Decomposition During Search

Martin Mann[1,⋆], Guido Tack[2], and Sebastian Will[1,⋆⋆]

[1] Bioinformatics, Albert-Ludwigs-University Freiburg, Germany
{mmann,will}@informatik.uni-freiburg.de
[2] Programming Systems Lab, Saarland University, Germany
tack@ps.uni-sb.de

Abstract. We describe *decomposition during search* (DDS) as a novel search algorithm for counting the solutions of a CSP. DDS enhances standard tree search by dynamically decomposing sub-problems into independent problems, avoiding redundant work. The paper provides formal definitions and analysis of the introduced method. We integrate DDS into a modern constraint programming system, using Gecode as an example. Two applications, graph coloring and protein structure prediction, show the potential for huge benefits of DDS in practice.

1 Introduction

Only recently, counting and exhaustive enumeration of solutions of a CSP gained a lot of interest [1,8,13,18]. This is partly due to the higher complexity of counting compared to deciding satisfiability [16]. Notwithstanding the complexity, there is demand for solution counting in real applications. For instance, in Bioinformatics the number of optimal protein structures can be determined using CP [2], which is of importance for the study of protein energy landscapes [9,17,20] and originally motivated the presented work.

In general, solving CSPs is NP-complete. However, the counting of CSP solutions is an even harder problem in the complexity class #P. This class was defined by Valiant [19] as the class of counting problems that can be computed in nondeterministic polynomial time. Standard solving methods in constraint programming like Depth-First Search (DFS) combined with constraint propagation have proven well suited for finding *one* solution. This paper shows that DFS, however, leaves room for saving redundant work when counting *all* solutions.

Our new method identifies independent partial problems during search, by computing the connected components of the problem's associated constraint graph. The problem can then be decomposed into its independent partial problems. Separate counting in the partial problems still allows to infer the number of solutions of the complete problem, or to represent the complete solution space.

Instead of *statically* exploiting properties only of the initial constraint graph, our method *dynamically* analyzes the constraint graphs at each node of the

⋆ Supported by the EU project EMBIO (EC contract number 012835)
⋆⋆ Supported by the EU Network of Excellence REWERSE (project number 506779)

search tree. In particular, if the initial constraint graph is heavily connected, static methods have no impact.

Decomposing into connected components and, more generally, utilizing the special structure of the constraint graph has been discussed already for a long time. Originally, Freuder et al. [10] proposed statically decomposing a CSP and solving the partial problems independently. As a more recent example, Dechter et al. [8] introduced AND/OR search for solution counting, again relying on static analysis of the constraint graph. To our knowledge, dynamic decomposition was discussed more thoroughly only for special cases. Bayardo et al. [13] count models of 3-SAT by a Davis-Putnam procedure [5] extended with dynamic decomposition. Similar ideas are discussed for SAT-solvers in [4,15].

Contribution and overview. Our main contribution is the formal description and analysis of DDS in a CP context and its empirical evaluation, using a well integrated and competitive implementation of DDS for the Gecode library [11].

The paper starts by giving the most fundamental definitions in Sec. 2. In Sec. 3, after rephrasing the standard DFS method, we introduce DDS and discuss how properties of the constraint graph can be used to implement it. Furthermore, we show how DDS gives rise to specialized branching heuristics, and that DDS is compatible with global constraints.

In Sec. 4, we analytically compare DDS to DFS in terms of search tree size. We show that the search tree size of DDS is bounded by the one of DFS under certain conditions. Although reduction of the tree size cannot be guaranteed in general, our analysis gives insights into the method's nature.

While the presentation up to Sec. 4 stays independent of a particular constraint programming system, Sec. 5 discusses implementation details of integrating DDS into Gecode.

Using our Gecode implementation, we study the practical impact of DDS in Sec. 6. We count solutions for random instances of two important CSPs, graph coloring and protein structure prediction. Both examples are hard counting problems in #P. We report high average speedups and reductions in search tree size, even using our prototype implementation. We finish with a resume and outlook to future work in Sec. 7.

2 Preliminaries

CSPs. A *Constraint Satisfaction Problem (CSP)* P is a triple $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, where $\mathcal{X}$ is the set of variables, the domain $\mathcal{D}$ a function of variables to the powerset of a fixed universe $\mathcal{U}$, and $\mathcal{C}$ a set of constraints. An n-ary *constraint* c is defined by the tuple of its n variables $\mathrm{vars}(c)$ and a set of n-tuples $\mathrm{sol}(c) \subset \mathcal{P}(\mathcal{U})^n$. We feel free to interpret $\mathrm{vars}(c)$ as the set of variables of c.

Assignments and solutions. An *assignment* A of variables $\mathcal{X} = \{X_1, \ldots, X_n\}$ is a function $\mathcal{X} \to \mathcal{U}$. An assignment A is *solution of a constraint* c with $\mathrm{vars}(c) = (X_1, \ldots, X_n)$ iff $(A(X_1), \ldots, A(X_n)) \in \mathrm{sol}(c)$. An assignment is a *solution of a CSP* $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, iff for all $X \in \mathcal{X} : A(X) \in \mathcal{D}(X)$ and A is a solution of all constraints $c \in \mathcal{C}$. The *set of solutions of a CSP* P is denoted by $\mathrm{sol}(P)$.

Based on these definitions, some important properties of CSPs can be defined. A *CSP* $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ *is solved* iff $\forall X \in \mathcal{X} : |\mathcal{D}(X)| = 1$ and $\mathrm{sol}(P) \neq \emptyset$. *P is failed* iff $\exists X \in \mathcal{X} : \mathcal{D}(X) = \emptyset$. *P is satisfiable* iff $\mathrm{sol}(P) \neq \emptyset$.

Two CSPs $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ and $P' = (\mathcal{X}', \mathcal{D}', \mathcal{C}')$ are *equivalent*, we write $P \equiv P'$, iff $\mathrm{sol}(P)$=$\mathrm{sol}(P')$. *P and P′ are equal*, we write $P = P'$, iff $\mathcal{X} = \mathcal{X}'$, $\forall X \in \mathcal{X} : \mathcal{D}(X) = \mathcal{D}'(X)$, and $\mathcal{C} = \mathcal{C}'$. P' is *stronger than* P, we write $P' \sqsubseteq P$, iff $P' \equiv P$ and $\forall X \in \mathcal{X} : \mathcal{D}'(X) \subseteq \mathcal{D}(X)$.

