Constraint Programming

iZ-C Tutorial

version 3.7

NTT DATA SEKISUI SYSTEMS

Nov 20, 2020

CONTENTS:

CHAPTER

ONE

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 tutorial is an introduction to Constraint Solving, with some code samples written using iZ-C (C language version of iZ).

iZ-C is available from:

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

CHAPTER

TWO

INITIALIZATION AND FINALIZATION

Internal memories and structures must be initialized before all function calls of iZ-C. And memories should be freed at end of program.

These processes will be done by `cs_init()` and `cs_end()` called like following:

```
#include <stdio.h>
#include "iz.h"

int main(void)
{
    cs_init();

    // iZ-C functions should be here.

    cs_end();

    return 0;
}
```

CHAPTER

THREE

MODELING WITH IZ-C

3.1 Domain Variable (CSint type)

The Constraint Solving approach to solve a problem is based on two principles:

1. Modeling the problem as a set of variables, and a set of constraints over these variables. A solution for the problem will be found when the values (or possible values) for the variables will respect all the constraints.
2. Solving the problem is based on what is called the Constraint Propagation mechanism. It means that during the Tree-Search, constraints are used actively as soon as possible to detect inconsistencies, and to eliminate variable values which cannot occur in the solution of the problem.

At the step of modeling the problem, we don't need to care about how it will be solved. Our purpose at this level is just to express declaratively the problem, in terms of variables (the unknowns of the problem) and constraints over these variables (i.e. the restrictions over the possible values of the variables that must be satisfied).

With iZ-C, variables are called CSint variables, or integer domain variables. The set of possible values of a CSint variable is called its *domain*.

Examples of CSint and its domain: (cs_init and cs_end are omitted in these examples)

Code:

```
{
  CSint *vint = cs_createNamedCSint(0, 10, "vint"); // Create a CSint variable having␣
↪domain
  cs_printf("%T\n", vint);                          // between 0 and 10 and print it.
}
```

Output:

```
vint: {0..10}
```

Code:

```
{
  int domain[9] = {-2, 0, 1, 2, 3, 5, 6, 7, 10};     // Create a CSint variable having
  CSint *vint = cs_createCSintFromDomain(domain, 9); // domain defined by array␣
↪domain[9]
  cs_printf("%T\n", vint);                          // and print it.
}
```

Output:

```
{-2, 0..3, 5..7, 10}
```

3.2 Alphametic

Let us take the "Alphametic" as an example, where the problem consists in the following cryptogram where each letterhas to be replaced by a number between 0 and 9 such as the multiplication is correct, and where each integer between 0 and 9 is used exactly twice.

```
    A B C   (L1)
 *  D E F   (L2)
 ----------
    G H I   (L3)
  J K L     (L4)
M N O       (L5)
-----------
P Q R S T   (L6)
```

1. The solution of our problem will be the values of the CSint variables A, B, . . . , T.
2. The constraints between these variables are:

- a). L6 = L1 * L2
- b). L3 = F * L1
- c). L4 = E * L1
- d). L5 = D * L1
- e). L6 = 100 * L5 + 10 * L4 + L3
- f). A $\neq$ 0, D $\neq$ 0, G $\neq$ 0, J $\neq$ 0, M $\neq$ 0, P $\neq$ 0
- g). Each integer between 0 and 9 must appear exactly twice in {A, B, . . . , T}.

Where L1, L2, . . . , L6 are defined as:

- d1). L1 = 100 * A + 10 * B + C;
- d2). L2 = 100 * D + 10 * E + F;
- d3). L3 = 100 * G + 10 * H + I;
- d4). L4 = 100 * J + 10 * K + L;
- d5). L5 = 100 * M + 10 * N + O;
- d6). L6 = 10000 * P + 1000 * Q + 100 * R + 10 * S + T.

Let us express this problem using iZ-C:

```
#include "iz.h"

#define NB_DIGITS 20
CSint **Digit;  // Digit is an array of 20 CSint variables
CSint *L1, *L2, *L3, *L4, *L5, *L6;

// Digit[a] = A, Digit[b] = B, ..., Digit[t] = T
enum {a = 0, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t};

void constraints()
{
  int val;

  Digit = cs_createCSintArray(NB_DIGITS, 0, 9);
```

(continues on next page)

