Exact Algorithms for Minimum Routing Cost Trees

Matteo Fischetti* Giuseppe Lancia†‡ Paolo Serafini§

Revised, August 10, 2001

Abstract

Given a set of points and distances between them, a basic problem in network design calls for selecting a graph connecting them at minimum total *routing cost*, i.e., the sum over all pairs of points of the length of their shortest path in the graph. In this paper we describe some branch and bound algorithms for the exact solution of a relevant special case arising when the graph has to be a tree. One of the enhancements to our algorithms is the use of "LP shortcutting", which we introduce as a general purpose technique for speeding up the search. Besides network design, we show how trees of small routing cost find useful application in computational biology, where they can be used to determine good alignments of genomic sequences. This leads to a novel alignment heuristic that we analyze in our computational section.

Keywords: Network Design, Branch and Price, Computational Biology.

1 Introduction

The *minimum communication cost network* is a basic problem in network design, which can be described as follows: n points (e.g. cities) have to be connected by a network (e.g. telephone

*Dipartimento di Elettronica e Informatica, Università di Padova, fisch@dei.unipd.it

†Dipartimento di Elettronica e Informatica, Università di Padova, lancia@dei.unipd.it

‡Part of this work was completed while visiting Sandia Natl Labs, Albuquerque, NM

§Dipartimento di Matematica e Informatica, Università di Udine, serafini@dimi.uniud.it

lines). A given budget is available for the construction of the network, and each possible link has a building cost (to be paid only once) and a length, representing its usage cost. For each pair of points there is a demand, which is proportional to the expected usage of the best path between the points in the final network (e.g., expected number of telephone calls between two cities). The communication cost for a pair will be the length of the shortest path in the network, weighted by the pair's demand. The objective is to design a network whose building cost is within the budget and which minimizes the total communication cost.

The problem has been first studied by Scott [13] and Dionne and Florian [2] among others. Johnson, Lenstra and Rinnooy Kan [7] have shown that the problem is NP-hard, even in case all the demands are equal, all the building costs are 1, and the budget is $n-1$, i.e., the solution is a tree. This result justifies the use of either enumerative approaches like branch and bound ([2]), or heuristics and approximation algorithms ([13, 16]).

Our paper focuses on the relevant special case of finding a spanning tree of minimum communication cost when all the demands are equal, a problem denoted as the *optimum distance spanning tree* in Hu [6] and the *minimum routing cost tree* in Wong [16] and in Wu *et al.* [17]. We study exact algorithms for finding a minimum routing cost tree. The most effective one is a branch and bound procedure based on an integer programming formulation in which the variables are associated with the paths in the solution. This clearly leads to an exponential number of variables; however, we show how these variables can be considered only implicitly and generated quickly when needed, via a well–known method in combinatorial optimization called *column generation*. Besides an exponentially large number of variables, our formulation has also a large (although polynomial) number of constraints. Again, the standard technique to tackle problems of this size is the generation of rows at run time. In this paper we develop some conditions, which we call *LP shortcutting*, that allow us to branch or fathom a search node before completing the solution of the associated LP (needed to obtain the lower bound). This technique is perfectly general and can be applied to any IP formulation with column and row generation. The use of LP shortcutting can provide great savings on the final running time, as reported in our computational results section.

The study of the minimum routing cost tree problem is rather recent, as the original papers of the 70s were mainly focusing on the more general version of the problem. The NP-hardness proof in [7] constructs a graph whose edge lengths do not obey the triangle inequality. In [17], Wu *et al.* show that the non–metric case can be indeed reduced to the metric case, hereby proving NP-hardness for the latter case as well. They also report the first polynomial time approximation scheme, which shows that the optimal solution can be approximated within a factor of $(k+3)/(k+1)$ by the best k–star (a tree with at most k internal vertices). This result builds on previous studies on the approximability of the problem, due to Hu and Wong. Hu [6] derives weak conditions under which the optimum routing cost tree is a star. Wong [16] shows that a 2–approximation is obtained by simply taking the tree of shortest paths, choosing the root vertex producing the minimum routing cost. This idea is exploited by Gusfield [5] to obtain the first approximation algorithm for an alignment problem in computational molecular biology. It is suggested in [17] that the use of trees of small routing cost can in fact lead to better alignment algorithms than Gusfield's. In this paper we have implemented these ideas for some real alignment problems. The results confirm that optimizing the routing cost yields better alignments than [5].

The remainder of the paper is organized as follows. Section 2 introduces some basic notation and three integer programming formulations of the problem. The first two are equivalent (i.e. give the same bound), but one has an exponential number of variables and the other is based on multicommodity flows. The third formulation is much faster to solve, but produces weaker bounds. In section 3 we investigate some combinatorial lower bounds. In section 4 we describe the strategy of LP shortcutting. Section 5 discusses the main ingredients of our enumerative procedure, namely a local search algorithm for finding good feasible solutions, various branching rules, and some valid inequalities. In section 6 we outline an application of minimum routing cost trees to the problem of aligning a set of genomic sequences. Computational results are reported in section 7. Some conclusions are drawn in section 8.

2 Notation and Integer Programming Models

An instance of the Minimum Routing Cost Tree (MRCT) problem is specified by an undirected graph $G = (V, E)$ with nonnegative lengths on the edges. We let $|V| = n$, $|E| = m$, and $V = \{1, \ldots, n\}$. The *length* of an edge $e = \{i, j\}$ will be denoted as d_e or $d(i, j)$. We will assume $d_e \geq 0$ for all $e \in E$. A *pair* of vertices is an edge of the complete graph $K_n = (V, \Pi)$. For a subgraph G' of G and a pair $\{i, j\} \in \Pi$ of vertices, $d(i, j, G')$ is the distance (i.e. shortest path length) between i and j in G'. If G' is a tree then $d(i, j, G')$ is the length of the unique path connecting i and j. The routing cost of a spanning tree T is defined as $rc(T) := \sum_{\{i,j\} \in \Pi} d(i, j, T)$. For a tree T and an edge $e \in T$, the *load* of e in T, denoted $\Lambda(e, T)$, is the number of paths using e, i.e., $|S| \cdot |V - S|$, where S is one shore of the cut identified by the removal of e from T. We can rewrite the routing cost of a tree as $rc(T) = \sum_{e \in T} \Lambda(e, T) d_e$.

For each pair $h = \{i, j\} \in \Pi$, we denote by $\mathcal{P}^h$ the set of (simple) paths in G between i and j. The set of all paths in G will be denoted by $\mathcal{P}$. Note that a path of cardinality 1 is (isomorphic to) an edge, and hence $E \subseteq \mathcal{P}$. For each path $P \in \mathcal{P}$, we let $d_P := \sum_{e \in P} d_e$.

The basic formulation

We can formulate the MRCT problem as a mixed–integer program with decision variables x_P, for $P \in \mathcal{P}$, used to select a path between each pair of vertices. The constraints are such that, in a feasible solution, the set $\{e \in E \mid x_e = 1\}$ defines a tree. The following is an integer programming formulation of the minimum routing cost tree problem:

(IP1)
$$\min \sum_{P \in \mathcal{P}} d_P x_P \tag{1}$$

subject to

$$\sum_{P \in \mathcal{P}^h} x_P = 1 \qquad h \in \Pi \tag{2}$$

$$\sum_{P \in \mathcal{P}^h : P \ni e} x_P \leq x_e \qquad e \in E, h \in \Pi - \{e\} \tag{3}$$

$$\sum_{e \in E} x_e = n - 1 \tag{4}$$

$$x_e \geq 0, \text{ integer}, e \in E; \quad x_P \geq 0, \ P \in \mathcal{P} - E. \tag{5}$$

In this model there are exponentially many variables and $(m+1)\,(n-1)\,n/2-m+1 = O(m\,n^2)$ constraints. Constraints (2) force each pair to be connected. Constraints (3) say that only edges in the tree can be used by a path. Because connection is given by (2), for the x_e to induce a tree, it is enough to have $n-1$ edges, as required by (4). Note that we need not to impose integrality on all the variables, but only on the ones associated with edges. Indeed, since in a tree there is only one path for each pair of vertices, it can be easily seen that if the x_e are binary for all $e \in E$, then the x_P are binary for all $P \in \mathcal{P}$. One of the main difficulties in the design of branch and price algorithms is devising effective branching rules when the binary variables are priced out at run-time. A typical problem is that there may be no easy way to forbid that a variable that was fixed at 0 by branching, could still be a feasible solution for the pricing problem. This issue can be avoided here, by pricing out only the path–variables associated with paths in $\mathcal{P} - E$, and keeping the variables $x_e, e \in E$, that are candidate for branching, always present in each LP.

