VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
VNUHCM - UNIVERSITY OF SCIENCE
FACULTY: INFORMATION TECHNOLOGY

---o0o---

REPORT

PROJECT 02: COLORING PUZZLE

COURSE: CSC14003 – INTRODUCTION TO ARTIFICIAL INTELLIGENCE

TEACHER: NGUYỄN NGỌC THẢO
HỒ THỊ THANH TUYỀN
NGUYỄN HẢI ĐĂNG

MEMBERS: LÊ VŨ HUY – 19127421
NGUYỄN QUANG PHÚ – 19127507
LÂM HOÀNG PHÚC – 19127512

CLASS: 19CLC1 – TERM III/2020-2021

08-2021

CONTENTS

1. INTRODUCTION:

Coloring puzzle: Given a matrix of size $m \times n$, where each cell will be a non-negative integer or zero (empty cell). Each cell is considered to be adjacent to itself and 8 surrounding cells. Your puzzle needs to color all the cells of the matrix with either blue or red so that the number inside each cell corresponds to the number of blue squares adjacent to that cell *(see Figure 1).*

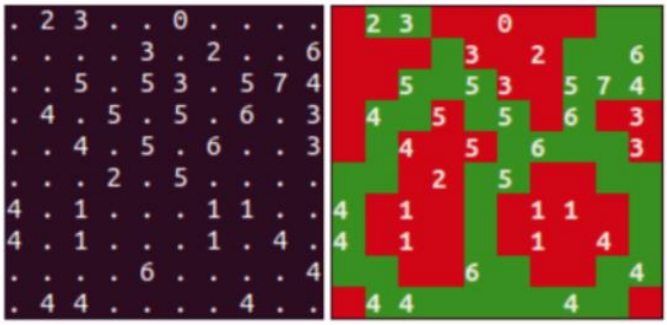


Figure 1 An example of input matrix (left) and output matrix (right)

The above problem will be solved in 3 directions:

- Write constraints for cells containing numbers to obtain a set of constraint clauses in CNF. Then, using the ***pysat*** library or the ***A**** algorithm to find the value for each variable and infer the result.
- Solve the problem directly using ***Brute-Force*** algorithm.
- Solve the problem directly using ***Backtracking*** algorithm.

Input: - *The first line* contains two positive integers m, n representing the number of rows and columns of the matrix.

- *The next lines* are non-negative integers representing the number of blue cells adjacent to it. If there is no constraint, the value of that cell is -1.

Output: The result of the problem is a matrix of two values **0** and **1** (value 0 *(False)* represents red, value 1 *(True)* represents blue).

Example:

Input	**Output**
3 3 3 5 3 4 6 4 2 3 2	1 1 1 1 0 1 0 1 0

2. APPROACH

2.1 Generating CNFs

Idea: Iterate the matrix, if a cell contains value which is greater than -1, we generate a set of clauses CNF. for the cell.Assume we have value k,n adjacency cells, we need to generate clauses for both the red and green case:

1. In the green case, we need to generate C_n^{k-1} clauses, which means we need to iterate C_n^{k-1} times, each iteration we choose a set of k-1 adjacency cells and make sure the set does not duplicate with any set in the list we use to save the set so far. After that we have a list of the sets of the cells, we need to consider 2 directions, the forward direction → and the reverse direction ← .

    a. The forward direction means if the cell we are working on and k-1 adjacency cells are green, the rest are red.In the forward direction and each set, we have n-k CNF clauses,each clause consists of the negative value of the cell, the negative values of cells in the set, a negative value of the adjacency cell not in the set, we have n-k cells not in the set, so we have n-k clauses, for instance, the cell 2 contains value 2, and the adjacency cells of the cell 2 are 1,3,4,5, the set has the cell 1, then we have multiple clauses (-2 v -1 v -3)^ (-2 v -1 v -4)^(-2v-1v-5) which means if the cell 2 and the adjacency cells 1 are green, and the rest are red.

    b. The reverse direction means if some adjacency cells are red, the cell we are working on and k-1 adjacency cells are green.In the reverse direction and each set in the list we have had so far, We have k clauses divided into 2 types.The first type consists of the positive value of the cell and the positive values of its adjacency cells which are not in the set. The second type consists of the positive values of the adjacency cells which are not in the set and a positive value of a cell in the set,we have k-1 cells in the set so we have k-1 clauses for this type. For instance, the cell 2 contains value 2, and the adjacency cells of the cell 2 are 1,3,4,5, we have a set of 1 which is a set in the list, then we have two clauses (2 v 3 v 4 v 5) ^(1 v 3 v 4 v 5), which means if the cell 3,4,5 are red, the cell 2 and the adjacency cell 1 are green.

