A Comparison of Decomposition Methods for the Maximum Common Subgraph Problem

Maël MINOT
LIRIS UMR 5205 CNRS
Université de Lyon
INSA de Lyon, France
mael.minot@liris.cnrs.fr

Samba Ndojh NDIAYE
LIRIS UMR 5205 CNRS
Université de Lyon
Université Lyon 1, France
samba-ndojh.ndiaye@liris.cnrs.fr

Christine SOLNON
LIRIS UMR 5205 CNRS
Université de Lyon
INSA de Lyon, France
christine.solnon@liris.cnrs.fr

***Abstract*—The maximum common subgraph problem is an $\mathcal{NP}$-hard problem which is very difficult to solve with exact approaches. To speed up the solution process, we may decompose it into independent subproblems which are solved in parallel. We describe a new decomposition method which exploits the structure of the problem to decompose it. We compare this structural decomposition with domain-based decompositions, which basically split variable domains. Experimental results show us that the structural decomposition leads to better speedups on two classes of instances, and to worse speedups on one class of instances.**

***Keywords*-Maximum Common Subgraph; Constraint Programming; Decomposition; Graph Triangulation**

I. Introduction

Searching for a maximum common subgraph has many applications, for example, in chemoinformatics, bioinformatics, or image processing where it gives a measure of the similarity between objects represented by graphs. However, this problem is $\mathcal{NP}$-hard and very challenging. To speed up the solution process, we may decompose the problem into independent subproblems which are solved in parallel, as proposed, for example, in [1], [2], [3] for the maximum clique problem, or in [4], [5] for constraint satisfaction problems. In most cases, the decomposition is done by splitting variable domains. In order to balance the workload on the workers, EPS ([4], [5]) splits the initial problem in a very large number of subproblems whose resolution time can be shared by workers: all subproblems are put in a queue and workers take subproblems when they need work. Experiments in [4], [5] show us that a good decomposition is generally obtained, with EPS, by generating about 30 subproblems per worker. In [5], the average speedup reported with EPS is close to $0.5k$, where k is the number of workers, on a large benchmark of instances. This is much better than the speedup obtained with a work stealing approach, where subproblems are dynamically generated by splitting the subproblem currently solved by a worker, whenever another worker has finished its own work. However, even when decomposing the initial problem into 30 subproblems per worker, it may happen that one subproblem is much more difficult than the others and becomes a bottleneck for the speedup. Let t_{max} be the time needed to solve the hardest subproblem, and t_0 be the time needed to solve the initial problem; the speedup cannot be higher than t_0/t_{max}.

A first goal of this paper is to study subproblem balance for the maximum common subgraph problem, and to show that the decomposition proposed in [4], [5] leads to very unbalanced subproblems: on our benchmark, t_0/t_{max} is equal to 4 when decomposing problems into 400 subproblems or so. As a consequence, the speedup cannot be higher than these bounds. Actually, with 30 subproblems per worker, we observe an average speedup of 3.

This motivates us to introduce and experimentally evaluate two new decomposition methods for the maximum common subgraph problem. The first one is a straightforward modification of EPS: instead of creating a subproblem for every possible value in a variable domain (while removing inconsistent subproblems), we introduce a binary decomposition where domains are split in two parts. The second one is fundamentally different and borrows features from the structural decomposition proposed in [6] and extended in [7], where a constraint satisfaction problem is decomposed into independent subproblems by exploiting the structure of its microstructure. Experimental evaluations show us that these two new decompositions lead to more balanced subproblems, and that structural decomposition allows us to increase the speedup with a reasonable number of workers for two classes of instances, whereas it slightly decreases it for one class.

The next section of this paper presents the background of this work: the maximum common subgraph problem, the domain decomposition method of [4], [5], and the structural decomposition method of [6]. Sections III and IV introduce two new decompositions for the maximum common subgraph problem. Section V presents an experimental comparison of these two new decompositions with the original decomposition of [4] on a classical benchmark.

II. Background

A. Maximum common subgraph

An undirected graph G is defined by a finite set of nodes N_G and a set of edges $E_G \subseteq N_G \times N_G$. Each edge is an undirected couple of nodes. In a directed graph, E_G is a set of arcs, which are directed couples of nodes. In the following definitions, we will only be considering undirected graphs. However, these notions may be applied to directed graphs as well.

Definition 1: Let G and G' be graphs. G is *isomorphic* to G' if there exists a bijective function $f : N_G \to N_{G'}$ which preserves edges, *i.e.*

$$\forall (u,v) \in N_G \times N_G, \{u,v\} \in E_G \Leftrightarrow \{f(u), f(v)\} \in E_{G'}$$

Definition 2: G' is an *induced subgraph* of G if $N_{G'} \subseteq N_G$ and $E_{G'} = E_G \cap (N_{G'} \times N_{G'})$. We say that G' is the subgraph of G induced by $N_{G'}$, denoted $G_{\downarrow N_{G'}}$.

In other words, G' is obtained from G by removing all nodes of G which are not in $N_{G'}$ and keeping only edges whose extremities are both in $N_{G'}$.

Definition 3: A *common subgraph* of two graphs G and G' is a graph isomorphic to subgraphs of G and G'.

Definition 4: A *Maximum Common Induced Subgraph* (MCIS) is a common induced subgraph which has a maximum number of nodes.

Figure 1 displays an example of MCIS.

