A Hybrid Evolutionary Approach for Solving Constrained Optimization Problems over Finite Domains

Alvaro Ruiz-Andino, Lourdes Araujo, Fernando Sáenz, José Ruz *

November 16, 2001

Abstract

A novel approach for the integration of evolution programs and constraint solving techniques over finite domains is presented. This integration provides a problem-independent optimization strategy for large-scale constrained optimization problems over finite domains. In this approach, genetic operators are based on an *arc-consistency* algorithm, and chromosomes are arc-consistent portions of the search space of the problem. The paper describes the main issues arising in this integration: chromosome representation and evaluation, selection and replacement strategies, and the design of genetic operators. We also present a parallel execution model for a distributed memory architecture of the previous integration. We have adopted a global parallelization approach that preserves the properties, behavior, and fundamentals of the sequential algorithm. Linear speedup is achieved since genetic operators are coarse-grained, as they perform a search in a discrete space carrying out arc-consistency. The implementation has been tested on a CRAY T3E multiprocessor using a complex constrained optimization problem.

1 Introduction

Evolution programs [30] arise from genetic algorithms [17] (also see [2]), but they consider a richer set of data structures for chromosome representation together with an expanded set of genetic operators. However, handling constraints in evolution programs introduces an additional complexity in the design of the genetic operators. In constrained problems, a minimal change to a feasible solution may be very likely to generate an infeasible one, but infeasible solutions cannot simply be dropped from the search space, because doing so would increase the dufficulty of generating good solutions. Roughly speaking, constraint

*The authors are with the University Complutense of Madrid, Ciudad Universitaria, 28040 Madrid, Spain. (e-mail: alvaro@sip.ucm.es; lurdes@sip.ucm.es; jjruz@eucmax.sim.ucm.es; fernan@eucmax.sim.ucm.es

handling methods in evolutionary computation can be divided into two categories: those aimed to solve constraint satisfaction problems, and those aimed to solve constrained optimization problems.

In recent years there have been several evolutionary approaches to the constraint satisfaction problem, especially for the MAXSAT (maximum satisfiability) problem. These approaches can be divided into three groups:

- Techniques based on exploiting heuristic information of the constraint network [13, 28, 39].
- Techniques that use a fitness function which is adapted during search [6, 14, 35].
- Hybrid techniques [12].

The work presented in this paper belongs to the second category, that is, it is focused on solving optimization problems. Several methods have been proposed for this class of constrained problems. These methods can be classified in the following groups:

- Specialized operators that transform feasible individuals into new feasible individuals [30].
- Penalty functions that reduce the fitness of infeasible solutions [23, 20].
- Repairing infeasible individuals [32, 31].
- Problem-specific representation and genetic operators [10, 40].

However, all these approaches require problem-specific programming. The approach presented in this paper is problem-independent, as long as the problem is suited to be modeled in a constraint programming language over finite domains.

Constraint programming over finite domains (CP(FD)) is one of the most important developments in programming languages in recent years [29, 42]. Constraint satisfaction is the core of many complex optimization problems arising in artificial intelligence [45], including temporal reasoning, resource allocation, scheduling, and hardware design, to name a few. Constraint programming languages [11, 9, 37, 44] provide support for specifying relationships, or *constraints*, among programmer-defined entities. These languages are becoming the method of choice for modeling many types of optimization problems, in particular, those involving heterogeneous constraints and combinatorial search. These languages can be used to model constraint satisfaction problems over finite domains (CSP(FD)), in which the goal is to find values for a set of variables that satisfy a given set of constraints.

The core of a constraint programming languages are constraint-solving methods, such as *arc-consistency*, an efficient and general technique that eliminates inconsistent values from the domains of the variables, reducing the size of the search space both before and while searching. Most constraint programming languages include some kind of enumeration technique, as well as a *branch and bound* (B&B) [25, 29] procedure for optimization problems in order to perform the search. However, for real-sized applications, the search space is too large to be exhaustively enumerated, even using B&B. Thus, hybridization with stochastic techniques such as evolution programs is a promising approach.

According to the previous considerations, evolution programs and constraint programming languages complement each other to solve large-scale constrained optimization problems over finite domains efficiently. The input to the evolution program is a constraint graph generated by a program written in a constraint programming language. Evolution programs are an adequate optimization technique for large search spaces, but they do not offer a problem-independent way to handle constraints. However, constraint-solving techniques are adequate for any discrete combinatorial problem.

This paper presents a novel approach that integrates constraint programming languages and evolution programs. In this approach, genetic operators are based on constraint-solving techniques, which provide a problem-independent optimization strategy to tackle efficiently constrained optimization problems over finite domains with a large search space. The paper describes the main issues arising in this integration: representation and evaluation, selection and replacement strategies, and the design of genetic operators. This approach has been implemented from scratch, but it can also be implemented taking advantage of existing constraint programming languages such as CHIP [11] or ILOG [22].

The integration of constraint-solving techniques over finite domains and genetic algorithms was first proposed by Paredis in [34] where a genetic algorithm was enhanced with forward checking to create chromosomes with unknown valued genes. However, our method makes use of a general constraint solver based on arc-consistency, and differs substantially in the representation and evaluation of chromosomes.

The rest of the paper is organized as follows. Section 2 revises the main concepts of constraint programming over finite domains. Section 3 presents the main points of the integration of constraint-solving and evolution programs. Section 4 briefly reviews the main approaches to parallelize an evolution program and describes our parallel execution model. Section 5 discusses the empirical results obtained for a complex constrained optimization problem, and finally Section 6 presents the conclusions.

2 Constraint Solving over Finite Domains

This section reviews the basic definitions, concepts and techniques used to solve constraint satisfaction problems over finite domains. They are the basis for the chromosome representation and the design of the genetic operators in the developed scheme of integration.

Definition 1 *Finite domain.*
A *finite domain* $\mathcal{FD}$ is a finite set of values $\{e_1, \ldots, e_k\}$. A *finite integer domain*

is a finite set of integers.

Definition 2 *Constraint Satisfaction problem over Finite Domains (CSP(FD))*
A constraint satisfaction problem over finite domains is stated as a tuple $\langle \mathcal{V}, \mathcal{C}, \mathcal{D} \rangle$, where:

- $\mathcal{V} \equiv \{v_1, \ldots, v_n\}$, is a set of variables. They are called *domain variables* as they are constrained to take a value from a finite domain.
- $\mathcal{C} \equiv \{c_1, \cdots, c_m\}$, is a set of constraints among the variables in $\mathcal{V}$, restricting the values that variables can simultaneously take.
- $\mathcal{D} : \mathcal{V} \rightarrow \mathcal{FD}$, is a mapping that assigns an initial finite domain to each variable in $\mathcal{V}$. $\mathcal{D}$ defines the *search space* of the problem, as the set of points (tuples) $\mathcal{D}(v_1) \times \ldots \times \mathcal{D}(v_n)$, where $\times$ denotes the Cartesian product.

Definition 3 *Constraint*
A *constraint* $c \equiv (V_c, R_c)$ is defined by a subset of variables $V_c \subseteq \mathcal{V}$, and a set of allowed tuples of values, R_c, subset of the Cartesian product of the domains of the involved variables V_c. That is, $R_c \subseteq \bigotimes_{v_i \in V_c} \mathcal{D}(v_i)$, where $\bigotimes$ denotes Cartesian product.

An assignment $A = (a_1, \ldots, a_n)$ of values to the variables in $\mathcal{V}$ *satisfies* a constraint $c \equiv (V_c, R_c)$ if and only if $A_{V_c} \in R_c$, where A_{V_c} denotes the projection of A over the subset of variables V_c. That is, R_c defines the points of the search space that satisfy the constraint. However, it is quite usual to define a constraint by means of an functional relation instead of an extensional one. Arithmetic constraints are a typical example of this situation.

The goal in a constraint satisfaction problem is to find an assignment of a value from each $\mathcal{D}(v_i)$ to each variable $v_i \in \mathcal{V}$ which satisfies every constraint $c_i \in \mathcal{C}$.

As an example of CSP(FD), let us consider an arithmetic problem involving two variables, that is, $\mathcal{V} = \{X, Y\}$, related by three arithmetic constraints: $\mathcal{C} = \{X + Y \geq 6,\ X \neq Y,\ |X - Y| \neq 1\}$, and where the initial domain for both variables is the finite interval of integer numbers from 1 to 5, that is, $\mathcal{D}(X) = \{1..5\}$, and $\mathcal{D}(Y) = \{1..5\}$. The assignment $\langle X = 4,\ Y = 2 \rangle$ is one of the eight solutions to this constraint satisfaction problem.

2.1 Arc-consistency

The basis for finite domain constraint solvers is an arc-consistency algorithm [33, 43] that eliminates unsupported (inconsistent) values from the domains of the variables.

Definition 4 *Supported value.*
Given a CSP $\langle \mathcal{V}, \mathcal{C}, \mathcal{D} \rangle$, and a constraint $c \in \mathcal{C}$, $c \equiv (V_c, R_c)$, a value $a \in \mathcal{D}(v_i)$, $v_i \in V_c$, is *supported* with respect to the constraint c if and only if for all $v_j \in V_c - \{v_i\}$ there exists a value $b_j \in \mathcal{D}(v_j)$ such that the tuple $(b_1, \cdots, b_{i-1}, a, b_{i+1}, \cdots, b_k) \in R_c$.

For example, consider the set of variables $\{X, Y\}$, the constraint $X + Y \geq 6$, and the domains $\mathcal{D}(X) = \{1, 2\}$ and $\mathcal{D}(Y) = \{1..5\}$. Value 1 in the domain of X is supported by value 5 in the domain of Y, since $1 + 5 \geq 6$. Similarly, value 2 is supported by values 4 and 5. Consequently, values 4 and 5 in the domain of Y are supported, whereas 1, 2 and 3 are not supported values.

Definition 5 *Arc-consistency.*
Given a CSP $\langle \mathcal{V}, \mathcal{C}, \mathcal{D} \rangle$, a constraint $c \in \mathcal{C}$, $c \equiv (V_c, R_c)$, $V_c \equiv \{v_1, \cdots, v_k\}$, is *arc-consistent* with respect to a search space $\mathcal{D}$ if and only if for all $v_i \in V_c$, for all $a_j \in \mathcal{D}(v_i)$, a_j is supported. A CSP $\langle \mathcal{V}, \mathcal{C}, \mathcal{D} \rangle$ is arc-consistent if and only if all $c_i \in \mathcal{C}$ are arc-consistent with respect to $\mathcal{D}$.

Convention 1 $\sqsubseteq$, $\sqsubset$
Given two search spaces $\mathcal{D}_1$ and $\mathcal{D}_2$, we will write $\mathcal{D}_1 \sqsubseteq \mathcal{D}_2$ if and only if $\forall v_i \in \mathcal{V}$: $\mathcal{D}_1(v_i) \subseteq \mathcal{D}_2(v_i)$. Similarly, $\mathcal{D}_1 \sqsubset \mathcal{D}_2$ will denote that $\mathcal{D}_1 \sqsubseteq \mathcal{D}_2$ and that there exists at least one variable $v_i \in \mathcal{V}$ such that $\mathcal{D}_1(v_i) \subset \mathcal{D}_2(v_i)$.

Definition 6 *Largest arc-consistent search subspace.*
Given a CSP $\mathcal{P} \equiv \langle \mathcal{V}, \mathcal{C}, \mathcal{D} \rangle$, $\mathcal{D}^*$ is the largest arc-consistent search space for $\mathcal{P}$ if and only if $\mathcal{D}^* \sqsubseteq \mathcal{D}$, $\langle \mathcal{V}, \mathcal{C}, \mathcal{D}^* \rangle$ is arc-consistent, and there is no other $\mathcal{D}'$ such that $\langle \mathcal{V}, \mathcal{C}, \mathcal{D}' \rangle$ is arc-consistent and $\mathcal{D}^* \sqsubset \mathcal{D}'$. The largest arc-consistent search space exists and it is unique [43].

