X Congresso Brasileiro de Inteligência Computacional (CBIC'2011), 8 a 11 de Novembro de 2011, Fortaleza, Ceará

A MULTI-THREAD GRASP/VND FOR THE CLUSTER EDITING PROBLEM

L. de O. Bastos, L. S. Ochi, F. Protti
Instituto de Computação
Universidade Federal Fluminense
Rua Passo da Pátria 156, São Domingos, Niterói
{lbastos,satoru,fabio}@ic.uff.br

Resumo – O problema de Edição de Clusters é definido como se segue: dado como entrada um grafo não-direcionado e sem laços $G = (V, E)$, pela adição e/ou remoção de arestas em G, este deve ser transformado num grafo de *clusters*, isto é, numa união de cliques disjuntas. Foi provado que o problema de Edição de Clusters é NP-Completo. Além disso, o problema modela várias aplicações práticas nas áreas de processamento de imagens, biologia computacional, entre outras. Este trabalho propõe um novo algoritmo GRASP/VND *multi-thread* para solucionar o problema heuristicamente, enquanto que uma formulação matemática é utilizada para solucionar o problema de forma exata. Além disso, um método para aplicação seletiva de busca local é utilizado para permitir a solução de instâncias muito grandes num tempo computacional razoável. Testes computacionais utilizando instancias da literatura mostraram que o algoritmo proposto encontra as soluções ótimas na grande maioria dos casos e, quando não o faz, obtém soluções muito próximas dos seus respectivos ótimos. Os tempos computacionais, por sua vez, se revelaram pequenos em comparação com os tempos de outros métodos da literatura.

Palavras-chave – Inteligência computacional, metaheuristicas, otimização combinatória, algoritmos paralelos, Edição de Clusters.

Abstract – The Cluster Editing problem is defined as follows: given as input an undirected loopless graph $G = (V, E)$, by adding edges to G and/or removing edges from G, it must be transformed into a cluster graph, that is, an union of disjoint cliques. The Cluster Editing problem was proved to be NP-complete. This problem models several practical applications in the fields of image processing, computational biology and more. This work proposes a new multi-thread GRASP/VND algorithm to solve the problem heuristically, while a mathematical formulation of the problem is used to exactly solve it. Furthermore, a selective local search method is used to find the solution of very large instances within a reasonable computational time. Tests on benchmark instances showed that the proposed algorithm obtains optimum solutions, in many cases, or very close to optimum ones. Besides that, the computational times are very small in comparison with other techniques found in literature.

Keywords – Computational intelligence, metaheuristics, combinatorial optimization, parallel algorithms, Cluster Editing.

1 INTRODUCTION

The Cluster Editing problem is defined as follows: Let $G = (V, E)$ be an undirected loopless graph, and for every pair (u, v) with $u, v \in V, u \neq v$ there is a cost function $s(u, v) \rightarrow \mathbb{R}$ such that $s(u, v) > 0$ if $(u, v) \in E$, and $s(u, v) < 0$ if $(u, v) \notin E$. The objective is to transform G into a disjoint union of cliques by adding and/or removing edges with minimum cost. If the cost function assigns value 1 for all pairs (u, v) then we have a special case of the Cluster Editing problem called Unweighted Cluster Editing problem.

The Cluster Editing problem was firstly studied in 1979 by Gupta and Palit [1]. Since then, several other authors studied the problem in many different contexts. Some of them proposed exact methods [2–5] while others relied on heuristics to solve the problem [3, 4, 6–8].

The Cluster Editing problem was proved to be NP-Complete in [9]. A mathematical formulation of the CLUSTER EDITING problem as an Integer Linear Program (ILP) was proposed in 1989 by Grötschel and Wakabayashi [10]. Later, in 2004, Charikar proposed a 4-approximation algorithm for a very similar and intuitive formulation called MinDisAgree [11].

This paper introduces a new multi-core GRASP/VND metaheuristic designed to solve the Cluster Editing problem. For the construction phase, two new greedy heuristics are proposed. For the local search phase, three new neighborhoods are also used. Finally, the algorithm is parallelized to make use of the multi-core CPUs widely available today. The objective is to reduce the computational time, within a fixed number of iterations, by increasing the number of threads used. Furthermore, it is expected that, within a computational time limit, increasing the number of threads improves solution quality.

The remainder of this work is organized as follows. Section 2 contains definitions and notation used throughout the paper. Section 3 details all aspects of the proposed GRASP/VND algorithm. The computational experiments and results are described in Section 4. Final considerations are done in Section 5.

