マッチ棒問題

2013.06.06　初版

【06.20 修正版】

mod166 版：mod160 の SQC を改善し、約 4.6 倍の性能向上を実現。
7 x 7 の探索が 20.7H で完了。

１．実測値による性能比較

1.1　測定環境について

今回の測定結果は、すべて下記のデスクトップ PC を使用したものである。

【PC】	【開発環境】
CPU:Intel Core i7 2600 3.4 GHz	コンパイラ：Visual C++ 2008 Express
キャッシュ容量：8 MB	
メモリ容量：4 GB	
グラフィックスドライバ：270.81	
OS :Windows7 Home premium （32ビット版）	

１．２　問題２の（２）：ｒ本のマッチ棒を孤立点がないように取り除いたときの配置の数

(本プログラムは課題とはなっていないが、関連性があるので記載)

【MT231.c】

M　N　R	配置数	探索時間(秒)
4　4　9	4975368	0.63
4　4　10	7082522	1.25
4　4　11	8502064	2.04
5　5　7	78494904	3.10
5　5　8	1382703980	16.61
5　5　9	1382703980	72.72

【les07-ex02-HPI】状態遷移マトリクスを用いたオリジナルプログラム

---- (10,10) ----　　137202 msec

0	1
1	212
2	22 278
3	1 547 004
4	79 847 833
5	3 266 980 696
6	110 355 685 974
7	3 164 929 415 252
8	78 655 530 649 408
9	1 720 472 712 642 520
10	33 529 780 127 200 800
11	587 972 655 594 333 440
12	9 352 846 816 868 387 669
13	135 871 476 036 610 693 224
14	1 812 998 854 989 003 399 836
15	22 329 610 241 912 674 472 796
16	254 929 712 696 202 762 837 897

les07-ex02 は、探索ではなく、状態遷移マトリクスを使った行列演算処理によって配置数を求めている。

1.3　問題３(課題プログラム)

【MT311】

M　N　R　S	配置数	探索時間(秒)
4　4　8　0	0	1.08
4　4　9　0	16	1.95
5　5　8　0	0	183.04
5　5　9　0	0	690.24

【各プログラムの実測結果サマリ】

表１　主なプログラムの測定結果一覧

	3	4	5	6	7	備考
MT311		(1.95)				(0,9)のみ
base model	0.19	0.52	890.42			
LBE0	0.17	0.30	184.94			LBE0
mod10	0.16	0.17	99.19			LBE1 = R + S
mod11	0.16	0.12	12.90	2931.23		LBE2 = 2R + S
mod14	0.16	0.78	9.06	2026.27		LBE3 = LBE1+LBE2
mod50	0.11	0.31	3.59	678.82		mod14+SymCheck
mod100	0.01	0.05	7.41	172.71		HJ base model
mod102	0.00	0.03	4.65	50.98		mod100 + SQC
mod104	0.00	0.03	4.54	31.26		mod102 + SQC refine
mod110	0.02	0.02	3.95	27.00		mod104 + Mapsソート
mod150	0.02	0.02	3.15	64.48		mod100 + SymCheck
mod154	0.02	0.02	1.90	17.24		mod104+SymCheck
mod160	0.02	0.02	1.76	15.43		mod110+SymCheck
mod166	**0.00**	**0.02**	**0.80**	**14.87**	**74495.19**	**mod160+SQC改善**

・最速の LBE 版(mod50)の改善率：　248 倍(base model との比較)
・最速の HJ 版(mod160)の改善率：　44 倍(最速 LBE 版との比較)
・最新版の mod166　　：　mod160 の約４．６倍高速

