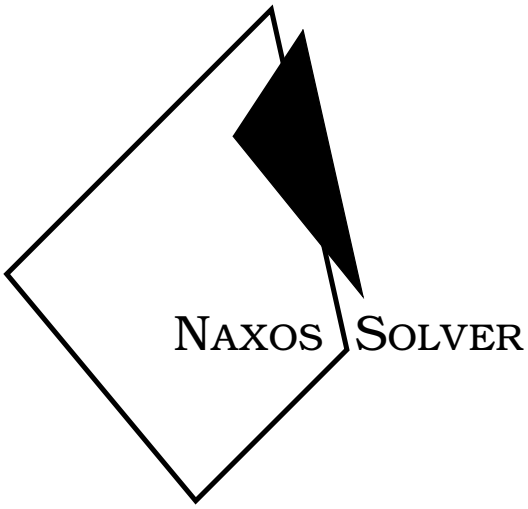


Reference Manual

Nikolaos Pothitos

May 1, 2013

Contents

1 Introduction

NAXOS SOLVER is a constraint satisfaction problems *solver*. 'NAXOS' is the name of the island[1] where the solver was built in the beginning. It is implemented and maintained by the author, under the supervision of Assistant Professor Panagiotis Stamatopoulos at the Department of Informatics and Telecommunications of the National and Kapodistrian University of Athens. The aim of this handbook is to provide all the information needed by an application developer of the library.

A *constraint satisfaction problem* (CSP) contains a set of *constrained variables,* that can be simply called *variables;* each variable corresponds to a *domain*. Constrained variables are connected to each other via a set of *constraints*. Generally speaking, a constraint that involves specific constrained variables is a set with all valid combinations of values that can be assigned. For example, if we take the variables x_1 and x_2 with domains $\{0, 1, 2, 3\}$, the equality constraint can be declared as $\mathcal{C}(\{x_1, x_2\}, \{(0, 0), (1, 1), (2, 2), (3, 3)\})$. Although this notation for the constraint is as generic as possible, in practice (i.e. in Constraint Programming) we use simple relations to describe the constraint networks. In the above example the constraint can be simply written as $x_1 = x_2$. A *solution* to a constraint satisfaction problem is a valid assignment of a value to every constraint variable, that satisfies all the constraints. Finally, it should be noted that the advantage of Constraint Programming is that it allows the separation of the problem declaration process and the solution generation mechanism.

NAXOS SOLVER is a library that solves constraint satisfaction problems, that was designed for the C++ object-oriented programming environment. The solver is threadsafe, i.e. safe to use in a multithreaded environment. 'Internal' classes, methods, functions, etc. that are not stated in this manual, should not be used by the application developer, as they may change in future. Still, to avoid misunderstandings, we should not name our own variables, classes, etc. with names that begin with '`Ns`' as this is the prefix for the solver built-in classes and constants. Finally, note that the solver does not check for any possible overflows (e.g. during the addition of two big integers) for performance reasons.

Part of the solver design and its naming conventions are influenced by

[1] Specifically, we refer to the village Coronis of Naxos.

the Standard Template Library (STL) modelling.[2] E.g. several iterators are used and implemented.

There is *no* distinction between handle-classes and implementation-classes, as in ILOG SOLVER. This distinction exists in ILOG SOLVER, because it attempts to automatically manage memory resources à la Java. In every handle-class there exists a reference to an implementation-class instance. It is possible that many handle-class instances point to the same implementation-class instance. The implementation-class instance is destructed only when all the handle-class instances that point to it are destructed. (Something similar happens with the references in Java.) Thus, in a function in ILOG SOLVER it is possible to construct automatic variables-instances in order to describe a constraint; the constraint and the variables involved will continue to exist after the function returns. In the same circumstance in NAXOS SOLVER we would have a segmentation fault.

2 Error Handling

When we write code, error handling is the first thing to take care of. In NAXOS SOLVER we should catch exceptions of type `NsException`. This class is a `logic_error` subclass, which inherits from `exception`. So it suffices to catch `exception` instances; this base class has a method `what()` that returns a string describing the error occurred.

```
#include <naxos.h>
using namespace naxos;
using namespace std;

int  main (void)
{
    try  {

        // ... CODE OF THE PROGRAM ... //

    } catch (exception& exc)  {
        cerr << exc.what() << "\n";
```

[2] B. Eckel and C. Allison. *Thinking In C++ Vol. 2: Practical Programming*. Prentice Hall, 2nd edition, 2003.

```
    } catch (...)  {
        cerr << "Unknown exception" << "\n";
    }
}
```

It is not a good programming practice to to use exceptions inside the algorithms body. In most cases, exceptions should 'wrap' our programs.

3 Constrained Variables

The solver supports *finite domain integer constrained variables*. The class that implements them is called `NsIntVar` and contains the following methods.

`NsIntVar(NsProblemManager& pm, NsInt min, NsInt max)` A constructor function for a constraint variable. Argument `pm` is the problem manager that the variable belongs to (see § 5). `min` and `max` are the bounds of its domain, that is also designated `[min..max]`.

Data-type `NsInt` can at least represent the integers that can be represented by data-type `long`. The minimum value that an `NsInt` can hold, equals to the constant `NsMINUS_INF` and the maximum equals to `NsPLUS_INF`. (The maximum value of the unsigned data-type `NsUInt` is `NsUPLUS_INF`.)

The minimum of a domain must be strictly greater than `NsMINUS_INF` and the maximum value must be strictly less than `NsPLUS_INF`, as those constants represent infinity, as we will see below.

