Artificial Intelligence 278 (2020) 103194

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

Variable neighborhood search for graphical model energy minimization

Abdelkader Ouali [a,*], David Allouche [b], Simon de Givry [b,*], Samir Loudni [a], Yahia Lebbah [c,*], Lakhdar Loukil [c], Patrice Boizumault [a]

[a] University of Caen Normandy, CNRS, UMR 6072 GREYC, 14032 Caen, France
[b] INRA, MIA Toulouse, UR-875, 31320 Castanet-Tolosan, France
[c] University of Oran1, Lab. LITIO, 31000 Oran, Algeria

ARTICLE INFO

Article history:
Received 14 October 2018
Received in revised form 3 September 2019
Accepted 27 October 2019
Available online 31 October 2019

Keywords:
Variable neighborhood search
Computational Protein Design
Parallelism
Complete search method
Anytime algorithm
Discrete graphical model
Combinatorial optimization
Markov Random Field
Most Probable Explanation
Cost Function Network

ABSTRACT

Graphical models factorize a global probability distribution/energy function as the product/sum of local functions. A major inference task, known as MAP in Markov Random Fields and MPE in Bayesian Networks, is to find a global assignment of all the variables with maximum a posteriori probability/minimum energy. A usual distinction on MAP solving methods is complete/incomplete, i.e. the ability to prove optimality or not. Most complete methods rely on tree search, while incomplete methods rely on local search. Among them, we study Variable Neighborhood Search (VNS) for graphical models. In this paper, we propose an iterative approach above VNS that uses (partial) tree search inside its local neighborhood exploration. The proposed approach performs several neighborhood explorations of increasing search complexity, by controlling two parameters, the discrepancy limit and the neighborhood size. Thus, optimality of the obtained solutions can be proven when the neighborhood size is maximal and with unbounded tree search. We further propose a parallel version of our method improving its anytime behavior on difficult instances coming from a large graphical model benchmark. Last we experiment on the challenging minimum energy problem found in Computational Protein Design, showing the practical benefit of our parallel version. A solver is available at https://github.com/toulbar2/toulbar2.

1. Introduction

Probabilistic graphical models [1] are formed by variables linked to each other by stochastic relationships. They enable to model complex systems with heterogeneous data and to capture uncertainty. Graphical models have been applied in a wide range of areas such as image analysis, speech recognition, bioinformatics, and ecology.

We focus on models with discrete variables like Markov Random Field and Bayesian Network. Our goal is to find a global assignment of all the variables with maximum a posteriori probability. This optimization task defines an NP-complete problem [2]. Solving methods can be categorized in two groups: exact and local search methods. Exact methods rely on tree search, variable elimination, linear programming, or a combination of them [3–5]. Graph-cut and message-passing algorithms like loopy belief propagation and variational approaches [6–10] are exact only in some particular cases (*e.g.,* binary

* Corresponding authors.
E-mail address: samir.loudni@unicaen.fr (S. Loudni).

https://doi.org/10.1016/j.artint.2019.103194
0004-3702/

image denoising or tree structured problems). Local search methods are stochastic algorithms like Gibbs sampling, Guided Local Search [11,12], and Stochastic Greedy Search [13]. Some of them have theoretical asymptotic proof of convergence, *i.e.,* the optimal solution is guaranteed to be found if infinite time is available. In practice, they may exhibit a better anytime behavior than exact methods on large and difficult problems [14,12,13], *i.e.,* they produce better solutions in less time.

A few attempts have been done to combine exact and local search methods. A simple way is to run sequentially a local search algorithm then tree search, where solutions found by local search will be used as initial upper bounds for branch and bound exact methods. Another approach is to design a local search framework where the neighborhood exploration is performed by tree search in a systematic or non-systematic way as it is done in Large Neighborhood Search (LNS) [15–19] and Variable Neighborhood Search (VNS) [20,21]. VNS/LDS+CP [21] combines a metaheuristic, VNS, with Limited Discrepancy Search (LDS) [22], a partial tree search method (Section 2). We propose in this paper an iterative variant of VNS/LDS+CP, called Unified Decomposition Guided VNS (UDGVNS), adapted to graphical models and able to prove optimality when the neighborhood size is maximal and with unbounded tree search.

Contributions and plan.

1. We introduce UDGVNS, a new iterative approach above DGVNS[1] (Decomposition Guided VNS) unifying complete and incomplete search methods. UDGVNS restores the completeness of DGVNS by applying successive calls with an increasing discrepancy limit.
2. We describe a coarse-grained parallel version called UPDGVNS (for Unified Parallel DGVNS) allowing asynchronous cooperative execution of UDGVNS processes with centralized information exchange as in [24,25]. As for UDGVNS, the parallel release enables one to control the compromise between optimality proof and anytime behavior. Compared to UDGVNS, the parallel version enables us to improve the anytime behavior on difficult instances.
3. We propose a new operator denoted add1/jump for managing the neighborhood size k that exploits the graph of clusters provided by a tree decomposition of the problem.
4. We present an extensive empirical study that includes a wide range of instances coming from various benchmarks (Cost Function Network (CFN), Computer Vision and Pattern Recognition (CVPR), Uncertainty in Artificial Intelligence (UAI) 2008 and Probabilistic Inference Challenge (PIC) 2011) which compares our techniques to state-of-the-art ones. Experimental results show that our approaches offer a good compromise between the number of problems completely solved compared to the quality of the best solution found.
5. We report experiments on the challenging minimum energy problem in Computational Protein Design (CPD). For this aim, we designed new larger instances that are well structured and supposed to be more difficult to solve than those generated in [26]. We show the practical benefit of our approaches compared to TOULBAR2 and FIXBB[2] the simulated annealing algorithm provided by the Rosetta package for CPD.

The paper is organized as follows. Section 2 recalls preliminaries. Section 3 presents UDGVNS. Section 4 describes our parallel version UPDGVNS. Sections 5-6 report experiments we performed. Finally, we conclude and draw some perspectives.

2. Preliminaries

2.1. Graphical model

Definition 1. A *probabilistic graphical model* (or Gibbs distribution) [1] is a triplet $(\mathcal{X}, \mathcal{D}, \mathcal{F})$ with $\mathcal{X} = \{X_1, \ldots, X_n\}$, a set of n random variables, $\mathcal{D} = \{D_1, \ldots, D_n\}$, a set of finite domains of values of maximum size $d = \max_{i=1}^{n} |D_i|$, and $\mathcal{F}$, a set of *potential functions*. Each variable X_i takes values in D_i. An assignment of $\mathcal{X}$ is a set $x = (x_1, \ldots, x_n)$, with $x_i \in D_i$. The set of all possible assignments of $\mathcal{X}$ is denoted $\Delta = \prod_{i=1}^{n} D_i$. Let $A = \{D'_1, \ldots, D'_n\}$ with $D'_i \subseteq D_i$ represent a restricted set of Δ called a *partial assignment*. If S is a subset of $V = \{1, \ldots, n\}$, X_S, x_S and Δ_S are respectively the subset of random variables $\{X_i, i \in S\}$, the assignment $(x_i, i \in S)$ obtained from x, and the set of all possible assignments of X_S. Given a set $\mathcal{S}$ of partitions of V, the set $\mathcal{F} = \{f_S\}_{S \in \mathcal{S}}$ of maps from Δ_S to $\mathbb{R}^+$ is said to factorize a joint probability distribution $\mathbb{P}$ iff:

$$\mathbb{P}(x) = \frac{1}{Z} \prod_{f_S \in \mathcal{F}} f_S(x_S) \tag{1}$$

where $Z = \sum_{x \in \Delta} \prod_{f_S \in \mathcal{F}} f_S(x_S)$ is the normalizing constant, also called the partition function.

Among the various tasks, the *Most Probable Explanation* (MPE) problem is to find the most likely assignment $x \in \Delta$ to all the variables in $\mathcal{X}$ maximizing $\mathbb{P}(x)$. By taking the opposite of the logarithm of $\mathbb{P}(x)$, *i.e.,*

[1] DGVNS [23] exploits, within VNS/LDS+CP, structural knowledge coming from tree decomposition in order to efficiently guide the exploration of large neighborhoods (Section 2).

[2] Fixed backbone design application.

Algorithm 1: Limited Discrepancy Search algorithm.

```
   Function LDS(ℓ, A, ub : In/Out, x : Out) : Boolean
       left ← true ; right ← true ;
       if (∃D_i ∈ A, |D_i| > 1) then
           Choose an unassigned variable X_i ∈ 𝒳 such that |D_i| > 1 ;
           Choose a value x_i ∈ D_i ;
           A' ← (A \ {D_i}) ∪ {{x_i}} ;
           if (lb(A') < ub) then
               left ← LDS(ℓ, A', ub, x) ;                          // left branch
           if (ℓ > 0) then
               A'' ← (A \ {D_i}) ∪ {D_i \ {x_i}} ;
               if (lb(A'') < ub) then
                   right ← LDS(ℓ − 1, A'', ub, x) ;                // right branch
           else
1              return false ;                                      // search is incomplete
       else
2          ub ← lb(A), x ← A ;                                     // new solution found
       return left ∧ right ;                                       // true if both branches are complete
```

$$-\log \mathbb{P}(x) = \sum_{f_S \in \mathcal{F}} -\log f_S(x_S) + \log Z = \sum_{f_S \in \mathcal{F}} \varphi(x_S) + \log Z$$

we obtain an additive model with $\varphi(x_S)$ called an *energy function*. Finding a solution of minimum energy is equivalent to MPE. In the rest of the paper, we consider energy minimization. When $\varphi(x_S)$ maps to $\mathbb{N}^+ \cup \{\infty\}$, the corresponding deterministic graphical model is called a *Cost Function Network* (CFN) [27]. Finding a solution of minimum cost is the same as doing energy minimization on the equivalent probabilistic model [28].

Specific solving methods have been proposed to solve these problems but two general approaches can be considered. The first one applies traditional search techniques based on backtracking or branch and bound. In the worst case, their time complexity is in $O(d^n)$ while being generally linear in space. The second one relies on methods that exploit the notion of decomposition of graphs and which are based on Dynamic Programming (DP) (see Section 2.4.1). These methods make it possible to guarantee a time complexity in $O(d^w)$ (where w is the minimum width over all the tree decompositions) but with an exponential space complexity.

2.2. DFBB and limited discrepancy search

Depth-First Branch and Bound (DFBB) methods explore a search tree in a systematic way by recursively choosing the next unassigned variable to assign and by choosing a value in its domain for the assignment (the *branch* part) until a better solution is found or it can be proved that the subtree rooted at the current search node has no better solutions and it can be pruned (the *bound* part). DFBB depends on its variable and value ordering heuristics for branching in order to find good solutions rapidly and to reduce the size of the search tree to be explored. It also depends on its lower bound computation in order to prune the search for minimization problems. Typically, lower bounds are built by dynamic programming with bounded memory, such as mini-buckets heuristic [29], or by solving a linear relaxation of the problem or its dual in an exact or approximate way. In the experiments, we exploit during search an approximate dual lower bound called Existential Directional Arc consistency (EDAC) [30] that performs fast incremental problem reformulations with extra domain value pruning. More information can be found in [31].

Limited Discrepancy Search (LDS) [22] is a heuristic method that explores the search tree in a non-systematic way by making a limited number of *wrong* decisions w.r.t. its value ordering heuristic. We assume a binary search tree where at each search node either the selected variable is assigned to its chosen preferred value (left branch) or the value is removed from the domain (right branch). Each value removal corresponds to a wrong decision made by the search, it is called a *discrepancy*. The number of discrepancies is limited by a parameter denoted ℓ. See Algorithm 1, where $\texttt{lb}(A)$ gives a lower bound on the minimum energy $\min_{x \in \prod_{D_i \in A} D_i} -\log \mathbb{P}(x)$ of the partial assignment A.

In order to detect if a complete search has been done, LDS returns true if and only if the discrepancy limit is never reached. Otherwise it returns false as soon as $\ell = 0$ (line 1). If it returns true then LDS is equivalent to a complete DFBB.

In order to produce better quality solutions as time passes, a simple strategy is to iterate LDS with an increasing number of discrepancies ℓ going from ℓ_{min} to ℓ_{max}. See Algorithm 2 for this Iterative LDS (ILDS) method, where $+_\ell$ is a special sum operator that will be discussed in Section 3. The minimum energy and its corresponding solution are provided in global variables ub and x. In the sequel, we give no initial upper bound ($ub = \infty$).

Proposition 1. $\texttt{ILDS}(0, n(d-1), +, \infty, \{\})$ *is a complete method with a worst-case time complexity exponential in the number of variables and a linear space complexity.*

Proof. First, we prove that ILDS returns true if and only if optimality was proven. Each iteration does at most ℓ discrepancies along the path from the root search node to a terminal node. With a sufficiently large discrepancy limit, LDS never

Algorithm 2: Iterative LDS algorithm.

```
   Function ILDS(ℓ_min, ℓ_max, +_ℓ, ub : In/Out, x : Out) : Boolean
       r ← 0 ;                                          // number of discrepancy iterations
       ℓ ← ℓ_min ;                                      // initial discrepancy limit
       while (ℓ ≤ ℓ_max) do
           opt ← LDS(ℓ, D, ub, x) ;
3          if (opt ∨ ub = lb(D)) then return true ;
           r ← r + 1 ;
4          if (ℓ < ℓ_max) then ℓ ← min(ℓ_max, ℓ_min +_ℓ r) ;
           else ℓ ← ∞
       return false ;
```

reaches its discrepancy limit ($\ell = 0$) and explores a complete search tree, so both LDS and ILDS (line 3) return true. Because we have in the worst case $n(d-1)$ value removals (right branches) to reach a terminal node before assigning all the variables, we can set $\ell_{max} = n(d-1)$ and at least, the last iteration is complete,[3] *i.e.,* $\texttt{LDS}(n(d-1), \mathcal{D}, ub, x)$ is equivalent to a complete DFBB. Here, d is the maximum domain size of all the variables.

