A Generalisation of the Backtracking Algorithm

M.R.C. van Dongen*

Cork Constraint Computation Centre, Computer Science Department, University College Cork,
dongen@cs.ucc.ie

Abstract. This paper presents a generalisation of the well known chronological backtracking algorithm. The algorithm is a generalisation in the sense that it can use *any* kind of constraint (as opposed to just unary constraints—the domains of the variables) to decompose a problem into smaller problems. This choice keeps the height of the search tree less than or equal to the number of variables. Most importantly, however, this choice will never increase the local branching factor of the search tree but will frequently make it smaller.

1 Introduction

This paper presents tools to analyse and dissect constraints. The tools are the building blocks for a new *generalised backtracking algorithm* which is a generalisation of the well known chronological backtracking algorithm. Generalised backtracking is sound and complete.

The motivation for the generalised backtracking algorithm is as follows. It has been observed in the mathematical community that a solution strategy for systems of multivariate polynomial equations where Gröbner bases with respect to total degree orders are factorised[1] and the induced problems are solved is to be preferred to a strategy where lexicographical Gröbner bases (elimination ideals/strict "lexicographical" rules) dictate the order in which equations should be used to decompose the problem [1, 3, 10, 15, 16, 18]. The reasons are two-fold. Firstly, the Gröbner bases with respect to total degree orders are (normally) easier to compute. Secondly, the polynomials occurring in the total degree bases (normally) have a lower degree thereby leading to factors of lower degree. Gräbe furthermore observes that problems coming from real life often fulfill the condition of being factorisable [10].

For example, in the presence of a *binary* algebraic constraint $(x-1)\times(x-y) = 0$ it is probably a good decision to factorise it to get two constraints $x = 1$ and $y = x$, to branch on these two constraints, and to do a bit of extra work to remove

* This work has received support from Science Foundation Ireland under Grant 00/PI.1/C075.

[1] Here and in the remainder of this paper a factorisation of a problem is a set of problems the union of whose solutions are equal to the solutions of the original problem. This is analogous to the notion of factorisation of a polynomial: the zeros of the original polynomial are equal to the zeros of the members of the factorisation.

duplicate solutions even if there is a *unary* constraint $(x-1) \times \cdots \times (x-d) = 0$, for some $d > 2$, for which a factorisation is known. There are at least two reasons why this is probably better. The first reason is that branching on this binary constraint requires only 2 branches as opposed to $d > 2$ branches for the unary constraint. The second reason is that the resulting constraints $x = 1$ and $y = x$ are about as tight as you can get. Both allow for the elimination of a variable, just like each of the members $x = i$, $1 \leq i \leq d$, of the factorisation of the unary constraint allow for the elimination of a single variable.

The strategy used by the chronological backtracking algorithm is similar to the "lexicographical" approach mentioned before. The generalised backtracking algorithm on the other hand is motivated by similar observations as the "total degree" approach. To be more specific, the generalised backtracking algorithm is not restricted to the use of unary constraints (the domains of the variables) alone to decompose problems.

Both the chronological backtracking algorithm and the generalised backtracking algorithm traverse search trees. The chronological backtracker decomposes problems at each internal node of the search tree by considering a unary constraint (the domain of a variable). For each member in the unary constraint it creates a sub-problem. The number of sub-problems that has to be considered is equal to the cardinality of the unary constraint. In terms of tree traversals the unary constraint determines the (local) number of branches whose sub-trees have to be traversed. This number is called the *local branching factor*. As already indicated, the generalised backtracking algorithm can use *any* kind of constraint to obtain a problem decomposition. It will be demonstrated that this will never result in a higher local branching factor but may result in a lower local branching factor in return for a marginal increase in the space complexity.

The chronological backtracking algorithm has received much attention from many researchers. Variants of the algorithm range from a vanilla version [9], to forward checking [11] and MAC [20], and to backjumping [6], conflict directed backjumping [19], and dynamic backtracking [8]. For detailed treatments of and surveys of backtracking the reader may wish to consult [5, 13, 14, 17]. The reader may wish to consult [7, 21] for an introductory treatment to backtracking.

It is a well established fact that in order to keep (backtrack) search efficient it is imperative that the branching factors of the nodes near the root of the search tree be kept as small as possible. The contribution of generalised backtracking is that it is the first attempt to keep the branching factor of the search tree small by analysing the structure of *any* kind of constraint and by using an alternative (exhaustive) way to enumerate the members of the constraint. This alternative way to explore the search space generalises the notion of static and dynamic variable orders.

The remainder of this paper is organised as follows. Section 2 is a brief introduction to constraint satisfaction. Section 3 introduces the notions of *covers* and *partitions* of constraints. This is followed by Section 4 which introduces the notions of a the *degree* of a constraint of *linear* constraints and shows how linear constraints can be used to simplify CSPs. Arguments are presented that this

simplification corresponds to a "localised" breadth-first search. Special kinds of partitions of constraints are discussed in Section 5. Section 5 also describes the generalised backtracking algorithm. Experimental results are discussed in Section 6. A summary is presented in Section 7.

2 Constraints and CSPs

Let $\cdot \prec \cdot$ be the usual lexicographical variable ordering. In this paper it is assumed that if x_i and x_j are two variables then $x_i \prec x_j \iff i < j$. Let $T = \{ x_{i_1}, \ldots, x_{i_m} \}$ be a non-empty set of variables. For the sake of this paper a *constraint* among the members of T is a subset of the cartesian product of the domains of the members of T. However, this is by no means a necessity. If C_T is a constraint then it will be assumed that the constraint is between the members of T. In addition it will be assumed that if $v \in C_T$ then $(x_{i_1}, \ldots, x_{i_m}) = v$ is a valid "assignment" which is allowed by C_T. A *Constraint Satisfaction Problem* (CSP) is a tuple (X, D, C), where X is a set of variables, $D(\cdot)$ is a function mapping each member of X to its domain, and C is a set of constraints.

