HO CHI MINH UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION AND TECHNOLOGY

Course: Introduction to Artificial Intelligence

PROJECT 2

SAT APPROACH
FOR 8-QUEENS PROBLEM

Class: 20CLC11

Instructors: Nguyễn Ngọc Thảo

Lê Ngọc Thành

Nguyễn Thành An

Hồ Thị Thanh Tuyến

Table of Contents

I. Team members' information

Student ID	Name
20127610	Trương Samuel
20127285	Trần Hồng Minh Phúc
20127001	Hà Quốc Anh

II. Self-assessment of each member

Student ID	Assigned tasks	Completion level
20127001	Task a Task f	100%
20127285	Task b Task e	100%
20127610	Task c Task d	100%

III. Check list

No.	Criteria	Check list	Note
1	Task a	✓	
2	Task b	✓	
3	Task c: Level 1	✓	
4	Task c: Level 2	✓	
5	Task d	✓	
6	Task e	✓	
7	Task f	✓	Don't have the visualization of the state at each step of the search path and don't create CNF text file.
8	Report + Video demo	✓	

IV. Task a

- The input(s) and output(s) of the problem:
  - Input: size of the board (8x8)
  - Output: 8 queens are placed on the board so each pair of them doesn't attack each other.
- The data structures that represent variables and any state of the program:
  - A 2D array(matrix) to represent the board. Cells with value 1 are where queens are placed; cells with value 0 are the valid position where queens can be placed on; and cells with value -1 are the invalid position that the queens can attack each other.
- The initial state is the empty board. The goal state is the board has 8 queens so that each pair of them doesn't attack each other.

V. Task b

(Constraints for row)

b[3][0] ∨ b[3][1] ∨ b[3][2] ∨ b[3][3] ∨ b[3][4] ∨ b[3][5] ∨ b[3][6] ∨ b[3][7] ∧

-b[3][0] ∨ -b[3][3] ∧

-b[3][1] ∨ -b[3][3] ∧

-b[3][2] ∨ -b[3][3] ∧

-b[3][3] ∨ -b[3][4] ∧

-b[3][3] ∨ -b[3][5] ∧

-b[3][3] ∨ -b[3][6] ∧

-b[3][3] ∨ -b[3][7] ∧

(Constraints for column)

b[0][3] ∨ b[1][3] ∨ b[2][3] ∨ b[3][3] ∨ b[4][3] ∨ b[5][3] ∨ b[6][3] ∨ b[7][3] ∧

-b[0][3] ∨ -b[3][3] ∧

-b[1][3] ∨ -b[3][3] ∧

-b[2][3] ∨ -b[3][3] ∧

-b[3][3] ∨ -b[4][3] ∧

-b[3][3] ∨ -b[5][3] ∧

-b[3][3] ∨ -b[6][3] ∧

-b[3][3] ∨ -b[7][3] ∧

(Constraints for primary diagonal)

-b[0][0] ∨ -b[3][3] ∧

-b[1][1] ∨ -b[3][3] ∧

-b[2][2] ∨ -b[3][3] ∧

-b[3][3] ∨ -b[4][4] ∧

-b[3][3] ∨ -b[5][5] ∧

-b[3][3] ∨ -b[6][6] ∧

-b[3][3] ∨ -b[7][7] ∧

(Constraints for secondary diagonal)

-b[6][0] ∨ -b[3][3] ∧

-b[5][1] ∨ -b[3][3] ∧

-b[4][2] ∨ -b[3][3] ∧

-b[3][3] ∨ -b[2][4] ∧

-b[3][3] ∨ -b[1][5] ∧

-b[3][3] ∨ -b[0][6] ∧

VI. Task e

- Our heuristic function will return the number of valid position that the queen can placed on.

- After placing the advance queen(s), our algorithm will find the min-row that haven't had any queen yet. Next, we will get succsesor(s) by checking which cell in that row is still able for the next queen standing. After finding all successor(s) in the row, we will add it to a queue. The queue will be sorted in ascending order basing on the heuristic value. Then take the one that have the lowest heuristic value to continue to solve the problem. The loop end when it reaches the goal state, or until the queue is empty.

