VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
UNIVERSITY OF SCIENCE

REPORT PROJECT 02

Coloring Puzzle

Course: Introduction to Artificial Intelligence

Ho Chi Minh City, 20th August 2021

TABLE OF CONTENT

I. Introduction

- Problem: solve the coloring puzzle problem.
- Approaching:
  + Analize how to define CNFs.
  + Design UIs.
  + Use the pysat library and CNFs to solve the problem.
  + Design brute-force and backtracking algorithms.
  + Implement brute-force and backtracking.
  + Connect the logic and how the UI works to emit the application.

II. Degree of completion

No	Criteria	Complete level	Notes
1	Describe the logical principles for generating CNFs correctly.	100%	
2	Generate CNFs automatically.	100%	
3	Use the pysat library to solve CNFs correctly.	100%	
4	Use A* to solve CNFs without a library.	0%	
5	Program brute-force algorithm to compare with A* (speed).	100%	
6	Program backtracking algorithm to compare with A* (speed).	100%	
7	Graphic interface.	100%	
8	Give at least 5 test cases with different sizes.	100%	
9	Comparing result and performance.	75%	Lack A* algorithm
10	Comply with the regulations of submission requirements.	100%	

III. Program specifications

1. Programming language

- This project is implemented in the Python language.

2. Environment

- Operating system: MacOS, Windows (Windows 10 is highly recommended).
- Python version: Python 3.9.2 64 bit or Python 3.9.6 64 bit.

3. Folder structure

- The group roster is illustrated in Group-roster.pdf.
- The video demo link is placed in Video-demo.txt.
- The source code is placed in the **source** folder which is included in the submission folder.
- The source folder has a structure like this:

```
source
|__utilities
|    |__combination_algos.py
|    |__constant.py
|    |__file_io.py
|    |__util_funcs.py
|__main.py
|__backtracking_algo.py
|__brute_force_algo.py
|__pysat_algo.py
|__UI.py
|__input.txt
|__test_cases.txt
```

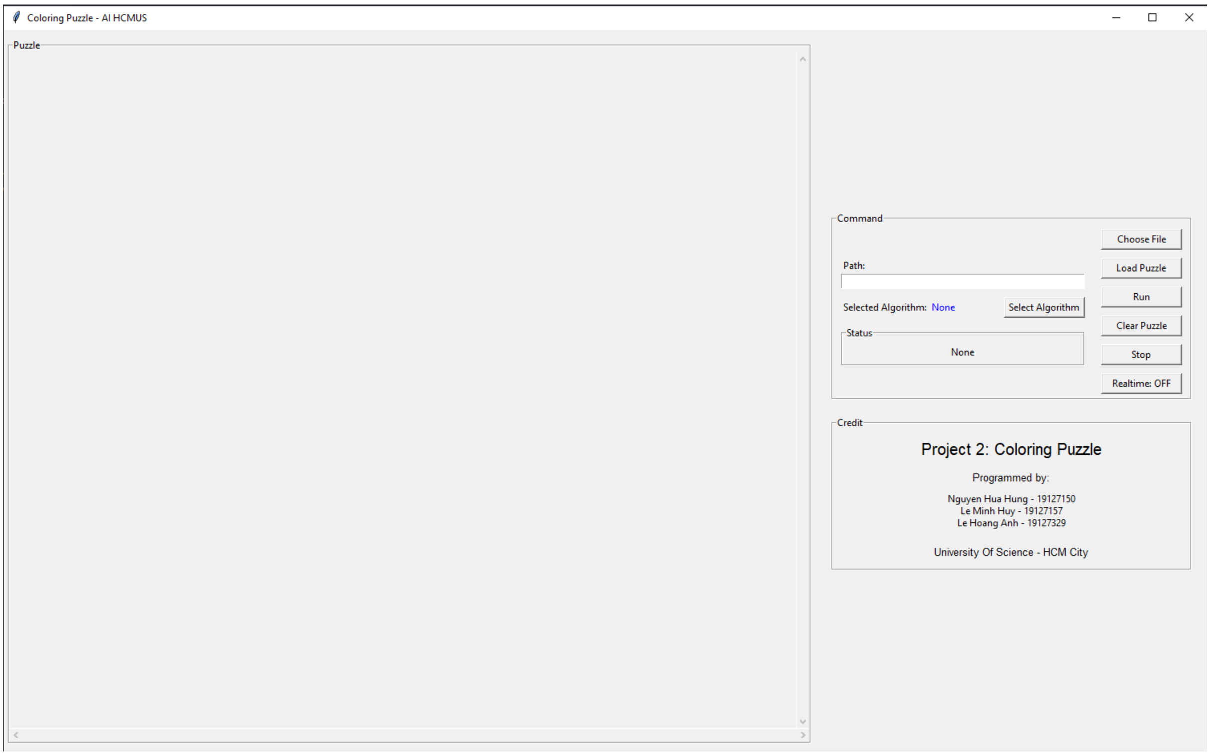
+ *utilities folder*:
    - *combination_algos.py*: generate combination.
    - *constant.py*: stores app constants.
    - *file_io.py*: input and output handler.
    - *util_funcs.py*: some utilities functions.
+ *main.py*: the entry point to run the program.
+ *backtracking_algo.py*: implement backtracking algorithm.
+ *brute_force_algo.py*: implement brute-force algorithm.
+ *pysat_algo.py*: implement program solving with pysat library.
+ *UI.py*: the GUI of the application.
+ *input.txt*: holds the input matrix. The format of this file is:
    - Two positive numbers n and m in the first line separated by a space: the size of the matrix.