The LP relaxation of (IP1), called (LP1) in the sequel, can be solved exactly in polynomial time by column–generation techniques. Therefore, we can use (LP1) to compute lower bounds. To this end, we define the following dual variables for (LP1), associated with the constraints (2), (3), and (4) respectively: u_h, for $h \in \Pi$; $v_{eh} \geq 0$, for $e \in E, h \in \Pi - \{e\}$; and w. The dual of (LP1) has the following constraints:

$$u_e + \sum_{h \in \Pi - \{e\}} v_{eh} + w \leq d_e \qquad e \in E \tag{6}$$

$$u_h - \sum_{e \in P} v_{eh} \leq d_P \qquad h \in \Pi, \ P \in \mathcal{P}^h, \ |P| \geq 2. \tag{7}$$

A variable with negative reduced cost in (LP1) corresponds to a dual constraint violated by the current dual solution. Assuming that the m columns for the edges are always present in the LP formulation, the constraints (6) will never be violated. To check whether there are violated

constraints (7) we proceed as follows. We rewrite constraints (7) as

$$\sum_{e \in P} (v_{eh} + d_e) \geq u_h \qquad h \in \Pi,\ P \in \mathcal{P}^h,\ |P| \geq 2. \tag{8}$$

For a fixed $h = \{i, j\}$, define $d'_e = v_{eh} + d_e$ for all $e \in E$. Then the constraint is violated for h if there exists an i–j path, other than h, whose length with respect to the costs d' is shorter than u_h. This can be checked by finding the *shortest* i–j path in $E - \{h\}$, with respect to the costs d'. Since all costs are nonnegative, the shortest path can be found in polynomial time, e.g., by Dijkstra's algorithm. If such a path P has $d'_P < u_h$, then x_P can be added to the set of primal variables for a new simplex iteration. On the other hand, if for all pairs h condition (8) is satisfied, the current LP solution is optimal. Note that, by a trivial modification of Dijkstra's labeling algorithm for the shortest path problem, we can retrieve not just the shortest path, but possibly many paths of length $d'_P < u_h$. This is advisable, as we noted that adding several negative reduced cost variables at each iteration of the column generation results in a smaller overall running time (see section 7).

An equivalent multicommodity flow formulation

The MRCT problem can alternatively be modeled as a multicommodity flow, in which each vertex sends $n - 1$ units of flow, one to each remaining vertex. Let A be the set of oriented arcs obtained by directing each edge of E in both possible ways. We define real variables $x^h_{(i,j)}$ for each $h = \{u, v\} \in \Pi$ and $(i, j) \in A$, representing the flow from u to v along the arc (i, j). Further, we have binary variables y_e for $e \in E$, which represent the edges of the tree. We obtain the following formulation:

(MCF)
$$\min \sum_{h \in \Pi} \sum_{(i,j) \in A} d(i, j)\, x^h_{(i,j)} \tag{9}$$

subject to

$$\sum_{(u,j) \in A} x^h_{(u,j)} - \sum_{(j,u) \in A} x^h_{(j,u)} = 1 \qquad h = \{u, v\} \in \Pi, u < v \tag{10}$$

$$\sum_{(i,j)\in A} x^h_{(i,j)} - \sum_{(j,i)\in A} x^h_{(j,i)} = 0 \qquad h = \{u,v\} \in \Pi, i \in V - \{u,v\} \tag{11}$$

$$x^h_{(i,j)} + x^h_{(j,i)} \leq y_e \qquad e = \{i,j\} \in E, h \in \Pi \tag{12}$$

$$\sum_{e\in E} y_e = n - 1. \tag{13}$$

There are $m(n^2 - n + 1) = O(mn^2)$ variables and $(m + n - 1)(n - 1)n/2 + 1 = O(m\,n^2)$ constraints. The constraints (10) and (11) (flow conservation) guarantee that all pairs are connected; constraints (12) control edge activation, while (13) force the support graph of y to be a tree. By the Flow Decomposition Theorem ([1],p. 80), any u–v flow can be decomposed into a set of u–v paths, and conversely. As a consequence, it is easy to see that the formulations (IP1) and (MCF) are equivalent, i.e. given a feasible solution to one of them, we can compute a feasible solution to the other of the same value. In particular, the two relaxations give the same lower bound, and hence the choice between them depends on how fast they can be solved and the amount of memory required. In both respects, we have found that (IP1) is to be preferred over (MCF).

Aggregate formulation

Here the idea is to bound the maximum number of paths that use an edge chosen in the tree. Let L^+_e be an upper bound of the load of an edge e if included in an optimal solution. Clearly, we can always set $L^+_e = \lfloor \frac{n}{2} \rfloor \cdot \lceil \frac{n}{2} \rceil$, but we can possibly use better bounds, depending on the input graph and the incumbent solution (see section 5.2). The new model, called (IP2), is the same as (1)–(5), with constraints (3) replaced by

$$\sum_{P\in\mathcal{P}:P\ni e} x_P \leq L^+_e x_e \qquad e \in E. \tag{14}$$

This formulation has only $O(n^2)$ constraints. Let u_h, $h \in \Pi$, be the dual variables associated to (2), and let $v_e \geq 0$, $e \in E$, be the dual variables for (14). The dual constraints corresponding to the variables x_P, $|P| \geq 2$, are

$$u_h - \sum_{e\in P} v_e \leq d_P \qquad h \in \Pi, P \in \mathcal{P}^h - \{h\}. \tag{15}$$

As to variable pricing, for each $e \in E$ we define $d'_e = d_e + v_e$ and rewrite constraints (15) as $\sum_{e \in P} d'_e \geq u_h$ for all $h \in \Pi, P \in \mathcal{P}^h$. We can then use a shortest path algorithm to find violated constraints (15).

The model (IP1) defined by (1)–(5) is highly degenerate and hence the solution of its LP relaxation is often unacceptably slow. For this reason, we have investigated the alternative formulations (MCF) and (IP2), and also the generation of constraints at run time for (IP1). Constraint generation requires the solution of several LPs, each of which can be solved faster since it is smaller and less degenerate. We observed that, by generating both columns and rows, the solution of (LP1) can be speeded up by a factor of 10.

According to our computational experiments, detailed in section 7, the best formulation is (IP1) with generation of both columns and rows, enhanced by LP shortcutting and additional cuts, as described in the following sections.

3 Lower bounds

In the branch and bound procedures studied in this paper, the integer decision variables correspond to the edges in the solution. Accordingly, at each node of the search tree we have three subsets of the edges, forming a partition of E. We denote by E_0 the set of all *forbidden* edges that are excluded from each solution at the node. E_1 is the set of all *included* edges, i.e., edges that must be in all solutions at the node. Finally, the set of *free* edges is $E_x = E - (E_0 \cup E_1)$. It is always required that E_1 is acyclic and $E - E_0$ is connected, or otherwise the node cannot contain any feasible solution. This partitioning scheme is independent of the approach used to obtain the lower bounds.

In this section we describe valid lower bounds for the problem. These bounds can be computed at each node of the search tree, and hence depend on G, E_1 and E_0. Let $l_{\mathrm{IP1}} = l_{\mathrm{MCF}}$ and l_{IP2} denote the lower bounds given by the optimum value of the linear programming relaxations of (IP1), (MCF) and (IP2), with respect to the given E_0 and E_1.

For all i and j in V, let $d(i, j, E_1)$ be the shortest path lengths in (V, E_1). We distinguish two

sets of vertex pairs: $\Pi_C = \{\{i,j\} \in \Pi \mid d(i,j,E_1) < \infty\}$, contains the pairs that are *connected* in (V, E_1), and $\Pi_D = \Pi - \Pi_C$ those that are *disconnected* in (V, E_1).

For $\{i,j\}$ in Π_C, the length of the path in the final tree is already known. So, we only need to bound the contribution for the pairs $\{i,j\}$ in Π_D. For $\{i,j\}$ in Π_D we denote by $c_1(i,j)$ (respectively, $c_2(i,j)$) a lower bound on the length of the path between i and j in a tree, in the case where the path consists of 1 edge (respectively, 2 or more edges). Then $c_1(i,j) = d(i,j)$ (possibly ∞ if $\{i,j\} \notin E_x$) and $c_2(i,j)$ is the length of a shortest i–j path in $E - E_0 - \{\{i,j\}\}$.

