Internal Report 2012-04 Augustus 2012

Universiteit Leiden

Opleiding Informatica

Shikaku

Stefan Schrama

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Shikaku

S.C.A. Schrama

August 26, 2012

Contents

1 Introduction

The focus of this thesis lies on the Japanese puzzle called Shikaku. Dr. J.H.P. Kwisthout and Dr. H.C.M. Kleijn acted as my supervisors.

I got the idea for my thesis from my friend Timo Morsink. He wrote his thesis about Hashiwokakero [4], also a Japanese puzzle. Of course it wouldn't be much of an effort to do the same thing, so I looked for another Japanese puzzle and I found Shikaku. I hadn't seen it before and couldn't find much research on the puzzle, so I decided to give it a try myself.

Ever since Sudoku became popular outside of Japan, more and more Japanese puzzles have made their way outside of Japan. The company behind these puzzles is *NIKOLI Co., Ltd.*. With a current staff of only 25, they publish a quarterly puzzle magazine with over 30 different kind of puzzles [5].

The published articles aren't always created by the staff of Nikoli, everybody can contribute to the magazine. The puzzles are of course reviewed by the staff before publishing. The reason why these puzzles can become so popular all over the world, is that it doesn't require a specific language to solve! All the puzzles require only logic thinking to solve.

In this thesis the puzzle Shikaku will be explained and something will be said about the background of the puzzle. Questions like how it was conceived and why these puzzles are often Japanese are answered. The focus of this thesis is to make an effective solver for Shikaku using heuristics. The code of this program is available in Appendix B. The results of the heuristics will be discussed in Section 4. Furthermore possible tactics for creating a generator will be given and some real world applications for the puzzle are discussed. The puzzles used to test the algorithm and the input files for the algorithm can be found in respectively Appendix A and C. This thesis ends in a conclusion and suggestions for future work.

2 What is Shikaku?

2.1 Rules

The Shikaku puzzle exists of an $n \times m$ grid with numbers on it (Figure 1). As there are no Shikaku puzzles with number 1 on it, the smallest possible puzzle consists of two cells, either on top of eachother or next to eachother.

There are four basic rules in Shikaku:

1. Divide the grid into rectangles.

2. Each rectangle should contain exactly one number.
3. The number indicates how many cells are contained in that rectangle.
4. Rectangles may not overlap.

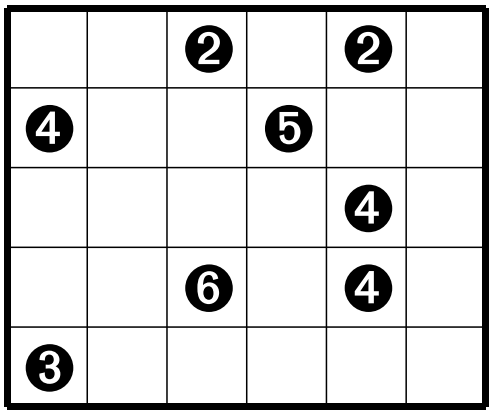


Figure 1: Example puzzle

With these rules you can complete the puzzle as is shown in Figure 2.

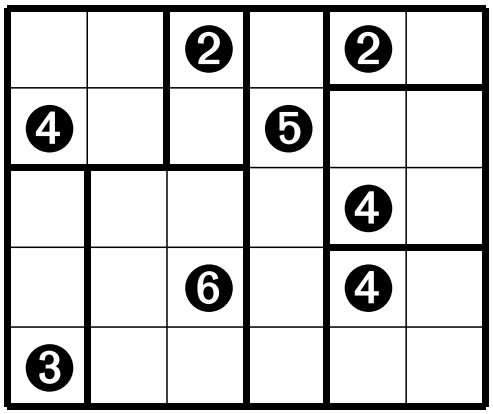


Figure 2: Example puzzle completed

2.2 Origins

Shikaku is, like most of these kind of puzzles of Japanese origins. The creator of the puzzle is Nikoli Co., Ltd.[5] a publishing company which published its first puzzle magazine in 1980. They claim their most famous puzzle is Sudoku[6].

In answer to a question about the origins of Shikaku, Junji Takeuchi, a staffmember at Nikoli, answered the following. "Shikaku was created by Yoshinao Anpuku in 1989 and published on 'Puzzle Communication Nikoli Vol.27' issued at Sep 1989. Anpuku is Nikoli's staff now. He got the idea from the classic figure puzzle called 'Tachiawase puzzle' in Japan. On that puzzle, a solver will cut a given figure and reassemble different shape. Japanese enjoy that puzzle since more than 100 years ago. He created Shikaku by taking away the element of reassembling from original rules and limiting ways of cutting to rectangular cut to give unique solution." [9]

Many logic puzzles have Japanese roots. Actually, the publisher Nikoli has gained worldwide recognition after their launch of Sudoku in Europe and the United States. Because of the success of Sudoku, other puzzles from this publisher have become more popular. This could explain why these type of puzzles are of Japanese origins.

2.3 Japanese puzzles in Computer Science

There are a lot of research papers and theses available concerning different types of logic puzzles from Japan. A lot of these focus on the solving or generating of these puzzles. At LIACS, Leiden University there are some good examples of theses in computer science concerning Japanese puzzles. For instance *Hashiwokakero* by T. Morsink [4], *Takegaki* by R. van Dam [2] and the research paper *A Discrete Tomography Approach to Japanese Puzzles* by K.J. Batenburg and W.A. Kosters [1].

