iZ-C チュートリアル

第 3.2 版 (2011 年 8 月 1 日 改訂)

(株)NTT データ セキスイシステムズ

はじめに

iZ は効率的で拡張性のある制約解消ライブラリです。iZ は資源割り当て、スケジューリング、プランニング、生産制御のような複雑な組み合わせ論的問題を宣言的に表現し、解決することを目的としています。本版では、iZ のうち C 言語用のライブラリ **iZ-C** について記述します。
本編は、iZ-C を用いたコーディング例による制約処理入門のチュートリアルです。
本チュートリアルは、読者が C 言語によるプログラミングについての基本的な知識を持っていることを想定して書かれています。チュートリアル中で扱われる iZ-C の API に関しては、リファレンスマニュアルを随時参照してください。

第 3.2 版での変更点

・BOOL 型は特定の環境で定義済みの型と衝突するのを防ぐため、IZBOOL 型という名前に変更されました。(IZBOOL 型は char を typedef したものであり、第 3 版との型の変更はありません)

1. iZ-C を用いたモデル化

制約解消による問題解決のアプローチは以下の２つの原理に基づいています。

1. 問題を一連の変数と変数間の制約によりモデル化します。問題の解決とは、すべての変数について制約をまもった値を見つけることになります。
2. 問題解決は、「制約伝播」と呼ばれる機構に基づいて行われます。制約伝播により、解の探索の間、制約は矛盾を検出し、解に含まれない値を変数から除去することができるだけ早く行えるように能動的に使用されます。

問題をモデル化する段階では、問題が「どのようにして」解かれるかを考える必要はありません。 この段階では、変数（問題において値を求めるべき対象）と変数間の制約（変数のとりうる値が満たさなければならない制限）により問題を宣言的に表現することを目指します。iZ-C では変数は領域変数と呼ばれ、CSint という型で表現されます。CSint 型の変数のとりうる値の集合を領域といいます。

以下に例を示します。

```
{
  CSint *vint = cs_createNamedCSint(0, 10, "vint");区間(0..10)を領域とする CSint 型の変数
  cs_printf("%T\n", vint);                          を生成し、その領域を表示する。
}
```

- vint: {0..10}

```
{
  int domain[9] = {-2, 0, 1, 2, 3, 5, 6, 7, 10};     domain 配列により定義される領域を持つ
  CSint *vint = cs_createCSintFromDomain(domain, 9);CSint 型の変数を生成し、その領域を
  cs_printf("%T\n", vint);                           表示する。
}
```

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

例:

例として、「覆面算」を取り上げてみましょう。A～T までの文字を 0～9 の数字に置き換えたときに以下の乗算の結果が正しくなるような文字と数字の対応を見つけることが問題です。また、0～9 の数字は各々 2 回ずつ出現するものとします。

```
    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. この問題の解は CSint 型の変数 variables A, B, ..., T の値で表現されます。
2. 変数の間の制約は、以下の通りです。

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 ≠ 0, D ≠ 0, G ≠ 0, J ≠ 0, M ≠ 0, P ≠ 0;`
g). 0 ～ 9 の数字は、{A, B, ..., T}中にそれぞれちょうど 2 回ずつ出現します。

なお、L1, L2, ..., L6 は以下のように定義されます。

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.`

では、この問題を iZ-C を用いて記述してみましょう。

```
#include "iz.h"

#define NB_DIGITS 20
CSint **Digit;                                          /*Digit is an array of 20 CSint variables*/
CSint *L1, *L2, *L3, *L4, *L5, *L6;
enum {a = 0, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t};
                                                  /*Digit[a] = A, Digit[b] = B, ..., Digit[t] = T*/
void constraints()
{
  int val;

  Digit = cs_createCSintArray(NB_DIGITS, 0, 9);

  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[h], L1));                                       (c)
  cs_Eq(L5, cs_Mul(Digit[d], L1));                                       (d)
  cs_Eq(L6, cs_VScalProd(3, L5, L4, L3, 100, 10, 1));                    (e)

  cs_NEQ(Digit[a], 0);                                                   (f)
  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);

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

2. 木探索と制約伝播

領域変数と制約を使って問題を表現したら、今度は変数の値を見つけねばなりません。（場合によっては、可能な値の集合を見つけるだけでもいいかもしれません。）値を見つけるのは制約伝播を伴った木探索によって行われます。
これは以下のようにして実行されます。
レベル n:

