PROBABILISTIC TABU SEARCH FOR TELECOMMUNICATIONS NETWORK DESIGN

Jiefeng Xu*, Steve Y. Chiu**, and Fred Glover*

*Graduate School of Business, University of Colorado at Boulder, CO 80309-0419.

** GTE Laboratories, Inc., 40 Sylvan Road, Waltham, MA 02254

Abstract. This paper presents a computational study of a network design problem arising in the telecommunication industry. The objective can be formulated as that of finding an optimal degree constrained Steiner tree in a graph whose nodes and edges are weighted by costs. We develop a probabilistic Tabu Search heuristic for this problem, addressing issues of move evaluations, error correction, tabu memories, candidate list strategies and intensification strategies based on elite solution recovery. Computational results for a test set of difficult problem instances show that the new heuristic yields optimal solutions for all problems that could be solved by exact algorithms, while requiring only a fraction of the solution time. In addition, for larger and more realistic sized problems, which the exact methods were unable to solve, computational results show our method also outperforms the best local search heuristic currently available.

Key words. Steiner Tree-Star, Tabu Search, Telecommunications Network Design.

This research was supported in part by the Air Force Office of Scientific Research AASERT grant #F49620-92-J-0248.DEF.

Published in *Combinatorial Optimization: Theory and Practice*, Vol. 1, No. 1 (1996), pp. 69-94.

1. Introduction

Designing a cost-effective digital network is a critical issue in the success of today's telecommunication industry. In this paper, we address a problem of designing a private line digital data service (DDS) network over a finite set of customer locations. To provide mutual communication between customers, each customer location must be connected to its designated wire center that in turn needs to be connected to one of the data switching offices with bridging capabilities, called a hub. Furthermore, if more than one hub is chosen to be active, those active hubs must be interconnected by digital lines so that the customers connected with different wire centers can communicate with each other via the resulting hub network.

The costs involved in DDS network design include the fixed bridging costs for every active hub in the network, the variable bridging costs for each incoming and outcoming lines in all active hubs, and the transport costs for each link between a customer location and its designated wire center, or between a wire center and its assigned hub, or between two active hubs. These costs are calculated according to the tariff charges established by the Federal Communications Commission (FCC). The problem is to design such a network that minimizes the total costs. Figure 1 illustrates a simple DDS network scenario.

As shown in this diagram, the number of lines between a wire center and its assigned hub is equal to the number of customer locations connected to that wire center.

==============================

Please Insert Figure 1 Here

==============================

The DDS network design problem can be simplified by reference to a Steiner Tree framework. (For more on Steiner Tree problems, see for example the survey by Duin and Voβ, 1994.) It is well known that the most cost effective architecture for interconnecting all active hubs is a minimum spanning tree over these active hubs. Since the linking cost per line between a wire center and a potential hub is fixed and the bridging cost per line for that hub is also known, we can pre-calculate the cost of connecting a customer location to a hub by adding up these two terms. Thus, the intermediate wire centers can be eliminated and therefore the DDS network problem can be converted into an extension of the Steiner Tree Problem, where the hubs define the Steiner nodes and the customer locations define the target nodes. More specifically, the new problem can be formulated as an extension of the standard Steiner Tree Problem that includes the addition of node-associated weights (fixed bridging costs for the hub) and of degree constraints on the target nodes. This problem was first investigated by Lee *et al* (1994) and then Lee, Chiu and Ryan (1995).

Since the target nodes form a star topology around the active Steiner nodes, and the active Steiner nodes are connected as a tree, this problem is also called a Steiner tree-star (STS) problem. Figure 2 illustrates a solution for the STS problem that corresponds the DDS network scenario in Figure 1.

==============================

Please Insert Figure 2 Here

==============================

Lee, Chiu and Ryan (1995) show that the STS problem is strongly NP-hard, investigate valid inequalities and facets of the underlying polytope of the STS problem, and implement a branch and cut procedure for solving the problem exactly. A separation procedure based on a maximal flow algorithm is devised to find the violated facets if the resulting relaxation solutions are fractional. The branch and cut procedure is also equipped with a specially designed heuristic to find a high quality upper bound solution within acceptable time bounds.

As in most other applications involving strongly NP hard problems , the computational complexity of the problem limits the size of problems that can be solved optimally. The branch and cut procedure of Lee, Chiu and Ryan (1995) is able to solve 100 node instances of the STS problem, but encounters severe difficulties for problems beyond that size. The design of an efficient heuristic therefore becomes of paramount importance for dealing with problems of larger sizes, and constitutes the key focus of our research.

In this paper, we explore an implementation of probabilistic Tabu Search (TS) for the STS problem. For a comprehensive overview of Tabu Search and its applications, see Glover and Laguna (1993) and Glover (1995).

This paper is organized as follows. Section 2 describes the probabilistic TS based heuristic for the STS problem and examines several relevant issues, such as long term memory, probabilistic move selection, neighborhood structure, move evaluation, error correction, candidate list strategies and an advanced restart/recovery strategy. In section 3, we report computational results on three sets of specially designed test problems, accompanied by comparisons with the solutions obtained by the exact algorithm and a heuristic approach called LS, which was found by Lee, Chiu and Ryan (1995) to be very effective for this problem. In addition, we provide an extension and improvement of the LS heuristic and report its performance. Section 4 summarizes our methodology and findings. In addition, we include a description of the LS heuristic in Appendix A and provide a high-level pseudocode in Appendix B to facilitate the reading and comprehension of our TS algorithm.

2. The Probabilistic TS Based Heuristic

We begin by indicating an apparent property of the STS problem that is fundamental for our heuristic design.

STS Property: In an optimal STS solution, every target node is linked to an active Steiner node that gives a minimum cost connection for the target node (over the active Steiner node set). Consequently, once the active Steiner nodes are selected, the links for the target nodes can be determined immediately.

The foregoing property is a trivial consequence of the STS definition. By reference to it, we focus our search on the goal of finding the active Steiner node set associated with an optimal solution. Once such a set is found, an optimal solution can be thereby discovered by first constructing a minimum spanning tree on the active Steiner nodes and then linking every target node to its cheapest active Steiner node (i.e. the active node to which it connects by the least cost link).

2.1. Neighborhood Structure and Moves. We divide the Steiner nodes into the disjoint subsets of active nodes (A) and inactive nodes ($\bar{\mathsf{A}}$). The moves that define the neighborhood structure for our procedure consist of transferring a chosen node from one of these two subsets to another, and of exchanging two nodes between these subsets. Specifically, we divide the transfer moves into the following two elementary types:

(1) constructive move: transfer a selected Steiner node from $\bar{\mathsf{A}}$ to A. This move increases the cardinality of the set A by one and thereby forms a new spanning tree by adding a node. This move is disallowed if the set $\bar{\mathsf{A}}$ is empty;

(2) destructive move: transfer a Steiner node from A to $\bar{\mathsf{A}}$. This move reduces the cardinality of the set A by one, deleting the active Steiner node from the current spanning tree, and requiring a new smaller spanning tree to be constructed. This move is disallowed if the set A is empty.

Any set A can be reached via a sequence of constructive and/or destructive moves starting from any solution configuration. Thus, constructive and destructive moves are considered to be elementary moves in the search process. Pairwise exchange (swap) moves, which exchange one active Steiner node with one inactive Steiner node, can be viewed as a combination of a constructive and a destructive move. Such a combined move leaves the cardinalities of both set A and $\bar{\mathsf{A}}$ unchanged, but requires the spanning tree to be reconstructed. The swap move is disallowed if either the set A or $\bar{\mathsf{A}}$ is empty.

Because a swap move involves a more significant change in the spanning tree (and hence requires a more complex evaluation of its consequences), we perform it more sparingly in the search process. In particular, we apply it chiefly to produce periodic perturbation and conditional oscillation. A perturbation step is guided by elementary moves and executed once for every certain number of iterations. The conditional oscillation step is designed to achieve a greater intensification of the search, by executing swap moves for some number of iterations when the search cannot improve the solution for a pre-defined duration. This two-level mechanism proves effective and efficient in our application, since we find that a dominant reliance on the elementary moves, when handled intelligently, yields good decisions with only occasional reliance on more complex moves.

2.2. Move Evaluation. Once the subset A is determined, the cost of the current STS solution can be calculated by: (1) constructing a minimum spanning tree over A and identifying the resulting cost, plus (2) linking every target node to its cheapest-link Steiner node and finding the sum of the resulting connection costs. The second part can be easily implemented by maintaining a presorted list for every target node, which records the connection costs from this target node to every Steiner node. Thus, the corresponding connection costs can be found in linear time for each target node. The key issue of the move evaluation therefore becomes the spanning tree construction.

