Miłosz Staniszewski 220629

Artificial Intelligence and Knowledge Engineering Laboratory

1. Constraint Satisfaction Problem
2. **Introduction**

Constraint Satisfaction Problem is defined as an object, which consists of variables, belonging in a domain, that have to satisfy defined constraints. In order to find a solution to those problems (such as logic puzzles, crosswords, etc.) many different algorithms are defined.

One of those algorithms is backtracking, where all possible values are assigned to a chosen variable. After that, each value is tested for inconsistency with defined constraints and if test is successful, the same happens with the next variable, and so on.

In order to minimize the search time, backtracking can be enhanced with some extra variants. One of those variants is forward-checking, where after assigning a value to a variable, one has to temporarily block or delete values from the domains of all unassigned variables, that don’t satisfy the constraints with the newly assigned one.

1. **Problem Analysis – Letter Mosaic**

The first task is to create a square board and fill it with two letters, so that there are no same three letters either in column or in row. Additionally, all rows and columns must be unique.

The variables for this task can be defined as a couple of coordinates for each cell of the board, that will contain exactly one letter.

The domain of values for each cell contains both letters used to fill the board.

There are four constraints that define, which value can be assigned to a variable. Two for allowing maximum two of the same letters in a consecutive cells either in row or in column, and two rules for having a unique row and column.

(for i not equal to j and n equal to the size of the board)

1. **Results**

To test my implementation, I used boards of different sizes, from 1 to 17 (bigger boards were often “freezing” during calculations). With bigger sizes, the harder it is for the algorithm to find a correct solution, so it takes more time. Due to the implemented solution, only the first possible result is displayed and there always is one, since there is no other constraints, that could make it impossible to find a solution. Here are average execution times for the boards of different size (without forward checking):

1 - <1ms

3 - <1ms

6 - <1ms

8 - <1ms

9 – 4ms

10 – 5ms

12 – 33ms

15 – 37s

17 – 28s

With forward checking:

1 - <1ms

3 - <1ms

6 - <1ms

8 - 1ms

10 – 53ms

12 – 149ms

(bigger boards “froze” during calculations)

1. **Problem Analysis – Hitori**

The second task is to implement a code that will try to solve a hitori puzzle. The puzzle is a square board filled with numbers, not greater than the size of the row/column. The goal is to cross out some of the numbers, so that no two the same numbers can be found in a column and a row, no two crossed-out numbers can be adjacent to each other and all of the uncrossed numbers should form a continuous field, so that every uncrossed number can be approached from another uncrossed number from either of the sides, but not from the diagonals.

