Automated Reasoning

Knights Tour

Edoardo Lenzi
University of Udine, Italy - Udine

January 2, 2021

Abstract

Given a $n \times n$ checkerboard with two knights and an arbitrary number of inaccessible cells find a visit that maximizes the number of visited cells following those constraints:

- the knights cannot walk on already visited/inaccessible cells
- the knights can only move one time per turn in alternated order
- the first stalled knight determines the end of the visit

NPC Consideration

Let simplify the problem considering only a **single knight without any inaccessible cell**; independently from the initial position **we can build** from the checkerboard **a graph** where each cell is a node and each arc is created following the knight movement pattern.

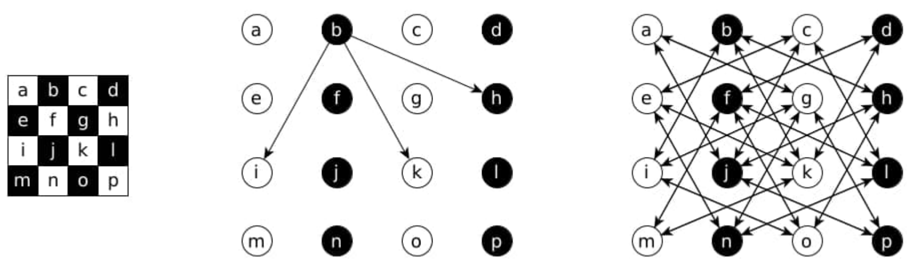

Figure 0.1: Consider the checkerboard on the left, starting from any node (i.e. from b), we can connect that cell to the reachable neighbors with a directed arc; proceeding recursively in this way we get the graph on the right.

Now if we place the knight in the `a` cell then we have to add an arc from any other node to `a`, in order to get an **instance really similar to the TSP** (Traveling Salesman Problem) problem. We assume that the weights on the arcs are all fixed to the same constant value.

> If we are able to prove that any instance of **the knights tour problem** can be converted into a TSP instance, then the whole problem **is for sure NPC** by definition.

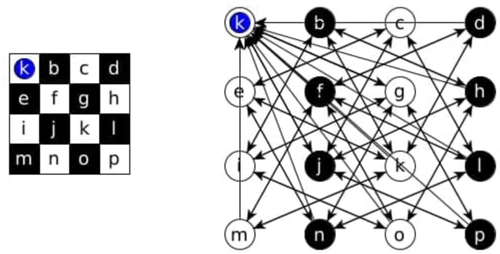

Figure 0.2: Knight in the *a* cell

Now lets assume for a moment that we have a small checkerboard (i.e. 4x4) and that even scanning the whole solution space **it isn't possible to cover the whole board with the knight** in a certain initial position.

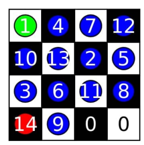

Figure 0.3: Hypothetical tour with two free leaves.

In this case the rigid structure of the checkerboard in combination with an uncomfortable initial position **does not allow to cover the full graph**; anyway this is not a big deal because **we can simply exclude a priori the uncovered nodes and get again a valid TSP instance**. The same kind of reasoning can be applied to the case in which we have an **arbitrary number of inaccessible cells**.

It is not trivial to demonstrate formally that even with two different knights we still get a TSP problem, from the point of view of a single knight; in this case we can think that **the path of the second knight is fixed a priori** and we can consider its covered cells **as inaccessible cells** to be excluded from the graph.

For sure this is not a complete and detailed demonstration but **shows in a while that the complexity of the problem is really high**. What is missing in the previous ideas is that **the two knights somehow are trying to collaborate in order to cover as much cells as possible** but we can always consider a valid solution from an **oracle and build two valid TSP instances** for the two knights dividing the cells in two partitions.

In conclusion we can assume, without fear, that **this problem is NPC.**

Naive Solution - Baseline

The fact that this problem is NPC means that in the best case we can find an **heuristic** able to decrease a little bit the complexity of a **full scan** of the search space (naive solution).

