Tabu Search and Ejection Chains—Application to a Node Weighted Version of the Cardinality-Constrained TSP

Buyang Cao • Fred Glover
Department of Transportation/Logistics Service, ESRI, Inc., 380 New York Street, Redlands, California 92373
Graduate School of Business and Administration, Campus Box 419,
University of Colorado at Boulder, Boulder, Colorado 80309-0419

A cardinality-constrained TSP (CC-TSP) problem requires the salesman to visit at least L and at most U cities, represented by nodes of a graph. The objective of this problem is to maximize the sum of weights of nodes visited. In this paper we propose a tabu search method based on ejection chain procedures, which have proved effective for many kinds of combinatorial optimization problems. Computational results on a set of randomly generated test problems with various implementations of the algorithm are reported.
(*Heuristic; Tabu Search; Ejection Chain; Traveling Salesman*)

1. Introduction

A *cardinality-constrained TSP* (*CC-TSP*) problem is one that requires the salesman to visit at least L and at most U cities, represented by nodes of a graph. The objective of this special TSP problem is to determine a node simple path that maximizes the sum of weights of nodes visited, subject to lower and upper bounds on the number of nodes included. This problem arises originally as a subproblem in a column generation method for solving a fractional covering problem with a large number of columns (Cao and von Stengel 1993). A path whose sum of node weights is positive may generate a profitable column, i.e., the corresponding variable can be profitably introduced into the basis, for an LP column generation method (see, e.g., Bradley et al. 1977). The CC-TSP problem is also fundamental for generating telecommunication or supply networks that will be "safe" (able to function) with a limited number of links. A full description of this application appears in the Appendix.

We may formulate the CC-TSP problem as follows. Let $G = (V, E)$ be a graph with node set $V = \{1, \ldots, n\}$ and the edge set E consisting of (unordered) pairs of nodes in V. A constant real weight $w(i)$ is associated with each node $i \in V$. For an abitrary path P in G, define $w(P) = \Sigma\ (w(i)$: node i belongs to $P)$. Finally, let $P(L, U)$ denote the set of node simple paths in G containing at least L and at most U nodes.

Then the problem we seek to solve is given by

CC-TSP

$$\text{subject to} \quad \max w(P)$$
$$P \in P(L, U).$$

The CC-TSP problem has been proved to be NP-hard (see Cao and von Stengel 1993, Fischetti et al. 1994). In this paper we propose a *tabu search* (TS) method based on ejection chain procedures to solve the CC-TSP problem efficiently. Our development is organized as follows. We begin in §2 by briefly describing the ideas of tabu search. Then in §3 we introduce the ejection chain moves that provide the foundation for the form of tabu search explored in this paper. Our specific design for adapting TS to exploit the structure of these moves is given in §4. Computational results on randomly generated test problems are presented in §5. Finally, conclusions are given in §6.

0025-1909/97/4307/0908$05.00

2. Tabu Search in Overview

TS is a *metaheuristic* which guides a local heuristic search procedure to explore the solution space, and has been widely applied to various combinatorial optimization problems (see, e.g., the surveys of Glover 1995, Glover and Laguna 1993). Given a function $f(x)$ to be optimized over a finite set X, TS proceeds iteratively from one point (solution) to another until a chosen termination criterion is satisfied. Each $x \in X$ has an associated *neighborhood* $N(x) \subset X$, and each solution $x' \in N(x)$ is reached from x by an operation called a *move*.

TS operates by strategically modifying $N(x)$, effectively replacing it by another set $N^*(x)$. The composition of $N^*(x)$ is determined by the search history, and results by excluding certain elements of $N(x)$ (classifying them as tabu) and by incorporating other elements not in $N(x)$, but encountered during previous search efforts.

A key aspect of tabu search is the use of special memory structures to determine $N^*(x)$, and hence to organize the way in which the space is explored. A short-term *recency based memory* is used to classify certain *attributes* of solutions recently visited as *tabu-active*. For example, such attributes can refer to variables that change their values, or to elements such as nodes or edges that may be added or dropped upon moving from one solution to another. In turn, solutions that contain tabu-active attributes (or a particular number or type of such attributes), become designated *tabu*, and hence are excluded from the modified neighborhood $N^*(x)$. The effect of classifying attributes as tabu-active in this manner is to render these solutions tabu that have been recently visited, thus preventing moves that lead immediately back to these solutions. In addition, other solutions that share tabu-active attributes with these solutions are also rendered tabu.

The process is managed by introducing one or several *tabu lists*, which record the historical information on the path in X produced by moves during the previous iterations. A solution $x' \in N(x)$ then is classified tabu, and made inaccessible by excluding it from $N^*(x)$, if specified attributes of x' appears on the tabu list(s). Tabu search also provides an additional degree of freedom in accepting $x' \in N(x)$ for inclusion in $N^*(x)$ by using an aspiration concept. In particular, the tabu status of a solution (or a move) can be overruled if certain *aspiration level* conditions are met, i.e., an aspiration level may be viewed as a threshold of attractiveness of the solution considered.

Typically, where $N^*(x)$ may be large or its elements expensive to evaluate, a candidate list strategy is employed so that only a subset of the elements of $N^*(x)$ become candidates to be reached at the next iterations—chiefly consisting of solutions that are likely to be of high quality. The move is then made from x to the best element of $N^*(x)$ that is contained on the candidate list. The meaning of *best* is determined not only by the objective function, but also by heuristic criteria established by the search history. The currently preferred solution is accepted regardless of whether its objective value is better or worse than the solution currently visited, and hence local optimality poses no barrier.

Additional longer term memory structures, based on elements of frequency and interdependence, are also used to guide the search. These structures are primarily designed to achieve goals of intensification and diversification. These goals respectively focus the search more fully in regions judged to be good, and drive the search into new (unexplored) regions. Intensification and diversification strategies often make reference to critical past solutions that periodically are admitted as members of $N(x)$, and thus enlarge the set of solutions available to be visited on the next step. Apart from these special past solutions, $N^*(x)$ is often determined implicitly by spanning solutions of $N(x)$ and checking whether they qualify as tabu or admissible.

Longer term memory, and the special intensification and diversification strategies associated with it are important for creating the most effective forms of tabu search. At the same time, studies that introduce new neighborhood structures sometimes benefit by focusing on the use of short term memory. While lacking the power of full TS implementation, these methods can help to isolate the features of neighborhoods that prove advantageous in particular settings. Such a focus also is at times useful to establish the form of TS strategies that can be exploited as components of a longer term framework.

We will adopt such an approach in the tabu search implementation of this paper, introducing special neighborhood structures for the CC-TSP, and studying these structures by focusing chiefly on short term strategies.

Longer term TS procedure for the CC-TSP that build on these foundations are addressed in a sequel.

