local search heuristic starting from a random species tree. The analyses of the small and medium data sets were performed on a Macintosh power PC laptop computer with a 1.5 GHz G4 processor and Mac OS X 10.4 operating system, and the analysis of the large data set was performed on a 3 GHz Intel Pentium 4 based personal computer with Windows XP operating system. The small data set, with 97 input gene trees, took 3 hours 15 minutes and 12 seconds to find an optimal species tree with a score of 13,393 gene duplications. The medium data set, with 599 input gene trees, took 24 hours 55 minutes and 41 seconds to find an optimal species tree with a score of 36,080 gene duplications. The large data set, with 3,978 input gene trees, took 62 hours 35 minutes

and 29 seconds to find an optimal species tree with 75,621 gene duplications.

This purpose of this experiment was to demonstrate how large genomic data sets could be incorporated into phylogenetic analyses using FastGeneDup. Like other attempts to build large plant trees from genome-scale data sets (20), the resulting species trees contain some anomalous relationships as well as some expected relationships. The presence of anomalous relationships are not surprising since the supertree analyses consisted only of a single run of the simple heuristic starting from a random tree. Extensions of our basic approach will undoubtedly improve the resulting species tree, and they further demonstrate the necessity of fast heuristics for the DUPLICATION problem. First, our experiment used mid-point rooting, which assumes that the sequences are evolving according to a molecular clock. However, it appears that the great majority of plant protein families reject the molecular clock assumption (33). Thus, our analyses likely suffer from incorrect rooting in many of the input gene trees. It is extremely difficult to know the true rooting of a gene family tree with a possible history of gene duplications. One solution to this problem would be to adapt the DUPLICATION problem to deal with unrooted gene trees (13). Adapting the DUPLICATION problem to unrooted gene trees would increase its computational complexity. A further problem with the input trees used in this study is that, in order to make them binary, some relationships were resolved arbitrarily, and many of the clades in the gene trees have little or no support. A supertree bootstrapping approach, which incorporates the uncertainty of gene tree clades into support for the species tree, may help address this problem (15; 11). This would require first performing a non-parametric bootstrapping analysis on each of the gene family data sets (24). Then the supertree bootstrapping analysis would consist of replicate analyses of the DUPLICATION problem, randomly sampling a single bootstrap tree from each gene tree family for each replicate (15; 11). Since the non-parametric bootstrapping consists of many replicates of the DUPLICATION problem, in large data sets, this would be impossible to do without a fast heuristic approach for the DUPLICATION problem. A final issue in our analysis is lack of taxonomic overlap among gene trees. In simulation, supertree analyses appear to perform better with more taxon overlap among input trees (6; 22). In some cases, using a constraint tree

species tree may help ameliorate problems from lack of taxonomic overlap among input trees. However, to directly address the problem one must either prune out taxa from the input trees or ideally, increase the taxon sampling in the gene trees. While we extensively sampled from the available plant protein sequences, there are many thousands of available EST sequences from plants that can greatly add to our taxon sampling for many gene families. EST sequences largely come from gene families, and they are ideal for analyses of the DUPLICATION problem. Our fast heuristic will allow us to further increase the size of the gene family data sets to incorporate EST data into large-scale phylogenetic analyses.

3.7 Outlook and conclusion

Despite the inherent complexity of the DUPLICATION problem, it has been an effective approach for incorporating data from gene families into a phylogenetic inference (49; 46; 15; 14). Yet, existing local search heuristics for the problem are slow and thus cannot utilize the vast quantities of newly available genomic sequence data. We introduced an algorithm that speeds up the stepwise search procedure of local search heuristics for the DUPLICATION problem. Our algorithm eliminates redundant calculations in computing the reconciliation cost for all trees resulting from pruning a given subtree and regrafting it to all possible positions. We implemented our algorithm as part of standard local search heuristics, and the resulting program, FASTGENEDUP, greatly improves upon the performance of GENETREE, a previous implementation to solve the DUPLICATION problem. Furthermore, FASTGENEDUP made it possible to compute a supertree with 624 leaves from 3,978 input gene trees, representing over 25% of all available plant protein sequences, in less than three days on a desktop computer.

Our algorithm may be extended to further improve upon its performance in phylogenetic inference. First, the algorithm does not eliminate redundant computations for trees resulting from pruning different subtrees and regrafting them to all possible positions. Eliminating those computations might lead to a further asymptotic improvement in solving the local search problem for the rSPR edit operations. Also, while our current implementation of the algorithm requires a starting species tree, such as a randomly generated tree, a tree-growing heuristic

(38) can be implemented that constructs a good staring tree using the input trees. The starting tree and the tree search also can be constrained to incorporate previous knowledge of the species phylogeny. Finally, it is often difficult to infer the root from a gene family tree. The DUPLICATION problem and the GD model can be modified (12) for unrooted gene trees. Depending on the modifications, our algorithm would still be applicable as a component of a more refined heuristic.

3.8 Acknowledgements

The authors thank Wen-Chieh Chang and Mike Sanderson for many invaluable discussions. This research was supported in part by NSF grant no. 0334832.