In this article, we restrict ourselves to the MCIS, but the results can easily be extended to the maximum common partial subgraph problem as shown in [8].

There exist two main approaches for solving MCIS: the first approach (described in II-B) explores the search space by branch and bound, and this may be achieved by using constraint programming (CP); the second approach (described in II-C) is based on a reformulation of MCIS into a maximum clique problem.

B. CP model for the MCIS

CP solves problems modelled as Constraint Satisfaction Problems (CSPs): a CSP is defined by a triple (X, D, C) such that X is a finite set of variables, D associates a domain $D(x_i)$ with every variable $x_i \in X$, and C is a set of constraints such that each constraint is defined over a subset of variables and gives lists of values these variables can be assigned simultaneously. A solution is an assignment of all variables satisfying every constraint. CP solves CSPs by building a search tree and propagating constraints at each node of the search tree in order to prune branches.

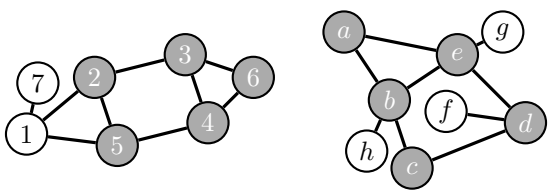


Figure 1. Example of MCIS. A bijection between MCIS nodes can be defined as follows: $2 \leftrightarrow d$, $3 \leftrightarrow e$, $4 \leftrightarrow b$, $5 \leftrightarrow c$, $6 \leftrightarrow a$.

[8] introduces a CSP model for solving MCIS with CP. Given two graphs G and G', this CSP associates a variable x_u with every node u of G, and the domain of this variable contains all nodes of G', plus an additional value $\perp$: variable x_u is assigned to $\perp$ if node u is not matched to any node of G'; otherwise x_u is assigned to the node of G' it is matched with. A set of edge constraints is introduced in order to ensure that variable assignments preserve edges between matched nodes, *i.e.*, $\forall \{u,v\} \subseteq N_G, (x_u = \perp) \vee (x_v = \perp) \vee (\{u,v\} \in E_G \Leftrightarrow \{x_u, x_v\} \in E_{G'})$.

MCIS is an optimization problem, the goal of which is to maximize the number of matched nodes. To ensure that, the CSP also contains:

- a variable $cost$ which corresponds to the number of nodes of G which are not matched;
- a variable $x_\perp$ whose domain is $D(x_\perp) = \{\perp\}$ so that it is forced to be assigned to $\perp$;
- a soft constraint $softAllDiff(\{x_u, u \in N_G\} \cup \{x_\perp\}, cost)$ which ensures that all x_u variables are assigned to values different from $\perp$ whenever it is possible, and that the $cost$ variable is equal to the number of x_u variables that should change their value so that they all have different values.

A solution is an assignment of all variables which satisfies all edge constraints while minimizing the value of $cost$.

[8] experimentally evaluates different constraint propagation techniques for this CSP modelling of the MCIS. The combination "MAC+Bound" generally obtains very good results and outperforms the state-of-the-art branch and bound approach of [9]: Maintaining Arc Consistency (MAC) [10] is used to propagate hard constraints; "Bound" checks whether it is possible to assign distinct values to enough x_u variables to surpass the best cost found so far (it is a weaker version of GAC(*softAllDiff*) [11] which computes the maximum number of variables that can be assigned distinct values).

C. Reformulation of MCIS as a maximum clique problem

We may solve MCIS by introducing a compatibility graph and searching for cliques in it [12], [13], [14].

Definition 5: A compatibility graph of two graphs G and G' is an undirected graph G_C whose set of nodes is $N_{G_C} = N_G \times N_{G'}$ and whose set of edges is $N_{G_E} = \big\{\{(u,u'),(v,v')\} \subseteq N_{G_C}$, (u,u') and (v,v') are compatible$\big\}$, where two nodes (u,u') and (v,v') of N_{G_C} are compatible if $u \neq v$ and $u' \neq v'$, and if they preserve edges (*i.e.* $\{u,v\} \in E_G \Leftrightarrow \{u',v'\} \in E_{G'}$).

A *clique* is a subgraph whose nodes are all linked pairwise. A clique is *maximal* if it is not strictly included in any other clique, and it is *maximum* if it is the biggest clique of a given graph, with respect to the number of nodes.

As illustrated in Figure 2, a clique in G_C corresponds to a set of compatible matchings of nodes in G and G'. Therefore, such a clique corresponds to a common induced subgraph, and a maximum clique of G_C is a MCIS of G and G'. It follows that any method able to find a maximum clique in a graph can be used to solve the MCIS problem. This yields similar results than a branch and bound approach [15].

D. Domain decomposition for CP

To speedup the solution process of CP, we may parallelize it as proposed in [4], [5]. The idea is to split the initial CSP into a large number of independent subproblems which are solved in parallel by workers. A key point to balance the workload is to generate more subproblems than workers (typically, 30 subproblems by worker). The decomposition into subproblems is computed by selecting a subset of variables and by enumerating the combinations of values of these variables that are not detected inconsistent by the propagation mechanism of a CP Solver. More precisely, a Depth-Bounded Depth-First Search (DBDFS) is used to compute subproblems. A DBDFS is a Depth-First Search that never visits nodes located at a depth greater than a given value. First, we consider a static ordering of the variable (usually, by non decreasingly domain sizes). Then, the main step of the algorithm is applied: define a depth p and perform a DBDFS with p as limit. This search triggers the constraint propagation mechanism each time a modification occurs. For each leaf of this search which is not a failure, the first p variables are assigned and so the subproblem defined by this assignment is consistent with the propagation. Thus the set of leaves defines a set S of subproblems. Next, if S is large enough, then the decomposition is finished. Otherwise, the main step is applied again until the expected number of subproblems is reached. From here onward, we will designate this method with "domain decomposition" (DOM) for clearer comparisons with other methods.

