Disolver 3.0 : the Distributed Constraint Solver

version 3.0

MSR-TR-2003-91

Youssef Hamadi
Microsoft Research
7 J J Thomson Avenue,
Cambridge CB3 0FB, UK
youssefh@microsoft.com

Contents

Chapter 1

Overview

1.1 General Description

DISOLVER 3.0 is a C++ constraint-based optimization library. It can be combined with any MPI toolkit [MPI94], to seamlessly run on parallel architectures. More precisely, it delivers:

- Complete search. Methods in this category perform tree-based exploration of a problem search space. DISOLVER 3.0 provides algorithms to perform *satisfaction* testing and *optimization*. Optimization is performed through efficient *branch & bound* algorithms.
- Parallel search. This category generalizes the previous algorithms, *e.g., parallel branch & bound*. It provides advanced *load balancing* and *knowledge sharing* controls.

1.2 Availability

Please contact the author if you consider to use DISOLVER 3.0.

Chapter 2

Sequential search

2.1 Getting Started

This part presents the basics of Constraint Programming (CP) with DISOLVER 3.0[1]. In the following, we show how to model a combinatorial problem by defining variables, their possible range of values and associated constraints. After this small introduction, a classical example is used to illustrate a CP modelling in DISOLVER 3.0.

2.1.1 Basics

In Constraint Programming a problem is represented through a set of variables which are restricted by a set of constraints. Each variable is defined with a *domain*, *i.e.,* a range of possible values. So far, DISOLVER 3.0 defines *finite domain* (FD) CP variables. Constraints represent restriction on value combinations. Variables express the "decision" components of the problems and constraints are here to restrict the choices on these components. Each constraint has a particular impact on related variables and uses a specific filtering algorithm to reduce the search space. Initially the search space corresponds to the Cartesian product of domain combinations. Constraint Programming employs the filtering algorithm associated to each constraint to efficiently prune the space. Sometimes this pruning is able to find a unique solution or to detect unsatisfiability. In most cases, a backtrack search is used conjointly with constraint filtering to locate solution tuples.

The main objective of Constraint Programming is to identify important variables and to define appropriate constraints in order to efficiently solve a problem.

2.1.2 A classical example

For illustration purposes we will use a very simple problem: compute a sum `S` with coins of 1, 10, 25 or 100 cents. We have respectively `n1, n10, n25, n100` coins of 1, 10, 25 and 100 cents. This problem is part of our benchmark set.

We can already analyze the input and find that we have four unknown values representing respectively the number of 1, 10, 25, 100 cents coins used to compute `S`. Let's call these variables `X1, X10, X25 and X100`. From the input we have the values for `S, n1, n10, n25, n100`.

We will use a constrained variable for each unknown value. The range of each variable is given by the number of available coins:

- `X1:[0..n1]`
- `X10:[0..n10]`
- `X25:[0..n25]`

[1]Interested readers could refer to [Hen89, BHZ06] for introductory materials around Constraint Programming.

- `X100:[0..n100]`

To restrict the combination of values of these variables we will use the following arithmetic constraint:
`S = X1 + X10*10 + X25*25 + X100*100`

With the previous modelling, DISOLVER 3.0 is able to find any combination of the variables equal to `S`. You can have a look at the `Benchs/` directory to test this problem.

If we set `S=253, n1=5, n10=10, n25=5 and n100=1`. DISOLVER 3.0 finds the two possible solutions:

- `X1 = 3, X10 = 5, X25 = 4, X100 = 1`
- `X1 = 3, X10 = 10, X25 = 2, X100 = 1`

2.2 Building blocks

In the previous section, we have presented the basic process of problem solving in Constraint Programming. We saw that a problem is defined by using variables with possible values and constraints. This section we present the different components of the library.

2.2.1 Problem

In DISOLVER 3.0 the most important object is the `Problem` object. This component is used to define variables, constraints and to run search algorithms. Any problem formulation has to start by its definition. Each problem is self-containing and therefore multiple problems can be manipulated at the same time.

```
Problem p;
```

2.2.2 Variables

How do we express variables in DISOLVER 3.0? We have two ways of doing this. DISOLVER 3.0 distinguishes among `CP` variables and `CSP` ones. They differ in their internal representation. A `CP` variable limits the internal representation to the bounds of the domain, while a `CSP` variable explicitly stores its domain values (with a compact internal representation). Obviously, the second type is more space consuming. Both variables are interchangeable in all constraints definitions.

In DISOLVER 3.0 the constraints are able to distinguish among these types and use this information to apply the best possible filtering of the search space. Usually, the filtering on `CSP` is more accurate and more time consuming. However, this extra filtering cost can highly benefit to search and the overall problem solving time can be reduced.

The choice among `CP` or `CSP` representation will depend on the problem. Sometimes the choice is obvious *e.g.,* variables with very large domains are more efficiently managed by `CP`. When the choice is not straightforward, it is better to test with the two previous categories.

There is an easy solution here which is to let the library decide for the representation. The decision is made on some threshold: "small" domains are stored through `CSP` while large domains use the other type.

Two low level types are used to control the range (and performances) of the variables. By default, (Disolver32.DLL) the `DISOLVER_INT` macro is set (see `global.h`) and the range is given by the C++ type, `INT`. On current 32-bits CPUs the largest possible domain for a variable is $-2^{31}-1$ to $2^{31}-1$, *i.e.,* nearly 2^{32} possible values for one variable. More generally, the previous range corresponds to `INT_MIN`, `INT_MAX - 1`.

When the `DISOLVER_INT` macro is unset, (Disolver64.DLL) the `__int64` C++ type is used. This type gives a range between `_I64_MIN` and `_I64_MAX - 1`. On 32-bits CPUs it corresponds to $-2^{63}-1$ to $2^{63}-1$.

If you do not have to represent extra large integer, always link your application to the Disolver32.DLL otherwise, link it to Disolver64.DLL.

If you want to try your application with both internal representation, do not forget to use the following macros. `DISOLVERDOMAIN` to represent the data-type used for domain representation (set to either `int` or `__int64`. `MINVALUE/MAXVALUE` to represent respectively lower/upper bounds limits.

Let's go back to our small example, with DISOLVER 3.0 we can represent the four unknown values of the problem like that:

```
Variable* X1 = p.createCSPVariable("1cent", 0, n1);
Variable* X10 = p.createCSPVariable("10cent", 0, n10);
Variable* X25 = p.createCSPVariable("25cent", 0, n25);
Variable* X100 = p.createCSPVariable("100cent", 0, n100);
```

We choose to use an explicit representation for the domains (`CSP`). The basic declaration uses three parameters. The first parameter is optional, its a string representing the name of the variable. It can be useful to manipulate variables, and can be retrieved through a `getName()` function call. The second and third parameters are made of integers representing respectively the lower (lb) and upper (ub) bounds of the domain. `CP` types are created through the `createVariable` function, while `CSP` types are created through the `createCSPVariable` function.

More clearly the following functions can be applied to a problem object:

```
Variable* createCPVariable(DISOLVERDOMAIN lb, DISOLVERDOMAIN ub);
Variable* createCSPVariable(DISOLVERDOMAIN lb, DISOLVERDOMAIN ub);
Variable* createCPVariable(char* name, DISOLVERDOMAIN lb, DISOLVERDOMAIN ub);
Variable* createCSPVariable(char* name, DISOLVERDOMAIN lb, DISOLVERDOMAIN ub);
```

It is also possible to let DISOLVER 3.0 decide for the best internal representation through the function:

```
Variable* createVariable(DISOLVERDOMAIN lb, DISOLVERDOMAIN ub);
```

Most of the time, you will have to create a large number of variables (typically n) with similar domains. The library offers simple ways to create vectors, matrices and half-matrices of variables:

```
Variable** createVariableVect(int n, DISOLVERDOMAIN lb, DISOLVERDOMAIN ub);
Variable** createCPVariableVect(int n, DISOLVERDOMAIN lb, DISOLVERDOMAIN ub);
Variable** createCSPVariableVect(int n, DISOLVERDOMAIN lb, DISOLVERDOMAIN ub);
```

The following functions create respectively $n \times m$ matrices and $n \times n$ half-matrices:

```
Variable*** createVariableMat(int n, int m, DISOLVERDOMAIN lb, DISOLVERDOMAIN ub);
Variable*** createCPVariableMat(int n, int m, DISOLVERDOMAIN lb, DISOLVERDOMAIN ub);
Variable*** createCSPVariableMat(int n, int m, DISOLVERDOMAIN lb, DISOLVERDOMAIN ub);

Variable*** createVariableHalfMat(int n, DISOLVERDOMAIN lb, DISOLVERDOMAIN ub);
Variable*** createCPVariableHalfMat(int n, DISOLVERDOMAIN lb, DISOLVERDOMAIN ub);
Variable*** createCSPVariableHalfMat(int n, DISOLVERDOMAIN lb, DISOLVERDOMAIN ub);
```

2.2.3 Constraints

DISOLVER 3.0 has a set of pre-defined constraints that the programmer can apply on variables. Constraints apply to any combination of variables type (CSP/CP). They are defined through a call to a problem object. In the following, we present the set of pre-defined constraints.

Is

This unary constraint uses one variable and one integer as parameters:

```
Constraint* createIs(&X, DISOLVERDOMAIN v);
```

It ensures that the variable `X` is equal to `v`.

Not

This unary constraint uses one variable and one integer as parameters:

```
Constraint* createNot(&X, DISOLVERDOMAIN v);
```

It ensures that `X` cannot be equal to `v`.

Equal

This binary constraint uses two variables as parameters:

```
Constraint* createEqual(&X, &Y);
```

It ensures that `X` and `Y` are equals.

Diff

This binary constraint uses two variables as parameters:

```
Constraint* createDiff(&X, &Y);
```

It ensures that `X` and `Y` are different.

Minus

This binary constraint uses two variables as parameters:

```
Constraint* createMinus(&X, &Y);
```

It ensures that `Y` is equal to `-X`.

Inf

This constraint uses two variables or one variable and one integer v as parameters:

```
Constraint* createInf(&X, &Y);
Constraint* createInf(&X, DISOLVERDOMAIN v);
```

It ensures that `X` is strictly less than `v`.

InfEqual

This binary constraint uses two variables as parameters:

```
Constraint* createInfEqual(&X, &Y);
```

It ensures that `X` is less or equal than `Y`.

Sup

This constraint uses two variables or one variable and one integer v as parameters:

```
Constraint* createSup(&X, &Y);
Constraint* createSup(&X, DISOLVERDOMAIN v);
```

It ensures that `X` is strictly more than `v`.

