Using Graph Decomposition for Solving Continuous CSPs

Christian Bliek[1], Bertrand Neveu[2], and Gilles Trombettoni[1]

[1] Artificial Intelligence Laboratory, EPFL
CH-1015 Lausanne, Switzerland
{cbliek,trombe}@lia.di.epfl.ch
[2] CERMICS, équipe Contraintes
2004 route des lucioles, 06902 Sophia-Antipolis Cedex, B.P. 93, France
Bertrand.Neveu@sophia.inria.fr

Abstract. In practice, constraint satisfaction problems are often structured. By exploiting this structure, solving algorithms can make important gains in performance. In this paper, we focus on structured continuous CSPs defined by systems of equations. We use graph decomposition techniques to decompose the constraint graph into a directed acyclic graph of small blocks. We present new algorithms to solve decomposed problems which solve the blocks in partial order and perform intelligent backtracking when a block has no solution.
For under-constrained problems, the solution space can be explored by choosing some variables as input parameters. However, in this case, the decomposition is no longer unique and some choices lead to decompositions with smaller blocks than others. We present an algorithm for selecting the input parameters that lead to good decompositions.
First experimental results indicate that, even on small problems, significant speedups can be obtained using these algorithms.

1 Introduction

In the area of continuous CSPs, research has traditionally focused on techniques to enforce some form of local consistency. These techniques are used in combination with dichotomic search to find solutions. We use Numerica, a state of the art system for solving the specific type of CSPs considered in this paper [Hentenryck *et al.*, 1997].

In practice, constraint satisfaction problems are often structured. However, little has been done to exploit the structure of continuous CSPs to make gains in performance. In this paper, we focus on efficient solution strategies for solving structured CSPs. We will restrict our attention to CSPs which are defined by nonlinear equations and study the general case in which the system is not necessarily square. This paper brings together techniques to decompose constraint graphs with backtracking algorithms to solve the decomposed systems.

Although our approach is general, we have chosen to present 2D mechanical configuration examples. By doing so, we do not want to convey that our approach applies only to this type of problems. We mainly use these examples for didactical reasons; they are easy to understand and to illustrate.

2 The Dulmage and Mendelsohn Decomposition

In this paper, *a constraint graph* G is a bipartite graph (V, C, E) where V are the variables, C are the constraints and there is an arc between a constraint in C and each of its variables in V.

A *maximum matching* of a bipartite constraint graph includes a maximum number of arcs which share no vertex. A matching implicitly gives a direction to the corresponding constraint graph; a pair (v, c) corresponds to a directed arc from c to v and directed arcs from v to other matched constraints connected to v. The D&M decomposition is based on the following theorem.

Theorem 1 *(Dulmage and Mendelsohn, 1958) Any maximum-matching of a constraint graph G gives a canonical partition of the vertices in G into three disjoint subsets: the under-constrained part, the over-constrained part and the well-constrained part.*

Observe that one or two of the three parts may be empty in the general case.

Starting from any maximum matching of the graph, the over-constrained part is formed with all nodes reachable from any non matched constraint in C by a reverse path. The under-constrained part is formed with all nodes reachable from any non matched variable in V by a directed path. The well-constrained part is formed with the other nodes and yields a perfect matching of the corresponding subgraph. Figure 1 shows an example. Variables are represented by circles, constraints by rectangles; a pair in the matching is depicted by an ellipse. The well-constrained part can be further decomposed. The perfect matching of

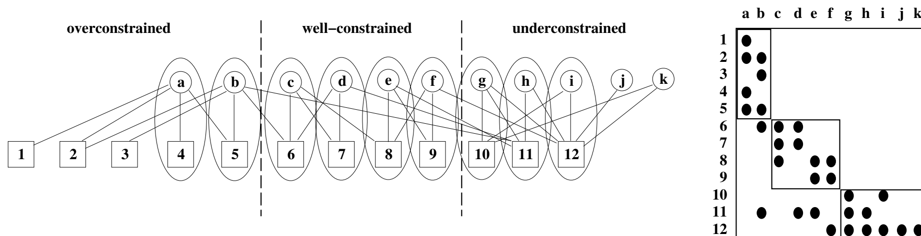

Fig. 1. The D&M decomposition of a constraint graph (left) and the equivalent matrix representation (right).

this part implicitly defines a directed graph. We then compute its strongly connected components, called *blocks*, to obtain a directed acyclic graph (DAG) of blocks.

It turns out that this decomposition, called the *fine* decomposition, is independent of the matching performed.

Theorem 2 *(König, 1916) Every perfect matching of a (square) bipartite graph leads to a unique decomposition into strongly connected components.*

Note that König's theorem does not apply in case of non perfect matching.

