A Learning Optimization Algorithm in Graph Theory

Versatile Search for Extremal Graphs Using a Learning Algorithm ⋆

Gilles Caporossi and Pierre Hansen

GERAD and HEC Montréal (Canada)
{Gilles.Caporossi,Pierre.Hansen}@gerad.ca

Abstract. Using a heuristic optimization module based upon Variable Neighborhood Search (VNS), the system AutoGraphiX's main feature is to find extremal or near extremal graphs, i.e., graphs that minimize or maximize an invariant. From the so obtained graphs, conjectures are found either automatically or interactively. Most of the features of the system relies on the optimization that must be efficient but the variety of problems handled by the system makes the tuning of the optimizer difficult to achieve. We propose a learning algorithm that is trained during the optimization of the problem and provides better results than all the algorithms previously used for that purpose.

Keywords: extremal graphs, learning algorithm, combinatorial optimization.

1 Introduction

A graph G is defined by a set V of vertices and a set E of edges representing pairs of vertices. A graph invariant is a function $I(G)$ that associates a numerical value to each graph $G = (V,E)$ regardless of the way vertices or edges are labelled. Examples of invariants are the number of vertices $|V| = n$, the number of edges $|E| = m$, the maximum distance between two vertices (diameter), the chromatic number (minimum number of colors needed so that each vertex is colored and two adjacent vertices do not share a color). Some more sophisticated invariants are related to spectral graph theory such as the *index* (largest eigenvalue of the adjacency matrix), the *energy* (sum of the absolute values of the eigenvalues of the adjacency matrix). A graph that minimizes or maximizes an invariant (or a function of invariants, which is also an invariant) is called extremal graph. The system AutoGraphiX (AGX) for computer assisted graph theory was developed at GERAD, Montreal. Since 1997, AGX led to the publication of more than 50 papers. The search for extremal graphs is the first goal of AGX and it is an important tool for graph theorists and mathematical chemists as it may be used to handle the following other goals:

- *Find a graph given some constraints*, achieved by the use of Lagrangian relaxation.
- *Refute or strengthen a conjecture*. Suppose a conjecture says that the invariant I_1 is larger than the invariant I_2 ($I_1 \geq I_2$), minimizing $I_1 - I_2$ could provide a counter-example if a negative value is obtained. Whether a counter-example is found or not,

⋆ The authors gratefully acknowledge the support from NSERC (Canada).

Y. Hamadi and M. Schoenauer (Eds.): LION 6, LNCS 7219, pp. 16–30, 2012.
© Springer-Verlag Berlin Heidelberg 2012

looking at the extremal values and the corresponding graphs could help strengthening or correcting the original conjecture.
- *Find conjectures*. Structural or numerical conjectures may be obtained automatically or interactively by analyzing or looking at the extremal graphs obtained.

Based upon the Variable Neighborhood Search metaheuristic (VNS) [16][17], Caporossi and Hansen [9] developped AGX. The extremal graphs obtained by AGX are studied either directly by the researchers or by automated routines that may identify properties of the extremal graphs and deduce conjectures on the problem under study [8][10].

Several graph theorists have used AGX (and the recent AGX2) for study of invariants which most interested them. Applications to mathematics concern spectral graph theory, *i.e.*, the index [11] and the algebraic connectivity [3], as well as several standard graphs invariants [2] and a property of trees [5]. Applications in mathematical chemistry concern the Randić (or connectivity) index [6,7,13,14], the energy [4], indices of irregularity [15] and the HOMO-LUMO gap [12]. This work has led to many extensions by several mathematicians and chemists.

AGX relies on the VNS but also on a large number of transformations used within the search for a local optimum in the Variable Neighborhood Descent (VND) phase of the algorithm. The good performance of the system depends on the user's knowledge to select the correct transformations to use. If the transformations are not appropriate, the optimization will have a poor performance, either because it fails to obtain good solutions or because it takes much too long time. Indeed, choosing a transformation that is too sophisticated will result in a waste of time while a transformation that is too simple will be fast but inefficient. The authors of the system, aware of this problem, proposed in the second version of AGX (called AGX 2) an algorithm that selects automatically its transformations [1], the Adaptive Local Search (ALS). While ALS is a step toward the automation of the selection of the transformations, it cannot as such be considered as a learning algorithm since it is unable to learn on large graphs (its learning is very time consuming on graphs with more than 12 vertices).

In this paper, we propose a new learning algorithm that could replace the original VND used in AGX 1 or the Adaptive Local Search (ALS) of AGX 2. As the ALS, the new Learning Descent (LD) does not require any knowledge in combinatorial optimization and is based upon the concept of transformation matrix. However, its learning capabilities are much more powerful. The next section of the paper describes the various optimization algorithms that have been used to search for extremal graphs. A comparison of the performance of the different algorithms is done in the third section and a short conclusion is drawn at the end of the paper.

2 The Variable Neighborhood Search in AGX

The optimization in AGX is done by VNS which is well suited to handle a wide variety of problems with little tuning, compared to most other methods such as tabu search or genetic algorithms.