An *arc-consistency algorithm* takes as input argument a constraint satisfaction problem $\mathcal{P} \equiv \langle \mathcal{V}, \mathcal{C}, \mathcal{D} \rangle$ and returns either the largest arc-consistent search subspace for $\mathcal{P}$, if it does not contain any empty domain, or `INCONSISTENT`, if it does. A simplified version of the arc-consistency algorithm is shown in Figure 1. It uses a queue to keep track of constraints to be visited. Unsupported values from the domains of the variables are removed until a fixed point is reached or until inconsistency is detected (a domain results empty). Whenever the domain of a variable is modified as the result of revisiting a constraint, the constraints where the variable appears are queued. The algorithm terminates when the queue is empty or when inconsistency is detected. A specialized version of this algorithm for a number of the most important classes of constraints runs in $O(ed)$ in time, where e is the number of constraints and d is the size of the largest domain [43].

For example, given the CSP(FD) $\langle \{X, Y\}, \{X+Y \geq 6, X \neq Y, |X-Y| \neq 1\}, \{1..3\} \times \{1..5\} \rangle$, the output of the arc-consistency algorithm will be $\mathcal{D}(X) = \{1..3\}$, $\mathcal{D}(Y) = \{3..5\}$ (unsupported values have been removed from the domains of the variables). However, for the same set of variables and constraints, but with the search space $\{1..3\} \times \{1..3\}$, the algorithm will detect inconsistency.

Constraint satisfaction problems usually describe hard search problems, therefore some kind of search with backtracking is necessary to solve them. Most constraints solvers perform a search that enforces arc-consistency at each node of the search tree. The arc-consistency algorithm is invoked each time a value is tried for a variable, removing unsupported values from the domains of the other

```
function Arc-consistency( ⟨𝒱,𝒞,𝒟⟩ ) :  Search subspace
begin
  InitQueue(Q)
  for each c_i ∈ 𝒞 do Enqueue(c_i, Q) end-for
  while Not EmptyQueue(Q) do
    DeQueue(Q, c ≡ (V_c, R_c))
    for each v ∈ V_c do
      NewDomain := RemoveUnsupported(𝒟(V_c), R_c)
      if NewDomain = ∅ then
        return INCONSISTENT
      elsif NewDomain ⊂ 𝒟(v) then
        𝒟(v) := NewDomain
        for each c' ≡ (V'_c, R'_c) such that v ∈ V'_c do
          Enqueue(Q, c')
        end-for
      end-if
    end-for
  end-while
  return 𝒟
end
```

Figure 1: Arc-consistency algorithm.

variables, thus reducing the search space. If some domain is left empty (inconsistency is detected) that branch cannot lead to a solution and backtracking is performed.

Definition 7 *Singleton domain.*
A finite domain d is a singleton domain if and only if it consists of just one value a, that is $d = \{a\}$. A domain with more than one value will be called non-singleton domain.

Definition 8 *Singleton variable.*
Given a search space $\mathcal{D}$, a domain variable v is said to be singleton if and only if $\mathcal{D}(v)$ is a singleton domain. Variables whose domains consists of multiple values are called non-singleton variables.

Definition 9 *Singleton search space.*
Given a search space $\mathcal{D}$, it will be called a singleton search space if and only if all domain variables are singleton, that is, it only contains one point. A search space with more than one point will be called non-singleton.

Note that arc-consistency is not complete, that is, a CSP $\mathcal{P}$ can be arc-consistent with respect to a search space $\mathcal{D}$ but there may be no solution to $\mathcal{P}$ in $\mathcal{D}$. For example, the CSP(FD) $\langle\{X,Y\}, \{X+Y \geq 6,\, X \neq Y,\, |X-Y| \neq 1\}, \{4,5\} \times \{4,5\}\rangle$ is arc-consistent (every constraint is arc-consistent by its own), but none of the four points in $\{4,5\} \times \{4,5\}$ is a solution. However, if $\mathcal{P}$ is arc-consistent with respect to $\mathcal{D}$ and $\mathcal{D}$ is singleton, then $\mathcal{D}$ is a solution to $\mathcal{P}$. Moreover, if $\mathcal{D}$ is singleton and $\mathcal{P}$ is not arc-consistent with respect to $\mathcal{D}$, then it is not a solution to $\mathcal{P}$. Therefore, solution and arc-consistent singleton search space are equivalent terms.

2.2 Arc-Consistent Search Subspaces

In this subsection we define a key concept, *arc-consistent subspace*, the basis for representation in our approach.

Definition 10 *Arc-Consistent Subspace.*
Given a CSP $\mathcal{P} \equiv \langle \mathcal{V}, \mathcal{C}, \mathcal{D} \rangle$, a search space $\mathcal{D}'$ is an arc-consistent subspace (AC-Subspace) with respect to $\mathcal{P}$ if and only if $\mathcal{D}' \sqsubseteq \mathcal{D}$ and the CSP $\langle \mathcal{V}, \mathcal{C}, \mathcal{D}' \rangle$ is arc-consistent.

Convention 2 $\mathcal{D} \sqcap v_i$ *in* d
Given a search space $\mathcal{D} \equiv \mathcal{D}(v_1) \times \ldots \times \mathcal{D}(v_n)$ and a finite domain d, $\mathcal{D} \sqcap v_i$ *in* d denotes the search space $\mathcal{D}(v_1) \times \ldots \times \mathcal{D}(v_{i-1}) \times (\mathcal{D}(v_i) \cap d) \times \mathcal{D}(v_{i+1}) \times \ldots \times \mathcal{D}(v_n)$.

The usual way to generate an arc-consistent subspace $\mathcal{D}'$ with respect to a CSP $\mathcal{P} \equiv \langle \mathcal{V}, \mathcal{C}, \mathcal{D} \rangle$ is to invoke the arc-consistency algorithm taking as input argument the CSP $\mathcal{P}$ plus an additional constraint over one of the variables in $\mathcal{V}$, that is, the CSP $\langle \mathcal{V}, \mathcal{C}, \mathcal{D} \sqcap v_i \ in \ d \rangle$.

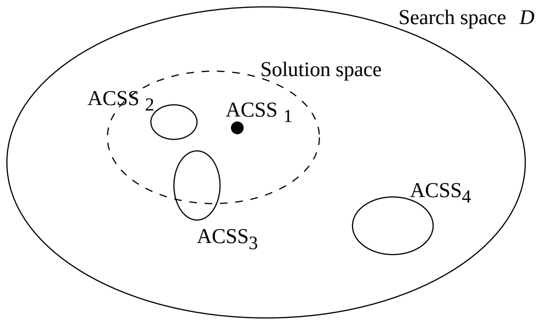

Figure 2: Arc-consistent subspaces.

Figure 2 depicts a graphical representation of different arc-consistent subspaces for a generic initial search space $\mathcal{D}$. Arc-Consistent subspace $ACSS_1$ (black dot) is a singleton search space, that is, a single point, and therefore it lies in the solution space. $ACSS_2$ is not singleton, but it only contains points that are solutions to the problem. $ACSS_3$ contains some points that are solutions to the problem and some others that are not. Note that it is also possible to have an AC-Subspace that do not contain any point solution to the problem, but it cannot exist a singleton AC-Subspace outside the solution space. Figure 3 illustrates the AC-Subspace concept in the context of an arithmetic CSP.

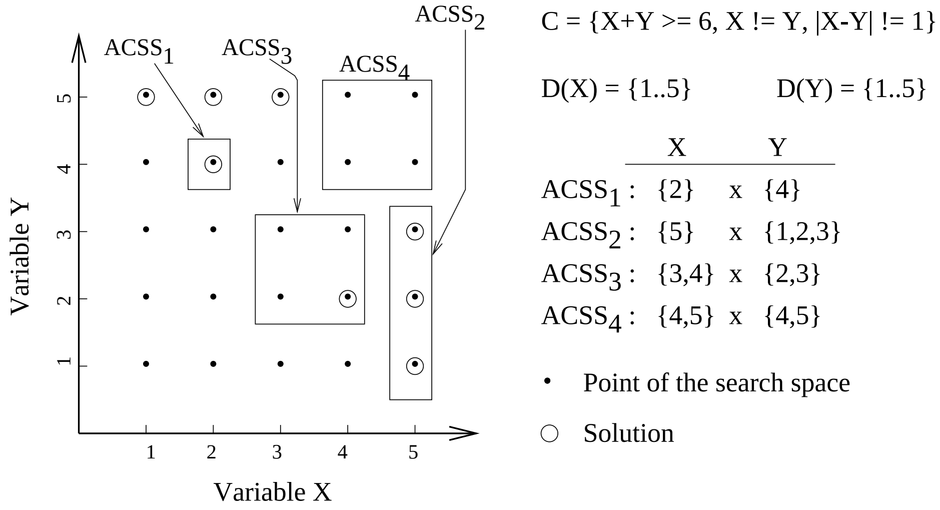

Figure 3: Examples of AC-Subspaces for the CSP $\langle\{X, Y\}, \{X+Y \geq 6, X \neq Y, |X-Y| \neq 1\}, \{1..5\} \times \{1..5\}\rangle$. $ACSS_1$ is singleton (thus a solution). $ACSS_2$ only contains solution points. $ACSS_3$ contains four points, but only one of them is a solution. $ACSS_4$ contains four points, none of them being solution, although it is also an Arc-Consistent Subspace.

2.3 Constrained Optimization Problems over Finite Domains

In many occasions, we are not looking for just a feasible solution to a constraint problem, but rather the best feasible solution according to an *objective function*. This kind of problem for which solutions are not equally preferred, are called *constraint optimization problems* over finite domains. Many complex search problems such as resource allocation, scheduling and hardware design can be stated as constrained optimization problems over finite domains.

Definition 11 *Constraint Optimization Problem over Finite Domains* (COP(FD)) A constraint optimization problem over finite domains is stated as a pair $\langle \mathcal{P}, f\rangle$, where $\mathcal{P} \equiv \langle \mathcal{V}, \mathcal{C}, \mathcal{D}\rangle$ is a constraint satisfaction problem, and $f : \mathcal{N} \times \cdots \times \mathcal{N} \rightarrow \mathcal{N}$ is the objective function to optimize (minimize or maximize).

An example of CSP is the *n-queens* problem, which requires to place n queens in a $n \times n$ chessboard in such a way that no queen attacks each other, i.e., they are not in the same column, row or diagonal. This problem can be modeled with a set of n variables $\mathcal{V} = \{v_1, \cdots, v_n\}$ and a search space $\mathcal{D} = d_1 \times \ldots \times d_n$, where each $d_i = \{1, \ldots, n\}$. Each variable represents one row of the chessboard. The assignment $v_3 = 5$ indicates that a queen is positioned in the fifth column of the third row. The set of constraints $\mathcal{C}$ is:

- $v_i \neq v_j, 1 \leq i < j \leq n$ (no two queens in the same column).

- $|v_i - v_j| \neq (j-i), 1 \leq i < j \leq n$ (no two queens in the same diagonal).

The constraint of no two queens in the same row is implicit in the representation.

As an example of COP(FD), consider the *valued n-queens* problem [34] an extension of the n-queens problem. Now there is a weight associated to each board location (i,j), defined by the matrix $w(i,j)$, $1 \leq i \leq n$, $1 \leq j \leq n$. Now, the goal is to find a solution to the n-queens problem that maximizes the sum of the weights of the positions where the queens are placed, that is, the objective function is:

$$f(\mathcal{V}) = \sum_{i=1}^{n} w(i, v_i)$$

Figure 4 shows an instance of the problem for eight queens, as well as its optimal solution.

	1	2	3	4	5	6	7	8
V1	6	28	$\boxed{20}$	9	15	18	19	22
V2	9	22	14	21	23	$\boxed{28}$	0	3
V3	16	5	6	8	0	14	5	$\boxed{11}$
V4	$\boxed{30}$	22	16	21	19	16	23	13
V5	18	30	19	$\boxed{30}$	22	3	23	25
V6	30	5	26	25	18	1	$\boxed{27}$	2
V7	20	21	6	6	$\boxed{21}$	13	3	31
V8	6	$\boxed{14}$	0	4	3	27	19	2

Figure 4: Instance of the valued 8-queens problem. Each square of the board shows its weight $w(i,j)$. Remarked squares form the optimal solution $\mathcal{V} = \{3, 6, 8, 1, 4, 7, 5, 2\}$, which yields a value $f(\mathcal{V}) = 181$.

