**Project Report**

1. **General information**

* **Team members:**
* Hồ Nhật Linh – 19127652
* Võ Quang Huy – 19127425
* Nguyễn Ngọc Anh Khoa – 19127445
* **Division of tasks:**

| **Member** | **Task** |
| --- | --- |
| Võ Quang Huy | Write CNF clauses, implement CNF and Backtracking. |
| Hồ Nhật Linh | Implement A\* and GUI, write project report. |
| Nguyễn Ngọc Anh Khoa | Generate test cases, run test cases to get time, implement Brute Force. |

* **The structure of the submission folder:**

----------AStar

----------Backtracking

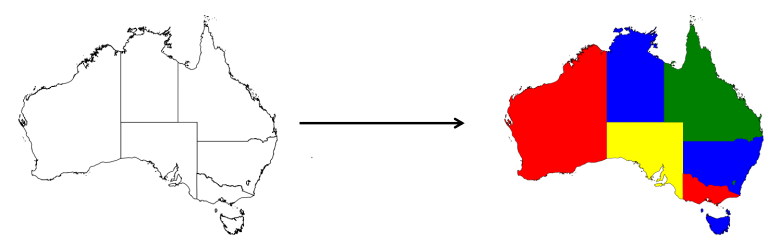
----------BruteForce

----------CNF

----------TestCases # Contains test cases, each folder contains test cases of the same size (red: many red cells, green: many green cells).

----------Videos #Contains videos recording the process of running the test cases.

1. **Problem analysis**

* The coloring problem is a classic example for the theory of CSP, depending on the condition of the shaded areas, the problem will become more difficult or easier, the simplest example that we often encounter is to color a map so that 2 adjacent regions have different colors. However, with more rigorous problems, we cannot just "rely" on CSP but must have more sophisticated solutions to address the problem. 

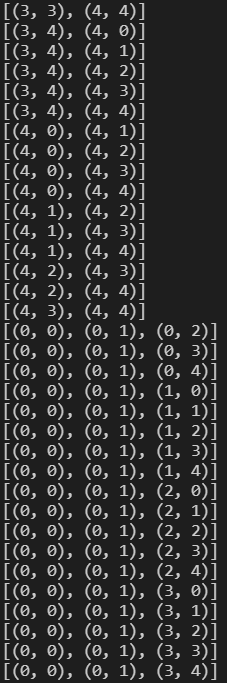
*(An example of a coloring problem. Source:* [Program like it’s 1970!. A little throwback to what AI used to… | by Fabian Stern | Towards Data Science](https://towardsdatascience.com/program-like-its-1970-6708df2ed101))

* For the requirement of the problem, we see that it contains a quite strict condition, when it is strong to specify that how the color of a cell should be to meet the required quantity of it and surrounding cells, in the other words, the number of solutions for a problem instance will not be many.
* First of all, we just consider the problem as an ordinary search problem, at each step we will have successors when coloring one more red cell to the current state. So, what kind of search strategies will we have?
* We can’t bear to ignore the "close friend" of CSP – Backtracking search, one of the biggest advantages that we can talk about is its lightness in memory consumption, Backtracking will only save 1 current route without no need to care about other branches of the search tree, of course we will also have to deal with it to avoid jumping into infinite loops.
* In contrast to the depth approach of Backtracking, we have the wide approaches. If simply using BFS, it will be difficult to find a solution if there are a lot of red cells, because the search has to dive too deep in the search tree. For GBFS, it is also a possible solution, but as mentioned, this is a problem containing a few solutions, moreover, there are many states with the same heuristic, so it is easy to get stuck in the dead ends. So, we will combine them, using the A\* algorithm to take the advantage of the above algorithms, perhaps the biggest concern with A\* is the memory consumption, when saving both frontier and expanded list will be very expensive. The most advantage of A\* that it gives us an optimal solution rather than the Backtracking.
* Secondly, we will use propositional logic, solving the problem by building propositions from the given condition. As mentioned, this problem gives us the condition, which is fairly strict, it is beneficial to use propositional logic because the tighter the condition is, the fewer and better the propositions are. The solution will also be neater and less expensive than the proposed search solutions above.
* Finally, we will use a ridiculous solution, which is Brute Force, perhaps the only thing that we can say about its benefits is its simplicity in implementation, but in terms of time can be a pain for computers running it. Brute force in this project will be used for comparison purposes only.

1. **Implementation**
2. **Brute force**

This is the simplest solution; we will generate all possible states in the program and then check whether each state meets the condition or not.