(continued from previous page)

```
  L1 = cs_VScalProd(3, Digit[a], Digit[b], Digit[c], 100, 10, 1);  // (d1)
  L2 = cs_VScalProd(3, Digit[d], Digit[e], Digit[f], 100, 10, 1);  // (d2)
  L3 = cs_VScalProd(3, Digit[g], Digit[h], Digit[i], 100, 10, 1);  // (d2)
  L4 = cs_VScalProd(3, Digit[j], Digit[k], Digit[l], 100, 10, 1);  // (d3)
  L5 = cs_VScalProd(3, Digit[m], Digit[n], Digit[o], 100, 10, 1);  // (d4)
  L6 = cs_VScalProd(5, Digit[p], Digit[q], Digit[r], Digit[s], Digit[t], // (d5)
                    10000, 1000, 100, 10, 1);

  cs_Eq(L6, cs_Mul(L1, L2));        // (a)
  cs_Eq(L3, cs_Mul(Digit[f], L1));  // (b)
  cs_Eq(L4, cs_Mul(Digit[e], L1));  // (c)
  cs_Eq(L5, cs_Mul(Digit[d], L1));  // (d)
  cs_Eq(L6, cs_VScalProd(3, L5, L4, L3, 100, 10, 1)); //(e)

  // (f)
  cs_NEQ(Digit[a], 0);
  cs_NEQ(Digit[d], 0);
  cs_NEQ(Digit[g], 0);
  cs_NEQ(Digit[j], 0);
  cs_NEQ(Digit[m], 0);
  cs_NEQ(Digit[p], 0);

  // (g)
  for (val = 0; val <= 9 ; val++) {
      cs_OccurConstraints(CSINT(2), val, Digit, NB_DIGITS);
  }
}
```

CHAPTER

FOUR

TREE-SEARCH AND CONSTRAINT PROPAGATION

After the problem has been expressed using domain variables and constraints, we have to find values (or sometimes only sets of possible values) for these variables. This is achieved by the Tree-Search process associated to Constraint Propagation.

This process is performed as follows:

Level n:

1. Choose a domain variable var which has not been instantiated yet. If there is no uninstantiated variable, then the problem is solved;
2. Set a Choice Point, i.e. remember the state of all the domain variables at this step;
3. Try to instantiate the domain variable var to a selected value val (which belong to its domain);
4. Check the result of instantiation of variable:
    - If instantiation has failed (i.e. if Constraint Propagation has found an inconsistency), then return to the most recent Choice Point, and restore the state of all the domain variables at the step the Choice Point was set (this "return-and-restore" process is called Backtrack).
    - If instantiation has failed and all possible values for var have already been tried, then Backtrack to the Choice point of the level (n - 1). (If n = 1, then it means that the problem itself has no solution).
    - Otherwise perform the next level (n + 1).

In fact this basic Tree-Search algorithm itself is implemented in C using iZ-C in 15 lines of code (cf. Reference Manual, "Context" section).

In the Tree-Search process, Constraint Propagation occurs at step (3), when a value for a domain variable is tried. This Constraint Propagation mechanism is triggered by a domain variable var when its domain changes. Then a domain variable V_i linked to var through a constraint $C_{i,j}$ will have its domain modified according to the new domain of var and the constraint $C_{i,j}$

As V_i itself may trigger Constraint Propagation again, this mechanism is recursive.

For example, let us consider two domain variables A and B, having their initial domain set to 1..8, and consider the constraint:

- A: {1..8}
- B: {1..8}
- A = B + 2

After this constraint is posted:

- A: {3..8}
- B: {1..6}

Let us modify the domain of A:

- $A \neq 5$

Then Constraint Propagation will give: $B \neq 3$

If we modify the domain of B:

- $B \leq 3$

Then Constraint Propagation will give: $A \leq 5$

Using iZ-C, the code and output would be:

Code:

```
#include "iz.h"

int main(int argc, char **argv)
{
  CSint *A, *B;

  cs_init();

  A = cs_createNamedCSint(1, 8, "A");
  B = cs_createNamedCSint(1, 8, "B");

  cs_printf("%T\n%T\n", A, B);

  cs_Eq(A, cs_Add(B, CSINT(2)));
  cs_printf("\nAfter 'A = B + 2':\n%T\n%T\n", A, B);

  cs_NEQ(A, 5);
  cs_printf("\nAfter 'A != 5':\n%T\n%T\n", A, B);

  cs_LE(B, 3);
  cs_printf("\nAfter 'B <= 3':\n%T\n%T\n", A, B);

  cs_end();

  return 0;
}
```

output:

```
A: {1..8}
B: {1..8}

After 'A = B + 2':
A: {3..8}
B: {1..6}

After 'A != 5':
A: {3, 4, 6..8}
B: {1, 2, 4..6}

After 'B <= 3':
A: {3, 4}
B: {1, 2}
```

Actually, Constraint propagation is performed automatically with iZ-C, when we set constraints (cs_Add constraint in the example) or when some changes occurs in the domain of CSint variables.

