Theory of Computer Games

Homework#1 - Dominosa

R04922058 鄭以琳

January 22, 2016

1 Game Description

Dominosa is a logic puzzle. It is played on a board with $(N+1) \times (N+2)$ numbers on it. Each number is an integer between $0 \sim N$. The player is given $\frac{(N+1)\times(N+2)}{2}$ domino bones. Each domino is a 1*2 rectangle containing distinct pair of numbers $(i, j), 0 < i \leq j \leq N$. The object is to locate all the domino bones onto the board. A bone can only be put at a location where the numbers on the bone is the same as the numbers on the board.

Figure 1

Take Figure **??** for example. The player is given a (7×8) board on the left and 28 domino bones on the right. Figure **??** shows one valid solution.

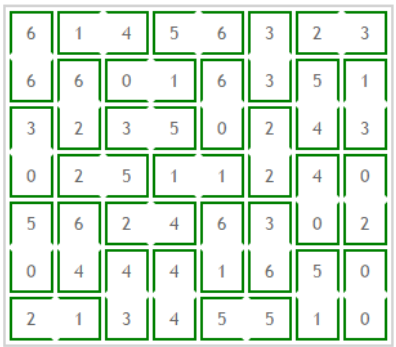

Figure 2

Note that for an $(N+1) \times (N+2)$ board, the maximum number on the bones will be N and each domino bone is used exactly once in the solution.

In my homework, I transformed this problem into another identical one:

> Given an $(N+1) \times (N+2)$ board, two adjacent grids can be circled to form a pair. Find a way to circle each grid exactly once, and let each pair of numbers $(i, j), 0 < i \leq j \leq N$ also appears exactly once.

2 Game Analysis

Naive Brute-force Approach

A naive searching algorithm is brute-force search, employing DFS to find every possible way of circling the numbers. Start from the top-left grid, find an uncircled adjacent grid and pair with it. In the next state, pick the first uncircled grid in row major and continue DFS. For an uncircled grid, there are at most four options to pair with, yielding a branching factor of DFS tree $e = 4$. For an $(N+1) \times (N+2)$ board, the player should circle $\frac{(N+1)\times(N+2)}{2}$ pairs. So the depth of a solution $d = \frac{(N+1)\times(N+2)}{2}$.

Although brute-force method is guaranteed to find a solution, it's inefficient. Worst case time complexity of this algorithm is $O(e^d) = O(4^{N^2})$.

There are about 10^{12} different valid ways to circle the numbers on a (10×11) board (calculated by DP method). When N becomes larger, the number grows dramatically. When $N = 15$, there are 10^{18} ways, which is 10^6 times larger than $N = 9$. A pattern database may not be able to store all these options. Even with enough memory, this approach may still be very time-consuming.

Useful Properties

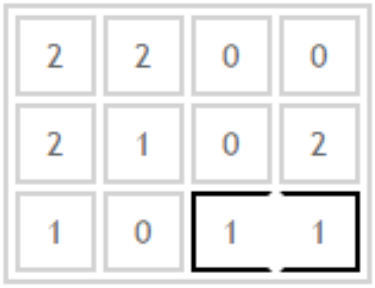

Figure 3

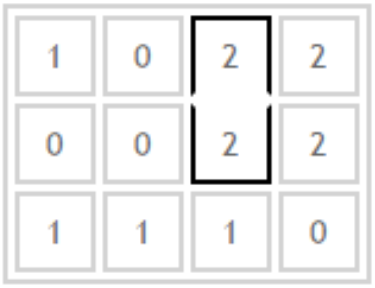

Figure 4

Forced Locations The constraint that each pair of numbers should be circled once may create some forced locations. Consider Figure **??**, the pair $(1, 1)$ can only be found at the bottom-right corner, which forced those two grids to be paired together.