Another condition for completeness is reached when LDS finds a solution with a cost equal to a known lower bound of the problem. In this case, LDS will stop branching and ILDS will return true before reaching its last iteration (line 3).

For simplicity reasons, let assume variables with Boolean domains ($d = 2$). The maximum height h of the explored search tree is therefore equal to the number of variables $h = n$. The number of terminal nodes with exactly ℓ discrepancies is bounded by $\binom{h}{\ell}$. In the worst case, ILDS runs for $\ell = 0..h$. The number of terminal nodes of LDS for $\ell = h$ is equal to $\binom{h}{0} + \binom{h}{1} + \binom{h}{2} + \ldots + \binom{h}{h} = \sum_{0 \leq k \leq h} \binom{h}{k} = 2^h$. Thus in the worst case, one iteration of LDS has a time complexity in $\Theta(2^h)$. By doing at most $h+1$ iterations (from $\ell = 0$ to $\ell_{max} = h$), ILDS will explore at least $\Omega(2^h)$ terminal nodes. The asymptotic time complexity of ILDS is therefore exponential in the number of variables.

Because LDS has a linear space complexity, thanks to its depth-first search principle as in DFBB, ILDS has also a linear space complexity. □

In [32], a similar stopping condition for optimality proof was presented. Because heuristics are often less informed near the root of the search tree, it is usually better to make wrong decisions at the beginning of the search [22,33]. It favors exploring new parts of the search tree, possibly finding better solutions that will prune the remaining part of the search tree already explored at previous iterations. Our actual implementation of LDS exploits this fact (right branch done before left branch when the discrepancy limit is not reached).

2.3. Variable neighborhood search

VNS [20] is a metaheuristic that uses a finite set of pre-selected neighborhood structures $N_k, k = 1, 2, ..., k_{max}$ to escape from local minima by systematically changing the neighborhood structure if the current one does not improve the incumbent solution. VNS repeatedly performs three major steps. In the first one, called *shaking*, a solution x' is randomly generated in the neighborhoods of x denoted $N_k(x)$. In the second one, a local search method is applied from x' to obtain a local optimum x''. In the third one, called *neighborhood change*, if x'' is better, than x is replaced with x'' and k is set to 1; otherwise, k is increased by one.

The use of VNS scheme for solving deterministic graphical models started with VNS/LDS+CP [21] and improved in DGVNS [23] (see section 2.4). This approach is related to LNS [15], but it adjusts dynamically the neighborhood size and exploits a tree decomposition of the constraint graph of the problem when the search seems to stagnate as in VNS.

2.4. Decomposition guided variable neighborhood search

Recently, Fontaine et al. [23] investigated the incorporation of tree decomposition in order to efficiently guide the exploration of large neighborhoods. They proposed Decomposition Guided VNS (DGVNS), a first local search approach that exploits the graph of clusters provided by a tree decomposition of the constraint graph of the problem to build relevant neighborhood structures. The next section 2.4.1 defines formally the constraints tree decomposition. Then we present in section 2.4.2 the construction of the initial solution exploited by DGVNS. In section 2.4.3, we detail the main DGVNS algorithm, and show in section 2.4.4 how to instantiate VNS/LDS+CP algorithm from DGVNS. Finally, we briefly discuss the impact of a tree decomposition on the performance of DGVNS.

[3] In practice, ℓ was less than or equal to 128 for all instances completely solved within 1 hour CPU time limit by LDS and VNS methods in Section 5.

Algorithm 3: Decomposition Guided VNS algorithm.

```
   Procedure DGVNS(ℓ, k_min, k_max, ub : In/Out, x : Out)
       let (C_T, T) be a tree decomposition of (X, D, F) ;
       LDS^r(n(d − 1), D, ub, x) ;                                   // find an initial solution
       c ← 1 ;                                                       // current cluster index
       i ← 0 ;                                          // nb. of successive failed neighborhood sizes
       k ← k_min ;                                                   // initial neighborhood size
       while (k ≤ k_max ∧ ¬TimeOut) do
5          A ← getNeighborhood(x, C_c, k) ;
           ub' ← ub ;
6          LDS^r(ℓ, A, ub', x') ;                                    // neighborhood search
           if (ub' < ub) then
7              x ← x',  ub ← ub' ;                                   // new best solution
8              i ← 0,  k ← k_min ;
           else
               i ← i + 1 ;
9              k ← min(k_max, k_min + i) ;
10         c ← 1 + c mod |C_T| ;                                     // visit next cluster
   Function getNeighborhood(x, C, k)
       if k ≥ |X| then
11         X_un ← X
       else
12         Cand ← CompleteCluster(C, k) ;
13         X_un ← Random(Cand, k) ;                  // Random selection of k conflict variables
14     A ← {D_i | X_i ∈ X_un} ∪ {{x_i} | X_i ∈ X\X_un} ;             // Unassign selected variables
       return A ;
```

2.4.1. Tree decomposition

Definition 2. A tree decomposition of a connected graphical model G 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 $\mathcal{X}$, called *clusters*, such that: (i) $\cup_{i \in I} C_i = \mathcal{X}$, (ii) $\forall f_S \in \mathcal{F}$, $\exists\ C_i \in C_T$ 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 3. The intersection of two clusters C_i and C_j is called a *separator*, and noted $sep(C_i, C_j)$.

Definition 4. 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 $sep(C_i, C_j) \neq \emptyset$.

The width w of a tree decomposition (C_T, T) is equal to $\max_{C_i \in C_T} |C_i| - 1$. The treewidth w^* of G is the minimum width over all the tree decompositions of G.

Several studies have focused on the computation of tree decompositions [34,35]. The proposed algorithms can be classified into two approaches: the exact algorithms that compute decompositions of optimal width (equal to the treewidth) and the heuristic methods, which do not offer a guarantee on optimality. The motivation of heuristic approaches is due to the fact that the optimal computation is an NP-hard problem [36]. Heuristic approaches dedicated to handle graphical models often use triangulation (such as *Minimum Fill-in* (min-fill) [37] and *Maximum Cardinality Search* (MCS) [38]). These heuristic approaches allow to process graphs of several thousand of vertices in reasonable time, but without guaranteeing the quality of the obtained decompositions in terms of deviation from the optimum w^*. Other heuristics and pre- or post-processing rules may be applied to reduce the width of the decomposition [35,34]. In this paper, we use a heuristic approach based on min-fill.

2.4.2. Initial solution and restricted LDS

VNS relies on an initial solution x. Without any infinite terms in the problem (corresponding to forbidden assignments or hard constraints), x can be produced by a greedy search algorithm such as $\text{LDS}(0, \mathcal{D}, \infty, x)$. Otherwise, we can either relax the problem (by replacing every infinite energy term by the sum of the greatest finite term of each original energy function in the problem) or rely on a complete search method. For that, we made a modified version of LDS, called Restricted LDS (LDS^r), that stops immediately after a first solution is found.[4]

In particular, $\text{LDS}^r(n(d-1), \mathcal{D}, \infty, x)$ will either find a solution of finite cost (*i.e.,* satisfying all the constraints) or prove the problem has no feasible solution.

[4] At the end of line 2 of Algorithm 1, it stops the recursive LDS procedure and returns false.

2.4.3. DGVNS algorithm

Algorithm 3 shows the pseudo-code of DGVNS. It exploits the graph of clusters provided by a tree decomposition of the constraint graph of the problem to build relevant neighborhood structures. Let $\mathcal{X}$ be the set of variables, and let $N_{k,c}$ be the neighborhood structure, where k is the neighborhood size and C_c is the cluster where the variables will be selected from.

First, an initial solution x is generated by LDS^r, as detailed in section 2.4.2. Second, to favor moves on regions that are closely linked, x is partially destroyed by unassigning a subset of k variables and an exploration of its (large) neighborhood is performed until the solution is repaired with a new one. To select the variables to be unassigned, DGVNS uses a neighborhood heuristic based on clusters (see function getNeighborhood, line 5): the set of candidate variables *Cand* to be unassigned are selected in the same cluster C_c. Indeed, the concept of cluster embodies this criterion, because of its size (smaller than the original problem), and by the strong connection of the variables it contains. If $(k > |C_c|)$, we complete the set of candidate variables to be unassigned by adding clusters adjacent to C_c in order to take into account the topology of the graph of clusters. This treatment is achieved by function CompleteCluster(C_c, k) (line 12). Third, a subset of k variables X_{un} is randomly selected in *Cand* (line 13), and then rebuilt using LDS^r. In the particular case where k is greater than the variables cardinality (line 11), the whole variables are selected. The neighborhood change in DGVNS is performed in the same way as in VNS. However, DGVNS considers successively all the clusters C_c. This ensures a better diversification by covering a large number of different regions, and to locate the region containing the global optimum.

The core of DGVNS is its reconstruction phase. It relies on a non local solver combining constraint propagation and Restricted LDS with a fixed discrepancy to explore the neighborhood of the solution. One advantage of this choice is its exploration speed that improves the quality profile and allows a more balanced exploration of the search tree. First, a subset of k variables are selected in $\mathcal{X}$. Then, a partial assignment $\mathcal{A}$ is generated from the current solution x by unfixing the k selected variables, and then re-built in the best way (line 6).

Let *succ* a successor function[5] and $N_{k,c}$ the current neighborhood structure: if LDS^r finds a (first) solution of better quality x' in the neighborhood of x (line 6), then x' becomes the current solution (line 7), k is reset to k_{min} (line 8), and the next cluster is considered (line 10). Otherwise, DGVNS looks for improvements in $N_{(k+1),succ(c)}$, a neighborhood structure where $(k+1)$ variables will be unassigned (line 9). In fact, when a local minimum is found in the current neighborhood, moving from k to $(k+1)$ will also provide some diversification by enlarging the neighborhood size. The search stops when it reaches the maximal neighborhood size allowed or a timeout.

2.4.4. From DGVNS to VNS/LDS+CP

When a tree decomposition of the constraint graph of the problem is not available, the constraints can be handled as a single cluster (i.e. $|C_T| = 1$), and DGVNS behaves as VNS/LDS+CP algorithm [23]. In this case, the search process completes the variables with LDS^r without taking into account the constraints connectivity, which is intuitively less efficient than the solving process when $|C_T| > 1$. More precisely, when the variables X_{un} are strongly connected through the constraint graph, LDS^r will efficiently instantiate these variables thanks to constraint propagation, called at every LDS search node, where it is well known that its effectiveness depends on the connectivity of the variables to be instantiated. This idea is cleverly exploited by DGVNS thanks to the concept of cluster provided by the tree decomposition of the constraint graph. In VNS/LDS+CP, the neighborhood heuristic (function getNeighborhood) randomly selects k variables to unassign among conflicted ones. Such a heuristic which is mainly based on random choices has a major drawback since it does not take advantage of the topology of the constraint graph. For instance, it may select unrelated variables (i.e., no constraint may be fully unassigned), and all selected variables may also have a high degree (i.e., they occur in many constraints). In such a case, it is unlikely to rebuild them without violating several constraints, and thus to find a solution of better quality than the current one. Nevertheless, as reported by [23], VNS/LDS+CP remains efficient on some problems, but DGVNS is much more efficient.

2.4.5. Synthesis

The motivation of exploiting the graph of clusters of a tree decomposition within VNS/LDS+CP algorithm is to build pertinent neighborhood structures enabling better diversification. Clearly, the quality of the tree decomposition impacts greatly the performance of DGVNS.

In our prior works [23], we have studied the impact of some parameters related to topological properties of the tree decomposition: the *width* of a tree decomposition w, *separators size*, and *the decomposability* of a problem $(\frac{w}{n})$, estimated by the ratio between the width of a tree decomposition and the number of variables. The width of a tree decomposition gives a good indication on the size of subproblems, while separators size provides information about the connectivity between clusters and the degree of their overlap.

From this study, we showed that DGVNS is very effective on problems that decompose into *weakly connected clusters of reasonable size*, i.e. tree decompositions characterized by low values of $(\frac{w}{n})$, and by separators of small size (clusters that do not overlap heavily), leading to more pertinent neighborhoods.

[5] $succ(c) = 1 + c \bmod |C_T|$.

