iZ-C Reference Manual

version 3.5

NTT DATA SEKISUI SYSTEMS

Dec 19, 2017

Contents:

CHAPTER 1

Preface

iZ is an efficient and extendable library for Constraint Solving. iZ is dedicated to expressing declaratively and solving complex combinatorial problems such as resource allocation, scheduling, planning or production control.

This manual is the reference of iZ-C (C language version of iZ), describing all the primitive functions available.

iZ-C is available from:

http://www.constraint.org/ja/izc_download.html

CHAPTER 2

Execution Environment

2.1 Initialize and Finalize

cs_init() must be called before iZ-C API functions. And *cs_end()* must be called when before program is terminated.

void **cs_init**()
: Set up internal memory for iZ-C.

void **cs_end**()
: Memory that is implicitly allocated (CSint variable creation, constraint declaration, ...) between *cs_init()* and cs_end() is freed when cs_end() is called.

  CSint variables should not be accessed after cs_end().

2.2 Error status

iZ-C library has a variable to hold error status. Library user can check error by reading this variable. (This varibale is called *err* in this manual.)

void **cs_initErr**()
: Internal *err* variable is initialized by calling cs_initErr(). (the value of *err* is set to CS_ERR_NONE).

  cs_initErr() is called implicitely by *cs_init()*.

int **cs_getErr**()
: Returns current value of *err*. If it returns *CS_ERR_NONE*, it means there has been no error raised between the last cs_getErr() or *cs_init()* call and this cs_getErr() call.

  err can be set following values:

  CS_ERR_NONE
  : No error has been occured.

CS_ERR_GETVALUE
cs_getValue() is called for non-instantiated variable.

CS_ERR_OVERFLOW
Overflow has been occured.

CS_ERR_NO_MEMORY
Cannot allocate memory.

void **cs_setErr** (int *code*)
Set value of *err* to *code*.

2.3 Error handler

Function can be registed to handle error. Registered function will be called after *err* is set when error has occured.

void **cs_setErrHandler** (int *code*, void (**func*)(void* data, void* ext), void* *ext*)

Register function *func* as error handler of *code*. Parameter *data* will have diffrent value as *code* (see following table) and parameter *ext* is same value as cs_setErrHandler is called.

Code	Data	Default action
CS_ERR_GETVALUE	Pointer to CSint	Print message and process is continued.
CS_ERR_OVERFLOW	NULL	Do nohitng and process is continued.
CS_ERR_NO_MEMORY	NULL	Call abort() (process will be terminated)

2.4 Version Information

const char* **cs_getVersion** ()
Returns version string of iZ-C. (ex: "3.5.0")

This string is equivarent to which is concatinated these following values using ".".

IZ_VERSION_MAJOR
Major version of iZ-C

IZ_VERSION_MINOR
Minor version of iZ-C

IZ_VERSION_PATCH
Patch level of iZ-C

2.5 Output of Statistics Information

void **cs_printStats** ()
Print statistcs values:

- Nb Fails (the number of fails that occurred during the generation process)
- Nb Choice Points (the number of instantiations on which it is possible to backtrack)
- Heap Size (the size of the heap which saves the context to enable backtrack).

Nb Fails and Nb Choice Points can be accessed directly using the *cs_getNbFails()* and *cs_getNbChoicePoints()*.

void **cs_fprintStats** (FILE *f)

It is similar to cs_printStats() except that it takes a pointer to FILE as an argument, and writes to the indicated file when the function is invoked.

CHAPTER 3

Constructors and Basic functions

A CSint variable is an integer domain variable. Only pointers to CSint (CSint*) are used in the pre-defined iZ functions.

3.1 Constructors

CSint variables can be constructed by these following functions:

CSint ***cs_createCSint** (int *min*, int *max*)
: Creates a CSint variable whose domain is { *min* .. *max* }.

CSint ***cs_createNamedCSint** (int *min*, int *max*, char **name*)
: Creates a CSint variable whose domain is { *min* .. *max* }, and associates a name to it. (It is equivalent to `cs_createCSint()` followed by `cs_setName()`.)

CSint ***CSINT** (int *n*)
: Creates an already instantiated (i.e. its domain is reduced to one element) CSint variable. This function is equivalent to cs_createCSint(*n*, *n*).