2 Preliminaries and Notation

Let $G = (V, E)$ be a connected, undirected loopless graph. The vertex set will be denoted by $V = \{v_1, ..., v_n\}$ with $n = |V|$ and the edge set by $E = \{e_1, ..., e_m\}$ with $m = |E|$. We also define the set $\overline{E}$ as the complement of E. An edge $e = (u, v)$ is called *positive* if $e \in E$ and *negative* if $e \in \overline{E}$. A cost is associated to every edge by the cost function $s(e) \in \mathbb{R}$ or $s(u, v) \in \mathbb{R}$ such that $s(e) > 0$ for positive edges e and $s(e) < 0$ for negative edges e. A positive edge may also be represented as e^+ or $(u, v)^+$, while a negative edge as e^- or $(u, v)^-$.

Let a cost function be also defined over a a set of edges $s(u, C) = \sum_{v \in C \setminus \{u\}} |s(u, v)|$. Here $s(u, C)$ is the cost of all edges, positive and negative, connecting u to the set of vertexes C, for $C \subseteq V$. To calculate the cost for the positive edges only, we define the function $s^+(u, C) = \sum_{v \in C \setminus \{u\}, s(u,v) > 0} s(u, v)$. And for the negative cost, we define the function $s^-(u, C) = \sum_{v \in C \setminus \{u\}, s(u,v) < 0} s(u, v)$.

An edition of G is defined as either an edge addition operation or an edge deletion operation. Addition operations act on negative edges turning them positive, while deletion operations turn positive edges into negative ones. These operations do not change the absolute cost associated to the modified edge, but only its sign.

Let $N^+(v, C) = \{u \in C \mid s(u, v) > 0\}$, for $C \subseteq V$, denote the set of positive neighbors of v relative to the subset of vertexes C. When $C = V$, $N^+(v, C)$ is simply written $N^+(v)$. Let also $N^-(v, C) = \{u \in C \mid s(u, v) < 0\}$ be the set of negative neighbors of v relative to the subset of vertexes C. Again, when $C = V$, $N^-(v, C)$ is simply written $N^-(v)$.

We define cost functions over a set of vertexes. Let $s^-(C) = \sum_{u,v \in C, s(u,v) < 0} |s(u, v)|$ be the negative cost of set C, which corresponds to the sum of the absolute costs of all negative edges connecting a pair of vertexes in C. Let also $s^+(C) = \sum_{u \in C, v \in V \setminus C, s(u,v) > 0} s(u, v)/2$ be the positive cost of C. The total cost of C is then given by $s(C) = s^-(C) + s^+(C)$. If C is a cluster in a given solution S then the cost of S is the sum of the costs of all its clusters. This corresponds exactly to the cost of adding and removing edges to transform G into S. The size of a solution S is also defined as the number of its clusters and is denoted by $|S|$.

3 GRASP/VND META-HEURISTIC

GRASP is a multi-start metaheuristic for combinatorial problems, in which each iteration consists basically of two phases: construction and local search. The construction phase builds a feasible solution, whose neighborhood is investigated until a local minimum is found during the local search phase. The best overall solution is kept as the result [12].

The algorithm proposed here is composed by a construction engine and a local search engine. The construction engine, as its name suggests, manages the implementation of the construction phase while the local search engine manages the application of the local search methodology. In this work the VND metaheuristic [12] is employed within GRASP local search phase, giving rise to the GRASP/VND hybrid metaheuristic.

In the remainder of this section, we describe in detail the phases of the proposed GRASP/VND, i.e., the construction and local search phases as well as the sequential and parallel approaches employed.

3.1 CONSTRUCTION PHASE

The construction phase builds, step-by-step, a feasible solution for the Cluster Editing problem. The evaluation of a feasible solution is given by the sum of the costs of all edges changed in order to transform the input graph into a solution graph. This is carried out in $O(n^2)$ time complexity. Once an infeasible solution means an incomplete solution, there is no way to determine in polynomial time how far a particular infeasible solution is from the optimal solution. Also, it is not possible to compare different infeasible solutions. Because of that, only feasible solutions are allowed as the final product in the construction phase.

For the construction phase we developed two greedy heuristics: Relative Neighborhood (RN) and Vertex Agglomeration (VA). The aim of these algorithms is to generate distinct, and relatively good, solutions as fast as possible.

3.1.1 RELATIVE NEIGHBORHOOD (RN)

The idea behind the RN heuristic is to find vertexes with highest positive neighborhoods relatively to other vertexes. The purpose is to use these vertexes to originate initial solution clusters. In addition, positive and negative neighborhoods are used to complete the clusters in order to minimize the cost of negative edges inside them as well as of positive edges interconnecting distinct clusters.