2. In the red case, we need to generate C_n^k clauses, which means we need to iterate C_n^k times, each iteration we choose a set of k adjacency cells and make sure the set does not duplicate with any set in the list we use to save the set so far. After that we have a list of the sets of the cells, we need to consider 2 directions, the forward direction → and the reverse direction ← .

    a. The forward direction means if the cell we are working on and k-1 adjacency cells are red, the rest are green.In the forward direction and each set, we have n-k CNF clauses.There are 2 types in n-k clauses.The first type consists of the negative value of the cell, the negative values of cells in the set.The second type consists of the negative values of cells in the set and a negative value of a cell not in the set,we have n-k-1(k is the number of the cells in the set, 1 is the number of the cells we are working on) cells not in the set, so we have n-k-1 clauses for the second type. For instance, the cell 2

contains value 2, and the adjacency cells of the cell 2 are 1,3,4,5, the set has the cell 1 and the cell 3, then we have multiple clauses (-2 v -1 v -3)^ (-3 v -1 v -4)^(-3v-1v-5) which means if the cell 1 and the adjacency cells 3 are green, and the rest are red.

b. The reverse direction means if n-k-1 adjacency cells and the cell we are working on are red,the rest which are k cells in green.In the reverse direction and each set in the list we have had so far, We have k clauses.Each clause consists of a positive value of a cell in the set, the positive values of its adjacency cells which are not in the set, the positive value of the sell.,we have k cells in the set so we have k clauses. For instance, the cell 2 contains value 2, and the adjacency cells of the cell 2 are 1,3,4,5, we have a set of 1 and 3 which is a set in the list, then we have two clauses (2 v 3 v 4 v 5) ^(1 v 2 v 4 v 5), which means if the cell 1 and the cell 3 are green, the rest is red.

If the value is zero, we just need to generate clauses for the case which is the cell in red color. *For example,* we have cell 2 which contains value 0, and we have adjacency cells 1,3,4,5 so that we have multiple clauses such as -2^-1^-3^-4^-5. If the value is 1 and the cell is green, we just have multiple clauses that satisfy the condition which is (value in cell) ^ -[each value in adjacency cells]. If the value is 1 and the cell is red, back to step 2. If the value of the cell is greater than the number of the adjacency cells, we do not consider the red case. If the value of the cell subtracts the number of the adjacency cells greater than 1, we do not consider the green case.

2.2 A* algorithm

Step 1: Initialize a matrix S of size m x n with all elements equal to 1 (Blue). The returned result is the matrix S. Let C[i,j] be the sum of adjacent blue cells of cell (i,j) in S. A is the destination matrix (input).

Step 2: Check whether the matrix S can still be colored red by:

- ∀ i, j (0 <= i, j < m, n), S[i,j] ≠ A[i,j], S[i,j] or one of its adjacent cells possible colored red.

  S[x,y] can be colored red, that is: When red in the cell (x,y), then C[x,y]<= A[x,y] and C[tx,ty] <= A[tx,ty] where tx, ty are row and column indexes of cells adjacent to cell (x,y).

If the matrix S can no longer be colored red, the message "NO SOLUTION" and the end.

Otherwise, continue to step 3.

Step 3: If C = A (C[i,j]=A[i,j] for all 0 <= i, j < m, n), the result is matrix S. Otherwise, continue to step 4.

Step 4:

i. Considering each cell (i,j), S[i,j] = 1 (Blue), calculate the heuristic value:

Assuming S[i,j] = 0 (Red), update the matrix S', C', then:

- h_1 = *Total number of cells (x,y) with C'[x,y] ≠ A[x,y] (Number of cells not completed).*
- h_2 *will be calculated as follows:*
  - *If the matrix S' can no longer be colored, then* $h_2 = \infty$.
  - *Otherwise,* $h_2 = C'[x,y] - A[x,y]$.

Heuristic of cell (i,j) will be: $H[i,j] = h_1[i,j] + h_2[i,j]$.

ii. Find the smallest H(x,y), assign S[x,y] = 0 (Red), update the matrix C.

iii. Go back to step 2.

Example :

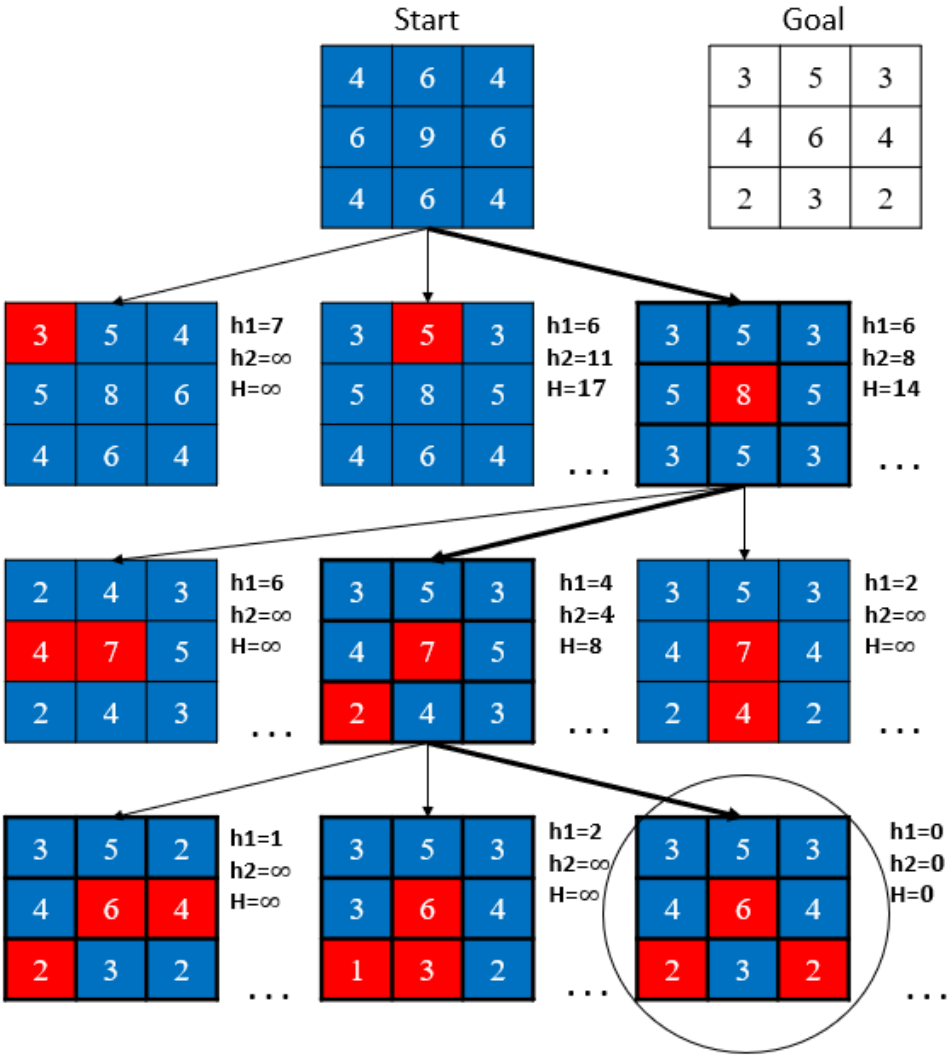
Input :

