Memetic Graph Clustering

Sonja Biedermann
University of Vienna
Vienna, Austria
sonja.biedermann@univie.ac.at

Monika Henzinger
University of Vienna
Vienna, Austria
monika.henzinger@univie.ac.at

Christian Schulz
University of Vienna
Vienna, Austria
christian.schulz@univie.ac.at

Bernhard Schuster
University of Vienna
Vienna, Austria
bernhard.schuster@univie.ac.at

Abstract

It is common knowledge that there is no single best strategy for graph clustering, which justifies a plethora of existing approaches. In this paper, we present a general memetic algorithm, VieClus, to tackle the graph clustering problem. This algorithm can be adapted to optimize different objective functions. A key component of our contribution are natural recombine operators that employ ensemble clusterings as well as multi-level techniques. Lastly, we combine these techniques with a scalable communication protocol, producing a system that is able to compute high-quality solutions in a short amount of time. We instantiate our scheme with local search for modularity and show that our algorithm successfully improves or reproduces all entries of the 10th DIMACS implementation challenge under consideration using a small amount of time.

2012 ACM Subject Classification Information systems → Clustering, Theory of computation → Evolutionary algorithms

Keywords and phrases Graph Clustering, Evolutionary Algorithms

Digital Object Identifier 10.4230/LIPIcs.SEA.2018.3

Related Version A full version of the paper is available at [9], `https://arxiv.org/abs/1802.07034`.

Funding The research leading to these results has received funding from the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013) /ERC grant agreement No. 340506.

Acknowledgements The authors acknowledge support by the state of Baden-Württemberg through bwHPC.

1 Introduction

Graph clustering is the problem of detecting tightly connected regions of a graph. Depending on the task, knowledge about the structure of the graph can reveal information such as voter behavior, the formation of new trends, existing terrorist groups and recruitment [42] or a natural partitioning of data records onto pages [17]. Further application areas include the study of protein interaction [35], gene expression networks [48], fraud detection [1], program optimization [29, 15] and the spread of epidemics [32] – possible applications are plentiful, as almost all systems containing interacting or coexisting entities can be modeled as a graph.

It is common knowledge that there is no single best strategy for graph clustering, which justifies a plethora of existing approaches. Moreover, most quality indices for graph clusterings have turned out to be NP-hard to optimize and are rather resilient to effective approximations, see e.g. [4, 12, 47], allowing only heuristic approaches for optimization. The majority of algorithms for graph clustering are based on the paradigm of intra-cluster density versus inter-cluster sparsity. One successful heuristic to cluster large graphs is the *multi-level* approach [11], e.g. the Louvain method for the optimization of modularity [10]. Here, the graph is recursively *contracted* to obtain smaller graphs which should reflect the same general structure as the input. After applying an *initial clustering* algorithm to the smallest graph, the contraction steps are undone and, at each level, a *local search* method is used to improve the clustering induced by the coarser level w.r.t some objective function measuring the quality of the clustering. The intuition behind this approach is that a good clustering at one level of the hierarchy will also be a good clustering on the next finer level. Hence, depending on the definition of the neighborhood, local search algorithms are able to explore local solution spaces very effectively in this setting. However, these methods are also prone to get trapped in local optima. The multi-level scheme can help to some extent since local search has a more global view on the problem on the coarse levels and a very fine-grained view on the fine levels of the multi-level hierarchy. In addition, as with many other randomized meta-heuristics, several repeated runs can be made in order to improve the final result at the expense of running time.

Still, even a large number of repeated executions can only scratch the surface of the huge search space of possible clusterings. In order to explore the global solution space extensively, we need more sophisticated meta-heuristics. This is where memetic algorithms (MAs), i.e. genetic algorithms combined with local search [25], come into play. Memetic algorithms allow for effective exploration (global search) and exploitation (local search) of the solution space. The general idea behind genetic algorithms is to use mechanisms inspired by biological evolution such as selection, mutation, recombination, and survival of the fittest. A genetic algorithm (GA) starts with a population of individuals (in our case clusterings of the graph) and evolves the population over several generational cycles (rounds). In each round, the GA uses a selection rule to select good individuals and combines them to obtain improved offspring [21]. When an offspring is generated an eviction rule is used to select a member of the population to be replaced by the new offspring. For an evolutionary algorithm it is of major importance to preserve diversity in the population [5], i.e., the individuals should not become too similar in order to avoid a premature convergence of the algorithm. This is usually achieved by using mutation operations and by using eviction rules that take similarity of individuals into account.

In this paper, we present a general memetic algorithm, VieClus (Vienna Graph Clustering), to tackle the graph clustering problem. This algorithm can be adapted to optimize different objective functions simply by using a local search algorithm that optimizes the objective function desired by the user. A key component of our contribution are natural recombine operators that employ ensemble clusterings as well as multi-level techniques. In machine learning, ensemble methods combine multiple weak classification (or clustering) algorithms to obtain a strong algorithm for classification (or clustering). More precisely, given a number of clusterings, the *overlay/ensemble clustering* is a clustering in which two vertices belong to the same cluster if and only if they belong to the same cluster in each of the input clusterings. Our recombination operators use the overlay of two clusterings from the population to decide whether pairs of vertices should belong to the same cluster [34, 43]. This is combined with a local search algorithm to find further improvements and also embedded into a multi-level algorithm to find even better clusterings. Our general principle is to randomize tie-breaking whenever possible. This diversifies the search and also improves solutions. Lastly, we combine these techniques with a scalable communication protocol, producing a system that is able to compute high-quality solutions in a short amount of time. In our experimental evaluation, we show that our algorithm successfully improves or reproduces all entries of the 10th DIMACS implementation challenge under consideration in a small amount of time. In fact, for most of the small instances, we can improve the old benchmark result *in less than a minute*. Moreover, while the previous best result for different instances has been computed by a variety of solvers, our algorithm can now be used as a single tool to compute the result.