As an example of the formulation of a constrained optimization problem in a constraint programming language, Figure 5 shows the code for the valued n-queens problem in the constraint programming language OPL [44]. Keyword `int` defines constant integers, whilst keyword `var int` declares domain variables over finite integer domains. The default optimization strategy is based on a depth-first branch & bound algorithm.

3 Constrained Evolution Programs

This section discusses in detail the hybridization of arc-consistency and evolution programs in order to solve constraint optimization problems over finite domains. These two mechanisms complement each other: constraint-solving techniques open a flexible and efficient way to handle constraints in evolution programs,

```
int N = ...;
int Weights[1..N,1..N] = ...;

var int ObjectiveVariable;
var int Queens[1..N] in 1..N;

maximize ObjectiveVariable subject to {
  ObjectiveVariable = sum(i in 1..N) Weights[i,Queens[i]];
  forall (ordered i,j in 1..N) {
     Queens[i] <> Queens[j];
     Queens[i] - Queens[j] <> j - i;
     Queens[j] - Queens[i] <> j - i;
  }
};
```

Figure 5: Valued n-queens in OPL.

while evolution programs allow to deal with large-scale search spaces. This hybridization implies coming up with a solution for chromosome representation, chromosome evaluation, and the design of genetic operators.

3.1 Representation

Classical approaches to handle constraints in evolution programs use one or more problem-specific tricks as penalty functions, repair algorithms or linear recombination. We will make use of generic constraint-solving techniques over finite domains in order to design a novel approach to handle constraints. In our approach, a chromosome is not represented as an array of values, but as an array of finite domains, that is, not just a single point but a subset of the points of the search space. Moreover, all chromosomes in the population will be arc-consistent search spaces (AC-Subspace) to the problem, generated by means of genetic operators based on the `Arc-consistency` function, which returns AC-Subspaces where inconsistent values have been removed from the domains of the variables. Therefore, a chromosome will be a subspace of the initial search space containing many or none solutions, and in particular it may be a singleton search space, thus a solution to the constraint satisfaction problem. Evaluation, selection and replacement strategies ensure convergence to singleton chromosomes. Given this chromosome representation, genetic operators are deterministic search procedures based on arc-consistency techniques that take as input argument an AC-Subspace, and generate a new AC-Subspace.

3.2 Evaluation

Since a chromosome is an AC-Subspace, chromosome evaluation is more elaborate than the calculation of a fitness function for a single point of the search space. In order to take advantage of the chromosome representation, we make a dual evaluation: for each chromosome we compute a *fitness* value and a *feasibility* value.

Fitness plays the usual role, so its calculation is based on the objective function. The need of the feasibility value comes from the fact that an AC-Subspace may contain just a single point (and in this case it is a solution to the problem), or it may contain many points (the whole search space in the worst case). Feasibility indicates how far away a chromosome is from being singleton, measuring the relative size of the AC-Subspace. Chromosomes closer to be singleton have a higher feasibility value. Selection mechanisms will exploit the advantages of this dual evaluation. Similar dual evaluation approaches have been proposed [10], although with a quite different view of the feasibility value.

Figure 6 shows the expected evolution of the population as generations transpire. The initial population is expected to have low average values for fitness and feasibility, whereas final population will have higher values, and in particular there will be singleton chromosomes, that is, solutions to the problem.

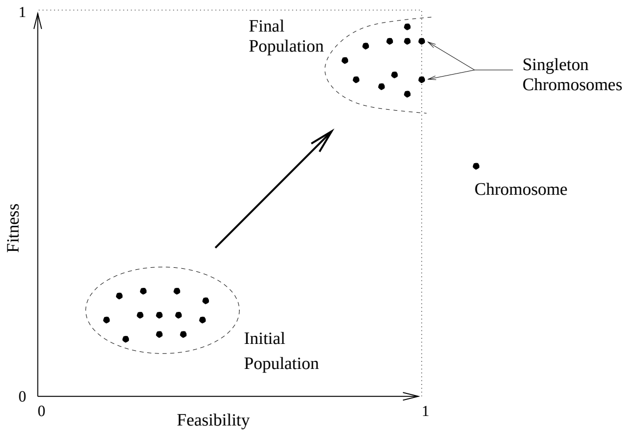

Figure 6: Expected evolution of the population in the fitness/feasible dual evaluation plane.

3.2.1 Fitness

In a COP(FD) the objective function $f : \mathcal{N} \times \cdots \times \mathcal{N} \rightarrow \mathcal{N}$ is defined for those points of the search space that satisfies the constraints. In our approach, a chromosome is not a point of the search space but an arc-consistent set of points that may include points that do not satisfy the constraints. Constraint programming itself offers the way to evaluate the fitness function of an AC-Subspace. In our context, a COP(FD) $\langle \mathcal{V}, \mathcal{C}, \mathcal{D}, f \rangle$ is defined using a constraint programming language, thus the objective function f is defined as a new domain variable v_{of} along with constraints relating v_{of} and $\mathcal{V}$. The arc-consistency algorithm prunes the domain of v_{of} as it prunes any other domain variable, so arc-consistency returns the lower and upper bound of the objective function for a given AC-Subspace. We will refer to v_{of} as the objective variable and $\mathcal{D}(v_{of})$ will denote its domain in a given search space $\mathcal{D}$.

For instance, let us return to the valued n-queens problem. The objective function to maximize is the sum of the weights of the positions occupied by the queens. The objective variable is defined as:

$$v_{of} = \sum_{i=1}^{n} w(i, v_i)$$

that is, an arithmetic constraint involving the variables v_i which represent the queens positions. In general, $\mathcal{D}(v_i)$ is not singleton, therefore $w(i, v_i)$ would be a finite domain instead of an integer, and so is the resulting sum.

Figure 7 shows the approach for computing fitness. A chromosome (an AC-Subspace, denoted by $\mathcal{D}$) has associated a domain $\mathcal{D}(v_{of})$, the domain of the variable v_{of} in the search space $\mathcal{D}$. Fitness is computed from the value of $\mathcal{D}(v_{of})$ in the AC-Subspace being evaluated ($\mathcal{D}$), normalized with respect to the size of $\mathcal{D}_{ini}(v_{of})$, the domain of v_{of} in the initial search space. First, a weighted average (fit) is computed from the minimum and maximum values of $\mathcal{D}(v_{of})$ ($min_{\mathcal{D}}$ and $max_{\mathcal{D}}$ resp.) The weight, w_{fit}, is a problem-dependent parameter. Its most usual value is 0.5. Finally, fit is normalized with respect to the minimum and maximum values of $\mathcal{D}_{ini}(v_{of})$ ($min_{\mathcal{D}_{ini}}$ and $max_{\mathcal{D}_{ini}}$ resp.), in order to obtain a final real value that ranges from 0 (worst) to 1 (best).

$$\begin{array}{ll} min_{\mathcal{D}_{ini}} = min(\mathcal{D}_{ini}(v_{of})) & max_{\mathcal{D}_{ini}} = max(\mathcal{D}_{ini}(v_{of})) \\ min_{\mathcal{D}} = min(\mathcal{D}(v_{of})) & max_{\mathcal{D}} = max(\mathcal{D}(v_{of})) \end{array}$$

$$fit = min_{\mathcal{D}} + w_{fit}(max_{\mathcal{D}} - min_{\mathcal{D}})$$

$$Fitness(\mathcal{D}) = \frac{fit - min_{\mathcal{D}_{ini}}}{max_{\mathcal{D}_{ini}} - min_{\mathcal{D}_{ini}}}$$

Figure 7: Fitness calculation for an AC-Subspace $\mathcal{D}$ (for a maximization problem).

3.2.2 Feasibility

The feasibility measures the relative size of the AC-Subspace with respect to the initial search space, indicating how far a chromosome is from being singleton.

Figure 8 shows the calculation of feasibility for a chromosome $\mathcal{D}$. First, we define the function $rs : \mathcal{V} \rightarrow \mathcal{R}$ which returns the normalized size of the domain of a variable v in a search space $\mathcal{D}$ with respect to its size in the initial search space $\mathcal{D}_{ini}$. Next, feasibility is computed as a weighted average between $fea_{v_{of}}$ and $fea_{\mathcal{V}}$, where weight, w_{fea}, is a problem-dependent parameter, $fea_{v_{of}}$ is the value returned by the rs function for the objective variable v_{of}, and $fea_{\mathcal{V}}$ is the average value of $rs(v_i)$, $v_i \in \mathcal{V}$. Feasibility values range from 0 (when $\mathcal{D} = \mathcal{D}_{ini}$) to 1 (when $\mathcal{D}$ is singleton).

$$rs(v) = \frac{size(\mathcal{D}(v))-1}{size(\mathcal{D}_{ini}(v))-1}$$

$$fea_{v_{of}} = rs(v_{of})$$

$$fea_{\mathcal{V}} = \frac{1}{n} \times \sum_{i=1}^{n} rs(v_i)$$

$$Feasibility(\mathcal{D}) = 1 - w_{fea} fea_{v_{of}} - (1 - w_{fea}) fea_{\mathcal{V}}$$

Figure 8: Feasibility calculation for an AC-Subspace $\mathcal{D}$. $\mathcal{D}_{ini}$ denotes the initial search space.

In order to clarify the evaluation of chromosomes, consider the valued 8-queens example problem. The chessboard on the left in Figure 9 represents a chromosome $\mathcal{D}_1$. Filled squares are the possible rows for each variable. $\mathcal{D}_1(v_2)$, the set of columns where queen at the second row may be placed, is $\{6\}$, thus v_2 is singleton in $\mathcal{D}_1$. So are v_4, v_7, and v_8. However, v_1, v_3, v_5, and v_6 are not singleton. For example $\mathcal{D}(v_1) = \{1, 3\}$, so the queen at the first row may be placed at columns 1 or 3. The chessboard on the right in Figure 9 represents a chromosome (AC-Subspace) $\mathcal{D}_2$, which is singleton and also a feasible solution to the problem. Note that $\mathcal{D}_2 \sqsubset \mathcal{D}_1$.

Table 1 illustrates values of fitness and feasibility for the chromosomes represented in Figure 9, as well as intermediates values taking part in the calculation (for $w_{ade} = 0.5$ y $w_{via} = 0.5$).

Table 1: Evaluation of chromosomes in Figure 9.

	$\mathcal{D}_1$	$\mathcal{D}_2$
$\mathcal{D}_{ini}(v_f)$	[26..220]	
$\mathcal{D}(v_f)$	[139..181]	[153..153]
fit	160	153
$Fitness$	0.69	0.65
$fea_{v_{of}}$	0.22	0.00
$fea_{\mathcal{V}}$	0.09	0.00
$Feasibility$	0.85	1.00

	1	2	3	4	5	6	7	8
V1	$\boxed{6}$	28	$\boxed{20}$	9	15	18	19	22
V2	9	22	14	21	23	$\boxed{28}$	0	3
V3	$\boxed{16}$	$\boxed{5}$	$\boxed{6}$	8	0	14	5	11
V4	30	22	16	21	19	16	$\boxed{23}$	13
V5	$\boxed{18}$	30	19	$\boxed{30}$	22	3	23	25
V6	$\boxed{30}$	5	26	$\boxed{25}$	18	1	27	2
V7	20	21	6	6	21	13	3	$\boxed{31}$
V8	6	14	0	4	$\boxed{3}$	27	19	2

$\mathcal{D}_1 = (\{1,3\},\{6\},\{1,2,3\},\{7\},\{1,4\},\{1,4\},\{8\},\{5\})$

	1	2	3	4	5	6	7	8
V1	6	28	$\boxed{20}$	9	15	18	19	22
V2	9	22	14	21	23	$\boxed{28}$	0	3
V3	16	$\boxed{5}$	6	8	0	14	5	11
V4	30	22	16	21	19	16	$\boxed{23}$	13
V5	$\boxed{18}$	30	19	30	22	3	23	25
V6	30	5	26	$\boxed{25}$	18	1	27	2
V7	20	21	6	6	21	13	3	$\boxed{31}$
V8	6	14	0	4	$\boxed{3}$	27	19	2

$\mathcal{D}_2 = (\{3\},\{6\},\{2\},\{7\},\{1\},\{4\},\{8\},\{5\})$

Figure 9: Example of chromosomes for the valued 8-queens problem.