`NsIntVar()` This constructor creates a variable that can be initialized afterwords by assigning an expression to it (using the overloaded operator '='), as on the third line of the following example.

```
NsIntVar  X(pm,0,3), Y(pm,-1,15), Z;
NsIntVar  W = X + 3*Y;
Z = W * W;
```

On the second line of the example, we used another constructor function, that takes an *Expression* as argument; here the expression is '`X + 3*Y`'.

`void remove(NsInt val)` Removes the value `val` from the domain of the variable.

`void remove(NsInt min, NsInt max)` Removes every value of the domain that is in the range $[\texttt{min}, \texttt{max}]$. Instead of removing those values one by one using `remove(val)`, it is more efficient to call this method.

`void removeAll()` 'Empties' the domain of the variable. Practically, the solver only informs the problem manager that an inconsistency occurred, due to an empty domain. This method is useful when we want to make search fail. E.g. when we want a goal to fail during its execution, we call this method for any variable.[3]

`void set(NsInt val)` Assigns the value `val` to the variable; thus, the variable becomes *bound* (instantiated).

The following methods do not alter the variable for which they are called.

`bool contains(NsInt val)` Returns `true` if the domain of the variable contains the value `val`.

`NsInt min()` The minimum value of the domain.

`NsInt max()` The maximum value of the domain.

`NsUInt size()` The number of the values in the domain.

`bool isBound()` Returns `true` if the variable is bound. 'Bound' is synonym to 'instantiated,' 'singleton,' and 'fixed' and means that a value has been assigned to the variable, or, in other words, that the domain contains only one value.

`NsInt value()` It is used only when the variable is bound and returns its (unique) value. If it is called for an unbound variable, an exception is thrown.

If the variable is bound, this method is equivalent to `min()` and `max()`, but we use this method for better code readability.

[3] In order to show that a goal succeeds, we make `GOAL()` return `0`; but in order to show that a goal failed, this is a—less elegant—way to do it. (For more information about the goals mechanism see § 8.)

NsInt next(NsInt val) Returns the smallest value in the domain, that is strictly greater than `val`. If such a value does not exist, the function returns `NsPLUS_INF`. The argument `val` may be equal to the special values `NsMINUS_INF` and `NsPLUS_INF` too.

NsInt previous(NsInt val) Returns the greatest value in the domain, that is strictly less than `val`. If such a value does not exist, the function returns `NsMINUS_INF`. The argument `val` may be equal to the special values `NsMINUS_INF` and `NsPLUS_INF` too.

Two iterators for the class accompanied by examples follow.

NsIntVar::const_iterator Iterates through all the values of the domain of the variable. E.g. the following code prints the values of the variable `Var`, in ascending order.

```
for (NsIntVar::const_iterator v = Var.begin();
                              v != Var.end();   ++v)
        cout << *v << "\n";
```

NsIntVar::const_reverse_iterator Iterates through the values of the domain of the variable in reverse order. E.g. the following code prints the values of the variable in descending order.

```
for (NsIntVar::const_reverse_iterator v=Var.rbegin();
                              v != Var.rend();   ++v)
        cout << *v << "\n";
```

NsIntVar::const_gap_iterator Iterates through the gaps that exist inside the domain of the variable. Mathematically speaking, it gives all the values (inside $[\texttt{min}, \texttt{max}]$) of the complement of the domain (where `min` is the minimum and `max` is the maximum of the domain). E.g.

```
for (NsIntVar::const_gap_iterator g = Var.gap_begin();
                                  g != Var.gap_end();  ++g)
        cout << *g << "\n";
```

Finally, the operator '<<' has been overloaded in order to print the variable to an output stream, by writing e.g. '`cout << Var;`'.

4 Constrained Variable Arrays

The default (flexible) array class, that contains constrained variables, is `NsIntVarArray`. The i-th variable of the array `VarArr` is accessed as usually, via `VarArr[`i`]`. The default data-type for i is `NsIndex`.

An `NsIntVarArray` is initially empty. We can insert a constrained variable to it, either in front of it, or at its back, in the way that we insert elements into a linked list. Its constructor function does not have any arguments. The description for the rest of the class functions follow.

`NsIndex size()` Returns the size of the array.

`bool empty()` Returns `true` if the array is empty.

`NsIntVar& back()` The last constrained variable of the array.

`NsIntVar& front()` The first constrained variable of the array.

`void push_back(`*Expression***`)`** Inserts at the end of the array the variable that is described by the *Expression*. In the following section it is explained that an *Expression* can be simply a constrained variable, or a combination of variables. E.g.

```
VarArr.push_back( NsIntVar(pm, 10, 30) );
VarArr.push_back( 3*VarX + VarY );
VarArr.push_back( VarX > 11 );
```

The first statement above inserts at the end of `VarArr` a new constrained variable with domain `[10..30]`.

The last statement inserts a constrained variable at the end of the array. This variable will equal `1` if it holds that `VarX > 11`, otherwise it will equal `0`. (It is possible that its domain will be `[0..1]`, if `VarX` contains some values less than `11` and some other values greater than `11`.) So we have a *meta-constraint*.

`void push_front(`*Expression***`)`** Like `push_back()` but the insertion takes place at the beginning of the array.

Iterators for arrays follow. We can use them in order to iterate easily through the variables of an array.