Constraint propagation. We assume a unique mapping from constraints to propagators, which are contracting and monotonic functions that maintain the solutions of the constraint. We assume a function propagate that computes the mutual fixpoint of all propagators. Propagation maps a CSP $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ to a CSP $\mathrm{propagate}(P) = (\mathcal{X}, \mathcal{D}', \mathcal{C}')$ such that $\mathrm{propagate}(P) \sqsubseteq P$. We use the short-cut P_p for denoting $\mathrm{propagate}(P)$.

Branching and search trees. To find solutions of a propagated but unsolved CSP, a branching is neccessary.

Definition 1 (Branching). *A* (binary) branching B *consists of two functions* lb *and* rb *from a CSP to a CSP.* $\mathrm{lb}^B(P)$ *and* $\mathrm{rb}^B(P)$ *are only defined for a CSP P that is neither failed nor solved.*

A branching B is *complete* iff $\mathrm{sol}(P) = \mathrm{sol}(\mathrm{lb}^B(P)) \cup \mathrm{sol}(\mathrm{rb}^B(P))$. It is *non-overlapping* iff $\mathrm{sol}(\mathrm{lb}^B(P)) \cap \mathrm{sol}(\mathrm{rb}^B(P)) = \emptyset$. A branching B is *strict* iff for any P that is neither failed nor solved, $\mathrm{propagate}(\mathrm{lb}^B(P)) \sqsubseteq P$ and $\mathrm{propagate}(\mathrm{rb}^B(P)) \sqsubseteq P$. In the following we consider only complete, strict and non-overlapping branchings.

A branching B in combination with propagation specifies a search tree for a CSP P. A *binary tree T* with nodes in $\mathcal{V}$ is recursively defined to be either a leave $T = x \in \mathcal{V}$ or a tuple $T = (x, l, r)$ where $x \in \mathcal{V}$ is the *root of T*, the tree l the *left sub-tree of T* and the tree r is the *right sub-tree of T*. A *search tree* is a finite binary tree, where each node is a CSP.

The search tree $\mathrm{bst}^B(P)$ is defined recursively by

$$\mathrm{bst}^B(P) = \begin{cases} P_\mathrm{p} & \text{if } P_\mathrm{p} \text{ is solved or failed} \\ (P_\mathrm{p}, \mathrm{bst}^B(\mathrm{lb}^B(P_\mathrm{p})), \mathrm{bst}^B(\mathrm{rb}^B(P_\mathrm{p}))) & \text{otherwise,} \end{cases}$$

An example for a binary search tree is given in Fig. 1.

Constraint graph. The *constraint graph of a CSP* $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ is a hypergraph $G_P = (V, E)$, where $V = \mathcal{X}$ and $E = \{\mathrm{vars}(c) \mid c \in \mathcal{C}\}$.

3 Decomposition During Search

In this section, first we recapitulate the counting of solutions of a CSP by simple depth-first search (DFS). Then, we show how the independence of parts of a CSP leads to redundant work of a DFS engine. Eliminating this redundant work is the basic idea of our central contribution, *decomposition during search*, which is presented in the rest of this section.

Algorithm 1 Counting by DFS

```
1: function CDFS($\mathcal{X}, \mathcal{D}, \mathcal{C}$)
2:     $(\mathcal{X}, \mathcal{D}', \mathcal{C}') \leftarrow$ Propagate($\mathcal{X}, \mathcal{D}, \mathcal{C}$)
3:     if IsFailed($\mathcal{X}, \mathcal{D}', \mathcal{C}'$) then return 0
4:     else if IsSolved($\mathcal{X}, \mathcal{D}', \mathcal{C}'$) then return 1
5:     else                                                              ▷ branching
6:         return CDFS(LeftBranch($\mathcal{X}, \mathcal{D}', \mathcal{C}'$)) + CDFS(RightBranch($\mathcal{X}, \mathcal{D}', \mathcal{C}'$))
7:     end if
8: end function
```

3.1 Counting DFS

The usual approach to count solutions of a CSP is by DFS in combination with constraint propagation. Algorithm 1 presents a recursive formulation, which we call Counting Depth-First Search (CDFS).

In our formulation, CDFS$(\mathcal{X}, \mathcal{D}, \mathcal{C})$ yields the number of solutions of the CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$. Note that the function Propagate performs full propagation of constraints to the domains (also, entailed constraints of $\mathcal{C}$ are removed in $\mathcal{C}'$) in line 2. The tests whether the propagated CSP $(\mathcal{X}, \mathcal{D}', \mathcal{C}')$ is failed or solved are in line 3 and 4. If the CSP is neither failed nor solved, the branching is applied in line 6. The functions LeftBranch and RightBranch yield the sub-problems generated by the branching (see lb^B and rb^B in Def. 1). Finally, the solution count of each sub-problem adds to the total number of solutions in line 7, because we consider only complete, strict and non-overlapping branchings.

A common branching strategy for solving CSPs is described by the variable-value branching.

Definition 2 (Variable-Value Branchings). *A* variable-value branching B *is a (binary) branching defined by associated functions* varsel^B *and* valsel^B, *where* varsel^B *assigns to an unsolved CSP* $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ *a variable* $X \in \mathcal{X}$, *where* $|\mathcal{D}(X)| > 1$, *and* valsel^B *assigns to a variable* X *a value in* $\mathcal{D}(X)$ *and furthermore by a relation* $\theta^B \in \{=, \leq, \geq, <, >\}$, *such that*

$$\mathrm{lb}^B(P) = (\mathcal{X}, \mathcal{D}, \mathcal{C} \cup \{X\theta^B V\}) \text{ and } \mathrm{rb}^B(P) = (\mathcal{X}, \mathcal{D}, \mathcal{C} \cup \{\neg X\theta^B V\}),$$

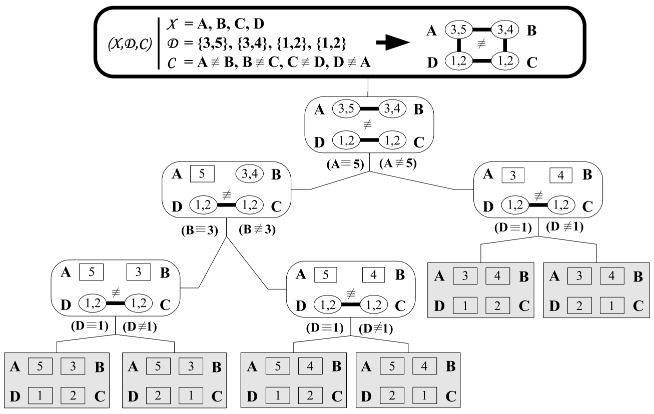
where $X = \mathrm{varsel}^B(P)$ *and* $V = \mathrm{valsel}^B(X)$.