CSint ***cs_createCSintFromDomain** (int **array*, int *size*)
: Creates a CSint variable its domain is defined by the *array* (of *size* integers).

CSint ****cs_createCSintArray** (int *nbVars*, int *min*, int *max*)
: Creates an array of *nbVars* CSint variables, which all have the same { *min* .. *max* } domain.

  <note>This function returns pointer to array of CSint pointers. Don't call free() for this array because array is managed by iZ-C.

3.2 Functions to Access Domain Variable

Following basic functions give access to CSint (a domain variable):

int **cs_getMin** (CSint **vint*)
: Returns the minimum value of the domain of *vint*.

int **cs_getMax** (CSint *vint*)
Returns the maximum value of the domain of *vint*.

int **cs_getNbElements** (CSint *vint*)
Returns the number of elements in the domain of *vint*.

int **cs_getNbConstraints** (CSint *vint*)
Returns the number of constraints *vint* is involved in. (Constraints which failed when setting are not included.)

char ***cs_getName** (CSint *vint*)
Returns the name of *vint* (which has been set using *cs_setName()*). The name of a CSint variable can also be displayed by using the "%T" format conversion string of the *cs_printf()* or *cs_fprintf()* functions.

int **cs_getNextValue** (CSint *vint*, int *val*)
Returns the first element in the domain of *vint* which is strictly greater than *val*. If *val* is greater than cs_getMax(*vint*), then it returns INT_MAX (the maximum value of an int).

In the following code, cs_getNextValue() is used as an expression of a 'for' statement:

```
void display(CSint *vint)
{
  int val;
  for (val = cs_getMin(vint); val <= cs_getMax(vint); val = cs_getNextValue(vint,␣
↪val))
      printf("%d ", val);
}
```

The display() function defined above will print on the stdout file all the possible values of a CSint variable (i.e. its domain) in increasing order.

int **cs_getPreviousValue** (CSint *vint*, int *val*)
Returns the first element in the domain of *vint* which is strictly lower than *val*. If *val* is lower than cs_getMin(*vint*), then it returns INT_MIN (the minimum value of an int).

int ***cs_getDomain** (CSint *vint*)
Returns an array of cs_getNbElements(*vint*) elements whose values are the elements of the domain of vint. (Returned array should be 'free'ed by user)

void **cs_freeDomain** (int* *array*)
Free *array* allocated by *cs_getDomain()*.

IZBOOL char **cs_isIn** (CSint *vint*, int *val*)
Returns TRUE if *val* is an element of the domain of *vint*. Otherwise it returns FALSE.

void **cs_setName** (CSint *vint*, char *name*)
Associate a *name* to a CSint variable *vint*. When the *vint* will be displayed (using *cs_printf()* or *cs_fprintf()*), its associated name can also be displayed (if the format "%T" is specified).

IZBOOL **cs_isFree** (CSint *vint*)
Returns TRUE if *vint* is not yet instantiated (i.e. cs_getNbElements(*vint*) > 1, i.e. cs_getMin(*vint*) < cs_getMax(*vint*)). Otherwise returns FALSE.

IZBOOL char **cs_isInstantiated** (CSint *vint*)
Returns TRUE if *vint* is instantiated (i.e. cs_getNbElements(*vint*) == 1, i.e. cs_getMin(*vint*) == cs_getMax(*vint*)). Otherwise returns FALSE.

int **cs_getValue** (CSint *vint*)
Returns the value of *vint* in case *vint* has been instantiated. An error occurs (i.e. a call to *cs_getErr()* returns CS_ERR_GETVALUE) if *vint* has not been instantiated yet, and cs_getValue(*vint*) returns cs_getMin(*vint*).

void **cs_printf** (const char **control*, ...)
Is similar to the standard printf() C function except that three new conversion characters dedicated to CSint variables and CSint variable arrays has been added:

- `cs_printf("%T", vint)` prints the name of vint (if it has one) followed by its domain.
- `cs_printf("%D", vint)` prints only the domain of vint on the standard output file, stdout.
- `cs_printf("%A", array, size)` prints the array of CSint variables (with their name, if any) array, of size elements, separated by a comma.

void **cs_fprintf** (FILE **f*, const char **control*, ...)
It is similar to cs_printf() except that it takes a pointer to FILE as an argument, and writes to the indicated file when the function is invoked.