`NsIntVarArray::iterator` With this iterator we can access all the variables of the array in turn. E.g. the following code removes the value `2` from every variable of `VarArr`:

```
for (NsIntVarArray::iterator  V = VarArr.begin();
                          V != VarArr.end();   ++V)
        V->remove(2);
```

`NsIntVarArray::const_iterator` This is an iterator like the above one, but it is useful only when we do not modify the constrained variables. E.g. we can use it in order to print the variables of an array.

```
for (NsIntVarArray::const_iterator  V=VarArr.begin();
                                 V != VarArr.end(); ++V)
        cout << *V << "\n";
```

Finally, the operator '`<<`' has been overloaded for the arrays too. We can type an array by writing for example '`cout<<VarArr;`'.

4.1 Generic Arrays

In case we want to create an array with the philosophy and methods of `NsIntVarArray`, we can use the template class `NsDeque<`*data_type*`>`. E.g. with

```
NsDeque<int>  arr;
```

we declare that `arr` is a flexible array of integers, initially empty. With

```
NsDeque<int>  arr(5);
```

we declare that it initially contains `5` elements. The *data_type* does not have any sense to be `NsIntVar`, because in this case we can use directly `NsIntVarArray`, instead of `NsDeque`.[4]

[4] `NsDeque` is actually an extension of `std::deque`, which is included in C++ standard library. The difference in `NsDeque` is that it always checks that we are inside the array bounds; if we exceed them, the corresponding exception is thrown.

5 Problem Manager

Before the problem statement, we should declare a *problem manager* (class `NsProblemManager`). This manager holds all the information needed about the variables, the constraint network being built and the goals that are going to be executed. The constructor function does not have any argument. The other functions follow.

`void add(`*ExprConstr***`)`** Adds the constraint described by the constraint expression *ExprConstr* (see § 6.1). In a constraint expression we can use condition operators (<, ==, !=, etc.), or built-in expressions like `NsAllDiff()` that states that all the variables of an array must have different values. E.g.

```
pm.add( 3*VarX != VarY/2 );
pm.add( VarW == -2 || VarW >= 5 );
pm.add( NsAllDiff(VarArr) );
```

`void addGoal(NsGoal* goal)` Adds `goal` into the list with the goals that have to be executed/satisfied (see § 8).

`bool nextSolution()` Finds the next solution. The goals that have been added are going to be satisfied. If there is no solution, `false` is returned.

`void minimize(`*Expression***`)`** It gives solver the instruction to minimize the value of *Expression*. Every time that the solver finds a solution, the *Expression* will be less than the one of the previous solution (branch-and-bound algorithm). In fact, if `nextSolution()` gives a solution and the *Expression* maximum value is a, the constraint *Expression* $< a$ is imposed, for the next time `nextSolution()` is called. So, after each solution, the *Expression* gets reduced. If it cannot be further reduced, `nextSolution()` returns `false` and we should have stored somewhere the last (best) solution. E.g.

```
pm.minimize( VarX + VarY );
while (pm.nextSolution() != false)
    {  /* STORE SOLUTION */  }
```

If we wish to maximize an *Expression*, we can simply put a '–' in front of it and call `minimize(-`*Expression*`)`.

`void objectiveUpperLimit(NsInt max)` During search, this method defines an upper bound for the solution cost equal to `max`; the solution cost is expressed by the `NsProblemManager::minimize()` argument. In other words, the constraint '*cost_variable* $\leq$ `max`' is imposed.

Besides, after the beginning of search—when `nextSolution()` is called for the first time—we cannot add more constraints by calling `NsProblemManager::add`, e.g. via '`pm.add(X <= 5)`'. Only this function (`objectiveUpperLimit`) can impose such a constraint, but only onto the cost variable.

`void timeLimit(unsigned long secs)` Search will last at most `secs` seconds. After this deadline, `nextSolution()` will return `false`.

`void realTimeLimit(unsigned long secs)` It works like the previous function, but the `secs` seconds here are real time; in the previous function it was the time that CPU has spent for the solver *exclusively*.

`void backtrackLimit(unsigned long x)` If `x` is greater than zero, it makes `nextSolution()` return `false` after search has backtracked `x` times, from the moment that this function was called.

`unsigned long numFailures()` Returns the number of failures during search.

`unsigned long numBacktracks()` Returns how many times the solver has backtracked during search.

`unsigned long numGoals()` Returns how many goals have been executed.

`unsigned long numVars()` Returns the number of the constrained variables that have been created. Note that the number includes intermediate/auxiliary variables—if any—that the solver has automatically constructed.

`unsigned long numConstraints()` Returns the number of the problem constraints. Note that the number includes intermediate/auxiliary constraints—if any—that the solver has automatically created.

`unsigned long numSearchTreeNodes()` Returns the number of nodes of the search tree that the solver has already visited.

`void searchToGraphFile(char *fileName)` Stores into a file named `fileName` a representation of the search tree. The file format is called `dot` and the application Graphviz can graphically display it.

`void restart()` Restores the constrained variables (i.e. their domains) to the state they were initially, a little bit after the *first* `nextSolution()` was called. More specifically, it restores the state that existed immediately before the first `nextSolution()` call, but keeps the changes that were made in order to achieve the first arc-consistency of the constraint network (see § 8 for the arc-consistency definition). In other words, this function restores the constraint/variable network to the first arc-consistency state that took ever place (before the execution of any goal).

This method also cancels all the goals that were about to be executed. That is, if we wish to begin search (with a `nextSolution()` call) after a `restart()`, we have to declare a goal to be executed (using `addGoal()`), otherwise there is no goal!

`restart()` does *not* affect the variable that was the argument of `minimize()`—also known as *objective* or *cost variable*. That is, it does *not* restore this variable to its initial state. For example, if the objective variable had initially the domain `[0..100]` and before `restart()` had the domain `[0..10]`, then after `restart()` is called, the domain will be kept `[0..10]`.

