Exploiting Separators for Guiding VNS

Samir Loudni and Mathieu Fontaine and Patrice Boizumault

Université de Caen Basse-Normandie, UMR 6072 GREYC, F-14032 Caen, France
CNRS, UMR 6072 GREYC, F-14032 Caen, France

Abstract

This paper presents two extensions for DGVNS (Decomposition Guided VNS) method, that exploit both the *graph of clusters* and *separators* between these clusters, to efficiently guide the exploration of large neighborhoods in VNS. Experiments performed on challenging instances of the tagSNP selection problem show the appropriateness and the efficiency of our approach.

Keywords: VNS Hybrid, Tree decomposition, Separators, Cost Function Network.

1 Introduction

Tree decomposition, introduced by Robertson and Seymour [7], aims to decompose a problem into subproblems (called clusters) constituting an acyclic graph. Each cluster corresponds to a subset of variables that are strongly connected. For large problems that exhibit a highly structured constraints graph, exploiting such structural properties will lead these problems to be solved more efficiently (see [8,1] for complete search methods).

In a recent paper [2] we have introduced DGVNS (Decomposition Guided VNS), a new approach that exploits the *graph of clusters* obtained from a tree decomposition of the constraints graph, to efficiently guide the exploration of large neighborhoods in VNS. In this paper, we propose two extensions for DGVNS, noted SGVNS (Separator-Guided VNS) and ISGVNS (Intensified SGVNS),

that exploits both the graph of clusters and the *separators* between these clusters. Experimental results over challenging instances of the tagSNP selection problem show that (i) our two approaches clearly outperform `VNS/LDS+CP` [6], and (ii) `ISGVNS` is very effective compared to `DGVNS` and `SGVNS`.

2 Definitions and Notations

A Cost Function Network (CFN) is a pair (X, W) where $X = \{x_1, \ldots, x_n\}$ is a set of n variables (with a maximum domain size d) and W is a set of e cost functions (see Fig. 3). Each variable $x_i \in X$ has a finite domain D_i of values that can be assigned to it. A value a in D_i is denoted (x_i, a). For a set of variables $S \subseteq X$, D^S denotes the cartesian product of the domains of the variables in S. A *complete* assignment t=$(a_1, ..., a_n)$ is an assignment of all variables; on the contrary, it will be called a *partial* assignment. For a given complete assignment t, $t[S]$ denotes the projection of t over S. A cost function $w_S \in W$, with scope $S \subseteq X$, is a function $w_S : D^S \mapsto [0, k_\top]$ where , $k_\top$ is a maximum integer cost (finite or not) used to represent forbidden assignments (expressing hard constraints). Costs are combined using the bounded addition defined by $\alpha \oplus \beta = \max(k_\top, \alpha + \beta)$. Solving a CFN consists in finding a complete assignment t minimizing $\oplus_{w_S \in W} w_S(t[S])$.

Definition 2.1 A tree decomposition of a connected CFN is a pair (C_T, T) where $T = (I, A)$ is a tree with nodes set I and edges set A and $C_T = \{C_i \mid i \in I\}$ is a family of subsets of X (called *clusters* [7]) such that: (i) $\cup_{i \in I} C_i = X$, (ii) $\forall\, w_S \in W$, $\exists\ C_i \in C_T$ s.t. $S \subseteq C_i$, (iii) $\forall\, i, j, k \in I$, if j is on the path from i to k in T, then $C_i \cap C_k \subseteq C_j$.

Definition 2.2 The intersection of two clusters C_i and C_j is called a *separator*, and noted $sep(C_i, C_j)$.

Definition 2.3 A graph of clusters for a tree decomposition (C_T, T) is an undirected graph $G = (C_T, E)$ that has a vertex for each cluster $C_i \in C_T$, and there is an edge $(C_i, C_j) \in E$ when $sep(C_i, C_j) \neq \emptyset$.

