Journal of Functional Programming 1 (2): 213–227, April 1991

Domain generating functions for solving constraint satisfaction problems

FRANÇOIS MAJOR, GUY LAPALME and ROBERT CEDERGREN*

*Département d'informatique et de Recherche Opérationnelle, * Département de biochimie, Université de Montréal, Canada*

Abstract

This paper presents an application of functional programming: searching a domain for elements which satisfy certain constraints. We give a very general formulation of the problem and describe 'generate and test', 'backtracking' and 'forward checking' algorithms. We then introduce the concept of domain generating functions to capture a common optimization during the search process: using partial solutions to reduce the size of the search space. We compare the efficiency of the original algorithms and those using domain generating functions first with the 'classical' n-queens example, and then with a problem having larger domains to search which was inspired by an application in macromolecular structure determination. Using algorithms coded in Miranda, Haskell and Common Lisp, we show that a high order (lazy) functional language is a useful and efficient tool for prototyping search methods in large complex domains.

Capsule review

One of the powers of higher-order functional programming is the ease with which one can abstract common behaviors. This paper provides an example of abstraction principles applied to a fairly general class of problems, which the authors call *constraint satisfaction problems*. Simply put, such problems involve a search for 'assignments' to a set of variables so as to satisfy a given set of constraints. The authors present three general *search strategies* for this problem, but in addition introduce the idea of a *domain generating function* which utilizes problem-specific information to narrow the search space dynamically. The solutions are presented in a parameterized way using higher-order functions and polymorphic data structures, thus allowing the same strategies to be used in many different specific applications. After demonstrating the ideas using the classical n-queens problem, the authors tackle a more realistic problem from the field of molecular biology: the *protein folding problem*. Timing results are given for specific implementations of Miranda, Haskell, and Common Lisp, and observations are made concerning the utility of lazy evaluation.

1 Introduction

This paper presents a functional approach to solving combinatorial problems searching a domain for elements which satisfy certain constraints, i.e., a Constraint Satisfaction Problem (CSP). When such problems can be modeled using continuous

real variables, they can be dealt with using classical operations research techniques such as the simplex method. However, when discrete solutions are sought, we have to resort to enumeration techniques such as backtracking (Knuth, 1975) or forward checking (Haralick, 1980). This problem has also attracted much interest in the logic programming community (Jaffar and Michaylov, 1987; Hentenryck, 1989), where the emphasis is put on solving problems in restricted domains.

This paper gives a very general formulation of this problem. Section 2 describes the 'generate and test', 'backtracking' and 'forward checking' algorithms, it illustrates their use and it compares their efficiency by solving the classical n-queens problem. Section 3 then introduces the notion of the domain generating function which uses the information gained in a partial solution to reduce the size of domains to be searched; the technique is also used in solving the n-queens problem. Section 4 then applies this approach in a more complicated setting using a problem inspired by a real application encountered in the field of molecular biology, which first prompted this work. Finally, we compare these approaches and show that domain generating functions are very powerful tools for solving constraint satisfaction problems. We also argue that lazy evaluation is useful in this case; the forward checking methods can be implemented in a straightforward way (even with domain reducing functions) without losing efficiency.

2 Solving a constraint satisfaction problem

2.1 Definitions

Given X a set of variables $\{x_1, x_2 \ldots x_n\}$, D a set of domains of values for each variable $\{d_1, d_2 \ldots d_n\}$ and C a set of binary constraints $\{c_{p,q} | p \in \{1 \ldots n\}, q \in \{1 \ldots p-1\}\}$. Constraint $c_{p,q}$ indicates the values of domain d_p that are compatible with d_q. These subsets are not usually given extensionally, but are defined as equations, inequalities or predicates. A variable assignment $\sigma = \{v_1, v_2 \ldots v_n\}$ is the assignment of value v_i to x_i, $v_i \in d_i$, $i \in 1 \ldots n$. Solving a CSP involves finding all value assignments such that all constraints $c_{p,q}$ are satisfied. We present here the case of binary constraints, but the functions we describe can also handle n-ary constraints.