3.3 Constraints for Domain

Following functions are constraints, and may fail (i.e. return a FALSE value) when posted:

IZBOOL **cs_InArray** (CSint **vint*, int **array*, int *size*)
Constrains the domain of CSint variable *vint* to be the domain contained in the *array* .

IZBOOL **cs_NotInArray** (CSint **vint*, int **array*, int *size*)
Constrains the CSint variable *vint* not to have the values in the *array* . All the values of the domain of *vint* that are in array are removed.

It could be defined as:

```
IZBOOL cs_notInArray(CSint *vint, int *array, int size) {
  int i;
  for (i = 0; i < size; i++)
   if (!cs_NEQ(vint, array[i])
     return FALSE;

  return TRUE;
}
```

IZBOOL **cs_InInterval** (CSint **vint*, int *min*, int *max*)
Constrains the CSint variable *vint* to be { *min* .. *max* } (i.e. cs_getMin(*vint*) >= min and cs_getMax(*vint*) <= max).

It could be defined as:

```
IZBOOL cs_InInterval(CSint *vint, int min, int max) {
  return(cs_GE(vint, min) && cs_LE(vint, max));
}
```

IZBOOL **cs_NotInInterval** (CSint **vint*, int *min*, int *max*)
Constrains the CSint variable *vint* not to have the values in { *min* .. *max* }. All the values of the domain of *vint* that are in {min..max} are removed.

It could be defined as:

```
IZBOOL cs_NotInInterval(CSint *vint, int min, int max) {
  int i;
  for (i = min; i <= max; i++)
    if (!cs_NEQ(vint, i))
      return FALSE;
```

```
  return TRUE;
}
```

CHAPTER 4

Arithmetic Constraints

These constraints enable the user to create a new CSint variable defined by arithmetic relations between already existing CSint variables. These constraints never fails when the user posts it.

CSint ***cs_Add** (CSint **vint1*, CSint **vint2*)
Returns a CSint variable defined as the sum of *vint1* and *vint2* .

CSint ***cs_VAdd** (int *nbVars*, CSint **vint*, ...)
cs_VAdd() accepts a variable number of arguments (number specified by the first argument *nbVars*).

It returns a CSint variable defined as the sum of the CSint variables given in arguments.

`cs_VAdd(n, V1, V2, ..., Vn)` = V1 + V2 + ... + Vn

CSint ***cs_Sub** (CSint **vint1*, CSint **vint2*)
Returns a CSint variable defined as the difference between *vint1* and *vint2* .

CSint ***cs_VSub** (int *nbVars*, CSint **vint*, ...)
cs_VSub() accepts a variable number of arguments (number specified by the first argument *nbVars*).

It returns a CSint variable defined as the difference between the first argument and the sum of all the others.

`cs_VSub(n, V1, V2, V3, ..., Vn)` = V1 - V2 - V3 - ... - Vn

CSint ***cs_Mul** (CSint **vint1*, CSint **vint2*)
Returns a CSint variable defined as the product of *vint1* and *vint2* .

CSint ***cs_VMul** (int *nbVars*, CSint **vint*, ...)
cs_VMul() accepts a variable number of arguments (number specified by the first argument *nbVars*).

It returns a CSint variable defined as the product of the CSint variables given in arguments.

`cs_VMul(n, V1, V2, ..., Vn)` = V1 * V2 * ... * Vn

CSint ***cs_Div** (CSint **vint1*, CSint **vint2*)
Returns a CSint variable defined as the quotient of *vint1* and *vint2* .

CSint ***cs_VDiv** (int *nbVars*, CSint **vint*, ...)
cs_VDiv() accepts a variable number of arguments (number specified by the first argument *nbVars*).

CSint ***cs_Sigma**(CSint ***array*, int *size*)
Returns a CSint variable constrained to be equal to the sum of the *size* CSint variables referenced by *array*.

CSint ***cs_ScalProd**(CSint ***array*, int **vector*, int *size*)
Returns a CSint variable constrained to be equal to the scalar product of the vector of CSint variables *array* and the vector of integer *vector* (both of *size* elements).

