Special Lecture on Computer Science III - Final Report

GILLARD Antonino Yann William

January 2026

Contents

1 Introduction

The N-queens problem is a famous optimization problem. Let $N \in \mathbb{N}$. We consider a chess board of size $N \times N$. We want to place N queen pieces on the board such that they can not attack one another. In chess, a queen can attack pieces who are on the same row, column or diagonal as itself.

In this class, we have already seen methods on how to tackle that problem. In this report, we will solve that problem with two different models :

- a constraint programming model using only boolean variables and pseudo boolean constraints,
- a QUBO model.

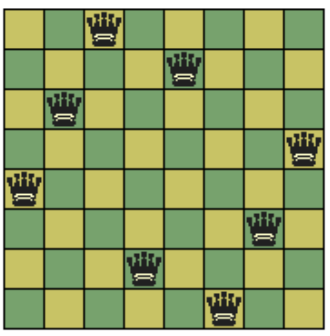

Figure 1: A solution to the N-queens problem for $N = 8$

In order to compare the performance of these two models we will use the MiniZinc Model given in Lecture 3 as a baseline. Instead of using boolean variable, it used N integer variables of value in $[\![1, N]\!]$.

2 Model 1 - Using boolean variables

2.1 Modeling

In this section, we model our board with N^2 boolean variables such that :

$$\forall i \in [\![1, N]\!] \quad \forall j \in [\![1, N]\!] \quad x_{i,j} = \begin{cases} 1, & \text{if a queen is present on tile } (i, j), \\ 0, & \text{otherwise.} \end{cases}$$

Now that we have our board, we can use the rules of chess to define our constraints. Queens can attack pieces on the same row and on the same column, meaning that we can only have 1 queen per row and one queen per column. As such, we have the follwing constraints :

$$\forall i \in [\![1, N]\!] \quad \sum_{j=1}^{N} x_{i,j} = 1, \quad \text{(column constraints)}$$

$$\forall j \in [\![1, N]\!] \quad \sum_{i=1}^{N} x_{i,j} = 1, \quad \text{(row constraints)}$$

These 2 constraints also include the constraint that there must be N queens placed on the board. Finally, we add the constraints on the diagonals. There must be at most a queen per diagonal.

$$\forall d \in [\![0, N-1]\!] \quad \sum_{k=0}^{N-d-1} x_{1+k,1+k+d} \leq 1,$$

$$\forall d \in [\![1, N-1]\!] \quad \sum_{k=0}^{N-d-1} x_{1+k+d,1+k} \leq 1,$$

(top left to bottom right diagonals)

$$\forall d \in [\![0, N-1]\!] \quad \sum_{k=0}^{N-d-1} x_{N-d-k,1+k} \leq 1,$$

$$\forall d \in [\![1, N-1]\!] \quad \sum_{k=0}^{N-d-1} x_{N-k,1+d+k} \leq 1,$$

(bottom left to top right diagonals)

2.2 Implementation

Translating these constraints into MiniZinc gives us :

```
% Size of board
int: N = 10;

% Board variables
array[1..N, 1..N] of var 0..1: positions;

% 1 queen per row / column
constraint forall (i in 1..N)
    (sum (j in 1..N) (positions[i, j]) == 1);
constraint forall (j in 1..N)
    (sum (i in 1..N) (positions[i, j]) == 1);

% Diagonal constraints
constraint forall (d in 0..(N-1))
    (sum (k in 0..(N-d-1)) ( positions[1+k, 1+k+d]  ) <= 1);
constraint forall (d in 1..(N-1))
    (sum (k in 0..(N-d-1)) ( positions[1+k+d, 1+k]  ) <= 1);
constraint forall (d in 0..(N-1))
    (sum (k in 0..(N-d-1)) ( positions[N-d-k, 1+k]  ) <= 1);
constraint forall (d in 1..(N-1))
    (sum (k in 0..(N-d-1)) ( positions[N-k, 1+k+d]  ) <= 1);

solve satisfy;

output [show2d(positions)];
```

2.3 Results

To properly evaluate our program, we remove the command which prints the solution. By performing subsequent execution, we have the following execution times :