A classical problem for the comparison of CSP solving methods is the n-queens problem, which consists of finding a way of placing n queens on a $n \times n$ chess board such that no queen can attack another. The position for a queen is given by its row and column number. [This is not the most efficient way of representing a solution to this problem, but it is very straightforward and it serves only as an illustration of our technique which is expanded upon later.] So here X is $\{x_1 \ldots x_n\}$ and D is $\{d_1 \ldots d_n\}$ where each d_i is $\{(1 \ldots n, 1 \ldots n)\}$. Two queens attack each other if they are on the same line, column or on the same diagonal

$$\{c_{(i,j),(i',j')} \equiv i \neq i' \wedge j \neq j' \wedge |i-i'| \neq |j-j'|,\ i \in \{1 \ldots n\},\ j \in \{1 \ldots i-1\}\}$$

We now formulate the solution to a CSP using three standard approaches: generate and test; backtracking; and forward checking. This shows the close relations between these approaches when a functional language is used for implementation. We use Miranda (Turner, 1985) to express our algorithms (we only use the subset of Miranda

https://doi.org/10.1017/S0956796800020050 Published online by Cambridge University Press

used by Bird and Wadler (1988)), but the same technique could easily be applied using other lazy functional languages such as LML (Augustsson and Johnson, 1989) or Haskell (Hudak and Wadler, 1990). [*Miranda* is a registered trademark of Research Software Ltd.] In fact, we have translated our algorithms into Haskell and tested them using the Haskell-B compiler for Chalmers University, Gothenburg, Sweden. A Common Lisp version of the functions has also been tested using Allegro Common Lisp for Franz Inc. We see that lazy evaluation is basically not essential to this method, but in some cases it makes it possible to solve problems that would require too much memory using an applicative order of evaluation.

2.2 Generate and test

A constraint-satisfaction problem can be solved using a function that generates the exhaustive set of assignments, and another that checks the constraints for each assignment. Given the following types

$$\begin{aligned} &domain* &&= = [*] \\ &solution* &&= = [*] \\ &generator* &&= = [domain*] \rightarrow [solution*] \\ &test* &&= = solution* \rightarrow bool \\ &constraint* &&= = solution* \rightarrow * \rightarrow bool \end{aligned}$$

where domains and solutions are lists of values; a generator is a function that, starting from a list of domains, gives a list of potential solutions; a test verifies if a solution is acceptable or not; and a constraint verifies if a new value is compatible with a list of previously assigned variables (i.e., a partial solution). We can define the 'generate and test' paradigm as

$$\begin{aligned} &generate :: generator* \\ &generate\ [\,] = [[\,]] \\ &generate\ (domain : domains) \\ &\quad = concat\ [map(value :)(generate\ domains) \mid value \leftarrow domain] \end{aligned}$$

$$\begin{aligned} &generate\_test :: generator* \leftarrow test* \rightarrow [domain*] \rightarrow [solution*] \\ &generate\_test\ gen\ test\ domains = filter\ test\ (gen\ domains) \end{aligned}$$

For the n-queens problem, two queens attack each other if they are on the same column or on the same diagonal (we generate potential solutions that ensure that $i > j'$)

$$safe\ (i, j)\ (i', j') = j \neq j' \ \&\ (i - i') \neq abs\ (j - j')$$

For checking if all elements of a solution are consistent with respect to each other

$$\begin{aligned} &tst\ [x, y] = safe\ x\ y \\ &tst\ (x : xs) = and\ [safe\ x\ x' \mid x' \leftarrow xs]\ \&\ tst\ xs \end{aligned}$$

Solving the n-queens problem is only a matter of calling the *generate_test* function with the appropriate parameters

$$solve\ n = generate\_test\ generate\ tst\ [[(i, j) \mid j \leftarrow [1..n]] \mid i \leftarrow [n, n-1..1]]$$

https://doi.org/10.1017/S0956796800020050 Published online by Cambridge University Press

2.3 Backtracking

The previous method is easily implemented but is inefficient because the algorithm generates many complete assignments which are rejected by the test filter. One way to help alleviate this problem is to verify the constraints between the 'current' variable and the previously assigned values. This approach not only reduces the number of assignments but also insures that a complete assignment (a compatible value is given for each variable) is a solution to the problem.

This approach can be implemented using 'backtracking' which constructs partial assignments $\sigma_k\{v_1, v_2 \ldots v_k\}$ such that $\sigma_{k+1} \Rightarrow \sigma_k, 1 \leqslant k < n$; this means that a partial assignment which does not satisfy the constraints between the first k variables cannot be part of a full solution.

The algorithm is implemented by building partial solutions incrementally such that at iteration k we only generate values in the domain d_k that satisfy the constraints $c_{k,1} \ldots c_{k,k-1}$. A partial solution is represented as two parameters $\{v_1, v_2 \ldots v_k\}$ and $\{d_{k+1}, d_{k+2} \ldots d_n\}$, $1 \leqslant k \leqslant n$ where the first is the set of values assigned to the first k variables, and the second is the set of domains for the yet unassigned variables. At iteration k, only values of d_k satisfying $c_{k,1} \ldots c_{k,k-1}$ (in the case of binary constraints) will be tentatively assigned to x_k. This process can be seen as narrowing the domain d_k. *cst* can deal with n-ary constraints, and not only binary ones. So, constraints of any arity are implemented 'transparently'. In this text, we simplify the formulas using binary constraints of the form $c_{i,j}$, but they could be extended to functions of any arity.

```
narrow :: solution * → constraint * → domain * → domain *
narrow [] cst dom = dom
narrow sol cst dom = filter (cst sol) dom

backtrack :: solution * → [domain *] → constraint * → [solution *]
backtrack sol [] cst = [sol]
backtrack sol (dom : doms) cst
   = concat [backtrack (sol ++ [value]) doms cst | value ← narrow sol cst dom]
```

