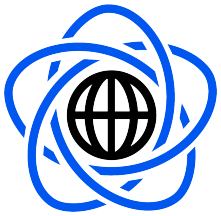
VAnishing DOmino pRoblem - VaDoR

Mateusz Bysiek, Radoslaw Lojek, Stanislaw Peryt; Computer Science, MiNI, WUT

17 October 2012



Contents

1 Problem description

Vanishing domino problem. Full definition is available in the document provided by laboratories supervisor. From now on, the problem will be referred to as VaDoR.

2 Estimation of complexity

2.1 Number of states

Let us assume that we have a set of n domino pieces that are arranged on a board. Then, let us assume that we are taking away pieces from the board, one at a time. If we consider each state after removing a piece, how many different states do we have? For a given state, for each piece, we have a choice: the piece either is on the board, or it is not. In total we have 2 possible scenarios for one piece, and n pieces, which gives us exactly 2^n different states.

2.2 Relations between the states

We can construct a following directed acyclic graph (DAG) G:

1. each vertex V represents a state (configuration of pieces on the board)
2. each directed edge E represents the relation between states. Edges are positioned in such way, that start point is at state V_1, and such that the end point of the edge is connected to the state V_2 that can be obtained directly from state V_1 by removing one domino piece from the board.

2.3 Estimation

Having a DAG G, the domino problem is equivalent to finding the longest path of DAG. Unfortunately, graph has to be constructed at runtime, and that takes time.

Since, for each state, the calculation of related states will be performed (and there are 2^n states in total) I estimate the complexity of algorithm as exponential. I also informally classify this problem as NP. After the DAG is constructed, by finding its longest path, an optimal solution is obtained.

My estimations indicate that calculation of related states for a given single state takes polynomial time. Finding the longest path in DAG takes linear time. Therefore, the total complexity of the accurate algorithm should be roughly $2^n * n^2 + n$

3 Description of the accurate algorithm

3.1 Input

XML file.

```
<domino_board width="5" height="4">
    <piece x="0" y="0" orientation="vertical" value1="2" value2="1" />
    <piece x="0" y="2" orientation="vertical" value1="3" value2="2" />
    <piece x="1" y="0" orientation="horizontal" value1="1" value2="0" />
    <piece x="1" y="1" orientation="vertical" value1="3" value2="0" />
    <piece x="1" y="3" orientation="horizontal" value1="0" value2="0" />
    <piece x="2" y="1" orientation="vertical" value1="1" value2="1" />
    <piece x="3" y="0" orientation="horizontal" value1="3" value2="3" />
    <piece x="3" y="1" orientation="vertical" value1="0" value2="2" />
    <piece x="3" y="3" orientation="horizontal" value1="2" value2="2" />
    <piece x="4" y="1" orientation="vertical" value1="4" value2="4" />
</domino_board>
```

Description of the `piece` tag:

- x - coordinate, from zero, increasing from the right to the left side of the board
- y - coordinate, from zero, increasing from the top to the bottom side of the board
- orientation:
  - *horizontal* - the piece starts at (x, y) and ends at $(x + 1, y)$
  - *vertical* - the piece starts at (x, y) and ends at $(x, y + 1)$
- value1 - value of the beginning of the piece i.e. value at (x, y)
- value2 - value of the end of the piece i.e. location depends on the orientation

If a tag `removed_pieces` is present inside the `domino_board` tag, it is ignored. If an attribute `order` is attached to the `piece` tag, it is also ignored. That way the output file can be used as an input, and pieces can be rearanged and put back on the board using a text editor, without worrying about problems with parsing.

3.2 Output

Another XML file.

```
<domino_board width="5" height="4">
    <piece x="0" y="0" orientation="vertical" value1="2" value2="1" />
    <piece x="0" y="2" orientation="vertical" value1="3" value2="2" />
    <piece x="3" y="0" orientation="horizontal" value1="3" value2="3" />
    <piece x="4" y="1" orientation="vertical" value1="4" value2="4" />
    <removed_pieces>
       <piece order="0" x="1" y="0" orientation="horizontal" value1="1" value2="0" />
       <piece order="1" x="1" y="1" orientation="vertical" value1="3" value2="0" />
       <piece order="2" x="2" y="1" orientation="vertical" value1="1" value2="1" />
       <piece order="3" x="3" y="1" orientation="vertical" value1="0" value2="2" />
       <piece order="4" x="1" y="3" orientation="horizontal" value1="0" value2="0" />
       <piece order="5" x="3" y="3" orientation="horizontal" value1="2" value2="2" />
    </removed_pieces>
</domino_board>
```

It is an extension of an input format. Program will be designed in such a way, that output from the program can be used again as an input (as it is described in “Input” section).