E. Structural decomposition for CSPs

A dual approach to decompose a CSP is to exploit its structure. A first way to do that is to compute a tree decomposition of the constraint hypergraph (which associates a node with each variable, and an hyperedge with each constraint) in order to identify independent subproblems, as proposed in [16]. However, this kind of approach is useless for solving MCIS because the scope of the *softAllDiff* constraint is the whole set of variables, so that the tree decomposition contains only one node (*i.e.* one subproblem).

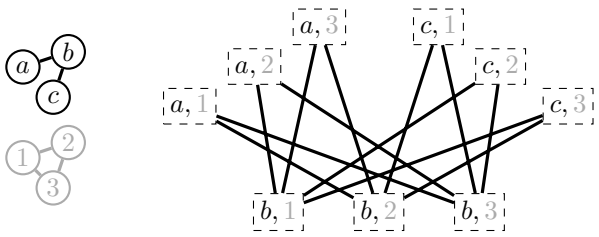


Figure 2. Two graphs and their compatibility graph G_C. For example, since $\{a, c\}$ is not an edge while $\{1, 3\}$ is an edge, the edge $\{(a, 1), (c, 3)\}$ does not belong to G_C. One of the many maximum cliques in G_C is $\{\{a, 3\}, \{b, 2\}\}$ and it corresponds to a MCIS.

Another structural approach for decomposing a CSP into independent subproblems has been proposed by [6] and is called TR-decomposition. The idea is to reformulate the CSP as the problem of finding a clique of size n (with n the number of variables in the CSP) in its *microstructure*. The microstructure of a CSP is a graph similar to the compatibility graph described in II-C. Each node is associated with a $(variable, value)$ couple. There is an edge between two nodes if and only if the corresponding assignments are compatible, *i.e.* if every constraint of the problem is satisfied when those two assignments are made simultaneously. Thereby, a clique of size n in the microstructure is a solution of the CSP.

The main idea behind TR-decomposition is to triangulate the microstructure to generate subproblems more easily. Indeed, a triangulated graph G cannot contain more than $|N_G|$ maximal cliques, whereas non triangulated graphs may contain an exponential number of maximal cliques.

Definition 6: A graph G is triangulated if and only if every cycle whose length is 4 or more has a *chord*. A chord is an edge whose extremities are two non-consecutive nodes of a same cycle.

The process of turning a given graph into a triangulated graph *only by adding edges* is called *triangulation*. Since triangulation adds edges without removing any, the maximal cliques of the original graph can still be found, included in the maximal cliques of the triangulated graph. To proceed with TR-decomposition, one must recover these cliques within the newly formed ones.

To sum up, TR-decomposition consists in building the microstructure of the problem, triangulating it, computing the maximal cliques of the triangulated microstructure and solving separately the subproblems induced by those cliques.

III. Binary domain decomposition for MCIS

The DOM decomposition introduced in [4] basically creates subproblems by assigning some variables while removing inconsistent subproblems. When applying this decomposition method to the CP model of the MCIS recalled in II-B, we obtain subproblems by assigning a subset of variables $X_a \subseteq \{x_u, u \in N_G\}$ to nodes of G' or to $\perp$. First experiments (reported in Section V) have shown us that this decomposition leads to very unbalanced subproblems, and therefore very low speedups. Actually, assigning a variable x_u to a node $v \in N_{G'}$ often strongly reduces variable domains, as the propagation of edge constraints removes from the domains of the variables associated with neighbours of u all nodes of $N_{G'}$ which are not neighbours of v. However, assigning a variable x_u to $\perp$ (*i.e.*, deciding

that u will not be matched) never reduces the domains of the other variables (except when the number of variables assigned to $\perp$ becomes equal to the cost bound so that $\perp$ is removed from all other domains). This explains why some subproblems are very easy to solve whereas others are much harder, even though they all have the same search space size.

In order to try to generate more balanced subproblems, we introduce another way of decomposing domains, called BIN hereafter. It is a straightforward adaptation of the decomposition of [4]. We perform a DBDFS, but instead of creating $|N_{G'}| + 1$ branches at each node (one for each possible value in the domain of the variable), we only create two branches: one where the variable is assigned to $\perp$, and one where $\perp$ is removed from the variable domain.

IV. Structural decomposition for MCIS

We introduce a new way to decompose MCIS into independent subproblems, guided by the structure of the problem. This method, called STR, is an adaptation of the TR-decomposition of [6] to the CP model of [8]. While TR-decomposition was initially defined for binary CSP instances, the CP model of [8] is a *soft* CSP. Nevertheless, MCIS may be solved by finding a maximum clique in the compatibility graph. Likewise [6], we propose to use the class of triangulated graphs to take advantage of their property to have a few maximal cliques.