3. Moves Using Ejection Chains

Moves based on ejection chain strategies have proved quite effective for a variety of combinatorial problems (see, e.g., Glover 1992, 1995, Laguna et al. 1995, Dorndorf and Pesch 1994, Pesch and Glover 1995, Rego and Roucairol 1995). We propose special types of ejection chain moves for the CC-TSP problem as follows.

Suppose a *current solution* (a path) Q is identified, and the nodes in it are indexed from 1 to q in the sequence they are visited by the path, where $L \le q \le U$ (see Figure 1).

Define:

V_Q = the set of nodes in the solution Q;

E_Q = the set of edges in Q;

$T(i) = \Sigma_{k \ge i, k \in V_Q} w(k)$, hence $T(i) = T(i+1) + w(i)$, $\forall i < q$ and $T(q) = w(q)$, where $i \in V_Q$; and

$N(i)$ = the set of nodes not in the set V_Q and adjacent to node $i \in V_Q$.

In the following we define moves for transforming a given path Q into a new one, as a basis for applying a tabu search procedure to the cardinality-constrained TSP problem. We assume that for each $i \in V_Q$, the nodes (elements) in $N(i)$ are arranged in nonincreasing order of their weights, hence $N(i)$ implicitly identifies a vector as well as a set. Subsequently, in §4, we describe the tabu search rules to modify the collection of moves treated as admissible within the TS framework.

3.1. First Order Ejection/Insert Move

A *first order ejection/insert move* introduces a *single* new node $k \notin V_Q$ into the current solution, allowing a node in V_Q to be ejected, thus creating a new solution (path).

Given two nodes i and j in V_Q, the intersection of $N(i)$ and $N(j)$ is denoted by $\alpha_{ij} = N(i) \cap N(j)$, where α_{ij} is the row vector with dimension $|N(i) \cap N(j)|$. The natural way of generating α_{ij}, following the ordering of $N(i)$ and $N(j)$, causes its nodes to be ordered in the same manner; that is, the first element of α_{ij} is a node with the maximum weight among all nodes adjacent to i and j simultaneously, the second element of α_{ij} has the second largest weight and so on. We will say that two nodes of V_Q are *neighboring* if they are successively indexed in V_Q.

Figure 1 The Initial Solution

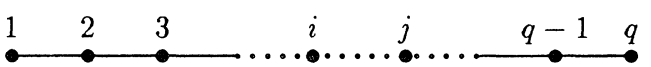
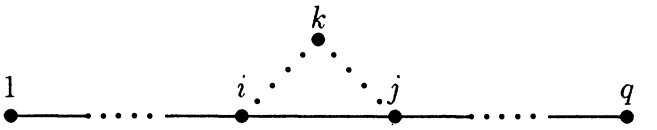


Figure 2 Insertion



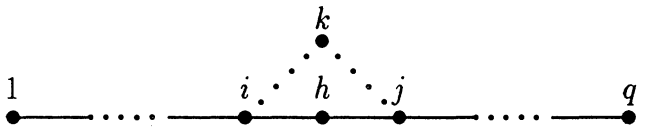
(a) Given two neighboring nodes i and j ($j = i + 1$) in V_Q, some node k may possibly be inserted between them only if $k \in \alpha_{ij}$. For the path Q with q nodes, there are $q - 1$ different pairs of nodes between which other nodes can be inserted. Additionally, a node can be attached before node 1 or after node q. (We allow for the possibility of such an insertion or attachment even if $q = U$). For each pair of neighboring nodes, we consider only a single "best" node as a candidate to be inserted between them. Similarly, only one "best" node will be considered as a candidate to be attached before node 1 (after node q). These $q + 1$ selected candidates constitute the first part, $CL1$, of the candidate list for the possible first order ejection/insert move. The elements of $CL1$ are chosen according to Figure 2.

(i) Select the node with the smallest index in $N(1)$ ($N(q)$), as the candidate for being attached before node 1 (after node q), that is, to become the element of the candidate list $CL1$.

(ii) Given any pair of neighboring nodes in V_Q, i and j ($j = i + 1$), choose the admissible node with the smallest index in α_{ij}, to become an element of $CL1$.

After a node is inserted or attached, the number of nodes in Q will be increased by 1. If the number of nodes in the new solution exceeds the upper bound, we delete a node at either end of Q, excluding the node just added. (We also allow such an end node to be deleted if it has a negative weight and the resulting path contains at least L nodes. However, we may treat this move efficiently by considering a special case where we delete an endpoint *before* considering the other moves.) The change in the value of the objective function (i.e., the sum of node-weights of the path), that results by in-

Figure 3 Ejection and Insertion



serting or attaching the node $k \notin V_Q$ and $k \in CL1$ to Q, is determined by

$$\Delta_k = \begin{cases} w(k) & \text{if no node is dropped,} \\ w(k) - w(p) & \\ & \text{if node } p \text{ is dropped, where } p = 1 \text{ or } q. \end{cases} \quad (1)$$

This quantity provides a basis to decide which node will be finally inserted or attached to the current solution Q.

(b) Consider three successively neighboring nodes, i, h and j, in the current solution Q, i.e., $h = i + 1$ and $j = h + 1$. Although i and j are no longer neighboring, we still may create the vector α_{ij} as before from the intersection of $N(i)$ and $N(j)$. Any node in α_{ij} can be inserted between i and j while the node h will be ejected in order to keep Q a path (see Figure 3). Given any three sequential nodes, say i, h and j in Q, we consider only one node in α_{ij} as the candidate for insertion. The "best" candidate in α_{ij} is the admissible node (element) with the smallest index. Therefore, for the path Q with q nodes, there are $q - 2$ such candidate pairs that constitute the second part, $CL2$, of the candidate list for the possible first order ejection/insert move. The change in the value of the objective function due to the insertion of node k and the ejection of node h in this case is given by

$$\Delta_k = w(k) - w(h). \quad (2)$$

Note, we also may allow the possibility of a move where node k simply replaces node 1 or node q (here either i or j is not defined, and $h = 1$ or q). However, if we use the special case of allowing an endpoint node to be deleted before applying the other steps, then this type of move is implicitly handled by the move described earlier.

At each iteration, a node k from $CL1 \cup CL2$ that possesses a maximum value among the evaluations (1) and (2) is chosen to be added to Q, and a node in V_Q of the current solution may be ejected according to the selection of the node k. We obtain then a new solution (path) Q by this move for the next iteration.

We now enlarge the set of moves to include more advanced uses of chain structures.

3.2. Subpath Ejection Move

In this move, for any node i (except for nodes 1 and q) in V_Q, we select a node $1' \notin V_Q$ which is adjacent to i in G. The node $1'$ is referred to as a *reference node*, which will be used to grow a chain P which will be attached to Q at node i. The process of attaching the new chain P to Q at node i is accompanied by simultaneously ejecting either subpath $(1, \ldots, i-1)$ or $(i+1, \ldots, q)$ of Q. A new solution (path) Q is therefore obtained (see Figure 4).