Let G be a graph and consider a transformation, for example *move* that consists in removing an edge from G and inserting it in another place on G. This transformation may

be used to define $N(G)$, the neighborhood of G, or set of all graphs that may be constructed from G by the transformation *move*. Such neighborhoods could be extended to a succession of transformations. One thus defines the nested neighborhoods $N^k(G)$, the set of graphs that could be constructed by applying k times the chosen transformation to G. This concept of neighborhoods plays an important role in VNS and the definition of a multitude of these neighborhoods is plainly used in the AGX implementation to handle efficiently a wide variety of different problems that would require different neighborhoods (or transformations) for good results.

In AGX, the standard implementation of VNS is used, alternating Local Search and variable magnitude perturbations as described on *figure 1*.

Initialization:
- Select the neighborhood structure N^k and a stopping condition.
- Let G be an initial (usually random) graph.
- Let G^* denote the best graph obtained to date.

Repeat until condition is met:
- Set $k \leftarrow 1$;
- **Until** $k = k_{max}$**,do:**
  - *(a)* Generate a random graph $G' \in N^k(G)$;
  - *(b)* Apply LS to G'
  - Denote G'' the obtained local optimum $G'' = LS(G')$;
  - *(c)* If G'' is better than G,
    - Let $G^* \leftarrow G''$ and
    - $k \leftarrow 1$
  - otherwise,
    - set $k \leftarrow k+1$.

done

Fig. 1. Rules of Variable Neighborhood Search

2.1 The VND Algorithm in AGX 1

The choice of a good transformation within the local search is a key to success. To add flexibility to the system, different transformations are implemented that could be used one after the other on the same problem. Thus, the Variable Neighborhood Descent is a succession of local searches involving different transformations used for the search. The program performs a local search for each transformation in the list until none of them succeeds. VND could be considered as an extension of local search as it provides a local optimum with respect to each of the transformations used in the search. The VND algorithm is described on figure 2.

While the general VNS parameter k could often be set to the default value, the choice of the list of transformations in VND is much more critical. For instance, if the number of vertices and edges are fixed, any transformation that results in a modification of these numbers will be useless.

A variety of neighborhoods are implemented in AGX to adapt the system to different kinds of problems. Some of these transformations were specially designed to handle special classes of problems (for example, *2-Opt* is well suited for problems with fixed

Initialization:
Select a list of neighborhood structures
$N_l(G)$, $\forall l = 1 \ldots L$, that will be used.
Consider an initial graph G,
set $improved \leftarrow true$.
Until $improved = false$ **do**
    Set $improved = false$
    Set $l = 1$
    Until $l = L$ **do**
        (a) Find the best graph $G' \in N_l(G)$.
        (b) If G' is better than G,
                set $G \leftarrow G'$,
                set $improved \leftarrow true$ and
                return to step *(a)*.
            Otherwise,
                set $l \leftarrow l + 1$;
    done
done

Fig. 2. Rules of Variable Neighborhood Descent

numbers of edges where *simple_move* is very inefficient). The set of transformations used in AGX is described in [9].

To take advantage of the capabilities of AGX, the researcher must have sufficient knowledge in combinatorial optimization, which is not necessarily the case.

2.2 The Adaptive Local Search in AGX 2

The Adaptive Local Search (ALS) may be viewed as meta-transformations that could eventually be used within the VND frame. However, by themselves, ALS replaces most of the transformations available within AGX 1. Each transformation is described as a replacement of an induced subgraph g' of G by another subgraph g". Considering 4 vertices, at most 6 edges could be present in any graph. It is therefore possible to consider up to $2^6 = 64$ labelled subgraphs on 4 vertices. ALS enumerates all the subgraphs g' with 4 vertices in G. It then considers replacing g' by an alternative subgraph g''. As enumerating and evaluating all the alternative subgraphs g'' to replace g' would be very time consuming, replacing g' by g'' will only be evaluated if there are good reasons to believe it is worthwhile. The implementation of this method encodes each subgraph g' or g'' as a label (number) based upon the 64 patterns as follows.

After relabeling its vertices from 1 to 4 by preserving their order, each subgraph g' is characterized by a unique label from 0 to 63 as follows:
pattern 0 (vector = 000000): empty subgraph
pattern 1 (vector = 000001): $E = \{(1,2)\}$
pattern 2 (vector = 000010): $E = \{(1,3)\}$
:
pattern 13 (vector = 001101): $E = \{(1,2),(1,4),(2,3)\}$
:
pattern 63 (vector = 111111): complete subgraph on 4 vertices.

A 64×64 transformation matrix $T = \{t_{ij}\}$ is used to store information on the performance of each possible transformation from pattern i to pattern j.

In ALS, T is a binary matrix indicating whether a transformation t_{ij} from the pattern i to the pattern j was ever found useful.