3 Decomposition Guided VNS

`DGVNS` (Decomposition Guided `VNS` [2]) extends `VNDS` [4], by exploiting the graph of clusters in order to build neighborhood structures enabling a better diversification. `DGVNS` uses neighborhood structures $N_{k,i}$, where k is the neighborhood dimension and C_i is the cluster the variables will be selected in. Fig. 1 (left) depicts the pseudo-code of `DGVNS`. It starts from an initial solution S which is randomly generated (function `genInitSol`, line 5). A subset of k

```
Function DGVNS(X, P, k_init, k_max, δ_max);
(1)  begin
(2)      let G be the constraints graph of P;
(3)      let (C_T, T) a tree decomposition of G;
(4)      let C_T = {C_1, C_2, ..., C_p};
(5)      S ← genInitSol();
(6)      k ← k_init, i ← 1;
(7)      while (k < k_max) ∧ notTimeOut do
(8)          C_s ← CompleteCluster(C_i);
(9)          X_un ← Hneighborhood(C_s, N_{k,i}, S);
(10)         A ← S\{(x_i, a) | x_i ∈ X_un};
(11)         S' ← Rebuild(A, X_un, δ_max, S);
(12)         ChangeNeighborDGVNS(S, S', k, i)
(13)     endwhile
(14)     return S
(15) end
```

```
Function ISGVNS(X, P, k_init, k_max, δ_max);
(1)  begin
(2)      let G be the constraints graph of P;
(3)      let (C_T, T) a tree decomposition of G;
(4)      let C_T = {C_1, C_2, ..., C_p};
(5)      S ← genInitSol();
(6)      k ← k_init, i ← 1;
(7)      T_List ← ∅, P_List ← ∅;
(8)      while (k < k_max) ∧ notTimeOut do
(9)          C_s ← CompleteCluster(C_i);
(10)         X_un ← Hneighborhood(C_s, N_{k,i}, S);
(11)         A ← S\{(x_i, a) | x_i ∈ X_un};
(12)         S' ← Rebuild(A, X_un, δ_max, S);
(13)         ChangeNeighborISGVNS(S, S', k, i, C_s)
(14)     endwhile
(15)     return S
(16) end
```

Fig. 1. Pseudo-codes of DGVNS (left) and ISGVNS (right).

variables C_s is randomly selected among conflicted ones by the neighborhood heuristic Hneighborhood (line 9). A partial assignment $\mathcal{A}$ is generated from the current solution S by unassigning the k selected variables (line 10). Then, unassigned variables are rebuilt (line 11) by a partial tree search LDS [5] combined with Constraint Propagation (see [6] for more details). The search stops when the maximal dimension size k_{max} is reached or the $TimeOut$ (line 7).

DGVNS **favors moves on regions that are closely linked.** The concept of cluster embodies this criterion, because of its size, and by the strong connection of the variables it contains. The k variables to be unassigned are selected in a same cluster C_i. If $(k > |C_i|)$, then the set of potential variables C_s is completed by adding the clusters C_j adjacent to C_i in order to take into account the topology of the graph of clusters (function CompleteCluster (line 8)). So, the neighborhood structure $N_{k,i}$ is constituted by the set of all subsets of k variables among C_s (line 9).

The aim of diversification is to sample a large number of different regions, in order to explore the whole search space, and to locate the region containing the global optimum. To perform a better diversification, DGVNS considers successively all the C_i (see procedure ChangeNeighborDGVNS, Fig. 2). Let p be the total number of clusters in C_T, $succ$ a successor function [1], and $N_{k,i}$ the current neighborhood structure: if the rebuild step finds a better solution S' in the neighborhood of S (line 2), then S' becomes the current solution (line 3), k is reset to k_{init} (line 4), and the next cluster is considered (line 5). Otherwise, DGVNS looks for improvements in $N_{(k+1),succ(i)}$ (neighborhood structure where $(k+1)$ variables of C_s will be unassigned (line 7)).

[1] $succ(i) = i + 1$ if $i < p$, otherwise $succ(p) = 1$.

First, diversification performed by moving from cluster C_i to cluster $C_{succ(i)}$ is necessary. Experiments we performed have shown that remaining in the same cluster leads to lower improvements: selecting a new cluster enables to visit new parts of the search space. Second, when a local minimum is got in the current neighborhood, moving from k to $(k+1)$ will also provide some diversification by enlarging the neighborhood size.

4 Exploiting Separators to Guide VNS

Variables occurring in a separator $sep(C_i, C_j)$ constitute "key-points" of the problem because their re-assignments (in the rebuild-step) will directly impact the variables of both C_i and C_j. `SGVNS` (Separator-Guided `VNS`) exploits **variables occurring in separators** in order **to guide the diversification effort** towards clusters that are more likely to lead to larger improvements.

Let S' be a new solution better than the current solution S and C_i be the current cluster. Let V_c be the set of all the variables that have been re-assigned to obtain S'. Let C_w be the set of clusters C_j such that $sep(C_i, C_j)$ shares at least one variable with V_c (except those of C_s [2]). `DGVNS` performs diversification by considering $C_{succ(i)}$ as the next cluster (see Section 3). `SGVNS` performs diversification by considering successively the clusters $C_j \in C_w$. These clusters are more appropriate because they contain at least one re-assigned variable in their separators and therefore are more likely to lead to larger improvements.