In iZ-C, a pre-defined functions integrates the Tree-Search mechanism explained before. `cs_search()` is one of these functions takes a *findFreeVar()* function as an argument, which enable to have control on the order in which CSint variables are chosen (cf step (1) of the Tree-Search process).

To illustrate the `cs_search()` function, let us go back to the "Alphametic".

Code:

```
void printSolution(CSint **allvars, int nbVars)
{
  cs_printf("  %D\n", L1);
  cs_printf("* %D\n", L2);
  cs_printf("-----\n");
  cs_printf("  %D\n", L3);
  cs_printf(" %D \n", L4);
  cs_printf("%D  \n", L5);
  cs_printf("-----\n");
  cs_printf("%D  \n", L6);
  cs_printStats();
}

int main(int argc, char **argv)
{
  cs_init();

  constraints();
  cs_search(Digit, NB_DIGITS, cs_findFreeVarNbElements);
  printSolution(Digit, NB_DIGITS);

  cs_end();

  return 0;
}
```

Output:

```
  179
* 224
-----
  716
 358
358
-----
40096

Nb Fails = 114
Nb Choice Points = 181
Heap Size = 1024
```

The printed solution is found within 114 Backtracks (number of fails) and 181 Choice Points.

The `cs_findFreeVarNbElements()` function, passed as an argument of `cs_search()` is a pre-defined iZ-C function that returns the non instantiated CSint variable which has the smallest domain, i.e. the lowest number of elements. This strategy is often a good one, but other strategies can be implemented by rewriting your own *findFreeVar()* function.

One may want to know if this solution is the only one. This new problem can be solved using the `cs_findAll()`

function instead of `cs_search()`.

Code:

```
int main(int argc, char **argv)
{
  cs_init();

  constraints();
  cs_findAll(Digit, NB_DIGITS, cs_findFreeVarNbElements, printSolution);
  cs_printStats();

  cs_end();

  return 0;
}
```

Output:

```
  179
* 224
-----
  716
 358
358
-----
40096

Nb Fails = 114
Nb Choice Points = 181
Heap Size = 1024

Nb Fails = 1436
Nb Choice Points = 2199
Heap Size = 2048
```

1436 backtracks are necessary to find that the "Alphametic" has only one solution. The fourth argument of `cs_findAll()` is a function that will be called each time a solution is found.

As we have seen with this small example, Constraint Solving is consists from three steps:

1. Declare the domain variables
2. Express the constraints on these variables;
3. Find values for these variables compatible with the constraints.

iZ-C provides:

- CSint variable for step 1.
- Many pre-defined constraints for step 2. And also provides ways of defining new customized constraints (cf. *Define a New Constraint using Demons*)
- Pre-defined Tree-Search functions for step 3. And also provides ways of defining new customized heuristics (cf. Reference Manual, "Heuristics and Generation Mechanisms", "Context").

CHAPTER

FIVE

DEFINE A NEW CONSTRAINT USING DEMONS

Demons are useful to define new constraints.

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.

For example, here is the code of the well-known N-Queens problem, implemented using `cs_eventKnown()`.

(The 8-Queens problem consists in positioning 8 queens on a chess board in such a way that no queen is on the same horizontal, vertical or diagonal line as another.)

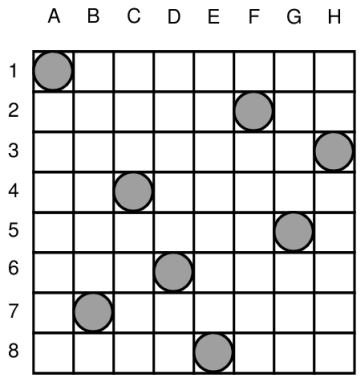
When a queen is positioned (i.e. when a CSint variable allvars[i] is known), then we have to express that no other queen should be on the same row nor diagonal.

```
#include <stdlib.h>
#include "iz.h"

// This function is called when allvars[index] is instantiated
// It returns FALSE if Constraints Propagation fails
IZBOOL knownQueen(int val, int index, CSint **allvars, int NbQueens, void *extra)
{
  int i;

  for (i = 0; i < NbQueens; i++) {
    if (i != index) {
      CSint *var = allvars[i];

      if (!cs_NEQ(var, val)) return FALSE;  // No queens on the same row
      if (!cs_NEQ(var, val + index - i)) return FALSE; //No queens on the same␣
↪diagonal
      if (!cs_NEQ(var, val + i - index)) return FALSE;
    }
  }
  return TRUE;
}

int main(int argc, char **argv)
{
  int NbQueens = (argc > 1) ? atoi(argv[1]) : 8;
  CSint **allvars;

  cs_init();

  allvars = cs_createCSintArray(NbQueens, 1, NbQueens);

  cs_eventKnown(allvars, NbQueens, knownQueen, NULL);
  cs_search(allvars, NbQueens, cs_findFreeVarNbElementsMin);
  cs_printf("%A\n", allvars, NbQueens);

  cs_printStats();

  cs_end();

  return 0;
}
```