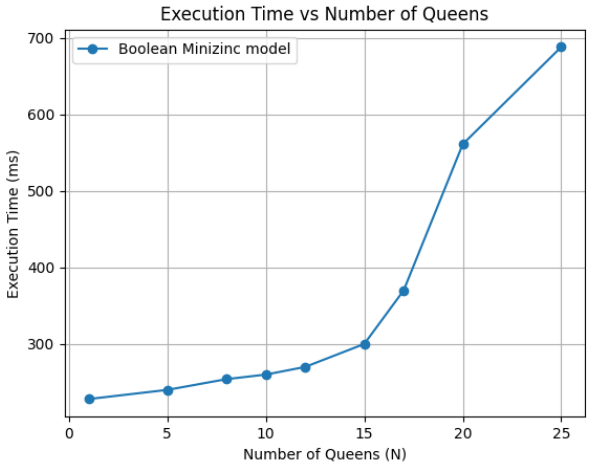

Figure 2: Execution time of the boolean variable minizinc model for different values of N

We could have expected, an exponential growth of the execution times as N grows larger. Although, it is not obvious in this graph, the execution time for $N = 26$ takes way longer than the

one for $N = 25$ (more than 10s). It can also be interesting to note that the times for $N = 1, 5and10$ are similar (approx. 300ms). We expect from a well defined model and a good solver that the case $N = 1$ be over as fast as possible. In the case $N = 1$, compile time is almost all of the execution time. The compilation reduces the program down to :

```
solve  satisfy;
```

Thus, we can observe that solving the N-queens problem with this model in MiniZinc for $N \leq 10$ gives us times similar to executing a trivial problem. The great increase in execution time for greater values of N may be caused by the complexity of the problem who grows exponentially but also perhaps because we have N^2 variables. As the number of constraints for each variable increases with greater values of N, having a model with N^2 may make things worse for the execution time.

3 Model 2 - QUBO

3.1 Formal definition

Since we are already working with boolean variables, we keep the earlier model for our QUBO model. We now need to translate the constraints into an objective loss to minimize.
The following constraints

$$\forall i \in [\![1, N]\!] \quad \sum_{j=1}^{N} x_{i,j} = 1, \quad \forall j \in [\![1, N]\!] \quad \sum_{i=1}^{N} x_{i,j} = 1,$$

translate to adding

$$-\sum_{i=1}^{N} \left(\sum_{j=1}^{N} x_{i,j} + 2 \sum_{k<l}^{N} x_{i,k} x_{i,l} \right) \text{ and } -\sum_{j=1}^{N} \left(\sum_{i=1}^{N} x_{i,j} + 2 \sum_{k<l}^{N} x_{k,j} x_{l,j} \right),$$

to the loss function. The diagaonal constraints

$$\forall d \in [\![0, N-1]\!] \quad \sum_{k=0}^{N-d-1} x_{1+k,1+k+d} \leq 1, \quad \forall d \in [\![1, N-1]\!] \quad \sum_{k=0}^{N-d-1} x_{1+k+d,1+k} \leq 1,$$
$$\forall d \in [\![0, N-1]\!] \quad \sum_{k=0}^{N-d-1} x_{N-d-k,1+k} \leq 1, \quad \forall d \in [\![1, N-1]\!] \quad \sum_{k=0}^{N-d-1} x_{N-k,1+d+k} \leq 1,$$

translate to

$$\sum_{d=0}^{N-1} \left(\sum_{0 \leq k < l < N-d-1} x_{1+k+d,1+k} \quad x_{1+l+d,1+l} \right), \quad \sum_{d=1}^{N-1} \left(\sum_{0 \leq k < l < N-d-1} x_{1+k,1+k+d} \quad x_{1+k,1+l+d} \right),$$
$$\sum_{d=0}^{N-1} \left(\sum_{0 \leq k < l < N-d-1} x_{N-d-k,1+k} \quad x_{N-d-l,1+l} \right), \quad \sum_{d=1}^{N-1} \left(\sum_{0 \leq k < l < N-d-1} x_{N-k,1+d+k} \quad x_{N-l,1+d+l} \right)$$

