© International Journal of Combinatorial Optimization Problems and Informatics, Vol. 3, No. 1, Jan-April 2012, pp. 3-11. ISSN: 2007-1558.

GRASP with Path Relinking for the SumCut Problem

Jesús Sánchez-Oro and Abraham Duarte
Dept. Ciencias de la Computación
Universidad Rey Juan Carlos (Spain)
{jesus.sanchezoro, abraham.duarte}@urjc.es

Abstract. This paper proposes a GRASP algorithm combined with Path Relinking to solve the SumCut minimization problem. In the SumCut problem one is given a graph with n nodes and must label the nodes in a way that each node receives a unique label from the set$\{1, 2, \ldots, n\}$, in order to minimize the sum cut of the generated solution. The SumCut problem is really important in archeology (in seriation tasks) and in genetics, helping in the Human Genome Project. This problem is equivalent to the Profile problem, because a solution for SumCut is reversal solution for Profile problem. Experimental results show that the GRASP and Path Relinking methods presented outperform in terms of average percentage deviation the results from the State of the Art using shorter CPU time.

Keywords: metaheuristic, GRASP, Path Relinking, SumCut.

1. INTRODUCTION

Let $G = (V, E)$ be an undirected graph, where V denotes the set of vertices and E the set of edges. Let $n = |V|$ and $m = |E|$. A layout φ of the vertices of G is a one-to-one mapping from the set V onto the integers $\{1,2, \ldots, n\}$ where each vertex $v \quad V$ has a unique label $\varphi(v) \quad \{1,2, \ldots, n\}$.Given a graph G, and a vertex v placed in position $i = \varphi(v)$, we define the left set $L(i, \varphi, G)$ and the right set $R(i, \varphi, G)$ as:

$$L(i,\varphi,G) = \{u \in V : \varphi(u) \le i\} \tag{1}$$
$$R(i,\varphi,G) = \{u \in V : \varphi(u) > i\} \tag{2}$$

Set $L(i, \varphi, G)$ contains vertices on the left side of position *i*, including itself. On the other hand, set $R(i, \varphi, G)$ stores vertices on the right side of position *i*. Taking into account the definition of these two set, the vertex cut at position *i* of φ, is computed as:

$$\delta(i,\varphi,G) = \left|\{u \in L(i,\varphi,G) : \exists v \in R(i,\varphi,G) : uv \in E\}\right| \tag{3}$$

The SumCut of a layout φ, denoted as $SC(\varphi, G)$, is calculated as the sum of vertices cut of each position. In mathematical terms:

$$SC(\varphi,G) = \sum_{i=1}^{n} \delta(i,\varphi,G) \tag{4}$$

The SumCut minimization problem then consists of minimizing the value of $SC(\varphi, G)$ over all possible layoutsΠ :

$$SC(G) = \min_{\varphi \in \Pi_n} SC(\varphi,G) \tag{5}$$

The Profile problem, and in the same way the SumCut, were proved to be NP-Complete for cobipartite graphs in Lin et al. [1], and they were also proved to be NP-Complete for general graphs in Gibbons et al. [2], Golovach [3] and Yuan and Lin [4].

The SumCut is equivalent to the Profile minimization problem as it was stated in Agrawal et al. [5]. Specifically, the reverse solution of the SumCut corresponds to a solution of the Profile. Both optimization problems have been extensively studied. See for instance [6][7][8]. Practical applications of these problems appear in genetics [9]. The goal of the Human Genome Project consist of sequencing the DNA of humans as well as other species with the target of elucidating the genetic information contained therein. In order to construct a physical map of a large DNA molecule it is necessary to extract clones from it. Then a fingerprint of each clone is obtained. Finally, DNA molecule is reassembled determining how the clones overlap among them. Each clone is a sequence of nucleotides drawn from the set {A,C,T,G}, so the reassembly process consists of permuting a linear layout of a graph.

Received Oct 10, 2011 / Accepted Dec 10, 2011

In Kendall [10] is described an application in archeology, where it is necessary to serialize different artifacts (fossil, hardware, jewels, etc.). The serialization is known in archeology as “seriation” and consists of placing in chronological order different artifacts in the same culture using a relative dating method. Specifically, the practical application is based on the re-arrangement of a matrix, which can be translated on the reordering of a linear layout of a graph.