Though the minimum spanning tree problem can be solved trivially over a finite node set, it is very expensive computationally to do this for every move evaluation. Instead, a good and quick cost estimate of the new spanning tree is desired.

Constructive Move Estimate. A simple cost estimate suffices for a constructive move by linking the added Steiner node to its cheapest Steiner node in the tree. This increases the cost of the spanning tree by adding one new edge. The resulting spanning tree is not necessarily the minimum one and so the estimate is an upper bound on the true minimum cost. However, the added edge belongs to a minimum spanning tree for this choice of active nodes.

Destructive Move Estimate. The destructive move estimate becomes more complex, since dropping a Steiner node deforms the spanning tree. We use a candidate list strategy to simplify the estimate procedure and to concentrate attention on a promising subset of moves. In particular, we avoid evaluating the whole neighborhood by restricting attention to destructive moves involving only those Steiner nodes whose degree does not exceed three in the current spanning tree. The prototypes for reforming the new spanning tree are established for the cases of Steiner nodes with degree one, two, three respectively.

These are illustrated in Figure 3 where the dotted line represents the link before the move, the solid line represents the link after the move and the bold circle denotes the Steiner node that is moved.

==============================

Please Insert Figure 3 Here

==============================

In (c) of Figure 3, the reconnection consists of adding the two shorter edges of the indicated triangle (formed by the three nodes previously connected to the node dropped). Once again, these estimates provide a quick approximation of the optimal cost of the new spanning tree.

Swap Move Estimate: The swap move estimate makes use of both the destructive and constructive move estimates. Consider two Steiner nodes x and y where $\mathsf{x} \in \mathsf{A}$ and $\mathsf{y} \in \bar{\mathsf{A}}$, the swap of x and y can be performed by first dropping node x and then adding node y. We estimate the swap cost by first calculating the cost of dropping node x, and then adjusting the cost by adding node y to the newly formed spanning tree.

For a swap move evaluation, effort must be taken to reduce the computational expense when the number of Steiner nodes is moderately large. For that purpose, a natural candidate list is constructed to isolate a promising subset of the swap moves. This candidate list restricts attention to pairs (x, y) whose elements are drawn from the k best destructive and constructive moves where k is an integer in the range of 5 to 15. This candidate list strategy is loosely supported by the idea of the Proximate Optimality Principle (POP) that says good solutions at one level (for example, produced by either the destructive or the constructive move) are likely to be found close to good solutions at an adjacent level (i.e., produced by the swap move). As a consequence, this candidate list is used to screen for the good partial moves whose composition may give a good candidate to evaluate. Such a candidate list strategy proves to be much faster than evaluating the whole swap neighborhood, yet yields comparably good outcomes, as the computational tests disclose in section 3.

Since the swap moves are performed only periodically (conditionally), we collect the k best destructive and constructive moves only before such a series of swap moves is to initiated. Furthermore, the collection of the good partial moves makes use of pool storing and isolating a certain number of best moves for probabilistic move selection, as described later.

2.3. Error Correction.. The approximations used for move evaluation raise the issue of error correction. The error – that is, the difference between the estimated and true cost of a minimum spanning tree – can be readily corrected by applying a minimum spanning tree algorithm to the current active Steiner node set. This correction process is conducted when the estimated cost discloses that a “new best” solution is found (since the estimate can never understate the true cost). It is also performed on a periodic basis to counter a progressive accumulation of error. This periodic correction seeks to balance the tradeoff between expected accuracy and speed of executing the algorithm.

To achieve this, we manipulate a priority queue that includes a selected number of elite solutions encountered so far during the search, where these solutions consist of those actually visited (without error correction) and also of those that may potentially be visited by means of currently available candidate moves. The priority queue is ordered by the estimated costs of its component solutions. Error correction is then periodically performed on each element in this queue. Once an element’s true cost is thus identified, this element is marked so that no repetitive error correction is executed on this element in the future. At the same time, the element is repositioned in the queue according to its new cost. Thus, when a new elite element is encountered whose estimated cost is better (smaller) than the cost of the current worst element of the queue, the new element is added and the worst element is dropped from the queue. Because of periodic updating, the costs associated with queue elements can be a mix of true costs and estimated costs. The updating of the priority queue is further enhanced by applying a sorted pointer list to facilitate the add and drop operations. As a byproduct, this elite list can also serve as a pool of points for advanced restarting. We describe this option later.

The cost approximation inevitably has a significant influence on move selection. To further compensate for the effects of approximation, we also use a move selection rule based on probabilistic tabu search, as described in subsection 2.5.

2.4. Tabu Search Memories. The core of the algorithm is the use of TS memory structures to guide the search process. A short term memory is employed principally to prevent the search from being trapped in a local optimum, and to introduce vigor in the search process. A long term memory is used to handle more advanced issues, such as intensification and diversification.

Short Term TS Memory. The short term memory operates by imposing restrictions on the composition of new solutions generated (typically expressed as a restriction on *attributes* of these solutions). For elementary moves, we impose restrictions that assure a move cannot be "reversed". In particular, if the node x is added to the active Steiner node set A (transferred from $\bar{\mathsf{A}}$ to A, we forbid this node to be transferred back to $\bar{\mathsf{A}}$ for several iterations. Similarly, if the node y is dropped from the active node set A (transferred from A to $\bar{\mathsf{A}}$, we forbid the node to be transferred back to A for several iterations. For swap moves, we impose the tabu restrictions on moves in both directions. If an active node x is swapped with an inactive node y on the current move (x is transferred from A to $\bar{\mathsf{A}}$ and y is transferred from $\bar{\mathsf{A}}$ to A), the restriction inhibits both transferring node x back to A and transferring node y back to $\bar{\mathsf{A}}$. Such a restrictive mechanism prevents the search from revisiting a local optimum in the short term and greatly diminishes the chance of cycling in the long term.

How long a given restriction operates depends on a parameter called the tabu tenure, which identifies the number of iterations a particular tabu restriction remains in force. The tabu tenure can be either fixed or variable, but a tenure that varies within a small range about a central value often proves more robust. Moreover, in our application, we allow the central value to differ according to the move type. Since a constructive move that adds a node to A introduces a fixed cost, and thus makes the move appear less attractive than a destructive move, we assign a longer tabu tenure to avoid destructive moves than to avoid constructive moves. Customary criteria, by contrast, would assign tabu tenures based on the relative sizes of A and $\bar{\mathsf{A}}$. That is, longer tenures would be given to placing a node back in the smaller set, because there are more options to move a node from the larger set to the smaller set than to move a node in the opposite direction. We have not tested this alternative in this study.

An important TS component is the use of aspiration criteria to allow a restriction to be overridden if the outcome of the move is sufficiently desirable. A commonly-used criterion is to override the restriction if the current candidate move would lead to a new best solution. However, as stated before, our move costs are estimated and therefore contaminated with some “noise”. Accordingly we use a simple but a more robust aspiration criterion to accept any move that leads to a solution better than the third best, provided the move does not duplicate the cost of either of the two current best solutions. To apply this criterion, we perform an immediate correction of any cost that falls in this high category. Again, the top three best solution can be readily found from the ordered elite list.

Short term memory can be efficiently implemented using a recency-based memory structure. To illustrate this, let iter denote the current iteration number and let $\mathsf{tabu\_add(x)}$ and $\mathsf{tabu\_drop(y)}$ denote the future iteration values governing the duration that will forbid a reversal of the moves of adding node x and dropping node y (i.e. by preventing node x from being dropped and node y from being added). Similarly, let $\mathsf{tabu\_add\_tenure}$ and $\mathsf{tabu\_drop\_tenure}$ be the values of tabu tenures for these two moves. Initially, $\mathsf{tabu\_add(z)}$ and $\mathsf{tabu\_drop(z)}$ are set to zero for all nodes z, and iter starts at one. When the TS restriction is imposed, we update the recency memory as follows:

$\mathsf{tabu\_drop(x)} = \mathsf{iter} + \mathsf{tabu\_drop\_tenure}$ (for the constructive move of adding node x, to prevent x from being dropped),

$\mathsf{tabu\_add(y)} = \mathsf{iter} + \mathsf{tabu\_add\_tenure}$ (for the destructive move of dropping node y, to prevent y from being added).

