

COMPSCI 271

Introduction to AI

Approaches for modeling and solving Sudoku puzzle

Team members:

Tianze Shou         52801260

Yumeng Zhang 71169660

Haozhe Jiang       68249954

Jinsheng Zhu 33882845

**I. Problem Overview**

Sudoku is a popular logic-based, combinatorial, number-placement puzzle. The standard version of sudoku is placed on a 9x9 matrix grid filled of either digit or blank with total 9 3x3 square subgrids that composed the grid. The goal of the puzzle is to fill out all the blanks with digits, which completes the puzzle under the constraints that repeated numbers do not exist in the same row, column and 3x3 subgrid.

The standard version of Sudoku may be granted as a mildly challenging puzzle for average human solvers, but a vanilla brute force program solver with mediocre computational hardware support can solve it within seconds. The motivation of developing more efficient sudoku puzzle solver is to solve scaled size problem such as 16x16 sudoku or larger. In this report, we dedicate on modeling and solving 16x16 sudoku problem with two different approaches.

Same as the standard sudoku(Fig.16), the 16x16 sudoku matrix grid contains 16 subgrids, each of them is a 4x4 square. Each row, column and subgrids cannot contain the same number for twice or more within the range of 1 to 16. If we model the 16x16 sudoku as a constraint satisfaction problem(CSP), the backtrack search tree of 16x16 sudoku is significantly larger than the standard version with almost doubled branch factor and tripled depth (10000 times at least).

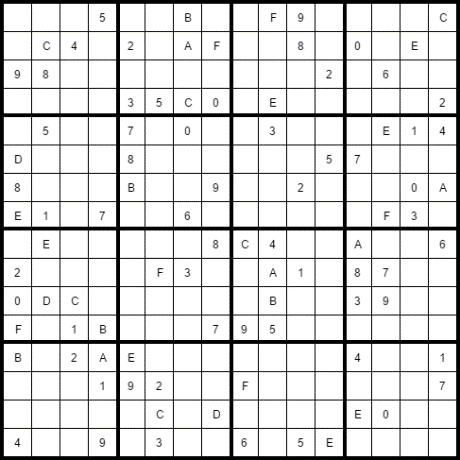
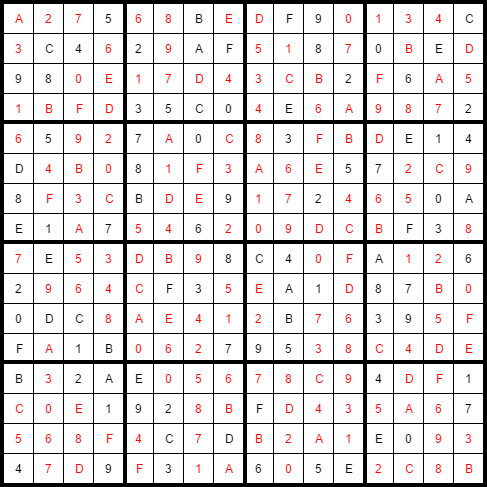
 

Fig.1 *16x16 sudoku puzzle and solution*

The first approach we applied is to model the 16x16 sudoku as a CSP problem. Each of the blank in the generated problem represents a variable with the domain range from 1 to 16. The constraints prevent the number from existing in one variable’s domain if there is assignment of that number in the same row, column or subgrid. A common strategy to solve such problem is the backtrack algorithm. An optimized backtrack algorithm can solve the hardest 16x16 sudoku problem in bounded time. Section III discreetly describe the implementation of our backtracking algorithm and corresponding optimization in value and variable ordering and inference.

Certain problem in CSPs are belong to domain of exact cover problem. Our strategy is to represent the problem with Matrix M, there the row of M denotes possible partial solution to the problem, and the column of the M serve as the constraints that need to be satisfied by the problem. In the matrix, each cell with value of 1 indicates that the corresponding row satisfy the constraint imposed by this column. However, each column must only contain a single “one” to be exactly covered by a particular partial solution, and it is also the reason why the problem represented by this matrix is called exact cover problem. The detail explanation of how to model sudoku as an exact cover problem is shown in section III.