SupEqual

This binary constraint uses two variables as parameters:

```
Constraint* createSupEqual(&X, &Y);
```

It ensures that `X` is more or equal to `Y`.

Plus

Uses three variables or a combination of DISOLVERDOMAIN values and variables as parameters:

```
Constraint* createPlus(&X, &Y, &Z);
Constraint* createPlus(DISOLVERDOMAIN X, &Y, &Z);
Constraint* createPlus(&X, DISOLVERDOMAIN Y, &Z);
Constraint* createPlus(&X, &Y, DISOLVERDOMAIN Z);
```

Ensures that `X + Y = Z`.

Minus

Uses three variables or a combination of DISOLVERDOMAIN values and variables as parameters:

```
Constraint* createMinus(&X, &Y, &Z);
Constraint* createMinus(DISOLVERDOMAIN X, &Y, &Z);
Constraint* createMinus(&X, DISOLVERDOMAIN Y, &Z);
Constraint* createMinus(&X, &Y, DISOLVERDOMAIN Z);
```

Ensures that `X - Y = Z`.

Mult

Uses three variables or a combination of DISOLVERDOMAIN values and variables as parameters:

```
Constraint* createMult(&X, &Y, &Z);
Constraint* createMult(DISOLVERDOMAIN X, &Y, &Z);
Constraint* createMult(&X, DISOLVERDOMAIN Y, &Z);
Constraint* createMult(&X, &Y, DISOLVERDOMAIN Z);
```

Ensures that `X * Y = Z`.

PlusList

Applies on n variables:

```
Constraint* plus = createPlusList(&Z, int n);
Constraint* plus = createPlusList(DISOLVERDOMAIN Z, int n);
plus->addVar(&X1);
plus->addVar(&X2);
...
plus->addVar(&Xn);

Constraint* plus = createPlusList(&Z, int n, Variable** Xi);
```

Ensures that `X1 + X2 + ... + Xn = Z`, where `Z` can be a variable or an integer of the `DISOLVERDOMAIN` type. The function `addVar` is used to link the X_is variables one by one. When these variables were created as a vector, it is possible to pass it as the last parameter.

To ensure that `c1 * X1 + c2 * X2 + ... + cn * Xn = Z`, the function `addVar(ci, &Xi)` can be used.

Sum

Applies on n variables:

```
Constraint* plus = createSum(&Z, int n);
plus->addVar(DISOLVERDOMAIN c1, &X1);
plus->addVar(DISOLVERDOMAIN c2, &X2);
...
plus->addVar(DISOLVERDOMAIN cn, &Xn);

Constraint* plus = createSum(&Z, int n, Variable** Xi);
```

Ensures that `c1 * X1 + c2 * X2 + ... + cn * Xn = Z`. Unlike `PlusList`, this constraint does not use intermediate variables. Therefore it is less memory consuming but not necessarily better since the pruning is somewhat approximated. Therefore, we recommend the use of `PlusList` for most applications.

BitVect

Applies on 1 numerical variable with positive range and n variables with a Boolean (0/1) range:

```
Constraint* b = createBitVect(&X, int n);
b->addVar(&X1);
b->addVar(&X2);
...
b->addVar(&Xn);
```

Ensures that the vector $\langle X_1 \dots X_n \rangle$ corresponds to the binary representation of the unsigned integer X. In other words, the constraint ensures that $X = \sum_{i \in 1 \dots n} 2^{i-1} X_i$. Note that the order in which the variables are added matters: the bits variables have to be added by increasing weight.

Min

Uses three variables as parameters:

```
Constraint* createMin(&X, &Y, &Z);
```

Ensures that `min(X, Y) = Z`.

MinList

Applies on n variables:

```
Constraint* mini = createMinList(&Z, int n);
Constraint* mini = createMinList(DISOLVERDOMAIN Z, int n);
mini->addVar(&X1);
mini->addVar(&X2);
...
mini->addVar(&Xn);

Constraint* mini = createMinList(DISOLVERDOMAIN Z, int n, Variable** Xi);
```

Ensures that `min(X1, X2, ... , Xn) = Z`.

Max

Uses three variables as parameters:

```
Constraint* createMax(&X, &Y, &Z);
```

Ensures that `max(X, Y) = Z`.

MaxList

Applies on n variables:

```
Constraint* maxi = createMaxList(&Z, int n);
Constraint* maxi = createMaxList(DISOLVERDOMAIN Z, int n);
maxi->addVar(&X1);
maxi->addVar(&X2);
...
maxi->addVar(&Xn);

Constraint* maxi = createMaxList(DISOLVERDOMAIN Z, int n, Variable** Xi);
```

Ensures that `max(X1, X2, ... , Xn) = Z`.

Modulo

Uses three variables as parameters:

```
Constraint* createModulo(&X, &Y, &Z);
```

Ensures that `X % Y = Z`.

Abs

This binary constraint uses two variables as parameters:

```
Constraint* createAbs(&X, &Y);
```

It ensures that `Y` is equal to `abs(X)`.

AllDiff

This constraint takes a set of n variables and ensures that they have different values. To define an `AllDiff` constraint among variables `X,Y,Z`, we do the following:

```
Constraint* diff = createAllDiff(3);
diff->addVar(&X);
diff->addVar(&Y);
diff->addVar(&Z);
Constraint* diff = createAllDiff(n, Variable** Xi);
```

First we define an `AllDiff` constraint of size 3 called `diff` then, we apply it on the three variables by using the `addVar` primitive. Again, the function with three parameters uses a vector of variables and therefore does not require calls to `addVar`.

When, according to the modelling, it appears that the pruning is made on the bounds of the variables, a specific propagator `AllDiffBounds` can be used [LOQTvB03]:

```
Constraint* diff = createAllDiffBounds(n);
diff->addVar(&X1);
...
diff->addVar(&Xn);

Constraint* diff = createAllDiffBounds(n, Variable** Xi);
```

Element

This constraint implements the notion of "indirection" in Constraint Programming.

```
Constraint* createElement(DISOLVERDOMAIN* Tab, int n, &X, &Y);
```

The first argument is a vector of n `DISOLVERDOMAIN` values. The third and fourth arguments are DISOLVER 3.0 variables. The constraints ensures that `Tab[X] = Y`.

This constraint can also use a vector of n DISOLVER 3.0 variables (CSP/CP):

```
Constraint* createElement(Variable** Tab, int n, &X, &Y);
```

Occur

The value `Value` occurs `Condition` times in the n associated variables. The condition and the value can be integers and/or variables.

```
Constraint* occ = createOccur(&Condition, &Value, int n);
Constraint* occ = createOccur(&Condition, DISOLVERDOMAIN Value, int n);
Constraint* occ = createOccur(DISOLVERDOMAIN Condition, &Value, int n);
Constraint* occ = createOccur(DISOLVERDOMAIN Condition, DISOLVERDOMAIN Value, int n);
occ->addVar(&X1);
occ->addVar(&X2);
...
occ->addVar(&Xn);

Constraint* occ = createOccur(&Condition, &Value, int n, Variable** Xi);
Constraint* occ = createOccur(&Condition, DISOLVERDOMAIN Value, int n, Variable** Xi);
Constraint* occ = createOccur(DISOLVERDOMAIN Condition, &Value, int n, Variable** Xi);
Constraint* occ = createOccur(DISOLVERDOMAIN Condition, DISOLVERDOMAIN Value, int n, Variable** Xi);
```

AtLeast

The value `Value` occurs at least `Condition` times in the n associated variables. The condition is an integer; the value can be represented by an integer or by a variable.

```
Constraint* at = createAtLeast(DISOLVERDOMAIN Condition, &Value, int n);
Constraint* at = createAtLeast(DISOLVERDOMAIN Condition, DISOLVERDOMAIN Value, int n);
at->addVar(&X1);
at->addVar(&X2);
...
at->addVar(&Xn);

Constraint* at = createAtLeast(DISOLVERDOMAIN Condition, &Value, int n, Variable** Xi);
Constraint* at = createAtLeast(DISOLVERDOMAIN Condition, DISOLVERDOMAIN Value, int n, Variable** Xi);
```

AtMost

The value `Value` occurs at most `Condition` times in the n associated variables. The condition is an integer, the value can be represented by an integer or by a variable.

```
Constraint* am = createAtMost(DISOLVERDOMAIN Condition, &Value, int n);
Constraint* am = createAtMost(DISOLVERDOMAIN Condition, DISOLVERDOMAIN Value, int n);
am->addVar(&X1);
am->addVar(&X2);
...
am->addVar(&Xn);

Constraint* am = createAtMost(DISOLVERDOMAIN Condition, &Value, int n, Variable** Xi);
Constraint* am = createAtMost(DISOLVERDOMAIN Condition, DISOLVERDOMAIN Value, int n, Variable** Xi);
```

2.2.4 Boolean constraints

Variables ranging over $\{0, 1\}$ can encode Booleans. DISOLVER 3.0 provides a complete set of Boolean constraints which can be used on such variables:

And

Uses three variables:

```
Constraint* createAnd (&X, &Y, &Z)
```

Ensures that `Z` is the logical conjunction of `X` and `Y`, *i.e.* it is true iff both of these variables are true.

Or

Uses three variables:

```
Constraint* createOr (&X, &Y, &Z)
```

Ensures that `Z` is the logical disjunction of `X` and `Y`, *i.e.* it is true iff at least one of these variables is true.

Negate

Uses two variables:

```
Constraint* createNegate (&X, &Y)
```

Ensures that `Y` is the logical negation of `X`, *i.e.* it is true iff this variable is false.

Of course, in all the previous constraints, variables `X`, `Y` and `Z` are assumed to range over $\{0, 1\}$. Note that the approach used in DISOLVER 3.0 is to define one constraint per basic operator in the complete set $\{\wedge, \vee, \neg\}$. This set is clearly sufficient to express any constraint of propositional logic. Also, similarly to numerical operators, each unary (*resp.* binary) numerical operator is provided as a binary (*resp.* ternary) constraint stating the *relation* between the inputs of the operator and its result.

2.2.5 Reified Constraints

It is sometimes convenient to capture the truth value of some constraints into *pseudo-Boolean* variables, *i.e.* variables ranging over $\{0, 1\}$ and whose value can be interpreted both in a numerical and in a Boolean way. Pseudo-Boolean variables also allow to construct complex expressions which are hard to express otherwise. Typical cases are sums of the form $\sum_i \{x_i \mid cond(x_i)\}$, in which only the elements respecting a particular condition must be added together (see the end of this section for an example of use of this constraint).