Each node in $N(i)$, $\forall i \in V_Q$, may be a possible reference node. An effort to examine all nodes in $V_Q \backslash \{1, q\}$ in order to get a "best" candidate chain for switching is obviously time-consuming. We employ the following measure to reduce this effort.

At each iteration, we select a subset $V' \subset V_Q$ by using a *block-random scan* approach. Specifically, we choose $|V'| \ll |V_Q|$ by partitioning V_Q into several disjoint blocks, selecting in each block a node randomly as an element of V'. For each node i in V' the first element in $N(i)$ is taken as the reference node and the chain P is extended based on the following procedure until it has p' nodes.

Chain extension procedure

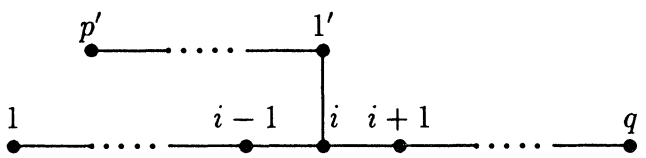
Step 1: count:= 1, P:= {reference_node}.

Step 2: If count = p', stop.

Otherwise, take the first element v from N (reference_node), count:= count + 1, P:= $P \cup \{v\}$, reference_node:= v. Repeat step 2.

Given a node i in the path Q, in the case where $(1, \ldots, i-1)$ is ejected, the value of p' (the number of nodes in the chain P) must satisfy

Figure 4 Subpath Ejection



$$L \leq (q + 1) + p' - i \leq U. \tag{3}$$

It should be noted that here i is also used to indicate the position index of a node occuring in the path. With respect to all elements in V' the chain extension procedure will produce $|V'|$ chains, which constitute the candidate list (denoted by $CL1$) for this kind of subpath ejection move. The evaluation of introducing a chain $(1', \ldots, p')$ to the solution and dropping a subpath before node i in Q is given by

$$w(1') + \cdots + w(p') + T(i). \tag{4}$$

Likewise we can consider the subpath ejection in reverse, i.e., the subpath after i in Q will be dropped. In this case, we generate V' and a corresponding candidate list $CL2$ in the same way described above. The value of p', however, is determined by

$$L \leq p' + i \leq U. \tag{5}$$

The evaluation of a reverse subpath ejection is

$$w(1') + \cdots + w(p') + S(i), \tag{6}$$

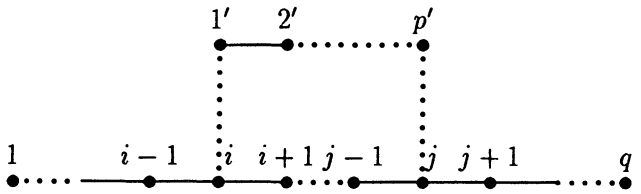
where $S(i) = \Sigma_{k \leq i, k \in V_Q} w(k)$, and $i \in V_Q$.

A chain P (candidate) in $CL1$ or $CL2$ with the maximal value of the evaluation (4) or (6) is chosen as the basis for executing a subpath ejection move.

3.3. Chain Insertion/Ejection Move

In this move a chain composed of nodes not in V_Q will be inserted into Q, and at the same time a subpath of Q will be deleted (see Figure 5). Similar to the subpath ejection move, we select first a subset V' from V_Q such that $|V'| \ll |V_Q|$ by using the same approach described above, then for each $i \in V'$ define the first two elements in $N(i)$ as the reference nodes. From each reference node a chain will be grown by the chain extension procedure given in the last section, and the maximal number of nodes involved in the chain is p'. However, unlike the subpath ejection move, there is no relationship similar to (3) that can determine the value of p' in the chain insertion/ejection move. Therefore, for each chain the value of p' identifying the number of nodes in the chain is determined by randomly generating an integer between 1 and q. Then a chain will be extended from a reference node until the number of nodes reaches p', or stopping sooner if p' nodes cannot be attained.

Figure 5 **Chain Insertion/Ejection**



Suppose that a chain is inserted between two nodes i and j in Q, while the subpath between these two nodes should be deleted. Given any choice of i and chain $(1', \ldots, p')$, there will be a restricted set of candidates for the node j in Q, i.e., the nodes in Q which are adjacent to the node p'. Note that if a chain cannot be grown until the desired number of nodes is reached, the value of p' is modified correspondingly. It is conceivable that at times there is no candidate for j with respect to a certain p', since the number of nodes in a chain is determined randomly and the last node of the chain, i.e., p', is selected without considering its conjunction with the nodes in Q. Furthermore, the value of j (which is also used to represent the position index of a node in Q) should satisfy the following expression:

$$L \leq q + p' - |j - i| + 1 \leq U. \tag{7}$$

Among all js satisfying (7) we choose j such that $|j - i|$ is minimal. [All chains, is and js determined in this way are the members of the candidate list CL for the possible chain insertion/ejection move.]

In CL the chain $(1', \ldots, p')$ is chosen to be inserted between nodes i and j that maximizes the value

$$S(i) + T(j) + w(1') + \cdots + w(p'), \tag{8}$$

and the subpath between i and j in the current Q is dropped. A new solution (path) Q is hence obtained.

3.4. Path Switching/Ejection Move

If a solution (path) is available and the edges connecting nodes in V_Q with nodes in $V \backslash V_Q$ are deleted, then the graph can be partitioned into several connected components. These components can be found by *depth-first-search* in polynomial time. In each component except the component whose nodes are in V_Q, we will find a path greedily, and the number of nodes in the path can be up to U (the fixed upper bound defined previously) if

it is possible. The path is denoted by P, and the set of nodes in P is $V_P = \{1', \ldots, p'\}$, where $1 \leq p' \leq U$. For a path P in some component and the current solution Q, there are three possibilities.

(a) *There is no link between P and Q.* If $p' \geq L$, then the solution Q is replaced by P. In this case, the solution produced for the next iteration creates a potential diversification effect.

(b) *There is only one link between P and Q.* Suppose that edge $(i', i) \in E$ is the link between P and Q, where $i' \in V_P$ and $i \in V_Q$. We have then four possibilities to construct a new solution, i.e., $(1', \ldots, i', i, \ldots, 1)$, $(1', \ldots, i', i, \ldots, q)$, $(p', \ldots, i', i, \ldots, 1)$, $(p', \ldots, i', i, \ldots, q)$. In case the number of nodes in the resulting path exceeds U, the following *adjustment procedure* is employed. Compare the weights of two nodes at the both ends of the new path, delete the node with smaller weight, and repeat this procedure until the path satisfies the upper bound.