This implementation of backtracking uses the 'list of successes' technique advocated by Wadler (1985): computing the list of all solutions is conceptually simpler than having to go back to a previous point in case of a failure. Thanks to the lazy evaluation, this approach is not too costly in terms of space because only the elements of the list of results needed for finding a solution are computed.

For the n-queens problem, the constraint function is applied when a new variable is assigned. It receives a partial solution as well as the new value. We then verify that the new queen will not be attacked by a previously assigned one

```
cst s x = and (map (safe x) s)
```

Now solving the problem for n queens is only a matter of calling the *backtrack* function with the appropriate parameters

```
solve n = backtrack [] [[(i, j) | j ← [1..n]] | i ← [1..n]] cst
```

https://doi.org/10.1017/S0956796800020050 Published online by Cambridge University Press

Table 1 gives the results with these functions. For each number of queens, the table gives the execution time (in CPU seconds computed by the Miranda (version 2.015) interpreter on a Sun Sparc Station 1), and the number of reductions as given by the Miranda interpreter. We also give the execution times for the Haskell and Lisp versions. We see that the Haskell compiler generates code that executes four times faster than the Miranda interpreter; however, the Haskell compiler we currently use is very slow and error messages are cryptic, so we were grateful that we had a running Miranda version before using Haskell. We also did a very straightforward translation into Common Lisp, and were surprised at the speed of the resulting code; it appears that this high-level approach to the specification of search algorithms is also useful in a functional language that uses an applicative order.

Table 1. *Results for the n-queens problem.*

		Backtracking		Forward checking	
n	*Language*	CPU time	# reductions	CPU time	# reductions
6	*Miranda*	1·03	68,672	0·62	41,192
	Haskell	0·20		0·10	
	Lisp	0·05		0·07	
8	*Miranda*	23·92	1,498,888	9·45	594,539
	Haskell	5·80		2·00	
	Lisp	1·17		0·87	
9	*Miranda*	120·90	7,627,205	36·87	2,470,079
	Haskell	33·20		9·80	
	Lisp	5·38		3·20	

We also carried out an experiment in Miranda where this high-level specification of search control was compared with the *ad hoc* solution given by Bird and Wadler (1988) which uses backtracking. Our general version runs a little faster than the *ad hoc* solution. This is a very convincing example that higher-order functional style does not necessarily imply an execution time cost. The same kind of results were observed in the solution of the 'instant insanity' cube problem, also described by Bird and Wadler (1988).

2.4 Forward checking

Although backtracking is more efficient than 'generate and test', it often has to 'rediscover' that one value is incompatible with another. This is because after backtracking from an impossible solution, the work done to discover that an assignment is not good is deleted. The fact that a value of d_i is incompatible with one value of d_k,$k > i$ has to be 'rediscovered' at each narrowing of d_k. Even worse, if no value of d_k is compatible with a value of d_i, a whole subtree search has to be expanded uselessly until level k. For these reasons, forward checking algorithms are often used instead of backtracking (Haralick, 1980; Hentenryck, 1989).

https://doi.org/10.1017/S0956796800020050 Published online by Cambridge University Press

In both methods, the solution is generated by building partial assignments $\sigma_k\{v_1, v_2 \ldots v_k\}$ such that $\sigma_{k+1} \Rightarrow \sigma_k, 1 \leqslant k < n$. Forward checking incrementally builds valid solutions such that at iteration k only the values satisfying $c_{k,1} \ldots c_{k,k-1}$ will be assigned to x_k, but also the domains $\{d_{k+1} \ldots d_n\}$ will be narrowed by checking the constraints $\{c_{p,q} \mid p \in \{k+1 \ldots n\}, q \in \{1 \ldots k-1\}\}$. This *a priori* narrowing of the dimension of these domains reduces the work to be done in subsequent steps, and can thus more quickly determine whether a partial solution is a dead-end or not. The number of variable assignments and constraint checkings is reduced, resulting in a method often more efficient than backtracking (Haralick, 1980). This paradigm is expressed as follows

```
forward_check :: solution * → [domain *] → constraint * → [solution *]
forward_check sol [] cst = [sol]
forward_check sol domains cst
   = [],                                          member doms []
   = concat [forward_check (sol++[value]) doms cst | value ← domk], otherwise
   where (domk : doms) = map (narrow sol cst) domains
```