Reducing the profile of a matrix is a relevant problem in mathematics since it leads to a reduction of the amount of space needed for some storage scheme. On the whole, it achieves an improvement of the performance of several operations such as Choleski factorization of non-singular systems of equations [11]. Recently the profile reduction has been used in new areas like information retrieval to browse hypertext [12].

The Profile problem was originally proposed as a way to reduce the storage space needed to save sparse matrix [13] but it was proved that it is equivalent to the SumCut problem [5]. An important application of the Profile problem arises in clone fingerprinting [9].

Cuthill and McKee proposed the Reverse Cuthill-McKee (RCM) algorithm [14] in order to get the minimum profile of a graph. If we want to get a solution for the SumCut problem it is only need to reverse the solution generated.

Gibbs et al. solve the SumCut problem using a new algorithm based on the RCM [15]. The paper describes three problems of the RCM and presents a new algorithm that solves the described problems.

Lewis also describes a method to re-order sparse matrices in order to reduce their profile [16] using the Gibbs-King algorithm to improve the results from the RCM algorithm.

The previous algorithms are used in Profile and SumCut problems and also in Bandwidth problem, with differences only in the last step: the numbering of the nodes.

Until now, the best heuristic proposed to obtain better solutions in the SumCut problem is presented in Lewis [17]. It uses the Simulated Annealing to reduce the profile of a matrix. The algorithm starts with a previously calculated solution and it improves the solution by using the Simulated Annealing technique. The original solution is calculated using either the RCM algorithm or the Gibbs-King algorithm, and the instances used are a subset of the Harwell-Boeing graphs set.

2. GRASP

The GRASP metaheuristic was developed in the late 1980s [18] and the acronym was coined in Resende et al. [19]. GRASP can be divided into two phases. The first one consists of constructing a solution and then, in the second phase, improves the incumbent solution with a local search procedure. In this section we describe two constructive procedures and two local search methods for the SumCut problem. Figure 1 illustrates the pseudo-code for the GRASP algorithm.

```
begin GRASP(G, nIter)
    bestSolution =∅;
    for i=1:nIter
        solution = construct();
        improve(solution);
        if (SC(bestSolution) < SC(solution))
            bestSolution = solution;
        endif
    endfor
end
```

Figure 1. Pseudo-code for GRASP algorithm

2.1 Construction procedures

We have designed two constructive algorithms C1 and C2 for the SumCut problem. C1 implements a typical GRASP construction where each candidate element is initially evaluated by a greedy function to construct a Restricted Candidate List (RCL) and one element is selected at random from the RCL. C1 assigns the smaller available label to the selected vertex.

The constructive procedure C1 starts by creating the set of unlabelled vertices U (initially U = V) and the set of labelled vertices L = V\U. The constructive procedure labels one vertex in each iteration. The first node u_0 is selected according to its degree. Specifically, the vertex with the minimum degree is selected (ties are broken at random). The vertex u_0 is labeled l_0=1. Then sets U and L are properly updated (i.e, U = U\{u_0} and L= L ∪ {u_0}).

Once the first label l_0 is assigned to vertex u_0, C1 constructs the candidate list CL with the vertices adjacent to u_0, not previously labeled (i.e. v L).Then, all the vertices in the CL are evaluated with a greedy function g. For this problem we propose the following equation:

$$g(v) = |N_L(v)| - |N_U(v)| \tag{6}$$

where $|N_L(v)|$ indicates the number of vertices adjacent to v that has been already labeled and $|N_U(v)|$ is the number of vertices adjacent with vthat has not been labeled yet. The constructive procedure stores in $g_{\min}$ and $g_{\max}$, respectively, the minimum and maximum value calculated with the greedy function. The next step consists of constructing the Restricted Candidate List, RCL with all the unselected vertices in CL having a value of the greedy function greater than or equal to a specified cutoff value, th. In mathematical terms:

$$RCL = \{v \in CL : g(v) > th\} \tag{7}$$

$$\text{Where } th = g_{\min} + \alpha_1 \cdot (g_{\max} - g_{\min}) \tag{8}$$

$$\text{And } g_{\min} = \arg\min_{u \in CL}\{g(u)\} \text{ and } g_{\max} = \arg\max_{u \in CL}\{g(u)\} \tag{9}$$

The C1 procedure randomly selects an element u from the RCL and assigns the next label to it. After that, it updates the CL with the adjacent vertices of u. This method ends when all the vertices are labeled.