More precisely, we triangulate the compatibility graph with the *MinFill* algorithm [17]. This algorithm adds edges, called *fill edges*. The fill edges add erroneous compatibilities so that a maximum clique in the triangulated graph is not invariably the best solution anymore, but simply is a *subproblem* in which we might find solutions. Actually, a maximum clique of the original compatibility graph is still a clique (not necessarily maximum or maximal) in the triangulated graph. Moreover, it is bound to appear in at least one maximal clique of the triangulated compatibility graph. Therefore, each maximal clique of the triangulated graph defines a subproblem (actually an induced subgraph of the compatibility graph) in which we may find a maximum clique of the initial compatibility graph. Such a subproblem can be seen as MCIS instances, with smaller graphs, that can be solved using the CP model of [8].

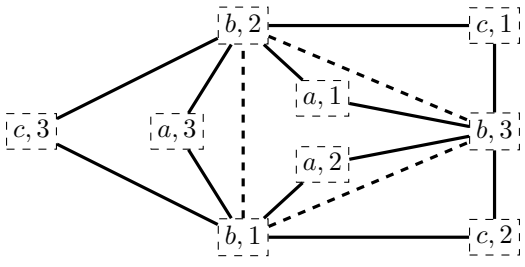


Figure 3. A triangulated version of the compatibility graph of Figure 2. The edges that were added (fill edges) are shown with dashed lines.

Algorithm 1: STR decomposition

Input: 2 graphs G and G', a number k of subproblems
Output: A set of subproblems

1 Build the compatibility graph G_C associated with G and G'
2 Triangulate G_C (with *MinFill* [17]) to obtain G_T
3 Compute the set S of all maximal cliques of G_T
4 Mark all cliques of S as decomposable
5 **while** $|S| < k$ *and* S *contains decomposable cliques* **do**
6 Remove from S its largest decomposable clique K
7 Let $G_{C\downarrow K}$ be the subgraph of G_C induced by K
8 Triangulate $G_{C\downarrow K}$ (with MinFill [17]) to obtain $G_{T\downarrow K}$
9 Compute the set S_K of all maximal cliques of $G_{T\downarrow K}$
10 **if** $S_K = \{K\}$ **then** mark K as non decomposable;
11 **else** mark all cliques of S_K as decomposable;
12 **for** *each clique* $K' \in S_K$ **do**
13 **if** $\forall K'' \in S, K' \not\subseteq K''$ **then** add K' to S;
14 **return** *the set of all subproblems associated with cliques of* S

More precisely, given a maximal clique K of the triangulation of the compatibility graph associated with G and G', we define a subproblem which has the same variables and constraints as the initial problem (described in II-B). However, the domain of every variable x_u is restricted to

$$D(x_u) = \{v \in N_{G'} \mid (u, v) \in K\} \cup \{\perp\}$$

The domains of $x_\perp$ and $cost$ remain unchanged. For example, in Figure 3, the subset of nodes $\{(b,1),(b,3),(c,2)\}$ is a maximal clique of the triangulated graph. In the associated subproblem, $D(x_b) = \{1, 3, \perp\}$, $D(x_c) = \{2, \perp\}$, and $D(x_a) = \{\perp\}$. Every solution of every subproblem corresponds to a maximal clique of the compatibility graph, and therefore to a common induced subgraph. In our example, the subproblem has two solutions of cost 1 (*i.e.*, $\{x_a = \perp, x_b = 1, x_c = 2\}$ and $\{x_a = \perp, x_b = 3, x_c = 2\}$) which are optimal and correspond to two maximal cliques of the compatibility graph (*i.e.*, $\{(b,1),(c,2)\}$ and $\{(b,3),(c,2)\}$) as edge $\{(b,1),(b,3)\}$ is a fill edge which has been added by the triangulation.

STR may generate subproblems with very unbalanced sizes. In order to generate better balanced subproblems, and also to control the number of subproblems that are generated, we propose to recursively decompose the largest subproblems by running TR-decomposition on them again. This is described in Algorithm 1. We first triangulate the compatibility graph G_C and store all its maximal cliques in a set S (lines 1–3). Then, while $|S|$ is lower than the required number of subproblems, we remove from it its largest clique K, triangulate the subgraph of G_C induced by K and compute the set S_K of all its maximal cliques (lines 6–9). If there is only one maximal clique in the triangulation of the subgraph of G_C induced by K, it means that K cannot be decomposed any more (line 10). Some cliques in S_K may be subsets of cliques in S. These non maximal cliques are not added to S; all other cliques of S_K are added to S

(lines 12–13). Finally, once S contains enough cliques (or no more clique is decomposable), we build a subproblem for each clique, and return this set of subproblems (line 14).

Some maximal cliques may have large intersections. In such a case, the corresponding subproblems also have large intersections, meaning that we might solve a same part of subproblems several times. In order to reduce redundancies, some subproblems are merged, *i.e.* they are replaced by a new subproblem obtained by merging their variables' domains. Note that any solution will still be present in the resulting subproblem, since domains can only get bigger during this process. To prevent subproblems from growing back to the size of the initial problem, we introduce the notion of *gain*, which expresses the evolution of the size during an hypothetical fusion.

Definition 7: Let S_1 and S_2 be two distinct subproblems. The *gain* offered by the fusion of S_1 and S_2 is given by the following quotient: $gain(S_1, S_2) = \frac{size(S_1)+size(S_2)}{size(S_1 \cup S_2)}$, where $size(S_i)$ is the product of the sizes of variable domains of S_i, and $S_1 \cup S_2$ represents the subproblem that would be created if S_1 and S_2 were to be merged.