(c) *There is more than one link between P and Q.* In this case we generate new paths by the following two rules:

(i) perform the procedure described in (b) for each link;

(ii) given a pair of links, say, (i', i) and (j', j), $i', j' \in V_P$ and $i, j \in V_Q$, where $i' \neq j'$ and $i \neq j$, perform the chain insertion/ejection move of §3.3. This move may produce the following two results:
$(1', \ldots, i', i, \ldots, j, j', \ldots, p')$ (delete the subpath between i' and j'), and $(1, \ldots, i, i', \ldots, j', j, \ldots, p)$ (delete the subpath between i and j).

If the number of nodes in the path obtained by (ii) exceeds U, then the adjustment procedure in (b) is applied to the path, to ensure that feasibility is met.

Suppose there are $t(>1)$ links between P and Q, then $4t + (t-1)t$ potential solutions (paths) may be generated in this step (by applying both (i) and (ii)), and among them one with the best objective function value is chosen as the candidate for the new solution.

If there exists more than one connected component (except for the component that contains the nodes in V_Q), one of the procedures (a), (b), and (c) may be performed for each component. From the candidates thus generated from all components we choose a solution with the best value of the objective function, i.e., the path with the maximal sum of node-weights, as the new solution Q.

The number of candidates provided by (c) can be large, and therefore we subsequently describe strategies for generating a restricted subset of these candidates as a basis for choosing a "best member."

4. Tabu Restrictions and Aspiration Level

A new solution (path) can be obtained by applying any move developed in the previous sections. To determine the admissibility of such moves in the tabu search context, we focus attention on the short term memory components of tabu search.

The *attributes* of moves and solutions that we rely upon to define tabu status consist of edges of paths generated. Among these attributes we identify several to be *critical attributes* of the move, defined as linking edges and breaking edges below. These are the attributes that we allow to be classified tabu-active, and hence to determine the tabu status of a move. In the present case, we use a simple short term memory in which the tabu-active classification is based directly on recency. Here, in particular, a move is classified *tabu* if it reinstates critical attributes that were changed by recent earlier moves (executed within a specified interval of recent iterations). Moves that do not reinstate critical attributes for an appropriate period are anticipated to generate a robust and nonrepetitive search trajectory. The *tabu rules* we propose for implementing this strategy are quite straightforward.

For the first order ejection/insert move the tabu rules are defined as follows: an edge which is introduced to the solution Q, i.e., which is included in the set V_Q at the current iteration, cannot be dropped from V_Q during the next t_1 iterations (the *size* of *tabu list* 1), while the edge removed from the set V_Q at the current iteration is forbidden to be introduced into V_Q again within the next t_2 iterations (the size of tabu list 2). These tabu restrictions can be overruled if the aspiration level defined later is met.

In a (reverse) subpath ejection move we define the edge $(1', i)$ as the *linking edge*, and the edges $(i-1, i)$ and $(i, i+1)$ as the *breaking edges* (see Figure 4). The tabu rule for subpath ejection moves then stipulates: a

linking edge cannot be a breaking edge during the next t_1 iterations and a breaking edge cannot become a linking edge during the next t_2 iterations (unless the aspiration level is reached), where t_1 and t_2 are the sizes of two associated tabu lists.

Tabu restrictions for chain insertion/ejection moves define the edges $(1', i)$ and (p', j) in the move to be *linking edges*, the edges $(i, i + 1)$ and $(j - 1, j)$ to be *breaking edges* (see Figure 5). These restrictions specify that a linking edge of the current iteration cannot be a breaking edge during the next t_1 iterations, while a breaking edge at the current iteration cannot become a linking edge within the next t_2 iterations (again, subject to aspiration level conditions).

We note that a path switching/ejection move is composed of subpath ejection and chain insertion/ejection moves. Therefore, the tabu rules for subpath ejection and chain insertion/ejection moves are applied to determine tabu status for the path switching/ejection move.

Our aspiration criterion in this study consists of the simple aspiration by objective, i.e., the tabu status of a move can be overridden if the value of the objective function obtained by this move is better than the value of the best solution until now.

5. Implementation and Experiments

The algorithm containing the features discussed above has been implemented for solving CC-TSP problems. In this section some implementation details and numerical experiments are presented.

5.1. Implementation Details

Each move described in §3 is performed for a certain number of sequential iterations (i.e., the consecutive iterations without improvement by using that move alone) in the solution procedure. Except for the first order ejection/insert move, after executing each move, the first order ejection/insert move is applied to the resulting solution. The purpose of this is to use the first order move as a "fine" modification to yield better solutions. In order to provide a fair comparison of different methods, the stopping criterion used is the maximum allowed CPU time, i.e., the algorithm terminates if a fixed amount of CPU time has been expended. The best solution at hand is the candidate solution for the corresponding CC-TSP problem.

Our findings indicate that the path switching/ejection move described in §3.4 is the most important part of the algorithm for the CC-TSP problems. The procedure using the path switching/ejection move provides solutions much better than those provided without this move. The path switching/ejection move shows an ability to obtain solutions quite different from those already obtained, that is, this move makes it possible to explore useful areas that cannot be visited by other moves. As discussed in §3.4, if there are $t(>1)$ links between the greedily generated path P and the current solution (path) Q, then there are $4t + (t - 1)t$ candidates for the path switching/ejection move in one connected component. For the path switching/ejection move, these $4t + (t - 1)t$ potential solutions constitute the neighborhood of a solution at a given iteration. However, as the size (number of nodes and edges) of the problem increases, this neighborhood likewise grows. Considerable computational expense can result by evaluating moves from the entire neighborhood. To overcome this difficulty we have implemented the following *candidate list strategies*.

Strategy 1. To provide a baseline, we incorporate a very simple strategy that constructs a candidate list by sampling from the whole neighborhood space at random. From the explanation of §3.4 we know that the new solutions in the path switching/ejection move are produced by the links between P and Q. Among all t links, a number $t'(<t)$ will be sampled randomly to produce $4t' + 2(t' - 1)t'$ possible candidates to provide a new solution (path).

Strategy 2. Our second strategy makes use of relevant information generated during the solution procedure. A link between P and Q is selected as the element for the candidates based on the *greedy selection rule*: for each node i' in the greedily generated path P, we calculate two evaluations $S1(i')$ and $T1(i')$ as we calculate $S(i)$ and $T(i)$ for each node i in Q. Let (i', i) be the link given by $i' \in P$, $i \in Q$ and $(i', i) \in E$. During the path switching/ejection move, the move (i) in case (c) (see §3.4) becomes a candidate if the following ratios inequalities hold for a given η value, where η is chosen between 0 and 1: $S1(i')/S(i) > \eta$ and $T1(i')/T(i) > \eta$.

Let (j', j) be another link, then the move (ii) in the case (c) will be carried out if $(S1(i') + T1(j'))/(S(i) + T(j)) > \eta$ and $(S1(j') - S1(i'))/(S(j) - S(i)) > \eta$, where we assume that $i' < j'$ and $i < j$. The basic idea of this is to eliminate some potential solutions which may not be accepted for the new solution.