We cannot call this function inside goals, but outside them. E.g. we can call it at the code 'level' we call `nextSolution()`.

6 Expressions

In order to connect the variables, we take advantage of the overloaded operators and we create expressions and combinations of them. A simple expression can be even a variable or an integer. An expression is designated *Expression*.

6.1 Expressions for Constraints

Expressions for constraint are denoted *ExprConstr* and they are a subcategory of the general expression category *Expression*. They are mainly used as `NsProblemManager::add()` arguments and for the creation of meta-constraints. The following are *ExprConstr*:

- *Expression*$_1$ op *Expression*$_2$, op $\in \{$<, <=, >, >=, ==, !=$\}$
- `!(` *ExprConstr* `)`
- *ExprConstr*$_1$ op *ExprConstr*$_2$, op $\in \{$`&&`, `||`$\}$
- `NsIfThen(` *ExprConstr*$_1$ `,` *ExprConstr*$_2$ `)`
- `NsEquiv(` *ExprConstr*$_1$ `,` *ExprConstr*$_2$ `)`
- `NsCount(` $VarArr$`,` $IntArr_1$`,` $IntArr_2$ `)`
- `NsAllDiff(` $VarArr$ `)`

So, the definition is recursive. The last expression means that the constrained variables inside $VarArr$ (an `NsIntVarArray`) are different between them.[5]

On the other hand, the array $VarArr$ as declared in the expression `NsCount(`$VarArr$`,`$Values$`,`$Occurrences$`)` consists of *constrained* variables, but the other two arrays, namely $Values$ and $Occurrences$, contain integers, as their type is `NsDeque<NsInt>`. The array $Values$ contains the *values* to be distributed inside $VarArr$, while the array $Occurrences$ contains how many times each value (in $Values$) appears inside $VarArr$. E.g. if $Values[i] =$ `34` and the corresponding $Occurrences[i] =$ `2`, then the value `34` will be assigned to exactly `2` constrained variables in $VarArr$.

[5] If we use the expression `NsAllDiff(`$VarArr$`,`$Capacity$`)`, where $Capacity$ is a positive integer, then there will be more constraint propagation. E.g. if $VarArr =$ {`[1..2], [1..2], [1..5]`}, then by declaring `NsAllDiff(`$VarArr$`,1)` it infers that $VarArr =$ {`[1..2],[1..2],[3..5]`}, but using the simple expression `NsAllDiff(` $VarArr$ `)` we would have a value removal only when a variable of $VarArr$ becomes assigned. However, this more powerful consistency may 'cost' in terms of time. Finally, the integer $Capacity$ is the maximum number of occurrences that a value can have inside the array $VarArr$.

The constraint `NsIfThen(`p`,`q`)` implies the logical proposition $p \Rightarrow q$, and the constraint `NsEquiv(`p`,`q`)` means $p \Leftrightarrow q$.[6]

Some examples of expressions for constraints follow:

```
VarX < VarY
!( X==Y  ||  X+Y==-3 )
(X==Y) != 1
```

6.2 General Expressions

Apart from *ExprConstr*, the following also belong to the category of general expressions *Expression*:

- *Expression*$_1$ op *Expression*$_2$, op $\in \{$`+`, `-`, `*`, `/`, `%`$\}$
- `NsAbs(` *Expression* `)`
- `NsMin(` *VarArr* `)`
- `NsMax(` *VarArr* `)`
- `NsSum(` *VarArr* `)`
- `NsSum(` *VarArr*`,` *start*`,` *length* `)`
- *IntArr*`[` *Expression* `]`

An *Expression*—except from describing a constraint—can be assigned to a variable. E.g. we can write

```
NsIntVar  X = Y + 3/Z;
NsIntVar  W = NsSum(VarArrA) - (NsMin(VarArrB) == 10);
```

Note that instead of writing `VarArr[0] + VarArr[1] + VarArr[2]`, it is more efficient to use the equivalent expression `NsSum(VarArr,0,3)`. The second and third argument of `NsSum` are respectively the position of the first element of `VarArr` that will be included in the sum and the number of the next elements that (together with the first element) will be included in the sum. If neither of these two arguments exist, then all the array is included in the sum.

[6] The two constraints can also be expressed with an equivalent way as `(` `!`p `||` q `)` and `(` p `==` q `)` respectively.

`NsSum` is the more efficient expression, because for the other expression, *an intermediate variable will be created* that will be equal to `VarArr[0] + VarArr[1]`; the variable `VarArr[2]` will be afterwards added to the intermediate variable. This intermediate variable will not be created if we use `NsSum`.

`NsAbs` gives the absolute value. `NsMin` and `NsMax` give respectively the minimum and the maximum of the array that they accept as an argument.

6.2.1 The Element Constraint

A separate paragraph for the last expression '$IntArr$`[` $Expression$ `]`' is dedicated, because it has to do with the special *element* constraint.[7] For simplicity reasons we take that the $Expression$ is simply the constrained variable $VarIndex$, that is used as an 'index' in the array of integers $IntArr$. Note that $IntArr$ is an array containing *integer* values, because it is an `NsDeque<NsInt>` instance; it does *not* contain constrained variables, as it is not an `NsIntVarArray` instance.

In order to understand the constraint usability, we will see an example. Let the array `NsDeque<NsInt> grades` contains eight students' grades. Also, let the domain of $VarIndex$ contain all the array positions, that is `[0..7]`. If we want the domain of the constrained variable $VarValue$ to contain every grade, we declare the constraint '$VarValue$ `==` `grades[`$VarIndex$`]`'.