3 Covers and Partitions of Constraints

This section presents methods to decompose any CSP into several CSPs whose solutions are pairwise disjoint and the union of whose solutions is equal to the solutions of the original CSP. It is shown that certain kinds of decompositions are essentially the same as the decompositions that are (implicitly) computed by the chronological backtracking algorithm.

Let S be a set and let 2^S denote the *power set* of S, i.e. the set of all subsets of S. A set $\kappa \subseteq 2^S$ is called a *cover* of S if $S = \cup_{c \in \kappa} c$. The set containing all covers of S is denoted $K(S)$, i.e. $K(S) = \left\{ \kappa \subseteq 2^S : S = \cup_{c \in \kappa} c \right\}$. Partitions are covers whose members are pairwise disjoint. The set of all partitions of S is denoted $\Pi(S)$. The *maximal partition* of a set S is the set $\{ \{ s \} : s \in S \}$.

The following proposition tells us that if we factorise a constraint C_T from a given CSP then the solutions of that CSP are equal to the union of the solutions of the CSPs that can be obtained by replacing C_T by the members of its factorisation.

Proposition 1 (Covers of Constraints). *Let* (X, D, C) *be a* CSP*, let* $T \subseteq X$*, let* $C_T \in C$*, let* $C'(c) = (C \setminus \{ C_T \}) \cup \{ c \}$ *and let* $\cdot \bowtie \cdot$ *denote* natural join*. The following holds for all covers* κ *of* C_T*:*

$$\bowtie_{c \in C} c = \bigcup_{c' \in \kappa} \bowtie_{c \in C'(c')} c \,.$$

To prove the proposition is not difficult. A formal proof may be found in [22]. Notice that partitions are covers. Therefore, Proposition 1 also applies to partitions and maximal partitions.

Proposition 1 allows for the decomposition of a CSP into a collection of CSPs. The collection represents the CSP in the sense that the union of the solutions of the members of that collection is equal to the solutions of that CSP. The following example demonstrates how Proposition 1 can be used to explain how the chronological backtracking algorithm works.

Example 2 (Chronological Backtracking). Let $\mathcal{C} = (X, D, C)$ be the CSP, where $X = \{x, y\}$, $C = \{C_{\{x\}}, C_{\{y\}}, C_{\{x,y\}}\}$, $D(x) = \{1, 2\}$, $D(y) = \{1, 2, 3\}$, $C_{\{x\}} = \{1, 2\}$, $C_{\{y\}} = \{1, 2, 3\}$ and $C_{\{x,y\}} = \{(1, 1), (1, 2), (2, 3)\}$. The solution set of $\mathcal{C}$ is $C_{\{x,y\}}$. To backtrack with x as the current variable corresponds to the application of Proposition 1 to the CSP for the maximal partition $\pi = \left\{C'_{\{x\}}, C''_{\{x\}}\right\}$, where $C'_{\{x\}} = \{1\}$ and $C''_{\{x\}} = \{2\}$. The application of Proposition 1 to π allows for the decomposition of the constraint $C_{\{x\}}$ into the two constraints $C'_{\{x\}}$ and $C''_{\{x\}}$ that can be used to dissect $\mathcal{C}$ into the two CSPs $\mathcal{C}'$ and $\mathcal{C}''$, where

$$\mathcal{C}' = \left(X, D, \left\{C'_{\{x\}}, C_{\{y\}}, C_{\{x,y\}}\right\}\right),$$
$$\mathcal{C}'' = \left(X, D, \left\{C''_{\{x\}}, C_{\{y\}}, C_{\{x,y\}}\right\}\right).$$

The solutions of $\mathcal{C}'$ are given by $C'_{\{x\}} \bowtie C_{\{y\}} \bowtie C_{\{x,y\}} = \{(1, 1), (1, 2)\}$ and the solutions of $\mathcal{C}''$ are given by $C''_{\{x\}} \bowtie C_{\{y\}} \bowtie C_{\{x,y\}} = \{(2, 3)\}$. The union of the solutions of $\mathcal{C}'$ and $\mathcal{C}''$ is equal to the solution set of $\mathcal{C}$. If the "standard" lexicographical heuristics are used then a chronological depth-first backtracking algorithm will first solve $\mathcal{C}'$ and then solve $\mathcal{C}''$.

4 Linear Constraints

This section discusses how to use certain properties of constraints to simplify binary CSPs. In particular it is shown that what are called *linear* constraints (many-to-one-relations)[2] can be used to simplify binary CSPs. The simplifications consist of a transformation of a binary CSP to a binary CSP where a variable has been eliminated (modulo renaming). The sizes of the domains in the resulting CSP are less than or equal to the sizes of the domains in the original CSP. The number of binary constraints in the resulting CSP is less than the number of binary constraints in the original CSP. It is argued that such transformation can be regarded as a "localised" breadth-first search.

[2] A suitable name for linear constraints could also have been *functional* constraints had it not been for the fact that there is a "name clash" for such constraints. For example, [23] and [4] both use functional constraints with a different meaning. Functional constraints in the context of [23] are what will be called *bi-linear* constraints here further on. They correspond to one-to-one relations. Functional constraints in the context of [4] correspond to the notion of what will be called linear binary constraints in this work.