To allow this use of pseudo-Boolean variables, DISOLVER 3.0 defines *reified* versions of a number of basic constraints. Compared to the non-reified versions, reified constraints take an additional argument, which is the variable representing the validity of the constraint. For instance, whereas the binary constraint `InfEqual` represents the relation $x \leq y$, the reified version `InfEqualB`, which takes 3 arguments, will impose the equivalence $(x \leq y) \Leftrightarrow b$. Note that this is consistent with our constraint-based approach, in which functions are encoded as relations: the inequality function $\leq$ can be seen as a Boolean function and its reified counterpart just captures the relation between the input $\langle x, y \rangle$ and this Boolean result. From an operational viewpoint, setting the Boolean to true or false will impose the constraint $x \leq y$ or $x > y$; conversely `b` will be set accordingly if the solver detects that one of these inequalities holds.

The list of the reified constraints defined in DISOLVER 3.0 is the following:

InfEqual

Uses three variables:

```
Constraint* createInfEqual (&X, &Y, &B)
```

Ensures that `(X <= Y) iff B`. Note that `B` must range over $\{0, 1\}$.

SupEqual

Uses three variables:

```
Constraint* createSupEqual (&X, &Y, &B)
```

Ensures that `(X >= Y) iff B`. Note that `B` must range over $\{0, 1\}$.

Inf

Uses three variables:

```
Constraint* createInf (&X, &Y, &B)
```

Ensures that `(X < Y) iff B`. Note that `B` must range over $\{0, 1\}$.

Sup

Uses three variables:

```
Constraint* createSup (&X, &Y, &B)
```

Ensures that `(X > Y) iff B`. Note that `B` must range over $\{0, 1\}$.

Equal

Uses three variables:

```
Constraint* createEqual (&X, &Y, &B)
```

Ensures that `(X = Y) iff B`. Note that `B` must range over $\{0, 1\}$.

Diff

Uses three variables:

```
Constraint* createDiff (&X, &Y, &B)
```

Ensures that `(X != Y) iff B`. Note that `B` must range over $\{0, 1\}$.

2.2.6 Operators

Many problems are expressed using numerical and/or Boolean expressions which have to be decomposed into basic constraints. For instance, expressing the constraint $A \geq 2.B + 10$ requires to introduce an intermediate variable $I1$ for the term $2.B$, an intermediate variable $I2$ for the term $I1 + 10$, to express the basic constraints between B, $I1$ and $I2$ and to impose the inequality $A \geq I2$. This can be written as follows:

```
...
Variable*  I1 = p.createCPVariable (``I1'', 2*B->getLb(), 2*B->getUb());
Constraint* c1 = p.createMult (B, 2, I1);
Variable*  I2 = p.createCPVariable (``I2'', I1->getLb()+10, I1->getUb()+10);
Constraint* c2 = p.createPlus (I1, 10, I2);
Constraint* c3 = p.createSupEqual (A, I2);
```

This decomposition becomes tedious when the expressions get complex. To cope with this problem, DISOLVER 3.0 provides an overloading of the basic arithmetic and Boolean operators of C++. When applied to objects of type `CP`, these operators return a new variable connected to its arguments by the corresponding constraint. For instance, applying the addition or the multiplication operator on two variables (or a variable and an integer), we obtain a new variable connected to its arguments by the constraint `Plus` or `Mult`. We can therefore replace the previous code by:

```
Variable&  I2 = ((*B) * 2) + 10;
Constraint* c2 = p.createSupEqual (A, &I2);
```

Or even:

```
Constraint* c2 = p.createSupEqual (A,  &(((*B) * 2) + 10));
```

Note that:

- These operators take *references* as arguments and also return a *reference* to the created variable; the `&` (address) and `*` (dereferencing) operators therefore have to be used to ensure a correct typing.
- Operators represent mere syntactical sugar for primitive constraints, *each operator introduces a new intermediate variable and a new constraint relating this variable to the arguments of the operator*. The bounds of these intermediate variables are initialised automatically in the code of each operator, which prevents mistakes that could occur when manually adding intermediate variables.

Arithmetic operators

DISOLVER 3.0 defines the following arithmetic operators:

operator	arguments	associated constraints
`+`	two variables	`Plus`
	one variable, one int	`Plus`
`-`	one variable	`Minus`
	two variables	`Minus`
	one variable, one int	`Plus`/ `Minus`
`*`	two variables	`Mult`
	one variable, one int	`Mult`
`abs`	one variable	`Abs`
`minimum`	2 vars / 1 var, 1 int	`Min`
`maximum`	2 vars / 1 var, 1 int	`Max`
`sumVars`	list of vars	`PlusList`
`linearTerm`	list of coefs/vars	`Sum`

In the previous table, we have indicated by which constraint the new variable is connected to its arguments. Note that the operators and functions can be arbitrarily nested, for instance the expression `2*abs(X)` (which we can alternatively write `2*maximum(X, -X)`) is correct provided `X` is of type `Variable&`; this expression is itself of type `Variable&`.

Boolean operators

The operators of addition, substraction, and multiplication, create a variable which, intuitively, represents an integer value. Similarly, some operators return a value which represents a Boolean (encoded as an integer with value 0/1). DISOLVER 3.0 defines the following operators:

operator	arguments	associated constraints
`<, <=, >=, >`	2 vars / 1 var, 1 int	`InfEqual`, `Sup`, ...
`==, !=`	2 vars / 1 var, 1 int	`Equal`, `Diff`, ...
`&&`	2 vars / 1 var, 1 int	`And`
`||`	2 vars / 1 var, 1 int	`Or`
`!`	1 var	`Negate`

These operators can be used to define complex combinations of constraints with a simple syntax.

A source of errors in the use of operators is that the relational operators $<$, $\leq$, $\geq$, $>$, $=$ and $\neq$ should *NOT* be used directly to impose an inequality or equality. Directly writing:

```
x >= y + 1; // NEVER DO THAT!!
```

where x and y are references to variables, is accepted by the compiler, just as it would be if x and y were integers. But, just as on integers, this expression does not impose anything, it simply creates a variable which will be set to 1 if the inequality holds and to 0 otherwise. This will have no visible effect if the variable is not captured by the user. Consequently: *the variables returned by the overloaded operators should always be captured by the user.*

2.2.7 Search

After modelling your problem with the DISOLVER 3.0 constructs, you can search for solution(s). DISOLVER 3.0 provides two functions for satisfaction testing. The first one `solve()` stops when a solution is found or when the unsatisfiability is proved. This function is applied to a `Solver` object. Instances of this object can be created through a call to a problem object.

```
Problem p;
Solver* solver = p.createTreeSolver();
int solver->solve();
```

It returns an integer which encodes the result of the search process. When the value 0 is returned the solver found one or more solutions. When -1 is returned, the problem has no solution.

If `solve` returns 0, you can access the solution through the `getValue()` function applied to a variable object.

```
DISOLVERDOMAIN X->getValue();
```

You can access information or statistics on the search process through the function `print()` applied to a `Solver` object. The section 2.3.8 provides several functions to selectively access these information.

```
solver->print();
```

Here are the search statistics when we solve the small *money* example:

```
Disolver 3.0, 32bits, 1 thread(s)
hit limit:      0
#solution:      1
search#0:       SOLVERESTART
heuristics:     DOMWDEG+LEX
time:           0
#fails:         0
#nodes:         1
#restarts:      0
#constraints:   1
#variables:     4 (4)   4/0/0Kb
#references:    0
trail resizes:  0/0/0/0
```

The first line gives the version of the toolkit along the size of the domains and the number of threads used by OpenMP. The "hit limit" tag is set to 1 if the search was interrupted by one explicit limit (e.g., time limit, backtracking limit, etc.) see 2.3.7 for more details. The number of solution(s) is then presented. The category of the search algorithm is shown, here we can see that the default strategy corresponding to a call to the solve function without any argument uses a restart-based backtracking search (see 2.3.3). The (default) heuristics used during the search are then presented. Finally, the time, the number of backtracking (#fails), and the number of nodes are presented.

The programmer can also ask the solver to search for all solutions:

```
solver->solveAll();
solver->solveAll(int (*foo)());
```

In the second call, `foo()` is a c++ function which is called each time the solver finds a solution. This function can then manipulate (print-out, store, etc.) solutions. This function must return 1 in order to look for another solution, 0 otherwise.

The complete "money" example

```
#include "disolver.h"

void money(int s, int n1, int n10, int n25, int n100){
  Problem p;
  Solver* solver = p.createTreeSolver();
  Variable* X1 = p.createCSPVariable("1cent", 0, n1);
  Variable* X10 = p.createCSPVariable("10cent", 0, n10);
  Variable* X25 = p.createCSPVariable("25cent", 0, n25);
  Variable* X100 = p.createCSPVariable("100cent", 0, n100);

  Constraint* prod = p.createSum(s, 4);
  prod->addVar(X1);
  prod->addVar(10, X10);
  prod->addVar(25, X25);
  prod->addVar(100, X100);

  solver->solve();
  solver->print();

  cout<<"\nSolution: "<<s<<" = "<<X1->getValue()<<" + 10*"<<X10->getValue()
  <<" + 25*"<<X25->getValue()<<" + 100*"<<X100->getValue()<<endl;
}

main(int argc, const char *argv[]){
  money(253, 5, 10, 5, 1);
}
```

2.2.8 Optimization

DISOLVER 3.0 can solve optimization problems. The programmer has to set up a constraint variable representing the objective function. The solver minimizes or maximizes the value of the objective function during the search process. Complex objective functions can be defined through combination of low-level constraints. DISOLVER 3.0 uses *branch & bound* to tackle optimization problems.