1. まだ即値化されていない（＝値が 1 つに決まっていない）領域変数 `var` を選択します。まだ即値化されていない変数がなければ問題は解けたことになります。
2. *チョイスポイント*を設定して、このレベルでのすべての領域変数の状態を記憶します。
3. 領域に含まれる値 `val` を選択し、変数 `var` をこの値に即値化してみます。
4. - 3 が失敗した場合(つまり、制約伝播によって矛盾が検出された場合)、直近のチョイスポイントに戻り、すべての領域変数の状態をこのチョイスポイントが設定されたときの状態に戻します(この「戻って復元する」プロセスをバックトラックと呼びます)。
   - 3 が失敗し、かつ `var` についてすべての可能な値をすでに試した場合には、レベル n – 1 のチョイスポイントにバックトラックします。(もし n = 1 ならこの問題には解がないことになります)。
   - 上記のいずれでもない場合には、次の n + 1 レベルを実行します。

iZ-C ライブラリでは、この基本的な木探索のアルゴリズムは iZ-C の機能を使って C 言語で 15 行で書かれています。(リファレンスマニュアルの 7.コンテキスト を参照)。

この木探索では、制約伝播が上記のステップ 3 の選択された領域変数を選択された値に即値化してみるところで起きます。この制約伝播のメカニズムは、領域変数 `var` の領域に変化が起きたことをきっかけにして起動されます。このとき領域変数 V_i が制約 $C_{i,j}$ によって 変数 `var` に結びつけられていれば、変数 `var` の新しい領域と制約 $C_{i,j}$ に応じて変数 V_i の値が変化します。
同様にして今度は変数 V_i が制約伝播のきっかけになるので、このメカニズムは再帰的に実行されます。

例として、2 つの領域変数 `A` と `B` が以下のようにいずれも初期値として領域 `1..8` を持つとしたとき、以下の制約が設定されるものとしましょう。

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

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

この制約が設定されると、以下のように領域変数の領域が変化します。

```
A: {3..8};
B: {1..6}
```

さらに A の領域を変更してみます。

```
A ≠ 5
```

すると、制約伝播により B ≠3 となります。

また、B の領域を以下のように変化させると、

```
B ≤ 3
```

制約伝播により A ≤ 5 となります。

iZ-C を使ったプログラム例と出力例を以下に示します。

```
#include "iz.h"

int main(int argc, char **argv)
{
  CSint *A = cs_createNamedCSint(1, 8, "A");
  CSint *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);

  return 0;
}
```

◆

```
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}
```

iZ-C では制約伝播は、制約を設定したとき（上の例では cs_Add 制約を設定したとき）、あるいは CSint 型の変数に何か変化が起きたときに自動的に実行されます。

iZ-C には既述の木探索のメカニズムが、組み込みの関数として含まれています。cs_search()はそうした関数の１つですが、引数として findFreeVar() 関数をとり、これによって CSint 型変数を選択する順序（既述の木探索のステップ 1 を参照）を制御することができるようになっています。

cs_search()関数の使い方を示すために、「覆面算」の問題に戻ります。

```
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)
{
  constraints();
  cs_search(Digit, NB_DIGITS, cs_findFreeVarNbElements);
  printSolution(Digit, NB_DIGITS);
  return 0;
}
```

結果は以下のようになります。

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

     Nb Fails = 134
     Nb Choice Points = 217
     Heap Size = 4096
```

出力された解は、134 回のバックトラック（フェイルの回数）と 217 個のチョイスポイントののちに見つかりました。
ここで cs_search()の引数として渡されている cs_findFreeVarNbElements() 関数は iZ-C の組み込み関数で、 即値化されていない CSint 型変数のうちもっとも領域の小さいもの、言い換えれば要素数の最も少ないものを返します。この戦略は、多くの場合に効率的ですが、自分で findFreeVar()を書き換えれば、別の戦略を実装することもできます。

また、選られた解が唯一のものであるかどうかが知りたくなるかもしれません。この場合には cs_search()のかわりに cs_findAll()関数を使えば答を得ることができます。

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

```
{
  constraints();
  cs_findAll(Digit, NB_DIGITS, cs_findFreeVarNbElements, printSolution);
  cs_printStats();
  return 0;
}
```

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

     Nb Fails = 134
     Nb Choice Points = 217
     Heap Size = 4096

     Nb Fails = 1669
     Nb Choice Points = 2600
     Heap Size = 8192
```