3 3

3 5 3

4 6 4

2 3 2



2.3 Brute-Force algorithm

a. Idea: Recursing through all positions which have a number of adjacent cells larger than -1. Base case of recursion is equality between size of POSITIONS and size of path.

b. Variables:

i. POSITIONS: list of positions have number of adjacent cell larger than -1.
ii. path: list of marked satisfied positions.
iii. COLORED_BOARD: result

c. Algorithm:

i. Check whether this path in base case, if false go to step iii.
ii. Check whether COLORED_BOARD satisfy all positions in path.
iii. Choose a position in POSITIONS which is not in path.
iv. Append this position into path.
v. Generate all combinations of colored adjacent cells which satisfy this cell number.
vi. Loop through all combinations:
    1. Coloring each cell of combination
    2. Back to step i with current path (recursion)
    3. If result = True, return True
    4. Make each cell of combination colorless
vii. Move this position out of path
viii. Return False

d. Example:

Input:
2 2
3 3
3 3

1. POSITIONS = [[0,0], [0,1], [1,0], [1,1]]
2. Path = []
3. Size of path is 0
4. pos = [0,0] (not in path)
5. combinations = (
    [[0,0], [0,1], [1,0]],
    [[0,0], [0,1], [1,1]],
    [[0,0], [1,0], [1,1]],
    [[0,1], [1,0], [1,1]],...
)
6. choose [[0,0], [0,1], [1,0]] and color all position in this combination.
7. path = [[0,0]]
8. pos = [0,1] (not in path).
9. combinations = () (because all adjacent cells are satisfied).
10. path = [[0,0], [0,1]]
11. pos = [1,0] (not in path).
12. combinations = () (because all adjacent cells are satisfied).
13. path = [[0,0], [0,1] , [1,0]]
14. pos = [1,1] (not in path)
15. combinations = () (because all adjacent cells are satisfied).
16. path = [[0,0], [0,1], [1,0], [1,1]]

17. size of path == size of POSITIONS
18. all positions in path are satisfied => result = [[“B”,”B”],[“B”,”R”]]

2.4 Backtracking algorithm

Step 1: Initialize a matrix C of size m x n with all elements being 0 (Red). C is the result matrix. Let H[i,j] be the sum of adjacent blue cells of cell (i,j) in C. A is the target matrix (input).

Step 2: Consider each element A[i,j] (0 <= i, j < m, n) in the matrix:

- If the element A[i,j] ≠ -1 and H[i,j] < A[i,j], then select cell (i,j) as the starting point for backtracking.

  Backtracking(x,y):

  i. If H = A (H[i,j]=A[i,j] for all 0 <= i, j < m, n) then stop backtracking.

  ii. Consider the adjacent (tx,ty) cells of (x,y):

  - If C[tx,ty] = 0 (red) and the cell [tx,ty] can be colored green (when it is colored blue, the H of its adjacent cells is still less than or equal to a[i,j]) then backtracking:

    C[tx,ty] = 1 (Blue)

    Backtracking(tx,ty)

    C[tx,ty] = 0 (red)

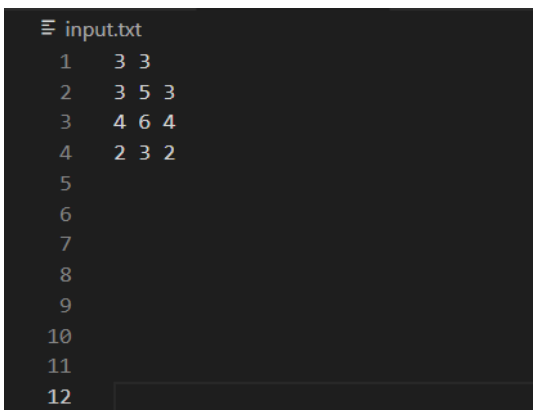
- If H = A (H[i,j]=A[i,j] for all 0 <= i, j < m, n) then return the result.