Computational experiments demonstrate the superiority of the method with strategy 2 over that with strategy 1. The method with strategy 2 either needs less computing time to get the same results, or obtains better solutions. The test problems were all solved by the method with strategy 2, if the path switching/ejection move is used.

In general, the moves (especially the path switching/ejection move) developed in this paper can produce quite diverse solutions, which are crucial for obtaining high quality solutions. However, according to our preliminary experiments we still need a relatively large number of iterations in order to produce diverse solutions (paths).

Thus, to compensate for this in our short term TS implementation, we incorporate random restarting as a simple approximation to a diversification strategy. Although we reserve the study of more advanced diversification strategies to a sequel, we also introduce a refinement of random restarting in our present study. This refinement of random restarting is embodied in a conditional variant that allows the TS performance to dictate the basis for restarting (as described in the next section). Our computational experiments demonstrate this conditional strategy works quite effectively.

The test problems were randomly generated. The maximal number of edges incident on a vertex in G is bounded by $0.15|V|$. For a given vertex i whose upper bound of incident edges is not reached, a random number $j \neq i$ and $1 \leq j \leq |V|$ is generated. Then edge (i, j) will be generated as long as the upper bound of incident edges for vertex j is not exceeded. Otherwise, a "dummy" edge is generated (i.e., one which does not really occur in the resulting graph).

The test problems are divided into three groups: (1) small-sized problems, $70 \leq |V| \leq 90$ (No. 1–No. 9); (2) medium-sized problems, $100 \leq |V| \leq 120$ (No. 10–No. 18), and (3) moderately large problems, $130 \leq |V| \leq 170$ (No. 19–No. 33). The weights of nodes range over the interval [1, 200]. Preliminary computational experiments on a set of training problems led us to adopt the following parameters for the methods with TS. The lengths of the two tabu lists vary from 9 to 19 and from 7 to 17, respectively. Specifically, $t1$ and $t2$ are, respectively, set to 9 and 7 for small problems (No. 1–No. 9), to 15 and 13 for medium problems (No. 10–No. 18) and to 19 and 17 for moderately large problems (No. 19–No. 33). In addition to these three groups, we also tested an additional set of *large problems*, whose specifications are described subsequently.

5.2. Computational Results

The computational experiments are designed to address the following questions. Compared to the simple ejection chain rule (the first order ejection/insert move) how strong are the advanced ejection chain rules? Does tabu search obtain better solutions than repeated application of the heuristic without incorporating tabu search?

To answer these questions, the following four methods are employed to solve the test problems:

1. A1: Simple moves and no tabu search.
2. A2: Advanced moves and no tabu search.
3. A3: Simple moves and tabu search.
4. A4: Advanced moves and tabu search.

Each CC-TSP problem is solved by the indicated four algorithms with two different fixed CPU times (short and long). The fixed CPU time is determined according to the size of the problem: (1) small size ($70 \leq |V| \leq 90$), short time = 25 seconds, long time = 50 seconds; (2) medium size ($100 \leq |V| \leq 120$), short time = 65 seconds, long time = 130 seconds; (3) moderately large size ($130 \leq |V| \leq 170$), short time = 100 seconds, long time = 200 seconds.

All methods were implemented with random restarting. For the methods with TS, two versions of random restarting were used, an *unconditional restart* method (*k-version*) and a *conditional restart* method (*h-version*). In unconditional method, the total CPU time is divided into k equal intervals, and the procedures are restarted after each interval. In the conditional method, the random restart is invoked only if h iterations occur without improvement.

For the unconditional method, two values of k (small k and larger k were tested for each problem size: (1) 70

$\leq |V| \leq 90$, small $k = 5$, large $k = 10$; (2) $100 \leq |V| \leq 120$, small $k = 10$, large $k = 15$; (3) $130 \leq |V| \leq 170$, small $k = 15$, large $k = 25$. For the conditional method, we tested just three values of h, equal to 15, 20 and 30.

Our algorithms were implemented in PASCAL and the computations were done on Apollo Workstation (UNIX system). The various computational experiments were made with respect to the different size (i.e., the number of nodes and edges) of problems as well as the admitted length of the required path in the problem. The upper bounds U for path lengths are listed in the tables. Since the weights of nodes in our problems are nonnegative, the lengths of all paths obtained equalled U, and the lower bound L was irrelevant. Table 1 and Table 2 present the results obtained by A1–A4 for the short and long CPU times, where A3 and A4 use the conditional restart with $h = 15$. The results obtained by A3 and A4 with the unconditional restart

Table 1 Computational Results (Short CPU Time)

Problem No.	$\|V\|$	$\|E\|$	U	Best Solution A1	A2	A3*	A4*
1	70	290	17	1,363.7	1,377.9	1,363.7	1,377.9
2	70	290	35	2,390.4	2,420.3	2,403.0	2,421.1
3	70	290	51	3,045.7	3,038.6	3,053.6	3,043.5
4	80	398	18	1,542.6	1,544.1	1,548.5	1,548.5
5	80	398	40	2,995.7	3,003.7	3,012.6	3,012.6
6	80	398	63	3,747.1	3,768.4	3,778.8	3,790.1
7	90	494	15	1,382.0	1,382.0	1,382.0	1,382.0
8	90	494	45	3,297.0	3,317.0	3,324.5	3,324.5
9	90	494	68	4,145.3	4,145.3	4,168.4	4,180.6
10	100	606	19	2,616.0	2,616.0	2,616.0	2,610.7
11	100	606	50	5,903.5	5,907.9	5,922.3	5,933.6
12	100	606	73	7,594.6	7,599.2	7,628.0	7,631.6
13	110	692	25	3,365.0	3,351.6	3,367.8	3,364.2
14	110	692	55	6,264.0	6,278.0	6,314.3	6,278.0
15	110	692	81	7,782.9	7,784.2	7,830.4	7,809.7
16	120	838	20	2,722.1	2,726.0	2,721.1	2,721.1
17	120	838	60	6,895.1	6,910.4	6,919.1	6,919.1
18	120	838	86	8,587.8	8,657.7	8,618.2	8,626.2
19	130	846	27	4,867.4	4,885.4	4,885.4	4,865.7
20	130	846	65	10,163.6	10,169.3	10,178.9	10,177.9
21	130	846	92	12,741.8	12,776.4	12,772.5	12,812.9
22	140	993	23	4,325.7	4,325.7	4,325.7	4,332.4
23	140	993	70	11,285.5	11,285.5	11,285.5	11,285.5
24	140	993	100	14,328.6	14,314.3	14,373.5	14,373.5
25	150	1,134	21	3,908.8	3,927.2	3,924.6	3,930.3
26	150	1,134	75	12,032.5	12,106.2	12,094.5	12,128.4
27	150	1,134	109	15,511.8	15,511.8	15,543.7	15,543.7
28	160	1,284	30	5,528.9	5,523.1	5,528.9	5,523.1
29	160	1,284	80	12,783.6	12,757.2	12,783.6	12,783.6
30	160	1,284	120	16,507.1	16,522.4	16,550.8	16,574.0
31	170	1,440	25	4,671.0	4,671.0	4,671.8	4,675.5
32	170	1,440	85	13,746.2	13,746.2	13,766.8	13,805.8
33	170	1,440	130	17,830.4	17,839.0	17,823.9	17,831.5

*Conditional restart, $h = 15$.

