LAAI Module 1: Project Report
Nims Game in Prolog and Minizinc

Memoona Shah (0001056334),
Rooshan Saleem Butt (0001026467)

September 5, 2022

Contents

0.1 Introduction

In this project we chose to write codes in prolog as well as minizinc to design a solver for an interesting game of Nims. For programming the game Prolog programming language was used. We had to make use of different types of methods that Prolog provides such as declarative Prolog for querying facts and rules. In this program we got to use lists, recursive rules, etc. In the part one, with prolog a game is played between a computer and a human and a winner is declared in the end. In the second part, the same game is played in minizinc where all possible moves are given by the optimization code.

0.2 The Game of Nims and its rules

Nim is a mathematical game of strategy in which two players take turns removing (or "nimming") objects from distinct heaps or piles. On each turn, a player must remove at least one object, and may remove any number of objects provided they all come from the same heap or pile. Depending on the version being played, the goal of the game is either to avoid taking the last object or to take the last object.

The game has only 3 rules.
- start with 12 sticks in the pile
- each player takes 1, 2, or 3 sticks in turn
- the player who takes the last stick wins.

0.3 Prolog Part

0.3.1 Goal

A user starts the game by entering a command nims. The goal initiates and starts the game with next predicate. nim:-next(numberofSticks).

0.3.2 Next Predicate

The next predicate first asks the user to enter a number that represents the number of sticks that the player wants to pick. At the same time, also prints the instruction that the number of sticks can not be more than three and not less than 0. Then it checks if the number entered by the user is indeed between 1 and 3. This is done by member predicate of prolog. It then calculates the remaining number of sticks in the pile and outputs the status of the game.

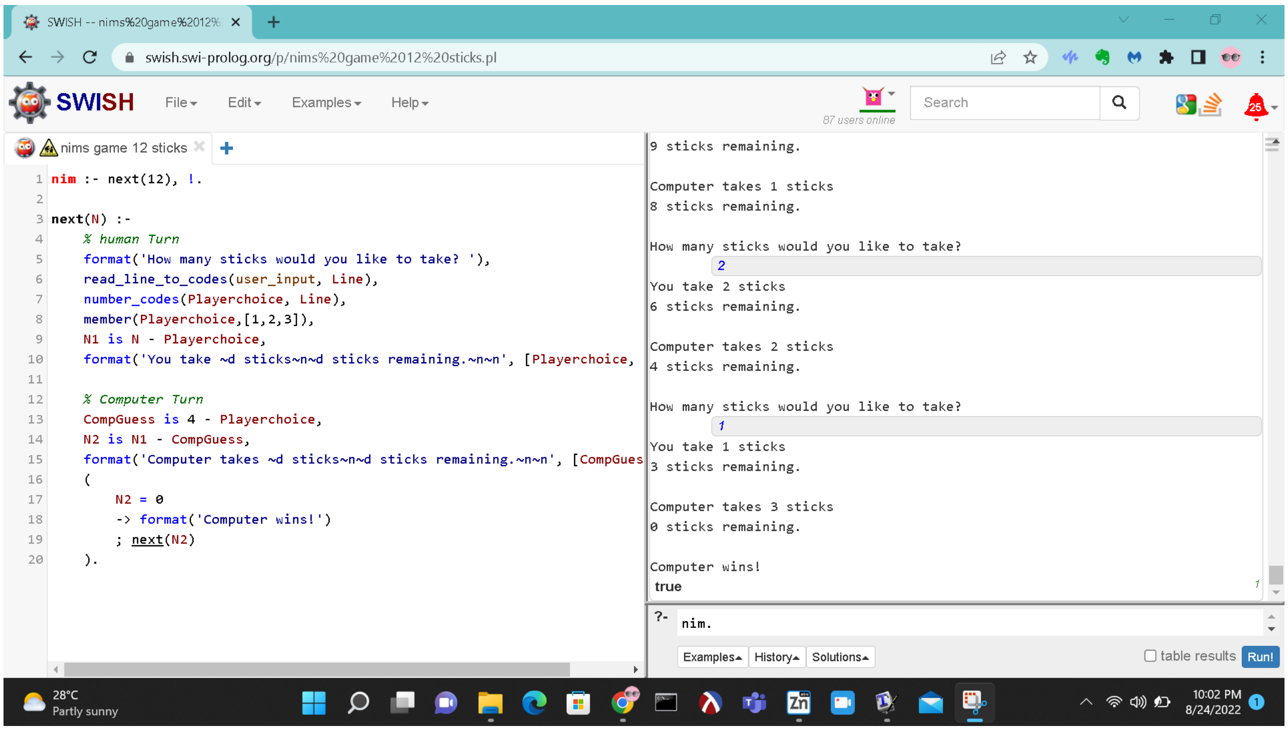
After that, the next predicate is designed to choose a number for the computer player that will be four minus the number of sticks the human player had chosen. This is the ideal trick with ensures that the computer always wins. subsequently, again the number of sticks remaining in the pile are calculates by simple subtraction operation and if the the number of sticks in the current pile are zero, then the computer wins. If not then next predicate is called again recursively with new number of sticks in the pile as its argument.