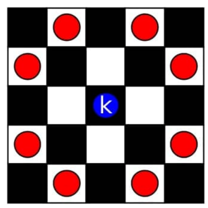

Figure 0.4: If the knight is in a central position it can reach 8 cells

We can try to **estimate the average degree of the nodes** of a $n \times n$ checkerboard:

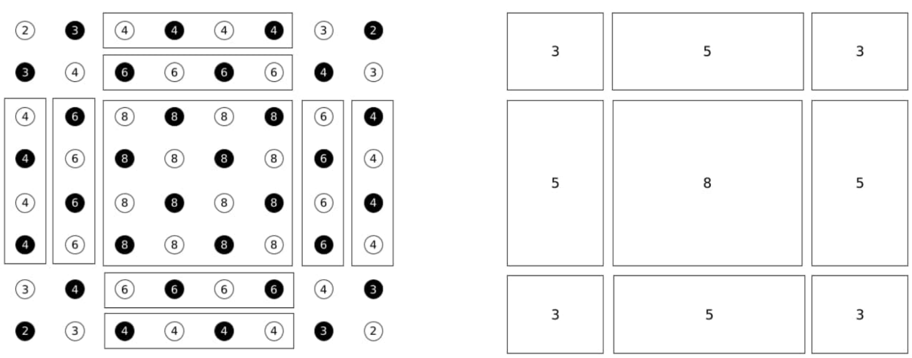

Figure 0.5: Estimation of the node degrees

$$
\begin{aligned}
\text{avgDegree}(n) &= \frac{8 * (n-4)^2 + 3 * 16 + 5 * (n-4) * 8}{n^2} \\
\text{avgDegree}(8) &= 5.25 \\
\text{avgDegree}(16) &= 6.56
\end{aligned} \tag{1}
$$

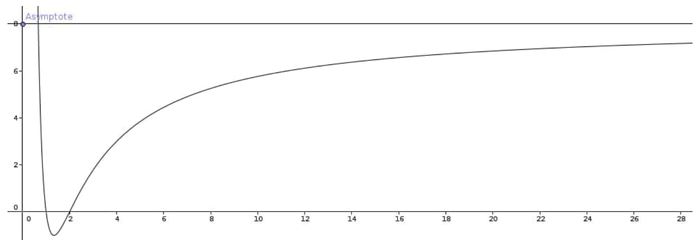

Figure 0.6: Estimation of the node degrees

For our instances (8, 10, 12, 14, 16) **the average degree of a node is ∼6** so we can **estimate also the dimensions of the search space** of the naive solution with the following approximations:

- Max **number of turns** $= n \times n$
- In the i-th turn the **probability that a cell was already covered** is $\frac{i}{n \times n}$ (heavy approximation)
- In the i-th turn a knight has $6 * (1 - \frac{i}{n \times n})$ **possible choices for a move**

> With a large approximation error the **search space** is $O((6 * \frac{1}{2})^{n^2}) = O(3^{n^2})$; so even the smallest test environment 8×8 has a search space of $\sim 3e30$ which is impossible to scan exhaustively.

COP to CSP

The first question that comes to my mind when I read this problem was:

> Is it possible to convert this COP into a CSP? Is it possible to estimate a priori the (optimum) maximum number of covered cells?

In my opinion **this is not possible** because the positions of the knights are randomly initialized such as the number of inaccessible cells; this random factor combined with the knight move pattern **is extremely hard (maybe impossible) to be predicted a priori in polynomial time**.

Therefore **this problem will be encoded as a maximization COP**.

Data Structures

We can use a **3D matrix** of variables as data structure; the idea is that we have a $n \times n$ **checkerboard** (2D matrix) **screenshot for each turn** (introduce the time notion).

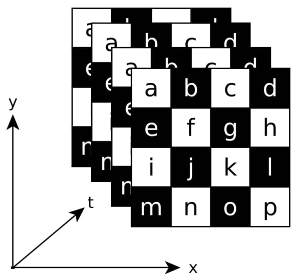

Figure 0.7: Naive 3D data structure

The **memory cost** of this data structure is $(n \times n) * n^2 = n^4$ because the checkerboard has $n \times n$ cells and the maximum number of turns is n^2. This is **acceptable** because the biggest instance is 16×16 and if (for instance) an integer takes 32 bits (such as in Java) then the whole data structure takes $(4 * 16^4)/10^6 = 0.26$ MB.

This representation is nice because it **models explicitly the time (turn) notion** and in the case of **ASP implementation can be encoded in a very simple model** declaration.