Table 2 Computational Results (Long CPU Time)

Problem No.	$\|V\|$	$\|E\|$	U	Best Solution A1	A2	A3*	A4*
1	70	290	17	1,363.7	1,377.9	1,363.7	1,384.2
2	70	290	35	2,410.5	2,420.3	2,420.5	2,421.1
3	70	290	51	3,045.7	3,048.7	3,053.6	3,057.6
4	80	398	18	1,542.6	1,544.1	1,548.5	1,548.5
5	80	398	40	2,995.7	3,003.7	3,012.6	3,012.6
6	80	398	63	3,747.1	3,778.8	3,778.8	3,790.1
7	90	494	15	1,382.0	1,382.0	1,382.0	1,382.0
8	90	494	45	3,297.0	3,324.5	3,324.5	3,324.5
9	90	494	68	4,145.3	4,145.3	4,184.1	4,189.2
10	100	606	19	2,616.4	2,616.4	2,616.0	2,616.4
11	100	606	50	5,903.5	5,922.3	5,922.3	5,933.6
12	100	606	73	7,594.6	7,622.2	7,631.2	7,631.6
13	110	692	25	3,365.0	3,380.6	3,383.5	3,398.8
14	110	692	55	6,314.3	6,315.1	6,314.3	6,415.1
15	110	692	81	7,782.9	7,825.0	7,830.4	7,830.4
16	120	838	20	2,722.1	2,726.0	2,721.1	2,733.6
17	120	838	60	6,919.0	6,910.4	6,919.1	6,919.1
18	120	838	86	8,587.8	8,657.7	8.657.7	8,657.7
19	130	846	27	4,867.4	4,885.4	4,885.4	4,885.4
20	130	846	65	10,178.9	10,178.9	10,178.9	10,180.2
21	130	846	92	12,741.8	12,776.4	12,809.7	12,812.9
22	140	993	23	4,325.7	4,325.7	4,325.7	4,332.4
23	140	993	70	11,285.5	11,285.5	11,285.5	11,285.5
24	140	993	100	14,373.5	14,338.6	14,387.3	14,405.8
25	150	1,134	21	3,924.6	3,927.2	3,929.5	3,933.5
26	150	1,134	75	12,094.5	12,106.2	12,114.9	12,128.4
27	150	1,134	109	15,511.8	15,511.8	15,543.7	15,543.7
28	160	1,284	30	5,528.9	5,523.1	5,528.9	5,528.9
29	160	1,284	80	12,783.6	12,783.6	12,785.7	12,825.5
30	160	1,284	120	16,507.1	16,522.4	16,557.4	16,579.8
31	170	1,440	25	4,672.8	4,687.1	4,671.8	4,695.7
32	170	1,440	85	13,746.2	13,778.8	13,766.8	13,805.8
33	170	1,440	130	17,830.4	17,839.0	17,839.0	17,852.4

*Conditional restart, $h = 15$.

for small and large k values are presented in Table 3. Finally, the results obtained by A3 and A4 for the conditional restart with values of h equal to 20 and 30 are listed in Table 4.

In order to investigate the efficiency of tabu search in solving problems of still greater dimensions, we also tested a set of problems containing from 200 to 700 nodes and from 1,385 to 11,000 edges, with the following specifications.

The maximal number of edges incident on a vertex of the graph is bounded by $0.09|V|$, where $|V|$ is the number of nodes in the graph. Both positive and negative node weights were used. For problems (No. 1–No. 3), the weights range over the interval $[-50, 150]$, and for problems (No. 4–No. 18), the weights range over the interval $[-100, 150]$. These problems were solved by A2 and A4, respectively, and the results are listed in Table 5. L and U are respectively the lower and upper bounds

Table 3 Computational Results (Unconditional Restart)

Problem No.	$\|V\|$	$\|E\|$	U	A3		A4	
				Small k	Large k	Small k	Large k
1	70	290	17	1,363.7	1,363.7	1,385.6	1,385.9
2	70	290	35	2,403.0	2,403.0	2,410.5	2,408.9
3	70	290	51	3,053.6	3,053.6	3,034.3	3,038.6
4	80	398	18	1,544.1	1,544.1	1,544.1	1,544.1
5	80	398	40	3,012.6	3,012.6	3,019.0	3,013.7
6	80	398	63	3,778.8	3,778.8	3,788.5	3,782.5
7	90	494	15	1,382.0	1,382.0	1,391.0	1,381.9
8	90	494	45	3,301.2	3,301.2	3,317.2	3,324.5
9	90	494	68	4,164.0	4,184.1	4,177.6	4,186.3
10	100	606	19	2,616.0	2,616.0	2,616.0	2,617.9
11	100	606	50	5,925.3	5,925.3	5,922.5	5,933.6
12	100	606	73	7,628.0	7,638.0	7,628.0	7,637.7
13	110	692	25	3,367.8	3,367.8	3,386.6	3,386.6
14	110	692	55	6,306.6	6,314.3	6,321.6	6,295.3
15	110	692	81	7,830.4	7,830.4	7,830.4	7,830.4
16	120	838	20	2,708.2	2,721.1	2,722.6	2,721.1
17	120	838	60	6,919.0	6,919.1	6,911.1	6,919.0
18	120	838	86	8,624.9	8,657.7	8,621.1	8,621.1
19	130	846	27	4,864.8	4,864.8	4,864.8	4,885.4
20	130	846	65	10,178.9	10,178.9	10,193.8	10,193.8
21	130	846	92	12,824.8	12,809.7	12,824.8	12,803.2
22	140	993	23	4,325.7	4,325.7	4,325.7	4,325.7
23	140	993	70	11,285.5	11,285.5	11,288.1	11,288.1
24	140	993	100	14,389.7	14,389.7	14,406.8	14,406.8
25	150	1,134	21	3,921.5	3,921.5	3,925.0	3,925.0
26	150	1,134	75	12,114.9	12,114.9	12,115.9	12,124.4
27	150	1,134	109	15,543.7	15,543.7	15,544.3	15,544.3
28	160	1,284	30	5,528.9	5,528.9	5,528.9	5,528.9
29	160	1,284	80	12,797.9	12,797.9	12,804.9	12,804.8
30	160	1,284	120	16,561.9	16,592.9	16,606.2	16,606.2
31	170	1,440	25	4,676.1	4,676.1	4,689.4	4,689.3
32	170	1,440	85	13,780.6	13,771.0	13,791.2	13,791.2
33	170	1,440	130	17,839.0	17,845.6	17,839.0	17,839.0

on the number of nodes in a CC-TSP path. The computational experiments were performed on a Sparc IPX (SunOS), alloting 250 seconds time for solving these problems.