0.3.3 Code

```
nim :- next(12), !.

next(N) :-
% human Turn
format('How many sticks would you like to take? '),
read_line_to_codes(user_input, Line),
number_codes(Playerchoice, Line),
member(Playerchoice,[1,2,3]),
N1 is N - Playerchoice,
format('You take ~d sticks~n~d sticks remaining.~n~n', [Playerchoice, N1]),

% Computer Turn
```

```
CompGuess is 4 - Playerchoice,
N2 is N1 - CompGuess,
format('Computer takes ~d sticks~n~d sticks remaining.~n~n', [CompGuess, N2]),
(
N2 = 0
-> format('Computer wins!')
; next(N2)
).
```

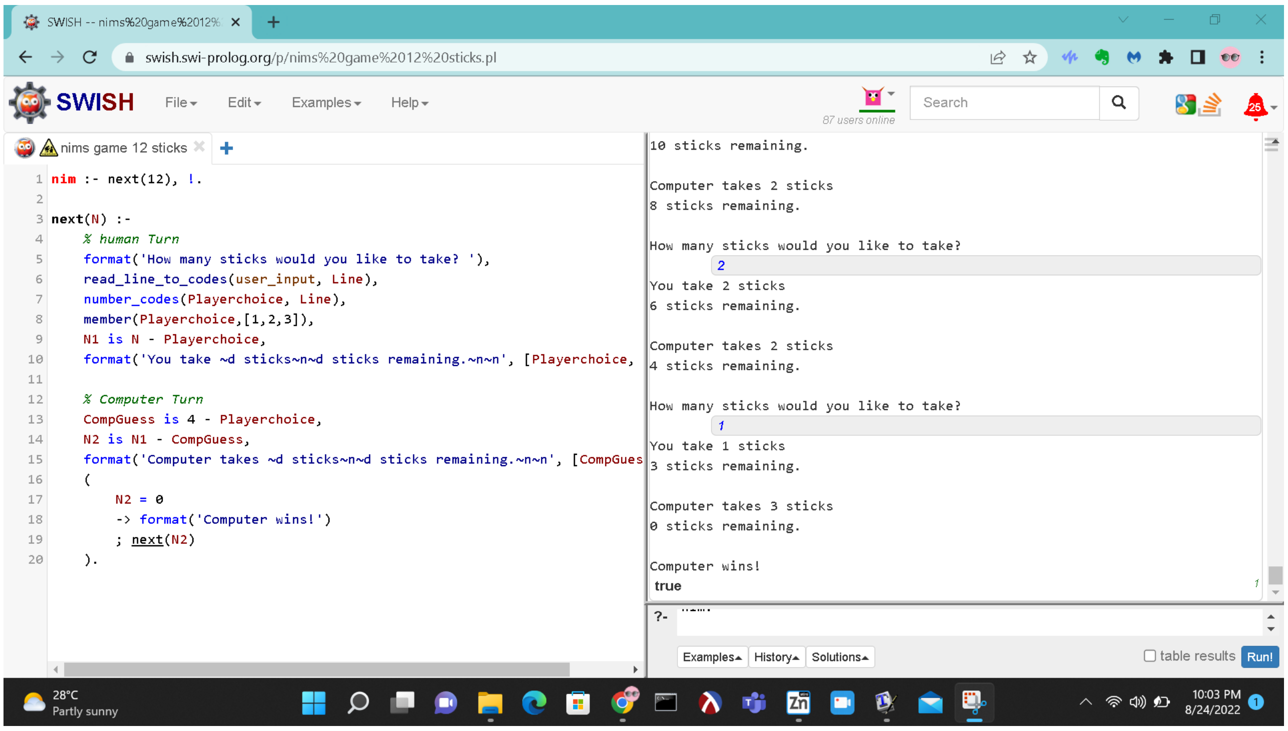
0.3.4 Results



```
nim :- next(12), !.

next(N) :-
    % human Turn
    format('How many sticks would you like to take? '),
    read_line_to_codes(user_input, Line),
    number_codes(Playerchoice, Line),
    member(Playerchoice,[1,2,3]),
    N1 is N - Playerchoice,
    format('You take ~d sticks~n~d sticks remaining.~n~n', [Playerchoice,

    % Computer Turn
    CompGuess is 4 - Playerchoice,
    N2 is N1 - CompGuess,
    format('Computer takes ~d sticks~n~d sticks remaining.~n~n', [CompGues
    (
        N2 = 0
        -> format('Computer wins!')
        ; next(N2)
    ).
```

```
9 sticks remaining.

Computer takes 1 sticks
8 sticks remaining.

How many sticks would you like to take?
        2
You take 2 sticks
6 sticks remaining.

Computer takes 2 sticks
4 sticks remaining.

How many sticks would you like to take?
        1
You take 1 sticks
3 sticks remaining.

Computer takes 3 sticks
0 sticks remaining.

Computer wins!
true

?- nim.
```