After calculating $s(u, N^+(u))$ for all vertexes $u \in V$ a Candidate List (CL) is formed by sorting all vertexes in decreasing order of $s(u, N^+(u))$. Then, the RN heuristic selects the first disjoint k vertexes in CL to be cluster seeds, i.e., to be the first element in a new cluster. Being disjoint is important for the candidate vertexes because the neighbors of a candidate vertex will likely be in the same cluster as well as their relative neighborhoods.

After the definition of the solution clusters, a partial solution $S = (V', G')$ is defined, as well as the set of remaining vertexes given by $V_r = w \in V, w \notin V'$. Next, the RN heuristic will randomly pick a vertex $w \in V_r$ and add it to one of the existing clusters of S based on the value of $RN(w, C_i) = s(w, N^+(w, C_i)) - s(w, N^-(w, C_i))$ where $C_i \subset V'$ is a cluster. The value of $RN(w, C_i)$ represents the cost of all positive edges connecting w to C_i minus the cost of all negative edges connecting w to C_i. The vertex w will move to the cluster C_i with maximum value $RN(w, C_i)$. This process continues until $V' = V$. At this

moment, the k clusters will lead S to feasibility since all positive edges connecting distinct clusters are removed and all negative edges inside a cluster are inserted.

By randomly picking remaining vertexes in V_r while building the solution, the RN heuristic allows diversification, that is, the production of different solutions. Additional diversification is obtained by randomly selecting k. Since k is an input parameter to RN, this task is performed by the construction engine (see Section 3.1.3).

The time complexity of RN Heuristic is $O(n^2)$.

3.1.2 VERTEX AGGLOMERATION (VA)

The algorithm starts by selecting k random vertexes to form clusters containing one vertex each, and, in so doing, forms a partial solution $S = (V', G')$. Let V_r be the set of remaining vertexes, i.e., $V_r = w \in V \backslash V'$. Next, it agglomerates the remaining vertexes $w \in V_r$ by adding them, one at a time, to existing clusters. The selection of a candidate vertex from the remaining vertexes set V_r is random. Once a vertex w is taken from V_r, the value $s(w, N^+(w, C_i))$ is calculated for all existing clusters C_i. The vertex is then added to the cluster with maximum value $s(w, N^+(w, C_i))$. This process continues until V_r becomes empty. Note that, while RN heuristic relies on the value of RN to allocate vertexes to clusters, VA heuristic uses only the value $s(w, N^+(w, C_i))$.

As in the RN algorithm, the set of k clusters correspond to a solution to the problem and its evaluation is the sum of the costs of all modified edges. The time complexity of the VA heuristic is also $O(n^2)$.

3.1.3 CONSTRUCTION ENGINE

Since the GRASP construction phase consists of generating a single solution, a construction engine was developed in order to select, at each iteration, a construction algorithm. This selection can be random or cyclic. Its also possible to dynamically keep track of the effectiveness of each construction algorithm, based on solution quality, and select the best algorithm.

Another important feature of the construction engine is to provide a good value of k for the construction algorithms. The strategy for selecting k intends not only to allow some diversification, but also make a choice of k with a reasonable quality. For this aim, the construction engine keeps track of the number of clusters of the incumbent solution. Thus, it is possible to define a range for generating k with a good probability of balance between diversification and quality.

The pseudo-code for the construction engine is shown in Algorithm 1. The algorithm starts by getting the size of the incumbent solution $k_{incumbent}$, the input graph G and the set of construction algorithms C (step 1). In step 2 a construction algorithm is selected. The value of k is defined through steps 3 to 5. Although omitted in pseudo-code, in the first iteration (when there is no incumbent solution) we set $k_{min} \leftarrow 1$ and $k_{max} \leftarrow n$. Finally, the solution is constructed in step 6 and returned as output in step 7.

Algorithm 1 Construction Engine Pseudo-code

1: get $k_{incumbent}, G, C$ as input
2: select randomly $constructor \in C$
3: $k_{min} \leftarrow max(k_{incumbent} - sqrt(n), 1)$
4: $k_{max} \leftarrow min(k_{incumbent} + sqrt(n), n)$
5: $k \leftarrow random(k_{min}, k_{max})$
6: $S \leftarrow constructor(G, k)$
7: return S as output

3.2 LOCAL SEARCH PHASE

After a solution is constructed, a local search phase should be executed as an attempt to improve the initial solution. A local search phase needs the definition of, at least, one search neighborhood. The definition of this neighborhood is crucial for the performance of the local search phase. Moreover, the definition of several neighborhoods are mostly better than defining only a single search neighborhood. Therefore, three neighborhoods are proposed here: Cluster Split (CS), Empty Cluster (EC) and Vertex Move (VM). The ideas behind these neighborhoods are detailed in the following sections.