この「覆面算」が唯一の解を持つことがわかるまでに 1669 回のバックトラックが必要でした。

なお、cs_findAll() の 4 番目の引数は解が 1 つ見つかるたびに呼ばれます。

この小さな例でもわかる通り、制約による問題解決は以下の 3 ステップよりなります。

1. 領域変数を宣言します。
2. 領域変数間の制約を表現します。
3. 制約を充たす変数の値を見つけます。

iZ-C は

- 第 1 の点については
  CSint 型の変数を提供します。
- 第 2 の点については
  多くの組み込みの制約を提供するとともに、新しい利用者定義の制約を定義する方法も提供します(3.デモンを用いた制約の定義 を参照してください)。
- 第 3 の点については
  組み込みの木探索の関数を提供するとともに、新しい利用者提供のヒューリスティクスを定義する方法も提供します(リファレンスマニュアルの 5. ヒューリスティクスと生成のメカニズム および 7.コンテキスト を参照してください)。

3. デモンを用いた制約の定義

iZ-C では新しい制約を定義するためにデモンと呼ばれるインタフェースを提供しています。
制約伝播の際には CSint 型変数の領域の変化を表わす幾つかのイベントが発生します。イベントは排他的な以下の 4 種類に分類されます。

- Known: 領域は 1 つの値に縮小された (CSint 型変数が即値化された場合です。)
  Ex.: `cs_EQ(vint, 0); /*Known イベントが発生する例*/`
- NewMin: 領域の下限が変更された(必ず以前より大きな値に変化します。)
  Ex.: `cs_GT(vint, 0);`
- NewMax: 領域の上限が変更された(必ず以前より小さな値に変化します。)
  Ex: `cs_LT(vint, 0);`
- Neq: 領域から 1 つの値が取り除かれた
  Ex: `cs_NEQ(vint, 0);`

これらのイベントが排他的であるとは、以下のことを意味します。
- *known* イベント発生する場合には、*newMin*, *newMax*, *neq* イベントは発生しません。
- *newMin* イベントおよび *newMax* イベントが発生する場合には、*neq* イベントは発生しません。

例えば、

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

の場合には *known* イベントが発生し(vint は 10 に即値化される)、*newMin* イベントは発生しません。
また、

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

の場合には、*newMin* イベントが発生し(vint は 1 より大きくなる)、*neq* イベントは発生しません。

例として、有名な N-クイーン の問題を cs_eventKnown() を使って記述してみましょう。

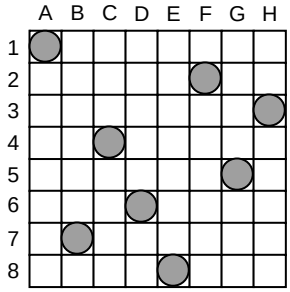
(8-クイーンは 8 つのクイーンをチェス盤上に、互いに取られないように垂直、水平、対角線の位置を避けて配置するという問題です。)

クイーンの位置が決まったとき(すなわち CSint 変数 allvars[i] で ***known*** が発生したとき),
他のクイーンが同じ列および対角線上にないことを表現する必要があります。

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

IZBOOL knownQueen(int val, int index, CSint **allvars, int NbQueens, void *extra)
                                          This function is called when allvars[index] is instantiated
                                          It returns FALSE if Constraints Propagation fails
{
  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_createCSintArray(NbQueens, 1, NbQueens);

