

Project 02 - Report

# **SAT approach for 8-queens**

---



## Content:

1. General information
  - a. Group member
  - b. Responsibilities
  - c. Self-assessment
2. Task a
3. Task b
4. Task c
5. Task d
6. Task e
7. Task f

## General information

### Group member

1. 19125007 - Lê Thảo Huyền
2. 19125033 - Nguyễn Ngọc Băng Tâm
3. 19125075 - Lâm Bích Vân

### Responsibilities

Lê Thảo Huyền	task c, task d, write report
Nguyễn Ngọc Băng Tâm	task e, task f
Lâm Bích Vân	task a, task b, prepare the report

### Self-assessment

No.	Criteria	Scores	Self-assessment
1	Task a	10%	10/10
2	Task b	10%	10/10
3	Task c includes Level 1: Level 2:	(20%) 10% 10%	20/20
4	Task d	20%	20/20
5	Task e	20%	20/20
6	Task f	10%	10/10
7	Comply with the regulations of submission requirements	10%	10/10
Total		100%	100

## Task a

### The input(s) and output(s) of the problem

According to the documentation, the input state of the problem is an empty chessboard (since no pieces have been placed) and the output state will be when 8 pieces are placed without any violations. , without any queen attacking other queens vertically, horizontally, diagonally (goal states)

### The data structures that represent variables and any state of the program

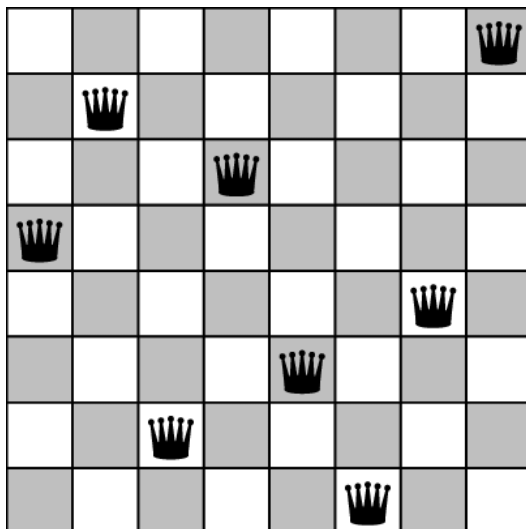
We number the board from 0 to 7 from right to left to represent the horizontal coordinates of each queen. Similarly, we number 0 to 7 from top to bottom to represent vertical coordinates

Each square on the chessboard will own a set of 3 variables (x,y,b), in which:

- x represents the horizontal coordinates of the square on the chessboard, for example the cell in the 3rd column will own  $x=3$
- y represents the vertical coordinates of the square on the chessboard, for example the square in the 4th row will have  $y=4$
- b represents the state of the checkerboard. If the chess piece has a queen  $b=1$ , if the chess piece has no queen  $b=0$

For example, the chessboard (i) will be represented by (ii)

(i)



(ii)

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

Correspondingly, demonstrates how the initial and goal states will look like.

**Initial state:** Initial state will be an empty chessboard

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

**Goal states:** The 8 queens problem has 92 goal states. However, if we don't consider the possible overlapping goal states due to the rotation of a goal state by 90, 180, or 270 degrees, we basically have 12 essentially independent goal states and cannot form each other.

1.		0	1	2	3	4	5	6	7		2.		0	1	2	3	4	5	6	7
0	0	0	0	1	0	0	0	0	0		0	0	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	0	1	0		1	0	1	0	0	0	0	0	0	0
2	0	0	1	0	0	0	0	0	0		2	0	0	0	1	0	0	0	0	0
3	0	0	0	0	0	0	0	0	1		3	0	0	0	0	0	0	1	0	0
4	0	1	0	0	0	0	0	0	0		4	0	0	1	0	0	0	0	0	0
5	0	0	0	0	1	0	0	0	0		5	0	0	0	0	0	0	0	0	1
6	1	0	0	0	0	0	0	0	0		6	0	0	0	0	0	1	0	0	0
7	0	0	0	0	0	1	0	0	0		7	1	0	0	0	0	0	0	0	0

3.		0	1	2	3	4	5	6	7		4.		0	1	2	3	4	5	6	7
0	0	0	0	1	0	0	0	0	0		0	0	0	0	1	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0		1	0	0	0	0	0	1	0	0	0
2	0	0	0	0	0	0	0	1	0		2	0	0	0	0	0	0	0	0	1
3	0	0	1	0	0	0	0	0	0		3	0	0	1	0	0	0	0	0	0
4	0	0	0	0	0	1	0	0	0		4	1	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	1		5	0	0	0	0	0	0	1	0	0
6	0	0	0	0	1	0	0	0	0		6	0	0	0	0	1	0	0	0	0
7	1	0	0	0	0	0	0	0	0		7	0	1	0	0	0	0	0	0	0



5.