3.2.1 CLUSTER SPLIT

The objective of the Cluster Split (CS) neighborhood is to identify a cluster C with high negative cost $s^-(C)$ and try to split them in two clusters in order to improve overall solution quality. The negative cost of a cluster is the sum of the costs of all negative edges connecting a pair of vertexes in that cluster.

The first task performed by the CS heuristic, after getting the initial solution S as input, is to select a cluster C to analyze. To accomplish this, the CS heuristic calculates $s^-(C)$ for all clusters in S and chooses the one with maximum value $s^-(C)$, say C'. Next, the values $s^+(u, C')$ are calculated for all $u \in C'$. These vertexes are listed in decreasing order of values $s^+(u, C')$ in a candidate list. So, the vertex $u \in C'$ in the candidate list's first position and the next vertex v not adjacent to u are selected

to be new clusters C_u, C_v. Finally, the remaining vertexes w in the candidate list are allocated to new clusters based on values $s^+(w, C_u)$ and $s^+(w, C_v)$, that is, w will be allocated to the cluster with maximum positive connectivity. The old cluster is then eliminated and the new solution evaluated. The time complexity of CS neighborhood is $O(n^2)$.

3.2.2 EMPTY CLUSTER

The idea of the Empty Cluster (EC) neighborhood is to locate a cluster C with high extra-cluster negative neighborhood cost and, then, try to eliminate it by reallocating its vertexes to other clusters. The negative neighborhood of a cluster C is given by $N^-(C, V \backslash C) = \cup_{w \in C} N^-(w, V \backslash C)$. Thus, calculating $s^-(N^-(C, V \backslash C))$ for all clusters is enough to identify the candidate cluster for deflation. Once the candidate cluster C is selected, a vertex $w \in C$ is moved to another cluster C_i such that the value of $s^+(w, C_i)$ is maximum among all clusters except C. This process terminates when $C = \emptyset$.

After emptying the candidate cluster, the resulting solution is evaluated. The time complexity of EC neighborhood is $O(n^2)$.

3.2.3 VERTEX MOVE

The Vertex Move (VM) neighborhood tries to polish good solutions by moving vertexes to other clusters, one at a time. A vertex u is moved from its original cluster to the cluster that maximally improves solution quality. Once the time complexity of evaluating the solution is $O(n^2)$, it is computationally very expensive to evaluate the solution after every move. So, an alternative evaluation for moving a vertex w from cluster C_w to C_i is done by calculating $s^+(w, C_w) - s^-(w, C_w) - s^+(w, C_i) + s^-(w, C_i)$. This evaluation is carried out in $O(n)$ time complexity. Once there are $|S|$ clusters in the solution, the step of moving one vertex is done in $O(n \times |S|)$ time complexity. After trying to move all vertexes the VM neighborhood stops. The complete VM neighborhood's work is done in $O(n^2 \times |S|)$ time complexity.

3.2.4 LOCAL SEARCH ENGINE AND VND

Besides the benefits of defining different neighborhoods, it is also very important changing the neighborhood used while conducting the local search, as it may yield better results than using always the same neighborhood.

The above-mentioned local search engine is the module responsible for selecting a search strategy and to benefit from the different neighborhoods available.

A simple search strategy is to apply a neighborhood repeatedly until no improvement is detected, and then switch to the next neighborhood. The order of application of the neighborhoods may be random.

Instead of using this simple search strategy, it is interesting to explore the search strategies of the Variable Neighborhood Descent (VND) metaheuristic. Results reported in literature show improvements of the hybrid GRASP/VND over the pure GRASP [12]. The basic idea of VND is to systematically change the neighborhood within a local search. Changing a neighborhood in VND can be done in three different ways: (i) deterministic; (ii) stochastic; (iii) both deterministic and stochastic.

In this work the local search follows the third VND schema, that is, deterministic and stochastic. The Algorithm 2 illustrates the pseudo-code for the local search methodology used here. The algorithm starts by getting a solution S from the construction phase and the set of neighborhoods N (step 1). In step 2 the set of neighborhoods is randomized. This is important because it may yield better results by modifying the sequence in which the neighborhoods are changed at each iteration. Steps 4 to 12 perform the VND itself: a new solution S' is produced in step 5 by applying the current neighborhood N_k. If S' is better than the incumbent solution S, S is updated and the neighborhood index k is restarted. Otherwise, k is incremented to allow the next neighborhood to be selected. This process stops when all the neighborhoods have been selected and no solution update is performed. The incumbent solution is returned as the output in step 13.