We now consider the constructive algorithm C2, based on another strategy where randomization and greedy selection are interchanged. This strategy was introduced in Werneck and Resende [20] and recently used in Werneck and Resende [21], Duarte et al. [22] and Martí et al. [23]. Specifically, in C2 we first randomly choose candidates from the CL (defined as above) and then evaluate them according to the same greedy function. More formally, RCL is constructed by selecting at randomα_2 |CL| elements from CL. Then, the vertex u RCL with the largest value of g is selected to become part of the solution under construction.C1 and C2 have two search parameters, α_1 and α_2.

2.2 Local search procedures

A solution to the SumCut problem can be represented with a permutation. This kind of solution representation has associated two different types of moves: interchange and insertion. The first one consists of swapping the labels of two different vertices. That is, given two verticesu, v V, the operator interchange(u, v) assigns the label φ(u) to vertex v and the label φ(v) to vertex u obtaining a new labeling φ′.

On the other hand, the insertion move consists of inserting a vertex in a different position. That is, given a vertex v and a position i,insertion(v, i) assigns the label i to vertex v (i.e., after the move φ(v) = i) and all the nodes between u and v will change their labels as:

$$\varphi(v_j) = \begin{cases} \varphi(v_{j+1}) \text{ for } i < j \le \varphi(v) \\ \varphi(v_{i-1}) \text{ for } i > j \ge \varphi(v) \end{cases} \tag{10}$$

The local search procedure based on interchanges, LS_Interchange, has two input arguments (i.e., the constructed solution, φ, and the graph, G). This procedure is illustrated in Figure 2. The algorithm performs moves while improving (see steps 2 to 16). Vertices are scanned at random. First of all, the algorithm selects a starting node s (step 4). After that, the local search procedure starts scanning the solution in order starting at position φ(s). In each step a vertex v placed in a position φ(v) is selected (step 6). Then, the vertex u (step

8) is the one placed in position $\varphi(v) + 1$. The remaining vertices are explored in order taking into account that after exploring the vertex in position n, it is explored the vertex in position 1.

The procedure tries to perform the corresponding interchange (step 9). If this move reduces the objective function (step 10), the original solution is updated (step 12). Otherwise, the method performs moves until no further improvement is reached.

```
begin LS_Interchange(φ,G)
    1     improve = true;
    2     while (improve) do
    3       improve = false;
    4       s = random(n);
    5       for i = 1 to n do
    6         v = (φ(s) + i) mod n;
    7         for j = 1 to n do
    8           u = (φ(v) + j) mod n;
    9           φ' = interchange(u,v);
    10          if SC(φ',G) < SC(φ,G) then
    11            improve = true;
    12            φ = φ';
    13          end
    14        end
    15      end
    16    end
end
```

Figure 2. Pseudo-code for Local Search

The local search procedure based on insertions is quite similar. Specifically, instruction in step 6 is substituted by $\varphi' = \text{insert}(u, \varphi(v))$.

The SumCut and Profile are problems in which a simple movement in a solution generated may change the objective function value, because as the solution is based on the sum of the cut of all vertices, each vertex will contribute to the objective function. For that reason the selection of a vertex that will improve the global objective function value is a challenge for solution methods based on heuristic optimization.

3. PATH RELINKING

Path Relinking (PR) was firstly introduced in 1977 [24] and updated in 1998 [25]. This algorithm generates new solutions by exploring trajectories that connect high-quality solutions by starting from one of these solutions, called the initiating solution, and generating a path in the neighborhood space that leads toward the other solutions, called guiding solutions. This is accomplished by selecting moves that introduce attributes contained in the guiding solutions, and incorporating them in an intermediate solution initially originated in the initiating solution. In this section we explore two different adaptations of PR to the SumCut problem.