A trivial lower bound is given by the sum over all pairs of their shortest path distance:

$$l_{\mathrm{SUM}} = \sum_{\{i,j\} \in \Pi_C} d(i,j,E_1) + \sum_{\{i,j\} \in \Pi_D} \min\{c_1(i,j), c_2(i,j)\}. \tag{16}$$

The trivial lower bound can be improved as follows. For all $\{i,j\} \in \Pi_D$, define $\delta(i,j) := c_2(i,j) - c_1(i,j)$. Given any tree $T = (V, E(T))$ with $E_1 \subseteq E(T)$ and $E_0 \cap E(T) = \emptyset$, we denote by $X = E(T) \cap \Pi_D$ the set of pairs in Π_D that are connected by an edge of T, while all the other pairs in Π_D are connected by a path of at least two edges. Then the routing cost of a tree is at least $\sum_{\{i,j\} \in \Pi_C} d(i,j,E_1) + \min_{X \subseteq \Pi_D} \left(\sum_{\{i,j\} \in X} c_1(i,j) + \sum_{\{i,j\} \in \Pi_D \setminus X} c_2(i,j) \right)$, which can be rewritten as $\sum_{\{i,j\} \in \Pi_C} d(i,j,E_1) + \sum_{\{i,j\} \in \Pi_D} c_2(i,j) - \max_{X \subseteq \Pi_D} \sum_{\{i,j\} \in X} \delta(i,j)$.

Dionne and Florian [2] strengthened this bound by constraining the size of X to be exactly $n - 1 - |E_1|$, hence taking the $n - 1 - |E_1|$ largest $\delta(i,j)$'s. We can do better by imposing that $X \cup E_1$ must be a tree, hence finding the maximum (with respect to costs $\delta(i,j)$) spanning tree $T = (V, E(T))$ with $E_1 \subseteq E(T)$ and $E_0 \cap E(T) = \emptyset$, and setting $X = T - E_1$. This leads to the combinatorial lower bound

$$l_{\mathrm{COMB}} = \sum_{\{i,j\} \in \Pi_C} d(i,j,E_1) + \sum_{\{i,j\} \in \Pi_D} c_2(i,j) - \sum_{\{i,j\} \in X} \delta(i,j). \tag{17}$$

Quite surprisingly, this bound turns out to be very close to the bound given by the LP relaxation of formulation (IP2) (see Table 2), although it is much faster to compute. However, it is considerably weaker than l_{IP1} and it is not enough powerful to solve instances on complete metric graphs with more than 15 vertices.

A different lower bound is based on dynamic programming, but has space and time complexity which make it of little use for instances involving more than very few vertices. For a subset $S \subseteq V$ of vertices, let $l_{\mathrm{DP}}(S)$ be the lower bound on the routing cost of a tree spanning the vertices in S. Also, for $i \in S$, let $w(i,S) := \sum_{j \in S:\{i,j\} \in \Pi_C} d(i,j,E_1) + \sum_{j \in S:\{i,j\} \in \Pi_D} d(i,j,E \setminus E_0)$ be the minimum total length for connecting i to all other vertices in S. Given a tree T over S, any edge $\{i,j\} \in T$ splits T into two subtrees, one covering i over some $S' \subseteq S$, and one covering j over $S'' = S \setminus S'$. For each $u \in S'$ we must pay at least $d(i,j) + w(j,S'')$ to connect u to all the vertices in S'', and similarly for each $v \in S''$ we have to pay $d(i,j) + w(i,S')$ to connect v to all vertices in S'. Pairs that have both endpoints in S' or S'' are accounted for in $l_{\mathrm{DP}}(S')$ and $l_{\mathrm{DP}}(S'')$. We get the following recurrence:

$$l_{\mathrm{DP}}(S) = \min \{ l_{\mathrm{DP}}(S') + l_{\mathrm{DP}}(S'') + |S'| w(j,S'') + |S''| w(i,S') + |S'||S''| d(i,j) \} \qquad (18)$$

where the minimum is taken over all $\emptyset \subset S' \subset S$, $i \in S'$, $j \in S'' := S \setminus S'$ and $\{i,j\} \in \Pi_D$. Computing the bound $l_{\mathrm{DP}}(V)$ has space complexity $O(n\,2^n)$ and time complexity $O(n^2\,2^n)$ which is due to the dominating cost of computing all $w(i,S)$. Note that solving the minimum routing cost tree problem by complete enumeration has cost $O(n^n)$ since there are n^{n-2} trees and each routing cost can be computed in time $O(n^2)$.

4 LP shortcut

For an LP-based branch and bound where both rows and columns are generated, the following *LP–shortcut* strategy can be used to speed up the computation. Given an incumbent solution of value u, the main use of an LP solution is to provide a lower bound l which is used to decide whether to branch ($l < u$) or to fathom the current branching node ($l \geq u$). Now, if we know lower and upper limits on l, i.e. $l' \leq l \leq l''$, we can branch, in a "preemptive" way, as soon as $l'' < u$, or fathom the node, in a "preemptive" way, as soon as $l' \geq u$. We can obtain these limits at the end of each phase of row and column generation. Indeed, whenever there are no violated constraints (i.e. at the end of a phase of row generation), the optimal solution to the current

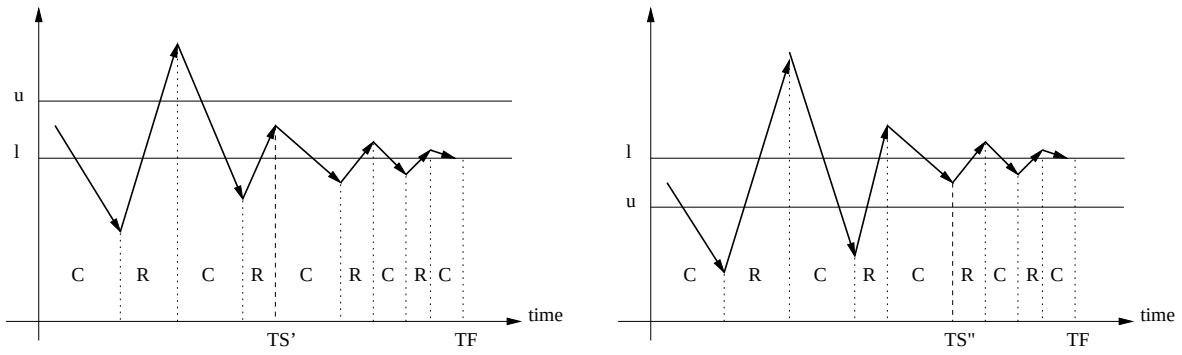

Figure 1: LP–shortcut. Preemptive branching (left) and preemptive fathoming (right). Labels C and R refer to column and row generation phases, respectively.

LP gives an upper bound on l. Analogously, after column generation we have an LP solution which includes (implicitly) all variables, but possibly does not satisfy all constraints, hence its value is a lower bound on l. Note that the values l' and l'' computed this way are monotonically approaching l from below and from above, respectively (assuming rows and columns are always added and never removed). The situation is illustrated graphically in Figure 1. In the left drawing, shortcutting implies that branching can be performed at time TS$'$, while the final bound value l is obtained only at time TF. In the right drawing, the node can be fathomed at time TS$''$. Although the idea of using intermediate bounds for fathoming the node is known, the use of preemptive branching is (to the best of our knowledge) new. Note that, by adding novel cuts to the LP, it is possible to improve the lower bound, hence preemptive branching could be disadvantageous in the cases where additional cuts are considered. In practice, however, this can be taken into account heuristically as follows: If at time TS$'$ the relative gap between the current LP value and u is smaller than a given threshold α, then we do not apply shortcut and keep optimizing the node, in the hope that the addition of cuts will be sufficient to fathom the problem. In our runs, we have used $\alpha = 0.5\%$.

We have implemented this idea in a general–purpose IP solver. For the solution of our problem, LP–shortcut gives a global (over all nodes of the search tree) speed up of about 45% with respect to the full LP solution at each node. It is worth noting that at a single node the

speed up can be larger than 90%; however, the work saved at a node (e.g. generation of useful columns) may be needed at a later time, so the overall saving is less impressive than the one obtained, e.g., at the root node.

