

## Problem 3: 8-Queens-Problem++

### Problem Description

In the classical 8 queens problem<sup>1</sup>, the goal is to place 8 queens on a 8x8 chessboard such that they can not strike each other. A queen can strike another queen, if they are both in the same row, column, or diagonal.

This exercise is based on this problem, but with several important modifications:

- The size of the board is **not necessarily 8x8, but can be 5x5, 6x6,...**
- We do **not focus our attention on finding the (theoretical) maximum number of queens** that can be placed on a board. This means, for example, that solutions with 7 queens on an empty 8x8 chessboard will also be considered as valid solutions, as long as no additional queen can be placed.
- In addition to the queens, there may also be pawns on the chessboard. **Pawns can 'block' the influence of a queen, see below for more details.**

You will be given several initial chessboards, with a few queens and pawns. Your task will be to fill the chessboard with new queens, until no new queen can be placed.

In order to produce such an algorithm, you should construct a knowledge base, with which one can infer which fields on the board cannot be occupied by a new queen. **Such fields will be called danger fields.**

The location of the next queen will always be chosen as follows:

*In the first row (starting from the top) where there is an empty field, choose the first column (starting from the left) that is empty; place the next Queen there.*

You do **not** need to implement that part of the algorithm. Your task is only to set up a knowledge base that determines which fields are in danger.

Since this problem is based on general logic rules, it can be solved using propositional logic.

---




<sup>1</sup>[https://en.wikipedia.org/wiki/Eight\\_queens\\_puzzle](https://en.wikipedia.org/wiki/Eight_queens_puzzle)

## Rules

To visualize the position of the queens, pawns, and danger fields, an  $n \times n$  chessboard will be represented as a 2-dimensional numpy array “chessboard[i][j]“. The  $(i, j)$ -th entry of this array corresponds to the  $(i, j)$ -th field on the chessboard, for  $0 \leq i, j \leq n - 1$ , where  $(0, 0)$  is in the upper left corner,  $i$  indicates the column-number, while  $j$  indicates the row-number. See also the image below, in the case of a 5x5 chessboard: Each entry














(0,0)	(1,0)	(2,0)	(3,0)	(4,0)
(0,1)	(1,1)	(2,1)	(3,1)	(4,1)
(0,2)	(1,2)	(2,2)	(3,2)	(4,2)
(0,3)	(1,3)	(2,3)	(3,3)	(4,3)
(0,4)	(1,4)	(2,4)	(3,4)	(4,4)

can have four different values:

- 0, which means that the field is **empty**
- 'Q', which means that the field has a **Queen**, which will be represented as 
- 'P', which means that the field has a **Pawn**, which will be represented as 
- 'D', which means that the field is **in Danger**, which will be represented as 

## Queens












Just like for regular chess, given an empty board with just one queen, any field that is in the same column, row, or diagonal as the queen is in danger. See also the example below:

## Pawns

Pawns are able to block the danger influence of queens, meaning that all tiles 'behind' a pawn are not in danger. A more practical formulation would be that *any field on the*

*same column, row, or diagonal as a queen is in danger, unless there is a pawn in between.* The chessboard below illustrates this rule:

## Passing the Exercise

You will have passed the exercise if you can successfully solve all 8 scenarios that are given to you together with this template. Additionally, your code also has to solve 5 hidden scenarios, that are only available on Artemis.

If your implemented function computes the knowledge base such that every scenario is solved correctly (including the hidden ones), you successfully completed this programming exercise.

Your code has to compute a valid solution for each of the scenarios within 5 min on our machine. If your code takes longer to compute a solution, you will fail this submission. Don't worry about the computation time too much as usually the algorithm produces a solution within seconds for our specific exercise. Your submission will be evaluated after the deadline, but until then you can update your solution as many times as you like. The last submitted solution will be graded.