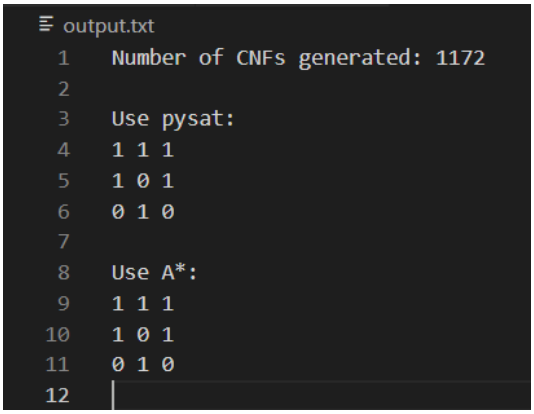
3. EXPERIMENT

3.1 Generate CNFs and Use PySAT to solve CNFs

The program reads the input.txt file and outputs the results to the output.txt file (see illustrations):

The results of the A* algorithm will be output together.





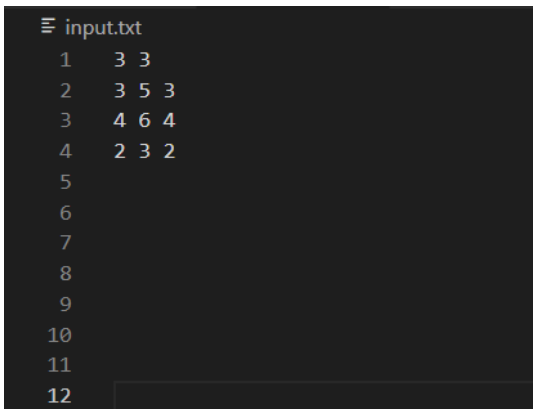
No.	Test case	Experimental results	Evaluation
1	3 3 3 5 3 4 6 4 2 3 2	Number of CNFs generated: 1172 Use pysat: 1 1 1 1 0 1 0 1 0	Accepted
2	5 5 -1 5 6 -1 3 5 7 8 7 -1 4 6 8 7 5 5 6 7 7 -1 -1 4 5 5 -1	Number of CNFs generated: 3712 Use pysat: 1 1 1 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 1 0 1 1	Accepted
3	7 7 -1 5 6 -1 3 2 -1 5 7 8 7 -1 3 1 4 6 8 7 5 2 -1 5 6 7 7 -1 3 0 -1 4 5 5 -1 2 0 3 3 2 2 2 2 -1 -1 1 0 0 -1 1 1	Number of CNFs generated: 12724 Use pysat: 1 1 1 1 1 0 1 0 1 1 1 0 0 0 1 1 0 1 1 0 0 0 1 1 1 1 0 0 1 1 0 1 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1	Accepted
4	10 10 -1 2 3 -1 -1 0 -1 -1 -1 -1 -1 -1 -1 -1 3 -1 2 -1 -1 6 -1 -1 5 -1 5 3 -1 5 7 4 -1 4 -1 5 -1 5 -1 6 -1 3 -1 -1 4 -1 5 -1 6 -1 -1 3 -1 -1 -1 2 -1 5 -1 -1 -1 -1 4 -1 1 -1 -1 -1 1 1 -1 -1 4 -1 1 -1 -1 -1 1 -1 4 -1 -1 -1 -1 -1 6 -1 -1 -1 -1 4 -1 4 4 -1 -1 -1 -1 4 -1 -1	Number of CNFs generated: 17589 Use pysat: 0 1 1 0 0 0 0 0 1 1 0 0 0 1 0 0 0 1 1 1 0 0 1 1 1 0 0 1 1 1 0 1 1 0 1 1 0 1 0 0 0 1 0 0 0 1 1 1 1 0 1 1 0 0 1 1 0 0 1 1 1 0 0 0 1 0 0 0 0 1 1 0 0 0 1 0 0 0 0 1 1 1 0 0 1 1 0 0 1 1 0 1 1 1 1 1 1 1 1 0	Accepted
5	15 15 -1 -1 -1 5 5 -1 5 -1 -1 -1 5 -1 6 -1 4 -1 8 -1 -1 -1 7 -1 7 -1 -1 7 -1 -1 -1 5 5 -1 -1 6 -1 -1 8 -1 8 -1 -1 8 -1 7 -1 -1 -1 7 -1 -1 -1 -1 -1 5 -1 6 -1 -1 -1 -1 -1 6 7 -1 -1 -1 -1 -1 -1 -1 -1 8 -1 5 2 4 -1 -1 -1 8 -1 -1 -1 3 -1 6 -1 -1 -1 3 -1 -1 -1 6 -1 6 -1 -1 -1 -1 -1 -1 -1 4 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 7 -1 5 -1 5 -1 5 -1 -1 4 -1 -1 -1 4 4 -1 -1 -1 6 -1 -1 -1 5 -1 -1 -1 5 -1 -1	Number of CNFs generated: 26698 Use pysat: 0 1 1 1 0 1 0 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 0 1 0 1 1 1 0 0 1 0 1 0 0 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 0 0 0 0 0 1 1 1 0 1 1 0 1 1 1 0 1 0 1 1 1 1 0 0 0 0 0 1 1 0 0 0 0 1 1 0 0 1 1	Accepted