In these papers different approaches to solve the puzzle are discussed and implemented. Because of the logical nature of the puzzles, logical rules are used to solve the puzzles. This is also the main reason for computer scientists to study Japanese puzzles, or puzzles in general. What is the best way to solve the puzzle or how can these puzzles be created? In the case of the classic Japanese puzzle, or *Nonogram*, there is even an aesthetic aspect. As Batenburg and Kosters [1] describe in their article, there are ways to convert images (even colour images) into Nonograms.

Although not in the field of computer science *Deductive Puzzling* by J.J. Wanko [10] does deserve mentioning, because both puzzles discussed form the motivation for writing this thesis. In the article the use of logical puzzles like Shikaku and Hashiwokakero is discussed to improve the deductive reasoning skills of middle school students.

2.4 Complexity

The complexity of mathematical problems is studied to see if a problem can be solved and if so, how hard it is to solve the problem. This is done using Turing machine. There are several kinds of Turing solvable problems. The easy solvable class P problems are problems which are solvable in polynomial time using a deterministic Turing machine. The easy verifiable class NP problems are problems for which given answers are easy to check for validity. NP problems are solvable in polynomial time using a nondeterministic Turing machine.

The class of NP problems contains a subset of problems called NP-complete. These are problems to which every problem in NP can be converted to. NP-hard is set of problems that can be reduced from an NP-complete problem in polynomial time, but does not have to be an NP-complete problem by itself. This means that an NP-hard problem is at least as difficult as an NP-complete problem.

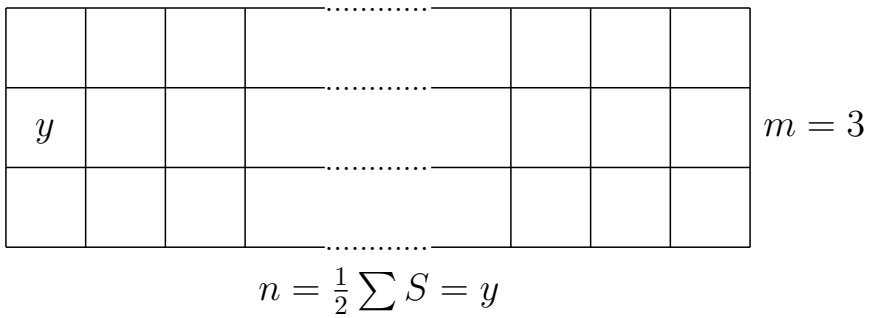
Most common puzzles and games have had their complexity proven in some way. Although quite popular in Japan, Shikaku still has been mostly overlooked in the scientific community. A proof of it's complexity is therefore not yet available. A complete proof of the complexity of Shikaku turned out to be too much for this thesis. However, an NP-hardness proof for a more simplified version of Shikaku is not too difficult to give. In this unconstrained version of Shikaku not all rectangles have to contain a number. Using a reduction from Partition (a proven NP-complete problem) in polynomial time, defined as follows [3]:

Partition: Given a multiset S of integers, can S be partitioned in S_1 and S_2 such that $\sum S_1 = \sum S_2 = \frac{1}{2}\sum S$?

Unconstrained Shikaku: Can we place t tiles with area $\{x_1, \ldots, x_k, \ldots, x_t\}$ on an $n \times m$ board with k numbers ($k \leq t$), where tiles with area x_i must be placed on a cell with number i; note that there may be less numbers on the board than tiles; when all numbers are covered, the remaining tiles may be placed freely.

Reduction from partition: Let $\{x_1, \ldots, x_t\} \cup \{x_y\}$ divide the tiles, with $\{x_1, \ldots, x_t\} = S$ and $y = \frac{1}{2}\sum S$. Let the $m \times n$ board be as follows: $m = 3, n = \frac{1}{2}\sum S$.

Note that x_y can only be placed in the middle row, as this is the only row with a numbered cell and x_y won't fit anywhere else. Note that for the remaining



elements in S to be placed on the board, they need to be partitioned in two subsets, each with total area $\frac{1}{2}\sum S$. It is easy to see that $\{x_1, \ldots, x_t\}$ can be placed on the board if and only if the corresponding partition problem has a solution.

Although this does not prove Shikaku is NP-hard, it is to be expected that as this version of Shikaku is NP-hard, the real Shikaku is NP-hard as well.

3 Solving the puzzle

To solve the Shikaku puzzle there are different approaches that can be taken. Examples are *brute-force*, *evolutionary algorithms*, and *backtracking*. To more easily incorporate the heuristics (Section 3.2), backtracking is the approach chosen.

Firstly the main algorithm is explained and the problems encountered. Secondly the heuristics to improve the algorithm are discussed.

3.1 Backtracking

As stated before, the easiest way to solve a puzzle like this is via backtracking. Just keep placing the rectangles until you can't and keep changing the last one until you can go further with the placing. In Section 3.1.1 the pseudo code for such a solver is presented.