BIBLIOGRAPHY

[1] Abrahamson, K., Downey, R. G., and Fellows, M. R. (1993). Fixed-parameter intractability ii. *STACS '93, Lecture Notes in Computer Science*, 665:374–385.

[2] Allen, B. L. and Steel, M. (2001). Subtree transfer operations and their induced metrics on evolutionary trees. *Annals of Combintorics*, 5:1–13.

[3] Anstee, R. P. and Farber, M. (1984). Characterizations of totally balanced matrices. *Journal of Algorithms*, 5:215–230.

[4] Bakonyi, M. and Bono, A. (1997). Several results on chordal bipartite graphs. *Czechoslovak Math. J.*, 47:577–583.

[5] Bender, M. A. and Farach-Colton, M. (2000). The LCA problem revisited. In *Latin American Theoretical INformatics*, pages 88–94.

[6] Bininda-Emonds, O. and Sanderson, M. (2001). Assessment of the accuracy of matrix representation with parsimony analysis supertree construction. *Systematic Biology*, 50:565–579.

[7] Bonizzoni, P., Vedova, G. D., and Dondi, R. (2003). Reconciling gene trees to a species tree. In *CIAC2003 - Italian Conference on Algorithms and Complexity*, Rome, Italy.

[8] Bordewich, M. and Semple, C. (2004). On the computational complexity of the rooted subtree prune and regraft distance. *Annals of Combintorics*, 8:409–423.

[9] Brandstädt, A. (1993). Special graph classes - a survey. Technical report, Schriften reihe des Fachbereichs Mathematikik SM-DU-1993, Universität Duisburg.

[10] Brouwer, A. E., Duchet, P., and Schrijver, A. (1983). Graphs whose neighborhoods have no special cycles. *Discrete Mathematics*, 47:177–182.

[11] Burleigh, J., Driskell, A., and Sanderson, M. (2006). Supertree bootstrapping methods for assessing phylogenetic variation among genes in genome-scale data sets. *Systematic Biology*, 55:426–440.

[12] Chen, K., Durand, D., and Farach-Colton, M. (2000a). Notung: a program for dating gene duplications and optimizing gene family trees. *Journal of Computational Biology*, 7(3/4):429–447.

[13] Chen, K., Durand, D., and Farach-Colton, M. (2000b). Notung: dating gene duplications using gene family trees. In *RECOMB*, pages 96–106.

[14] Cotton, J. and Page, R. (2004). *Tangled tales from multiple markers: reconciling conflict between phylogenies to build molecular supertrees*, pages 107–125. Springer-Verlag.

[15] Cotton, J. and Page, R. D. M. (2002). Vertebrate phylogenomics: reconciled trees and gene duplications. In *Pacific Symposium on Biocomputing*, pages 536–547.

[16] Dondoshansky, I. (2002). Blastclust version 6.1.

[17] Downey, R. G. and Fellows, M. R. (1992). Fixed-parameter intractability. In *Proc. 7th Structure in Complexity Theory Conference*, pages 36–49. IEEE Computer Society Press.

[18] Downey, R. G. and Fellows, M. R. (1993a). Fixed-parameter tractability and completeness iii: Some structural aspects of the w heirarchy. In *Complexity Theory: Current Research*, pages 191–226. Cambridge University Press.

[19] Downey, R. G. and Fellows, M. R. (1993b). Parameterized computational feasibility. In Ambos-Spies, K., Homer, S., and Schoning, U., editors, *Complexity Theory: Current Research*. Cambridge University Press.

[20] Driskell, A., An, C., Burleigh, J., McMahon, M., OMeara, B., and Sanderson, M. (2004). Prospects for building the tree of life from large sequence databases. *Science*, 306:1172–1174.

[21] Eulenstein, O. (1998). *Predictions of gene-duplications and their phylogenetic development.* PhD dissertation, University of Bonn, Germany. GMD Research Series No. 20 / 1998, ISSN: 1435-2699.

[22] Eulenstein, O., Chen, D., Burleigh, J., Fernndez-Baca, and D., Sanderson, M. (2004). Performance of flip supertree construction with a heuristic algorithm. *Systematic Biology*, 53:299–308.

[23] Fellows, M., Hallett, M., Korostensky, C., and Stege., U. (1998). Analogs & duals of the mast problem for sequences & trees. In *European Symposium on Algorithms (ESA), LNCS 1461*, pages 103–114.

[24] Felsenstein, J. (1985). Confidence limits on phylogenies: an approach using the bootstrap. *Evolution*, 39:783–791.

[25] Golumbic, M. C. (1980). *Algorithmic Graph Theory and Perfect Graphs.* Academic Press.

[26] Golumbic, M. C. and Goss, C. F. (1978). Perfect elimination and chordal bipartite graphs. *J. Graph Theory*, 2:155–163.

[27] Goodman, M., Czelusniak, J., Moore, G. W., Romero-Herrera, A. E., and Matsuda, G. (1979). Fitting the gene lineage into its species lineage. a parsimony strategy illustrated by cladograms constructed from globin sequences. *Systematic Zoology*, 28:132–163.