Once a node x has been added, by transferring it from $\bar{\mathsf{A}}$ to A, we refer to it notationally as a node y in A, to consider the possibility of dropping such a node by transferring it from A to $\bar{\mathsf{A}}$. Thus the restriction to prevent such a node y from being dropped is enforced when tabu_drop(y) > iter. Similarly the restriction to prevent a node x from being added (where x was previously denoted as y when dropped) is enforced when tabu_add(x) > iter. As previously noted, we select the central value for tabu_add_tenure to be smaller than that of tabu_drop_tenure. Let best_move_cost be the evaluation (estimated cost) of the move we select. Also define cost(.) as the move evaluation value. Then the move selection procedure incorporating the TS restrictions and aspiration criteria proceeds as follows:

```
Assign a large value to best_move_cost.
For each candidate inactive Steiner node x, do
    if cost(x) < best_move_cost do
        if the aspiration criterion is satisfied or tabu_add(x) <= iter do
            best_move_cost = cost(x).
For each candidate active Steiner node y, do
    if cost(y) < best_move_cost do
        if the aspiration criterion is satisfied or tabu_drop(y) <= iter do
            best_move = cost(y).
```

For the exchange move, we have

```
Assign a large value to best_move_cost.
For each candidate pair of inactive node x and active node y, do
    if pair (x, y) is in candidate list, do
        if cost(x, y) < best_move_cost do
            if cost(x, y) < best_sol_cost or
            ( tabu_add(x) <= iter or tabu_drop(y) <= iter) do
                best_move_cost = cost(x, y).
```

Long Term TS Memory. The long term TS memory we employ makes use of a frequency based memory structure to achieve a diversification effect, encouraging the search to explore regions less frequently visited.

More specifically, we use this memory to discourage moves that occurred frequently during the search (and consequently to encourage moves that occurred less frequently). A transition measure is used to record the number of times each Steiner node changes from an active status to an inactive status or vice versa. Let $\mathsf{frequency0}(\mathsf{x})$ be the number of times that Steiner node x is changed from *active* to *inactive*, $\mathsf{frequency1}(\mathsf{x})$ be the number of times that Steiner node x is changed from *inactive* to *active*. These frequencies can easily be updated as follows:

$$\mathsf{frequency0}(\mathsf{x}) = \mathsf{frequency0}(\mathsf{x}) + 1 \quad \text{if the move is destructive;}$$

$$\mathsf{frequency1}(\mathsf{x}) = \mathsf{frequency1}(\mathsf{x}) + 1 \quad \text{if the move is constructive.}$$

This transition measure is then normalized to lie in the interval [0,1] by dividing by the maximum of $\mathsf{frequency0}(.)$ or $\mathsf{frequency1}(.)$ as appropriate. This normalized value is then linearly scaled by a selected constant to create a penalty term. The penalty term is added to the corresponding move evaluation so that the frequency factor is taken into account in the move selection procedure.

It should be cautioned, however, that there are additional uses of long term memory in tabu search that we have not undertaken to examine here, including those with the goal of balancing both intensification and diversification simultaneously.

2.5. Probabilistic Move Selection. The move selection approach of probabilistic tabu search, as formulated by Glover (1989), has been successfully applied in solving general 0-1 mixed integer programming problems (see Glover and Løkketangen, 1994). The fundamental idea is simply to translate tabu restrictions and aspirations into penalties and inducements that modify the standard evaluations, and then to map these modified evaluations into probabilities that are strongly biased to favor the highest evaluations. We are particularly motivated to apply this approach in the present setting as a result of observations of Glover and Løkketangen (1994) concerning the uses of probabilities to combat "noise". Specifically, recall that we refine the candidate list and create the move evaluation based on a cost approximation. Thus the move evaluation is contaminated by a form of noise, so that a "best evaluation" does not necessarily correspond to a "best move". Therefore we seek a way to assign probabilities that somehow compensates for the noise level.

Probabilistic move selection also has an evident diversification role. It should be noted that the probabilistic mechanism of tabu search is somewhat different from that of approaches like simulated annealing (SA), not only by incorporating the influence of memory but by accounting for evaluation differences at a more refined level. In contrast to simulated annealing, for example, tabu search does not resort to random sampling (which excludes direct comparison of different evaluations), and does not consider all improving moves as having the same status (which in SA compels any such move encountered to be selected).

Again, following the approach used in the mixed integer programming setting, we apply probabilistic tabu search in the following form (subject to a special modification indicated subsequently).

Step 1 Generate the candidate list and evaluate the moves of this list, assigning penalties to moves that are tabu.

Step 2 Take the move from the candidate list with the highest evaluation.
If the move satisfies the aspiration criterion, accept it and exit;
Otherwise, continue to *Step 3*.

Step 3 Accept the move with probability p and exit;
Or reject the move with probability $1 - \mathsf{p}$, go to *Step 4*.

Step 4 Remove the move from the candidate list.
If the list is now empty, accept the first move of the original candidate list and exit. Otherwise, go to *Step 2*.

In practice, if the candidate list is moderately large, the foregoing procedure can be simplified by considering a reduced number of moves for probabilistic selection. For this, we create a pool to store a certain number of best moves from the candidate list (penalizing tabu moves as before), thus effectively creating a new and smaller candidate list. This simplification is based on the high probability of choosing one of the first d moves, for modest values of p, even if d is relatively small. To illustrate this, we observe the following property:

Property: For the indicated probabilistic move selection, the probability of choosing one of the d best moves in the candidate list is $1 - (1 - p)^{d}$.

Thus if $p = 0.3$, the probability is about 0.832 for picking one of the top five moves, and about 0.972 for picking one of the top ten moves. We selected $p = 0.3$ as a basis for our subsequent experiments.

In addition, as noted previously, this pool can also serve to identify the k best destructive and constructive moves for generating the candidate list for swap moves at the next iteration. The candidate list produced at the preceding iteration is not the same as that produced at the beginning of the swap move. However, for problems with reasonable size, we observe that the difference between these two lists is very insignificant. We adopt this approximated candidate list obtained from the move selection pool at the iteration preceding the swap move (where constructive and destructive moves are evaluated). This reduces the effort of collecting k best partial moves, and consequently allows us to perform the swap moves more frequently during the search. To implement this approach, we simply set the size of the pool equal to k. Thus, at the iteration preceding the swap move, the k best moves are collected separately for destructive and constructive moves, and then preserved for constructing the candidate list used at the next iteration. Finally, these two lists of k best moves are merged into the same-sized pool for probabilistic move selection.

Instead of using the static value of selection probability p in *Step 3*, we introduce a modification to take fuller account of the relative move evaluations. Specifically, we fine-tune the probability of selection based on the ratio of the move evaluation currently examined to the value of the best solution found so far. This selection probability is calculated by $\mathsf{p}^{\alpha\mathsf{r}-\beta}$ where r represents the indicated ratio and α and β are positive parameters. Recall that our aspiration criterion is to accept the move that leads to a solution better than the third best solution (i.e. with a lower cost evaluation) and does not duplicate the first two best solution, thus $\mathsf{r} > 1$ even in some cases where the aspiration criterion is satisfied. With the values of α and β set appropriately, the new probability function provides a fine-tuned probability to discriminate among different evaluations, and favor those proportionately closer to the best solution value. This increases the chance of selecting "good" moves. For example, if α is set to 1.0 and β is set to 0.15, then a move with an evaluation 1.01 times the best solution cost ($\mathsf{r} = 1.01$) has a selection probability of 0.355, which is higher than the base probability 0.3; for a move with $\mathsf{r} = 1.2$, the selection probability is 0.282, which is lower than the base probability 0.3. In particular, the additional fine-tuned mechanism yields probabilities greater than p for $\mathsf{r} \leq (1+\beta)/\alpha$, and probabilities less than p for $\mathsf{r} > (1+\beta)/\alpha$.

2.6. Advanced Restarting and Recovery. Strategies that employ random restarting are often used with simple local search (of the type that always terminates with a local optimum). In this case, restarting gives a chance to find other local optima. In TS, restarting takes a range of forms, according to the degree of emphasis placed on diversification or intensification – that is , according to the degree of driving away from regions already explored, or driving toward particularly good elements.

In an intensification strategy, which is the type we consider here, one of the variants is not to return to an initial "null" state to restart from scratch, but instead to jump directly to an elite solution found previously in the search. This approach involves a form of solution recovery that bears a superficial resemblance to the "best branch" rule of branch and bound. However, a critical difference is that the flexibility of the search structure in our approach makes it possible to locate and return to solutions in a much more fluid manner than permitted in branch and bound. Further, the sequence of recovering such solutions (which in some instances involves recovering multiple solutions in parallel (Glover, 1977)), follows a more flexible pattern, as does the mode of expecting the solutions once recovered.