3 Overview

In this part, we give a general overview of the algorithms described in this paper and of how they work together. As input, we have a set of numeric equations which may be non-square. We make the general assumption that square $n \times n$ systems give a discrete set of solutions. This is in fact a basic assumption of Numerica [Hentenryck *et al.*, 1997], the tool we use to solve systems of equations. In a number of pathological cases, this assumption is not verified, then we can for example use probabilistic methods to diagnose the problem (see Section 7).

We first perform a D&M decomposition [Pothen and Chin-Fan, 1990] and handle the three parts, if present, in the order over-constrained, well-constrained and finally under-constrained part.

Over-constrained Part In this part, the number of equations m is greater than the number n of variables. If the corresponding equations are independent, the system has no solution. We can deal with this situation in a number of ways. First, through a backtrack-free selection process, we could let the user remove $m - n$ equations to make the part well-constrained[1]. Alternatively, the $m - n$ extra constraints might be kept as *soft* constraints. They are simply verified after the solution process. If the $m - n$ equations are redundant, we get solutions, otherwise there is no solution.

Well-constrained Part As explained in Section 2, for this part, we can perform a fine decomposition. The result is a DAG of blocks. Each block is solved by Numerica and, in the subsequent blocks, the implied variables are replaced by the values found. To ensure completeness, backtracking is performed as soon as a block has no solution. Section 5 describes several intelligent backtracking algorithms that take advantage of the partial order of the DAG to avoid useless work. Observe that this process looks like the resolution of a finite CSP: the blocks are the variables and the solutions to a block form the domain of the variable.

Under-constrained Part Once we have solved the well-constrained part and replaced the variables by the values found, we are left with the under-constrained part. This part contains n variables and m equations with $n > m$. At this point $r = n - m$ *driving input variables* (*divs*) must be given a value to obtain a $m \times m$ problem. There are a number of issues related to the selection of the r *divs* in the under-constrained part.

First, some sets of r input variables may lead to "badly-constrained" systems, that is, systems for which there exists no perfect matching. This means that we cannot arbitrarily choose r of the n variables as *divs*. Section 6.1 presents a new algorithm which allows the selection of the r *divs* one by one and forbids certain future choices after each selection. This approach might be used for example in physical simulation [Serrano, 1987], where the user explicitly changes different parameters in order to understand the behavior of a system.

[1] This process is the dual of the one for removing variables in the under-constrained part as described in Section 6.1.

Second, König's theorem does not hold for the under-constrained part. That is, the DAG of blocks obtained is not unique and depends on the matching. In particular, certain choices of *divs* lead to decompositions with smaller blocks than others and are usually easier to solve. Section 6 presents new algorithms to find decompositions with small blocks. Interactive sketching applications (see [Bliek *et al.*, 1998]) should favor this type of decompositions.

4 Examples

In this section, we present 2 examples that will be used throughout this paper.

Didactic Example (Figure 2) Various points (white circles) are connected with

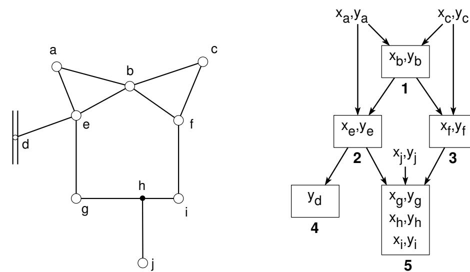

Fig. 2. Didactic problem and a corresponding DAG of blocks.

rigid rods (lines). Rods only impose a distance constraint between two points. Point h (black circle) differs from the others in that it is attached to the rod $\langle g, i \rangle$. Finally, point d is constrained to slide on the specified line. The problem is to find a feasible configuration of the points so that all constraints are satisfied.

Mini-robot (Figure 3) The triangle a–b–c represents the base and is fixed. Since the robot is symmetrical, let us describe the left half. Point d is the left shoulder. The rod $\langle a, d \rangle$ has a variable length r_s, and is used to rotate the rod $\langle c, d \rangle$. On this rod we have an arm mechanism whose length is variable and depends on the parameter r_a. The gray point e is constrained to slide on the rod $\langle c, d \rangle$. The black point f is attached to both the rod $\langle e, g \rangle$ and $\langle d, h \rangle$, and hereby forces these rods to act like a scissor. The position of the end point i of the arm can now be positioned by controlling r_a and r_s.

The decomposition shown in the right side of Figure 3 corresponds to the above description of the robot. That is, r_a, r'_a, r_s and r'_s are the driving inputs that control the endpoints. Suppose however that we want the robot to pick up an object. In this case the end points i and i' should go to a specific location. The question now is: what values should r_a, r'_a, r_s, r'_s take to get the two endpoints at the desired spot? The decomposition of this inverse problem is shown in the middle of Figure 3.