Based upon this definition of patterns, the principle of the ALS is to use the selected transformations to try to obtain a better graph. Once the graph could no more be improved by the selected transformations (a local optimum is found with respect to the considered transformations), the algorithm searches for transformations that were not considered but that could improve the current solution. For this search, all potential transformations are considered and those that could improve the graph are added to the set of selected transformations (by setting the corresponding $t_{ij} = true$). This step is very time consuming and is only done for small graphs (12 vertices or less). After selection of new transformations the matrix T is updated to take symmetry into account (the same graph g' may correspond to different patterns according to the labeling of the vertices). A formal description of the algorithm is given on *figure 3*.

When working on large graphs, ALS has to be trained before the optimization as this training will never be modified when optimizing large graphs. The training and optimization phases are thus well separated in ALS (for large graphs, steps 2 and 3 of the algorithm are omitted).

2.3 The Learning Descent

As opposed to the ALS algorithm, the LD algorithm performs the training during the optimization phase and always continues learning. The training and optimization phases occurs at the same time.

The LD algorithm on *figure 5* could be described by the following observations:

1. The pertinence of changing g' into g'' (replacing pattern p' by pattern p'') is memorized in a 64×64 matrix T which is initially set to $T = \{t_{ij} = 0\}$.
2. During the optimization, each induced subgraph g' is considered for replacement by any possible alternative subgraph g" but this replacement will not necessarily be evaluated.
3. The probability to test the replacement of pattern i (g') by j (g") is $p = sig(t_{ij}) = \frac{1}{1+e^{-t_{ij}}}$. The initial probability to test a replacement is 50% according to point 1.
4. For any tested transformation, if changing g' (with pattern p') to g'' (with pattern p'') improves the solution, the entry $t_{p',p''}$ of T is increased by δ^+ (and reduced by δ^- otherwise), with $\delta^+ > \delta^-$ because it is more important to use an improving transformation than to avoid a bad one. Also, a good transformation may fail, specially if the graph already has a good performance (here, we use $\delta^+ = 1$ and $\delta^- = 0.1$). The probability to test a transformation increases when it succeeds, but decreases if it does not.

As often used in neural networks, the sigmoid function $sig(x)$ allows the probability to test a transformation to change according to its performance without completely avoiding any transformation (which allows the system to always continue learning). The *figure 4* represents the replacement of *pattern 60* by *pattern 27* on a given graph G for the induced subgraph g' defined by vertices 1, 3, 5 and 6.

```
Step 1: Initialization
Load the last version of the matrix T for the problem under study if it exists initialize T = {t_ij = 0}
otherwise.
Step 2: Find interesting transformations
Set f ← false (this flag indicates that no pattern was added to the list at this iteration).
For each subgraph of the current graph with n' vertices do:
    Let p_i be the corresponding pattern
    for each alternative pattern p_j do:
        if replacing the subgraph p_i by the pattern p_j
         would improve the current solution:
            update the matrix T by setting t_ij ← true
            set f ← true.
     done
done
Step 3: Update T for symmetry
If f = true: Update the matrix T to take symmetry into account:
    for each t_ij = true do:
        for each pattern (i', j') obtained from (i, j)
         by relabelling the vertices do:
            set t_i'j' ← true.
        done
    done
    If f = false:
        Stop; a local optimum is found.
        Save the matrix T
Step 4: Apply Local Search
set improved ← true
while improved = true do:
    set improved ← false
    For each subgraph g' of G on n' vertices do:
        let p_i be the corresponding pattern
        For each alternative pattern p_j:
            if t_ij = true do:
                if replacing p_i by p_j in G improves the
                solution:
                    apply the change
                    set improved ← true
            done
    done
done
Go to Step 2
```

Fig. 3. Rules of the Adaptive Local Search

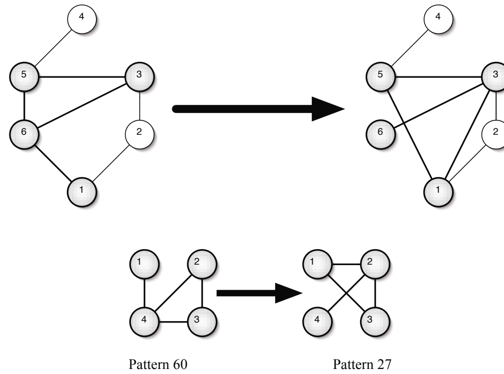


Fig. 4. Illustration of the transformation of G (left) to G'(right)

Note that if the algorithm were restricted to Step 2, it would tend to reduce the probability to use any transformation when good solutions are encountered since few transformations would improve such solutions. To avoid this problem, the matrix T is centered after each local search to an average value $\bar{t} = 0$.

3 Performance Comparison

In order to compare the performance of the different versions of the optimization module in AGX-1 and AGX-2, the AGX-1 moves were added to AGX-2. The various difference in the AGX-1 and AGX-2 program are thus not taken into account, which allows a more realistic comparison of both methods.

