CS420 - Project 02
SAT approach for 8-queens problem

Demo link:

- [CS420 - Artificial Intelligence - 8 Queens SAT solver A* demo - YouTube]
- [CS420 - Artificial Intelligence - 8 Queens CNF Generator & Solver & Visualizer - YouTube]

Group roster

Team members:

19125001 - Chu Đức An

19125051 - Nguyễn Thiệu Khang

19125064 - Từ Tấn Phát

Task divisions

No.	Task	Person	Percent
1	Formulate problems (input, output, data structure, understand solution methods,...)	Phát	100%
2	Write CNF clause for b	Phát	100%
3	Generate CNF from chess placement (level 1)	An	100%
4	Generate CNF from chess placement (level 2)	An	100%
5	Generate CNF clause and find satisfied values	An	100%
6	Define A* problem and solution (states, heuristics, cost)	Phát	100%
7	Input CNF clause from text	Khang	100%

8	Implement A* solver in Python	Khang	100%
9	Visualize chessboard solution step by step	Khang	100%
10	User interface for visualization	Khang	100%
11	Record the running program and upload to youtube	An	100%
12	Debugging and write all solutions to a, b and e in report	Phát	100%
13	Finalize CNF format for (c), (d), (e) and result format	All members	100%
14	Proofread report and validate source code	All members	100%

Completion level estimation

No.	Criteria	Self-estimate	Maximum score
1	Task a	10%	10%
2	Task b	10%	10%
3	Task c includes: Level 1 Level 2	20%	20% 10% 10%
4	Task d	20%	20%
5	Task e	20%	20%
6	Task f	10%	10%
7	Comply with the regulations of submission requirements	5% (demo does not provide all test cases)	10%
Total		95%	100%

1. Problem formulation (answer for <a>)

Input:
N: the number of queens placed in advance
The next input is an array of N variables representing the positions of the placed queen. Each of which has a format (i, j) where 0 <= i, j <= 7; i is the row number and j is the column number.
For example: N = 3, [(1, 2), (3, 4), (2, 7)]
The chess board according to this input will resemble this.

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

Output: the possible arrangement of remaining queens so that no queen attacks each other OR an invalid assignment is announced.
For example, with the input example, this output will produce “Invalid assignment” because there is already a pair of queens that attack each other.

State: we need the following variables and data structures to represent the state of the program.

- A boolean value valid_state to indicate whether or not the chess board still satisfies no-attacking-pair-of-queens constraint.
- An integer to n to show how many of the queens have been placed.
- A 2-dimensional matrix chess_board that has 8 columns and 8 rows, each numbering from 0 - 7, for example chess_board[0][1] points to the chess tile at row 0 and column 1.
- Each tile in the matrix will be a tuple (has_queen, can_place_queen), where has_queen is a boolean indicating whether we have placed a queen on that tile already, and can_place_queen is another boolean to indicate that we can place a queen on that tile to achieve a correct arrangement.

In the **initial state**, there will be n tiles in the matrix containing (has_queen = true, can_place_queen), or none at all.
Example:

	0	1	2	3	4	5	6	7
0	False, false	**True, true**	False, false	False, false	False, false	False, false	False, false	False, false
1	False, false	False, false	False, false	**True, true**	False, false	False, false	False, false	False, false
2	False, true	False, false	False, false	False, false	False, false	False, true	False, false	False, true
3	False, true	False, false	False, true	False, false	False, false	False, false	False, true	False, true
4	False, false	False, false	False, true	False, false	**True, true**	False, false	False, false	False, true
5	False, true	False, false	False, true	False, false	False, false	False, false	False, false	False, false
6	False, true	False, false	False, false	False, false	False, false	False, true	False, false	False, false
7	False, true	False, false	False, true	False, false	False, false	False, true	False, true	False, true

In a **valid state** (where there are no attacking pairs of queen), there will be no tuple in the matrix that contains (has_queen = true, can_place_queen = false). If one such tuple exists, we can conclude that the queens we have placed up until now are invalid.
In an unfinished state, there will still be tuples containing (has_queen = false, can_place_queen = true).
Finally, in the **goal state**, there will be 8 queens on the chess board, so the matrix will contain 8 tuples with (has_queen = true, can_place_queen = true) and the rest has to be (has_queen = false, can_place_queen = false).