2 Preliminaries

2.1 Basic Concepts

Let $G = (V = \{0, \ldots, n-1\}, E)$ be an undirected graph and $N(v) := \{u : \{v, u\} \in E\}$ denote the neighbors of v. The degree of a vertex v is $d(v) := |N(v)|$. The problem that we tackle in this paper is the *graph clustering* problem. A clustering $\mathcal{C}$ is a partition of the set of vertices, i.e. a set of disjoint *blocks/clusters* of vertices $V_1, \ldots, V_k$ such that $V_1 \cup \cdots \cup V_k = V$. However, k is usually not given in advance. The term $\mathcal{C}[v]$ refers to the cluster of a node v. A size-constrained clustering constrains the size of the blocks of a clustering by a given upper bound U. A clustering is *trivial* if there is only one block, or all clusters/blocks contain only one element, i.e., are singletons. We identify a cluster V_i with its node-induced subgraph of G. The set $E(\mathcal{C}) := E \cap (\cup_i V_i \times V_i)$ is the set of *intra-cluster edges*, and $E \setminus E(\mathcal{C})$ is the set of *inter-cluster edges*. We set $|E(\mathcal{C})| =: m(\mathcal{C})$ and $|E \backslash E(\mathcal{C})| =: \overline{m}(\mathcal{C})$. An edge running between two blocks is called *cut edge*. There are different objective functions that are optimized in the literature. We review some of them in Section 2.3. Our main focus in this work is on *modularity*. However, our algorithm can be generalized to optimize other objective functions. The *graph partitioning problem* is also looking for a partition of the vertices. Here, a *balancing constraint* demands that all blocks have weight $|V_i| \leq (1+\epsilon)\lceil \frac{|V|}{k} \rceil =: L_{\max}$ for some imbalance parameter ϵ. A vertex is a *boundary vertex* if it is incident to a vertex in a different block. The objective is to minimize the total *cut* $|E \cap \bigcup_{i<j} V_i \times V_j|$. Throughout the paper, given a set S, the operator $\in_{\text{rnd}}$ selects an element of S uniformly at random.

2.2 Ensemble/Overlay Clusterings

In machine learning, ensemble methods combine multiple weak classification algorithms to obtain a strong classifier. These base clusterings are used to decide whether pairs of vertices should belong to the same cluster [34, 44]. Given two clusterings, the *overlay clustering* is a clustering in which two vertices belong to the same cluster if and only if they belong to the same cluster in each of the input clusterings. More precisely, given two clusterings $\mathcal{C}_1$

and $\mathcal{C}_2$ the *overlay clustering* is the clustering where each block corresponds to a connected component of the graph $G_{\mathcal{E}} = (V, E \backslash \mathcal{E})$ where $\mathcal{E}$ is the union of the cut edges of $\mathcal{C}_1$ and $\mathcal{C}_2$, i.e. all edges that run between blocks in either $\mathcal{C}_1$ or $\mathcal{C}_2$. Intuitively, if the input clusters agree that two vertices belong to the same cluster, the two vertices belong together with high confidence. It is easy to see that the number of clusters in the overlay clustering cannot be smaller than the number of clusters in each of the input clusterings.

An overlay clustering can be computed in expected linear-time. More precisely, given two clusterings $\{\mathcal{C}_1, \mathcal{C}_2\}$, we use the following approach to compute the overlay clustering. Initially, the overlay clustering $\mathcal{O}$ is set to the clustering $\mathcal{C}_1$. We then iterate through the remaining clusterings and incrementally update the current solution $\mathcal{O}$. To this end, we use pairs of cluster IDs (i, j) as a key in a hash map $\mathcal{H}$, where i is a cluster ID of $\mathcal{O}$ and j is a cluster ID of the current clustering $\mathcal{C}$. We initialize $\mathcal{H}$ to be the empty hashmap and set a counter c to zero. Then we iterate through the nodes. Let v be the current node. If the pair $(\mathcal{O}[v], \mathcal{C}[v])$ is not contained in $\mathcal{H}$, we set $\mathcal{H}(\mathcal{O}[v], \mathcal{C}[v])$ to c and increment c by one. Afterwards, we update the cluster ID of v in $\mathcal{O}$ to be $\mathcal{H}(\mathcal{O}[v], \mathcal{C}[v])$. At the end of the algorithm, c is equal to the number of clusters contained in the overlay clustering and each vertex is labeled with its cluster ID in $\mathcal{O}$.

2.3 Objective Functions

There are a variety of measures used to assess the quality of a clustering, such as *coverage* [11], *performance* [46], *inter-cluster conductance* [24], *surprise* [3], *map equation* [38] and *modularity* [33]. The most simple index realizing a traditional measure of clustering quality is coverage. The coverage of a graph clustering $\mathcal{C}$ is defined as the fraction of intra-cluster edges within the complete set of edges $\text{cov}(\mathcal{C}) := \frac{m(\mathcal{C})}{m}$. Intuitively, large values of coverage correspond to a good quality of a clustering. However, one principal drawback of coverage is that the converse is not necessarily true: coverage takes its largest value of 1 in the trivial case where there is only one cluster. *Modularity* fixes this issue by comparing the coverage of a clustering to the same value after rearranging edges randomly keeping node degrees. *Performance* is the fraction of correctly classified vertex pairs, w.r.t the set of edges. *Inter-cluster conductance* returns the worst (i.e. the thickest) bottleneck created by separating a cluster from the rest of the graph. *Surprise* measures the probability that a random graph $\mathcal{R}$ has more intra-cluster edges. *Map equation* is a flow-based and information-theoretic method to assess clustering quality. Our focus in this work is on modularity as it is a widely accepted quality function and has been the main objective function in the 10th DIMACS implementation challenge [7]. For further discussions of these indices we refer the reader to the given references, and simply state the formal definition of modularity here:

$$\mathcal{Q}(\mathcal{C}) := cov(\mathcal{C}) - \mathbb{E}[cov(\mathcal{C})] = \frac{m(\mathcal{C})}{m} - \frac{1}{4m^2} \sum_{V_i \in \mathcal{C}} \left(\sum_{v \in V_i} d(v) \right)^2$$

2.4 Related Work

This paper is a summary and extension of the bachelor's thesis by Sonja Biedermann [8]. There has been a *significant* amount of research on graph clustering. We refer the reader to [20, 23] for thorough reviews of the results in this area. Here, we focus on results closely related to our main contributions. It is common knowledge that there is no single best strategy for graph clustering. Moreover, most quality indices for graph clusterings have turned out to be NP-hard to optimize and rather resilient to effective approximations, see e.g. [4, 12, 47], allowing only heuristic approaches for optimization. Other approaches often rely on specific strategies with high running times, e.g. the iterative removal of central edges [33], or the direct identification of dense subgraphs [16]. Provably good methods with a decent running time include algorithms that have a spectral embedding of the vertices as the basis for a geometric clustering [13], min-cut tree clustering [19], a technique which guarantees certain bounds on bottlenecks and an approach which relies on random walks in graphs staying inside dense regions with high probabilities [46]. The Louvain method is a multi-level clustering algorithm introduced by Blondel et al. [10] that optimizes modularity as an objective function. As we use this method in our algorithm to create the initial population as well as to improve individuals after recombination, we go into more detail for that method in Section 2.6.