(In case we declare another constraint, e.g. '$VarValue$ `>= 9`', the domain of $VarIndex$ will be limited in order to contain only the students' numbers whose grades are 9 or 10.)[8]

6.3 Expressions for Arrays

Finally there is a special independent expression category, that can be assigned to arrays of constrained variables (`NsIntVarArray`). It contains the following expressions for the Inverse constraint (see § 6.3.1).

- `NsInverse(` $VarArr$ `)`

[7] The name 'element' comes from logic programming.

[8] We saw that the expression '$VarValue$ `==` $IntArr$`[`$VarIndex$`]`' declares an element constraint, but the same constraint can also be declared with a logic programming style as '`NsElement(`$VarIndex$`,`$IntArr$`,`$VarValue$`)`'.

- `NsInverse(` $VarArr$`,` $maxdom$ `)`

$maxdom$ is the size of the inverse array that will be created. If this argument does not exist, it is taken that $maxdom = \max_{V \in VarArr}\{V.\max\}$. In any case, $maxdom$ should be greater or equal than this value. E.g.

```
NsIntVarArray  VarArrB = NsInverse(VarArrA);
NsIntVarArray  VarArrC;
VarArrC = NsInverse(VarArrA, 100);
```

6.3.1 The Inverse Constraint

The *Inverse* constraint is applied between two arrays of constrained variables. Let Arr be an array that contains variables with positive values in their domains. We want $ArrInv$ to be the 'inverse' array of Arr. Still, let D_x be the domain of the constrained variable x. Then it holds that:

$$\forall v \in D_{ArrInv[i]}, \quad D_{Arr[v]} \ni i \ .$$

If there is no v such that $i \in D_{Arr[v]}$, then the domain of $ArrInv[i]$ will *only* contain the special value -1.

In a simpler notation, we can write that it holds:

$$Arr[ArrInv[i]] = i \quad \text{and} \quad ArrInv[Arr[i]] = i \ .$$

That is why the constraint is called 'Inverse.' Of course, the above relations would have sense, if the variables of the two arrays were bound and if the unique value that each variable had was designated with the variable name itself. It should also apply that $\forall i$, $ArrInv[i] \neq -1$.

Usefulness of the Constraint. This constraint can be used in dual modelings of a problem. E.g. suppose that we have a number of tasks to assign to some workers. One modelling could be to have a variable for each task with the set of workers as domain. Another modelling is to have a variable for each worker with the set of tasks as domain. Obviously there exist some constraints in the problem. Some constraints may be declared more easily using the first modelling, but there may be other constraints that would be declared more easily and naturally using the second modelling.

In this case the solver gives the possibility to use both modellings. However, the variables of the two modellings are not irrelevant. We should declare something like

$$X[i] = j \iff Y[j] = i \ .$$

This is done using an Inverse constraint.

7 Examples

7.1 N-Queens Problem

A real problem will be declared as an example.

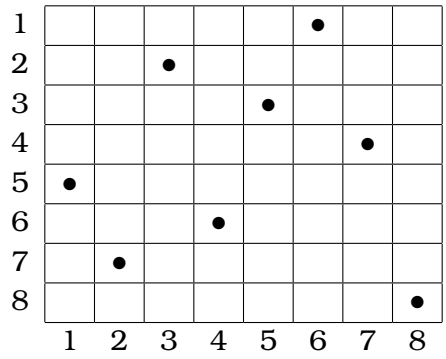


Figure 1: 8 queens that are not attacked

7.1.1 Definition

In the N queens problem we should place N queens on an $N \times N$ chessboard, so that no queen is attacked. In other words we should place N items on an $N \times N$ grid, in a way that no two items share the same line, column or diagonal. Figure 1 displays an example for $N = 8$.

So in each column $0, 1, \ldots, N-1$ we will have a queen. It remains to find out the *line* where each queen will be placed. Therefore we ask to assign values to the variables X_i with $0 \leq X_i \leq N-1$, where X_i is the line on which the queen of column i is placed.

Regarding the constraints, first of all no two queens should share the same line, i.e.

$$X_i \neq X_j, \quad \forall i \neq j \ . \tag{1}$$

They should not also share the same diagonal, consequently

$$X_i + i \neq X_j + j \quad \text{and} \quad X_i - i \neq X_j - j, \quad \forall i \neq j \ . \tag{2}$$

$X_i + i$ corresponds to the first diagonal and $X_i - i$ to the second diagonal for the queen of column i.

7.1.2 Code

In the solver code, the variables X_i are represented by the array `Var`, that according to (1) should have different elements. Concerning (2), we create two other arrays, `VarPlus` and `VarMinus`, with the elements $X_i + i$ and $X_i - i$ respectively. For these arrays we will also declare that their elements shall differ.

```
int  N = 8;
NsProblemManager  pm;

NsIntVarArray  Var, VarPlus, VarMinus;
for (int i=0;  i<N;  ++i)   {
    Var.push_back( NsIntVar(pm, 0, N-1) );
    VarPlus.push_back(  Var[i] + i );
    VarMinus.push_back( Var[i] - i );
}
pm.add( NsAllDiff(Var) );
pm.add( NsAllDiff(VarPlus) );
pm.add( NsAllDiff(VarMinus) );

pm.addGoal( new NsgLabeling(Var) );
while (pm.nextSolution() != false)
    cout << "Solution: " << Var << "\n";
```

7.2 $SEND + MORE = MONEY$

Another example is a known *cryptarithm* problem. In those problems we have some arithmetic relations between words, such as $SEND + MORE =$

