DePosit, Mass, Tolleson
CS541 Artificial Intelligence, Prof. Rhodes
Winter 2020 Term Project Report

Table of Contents

Overview

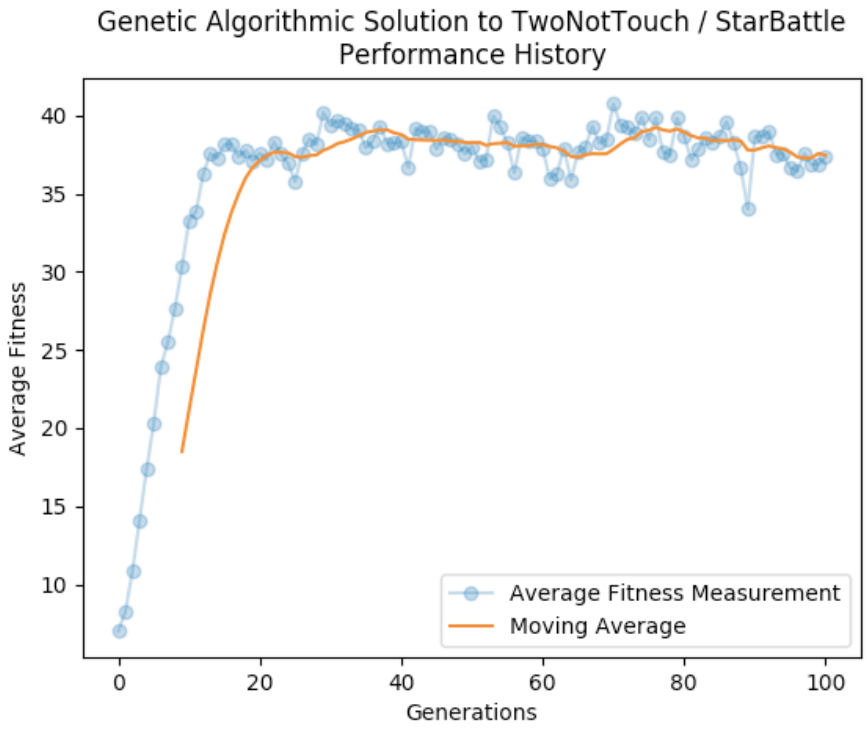
For our final coding project, we chose to create solvers for the puzzle TwoNotTouch (as so named in The New York Times), also more commonly known as StarBattle. We chose to primarily attempt to solve a puzzle of regular difficulty which is a 10x10 board with 10 regions onto which a user is to place 20 stars. Each row, column, and region need to contain exactly 2 stars, and no 2 stars may touch one another.

We attempted two methods of solving this problem; a genetic algorithm and constraints satisfaction. We adapted the genetic algorithmic solver from the 8-Queens programming assignment since the environments are structured similarly. A novel constraint satisfaction algorithm was coded. In this paper we will summarize the logic behind both approaches and compare their results.

Old & Busted – Genetic Algorithm

For our second programming assignment for this course, we were tasked with solving the 8-Queens problem using a genetic algorithm. While three of us coded very different implementations of the genetic algorithmic solver, the underlying logic for all three are based on the Genetic Algorithm described in our class lecture and the Russell Norvig textbook. [1] At a very high level, these are the steps the original genetic algorithm took:

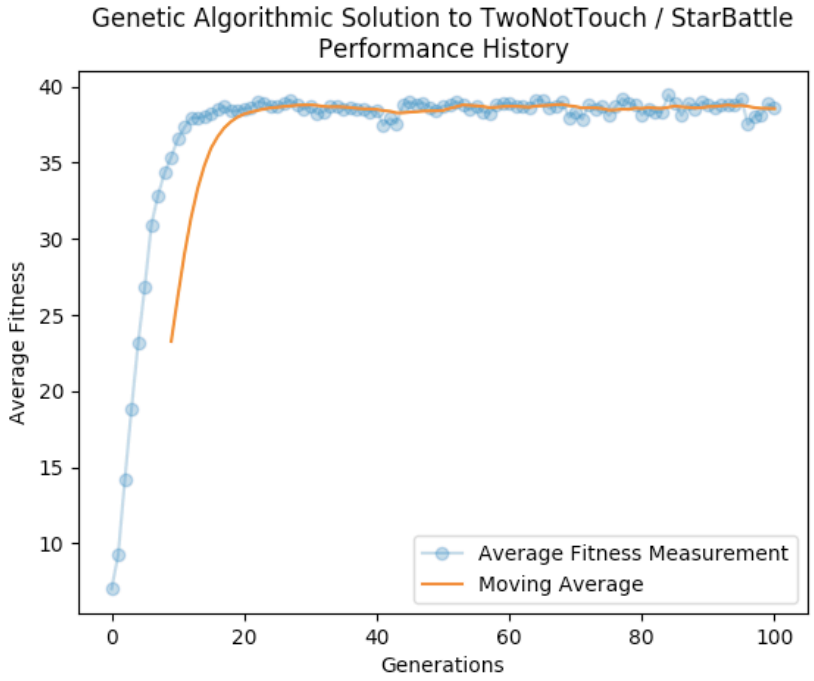
1) Create k random specimens for the first generation. Each specimen represents the location of 20 stars on a 10x10 board.
2) Calculate the fitness for each specimen. For this puzzle we calculated the maximum fitness to be 55, from which we subtract the number of pairs of adjoining stars and the number of regions containing less than 2 stars.
3) Create the next generation by simulated evolution:
    a. weighted random selection based on relative fitness for breeding pairs,
    b. crossover of star location segments between pairs, and
    c. random mutation of one specific star location.
4) Calculate fitness for each specimen in new generation.
5) Repeat for N generations.



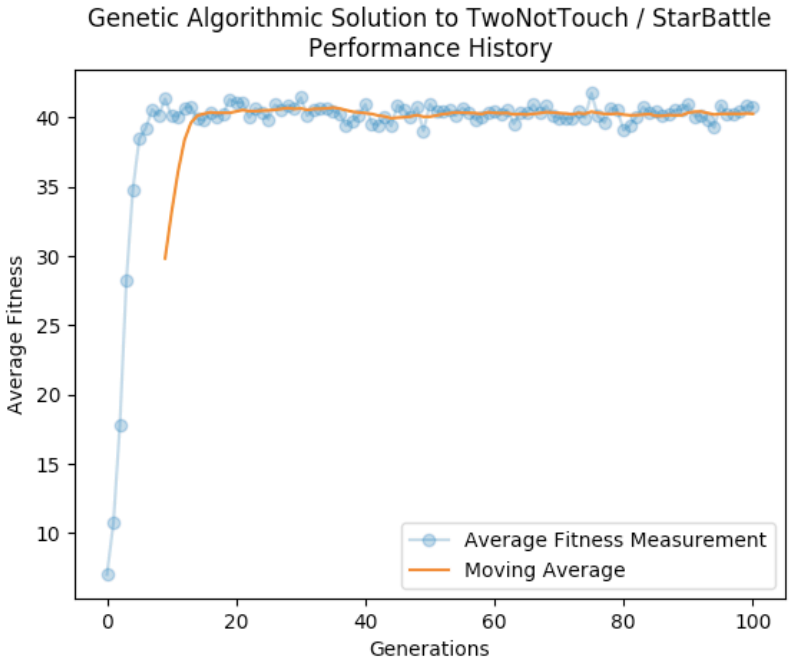
The graph above demonstrates that the GA gets us into an average generation fitness of approximately 37.5 (55 is the maximum fitness measure) fairly quickly. However, the performance plateaus fairly quickly thereafter.

After getting these unimpressive results, we tweaked some parameters of the algorithm. Here are some of the changes we made in attempt to eke out more performance from the algorithm.