Another valid data structure is a **single matrix** $n \times n$ in which each cell is an integer $\in \{0..(n \times n)\}$.

2	13	14	1
9	10	5	6
15	4	11	12
0	7	8	3

Figure 0.8: Naive 2D data structure

The meaning is the following:

- 0: the cell was **not visited**
- 1: the cell was **inaccessible**
- **even** labeled cell: on the **path of knight 1**
- **odd** labeled cell: on the **path of knight 2**

This is a **very compact representation** that gives us at the same time both the paths followed by the knights and the free/occupied cells.

Finally we can adopt as data structure **two 1D arrays per knight** in which we will **store the coordinates (x and y) of each move** of the knight; this is a good choice but it **requires an additional data structure** (for instance a matrix) in order to keep track of the already visited cells **or an heavy constraint** (for instance `all_different`) to ensure that the knight is moving only **on free cells**.

In the following chapters we will discuss separately the Minizinc and ASP implementations that are out of the scope of this generic discussion.

Minizinc Implementation

In this chapter we will see **some different implementations** and provide some considerations on the implementation choice.

For Minizinc the **fastest data structure is the 2D matrix** described in the previous chapter (this was my first experimental result); the advantage of this structure is that **it can be also easily managed and optimized** with global constraints and `int_search`.

Any implementation can be decomposed into a **database file** and a **model declaration**; the database contains only the **initialization parameters** (`n, k, inaccessible cells and knights starting positions`) while the main interesting sections of the model declaration are:

- Constraints for the **knight movement pattern**
- Constraints for the **turn** of the knights
- Constraints for the **free cells** (labeled with `0`)
- Maximization criterion and **search strategy**

The declaration of the **movement pattern constraint** is the most critical part of the whole model; in fact doesn't exists any keyword or global constraint (not a single one among the 195 builtin global constraints, I double check them manually one by one!) able to simplify this declaration.

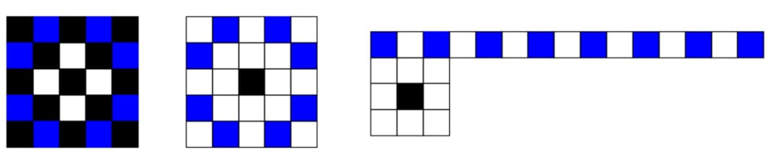

Figure 0.9: The knight movement pattern starting from a black cell means consider all the white cells in the second concentric square (the opposite when starting from a white cell). If you "unroll" the second concentric square then you have to consider only the odd indexes of the cells.

I thought a lot about the best way for encoding this pattern but **in Minizinc the best way is to enumerate all the eight possible choices**.

Even in the literature (as far as I know) there is not any other possible way of declare the movement pattern else than list, one by one, all the eight possible movements in a disjunction.

There are two ways for declaring this pattern, the first is a disjunction of `if-then-else` statements (this is the safest way because you can check the checkerboard boundaries adding a candidate step only when needed); the second method consists in using a `forall` cycle in combination with the `exists` quantification, as shown in the Minizinc benchmark *knights.mzn* [1].

It does not matter which type of declaration you choose it will not improve considerably the performance of the solution, it is only a syntactic sugar (I have tried both experimentally on the same benchmark).

Another consideration about the complexity of the problem is that if we check the official benchmark [1] released by the developers of Minizinc for the knight tour, we found that the instances are checkerboards 8×8 with at most 14 turns. This is a fairly easy task compared on our task where only the minimum instance is an 8×8 checkerboard, with two knights (not one) randomly positioned and ~ 64 turns. If we try to increase the number of turns and run the benchmark script, the running time increases as well exponentially (with 60 turns it takes more than 5 minutes on my pc).

Naive Implementation

The code is reported in Appendix A , if we run the benchmark on this model we get similar results:

n	time	coverage	optimum
4	0.15s	15/16	T
5	22.2s	23/25	T
6	timeout	24/36	F

Table 0.1: Average experimental results on small benchmarks

This solution apply an efficient global constraint (modified in order to ignore the values `0` and `1`) and a good data structure but **it cannot deal with even small instances** with $n > 5$.

An important complexity factor is carried by the basic search strategy `solve maximize`; in the next section I will try to use `int_search` instead.