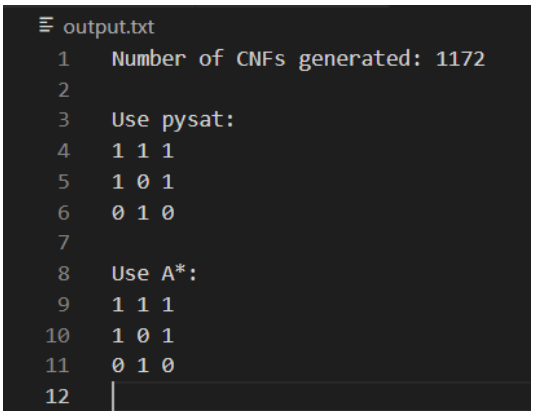
	-1 -1 7 -1 -1 -1 -1 -1 5 -1 -1 -1 -1 -1 -1 -1 6 -1 8 -1 7 -1 -1 6 -1 -1 6 -1 -1 -1 -1 6 -1 -1 -1 -1 -1 8 -1 -1 -1 8 -1 7 -1 -1 -1 -1 -1 5 -1 -1 -1 8 -1 -1 -1 -1 -1 -1 -1 -1 -1 4 -1 3 -1 -1 5 -1 4 -1 5 -1 2	1 0 1 0 1 1 1 0 0 1 1 0 0 1 0 1 1 0 1 1 1 0 1 1 0 0 1 0 1 1 1 1 1 0 0 0 0 1 1 0 0 1 1 1 0 0 0 1 1 1 1 1 1 0 0 1 0 0 0 0 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 0 1 1 0 0 1 1 1 1 0 1 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 0 0 0 0 1 1 0 0 1 1 0 0	

Comment: Using A* algorithm to solve CNFs is really effective. All cases run very fast and give correct results. The running time depends on the size of the matrix.

3.2 Apply A* to solve CNFs

The program reads the input.txt file and outputs the results to the output.txt file (see illustrations):





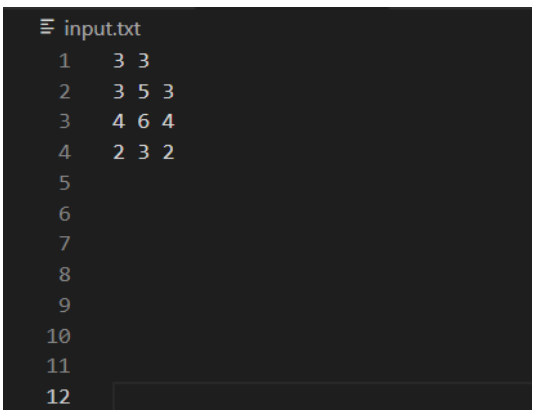
No.	Test case	Experimental results	Execution time (s)	Evaluation
1	3 3 3 5 3 4 6 4 2 3 2	1 1 1 1 0 1 0 1 0	0s	Accepted
2	5 5 -1 5 6 -1 3 5 7 8 7 -1 4 6 8 7 5 5 6 7 7 -1 -1 4 5 5 -1	1 1 1 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 1 0 1 1	0.036s	Accepted
3	7 7 -1 5 6 -1 3 2 -1 5 7 8 7 -1 3 1 4 6 8 7 5 2 -1 5 6 7 7 -1 3 0 -1 4 5 5 -1 2 0 3 3 2 2 2 2 -1 -1 1 0 0 -1 1 1	1 1 1 1 1 0 1 0 1 1 1 0 0 0 1 1 0 1 1 0 0 0 1 1 1 1 0 0 1 1 0 1 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1	0.622s	Accepted
4	10 10 -1 2 3 -1 -1 0 -1 -1 -1 -1	NO SOLUTION	4.533s	Wrong answer

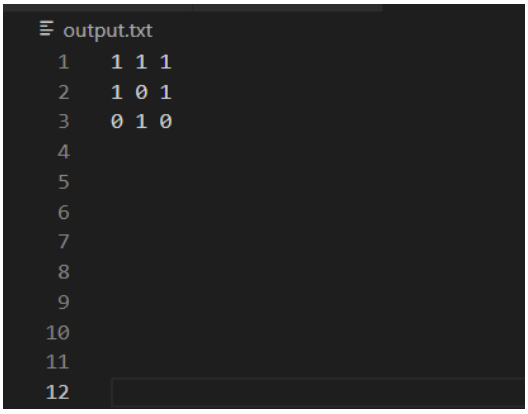
	-1 -1 -1 -1 3 -1 2 -1 -1 6 -1 -1 5 -1 5 3 -1 5 7 4 -1 4 -1 5 -1 5 -1 6 -1 3 -1 -1 4 -1 5 -1 6 -1 -1 3 -1 -1 -1 2 -1 5 -1 -1 -1 -1 4 -1 1 -1 -1 -1 1 1 -1 -1 4 -1 1 -1 -1 -1 1 -1 4 -1 -1 -1 -1 -1 6 -1 -1 -1 -1 4 -1 4 4 -1 -1 -1 -1 4 -1 -1			
5	15 15 -1 -1 -1 5 5 -1 5 -1 -1 -1 5 -1 6 -1 4 -1 8 -1 -1 -1 7 -1 7 -1 -1 7 -1 -1 -1 5 5 -1 -1 6 -1 -1 8 -1 8 -1 -1 8 -1 7 -1 -1 -1 7 -1 -1 -1 -1 -1 5 -1 6 -1 -1 -1 -1 -1 6 7 -1 -1 -1 -1 -1 -1 -1 -1 8 -1 5 2 4 -1 -1 -1 8 -1 -1 -1 3 -1 6 -1 -1 -1 3 -1 -1 -1 6 -1 6 -1 -1 -1 -1 -1 -1 -1 4 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 7 -1 5 -1 5 -1 5 -1 -1 4 -1 -1 -1 4 4 -1 -1 -1 6 -1 -1 -1 5 -1 -1 -1 5 -1 -1 -1 -1 7 -1 -1 -1 -1 -1 5 -1 -1 -1 -1 -1 -1 -1 6 -1 8 -1 7 -1 -1 6 -1 -1 6 -1 -1 -1 -1 6 -1 -1 -1 -1 -1 8 -1 -1 -1 8 -1 7 -1 -1 -1 -1 -1 5 -1 -1 -1 8 -1 -1 -1 -1 -1 -1 -1 -1 -1 4 -1 3 -1 -1 5 -1 4 -1 5 -1 2	NO SOLUTION	31.15s	Wrong answer