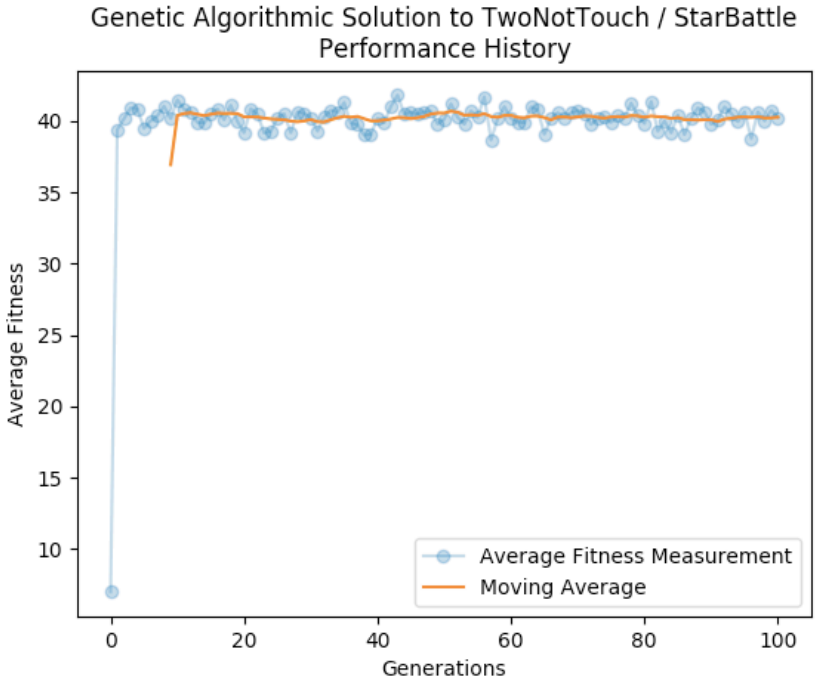
1) Mutation – instead of changing just one-star location in a mutation, we reduce the risk of creating illegal star locations by swapping two stars within a random set of stars. Each specimen contains two sets of ten stars. Each set of stars consists of a list of 10 integers. The combination of an index value and a list value is the row/column representation of a star location. By swapping two-star column values we ensure that each row/column pair within a star set is unique. This didn't change the performance in a meaningful way.



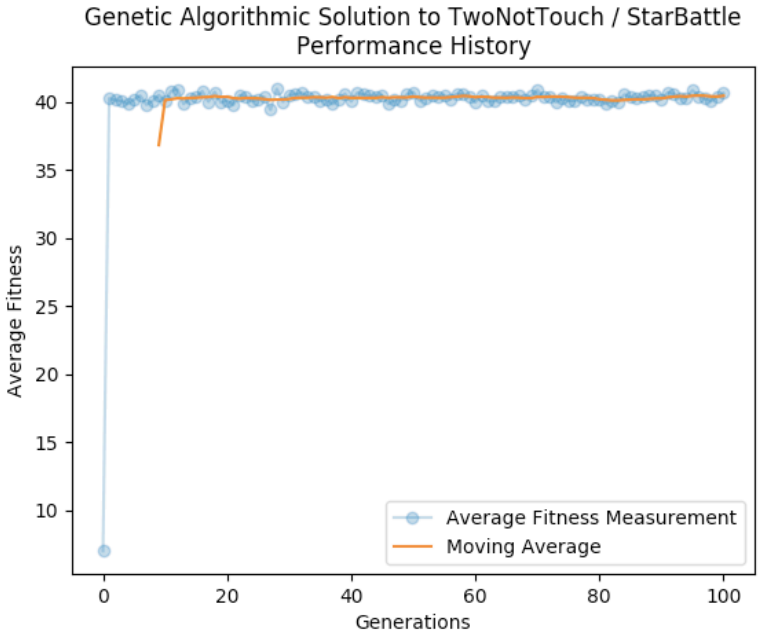
2) Crossover – we changed the crossover function from swapping the tail ends of a random star set to swapping entire star sets. This managed to push our plateau up from about 37.5 to about 40.



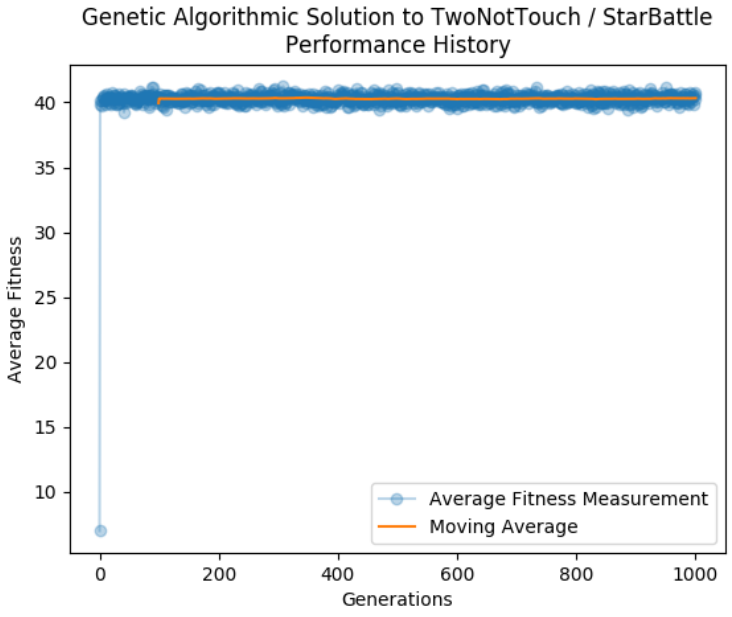
3) Natural Selection – We modified the probability of a specimen being selected for reproducing a member of the next generation such that if a specimen's fitness evaluation is below 44, the probability of reproduction is reduced to 0. This produced similar results as previous iterations.