Figure 1: *NIMS Game* .



```
nim :- next(12), !.

next(N) :-
    % human Turn
    format('How many sticks would you like to take? '),
    read_line_to_codes(user_input, Line),
    number_codes(Playerchoice, Line),
    member(Playerchoice,[1,2,3]),
    N1 is N - Playerchoice,
    format('You take ~d sticks~n~d sticks remaining.~n~n', [Playerchoice,

    % Computer Turn
    CompGuess is 4 - Playerchoice,
    N2 is N1 - CompGuess,
    format('Computer takes ~d sticks~n~d sticks remaining.~n~n', [CompGues
    (
        N2 = 0
        -> format('Computer wins!')
        ; next(N2)
    ).
```

```
10 sticks remaining.

Computer takes 2 sticks
8 sticks remaining.

How many sticks would you like to take?
       2
You take 2 sticks
6 sticks remaining.

Computer takes 2 sticks
4 sticks remaining.

How many sticks would you like to take?
       1
You take 1 sticks
3 sticks remaining.

Computer takes 3 sticks
0 sticks remaining.

Computer wins!
true
```

Figure 2: *NIMS Game* .

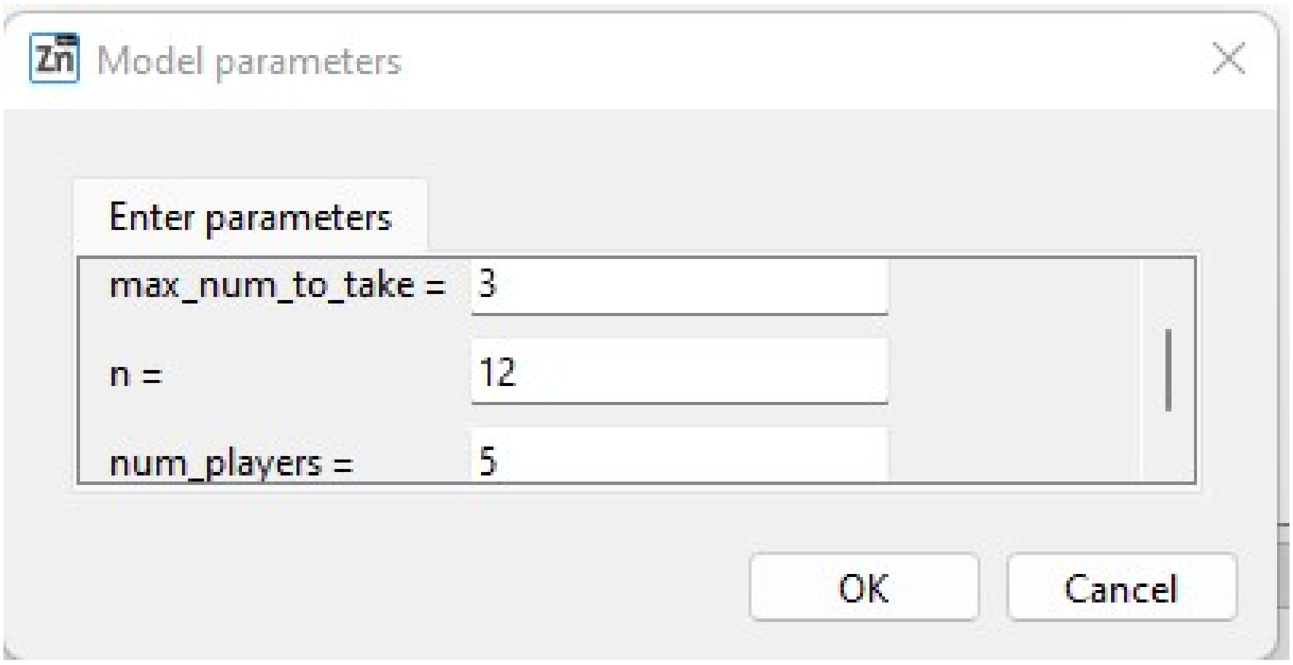
0.3.5 Conclusion

In this part of the project, we learned to use recursion in prolog and use of lists to be able to create a basic two player game. We also took input from the user which was the human player. The goal was to create a game that the computer always wins. and to do that we designed our code as such that no matter what number of sticks the user chooses, the computer always outsmarts the user by using a mathematical trick.

0.4 MINIZINC part

0.4.1 Description

In the minizinc part, the programme asks the user to enter the model paremeters, such as: -