Proposition 1. *Any variable-value branching B is complete, strict and non-overlapping.*

The variable selection heuristic, given by varsel^B, is essential for the quality of a branching and will be discussed in Sec. 3.3.

Figure 1 presents an example CSP and a corresponding search tree for CDFS solution counting. Each node corresponds to a propagated sub-problem of the initial problem and is visualized as a constraint graph.

Even the tiny example in Fig. 1 demonstrates that CDFS performs redundant work for this problem: The partial problem on the variables C and D is solved redundantly for each solution of the partial problem on A and B. We say that



$(\mathcal{X},\mathcal{D},\mathcal{C})$
$\mathcal{X}$ = A, B, C, D
$\mathcal{D}$ = {3,5}, {3,4}, {1,2}, {1,2}
$\mathcal{C}$ = A ≠ B, B ≠ C, C ≠ D, D ≠ A
(A=5)
(A≠5)
(B=3)
(B≠3)
(D=1)
(D≠1)

Fig. 1. DFS search tree traversed by CDFS.

$\{A, B\}$ and $\{C, D\}$ are *independent* sets of variables, and that we can solve them individually by *restricting* a CSP to a subset of its variables.

Restriction and independence. The *restriction of a function f to a set X*, $f_{|X}$, is defined as usual. Restriction on sets is understood elementwise. The restriction of a CSP $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ to a set of variables $\mathcal{X}' \subseteq \mathcal{X}$ is defined by $P_{|\mathcal{X}'} = (\mathcal{X}', \mathcal{D}_{|\mathcal{X}'}, \mathcal{C}_{|\mathcal{X}'})$, where $\mathcal{C}_{|\mathcal{X}'} = \{c \in \mathcal{C} \mid \text{vars}(c) \subseteq \mathcal{X}'\}$.

Definition 3 (Independence). *A subset $\hat{\mathcal{X}}$ of $\mathcal{X}$ is* independent *in a CSP $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, iff* $\text{sol}(P) = \emptyset$ *or* $\text{sol}(P)_{|\hat{\mathcal{X}}} = \text{sol}(P_{|\hat{\mathcal{X}}})$.

Ordering of CSPs. We extend the partial order $\sqsubseteq$ on CSPs to capture restrictions. A CSP P' is *stronger or smaller* than P, write $P' \leq P$, iff $\mathcal{X}' \subseteq \mathcal{X}$, $\forall X \in \mathcal{X}' : \mathcal{D}'(X) \subseteq \mathcal{D}(X)$, and $\text{sol}(P') \subseteq \text{sol}(P)_{|\mathcal{X}'}$. P' is strictly stronger or smaller than P ($P' < P$) iff $P' \leq P$ and $\exists X \in \mathcal{X}' : \mathcal{D}'(X) \subset \mathcal{D}(X)$, or $\mathcal{X}' \subset \mathcal{X}$.

Definition 4 (Partial and Sub-problems). *Let $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ be a CSP. For a constraint c, $P' = (\mathcal{X}, \mathcal{D}, \mathcal{C} \cup \{c\})$ is a* sub-problem of P. *For some $\hat{\mathcal{X}}$ independent in P, $P_{|\hat{\mathcal{X}}}$ is a* partial problem of P.

For a sub-problem P' of a CSP P, we have $P' \leq P$. For a partial problem $P_{|\hat{\mathcal{X}}}$ of P, $P_{|\hat{\mathcal{X}}} \leq P$ and $P_{|\hat{\mathcal{X}}} < P$ iff $\hat{\mathcal{X}} \subset \mathcal{X}$. The following proposition states that independent partial problems can be solved independently.

Proposition 2. *For $\hat{\mathcal{X}}$ independent in P,* $\text{propagate}(P)_{|\hat{\mathcal{X}}} = \text{propagate}(P_{|\hat{\mathcal{X}}})$.

3.2 Dynamic Decomposition

The central idea of our approach is to detect independent partial problems during search, to enumerate partial solutions of the partial problems independently, and

Algorithm 2 Counting by Decomposition During Search

```
1: function CDDS($\mathcal{X}, \mathcal{D}, \mathcal{C}$)
2:     $(\mathcal{X}, \mathcal{D}', \mathcal{C}') \leftarrow$ Propagate($\mathcal{X}, \mathcal{D}, \mathcal{C}$)
3:     if IsFailed($\mathcal{X}, \mathcal{D}', \mathcal{C}'$) then return 0
4:     else if IsSolved($\mathcal{X}, \mathcal{D}', \mathcal{C}'$) then return 1
5:     else if IsDecomposable($\mathcal{X}, \mathcal{D}', \mathcal{C}'$) then                ▷ decomposition
6:         return CDDS(LeftBranch($\mathcal{X}, \mathcal{D}', \mathcal{C}'$)) $\cdot$ CDDS(RightBranch($\mathcal{X}, \mathcal{D}', \mathcal{C}'$))
7:     else                                                             ▷ normal branching
8:         return CDDS(LeftBranch($\mathcal{X}, \mathcal{D}', \mathcal{C}'$)) $+$ CDDS(RightBranch($\mathcal{X}, \mathcal{D}', \mathcal{C}'$))
9:     end if
10: end function
```

finally to combine the solutions, or to compute the number of solutions. We call this *Decomposition During Search (DDS)*.

In contrast to static decomposition of the initial model, the dynamic approach has one huge benefit: *constraint propagation and search make the CSP decomposable*. This is due to *entailment* of constraints and *assignment* of variables. Before going into these details, let us present DDS in a way that abstracts from how to detect that a problem is decomposable.

DDS. Algorithm 2 is a simple version of counting DDS (CDDS). The code extends CDFS (Algorithm 1) in lines 5 to 7, which correspond to the case discrimination whether the CSP is decomposable or not, and the decomposition into independent partial problems in line 6. We introduce a special branching that supports such a case distinction, a decomposing branching.