Firstly, to create states, we will consider the matrix as a 2-dimensional array, each position will be the array size \* row + column. Then, we proceed to consider all the possible sizes of the matrix, for each size, we will take the combination of all possible positions of the red cells to conduct the goal test, because now we have translated positions in the matrix to integers (instead of a pair of indexes), we can write a function taking the combination from all positions in the matrix with the given size. For example, a 3x3 matrix, we consider the state size to be 2, then there will be combinations of positions of red cells (rows, columns): we get combinations of (0, 1), (0, 2) , (1, 0), (1, 1)… then the corresponding positions will be [(0,0), (0, 1)], [(0, 0), (0, 2)], [( 1, 0), (1, 1)]… for each such combination of positions, we will proceed to check whether the current state is satisfied or not.

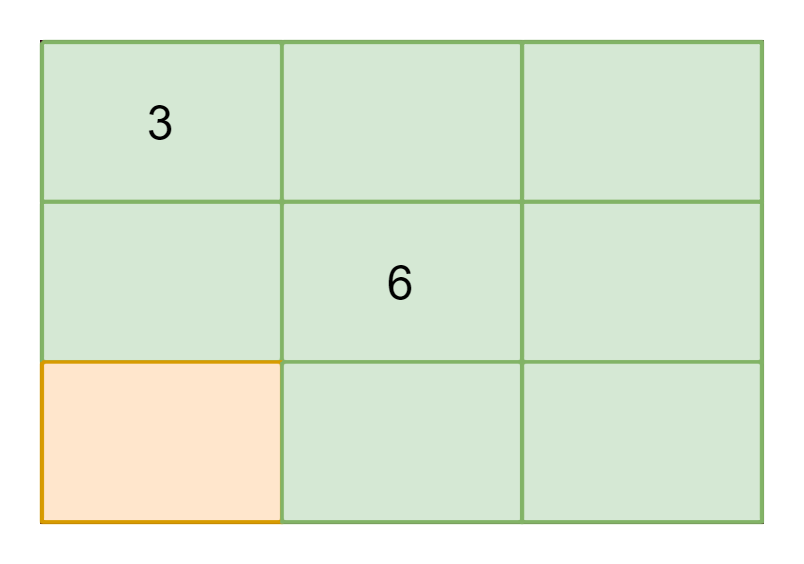


*(An example of states we've colored, including the locations of the red cells)*

About how to check a state is the goal state or not, we will proceed to go through all the elements in the matrix, if an element has a number (it contains condition), we will count how many green cells being around it, if this counter is equal to the value in the matrix, then this cell is satisfied. Just like that, if all cells in the matrix are satisfied then this is the goal state.

1. **Backtracking**
2. **A\***

To build the A\* algorithm, we first refer to the formula calculating the heuristic value of a state, there will be many heuristic functions, but we will choose one function so that a state having strong representation, the closer to the target, the smaller the heuristic value is. Here we will use a heuristic function being the sum of all the differences of the number of green squares around a cell minus the number of green neighbors the cell needs (only consider the conditional cells).



*(An example shows how to calculate the heuristic value of a state)*

In the above state, we can calculate the heuristic at cells (0, 0) (3) and (1, 1) (6). In cell (0, 0), we have the number of green cells adjacent to it is 4, while it needs 3 green cells, the heuristic value in this cell is 1. In cell (1, 1), we have the number of green cells adjacent to it is 5, while it needs 6 green cells, the heuristic value in this cell is 0. So the sum heuristic of this state is 1.

We also see that this is a consistent heuristic function because between a state and the goal, there will not be another better state that has a smaller heuristic value.

Next is the value of the path (path cost), because of the abstraction of how to define the “distance” between 2 states. We consider that the path cost is always 1 for going from a state to another state (coloring one more red cell). So, the path cost is always equal to the number of red slots.

Next is how to save the frontier as well as the expanded list. We will save a state as the positions of the red cells in the matrix.

Ảnh có chứa văn bản

Mô tả được tạo tự động

(An example of saved states)

Since this is A\*, the states with min Fn value regularly get during the search, saving the frontier as a priority queue will make the algorithm very slow because we need O(n) to search in the queue. Therefore, we will use a hash table to make the process of searching faster, each entry of the hash table will be a list of states with the same key, here we calculate the key by summing the positions of red cells in the matrix (positions are converted to integers by the formula = size \* row + column).



(Ví dụ 2 trạng thái trong A\*)

In the above example, the matrix has size 5, the red square (3, 0) gives us the position of 5 \* 3 + 0 = 15, the red square (4, 2) gives us the position of 4 \* 5 + 2 = 22. So, we have the key of the state as 15 + 22 = 37. As we can see, it is quite possible to have a collision, so for each entry, we store a list of states with the same key. Although, in theory, it is still O(n), in reality, the fact that 2 states have the same key is not often, so, in practice, it will be much faster than the priority queue.