3.3 Integration Algorithm

The main algorithm Arc-Consistent Evolution Program follows the structure of a steady-state genetic algorithm. Figure 10 shows the pseudo-code. Functions in **bold** face invoke the Arc-Consistency algorithm. Words in *italics* are parameters to be set.

The `initialization` step is a loop that generates a certain number (*population_size*) of chromosomes by means of the `AC-random-search` procedure. The `evaluation` step computes the fitness and feasibility values for each AC-Subspace (chromosome) in the population. In the `selection` step, some chromosomes are marked as survivors, so they won't be replaced by the new AC-Subspaces generated by crossover. Best chromosomes are more likely to be selected, as `rank-selection` procedure performs a biased stochastic selection. An elitist policy is used, so the best chromosome always survives. The `alteration` step is divided into the application of two genetic operators, `AC-crossover` and `AC-mutation`, both AC-Subspace generators guided by the arc-consistency algorithm. New chromosomes generated by crossover take the place in the population of those chromosomes not chosen to survive. Mutation is applied to those chromosomes selected to survive with a probability *mutation_rate*. New AC-Subspace generated by means of the mutation operators replaces the chromosome used to generate it.

3.4 Selection and Replacement Strategies

The dual evaluation of chromosomes allows a more elaborate and sensible policy to perform the selective pressure. There are two points in a steady-state genetic algorithm where chromosomes have to be selected: chromosomes that survive to the next generation and chromosomes that take part in crossover. In both

```
function AC-evolution-program ( ⟨𝒱,𝒞,𝒟⟩, ObjectiveVariable ):  Solution
Var    X, X1, X2 :  AC-Subspace
       Pop :  Array[1..population_size] of AC-Subspace
begin
  /* initialization */
  for i := 1 to population_size do
     Pop[i] := AC-random-search(𝒱,𝒞,𝒟)
  end-for
  /* evaluation */
  for i := 1 to population_size do
     Fitness[i] := AC-fitness(Pop[i], 𝒟, ObjectiveVariable)
     Feasibility[i] := AC-feasible(Pop[i], 𝒟, ObjectiveVariable))
  end-for
  /* main loop */
  for t := 1 to number_of_generations do
     /* selection */
     for i := 1 to (1-xover_rate) * population_size - 1 do
        X := rank-selection(Pop, Fitness + fea_weight(t)*Feasibility)
        mark X to survive
     end-for
     mark best singleton chromosome in Pop to survive
     /* alteration */
     for i := 1 to population_size do
        if not marked-to-survive(Pop[i]) then
           X1 := rank-selection(Pop, Fitness)
           X2 := rank-selection(Pop, Feasibility)
           Pop[i] will-be-replaced-by AC-crossover(X1,X2,𝒱,𝒞,𝒟)
        else
           if random-between(0,1) > mutation_rate then
             Pop[i] will-be-replaced-by AC-mutation(Pop[i],𝒱,𝒞)
           end-if
        end-if
     end-for
     update Pop
     evaluate Pop
  end-for
  return best singleton chromosome in Pop
end
```

Figure 10: AC evolution program

cases we use a rank-based selection (Figure 11) with a quadratic bias, but with a different sorting criterion.

```
function rank-selection ( Pop, Criterion ) :  Chromosome
begin
  Sort Pop by Criterion
  x := random_real_between(0,1)
  i := population_size * x^2
  return Pop[i]
end
```

Figure 11: Rank based selection with a quadratic bias.

In order to select chromosomes to survive, a value $surv(i,t)$ is computed for each chromosome i for each generation t. $surv(i,t)$ is a linear combination of the fitness and the feasibility: $surv(i,t) = fitness(i) +$ ***fea_weight(t)*** $\times feasibility(i)$. As the number of generations increases, the weight of fitness (***fea_weight(t)***) increases, in order to help convergence towards singleton chromosomes (solutions). Figure 12 illustrates this policy by means of level lines of equal probability in a representation of the population in the fitness/feasibility plane. It shows examples of the level lines of equal probability to be chosen to survive for the first and last generations. Chromosomes with higher fitness are preferred in the first generations, whilst in the last generations chromosomes with higher feasibility are preferred, helping convergence towards feasible solutions.

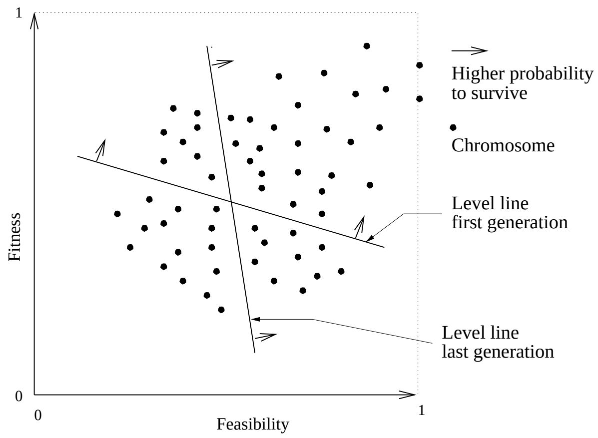

Figure 12: Level lines of equal probability for the selection of chromosomes to survive.

In order to select chromosomes to take part in crossover, the sorting criterion to select one of the parents is the fitness value, whereas the other parent is selected using the feasibility value as the sorting criterion. This heuristic leads to the crossover of a promising parent in terms of fitness, with a promising parent in terms of being closer to be a solution, expecting to produce fit and feasible chromosomes. Figure 13 illustrates this policy by means of level lines of equal probability in a representation of the population in the fitness/feasibility plane. Level lines of probability for choosing the 'fit' chromosome are horizontal, while level lines of probability for choosing the 'feasible' chromosome are vertical.

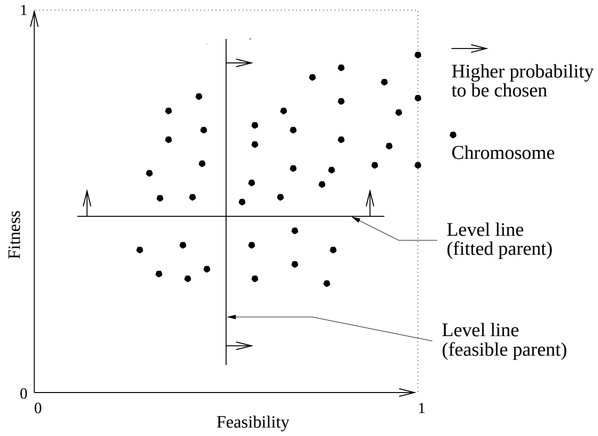

Figure 13: Level lines of equal probability for selection of chromosomes to take part in crossover.

3.5 Initial Population Generation

Each chromosome of the initial population is generated by means of the `AC-random-search` function. This heuristic and stochastic procedure generates a new AC-Subspace which is a randomly generated subspace of the AC-Subspace taken as input argument. In order to generate the initial members of the population, `AC-random-search` is called with the whole search space as an input argument. Figure 14 shows the `AC-random-search` algorithm.

A common way to implement an AC-Subspace generator is a variable-value choice heuristic. Variable ordering is randomly established (permutation `p[1..n]`). The domain to be sequentially assigned to each variable $v_{p[i]}$ is a singleton value randomly chosen from variable's current domain. If the assignment leads to inconsistency, the AC-Subspace is left unchanged (thus backtracking never occurs) and the algorithms proceeds with the next variable.

Continuing with the valued 8-queens problem, Figure 15 shows an example of the execution of the `AC-random-search` algorithm with the initial search space as input $\mathcal{D} = \{1, \ldots, 8\} \times \ldots \times \{1, \ldots, 8\}$. Variables are processed in the following randomly chosen order: v_2, v_4, v_5, v_1, v_6, v_3, v_7, v_8. Each variable is

```
function AC-random-search( V,C,D ):  AC-Subspace
begin
  p[1..n] := random-permutation-between(1, n)
  for i := 1 to n do
     k := random_int_between( min(D(v_p[i])), max(D(v_p[i])) )
     NewD := Arc-Consistency(V,C, D ⊓ v_p[i] in {k} )
     if NewD != INCONSISTENT then
        D := NewD
     end-if
  end-for
  return D
end
```

Figure 14: AC random search

assigned a randomly chosen value from its domain. The first four assignments (black circles) prunes the search space (black squares). Each of the last four assignments lead to inconsistency, therefore the ACSS is not updated, which leaves the board in the last arc-consistent state (board B4). For example, board B5 shows that starting from board B4 and assigning to v_6 the value 7 (randomly chosen among $\{1,7\}$, the domain of v_6 in B4) the arc-consistency algorithm detects inconsistency, as v_7 and v_8 become singleton (gray circles), which leaves no option for v_3. Similar situations occur when processing variables v_3, v_7, and v_8. The final AC-Subspace obtained is displayed in board B4: $\{3\} \times \{6\} \times \{4,8\} \times \{2\} \times \{5\} \times \{1,7\} \times \{4,8\} \times \{1,4,7\}$.

3.6 Genetic Operators

Genetic operators generate the new chromosomes that will be added to the next population, taking as input chromosomes from the current population. In our approach, genetic operators generate AC-Subspace whose arc-consistency is guaranteed by the `Arc-Consistency` algorithm. Genetic operators implement stochastic AC-Subspace builders, taking previous AC-Subspace as an input information to guide the search for a new AC-Subspace. Genetic operators are clustered in two classes: mutation and crossover. Mutation creates a new AC-Subspace by means of a small change in a single chromosome, whereas crossover searches for a new AC-Subspace combining information from two chromosomes. The design of genetic operators is a crucial point in all evolution algorithms because they must guarantee that new individuals inherit their ancestors' properties, and also they must allow the exploration of new areas of the search space.

3.6.1 Crossover

New chromosomes generated by means of the crossover operator replace those not selected to survive. Parent chromosomes are chosen using the `rank-selection`

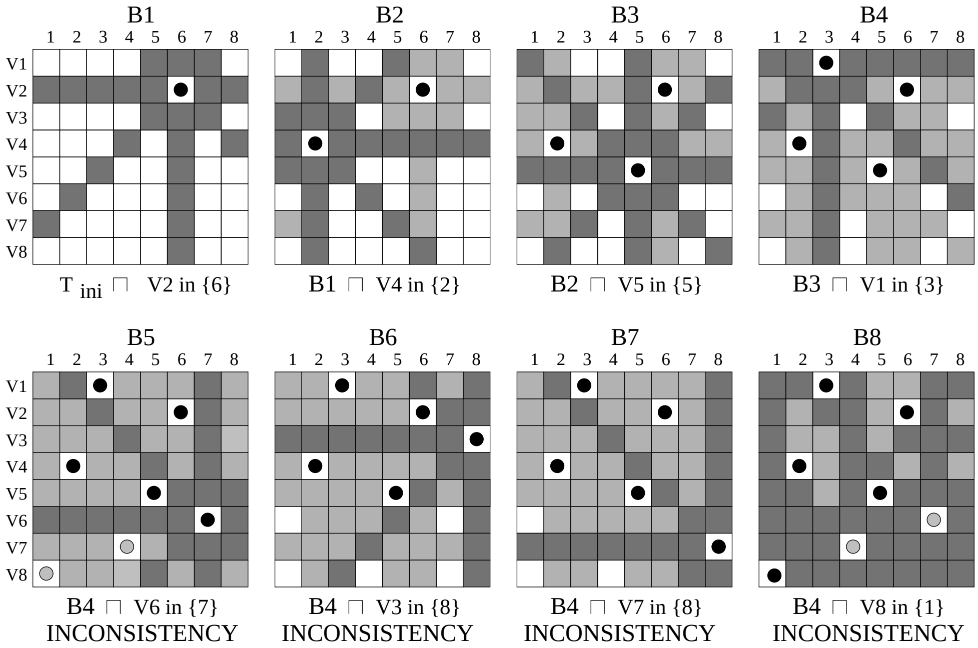

Figure 15: Example of the generation of an initial chromosome for the valued 8-queens problem. White squares denote allowed values, while gray ones denote forbidden values. Light gray squares come from the previous board, while dark gray ones represent values removed in the current arc-consistency enforcement. Circles correspond to singleton variables. Black circles are assignments, while gray ones denote that the value is the result of the current arc-consistency enforcement.

function, as algorithm in Figure 10 shows. One parent is selected biased towards fitness (probably a high fit but not feasible solution) and the other parent is selected biased towards feasibility (probably a feasible solution, but low fit).