To illustrate this new feature we need a new problem. Let us consider the *Optimal Golomb Ruler* (OGR) problem [She]. A simple definition for this problem is given by J. B. Shearer: *"A Golomb ruler is a set of integers (marks)* $a(1) < ... < a(n)$ *such that all the differences* $a(i) - a(j)$ $(i > j)$ *are distinct. Clearly we may assume* $a(1) = 0$*. Then* $a(n)$ *is the length of the Golomb ruler. For a given number of marks,* n*, we are interested in finding the shortest Golomb rulers. Such rulers are called optimal. "*

This problem is part of our suite of benchmarks.

```
#include "disolver.h"
int results[] = {0, 0, 1, 3, 6, 11, 17, 25, 34, 44, 55, 72, 85, 106, 127};

void golomb(Problem* p, int n){
  Variable** X = p->createCPVariableVect(n, 0, n * n - 1);
  Variable*** D = p->createCSPVariableHalfMat(n, 0, n * n, IMPLICIT);

  int k = 0;
  for(int i = 0; i < n; i++)
    for(int j = 0; j < i; j++){
```

```
      int lb = (i - j) * (i - j + 1) / 2;
      //sum of the first i - j integers

      if((i - j < n - 1)&&(i - j + 1 < 15)) lb = results[i - j + 1];
      //improve lb with smaller OGR sizes

      p->createSup(D[i][j], lb - 1);
      k++;
    }

  Constraint* diff = p->createAllDiff(n * (n - 1) / 2);
  for(int i = 0; i < n; i++)
    for(int j = 0; j < i; j++){
       minus = p->createMinus(X[i], X[j], D[i][j]);
       diff->addVar(D[i][j]);
    }

for(int i = 0; i < n - 1 ; i++)
p->createInf(X[i], X[i+1]);

//start
p->createIs(X[0], 0);

//symetry break
p->createInf(D[1][0], D[n-1][n-2]);

Solver* solver = p->createTreeSolver();
solver->unsetSilent();

solver->minimize(X[n-1], LEX, LEX, INT_MAX);


if(p->getMyRank() == 0){//proc#0 print out the solution
solver->print();
cout<<"\nSolution: ";
for(int i = 0; i < n; i++){
cout<<X[i]->getValue()<<" ";
}
  }
}

int main(int argc, char** argv){
  int n = 4;
  if(argc > 1) n = atoi(argv[1]);

  Problem p();
  golomb(&p, n);
}
```

We store the location of each mark in a vector `X`. The difference among any two marks is stored in a matrix `D`. Each variable $X[i]$ can take any value among 0 and $n^2 - 1$. Each difference $D[i][j]$ holds between lb and n^2. The lower bound, lb is at least equal to the sum of the first $i - j$ integers. However, we can take

advantage of previously known OGR sizes to improve it.

An `AllDiff` constraint `diff` is set on the matrix D. That constraint will ensure that each difference $D[i][j]$ is unique.

The marks are then ordered. These constraints are not necessary to express the problem but they are very useful for the efficiency of the search process (*i.e.,* symmetry break). We state that $X[i] < X[i+1]$.

After that we enforce the first mark to be set at 0 by using the primitive `setValue`. This primitive is equivalent to an `Is` constraint.

We can then break some symmetry by stating that the first difference is less than the last one: $D[1][0] < D[n-1][n-2]$.

In this problem the last mark $X[n-1]$ must be minimized. We ask DISOLVER 3.0 to do that by using the `minimize` function:

```
Solver->minimize(&X[n-1], LEX, LEX, INT_MAX);
```

The first argument is the "objective" variable to minimize, the second and third arguments are tags which are discussed in the "Advanced usage" section (2.3). The last argument is an integer representing the initial value of the cost variable. Here we use the pre-defined value `INT_MAX` but $n^2 - 1$ could have been used since it is the upper bound of the objective variable.

With $n = 10$, the output of the previous program is the following:

```
Proc#0, at time 0.000000 s, 0 / 9
minimize best cost = 80 : 0, 1, 3, 7, 12, 20, 30, 44, 65, 80, 1, 3, 2, 7, 6, 4,
12, 11, 9, 5, 20, 19, 17, 13, 8, ...

Proc#0, at time 0.000000 s, 3 / 16
minimize best cost = 75 : 0, 1, 3, 7, 12, 20, 34, 49, 59, 75, 1, 3, 2, 7, 6, 4,
12, 11, 9, 5, 20, 19, 17, 13, 8, ...

Proc#0, at time 0.000000 s, 8 / 28
minimize best cost = 73 : 0, 1, 3, 7, 12, 22, 35, 49, 65, 73, 1, 3, 2, 7, 6, 4,
12, 11, 9, 5, 22, 21, 19, 15, 10, ...

Proc#0, at time 0.000000 s, 26 / 58
minimize best cost = 72 : 0, 1, 3, 7, 12, 26, 41, 54, 62, 72, 1, 3, 2, 7, 6, 4,
12, 11, 9, 5, 26, 25, 23, 19, 14, ...

Proc#0, at time 0.000000 s, 67 / 123
minimize best cost = 70 : 0, 1, 3, 7, 15, 24, 34, 54, 59, 70, 1, 3, 2, 7, 6, 4,
15, 14, 12, 8, 24, 23, 21, 17, 9, ...

Proc#0, at time 0.000000 s, 115 / 200
minimize best cost = 68 : 0, 1, 3, 7, 15, 31, 36, 49, 58, 68, 1, 3, 2, 7, 6, 4,
15, 14, 12, 8, 31, 30, 28, 24, 16, ...

Proc#0, at time 0.000000 s, 231 / 371
minimize best cost = 66 : 0, 1, 3, 7, 17, 22, 35, 46, 58, 66, 1, 3, 2, 7, 6, 4,
17, 16, 14, 10, 22, 21, 19, 15, 5, ...

Proc#0, at time 0.015600 s, 302 / 475
minimize best cost = 62 : 0, 1, 3, 7, 18, 30, 38, 43, 52, 62, 1, 3, 2, 7, 6, 4,
18, 17, 15, 11, 30, 29, 27, 23, 12, ...
```

```
Proc#0, at time 0.046800 s, 1626 / 2357
minimize best cost = 60 : 0, 1, 3, 11, 17, 29, 36, 51, 56, 60, 1, 3, 2, 11, 10,
8, 17, 16, 14, 6, 29, 28, 26, 18, 12, ...

Proc#0, at time 0.374402 s, 13266 / 19471
minimize best cost = 55 : 0, 1, 6, 10, 23, 26, 34, 41, 53, 55, 1, 6, 5, 10, 9, 4
, 23, 22, 17, 13, 26, 25, 20, 16, 3, ...

        Disolver 3.0, 32bits, 1 thread(s)
        hit limit:      0
        #solution:      10      best bound: 55
        search#0:       BRANCH-AND-BOUND
        heuristics:     LEX+LEX
        time:           3.03125 to first:0      to best:0.75
        #fails:         50652
        #nodes:         71239   (23501.5/sec)
        #restarts:      0
        #constraints:   102
        #variables:     55 (55) 10/45/0Kb
        #references:    0
        trail resizes:  0/0/0/0

Solution: 0 1 6 10 23 26 34 41 53 55
```

You can see that DISOLVER 3.0 outputs the intermediate solutions (call to `unsetSilent()` function). At the end of the computation, it has proved that the best length with 10 marks is 55. Let us remark that this modelling takes advantages of information on smaller OGR sizes in order to improve the lower bounds of distances between marks. You can try the algorithm without this optimization.

	Optimal solution		Proof		
n	*size*	*time*	*#fails*	*overall time*	*#fails*
8	34	0s	80	0.03s	1228
9	44	0.03s	1019	0.20s	7697
10	55	0.43s	13266	1.64s	50652
11	72	1.85s	41141	36.5s	854179
12	85	3.28m	3645884	5.73m	6362607

Table 2.1: (Disolver 2.43) Optimal Golomb Ruler

The table 2.1 presents (with Disolver 2.43) for various n respectively, the optimal size of the ruler, the time to compute the optimal solution, the related backtracking, the overall time (including the proof of optimality) and the overall backtracking. The parallel search section presents some results with multi-threading search (see table 3.3).

2.2.9 Modelling overview

Figure 2.1 presents an overall overview of CP problem modelling. We can see that a CP modelling m must be defined with respect to some problem P. The modelling m is then provided as an input to some search algorithm which interacts with the *Constraints engine*.

A search algorithm can take advantage of various heuristics. DISOLVER 3.0 provides some pre-defined ones (see section 2.3). It is up to the user to make a choice here. More generally, it is sometimes possible to

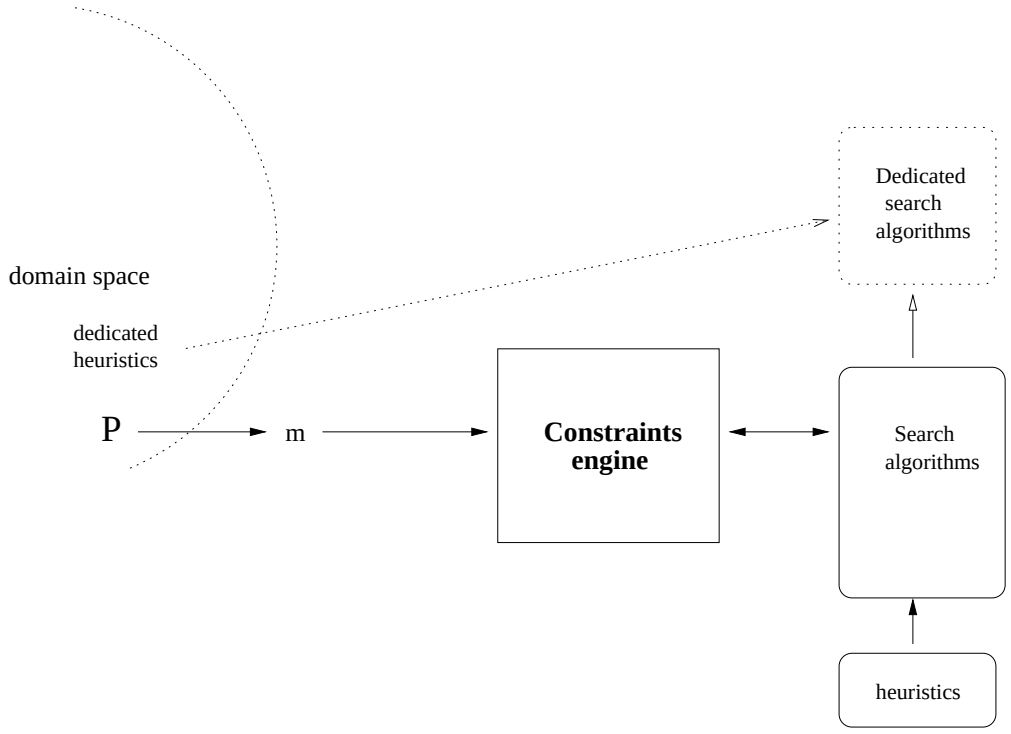

Figure 2.1: Constraint Programming modelling

adapt a heuristic upcoming from the original problem space. This is performed through the definition of a specific solver and will be presented in section 2.3.10.

2.3 Advanced usage

2.3.1 Search control

Disolver 3.0 provides several mechanisms to tune and direct the search algorithms. For example with the previous `solve` function applied to a solver object:

```
int solve();
int solve(int varpolicy, int valpolicy);
int solve(Variable** l, int n);
int solve(Variable** l, int n, int varpolicy, int valpolicy);
```

The `varpolicy` tag corresponds to a variable choice heuristic:

- `LEX`, lexicographical ordering.
- `MINDOM` selects the variable with the smallest domain to instantiate.
- `MAXDEG` selects the variable with the largest degree in the constraint network.
- `DOMDEG` applies the dom/deg heuristic [BR96].
- `DOMWDEG` applies the dom/wdeg heuristic (weighted degree) [BHLS04].