3.1.1 Pseudo Code

The following pseudo code represents the main solving algorithm. For every number on the grid, there is a set of rectangles and all possible directions of that rectangle. For every number on the grid the algorithm tries the first possible rectangle containing that number. If there is a rectangle that can be placed, it is called a legal placement. If no rectangle can be placed, remove the last rectangle placed and place the next possible rectangle of that number. If

that number has no other possible rectangles keep undoing rectangles until there is a number with a rectangle that can be placed. When the last number has its rectangle placed, the algorithm ends.

for every number **do**
  for every rectangle of number **do**
    if placement is legal **then** next number
    else if placement isn't legal & ∃ next rectangle **then** try next rectangle
    else undo last number and try next rectangle of that number
    end if
  end for
end for

3.1.2 Problems

Despite its simple design, the algorithm does have its problems and limitations, mainly concerning memory usage. The first of these problems was a wrongly chosen data structure called *vector*. Although dynamically in nature, the freeing of memory space did not occur during execution, which resulted in the growing memory usage. To counter this problem the data structure was changed to a *pointer array* which does free its memory space during execution.

Although smaller puzzles ($m \times n$ with $m, n \leq 10$) are being solved now, the bigger puzzles still encountered memory problems. This is caused by the massive amounts of search space used. For a medium puzzle of size 10×10 the amount of rectangles in the search space can reach numbers in the 500's and for the larger puzzles this number is even bigger. Apparently this caused a big memory problem.

To reduce the search space, a function was written called calcrects (Appendix B), which first calculated all the rectangles per grid value that could be placed on the board legally if it is the first rectangle to be placed. The rest of the rectangles are discarded. This resulted in an enormous shrinking of the search space by up to 85 percent.

Even though the algorithm is now a lot faster and can solve more puzzles, it still has its limits. Due to the exponential increase in search space, the really large puzzles ($m \times n$ with $m, n \geq 15$) still pose problems for the algorithm. This final algorithm is from now on called *Normal*, as there are no heuristics implemented yet. The code can be found in Appendix B.

3.2 Heuristics

Now that the solver is working, it is time to see of the solving process can be sped up by implementing some heuristics.

3.2.1 Highest value first

Higher values on the grid usually have less rectangles with legal placements. Other values on the grid and the edges of the grid are more easily reached by these higher values, but the constraints of Shikaku hinder the rectangle to expand further in those directions. Because of the smaller number of possible rectangles it is recommended to start with these values.

3.2.2 Values with one option first

It is possible for a value on the grid to only have a single possible rectangle associated with it. To avoid replacing the same rectangle during the solving process, these rectangles are placed first. An example of this is shown in figure 3.

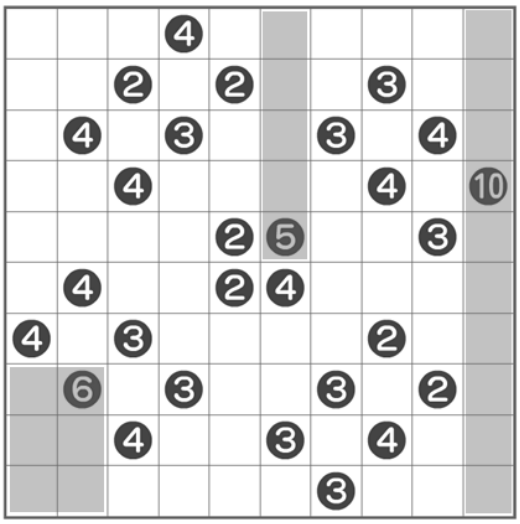


Figure 3: Numbers with only one option

3.2.3 Implementing the heuristics

Implementing the aforementioned heuristics proved to be quite simple. Because the algorithm works his way through an array of values, the only thing that has to be done is sort the array in the order desired.

For the "highest value first"-heuristic, the array is sorted from high to low using *insertion sort*. For the "values with one option first"-heuristic the number of possible rectangles are counted. If this number is one, the value is sorted to the front of the array. This is achieved by switching the value with its left neighbour until the front of the array is reached.

3.3 Testing

To test if the heuristics described in Section 3.2 are in fact an improvement to the solver, six sample puzzles are used (see Appendix A). All six samples are from the Nikoli puzzle website [6]. The first three sample are 10×10 puzzles and are marked as easy. The fourth sample is an 18×10 puzzle and is also marked as easy. The fifth and sixth puzzles are also 18×10, but are marked medium difficulty.

The testing is done on a Asus EeePC with an Intel® Atom™ CPU N450 at 1.66GHz and 1GB of system memory. The operating system used is Ubuntu 10.04 (lucid) with Linux kernel 2.6.32-42-generic.

