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), in our scope, is the application of computing to solve non-trivial problems and analyse complex, real-world, situations. AI has been extensively researched and applied to many situations at this point in time. However, the basis of testing many AI approaches remains in the application of AI to well defined games, such as chess, and logic puzzles such as Sudoku or, in our case, Picross.

Picross, also known as Nonograms, is a logic puzzle in which the player uses provided rules to mark (or omit from marking) cells in a grid. The rules are given as a ordered list of numbers, each referring to a contiguous sequence of filled cells in the grid. The puzzles come in two formats, monochrome and multicoloured. Cell sequences must be separated, either by a space in same-colour sequences, or a break by a sequence of a differing colour. Typically, the rules are given as lists of numbers at the sides of the grid (see Figure 1). As these puzzles (typically) have only one solution, the best method to solve them is determined by the speed of the solution. We decided on looking at picross puzzles due to their relative novelty, and we were also attracted by the goal of solving a puzzle being to produce an image - a quality that Sudoku cannot compete with.

Our goal is to analyse the solving of picross puzzles, and determine which is the best among the 4 approaches we are examining. These types of results can be

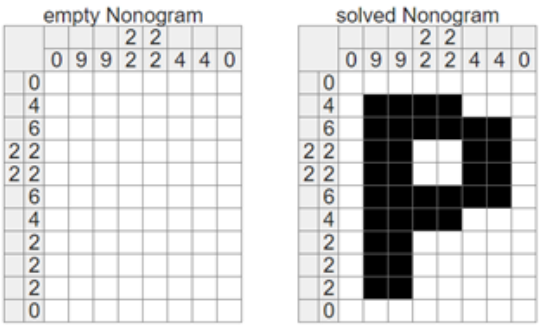

Fig. 1. An example of a picross puzzle, with the start state on the left and completed puzzle on the right. The numbered hints describe how many contiguous blocks of cells are filled. Black cells mark filled cells and blank cells mark unfilled cells. Example from `https://en.wikipedia.org/wiki/Nonogram`, sourced 2021-04-28.

used for picross design feedback, content generation, or difficulty estimation. One potential use is to act as an automated difficulty grading system to be used by picross puzzle designers. Depending on the approach and the time taken to solve the puzzle by a given algorithm, a puzzle curation system can sort puzzles in terms of their difficulty. 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 - uniform cost search, Q-learning, CSP and GA - that we have decided to apply to solving picross puzzles. The paper also evaluates these algorithms based on their execution time, memory usage and other metrics. Through the process of implementation and evaluation we show the pros and cons of each algorithm, and we also examine 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[18].

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.

Genetic algorithms have found much use in puzzle-based problem solving. According to [17], genetic algorithms are well suited for multi-objective optimisation problems, such as sudoku puzzles. Search algorithms are a commonly deployed in games to find an optimal solution to a problem with many path options. A problem encountered by search algorithms is the high time and space complexities required when the search domain becomes exceedingly large. Genetic algorithms are a way of mitigating this downside of search algorithms with respect to solution searching, and have been

demonstrated as effective in solving other constraint based puzzles such as sudoku.

A genetic algorithm is a stochastic optimization algorithm. It takes inspiration from biological ideas such as the theory of evolution put forth by Charles Darwin. In this theory, genetic characteristics are passed down through successive generations following successful reproduction. Successful production of offspring is dependent on many factors, including the suitability of the parents to the local environment. When applied to optimisation algorithms, parents may be thought of as an instance of an agent attempting to reach a goal. A generation is a population of these instances with slight differences between their characteristic parameters. Following each iteration of the agents attempting to reach a goal state, the agents with the best progress are given a higher probability of producing offspring. The next generation is produced as a function of this probability. Thus, each cohort of agents is a slight modification of the previous population, weighted towards the most successful members of the previous generation. Over many iterations, the population should become optimized to the environment, thus having more success in reaching the goal state.

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. We would either 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

First the state space and the actions space had to be defined for the problem in order to create a Q-learning agent.

The state of a given puzzle was defined as the the rows that were filled in by the Q-leaning agent. If there were different permutation chosen for a given row of a given puzzle, the state of the puzzle would be different.

The action space for a given puzzle would be defined by getting the permutations of the row that are left to be filled in. Choosing a row to fill would be considered an action that would consequently move the agent to a different state. If an action is chosen for a row it is not possible to reverse that action. The row would be filled if an action was taken and as a result, there would not be any actions available for that row.

The state space contains two methods of exiting the puzzle. The first method is to fill in all rows the to not have any more actions to take. If there are no more actions to take, the algorithm checks whether the solution to the puzzle is correct. If the answer to the puzzle is correct, the reward returned is 1000. If the completed puzzle is incorrect, the reward returned is -100.