These observations can be developed further as follows. Denote by $l'(i)$ the value of the LP at the end of the i–th phase of column generation, where $l'(i)$ is a lower bound for each i, and $l'(i) \leq l'(i+1)$. We have observed that the convergence of $l'(i)$ to the actual best lower bound l is quite slow, that is $l'(i+1) - l'(i)$ decreases very rapidly, while the time needed to solve the LPs increases due to the increased size. The key observation is that after few iterations (e.g., less than 10) we have already a bound that is 99% of the best bound, while we have spent almost as little as 20% of the total time. In other words, 80% of the time is actually spent in improving the bound by at most 1%. On the instances that we used for our computational experiments, we observed that six iterations gave a good trade off between the quality of the bound and the time needed to compute it. Hence, in our branch and bound we used the lower bound $l'(6)$. This way we have been able to solve problems otherwise unsolvable in reasonable time with the use of the original lower bound. As observed before, a preemptive branching can be a disadvantage in those cases where the actual best bound would have been sufficient to fathom the node. To contrast this drawback we keep optimizing the LP if the ratio between $l'(6)$ and u is smaller than a given threshold α (heuristically set to 0.5%).

5 The branch–and–bound ingredients

5.1 A local–search heuristic

The procedure we use to obtain a good starting feasible solution is based on the standard local search neighborhood for problems whose solution is a tree. We start with any tree T (e.g., the best 2–star, which gives a $\frac{5}{3}$–approximation [17]). For each edge $e \notin T$, we try adding e to T, thus creating a cycle C_e that we break by removing the edge $f \in C_e$ for which $rc(T - \{f\} \cup \{e\})$ is minimum. Among all possible $e \notin T$, we choose to add to T that edge which gives the largest decrement in the routing cost of the tree. If no $e \notin T$ gives a decreased routing cost, we stop

and output T; otherwise, we update the tree and iterate the same procedure. This search is very fast and is repeated from different starting trees T. Notice that, for each attempted move, the computation of the new routing cost can be done by only considering the edges of the cycle, the load of all the other edges remaining unaffected by the move. Further, to update the loads, it is enough to store for each edge the sizes of the two shores of the cut identified by it. When implemented this way, with a suitable data structure, the search takes only linear time per move. Hence the convergence to a local optimum is usually very fast (e.g., less than 1 second for $n = 30$).

From our experiments, it appears that this heuristic is extremely effective on graphs of the size tackled by our program (up to 30 vertices), and, iterated for 100 starting trees, yielded the optimal solution on most instances in our test–bed.

5.2 Node tightening

At any node of the branch and bound tree we have a set $E_1 \subseteq E$ of edges that have been included in the solution and a set $E_0 \subseteq E$ of edges that are forbidden from being in the solution. We enlarge these sets by adding to E_0 any edge $e \in E_x$ which creates a cycle in E_1. As a result, every $e \in E_x$ has its endpoints in different connected components of (V, E_1). Furthermore, we add to E_1 any edge e that is the unique edge of E_x in the cutset defined by one connected component of E_1. We then try to fix some more edges by using the value u of the current incumbent solution. In particular, we compute for each $e \in E - E_0$ an upper bound L_e on the load that e can have in an optimal solution corresponding to the current node and, if $L_e = 0$, we add e to E_0 (in case $E - E_0$ becomes disconnected, we fathom the node).

To be more specific, for each $a = \{i, j\} \in E - E_0$ let n_i and n_j be the sizes of the connected components containing i and j in $(V, E_1 - \{a\})$; without loss of generality, assume $n_i \leq n_j$. The *forest load* F_a of a, defined as $n_i \times (n - n_i)$, is the smallest possible load that a can have in the final tree. For any $e \in E - E_0$ and an integer "tentative" load L_e, consider the minimum spanning tree T^0 of $E - E_0$, with $E_1 \cup \{e\} \subseteq T^0$, computed with respect to costs $c_a = F_a d_a$ for each $a \in E - (E_0 \cup \{e\})$ and cost $c_e = L_e d_e$ for edge e. Let $c(T^0) = \sum_{a \in T^0} c_a$ be its

overall cost. Then, e can have load L_e or more only if $c(T^0) < u$. In fact, the routing cost of a tree T containing $E_1 \cup \{e\}$ and with load at least L_e on e is $\sum_{a \in T-\{e\}} \Lambda(a,T)d_a + \Lambda(e,T)d_e \geq \sum_{a \in T-\{e\}} F_a d_a + L_e d_e = c(T) \geq c(T^0)$. Accordingly, we find a value $k \in 1, \ldots, \lfloor n/2 \rfloor$ such that for $L_e = k\,(n-k)$ we have $c(T^0) \geq u$ while for $L_e = (k-1)(n-k+1)$ we have $c(T^0) < u$. If $k = 1$, then 0 is the maximum load for e, hence this edge can be added to E_0. This in turn may imply further tightening of loads for other edges, so we cycle through this procedure as long as some edges get deleted. The above reduction criterion is often effective and can be used in all our formulations. In addition, values $L_e > 0$ can be used in the formulation (IP2) to strengthen constraints (14).

5.3 Branching

At each node we can branch on a fractional "edge"–variable x^*_e, $e \in E$, by fixing it to 0 or 1. There are many general criteria to choose the branching variable, as described, e.g., in Savelsbergh [**?**]. As to problem–specific rules, one can choose the branching variable so as to let E_1 grow as a forest (similarly to Kruskal's algorithm for the MST). Alternatively, one can always choose it in the cut defined by the component of $\{a \in E : x^*_a = 1\}$ containing vertex 1, this way trying to grow a single tree like in Prim's algorithm for the MST. Our computational tests have shown that the greater freedom allowed by the first strategy gives a better performance. When choosing the branching edge we use one of the following criteria.

Fixed order criteria: The edges are sorted according to some rule at the end of the root node. Then they are always scanned in this order, and we branch on the first fractional variable encountered. Ordering rules are explained below.

F1: *Smaller length.* Edges are sorted by nondecreasing length.

F2: *Larger length.* Edges are sorted by nonincreasing length.

F3: *Based on heuristic solution.* Edges are sorted according to how critical they are in the heuristic solution T_H at the root node (which is usually optimal). An edge e is considered to be more critical than a if $\Lambda(e,T_H) > \Lambda(a,T_H)$ or $\Lambda(e,T_H) = \Lambda(a,T_H)$ and $d(e) \leq d(a)$.

Variable order criteria: We look at the current subproblem and its solution, and choose the branching edge according to one of the following criteria.

V1: *Most fractional.* The standard choice for 0–1 variables.

V2: *Smaller load* $\times$ *length.* We compute the forest load F_e of all edges (defined in 5.2) and branch on the fractional edge with minimum $F_e\, d_e$.

V3, V4, V5: *Best expected lower bounds (max min, max max, max diff).* These rules are based on the following idea. Assume we knew the lower bounds $LB(e,0)$ and $LB(e,1)$ that we would get by fixing x_e to 0 and 1 respectively, for all $e \in E_x$. Then, we may want to branch on an edge for which $\min\{LB(e,0), LB(e,1)\}$ is maximum, trying to get as high as possible lower bounds for both subproblems. Alternatively, we may want to maximize $\max\{LB(e,0), LB(e,1)\}$ so that at least one of the two subproblems is likely to be fathomed. A final idea is that of maximizing $|LB(e,0) - LB(e,1)|$, i.e. trying to make one of them promising and the other unappealing. Computing l_{IP1} for all e is too expensive, hence we use the very fast combinatorial lower bound l_{COMB} (roughly equivalent to l_{IP2}).

We found branching to be of crucial importance. Indeed, some rules are consistently quite faster than some others. In particular, F2 is much worse than F1, and V3 much worse of V4, which is similar to V5. According to our computational results, reported in [4], the rules F3 and V2 outperform all other rules.

5.4 Valid inequalities

In this section we discuss some valid inequalities for our model. It should be remarked that the solution of our LP's is itself quite time–demanding, so our strategy of shortcutting implies that most of the times we do not solve optimally the LP nor add valid inequalities. However, on small instances (or when the lower bound is very close to the value of the incumbent solution) we do solve the LP optimally and try to improve the bound by using valid cuts.

$\tilde{L}_{ij}$	$\tilde{D}_i$	$\tilde{D}_j$	$\tilde{x}_{ij}$
0	1	1	0
	1	$n-2$	
	$n-2$	1	
$n-1$	0	1	1
	1	0	
	0	$n-2$	
	$n-2$	0	
$2(n-2)$	1	1	1
	1	$n-3$	
	$n-3$	1	
$k(n-k)$	1	1	1
	1	$n-k-1$	
	$n-k-1$	1	
	$k-1$	$n-k-1$	
	$n-k-1$	$k-1$	
	1	$k-1$	
	$k-1$	1	

Table 1: All cases for extreme feasible vectors $(\tilde{L}_{ij}, \tilde{D}_i, \tilde{D}_j, \tilde{x}_{ij})$.

