BIENVENUE À OUIDAH !

NOUS SOMMES RAVIS DE VOUS ACCUEILLIR À **L'ACP SUMMER SCHOOL 2025**. CINQ JOURS D'ÉCHANGES PASSIONNANTS ET DE DÉCOUVERTES VOUS ATTENDENT DANS CE CADRE UNIQUE.

ENJOY YOUR STAY !

25-29 août 2025 / IRSP, Ouidah, Bénin



Modelling In CP

Hélène Verhaeghe





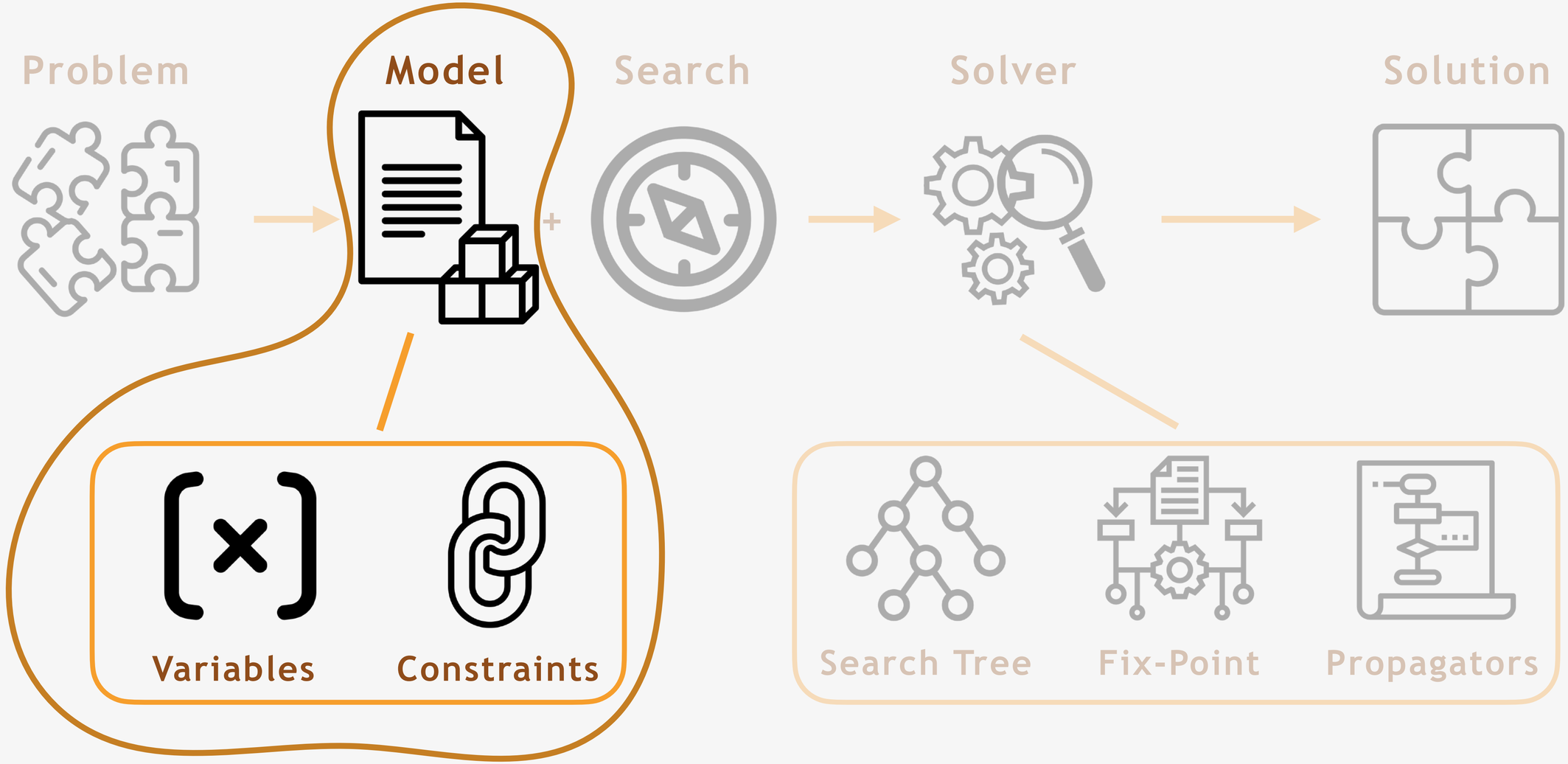




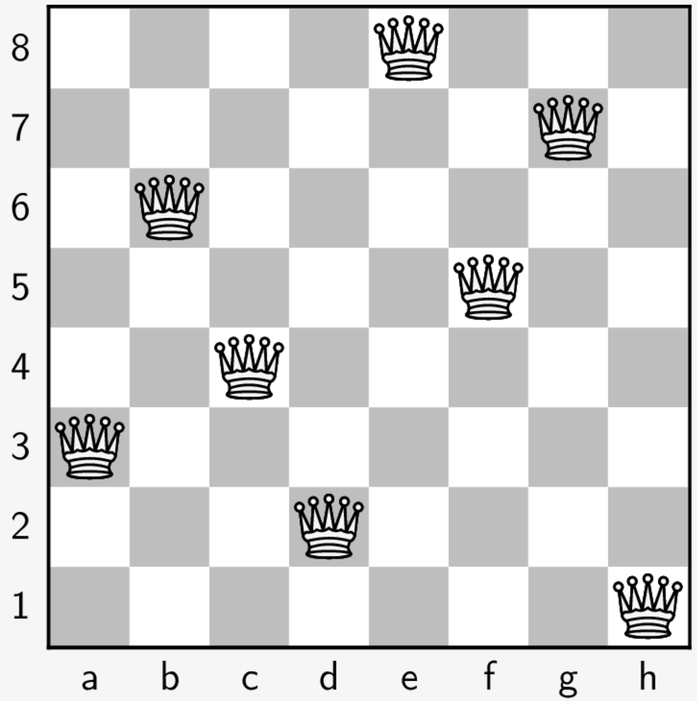




THE CONSTRAINT PROGRAMMING PARADIGM



N-Queen Problem

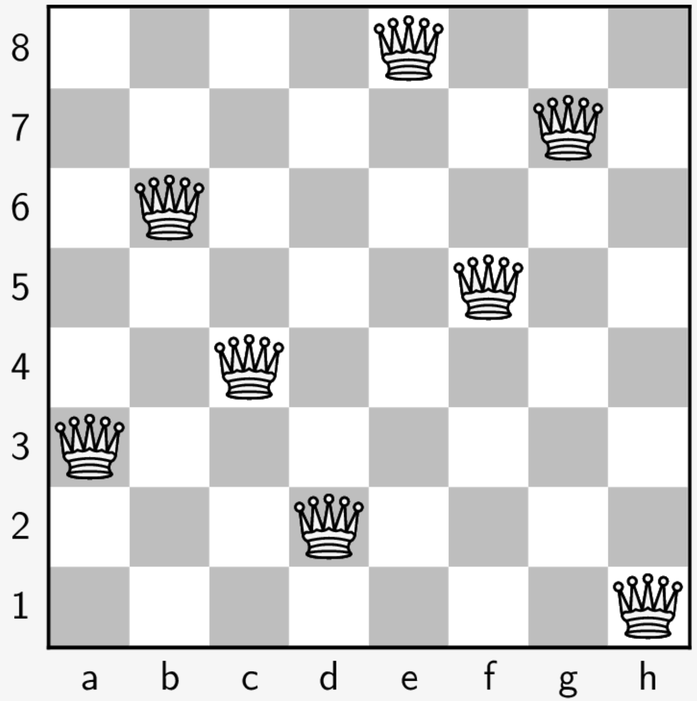


Assigning N Queens on an NxN Board Without Them Attacking Each Other

MODEL = DECLARATIVE DEFINITION OF A SOLUTION

In natural language, the description of a solution is:

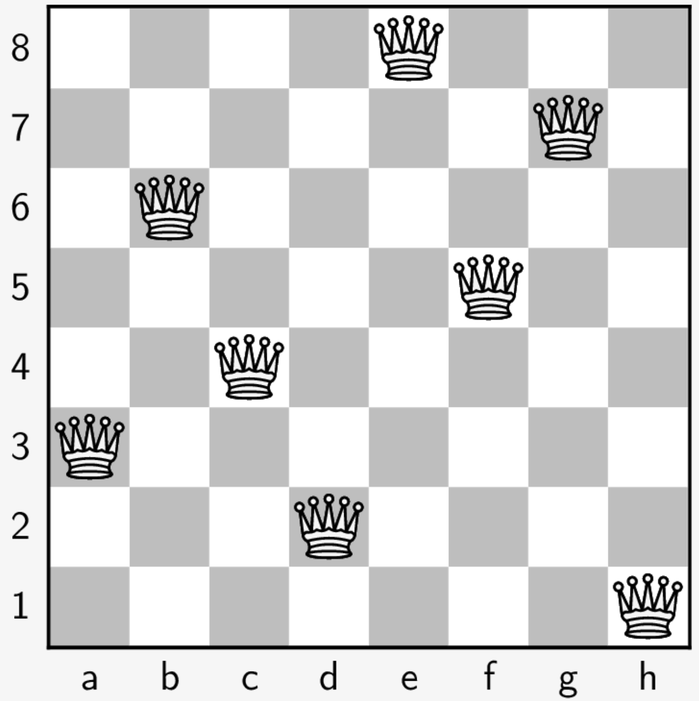
N-Queen Problem



Assigning N Queens on an NxN Board Without Them Attacking Each Other

MODEL = DECLARATIVE DEFINITION OF A SOLUTION

N-Queen Problem



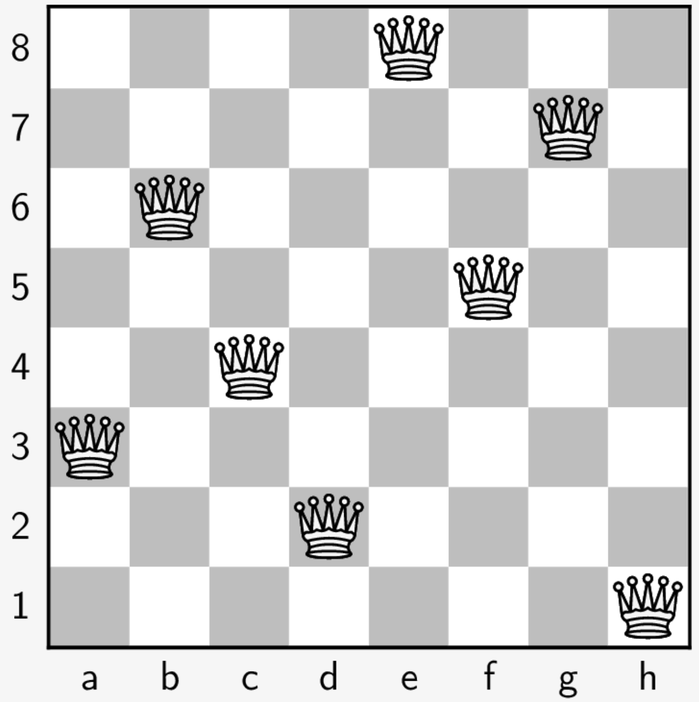
Assigning N Queens on an NxN Board Without Them Attacking Each Other

In natural language, the description of a solution is:

- We have a grid of NxN cells

MODEL = DECLARATIVE DEFINITION OF A SOLUTION

N-Queen Problem



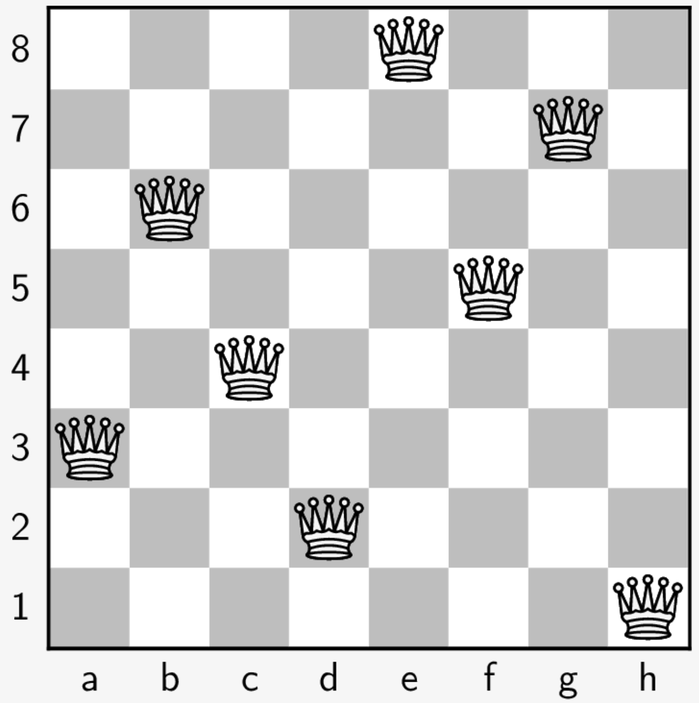
Assigning N Queens on an NxN Board Without Them Attacking Each Other

In natural language, the description of a solution is:

- We have a grid of NxN cells
- Cells can have or not a queen on them

MODEL = DECLARATIVE DEFINITION OF A SOLUTION

N-Queen Problem



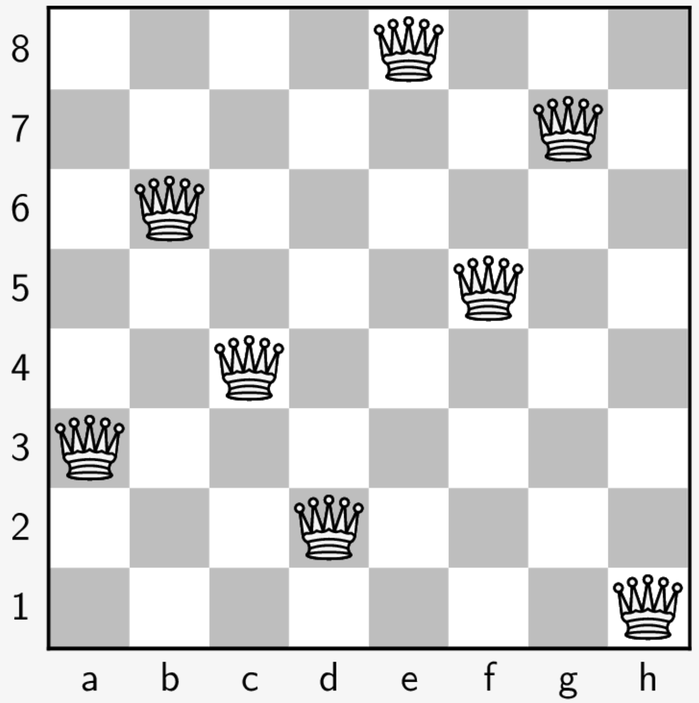
Assigning N Queens on an NxN Board Without Them Attacking Each Other

In natural language, the description of a solution is:

- We have a grid of NxN cells
- Cells can have or not a queen on them
- On each line, there can be maximum one queen

MODEL = DECLARATIVE DEFINITION OF A SOLUTION

N-Queen Problem



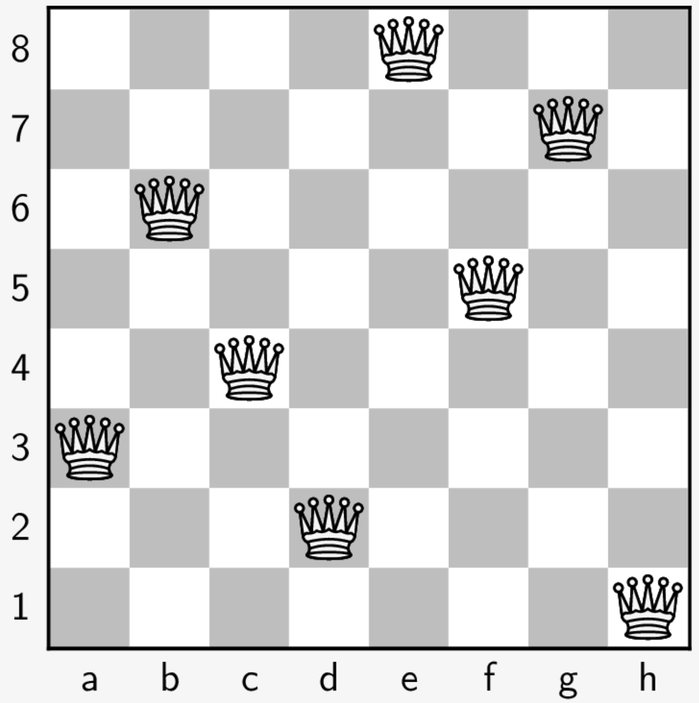
Assigning N Queens on an NxN Board Without Them Attacking Each Other

In natural language, the description of a solution is:

- We have a grid of NxN cells
- Cells can have or not a queen on them
- On each line, there can be maximum one queen
- On each column, there can be maximum one queen

MODEL = DECLARATIVE DEFINITION OF A SOLUTION

N-Queen Problem



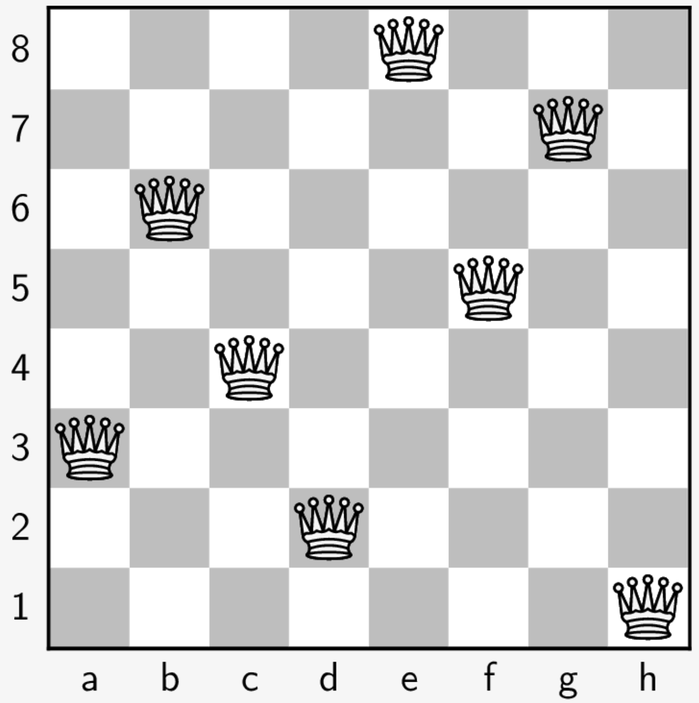
Assigning N Queens on an NxN Board Without Them Attacking Each Other

In natural language, the description of a solution is:

- We have a grid of NxN cells
- Cells can have or not a queen on them
- On each line, there can be maximum one queen
- On each column, there can be maximum one queen
- On each diagonal, there can be maximum one queen

MODEL = DECLARATIVE DEFINITION OF A SOLUTION

N-Queen Problem



Assigning N Queens on an NxN Board Without Them Attacking Each Other

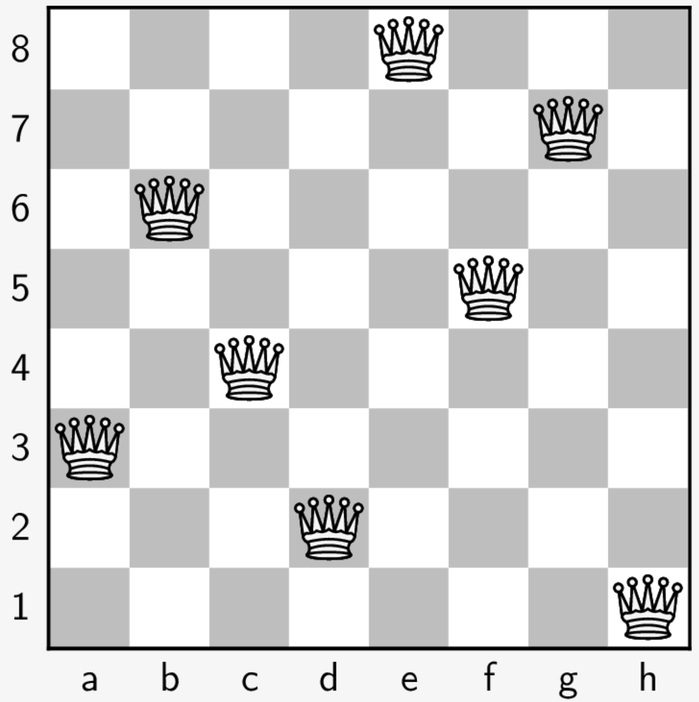
In natural language, the description of a solution is:

- We have a grid of NxN cells
- Cells can have or not a queen on them
- On each line, there can be maximum one queen
- On each column, there can be maximum one queen
- On each diagonal, there can be maximum one queen

This can then be formally translated into:

MODEL = DECLARATIVE DEFINITION OF A SOLUTION

N-Queen Problem



Assigning N Queens on an NxN Board Without Them Attacking Each Other

In natural language, the description of a solution is:

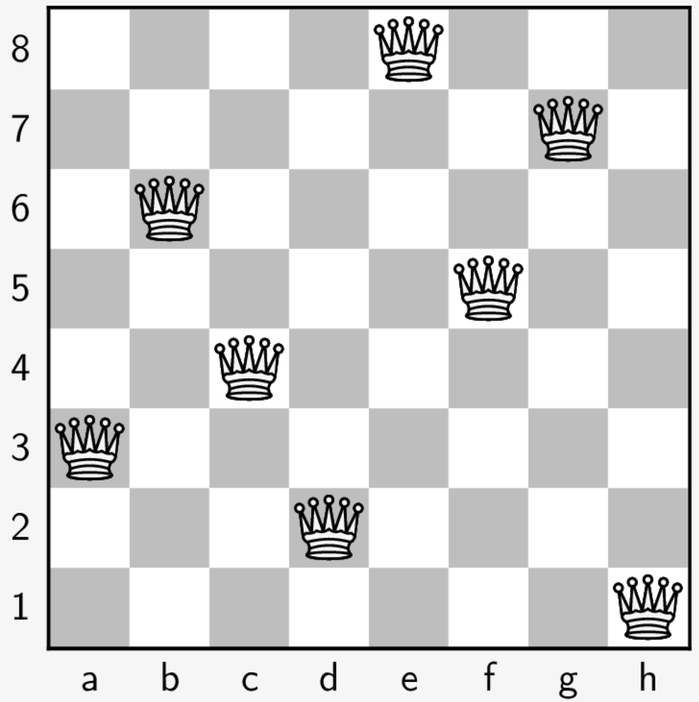
- We have a grid of NxN cells
- Cells can have or not a queen on them
- On each line, there can be maximum one queen
- On each column, there can be maximum one queen
- On each diagonal, there can be maximum one queen

This can then be formally translated into:

- Variables cell[i][j] for i,j ∈ {1..N}

MODEL = DECLARATIVE DEFINITION OF A SOLUTION

N-Queen Problem



Assigning N Queens on an NxN Board Without Them Attacking Each Other

In natural language, the description of a solution is:

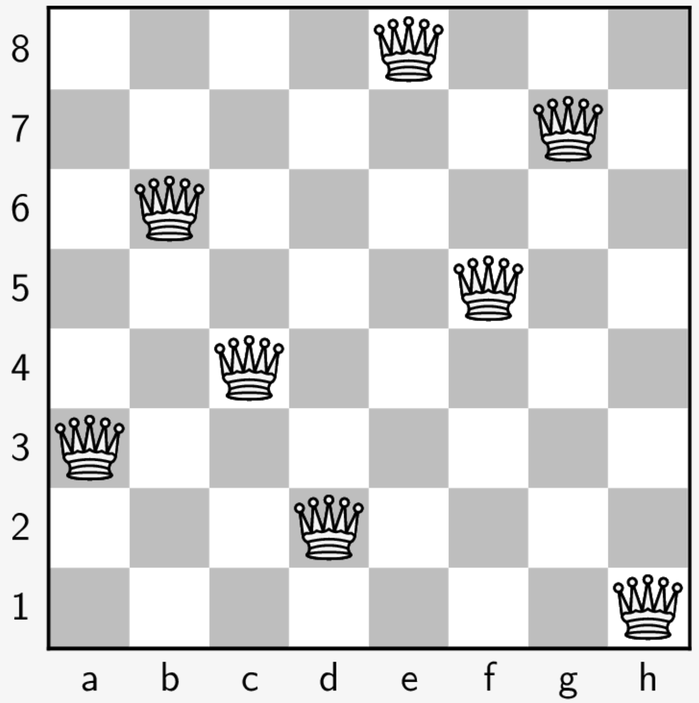
- We have a grid of NxN cells
- Cells can have or not a queen on them
- On each line, there can be maximum one queen
- On each column, there can be maximum one queen
- On each diagonal, there can be maximum one queen

This can then be formally translated into:

- Variables cell[i][j] for i,j ∈ {1..N}
- cell[i][j] = {0,1} (0 if no queen, 1 if queen on cell (i,j))

MODEL = DECLARATIVE DEFINITION OF A SOLUTION

N-Queen Problem



Assigning N Queens on an NxN Board Without Them Attacking Each Other

In natural language, the description of a solution is:

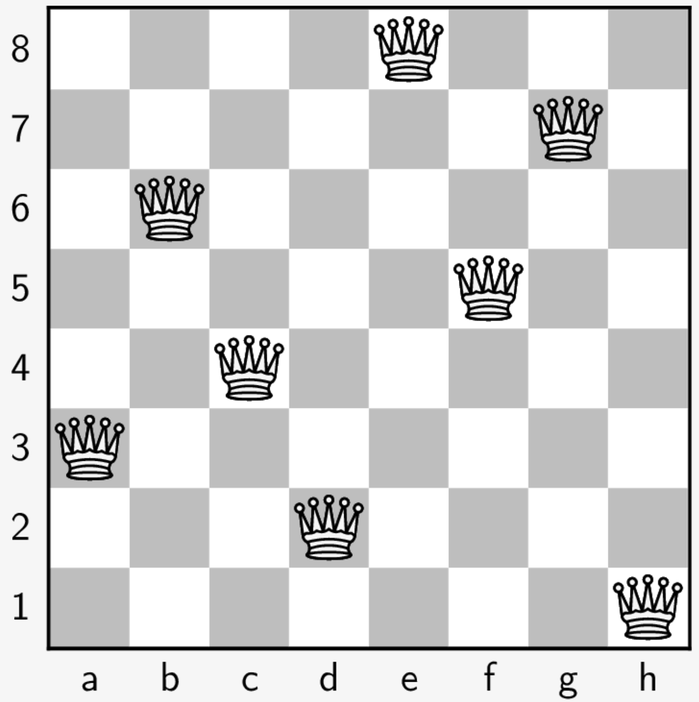
- We have a grid of NxN cells
- Cells can have or not a queen on them
- On each line, there can be maximum one queen
- On each column, there can be maximum one queen
- On each diagonal, there can be maximum one queen

This can then be formally translated into:

- Variables cell[i][j] for i,j ∈ {1..N}
- cell[i][j] = {0,1} (0 if no queen, 1 if queen on cell (i,j))
- sum({cell[i][j] ı j ∈ {1..N}}) <= 1 for i ∈ {1..N}

MODEL = DECLARATIVE DEFINITION OF A SOLUTION

N-Queen Problem



Assigning N Queens on an NxN Board Without Them Attacking Each Other

In natural language, the description of a solution is:

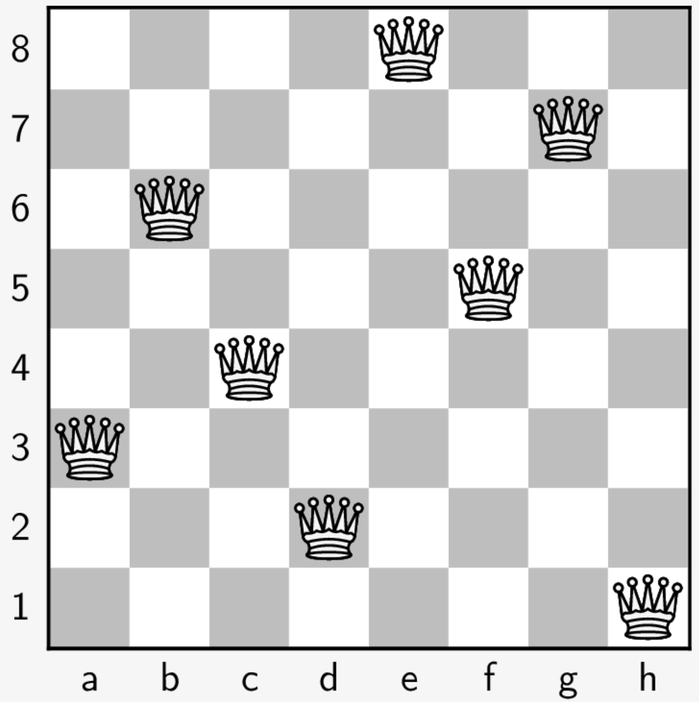
- We have a grid of NxN cells
- Cells can have or not a queen on them
- On each line, there can be maximum one queen
- On each column, there can be maximum one queen
- On each diagonal, there can be maximum one queen

This can then be formally translated into:

- Variables cell[i][j] for i,j ∈ {1..N}
- cell[i][j] = {0,1} (0 if no queen, 1 if queen on cell (i,j))
- sum({cell[i][j] ı j ∈ {1..N}}) <= 1 for i ∈ {1..N}
- sum({cell[i][j] ı i ∈ {1..N}}) <= 1 for j ∈ {1..N}

MODEL = DECLARATIVE DEFINITION OF A SOLUTION

N-Queen Problem



Assigning N Queens on an NxN Board Without Them Attacking Each Other

In natural language, the description of a solution is:

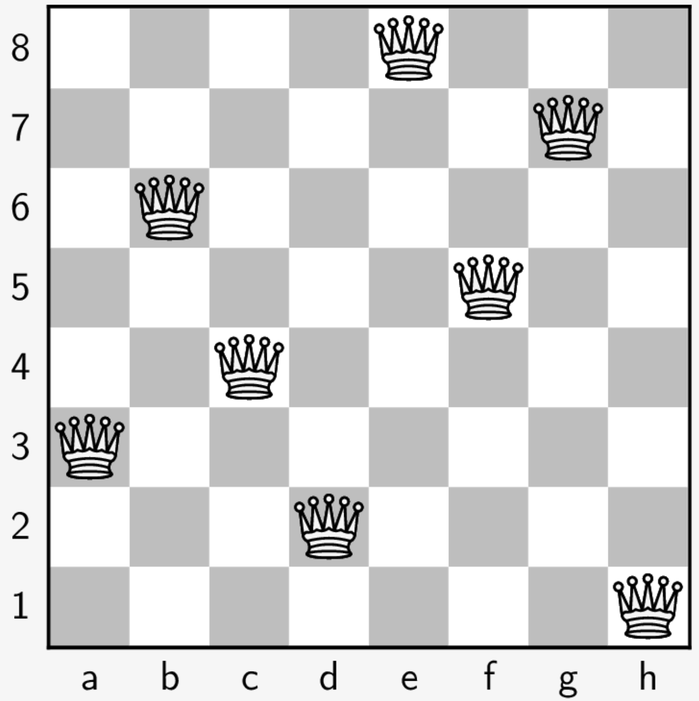
- We have a grid of NxN cells
- Cells can have or not a queen on them
- On each line, there can be maximum one queen
- On each column, there can be maximum one queen
- On each diagonal, there can be maximum one queen

This can then be formally translated into:

- Variables cell[i][j] for i,j ∈ {1..N}
- cell[i][j] = {0,1} (0 if no queen, 1 if queen on cell (i,j))
- sum({cell[i][j] ı j ∈ {1..N}}) <= 1 for i ∈ {1..N}
- sum({cell[i][j] ı i ∈ {1..N}}) <= 1 for j ∈ {1..N}
- sum({cell[i][i+o] ı o ∈ {-N..N}}) <= 1 for i ∈ {1..N

MODEL = DECLARATIVE DEFINITION OF A SOLUTION

N-Queen Problem



Assigning N Queens on an NxN Board Without Them Attacking Each Other

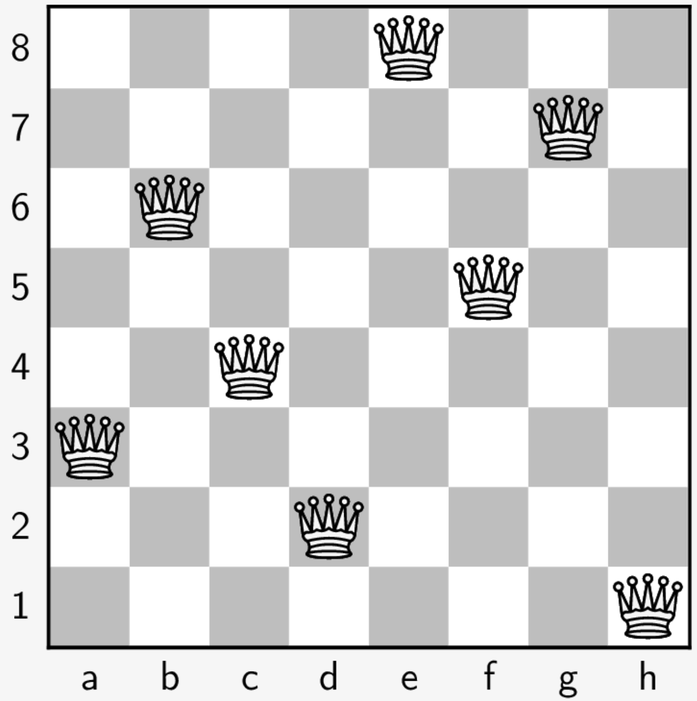
In natural language, the description of a solution is:

- We have a grid of NxN cells
- Cells can have or not a queen on them
- On each line, there can be maximum one queen
- On each column, there can be maximum one queen
- On each diagonal, there can be maximum one queen

This can then be formally translated into:

- Variables cell[i][j] for i,j ∈ {1..N}
- cell[i][j] = {0,1} (0 if no queen, 1 if queen on cell (i,j))
- sum({cell[i][j] ı j ∈ {1..N}}) <= 1 for i ∈ {1..N}
- sum({cell[i][j] ı i ∈ {1..N}}) <= 1 for j ∈ {1..N}
- sum({cell[i][i+o] ı o ∈ {-N..N}}) <= 1 for i ∈ {1..N
- sum({cell[i+o][i] ı o ∈ {-N..N}}) <= 1 for i ∈ {1..N}

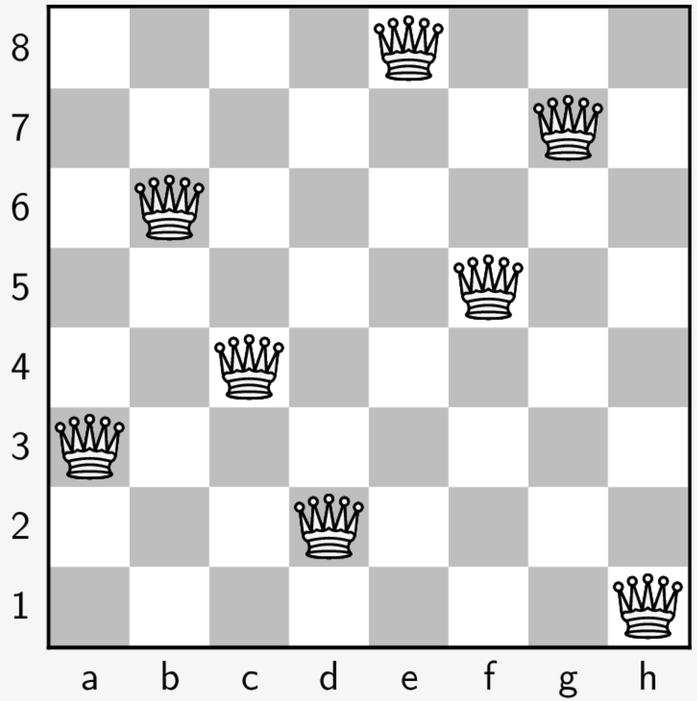
N-Queen Problem



Assigning N Queens on an NxN Board Without Them Attacking Each Other

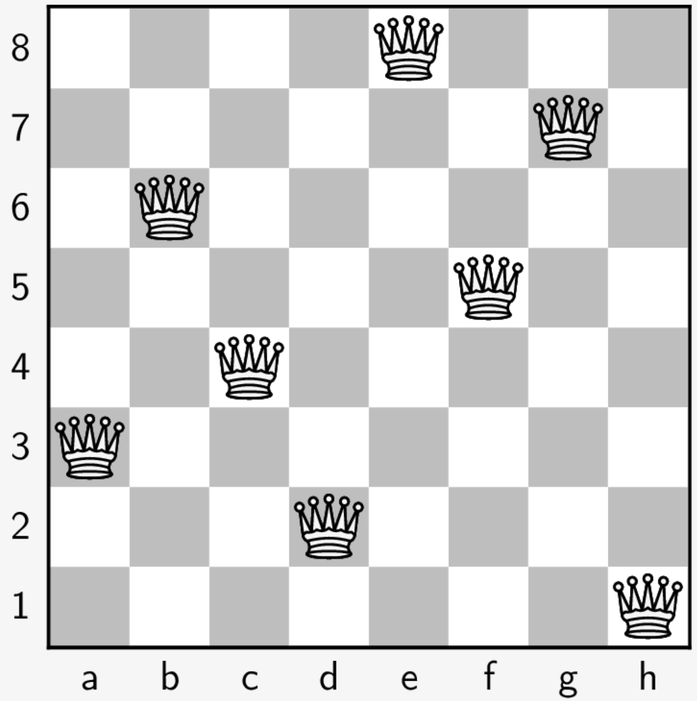
In natural language, another description of a solution is:

N-Queen Problem



Assigning N Queens on an NxN Board Without Them Attacking Each Other

N-Queen Problem

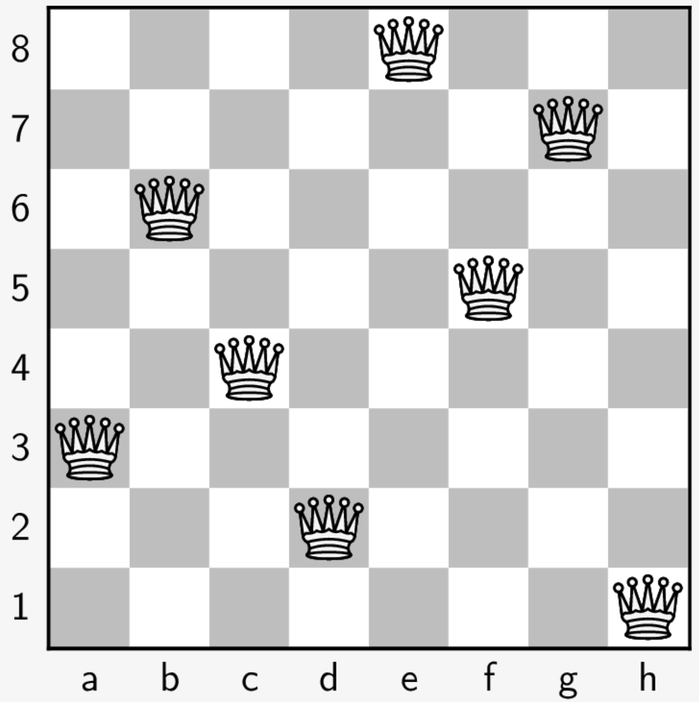


Assigning N Queens on an NxN Board Without Them Attacking Each Other

In natural language, another description of a solution is:

- We have a grid of NxN cells

N-Queen Problem



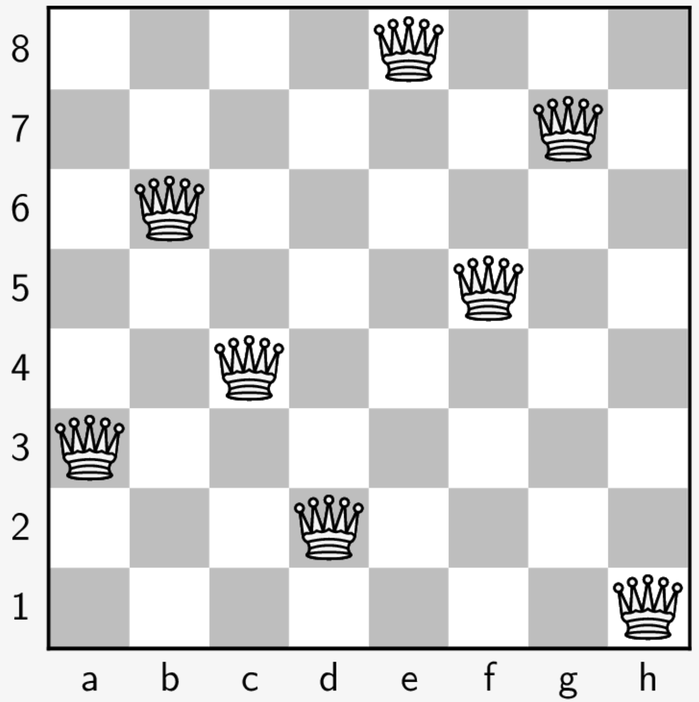
Assigning N Queens on an NxN Board Without Them Attacking Each Other

In natural language, another description of a solution is:

- We have a grid of NxN cells
- There is one queen per column of the grid

MULTIPLE MODELS FOR ONE PROBLEM

N-Queen Problem

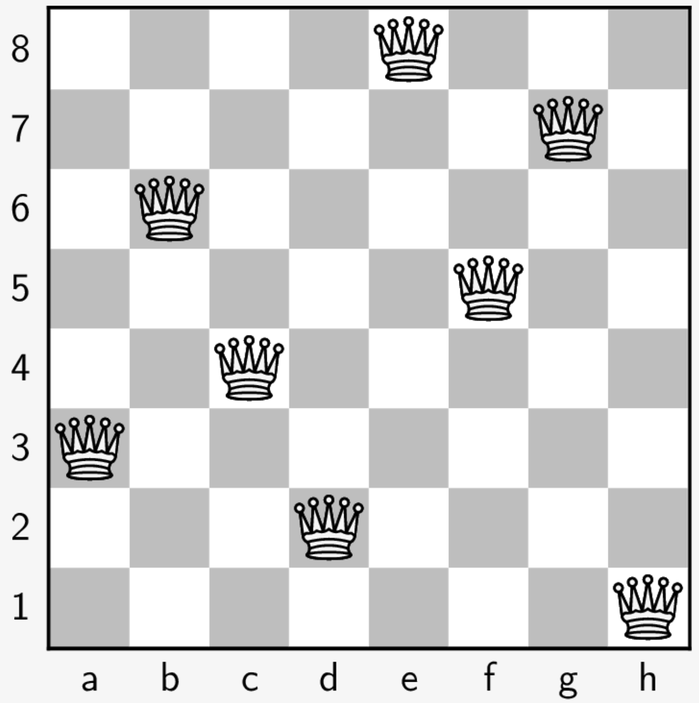


Assigning N Queens on an NxN Board Without Them Attacking Each Other

In natural language, another description of a solution is:

- We have a grid of NxN cells
- There is one queen per column of the grid
- Each of these queen cannot attack each other

N-Queen Problem



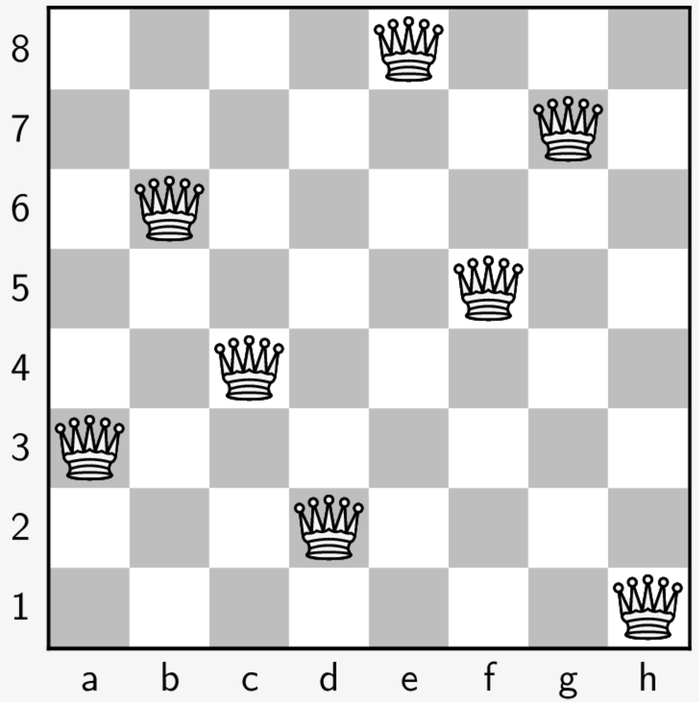
Assigning N Queens on an NxN Board Without Them Attacking Each Other

In natural language, another description of a solution is:

- We have a grid of NxN cells
- There is one queen per column of the grid
- Each of these queen cannot attack each other
- On each line, there can be maximum one queen

MULTIPLE MODELS FOR ONE PROBLEM

N-Queen Problem



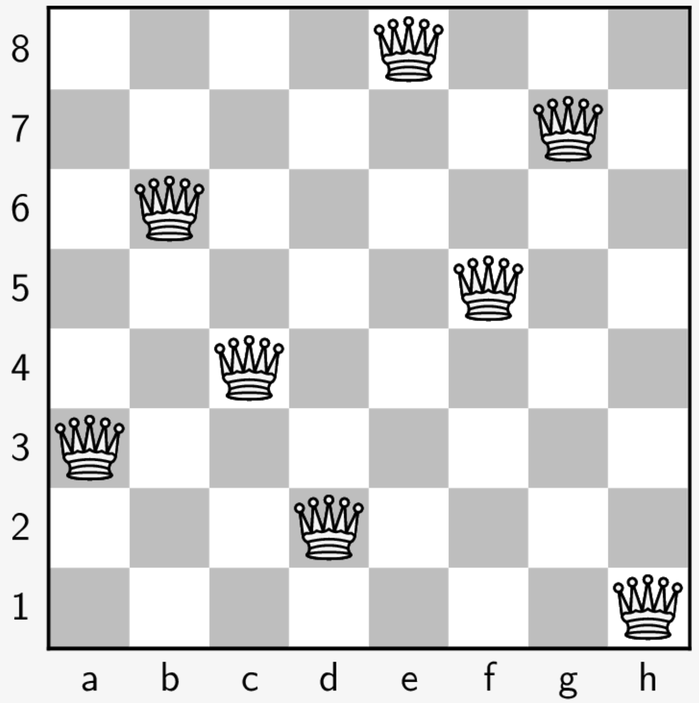
Assigning N Queens on an NxN Board Without Them Attacking Each Other

In natural language, another description of a solution is:

- We have a grid of NxN cells
- There is one queen per column of the grid
- Each of these queen cannot attack each other
- On each line, there can be maximum one queen
- On each column, there can be maximum one queen

MULTIPLE MODELS FOR ONE PROBLEM

N-Queen Problem



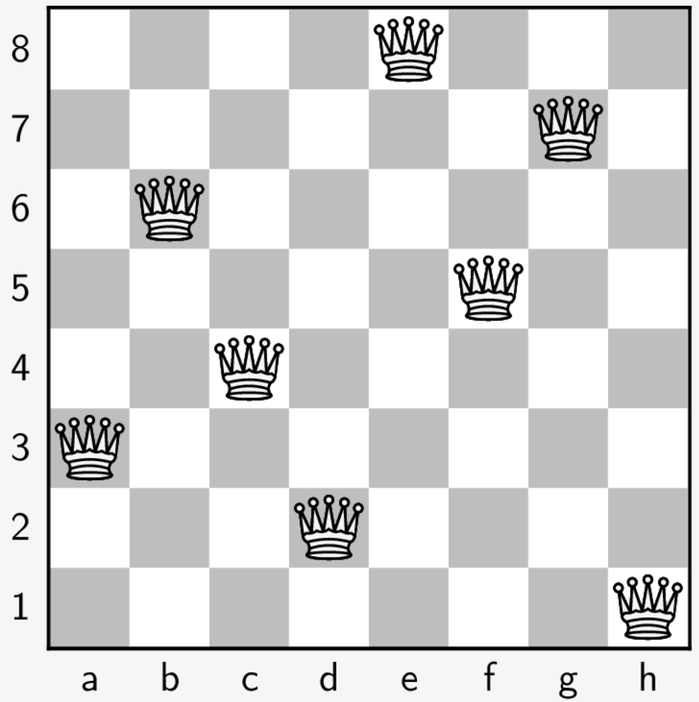
Assigning N Queens on an NxN Board Without Them Attacking Each Other

In natural language, another description of a solution is:

- We have a grid of NxN cells
- There is one queen per column of the grid
- Each of these queen cannot attack each other
- On each line, there can be maximum one queen
- On each column, there can be maximum one queen
- On each diagonal, there can be maximum one queen