- n next lines: with m numbers represent the value of a cell.

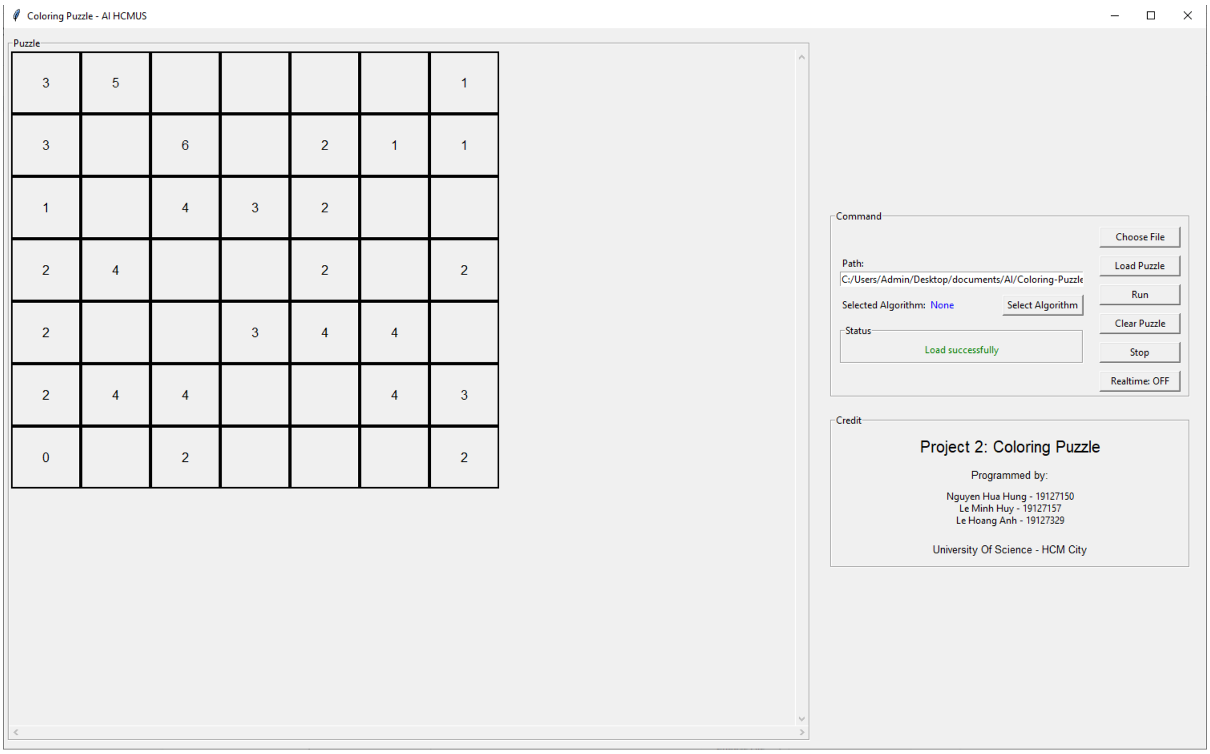
+ *test_cases.txt*: holds all the test cases with different sizes.
+ backtracking_output.txt: holds the output matrix when executing the **backtracking** algorithm. The format of this file is:
  - Two positive numbers n and m in the first line separated by a space: the size of the matrix.
  - n next lines: with m numbers represent the value of a cell.
+ brute_force_output.txt: holds the output matrix when executing the **brute force** algorithm. The format of this file is:
  - Two positive numbers n and m in the first line separated by a space: the size of the matrix.
  - n next lines: with m numbers represent the value of a cell.
+ pysat_output.txt: holds the output matrix when executing the **pysat**. The format of this file is:
  - Two positive numbers n and m in the first line separated by a space: the size of the matrix.
  - n next lines: with m numbers represent the value of a cell.