Thus, the final loss function is

$$\begin{aligned} L = &- \sum_{i=1}^{N} \left(\sum_{j=1}^{N} x_{i,j} + 2 \sum_{k<l}^{N} x_{i,k} x_{i,l} \right) - \sum_{j=1}^{N} \left(\sum_{i=1}^{N} x_{i,j} + 2 \sum_{k<l}^{N} x_{k,j} x_{l,j} \right) \\ &+ \sum_{d=0}^{N-1} \left(\sum_{0 \leq k<l<N-d-1} x_{1+k+d,1+k} \quad x_{1+l+d,1+l} \right) + \sum_{d=1}^{N-1} \left(\sum_{0 \leq k<l<N-d-1} x_{1+k,1+k+d} \quad x_{1+k,1+l+d} \right) \\ &+ \sum_{d=0}^{N-1} \left(\sum_{0 \leq k<l<N-d-1} x_{N-d-k,1+k} \quad x_{N-d-l,1+l} \right) + \sum_{d=1}^{N-1} \left(\sum_{0 \leq k<l<N-d-1} x_{N-k,1+d+k} \quad x_{N-l,1+d+l} \right). \end{aligned}$$

In order to properly define our problem as a QUBO problem, we need to put the loss function in the following form

$$L = X^T Q X,$$

where $M \in \mathbb{N}$, $X \in \{0,1\}^M$, $Q \in \mathcal{M}_{M,M}(\{0,1\})$.
As such, let us define $\varphi : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ such that

$$\forall i \in [\![1, N]\!] \quad \forall j \in [\![1, N]\!] \quad \varphi(i,j) = N(i-1) + j,$$

and let $X = (X_k)_{k \in [\![1, N^2]\!]}$ such that

$$\forall i \in [\![1, N]\!] \quad \forall j \in [\![1, N]\!] \quad X_{\varphi(i,j)} = x_{i,j}.$$

3.2 Implementation

We will now use the Fixstars Amplify API and Python to solve the N-queens problem with the above QUBO model. Thankfully, the Fixstars Amplify API auto generates the penalty function from the constraints we have defined earlier. As such, we do not need to properly define the matrix Q.
Here is our Python code :

```
# Environment variable dependency
import os
from dotenv import load_dotenv

# Amplify dependencies
from amplify import VariableGenerator, AmplifyAEClient, ConstraintList
from amplify import equal_to, less_equal, solve

# Import environment variables
load_dotenv()
token = os.getenv("FIXSTARS_TOKEN")

# Global variables declaration
N = 10

# Board initialization
gen = VariableGenerator()
board = gen.array("Binary", shape=(N, N))
```

```
# Constraints initialization
constraints = ConstraintList()

# Row constraints
for i in range(0,N):
    row_sum = 0
    for j in range(0,N):
        row_sum += board[i,j]
    constraints += equal_to(row_sum, 1)

# Column constraints
for j in range(0,N):
    col_sum = 0
    for i in range(0,N):
        col_sum += board[i,j]
    constraints += equal_to(col_sum, 1)

# Diagonal constraints
for d in range(0, N):
    diag_sum = 0
    for k in range(0, N-d):
        diag_sum += board[k,k+d]
    constraints += less_equal(diag_sum, 1)

for d in range(1, N):
    diag_sum = 0
    for k in range(0, N-d):
        diag_sum += board[k+d,k]
    constraints += less_equal(diag_sum, 1)

for d in range(0, N):
    diag_sum = 0
    for k in range(0, N-d):
        diag_sum += board[N-d-k-1, k]
    constraints += less_equal(diag_sum, 1)

for d in range(1, N):
    diag_sum = 0
    for k in range(0, N-d):
        diag_sum += board[N-k-1,k+d]
    constraints += less_equal(diag_sum, 1)

model = constraints

# Initializing and calling the client solver
client = AmplifyAEClient()
client.token = token
client.parameters.time_limit_ms = 1000    # Set run time to 10000 ms

print("Calling solver ...")
```

```
result = solve(model, client)
best = result.best

print(f"Execution time: {result.execution_time} seconds")
print(f"Best solution :\n{board.evaluate(result.best.values)}")
```

3.3 Results

By querying multiple requests to the server for different values of N, we get the following execution times :