	0	1	2	3	4	5	6	7
0	False, false	False, false	**True, true**	False, false	False, false	False, false	False, false	False, false
1	False, false	False, false	False, false	False, false	False, false	**True, true**	False, false	False, false
2	False, false	False, false	False, false	False, false	False, false	False, false	False, false	**True, true**

3	**True, true**	False, false	False, false	False, false	False, false	False, false	False, false	False, false
4	False, false	False, false	False, false	**True, true**	False, false	False, false	False, false	False, false
5	False, false	False, false	False, false	False, false	False, false	False, false	**True, true**	False, false
6	False, false	False, false	False, false	False, false	**True, true**	False, false	False, false	False, false
7	False, false	**True, true**	False, false	False, false	False, false	False, false	False, false	False, false

2. Write sample CNF clauses (answer for <b>)

A CNF clause to describe restrictions required when Florence places a queen in the cell[3][3].
To write this CNF clause, we will define the following syntax conventions:

- [i][j] will point to the corresponding cell in the matrix, in the i-th row and the j-th column.
- Q[i][j] will indicate that we can place a queen on the cell [i][j]. On the contrary, not(Q[i][j]) indicates we cannot place queen on that cell.

Thus, we can write the restrictions on the chessboard as follows.

a. There can only be at least and at most one queen on each row.

Clause (a): not(Q[3][0]) ^ not(Q[3][1]) ^ not(Q[3][2]) ^ not(Q[3][4]) ^ not(Q[3][5]) ^ not(Q[3][6]) ^ not(Q[3][7])

b. There can only be at least and at most one queen on each column.

Clause (b): not(Q[0][3]) ^ not(Q[1][3]) ^ not(Q[2][3]) ^ not(Q[4][3]) ^ not(Q[5][3]) ^ not(Q[6][3]) ^ not(Q[7][3])

c. There can only be at least and at most one queen on each diagonal.

Clause (c): not(Q[0][0]) ^ not(Q[1][1]) ^ not(Q[2][2]) ^ not(Q[4][4]) ^ not(Q[5][5]) ^ not(Q[6][6]) ^ not(Q[7][7]) ^ not(Q[0][6]) ^ not(Q[1][5]) ^ not(Q[2][4]) ^ not(Q[4][2]) ^ not(Q[5][1]) ^ not(Q[6][0])

d. We can place another queen on an unrestricted cell

Clause (d): Q[0][1] v Q[0][2] v Q[0][4] v Q[0][5] v Q[0][7] v Q[1][0] v Q[1][2] v Q[1][4] v Q[1][6] v Q[1][7] v Q[2][0] v Q[2][1] v Q[2][5] v Q[2][6] v Q[2][7] v Q[4][0] v Q[4][1] v Q[4][5] v Q[4][6] v Q[4][7] v Q[5][0] v Q[5][2] v Q[5][4] v Q[5][6] v Q[5][7] v Q[6][1] v Q[6][2] v Q[6][4] v Q[6][5] v Q[6][7] v Q[7][0] v Q[7][1] v Q[7][2] v Q[7][4] v Q[7][5] v Q[7][6]

Finally, the CNF clauses to represent all these restrictions is clause (a) ^ clause (b) ^ clause (c) ^ clause (d).

3. A* algorithm to solve 8-queens problem (answer for <e>)

Our team has many approaches to solving this 8-queen problem, using a combination between the CNF clauses and implementations of A* algorithm. The main difference between these approaches does not lie on how we form the CNF, but on the way we choose the heuristic and cost functions to carry out traversal of A* graph.

Before we move on to the detailed description of the A* algorithm, we have to define the inputs and state representation. We first initialize our prerequisite CNF clauses (our knowledge base), consisting of rules of the 8 queen problems, such as there has to be only 1 queen at every row, column and diagonal. These CNF clauses will be used to continuously compare with new CNF clause we obtain during A* traversal.

The input file carries the number of starting queens and their positions. Since the problem assumes that the input will give the correct positions of the starting queens, meaning no 2 queens will attack each other, we see that they already satisfy all CNF clauses in our knowledge base, so no preprocessing or comparison is required.

Then, we come to different ways we can implement this A* algorithm (**note, in the code, we only implement methods in (a), the other approaches only show our group's different ideas and how we worked out the solutions**):

a. Choosing the heuristic as the number of possible placements remaining, the cost equals 0 or the number of queens we have placed/moved.

Each state in the problem (or each node in a graph) will be represented as a CNF clause showing where we placed the queens, for example: q(34) ^ q(47) means we placed 2 queens, at positions (34/8, 34%8) = (4,2) and (47/8,47%8) = (5,7) on the chessboard. q(x) means we have already placed a queen at position x. In the state representation we do not need to mention p(x) (p(x) means we can place a queen at position x, the rest of x where p(x) is not mentioned will imply that we cannot place any queen at position x). By comparing the obtained CNF clause of the current state and all the CNF clauses in our knowledge base, we can derive the list of cells that are still possible for placement.