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

  cs_printStats();
  return 0;
}
```

実行結果は以下のようになります。

```
→   1, 7, 4, 6, 8, 2, 5, 3

    Nb Fails = 11
    Nb Choice Points = 20
    Heap Size = 4096
```



8-クイーンは 92 の解を持ちますがこれは

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

を

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

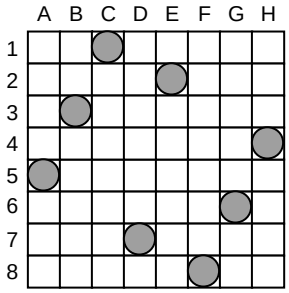
の代わりに使えば求められます。

found() は以下のように定義します。

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

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

92 の解は 289 回のバックトラックにより見つかります。55 番目に求まる解は興味深いもので、

各行は 5, 3, 1, 7, 2, 8, 6, 4 という列に配置されます。

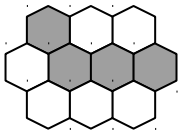


4. 例題

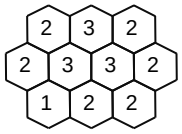
この節では制約解消の手法により解くことのできる例題をいくつか示し、それらを iZ-C を使って記述し、解いてみます。

4.1. 画像の復号化

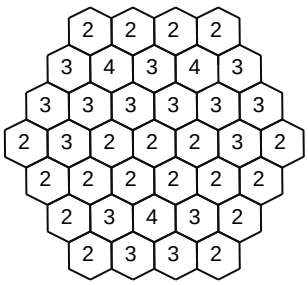
以下のように、白か黒かいずれかの可能性のあるセルよりなるグリッドを考えてみます。



このグリッドを、各セルについて隣り合ったセルのうちいくつのセルが黒いかによって符号化することにします。
例えば、上のグリッドは以下のように符号化されます。



以下のように符号化されたグリッドを元の画像に復元する問題を解くことにします。



モデル：
グリッドを構成するセルは 0 か 1 かを値として持つ CSint 変数として記述できます。 For each cell of the grid referenced by the 添字 i で参照されるグリッドの各セルに対し、以下のように制約を設定します。

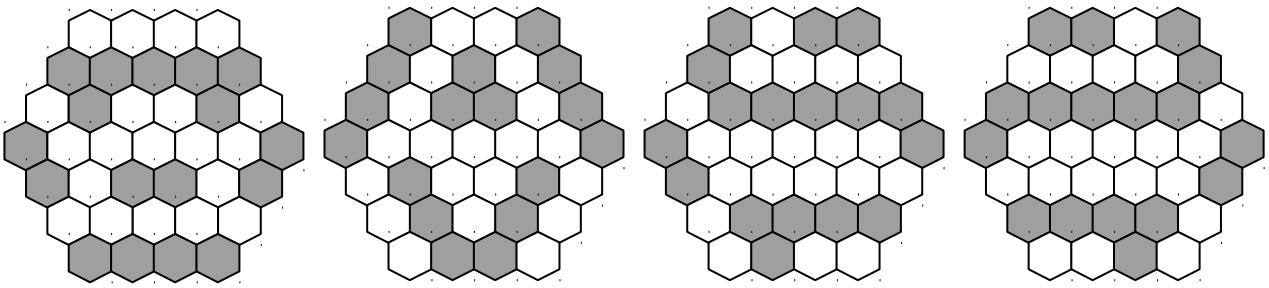
```
cs_EQ(getSum(i, cell), Code[i]);
```

i 番目のセルについて、隣り合ったセルの CSint 型変数の値の和が Code[i] に等しいことを表わしています。
getSum() はユーザ定義の関数で、組み込みの cs_Sigma() 制約を使って隣りのセルの変数の値の合計を返します。
木探索のためには組み込みの cs_findAll() 関数を使い、すべての解の探索を行います。

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

この関数は、解が見つかるたびに、第 4 引数で指定された関数を呼び出すという点を除けば、cs_search()と同じ動きをします。

iZ-C は 4 つの解があり、かつそれ以上解がないことを 9 回のバックトラックで見つけます。見つけた 4 つの解を以下に示します。



iZ-C を使って書かれたソースプログラムと実行結果を以下に示します。

```
#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");
}

void printSolution(CSint **cell, int nbCells)
{
  int i;
  int n = 9;
  static 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 izint **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_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);

  return 0;
}
```

```
→   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
```

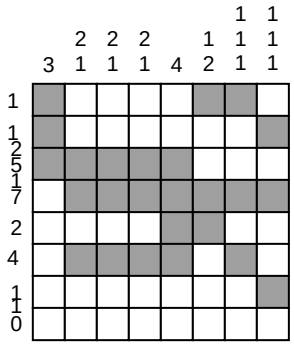
```
Nb Fails = 9
Nb Choice Points = 24
Heap Size = 4096
Elapsed Time = 0.030000s
```

興味深いことに、1 つの符号化に対して複数の解が存在し得ます。
符号化の方法を少し変えてやれば(つまり、セル自体を隣接するセルに含めなければ)、この 37 セルよりなるグリッドについての符号化が 1 つだけ解を持つようにできます。

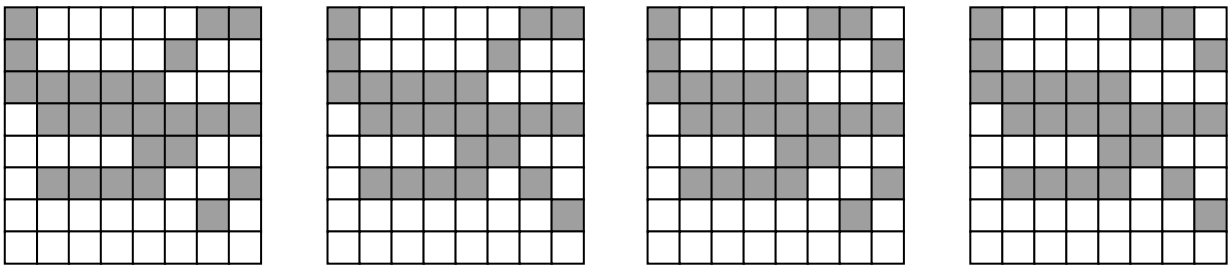
4.2. 複合魔法陣

今度の問題は、方陣の各行・各列について連続した黒いピクセルの数がわかっているときに、元の画像を復元することです。

例：



上記の例に関しては 4 つの解があり、89 回のバックトラックで見つかります。



iZ-C を使って書かれたソースコードと実行結果を以下に示します。