The use of advanced restart/recovery strategies as an intensification component in Tabu Search has proved effective in a number of applications, including job scheduling, flow shop scheduling and quadratic assignment problems. One of the highly successful variants (Nowicki and Smutnicki (1993, 1994)) records TS memory along with the solutions saved, together with an added tabu restriction to avoid revisiting the same solution visited in earlier search departing from the same solution. Another successful variant (Voss, 1993) requires the solutions recovered to exhibit a certain diversity in relation to each other.

In this application, we employ a variant that postpones the recovery of elite solutions until the last stage of the search. Each recovered solution launchs a search that constitutes a fixed number of iterations before selecting the next solution to recover. The same elite solution list maintained for error correcting, described in section 2.3, serves naturally as a pool of solution for this final stage. Solutions are recovered from this pool in *reverse order*, that is, by stating from the solution with the worst evaluation and working toward the solution with the best evaluation. The list is updated each time a solution is found better than the current worst solution in this elite pool. We merely insert the new solution in its proper location, dropping the worst solution. To enable more elite solutions to be recovered, we thus allow the number of solutions recovered to be larger than the size of the original size of the elite pool. We implement the elite pool for restart/recovery as a circular list, that is, when the best solution (last element) in this pool is recovered, we move back around to the current worst solution (first element) and work toward the best solution again. For each solution recovered, all tabu restrictions are overridden and reinitialized. Thus, while this approach is somewhat simpler than that of Nowicki and Smutnicki (1993, 1994), for example, by not saving the associated tabu memory and its added TS restriction, we anticipate that our use of probabilistic move selection establishes an implicit diversification that compensates for this use of memory. (This type of compensation effect is one of the motivating features of probabilistic TS (Glover, 1989).)

Our simple recovery strategy coupled with probabilistic TS works well for our hardest test problem set, as the subsequent computational results disclose.

3. Computational Results

In this section, we report our computational outcomes for three sets of test problems. The first two sets of problems are generated randomly from distributions whose parameters are selected to create the most difficult problem instances from a computational standpoint. The last set of problems is generated to represent a special case arising in Steiner Tree problems. All test data are available on request from the authors for the purpose of comparability.

3.1. Parameter Settings of Probabilistic TS. An initial solution for our TS approach is produced simply by connecting every target node to its cheapest-link Steiner node, and then constructing a minimum spanning tree on the set of selected Steiner nodes. Since this initial solution does not address the tradeoff between Steiner node costs and target node costs, it is usually a poor quality solution. Our TS approach starts from this solution to search for progressively better solutions.

Tabu tenures for the three types of moves in the TS procedure are randomly generated from an associated (relatively small) interval each time a move is executed. The interval [1,3] is used for constructive moves and the interval [2,5] is used for destructive moves. In the case of swap moves, an interval of [1,3] is used for each of the two elementary moves composing the swap. Most TS applications use intervals that are centered around somewhat larger values. Apparently, the ability to use these small intervals successfully, without cycling, is aided by the fact that the search oscillates between the two different types of elementary moves. The smaller tabu tenures conceivably help the search explore promising regions more thoroughly.

Swap moves are executed either once every seven iterations or in a block of five consecutive iterations when no "new best" solution is found during the most recent 200 iterations. The search process terminates upon reaching $\mathsf{max\_iter}$ iterations, where $\mathsf{max\_iter}$ is set to $\mathsf{min}(20000, \mathsf{max}(3000, \mathsf{n}^2))/2$. The error correction procedure is executed each time a "new best" solution is found, and is applied to the current solution after every three accumulated moves, not counting destructive moves that drop nodes of degree one. Error correction is also applied every 200 iterations to the priority queue that stores the twenty best solutions.

Long term memory is activated after 500 iterations, so that it can be based on relatively reliable frequency information. The penalty term based on long term memory is calculated by multiplying 300 by the normalized frequency for elementary moves, and multiplying 150 by the sum of the two respective normalized frequencies for swap moves. In probabilistic move selection, we choose the probability of acceptance $\mathsf{p} = 0.3$, as previously noted. We additionally use the simplification of shrinking the candidate list for the probabilistic rule to contain the ten best moves (adjusted for tabu penalties), since the probability of selecting a move outside the reduced list would be less than 0.03. The number k, which indicates the number of best destructive and constructive moves to be selected for constructing a candidate list for swap moves, is also set to ten. The two parameters used in the fine-tuned selection probability function, α and β, are set to 1.0 and 0.15 respectively.

For the advanced restart/recovery strategy, the number of solutions recovered ($\mathsf{num\_recover}$) is selected as $\max\{40, 10 * \lceil 0.01 * \mathsf{max\_iter}/30 \rceil\}$. For each recovered solution, a block of 30 iterations is executed. Thus, the recovery strategy begins at iteration $\mathsf{iter\_recover} = \mathsf{max\_iter} - 30 * \mathsf{num\_recover}$ and is executed every 30 iterations thereafter. If $\mathsf{iter\_recover}$ is negative, the restart is never executed since the iteration counter starts at zero. However, since $\mathsf{max\_iter}$ is at least 1,500 in our setting, the number $\mathsf{iter\_recover}$ is at least 300, indicating that recovery is always executed.

To verify and validate the performance of probabilistic tabu search (PTS), a simple version of TS is adapted for comparison. Instead of incorporating the long term memory function and the probabilistic move selection as in PTS, this simple version employs the short term memory described in section 2.4 together with the most commonly-used aspiration criterion (override the TS restriction if a “new best” solution is found),. This approach, which we denote by TSS, also evaluates the whole neighborhood of the swap moves rather than using the candidate list in PTS. All parameters required in TSS, including max_iter, are set to the same as those in PTS, so comparisons between PTS and TSS are meaningful.

3.2. Tests on Randomly Generated Problems. The locations of target nodes and Steiner nodes are randomly generated in Euclidean space with coordinates from the interval [0, 1000]. Euclidean distances are used because they are documented to provide the most difficult instances of classical randomly generated Steiner Tree problems (see Chopra and Rao 1994). The fixed cost of selecting a Steiner node is generated randomly from the interval [10,1000], which provides the most difficult tradeoff with the other parameters selected (see Lee, Chiu and Ryan, 1995).

The first set of test problems is generated as in Lee, Chiu and Ryan (1995) , and is restricted to problems of relatively small dimensions that were capable of being solved by the branch and cut approaches of this study. Problems from the second test set have larger (more realistic) dimensions, and are beyond the ability of current exact methods to solve. The tables that report our results represent the problem dimensions by m and n, which identify the number of target and Steiner nodes respectively. CPU times represent seconds on a Sun Sparc workstation 10 , Model 512.

The first set consists of 23 test problems where m ranges from 50 to 150, and n ranges from 10 to 90. For comparison, we list the solution values and CPU times for the branch and cut algorithm described in Lee, Chiu and Ryan (1995). We also include solution information for a special heuristic (denoted LS) that is described in Lee, Chiu and Ryan (1995) and provides the upper bound for their exact algorithm. This heuristic strategically generates a set of initial solutions and then improves them using local search. The computational results in Lee, Chiu and Ryan (1995) showed that for problems with sizes and parameters distributions represented by our first problem set, this heuristic can find solutions within 0.6% from the optimum while requiring only a fraction of the time required for the exact method. We list this heuristic in the appendix. Finally, Results for two tabu search methods, PTS and TSS, are reported. Computational results for the first set of test problems appear in Table 1. For ease of comparison, we only list the percentages of the cost and CPU times for PTS, LS and TSS in relation to the values of these elements for the exact method.