Recently, the 10th DIMACS challenge on graph partitioning and graph clustering compared different state-of-the-art graph clustering algorithms w.r.t optimizing modularity [6]. Most of the best results have been obtained by Ovelgönne and Geyer-Schulz [34] as well as Aloise et al. [2]. CGGC [34] used an ensemble learning strategy during the challenge. From several weak clusterings an overlap clustering is computed and a more expensive clustering algorithm is applied. VNS [2] applies the Variable Neighborhood Search heuristic to the graph clustering problem. Later, Džamić et al. [18] outperformed VNS by proposing an ascent-decent VNS. We compare ourselves to the results above in Section 4.

Evolutionary Graph Clustering. Tasgin and Bingol [45] introduce a genetic algorithm using modularity as a quality measure. They chose an integer encoding for representing the population clusterings. Individuals are randomly initialized, with some bias for assigning direct neighbors to the same cluster. Recombination is done one way, i.e. instead of using mutual exchange, clusters are transferred from a source individual to a target individual. More precisely, the operation picks a random vertex in the source individual and clusters all vertices of its cluster together in the destination individual. This is repeated several times in one recombination operation. As for mutation, the authors chose to pick one vertex and move it to a random cluster, which is almost guaranteed to decrease the fitness. This approach does not use local search to improve individuals.

A mutation-less agglomerative hierarchical genetic algorithm is presented by Lipczak and Milios in [27]. The authors represent each cluster as one individual and use one-point and uniform crossovers in that representation. Two synthetic networks have been used for the experimental evaluation. An important advantage of this algorithm is the possibility to distribute its computations.

2.5 Karlsruhe High-Quality Partitioning

Within this work, we use the open source multi-level graph partitioning framework KaHIP [41] (Karlsruhe High-Quality Partitioning). More precisely, we employ partitioning tools contained therein to create high-quality partitions of the graphs. We shortly outline its main components. KaHIP implements many different algorithms, for example, flow-based methods and more-localized local searches within a multi-level framework, as well as several coarse-grained parallel and sequential meta-heuristics. Recently, specialized methods to partition social networks and web graphs have been included in the framework [30].

2.6 Multi-level Louvain Method

The Louvain method is a multi-level clustering algorithm introduced by Blondel et al. [10]. It is an approach to graph clustering that optimizes modularity as an objective function. Since we instantiate our memetic algorithm to optimize for modularity, we give more detail in order to be self-contained. The algorithm is organized in two phases: a local movement phase and contraction/uncontraction phase. The first phase, *local movement*, works in rounds and is done as follows: In the beginning, each vertex is a singleton cluster. Vertices are then traversed in random order and always moved to the neighboring cluster yielding the highest modularity increase. Computing the best move can be done in time proportional to the degree of a vertex by storing cumulative vertex degrees of clusters. More precisely, the gain in modularity by removing a vertex from a cluster can be computed by

$$\Delta Q(u) = \frac{s}{m} + \frac{d(u)}{4m^2} + \frac{(\sum_{v \in C} d(v))^2}{4m^2} - \frac{(d(u) + \sum_{v \in C} d(v))^2}{4m^2}.$$

A similar formula holds when adding a singleton vertex to a cluster. Once a vertex is moved, the cumulated degrees of the affected clusters are updated. Hence, the best move can be found in time proportional to the degree of a vertex. The local movement algorithm stops when a local maximum of the modularity is attained, that is when no vertex move a yields modularity gain. The *second phase* of the algorithm consists in contracting the clustering. Contracting the clustering works as follows: each block of the clustering is contracted into a single node. There is an edge between two vertices u and v in the contracted graph if the two corresponding blocks in the clustering are adjacent to each other in G, i.e. block u and block v are connected by at least one edge. The weight of an edge (A, B) is set to the sum of the weight of the edges that run between block A and block B of the clustering. Moreover, a self-loop is inserted for each vertex in the contracted graph. The weight of this edge is set to be the cumulative weight of the edges in the respective cluster. Note that due to the way the contraction is defined, a clustering of the coarse graph corresponds to a clustering of the finer graph with the same objective. The algorithm then continues with the local movement phase on the contracted graph. In the end, the clustering contractions are undone and at each level local movement improves the current clustering w.r.t. modularity.

3 Memetic Graph Clustering

We now explain the components of our memetic graph clustering algorithm. Our algorithm starts with a population of individuals (in our case one individual is a clustering of the graph) and evolves the population into different populations over several rounds. In each round, the GA uses a selection rule based on the fitness of the individuals (in our case the objective function of the clustering problem under consideration) of the population to select good individuals and recombine them to obtain improved offspring. Our selection process is based on the tournament selection rule [31], i.e. $\mathcal{C}$ is the fittest out of two random individuals R_1, R_2 from the population. When an offspring is generated an elimination rule is used to select a member of the population and replace it with the new offspring. In general, one has to take both into consideration, the fitness of an individual and the distance between individuals in the population [36] in order to avoid premature convergence of the algorithm. We evict the solution that is *most similar* with respect to the edges that run in between clusters with the offspring among those individuals in the population that have a worse or equal objective than the offspring itself. If there is no such individual, then the offspring is rejected and not inserted into the population. The difference between two individuals is defined as the size of the symmetric difference between their sets of cut edges. Our algorithm generates only one offspring per generation.

The core of our algorithm are our novel recombination and mutation operations. We provide two different kinds of operations, flat- and multi-level recombination operations. We also define a mutation operator that splits clusters by employing graph partitioning techniques. In any case, the offspring is typically improved using a local search algorithm that optimizes for the objective function of the graph clustering problem. We continue this section by explaining the details of our algorithm with a focus on modularity – the results are transferable to other clustering problems by using local search algorithms that optimize the objective function of the problem under consideration.