4) Finally, we observed that most of the gains from the crossover occurs rather quickly. So, we capped the crossovers at 40 turns, after which future generations are modified only by mutation. Since we capped the crossovers, we also increased the mutation rate from 20% to 90%. This seems to get us up to that familiar plateau quicker and decreases our overall processing time.



5) The only thing left to try at this point is to increase the number of generations from 100 to experimentally larger generation caps. Essentially at this point we are just hoping to randomly find a solution. As of the writing of this draft report, so solution has been found in any of the GA experiments.



New and Busted – AC-3 Constraint Satisfaction

In addition to using a genetic algorithm to solve these puzzles, early discussions led us to attempt a constraint satisfaction approach to solving the problem. As we researched, this attempt became two attempts utilizing different CSP approaches. The first of was AC-3, which appears at a glance to be a good fit for solving this class of problem. Two Not Touch/Star Battle puzzles have logical sets of variables, domains and constraints. The variables in question are the stars (or more specifically, the positions within the grid which contain a star). Their domain is the set of possible valid locations within that grid for any of those stars. The constraints are as listed initially in the formulation of the problem: two stars per row, two per column, two per region, and no two stars anywhere in the puzzle which have direct adjacency to one another.

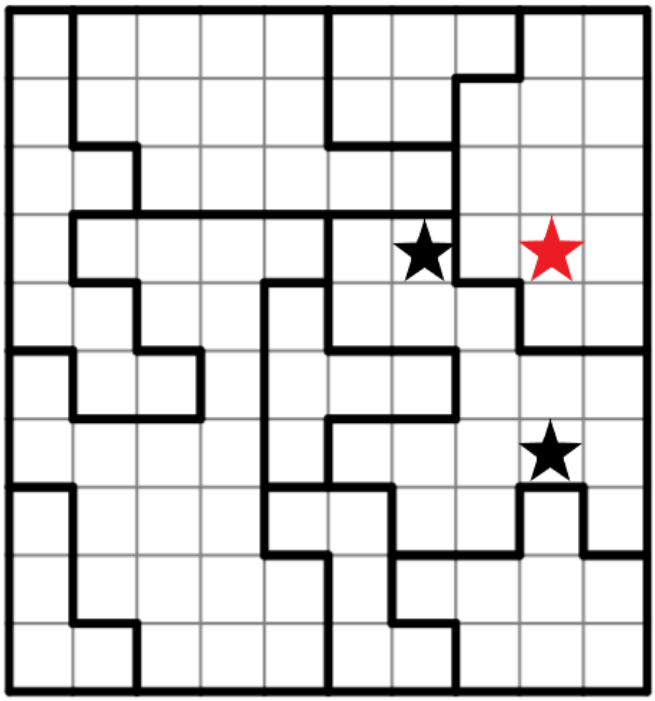
What was extremely difficult about implementing AC-3 is the complexity of satisfying all of these criteria – it was easy enough to model variables and a domain to which those variables could be designed. And it was easy to check for constraints across any individual arc. But the quantity of arcs and the lack of initial domain constraint became a difficult issue to overcome. In part, this difficulty stemmed from the way in which AC-3 was modeled in the code. The variables were modeled as a list of lists (each variable was a list of grid coordinates, stored within another list of all the variables). While this had some obvious benefits for constraint checking—for instance, it was very easy to check adjacency between any two variables using this structure—it also made the overall structure of the code much more complex. Similarly, the problem of generating all of the arcs in AC3 and then utilizing them in a way that accurately solved the problem proved very difficult to overcome. In the 2D model, the arcs were represented as a list of lists of lists... that is, a list of pairs of coordinates, each of which were stored in list form. The total number of arcs for the problem was $(n2 - n)$. Ultimately, the AC3 code was simply too unwieldy to process this for each calculation, develop a set of domains, assign a variable (or move an existing variable which has been flagged as not fitting into the constraints) and continue. At its best, the code would assign all the variables and then move them forever without ever producing a correct answer to the problem.