Let $\varphi_X = (4, 2, 3, 5, 1)$ and $\varphi_Y = (3, 4, 5, 1, 2)$ be two solutions for the SumCut problem. Given these solutions, the PR method operates over the set D of vertices that are not allocated in each solution in the same position. Considering the solutions φ_X and φ_Y the set $D = \{1,2,3,4,5\}$ since all the elements are allocated in different positions. D is thus considered the candidate of vertices to be examined. To create a path from X to Y, we search in X for the vertex u with label $\varphi_X(u) = \varphi_Y(v)$ and perform $\text{interchange}(u, v)$, where each interchange generates a different intermediate solution. Then the vertex v is removed from D and the next vertex is selected from D. PR performs a greedy selection in each step, i.e., PR evaluates all possible movements $\text{interchange}(u, v)$ for all v V and performs the best one in terms of objective function. Figure 3 shows an example of PR execution.

In this paper we propose two different Path Relinking algorithms. Specifically, Static Path Relinking (SPR) and Dynamic Path Relinking (DPR).

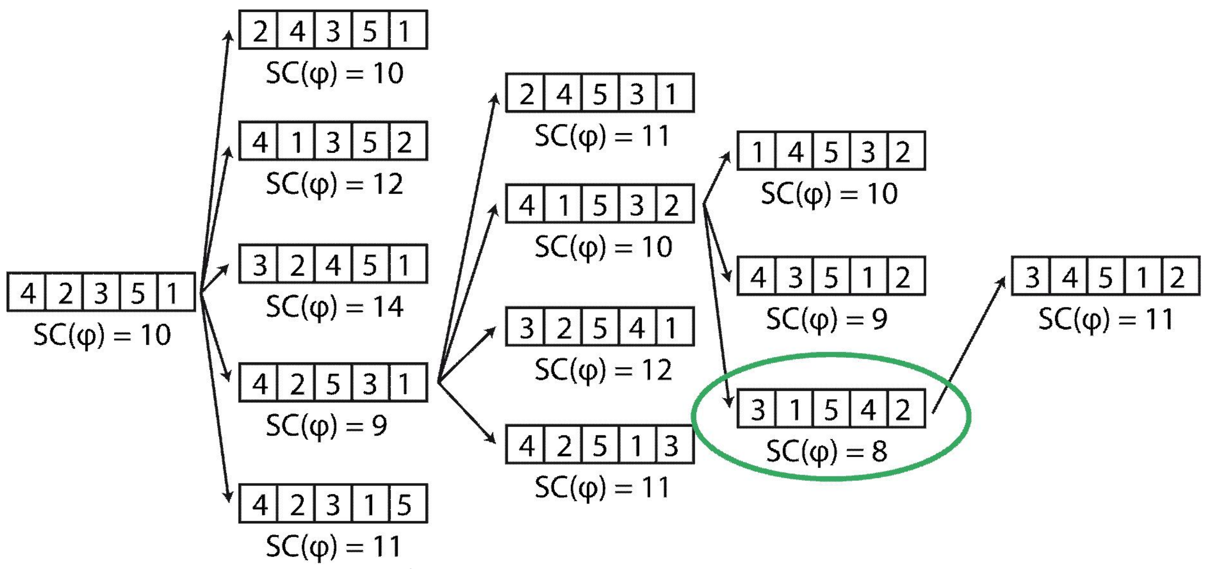

Figure 3. Example of Path Relinking where the best solution in the path is highlighted.

3.1 Static Path Relinking

The first step of Static Path Relinking (SPR) is the creation of the reference set (RefSet). Specifically, the GRASP procedure is executed for maxIter iterations generating a pool p of different solutions ($|p| \leq$ maxIter). The SPR then selects b (with $|RefSet| = b$) solutions from p where b/2 are selected according to its quality (objective function) and the b/2 are selected according to its diversity (distance function). The distance between two solutions is computed as the sum of the differences (in absolute value) between each pair of labels in the corresponding labeling. In mathematical terms:

$$d(\varphi_1, \varphi_2) = \sum_{i \in |\varphi|} \left| \varphi_1(i) - \varphi_2(i) \right| \tag{11}$$