Invalid Circling Since every grid should be paired, if circling two grids makes another grid out of option, the circling is invalid. Take Figure **??** for example - If $(2, 2)$ in the third column is circled, the top-right grid can only be paired with another 2 below, which is invalid because already exists a $(2, 2)$ pair. Thus, $(2, 2)$ in the third column can't be circled.

Possible Accelerations

Move Ordering Naive brute-force algorithm picks the next grid to circle in row major order. However, it will be more efficient to pick a grid with minimum branching factor first. This could reduce the chance of traversing wrong paths. In this problem, forced locations should be circled first, since its branching factor is always one. Forced locations will always be part of the solution.

Eliminate Used Grids and Pairs After circling two grids and forming a pair, any pair associated with the two grids and the corresponding numbers should be eliminated. Consider Figure **??**, circling the top-right pair $(0, 2)$ (marked with black border) makes the other five pairs invalid (marked with red borders).

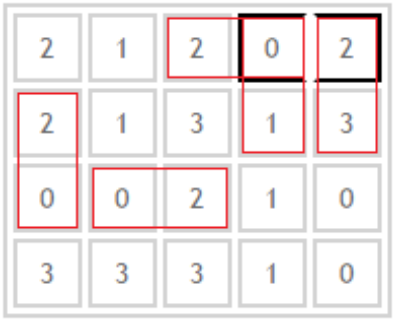

Figure 5

Set counters to maintain the remaining count of valid options for each unused grids and pairs. They can be useful for deciding the move ordering.

Detect Invalid Moves As discussed in the prior section, invalid circling can never lead to a solution. If those invalid moves can be detected earlier, states that are not likely leading to the goals will not be explored further. A way to do so is to check whether every unused grid and unused pair has at least one option after a move.

3 Problem Transformation

Exact Cover Problem

This problem can be transformed into another well-known problem called "Exact Cover Problem":

> Given a set X and a collection of subsets $S = \{S_1, S_2, \cdots S_n\}$, an exact cover is a set $S^* \in S, s.t.$ each element in X is contained in exactly one subset in S^*.

The set X in Dominosa is a set containing every grid and every possible pair of numbers. S is a set containing the choices of circling. Each subset in S can be represented by two grids $(r1, c1), (r2, c2)$ and their corresponding numbers $(v1, v2)$.

The goal of this problem is to find a subset in S (i.e. find a way of circling) s.t. every grid and every pair is used exactly once.

Knuth's Algorithm X

Donald Knuth presented an algorithm named Algorithm X to solve the exact cover problem. The problem is represented using a matrix A consisting of 0s and 1s. Each column represents an element in X. Each row represents a subset in S. The goal is to select a subset of rows so that the digit 1 appears in each column exactly once.

This is a recursive algorithm. Main concepts are stated below:

1. If A contains no columns, terminate successfully.
2. Otherwise, choose a column c.
3. For each row r that $A_{r,c} = 1$
   (a) Delete each column j and each row i in A, where $A_{r,j} = 1$ and $A_{i,j} = 1$.
   (b) Solve matrix A recursively.
   (c) If a solution is found, add r to the solution and return.

Dancing Links

In order to solve the exact cover problem, Knuth also presented a data structure "Dancing links". Using a naive implementation to store matrix A will spend lots of time searching for $1s$. Therefore, Knuth implemented a sparse matrix where only $1s$ are stored. Each node in the matrix represents an 1, and each node points to its four adjacent nodes. Consider Figure **??**, instead of storing the exact matrix on the left, use 2D doubly linked list on the right to represent the matrix.

$$\begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}$$

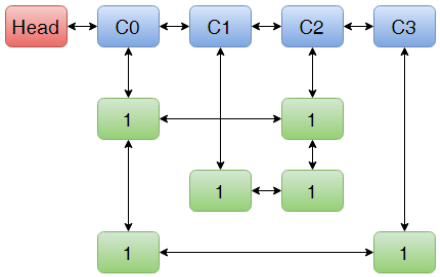

Figure 6

This method can improve the search time complexity from $O(n)$ to $O(1)$.