【base model】実測結果詳細(les07-ex03)

```
      M  N    S  R
---[ 4, 4] / ( 0, 9)  :      16          0 msec   [  MT311 : 1.95 秒 ]
---[ 4, 4] / ( 1, 9)  :    4972         62 msec
---[ 4, 4] / ( 2, 8)  :    1798         62 msec
---[ 4, 4] / ( 3, 7)  :     168         47 msec
---[ 4, 4] / ( 4, 7)  :    5720         78 msec
---[ 4, 4] / ( 5, 6)  :     312         31 msec
---[ 4, 4] / ( 6, 6)  :    4230         63 msec
---[ 4, 4] / ( 7, 5)  :      80          0 msec
---[ 4, 4] / ( 8, 5)  :    1164         16 msec
---[ 4, 4] / ( 9, 5)  :    7144         31 msec
---[ 4, 4] / (10, 4)  :      84          0 msec
---[ 4, 4] / (11, 4)  :     784          0 msec
---[ 4, 4] / (12, 4)  :    3020         16 msec
---[ 4, 4] / (13, 3)  :       8          0 msec
---[ 4, 4] / (14, 3)  :     156          0 msec
===[ 4, 4]              total      515 msec

---[ 5, 5] / ( 0,14)  :     448       7722 msec
---[ 5, 5] / ( 1,13)  :     208      21138 msec
```

```
---[ 5, 5] / ( 2,12) :       16        27425 msec
---[ 5, 5] / ( 3,12) :    28030        98608 msec
---[ 5, 5] / ( 4,11) :     2092        62244 msec
---[ 5, 5] / ( 5,11) :   151276       153192 msec
---[ 5, 5] / ( 6,10) :     6716        63024 msec
---[ 5, 5] / ( 7,10) :   196528       124737 msec
---[ 5, 5] / ( 8, 9) :     4896        37472 msec
---[ 5, 5] / ( 9, 9) :    99740        64163 msec
---[ 5, 5] / (10, 8) :     1038        14976 msec
---[ 5, 5] / (11, 8) :    21422        23072 msec
---[ 5, 5] / (12, 7) :       40         4336 msec
---[ 5, 5] / (13, 7) :     1624         6147 msec
---[ 5, 5] / (14, 7) :    19112         8346 msec
===[ 5, 5]                  total    890418 msec
```

後述のように base model は、マッチ棒を列毎のビットパターンで管理するデータ構造を採用したもので、性能改善の機能は未実装の版である。それでも、MT311 に較べて約 1000 倍以上の差がある。

【最速データ詳細】les07-ex03-mod160

```
---[ 6, 6] / ( 0,19) :       16 (     1761)         15 msec
---[ 6, 6] / ( 1,19) :    13632 (   361198)        265 msec
---[ 6, 6] / ( 2,18) :     2752 (   323816)        203 msec
---[ 6, 6] / ( 3,17) :       72 (   116834)         78 msec
---[ 6, 6] / ( 4,17) :   371612 ( 2678292)        780 msec
---[ 6, 6] / ( 5,16) :    20300 (   569214)        171 msec
---[ 6, 6] / ( 6,16) :  4239420 ( 6960858)       1653 msec
---[ 6, 6] / ( 7,15) :   161024 ( 1157496)        281 msec
---[ 6, 6] / ( 8,15) : 11181548 (10533220)       2667 msec
---[ 6, 6] / ( 9,14) :   278384 ( 1546066)        343 msec
---[ 6, 6] / (10,14) : 11129340 (12316203)       3136 msec
---[ 6, 6] / (11,13) :   165500 ( 1711480)        358 msec
---[ 6, 6] / (12,13) :  5117504 (12948555)       2730 msec
---[ 6, 6] / (13,12) :    37360 ( 1764725)        312 msec
---[ 6, 6] / (14,12) :  1138828 (13127423)       2090 msec
===[ 6, 6]                                total    15428 msec

---[ 7, 7] / ( 0,26) :      544 (   163767)       2293 msec
---[ 7, 7] / ( 1,25) :      248 (   167448)       1170 msec
---[ 7, 7] / ( 2,24) :       16 (    54288)        390 msec
---[ 7, 7] / ( 3,24) :    51436 ( 4699243)     128123 msec
---[ 7, 7] / ( 4,23) :     2232 (   703693)       6443 msec
---[ 7, 7] / ( 5,23) :  6313756 (21594872)    2772125 msec
---[ 7, 7] / ( 6,22) :   288132 ( 2148288)      55910 msec
---[ 7, 7] / ( 7,22) : 121445596 (43053038)   19773253 msec
---[ 7, 7] / ( 8,21) :  4077696 ( 3334695)     271300 msec
---[ 7, 7] / ( 9,21) : 653897876 (55786580)   64762875 msec
---[ 7, 7] / (10,20) : 15912022 ( 3783066)     705792 msec
```

時間がかかるので、中断

【最新版 mod166 データ詳細】les07-ex03-mod166　2013.06.20　追加

```
---[ 6, 6] / ( 0,19) :       16 (     1761)         15 msec
---[ 6, 6] / ( 1,19) :    13632 (   361198)        296 msec
---[ 6, 6] / ( 2,18) :     2752 (   323816)        203 msec
---[ 6, 6] / ( 3,17) :       72 (   116834)         78 msec
---[ 6, 6] / ( 4,17) :   371612 ( 2678292)        795 msec
---[ 6, 6] / ( 5,16) :    20300 (   569214)        188 msec
---[ 6, 6] / ( 6,16) :  4239420 ( 6960858)       1591 msec
---[ 6, 6] / ( 7,15) :   161024 ( 1157496)        280 msec
```