Algorithm 2 VND pseudo-code

```
1:  get S, N as input
2:  randomize N
3:  k ← 1
4:  while k ≤ k_max do
5:     S' ← N_k(S)
6:     if S' is better than S then
7:        k ← 1
8:        S ← S'
9:     else
10:       k ← k + 1
11:    end if
12: end while
13: return S as output
```

A very important task, especially for larger instances, is to avoid applying local search phase on not-promising solutions. Since the computational time spent by VND is much bigger than the time spent in construction phase and increases with n, not

doing so may prevent the solution of larger instances within a reasonable amount of computational time. For instance, a small test done with a test problem of 1051 vertexes showed that the local search phase is more than 500 times slower than construction phase.

The strategy for reducing computational times carried out by the local search engine consists of deciding when to apply VND based on solution quality. Local search will be applied to a solution only if it is better than a target cost given by $s_{target} = s(S_{incumbent}) \times f$ where $s(S_{incumbent})$ is the cost of the incumbent solution and f is a selection factor. Thus, if $f = 0$, no local search is performed and the solution quality will be usually poor. As f increases, more and more local searches are applied increasing the computational time, but generally improving solution quality. A good f value is the one that balances solution quality and computational time.

Algorithm 3 shows the local search engine's pseudo-code. In step 1, the current solution, the incumbent solution, the set of neighborhoods and the selection factor are taken as input. The target cost is calculated in step 2. The current solution is locally searched in step 4 only if its cost is smaller than or equal to the target cost. Finally, the resulting solution is returned in step 8. Clearly, if the current solution is not locally searched then it is returned as the resulting solution (see step 6).

Algorithm 3 Local search engine pseudo-code

1: get $S, S_{incumbent}, N, f$ as input
2: $s_{target} = s(S_{incumbent}) \times f$
3: **if** $s(S) <= s_{target}$ **then**
4: $S' \leftarrow$ vnd(S, N)
5: **else**
6: $S' \leftarrow S$
7: **end if**
8: return S' as output

3.3 SEQUENTIAL GRASP/VND

The sequential GRASP pseudo-code is illustrated by Algorithm 4. Initially, the input graph G, the set of construction algorithms C, the set of neighborhoods N, the selective search parameter f and the maximum number of iterations are taken as input in step 1. Then, the incumbent solution is initialized as the empty set in step 2, and the GRASP iterations are carried out from steps 4 to 11. Note that the construction phase occurs in step 5, while the local search is carried out in step 6. If any improvement occurs, the incumbent solution is updated in step 8. Finally, the incumbent solution is output in step 12.

Algorithm 4 Sequential GRASP pseudo-code

1: get G, C, N, f, it_{max} as input
2: $S_{incumbent} \leftarrow \emptyset$
3: $it_{current} \leftarrow 0$
4: **while** $it_{current} < it_{max}$ **do**
5: $S \leftarrow constructionEngine(\mid S_{incumbent} \mid, G, C)$
6: $S \leftarrow localSearchEngine(S, S_{incumbent}, N, f)$
7: **if** $s(S) < s(S_{incumbent})$ **then**
8: $S_{incumbent} \leftarrow S$
9: **end if**
10: $it_{current} \leftarrow it_{current} + 1$
11: **end while**
12: return $S_{incumbent}$ as output

3.4 MULTI-THREAD APPROACH

The GRASP heuristic has an inherent parallel nature which easily results in effective parallel implementations [13]. As each GRASP iteration is isolated, any information from previous iterations is required. One simple way of parallelization is to equally divide iterations among processors. However, this approach may cause unbalancing since some iterations require more computational power than others. In this case, some sort of communication can be done to redistribute iterations and improve balancing.

In this paper, the objective of parallelization is to get benefit of the multi-core CPUs widely available today. This means that the programming model will benefit of shared memory, since the critical regions are protected. Then, by sharing the incumbent solution, all threads can update only one global best solution, and by sharing an iteration counter variable, no balancing is needed because each thread will update it after performing its job.

The parallel GRASP algorithm pseudo-code is nearly the same as sequential GRASP as illustrated by Algorithm 4. The differences follow. Variables $it_{current}$ and $S_{incumbent}$ are placed in shared memory area, which makes steps 8 and 10 critical

regions. So, only one thread can access these objects at a time. Steps 4 to 11 are performed in parallel by each thread. Before step 4 and after step 11 there is only one thread. Between them there are p threads where $1 \leq p \leq processors$. When $p = 1$ the parallel GRASP algorithm behaves exactly as the sequential version.