Algorithm 4: Unified Decomposition Guided VNS algorithm.

```
   Procedure UDGVNS(ℓ_min, ℓ_max, +_ℓ, k_min, k_max, +_k, ub : In/Out, x : Out, opt : Out)
       let (C_T, T) be a tree decomposition of (X, D, F) ;
15     opt ← LDS^r(n(d − 1), D, ub, x) ;                          // find an initial solution
16     if (ub = lb(D)) then opt ← true ;
       c ← 1 ;                                                    // current cluster index
       r ← 0 ;                                                    // number of discrepancy iterations
       ℓ ← ℓ_min ;                                                // initial discrepancy limit
       while (¬opt ∧ ℓ ≤ ℓ_max) do
           i ← 0 ;                                                // nb. of successive failed neighborhood sizes
17         k ← k_min ;                                            // initial neighborhood size
           while (¬opt ∧ k ≤ k_max) do
               A ← getNeighborhood(x, C_c, k) ;
               ub' ← ub ;
18             opt ← LDS^r(ℓ, A, ub', x');                        // neighborhood search
19             if (ub' = lb(D)) then opt ← true ;
20             else if (A ≠ D) then opt ← false ;
21             if (ub' < ub) then
                   x ← x', ub ← ub' ;                             // new best solution
22                 i ← 0, k ← k_min ;
23                 r ← 0, ℓ ← ℓ_min ;
               else
                   i ← i + 1 ;
                   if (k < k_max) then
                       k ← min(k_max, k_min +_k i) ;
                   else k ← ∞
24             c ← 1 + c mod |C_T| ;                              // visit next cluster
           r ← r + 1 ;
           if (ℓ < ℓ_max) then
               ℓ ← min(ℓ_max, ℓ_min +_ℓ r) ;
           else
               ℓ ← ∞
       // End
```

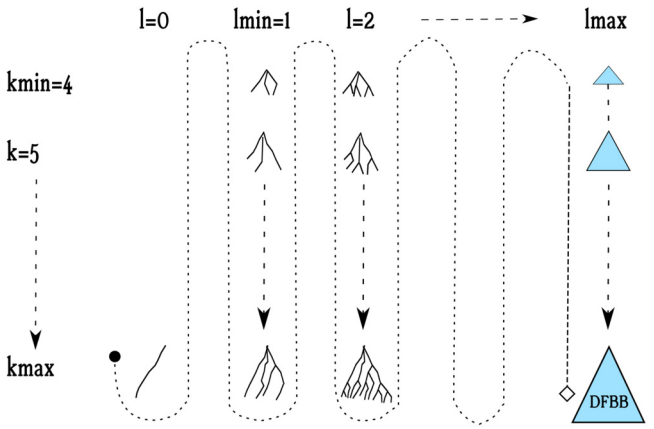

Fig. 1. A general overview of UDGVNS algorithm, exploring successively different search trees, starting from an initial greedy search (•), and ending to a complete search (◇).

3. Unified decomposition guided VNS

We present UDGVNS in Algorithm 4, an iterative DGVNS method, unifying two complete and incomplete search methods. As done by Iterative LDS, UDGVNS restores the completeness of DGVNS by applying successive calls with an increasing discrepancy limit.

3.1. UDGVNS algorithm

As in the previous VNS algorithms, the initial solution of UDGVNS is obtained by LDS^r (line 15), corresponding to a greedy search with no discrepancy if the problem to be solved has only finite energy terms. Then UDGVNS tries to improve the current solution by doing several neighborhood explorations of increasing search complexity, by controlling two parameters, the discrepancy limit (ℓ) and the neighborhood size (k), as shown in Fig. 1. It starts from a small neighborhood with a few variables unassigned ($k = k_{min} = 4$). It explores the neighborhood using LDS with a small discrepancy limit

initially set to one ($\ell = \ell_{min} = 1$). The unassigned variables are selected from a current cluster of a tree decomposition (and its neighbor clusters if needed) as done in DGVNS. If no solution is found then UDGVNS increases its neighborhood size until all the variables of the problem are included in the neighborhood ($k = k_{max} = n$). If still no solution is found then UDGVNS increases its discrepancy limit and resets its neighborhood size to k_{min}. The last iteration corresponds to a complete search on the whole problem for proving optimality ($k = k_{max} = n$, $\ell = \ell_{max} = n(d-1)$). As soon as a better solution is found by the current neighborhood search (line 18), UDGVNS stops the search in order to reinitialize the two parameters to their minimum value (lines 22–23). By doing so, it favors finding the next solutions more rapidly, as it is faster to explore many small neighborhoods than a larger one, improving the anytime behavior of the search.

Proposition 2. UDGVNS$(1, n(d-1), +, 1, n, +, \infty, \{\}, opt)$ *is a complete method with a worst-case time complexity exponential in the number of variables and a linear space complexity.*

Proof. For UDGVNS, optimality can be proven in two cases: (i) when the current neighborhood corresponds to the whole problem (condition falsified at line 20, since $(A \neq \mathcal{D})$ is false) and the discrepancy value is greater than or equal to the maximum number of right branches (checked as before during LDS searches at lines 15, 18), and thus DGVNS behaves as an exhaustive search, or (ii) by examining the bounds at the root node (line 16) and after each neighborhood search (line 19). In this case, the search space is implicitly explored by the algorithm, therefore optimality is proven. Case (i) is always reached when $k = k_{max} = n$ and $\ell = \ell_{max} = n(d-1)$, corresponding here to the last iteration of the two *While* loops of UDGVNS. Notice that in this case $(k = k_{max})$ and $(\ell = \ell_{max})$, all the variables are selected candidates (see line 11) to be explored exhaustively by an LDS stopping at the first solution better than the current bound. If the current bound is not optimal, the search restarts until reaching optimality. In practice, optimality proofs are often produced at smaller ℓ, but still for $k = n$ (or before if condition at line 19 becomes true).

Assuming a complete search tree over Boolean variables ($d = 2$), the worst-case time complexity of the initial LDS^r at line 15 is in $\Theta(2^n)$. The inner LDS^r at line 18 with k variables and ℓ discrepancies has a search tree with maximum height $h = k$, assuming variables with Boolean domains. In the worst case, its asymptotic time complexity is in $\Theta(2^h)$ for $\ell = h = k$ (see Proposition 1). The number of LDS^r calls depends on the problem upper bound. Each time a strictly better upper bound is found (line 21), LDS^r is stopped and we reset k and ℓ to their minimum value (lines 22–23). When the energy functions map to $\mathbb{N}^+ \cup \{\infty\}$ as in Cost Function Network (CFN), there will be a finite number of upper bound improvements. Moreover if $+_\ell$ and $+_k$ are strictly increasing functions, then there is a finite number of LDS^r calls until $\ell = \ell_{max}$ and $k = k_{max}$, and no better solution exists (otherwise, $\ell = \ell_{min}$ and $k = k_{min}$, and the search will continue). Thus, UDGVNS terminates and returns $opt =$ **true** if and only if it exists a feasible optimal solution. It has the same exponential time and linear space complexities as ILDS. □

3.2. Strategies for managing parameters of UDGVNS

UDGVNS has to control the evolution of two parameters. For each parameter, ℓ and k, we have tested three updating rules: increase by one at each iteration ($+_{\ell/k} = +$), multiply by two at each iteration ($a +_{\ell/k} b = \texttt{mult2}(a, b) = a \times 2^b$), and apply a Luby strategy [39] ($a +_{\ell/k} b = \texttt{Luby}(a, b) = a \times luby(1 + b)$).[6]

Operator $+_k$ controls the compromise between intensification and diversification. The goal of the Luby strategy is to exponentially increase the number of small neighborhoods explored compared to the number of larger ones. Whereas classical VNS algorithms will get stuck on large problems,[7] trying to diversify the search by exploring larger neighborhoods, VNS using Luby will spend more time on small neighborhoods in order to locally improve the current solution, favoring intensification. By adding randomness on variable and/or value ordering heuristics[8] used by LDS^r, it is possible to find a better solution even when the discrepancy limit decreases when applying the Luby strategy. The mult2 strategy reduces the number of neighborhood explorations at a given discrepancy limit, in order to try larger discrepancy limits more rapidly. If the problem is solvable by a complete search within the time limit, it will also speed-up the optimality proof.

Operator $+_\ell$ controls the compromise between incomplete and complete search. Using a fast growing strategy emphasis completeness whereas a slow growth should favor anytime behavior. The mult2 strategy tends to favor a non-decreasing worst-case complexity of the successive neighborhood searches, especially when going from ℓ with $k = k_{max}$ to 2ℓ with $k = k_{min}$ (for sufficiently large k_{min}).

We noticed that it is worthwhile to cover all the variables by the union of the explored neighborhoods in order to not miss some important variables. We tested a fourth strategy for k which consists in a slow increment (by $+1$) at the beginning until $k = \max_{i \in I}(|C_i|) + |C_T| - 1$ then it *jumps* directly to $k = k_{max}$. This ensures that k grows slowly until the largest cluster has been totally explored by at least one neighborhood search. Then, when $k = k_{max} = |\mathcal{X}|$, UDGVNS does a restart, looking for an improved starting solution, using LDS^r applied on the whole problem. If it fails to find a better

[6] Recall $luby(i) = \{1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, \ldots\}$.

[7] Although it is possible to add a limit on the number of backtracks per neighborhood search as it is done in Large Neighborhood Search methods [15].

[8] Adaptive heuristics such as weighted degree heuristic [40] will also modify the variable ordering from one search to another.

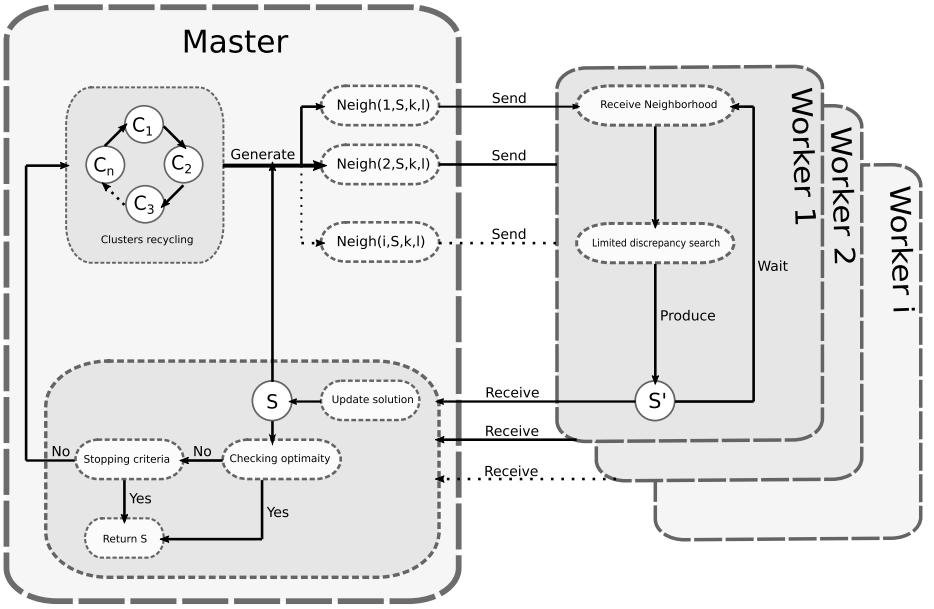

Fig. 2. A general overview of UPDGVNS algorithm. The set $\{C_1, C_2, \ldots, C_n\}$ corresponds to the n clusters provided by the tree decomposition. The total number of available workers is denoted by i. The master's global solution is denoted by S. Local solutions found by workers are denoted by S'.

solution, or prove optimality, a larger discrepancy can be selected and UDGVNS continues its intensification process starting with a small neighborhood size (line 17).

4. A parallel version of UDGVNS

This section describes how UDGVNS has been parallelized. We called the resulting algorithm Unified Parallel DGVNS (UPDGVNS). Section 4.1 provides a general overview of the parallel version. The UPDGVNS algorithm is detailed in section 4.2. A more detailed discussion about UPDGVNS properties is given in section 4.3.

4.1. UPDGVNS in nutshell

The parallel version relies on a master/worker model and exploits the UDGVNS framework to control the compromise between optimality proof and anytime behavior. UPDGVNS enhances the optimization process of UDGVNS by enabling better diversification. More precisely, UPDGVNS uses the master process as a diversification component to explore the search space on a global scale, while using the worker process as an intensification component to exploit the search space on the neighborhood provided by the master. Fig. 2 provides an overview of UPDGVNS method based on the following three main components:

- a *master* process, on the left side, which determines the neighborhoods to be explored and updates the global solution S at each iteration.
- a set of asynchronous *worker* processes, on the right side, which explore independently the parts of the search space assigned by the master process.
- an interaction model based on asynchronous communication between the master and the workers, where the master process controls the communication over the entire processes.

4.2. UPDGVNS algorithm

Algorithms 5 and 6 depict the pseudo-code of UPDGVNS in more details. Let $\mathcal{P}$ be a data structure allowing to manage a list of parameters used by each worker process for the exploration of the neighborhood of a solution x (i.e., i, k, r, ℓ, cl, x, ub, opt). Initially, the master (see Algorithm 5) initiates the search by launching (at line 26) npr[9] worker processes in parallel with the same initial solution (line 25). This is done by initializing the different parameters for the neighborhood exploration and sending them to each worker p. Each worker process obtains from the master a copy of the current best solution x, the index c of the cluster to be processed and performs destroy and repair operations on its local copy (see

[9] Worker processes are ranked from 1 to npr, while the masker is ranked zero.