`ISGVNS` (Intensified `SGVNS`) aims to intensify the exploration around the re-assigned variables. To achieve this goal, a *propagation list* endowed with a dynamic *tabu list* are used. The first list manages the set of candidate clusters to be examined after each improvement, while the second list ensures that variables involved in the selection of these candidate clusters (i.e. variables of V_c) will not be considered in $N_{k,i}$ by the function `Hneighborhood`. Fig. 2 (right) depicts the pseudo-code of procedure `ChangeNeighborISGVNS`. It uses a propagation list P_{List} and a dynamic tabu list T_{List} of size L.

The intensification effort of `ISGVNS` is performed using a propagation list. As for `SGVNS`, V_c is the set of all variables that have been re-assigned (line 4), and C_w is the set of clusters C_j having at least one re-assigned variable in a separator (line 5). Contrary to `SGVNS`, each cluster $C_j \in C_w$ is added to P_{List} (line 7), and each variable $x \in V_c$ is made tabu for the next L iterations (line 9). The value of L is set to the size of P_{List}. This prevents re-assigning x until all the clusters $C_j \in C_w$ of P_{List} have been considered. Finally, the next cluster to be considered is selected from P_{List}, if it is not empty (lines 13 and 14). Otherwise, the successor of C_i in C_T is considered (line 16).

[2] This prevents selecting clusters already considered in the current neighborhood $N_{k,i}$.

```
Procedure ChangeNeighborDGVNS(S, S', k, i);
(1)  begin
(2)     if f(S') < f(S) then
(3)        S ← S';
(4)        k ← k_init;
(5)        i ← succ(i);
(6)     else
(7)        k ← k + 1;
(8)        i ← succ(i)
(9)     endif
(10) end
```

```
Procedure ChangeNeighborISGVNS(S, S', k, i, C_s);
(1)  begin
(2)     if f(S') < f(S) then
(3)        S ← S'; k ← k_init;
(4)        V_c ← {x | S'[x] ≠ S[x]};
(5)        C_w ← {C_j | C_j ∉ C_s ∧ C_j ∩ V_c ≠ ∅};
(6)        for each C_i ∈ C_w do
(7)           insert i in P_List
(8)        endfor
(9)        make tabu each x ∈ V_c ;
(10)    else
(11)       k ← k + 1
(12)    endif
(13)    if P_List is not empty then
(14)       i ← P_List.next
(15)    else
(16)       i ← succ(i)
(17)    endif
(18) end
```

Fig. 2. Steps of neighborhood change for `DGVNS` (left) and `ISGVNS` (right).

`ISGVNS` enforces a balance between intensification and diversification. As long as no improvement is made, `ISGVNS` behaves as `DGVNS` by considering successively all the C_i. But, when a solution is improved, `ISGVNS` switches to an intensification scheme until all clusters of P_{List} have been examined.

5 The tagSNP Selection Problem

The selection of tagSNPs has became a very active area of research in genotyping [3]. A Single Nucleotide Polymorphism (SNP) is a DNA sequence variation occurring when a single nucleotide - `A`, `T`, `C` or `G` - in the genome differs between members of a biological species or paired chromosomes in an individual. SNPs act as biological markers that may help predict risk of developing particular diseases. The tagSNP problem consists in selecting a small subset of SNPs, called tagSNPs, that captures most of the genetic information.

A correlation measure r^2 between any pair of SNPs has been introduced in [3]. A tagSNP p_i is said to be representative of another SNP p_j if p_i and p_j are considered as enough correlated (i.e., $r^2(p_i, p_j) \geq r_0$, where r_0 is a minimum threshold). The tagSNP problem consists in selecting a minimum number of SNPs such that all SNPs are covered. Other criteria [8] can also be considered: (i) maximizing the weighted coverage sum of unselected SNPs and (ii) maximizing the dispersion between selected SNPs (i.e., tagSNPs).

This problem is modeled as a binary CFN (see Section 2). Two variables i_s and i_r are associated to each SNP p_i: i_s is a boolean variable that indicates if p_i is a tagSNP; i_r is a variable representing the tagSNP covering p_i (its

domain is the set of neighbors of p_i together with p_i itself). For each pair of SNPs (p_i, p_j) s.t. $r^2(p_i, p_j) \geq r_0$, the following (hard) constraints are enforced: $i_s \Rightarrow (i_r = p_i)$ and $(i_r = p_j) \Rightarrow j_s$. Such constraints are encoded as binary cost functions (with 0 or $k_\top$ costs). Preferences (i) and (ii) are respectively captured by unary and binary cost functions (see [8] for more details).