3.1 Initialization/Creating Individuals

We initialize our population using the following modified Louvain algorithm. We also use the following algorithm to create an individual in recombine and mutation operations. Depending on the objective of the evolutionary algorithm, the choice of algorithms to initialize the population may be different. We describe our scheme for modularity.

We mainly use the Louvain method to create clusterings. Due to the non-deterministic nature of this algorithm – the order of nodes visited is randomized – we obtain different initial graph clusterings and later on individuals. In order to introduce more diversification, we modify the approach by using a different coarsening strategy based on the size-constrained label propagation algorithm [30]. Label propagation works similar to the local movement phase of the Louvain method. However, the objective of the algorithm is different. Without size-constraints, it was proposed by Raghavan et al. [37]. Initially, each vertex is in its own cluster/block, i.e. the initial block ID of a vertex is set to its vertex ID. The algorithm then works in rounds. In each round, the vertices of the graph are traversed in a random order. When a vertex v is visited, it is *moved* to the block that has the strongest connection to v, i.e. it is moved to the cluster V_i that maximizes $\omega(\{(v,u) \mid u \in N(v) \cap V_i\})$ whilst not overloading the target cluster w.r.t to the size-constraint bound U. Ties are broken randomly. The process is repeated until the process has converged. Here, we perform at most ℓ rounds of the algorithm, where ℓ is a tuning parameter, and stop the algorithm if less than five percent of the vertices changed its cluster during one round. One LPA round can be implemented to run in $\mathcal{O}(n+m)$ time.

We modify the coarsening stage of the Louvain method using size-constrained label propagation as follows: For the first $\lambda \in_{\text{rnd}} [0,4]$ levels of the multi-level hierarchy we use size-constraint label propagation to compute the clustering to be contracted instead of the local movement phase to compute a clustering. We choose $U \in_{\text{rnd}} [n/10, n]$ in the beginning of the multi-level algorithm. Afterwards, we switch to the local movement phase of the Louvain method to compute a clustering to be contracted. Note that for $\lambda = 0$ the method is the Louvain method. In any case, in the uncoarsening phase local movement improves the current clustering by optimizing modularity.

3.2 Recombination

We define flat- and multi-level recombination operations which we are going to explain now. *Both* operations use the notion of overlay clustering to pass on good parts of the solutions to the offspring. Some of our recombination operators ensure that the offspring has non-decreasing fitness. Moreover, our recombine operations can recombine a solution from the population with an arbitrary clustering of the graph. Due to the fact that our local search and multi-level algorithms are randomized, a recombine operation performed twice using the same parents can yield different offsprings.

Flat Recombination. The *basic flat recombination operation* starts by taking two clusterings $\mathcal{C}_1, \mathcal{C}_2$ from the population as input and computes its overlay. The overlay is then contracted such that vertices represent blocks in the overlay clustering and edges represent edges between vertices of two blocks. Edge weights are equal to the summed weight of the edges that run between the respective blocks, and self-loops are inserted with the edge weight being equal to the summed internal edge weight of that corresponding block. Note that due to the way contraction is defined, a clustering of the contracted graph can be transformed into a clustering of the input network having the same modularity score. Hence, we use the Louvain method to cluster the contracted graph, which then constitutes the offspring. Also, note that the contraction of the overlay ensures that vertices that are clustered together in both inputs will belong to the same cluster in the offspring. Only vertices which are split by one of the input clusterings will be affected by the Louvain method. What follows are multiple variations of the basic flat recombination method that ensure non-decreased fitness of the input individual, i.e. the offspring has fitness at least as good as the better input individual.

Apply Input Clustering. This operator uses the better of the two parent clusterings as a starting point for the Louvain method on the contracted graph. Due to the way contraction is defined the objective function of the starting point on the contracted graph is the same as of the corresponding clustering on the input network. The resulting offspring can be expected to be less diverse than when using the plain recombination operator, but may significantly more improve upon the parent.

Cluster-/Partition Recombination. This operator differs from the previous two operators insofar that only one of the parents is selected from the population. The other parent is manufactured on the spot by running either the size-constrained label propagation (SCLP) or a graph partitioning algorithm on the input graph. The resulting clustering can be very different from to the other parent. This introduces more diversification which helps local search to explore a larger search space. When using SCLP clustering for the recombination, we use the parameters as described in Section 3.1. When using a partition, we use the KaHIP with the fastsocial preconfiguration using $k \in_{\text{rnd}} [2, 64]$ and $\epsilon \in_{\text{rnd}} [0.03, 0.5]$.

Multi-level Recombination. We now explain our multi-level recombine operator. This recombine operator also ensures that the solution quality of the offspring is *at least as good as the best of both parents.* For our recombine operator, let $\mathcal{C}_1$ and $\mathcal{C}_2$ again be two individuals from the population. Both individuals are used as input for the multi-level Louvain method in the following sense. Let $\mathcal{E}$ be the set of edges that are cut edges, i.e. edges that run between two blocks, in either $\mathcal{C}_1$ *or* $\mathcal{C}_2$. All edges in $\mathcal{E}$ are blocked during the coarsening phase, i.e. they are *not* contracted during the coarsening phase. In other words, these edges cannot be contracted during the multi-level scheme. We ensure this by modifying the local movement phase of the Louvain method, i.e. clusters can only grow inside connected components of the overlay $(V, E \backslash \mathcal{E})$. We stop contracting clusterings when no contractable edge is left.

When coarsening is stopped, we then apply the better out of both input individuals w.r.t. the objective to the coarsest graph and use this as initial clustering instead of running the Louvain method on the coarsest graph. However, during uncoarsening local search still optimizes modularity on each level of the hierarchy. Note that this is possible since we did not contract any cut edge of both inputs. Also, note that this way we obtain a clustering of the coarse graph having a modularity score being equal to the better of both input individuals. Local movement guarantees no worsening of the clustering. Hence, the offspring is at least as good as the best input individual. Note that the coarsest graph is the same as the graph obtained in a flat recombination. However, the operation now is able to pass on good parts of the solution on multi-levels of the hierarchy during uncoarsening and hence has a more fine-grained view on both individuals.

