homework AI

Benedetta Conti 2045621

January 2026

1 Introduction

For this project, I chose the N queens problem and modeled it both as a search problem, solved using the A* algorithm, and then solved it using a solver, treating the problem as a CSP.
Additionally, a small animation was implemented using the matplotlib library, which allows visualizing the board and the solutions found.
Several experiments were conducted in which the parameter N, indicating the size of the chessboard and thus the number of queens to be placed while ensuring no conflicts, was scaled. The metrics that were analyzed in each experiment, using A*,are the following:

- **running time**
- **nodes generated**
- **nodes expanded**
- **max number of nodes in memory**

The analysis of A* was conducted using two different heuristics, discussed in more detail in the dedicated chapter, to observe how the behavior of this algorithm changes based on the chosen heuristic function. The first heuristic function used is less informative, while the second is based on the number of conflicts, allowing fewer states to be explored. For each of these two experiments, graphs were created showing the variation of the previously defined metrics.

The metrics that were analyzed in each experiment, using the solver, are the following:

1. **running time**
2. **assertions**: number of assertions added by the solver
3. **decisions**: number of logical decision taked by the solver
4. **conflicts**: number of conflicts foundt by the solver
5. **propagations**: number of propagation executed by the solver

2 Problem

The chosen problem is N **queens**, whose goal is to place N queens on an NxN chessboard ensuring that no queen attacks another.
More precisely, two queens attack each other when they are in any of the following situations:

1. they are both in the same row
2. they are both in the same column
3. they are both on the same diagonal

Two images are presented below. In the first image it is possible to see that on the chessboard there are queens attacking each other. In the second image no queen is attacking another, therefore it is a goal state.
The images presented previously are generated using Python. The complete script responsible for creating these visualizations are available in the GitHub repository.

Two different solution approaches were used for the problem:

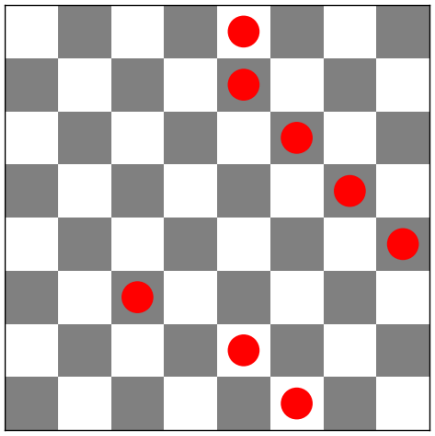

Figure 1: 8-Queens problem with conflicts

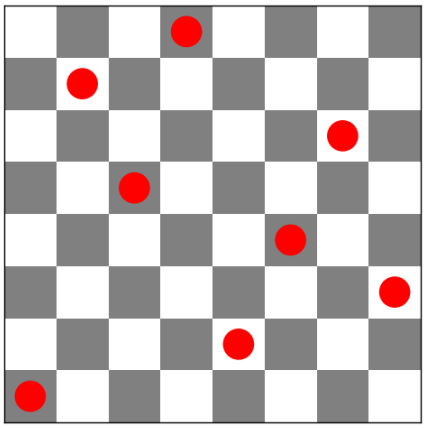

Figure 2: Solution of the 8-Queens problem

1. in the first case, the problem was treated as a search problem and the chosen algorithm was **A***.
2. in the second case, the problem was treated as a **CSP**, described by a set of variables, each with its domain, and a set of constraints.

The **Node** class has been defined to correctly implement A*, and it contains the following information:

- **state**: this field contains the chessboard configuration
- **parent**: this field contains the node's parent
- **g**: this field contains the cost
- **h**: this field contains the value of the heuristic function
- **f=g+h**: this filed contains the sum of the path cost and the value of the heuristic

3 Implementation of A*

The implementation of the **A*** algorithm is generic; in fact, it can use different heuristics that are passed as input, just as it can be used for various problems.
The **frontier**, used to keep track of candidate nodes to be explored and organized in such a way as to select first the nodes with a smaller f function value, has been implemented through the **heapq** module, used to define a data structure that, in this case, keeps the node with the lowest value on top, ready to be selected next.
The main functions of heapq that were used in the algorithm are now described to provide a clear view of all the steps:

- **heapq.heappush(frontier,node)**: this function is used to add the node in the frontier
- **heapq.heappop(frontier)**: this function is used to remove from the frontier the first node, the one with the minimum value of the function f
- **heap.heapify(frontier)**: this function is used to trasform the frontier in a valid heap

It's now proposed the precise description of what the algorithm A* implemented does in each step.