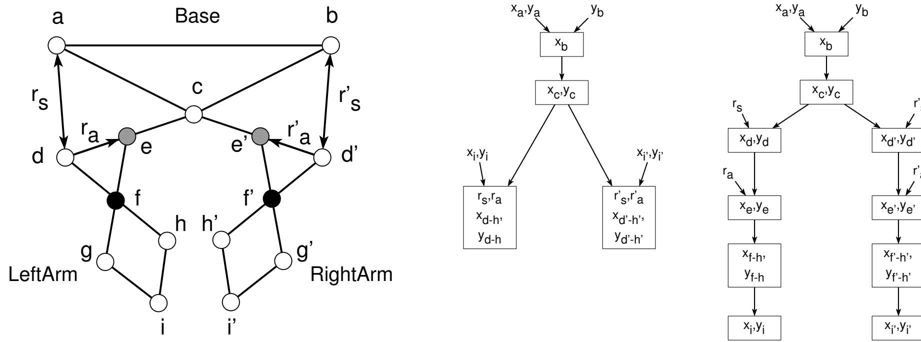

Fig. 3. Minirobot (left), DAG of inverse problem (middle) and driving problem (right).

Finally, consider the situation in which a designer is sketching the points and rods for this robot using a computer, and wants to validate his drawing. In this case, the input variables may be chosen so that the validation problem is as simple as possible. In Section 6, we present an algorithm that is designed to do exactly that. In this example, the best solution is to take y_d, y_b, x_i, x'_i, y'_e, x'_d, y'_d as input parameters, which leads to a decomposition with one block of size 5 and other blocks of size 1, 2 or 3.

5 Solving Well-Constrained Systems

5.1 Introduction

In case the D&M decomposition consists only of a well constrained part, we can perform a fine decomposition. By doing so we avoid to send the system as a whole to the solver. Indeed, we will only need to solve smaller systems which correspond to the blocks in the DAG. To see this, consider the decomposition shown in Figure 2 on the right. Respecting the order of the DAG, we first obtain a solution for block 1. We can now substitute the values for the corresponding variables appearing in the equations of block 2 and obtain a solution from the solver. Then we process block 3 in a similar fashion, followed by block 4 and 5.

When a block has no solution, one has to backtrack. A chronological backtracker goes back to the previous block. It tries a different solution for that block and restarts to solve the subsequent blocks. However, this approach is inefficient. Indeed, in the example above, suppose block 5 had no solution. Chronological backtracking would go back to block 4, find a different solution for it, and solve block 5 again. Clearly, the same failure will be encountered again in block 5.

A better strategy is to reconsider only those blocks which might have caused the problem. We could follow the approach used by Conflict Based Backjumping (`CBJ`) [Prosser, 1993]. When no solution is found for block 5, one would go back directly to block 3. However, when jumping back, `CBJ` erases all intermediate search information. Here, the solution of block 4 would be erased when jumping

back to block 3. This is unfortunate; the solution of block 4 is still valid and the solver may have spent a considerable amount of time finding it.

This problem can be avoided by holding on to the intermediate search information. This approach is taken in dynamic backtracking (DB) [Ginsberg, 1993]. As CBJ, DB jumps back to the cause of the problem. However, when doing so, it does not erase intermediate nogoods. Instead, it moves the problem variable to the set of uninstantiated variables and removes only nogoods based on its assignment. By doing so, DB effectively reorders variables. Unfortunately, this reordering is incompatible with the partial order imposed by the DAG. We therefore need to resort to algorithms that keep intermediate search information but are also flexible enough to respect the partial order imposed by the decomposition. General Partial Order Backtracking (GPB) [Bliek, 1998] satisfies these requirements. Below we present a specific instance of GPB that can be used to solve decomposed problems.

5.2 Solving Partially Ordered Problems with GPB

We first briefly describe the GPB algorithm for solving discrete CSPs. At all times GPB maintains a complete set of assignments X which are incrementally modified until all constraints are satisfied. The search process is driven by the addition of new nogoods. A *nogood* γ is a subset of assignments of values to variables which are incompatible. X is modified incrementally so as to remain compatible with the current set of nogoods. When a new nogood is added, the value of one of its variables, say y, will be changed. By choosing y, γ becomes an ordered nogood, denoted by $\vec{\gamma}$. We call y the *conclusion* variable, denoted by $c(\vec{\gamma})$, and the remaining variables *antecedent* variables $a(\vec{\gamma})$. An ordered nogood $\vec{\gamma}$ defines an ordering relation $x < c(\vec{\gamma})$ for each antecedent $x \in a(\vec{\gamma})$.