All the standard tree-inequalities on the x_e variables, $e \in E$, are valid for our model, but some of them are already implied as, e.g., the valid inequality $\sum_{e \in \delta(S)} x_e \geq 1$ for $S \subseteq V$. On the other hand, the inequalities

$$\sum_{e \in E(S)} x_e \leq |S| - 1 \qquad S \subseteq V \tag{19}$$

are non–redundant, and can be separated by using a standard technique based on maximum flow computations [8].

We also studied a family of inequalities derived by considering a small portion of a tree and the relationship between the variables involved. In particular, consider an edge $\{i, j\}$, and denote by L_{ij} its variable load, i.e., $L_{ij} := \sum_{P \ni \{i,j\}} x_P$. The basic inequality we start with is

$$(n-1)x_{ij} \leq L_{ij} \leq \left\lfloor \frac{n}{2} \right\rfloor \left\lceil \frac{n}{2} \right\rceil x_{ij}. \tag{20}$$

To strengthen this inequality we use information on the degree of i and j (e.g., if either i or j is a leaf, then L_{ij} must be equal to $n-1$). Define $D_i := \sum_{e \in \delta(i)} x_e - x_{ij}$ as the number of neighbors of i in a solution (excluding j), and let D_j be defined similarly. We want to derive a

general valid inequality that links the variables L_{ij}, D_i, D_j, and x_{ij}, the most general form of which will be $\alpha L_{ij} + \beta D_i + \gamma D_j + \rho x_{ij} \leq \epsilon$ for some unknown real coefficients α, β, γ, ρ and ϵ. Given a fractional solution $(L^*_{ij}, D^*_i, D^*_j, x^*_{ij})$ the most violated valid inequality of the above type can be found by solving the linear program:

$$\max_{-1 \leq \alpha,\beta,\gamma,\rho,\epsilon \leq 1} \quad \alpha L^*_{ij} + \beta D^*_i + \gamma D^*_j + \rho x^*_{ij} - \epsilon \tag{21}$$

subject to

$$\alpha \tilde{L}_{ij} + \beta \tilde{D}_i + \gamma \tilde{D}_j + \rho \tilde{x}_{ij} \leq \epsilon \qquad \text{for all extreme feasible vectors } (\tilde{L}_{ij}, \tilde{D}_i, \tilde{D}_j, \tilde{x}_{ij}), \tag{22}$$

where the finitely–many extreme feasible vectors $(\tilde{L}_{ij}, \tilde{D}_i, \tilde{D}_j, \tilde{x}_{ij})$ can easily be obtained by enumeration, as illustrated in Table 1. An alternative way for finding these inequalities, consists of analyzing off–line the facets of the polyhedron whose extreme points are listed in Table 1. In order to do this, we used the package `porta` [9] to compute the facets of the convex hull of all extreme feasible vectors $(\tilde{L}_{ij}, \tilde{D}_i, \tilde{D}_j, \tilde{x}_{ij})$ for various values of n and then we derived the form of general valid inequalities which define some of these facets. An example of such inequality is:

$$L_{ij} \leq \left\lfloor \frac{n}{2} \right\rfloor \left\lceil \frac{n}{2} \right\rceil x_{ij} - \left\lfloor \frac{n-2}{2} \right\rfloor \left\lceil \frac{n-2}{2} \right\rceil (2 - D_i - D_j). \tag{23}$$

To show this inequality is valid, first we observe that it can be tight only if $x_{ij} = 1$ and $D_i + D_j \leq 1$, i.e., i or j is a leaf. In such case, the inequality simply states that $L_{ij} \leq n - 1$.

6 An application to computational biology

Comparing genomic sequences drawn from individuals of the same or different species is one of the fundamental problems in molecular biology. These comparisons can suggest evolutionary relationships, identify highly conserved DNA regions, spot fatal mutations, etc. Therefore, the mathematical formulation and solution of the so–called *Multiple Sequence Alignment* problem constitutes a fundamental challenge for computational molecular biologists.

A genomic sequence is a string over the 4–symbols alphabet of nucleotides or the 20–symbols alphabet of amino acids. Aligning a set of sequences consists in arranging them in a matrix

having each sequence in a row. This is obtained by possibly inserting gaps (represented by the '-' character) in each sequence so that they all result of the same length. The goal of identifying common patterns is pursued by attempting as much as possible to place the same character in every column. The following is a simple example of an alignment of the sequences `ATTCGAC`, `TTCCGTG` and `ATCGTC`:

```
A  T  T  -  C  G  A  -  C
-  T  T  C  C  G  -  T  G
A  -  T  -  C  G  -  T  C
```

The multiple sequence alignment problem has been formalized as an optimization problem. The most popular objective function for multiple alignment generalizes ideas from optimally aligning two sequences. This problem, called pairwise alignment, is formulated as follows: Given symmetric costs $c(a, b)$ for replacing a symbol a with a symbol b and costs $c(a, -)$ for deleting/inserting symbol a, find a minimum–cost set of symbol operations that turn a sequence S' into a sequence S''. It is well known that this problem can be solved by dynamic programming in time and space $O(l^2)$, where l is the maximum length of the two sequences. The value of an optimal solution is called the *edit distance* of S' and S'' and is denoted by $d(S', S'')$.

An *alignment* $\mathcal{A}$ of two or more sequences is an array having the (gapped) sequences as rows. The value $d_{\mathcal{A}}(S', S'')$ of an alignment of two sequences S and S' is obtained by adding up the costs for the pairs of characters in corresponding positions, and $d(S', S'') = \min_{\mathcal{A}} d_{\mathcal{A}}(S', S'')$. In the *Sum–of–Pairs* (SP) score, the cost of an alignment of many sequences is obtained by adding the costs of the symbols matched up at the same positions, over all the pairs of sequences, i.e., $SP(\mathcal{A}) := \sum_{\{S', S''\}} d_{\mathcal{A}}(S', S'')$ (where it is assumed that $c(-, -) = 0$).

Pioneering work of Sankoff and co-authors [11, 12] led to an exponential-time dynamic programming solution to the SP-alignment problem. A straightforward implementation takes time proportional to $2^n l^n$, for a problem with n sequences each of length at most l. In typical real–life instances, while n is usually fairly small, l is in the order of several hundreds, and the dynamic programming approach turns out to be infeasible for all but tiny problems. In fact, constructing *optimal* alignments was shown to be computationally expensive (Wang and Jiang [15]).

Due to the complexity of the alignment problem, most existing algorithms are heuristics based on the so called "progressive" approach: The alignment is incrementally built by considering the sequences one at a time. Effective progressive alignment methods proceed by first finding a heuristic tree spanning the sequences, and then by using the tree as a guide for aligning them iteratively, as described in the following section.

6.1 Tree-based Progressive Alignments

A popular approach to multiple alignments is due to Feng and Doolittle [3] who showed how to use any tree to align a set of n sequences. The appeal of the approach is that for $n-1$ out of $n(n-1)/2$ pairs, the pairwise alignment induced is in fact optimal. Indeed, they showed that for any tree T over a set of sequences (viewed as vertices in a graph), there exists a multiple alignment $\mathcal{A}(T)$ of the sequences such that $d_{\mathcal{A}(T)}(S', S'') = d(S', S'')$ for all the pairs of sequences (S', S'') connected by an edge of T. This can be readily understood with the help of Figure 2. The final alignment is built as follows: (i) pick an edge of the tree and align recursively the sequences on both sides of the cut; (ii) align optimally the two sequences at the endpoints of the edge; (iii) use this optimal alignment to merge the sub–alignments into a complete solution, by inserting columns of gaps in a sub–alignment wherever the optimal pairwise alignment inserts a gap in one of the two sequences. This way, the cost in the resulting alignment for the two sequences at the endpoints of the edge is their edit distance, as claimed.

Typically, the cost function obeys the triangle inequality, and then the edit distance induces a metric over the space of all sequences. Hence, it is easy to compute upper bounds on the distance in the final alignment for pairs that are not endpoints of a tree edge: by the triangle inequality, $d_{\mathcal{A}(T)}(S', S'') \leq d(S', S'', T)$, where $d(S', S'', T)$ is length of the path in T between two sequences S' and S''. This inequality suggests that, if we want to minimize the total pairwise distance in the resulting multiple alignment, a good tree to use is one which minimizes the routing cost.