$MONEY$. Each letter of the words represents a specific digit (from 0 to 9); thus, each word represents a decimal number. Two different letters should not represent the same digit. E.g. for the equation $SEND + MORE = MONEY$, we will put the same digit in the positions where E appears. The same applies for the rest of the letters, that should however be assigned different digits than the one for E. After all the assignments the relation of the cryptarithm should be valid. This is the problem declaration for the solver:

```
NsProblemManager  pm;

NsIntVar  S(pm,1,9), E(pm,0,9), N(pm,0,9), D(pm,0,9),
          M(pm,1,9), O(pm,0,9), R(pm,0,9), Y(pm,0,9);

NsIntVar  send  =           1000*S + 100*E + 10*N + D;
NsIntVar  more  =           1000*M + 100*O + 10*R + E;
NsIntVar  money = 10000*M + 1000*O + 100*N + 10*E + Y;

pm.add( send + more == money );

NsIntVarArray  letters;
letters.push_back( S );
letters.push_back( E );
letters.push_back( N );
letters.push_back( D );
letters.push_back( M );
letters.push_back( O );
letters.push_back( R );
letters.push_back( Y );
pm.add( NsAllDiff(letters) );

pm.addGoal( new NsgLabeling(letters) );
if (pm.nextSolution() != false)   {
    cout << "    " << send.value() << "\n"
         << " +  " << more.value() << "\n"
         << " = " << money.value() << "\n";
}
```

If we execute the code we take the result:

```
9567
```

```
 +  1085
 = 10652
```

7.3 How Do We State and Solve a Problem?

In the previous sections we stated some problems-examples; but what are the steps in order to state and solve another problem?

In § 2 we saw the source code basis to solve a problem. Under the comment `CODE OF THE PROGRAM` we insert the main part of our code.

Our code is summarized into the following triptych:

1. Constrained variables (`NsIntVar`) declaration, together with their domains.
2. Constraints statement (`pm.add(·)`).
3. Goals declaration (`pm.addGoal(new NsgLabeling(·))`) and search for solutions (`pm.nextSolution()`).

The first thing to do is to create a problem manager (`pm`), to store the whole constraint network. The declaration is

```
NsProblemManager  pm;
```

Next, we declare the constrained variables of the problem. Remember that while a simple variable (e.g. `int x`) stores only one value (e.g. `x=5`), a *constrained* variable stores a *range*, or, better, a domain. E.g. with the declaration `NsIntVar V(pm,0,5)`, the domain of `V` is the integer values range `[0..5]`.

When there are many constrained variables, then we use constrained variables arrays `NsIntVarArray`, as in the N Queens problem for example (§ 7.1). E.g.

```
NsIntVarArray  R;
```

The array `R` is initially empty. It is not possible to define a priori neither the array size, nor the included constrained variables domains. We can do this through an iteration

```
for (i=0;  i < N;  ++i)
        R.push_back( NsIntVar(pm,min,max) );
```

. . . in the way we insert items into a list. In place of `min` and `max` we put the minimum and maximum domain value, respectively. Next, we declare the existing constraints through `pm.add(·)` calls. . .

Before the end, if we solve an *optimization* problem, it remains to declare the parameter to optimize. When we find out this parameter-variable, we will pass it as an argument of `pm.minimize(·)`. This method is unnecessary when we seek for *any* solution of the problem.

We can now add a goal to be satisfied through the statement:

```
pm.addGoal( new NsgLabeling(R) );
```

This goal instructs the solver to assign values to the constrained variables of the array `R`. If we do not state this goal, the solver will not instantiate the variables `R[i]`, but it will only check the satisfaction of the constraints between *ranges* and the variables will not become fixed.

Finally, we execute `pm.nextSolution()` to find a solution. This function is called inside a loop; every time it returns `true`, we have another unique problem solution.

If we have previously called `pm.minimize(·)`, the solver guarantees that each new solution will be better from the previous one. In case `pm.nextSolution()` returns `false`, then either the solution cost cannot be further improved, or there is not any other solution. Thus we should have stored somewhere the last solution (and perhaps its cost too), in order to print it in the end, as in the following code for example:

```
NsDeque<NsInt>  bestR(N);

while ( pm.nextSolution()  !=  false )    {

        //  Record the (current) best solution.
        for (i=0;  i < N;  ++i)
                bestR[i] = R[i].value();
}

//  Print the best solution...
```

Note that when `nextSolution` seeks a solution, the constrained variables should not be destructed. Hence, it makes no sense to define a local constrained variable in a function and call `nextSolution` in another

function. Finally, a constrained variable definition should be straight; we cannot write something like the following, because in every iteration, the variable `vSum` is actually redefined:

```
NsIntVar  vSum;
for (i=0;  i < N;  ++i)
        vSum += R[i];  // WRONG!
```

The above can be simply stated as `NsIntVar vSum = NsSum(R);`

8 Search via Goals

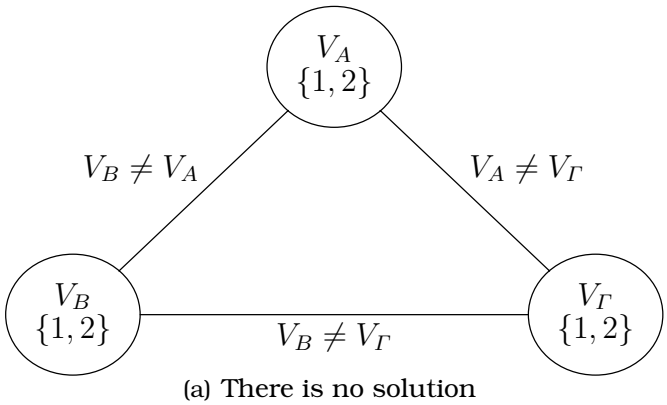
8.1 Introduction

A pair of variables (x, x') is *consistent,* if for each value v in the domain of x, there is a value v' in the domain of x' such that every constraint that connects the two variables is satisfied. When every pair of variables is consistent, then we say that the constraint network is *arc-consistent*. Arc-consistency does not necessarily mean that we have a solution—but if the constraint network is not arc-consistent, we are sure that there is no solution. So we have to combine arc-consistency with a search method. Besides, arc-consistency reduces the search space that a search method—such as depth first search (DFS), or limited discrepancy search (LDS) etc.—has to explore.