The corresponding output is:

```
1, 7, 4, 6, 8, 2, 5, 3

Nb Fails = 11
Nb Choice Points = 20
Heap Size = 1024
```

Solution “1, 7, 4, 6, 8, 2, 5, 3” is visualized as following:



The 8-Queens problem has 92 solutions, which can be found using

```
cs_findAll(allvars, NbQueens, cs_findFreeVarNbElementsMin, found)
```

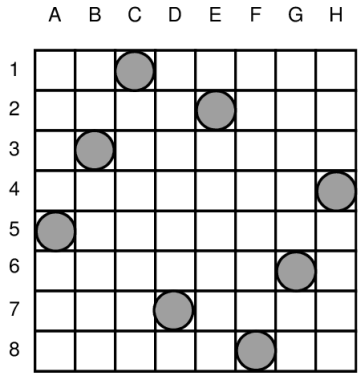
instead of

```
cs_search(allvars, NbQueens, cs_findFreeVarNbElementsMin);
```

found() being defined as:

```
void found(CSint **allvars, int NbQueens)
{
  static NbSolutions = 0;

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

The 92 solutions are found within 289 backtracks. We can notice the 55th one, which is interesting: 5, 3, 1, 7, 2, 8, 6, 4



All the cs_event() functions take as their third argument the reference to the function that will called when the event will be triggered. These functions (allKnown(), known(), newMin(), newMax(), neq()) must return TRUE or FALSE.

CHAPTER

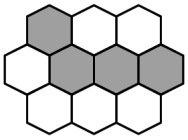
SIX

EXAMPLES

We describe in this Section some examples of toy-problems that can be solved using Constraint Solving, and give the corresponding code in iZ-C.

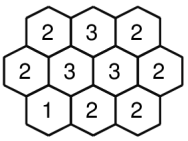
6.1 Image Decoding

Let us consider the following grid, where cells can be white or black:

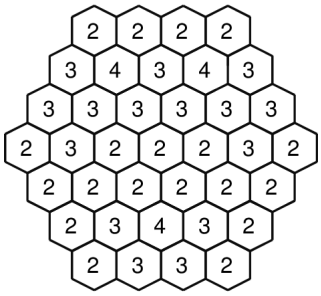


This grid can be coded in writing in each cell the number of black neighbor cells (a cell is considered as a neighbor of itself).

For example, the coding for the grid above would be:



The problem consists in finding the original image(s) whose coding is given below:



Model:

Each cell of the grid will be described by a CSint variable whose possible values are 0 and 1. For each cell of the grid referenced by the index i, we will post a constraint:

```
cs_EQ(getSum(i, cell), Code[i]);
```

Which means that for a given cell referenced by i, the sum of the neighbor CSint variables must be equal to the Code[i].

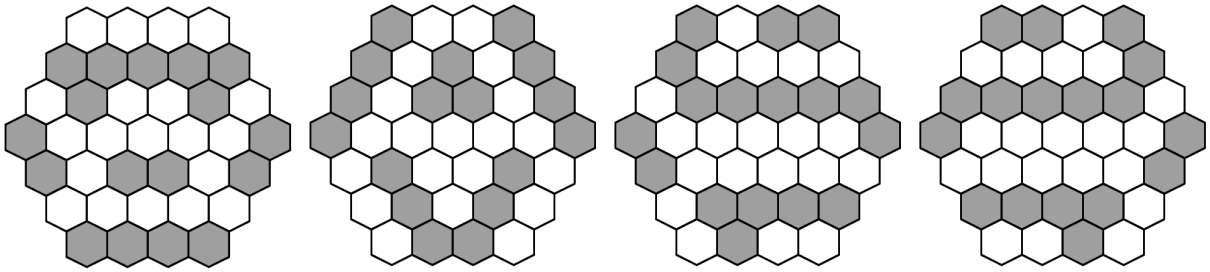
getSum() is a user-defined function using the pre-defined cs_Sigma() constraint, returns the sum of the neighbors.

For the tree-search, we will use the pre-defined cs_findAll() function that search for all the solutions:

```
cs_findAll(cell, NB_CELLS, cs_findFreeVar, printSolution);
```

`cs_findAll()` is similar to `cs_search()` except that it takes as its fourth argument a function which will be called every time a solution is found.

iZ-C finds the four solution, and proves there is no more, within 9 backtracks. The four solutions are:



The iZ-C code and the corresponding output is given below:

Code:

```
#include <stdlib.h>
#include <time.h>
#include "iz.h"

#define NB_CELLS 77
#define N -1

int Code[NB_CELLS] = {N,N,N,N,N,N,N,N,N,
                        N,N,2,2,2,2,N,N,
                      N,N,3,4,3,4,3,N,N,
                        N,3,3,3,3,3,3,N,
                      N,2,3,2,2,2,3,2,N,
                        N,2,2,2,2,2,2,N,
                      N,N,2,3,4,3,2,N,N,
                        N,N,2,3,3,2,N,N,
                      N,N,N,N,N,N,N,N,N};

void printCells(CSint **cell, int n, int nbCells)
{
  int i;

  for (i = 0; i < nbCells; i++) {
    if (Code[i + n] != N)
      printf("%d ", cs_getValue(cell[i + n]));
    else
      printf("  ");
  }
  printf("\n");
}
```

(continues on next page)

(continued from previous page)

```
void printSolution(CSint **cell, int nbCells)
{
  int i;
  int n = 9;
  static int NbSolutions = 0;

  printf("\nSolution %d:\n", ++NbSolutions);
  for (i = 1; i < 8; i++)
    if (i % 2) {
      printf(" ");
      printCells(cell, n, 8);
      n += 8;
    }
    else {
      printCells(cell, n, 9);
      n += 9;
    }
}

CSint *getSum(int i, CSint **c)
{
  CSint **array = (CSint**) malloc(7 * sizeof(CSint*));
  int n = 0;

  array[n++] = c[i];
  if (Code[i + 1] != N) array[n++] = c[i + 1];
  if (Code[i + 9] != N) array[n++] = c[i + 9];
  if (Code[i + 8] != N) array[n++] = c[i + 8];
  if (Code[i - 1] != N) array[n++] = c[i - 1];
  if (Code[i - 9] != N) array[n++] = c[i - 9];
  if (Code[i - 8] != N) array[n++] = c[i - 8];

  return(cs_Sigma(array, n));
}

int main(int argc, char **argv)
{
  int i;
  clock_t t0 = clock();
  CSint **cell;

  cs_init();

  cell = cs_createCSintArray(NB_CELLS, 0, 1);

  for (i = 0; i < NB_CELLS; i++)
    if (Code[i] == N)
      cs_EQ(cell[i], 0);
    else
      cs_EQ(getSum(i, cell), Code[i]);

  if (!cs_findAll(cell, NB_CELLS, cs_findFreeVar, printSolution))
    printf("No solution\n");

  cs_printStats();
  printf("Elapsed Time = %fs\n", (double) (clock() - t0) / CLOCKS_PER_SEC);
```

(continues on next page)

(continued from previous page)

```
  cs_end();

  return 0;
}
```

Output:

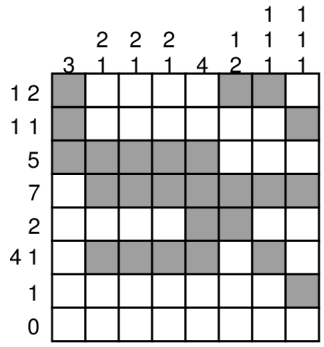
```
Solution 1:
     0 0 0 0
    1 1 1 1 1
   0 1 0 0 1 0
  1 0 0 0 0 0 1
   1 0 1 1 0 1
    0 0 0 0 0
     1 1 1 1

Solution 2:
     1 0 0 1
    1 0 1 0 1
   1 0 1 1 0 1
  1 0 0 0 0 0 1
   0 1 0 0 1 0
    0 1 0 1 0
     0 1 1 0

Solution 3:
     1 0 1 1
    1 0 0 0 0
   0 1 1 1 1 1
  1 0 0 0 0 0 1
   1 0 0 0 0 0
    0 1 1 1 1
     0 1 0 0

Solution 4:
     1 1 0 1
    0 0 0 0 1
   1 1 1 1 1 0
  1 0 0 0 0 0 1
   0 0 0 0 0 1
    1 1 1 1 0
     0 0 1 0