CSint ***cs_VScalProd**(int *nbVars*, CSint **vint*, ...)
cs_VScalProd() accepts a variable number of arguments (number specified by the first argument *nbVars*).

cs_VScalProd(n, V1, V2, . . . , Vn, c1, c2, . . . , cn) returns the scalar product of the vector of CSint variables (V1, V2, . . . , Vn) and the vector of integer (c1, c1, . . . , cn).

CSint ***cs_Abs**(CSint **vint*)
Returns a CSint variable constrained to be equal to the absolute value of the CSint variable *vint* .

CHAPTER 5

Relation Constraints

The following functions set constraints between already existing CSint variables These functions may fail (i.e. return a FALSE value) when set.

IZBOOL **cs_Le**(CSint **vint1*, CSint **vint2*)
vint1 must be lower than or equal to vint2 (i.e. cs_getMin(vint1) <= cs_getMin(vint2) and cs_getMax(vint1) <= cs_getMax(vint2)).

IZBOOL **cs_Ge**(CSint **vint1*, CSint **vint2*)
vint1 must be greater than or equal to vint2.

IZBOOL **cs_Lt**(CSint **vint1*, CSint **vint2*)
vint1 must be strictly lower than vint2.

IZBOOL **cs_Gt**(CSint **vint1*, CSint **vint2*)
vint1 must be strictly greater than vint2.

IZBOOL **cs_Eq**(CSint **vint1*, CSint **vint2*)
vint1 and vint2 are constrained to be equal.

IZBOOL **cs_Neq**(CSint **vint1*, CSint **vint2*)
vint2 and vint2 are constrained not to be equal.

The following similar functions are useful when one of the two operands is a constant:

- `IZBOOL cs_LE(CSint *vint, int val)`
- `IZBOOL cs_GE(CSint *vint, int val)`
- `IZBOOL cs_LT(CSint *vint, int val)`
- `IZBOOL cs_GT(CSint *vint, int val)`
- `IZBOOL cs_EQ(CSint *vint, int val)`
- `IZBOOL cs_NEQ(CSint *vint, int val)`

For example, cs_LE() could be defined as:

```
IZBOOL cs_LE(CSint *vint, int val) {
  return(cs_Le(vint, CSINT(val));
}
```

IZBOOL **cs_AllNeq**(CSint ***array*, int *size*)
The CSint variables of array must have different values. It could be written like:

```
IZBOOL cs_AllNeq(CSint **array, int size) {
  int i, j;
  for (i = 0; i < size - 1; i++)
    for (j = i + 1; j < size; j++)
      cs_Neq(array[i], array[j]);
}
```

The following functions return CSint variable which indicates whether relation constraint is satisfied or not. (satisfied = 1, NOT satisfied = 0)

CSInt* **cs_ReifLe**(CSint **vint1*, CSint **vint2*)
Returns a CSint variable which idicates whether vint1 is lower than or equal to vint2 by value 1 and 0.

CSInt* **cs_ReifGe**(CSint **vint1*, CSint **vint2*)
Returns a CSint variable which idicates whether vint1 is greater than or equal to vint2 by value 1 and 0.

CSInt* **cs_ReifLt**(CSint **vint1*, CSint **vint2*)
Returns a CSint variable which idicates whether vint1 is lower than vint2 by value 1 and 0.

CSInt* **cs_ReifGt**(CSint **vint1*, CSint **vint2*)
Returns a CSint variable which idicates whether vint1 is greater than vint2 by value 1 and 0.

CSInt* **cs_ReifEq**(CSint **vint1*, CSint **vint2*)
Returns a CSint variable which idicates whether vint1 is equal to vint2 by value 1 and 0.

CSInt* **cs_ReifNeq**(CSint **vint1*, CSint **vint2*)
Returns a CSint variable which idicates whether vint1 is different from vint2 by value 1 and 0.

The following functions are useful when second CSint variable is a constant:

- `CSint* cs_ReifLE(CSint *vint, int val)`
- `CSint* cs_ReifGE(CSint *vint, int val)`
- `CSint* cs_ReifLT(CSint *vint, int val)`
- `CSint* cs_ReifGT(CSint *vint, int val)`
- `CSint* cs_ReifEQ(CSint *vint, int val)`
- `CSint* cs_ReifNEQ(CSint *vint, int val)`

CHAPTER 6

High-level Constraints

These following constraints are more complex ones. They could be written using the previous primitives and *demon* (see *Demon*).