Definition 5 (Decomposing Branchings). *A branching is called* decomposing *iff there is a CSP P, such that*

$$\mathrm{sol}(\mathrm{lb}^B(P)) \times \mathrm{sol}(\mathrm{rb}^B(P)) = \mathrm{sol}(P).$$

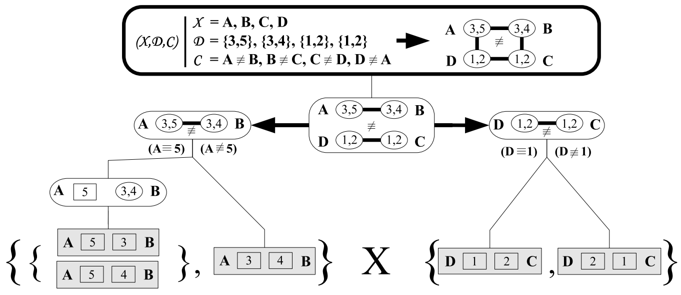
We call a CSP $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ decomposable*, iff there exists a set $\hat{\mathcal{X}} \subset \mathcal{X}$ that is independent in P.*

For a branching B, we define a corresponding decomposing branching *B^* by $\mathrm{lb}^{B^*}(P) = \mathrm{lb}^B(P)$ and $\mathrm{rb}^{B^*}(P) = \mathrm{rb}^B(P)$ whenever P is not decomposable, otherwise $\mathrm{lb}^{B^*}(P) = P_{|\hat{\mathcal{X}}}$ and $\mathrm{lb}^{B^*}(P) = P_{|\mathcal{X}\setminus\hat{\mathcal{X}}}$ for some deterministically selected true sub-set $\hat{\mathcal{X}} \subset \mathcal{X}$ that is independent in P. In the latter case the branching is said to* decompose *(the problem P).*

Note that decomposing branchings are not strict, because they do not necessarily produce stronger CSPs. However, they are weakly strict *in that they produce* stronger or smaller *CSPs*

DDS solves the CSP from Fig. 1 as shown in Fig. 2, avoiding the redundant work. As a technicality, we combine all solved partial problems with an arbitrary unsolved partial problem. The generated problems of LeftBranch and RightBranch are thus always unsolved. The number of solutions is found using only one decomposition and two branchings instead of five branchings.

This includes that the number of solutions of a propagated CSP with only one unassigned variable can be derived directly from its domain size.



(X,D,C)
X = A, B, C, D
D = {3,5}, {3,4}, {1,2}, {1,2}
C = A ≠ B, B ≠ C, C ≠ D, D ≠ A
(A = 5)
(A ≠ 5)
(D = 1)
(D ≠ 1)
X

Fig. 2. Search tree traversed by CDDS.

Detecting decomposability. The following proposition shows that the connected components of the constraint graph represent independent partial problems. Connected components detection takes linear time in the size of the graph.

Proposition 3. *For a CSP P, the set of nodes of a connected component of the corresponding constraint graph G_P is independent in P.*

Proof. Given a CSP $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, let $\hat{\mathcal{X}}$ be a connected component of the constraint hyper-graph G_P. We show that for $\mathrm{sol}(P) \neq \emptyset$, $\mathrm{sol}(P)_{|\hat{\mathcal{X}}} = \mathrm{sol}(P_{|\hat{\mathcal{X}}})$.

Case $\subseteq$. Let $A \in \mathrm{sol}(P)_{|\hat{\mathcal{X}}}$. In particular, A is a solution of all $c \in \mathcal{C}_{|\hat{\mathcal{X}}}$ and for all $X \in \hat{\mathcal{X}} : A(X) \in \mathcal{D}(X)$. Hence, $A \in \mathrm{sol}(P_{|\hat{\mathcal{X}}})$.

Case $\supseteq$. Let $A \in \mathrm{sol}(P_{|\hat{\mathcal{X}}})$. Take an arbitrary $A' \in \mathrm{sol}(P)$. We need to show that $B = A \times A'_{|\mathcal{X} \setminus \hat{\mathcal{X}}} \in \mathrm{sol}(P)$. Then, this implies $A \in \mathrm{sol}(P)_{|\hat{\mathcal{X}}}$. $B \in \mathrm{sol}(P)$ holds, since $\forall X \in \mathcal{X} : B(X) \in \mathcal{D}(X)$ and B is a solution of all constraints in $\mathcal{C}$. For seeing the latter, take an arbitrary $c \in \mathcal{C}$. Now, either $\mathrm{vars}(c) \in \hat{\mathcal{X}}$ or $\mathrm{vars}(c) \notin \hat{\mathcal{X}}$ since $\hat{\mathcal{X}}$ is a connected component of G_P.

Now we come back to the benefits of decomposing dynamically. Propagation and branching narrow the domains of the problem's variables. This results in looser constraint graphs with more potential for decomposition:

Entailment. For a CSP $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, a constraint $c \in \mathcal{C}$ is *entailed* by $\mathcal{D}$, iff for any $P' = (\mathcal{X}', \mathcal{D}', \mathcal{C}') \sqsubseteq P$ and $P'' = (\mathcal{X}', \mathcal{D}', \mathcal{C}' \setminus \{c\})$, $\mathrm{propagate}(P') = \mathrm{propagate}(P'')$. In other words, if c will not contribute propagation in any stronger CSP. If c is entailed, clearly $P' \equiv P''$. It is also obvious that the constraint graph for P'' is looser than the one for P', it contains one edge less. It is thus vital to our approach to detect entailment of constraints as early as possible, and to remove them from $\mathcal{C}$.

Assignment. A variable $X \in \mathcal{X}$ is *assigned* iff $|\mathcal{D}(X)| = 1$. Clearly, an assigned variable is independent of all other variables of the CSP. This implies that edges to assigned variables can be removed from the constraint graph.

Putting things together. DDS thus consists of the following parts: (1) Construction of the constraint graph, avoiding entailed constraints and assigned variables. (2) Computation of the connected components of the graph to detect independent partial problems. (3) A decomposing branching.

Enumerating solutions. A simple modification of the counting algorithm yields an enumeration strategy. Instead of computing solution counts, we can build up a tree-shaped compact representation of the solution space. Examples are given at the bottom of Fig. 2 and later in Fig. 4c. The compact representation finally can be expanded in order to enumerate the solutions.

3.3 Variable selection heuristics

How a variable-value branching B selects the variable to branch on is an essential decision for the search process. Different heuristics can easily lead to exponential differences in the size of the search tree. Usually, problem specific heuristics try to avoid or minimise these cases. A common method for varsel^B is 'first-fail', choosing a variable with minimal domain size. Other strategies use the degree in the constraint graph, or the minimal/maximal/median value in the domain.

In order to maximize the number of decompositions during search, a special heuristic can be used that analyzes the constraint graph and forces it to decompose. Such a heuristic can employ well-known graph algorithms such as cut-point, (minimal) cut-set, or bridge detection.