3.1 Experiments Description

AGX-1 needs an input from the user for good performance. The program was run with two different settings. The novice setting consists in using all the available neighborhoods for the VND, except those that modify the number of vertices. The expert setting consists in properly chosen neighborhoods. To ensure the reliability of a result stating that new methods are better than older ones, the experimental protocol always has a bias in favor of older methods. The choice of the neighborhoods used in the *expert mode* of AGX 1 is done after results obtained by all the combinations of the 4 transformations (add/remove, move, detour/short cut and 2-Opt) are known. The chosen strategy for *AGX-1-e* (expert strategy) was the one providing the best result in success rate as first criterion, using average value and finally cpu time in case of ties. We noticed that the best performance for a given problem but with different number of vertices was not necessarily due to the same scheme. This indicates clearly a bias favorable to *AGX 1-e* since it is difficult that an expert would identify this choice before the tests are done.

Step 1: Initialization
Load the last version of the matrix T for the problem under study if it exists and initialize $T = \{t_{ij} = 0\}$ otherwise.
Step 2: Apply Local Search
set $improved \leftarrow true$
while $improved = true$ **do:**
  set $improved \leftarrow false$
  For each subgraph g' **of** G **on** n' **vertices do:**
    let p_i be the corresponding pattern
    For each alternative pattern p_j
    (corresponding to g''**):**
      let x be an uniform 0-1random number.
      if $x \leq sig(t_{ij})$ **do:**
        if replacing g' by g'' in G improves the
        solution:
          apply the change
          set $improved \leftarrow true$
          set $t_{ij} = t_{ij} + \delta^{+}$.
        otherwise:
          set $t_{ij} = t_{ij} - \delta^{-}$.
      done
  done
done
Step 3: Scale the matrix T
  Let $\bar{t}$ be the average value of the terms $t_{ij} \neq 0$.
  For each $t_{ij} \neq 0$:
    set $t_{ij} = t_{ij} - \bar{t}$.
Step 4: Save the matrix T for future usage

Fig. 5. Rules of the Learning Descent

AGX-2 is designed to run without knowledge from the user and was also run in the three following modes : the ”complete” mode *AGX 2-c* in which all the possible transformations involving 4 vertices are considered, the adaptive mode *AGX 2-als* in which the software chooses the useful neighborhoods from 5 runs on the same problem restricted to 10 and 12 vertices, and the learning descent mode *AGX 2-ld* in which the probability to use a given transformation is adjusted during the optimization.

To avoid any bias favorable to AGX 2-ld, the training on small graphs for AGX 2-als is done prior to the experiments and the training time (time needed by the system to identify which transformations to use) is not considered. On the opposite, the results of training during the optimization with AGX 2-ld was systematically erased after each optimization, so that the benefits from previous runs is avoided, which is a bias against AGX 2-ld.

The five optimization schemes compared are noted as follows.

- *1 b :* version using the all neighborhoods available in AGX-1,
- *1 e :* version using the best combination of neighborhoods available in AGX-1 (expert mode),

- *2 c :* version using AGX-2 considering all the possible transformations on 4 vertices (with the statistics matrix $T = \{t_{ij} = true\}$),
- *2 als :* version using AGX-2 and the adaptive local search,
- *2 ld :* version using AGX-2 and the learning descent.

The performance of these various algorithms was tested against 12 different and representative problems. The problems used are described in the following section. Each problem was solved 10 times for graphs with 13, 15 and 20 vertices with each of the 5 optimization scheme. In all cases, the total CPU time allowed was 300 seconds and the program was stopped if no improvement was encountered for 60 consecutive seconds. To reduce bias due to the implementation, all the tests were achieved with the same program in which the different versions of the optimizer are available thru parameters. All these tests were achieved on a Sun with 2 Dual Core AMD Opteron(tm) Processor 275 (2.2 GHz) with 4Go RAM memory running Linux CentOS-4 operating system. The performance of each strategy was measured in 3 ways. The first part (Average Z) of each table indicates the average value obtained among the 10 runs; the second (Success) indicates the number of times the best value was obtained and the last (CPU Time) indicates the average CPU time required to reach the best value found. If the best value was never attained by a given strategy, a ”-” is displayed.

3.2 Results Analysis

Among the 360 instances tested, *AGX 2-ld* succeeded 275 times (76.4 %), which is the best performance, followed by *AGX 2-als* with 255 successes (70.8 %), *AGX 2-c* with 229 (63.6 %) successes and *AGX 1-e* with 176 (48.8 %) successes, followed by *AGX 1-b* which was only successful 61 times (16.9 %).

Regardless of the problem under study, AGX 1-b (which was often before AGX 2 was developed) shows very poor performance. Even with the experimental bias, the AGX 2 strategies are far better, first because they involve a wide range of transformations that were not implemented in AGX 1, and also because the VND used with AGX 1 spends some time trying to unsuccessfully optimize with a transformation before switching to the next, which is not the case in any of the AGX 2 local search scheme. If one should compare the strategies that do not involve any knowledge of the problem or of the optimization procedure (*1 b, 2 als, 2 c and 2 ld*), which is the most important for the novice point of view, AGX 2 with its stochastic local search seems to be the best choice.