However, there is a much simpler way that this code could have been executed which would produce a correct answer sometimes. Unfortunately, this solution was hit upon much too late in the process to have been put into production, and it also has some inherent problems. Instead of approaching any puzzle in which we have to place more than one star per set of constraints as an $n \; x \; n$ grid with mn variables, where m is the number of stars to be placed within each constraint, the problem becomes much simpler when approached as an $n \; x \; n$ grid in which only one star has to be placed within each set of constraints, and which is solved multiple times and combined.

For instance, if we're solving an 8x8 2-star puzzle, all of the complexity issues with AC-3 can be halved by solving it more than twice for one star in each row, column and region, with no adjacent stars. Once two one-star solutions are generated, the same constraints can be applied to those two solutions, in order to check them against one another. However, this

simplification of the formulation does not constitute a foolproof method for producing a result – there are puzzles for which this will provably not work. Applying this method would require additional constraint checking, to determine whether the puzzle was solvable through this method. In this case, it would have to be done through the addition, during comparison of two solutions, of a constraint which checked whether any one star in any region shared a row AND a column with both stars in a different region.

To illustrate this point, the figure below contains part of the correct solution for this puzzle, assuming a 2-star solution. However, there is no way for the red star to fulfill the criteria of either of the two one-star solutions that we would hope to merge into a two-star solution for the same puzzle.



As the black stars in the figure are in the same region, the assumption would be that, given two 1-star solutions for this puzzle, each solution would contain one of those two correctly placed black stars. However, in a one-star puzzle, this would violate the constraint of not having more than one star in a row or column. The red star would have to occupy its current position in one of those two solutions, which would then not be a solution. So, while this puzzle is perfectly solvable using other methods or constraints, we would never be able to find two mutually exclusive solutions which would allow us to merge them into a single, accurate result. After manually attempting to solve several of these problems in this fashion, we estimate that fewer than 50% of overall puzzles are solvable using the combined solutions of two single-star problems.

Ultimately, AC-3 proved to be the wrong option for attempting to solve this problem. Given that the ideal starting state for AC-3 involves some initial restriction of the domain (down to a

subset of all possible domains), and that we are starting with a blank grid and no additional input, the algorithm runs forever attempting to make any sort of headway on solving the problem. It will assign variables, attempt to constrain domains, run into a problem, and back all of its progress out - over and over and over again. Reformulating the problem to attempt to find a solution several times over the course of the week has left the code in a state in which it will create all the possible arcs, create a set of domains out of the board and initialize the variables, but not run the algorithm. As the algorithm will just run forever without finding a solution, perhaps it is better that it does not attempt to do so. In different circumstances, such as being given a partially solved puzzle, AC-3 could perhaps eventually produce a viable solution. But without that leg up, it simply does not know where or how to start.

New Hotness - Constraints Satisfaction

I, *Evan DePosit*, chose to solve the puzzle using the constraint satisfaction Backtracking Search algorithm from the Russel and Norvig Text. Initially, I had planned on using the AC-3 algorithm, but I read that AC-3 can only solve the simplest of Sudokus. AC-3 only works for simple sudokus because often you can figure out which number goes in a square by determining that it can't be any of the other numbers that are in the same row, column, or group as it. This is what makes partially filled out sudokus quite easy. However, the "two not touch" puzzle has no initial placement of stars; therefor, one must start by guessing or searching for a solution. For this reason, I decided that the backtracking search would be more fruitful.

When the program is run, the board is transformed into a constraint satisfaction problem by creating two variables for each region of the board, so that each variable represents one of the stars in a region. The domains are initiated for each star to include the entire set of coordinates for its corresponding region. The constraint is a function that takes in a list of assignments and outputs True if they match the constraints of the problem and false if they do not. The constraint function, notTooClose(), checks for four things: that no assignment is the same as another, that no assignment is adjacent to another, and that there are no more than two assignments per row and column.