MULTIPLE MODELS FOR ONE PROBLEM

N-Queen Problem



Assigning N Queens on an NxN Board Without Them Attacking Each Other

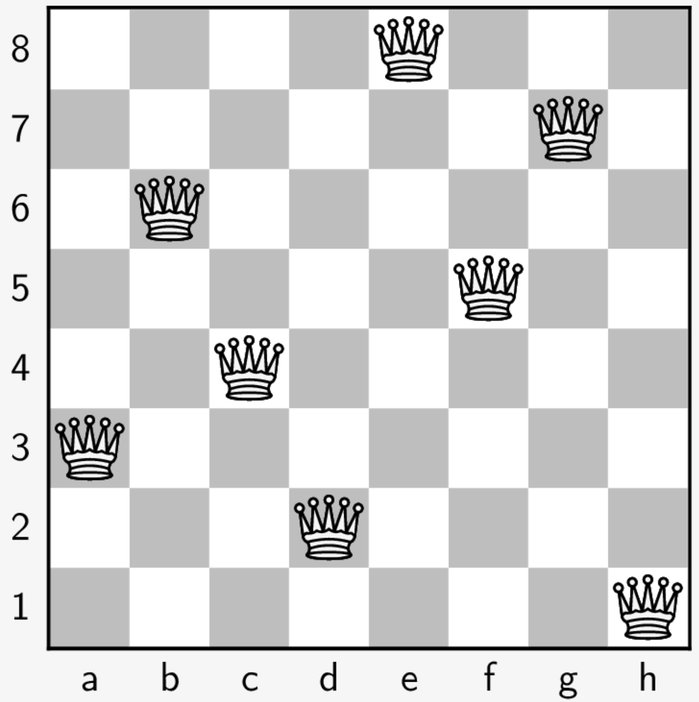
In natural language, another description of a solution is:

- We have a grid of NxN cells
- There is one queen per column of the grid
- Each of these queen cannot attack each other
- On each line, there can be maximum one queen
- On each column, there can be maximum one queen
- On each diagonal, there can be maximum one queen

This can then be formally translated into:

MULTIPLE MODELS FOR ONE PROBLEM

N-Queen Problem



Assigning N Queens on an NxN Board Without Them Attacking Each Other

In natural language, another description of a solution is:

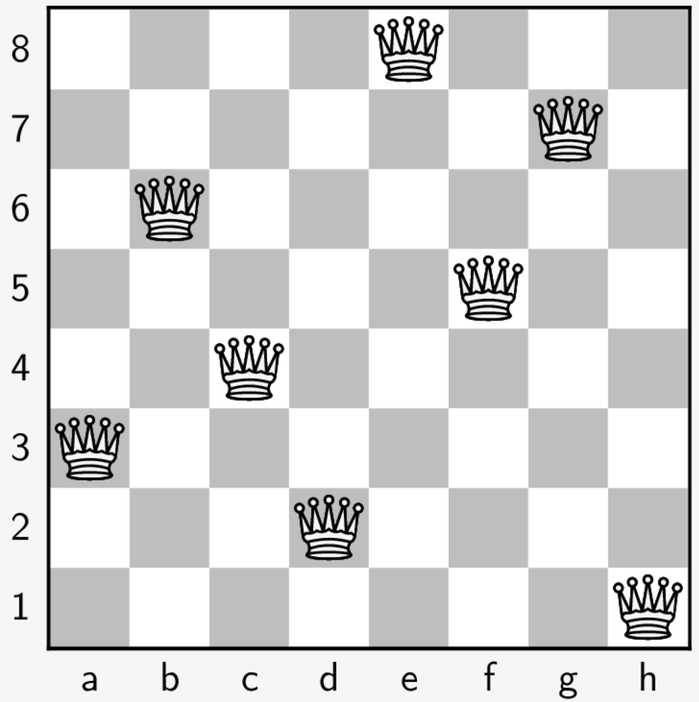
- We have a grid of NxN cells
- There is one queen per column of the grid
- Each of these queen cannot attack each other
- On each line, there can be maximum one queen
- On each column, there can be maximum one queen
- On each diagonal, there can be maximum one queen

This can then be formally translated into:

- Variables queens[I] for i ∈ {1..N}

MULTIPLE MODELS FOR ONE PROBLEM

N-Queen Problem



Assigning N Queens on an NxN Board Without Them Attacking Each Other

In natural language, another description of a solution is:

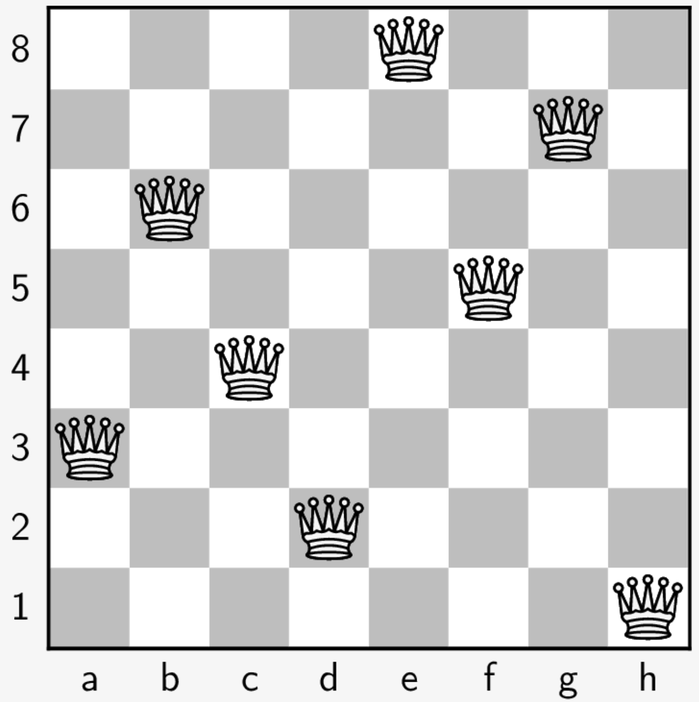
- We have a grid of NxN cells
- There is one queen per column of the grid
- Each of these queen cannot attack each other
- On each line, there can be maximum one queen
- On each column, there can be maximum one queen
- On each diagonal, there can be maximum one queen

This can then be formally translated into:

- Variables queens[I] for i ∈ {1..N}
- queens[i] = {1..N}

MULTIPLE MODELS FOR ONE PROBLEM

N-Queen Problem



Assigning N Queens on an NxN Board Without Them Attacking Each Other

In natural language, another description of a solution is:

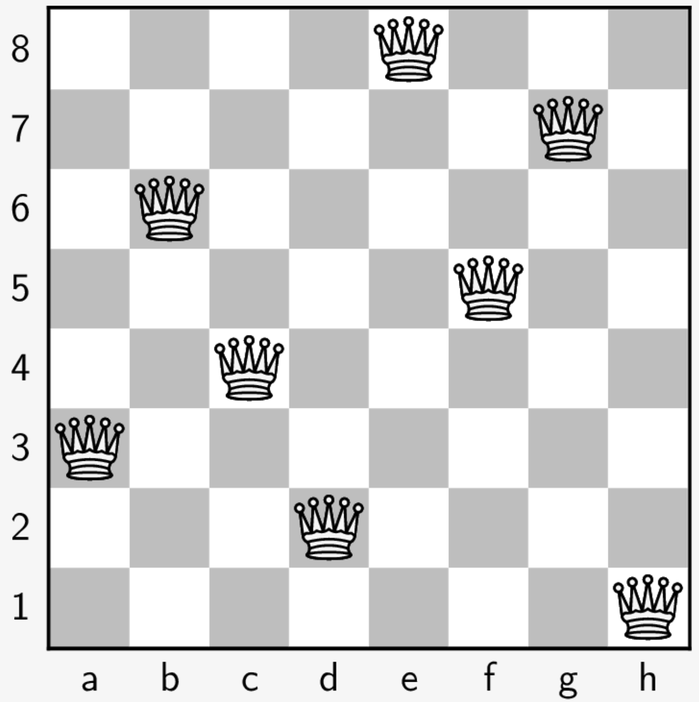
- We have a grid of NxN cells
- There is one queen per column of the grid
- Each of these queen cannot attack each other
- On each line, there can be maximum one queen
- On each column, there can be maximum one queen
- On each diagonal, there can be maximum one queen

This can then be formally translated into:

- Variables queens[I] for i ∈ {1..N}
- queens[i] = {1..N}
- allDifferent(queens)

MULTIPLE MODELS FOR ONE PROBLEM

N-Queen Problem



Assigning N Queens on an NxN Board Without Them Attacking Each Other

In natural language, another description of a solution is:

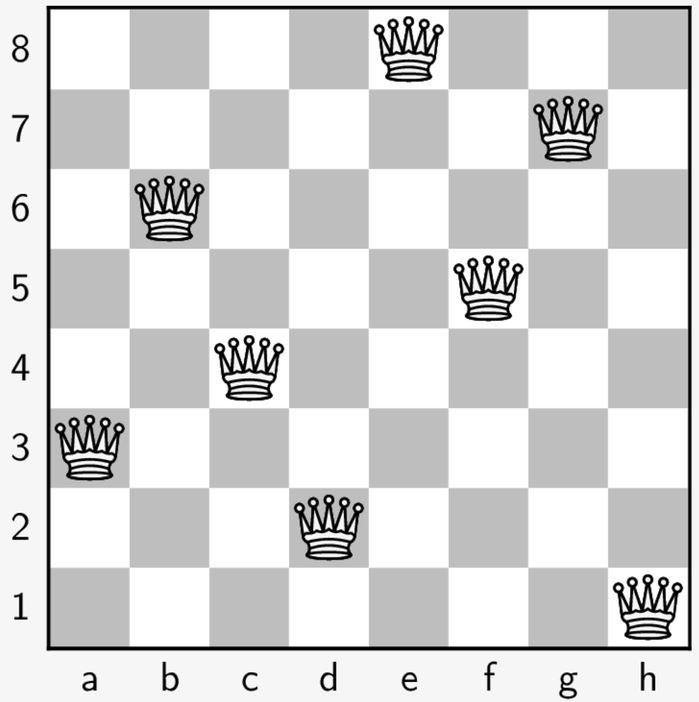
- We have a grid of NxN cells
- There is one queen per column of the grid
- Each of these queen cannot attack each other
- On each line, there can be maximum one queen
- On each column, there can be maximum one queen
- On each diagonal, there can be maximum one queen

This can then be formally translated into:

- Variables queens[I] for i ∈ {1..N}
- queens[i] = {1..N}
- allDifferent(queens)
- allDifferent({queens[i]-i+1 ı i ∈ {1..N}})

MULTIPLE MODELS FOR ONE PROBLEM

N-Queen Problem



Assigning N Queens on an NxN Board Without Them Attacking Each Other

In natural language, another description of a solution is:

- We have a grid of NxN cells
- There is one queen per column of the grid
- Each of these queen cannot attack each other
- On each line, there can be maximum one queen
- On each column, there can be maximum one queen
- On each diagonal, there can be maximum one queen

This can then be formally translated into:

- Variables queens[I] for i ∈ {1..N}
- queens[i] = {1..N}
- allDifferent(queens)
- allDifferent({queens[i]-i+1 ı i ∈ {1..N}})
- allDifferent({queens[i]+i-1 ı i ∈ {1..N}})



MULTIPLE MODELS FOR ONE PROBLEM

MULTIPLE MODELS FOR ONE PROBLEM

Which model choose?

MULTIPLE MODELS FOR ONE PROBLEM

Which model choose?

- On a modelling point of view: The most concise and more interpretable



MULTIPLE MODELS FOR ONE PROBLEM

Which model choose?

- On a modelling point of view: The most concise and more interpretable
- On a solving point of view: It might depends on multiple factors (solvers, propagators, search). Rule of thumb, fewer variables helps!

MULTIPLE MODELS FOR ONE PROBLEM

Which model choose?

- On a modelling point of view: The most concise and more interpretable
- On a solving point of view: It might depends on multiple factors (solvers, propagators, search). Rule of thumb, fewer variables helps!

Modelling Is an Art, for Which Practice and Use Make You Better at It!

Modelling Language



https://docs.minizinc.dev/en/stable/

https://www.minizinc.org/

- Dedicated language



https://docs.minizinc.dev/en/stable/

https://www.minizinc.org/

- Dedicated language
- High-level modelling language



https://docs.minizinc.dev/en/stable/

https://www.minizinc.org/

- Dedicated language
- High-level modelling language
- Automatically compiles into FlatZinc models (low-level), inputed then to CP solvers



https://docs.minizinc.dev/en/stable/

https://www.minizinc.org/

- Dedicated language
- High-level modelling language
- Automatically compiles into FlatZinc models (low-level), inputed then to CP solvers
- Separation between the data and the model



https://docs.minizinc.dev/en/stable/

https://www.minizinc.org/

- Dedicated language
- High-level modelling language
- Automatically compiles into FlatZinc models (low-level), inputed then to CP solvers
- Separation between the data and the model

Listing 2.2.6 Example data file for generalised Sudoku problem (sudoku.dzn) ▶

```
S=3;
start=[|
0, 0, 0, 0, 0, 0, 0, 0, 0|
0, 6, 8, 4, 0, 1, 0, 7, 0|
0, 0, 0, 0, 8, 5, 0, 3, 0|
0, 2, 6, 8, 0, 9, 0, 4, 0|
0, 0, 7, 0, 0, 0, 9, 0, 0|
0, 5, 0, 1, 0, 6, 3, 2, 0|
0, 4, 0, 6, 1, 0, 0, 0, 0|
0, 3, 0, 2, 0, 7, 6, 9, 0|
0, 0, 0, 0, 0, 0, 0, 0, 0|];
```

https://docs.minizinc.dev/en/stable/

https://www.minizinc.org/

- Dedicated language
- High-level modelling language
- Automatically compiles into FlatZinc models (low-level), inputed then to CP solvers
- Separation between the data and the model



https://docs.minizinc.dev/en/stable/

https://www.minizinc.org/

Listing 2.2.5 Model for generalized Sudoku problem (sudoku.mzn) ▶

```
include "alldifferent.mzn";

int: S;
int: N = S * S;
int: digs = ceil(log(10.0,int2float(N))); % digits for output

set of int: PuzzleRange = 1..N;
set of int: SubSquareRange = 1..S;

array[1..N,1..N] of 0..N: start; %% initial board 0 = empty
array[1..N,1..N] of var PuzzleRange: puzzle;

% fill initial board
constraint forall(i,j in PuzzleRange)(
    if start[i,j] > 0 then puzzle[i,j] = start[i,j] else true endif );

% All different in rows
constraint forall (i in PuzzleRange) (
                  alldifferent( [ puzzle[i,j] | j in PuzzleRange ]) );

% All different in columns.
constraint forall (j in PuzzleRange) (
                  alldifferent( [ puzzle[i,j] | i in PuzzleRange ]) );

% All different in sub-squares:
constraint
        forall (a, o in SubSquareRange)(
                alldifferent( [ puzzle[(a-1) *S + a1, (o-1)*S + o1] |
                                        a1, o1 in SubSquareRange ] ) );

solve satisfy;

output  [ show_int(digs,puzzle[i,j]) ++ " " ++
         if j mod S == 0 then " " else "" endif ++
         if j == N then
             if i != N then
                 if i mod S == 0 then "\n\n" else "\n" endif
         else "" endif else "" endif
         | i,j in PuzzleRange ] ++ ["\n"];
```

https://cpmpy.readthedocs.io/en/latest/

https://github.com/CPMpy/cpmpy

- Library in python

https://cpmpy.readthedocs.io/en/latest/

https://github.com/CPMpy/cpmpy

- Library in python
- Designed for easy integration with ML libraries

https://cpmpy.readthedocs.io/en/latest/

https://github.com/CPMpy/cpmpy

- Library in python
- Designed for easy integration with ML libraries
- Aim as the support of multiple solving paradigm and solvers (CP, SAT, SMT, PB,…)

https://cpmpy.readthedocs.io/en/latest/

https://github.com/CPMpy/cpmpy

- Library in python
- Designed for easy integration with ML libraries
- Aim as the support of multiple solving paradigm and solvers (CP, SAT, SMT, PB,…)
- High-level model using globals, nested constraints,…

https://cpmpy.readthedocs.io/en/latest/

https://github.com/CPMpy/cpmpy

- Library in python
- Designed for easy integration with ML libraries
- Aim as the support of multiple solving paradigm and solvers (CP, SAT, SMT, PB,…)
- High-level model using globals, nested constraints,…
- Automatic transformation of the model to the chosen paradigm

https://cpmpy.readthedocs.io/en/latest/

https://github.com/CPMpy/cpmpy

CPMPY

- Library in python
- Designed for easy integration with ML libraries
- Aim as the support of multiple solving paradigm and solvers (CP, SAT, SMT, PB,…)
- High-level model using globals, nested constraints,…
- Automatic transformation of the model to the chosen paradigm

https://cpmpy.readthedocs.io/en/latest/

https://github.com/CPMpy/cpmpy

```
#!/usr/bin/python3
"""
Sudoku problem in CPMpy
"""

# load the libraries
import numpy as np
from cpmpy import *

e = 0 # value for empty cells
given = np.array([
    [e, e, e,  2, e, 5,  e, e, e],
    [e, 9, e,  e, e, e,  7, 3, e],
    [e, e, 2,  e, e, 9,  e, 6, e],

    [2, e, e,  e, e, e,  4, e, 9],
    [e, e, e,  e, 7, e,  e, e, e],
    [6, e, 9,  e, e, e,  e, e, 1],

    [e, 8, e,  4, e, e,  1, e, e],
    [e, 6, 3,  e, e, e,  e, 8, e],
    [e, e, e,  6, e, 8,  e, e, e]])


# Variables
puzzle = intvar(1,9, shape=given.shape, name="puzzle")
```

CPMPY

- Library in python
- Designed for easy integration with ML libraries
- Aim as the support of multiple solving paradigm and solvers (CP, SAT, SMT, PB,…)
- High-level model using globals, nested constraints,…
- Automatic transformation of the model to the chosen paradigm

https://cpmpy.readthedocs.io/en/latest/

https://github.com/CPMpy/cpmpy

```
#!/usr/bin/python3
"""
Sudoku problem
"""

# load the lib
import numpy a
from cpmpy imp

e = 0 # value
given = np.arr
    [e, e, e,
    [e, 9, e,
    [e, e, 2,

    [2, e, e,
    [e, e, e,
    [6, e, 9,

    [e, 8, e,
    [e, 6, 3,
    [e, e, e,


# Variables
puzzle = intva
```

```
model = Model(
    # Constraints on values (cells that are not empty)
    puzzle[given!=e] == given[given!=e], # numpy's indexing, vectorized equality
    # Constraints on rows and columns
    [AllDifferent(row) for row in puzzle],
    [AllDifferent(col) for col in puzzle.T], # numpy's Transpose
)

# Constraints on blocks
for i in range(0,9, 3):
    for j in range(0,9, 3):
        model += AllDifferent(puzzle[i:i+3, j:j+3]) # python's indexing


# Solve and print
if model.solve():
    #print(puzzle.value())
    # pretty print, mark givens with *
    out = ""
    for r in range(0,9):
        for c in range(0,9):
            out += str(puzzle[r,c].value())
            out += '* ' if given[r,c] else '  '
            if (c+1) % 3 == 0 and c != 8: # end of block
                out += '| '
        out += '\n'
        if (r+1) % 3 == 0 and r != 8: # end of block
            out += ('-'*9)+'+-'+('-'*9)+'+'+('-'*9)+'\n'
    print(out)
else:
    print("No solution found")
```



https://docs.minizinc.dev/en/stable/

https://www.minizinc.org/

- XML-based files



https://docs.minizinc.dev/en/stable/

https://www.minizinc.org/

- XML-based files
- High-level modelling language



https://docs.minizinc.dev/en/stable/

https://www.minizinc.org/

- XML-based files
- High-level modelling language
- Keep the structure high-level for the solvers



https://docs.minizinc.dev/en/stable/

https://www.minizinc.org/

XCSP3

- XML-based files
- High-level modelling language
- Keep the structure high-level for the solvers



```xml
<instance format="XCSP3" type="CSP">
  <variables>
    <array id="x" size="[9][9]"> 1..9 </array>
  </variables>
  <constraints>
    <group class="rows">
      <allDifferent> %... </allDifferent>
      <args> x[0][] </args>
      <args> x[1][] </args>
      <args> x[2][] </args>
      <args> x[3][] </args>
      <args> x[4][] </args>
      <args> x[5][] </args>
      <args> x[6][] </args>
      <args> x[7][] </args>
      <args> x[8][] </args>
    </group>
    <group class="columns">
      <allDifferent> %... </allDifferent>
      <args> x[][0] </args>
      <args> x[][1] </args>
      <args> x[][2] </args>
      <args> x[][3] </args>
      <args> x[][4] </args>
      <args> x[][5] </args>
      <args> x[][6] </args>
      <args> x[][7] </args>
      <args> x[][8] </args>
    </group>
```

https://docs.minizinc.dev/en/stable/

https://www.minizinc.org/

XCSP3

- XML-based files
- High-level modelling language
- Keep the structure high-level for the solvers

```xml
<instance format="XCSP3" type="CSP">
  <variables>
    <array id="x" size="[9][9]"> 1..9 </array>
  </variables>
  <constraints>
    <group class="rows">
      <allDifferent> %... </allDifferent>
      <args> x[0][] </args>
      <args> x
      <args> x
      <args> x
      <args> x
      <args> x
      <args> x
      <args> x
      <args> x
    </group>
    <group cla
      <allDiff
      <args> x
      <args> x
      <args> x
      <args> x
      <args> x
      <args> x
      <args> x
      <args> x
      <args> x
    </group>
```

```xml
    <group class="blocks">
      <allDifferent> %... </allDifferent>
      <args> x[0..2][0..2] </args>
      <args> x[0..2][3..5] </args>
      <args> x[0..2][6..8] </args>
      <args> x[3..5][0..2] </args>
      <args> x[3..5][3..5] </args>
      <args> x[3..5][6..8] </args>
      <args> x[6..8][0..2] </args>
      <args> x[6..8][3..5] </args>
      <args> x[6..8][6..8] </args>
    </group>
    <instantiation class="clues">
      <list> x[0][1] x[0][6..8] x[1][2] x[1][5] x[1][7..8] x[2][2] x[2][5]
       x[2][8] x[3][1] x[3][4] x[3][6..7] x[4][1] x[4][4] x[4][7] x[5][1..2]
       x[5][4] x[5][7] x[6][0] x[6][3] x[6][6] x[7][0..1] x[7][3] x[7][6]
       x[8][0..2] x[8][7] </list>
      <values> 4 1 7 9 2 8 5 4 6 5 8 8 7 9 1 5 9 3 1 9 6 4 3 4 7 5 7 1 2 9 2
       8 6 </values>
    </instantiation>
  </constraints>
</instance>
```

https://docs.minizinc.dev/en/stable/

https://www.minizinc.org/



https://docs.minizinc.dev/en/stable/

https://www.minizinc.org/

- Python library



https://docs.minizinc.dev/en/stable/

https://www.minizinc.org/

- Python library
- High-level modelling language



https://docs.minizinc.dev/en/stable/

https://www.minizinc.org/

- Python library
- High-level modelling language
- Compilation into XCSP3 files



https://docs.minizinc.dev/en/stable/

https://www.minizinc.org/

- Python library
- High-level modelling language
- Compilation into XCSP3 files
- Direct integration with solvers Ace and Choco



https://docs.minizinc.dev/en/stable/

https://www.minizinc.org/

PYCSP3

- Python library
- High-level modelling language
- Compilation into XCSP3 files
- Direct integration with solvers Ace and Choco




```python
from pycsp3 import *

clues = data  # if not 0, clues[i][j] is a value imposed at row i and col j

# x[i][j] is the value at row i and col j
x = VarArray(size=[9, 9], dom=range(1, 10))

satisfy(
   # imposing distinct values on each row and each column
   AllDifferent(x, matrix=True),

   # imposing distinct values on each block  tag(blocks)
   [AllDifferent(x[i:i + 3, j:j + 3]) for i in [0, 3, 6] for j in [0, 3, 6]],

   # imposing clues  tag(clues)
   [x[i][j] == clues[i][j] for i in range(9) for j in range(9) if clues and clues[i][j] > 0]
)
```

https://docs.minizinc.dev/en/stable/

https://www.minizinc.org/

Variables and Domains

Boolean Variables

BOOLEAN VARIABLES

BOOLEAN VARIABLES

X is a boolean variable:

BOOLEAN VARIABLES

X is a boolean variable:

- If dom(X)={false, true}

BOOLEAN VARIABLES

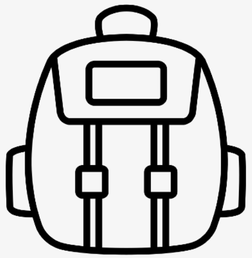
X is a boolean variable:

- If dom(X)={false, true}
- or dom(X) = {0,1} (0=false, 1=true)

BOOLEAN VARIABLES

X is a boolean variable:

- If dom(X)={false, true}
- or dom(X) = {0,1} (0=false, 1=true)

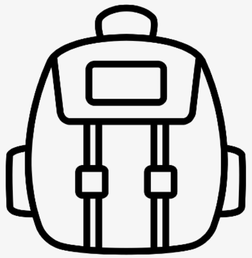


Knapsack problem: given a knapsack of capacity C, and the list of objects o1, o2,…, oN. Each object is associated with a weight w[i] and a utility u[i]. What objects should we select to maximise the total utility?

BOOLEAN VARIABLES

X is a boolean variable:

- If dom(X)={false, true}
- or dom(X) = {0,1} (0=false, 1=true)



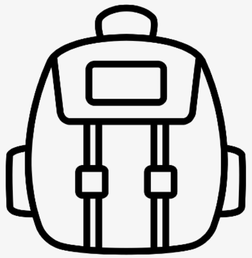
Knapsack problem: given a knapsack of capacity C, and the list of objects o1, o2,…, oN. Each object is associated with a weight w[i] and a utility u[i]. What objects should we select to maximise the total utility?

Declarative description of a solution:

BOOLEAN VARIABLES

X is a boolean variable:

- If dom(X)={false, true}
- or dom(X) = {0,1} (0=false, 1=true)



Knapsack problem: given a knapsack of capacity C, and the list of objects o1, o2,…, oN. Each object is associated with a weight w[i] and a utility u[i]. What objects should we select to maximise the total utility?

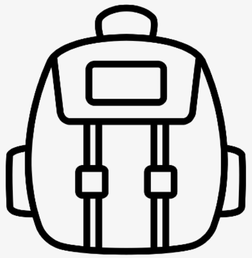
Declarative description of a solution:

- We have a list of N items

BOOLEAN VARIABLES

X is a boolean variable:

- If dom(X)={false, true}
- or dom(X) = {0,1} (0=false, 1=true)



Knapsack problem: given a knapsack of capacity C, and the list of objects o1, o2,…, oN. Each object is associated with a weight w[i] and a utility u[i]. What objects should we select to maximise the total utility?

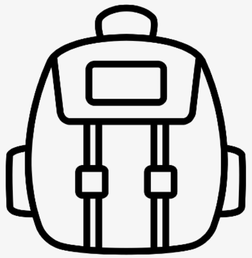
Declarative description of a solution:

- We have a list of N items
- Each item can be in or out of the bag

BOOLEAN VARIABLES

X is a boolean variable:

- If dom(X)={false, true}
- or dom(X) = {0,1} (0=false, 1=true)



Knapsack problem: given a knapsack of capacity C, and the list of objects o1, o2,…, oN. Each object is associated with a weight w[i] and a utility u[i]. What objects should we select to maximise the total utility?

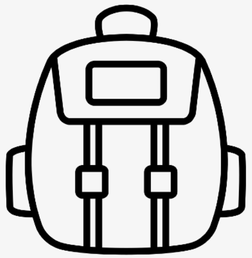
Declarative description of a solution:

- We have a list of N items
- Each item can be in or out of the bag
- The total weight of the items in the bag should be lower or equal to the capacity

BOOLEAN VARIABLES

X is a boolean variable:

- If dom(X)={false, true}
- or dom(X) = {0,1} (0=false, 1=true)



Knapsack problem: given a knapsack of capacity C, and the list of objects o1, o2,…, oN. Each object is associated with a weight w[i] and a utility u[i]. What objects should we select to maximise the total utility?

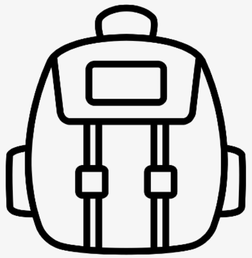
Declarative description of a solution:

- We have a list of N items
- Each item can be in or out of the bag
- The total weight of the items in the bag should be lower or equal to the capacity
- We can also compute the total utility of the items in the bag

BOOLEAN VARIABLES

X is a boolean variable:

- If dom(X)={false, true}
- or dom(X) = {0,1} (0=false, 1=true)



Knapsack problem: given a knapsack of capacity C, and the list of objects o1, o2,…, oN. Each object is associated with a weight w[i] and a utility u[i]. What objects should we select to maximise the total utility?

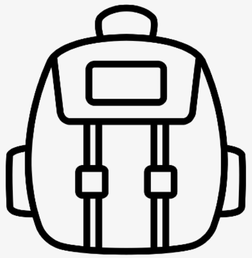
Declarative description of a solution:

- We have a list of N items
- Each item can be in or out of the bag
- The total weight of the items in the bag should be lower or equal to the capacity
- We can also compute the total utility of the items in the bag

BOOLEAN VARIABLES

X is a boolean variable:

- If dom(X)={false, true}
- or dom(X) = {0,1} (0=false, 1=true)



Knapsack problem: given a knapsack of capacity C, and the list of objects o1, o2,…, oN. Each object is associated with a weight w[i] and a utility u[i]. What objects should we select to maximise the total utility?

Declarative description of a solution:

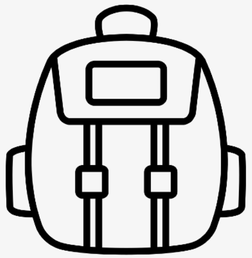
- We have a list of N items
- Each item can be in or out of the bag
- The total weight of the items in the bag should be lower or equal to the capacity
- We can also compute the total utility of the items in the bag

Declarative description of the best solution:

BOOLEAN VARIABLES

X is a boolean variable:

- If dom(X)={false, true}
- or dom(X) = {0,1} (0=false, 1=true)



Knapsack problem: given a knapsack of capacity C, and the list of objects o1, o2,…, oN. Each object is associated with a weight w[i] and a utility u[i]. What objects should we select to maximise the total utility?

Declarative description of a solution:

- We have a list of N items
- Each item can be in or out of the bag
- The total weight of the items in the bag should be lower or equal to the capacity
- We can also compute the total utility of the items in the bag

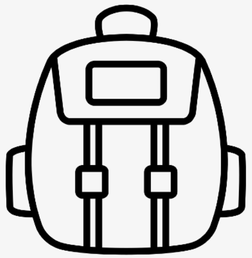
Declarative description of the best solution:

- A solution, but with the highest possible total utility

BOOLEAN VARIABLES

X is a boolean variable:

- If dom(X)={false, true}
- or dom(X) = {0,1} (0=false, 1=true)



Knapsack problem: given a knapsack of capacity C, and the list of objects o1, o2,…, oN. Each object is associated with a weight w[i] and a utility u[i]. What objects should we select to maximise the total utility?

Declarative description of a solution:

- We have a list of N items
- Each item can be in or out of the bag
- The total weight of the items in the bag should be lower or equal to the capacity
- We can also compute the total utility of the items in the bag

This can then be formally translated into:

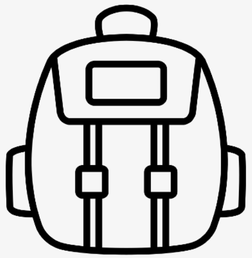
Declarative description of the best solution:

- A solution, but with the highest possible total utility

BOOLEAN VARIABLES

X is a boolean variable:

- If dom(X)={false, true}
- or dom(X) = {0,1} (0=false, 1=true)



Knapsack problem: given a knapsack of capacity C, and the list of objects o1, o2,…, oN. Each object is associated with a weight w[i] and a utility u[i]. What objects should we select to maximise the total utility?

Declarative description of a solution:

- We have a list of N items
- Each item can be in or out of the bag
- The total weight of the items in the bag should be lower or equal to the capacity
- We can also compute the total utility of the items in the bag

Declarative description of the best solution:

- A solution, but with the highest possible total utility

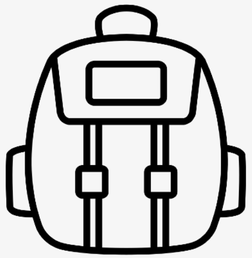
This can then be formally translated into:

- Variables X[i] for i ∈ {1..N}

BOOLEAN VARIABLES

X is a boolean variable:

- If dom(X)={false, true}
- or dom(X) = {0,1} (0=false, 1=true)



Knapsack problem: given a knapsack of capacity C, and the list of objects o1, o2,…, oN. Each object is associated with a weight w[i] and a utility u[i]. What objects should we select to maximise the total utility?

Declarative description of a solution:

- We have a list of N items
- Each item can be in or out of the bag
- The total weight of the items in the bag should be lower or equal to the capacity
- We can also compute the total utility of the items in the bag

Declarative description of the best solution:

- A solution, but with the highest possible total utility

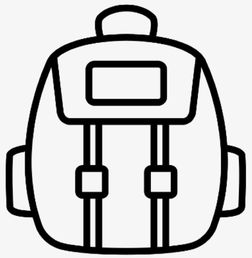
This can then be formally translated into:

- Variables X[i] for i ∈ {1..N}
- dom(X[i]) = {0,1}

BOOLEAN VARIABLES

X is a boolean variable:

- If dom(X)={false, true}
- or dom(X) = {0,1} (0=false, 1=true)



Knapsack problem: given a knapsack of capacity C, and the list of objects o1, o2,…, oN. Each object is associated with a weight w[i] and a utility u[i]. What objects should we select to maximise the total utility?

Declarative description of a solution:

- We have a list of N items
- Each item can be in or out of the bag
- The total weight of the items in the bag should be lower or equal to the capacity
- We can also compute the total utility of the items in the bag

Declarative description of the best solution:

- A solution, but with the highest possible total utility

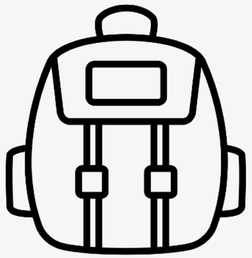
This can then be formally translated into:

- Variables X[i] for i ∈ {1..N}
- dom(X[i]) = {0,1}
- sum({w[i]*X[i]=1 ı for i ∈ {1..N}}) <= C

BOOLEAN VARIABLES

X is a boolean variable:

- If dom(X)={false, true}
- or dom(X) = {0,1} (0=false, 1=true)



Knapsack problem: given a knapsack of capacity C, and the list of objects o1, o2,…, oN. Each object is associated with a weight w[i] and a utility u[i]. What objects should we select to maximise the total utility?

Declarative description of a solution:

- We have a list of N items
- Each item can be in or out of the bag
- The total weight of the items in the bag should be lower or equal to the capacity
- We can also compute the total utility of the items in the bag

Declarative description of the best solution:

- A solution, but with the highest possible total utility

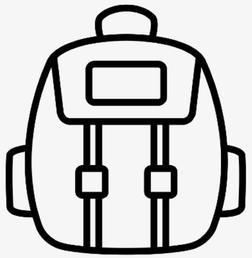
This can then be formally translated into:

- Variables X[i] for i ∈ {1..N}
- dom(X[i]) = {0,1}
- sum({w[i]*X[i]=1 ı for i ∈ {1..N}}) <= C
- sum({u[i]*X[i]=1 ı for i ∈ {1..N}})

BOOLEAN VARIABLES

X is a boolean variable:

- If dom(X)={false, true}
- or dom(X) = {0,1} (0=false, 1=true)



Knapsack problem: given a knapsack of capacity C, and the list of objects o1, o2,…, oN. Each object is associated with a weight w[i] and a utility u[i]. What objects should we select to maximise the total utility?

Declarative description of a solution:

- We have a list of N items
- Each item can be in or out of the bag
- The total weight of the items in the bag should be lower or equal to the capacity
- We can also compute the total utility of the items in the bag

Declarative description of the best solution:

- A solution, but with the highest possible total utility

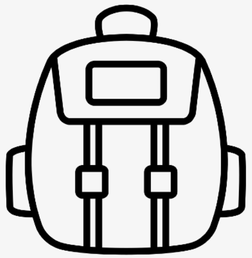
This can then be formally translated into:

- Variables X[i] for i ∈ {1..N}
- dom(X[i]) = {0,1}
- sum({w[i]*X[i]=1 ı for i ∈ {1..N}}) <= C
- sum({u[i]*X[i]=1 ı for i ∈ {1..N}})

BOOLEAN VARIABLES

X is a boolean variable:

- If dom(X)={false, true}
- or dom(X) = {0,1} (0=false, 1=true)



Knapsack problem: given a knapsack of capacity C, and the list of objects o1, o2,…, oN. Each object is associated with a weight w[i] and a utility u[i]. What objects should we select to maximise the total utility?

Declarative description of a solution:

- We have a list of N items
- Each item can be in or out of the bag
- The total weight of the items in the bag should be lower or equal to the capacity
- We can also compute the total utility of the items in the bag

Declarative description of the best solution:

- A solution, but with the highest possible total utility

This can then be formally translated into:

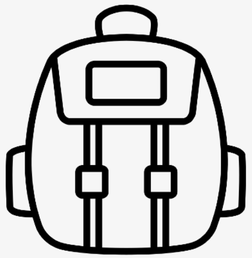
- Variables X[i] for i ∈ {1..N}
- dom(X[i]) = {0,1}
- sum({w[i]*X[i]=1 ı for i ∈ {1..N}}) <= C
- sum({u[i]*X[i]=1 ı for i ∈ {1..N}})

And to get the best solution

BOOLEAN VARIABLES

X is a boolean variable:

- If dom(X)={false, true}
- or dom(X) = {0,1} (0=false, 1=true)



Knapsack problem: given a knapsack of capacity C, and the list of objects o1, o2,…, oN. Each object is associated with a weight w[i] and a utility u[i]. What objects should we select to maximise the total utility?

Declarative description of a solution:

- We have a list of N items
- Each item can be in or out of the bag
- The total weight of the items in the bag should be lower or equal to the capacity
- We can also compute the total utility of the items in the bag

Declarative description of the best solution:

- A solution, but with the highest possible total utility

This can then be formally translated into:

- Variables X[i] for i ∈ {1..N}
- dom(X[i]) = {0,1}
- sum({w[i]*X[i]=1 ı for i ∈ {1..N}}) <= C
- sum({u[i]*X[i]=1 ı for i ∈ {1..N}})

And to get the best solution

- Maximise sum({u[i]*X[i]=1 ı for i ∈ {1..N}})

Integer Variables



INTEGER VARIABLES

INTEGER VARIABLES

X is an integer variable:

INTEGER VARIABLES

X is an integer variable:

- If dom(X) $\subseteq \mathbb{Z}$

INTEGER VARIABLES

X is an integer variable:

- If dom(X) $\subseteq \mathbb{Z}$
- CP solvers works mostly with finite domains

INTEGER VARIABLES

X is an integer variable:

- If dom(X) $\subseteq \mathbb{Z}$
- CP solvers works mostly with finite domains

```
  SEND
+ MORE
------
 MONEY
```

Classical “Send More Money” verbal arithmetic problem: each letter is to be replaced with a distinct digit such that the equation holds. Leading digit should not be 0.

INTEGER VARIABLES

X is an integer variable:

- If dom(X) $\subseteq \mathbb{Z}$
- CP solvers works mostly with finite domains

Declarative description of a solution:

```
  SEND
+ MORE
------
 MONEY
```

Classical “Send More Money” verbal arithmetic problem: each letter is to be replaced with a distinct digit such that the equation holds. Leading digit should not be 0.

INTEGER VARIABLES

X is an integer variable:

- If dom(X) $\subseteq \mathbb{Z}$
- CP solvers works mostly with finite domains

Declarative description of a solution:

- We have 8 letters: D, E, M, N, O, R, S, Y

```
  SEND
+ MORE
------
 MONEY
```

Classical “Send More Money” verbal arithmetic problem: each letter is to be replaced with a distinct digit such that the equation holds. Leading digit should not be 0.

INTEGER VARIABLES

X is an integer variable:

- If dom(X) $\subseteq \mathbb{Z}$
- CP solvers works mostly with finite domains

Declarative description of a solution:

- We have 8 letters: D, E, M, N, O, R, S, Y
- We need to assign a digit (from 0 to 9) to each of them

```
 SEND
+MORE
-----
MONEY
```

Classical “Send More Money” verbal arithmetic problem: each letter is to be replaced with a distinct digit such that the equation holds. Leading digit should not be 0.

INTEGER VARIABLES

X is an integer variable:

- If dom(X) $\subseteq \mathbb{Z}$
- CP solvers works mostly with finite domains

Declarative description of a solution:

- We have 8 letters: D, E, M, N, O, R, S, Y
- We need to assign a digit (from 0 to 9) to each of them
- All the letters should have a different digit

```
  SEND
+ MORE
------
 MONEY
```

Classical “Send More Money” verbal arithmetic problem: each letter is to be replaced with a distinct digit such that the equation holds. Leading digit should not be 0.

INTEGER VARIABLES

X is an integer variable:

- If dom(X) $\subseteq \mathbb{Z}$
- CP solvers works mostly with finite domains

Declarative description of a solution:

- We have 8 letters: D, E, M, N, O, R, S, Y
- We need to assign a digit (from 0 to 9) to each of them
- All the letters should have a different digit
- Letters representing the leading digits (M and S) should not be 0

```
  SEND
+ MORE
------
 MONEY
```

Classical “Send More Money” verbal arithmetic problem: each letter is to be replaced with a distinct digit such that the equation holds. Leading digit should not be 0.

INTEGER VARIABLES

X is an integer variable:

- If dom(X) $\subseteq \mathbb{Z}$
- CP solvers works mostly with finite domains

```
  SEND
+ MORE
------
 MONEY
```

Classical “Send More Money” verbal arithmetic problem: each letter is to be replaced with a distinct digit such that the equation holds. Leading digit should not be 0.

Declarative description of a solution:

- We have 8 letters: D, E, M, N, O, R, S, Y
- We need to assign a digit (from 0 to 9) to each of them
- All the letters should have a different digit
- Letters representing the leading digits (M and S) should not be 0
- The sum should hold once we have replaced the letters by the digits

INTEGER VARIABLES

X is an integer variable:

- If dom(X) $\subseteq \mathbb{Z}$
- CP solvers works mostly with finite domains

Declarative description of a solution:

- We have 8 letters: D, E, M, N, O, R, S, Y
- We need to assign a digit (from 0 to 9) to each of them
- All the letters should have a different digit
- Letters representing the leading digits (M and S) should not be 0
- The sum should hold once we have replaced the letters by the digits

```
  SEND
+ MORE
------
 MONEY
```

Classical “Send More Money” verbal arithmetic problem: each letter is to be replaced with a distinct digit such that the equation holds. Leading digit should not be 0.

This can then be formally translated into:

INTEGER VARIABLES

X is an integer variable:

- If dom(X) $\subseteq \mathbb{Z}$
- CP solvers works mostly with finite domains

```
  SEND
+ MORE
------
 MONEY
```

Classical “Send More Money” verbal arithmetic problem: each letter is to be replaced with a distinct digit such that the equation holds. Leading digit should not be 0.

Declarative description of a solution:

- We have 8 letters: D, E, M, N, O, R, S, Y
- We need to assign a digit (from 0 to 9) to each of them
- All the letters should have a different digit
- Letters representing the leading digits (M and S) should not be 0
- The sum should hold once we have replaced the letters by the digits

This can then be formally translated into:

- Variables X[i] for i $\in$ {D, E, M, N, O, R, S, Y}

INTEGER VARIABLES

X is an integer variable:

- If dom(X) $\subseteq \mathbb{Z}$
- CP solvers works mostly with finite domains

```
  SEND
+ MORE
------
 MONEY
```

Classical “Send More Money” verbal arithmetic problem: each letter is to be replaced with a distinct digit such that the equation holds. Leading digit should not be 0.

Declarative description of a solution:

- We have 8 letters: D, E, M, N, O, R, S, Y
- We need to assign a digit (from 0 to 9) to each of them
- All the letters should have a different digit
- Letters representing the leading digits (M and S) should not be 0
- The sum should hold once we have replaced the letters by the digits

This can then be formally translated into:

- Variables X[i] for i $\in$ {D, E, M, N, O, R, S, Y}
- dom(X[i]) = {0..9}

INTEGER VARIABLES

X is an integer variable:

- If dom(X) $\subseteq \mathbb{Z}$
- CP solvers works mostly with finite domains

```
  SEND
+ MORE
------
 MONEY
```

Classical “Send More Money” verbal arithmetic problem: each letter is to be replaced with a distinct digit such that the equation holds. Leading digit should not be 0.

Declarative description of a solution:

- We have 8 letters: D, E, M, N, O, R, S, Y
- We need to assign a digit (from 0 to 9) to each of them
- All the letters should have a different digit
- Letters representing the leading digits (M and S) should not be 0
- The sum should hold once we have replaced the letters by the digits

This can then be formally translated into:

- Variables X[i] for i $\in$ {D, E, M, N, O, R, S, Y}
- dom(X[i]) = {0..9}
- AllDifferent(X)

INTEGER VARIABLES

X is an integer variable:

- If dom(X) $\subseteq \mathbb{Z}$
- CP solvers works mostly with finite domains

```
  SEND
+ MORE
------
 MONEY
```

Classical “Send More Money” verbal arithmetic problem: each letter is to be replaced with a distinct digit such that the equation holds. Leading digit should not be 0.

Declarative description of a solution:

- We have 8 letters: D, E, M, N, O, R, S, Y
- We need to assign a digit (from 0 to 9) to each of them
- All the letters should have a different digit
- Letters representing the leading digits (M and S) should not be 0
- The sum should hold once we have replaced the letters by the digits

This can then be formally translated into:

- Variables X[i] for i ∈ {D, E, M, N, O, R, S, Y}
- dom(X[i]) = {0..9}
- AllDifferent(X)
- X[M]!=0, X[S]!=0

INTEGER VARIABLES

X is an integer variable:

- If dom(X) $\subseteq \mathbb{Z}$
- CP solvers works mostly with finite domains

```
  SEND
+ MORE
------
 MONEY
```

Classical “Send More Money” verbal arithmetic problem: each letter is to be replaced with a distinct digit such that the equation holds. Leading digit should not be 0.