We merge two subproblems if the gain is greater than 1, giving priority to pairs offering the largest gains.

V. Experimental evaluation

A. Experimental setup

Our algorithms are developed in C and compiled with GCC's `-O3` optimization option. Programs are executed with a time limit of three hours on an Intel® Xeon® CPU E5-2670 0 at 2.60 GHz processor, with $20,480$ KB of cache memory and 4 GB of RAM.

Instances are taken from the benchmark presented in [15] and are used in their labeled and directed versions. The number of different labels equals 15 % of the number of nodes in each graph. We take these labels into account like it is done in [8], by ensuring matched nodes have the same label. By similar rules, edges between pairs of matched nodes must also have the same label. Each domain is therefore reduced to values associated to nodes having the exact same label than the node associated to the variable.

The benchmark contains three classes of instances (see [15] for details):

- bvg *Bounded Valence Graphs*, where degrees cannot be higher than a given value for each graph.
- rand *Random graphs*, in which edges are set at random.
- mnD *Meshes* with two (m2D), three (m3D) or four (m4D) dimensions.

We consider instances such that graphs have a number of nodes n ranging from 10 up to 100, and maximum common subgraphs have a number of nodes ranging from $0.1n$ to $0.3n$.

The MCIS is an optimization problem. As pointed out in [1], when solving an optimization problem with a parallel approach, a key point is to solve first the most promising subproblems (which contain better solutions) in order to allow more pruning when solving the remaining subproblems. In this experimental study, our goal is to evaluate the capability of DOM, BIN and STR to generate balanced independent subproblems, and we don't want to introduce a bias with ordering heuristics for choosing the most promising subproblems. Furthermore, heuristic algorithms are able to very quickly find near-optimal solutions to MCIS. For example, the *AntClique* algorithm of [18] is able to find a maximum clique (corresponding to a MCIS) for 90 % of our instances in a few seconds. However, proving the optimality of these optimal solutions with our baseline sequential approach (*MAC+Bound* [8]) is still very challenging. Therefore, in this experimental study, we consider the problem of proving the optimality of a common subgraph. We selected instances which are solved within our time limit of three hours by our baseline sequential method. Since decomposition approaches are specifically designed for instances that are hard to solve, we reduced the scope to instances that are not solved in 100 seconds by the standard method. These constraints brought us to the final number of 109 instances.

For each instance, we began by testing STR. STR aims to generate a number of subproblems close to a parameter k by successive clique decompositions. In these experiments, k has been set to $1,500$. This number is then brought down to k' by subproblem fusions, as explained in Section IV. To keep our study as fair as possible, BIN and DOM are then asked to generate k' subproblems, thus solving as many subproblems as STR. Once the number of subproblems is fixed in this way (with a possibly different number for each instance), we try to solve those instances with different numbers of subproblems per workers (hence with different numbers of workers).

B. Notations

For an initial instance i defined by two graphs G and G', we introduce the following notations:

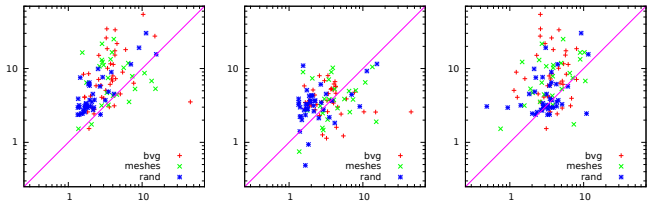
- s_0 is the size of i, *i.e.*, the product of the sizes of its domains;
- t_0 is the time to solve i;
- k' is the number of subproblems remaining after STR's fusions for i (at most $1,500$);
- t_{dec} is the time to decompose i into k' subproblems;
- s_{all} is the sum of sizes of all subproblems of i;
- t_{all} is the sum of times to solve all subproblems of i;
- t_{max} is the time to solve the hardest subproblem of i;
- t_{min} is the time to solve the easiest subproblem of i.

C. Search space size and decomposition time

When decomposing the initial instance into subproblems, some inconsistent values are filtered out, either by propagating constraints during DBDFS (for DOM and BIN), or when reconstructing domains from cliques (for STR). For each

Table I
REDUCTION OF SIZE AND DECOMPOSITION TIME.

	bvg		mesh		rand		all instances	
	$\frac{s_0}{s_{all}}$	$\frac{t_{dec}}{t_0}$	$\frac{s_0}{s_{all}}$	$\frac{t_{dec}}{t_0}$	$\frac{s_0}{s_{all}}$	$\frac{t_{dec}}{t_0}$	$\frac{s_0}{s_{all}}$	$\frac{t_{dec}}{t_0}$
DOM	$3e^{2}$	$2e^{-4}$	$1e^{2}$	$1e^{-4}$	$1e^{3}$	$6e^{-4}$	$6e^{2}$	$4e^{-4}$
BIN	1	$3e^{-5}$	1	$3e^{-5}$	1	$1e^{-4}$	1	$7e^{-5}$
STR	$5e^{8}$	$2e^{-2}$	$6e^{5}$	$3e^{-2}$	$1e^{6}$	$8e^{-2}$	$2e^{8}$	$5e^{-2}$



(a) DOM/STR (b) DOM/BIN (c) BIN/STR

Figure 4. Speedup bounds without t_{dec}: Each point (x, y) in (a) (resp. (b) and (c)) corresponds to an instance such that $x = t_0/t_{max}$ for DOM (resp. DOM and BIN) and $y = t_0/t_{max}$ for STR (resp. BIN and STR).

decomposition method, Table I displays s_0/s_{all}, followed by t_{dec}/t_0, on average for all instances of each class. It shows us that BIN does not filter domains at all so that the search space is not reduced (as pointed out in Section III). DOM (resp. STR) reduces the search space by a factor of 600 (resp. 200 millions), on average for all instances in all classes. This factor depends on instance classes and decomposition methods. In particular, for DOM (resp. STR), the search space reduction is lower than the average reduction on bvg and mesh (resp. mesh and rand) instances, whereas it is higher on rand (resp. bvg) instances.

