VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF SCIENCE

PROJECT 2

COLORING PUZZLE

SUBJECT: Introduction to Artificial Intelligence

| INSTRUCTORS |

Teacher Nguyễn Ngọc Thảo

Teacher Hồ Thị Thanh Tuyến

Teacher Lê Ngọc Thành

| PREPARED BY |

19127216 – Đặng Hoàn Mỹ

19127321 – Trần Xuân Sơn

19127603 – Đỗ Tiến Trung

HỒ CHÍ MINH CITY – AUGUST 2021

TABLE OF CONTENT

TEAM ROSTER

Student ID.	Full name	Tasks	Completion
19127216	Đặng Hoàn Mỹ	Prepare testcases Report Video demo GUI	100%
19127321	Trần Xuân Sơn	Generate CNFs pySAT A*	80%
19127603	Đỗ Tiến Trung	Prepare testcases Report Brute Force Backtracking	100%

INTRODUCTION

The CNF Satisfiability Problem (CNF-SAT) is a rendition of the Satisfiability Problem, where the Boolean equation is indicated in the Conjunctive Normal Form (CNF), that implies that it is a combination of statements, where a provision is a disjunction of literals, and an exacting is a variable or its invalidation.

This product of an AND of Ors helps us solve the problems once those have been encoded into Boolean logic. The solutions can be found impulsively by various algorithms and automatic solvers such as SAT solvers. These problems can be the n-queens problem, Sudoku puzzles, Einstein riddle, n-puzzles, Graeco-Latin squares...

Especially, the Coloring Puzzle belongs to that group of problems mentioned above. The puzzle needs filling with either green or red color and the number inside the cell corresponds to the number of green squares adjacent to that cell.

Given a matrix of size $m \times n$, this problem will be solved by pySAT library, A* algorithm, brute-force algorithm and backtracking algorithm, all of these use the logical principles of generated CNFs.

The plan that we apply for this project about the Coloring Puzzle problem is first producing the CNFs by generating many cases on the paper. Subsequently, the team members start discussing and find how to sow the seeds of those CNFs as a premise for the algorithms behind.

Searching on google shows us the solution to this problem is similar to the game fill-a-pix. This helps us process those CNFs faster and ready to deal with the pySAT library (it is almost the same as solving the n-queens problem we used in the previous exercise).

From there, we extend the application of those CNFs to solve the Coloring Puzzle by other algorithms according to the requirements of the problem.

APPROACH

With the steps suggested in the assigned problem description, we approach the problem by starting to generate the logical variable assigned to each cell of the matrix.
We found out that we have to check out the number of cells that will be colored green and infer which other cells will be marked red, then reverse from red to green.
We proceeded to mark from number 1 to $m \times n$ to each cell to analyze the **CNFs**.

Thereafter, getting the cells around the numbered cells in the matrix in the input file. (gen_moves(i, j) function).

We noticed the similarity in the cells when coloring and writing CNFs, for example with a 4 x 4 matrix, in the upper left corner position with the number 2, the cells we need to focus on are 1, 2, 5 and 6. The logical pair we need to consider is $(-1 \vee -2 \vee -5) \wedge (-1 \vee -2 \vee -6) \wedge (-1 \vee -5 \vee -6) \wedge (-2 \vee -5 \vee -6)$.
Same with the opposite case $(1 \vee 2 \vee 5) \wedge (1 \vee 2 \vee 6) \wedge (1 \vee 5 \vee 6) \wedge (2 \vee 5 \vee 6)$.

Afterward, we see that CNFs are generated by combining k possible ordered and non-repetitive elements of a list of n cells. Thus we use combinations in the itertools library to generate that combination.

Then, we have CNFs made up of those combinations and return the array of those clauses.

Briefly about **pySAT library** and Glucose3 SAT solver, this module is used to solve problems by including CNFs.
“Glucose is based on a new scoring scheme for the clause learning mechanism of so-called "Modern" SAT solvers. It is designed to be parallel since 2014. This page summarizes the techniques embedded in all the versions of glucose. The name of the Solver name is a contraction of the concept of "glue clauses", a particular kind of clauses that glucose detects and preserves during search.”