4 Results

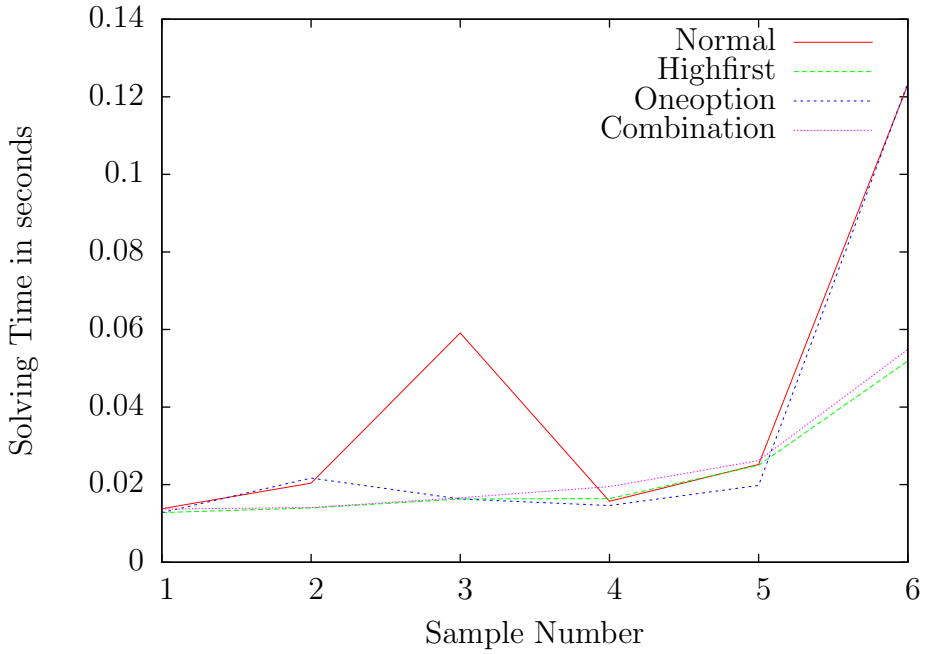


Figure 4: Execution times

To get a good indication of the time needed to solve a puzzle and rule out accidental slow solves due to rounding errors, every sample is solved 100 times per solver type (Normal (Section 3.1.1), Highfirst (Section 3.2.1), Oneoption (Section 3.2.2) and a Combination). The average execution time per sample is shown in Figure 4.

As can be seen in Figure 4, the Normal, unaltered algorithm does not always have the highest execution time. This is most likely because some puzzles already have a somewhat favorable design. That is to say, from left to right and from top to bottom the values are from high to low. In which case, the extra algorithm to sort the values of the board takes more time than the speed gained with it.

On most occasions the Oneoption heuristic is the quickest, but with sample six, the Highfirst algorithm is more than twice as fast as both the Normal and the Oneoption algorithm.

The Combination option is never the best way to solve the puzzle. The fact that extra calculations are needed, makes this option always slower than the fastest single option algorithm.

Ideally the solver should be compared to other solvers. Unfortunately there were no solvers available to test. The only references found as to how a solver worked was on the website The Shikaku Room [7] and in the presentation Shikaku as a Constraint Problem by Helmut Simonis [8].

On The Shikaku Room the creator explains he used the known logical rules to solve the puzzles, but speeds the process up using a so called Hilbert R-tree.

In his presentation Simonis describes how Shikaku can be described as a constraint problem. Although there is no solver available using this method, one could create one using the research Simonis presented.

5 Generating a puzzle

Generating a puzzle is not a trivial thing and there is more than one way to do this. One way is to randomly create rectangles within a given rectangular grid and then place the number corresponding with the surface area of the rectangle randomly inside it. The biggest problem with this approach is that you are very likely to end up with a lot of very small or $1 \times x$ rectangles to fill in the gaps of the created rectangles. Even an outcome with 1×1 rectangles is possible, which is not a valid Shikaku puzzle.

A different approach is to first divide the size of the surface area of the grid into smaller numbers, which you then use to create rectangles. This does eliminate the possibility of 1×1 rectangles, because you can choose not to

create these rectangles before creating the puzzle. However, the creation of the puzzle using these rectangles can take a while, because randomly placing rectangles does not always create a puzzle on the first go. Placing the values in their respective rectangles can be done after the creation of the rectangles, or after the rectangles have been placed on the puzzle grid. This of course can be done randomly within the rectangle.

6 Real world applications

Although solving puzzles using algorithms is an interesting subject to some people, the possibility of using these algorithms as a real world application can be attractive to a much larger group of people.

The concept of dividing spaces is something used in real life on a lot of occasions. Examples are office spaces, new housing development and you can even think of things on a smaller scale like the use of a garage or storage box.

To use the algorithm for assigning objects(offices, boxes etc.) to a space (building floor, garage etc.) there are some changes to be made. For these kind of uses of the algorithm, it isn't always important where an object is placed. Unlike the rectangles in Shikaku, which all have a single cell which has to be covered(the numbered cell). Furthermore, the shape of the object can be fixed, or at least have some restrictions. In case of a cardboard box the size and shape are fixed. When designing an office floor plan, the size of the offices are somewhat fixed with a minimum and maximum size. The shape however is far more important, you wouldn't want to end up with an office of 1×12 metres.

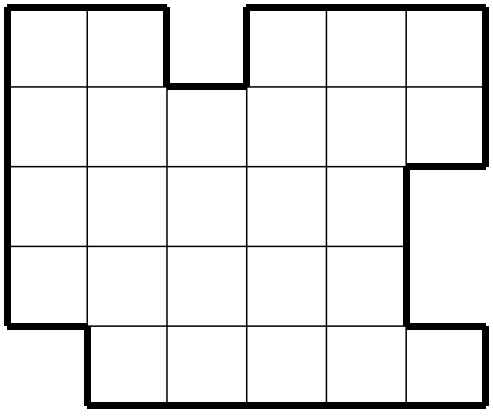


Figure 5: This kind of space could be made usable

There is however a problem that arises when using this algorithm. Due to the algorithm using integer arrays as a way to move through a surface you can

only divide rectangular spaces. Circular and other spaces with round shapes aren't common, but they do exist. Other non-rectangular spaces pose the same problem, although it is possible to use spaces which have rectangular indentations as in Figure 5.