Comment: Using A* algorithm to solve CNFs depends a lot on the heuristic function. If the estimation difference is too much, it will lead to wrong results. Cases 1, 2 and 3 all run very fast and give correct results; cases 4, 5 give wrong results. The running time depends on the size of the matrix. Algorithm A* uses a lot of memory.

3.3 Apply Brute-Force algorithm to solve the problem

The program reads the input.txt file and outputs the results to the output.txt file (see illustrations):





No.	Test case	Experimental results	Execution time (s)	Evaluation

1	3 3 3 5 3 4 6 4 2 3 2	1 1 1 1 0 1 0 1 0	0s	Accepted
2	5 5 -1 5 6 -1 3 5 7 8 7 -1 4 6 8 7 5 5 6 7 7 -1 -1 4 5 5 -1	1 1 1 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 1 0 1 1	0.922s	Accepted
3	7 7 -1 5 6 -1 3 2 -1 5 7 8 7 -1 3 1 4 6 8 7 5 2 -1 5 6 7 7 -1 3 0 -1 4 5 5 -1 2 0 3 3 2 2 2 2 -1 -1 1 0 0 -1 1 1	1 1 1 1 1 0 1 0 1 1 1 0 0 0 1 1 0 1 1 0 0 0 1 1 1 1 0 0 1 1 0 1 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1	135s	Accepted
4	10 10 -1 2 3 -1 -1 0 -1 -1 -1 -1 -1 -1 -1 -1 3 -1 2 -1 -1 6 -1 -1 5 -1 5 3 -1 5 7 4 -1 4 -1 5 -1 5 -1 6 -1 3 -1 -1 4 -1 5 -1 6 -1 -1 3 -1 -1 -1 2 -1 5 -1 -1 -1 -1 4 -1 1 -1 -1 -1 1 1 -1 -1 4 -1 1 -1 -1 -1 1 -1 4 -1 -1 -1 -1 -1 6 -1 -1 -1 -1 4 -1 4 4 -1 -1 -1 -1 4 -1 -1	Run time is too large (> 10 minutes).	>600s	Time exceeds 10 minutes.
5	15 15 -1 -1 -1 5 5 -1 5 -1 -1 -1 5 -1 6 -1 4 -1 8 -1 -1 -1 7 -1 7 -1 -1 7 -1 -1 -1 5 5 -1 -1 6 -1 -1 8 -1 8 -1 -1 8 -1 7 -1 -1 -1 7 -1 -1 -1 -1 -1 5 -1 6 -1 -1 -1 -1 -1 -1 6 7 -1 -1 -1 -1 -1 -1 -1 -1 8 -1 5 2 4 -1 -1 -1 8 -1 -1 -1 3 -1 6 -1 -1 -1 3 -1 -1 -1 6 -1 6 -1 -1 -1 -1 -1 -1 -1 4 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 7 -1 5 -1 5 -1 5 -1 -1 4 -1 -1 -1 4 4 -1 -1 -1 6 -1 -1 -1 5 -1 -1 -1 5 -1 -1 -1 -1 7 -1 -1 -1 -1 -1 5 -1 -1 -1 -1 -1 -1 -1 6 -1 8 -1 7 -1 -1 6 -1 -1 6 -1 -1 -1 -1 6 -1 -1 -1 -1 -1 8 -1 -1 -1 8 -1 7 -1 -1 -1 -1 -1 5 -1 -1 -1 8 -1 -1 -1 -1 -1 -1 -1 -1 -1 4 -1 3 -1 -1 5 -1 4 -1 5 -1 2	Run time is too large (> 10 minutes).	>600s	Time exceeds 10 minutes.

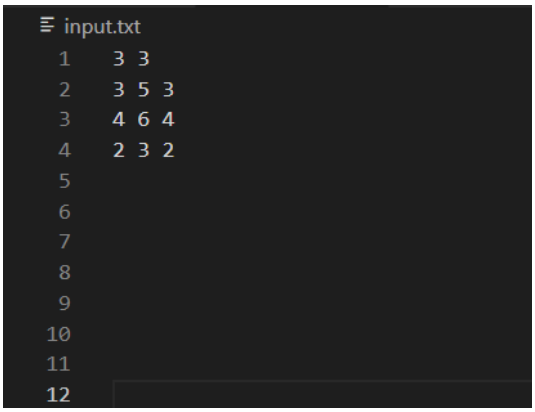
Comment: Using Brute-Force algorithm to solve the problem:

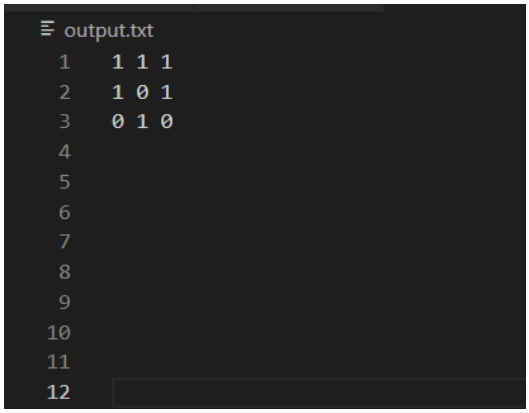
Cases 1, 2 and 3 run relatively fast; Cases 4 and 5 do not result because the running time is too large (>10 minutes). The running time depends on the size of the matrix.