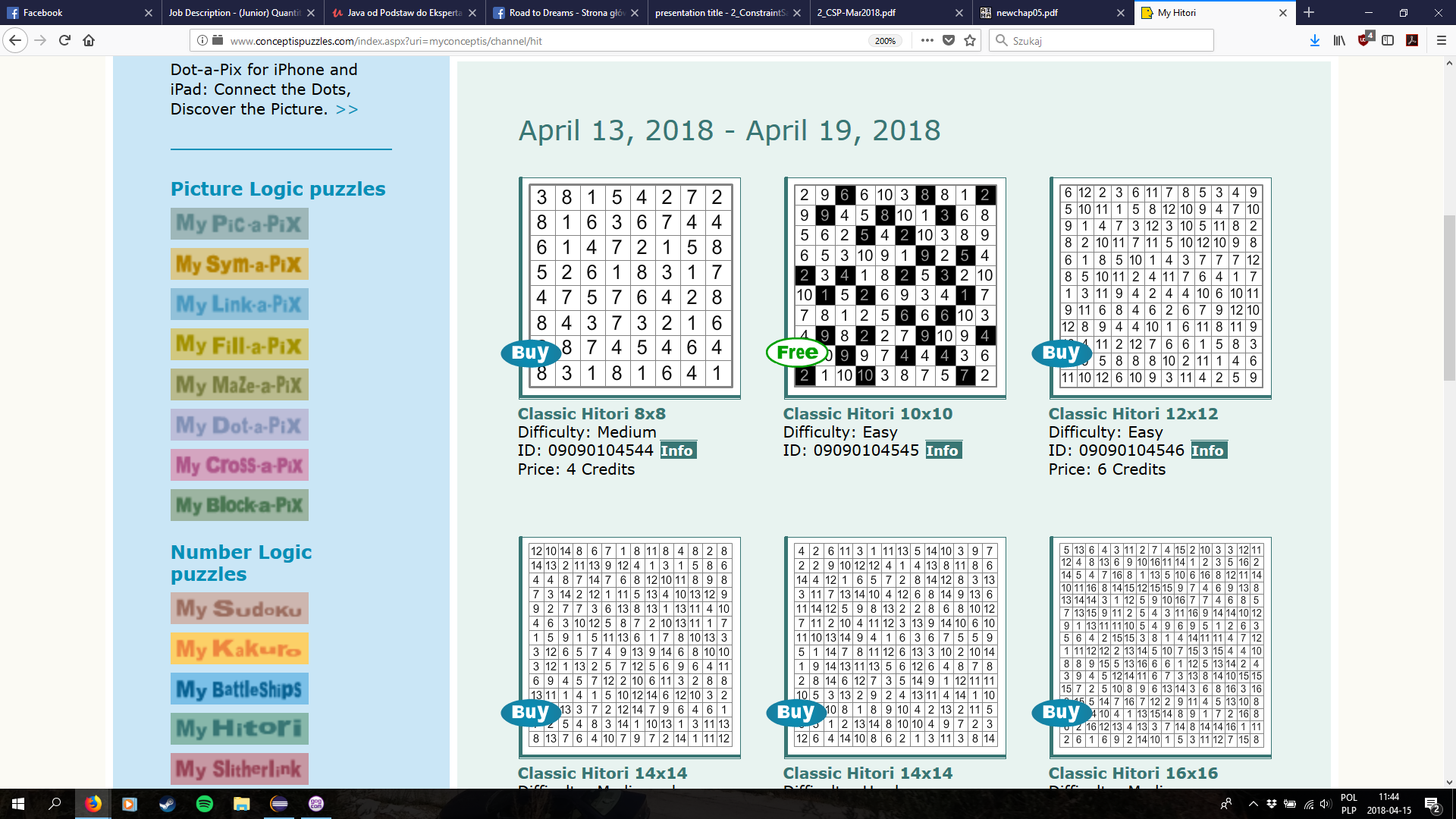
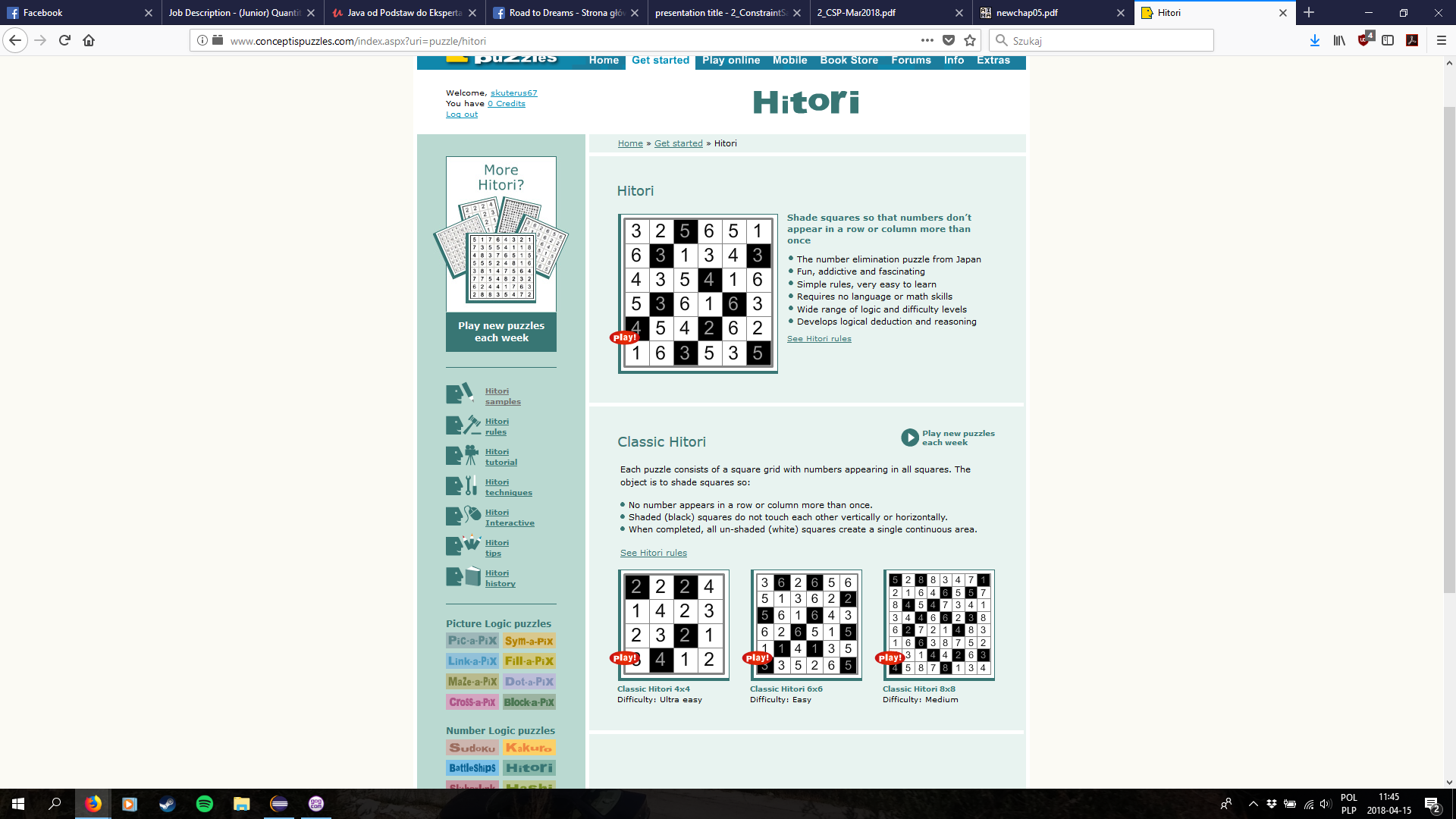
The variables for this problem are cells, each one containing place for one number.