The ”Best Z” line on the tables indicates the best obtained value during the whole experiment, which may be (but is not always) the best possible value. The *Min* or *Max* at the top left of each table recalls wether the objective is to be minimized or maximized.

- **Problem 1 :** Minimize the energy E among trees, where $E = \sum_{i=1}^{n} |\lambda_i|$ is the sum of the absolute values of the eigenvalues of the adjacency matrix of the graph. For this problem, the number of edges is fixed by the number of vertices ; not all the transformations are therefore needed. The optimal solution to this problem, is a path and the corresponding value of the energy is $E = 2\sqrt{n-1}$.

- **Problem 2 :** Minimize the value of the Randić index [18] among bicyclic connected graphs. The Randić index is defined as $\chi = \sum_{(ij)\in E} \frac{1}{\sqrt{d_i d_j}}$ where d_i is the degree of vertex i. The solution to this problem is a star to which are added two edges adjacent to the same vertex. The optimal value is $\chi = \frac{n-4}{\sqrt{n-1}} + \frac{2}{\sqrt{2n-2}} + \frac{1}{\sqrt{3n-3}}$.
- **Problem 3 :** Same as *problem 2*, except that the objective function is maximized instead of being minimized. The optimal solution is known to be two cycles sharing an edge or two cycles joined by an edge and the corresponding value is $\chi = \frac{4}{\sqrt{6}} + \frac{3n-10}{6}$.
- **Problem 4 :** Minimize the sum of the average degree of the graph $\bar{d}$ and the proximity p, where $\bar{d} = \sum_{i=1}^{n} \frac{d_i}{n}$ and $p = \frac{1}{n-1} \min_i(\sum_{j=1}^{n} d_{ij})$ where d_{ij} is the distance between the vertices i and j. In this case, the search space is only restricted by the connexity constraint. The optimal solution to the problem is a star and the optimal value is $Z = n+1-\frac{2}{n}$.
- **Problem 5 :** Maximize the size of the maximum stable set, the maximum number of vertices to select such that no selected vertex is adjacent to another selected vertex, among connected graphs with number of edges equals twice the number of vertices ($m = 2n$).
- **Problem 6 :** Maximize the matching number, the number of edges to be selected such that no vertex is adjacent to two selected edges, among the same set of graphs as *problem 5* ($m = 2n$). There are lots of graphs maximizing this invariant under the given constraints, but the maximal value cannot exceed $Z = \lfloor \frac{n}{2} \rfloor$, which is attained here.
- **Problem 7 :** Maximize the *index*, value of the largest eigenvalue of the adjacency matrix, among the same set of graphs as *problem 5* or *problem 6* ($m = 2n$).
- **Problem 8 :** This problem is the same as *problem 7* except that the objective function is to be minimized instead of being maximized.
- **Problem 9 :** Minimize the *index* among trees.
- **Problem 10 :** Maximize the *diameter*, maximum distance between two vertices of the graph, among the same set of graphs as *problems 5, 6, 7* and *8*.
- **Problem 11 :** Maximize the *diameter* among connected graphs graphs with $m \geq 2n$.
- **Problem 12 :** Maximize the size of the maximum stable set among connected graphs graphs with $m \geq 2n$.

Problems 1 to 10 (except problem 4) have a fixed number of edges and of vertices. This corresponds to the problems we encounter most often, particularly for parametric analysis on the order and the size of the graph. In problems 5,6,7,8 and 10, the number of edges is fixed to twice the number of vertices. The number of graphs satisfying this condition is rather large, which makes the combinatorial aspect of the optimization important. Such problems are interesting benchmarks for the optimization routine.

The Problems 5 and 12 are NP-Complete. These two problems provide information on the capability of various strategies to handle problems which are more time consuming. AGX 1-b completely fails, and AGX 2-c is not very efficient either; this is because they are among all the two strategies that perform a large number of useless computations of the objective function.

Table 1. Results for problem 1 (the graphic represents the number of successes)

Min	Average Z			Success			CPU Time		
n	**13**	**15**	**20**	**13**	**15**	**20**	**13**	**15**	**20**
1 b	9.36	12.63	15.69	5	1	0	55.1	50	-
1 e	6.93	7.48	11.60	10	10	1	17.2	47.7	121.8
2 c	6.93	7.48	8.72	10	10	10	0	0	39.2
2 als	6.93	7.48	8.72	10	10	10	0	0	0
2 ld	6.93	7.48	8.72	10	10	10	0.5	0.9	2.4
Best Z	6.93	7.48	8.72						