7 Conclusion and future work

The results show that although the heuristics do have a mostly positive effect on the algorithm, there is still room for improvement. Some cells on the grid are only reachable by a single value and its corresponding rectangles, thus limiting the number of possible rectangles. One way to implement this is by creating a matrix containing each cell on the grid, each of those cells is then connected to every number on the grid that could reach that cell with at least one of it's rectangles. In the case that a cell in the matrix only has a single number connected to it, it is obvious that number has to cover that cell. Thus limiting the amount of rectangles for that number.

Another heuristic using more in depth possibilities in mind while placing a rectangle could improve the algorithm concerning actual steps within the algorithm, but is likely to take up more time. In addition to this, a question might be if the way the heuristics are evaluated, by the time it takes to solve a puzzle, is the best way to evaluate them. As mentioned in Section 4, there are some problems with getting an exact time. A better way might be to count the number of steps the algorithm takes to solve a puzzle. One could think of the amount of rectangle placements and or removals as a deterministic way to evaluate a heuristic.

It's possible that an algorithm using these heuristics speeds up the solving process even further. And with the code already provided in this thesis, the first steps have been made to create a perfect solver more easily. With further research it is also important to use more samples. By using only six samples, the impact of a puzzle in which high numbers or numbers with only one possible rectangle are already in the top or second row of the puzzle is a lot bigger. Using more samples would make this impact a lot smaller.

The setup of the solver (using the logical rules of the puzzle) is similar to the way the puzzles are solved on the website The Shikaku Room [7]. The main difference is the use of the Hilbert R-tree. This does seem to solve the problem with bigger puzzles, as the largest ones on the website are 20 x 20. Unfortunately the author does not share his code.

There are strategies provided to generate puzzles. And although not implemented in this thesis, others could use these strategies to make a puzzle generator.

More work and research can also be done on the real world application of the algorithm. An actual conversion or implementation could be written. By changing the function that creates the actual rectangles, it is also possible to create surfaces not limited to this form. Making other shapes usable by the algorithm.

8 Acknowledgements

I would like to thank my supervisors Johan Kwisthout and Jetty Kleijn for their help and guidance. With their help I was able to continue writing this thesis and improving the code for the solver. I would also like to thank my family for their continued support and encouragements, without them it would have been much harder to complete this thesis.

References

[1] K.J. Batenburg & W.A. Kosters, A Discrete Tomography Approach to Japanese Puzzles. In *Proc. Belgian-Dutch Conf. Artificial Intelligence*, 243–250, 2004

[2] R. van Dam, Takegagi. Unpublished bachelor's thesis, LIACS Leiden University, 2008-13 (2008)

[3] J. Kwisthout, personal communication (March 19th, 2012)

[4] T. Morsink, Hashiwakakero. Unpublished bachelor's thesis, LIACS Leiden University, 2009-11 (2009)

[5] Website Nikoli, `http://www.nikoli.co.jp/en/` Last retrieved August 23rd, 2012.

[6] Puzzle website Nikoli, `http://www.nikoli.com/en/` Last retrieved August 23rd, 2012.

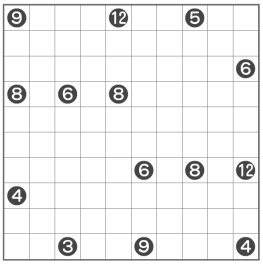
[7] Website The Shikaku Room, `http://www.shikakuroom.com/` Last retrieved August 23rd, 2012.

[8] H. Simonis, Shikaku as a Constraint Problem. CSCLP 2009 (2009)

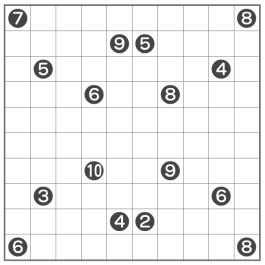
[9] J. Takeuchi, personal communication (March 15th, 2012)

[10] J.J. Wanko, Deductive Puzzling. *Mathematics Teaching in the Middle School*, 15:524-529 (2010)