1. Number of match sticks (n)
2. Max number of match sticks that a user can take in a turn (*max_num_to_take*)
3. Number of Players (*num_player*)



max_num_to_take = 3
n = 12
num_players = 5

Then it calculates possible number of solution satisfying the constrains.

0.4.2 Decision Variables

Decision variables are variables in the sense of mathematical or logical variables. Unlike parameters and variables in a standard programming language, the modeller does not need to give them a value. Rather the value of a decision variable is unknown and it is only when the MiniZinc model is executed that the solving system determines if the decision variable can be assigned a value that satisfies the constraints in the model and if so what this is. Here we have used 'x' as decision variable. It is the number of match sticks taken in a move by a particular player:-

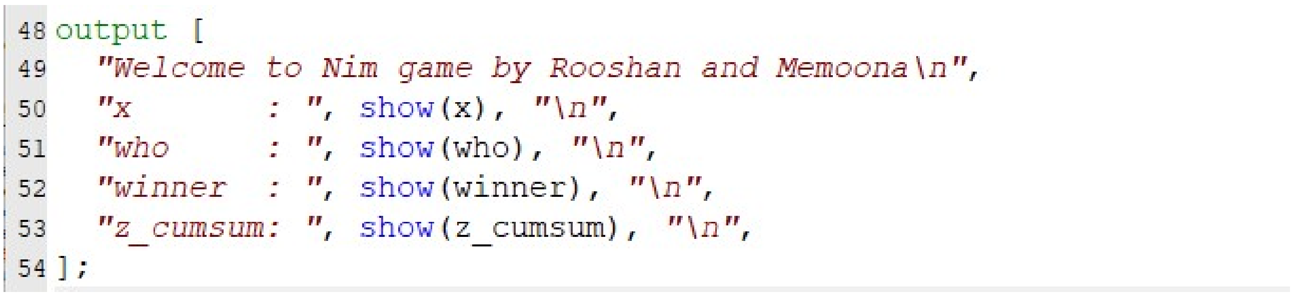
```
array[1..n]  of  var  0..max_num_to_take: x;
```

0.4.3 Constraints

The next component of the model are the constraints. These specify the Boolean expressions that the decision variables must satisfy to be a valid solution to the model. Here following are the constrains: -
1. No player can skip it's turn, it has to take a match sticks within the given range as defined at the beginning of the game.
2. Each player will have it's turn in an orderly manner i.e. turns are not random.
3. The z_cumsum must be equal to the number of match sticks already taken by the players in the game.
4. Only the person taking the last match stick will be the winner.