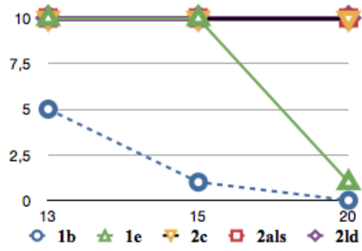


Table 2. Results for problem 2

Min	Average Z			Success			CPU Time		
n	**13**	**15**	**20**	**13**	**15**	**20**	**13**	**15**	**20**
1 b	4.56	5.74	6.95	4	1	0	76.3	99	-
1 e	3.99	4.29	5.53	10	10	2	26.3	78.3	149
2 c	3.99	4.29	4.94	10	10	10	5.4	11.2	71.1
2 als	3.99	4.29	4.94	10	10	10	0.4	0.9	3.7
2 ld	3.99	4.29	4.94	10	10	10	0.5	0.8	2.9
Best Z	3.99	4.29	4.94						

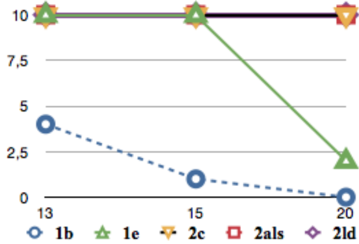


Table 3. Results for problem 3

Max	Average Z			Success			CPU Time		
n	**13**	**15**	**20**	**13**	**15**	**20**	**13**	**15**	**20**
1 b	6.02	6.09	7.67	0	0	0	-	-	-
1 e	6.47	7.46	9.88	10	9	1	22.7	0	0
2 c	6.47	7.47	9.97	10	10	10	4.3	14.1	92.3
2 als	6.47	7.47	9.97	10	10	10	0.3	0.6	3.6
2 ld	6.47	7.47	9.97	10	10	10	0.7	1.3	5.4
Best Z	6.47	7.47	9.97						

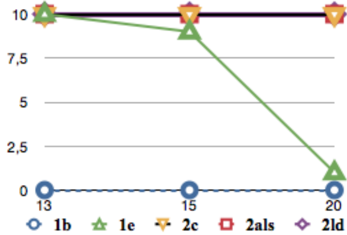


Table 4. Results for problem 4

Min	Average Z			Success			CPU Time		
n	**13**	**15**	**20**	**13**	**15**	**20**	**13**	**15**	**20**
1 b	2.85	2.89	3.26	10	7	0	18	35.9	-
1 e	2.85	2.87	2.95	10	9	4	0.7	25.4	33.4
2 c	2.85	2.87	2.9	10	10	10	4.4	10.7	82.4
2 als	2.85	2.87	2.9	10	10	10	0.6	1.4	6.6
2 ld	2.85	2.87	2.9	10	10	10	0.7	1.2	6.2
Best Z	2.85	2.87	2.9						

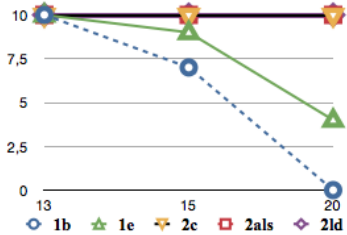


Table 5. Results for problem 5

Max	Average Z			Success			CPU Time		
n	**13**	**15**	**20**	**13**	**15**	**20**	**13**	**15**	**20**
1 b	7.2	8.6	12.1	0	0	0	-	-	-
1 e	8.4	9.4	12.8	5	2	2	24	57.3	5.7
2 c	8.2	9.4	11.5	3	0	0	21.8	-	-
2 als	8.3	9.8	12.7	3	1	2	2.9	10.3	35.4
2 ld	8.7	10	13.1	7	2	4	24.4	39.8	38.9
Best Z	9	11	14						

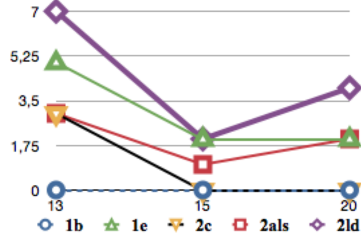


Table 6. Results for problem 6

Max	Average Z			Success			CPU Time		
n	**13**	**15**	**20**	**13**	**15**	**20**	**13**	**15**	**20**
1 b	6	7	8.6	10	10	2	0	0.3	48.5
1 e	6	7	10	10	10	10	0.1	0	0
2 c	6	7	9.8	10	10	8	0	0.5	53.6
2 als	6	7	10	10	10	10	0	0	4.3
2 ld	6	7	10	10	10	10	0	0.1	4.3
Best Z	6	7	10						

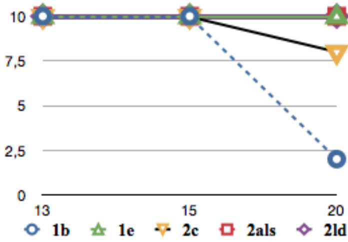


Table 7. Results for problem 7

Max	Average Z			Success			CPU Time		
n	**13**	**15**	**20**	**13**	**15**	**20**	**13**	**15**	**20**
1 b	4.92	5.28	5.91	0	0	0	-	-	-
1 e	5.91	6.2	6.57	1	0	0	100.9	-	-
2 c	5.92	6.18	6.59	10	4	0	27	84.3	-
2 als	5.92	6.29	6.95	10	10	0	3.5	15.8	-
2 ld	5.92	6.28	6.99	10	9	2	6.4	20.9	53.3
Best Z	5.92	6.29	7.25						