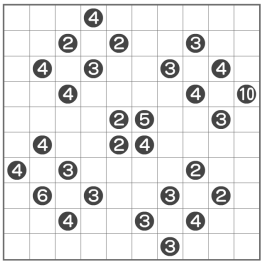
A Samples



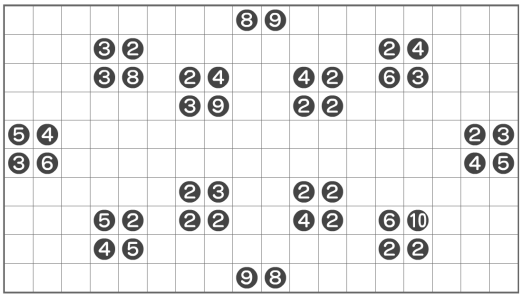
(a) Sample 1



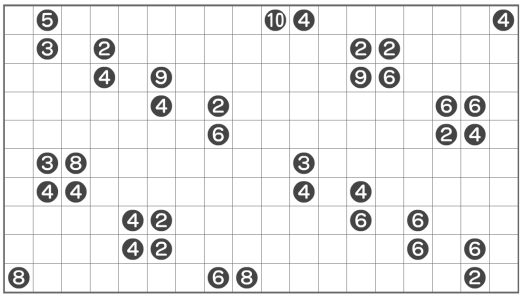
(b) Sample 2



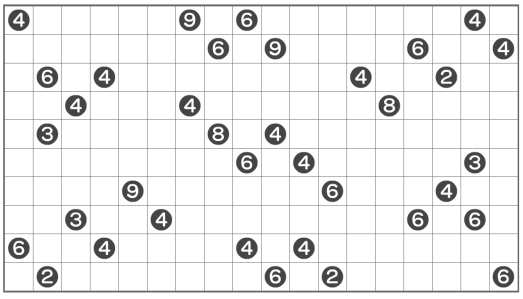
(c) Sample 3



(d) Sample 4



(e) Sample 5



(f) Sample 6

Figure 6: Samples

B Program code

```
#include <iostream>
#include <cstdlib>
#include <fstream>
#include <sstream>
#include <string.h>
#include <stdio.h>
#include <vector>
using namespace std;

class Shikaku {
public:
   void loadpuzzle();
      //load the puzzle from a file
   void printpuzzle();
      //print the puzzle
   void calcrects();
      //calculate all possible rectangles per gridvalue
   void backtrack(int rec);
      //the backtracking function
   void insertsort();
      //sort the value-array highest to lowest
   void onesort(int one);
      //move value with one possibility up the value-list
   void oneposs();
      //sort the value-array to place the easy values first
   int factoring(int f);
      //break a value down in factors
   void place(int num, int m1, int n1, int m2, int n2);
      //place a rectangle
   void undo(int val, int m0, int n0, int m1, int n1, int m2,
            int n2); //undo a placement
   bool checked(int m0, int n0, int m1, int n1, int m2, int n2);
      //check if placement is legal
private:
   ifstream file;
   string line;
   int n, m;
      //size of the grid n x m
   int ** v;
```

```
      //the puzzle
   int ** numbers;
      //all the numbers on the grid
   int count;
      //number of values on the grid
   int * factors;
      //array of factors of the current gridvalue
   int fsize;
      //size of factors
   vector< vector< vector< int > > > rects;
      //array of all possible rectangles
};//Shikaku

int Shikaku::factoring(int f) {
   //break a value down in factors
   int fsize = 1;
   for(int g = 1; g < f; g++){
      if((f%g) == 0) fsize++;
   }
   factors = new int[fsize];
   int i = 0;
   for(int g = 1; g < f; g++){
      if((f%g) == 0){
         factors[i] = g;
         i++;
      }
   }
   factors[i] = f;
   return fsize;
}//factoring

void Shikaku::undo(int val, int m0, int n0, int m1, int n1,
                   int m2, int n2) {
   //undo a placement
   for(int j = m1; j <= m2; j++){
      for(int i = n1; i <= n2; i++){
         v[j][i] = -1;  //set all values within
                        the rectangle to -1
      }
   }
   v[m0][n0] = val;  //place the correct value
```

```
                    back on the grid
}//undo

void Shikaku::place(int num, int m1, int n1, int m2, int n2) {
   //place a rectangle
   for(int j = m1; j <= m2; j++){
      for(int i = n1; i <= n2; i++){
         v[j][i] = num; //set all values within the
                        rectangle to num
      }
   }
}//place

bool Shikaku::checked(int m0, int n0, int m1, int n1, int m2,
                    int n2){
   //Check if the rectangle doesn't leave the grid or overlaps
   another value or rectangle
   if(m1<0 || n1<0 || m2>m-1 || n2>n-1) return false;
      //check for leaving the grid
   for(int j = m1; j <= m2; j++){
      for(int i = n1; i <= n2; i++){
         if(i == n0 && j == m0){
            if(i < n2) i++;
            else if (j < m2){
               i = n1;
               j++;
            }else return true;
         }
         if(v[j][i] != -1) return false;  //check for overlap
      }
   }
   return true;
}//checked

void Shikaku::insertsort() {
   //sort the array from high to low
   int i, j, tmp0, tmp1, tmp2;
   for(i = 2; i < count+1; i++){
      j = i;
      while (j > 1 && numbers[j-1][0] < numbers[j][0]) {
         tmp0 = numbers[j][0];
```

```
            tmp1 = numbers[j][1];
            tmp2 = numbers[j][2];

            numbers[j][0] = numbers[j-1][0];
            numbers[j][1] = numbers[j-1][1];
            numbers[j][2] = numbers[j-1][2];

            numbers[j-1][0] = tmp0;
            numbers[j-1][1] = tmp1;
            numbers[j-1][2] = tmp2;

            j--;
         }
      }
   }//insertsort

   void Shikaku::onesort(int one) {
      //sort a value to the front of the array
      int j = 0;
      int temp0, temp1, temp2;
      while(one-j-1 > 0){
         temp0 = numbers[one-j][0];
         temp1 = numbers[one-j][1];
         temp2 = numbers[one-j][2];
         numbers[one-j][0] = numbers[one-j-1][0];
         numbers[one-j][1] = numbers[one-j-1][1];
         numbers[one-j][2] = numbers[one-j-1][2];
         numbers[one-j-1][0] = temp0;
         numbers[one-j-1][1] = temp1;
         numbers[one-j-1][2] = temp2;
         j++;
      }
   }//onesort

   void Shikaku::oneposs() {
      //check if a value has only one placable rectangle
      if no other rectangles are placed
      int checkcount = 0;
      int i;
      for(i = 1; i <= count; i++){
         int fsize = factoring(numbers[i][0]);
```

```
int floc = numbers[i][4];
int dm = factors[floc];
int dn = numbers[i][0] / factors[floc];
int m0 = numbers[i][1];
int n0 = numbers[i][2];
int offset = 0;
int temp = numbers[i][3];
while(temp >= 0){
   temp -= dn;
   offset++;
}
offset--;
int m1 = numbers[i][1] - offset;
int n1 = numbers[i][2] - (numbers[i][3]%dn);
int m2 = m1 + dm - 1;
int n2 = n1 + dn - 1;
if(checked(m0,n0,m1,n1,m2,n2)){
   checkcount++;
}
if(checkcount > 1){
   checkcount = 0;
   numbers[i][3] = 0;
   numbers[i][4] = 0;
}else if(numbers[i][3] < numbers[i][0]-1){
   numbers[i][3]++;
   delete [] factors;
   i--;
}else{
   numbers[i][3] = 0;
   if(numbers[i][4] < fsize-1){
      numbers[i][4]++;
      delete [] factors;
      i--;
   }else{
      numbers[i][4] = 0;
      delete [] factors;
      if(checkcount == 1){
         onesort(i); //only one rectangle, thus sort
                     it to the front of the array
         checkcount = 0;
      }
```

```
         }
      }
   }
}//oneposs

void Shikaku::backtrack(int rec) {
   int m0 = numbers[rec][1];
   int n0 = numbers[rec][2];