For the forward-checking, domains are narrowed at each iteration depending on the last variable assignment, so constraints do not have to be reverified when a new variable is assigned. In the n-queens problem, we only need to check the last variable

```
cst sol x = safe x (last sol)
```

When solving, we only change the name of the algorithm

```
solve n = forward_check [] [[(i, j) | j ← [1..n]] | i ← [1..n]] cst
```

Table 1 gives the results for the forward-checking algorithm where we see that forward-checking is a great improvement over backtracking, execution times being typically three times faster. As forward checking can be implemented quite directly in a higher order functional language and is not much more complicated than backtracking, it should always be considered in solving a CSP. The same remarks made in the previous section also apply for the Haskell and Lisp versions.

3 Domain generating functions

In order to solve a specific problem and to improve the performance of an algorithm, we can often use additional information specific to the particular problem. Since this type of 'help' is problem-dependent, it should not be integrated in the general solving methods discussed previously but as we show in this section, it can be parameterized using functions that here we call either help or domain generating functions.

In the algorithms for CSP solving, we introduce a parameter which is a function that allows the generation of values for a given variable. The parameters of the function are the partial solution constructed up until then and the current domain of the variable. So, the idea is as follows: before reducing a domain using the constraints (as we do for backtracking and forward-checking), we use the additional information to generate a (smaller) subdomain of possible compatible values with the variables

https://doi.org/10.1017/S0956796800020050 Published online by Cambridge University Press

already assigned; the rest of the algorithm will then do less work because the domain of 'interesting' values can be greatly reduced.

More formally, for domain d_k we define d_k^{min} as the set of values that will be used in at least one solution of the problem. So, one way to evaluate the help given by the function is to compare the generated domain d_k^g with d_k^{min}. Finding a function that generates a domain such that $d_k^{min} \subseteq d_k^g \subset d_k$ should help the algorithm.

3.1 Backtracking with help

To implement this paradigm in backtracking, we use the help function to generate the domain $d_k^g \subseteq d_k$ before verifying the constraints. Help is defined as a function that transforms the domain given a partial solution

```
help * = = solution * → domain * → domain *

narrow' :: solution * → constraint * → help * → domain * → domain *
narrow' [] cst hlp dom = dom
narrow' sol cst hlp dom = filter (cst sol) (hlp sol dom)

backtrack' :: solution * → [domain *] → constraint * → help * → [solution *]
backtrack' sol [] cst hlp = [sol]
backtrack' sol (dom:doms) cst hlp
  = concat[backtrack'(sol ++ [value])doms cst hlp | value ← narrow' sol cst hlp dom]
```

Now we define a help function for the n-queens problem that removes from the domain the positions that are on the same column as another queen. Only the diagonals have to be verified because *gen* guarantees that this attack will never occur, so *safe* can be simplified. This gives the following backtracking algorithm with a help function

```
safe (i, j) (i', j') = i − i' ≠ abs(j − j')
gen s d = d −− [(fst (hd d), snd x) | x ← s]
solve n = backtrack' [] [[(i, j) | j ← [1..n]] | i ← [1..n]] cst gen
```

Table 2 gives the results for these 'better informed' algorithms. Comparing these results with the ones for backtracking in Table 1, we can see that the domain generating functions are not really useful in this example, execution times and the number of reductions being roughly the same.

3.2 Forward-checking with help

The same idea can be implemented in forward-checking where the domains $d_k \ldots d_n$ have to be narrowed

```
forward_check' :: solution * → [domain *] → constraint * → help * → [solution *]
forward_check' sol [] cst hlp = [sol]
forward_check' sol domains cst hlp
  = [],                                                    member doms []
```

https://doi.org/10.1017/S0956796800020050 Published online by Cambridge University Press

Table 2. *Results for the n-queens problem with help functions.*

		Backtracking		Forward checking	
n	*Language*	CPU time	# reductions	CPU time	# reductions
6	*Miranda*	1·08	68,822	0·78	52,675
	Haskell	0·30		0·20	
	Lisp	0·05		0·02	
8	*Miranda*	24·10	1,447,596	12·27	761,298
	Haskell	8·40		3·20	
	Lisp	1·27		1·15	
9	*Miranda*	122·17	7,180,211	54·12	3,240,573
	Haskell	46·80		15·00	
	Lisp	6·02		5·12	

$$= \textbf{concat}[\textbf{forward\_}\mathit{check'}\ (\mathit{sol}{+\!\!+}[\mathit{value}])\ \mathit{doms}\ \mathit{cst}\ \mathit{hlp}\mid \mathit{value} \leftarrow \mathit{domk}],$$

otherwise

where $(\mathit{domk}: \mathit{doms}) = \mathit{map}\ (\mathit{narrow'}\ \mathit{sol}\ \mathit{cst}\ \mathit{hlp})\ \mathit{domains}$