Naive Implementation - `int_search`

I have tried, thanks to the help of a simple script, every possible configuration of `int_search` (14 `constrain_choices` and 10 `var_choices` = 140 different configurations).

```
solve :: int_search(occ, anti_first_fail, indomain_middle, complete)
   maximize visited_cells;
```

Figure 0.10: The best configuration for the model

n	time	coverage	optimum
4	0.15s	15/16	T
5	0.20s	24/25	T
6	0.54s	35/36	T
7	4.86s	48/49	T
8	timeout	62/64	T
9	timeout	76/81	F

Table 0.2: Average experimental results on small benchmarks

Simply **changing the search strategy** we were able to increase `n` from 6 to 8 and get an **optimum coverage**; the great aspect of this search is that **even if the solution is not optimal it is really close to the optimum** while in the first example it was really far from the optimum. Now now on this will be our new baseline.

Naive Implementation - `seq_search`

Reading the Minizinc manual [2] I discovered that is also possible to apply **multiple/different search strategies, in cascade**, for different variables with `seq_search`. In this case **is not possible to test every possible combination of parameters** because the number of tests grows exponentially with the number of `int_search` applied (140^n where n is the number of **`int_search`**). Anyway I've tried to **fix the configuration used in the previous example and test against a single** `int_search` for each variable (`halting_time1`, `halting_time2`, `visited_cells`) in a one vs all fashion.

```
solve :: seq_search([
   int_search(occ, anti_first_fail, indomain_middle, complete),
   int_search([visited_cells], occurrence, outdomain_max, complete),
   int_search([halting_time1], input_order, indomain_interval,
       complete),
   int_search([halting_time1], input_order, indomain_interval,
       complete)
]) maximize visited_cells;
```

Figure 0.11: The best configuration found

n	time	coverage	optimum
4	0.14s	15/16	T
5	0.19s	24/25	T
6	0.53s	35/36	T
7	4.39s	48/49	T
8	timeout	62/64	T
9	timeout	78/81	T
10	timeout	94/100	F

Table 0.3: Average experimental results on small benchmarks

On small instances we gain really small fractions of seconds but **the 7th instance shows a constant improvement** of half a second; moreover in **the 9th instance we've found an optimal solution** with 96% of coverage (while in the previous example we get a local optimum with 93% coverage).

This shows clearly that sequential search should make the difference with big instances (even though it is hard to prove).

I have also tried to partition the checkerboard and apply different search strategies on the partitions but this always lead to very bad results; anyway I guess that this method may be applied successfully in other problems, in order to, somehow, break the symmetry and detect a fail solution sooner.

I thought a lot about how to **break the symmetry**, for example forcing some lexicographical order o applying a greedy strategy for the movements, but in any case you will probably detect sooner local optimums but preclude definitely global optimums.

How to Cover the Whole Benchmark

We have to keep in mind that this script should work also on 16×16 checkerboards; if we try to run the previous implementation on a 12×12 it will output `UNKNOWN` since the search strategy seems extremely efficient on small instances but **unable to find a single local optimum solution**.

If we apply instead `indomain_middle` as `constrain_choice` it will rarely scan exhaustively the whole search space even on small instances but find really fast a local optimum which is, in most of the cases, also a global optimum.

Recently was introduced in the last version of Minizinc an annotation called `restart` this is still a little bit buggy (i.e. `luby` and `geometric restarts` crashes) but `linear_restart` gives a little contribution to my final implementation with the biggest instances.

Multithreading, even on small thread pools, impacts drammatically on the performances. Now we were able to cover also the 14×14 checkerboard with only 4 threads; unfortunately my laptop is not able to compute the 16×16

instance but I guess that on stronger machines with larger thread pools it will run.

ASP Implementation

In this chapter we will see the ASP implementation, in this case the first solution tested was also the best, I don't want to compare it with some other choices (as done in the previous chapter) because this should be a short paper and the solution proposed runs very well also on the 16×16 checkerboard.

Naive Implementation

Let me explain the proposed solution chunk by chunk in the simplest possible way.