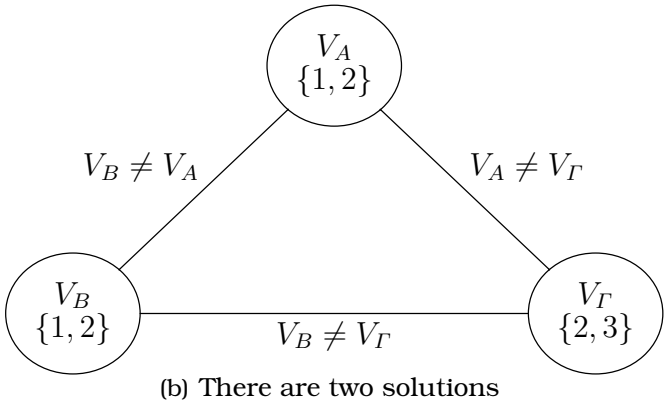
It is known that in most problems arc-consistency does not suffice to find a solution (see also Fig. 2). After a specific point, we should begin searching, by repeating the assignment of values to variables and by checking every time—e.g. after every assignment—if the constraint network is arc-consistent, according to the *maintaining arc-consistency* (MAC) methodology.[9] If an assignment causes an inconsistency, then it should be canceled and another value should be chosen.

In order to facilitate, or, better, to guide search, a *goals mechanism* has been implemented in the solver. The application developer that uses the solver can declare their own goals, or they can use the built-in ones. A goal often makes an assignment to a constrained variable, or it removes a value

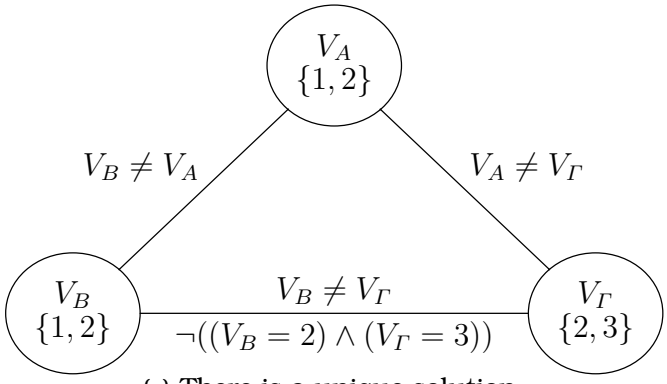
[9] D. Sabin and E. C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *PPCP'94: Proc. 2nd Int'l Workshop on Principles and Practice of Constraint Programming,* Washington, LNCS 874, pp. 125–129, 1994.



(a) There is no solution



(b) There are two solutions



(c) There is a unique solution

Figure 2: Three arc-consistent constraint networks

from the domain. If search reaches a dead-end, the solver automatically cancels the goals that guided to it and the constraint network with its variables is restored back to the state before those goals were executed.

Generally speaking, a goal can assign or remove values to one or more variables, or it can be used to choose a variable in order to be successively assigned a value. In this way it defines the search method. While a goal terminates, it can optionally generate another goal; this possibility provides recursion characteristics to the goals mechanism. Last but not least, there are also the AND and OR *meta-goals*. They are called 'meta-goals' because each of them is used to manipulate two *other* goals, namely *subgoals*. An AND-goal succeeds if its two subgoals succeed both, while an OR-goal succeeds if one or more of its subgoals succeed.

It is worth to mention that the OR-goals are also known as *choice points*. Indeed, they are points where we have two alternatives, that is points where the search tree branches off. During the execution of an OR-goal, its first subgoal is chosen and if it finally fails, the solver cancels all the chain modifications that were made on the domains of the variables (after the first subgoal execution); the second subgoal is then tried. If the second subgoal also fails, then the whole OR-goal fails.

8.2 Object-Oriented Modelling

The declaration for the basic goal class in NAXOS SOLVER follows.

```
class  NsGoal  {
    public:
        virtual bool  isGoalAND (void)  const;
        virtual bool  isGoalOR  (void)  const;
        virtual NsGoal*  getFirstSubGoal  (void)  const;
        virtual NsGoal*  getSecondSubGoal (void)  const;

        virtual NsGoal*  GOAL (void) = 0;
};
```

The `NsgAND` and `NsgOR` meta-goal classes derive from the above `NsGoal` class. `NsgAND` and `NsgOR` constructor functions take two arguments (of `NsGoal*` type), that represent their two subgoals. Every `NsGoal` member-function—apart from `GOAL()`—has to do with meta-goals. The application

developer that wants to define their own goals, has to take only care of `GOAL()`.

Every custom goal defined by the application developer should be a class that (directly or indirectly) extends `NsGoal`. Subsequently, function `GOAL()` should be defined in every goal class. Evidently, the goal class may also contain other member-functions—except from the ones also contained in the basic class, to avoid misunderstandings.