A branching that is based solely on properties of the constraint graph is not complete. For DDS, we thus need a combined branching. If forced decomposition is not possible (e.g. because no cut points exist), the combined branching falls back to a standard variable-value branching.

We can use the variable selection itself to guide the partial problem ordering after decomposition, choosing the first component to be the one that contains the preferred variable.

3.4 Global Constraints

One of the key features of any constraint solver is the use of global constraints to strengthen propagation. Therefore, DDS has to support global constraints in order to be practically useful.

For an n-ary constraint, the constraint graph contains a hyperedge that connects all n variable nodes. Now consider, as an example, the all-different constraint with variable domains $X_1 = \{0, 1\}, X_2 = \{0, 1\}, X_3 = \{2, 3\}, X_4 = \{2, 3\}$. The domain-consistent propagator for this constraint is clearly at a fixpoint. Furthermore, it is easy to see that the variable sets $\{X_1, X_2\}$ and $\{X_3, X_4\}$ are independent concerning all-different. However, the constraint graph contains just one hyperedge between all four variables. Thus, in the current set-up, the all-different constraint *prevents decomposition*.

We have found two remedies for this problem. First, an n-ary propagator can *decompose itself* into several smaller propagators. In Algorithm 2, this means that PROPAGATE returns a modified set $\mathcal{C}$. Second, we can introduce more than one hyperedge per propagator into the constraint graph: $E = \{V \mid V \in \text{indvars}(c), c \in \mathcal{C}\}$. $\text{indvars}(c)$ is the set of independent subsets of $\text{vars}(c)$.

4 Theoretical Analysis of DDS

Here, we compare CDFS and CDDS analytically by the size of their corresponding search trees. Specifically, we study the case that both algorithms count the solutions of a CSP P using the same heuristic, specified by a branching B.

Both CDFS and DDFS will completely search through their corresponding binary search trees $\mathrm{bst}^B(P)$ and $\mathrm{bst}^{B^*}(P)$. Since, in particular, we can avoid to re-propagate in the immediate children of a decomposition node in DDS, a good measure for the propagation in DFS and DDS is the number of branch nodes. For a search tree T, the *set of branch nodes* $\mathrm{bn}(T)$ contains the nodes of T that represent neither decompositions, nor failures, nor solutions.

For the comparison, we restrict ourselves to variable-value branchings B that are *dependency restricted*, i.e. for any independent set $\hat{\mathcal{X}}$ of a CSP $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, $\mathrm{varsel}(P) \in \hat{\mathcal{X}} \implies \mathrm{varsel}(P) = \mathrm{varsel}(P_{|\hat{\mathcal{X}}})$. Both restrictions are small in practice, since most practically used branchings have these properties, e.g. the wide-spread first-fail heuristic.

We will show that, in these terms, DDS performs at least as good as DFS, if we never decompose unsatisfiable sub-problems. Therefore, we introduce the decomposing branching B^+, which is defined for a branching B just like the corresponding decomposing branching B^* (cf. Def. 5) but redefines lb and rb only if P is decomposable *and* satisfiable.

Intentionally, the restriction introduced by B^+ neither covers the case of decomposing unsatisfiable sub-problems nor is efficiently detectable, wich is discussed subsequently.

Theorem 1. *For a CSP P and a dependency restricted variable-value branching B, we have*
$$|\,\mathrm{bn}(\mathrm{bst}^B(P))| \;\geq\; |\,\mathrm{bn}(\mathrm{bst}^{B^+}(P))|.$$

The theorem is shown by induction over CSPs P in the $<$-ordering. A proof can be based upon the following lemma.

Lemma 1. *Let $\hat{\mathcal{X}}$ be independent in the CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, B is a dependency restricted, variable-value branching. If P is satisfiable, then*

$$|\mathrm{bn}(\mathrm{bst}^B(P))_{|\hat{\mathcal{X}}}| \geq |\,\mathrm{bn}(\mathrm{bst}^{B^+}(P_{|\hat{\mathcal{X}}}))|. \tag{1}$$

The lemma is shown by induction over P in the $<$-ordering and size of $\hat{\mathcal{X}}$.

Since satisfiablity is not efficiently detectable, we need to consider the case of decomposing unsatisfiable nodes. There, we cannot give an analytical performance guarantee for DDS. Since DDS is bound to enumerate one independent component at a time, DFS may uncover an inconsistency with less search nodes than DFS. For example, this happens when only one of the independent sets is inconsistent and is selected as the second component. Then DDS detects the inconsistency only after completely enumerating the first component, which is potentially avoided in DFS.

To obtain the full performance guarantee, one can explore all components of a decomposition in a pseudo-parallel fashion, mimicking how DFS would switch between components. However, pseudo-parallel exploration would add a runtime overhead and is difficult to implement. Our empirical evaluation of DDS supports our design decision to not implement such a pseudo-parallel exploration.

The discussion in this section provides some insight into DDS, but we expect that specialized search heuristics tailored for DDS can be more beneficial than performance guarantees for exactly the same heuristic.

5 Implementation Notes

DDS has been implemented using the constraint programming library Gecode. In this section, we give an overview of relevant technical details of Gecode, and we discuss the three main additions to Gecode that enable DDS: access to the constraint graph, a decomposing branching, and a specialized search engine. The changes to Gecode comprise just 2500 lines of code. The source code of the prototype implementation is available from the authors upon request.

Gecode. The Gecode library [11] is an open source constraint solver implemented in C++. It lends itself to a prototype implementation of DDS because of three facts: (1) Full source code enables access to all internal data structures. (2) Search is based on recomputation and copying, which significantly eases the implementation of specialized branchings and search engines. (3) It provides good performance, so that benchmarks give meaningful results.

Constraint Graph. In most CP systems, the constraint graph is implicit in the data structures for variables and propagators. Gecode, e.g., maintains a list of propagators, and each propagator has access to the variables it depends on.

For DDS, a more explicit representation is needed that supports the computation of connected components. We can thus either maintain an additional, explicit constraint graph during propagation and search, or extract the graph from the implicit information each time we need it. For the prototype implementation, we chose the latter approach. Our implementation uses the data structures and algorithms for connected components provided by the boost graph library [6].

Decomposing branching. Branchings in Gecode are fully programmable. A branching has to support three operations: STATUS, DESCRIPTION, and COMMIT. The first operation determines whether the branching can actually create a branch. The second, DESCRIPTION, returns an abstract description of the branch. The third, COMMIT, executes the branching according to a given description and alternative number.