Constraints that have finitely many members can be translated to polynomial ideals (systems of polynomial equations) and vice versa [22, Chapter 4]. This suggests that constraints also have "degrees" and "total degrees." The following is an attempt to generalise these notions of degree and total degree to that of the degree of a set of variables in a constraint. It is a reformulation of the notion presented in [22, Chapter 5]. Notions similar to that of the degree of a constraint do not seem to have appeared before. The number of substitutions of values for a variable in a polynomial for which the polynomial vanishes ("becomes" zero) corresponds to a branching factor of a constraint in a search tree. As will be shown later, the degree of a set of variables in a constraint relates these variables to the branching factor. Constraints with low degrees correspond to low branching factors in search.

```
function deg(C_T, S) : Integer
var R, Partitions, Projections;
begin
    if C_T = ∅ then
        return 0;
    else if |S| = 1 then
        R := T \ S;
        Projections := { projection of t onto R : t ∈ C_T };
        return max_{p ∈ Projections} |{ t ∈ C_T : p = projection of t onto R }|;
    else
        Partitions := { π ∈ Π(C_T) : (∀c ∈ π)(∃x ∈ S)(deg(c, { x }) = 1) };
        return min({ |π| : π ∈ Partitions });
    fi;
end;
```

Fig. 1. Degree function.

Definition 3 (Degree). Let S and T be non-empty sets of variables such that $S \subseteq T$. Furthermore, let C_T be a constraint whose cardinality is finite, and let $\deg(\cdot,\cdot)$ be the function depicted in Figure 1. The number $\deg(C_T, S)$ is called the *degree* of S in C_T. The degree of $\{ x \}$ in C_T is also called the *degree* of x in C_T or the *x-degree* of C_T.

If $|S| = 1$ then the degree of S in C_T is the maximum number of solutions for the variable in S that are "allowed" by C_T given fixed assignments to the variables in $T \setminus S$. For binary constraints the degree of x in $C_{\{ x,y \}}$ is the maximum support size in $D(x)$ for y. A constraint C_T is called *linear* in $x \in T$ if the degree of x in C_T is one. A binary constraint $C_{\{ x,y \}}$ is called *bi-linear* if it is linear in both x and y. A constraint C_T is called *sub-linear* (in $S \subseteq T$) if $C_T = \emptyset$. Finally, a constraint C_T is called *linear* if it is linear in some variable in T.

Example 4 (Degree). Consider the binary constraint $C_{\{ x,y \}}$ given by:

$$C_{\{ x,y \}} = \{ (0,0), (0,1), (0,2), (0,3), (1,0), (2,0) \} .$$

The constraint $C_{\{x,y\}}$ is not linear in x. For example, the support for $y = 0$—the tuples whose projection onto y is equal to 0—is $\{0, 1, 2\}$. Therefore, the degree of x in $C_{\{x,y\}}$ is at least three. Similarly, $C_{\{x,y\}}$ is not linear in y either. It is left to the reader to verify that the degree of x in $C_{\{x,y\}}$ is three and that the degree of y in $C_{\{x,y\}}$ is four. Note that $d_{xy} = \deg(C_{\{x,y\}}, \{x, y\}) > 1$ because $C_{\{x,y\}}$ is non-empty and is neither linear in x nor linear in y. However, it can be shown that $d_{xy} = 2$. For example, consider the following two constraints:

$$C'_{\{x,y\}} = \{(0,1),(0,2),(0,3)\},$$
$$C''_{\{x,y\}} = \{(0,0),(1,0),(2,0)\}.$$

Both $C'_{\{x,y\}}$ and $C''_{\{x,y\}}$ are linear. The former is linear in x and the latter is linear in y. The set $\pi = \left\{ C'_{\{x,y\}}, C''_{\{x,y\}} \right\}$ is a partition of $C_{\{x,y\}}$. It follows from the definition of the degree of $\{x, y\}$ in $C_{\{x,y\}}$ that $d_{xy} \leq |\pi| = 2$. As observed before $C_{\{x,y\}}$ is not linear in x or in y. Therefore, $d_{xy} > 1$. Clearly, $d_{xy} = 2$.

Linear constraints can be used to simplify binary CSPs. If an arc-consistent constraint $C_{\{x,y\}}$ is linear then the variables x and y can be *amalgamated* into a "super-variable" which represents the values in the Cartesian product of the domains of x and y that are in $C_{\{x,y\}}$. The cardinality of the domain of the super-variable is equal to $\max(|D(x)|, |D(y)|)$ and the number of constraints in the resulting CSP will be less than the number of constraints of the original CSP. The transformation will leave all constraints of the form $C_{\{w\}}$ or $C_{\{w,z\}}$ intact, for w and $z \notin \{x, y\}$.

Thus, linear binary constraints allow for the elimination of a variable without causing an increase in the domain sizes of the variables or the number of constraints. The remainder of this section provides concrete examples about the flavour of linear constraints and how to exploit their properties.

Example 5 (Singleton Domains). During backtrack search it often occurs that the domain of a future variable reduces to a singleton set. Let x be such variable.

If the problem is binary and if the problem is arc-consistent then x can be removed. Should there be solutions then the projections of these solutions onto the domain of x will be the value in its domain.