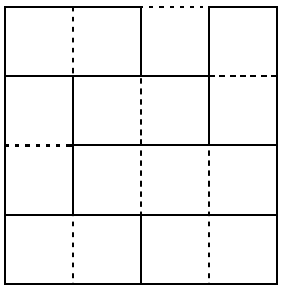
```
---[ 6, 6] / ( 8,15) :  11181548   (10533220)        2511 msec
---[ 6, 6] / ( 9,14) :    278384   ( 1546066)         343 msec
---[ 6, 6] / (10,14) :  11129340   (12316203)        2933 msec
---[ 6, 6] / (11,13) :    165500   ( 1711480)         343 msec
---[ 6, 6] / (12,13) :   5117504   (12948555)        2605 msec
---[ 6, 6] / (13,12) :     37360   ( 1764725)         312 msec
---[ 6, 6] / (14,12) :   1138828   (13127423)        2028 msec
===[ 6, 6]                              total    14867 msec

---[ 7, 7] / ( 0,26) :       544   (  163767)         702 msec
---[ 7, 7] / ( 1,25) :       248   (  167448)         686 msec
---[ 7, 7] / ( 2,24) :        16   (   54288)         358 msec
---[ 7, 7] / ( 3,24) :     51436   ( 4699243)       38953 msec
---[ 7, 7] / ( 4,23) :      2232   (  703693)        4618 msec
---[ 7, 7] / ( 5,23) :   6313756   (21594872)      645831 msec
---[ 7, 7] / ( 6,22) :    288132   ( 2148288)       24429 msec
---[ 7, 7] / ( 7,22) : 121445596   (43053038)     4305810 msec
---[ 7, 7] / ( 8,21) :   4077696   ( 3334695)       85956 msec
---[ 7, 7] / ( 9,21) : 653897876   (55786580)    13918157 msec    (mod160 の約４．６倍高速)
---[ 7, 7] / (10,20) :  15912022   ( 3783066)      200102 msec
---[ 7, 7] / (11,20) : 1344868044   (59804918)    25494656 msec
---[ 7, 7] / (12,19) :  23186912   ( 3866865)      297524 msec
---[ 7, 7] / (13,19) : 1301838656   (60508959)    29177339 msec
---[ 7, 7] / (14,18) :  14969880   ( 3874328)      288959 msec
===[ 7, 7]                              total 74495191 msec     （２０．７Ｈ）
```

２．解法プログラムの概要

2.0　データ表現

１）マッチ棒の配置の表現(Ｍ行Ｎ列、Ｍ≦Ｎ)

0 1　2　3　4 5　6　7　8　9 10



１）マッチ棒の上から下への並びを段と呼ぶことにする。各段には 0～2N+2 の番号をつける。
２）マッチ棒の有無を１：０で表現し、各段毎のビットマップで管理する。
３）奇数段を列、偶数段を行と呼ぶ。
４）列のビットマップは、０～2^M－１個の 2^M 個のパターンが存在する。
５）行のビットマップは、０～2^{M+1}－１個の 2^{M+1} 個のパターンが存在する。
６）配置全体は、Pos[]に各段のビットマップを入れて管理する。

なお、便宜上、0 段と 2N+2 段も Pos[0]=0、Pos[2N+2]=0 と考える。
ｉ段のビットマップパターンをその大文字Ｉで表現することがある。

２）データ構造

・Rmap[X] ：ある列のビットマップＸに対応する取り除いたマッチ棒の数を得るための配列。
（要は、Ｘの中の０の数を事前に計算して格納しておく）
・Cmap[Y] ：ある行のビットマップＹに対応する取り除いたマッチ棒の数を得るための配列。
・Smap[I, J, K] ：接続性を判定するための配列。（論理的には３次元であるが、実装上は１次元）
接続性(孤立したマッチ棒が存在してはならないという条件)は、ある列ｊの接続性は、その前後の行のビットマアップ（I, K）を合わせた３つの段のパターンで決まってしまうことに注意。
i, j, k（偶数段から始まる３つの段のビットマップパターン）から接続性の可否を判定するビット配列。具体的な判定方法は下記の接続性テストを参照。

3）Tips

接続性テスト：setsuzoku(I, J, K)

ある列のビットパターンを J 、その前後の行のビットパターンを I, K とする。

【判定方法】

```
even = (I & K) | (J & (J<<1));        //  1 の数が偶数なら 1 、そうでなければ 0
odd  = (I ^ K) ^ (J ^ (J<<1));        // 1 の数が奇数なら 1 、そうでなければ 0
err  = (odd & ~even) & ColMask;    // odd には 1 が 3 個の場合が含まれるのでそれを除外
err = 0 なら接続性テスト OK
```

注：このアルゴリズムは M(行数)には関係なく一括して判定できる点が優れている。

逆の考え方をすれば、I, J と接続性の条件から K のパターンを生成することは可能である。テストして OK のものを拾い出すより、この方が性能は良い筈。　（時間切れのため、未実装）