Define two **domain predicates** `pos/1` and `time/1` in order to ensure, in the bodies of the clauses whether a free variable is a **time** or a **position coordinate**.

```
pos(1..n).
time(1..n**2).
```

In order to specify the movement pattern, define the `candidateStep(T,X,Y)` predicate as a possible cell with coordinates `(X,Y)`, reachable from knight position (`occ(T,X2,Y2)`) at time `T`.

```
candidateStep(T,X+2,Y+1) :- occ(T,X,Y), time(T), pos(X+2), pos(Y+1),
    T>1.
candidateStep(T,X+2,Y-1) :- occ(T,X,Y), time(T), pos(X+2), pos(Y-1),
    T>1.

% ...

candidateStep(T,X-2,Y-1) :- occ(T,X,Y), time(T), pos(X-2), pos(Y-1),
    T>1.
```

We have to specify $T > 1$ due to our encoding choice described in the previous chapter.

Define the predicate `step(T,X,Y)` as the definite decision of the knight to visit the cell `(X,Y)` at time `T+2` (next turn). It could never happen that a step has a correspondent `candidateStep`, this constraint forces the knight to adopt the knight movement patter; moreover foreach time `T` only one `step` can be chosen (ensures a single movement per turn).

```
occ(T+2,X,Y) :- step(T,X,Y), time(T), pos(X), pos(Y), time(T+2).
 :- step(T,X,Y), not candidateStep(T,X,Y), pos(X), pos(Y), time(T).
0 { step(T,X,Y) : pos(X), pos(Y) } 1 :- time(T).
```

We have almost described the whole problem, the only missing part is the goal definition; define as `haltingTime(T)` the turn in which a knight is stalled, then declare that it could never happen that a stalled knight can move any more.

```
haltingTime(T+2) :- time(T), {step(T,X,Y) : pos(X), pos(Y) } = 0, T > 3.
 :- occ(T, X, Y), haltingTime(T), time(T), pos(X), pos(Y).
```

Finally define `coverage` as the minimum `haltingTime` and try to maximize it as search strategy.

```
coverage(S) :- S = #min {T : haltingTime(T)}, time(S).
#maximize { T@1 : coverage(T) }.
```

Considerations

In my opinion this is an efficient representation, it models the checkerboard implicitly with the `occ` predicate so it should be a light data structure in terms of memory. It is true that **it uses the negation** but I'm not able to design any alternative declaration without using negation (not even using negative predicates); anyway **it works very well also on the biggest instance** 16×16.

Code Overview

In this chapter we will see a brief overview of the architecture of the code hosted on GitHub [3]; the code was written and tested in **Python 3.8.6** [4], **Minizinc 2.5.3** [5] and **Clingo 5.4.0** [6].

Installation

In order to install **AR-Knights-Tour** make sure to have installed Python, Minizinc and Clingo on your machine (better if you upgrade them to the latest stable version).

```
$> git clone https://github.com/EdoardoLenzi9/AR-Knights_Tour
$> cd AR-Knights_Tour/
$> python3 -m pip install -r requirements.txt

$> nano .env
# replace with your minizinc installation folder
MINIZINC="/home/<user>/App/MiniZincIDE-2.5.3-bundle-linux-x86_64"

$> python3 knights_tour.py -h
```

Figure 0.12: Launch these commands on a full-screen shell

Figure 0.13: If your installation was completed successfully, you will see the helper

The script has it's own cli parser and behaves exactly as any other native bash command; the main cli arguments are:

- **generate**: generates a new benchmark (100 instances)
- **run**: runs a benchmark script (from the folder */assets/runs*)
- **clean**: delete old results, logs and temporary files (from */assets/logs*)

```
$> python3 knights_tour.py --clean
$> python3 knights_tour.py --generate my_benchmark.json
$> python3 knights_tour.py --run my_benchmark.json
$> cd assets/logs ; ls
$> echo "Here there are your results!"
```

Figure 0.14: Example of cli commands

Project Pipeline

Starting from a **benchmark** (a `.json` file saved in `assets/runs/`) and the **templates** of the models and the commands, a new **log folder is created** (in `assets/logs`).

Inside the log folder there are: the result, the model, the database and the bash command (this **allows to reproduce the result at any time**).