In the case of the *n*-queens problem, the help function only removes the last assigned value

$$\mathit{safe}\ (i, j)\ (i', j') = i - i' \neq \mathit{abs}(j - j')$$
$$\mathit{gen}\ s\ d = d - - [(\mathit{fst}(\mathit{hd}\ d), \mathit{snd}(\mathit{last}\ d))]$$
$$\mathit{solve}\ n = \mathit{forward\_check'}\ [\,]\ [[(i, j) \mid j \leftarrow [1..n]] \mid i \leftarrow [1..n]]\ \mathit{cst}\ \mathit{gen}$$

Table 2 gives the results for the forward-checking using help functions. However, the CPU time does not decrease as expected. This is mainly due to the poor efficiency of the list difference operator($- -$) used in the help function. In this case, the help functions for forward_*checking* imply more work than that which is saved in the reduction of domains. The goal of this section was only to illustrate the formulation of a domain generation function using the same classical problem. Domain generating functions will prove useful in a problem where the domains are much larger than this simple example.

3.3 Properties of 'helped' algorithms

Given d_k the domain of values for variable k at iteration k, $d_k^{\min}$ the domain of values for variable k once d_k is narrowed by the constraints, and d_k^g the domain generated by the help function, we have the following properties

1. If $d_k^g = d_k^{\min}$ then no constraint needs to be checked; this property is not very 'practical', though, because it would imply that constraints were not useful in the formulation of the problem.
2. If the help function is such that $d_k^{\min} \subseteq d_k^g \subseteq d_k$, then *backtrack'* and *forward check'* keep the induction property: $\sigma_{k+1} \Rightarrow \sigma_k$ and guarantee that all solutions will be generated.

https://doi.org/10.1017/S0956796800020050 Published online by Cambridge University Press

3. The efficiency ratio between a 'helped' algorithm and the original one is given by the cost of applying, at each domain narrowing, the help function, and the cost for making d_k^g equal to $d_k^{\min}$ compared with the cost of narrowing d_k to get $d_k^{\min}$.

The evaluation of the efficiency of algorithms derived from backtracking can be quite complex (Knuth, 1975). The usual criterion is the number of constraint checks. To simplify, consider the $n(n-1)/2$ binary constraints between n variables: if each domain has dimension m, the maximal number of constraint verifications to be made by the generate and test algorithm is $(n(n-1)/2)m^n$.

Haralick (1980) describes a probabilistic analysis for backtracking and forward-checking given that the dimension of the domains is constant (m) and the probability p that a constraint between two variables is satisfied is independent of the n variables and of the preceding computations.

The efficiency ratio between a helped and a conventional algorithm is given by the cost of applying the help function plus the cost for making d_k^g equal to $d_k^{\min}$ against the cost for removing the $|d_k^g| - |d_k^{\min}|$ supplementary values of d_k. So, in some cases, the use of the help function in forward-checking can be more costly than simply narrowing d_k with the constraints. This explains that for the n-queens problem, help functions did not reduce execution times compared to the 'original' versions.

4 Blocks topology generation

This work considers research in the field of molecular biology. A fundamental problem in molecular biology is to predict the native structure of a protein from its amino acid sequence. This problem is often called 'the protein folding problem' (Kolata, 1986; Fasman, 1989), and it can be defined as a CSP. It consists of finding the spatial arrangement of all atoms consistent with known constraints. For this more realistic problem, 'domain generating functions' appear more useful. (Details of the model and the implementation can be found in Major *et al.* (1990).)

This application can be abstracted somewhat by using a simplified model consisting of blocks in 3D space which have similar properties to the ones of the 'real' protein model: the 3D-blocks world. In this representation, unit-size blocks are used to represent groups of atoms and the space is represented by a simple 3D lattice (Major *et al.*, 1988) whose coordinates vary between values given for *xmin*, *ymin*, *zmin*, *xmax*, *ymax* and *zmax*

$$
\begin{aligned}
&block == (name, position)\\
&name ::= A \mid B \mid C \mid D\\
&position == [num]\\
&block == (name, position)\\
&block\_name\,(n, p) = n\\
&block\_pos\,(n, p) = p
\end{aligned}
$$