Nogoods are generated when the current assignments violate a constraint. In this case the nogood is the constraint violation. Nogoods are also generated when for a domain of a given variable y, all values are ruled out by nogoods. In this case a new nogood $\beta(y)$ is inferred that contains the assignments of all the antecedent variables $a(\vec{\gamma}_i)$ appearing in every nogood $\vec{\gamma}_i$ with $y = c(\vec{\gamma}_i)$. In addition to the conclusion variables of a nogood, one may also modify the assignment of other variables in X, as long as the new values are acceptable. A value v is *acceptable* for the variable x if $x = v$ is compatible with the antecedents of any of the current nogoods and if v is in the live domain of x. The *live domain* of a variable is the set of values of its domain that is not ruled out by a conclusion of a nogood. When a new nogood with conclusion y is added, all nogoods $\vec{\gamma}_i$ for which $y \in a(\vec{\gamma}_i)$ are discarded. By doing so, an assignment is ruled out by at most one nogood. The space complexity of GPB is therefore polynomial. The problem has no solution when the empty nogood is inferred.

We now present an instance of GPB, called GPB$_I$. We will see that this algorithm can easily be adapted to solve decomposed continuous CSPs. Instead of modifying a complete set of assignments, GPB$_I$ incrementally extends a consistent partial set of assignments. We therefore have a set of instantiated variables I and a set of uninstantiated variables U. To ensure termination, it is required

that the selection of the conclusion of a nogood respects a partial order. In $\mathtt{GPB}_I$ we use an ordering scheme $<_I$ defined as follows. The variables in I respect the total order defined by the instantiation sequence and any variable in I precedes any variable in U. Abusing notation, we define the antecedents of a variable y as $a(y) = \{x \mid x < y\}$. The descendants of y are defined by $D(y) = \{x \mid y <_t x\}$, where $<_t$ is the transitive closure of $<$. With these definitions, we modify GPB to obtain $\mathtt{GPB}_I$.

```
algorithm GPB_I
    Until U is empty or the empty nogood is inferred do
        Select a variable x ∈ U for which a(x) ⊆ I and
        assign an acceptable value v to x.
        if x = v violates some constraint with variables in I then
            Generate a nogood γ corresponding to the constraint violation and
            Backtrack(γ),
        else
            move x from U to I.
        end
    end
end.
procedure Backtrack (γ)
    Select y as conclusion of γ so that y follows a(γ⃗) in the ordering scheme and
    store γ⃗.
    Discard all nogoods γ⃗_i for which y ∈ a(γ⃗_i) and
    move y and the variables D(y) in I to U.
    if the live domain of y is empty then
        Backtrack(β(y)).
    end
end.
```

Algorithm 1: $\mathbf{GPB_I}$

$\mathtt{GPB}_I$ is an instance of GPB and therefore terminates. It is also systematic since it satisfies an additional restriction on the assignments that may be changed. We refer the reader to [Bliek *et al.*, 1998] for a detailed discussion.

As described above, $\mathtt{GPB}_I$ halts as soon as it finds a solution. To find all solutions to a given CSP, the algorithm can be modified as follows. When a solution is found, it is reported and a new nogood is generated that rules out exactly this set of assignments. We then backtrack from this nogood and restart the search loop to find the next solution.

$\mathtt{GPB}_I$ solves discrete CSPs. We now adapt it to solve continuous problems that are decomposed into a DAG of blocks. Here the blocks take over the role of the variables. The discrete domain[2] of possible values for a block x is the set

[2] As stated earlier, we assume that we have a discrete set of solutions for each block.

of solutions, denoted by $\sigma(x)$, of the corresponding subproblem. Recall that, for a given block x, the solutions of the parent blocks, denoted by $p(x)$, are first substituted into the equations of block x. The resulting system is then solved over the continuous domains of the variables of the block. By solving the system, all values in the continuous domains are eliminated, except for $\sigma(x)$. One can view this as the addition of a nogood, denoted by $P(x)$, that has the given block x as conclusion and $p(x)$ as antecedent variables.

We can now modify $\texttt{GPB}_I$ to solve decomposed problems. The main difference is that the domain $\sigma(x)$ of a block x is not known *a priori* and has to be computed based on the values of $p(x)$. We therefore have to make sure that $\sigma(x)$ is recomputed every time any of the values of $p(x)$ changes. The resulting algorithm is called GPB_Δ. The backtrack procedure remains the same. However,

algorithm $\mathbf{GPB}_\Delta$
- **Until** *U is empty or the empty nogood is inferred* **do**
  - select a block $x \in U$ for which $a(x) \cup p(x) \subseteq I$
  - **if** *$\sigma(x)$ is outdated with respect to $p(x)$* **then**
    - recompute $\sigma(x)$ using the new values for $p(x)$ and backtrack($P(x)$),
  - **else**
    - assign an acceptable value v to x and move x from U to I.
  - **end**
- **end**

end.

Algorithm 2: $\mathbf{GPB}_\Delta$

in a practical implementation, care has to be taken to represent and handle the nogoods of the type $P(x)$.