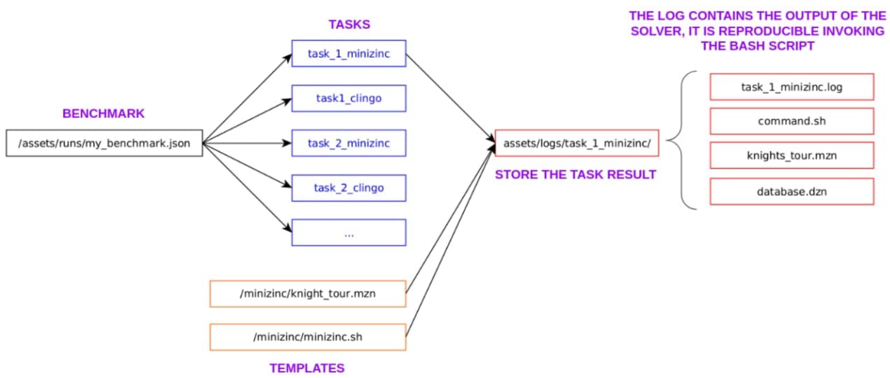

Figure 0.15: Pipeline

In terms of **business logic** the idea is that the **builders scripts** take care of replacing inside the templates the correct definitions of the parameters of the **task** (building model, database and bash command in the log folder).

The **solver** is invoked by `CliHandler` which **spawns a new process for each task**; when the sub-process ends the **output** (`Solution`) **is captured, parsed, validated and stored in the log**.

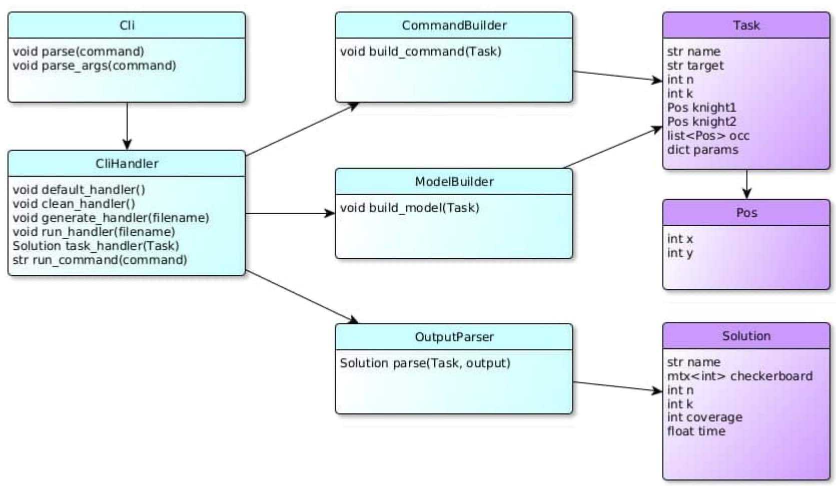

Figure 0.16: Script architecture: data classes in purple and business logic classes in light blue.

Benchmark Results

In this chapter we will analyze the outcome of a whole benchmark test and briefly comment it out. Here we present only the average results because them are too many to fit a page but the original results are stored in the log of the GitHub repository [3].

n	**time**	**coverage**	**target**
8×8	5 mins	$0/64 \pm 0$	Minizinc
8×8	5 mins	$0/64 \pm 0$	Clingo
10×10	5 mins	$0/100 \pm 0$	Minizinc
10×10	5 mins	$0/100 \pm 0$	Clingo
12×12	5 mins	$0/144 \pm 0$	Minizinc
12×12	5 mins	$0/144 \pm 0$	Clingo
14×14	5 mins	$0/196 \pm 0$	Minizinc
14×14	5 mins	$0/196 \pm 0$	Clingo
16×16	5 mins	$0/256 \pm 0$	Minizinc
16×16	5 mins	$0/256 \pm 0$	Clingo

Table 0.4: Average experimental results on a benchmark.