We first define a function to build blocks according to a domain of positions and a block name

$$
\begin{aligned}
&block\_make :: [position] \to name \to [block]\\
&block\_make\ dom\ n = [(n, p) \mid p \leftarrow dom]
\end{aligned}
$$

https://doi.org/10.1017/S0956796800020050 Published online by Cambridge University Press

The default block value domains, for a specific block, is the entire lattice

```
block_default :: name → [block]
block_default = block_make lattice
                where
                lattice = [[x, y, z] | x ← [xmin..xmax];
                                       y ← [ymin..ymax];
                                       z ← [zmin..zmax]]
```

Now, spatial relationships among the 3D-blocks can be introduced. The distance between two blocks is the square of the Euclidean distance. Two adjacent blocks are those that are separated by adistance of 1. A collision occurs when two blocks have the same position

```
distance :: position → position → num
distance [x, y, z] [x', y', z'] = (x−x')^2+(y−y')^2+(z−z')^2

adjacent, collision :: position → position → bool
adjacent x y = distance x y = 1
collision x y = x=y
```

Where atomic distance constraints are used to generate protein topologies in the 'real' model, block distances are used in this simplified model. So, more precisely, searching for the topology of a set of blocks consists of finding their positions in 3D-space such that a set of distance constraints are satisfied.

4.1 Implementing the constraints

The goal of such a system is to reproduce all possible structures where only some topological constraints are known. In molecular biology, such constraints can be determined in the laboratory. Here, however, we use only examples of constraints in the 3D-blocks world. Suppose we wish to find the block topology of a complex composed of four single blocks: A, B, C and D. The constraints are: A is adjacent to B; B is adjacent to C; C is adjacent to D; and A is at distance 3 from D. These constraints have been chosen to resemble a set of constraints for the yeast transfer RNA molecule (Rich and RajBhandary, 1976).

The following function implements these constraints

```
cst :: block → block → bool
cst (n, p) (n', p')
  = distance p p' = 3, case = (A, D)
  = adjacent p p',     case = (A, B) \/ case = (B, C) \/ case = (C, D)
  = ~collision p p',   otherwise
    where
      case = (n, n')
```

In the case of backtracking, these constraints have to be satisfied for each block in the partial solution

```
check s x = and [cst y x | y ← s]
```

https://doi.org/10.1017/S0956796800020050 Published online by Cambridge University Press

We fix the first block at [0,0,0] so that situations which differ only by a translation of A are not uselessly calculated. We define the domains of values for the other blocks as the default domains

$$solve = backtrack\ [(A,[0,0,0])]\ [block\_default\ n \mid n \leftarrow [B,C,D]]\ check$$

In the case of forward-checking, we only have to check the last assigned value

$$check\ s\ x = cst\ (last\ s)\ x$$
$$solve = forward\_check\ [(A,[0,0,0])]\ [block\_default\ n \mid n \leftarrow [B,C,D]]\ check$$

We defined two sets of volume coordinates, one with $m = 343(7^3)$ (seven positions in each dimension) and the other with $m = 1{,}331(11^3)$ such that p, the probability of randomly placing a block adjacent to another are approximatively, and respectively, 0·017 and 0·0045; this is more than 30 and 130 times smaller than in the 8-queens problem where the probability of placing a queen on a chessboard randomly such that the queen is safe is approximately $p = 0{\cdot}65$. Table 3 shows that forward-checking is about six times faster than backtracking. Given the time required in the n-queens problems described in the previous section, it is a surprise that the execution times for Haskell are now slower than the Miranda version. The Lisp version is still much faster than Miranda.

Table 3. *Results for the 3D-blocks topology problem.*

		Block D with DGF		Block D without DGF	
Algorithm	Language	$P = 0{\cdot}017$	$P = 0{\cdot}0045$	$P = 0{\cdot}017$	$P = 0{\cdot}0045$
Backtracking	Miranda	20·48	81·55		
	Haskell	32·10	117·60		
	Lisp	2·05	8·23		
Forward-checking	Miranda	3·47	13·22		
	Haskell	6·90	26·60		
	Lisp	0·67	2·25		
Backtracking'	Miranda	1·18	1·18	15·37	57·88
	Haskell	2·80	2·80	24·40	94·50
	Lisp	0·13	0·25	1·15	6·15
Forward-checking'	Miranda	1·13	1·63	0·87	1·87
	Haskell	3·20	4·10	1·70	4·20
	Lisp	0·22	0·60	0·22	0·62