The idea of relating routing cost and SP value is exploited in the first approximation algorithm for the SP-alignment problem, due to Gusfield [5], which has a performance ratio of

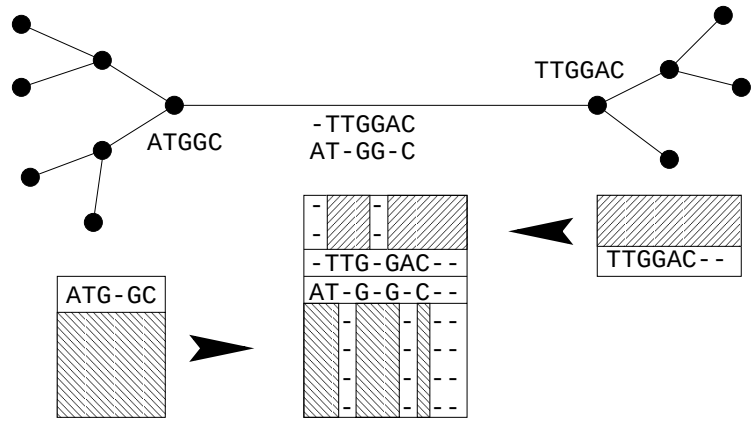

Figure 2: Feng and Doolittle's progressive alignment.

$2 - \frac{2}{n}$. Gusfield's approximation algorithm for the SP-alignment problem is based on Wong's 2-approximation for minimum routing cost trees [16]. Wong's algorithm considers, in turn, the shortest path tree rooted at every vertex, the best one having routing cost at most twice as large as the total cost of the graph itself. For complete metric graphs, every shortest path tree is isomorphic to a star. Furthermore, in this case, the cost of the graph is the sum of pairwise edit distances between sequences, which is a lower bound on the best SP–alignment cost. It then follows that a multiple alignment derived from the best star gives a 2–approximation for the SP–alignment problem [5].

In this paper we have pushed this idea further, by considering alignment trees of minimum routing cost (not necessarily stars). This approach has two immediate advantages over star alignments. First, an unrestricted tree may be more relevant than a star from an evolutionary point of view. Second, Gusfield reports that his approximation algorithm is not suited for families of very dissimilar sequences. In these cases, our alignment heuristic is preferable since for graphs with very dissimilar edge lengths, the minimum routing cost tree is almost never a star.

By the same argument as before, aligning via the minimum routing cost tree still gives a worst case performance guarantee of 2. However, as reported by Gusfield, this bound is really pessimistic and even the star alignment performs much better in practice (Gusfield reports

solutions within 15% of the lower bound, on average). In our computational results section we will show how choosing a better tree than a star typically yields an improvement of several percentage points on the average quality of the heuristic.

7 Computational results

Here we report on the extensive experiments carried out to evaluate all the ideas outlined in the paper. Since we have identified several aspects which can affect the final running time significantly (e.g., generation of columns, rows, LP–shortcut, branching rule, cut generation, etc.), trying out all the combinations of settings would result in a huge number of possibilities. Hence, we have chosen to evaluate them separately, when possible, in order to identify the best combination. For most options, there is a very sharp cut between one setting and another, clearly identifying the best choice. In some other cases (e.g. choice of the branching rule) the best setting is not completely clear. In the remainder of this section we first address these issues separately, so as to identify the best setting of all parameters. Then, we report on the computational results for the algorithm when all parameters are set to their best value. In the final part of the section, we describe the computational results for the multiple alignment problem. All algorithms were coded in C and run on a Pentium Celeron PC, 300 MHz with 64 MByte RAM, under Linux RedHat 6.0, and LP solver CPLEX 5.0.

7.1 Branch and Bound

• **Instance generation.** To test our algorithms, we have used two types of graphs, namely *Euclidean* and *random* graphs. A Euclidean graph on n vertices is obtained by choosing n points uniformly at random in the unit square. The lengths of the edges coincide with the Euclidean distances multiplied by n^2 and rounded to the nearest integer. In a random graph, the lengths of the edges are random integers in the range $\{1, \ldots, n^2\}$. To generate a graph with m edges, we first pick a spanning tree uniformly at random, and then add $m - (n-1)$ random edges uniformly chosen among the remaining edges. In our tests we have used *complete* graphs with

n	l_{IP1}	l_{IP2}	l_{COMB}	l_{DP}	l_{SUM}	UB
10	1.000	.958	.955	.963	.834	1.015
10	1.000	.960	.961	.968	.886	1.000
10	1.000	.967	.940	.960	.888	1.011
10	1.000	.997	.979	.997	.929	1.000
10	1.000	.898	.892	.913	.820	1.000
10	1.000	.875	.864	.961	.775	1.031
10	1.000	.909	.910	.944	.884	1.023
10	1.000	.904	.900	.918	.848	1.000
15	1.000	.875	.862	.928	.836	1.047
15	1.000	.878	.872	.896	.824	1.040
15	1.000	.867	.867	.902	.810	1.065
15	1.000	.866	.861	.876	.807	1.050
15	1.000	.894	.895	.924	.858	1.076
15	1.000	.869	.867	.892	.804	1.036
15	1.000	.871	.867	.907	.821	1.031
15	1.000	.895	.891	.902	.847	1.034
20	1.000	.889	.890	.914	.865	1.088
20	1.000	.875	.875	.899	.843	1.064
20	1.000	.893	.893	.917	.873	1.066
20	1.000	.870	.870	.915	.842	1.060

Table 2: Comparisons of various lower bounds and primal heuristic.

$m = \binom{n}{2}$, *dense* graphs with $m = \max\{\binom{n}{2}/3, 4n\}$ and *sparse* graphs with $m = 3n$. From our results, it appears that complete Euclidean graphs provide the most difficult instances.

• **Choice of the formulation.**

To choose the best algorithm, we started with a comparison of lower bounds. In Table 2 we report in each row the value of our five lower bounds (see section 3) and of our initial heuristic solution for 20 complete Euclidean graphs on 10, 15 and 20 vertices. The values are normalized so as $l_{\mathrm{IP1}} = 1.0$. From this table it appears that l_{IP1} is a much better bound than l_{IP2}, while l_{IP2} and l_{COMB} are comparable (their ratio is 1.005 on average). The bound l_{DP} is the second best, but its use is restricted to very small problems; in fact, on our hardware its computation required about 1 second for $n = 10$, 5 seconds for $n = 12$, 4 minutes for $n = 15$, and 18 hours for $n = 20$.

The bound l_{COMB} is an improvement over Dionne and Florian's bound [2] for the minimum communication cost network, when applied to routing trees. In [2], the only complete graphs solved had 10 vertices and no description is given on how they were generated. Further, the

problem was to minimize the communication cost of a generic network, which generalizes our MRCT. We have therefore used our implementation of Dionne and Florian's bound for MRCT, and it turned out that this bound is too weak to solve even Euclidean complete graphs of 15 vertices within 1 hour of CPU time. From Table 2 it appears that the gap $UB - l_{\mathrm{COMB}}$ at the root node is of the order of 20 percent, which is too large to be filled quickly by branch and bound (where UB is computed by the local search heuristic described in section 5.1).

The same comments apply to our model (IP2), which is not appropriate for complete graphs of more than 15 vertices. In fact, although the LPs are much faster to solve than for model (IP1), the bound is weaker and pruning occurs only very deeply down the search tree, so that the overall running time is very large. We also investigated possible strengthenings of (IP2) given by the inclusion of constraints that force a minimum number of paths to use an edge in the tree. However, the involved overhead was not compensated for by the increased value of the bound.

Hence the only choice for the best model is between (IP1) and (MCF). Model (MCF) can be very slow to solve, because of its large number of rows and columns. The time for solving the LP at the root node, for complete graphs on 15 vertices, was 890 seconds (on average) by using primal simplex on (MCF). With the feature NETOPT of CPLEX (specifically designed for problems whose constraint matrix is related to a network), this time decreased to just 40 seconds. However, this is still worse than the time needed when using formulation (IP1), which is 31 seconds on average. Furthermore, because of a larger memory requirement, there were several instances (including all instances with $n \geq 25$) that ran out of memory with (MCF) but were solved with (IP1). Hence, a branch and bound method based on the model (IP1) seems to be the best algorithm among the ones we studied, so we concentrate on it in the sequel.

• **Dealing with degeneracy.** As we observed in section 2, our LP model is very degenerate. The presence of all rows causes two types of problems during column generation. First, each LP is very expensive because of its size. Second, we may go through long runs of LPs in which the objective function value does not change. An example of this behavior is the following. For a complete Euclidean graph on 20 vertices, 19 pricings and 1066 seconds were needed before the

first change, of 0.06%, in the LP value, which arose after generating 2245 columns, with full row size. The final LP value, at the end of column generation, was 93.8% of the starting one, and was obtained after 2631 seconds and 144 LPs. The same bound was obtained in 351 seconds and 359 LPs with column *and* row generation, the final LP having 3342 rows instead of 36291. We also solved 5 Euclidean instances of sizes 10, 15, 20 and 25 vertices each. For each instance we ran the LP with only column generation, and with both row and column generation. The results were roughly the same for each size: the running time with row generation is within 10%–20% of the time without row generation, and the final number of rows is about 10% of the total. The number of LPs solved when all rows are present is about 40% of the number of LPs solved with row generation.