Algorithm 5: Master algorithm for Unified Parallel DGVNS algorithm.

```
Procedure master(npr, ℓ_min, ℓ_max, +_ℓ, k_min, k_max, +_k, ub : In/Out, x : Out, opt : Out)
    let (C_T, T) be a tree decomposition of (X, D, F) ;
25  opt ← LDS^r(n(d − 1), D, ub, x) ;                                   // find an initial solution
    if (ub = lb(D)) then opt ← true if (¬opt) then
        c ← 1 ;                                                          // current cluster index
26      for each worker p = 1, ..., npr do
            // initial parameters of neighborhood exploration
            P[p].ub ← ub, P[p].x ← x, P[p].cl ← c, P[p].ℓ ← ℓ_min, P[p].k ← k_min ;
            P[p].r ← 0 ;                                                 // number of discrepancy iterations
            P[p].i ← 0 ;                                                 // nb. of succ. failed neighb. sizes
            Send(p, P[p]);
            c ← 1 + c mod |C_T| ;                                        // visit next cluster
27      while (¬opt ∧ ¬TimeOut) do
            Receive(p, P[p]) ;                                           // wait a new solution from worker p
28          if (P[p].opt) then opt ← true // optimality proof check;
            if (P[p].k ≥ k_max ∧ P[p].ℓ ≥ ℓ_max) then TimeOut ← true;
            // update parameters of neighborhood exploration
            P[p].cl ← c;
            if (P[p].ub < ub) then
                ub ← P[p].ub, x ← P[p].x ;                               // new best solution
                P[p].i ← 0, P[p].k ← k_min ;
                P[p].r ← 0, P[p].ℓ ← ℓ_min ;
            else
                P[p].ub ← ub, P[p].x ← x ;
                P[p].i ← P[p].i + 1 ;
                if (P[p].k < k_max) then
                    P[p].k ← min(k_max, k_min +_k P[p].i) ;
                else
                    P[p].i ← 0, P[p].k ← k_min ;
                    P[p].r ← P[p].r + 1 ;
                    if (P[p].ℓ < ℓ_max) then
                        P[p].ℓ ← min(ℓ_max, ℓ_min +_ℓ P[p].r) ;
            c ← 1 + c mod |C_T|;
29          if (¬opt ∧ ¬TimeOut) then Send(p, P[p])
```

Algorithm 6: Worker algorithm for UPDGVNS algorithm.

```
Procedure worker
    while (¬TimeOut) do
        Receive(0, P);
        A ← getNeighborhood(P.x, C_{P.cl}, P.k) ;
        P.opt ← LDS^r P.ℓ, A, P.ub, P.x ;
        if (P.ub = lb(D)) then P.opt ← true else if (A ≠ D) then P.opt ← false Send(0, P);
```

Algorithm 6). As soon as a new solution x' is found by a worker p, it is sent to the master as well as its status (i.e. flag *opt*) by checking whether x' is proven optimal, by setting flag *opt* to *true*. In the second while loop (at line 27), the master waits for new solutions found by each worker process. Like UDGVNS, the master controls how the discrepancy limit ℓ and the neighborhood size k evolve during successive explorations and updates the shared global best solution x according to solutions sent by the workers. However, contrary to UDGVNS, whenever k reaches k_{max} and the discrepancy value is greater than or equal to ℓ_{max} or the internal flag *opt* of the worker is *true* (line 28), the whole process stops and the master returns the (optimal) solution. Otherwise, if the time limit is not reached, the workers that are ready to restart a new exploration are re-launched starting from the best available overall solution on the next clusters (line 29).

4.3. UPDGVNS properties

This section discusses in depth the key properties of UPDGVNS that contribute to its success, namely diversification and workload distribution between workers.

4.3.1. Diversification in UPDGVNS

Regarding the parallelization scheme, diversification is ensured in three ways:

1. the parallel exploration of different clusters provides a form of diversification to UPDGVNS method by exploring independently different parts of the search space. Moreover, the unassigned variables are selected from a current cluster of a tree decomposition (and its neighbor clusters if needed) in a random way as done in DGVNS algorithm.

2. the control of the size of the neighborhood (k) and the discrepancy limit (ℓ) is done per worker in an asynchronous and independent manner. This allows relaunching each worker asking for a next cluster with diverse parameters and different initial partial assignments, thus enhancing the diversification scheme of UPDGVNS.
3. to prevent redundant searches among different workers, i.e., case where two or more identical[10] neighborhoods are explored, we add randomness in the variable ordering heuristic used in LDS^r algorithm. More precisely, our heuristic for variable ordering first selects the variable having the lowest ratio domain cardinality divided by weighted degree (dom/wdeg), breaking ties by selecting one variable randomly. This leads to variations in the exploration of the search tree performed by each worker.

All these features provide a high level of diversification by exploring different regions in parallel and allow to decrease the probability that different workers perform the same exploration of the search space even for $k = k_{max} = n$.

4.3.2. Workload distribution in UPDGVNS

In our parallel algorithm, we do not decompose the whole search space into a partition of subproblems but rather explore different randomly-selected subproblems which may overlap. First, each subproblem is related to a particular cluster in the tree decomposition and each time a worker asks for a job it gets the next (current) cluster c in a global round-robin fashion (Algorithm 6) and explores the assigned cluster independently starting from the best overall available solution. So, in terms of workload distribution between workers, this remains more or less balanced. Moreover, since the proof of optimality is performed sequentially, i.e., by a single worker who explores the entire search tree, this does not induce unbalanced workloads among workers.

Second, the way the size of the neighborhood (k) and the discrepancy limit (ℓ) evolve is done per worker in an asynchronous and independent manner. This can lead to a situation where one worker may search on a small subproblem, while another worker on a larger subproblem. If the larger subproblem is not defined over the whole problem, the balance on subproblems is quickly established with the rest of the workers once the current worker finishes its exploration. This is achieved by the master which updates parameters of the neighborhood exploration accordingly (i.e., neighborhood size and discrepancy limit) in order to relaunch the worker on the next cluster. In the end, the first worker finishing its search on the whole problem with no discrepancy limit will report optimality and stop UPDGVNS.

5. UAI evaluation results

5.1. Benchmarks description

We performed experiments on probabilistic and deterministic graphical models coming from a large multi-paradigm benchmark repository [28].[11] Among the 3016 available instances, we selected all the instances that were used in previous Uncertainty in Artificial Intelligence (UAI) competitions, in image analysis, or in CFN optimization. It includes 319 instances from *UAI 2008 Evaluation* and Probabilistic Inference Challenge (PIC) 2011,[12] 1461 instances from the Computer Vision and Pattern Recognition (CVPR) OpenGM2 benchmark[13] [41], and 281 instances from the Cost Function Library.[14] In order to have a fair comparison between solvers, we preprocessed all the instances by polynomial-time problem reformulations and simplifications that remove variables (using bounded and functional variable elimination [42]), values (using *dead-end elimination* [43]), and fixed-value potentials, after an initial lower bound computation by *Equivalence Preserving Transformations* [31] (enforcing *Virtual Arc Consistency (VAC)* as a message-passing algorithm). The resulting instances are smaller while preserving the same optimum. We used TOULBAR2 with options *-A -z=2* for this preprocessing step. We kept 1669 non-trivial instances (with more than one variable) for the experiments. The number of variables n ranges from 4 (CVPR-GeomSurf-3-gm13) to 48,566 (CVPR-ColorSeg-8-crops-small) with mean value $n \approx 403.4$ (instead of $1,316.5$ before preprocessing). For DGVNS methods, we built tree decompositions using *min-fill* heuristic. Because the number of clusters $m = |C_T|$ can be very large ($m \approx 256.7$), we merged any pair of connected clusters (C_i, C_j) when the separator size is too large compared to the individual cluster sizes ($|sep(C_i, C_j)| > 0.7 \min(|C_i|, |C_j|)$), resulting in $m \approx 19.9$ and mean treewidth $\max_{i \in I}(|C_i|) \approx 76.9$ (instead of 49.6 without merging). In order to experiment sequential and parallel methods on the most difficult instances, we selected a subset of instances unsolved in 1 hour by all our DFBB, LDS, and VNS algorithms. We took at most twenty instances per problem category (avoiding over-representation issues by some categories), resulting in a selection of 114 difficult instances.[15]

[10] This may arise when all the variables of the problem are included in the neighborhood ($k = k_{max} = n$).
[11] genoweb.toulouse.inra.fr/~degivry/evalgm.
[12] graphmod.ics.uci.edu/uai08/Evaluation/Report/Benchmarks, www.cs.huji.ac.il/project/PASCAL.
[13] hci.iwr.uni-heidelberg.de/opengm2.
[14] costfunction.org/benchmark.
[15] UAI DBN, Grid, Linkage, ObjectDetection, CVPR ChineseChars, ColorSeg-8, InPainting-4, InPainting-8, ProteinInteraction, and CFN CELAR, ProteinDesign, SPOT5, Warehouse categories.

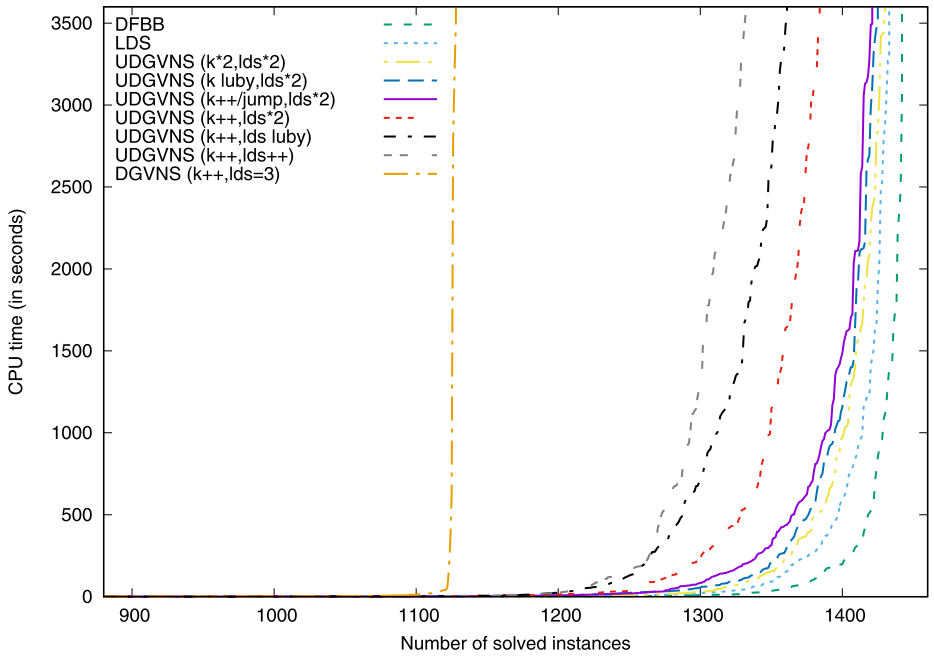

Fig. 3. Number of instances solved by our approach as times passes on a restricted benchmark set (Methods are sorted as timeout limit (3600s)).

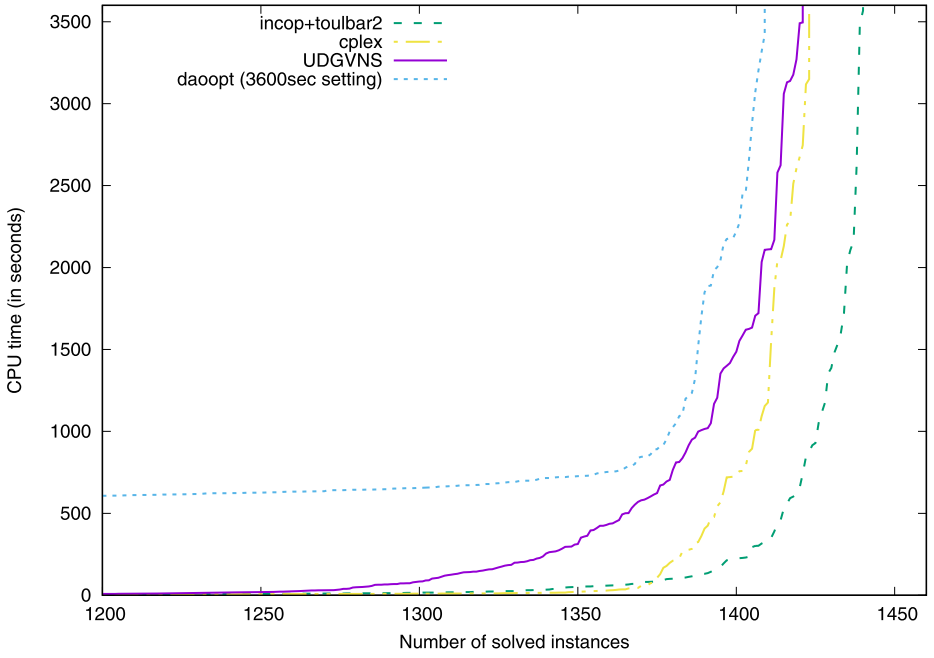

Fig. 4. Number of instances solved by each method as time passes (UDGVNS = UDGVNS(k add1/jump, ℓ mult2)).

5.2. Experimental protocol

LDS^r employs a randomized (for breaking ties) dynamic variable ordering heuristic.[16] Its value ordering heuristic chooses the EAC value as the preferred value and lower bounds are deduced by enforcing EDAC, as explained

[16] Weighted degree heuristic as defined in [40].