After that, the SPR procedure generates a path for each pair of solutions in the RefSet. The algorithm ends when all the pairs of solutions have been combined. Figure 4 shows the pseudo-code for SPR algorithm.

```
begin SPR(G,p,b)
    refSet = ∅;
    for i=1:p do
      sol = construct();
      improve(sol);
      addIfBest(sol,refSet,b/2);
      addIfDiverse(sol,refSet,b/2);
    endfor
    for each src   refSet do
      for each dst   refSet do
        if (src <> dst) then
          bestCombination = combine(src,dst);
          updateBest(bestCombination);
        endif
      endfor
    endfor
end
```

Figure 4. Pseudo-code for SPR algorithm

After that, the SPR procedure generates a path for each pair of solutions in the RefSet. The algorithm ends when all the pairs of solutions have been combined. Figure 4 shows the pseudo-code for SPR algorithm.

3.2 Dynamic Path Relinking

Dynamic Path Relinking (DPR) generates the RefSet by executing the GRASP and the selecting the first b different solutions. Once the RefSet has been generated, a new solution is created using GRASP algorithm, which will be the initiating solution. Then, a solution is selected at random from the RefSet, which will be the guiding solution. DPR generates a path from the initiating to the guiding solution, returning the best solution found in the path. Finally, DPR tests if the returned solution qualifies to enter in the RefSet. Specifically, to test if a solution should enter into the RefSet it is necessary to check three statements:

1. If the solutions is better than the best in the RefSet (in terms of objective function), sis admitted in the RefSet.
2. If the solutions is worse than the worst solution in the RefSet (in terms of objective function), sis not admitted in the RefSet
3. Otherwise, if the solution s is different enough (in terms of the distance function), sis admitted in the RefSet.

Given solution, φ_1 constructed with the GRASP and a solution φ_2 RefSet we say that these two solutions are different enough, if d (φ_1, φ_2)>dth, where the distance threshold is computed as dth = δ dmax,and dmax is the maximum distance between two solutions, that can be evaluated as follows:

$$d\max = \sum_{1<i<n} \left| i - (n - i + 1) \right| \tag{12}$$

Finally, if the solution is able to enter into the RefSet, it will replace the most similar solution of the RefSet among the set of solutions that are worse than the new one. It is important to remark that "similarity" is evaluated with the distance function described above. Figure 5 shows the pseudo-code for the Dynamic Path Relinking algorithm.

```
begin DPR(G, p, b)
    refSet = ∅;
    while size(refSet) < b do
        sol = construct();
        improve(sol);
        addIfDifferent(sol,refSet);
        p = p – 1;
    endfor
    for i=1:p do
        solution = construct();
        improve(solution);
        refSetSol = selectRandom(refSet);
        bestCombined = combine(solution,refSetSol);
        checkIfEnter(bestCombined,refSet);
    endfor
end
```

Figure 5. Pseudo-code for DPR algorithm

4. COMPUTATIONAL RESULTS

This section describes the computational experiments performed to test the efficiency of the GRASP heuristic. We implemented the methods in Java and the experiments were carried out on an Intel Core 2 Duo E4800 computer running at 3.00 Ghz with 3 GB of RAM.

We used one set of test problems in our experiments. A total of 22 instances were considered. The set of instances are derived from the Harwell-Boeing Sparse Matrix Collection [26]. This collection consists of a set of standard test matrices arising from problems in lineal systems, least squares, and eigenvalue calculations from a wide variety of scientific and engineering disciplines. The problems range from small matrices, used as counter-examples to hypotheses in sparse matrix research, to large matrices arising in applications. Graphs are derived from these matrices as follows. Let M_{ij} denote the element of the i-th row and j-th column of the $n \times n$ sparse matrix M. The corresponding graph has n vertices. Edge (i, j) exists in the graph if and only if $M_{ij} \neq 0$.

The computational experiments are divided into two parts. In the first one, we evaluate the performance of the proposed strategies. We use a subset of the set of instances of 10 representative examples (i.e., different sizes and densities). Then, one time we have identify the best combination of search strategies we compare it with the state-of-the-art.