Constraint functions which return CSint variable can return NULL to indicate fail.

IZBOOL **cs_IfEq** (CSint **vint1*, CSint **vint2*, int *val1*, int *val2*)
If *vint1* is instantiated to the value *val1* , then *vint2* will be instantiated to *val2* .

If *vint2* is instantiated to the value *val2* , then *vint1* will be instantiated to *val1* .

IZBOOL **cs_IfNeq** (CSint **vint1*, CSint **vint2*, int *val1*, int *val2*)
The couple of CSint variables (*vint1* , *vint2*) cannot have the values (*val1* , *val2*).

CSint ***cs_Occur** (CSint **vint*, int *val*, CSint ***array*, int *size*)
The value *val* must appear *vint* times in the *array* .

This function is deprecated and will be deleted in future version. Please use `cs_OccurConstraints()`.

CSint ***cs_OccurDomain** (int *val*, CSint ***array*, int *size*)
Returns CSint variable which counts the value *val* in the *array*.

IZBOOL **cs_OccurConstraints** (CSint **vint*, int *val*, CSint ***array*, int *size*)
The value *val* must appear *vint* times in the *array*.

CSint ***cs_Index** (CSint ***array*, int *size*, int *val*)
Returns CSint variable *index* which is constrainted *array* [*index*] = *val*.

i ∈ domain(*index*) ⇔ val ∈ domain(*array* [*i*])

CSint ***cs_Element** (CSint **index*, int **values*, int *size*)
Returns CSint variable *element* which is constrainted *element* = *values* [*index*].

CSint ***cs_VarElement** (CSint **index*, CSint ***array*, int *size*)
Returns CSint variable *element* which is constrainted *element* = *array* [*index*].

CSint ***cs_Min** (CSint ***array*, int *size*)
Returns a CSint variable equal to the minimum of the CSint variables referenced by *array* .

CSint ***cs_VMin** (int *nbVars*, CSint **vint*, ...)
cs_VMin() accepts a variable number of arguments (number specified by the first argument *nbVars*).

CSint ***cs_Max** (CSint ***array*, int *size*)
Returns a CSint variable equal to the maximum of the CSint variables referenced by *array* .

CSint ***cs_VMax** (int *nbVars*, CSint **vint*, ...)
cs_VMax() accepts a variable number of arguments (number specified by the first argument *nbVars*).

CHAPTER 7

Generation Mechanisms and Heuristics

After the constraints have been posted, one may want to try to find a solution compatible with these constraints.

The basic principle of generation is:

1. Find a free CSint variable *var* among *allvars*.
2. Select a value *val* in the domain of *var* and try to instantiate: `cs_EQ(var, val)`.

   If it fails, then set `cs_NEQ(var, val)` and go to step 2.

   If it succeeds, then go to step 1.

7.1 Basic Functions for Generation

IZBOOL **`cs_search`** (CSint ***allvars*, int *nbVars*, CSint* (***findFreeVar*)(CSint **allvars, int nbVars))
: This function tries to instantiate all the CSint variables referenced by *allvars*. It may fail if there is no solution.

  The *findFreeVar* function finds a non-instantiated CSint variable among *allvars*.

IZBOOL **`cs_Vsearch`** (int *nbVars*, CSint **vint*, ...)
: cs_Vsearch() accepts a variable number of arguments (number specified by the first argument *nbVars*).

7.2 Pre-defined Choice Functions

CSint ***`cs_findFreeVar`** (CSint ***allvars*, int *nbVars*)
: The first non-instantiated CSint variable in *allvars* is returned.

CSint ***`cs_findFreeVarNbElements`** (CSint ***allvars*, int *nbVars*)
: The non-instantiated CSint variable which has the lowest number of elements in its domain is returned.

CSint ***`cs_findFreeVarNbElementsMin`** (CSint ***allvars*, int *nbVars*)
: Among the CSint variables which all have the same lowest number of elements, it returns the one with the lowest minimum domain.