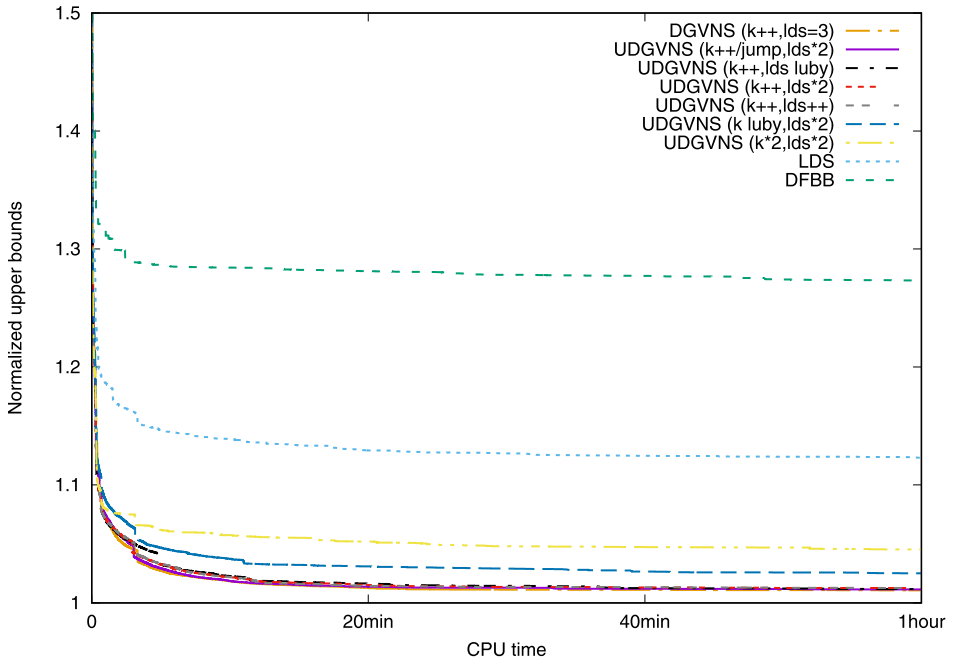

Fig. 5. Average evolution of normalized upper bounds for UDGVNS versus DFBB and LDS algorithms on 114 difficult instances.

in [30]. In the following, we set $k_{min} = 4$, $k_{max} = n = |\mathcal{X}|$, $\ell_{min} = 1$, and $\ell_{max} = n(d-1)$. DFBB corresponds to UDGVNS$\infty, \infty, +, \infty, \infty, +$, LDS corresponds to UDGVNS$(1, \infty, \texttt{mult2}, |\mathcal{X}|, |\mathcal{X}|, +)$, DGVNS to UDGVNS$3, 3, +, k_{min}, k_{max}, +$. These methods and their parallelization based on MPI (UPDGVNS) have been implemented into the new version 1.0.0 of TOULBAR2.[17]

We compared with state-of-the-art exact solvers. DAOOPT[18] won PIC 2011. It has a time-bounded initial phase of lower bound computation based on *Message Passing Linear Programming* algorithm [8,9] and mini-bucket elimination [29] with iterative *min-fill* heuristic, further improved by Join Graph Linear Programming [44]. It also finds good initial upper bounds using LDS (with discrepancy limit set to 1) and stochastic local search GLS^+ [12]. We used the standalone code of DAOOPT version 0.99.7g-UAI12 (with option settings –mplp=2000 –mplps=60 –slsX=20 –slsT=10 -t 30000 –orderTime=180 –jglp=1000 –jglps=60 -i 35 -m 4096 –match -y –lds=1 for 3600-second time limit). We tested three parameter settings as suggested in [45], controlling the time spent to compute initial lower and upper bounds. In the 3600 sec. setting, SLS is run 20 times with 10 seconds per run. The best solution found is used as an initial upper bound for an AND/OR exhaustive tree search. We compared also with an older version of TOULBAR2, namely INCOP+TOULBAR2[19] [28] won the *UAI 2010 Evaluation* at 20-minute time limit. INCOP+TOULBAR2 takes a starting solution from the best result of three runs of the *IDWalk* [46] local search algorithm (100,000 local moves per run). It is followed by an exhaustive hybrid best-first search [5]. IBM ILOG CPLEX 12.7.0.0 (using parameters EPAGAP, EPGAP, and EPINT set to zero to avoid premature stop) was reported as being very competitive on some image analysis [41] and Markov Random Field problems [28]. CPLEX explores its search tree using best-first search. It applies several heuristics methods to find good solutions before and during the search. We also compared with message-passing algorithms: LIBDAI[20] [47], winner of the *UAI 2010 Evaluation* at 20 sec. and 1 h. time limits, MPLP2[21] [8,9], and TRW-S[22] [6]. Note that LIBDAI and TRW-S are applied on the original instances rather than the preprocessed ones as we found they produced better results without applying VAC first. All solvers read problems in *uai* tabular format, except CPLEX which uses the local polytope formulation (called support encoding in [28]). All computations were performed on a cluster of 48-core AMD Opteron 6176 at 2.3 GHz and 384 GB of RAM with a 1-hour CPU time limit.[23]

[17] https://github.com/toulbar2/toulbar2.
[18] https://github.com/lotten/daoopt.
[19] www.inra.fr/mia/T/toulbar2 version 0.9.8 with parameters *-i -dee -hbfs*.
[20] bitbucket.org/jorism/libdai.git version 0.3.2 using UAI 2010 settings.
[21] cs.nyu.edu/~dsontag/code/README_v2.html using 2.10^{-7} int gap thres.
[22] github.com/opengm/opengm v2.3.5 with TRW-S v1.3 stopping after 100,000 iterations or 10^{-5} gap thres.
[23] Using parameter *-pe parallel_smp* $\min(2x, 30)$ on a SUN Grid Engine for a method exploiting x core(s) to ensure half-load of the nodes on the cluster.

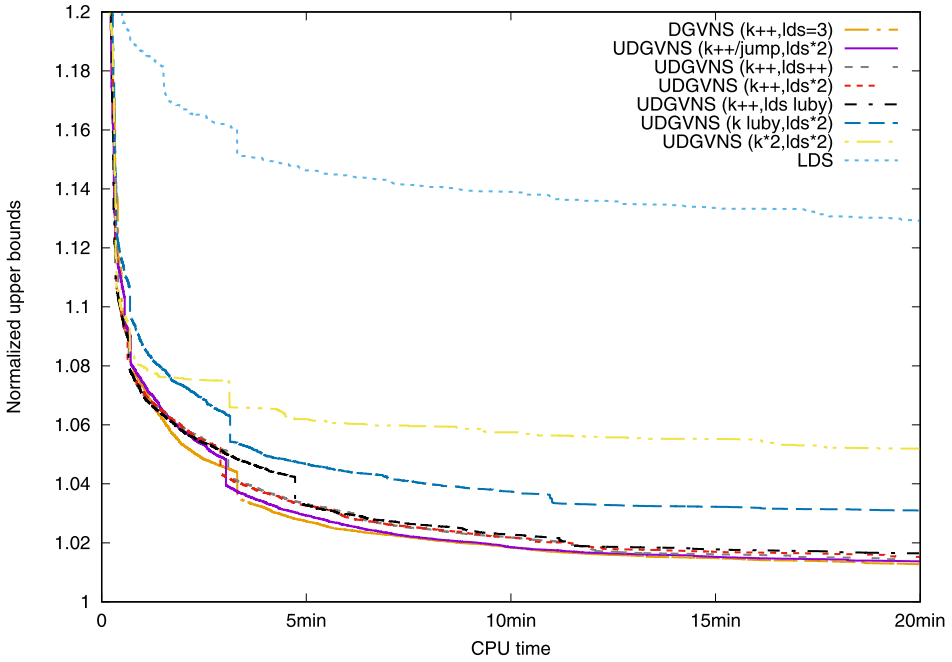

Fig. 6. Any time upper bound zoom for UDGVNS versus LDS.

5.3. Experimental results

5.3.1. Optimality results

The efficiency of DFBB, LDS, and VNS methods to prove optimality is shown in the cactus plot of Fig. 3. DFBB was slightly more efficient than LDS and solved 1442 (resp. 1433) instances among 1669 in 1-hour time limit. They are followed by three UDGVNS strategies with (k mult2, ℓ mult2) (1430 solved), (k Luby, ℓ mult2) (1425 solved) and (k add1/jump, ℓ mult2) (1421 solved), remaining very close in terms of performance. Next, a set of three less-and-less efficient UDGVNS strategies rise: (k add1, ℓ mult2) (1384 solved), (k add1, ℓ Luby) (1361 solved) and (k add1, ℓ add1) (1333 solved), showing the importance of faster discrepancy increase to speed up optimality proofs. The worst strategy here was using a fixed discrepancy level ($\ell = 3$ as originally proposed in [23]) which solved 1128 instances. Fig. 4 compares the cactus plot of UDGVNS versus CPLEX, DAOOPT, and INCOP+TOULBAR2. DAOOPT (3600 sec. setting) solved 1409 instances, CPLEX solved 1423, and INCOP+TOULBAR21440 instances.

5.3.2. Anytime upper bound profiles

In order to summarize the evolution of upper bounds as time passes, we took a subset of 114 difficult instances, unsolved in 1 hour by our DFBB, LDS, and VNS methods (whereas CPLEX could solve 17 of these instances). Specifically, for each instance I we normalize all energies as follows: the best, potentially suboptimal solution found by any algorithm is 1, the worst solution is 2. This normalization is invariant to translation and scaling. Fig. 5 shows the upper bound behavior for different VNS strategies compared to DFBB and LDS. Fig. 6 reports an anytime upper bound zoom. The ranking of best methods is the opposite of the cactus plot order, except for (k add1/jump, ℓ mult2) which comes in second position. According to details in Fig. 6, the $\ell = 3$ strategy got the best upper bounds in average, but still very close to the other VNS strategies, except may-be (k Luby, ℓ mult2) and (k mult2, ℓ mult2). We conclude that our new iterative UDGVNS method (especially (k add1/jump, ℓ mult2)) offers a good compromise between anytime behavior and optimality proof. These results also show that variable neighborhood search is by far superior to classical systematic DFBB or non-systematic LDS tree search methods, improving by more than 20% (resp. 10%) the quality of the solutions.

5.3.3. Comparing UDGVNS with state-of-the-art methods

In the following figures, we assume UDGVNS with (k add1/jump, ℓ mult2) strategy. Fig. 7 compares UDGVNS with state-of-the-art methods. Message-passing algorithms like TRW-S and LIBDAI gave the worst results. They could not find any solution for 20 (resp. 19) instances (mostly in UAI/Linkage and CFN/SPOT5 categories, both containing hard constraints). The same problem occurred for MPLP2 on 5 instances (SPOT5), but it obtained much better results on the remaining instances. CPLEX ran out of memory on two instances without producing any solution due to the heavy local polytope encoding (CFN/Warehouse/capb, capmq5). All other methods got better results in average and produced at least one solution per instance. According to its initial phase setting, DAOOPT provides different anytime behaviors, very close to the best solutions

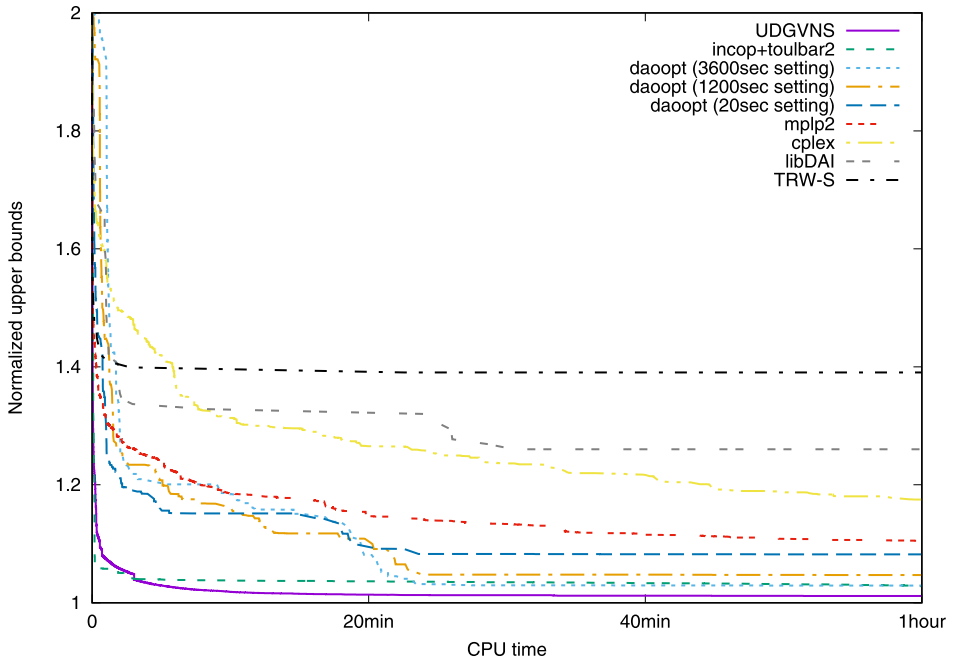

Fig. 7. Comparing the anytime behavior of UDGVNS against state-of-the-art methods.