3.3 Mutation

Our recombination operators can only decrease the number of clusters present in the solution. To counteract this behavior, we define a mutation operator that selects a subset of clusters and splits each of them in half. Splitting is done using the KaHIP graph partitioning framework. The number of clusters to be split is set to $p_s|\mathcal{C}|$ where p_s is the splitting probability and $|\mathcal{C}|$ is the number of clusters that the clustering selected from the population has. That means that we split a cluster into two balanced blocks such that there are only a small amount of edges running between them. We do this operation with two individuals from the population that are found by tournament selection. This results in two output individuals which are used as input to a multi-level recombination operation. The computed offspring of that operation is then inserted into the population as described above.

3.4 Parallelization

We now explain the island-based parallelization that we use. We use a parallelization scheme that has been successfully used in graph partitioning [40]. Each processing element (PE) basically performs the same operations using different random seeds. First, we estimate the population size $\mathcal{S}$: each PE creates an individual and measures the time $\bar{t}$ spent. We then choose $\mathcal{S}$ such that the time for creating $\mathcal{S}$ clusterings is approximately t_{total}/c where the fraction c is a tuning parameter and t_{total} is the total running time that the algorithm is given to produce a clustering of the graph. The minimum amount of individuals in the population is set to 3, the maximum amount of the individuals in the population is set to 100. The lower bound on the population size is chosen to ensure a certain minimum of diversity, while the upper bound is used to ensure convergence. Each PE then builds its own population. Afterwards, the algorithm proceeds in rounds as long as time is left. Either a mutation or recombination operation is performed. Over time, the best individuals are exchanged between the PEs. Our communication protocol is similar to *randomized rumor spreading* which has shown to be scalable in previous work [40]. We refer to the full version of the paper [9] for a description of the communication protocol.

4 Experimental Evaluation

System and Methodology. We implemented the memetic algorithm described in the previous section within the KaHIP (Karlsruhe High Quality Partitioning) framework. The code is written in C++ and MPI. It has been compiled using g++-5.2 with flags `-O3` and OpenMPI 1.6.5. We refer to the algorithm presented in this paper as VieClus. The code is available at `http://vieclus.taa.univie.ac.at/`. Throughout this section, our main objective is modularity. All experiments comparing VieClus with competing algorithms are performed on a cluster with 512 nodes, where each node has two Intel Xeon E5-2670 Octa-Core (Sandy Bridge) processors clocked at 2.6 GHz, 64 GB main memory, 20 MB L3- and 8x256 KB L2-Cache and runs RHEL 7.4. We use the *arithmetic mean* when averaging over solutions of the same instance and the *geometric mean* when averaging over different instances in order to give every instance a comparable influence on the final result. It is well known that the algorithms that scored most of the points during the 10th DIMACS challenge compute better results than the Louvain method. Hence, we refrain from doing additional experiments with the Louvain method.

Table 1 Results of our algorithm on the benchmark test set. Columns from left to right: average modularity achieved by our algorithm (Avg. $\mathcal{Q}$), $\bar{t}$ average time in minutes needed to beat the old challenge result, the best score computed of *our* algorithm (Max. $\mathcal{Q}$), the best scores achieved by challenge participants, running time in minutes needed to create the previous best entry according to [26] and reference to the solver that achieved the result during the 10th DIMACS challenge.

Graph	Avg. $\mathcal{Q}$	$\bar{t}$ [m]	Max. $\mathcal{Q}$	$\mathcal{Q}$ [7]	t_{sol}[m]	Solver
Small Instances						
`as-22july06`	**0,679 391**	<1	**0,679 396**	0,678 267	6,6	CGGC[34]
`astro-ph`	**0,746 285**	<1	**0,746 292**	0,744 621	11,9	VNS[2]
`celegans_metabol`	*0,453 248*	<1	*0,453 248*	0,453 248	<1	VNS[2]
`cond-mat-2005`	**0,750 065**	<1	**0,750 171**	0,746 254	40,9	CGGC[34]
`email`	*0,582 829*	<1	*0,582 829*	0,582 829	<1	VNS[2]
`PGPgiantcompo`	**0,886 853**	<1	**0,886 853**	0,886 564	1,9	CGGC[34]
`polblogs`	*0,427 105*	<1	*0,427 105*	0,427 105	<1	VNS[2]
`power`	**0,940 975**	<1	**0,940 977**	0,940 851	<1	VNS[2]
`smallworld`	**0,793 186**	<1	**0,793 187**	0,793 042	16,8	VNS[2]
`memplus`	**0,701 242**	2,6	**0,701 275**	0,700 473	3,2	CGGC[34]
`G_n_pin_pout`	**0,500 457**	3,9	**0,500 466**	0,500 098	64,8	CGGC[34]
`caidaRouterLevel`	**0,872 804**	5,0	**0,872 828**	0,872 042	81,0	CGGC[34]
`rgg_n17`	**0,978 448**	5,0	**0,978 454**	0,978 324	37,5	VNS[2]
`luxembourg.osm`	**0,989 665**	7,3	**0,989 672**	0,989 621	40,9	VNS[2]
Large Instances						
`coAuthorsCiteseer`	**0,906 804**	3,9	**0,906 830**	0,905 297	91,3	CGGC[34]
`citationCiteseer`	**0,825 518**	12,9	**0,825 545**	0,823 930	77,6	CGGC[34]
`coPapersDBLP`	**0,868 019**	20,5	**0,868 058**	0,866 794	603,3	CGGC[34]
`belgium.osm`	**0,995 062**	29,5	**0,995 064**	0,994 940	102,9	CGGC[34]
`ldoor`	**0,970 521**	35,1	**0,970 555**	0,969 370	485,6	ParMod[14]
`eu-2005`	**0,941 575**	65,8	**0,941 575**	0,941 554	341,5	CGGC[34]
`in-2004`	**0,980 684**	237,4	**0,980 690**	0,980 622	244,0	CGGC[34]
`333SP`	**0,989 316**	297,1	**0,989 356**	0,989 095	976,9	ParMod[14]
`prefAttachment`	0,315 843	*	**0,316 089**	0,315 994	1 353,1	VNS[2]

Parameters. Our algorithm is not very sensitive to the precise choice parameters. We *did not* perform a tuning of the parameters of the algorithm, rather we chose the parameters described above and below to be reasonable and to introduce a large amount of diversification or we chose parameters that were a good choice in previous evolutionary algorithms [40]. We expect the parameters to work well with a varity of instances. As our main design goal is to introduce as much diversification as possible, we use all recombination and mutation operations in our algorithm. The ratio of mutation to recombine operations has been set to 1:9 as this has been a good choice in previous evolutionary algorithms [40]. When we perform a recombine operation, we pick the recombine operation uniformly at random and diversify the parameters as described above. When performing a mutation operation, we use a splitting probability p_s uniformly at random in $[0.01, 0.1]$. We invest 1/10 of the total time to create the initial population.