- `BRELAZ` this heuristic applies a mindom fail-first principle which breaks ties by returning the first variable connected to the largest number of unassigned variables in the constraint graph [Bre79].
- `BRELAZR` same as above but returns the first variable connected to the smallest number of unassigned variables [GS97].

The `valpolicy` tag is used for value selection:

- `LEX`, lexicographical ordering.
- `INVLEX`, for reverse lex. ordering.
- `MIDDLE`, middle (or median) value first. This heuristic works only with `CSP` variables. It becomes equivalent to `LEX` when applied to `CP` variables.
- `RANDOM`, selects a value from the domain at random. This heuristic can be combined with the `setBtkLimit` function to implement *random restarts* strategies [GSK98]. The method `solveRestart` implements exactly this (see section 2.3.3).
- `DICHO`, applies the "least commitment" principle through successive dichotomic splits. The heuristic is dynamically switched to a `LEX` ordering whenever the domain size is smaller than the `DICHO_LIMIT` value which is defined in `global.h`.

Disolver 3.0 tracks the definition of each variable in the engine then uses this knowledge to perform backtracking search. However it is possible to restrict backtracking to a peculiar set of the problem's variables or to use a specific ordering of the variables during backtracking search. This is done by constructing a vector of pointers to variables and by passing it to the engine. This is expressed for each search procedure with the first two parameters, l and n.

The previous features are available with the `solveAll` function and with optimization functions:

```
int solveAll();
int solveAll(int varpolicy, int valpolicy);
int solveAll(Variable** l, int n);
int solveAll(Variable** l, int n, int varpolicy, int valpolicy);

int minimize(Variable* obj, int varpolicy, int valpolicy, int bound);
int minimize(Variable** l, int n, Variable* obj, int varpolicy, int valpolicy, int bound);
int maximize(Variable* obj, int varpolicy, int valpolicy, int bound);
int maximize(Variable** l, int n, Variable* obj, int varpolicy, int valpolicy, int bound);
```

2.3.2 Optimistic Partitioning

The previous `minimize` and `maximize` procedures are looking for incremental improvements of the cost function. When a new solution is found with some cost c, the solver searches for a solution with a new cost $c' \leq c - 1$. For the vast majority of applications, this is a suitable approach. However, Disolver 3.0 offers two extra procedures which perform an optimistic partitioning of the cost function in order to speed-up the overall search process. Here, when a solution is found with some cost c, the solver first looks for a solution with a cost $c' \in [c.lb..c.mid]$, where $c.lb$ and $c.mid$ represent respectively the lower bound and the median value of the variable figuring the cost function (the upper bound is $c - 1$). When we apply a maximizing process, things are similar but the solver focuses on the right part of the domain first. The previous partitioning is implemented by the following procedures:

```
int minimizeOpt(Variable* obj, int varpolicy, int valpolicy, int bound);
int minimizeOpt(Variable** l, int n, Variable* obj, int varpolicy, int valpolicy, int bound);
int maximizeOpt(Variable* obj, int varpolicy, int valpolicy, int bound);
int maximizeOpt(Variable** l, int n, Variable* obj, int varpolicy, int valpolicy, int bound);
```

2.3.3 Backtrack search with restarts

Search for a single solution can sometimes benefits from randomization. Indeed, poor heuristics can result in early mistakes which can be very expensive to recover from. Randomized backtracking search is a general technique which can quickly escape from the consequences of early wrong choices [GSK98]. The idea is to perform random probes in the search tree and to stop each probe when some backtrack limit has been reached.

The difficulty with this method is to determine the right backtrack limit. DISOLVER 3.0 implements a variation of random restarts which is able to dynamically revise its limits.

At the beginning, the `setBtkLimit` method can be applied to the solver object in order to set up the initial backtrack limit. A geometric series is then used to increase this limit. The factor of this series is defined through the `setCutoffIncrease` method which has a `double` argument. By default, the increase appears every 5 probes. However a second geometric series can be used to control this value. The method `setRepeatIncrease` specifies this increase. Both series have predefined factors `RESTART_CUTOFF_INCREASE` and `RESTART_REPEAT_INCREASE` which can be accessed through the `global.h` file.

Parameters,

```
solver.setBtkLimit(100);
solver.setCutoffIncrease(1.1);
solver.setRepeatIncrease(1.2);
```

gives the following cutoff serie: 100, 100, 100, 100, 100, 110, 110, 110, 110, 110, 110, 121, 121, 121, 121, 121, 121, 121, etc.

The following functions applied to a solver object implement random restarts:

```
int solveRestart();
int solveRestart(int varpolicy);
int solveRestart(Variable** l, int n);
int solveRestart(Variable** l, int n, int varpolicy);
```

To illustrate the efficiency of the restart algorithm we use the classical *magic square* puzzle (see DISOLVER 3.0's benchmark suite).

		Deterministic		Randomized		
n	*time(s)*	*#fails*	*#choices*	*time(s)*	*#fails*	*#choices*
5	0	190	251	0.04	785	1062
6	13.85	314797	362000	0.1	1378	2093
7	1.25	31240	37062	0.21	3224	4903
8	>5min	-	-	0.04	689	1014
9	>5min	-	-	0.03	281	434
10	>5min	-	-	0.14	1796	2443
11	>5min	-	-	4.28	36475	50980
12	>5min	-	-	7.03	50379	70770
13	>5min	-	-	15.28	94186	131742
14	>5min	-	-	37.29	205502	282605
15	>5min	-	-	46.62	208956	285379

Table 2.2: (Disolver 2.43) Magic square, deterministic v randomized search

Table 2.2 presents in the left part results obtained with the deterministic `solve(MINDOM, MIDDLE)` algorithm while the right part shows results with `solveRestart(MINDOM)`. The randomized search used a starting cutoff set to n^3, a cutoff increase factor of 1.1 and a repeat increase factor of 1.0.

2.3.4 Search with continuations and Quantified CSPs

Monolithic search strategies may not be able to address very large search spaces. One solution is to divide the space into different sub-problems which can benefit from a specific process. This results in a meta-strategy which opportunistically combines low level search on different parts of the space.

DISOLVER 3.0 allows the succession of different strategies on different parts of a search space. For instance, successive tree-based searches can be combined on different sub-set of the variables.

There is no limit to these combinations, each time a search strategy finds a solution (leaf in the tree); it automatically starts the following strategy which can run a similar process and ultimately reach a leaf which will start a third strategy, etc. In the end, the last strategy exhausts its sub-space and backtracks to the previous search process which eventually repeats the same scenario after finding a new solution, etc. One example is presented on figure 2.2 where two tree-based algorithms are applied on different sub-spaces (see [Ham04] for an example of a particular meta strategy).

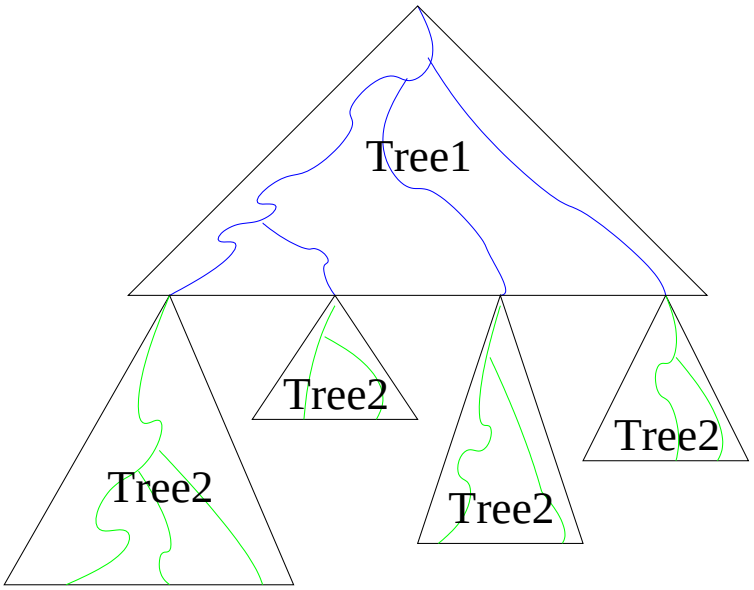

Figure 2.2: Successive tree-based searches

The definition of sub-spaces is accomplished by partitioning the problem's variables. A specific strategy is associated to each partition. The previously defined monolithic strategies are extended with a new parameter which points to a C++ function called each time a solution is found. This function can then start a new strategy which could ultimately repeat a similar process by calling a different function, etc.

It is up to the end-user to define the previous functions which have to correctly call the appropriate strategies. By convention, these functions communicate with the calling strategy through a return value. Returning 0 requires the caller to stop its exploration. For instance in figure 2.2, the function after defining and calling the search process represented by *Tree2* could return 0 and stop the exploration in *Tree1*.

In addition, this mechanism is a simplistic way to solve quantified constraint satisfaction problems (QCSPs) (see [BLV07] for a presentation). Universal quantifiers can be represented by solveAll searches, while existential quantifiers can be represented by solve (possibly with restarts) searches. Both can be combined to figure a QCSP formula.

The following monolithic strategies can all be linked together through their last argument which points to the correct end user function:

```
solve(int (*solutionHandler)());
solve(int varpolicy, int valpolicy, int (*solutionHandler)());
```

```
solve(Variable** l, int n, int (*solutionHandler)());
solve(Variable** l, int n, int varpolicy, int valpolicy, int (*solutionHandler)());

solveRestart(int (*solutionHandler)());
solveRestart(int varpolicy, int (*solutionHandler)());
solveRestart(Variable** l, int n, int (*solutionHandler)());
solveRestart(Variable** l, int n, int varpolicy, int (*solutionHandler)());

solveAll(int (*solutionHandler)());
solveAll(int varpolicy, int valpolicy, int (*solutionHandler)());
solveAll(Variable** l, int n, int (*solutionHandler)());
solveAll(Variable** l, int n, int varpolicy, int valpolicy, int (*solutionHandler)());

minimize(Variable* obj, int varpolicy, int valpolicy, int borne,
 int (*solutionHandler)());
minimize(Variable** l, int n, Variable* obj, int varpolicy, int valpolicy, int borne,
 int (*solutionHandler)());
maximize(Variable* obj, int varpolicy, int valpolicy, int borne,
 int (*solutionHandler)());
maximize(Variable** l, int n, Variable* obj, int varpolicy, int valpolicy, int borne,
 int (*solutionHandler)());

minimizeOpt(Variable* obj, int varpolicy, int valpolicy, int borne,
 int (*solutionHandler)());
minimizeOpt(Variable** l, int n, Variable* obj, int varpolicy, int valpolicy, int borne,
 int (*solutionHandler)());
maximizeOpt(Variable* obj, int varpolicy, int valpolicy, int borne,
 int (*solutionHandler)());
maximizeOpt(Variable** l, int n, Variable* obj, int varpolicy, int valpolicy, int borne,
 int (*solutionHandler)());
```

2.3.5 Generalized arc-consistency

When no particular semantic is known for a constraint, generalized arc-consistency can be applied thanks to the generic Disolver 3.0 `Tuples` constraints. It extensively define the list of allowed or disallowed tuples for a particular combination of variables. It applies a GAC process pretty close to AC-2001 [BR97, BRYZ05].