4. Usage

- Step 1: Unzip the submission folder and open the source folder.
- Step 2:
  + Before running the program, please check whether python exists in your computer or not. If not, please install Python first.
  + Install pysat library:
    - Open the command line (admin mode is highly recommended).
    - Run the command: ***pip install pysat***
    - If the previous step can not install pysat, run this command: ***pip install python-sat[pblib,aiger]***
- Step 3: Open the command line, change the directory to ***source*** folder.
- Step 4: Run this command ***python main.py*** or ***python3 main.py***
- Step 5: After step 4, there is a window like this



+ Step 5.1: Click the **"Choose File"** button or copy path of the input file (the format of input file is represented at section 3).
+ Step 5.2: Click the **"Load Puzzle"** button to load the puzzle. The puzzle looks like this:



+ Step 5.3: Click the **"Select Algorithm"** button to choose the appropriate algorithm (use pysat, A*, brute-force or backtracking).

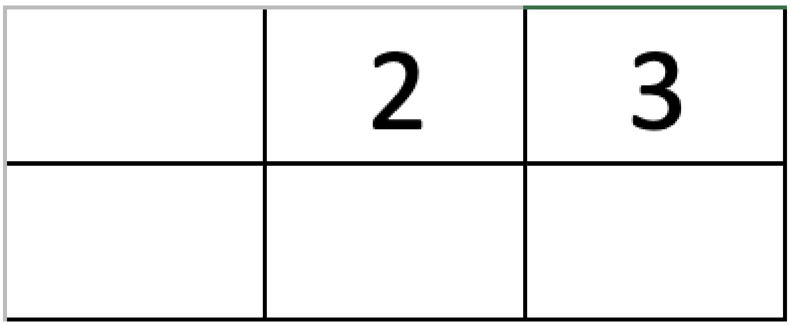
+ Step 5.4: Click the **"Run"** button to see the solution. When it is done, the output result will be stored in a corresponding file.

+ If the user wants to see the real time coloring process, click the **"Realtime"** button to turn on this mode.
+ If the program takes a great deal of time to run
  - Click the **"Stop"** button to stop the process (real time).
  - Close the windows application (no real time).

IV. Algorithms description

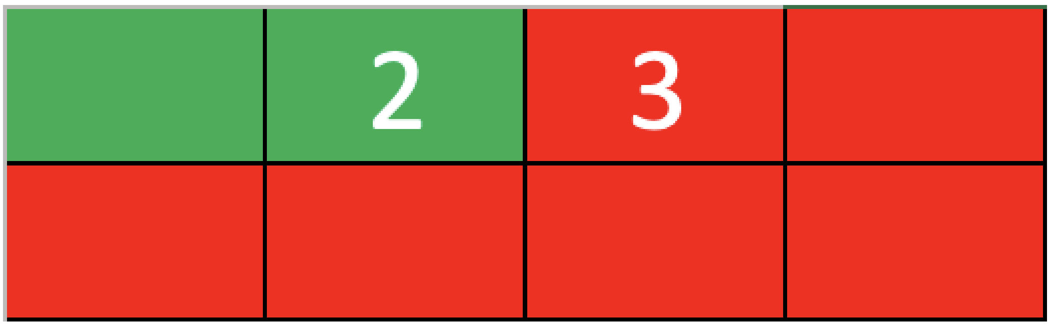
1. CNF details

- Describe how to generate a set of clauses in CNF form based on below picture (considers the second cell in the first row):
  + Value of each cell starts from 1 to 6 (from left to right and top to bottom).