Algorithm 1 Algoritmo A*

```
start_time ← current time                          ▷ it memorizes the starting time
start_state ←initial state of the problem          ▷ corresponds to the empty chessboard
create a node n_0 with:
    state = start_state
    g = 0
    h = heuristic(start_state)
frontier ← empty priority queue
add n_0 in frontier          ▷ the frontier is ordered by ascending f = g + h
frontier_states ← {start_state → n_0}   ▷ Dictionary used to have a fast access and reduce the time complexity
explored ← ∅                                       ▷ Set of the explored states
Initialization of metrics                          ▷ Expanded nodes, generated nodes
while frontier is not empty do
    n ← extract the node with minimum f from frontier
    remove n.state from frontier_states
    if n.state is a goal state then
        compute the total time of execution
        return solution path and the metrics
    end if
    add n.state a explored
    udpate the number of expanded nodes
    actions ← appliable actions in n.state
    num_children ← |actions|                       ▷ number of successor's states
    for each action in actions do
        child_state ← results of applying action
        g' ← n.g + cost(n.state, action)
        h' ← heuristic(child_state)
        create the child node child
        update the number of generated nodes
        if child.state ∉ explored and child.state ∉ frontier_states then
            add child in frontier
            update frontier_states
        else if child.state ∈ frontier_states then
            if g' < g  then
                substitute the new node foundt with the one that is in frontier
                                                   ▷ Foundt a better path
            end if
        end if                          5
    end for
end while
return failure                                     ▷ No solution foundt
```

The heuristics that have been implemented are 2, in order to visualize the different behavior of the algorithm depending on the heuristic, which is discussed in more detail in the chapter dedicated to experiments. The heuristics are:

- the number of queens still to be placed on the chessboard
- the number of queens that attack each other

In particular the use of the dictionary is particularly important and advantageous for reducing execution time. In fact, if the aforementioned dictionary had not been used, to check whether a node was already present in the frontier, the cost would have been:

$$O(n)$$

where **n** is the length of the frontier.
Thanks to the use of the dictionary the cost for this search is just:

$$O(1)$$

The impact of this choice is not visible when the size of the chessboard is particularly small; however, when the border is large, this choice shows its advantages in terms of computation time.

4 Implementation of CSP

The problem is now modeled as a Constraint Satisfaction Problem (CSP) and the solver used is **z3**.
In particular, a CSP is a problem defined by a set of **variables**, each with its own **domain**, and a set of **constraints** that must be satisfied in order to consider the problem solved. It's now given a description of these elements.

4.1 Variables

The n variables (where it is remembered that n is the size of the chessboard and therefore the number of queens to be placed) have been modeled as integer variables. In particular, a variable:

$$Qr$$

indicates the column in which the queen occupying row **r** must be placed.

4.2 Domains

The domain of each variable is as follows:

$$0 \leq Q_i < n \quad \forall i \in \{0, \ldots, n-1\}$$

This domain specifies that each variable can take a value between 0 and n-1, indicating the column in which the respective queen is located.

4.3 Constraints

The constraints are necessary to ensure that no queen attacks another, and consequently they must be ensured to prevent diagonal attacks and column attacks. In particular, it is important to specify that, by defining the variables in the manner previously described, it is automatically ensured that there are no row constraints, meaning that each queen occupies a separate row and therefore no more than one queen can be found on the same row.
Two queens are on the same column, causing a conflict, when:

$$Qi = Qj$$

consequently, to avoid this the following constraint is imposed:

$$Q_i \neq Q_j \quad \forall i \neq j, \quad i, j \in \{0, \ldots, n-1\}$$

Two queens cause a conflict on the two diagonals when:

$$|Q_i - Q_j| = |i - j|$$

consequently, in order to ensure the complete absence of conflicts, the following constraint is imposed:

$$|Q_i - Q_j| \neq j - i \quad \forall i < j, \quad i, j \in \{0, \ldots, n-1\}$$

5 Experiments

Several experiments were carried out by varying the parameter **n**, which indicates the size of the chessboard.
For each experiment performed with **A***, the following metrics were measured: expanded nodes, generated nodes, and running time.
Graphs are indeed presented below.
The following graph highlights the increase in expanded nodes as the parameter **n** varies, emphasizing how quickly this number grows.