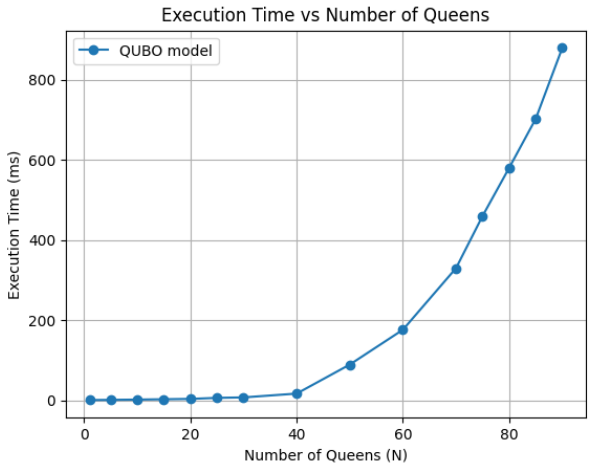

Figure 3: Execution time of the QUBO model for different values of N

We can also observe an exponential growth as N grows larger. Here, it is not computational power which does not allow us to look at cases where $N \geq 91$, but it is the amount of variable which blocks us. Indeed, Fixstars API is a service with different subscription tiers. For $N > 90$, the number of variables exceeds the maximum job size of 8192. Performance wise, our model is similar to the 1-queen problem for $N \leq 40$. These values are also quite low, being expressed in microseconds unlike the previous model.

4 Comparing different model

4.1 Reminder - Baseline model

Unlike our previous model, here we consider N different integer variables of value ranging from 1 to N. Let us consider for $i \in [\![1, N]\!]$, $Q_i \in [\![1, N]\!]$ be the row the queen in column k is placed. This problem formulation already fits the row constraint we developed previously. To translate the column constraint, here all variable must be different from one another, or

$$\forall i \in [\![1, N]\!] \quad \forall j \in [\![1, N]\!] \quad i \neq j \implies Q_i \neq Q_j.$$

As for the diagonal constraints, they become :

$$\forall i \in [\![1, N]\!] \quad \forall j \in [\![1, N]\!] \quad i \neq j \implies Q_i + i \neq Q_j + j$$
$$\forall i \in [\![1, N]\!] \quad \forall j \in [\![1, N]\!] \quad i \neq j \implies Q_i - j \neq Q_j - j$$

4.2 Implementation of the baseline model

The lecture slide uses this MiniZinc example, available from the MiniZinc documentation.

```
int: n;
array [1..n] of var 1..n: q; % queen in column i is in row q[i]

include "alldifferent.mzn";

constraint alldifferent(q);                          % distinct rows
constraint alldifferent([ q[i] + i | i in 1..n]); % distinct diagonals
constraint alldifferent([ q[i] - i | i in 1..n]); % upwards+downwards

% search
solve :: int_search(q, first_fail, indomain_min)
      satisfy;
output [ if fix(q[j]) == i then "Q" else "." endif ++
         if j == n then "\n" else "" endif | i,j in 1..n]
```

With a few iteration of this implementation, we get the following execution times :

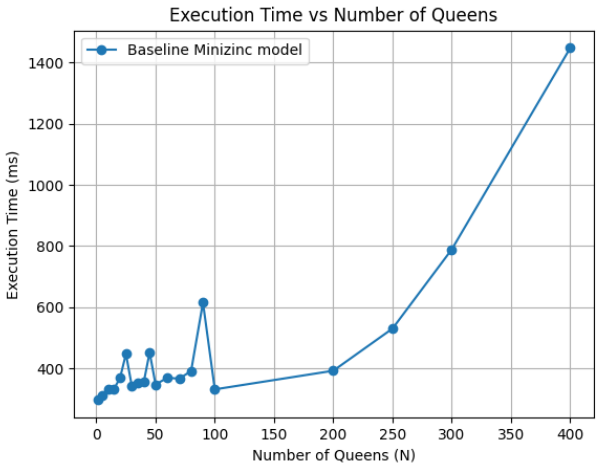

Figure 4: Execution time of the baseline model for different values of N

We can observe the same overall figure of an exponential increase as N gets larger. However, here, for $N \leq 100$, we can observe fluctuations. Certains cases are more complex than others. This can be attributed to the solver not taking the best path first and thus being longer for some cases more than others. This model can go up to $N = 400$ with only 1.4s of execution time.
Since all models follow an exponential growth, we will use log scale to compare models with one another.