If C_T is a constraint which involves x and if the domain of x is a singleton then C_T is linear or sub-linear in x. C_T is linear or sub-linear in x because any assignment to the variables in $T \setminus \{x\}$ which satisfies the projection of C_T onto $T \setminus \{x\}$ can be extended to at most one assignment to the variables in T such that this extended assignment satisfies C_T. If the projection of C_T onto the domain of x is non-empty then the constraint C_T can be contracted to a constraint on $T \setminus \{x\}$ without "losing" any solutions; the solutions for x can always be recovered.

The *reason* why the solutions can be recovered is that T is linear in x. Therefore, there is a function from the variables in $T \setminus \{x\}$ to x which can

be used to "recover" x. If C_T is binary, then the contraction of C_T entails the creation of a unary constraint on the remaining variable, say y, in $T \setminus \{ x \}$. If the problem is arc-consistent then the contraction of C_T is the same as $D(y)$ and it can be ignored. In binary CSPs that are arc-consistent, variables whose domains are singletons can therefore be eliminated.

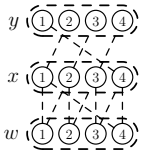

Fig. 2. Before Amalgamation.

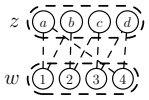

Fig. 3. After Amalgamation.

In the previous example it was argued that a variable x whose domain is a singleton can be removed from binary arc-consistent CSPs because there is a function from the variables in $T \setminus \{ x \}$ to x and that this mapping could be used to recover the value of x. This is *exactly* the same reason as the one upon which MAC (a backtracker which maintains arc-consistency [20]) relies, namely that after the assignment of a value to the current variable and after arc-consistency processing the current variable can be removed from a problem if it is arc-consistent because there is a function from the future variables to the value in the domain of the current variable. Some people may argue that x can be removed because its *only* value has been "saved" and can therefore be recovered. Other people may argue that after the assignment to x any arc-consistent constraint between x and another variable has "become" universal and can therefore be removed. However, the concept of x being "dependent on" a function is more general because, as the following example will demonstrate, it allows for the simultaneous recovery of *several different* values as opposed to only one.

Example 6 (Amalgamation of Nodes (1)). Consider the binary CSP whose micro-structure is depicted in Figure 2. The dashed lines represent the tuples that are allowed. The constraint $C_{\{ w,x \}}$ is not linear. The remaining constraint $C_{\{ x,y \}}$ is bi-linear.

Consider the sub-problem consisting of the two variables x and y, their domains, and the constraint $C_{\{ x,y \}}$. As it turns out the sub-problem has exactly four solutions. The nodes x and y can be transformed into a new node z whose domain contains four values a, b, c and d without increasing the maximum domain size. The transformation is such that these four values represent the four solutions of the sub-problem.

Figure 3 depicts the micro-structure of the same CSP where x and y have been "amalgamated" into one fresh variable z. The value a in the domain of z represents the tuple $(x, y) = (1, 2)$, b corresponds to $(x, y) = (2, 3)$, c corresponds to $(x, y) = (3, 4)$, and d corresponds to $(x, y) = (4, 1)$. The problem is

satisfiable if and only if the original problem is satisfiable, and its solutions are in one-to-one correspondence with the solutions of its original problem. The structure of the new problem is simpler than that of the original problem. Transformations, like the one from the CSP whose micro-structure is depicted in Figure 2 to the CSP whose micro-structure is depicted in Figure 3 may also be regarded as the elimination of a variable which linearly depends on a linear constraint (modulo renaming).

For binary CSPs the worst-case time-complexity for the detection of *all* linear constraints is in $\mathcal{O}(ed^2)$. This is exactly the worst case time-complexity for making a CSP arc-consistent, an "overhead" which is considered to be well spent by researchers in the constraint satisfaction area. It is not difficult to see how to incorporate part of the work for the detection of linear constraints into existing arc-consistency algorithms. If the domain sizes are large then most binary constraints are not linear and this can be found out without much overhead. The reason why this does not require much overhead is that it is not difficult (on average) to detect that there are at least two tuples in a binary constraint whose first members are equal and to find two tuples in a binary constraint whose second members are equal. However, when domain sizes become small a relatively large proportion of all the possible binary constraints are linear. To detect that a constraint is linear is relatively easy if the sizes of the domains are small.

The following example demonstrates that, in the presence of constraint propagation, to amalgamate the variables of a linear binary constraint does not only allow for the elimination of variables but may sometimes allow for the elimination of values.

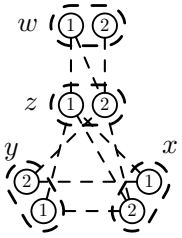

Fig. 4. Before Amalgamation of x and y.

Fig. 5. After Amalgamation of x and y.

Example 7 (Amalgamation of Nodes (2)). Consider the CSP whose micro-structure is depicted in Figure 4. The CSP consists of four variables w, x, y and z, their domains, and four binary constraints $C_{\{\,w,z\,\}}$, $C_{\{\,x,z\,\}}$, $C_{\{\,x,y\,\}}$, and $C_{\{\,y,z\,\}}$. The CSP is arc-consistent. The binary constraints $C_{\{\,w,z\,\}}$, $C_{\{\,x,z\,\}}$, $C_{\{\,x,y\,\}}$, and $C_{\{\,y,z\,\}}$ which were mentioned before are explicit. Besides these explicit constraints there are also implicit constraints. These constraints are determined by projections, (natural) joins, and intersections of constraints. For example, there