正方形の数のカウント：SquareCount(u, row, col)

i 段~ k 段で構成される大きさ u = (k － I) / 2 の正方形のカウント

```
row  = I & K  （ビットマップの両方が 1 のものに絞る）
col  = i+1 行から k－1 行までの and 積(ビットマップのすべてが 1 のものに絞る)
```

以下の手順で、u, row, col から正方形の数をカウントする。

```
low  = col & (－col )   col の LSB を求める   low は正方形の下辺
high = low  <<  u       正方形の上辺
col & high == 0 なら、low を底辺とする正方形は成立しない。
c = (high－low)         c は左辺・右辺のビットマップ
row & c == c なら正方形が成立しているのでカウントする
col ^= low              // low を底辺とする正方形を除外
col が 0 になるまで上記を繰り返す。
```

BitCount(X)

X に含まれる 1 の数を数える関数。

以下のトリッキーなコードが有名である。簡単なので多用している。（普通のコードでも問題ないが）

```
w = Bits;
c = 0;
while(w)  { c++;   w = w & (w-1); }
```

➔　c には Bits に含まれる’1’の数が入る。

【補足】各段毎にビットマップを管理することのメリット

① 接続性テストが簡単に一括して行える

② 正方形のカウントを一括して行え、u,row,col をキーにして配列(SQC)で管理できる。

2.1 base model

まず、正しく動作するものが必要であると考え、基本となる探索機構のみを実装した版を開発した。本プログラムの目的は、**2.0 のデータ表現の検証**と**その後の改良版の Reference model として使用**することである。(遅くても、正しい結果がえられること)

【基本アルゴリズム】

1）全体の処理構造：第 1 段～第 2N+1 段まで、各段のパターンを再帰的にテストしていく。その時、取り除いたマッチ棒の数(r)、正方形の数(s)を累積していく。もし、r, s のいづれかが上限値 R,S を超えた場合、探索を中止する。(バックトラック)

２）奇数段(列)の処理：
列データのパターンをもとに以下の処理を順次テストする。
・列データをもとに取り除かれたマッチ棒の数を累積する
・追加された列データによって完成する正方形の数を求め（SquareCount）累積する
３）偶数段(行)の処理：
行データのパターンをもとに以下の処理を順次テストする。
・行データをもとに取り除かれたマッチ棒の数を累積する。
・追加された行データによって接続性テストの確認をする。
４）２N+2 段目の処理：
最終列データを０として、接続性テストの確認をし、ｒ、ｓが上限値と一致していることを確認する。
もし、問題がなければ、解のカウントをする。

2.2 データの並べ替え

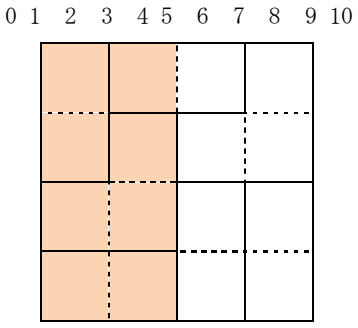
各段におけるパターンをテストする順序を「取り除くマッチ棒の数の昇順」にしてはどうか？ そうすれば、取り除いたマッチ棒の数が上限値を超えた時点で、それ以降のパターンのテストを中止することができ、速くなると思われる。
-→ 実装してみたが、効果は極めてすくなく、策に溺れた感じ。

３．下限予測値 LBE（Lower Bound Estimater）による探索量の削減

パズルの解法では、下限予測値を求める事ができる場合、その情報をもとにそれ以降の探索を続けるべきかどうかを判断して探索量を削減することは、よく使われる手法である。
ＬＢＥには２種類の作り方がある。１つは、数学的な考察により論理的に導かれる場合（今回のＬＢＥ0）と、モデルを簡略化し、簡易探索により作成されるテーブルなどを利用する場合（今回の LBE1,2,3,）がある。

3.1 LBE0

ｎ段までの行の探索が完了した時点を考える。
[説明の為の記号]
R ：取り除くマッチ棒の総数
S ：正方形の総数
r ：ｎ段までに取り除いたマッチ棒の数
s ：ｎ段までに含まれる正方形の数



↑ここまで探索済（n=4）

（N－n / 2）＊M は、(n+1)段以降に存在する１ｘ１の正方形の数

ss＝ （N－n / 2）＊M －（S－s）は、少なくとも削除しなければならない１ｘ１の正方形の数。
一方、(R－r）本のマッチ棒で削除できる１ｘ１の正方形は最大で（R－r ）＊2 個であるから、
ss ＞ (R－r) ＊2 の条件が成立した時点で探索を中止してよい。
（これ以上続けても、正方形の数は上限値 S をオーバーするので)