4 COMPUTATIONAL RESULTS

In this section, we evaluate the GRASP/VND algorithm on real world test problems. These problems are the real world instances presented in [2] and generated from the COG database [14]. All tests were run on a PC desktop equipped with a 6-core 3.33 GHz Intel Core I7 980X processor and 24 Gbytes of RAM under Windows 7 x64 operating system. The CPLEX 12 solver was used to optimally solve instances through the MinDisAgree ILP formulation [11]. The GRASP/VND code was written in C++ language and compiled with MinGW-w64. The OpenMP API was used for the implementation of the parallel version of GRASP/VND.

4.1 EXPERIMENTS

The main objective of the experiments is to evaluate the GRASP/VND algorithm as well as the efficiency of the parallelization approach. The problem set is composed of a total of 3964 instances. Table 1 shows more details. The first line classifies instances by size and the second line informs the amount of instances having that size. As we can see, most instances are small and have about 3 to 49 vertexes. Even though, there is a reasonable amount of instances of average and large sizes. There are also four very large instances with more than 1400 vertexes (containing 2362, 2054, 3387 and 8836 vertexes).

Table 1: Instances from COG database.

size	3-49	50-99	100-149	150-199	200-249	250-299	300-1400	>1400	total
qt	3453	341	78	22	24	20	22	4	3964

All the tests in this paper were repeated 10 times and all the results, except when explicitly stated, are averages of these 10 repetitions.

Before evaluating the effectiveness of GRASP/VND for all instances, a subset of five instances with 200 to 250 vertexes and known optimum values was chosen to test the scalability factor of the parallel version of the algorithm. The tests were divided in two categories of six tests. First, given 1000 iterations, the number of threads was increased from 1 to 6, for each test, and the average computational times and gaps were recorded. Next, the number of threads was fixed at 6 and the number of iterations increased from 1000 to 6000, 1000 at a time, for each test. The results of these tests are shown in Tables 2 and 3.

Table 2: Results for GRASP/VND running 1000 iterations.

threads	1	2	3	4	5	6
gap %	1.04	1.09	1.01	1.32	1.21	1.12
time (s)	25.72	12.83	8.63	6.44	5.14	4.33
speedup	1.00	2.00	2.98	3.99	5.00	5.94

In Table 2, the first line shows the number of started threads. The column corresponding to one thread shows results for the sequential version of the algorithm. The second line shows the percent gap from the optimal solution, given by $gap = (S_{GRASP/VND} - S_{OPTIMUM}) * 100/S_{OPTIMUM}$ where $S_{GRASP/VND}$ is the average solution cost found by GRASP/VND and $S_{OPTIMUM}$ is the cost of the optimum solution. The third line contains the average time in seconds and the last line shows the speedup factor given by $speedup_t = time_1/time_t$ where t is the corresponding number of threads. These results show a small variation in solution gap when the number of threads changes, but no clear relation is easily established. Furthermore, the absolute difference between the smallest and the largest gap is only 0.28%. By the other hand, the speedup factor is practically linear and one can see that doubling the number of threads reduced the time by half. Therefore, increasing the number of threads is very effective to reduce the computational time and achieve the solution of larger instances in a small amount of time when compared to the sequential version of the algorithm.

Table 3: Results for GRASP/VND with 6 threads.

iterations	1000	2000	3000	4000	5000	6000
gap %	1.12	0.94	0.96	0.73	0.73	0.61
time (s)	4.33	10.27	13.15	17.21	21.52	27.26

In Table 3 all tests were carried out with 6 threads and variable number of iterations. The first line shows the number of iterations, while the second one brings the percent gap from the optimal solution. The third line shows average times in seconds. As we can see, increasing the number of iterations also increases computational time, and a gap reduction follows. However, when the iterations were multiplied by a factor of 6, the computational average gap was divided only by a factor of 2. This shows

that the strategy of increasing the number of iterations in order to improve solution quality has a limited application, due to loss of efficiency.

Once the effectiveness of the parallelization of the GRASP/VND algorithm has been proved, the remaining instances were all tested with the parallel version using 6 threads and a limit of 2000 iterations. Before testing the GRASP/VND algorithm all instances were tested in the CPLEX solver equipped with the MinDisAgree formulation, using the default CPLEX configuration and a time limit of 24 hours. The formulation was able to solve optimally instances up to 300 vertexes. This corresponds to 3938 out of the 3964 instances in the test set. The GRASP/VND results are also compared to the results of the ILP branch and cut approach proposed in [2]. Once the tests were done in different computers the computational times were adjusted according to the performance index calculated by the PassMark software for both processors: AMD Opteron 275 scoring 2498 and Intel Core i7 980X scoring 10604. It is also worth noticing that the tests were performed over raw instance data, while the tests in [2] were done over reduced data.

