Hindawi Publishing Corporation
The Scientific World Journal
Volume 2014, Article ID 353101, 10 pages
http://dx.doi.org/10.1155/2014/353101

Research Article

An Iterated Tabu Search Approach for the Clique Partitioning Problem

Gintaras Palubeckis, Armantas Ostreika, and Arūnas Tomkevičius

Faculty of Informatics, Kaunas University of Technology, Studentu Street 50-408, 51368 Kaunas, Lithuania

Correspondence should be addressed to Gintaras Palubeckis; gintaras@soften.ktu.lt

Received 8 November 2013; Accepted 15 January 2014; Published 4 March 2014

Academic Editors: S. Bureerat, N. Gulpinar, and W. Ogryczak

Given an edge-weighted undirected graph with weights specifying dissimilarities between pairs of objects, represented by the vertices of the graph, the clique partitioning problem (CPP) is to partition the vertex set of the graph into mutually disjoint subsets such that the sum of the edge weights over all cliques induced by the subsets is as small as possible. We develop an iterated tabu search (ITS) algorithm for solving this problem. The proposed algorithm incorporates tabu search, local search, and solution perturbation procedures. We report computational results on CPP instances of size up to 2000 vertices. Performance comparisons of ITS against state-of-the-art methods from the literature demonstrate the competitiveness of our approach.

1. Introduction

Clique partitioning is an important combinatorial optimization problem with many real-life applications. It can be stated as follows. Suppose that there are n objects and, in addition, there is an $n \times n$ symmetric matrix, $D = (d_{ij})$, whose entry d_{ij} represents the dissimilarity between objects i and j. These data can be modeled by considering a complete edge-weighted undirected graph with vertices corresponding to objects and edge weights given by the dissimilarity matrix D. Let this graph be denoted by G and its vertex set by V. The clique partitioning problem (CPP for short) is to partition the vertex set V into mutually disjoint subsets such that the sum of the edge weights over all cliques induced by the subsets is as small as possible. Henceforth, we will denote a feasible solution to the CPP as $P = (V_1, \ldots, V_m)$, where $m \in [1, \ldots, n]$, $\bigcup_{k=1}^{m} V_k = V$, and $V_k \cap V_l = \emptyset$ for each pair $k, l \in \{1, \ldots, m\}$, $k \neq l$. The set of all such solutions is denoted by Π. Mathematically, the clique partitioning problem can be expressed as

$$\min_{P \in \Pi} F(P) = \sum_{k=1}^{m} \sum_{i,j \in V_k, i<j} d_{ij}. \quad (1)$$

The CPP bears some resemblance to the maximally diverse grouping problem (MDGP) [1–3]. There are two main differences between the CPP and the MDGP. First, the latter assumes that the number of groups is fixed a priori. Meanwhile, in the case of CPP, the number of clusters is allowed to vary throughout solution process and is part of the output of algorithms designed for the CPP. Second, in the MDGP, the size of each group is either fixed or bounded from above and possibly from below. No constraints are imposed on the size of clusters in the formulation of the CPP. It follows from the above observations that nontrivial instances of (1) are defined by dissimilarity matrices with both positive and negative entries.

The clique partitioning problem is of interest in several contexts, one of them being combinatorial data analysis. In this context, the objects are characterized by a set of attributes. The values of an attribute are represented by a binary equivalence relation. Then the CPP can be interpreted as the problem of aggregation of binary relations. In this case, the entry d_{ij} of the matrix D represents the number of attributes on which objects i and j disagree minus the number of attributes on which they agree. More details on this approach to data analysis can be found, for example, in

[4–7]. The applicability of the CPP has also been reported in a number of other settings such as assigning flights to airport gates [8, 9], modularity maximization in networks [10], genomics [11], clustering [12, 13], and group formation in cellular manufacturing [14, 15].

Given the practical importance of the problem, many methods, both exact and heuristic, have been proposed in the literature. One of the first exact algorithms for the CPP was developed by Grötschel and Wakabayashi [16]. Their algorithm is based on the cutting plane technique. It uses polyhedral results presented in the companion paper [17]. Several new classes of facet-defining inequalities of the clique partitioning polytope were introduced by Oosten et al. [14]. The usefulness of the inequalities was demonstrated by performing experiments on a set of CPP instances arising in flexible manufacturing. Branch-and-bound algorithms for the CPP were proposed in [18, 19]. Computational results reported in [18, 19] show that these algorithms perform significantly better than the cutting plane method of Grötschel and Wakabayashi. Mehrotra and Trick [20] developed a branch-and-price algorithm for solving the CPP. Actually, their approach is applicable to the formulation that captures both the CPP and the capacitated clustering problem. Sukegawa et al. [21] presented a problem size reduction technique based on the Lagrangian relaxation and the pegging test. Its validity has been verified through extensive numerical experiments. Recently, a branch-and-bound algorithm for the CPP was proposed by Jaehn and Pesch [22]. Their algorithm incorporates several useful features including improved constraint propagation techniques for fixing edges at the nodes of the search tree.

It is well known that the CPP given by (1) is NP-hard in its general form. Thus, CPP instances of larger size can be solved only using heuristic algorithms. Perhaps the most traditional way to approach a combinatorial optimization problem is to resort to local search (LS) techniques. LS algorithms for solving the CPP were developed by Régnier [23] and by Marcotorchino and Michaud [24]. A multistart LS procedure was considered by Guénoche [7]. The drawback of LS techniques is that they might get trapped into poor quality local optima. The other way to approach the problem is to use metaheuristic search methods. Simulated annealing and tabu search implementations for the CPP were proposed by de Amorim et al. [25]. It was found that they performed very favourably in comparison to Régnier's heuristic. Kochenberger et al. [11] presented an approach that relies on the idea of recasting the CPP into the form of a binary quadratic program. The program is solved using a tabu search method incorporating strategic oscillation. Dorndorf and Pesch [18] proposed an ejection chain algorithm for clique partitioning. The algorithm takes advantage of the variable depth local search strategy of Kernighan and Lin [26]. Charon and Hudry [27] offered an adaptation of the noising method to the CPP. They investigated a number of variations of this method, differing in the way of adding noise to the data. Computational experiments were carried out on graphs of order up to 500. Brusco and Köhn [6] developed two versions of the neighborhood search algorithm with different search intensification components. Specifically, the first version uses Régnier's LS procedure, whereas the second one uses the tabu search algorithm. Both versions of the method were shown to be superior to simulated annealing and tabu search implementations from [25].