Instances. We use the graphs that have been used for the 10th DIMACS implementation challenge on graph clustering and graph partitioning [7]. A list of the instances as well as the modularity scores that have been obtained during the challenge can be found in Table 1. We exclude the instances `cage15`, `audikw1`, `er-fact1.5-scale25`, `kron_*`, `uk-2002`, `uk-2007-05` from the challenge testbed in our evaluation, because they are either too large to be feasible for an evolutionary algorithm or they do not contain a significant cluster structure as indicated by the reported modularity score [6].

4.1 Evolutionary Graph Clustering

We now run our algorithm on all the DIMACS instances under consideration using the rules used there, i.e. *running time is not an issue* but we want to achieve modularity values as large as possible for each instance. On the small instances, we give our algorithm 2 hours of time to compute a solution and on the large instances, we set the time limit to 16 hours. In any case, we use 16 cores of our machine, i.e. one node of the machine. We perform the test five times with different random seeds. Table 1 summaries the results of our experiment, and the technical report [9] contains convergence plots for a selected subset of the instances.

First of all, on *every* instance under consideration, our algorithm is able to compute a result that is better than the currently reported modularity value in literature. More precisely, in 98 out of 115 runs of our algorithm, the previous benchmark result of the 10th DIMACS implementation challenge is outperformed. In further 15 out of 115 runs, we reproduce the results. This is the case for each of the runs for the graphs `celegans_metabolic`, `email` and `polblogs`. The two cases in which our algorithm does not beat the previous best solver, are 2 runs on the graph `prefAttachment`. The other 3 runs on that graph outperform the previous result. The time needed to compute a clustering having a similar score on that graph is roughly 95% of the total running time. Moreover, as convergence plots in [9] show this may be fixable by giving the algorithm a larger amount of time to compute the solution.

Overall, the time needed to outperform the previous benchmark results ranges from less than a minute to a couple of minutes on the small instances. On the large instances our algorithm needs more time, but on most instances, the previous benchmark result can be computed in roughly an hour. Table 1 also reports the running time of the solver that obtained the result during the DIMACS challenge. Our algorithm is faster in every case compared to the previous solver (eventually by more than an order of magnitude), however, the machines used for the experiment are different and our algorithm is a parallel algorithm whereas previous solvers are sequential. Note that the final improvements over the old result are fairly small (<0.1% on average). This is not surprising, as previous solvers already invested a large amount of time to compute the results. However, note that the previous result has been computed by different solvers and our evolutionary algorithm can be seen as a single tool to compute the result.

We now compare the results with recently published results [26, 22, 28, 39]. LaSalle [26] reports results on all graphs from the DIMACS challenge subset. However, each of the best computed result is worse than the result computed by the respective best algorithm during the DIMACS challenge (and hence worse compared to our algorithm). Moreover, LaSalle [26] reports that his Nerstrand algorithm is on average equal or slightly better than the Louvain method on the instances used here. Lu et al. [28] present a result for `coPapersDBLP` ($\mathcal{Q}$ = 0,858 088) and Ryu and Kim [39] report modularity for `email` ($\mathcal{Q}$ = 0,568) which are worse compared to the results that we report here. Hamann et al. [22] report the result for `in-2004` ($\mathcal{Q}$ = 0,980) which is comparable to the result that we report here. Džamić [18] build upon VNS proposing an ascent-decent VNS. Results are reported for `celegans_metabolic` ($\mathcal{Q}$ = 0.453248), `email` ($\mathcal{Q}$ = 0,582 829), `polblogs` ($\mathcal{Q}$ = 0,427 105) for which their algorithm computes the same results as all other tools reported in Table 1. Moreover, the following best results are reported `as-22july06` ($\mathcal{Q}$ = 0,678 381), `astro-ph` ($\mathcal{Q}$ = 0,745 246), `cond-mat-2005` ($\mathcal{Q}$ = 0,747 181), `PGPgiantcompo` ($\mathcal{Q}$ = 0.886647), as well as `power` ($\mathcal{Q}$ = 0.940974) which are indeed better than the previous best result obtained during the 10th DIMACS challenge, but still worse than average result of our algorithm in every case. Hence, overall we consider our algorithm as a new state-of-the-art heuristic for solving the modularity clustering problem.

Table 2 Columns from left to right: the approximate bound computed with the heuristic algorithm to balance cluster volumes and the ratio of our best value and the bound.

Graph	B	$\mathcal{Q}/B$
Small Instances		
as-22july06	0,973 684	0,697 758
astro-ph	0,999 070	0,746 987
celegans_metabol	0,888 889	0,509 904
cond-mat-2005	0,999 468	0,750 570
email	0,900 000	0,647 588
PGPgiantcompo	0,989 796	0,895 996
polblogs	0,996 168	0,428 748
power	0,975 610	0,964 501
smallworld	0,995 652	0,796 651
memplus	0,984 127	0,712 586
G_n_pin_pout	0,993 865	0,503 555
caidaRouterLevel	0,997 758	0,874 789
rgg_n17	0,992 537	0,985 811
luxembourg.osm	0,996 283	0,993 364
Large Instances		
coAuthorsCiteseer	0,995 575	0,910 861
citationCiteseer	0,993 289	0,831 123
coPapersDBLP	0,995 283	0,872 172
belgium.osm	0,998 358	0,996 701
ldoor	0,989 796	0,980 561
eu-2005	0,996 564	0,944 821
in-2004	0,999 136	0,981 538
333SP	0,995 726	0,993 603
prefAttachment	0,875 000	0,361 245

4.2 Approximate Bound for Modularity

The modularity clustering problems seeks to maximize cov($\mathcal{C}$) - $\mathbb{E}$[cov($\mathcal{C}$)]. However, the problem is NP-complete and our algorithm heuristically finds solutions. Moreover, modularity is trivially bounded by 1. The true optimum clustering, however, has typically a value below that. Thus we try to find a bound or a value to normalize the score such that one gets a clearer picture on how far away from the optimum score the achieved modularity value is.