To effectively test the GRASP/VND algorithm, the instances were divided in three subsets: instances with known optimum, instances with unknown optimum, and very large instances with $n > 1400$. For the first subset, an additional stop condition was set: "stop when the target solution is found". The target was defined as the optimum solution.

Table 4 shows the results for the instances solved optimally by MinDisAgree formulation. The first two columns contain the instance size range and the number of instances within the respective range. The following two columns show the MinDisAgree results, i.e., the optimum gap and the computational time in seconds. Next, the results for the GRASP/VND algorithm are shown: column five contains the solution gap and column six the time in seconds. Finally, columns seven to nine shows results for the ILP algorithm. The seventh column contains the amount of instances, according to size range, processed by the ILP algorithm after reduction. Authors claim in [2] that, after reduction, most instances were trivially solved and only the remaining ones were submitted to the ILP algorithm. Next columns contain the solution gap and the computational time. These results show that the MinDisAgree formulation was able to optimally solve all instances up to 299 vertexes within 1 hour and 3 minutes in average. The ILP algorithm took up 2 hours and 51 minutes in average and did not solve two instances. The GRASP/VND algorithm solved optimally all instances up to 100 vertexes. Only 9 out of 3938 instances in Table 4 were not solved optimally by GRASP/VND. Nevertheless, the maximum average computational gap was 0.15%. In terms of computational time, GRASP/VND was more than 400 times faster than MinDisAgree and more than 1000 faster than ILP.

Table 4: Instances with known optimum. [1] One instance was not solved within the time limit of 24 hours.

Instance		MinDisAgree		GRASP/VND		ILP		
size	qt	gap %	time	gap %	time	qt red.	gap %	time
3-49	3453	0.00	0.104	0.00	0.001	297	0.00	0.004
50-99	341	0.00	3.024	0.00	0.013	52	0.00	1.642
100-149	78	0.00	218.833	0.03	0.362	16	0.00	77.739
150-199	22	0.00	1,267.280	0.01	1.588	10	0.00	259.129
200-249	24	0.00	1,137.710	0.07	3.522	9	0.00[1]	3,222.618
250-299	20	0.00	3,733.377	0.15	9.810	2	0.00[1]	10,227.571

The subset of the 22 unknown optimum instances having 300 to 1400 vertexes was solved by GRASP/VND in an average of 11 minutes and 19 seconds. Unfortunately, ILP algorithm authors do not publish optimums for the 9 instances they solved. They, also, do not specify which instances were solved but, once larger instances are much harder to solve exactly, we assume that smaller ones were solved. By analyzing the instance set, it was possible to determine that, without reduction, the ninth instance in terms of size has 391 vertexes. The average computational time for ILP is 13 hours and 25 minutes.

Table 5 shows the results for very large instances. These four instances are not cited in [2]. Their sizes are so large that solving then (even heuristically) is still a challenge. This is an opportunity to use selective search. The first column shows the instance sizes, and next all the pairs of columns show the gap and the computational time for a given selective search factor. Here, the gap is relative to the best solution found in the tests. To better understand these results, when $f = 0$ no local search is applied, hence the computational time is small and the solution quality is poor. When $f = 1$ a local search is done only when the constructive solution is better than the incumbent solution. This condition occurs in the first iteration and then very rarely. The larger the value of f is, more and more local searches are performed improving solution quality but taking a lot of additional time. The results show that a factor of 1.3 yielded an improvement of 16.14% for the first instance, while rising time by a factor of 18.7. The second instance improved 16.63% but the computational time increased 125.3 times. Third and fourth instances could not be solved using $f = 1.3$ within the limit of three hours. The third instance, when solved with $f = 1.2$, improved solution gap 11.52% but within an excessive computational time. Using $f = 1.1$ in this case yielded nearly the same result but with a 87 times faster execution. Finally the gigantic 8836 vertexes instance could only be solved with $f = 1.0$ within the time limit, improving solution quality 4.49% when compared to no local search solution.

5 CONCLUSION

This work studied the Cluster Editing problem. To attack it, a multi-thread GRASP/VND hybrid meta-heuristic was proposed. Constructive and local search heuristics were developed and a VND module was used in the GRASP local search phase. A shared

Table 5: Selective search results on very large instances.