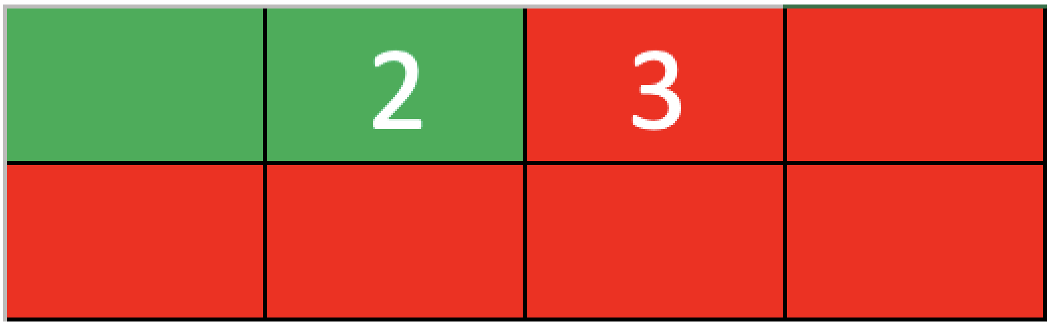
- Clauses that coloring red for the cells adjacent to a cell is a set of boolean expressions (each clause is a set of conjunction of negate green cells implies another negate cell in the set of remaining cells).
    + A clause in normal form: $(1 \lor 2) \Rightarrow (\neg 3 \land \neg 4 \land \neg 5 \land \neg 6)$
    + This clause in CNF form:
      $(\neg 1 \lor \neg 2 \lor \neg 3) \land (\neg 1 \lor \neg 2 \lor \neg 4) \land (\neg 1 \lor \neg 2 \lor \neg 5) \land (\neg 1 \lor \neg 2$
    + All clauses for this case (normal form) with clause in front of implication sign is all combinations (m-combination of a set S that described at the next bullet point):
      $((1 \lor 2) \Rightarrow (\neg 3 \land \neg 4 \land \neg 5 \land \neg 6)) \land$
      $((1 \lor 3) \Rightarrow (\neg 2 \land \neg 4 \land \neg 5 \land \neg 6)) \land$
      …………
      $((5 \lor 6) \Rightarrow (\neg 1 \land \neg 2 \land \neg 3 \land \neg 4)) \land$
- Clauses that coloring green for the cells adjacent to a cell is all the combinations (k-combination of a set S) is a set of conjunction of a set of disjunction of a combination:
    + S is a set of all cells adjacent to that cell.
    + n is the number of cells adjacent to that cell (includes itself).
    + m is the number of cells that need to color green.
    + k is the difference between n and m.
    + Details:
        - S = [1, 2, 3, 4, 5, 6] (mark cells from left to right and top to bottom respectively).
        - n = 6 and m = 2 => k = n - m = 4.
        - List of combinations: [1, 2, 3, 4], [1, 2, 3, 5], [1, 2, 3, 6], [1, 2, 4, 5], [1, 2, 4, 6], [1, 2, 5, 6], [1, 3, 4, 5], [1, 3, 4, 6], [1, 3, 5, 6], [1, 4, 5, 6], [2, 3, 4, 5], [2, 3, 4, 6], [2, 3, 5, 6], [2, 4, 5, 6], [3, 4, 5, 6].
        - Clauses in CNF form:
          $(1 \lor 2 \lor 3 \lor 4) \land (1 \lor 2 \lor 3 \lor 5) \land (1 \lor 2 \lor 3 \lor 6) \land$
          $(1 \lor 2 \lor 4 \lor 5) \land (1 \lor 2 \lor 4 \lor 6) \land (1 \lor 2 \lor 5 \lor 6) \land$
          $(1 \lor 3 \lor 4 \lor 5) \land (1 \lor 3 \lor 4 \lor 6) \land (1 \lor 3 \lor 5 \lor 6) \land$
          $(1 \lor 4 \lor 5 \lor 6) \land (2 \lor 3 \lor 4 \lor 5) \land (2 \lor 3 \lor 4 \lor 6) \land$
          $(2 \lor 3 \lor 5 \lor 6) \land (2 \lor 4 \lor 5 \lor 6) \land (3 \lor 4 \lor 5 \lor 6)$
    - All the clauses for this case (the second cell in the first row) is a set of conjunction of both clauses of the first and the second bullet points.