These constraints are very useful when the problem can be defined by the set of allowed or disallowed combination of values between variables, *e.g.,* configuration problems, data base querying, etc.

```
Constraint* createTuples(&X, &Y);
```

The previous constructor is used to define combinations between two variables.

This constraint can manipulate any type of CP variables. However, the performance should be better with the `CSP` type. Once the combination of variables is defined, the user has to populate the constraint with the list of allowed or disallowed tuples. This starts with the opening of the constraint which defines the type of tuples.

```
void open(int type);
```

Type equal to 0 is used in order to specify disallowed combination of values, 1 is used to specify allowed tuples. It is better to use the first type when the constraint is loose and the second type when it is tight. After the opening of the constraint, tuples of values can be provided through the following method.

```
void addTuple(DISOLVERDOMAIN a, DISOLVERDOMAIN b);
```

Finally, the constraint can be closed through the following method.

```
void close();
```

To illustrate this constraint, we consider a classical benchmark, *random binary CSPs*. The following code generates at random $p_1 \times n(n-1)/2$ binary `Tuples` constraints among n variables with regular domains $[0..d-1]$. Each constraint, disallows $p_2 \times d^2$ pair of values.

```
#include "disolver.h"
int getRand(int r){
  //value in [0 .. r-1]
  return ((float)rand()/RAND_MAX) * r;
}
void random(Problem* p, int n, int d, float p1, float p2, int h){
  Variable** X = p->createCSPVariableVect(n, 0, d - 1);

  int nCtr = n*(n-1)/2*p1;
  int nNeg = d*d*p2;

  //track constraints
  Constraint*** matCtr = new Constraint** [n];
  for(int i = 0; i < n; i++){
    matCtr[i] = new Constraint* [n - 1];
    for(int j = 0; j < i; j++)
      matCtr[i][j] = NULL;
  }
  //track tuples
  int** matTup = new int* [d];
  for(int i = 0; i < d; i++)
    matTup[i] = new int [d];

  for(int c = 0; c < nCtr; c++){
    Constraint* t;
    int r = getRand(n * (n - 1) / 2 - c);
    int rank = -1;
    for(int k = 0; k < n; k++)
      for(int l = 0; l < k; l++){
        rank++;
        if(rank == r)
          if(matCtr[k][l] == NULL){
            t = p->createTuples(X[k], X[l]);
            matCtr[k][l] = t;
          }
          else rank--;
    }
    t->open(0);
    for(int i = 0; i < d; i++)
    for(int j = 0; j < d; j++)
      matTup[i][j] = 0;
    for(int k = 0; k < nNeg; k++){
```

```
      r = getRand(d * d - k);
      rank = -1;
      for(int i = 0; i < d; i++)
        for(int j = 0; j < d; j++){
          rank++;
          if(rank == r)
            if(matTup[i][j] == 0){
              t->addTuple(i, j);
              matTup[i][j] = 1;
          }
          else rank--;
        }
    }
    //t->close();
  }
  //close constraints
  for(int i = 0; i < n; i++)
    for(int j = 0; j < i; j++)
      if(matCtr[i][j] != NULL)
      try{
        matCtr[i][j]->close();
      }
      catch(DisolverException){
        cerr<<"\n no solution!";
        exit(-1);
      }
  if(!p->getMyRank())cout<<"\nnCtr="<<nCtr<<" nNeg="<<nNeg;

  Solver* s = p->createTreeSolver();

  //cout<<"\n SAC="<<p->singletonAC()<<flush;
  //s->unsetSilent();
  int ret = s->solveAll(h, LEX);
  //int ret = s->solve();
  if(ret != -1){
    if(!p->getMyRank()){
      s->print();
      cout<<"\nsolution: ";
      for(int i = 0; i < n; i++) cout<<X[i]->getValue()<<",";
    }
  }
  else cout<<"\n no solution!";
}
int main(int argc, char** argv){
  if(argc != 7){
    cerr<<"\n n, d, p1, p2, seed, heur(mindom, maxdeg, domdeg, brelaz, lex)";
    exit(-1);
  }
  srand(100);
  int n = atoi(argv[1]);
  int d = atoi(argv[2]);
  float p1 = atof(argv[3]);
```

```
  float p2 = atof(argv[4]);
  srand(atoi(argv[5]));
  int h = MINDOM;
  if(strcmp(argv[6], "mindom") == 0) h = MINDOM;
  if(strcmp(argv[6], "maxdeg") == 0) h = MAXDEG;
  if(strcmp(argv[6], "domdeg") == 0) h = DOMDEG;
  if(strcmp(argv[6], "domwdeg") == 0) h = DOMWDEG;
  if(strcmp(argv[6], "brelaz") == 0) h = BRELAZ;
  if(strcmp(argv[6], "lex") == 0) h = LEX;

  Problem p(argc, argv);// init message passing interface
  random(&p, n, d, p1, p2, h);
}
```

2.3.6 Singleton arc-consistency

Disolver 3.0 can compute the singleton arc-consistency (SAC) property [BD05]. Since this computation is quite expensive we provide two methods which can be applied to a problem object.

```
int singletonAC();
int singletonACFull();
```

The first one uses a single pass over the variables, while the second one repeat the process until reaching a fix point. Each method returns -1 when the problem has no solution.

This computation has a polynomial time complexity, however with big domains and/or large number of variables applying SAC can be too time consuming.

2.3.7 More controls

It is possible to bound the exploration achieved during search tree exploration. The following methods can be applied to solver objects.

```
void setBtkLimit(n);
```

The previous method uses an integer which bounds the number of backtracks. In an optimization setting, the solver returns with the best solution found so far.

It is also possible to specify a time limit in milliseconds:

```
void setTimeLimit(1000);//stop search after 1s
```

In branch-and-bound optimization, improvement of the cost function can be very slow. In order to escape from this, Disolver 3.0 provides a control which stops the search when the objective function was not improved for some period of time (in milliseconds again):

```
void setTimeUtilityLimit(1000);//stop branch-and-bound search if no improvement after 1s
```

A function is provided to check afterward if any of the previous limit (backtrack or time) has been hit during search:

```
int getHitLimit();//true if search was interrupted by some limit
```

By default any branch & bound search does not output intermediate solutions. It is possible to change that behaviour by calling the following methods on a solver object:

```
void unsetSilent();
void setSilent();
```

DISOLVER 3.0 can take a snapshot of the state of a problem. This is very useful to record the state of all the variables.

```
void store();
void restore();
```

These methods can be applied to any `Problem` instance. They implement a stack mechanism, *i.e.,* you can take successive snapshots.

2.3.8 Getting some statistics

The following functions are applied to solver objects.

```
int getNbFails();
int getNbChoices();
int getNbSol();
```

These functions return respectively, the number of backtracks, the number of choice nodes and the number of solutions.

```
double getTime();
double getTimeToFirst();
double getTimeToOpt();
```

Return time (CPU+kernel) usage. Respectively, the overall time used by the search process, the time to compute the first solution and to compute the optimal solution (optimization).

```
DISOLVERDOMAIN getFirstBound();
DISOLVERDOMAIN getBestBound();
```

Return respectively the first bound and the best bound (optimization).

2.3.9 Defining new constraints

DISOLVER 3.0 does not provide an easy way to define a new constraint. However, it offers the generic constraint `Tuples` which can be defined through the list of allowed or disallowed combinations of values. This constraint is presented in section 2.3.5. It allows the easy implementation of any domain specific constraint. Its underlying mechanism is made of generalized arc-consistency which guarantees powerful propagations.

2.3.10 Defining new tree search procedures

As presented in section 2.1, some domain space can benefit from dedicated heuristics. It is possible to implement these special heuristics with DISOLVER 3.0 through different mechanisms.

Problem specific static variable ordering There are many ways to define specific search trees with DISOLVER 3.0. The easiest way is to specify an ordered list of variables to some solver and to use the `LEX` variable ordering heuristic. Of course the best static order has to be decided beforehand. The given order will highly depend on the application. However, some general principles exist. The first idea is to start the exploration with the most critical decision variables. The second one is to always relate dependent variables in the ordering. Dependency here is hard to express but usually it can be caught by considering the problem model.

The previous strategy can be used with all the search functions (solve, solveAll, minimize/maximize). This is presented in previous sections through the first two parameters, *e.g.,* `solve(Variable** l, n)`.

Specific solver The most generic way to define problem specific explorations is to derive a subclass from `Solver`. This subclass can be used to override DISOLVER 3.0 pre-built variable and value selection methods; respectively, `Variable* varChoice()` and `DISOLVERDOMAIN valChoice(Variable*)`.

```
class MySolver : public Solver{
 public:
    MySolver(Problem* p): Solver(p){}
    Variable* varChoice(){
     //Input: current state of the exploration
     //Output: the most interesting variable to branch on
    };
    DISOLVERDOMAIN valChoice(Variable* var){
     //Input: var, variable selected by the variable selection heuristic
     //Output: return the most interesting value for var
    };
};
```

Any domain specific dynamic variable and/or value orderings can easily be expressed like that. The variable/value selection methods have to explore the modelling in order to select the most relevant variable/value. Several low level methods can help here:

```
int grounded();
```

Applied to a variable returns 1 if the variable is grounded, *i.e.,* has a value, 0 otherwise. This method can be used in `varChoice`.

```
DISOLVERDOMAIN getLb();
DISOLVERDOMAIN getUb();
```

Applied to a variable, return respectively the lower/upper bounds of the variable. These methods are useful with `valChoice`. Finally, let us remark that the previously defined `store()` and `restore()` methods which apply to a `Problem` object are very relevant here. Indeed, they allow the selection of the best variable and/or value through successive attempts.

Note that all the previous methods can be applied seamlessly to `CSP` objects. Classical variable/value heuristics are still available. This means that one can just define a specific variable or value heuristic and combine it with pre-existing ones, *e.g.,* a specific variable selection combined with `LEX`.