この極めて単純な LBE0 を base model に追加したのが LBE0 版である。 結果は表１にあるように、**約 4.8 倍の改善**となった。(実際のコードは２行追加しただけであるが)

補足：LBE0 は、マッチ棒の配置が下手で正方形の削除が足りないものを判定している。逆に、正方形を消しすぎた場合を予測する上限予測値（UBE）があれば、更に性能は向上するはずである。残念ながら、本

問題の UBE は見つかっていない。

3.2　LBE1,2,3 （簡易探索による LBE の作成）

【簡略化モデル】

取り除くマッチ棒の数は、接続性の制約をうけ、連続する 3 段のパターンで決定されることは既に述べた。一方、大きさ u の正方形の数を決定するには、2u+1 段のパターンが必要である。しかし、u=1 の場合は、3 段のパターンでよいことになる。そこで、取り除くマッチ棒の数と 1 x 1 の正方形の数に限定した簡略化モデルを考える。

【R+S】（RSband）

取り除くマッチ棒の数 R を多くすると、正方形の数 S は少なくなる。逆に、S を大きくとろうとすると、R は小さくせざるをえない。そこで、**σ ＝R＋S を評価値**として考える。σ は探索を進めるに従って、単調に増加していくことが予測される。しかし、どのパスを経由してきたかによって、σ の値は様々な値をとるであろう。そこで、ある時点(i 段のパターン I 経由で j 段のパターン J)に至る **σ の最小値を RSband** と定義する。LBE として使用するために、RSband はゴールから逆にたどるパスで σ を計算することにする。

そうすると、ある時点まで探索したとき、

R　：取り除くマッチ棒の総数
S　：残すべき正方形の総数
r　：n 段までに取り除いたマッチ棒の数
s　：n 段までに残された正方形の数
残された評価値：φ＝(R－r)＋(S－s)
ある時点からゴールまでの評価値　：σ

とすると、　φ　＜　σ であれば、探索を続ける意味はなくなる。

RSband の求め方は、6.2 を参照。

【実装】

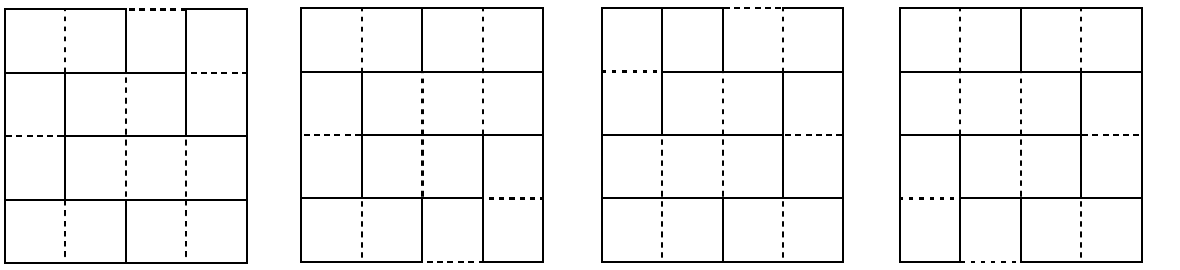
σ を単純に実装したのが、mod10 版である。LBE0 版の更に倍の性能となった。しかし、σ のバリエーションは他にも考えられる。いくつかの版を作成して実測を試みた結果では、**mod11 版(2R+S)が LBE0 版の 14.3 倍**という極めてよい結果をもたらすことが判明した。また、mod14 版は、R+S と 2R+S の合わせ技である。いまのところこれが最良である。単独の評価値として、2R+S が最良かどうかはまだ確認できていない。3R+2S、5R+2S なども興味がわくところである。

4．対称性を利用した探索量の削減

4.1 解の区分

下図のように、ある解に対称変換（上下反転、左右反転など）を施すことによって、一般には４つ１組の解が得られる。M=N の場合には、更に、９０°回転操作が加わって、８つ組の解が得られる。(稀なケースとして、これらの操作によって別解が得られない場合もある)

ある解 A　　A の上下反転解　　A の左右反転解　　A の上限反転・左右反転解



この様な解は、どれか１つ（代表解ととする）を求めれば、他の解は対称変換によって得ることができる。
代表解を判定する条件を明確にし、他の解であることが判明した時点でその探索を中止することができれば、探索量の大幅削減を期待できる。判定条件はなるべく探索の浅いところで実施できるものが望ましい。

4.2　代表解の判定条件

深さ半分までの探索を実行した時点で、以下の判定を行う。

条件１：(左右反転解の判定)