4 Implementation Details

I implemented Algorithm X and dancing links in my homework. Given Figure **??** as initial board, $(0, 0)$ is the top-left grid, some implementation details are discussed below.

1	1	0	1
2	0	2	2
0	0	2	1

Figure 7

Data Structure

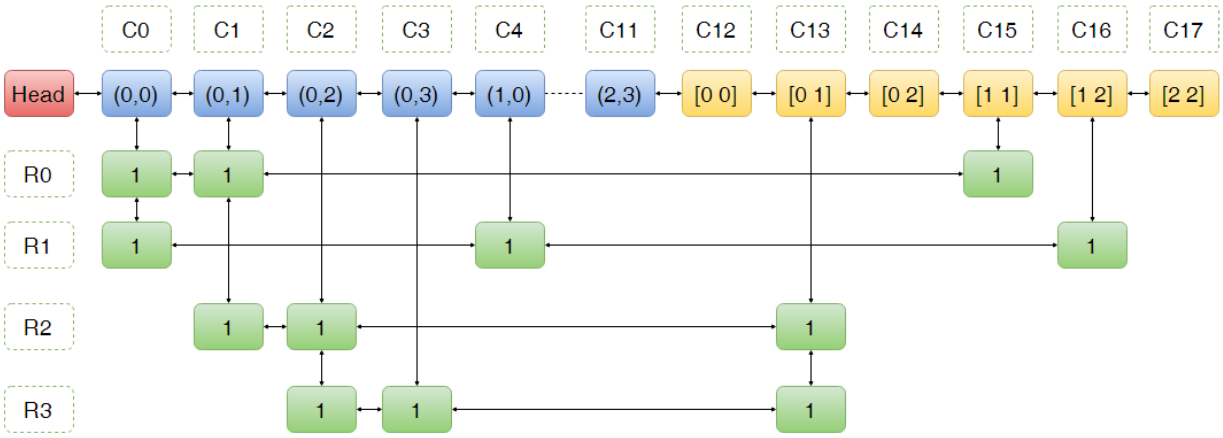

Figure 8

Figure **??** shows a part of the dancing links corresponding to the initial board. Blue columns (C_0 – C_{11}) represent the grids. Yellow columns (C_{12} – C_{17}) represent the pairs.

Each row represents an option of circling, so each of them contains exactly three 1s. For example, $R0$ means circling grids $(0, 0)$ and $(0, 1)$ can form a pair $(1, 1)$; $R1$ means circling grids $(0, 0)$ and $(1, 0)$ can form a pair $(1, 2)$. There are $2RC - R - C$ pairs can be circled on an $R \times C$ board. Row count of the initial dancing links is $2RC - R - C$.

The goal is to find a collection of rows, so that each column has exactly one 1.

Algorithm

Recursively solve the problem by DFS method. In each stage, search for a column c with minimum number of 1s. In this example, $C4$ is most likely to be chosen.

For the chosen column c, arbitrarily pick a row r which has an 1 in column c. $R1$ is the only candidate here. Note that if there are multiple choices, pick any one and continue DFS. If in further states, the algorithm failed to find a solution, it will return to this stage and try another row.

Eliminate Collisions

After choosing a row r, the rows containing collision 1s need to be removed. Choosing $R1$ to be part of the solution means that we formed a pair $(1, 2)$ by circling grids $(0, 0)$ and $(1, 0)$. Therefore, any other choices containing those two grids or the pair should be removed. There is only one row $R0$ needs to be removed in this example. Since column c is already covered by the choice, it will also be removed.

The remaining state is illustrated in Figure **??**.