In Figure 4, we illustrate $\texttt{GPB}_\Delta$ on the example of Figure 2. Suppose we solve block 1 and select first the solution where b is above a and c. Then we proceed to solve block 2 and select the solution where e is above a and b. We make a similar choice for f whose solutions are computed in block 3. Now, in block 4, we select one of the two possible locations for d. Finally, we reach block 5 to find out that there is no solution (dashed). This situation is shown on the left in Figure 4. At this point, we add a nogood $P(5)$ which is based on the solutions of blocks 2 and 3 and rules out all possible values for block 5. We find that the domain of 5 is empty and infer a nogood $\beta(5)$ which states that the solutions of block 2 and 3 are incompatible. Since block 3 was instantiated after block 2 we select block 3 as conclusion of this nogood. We now select the other solution for this block where point f is below b and c. Once again we find that block 5 has no solution (second picture in Figure 4) and backtrack to block 3. However, this time, both possible solutions of block 3 are ruled out by nogoods, so we continue backtracking. Since the two nogoods have block 2 as antecedent and $P(3)$ has block 1 as antecedent, we generate a new nogood stating that the solutions of

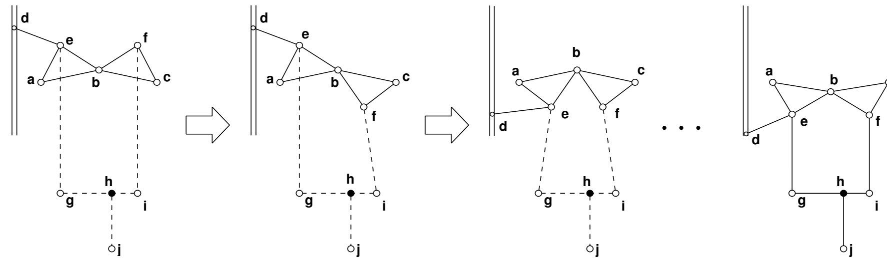

Fig. 4. Example problem.

block 1 and 2 are incompatible. As shown in the third picture in Figure 4, we now use the other solution for block 2. This search process continues until we reach the configuration, depicted on the right in Figure 4, that satisfies all constraints.

Observe that $\mathtt{GPB}_\Delta$, did not erase the solutions of block 4 when it backtracked to block 3. However, they would have been erased by `CBJ`, which would later need to recompute them.

5.3 Examples

We found that, as compared to solving the system as a whole, the use of $\mathtt{GPB}_\Delta$ on decomposed systems is very effective. Let us illustrate this point with some examples. As it is usually done when solving CSPs defined by systems of equations, we report the running times to find all solutions. All the tests were performed on a Sun SparcStation 5 with Ilog Numerica 1.0.

On the small didactic example of Figure 2, $\mathtt{GPB}_\Delta$ needs only 2.9 seconds, while without decomposition Numerica needs 4.8 seconds. The speedups are more important when the examples are somewhat more complicated. Consider the mini-robot positioning problem shown in Figure 3. Without decomposing the problem, it takes Numerica 2153 seconds to solve the problem. Using the decomposition shown in the middle in Figure 3, with $\mathtt{GPB}_\Delta$ this running time is reduced to 33.3 seconds.

In some cases very small blocks can be found. In this case, the backtrack search over the discrete sets of solutions is dominant. Figure 5 shows an example of such a situation. As before, arcs represent distance constraints and we have fixed the two coordinates, x_a and y_a, of point a as well as one coordinate, x_b, of point b. In addition to what is shown in the figure, there are two variables that measure the height of each of the legs defined as $r = (y_i + y_h)/2$ and $r' = (y_{i'} + y_{h'})/2$. We limited the possible range on these two variables to stay within the interval ρ shown in Figure 5. By doing so, there is only one solution which is the one shown in the figure. Without decomposition, it takes Numerica 2091 seconds to solve this problem. With the decomposition shown on the right in Figure 5, $\mathtt{GPB}_\Delta$ solves the problem in 29 seconds.

6 Handling Structurally Under-constrained Problems

As discussed in Section 3, in this case, we need to find r *divs* such that the remaining constraint graph has a perfect matching. Two different approaches for

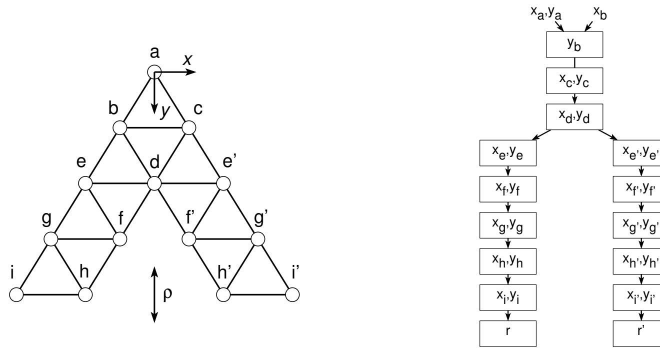

Fig. 5. A configuration of triangles corresponding to a pyramid tower.

doing so are presented below. Once the *divs* are selected, the problem becomes well-constrained and can be solved by GPB_{Δ}.