Description of extension the `piece` tag:

- order - number, from zero, increasing in order in which the pieces were removed from the board

The `removed_pieces` tag contains all of the pieces that were removed from the board, with complete information about their origin.

3.3 Class variables

Main class of the program is called domino_problem.

Rough list of fields of the class:

```
/* elements_t is a list of domino pieces */
   // collection of pieces
   elements_t elements;
   size_t width;
   size_t height;
   // collection of halves of pieces that come from 'elements' field
   board_t board;
   // pieces currently on the board
   elements_t on_board;
   // possible to remove in the next turn
   elements_t possible;
   // not longer on board, removed in the previous turns
   elements_t removed;
```

```
14    // algorithm does not know anything about these pieces
15    elements_t unresolved;
16    // possible to remove if other pieces are placed right
17    elements_t checked;
18    // impossible to remove due to size of the board
19    elements_t invalid;
```

There is also a supporting variable, graph, which stores data about every state analyzed by the algorithm.

```
1     graph_t graph
```

3.4 Algorithm

Because of large number of loops involved in the algorithm, I will not provide pseudocode, and will write down a description of instructions instead.

1. Input data is read: width and height.
2. All pieces present on board are stored into 'elements' field this list does not change with time
3. 'board' is generated for convienience.
4. 'elements' are copied into other lists: 'on_board' and 'unresolved'
5. 'unresolved' are partitioned into 'checked' and 'invalid'
6. add current problem to the graph
7. CHECKING: each piece from 'checked' is examined, and if it can be removed from the board it is copied to 'possible'
8. for each 'possible' piece, new object of class domino_problem is created via copy constructor
   (a) from each copy of current domino_problem, the corresponding piece of 'possible' is moved to 'removed', and the same piece is deleted from 'checked', 'on_board' and 'board'.
   (b) all lists of 'possible' from current domino_problem copies are cleared
   (c) add all copies to the graph with exception of those that already exist in the graph, because removing first piece A and then B gives the same resulting state as removing first piece B and then A. Calculating all possibilities with duplicates would result in factorial complexity of the algorithm ($n!$).
   (d) connect current problem to all of the copies
   (e) for each copy, perform all steps starting from CHECKING
9. after the graph is constructed, find the longest path, starting from initial domino_problem.

4 Description of the approximate algorithms

4.1 By Mateusz Bysiek

4.1.1 Input

The same as in case of accurate algorithm.

4.1.2 Class variables and algorithm

Development of approximations of NP problems is usually a consequence of deep understanding of the problem, and extensive experience in solving it using some kind of accurate algorithm. Without them, randomization can be used. My approximate algorithm is a polynomial time random algorithm.

It uses the same data structures as mentioned in the accurate algorithm, but it simply randomly chooses one possible option and tries to resolve it (in step 8. it does not have any loop). If a dead end is reached, the algorithm ends and returns so-far-calculated path.

4.2 by Stanislaw Peryt

4.2.1 Input Data

collection of domino pieces from input file

4.2.2 Description and Class Variables

This algorithm is based on the idea of Greedy algorithms. I assume some additional classification which is done only once for each element. For classification purposes each dominoPiece should contain some additional fields:

```
boolean isIndependent; //can be removed even from empty board, so it does not have to
   // be removed immediately
boolean canBeRemoved; //initially false. Changes to true, when fulfills removal conditions
list<waitingFor>; // list of pieces for which it is waiting;
list<waitForMe>; // list of pieces waiting for it;
```

I decided to add this classification, because in greedy strategy, we could remove pieces immediately when they fulfill removal requirements. But it may happen, that there are some pieces, for which pieces we already removed are necessary to fulfill removal requirements. When we have this classification based on scopes of domino pieces, we can wait with removing, and eliminate at least some of those situations described above, providing better local optimal choices.