is an implicit constraint $C_{\{x,y,z\}}$ between x, y and z. $C_{\{x,y,z\}} = \{(2,1,1)\}$. The constraint between x, y, and z may also be considered as a constraint between x and y on the one hand and z on the other. This constraint is given by $C_{\{(x,y),z\}} = \{((2,1),1)\}$. The members of $C_{\{(x,y),z\}}$ are in one-to-one correspondence to the solutions of the sub-problem involving the variables x, y, and z, their domains, and the constraints $C_{\{x,y\}}$, $C_{\{x,z\}}$, and $C_{\{y,z\}}$. $C_{\{x,y\}}$ is linear. Therefore, the nodes x and y can be amalgamated into a node u whose domain contains one value for each of the tuples that are "allowed" by $C_{\{x,y\}}$ without increasing the size of the domains. Renaming (x,y) to u, $(1,2)$ to a, and $(2,1)$ to b results in a CSP the solutions of which are in one-to-one correspondence with the solutions of the original CSP. In particular, $(w,z,u) = (1,1,b) \iff (w,x,y,z) = (1,2,1,1)$. The micro-structure of the resulting CSP is depicted in Figure 5. Note that the CSP is not arc-consistent. The values 2 in the domain of z and a in the domain of u have lost support as a "result" of the constraint $C_{\{(x,y),z\}}$ which was implicit between x, y, and z. It is straightforward to make the CSP arc-consistent again.

To conclude this section it should be observed that the amalgamation of two variables x and y may be regarded as a "localised" breadth-first search of depth two. To see why this is true observe that the domain of the amalgamation of two variables contains the representatives of the values in the constraint between the variables. This set is equal to the set containing the allowed assignments of the nodes of the search tree at depth two which uses an ordering where x and y are the first variables. The advantage of amalgamation is that no decision has to be made yet about which value to assign to which variable, that it simplifies the problem, and that it allows for cheap constraint propagation. The advantage of not making a decision about which variable is to become the next current variable is that to postpone this decision may avoid assignments leading to traversals of sub-trees that are infeasibly large.

5 Linear Partitions and Generalised Backtracking

The previous sections have demonstrated the usefulness of partitions of constraints and linear constraints. This section studies a special kind of partition called *linear* partitions and a function to compute such partitions. The function is non-trivial in the sense that the cardinality of its result is "low." We shall see that linear partitions and the transformation to amalgamate nodes can be used to enumerate the nodes in the search tree of constraints more efficiently than chronological backtracking.

In the following let $\mathrm{proj}_{x_{i_j}}(\cdot)$ be the *projection function* defined as follows:

$$\mathrm{proj}_{x_{i_j}}((v_{i_1},\ldots,v_{i_m})) = \begin{cases} v_{i_j} & \text{if } 1 \le j \le m; \\ \bot & \text{otherwise.} \end{cases}$$

Definition 8 (Layer). Let S be a non-empty set of variables, let $x \in S$, and let C_S be a non-empty constraint. Furthermore, let $C_{\{x\}} = \{\mathrm{proj}_x(t) : t \in C_S\}$. A

set $S_x \subseteq C_S$ is called an *x-layer* of C_S if $|S_x| = |C_{\{x\}}|$ and $\{ \mathrm{proj}_x(t) : t \in S_x \} = C_{\{x\}}$.

Example 9 (Layer). Let $C_{\{x,y\}} = \{ (0,0), (0,1), (1,2) \}$. There are two x-layers of $C_{\{x,y\}}$. They are given by $\{ (0,0), (1,2) \}$ and by $\{ (0,1), (1,2) \}$. The only y-layer of $C_{\{x,y\}}$ is given by $C_{\{x,y\}}$ itself.

Lemma 10 (Linearity of Layers of Binary Constraints). *Let $C_{\{x,y\}}$ be a non-empty binary constraint and let $(z, z') \in \{ (x,y), (y,x) \}$. If S_z is a z-layer of $C_{\{x,y\}}$ then S_z is linear in z'.*

Proof. Let $S_z = \{ (x_1, y_1), \ldots, (x_m, y_m) \}$. Without loss of generality assume that $z = x$. Then $x_i \neq x_j \Leftrightarrow i \neq j$, for $1 \leq i, j \leq m$, and $x_i \mapsto y_i$ defines a *function* from $z = x$ to $y = z'$.

Lemma 11 (Monotonicity). *Let $C_{\{x,y\}}$ be a binary constraint, let $z \in \{x, y\}$, and let S_z be a z-layer of $C_{\{x,y\}}$. Furthermore, let $d_w = \deg(C_{\{x,y\}}, \{w\})$, and let $d'_w = \deg(C_{\{x,y\}} \setminus S_z, w)$, for $w \in \{x, y\}$. Then $\min(d'_x, d'_y) < \min(d_x, d_y)$.*

Proof. Trivial.

```
function P_l(C_i) :
begin var B_i, C_{i+1}, R_i, z;
    if C_i = ∅ then
        return ∅;
    else
        if deg(C_i, {x}) > deg(C_i, {y}) then
            z := x;
        else if deg(C_i, {x}) < deg(C_i, {y}) then
            z := y;
        else
            z := any member from {x, y};
        fi;
        B_i := any z-layer of C_i;
        C_{i+1} := C_i \ B_i;
        R_i := {B_i} ∪ P_l(C_{i+1});
        return R_i;
    fi;
end;
```

Fig. 6. Partition function

A *linear partition* is a partition whose members are linear. The following defines a function to transform a binary constraint to a linear partition of that constraint.