`GOAL()` is a critical method, as the solver executes it every time it tries to satisfy a goal. This method returns a pointer to another `NsGoal` instance, i.e. it returns the next goal to be satisfied. If the pointer equals to `0`, this means that the current goal succeeded (was satisfied) and thus no other goal has to be created.

So, an example follows, illustrating goals already built in the solver, as they are widely used. These goals describe the search method *depth-first-search* (DFS).

```
class NsgInDomain : public NsGoal  {
  private:
    NsIntVar&  Var;

  public:
    NsgInDomain (NsIntVar& Var_init)
      : Var(Var_init)    {    }

    NsGoal*  GOAL (void)
    {
       if (Var.isBound())
           return  0;
       NsInt  value = Var.min();
       return ( new NsgOR( new NsgSetValue(Var,value),
                       new NsgAND( new NsgRemoveValue(Var,value),
                                   new NsgInDomain(*this) ) ) );
    }
};


class NsgLabeling : public NsGoal  {
  private:
    NsIntVarArray&  VarArr;

  public:
    NsgLabeling (NsIntVarArray& VarArr_init)
      : VarArr(VarArr_init)    {    }
```

```
    NsGoal*  GOAL (void)
    {
        int  index = -1;
        NsUInt  minDom = NsUPLUS_INF;
        for (NsIndex i = 0;   i < VarArr.size();   ++i)   {
            if ( !VarArr[i].isBound()
                     && VarArr[i].size() < minDom )
            {
                minDom = VarArr[i].size();
                index = i;
            }
        }
        if (index == -1)
            return  0;
        return  ( new NsgAND( new NsgInDomain(VarArr[index]),
                              new NsgLabeling(*this) ) );
    }
};
```

We observe the operator `new` in the return value of `GOAL()` (when it is not `0`) and in the meta-goals (`NsgAND` and `NsgOR`) constructor functions. `new` is necessary when constructing a pointer to a goal. The solver is responsible to destruct the goal when it becomes useless, using the `delete` operator. That is why *all the goals that we create must be constructed with the* `new` *operator and we must* not `delete` *them by ourselves*.

Regarding the practical meaning of the example, when we ask the solver to satisfy the goal `NsgLabeling(VarArr)`, we except that all the variables of `VarArr` will be assigned values. Thus, the function `GOAL()` of `NsgLabeling` chooses a variable (specifically, the one with the smallest domain size according to the first-fail heuristic). Then it asks (via the goal `NsgInDomain` that assigns to a variable, its domain minimum value) to instantiate the variable *and* to satisfy the goal `this`. This goal—that refers to a kind of 'recursion'—constructs another `NsgLabeling` instance, that is identical to the current one. In fact, `this` tells the solver to assign values to the rest of `VarArr` variables. When `GOAL()` returns `0`, we have finished (Fig. 3).

While `NsgLabeling` chooses a variable to be instantiated, `NsgInDomain` chooses the value to assign to the variable. More specifically, it always chooses the minimum value of the domain of the variable. Then it calls the built-in goal `NsgSetValue` that simply assigns the value to the variable.

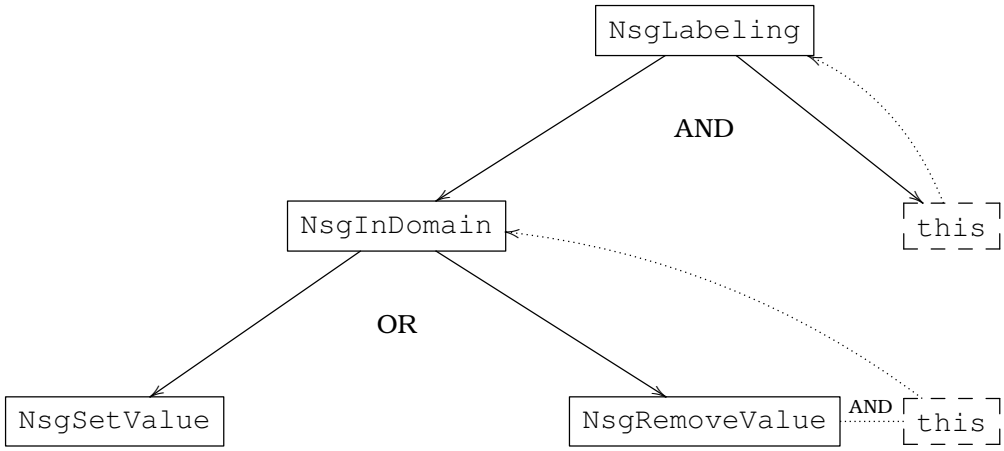


Figure 3: The combination of the goals that compose `NsgLabeling`

If it is proved afterwards that this value does not guide to a solution, it is removed from the domain by the goal `NsgRemoveValue` and another value will be assigned (by '`NsgInDomain(*this)`').

Usually, when we face difficult and big problems, we should define our own goals, like `NsgLabeling` and `NsgInDomain`. The aim is to make search more efficient by using heuristic functions to take better decisions/ choices.

Acknowledgements

I am grateful to Prof. Panagiotis Stamatopoulos, not only for his continuous and steady guidance, but also because he inspired, encouraged and embraced this attempt. I also thank Kyriakos Zervoudakis, a B.Sc. and M.Sc. graduate of the Department of Informatics and Telecommunications, that helped me in my first steps in Constraint Programming in 2004. Finally, many thanks to all the students of the Department that used the solver, either in the context of diploma theses, or through the Logic Programming course; we managed to improve the solver through the interaction and conversations between us. Unfortunately the name list is too big to quote, but I personally thank you one and all!

Index