Search space reduction should be considered together with the time spent for the decomposition (t_{dec}). Decomposition times of DOM and BIN are similar, and represent less than 0.04 % of t_0 (on average for all instances) for DOM and 0.007 % for BIN. Decomposition times of STR tend to be higher, and are close to 5 % of t_0.

D. Speedup bounds

The time needed to solve the initial problem divided by the time needed to solve the hardest subproblem (*i.e.*, t_0/t_{max}) provides a first upper bound on the speedup. For each decomposition method, Table II displays this bound (minimum, average and maximum values for each class

Table II
SPEEDUP BOUND WHEN IGNORING t_{dec} (*i.e.*, t_0/t_{max}).

	bvg			mesh			rand			all
	min	avg	max	min	avg	max	min	avg	max	avg
DOM	1.6	4.9	45.0	1.4	4.9	15.2	1.3	2.9	15.5	4.1
BIN	1.1	4.0	8.0	0.7	4.6	10.5	0.5	3.7	11.6	4.0
STR	1.5	11.1	54.3	1.5	9.1	25.1	2.4	5.5	30.2	8.4

Table III
SPEEDUP BOUND WHEN INTEGRATING t_{dec} (*i.e.*, $t_0/(t_{dec} + t_{max})$).

	bvg			mesh			rand			all
	min	avg	max	min	avg	max	min	avg	max	avg
DOM	1.6	4.8	44.6	1.4	4.9	15.2	1.3	2.9	15.4	4.1
BIN	1.1	4.0	8.0	0.7	4.6	10.5	0.5	3.7	11.5	4.0
STR	1.5	8.6	28.9	1.4	8.2	24.1	1.2	4.0	13.1	6.7

Table IV
SUBPROBLEM SOLVING TIME: MIN (t_{min}/t_0), AVG ($t_{all}/(k' \times t_0)$) AND MAX (t_{max}/t_0).

	bvg			mesh			rand		
	min	avg	max	min	avg	max	min	avg	max
DOM	$0.05e^{-3}$	$4e^{-3}$	$320e^{-3}$	$0.1e^{-3}$	$7e^{-3}$	$293e^{-3}$	$0.1e^{-3}$	$4e^{-3}$	$491e^{-3}$
BIN	$6e^{-3}$	$48e^{-3}$	$273e^{-3}$	$2e^{-3}$	$44e^{-3}$	$317e^{-3}$	$14e^{-3}$	$78e^{-3}$	$306e^{-3}$
STR	$0.1e^{-3}$	$15e^{-3}$	$175e^{-3}$	$0.1e^{-3}$	$12e^{-3}$	$179e^{-3}$	$0.3e^{-3}$	$38e^{-3}$	$265e^{-3}$

of instances). On average for all instances of all classes, the largest speedup bound is obtained with STR (8.4), the second largest by DOM (4.1), and the smallest by BIN (4.0). However, all these speedup bounds are rather low when considering the fact that each instance has been decomposed into 470 subproblems in average (min. 192, max. 1053). Again, we observe differences depending on the classes. In particular, for DOM and STR, the speedup bound is higher than the average on bvg and mesh instances, whereas it is lower on rand instances.

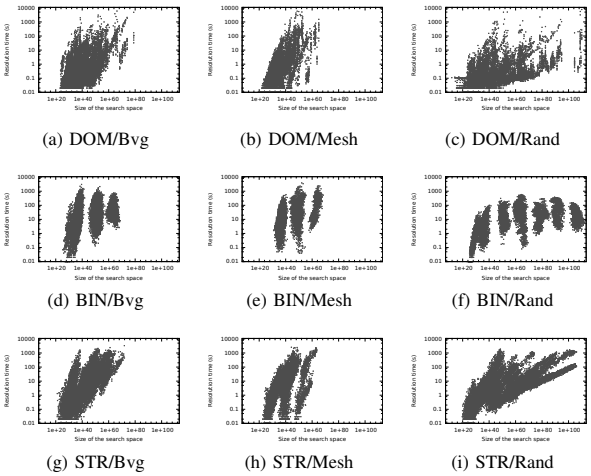
Figure 4(a) compares DOM with STR for each instance separately. It shows us that if for a majority of instances the speedup bound is larger with STR than with DOM, there are some instances (mainly of the bvg and mesh types) for which the bound is larger with DOM than with STR. Figure 4(b) and (c) compare DOM with BIN, and BIN with STR, and the same remark holds for these plots.

However, if the time spent by the decomposition step (t_{dec}) is rather small for DOM and BIN, it is larger for STR. Hence, a tighter upper bound on the speedup may be obtained by dividing t_0 by $t_{dec} + t_{max}$. These speedups are displayed in Table III. It can be noticed that STR still obtains the largest average speedup bounds, though DOM obtains rather close bounds.