Problem	Exact Method		PTS/Exact		LS/Exact		TSS/Exact	
$(m \times n)$	Cost	CPU (sec.)	Cost (%)	CPU (%)	Cost (%)	CPU (%)	Cost (%)	CPU (%)
50×10	15010	1	100.00	100.00	100.00	100.00	100.00	100.00
100×10	23603	3	100.00	33.33	100.00	66.67	100.00	33.33
150×10	37526	10	100.00	20.00	100.00	90.00	100.00	10.00
50×20	11562	13	100.00	7.69	100.07	7.69	100.00	7.69
100×20	18815	31	100.00	9.68	100.00	19.35	100.00	12.90
150×20	28240	26	100.00	19.23	100.00	73.08	100.00	26.92
50×30	9965	43	100.00	6.98	100.00	2.33	100.00	6.98
100×30	18702	109	100.00	4.59	100.00	8.26	100.00	6.42
150×30	25026	442	100.00	1.81	100.00	6.79	100.00	2.71
50×40	11083	508	100.00	0.59	100.08	0.20	100.00	0.98
100×40	17738	139	100.00	4.32	100.40	8.63	100.00	7.91
150×40	23781	570	100.00	1.58	100.00	7.19	100.00	3.33
50×50	10549	521	100.00	0.77	100.13	0.38	100.00	1.34
100×50	16703	493	100.00	1.42	100.00	3.25	100.00	3.65
150×50	23428	441	100.00	2.49	100.00	12.47	100.00	6.12
50×60	9692	448	100.00	1.34	100.00	0.45	100.00	2.68
100×60	15988	2216	100.00	0.45	100.23	1.08	100.00	1.17
150×60	21893	1349	100.00	1.19	100.07	6.38	100.00	3.26
50×70	9969	5185	100.00	0.21	100.96	0.06	100.00	0.44
100×70	15456	6277	100.00	0.27	100.04	0.37	100.04	0.73
150×70	22204	20512	100.00	0.12	100.09	0.36	100.00	0.37
20×80	4812	2261	100.00	0.40	103.08	0.04	100.00	0.75
10×90	3435	4759	100.00	0.25	100.00	0.02	100.29	0.36
Average	—	—	100.00	9.51	100.25	18.05	100.01	10.44

Table 1: Computational Results on Small Size Problems

From Table 1, we find that the computation times for the exact method increase exponentially with m and n, and are particularly sensitive to n. Consequently, it is not surprising that the larger instances of the STS problem were not capable of being solved using the current exact method. For the LS, PTS and TSS procedures, the CPU times grow at a far smaller rate. While the value of n correlates with the difficulty of the problem, the value of m chiefly influences the time required for the move evaluation. All three heuristics take only a fraction of the time required by the exact method. Among the three, PTS takes less time than TSS for most cases due to the effect of the candidate list. PTS and TSS take about the same time as LS for most of the test problems, but require more time when n increases significantly (i.e. last two problems in Table 1). For these problems from the small size group, LS finds optimal solutions in thirteen cases and finds solutions relatively close to the optimum (within 3.08%) in the remaining ten cases. TSS finds optimal solutions in 21 cases and near-optimal solutions in two cases. PTS is the best by finding all optimal solutions.

The dimensions for the second set of the test problems are as follows. The value of n for the first fifteen problems ranges from 100 to 200 in increments of 25. For each n, three problems are generated by setting m equal to n, $n + 50$ and $n + 100$ respectively. The last six problems in this set are designed to be particularly hard and have dimensions 250×250, 300×250, 350×250, 100×300, 200×300 and 300×300. Since exact methods are unable to handle problems of this second set, comparisons are performed among the three heuristics. In addition, we examine the results of extending the LS method to include the probabilistic influence of probabilistic tabu search, also without using TS memory, so that the decisions are based solely on the LS choice criteria. We denote this latter method by LS-PTS^0. Note that LS runs much faster than PTS and TSS when n is moderate large, and the gap between speeds becomes significantly enlarged on the problems of large dimensions. This makes any direct comparisons between LS-PTS^0 and its TS counterparts misleading. To avoid this imperfection, we allow LS-PTS^0 to be restarted from scratch $\mathsf{n} + \mathsf{m}$ times, so the computation devoted to LS-PTS^0 approximately matches that of

PTS and TSS for the first few smallest problems in our subsequent test sets. Clearly, LS-PTS^0 is an enhancement of LS and performs significantly better than LS from our preliminary tests, hence increasing the motivation to test it on the harder problems.

We evaluate the PTS heuristic for second set problems by comparing its performance to that of the LS-PTS^0 and TSS. Table 2 lists the costs and CPU times by PTS and the percentage of the other two heuristics.

Problem	PTS		TSS/PTS		LS-PTS^0/PTS	
($m \times n$)	Cost	CPU (sec.)	COST (%)	CPU (%)	Cost (%)	CPU (%)
100×100	16166	52	100.00	286.54	100.00	163.46
150×100	19593	73	100.00	367.12	100.00	350.68
200×100	25102	92	100.00	453.26	100.64	589.13
125×125	16307	127	100.00	406.30	100.00	166.14
175×125	20878	156	100.00	489.10	100.44	307.05
225×125	25706	206	100.06	502.43	100.23	442.23
150×150	19056	232	100.00	435.34	100.00	186.21
200×150	24374	289	100.02	501.04	100.62	302.42
250×150	28248	355	100.00	558.03	100.01	488.17
175×175	20907	321	100.05	547.04	100.24	260.75
225×175	25017	369	100.00	617.62	100.30	433.33
275×175	27672	429	100.00	644.99	100.10	596.04
200×200	24198	421	100.02	527.32	100.25	341.57
250×200	26122	525	100.00	597.14	100.37	513.33
300×200	29879	551	100.04	715.25	100.56	811.07
250×250	25566	659	100.29	645.37	100.63	552.05
300×250	29310	737	100.00	704.07	100.50	811.26
350×250	32664	903	100.03	779.51	100.59	1015.95
100×300	13120	401	100.00	478.55	100.18	148.63
200×300	21238	645	101.36	741.71	100.88	418.29
300×300	28722	1287	100.05	522.92	100.06	584.15
Average	—	420.48	100.09	548.66	100.31	451.52

Table 2: Computational Results on Larger Size Problems

From Table 2, we observe that TSS consistently outperforms LS-PTS^0. Out of the 21 test problems in the second set, TSS obtains better solutions than LS-PTS^0 on seventeen. PTS achieves even better outcomes than TSS, obtaining better solutions in nine cases (and matching TSS's solutions in remaining cases). The average improvement rate of PTS is 0.09% over TSS and 0.31% over LS-PTS^0. Given the fact that LS and TSS can find solutions on average very close to the optimum in Table 1, such an improvement by PTS on the second set problems is notable. The performance indicates that the contribution of memory to the success of the TS procedure is apparently quite significant. It also shows that the additional advanced diversification components in PTS, such as probabilistic move selection, the advanced restart/recovery strategy, etc., provide effective enhancements over the simple TSS heuristic.

The results in Table 2 also show the advantage of PTS over the other two heuristics in terms of the computational times. The times for LS-PTS^0 average 4.52 times longer than those for PTS, although these two heuristics take almost the same time in Table 1. The comparison between CPU times of the two TS approaches is interesting. Though PTS requires additional time to handle the stochastic aspects and the solution recovery mechanism that are not included in TSS, PTS is much superior to TSS, requiring an average only 18.23% of the time required by TSS. This superority is due to the efficiency contributed by the candidate list strategy adopted by PTS. Using this candidate list approach, PTS only needs to evaluate at most k^2 swap moves (generated by combining the

k best destructive moves and the k best constructive moves), while TSS examines the whole neighborhood with n^2 swap moves. The fact that we were able to choose k to be far less than n, without noticeably degrading the quality of the best swap moves presented for selection, resulted in a significant reduction in computational expense. The difference in efficiency grows as the problem dimensions increases. (A few exceptions occur in Table 2, which may be due to the size of the target nodes and the stochastic nature of PTS, but the overall pattern is apparent.)

3.3. Tests on STS on Grid Graph. We have additionally tested our probablistic TS method on a third set of problems which have special structure. Since our previous experiments show that each of PTS, LS-PTS^0 and TSS can easily solve the small size problems, the comparative tests are concentrated on problems with larger dimensions. We have made no effort to exploit the special structure of this additional problem set to improve the performance of our methods, but adhere to our original designs in order to test the potential of our PTS methodology to solve a wide range of telecommunication network design problems without specialization.

It is well known that the Steiner Tree Problem on a grid graph is generally harder than the problems on the Eculidean plane (See e.g. Chopra, Gorres and Rao, 1992). To test the effectiveness of our TS approaches in this case, we generate the third problem set as follows. We randomly select m nodes as target nodes from an $s \times s$ grid graph, hence selecting the remaining $s^2 - m$ grid nodes as Steiner nodes. The distance between either a target node and a Steiner node, or between two Steiner nodes, is defined as follows. For a given problem, randomly select two integers a and b from the inteval [0,100]. If the two nodes are adjacent in a row, then the distance between them is a; if the two nodes are adjacent in a column, then the distance is b; if the two nodes are not adjacent either in a row or in a column, the distance is a large cost equal to the sum of the distances between all adjacent nodes, that is $(a + b)s(s - 1)$. Again, the setup cost for each Steiner node is randomly generated from the interval [10,1000], as in the previous test sets. We generate nineteen test problems in total with s equal to 10, 15, 20, 25, 30. The dimensions of the problems are listed in the first column in Table 3. We also report the computational performance of PTS, TSS and LS-PTS^0 in Table 3.