Nb Fails = 9
Nb Choice Points = 24
Heap Size = 1024
Elapsed Time = 0.000587s
```

It is interesting to note that for a given coding, there may be several corresponding solutions. If we change slightly the coding system (if the cell itself is now not considered as its neighbor), then one coding will give only one solution on a 37 cells grid.

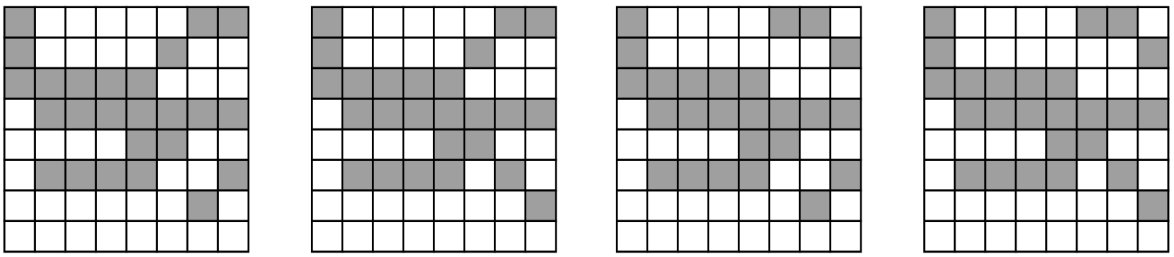
6.2 Multi Magic Square

The problem consists in finding the original image(s) when the sum of consecutive pixels are known .

Example :



There are in fact four solutions for this specific problem, found in 89 backtracks:



The iZ-C code and the corresponding output is given below:

Code:

```
#include <stdlib.h>
#include <time.h>
#include "iz.h"

#define N 8

int row_1[] = {2, 1, 2};
int row_2[] = {2, 1, 1};
int row_3[] = {1, 5};
int row_4[] = {1, 7};
int row_5[] = {1, 2};
int row_6[] = {2, 4, 1};
int row_7[] = {1, 1};
int row_8[] = {1, 0};

int col_1[] = {1, 3};
int col_2[] = {2, 2, 1};
int col_3[] = {2, 2, 1};
int col_4[] = {2, 2, 1};
int col_5[] = {1, 4};
```

(continues on next page)

(continued from previous page)

```
int col_6[] = {2, 1, 2};
int col_7[] = {3, 1, 1, 1};
int col_8[] = {3, 1, 1, 1};

int *row[N] = {row_1,
               row_2,
               row_3,
               row_4,
               row_5,
               row_6,
               row_7,
               row_8};

int *col[N] = {col_1,
               col_2,
               col_3,
               col_4,
               col_5,
               col_6,
               col_7,
               col_8};

struct Tarray {
  int *_array;
  int _arraySize;
};

int getIndex(int index, CSint **tint, int size)
{
  int i = 0;

  index--;
  while ((index >= 0) && cs_isInstantiated(tint[index]) && (cs_getValue(tint[index])␣
↪== 1))
    index--;

  if (index < 0)
    return i;

  if (cs_isFree(tint[index]))
    return -1;

  while (index >= 0) {
    while ((index >= 0) && cs_isInstantiated(tint[index]) && (cs_
↪getValue(tint[index]) == 0))
      index--;

    if (index < 0)
      return i;

    if (cs_isFree(tint[index]))
      return -1;

    index--;
    i++;

    while ((index >= 0) && cs_isInstantiated(tint[index]) && (cs_
↪getValue(tint[index]) == 1))
```

(continues on next page)

(continued from previous page)

```
      index--;

    if (index < 0)
      return i;

    if (cs_isFree(tint[index]))
      return -1;

    index--;
  }

  return i;
}

IZBOOL known(int val, int index, CSint **tint, int size, void *Sarray)
{
  int *array = ((struct Tarray*) Sarray)->_array;
  int arraySize = ((struct Tarray*) Sarray)->_arraySize;

  if (val == 1) {
    int i = getIndex(index, tint, size);
    int j, nGoal, nCons;

    if (i < 0)
      return TRUE;

    if (i >= arraySize)
      return FALSE;

    nGoal = array[i];
    nCons = 1;
    j = index - 1;

    while ((j >= 0) && (cs_getValue(tint[j]) == 1)) {
      j--;
      nCons++;
    }

    j = index + 1;

    while ((j < size) && cs_isInstantiated(tint[j]) && (cs_getValue(tint[j]) == 1)) {
      j++;
      nCons++;
    }

    if (nCons > nGoal)
      return FALSE;

    if (nCons == nGoal)
      if (j < size)
        return(cs_EQ(tint[j], 0));

    if (nCons < nGoal) {
      while ((j < size) && (nCons < nGoal)) {
        if (!cs_EQ(tint[j], 1))
          return FALSE;
```

(continues on next page)

(continued from previous page)

```
        j++;
        nCons++;
      }

      if (nCons != nGoal)
        return FALSE;

      if (j < size)
        return(cs_EQ(tint[j], 0));
    }
  }

  return TRUE;
}

int Mcons(CSint **tint, int size, int *array, int arraySize)
{
  struct Tarray *Sarray = (struct Tarray*) malloc(sizeof(struct Tarray));
  int i, s = 0;

  for (i = 0; i < arraySize; i++)
    s += array[i];