– Please read the green comment code below:

```
143 # A* algorithm function
144 def A_star(board, queens):
145     queue = []
146     visited = []
147 
148     for item in queens:
149         visited.append(item)
150 
151     queue.append(queens)
152     current = []
153 
154     while len(queue) > 0 :  # Check if the queue is empty or not
155 
156         current = queue.pop(0)  # Get the current state in the queue
157 
158         # Initial the current state and check if is the goal state or not
159         board = initial_state_2(current)
160         if is_goal(board, current):
161             return current
162 
163 
164         # From line 168 to line 177
165         # When we get the index of successor, we will add it to the current state
166         #   (current state will will have the index of queens that we have placed before).
167         # Each valid index we found will be added to queue. The queue will have the list of
168         #   indexes of queens that we can place.
169         successors = get_suitable_cell(board)
170 
171         for item in successors:
172             if item not in visited:
173                 temp = []
174                 visited.append(item)
175                 current.append(item)
176 
177                 for item in current:
178                     temp.append(item)
179 
180                 queue.append(temp)  # The queue saves the index of cell(s) that can place queen(s), but it is
181                                     #   sorted base on the heuristic value after placing all the queen in current.
182                 current.pop()
183         queue = sort_queue(queue)
184 
185     return None
```

– This is the function of finding the successor(s). It will return the list of indexes that the queen can place on that cell.

```
#Find the suitable cell(s) left on the row
def get_suitable_cell(board):
    cells = []
    for i in range(SIZE):
        if sum(board[i]) == 0:
            for j in range(SIZE):
                if is_possible(i,j,board):
                    cells.append((i,j))

            return cells

    return cells
```

– The goal state must be required to satisfy two these conditions: the number of queens equal to the board size and none of them attacking each other.

```python
def is_goal(board, queens):
    return len(queens) == SIZE and is_valid(board)
```

– The function below is to check if our queens have attacked each other or not.

```python
def is_possible(row, col, board):
    #Checking in the same row and column
    for item in range(SIZE):
        if board[item][col]  and item != row :
            return False
        if board[row][item]  and item != col :
            return False

    #Checking 2 diagonals
    for item in range(-SIZE, SIZE + 1):
        if item == 0:
            continue
        x = row + item
        y = col + item
        if x >= 0 and x < SIZE and y >=0 and y < SIZE:
            if board[x][y] :
                return False
        x = row - item
        y = col + item
        if x >= 0 and x < SIZE and y >= 0 and y < SIZE:
            if board[x][y] :
                return False

    return True

#Check if the queens have attack each orthers or not
def is_valid(board):
    for i in range(SIZE):
        for j in range(SIZE):
            if board[i][j] and not is_possible(i,j,board):
                return False
    return True
```

– This is the sorting function, which we have mentioned above.

```
def sort_queue(queue):
    for i in range(len(queue)):
        min_ind = i
        for j in range(len(queue)):
            if heruistic_value(queue[min_ind]) < heruistic_value(queue[j]):
                min_ind = j

            queue[i], queue[min_ind] = queue[min_ind], queue[i]

    return queue
```

– If the while-loop end because of the queue is empty, we will return None to show that there is not any solutions with the advance queen(s).

```
if queens is None:
    print("No solution")
else:
    print("Final State")
    board = initial_state(board,queens)
    print_state(board)
```

- For example, we will place two first queens at (2, 0) and (0, 3).

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

- Basing on our algorithm, it will find the min-row that does not have a queen. In this case is row 1. Then, it will look for the cell that is not attacked. So, the successor will have four elements as four coordinates: (1, 4), (1, 5), (1, 6), (1, 7). Because it is required to add to the current and each element will be added to the queue. Therefore, after adding to the queue, we will have four elements:

    1. [(2,0), (0,3), (1,4)]
    2. [(2,0), (0,3), (1,5)]
    3. [(2,0), (0,3), (1,6)]
    4. [(2,0), (0,3), (1,7)]

- In the next step, we sort it base on the heuristic value and then the queue will change like this:

    1. [(2,0), (0,3), (1,6)]
    2. [(2,0), (0,3), (1,7)]
    3. [(2,0), (0,3), (1,5)]
    4. [(2,0), (0,3), (1,4)]

- This is the heuristic that we have calculated:

h(1,4) = 17
h(1,5) = 16
h(1,6) = 14
h(1,7) = 15

- Now it will check the conditions of the while-loop and we see that the queue is not empty, so we continue to get the current sate. As a result, the current state is [(2,0), (0,3), (1,6)] and we check if it is the goal state or not.

- Finally, continue until the while-loop end.

VII. Evidence videos

[Demo 8Queens Problem | Artificial Intelligence - YouTube]

VIII. References

The N-queens Problem | OR-Tools | Google Developers

Notes on Chapter 7: Logical Agents and Propositional Satisfiability (sfu.ca)

Pygame Mouse Click and Detection - CodersLegacy

How to make a text input box python pygame Code Example (codegrepper.com)

How to Draw a Chessboard in Python/Pygame #Chessboard - YouTube

Artificial Intelligence - 8 Queens SAT solver A* demo - YouTube