Problem	PTS		TSS/PTS		LS-PTS^0/PTS	
$(m \times n)$	Cost	CPU (sec.)	COST (%)	CPU (%)	Cost (%)	CPU (%)
50×50	50712	6	110.02	150.00	114.10	100.00
100×125	147920	142	124.4	395.08	158.70	98.21
125×100	52432	68	102.24	433.26	100.89	322.07
100×300	79770	574	106.94	609.52	130.37	87.76
150×250	188483	897	101.22	583.73	114.99	98.16
200×200	318339	972	104.76	498.13	125.16	167.69
250×150	273318	737	101.66	423.91	104.49	185.24
300×100	549320	133	100.88	4102.22	100.88	812.02
125×500	218076	1247	123.48	976.39	135.21	189.90
225×400	397477	2327	114.24	903.53	132.86	252.51
325×300	678786	2868	104.86	668.79	116.47	324.02
425×200	818524	1720	101.10	538.56	104.19	714.70
175×450	297091	2154	113.48	742.37	119.96	301.99
275×350	420354	2903	107.68	707.51	112.69	282.26
375×250	206739	1085	100.29	1149.76	105.31	941.47
475×150	2067345	1153	100.38	392.28	101.05	850.48
400×500	903812	8314	105.08	801.99	122.59	484.39
500×400	653630	5111	110.29	943.56	119.49	923.26
450×450	792166	4193	102.51	1067.45	113.23	833.01
Average	—	1926.53	107.13	846.73	117.51	419.42

Table 3: Results on Problems on Grid Graph

The outcomes in Table 3 validate the general observations for the previous experiments. However, the differences between the methods are more pronounced. First, in nineteen test problems, none of the solutions by TSS and LS-PTS^0 can match the solutions by PTS. Also none of the solutions by LS-PTS^0 are better than any of those by TSS. This additionally strengthens the conclusion that the TS based memory choice is more effective than the memoryless form of PTS embedded in the LS-PTS^0 approach. Apart from this, the gaps between the solution qualities obtained by the various heuristics are consistent. For example, we observe from Table 3 that the average percentage numbers by which the TSS and LS-PTS^0 solution costs exceed those of PTS are 107.13% and 117.51% respectively. The corresponding percentage numbers are 100.09% and 100.31% in Table 2. This shows that the advanced components in PTS work with increasing effectiveness as the problems become harder. This conclusion is further supported by the fact that the problems of the third set required substantially more computational time, indicating that this set is much harder to solve than the other two sets.

An additional observation is relevant. Recall that we deliberately assign every diagonal or bridge link a very large cost in the third problem set that outweighs the other cost factors. Since every target node must connect to an active Steiner node, and all active Steiner nodes must be interconnected as a spanning tree, the expensive diagonal and bridge links typically can not all be excluded from the solutions. However, the best solution will be those with a minimum number of diagonal and bridge links. This property allows us to visually compare the quality of the solutions obtained by the various heuristics. Figures 4, 5, and 6 illustrate the final solutions for problem 50×50 in Table 3 by LS-PTS^0, TSS and PTS respectively, where the dots represent the target nodes, the small circles designate the Steiner nodes. In addition, we use the solid lines to represent the links between active Steiner nodes, and the dash lines for the links between target nodes and active Steiner nodes.

==============================

Please Insert Figure 4 Here

==============================

==============================

Please Insert Figure 5 Here

==============================

==============================

Please Insert Figure 6 Here

==============================

From the three preceding figures, we find that the solutions provided by various heuristics are quite distinctive. The PTS solution uses fourteen diagonal links while the TSS and LS-PTS^0 solutions respectively use seventeen and eighteen diagonal links . All three solutions are locally optimal. Visually examining these three local optima discloses that there exist long paths linking these solutions with each other, indicating that long sequences of moves may be required for moving from each such solution to the others. In this case, intelligently guided diversification plays a very important role in effective search. This can partially explain the outstanding performance of PTS over TSS and LS-PTS^0, since PTS has been designed to marry the diversifying influence of the biased probabilistic choice with the intensifying influence of the advanced restart/recovery strategy.

To investigate the progress of our algorithm, we plot a performance graph for the (50×50) problem in Table 3, for our PTS and TSS heuristics. The performance measure on the vertical axis is the ratio of the current best known solution value to the best value ultimately obtained by any of the methods. This measure is recorded each time an improved solution is found during the search. Since TSS takes longer to perform an iteration than PTS, we have adopted a normalized iteration ratio (current iteration count divided by the maximum iteration count) to represent the progress of the search times. The iteration ratio is shown logarithmically on the horizontal axis. Figure 7 displays the improvements of our algorithm as a function of time.

==============================

Please Insert Figure 7 Here

==============================

Figure 7 clearly demonstrates that the initial solution is poor compared with the best-known cost. Both heuristics improve the solution quality quickly in the early stage of solving the particular illustrated problem. We see that PTS improves the solution effectively throughout the search process, while TSS stalls after reaching around 20% of the total computation time and fails to improve the best solution during the remaining 80% of the execution time. Note in this problem, the solution recovery stage of PTS begins at iteration 300, that is at iteration ratio of 0.2, which corresponds approximately to the iteration ratio at which TSS stalls. Since the PTS search continues to improve the solution, this indicates that our solution recovery strategy in PTS is effective for locating new and better solutions. Given sufficient computational time, PTS is good at finding solutions of extremely high quality.

4. Conclusion

We have developed and tested alternative tabu search implementations for the Steiner Tree Star problem in telecommunication network design. In our approach, the search incorporates constructive and destructive moves as well as exchange moves to explore different neighborhood structures. We introduce evaluation estimates to allow moves to be selected more efficiently, accompanied by an error correction procedure in order to offset the risk of making improper choices. Long term memory, probabilistic move selection and an intensification strategy based on elite solution recovery are also included in the more advanced version.

Numerical tests for two sets of test problems, from distributions designed to make the problem hard to solve, show that for the 23 smaller test problems, tabu search yields optimal solutions in all cases while using only a fraction of the CPU time required by the exact method (which was specialized for this problem class by prior research). For the 21 larger problems. which the exact method is incapable of handling, tabu search consistently outperforms the best local search heuristic available, and likewise outperforms an enhancement of this heuristic designed in this study. Our outcomes demonstrate the effectiveness of long term memory and probabilistic move selection for obtaining better results than relying on short term memory alone.

To better understand the relative performance of the approaches tested, we designed an additional problem set representing the Steiner Tree Star problem on a grid graph that is considered to be harder than the randomly generated problems for other methodologies in the literature. Our experiments show that the probabilistic TS approach works well in this case, and the ranking of the methods remains the same as for the randomly generated problems. In addition, we find that our probabilistic TS approach is particularly effective for problems of this hardest set compared with the other heuristics. Among the several components employed by our TS algorithm, the basic TS short term memory effectively overcomes the limitation of local optimality to achieve optimal or near-optimal solutions for small problem instances. Advanced components such as the probabilistic move selection and solution recovery strategy improve the performance and are responsible for finding extremely high-quality solutions for the large and hard test problems.

Future improvements of our TS approach are anticipated to result by including additional long term memory functions (we examine only one) and by using more refined candidate list strategies. Intensification procedures can take advantage of the fact that some of the Steiner nodes always reside in the active set for good solutions, while other are always inactive. Uses of frequency based memory in an intensification strategy, which afford additional information for probabilistic TS designs, may also provide useful enhancements for solving these types of problems.

Finally, we note that the parameters of our probabilistic TS approach are set arbitarily or by common sense in this paper. Systematically fine-tuning these parameters using statistical tests can further improve the performance of our approach, as shown in our current research (Xu, Chiu and Glover, 1996).

References

CHOPRA, S. AND M. R. RAO, "On the Steiner Tree Problem I & II", *Mathematical Programming*, **64**, (1994) 209-246.

CHOPRA, E. GORRES AND M.R. RAO, "Solving the Steiner Tree Problem on a Graph Using Branch and Cut", *ORSA Journal on Computing*, **4** (1992) 320-336.