Given that generating both rows and columns is the best way to go, we then determined the most effective way of doing so. Given a current LP solution, *pricing* is the procedure which checks if all variables price–out correctly, and, if not, returns a set of new variables to add to the LP. Similarly, *separation* is the procedure which checks if all constraints are satisfied, and, if not, returns a set of violated constraints. A pricing (separation) is called *successful* if it finds new variables (constraints) to add to the LP. If the last procedure called was a successful pricing (separation) then the next one will also be pricing (separation) if we *loop through columns* (*rows*), or otherwise it will be separation (pricing). Hence, there are four possible ways of proceeding, depending on looping through column (LC) and looping through rows (LR) being true or false. On the same instances as before, we obtained comparable results for all sizes. For instance, for $n = 20$ we found the average values reported in Table 3 for total number of rows r, columns c, LPs l and time t (all entries are normalized so as to be 1.0 for LC = LR = FALSE). The strategy of LP-shortcutting requires to use LC=LR=TRUE. This setting, in combination with LP–shortcutting, outperforms the setting LC=TRUE, LR=FALSE, resulting about twice as fast, amortized over all nodes of the search tree. Hence we have used LC=LR=TRUE.

• **LP shortcutting.** As explained in section 4, the scheme of alternating the generation of variables and constraints provides us with a family of lower bounds $l'(k)$ at the end of each k–th phase of column generation. The quality of these bounds improves very rapidly at the

	LC=FALSE				LC=TRUE			
	r	c	l	t	r	c	l	t
LR = FALSE	1.00	1.00	1.00	1.00	0.91	0.99	1.14	0.60
LR = TRUE	1.02	0.99	1.49	1.07	1.01	1.04	1.39	0.89

Table 3: Looping through columns and/or rows.

k	$n = 20$		$n = 20$		$n = 25$		$n = 25$	
	$l'(k)$	t	$l'(k)$	t	$l'(k)$	t	$l'(k)$	t
1	0.912	0.02	0.937	0.01	0.908	0.00	0.923	0.00
2	0.939	0.06	0.961	0.02	0.927	0.00	0.943	0.00
3	0.960	0.11	0.972	0.05	0.948	0.01	0.959	0.01
4	0.971	0.17	0.981	0.09	0.967	0.02	0.972	0.03
5	0.982	0.24	0.988	0.16	0.977	0.05	0.980	0.06
6	0.989	0.31	0.993	0.25	0.983	0.09	0.986	0.12
7	0.995	0.43	0.995	0.38	0.989	0.14	0.990	0.16
8	0.998	0.61	0.997	0.47	0.993	0.23	0.994	0.23
9	0.999	0.70	0.999	0.62	0.995	0.30	0.997	0.32
10	0.999	0.88	0.999	0.71	0.997	0.39	0.998	0.43
∞	1.000	1.00	1.000	1.00	1.000	1.00	1.000	1.00

Table 4: Trade–off between lower bound value and time to compute it.

beginning, while later the improvements are quite slow and costly. In Table 4 we show various lower bounds $l'(k)$ and time to compute them as a function of k, for four Euclidean graphs, two on 20 and two on 25 vertices. With LP-shortcutting active, we use the bound $l'(6)$, unless the relative distance to the incumbent solution is less than 0.5%, in which case we use $l'(\infty)$. LP–shortcutting implies also preemptive branching whenever the lower bound is provably not strong enough to fathom the node. We have tested LP-shortcutting on 100 problems of the most difficult type (complete, Euclidean graphs) with 10 to 25 vertices. All problems were solved more quickly with shortcutting active. The average saving in solving a problem with shortcutting was 44%. The maximum saving was 91% (achieved on 3 problems), the minimum 28% (on just one problem).

• **Variables and cuts.** A typical behavior of column–generation algorithms is that the running time and number of LPs solved is inversely proportional to the number of columns generated per iteration. This behavior was observed also in our case. For instance, solving a typical Euclidean problem on 15 vertices takes more than 600 seconds and more than 2500 LPs when only one

column is generated at a time. Generating 5 columns at a time already halves these numbers. Since in our model we find shortest paths for each pair, we can generate up to $\binom{n}{2}$ columns at a time: by generating as many paths as possible at each pricing, the problem is solved in about 100 seconds. Finally, to be able to generate even more than $\binom{n}{2}$ columns at each pricing call, we modified Dijkstra's algorithm as follows. In section 2 we showed that a variable x_P with $P \in P^{ij}$ has negative reduced cost if $d'(P) < u_{ij}$. Thus, whenever in Dijkstra's algorithm j is labeled with a value smaller than u_{ij}, we have found a good column to generate. Hence, we do not only keep track of the *shortest* path, but also of other paths of negative reduced cost. With this enhancement, the running time and number of LPs went down, on average, of an extra 3%.

As far as the cutting planes are concerned, the most effective ones are the subtour–elimination constraints (19). We ran 20 instances with and without the use of these cuts, and observed that the average number of nodes of the search tree decreased by 8% when using (19). The running time, however, decreased only by 4.5%, because of the extra work implied by cut separation. For the instances of our test–bed, instead, no significant improvement is achieved by using the other families of cuts.

• **Final performance.** We finally ran the algorithm with the above optimal settings. These include column generation with Dijkstra variant for generating a maximum number of columns, row generation, LP–shortcut with $\alpha = 0.5\%$, use of cutting planes (19), and branching rule V2 with choice of the branching variable according to the Kruskal–like scheme. In Table 5 we report the results for 210 graphs on 10, 15, 20, 25, 30, 40 and 50 vertices. For each size we have generated 5 Euclidean and 5 random (complete, dense, and sparse) graphs. We set a time limit of 3600 CPU–seconds. Each row reports the number of problems solved (out of 5). For the problems solved to proven optimality, we report the average (maximum) number of nodes of the search tree, number of LPs solved, and running time in seconds. As anticipated, it turns out that Euclidean problems are more difficult than random ones: we can solve them for up to size 30 vertices within the one–hour time limit. For random problems we can solve instances up to size 40 and, rarely, size 50. In our runs we found a large variance in the hardness of the problems, expecially for the random ones, which are very sensitive to the range in which

n	Euclidean				rnd			
	solved	nodes	lps	time	solved	nodes	lps	time
C 10	5/5	5 (9)	36 (58)	0 (1)	5/5	1 (1)	7 (9)	0 (0)
C 15	5/5	19 (31)	363 (523)	96 (155)	5/5	7 (15)	92 (153)	6 (16)
C 20	5/5	91 (277)	1784 (5404)	587 (1476)	5/5	5 (9)	206 (312)	84 (136)
C 25	4/5	44 (52)	2017 (2415)	1832 (2028)	5/5	15 (23)	316 (419)	335 (531)
C 30	2/5	31 (40)	583 (658)	2918 (3341)	5/5	45 (119)	896 (2181)	885 (2320)
C 40	0/5	-	-	-	3/5	19 (26)	546 (853)	2962 (3501)
C 50	0/5	-	-	-	0/5	-	-	-
D 10	5/5	1 (3)	10 (22)	0 (0)	5/5	1 (1)	8 (19)	0 (0)
D 15	5/5	65 (125)	566 (972)	34 (63)	5/5	12 (27)	108 (254)	6 (18)
D 20	5/5	32 (59)	345 (621)	86 (159)	5/5	17 (61)	252 (838)	52 (191)
D 25	4/5	23 (65)	484 (1212)	256 (631)	5/5	25 (46)	411 (727)	262 (570)
D 30	5/5	101 (157)	586 (2313)	1995 (2412)	5/5	21 (27)	348 (465)	408 (632)
D 40	0/5	-	-	-	3/5	18 (23)	768 (1117)	2614 (3421)
D 50	0/5	-	-	-	1/5	15 (15)	759 (759)	3419 (3419)
S 10	5/5	1 (1)	3 (6)	0 (0)	5/5	2 (5)	15 (24)	0 (0)
S 15	5/5	20 (71)	131 (435)	7 (27)	5/5	1 (3)	13 (24)	0 (0)
S 20	5/5	14 (22)	154 (225)	30 (40)	5/5	14 (45)	177 (489)	45 (161)
S 25	5/5	21 (59)	275 (621)	109 (259)	5/5	10 (17)	180 (275)	96 (162)
S 30	4/5	33 (51)	726 (1061)	1904 (2334)	5/5	22 (29)	388 (583)	369 (493)
S 40	1/5	9 (9)	413 (413)	3291 (3291)	3/5	8 (15)	207 (411)	868 (1020)
S 50	0/5	-	-	-	1/5	11 (11)	952 (952)	3021 (3021)