We now comment on the outcomes.

(1) The results confirm the strength of the advanced moves for obtaining solutions with high qualities. The CC-TSP problems could not be solved satisfactorily without the advanced moves. In fact, the simple move (the first order ejection/insert move) is too weak to be used alone for solving the CC-TSP problems effectively. The outcomes in Table 1 and Table 2 show that when more CPU time is allowed to solve the problems, the improvements obtained by the methods with the simple move are not as significant as those obtained by the methods with the advanced moves. (Compare the results of A1 with those of A2, or the results of A3 with those of A4.)

(2) The computational results verify the superiority of the methods with tabu search over the methods without

Table 4 Computational Results (Conditional Restart)

Problem No.	$\|V\|$	$\|E\|$	U	A3		A4	
				$h = 20$	$h = 30$	$h = 20$	$h = 30$
1	70	290	17	1,363.7	1,363.7	1,385.6	1,386.5
2	70	290	35	2,403.0	2,403.0	2,430.3	2,430.3
3	70	290	51	3,053.6	3,053.6	3,053.6	3,057.2
4	80	398	18	1,548.5	1,548.5	1,548.5	1,548.5
5	80	398	40	3,012.6	3,012.6	3,012.6	3,019.0
6	80	398	63	3,778.8	3,778.1	3,788.1	3,790.1
7	90	494	15	1,382.0	1,382.0	1,382.0	1,385.0
8	90	494	45	3,324.5	3,301.2	3,324.5	3,324.5
9	90	494	68	4,167.6	4,164.0	4,181.8	4,192.5
10	100	606	19	2,616.0	2,616.0	2,616.4	2,617.9
11	100	606	50	5,925.3	5,925.3	5,933.6	5,933.6
12	100	606	73	7,638.3	7,628.0	7,623.9	7,631.6
13	110	692	25	3,383.5	3,367.8	3,383.5	3,406.9
14	110	692	55	6,306.6	6,306.6	6,314.8	6,321.6
15	110	692	81	7,830.4	7,830.4	7,830.4	7,830.4
16	120	838	20	2,721.1	2,721.1	2,727.6	2,733.6
17	120	838	60	6,919.0	6,915.6	6,911.1	6,919.1
18	120	838	86	8,623.4	8,623.4	8,657.7	8,657.7
19	130	846	27	4,885.4	4,864.8	4,864.8	4,885.4
20	130	846	65	10,189.9	10,165.8	10,189.9	10,195.9
21	130	846	92	12,797.9	12,818.8	12,824.8	12,824.8
22	140	993	23	4,325.7	4,325.7	4,325.7	4,332.4
23	140	993	70	11,285.5	11,285.5	11,285.5	11,285.5
24	140	993	100	14,373.5	14,373.5	14,389.7	14,406.8
25	150	1,134	21	3,924.6	3,924.6	3,933.5	3,933.5
26	150	1,134	75	12,114.9	12,114.9	12,130.5	12,130.5
27	150	1,134	109	15,543.7	15,543.7	15,543.7	15,543.7
28	160	1,284	30	5,528.9	5,528.9	5,523.1	5,528.9
29	160	1,284	80	12,783.6	12,783.6	12,818.1	12,821.3
30	160	1,284	120	16,563.9	16,574.7	16,570.9	16,592.9
31	170	1,440	25	4,671.8	4,671.8	4,691.0	4,695.7
32	170	1,440	85	13,783.1	13,783.1	13,779.5	13,813.1
33	170	1,440	130	17,824.9	17,823.9	17,842.1	17,852.7

tabu search. This superiority is amplified as increases occur in the size of a CC-TSP, or in the upper bound on the number of nodes in a CC-TSP path, as demonstrated by the outcomes presented in Table 5. (A2 obtains the most satisfactory solutions if tabu search is not used, while A4 provides the best solutions if tabu search is employed.) Considering the results listed in the first two tables, if not enough CPU time is allowed to solve the problems (i.e., only the short CPU time is used), it is difficult to determine a clear winner. However given enough CPU time (using the larger time limit) A4 produces best solutions to all test problems, and yields a very stable performance. Although A2 uses the advanced moves (except the path switching/ejection move) and A3 uses only the simple move, the incorporation of tabu search within A3 causes it to overcome the fact that it uses weak moves, and its performance on average is actually better than that of A2. This demonstrates that even short term TS provides a useful and natural way to get better results out of ejection chain heuristics.

Table 5 Computational Results (Larger Problems)

Problem No.	$\|V\|$	$\|E\|$	L	U	A2	A4
1	200	1,385	20	50	5,764.25	5,822.97
2	200	1,385	20	70	7,280.20	7,309.80
3	200	1,385	30	100	9,008.08	9,035.41
4	300	3,414	40	80	8,542.60	8,651.35
5	300	3,414	60	100	9,659.50	9,676.66
6	300	3,414	70	180	11,200.68	11,313.39
7	400	4,225	80	100	10,707.18	10,775.20
8	400	4,225	100	160	14,335.79	14,358.12
9	400	4,225	150	280	15,866.11	16,065.31
10	500	6,618	100	150	15,033.71	15,250.60
11	500	6,618	200	250	19,238.34	19,490.33
12	500	6,518	250	350	19,575.71*	19,685.53*
13	600	9,175	90	110	12,990.78	13,086.68
14	600	9,175	215	245	22,318.92	22,586.61
15	600	9,175	260	400	24,639.99*	24,731.88*
16	700	11,000	100	120	14,467.02	14,654.82
17	700	11,000	300	350	28,349.66	28,472.72
18	700	11,000	400	500	28,812.03*	29,069.82*

*The upper bound of number of nodes in a path is not reached.

(3) According to the computational experiments, the random restart is generally helpful to obtain diverse solutions, which contribute to better outcomes. However, we can profit from the random restart only if this strategy is applied appropriately. Considering the results of the conditional method (tables 2 and 4), it can be seen on average for A3 that the method with $h = 15$ or $h = 20$ (smaller h value, hence more restarts) provides the best solutions, while for A4 the method with $h = 30$ (the biggest h value, hence fewer restarts) produces the best solutions. This observation is not surprising. Since the simple move is too weak to explore the space effectively, for a given CPU time the creation of additional starting solutions can help the procedure to explore additional regions, which cannot be easily visited by the simple move alone. In A2 a new initial solution is created randomly as soon as the solution procedure is trapped at a local optimum, following the usual random restart design, with the hope that this may help to explore new solution regions. Nevertheless, the diversification provided by this restarting is somewhat less than it might be as a result of relying on randomness, without reference to historical information. The advanced moves within A4 explore the regions efficiently according to at least short term TS information. When the search procedure of advanced moves is interrupted by the random restart after a short time (involving fewer iterations) and the search procedure is exploring a promising region, then the quality of the best solution deteriorates. Therefore, the use of more advanced moves with TS implies that more search time should be allowed before restarting. We can conclude that A4 explores solution space more efficiently than A2 does based on the outcomes in Table 5. One reason is the utilization of historical information (solution trajectories). Our design for using this information is certainly not the most sophisticated possible, which suggests that additional gains may result by more advanced uses of history, an area we intend to explore in the future. The relevance of effective use of such information for solving larger problems, as shown by our outcomes, is particularly noteworthy.