2. Pysat
    - Using **Glucose 3 SAT solver** in pysat.solver lib.
    - Using **add_clause()** method to add each clause in a set of CNF clauses generated.
    - And then, **using solve()** function to solve this problem. This function will return True if solved, otherwise return False.
    - Finally, using **get_model()** to get a value of all variables in CNF clauses (positive value if this cell is green color, negative value if this cell is red color).
3. Backtracking
    - Loop through all cells that contain a number from left to right and top to bottom.
    - At each cell, get all the combinations (k-combination of a set S):
        + S is a set of all the red cells adjacent to that cell (includes itself).
        + k is the number of **remaining** cells that need to color green.
        + Example: the third cell in the first row.



            ○ S = [3, 4, 6, 7, 8] (mark cells from left to right and top to bottom respectively) and k = 2.
            ○ List of combinations: [3, 4], [3, 6], [3, 7], [3, 8], [4, 6], [4, 7], [4, 8], [6, 7], [6, 8], [7, 8].
    - Coloring all the cells in a combination (set of numbers) to green and jump to the next cell that contains a number.
    - If the current cell has the number of the green cells adjacent to itself that is greater than the number inside or looped through all combinations of that cell, then the **backtracking** function will go back to the previous cell and try another combination.
    - The **backtracking** function will run until the completion of coloring the last cell that contains a number.

4. Brute force
    - Loop through all cells that contain a number from left to right and top to bottom.
    - At each cell, get all the combinations (k-combination of a set S):
        + S is a set of all the cells adjacent to that cell.
        + k is the number of cells that need to color green (the number inside itself).
    + Example: the third cell in the first row.



        - S = [2, 3, 4, 6, 7, 8] (mark cells from left to right and top to bottom respectively) and k = 3.
        - List of combinations: [2, 3], [2, 4], [2, 6], [2, 7], [2, 8], [3, 4], [3, 6], [3, 7], [3, 8], [4, 6], [4, 7], [4, 8], [6, 7], [6, 8], [7, 8].
    - Coloring all cells in a combination to green and jump to the next cell that contains a number.
    - If the current cell has the number of the green cells adjacent to itself that is greater than the number inside or looped through all combinations of that cell, then the **brute force** function will go back to the previous cell and try another combination.
    - The **brute force** function will run until the completion of coloring the last cell that contains a number.

V. Experiment

- The experiment is done using a console. The result is represented in the table below.
- Because of doing experiments on the console, our team has to change the code in *main.py*. After the testing process, we change the code to the origin that is the submission version in the source folder.
- Because some cases take a lot of time to solve, the time limit is 2 minutes to avoid memory overloading and time consuming.
- In this experiment, three algorithms are used except A* because of incomplete implementation.

- All test cases are stored in file test_cases.txt with different sizes.

Algorithm	Running time (seconds) per size					
	3x3	5x5	7x7	9x9	11x11	20x20
Pysat	0.00598	0.01795	0.09175	0.10362	0.29026	0.65518
Brute-force	0.01598	timeout	timeout	timeout	timeout	timeout
Backtracking	0.00102	0.00199	0.00096	0.00099	0.00295	0.23636
A*	N/A	N/A	N/A	N/A	N/A	N/A

VI. Conclusion

- Pysat library runs with the average time when the size of the test case increases.
- Brute-force takes a lot of time to run when the size scales up.
- Backtracking is the fastest solution (according to the result table).

VII. References

1. Pysat library:
   https://pysathq.github.io/docs/html/api/solvers.html
2. Python installation guide line:
   https://www.python.org/downloads/release/python-396/
3. Tkinter tutorials and document:
   https://www.tutorialspoint.com/python/python_gui_programming.htm
   https://docs.python.org/3/library/dialog.html
   https://youtu.be/0WafQCaok6g
   https://youtu.be/jnrCpA1xJPQ
4. Threading in python:
   https://docs.python.org/3/library/threading.html