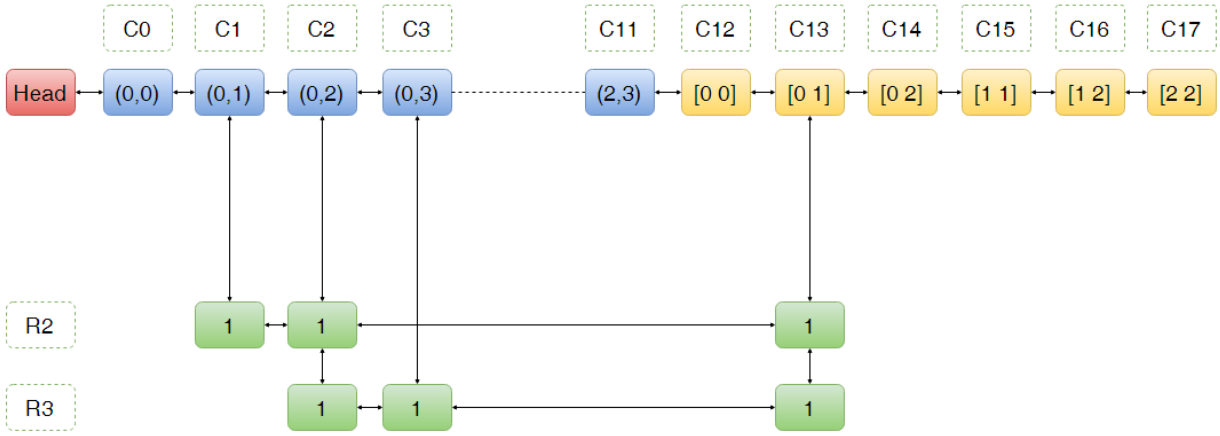

Figure 9

Detect Invalid moves

If removing a column and some rows after a choice leaving another column c' with no $1s$, this choice can never lead to a solution because c' will not be covered. In this case, algorithm should choose another option or return to the prior state (if there are no other options).

If the input is a valid board, the algorithm will terminate successfully and return the answer.

5 Experiment and Analysis

The websites I used to generate boards are listed below:

- `http://www.puzzle-dominosa.com/`
- `http://www.chiark.greenend.org.uk/~sgtatham/puzzles/js/dominosa.html`
- `http://coderazzi.net/javascript/dominosa/dominosa.html`

Branching Factor

The minimum branching factor at every state will not exceed four. This is because each grid can have at most four other grids to be paired with. Moreover, since the algorithm removes collision rows at every state, the branching factor decreases quickly.

Although some columns may have lots of $1s$ at initial state, the algorithm chooses the one with minimum options. I ran some experiments recording the minimum number of $1s$ in remaining columns at each state, and it turns out that the minimum number is never greater than two. In lots of cases, the minimum number is always one, which means DFS can always find the correct path.

I think it's because this game is originally designed for humans. Designers will often give players some good options to start with. Otherwise, the game will be too difficult to solve.

I randomly generated five 31×32 boards. There are $\frac{31 \cdot 32}{2} = 496$ pairs need to be circled, so the answer will be at 496^{th} layer. Table **??** shows the result of my experiment.

Board ID	# 1	# 2	# 3	# 4	# 5
# States traversed before finding a solution	4977	644	989	496	157315
# Minimum options >1	166	6	26	0	4796
Runtime (seconds)	0.14	0.02	0.03	0.04	3.2

Table 1

Since the minimum number of 1s is less than or equal to 2, DFS seldom needs to return and try another choice. In these five cases, program halts successfully in less than 5 seconds.

Board # 4 is the optimal case. It could find a state with only one option at every layer, so after traversing for 496^{th} layers, a solution is found.