Implementation

The backtracking algorithm first checks if all the variables have assignments. If a solution has not been found, the select_unassigned_var() function returns the variable that has the smallest domain. The reason for this is that if the value can be found to not be a viable solution early, a large chunk can be pruned from the tree. This follows the minimum-remaining-value heuristic. Next, a value is chosen to try that is returned from the order_domain_values function. Although some variations of the backtracking algorithm will select a specified order to try values, this implementation does not. Instead, this function simply returns a subset of the domain of each variable that is arc consistent with the assignments made so far. Next, the inference() function restricts the domain of each variable by creating a set of coordinates that can no longer be tried. It does this by taking the new value and returning all the neighbors of that value, the value itself, and any coordinates in the same row or column as the value if two stars are in that row or column. These unavailable values are then deleted from the domains of the other variables by the order_domain_values() function. If the value that is tried results in failure, the unavailable values are added back to the domains of variables, the algorithm backs up, and the next value is tried after the one that resulted in failure.

```
def backtrack(self, assignment):
    #if assignment is complete then return assignment
    if is_complete(assignment):
        return assignment
    var = self.select_unassigned_var(assignment)

    for value in self.order_domain_values(var, assignment):
        #if value is consistent with assignment then
        if self.value_consistent_with_assignment(value, assignment):

            # add {var=value} to assignment
            assignment.vals[var]=value

            # inferecnes<-(csp, var, assignment) use to for arc consistency
            inferences = self.inference(var, assignment)

            #add inferecnes to assignment
            newUnavailable = inferences - assignment.unavailable
            assignment.unavailable = assignment.unavailable | newUnavailable

            #result <- backTrack(assignment, csp)
            result = self.backtrack(assignment)

            #if result != failsure
            if result:
                return result

            #remove {var=val} and inferences from assignemnt
            assignment.vals[var]=None
            assignment.unavailable = assignment.unavailable - newUnavailable

    #return failsure
    return None
```