Declarative description of a solution:

- We have 8 letters: D, E, M, N, O, R, S, Y
- We need to assign a digit (from 0 to 9) to each of them
- All the letters should have a different digit
- Letters representing the leading digits (M and S) should not be 0
- The sum should hold once we have replaced the letters by the digits

This can then be formally translated into:

- Variables X[i] for i ∈ {D, E, M, N, O, R, S, Y}
- dom(X[i]) = {0..9}
- AllDifferent(X)
- X[M]!=0, X[S]!=0
- X[S]*1000+X[E]*100+X[N]*10+X[D] + X[M]*1000+X[O]*100+X[R]*10+X[E] = X[M]*10000+X[O]*1000+X[N]*100+X[E]*10+X[Y]

Other Kind of Variables



ENUMERATED NON-INTEGER VALUE

ENUMERATED NON-INTEGER VALUE

X is a variable of enumerated non-integer value:



ENUMERATED NON-INTEGER VALUE

X is a variable of enumerated non-integer value:

- If dom(X) = {value1, value2, value3}



ENUMERATED NON-INTEGER VALUE

X is a variable of enumerated non-integer value:

- If dom(X) = {value1, value2, value3}
- Map it to integers: value1 = 1, value2 = 2,…



ENUMERATED NON-INTEGER VALUE

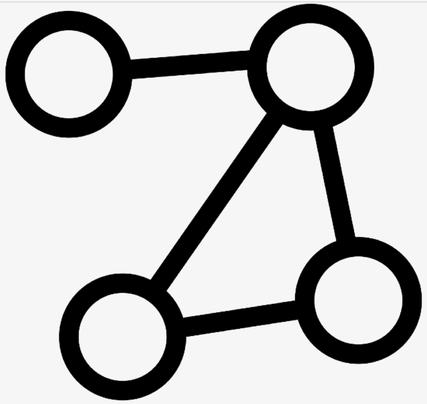
X is a variable of enumerated non-integer value:

- If dom(X) = {value1, value2, value3}
- Map it to integers: value1 = 1, value2 = 2,…
- Some arithmetic constraints might not make sense anymore (e.g., X < Y, X+Y = Z)

ENUMERATED NON-INTEGER VALUE

X is a variable of enumerated non-integer value:

- If dom(X) = {value1, value2, value3}
- Map it to integers: value1 = 1, value2 = 2,…
- Some arithmetic constraints might not make sense anymore (e.g., X < Y, X+Y = Z)

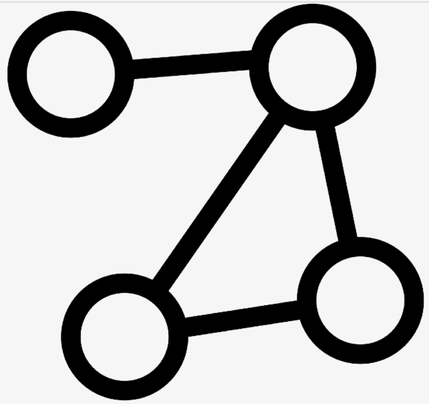


Map colouring (Graph colouring): Given a map, how can we colour the countries, using K colours, such that not two countries have the same colour

ENUMERATED NON-INTEGER VALUE

X is a variable of enumerated non-integer value:

- If dom(X) = {value1, value2, value3}
- Map it to integers: value1 = 1, value2 = 2,…
- Some arithmetic constraints might not make sense anymore (e.g., X < Y, X+Y = Z)



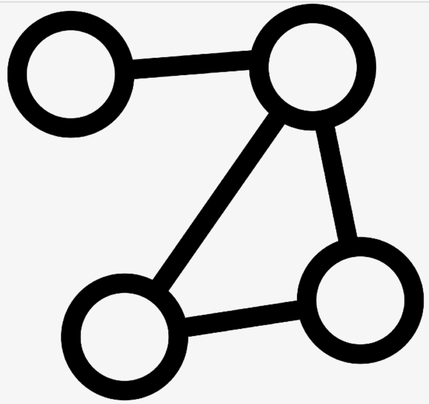
Map colouring (Graph colouring): Given a map, how can we colour the countries, using K colours, such that not two countries have the same colour

Declarative description of a solution:

ENUMERATED NON-INTEGER VALUE

X is a variable of enumerated non-integer value:

- If dom(X) = {value1, value2, value3}
- Map it to integers: value1 = 1, value2 = 2,…
- Some arithmetic constraints might not make sense anymore (e.g., X < Y, X+Y = Z)



Map colouring (Graph colouring): Given a map, how can we colour the countries, using K colours, such that not two countries have the same colour

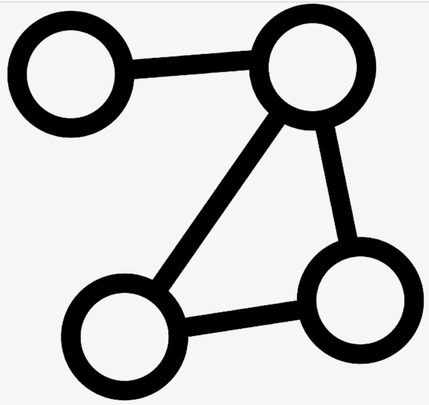
Declarative description of a solution:

- We have K colours: green, blue, yellow,…

ENUMERATED NON-INTEGER VALUE

X is a variable of enumerated non-integer value:

- If dom(X) = {value1, value2, value3}
- Map it to integers: value1 = 1, value2 = 2,…
- Some arithmetic constraints might not make sense anymore (e.g., X < Y, X+Y = Z)



Map colouring (Graph colouring): Given a map, how can we colour the countries, using K colours, such that not two countries have the same colour

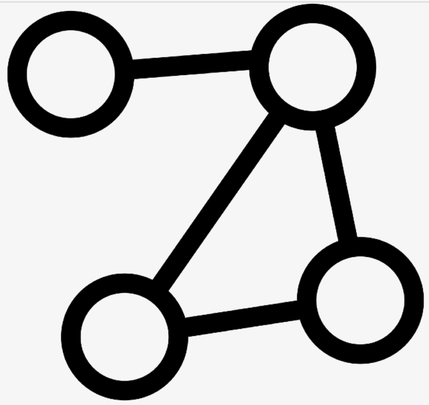
Declarative description of a solution:

- We have K colours: green, blue, yellow,…
- If countryA shares a border with countryB, then their colours are different

ENUMERATED NON-INTEGER VALUE

X is a variable of enumerated non-integer value:

- If dom(X) = {value1, value2, value3}
- Map it to integers: value1 = 1, value2 = 2,…
- Some arithmetic constraints might not make sense anymore (e.g., X < Y, X+Y = Z)



Map colouring (Graph colouring): Given a map, how can we colour the countries, using K colours, such that not two countries have the same colour

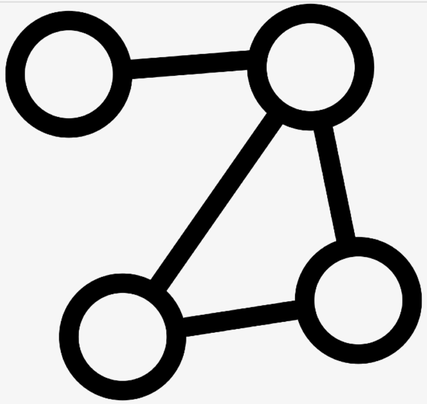
Declarative description of a solution:

- We have K colours: green, blue, yellow,…
- If countryA shares a border with countryB, then their colours are different

ENUMERATED NON-INTEGER VALUE

X is a variable of enumerated non-integer value:

- If dom(X) = {value1, value2, value3}
- Map it to integers: value1 = 1, value2 = 2,…
- Some arithmetic constraints might not make sense anymore (e.g., X < Y, X+Y = Z)



Map colouring (Graph colouring): Given a map, how can we colour the countries, using K colours, such that not two countries have the same colour

Declarative description of a solution:

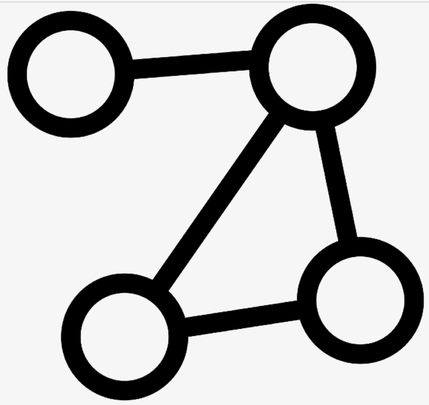
- We have K colours: green, blue, yellow,…
- If countryA shares a border with countryB, then their colours are different

This can then be formally translated into:

ENUMERATED NON-INTEGER VALUE

X is a variable of enumerated non-integer value:

- If dom(X) = {value1, value2, value3}
- Map it to integers: value1 = 1, value2 = 2,…
- Some arithmetic constraints might not make sense anymore (e.g., X < Y, X+Y = Z)



Map colouring (Graph colouring): Given a map, how can we colour the countries, using K colours, such that not two countries have the same colour

Declarative description of a solution:

- We have K colours: green, blue, yellow,…
- If countryA shares a border with countryB, then their colours are different

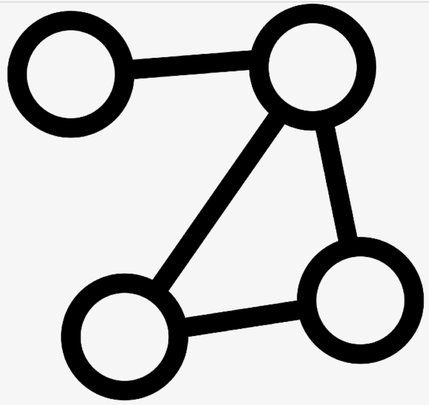
This can then be formally translated into:

- Variables X[i] for i ∈ {countryA, countryB,…}

ENUMERATED NON-INTEGER VALUE

X is a variable of enumerated non-integer value:

- If dom(X) = {value1, value2, value3}
- Map it to integers: value1 = 1, value2 = 2,…
- Some arithmetic constraints might not make sense anymore (e.g., X < Y, X+Y = Z)



Map colouring (Graph colouring): Given a map, how can we colour the countries, using K colours, such that not two countries have the same colour

Declarative description of a solution:

- We have K colours: green, blue, yellow,…
- If countryA shares a border with countryB, then their colours are different

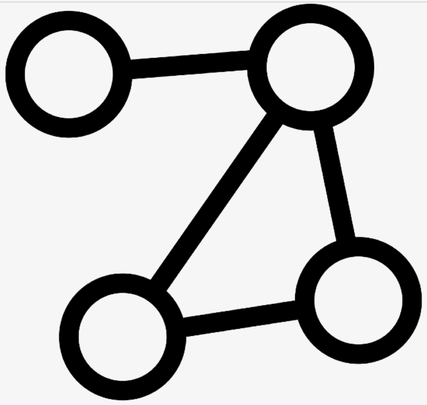
This can then be formally translated into:

- Variables X[i] for i ∈ {countryA, countryB,…}
- dom(X[i]) = {green, blue, yellow,…}

ENUMERATED NON-INTEGER VALUE

X is a variable of enumerated non-integer value:

- If dom(X) = {value1, value2, value3}
- Map it to integers: value1 = 1, value2 = 2,…
- Some arithmetic constraints might not make sense anymore (e.g., X < Y, X+Y = Z)



Map colouring (Graph colouring): Given a map, how can we colour the countries, using K colours, such that not two countries have the same colour

Declarative description of a solution:

- We have K colours: green, blue, yellow,…
- If countryA shares a border with countryB, then their colours are different

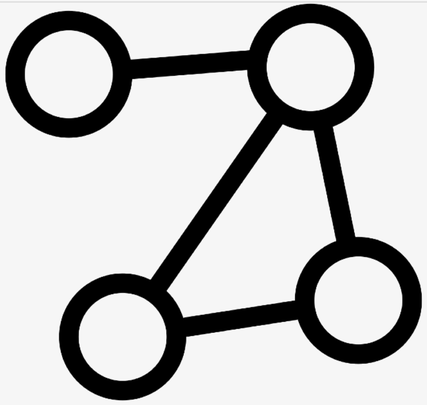
This can then be formally translated into:

- Variables X[i] for i ∈ {countryA, countryB,…}
- dom(X[i]) = {green, blue, yellow,…}
- X[i]!=X[j] if i shares a border with j

ENUMERATED NON-INTEGER VALUE

X is a variable of enumerated non-integer value:

- If dom(X) = {value1, value2, value3}
- Map it to integers: value1 = 1, value2 = 2,…
- Some arithmetic constraints might not make sense anymore (e.g., X < Y, X+Y = Z)



Map colouring (Graph colouring): Given a map, how can we colour the countries, using K colours, such that not two countries have the same colour

Declarative description of a solution:

- We have K colours: green, blue, yellow,…
- If countryA shares a border with countryB, then their colours are different

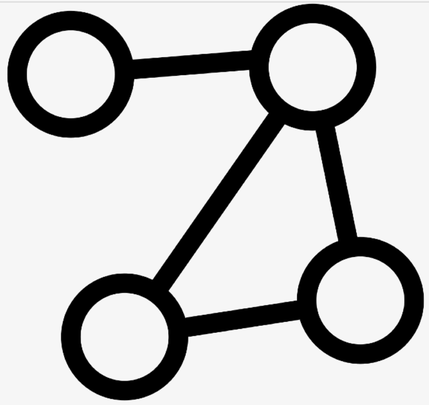
This can then be formally translated into:

- Variables X[i] for i ∈ {countryA, countryB,…}
- dom(X[i]) = {green, blue, yellow,…}
- X[i]!=X[j] if i shares a border with j

ENUMERATED NON-INTEGER VALUE

X is a variable of enumerated non-integer value:

- If dom(X) = {value1, value2, value3}
- Map it to integers: value1 = 1, value2 = 2,…
- Some arithmetic constraints might not make sense anymore (e.g., X < Y, X+Y = Z)



Map colouring (Graph colouring): Given a map, how can we colour the countries, using K colours, such that not two countries have the same colour

Declarative description of a solution:

- We have K colours: green, blue, yellow,…
- If countryA shares a border with countryB, then their colours are different

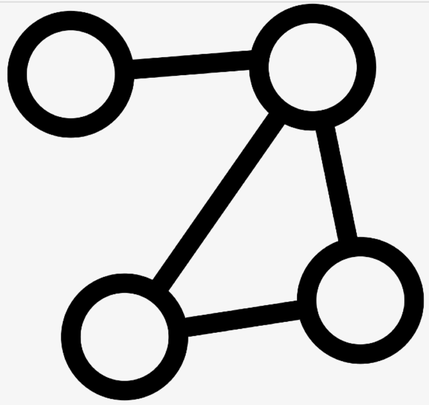
- Mapping to integer domains: green=1, blue=2, yellow=3,…

This can then be formally translated into:

- Variables X[i] for i ∈ {countryA, countryB,…}
- dom(X[i]) = {green, blue, yellow,…}
- X[i]!=X[j] if i shares a border with j

ENUMERATED NON-INTEGER VALUE

X is a variable of enumerated non-integer value:
- If dom(X) = {value1, value2, value3}
- Map it to integers: value1 = 1, value2 = 2,…
- Some arithmetic constraints might not make sense anymore (e.g., X < Y, X+Y = Z)



Map colouring (Graph colouring): Given a map, how can we colour the countries, using K colours, such that not two countries have the same colour

Declarative description of a solution:
- We have K colours: green, blue, yellow,…
- If countryA shares a border with countryB, then their colours are different

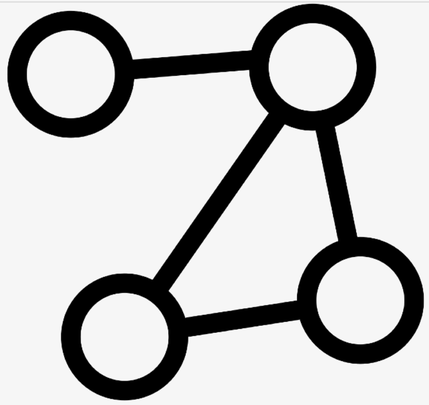
- Mapping to integer domains: green=1, blue=2, yellow=3,…

This can then be formally translated into:
- Variables X[i] for i ∈ {countryA, countryB,…}
- dom(X[i]) = {green, blue, yellow,…}
- X[i]!=X[j] if i shares a border with j

And with the mapping:

ENUMERATED NON-INTEGER VALUE

X is a variable of enumerated non-integer value:
- If dom(X) = {value1, value2, value3}
- Map it to integers: value1 = 1, value2 = 2,…
- Some arithmetic constraints might not make sense anymore (e.g., X < Y, X+Y = Z)



Map colouring (Graph colouring): Given a map, how can we colour the countries, using K colours, such that not two countries have the same colour

Declarative description of a solution:
- We have K colours: green, blue, yellow,…
- If countryA shares a border with countryB, then their colours are different

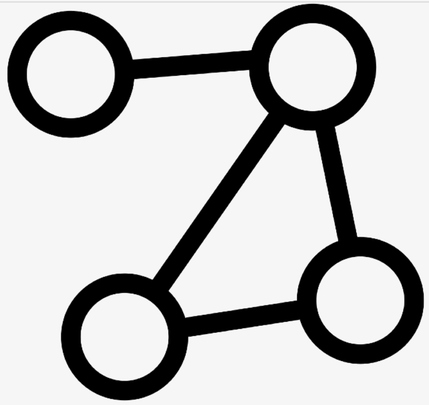
- Mapping to integer domains: green=1, blue=2, yellow=3,…

This can then be formally translated into:
- Variables X[i] for i ∈ {countryA, countryB,…}
- dom(X[i]) = {green, blue, yellow,…}
- X[i]!=X[j] if i shares a border with j

And with the mapping:
- Variables X[i] for i ∈ {countryA, countryB,…}

ENUMERATED NON-INTEGER VALUE

X is a variable of enumerated non-integer value:
- If dom(X) = {value1, value2, value3}
- Map it to integers: value1 = 1, value2 = 2,…
- Some arithmetic constraints might not make sense anymore (e.g., X < Y, X+Y = Z)



Map colouring (Graph colouring): Given a map, how can we colour the countries, using K colours, such that not two countries have the same colour

Declarative description of a solution:
- We have K colours: green, blue, yellow,…
- If countryA shares a border with countryB, then their colours are different

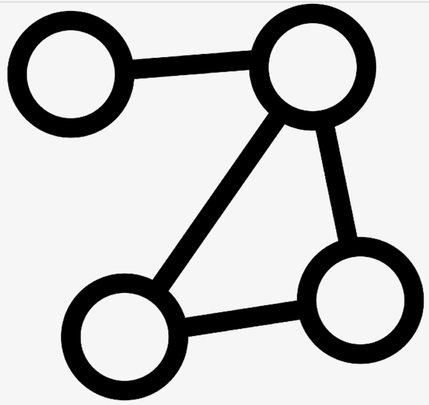
- Mapping to integer domains: green=1, blue=2, yellow=3,…

This can then be formally translated into:
- Variables X[i] for i ∈ {countryA, countryB,…}
- dom(X[i]) = {green, blue, yellow,…}
- X[i]!=X[j] if i shares a border with j

And with the mapping:
- Variables X[i] for i ∈ {countryA, countryB,…}
- dom(X[i]) = {1,2,3,…}

ENUMERATED NON-INTEGER VALUE

X is a variable of enumerated non-integer value:
- If dom(X) = {value1, value2, value3}
- Map it to integers: value1 = 1, value2 = 2,…
- Some arithmetic constraints might not make sense anymore (e.g., X < Y, X+Y = Z)



Map colouring (Graph colouring): Given a map, how can we colour the countries, using K colours, such that not two countries have the same colour

Declarative description of a solution:
- We have K colours: green, blue, yellow,…
- If countryA shares a border with countryB, then their colours are different

- Mapping to integer domains: green=1, blue=2, yellow=3,…

This can then be formally translated into:
- Variables X[i] for i ∈ {countryA, countryB,…}
- dom(X[i]) = {green, blue, yellow,…}
- X[i]!=X[j] if i shares a border with j

And with the mapping:
- Variables X[i] for i ∈ {countryA, countryB,…}
- dom(X[i]) = {1,2,3,…}
- X[i]!=X[j] if i shares a border with j

STRUCTURED VARIABLES

STRUCTURED VARIABLES

Other variables exists, but not supported by a majority of languages and solvers:

STRUCTURED VARIABLES

Other variables exists, but not supported by a majority of languages and solvers:

- Set variables:
  - Domain: a collection of possible sets

- Ref: “Finite Set Constraints in Oz” - T. Müller, M. Müuller - 13th Workshop Logische Programmierung, 1997

STRUCTURED VARIABLES

Other variables exists, but not supported by a majority of languages and solvers:

- Set variables:
  - Domain: a collection of possible sets
    - Ref: “Finite Set Constraints in Oz” - T. Müller, M. Müuller - 13th Workshop Logische Programmierung, 1997
- Interval variables (scheduling variable):
  - Domain: an interval of time where some activity can be scheduled
    - Ref: “Interval constraint programming in C++” - E. Hyvönen, S. De Pascale, A. Lethola - Constraint Programming, 1994

STRUCTURED VARIABLES

Other variables exists, but not supported by a majority of languages and solvers:

- Set variables:
  - Domain: a collection of possible sets
    - Ref: “Finite Set Constraints in Oz” - T. Müller, M. Müuller - 13th Workshop Logische Programmierung, 1997
- Interval variables (scheduling variable):
  - Domain: an interval of time where some activity can be scheduled
    - Ref: “Interval constraint programming in C++” - E. Hyvönen, S. De Pascale, A. Lethola - Constraint Programming, 1994
- Graph variables:
  - Domain: a collection of possible graphs
    - Ref: “Cp (graph): Introducing a graph computation domain in constraint programming” - G. Dooms, Y. Deville, P. Dupont - CP2005

STRUCTURED VARIABLES

Other variables exists, but not supported by a majority of languages and solvers:

- Set variables:
  - Domain: a collection of possible sets
    - Ref: “Finite Set Constraints in Oz” - T. Müller, M. Müuller - 13th Workshop Logische Programmierung, 1997
- Interval variables (scheduling variable):
  - Domain: an interval of time where some activity can be scheduled
    - Ref: “Interval constraint programming in C++” - E. Hyvönen, S. De Pascale, A. Lethola - Constraint Programming, 1994
- Graph variables:
  - Domain: a collection of possible graphs
    - Ref: “Cp (graph): Introducing a graph computation domain in constraint programming” - G. Dooms, Y. Deville, P. Dupont - CP2005
- Sequence variables:
  - Domain: Insertion graph with the possible insertions
    - Ref: “Sequence Variables for Routing Problems” - A. Delecluse, P. Schaus, P. Van Hentenryck - CP2022

PyCSP3



VARIABLES AND DOMAINS IN PYCSP3

https://pycsp.org/documentation/cheatsheet

```
# single variables
v = Var(dom=range(15))         # a variable with domain {0, 1,..., 14}
b = Var(0, 1)                  # a 0/1 variable (can be seen as a Boolean variable)
v = Var(dom=(0, 2, 4, 6, 8))   # a variable with domain {0, 2, 4, 6, 8}
v = Var(dom={"a", "b", "c"})   # a Variable with symbolic domain {a, b, c}

# arrays of variables
x = VarArray(size=10, dom={0, 1})          # a 1-dimensional array of 0/1 variables
m = VarArray(size=[5, 5], dom=range(10))   # a 2-dimensional array of variables with domain {0, 1,..
c = VarArray(size=[4, 3, 4], dom={1, 5, 10, 20})  # a 3-dimensional array of variables

# arrays of variables with different domains
m = VarArray(size=[5, 5], dom=lambda i,j: range(5) if j == 0 else range(10))

print(m[:,j])        # print the jth column of m (indexing as in NumPy)
print(m[0:3,0:3])    # print a block of m (indexing as in NumPy)
```

Constraints

INTENSION

- Arithmetic Intension
  - X+3*Y-Z = 15, X<Y, abs(X-Y)=2,…
- Logical Intension
  - X ı Y, X&Y,…

Use intension when no globals exists for it

INTENSION

- Arithmetic Intension
  - X+3*Y-Z = 15, X<Y, abs(X-Y)=2,…
- Logical Intension
  - X ı Y, X&Y,…

Use intension when no globals exists for it

```
v + w < 10
abs(x[0 - x[1]) >= 7

(v < 2) | (w == 4)     # use of the logical operator 'or' with '|'
either(v < 2, w == 4)  # (equivalent form)

(v < 2) & (w == 4)     # use of the logical operator 'and' with '&'
both(v < 2, w == 4)  # (equivalent form)
```

Sum or weighted sum of variables

$$\sum_i x[i] == Y$$

$$\sum_i x[i] * w[i] == Y$$

ARITHMETIC GLOBALS: SUM

Sum or weighted sum of variables

$$\sum_i x[i] == Y$$

$$\sum_i x[i] * w[i] == Y$$

```
Sum(x) == 12                              # simple sum

Sum(x[i] * c[i] for i in range(5)) > 28   # weighted sum
x * c  > 28                               # equivalent form (dot product)

Sum(x[i] >= 2 for i in range(4))  >= 3    # sum of expressions
```

ARITHMETIC GLOBALS: MAXIMUM/MINIMUM

Extract the minimum or maximum values from variables

```
Maximum(x) > 1          # the maximum value in x must be greater than 1
Minimum(x) == v         # the value of v corresponds to the minimum value in x

Maximum(x[0]+x[1], x[1]+x[2], x[2]+x[3]) >= 5   # when using expressions
```

ARITHMETIC GLOBALS: ALLDIFFERENT

Ensure every variables takes a different value

X!=Y, Y!=Z, X!=Z

```
AllDifferent(x[0], x[3], x[8])  # the three variable must take different values
AllDifferent(x)                 # all variables in x must take different values
AllDifferent(x, excepting=0)    # all variables in x, different from 0, must take different values

AllDifferent(abs(x[1]-x[0]), abs(x[2]-x[1]), abs(x[3]-x[2]))  # with a sequence of expressions
AllDifferent(abs(x[i+1]-x[i]) for i in range(3))              # with a generator of expressions

AllDifferent(m, matrix=True)              # all rows and columns must have different values
AllDifferent(m, matrix=True, excepting=0) #   (while ignoring the value 0)
```

ARITHMETIC GLOBALS: ALLDIFFERENT

Ensure every variables takes a different value
X!=Y, Y!=Z, X!=Z

```
AllDifferent(x[0], x[3], x[8])  # the three variable must take different values
AllDifferent(x)                 # all variables in x must take different values
AllDifferent(x, excepting=0)    # all variables in x, different from 0, must take different values

AllDifferent(abs(x[1]-x[0]), abs(x[2]-x[1]), abs(x[3]-x[2]))  # with a sequence of expressions
AllDifferent(abs(x[i+1]-x[i]) for i in range(3))              # with a generator of expressions

AllDifferent(m, matrix=True)                # all rows and columns must have different values
AllDifferent(m, matrix=True, excepting=0)   #   (while ignoring the value 0)
```

Why would you create a specific constraint for it and not use the decomposition?

ARITHMETIC GLOBALS: ALLDIFFERENT

Ensure every variables takes a different value

X!=Y, Y!=Z, X!=Z

```
AllDifferent(x[0], x[3], x[8])  # the three variable must take different values
AllDifferent(x)                 # all variables in x must take different values
AllDifferent(x, excepting=0)    # all variables in x, different from 0, must take different values

AllDifferent(abs(x[1]-x[0]), abs(x[2]-x[1]), abs(x[3]-x[2]))  # with a sequence of expressions
AllDifferent(abs(x[i+1]-x[i]) for i in range(3))              # with a generator of expressions

AllDifferent(m, matrix=True)                # all rows and columns must have different values
AllDifferent(m, matrix=True, excepting=0)   #   (while ignoring the value 0)
```

Why would you create a specific constraint for it and not use the decomposition?

- More concise models

ARITHMETIC GLOBALS: ALLDIFFERENT

Ensure every variables takes a different value

X!=Y, Y!=Z, X!=Z

```
AllDifferent(x[0], x[3], x[8])  # the three variable must take different values
AllDifferent(x)                 # all variables in x must take different values
AllDifferent(x, excepting=0)    # all variables in x, different from 0, must take different values

AllDifferent(abs(x[1]-x[0]), abs(x[2]-x[1]), abs(x[3]-x[2]))  # with a sequence of expressions
AllDifferent(abs(x[i+1]-x[i]) for i in range(3))              # with a generator of expressions

AllDifferent(m, matrix=True)                # all rows and columns must have different values
AllDifferent(m, matrix=True, excepting=0)   #   (while ignoring the value 0)
```

Why would you create a specific constraint for it and not use the decomposition?

- More concise models
- Allows more high-level reasoning (c.f., the propagators)

ARITHMETIC GLOBALS: ALLEQUAL

Ensure every variables takes the same value

```
AllEqual(x[0], x[3], x[8])                                  # the three variables must take equal valu
AllEqual(abs(x[1]-x[0]), abs(x[2]-x[1]), abs(x[3]-x[2]))  # the three expressions must take equal va
```

COUNTING GLOBALS: COUNT

Count occurence of a given value
Allow to put bounds on the number times this value is used

```
Count(x, value=1) >= 1       # at least 1 variable of x must take the value 1
AtLeastOne(x, value=1)       # (equivalent form)

Count(x, value=0) <= 1       # at most 1 variable of x must take the value 0
AtMostOne(x, value=0)        # (equivalent form)

Count(x, value=1) == 1       # exactly 1 variable of x must take the value 1
ExactlyOne(x, value=1)       # (equivalent form)

Count(x, value=1) == v       # the count must be equal to the value of v
Count(x, values=[1,2]) == 3  # excatly 3 variables must take their values in {1, 2}

Count([x[0]+x[1], x[1]+x[2], x[2]+x[3]], value=3) == 2  # with expressions

Exist(x[i] != y[i] for i in range(n))     # at least one expression must evaluate to 1 (true)
NotExist(x[i] != y[i] for i in range(n))  # none expression must evaluate to 1
AllHold(x[i] != y[i] for i in range(n))   # all expressions must evaluate to 1

Hamming(x, y) > 3                         # the Hamming distance must be greater than 3
```

COUNTING GLOBALS: COUNT

Count occurence of a given value
Allow to put bounds on the number times this value is used

Knapsack variant: with a limit of K objects in the Knapsack

Count(x, value=1) <= K

```
Count(x, value=1) >= 1       # at least 1 variable of x must take the value 1
AtLeastOne(x, value=1)       # (equivalent form)

Count(x, value=0) <= 1       # at most 1 variable of x must take the value 0
AtMostOne(x, value=0)        # (equivalent form)

Count(x, value=1) == 1       # exactly 1 variable of x must take the value 1
ExactlyOne(x, value=1)       # (equivalent form)

Count(x, value=1) == v       # the count must be equal to the value of v
Count(x, values=[1,2]) == 3  # excatly 3 variables must take their values in {1, 2}

Count([x[0]+x[1], x[1]+x[2], x[2]+x[3]], value=3) == 2  # with expressions

Exist(x[i] != y[i] for i in range(n))     # at least one expression must evaluate to 1 (true)
NotExist(x[i] != y[i] for i in range(n))  # none expression must evaluate to 1
AllHold(x[i] != y[i] for i in range(n))   # all expressions must evaluate to 1

Hamming(x, y) > 3                         # the Hamming distance must be greater than 3
```

COUNTING GLOBALS: CARDINALITY

Generalisation of count, or multiple values at the same time

Cardinality(x, occurrences={0:3, 1:range(2,5), 2:3})

Is equivalent to

Count(x,value=0) == 3

Count(x,value=1) >= 2

Count(x,value=1) < 5

Count(x,value=2) == 3

```
Cardinality(x, occurrences={0:3, 1:2, 2:3})              # 0 must occur 3 times, 1 two times, 2 ...
Cardinality(x, occurrences={0:3, 1:range(2,5), 2:3})     # 1 must occur between 2 and 4 times

Cardinality(x, occurrences={i:y[i] for i in range(3)})  # i must occur y[i] times
```

COUNTING GLOBALS: NVALUES

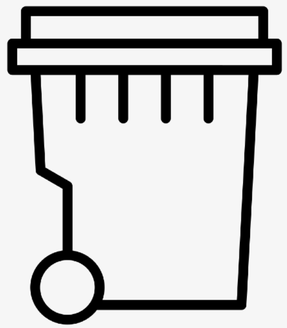
Count the number of distinct values

NValues(x) == ıxı Is equivalent to AllDifferent(x)
NValues(x) == 1 Is equivalent to AllEqual(x)

```
NValues(x) <= 2  # the number of distinct values assigned to variables in x must be less or equal to
NValues(x) == v  # the number of distinct values assigned to variables in x is equal to the value of

NValues(x[0]+x[1], x[1]+x[2], x[2]+x[3], x[3]+x[4]) > 2  # when using expressions
```

PROBLEM-SOLVING GLOBALS: BINPACKING



Binpacking problem (satisfaction problem): Given M bins of capacity C, given N objects of size s[i], find an assignment of the object to the bins, such that the capacities are respected

Corresponds to

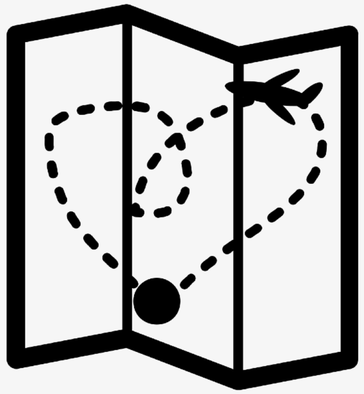
sum({(x[i]==j)*s[I] ı i ∈ {1..N}) <= C for each bin j ∈ {1..M}

```
x = VarArray(size=6, dom={0, 1, 2})     # 6 items to possibly be put in bins 0, 1 and 2

BinPacking(x, sizes=[8,7,3,9,5,7]) <= 12              # the capacity of each bin is 12
BinPacking(x, sizes=[8,7,3,9,5,7], limits=[9,12,11])   # capacity of bins 0, 1 and 2 are 9, 12 and 1

BinPacking(x, sizes=[8,7,3,9,5,7], loads=[y[0],y[2],y[4]])  # load of bins given by variables of y
```

PROBLEM-SOLVING GLOBALS: CIRCUIT



Hamiltonian circuit (satisfaction version of the Travelling Salesman Problem): Given a graph with N vertices, can we select N edges forming a path such that every vertex is visited once

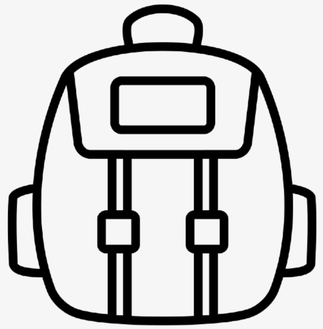
For Circuit, vector x is a successor representation, i.e., vertex x[i] is the successor of i in the circuit

```
x = VarArray(size=6, dom=range(6))

Circuit(x)          # circuit of any size (self-loop are ignored)

Circuit(x, size=3)  # circuit of size 3
Circuit(x, size=v)  # circuit of size given by v
```

PROBLEM-SOLVING GLOBALS: KNAPSACK



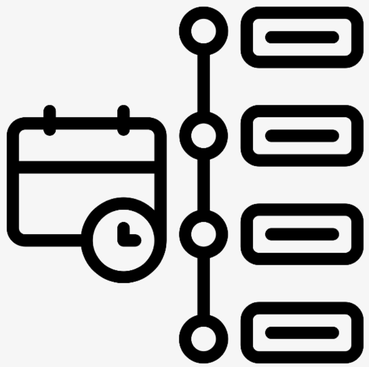
Knapsack problem (satisfaction problem): given a knapsack of capacity C, and the list of objects o1, o2,…, oN. Each object is associated with a weight w[i] and a utility u[i]. Which objects can be selected to not overload the knapsack?

Optimisation version would be:
Knapsack(x,weights=…, wcondition=…, profits=…) == y
maximize(y)

```
x = VarArray(size=5, dom={0, 1})     # 5 possible items to be put in the knapsack

Knapsack(x, weights=[4,8,3,5,7], wlimit=20, profits=[2,3,4,4,3]) >= 12           # with a weight con
Knapsack(x, weights=[4,8,3,5,7], wcondition=lt(20), profits=[2,3,4,4,3])) >= 12  # with a weight con
```

SCHEDULING GLOBALS: NOOVERLAP



Planning of tasks in a calendar: Given N tasks with duration d[i] to plan at a given time, two tasks cannot happen at the same time

Corresponds to:

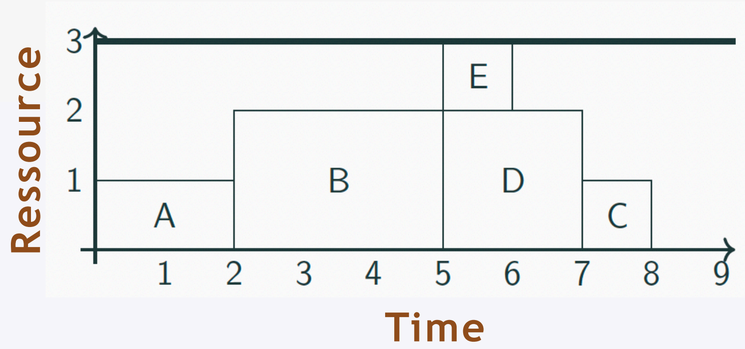
x[i]+d[i] <= x[j] OR x[j]+d[j] <= x[i] for each pair i,j of tasks

```
NoOverlap(origins=x, lengths=[3,5,2,4])                    # one-dimensional form
NoOverlap(origins=x, lengths=y)

NoOverlap(origins=m1, lengths=[[5,4], [2,3], [3,2], [3,4]])  # two-dimensional form
NoOverlap(origins=m1, lengths=m2)
```

SCHEDULING GLOBALS: CUMULATIVE

RCPSP (Resource-Constrained Project Scheduling Problem): Given M ressources of capacity R, N tasks of duration d[i] and usage u[i][r] of ressource r, and given some precedences between pairs of task (i,j), can you plan the schedule such that the ressources are not overused? Can you minimise the total duration of the planning (optimisation version)?

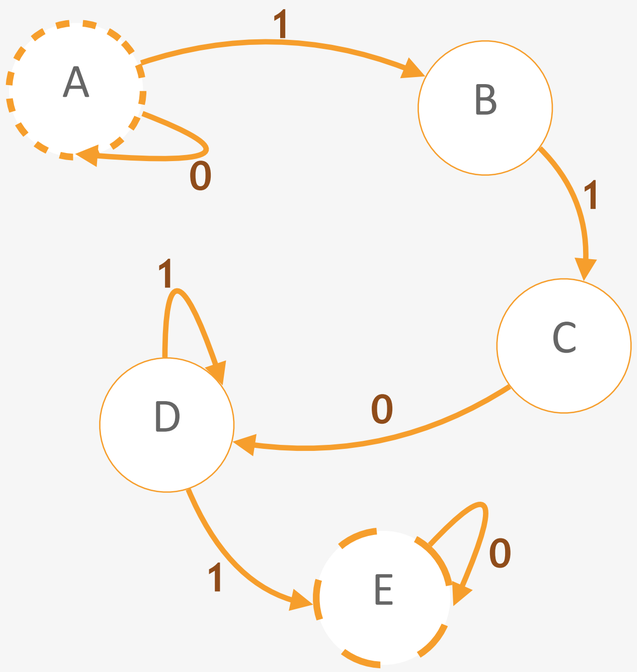



```
x = VarArray(size=5, dom=range(8))
lengths = [3,2,2,4,2]
heights = [3,2,2,2,3]

Cumulative(origins=x, lengths=lengths, heights=heights) <= 5
Cumulative(tasks=[(x[i],lengths[i],heights[i]) for i in range(5)]) <= 5
Cumulative(Task(origin=x[i],length=lengths[i],height=heights[i]) for i in range(5)) <= 5

Cumulative(
  tasks=[Task(origin=x[i],length=lengths[i],height=heights[i]) for i in range(5)]
) <= 5
```

REGULAR



Ensure a sequence of variables is accepted by an automaton (e.g., Regex)

```
a, b, c, d, e = "a", "b", "c", "d", "e"  # names of states
t = [(a,0,a), (a,1,b), (b,1,c), (c,0,d), (d,0,d), (d,1,e), (e,0,e)]
A = Automaton(start=a, transitions=t, final=e)

x in A   # the tuple/word formed by values assigned to variables in x must be accepted by Automaton
```

Given an array of values or integer variables, allows a mapping between a variable and the content of the array at that index

a[Y] == Z

Corresponds to

If Y == 0 then Z == a[0], if Y==1 then Z == a[1],…

Useful to compute mapping and functional values. For example, if Y represents the ID of an object, and a the array containing the respective weights, a[Y] represents the weight of the object which ID is assigned to Y

```
x[v] >= 1      # v is a variable (of the model)
x[v] == w

m[v][w] == 0   # m is a two-dimensional lists
```

CHANNEL

Channel is the bijection version of elements.

```
x = VarArray(size=5, dom=range(5))
y = VarArray(size=5, dom=range(5))

Channel(x)    # imposes x[i] = j  <=> x[j] = i
Channel(x,y)  # imposes x[i] = j <=> y[j] = i

z = VarArray(size=6, dom={0,1})
v = Var(range(6))

Channel(z,v)   # imposes z[i] = 1 <=> v=i
```

ORDERING GLOBALS: LEX

Enforce a lexical ordering on tuples of variables

LexIncreasing([X[0],X[1],X[2]],[Y[0],Y[1],Y[2]])

Corresponds to

(X[0] < Y[0]) ı ((X[0] == Y[0]) & (X[1] < Y[1])) ı ((X[0] == Y[0]) & (X[1] == Y[1]) & (X[2] <= Y[2]))

```
LexIncreasing(x,y)                  # x is less than or equal to y
LexIncreasing(x, y, strict=True)    # x is strictly less than y

LexDecreasing(x, y, z, strict=True) # x is greater than or equal to y, itself greater than or eqaul
LexDecreasing(m, matrix=True)       # all rows are lexicographically ordered as well as all columns
```

EXTENSION (TABLE)

x[0]	x[1]	x[2]	x[3]
1	2	3	2
2	1	1	2
2	3	2	1
3	1	2	3

Extension are the most generic constraints.

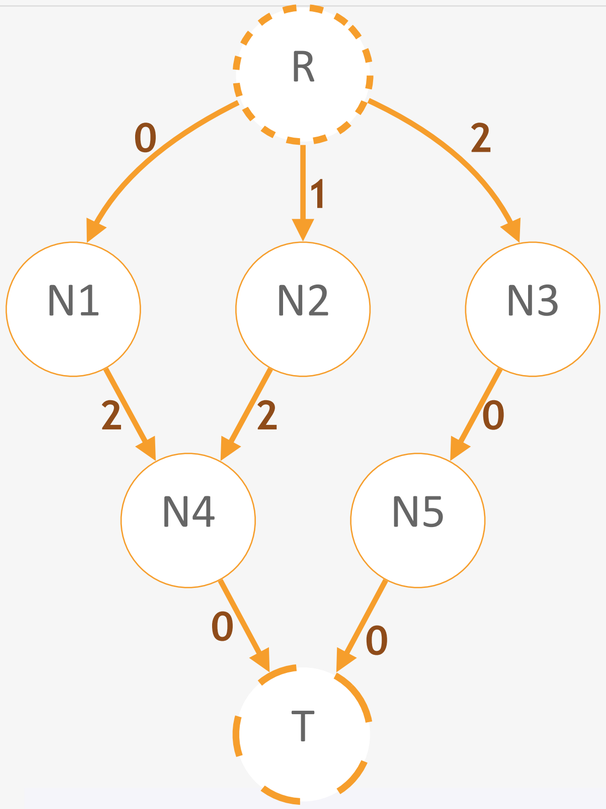
- A table list the solutions allowed or not by the constraint
- For any constraints, we can list their solutions, therefore any constraints can be represented by a table
- They can represent any solution

```
x = VarArray(size=4, dom={1,2,3})

x[0] in {1,3}        # unary positive constraint (allowed values)
x[2] not in {2,3}    # unary negative constraint (forbidden values)

x in {(1,2,3,2), (2,1,1,2), (2,3,2,1), (3,1,2,3)}                  # ordinary positive table
(x[3],x[1],x[0],x[2]) not in {(1,2,3,2), (2,1,1,2), (3,1,2,3)}     # ordinary negative table
(x[0],x[1],x[2]) in {(1,ANY,2), (2,1,ANY), (3,1,3)}                # starred table
```

EXTENSION (MDD)



Multi-valued Decision Diagrams are a compact graph representation of tables

- Each layer corresponds to a variable
- Each edge is associated with a value
- Each path from top to bottom represents a solution
- Fixed length automaton

```
r, n1, n2, n3, n4, n5, t = "r", "n1", "n2", "n3", "n4", "n5", "t"  # names of nodes
transitions = [(r,0,n1), (r,1,n2), (r,2,n3), (n1,2,n4), (n2,2,n4), (n3,0,n5), (n4,0,t), (n5,0,t)]
M = MDD(transitions)

(x[0],x[1],x[2]) in M   # values assigned to variables in x must correspond to a path in the MDD M
```

Objective

OBJECTIVE FOR COP

OBJECTIVE FOR COP

Objective = goal to discriminate among solutions

OBJECTIVE FOR COP

Objective = goal to discriminate among solutions

OBJECTIVE FOR COP

Objective = goal to discriminate among solutions

We are looking for the “best” solution

OBJECTIVE FOR COP

Objective = goal to discriminate among solutions

We are looking for the “best” solution

OBJECTIVE FOR COP

Objective = goal to discriminate among solutions

We are looking for the “best” solution

But what is the “best”?

OBJECTIVE FOR COP

Objective = goal to discriminate among solutions

We are looking for the “best” solution

But what is the “best”?

OBJECTIVE FOR COP

Objective = goal to discriminate among solutions

We are looking for the “best” solution

But what is the “best”?

Minimum or Maximum value of an objective function

OBJECTIVE FOR COP

Objective = goal to discriminate among solutions

We are looking for the “best” solution

But what is the “best”?

Minimum or Maximum value of an objective function
The objective function helps compute a score for each solutions

OBJECTIVE FOR COP

Objective = goal to discriminate among solutions

We are looking for the “best” solution

But what is the “best”?