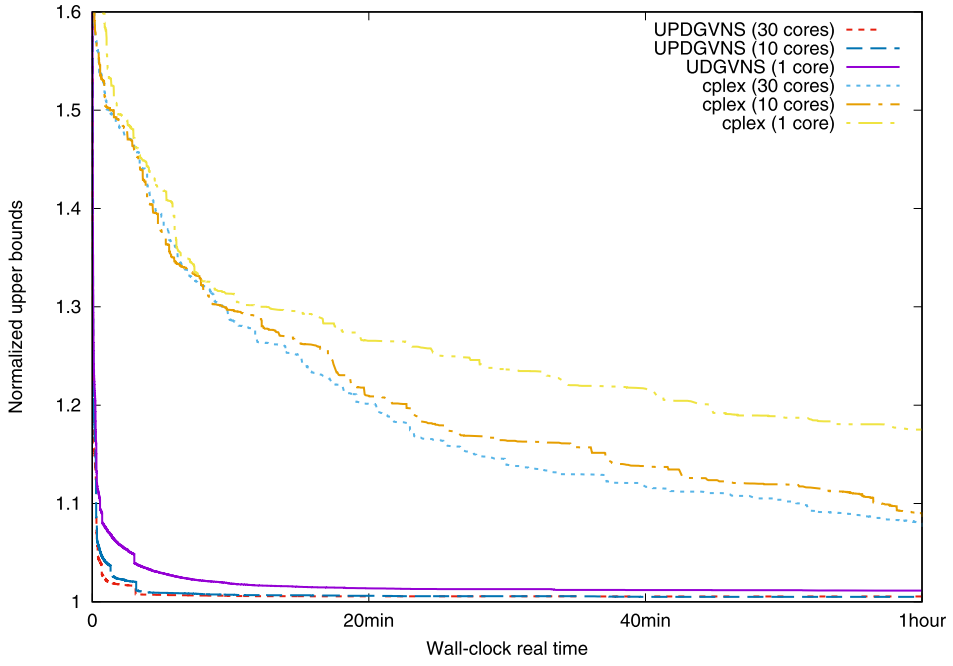

Fig. 8. Anytime upper bound with 1, 10 and 30 processors respectively for CPLEX and UPDGVNS (UPDGVNS = UPDGVNS(k add1/jump, ℓ mult2)).

in 1 hour. UDGVNS performed the best, slightly better than INCOP+TOULBAR2, improving by 1.7% (resp. 2.3%) on average after 1 hour (20 min).

5.3.4. Parallelization

Finally, in order to evaluate the impact of core numbers, we consider the anytime upper bound behavior of the parallel release: UPDGVNS using (k add1/jump, ℓ mult2) with $\ell_{min} = 3$, taken from the best strategies enlightened by UDGVNS. We made a comparison with CPLEX using 10 and 30 cores. Fig. 8 shows that CPLEX with 10 or 30 cores exhibits better anytime behavior than CPLEX using 1 core, but still being far from the other competitors (30 cores gave solutions 10% higher

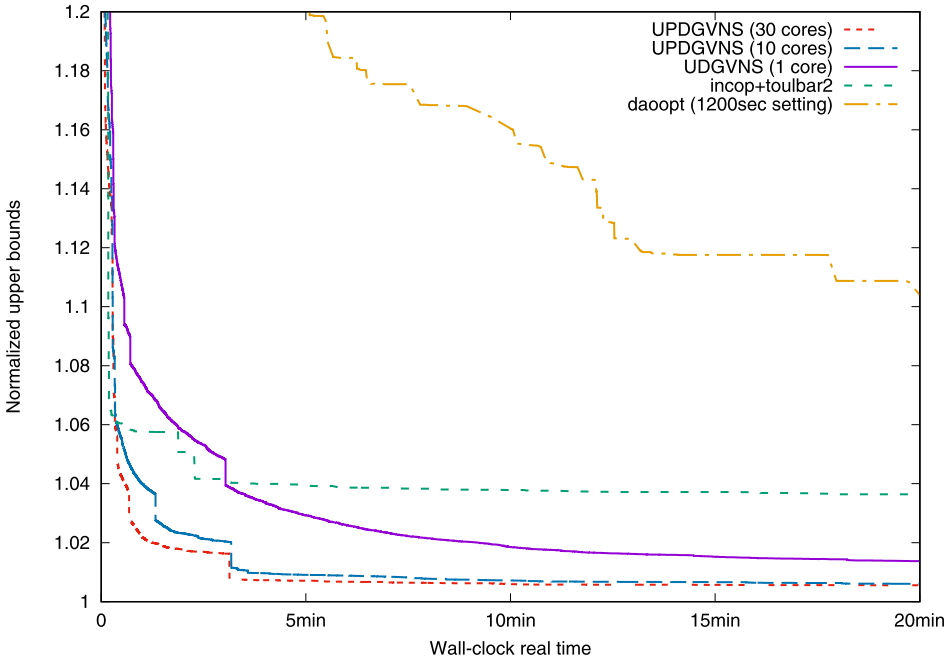

Fig. 9. Comparing the anytime behavior of parallel UPDGVNS.

Table 1
Best CPU time (in seconds) for sequential versions and best wall-clock time for multiple-core ones to find and prove optimality on Pedigree instances. A "–" means no proof of optimality in 1 hour (except DAOOPT with no time limit) (in parenthesis, unnormalized upper bound found after 1 hour).

Linkage *(optimum/worst solution)*	pedigree19 *(4625/21439)*	pedigree31 *(5258/166553)*	pedigree44 *(6651/104904)*	pedigree51 *(6406/629929)*
CPLEX (1 core)	790	59.3	6.35	36.23
CPLEX (10 cores)	191	9.00	2.48	9.43
CPLEX (30 cores)	**75**	**7.17**	**2.69**	**5.34**
DAOOPT (1 core)	375,110	16,238	95,830	101,788
DAOOPT (20 cores)	27,281	1,055	6,739	6,406
DAOOPT (100 cores)	7,492	201	1,799	1,578
UDGVNS (1 core)	*–(4949)*	*–(5258)*	*–(6722)*	*–(6406)*
UPDGVNS (10 cores)	*–(4762)*	3,341	*–(6651)*	*–(6406)*
UPDGVNS (30 cores)	*–(4626)*	1,775	*–(6651)*	*–(6406)*

than UPDGVNS after 1 hour). We could not compare with the parallel version of DAOOPT as it is based on a different cluster engine (*condor*) and it does not parallelize its initial phase.

Fig. 9 shows that UPDGVNS (with 10 or 30 cores) provides slightly better upper bounds than UDGVNS (using 1 core) in less than 20 min. The results seem to be, in average, poorly sensitive to the cores number, due to the fact that the 10-core curve is extremely close from the 30-core one. Fig. 9 also compares the anytime upper bound quality with those provided by single-core INCOP local solver followed by a hybrid best-first search in TOULBAR2 and by DAOOPT with options tuned for the 1200-second UAI 2014 challenge. The 10 and 30-core UPDGVNS curves converge quickly in less than 2 min. INCOP+TOULBAR2 quickly drops out around 1 min and never reaches the same quality level. UDGVNS converges slower but still going down after 20 min. DAOOPT (1200 sec. setting) gave results 10% in average worse than UPDGVNS with either 10 and 30 processors.

The trends observed over all instances are quite similar to those obtained on selected instances for each family. The only one exception is the Pedigree instances. Table 1 gives the solving time to find and prove optimality on UAI-Linkage category for U(P)DGVNS, CPLEX and DAOOPT (in parenthesis, unnormalized upper bound found after 1 hour). We report DAOOPT time from [48], obtained on a cluster of dual 2.67 GHz Intel Xeon X5650 6-core CPUs and 24 GB of RAM. We can see that UDGVNS and UPDGVNS (with 10 or 30 cores) are clearly dominated by CPLEX which exhibits better results. DAOOPT remains competitive on these instances but still being far from CPLEX in terms of CPU times.

Table 2
Comparing UDGVNS(k add1/jump, ℓ mult2), UPDGVNS(npr, k add1/jump, ℓ mult2) and TOULBAR2 in terms of optimality results within the 24-hours time limit. A '–' indicates that the corresponding solver failed to prove optimality.

Instance	Time (s)		
	UDGVNS	UPDGVNS	TOULBAR2
5dbl	1,828.27	791.16	783.18
3r8q	–	–	41,700.10
4bxp	–	–	4,261.67
1f00	–	–	9,749.00
1xaw	–	–	2,917.04
5e10	839.52	196.43	1,171.98
2gee	–	–	9,795.59
5e0z	416.12	172.96	999.66
3lf9	–	–	2,960.64
5eqz	–	–	41,813.00

6. Computational protein design

In Computational Protein Design (CPD), the challenge of identifying a protein that performs a given task is defined as the combinatorial optimization of a complex pairwise energy function over amino acid sequences and 3D geometry [49]. This holds great interest in medicine, synthetic biology, nanotechnologies and biotechnologies [50][51][52]. We used the CPD problem as a difficult benchmark to test our tree-decomposition based methods. For that, we generated 21 large instances with small treewidth selected from the Protein Data Bank[24] (PDB). These instances have been selected on the basis of 3D criteria described in a supplementary material.[25] The instances contain from $n = 107$ up to 292 variables with a maximum domain size from $d = 383$ to 450, and between $1,623$ and $6,208$ binary cost functions. The min-fill treewidth ranges from $w = 21$ to 68, resulting in small ratios of treewidth per number of variables, from $\frac{w}{n} = 0.16$ to 0.34.

6.1. Experimental protocol

We compared UDGVNS and UPDGVNS with VNS/LDS+CP and Fixed BackBone (a CPD dedicated algorithm provided by the Rosetta package). We also compared with TOULBAR2. We tested TOULBAR2 using Virtual Arc Consistency [31] in pre-processing and hybrid best-first search with tree decomposition (BTD-HBFS) using min-fill heuristic. Dead end elimination is turned off, according to [26] (the resulting command line is: TOULBAR2 -B=1 -A -DEE: -O=-3). The TOULBAR2 experiments use a 24-hour CPU time limit.

Concerning the VNS methods, in order to evaluate variability due to the random selection of neighborhoods, a set of 10 runs per instance with different seeds has been performed with a time limit of 1-hour per run. For the parallel strategy, the number of processes *npr* is set to 96 (i.e. maximum number of available processors). For UDGVNS and UPDGVNS, following the results observed in Section 5, we considered the following two settings for operators $+_k$ and $+_\ell$: (k add1, $\ell = 3$) which yields the best anytime performances and (k add1/jump, ℓ *mult*2) which offers the best compromise between both anytime performances and optimality proof. k_{min}, k_{max} and ℓ_{min} have been respectively set to 4, n (the total number of variables) and 3 (they correspond to the same parameter settings as those described in [23]). All computations were performed on a cluster of 96-core AMD Opteron 6174 at 2.2 GHz and 256 GB of RAM.

6.2. Experimental results

As for UAI instances, we evaluated the effectiveness of UDGVNS and UPDGVNS on CPD instances in terms of optimality proof (cf. Section 6.2.1) and solution quality vs. CPU time (cf. Section 6.2.2). We also study in Section 6.2.3 the impact of varying the number of processes for the parallel release UPDGVNS.

6.2.1. Optimality results

Our first set of experiments aims at evaluating the efficiency of UDGVNS and UPDGVNS in terms of optimality proof comparing with TOULBAR2. We used the setting (k add1/jump, ℓ *mult*2) with $\ell_{min} = 3$, taken from the best strategies enlightened by UDGVNS. Table 2 reports the CPU-times required by UDGVNS, UPDGVNS and TOULBAR2 to find and prove the optimum within the 3600-second time limit (24 hours for TOULBAR2). As we can see, TOULBAR2 clearly outperforms both VNS methods: UDGVNS and UPDGVNS were able to prove optimality only on 3 instances among the 9 ones closed by TOULBAR2. However, on two instances (5e10 and 5e0z), UDGVNS is up to 2.40 times faster than TOULBAR2 while UPDGVNS increases speeds by up to 5.96 times, despite the fact that BTD-HBFSbenefits from the lower bounds reported by HBFS in

[24] www.pdb.org.
[25] genoweb.toulouse.inra.fr/~degivry/evalgm/CFN/ProteinDesignUAI2017.

Table 3
Comparative evaluation on CPD instances. In bold instances completely solved by TOULBAR2. *ncl* is the number of clusters. ΔE is in Rosetta energy unit. It has been obtained by the difference of solution costs divided by the cost shift used during modeling, in this case 10^8.