[28] Górecki, P. and Tiuryn, J. (2004). On the structure of reconciliations. In *Recomb Comparative Genomics Workshop 2004*, volume 3388.

[29] Guigó, R., Muchnik, I., and Smith, T. F. (1996). Reconstruction of ancient molecular phylogeny. *Molecular Phylogenetics and Evolution*, 6(2):189–213.

[30] Hallett, M. T. and Lagergren, J. (2000). New algorithms for the duplication-loss model. In *RECOMB*, pages 138–146.

[31] Hammer, P. L., Maffray, F., and Preissmann, M. (1989). A characterization of chordal bipartite graphs. Rutcor research report, Rutgers University.

[32] Harel, D. and Tarjan, R. E. (1984). Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355.

[33] Hartmann, S., Lu, D., Philips, J., and Vision, T. (2006). Phytome: A platform for plant comparative genomics. *Nucleic Acids Research*, 34:D724–D730.

[34] Hoffman, A. J., Kolen, A. A. J., and Sakarovitch, M. (1985). Totally balanced and greedy matrices. *SIAM J. Alg. Discrete Methods*, 6:721–730.

[35] Johnson, C. R. and Miller, J. (1997). Rank decomposition under combinatorial constraints. *Linear Algebra Appl.*, 251:97–104.

[36] Johnson, C. R. and Whitney, G. T. (1991). Minimum rank completions. *Linear and Miltilinear Algebra*, 28:271–273.

[37] Kaplan, H., Shamir, R., and Tarjan, R. E. (1999). Tractability of paramererized completion problems on chordal, strongly chordal, and proper interval graphs. *SIAM Journal of Computation*, 28:1906–1922.

[38] Kitching, I., Williams, D., Forey, P. L., and Humphries, C. J., editors (1998). *Cladistics: The Theory and Practice of Parsimony Analysis*, chapter 3.1 Discovering the most parsimonious cladograms, pages 43–44. Science.

[39] Kloks, T. and Kratsch, D. (1995). Computing a perfect edge without vertex elimination ordering of a chordal bipartite graph. *Information Processing Letters*, 55:11–16.

[40] Lubiw, A. (1987). Doubly lexical orderings of matrices. *SIAM Journal of Computation*, 16:854–879.

[41] Ma, B., Li, M., and Zhang, L. (1998). On reconcstructing species trees from gene trees in term of duplications and losses. In *RECOMB*, pages 182–191.

[42] McKee, T. A. and McMorris, F. R. (1999). *Topics in Intersection Graph Theory*. SIAM monographs on Discrete Mathematics and Appications.

[43] Mirkin, B., Muchnik, I., and Smith, T. F. (1995). A biology consistent model for comparing molecular phylogenies. *Journal of Computational Biology*, 2(4):493–507.

[Page] Page, R. D. M. Genetree. `http://taxonomy.zoology.gla.ac.uk/rod/genetree/genetree.html`.

[44] Page, R. D. M. (1994). Maps between trees and cladistic analysis of historical associations among genes, organisms, and areas. *Systematic Biology*, 43(1):58–77.

[45] Page, R. D. M. (1998). GeneTree: comparing gene and species phylogenies using reconciled trees. *Bioinformatics*, 14(9):819–820.

[46] Page, R. D. M. (2000). Extracting species trees from complex gene trees: reconciled trees and vertebrate phylogeny. *Molecular Phylogenetics and Evolution*, 14:89–106.

[47] Paige, R. and Tarjan, R. E. (1987). Three partition refinement algorithms. *SIAM Journal of Computation*, 16:973–989.

[48] Saitou, N. and Nei, N. (1987). The neighbour-joining method: a new method for reconstructing phylogenetic trees. *Journal of Molecular Biology and Evolution*, 4:406–425.

[49] Slowinski, J. B., Knight, A., and Rooney, A. P. (1997). Inferring species trees from gene trees: A phylogenetic analysis of the elapidae (serpentes) based on the amino acid sequences of venom proteins. *Molecular Phylogenetics and Evolution*, 8(3):349–362.

[50] Solomon, M. and Finkel, R. A. (1980). A note on enumerating binary trees. *J. ACM*, 27:3–5.

[51] Stege, U. (1999). Gene trees and species trees: The gene-duplication problem is fixed-parameter tractable. In *Proceedings of the 6th International Workshop on Algorithms and Data Structures, LNCS 1663*, Vancouver, Canada.

[52] Swofford, D. L. (2002). PAUP*: Phylogenetic analysis using parsimony (*and other methods), version 4.0b10.

[53] Thompson, J., Higgins, D., and Gibson, T. (1994). Clustal w: Improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific penalties and weight matrix choice. *Nucleic Acids Research*, 22:4673–4680.

[54] Yannakakis, M. (1981). Computing the minimum fill-in is np-complete. *SIAM Journal of Alg. Discrete Methods*, 2:77–79.

[55] Zhang, L. (1997). On a Mirkin-Muchnik-Smith conjecture for comparing molecular phylogenies. *Journal of Computational Biology*, 4(2):177–187.