DUIN, C. AND S. VOβ, "Steiner Tree Heuristics - A Survey", in: Operations Research Proceedings 1993, Papers of the 22nd Annual Meeting of DGOR in Cooperation with NSOR (Springer-Verlag, 1994) 485-496.

GLOVER, F., "Heuristics for Integer Programming Using Surrogate Constraints", *Decision Sciences*, **8** (1977) 156-166.

GLOVER, F., "Tabu Search - Part I", *ORSA Journal on Computing*, **3** (1989) 190-206.

GLOVER, F. (1995), "Tabu Search Fundamentals and Uses", *Working Paper*, Graduate School of Business, UNiversity of Colorado at Boulder, Boulder, Corado, USA.

GLOVER, F. AND M. LAGUNA, " Tabu Search", in: C. Reeves, (eds.), *Modern Heuristics for Combinatorial Problems* (Blackwell Scientific Publishing, 1993) 71-140.

GLOVER, F. AND A. LØKKETANGEN (1994), "Probabilistic Tabu Search for Zero-One Mixed Integer Programming Problems", *Working Paper*, Graduate School of Business, University of Colorado at Boulder, Boulder, Colorado, USA.

LEE, Y., L. LU, Y. QIU AND F. GLOVER, "Strong Formulations and Cutting Planes for Designing Digital Data Service Networks", *Telecommunication Systems*, **2** (1994) 261-274.

LEE, Y., S. Y. CHIU AND J. RYAN (1995), "Branch and Cut Algorithms for a Steiner Tree-Star Problem", to appear in *ORSA Journal on Computing*.

NOWICKI, E. AND C. SMUTNICKI (1993), "A Fast Taboo Search Algorithm for the Job Shop Problem", *Report 8/93*, Institute of Engineering Cybernetics, Technical University of Wroclaw, Poland.

Nowicki, E. and C. Smutnicki (1994), "A Fast Tabu Search Algorithm for the Flow Shop Problem", Institute of Engineering Cybernetics, Technical University of Wroclaw, Poland.

Voss, S., "Tabu Search: Applications and Prospects", in: D.-Z. DU and P.M. PARDALOS (eds.), *Netork Optimization Problems* (World Scientific, 1993) 333-353.

Xu, J., S. Y. Chiu and F. Glover (1996), Fine-Tuning a Tabu Search Algorithm with Statistical Tests, *Working Paper*, Graduate School of Business, University of Colorado at Boulder, Boulder, Colorado, USA.

Appendix A

In this appendix, we describe the heuristic procedure that has been used to provide an initial upper bound on the optimal solution value in the branch-and-cut algorithm. The following notation and definitions will be used for that purpose. First recall that m is the number of target nodes and n is the number of steiner nodes. A *star* is a subgraph that consists of a single steiner node (the *center* of the star) and a set of target nodes with edges connecting them to the center. The *weight* of a star is equal to the sum of its edge costs and its steiner node cost. The *size* of a star is equal to the number of target nodes contained in that star. Finally, the *steiner number* and *steiner spanning tree* of a solution are defined respectively as the number of steiner nodes being used and the spanning tree connecting these nodes in that particular solution.

Heuristic procedure for minimum steiner tree-star problems:

For star size $k = 2, 3, ..., m$, repeat the following steps:

Step 1. (Generating an initial current solution)

Step 1.1 Label all nodes in $M \cup N$ "unselected" and set $i = 1$. While $i \leq \min\left\{\left\lceil \frac{m}{k} \right\rceil, n\right\}$, determine the minimum-weight star of size k that contains only unselected nodes (the last iteration may find a smaller star), and then label all the nodes in the star "selected"; set $i = i + 1$. Each selected terminal node has been currently assigned to the center of its star.

Step 1.2 Reassign each selected terminal node in M to its closest selected steiner node in N if necessary.

Step 1.3 Assign each unselected terminal node in M to its closest steiner node in N and then label it "selected".

Step 1.4 Connect all selected steiner nodes in N with a minimal spanning tree of these nodes only.

Step 2. (If the steiner number of the current solution is greater than or equal to 2, try to improve the solution as follows:)

Step 2.1 Unselect any selected steiner node in N that is a leaf node in the current steiner spanning tree T and has no terminal nodes assigned to it.

Step 2.2 Unselect any selected steiner node in N that is a leaf node in the current steiner spanning tree T and has only one terminal node assigned to it. Reassign the terminal node to its closest selected steiner node.

Step 2.3 Unselect any selected steiner node in N that is connected to exactly two other selected steiner nodes in the current steiner spanning tree T and has no terminal nodes assigned to it. Connect the two selected steiner nodes directly.

Step 3. For each unselected steiner node in N, repeat the following steps:

Step 3.1 Generate a new temporary solution by adding the unselected steiner node to the current solution as follows: Connect the new steiner node to its closest selected steiner node. Reassign terminal nodes to the newly-added steiner node if it is closer.

Step 3.2 Replace the current solution with the temporary solution if the latter is better.

Step 4. If the steiner number is greater than or equal to 3, reconnect the selected steiner nodes in N with a minimal steiner spanning tree.

Step 5. If any improvement is made to the current solution in Step 2, 3, or 4, go back to Step 2.

If the current solution is better than the best solution found, record the current solution as the new best solution found.

Appendix B

Here we sketch a high-level pseudocode to describe our TS algorithm. We define $iter$ as the iteration counter and introduce $iter\_swap$ such that the search will perform swap moves if and only if $iter\_swap > iter$. When this condition does not hold, the elementary destructive and constructive moves are executed. We let cur_sol and $best\_sol$ denote the current solution and the best solution so far, and cur_cost and $best\_cost$ denote their correponding objective function values. The values n_1, n_2 and n_3 identify the pre-defined iteration counters and max_iter identifies the maximum iteration number. In addition, $best\_iter$ denotes the iteration counter at which the current $best\_sol$ is obtained. The search procedure is defined as follows:

Tabu Search Procedure

```
$iter = 0.$
$iter\_swap = 0.$
Find the initial solution and set it as the current solution.
 While  $(iter < max\_iter)$  do
          if    $(iter \;\mathbf{mod}\; n_1 = 0)$ or $(iter\_swap > iter)$ do
                  evaluate the candidate swap moves, update
                  the elite solution list if necessary, and select
                  the "best" candidate move probabilistically.
          else
                  evaluate the candidate destructive and
                  constructive moves, update the elite solution
                  list if necessary, and select the "best"
                  candidate move probabilistically.
         Perform the selected move.
         Update the TS restrictions and update the
         recency and frequency memories.
         Execute the error-correction if necessary.
          if  $(cur\_cost < best\_cost)$  do
              $best\_sol = cur\_sol;$
              $best\_cost = cur\_cost;$
              $best\_iter = iter.$
          if  $(iter > best\_iter + n_2)$  do
              $iter\_swap = iter + n_3.$
         $iter = iter + 1.$
         Recover a selected elite solution to become
         the new current solution by the repetitive
         circular list recover strategy.
```