左半分のまでの探索で取り除いたマッチ棒の数を R_L

右半分で取り除くであろうマッチ棒の数を R_R　（R_R は、R_L 他より求めることができる）

$R_L < R_R$　：代表解　（得られた解の数を２倍として計上）

$R_L = R_R$　：判定不明　（従来どおり、１個として計上）

$R_L > R_R$　：代表解ではないので探索を中止する

条件２：(上下反転解の判定)

中央行(または列)のビットマップを C、C の上下のビットを入れ変えたものを $\tilde{C}$ とする。

$C < \tilde{C}$　：代表解　（得られた解の数を２倍として計上）

$C = \tilde{C}$　：判定不明　（従来どおり、１個として計上）

$C > \tilde{C}$　：代表解ではないので探索を中止する

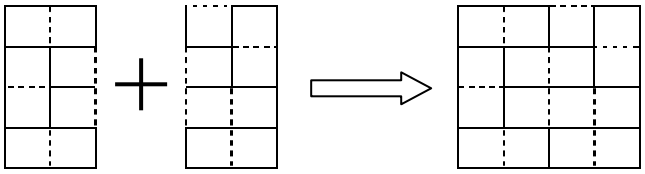
補足：今回の問題では、解の区分毎の数は求められていないと考え、上記のように簡単な判定条件に留めた。
また、M=N の場合の 90° 回転対称性の判定についても見送った。
条件設定がうまくいけば、探索空間は約１／４に削減できるので、性能的には４倍弱が期待値であるが、実装した結果では約３倍の向上となった。(mod14 → mod50)

５．部分解合成法：部分解から全体解を生成(Half Join)

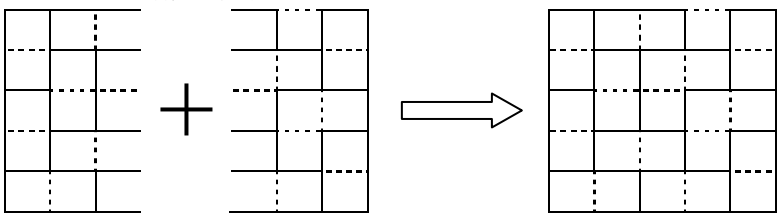
5.0　概要

下図の様に、M x N の配置を２分割した部分解(部品)から全体解を合成する方式である。

(1) N が偶数の場合



(2) N が奇数の場合



部分解を求める時間は探索の深さが半分になるので、全体的には問題にならない。しかし、部品をすべてメモリに常駐させて、2 つの部品の結合処理を行うのが基本となるため部品の数がどの位になるかは重要な要因である。 結合処理のためにどのようなデータを保持すべきか(結合条件の判定に必要な情報に絞る)、それをどのようなデータ構造で管理するかというのは重要な設計指針である。
また、プログラム構造的には部品生成処理（PGen）と結合処理（Join）とに分けられる。PGen の一部として今までの探索プログラムは利用可能である。結合処理では、キャッシュミスの多発に留意することが肝要である。

5.1　調査・検討

【部品数】

部品の数が M, N の増加とともに、どのような伸びを示すかは、部分解合成法を適用できるかどうかを判断する重要事項である。例えば、7 x 7 の問題まで解きたい場合、部品の総データ量(データ量＊部品数)が約 1 GB に納まるかどうかが分岐点と考えている。

６ｘ６までの部品数を求めることは既存のソルバーの探索ノードのヒストグラムを出力するだけなので、簡単である。７ x ７は解けていないので、推測するしかない。以下のデータは、mod14 を使用して得た探索ノードのヒストグラムである。このデータから推測すると、７ｘ７は、メモリ常駐が可能かどうか心配になる。

---[5, 5] / (7,10)　:　196528　　2169 msec

0	1
1	5
2	28
3	551
4	3295
5	43273
6	**76554**
7	932566
8	835003
9	6879214
10	2572038
11	203244

---[6, 6] / (10,14)　:　11129340　　542476 msec

0	1
1	8
2	65
3	2240
4	14608
5	358866
6	**494682**
7	**12316203**
8	**7562530**
9	175992421
10	87968525
11	1283965824
12	293064301
13	11407724

中央段の Hist データが部品数となるものである。

N が偶数のときは中央段ではなく、(N,N+2) の分割の方が Parts が少なくなりよいかもしれない。

【結合条件と JoinKey】

結合処理に必要な情報を整理するため、結合条件を明確にする。

①中央段のビットパターンが一致すること

②取り除くマッチ棒の数の合計が上限値 R となること

③結合後の正方形の総数が、上限値 S となること

　枝刈を行うため、部品内の正方形の数も用意する　→ ③’

