Analysis of different AI Strategies for Solving Picross Puzzles

CS7IS2 Project (2020/2021)

Andrej Liadov, Edvinas Teiserskis, Min Wu, Edmond Cheng, Adam McQuade

liadova@tcd.ie, teiserse@tcd.ie, wumi@tcd.ie, chenge@tcd.ie, amcquade@tcd.ie

Abstract. Research in Artificial Intelligence has always had a very strong relationship with games and game-playing. Picross (a.k.a nonograms) are logic puzzles with simple rules and challenging solutions, and in this case our research provides a survey that analyses and compares different AI algorithms - uniform cost search, Q-learning, constraint satisfaction problems (CSPs) and genetic algorithms (GA) applied to solve picross puzzles. Through the process of implementation and evaluation we analyse there algorithms and find out their common aspects, differences, connections between methods, drawbacks and open problems.

Keywords: artificial intelligence, uniform cost search, Q-learning, CSPs, Genetic Algorithms

1 Introduction

Artificial intelligence (AI) is the emulation of human intelligence of computers that are designed to think and behave like humans. As AI has evolved to have a huge global influence, it operates by integrating vast volumes of data with quick, iterative analysis and intelligent algorithms, which enables the program to learn automatically from patterns or features in the data.

Picross, also known as Nonograms, is a wonderful series of games in which players use logic to solve puzzles that the cells in a grid must be marked or just left clear constrained by the numbers at the side of the grid(as Figure 1 shows). The least time to solve the puzzles is the key to win. However, playing games like a human is other than thinking about games like a human or learning like one. There is a famous saying that game-playing is the Drosophila of AI. And yet in our research it could be that the task of playing picross, once it's converted into the task of searching some certain nodes evaluated by time consumed, it is also a different kind of intelligence.

Our motivation is no more than analyzing the problem of picross and deduce what the best strategy among the 4 mentioned AI algorithms, which can be used to picross design feedback, content generation, or difficulty estimation. One potential use is in order for humans to use these computer-generated strategies, the strategies must be both efficient in the domain of interest and concisely

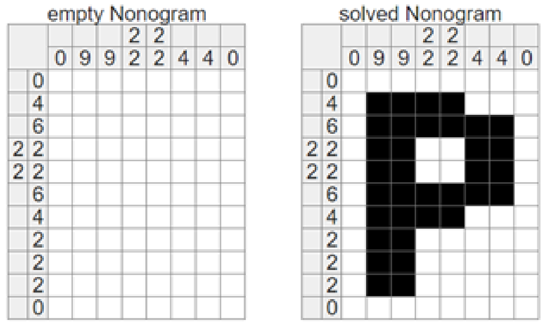

Fig. 1. An example of a picross puzzle, with the start state on the left and completed puzzle on the right. The numbered hints describe how many contiguous blocks of cells are filled with true. We mark black cells are true and cells left blank are false.

articulated so that a designer can consider the whole strategy in their head. For example, there are dozens of documented strategies for picross[13], and puzzle designers construct puzzles and rank their difficulty based on which of these strategies are used[14].

This paper presents a comparison of 4 different AI algorithms including uniform cost search, Q-learning, CSP and GA that have been used to solve the picross puzzles. The paper also evaluates these algorithms based on their execution time, memoty usage and so on. And through the process of implementation and evaluation we prove the state-of-the-art of all these algorithms and we also find out their common aspects and differences, connections between methods, drawbacks and open problems. Our recording of presentation link of this research is xxx.

2 Related Work

Picross, like other logic puzzles such as Sudoku, has special answers which can be solved by deducing pieces of the answer in any order. In Sudoku, the squares are filled with one number each while in Picross each square is either filled in or left blank. Prior work has also examined the solutions of logic puzzles, but most existing solvers are programs written from scratch for the express purpose of solving paint-by-number puzzles[3–6]. Browne provides a deductive search algorithm that is intended to mimic the constraints and method of human solvers[1]. The complexity of picross can be estimated by calculating the number of steps that can be solved one row/column at a time according to Batenburg and Kosters[2].

Heuristic search is a technique that uses a heuristic value for optimizing the search. Many heuristic solving steps are given in order to determine the value of some pixels in a single row or column[8], and in order to decide which pixels can be assigned a certain value that they use run ranges[9]. The methodology behind is to assign an integer range to each row or column description, the lowest and largest pixel number can be used to contain the run's black pixels corresponding to the definition of the integer. An example of applying heuristic steps to fill the pixels is Teal's Nonogram Solver[7], the input for this solver is given by a single string containing the row and column definitions one by one. In other words, this solver runs through all rows and columns one by one, while applying heuristic steps in order to fill pixels. However, the solver leaves several pixels undetermined, which can clearly be filled by logic decisions. These are pixels that need input from the last pixels in a run, usually at the end of a row or column. Based on this, we will choose GA to solve the picross puzzles because as GA is a class of heuristic optimization methods, it mimics the process of natural evolution by modifying a population of individual solutions which makes it easier to achieve our goal. A-star is another heuristic algorithm that is commonly used in pathfinding and graph traversal, which is the method of mapping an easily traversable path between multiple nodes[16].