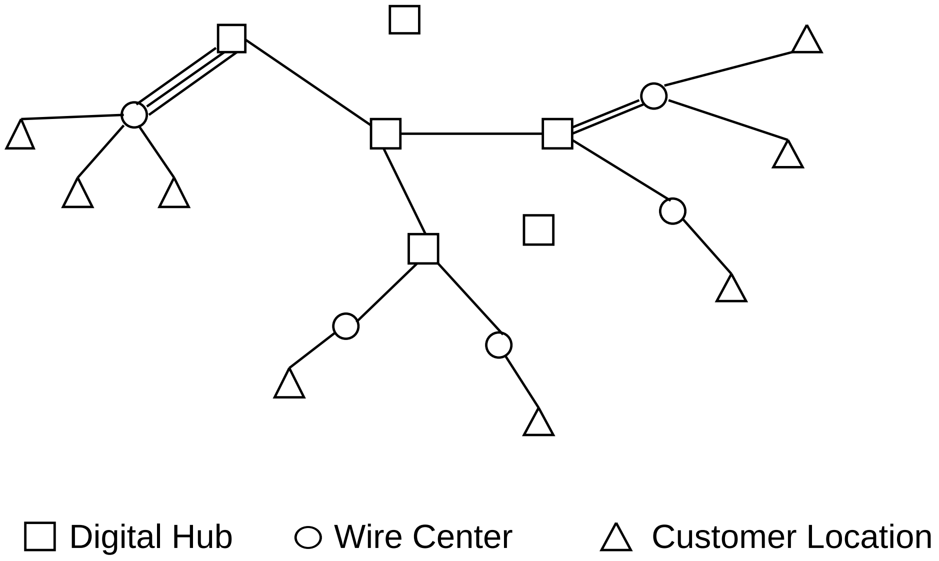

Figure 1: A DDS Network Scenario

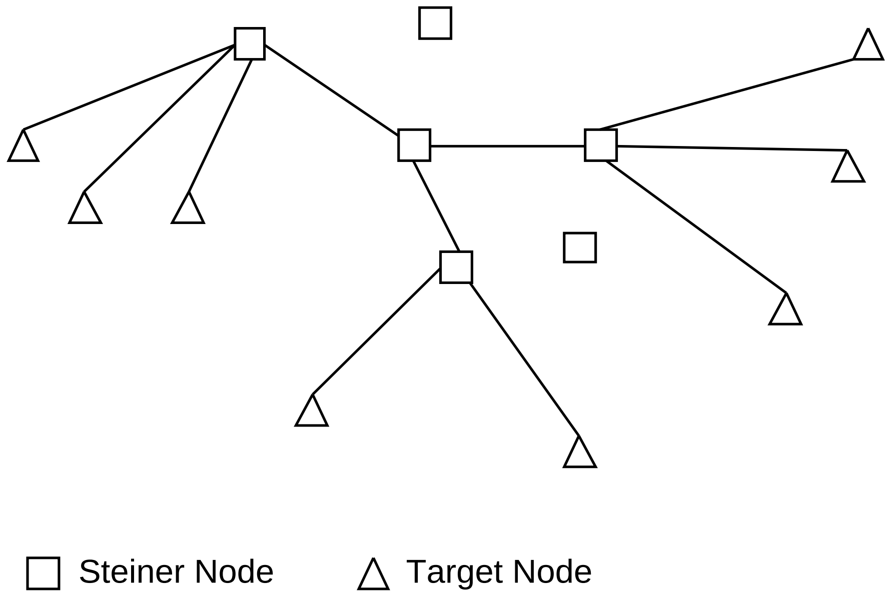

Figure 2: A Solution for a STS Problem

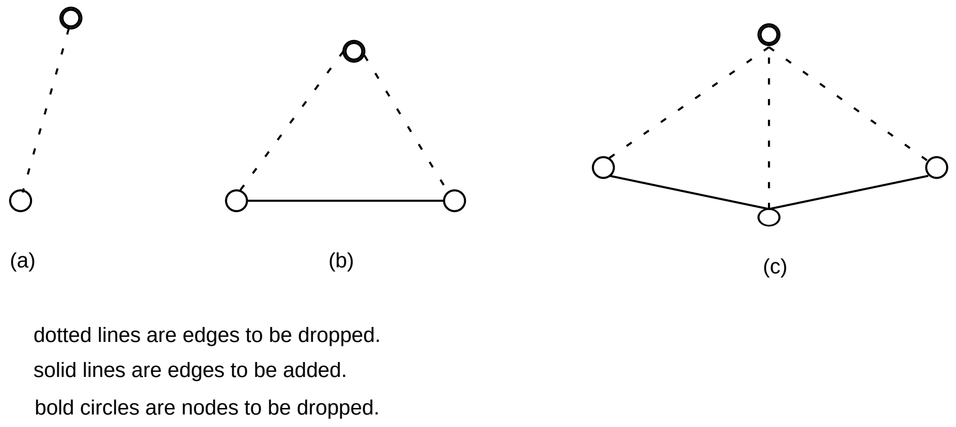
(a)
(b)
(c)
dotted lines are edges to be dropped.
solid lines are edges to be added.
bold circles are nodes to be dropped.

Figure 3: Modifying the Spanning Tree by Destructive Moves

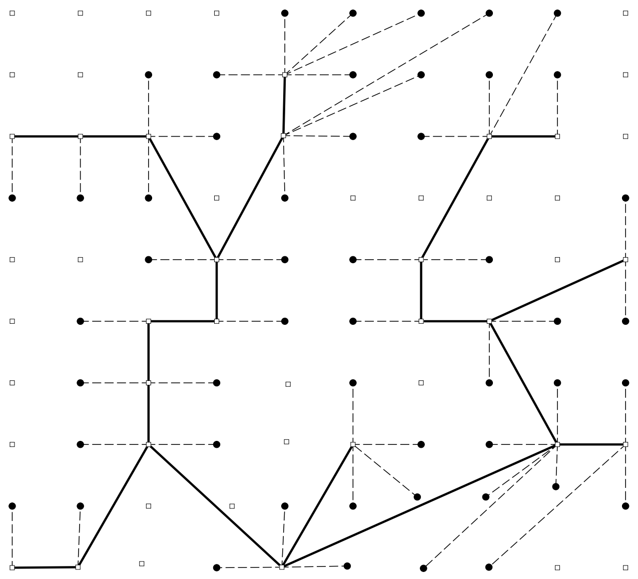

Figure 4: A Solution by LS-PTS^0

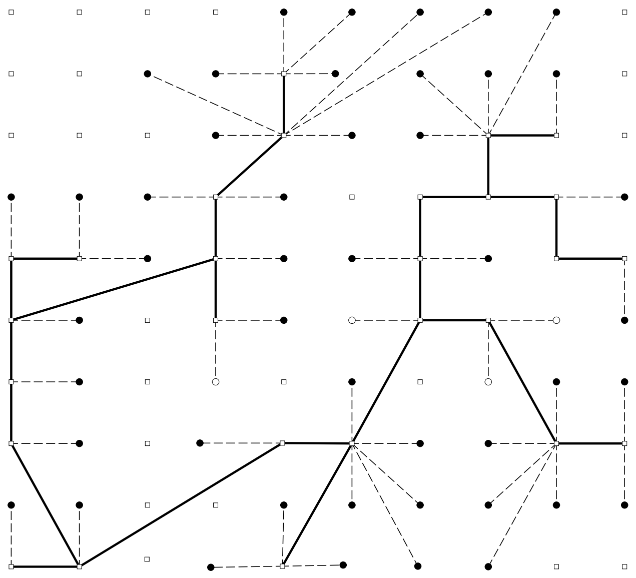

Figure 5: A Solution by TSS

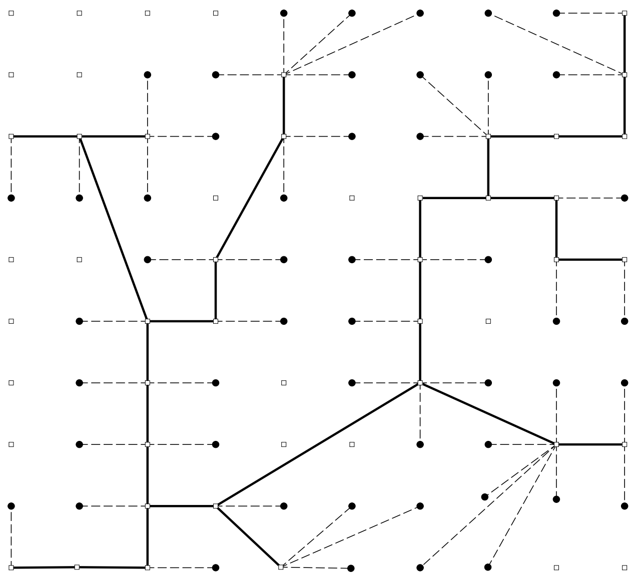

Figure 6: A Solution by PTS

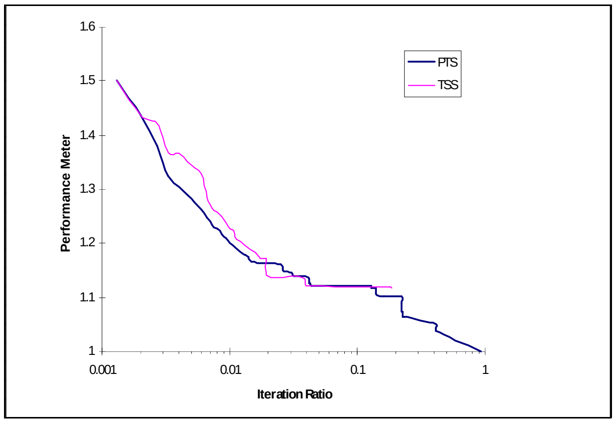

Figure 7: The Performance of the Algorithm over Time