Chapter 3

Parallel search

3.1 Message passing interface

DISOLVER 3.0 uses MPI [MPI94] an external library to perform message passing operations. MPI is a standard library for supercomputing applications which allows a large degree of portability for DISOLVER 3.0. This library is available on a wide variety of architectures from laptop to supercomputers and computational Grids [KTF03].

In order to use the parallel features of DISOLVER 3.0 some MPI `C/C++` library must be correctly installed on your system. Several free or commercial implementations are available for various operating systems. DISOLVER 3.0 has been fully tested with the Microsoft MPI (MS-MPI) toolkit. It has to be installed separately (see http://www.microsoft.com/downloads).

To link your application with the MS-MPI version of the library use the file Disolver32MPI.DLL for 32 bits domains or Disolver64MPI.DLL if you want to use 64 bits encoding.

3.2 MPI setup

MPI uses command line arguments to communicate with the application. The principal information is the number of processes which are going to be part of the parallel computation. DISOLVER 3.0 gathers this information through a transfer of the command line arguments to a `Problem` object.

```
main(int argc, char** argv){
  Problem p(argc, argv);// init message passing interface
    <MIMD parallel computation here>
}
```

The user can apply two functions to a problem object to access this information:

- `int getNbProcs()`, which returns the overall number p of participating processes.
- `int getMyRank()`, which returns the rank of the caller. This rank goes from 0 to $p-1$.

The second method is very useful to filter outputs.

3.3 Parallel tree search

This chapter presents parallel tree search algorithms. These algorithms generalize DISOLVER 3.0's tree search procedures through a divide and conquer approach. These new algorithms use a set of advanced knowledge sharing mechanisms to transparently perform load-balancing and knowledge sharing among sub-trees.

3.3.1 Search space split

Parallel tree search algorithms start with an initial split of the problem's search space. With DISOLVER 3.0 this operation is completely transparent to the user. The library provides two methods to split the search space.

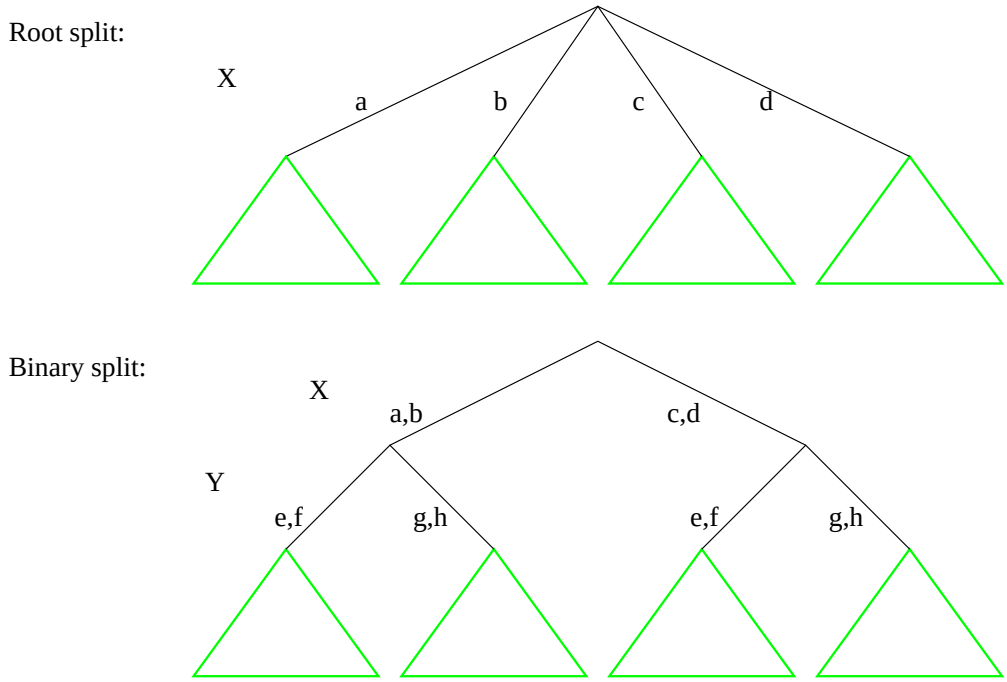

Figure 3.1: DISOLVER 3.0's generic search split algorithms

The first one performs a *root split* of the search space. This operation splits the domain of the best possible variable in p sub-domains. Each sub-domain is then allocated to a distinct process. The *root* variable is selected through some variable ordering heuristic (see section 2.3). This process is presented in the first part of figure 3.1. The example assumes $p = 4$, $X = \{a, b, c, d\}$ is the first variable returned by the variable selection heuristic. The second method performs a *binary split* of the domains of the best $log_2 p$ variables and allocates each sub-space to a distinct process. This process is presented in the second part of figure 3.1. We assume here that $Y = \{e, f, g, h\}$ is the second-best variable according to the heuristic.
DISOLVER 3.0's default policy uses the previous *Root split* process.

3.3.2 Load balancing

This section presents the load balancing algorithm implemented by DISOLVER 3.0. This algorithm is very powerful [RK93] and effectively implemented. It performs the load balancing of sub-trees from loaded processes to idle ones.

Processes can become idle for many reasons. First, different sub-trees may have different complexities. Indeed some sub-trees may be rapidly proved inconsistent while other ones may involve large and deep explorations (*e.g.,* trashing). Second, hardware may be non-uniform or may support different loads in time sharing systems.

DISOLVER 3.0's load balancing algorithm is **transparently** applied in these situations to dynamically re-balance the work among processes. This is presented in figure 3.2 where the second process exhausts its sub-tree. This triggers the selection of some external process for load balancing. A special request (1) is

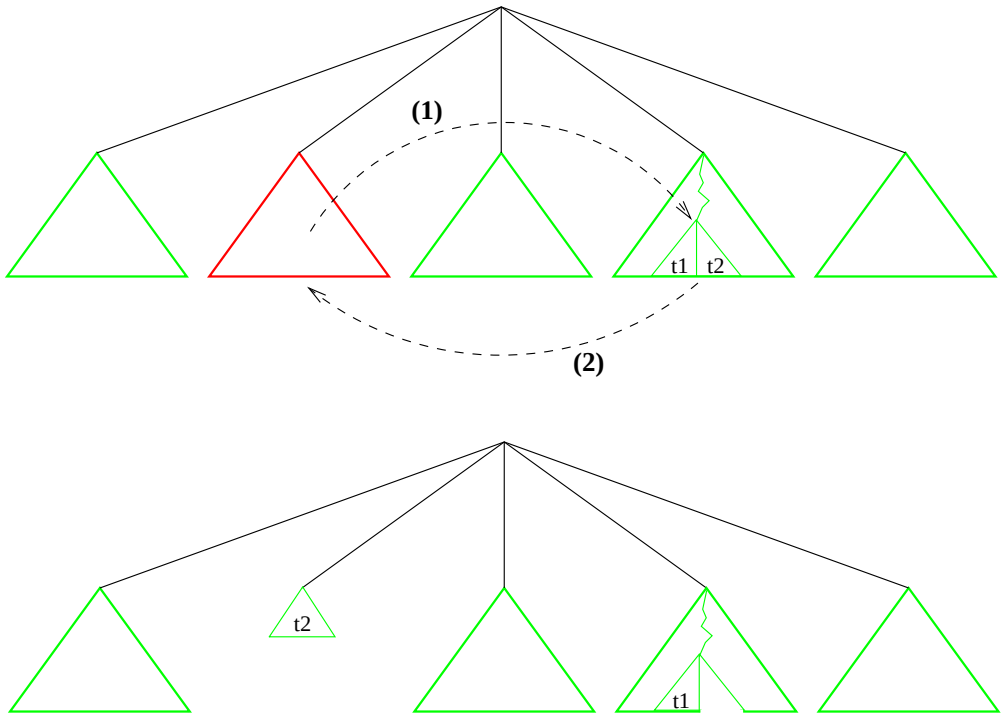

Figure 3.2: DISOLVER 3.0's generic load balancing algorithm

addressed to this process which in turns splits its remaining sub-space in two equal parts (t_1 and t_2). One part is kept for exploration, the other one is exported (2).

DISOLVER 3.0's implements several policies to select the load balancing process.

- `LB_RANDOM`, which randomly select a process.
- `LB_ROUND_ROBIN`, which performs a round-robin among processes: $Proc_0 \rightarrow Proc_1, ..., Proc_{p-1} \rightarrow Proc_0$.
- `LB_ALL`, which interrogates the whole system through $p-1$ requests.
- `LB_LAST_BEST`, this policy is only applicable for optimization. It interrogates the process which provided the most recent improvement of the objective function.

The default policy is `LB_ALL`.

3.3.3 Parallel tree search

As stated before, applying parallel tree search is completely transparent. Indeed, after the previous MPI setup `solve` and `solveAll` algorithms can be called as before. However, let us remark that the selected variable ordering heuristic will directly impact the previous tree-split operations (see section 3.3.1).

We decided to use the *magic square* problem to introduce parallel tree search. A magic square consists of $n \times n$ distinct positive integers such that the sum of the n numbers in any horizontal, vertical, or main diagonal line is always the same magic constant. This problem is excessively hard for CP. It can then benefit from a divide and conquer approach.