0.4.4 Output

Output is displayed in the following sequence:-



```
output [
  "Welcome to Nim game by Rooshan and Memoona\n",
  "x       : ", show(x), "\n",
  "who     : ", show(who), "\n",
  "winner  : ", show(winner), "\n",
  "z_cumsum: ", show(z_cumsum), "\n",
];
```

0.4.5 Search

By default in MiniZinc, there is no declaration of how we want to search for solutions. This leaves the search completely up to the underlying solver. But sometimes, we may want to specify how the search should be undertaken. This requires us to communicate to the solver a search strategy. Note that the search strategy is not really part of the model. Indeed it is not required that each solver implements all possible search strategies. MiniZinc uses a consistent approach to communicating extra information to the constraint solver using annotations. Following is our search strategy, using annotation, to get the solutions:-

```
solve :: int_search(x, first_fail, indomain_min)
maximize sum(i in 1..n)
(bool2int(winner[i] = 1));
```

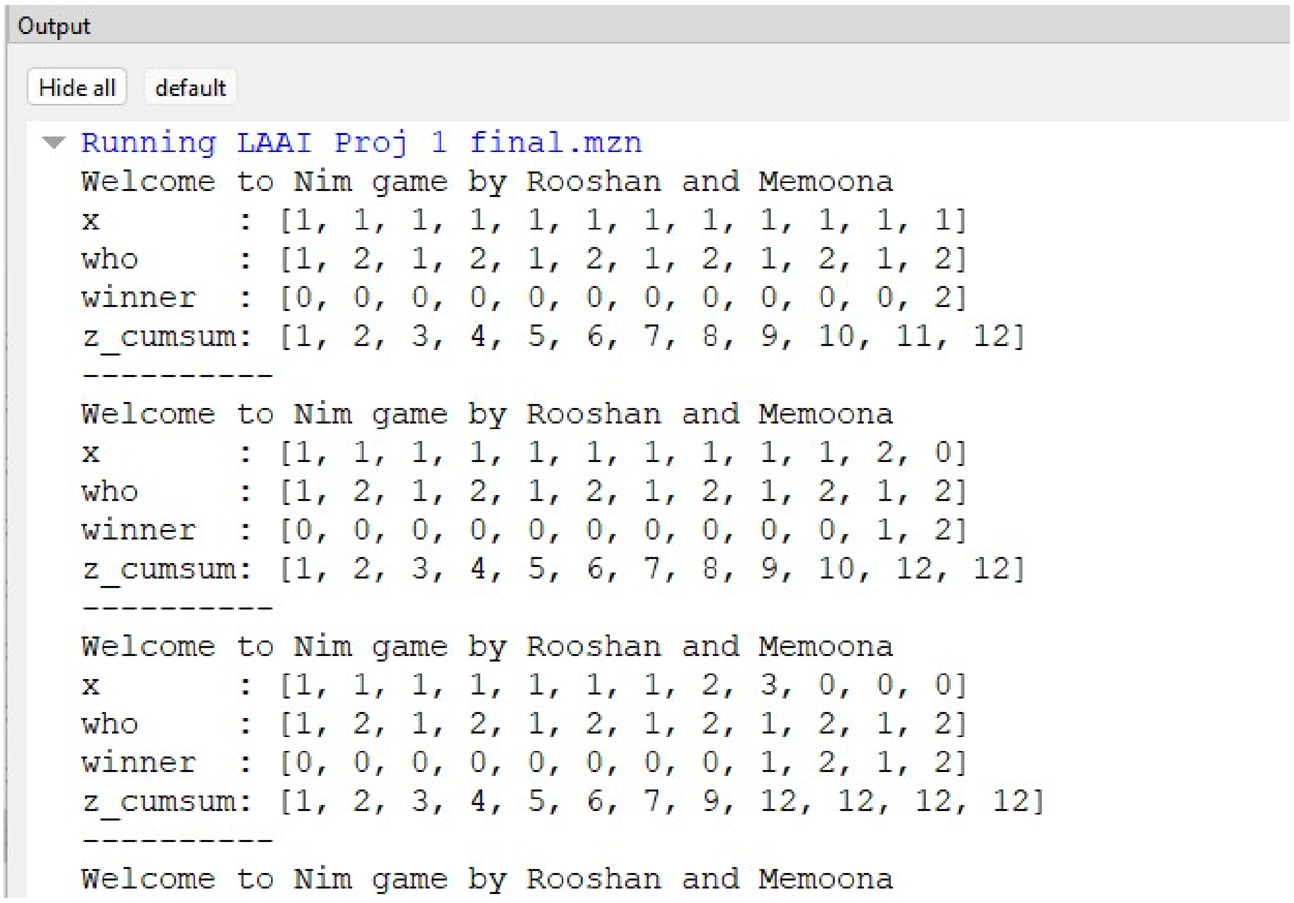
This search annotation means that we should search by selecting from the array of integer variables x, the variable with the smallest current domain (this is the first_fail rule), and try setting it to its smallest possible value (indomain_min value selection).

