VAnishing DOmino pRoblem - VaDoR

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

17 October 2012

1 Problem description

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

```xml
<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>
```

- 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

3.2 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;
   // algorithm does not know anything about these pieces
   elements_t unresolved;
   // possible to remove if other pieces are placed right
   elements_t checked;
   // impossible to remove due to size of the board
   elements_t invalid;
```

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

```
        graph_t graph
```

3.3 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 Input

The same as in case of accurate algorithm.

4.2 Class variables and algorithm

4.2.1 By Mateusz Bysiek

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.