6.1 Backtrack-Free Driving Inputs' Selection

The algorithm shown below is based on the D&M properties. It allows to choose the *divs* one by one in a backtrack-free manner. The time complexity of the algo-

```
algorithm Free-divs-selection (a constraint graph; its D&M decomposition)
    while r divs have not been chosen do
        choose as div any variable v in the under-constrained part
        if v is matched in the current matching then
            – invert in the current matching an alternating path from v to an un-
            matched variable
            – apply a D&M decomposition on the new matching (which transfers
            some nodes from the under-constrained to the well-constrained part)
        end
    end
end.
```

Algorithm 3: The backtrack-free *div* selection

rithm is $O(r \times (n+m))$ for a constraint graph with n variables and m constraints. Indeed, one path inversion and one "D&M part retrieval" is necessary for each of the r *div* selections. Figure 6 illustrates this algorithm. The correctness of the backtrack-free *div* selection follows directly from the D&M properties. The proof can be found in the extended version of this paper [Bliek *et al.*, 1998].

6.2 Finding Small Blocks: OpenPlan_{sb}

We present below an algorithm called $OpenPlan_{sb}$[3] which finds decompositions with well-constrained square blocks whose largest block is of minimum size.

[3] *sb* stands for *small blocks.*

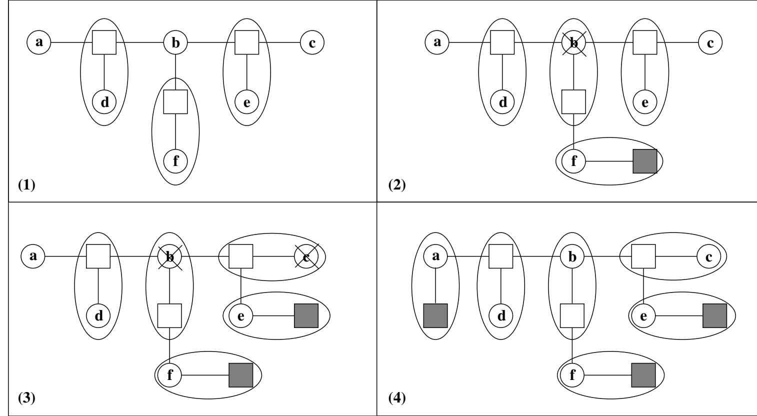

Fig. 6. Backtrack-free *div* selection. (1) Initial maximum matching. 3 *divs* are to be selected. (2) Selection of variable f as *div*: variable b is forbidden for further selection. (3) The selection of e forbids c. (4) The selection of a makes the problem well-constrained.

```
algorithm OpenPlan_sb (G: a constraint graph): a DAG of blocks
    let D be an empty DAG of blocks
    while constraints remain in the constraint graph do
        select-free-block_sb: select a free square block b of smallest size for which there
        exists a perfect matching
        add b in D (along with the corresponding directed arcs)
        remove b from G
    end
    return D
end.
```

Algorithm 4: OpenPlan$_{sb}$

OpenPlan$_{sb}$ is a specialized version of an algorithm called OpenPlan: the procedure *select-free-block* of OpenPlan can select any free block whereas *select-free-block*$_{sb}$ imposes restrictions. OpenPlan is based on the PDOF algorithm used for maintaining constraints in interactive applications [Vander Zanden, 1996]. It builds a DAG of blocks in reverse order from the leaves to the roots. A *free* block has variables which are linked only to constraints within the block. Iteratively selecting and removing blocks which are free ensures that a DAG of blocks is built, that is, that no directed cycle can appear between blocks [Trombettoni, 1997].

We now detail how OpenPlan$_{sb}$ finds the best decomposition of the constraint graph of the didactic problem. The process is illustrated on the right side of Figure 7. First, the block $[c_7,y_d]$, a 1×1 free block, is selected and removed: x_d becomes a *div*. Now there is no more 1×1 free block available so that a 2×2 free block, for example $[c_1,c_3,x_a,y_a]$, is selected and removed. Then, the free block $[c_4,c_5,x_c,y_c]$ is selected and removed. This frees the block $[c_6,c_9,x_f,y_f]$ which is selected and removed. The block $[c_2,x_b]$ is now free; y_b becomes a *div*. In the