Minimum or Maximum value of an objective function
The objective function helps compute a score for each solutions

```
minimize(v)                      # the value of the (single) variable v
minimize(v + w * 10)             # the value of an expression
minimize(Sum(x))                 # the sum of x
minimize([v, w] * [3, 2])        # a dot product
minimize(Minimum(x))             # the minimum value in x
minimize(Maximum(x))             # the maximum value in x
minimize(NValues(x))             # the number of different values in x
minimize(Sum(x) - Maximum(y) * 3)  # the value of a complex expression
```

Advanced Modelling Techniques

Channeling and Redundant Constraints



REDUNDANT CONSTRAINTS

Example: given X, Y, Z, you want X to take the smallest value and Z the highest and all to be different.

$$\text{allDifferent}(X,Y,Z)$$

$$X < Y,\ X < Z,\ Y < Z$$

$$X == \text{Minimum}(X,Y,Z),\ Z == \text{Maximum}(X,Y,Z)$$

REDUNDANT CONSTRAINTS

Example: given X, Y, Z, you want X to take the smallest value and Z the highest and all to be different.

allDifferent(X,Y,Z)

X < Y, X < Z, Y < Z

X == Minimum(X,Y,Z), Z == Maximum(X,Y,Z)

REDUNDANT CONSTRAINTS

Example: given X, Y, Z, you want X to take the smallest value and Z the highest and all to be different.

allDifferent(X,Y,Z)

X < Y, X < Z, Y < Z

X == Minimum(X,Y,Z), Z == Maximum(X,Y,Z)

REDUNDANT CONSTRAINTS

Example: given X, Y, Z, you want X to take the smallest value and Z the highest and all to be different.

allDifferent(X,Y,Z)

X < Y, X < Z, Y < Z

X == Minimum(X,Y,Z), Z == Maximum(X,Y,Z)

REDUNDANT CONSTRAINTS

Example: given X, Y, Z, you want X to take the smallest value and Z the highest and all to be different.

allDifferent(X,Y,Z)

X < Y, X < Z, Y < Z

X == Minimum(X,Y,Z), Z == Maximum(X,Y,Z)

Redundant constraints do not help describe the solution, they do not bring more information than what is already in the model

REDUNDANT CONSTRAINTS

Example: given X, Y, Z, you want X to take the smallest value and Z the highest and all to be different.

allDifferent(X,Y,Z)

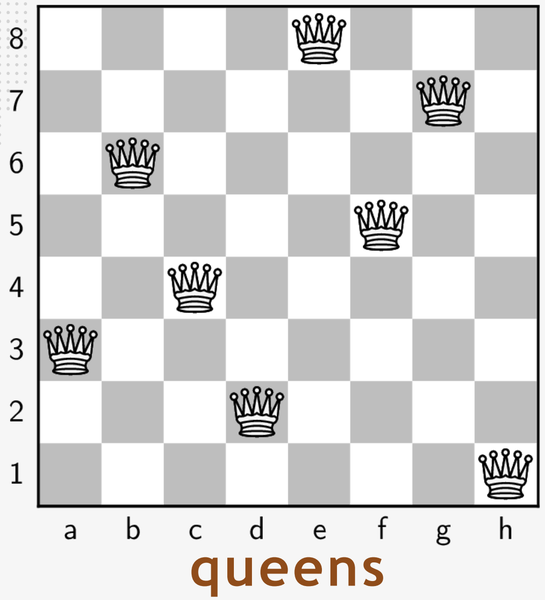
X < Y, X < Z, Y < Z

X == Minimum(X,Y,Z), Z == Maximum(X,Y,Z)

Redundant constraints do not help describe the solution, they do not bring more information than what is already in the model

Why having redundant constraints? Spoiler: the diversity of propagators can help with the solving

N-Queen Problem



queens

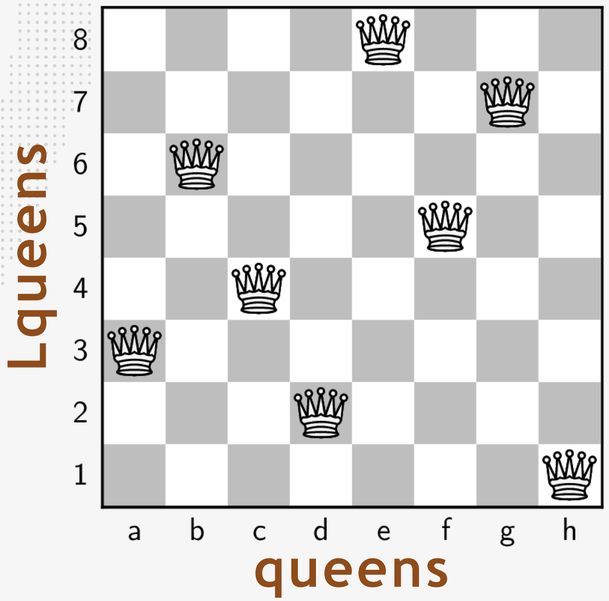
Assigning N Queens on an NxN Board Without Them Attacking Each Other

Model with one queen per column

- Variables queens[i] for i ∈ {1..N}
- queens[i] = {1..N}
- allDifferent(queens)
- allDifferent({queens[i]-i+1 ı i ∈ {1..N}})
- allDifferent({queens[i]+i-1 ı i ∈ {1..N}})

CHANNELING: LINKING ALTERNATIVE MODELS

N-Queen Problem



Assigning N Queens on an NxN Board Without Them Attacking Each Other

Model with one queen per column

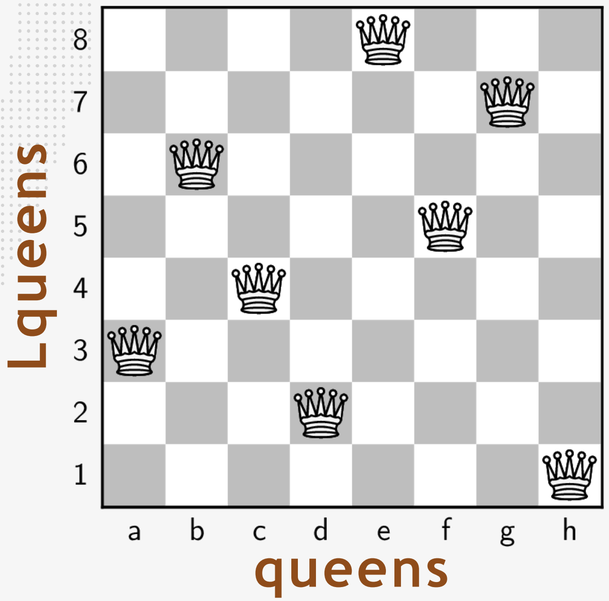
- Variables queens[i] for i ∈ {1..N}
- queens[i] = {1..N}
- allDifferent(queens)
- allDifferent({queens[i]-i+1 ı i ∈ {1..N}})
- allDifferent({queens[i]+i-1 ı i ∈ {1..N}})

Model with one queen per line

- Variables Lqueens[i] for i ∈ {1..N}
- Lqueens[i] = {1..N}
- allDifferent(Lqueens)
- allDifferent({Lqueens[i]-i+1 ı i ∈ {1..N}})
- allDifferent({Lqueens[i]+i-1 ı i ∈ {1..N}})

CHANNELING: LINKING ALTERNATIVE MODELS

N-Queen Problem



Assigning N Queens on an NxN Board Without Them Attacking Each Other

Model with one queen per column

- Variables queens[i] for i ∈ {1..N}
- queens[i] = {1..N}
- allDifferent(queens)
- allDifferent({queens[i]-i+1 ı i ∈ {1..N}})
- allDifferent({queens[i]+i-1 ı i ∈ {1..N}})

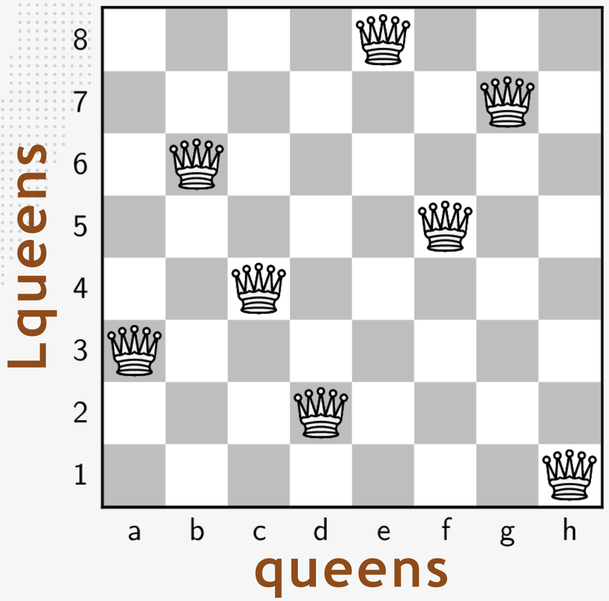
Model with one queen per line

- Variables Lqueens[i] for i ∈ {1..N}
- Lqueens[i] = {1..N}
- allDifferent(Lqueens)
- allDifferent({Lqueens[i]-i+1 ı i ∈ {1..N}})
- allDifferent({Lqueens[i]+i-1 ı i ∈ {1..N}})

Channel(queens,Lqueens)

CHANNELING: LINKING ALTERNATIVE MODELS

N-Queen Problem



Assigning N Queens on an NxN Board Without Them Attacking Each Other

Model with one queen per column

- Variables queens[i] for i ∈ {1..N}
- queens[i] = {1..N}
- allDifferent(queens)
- allDifferent({queens[i]-i+1 ı i ∈ {1..N}})
- allDifferent({queens[i]+i-1 ı i ∈ {1..N}})

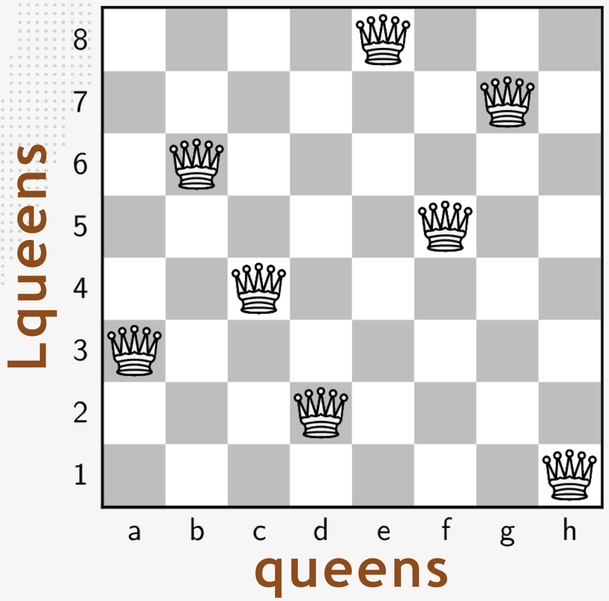
Model with one queen per line

- Variables Lqueens[i] for i ∈ {1..N}
- Lqueens[i] = {1..N}
- allDifferent(Lqueens)
- allDifferent({Lqueens[i]-i+1 ı i ∈ {1..N}})
- allDifferent({Lqueens[i]+i-1 ı i ∈ {1..N}})

Channel(queens,Lqueens)

CHANNELING: LINKING ALTERNATIVE MODELS

N-Queen Problem



Assigning N Queens on an NxN Board Without Them Attacking Each Other

Model with one queen per column

- Variables queens[i] for i ∈ {1..N}
- queens[i] = {1..N}
- allDifferent(queens)
- allDifferent({queens[i]-i+1 ı i ∈ {1..N}})
- allDifferent({queens[i]+i-1 ı i ∈ {1..N}})

Model with one queen per line

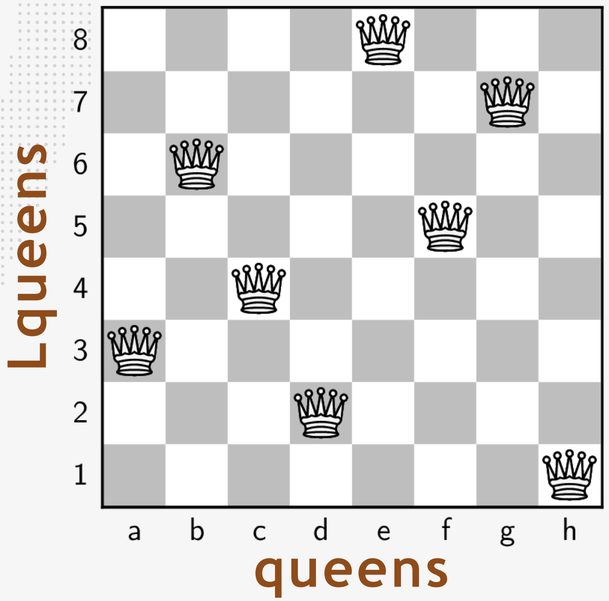
- Variables Lqueens[i] for i ∈ {1..N}
- Lqueens[i] = {1..N}
- allDifferent(Lqueens)
- allDifferent({Lqueens[i]-i+1 ı i ∈ {1..N}})
- allDifferent({Lqueens[i]+i-1 ı i ∈ {1..N}})

Channel(queens,Lqueens)

queens[i] == j <=> Lqueens[j] == I

CHANNELING: LINKING ALTERNATIVE MODELS

N-Queen Problem



Assigning N Queens on an NxN Board Without Them Attacking Each Other

Model with one queen per column

- Variables queens[i] for i ∈ {1..N}
- queens[i] = {1..N}
- allDifferent(queens)
- allDifferent({queens[i]-i+1 ı i ∈ {1..N}})
- allDifferent({queens[i]+i-1 ı i ∈ {1..N}})

Model with one queen per line

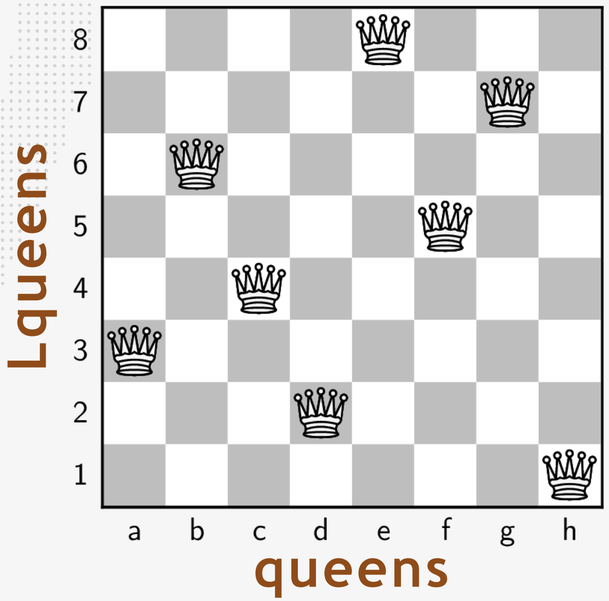
- Variables Lqueens[i] for i ∈ {1..N}
- Lqueens[i] = {1..N}
- allDifferent(Lqueens)
- allDifferent({Lqueens[i]-i+1 ı i ∈ {1..N}})
- allDifferent({Lqueens[i]+i-1 ı i ∈ {1..N}})

Channel(queens,Lqueens)

queens[i] == j <=> Lqueens[j] == I

CHANNELING: LINKING ALTERNATIVE MODELS

N-Queen Problem



Assigning N Queens on an NxN Board Without Them Attacking Each Other

Model with one queen per column

- Variables queens[i] for i ∈ {1..N}
- queens[i] = {1..N}
- allDifferent(queens)
- allDifferent({queens[i]-i+1 ı i ∈ {1..N}})
- allDifferent({queens[i]+i-1 ı i ∈ {1..N}})

Model with one queen per line

- Variables Lqueens[i] for i ∈ {1..N}
- Lqueens[i] = {1..N}
- allDifferent(Lqueens)
- allDifferent({Lqueens[i]-i+1 ı i ∈ {1..N}})
- allDifferent({Lqueens[i]+i-1 ı i ∈ {1..N}})

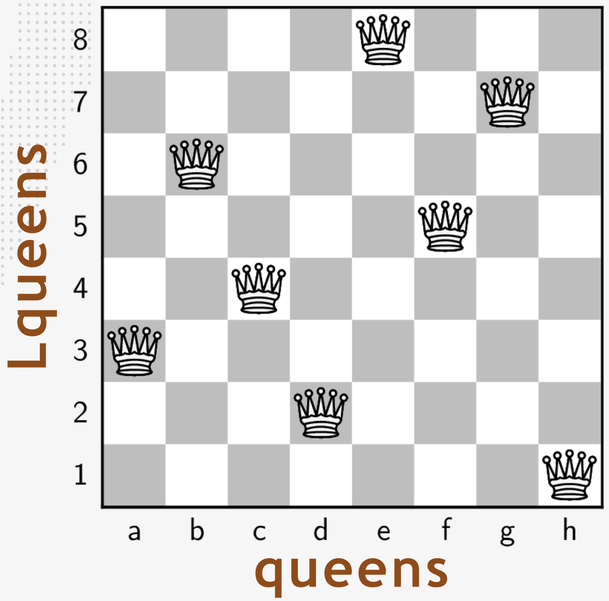
Channel(queens,Lqueens)

queens[i] == j <=> Lqueens[j] == I

queens[a] == 3 <=> Lqueens[3] == a

CHANNELING: LINKING ALTERNATIVE MODELS

N-Queen Problem



Assigning N Queens on an NxN Board Without Them Attacking Each Other

Model with one queen per column

- Variables queens[i] for i ∈ {1..N}
- queens[i] = {1..N}
- allDifferent(queens)
- allDifferent({queens[i]-i+1 ı i ∈ {1..N}})
- allDifferent({queens[i]+i-1 ı i ∈ {1..N}})

Model with one queen per line

- Variables Lqueens[i] for i ∈ {1..N}
- Lqueens[i] = {1..N}
- allDifferent(Lqueens)
- allDifferent({Lqueens[i]-i+1 ı i ∈ {1..N}})
- allDifferent({Lqueens[i]+i-1 ı i ∈ {1..N}})

Channel(queens,Lqueens)

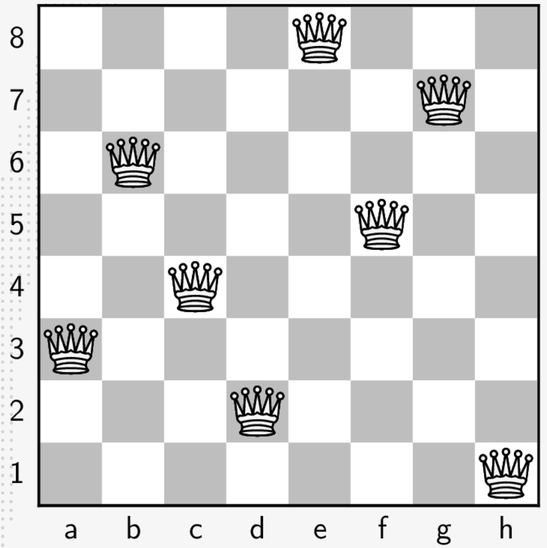
queens[i] == j <=> Lqueens[j] == I

queens[a] == 3 <=> Lqueens[3] == a
queens[b] == 6 <=> Lqueens[6] == b

Breaking Symmetries

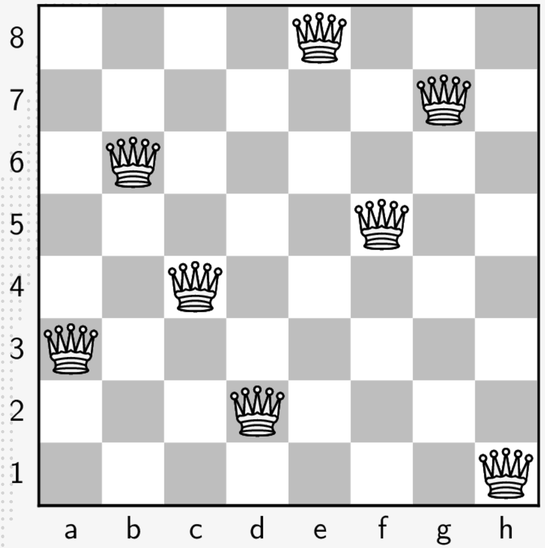
N-Queen Problem

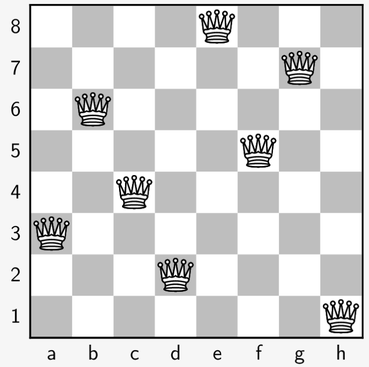
Assigning N Queens on an NxN Board Without Them Attacking Each Other



N-Queen Problem

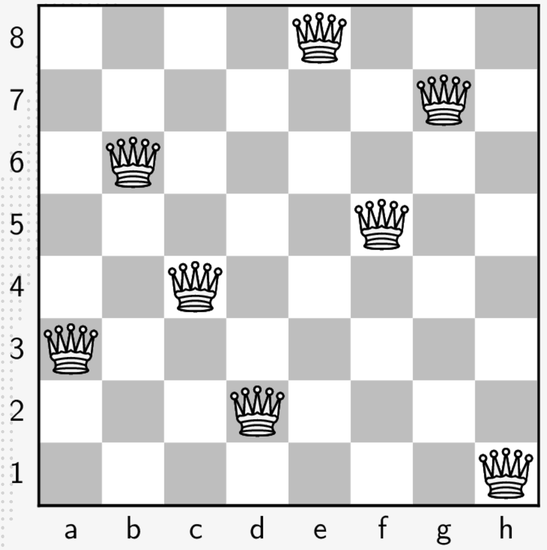
Assigning N Queens on an NxN Board Without Them Attacking Each Other

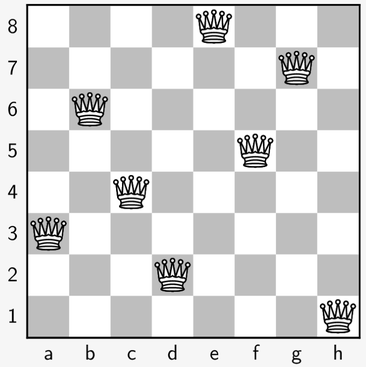


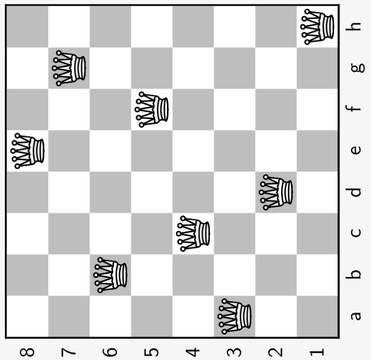


N-Queen Problem

Assigning N Queens on an NxN Board Without Them Attacking Each Other

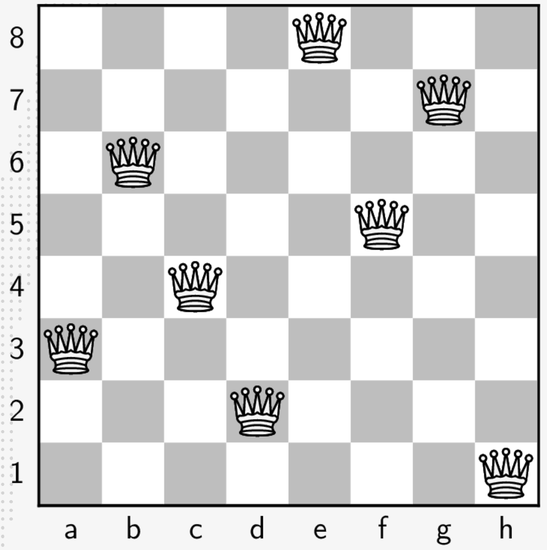


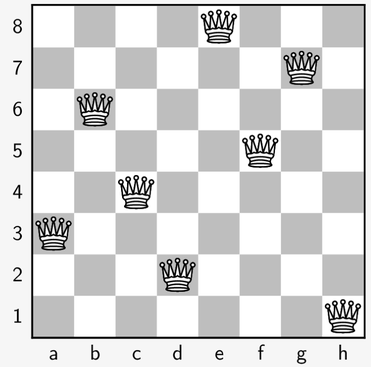


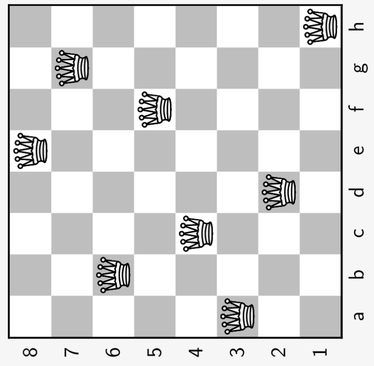


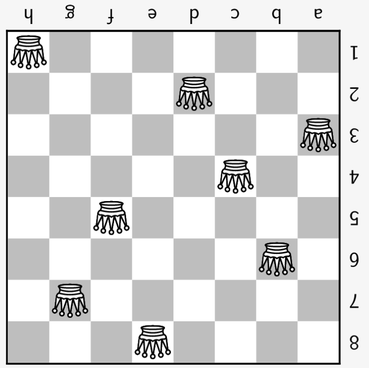
N-Queen Problem

Assigning N Queens on an NxN Board Without Them Attacking Each Other



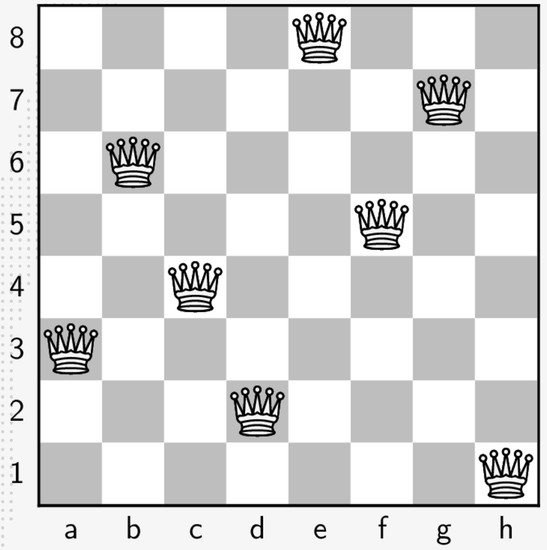


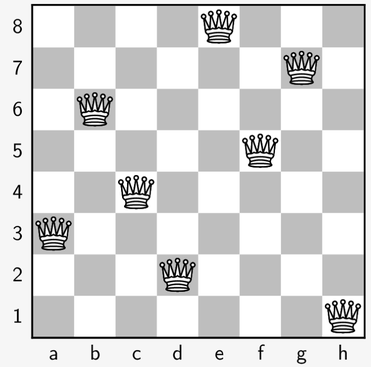


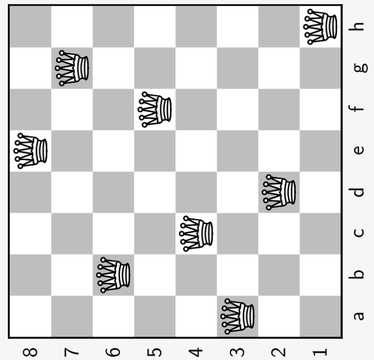


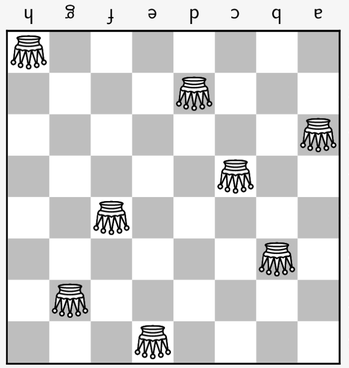
N-Queen Problem

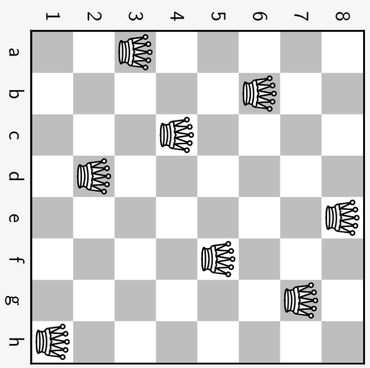
Assigning N Queens on an NxN Board Without Them Attacking Each Other





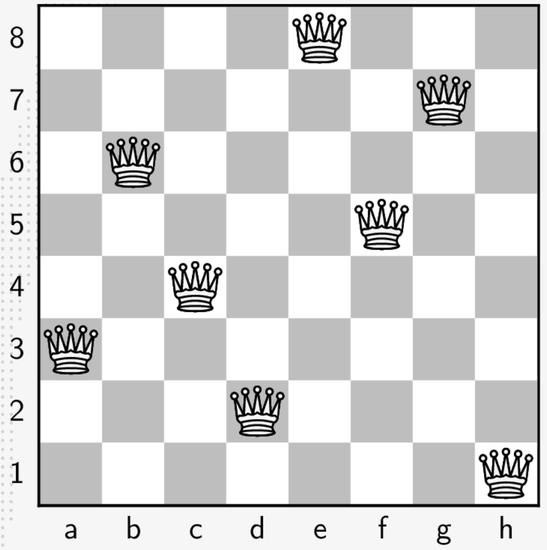


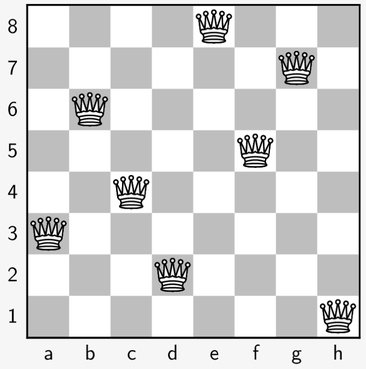


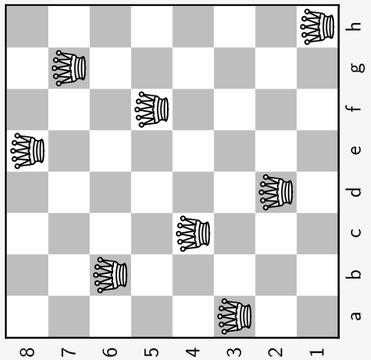


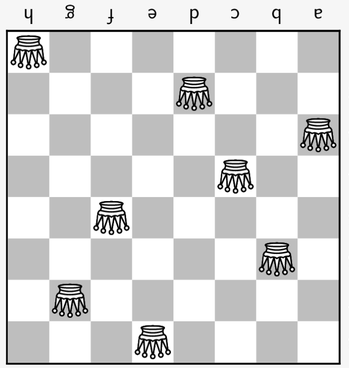
N-Queen Problem

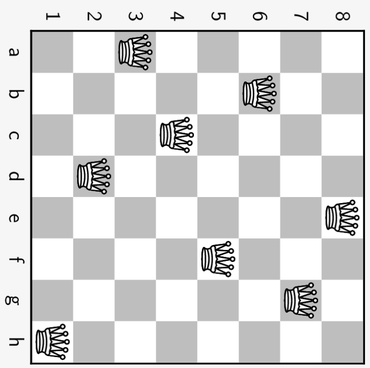
Assigning N Queens on an NxN Board Without Them Attacking Each Other

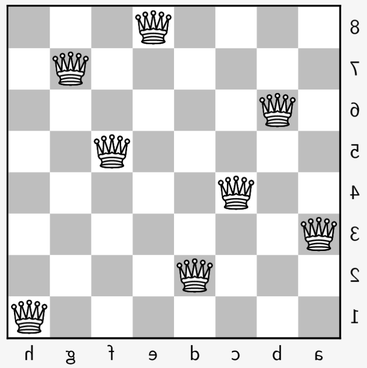






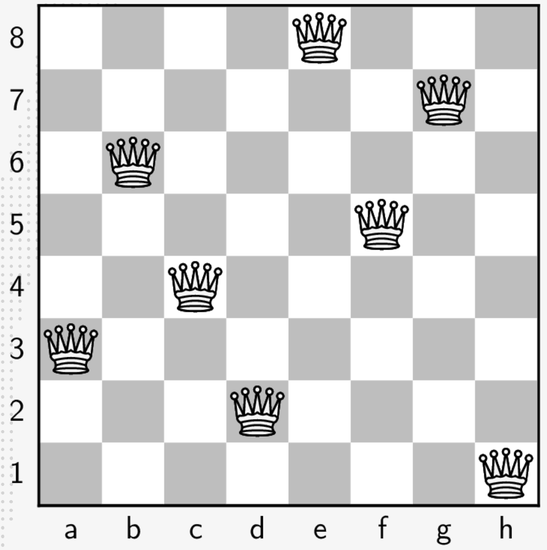


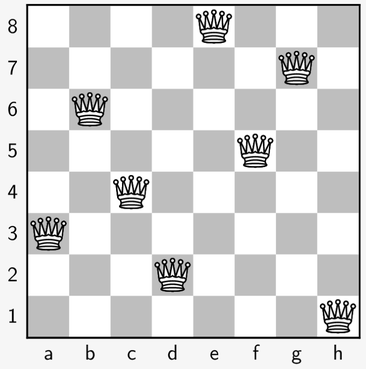


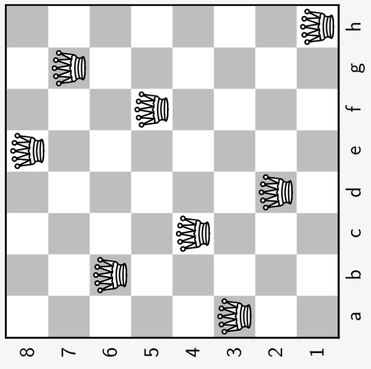


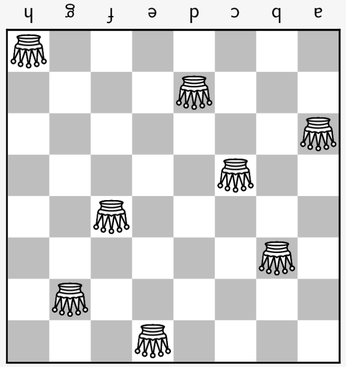
N-Queen Problem

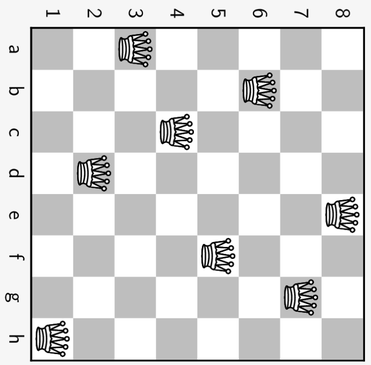
Assigning N Queens on an NxN Board Without Them Attacking Each Other

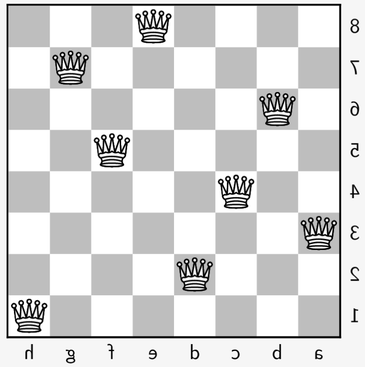


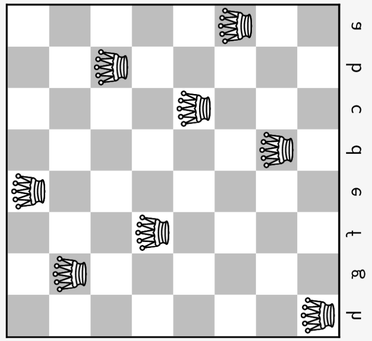






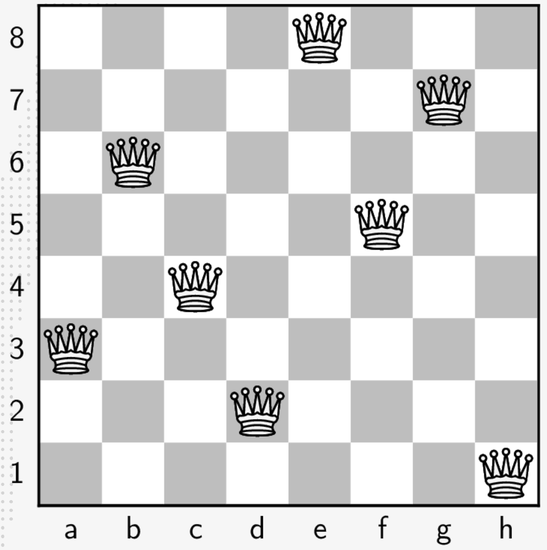


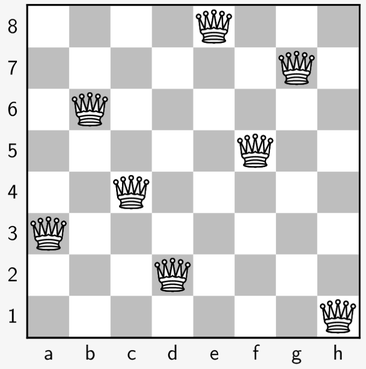


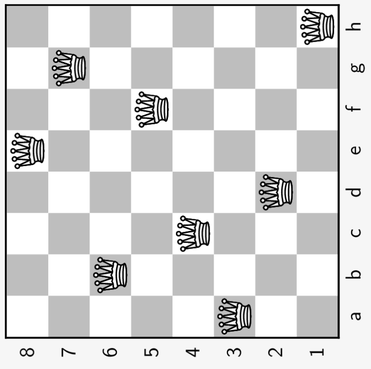


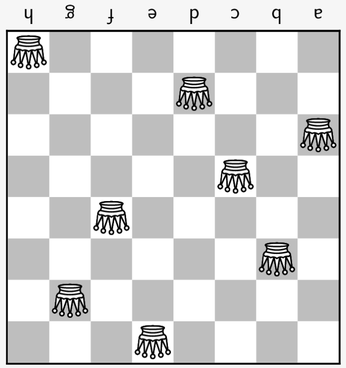
N-Queen Problem

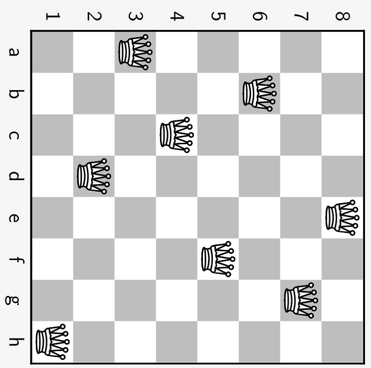
Assigning N Queens on an NxN Board Without Them Attacking Each Other

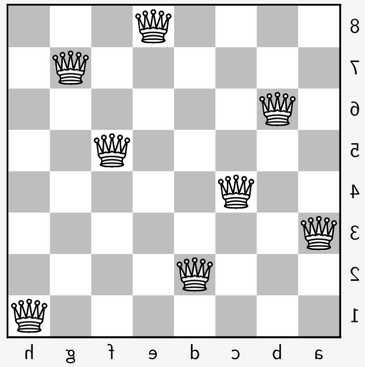


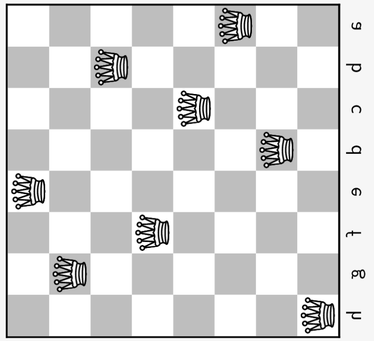


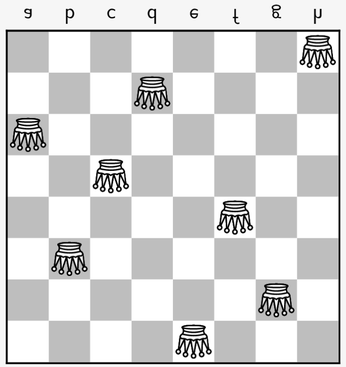






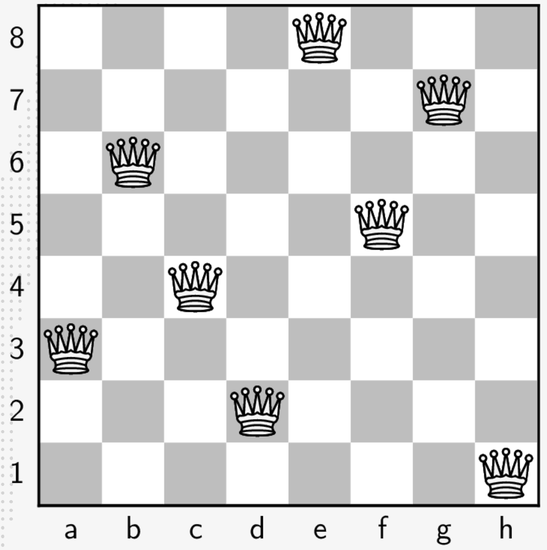


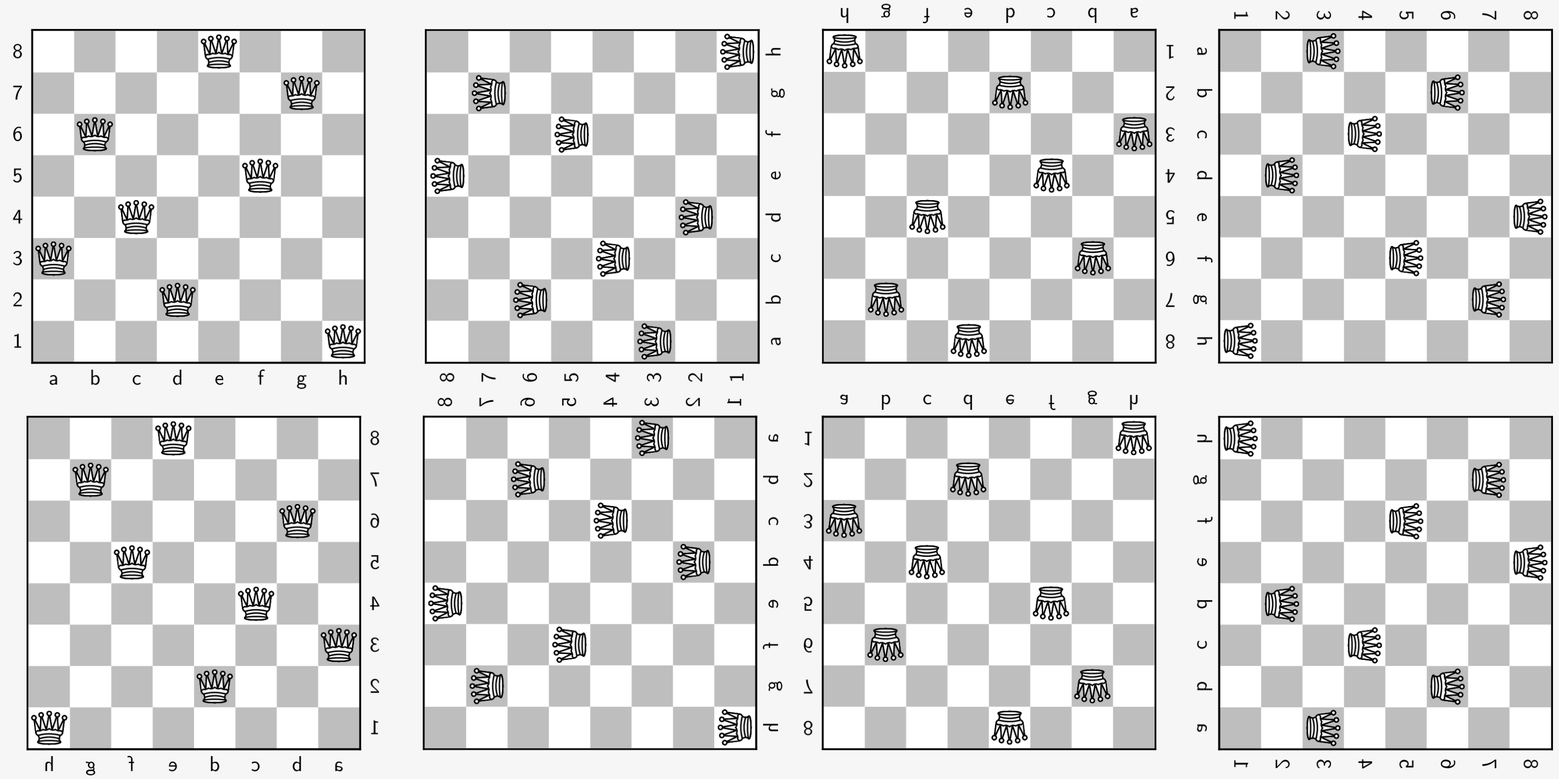




N-Queen Problem

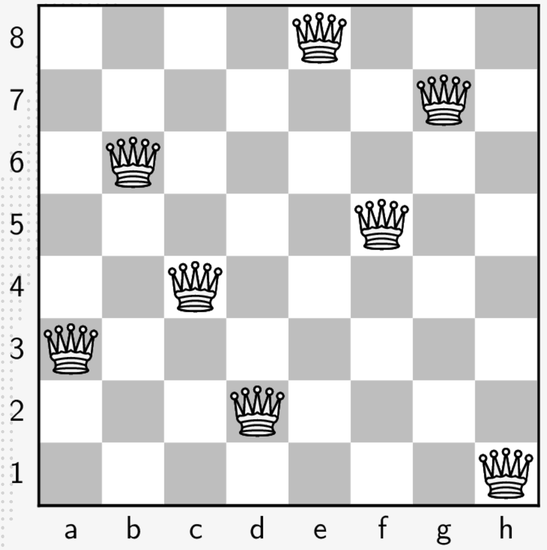
Assigning N Queens on an NxN Board Without Them Attacking Each Other

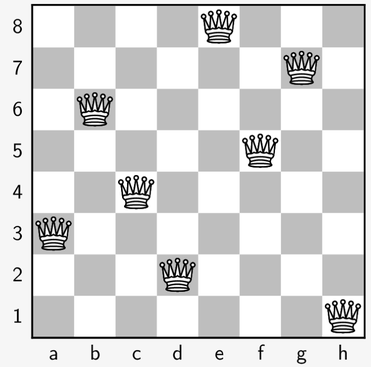


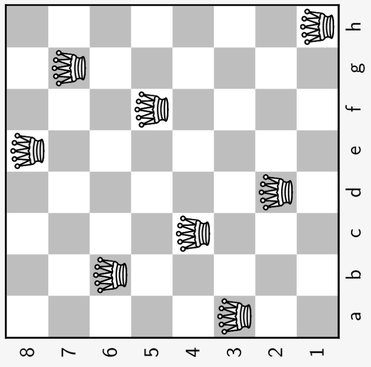


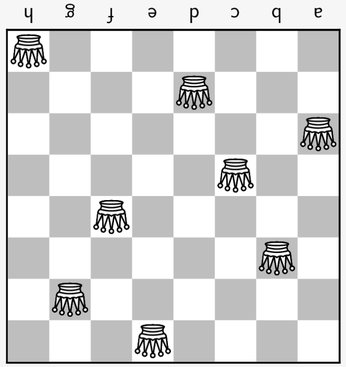
N-Queen Problem

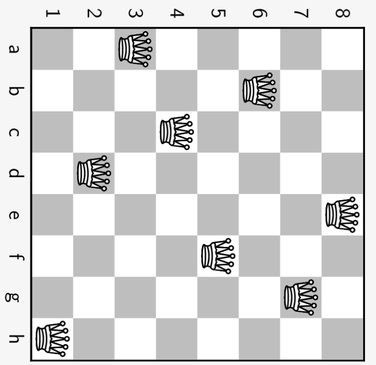
Assigning N Queens on an NxN Board Without Them Attacking Each Other

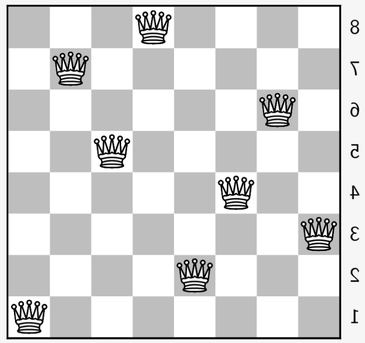


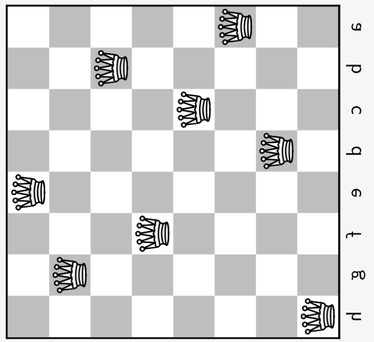


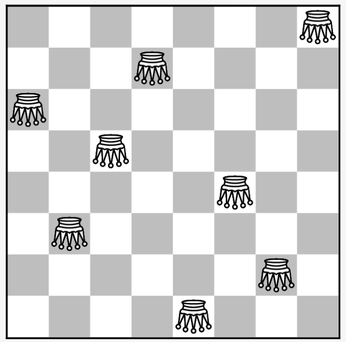


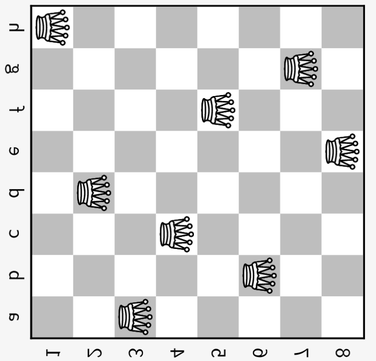


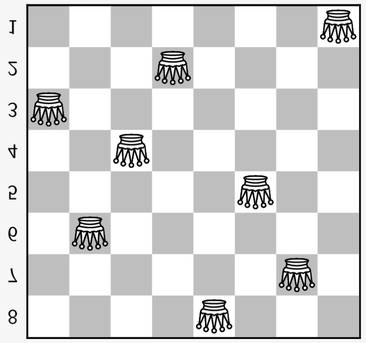






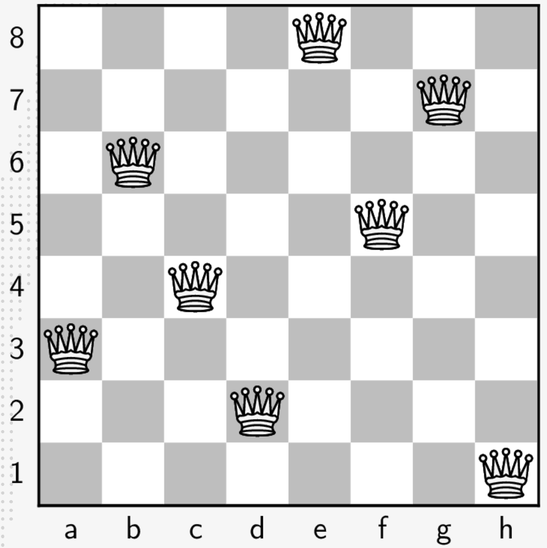


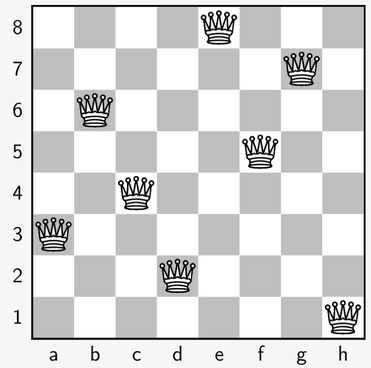


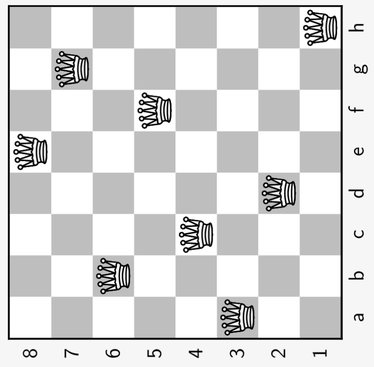


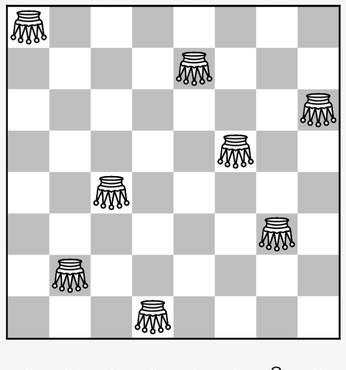
N-Queen Problem

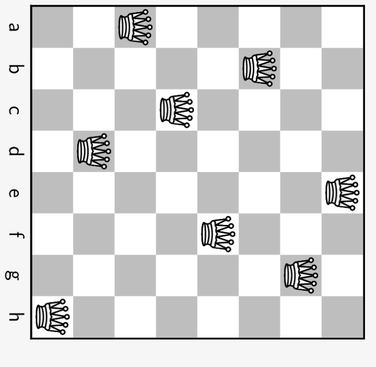
Assigning N Queens on an NxN Board Without Them Attacking Each Other