The focus of this paper is on developing an iterated tabu search (ITS) algorithm for solving the CPP. The primary intention is to combine search intensification and diversification components in order to achieve better performance, compared with that achievable by the best methods in the literature. Our strategy is to test the algorithm on a suite of larger CPP instances than those considered in the previous studies. We report computational experience on problem instances whose size goes up to 2000 vertices. We compare our ITS technique against the heuristics of Brusco and Köhn [6], which are the most successful of the current algorithms for the clique partitioning problem.

It should be noted that various implementations of the iterated tabu search method have also been proposed for other optimization problems. Excellent results have been reported in a number of papers, including [28–34]. The tabu search metaheuristic used in ITS is a general-purpose optimization method, based on which specific algorithms for a variety of optimization problems have been developed. The origins of tabu search go back to the seminal work of Glover [35]. The basic concepts of the modern form of tabu search have been presented by Glover in [36].

The remainder of this paper is arranged as follows. In the next section, we present a detailed description of the ITS algorithm for the CPP. In Section 3, we report the results of computational experiments. Concluding remarks are given in the last section.

2. The Algorithm

In this section, we describe the components of our iterated tabu search algorithm for solving the clique partitioning problem. The essence of the algorithm is simple: improve an initial partition by repeatedly applying tabu search (TS) and solution perturbation procedures. Run duration of the first of them is controlled by imposing a limit on the number of iterations. Upon discovering an improving solution by TS, this solution is submitted to a local search procedure, which explores its neighborhood in an attempt to make further improvements. The implementation details of all these ingredients of the approach are explained next.

2.1. General Scheme. An important aspect of the clique partitioning problem is that the number of clusters is not fixed. Thus, not only the content of the clusters but also their number is varying throughout the search process. In this regard, it is convenient to have an empty cluster added to the partition vector P. We assume in the description of algorithms that m counts all clusters including the empty one. There are two possible scenarios in managing a partition when the number of clusters is changing. An easy case is when a vertex is ordered to move to the empty cluster in P. In this case, a new empty cluster is added to the current partition. More processing is required if the last

vertex of some cluster, say V_l, is relocated. In this case, V_l becomes empty. A possible strategy is to remove V_l from the partition P and renumber the remaining clusters. However, in our approach, renumbering may prove to be unduly time consuming. In particular, this operation implies the need for additional computations when updating tabu information. We have implemented another strategy, which rests on the idea of retaining all emptied clusters. We use a bit-vector of flags $(a_1, \ldots, a_m)$. Its components are set to 0 for all nonempty clusters as well as precisely one selected empty cluster. Let the latter be denoted by V_e. The components of the vector corresponding to other empty clusters are equal to 1. Thus, in the situation outlined above, the cluster V_l is made inactive by setting $a_l := 1$. Obviously, if later in the search a vertex is moved to V_e, then V_l can be selected as a new active empty cluster. In this case, the algorithm sets $a_l := 0$ and $e := l$.

We now present an iterated tabu search algorithm for the clique partitioning problem. The algorithm iteratively invokes two procedures, TS (Tabu Search) and GSP (Get Start Partition), which are detailed in the forthcoming subsections.

Consider the following ITS.

(1) Compute the initial value for the variable m denoting the number of clusters (including the empty one).

(2) Generate a feasible solution $P = (V_1, \ldots, V_m)$ to the problem at random (let V_m in P be an empty cluster). Set $a_k := 0, k = 1, \ldots, m, e := m, m^* := m - 1, P^* := (V_1^*, \ldots, V_{m-1}^*), V_k^* = V_k, k = 1, \ldots, m - 1$, and $F^* := F(P)$.

(3) Apply the tabu search procedure TS (P, m, e, P^*, F^*).

(4) Check if a stopping rule is satisfied. If so, then go to (6). Otherwise go to (5).

(5) Apply the procedure GSP (P, m, e, K, L), where K and L are randomly chosen values for the solution perturbation parameters. Return to (3).

(6) Stop with the partition $P^* = (V_1^*, \ldots, V_{m^*}^*)$. The objective function value on P^* is equal to F^*.

A possible option to initialize the variable m in Step (1) of ITS is to use a fast constructive heuristic for the CPP. Our choice is to employ a randomized variant of the agglomerative heuristic (AH). This heuristic has been described in [19]. It begins with each vertex declared as a separate cluster and, in each step, merges two clusters into a larger one. This process is modelled by a graph, with vertices representing clusters. In order to select two clusters for agglomeration, AH compares the weights of all the edges of this graph. Among them, an edge having minimum weight is chosen, breaking ties arbitrarily. Let this edge be (k, l). If its weight is nonnegative, then the algorithm terminates. Otherwise, it merges the clusters represented by vertices k and l into a single cluster. During this operation, for each cluster V_q with $q \notin \{k, l\}$, the edges (q, k) and (q, l) are replaced by one edge connecting q and vertex, say k, representing the merged cluster. The weight of this edge is set to the sum of the weights of the edges (q, k) and (q, l). The merging step of the heuristic is illustrated in Figure 1. The details of AH can be found in [19]. In the ITS framework, we use a version of AH in which an edge for merging is selected randomly from a set of edges having the smallest weights. The size of this set in our implementation is at most 5. Randomization of AH is useful when ITS is run in a multistart fashion.

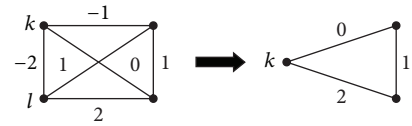

Figure 1: Merging step of the agglomerative heuristic.

In Step (2) of the algorithm, a random initial solution to the problem is generated. This is done by first randomly generating a permutation of vertices. Then, a partition is constructed by assigning either $\lfloor n/(m-1) \rfloor$ or $\lceil n/(m-1) \rceil$ consecutive vertices from the permutation to each of $m - 1$ nonempty clusters. Before entering the search phase, this partition is saved as the best found solution P^*. The number of nonempty clusters in the best solution is denoted by m^*. By running the TS procedure, P^* is replaced with a solution having a smaller objective function value.