CSint ***cs_findFreeVarNbConstraints** (CSint ***allvars*, int *nbVars*)
The non-instantiated CSint variable which is involved in the greatest number of constraints is returned.

For example, cs_findFreeVarNbElements() could be defined as:

```
CSint *cs_findFreeVarNbElements(CSint **allvars, int nbVars) {
  int i;
  int nbElements, nbElementsOpt;
  CSint *varOpt = NULL;

  nbElementsOpt = INT_MAX;
  for (i = 0; i < nbVars; i++) {
    nbElements = cs_getNbElements(allvars[i]);
    if ((nbElements > 1) && (nbElements < nbElementsOpt)) {
      nbElementsOpt = nbElements;
      varOpt = allvars[i];
    }
  }
  return(varOpt);
}
```

7.3 Number of Fails and Choice Points

One can define heuristics that may depend on the number of Fails or the number of Choice Points that have occured, using these functions:

int **cs_getNbFails** ()
Returns the number of Fails that have occured until this call. Functions that manage this parameter are:

- *cs_search()*
- *cs_searchFail()*
- *cs_Vsearch()*
- *cs_searchCriteria()*
- *cs_searchCriteriaFail()*
- *cs_searchMatrix()*
- *cs_searchMatrixFail()*
- *cs_findAll()*
- *cs_minimize()*

int **cs_getNbChoicePoints** ()
Returns the number of Choice Points that have been raised until this call. Functions that manage this parameter are same as *cs_getNbFails()*.

7.4 Sophisticated Generation Functions

IZBOOL **cs_searchCriteria** (CSint ***allvars*, int *nbVars*, int (**findFreeVar*)(CSint **allvars, int nbVars), int (*criteria)(int index, int val))
In *cs_search()*, when a CSint variable *var* has been chosen, the values are tried in increasing order (from `cs_getMin(var)` to `cs_getMax(var)`, avoiding all the values which are not in the domain of *var*).

cs_searchCriteria() enables to control this selection order. The value which minimize *criteria()* will be selected first.

IZBOOL **cs_searchMatrix**(CSint ***matrix*, int *NbRows*, int *NbCols*, int (**findFreeRow*)(CSint ***matrix, int NbRows, int NbCols), int (*findFreeCol)(int row, CSint **Row, int NbCols), int (*criteria)(int row, int col, int val))

cs_search() and *cs_searchCriteria()* are used to generate a vector of CSint variables.

cs_searchMatrix() does the same for a matrix. *findFreeRow()* returns the index of the Row the user wants to chose. After a row has been chosen, *findFreeCol()* returns the index of the Col to be chosen. After a CSint variable (Col, Row) has been chosen, a value which minimize *criteria()* is selected.

IZBOOL **cs_searchFail**(CSint **allvars*, int *nbVars*, CSint* (**findFreeVar*)(CSint **allvars, int nbVars), int *NbFailsMax*)

cs_searchFail() is similar to *cs_search()* except that it stops, and return FALSE (i.e. fails) if the number of backtracks reaches *NbFailsMax*.

One can check if the fail has occured because of the number of backtracks by checking NbFails (cf. *cs_getNbFails()* function).

IZBOOL **cs_searchCriteriaFail**(CSint **allvars*, int *nbVars*, int (**findFreeVar*)(CSint **allvars, int nbVars), int (*criteria)(int index, int val), int *NbFailsMax*)

cs_searchCriteriaFail() is similar to *cs_searchCriteria()* except that it stops, and return FALSE (i.e. fails) if the number of backtracks reaches *NbFailsMax*.

One can check if the fail has occured because of the number of backtracks by checking *NbFails* (cf. *cs_getNbFails()* function).

IZBOOL **cs_searchMatrixFail**(CSint ***matrix*, int *NbRows*, int *NbCols*, int (**findFreeRow*)(CSint ***matrix, int NbRows, int NbCols), int (*findFreeCol)(int row, CSint **Row, int NbCols), int (*criteria)(int row, int col, int val), int *NbFailsMax*)

cs_searchMatrixFail() is similar to *cs_searchMatrix()* except that it stops, and return FALSE (i.e. fails) if the number of backtracks reaches *NbFailsMax*.

One can check if the fail has occured because of the number of backtracks by checking *NbFails* (cf. *cs_getNbFails()* function).