Domain of values for the variables is a value that should tell us whether a number in a cell is crossed-out or not.

There are four constraints: two checking whether a number is occurring more than once in a row, and a column, one for checking, if for every crossed number, every adjacent number is uncrossed, last one for checking, if all uncrossed numbers are connected to each other.

1. **Results**

To test the results of my implementation, I used free puzzles that are available at the conceptispuzzles.com website. The puzzles were different sizes and required using different techniques to solve, which resulted in better bug-fixing the code.



Pure backtracking:

Hitori size 4 – 2ms

Hitori size 6 – 14ms

Hitori size 6 – 33ms

Hitori size 8 – 143ms

Hitori size 8 – 4,5s

Hitori size 10 – 5,4s

Backtracking with forward checking:

Hitori size 4 – 2ms

Hitori size 6 – 15ms

Hitori size 6 – 29ms

Hitori size 8 – 144ms

Hitori size 8 – 4,5s

Hitori size 10 – 5,4s

1. **Evaluation**

Applied forward-checking technique does not distinguish differences between proposed progress in assigning variables and thus checks all remaining constraints every iteration of recurrence. This does not decrease calculation time and sometimes even makes it longer or exceeds computer’s calculating capabilities. This can be mostly seen in letter mosaic problem and not in hitori due to the fact, that the constraints checking hitori correctness are checking all cells even without forward checking (recursive checking of a constraint, where all uncrossed numbers must be adjacent to each other), so the execution time is not very different.

1. **Credits**

* H. Kwaśnicka – Introduction to Artificial Intelligence. Constraint Satisfaction Problems, Wrocław 2018
* S. Russell, P. Norvig – Artificial Intelligence: A Modern Approach, Berkeley, 1994
* F. Bacchus, A. Grove – Principles and Practise of Constraint Programming. On The Forward Checking Algorithm, New York, 1995
* H. Ghaderi – Backtracking Search (CSPs), Torinto, 2017
* Conceptis Puzzles – Hitori, <http://www.conceptispuzzles.com/index.aspx?uri=myconceptis/channel/hit>, <http://www.conceptispuzzles.com/index.aspx?uri=puzzle/hitori>