Table 5: Results for Complete (C), Dense (D), and Sparse (S), graphs.

the random numbers are chosen. The instances of Table 5 were generated with lengths in the interval $\{1, \ldots, n^2\}$. To investigate further how the complexity of the problem depends on the variability of edge–lengths, we performed the following experiment. We generated 100 problems, for complete random graphs of $n = 30$ vertices. The problems are divided into 10 classes of 10 instances each. Each class contains graphs with edge weights drawn in the interval $d_{\min} = 10^k, \ldots, d_{\max} = 10^{k+i}$ for $k = 0, \ldots, 3$ and $i = k+1, \ldots, 4$. The results are reported in Table 6. Each instance was given a 2 hours maximum time. In the column "solved" we report the total number of instances solved and how many were solved during the 2nd hour of computation. From Table 6 one can colnclude that the complexity of the prblems decreases as the ratio $d_{\max}/d_{\min}$ increases.

7.2 Alignment problems

As outlined in section 6, we have developed a multiple alignment heuristic based on the minimum routing cost tree on the graph of sequences weighted by the edit distances. The heuristic has been

$d_{\min}$	$d_{\max}$	solved	nodes	lps	time
1	10	4,3	135 (212)	4843 (6758)	4996 (6792)
1	100	9,0	38 (170)	872 (3707)	482 (2486)
1	1000	10,0	43 (95)	899 (2022)	411 (1184)
1	10000	10,0	22 (43)	392 (767)	173 (499)
10	100	5,2	80 (136)	3252 (4127)	4314 (6469)
10	1000	10,1	119 (376)	2154 (6239)	1210 (3716)
10	10000	10,0	28 (65)	601 (1734)	329 (1060)
100	1000	6,4	66 (103)	4032 (5663)	4810 (6857)
100	10000	10,1	42 (118)	1126 (3646)	1061 (5001)
1000	10000	6,4	94 (117)	2839 (3493)	4214 (6194)

Table 6: Comparisons of various lower bounds and primal heuristic.

tested on several families of proteins obtained from the public–domain data base PRINTS [10]. Each family contains from 10 to 20 sequences, each with 200 to 600 amino acids. The results are reported in Table 7. For each instance we report the routing cost of the best star and of the best tree (this latter found by our MRCT algorithm), and the value of the final alignment obtained by using these trees within Gusfield's heuristic framework outlined in section 6. The score matrix used is due to Taylor [14]. The values have been normalized so that, for each instance, the SP cost of the best star alignment is 1000. Each MRCT instance was solved within a time limit of 10 minutes. From Table 7 we see that a reduction of roughly 10% in the routing cost of a MRCT solution with respect to the best star, corresponds to a proportional (about 6% on average) reduction in the alignment cost. By comparing the alignment value to some simple lower bound, we found that our heuristic value is within 8% from the optimum, on average.

8 Conclusions

In this paper we have developed, implemented and tested several algorithms for the minimum routing cost tree problem. Further, we have devised techniques for speeding up the search, which have proved very effective in practice. The problem, however, seems particularly hard to solve exactly, and the size of graphs which can be effectively attacked within 1 hour of computing time on a PC, is roughly 30–40 vertices. Beyond this size, the formulations are still useful, in

instance	best star		best rc tree	
	rc value	SP value	rc value	SP value
ABHYDRO	1398	1000	1263	941
AOTCASE	1259	1000	1232	979
BARWIN	1189	1000	1063	864
CAT-I	1191	1000	1074	814
CAT-II	1186	1000	1142	848
CAT-III	1428	1000	972	859
CAT-IV	1260	1000	856	749
CAT-V	1230	1000	946	789
CAT-VI	1175	1000	1078	876
CAT-VII	1241	1000	1115	900
CAT-VIII	1200	1000	1042	861
CRB1	1370	1000	1158	1002
CRB2	1420	1000	1280	983
DHFR1	1427	1000	1242	972
EGF-I	1268	1000	1176	965
EGF-II	1276	1000	1197	982
EGFTGF	1329	1000	1207	973
FUM	1335	1000	1275	911
GPROT	1203	1000	1159	999
HTHRSR	1441	1000	1434	995
KRINGLE	1356	1000	1253	990
LYZL1	1136	1000	1054	922
LYZL2	1451	1000	1292	981
MC582-12	1404	1000	1395	1001
MC586-10	1363	1000	1359	984
NGF	1273	1000	1217	998
NIT	1372	1000	1309	976
POTX1	1140	1000	1112	985
POTX2	1320	1000	1240	939
RIBO	1171	1000	1156	936
STRO1	1298	1000	1261	990
STRO2	1308	1000	1267	986
STRO3	1178	1000	1008	975
STRO4	1311	1000	1126	924
VACATP	1246	1000	1221	986

Table 7: Alignment problems from the data base PRINTS.

conjunction with local search heuristics, to provide an upper bound on the approximation error. Since the lower bound is reasonably tight, after one hour we can stop the search and have a solution which is, on average, within 1% from optimality.

The IP models described here can be extended to the more general case of minimum communication cost network.

Finally, our application to Computational Biology shows, once more, the usefulness of Optimization and Mathematical Programming techniques when applied to different domains. We believe that these methodologies can lead to the design of powerful new alignment programs.

Acknowledgments

The authors wish to thank two anonymous referees for their precious comments and suggestions. The work of the first two authors was partially supported by M.U.R.S.T. through a project PRIN on Computational Biology. The work of the second and third authors was partially supported by M.U.R.S.T. through the project MOST.

References

[1] R. K. Ahuja, T. L. Magnanti and J. B. Orlin, *Network flows: theory, algorithms and applications*, Prentice–Hall, Englewood Cliffs, New Jersey, USA, 1993.

[2] R. Dionne and M. Florian, Exact and approximate algorithms for optimal network design, *Networks* 9:1 (1979), 37–60. 1979.

[3] D. Feng and R. Doolittle. Progressive sequence alignment as a prerequisite to correct phylogenetic trees. *J. Molec. Evol.* 25 (1987), 351–360.

[4] M. Fischetti, G. Lancia and P. Serafini, Branch–and–Price Algorithms for the Minimum Routing Cost Tree Problem, Technical Report, Dept. of Electronics and Informatics, University of Padova, 2001

[5] D. Gusfield, Efficient methods for multiple sequence alignment with guaranteed error bounds, *Bulletin of Mathematical Biology* 55 (1993), 141–154.

[6] T. C. Hu, Optimum communication spanning trees, *SIAM J. Comp.* 3 (1974), 188–195.

[7] D. S. Johnson, J. K. Lenstra and A. H. G. Rinnooy Kan, The Complexity of the Network Design Problem, *Networks* 8 (1978), 279–285.

[8] G. L. Nehmauser and L. A. Wolsey, *Integer and Combinatorial Optimization*, John Wiley and Sons, NY, 1988.

[9] PORTA, http://www.iwr.uni-heidelberg.de/iwr/comopt/software/PORTA. a POlyhedron Representation Transformation Algorithm.

[10] PRINTS, http://bmbsgi11.leeds.ac.uk/cgi-bin/prints.sh

[11] D. Sankoff and J. B. Kruskal, eds. *Time warps, string edits and macromolecules: the theory and practice of sequence comparison*, Addison Wesley, Reading, MA, 1983.

[12] D. Sankoff, C. Morel and R. J. Cedergren, Evolution of the 5S Ribosomal RNA, *Nature New Biology* 245 (1973), 232–234.

[13] A. J. Scott, The Optimal Network Problem: Some Computational Procedures, *Transportation Research* 3 (1969), 201–210.

[14] W. R. Taylor and D. T. Jones. Deriving an Amino Acid Distance Matrix, *J. Theor. Biol.* 164 (1993), 65–83.

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

[16] R. Wong, Worst–case Analysis of Network Design Problem Heuristics, *SIAM J. Algebr. Discr. Meth.* 1 (1980), 51–63.

[17] B.Y. Wu, G. Lancia, V. Bafna, K.M. Chao, R. Ravi, C.Y. Tang, A Polynomial–time Approximation Scheme for Minimum Routing Cost Spanning Trees, *SIAM J. Comp.* 29:3 (1999), 761–778.