**II. Implementation of the CSP solver**

The definition of a CSP problem consists of three major classes includes variable, domain and constraints. The definition of sudoku as a constraint satisfaction problem is shown below.

Variable X denotes each cell in the game board, where the subscript I and superscript j starting from 1 to 16 indicate the cell is in the intersection between row I and column j. For example, is the cell at the left-top corner in our coordinate system. Each cell has domain consists from number 1 to 16, except those already been initialized. A cell is equivalent to clue if it is not a blank when the board initialized. Therefore, the domain of a clue is fixed, and it is also a major indicator of the difficulty for the board generated. We construct three set of *Alldiff* constraint for the sudoku problem: there is no repeated number in each row, column and subgrid.

**Backtracking**

Backtracking is the natural approach to solve a CSP problem. Backtracking recursively searches/assign value for each variable in a DFS manner until:

Fail - backtrack to the last variable assignment if there is a variable with empty domain.

Success - All the variables are assigned.

It is guaranteed for backtracking to find the solution if it exists in the search space. But this is in the cost of performance since backtracking must exhaustively search the tree before a solution was found. The backtracking algorithm we implemented in C++ incorporate several optimization strategies. First, we applied MRV(minimum-remaining-value) heuristic to determine which variable to explore first. The motivation of MRV is to effectively reduce the branch at each level of the tree. Intuitively, if we pick variable with minimum value remaining in its domain every round, the branch factor at the top of the tree is greatly reduced, and the inference rule is likely to drop the branch factor at the bottom of the tree as well, therefore effectively prune the tree. AC-3 is another technique we applied to prune the initial tree by ensure arc-consistency between every two variables before running backtrack. The search space before and after running AC-3 contains the same amount of solutions, except the latter will be equal or smaller than the former in size, thus validating the point that AC-3 is “safe” to apply. We didn’t add AC-3 in the recursion of backtracking search due to the high expense of AC-3 and its conflicting role compare to forward checking. Forward checking is a way to inference the search space based on the last variable assignment. For the sudoku problem, the forward checking will remove the newly assigned value in the domain of every variable that is in the same row, column or subgrid with the newly assigned variable.

Our backtracking algorithm is only semi-optimized, there are many other available optimizations we could possibly explore. For variable and value selection, there are degree heuristic (tie breaker of MRV) and Least-constraining-value heuristic. Implementing backjumping and constraint learning could avoid bad selection of value and branch with a dead end (no solution). Sudoku puzzle also has its unique optimization technique such as naked single [3]. In summary, backtracking can be an efficient algorithm in solving sudoku, but semi-optimized backtracking fall quite short in front of the solver implemented by algorithm X/Dancing Link. In section III, we will introduce how to model sudoku as an exact cover problem and solve it by algorithm X.

**III. Exact cover problem/Algorithm X/Dancing link**

**Mathematical definition:**

If we have a set U, and a S as a collection of subset of set U, S is an exact cover of U if and only if every single element in set U is contained in exactly subset in S.

For example, Assume we have a set X = {1,2,3,4,5,6,7} and a collection S = {{1,3},{2,4,7},{3,5,6},{1,5,7},{1}} where the sets contained in S are subset of X. There is a subcollection S\* belong to S, and it is an exact cover respect to X if and only if the union of all the subsets inside S\* are equivalent to X, and their intersection is an empty set. In this case, the subcollection S\*= {{2,4,7},{3,5,6},{1,5,7}}.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| {1,3} | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| {2,4,7} | 0 | 1 | 0 | 1 | 0 | 0 | 1 |
| {3,5,6} | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| {1,5,7} | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| {1} | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

Fig.2 *example matrix for exact cover problem*

**Algorithm X**

To find an exact cover of X from collection S, Donald Knuth proposed algorithm X in 2011. To apply algorithm X, we can represent the problem as a matrix M in Fig 2. Each column of the matrix denotes one element in the set X, and each row represents a subset inside collection S. A one in the matrix means the corresponding row/subset contains the elements represented by the column, and 0 means not. The pseudocode of algorithm X is shown in several steps:

1. select a column j which has minimum amount of 1s

2. choose a row i in j where M[i][j]=1 and delete itself and every other row that shares a one in any columns with the selected row

3. Delete the column where the choose row has a “one” on.

4. recursively call the algorithm with a reduced matrix.