Figure: Backtracking Algorithm

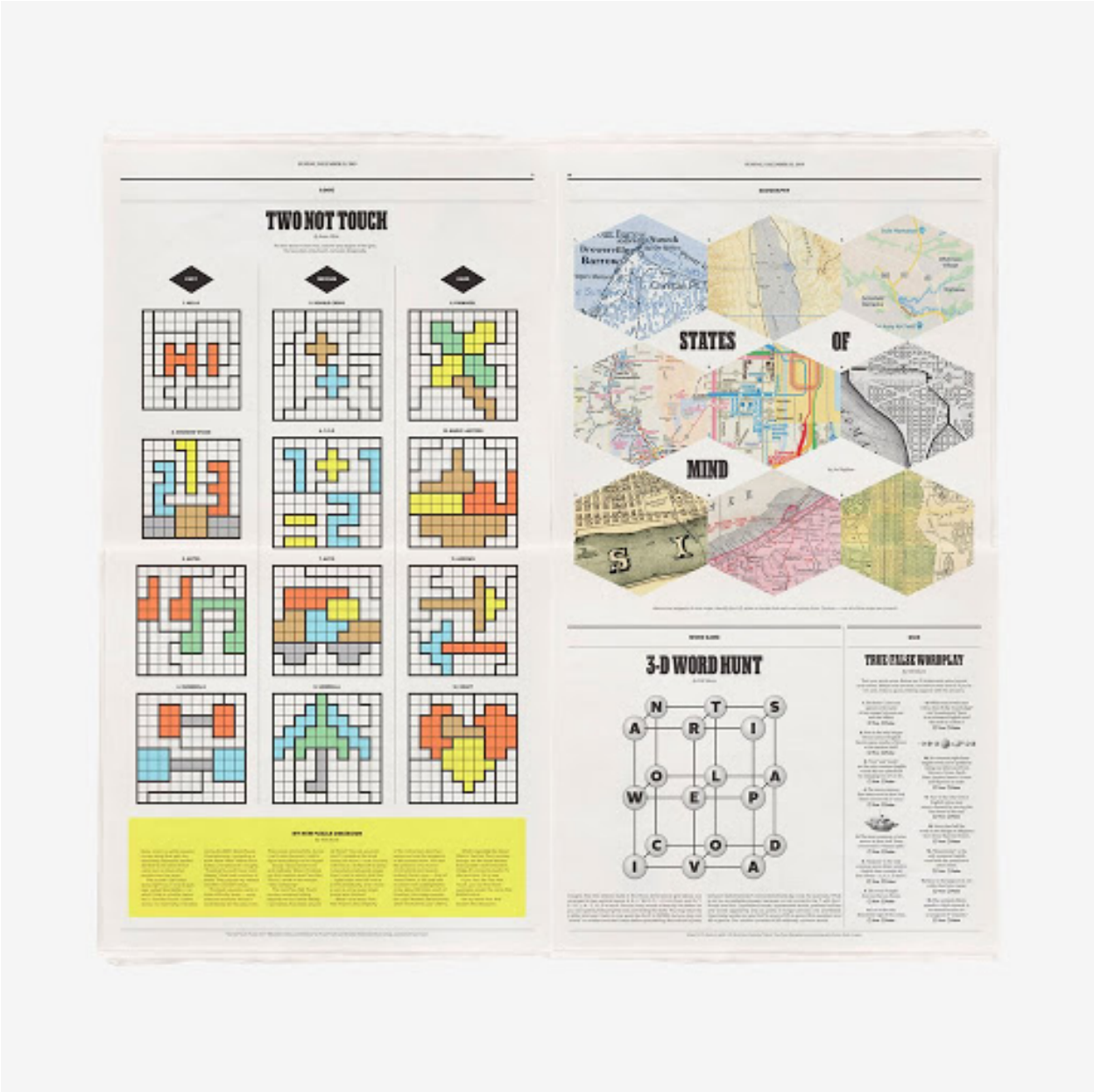
Results

I tested the backtracking algorithm on 3 easy puzzles, 1 medium puzzle, and 2 hard puzzles. The algorithm solved all but one of the hard puzzles. The one that it couldn't solve was the heart shaped puzzle that is the bottom right of Appendix A. The algorithm returns a None object when it fails to find a solution. There are no other clues as to why it failed other than it returned None, so discovering the reason for the failure will be an entire new undertaking.

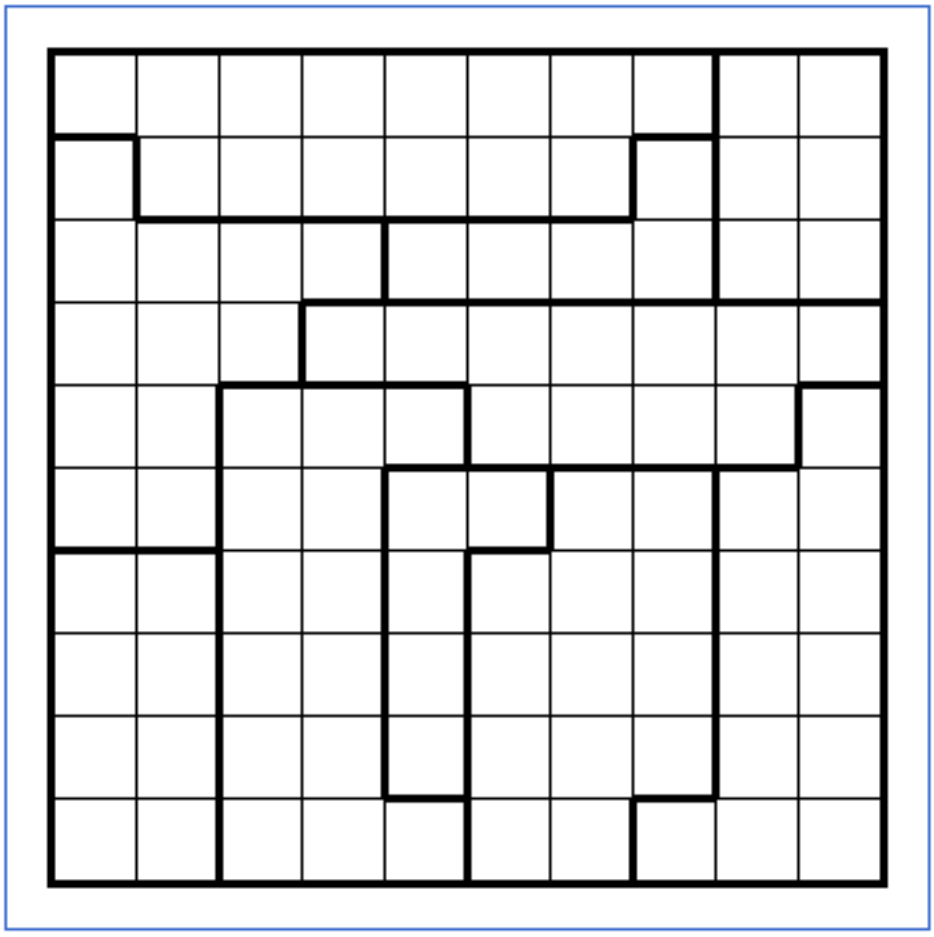
Conclusion

In short, for this application TwoNotTouch / StarBattle, Genetic Algorithm does not work, AC-3 could work in some cases, and Backtracking Algorithm usually works (unless things get hard).