```
#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};
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))
      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;
        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 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;

  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);

  return 0;
```

```
|}

→   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
    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 = 184
    Heap Size = 4096
    Elapsed Time = 0.050000s
```

4.3. Sgn 制約

ユーザー定義の制約をデモンを用いて追加する例として、ある CSint 型変数の sign を返す新しい制約 Sgn() を実装してみましょう。ある CSint 型変数の sign は領域として{-1, 0, 1}を持つような CSint 型変数として定義できます。

```
Sgn(vint) = -1⇔ vint < 0;
Sgn(vint) =0  ⇔ vint = 0;
Sgn(vint) =1  ⇔ vint > 0.
```

この制約による制約伝播の完全な定義は以下に示すようなものです。

1. vint で発生したイベントは以下のようにして変数 s (Sgn(vint)として定義された変数)に伝播します。

a. vint で *known* イベントが発生したとき(後述の knownVint()関数の動作)
   vint の値をチェックして, それに応じて s を即値化。

b. vint で *newMin* イベントが発生したとき(後述の newMinVint() 関数の動作)
   if cs_getMin(vint) = 0 then s ≥ 0;
   if cs_getMin(vint) > 0 then s > 0 (i.e. s = 1).

c. vint で *newMax* イベントが発生したとき(後述の newMaxVint() 関数の動作)
   if cs_getMax(vint) = 0 then s ≤ 0;
   if cs_getMax(vint) < 0 then s < 0 (i.e. s = -1).

d. vint で *neq* イベントが発生したとき(後述の neqVint() 関数の動作)
   if vint ≠ 0 then s ≠ 0.

2. 同様に、s で発生したイベントは以下のようにして vint に伝播します。

   a. s で*known* イベントが発生したとき(後述の knownS() 関数の動作)
      if s = -1 then vint < 0;
      if s = 0 then vint = 0;
      if s = 1 then vint > 0.

   b. s で *newMin* イベントが発生したとき(後述の newMinS() 関数の動作)
      s で発生する *newMin* イベントは s ≥ 0 しかありえないから、
      vint ≥ 0.

   c. s で *newMax* イベントが発生したとき(後述の newMaxS() 関数の動作)
      s で発生する *newMax* イベントは s ≤ 0 しかありえないから、
      vint ≤ 0.

   d. s で *neq* イベントが発生したとき(後述の neqS() 関数の動作)
      s で発生する *neq* イベントは s ≠ 0 しかありえないから、
      vint ≠ 0.

上述の Sgn() の定義を実装したコードと、使用例のコードを以下に示します。 例は等式 Sgn(A) + Sgn(B) = A * B, with A ≠ 0 の解を見つけるというものです。

```
#include "iz.h"

                                                                    Events on vint
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;
}

                                                                    Events on s
static IZBOOL knownS(int valS, int index, CSint **tint, int size, void *vint)
{
  if (valS < 0)
    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);

  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, newMinVint, (void*) vint);
  cs_eventNewMax(&s, 1, newMaxVint, (void*) vint);
  cs_eventNeq(&s, 1, neqS, (void*) vint);

  return(s);
}

int main(int argc, char **argv)
{
Here we solve the equation Sgn(A) + Sgn(B) = A * B, with A and B in -10..10, and A ≠ 0

  CSint *A = cs_createNamedCSint(-10, 10, "A");
  CSint *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();
  return 0;
}
```

The array is reduced to one element: `vint`

```
→   A: 1
    B: 2

    Nb Fails = 11
    Nb Choice Points = 13
    Heap Size = 4096
```