5. if the recursive call returns false, recover the original matrix

6. do step 2 with a different row i

7. if the recursive call returns false for every single row selection i, return false

Based condition:

1. if the matrix is empty, return true

2. if the matrix has an empty column, return false

Like backtracking, Algorithm X is guaranteed to find the solution if there is an exact cover subcollection exists in the matrix M.

**Sudoku as an exact cover problem**

Modeling sudoku as an exact cover problem requires the construction of matrix M with four different constraints:

1. Row constraints 2. Column constraints 3. Cell constraints 4. Subgrid constraints

Each constraint yields a matrix, and the final matrix M is concatenated by four sub-matrix side by side. A basic rule is that the column represents constraints and the row represent a possible entry/number placement for each cell, which means there are 16 rows for a single cell in the case of 16x16 sudoku. To illustrate this procedure, we could use the matrix for row constraint as an example, in Fig. 3, G[0][0][1] and G[0][1][1] conflicts because they contains the same constraint and they are in the same row. It means we cannot place same number in the same row or there will be two “one”s in one column. If we arrange the matrix in this pattern, cells in the same row cannot have the same value where cannot coexist with .If x=y. The matrix for row constraint alone has 256 columns and 4096 rows. Since we have four constraints in total we could apply the same methodology and yield four matrices with the same size. For other three constraints, their constraint matrices are shown in Fig.4, Fig.5 and Fig.6 respectively [1]. The final matrix is shown in Fig 7.

**Dancing link(DLX)**

Dancing link is a data structure proposed by Donald Knuth in 2011 to accommodate the operations in algorithm X, especially for step 2 and step 5. Dancing link is a two-dimensional link list that’s resemble the matrix M in Fig 7, where each node in the link list represents a 1 in the cell of matrix M, where save the time for row and column traversal. It has an additional header row at the top for the column travel and deletion. If we treat each row or column as a one-dimensional link-list, the tail and head of the link-list is connected. Each node has reference toward its neighbor node, include the one on the top, bottom, left and right, but not the diagonal. A detailed deletion operation in dancing link is described in this article at this website *http://www.geeksforgeeks.org/exact-cover-problem-algorithm-x-set-2-implementation-dlx/* [2].

Dancing link plays a neat trick to simplify the deletion and recover operation in algorithm X. The node removed in the dancing link are physically removed, but will not get deleted. The removed node is merely hidden from rest of the data structure, but the node itself still maintains reference to the dancing link. If we have the reference to a deleted node, pointer manipulation is enough to add the node back to the main data structure and dynamic memory allocation is unnecessary.