Appendix A – TwoNotTouch from the New York Times [2]

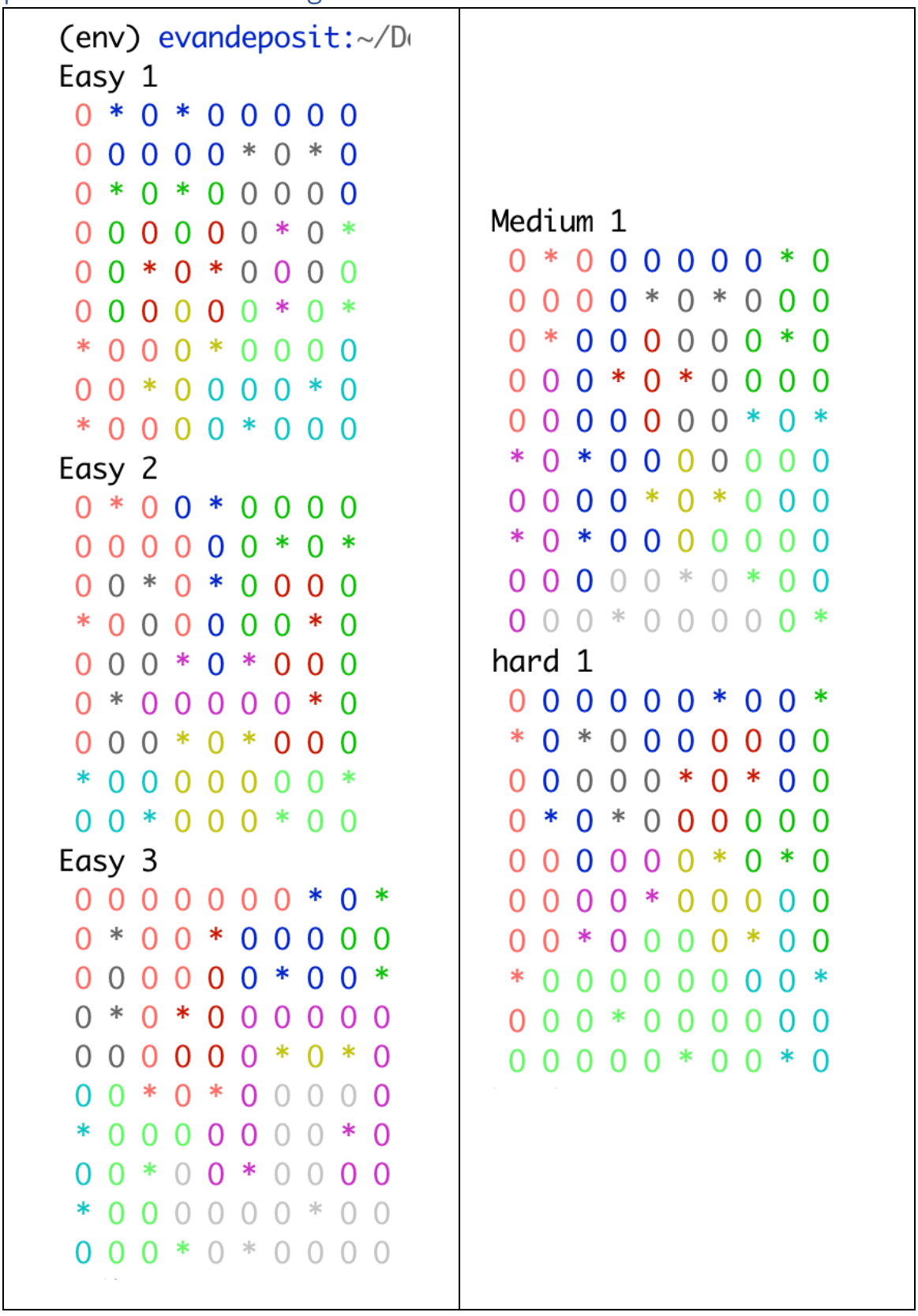


Appendix B – StarBattle [3]



10x10/2★ Normal Star Battle Puzzle ID: 6,650,565

Appendix C – Backtracking Results



Works Cited

[1] P. N. Stuart Russell, Artificial Intelligence: A Modern Approach, 3rd Edition, Upper Saddle River: Prentice Hall, 2010, pp. 126-129.

[2] "The New York Times," *The New York Times,* 2019.

[3] Puzzle Star Battle, "Puzzle-Star-Battle," 2020. [Online]. Available: https://www.puzzle-star-battle/com/?size=5. [Accessed 2020].