A decomposing branching in Gecode is a wrapper around a variable-value branching. The STATUS method is just delegated to the wrapped branching. The actual work is done by DESCRIPTION: it requests the constraint graph and performs the connected component analysis. If decomposition is possible, a special description is returned, representing the independent subsets $\hat{\mathcal{X}}_i \subset \mathcal{X}$. Otherwise, DESCRIPTION is delegated to the embedded variable-value branching.

When COMMIT is invoked with a variable-value description, the call is again delegated to the embedded branching. For a decomposition description, the branching's list of variables is updated to $\hat{\mathcal{X}}_i$ for branch i, those still active in the selected component. Branching, in contrast to how it is theoretically presented in this paper, is n-ary.

Decomposition search engine. We developed three specialized search engines for DDS. A graphical search engine based on Gecode's *Gist* (graphical interactive search tool) displays the search tree with special decomposition nodes, and allows to get an overview of where and how a particular problem can be decomposed. The counting search engine computes the number of solutions. The general-purpose search engine allows to incrementally search the whole tree and access all the partial solutions. All search engines accept cut-off parameters for the number of (full, not partial!) solutions to be explored.

6 Application and Empirical Results

To illustrate possible use cases of DDS, we applied the search strategy to two counting problems, graph coloring and optimal protein structure prediction. Both applications show tremendous reductions in runtime and search tree size.

The applications were realized using our DDS implementation in Gecode. Only the search strategy was changed (DFS vs. DDS) – modelling, variable and value selection were kept the same for an appropriate comparison.

6.1 Graph coloring

The problem. We applied DDS to the problem of graph coloring, more precisely vertex coloring: Each vertex of a given undirected graph is colored such that no two adjacent vertices are assigned the same color. We determine the chromatic polynomial for the chromatic number, i.e. the number of ways a graph can be coloured using the minimal number of colors possible.

The constraint model. For a given undirected graph g and a number of colors c we introduce one variable per node with the values $0..(c-1)$. For each maximal clique of size > 2, we post an all-different constraint on the corresponding variables. For all remaining edges we add binary unequality constraints.

The test sets. We generated the two test sets GC-30 and GC-50 of graphs with 30 and 50 nodes. For each size, random graphs were obtained by inserting an edge of the complete graph with a fixed uniform edge propability P^e. This was done using the Erdős-Rényi random graph generator GTgraph [12] built for the 9th DIMACS Implementation Challenge. For each edge propability P^e from 16 to 40 percent, 2000 graphs were generated and their coloring counted via DFS and DDS. To handle highly degenerated problems as well, we stopped after 1 million solutions.

Results. For the test sets, Tab. 1 compares the time consumption and search tree size by average ratios of DFS and DDS ($\frac{CDFS}{CDDS}$). A figure of 100 thus means

CDFS/CDDS:	Test set	16 %	18 %	20 %	22 %	24 %	28 %	32 %	36 %	40 %
Rel. runtime:	GC-30	411.2	197.7	75.74	34.6	23.1	11.9	3.85	2.74	2.14
	GC-50	242.7	151.8	34.23	16.5	18.2	3.4	2.71	–	–
Search tree size:	GC-30	680.3	344.4	142.0	74.48	62.27	33.96	10.90	6.86	4.97
	GC-50	646.1	383.8	94.28	47.26	41.69	11.6	9.28	–	–

Table 1. Average ratios of CDFS vs. CDDS for various edge propabilities

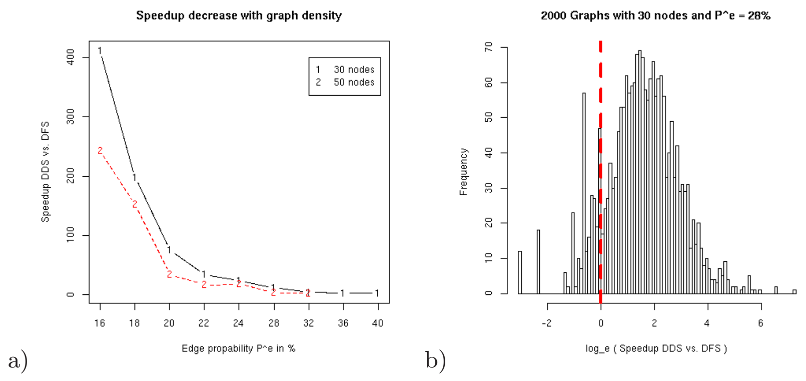


Fig. 3. (a) Avg. speedup decrease from 400 to 2 by graph density, (b) Histogram of logarithmic speedup for $P^e = 28$ and 30 nodes (the dashed line marks equal runtime).

that CDDS is 100 times as fast as CDDS, or that the CDFS search tree has 100 times as many nodes as the one for CDDS. A dash means that most of the problems were not solved within a given time-out.

The presented runtime ratios show the high speedup for graphs with edge propabilities $P^e \leq 40\%$. The distribution of speedup is exemplified in Fig. 3b. The speedup corresponds to an even larger reduction of the search tree for DDS, which was only increased for 0.5% of all problems. Furthermore, sparse graphs yield a much higher runtime improvement than dense graphs, visualized by Fig. 3a. The number of fails and propagations shows no visible correlation as speedup or the number of clonings.

Still, the reduction of clonings is not completely reflected in runtime speedup, whereby the computational overhead of DDS in the current prototypic implementation becomes visible. Anyway, our data shows that DDS is well suited to improve solution counting even for dense graphs with P^e about 40%. We expect even higher speedups and search-tree reductions if the solutions are counted completely, i.e. without the current upper bound of 1 million.

Table 2 suggests that the speedup decreases with increasing number of nodes to color in the graph. With increasing number of nodes, the graph *as well as the constraint graph* grow quadratically. As our prototype implementation has to rebuild the constraint graph at each step, this explains the lower speedup. A system using an incremental approach may thus perform even better.

a)

	H	P
H	-1	0
P	0	0

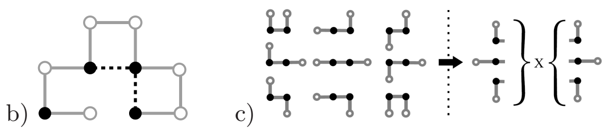


Fig. 4. a) Contact energy function b) Example HP-structure with energy -2 (H-monomer: black, P-monomer: white, structure back bone: grey, HH-contact: dotted) c) Compression of the structure/solution space representation for PHHP from 9 structures down to 3 by 3 partial structures.