Conclusions

Appendix A - Minizinc Code

```
n = 4;
k = 1;
initial_occ = array2d(CELL_DOMAIN, CELL_DOMAIN, [ 2, 0, 0, 1,
                                                   0, 0, 0, 0,
                                                   0, 0, 0, 0,
                                                   0, 0, 0, 3 ]);
```

```
% params
int: k;
int: n;
int: max_time = n*n;

% domains
set of int: CELL_DOMAIN  = 1..n;
set of int: TIME_DOMAIN  = 1..max_time;
set of int: BOOL         = 0..1;

% variables
var 1..max_time: halting_time1;  % knight 1 - halting_time
var 1..max_time: halting_time2;  % knight 2 - halting_time
var 1..max_time: visited_cells;  % number of visited cells

array[ CELL_DOMAIN, CELL_DOMAIN ] of 0..max_time: initial_occ;
array[ CELL_DOMAIN, CELL_DOMAIN ] of var 0..max_time: occ;


% init positions and checkerboard O(n x n)
constraint forall(i, j in 1..n)(
    if initial_occ[i, j] > 0 then
        occ[i, j] = initial_occ[i, j]
    else
        % a cell can be labeled with "1" only if initialized as 1
        occ[i, j] != 1
    endif
);

```

```

% turn constraint, one move per turn
include "globals.mzn";
include "fzn_alldifferent_except.mzn";

predicate alldifferent_except_01(array[$X] of var int: vs) =
   fzn_alldifferent_except(array1d(vs),{0, 1});

constraint alldifferent_except_01([ occ[i, j] | i,j in 1..n ]);
constraint forall(i,j in 1..n)(occ[i,j] <= visited_cells);


% motion pattern for knight 1
constraint forall(i, j in 1..n, t in 1..((n*n) div 2)-2) (
    (2*t) <= halting_time1 /\
    (2*t) <= halting_time2 /\
    occ[i,j] = (2*t)      ->  exists(k in {-2, 2}, l in {-1, 1})(
                                  (occ[i+k, j+l] = (2*t)+2) \/
                                  (occ[i+l, j+k] = (2*t)+2)
                              ) \/
                              halting_time1 = (2*t) );


% motion pattern for knight 2
constraint forall(i, j in 1..n, t in 1..((n*n) div 2)-2) (
    ((2*t)+1) <= halting_time1 /\
    ((2*t)+1) <= halting_time2 /\
    occ[i,j] = ((2*t)+1)      ->  exists(k in {-2, 2}, l in {-1, 1})(
                                      (occ[i+k, j+l] = (2*t)+3) \/
                                      (occ[i+l, j+k] = (2*t)+3)
                                  ) \/
                                  halting_time2 = ((2*t)+2) );


% maximization criterion
visited_cells = min(halting_time1, halting_time2);


% search strategy
solve maximize visited_cells;
```

This encoding is very compact and readable but generates a warning because sometimes it access an in-existent memory allocation. The following code, even if more verbose and a little bit slower, solves this problem.

```
constraint forall(i, j in 1..n, t in 1..((n*n) div 2)-2) (
    (2*t) <= halting_time1 /\
    (2*t) <= halting_time2 /\
    occ[i,j] = (2*t)      ->
```

```
    if i<n-1 /\ j<n  then (occ[i+2, j+1] = (2*t)+2) else false endif \/
    if i<n-1 /\ j>1  then (occ[i+2, j-1] = (2*t)+2) else false endif \/
    if i>2  /\ j<n   then (occ[i-2, j+1] = (2*t)+2) else false endif \/
    if i>2  /\ j>1   then (occ[i-2, j-1] = (2*t)+2) else false endif \/
    if i<n  /\ j<n-1 then (occ[i+1, j+2] = (2*t)+2) else false endif \/
    if i<n  /\ j>2   then (occ[i+1, j-2] = (2*t)+2) else false endif \/
    if i>1  /\ j<n-1 then (occ[i-1, j+2] = (2*t)+2) else false endif \/
    if i>1  /\ j>2   then (occ[i-1, j-2] = (2*t)+2) else false endif \/
    halting_time1 = (2*t) );


constraint forall(i, j in 1..n, t in 1..((n*n) div 2)-2) (
    ((2*t)+1) <= halting_time1 /\
    ((2*t)+1) <= halting_time2 /\
    occ[i,j] = ((2*t)+1)     ->
   if i<n-1 /\ j<n  then (occ[i+2, j+1] = ((2*t)+3)) else false endif \/
   if i<n-1 /\ j>1  then (occ[i+2, j-1] = ((2*t)+3)) else false endif \/
   if i>2  /\ j<n   then (occ[i-2, j+1] = ((2*t)+3)) else false endif \/
   if i>2  /\ j>1   then (occ[i-2, j-1] = ((2*t)+3)) else false endif \/
   if i<n  /\ j<n-1 then (occ[i+1, j+2] = ((2*t)+3)) else false endif \/
   if i<n  /\ j>2   then (occ[i+1, j-2] = ((2*t)+3)) else false endif \/
   if i>1  /\ j<n-1 then (occ[i-1, j+2] = ((2*t)+3)) else false endif \/
   if i>1  /\ j>2   then (occ[i-1, j-2] = ((2*t)+3)) else false endif \/
   halting_time2 = ((2*t)+2)
);
```