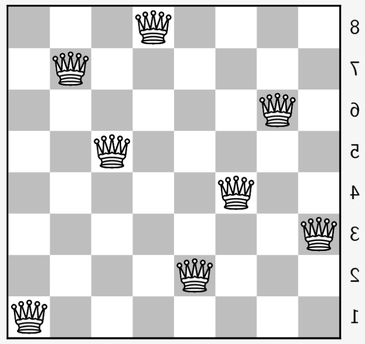


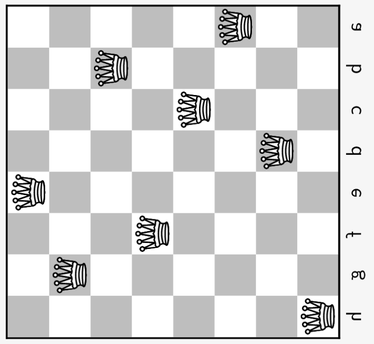


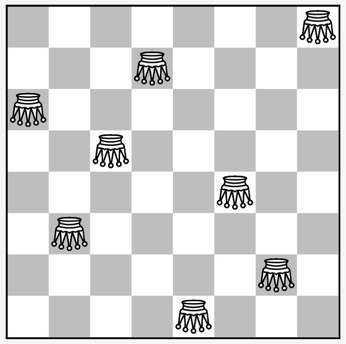


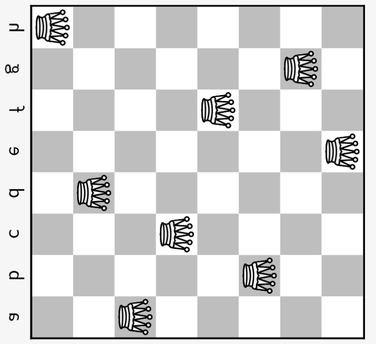


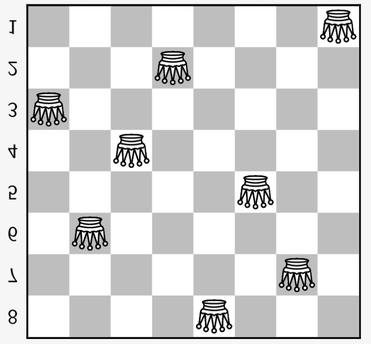


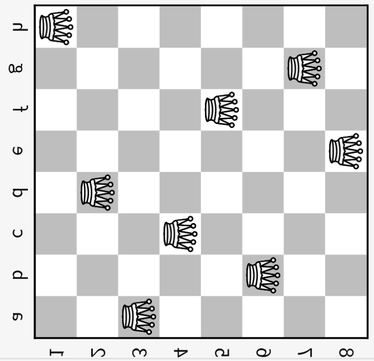






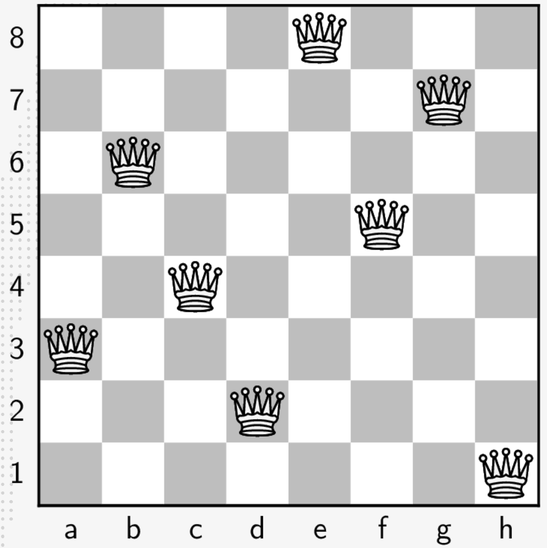


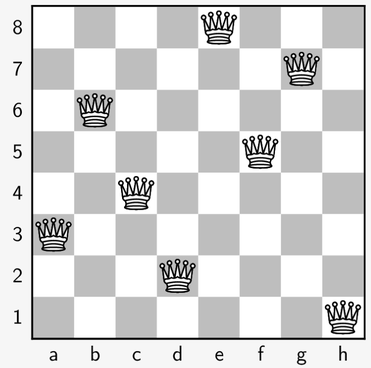


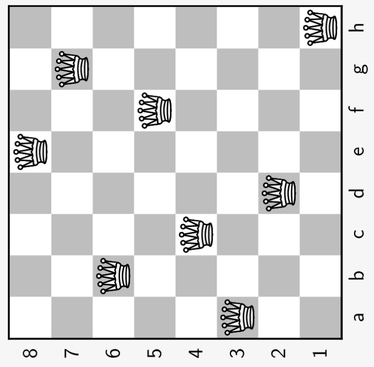


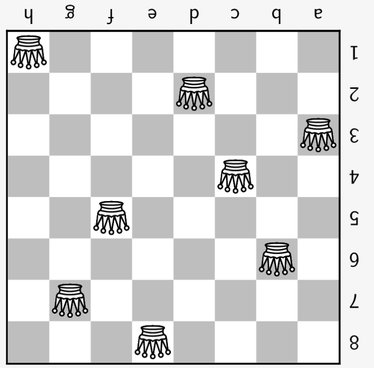
N-Queen Problem

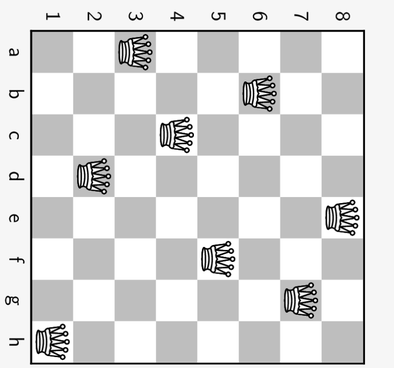
Assigning N Queens on an NxN Board Without Them Attacking Each Other

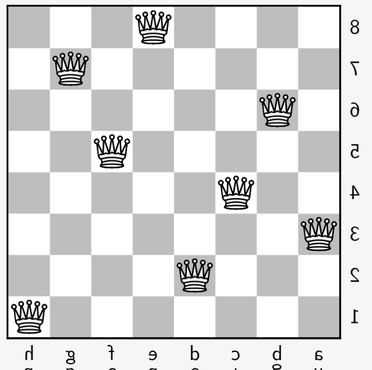


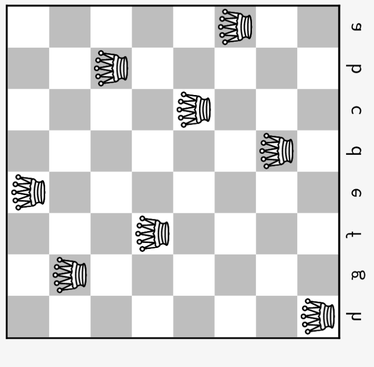


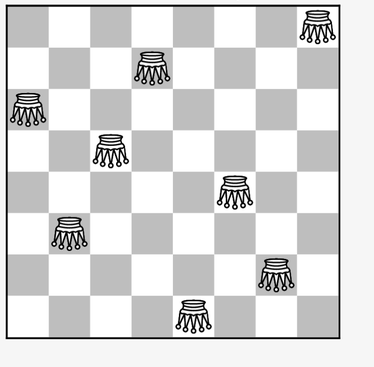


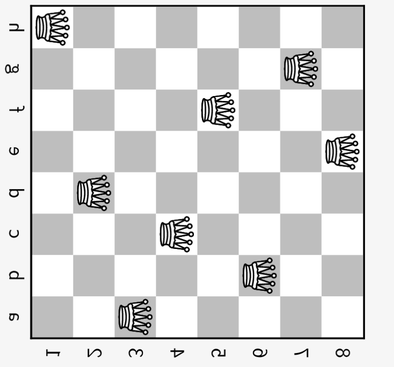


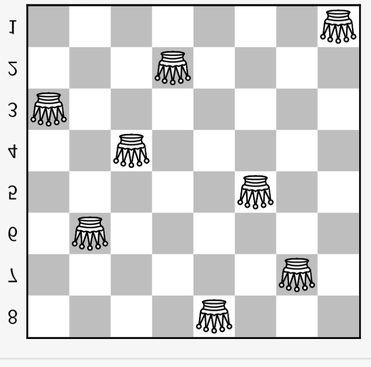


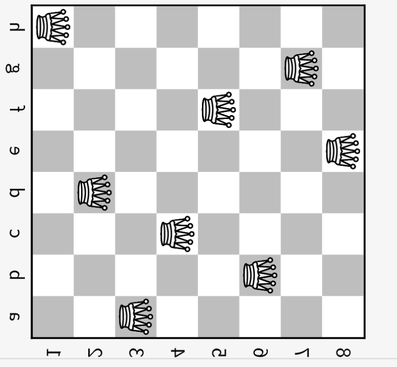


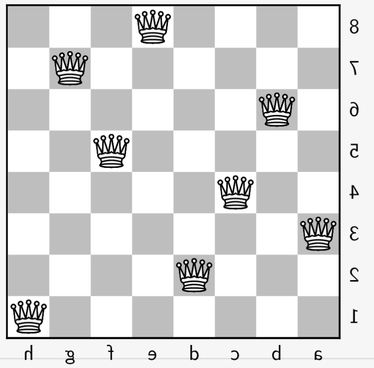






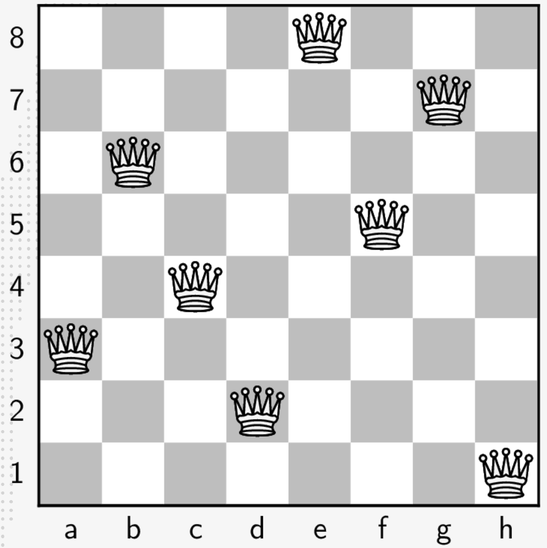


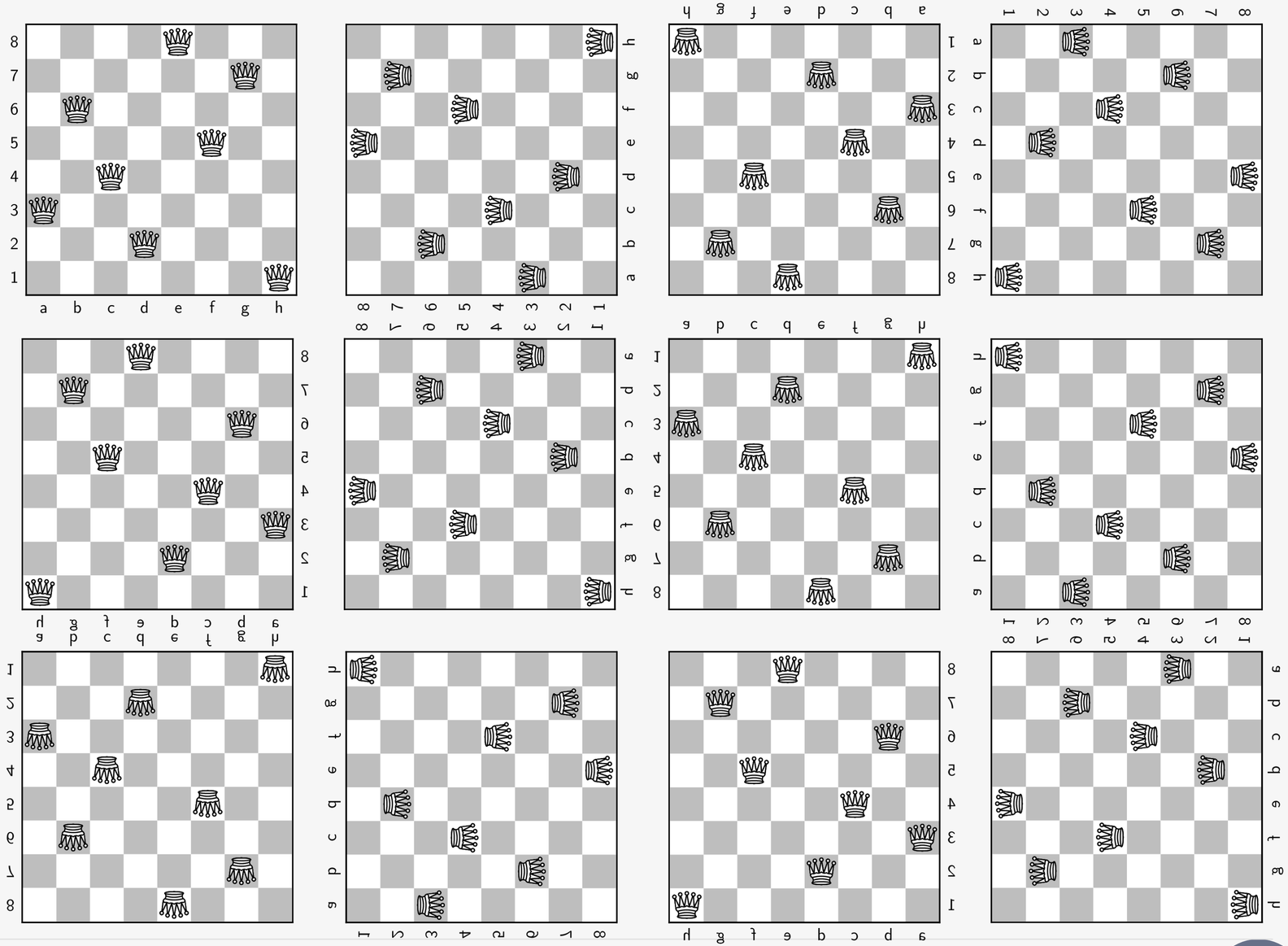




N-Queen Problem

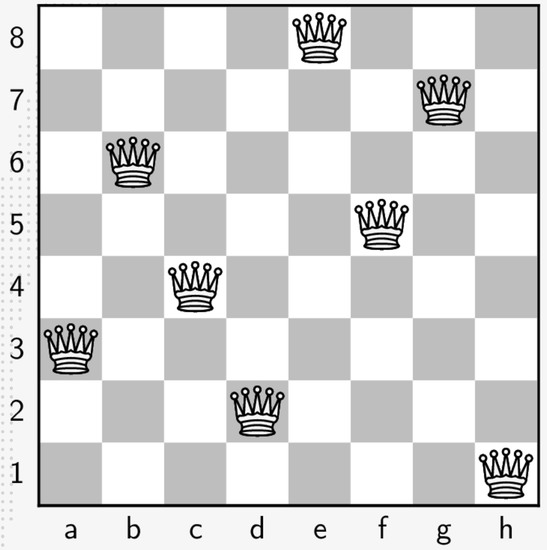
Assigning N Queens on an NxN Board Without Them Attacking Each Other

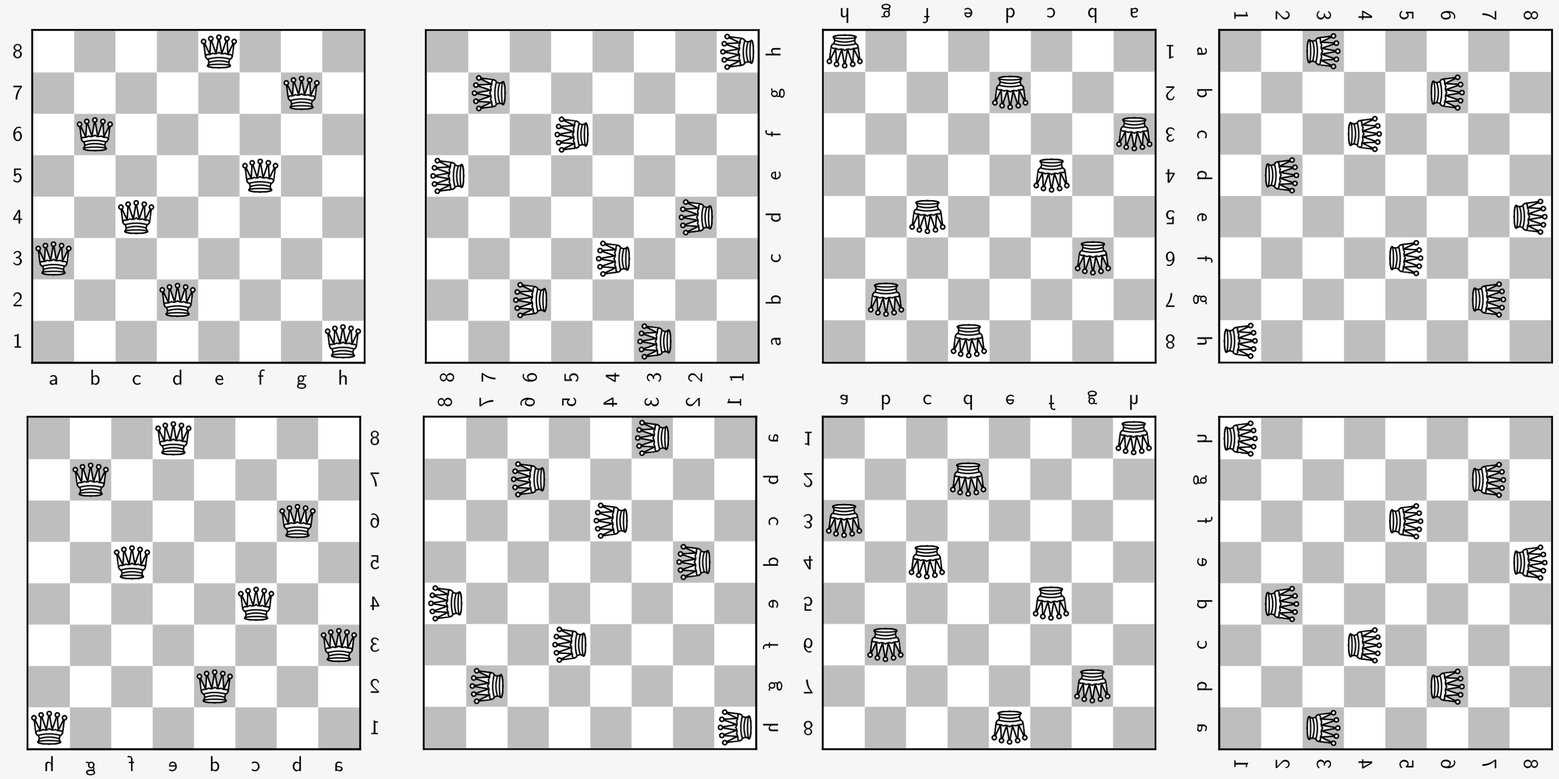




N-Queen Problem

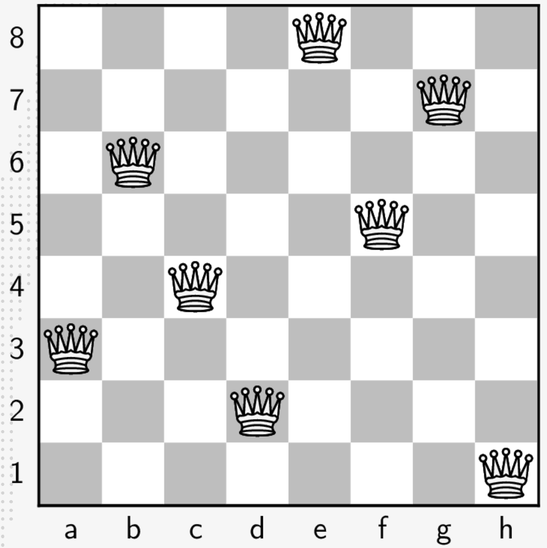
Assigning N Queens on an NxN Board Without Them Attacking Each Other



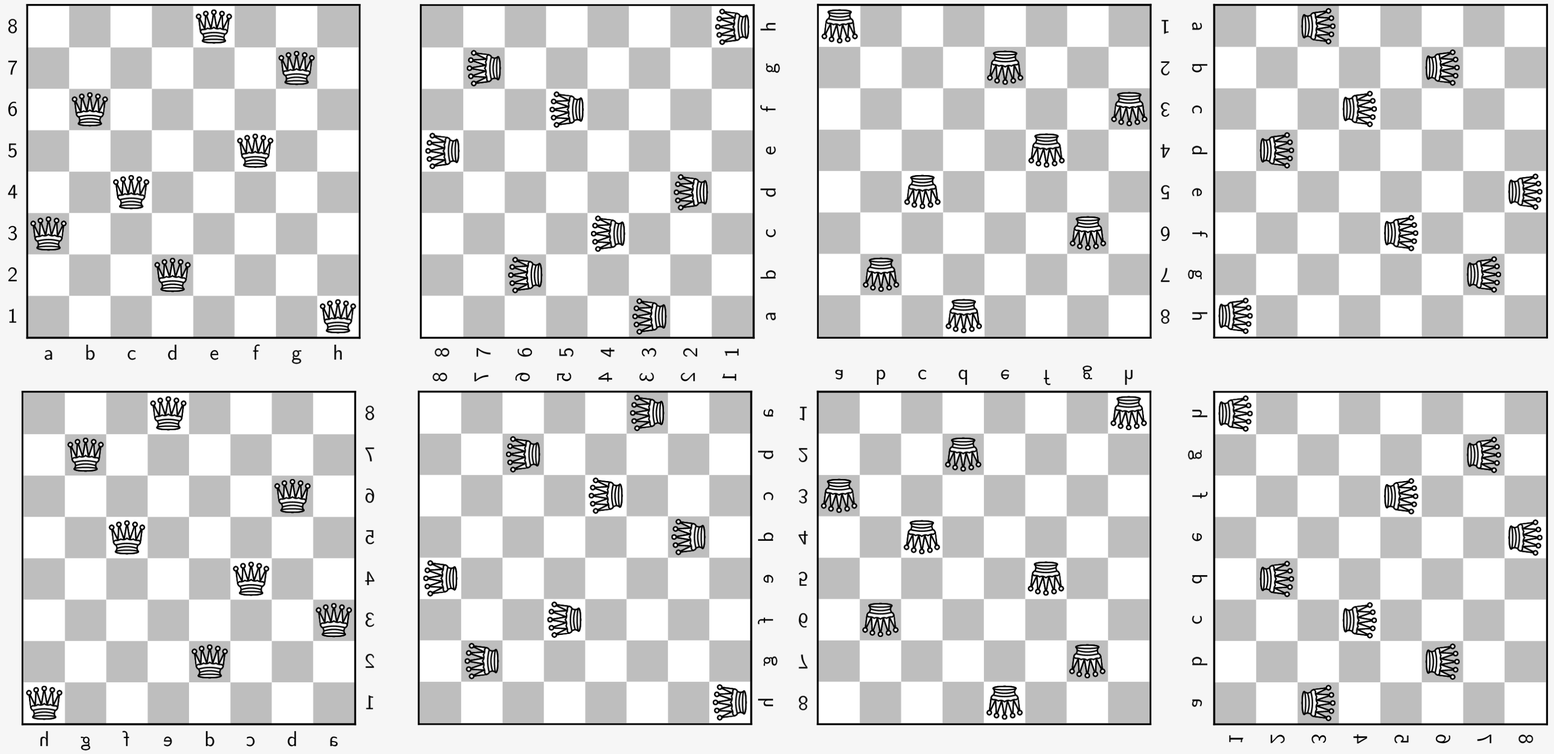


N-Queen Problem

Assigning N Queens on an NxN Board Without Them Attacking Each Other

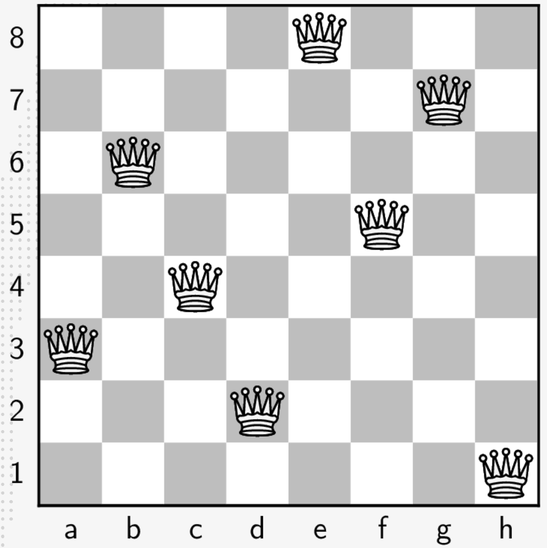


q[a]<= N/2

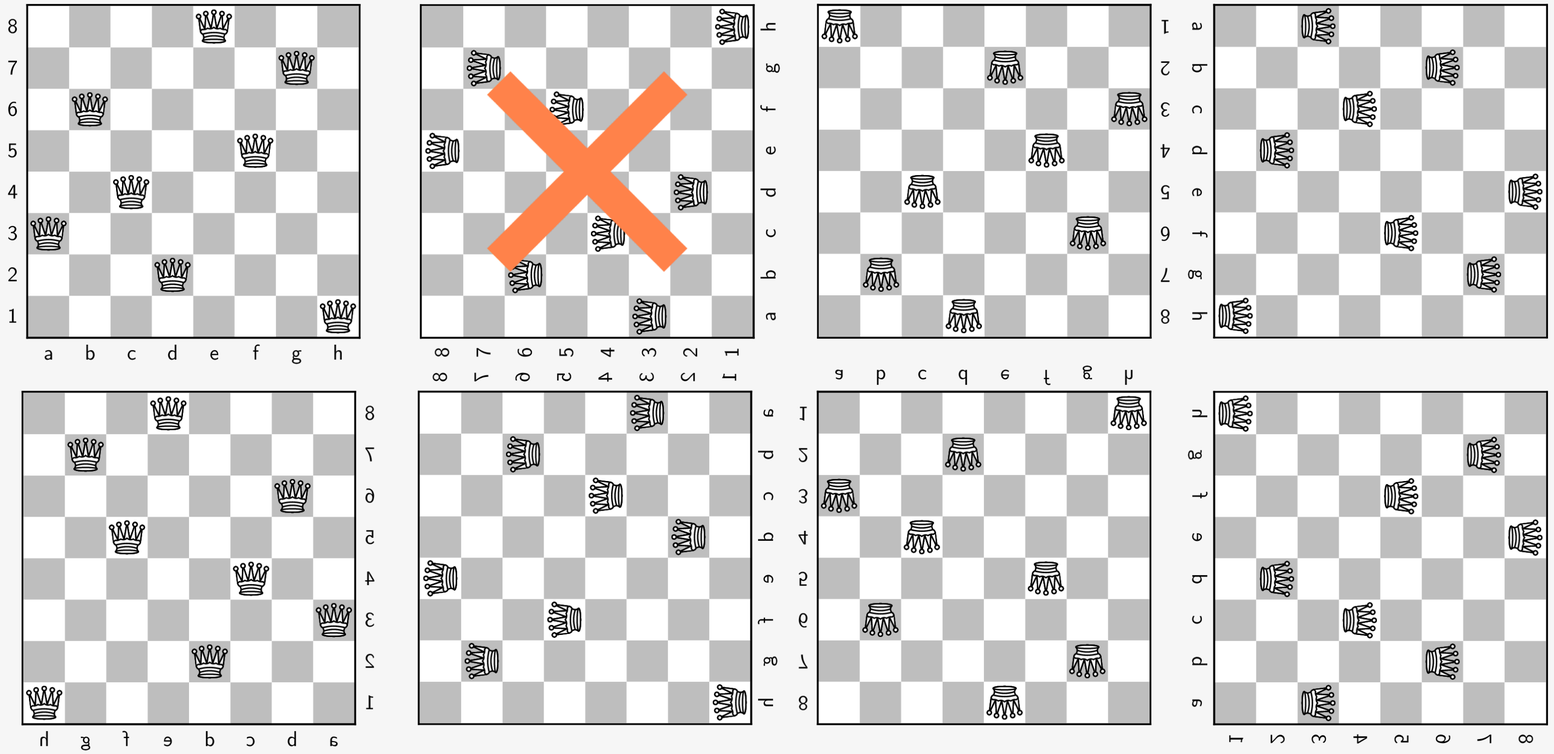


N-Queen Problem

Assigning N Queens on an NxN Board Without Them Attacking Each Other

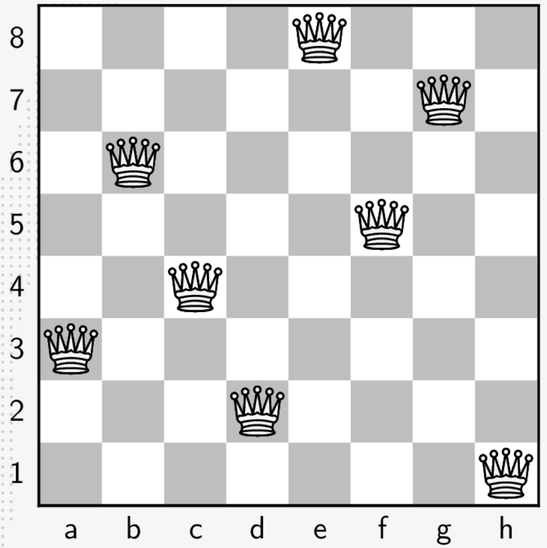


q[a]<= N/2

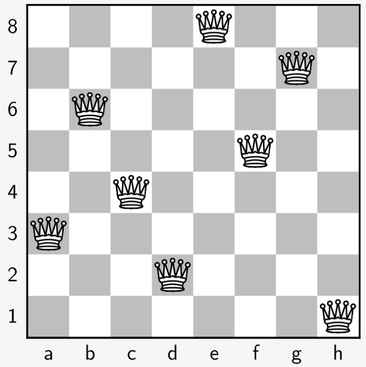


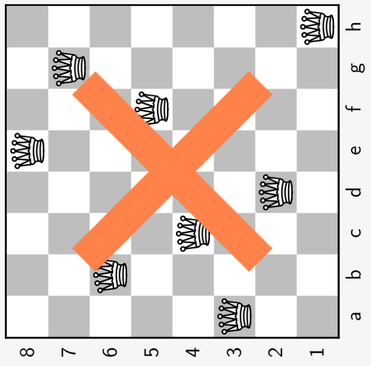
N-Queen Problem

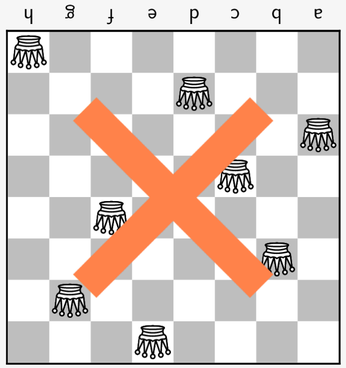
Assigning N Queens on an NxN Board Without Them Attacking Each Other