   int rectangle = numbers[rec][5];
   int m1 = rects[rec][rectangle][0];
   int n1 = rects[rec][rectangle][1];
   int m2 = rects[rec][rectangle][2];
   int n2 = rects[rec][rectangle][3];
   bool legal = checked(m0,n0,m1,n1,m2,n2);
   if(legal){//rectangle can be placed
      place(rec, m1, n1, m2, n2);//place the rectangle
      if(count==rec){   //if this is the last rectangle,
                        the puzzle is solved
         cout << "Done!" << endl;
         printpuzzle();
         exit(1);
      }else{
      backtrack(rec+1);//else next value
      }
   }else{//can't place rectangle
      if(numbers[rec][5] < numbers[rec][6]-1){
         //if more possible rectangles for this value...
         numbers[rec][5]++;
         backtrack(rec);//backtrack next rectangle
      }else{//if no more possible rectangles for this value
         numbers[rec][5] = 0;
         int back = 1;
         while(back < rec){
            m0 = numbers[rec-back][1];
            n0 = numbers[rec-back][2];
            rectangle = numbers[rec-back][5];
            m1 = rects[rec-back][rectangle][0];
            n1 = rects[rec-back][rectangle][1];
            m2 = rects[rec-back][rectangle][2];
            n2 = rects[rec-back][rectangle][3];
            undo(numbers[rec-back][0],m0,n0,m1,n1,m2,n2);
```

```
               //remove rectangle from grid
            if(numbers[rec-back][5] < numbers[rec-back][6]-1){
               numbers[rec-back][5]++;
               backtrack(rec-back);
                  //try other rectangle from previous value
            }else{//else keep removing rectangles
               numbers[rec-back][5] = 0;
               back++;
            }
         }
      }
   }
}//backtrack

void Shikaku::calcrects() {
   //calculate all possible rectangles per gridvalue
   int countrects = 0;
   int countrectangles = 0;
   rects.push_back(vector< vector< int > > () );
   for(int i = 1; i <= count; i++){
      rects.push_back(vector< vector< int > > () );
      int fsize = factoring(numbers[i][0]);
      for(int j = 0; j < fsize; j++){
         int dm = factors[j];
         int dn = numbers[i][0] / factors[j];
         int m0 = numbers[i][1];
         int n0 = numbers[i][2];
         for(numbers[i][3] = 0; numbers[i][3] < numbers[i][0];
            numbers[i][3]++){ //calculate all the rectangles
            int offset = 0;
            int temp = numbers[i][3];
            while(temp >= 0){
               temp -= dn;
               offset++;
            }
            offset--;
            int m1 = numbers[i][1] - offset;
            int n1 = numbers[i][2] - (numbers[i][3]%dn);
            int m2 = m1 + dm - 1;
            int n2 = n1 + dn - 1;
            countrectangles++;
```

```
            if(checked(m0,n0,m1,n1,m2,n2)){
               //if this rectangle could be placed
               countrects++;
               int k = numbers[i][6];
               rects[i].push_back(vector< int > () );
                  //add the rectangle to the array
               rects[i][k].push_back(m1);
               rects[i][k].push_back(n1);
               rects[i][k].push_back(m2);
               rects[i][k].push_back(n2);
               numbers[i][6]++;
            }
         }
      }
      delete[] factors;
   }
}//calcrects

void Shikaku::printpuzzle() {
   //print the puzzle grid
   for(int j = 0; j < m; j++){
      cout << "-";
      for(int i = 0; i < n; i++){
         cout << "---";
      }
      cout << endl << "|";
      for(int i = 0; i < n; i++){
         if(v[j][i]==-1) cout << "  |";
         else if(v[j][i]<10 && v[j][i]>0){
            cout << "0" << v[j][i] << "|";
         }
         else cout << v[j][i] << "|";
      }
      cout << endl;
   }
   cout << "-";
   for(int j = 0; j < n; j++){
      cout << "---";
   }
   cout << endl;
}//printpuzzle
```

```
void Shikaku::loadpuzzle() {
   //Check if file is available
   file.open("FILENAME", ios::in);
      //replace FILENAME with the puzzlefile
         with format as in Appendix C
   if(!file) {
      cerr << "Unable to open puzzlefile." << endl;
      exit(1);
   }

   getline(file, line);
   stringstream ss(line);
   ss >> m >> n;
   v = new int*[m];
   for(int i = 0; i < m; i++){
      v[i] = new int[n];
      for(int j = 0; j < n; j++){
         v[i][j] = -1;
      }
   }

   count = 0;

   while(getline(file, line)){
      istringstream d(line);
      string token;
      int h,i,j;
      getline(d, token, ',');//Get the number
      istringstream number(token);
      number >> h;
      getline(d, token, ',');//Get the n-position
      istringstream posn(token);
      posn >> i;
      getline(d, token, ',');//Get the m-position
      istringstream posm(token);
      posm >> j;
      v[i][j] = h;//Set the number on the n,m position
      count++;
   }
```

```
   file.close();

   numbers = new int*[count+1];
   int k = 1;
   for(int j = 0; j < m; j++){
      for(int i = 0; i < n; i++){
         if(v[j][i]!=-1){
            numbers[k] = new int[7];
            numbers[k][0] = v[j][i];
               //the gridvalue
            numbers[k][1] = j;
               //the m-location of the value
            numbers[k][2] = i;
               //the n-location of the value
            numbers[k][3] = 0;
               //location of original value within rectangle
            numbers[k][4] = 0;
               //location of factor in use
            numbers[k][5] = 0;
               //rectangle in use
            numbers[k][6] = 0;
               //number of rectangles
            k++;
         }
      }
   }
}//loadpuzzle

int main() {
   Shikaku skk;
   skk.loadpuzzle();
   skk.printpuzzle();
   skk.insertsort();
      //use this function for the highest-value-first heuristic
   skk.oneposs();
      //use this function for the one-possible-rectangle heuristic
   skk.calcrects();
   skk.backtrack(1);
}//main
```