Proposition 12 (Linear Partition of Binary Constraint). *Let $C_{\{x,y\}}$ be a non-empty finite constraint. Furthermore, let $d_z = \deg(C_{\{x,y\}}, \{z\})$, for $z \in \{x, y\}$. Finally, let $P_l(\cdot)$ be the function defined in Figure 6, then $P_l(C_{\{x,y\}})$ is a linear partition of $C_{\{x,y\}}$. Furthermore, $|P_l(C_{\{x,y\}})| \leq \min(d_x, d_y)$.*

Proof. Let $C_1 = C_{\{x,y\}}$. To prove that the proposition is correct it has to be demonstrated that $P_l(C_1)$ terminates, that $P_l(C_1)$ is a partition of C_1, that the members of $P_l(C_1)$ are linear, and that the cardinality of $P_l(C_1)$ does not exceed $\min(d_x, d_y)$.

termination Assume that $P_l(C_1)$ does not terminate. By assumption $|C_1|$ is finite. It follows from the non-termination of $P_l(C_1)$ and its termination criterion that $C_i \supset \emptyset$ for $i \in \mathbb{N} \setminus \{0\}$. This together with the definition of B_i allows us to infer that $\emptyset \subset B_i \subseteq C_i$ must hold. Therefore, $C_{i+1} = C_i \setminus B_i \subset C_i$ and it follows that the sequence

$$C_1 \supset C_2 \supset C_3 \supset \cdots$$

is infinite. This contradicts the premise that $|C_1|$ is finite.

partition property Let $C_1 = C_{\{x,y\}}$ and let $d = |P_l(C_1)|$. To prove that $P_l(C_1)$ is a partition of C_1 we must prove that $C_1 = \cup_{c \in P_l(C_1)} c$ and that $(\forall C_S, C_T \in P_l(C_1))(C_S \neq C_T \iff \emptyset = C_S \cap C_T)$.
First note that $P_l(C_i) = \cup_{i=1}^{d} \{B_i\}$. Next note that $C_{i+1} \cup B_i = C_i$, for $i = d, d-1, \ldots, 1$. Clearly, $P_l(C_1)$ is a cover of C_1. To see why the members of $P_l(C_1)$ are pairwise disjoint, observe that $B_j \subseteq C_{i+1} = C_i \setminus B_i$, for $1 \leq i < j \leq d$.

linearity property By Lemma 10, B_i is linear, for $1 \leq i \leq d$.

cardinality property Use Lemma 11 and induction on $\min(d_x, d_y)$.

Definition 13 (Generalised Branching Factor). The *generalised branching factor* of a linear partition of a constraint is given by the cardinality of that partition.

Note that maximal partitions of unary constraints are linear. Therefore, the notions of generalised branching factor and that of the branching factor coincide for maximal partitions of unary constraints.

The application of linear partitions will become apparent in the next example. Before we go on to that example, it should be pointed out that the minimum of the degrees of the variables that are involved in an arc-consistent binary constraint, cannot exceed the minimum of their domain sizes. This is formulated as the following proposition.

Proposition 14. *Let $C_{\{x,y\}}$ be a non-empty constraint, let $d_z = \deg(C_{\{x,y\}}, \{z\})$, and let $D(z) = \left\{ \mathrm{proj}_z(t) : t \in C_{\{x,y\}} \right\}$, for $z \in \{x, y\}$, then*

$$\min(d_x, d_y) \leq \min(|D(x)|, |D(y)|)\,.$$

Proof. $|D(y)| \geq d_x$ and $|D(x)| \geq d_y$.

In the previous section we saw that backtracking uses linear partitions of unary constraints to enumerate the members of the domain of the current variable. We have also seen that linear binary constraints can be used to amalgamate two variables. We have argued that this may be viewed as variable elimination (modulo renaming). By combining linear constraints and amalgamation, we can obtain lower (generalised) branching factors.

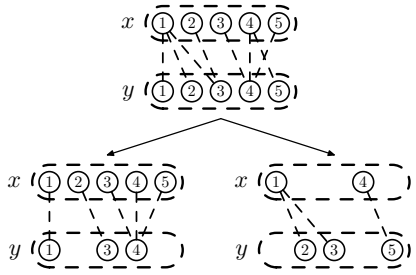

Fig. 7. Linear partition.

Example 15 (Generalised Backtracking). Consider the constraint $C_{\{x,y\}}$ whose micro-structure is depicted at the top of Figure 7. The constraint is cubic in x and in y. The two constraints whose micro-structures are depicted at the bottom of Figure 7 form the partition $\pi = \{\mathcal{C}_1, \mathcal{C}_2\}$ of $C_{\{x,y\}}$, where

$$\mathcal{C}_1 = \{(1,1),(2,3),(3,4),(4,4),(5,4)\},$$
$$\mathcal{C}_2 = \{(1,2),(1,3),(4,5)\}.$$

$\mathcal{C}_1$ is linear in y, whereas $\mathcal{C}_2$ is linear in x. The partition was computed using $P_l(\cdot)$ by always selecting the lexicographically smallest z-layer to compute the sets B_i.

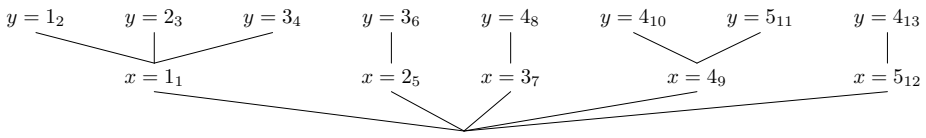

Fig. 8. In-order search tree for $x \prec y$. Branching factor is 5. #Visited nodes is 13.