To do so, let us define $\sum_{v \in V_i} d(v)$ to be the volume $\text{vol}(V_i)$ of the cluster V_i. Note that the formula $\frac{1}{4m^2} \sum_{V_i \in \mathcal{C}} \text{vol}(V_i)^2$ is minimized if all clusters have the same volume. But again, finding the optimum value of this equation is hard. As $\sum_{V_i \in \mathcal{C}} \text{vol}(V_i) = 2m$, the equation is minimized when $\text{vol}(V_i) = 2m/k \ \forall i = 1 \dots k$, where k is the number of clusters. Thus it follows that $\frac{1}{4m^2} \sum_{V_i \in \mathcal{C}} \text{vol}(V_i)^2 \geq \frac{1}{4m^2} \cdot k \cdot (\frac{2m}{k})^2 = \frac{1}{k}$. This gives an upper of $1 - \frac{1}{k}$ for $\mathcal{Q}(\mathcal{C})$ if the number of clusters is an input to the clustering problem. This upper bound is, however, not very far from the trivial upper bound of large k.

Thus for each graph we also clustered the nodes into cluster V_i using the following heuristic to try to balance the volumes of the resulting clusters as much as possible: We sort the nodes in decreasing order of their degree and then assign them step by step to the cluster having the smallest volume. We break ties by using the cluster with the smallest ID among those having the smallest volume. For this heuristic, we use the number of clusters that our algorithm has computed as value for k. For the resulting clustering $\mathcal{C}^*$ we compute the value $\mathbb{E}[\text{cov}(\mathcal{C}^*)]$ and use $B := 1 - \mathbb{E}[\text{cov}(\mathcal{C}^*)]$ as a value to normalize $\mathcal{Q}(\mathcal{C})$. Table 2 shows the resulting bound and compares them against our results.

5 Conclusion

We presented a parallel memetic algorithm, VieClus, that tackles the graph clustering problem. A key component of our contribution are natural recombine operators that employ ensemble clusterings as well as multi-level techniques. We combine these techniques with a scalable communication protocol, producing a system that is able to reproduce or improve previous all entries of the 10th DIMACS implementation challenge under consideration as well as results recently reported in the literature in a short amount of time. Moreover, while the previous best result for different instances has been computed by a variety of solvers, our algorithm can now be used as a single tool to compute the result. Hence, overall we consider our algorithm as a new state-of-the-art heuristic for solving the modularity clustering problem. In the future, it may be interesting to instantiate our scheme for different objective functions depending on the application domain or to use a more diverse set of initial solvers to create the population. We also want to look at distributed memory parallel multi-level algorithms for the problem that can use the depicted algorithm as an initial clustering scheme on the coarsest level of the hierarchy.

References

1 L. Akoglu, H. Tong, and D. Koutra. Graph Based Anomaly Detection and Description: A Survey. *Data Min. Knowl. Discov.*, 29(3):626–688, may 2015. `doi:10.1007/s10618-014-0365-y`.

2 D. Aloise, G. Caporossi, S. Perron, P. Hansen, L. Liberti, and M. Ruiz. Modularity Maximization in Networks by Variable Neighborhood Search. In *10th DIMACS Impl. Challenge Workshop*. Georgia Inst. of Technology, Atlanta, GA, 2012.

3 V. Arnau, S. Mars, and I. Marín. Iterative Cluster Analysis of Protein Interaction Data. *Bioinformatics*, 21(3):364–378, 2004.

4 G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation: Combinatorial Optimization Problems and their Approximability Properties.* Springer Science & Business Media, 2012.

5 T. Bäck. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms.* PhD thesis, Informatik Centrum Dortmund, Germany, 1996.

6 D. Bader, A. Kappes, H. Meyerhenke, P. Sanders, C. Schulz, and D. Wagner. Benchmarking for Graph Clustering and Partitioning. In *Encyclopedia of Social Network Analysis and Mining*. Springer, 2014.

7 D. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, editors. *Proc. of the 10th DIMACS Impl. Challenge*, Cont. Mathematics. AMS, 2012.

8 S. Biedermann. Evolutionary Graph Clustering. Bachelor's Thesis, Universität Wien, 2017.

9 S. Biedermann, M. Henzinger, C. Schulz, and B. Schuster. Memetic Graph Clustering (see ArXiv preprint arXiv:1802.07034). *Technical Report. arXiv:1802.07034*, 2018.

10 V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast Unfolding of Communities in Large Networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, 2008. URL: `http://stacks.iop.org/1742-5468/2008/i=10/a=P10008`.

11 U. Brandes. *Network Analysis: Methodological Foundations*, volume 3418. Springer Science & Business Media, 2005.

12 U. Brandes, D. Delling, M. Gaertler, R. Gorke, M. Hoefer, Z. Nikoloski, and D. Wagner. On Modularity Clustering. *IEEE Transactions on Knowledge and Data Engineering*, 20(2):172–188, 2008.

13 U. Brandes, M. Gaertler, and D. Wagner. Engineering Graph Clustering: Models and Experimental Evaluation. *ACM Journal of Experimental Algorithmics*, 12(1.1):1–26, 2007.

14 Ü. V. Çatalyürek, K. Kaya, J. Langguth, and B. Uçar. A Divisive Clustering Technique for Maximizing the Modularity. In *10th DIMACS Impl. Challenge Workshop*. Georgia Inst. of Technology, Atlanta, GA, 2012.

15 J. Demme and S. Sethumadhavan. Approximate Graph Clustering for Program Characterization. *ACM Trans. Archit. Code Optim.*, 8(4):21:1–21:21, 2012. doi:10.1145/2086696.2086700.

16 I. Derényi, G. Palla, and T. Vicsek. Clique Percolation in Random Networks. *Physical review letters*, 94(16):160202, 2005.

17 A. A. Diwan, S. Rane, S. Seshadri, and S. Sudarshan. Clustering Techniques for Minimizing External Path Length. In *Proceedings of the 22th International Conference on Very Large Data Bases*, VLDB '96, pages 342–353, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc. URL: http://dl.acm.org/citation.cfm?id=645922.673636.