Different crossovers operators were tried on a set of benchmarks; an uniform AC-crossover search algorithm, as implemented in Figure 16, showed best results in average. Given two AC-Subspaces, the AC-crossover operator generates a new AC-Subspace, which is a mixture of the two parents. In the first place, k (a random value between 1 and $n-1$) randomly chosen variables $v_{p[1]}$ through $v_{p[k]}$ are assigned the corresponding domain from the first parent ($v_{p[i]}$ *in* $ACSS_1(v_{p[i]})$, line 5). Next, arc-consistency is enforced, and finally, remaining variables $v_{p[k+1]}$ through $v_{p[n]}$ are assigned, enforcing arc-consistency, the corresponding domain from the second parent ($v_{p[i]}$ *in* $ACSS_2(v_{p[i]})$, line 9). If the arc-consistency algorithm detects inconsistency, the domain of the variable is left unchanged.

Figure 17 illustrates an example of crossover for the valued 8-queens prob-

```
function AC-crossover( ACSS_1, ACSS_2, V, C, D_ini ) :  AC-Subspace
begin
  p[1..n] = random-permutation-between(1, n)
  k := random_int_between(1, n-1)
  ACSS := D_ini
  for i = 1 to k do
     ACSS := ACSS ⊓ v_p[i] in ACSS_1(v_p[i])
  end-for
  ACSS := Arc-Consistency(V,C, ACSS )
  for i = k+1 to n do
     NewACSS := Arc-Consistency(V,C, ACSS ⊓ v_p[i] in ACSS_2(v_p[i]) )
     if NewACSS != INCONSISTENT then
        ACSS := NewACSS
     end-if
  end-for
  return ACSS
end
```

Figure 16: AC-Crossover. $ACSS_1$ and $ACSS_2$ are the parent chromosomes and $\mathcal{D}_{ini}$ the initial search space.

lem. Board B5 represents the new chromosome generated from the crossover of chromosomes represented in P1 and P2. P2 is a singleton chromosome, whereas P1 is not. The domain of the objective variable (sum of the possible occupied squares) is {121..190} and {155} for P1 and P2, resp. Variables are processed in the following order, randomly chosen: $v_1, v_2, v_5, v_4, v_3, v_7, v_8, v_6$. A random number k of variables are taken directly from parent P1, let $k = 3$. Therefore, domain {4,5} is assigned to variable v_1, value 2 is assigned to v_2, and domain {1,4,7,8} is assigned to v_5. Then, arc-consistency is enforced, obtaining board B1. The rest of variables are sequentially processed in the established order, v_4, v_3, v_7, v_8, v_6, by assigning to each variable the corresponding domain from parent P2 and enforcing arc-consistency. That is, starting from B1 we try B1 $\sqcap$ v_4 *in* {1} obtaining board B2 after arc-consistency enforcing. Next, B2 $\sqcap$ v_3 *in* {4} is tried, which leads to inconsistency (no value left for v_5), and therefore board B2 is kept. Then, we try B2 $\sqcap$ v_7 *in* {3} obtaining B4, and next B4 $\sqcap$ v_8 *in* {6} obtaining B5, which is a singleton chromosome. Therefore, B5 is the resulting offspring, as any further assignment will leave the board unchanged or lead to inconsistency. B5 yields an value of 168 for the objective function.

3.6.2 Mutation

The traditional mutation operator generates a new chromosome by means of a small change in an old one, allowing the exploration of new areas of the search space, and escaping from local optima. The mutation operator in our system

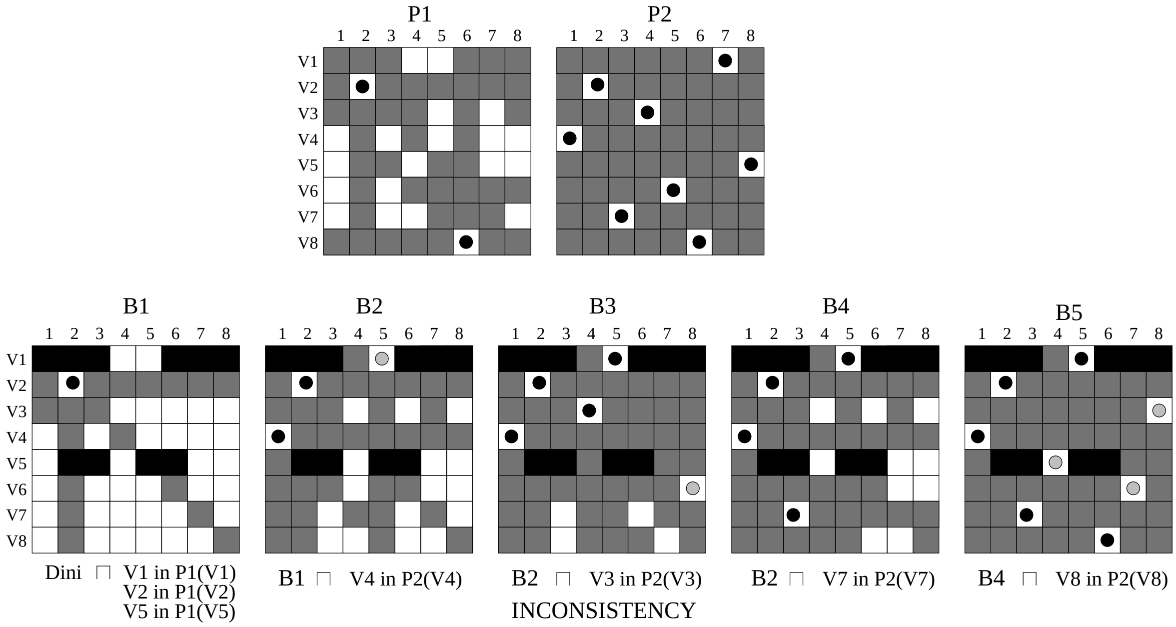

Figure 17: Example of crossover of two chromosomes for the valued 8-queens problem. White squares denote allowed values. Black squares represent forbidden values because of the assignment of the domains from parent P1. Grey squares represent values removed because of arc-consistency enforcement. Circles correspond to singleton variables. Black circles are assignments, while gray ones denote that the value is the result of the current arc-consistency enforcement.

has been designed not just to allow exploration but it also performs local search, a weak point in traditional genetic algorithms. Figure 18 shows the algorithm of the mutation operator. The role played by mutation depends on the quality of the input chromosome (function `evaluate-quality`, line 1), taking into account both the fitness and feasibility values:

- **Singleton, high fitness** (lines 2 to 8, Figure 18)
  The purpose of the mutation operator in this case is to perform a fine tuning towards the optimal solution. A new chromosome is obtained by means of a *local search* around the input chromosome, which will be replaced by the new one if and only if it is a better fit chromosome (lines 5 to 8). The local search is performed in two steps:

  1. A new AC-Subspace is generated from the input chromosome by means of the `AC-enlarge` function (line 3). The intended purpose of `AC-enlarge` is to generate a non-singleton AC-Subspace *around* a singleton input chromosome. Figure 19 shows the `AC-enlarge` al-

```
function AC-mutation(ACSS, V, C):  AC-Subspace
begin
  switch evaluate-quality(ACSS) of
    case singleton, high fitness:  /* local tuning */
      ACSS_enlarged := AC-enlarge(V, C, ACSS, D_ini)
      NewACSS := AC-random-search(V, C, ACSS_enlarged)
      if NewACSS better-than ACSS then
        return NewACSS
      else return ACSS
      end-if
    case singleton, low fitness:  /* enlarge it */
      return AC-enlarge(V, C, ACSS)
    case non-singleton, high fitness:  /* try to make it singleton */
      return AC-random-search(V, C, ACSS)
    case non-singleton, low fitness:  /* replace it */
      return AC-random-search(V, C, D_ini)
  end-switch
end
```

Figure 18: AC-Mutation

```
function AC-enlarge (V, C, ACSS, D_ini) :  AC-Subspace
begin
  ACSS_aux := D_ini
  p[1..n] := random-permutation-between(1, n)
  repeat
    NewACSS := ACSS_aux
    ACSS_aux := Arc-Consistency (V, C, NewACSS ⊓ v_p[i] in ACSS(v_p[i]) )
  until singleton(ACSS_aux)
  return NewACSS
end
```

Figure 19: `AC-enlarge` *enlarges* a solution (singleton AC-Subspace) returning a non-singleton AC-Subspace.

gorithm. Variables are processed one by one in a randomly chosen order. Each variable takes its value from the input chromosome and then, arc-consistency is enforced. This process continues until a singleton AC-Subspace is obtained. The result is the last non-singleton AC-Subspace generated.

2. Next, a local random search (line 4) is performed within the new non-singleton AC-Subspace in order to obtain a new singleton chromosome. This is achieved with the `AC-random-search` function, previously presented. In this case, the input search space is not the whole search space but the AC-Subspace obtained in step 1.

- **Singleton, low fitness** (lines 9-10)
  Now, the intended purpose of the mutation operator is to replace the low-fit chromosome by a new one that will allow the exploration of new areas of the search space. Therefore, a new chromosome is generated using the `AC-enlarge` function, taking the old chromosome as input.

- **Non-singleton, high fitness** (lines 11-12)
  The input chromosome is a promising one, but it is not singleton, so there is no guarantee that a solution lies in it. The mutation operator tries to increase its feasibility by means of the `AC-random-search` function.

- **Non-singleton, low fitness** (lines 13-14)
  A new chromosome is generated with the same method used for the initial population, in order to allow the exploration of new areas of the search space.

The function `evaluate-quality` checks whether the chromosome is singleton and decides if it is of *low* or *high* fitness. A singleton (resp. non-singleton) chromosome will be classified as *highly fit* if its fitness value is above the average fitness of singleton (resp. non-singleton) chromosomes in the current population.

Figure 20 shows an example of mutation, for the valued 8-queens problem, in the case of a singleton chromosome with high fitness, that is, the random local search. Board B represents the input chromosome, which has a fitness value of 147. In the first step, the `AC-enlarge` algorithm is used to generate a non-singleton chromosome around the input one. The order of processing variables, randomly chosen, is: $v_1, v_5, v_3, v_6, v_4, v_7, v_2, v_8$. Variable v_1 is assigned the corresponding value from the input chromosome (represented in board B) and arc-consistency is enforced, obtaining board B1. Analogously, v_5 is assigned value $\{4\}$ (board B2), and v_3 is assigned $\{8\}$ (board B3). Arc-consistency enforcement after assigning value $\{1\}$ to variable v_6 leads to a singleton chromosome (board B4), therefore the final result of `AC-enlarge` is the last non-singleton chromosome, that is, board B3. The second step of mutation consists of an `AC-random-search` that takes as input the result of the first step. A new random order of processing variables is generated: $v_8, v_7, v_3, v_4, v_1, v_2, v_5, v_6$. Variable v_8 is assigned value $\{6\}$, randomly chosen from the domain of v_8 in board B3 ($\{2, 5, 6\}$), which leads to inconsistency, so board B3 is kept. Similarly, variable

v_7 is assigned value $\{5\}$, obtaining the final mutated chromosome, represented in board B6, which has a fitness value of 181.

4 Parallel Execution Model

This section presents a parallel execution model of the hybridization introduced in the previous section, for a distributed memory multiprocessor, There are three main approaches to parallelize an evolution program [18, 7]:

- Global parallelization [16, 1, 21, 8]. Selection and mating consider all the individuals in the population, but the application of genetic operators and/or the evaluation of individuals is performed in parallel.
- Subpopulation model [19, 36, 4]. The population is divided into multiple subpopulations or demes that evolve isolated, exchanging individuals occasionally.
- Hybrid algorithms [26, 3], which combine both approaches.