Move Ordering

Another experiment I did is to pick an arbitrary column at each state instead of always choosing the one with minimum number of options. The result is shown in Table **??**

Board ID	#1	#2	#3	#4
N	5	10	15	30
# States traversed before finding a solution (Choose the optimal column)	21	129	136	4977
# States traversed before finding a solution (Choose an arbitrary column)	192	305058	-	-
Runtime (seconds) (Choose the optimal column)	0.01	0.01	0.01	0.14
Runtime (seconds) (Choose the optimal column)	0.01	0.2	>5min	>5min

Table 2

When $N \geq 15$, without move ordering optimization, a solution can not be found in five minutes. Table **??** shows the importance of choosing a good path to DFS.

6 Screenshot

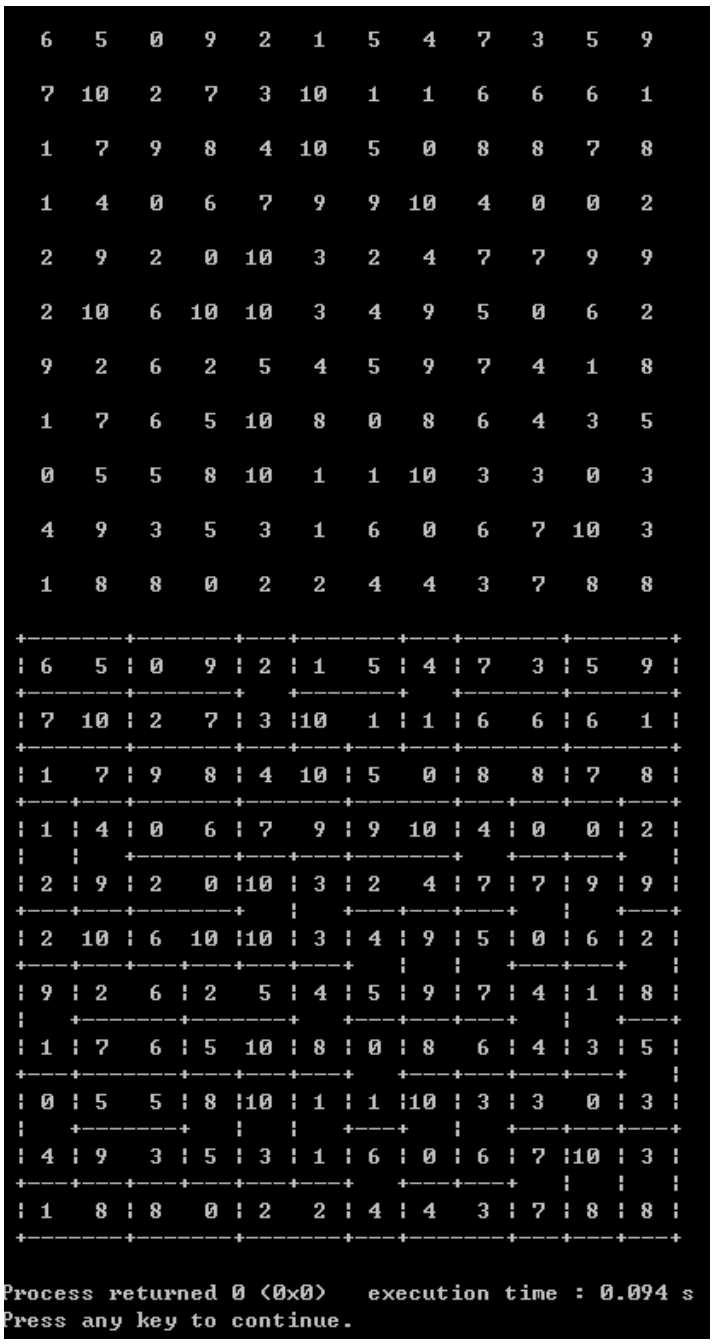

Figure 10: N=10

References

[1] http://www.mathematica-journal.com/2014/10/three-ways-to-solve-domino-grids/

[2] http://11011110.livejournal.com/128249.html

[3] http://mypaper.pchome.com.tw/zerojudge/post/1325718060

[4] http://www.cnblogs.com/grenet/p/3145800.html