The heart of the ITS algorithm is the loop comprising Steps (3) to (5). Inside this loop, the procedures TS and GSP are executed intermittently. The input to each of them includes the current solution specified by the triplet (P, m, e). The parameters K and L passed to GSP are used to control the solution perturbation process. The meaning of these parameters is explained later in this section, where also a description of the GSP procedure is given. In Step (4) of ITS, a stopping criterion is required to be specified. It may be any, for example, upper bound on the number of calls to TS or a stopping rule based on the CPU clock. We performed computational experiments using time limit as the stopping condition.

2.2. Tabu Search. The type of moves used in our implementation of tabu search is relocation of a vertex from its current cluster to a different one. Given a partition $P \in \Pi$, we define the *relocation neighborhood* $N_1(P)$ to be a set of all solutions that can be obtained from P by relocating a single vertex. In the search process, it is important to efficiently compute the differences between the values of the objective function at the solutions in the neighborhood $N_1(P)$ and the value of the objective function at the current solution P. This can be done by taking advantage of an auxiliary $n \times m$ matrix $C = (c_{ik})$, where $c_{ik} = \sum_{j \in V_k} d_{ij}$, $i \in V$, and $k \in \{1, \ldots, m\}$. Consider a vertex $i \in V$ and suppose that its owning cluster in P is V_l. Let $\Delta(P, i, k)$ denote the change in the value of the objective function caused by relocating the vertex i from the cluster V_l to the cluster V_k, $k \neq l$ (see Figure 2). In the literature, the cost variation between two solutions, like $\Delta(P, i, k)$, is called the move gain. We can express $\Delta(P, i, k)$ in terms of the entries of the matrix C

$$\Delta(P, i, k) = c_{ik} - c_{il}. \tag{2}$$

In the description of ITS components given below, we will denote by $\rho(i, P)$ the index of the cluster in P which

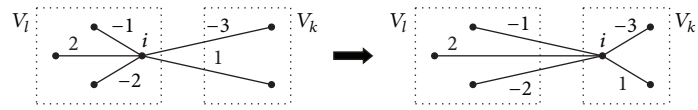

Figure 2: Relocation move.

the vertex i belongs to. Thus, if $i \in V_l \in P$, then $\rho(i, P) = l$. The first component we present is the tabu search procedure. It maintains three data structures to store the tabu status of moves: the matrix $T = (t_{ik})$ with rows and columns corresponding to vertices and clusters, respectively, and lists τ and τ'. The lists are used to represent, in a compact form, moves that are forbidden for a certain number of iterations. If, for example, vertex i is relocated from cluster V_l of size greater than one to a different cluster, then t_{il} is set to 1, the vertex i is appended to the list τ, and the cluster index l is appended to the list τ'. The reason behind the introduction of the lists τ and τ' is to efficiently flip t_{ik} back from 1 to 0, whenever a specified number of iterations have been executed. This number is called *tabu tenure* and is considered as a parameter, denoted as $\bar{t}$, of the TS procedure. Another parameter of TS is the number of iterations $\bar{\alpha}$. The procedure can be stated as follows.

Consider the following TS(P, m, e, P^*, F^*).

(1) Initialize t_{ik} for each vertex $i \in V$ and each cluster V_k, $k \in \{1, \ldots, m\}$, with 0. Set $\alpha := 0$ and $f := F(P)$.

(2) Increase α by 1. Set $\Delta^* := \infty$, $\Delta' := F^* - f$, and $b := 0$.

(3) Iterating through all vertices $i \in V$ and clusters V_k, $k \in \{1, \ldots, m\}$, such that $k \neq \rho(i, P)$ and $a_k = 0$, perform the following steps.

  (3.1) Compute $z := \Delta(P, i, k)$ by (2). If $z < \Delta'$, then proceed to (3.2). Otherwise, check whether one of the following conditions holds: (i) $t_{ik} = 1$; (ii) $b > 0$; (iii) $|V_{\rho(i,P)}| = 1$ and $k = e$. If so, then go to (3.4); if not, go to (3.3).

  (3.2) Increase b by 1 and set $\Delta^* := z$, $v := i$, and $q := k$ with probability $1/b$. Go to (3.4).

  (3.3) If $z < \Delta^*$, then set $\Delta^* := z$, $v := i$, $q := k$, and $\eta := 1$. Otherwise, if $z = \Delta^*$, then increase η by 1 and set $v := i$ and $q := k$ with probability $1/\eta$.

  (3.4) Repeat (3.1)–(3.3) until all pairs i, k have been examined.

(4) Save $\rho(v, P)$ as l and $|V_l|$ as h. For each $i \in V \setminus \{v\}$, subtract d_{iv} from c_{il} and add it to c_{iq}. Move the vertex v from the cluster V_l to the cluster V_q. Increase f by Δ^*. If V_l becomes emptied, then mark this cluster as inactive by setting $a_l := 1$. Otherwise, if $q = e$, then perform the following operations. If there exists a cluster V_r such that $a_r = 1$, then set $a_r := 0$ and $e := r$. Otherwise, all clusters appear to be active, and, in this case, increment m by 1, add an empty cluster V_m to P, initialize the mth column of both T and C with zero vector, and set $e := m$, $a_m := 0$.

(5) If $b > 0$, then proceed to (6). Otherwise, go to (7).

(6) Call the local search procedure LS(P, m, e). Let P also denote the solution returned by it. Form a partition $P^* := (V_1^*, \ldots, V_{\widetilde{m}}^*)$ by identifying nonempty clusters V_r^*, $r = 1, \ldots, \widetilde{m}$, among $V_1, \ldots, V_m$. Set $m^* := \widetilde{m}$, $F^* := F(P^*)$, and $f := F^*$.

(7) If $\alpha = \bar{\alpha}$, then return. Otherwise, perform the following operations. If $\alpha > \bar{t}$, then set $t_{\tau(1)\tau'(1)} := 0$ and remove $\tau(1)$ and $\tau'(1)$ from the lists τ and τ', respectively. If $\alpha \leqslant \bar{t}$, then bypass these modifications of T, τ, and τ'. In both cases, add the vertex v at the end of the list τ. Check whether $h = 1$. If so, then add the empty cluster index e at the end of the list τ' and set $t_{ve} := 1$. If not, then add l at the end of τ' and set $t_{vl} := 1$. Go to (2).