  Sarray->_array = array;
  Sarray->_arraySize = arraySize;

  return(cs_EQ(cs_Sigma(tint, size), s) &&
       cs_eventKnown(tint, size, known, (void*) Sarray));
}

void printSolution(CSint **allvars, int nbVars)
{
  int i, j, n = 0;
  static int NbSolution = 0;

  printf("\nSolution %d (NbFails = %d):\n", ++NbSolution, cs_getNbFails());
  for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++)
      cs_printf("%D ", allvars[n++]);
    printf("\n");
  }
}

void fail()
{
  fprintf(stderr, "Fail!\n");
  exit(-1);
}

int main(int argc, char **argv)
{
  clock_t t0 = clock();
  CSint *matrixRC[N][N];
  CSint *matrixCR[N][N];
  CSint *allvars[N * N];
  int i, j, n = 0;
```

(continues on next page)

(continued from previous page)

```
  cs_init();

  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      allvars[n++] = matrixCR[j][i] = matrixRC[i][j] = cs_createCSint(0, 1);

  for (i = 0; i < N; i++) {
    if (!Mcons(matrixRC[i], N, &row[i][1], row[i][0])) fail();
    if (!Mcons(matrixCR[i], N, &col[i][1], col[i][0])) fail();
  }

  if (!cs_findAll(allvars, N * N, cs_findFreeVar, printSolution))
    printf("No solution\n");

  cs_printStats();
  printf("Elapsed Time = %fs\n", (double) (clock() - t0) / CLOCKS_PER_SEC);

  cs_end();

  return 0;
}
```

Output:

```
Solution 1 (NbFails = 59):
1 0 0 0 0 0 1 1
1 0 0 0 0 1 0 0
1 1 1 1 1 0 0 0
0 1 1 1 1 1 1 1
0 0 0 0 1 1 0 0
0 1 1 1 1 0 0 1
0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0

Solution 2 (NbFails = 59):
1 0 0 0 0 0 1 1
1 0 0 0 0 1 0 0
1 1 1 1 1 0 0 0
0 1 1 1 1 1 1 1
0 0 0 0 1 1 0 0
0 1 1 1 1 0 1 0
0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0

Solution 3 (NbFails = 71):
1 0 0 0 0 1 1 0
1 0 0 0 0 0 0 1
1 1 1 1 1 0 0 0
0 1 1 1 1 1 1 1
0 0 0 0 1 1 0 0
0 1 1 1 1 0 0 1
0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0

Solution 4 (NbFails = 71):
1 0 0 0 0 1 1 0
1 0 0 0 0 0 0 1
```

(continues on next page)

(continued from previous page)

```
1 1 1 1 1 0 0 0
0 1 1 1 1 1 1 1
0 0 0 0 1 1 0 0
0 1 1 1 1 0 1 0
0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0

Nb Fails = 89
Nb Choice Points = 18

Nb Fails = 89
Nb Choice Points = 184
Heap Size = 1024
Elapsed Time = 0.000951s
```

6.3 Sgn Constraint

For example, let us implement a new Sgn() constraint that gives the sign of a CSint variable. The sign of a CSint variable will be defined as a CSint variable whose domain is {-1, 0, 1}:

- Sgn(vint) = -1 $\Leftrightarrow$ vint < 0
- Sgn(vint) =0 $\Leftrightarrow$ vint = 0
- Sgn(vint) =1 $\Leftrightarrow$ vint > 0

Following is the exhaustive definition of the Constraint Propagation associated to this constraint:

1. An event on *vint* will be propagated on *s* (defined as *Sgn(vint)*) the following way:
   a. *known* event on *vint*: (performed by knownVint() function)
      - Check the value of *vint*, and instantiate *s* accordingly.
   b. *newMin* event on *vint*: (performed by newMinVint() function)
      - if cs_getMin(*vint*) = 0 then $s \geq 0$;
      - if cs_getMin(*vint*) > 0 then *s* > 0 (i.e. *s* = 1).
   c. *newMax* event on *vint*: (performed by newMaxVint() function)
      - if cs_getMax(*vint*) = 0 then $s \leq 0$;
      - if cs_getMax(*vint*) < 0 then *s* < 0 (i.e. *s* = -1).
   d. *neq* event on *vint*: (performed by neqVint() function)
      - if *vint* $\neq 0$ then s $\neq 0$.
2. Similarly, an event on *s* will be propagated on *vint*
   a. *known* event on *s*: (performed by knownS() function)
      - if *s* = -1 then *vint* < 0;
      - if *s* = 0 then *vint* = 0;
      - if *s* = 1 then *vint* > 0.
   b. *newMin* event on *s*: (performed by newMinS() function)

      The only possible *newMin* event on *s* is $s \geq 0$, so:

- *vint* ≥ 0.

c. *newMax* event on *s*: (performed by newMaxS() function)

The only possible *newMax* event on *s* is $s \leq 0$, so:

- vint ≤ 0.

d. *neq* event on *s*: (performed by neqS() function)

The only possible *neq* event on *s* is $s \neq 0$, so:

- vint $\neq 0$.

Here is the corresponding code of the definition of Sgn(), followed by an example of use (which consists in finding a solution for the equation Sgn(A) + Sgn(B) = A * B, with A $\neq 0$):

Definition of Sgn():