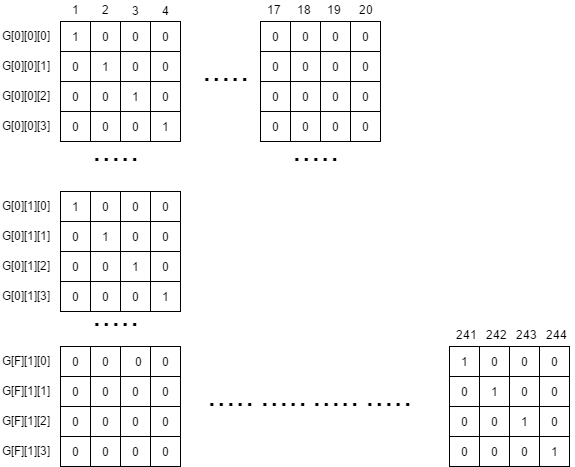


Fig.3 *matrix for row constraint alone*

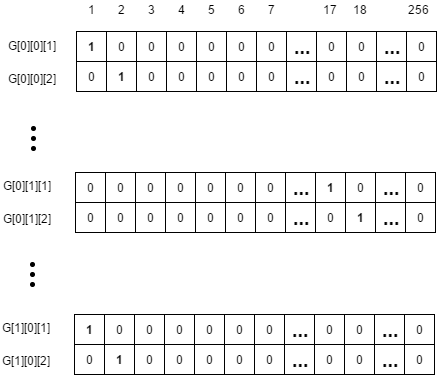


Fig.4 *matrix for column constraint*

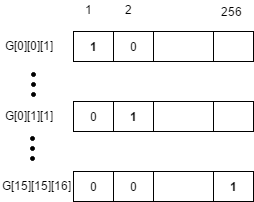


Fig.5 *matrix for cell constraint*

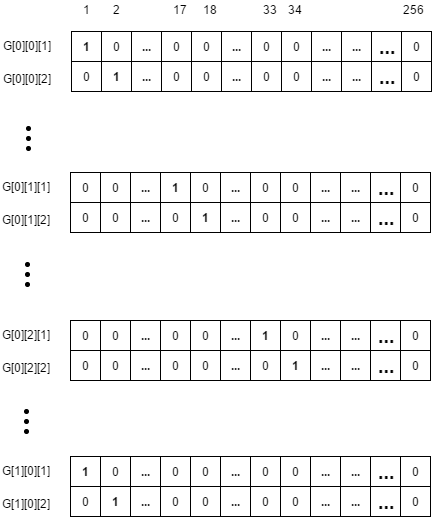


Fig.6 *matrix for subgrid constraint*

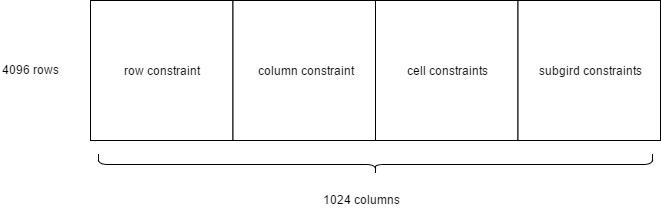


Fig.7 *matrix for 16x16 sudoku problem*

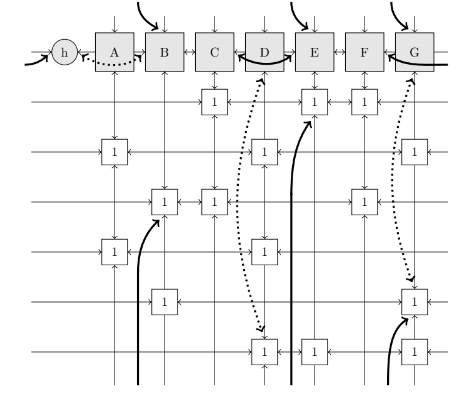


Fig.8 *Dancing link*

**IIII. Tool**

All the programs are implemented in C++. We choose C++ because it is more convenient than C with standard library and has superior performance compared to other high-level languages such as java/python. C++ is also the natural choice to implement dancing link because there is lot of pointer manipulation in algorithm X and C++ has built in pointer type. We mainly use visual studio as the IDE for development, but there is also a makefile included since part of the testing is performed under Linux based OS.

**IIIII. Performance measurement**

Since we only implemented solver program, all the test cases are retrieved from online resource/generator and there is no strict definition for difficulties in the generated test case. We set up test cases with 15 different game boards and measure the average time consumed by both algorithm. DLX has an average better performance compared to semi-optimized backtracking.

|  |  |  |
| --- | --- | --- |
| Backtracking | DLX | Speedup |
| 41.236s | 0.0879s | 46912.4% |

Fig.9 *Dancing link*

We want to exploit the potential of DLX by applying problem with larger search space. We input sudoku with 36x36 and 49x49 board size, the increase of computational time is in exponential relation with the size of board. This is expected since algorithm X is essentially a backtracking algorithm with a different problem search space.