In the above description, α is the iteration counter, f is the value of the current solution, and b stands for the number of solutions found in the current iteration, which are better than the best partition, P^*, recorded so far. The counter b is increased if and only if the move gain Δ is strictly less than the threshold Δ'. In Step (3) of TS, there are two possible cases to consider for a feasible pair consisting of vertex i and cluster V_k. If $\Delta < \Delta'$, then a new improving solution specified by the pair (i, k) is found. It is always accepted if $b = 1$ and accepted with probability $1/b$ if $b > 1$. If, however, $\Delta \geqslant \Delta'$, then conditions (i)–(iii) in Step (3.1) are checked. Condition (iii) is reasonable because it makes no sense to relocate the last vertex of a cluster to the empty cluster. If at least one of the conditions (i)–(iii) is satisfied, then the pair (i, k) is immediately rejected. Otherwise, (i, k) is compared with the best-gain move in Step (3.3). The selected move is represented by the pair (v, q). In Step (4) of TS, $\rho(v, P)$ and $|V_{\rho(v,P)}|$ are saved in order to be used later when updating the tabu data structures at the end of the iteration. The same step also updates both the current solution and the matrix C and, if needed, introduces a new active empty cluster. After these rearrangements, the local search procedure is applied only when an improving solution was found. The resulting partition is saved as the new best solution. While doing this, the partition is shrunk by removing all empty clusters. Step (7) of TS updates the tabu information. If $\alpha > \bar{t}$, then the tabu status of the oldest pair (vertex, cluster) is revoked by appropriately modifying the data structures T, τ, and τ'. For any α, the pair consisting of the selected vertex and, depending on the value of h, either its previous cluster or the empty one is made forbidden for the next $\bar{t}$ iterations.

Our local search procedure for the CPP involves moves of two types. One of them is the same as that used in the TS algorithm. Another type of move is to simultaneously relocate two vertices from their current cluster to a different one.

Given a partition P, two vertices i and j such that $\rho(i,P) = \rho(j,P)$, and a cluster $k \neq \rho(i,P)$, we denote by $P(i,j,k)$ the solution that is derived from P by moving vertices i and j from the cluster $V_{\rho(i,P)}$ to the cluster V_k. Let $N_2(P)$ stand for the set of all solutions that can be obtained in this way. The gain $\delta(P,i,j,k)$ of moving from solution P to solution $P(i,j,k) \in N_2(P)$ can be efficiently calculated as follows:

$$\begin{aligned}\delta(P,i,j,k) &= F(P(i,j,k)) - F(P) \\ &= c_{ik} - c_{il} + c_{jk} - c_{jl} + 2d_{ij},\end{aligned} \tag{3}$$

where $l = \rho(i,P) = \rho(j,P)$. A move of the second type is illustrated in Figure 3. Notice there that relocating a single vertex, i or j, does not lead to an improvement.

At each iteration, the local search (LS) procedure first explores the neighborhood $N_2(P)$ and, if no solution better than P is found, then explores the neighborhood $N_1(P)$. The procedure consists of the following steps.

Consider the following LS(P,m,e).

(1) Randomly generate a permutation of vertices, denoted by $(\pi(1),\dots,\pi(n))$, and a permutation of clusters, denoted by $(\pi'(1),\dots,\pi'(m))$.

(2) Initialize Δ^* with 0.

(3) For $i' = 1,\dots,n-1$ and $j' = i'+1,\dots,n$, do the following.

(3.1) Set $i := \pi(i')$ and $j := \pi(j')$. If either $\rho(j,P) \neq \rho(i,P)$ or $d_{ij} \geqslant 0$, then go to (3.4). Otherwise proceed to (3.2).

(3.2) Iterating through all clusters $k := \pi'(k')$, $k' = 1,\dots,m$, such that $k \neq \rho(i,P)$ and $a_k = 0$, perform the following steps.

(3.2.1) Compute $z := \delta(P,i,j,k)$ by (3).

(3.2.2) Check whether $z < \Delta^*$. If so, then set $\Delta^* := z$ and $q := k$.

(3.3) If $\Delta^* < 0$, then set $v := i$, $u := j$ and go to (7).

(3.4) Repeat (3.1)–(3.3) until all pairs i', j' have been examined.

(4) For $i' = 1,\dots,n$, do the following.

(4.1) Set $i := \pi(i')$.

(4.2) Iterating through all clusters $k := \pi'(k')$, $k' = 1,\dots,m$, such that $k \neq \rho(i,P)$ and $a_k = 0$, perform the following steps.

(4.2.1) Compute $z := \Delta(P,i,k)$ by (2).

(4.2.2) Check whether $z < \Delta^*$. If so, then set $\Delta^* := z$ and $q := k$.

(4.3) If $\Delta^* < 0$, then set $v := i$ and go to (6).

(5) Return (because $\Delta^* = 0$, which means that no improving move is detected).

(6) Update the current solution P and auxiliary data as in Step (4) of TS. Go to (8).

(7) Let $l := \rho(v,P)$. For each $i = 1,\dots,n$ different from v and u, subtract $d_{iv} + d_{iu}$ from c_{il} and add it to c_{iq}. Also, subtract d_{vu} from both c_{vl} and c_{ul} and add d_{vu} to both c_{vq} and c_{uq}. Move the vertices v and u from the cluster V_l to the cluster V_q. If V_l becomes emptied, then set $a_l := 1$. Otherwise, if $q = e$, then perform the following operations. If there exists a cluster V_r such that $a_r = 1$, then set $a_r := 0$ and $e := r$. If no such cluster exists, then increment m by 1, add an empty cluster V_m to P, initialize the mth column of C with zero vector, and set $e := m$, $a_m := 0$.

(8) If m has been increased (either in Step (6) or in Step (7)), then expand π' by setting $\pi'(m) := m$. Go to (2).

Each iteration of the described procedure scans the neighborhood $N_2(P)$ in Step (3) and the neighborhood $N_1(P)$ in Step (4). Both vertices and clusters are considered in the order given by random permutations. Such an approach allows us to introduce some extra randomization in the ITS algorithm. In Step (3) of the LS procedure, only pairs of vertices connected by a negative edge and, of course, belonging to the same cluster are examined. Indeed, if $d_{ij} \geqslant 0$ for some $i, j \in V$, then, as it follows from (3), it is meaningful to ignore simultaneous relocation of the vertices i and j and to evaluate moves involving only one of these vertices. For a pair of vertices passing the test in Step (3.1), the aim is to identify a move with negative value of δ calculated from (3). If two or more such moves exist, then the one with the minimum δ is selected. Throughout this process, the value of the best move is saved as Δ^*. The loop in Step (3.2) evaluates the quality of partitions obtained from P by relocating the vertices i and j to other clusters than their own. Provided that Δ^* is negative, the index of the best cluster is stored in the variable q. If at the end of the loop $\Delta^* < 0$, then the current solution P is replaced by the partition $P(i,j,q) \in N_2(P)$ in Step (7) of LS. If, however, no improving neighbor in $N_2(P)$ is found, then Step (4) is executed. Its structure is very similar to that of Step (3). If, for a vertex i and at least one cluster, the value of Δ is negative, then the move involving i is accepted and the current solution is replaced with a better one in Step (6). Reaching Step (5) means that a locally optimal solution is obtained, and no improving move of two considered types is available. Step (7) of LS is similar to Step (4) of TS. Little difference is seen in formulas used for updating the matrix C.