The Backtracking Heuristic (BH) methodology consists in to construct blocks of items by combination between heuristic, that solve mathematical programming models, and backtrack search algorithm to figure out the best heuristics and their best ordering[17].

Backtracking Search for CSP: Some hobbyists have created programs that solve Sudoku puzzles using a backtracking algorithm, which is a form of brute force search[12]. Backtracking is a depth-first search (as comparison to a breadth-first search) so it can completely investigate one branch to a potential solution before going on to another. A brute force algorithm visits the empty cells in some order, filling in digits sequentially, or backtracking when the number is found to be not valid[10, 11].

Reinforcement Learning for CSP: According to [15], the constraints could be presented as an image, and hence Mehta chose the algorithm used for Sudoku to be Deep Q-Learning. The Q agent is trained with no rules of the game, with only the reward corresponding to each state's action. This paper[15] contributes to choosing the reward structure, state representation, and formulation of the deep neural network.

3 Problem Definition and Algorithm

This section formalises the problem you are addressing and the models used to solve it. This section should provide a technical discussion of the chosen/implemented algorithms. A pseudocode description of the algorithm(s) can also be beneficial to a clear explanation. It is also possible to provide one example that clarifies the way an algorithm works. It is important to highlight in this section the possible parameters involved in the model and their impact, as well as all the implementation choices that can impact the algorithm.

3.1 Uniform Cost Search

We have implemented an uniform cost search solution to solve picross puzzles - while this is an established search algorithm, this requires our problem to be represented as a searchable graph.

We did this by modelling the puzzle as a graph - The start state is an empty puzzle, and the next states are potential moves of filling in a square as a particular colour, out of all unfilled squares (our only factor of filtering is if a row/column combination allows for a colour). The cost function we have implemented is looking at the row/column of the square in question, and calculating:

$$(1 - row_{proportion}) * (1 - column_{propotion})$$

where $row/column_{proportion}$ is the number of squares of that colour in that row/column divided by the height/width of the puzzle. As such, in the intersection of a full row/column pair, the action cost is zero, of a nearly full pair close to zero and for a sparse pair approaching one. This gives us a cost function which makes cheaper locations that will be more likely to contain a square of that colour.

While the original plan was to add onto this a heuristic in order to implement a-star search, we could not come up with a sensible way to derive a heuristic from the state of the puzzle - either we would produce a calculation that would most likely not correspond to a useful way to estimate the "completeness" of a puzzle, or we would effectively be encoding another method like constraint programming within the heuristic itself (making the surrounding search implementation redundant).

Due to the nature of the puzzle, using search is expected to be the worst performing method, and is mainly created as an example. Since the puzzle can have an inordinate number of states, and there is a massive expansion of state transitions at each state, using a search approach immediately hits major speed and memory limits.

For example, taking a monochrome 5x5 puzzle, we have 2^{25} total possible states, and most states have a numerous amount of state transitions - 25 from the initial state, 24 from *each* state following those, and so on. As such, the frontier of the search grows at a very rapid pace - this both makes each state take additional computing time, and quickly fills up the physical limits of available memory on most computers.

Some reduction occurs from the fact that the cost and result of two paths leading to the same state are regardless of order, allowing us to collapse in duplicate entries. However, this does not nearly reduce the load of the approach to a state where is it comparable to other methods that actively check against the puzzle's rules and full current state in a sensible manner.

3.2 Q-Learning

q-learning text goes here

3.3 Constraint Programming

The basis of the constraint programming algorithm applied was backtracking, with constraint propagation. Depth first search was used as the backend to apply backtracking. The algorithm starts off with an empty puzzle. All permutations of the first row are generated and filtered. The filtering process here refers to constraint propagation. Constraint propagation is applied as follows:

1. Generate all permutation of row
2. For each row, check if the column constraint is broken. A column constraint is broken when that column does not contain that colour.
3. Return list of all feasible rows

The returned list of rows is used to set the states of the backtracking algorithm. Every time a row is set into the nonogram puzzle, the state of the nonogram is checked. Every column in the puzzle is checked. If the number of occurrences of any given colour is greater than the what the constraints dictate, then that row is removed. Figure 2 below displays this process. This checking is done every time a new row is added.