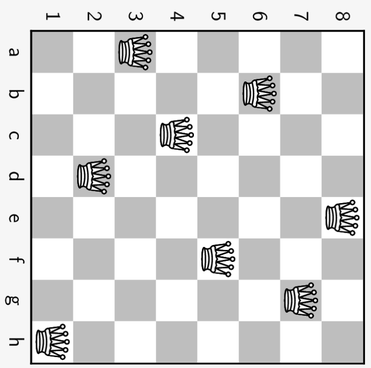


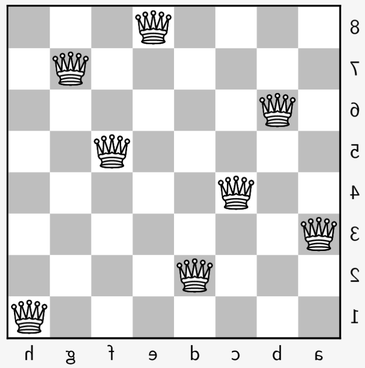
q[a]<= N/2

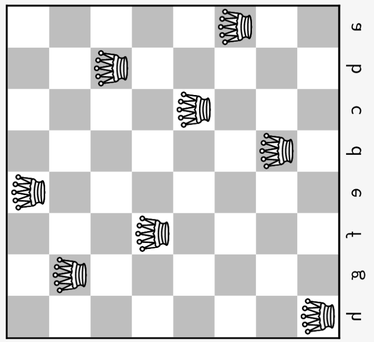


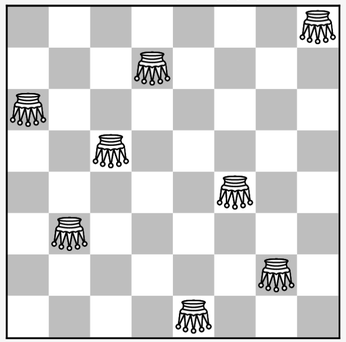


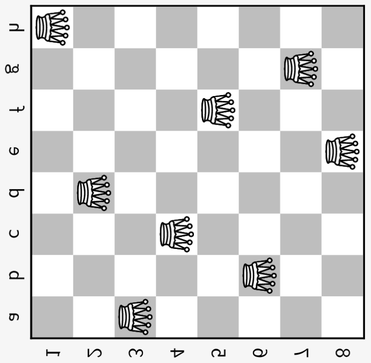






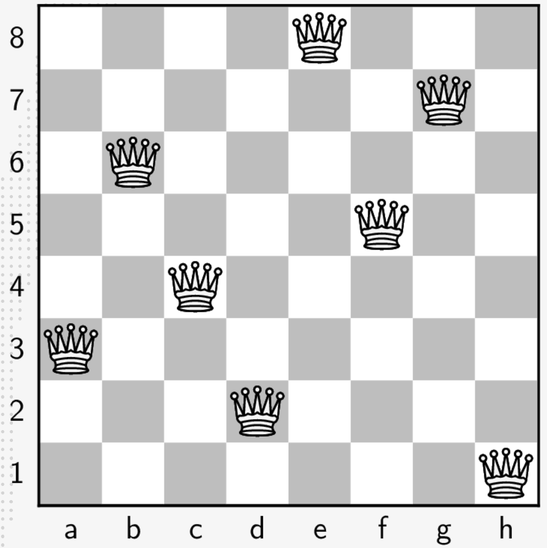




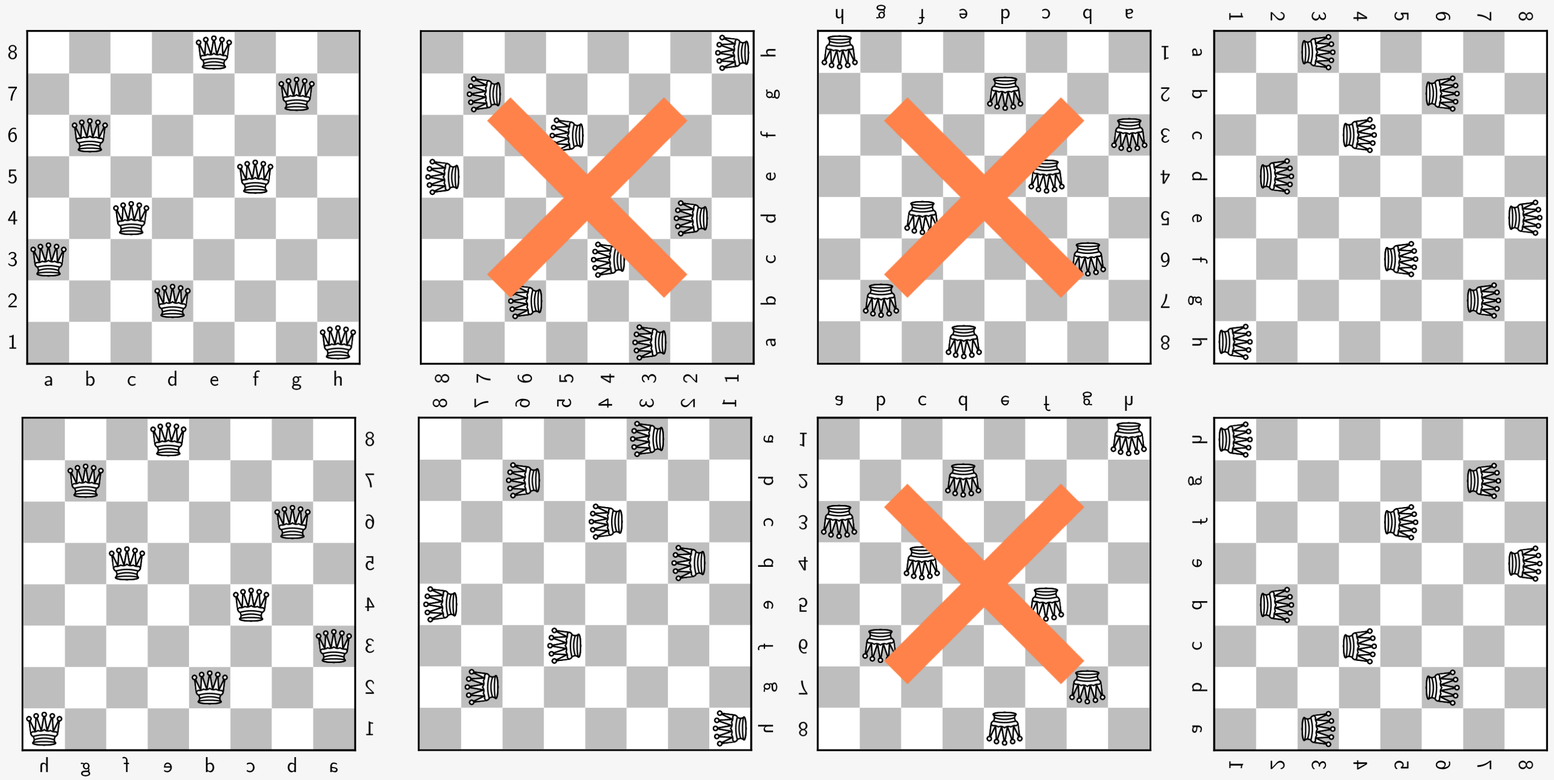


N-Queen Problem

Assigning N Queens on an NxN Board Without Them Attacking Each Other

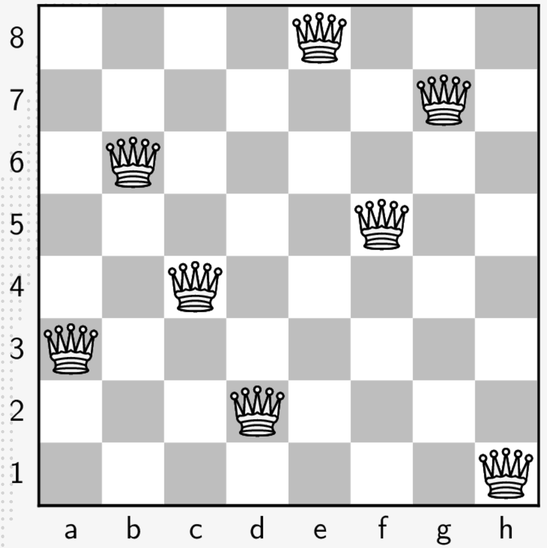


q[a]<= N/2

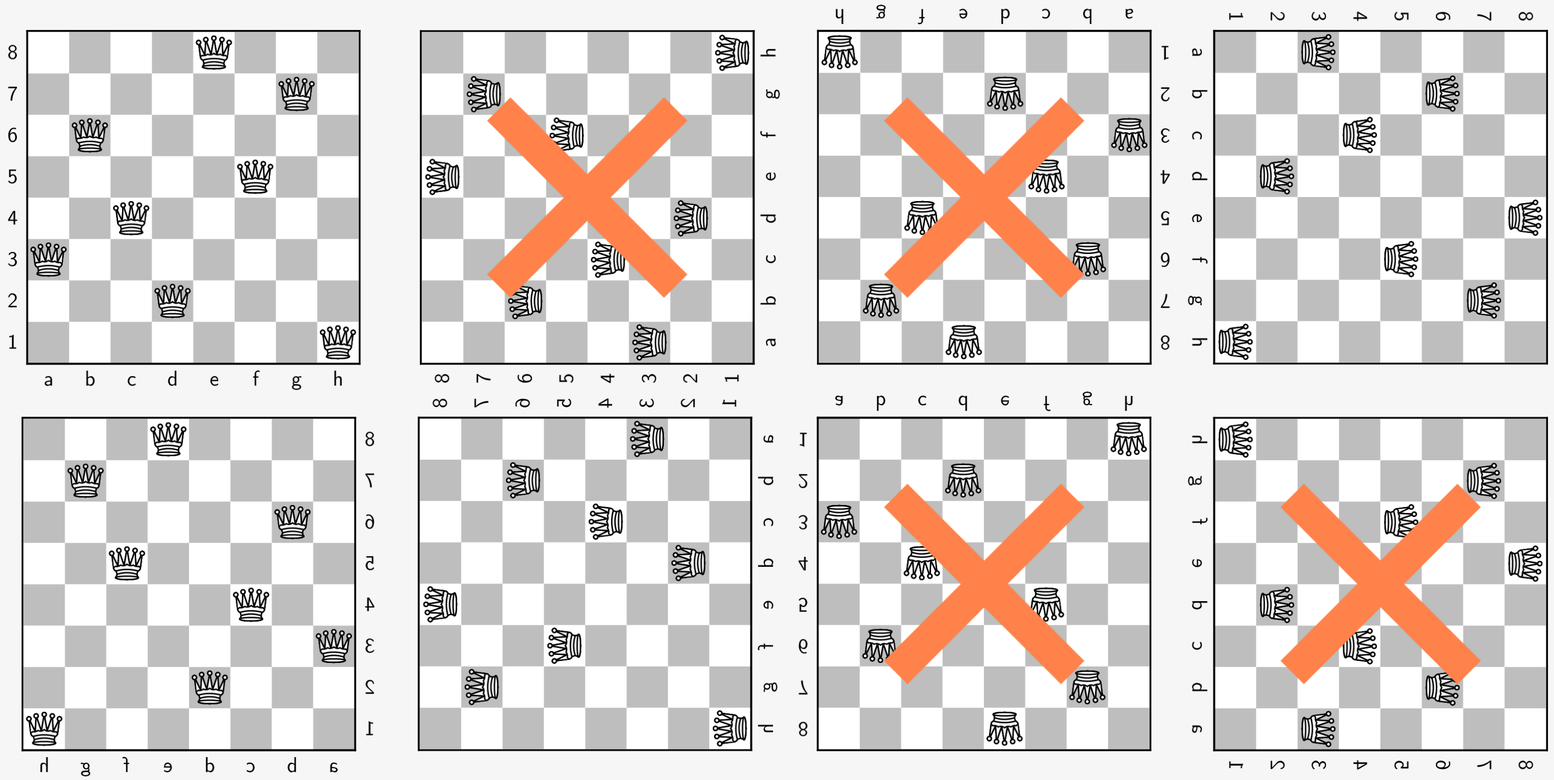


N-Queen Problem

Assigning N Queens on an NxN Board Without Them Attacking Each Other

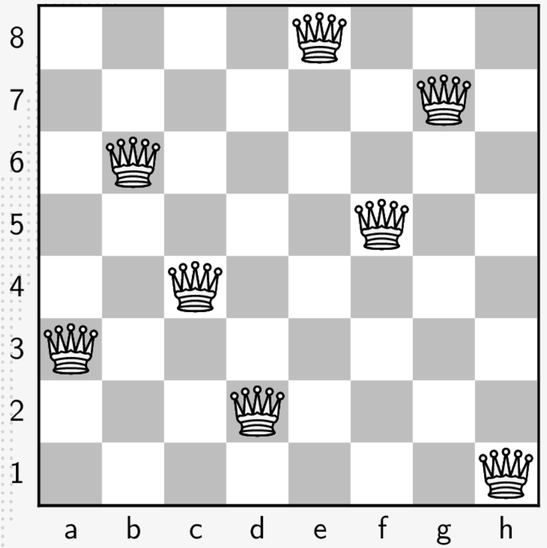


q[a]<= N/2



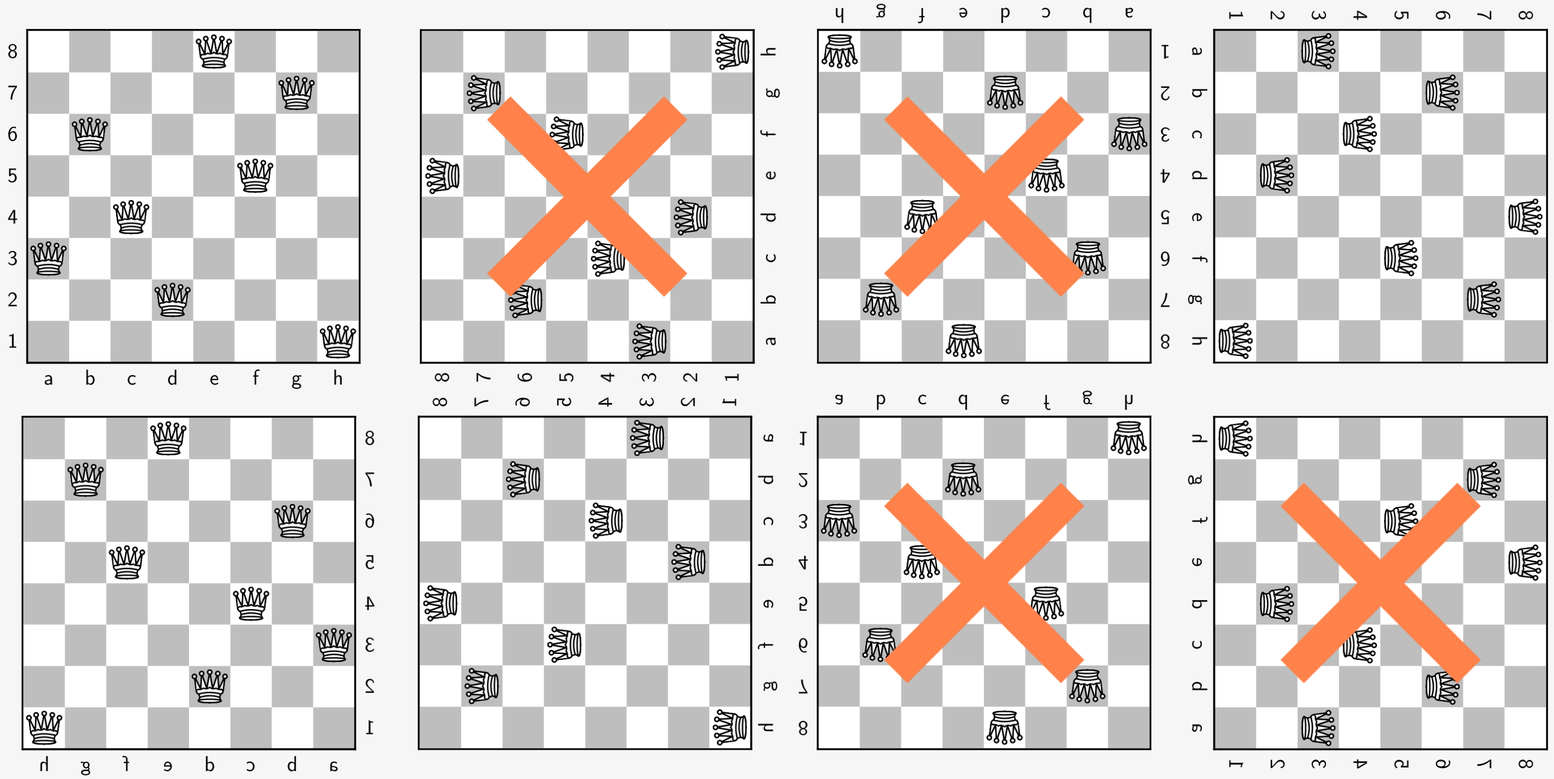
N-Queen Problem

Assigning N Queens on an NxN Board Without Them Attacking Each Other



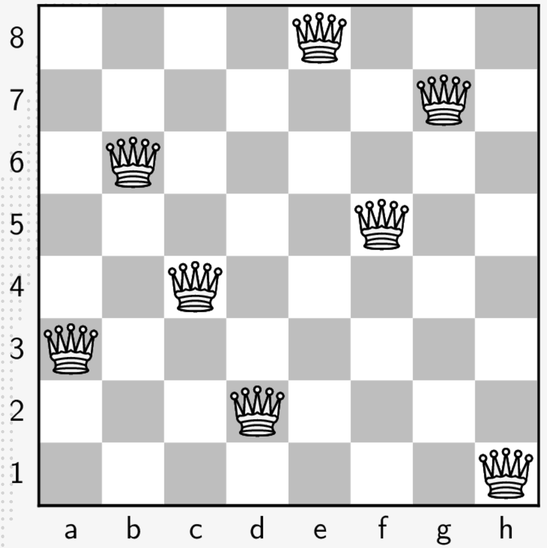
q[a]<= N/2

Lq[1]<= N/2



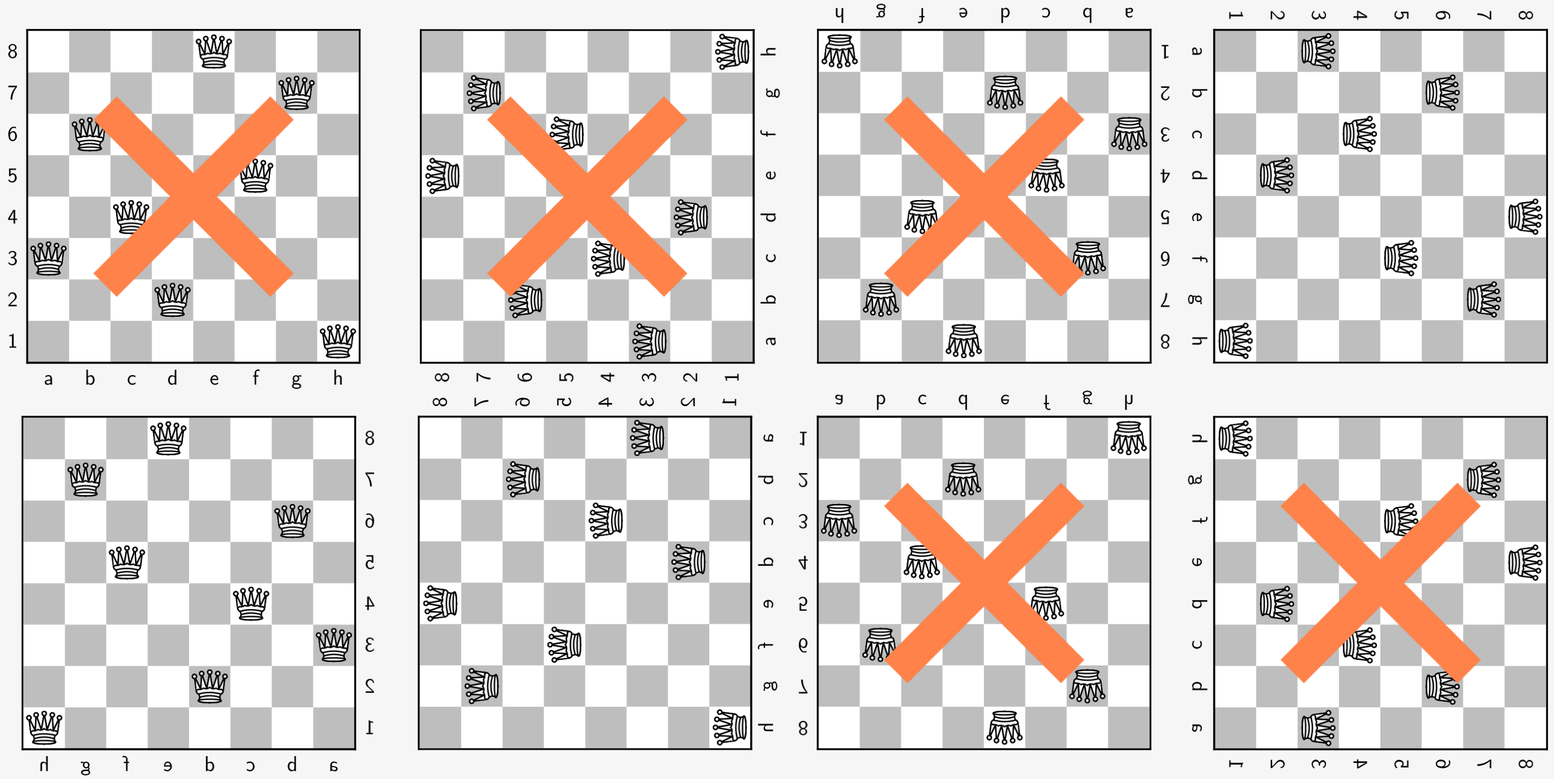
N-Queen Problem

Assigning N Queens on an NxN Board Without Them Attacking Each Other



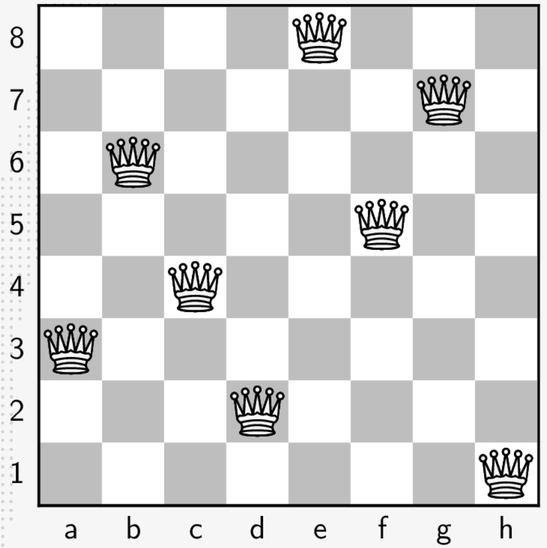
q[a]<= N/2

Lq[1]<= N/2



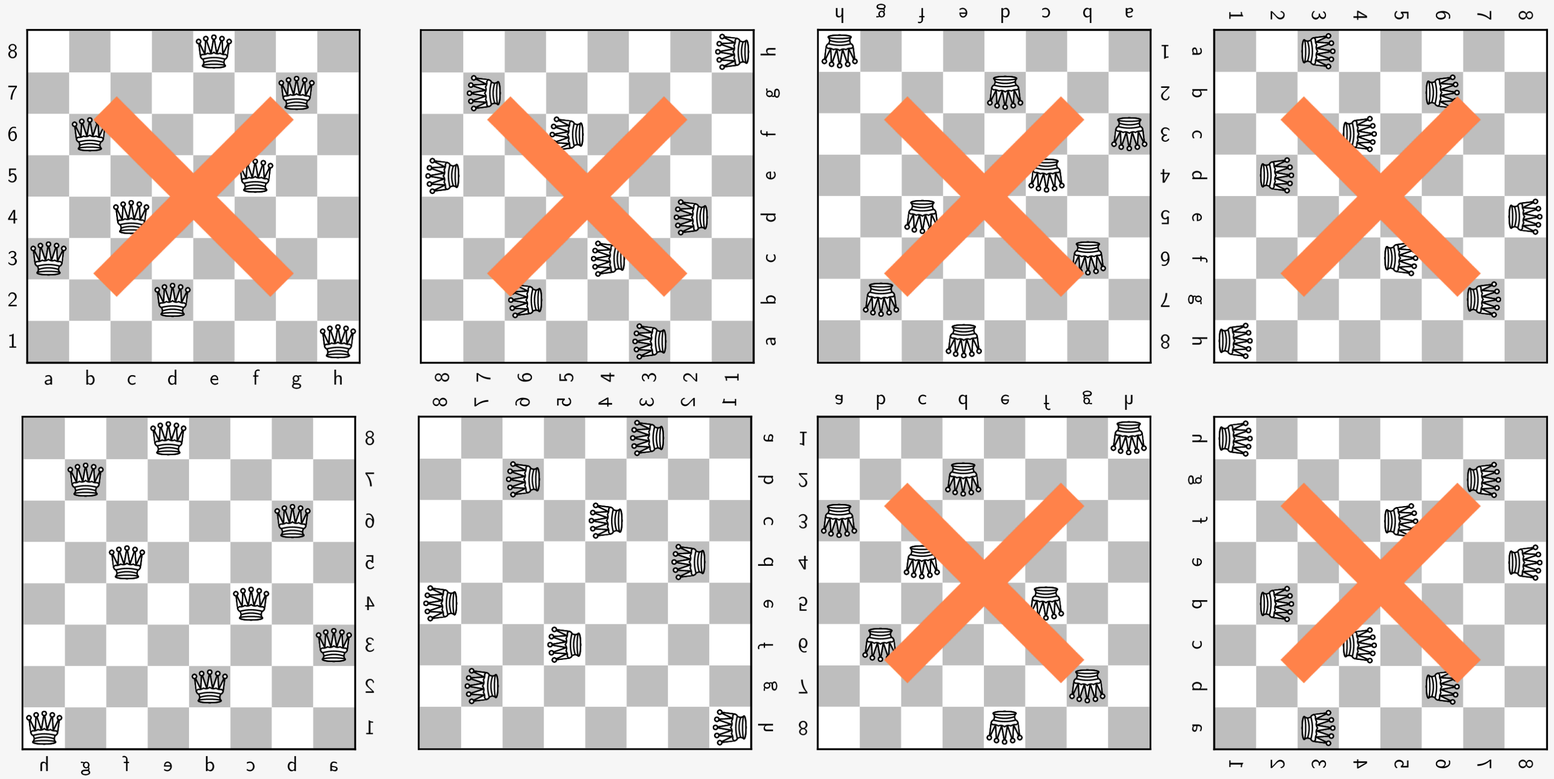
N-Queen Problem

Assigning N Queens on an NxN Board Without Them Attacking Each Other



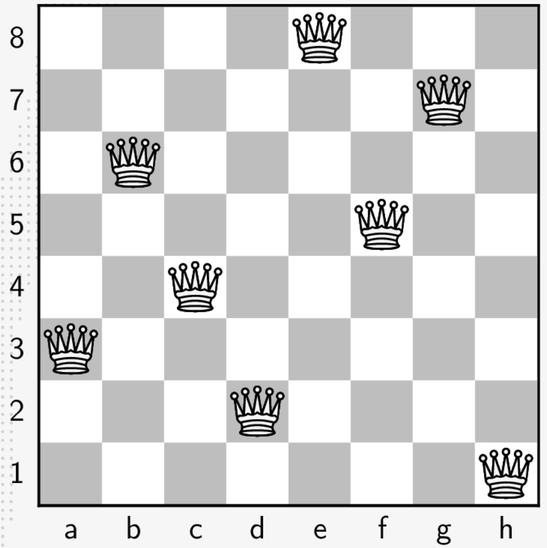
q[a]<= N/2

Lq[1]<= N/2



N-Queen Problem

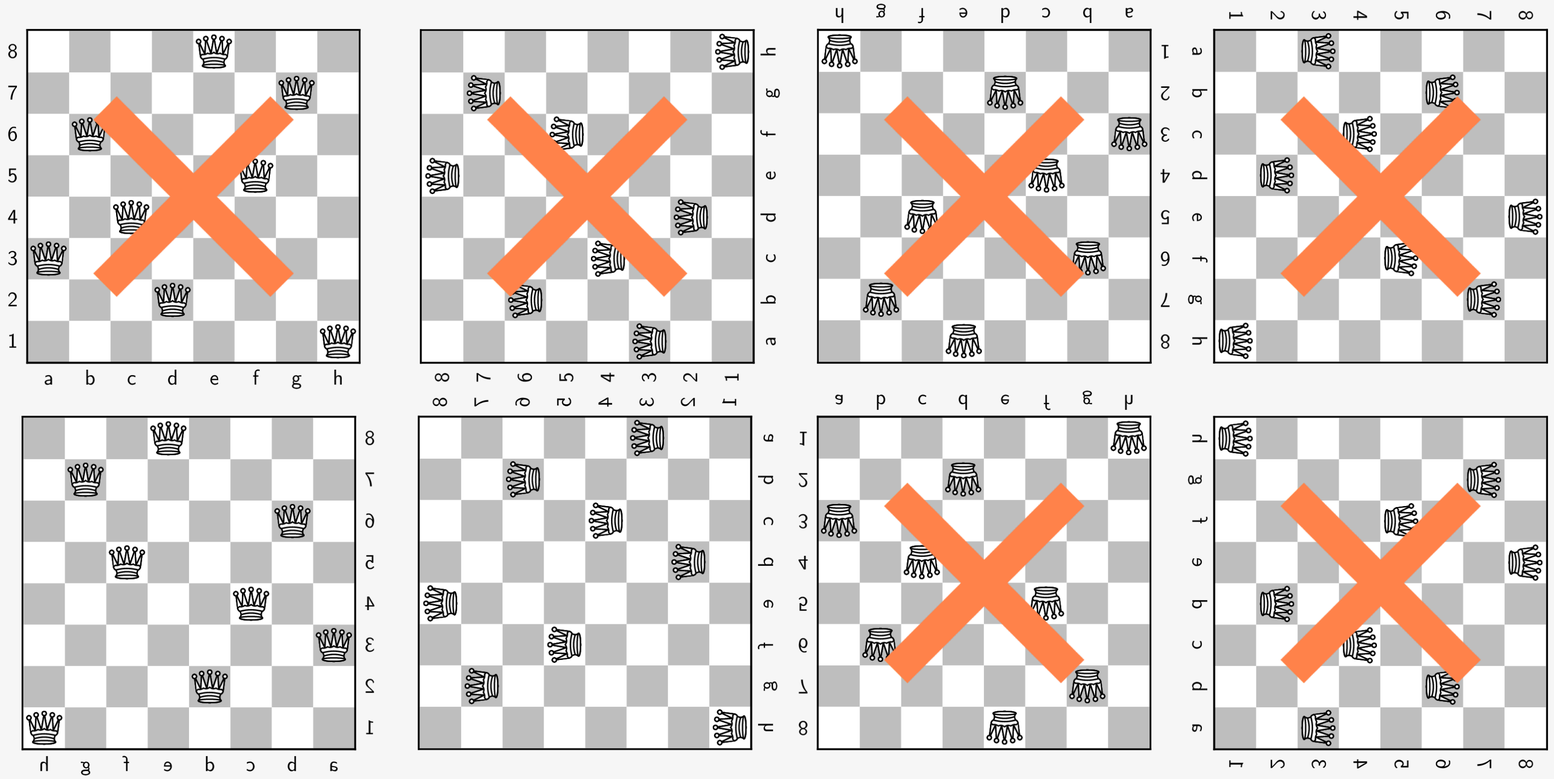
Assigning N Queens on an NxN Board Without Them Attacking Each Other



q[a]<= N/2

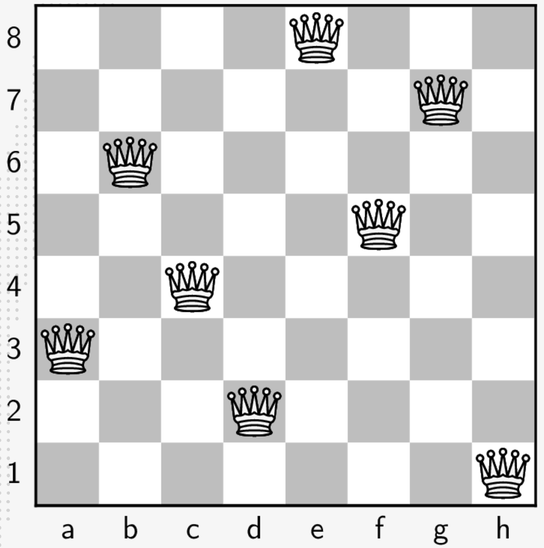
Lq[1]<= N/2

LexIncreasing([q[a],q[b]], [Lq[1],Lq[2]])



N-Queen Problem

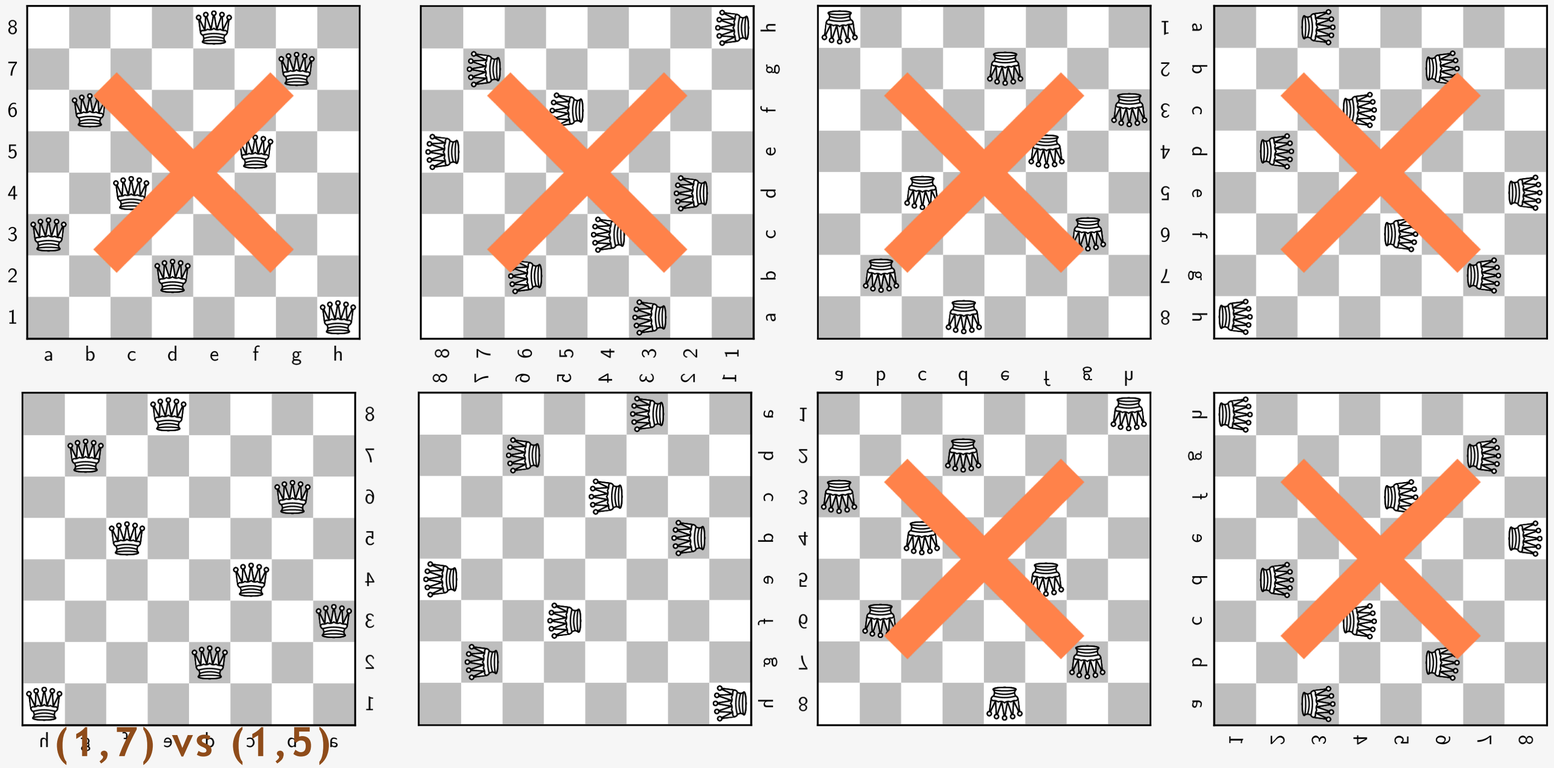
Assigning N Queens on an NxN Board Without Them Attacking Each Other



q[a]<= N/2

Lq[1]<= N/2

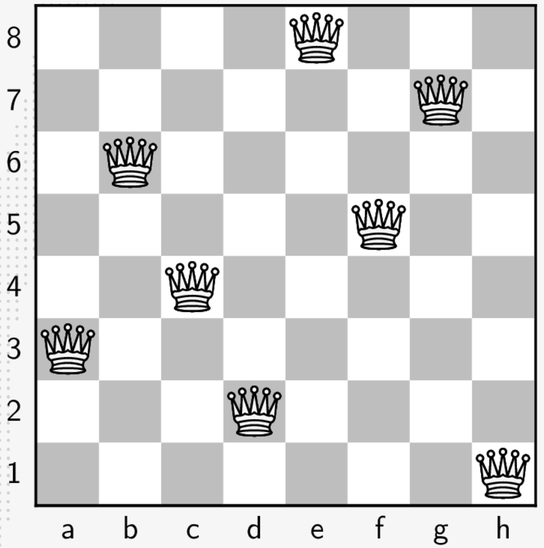
LexIncreasing([q[a],q[b]], [Lq[1],Lq[2]])



(1,7) vs (1,5)

N-Queen Problem

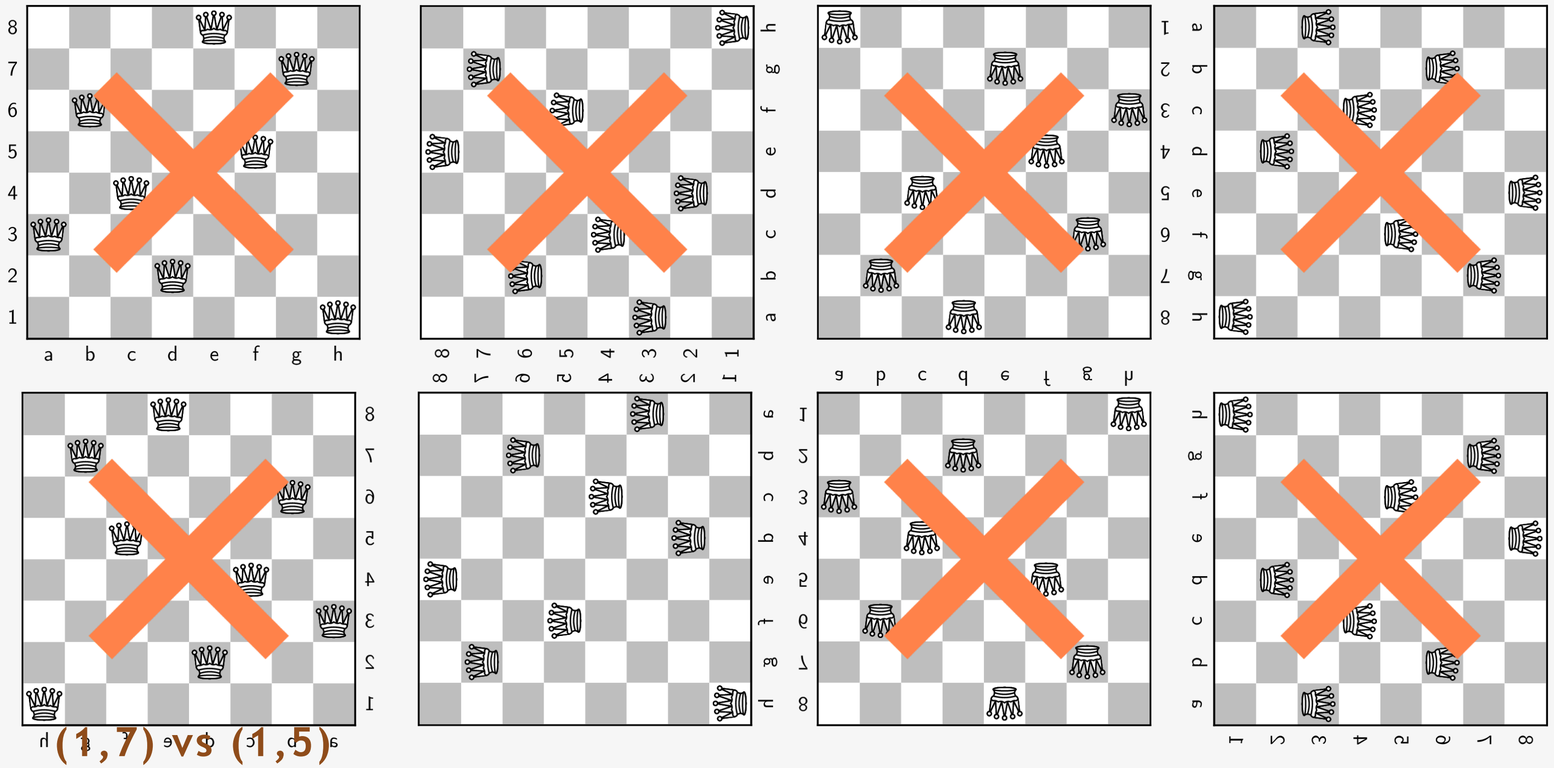
Assigning N Queens on an NxN Board Without Them Attacking Each Other



q[a]<= N/2

Lq[1]<= N/2

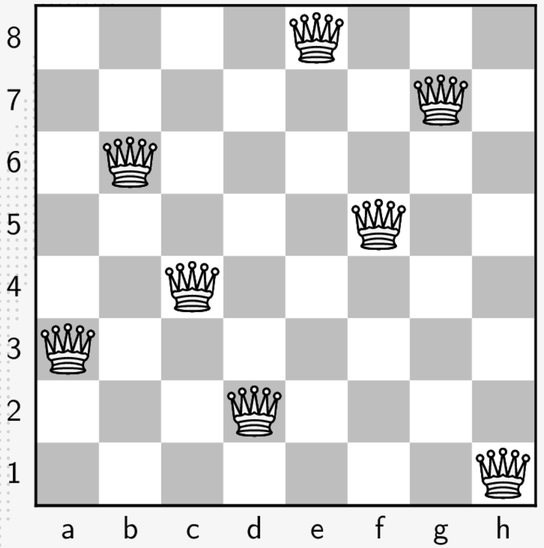
LexIncreasing([q[a],q[b]], [Lq[1],Lq[2]])



(1,7) vs (1,5)

N-Queen Problem

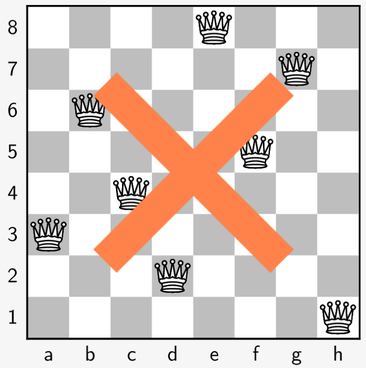
Assigning N Queens on an NxN Board Without Them Attacking Each Other

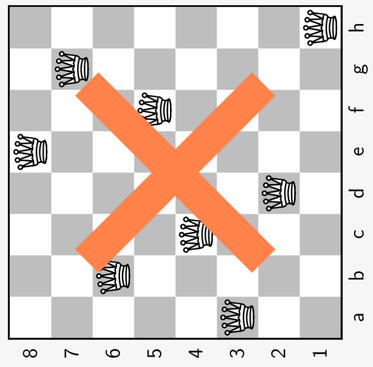


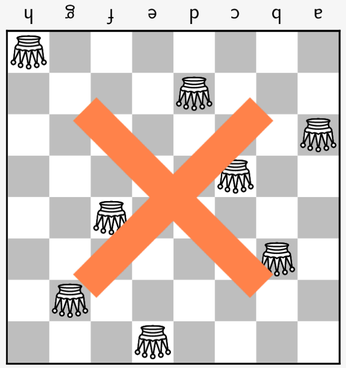
q[a]<= N/2

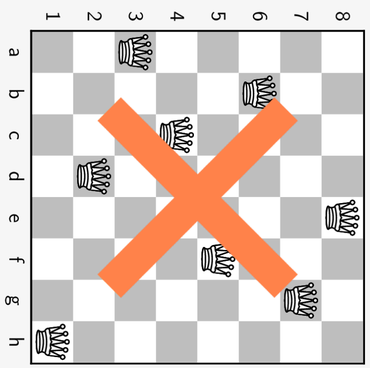
Lq[1]<= N/2

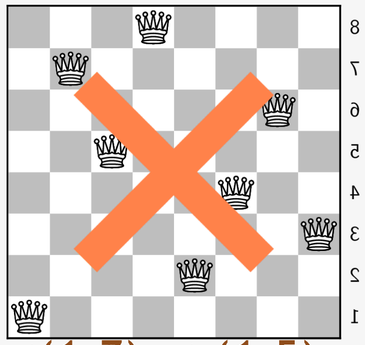
LexIncreasing([q[a],q[b]], [Lq[1],Lq[2]])



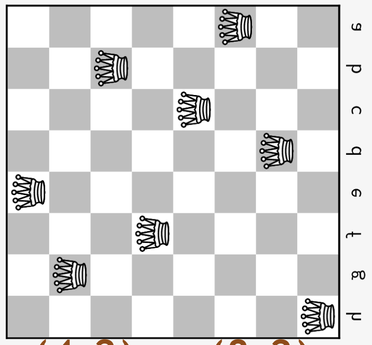




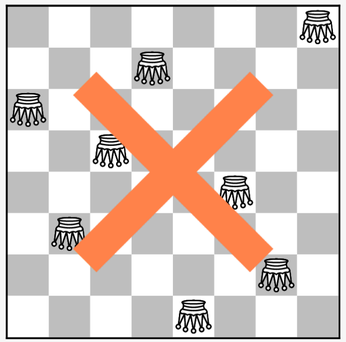


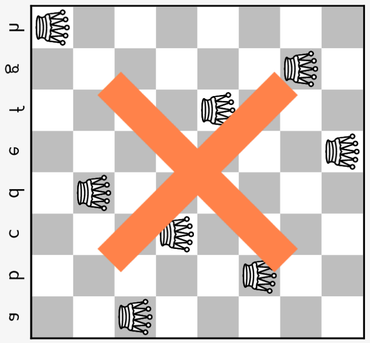


(1,7) vs (1,5)



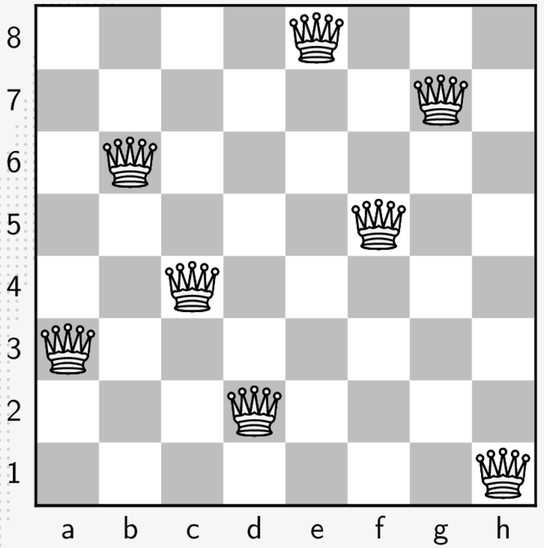
(4,2) vs (8,2)





N-Queen Problem

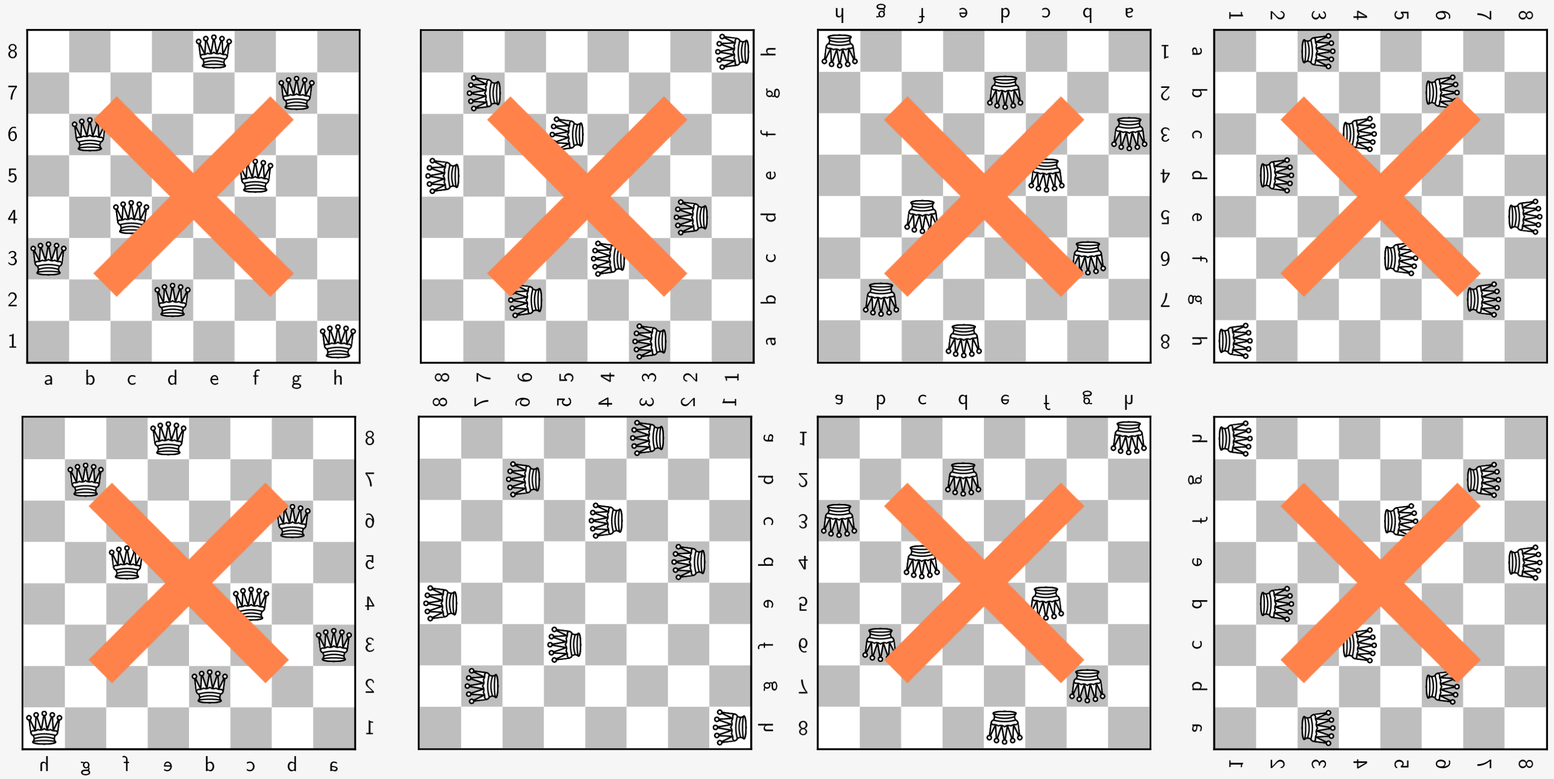
Assigning N Queens on an NxN Board Without Them Attacking Each Other



q[a]<= N/2

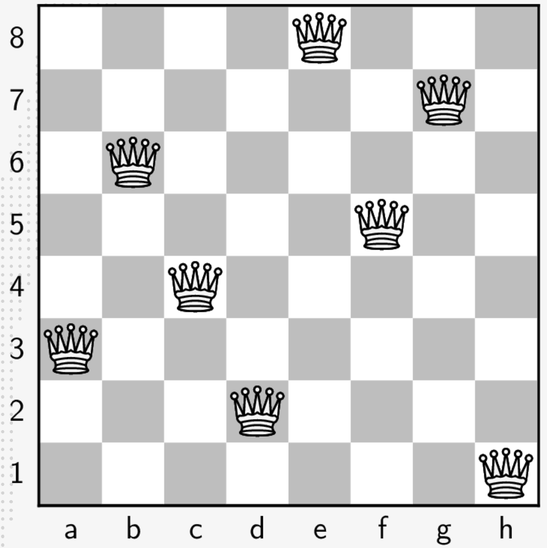
Lq[1]<= N/2

LexIncreasing([q[a],q[b]], [Lq[1],Lq[2]])



N-Queen Problem

Assigning N Queens on an NxN Board Without Them Attacking Each Other

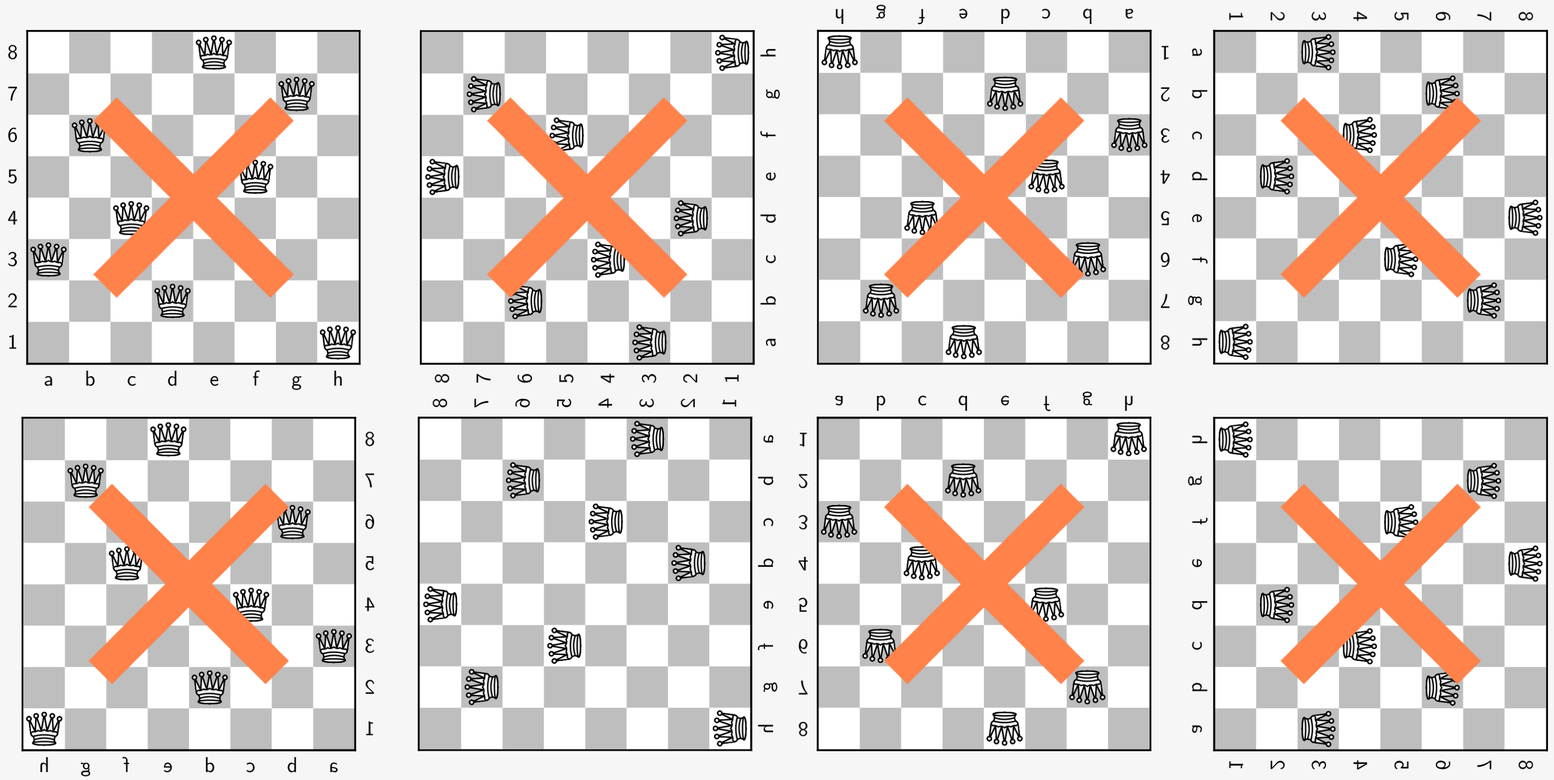


q[a]<= N/2

Lq[1]<= N/2

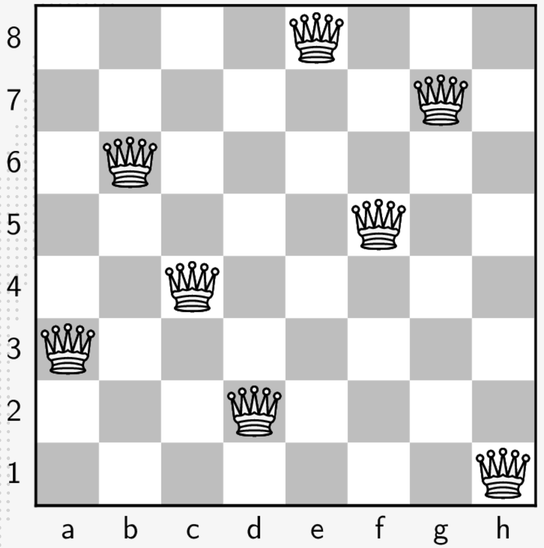
LexIncreasing([q[a],q[b]], [Lq[1],Lq[2]])

LexIncreasing([q[h],q[g]], [N-Lq[1]+1,N-Lq[2]+1])



N-Queen Problem

Assigning N Queens on an NxN Board Without Them Attacking Each Other

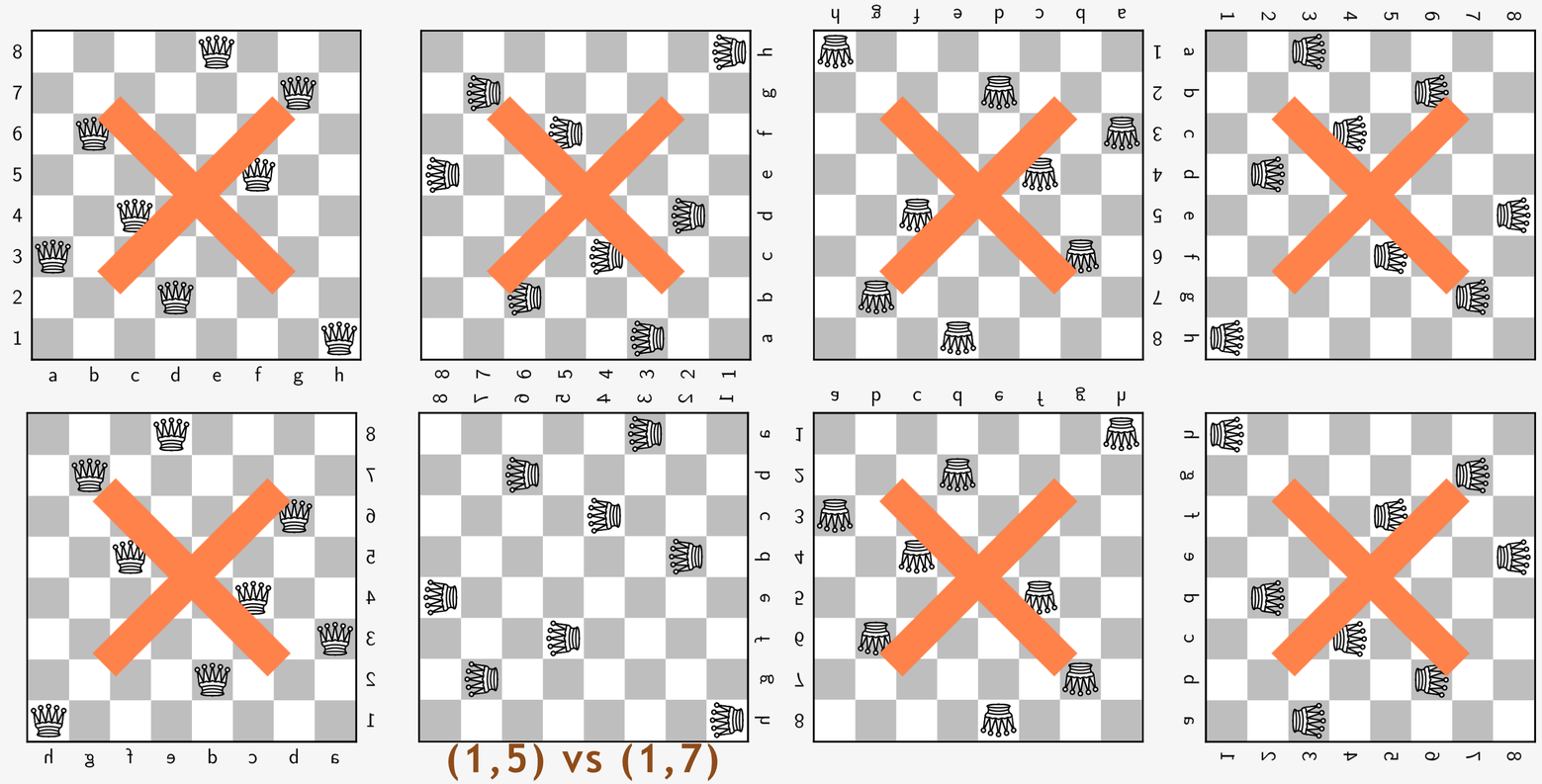


q[a]<= N/2

Lq[1]<= N/2

LexIncreasing([q[a],q[b]], [Lq[1],Lq[2]])

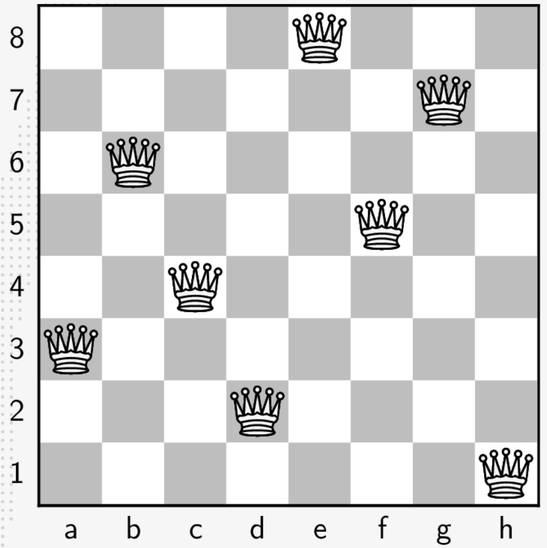
LexIncreasing([q[h],q[g]], [N-Lq[1]+1,N-Lq[2]+1])



(1,5) vs (1,7)

N-Queen Problem

Assigning N Queens on an NxN Board Without Them Attacking Each Other

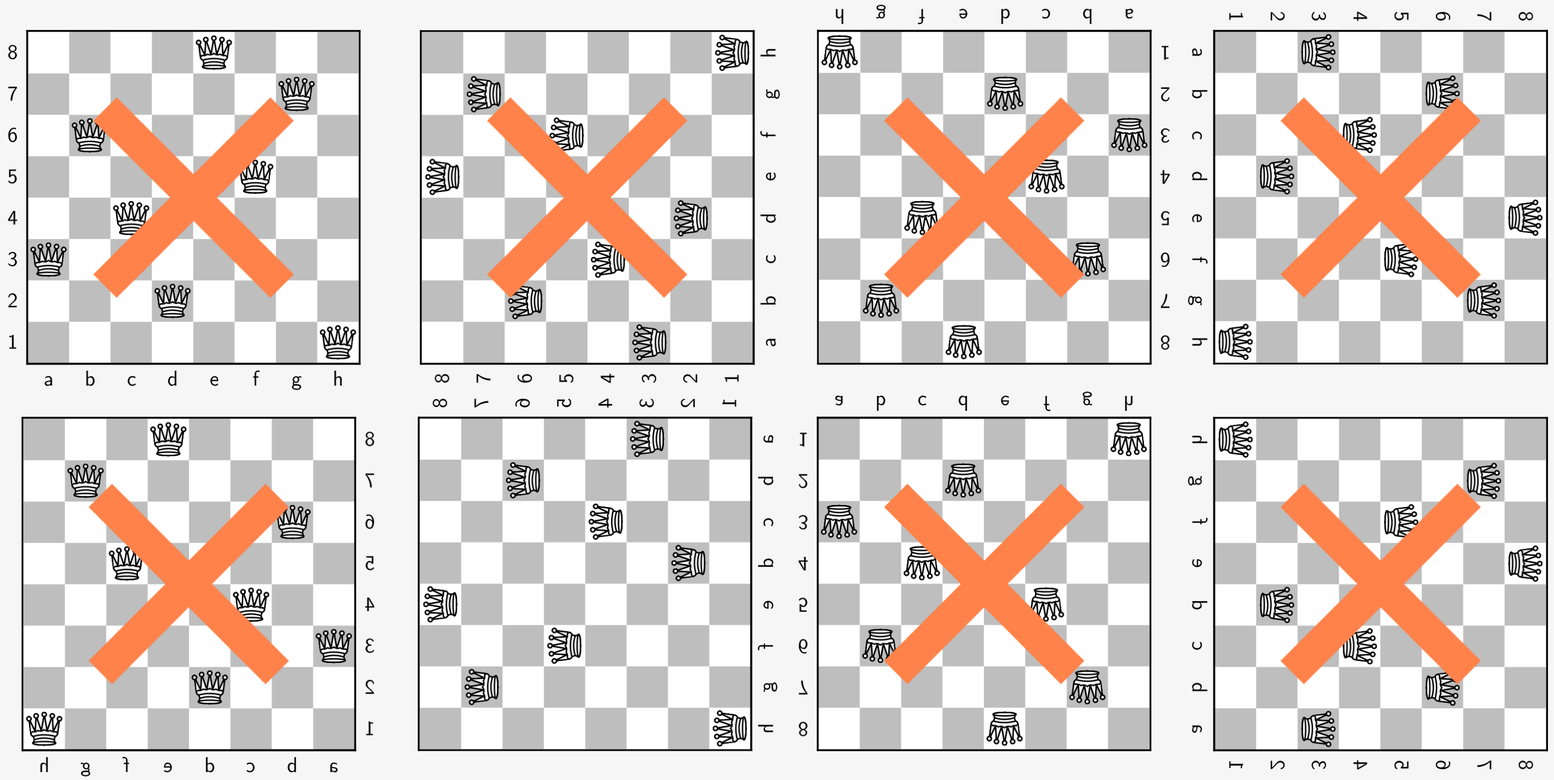


q[a]<= N/2

Lq[1]<= N/2

LexIncreasing([q[a],q[b]], [Lq[1],Lq[2]])

LexIncreasing([q[h],q[g]], [N-Lq[1]+1,N-Lq[2]+1])



LEX-LEADER

Given a problem with N variables X, if the permutation of variables $\sigma(X)$ is also solution, then adding a lexicographic ordering constraint between the two break the symmetry

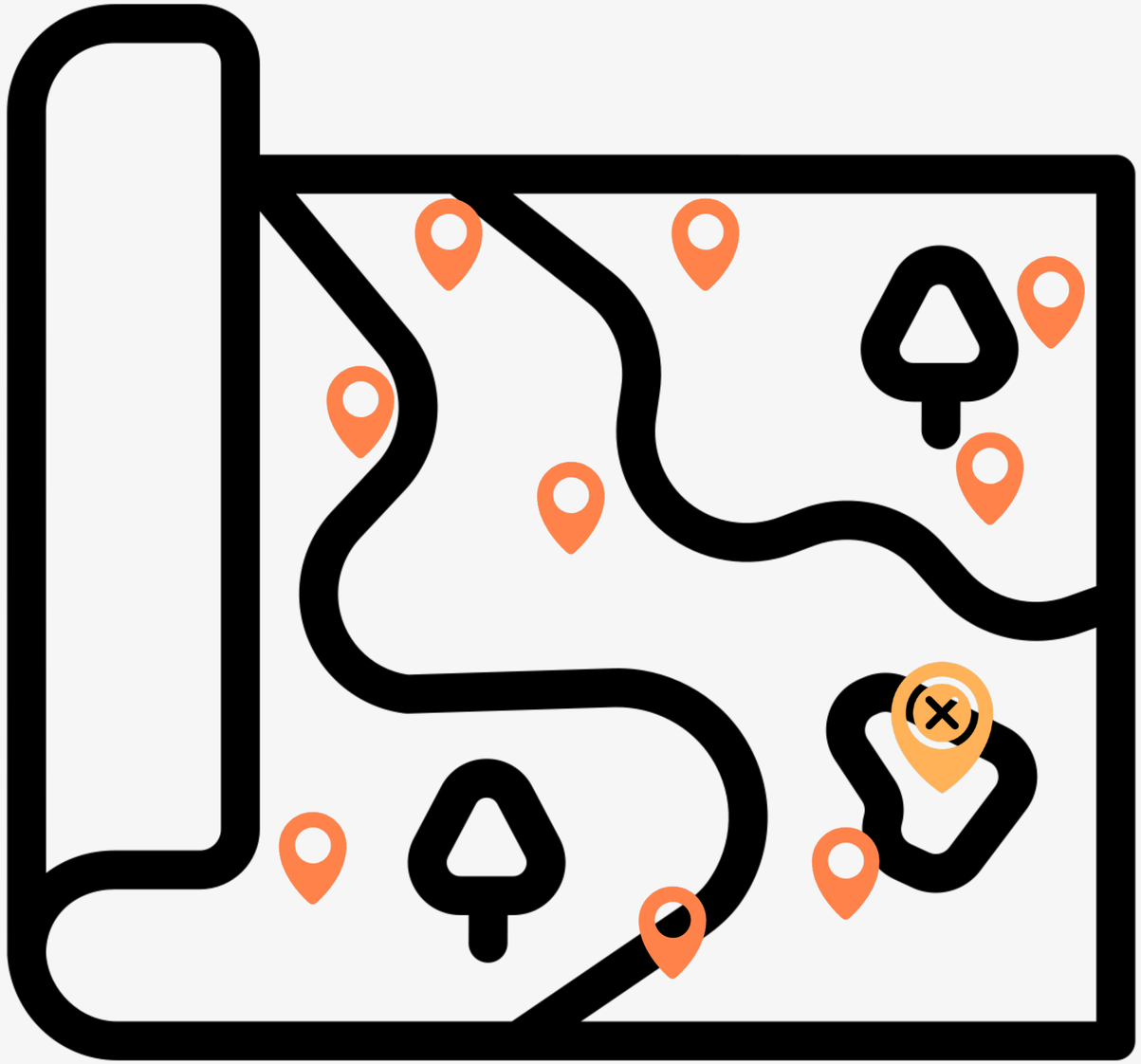
$$\text{LexIncreasing}(X,\sigma(X))$$

And the smallest solution (lexicographically speaking) will be returned

SYMMETRY BREAKING: UTILITY?