IZBOOL **cs_findAll**(CSint **allvars*, int *nbVars*, CSint* (**findFreeVar*)(CSint **allvars, int nbVars), void (*found)(CSint **allvars, int nbVars))

Search for all possible solutions. Each time a solution is found, the *found()* function is called.

Typically, the *found()* function can be used to display the solutions:

```
void found(CSint **allvars, int nbVars) {
  static NbSolutions = 0;
  printf("Solution %d\n", ++NbSolutions);
  cs_printf("%A\n\n", allvars, nbVars);
}
```

IZBOOL **cs_minimize**(CSint **allvars*, int *nbVars*, CSint* (**findFreeVar*)(CSint **allvars, int nbVars), CSint *cost, void (*found)(CSint **allvars, int nbVars, CSint *cost))

Try to find a solution minimizing the cost. At each step of the minimization, the *found()* function is called.

Typically, the found function can be used to display the current solution:

```
void found(CSint **allvars, int nbVars, CSint *cost) {

  static NbSolutions = 0;

  printf("Solution %d\n", ++NbSolutions);
  cs_printf("%A\n", allvars, nbVars);
```

```
  cs_printf("Cost = %T\n\n", cost);
}
```

cs_minimize() could be defined as:

```
IZBOOL cs_minimize(CSint **allvars, int nbVars, CSint* (*findFreeVar)(CSint␣
↪**allvars, int nbVars),  CSint *cost, void (*found)(CSint **allvars, int nbVars,
↪ CSint *cost)) {
  char gFirst, g;
  gFirst = g = cs_search(allvars, nbVars, findFreeVar);
  while (g) {
    int currentCost = getMin(cost);
    found(allvars, nbVars, cost);
    cs_restoreAll();
    g = (cs_LT(cost, currentCost) && cs_search(allvars, nbVars, findFreeVar));
  }
  return gFirst;
}
```

CHAPTER 8

Demon

During constraint propagation, some events (i.e. changes in CSint variables domains) will occur on CSint variables. These events are categorized into four exclusive types:

Known: the domain has been reduced to one value (the CSint variable is instantiated);

Ex.: `cs_EQ(vint, 0); /* raise a Known evnent */`

NewMin: the minimum value of the domain has been changed (became strictly greater);

Ex.: `cs_GT(vint, 0);`

NewMax: the maximum value of the domain has been changed (became strictly lower);

Ex: `cs_LT(vint, 0);`

Neq: one value has been removed from the domain;

Ex: `cs_NEQ(vint, 0);`

These events are exclusive, that is:

- a known event will not raise neither a newMin, newMax, nor a neq event;
- newMin nor newMax events will not raise any neq event;

For example,

```
CSint *vint = cs_createCSint(0, 10);
cs_GE(vint, 10);
```

will raise a known event (*vint* will be instantiated to 10) but not a newMin event.

```
CSint *vint = cs_createCSint(0, 10);
cs_NEQ(vint, 0);
```

will raise a newMin event (*vint* will be strictly greater than 1), but not a neq event.

IZBOOL **cs_eventAllKnown** (CSint **_array_, int _size_, IZBOOL (*_allKnown_)(CSint **array, int size, void *extra), void *_extra_)

The function *allKnown()* will be called when all the CSint variables referenced by *array* are instantiated (i.e. known).

In the case where all the CSint variables referenced by *array* are instantiated when cs_eventAllknown() is called, it returns the result of the *allKnown()* call, otherwise it returns TRUE.

The *extra* argument is a pointer to a void variable that can be used freely.

Each time *allKnown()* is called, this *extra* value is passed.

IZBOOL **cs_eventKnown** (CSint **_array_, int _size_, IZBOOL (*_known_)(int val, int index, CSint **array, int size, void *extra), void *_extra_)

The function *known()* will be called when an *array* [*index*] CSint variable is instantiated.

The *extra* argument is a pointer to a void variable that can be used freely.

Each time *known() is called, this *extra* value is passed.

void **cs_eventNewMin** (CSint **_array_, int _size_, IZBOOL (*_newMin_)(CSint *vint, int index, int oldMin, CSint **array, int size, void *extra), void *_extra_)

The function *newMin()* will be called each time when the minimum value of *array* [*index*] CSint variable is increased.