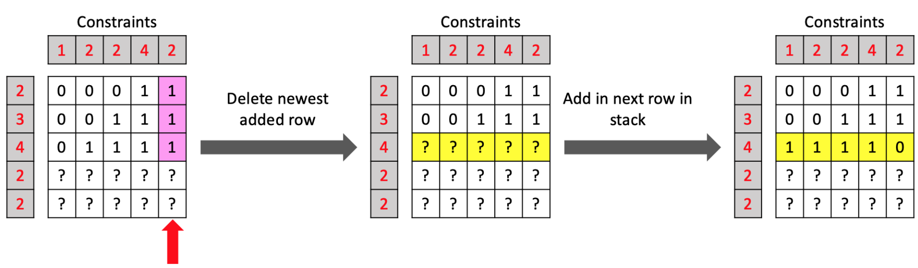

Fig. 2. Backtracking

The overall algorithm is shown below.

3.4 Genetic Algorithms

genetic algorithm text goes here

4 Experimental Results

This section should provide the details of the evaluation. Specifically:

– Methodology: describe the evaluation criteria, the data used during the evaluation, and the methodology followed to perform the evaluation.

Algorithm 1 CSP Algorithm

```
stack = LIFO_stack
all_rows = generate_all_rows[row_index]
viable_rows = filter_rows(all_rows)
while stack != empty do
    row = stack.pop
    puzzle.insert_row(row)
    check = puzzle.check_constraints
    if check == False then
        puzzle.delete_row(row) → continue
    while index < puzzle.height do
        stack.push(generate_all_rows(index)
```

– Results: present the results of the experimental evaluation. Graphical data and tables are two common ways to present the results. Also, a comparison with a baseline should be provided.
– Discussion: discuss the implication of the results of the proposed algorithms/models. What are the weakness/strengths of the method(s) compared with the other methods/baseline?

5 Conclusions

Provide a final discussion of the main results and conclusions of the report. Comment on the lesson learnt and possible improvements.

A standard and well formatted bibliography of papers cited in the report. For example:

References

1. Cameron Browne. 2013. Deductive search for logic puzzles. In Computational Intelligence in Games (CIG), 2013 IEEE Conference on. IEEE.
2. K Joost Batenburg and Walter A Kosters. 2012. On the difficulty of Nonograms. ICGA Journal 35, 4 (2012), 195–205.
3. Jan Wolter's pbnsolve Program https://webpbn.com/pbnsolve.html
4. Mirek and Petr Olšák's Nonogram Solver http://www.olsak.net/grid.html#English
5. Steve Simpson's Nonogram Solver http://www.comp.lancs.ac.uk/~ss/software/nonowimp/
6. Jakub Wilk's Nonogram Solver http://jwilk.nfshost.com/software/nonogram.html
7. http://a.teall.info/nonogram
8. S. Salcedo-Sanz, E.G. Ort'iz-Garc'ia et al., Solving Japanese Puzzles with Heuristics, IEEE Symposium on Computational Intelligence and Games, 2007, 224-231, CIG, 2007.
9. C.H. Yu, H.L. Lee, L.H. Chen, An efficient algorithm for solving nonograms, Springer Science+Business Media, Springer Verlag, Applied Intelligence 35(1): 18-31, 2011
10. Norvig, Peter. "Solving Every Sudoku Puzzle". Peter Norvig (personal website). Retrieved 24 December 2016. http://www.norvig.com/sudoku.html

11. Zelenski, Julie (July 16, 2008). Lecture 11 — Programming Abstractions (Stanford). Stanford Computer Science Department. https://www.youtube.com/watch?v=p-gpaIGRCQI
12. Brute Force Search, December 14th, 2009. http://intelligence.worldofcomputing/brute-force-search
13. Andrew C Stuart. 2007. The Logic of Sudoku. Michael Mepham Publishing.
14. Andrew C Stuart. 2012. Sudoku Creation and Grading. (January 2012). http://www.sudokuwiki.org/SudokuCreationandGrading.pdf [Online, accessed 8 Mar 2017].
15. Mehta, Anav. "Reinforcement Learning For Constraint Satisfaction Game Agents (15-Puzzle, Minesweeper, 2048, and Sudoku)." arXiv preprint arXiv:2102.06019 (2021).
16. Solving 8-Puzzle using A* Algorithm https://blog.goodaudience.com/solving-8-puzzle-using-a-algorithm-7b509c331288
17. Jonatã L., Araújo P., Pinheiro P.R. (2011) Applying Backtracking Heuristics for Constrained Two-Dimensional Guillotine Cutting Problems. In: Liu B., Chai C. (eds) Information Computing and Applications. ICICA 2011. Lecture Notes in Computer Science, vol 7030. Springer, Berlin, Heidelberg.