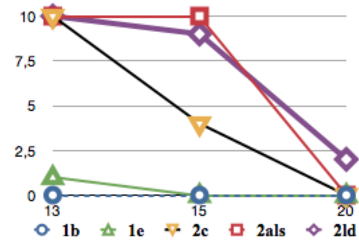


Table 8. Results for problem 8

Min	Average Z			Success			CPU Time		
n	**13**	**15**	**20**	**13**	**15**	**20**	**13**	**15**	**20**
1 b	4.86	5.13	5.65	0	0	0	-	-	-
1 e	4.02	4.03	4.13	4	1	0	0	0	-
2 c	4	4	4.3	10	10	0	11.2	48.3	-
2 als	4	4	4.01	10	10	7	2.7	8.5	131.2
2 ld	4	4	4	10	10	10	3.3	9.6	74.1
Best Z	4	4	4						

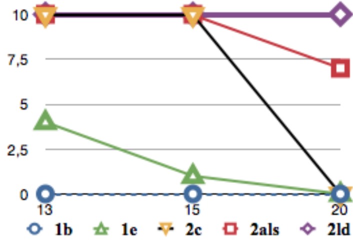


Table 9. Results for problem 9

Min	Average Z			Success			CPU Time		
n	**13**	**15**	**20**	**13**	**15**	**20**	**13**	**15**	**20**
1 b	2.35	2.68	3.26	0	0	0	-	-	-
1 e	1.95	1.96	2.02	10	10	3	0	18.6	52.6
2 c	1.95	1.96	1.98	10	10	10	4.3	12.3	82.3
2 als	1.95	1.96	1.98	10	10	10	0.4	0.7	3.6
2 ld	1.95	1.96	1.98	10	10	10	0.6	1.1	3.6
Best Z	1.95	1.96	1.98						

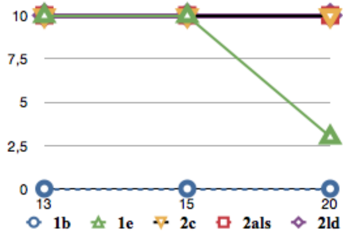


Table 10. Results for problem 10

Max	Average Z			Success			CPU Time		
n	**13**	**15**	**20**	**13**	**15**	**20**	**13**	**15**	**20**
1 b	5.6	5	4.9	1	0	0	84.3	-	-
1 e	6.1	6.7	7	2	0	0	35.4	-	-
2 c	5.9	6.8	10.7	1	0	1	6.9	-	66.6
2 als	6.3	7.5	11.4	3	1	5	19.3	8.6	30.5
2 ld	6.8	7.7	11.1	8	2	4	26.3	45.4	15.6
Best Z	7	9	12						

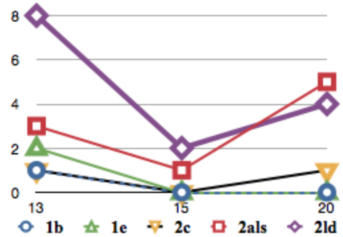


Table 11. Results for problem 11

Max	Average Z			Success			CPU Time		
n	**13**	**15**	**20**	**13**	**15**	**20**	**13**	**15**	**20**
1 b	5.2	3.1	3	0	0	10	-	-	0.5
1 e	6.5	4.3	3	6	1	10	26.3	57.1	0.1
2 c	4.8	3.5	3	0	0	10	-	-	3.4
2 als	4.7	3.6	3	1	0	10	20.7	-	6.7
2 ld	6.1	4.2	3	6	0	10	70.5	-	6.1
Best Z	7	8	3						

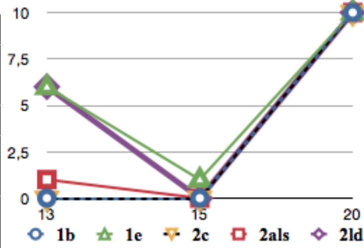
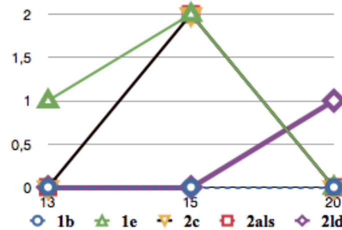


Table 12. Results for problem 12

Max	Average Z			Success			CPU Time		
n	**13**	**15**	**20**	**13**	**15**	**20**	**13**	**15**	**20**
1 b	6.5	6.9	6.4	0	0	0	-	-	-
1 e	8.4	8.6	7.7	1	2	0	81.5	42.7	-
2 c	8.1	8.9	7.9	0	2	0	-	58	-
2 als	8.2	8.9	8.1	0	2	0	-	42.6	-
2 ld	8.4	8.1	8.5	0	0	1	-	-	6.4
Best Z	10	10	10						



We notice that problems 10 and 11 have a low success rate. The problem 11 on 20 vertices was completely missed by all the strategies as the best found value is only 3 even if the optimal value should be at least as good as that of problem 10. This is an important phenomenon for researchers using AGX; even if the diameter is easy to compute, it has really bad properties from the optimization point of view. Depending on the current graph, changing the value of the diameter by 1 may involve a large number of transformations and no strategy is powerful enough in this case. This phenomena is called *plateau* and some practical ways to handle it are described in [9,1].