The *extra* argument is a pointer to a void variable that can be used freely.

Each time *newMin()* is called, this *extra* value is passed.

void **cs_eventNewMax** (CSint **_array_, int _size_, IZBOOL (*_newMax_)(CSint *vint, int index, int oldMax, CSint **array, int size, void *extra), void *_extra_)

The function *newMax()* will be called each time when the maximum value of *array* [*index*] CSint variable is decreased.

The *extra* argument is a pointer to a void variable that can be used freely.

Each time *newMax()* is called, this *extra* value is passed.

void **cs_eventNeq** (CSint **_array_, int _size_, IZBOOL (*_neq_)(CSint *vint, int index, int neqValue, CSint **array, int size, void *extra), void *_extra_)

The function *neq()* will be called each time when a value is removed from the *array* [*index*] CSint variable.

The *extra* argument is a pointer to a void variable that can be used freely.

Each time *neq()* is called, this *extra* value is passed.

void **cs_backtrack** (CSint *_vint_, int _i_, void (*_backtrack_)(CSint *vint, int i))

The function *backtrack()* is called when context is restored.

vint and *i* given to cs_backtrack are passed to *backtrack()*.

CHAPTER 9

Context

The following functions give an access to the heap where all the information on the events that had occurred on all the CSint variables created.

By events we mean changes in the domains of the CSint variables (like known, newMin. newMax, neq) caused by Constraint Propagation or constraint settings.

These functions will be used to create your own generation function, but are not needed when the primitives like *cs_search()* are used.

int **cs_saveContext**()
: Saves the context, i.e. the domains and the constraints of all the CSint variables that are available when the call is made.

  It returns an integer *label* used to call functions such as *cs_forgetSaveContextUntil()*, *cs_acceptContextUntil()* or *cs_restoreContextUntil()* take an integer *label* parameter.

void **cs_forgetSaveContext**()
: Cancels the last *cs_saveContext()* call. This functions should be used instead of *cs_acceptContext()* inside cs_event* functions.

void **cs_restoreAndSaveContext**()
: The context is restored as same as before the last *cs_saveContext()* call, and *cs_saveContext()* is called again.

void **cs_acceptContext**()
: All the changes made on the CSint variables since the last *cs_saveContext()* call are accepted (i.e., this context cannot be restored any more). This function frees the memory of the heap since the last *cs_saveContext()* call.

  As it cuts possible backtrack until the last *cs_saveContext()* call, *cs_acceptContext()* should not be called inside a cs_event* functions.

void **cs_acceptAll**()
: All the changes made on the CSint variables since the first *cs_saveContext()* call are accepted. This function frees all the memory of the heap.

void **cs_restoreContext** ()
The context is restored as same as before the last *cs_saveContext()* call.

void **cs_restoreAll** ()
The context is restored as same as before the first *cs_saveContext()* call.

These following functions take an integer *label* as an argument. This *label* must be an integer returned by *cs_saveContext()*.

void **cs_forgetSaveContextUntil** (int *label*)
Cancels the all the cs_saveContext() calls until the one referenced by *label*.

void **cs_acceptContextUntil** (int *label*)
All the changes made on the CSint variables since the *cs_saveContext()* call referenced by *label* are accepted (i.e., this context cannot be restored any more). This function frees the memory of the heap since the *cs_saveContext()* call referenced by *label*.

void **cs_restoreContextUntil** (int *label*)
The context is restored as same as before the *cs_saveContext()* call referenced by *label*.

Ex: the cs_search() procedure could be implemented as follows:

```
IZBOOL cs_search(CSint **allvars, int nbVars, CSint* (*findFreeVar)(CSint **allvars,␣
↪int nbVars)) {
  CSint *var = findFreeVar(allvars, nbVars);
  if (var) {
    int val;
    cs_saveContext();

    for (val = cs_getMin(var); val <= cs_getMax(var); val = cs_getNextValue(var,␣
↪val)) {
      if (cs_EQ(var, val) && cs_search(allvars, nbVars, findFreeVar))
        return TRUE;

      cs_restoreAndSaveContext();
    }

    cs_acceptContext();
    return FALSE;
  }
  else {
    return TRUE;
  }
}
```

Index

C

I