Here is the complete source code which is part of the distribution.

```
////////////////////////////////////////////////////////////
// A (normal) magic square consists of the distinct positive
// integers 1, 2, ...,  such that the sum of the n numbers in
// any horizontal, vertical, or main diagonal line is always
// the same magic constant.
////////////////////////////////////////////////////////////

#include "disolver.h"
int N;
Problem* P;
Solver* Solve;
Variable*** Square;
Variable* Sum;

int solutionHandler(){
  // prints solutions, returns 0 to stop enumeration, 1 otherwise
  if(P->getMyRank() == 0){// One node prints the solution (in case of // search)
    cout<<"\n Sum="<<Sum->getValue()<<endl;
    for(int i = 0; i < N; i++){
      for(int j = 0; j < N; j++)
        cout<<"\t"<<Square[j][i]->getValue();
      cout<<endl;
    }
  }

  if(Solve->getNbSol() < 5) return 1;
  else return 0;
}

void magicSquare(int h){
  Solve = P->createTreeSolver();
  int sum = N*(N*N+1)/2;
  Square = P->createCSPVariableMat(N, N, 1, N * N);

  Constraint* d = P->createAllDiff(N * N);
  for(int i = 0; i < N; i++)
    for(int j = 0; j < N; j++)
      d->addVar(Square[i][j]);

  //constraints on the col: Xi1+Xi2+..Xin = sum
  for(int i = 0; i < N; i++){
    Constraint* plus = P->createPlusList(sum, N);
    for(int j = 0; j < N; j++)
      plus->addVar(Square[i][j]);
  }

  //constraints on the line: X1i+X1i+..Xni = sum
  for(int j = 0; j < N; j++){
    Constraint* plus = P->createPlusList(sum, N);
    for(int i = 0; i < N; i++)
```

```
      plus->addVar(Square[i][j]);
  }

  //constraints on first diagonal
  Constraint* plusDiag1 = P->createPlusList(sum, N);
  for(int i = 0; i < N; i++)
    plusDiag1->addVar(Square[i][i]);


  //constraints on second diagonal
  Constraint* plusDiag2 = P->createPlusList(sum, N);
  for(int i = 0; i < N; i++)
    plusDiag2->addVar(Square[i][N-i-1]);


  // symmetry break
  P->createInf(Square[0][0], Square[0][N-1]);
  P->createInf(Square[0][0], Square[N-1][N-1]);
  P->createInf(Square[0][0], Square[N-1][0]);
  P->createInf(Square[0][N-1], Square[N-1][0]);

  Solve->unsetSilent();
  int ret = Solve->solveRestart(MINDOM);

  if(!ret){// One node prints the solution (in case of // search)
    if(P->getMyRank() == 0){
      cout<<"\n Sum="<<sum<<endl;
      for(int i = 0; i < N; i++){
        for(int j = 0; j < N; j++)
          cout<<"\t"<<Square[j][i]->getValue();
        cout<<endl;
      }
      Solve->print();
    }
  }
  //P->print();
}

int main(int argc, char** argv){
  N = 5;
  int h = MIDDLE;
  if(argc > 1) N = atoi(argv[1]);
  if(argc > 2){
    N = atoi(argv[1]);
    if(strcmp(argv[2], "lex") == 0) h = LEX;
    if(strcmp(argv[2], "random") == 0) h = RANDOM;
  }
  P = new Problem(argc, argv);// init message passing interface
  magicSquare(h);
  delete P;
}
```

The call to the search algorithm is performed as for central search. Interestingly we can see that the rank of the caller is used to filter the results. Here the process of rank 0 is selected to output the solution and various statistics on the search process. This filtering is important since by default all the processes share the same output flow.

At the end of a successful search, the p processes record the same solution, *i.e.,* any process can be selected to output the results. In other words, their variables are assigned to the same values.

To access information related to the performance of parallel search, the following methods can be applied to a `Solver` object:

- `getBestMsg()`, returns the number of message passing operations used by the successful process (including operations related to termination detection [CL85].
- `getSystemMsg()`, returns the overall number of message passing operation performed (including operations related to termination detection).
- `getSystemNbFails()`, overall amount of backtracking.
- `getSystemNbChoices()`, overall number of nodes.

		Successful thread		Overall system		
p	*time*	*#fails*	*#choices*	*overall time(s)*	*#fails*	*#choices*
1	2.6h	175M	210M	-	-	-
2	4.47m	4.4M	5.4M	7.63m	8M	9.3M
3	0s	54	109	0.01s	171	307
4	0.03s	504	944	0.19s	2990	3854
5	7.21s	94064	128808	33.26s	529066	632083
6	22.21s	340944	408055	2.06m	2M	2.3M
7	7.07s	94064	128808	49.59s	772746	907800
8	0.28s	3833	5215	2.43s	34330	41689

Table 3.1: (Disolver 2.43)Magic square, $n = 8$, multi-threading

Table 3.1 presents the results for various p with $n = 8$. We can see that this problem exhibits superlinear speed-up. Which is caused by the non-uniform distribution of the solutions in the search space. This is clearly a useful property of parallel tree search [PN88, RK93, Ham02]. The *Successful thread* corresponds to the first process with the longest parallel time.

The previously defined `solveAll` method is generalized to benefit from parallel tree search. At the end of the computation each process stores the overall number of solutions.

Parallel search and the n-queens problem

We present here some multi-threading experiments with the n-queens problem. This problem is especially well suited for parallel search. Indeed, even if solutions can be perceived as regulars, they are not evenly distributed in the search space. This can raise superlinear speed-up for many instances [RK93]. Results are presented in table 3.2

We can remark some super linear speed-up ($t_1/t_p > p$). With $n = 100$, superlinear speed-ups occur up to $p = 5$. With $n = 200$, some solution is quickly found by sequential search and the multi-threading search cannot improve the results. The most interesting situation takes place with $n = 300$, here sequential search takes more than 10 minutes when $p = 1$ and $p = 3$. Interestingly, with $p = 2$ and $p = 4$ the search is

		Successful thread		Overall system		
n	*p*	*time*	*#fails*	*overall time*	*#fails*	*#messages*
200	1	9.34s	66233	-	-	-
	2	0.09s	8	0.15s	8	1
	4	0.10s	1	0.35s	9	3
	8	0.07s	3	0.70s	61	7
	16	0.09s	1	1.03s	20	15
400	1	0.59s	6	-	-	-
	2	0.57s	6	1.35s	96	1
	4	0.62s	27	2.59s	114	3
	8	0.65s	8	5.09s	114	7
	16	0.65s	8	10.39s	253	15
800	1	>5min	-	-	-	-
	2	>5min	-	-	-	-
	4	4.06s	4	17.59s	35	3
	8	3.95s	4	34.89s	110	7
	16	4.25s	0	70.67s	236	15

Table 3.2: (Disolver 2.43) n-queens, multi-threading with a (MINDOM, LEX) strategy

backtrack-free. The speed-ups are then strictly superlinear.

To close this section we can stress that multi-threading or parallel search can have large benefits, especially when the sequential search use a poorly informed heuristic for value section.

3.3.4 Parallel branch & bound

As for search, optimization benefits from parallelism without any change to the modelling. For example in the Golomb Ruler model presented in 2.2.8, we just have to add MPI parameters to the problem object.

	Optimal solution		Proof		
p	*time*	*#fails*	*time*	*#fails*	
1	0.78s	13266	2.92s	50652	-
2	0.53s	6526	1.96s	51658	1443
3	0.67s	7611	1.79s	62289	3317
4	0.43s	6044	1.64s	65089	7195
5	0.59s	5255	1.73s	66943	12219
10	1.28s	5639	2.34s	92021	22813
20	0.45s	2620	4.5s	76090	49982

Table 3.3: (Disolver 2.43) Optimal Golomb Ruler, $n = 10$, multi-threading

Multi-threading results with various p are presented in Table 3.3. From left to right we have, the parallel time to find the optimal solution and the corresponding backtracking effort. The right part presents the parallel time and the overall backtracking and message passing efforts.

Bibliography

[BD05] C. Bessiére and R. Debrune. Optimal and suboptimal singleton arc consistency algorithms. In *IJCAI*, 2005.

[BHLS04] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *ECAI*, pages 146–150, 2004.

[BHZ06] L. Bordeaux, Y. Hamadi, and L. Zhang. Propositional satisfiability and constraint programming: A comparative survey. *ACM Computing Survey*, 9(2):135–196, 06.

[BLV07] Marco Benedetti, Arnaud Lallouet, and Jérémie Vautard. Qcsp made practical by virtue of restricted quantification. In Manuela M. Veloso, editor, *IJCAI*, pages 38–43, 2007.

[BR96] C. Bessière and J. C. Régin. MAC and combined heuristics: two reasons to forsake FC (and CBJ?) on hard problems. In *Principles and Practice of Constraint Programming*, pages 61–75, 1996.

[BR97] C. Bessire and J.C. Rgin. Arc consistency for general constraint networks: preliminary results. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, San Francisco, August23–29 1997. Morgan Kaufmann Publishers.

[Bre79] D. Brelaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22:251–256, 1979.

[BRYZ05] C. Bessiere, J.C. Rgin, R.H.C. Yap, and Y. Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165:165–185, 2005.

[CL85] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *TOCS*, 3(1):63–75, Feb 1985.

[GS97] C. P. Gomes and B. Selman. Algorithm portfolio design: Theory vs. practice. In *Proceedings of UAI-97*, Providence, RI., USA, 1997.

[GSK98] C.P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proc. AAAI-98, Madison, WI*, 1998.

[Ham02] Y. Hamadi. Interleaved backtracking in distributed constraint networks. *Int. J. on Artif. Intelligence Tools (IJAIT)*, 11(4):167–188, 2002.

[Ham04] Y. Hamadi. Cycle-cut decomposition and log-based reconciliation. In *ICAPS, W. Connecting Planning Theory with Practice*, pages 30–35, 2004.

[Hen89] P. V. Hentenryck. *Constraint Satisfaction in Logic Programming.* The MIT Press, 1989.

[KTF03] N. Karonis, B. Toonen, and I. Foster. Mpich-g2: A grid-enabled implementation of the message passing interface. *Journal of Parallel and Distributed Computing (JPDC)*, 63(5):551–563, May 2003.

[LOQTvB03] Alejandro López-Ortiz, Claude-Guy Quimper, John Tromp, and Peter van Beek. A fast and simple algorithm for bounds consistency of the alldifferent constraint. In *IJCAI*, pages 245–250, 2003.

[MPI94] Message Passing Interface Forum MPIF. MPI: A message-passing interface standard. *Int. Journal of Supercomputer Applications*, 8(3/4), 1994.

[PN88] E. A. Pruul and G. L. Nemhauser. Branch-and-bound and parallel computation: A historical note. *Operations Research Letters*, 7(2), April 1988.

[RK93] V. N. Rao and V. Kumar. On the efficiency of parallel backtracking. *IEEE Transactions on Parallel and Distributed Systems*, 4(4):427–437, Apr 1993.

[She] J. B. Shearer. Shearer's main golomb page. http://www.research.ibm.com/ people/s/shearer/comcomp.html.

Appendix A

Exit codes

When Disolver 3.0 runs out of memory or when some constraint is not properly applied by the end user, the solver interrupts the application (it throws a `DisolverException` object). This appendix provides a list of the error codes with a quick description of the related problem.

- `-100`, this error is returned by a constraint which cannot initialize itself. This happens when initial constraint propagations are inconsistent. There are two possible explanations. Either the instance is unsatisfiable, and this fact is discovered without any explicit search process. Either there is a problem in the problem definition. Since constraint propagation starts with constraint definitions, you should always guard the definition of your problem by a try/catch construct able to catch that exception.
- `-101`, returned when the memory is exhausted. Add more RAM or simplify your modelling, for instance by turning space consuming `CSP` variables into `CP` ones or memory consuming `PlusList` constraints into `Sum` ones.

Index