C Puzzle files

```
\\sample1_easy.txt
10 10
9,0,0
12,0,4
5,0,7
6,2,9
8,3,0
6,3,2
8,3,4
6,6,5
8,6,7
12,6,9
4,7,0
3,9,2
9,9,5
4,9,9

\\sample2_easy.txt
10 10
7,0,0
8,0,9
9,1,4
5,1,5
5,2,1
4,2,8
6,3,3
8,3,6
10,6,3
9,6,6
3,7,1
6,7,8
4,8,4
2,8,5
6,9,0
8,9,9

\\sample3_easy.txt
10 10
4,0,3
```

```
2,1,2
2,1,4
3,1,7
4,2,1
3,2,3
3,2,6
4,2,8
4,3,2
4,3,7
10,3,9
2,4,4
5,4,5
3,4,8
4,5,1
2,5,4
4,5,5
4,6,0
3,6,2
2,6,7
6,7,1
3,7,3
3,7,6
2,7,8
4,8,2
3,8,5
4,8,7
3,9,6

\\sample4_easy.txt
10 18
8,0,8
9,0,9
3,1,3
2,1,4
2,1,13
4,1,14
3,2,3
8,2,4
2,2,6
4,2,7
4,2,10
```

```
2,2,11
6,2,13
3,2,14
3,3,6
9,3,7
2,3,10
2,3,11
5,4,0
4,4,1
2,4,16
3,4,17
3,5,0
6,5,1
4,5,16
5,5,17
2,6,6
3,6,7
2,6,10
2,6,11
5,7,3
2,7,4
2,7,6
2,7,7
4,7,10
2,7,11
6,7,13
10,7,14
4,8,3
5,8,4
2,8,13
2,8,14
9,9,8
8,9,9

\\sample5_medium.txt
10 18
5,0,1
10,0,9
4,0,10
4,0,17
3,1,1
```

```
2,1,3
2,1,12
2,1,13
4,2,3
9,2,5
9,2,12
6,2,13
4,3,5
2,3,7
6,3,15
6,3,16
6,4,7
2,4,15
4,4,16
3,5,1
8,5,2
3,5,10
4,6,1
4,6,2
4,6,10
4,6,12
4,7,4
2,7,5
6,7,12
6,7,14
4,8,4
2,8,5
6,8,14
6,8,16
8,9,0
6,9,7
8,9,8
2,9,16

\\sample6_medium.txt
10 18
4,0,0
9,0,6
6,0,8
4,0,16
6,1,7
```

```
9,1,9
6,1,14
4,1,17
6,2,1
4,2,3
4,2,12
2,2,15
4,3,2
4,3,6
8,3,13
3,4,1
8,4,7
4,4,9
6,5,8
4,5,10
3,5,16
9,6,4
6,6,11
4,6,15
3,7,2
4,7,5
6,7,14
6,7,16
6,8,0
4,8,3
4,8,8
4,8,10
2,9,1
6,9,9
2,9,11
6,9,17
```