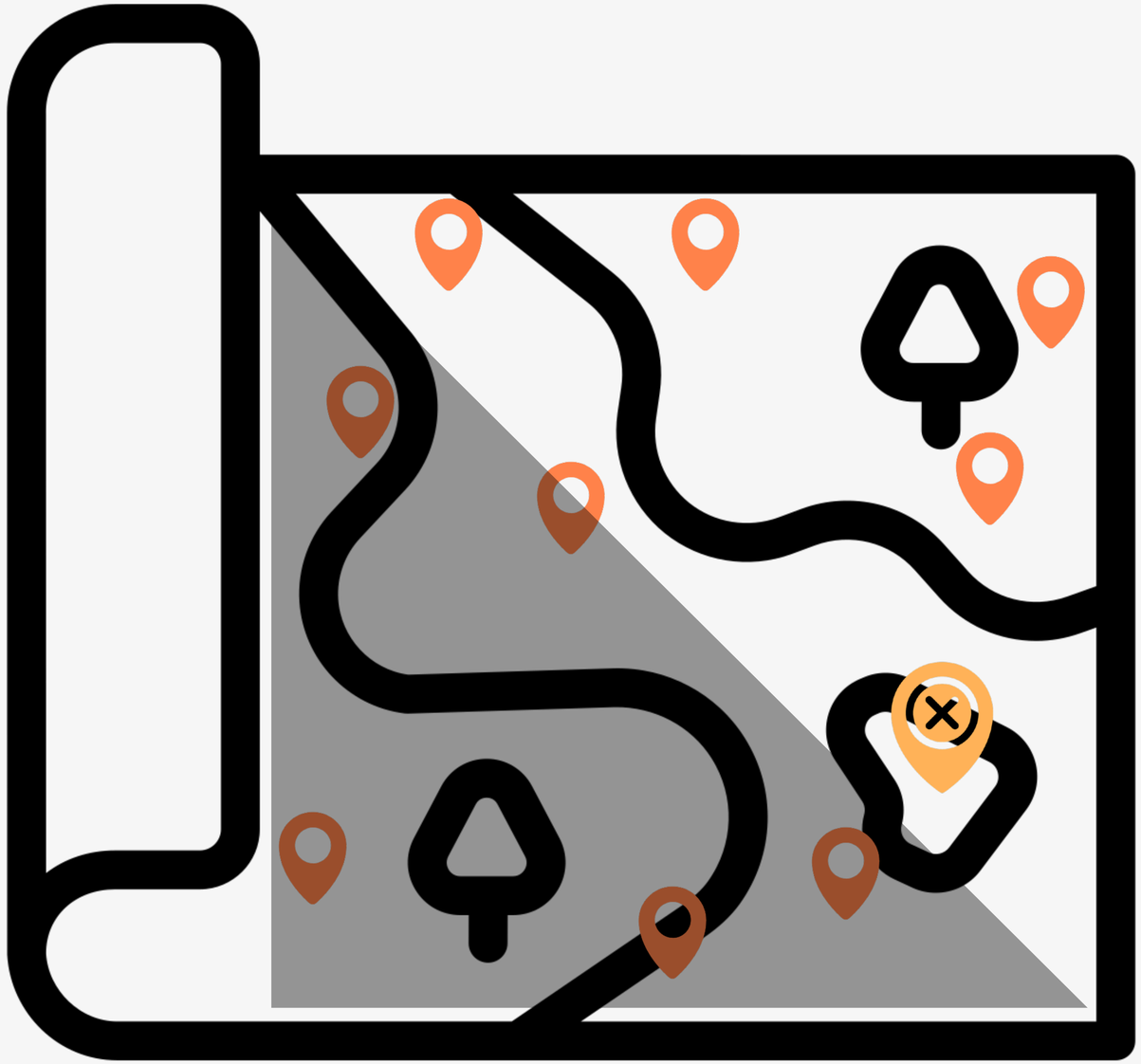
But why remove the symmetries?

SYMMETRY BREAKING: UTILITY?



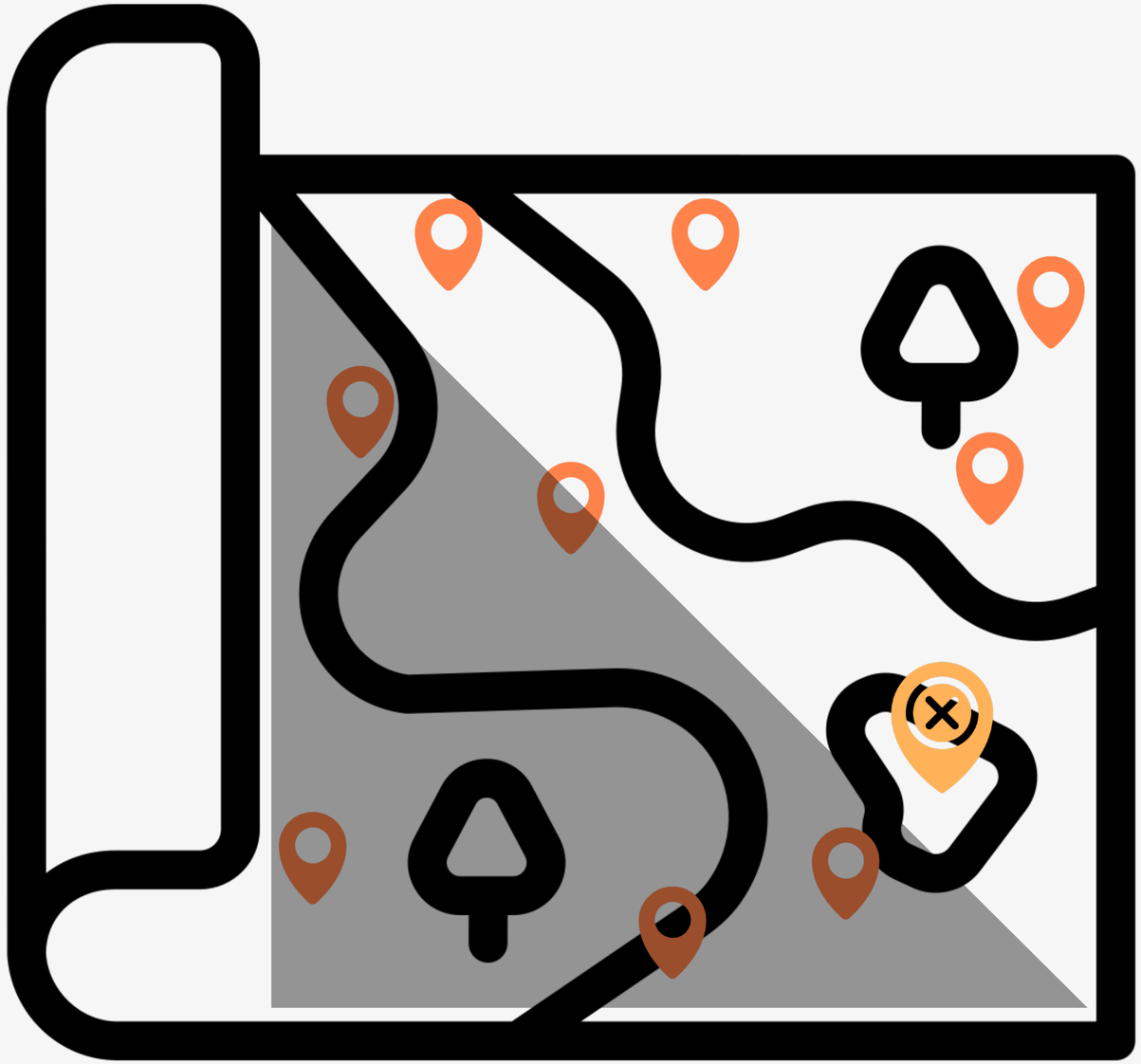
But why remove the symmetries?

SYMMETRY BREAKING: UTILITY?



But why remove the symmetries?

SYMMETRY BREAKING: UTILITY?



But why remove the symmetries?

Spoiler : it help the search!

Automatic Modelling

LARGE LANGUAGE MODEL: HOLY GRAIL 2.0?

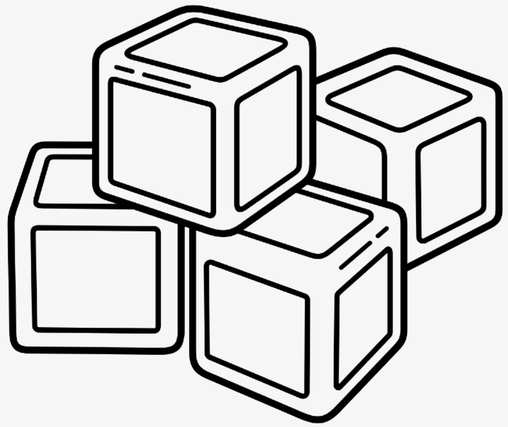
Claude

ChatGPT

Gemini

LLaMA

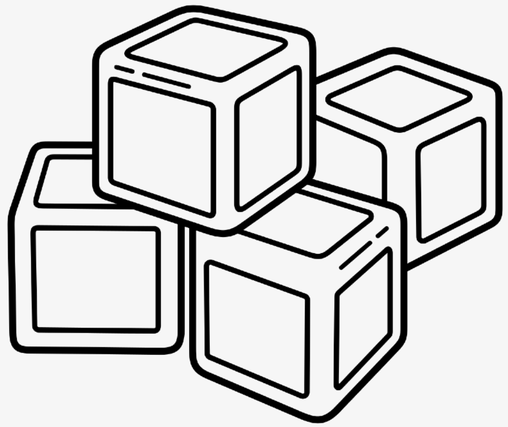
A SMALL EXAMPLE



Given the List of Word Using 24 Letters: BUOY, CAVE, CELT, FLUB, FORK, HEMP, JUDY, JUNK, LIMN, QUIP, SWAG, VISA, and WISH

Can we map the 24 letters to 4 cubes (1 letter by face of a cube) such that we can use them to form the words?

A SMALL EXAMPLE

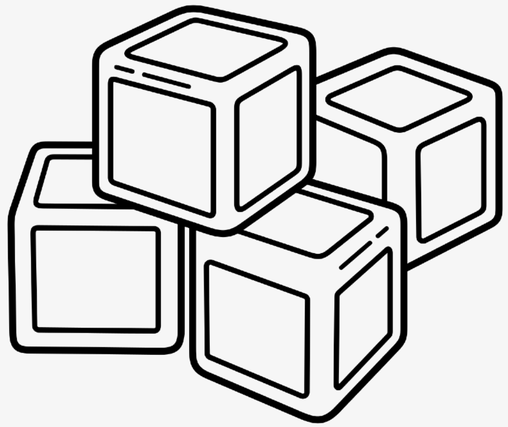


Given the List of Word Using 24 Letters: BUOY, CAVE, CELT, FLUB, FORK, HEMP, JUDY, JUNK, LIMN, QUIP, SWAG, VISA, and WISH

Can we map the 24 letters to 4 cubes (1 letter by face of a cube) such that we can use them to form the words?

What a modeller would do:

A SMALL EXAMPLE

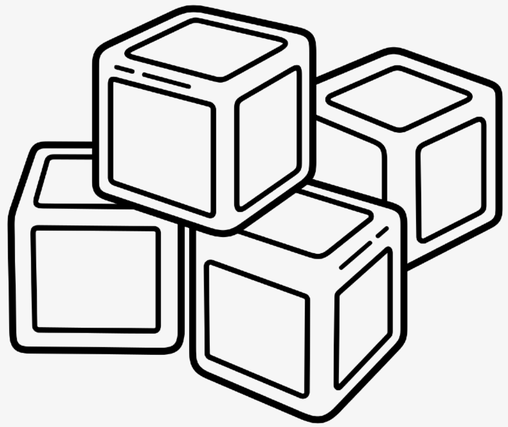


Given the List of Word Using 24 Letters: BUOY, CAVE, CELT, FLUB, FORK, HEMP, JUDY, JUNK, LIMN, QUIP, SWAG, VISA, and WISH

Can we map the 24 letters to 4 cubes (1 letter by face of a cube) such that we can use them to form the words?

What a modeller would do:
- Map each letter to an integer

A SMALL EXAMPLE

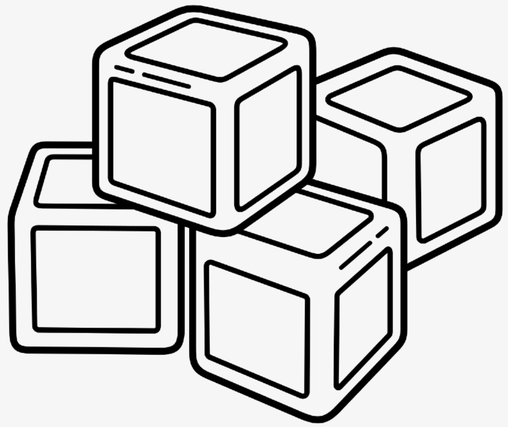


Given the List of Word Using 24 Letters: BUOY, CAVE, CELT, FLUB, FORK, HEMP, JUDY, JUNK, LIMN, QUIP, SWAG, VISA, and WISH

Can we map the 24 letters to 4 cubes (1 letter by face of a cube) such that we can use them to form the words?

What a modeller would do:
- Map each letter to an integer
- Create a variable for each letter indicating which cube they are on

A SMALL EXAMPLE

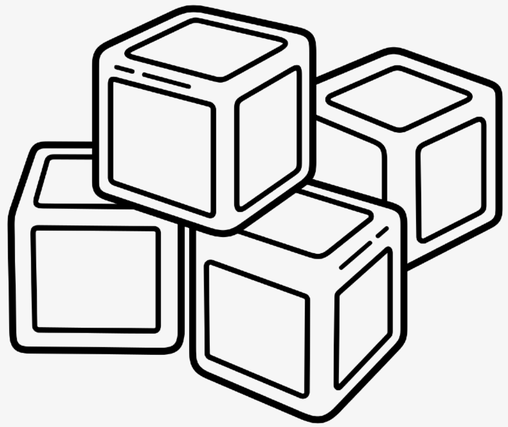


Given the List of Word Using 24 Letters: BUOY, CAVE, CELT, FLUB, FORK, HEMP, JUDY, JUNK, LIMN, QUIP, SWAG, VISA, and WISH

Can we map the 24 letters to 4 cubes (1 letter by face of a cube) such that we can use them to form the words?

What a modeller would do:
- Map each letter to an integer
- Create a variable for each letter indicating which cube they are on
- For each cube id, a count constraint to ensure every cube ID is used exactly 6 times (one letter on each face)

A SMALL EXAMPLE

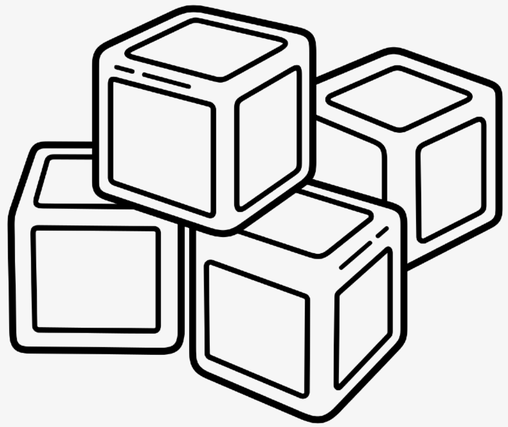


Given the List of Word Using 24 Letters: BUOY, CAVE, CELT, FLUB, FORK, HEMP, JUDY, JUNK, LIMN, QUIP, SWAG, VISA, and WISH

Can we map the 24 letters to 4 cubes (1 letter by face of a cube) such that we can use them to form the words?

What a modeller would do:
- Map each letter to an integer
- Create a variable for each letter indicating which cube they are on
- For each cube id, a count constraint to ensure every cube ID is used exactly 6 times (one letter on each face)
- For each word, an allDifferent constraints on the variables corresponding to the letters of the word (one letter per cube)

A SMALL EXAMPLE

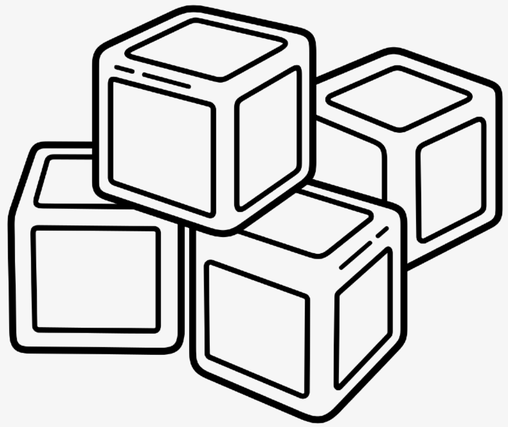


Given the List of Word Using 24 Letters: BUOY, CAVE, CELT, FLUB, FORK, HEMP, JUDY, JUNK, LIMN, QUIP, SWAG, VISA, and WISH

Can we map the 24 letters to 4 cubes (1 letter by face of a cube) such that we can use them to form the words?

What a modeller would do:
- **Map each letter to an integer**
- **Create a variable for each letter indicating which cube they are on**
- **For each cube id, a count constraint to ensure every cube ID is used exactly 6 times (one letter on each face)**
- **For each word, an allDifferent constraints on the variables corresponding to the letters of the word (one letter per cube)**
- **Symmetry breaking constraint by assigning letter i of word 1 to cube i**

A SMALL EXAMPLE



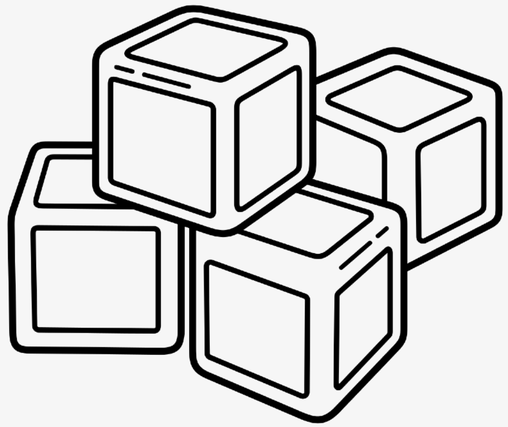
Given the List of Word Using 24 Letters: BUOY, CAVE, CELT, FLUB, FORK, HEMP, JUDY, JUNK, LIMN, QUIP, SWAG, VISA, and WISH

Can we map the 24 letters to 4 cubes (1 letter by face of a cube) such that we can use them to form the words?

What a modeller would do:
- Map each letter to an integer
- Create a variable for each letter indicating which cube they are on
- For each cube id, a count constraint to ensure every cube ID is used exactly 6 times (one letter on each face)
- For each word, an allDifferent constraints on the variables corresponding to the letters of the word (one letter per cube)
- Symmetry breaking constraint by assigning letter i of word 1 to cube i

In a more formal way:

A SMALL EXAMPLE



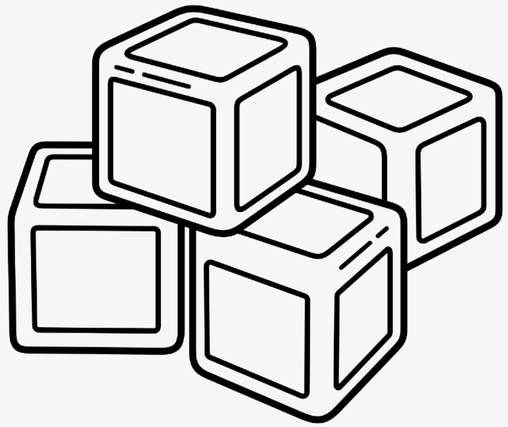
Given the List of Word Using 24 Letters: BUOY, CAVE, CELT, FLUB, FORK, HEMP, JUDY, JUNK, LIMN, QUIP, SWAG, VISA, and WISH

Can we map the 24 letters to 4 cubes (1 letter by face of a cube) such that we can use them to form the words?

What a modeller would do:
- Map each letter to an integer
- Create a variable for each letter indicating which cube they are on
- For each cube id, a count constraint to ensure every cube ID is used exactly 6 times (one letter on each face)
- For each word, an allDifferent constraints on the variables corresponding to the letters of the word (one letter per cube)
- Symmetry breaking constraint by assigning letter i of word 1 to cube i

In a more formal way:
- B=1, U=2, O=3,…

A SMALL EXAMPLE



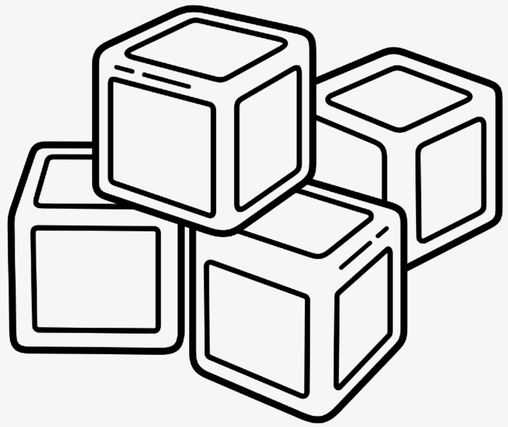
Given the List of Word Using 24 Letters: BUOY, CAVE, CELT, FLUB, FORK, HEMP, JUDY, JUNK, LIMN, QUIP, SWAG, VISA, and WISH

Can we map the 24 letters to 4 cubes (1 letter by face of a cube) such that we can use them to form the words?

What a modeller would do:
- Map each letter to an integer
- Create a variable for each letter indicating which cube they are on
- For each cube id, a count constraint to ensure every cube ID is used exactly 6 times (one letter on each face)
- For each word, an allDifferent constraints on the variables corresponding to the letters of the word (one letter per cube)
- Symmetry breaking constraint by assigning letter i of word 1 to cube i

In a more formal way:
- B=1, U=2, O=3,…
- set_of_words = {[1,2,3,4], [5, 6, 7, 8],…}

A SMALL EXAMPLE



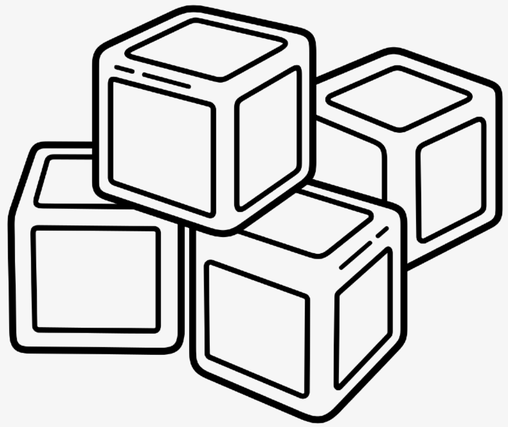
Given the List of Word Using 24 Letters: BUOY, CAVE, CELT, FLUB, FORK, HEMP, JUDY, JUNK, LIMN, QUIP, SWAG, VISA, and WISH

Can we map the 24 letters to 4 cubes (1 letter by face of a cube) such that we can use them to form the words?

What a modeller would do:
- Map each letter to an integer
- Create a variable for each letter indicating which cube they are on
- For each cube id, a count constraint to ensure every cube ID is used exactly 6 times (one letter on each face)
- For each word, an allDifferent constraints on the variables corresponding to the letters of the word (one letter per cube)
- Symmetry breaking constraint by assigning letter i of word 1 to cube i

In a more formal way:
- B=1, U=2, O=3,…
- set_of_words = {[1,2,3,4], [5, 6, 7, 8],…}
- dom(cube_of_letter[i]) = {1, 2, 3, 4} $\forall$ i $\in$ {1,…,24}

A SMALL EXAMPLE



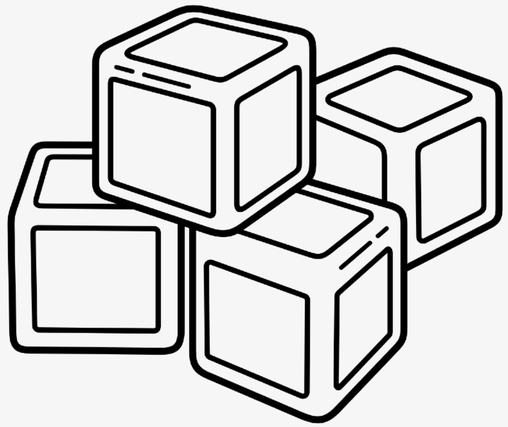
Given the List of Word Using 24 Letters: BUOY, CAVE, CELT, FLUB, FORK, HEMP, JUDY, JUNK, LIMN, QUIP, SWAG, VISA, and WISH

Can we map the 24 letters to 4 cubes (1 letter by face of a cube) such that we can use them to form the words?

What a modeller would do:
- Map each letter to an integer
- Create a variable for each letter indicating which cube they are on
- For each cube id, a count constraint to ensure every cube ID is used exactly 6 times (one letter on each face)
- For each word, an allDifferent constraints on the variables corresponding to the letters of the word (one letter per cube)
- Symmetry breaking constraint by assigning letter i of word 1 to cube i

In a more formal way:
- B=1, U=2, O=3,…
- set_of_words = {[1,2,3,4], [5, 6, 7, 8],…}
- dom(cube_of_letter[i]) = {1, 2, 3, 4} $\forall$ i $\in$ {1,…,24}
- count(cube_of_letter, cID, 6) $\forall$ cID $\in$ {1, 2, 3, 4}

A SMALL EXAMPLE



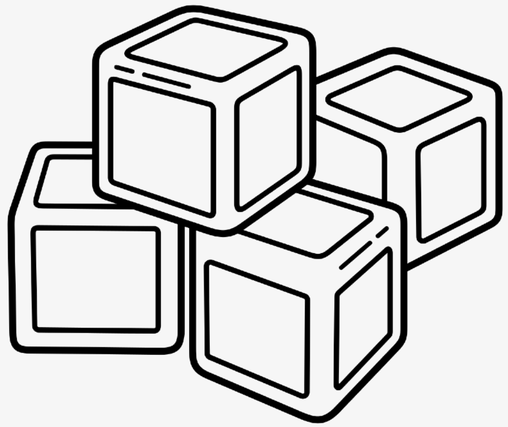
Given the List of Word Using 24 Letters: BUOY, CAVE, CELT, FLUB, FORK, HEMP, JUDY, JUNK, LIMN, QUIP, SWAG, VISA, and WISH

Can we map the 24 letters to 4 cubes (1 letter by face of a cube) such that we can use them to form the words?

What a modeller would do:
- Map each letter to an integer
- Create a variable for each letter indicating which cube they are on
- For each cube id, a count constraint to ensure every cube ID is used exactly 6 times (one letter on each face)
- For each word, an allDifferent constraints on the variables corresponding to the letters of the word (one letter per cube)
- Symmetry breaking constraint by assigning letter i of word 1 to cube i

In a more formal way:
- B=1, U=2, O=3,…
- set_of_words = {[1,2,3,4], [5, 6, 7, 8],…}
- dom(cube_of_letter[i]) = {1, 2, 3, 4} ∀ i ∈ {1,…,24}
- count(cube_of_letter, cID, 6) ∀ cID ∈ {1, 2, 3, 4}
- allDifferent([cube_of_letter[word[j]] for j ∈ {1, 2, 3, 4}]) ∀ word in set_of_words

A SMALL EXAMPLE



Given the List of Word Using 24 Letters: BUOY, CAVE, CELT, FLUB, FORK, HEMP, JUDY, JUNK, LIMN, QUIP, SWAG, VISA, and WISH

Can we map the 24 letters to 4 cubes (1 letter by face of a cube) such that we can use them to form the words?

What a modeller would do:
- Map each letter to an integer
- Create a variable for each letter indicating which cube they are on
- For each cube id, a count constraint to ensure every cube ID is used exactly 6 times (one letter on each face)
- For each word, an allDifferent constraints on the variables corresponding to the letters of the word (one letter per cube)
- Symmetry breaking constraint by assigning letter i of word 1 to cube i

In a more formal way:
- B=1, U=2, O=3,…
- set_of_words = {[1,2,3,4], [5, 6, 7, 8],…}
- dom(cube_of_letter[i]) = {1, 2, 3, 4} $\forall$ i $\in$ {1,…,24}
- count(cube_of_letter, cID, 6) $\forall$ cID $\in$ {1, 2, 3, 4}
- allDifferent([cube_of_letter[word[j]] for j $\in$ {1, 2, 3, 4}]) $\forall$ word in set_of_words
- Some symmetry breaking constraint: cube_of_letter[i] = i $\forall$ i $\in$ {1, 2, 3, 4}

ChatGPT

Partager

You are given 13 words: BUOY, CAVE, CELT, FLUB, FORK, HEMP, JUDY, JUNK,LIMN, QUIP, SWAG, VISA, and WISH. These words collectively contain 24 unique letters.The goal is to assign each of these 24 letters to one of the six faces of four different cubessuch that each letter appears on exactly one face of one cube. The assignment must ensurethat for each word, its four letters are distributed across the four different cubes. Write aMiniZinc model for solving this problem.

Got it. This is essentially a **"four cubes word puzzle"**: you have 24 unique letters that must be placed onto four cubes (6 letters per cube, 24 total). The condition is that for each word, its letters must lie on *four different cubes* — meaning no two letters of the same word may share a cube.

Let me walk step by step and then give you the **MiniZinc model**.

Seems a Good Start, Let's Continue…

ChatGPT

Partager

You are given 13 words: BUOY, CAVE, CELT, FLUB, FORK, HEMP, JUDY, JUNK,LIMN, QUIP, SWAG, VISA, and WISH. These words collectively contain 24 unique letters.The goal is to assign each of these 24 letters to one of the six faces of four different cubessuch that each letter appears on exactly one face of one cube. The assignment must ensurethat for each word, its four letters are distributed across the four different cubes. Write aMiniZinc model for solving this problem.

Got it. This is essentially a **"four cubes word puzzle"**: you have 24 unique letters that must be placed onto four cubes (6 letters per cube, 24 total). The condition is that for each word, its letters must lie on *four different cubes* — meaning no two letters of the same word may share a cube.

Let me walk step by step and then give you the **MiniZinc model**.

ChatGPT

Partager

Step 1: Extract the letters

Words:

- BUOY, CAVE, CELT, FLUB, FORK, HEMP, JUDY, JUNK, LIMN, QUIP, SWAG, VISA, WISH

Unique letters count = 24 (fits exactly into 4 cubes × 6 faces).

Step 2: Representation

- We have 24 letters, each letter must be assigned to one cube index in {1,2,3,4}.
- Each cube must get exactly 6 letters.
- For each word of length 4, all its letters must be on different cubes.

ChatGPT

Partager

Step 1: Extract the letters

Words:

- BUOY, CAVE, CELT, FLUB, FORK, HEMP, JUDY, JUNK, LIMN, QUIP, SWAG, VISA, WISH

Unique letters count = 24 (fits exactly into 4 cubes × 6 faces).

Correct Rewording of the Problem

Step 2: Representation

- We have 24 letters, each letter must be assigned to one cube index in {1,2,3,4}.
- Each cube must get exactly 6 letters.
- For each word of length 4, all its letters must be on different cubes.

ChatGPT

Partager

Step 3: MiniZinc model

We'll encode:

- Decision variables: `cube[l]` = cube assignment for letter `l`.
- Constraint: each cube has 6 letters.
- Constraint: each word's letters map to distinct cube assignments.

MiniZinc Code

minizinc — Copier — Modifier

```
% Four Cubes Word Puzzle in MiniZinc

include "alldifferent.mzn";

% 24 unique letters (we index them)
set of int: LETTERS = 1..24;

% We have 4 cubes
set of int: CUBES = 1..4;
```

ChatGPT

Partager

Step 3: MiniZinc model

We'll encode:

- Decision variables: `cube[l]` = cube assignment for letter `l`.
- Constraint: each cube has 6 letters.
- Constraint: each word's letters map to distinct cube assignments.

Good Choice of Variable

MiniZinc Code

minizinc — Copier — Modifier

```
% Four Cubes Word Puzzle in MiniZinc

include "alldifferent.mzn";

% 24 unique letters (we index them)
set of int: LETTERS = 1..24;

% We have 4 cubes
set of int: CUBES = 1..4;
```

CHATGPT AS A MODELLER

ChatGPT

Partager

Step 3: MiniZinc model

We'll encode:

- Decision variables: `cube[l]` = cube assignment for letter `l`.
- Constraint: each cube has 6 letters.
- Constraint: each word's letters map to distinct cube assignments.

MiniZinc Code

minizinc — Copier — Modifier

```
% Four Cubes Word Puzzle in MiniZinc

include "alldifferent.mzn";

% 24 unique letters (we index them)
set of int: LETTERS = 1..24;

% We have 4 cubes
set of int: CUBES = 1..4;
```

Good Choice of Variable

Plans To Use an allDifferent! Promising!

CHATGPT AS A MODELLER

ChatGPT

Partager

Copier Modifier

```
% Decision variable: cube assignment for each letter
array[LETTERS] of var CUBES: cube;

% Each cube must have exactly 6 letters
constraint
  forall(c in CUBES) (
    sum([bool2int(cube[l] = c) | l in LETTERS]) = 6
  );
```

CHATGPT AS A MODELLER

ChatGPT

Partager

Copier Modifier

```
% Decision variable: cube assignment for each letter
array[LETTERS] of var CUBES: cube;

% Each cube must have exactly 6 letters
constraint
  forall(c in CUBES) (
    sum([bool2int(cube[l] = c) | l in LETTERS]) = 6
  );
```

First Usual Mistake: Under-Use of Global Constraints

CHATGPT AS A MODELLER

ChatGPT

Partager

Copier Modifier

```
% Decision variable: cube assignment for each letter
array[LETTERS] of var CUBES: cube;

% Each cube must have exactly 6 letters
constraint
  forall(c in CUBES) (
        count_eq(cube, c, 6)
  );
```

First Usual Mistake: Under-Use of Global Constraints

CHATGPT AS A MODELLER

ChatGPT

Partager

Copier Modifier

```
% Mapping letters to indices
% We give each letter a fixed index
enum L = {B,U,O,Y,C,A,V,E,T,L2,F,L,H,M,P,J,D,K,N,Q,I,S,W,G,R};

% To avoid duplicates in CELT/LIMN we disambiguate (L2, etc.)
% Assign indices manually
array[L] of int: idx =
  [B=1,U=2,O=3,Y=4,C=5,A=6,V=7,E=8,
   T=9,L2=10,F=11,L=12,H=13,M=14,P=15,
   J=16,D=17,K=18,N=19,Q=20,I=21,S=22,
   W=23,G=24,R=25]; % adjust if extra letters

% Instead of enum mess, simpler: we define words as lists of indices
% Suppose we have precomputed the indices for each word
array[int] of set of int: words = [
  {1,2,3,4},   % BUOY
  {5,6,7,8},   % CAVE
  {8,9,10,5},  % CELT
  {11,12,2,4}, % FLUB
  {11,3,18,9}, % FORK
  {13,14,15,8},% HEMP
  {16,2,17,4}, % JUDY
  {16,2,18,19},% JUNK
  {12,21,14,19},% LIMN
  {20,2,21,15},% QUIP
  {22,23,6,24},% SWAG
  {7,21,22,6}, % VISA
  {23,21,22,4} % WISH
];
```

CHATGPT AS A MODELLER

ChatGPT

Partager

Copier Modifier

```
% Mapping letters to indices
% We give each letter a fixed index
enum L = {B,U,O,Y,C,A,V,E,T,L2,F,L,H,M,P,J,D,K,N,Q,I,S,W,G,R};

% To avoid duplicates in CELT/LIMN we disambiguate (L2, etc.)
% Assign indices manually
array[L] of int: idx =
  [B=1,U=2,O=3,Y=4,C=5,A=6,V=7,E=8,
   T=9,L2=10,F=11,L=12,H=13,M=14,P=15,
   J=16,D=17,K=18,N=19,Q=20,I=21,S=22,
   W=23,G=24,R=25]; % adjust if extra letters

% Instead of enum mess, simpler: we define words as lists of indices
% Suppose we have precomputed the indices for each word
array[int] of set of int: words = [
  {1,2,3,4},   % BUOY
  {5,6,7,8},   % CAVE
  {8,9,10,5},  % CELT
  {11,12,2,4}, % FLUB
  {11,3,18,9}, % FORK
  {13,14,15,8},% HEMP
  {16,2,17,4}, % JUDY
  {16,2,18,19},% JUNK
  {12,21,14,19},% LIMN
  {20,2,21,15},% QUIP
  {22,23,6,24},% SWAG
  {7,21,22,6}, % VISA
  {23,21,22,4} % WISH
];
```

Second Usual Mistake: Data Encoding

CHATGPT AS A MODELLER

ChatGPT

Partager

Copier Modifier

```
% Mapping letters to indices
% We give each letter a fixed index
enum L = {B,U,O,Y,C,A,V,E,T,L2,F,L,H,M,P,J,D,K,N,Q,I,S,W,G,R};

% To avoid duplicates in CELT/LIMN we disambiguate (L2, etc.)
% Assign indices manually
array[L] of int: idx =
  [B=1,U=2,O=3,Y=4,C=5,A=6,V=7,E=8,
   T=9,L2=10,F=11,L=12,H=13,M=14,P=15,
   J=16,D=17,K=18,N=19,Q=20,I=21,S=22,
   W=23,G=24,R=25]; % adjust if extra letters

% Instead of enum mess, simpler: we define words as lists of indices
% Suppose we have precomputed the indices for each word
array[int] of set of int: words = [
  {1,2,3,4},   % BUOY
  {5,6,7,8},   % CAVE
  {8,9,10,5},  % CELT
  {11,12,2,4}, % FLUB
  {11,3,18,9}, % FORK
  {13,14,15,8},% HEMP
  {16,2,17,4}, % JUDY
  {16,2,18,19},% JUNK
  {12,21,14,19},% LIMN
  {20,2,21,15},% QUIP
  {22,23,6,24},% SWAG
  {7,21,22,6}, % VISA
  {23,21,22,4} % WISH
];
```

Second Usual Mistake: Data Encoding

CHATGPT AS A MODELLER

ChatGPT

Partager

Copier Modifier

```
% Mapping letters to indices
% We give each letter a fixed index
enum L = {B,U,O,Y,C,A,V,E,T, L2, F,L,H,M,P,J,D,K,N,Q,I,S,W,G,R};

% To avoid duplicates in CELT/LIMN we disambiguate (L2, etc.)
% Assign indices manually
array[L] of int: idx =
  [B=1,U=2,O=3,Y=4,C=5,A=6,V=7,E=8,
   T=9, L2=10, F=11,L=12,H=13,M=14,P=15,
   J=16,D=17,K=18,N=19,Q=20,I=21,S=22,
   W=23,G=24,R=25]; % adjust if extra letters

% Instead of enum mess, simpler: we define words as lists of indices
% Suppose we have precomputed the indices for each word
array[int] of set of int: words = [
  {1,2,3,4},   % BUOY
  {5,6,7,8},   % CAVE
  {8,9,10,5},  % CELT
  {11,12,2,4}, % FLUB
  {11,3,18,9}, % FORK
  {13,14,15,8},% HEMP
  {16,2,17,4}, % JUDY
  {16,2,18,19},% JUNK
  {12,21,14,19},% LIMN
  {20,2,21,15},% QUIP
  {22,23,6,24},% SWAG
  {7,21,22,6}, % VISA
  {23,21,22,4} % WISH
];
```

Second Usual Mistake: Data Encoding

CHATGPT AS A MODELLER

ChatGPT

Partager

Copier Modifier

```
% Mapping letters to indices
% We give each letter a fixed index
enum L = {B,U,O,Y,C,A,V,E,T,L2,F,L,H,M,P,J,D,K,N,Q,I,S,W,G,R};

% To avoid duplicates in CELT/LIMN we disambiguate (L2, etc.)
% Assign indices manually
array[L] of int: idx =
  [B=1,U=2,O=3,Y=4,C=5,A=6,V=7,E=8,
   T=9,L2=10,F=11,L=12,H=13,M=14,P=15,
   J=16,D=17,K=18,N=19,Q=20,I=21,S=22,
   W=23,G=24,R=25]; % adjust if extra letters

% Instead of enum mess, simpler: we define words as lists of indices
% Suppose we have precomputed the indices for each word
array[int] of set of int: words = [
  {1,2,3,4},   % BUOY
  {5,6,7,8},   % CAVE
  {8,9,10,5},  % CELT
  {11,12,2,4}, % FLUB
  {11,3,18,9}, % FORK
  {13,14,15,8},% HEMP
  {16,2,17,4}, % JUDY
  {16,2,18,19},% JUNK
  {12,21,14,19},% LIMN
  {20,2,21,15},% QUIP
  {22,23,6,24},% SWAG
  {7,21,22,6}, % VISA
  {23,21,22,4} % WISH
];
```

V

Second Usual Mistake: Data Encoding

CHATGPT AS A MODELLER

ChatGPT

Partager

Copier Modifier

```
% Mapping letters to indices
% We give each letter a fixed index
enum L = {B,U,O,Y,C,A,V,E,T,L2,F,L,H,M,P,J,D,K,N,Q,I,S,W,G,R};

% To avoid duplicates in CELT/LIMN we disambiguate (L2, etc.)
% Assign indices manually
array[L] of int: idx =
  [B=1,U=2,O=3,Y=4,C=5,A=6,V=7,E=8,
   T=9,L2=10,F=11,L=12,H=13,M=14,P=15,
   J=16,D=17,K=18,N=19,Q=20,I=21,S=22,
   W=23,G=24,R=25]; % adjust if extra letters

% Instead of enum mess, simpler: we define words as lists of indices
% Suppose we have precomputed the indices for each word
array[int] of set of int: words = [
  {1,2,3,4},   % BUOY    V
  {5,6,7,8},   % CAVE    V
  {8,9,10,5},  % CELT
  {11,12,2,4}, % FLUB
  {11,3,18,9}, % FORK
  {13,14,15,8},% HEMP
  {16,2,17,4}, % JUDY
  {16,2,18,19},% JUNK
  {12,21,14,19},% LIMN
  {20,2,21,15},% QUIP
  {22,23,6,24},% SWAG
  {7,21,22,6}, % VISA
  {23,21,22,4} % WISH
];
```

Second Usual Mistake: Data Encoding

CHATGPT AS A MODELLER

ChatGPT

Partager

Copier Modifier

```
% Mapping letters to indices
% We give each letter a fixed index
enum L = {B,U,O,Y,C,A,V,E,T,L2,F,L,H,M,P,J,D,K,N,Q,I,S,W,G,R};

% To avoid duplicates in CELT/LIMN we disambiguate (L2, etc.)
% Assign indices manually
array[L] of int: idx =
  [B=1,U=2,O=3,Y=4,C=5,A=6,V=7,E=8,
   T=9,L2=10,F=11,L=12,H=13,M=14,P=15,
   J=16,D=17,K=18,N=19,Q=20,I=21,S=22,
   W=23,G=24,R=25]; % adjust if extra letters

% Instead of enum mess, simpler: we define words as lists of indices
% Suppose we have precomputed the indices for each word
array[int] of set of int: words = [
  {1,2,3,4},   % BUOY
  {5,6,7,8},   % CAVE
  {8,9,10,5},  % CELT
  {11,12,2,4}, % FLUB
  {11,3,18,9}, % FORK
  {13,14,15,8},% HEMP
  {16,2,17,4}, % JUDY
  {16,2,18,19},% JUNK
  {12,21,14,19},% LIMN
  {20,2,21,15},% QUIP
  {22,23,6,24},% SWAG
  {7,21,22,6}, % VISA
  {23,21,22,4} % WISH
];
```

V

V

X {5,8,12,9}

Second Usual Mistake: Data Encoding

CHATGPT AS A MODELLER

ChatGPT

Partager

Copier Modifier

```
% Mapping letters to indices
% We give each letter a fixed index
enum L = {B,U,O,Y,C,A,V,E,T, L2, F,L,H,M,P,J,D,K,N,Q,I,S,W,G,R};

% To avoid duplicates in CELT/LIMN we disambiguate (L2, etc.)
% Assign indices manually
array[L] of int: idx =
  [B=1,U=2,O=3,Y=4,C=5,A=6,V=7,E=8,
   T=9, L2=10, F=11,L=12,H=13,M=14,P=15,
   J=16,D=17,K=18,N=19,Q=20,I=21,S=22,
   W=23,G=24,R=25]; % adjust if extra letters

% Instead of enum mess, simpler: we define words as lists of indices
% Suppose we have precomputed the indices for each word
array[int] of set of int: words = [
  {1,2,3,4},   % BUOY    V
  {5,6,7,8},   % CAVE    V
  {8,9,10,5},  % CELT    X {5,8,12,9}
  {11,12,2,4}, % FLUB    X {11,12,2,1}
  {11,3,18,9}, % FORK
  {13,14,15,8},% HEMP
  {16,2,17,4}, % JUDY
  {16,2,18,19},% JUNK
  {12,21,14,19},% LIMN
  {20,2,21,15},% QUIP
  {22,23,6,24},% SWAG
  {7,21,22,6}, % VISA
  {23,21,22,4} % WISH
];
```

Second Usual Mistake: Data Encoding

CHATGPT AS A MODELLER

ChatGPT

Partager

Copier Modifier

```
% Mapping letters to indices
% We give each letter a fixed index
enum L = {B,U,O,Y,C,A,V,E,T, L2, F,L,H,M,P,J,D,K,N,Q,I,S,W,G,R};

% To avoid duplicates in CELT/LIMN we disambiguate (L2, etc.)
% Assign indices manually
array[L] of int: idx =
  [B=1,U=2,O=3,Y=4,C=5,A=6,V=7,E=8,
   T=9, L2=10, F=11,L=12,H=13,M=14,P=15,
   J=16,D=17,K=18,N=19,Q=20,I=21,S=22,
   W=23,G=24,R=25]; % adjust if extra letters

% Instead of enum mess, simpler: we define words as lists of indices
% Suppose we have precomputed the indices for each word
array[int] of set of int: words = [
  {1,2,3,4},   % BUOY    V
  {5,6,7,8},   % CAVE    V
  {8,9,10,5},  % CELT    X {5,8,12,9}
  {11,12,2,4}, % FLUB    X {11,12,2,1}
  {11,3,18,9}, % FORK    X {11,3,25,18}
  {13,14,15,8},% HEMP
  {16,2,17,4}, % JUDY
  {16,2,18,19},% JUNK
  {12,21,14,19},% LIMN
  {20,2,21,15},% QUIP
  {22,23,6,24},% SWAG
  {7,21,22,6}, % VISA
  {23,21,22,4} % WISH
];
```

Second Usual Mistake: Data Encoding

CHATGPT AS A MODELLER

ChatGPT

Partager

Copier Modifier

```
% Mapping letters to indices
% We give each letter a fixed index
enum L = {B,U,O,Y,C,A,V,E,T,L2,F,L,H,M,P,J,D,K,N,Q,I,S,W,G,R};

% To avoid duplicates in CELT/LIMN we disambiguate (L2, etc.)
% Assign indices manually
array[L] of int: idx =
  [B=1,U=2,O=3,Y=4,C=5,A=6,V=7,E=8,
   T=9,L2=10,F=11,L=12,H=13,M=14,P=15,
   J=16,D=17,K=18,N=19,Q=20,I=21,S=22,
   W=23,G=24,R=25]; % adjust if extra letters

% Instead of enum mess, simpler: we define words as lists of indices
% Suppose we have precomputed the indices for each word
array[int] of set of int: words = [
  {1,2,3,4},   % BUOY
  {5,6,7,8},   % CAVE
  {8,9,10,5},  % CELT
  {11,12,2,4}, % FLUB
  {11,3,18,9}, % FORK
  {13,14,15,8},% HEMP
  {16,2,17,4}, % JUDY
  {16,2,18,19},% JUNK
  {12,21,14,19},% LIMN
  {20,2,21,15},% QUIP
  {22,23,6,24},% SWAG
  {7,21,22,6}, % VISA
  {23,21,22,4} % WISH
];
```

V
V
X {5,8,12,9}
X {11,12,2,1}
X {11,3,25,18}
…

Second Usual Mistake: Data Encoding

CHATGPT AS A MODELLER

ChatGPT

Partager

Copier Modifier

```
% Mapping letters to indices
% We give each letter a fixed index
enum L = {B,U,O,Y,C,A,V,E,T,L2,F,L,H,M,P,J,D,K,N,Q,I,S,W,G,R};

% To avoid duplicates in CELT/LIMN we disambiguate (L2, etc.)
% Assign indices manually
array[L] of int: idx =
  [B=1,U=2,O=3,Y=4,C=5,A=6,V=7,E=8,
   T=9,L2=10,F=11,L=12,H=13,M=14,P=15,
   J=16,D=17,K=18,N=19,Q=20,I=21,S=22,
   W=23,G=24,R=25]; % adjust if extra letters

% Instead of enum mess, simpler: we define words as lists of indices
% Suppose we have precomputed the indices for each word
array[int] of set of int: words = [
  {1,2,3,4},   % BUOY
  {5,6,7,8},   % CAVE
  {8,9,10,5},  % CELT
  {11,12,2,4}, % FLUB
  {11,3,18,9}, % FORK
  {13,14,15,8},% HEMP
  {16,2,17,4}, % JUDY
  {16,2,18,19},% JUNK
  {12,21,14,19},% LIMN
  {20,2,21,15},% QUIP
  {22,23,6,24},% SWAG
  {7,21,22,6}, % VISA
  {23,21,22,4} % WISH
];
```

V
V
X {5,8,12,9}
X {11,12,2,1}
X {11,3,25,18}
…

Second Usual Mistake: Data Encoding

CHATGPT AS A MODELLER

ChatGPT

Partager

Copier Modifier

```
% Mapping letters to indices
% We give each letter a fixed index
enum L = {B,U,O,Y,C,A,V,E,T, L2, F,L,H,M,P,J,D,K,N,Q,I,S,W,G,R};

% To avoid duplicates in CELT/LIMN we disambiguate (L2, etc.)
% Assign indices manually
array[L] of int: idx =
  [B=1,U=2,O=3,Y=4,C=5,A=6,V=7,E=8,
   T=9, L2=10, F=11,L=12,H=13,M=14,P=15,
   J=16,D=17,K=18,N=19,Q=20,I=21,S=22,
   W=23,G=24,R=25]; % adjust if extra letters

% Instead of enum mess, simpler: we define words as lists of indices
% Suppose we have precomputed the indices for each word
array[int] of set of int: words = [
  {1,2,3,4},   % BUOY
  {5,6,7,8},   % CAVE
  {8,9,10,5},  % CELT
  {11,12,2,4}, % FLUB
  {11,3,18,9}, % FORK
  {13,14,15,8},% HEMP
  {16,2,17,4}, % JUDY
  {16,2,18,19},% JUNK
  {12,21,14,19},% LIMN
  {20,2,21,15},% QUIP
  {22,23,6,24},% SWAG
  {7,21,22,6}, % VISA
  {23,21,22,4} % WISH
];
```

V

V

X {5,8,12,9}

X {11,12,2,1}

X {11,3,25,18}

…

B=1, U=2, O=3, Y=4

Second Usual Mistake: Data Encoding

CHATGPT AS A MODELLER

ChatGPT

Partager

Copier Modifier

```
% Mapping letters to indices
% We give each letter a fixed index
enum L = {B,U,O,Y,C,A,V,E,T, L2, F,L,H,M,P,J,D,K,N,Q,I,S,W,G,R};

% To avoid duplicates in CELT/LIMN we disambiguate (L2, etc.)
% Assign indices manually
array[L] of int: idx =
  [B=1,U=2,O=3,Y=4,C=5,A=6,V=7,E=8,
   T=9, L2=10, F=11,L=12,H=13,M=14,P=15,
   J=16,D=17,K=18,N=19,Q=20,I=21,S=22,
   W=23,G=24,R=25]; % adjust if extra letters

% Instead of enum mess, simpler: we define words as lists of indices
% Suppose we have precomputed the indices for each word
array[int] of set of int: words = [
  {1,2,3,4},   % BUOY
  {5,6,7,8},   % CAVE
  {8,9,10,5},  % CELT
  {11,12,2,4}, % FLUB
  {11,3,18,9}, % FORK
  {13,14,15,8},% HEMP
  {16,2,17,4}, % JUDY
  {16,2,18,19},% JUNK
  {12,21,14,19},% LIMN
  {20,2,21,15},% QUIP
  {22,23,6,24},% SWAG
  {7,21,22,6}, % VISA
  {23,21,22,4} % WISH
];
```

V B=1, U=2, O=3, Y=4

V C=5, A=6, V=7, E=8

X {5,8,12,9}

X {11,12,2,1}

X {11,3,25,18}

…

Second Usual Mistake: Data Encoding

CHATGPT AS A MODELLER

ChatGPT

Partager

Copier Modifier

```
% Mapping letters to indices
% We give each letter a fixed index
enum L = {B,U,O,Y,C,A,V,E,T, L2, F,L,H,M,P,J,D,K,N,Q,I,S,W,G,R};

% To avoid duplicates in CELT/LIMN we disambiguate (L2, etc.)
% Assign indices manually
array[L] of int: idx =
  [B=1,U=2,O=3,Y=4,C=5,A=6,V=7,E=8,
   T=9, L2=10, F=11,L=12,H=13,M=14,P=15,
   J=16,D=17,K=18,N=19,Q=20,I=21,S=22,
   W=23,G=24,R=25]; % adjust if extra letters

% Instead of enum mess, simpler: we define words as lists of indices
% Suppose we have precomputed the indices for each word
array[int] of set of int: words = [
  {1,2,3,4},   % BUOY
  {5,6,7,8},   % CAVE
  {8,9,10,5},  % CELT
  {11,12,2,4}, % FLUB
  {11,3,18,9}, % FORK
  {13,14,15,8},% HEMP
  {16,2,17,4}, % JUDY
  {16,2,18,19},% JUNK
  {12,21,14,19},% LIMN
  {20,2,21,15},% QUIP
  {22,23,6,24},% SWAG
  {7,21,22,6}, % VISA
  {23,21,22,4} % WISH
];
```

V B=1, U=2, O=3, Y=4

V C=5, A=6, V=7, E=8

X {5,8,12,9} L=9, T=10 (Unordered?)