To check a state is the goal state or not, we save the heuristic value of the state, if the heuristic value is 0, we continue checking every slot in the matrix, each slot needs to have the number of green neighbors equal to the carried number, if all the slots are satisfied then the state is the goal state.

We build the A\* algorithm after having the above means, firstly we create an empty frontier, an expanded list, and push the initial state (no red cells) to the frontier. While the frontier is not empty, we get the state with the smallest Fn (sum of path cost and heuristic), check the state is the goal state or not, then get all of its successors by temporarily coloring a green slot as red, each successor will be consider being contained in the frontier and the expanded list or not, we ignore if it belongs to the expanded list or to the frontier, otherwise, we add a new ready state to the frontier, we also recover the parent state (coloring green for the processed slot) after considering the successor to generate other successors.

1. **CNF**
2. **The logical principles for generating CNFs:**

* Based on the digit in the cell 🡪 Calculate its combinatorics of surrounding cells which is the number of clauses of that cell.
* In each clause of the main cell: The number of surrounding cells that are green is equal to the digit in the main cell and other surrounding cells are red.
* For example, we have a 3x3 table and cell at (2, 2) has digit: 4 (table below)

|  | 1 | 2 | 3 |
| --- | --- | --- | --- |
| 1 | . | . | . |
| 2 | . | 4 | . |
| 3 | . | . | . |

The combinatorics of that cell would be: **9C4 = 126 clauses**

Assume that we marked 4 cells: (1, 1), (1, 2), (1, 3), and (2, 1) as **green**. Others (2, 2), (2, 3), (3, 1), (3, 2), and (3, 3) (5 in total) cells must be masked as **red only**. Thus, in contrast, if 5 cells were masked as **red**, the other would be masked as **green**.

* The clause: ((1, 1) ∧ (1, 2) ∧ (1, 3) ∧ (2, 1)) ⬄ (¬(2, 2) ∧ ¬(2, 3) ∧ ¬(3, 1) ∧ ¬(3, 2) ∧ ¬(3, 3)).
* **CNF** would be: *(¬(1, 1) ∨ ¬(1, 2) ∨ ¬(1, 3) ∨ ¬(2, 1) ∨ ¬(2, 2))* ***∧*** *(¬(1, 1) ∨ ¬(1, 2) ∨ ¬(1, 3) ∨ ¬(2, 1) ∨ ¬(2, 3))* ***∧*** *(¬(1, 1) ∨ ¬(1, 2) ∨ ¬(1, 3) ∨ ¬(2, 1) ∨ ¬(3, 1))* ***∧*** *(¬(1, 1) ∨ ¬(1, 2) ∨ ¬(1, 3) ∨ ¬(2, 1) ∨ ¬(3, 2))* ***∧*** *(¬(1, 1) ∨ ¬(1, 2) ∨ ¬(1, 3) ∨ ¬(2, 1) ∨ ¬(3, 3))* ***∧*** *((2, 2) ∨ (2, 3) ∨ (3, 1) ∨ (3, 2) ∨ (3, 3) ∨ (1, 1))* ***∧*** *((2, 2) ∨ (2, 3) ∨ (3, 1) ∨ (3, 2) ∨ (3, 3) ∨ (1, 2))* ***∧*** *((2, 2) ∨ (2, 3) ∨ (3, 1) ∨ (3, 2) ∨ (3, 3) ∨ (1, 3))* ***∧*** *((2, 2) ∨ (2, 3) ∨ (3, 1) ∨ (3, 2) ∨ (3, 3) ∨ (2, 1))*
* Consequently, we would have **126 CNFs** like this for that cell.`
* Note: If the cell at (3,3) has digit: 3 *(table1)*, there will be **4C3 CNFs** for this cell (***it*** ***won’t depend on the cell with digit 4***). So, **4C3 CNFs from this cell** will use **logical conjunction ‘∧’** to combine with **9C4 CNFs from the 4 digit-cell.** Thus, one of the results would be like *(table2).*

|  | 1 | 2 | 3 |
| --- | --- | --- | --- |
| 1 | . | . | . |
| 2 | . | 4 | . |
| 3 | . | . | 3 |
|  | 1 | 2 | 3 |
| 1 | . | . | . |
| 2 | . | 4 | . |
| 3 | . | . | 3 |



1. **Algorithm for generating CNF:**

* For every cell that has a digit *k* and *n* surrounding cells, there will be  **CNFs** equivalent withways to mark ***k* surrounding cells as green**.