For this algorithm we also need: board contains all domino pieces that have remained on the board roots contains possible starting pieces, i.e. containing zero neighbors contains domino pieces, which are neighbors of deleted piece, i.e. pieces alongside empty fields

4.2.3 Algorithm

1. Create board from input
2. Find possible roots and assign them to neighbors
3. for each dominoPiece in board assign scopes and waiting property
   (a) determine scopes based on numbers given on dominoPiece
   (b) determine dominoPieces on scopes and assign them to the waitForMe list
   (c) tell those pieces to assign this dominoPiece on their waitingFor list
4. stop=false;
5. while(!stop){
   - for each dominoPiece in neighbors{
     (a) if dominoPiece can be removed {
       i. if (domino Piece isIndependent) and (list of pieces it is waiting for is not empty) {
          wait with removal;
          }
       ii. else remove it(from neighbors and board) and tell waiting pieces, if there were any waiting for this piece, that it does not need waiting any more. (i.e. delete piece from waiting list)
     }
     }
     determine new neighbors into temp (each removed piece can produce at most 6 new neighbors)
   - if temp equals neighbors{
     – check if (something can be removed): remove first;
     – else stop=true;
     }
   - else neighbors=temp;

   }
6. calculate score

4.2.4 Estimation of complexity

Partial Complexities:
n-complexity of classification
n^2-complexity of part at while(!stop) loop

In order to associate attributes for classification we need to iterate once through whole collection which is of size n. Then in the while loop we are trying to remove pieces. In worst case we delete only one piece in one iteration which gives us at most n runs of while and inside while loop, we iterate through neighbors collection which is always smaller than n, but it is n dependent. Therefore for each n, algorithm executes number of operations proportional to n, which gives us quadratic time complexity.

Final worst case complexity : $n^2 + n$

4.3 by Radoslaw Lojek

4.3.1 Introduction

The solution I have designed is the most intuitive and straight forward algorithm. It checks all possible elements along its way. If a domino piece satisfies necessary conditions of being removed, it is thrown away from a board of elements. The operation is repeated for all consecutive elements until nothing can't be removed, or the domino board is empty.

4.3.2 Data structure

The data structure I have decided to use is a two dimensional array of objects, where object is a single domino field. Each object consists of fields such as: its value, location X and Y, boolean saying if a piece is placed horizontally/vertically, boolean saying if it is successor or predecessor field (value above or on the left side of a domino piece is successor, otherwise predecessor) and of course pointer to the other half of the piece. If a single cell in a matrix is empty, meaning that no piece is there, it is marked as a null pointer.

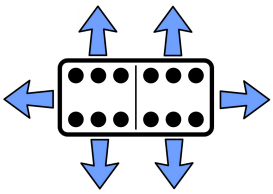
```
class dominoField {
   int value;
   int X;
   int Y;
   boolean isSuccessor;
   boolean isVertical;
   dominoField *twinField;
}
```

4.3.3 Algorithm Pseudo-Code

The algorithm takes as a parameters two dimensional array of elements and its vertical and horizontal size. Returns elements that were removed during the calculations. Below algorithm is simple, since it does not involve more complicated concepts such as Artificial Intelligence or complex graph algorithms.

```
List<dominoPiece> solveDominoProblem(dominoField **board, int dimX, int dimY){
   List<dominoPiece> result_list;
   int iterPiecesRemoved = 1;

   while iterPiecesRemoved != 0 {
      iterPiecesRemoved = 0;
      for i = 0 to dimX {
         for j = 0 to dimY {
            if !isEmpty(board,i,j) and isElemRemovable(board, dimX, dimY, i, j){
               result_list.add( removePiece(board,i,j) );
               iterPiecesRemoved = iterPiecesRemoved + 1;
            }
         }
      }
   }
   return result_list;
}
```



I will let myself omit pseudo code of functions used above. Just briefly explain how isElemRemovable will work: depending on the location of a 'twinField' of a single piece, function counts the number of empty fields in each possible direction (as depicted on the right). Once the directions are counted, function checks if they meet requirements for the piece to be removed - if it does, returns true or false otherwise.

4.3.4 Time complexity

Time complexity of the considered algorithm is n^2, since we go in each iteration through all possible domino fields n times. Each iteration is repeated in worst case *1/2*n* times, because may occur situation in which only one domino element (two fields) disappears in one *while* loop iteration.

4.3.5 Example