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

6.

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

7.

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

8.

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



9.

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

10.

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

11.

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

12.

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



## Task b

We assign position to save the order of each cell

position =  $y \cdot 8 + x + 1$

	0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7	8
1	9	10	11	12	13	14	15	16
2	17	18	19	20	21	22	23	24
3	25	26	27	28	29	30	31	32
4	33	34	35	36	37	38	39	40
5	41	42	43	44	45	46	47	48
6	49	50	51	52	53	54	55	56
7	57	58	59	60	61	62	63	64

CNF for cell 28 ([3][3]):

**Queens cannot lie on the same diagonal.**

Diagonal 1

-28 v -1                    (-b[3][3] v -b[0][0])

-28 v -10                    (-b[3][3] v -b[1][1])

-28 v -19                    (-b[3][3] v -b[2][2])

-28 v -37                    (-b[3][3] v -b[4][4])

-28 v -46                    (-b[3][3] v -b[5][5])

-28 v -55                    (-b[3][3] v -b[6][6])

-28 v -64                    (-b[3][3] v -b[7][7])

## Diagonal 2

-28 v -7	$(-b[3][3] \vee -b[6][0])$
-28 v -14	$(-b[3][3] \vee -b[5][1])$
-28 v -21	$(-b[3][3] \vee -b[4][2])$
-28 v -35	$(-b[3][3] \vee -b[2][4])$
-28 v -42	$(-b[3][3] \vee -b[1][5])$
-28 v -49	$(-b[3][3] \vee -b[0][6])$

### Queens cannot lie on the same row.

-28 v -25	$(-b[3][3] \vee -b[0][3])$
-28 v -26	$(-b[3][3] \vee -b[1][3])$
-28 v -27	$(-b[3][3] \vee -b[2][3])$
-28 v -29	$(-b[3][3] \vee -b[4][3])$
-28 v -30	$(-b[3][3] \vee -b[5][3])$
-28 v -31	$(-b[3][3] \vee -b[6][3])$
-28 v -32	$(-b[3][3] \vee -b[7][3])$

### Queens cannot lie on the same column.

-28 v -4	$(-b[3][3] \vee -b[3][0])$
-28 v -12	$(-b[3][3] \vee -b[3][1])$
-28 v -20	$(-b[3][3] \vee -b[3][2])$
-28 v -36	$(-b[3][3] \vee -b[3][4])$
-28 v -44	$(-b[3][3] \vee -b[3][5])$
-28 v -52	$(-b[3][3] \vee -b[3][6])$
-28 v -60	$(-b[3][3] \vee -b[3][7])$

## Task c

Level 1: Each queen can only move on a single column

At this level to generate all CNF clauses for the whole board we need to take into consideration the following constraints:

1. Each column has one queen (e.g first column must have one queen:  $1 \vee 9 \vee 17 \vee 25 \vee 33 \vee 41 \vee 49 \vee 57$ ).
2. Each pair of cells that are on the same row cannot both have queens (both true) (e.g  $-2, -1 \vee -3$ ).
3. Each pair of cells that are on the same diagonal cannot both have queens (both true) (e.g  $-1 \vee -10$ )

We do not need to generate the CNFs for the constraint of “each pair of queens that are on the same column cannot be both true” since constraints 1 and 2 have already ensured that. This is because as every column has at least one queen (from constraint 1) if one column has more than two queens, one of them is definitely on the same row with some queens in other columns, which violates constraint 2.

Level 2: The position of the queens is any cell on the chessboard

At this level, we need to take into consideration the constraints 2 and 3 above plus two following constraints:

4. There are some cells that have queens (i.e  $1 \vee 2 \vee 3 \vee 4 \vee 5 \vee \dots \vee 64$ ).
5. Each pair of cells that are on the same column cannot both have queens (both true) (e.g  $-1 \vee -9$ ).

Since the queens can be on any cell we have constraint 4. However, this constraint will lead to some solutions that have less than 8 satisfying positions. Thus when receiving all solutions from PySat Solver we need to choose the ones that have exactly 8 positive values.