（最終的にはすべての Pos[]データが必要）

④N が偶数のとき、中央列とその前後の列に関して、接続性が保証されること

これをもとに、最初のデータ構造を以下の様に決定した。(結果をみながら徐々に見直す方針)

JoinKey　：Hash Join を行うため、①，②、③ ’ の情報をキーとするインデクスを作成する。

①	②	③ ’

①：map 中央段のビットマップ　　(M+1 ビット)

②：r　　取り除かれたマッチ棒の数　　(6 ビット)

③ ’：s　　各部品に含まれる正方形の数（4 ビット)

保持データ：結合処理に必要な情報を Parts と Maps に分けて管理する。

Parts：結合判定条件で使用頻度の高い情報を保持する。具体的には、①、②、③ ’ と Maps へのポインタ。
キャッシュミスを少なくするため、JoinKey の昇順に並べておく。

Maps：Pos[]データのすべて。(具体的には③，④で使用するため)
部品の検索順のそのまま格納する。

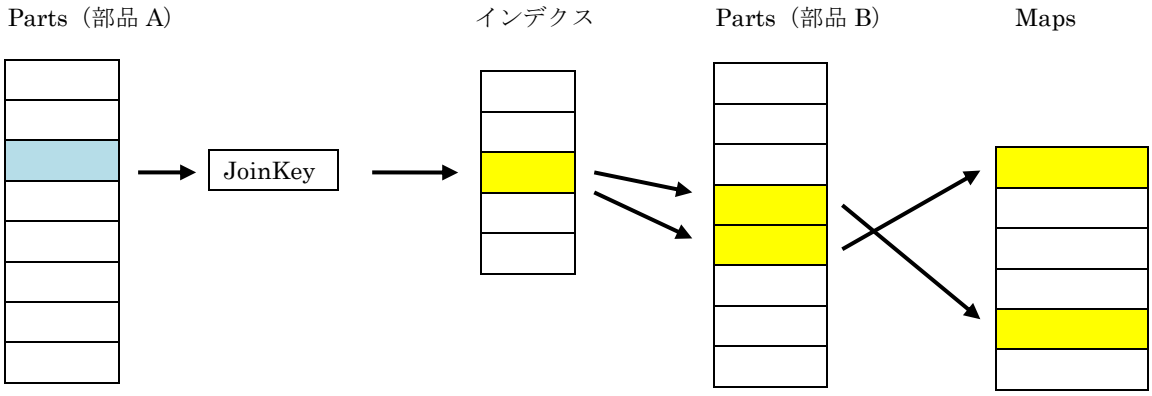
5.2　PGen

部品の探索処理は、既存の mod14 版を利用し、データの再配置処理(JoinKey の昇順にソート)を追加する。

データのソートは、JoinKey のインデクス作成と合わせて、2 パス方式で行っている。最初のパスでは、JoinKey のヒストグラムを作成(IXC)し、その IXC データをもとに、Parts,　Maps のデータをどの位置に配置すべきかというレイアウト(IXP)を生成する。 2 nd パスでは、IXP,と IXC の情報から、Parts,Maps の所定の位置へのデータ格納を行う。

5.3　Join 処理

【テーブルの利用イメージ】



2 つの部品 PartsA, PartsB のデータを以下の表記とする。

mapA,mapB：それぞれの中央段のビットマップ

rA, rB　　：各部品で取り除かれたマッチ棒の数

sA, sB　　：各部品に含まれる正方形の数

cA, cB　　：中央段で取り除かれたマッチ棒の数　（Rmap/Cmap より求める）

PartsA は、Parts 配列をシーケンシャルにスキャンして情報を取得する。結合相手の PartsB は、PartsA の情報から JoinKey を生成し、インデクス(IXP)を経由して、PartsB の範囲を特定する。この範囲の PartsB を対象に結合条件の判定を行う。

【PartsB の JoinKey の生成】

mapB = mapA　　（①の条件）：
rB　　= R － rA + cA　（②の条件より）
sB　　= 0 ~ (S－ s A)　(③の条件があるが、正確な正方形の数は簡単には計算できない)

5.4　部分解合成法における代表解の判定条件

部分解合成法と代表解の判定条件の相性は非常によく、実装は比較的容易である。

条件１：(左右反転解の判定)

部分解には、部品 id (Parts 内での並び順) が付与されている。その大小関係で判定する。

即ち、4.1 の条件１を R_L ：左側の部品 id、　R_R ：右側の部品 id と読み替えればよい。

条件２：(上下反転解の判定)