4 Conclusion

From these experiments, we first notice that the *VNS-LD* and *VNS-ALS* algorithms are clearly more efficient even if the performance of *VNS-VND e* is overestimated. The good performance of the *VNS-LD* may be due to the wide range of transformations implicitly considered in *VNS-LD* that are not implemented in *VNS-VND*. However, this is not the only reason because some tests were achieved always using all the transformations available in LD (by artificially setting the probability to select any of them to 1) and the best results were only found 269 times. Indeed, one of the forces of LD and ALS are that unlike VND which wastes some time trying to unsuccessfully optimize with a transformation before switching to the next one, LD uses any interesting transformation and concentrates on the most performing ones. Overall, *VNS-LD* performs better than *VNS-ALS* even if this last algorithm's training time is not considered here (training was achieved before the tests on smaller graphs); furthermore, *VNS-LD*'s training was erased between two tests and the system had to learn from scratch at each run.

References

1. Aouchiche, M., Bonnefoy, J.-M., Fidahoussen, A., Caporossi, G., Hansen, P., Lacheré, J., Monhait, A.: Variable neighborhood search for extremal graphs. 14. the autographix 2 system. In: Liberti, L., Maculan, N. (eds.) Global Optimization. From Theory to Implementation. Springer Science, NewYork (2006)
2. Aouchiche, M., Caporossi, G., Hansen, P.: Variable neighborhood search for extremal graphs 8. variations on graffiti 105. Congressus Numerantium 148, 129–144 (2001)
3. Belhaiza, S., Abreu, N.M.M., Hansen, P., Oliveira, C.S.: Variable neighborhood search for extremal graphs. 11. bounds on algebraic connectivity. In: Hertz, A., Avis, D., Marcotte, O. (eds.) Graph Theory and Combinatorial Optimization. Springer, New York (2005)
4. Caporossi, G., Cvetković, D., Gutman, I., Hansen, P.: Variable neighborhood search for extremal graphs. 2. finding graphs with extremal energy. Journal of Chemical Information and Computer Sciences 39, 984–996 (1999)
5. Caporossi, G., Dobrynin, A.A., Gutman, I., Hansen, P.: Trees with palindromic Hosoya polynomials. Graph Theory Notes of New York 37, 10–16 (1999)
6. Caporossi, G., Gutman, I., Hansen, P.: Variable neighborhood search for extremal graphs: 4. chemical trees with extremal connectivity index. Computers and Chemistry 23, 469–477 (1999)
7. Caporossi, G., Gutman, I., Hansen, P., Pavlović, L.: Graphs with maximum connectivity index. Computational Biology and Chemistry 27, 85–90 (2003)

8. Caporossi, G., Hansen, P.: Finding Relations in Polynomial Time. In: Proceedings of the XVI International Joint Conference on Artificial Intelligence, pp. 780–785 (1999)
9. Caporossi, G., Hansen, P.: Variable neighborhood search for extremal graphs. 1. the autographix system. Discrete Math. 212, 29–44 (2000)
10. Caporossi, G., Hansen, P.: Variable neighborhood search for extremal graphs. v. three ways to automate finding conjectures. Discrete Math. 276, 81–94 (2004)
11. Cvetković, D., Simić, S., Caporossi, G., Hansen, P.: Variable neighborhood search for extremal graphs. iii. on the largest eigenvalue of color-constrained trees. Linear and Multilinear Algebra 49, 143–160 (2001)
12. Fowler, P.W., Hansen, P., Caporossi, G., Soncini, A.: Variable neighborhood search for extremal graph. 7. polyenes with maximum homo-lumo gap. Chemical Physics Letters 342, 105–112 (2001)
13. Gutman, I., Miljković, O., Caporossi, G., Hansen, P.: Alkanes with small and large randić connectivity indices. Chemical Physics Letters 306, 366–372 (1999)
14. Hansen, P., Mélot, H.: Variable neighborhood search for extremal graphs. 6. analyzing bounds for the connectivity index. Journal of Chemical Information and Computer Sciences 43, 1–14 (2003)
15. Hansen, P., Mélot, H.: Variable neighborhood search for extremal graphs. 9. bounding the irregularity of a graph. In: Fajtlowicz, S., Fowler, P., Hansen, P., Janowitz, M., Roberts, F. (eds.) Graphs and Discovery. American Mathematical Society, Providence (2005)
16. Hansen, P., Mladenović, N.: Variable neighborhood search: Principles and applications. European J. Oper. Res. 130, 449–467 (2001)
17. Mladenović, N., Hansen, P.: Variable neighborhood search. Comput. Oper. Res. 24, 1097–1100 (1997)
18. Randić, M.: On characterization of molecular branching. Journal of the American Chemical Society 97, 6609–6615 (1975)