6 Experiments

Experimental protocol. We have selected ten challenging instances derived from human chromosome-1-data [3] with r_0=0.5. Seven instances are of medium size, while the three other ones are large ones. Fig. 3 (leftmost column) gives the characteristics of each instance: its name, its number of variables (n), its number of cost functions (e) and its maximum domain size (d). We used the following parameter settings, which are the best values found (see [6] for more details): k_{min}=4, k_{max}=n and δ_{max}=3. *TimeOut* was set to 2 hours (resp. 4 hours) for medium-size (resp. large-size) instances. A set of 50 runs per instance has been performed using an AMD-Opteron with 2.1 GHz CPU and 256 GB of RAM. All search strategies have been implemented in C++. For each instance and each method, Fig. 3 reports the number of successful runs to reach the optimum, the average number of performed iterations, the average CPU time (in seconds) for the successful runs, the average cost and the best cost (between parentheses) for unsuccessful runs. Four methods are compared: `DGVNS`, `SGVNS`, `ISGVNS` and `VNS/LDS+CP` [6] (an instance of `VNDS` [4] which uses neighborhood structures N_k of dimension k). Tree decompositions are built using the *Maximum Cardinality Search* (`MCS`) heuristic [9].

Impact of the tree decomposition. For medium-size instances, `DGVNS`, `SGVNS` and `ISGVNS` clearly outperform `VNS/LDS+CP` (see Fig. 3). The three methods reach the optimum on each run. `VNS/LDS+CP` gets the same success rates on two instances, but is on average 3 times slower. On instances #3792 and #8956, `VNS/LDS+CP` gets successful runs only very few times. It remains less competitive both in terms of successful runs and CPU times on the other instances. On large-size instances, `VNS/LDS+CP` never finds the optimum.

Impact of the separators. Fig. 3 also compares `SGVNS` with `DGVNS`. For medium-size instances, `SGVNS` and `DGVNS` reach the optimum on each run. However, `SGVNS` is faster on three instances, and slower on four instances. For large-size instances, `DGVNS` clearly outperforms `SGVNS`. For instance #17034, the success rate is improved about 16% (from 66% to 82%), the mean deviation from the optimum decreases from 1.43% to 0.63%, and `DGVNS` is 4 times faster than `SGVNS`. These results show that `SGVNS` is competitive for medium-size