E. Time needed to solve subproblems

If the time needed to solve the hardest subproblem is a bottleneck, the speedup also depends on the hardness of all subproblems. To give an insight into this hardness, Table IV displays the time ratio between the easiest subproblem and the initial instance (t_{min}/t_0), the average time ratio between a subproblem and the initial instance ($t_{all}/(k' \times t_0)$), and the time ratio between the hardest subproblem and the initial instance (t_{max}/t_0), on average for all instances of each class.

On average, STR generates harder subproblems than DOM: 4 times as hard for bvg, twice as hard for mesh and 10 times as hard for rand. On the other hand, the hardest



(a) DOM/Bvg (b) DOM/Mesh (c) DOM/Rand

(d) BIN/Bvg (e) BIN/Mesh (f) BIN/Rand

(g) STR/Bvg (h) STR/Mesh (i) STR/Rand

Figure 5. Size w.r.t. time: for each decomposition method $d \in \{\text{DOM}, \text{BIN}, \text{STR}\}$, and each instance class $c \in \{\text{bvg}, \text{mesh}, \text{rand}\}$, the subfigure d/c plots a point (x, y) for each subproblem obtained when decomposing an instance of c with d, where x is the size of the search space of this subproblem, and y is the time needed to solve it.

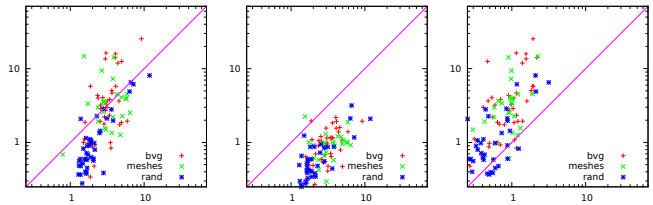
Table V
SPEEDUP WITH 30 SUBPROBLEMS PER WORKER.

	bvg			**mesh**			**rand**			**all**
	min	avg	max	min	avg	max	min	avg	max	avg
DOM	1.5	3.2	9.2	0.8	3.5	6.4	1.3	2.5	11.9	3.0
BIN	0.2	1.0	2.2	0.1	0.9	2.2	0.1	0.6	3.2	0.8
STR	0.3	4.8	25.4	0.6	4.0	14.8	0.3	1.5	8.1	3.3

subproblem for each instance is easier when decomposing with STR than when decomposing with DOM.

F. Correlation between time and search space size

Table IV shows us that the subproblems obtained when decomposing an initial instance are of very fluctuating difficulty. Actually, in most cases, there are a lot of very easy subproblems, and only very few hard subproblems. To obtain a speedup as close as possible to the bounds displayed in Table III, a key point is to be able to identify these hard instances so that we can solve them first and better balance the workload. Therefore, we plot in Figure 5 the relation between the size of the search space of a subproblem, and the time needed to solve it. It demonstrates that two subproblems with a same size may be of very different difficulty. However, we note that, for STR, the rightmost points (corresponding to the largest subproblems) tend to have large y values (corresponding to the hardest subproblems). This remark still holds for DOM, to a lesser extent. This is no longer true for BIN, for which there is no satisfying correlation between size and time. Note that for all decomposition methods, there are subproblems which have rather small sizes but which are hard to solve.



(a) DOM/STR (b) DOM/BIN (c) BIN/STR

Figure 6. Speedups with 30 subproblems per worker: Each point (x, y) in (a) (resp. (b) and (c)) corresponds to an instance such that x is the speedup for DOM (resp. DOM and BIN) and y for STR (resp. BIN and STR).

G. Speedup with 30 subproblems per worker

Table V displays the speedup observed with 30 subproblems per worker, for each decomposition method, and each class of instances (minimum, average and maximum speedups). This number of subproblems per worker was described as generally efficient by [5]. Let t_{real} be the total time spent by the last worker which has completed its work. The speedup is $t_0/(t_{dec} + t_{real})$.

We note that BIN has very low speedups. STR has better speedups than DOM on bvg and mesh instances, whereas it has a lower speedup on rand instances. These rankings basically correspond to rankings observed on speedup bounds for DOM and STR.

Let us finally note that these speedups are quite low, even for the best decomposition method STR. As a comparison, [4] reports results on 20 CSPs (and many instances for each CSP). With 40 workers and $40 \times 30 = 1,200$ subproblems by instance, the average speedup is 21.3, and the worst speedup is 8.6.

H. Speedup with different numbers of workers

Experiments in [4], [5] show us that a good EPS decomposition is generally obtained by generating about 30 subproblems per worker. In this study, we generated approximately as many subproblems with each method and tried solving them with different numbers of workers, thus varying the number of subproblems per worker.

Figure 7 compares speedups obtained with STR and DOM when varying the number of subproblems per worker from 1 to 50. When the number of subproblems per worker is higher than 35, speedups with DOM are higher than speedups with STR. However, when the number of subproblems per worker is lower than 35, STR obtains better speedups than DOM. This may come from the fact that even though STR generates harder subproblems, on average, than DOM (see Table IV), its hardest subproblems are easier.