## Programming Framework

For this programming exercise a *Jupyter Notebook* will be used. The template for the exercise can be downloaded from ARTEMIS<sup>2</sup>. Since you only have to implement one single function, only minor programming skills in Python are necessary to complete this exercise. The following steps are required to correctly set up the environment for the programming exercise:

1. **Installation of Anaconda and Download of the AIMA Python Code:** If you do not already have the *Jupyter Notebook* environment installed on your machine, the installation is the first step you have to perform. We recommend to install *Anaconda*, since this will set up the whole environment for you. Furthermore, the template for the programming exercise is based on the code from the *AIMA python*<sup>3</sup> project. Instructions on how to install *Anaconda* and the *AIMA python* code can be found in the “*AIMA installation instructions*” file on Moodle<sup>4</sup>.

<sup>2</sup><https://artemis.ase.in.tum.de/courses/159/exercises>

<sup>3</sup><https://github.com/aimacode/aima-python>

<sup>4</sup>[https://www.moodle.tum.de/pluginfile.php/3159598/mod\\_resource/content/5/AIMAinstallation.pdf](https://www.moodle.tum.de/pluginfile.php/3159598/mod_resource/content/5/AIMAinstallation.pdf)

2. **Retrieving the template:** You can pull the template from ARTEMIS using git from the directory of your choice. You can see the exact link on ARTEMIS, but it should look something like this:

```
git clone https://<your_TUM_ID>@bitbucket.ase.in.tum.de/scm/GKI21LOGIC/gki21logic-<your_TUM_ID>.git
```

where the two occurrences of `<your_TUM_ID>` should be replaced by your TUM ID. You may need to enter your TUM credentials (TUM ID and password). This will download the template into the directory `gki21logic-<your_TUM_ID>`. To avoid issues with relative file paths, we recommend to copy all files contained in this directory to the root-directory of the *aima-python* project that you downloaded in the previous step, and solve the exercise from there.

After completing the above steps, you are all set up to start with the exercise. Open the *Jupyter Notebook* **Logic\_Exercise.ipynb** and go through the exercise. The notebook will introduce your task: implement the function which generates the knowledge base necessary to solve an intersection problem. To this end, implement the function `generate_knowledge` in `generate_knowledge.py`. The file `generate_knowledge.py` already contains the empty function `generate_knowledge` and is the only file you have to work on and submit. You are allowed to define other functions in `generate_knowledge.py` if it helps solving the task, but you are not allowed to import any extra packages beside the ones already imported in `generate_knowledge.py`.

An example on how to add propositional sentences to the knowledge base in such a way that they are compatible with the remaining code of the notebook is shown in the notebook. If you execute the notebook, the inference algorithm will solve the chosen intersection problem based on the knowledge base you specified. The inferred chessboard will be visualized at the bottom of the notebook, such that you can easily debug your knowledge base function.

## Inference Algorithm

The inference algorithm for this exercise uses the *Davis-Putnam-Logemann-Loveland (DPLL)* algorithm, which is not part of the lecture. It is a backtracking algorithm for inference in propositional logic, and functions in a similar way as the backwards-chaining algorithm discussed in the lecture. If you are curious to discover how the algorithm works, you can check out e.g., [https://en.wikipedia.org/wiki/DPLL\\_algorithm](https://en.wikipedia.org/wiki/DPLL_algorithm). The algorithm is able to handle knowledge bases with arbitrary structure (i.e., it does not require Horn-clauses), and is therefore recommended for this exercise.

## Submission

For the submission, you have to upload the `generate_knowledge.py` file containing your implementation of the knowledge base to ARTEMIS. To do so, copy your solution for `generate_knowledge.py` back into the previously cloned `gki21logic-<your_TUM_ID>` directory, and execute the following commands from the directory `gki21logic-<your_TUM_ID>`:

```
git add .
git config user.email "<your.TUM@email.de>"
git config user.name "<Your Name>"
```

```
git commit -m "<Write a commit message here.>"
git push
```

You may need to enter you TUM credentials (TUM ID and password) for the upload.  
Submissions will close on **21th January at 23:59**.

### ATTENTION

- Your code should **not** print anything.
- **Do NOT** rename the submitted file or the function name. If you do you will fail!
- **Do NOT** import any additional modules for your solutions. If you do you will fail!
- Like the rest of the programming exercises, this is an individual project and work **must** be your own. We will use a plagiarism detection tool and any copied code will cancel all bonus points from programming exercises for both the copier and the copied person!