Considering the results obtained by A3 and A4 of the unconditional method, for A3 (random restart with simple move and TS) larger k values (i.e., more intervals) help usually to obtain better solutions. For A4 (random restart with advanced moves and TS), however, it is difficult to determine which k value is more suitable for better solutions. Actually, it is nearly impossible to define a reasonable k for different problems.

The results disclose the conditional restart is clearly superior to the unconditional restart. The reason is similar to those already discussed. In the unconditional method the random restart is invoked after every interval, while in the conditional version the random restart is invoked only after h iterations without improvement. In the latter case, the exploration of a "promising" region is less likely to be interrupted.

6. Conclusions

We have developed a tabu search approach for solving the CC-TSP problems, and propose what we understand to be the first computational study of this problem. Our approach is based on ejection chain strategies, which lead to a variety of special moves for this problem. The computational experiments verify the importance of advanced ejection chain rules for obtaining satisfactory solutions to complicated problems. We also

show that a candidate list based on the relevant information is superior to a simple random sampling strategy. Finally, we show that short term tabu search enhances the power of the ejection chain, and that random restarting should be applied conditionally rather than unconditionally to achieve the best solutions. We remark that we have also used this approach to find a profitable column in a column generation procedure for solving fractional covering problems. These results further confirm the efficiency of the algorithm developed in this paper.

Appendix

Column Generation for Search-Hide on a Graph

Let $G = (V, E)$ be a finite (undirected) graph and k be an integer, $1 \le k \le |V|$. There are two players, called *inspector* and *hider* respectively. The hider selects one of nodes in the graph to hide there permanently. The inspector selects independently a simple path with up to k nodes, called trace, trying to find the hider. The payoff to the inspector is 1 if the node chosen for hiding lies on the trace and 0 otherwise; the payoff to the hider is the negative of that.

The game can be considered as a matrix game, with the hider as row player and the inspector as column player. A matrix entry is the payoff to the inspector, and the entry a_{ij}, $1 \le i \le |V|$ and $1 \le j \le m$ (where m is the number of all possible traces) is given by:

$$a_{ij} = \begin{cases} 1 & \text{if the } i\text{th node is in the } j\text{th trace,} \\ 0 & \text{otherwise.} \end{cases}$$

For the matrix game with payoff matrix A, an optimal mixed strategy of the column player is given by probabilities $p_1, \ldots, p_m$ so that

$$\sum a_{ij}p_j \ge v, \quad i = 1, \ldots, |V|,$$

and so that v is maximal, where v is the value of the game. By defining $x_j = p_j/v$, we can obtain the following LP:

$$\min \sum x_j$$

subject to

$$\sum a_{ij}x_j \ge 1, \quad 1 \le i \le |V|,$$

$$x_j \ge 0, \quad 1 \le j \le m.$$

This LP is known as the fractional covering problem. It is conceivable that as the size (the number of nodes and edges) of a graph increases and k is greater than 2, the number of all possible traces (the value of m) is enormous, i.e., the above LP contains many columns. Since it is impossible or impracticable to generate explicitly all columns in advance, column generation method is a natural alternative to solve this kind of problems.

During the solution procedure, we must identify the column A_j with the least reduced cost c' among all nonbasic columns, that is,

$$c' = \min(c_k - uA_k), \quad 1 \le k \le m,$$

where u is the vector of dual variables. If $c' \ge 0$, then the simplex optimality criterion holds. Otherwise, A_j should enter the basis.

If A_k corresponds a trace, the above formula has the following interpretation: the vector u is a weight vector over all nodes in G, and uA_k is the sum of node weights of this trace under the given u. A column can enter the basis profitably only if the sum of node weights of this trace is greater than c_k. The conclusion is: to find a profitable column, we must find a trace (path) with the maximal sum of node weights, that is, a corresponding CC-TSP must be solved.[1]

[1] This research is partially supported by the Volkswagen Foundation while the first author was at the University of the Federal Armed Forces in Munich, Germany. We appreciate the useful comments of two anonymous referees.

References

Bradley, S. P., A. C. Hax, and T. L. Magnanti, *Applied Mathematical Programming*, Addison-Welsey, Reading, MA, place, 1977.

Cao, B. and B. von Stengel, "Search-Hide Games on Graphs," Technique Report S-9303, University of the Federal Armed Forces Munich, Neubiberg, Germany, 1993.

Dorndorf, U. and E. Pesch, "Fast Clustering Algorithms," *ORSA J. Computing*, 6 (1994), 141–153.

Fischetti, M., H. W. Hamacher, K. Jørnsten, and F. Maffioli, "Weighted k-Cardinality Trees: Complexity and Polyhedral Structure," *Networks*, 24 (1994), 11–21.

Glover, F., "Future Paths for Integer Programming and Links to Artificial Intelligence," *Computer and Oper. Res.*, 13 (1986), 533–549.

——, "Ejection Chains, Reference Structures and Alternating Path Methods for Traveling Salesman Problems," School of Business, University of Colorado at Boulder, Boulder, CO, 1992.

——, "Tabu Search Fundamentals and Uses," School of Business, University of Colorado at Boulder, Boulder, CO, 1995.

—— and M. Laguna, "Tabu search," in C. R. Reeves (Ed.), *Modern Heuristics Technique for Combinatorial Problems*, Blackwell Scientific Publications, Osney Mead, Oxford, England, 1993, 71–140.

Laguna, M., J. P. Kelley, J. L. Gonzalez-Velarde, and F. Glover, "Tabu Search for the Multilevel Generalized Assignment Problem," *European J. Oper. Res.*, 82 (1995), 176–189.

Lawer, E. L., J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, *The Traveling Salesman Problem*, John Wiley and Sons, New York, 1985.

Pesch, E. and F. Glover. "TSP Ejection Chains," to appear in *Discrete Applied Math*.

Rego, C. and C. Roucairol, "Parallel Tabu Search Algorithm Using Ejection Chains for VRP," *Proc. Metaheuristics International Conf.*, Breckenridge, CO, (1995), 253–259.

Accepted by Thomas M. Liebling; received May 1994. *This paper has been with the authors* 13 *months for* 1 *revision.*