X {11,12,2,1}

X {11,3,25,18}

…

Second Usual Mistake: Data Encoding

CHATGPT AS A MODELLER

ChatGPT

Partager

Copier Modifier

```
% Mapping letters to indices
% We give each letter a fixed index
enum L = {B,U,O,Y,C,A,V,E,T,L2,F,L,H,M,P,J,D,K,N,Q,I,S,W,G,R};

% To avoid duplicates in CELT/LIMN we disambiguate (L2, etc.)
% Assign indices manually
array[L] of int: idx =
  [B=1,U=2,O=3,Y=4,C=5,A=6,V=7,E=8,
   T=9,L2=10,F=11,L=12,H=13,M=14,P=15,
   J=16,D=17,K=18,N=19,Q=20,I=21,S=22,
   W=23,G=24,R=25]; % adjust if extra letters

% Instead of enum mess, simpler: we define words as lists of indices
% Suppose we have precomputed the indices for each word
array[int] of set of int: words = [
  {1,2,3,4},   % BUOY
  {5,6,7,8},   % CAVE
  {8,9,10,5},  % CELT
  {11,12,2,4}, % FLUB
  {11,3,18,9}, % FORK
  {13,14,15,8},% HEMP
  {16,2,17,4}, % JUDY
  {16,2,18,19},% JUNK
  {12,21,14,19},% LIMN
  {20,2,21,15},% QUIP
  {22,23,6,24},% SWAG
  {7,21,22,6}, % VISA
  {23,21,22,4} % WISH
];
```

V B=1, U=2, O=3, Y=4
V C=5, A=6, V=7, E=8
X {5,8,12,9} L=9, T=10 (Unordered?)
X {11,12,2,1} 4 Is Y ?
X {11,3,25,18}
…

Second Usual Mistake: Data Encoding

CHATGPT AS A MODELLER

ChatGPT

Partager

Copier Modifier

```
% Mapping letters to indices
% We give each letter a fixed index
enum L = {B,U,O,Y,C,A,V,E,T, L2, F,L,H,M,P,J,D,K,N,Q,I,S,W,G,R};

% To avoid duplicates in CELT/LIMN we disambiguate (L2, etc.)
% Assign indices manually
array[L] of int: idx =
  [B=1,U=2,O=3,Y=4,C=5,A=6,V=7,E=8,
   T=9, L2=10, F=11,L=12,H=13,M=14,P=15,
   J=16,D=17,K=18,N=19,Q=20,I=21,S=22,
   W=23,G=24,R=25]; % adjust if extra letters

% Instead of enum mess, simpler: we define words as lists of indices
% Suppose we have precomputed the indices for each word
array[int] of set of int: words = [
  {1,2,3,4},   % BUOY
  {5,6,7,8},   % CAVE
  {8,9,10,5},  % CELT
  {11,12,2,4}, % FLUB
  {11,3,18,9}, % FORK
  {13,14,15,8},% HEMP
  {16,2,17,4}, % JUDY
  {16,2,18,19},% JUNK
  {12,21,14,19},% LIMN
  {20,2,21,15},% QUIP
  {22,23,6,24},% SWAG
  {7,21,22,6}, % VISA
  {23,21,22,4} % WISH
];
```

V B=1, U=2, O=3, Y=4

V C=5, A=6, V=7, E=8

X {5,8,12,9} L=9, T=10 (Unordered?)

X {11,12,2,1} 4 Is Y ?

X {11,3,25,18} ??

…

Second Usual Mistake: Data Encoding

CHATGPT AS A MODELLER

ChatGPT

Partager

Copier Modifier

```
% Mapping letters to indices
% We give each letter a fixed index
enum L = {B,U,O,Y,C,A,V,E,T, L2, F,L,H,M,P,J,D,K,N,Q,I,S,W,G,R};

% To avoid duplicates in CELT/LIMN we disambiguate (L2, etc.)
% Assign indices manually
array[L] of int: idx =
  [B=1,U=2,O=3,Y=4,C=5,A=6,V=7,E=8,
   T=9, L2=10, F=11,L=12,H=13,M=14,P=15,
   J=16,D=17,K=18,N=19,Q=20,I=21,S=22,
   W=23,G=24,R=25]; % adjust if extra letters

% Instead of enum mess, simpler: we define words as lists of indices
% Suppose we have precomputed the indices for each word
array[int] of set of int: words = [
  {1,2,3,4},   % BUOY
  {5,6,7,8},   % CAVE
  {8,9,10,5},  % CELT
  {11,12,2,4}, % FLUB
  {11,3,18,9}, % FORK
  {13,14,15,8},% HEMP
  {16,2,17,4}, % JUDY
  {16,2,18,19},% JUNK
  {12,21,14,19},% LIMN
  {20,2,21,15},% QUIP
  {22,23,6,24},% SWAG
  {7,21,22,6}, % VISA
  {23,21,22,4} % WISH
];
```

V B=1, U=2, O=3, Y=4

V C=5, A=6, V=7, E=8

X {5,8,12,9} L=9, T=10 (Unordered?)

X {11,12,2,1} 4 Is Y ?

X {11,3,25,18} ??

…

Second Usual Mistake: Data Encoding

CHATGPT AS A MODELLER

ChatGPT

Partager

Copier Modifier

```
% Mapping letters to indices
% We give each letter a fixed index
enum L = {B,U,O,Y,C,A,V,E,T, L2, F,L,H,M,P,J,D,K,N,Q,I,S,W,G,R};

% To avoid duplicates in CELT/LIMN we disambiguate (L2, etc.)
% Assign indices manually
array[L] of int: idx =
  [B=1,U=2,O=3,Y=4,C=5,A=6,V=7,E=8,
   T=9, L2=10, F=11,L=12,H=13,M=14,P=15,
   J=16,D=17,K=18,N=19,Q=20,I=21,S=22,
   W=23,G=24,R=25]; % adjust if extra letters

% Instead of enum mess, simpler: we define words as lists of indices
% Suppose we have precomputed the indices for each word
array[int] of set of int: words = [
  {1,2,3,4},   % BUOY
  {5,6,7,8},   % CAVE
  {8,9,10,5},  % CELT
  {11,12,2,4}, % FLUB
  {11,3,18,9}, % FORK
  {13,14,15,8},% HEMP
  {16,2,17,4}, % JUDY
  {16,2,18,19},% JUNK
  {12,21,14,19},% LIMN
  {20,2,21,15},% QUIP
  {22,23,6,24},% SWAG
  {7,21,22,6}, % VISA
  {23,21,22,4} % WISH
];
```

V B=1, U=2, O=3, Y=4

V C=5, A=6, V=7, E=8

X {5,8,12,9} L=9, T=10 (Unordered?)

X {11,12,2,1} 4 Is Y ?

X {11,3,25,18} ??

…

Proof: BOYD all Letters Are Reused, 1 Is Only Once

Second Usual Mistake: Data Encoding

CHATGPT AS A MODELLER

ChatGPT

Partager

Copier Modifier

```
% Constraint: each word's letters must be on different cubes
constraint
  forall(w in index_set(words)) (
    alldifferent([ cube[l] | l in words[w] ])
  );

solve satisfy;

output [
  "Letter assignments:\n"
] ++
[
  "Letter " ++ show(l) ++ " -> Cube " ++ show(cube[l]) ++ "\n"
  | l in LETTERS
];
```

CHATGPT AS A MODELLER

ChatGPT

Partager

Copier Modifier

```
% Constraint: each word's letters must be on different cubes
constraint
  forall(w in index_set(words)) (
    alldifferent([ cube[l] | l in words[w] ])
  );

solve satisfy;

output [
  "Letter assignments:\n"
] ++
[
  "Letter " ++ show(l) ++ " -> Cube " ++ show(cube[l]) ++ "\n"
  | l in LETTERS
];
```

Good! This Times It Used a Global!

CHATGPT AS A MODELLER

ChatGPT

Partager

Copier Modifier

```
% Constraint: each word's letters must be on different cubes
constraint
  forall(w in index_set(words)) (
    alldifferent([ cube[l] | l in words[w] ])
  );

solve satisfy;

output [
  "Letter assignments:\n"
] ++
[
  "Letter " ++ show(l) ++ " -> Cube " ++ show(cube[l]) ++ "\n"
  | l in LETTERS
];
```

Good! This Times It Used a Global!

Third Usual Mistake: no Advanced Modelling (Symmetry Breaking)

ChatGPT

Partager

Notes

1. You'll need to **fix the letter-to-index mapping carefully** so that `words` corresponds exactly to indices 1..24. I gave the structure, but you must clean up duplicates like `L` in CELT vs. LIMN by giving them the same index (not two).
2. Each `word` constraint uses `alldifferent` to force all 4 letters to be on different cubes.
3. The cube size constraint ensures exactly 6 letters per cube.

Would you like me to **refine this into a fully consistent MiniZinc file with the exact 24-letter set mapped properly**, so you can run it immediately without having to fix the indices yourself?

Poser une question

ChatGPT peut commettre des erreurs. Il est recommandé de vérifier les informations importantes. Voir les préférences en matière de cookies.

ChatGPT

Partager

Notes

1. You'll need to **fix the letter-to-index mapping carefully** so that `words` corresponds exactly to indices 1..24. I gave the structure, but you must clean up duplicates like `L` in CELT vs. LIMN by giving them the same index (not two).
2. Each `word` constraint uses `alldifferent` to force all 4 letters to be on different cubes.
3. The cube size constraint ensures exactly 6 letters per cube.

Would you like me to **refine this into a fully consistent MiniZinc file with the exact 24-letter set mapped properly**, so you can run it immediately without having to fix the indices yourself?

Poser une question

ChatGPT peut commettre des erreurs. Il est recommandé de vérifier les informations importantes. Voir les préférences en matière de cookies.

CHATGPT AS A MODELLER

ChatGPT

Partager

Notes

1. You'll need to **fix the letter-to-index mapping carefully** so that `words` corresponds exactly to indices 1..24. I gave the structure, but you must clean up duplicates like `L` in CELT vs. LIMN by giving them the same index (not two).
2. Each `word` constraint uses `alldifferent` to force all 4 letters to be on different cubes.
3. The cube size constraint ensures exactly 6 letters per cube.

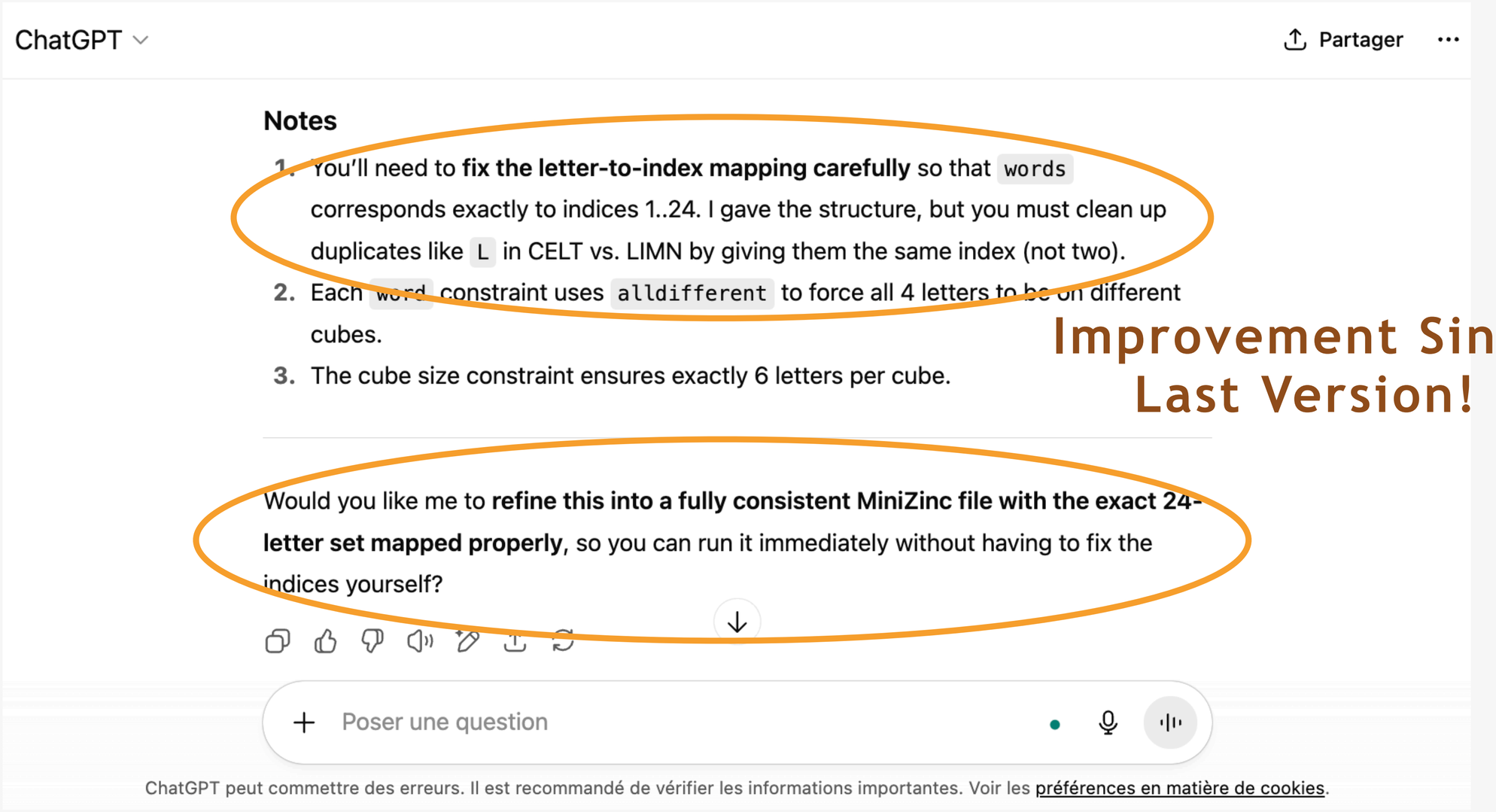
Improvement Since Last Version!

Would you like me to **refine this into a fully consistent MiniZinc file with the exact 24-letter set mapped properly**, so you can run it immediately without having to fix the indices yourself?

Poser une question

ChatGPT peut commettre des erreurs. Il est recommandé de vérifier les informations importantes. Voir les préférences en matière de cookies.

CHATGPT AS A MODELLER



Improvement Since Last Version!

CHATGPT AS A MODELLER

ChatGPT Partager

yes, fix it

Perfect. Let's carefully build the **exact consistent MiniZinc model**.
We have 24 unique letters across the 13 words:

Words:
BUOY, CAVE, CELT, FLUB, FORK, HEMP, JUDY, JUNK, LIMN, QUIP, SWAG, VISA, WISH

Unique letters (24 total):
A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, U, V, W, Y, Z? Wait—let's check carefully.

Step 1: Collect unique letters

- BUOY → B, U, O, Y
- CAVE → C, A, V, E
- CELT → C, E, L, T
- FLUB → F, L, U, B
- FORK → F, O, R, K
- HEMP → H, E, M, P
- JUDY → J, U, D, Y
- JUNK → J, U, N, K
- LIMN → L, I, M, N
- QUIP → Q, U, I, P
- SWAG → S, W, A, G
- VISA → V, I, S, A
- WISH → W, I, S, H

Now union all:
A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, U, V, W, Y

That is **23 letters**.
But problem statement said *24 unique letters*. Let's double check.

CHATGPT AS A MODELLER

ChatGPT — Partager

yes, fix it

Perfect. Let's carefully build the **exact consistent MiniZinc model**.
We have 24 unique letters across the 13 words:

Words:
BUOY, CAVE, CELT, FLUB, FORK, HEMP, JUDY, JUNK, LIMN, QUIP, S

Unique letters (24 total):
A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, U, V, W, Y, Z? Wait- carefully.

Step 1: Collect unique letters

- BUOY → B, U, O, Y
- CAVE → C, A, V, E
- CELT → C, E, L, T
- FLUB → F, L, U, B
- FORK → F, O, R, K
- HEMP → H, E, M, P
- JUDY → J, U, D, Y
- JUNK → J, U, N, K
- LIMN → L, I, M, N
- QUIP → Q, U, I, P
- SWAG → S, W, A, G
- VISA → V, I, S, A
- WISH → W, I, S, H

Now union all:
A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, U, V, W, Y

That is **23 letters**.
But problem statement said *24 unique letters*. Let's double check.

ChatGPT — Partager

Step 2: Verify counts

- Alphabetically sorted:
  A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, U, V, W, Y

Indeed that's **23**, not 24.

Maybe we missed one?
Look at words again:

- BUOY
- CAVE
- CELT
- FLUB
- FORK
- HEMP
- JUDY
- JUNK
- LIMN
- QUIP
- SWAG
- VISA
- WISH

Yes, looks like Z is missing, X is missing, T is included, but Y present.
I think the problem statement must be slightly off — we only have 23 unique letters.

CHATGPT AS A MODELLER

ChatGPT — Partager

yes, fix it

Perfect. Let's carefully build the **exact consistent MiniZinc model**.
We have 24 unique letters across the 13 words:

Words:
BUOY, CAVE, CELT, FLUB, FORK, HEMP, JUDY, JUNK, LIMN, QUIP, S

Unique letters (24 total):
A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, U, V, W, Y, Z? Wait- carefully.

Step 1: Collect unique letters

- BUOY → B, U, O, Y
- CAVE → C, A, V, E
- CELT → C, E, L, T
- FLUB → F, L, U, B
- FORK → F, O, R, K
- HEMP → H, E, M, P
- JUDY → J, U, D, Y
- JUNK → J, U, N, K
- LIMN → L, I, M, N
- QUIP → Q, U, I, P
- SWAG → S, W, A, G
- VISA → V, I, S, A
- WISH → W, I, S, H

Now union all:
A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, U, V, W, Y

That is **23 letters**.
But problem statement said *24 unique letters*. Let's double check.

ChatGPT — Partager

Step 2: Verify counts

- Alphabetically sorted:
  A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, U, V, W, Y

Indeed that's **23**, not 24.

Maybe we missed one?
Look at words again:

- BUOY
- CAVE
- CELT
- FLUB
- FORK
- HEMP
- JUDY
- JUNK
- LIMN
- QUIP
- SWAG
- VISA
- WISH

Yes, looks like Z is missing, X is missing, T is included, but Y present.
I think the problem statement must be slightly off — we only have 23 unique letters.

Step 3: Adjust assumption

- If we must fill 4 cubes × 6 sides = 24 letters, we need 24 unique letters.
- Possibly the puzzle setter miscounted, or one word has a hidden different letter?

Let's recount CELT → includes T (so T is in set).
So add T.
Yes, I forgot to list **T** earlier!

Now total = 24.

Final unique letters =
A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, Y

Exactly **24**.

Step 4: Define mapping

Let's map indices 1–24:

1: A
2: B
3: C
4: D
5: E
6: F
7: G
8: H
9: I
10: J
11: K
12: L
13: M
14: N
15: O
16: P
17: Q
18: R
19: S
20: T
21: U
22: V
23: W
24: Y

Step 5: Words as sets of indices

- BUOY → {2, 21, 15, 24}
- CAVE → {3, 1, 22, 5}
- CELT → {3, 5, 12, 20}
- FLUB → {6, 12, 21, 2}
- FORK → {6, 15, 18, 11}
- HEMP → {8, 5, 13, 16}
- JUDY → {10, 21, 4, 24}
- JUNK → {10, 21, 14, 11}
- LIMN → {12, 9, 13, 14}
- QUIP → {17, 21, 9, 16}
- SWAG → {19, 23, 1, 7}
- VISA → {22, 9, 19, 1}
- WISH → {23, 9, 19, 8}

CHATGPT AS A MODELLER

ChatGPT

Partager

Step 6: Final MiniZinc model

minizinc — Copier — Modifier

```
% Four Cubes Word Puzzle in MiniZinc

include "alldifferent.mzn";

% 24 unique letters
set of int: LETTERS = 1..24;
set of int: CUBES = 1..4;

array[LETTERS] of string: letter =
  ["A","B","C","D","E","F","G","H","I","J","K","L",
   "M","N","O","P","Q","R","S","T","U","V","W","Y"];

% Decision variable: which cube each letter belongs to
array[LETTERS] of var CUBES: cube;

% Each cube has exactly 6 letters
constraint
  forall(c in CUBES) (
    sum([bool2int(cube[l] = c) | l in LETTERS]) = 6
  );

% Word definitions (letters as indices)
array[int] of set of int: words = [
  {2,21,15,24},    % BUOY
  {3,1,22,5},      % CAVE
  {3,5,12,20},     % CELT
  {6,12,21,2},     % FLUB
  {6,15,18,11},    % FORK
  {8,5,13,16},     % HEMP
  {10,21,4,24},    % JUDY
  {10,21,14,11},   % JUNK
  {12,9,13,14},    % LIMN
  {17,21,9,16},    % QUIP
  {19,23,1,7},     % SWAG
  {22,9,19,1},     % VISA
  {23,9,19,8}      % WISH
];

% Constraint: all letters in each word must be on different cubes
constraint
  forall(w in index_set(words)) (
    alldifferent([ cube[l] | l in words[w] ])
  );

solve satisfy;

output [
  "Letter " ++ letter[l] ++ " -> Cube " ++ show(cube[l]) ++ "\n"
  | l in LETTERS
];
```

CHATGPT AS A MODELLER

ChatGPT

Partager

Step 6: Final MiniZinc model

minizinc — Copier — Modifier

```
% Four Cubes Word Puzzle in MiniZinc

include "alldifferent.mzn";

% 24 unique letters
set of int: LETTERS = 1..24;
set of int: CUBES = 1..4;

array[LETTERS] of string: letter =
  ["A","B","C","D","E","F","G","H","I","J","K","L",
   "M","N","O","P","Q","R","S","T","U","V","W","Y"];

% Decision variable: which cube each letter belongs to
array[LETTERS] of var CUBES: cube;

% Each cube has exactly 6 letters
constraint
  forall(c in CUBES) (
    sum([bool2int(cube[l] = c) | l in LETTERS]) = 6
  );
```

```
% Word definitions (letters as indices)
array[int] of set of int: words = [
  {2,21,15,24},    % BUOY
  {3,1,22,5},      % CAVE
  {3,5,12,20},     % CELT
  {6,12,21,2},     % FLUB
  {6,15,18,11},    % FORK
  {8,5,13,16},     % HEMP
  {10,21,4,24},    % JUDY
  {10,21,14,11},   % JUNK
  {12,9,13,14},    % LIMN
  {17,21,9,16},    % QUIP
  {19,23,1,7},     % SWAG
  {22,9,19,1},     % VISA
  {23,9,19,8}      % WISH
];

% Constraint: all letters in each word must be on different cubes
constraint
  forall(w in index_set(words)) (
    alldifferent([ cube[l] | l in words[w] ])
  );

solve satisfy;

output [
  "Letter " ++ letter[l] ++ " -> Cube " ++ show(cube[l]) ++ "\n"
  | l in LETTERS
];
```

Still Not the Best Model, but Correct at Least

ERRORS TO LOOK FOR

ERRORS TO LOOK FOR

LLMs are improving bits by bits, but they are still not fully reliable as CP modellers:

ERRORS TO LOOK FOR

LLMs are improving bits by bits, but they are still not fully reliable as CP modellers:

-Compilation errors

ERRORS TO LOOK FOR

LLMs are improving bits by bits, but they are still not fully reliable as CP modellers:

-Compilation errors

-Data manipulation errors

ERRORS TO LOOK FOR

LLMs are improving bits by bits, but they are still not fully reliable as CP modellers:

-Compilation errors

-Data manipulation errors

-Faulty model (missing/wrong constraints)

ERRORS TO LOOK FOR

LLMs are improving bits by bits, but they are still not fully reliable as CP modellers:

-Compilation errors

-Data manipulation errors

-Faulty model (missing/wrong constraints)

-Under-use of global constraints

ERRORS TO LOOK FOR

LLMs are improving bits by bits, but they are still not fully reliable as CP modellers:

-Compilation errors

-Data manipulation errors

-Faulty model (missing/wrong constraints)

-Under-use of global constraints

-Under-use of advanced modelling tricks

TRENDING RESEARCH TOPIC


LLMs meet Constraint Solving
CP/SAT 2025 Workshop
Monday August 11, 2025
Glasgow, Scotland

LLMs meet Constraint

CP-Bench: Evaluating Large Language Models for Constraint Modelling

Kostis Michailidis[a,*], **Dimos Tsouros**[a] **and Tias Guns**[a]

[a]DTAI, KU Leuven, Belgium

Abstract. Combinatorial problems are present in a wide range of industries. Constraint Programming (CP) is a well-suited problem-solving paradigm, but its core process, namely constraint modelling, is a bottleneck for wider adoption. Aiming to alleviate this bottleneck, recent studies have explored using Large Language Models (LLMs) as modelling assistants, transforming combinatorial problem descriptions to executable constraint models, similar to coding assis-

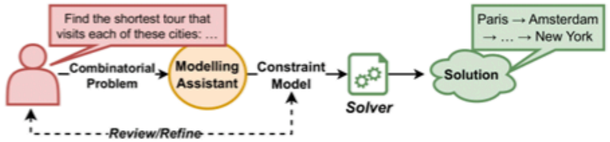


Figure 1: LLM-driven constraint modelling: users state a combinato-

LLMs meet Constraint

CP-Bench: Evaluating Large Language Models for

Abstract. Combinatorial pro
industries. Constraint Progran
solving paradigm, but its core
is a bottleneck for wider adop
neck, recent studies have exp
(LLMs) as modelling assistant
descriptions to executable cons

Constraint Modelling with LLMs Using In-Context Learning

Kostis Michailidis
DTAI, KU Leuven, Belgium

Dimos Tsouros
DTAI, KU Leuven, Belgium

Tias Guns
DTAI, KU Leuven, Belgium

Abstract

Constraint Programming (CP) allows for the modelling and solving of a wide range of combinatorial problems. However, modelling such problems using constraints over decision variables still requires significant expertise, both in conceptual thinking and syntactic use of modelling languages. In this work, we explore the potential of using pre-trained Large Language Models (LLMs) as coding

LLMs meet Constraint

CP-Bench: Evaluating Large Language Models for

g with LLMs Using In-Context

Do LLMs Understand Constraint Programming? Zero-Shot Constraint Programming Model Generation Using LLMs

Yuliang Song and Eldan Cohen

Department of Mechanical and Industrial Engineering,
University of Toronto, Toronto, Canada
yl.song@mail.utoronto.ca, ecohen@mie.utoronto.ca

Abstract. Large language models (LLMs) have gained significant attention for their ability to solve complex tasks such as coding and reasoning. In this work, we aim to evaluate their ability to generate con-

or the modelling and solving of a wide range of combinatorial
oblems using constraints over decision variables still requires
ual thinking and syntactic use of modelling languages. In
using pre-trained Large Language Models (LLMs) as coding

LLMs meet Con

CP-Benc

Do LLMs Understand Constrai
Zero-Shot Constraint Progra
Generation Using I

2023

Yuliang Song and Eldan Cohen

Department of Mechanical and Industrial Engineering,
University of Toronto, Toronto, Canada
yl.song@mail.utoronto.ca, ecohen@mie.utoronto.ca

Abstract. Large language models (LLMs) have gained significant attention for their ability to solve complex tasks such as coding and reasoning. In this work, we aim to evaluate their ability to generate con-

Holy Grail 2.0: From Natural Language to Constraint Models

Dimos Tsouros
KU Leuven, Belgium

Hélène Verhaeghe
KU Leuven, Belgium

Serdar Kadıoğlu
AI Center of Excellence, Fidelity Investments, USA
Department of Computer Science, Brown University, USA

Tias Guns
KU Leuven, Belgium

Abstract

Twenty-seven years ago, E. Freuder highlighted that "Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it". Nowadays, CP users have great modeling tools available (like Minizinc and CPMpy), allowing them to formulate the problem and then let a solver do the rest of the job, getting closer to the stated

'or the modelling and solving of a wide range of combinatorial
oblems using constraints over decision variables still requires
ual thinking and syntactic use of modelling languages. In
using pre-trained Large Language Models (LLMs) as coding

LLM

Holy Grail 2.0: From Natural Language to

MCP-Solver: Integrating Language Models with Constraint Programming Systems

Stefan Szeider

Algorithms and Complexity Group
TU Wien, Vienna, Austria
www.ac.tuwien.ac.at/people/szeider/

SA

Abstract

The MCP Solver bridges Large Language Models (LLMs) with symbolic solvers through the Model Context Protocol (MCP), an open-source standard for AI system integration. Providing LLMs access to formal solving and reasoning capabilities addresses their key deficiency while leveraging their strengths. Our implementation offers

Constraint programming represents one of the closest
ly Grail of programming: the user states the problem,
ıt modeling tools available (like Minizinc and CPMpy),
allowing them to formulate the problem and then let a solver do the rest of the job, getting closer to the stated

Do

Generation Using I

Yuliang Song and Eldan Cohen

Department of Mechanical and Industrial Engineering,
University of Toronto, Toronto, Canada
yl.song@mail.utoronto.ca, ecohen@mie.utoronto.ca

or the modelling and solving of a wide range of combinatorial
oblems using constraints over decision variables still requires
ual thinking and syntactic use of modelling languages. In
using pre-trained Large Language Models (LLMs) as coding

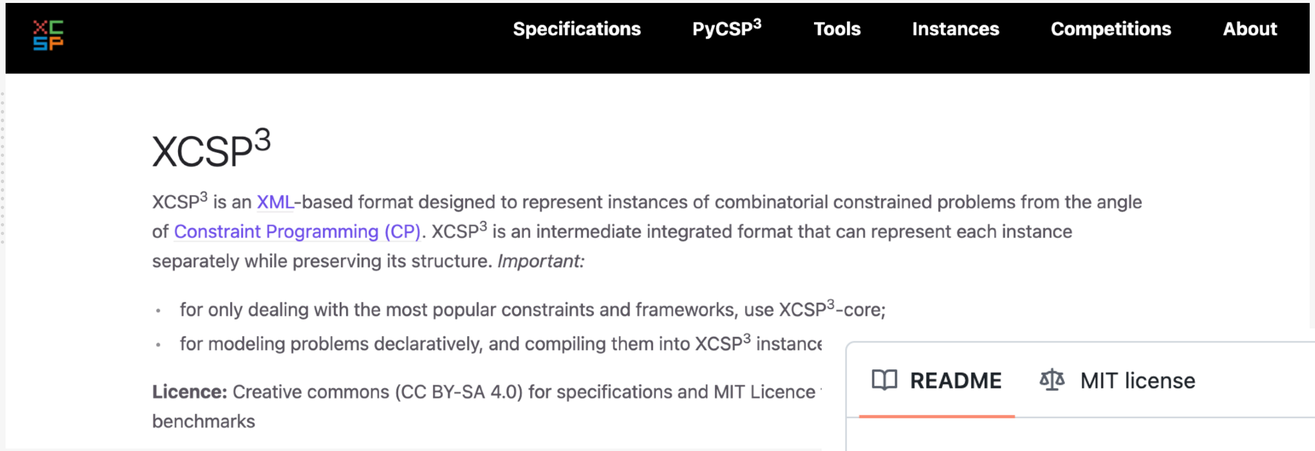
Abstract. Large language models (LLMs) have gained significant attention for their ability to solve complex tasks such as coding and reasoning. In this work, we aim to evaluate their ability to generate con-

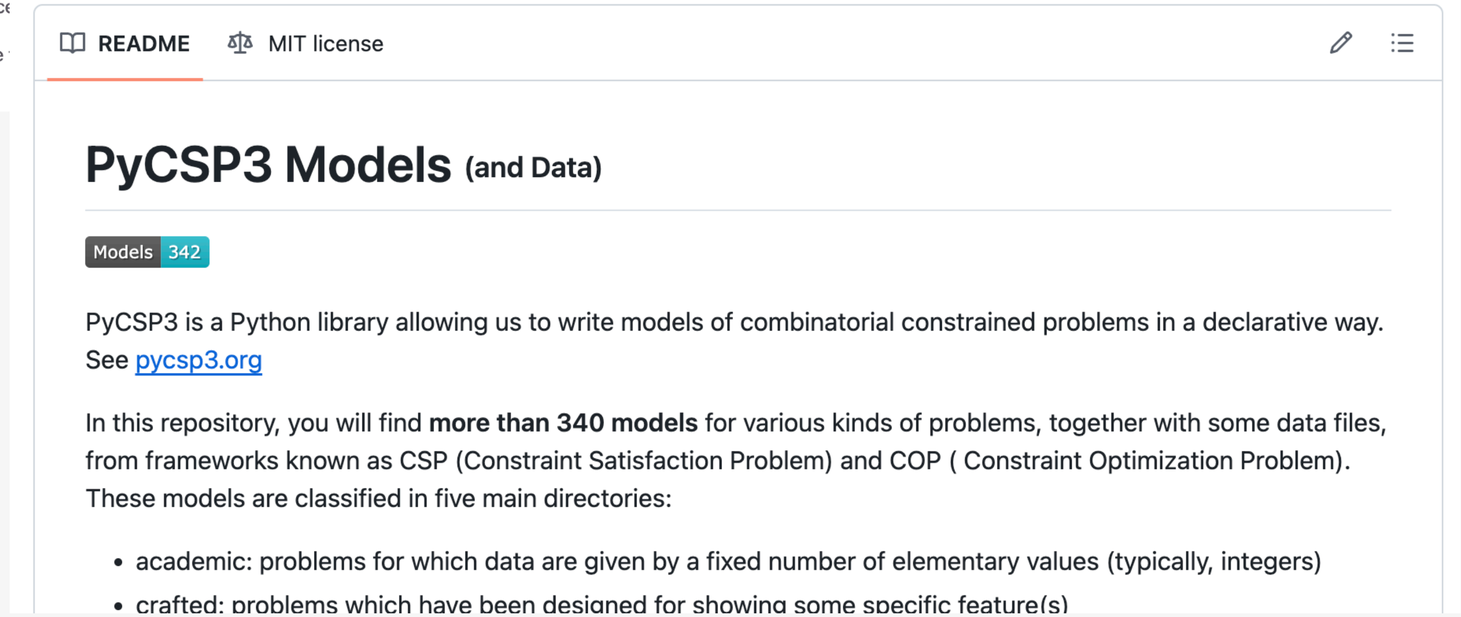
To Go Further

XCSP3 AND PYCSP3 REPOSITORY OF INSTANCES

https://xcsp.org/

https://github.com/xcsp3team/PyCSP3-models





CSPLIB REPOSITORY OF INSTANCES

https://www.csplib.org/

CSPLib

CSPLib: A problem library for constraints

Overview

CSPLib is a library of test problems for constraint solvers. The library consists of:

- Problems organised by subject area and problem number

CSPLib also maintains a list of constraint languages and the tools which can be used to solve problems in these languages.

The main motivation for CSPLib is to focus research in constraints away from purely random problems and onto more structured problems.

We currently have 96 problems in 16 categories proposed by 74 authors.

Please submit any updates or corrections to CSPLib to our git repository, either as a new issue, or pull request.

The current editors-in-chief and maintainers of CSPLib are Chris Jefferson and Özgür Akgün.

We are grateful to the previous editors of CSPLib: Ian Miguel, Brahim Hnich, Ian Gent and Toby Walsh, website maintainers including Bilal Syed Hussain, and all the contributors to CSPLib since its creation in 1999.

Updates

MINIZINC REPOSITORY OF EXAMPLES

https://github.com/MiniZinc/minizinc-examples

README MIT license

The MiniZinc Examples Archive - v1.0.0

This repository contains a collection of small MiniZinc 2.0 models demonstrating various language features. These examples are referenced by the Basic Modeling for Discrete Optimization MOOC. The versions of the examples in this repository are maintained to the latest version of MiniZinc and may differ from reference documents.

CPMPY REPOSITORY OF EXAMPLES

https://github.com/CPMpy/cpmpy/tree/master/examples

README.md

CPMpy examples

Subdirectories:

- `advanced/` : examples involving machine learning, explanations and incremental solving.
- `csplib/` : runnable CSPlib examples
- `tutorial*/` : slides and runnable notebooks used in CPMpy tutorials

Notebook examples:

You can run these in Google Colab, on Binder or view them on GitHub.

HÅKAN KJELLERSTRAND'S REPOSITORY OF INSTANCES

https://www.hakank.org/

Welcome to hakank's Home Page

***Home of** https://www.hakank.org, **Håkan Kjellerstrand** (hakank@gmail.com)*

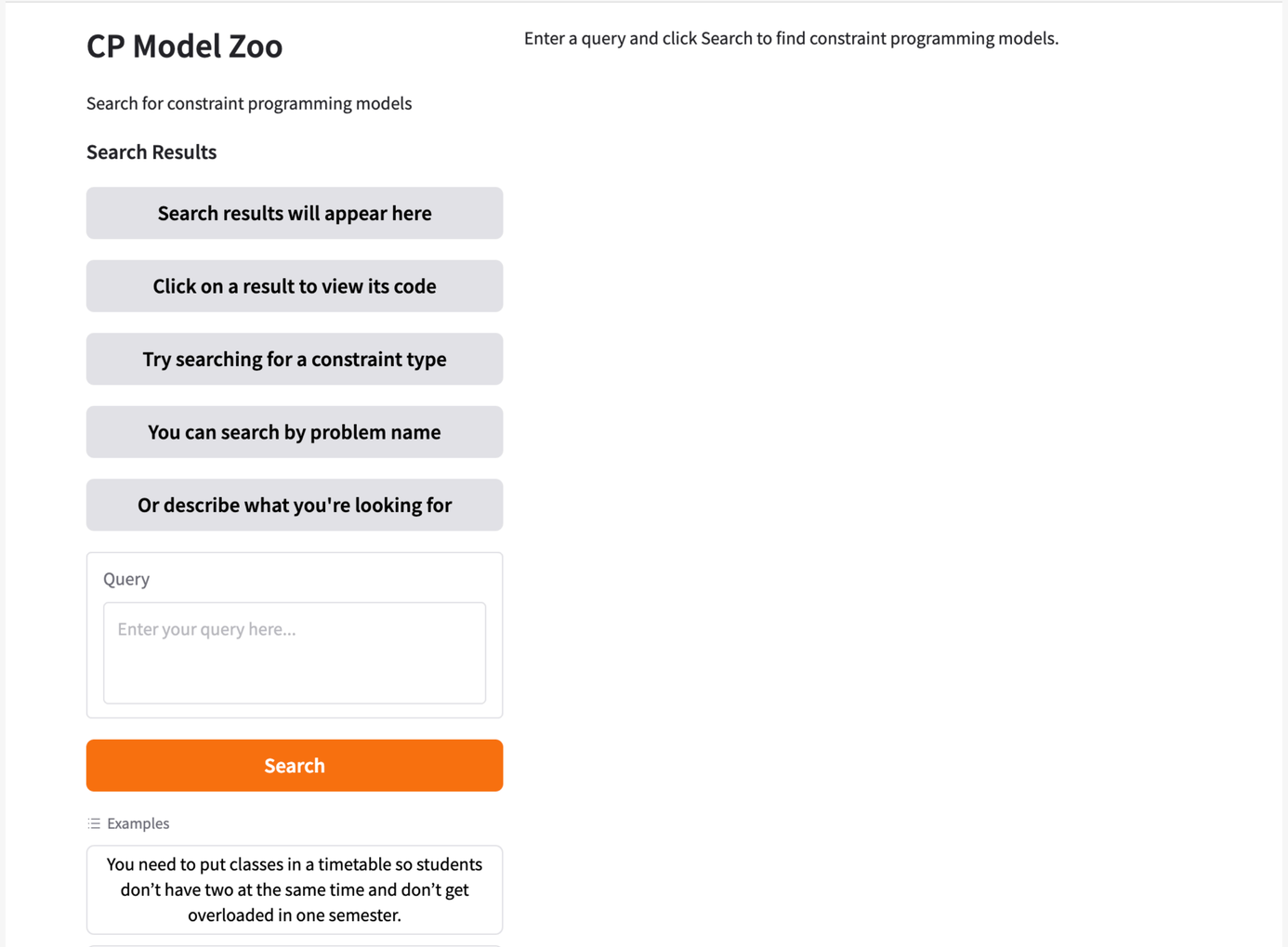
Svensk version ***av denna sida.***

hakank.blogg: my webblogg. Mostly in Swedish, but there are quite a few links which may be interesting even if you don't understand the text.
My Constraint Programming Blog: My blog about Constraint Programming and related paradigms.
Arrays in Flux: Blog about programming/programming languages, mathematics, machine learning/data mining, AI, etc.

Picture.
My Linkedin page
My Twitter page.
My GitHub page
My GitHub repo: https://github.com/hakank/hakank/
My Rosetta Code page
My Google Scholar page
My DBLP page
My ResearchGate page

CP-MODEL-ZOO MODEL RETRIEVAL TOOL

https://cp-model-zoo.info.ucl.ac.be/



Conclusion



RECIPE TO MODEL IN CP



RECIPE TO MODEL IN CP

- Step 1: Describe the solution of your problem, what elements, what properties it should have to be considered a solution



RECIPE TO MODEL IN CP

- Step 1: Describe the solution of your problem, what elements, what properties it should have to be considered a solution
- Step 2: Identify the data that will be used or are needed to encode the problem: parameters, mappings, ranges,…



RECIPE TO MODEL IN CP

- Step 1: Describe the solution of your problem, what elements, what properties it should have to be considered a solution
- Step 2: Identify the data that will be used or are needed to encode the problem: parameters, mappings, ranges,…
- Step 3: Identify the variables of the problem and their domains



RECIPE TO MODEL IN CP

- Step 1: Describe the solution of your problem, what elements, what properties it should have to be considered a solution
- Step 2: Identify the data that will be used or are needed to encode the problem: parameters, mappings, ranges,…
- Step 3: Identify the variables of the problem and their domains
- Step 4: Identify the constraints of the problem, use globals as much as possible

RECIPE TO MODEL IN CP

- Step 1: Describe the solution of your problem, what elements, what properties it should have to be considered a solution
- Step 2: Identify the data that will be used or are needed to encode the problem: parameters, mappings, ranges,…
- Step 3: Identify the variables of the problem and their domains
- Step 4: Identify the constraints of the problem, use globals as much as possible
- Step 5: Identify potential other formulations, symmetries of the problems, or redundant constraints that might improve the strength of the model (best to test solving metrics while doing that)

TO TAKE HOME

TO TAKE HOME

- There can be multiple models for the same problem

TO TAKE HOME

- There can be multiple models for the same problem
- Model can impact the efficiency of the solving process (number of variables, constraints used, symmetry breaking, redundant constraints,…)

TO TAKE HOME

- There can be multiple models for the same problem
- Model can impact the efficiency of the solving process (number of variables, constraints used, symmetry breaking, redundant constraints,…)
- Modelling is an art, for which practice and use make you better at it!