4.2 Using the domain generating functions

To generate the domains as needed, we define functions that generate the allowed positions of blocks so that they are adjacent or at a precise distance. The function *gen_dist* takes a distance and a position, and returns the domains of adjacent positions subject to the distance constraint. Then, *gen_dist* can be used to define *gen_adj*, a function that returns the adjacent position to its position argument

https://doi.org/10.1017/S0956796800020050 Published online by Cambridge University Press

```
gen_dist :: num → position → [position]
gen_dist d [x,y,z]
  = [[x,y,z] | dx ← [−d'..d']];
             x' ← [x+dx]; x' >= xmin; x' <= xmax; adx ← −[abs dx];
             dy ← [−d' + adx..d' − adx];
             y' ← [y+dy]; y' >= ymin; y' <= ymax; ady ← [abs dy];
             dz ← [−d' + adx + ady..d' − adx − ady];
             z' ← [z+dz]; z' >= zmin; z' >= zmax;
             (x−x')^2 + (y−y')^2 + (z−z')^2 = d']
  where
    d' = ceiling d

gen_adj = gen_dist 1
```

Now, the help function *gen* for backtracking can easily be written as follows

```
gen sol dom = block_make (gen_adj (block_pos blockA)) n, n = B
            = block_make (gen_adj (block_pos blockB)) n, n = C
            = block_make (gen_adj (block_pos blockC)) n, n = D
            where
              n = block_name (hd dom)
              blockA = get_block 1 sol
              blockB = get_block 2 sol
              blockC = get_block 3 sol
              get_block i = (!(i−1))
```

The *cst* and *check* functions are the same as before.

The *gen* function for *forward_check'* cannot be implemented as easily as *gen* for *backtrack'*. In *forward_check*, *check* verifies the values of a domain only with the last assigned value. The new values of the narrowed domain are compatible with the other assigned variables, since they were verified at each iteration when the variables were assigned. *gen* generates a domain in terms of all previously assigned variables. However in *forward_check'*, where a help function is a generation, not a limitation function, a problem occurs when *gen* is applied to generate a previously narrowed domain: the work previously carried out during constraint verification can be lost. If it is not possible to write such a function which considers all previously defined variables, two corrective measures can be implemented using either intersection of domains or additional constraint verifications.

A first method would consist of performing an intersection between the generated domain, d_k^g, and the previously narrowed domain d_k. Its disadvantage is that the efficiency of *gen* is dependent on the dimensions of domains. Moreover, an intersection could completely cancel the lazy evaluation.

The other method applies *cst* between the values of d_k^g and every affected variable which did not serve for the generation of the new domain as in backtracking. So, here we must pay the cost associated with the supplementary constraint verifications.

We choose the latter method, which has the further advantage that the cost of solving the problem is independent of m, the dimension of the initial domains. The

https://doi.org/10.1017/S0956796800020050 Published online by Cambridge University Press

following is the definition of the domain generating function using supplementary constraint checks

```
gen sol dom
  = block_make (gen_adj (block_pos blockA)) n, n = B
  = [x | x ← block_make (gen_adj (block_pos blockB)) n;
      cst blockA x],                          n = C & #sol > 1
  = [x | x ← block_make (gen_adj (block_pos blockC)) n;
      cst blockA x;
      cst blockB x],                          n = D & #sol > 2
  = dom,                                      otherwise
    where
      n = block_name (hd dom)
      blockA = get_block 1 sol
      blockB = get_block 2 sol
      blockC = get_block 3 sol
      get_block i = (!(i − 1))
```

Here, the tests on the length of *sol* are used to delay the evaluation of the help function until all previous variables are assigned in order to limit useless work. For the other approach, using domain intersection, we would only replace the *cst blockX x* expressions by one *member d x*.

Table 3 compares the CPU times, and it can be seen that the efficiency for solving topological constraints is improved by one order of magnitude using domain generating functions. It is clear that problems with a small p value can benefit from the use of domain generating functions, such as the case for protein structure prediction where an atom can be positioned anywhere in the Euclidean space. However, our knowledge of many regular motifs adopted by the atoms considerably reduces the search space. The numbers in Table 3 are given only as examples, and more experimentation would be necessary to produce more convincing statistics.

For example, it is not clear that 'help' forward-checking is always less efficient than 'help' backtracking. If a domain generation function is used for every variable of the problem, forward-checking with help is less efficient than backtracking with help. This is because forward-checking is uselessly filtering the domains of variables $k+1 \ldots n$ at iteration k. One might say that having lazy evaluation, we should not see any difference between both algorithms because the filtering of a domain should be delayed until required, until iteration $k-1$. However, the *member dom* [] expression that ensures the problem is still feasible forces the forward-checking to find at least one satisfying value in the default domain d_k at any iteration in $\{1 \ldots k\}$. The cost may be very low when no constraint has to be satisfied (in such a case the first domain's value is fine) or may cost more if a constraint occurs between variables i and j, $j-i > 1$. In any case, the work that has to be done to ensure consistency in forward-checking is paid and explains the difference in efficiency. For these reasons, the choice between backtracking and forward-checking with help can be difficult, and strongly dependent on the problem and constraints within the application.