4.2 の条件２をそのまま適用。

中央行(列)のビットマップは結合条件にもなっているので取り扱いが容易。

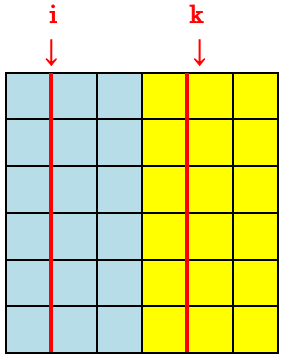
5.5　Join 用 SquareCount

Join 処理の中で、正方形の数をカウントする処理が重すぎるのは、明らかである。以下の対策をとる。

1) 関数 SquareCount(u, row, col) を SQC 配列へと変更する。　→ mod102 版

2) 基本モデルでは、PartsA に PartsB の各列を順次追加して、総数を求めているが、部品内部の正方形の数は事前に判っているので、双方の列によって成立する正方形をカウントして総数を求めるアルゴリズムに変更する。　→ mod104 版

i 段は、左半分、k 段は右半分の移動に限定して、i, k によって成立する正方形の数を集計する。



5.6　データ構造の見直し

1) Maps（Po s []）の並べ替え

N=7 の実測時間が極端に遅くなっている。おそらく、キャッシュミスが多発しているものと思われる。

そこで、データ構造と Join 処理を見直し、Maps における Pos[]データを到着順ではなく、Parts データと同じく、JoinKey 順に変更する。これに伴い、Parts データ内の pid フィールドは削除し、Parts のサイズを 8 Byte→ 4 Byte に節約する。　この対策は、Maps へのアクセス時のキャッシュミスの削減および、Parts データアクセス時のキャッシュミスの効果が期待できる。　→　mod110 版

5.7 さらなる性能改善（ mod160 → mod166 への改善内容）

mod160 版でも、７ｘ７の探索には時間がかかり、途中で断念せざるを得なかった。そこで、ソースコードを見直し、以下の改善を実施した。**これにより、約４．６倍の改善効果がえられた。**

１）命令数削減

ＳＱＣ関数からＳＱＣ配列への変更

2) キャッシュミス対策

a. アクセス頻度の高いＭａｐｓのデータをＰａｒｔｓデータに取り込み、Ｍａｐｓのアクセス頻度を低減。

b. ＳＱＣ配列のアクセス順序の変更（row, col, u ）→ (col, u, row)

６．状態遷移マトリクス

6.1 【ex02 のアルゴリズム】

前述のとおり、接続性の判定は、ある列とその前後の行のビットマップで決まる。これを利用して、状態遷移マトリクスを作成し、逐次、配置の数を求めていく。

X（i, I, r）：始点から i 行のビットマップ I に至るまでのパスの総数を格納する。
ただし、r 本のマッチ棒を取り除く配置のみとする
Tr（I, J, K）：ビットマップ I,J,K の接続性を判定する情報を格納する３次元配列。
１：接続条件 OK　　０：接続条件 NG

(計算法)

X[0, *, *] を 0 クリアし、X [0, 0, 0] = 1　とする。（初期値の設定）

I,J,K のすべてのパターン(2^{3M+2})について、以下の計算をする。

1) Tr(I,J,K) == 0 のときは接続条件が NG なので以下の処理はスキップ
2) X(i+2, *, *) を 0 クリアする
3) r=0~16 について、以下の計算をする。
rr = Rmap(J) + Cmap(K)
X(i+2, K, rr)　+= X(i, I,r)
4) 上記の処理を I = 0~2N まで繰り返す。
5) 以上により、X(2N+2,0,*) に求める配置の数が格納されている。

補足：Ｔｒマトリクスは、2^{3M+2} ビットの情報（2^{3M-1} バイト）を格納できなければならない。
M=10 では、約 537MB となる。これがメモリ制約からくる限界である。

6.2 【RSB（RSband）の計算法】

接続性の判定同様、１ｘ１の正方形の数も、連続する３つの段のビットマップで決まることは前述した。
従って、RSB の計算も、ex02 のアルゴリズムとほぼ同様に求めることができる。

RSB（j, I, J）：ゴールから j 段のビットマップ I に至るまでの評価値(R+S)の最小値を格納する。
ただし、I は j-1 段のビットマップ　　J は j 段のビットマップとする
Tr（I, J, K）：ビットマップ I,J,K の接続性を判定する情報を格納する３次元配列。
１：接続条件 OK　　０：接続条件 NG
SQC(I, J, K)：I,J,K によって構成される１ｘ１の正方形の数

(計算法)　j+1 段の RSB から j 段の RSB を求める手順を示す。
I,J,K のすべてのパターンについて以下の計算をすればよい。

$$RSB(j,I,J) = \min_K\{ RSB(j+1, J, K) + Rmap(J) \cdot Tr(I, J, K) \}$$　j=odd のとき

$$\min_K\{ RSB(j+1, J, K) + Rmap(J) + SQC(I, J, K) \}$$　j=even のとき

以上