We have adopted a global parallelization approach because of the following reasons:

- Properties, behavior, and theoretical fundamentals of the sequential algorithm are preserved.
- Crossover is coarse-grained, as it implies searching in a discrete search space performing arc-consistency.
- The higher communication rate of a global parallelization versus other approaches does not significantly penalize speedup, since modern distributed memory multiprocessors provide fast, low-latency asynchronous read/write access to remote processors' memory, avoiding rendezvous overhead.

A global parallelization of the presented constrained optimization evolution program is expected to achieve high speedups, since arc-consistency leads to coarse-grained genetic operators. Global parallelization implies a centralized population. Shared memory architectures support a straight implementation of this approach, whereas distributed memory architectures may suffer from communication overhead. We propose a global parallelization model for a distributed memory architecture based on a virtual centralized population, physically distributed among the processors in order to reduce communications. Target architecture is any modern distributed memory multiprocessor that allows fast asynchronous read/write access to remote processors' memory. This feature places them in a middle point between traditional shared and distributed memory architectures.

The data distribution model, shown in Figure 21, can be summarized as follows:

- The population is distributed among the processors in the system. Each processor owns a subset of the population and a local *localization table* indicating a processor where non-local chromosomes can be found.
- One processor of the system is distinguished as *master*. This processor behaves as any other, but it is also in charge of the sequential part of the algorithm, and keeps the shared *operation table*.
- The *master* produces the *operation table*, which reflects chromosomes selected to survive, to be mutated, and to take part in crossover (global mating). The operation table is broadcasted at the beginning of every generation, so each processor has a local copy of it.
- Genetic operations are performed in parallel. Coordination is achieved by means of atomic test & swap on the master processor's operation table. A processor may need to fetch (asynchronously) a chromosome from a remote processor's memory in order to perform the selected genetic operation.

At the beginning of each generation, each processor owns a subset of the population formed by:

- chromosomes generated by itself in the previous generation.
- chromosomes from the previous population fetched from a remote processor but not replaced in the current population (steady-state approach). Therefore, a chromosome may be present at many processors.

Figure 22 shows the algorithm executed in each processor. Initially, a subset of the population is generated (line 1). Every time a new chromosome is generated, its evaluation (fitness and feasible values) are asynchronously written to the master's memory. Lines 2 to 14 enclose the main loop; each iteration produces a new generation. Synchronization is needed at the beginning of each generation (line 3), in order to perform the global mating. The master establishes the genetic operations in order to generate the next population (line 5), filling the operation table, which is broadcasted to every processor (line 7). The loop in lines 8 to 12 performs genetic operations (crossover or mutation) until there are no more left. A processor may perform any of the pending operations (line 10), so it may need to fetch chromosomes from a remote processors' memory (line 9). The resulting offspring is kept in local memory, but the evaluation values are asynchronously written to master's memory (line 11).

Scheduling of pending operations is performed in a dynamic self-guided way, following a set of rules to minimize the number of chromosomes to be fetched from remote processors. Function `Fetch-Operation` (line 8), consults the local copy of the operation table and the localization table, choosing an operation to perform. In order to minimize the need to fetch remote chromosomes, the local operation table is scanned selecting operations in the following order:

1. Crossover of two local chromosomes.

2. Mutation of a local chromosome.

3. Crossover of a local chromosome with a remote one.

4. Mutation or crossover of remote chromosomes.

Once an operation is selected, the corresponding entry of the shared operation table is tested and updated in mutual exclusion. If the selected operation has already been performed by another processor, the local operation table is updated, and another operation is chosen. Otherwise, the processor writes its unique processor number in the shared operation table. Once every operation has been performed, local copies of the operation table reflect which processor has generated the new chromosomes, allowing the proper update of the localization table (line 13) for the next generation, discarding local copies of outdated chromosomes.

Figure 23 illustrates an example of operation fetching. Processor 1 (P1) selects in the first place the crossover operation (XV) that will replace chromosome number 3 (field index), because both parents (chromosomes 3 and 8) are in its local memory. P1 successfully tests and writes its processor number in the shared operation table. P2 behaves similarly with respect to operation 2. Once P1 has finished the crossover operation, it proceeds to select operation 1, as it owns one of the involved parents, writing its number in the shared operation table. P2 also tries to fetch operation 1, but it finds that operation has been already selected by processor 1, so P2 updates its local operation table and proceeds to select a new operation.

Figure 24 shows the time diagram for a generation. There is a sequential time fraction due to the generation (T_s) and broadcasting (T_{bop}) of the operation table. The time to perform a crossover in parallel (T_{xvp}) is the sequential time (T_{xvs}), increased with the time to select an operation (T_{fo}), the time to fetch the parents (T_{fp}) (only if necessary), and the time to write the evaluation values in master's memory (T_{we}). The policy to select the next genetic operator to apply favors choosing operations among local chromosomes, therefore it is expected to frequently avoid the overhead due to T_{fp}. The dynamic self-guided scheduling of the algorithm balances work load, minimizing idle time T_{id}, introduced by the necessary synchronization between generations.

Linear speedups will be obtained if communication overhead —T_{fo}, T_{we}, T_{bop}, and T_{fp}— is much smaller that genetic operation granularity (T_{xvs}), and when the number of genetic operations per generation is much greater than the number of processors. In this situation, T_{id} and T_s are much smaller than T_p.

5 Experimental Results

This section presents the empirical results obtained with our system PCSOS (Parallel Constrained Stochastic Optimization System) [38], a constraint programming environment that implements the presented work. Subsection 5.1 presents results obtained for the valued n-queens problem in order to discuss the sequential behavior of the system. This first part of the experiments were carried out on a Pentium 200 MHz under Linux.

Subsection 5.2 reports the evaluation of the parallel version of PCSOS using a complex constrained optimization problem arising in hardware design, the *channel routing problem* problem. Experiments have been carried out on a CRAY T3E, a distributed memory multiprocessor. Processing elements are connected by a bi-directional 3-D torus network achieving communication rates of 480 Mbytes/s. Parallel programming capabilities are extended through the Cray Shared Memory Library, which allows fast asynchronous read/write access to remote processors' memory.

5.1 The Valued n-queens Problem

In this subsection we compare the sequential performance of the presented approach versus a depth first branch & bound (B & B) algorithm, the usual optimization method used by constraint programming languages [29]. We report the results obtained for four different sizes of the board: 16, 32, 48, and 64. In each board, the value assigned to each of the $n \times n$ locations is a randomly generated integer between 0 and 31. Recall that the objective function is defined as the sum of the values of the board locations occupied by queens.

Table 2 shows, for each board size, the value of the best solution found for five different elapsed times. Ten runs were carried out for each board size, using different random seeds. We compare the average of the ten runs versus the results of the B&B algorithm, also implemented as part of the PCSOS system. For $n = 16$, the B&B algorithm founds the best solution in 88.6 seconds and proves its optimality in 98.6 seconds. However, for larger boards, an exhaustive search technique as B&B becomes an infeasible approach to solve this problem, whereas our approach is able to find good solutions within a reasonable time. It must be also noticed that the relatively low values of the standard deviation indicate that our method is highly robust.

5.2 The Channel Routing Problem

PCSOS parallel performance has been evaluated using a set of typical constrained optimization problems that is suitable to be modeled with constraints over finite domains. In this subsection we report the performance of the system for a VLSI design problem, the *channel routing problem*.

5.2.1 Benchmark Description

The channel routing problem [24] is a particular kind of the interconnection routing problem, which is one of the major tasks in the physical design of very large-scale integration (VLSI) circuits. The routing area is restricted to a rectangular channel. A channel consists of two parallel horizontal rows with numbered pins. Pins that belong to the same net are connected together subject to a set of routing constraints. The channel routing problem is to find routing

Table 2: Values of the best solution found versus different elapsed CPU times for the valued n-queens problem. First column indicates the size of the board. Second column corresponds to the CPU elapsed time in seconds. Third column shows the results obtained with the branch and bound algorithm. Last two columns, labeled with E.P. and Dev. resp., show the average and standard deviation of the results obtained with our approach for ten runs.

Size	Time	B&B	E.P.	Dev.
16	20	315	359	13.3
	40	355	379	16.2
	60	378	387	12.1
	80	398	392	11.6
	100	412	398	12.3
32	300	603	748	27.8
	600	609	802	20.0
	900	612	828	17.1
	1200	613	839	12.9
	1500	623	847	14.7
48	700	867	970	48.9
	1400	889	1056	46.3
	2100	898	1144	18.4
	2800	912	1181	17.3
	3500	912	1198	21.3
64	800	1075	1215	54.9
	1600	1102	1367	40.0
	2400	1140	1449	43.0
	3200	1157	1499	42.4
	4000	1157	1539	33.5

paths for a given set of nets in a given channel such that no segments overlap each other, and the routing area and the total length of routing paths are minimized. There are different approaches to the problem that impose different restrictions on the channel and routing paths. In this paper we consider the dogleg-free multilayer channel routing problem [47], which imposes the following three restrictions:

- The routing area in a channel is divided into several pairs of layers, one called a horizontal layer and the other a vertical layer. There are a fixed number of tracks in each horizontal layer.
- The routing path for every net consists of only one horizontal segment which is parallel to the two rows of the channel, and several vertical segments which are perpendicular to the two rows. Horizontal segments are placed only in horizontal layers and vertical segments are placed only in vertical layers. The ends of segments in a routing path are connected through via holes.
- No routing path can stretch over more than one pair of layers. Thus, for each net, we only need to determine the horizontal layer and the track for the horizontal segment. The positions for the vertical segments are determined directly after the horizontal segment is fixed.

The channel routing problem has been studied extensively in the VLSI design community. LaPaugh [24] proved that the problem is NP complete. Many algorithms were initially proposed for the problem [46, 15], most of them graph-based. More recently, neural network [41], simulated annealing [5], genetic algorithms [27], and constraint logic programming [47] have been proposed.

The above formulation can be stated as a constrained optimization problem over finite domains [47]. Each net is to be assigned a pair layer/track where the horizontal segment is to be placed. Let L be the number of layers and let T be the number of tracks in each layer. Each net is associated a domain variable v whose initial domain ranges from 0 to $(L \times T) - 1$, indicating the global position of the track where the horizontal segment is to be placed. The layer is v_i *div* T, where *div* stand for integer division, and the track number within the layer is v_i *mod* T, where *mod* stands for the remainder operation.

Given this problem representation, the set of domain variables $\mathcal{V}$ must satisfy the following two sets of constraints:

- *Horizontal constraints*, to avoid overlapping of horizontal segments. These constraints are straight forward to model: for each pair of nets n_i and n_j, if the rightmost terminal of n_i is equal or greater than the leftmost terminal of n_j, nets n_i and n_j cannot be assigned the same global track, that is, $v_i \neq v_j$.
- *Vertical constraints*, to avoid overlapping vertical segments. Modeling vertical constraints require a conditional precedence relation of tracks: for each terminal i, let net_{top} be the net where top terminal i belongs to, and

let net_{bot} be the net where bottom terminal i belongs to. If both nets are placed in the same layer, the track assigned to net_{top} must be grater than the track assigned to net_{bot}.

The benchmark suite given in [46] is extensively used in the VLSI design community. A representative component of the suite is the *Deutch's difficult problem*. The problem is to route a set of 72 nets on a channel where there are 174 terminals on each row. There are 117 vertical constraints and 846 horizontal constraints. The objective function to be minimized is the total length of the routing path, that is, the sum of the lengths of each routing path, for a given number of tracks per layer.

5.2.2 Performance of Sequential Execution

The channel routing problem has been tested for 2 layers/11 tracks, 3 layers/7 tracks, and 4 layers/5 tracks. For each number of layers/tracks five different settings of the parameters (population size, number of generations, and ratio of population replaced with offprings) were tried. In all settings the ***mutation_rate*** was set to 0.1, w_{fit} to 0.5, w_{fea} to 0, and ***fea_weight(t)*** increases linearly from 0.4 to 0.8. All five settings yield the same number of crossovers. For each setting the program was run ten times with a different initial random seed on the CRAY T3E making use of just one processor. Figure 25 shows the improvement of the solution quality (length of the routing path) versus the number of crossovers for the 4 layers/5 tracks version.