18 D. Džamić, D. Aloise, and N. Mladenović. Ascent–descent Variable Neighborhood Decomposition Search for Community Detection by Modularity Maximization. *Annals of Operations Research*, Jun 2017. doi:10.1007/s10479-017-2553-9.

19 G. W. Flake, R. E. Tarjan, and K. Tsioutsiouliklis. Graph Clustering and Minimum Cut Trees. *Internet Mathematics*, 1(4):385–408, 2004.

20 S. Fortunato. Community Detection in Graphs. *Physics reports*, 486(3):75–174, 2010.

21 D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, 1989.

22 M. Hamann, B. Strasser, D. Wagner, and T. Zeitz. Simple Distributed Graph Clustering using Modularity and Map Equation. *arXiv preprint arXiv:1710.09605*, 2017.

23 T. Hartmann, A. Kappes, and D. Wagner. Clustering Evolving Networks. In *Algorithm Engineering*, pages 280–329. Springer, 2016.

24 R. Kannan, S. Vempala, and A. Vetta. On clusterings: Good, bad and spectral. *Journal of the ACM (JACM)*, 51(3):497–515, 2004.

25 J. Kim, I. Hwang, Y. H. Kim, and B. R. Moon. Genetic Approaches for Graph Partitioning: A Survey. In *Proceedings of the 13th Annual Genetic and Evolutionary Computation Conference (GECCO'11)*, pages 473–480. ACM, 2011. doi:10.1145/2001576.2001642.

26 D. LaSalle. Graph Partitioning, Ordering, and Clustering for Multicore Architectures, 2015.

27 M. Lipczak and E. E. Milios. Agglomerative Genetic Algorithm for Clustering in Social Networks. In Franz Rothlauf, editor, *GECCO*, pages 1243–1250. ACM, 2009. URL: http://dblp.uni-trier.de/db/conf/gecco/gecco2009.html#LipczakM09.

28 H. Lu, M. Halappanavar, and A. Kalyanaraman. Parallel heuristics for scalable community detection. *Parallel Computing*, 47:19–37, 2015.

29 S. McFarling. Program Optimization for Instruction Caches. *SIGARCH Comput. Archit. News*, 17(2):183–191, 1989. doi:10.1145/68182.68200.

30 H. Meyerhenke, P. Sanders, and C. Schulz. Partitioning Complex Networks via Size-Constrained Clustering. In *SEA*, volume 8504 of *Lecture Notes in Computer Science*, pages 351–363. Springer, 2014.

31 B. L Miller and D. E Goldberg. Genetic Algorithms, Tournament Selection, and the Effects of Noise. *Evolutionary Computation*, 4(2):113–131, 1996.

32 M. E. J. Newman. Properties of Highly Clustered Networks. *Physical Review E*, 68(2):026121, 2003.

33 M. E. J. Newman and M. Girvan. Finding and Evaluating Community Structure in Networks. *Physical review E*, 69(2):026113, 2004.

34 M. Ovelgönne and A. Geyer-Schulz. An Ensemble Learning Strategy for Graph Clustering. In *Graph Partitioning and Graph Clustering*, number 588 in Contemporary Mathematics, 2013.

35 J. B. Pereira-Leal, A. J. Enright, and C. A. Ouzounis. Detection of Functional Modules from Protein Interaction Networks. *Proteins: Structure, Function, and Bioinformatics*, 54(1):49–57, 2004. `doi:10.1002/prot.10505`.

36 D. C. Porumbel, J.-K. Hao, and P. Kuntz. Spacing Memetic Algorithms. In *13th Annual Genetic and Evolutionary Computation Conference, GECCO 2011, Proceedings, Dublin, Ireland, July 12-16, 2011*, pages 1061–1068, 2011.

37 U. N. Raghavan, R. Albert, and S. Kumara. Near Linear Time Algorithm to Detect Community Structures in Large-Scale Networks. *Physical Review E*, 76(3), 2007.

38 M. Rosvall, D. Axelsson, and C. T. Bergstrom. The Map Equation. *The European Physical Journal-Special Topics*, 178(1):13–23, 2009.

39 S. Ryu and D. Kim. Quick Community Detection of Big Graph Data Using Modified Louvain Algorithm. In *High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2016 IEEE 18th International Conference on*, pages 1442–1445. IEEE, 2016.

40 P. Sanders and C. Schulz. Distributed Evolutionary Graph Partitioning. In *Proc. of the 12th Workshop on Algorithm Engineering and Experimentation (ALENEX'12)*, pages 16–29, 2012.

41 P. Sanders and C. Schulz. Think Locally, Act Globally: Highly Balanced Graph Partitioning. In *12th International Symposium on Experimental Algorithms (SEA'13)*. Springer, 2013.

42 S. E. Schaeffer. Survey: Graph Clustering. *Comput. Sci. Rev.*, 1(1):27–64, 2007. `doi:10.1016/j.cosrev.2007.05.001`.

43 C. L. Staudt and H. Meyerhenke. Engineering High-Performance Community Detection Heuristics for Massive Graphs. In *Proceedings 42nd Conference on Parallel Processing (ICPP'13)*, 2013.

44 C. L. Staudt and H. Meyerhenke. Engineering Parallel Algorithms for Community Detection in Massive Networks. *IEEE Trans. on Parallel and Distributed Systems*, 27(1):171–184, 2016. `doi:10.1109/TPDS.2015.2390633`.

45 M. Tasgin and H. Bingol. Community Detection in Complex Networks using Genetic Algorithm. In *ECCS '06: Proc. of the European Conference on Complex Systems*, 2006. `arXiv:cond-mat/0604419`.

46 S. M. Van Dongen. *Graph Clustering by Flow Simulation*. PhD thesis, Utrecht University, 2001.

47 D. Wagner and F. Wagner. Between Min Cut and Graph Bisection. In *Proceedings of the 18th International Symposium on Mathematical Foundations of Computer Science*, pages 744–750. Springer, 1993.

48 Y. Xu, V. Olman, and D. Xu. Clustering Gene Expression Data using a Graph-Theoretic Approach: an Application of Minimum Spanning Trees. *Bioinformatics*, 18(4):536–545, 2002.