6.2 Optimal protein structure prediction

The problem. The prediction of optimal (minimal energy) structures of simplified lattice proteins is a hard problem in Bioinformatics and has been proven to be NP-complete [3]. Here we focus on the HP-model introduced in [14]. In this model, a protein chain is reduced to a sequence of monomers of equal size, whereby the 20 aminoacids are grouped into hydrophobic (H) or polar (P). A structure is a self-avoiding walk of the underlying lattice (e.g. square or cubic). A contact energy function is used to determine the energy of a structure. The energy table and an example is given in Fig. 4. The problem is to predict minimal energy structures for a given lattice and HP-sequence.

The number and quality of optimal structures has application in the study of energy landscape properties, protein evolution and kinetics [9,17,20].

The constraint model. In [2], the problem was successfully modelled as CSP and named Constraint-based Protein Structure Prediction (CPSP). Herein, a variable is introduced for each sequence position and lattice points as domains[3]. The self-avoiding walk is modelled by a sequence of binary neighboring constraints (ensuring the connectivity of successive monomers) and a global all-different constraint for self-avoidingness.

CPSP uses a database of precalculated point sets, called H-cores, that represent possible optimal distributions of H-monomers. By that, the optimization problem becomes a satisfaction problem for a given H-core, if H-variables are restricted to these positions. All solutions of the CSP are structures with optimal energy, due to the optimality of the H-core, and the procedure is repeated with the remaining H-cores until all optimal structures are enumerated.

The test sets. We generated two test sets, PS-48 and PS-64, with uniformly distributed random HP-sequences of length 48 and 64. For the generation we used the free available CPSP implementation [7]. With only minimal modifications (new branching) we could use the included CSP model with DDS.

PS-48 contains 6350 HP-sequences and for each up to 1 million optimal structures in the cubic lattice were predicted. For the 2630 HP-sequences inside PS-64 up to 2 million structures have been predicted in the cubic lattice, due to the increasing degeneracy in sequence length.

[3] In practice, lattice positions are indexed by integers such that standard constraint solvers for finite domains over integers are applicable.

CDFS / CDDS	runtime	clonings	fails	propagations
PS-48	2.98	11.30	1.40	3.27
PS-64	4.23	25.33	1.76	5.43

Table 2. Average ratios for CPSP using CDFS vs. CDDS

Results. The average ratio results are given in Tab. 2. There, the enormous search tree reduction with an average factor of 11 and 25 respectively is shown. The reduction using DDS compared to DFS leads to much less propagations (3- to 5-fold). This and the slightly less fails result in a runtime speedup of 3-/4-fold using the same variable selection heuristics for both search strategies (maximal-degree in constraint graph combined with minimal domain size). Herewith, the immense possibilities of DDS even without advanced constraint-graph specific heuristics are demonstrated. This also shows the rising advantage of DDS to DFS for increasing problem size that leads to a higher degeneracy.

7 Discussion

The paper introduced a new search algorithm, Decomposition During Search (DDS). The strategy is designed to avoid much of the redundant work of standard backtracking search when solving $\#P$-hard counting problems with constraint programming. In the paper, we give a theoretical foundation and analysis of the method. As a major contribution, we show that the approach can be succesfully integrated into a full-featured constraint programming system, using Gecode as an example, and that DDS is compatible with such essential CP features as global constraints. Our results on graph coloring and protein structure prediction show the huge potential of DDS in terms of search tree size reduction and the already high true runtime speedup. The speedup proves that DDS can be implemented competitively, and with a reasonable overhead. We expect that we will achieve even higher speedups by improving the constraint graph representation and maintaining it incrementally, which is a current topic of research and development.

We envision promising future research in the following directions. First, as just mentioned, providing efficient access to the constraint graph. There, n-ary global constraints deserve special attention, when we are interested in reflecting the exact dependencies of the variables. All advances in this area will be beneficial not only for DDS but for other applications as well. Second, the development of specifically tailored heuristics for DDS. Such heuristics should aim at decomposing the problem as often as possible, and in a well-balanced way. Naturally, such heuristics can profit from employing information about the constraint graph and thus should be constraint-graph aware. Additionally, decomposition-directed heuristics might counteract problem specific heuristics. Balancing such heuristics is a further research direction. Whereas selecting variables with large degree is only a start, e.g. detection of cutting planes was already discussed for this purpose and awaits its integration into CP.

Acknowledgements. The authors thank Christian Schulte and Mikael Lagerkvist for help with the implementation and discussions about the architecture and the paper.

References

1. Ola Angelsmark and Peter Jonsson. Improved algorithms for counting solutions in constraint satisfaction problems. In *Proc. of CP*, pages 81–95, 2003.
2. Rolf Backofen and Sebastian Will. A constraint-based approach to fast and exact structure prediction in three-dimensional protein models. *Constraints*, 11(1):5 – 30, 2006.
3. B. Berger and T. Leighton. Protein folding in the hydrophobic-hydrophilic (HP) model is NP-complete. *Journal of Computational Biology*, 5(1):27–40, 1998.
4. Armin Biere and Carsten Sinz. Decomposing SAT problems into connected components. *Satisfiability, Boolean Modeling and Computation*, 2:191–198, 2006.
5. Elazar Birnbaum and Eliezer L. Lozinskii. The good old Davis-Putnam procedure helps counting models. *Journal of AI Research*, 10:457–477, 1999.
6. Boost graph library, 2007. Available as an open-source library from `www.boost.org`.
7. CPSP: Tools for constraint-based protein structure prediction, 2006. Available as an open-source library from `www.bioinf.uni-freiburg.de/sw/cpsp`.
8. Rina Dechter and Robert Mateescu. The impact of AND/OR search spaces on constraint satisfaction and counting. In *CP'2004*, 2004.
9. Christoph Flamm, Ivo L. Hofacker, Peter F. Stadler, and Michael T. Wolfinger. Barrier trees of degenerate landscapes. *Z.Phys.Chem*, 216:155–173, 2002.
10. Eugene C. Freuder and Michael J. Quinn. Taking advantage of stable sets of variables in constraint satisfaction problems. In *Proc. of IJCAI-85*, pages 1076–1078, 1985.
11. Gecode: Generic constraint development environment, 2007. Available as an open-source library from `www.gecode.org`.
12. GTgraph: A suite of synthetic graph generators, 2006. Available as an open-source library from `www-static.cc.gatech.edu/~kamesh/GTgraph`.
13. R. J. Bayardo Jr. and J. D. Pehoushek. Counting models using connected components. In *Proc. of the 7th Nat'l Conf. on AI*, 2000.
14. Kit Fun Lau and Ken A. Dill. A lattice statistical mechanics model of the conformational and sequence spaces of proteins. *American Chemical Society*, 22:3986 – 3997, 1989.
15. Wei Li and Peter van Beek. Guiding real-world SAT solving with dynamic hypergraph separator decomposition. In *Proceedings of the Sixteenth IEEE International Conference on Tools with Artificial Intelligence*, 2004.
16. Gilles Pesant. Counting solutions of CSPs: A structural approach. In *IJCAI-05*, page 260, 2005.
17. A. Renner and E. Bornberg-Bauer. Exploring the fitness landscapes of lattice proteins. In *2nd. Pacif. Symp. Biocomp.* Singapore, 1997.
18. Dan Roth. On the hardness of approximate reasoning. *Artif. Intelligence*, 82(1-2):273–302, 1996.
19. Leslie G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8(2):189–201, 1979.
20. M. Wolfinger, S. Will, I. Hofacker, R. Backofen, and P. Stadler. Exploring the lower part of discrete polymer model energy landscapes. *Europhysics Letters*, 74(4):725–732, 2006.