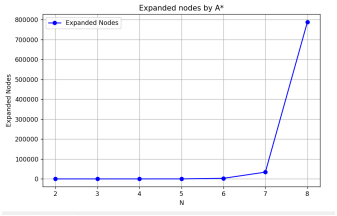

Figure 3: Expanded nodes by A*

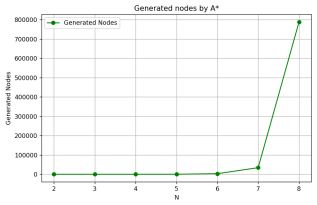

Figure 4: Generated nodes by A*

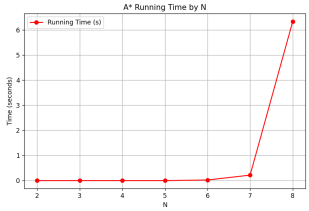

Figure 5: A* running time

We can see that the heuristic used, that is the number of queens that still need to be placed, although it is admissible and consistent, is not very informative, which is why a large number of states are explored and generated.
To demonstrate the different powers of a heuristic, a second experiment was conducted, using as a heuristic the number of conflicts between the queens that have already been placed, in order to observe the differences compared to the previous heuristic. This experiment also analyzes the previous metrics as the parameter N varies.

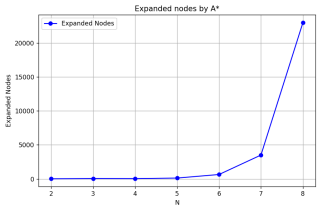

Figure 6: Expanded nodes by A*

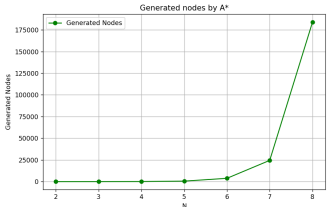

Figure 7: Generated nodes by A*

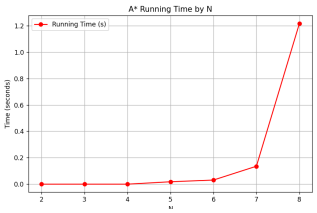

Figure 8: A* running time

The graphs highlighting this are shown below.

Here is a table showing the results obtained by the solver for different values of n. In this case, the following metrics were analyzed:

1. **running time**
2. **assertions**: number of assertions added by the solver
3. **decisions**: number of logical decision taked by the solver
4. **conflicts**: number of conflicts foundt by the solver
5. **propagations**: number of propagation executed by the solver

Table 1: Metrics of Z3 by changing n

N	Time (s)	Assertions	Decisions	Conflicts	Propagations
4	0.00061	10	28	7	55
5	0.00901	21	59	6	56
6	0.00678	28	155	11	147
7	0.01097	36	187	14	259
8	0.01500	45	250	18	340

6 How to Run

To run this project, the user needs to have Python installed and also the following dependencies:

- **Z3 solver**: required for constraint solving.
- **Matplotlib**: required for visualizing the results.

Once the dependencies are installed, the project can be executed by running:

```
python main.py
```

Upon execution, the user will be prompted to make the following choices:

1. **Select the approach:** choose between `A*` or `CSP`.
2. **Select the heuristic(if A* has been choosen):** choose between:
   - Number of remaining queens to place.
   - Number of conflicts.
3. **Select the experiment type:**
   - **Single experiment:** the user is asked to specify the size of the chessboard.
   - **Multiple experiments:** the user can test multiple board sizes at once. Enter the sizes as numbers separated by commas. For example, entering `2,5,7` will run experiments on boards of size 2, 5, and 7.

It's now showed a little example on how the terminal appear in the two cases:

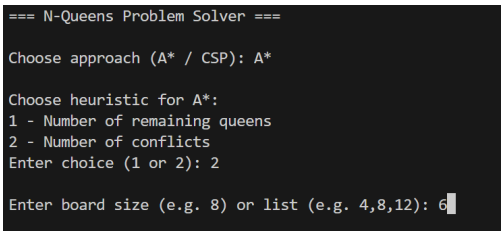

Figure 9: Example of choosing A*

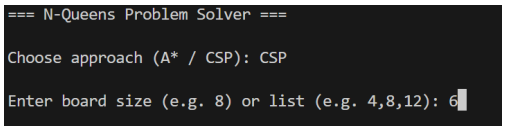

Figure 10: Example of choosing CSP