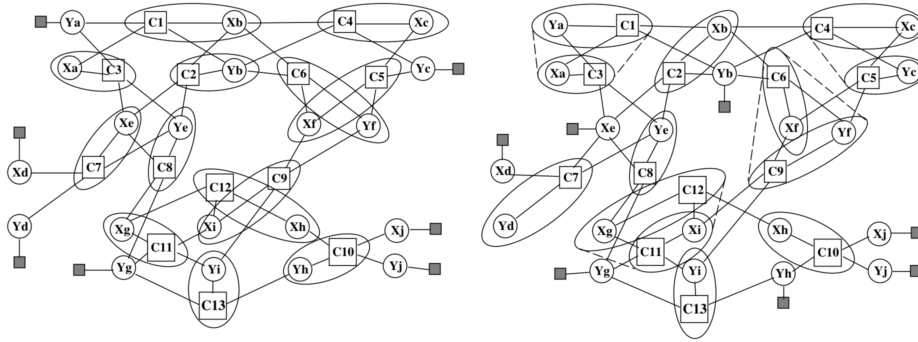

Fig. 7. A decomposition of the didactic problem with one 13×13 block (left) and a decomposition with four 2×2 blocks (a, b, f, (x_g, x_i)) and several 1×1 blocks (right).

same way, OpenPlan$_{sb}$ finally selects the blocks [c_8,y_e], [c_{11},c_{12},x_g,x_i], [c_{13},y_i] and [c_{10},x_h].

Proposition 1 *OpenPlan$_{sb}$ finds a decomposition whose largest block has a minimum size.*

The proof can be found in the extended paper [Bliek *et al.*, 1998].

Note that, as opposed to OpenPlan$_{sb}$, the maximum-matching algorithm finds decompositions with blocks of arbitrary size: both decompositions of Figure 7 could be indifferently obtained by maximum-matching whereas OpenPlan$_{sb}$ finds the one in the right.

Selecting a 1×1 free block is the only operation involved in the classical PDOF. This amounts to searching variables linked to only one constraint. However, if there is no 1×1 free block, *finding a well-constrained free block of minimum size is suspected to be NP-hard*, so that OpenPlan$_{sb}$ is exponential. Indeed, a naive algorithm which searches for a $k \times k$ valid block is $O(n^k)$, where n is the number of variables.

Nevertheless, this algorithm can sometimes be used for problems which may be numerically hard to solve. In this case, the overhead for obtaining the best decomposition can be neglected over the gain in solving small blocks. Our first experimental results presented in Section 6.4 seem to confirm this.

6.3 Finding Small Blocks: OpenPlan$_{hm}$

When the constraint graphs are large, we have to resort to heuristic methods to find good decompositions. To do so, another instance of OpenPlan, called OpenPlan$_{hm}$, is proposed for which the procedure select-free-block$_{hm}$ is a hill-climbing heuristic. If no 1×1 block has been found, a maximum matching of the constraint graph is performed. We consider first the smallest leaf b_{best} of this matching to be selected by *select-free-block$_{hm}$*. However, b_{best} is not the smallest possible free block, since other matchings could yield smaller leaves. Therefore, *select-free-block$_{hm}$* changes the matching so that a smaller leaf could appear. Instead of performing a full search over all matchings, a hill-climbing heuristic

is used. One tries to "break" b_{best} by inverting a path in the current matching from a *div* to a variable in b_{best}. If such a path is found which yields a smaller leaf, the process is reitered until reaching a fixed point. Note that the DAG of blocks may significantly change after a path inversion so that completely new blocks may appear.

```
function select-free-block_hm (G: a constraint graph): a free block
    if there exists a 1 × 1 free block b then return b
    perform a maximum matching of G that yields a DAG of blocks D_best
    let b_best be the smallest leaf block of D_best
    while D_best is changing do
        let cdivs be the set of divs in D_best such that an alternating path exists from
        a div in cdivs to a variable in b_best
        D ← D_best; b ← b_best
        for every div d in cdivs and every variable v in b_best do
            invert an alternating path from d to v that yields a DAG of blocks D'
            if D' has a smaller leaf block b' than b then D ← D'; b ← b'
        end
        D_best ← D; b_best ← b
    end
    return b_best
end.
```

Algorithm 5: Heuristic method to select a small well-constrained free block

In Figure 7, we show how OpenPlan_{hm} obtains the best decomposition (right) starting from the matching corresponding to the worst one (left). *select-free-block*$_{hm}$ first inverts the path from y_d to x_e, which yields the first free (1×1) block $[c_7, y_d]$. Then it selects the block $[c_1, c_3, x_a, y_a]$ by inverting the path from y_a to x_b. Now it can proceed until the best decomposition is obtained.

6.4 First Experimental Results

On the didactic problem, the *divs* on the left in Figure 7 lead to one unique block solved by Numerica in 10 s. With the *divs* in Figure 7 (right), the problem is solved by GPB_{Δ} in 3.3 s.

We also performed tests on a small distance problem in 3 D made of two tetrahedra and some additional rods shown in Figure 8.

The worst decomposition of this problem has two blocks of size 7. The best decomposition, obtained by both OpenPlan_{sb} and OpenPlan_{hm}, includes blocks of size 2 or 3 only. When the driving inputs correspond to the bad decomposition, Numerica takes 1284.8 s to solve the whole system. Solving the same problem with GPB_{Δ} takes 125.4 s. Now when the driving inputs correspond to the good decomposition, Numerica on the whole system takes 200 s whereas only 22.7 s are necessary to solve the same problem using GPB_{Δ}.