In the first experiment we compare our 2 proposed constructive methods in order to determine the best constructive procedure. Both procedures construct 100 different solutions, recovering the best one of all constructions. Parameters α_1 and α_2 have been set randomly for each iteration, in order to boost the diversity of the generated solutions. Table 1 shows the comparison between our constructive methods (C1 and C2), and the Cuthill-McKee algorithm (RCM), described in Cuthill and Mckee [14], which is one of the most used algorithms in the SumCut problem, and the Gibbs-King algorithm (GK), described in Lewis [16]and Poole, Stockmeyer and Gibbs[15]. We report for each method the average percentage deviation with respect to the best know solution (Dev.), the number of times that each method matches the best known solution (#Best) and the CPU time (Time) in seconds. Regarding RCM and GK we directly use the values published by the authors.

Table 1. Comparison of constructive methods

	Dev	#Best	Time
C1	25.64%	1	0.33
C2	13.32%	2	0.36
RCM	19.73%	2	*
GK	15.08%	1	*

Table 1 clearly shows that the best constructive procedure is C2 (13.32%) followed by GK (15.08%) method. It is important to realize that that the CPU time of GK and RCM were not published in the corresponding papers.

In our second experiment we study how the local search procedures interact with the constructive methods. Specifically, we compare constructive C1 coupled with local search based on Interchanges, C1+LS1, C1 with local search based on insertions, C1+LS2 and the same for the C2 procedure (i.e., C2+LS1 and C2+LS2).In order to prevent really long running time, all the algorithms are stopped after a maximum time of 1000 seconds. Table 2 shows the performance of the four considered methods considering the same statistics than above.

Table 2. Comparison between local search methods

	Dev	#Best	Time
C1+LS1	13.21%	2	228.52
C1+LS2	13.42%	1	291.87
C2+LS1	8.05%	1	205.97
C2+LS2	8.46%	2	264.33

Table 2 shows that the best algorithm is C2+LS1 in terms of the objective function and CPU time. However, C2+LS2 obtains a larger number of best solutions.

In the next experiment we compare our best GRASP procedures (i.e., GRASP1(C2+LS1) and GRASP2(C2+LS2)) using all the 22 instances.. As we said before, Table 2 shows that the best algorithm is C2+LS1 in terms of the objective function and CPU time, but we have choose to include C2+LS2 because it finds more best solutions than the C2+LS1.

Table 3. Comparison between Local Search and Simulated Annealing

	Dev	#Best	Time
GRASP1	8.02%	2	283.78
GRASP2	8.63%	4	357.36

We can see in Table 3 that although GRASP2 achieves better solutions more times, GRASP1 have a lower average percentage deviation. It is remarkable to say that GRASP1 also achieves a better CPU time than GRASP2.

The main objective of the next experiment is to compare the Static Path Relinking (SPR) and the Dynamic Path Relinking (DPR) with the best method identified in related literature. Specifically, they are compared with the Simulated Annealing, SA, presented in Lewis

[17]. In this case, we have executed both algorithms to improve the results of the GRASP1 algorithm. We have also compared both SPR and DPR with SA, which can be analyzed in Table 4.

Table 4. Comparison between Path Relinking and Simulated Annealing

	Dev	#Best	Time
SPR	1.48%	11	63.22
DPR	1.06%	14	45.57
SA	1.68%	16	179.62

Table 4 shows that our Path Relinking strategies clearly outperforms the best method identified in previous papers. Specifically, SPR and DPR obtain a deviation with respect to the best-known solutions of 1.48% and 1.06%, while SA obtains 1.68%. Additionally, the CPU time of our proposed procedures is considerable shorted than time of the SA procedure. On the other hand, the number of best solutions found by SA compares favorably to our two procedures. Specifically, SA found 16 best solutions (out of 22) while our best procedure obtains 14 (out of 22).

5. CONCLUSIONS

In this paper we present two different constructive methods for the SumCut problem as well as two local searches, deriving into four possible GRASP procedures and two post-optimization strategies based on the Path Relinking methodology. Experimental results show that our two best methods outperform to the previous Simulated Annealing in terms of average percentage deviation and CPU time. On the other hand, Simulated Annealing finds more times the best solution but much of merit seems to be related to GK constructive method.