2.3. Solution Perturbation. Another crucial component of the iterated tabu search algorithm is the solution perturbation procedure GSP. When applied within the ITS framework, it produces starting partitions for tabu search. Such partitions are generated by making a certain number of moves. In contrast to the commonly used method when moves are selected randomly, our procedure favours moves that minimize, to some degree, the degradation of the objective function value of the problem. Still, the procedure incorporates a randomization element. At each step, it selects a move at random from a list of the most attractive candidates. The upper limit on the cardinality of this list, denoted by L, is a parameter of the procedure. Another parameter, K, is the number of vertices that have to be moved from their current clusters to different

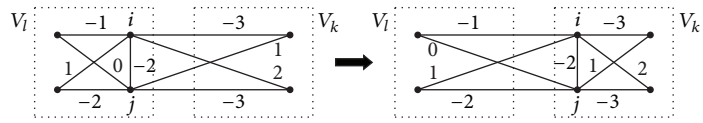

Figure 3: Simultaneously relocating two vertices.

ones. The input to GSP, of course, includes a partition P. In our implementation, ITS submits to GSP the partition that has been returned by the most recent invocation of the TS procedure. While selecting a perturbation move, GSP partially explores the neighborhood $N_2(P)$. Basically, GSP is reminiscent of the above described LS algorithm with Steps (4) and (6) removed. The solution perturbation procedure can be described as follows.

Consider the following GSP(P, m, e, K, L).

(1) Set $U := \emptyset$ and $W := \emptyset$.

(2) Iterating through all vertex pairs $i, j \in V \setminus U$ and clusters V_k such that $e \neq k \neq \rho(i, P) = \rho(j, P)$, $a_k = 0$, and $d_{ij} < 0$, perform the following steps.

(2.1) Compute $z := \delta(P, i, j, k)$ by (3). If $|W| < L$, then go to (2.3). Otherwise, proceed to (2.2).

(2.2) Identify a triplet $w' \in W$ such that $Z(w') \geqslant Z(w)$ for all $w \in W$. If $z < Z(w')$, then remove w' from W and go to (2.3). Otherwise, repeat from (2.1) until all proper combinations of the vertex pair and cluster have been examined.

(2.3) Create a triplet $w'' = (i, j, k)$ with weight $Z(w'') = z$ attached to it. Add w'' to W.

(3) If W is empty, then return with P. Otherwise, select a triplet w, say $w = (v, u, q)$, from W at random. Add the vertices v and u to U.

(4) Update the current solution P and auxiliary data as in Step (7) of LS. If $|U| < K$, then make the set W empty and go to (2). Otherwise, return with the partition P.

As can be seen from the description, a single iteration of GSP comprises Steps (2) to (4). When K is sufficiently less than n, GSP performs $\lceil K/2 \rceil$ iterations. In each of them, the set U of the relocated vertices is enlarged by the addition of a pair of vertices selected in Step (3). In the procedure, W stands for the candidate list of the best moves. A move in W is represented by the triplet consisting of two vertices and index of their target cluster. In Step (2), the neighborhood $N_2(P)$ is searched only partially. Relocating a vertex twice during the run of GSP is prevented by ignoring vertices that belong to the set U. Also, like in LS, the search is restricted only to pairs of vertices that are connected by a negative edge. In addition, the use of an empty target cluster is forbidden. The fitness of legal moves (triplets) is evaluated by (3). If the new triplet is better than the worst triplet in the candidate list W and the size of W is equal to L, then the worst triplet in W is replaced by the new one. In Step (3), the move to be performed is selected from the list W at random. The moved vertices are added to the set U. The solution P is updated precisely in the same way as in Step (7) of LS. The above outlined process is repeated until the cardinality of the set U reaches the prescribed limit K.

The value of the parameter K for each run of GSP is generated using three secondary parameters K_1, K_2, and $K_{\min}$. First, an integer K' from the interval $[\lfloor K_1 n \rfloor, \lceil K_2 n \rceil]$ is chosen uniformly at random. Then, K' is compared with $K_{\min}$. If $K' > K_{\min}$, then K is uniformly and randomly chosen from the integers in the interval $[K_{\min}, K']$. Otherwise, K is set to K'. Thus, in the general case, K is drawn from the interval whose right endpoint is not fixed throughout the execution of the ITS algorithm. The value of the parameter L for GSP is an integer randomly drawn from the interval $[L_1, L_2]$, where L_1 and $L_2 > L_1$ are some constants of the algorithm. The most appropriate values of K_1, K_2, $K_{\min}$, L_1, and L_2 should be selected experimentally.

3. Computational Experiments

The main purpose of experimentation was to show the attractiveness and competitiveness of our approach. In order to evaluate the performance of the developed algorithm, we compared it with two state-of-the-art techniques for the CPP, proposed by Brusco and Köhn [6], namely, the neighborhood search heuristic coupled with the relocation procedure in one case and with the tabu search algorithm in another case. In the study of Brusco and Köhn [6], these algorithms are denoted as NS-R and NS-TS, respectively. When referring to them in this paper, we will use the same names. To be more focused, we do not experiment with other existing methods for the CPP, which are less successful in comparison with both NS-R and NS-TS.