Furthermore, when generating CNFs for constraint 2, 3, 5, for each cell we just need to make a pair of it and another cell (on the same row, column, or diagonal) that have a larger assigned value (i.e  $v = 8*i + j + 1$ ) so that we will not redundantly include the same CNFs in our clauses set. For example, there is not  $(-2 \vee -1)$  which is the same as  $(-1 \vee -2)$  in our clauses.

## Task d

In this task, the clause set generated in task c is passed into PySat Solver. To receive all satisfying solutions we call the `enum_models()` method. Each solution is a list of positive values which represent satisfying positions. And as mentioned above, in level 2, we additionally need to check whether a solution has enough 8 satisfying values.

Here is the function for solving the set of clauses of both levels:

```
def print_solutions(self):  
    with Solver(bootstrap_with=self.clauses) as s:  
        for m in s.enum_models():  
            res = []  
            for v in m:  
                if v > 0:  
                    res.append(v)  
            if len(res) == self.N: # needed for level 2  
                print(res)
```

A demonstration of this task: <https://www.youtube.com/watch?v=sDhq1DwhEQw>.

## Task e

In this task, we view the problem of placing 8 queens on the chessboard as finding an assignment of all variables that satisfies all CNF clauses (which are the clauses that were generated in the Level 2 of task c).


At the beginning, we have  $m$  non-attacking queens, i.e we have  $m$  variables assigned with value True. In this solution, we do not allow overwriting these variables. Hence, there is still the case when we cannot find a solution because of the original  $m$  queens.

A **state** includes the following attributes:

- `true_variables`: A list of variables that have been assigned with value True (i.e the queens that have been placed on the chessboard); all remaining variables that are not in this list are assigned with value False. In all children, grandchildren, etc. states of this state, these True variables are not assigned back to False. We make sure that all there are no CNF clauses involving these True variables are False, or in other words, a state will never have attacking queens.
- `true_cnfs`: The CNF clauses that have already been evaluated to True, thanks to the current assignment. This can be derived by checking each variable in `true_variables` against all CNF clauses and finding out if the clause is True.
- `false_cnfs`: The CNF clauses that are currently evaluated as False. This can be derived by removing clauses from `true_cnfs` out of the list of all CNF clauses (generated from task c).
- `available_variables`: A list of variables that are currently assigned as False, but can be assigned as True without violating any clause in `true_cnfs`. In other words, these represent the cells that we could place a queen on without attacking other existing queens. This list can be derived by iterating through all variables not in `true_variables` and checking if it does not violate any clause in `true_cnfs`.

To summarize, the actual distinction between states solely relies on `true_variables` - the assignment of variables. The rest can be derived from this list and the list of all CNF clauses.

The **initial state** would have `true_variables` be the list representing  $m$  queens already placed on the board. The rest attributes can be derived as described above.



The **goal state** would have `true_variables` be a list of exactly 8 elements. We might note that in this state, the list `false_cnfs` is empty, and so is the list of `available_variables` (since we do not allow a state with attacking queens, once we've placed all 8 queens, all CNF clauses will be satisfied and no variables are available).

The **successors** of the current state can be determined by iterating through each variable in `available_variables`. For each variable, we generate a successor state with the variable assigned to be `True`. We also derived the new `true_cnfs`, `false_cnfs` and `available_variables` for the new state.

The **cost** of each transition from one state to another is 1 (placing one queen).

The **heuristic** of a state is calculated using the formula below:

```
heuristic = 8 - size(true_variables)
```

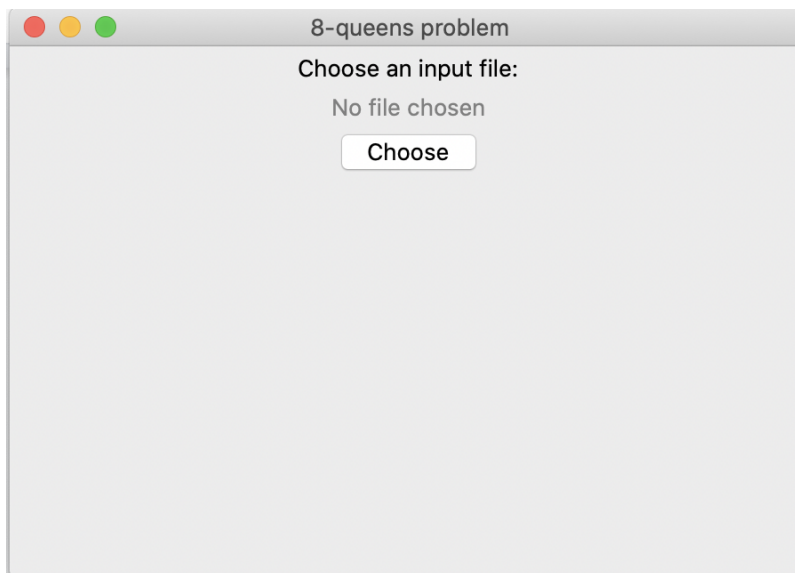
This heuristic is basically the sum of remaining queens to place. A state is considered to be close to the goal state if it has few more queens to place. This is based on the observation that at the goal state, the number of queens left is 0. Using this heuristic, when we arrive at the goal state, the heuristic would be  $8 - 8 = 0$ .