Setting (e) shows the worst behavior. It seems that the large population of this setting leads to replications and to a slow evolution. Setting (b) is also slow to improve the quality of the solutions due to the low percentage of crossover. (c) is the setting showing the best performance. The population size is large enough to have a wide variety of schematas, and every individual in the population is replaced every generation. Figure 26 shows the best solutions found 4 layers/5 tracks.

5.2.3 Performance of Parallel Execution

We have investigated the impact of the number of processors used by PCSOS on the time to reach solutions and on their quality, that is, the system speedup. The speedup of a parallel system for a given program is defined as the ratio of the parallel execution time and the sequential time.

The speedup obtained with PCSOS is almost linear in all cases. Figure 27 shows speedup obtained for five different parameter settings, all of them leading to the same number of genetic operations. Each reported result is the average of ten executions with a different random seed. Since the speedup obtained is linear, we can conclude that the times due to communication overhead (T_{fo}, T_{fp} and T_{we}), described in Figure 24, are negligible in comparison with the time to perform a crossover T_{xvs}. A lower number of crossovers per generation (small population and/or low crossover ratio) implies a higher sequential fraction and a higher idle time, thus reducing speedup. As expected, the settings scaling better, (a), (c) and (e) are those with both higher population size and ratio of population replaced.

A particular issue of the model affecting the speedup –the ratio of chromosomes fetched from a remote processor– has also been studied. Figure 28 illustrates the efficiency of the policy for selecting genetic operations, displaying the percentage of chromosomes that had to be fetched from a remote processor versus the number of processors. Solid lines correspond to the self-guided scheduling using the minimization rules described in Section 4. Dotted lines correspond to selecting the first pending genetic operation. Minimization rules divides by two the percentage of chromosomes to be fetched from a remote processor.

We have also investigated the relation between the quality of the solution found and the time required to find it, for different numbers of processors. Figure 29 shows the quality of the solution found versus the number of processors, for a fixed amount of time. Measurements have been taken with parameters setting (a) of Figure 27. For a fixed elapsed time, the solution quality increases with the number of processors, though with the characteristic oscillation of stochastic computations. Alternatively, time to obtain a certain solution quality decreases with the number of processors.

6 Conclusions

We have presented a method for solving constrained optimization problems over finite domains based on the integration of constraint-solving techniques and evolution programs. Constraint programming languages allow a declarative problem statement, while evolution programs are a useful optimization method for large discrete search spaces. The input to the evolution program is a constraint graph generated by a program written in a constraint programming language, thus the method is problem-independent, as long as the problem is suited to be modeled in a constraint programming language over finite domains. Chromosomes are arc-consistent subspaces of the search space of the problem, and genetic operators generate new arc-consistent subspaces, based on an arc-consistency algorithm. Arc-consistency guarantees that singleton arc-consistent subspaces are solutions to the constrained problem. A dual evaluation of chromosomes is performed: besides the usual fitness value, a *feasibility* value, computed from the size of AC-Subspace, indicates how far is a chromosome from being a solution. Selection mechanisms exploit the advantages of this dual evaluation.

The hybridization model is appropriate for a global parallelization scheme due to the coarse-grain of genetic operations. We have designed a global parallelization model based on a virtual centralized population, physically distributed among the processors in order to reduce communications. A target architecture is any distributed memory multiprocessor, such as the CRAY T3E, on which our system PCSOS has been developed. A complex constrained optimization problem over finite integer domains, coming from the field of hardware design, has been used to test the efficiency of the system. Linear speedups have been obtained when increasing the number of processors, showing that communication overhead is negligible versus the execution time of genetic operators.

Acknowledgment

This work has been supported by TIC 98-0445-C03-02 and XXXXX projects. Useful comments from anonymous referees are greatly acknowledged. The authors also thank the editor D.B. Fogel, for the careful reading and comments that improved the presentation of this paper.

References

[1] Abramson, D., Mills, G., Perkins, S. Parallelisation of a genetic algorithm for the computation of efficient train schedules. Proc. of the Parallel Computing and Transputers Conferences, 139-149, (1993).

[2] Bck, T. Hammel, H., Schvefel H.p.: Evolutionary Computation: Comments on the history and current state. IEEE trans. on Evolutionary computation, vol. 1, no 1, pp. 3-17, (1997).

[3] Bianchini, R, Brown, C.M. Parallel genetic algorithms on distributed-memory architectures. Transputer research and applications, 6, IOS Press, 67-82, (1993).

[4] Belding, T.C. The distributed genetic algorithm revisited. Proc. of the 6td Int. Conf. on Genetic Algorithms, 114-121 (1995).

[5] Brouwer, R.J., Banerjee, P.: Simulated annealing algorithm for channel routing on a hypercube multiprocessor. IEEE Int. Conf. on Computing Design. (1988) 4-7.

[6] Bowen, J., Dozier, G.: Solving Constraint Satisfaction Problems Using a Genetic / Systematic Search Hybride That Realizes When to Quit. Proc. of the Int. Conf. on Genetic Algorithms. Morgan Kaufmann (1995). 122-129.

[7] Cantú-Paz, E. A survey of parallel genetic algorithms. Calculateurs Paralleles, Reseaux et Systems Repartis. Vol. 10, No. 2. 141-171.

[8] Cantú-Paz, E., Goldberg, D.E. Modeling speedups of idealized bounding cases of parallel genetic algorithms. Proc. of the 7th Int. Conf. on Genetic Algorithms. (1997).

[9] Carlsson M., Ottosson G., Carlson B.: An Open-Ended Finite Domain Constraint Solver. Proc. Programming Languages: Implementations, Logics, and Programs, (1997).

[10] J. Beasley and P. Chu. A Genetic Algorithm for the Set Covering Problem. European Journal of Operational Research, Volume 94, No 2 (1996), 393-404 .

[11] Dincbas,M., Van Henteryck,P., Simmons,H., Aggoun,A.: The Constraint Programming Language CHIP. Proc. of the 2nd Int. Conf. on 5th Generation Computer Systems. (1988), 249-264.

[12] Dozier, G., Bowen, J., Homaifar A.: Solving Constraint Satisfaction Problems Using Hybrid Evolutionary Search. IEEE Trans. on Evolutionary Computation, Vol. 2, No 1. (1998) 23-33.

[13] Eiben A.E., Raué P.E., Ruttkay Z.: Constrained problems. Practical Handbook of Genetic Algorithms. L. Chambers, ed. CRC Press (1995). 307-365.

[14] Eiben A.E., Ruttkay Z.: Self-adaptivity for Constraint Satisfaction: Learning Penalty Functions. Proc. of the 3th IEEE Conf. on Evolutionary Computation (1996). 258-261.

[15] Fang,S.C., Feng,W.S., Lee,S.L.: A new efficient approach to multilayer channel routing problem. 29th ACM-IEEE Design Automation Conf. (1992), 579-584.

[16] Fogarty, T.C., Huang, R. Implementing the genetic algorithm on transputer based parallel processing systems. Parallel Problem Solving from Nature. 145-149 (1991).

[17] Goldberg, D.E.: Genetic Algorithms in Search, Optimization & Machine Learning. Addison-Wesley Publishing Co. Inc. Reading (1989).

[18] Grefenstette, J.J. Parallel adaptive algorithms for function optimisation. Tech. Rep. No. CS-81-19. Nashville, TN: Vanderbilt University Computer Science Department. (1981)

[19] Grosso, P.B. Computer simulations of genetic adaptation: Parallel subcomponent interaction in a multilocus model. Doctoral dissertation, The University of Michigan. (1985).

[20] Homaifar, A., Lai, S.H., Qi, X.: Constrained optimisation by genetic algorithms. Simulation, 62(4) (1994) 242-254.

[21] Hauser, R., Männer, R. Implementation of standard genetic algorithm on MIMD machines. Parallel Problem Solving from Nature. 504-513 (1994).

[22] Ilog Solver ++. Reference Manual. Ilog Inc. 1901 Landings Drive, Mountanview, CA. (1994)

[23] Joines, J., Houck C.: On the use of non-stationary penalty functions to solve non-linear constrained optimisation problems with gas. Proc. of the 1st IEEE Int. Conf. on Evolutionary Computation. Piscatway N.Y. IEEE Press (1994) 579-584.

[24] LaPaugh, A.S.: Algorithms for integrated circuits layouts: an analytical approach. PhD Dissertation, MIT Lab. of Computer Science (1980).

[25] Lawler, E., Wood, D.W.: Branch-and-bound methods: A survey. Operations Research, vol. 14 (1966) 339-356.

[26] Lin, S.C., Goodman, E.D., Punch, W.F. Investigating parallel genetic algorithms on job shop scheduling problems. Proc. of 6th Int. Conf. on Evolutionary Programming, 383-393, (1997)

[27] Liu, X., Sakamoto, A., Shimamoto, T.: Genetic channel routing. IEICE Trans. Fundamentals (1994) 492-501.

[28] Marchiori E.: Combining constraint processing and genetic algorithms for constraint satisfaction Problems. Proc. of 7th Int. Conf. on Genetic Algorithms. Morgan Kaufmann (1997). 330-337.

[29] Marriot, K., Stuckey, J.S.: Programming with Constraints: An Introduction. The MIT Press. (1998).

[30] Michalewicz, Z.: Genetic algorithms + Data Structures = Evolution Programs. 3rd Edition, Springer-Verlag (1996).

[31] Michalewicz, Z., Nazhiyath, G.: Genocop III. A co-evolutionary algorithm for numerical optimisation problems with non-linear constraints. Proc. of the 2nd IEEE Int. Conf. on Evolutionary Computation. NY. IEEE Press (1995) 647-651.

[32] Minton, D., Johnston, M.D., Philips, A.B., Laird, P.: Minimising conflicts: a Heuristic Repair Method for constraint satisfaction and scheduling problems. Artificial Intelligence 58 (1992).

[33] Mohr, R., Henderson, T.C.: Arc and path consistency revisited. Artificial Intelligence 28 (1996) 225-233.

[34] Paredis, J.: Genetic State-Search for constrained Optimisation Problems. 13th Int. Joint Conf. on Artificial Intelligence (1993).

[35] Paredis, J.: Co-evolutionary Computation. Artificial Life, 2(4), (1995). 355-375.

[36] Pettey C.B., Leuze, M.R., Grefenstette, J.J. A parallel genetic algorithm. Proc. of 2nd Int. Conf. on Genetic Algorithms, (1987). 155-161.

[37] Puget, J.F.: A C++ implementation of CLP. Proc. of Singapore International Conference on Intelligent Systems 94. (1994).

[38] Ruiz-Andino, A.: CSOS User's manual. Tech. Report No. 73.98. Department of Computer Science. Universidad Complutense de Madrid, (1998).

[39] Riff-Rojas, M.C.: Using knowledge of the constraint network to design an evolutionary algorithm that solves CSP. Proc. of the 4th IEEE Conf. on Evolutionary Computation. IEEE Press (1997). 279-284.

[40] Schnecke, V., Vornberger, O.: Hybrid Genetic Algorithms for Constraines Placement Problems. IEE Trans. on Evolutionary Computation, Vol. 1, No 4. (1997) 266-277.

[41] Takefuji: Neural network parallel computing. Kluwer Academic Press (1992).

[42] Tsang, E.: Foundations of constraint satisfaction. Academic Press (1993).

[43] Van Hentenryck P., Deville, Y., Teng C.M.: A generic Arc-consistency Algorithm and its Specialisations. Artificial Intelligence 57 (1992) 291-321.

[44] Van Hentenryck P.: The OPL Optimization Programming Language. The MIT Press (1998).

[45] Wallace, M.: Constraints in Planing, Scheduling and Placement Problems. Constraint Programming, Springer-Verlag (1994).

[46] Yoshimura, T., Kuh, E.S.: Efficient algorithms for channel routing. IEEE Trans. on CAD. (1982) 25-35.

[47] Zhou, N.F.: Channel Routing with Constraint Logic Programming and Delay. 9th Int. Conf. on Industrial Applications of Artificial Intelligence. Gordon and Breach. (1996) 217-231.