Instance	f = 0.0		f = 1.0		f = 1.1		f = 1.2		f = 1.3	
size	gap %	time	gap %	time	gap %	time	gap %	time	gap %	time
2,054	16.14	7.08	5.74	46.88	4.46	65.27	5.58	85.72	**0.00**	132.23
2,362	16.63	9.25	6.59	46.42	5.28	112.32	2.11	495.92	**0.00**	1,159.10
3,387	11.52	19.43	0.30	140.92	0.16	155.20	**0.00**	13,467.25	-	-
8,836	4.49	157.02	**0.00**	13,939.18	-	-	-	-	-	-

memory model was utilized to parallelize the algorithm. Benchmark results showed that the parallel GRASP/VND outperformed the sequential version, presenting a practically linear speedup in some cases. Moreover, when tested on literature instances, the proposed algorithm found optimum solutions for 99.77% of the instances up to 300 vertexes in less than 10 seconds in average. Although the algorithm cannot ensure optimality, it was able to solve larger instances hundreds of times faster than exact methods. It also solved very large instances, within a reasonable time, while exact methods were limited by much smaller instances.

As a future work, the improvement of the selective local search strategy as well as the development of new heuristics are suggested as an attempt to reduce computational times and to improve solution quality. This is desirable mainly for larger and harder instances for which the optimum is unknown.

REFERENCES

[1] A. Sen Gupta and A. Palit. "On clique generation using Boolean equations". In *Proceedings of the IEEE*, volume 67, pp. 178–180, 345 East 47 Street, New York. NY 10017., 1979. The lnstitute of Electrical and Electronics Engineers, Inc.

[2] S. Böcker, S. Briesemeister and G. Klau. "Exact Algorithms for Cluster Editing: Evaluation and Experiments". *Algorithmica*, pp. 1–19, 2009. 10.1007/s00453-009-9339-7.

[3] S. Rahmann, T. Wittkop, J. Baumbach, M. Martin, A. Truss and S. Böcker. "Exact and heuristic algorithms for weighted cluster editing". In *Computational systems bioinformatics: CSB 2007 Conference Proceedings*, edited by P. Markstein and Y. Xu, volume 6, pp. 391–400, 57 Shelton Street, Covent Garden, London WC2H 9HE, 2007. Imperial College Press.

[4] F. Dehne, M. A. Langston, X. Luo, S. Pitre, P. Shaw and Y. Zhang. "The Cluster Editing Problem: Implementations and Experiments". *Lecture Notes in Computer Science*, vol. 4169, pp. 13–24, 2006.

[5] J. Gramm, J. Guo, F. Hüffner and R. Niedermeier. "Graph-Modeled Data Clustering: Exact Algorithms for Clique Generation". *Theory of Computing Systems*, vol. 38, pp. 373–392, 2005. 10.1007/s00224-004-1178-y.

[6] A. Ben-Dor, R. Shamir and Z. Yakhini. "Clustering gene expression patterns". *Journal of Computational Biology*, vol. 6, no. 3/4, pp. 281–297, 1999.

[7] R. Sharan, A. Maron-Katz and R. Shamir. "CLICK and EXPANDER: a system for clustering and visualizing gene expression data". *Bioinformatics*, vol. 19, no. 14, pp. 1787–1799, 2003.

[8] T. Wittkop, J. Baumbach, F. P. Lobo and S. Rahmann. "Large scale clustering of protein sequences with FORCE -A layout based heuristic for weighted cluster editing". *BMC Bioinformatics*, vol. 8:396, 2007.

[9] R. Shamir, R. Sharan and D. Tsur. "Cluster graph modication problems". *Discrete Applied Mathematics*, vol. 144, pp. 173–182, 2004.

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

[11] M. Charikar, V. Guruswami and A. Wirth. "Clustering with Qualitative Information". *Journal of Computer and System Sciences*, vol. 71, pp. 360–383, 2005.

[12] F. W. Glover and G. A. Kochenberger. *Handbook of Metaheuristics (International Series in Operations Research & Management Science)*. Springer, January 2003.

[13] P. Pardalos, L. Pitsoulis and M. Resende. "A parallel GRASP for MAX-SAT problems". In *Applied Parallel Computing Industrial Computation and Optimization*, edited by J. Wasniewski, J. Dongarra, K. Madsen and D. Olesen, volume 1184 of *Lecture Notes in Computer Science*, pp. 575–585. Springer Berlin / Heidelberg, 1996.

[14] R. Tatusov, N. Fedorova, J. Jackson, A. Jacobs, B. Kiryutin, E. Koonin, D. Krylov, R. Mazumder, S. Mekhedov, A. Nikolskaya, B. S. Rao, S. Smirnov, A. Sverdlov, S. Vasudevan, Y. Wolf, J. Yin and D. Natale. "The COG database: an updated version includes eukaryotes". *BMC Bioinformatics*, vol. 4, no. 1, pp. 41, 2003.