VI. CONCLUSION

We have introduced a structural decomposition method for the MCIS problem derived from the TR-decomposition

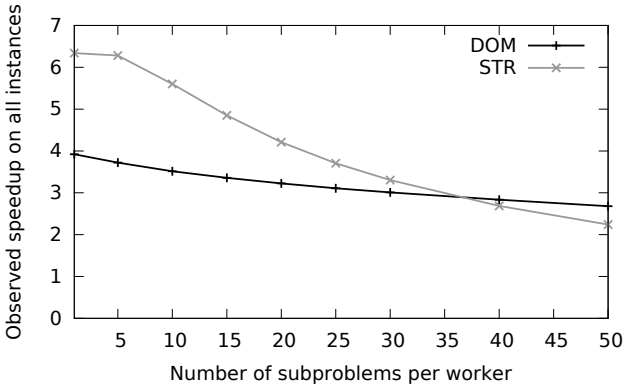


Figure 7. Observed speedups with different numbers of workers, for DOM and STR.

method defined in [6]. It relies on a triangulation of the compatibility graph of the problem and allows to decompose the problem into several independent subproblems. We have experimentally compared this structural decomposition with the domain decomposition method proposed by [4], showing that it leads to higher speedups for two classes of instances, and to weaker speedups for one class.

The decomposition of the initial instance into subproblems is time-consuming. In this paper, this decomposition step has not been parallelized. [5] proposes to parallelize this decomposition step, and this allows them to obtain better speedups. Our structural decomposition method could be parallelized. In particular, instead of treating each clique K separately (in the loop lines 5–13 of Algorithm 1), we could treat several cliques in parallel.

ACKNOWLEDGMENT

This work has been supported by the ANR project SoL-StiCe (ANR-13-BS02-0002-01).

REFERENCES

[1] C. McCreesh and P. Prosser, "The shape of the search tree for the maximum clique problem and the implications for parallel branch and bound," *ACM Transactions on Parallel Computing*, vol. 2, no. 1, p. 8, 2015.

[2] M. Depolli, J. Konc, K. Rozman, R. Trobec, and D. Janezic, "Exact parallel maximum clique algorithm for general and protein graphs," *Journal of chemical information and modeling*, vol. 53, no. 9, pp. 2217–2228, 2013.

[3] C. McCreesh and P. Prosser, "Multi-threading a state-of-the-art maximum clique algorithm," *Algorithms*, vol. 6, no. 4, pp. 618–635, 2013.

[4] J. Régin, M. Rezgui, and A. Malapert, "Embarrassingly parallel search," in *Principles and Practice of Constraint Programming - CP 2013, Uppsala, 2013.*, 2013, pp. 596–610.

[5] J. Régin, M. Rezgui, and A. Malapert, "Improvement of the embarrassingly parallel search for data centers," in *Principles and Practice of Constraint Programming - CP 2014, Lyon, 2014.*, 2014, pp. 622–635.

[6] P. Jégou, "Decomposition of domains based on the micro-structure of finite constraint-satisfaction problems," in *AAAI*, vol. 93, 1993, pp. 731–736.

[7] A. Chmeiss, P. Jégou, and L. Keddar, "On a generalization of triangulated graphs for domains decomposition of csps," in *IJCAI*, 2003, pp. 203–208.

[8] S. N. Ndiaye and C. Solnon, "Cp models for maximum common subgraph problems," in *Principles and Practice of Constraint Programming–CP 2011*. Springer, 2011, pp. 637–644.

[9] J. J. McGregor, "Backtrack search algorithms and the maximal common subgraph problem," *Software: Practice and Experience*, vol. 12, no. 1, pp. 23–34, 1982.

[10] D. Sabin and E. C. Freuder, "Contradicting conventional wisdom in constraint satisfaction," in *Principles and Practice of Constraint Programming*. Springer, 1994, pp. 10–20.

[11] T. Petit, J.-C. Régin, and C. Bessière, "Specific filtering algorithms for over-constrained problems," in *Principles and Practice of Constraint Programming – CP 2001*. Springer, 2001, pp. 451–463.

[12] E. Balas and C. S. Yu, "Finding a maximum clique in an arbitrary graph," *SIAM Journal on Computing*, vol. 15, no. 4, pp. 1054–1068, 1986.

[13] P. J. Durand, R. Pasari, J. W. Baker, and C.-c. Tsai, "An efficient algorithm for similarity analysis of molecules," *Internet Journal of Chemistry*, vol. 2, no. 17, pp. 1–16, 1999.

[14] J. W. Raymond, E. J. Gardiner, and P. Willett, "Rascal: calculation of graph similarity using maximum common edge subgraphs," *The Computer Journal*, vol. 45, no. 6, pp. 631–644, 2002.

[15] D. Conte, P. Foggia, and M. Vento, "Challenging complexity of maximum common subgraph detection algorithms: A performance analysis of three algorithms on a wide database of graphs." *J. Graph Algorithms Appl.*, vol. 11, no. 1, pp. 99–143, 2007.

[16] R. Dechter and J. Pearl, "Tree-Clustering for Constraint Networks," *Artificial Intelligence*, vol. 38, pp. 353–366, 1989.

[17] U. Kjaerulff, "Triangulation of graphs - algorithms giving small total state space," Judex R.R. Aalborg., Denmark, Tech. Rep., 1990.

[18] C. Solnon and S. Fenet, "A study of aco capabilities for solving the maximum clique problem," *Journal of Heuristics*, vol. 12, no. 3, pp. 155–180, 2006.