Appendix B - Clingo Code

```

%---------------------------------------------------------------------------%
%                               DOMAINS
%---------------------------------------------------------------------------%

pos(1..n).        % available positions (x, y coordinates)
time(1..n**2).    % turns domain


%---------------------------------------------------------------------------%
%                             CONSTRAINTS
%---------------------------------------------------------------------------%

% At time T, a candidateStep is a possible target for the next move
candidateStep(T,X+2,Y+1) :- occ(T,X,Y), time(T),T>1, pos(X+2), pos(Y+1).
candidateStep(T,X+2,Y-1) :- occ(T,X,Y), time(T),T>1, pos(X+2), pos(Y-1).
candidateStep(T,X+1,Y+2) :- occ(T,X,Y), time(T),T>1, pos(X+1), pos(Y+2).
candidateStep(T,X+1,Y-2) :- occ(T,X,Y), time(T),T>1, pos(X+1), pos(Y-2).
candidateStep(T,X-1,Y+2) :- occ(T,X,Y), time(T),T>1, pos(X-1), pos(Y+2).
candidateStep(T,X-1,Y-2) :- occ(T,X,Y), time(T),T>1, pos(X-1), pos(Y-2).
candidateStep(T,X-2,Y+1) :- occ(T,X,Y), time(T),T>1, pos(X-2), pos(Y+1).
candidateStep(T,X-2,Y-1) :- occ(T,X,Y), time(T),T>1, pos(X-2), pos(Y-1).

% When a step is selected (it must be a valid step)
% then the corresponding occ is true
occ(T+2,X,Y) :- step(T,X,Y), time(T), pos(X), pos(Y), time(T+2).

% It could never happen that a knight moves on an already-visited cell
 :- occ(T1,X,Y), occ(T2,X,Y), T1 < T2, time(T1), time(T2).

% It could never happen that a step is not a candidateStep
 :- step(T,X,Y), not candidateStep(T,X,Y), pos(X), pos(Y), time(T).

% For each turn the knight can do at most one step
0 { step(T,X,Y) : pos(X), pos(Y) } 1 :- time(T).

% It could never happen that after the haltingTime (last turn)
```

```
% a knight continues its tour
 :- occ(T, X, Y), haltingTime(T), time(T), pos(X), pos(Y).

% If at time T a knight is stalled (not a single candidateStep)
% then haltingTime(T) is true
haltingTime(T+2) :- time(T), {step(T,X,Y) : pos(X), pos(Y) } = 0, T > 3.

% Coverage is the maximization criterion = minimum haltingTime
coverage(S) :- S = #min {T : haltingTime(T)}, time(S).


%-------------------------------------------------------------------------%
%                            SEARCH STRATEGY
%-------------------------------------------------------------------------%

% Maximize the coverage
#maximize { T@1 : coverage(T) }.



%-------------------------------------------------------------------------%
%                                 OUTPUT
%-------------------------------------------------------------------------%

#show occ/3.
#show coverage/1.
%#show haltingTime/1.
```

Bibliography

[1] **Minizinc Knights Benchmarks**. https://github.com/MiniZinc/minizinc-benchmarks/blob/master/knights.

[2] **Minizinc Search Documentation**. https://www.minizinc.org/doc-2.5.0/en/mzn_search.html.

[3] **Knights Tour - GitHub**. https://github.com/EdoardoLenzi9/AR-Knights_Tour.

[4] **Python3 Official Page**. https://www.python.org/.

[5] **Minizinc Official Page**. https://www.minizinc.org/.

[6] **Clingo**. https://potassco.org/clingo/.

Disclaimer

The material contained in these text is restricted to students and professors of the course of the Master degree in Computer Science at University of Udine.

It prohibited any use other than that inherent to the course, and in particular is expressly prohibited its use for any commercial purposes and/or for profit.