```
#include "iz.h"

static IZBOOL knownVint(int val, int index, CSint **tint, int size, void *s)
{
  if (val < 0)
    return(cs_EQ((CSint*) s, -1));
  if (val > 0)
    return(cs_EQ((CSint*) s, 1));
  return(cs_EQ((CSint*) s, 0));
}

static IZBOOL newMinVint(CSint *vint, int index, int oldMin, CSint **array, int size,␣
↪void *s)
{
  if (cs_getMin(vint) == 0)
    return(cs_GE((CSint*) s, 0));
  if (cs_getMin(vint) > 0)
    return(cs_GT((CSint*) s, 0));
  return TRUE;
}

static IZBOOL newMaxVint(CSint *vint, int index, int oldMin, CSint **array, int size,␣
↪void *s)
{
  if (cs_getMax(vint) == 0)
    return(cs_LE((CSint*) s, 0));
  if (cs_getMax(vint) < 0)
    return(cs_LT((CSint*) s, 0));
  return TRUE;
}

static IZBOOL neqVint(CSint *vint, int index, int neqValue, CSint **array, int size,␣
↪void *s)
{
  if (neqValue == 0)
    return(cs_NEQ((CSint*) s, 0));
  return TRUE;
}

static IZBOOL knownS(int valS, int index, CSint **tint, int size, void *vint)
{
  if (valS < 0)
```

(continues on next page)

(continued from previous page)

```
    return(cs_LT((CSint*) vint, 0));
  if (valS > 0)
    return(cs_GT((CSint*) vint, 0));
  return(cs_EQ((CSint*) vint, 0));
}

static IZBOOL newMinS(CSint *s, int index, int oldMin, CSint **array, int size, void␣
↪*vint)
{
  return(cs_GE((CSint*) vint, 0));
}

static IZBOOL newMaxS(CSint *s, int index, int oldMin, CSint **array, int size, void␣
↪*vint)
{
  return(cs_LE((CSint*) vint, 0));
}

static IZBOOL neqS(CSint *s, int index, int neqValue, CSint **array, int size, void␣
↪*vint)
{
  return(cs_NEQ((CSint*) vint, 0));
}

CSint *Sgn(CSint *vint)
{
  int n = 0;
  int array[3];
  CSint *s;

  if (cs_getMin(vint) < 0)
    array[n++] = -1;
  if (cs_getMax(vint) > 0)
    array[n++] = 1;
  if (cs_isIn(vint, 1))
    array[n++] = 0;
  s = cs_createCSintFromDomain(array, n);

  // first parameter for each cs_event* is
  // a array of CSint* having one element, "vint".
  cs_eventKnown(&vint, 1, knownVint, (void*) s);
  cs_eventNewMin(&vint, 1, newMinVint, (void*) s);
  cs_eventNewMax(&vint, 1, newMaxVint, (void*) s);
  cs_eventNeq(&vint, 1, neqVint, (void*) s);

  cs_eventKnown(&s, 1, knownS, (void*) vint);
  cs_eventNewMin(&s, 1, newMinS, (void*) vint);
  cs_eventNewMax(&s, 1, newMaxS, (void*) vint);
  cs_eventNeq(&s, 1, neqS, (void*) vint);

  return(s);
}
```

Example of use:

```
int main(int argc, char **argv)
{
```

(continues on next page)

(continued from previous page)

```
  CSint *A, *B;

  cs_init();

  /* Sgn(A) + Sgn(B) = A * B, A and B are {-10..10}, and A != 0 */

  A = cs_createNamedCSint(-10, 10, "A");
  B = cs_createNamedCSint(-10, 10, "B");

  cs_Eq(cs_Add(Sgn(A), Sgn(B)), cs_Mul(A, B));
  cs_NEQ(A, 0);

  cs_Vsearch(2, A, B, cs_findFreeVar);
  cs_printf("%T\n%T\n", A, B);

  cs_printStats();

  cs_end();

  return 0;
}
```

Output:

```
A: 1
B: 2

Nb Fails = 11
Nb Choice Points = 13
Heap Size = 1024
```