In our examples, Lisp compiled code is much faster than both Miranda interpreted

https://doi.org/10.1017/S0956796800020050 Published online by Cambridge University Press

code and Haskell compiled code. This can surely be attributed to the lazy evaluation mechanism. We also did some experiments by increasing the domains of variation of variables; the largest values we could test before running out of memory in Lisp were between -9 and 9. On the other hand, the number of reductions for backtracking with help in Miranda (with a domain generating function for each variable) stayed the same. The other algorithms take longer to execute, but they always manage to terminate.

Of course, we can always simulate lazy evaluation in Lisp by writing a generator or by using the ones defined by Steele (1990), but then the algorithm is cluttered with explicit calls to an application-dependent generator. We are still convinced the lazy evaluation is useful in this case for prototyping, especially given that, in this kind of application, ease of change may be more important than raw speed.

5 Conclusion

The usefulness of domain generating functions for solving CSP problems is illustrated. Having functions to dynamically generate domains is very general, and can be adapted to many contexts. We have given a simple and a more complex example where 'symbolic constraints' can easily be specified. This approach could also be used to deal with 'numeric' constraints, and it is reminiscent of the 'column generation' technique used to solve large linear programming problems. An important benefit of a lazy functional language is the fact that quite often large areas of domains are never generated, because they are never needed. This approach, although powerful, requires some expertise in defining the right 'helps' for a given CSP. It would be interesting to develop a way to deduce these help functions directly from the expressions defining the constraints.

Acknowledgments

We would like to thank Patrice Boizumault who did a very careful reading and made many constructive comments about a previous version of the manuscript. We also thank the referees who gave valuable advice.

References

Augustsson, L. and Johnsson, T. 1989. The Chalmers Lazy-ML compiler. *The Computer Journal*, 32 (2): 127–41.

Bird, R. and Wadler, P. 1988. *Introduction to Functional Programming*. Prentice-Hall.

Fasman, G. D. 1989. Protein conformation prediction. *Trends Biol. Sci.*, 14: 295–99 (Jul.).

Haralick, R. M. 1980. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14: 263–313.

Hentenryck, P. V. 1989. *Constraint Satisfaction in Logic Programming*. MIT Press.

Hudak, P. and Wadler, P. (eds). 1990 *Report on the programming language Haskell, a non-strict purely functional language (Version 1.0)*. Technical Report YALEU/DCS/RR777, Yale University, Department of Computer Science. (Apr.)

Jaffar, J. and Michaylov, S. 1987. Methodology and implementation of a constraint logic programming system. In *Proceedings of the Fourth International Conference on Logic Programming*: 196–218. MIT Press.

Knuth, D. E. 1975. Estimating the efficiency of backtrack programs. *Mathematics of Computation*, 29 (1): 121–36.
Kolata, G. 1986. Trying to crack the second half of genetic code. *Science*, 233: 1037–39.
Major, F., Feldmann, R., Lapalme, G. and Cedergren, R. 1988. FUS: a system to simulate conformational changes in biological macromolecules. *CABIOS*, 4 (4): 445–51.
Major, F., Gautheret, D., Turcotte, M., Lapalme, G., Jolicoeur, L., Fillion, E. and Cedergren, R. J. The prediction of RNA 3-D structures by combining symbolic and numerical computation. (*Submitted for publication*, 1990).
Rich, A. and RajBhandary, U. L. 1976. Transfer RNA: molecular structure, sequence, and properties. *Annual Review of Biochemistry*, 45: 805–60.
Steele, G. 1990. *Common Lisp, the Language*. Digital Press.
Turner, D. A. 1985. Miranda—a non-strict functional language with polymorphic types. In P. Jouannaud (editor) *Conference on Functional Programming and Computer Architecture, volume* 201 *of Lecture Notes in Computer Science*, Springer-Verlag.
Wadler, P. 1985. How to replace failure by a list of successes. In P. Jouannaud (editor), *Conference on Functional Programming and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, Springer-Verlag.

François Major* and Guy Lapalme, Département d'informatique et de Recherche Operationnelle, Université de Montréal, CP 6128, Succ "A", Montréal, Quebec, Canada H3C 3J7.
(* Currently at: National Center for Biotechnology Information, National Library of Medicine, National Institutes of Health, 8600 Rockville Pike, Building 38A, Room 8N–805, Bethesda, MD 20894, USA.)
Robert Cedergren, Département de Biochimie, Université de Montréal, CP 6128, Socc "A", Montréal, Quebec, Canada H3C 3J7.