[3] `http://www.costfunction.org/benchmark`

Instance	Method	Succ.	Iter	Time	Avg
#3792	DGVNS	50/50	1,248	954	6,359,805
$n = 528,\ d = 59$	SGVNS	50/50	1,240	1,033	6,359,805
$e = 12,084$	ISGVNS	**50/50**	**1,192**	**853**	**6,359,805**
$S^* = 6,359,805$	VNS/LDS+CP	15/50	5,713	2,806	6,359,856
#4449	DGVNS	50/50	1,107	665	5,094,256
$n = 464,\ d = 64$	SGVNS	**50/50**	**1,113**	**661**	**5,094,256**
$e = 12,540$	ISGVNS	50/50	1,0512	675	5,094,256
$S^* = 5,094,256$	VNS/LDS+CP	48/50	5,181	2,616	5,094256
#8956	DGVNS	**50/50**	**1,477**	**4,911**	**6,660,308**
$n = 486,\ d = 106$	SGVNS	50/50	1,513	5,483	6,660,308
$e = 20,832$	ISGVNS	50/50	1,457	4,118	6,660,309
$S^* = 6,660,308$	VNS/LDS+CP	12/50	4776	8,665	6,660327
#9319	DGVNS	50/50	818.7	788	6,477,229
$n = 562,\ d = 58$	SGVNS	**50/50**	**797**	**500**	**6,477,229**
$e = 14,811$	ISGVNS	50/50	923	672	6,477,229
$S^* = 6,477,229$	VNS/LDS+CP	47/50	6,171	2,434	6,477,229
#15757	DGVNS	**50/50**	**511**	**60**	**2,278,611**
$n = 342,\ d = 47$	SGVNS	50/50	525	104	2,278,611
$e = 5,091$	ISGVNS	50/50	527	80	2,278,611
$S^* = 2,278,611$	VNS/LDS+CP	50/50	2,800	229	2,278,611
#16421	DGVNS	50/50	1,688	2,673	3,436,849
$n = 404,\ d = 75$	SGVNS	**50/50**	**1,587**	**2,025**	**3,436,849**
$e = 12,138$	ISGVNS	50/50	4,240	5,863	3,436,849
$S^* = 3,436,849$	VNS/LDS+CP	37/50	5,095	3,146	3,436,924
#16706	DGVNS	50/50	1,167	153	2,632,310
$n = 438,\ d = 30$	SGVNS	50/50	888	159	2,632,310
$e = 6,321$	ISGVNS	**50/50**	**872**	**89**	**2,632,310**
$S^* = 2,632,310$	VNS/LDS+CP	50/50	5,494	629	2,632,310
#10442	DGVNS	**50/50**	**2,264**	**4,552**	**21,591,913**
$n = 908,\ d = 76$	SGVNS	50/50	2,496	7,153	21,591,913
$e = 28,554$	ISGVNS	50/50	2,395	7,291	21,591,913
$S^* = 21,591,913$	VNS/LDS+CP	0/50	5,887	-	22,778,811 (22,490,938)
#14226	DGVNS	46/50	1,802	7,606	25,688,751
$n = 1,058,\ d = 95$	SGVNS	40/50	1,776	7,646	25,805,242
$e = 36,801$	ISGVNS	**50/50**	**1,818**	**9,596**	**25,665,437**
$S^* = 25,665,437$	VNS/LDS+CP	0/50	4,941	-	28,299,904(26,830,579)
#17034	DGVNS	**41/50**	**2,098**	**8,900**	**38,563,232**
$n = 1142,\ d = 123$	SGVNS	33/50	1,825	10,212	38,869,514
$e = 47,967$	ISGVNS	36/50	1,857	10,579	38,746,957
$S^* = 38,318,224$	VNS/LDS+CP	0/50	3,315	-	41,352,709 (39,850,974)

Fig. 3. Comparison on medium-size and large-size instances.

instances, and less effective for large-size ones.

For medium-size instances, ISGVNS is faster than DGVNS on four instances, slower on two instances, and similar on instance #4449. For large-size instances, ISGVNS performs better than DGVNS on instance #14226, and worse on instance #17034. Both methods obtain the same success rates on instance #10442, but DGVNS remains faster. These results show that ISGVNS is more effective than DGVNS, and confirm the importance of exploiting separators.

For medium-size instances, ISGVNS and SGVNS obtain the same success rates. However, ISGVNS is faster than SGVNS on four instances. For large-size

instances, ISGVNS improves the success rates on two instances. These results highlight the importance of the propagation list to enforce a tradeoff between intensification and diversification.

7 Conclusions

We have proposed two extensions of DGVNS that exploit both the graph of clusters and the separators to efficiently guide VNS. Experimental results over challenging instances of the tagSNP selection problem show that SGVNS and ISGVNS clearly outperform VNS/LDS+CP, and ISGVNS is very effective compared to DGVNS and SGVNS. We are currently parallelizing the exploration of clusters.

Acknowledgements. This work is partly supported by the ANR (French Research National Agency) funded project FiCOLOFO ANR-10-BLA-0214.

References

[1] R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artif. Intell.*, 38(3):353–366, 1989.

[2] M. Fontaine, S. Loudni, and P. Boizumault. Guiding VNS with tree decomposition. In *ICTAI*, pages 505–512. IEEE, 2011.

[3] S. Gopalakrishnan and Z.S. Qin. TagSNP selection based on pairwise LD criteria and power analysis in association studies. In *Pacific Symposium on Biocomputing*, pages 511–522. World Scientific, 2006.

[4] P. Hansen, N. Mladenovic, and D. Perez-Brito. Variable neighborhood decomposition search. *Journal of Heuristics*, 7(4):335–350, 2001.

[5] W. D. Harvey and M. L. Ginsberg. Limited discrepancy search. In *IJCAI'95*, pages 607–615, 1995.

[6] S. Loudni and P. Boizumault. Combining VNS with constraint programming for solving anytime optimization problems. *EJOR*, 191:705–735, 2008.

[7] N. Robertson and P.D. Seymour. Graph minors. ii. algorithmic aspects of tree-width. *J. Algorithms*, 7(3):309–322, 1986.

[8] M. Sánchez, D. Allouche, S. de Givry, and T. Schiex. Russian doll search with tree decomposition. In *IJCAI*, pages 603–608, 2009.

[9] R. E. Tarjan and M. Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.*, 13(3):566–579, 1984.