Instance	*ncl*	VNS/LDS+CP		UDGVNS		UPDGVNS		FIXBB	TOULBAR2		Speed-up		
		Succ.	Time (s)	Succ.	Time (s)	Succ.	Time (s)	ΔE	Time (s)	ΔE	(1/2)	(1/3)	(2/3)
5dbl	87	10/10	2,762±67	10/10	249±7	10/10	62±7	0.28	783	0	11.09	44.27	3.98
5jdd	168	0/10	–	10/10	1,611±20	10/10	243±13	4.08	20,662	0.04	–	–	6.63
3r8q	157	0/10	–	10/10	1,246±292	10/10	206±8	4.19	12,762	0	–	–	6.03
4bxp	108	10/10	1,214±32	10/10	907±75	10/10	120±19	0.26	2,966	0	1.33	10.12	7.57
1f00	177	0/10	–	0/10	–	10/10	271±11	4.39	9,749	0	–	–	–
2x8x	131	10/10	3,312±112	4/10	2,731±1,065	10/10	215±11	4.16	69,213	3.61	1.21	15.41	12.71
1xaw	66	0/10	–	2/10	3,585±35	10/10	167±39	2.73	2,804	0	–	–	21.50
5e10	74	10/10	1,667±86	10/10	193±6	10/10	30±2	0.27	1,172	0	8.63	55.78	6.45
1dvo	82	10/10	940±23	10/10	352±5	10/10	75±2	2.90	34,143	0.18	2.67	12.47	4.66
1ytq	67	10/10	2,235±211	10/10	292±1	10/10	87±1	1.67	17,064	0.31	7.65	25.57	3.33
2af5	140	0/10	–	10/10	1,029±19	10/10	281±13	4.37	86,029	0.60	–	–	3.65
1ng2	86	10/10	1,066±70	10/10	583±46	10/10	115±4	1.14	38,731	5.93	1.82	9.27	5.07
3sz7	79	0/10	–	10/10	627±54	10/10	144±11	3.11	82,626	0.54	–	–	4.35
2gee	110	10/10	1,648±8	10/10	961±5	10/10	149±27	1.68	5,021	0	1.71	11.07	6.45
5e0z	73	10/10	622±16	10/10	310±2	10/10	45±8	0.16	999	0	2.0	13.90	6.92
1yz7	87	10/10	2,149±10	10/10	2,081±38	10/10	133±4	2.91	83,817	3.21	1.03	16.18	15.67
3e3v	91	0/10	–	10/10	867±207	10/10	111±10	2.57	81,575	0.15	–	–	7.81
3lf9	72	10/10	1,636±32	10/10	167±1	10/10	56±2	2.41	2,667	0	9.81	29.06	2.96
1is1	107	10/10	3,179±52	10/10	444±27	10/10	213±13	3.46	63,832	0.42	7.15	14.91	2.08
5eqz	89	10/10	1,850±49	10/10	528±8	10/10	95±3	2.20	12,768	0	3.50	19.44	5.54
4uos	118	10/10	2,305±988	1/10	3,380±661	10/10	167±2	5.21	58,590	17.87	0.68	13.79	20.23

–: TimeOut **(1): VNS/LDS+CP(k add1, $\ell = 3$)** **(2): UDGVNS(k add1, $\ell = 3$)** **(3): UPDGVNS(npr, k add1, $\ell = 3$)**

individual clusters to improve its anytime behavior and from the global pruning lower bounds of BTD. This greatly improves the overall performance of TOULBAR2 compared to VNS/LDS+CP.

6.2.2. CPU time and solution quality results

Our second set of experiments aims at evaluating VNS capability with respect to finding the optimal solution or a solution of better quality on instances for which optimal solutions are unavailable. For this aim, we selected (k add1, $\ell = 3$) as a setting for operators $+_k$ and $+_\ell$.

Table 3 shows a comparative evaluation of VNS methods with FIXBB and TOULBAR2. For each instance and each VNS method, we report the number of successful runs to reach the optimum (or the best known solution for unsolved instances) within a 3600-second time limit, the average CPU times (in seconds) over the 10 runs (for unsuccessful runs, the CPU time is set to the time limit) ± the standard deviation. The energy gap ΔE between the best VNS solution and the two external references FIXBB and TOULBAR2 are also given. For TOULBAR2, reported CPU times correspond to times to find an optimal solution (for solved instances) or the best one (for unsolved ones) within the 24-hour time limit.

A) VNS methods vs. FIXBB. Rosetta Modeling suite is one of the most popular software package used in the CPD field. It provides a Monte Carlo based Simulated Annealing algorithm called FIXBB. In this work, the best solutions exhibited by 1000 FIXBB cycles performed on each CPD instance have been used as base-line to compare solution quality of the solutions provided by VNS methods when TOULBAR2 BTD-HBFS fails to solve the instance.

The FIXBB CPU times are two orders of magnitude higher than the 1 h. time limit imposed for VNS evaluation. They are not reported as they exceed 100 hours in sequential mode, even if FIXBB cycles are independent and thus are easy to parallelize. As we can see in Table 3 (see column (4) ΔE), the solution quality of FIXBB is in all cases inferior to the best solution found by the VNS methods. The energetic gap ΔE between FIXBB solution and the best VNS overall solution ranges between +0.16 and +5.20 Rosetta Energy Unit (R.E.U). As shown in [26] such a level of energy difference can strongly impact the designed protein solution (i.e. the corresponding sequences of the two methods can be far in terms of Hamming distance).

B) VNS methods vs. TOULBAR2. The comparison between best solutions found by VNS methods and TOULBAR2 shows that, excepted VNS/LDS+CP method, both UDGVNS (except on 1f00) and UPDGVNS provide in all cases the same or even better solution than TOULBAR2 (see column (5) ΔE in Table 3). On 11 instances unsolved by TOULBAR2, UDGVNS and UPDGVNS always obtain solutions of better quality. The energetic gap ΔE in the worst case reaches 17.86 R.U.E. Concerning the number of successful runs reported over the 10 runs, VNS/LDS+CP seems less robust respectively than UDGVNS and the parallel release UPDGVNS. This last one provides in all cases the best solution over all. Table 3 also compares the VNS methods in terms of speedups. We observe that speedup values are fluctuating from one instance to another, very likely due to the tree decomposition resulting from the 3D shape of each instance. For VNS/LDS+CP and UDGVNS, it ranges between 0.68 and 11.09 over the 14 instances solved by both methods. As expected, when tree decomposition and parallelization are used, not only the speed of resolution increases but the reliability too (speedup values between 9.27 and 55.78). Moreover,

Table 4
Comparing speed-ups of VNS methods with (k `add1`, $\ell = 3$) strategy to obtain the best solution computed by TOULBAR2 within the 3600-second time limit on solved CPD instances. A '–' indicates that the corresponding solver was not able to compute a solution of equal/better quality than TOULBAR2.

Instance	Speed-up		
	(TOULBAR2/VNS/LDS+CP)	(TOULBAR2/UDGVNS)	(TOULBAR2/UPDGVNS)
5dbl	0.28	3.14	12.55
3r8q	-	33.46	202.07
4bxp	3.51	4.69	35.56
1f00	-	17.02	35.93
1xaw	-	2.92	17.49
5e10	0.70	6.07	39.22
2gee	5.94	10.19	65.82
5e0z	1.60	3.22	22.36
3lf9	-	17.75	52.58
5eqz	22.61	79.18	439.48

Table 5
Comparing CPU times spent by VNS methods with (k `add1`, $\ell = 3$) strategy to obtain the best solution computed by TOULBAR2 within the 3600-second time limit on unsolved CPD instances. A '–' indicates that the corresponding solver was not able to compute a solution of equal/better quality than TOULBAR2.

Instance	Time (s)			Speed-up		
	VNS/LDS+CP	UDGVNS	UPDGVNS	(TOULBAR2 /VNS/LDS+CP)	(TOULBAR2 /UDGVNS)	(TOULBAR2 /UPDGVNS)
5jdd	-	1,611.39±20	235.52±11	-	12.82	87.73
2x8x	2,070.49±62	883.50±56	176.48±2	33.42	78.33	392.18
1dvo	885.47±21	264.56±3	68.83±2	38.55	129.05	496.04
1ytq	2,194.63±210	291.93±1	85.19±1	7.77	58.45	200.30
2af5	-	815.62±13	254.68±10	-	105.47	337.79
1ng2	530.56±3	304.94±2	83.88±3	73.00	127.01	461.74
3sz7	3,029.44±566	565.29±40	125.62±7	27.27	146.16	657.74
1yz7	970.24±4	419.47±3	115.22±2	86.38	199.81	727.45
3e3v	2,332.23±46	288.46±2	93.44±2	34.97	282.79	873.01
1is1	2,803.27±45	437.44±28	208.47±12	22.77	145.92	306.19
4uos	460.73±3	420.18±2	153.11±1	127.16	139.43	382.66

Table 6
Impact of the number of processes on the parallelization.

Instance	UDGVNS		UPDGVNS(10)		UPDGVNS(30)		UPDGVNS(96)		Speed-up				
	Succ.	Time (s)	Succ.	Time (s)	Succ.	Time (s)	succ.	Time (s)	(1/2)	(1/3)	(1/4)	(2/4)	(3/4)
5dbl	10/10	249±7	10/10	127±23	10/10	88±12	10/10	62±7	1.96	2.83	3.98	2.02	1.40
5jdd	10/10	1,611±20	10/10	603±67	10/10	342±22	10/10	243±13	2.67	4.71	6.63	2.47	1.40
3r8q	10/10	1,246±292	10/10	508±61	10/10	301±28	10/10	206±8	2.45	4.13	6.03	2.46	1.46
4bxp	10/10	907±75	10/10	301±39	10/10	186±26	10/10	120±19	3.01	4.88	7.57	2.50	1.55
1f00	0/10	–	10/10	589±75	10/10	361±47	10/10	271±11	–	–	–	2.17	1.32
2x8x	4/10	2,731±1,065	10/10	600±198	10/10	322±44	10/10	215±11	4.55	8.49	12.71	2.79	1.49
1xaw	2/10	3,585±35	10/10	434±162	10/10	219±56	10/10	167±39	8.25	16.37	21.50	2.60	1.31
5e10	10/10	193±6	10/10	59±15	10/10	35±6	10/10	30±2	3.26	5.54	6.45	1.97	1.16
1dvo	10/10	352±5	10/10	134±18	10/10	102±6	10/10	75±2	2.61	3.44	4.66	1.78	1.35
1ytq	10/10	292±1	10/10	148±12	10/10	102±5	10/10	87±1	1.97	2.87	3.33	1.68	1.16
2af5	10/10	1,029±19	10/10	529±47	10/10	362±20	10/10	281±13	1.94	2.84	3.65	1.87	1.28
1ng2	10/10	583±46	10/10	223±30	10/10	160±20	10/10	115±4	2.61	3.64	5.07	1.94	1.39
3sz7	10/10	627±54	10/10	392±39	10/10	221±20	10/10	144±11	1.60	2.83	4.35	2.71	1.53
2gee	10/10	961±5	10/10	252±28	10/10	203±44	10/10	149±27	3.81	4.72	6.45	1.69	1.36
5e0z	10/10	310±2	10/10	93±19	10/10	65±10	10/10	45±8	3.33	4.74	6.92	2.07	1.45
1yz7	10/10	2,081±38	10/10	242±19	10/10	174±24	10/10	133±4	8.59	11.98	15.67	1.82	1.30
3e3v	10/10	867±207	10/10	265±42	10/10	170±19	10/10	111±10	3.26	5.08	7.81	2.39	1.53
3lf9	10/10	167±1	10/10	100±17	10/10	71±4	10/10	56±2	1.66	2.36	2.96	1.78	1.25
1is1	10/10	444±27	10/10	454±87	10/10	317±30	10/10	213±13	.97	1.40	2.08	2.12	1.48
5eqz	10/10	528±8	10/10	187±22	10/10	120±4	10/10	95±3	2.82	4.40	5.54	1.96	1.26
4uos	1/10	3,380±661	10/10	217±18	10/10	189±6	10/10	167±2	15.58	17.91	20.23	1.29	1.12

–: TimeOut (1): UDGVNS (2): UDGVNS(10) (3): UPDGVNS(30) (4): UPDGVNS(96)

the comparison between UDGVNS and UPDGVNS shows significant accelerations (between 2.08 and 21.50), thus confirming the practical interest of parallelization in addition to the exploitation of problem decomposition.

C) Comparing anytime performances of VNS methods. We have also investigated the anytime performances of three VNS methods by reporting the CPU-times required within the time limit of 1 hour to reach a solution of equal quality computed

by TOULBAR2 within the 3600-second time limit. For solved instances, according to details in Table 4, both UDGVNS and UPDGVNS find optimal solutions more quickly than TOULBAR2. For UDGVNS, speedup values range from 2.92 to 79.18 with a mean of 17.76 over all the solved instances. For UPDGVNS, the ratio in terms of speedup is greatly amplified (between 12.55 and 439.48 with a mean of 92.30 over all the solved instances). On the 11 opened instances, results (summarized in Table 5) show a clear ordering in terms of CPU times across different solvers, from TOULBAR2, VNS/LDS+CP, UDGVNS, and UPDGVNS. The speedup values are significantly improved, in particular with UPDGVNS (between 87.73 and 873.01 with a mean of 447, 53 over all the unsolved instances). These results confirm the superiority of VNS methods in terms of anytime performance as compared to TOULBAR2.

One putative explanation of the observed performance ordering between VNS methods may be the problem exploration coverage with the picked neighborhoods during search. Indeed, respectively with 1, 034 and 597 the average total number of neighborhoods explored during search for the 21 CPD instances is in average 1.73 higher in VNS/LDS+CP than UDGVNS. Accordingly, tree decomposition picks more pertinent neighborhoods than VNS/LDS+CP and seems to increase the probability for full problem coverage, which can be explained by the decreasing of possible combinations as a consequence of the partition in clusters of variables. Besides, parallelization is one way to increase the coverage probability, because it is a simple way to increase the number of processed neighborhoods. Consequently, in practice, with 3, 320 neighborhoods in average, UPDGVNS explores in 5.56 times more subproblems than the corresponding sequential version. This fact can be an explanation of the good quality of the observed results.