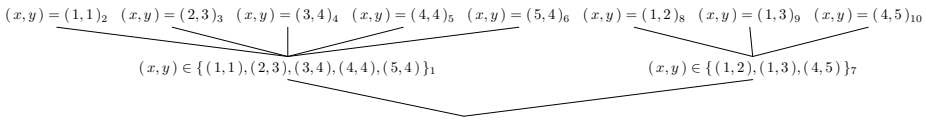

Fig. 9. Generalised search tree. Generalised branching factor is 2. #Visited nodes is 10.

Note that the generalised branching factor of π (the cardinality of π) is 2. This is strictly less than the x-degree and the y-degree of the original constraint. This demonstrates that the inequality in Proposition 12 may be strict.

The in-order search tree for the chronological backtracking algorithm for the variable order $x \prec y$ is depicted in Figure 8. The subscripts of the nodes and

leaves of the trees represent the visiting order. The tree has 13 visited nodes and a branching factor of 5 at the root. The number of leaves of the tree is equal to the cardinality of the constraint $C_{\{x,y\}}$. The search tree for $y \prec x$ has similar properties at that for $x \prec y$.

The *generalised search tree* corresponding to the partition π is depicted in Figure 9. The nodes and leaves of this tree are visited by a *generalised backtracking algorithm*. At the root of the tree there are two branches—one for each member of π. Each of the members of π is linear. As argued before, linear binary partitions of arc-consistent constraints correspond to the domain of a variable. As also argued before, a linear arc-consistent binary constraint can be used to eliminate a variable (modulo renaming) by amalgamating the variables that are involved.

1. The number of leaves of the generalised search tree is equal to the number of leaves of the in-order search trees. There is one leaf for each member of the constraint.
2. The linearity of π ensures that the maximum domain size does not increase.
3. The generalised branching factor at the root does not exceed the minimum domain size (the branching factors of the in-order search trees) of x and y and is usually smaller.
4. The linearity of π ensures that the height of the generalised search tree does not exceed the number of variables.
5. The number of visited nodes of the generalised backtracking tree is less than the number of nodes of each of the chronological backtracking trees. This is a consequence of 1, 2, and 3, and the fact that (for this example) the generalised branching factor is strictly less than the minimum domain size.

The most important result of the generalised backtracking approach is that it has decreased the generalised branching factor. Generalised backtracking works because the degrees of constraints determine the generalised branching factor. If the degree of a variable x in a binary constraint is d_x then x can be eliminated (modulo renaming) at the cost of a branching factor of d_x or less. We already observed that d_x never exceeds the cardinality of the domain of x and may be less than it. If d_x is less than the domain size of x then a smaller branching factor can be achieved than with the traditional backtracking approach. Since the domain sizes do not increase and since the height of the search tree remains the same this results in strictly fewer visited nodes.

Figures 7–9 suggest that every leaf in the tree (read the representatives of the members of $C_{\{x,y\}}$) can be visited. This is not true in general. There are at least two reasons. The first reason is that in general there may be more variables in a problem and the variables which will be the current variable at depth two from the root of the tree may be different. The second reason is that branches may become dead-ends as a result of the use of constraint propagation techniques.

6 Experimental Results

This section briefly discusses results from the application of a toy implementation in `Haskell` [12] of the generalised backtracking algorithm and discusses possible improvements on the implementation. It is still an item on the to-do-list to implement an efficient version of the algorithm.

The algorithm repeatedly selects a binary constraint. The constraint is always such that the domains of the variables that are involved are as small as possible. If the constraint is linear, it is used to eliminate a variable. Otherwise, if the constraint is universal, the constraint is removed. Otherwise the algorithm computes a linear partition of the constraint and uses it for branching. In addition, the algorithm maintains arc-consistency during search.

For RLFAP 3 and RLFAP 4 some large (but relatively easy) problems [2], it was observed that for deciding the satisfiability of these problems the generalised branching factor was almost always significantly smaller than the smallest domain size of any of the variables in the problem. It is recalled that the smallest domain size is a lower bound on the branching factor for chronological backtracking. For RLFAP 3, for example, the smallest domain size would almost always be between 20 and 40 and the ratio between the generalised branching factor and the smallest domain size would be between 0.6 and 0.9. Both problems could be solved without backtracking.

Constraints were implemented as black boxes (function calls) because an extensional representation would have been impossible due to the size of the input CSPs. The black box representation of constraints resulted in a lot of overhead for the computation of partitions. As mentioned before, it is our intention to look at more efficient implementation techniques.

The fact that the problems could be solved is not world shocking news; Any backtracker that maintains arc-consistency can do this almost effortlessly. However, the fact that the generalised branching factor b_g significantly improved upon the ordinary branching factor b_o may be significant. For example, a good heuristic for selecting a good constraint for partitioning need not be expensive. To partition a constraint $C_{\{x,y\}}$ is in $\mathcal{O}(d_x d_y)$, where d_x and d_y are the sizes of the domains of x and y. Furthermore, it seems that the variable elimination step which is carried out as part of the algorithm can be incorporated in the maintenance of arc-consistency without too much overhead (however, this still remains to be proved). Therefore, it seems that the "overhead" of partitioning is equivalent to about 1 times the work to make the remaining problem arc-consistent. Most of the time in search is spent on deciding that there are no solutions, i.e. *all* branches have to be searched and this requires at least b times the work to make the remaining problem arc-consistent, where b is the number of branches. If this "analysis" is correct then the savings of the generalised backtracking algorithm at the current level in the search tree are given by $b_o + 1 - b_g$ times the work of making the remaining problem arc-consistent.