First, initialize the Glucose3 module and in turn insert the CNFs clauses with the add_clause() function. Then we have the module resolve them and retrieve the model it has resolved. Accordingly, it is possible to derive a matrix of numbers 0 and 1, representing the colors, 1 being green and 1 being red.

With the **Brute Force algorithm**, we will start by getting all the cells in the clauses into the literals array. Then we take the cartesian product of 1 and -1 with the same length as the array literals. And multiply the numbers in that array by the array of literals to get the colorable instances in the matrix. Then checking if all CNFs are satisfied, returns a tuple consisting of True and the array of numbers of logical values. Since there are not any cases match, return False and None.

With the **Backtracking algorithm**, it will first take a cell to set the value True (eg cell 1), and then remove the CNFs containing this cell when it is True. Next, get the temp array containing the clauses that do not contain its negative truth (cell -1) because these CNFs are not satisfied. Then put this cell in the assignment array to mark assigned. And recursively to consider the remaining cells (2, 3, 4, ...) that exist in the clauses of temp. If the cells are fully assigned, True and the assignment array will be returned. If a clause is false, return False and None.
Continue to set the value False for that cell and consider the above conditions similarly. As both True and False are considered unsatisfactory, False and None are returned. It turns out that there is no solution to this puzzle.

With the **A* Algorithm**, we are not able to execute the problem using this algorithm because approaching the problem with heuristics is challenging.

EXPERIMENT

We experiment with the 5 different testcases on different testcases with sizes and times below (time is in seconds). The calculated time only calculates the running time of the algorithm, not including the matrix construction on the GUI. So we have some evaluation as follows about the testcases.

Methods and testcases	pySAT	Brute Force	Backtracking
3 x 3	0.002989	0.243149	0.007999
5 x 5	0.006005	1603.050403	0.026206
10 x 10	0.011997		355.996236
15 x 15	0.032004		9746.557383
20 x 20	0.063001		

The pySAT gives solution fastest in all the testcases. Other algorithms can be quick in the smallest case 3 x 3 but from the 5 x 5 case, Brute Force takes a huge amount of time. Same with the Backtracking algorithm, the 10 x 10 takes almost 6 minutes to solve and the 15 x 15 takes more than 2 hours to run.

Actually it is also dangerous to run large test cases with such slow algorithms, because those consume a lot of resources in the computer and we would wait for the output of the results. Therefore, we would not run those testcases.

In conclusion, the pySAT is the fastest and the Brute Force is the slowest due to it would try all the possible cases. The Backtracking is the medium one by trying and stopping at the right time when checking if it's wrong. And the A^* algorithm will not be shown up here because it could not be finished.

For more clarity, you can follow the video demo here. We need to run the program by running the file **main_GUI.py**.

CONCLUSION

When we received this topic, we also felt difficult. But after digging deeper, it also has certain challenges when it comes to completing it.

Like accessing to the pySAT library is as well as an opportunity to learn more about it and possibly self-study on problems that can be solved with CNFs through this library. The other algorithms are also quite difficult at first, but the effort also pays off when completing two algorithms Brute Force and Backtracking.

About the A* algorithm, it is probably a bit too difficult for us because there are almost not too many documents and instructions on how to solve the problem given in this project with those algorithms.

Regardless, we had a lot of fun working together and supporting each other through such a difficult project.

REFERENCES

- Posts, V. M. (2020, January 13). *solving SAT in python*. WordPress.Com. https://davefernig.com/2018/05/07/solving-sat-in-python/
- *The Glucose SAT Solver*. (n.d.). Laurent Simon. https://www.labri.fr/perso/lsimon/glucose/
- *Fill-a-Pix*. (n.d.). Conceptis Puzzles. Retrieved August 20, 2021, from https://www.conceptispuzzles.com/index.aspx?uri=puzzle/fill-a-pix
- Wikipedia contributors. (2021a, April 11). *Brute-force search*. Wikipedia. https://en.wikipedia.org/wiki/Brute-force_search
- Wikipedia contributors. (2021b, June 21). *Backtracking*. Wikipedia. https://en.wikipedia.org/wiki/Backtracking