4.3 MiniZinc models

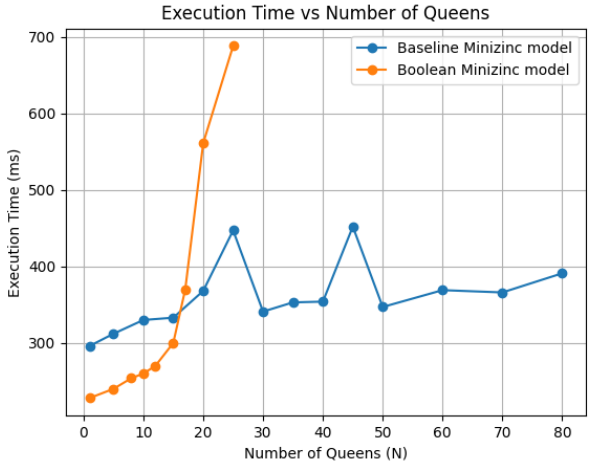

Figure 5: Comparing execution times for both MiniZinc models

The boolean variable MiniZinc model performs way worse than the integer variable MiniZinc model. Indeed, it is hard to have results for $N \geq 25$ for the boolean variable model due to the long execution times whereas the interger variable model computes for up to $N = 400$.

Difference between both models lie in the choice of variables. The choice to use integer variables already implements the row constraint. The column constraint becomes a simple difference check between all variables instead of computing a heavy sum of $\mathcal{O}(N)$ for each variables and thus a total cost of $\mathcal{O}(N^2)$. Furthermore instead of using N^2 varaiables, we only need to use N varaiables.

This comparaison shows the importance of well defining one's model for optimization problems by trying to imply as much constraints as possible in the definition of the variables.

4.4 Comparing the QUBO model with the integer variable MiniZinc model

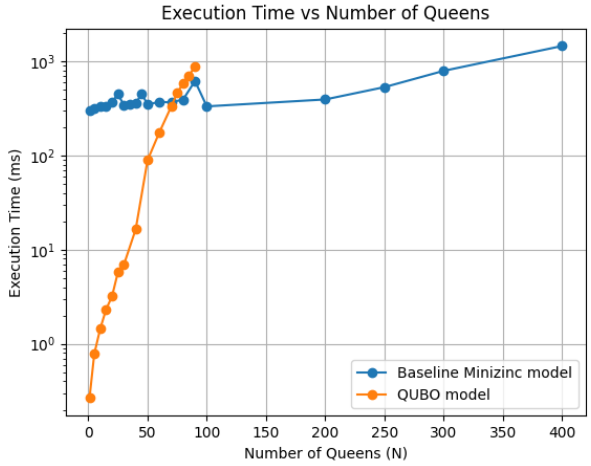

Figure 6: Execution time of the baseline model for different values of N

As for the QUBO model, although we cannot have values for the greater values of N like the MiniZinc baseline model due to the limitations of the Fixstars API, we can already observe that the values for $N = 90$ are similar to those for $N = 300$ of the MiniZinc baseline model.

An interesting aspect of the QUBO model though, is that for the cases where N is small, it outperforms the MiniZinc model. This is probably because the servers owned by Fixstars are

tuned to perform these kind of computations instead of running the MiniZinc models on one's own computer.

By applying what we saw from the analysis of the MiniZinc models, one might wonder if adapting the integer variable model to QUBO might improve performance, perhaps by using Unary encoding of the integer variables. Although it would decrease the number of variables by a small margin with N integer variables encoded each by $N-1$ boolean varaiables thus $N\,(N-1)$ boolean variables in total, the complexity of the constraints would not get better but would get worse since we also have to add constraints to keep the encoding of the integer variables.

5 Conclusion

In this report, we have introduced 2 boolean variable models to solve the N-queens problem : a MiniZinc model and a QUBO model. We used an integer variable MiniZinc model as a baseline model to compare execution time. Concerning the MiniZinc models, we can see the importance of a well defined model considering that one can compute solutions up to $N = 400$ whereas the other one struggles at $N \geq 25$. From these 2 models, we can also see the importance of how the number of variables and constraints can hinder a model meant to solve a constraint optimization problem. The baseline model shows that defining the variables such that we simplify the constraints and even imply some just by the definition of the variables can greatly improve the performance of a model. As for the QUBO model, we can see that QUBO is way more efficient that the boolean varaible model in MiniZinc, as it is able to go up to $N = 90$ at least. But it also shows the weakness of QUBO as even though it is faster for such computation, we do not have the same freedom in defining our variables and thus simplifying our model for better efficiency. As such, the integer variable MiniZinc model still outperforms our QUBO model.

6 Sources

You can find the source code used for this report in the following GitHub repository.