|  | 1 | 2 |
| --- | --- | --- |
| 1 | 2 | . |
| 2 | . | . |

* Assume that there is a cell has digit **2** and **4** surrounding cells:
* Firstly, we generate a **combinatorics list** contains ways to mark cells as green. For example: the **combinatorics list would be:**

[[(1,1), (1,2)], [(1,1), (2,1)], [(1,1), (2,2)], [(1,2), (2,1)], [(1,2), (2,2)], [(2,1), (2,2)]]**.**

* Then, for each *element in the* ***combinatorics list***, we mark other cells which are *not in the element* as red and generate another list containing those red cells. For example, we pick [(1,1), (1,2)] in the **combinatorics list** (called **green list**) then **the red list** would be [(2,1), (2,2)].
* The actual CNF would be like this:

(¬(1,1)∨ ¬(1,2)∨ ¬(2,1)) ∧ (¬(1,1)∨ ¬(1,2)∨ ¬(2,2)) ∧ ((2,1)∨(2,2)∨(1,1)) ∧ ((2,1)∨(2,2)∨(1,2))

* According to the CNF above, there would be:
* A loop for every green cell in the **green list,** we add it to the new **red list** which meansthereare 2 new lists generated**.**
* A loop for every cell in the **red list**, we add its negation to the new **green list** which is negated. Thus, there are 2 new lists generated and obviously, these lists are negated.
* Ultimately, combine those lists above then we have ***one CNF*** ***for one state of one cell in the table.***

1. **Experiment**

As we said, there are several ways to make the problem become harder or easier. So, we will consider some sort of elements: the size of the matrix, the number of green/red slots in the goal state, and the algorithm used. Because the running time will be too long (can exceed the day unit), we constraint the limit time for each test case to be 15 minutes.

| Size | Input | Algorithm | Running time in seconds |
| --- | --- | --- | --- |
| 5x5 |  | Brute Force | 53 |
| A\* | 1 |
| Backtracking | 0.009 |
| CNF | 0.0017 |
|  | Brute Force | Over limit time |
| A\* | 0.067 |
| Backtracking | 0.008 |
| CNF |  |
| 10x10 |  | Brute Force | Over limit time |
| A\* | Over limit time |
| Backtracking | 2.77 |
| CNF | 0.037 |
|  | Brute Force | Over limit time |
| A\* | 0.5 |
| Backtracking | 0.006 |
| CNF |  |
| 15x15 |  | Brute Force | Over limit time |
| A\* | 0.069 |
| Backtracking | 0.012 |
| CNF | 0.02 |
|  | Brute Force | Over limit time |
| A\* | Over limit time |
| Backtracking | 0.004 |
| CNF |  |
| 20x20 | Ảnh có chứa vòi hoa sen  Mô tả được tạo tự động | Brute Force | Over limit time |
| A\* | Over limit time |
| Backtracking | 52 |
| CNF | 0.1 |
|  | Brute Force | Over limit time |
| A\* | Over limit time |
| Backtracking | 0.0165 |
| CNF |  |

* We can see that the size of the matrix has a great effect on the running time of the A\* and Brute Force. With Brute Force, we have to generate and check many states. And A\*, since the size is big, it needs to generate a lot of states and make the search tree grow too large, it often takes too much time in the large case. Although Backtracking has little running time, it always often us non-optimal solutions.
* In the test cases, we tried to create two kinds of test cases consisting of more red slots cases and more green slots cases. In the implementation of Backtrack, we color green for red slots, in contrast to A\* (color red for green cells), because we separated the implementation tasks for different members. But in general, it’s beneficial if we use a color having the same with the color covering much more the other in the matrix, for example, if we have a matrix 20x20 that needs 15 red cells and 5 green cells, we can take advantage of it if we generate the first state is all red slots and use the green color to paint the matrix.
* CNF is stable and useful in every case, although the size and the red slots change through the test cases. Since we use the proportions, the program ran “smarter” than the algorithms above.

1. **Conclusions**

We should avoid using Brute Force because it’s very costly, we also need to run several days to find the solution.

Although A\* can give us an optimal solution, its memory consumption is a big problem. Moreover, in the experiments, we can see that if the input size increases too much, A\* takes much time to solve the problem because it needs to generate and check many states in the process. In A\*, we have recognized that, instead of using a queue or priority queue to store successors, we can have alternative techniques like a hash table, tree (AVL or RBT) to get less running time.

Backtracking shows its strength in running time, although the solution is costly and non-optimal, we shall apply it in case we want to get the solution as soon as possible.

We can see that the CNF is the best way to solve the problem among the proposed problems. It’s better than about time and memory aspects.

1. **References**

Implement the combinations function:

<https://docs.python.org/3/library/itertools.html>