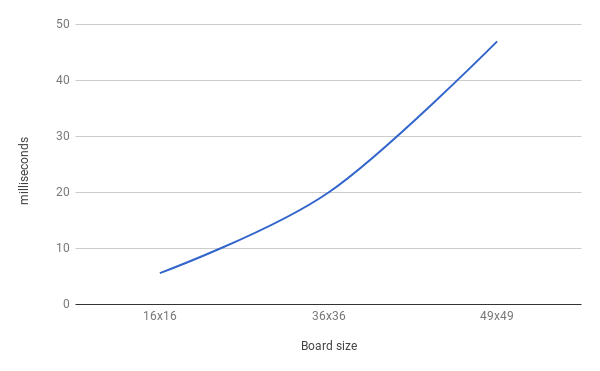


Fig.10 *Dancing link*

In conclusion, we believe it is better to model sudoku as an exact cover problem which yield a smaller search space compared to CSP. However, fully optimized backtracking algorithm with tree branch sufficiently pruned might yield performance as good as algorithm X. In the future, we should explore possible optimizations for dancing link consider both backtracking and algorithm X are DFS type of search and could share similar optimization techniques.

**Reference**

[1] M,Harrysson and H,Laestander, Solve sudoku efficiently with dancing link, 2014,KTH,DD144X,Avaliable at [*https://www.kth.se/social/files/58861771f276547fe1dbf8d1/HLaestanderMHarrysson\_dkand14.pdf*](https://www.kth.se/social/files/58861771f276547fe1dbf8d1/HLaestanderMHarrysson_dkand14.pdf)

[2] Exact Cover Problem and Algorithm X,*http://www.geeksforgeeks.org/exact-cover-problem-algorithm-x-set-2-implementation-dlx/*

[3] Simon Armstrong: SadMan Software: *http://www.sadmansoftware.com/ sudoku/solvingtechniques.htm*

**Appendix. A Benchmark Results(DLX)**

Platform: windows 10

Hardware: intel 6770HQ

hexa-20

Solution:

1 2 0 4 b f 6 9 c d e a 8 3 7 5

3 e c b 8 d a 1 f 0 5 7 2 4 6 9

a 5 9 7 2 3 e 4 6 1 8 b d 0 f c

8 f d 6 c 0 7 5 3 9 2 4 1 b a e

6 b 8 2 7 1 f e 9 3 a d 4 5 c 0

4 a 7 1 0 9 5 8 2 e 6 c 3 d b f

e 9 3 5 4 6 c d b f 0 8 a 1 2 7

0 d f c 3 b 2 a 1 4 7 5 6 9 e 8

b c 4 e a 2 d f 8 5 1 3 9 7 0 6

7 3 6 f 9 e 8 0 d a b 2 5 c 1 4

5 8 a 9 6 4 1 3 0 7 c f b e d 2

2 0 1 d 5 c b 7 e 6 4 9 f a 8 3

c 6 2 3 e a 0 b 4 8 9 1 7 f 5 d

d 7 b 8 f 5 9 6 a c 3 e 0 2 4 1

9 1 e 0 d 7 4 2 5 b f 6 c 8 3 a

f 4 5 a 1 8 3 c 7 2 d 0 e 6 9 b

CPU running time: 0.011747s node traveled: 159

hexa-21

Solution:

a 7 6 3 2 5 9 f b c 1 0 8 4 d e

c e 9 f b 7 0 4 6 d 3 8 a 2 1 5

2 0 d b 3 8 1 6 a 4 5 e f 7 9 c

8 1 4 5 d e a c 7 f 9 2 6 0 3 b

d f c 9 a 4 e 1 5 2 0 3 b 6 7 8

4 a 5 6 f 3 8 9 c 7 b d 0 e 2 1

1 3 0 2 7 6 b 5 f 8 e a 9 d c 4

7 b 8 e 0 2 c d 9 1 4 6 5 a f 3

3 2 a 1 8 9 4 7 e b d f c 5 0 6

0 9 b 4 e c 6 3 1 5 8 7 d f a 2

f 5 e c 1 0 d b 2 a 6 9 4 3 8 7

6 8 7 d 5 f 2 a 3 0 c 4 1 b e 9

b 6 1 0 9 d f e 8 3 2 5 7 c 4 a

e c 2 a 6 1 7 0 4 9 f b 3 8 5 d

9 d 3 7 4 b 5 8 0 e a c 2 1 6 f

5 4 f 8 c a 3 2 d 6 7 1 e 9 b 0

CPU running time: 0.021028s node traveled: 170

hexa-50

Solution:

4 1 2 a 7 6 d 3 f 0 b 5 e 8 9 c

b 8 3 d 5 f c e 7 1 6 9 a 0 2 4

7 f 5 e 0 4 9 a c 3 2 8 d 6 1 b

0 9 c 6 b 2 8 1 d 4 e a 5 f 3 7

a 6 1 f c e 3 5 4 9 d 7 0 b 8 2

3 7 8 c 2 d 6 9 0 a 5 b 4 1 e f

d 4 0 5 8 1 f b 2 6 c e 9 7 a 3

e b 9 2 4 a 0 7 3 8 1 f 6 c d 5

1 3 f 7 e 9 5 8 b d 0 4 2 a c 6

2 c b 9 a 7 4 0 5 e 8 6 3 d f 1

6 d e 4 1 3 b f a 2 7 c 8 5 0 9

8 5 a 0 d c 2 6 9 f 3 1 b 4 7 e

c e d b 9 0 a 4 6 7 f 2 1 3 5 8

5 a 4 1 3 b 7 2 8 c 9 0 f e 6 d

9 0 6 3 f 8 e c 1 5 4 d 7 2 b a

f 2 7 8 6 5 1 d e b a 3 c 9 4 0

CPU running time: 0.023317s node traveled: 163

hexa-80

Solution:

a e f 0 5 3 9 2 1 8 c b d 6 7 4

3 9 d 4 c b e 6 f 2 5 7 a 1 0 8

6 c 5 8 1 4 0 7 a d 9 e 2 b 3 f

1 7 2 b a d f 8 6 3 4 0 9 e 5 c

9 5 8 6 e 2 7 f c b 1 a 3 d 4 0

b f c e 9 6 5 a d 0 3 4 8 7 2 1

4 2 1 a 3 0 c d 5 7 6 8 f 9 e b

7 0 3 d 8 1 b 4 2 f e 9 c 5 a 6

2 b 4 c 6 e 3 5 9 1 7 d 0 f 8 a

e a 9 3 2 8 d c b 6 0 f 5 4 1 7

f 1 0 5 b 7 a 9 4 e 8 2 6 c d 3

8 d 6 7 4 f 1 0 3 5 a c b 2 9 e

c 3 e 1 d a 6 b 8 4 2 5 7 0 f 9

5 6 7 9 f c 4 e 0 a d 3 1 8 b 2

0 8 a f 7 5 2 1 e 9 b 6 4 3 c d

d 4 b 2 0 9 8 3 7 c f 1 e a 6 5

CPU running time: 0.016802s node traveled: 184

hexa-81

Solution:

a 9 3 7 4 d 1 2 8 f 6 b e 5 0 c

6 0 e f 8 9 b a 5 2 7 c d 1 4 3

d 2 1 b 3 6 c 5 e 0 4 a 7 f 8 9

c 4 8 5 e 0 f 7 1 3 9 d 6 b a 2

8 b 9 e 0 3 7 4 c 5 d f 2 a 6 1

f 1 d c 5 a 2 6 0 7 b 9 3 8 e 4

2 5 4 0 f 1 8 d 3 6 a e 9 7 c b

7 3 a 6 c e 9 b 2 4 1 8 5 d f 0

4 7 5 2 a f 0 e d 8 3 1 b c 9 6

0 e b a 6 8 d c 7 9 f 4 1 2 3 5

3 d f 1 2 b 5 9 6 a c 0 4 e 7 8

9 6 c 8 7 4 3 1 b e 5 2 a 0 d f

e 8 7 3 d c 6 f 4 b 2 5 0 9 1 a

1 a 6 4 b 5 e 0 9 c 8 7 f 3 2 d

b c 2 9 1 7 a 3 f d 0 6 8 4 5 e

5 f 0 d 9 2 4 8 a 1 e 3 c 6 b 7

CPU running time: 0.016552s node traveled: 162

hexa-82

Solution:

8 d 6 2 a 5 c 7 e 4 0 3 f 1 b 9

1 7 4 3 8 2 d 9 b 6 a f 0 5 e c

0 e 5 f 1 b 4 3 9 d 8 c 6 a 7 2

b c a 9 6 e 0 f 2 1 7 5 8 d 4 3

5 f 9 b 7 6 a 2 3 c d 8 e 4 0 1

d 6 e 4 c f 3 5 1 0 b 2 9 8 a 7

7 2 0 a d 1 8 e f 9 4 6 b c 3 5

c 8 3 1 b 4 9 0 7 a 5 e 2 f 6 d

e a 8 6 9 3 b d c 2 f 0 5 7 1 4

f 0 2 c e 7 5 4 8 3 6 1 a 9 d b

4 3 7 5 0 8 1 6 a b 9 d c e 2 f

9 1 b d f c 2 a 4 5 e 7 3 6 8 0

2 9 c 8 5 d 7 b 0 e 1 a 4 3 f 6

3 5 f e 4 0 6 8 d 7 2 9 1 b c a

a 4 d 0 3 9 e 1 6 f c b 7 2 5 8

6 b 1 7 2 a f c 5 8 3 4 d 0 9 e

CPU running time: 0.02844s node traveled: 1092

hexa-100

Solution:

7 0 e 3 4 8 a d 5 9 1 b f 6 c 2

6 b 1 8 e c 5 f 7 0 3 2 9 a 4 d

5 4 a d 2 9 b 3 f 6 c e 0 8 7 1

2 c f 9 6 7 0 1 a 8 4 d e 3 5 b

d e 8 7 b 3 f 0 1 2 9 a 5 4 6 c

0 6 c 2 a e 1 4 d 5 8 7 b f 9 3

4 5 9 b c d 2 7 0 3 6 f 8 1 a e

1 f 3 a 9 6 8 5 c e b 4 d 0 2 7

a d 4 6 0 2 e b 3 1 7 9 c 5 f 8

8 3 0 5 d f 7 6 2 b a c 4 e 1 9

9 1 b f 3 5 4 c 6 d e 8 7 2 0 a

e 7 2 c 1 a 9 8 4 f 0 5 3 b d 6

f 2 d e 7 b 6 a 9 4 5 3 1 c 8 0

b a 6 0 5 4 c e 8 7 d 1 2 9 3 f

3 9 5 1 8 0 d 2 b c f 6 a 7 e 4

c 8 7 4 f 1 3 9 e a 2 0 6 d b 5

CPU running time: 0.020423s node traveled: 205

hexa-101

Solution:

0 7 b e a 6 d 4 f 8 9 1 c 3 2 5

c 9 3 f 2 8 5 e 7 6 a 0 4 1 d b

1 5 6 a 7 c 0 3 d 2 b 4 f 9 e 8

2 4 8 d 1 b f 9 c e 5 3 6 0 7 a

e 8 1 3 9 7 c d a b 6 5 0 2 4 f

b c d 4 e a 6 5 8 f 0 2 9 7 1 3

9 a 5 2 3 0 b f 4 1 d 7 e 8 6 c

6 0 f 7 4 2 8 1 3 9 c e a 5 b d

f d 2 9 0 3 e 6 5 a 1 8 7 b c 4

5 6 7 c 8 9 4 2 0 d e b 3 a f 1

a e 4 b 5 f 1 c 2 3 7 6 8 d 0 9

3 1 0 8 b d a 7 9 c 4 f 2 e 5 6

7 f a 5 6 e 9 b 1 0 8 c d 4 3 2

d 3 e 6 f 5 7 a b 4 2 9 1 c 8 0

8 2 c 1 d 4 3 0 e 5 f a b 6 9 7

4 b 9 0 c 1 2 8 6 7 3 d 5 f a e

CPU running time: 0.021719s node traveled: 214