- Advantages: Easy to understand, easy to install, always gives correct results
- Disadvantages: Running time is very long because it has to run all the cases.

3.4 Apply Backtracking algorithm to solve the problem

The program reads the input.txt file and outputs the results to the output.txt file (see illustrations):





No.	Test case	Experimental results	Execution time (s)	Evaluation
1	3 3 3 5 3 4 6 4 2 3 2	1 1 1 1 0 1 0 1 0	0	Accepted
2	5 5 -1 5 6 -1 3 5 7 8 7 -1 4 6 8 7 5 5 6 7 7 -1 -1 4 5 5 -1	1 1 1 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 1 0 1 1	253s	Accepted
3	7 7 -1 5 6 -1 3 2 -1 5 7 8 7 -1 3 1 4 6 8 7 5 2 -1 5 6 7 7 -1 3 0 -1 4 5 5 -1 2 0 3 3 2 2 2 2 -1 -1 1 0 0 -1 1 1	Run time is too large (> 10 minutes).	>600s	Time exceeds 10 minutes.
4	10 10 -1 2 3 -1 -1 0 -1 -1 -1 -1 -1 -1 -1 -1 3 -1 2 -1 -1 6 -1 -1 5 -1 5 3 -1 5 7 4 -1 4 -1 5 -1 5 -1 6 -1 3 -1 -1 4 -1 5 -1 6 -1 -1 3 -1 -1 -1 2 -1 5 -1 -1 -1 -1 4 -1 1 -1 -1 -1 1 1 -1 -1 4 -1 1 -1 -1 -1 1 -1 4 -1 -1 -1 -1 -1 6 -1 -1 -1 -1 4 -1 4 4 -1 -1 -1 -1 4 -1 -1	Run time is too large (> 10 minutes).	>600s	Time exceeds 10 minutes.
5	15 15 -1 -1 -1 5 5 -1 5 -1 -1 -1 5 -1 6 -1 4 -1 8 -1 -1 -1 7 -1 7 -1 -1 7 -1 -1 -1 5 5 -1 -1 6 -1 -1 8 -1 8 -1 -1 8 -1 7 -1 -1 -1 7 -1 -1 -1 -1 -1 5 -1 6 -1 -1 -1 -1	Run time is too large (> 10 minutes).	>600s	Time exceeds 10 minutes.

	-1 6 7 -1 -1 -1 -1 -1 -1 -1 -1 8 -1 5 2 4 -1 -1 -1 8 -1 -1 -1 3 -1 6 -1 -1 -1 3 -1 -1 -1 6 -1 6 -1 -1 -1 -1 -1 -1 -1 4 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 7 -1 5 -1 5 -1 5 -1 -1 4 -1 -1 -1 4 4 -1 -1 -1 6 -1 -1 -1 5 -1 -1 -1 5 -1 -1 -1 -1 7 -1 -1 -1 -1 -1 5 -1 -1 -1 -1 -1 -1 -1 6 -1 8 -1 7 -1 -1 6 -1 -1 6 -1 -1 -1 -1 6 -1 -1 -1 -1 -1 8 -1 -1 -1 8 -1 7 -1 -1 -1 -1 -1 5 -1 -1 -1 8 -1 -1 -1 -1 -1 -1 -1 -1 -1 4 -1 3 -1 -1 5 -1 4 -1 5 -1 2			

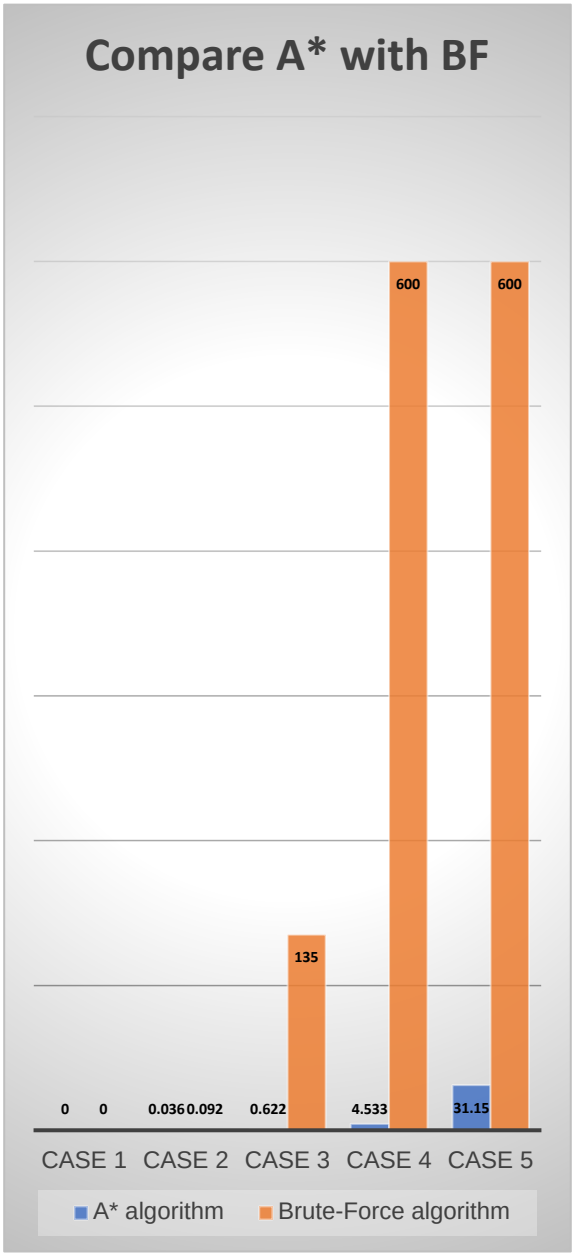
Comment: Using Backtracking algorithm to solve the problem:

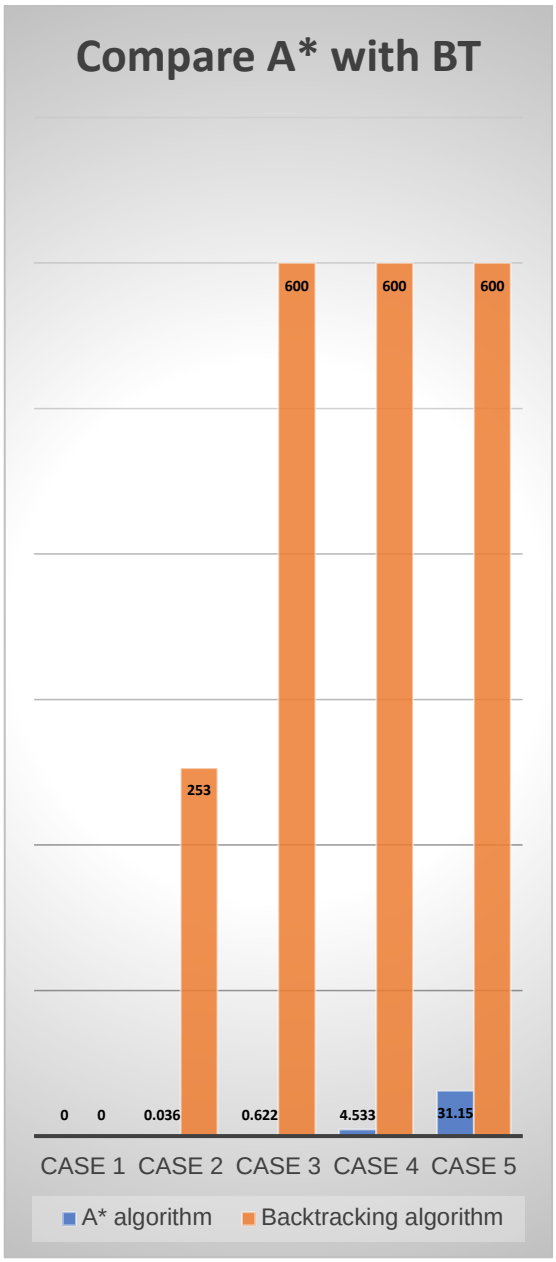
Cases 1, 2 run relatively fast; Cases 3, 4 and 5 do not result because the running time is too large (>10 minutes). The running time depends on the size of the matrix.

- Advantages: Easy to understand, easy to install, always gives correct results
- Disadvantages: Running time is very long because it has to run all the cases.

3.5 Compare A* algorithm with Brute-Force algorithm and Backtracking algorithm

No.	A* algorithm	Brute-Force algorithm	Backtracking algorithm
1	0s (Accepted)	0s (Accepted)	0s (Accepted)
2	0.036s (Accepted)	0.922s (Accepted)	253s (Accepted)
3	0.622s (Accepted)	135s (Accepted)	>600s
4	4.533s (Wrong answer)	>600s (Time exceeds 10 minutes)	>600s (Time exceeds 10 minutes)
5	31.15s (Wrong answer)	>600s (Time exceeds 10 minutes)	>600s (Time exceeds 10 minutes)





The chart compares the speed of the A algorithm with the BF algorithm and the BT algorithm.*

4. CONCLUSION

- Algorithm A* is an optimization algorithm that uses heuristic evaluation to find the next move. So, the A* algorithm is heavily dependent on the heuristic function. If the difference in heuristic estimates is large enough, it will lead to false results. The disadvantage of the A* algorithm is that it consumes a lot of memory.
- Algorithms Brute-Force and Backtracking applied in this problem will always have the right results, but their disadvantage is that the running time is very long, depending on the size of the matrix.
- We are able to use CNF clause with pysat module of Python to solve satisfied problems.

5. REFERENCES

- https://pysathq.github.io/?fbclid=IwAR1S52z5wXhfmNgNAOj4dn2DqJRgQRGMDjEHBFcHfwm_ZiBvqGcC1yBZInY
- https://github.com/Liuyubao/PL-Resolution-and-cnf?fbclid=IwAR3wNhAVWqRdUzAAOY36T1DjAdJOHUbwhw_53NFHfrYaPSQDU56XmKXxJNg
- https://en.wikipedia.org/wiki/A*_search_algorithm