Q-learning with a q-table is meant to choose actions in such a way that the reward is maximised. The state space contains very few solutions. In most of the puzzles tested there was a unique correct solution, in some cases there might have been several solutions. That means that the correct solution space is extremely sparse while the incorrect solution space is quite vast. The contrast between correct solution space and incorrect solution space was reflected in the calculation of reward. Taking an action that did not complete the puzzle should always result in a reward of +1. If the an action taken results in a correct solution, a reward returned was +1000 and if the solution was incorrect, a reward of -100 was returned.

The approach to exploration was a soft-max approach. The exploration rate would be high very high to begin with because nothing is known about the reward, action and state space. A high encourage the agent to choose random actions. A random number is generated between [0-1] and if epsilon is larger than the the random number, the agent picks a random action. If not, it picks the action with the highest q-value. Epsilon has a pre-defined decay rate. By the time the episode number reaches 100, the agent should be a hard-max solver, meaning the agent only picks the best q-value.

The number of episodes will determine how many times the problem will be solved and as a result the q-table will be updated in a proportional amount since the q-table will be updated after every action. The pseudo-code for an episode is given below.

Algorithm 1 Q-Learning Algorithm

```
for episode in num_episodes do
    agent.puzzle = read_puzzle()
    for stepinmax_steps do
        if agent.is_explore() then
            action = agent.get_random_action()
        else
            action = agent.get_action()
        new_state, reward, complete = agent.step(action)
        agent.update_qtable(action, new_state, reward)
        agent.puzzle = new_state.puzzle
        if complete then
            break
```

Updating the q-table is done the Bellman equation. This process is done by combined the old q-value with a learning rate, the reward, the discount rate and the max q-value in the future state. The equation is given below. Alpha is the learning rate and it determines how much influence the new q-value will have on the old one. The higher the learning rate, the more influence the future q-value will have.

$$NewQ(s,a) = (1-\alpha)Q(s,a) + \alpha[R(s,a) + \gamma maxQ'(s',a') - Q(s,a)] \quad (1)$$

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.

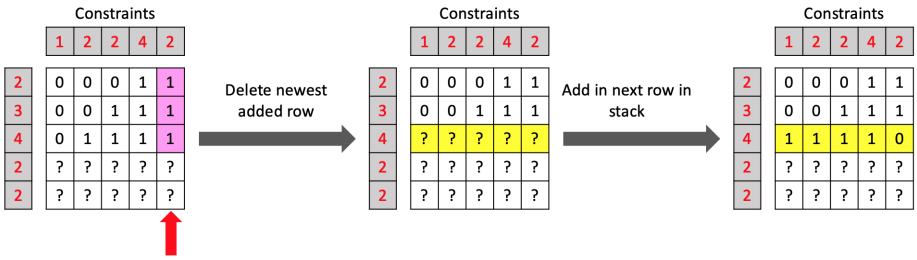

Fig. 2. Backtracking

The overall algorithm is shown below.

3.4 Genetic Algorithms

The approach taken in this project follows the typical genetic algorithm design implementation. The programme reads the rules file which determines the constraints for each nonogram puzzle.

Algorithm 2 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)
```

1. Selection: the parents for the next generation are selected based on a metric known as fitness. In this implementation, fitness is a measure of how many of the constraints are left unsatisfied by a parent solution. The agents are sorted by those with the lowest number of unsatisfied constraints and these are more likely to be parents.
2. Crossover: is the process of combining the characteristics of two parents to form a child. Using a probability function, a random split of each of two parents characteristics is taken and recombined into a new characteristic. This is undertaken for each pair of parents at the end of an iteration to create the new generation.
3. Mutation: this is the process of randomly altering minor details in the characteristics of the new generation of agents. In this case it implies that a random block within the nonogram will be flipped from filled to unfilled.

The programme terminates in either one of two ways. At the end of each iteration of agents, the solution is checked against that reached by the algorithm and terminates if the solution is reached. However, the programme is still liable to make no progression beyond local optima. To mitigate this, the algorithm is terminated and reinitialised with random values after 1000 iterations.

The overall algorithm is shown below.

Algorithm 3 Pseudo-code for Genetic Algorithm

```
population ← GenerateInitialPopulation(desiredPopulationSize)
population ← assignFitnessToEachMember(population)
while run − time < allowedTime do
    newGeneration ← selectMembersOfPopForCrossover(population)
    newGeneration ← mutatePopulation(newGeneration)
    population ← assignFitnessToEachMember(population)
    if highestFitnessInPopulation(population) >= fitnessRequired then
        return fittestMemeberOfPopulation(population)
return fittestMemeberOfPopulation(population)
```

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.
- 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. Using genetic algorithms to solve sudoku puzzles https://ieeexplore.ieee.org/document/4424632
18. 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.