0.4.6 Code

```
int: n; % number of matchsticks
int: max_num_to_take;
int: num_players;
% the move
array[1..n] of var 0..max_num_to_take: x; % decision variable
% which player of the move
array[1..n] of 1..num_players: who = [ 1+(i mod num_players) | i in 0..n-1] ;
array[1..n] of var 0..num_players: winner;
var 0..n: z =  sum(i in 1..n) (x[i]);
array[1..n] of var 0..n: z_cumsum; % cumulative sum of moves
constraint
 forall(i in 1..n) (
    z < n ->
    (
      who[i] = 1+((i-1) mod num_players)
      /\
      x[i] in 1..max_num_to_take
    )
 )
 /\ % no 0 before any real move
 forall(i in 1..n) (
    x[i] > 0 <->
    not exists(j in 1..i) ( x[j] = 0 )
 )
 /\ % calculate cumulative sum of moves, and announce the winner
 forall(i in 1..n) (
   z_cumsum[i] = sum(j in 1..i) ( x[j] )
   /\
   (z_cumsum[i] = n <-> winner[i] = who[i])
   /\
   (z_cumsum[i] != n <-> winner[i] = 0)
 )
 /\
 z <= n;

 solve :: int_search(x, first_fail, indomain_min)  maximize sum(i in 1..n) (bool2int(winner[i] = 1));

output [
  "Welcome to Nim game by Rooshan and Memoona\n",
  "x       : ", show(x), "\n",
  "who     : ", show(who), "\n",
  "winner  : ", show(winner), "\n",
  "z_cumsum: ", show(z_cumsum), "\n",
];
```

0.4.7 Results

Minizinc gives us multiple results on the basis of our search annotation



```
Running LAAI Proj 1 final.mzn
Welcome to Nim game by Rooshan and Memoona
x       : [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
who     : [1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
winner  : [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2]
z_cumsum: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
----------
Welcome to Nim game by Rooshan and Memoona
x       : [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 0]
who     : [1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
winner  : [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 2]
z_cumsum: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 12]
----------
Welcome to Nim game by Rooshan and Memoona
x       : [1, 1, 1, 1, 1, 1, 1, 2, 3, 0, 0, 0]
who     : [1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
winner  : [0, 0, 0, 0, 0, 0, 0, 0, 1, 2, 1, 2]
z_cumsum: [1, 2, 3, 4, 5, 6, 7, 9, 12, 12, 12, 12]
----------
Welcome to Nim game by Rooshan and Memoona
```

0.4.8 Conclusion

By implementing this project, we have practised the basic concepts required for the development of a model in minizinc. We have practiced declaring the variables, constraints and usind the search annotations in an effective way. Project also gave us the clarity on the working of AI languages and how they are different from the conventional programming languages.

0.5 References

[1] Basic Modelling in MiniZinc | The MiniZinc Handbook 2.5.5. https://www.minizinc.org/doc-2.5.5/en/modelling.html.

[2] Predicates and Functions | The MiniZinc Handbook 2.5.5. https://www.minizinc.org/doc-2.5.5/en/predicates.html.

[3] Search | The MiniZinc Handbook 2.5.5. https://www.minizinc.org/doc-2.5.5/en/mzn_search.html.

[4] ‘Nim’. Wikipedia, 31 Aug. 2022. Wikipedia, https://en.wikipedia.org/w/index.php?title=Nim\&oldid=1107774244.

[5] Sterling, Leon, and Ehud Y. Shapiro. The Art of Prolog: Advanced Programming Techniques. 2nd ed, MIT Press, 1994.

[6] SWISH -- SWI-Prolog for SHaring. https://swish.swi-prolog.org/.