3.1. Experimental Protocol. The described algorithm has been coded in the C programming language. Fortran implementations of NS-R and NS-TS were obtained from Brusco and Köhn [6]. All the tests have been carried out on a PC with an Intel Core 2 Duo CPU running at 3.0 GHz. As a testbed for evaluating the performance of the algorithms, we used a set of CPP instances from the literature as well as two sets of additional instances of our own. The first set consists of 7 benchmark instances originally considered by Charon and Hudry [27] (*rand100-100*, *rand300-100*, *rand500-100*, *rand300-5*, *zahn300*, *sym300-50*, and *regnier300-50*) and 6 instances introduced by Brusco and Köhn [6] (*rand200-100*, *rand400-100*, *rand100-5*, *rand200-5*, *rand400-5*, and *rand500-5*). For descriptions of these problem instances, see [6, 27]. In order to test the algorithms more thoroughly, we generated two additional sets of large random CPP instances. The first of

Table 1: Comparison of ITS with NS-R and NS-TS on the CPP instances considered by Brusco and Köhn.

Instance	Best known	ITS		NS-R	NS-TS
	Value $\widehat{F}$	$F_{\text{best}} - \widehat{F}$ ($F_{\text{aver}} - \widehat{F}$)	Succ.	$F - \widehat{F}$	$F - \widehat{F}$
rand100-5	−1407	0 (0)	10	0	0
rand100-100	−24296	0 (0)	10	0	0
rand200-5	−4079	0 (0)	10	0	0
rand200-100	−74924	0 (0)	10	0	0
rand300-5	−7732	0 (0)	10	3	3
rand300-100	−152709	0 (0)	10	0	0
sym300-50	−17592	0 (0)	10	0	0
regnier300-50	−32164	0 (0)	10	0	0
zahn300	−2504	0 (0)	10	1	0
rand400-5	−12133	0 (0.1)	9	37	13
rand400-100	−222757	0 (42.0)	7	110	208
rand500-5	−17127	0 (5.2)	7	58	41
rand500-100	−309125	118 (224.3)	0	118	735
Average		9.1 (20.9)	8.7	25.2	76.9

them consists of 20 weighted graphs of order 500. The weights of edges are integer numbers drawn uniformly at random from the interval $[-5, 5]$ for the first 10 graphs and from the interval $[-100, 100]$ for the remaining 10 graphs. The second additional set consists of three subsets, each of cardinality 5. The order of the graphs in the first to third subsets is 1000, 1500, and 2000, respectively. The edge weights are random integers uniformly distributed in range $[-100, 100]$.

The TS and GSP procedures described in the previous section have several parameters that affect the performance of the ITS algorithm. Their values were determined by conducting a preliminary experiment. Based on the obtained results, in the main experiments, the TS parameter $\overline{\alpha}$ was fixed at 200. As for the tabu tenure, we have found that a good choice is to take $\overline{t} = \min(10, n/4)$. The parameters used to generate K and L values were set as follows: $K_1 = 0.1$, $K_2 = 0.6$, $K_{\min} = 10$, $L_1 = 10$, and $L_2 = 300$. The only algorithm's parameter whose value is required to be submitted to our ITS code is a maximum CPU time limit per run.

Given its stochastic nature, we executed the ITS algorithm 10 times on each of the test problems. However, unlike ITS, both NS-R and NS-TS were run only once on each instance of size 500 or less. The reason behind such a decision was our intention to use the original executable code from [6], which does not have an option of restarting the algorithms. For graphs of order greater than 500, we used a very slightly modified version of the source (Fortran) code developed in [6]. In fact, we made just a couple of minor adjustments. First, we increased the size of arrays and matrices from 500 and 500×500 to 2000 and 2000×2000, respectively. Second, we included in the possibility to restart the algorithms an arbitrary number of times. We capitalized on this possibility in the experiment on the third dataset. Like in the case of ITS, we ran both NS-R and NS-TS 10 times on each instance in this set. It should be noted, however, that, due to a different Fortran compiler we used, the generated executable code may appear to be less efficient than that provided by Brusco and Köhn [6].

As mentioned above, the input to our ITS code includes the time limit per run. We imposed time limits of 100 seconds, 1000 seconds, 2000 seconds, 1 hour, 2 hours, and 5 hours for test problems with $n \leqslant 200$, $n = 300$, $400 \leqslant n \leqslant 500$, $n = 1000$, $n = 1500$, and $n = 2000$, respectively. The same cutoff times were used for NS-R and NS-TS as well.

3.2. Numerical Results. The results of solving CPP instances in the first dataset are summarized in Table 1. Its first column shows the instance names in which the integer preceding "-" indicates the number of vertices. The second column presents the best known values reported in the literature. For *rand100-100*, *rand300-5*, *rand300-100*, *sym300-50*, *regnier300-50*, *zahn300*, and *rand500-100*, the best solutions were obtained by Charon and Hudry [27]. The authors mention that the experiments on these instances took up to several days of the CPU time. For the remaining instances in Table 1, the best known results were reported by Brusco and Köhn [6]. The third column shows the gap of the value of the best solution out of 10 runs (the gap of the average value of 10 solutions) found by ITS to the value displayed in the second column. The fourth column gives the success rate of reaching the best known value $\widehat{F}$. The last two columns provide the deviation of the value of the solution delivered by NS-R and, respectively, NS-TS from the value $\widehat{F}$. The bottom row shows the results averaged over the whole set of instances.

From Table 1, we see that ITS is superior to both NS algorithms. To be fair, we have to keep in mind that both NS-R and NS-TS were run only once, as opposed to 10 runs in the case of ITS. Therefore, we should compare the results displayed in the last two columns with those shown in parentheses for the ITS algorithm. It can be observed that both variations of NS failed to solve some of the instances

Table 2: Comparison of ITS with NS-R and NS-TS on the CPP instances of size 500.

Instance	Best value	ITS		NS-R	NS-TS
	$\widehat{F}$	$F_{best} - \widehat{F}$ ($F_{aver} - \widehat{F}$)	Succ.	$F - \widehat{F}$	$F - \widehat{F}$
p500-5-1	−17691	0 (10.5)	1	30	0
p500-5-2	−17169	0 (1.5)	6	29	0
p500-5-3	−16815	0 (1.0)	3	92	129
p500-5-4	−16808	0 (3.5)	8	71	70
p500-5-5	−16957	0 (0.0)	10	53	51
p500-5-6	−16615	0 (2.9)	6	134	77
p500-5-7	−16649	0 (8.0)	1	59	71
p500-5-8	−16756	0 (0.9)	7	153	142
p500-5-9	−16629	0 (5.8)	4	17	77
p500-5-10	−17360	0 (0.0)	10	60	62
p500-100-1	−308896	0 (18.9)	4	9	1827
p500-100-2	−310163	0 (171.5)	2	1713	858
p500-100-3	−310477	0 (94.7)	4	2149	718
p500-100-4	−309567	0 (282.9)	1	904	796
p500-100-5	−309135	0 (41.6)	7	2528	0
p500-100-6	−310280	0 (66.7)	7	0	722
p500-100-7	−310063	0 (5.8)	9	1456	2088
p500-100-8	−303148	0 (344.0)	5	2205	1686
p500-100-9	−305305	0 (7.2)	9	1232	1110
p500-100-10	−314864	0 (7.6)	9	76	106
Average		0 (53.7)	5.7	648.5	529.5

of size up to 300. Meanwhile, ITS was able to reach the best known solutions for all of them in each of 10 runs.