We propose as future work the efficient computation of the objective function, so we can evaluate a solution faster. This improvement drives the procedure to perform more movements in the same time, which seems to end in a better solution. We are also trying to develop new local search methods in order to move only the more suitable nodes based in the characteristics of the solution.

References

[1] Lin, Y., Liu, Y., Wang, S., Yuan, J.: NP-completeness of the profile problem and the fill-in problem on cobipartite graphs. J.Mathem.Study (1998).

[2] Gibbons, A., Paterson, M., Torán, J., Diaz, J.: The Minsumcut problem. Algorithms and Data Structures (1991).

[3] Golovach, P.: The total vertex separation number of a graph. Diskretnaya Matematika (1997).

[4] Yuan, J., Lin, Y.: Profile minimization problem for matrices and graphs. Acta Mathematicae Applicatae Sinica (1994).

[5] Agrawal, A., Klein, P., Ravi, R.: Ordering problems approximated: single-processor scheduling and interval graph completition. Automata, Languages and Programming (1991).

[6] Penrose, M., Petit, J., Serna, M., Diaz, J.: Convergence theorems for some layout measures on random lattice and random geometric graphs. Journal Combinatorics, Probability and Computing (2000).

[8] Petit, J., Serna, M., Díaz,J.: A Survey of Graph Layout Problems. ACM Computing Surveys (2002).

[7] Petit, J., Serna, M., Trevisan, L., Díaz, J.: Approximating Layout Problems on Random Sparse Graphs. Universidad Politécnica de Valencia, Tech. Report (2001).

[9] Karp, R.: Mapping the Genome: Some Combinatorial Problems Arising in Molecular Biology. STOC'93 Proceedings of the twenty-fifth annual ACM symposium on Theory of computing (1993).

[10] Kendall, R.: Incidence Matrices, Interval Graphs and Seriation in archaeology. Pacific Journal of Mathematics (1969).

[11] Saad, Y.: Iterative Methods for Sparse Linear Systems. PWS Publishing Company (1996).

[12] Botafogo, R.A.: Cluster analysis for hypertext systems. Proceedings of the 16th Annul International ACM-SIGIR Conference on Research and Development in Information Retrieval (1993).

[13] Tewarson, R.: Sparse Matrices. Academic Press, New York (1973).

[14] Cuthill, E., Mckee, J.: Reducing the bandwidth of sparse symmetric matrices. ACM '69 Proceedings of the 1969 24th national conference (1969).

[15] Poole, W., Stockmeyer, P., Gibbs, N.: An Algorithm for Reducing the Bandwidth and Profile of a Sparse Matrix. SIAM Journal on Numerical Analysis (1976).

[16] Lewis, J.: The Gibbs-Poole-Stockmeyer and Gibbs-King Algorithms for Reordering Sparse Matrices. ACM Transactions on Mathematical Software - TOMS (1982).

[18] Resende, M.G.C., Feo, T.A.: A probabilistic heuristic for a computationally di cult set covering problem. Operations Research Letters (1989).

[17] Lewis, R.: Simmulated Annealing for Profile and Fill Reduction. University of British Columbia, Tech. Report (1993).

[19] Resende, M.G.C., Smith, S.H., Feo, T.A.: A greedy randomized adaptive search procedure for maximum independent set. Operations Research (1994).

[20] Werneck, R.F., Resende, M.G.C.: A hybrid heuristic for the p-median problem. Journal of heuristics (2004).

[21] Werneck, R.F., Resende, M.G.C.: A Hybrid Heuristic for the p-Median Problem. Journal of Heuristics (2004).

[22] Duarte, A., Martí, R., Pardo, E.G., Pantrigo, J.J.: Scatter Search for the cutwidth problem. Annals of Operations Research (2010).

[23] Resende, M.G.C., Martí, R., Gallego, M., Duarte, A.: GRASP and Path Relinking for the Max-Min Diversity Problem. Computers and Operations Research (2010).

[24] Glover, F.: Heuristics for integer programming using surrogate constraints. Decision Science, Vol. 8, No. 1 (1977).

[25] Glover, F.: A Template for Scatter Search and Path Relinking. the Third European Conference on Artificial Evolution, London (1998) 3-54.

[26] Matrix Market. [Online]. http://math.nist.gov/MatrixMarket/collections/hb.html