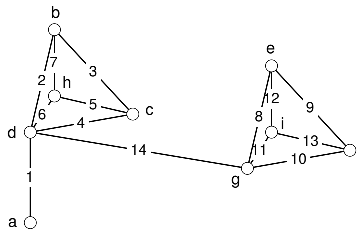

Fig. 8. A 3 D linkage with two tetrahedra.

7 Related Work

To our knowledge, no existing system which performs graph decomposition combines complete numerical solvers with backtracking. The system presented in [Serrano, 1987] uses a maximum-matching algorithm to decompose general design problems. However, completeness is not achieved; blocks are solved using a traditional Newton-Raphson method and no backtracking is performed. Furthermore, this work is not based on the D&M decomposition. In particular, the driving inputs cannot be selected in a backtrack-free fashion.

In [Ait-Aoudia *et al.*, 1993], the D&M technique is used to study geometric constraints. However, no attention is paid to the solution aspects.

The D&M decomposition does not take into account the values of the coefficients of the equations. However, some problems, whose D&M decomposition is structurally well constrained, are in fact made of dependent equations. One way to detect redundancy in nonlinear systems of equations is to calculate the Jacobian at various randomly selected points [Lamure and Michelucci, 1997]. This information could then be used to properly decompose this type of systems.

In the case of mechanical configuration, there exist a number of techniques to discover subsystems that are rigid [Fudos and Hoffmann, 1997]. By replacing these rigid subsystems by smaller ones, gains in performance can be made. This technique is complementary to ours and could be used to further improve the performance of our algorithms on mechanical configuration problems.

8 Conclusion

In this paper, we have presented techniques to solve structured continuous CSPs. Our approach is based on decomposition techniques by Dulmage & Mendelsohn and König, that decompose structured problems into a directed acyclic graph of blocks. The contribution of this paper is twofold. First, we propose new algorithms for solving structurally well-constrained problems. They combine the use of existing solvers, for solving the blocks, with intelligent backtracking techniques that use the partial order of the DAG to avoid useless work. Second, we present new algorithms to handle under-constrained problems. These algorithms allow the selection of driving input variables, whose values are assumed to be set externally. Input variables can either be selected through an interactive backtrack-free selection or can be selected automatically using a new algorithm to obtain de-

compositions with small blocks. We have presented a number of examples to illustrate that significant speedups can be obtained using these algorithms.

References

[Ait-Aoudia *et al.*, 1993] Samy Ait-Aoudia, Roland Jegou, and Dominique Michelucci. Reduction of constraint systems. In *Compugraphic*, 1993.

[Bliek *et al.*, 1998] Christian Bliek, Bertrand Neveu, and Gilles Trombettoni. Using graph decomposition for solving continuous csps. Technical Report 98-287, E.P.F.L., Lausanne, Switzerland, 1998.

[Bliek, 1998] Christian Bliek. Generalizing dynamic and partial order backtracking. In *AAAI 98: Fifteenth National Conference on Artificial Intelligence*, pages 319–325, Madison, Wisconsin, July 1998.

[Fudos and Hoffmann, 1997] Ioannis Fudos and Christoph Hoffmann. A graph-constructive approach to solving systems of geometric constraints. *ACM Transactions on Graphics*, 16(2):179–216, 1997.

[Ginsberg, 1993] M.L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, August 1993.

[Hentenryck *et al.*, 1997] Pascal Van Hentenryck, Laurent Michel, and Yves Deville. *Numerica : A Modeling Language for Global Optimization.* MIT Press, 1997.

[Lamure and Michelucci, 1997] Hervé Lamure and Dominique Michelucci. Qualitative study of geometric constraints. In Beat Brüderlin and Dieter Roller, editors, *Workshop on Geometric Constraint Solving and Applications*, pages 134–145, Technical University of Ilmenau, Germany, 1997.

[Pothen and Chin-Fan, 1990] Alex Pothen and Jun Chin-Fan. Computing the block triangular form of a sparse matrix. *ACM Transactions on Mathematical Software*, 16(4):303–324, 1990.

[Prosser, 1993] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, August 1993.

[Serrano, 1987] D. Serrano. *Constraint Management in Conceptual Design.* PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, October 1987.

[Trombettoni, 1997] Gilles Trombettoni. *Solution Maintenance of Constraint Systems Based on Local Propagation.* PhD thesis, University of Nice-Sophia Antipolis, 1997. In french.

[Vander Zanden, 1996] Bradley Vander Zanden. An incremental algorithm for satisfying hierarchies of multi-way, dataflow constraints. *ACM Transactions on Programming Languages and Systems*, 18(1):30–72, January 1996.