In Table 2, we summarize the results of an empirical evaluation of tested algorithms for instances in the second dataset. The structure of this table is the same as that of Table 1. As a basis for comparison, we use, for each instance, the value of the best solution obtained from 10 runs of the ITS algorithm. This value, denoted by $\widehat{F}$, is given in the second column. The results from Table 2 indicate that ITS significantly outperforms both reference algorithms in terms of solution quality. A direct comparison of algorithms across problem instances, using $F_{aver} - \widehat{F}$ as a metric for ITS effectiveness, shows that ITS yields better solutions than NS-R and NS-TS in 18 and, respectively, 17 cases out of 20. We see that NS-TS was able to reach the best results for 3 instances in the dataset. Another NS variation, NS-R, has obtained the best solution value in one case only.

Table 3 reports comparative results of ITS with NS-R and NS-TS for instances of size ranging from 1000 to 2000 vertices. The number of vertices is encoded in the instance name. The first three columns of the table have the same meaning as in Table 2. The fourth column displays the gap of the value of the best solution out of 10 runs (the gap of the average value of 10 solutions) found by NS-R to the best value $\widehat{F}$. The last column provides these statistics for the NS-TS method. The values shown in the second column were obtained by the ITS algorithm. From the results in Table 3, we see that the superiority of the proposed algorithm over NS-R and NS-TS is more pronounced than in the previous experiments. In fact, ITS dominates both NS-R and NS-TS on all instances in the third dataset. Both the NS variations failed to reach the best value in all cases. By analyzing the results in Tables 2 and 3, we also find that NS-R and NS-TS perform comparably in terms of solution quality, with NS-TS having a slight edge. A similar conclusion has been reached in the study of Brusco and Köhn [6].

4. Conclusions

In this paper we have presented an iterated tabu search algorithm for the clique partitioning problem. The described method incorporates tabu search, local search, and solution perturbation procedures. The latter is an essential component of the approach, because, according to our experience, to be successful, a tabu search-based algorithm for graph partitioning type problems should use a sufficiently powerful search diversification mechanism.

Experimental evaluations on three sets of CPP instances of size up to 2000 show that the proposed algorithm is able to produce solutions of high quality. In particular, we can conclude that our algorithm exhibits superior performance compared to the method of Brusco and Köhn. However, we surmise that, for the largest instances in the test suite, the best solutions obtained probably are not the best possible. We have experienced that in order to find improved solutions using ITS a big amount of CPU time is needed. The development

Table 3: Comparison of ITS with NS-R and NS-TS on larger CPP instances.

Instance	Best value $\widehat{F}$	ITS $F_{\text{best}} - \widehat{F}$ ($F_{\text{aver}} - \widehat{F}$)	NS-R $F_{\text{best}} - \widehat{F}$ ($F_{\text{aver}} - \widehat{F}$)	NS-TS $F_{\text{best}} - \widehat{F}$ ($F_{\text{aver}} - \widehat{F}$)
p1000-1	−883359	0 (2190.5)	7099 (11056.4)	6467 (12371.8)
p1000-2	−879792	0 (1507.5)	7851 (12533.1)	7262 (11585.2)
p1000-3	−862969	0 (1690.8)	6518 (9638.3)	7516 (9905.1)
p1000-4	−865754	0 (1167.4)	8388 (12325.2)	7346 (10487.2)
p1000-5	−887314	0 (2224.1)	6990 (11777.1)	4777 (10263.6)
p1500-1	−1614791	0 (6883.5)	22179 (26820.6)	17669 (23080.2)
p1500-2	−1642442	0 (5174.1)	21126 (26973.8)	20750 (27272.0)
p1500-3	−1600857	0 (2457.3)	12599 (22708.8)	12005 (19924.9)
p1500-4	−1633081	0 (3884.2)	19236 (25841.2)	12248 (21993.8)
p1500-5	−1585484	0 (3005.8)	16894 (24918.8)	12430 (20878.4)
p2000-1	−2489880	0 (4229.5)	32501 (40603.7)	25242 (40091.7)
p2000-2	−2479127	0 (4504.4)	34047 (41348.2)	21122 (37640.4)
p2000-3	−2527119	0 (4480.4)	29320 (37828.0)	18350 (33717.8)
p2000-4	−2511914	0 (4461.9)	28342 (38291.0)	32269 (42376.9)
p2000-5	−2499690	0 (7846.1)	32928 (42543.9)	30561 (37686.7)
Average		0 (3713.8)	19067.9 (25680.5)	15734.3 (23951.7)

of fast and powerful heuristic algorithms for the CPP is an important line of further research.

Conflict of Interests

The authors declare that they have no conflict of interests regarding the publication of this paper.

References

[1] Z. P. Fan, Y. Chen, J. Ma, and S. Zeng, "A hybrid genetic algorithmic approach to the maximallydiverse grouping problem," *Journal of the Operational Research Society*, vol. 62, no. 7, pp. 1423–1430, 2011.

[2] M. Gallego, M. Laguna, R. Martí, and A. Duarte, "Tabu search with strategic oscillation for the maximally diverse grouping problem," *Journal of the Operational Research Society*, vol. 64, no. 5, pp. 724–734, 2013.

[3] G. Palubeckis, E. Karčiauskas, and A. Riškus, "Comparative performance of three metaheuristic approaches for the maximally diverse grouping problem," *Information Technology and Control*, vol. 40, no. 4, pp. 277–285, 2011.

[4] J.-P. Barthélemy and B. Leclerc, "The median procedure for partitions," in *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, I. J. Cox, P. Hansen, and B. Julesz, Eds., vol. 19, pp. 3–34, 1995.