6.2.3. Parallelization

For the last experiment, we analyzed the performance of our parallel algorithm by measuring the speed-up on varying the number of processes. We consider the number of successful runs as well as the average CPU times $\pm$ the standard deviation (over the 10 runs) of the parallel release UPDGVNS (using k add1 with $\ell = 3$) to reach the optimum (or the best found solution for unsolved instances) within a time limit of 1-hour per run. We compare with UDGVNS with (k add1, $\ell = 3$). We set *npr* to 10, 30 and 96 respectively (including the master process).

As can be seen from Table 6, UPDGVNS with 10, 30 or 96 processors, are able to obtain better results faster than UDGVNS (using one core). Table 6 also reports the speed-up values for different numbers of processes. Comparing to 10-processes, the improvements to UDGVNS yield a speed-up of 1.6 to 15.58 (3.85 on average). Moreover, with the increase of the number of processes the gains in terms of CPU times are remarkably amplified. In the case of 30 processes the speed-up is 5.75 on average and for 96 processes it is 7.68 compared to the results of UDGVNS. These results show that our parallel release on CPD instances is less sensitive to the communication overhead when increasing the number of processes and it does not impact the overall efficiency of our approach.

7. Conclusion and perspectives

In this paper we proposed a unified view of VNS methods including various LDS and neighborhood evolution strategies. Experiments performed on difficult instances, coming from a large graphical model benchmark, showed that our hybrid method has a much better anytime behavior than existing tree search methods and still being competitive for proving optimality. UDGVNS takes advantage of the good convergence properties of DGVNS and proves optimality in many cases. On structured or unstructured problems of large sizes, like CPD, UDGVNS obtains solutions of (very) good quality, thus outperforming the state-of-the-art FIXBB Rosetta Modeling software package used in the CPD field. We further proposed a parallel version of our method improving its anytime behavior. It remains as future work to manage dynamically the tree-decomposition associated to the instance to solve. Another promising research direction is to use machine learning techniques to identify the best decompositions to be used for practical solving [53].

Declaration of competing interest

We wish to confirm that there are no known conflicts of interest associated with this publication and there has been no significant financial support for this work that could have influenced its outcome.

Acknowledgements

This work has been partially funded by the french “Agence Nationale de la Recherche”, reference ANR-16-C40-0028. We are grateful to the genotoul bioinformatics platform Toulouse Midi-Pyrenees (Bioinfo Genotoul) and the high performance center of Cerist-Algiers in Algeria for providing computing and storage resources. We thank Mathieu Fontaine for his contribution to the code of DGVNS.

References

[1] D. Koller, N. Friedman, Probabilistic Graphical Models: Principles and Techniques, The MIT Press, 2009.
[2] S. Shimony, Finding MAPs for belief networks is NP-hard, AI Commun. 68 (1994) 399–410.
[3] R. Marinescu, R. Dechter, Memory intensive and/or search for combinatorial optimization in graphical models, AI Commun. 173 (16–17) (2009) 1492–1524.

[4] L. Otten, R. Dechter, Anytime AND/OR depth-first search for combinatorial optimization, AI Commun. 25 (3) (2012) 211–227.
[5] D. Allouche, S. de Givry, G. Katsirelos, T. Schiex, M. Zytnicki, Anytime hybrid best-first search with tree decomposition for weighted CSP, in: Proc. of CP, 2015, pp. 12–28.
[6] V. Kolmogorov, Convergent tree-reweighted message passing for energy minimization, IEEE Trans. Pattern Anal. Mach. Intell. 28 (10) (2006) 1568–1583.
[7] M. Wainwright, M. Jordan, Graphical models, exponential families, and variational inference, Found. Trends Mach. Learn. 1 (1–2) (2008) 1–305.
[8] D. Sontag, T. Meltzer, A. Globerson, T. Jaakkola, Y. Weiss, Tightening LP relaxations for MAP using message-passing, in: Proc. of UAI, 2008, pp. 503–510.
[9] D. Sontag, D. Choe, Y. Li, Efficiently searching for frustrated cycles in MAP inference, in: Proc. of UAI, 2012, pp. 795–804.
[10] H. Wang, D. Koller, Subproblem-tree calibration: a unified approach to max-product message passing, in: Proc. of ICML, 2013, pp. 190–198.
[11] J. Park, Using weighted MAX-SAT engines to solve MPE, in: Proc. of AAAI, 2002, pp. 682–687.
[12] F. Hutter, H. Hoos, T. Stützle, Efficient stochastic local search for MPE solving, in: Proc. of IJCAI, 2005, pp. 169–174.
[13] O. Mengshoel, D. Wilkins, D. Roth, Initialization and restart in stochastic local search: computing a most probable explanation in Bayesian networks, IEEE Trans. Knowl. Data Eng. 23 (2) (2011) 235–247.
[14] R. Marinescu, K. Kask, R. Dechter, Systematic vs. non-systematic algorithms for solving the MPE task, in: Proc. of UAI, 2003, pp. 394–402.
[15] P. Shaw, Using constraint programming and local search methods to solve vehicle routing problems, in: Proc. of CP, 1998, pp. 417–431.
[16] L. Perron, P. Shaw, V. Furnon, Propagation guided large neighborhood search, in: Proc. of CP, 2004, pp. 468–481.
[17] M. Lombardi, P. Schaus, Cost impact guided lns, in: Proc. of Integration of AI and OR Techniques in Constraint Programming, 2014, pp. 293–300.
[18] J.J. Dekker, M.G. de la Banda, A. Schutt, P.J. Stuckey, G. Tack, Solver-independent large neighbourhood search, in: Proc. of CP, 2018, pp. 81–98.
[19] E. Demirovic, G. Chu, P.J. Stuckey, Solution-based phase saving for CP: a value-selection heuristic to simulate local search behavior in complete solvers, in: Proc. of CP, 2018, pp. 99–108.
[20] N. Mladenović, P. Hansen, Variable neighborhood search, Comput. Oper. Res. 24 (11) (1997) 1097–1100.
[21] S. Loudni, P. Boizumault, Solving constraint optimization problems in anytime contexts, in: Proc. of IJCAI, 2003, pp. 251–256.
[22] W. Harvey, M. Ginsberg, Limited discrepancy search, in: Proc. of IJCAI, 1995, pp. 607–615.
[23] M. Fontaine, S. Loudni, P. Boizumault, Exploiting tree decomposition for guiding neighborhoods exploration for VNS, RAIRO Oper. Res. 47 (2) (2013) 91–123.
[24] T. Davidovic, T. Crainic, MPI parallelization of variable neighborhood search, Electron. Notes Discrete Math. 39 (2012) 241–248.
[25] A. Ouali, S. Loudni, L. Loukil, P. Boizumault, Y. Lebbah, Replicated parallel strategies for decomposition guided VNS, Electron. Notes Discrete Math. 47 (2015) 93–100.
[26] D. Simoncini, D. Allouche, S. de Givry, C. Delmas, S. Barbe, T. Schiex, Guaranteed discrete energy optimization on large protein design problems, J. Chem. Theory Comput. 11 (12) (2015) 5980–5989.
[27] P. Meseguer, F. Rossi, T. Schiex, Soft constraints processing, in: Handbook of Constraint Programming, Elsevier, 2006, pp. 279–326, Ch. 9.
[28] B. Hurley, B. O'Sullivan, D. Allouche, G. Katsirelos, T. Schiex, M. Zytnicki, S. de Givry, Multi-language evaluation of exact solvers in graphical model discrete optimization, Constraints 21 (3) (2016) 413–434.
[29] R. Dechter, I. Rish, Mini-buckets: a general scheme for bounded inference, J. ACM 50 (2) (2003) 107–153.
[30] J. Larrosa, S. de Givry, F. Heras, M. Zytnicki, Existential arc consistency: getting closer to full arc consistency in weighted CSPs, in: Proc. of IJCAI, 2005, pp. 84–89.
[31] M. Cooper, S. de Givry, M. Sanchez, T. Schiex, M. Zytnicki, T. Werner, Soft arc consistency revisited, AI Commun. 174 (2010) 449–478.
[32] W. Karoui, M.-J. Huguet, P. Lopez, W. Naanaa, Yields: a yet improved limited discrepancy search for csps, in: Proc. of Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 2007, pp. 99–111.
[33] P. Prosser, C. Unsworth, Limited discrepancy search revisited, ACM J. Exp. Algorithmics 16 (2011) 1.6:1.1–1.6:1.18.
[34] H. Bodlaender, A. Koster, Treewidth computations I. Upper bounds, Inf. Comput. 208 (3) (2010) 259–275.
[35] H. Bodlaender, A. Koster, F. Van den Eijkhof, Preprocessing rules for triangulation of probabilistic networks, Comput. Intell. 21 (3) (2005) 286–305.
[36] S. Arnborg, et al., Complexity of finding embeddings in a k-tree, SIAM J. Algebraic Discrete Methods 8 (1987) 277–284.
[37] U. Kjærulff, Triangulation of Graphs – Algorithms Giving Small Total State Space, Tech. rep., Aalborg University, 1990.
[38] R.E. Tarjan, et al., Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs, SIAM J. Comput. 13 (3) (1984) 566–579.
[39] M. Luby, A. Sinclair, D. Zuckerman, Optimal speedup of Las Vegas algorithms, in: Proc. of TCS, 1993, pp. 128–133.
[40] F. Boussemart, F. Hemery, C. Lecoutre, L. Sais, Boosting systematic search by weighting constraints, in: Proc. of ECAI, 2004, pp. 146–150.
[41] J. Kappes, B. Andres, F. Hamprecht, C. Schnörr, S. Nowozin, D. Batra, S. Kim, B. Kausler, T. Kröger, J. Lellmann, N. Komodakis, B. Savchynskyy, C. Rother, A comparative study of modern inference techniques for structured discrete energy minimization problems, Int. J. Comput. Vis. 115 (2) (2015) 155–184.
[42] A. Favier, S. Givry, A. Legarra, T. Schiex, Pairwise decomposition for combinatorial optim. in graphical models, in: Proc. of IJCAI, 2011, pp. 2126–2132.
[43] S. de Givry, S. Prestwich, B. O'Sullivan, Dead-end elimination for weighted CSP, in: Proc. of CP, 2013, pp. 263–272.
[44] A. Ihler, N. Flerova, R. Dechter, L. Otten, Join-graph based cost-shifting schemes, in: Proc. of UAI, 2012, pp. 397–406.
[45] L. Otten, A. Ihler, K. Kask, R. Dechter, Winning the PASCAL 2011 MAP challenge with enhanced AND/OR branch-and-bound, in: Proc. of NIPS Workshop DISCML, 2012.
[46] B. Neveu, G. Trombettoni, F. Glover, ID walk: a candidate list strategy with a simple diversification device, in: Proc. of CP, 2004, pp. 423–437.
[47] J.M. Mooij, libDAI: a free and open source C++ library for discrete approximate inference in graphical models, J. Mach. Learn. Res. 11 (2010) 2169–2173.
[48] L. Otten, R. Dechter, And/or branch-and-bound on a computational grid, J. Artif. Intell. Res. 59 (2017) 351–435.
[49] D. Allouche, J. Davies, S. de Givry, G. Katsirelos, T. Schiex, S. Traoré, I. André, S. Barbe, S. Prestwich, B. O'Sullivan, Computational protein design as an optimization problem, AI Commun. 212 (2014) 59–79.
[50] P.-S. Huang, S.E. Boyken, D. Baker, The coming of age of de novo protein design, Nature 537 (2016) 320–327.
[51] S.M. Lippow, B. Tidor, Progress in computational protein design, in: Protein Technologies/Systems Biology, Curr. Opin. Biorecovery 18 (4) (2007) 305–311, https://doi.org/10.1016/j.copbio.2007.04.009, http://www.sciencedirect.com/science/article/pii/S0958166907000778.
[52] D.L. Trudeau, D.S. Tawfik, Protein engineers turned evolutionists—the quest for the optimal starting point, in: Pharmaceutical Biotechnology, Curr. Opin. Biotechnol. 60 (2019) 46–52, https://doi.org/10.1016/j.copbio.2018.12.002, http://www.sciencedirect.com/science/article/pii/S095816691830209X.
[53] M. Abseher, N. Musliu, S. Woltran, Improving the efficiency of dynamic programming on tree decompositions via machine learning, J. Artif. Intell. Res. 58 (2017) 829–858.

Glossary

BTD-HBFS: Backtrack Tree Decomposition - Hybrid Best First Search
CFN: Cost Function Network
CPD: Computational Protein Design
CVPR: Computer Vision and Pattern Recognition
DFBB: Depth-First Branch and Bound

DGVNS: Decomposition Guided VNS
EDAC: Existential Directional Arc consistency
FIXBB: Fixed BackBone
ILDS: Iterative LDS
LDS: Limited Discrepancy Search
LDSr: Restricted LDS
LNS: Large Neighborhood Search
MCS: Maximum Cardinality Search
min-fill: Minimum Fill-in
PIC: Probabilistic Inference Challenge
UAI: Uncertainty in Artificial Intelligence
UDGVNS: Unified Decomposition Guided VNS
UPDGVNS: Unified Parallel DGVNS
VAC: Virtual Arc Consistency
VNS: Variable Neighborhood Search
VNS/LDS+CP: VNS/LDS + Constraint Programming