Our goal in this case is to minimize the number of possible placement while placing all queens on the chessboard. If we happen to run out of possible placement cells but the all 8 queens have not been allocated, then we have to choose a different track of the A* tree traversal. For that reason, our team wants to choose the heuristic as the possible placement cells remaining.

For this approach, we will choose the heuristic as to how many available cells we have in the considered state. Now, to obtain the next state, we first choose one of the cells still available for placement. For example, given the current state as follows.

This state has cost = g = 4 (placed 4 queens), heuristic = 9 (for 9 possible queen placement)

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

The state representation is q(1) ^ q(16) ^ q(53) ^ q(58). We can see that empty cells can be derived easily from 8-queen rules and they consist of cells (1,3), (1,4), (1,5), (1,7), …By choosing to place a queen in one of these cells, we do not have to worry that they conflict with our CNF knowledge base. Suppose from the priority queue, we determine to choose cell (1,4) as the next placement, then the next state will now be q(1) ^ q(16) ^ q(53) ^ q(58) ^ q(12). Here is the chessboard for the next state:

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

Now the cost will be 5, the heuristic drops to 4. Suppose A* determines cell (4,6) to place a queen next.

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

The cost is 6, the heuristic is now 1. We can already see that this A* path is wrong, since there is no way to reach the goal state. The goal state demands that all 8 queens be placed and the remaining cells are unplaceable, but there is only 1 possible cell left and only 6 queens have been placed so far. This means we have to let A* choose another possible placement along the earlier branches.

We have also considered the option where the cost is dropped completely (cost = 0 for every state), meaning it is not important how many queens we placed, only the heuristic should be considered for faster way to completion. Regardless, for both choices of cost function, we obtain a complete result (and frankly, the shortest path, since every complete path requires us to place 8 queens anyway).

To summarize, we can see that the true goal cost is always 64 (make sure 8 queens are placed and the remaining 56 cells are marked as unavailable for placement), the cost increases by 1 when we place one more queen (or it may remain 0 for every state we arrive at), and the heuristic drops depending on the number of possible cells we can still place a queen on. The result is complete and optimal at the same time.

b. Choosing the heuristic as the number of possible placement remaining and the cost as the number of placed queens.

Similar to the method above, the state representation and the way we determine the heuristic remain unchanged. The difference here is that the cost function is not linear, and should be

determined by the number of cells we “covered” so far (the cells that have been marked as unavailable due to the queens we placed). The path cost is then obviously not a constant, but the difference between the number of covered cells in the next state and the current state.

For this approach, we will choose the heuristic as to how many available cells we have in the considered state. We still obtain the next state by choosing one of the available cells, while keeping in mind their cost also.

For example, given the current state as follows. This state has cost = g = 55 (placed 4 queens, 51 cells are now attacked or covered), heuristic = 9 (the remaining 9 possible queen placement)

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

Suppose A* determines we place the next queen in (1,4), the cost is now 62 and the heuristic drops to 4.

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

We have demonstrated how A* works with this heuristic and cost functions, but we can see an obvious flaw when choosing this approach. The number of tiles covered + the number of available tiles = all cells = 64 for whatever state we are in, which leads to a priority queue where all pushed nodes are of equal priority. Hence, this approach causes A* to move rather randomly and thus is not optimal at all. This method still produces the complete solution, however.

c. Choosing the heuristic as the number of attacking pairs of queens, the cost as the number of queens we have placed.

We keep the way we represent each state as a CNF clause showing where we placed the queens so far. In this case, the heuristic is the number of pairs of queens attacking each other and the cost is the count of queens we have placed. Obviously we have to make A* avoid the scenarios where there are attacking pairs of queens, because any further along that path will only lead to invalid results. The path cost is then uniformly 1.

This heuristic is admissible because the true path cost is always 8 to any solution, and the maximum number of attacking pairs of queens is 8C2 = 28 pairs, but we choose to ignore any road that leads to even 1 attacking pair of queens (always keep heuristic at 0).

For example, consider the following chessboard. At the start: cost g = 4 (4 queens placed) and heuristic h = 0.

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

Already from this example, we can see that there is a flaw in the heuristic. If we always try to keep h = 0, then this search becomes Uniform-cost search and not A* anymore. The results generated are complete and optimal, but have to go through a lot of states since there is virtually no heuristic.