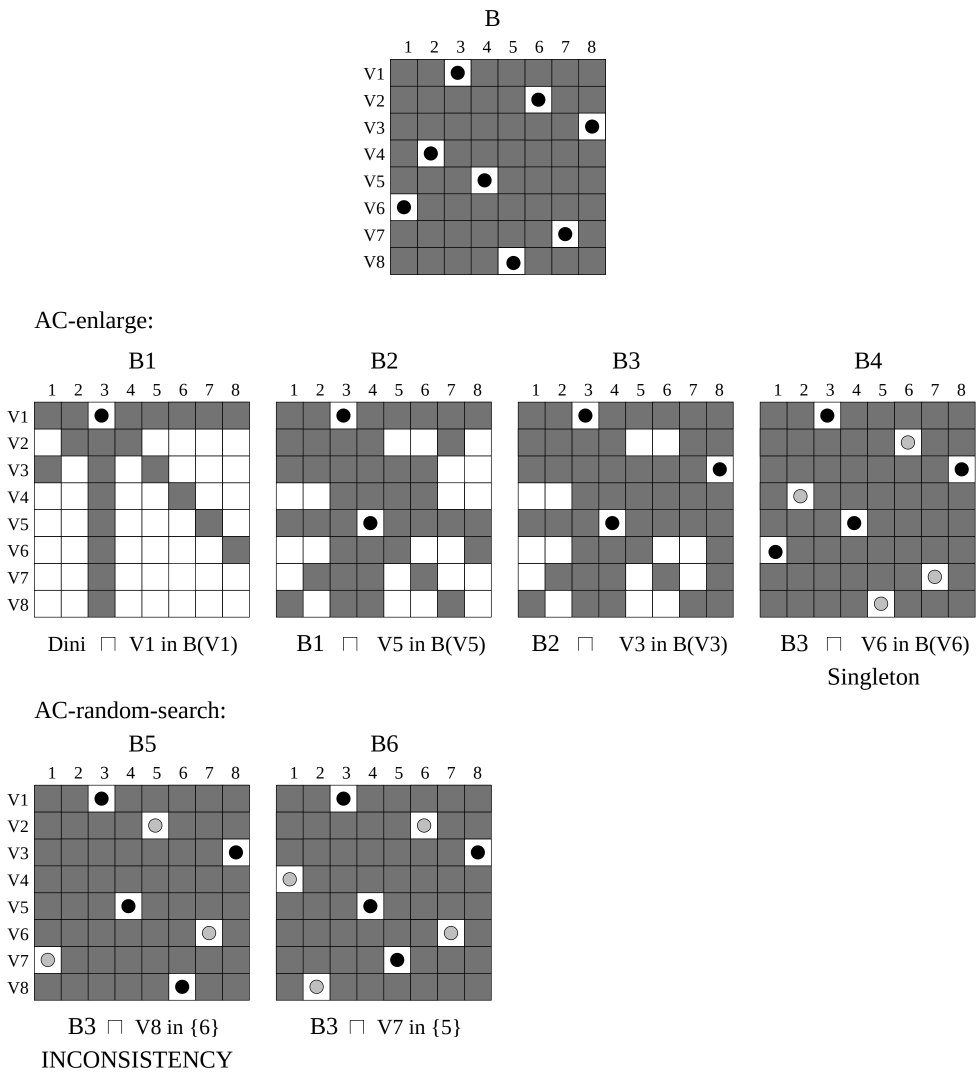

Figure 20: Example of mutation for the valued 8-queens problem. White squares denote allowed values, while gray ones denote forbidden values. Circles correspond to singleton variables. Black circles are assignments, while gray ones denote that the value is the result of the current arc-consistency enforcement.

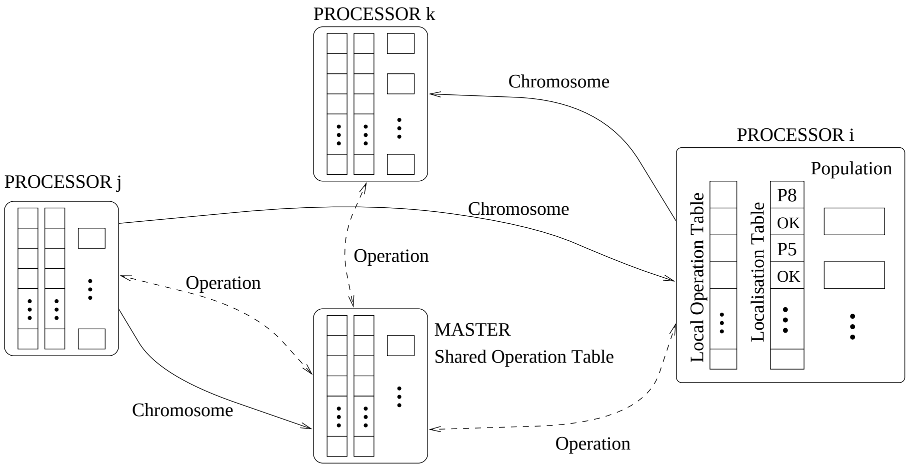

Figure 21: Data distribution model. Solid arrows represent fetching a remote chromosome. Dotted arrows represent mutual exclusion access to the shared operation table.

```
Procedure Parallel-AC-Evolution
begin
  Generate a subset of the initial population
  while not termination() do
     Synchronization
     if I-am-the-Master then
        Generate-Matings-and-Mutations(Operation-Table)
     end-if
     Broadcasting/Reception of Operation-Table
     while Fetch-Operation(Operation-Table, Localization-Table) do
        Fetch parents, if necessary, updating Localization-Table
        Perform AC-Crossover (or AC-Mutation)
        Write fitness-feasible to Master
     end-while
     Update(Localization-Table)
  end-while
end
```

Figure 22: Parallel constrained evolution program.

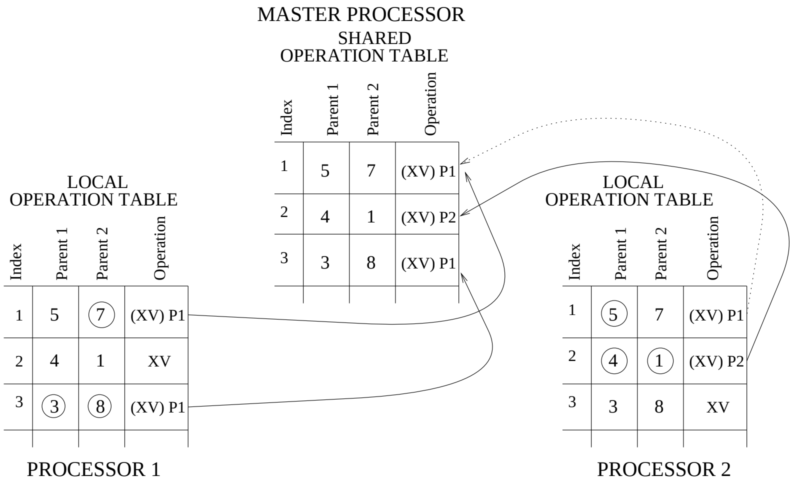

Figure 23: Fetch Operation example. Circled numbers denote local chromosomes. Abbreviation (genetic operation to perform) in brackets denotes initial value. Solid arrows denote successful test & write operations. Dotted arrows denote unsuccessful test & write operations on the shared operation table.

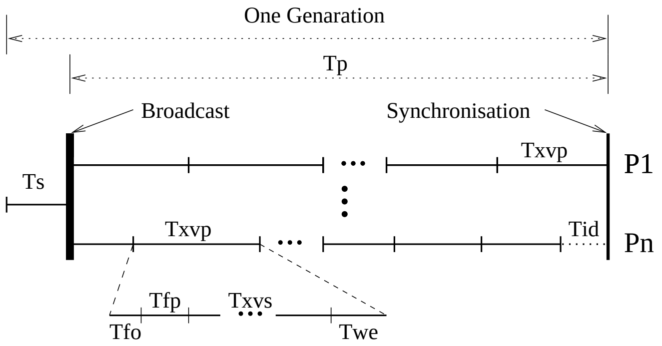

Figure 24: Time diagram for a generation. T_s = sequential fraction (global selection for mating). T_{bop} = broadcast of operation table. T_{xvs} = sequential genetic operation. T_{xvp} = parallel genetic operation. T_{fo} = fetch an operation from master. T_{fp} = fetch remote parents. T_{we} = write evaluation values to master. T_{id} = waiting for synchronization. T_p = parallel execution of genetic operators.

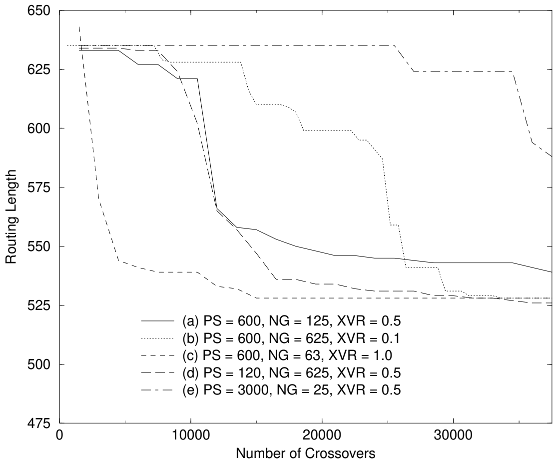

Figure 25: Performance of PCSOS with one processor for different parameter settings. PS = Population size. NG = No. of generations. XVR = Ratio of population replaced with offsprings.

Figure 26: Best routing found for the 4 layers/5 tracks instance.

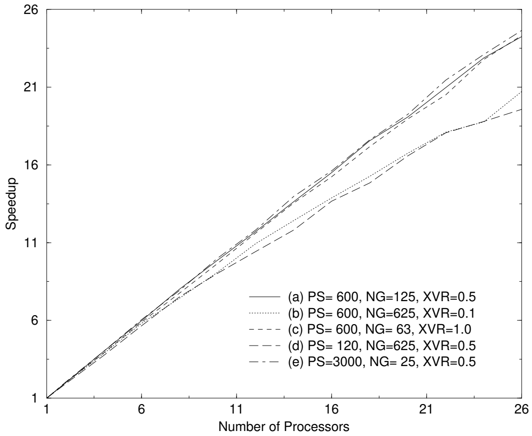

Figure 27: Speedup obtained for different parameter settings. PS = Population size. NG = No. of generations. XVR = Ratio of population replaced with offsprings.

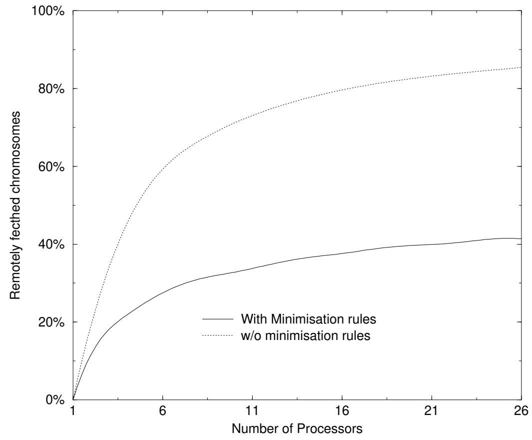

Figure 28: Percentage of chromosomes fetched from a remote processor, with and without the minimization policy.

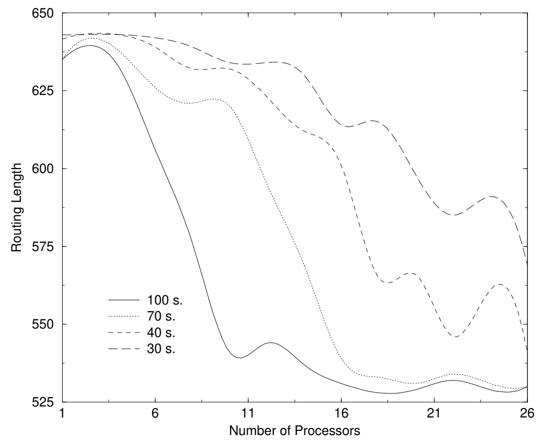

Figure 29: Quality of solution vs. number of processors for different fixed elapsed times.