Note: We decided to use this heuristic after examining a few different heuristics, including sum of the remaining queens to place and the number of available cells. However, through experimentation, this heuristic finds the solution in the relatively fastest time.

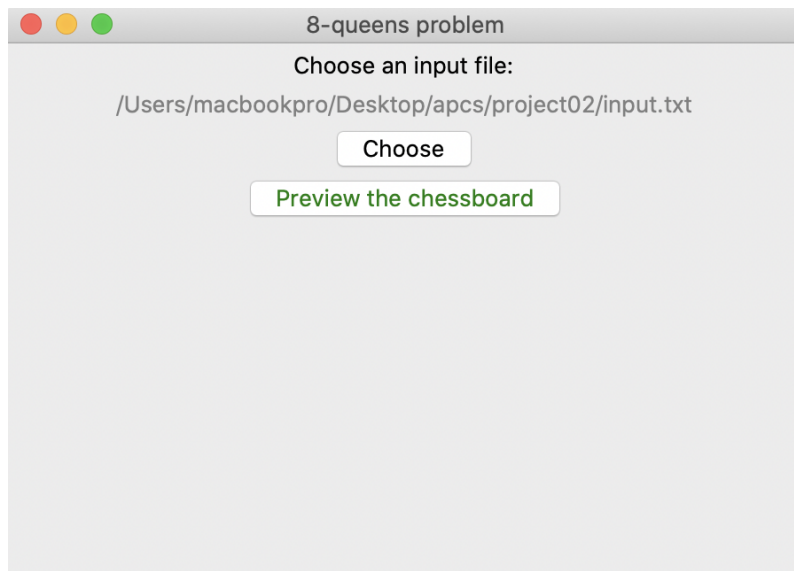
## Task f

To run the program and visualize the steps of finding the solution, we use 2 libraries: Tkinter and Matplotlib. Hence, before running the program, all packages in requirements.txt must be installed (could be in a virtual environment). Besides, the package python-tk and pysat should be installed on the computer for the program to work.

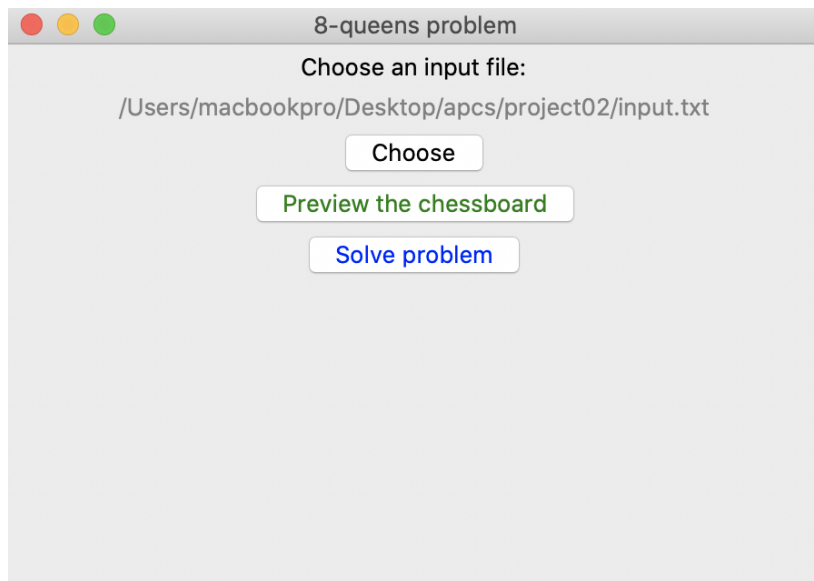
To use the application, run file main.py, then click the button “Choose” to choose a file.



After that, click “Preview” to view the chessboard.



Do not close the figure, then click "Solve":



A demonstration can be found [here](https://youtu.be/ESwpfBj8pb0). (<https://youtu.be/ESwpfBj8pb0>)