[5] J.-P. Barthélemy and B. Monjardet, "The median procedure in cluster analysis and social choice theory," *Mathematical Social Sciences*, vol. 1, no. 3, pp. 235–267, 1981.

[6] M. J. Brusco and H.-F. Köhn, "Clustering qualitative data based on binary equivalence relations: neighborhood search heuristics for the clique partitioning problem," *Psychometrika*, vol. 74, no. 4, pp. 685–703, 2009.

[7] A. Guénoche, "Consensus of partitions: a constructive approach," *Advances in Data Analysis and Classification*, vol. 5, no. 3, pp. 215–229, 2011.

[8] U. Dorndorf, F. Jaehn, and E. Pesch, "Modelling robust flight-gate scheduling as a clique partitioning problem," *Transportation Science*, vol. 42, no. 3, pp. 292–301, 2008.

[9] U. Dorndorf, F. Jaehn, and E. Pesch, "Flight gate scheduling with respect to a reference schedule," *Annals of Operations Research*, vol. 194, no. 1, pp. 177–187, 2012.

[10] D. Aloise, S. Cafieri, G. Caporossi, P. Hansen, S. Perron, and L. Liberti, "Column generation algorithms for exact modularity maximization in networks," *Physical Review E*, vol. 82, no. 4, Article ID 046112, 9 pages, 2010.

[11] G. Kochenberger, F. Glover, B. Alidaee, and H. Wang, "Clustering of microarray data via clique partitioning," *Journal of Combinatorial Optimization*, vol. 10, no. 1, pp. 77–92, 2005.

[12] P. Hansen and B. Jaumard, "Cluster analysis and mathematical programming," *Mathematical Programming*, vol. 79, no. 1–3, pp. 191–215, 1997.

[13] G. Kettleborough and V. J. Rayward-Smith, "Optimising sum-of-squares measures for clustering multisets defined over a metric space," *Discrete Applied Mathematics*, vol. 161, no. 16-17, pp. 2499–2513, 2013.

[14] M. Oosten, J. H. G. C. Rutten, and F. C. R. Spieksma, "The clique partitioning problem: facets and patching facets," *Networks*, vol. 38, no. 4, pp. 209–226, 2001.

[15] H. Wang, B. Alidaee, F. Glover, and G. Kochenberger, "Solving group technology problems via clique partitioning," *International Journal of Flexible Manufacturing Systems*, vol. 18, no. 2, pp. 77–97, 2006.

[16] M. Grötschel and Y. Wakabayashi, "A cutting plane algorithm for a clustering problem," *Mathematical Programming*, vol. 45, no. 1–3, pp. 59–96, 1989.

[17] M. Grötschel and Y. Wakabayashi, "Facets of the clique partitioning polytope," *Mathematical Programming*, vol. 47, no. 1–3, pp. 367–387, 1990.

[18] U. Dorndorf and E. Pesch, "Fast clustering algorithms," *ORSA Journal on Computing*, vol. 6, no. 2, pp. 141–153, 1994.

[19] G. Palubeckis, "A branch-and-bound approach using polyhedral results for a clustering problem," *INFORMS Journal on Computing*, vol. 9, no. 1, pp. 30–42, 1997.

[20] A. Mehrotra and M. A. Trick, "Cliques and clustering: a combinatorial approach," *Operations Research Letters*, vol. 22, no. 1, pp. 1–12, 1998.

[21] N. Sukegawa, Y. Yamamoto, and L. Zhang, "Lagrangian relaxation and pegging test for the clique partitioning problem," *Advances in Data Analysis and Classification*, vol. 7, no. 4, pp. 363–391, 2013.

[22] F. Jaehn and E. Pesch, "New bounds and constraint propagation techniques for the clique partitioning problem," *Discrete Applied Mathematics*, vol. 161, no. 13-14, pp. 2025–2037, 2013.

[23] S. Régnier, "Sur quelques aspects mathématiques des problèmes de classification automatique," *I.C.C. Bulletin*, vol. 4, pp. 175–191, 1965.

[24] J. F. Marcotorchino and P. Michaud, "Heuristic approach to the similarity aggregation problem," *Methods of Operations Research*, vol. 43, pp. 395–404, 1981.

[25] S. G. de Amorim, J.-P. Barthélemy, and C. C. Ribeiro, "Clustering and clique partitioning: simulated annealing and tabu search approaches," *Journal of Classification*, vol. 9, no. 1, pp. 17–41, 1992.

[26] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell System Technical Journal*, vol. 49, no. 2, pp. 291–307, 1970.

[27] I. Charon and O. Hudry, "Noising methods for a clique partitioning problem," *Discrete Applied Mathematics*, vol. 154, no. 5, pp. 754–769, 2006.

[28] N. Bilal, P. Galinier, and F. Guibault, "An iterated-tabu-search heuristic for a variant of the partial set covering problem," *Journal of Heuristics*, 2014.

[29] J.-F. Cordeau, G. Laporte, and F. Pasin, "Iterated tabu search for the car sequencing problem," *European Journal of Operational Research*, vol. 191, no. 3, pp. 945–956, 2008.

[30] Z. Fu, W. Huang, and Z. Lü, "Iterated tabu search for the circular open dimension problem," *European Journal of Operational Research*, vol. 225, no. 2, pp. 236–243, 2013.

[31] X. Lai and Z. Lü, "Multistart iterated tabu search for bandwidth coloring problem," *Computers and Operations Research*, vol. 40, no. 5, pp. 1401–1409, 2013.

[32] A. Misevicius, "An implementation of the iterated tabu search algorithm for the quadratic assignment problem," *OR Spectrum*, vol. 34, no. 3, pp. 665–690, 2012.

[33] G. Palubeckis, "Iterated tabu search for the maximum diversity problem," *Applied Mathematics and Computation*, vol. 189, no. 1, pp. 371–383, 2007.

[34] G. Palubeckis and D. Rubliauskas, "A branch-and-bound algorithm for the minimum cut linear arrangement problem," *Journal of Combinatorial Optimization*, vol. 24, no. 4, pp. 540–563, 2012.

[35] F. Glover, "Heuristics for integer programming using surrogate constraints," *Decision Sciences*, vol. 8, no. 1, pp. 156–166, 1977.

[36] F. Glover, "Future paths for integer programming and links to artificial intelligence," *Computers and Operations Research*, vol. 13, no. 5, pp. 533–549, 1986.