In summary, much work remains to be done both in terms of implementation and experimental evaluation. However, the results from a few tests seem promising.

7 Summary

In this paper, the new notion of the degree of a constraint has been presented. This notion allowed us to reason about branching factors in search and allowed us to generalise the branching strategy of chronological backtracking.

The notion of the degree of a set of variables in a constraint led to the notion of a linear constraint and it has been shown how linear constraints can be used for the simplification of CSPs by amalgamating the variables involved in a linear binary constraint. This amalgamation operation corresponds to a variable elimination (modulo renaming). Arguments have been presented that for binary CSPs the average costs for the detection of linear constraints is low if arc-consistency is maintained.

It has been shown that the essence of chronological backtracking is that it uses linear partitions of unary constraints (the domains of the variables) to decompose a CSP into a set of CSPs whose solutions are disjoint and the union of whose solutions is equal to the solutions of the original CSP. The cardinality of a linear partition is called the generalised branching factor of that partition. The minimal generalised branching factor for a chronological backtracking algorithm which maintains arc-consistency is equal to the minimum domain size.

A generalisation of the chronological backtracking algorithm has been presented. This algorithm, called generalised backtracking, is not restricted to the use of linear partitions of unary constraints to decompose CSPs but can use any kind of constraint for this purpose. A function $P_l(\cdot)$ has been presented to compute linear partitions of binary constraints. The cardinalities of these partitions are small. If $C_{\{x,y\}}$ is a constraint such that the size of the domain of x or y is equal to the minimum domain size then the generalised branching factor of $P_l(C_{\{x,y\}})$ is never larger then the minimum domain size but may be smaller.

A few results have been presented of applications of a toy implementation of the generalised backtracking algorithm. The results are promising in the sense that they demonstrated that significant reductions of the generalised branching factor can be obtained. With proper adjustments, it may be possible that the algorithm becomes an improvement on the standard backtracking algorithm in the sense that it will also save consistency-checks. Suggestions have been presented on how to properly implement the algorithm. However, future research has to demonstrate whether generalised backtracking can be implemented efficiently and proper experiments have to be set up to compare chronological and generalised backtracking. Of course, these experiments should be complemented by a theoretical investigation.

References

1. W. Boege, R. Gebauer, and H. Kredel. Some examples for solving systems of algebraic equations by calculating Groebner bases. *Journal of Symbolic Computation*, 2(1):83–98, 1986.
2. B. Cabon, S. De Givry, L. Lobjois, T. Schiex, and J.P. Warners. Radio link frequency assignment. *Journal of Constraints*, 4:79–89, 1999.

3. S.R. Czapor. Solving algebraic equations: Combining Buchberger's algorithm with multivariate factorization. *Journal of Symbolic Computation*, 7(1):49–54, 1989.
4. P. David. Using pivot consistency to decompose and solve functional CSPs. *Journal of Artificial Intelligence Research*, 2:447–474, May 1995. AI Access Foundation and Morgan Kaufmann Publishers.
5. R. Dechter and D. Frost. Backtracking algorithms for constraint satisfaction problems. Technical report, University of California, Irvine, 1999.
6. J. Gaschnig. Experimental case studies of backtrack vs. Waltz-type vs. new algorithms for satisficing assignment problems. In *Proceeding of the* 2^{nd} *Biennial Conference, Canadian Society for the Computational Studies of Intelligence*, pages 268–277, 1978.
7. M.L. Ginsberg. *Essentials of Artificial Intelligence*. Morgan Kaufmann Publishers, San Mateo, California, 1993.
8. M.L. Ginsberg and D.A. McAllester. GSAT and dynamic backtracking. In A. Borning, editor, *Principles and Practice of Constraint Programming*, number 874 in Lecture Notes in Computer Science, pages 243–265. Springer-Verlag, Berlin/Heidelberg, 1994.
9. S.W. Golomb and L.D. Baumert. Backtrack programming. *Journal of the ACM*, 12(4):516–524, 1965.
10. H.-G. Gräbe. On factorized Gröbner bases. Technical Report 6, Institut für Informatik, Universität Leipzig, Germany, 1994.
11. R.M. Haralick and G.L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3):263–313, 1980.
12. P. Hudak, J. Peterson, and J.H. Fasel. A gentle introduction to haskell, version 1.4, 1997.
13. G. Kondrak and P. van Beek. A theoretical evaluation of selected backtracking algorithms. In C.S. Mellish, editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 541–547, 1995.
14. G. Kondrak and P. van Beek. A theoretical evaluation of selected backtracking methods. *Artificial Intelligence*, 89:365–387, 1997.
15. H. Melenk. Solving polynomial equation systems by Groebner methods. CWI Quarterly 3, 1990.
16. H. Melenk. Algebraic soltion of nonlinear equation systems in REDUCE. Technical report, Conrad-Zuse-Zentrum für Informationstechnik, Berlin, Germany, 1993.
17. B.E. Nadel. Constraint satisfaction algorithms. *Computational Intelligence*, 5(4):188–224, 1989.
18. M. Pesch. Factorizing Gröbner bases, 1996.
19. P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.
20. D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In A.G. Cohn, editor, *Proceedings of the* 11^{th} *European Conference on Artificial Intelligence*, pages 125–129. John Wiley and Sons, 1994.
21. E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.
22. M.R.C. van Dongen. *Constraints, Varieties, and Algorithms*. PhD thesis, Department of Computer Science, University College Cork, Ireland, 2002.
23. Pascal Van Hentenryck, Yves Deville, and Choh-Man Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57(2–3):291–321, 1992.