Appendix: Proofs for Section 4

Proof (Theorem 1). Let B be a dependency restricted variable-value branching. We show the claim by induction over CPSs P in the $<$-ordering.

Base case. P is solved or failed. $\mathrm{bn}(\mathrm{bst}^B(P)) = \emptyset$ and $\mathrm{bn}(\mathrm{bst}^{B^+}(P)) = \emptyset$ by definition. **Induction step.**

A. If P is not decomposable or not satisfiable, the claim follows immediately by induction, since B and B^+ branch on P_p in exactly the same way.

B. Let $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ be satisfiable and decomposable with independent $\hat{\mathcal{X}} \subset \mathcal{X}$. Let $\mathrm{lb}^{B^+}(P_\mathrm{p}) = P_{\mathrm{p}|\hat{\mathcal{X}}}$ and $\mathrm{rb}^{B^+}(P_\mathrm{p}) = P_{\mathrm{p}|\mathcal{X}\setminus\hat{\mathcal{X}}}$.
We have to show that

$$|\,\mathrm{bn}(\mathrm{bst}^B(P))| \geq |\,\mathrm{bn}(\mathrm{bst}^{B^+}(P_{|\hat{\mathcal{X}}})) \cup \mathrm{bn}(\mathrm{bst}^{B^+}(P_{|\mathcal{X}\setminus\hat{\mathcal{X}}}))|.$$

Since $\hat{X}$ and $\mathcal{X} \setminus \hat{\mathcal{X}}$ are disjoint sets it is sufficient to show that

$$|\mathrm{bn}(\mathrm{bst}^B(P))_{|\hat{\mathcal{X}}}| \geq |\,\mathrm{bn}(\mathrm{bst}^{B^+}(P_{|\hat{\mathcal{X}}}))|.$$

This follows from Lemma 1 (since P is satisfiable.)

Proof (Lemma 1). Let B a branching as in the lemma. We show the claim by induction over P in the $<$-ordering and the size of $\hat{\mathcal{X}}$. **Base case.** If P is failed, nothing is to show. If $P_{|\hat{\mathcal{X}}}$ is solved, $\mathrm{bn}(\mathrm{bst}^{B^+}(P_{|\hat{\mathcal{X}}})) = \emptyset$ by definition. For $\hat{\mathcal{X}} = \emptyset$ the claim holds trivially. **Induction step.** We have two cases, one where we only branch and one where we decompose.

A. If P is not satisfiable, nothing is to show. If $P_{|\hat{\mathcal{X}}}$ is not decomposable, we distinguish two cases.

Case 1 $\mathrm{varsel}(P_\mathrm{p}) \in \hat{\mathcal{X}}$. Here, the claim follows immediately by induction, since B is dependency restricted(!) and therefore B and B^+ branch on P_p in exactly the same way.

Case 2 $\mathrm{varsel}(P_\mathrm{p}) \notin \hat{\mathcal{X}}$. Since P, and therefore P_p, is satisfiable at least one of $\mathrm{lb}^B(P_\mathrm{p})$ and $\mathrm{rb}^B(P_\mathrm{p})$ is satisfiable. We only show the case $\mathrm{lb}^B(P_\mathrm{p})$ satisfiable. By induction hypothesis holds 1 for $\mathrm{lb}^B(P_\mathrm{p})$, i.e.

$$|\mathrm{bn}(\mathrm{bst}^B(\mathrm{lb}^B(P_\mathrm{p})))_{|\hat{\mathcal{X}}}| \geq |\,\mathrm{bn}(\mathrm{bst}^{B^+}(\mathrm{lb}^B(P_\mathrm{p})_{|\hat{\mathcal{X}}}))|.$$

Then, $\mathrm{lb}^B(P_\mathrm{p})_{|\hat{\mathcal{X}}} = P_{\mathrm{p}|\hat{\mathcal{X}}}$, due to the independence of $\hat{\mathcal{X}}$ in P. Furthermore, $|\mathrm{bn}(\mathrm{bst}^B(P))_{|\hat{\mathcal{X}}}| \geq |\mathrm{bn}(\mathrm{bst}^B(\mathrm{lb}^B(P_\mathrm{p})))_{|\hat{\mathcal{X}}}|$.

B. Let $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, $P_{|\hat{\mathcal{X}}}$ is satisfiable and decomposable with independent $\hat{\mathcal{Y}} \subset \hat{\mathcal{X}}$. Let $\mathrm{lb}^{B^+}(P_{|\hat{\mathcal{X}}}) = P_{|\hat{\mathcal{Y}}}$ and $\mathrm{rb}^{B^+}(P) = P_{|\hat{\mathcal{X}}\setminus\hat{\mathcal{Y}}}$.
It suffices to show that

$$|\mathrm{bn}(\mathrm{bst}^B(P))_{|\hat{\mathcal{Y}}}| \geq |\,\mathrm{bn}(\mathrm{bst}^{B^+}(P_{|\hat{\mathcal{Y}}}))|.$$

and

$$|\mathrm{bn}(\mathrm{bst}^B(P))_{|\hat{\mathcal{X}}\setminus\hat{\mathcal{Y}}}| \geq |\,\mathrm{bn}(\mathrm{bst}^{B^+}(P_{|\hat{\mathcal{X}}\setminus\hat{\mathcal{Y}}}))|.$$

This follows by induction.