Università degli Studi di Udine - Dipartimento di Scienze matematiche, informatiche e fisiche

Automated Reasoning - Quantum Planning

Nazareno Piccin - 170521

March 23, 2024

Contents

1 Problem Definition **2**

2 Models **2**
- 2.1 General discussion . . . 2
- 2.2 Minizinc . . . 3
  - 2.2.1 Utility predicates and functions . . . 3
  - 2.2.2 V1 Model . . . 4
  - 2.2.3 V2 Model . . . 5
- 2.3 ASP . . . 7
  - 2.3.1 V1 Model . . . 7
  - 2.3.2 V2 Model . . . 8

3 Benchmarks **10**
- 3.1 Minizinc . . . 10
- 3.2 ASP models . . . 11

1 Problem Definition

The input data is n and a Boolean Matrix $n \times n$. The goal is reaching the unitary $n \times n$ Matrix. There is an unique schema of "evolution" rule: Choose two (different) rows i and j ($i < j$ or $j < i$ are both ok) and replace row i with their XOR (bit-to-bit). Find the minimum path leading to the unitary Matrix (btw, if the determinant of the starting matrix is not 0, the path exists always, consider that when you prepare tests).

2 Models

2.1 General discussion

To solve the problem we need to first understand what it means for matrix to be unitary.

Definition 2.1 (Unitary Matrix)**.** An invertible complex square matrix A is **unitary** if its matrix inverse A^{-1} equals its conjugate transpose A^*, that is if:

$$AA^* = A^*A = I$$

In the case of real numbers, the analogue of a unitary matrix is an orthogonal matrix. That means checking if:

$$AA^T = A^TA = I$$

A real square matrix is orthogonal if and only if its columns form an orthonormal basis. Since we have a boolean matrix, all unit vectors must have exactly only one 1 and for them to be orthogonal they must all have that 1 in a different dimension (i.e. row). We will then devise two different unitary checks:

- **V1**: compute the transpose of the matrix, multiply them and check if the resulting matrix is the identity matrix
- **V2**: check if for each row and column there is exactly one 1

Other useful definitions:

- $n \in \mathbb{N}$: matrix size
- $M \in \mathcal{M}_{n\times n}(\{0,1\})$: input matrix
- $max\_t \in \mathbb{N}$: maximum length of the plan
- $steps \in \{1, \dots, max\_t\}$: current length of the plan
- $A_t \in \mathcal{M}_{n\times n}(\{0,1\})$: computation matrix at time t ($A_0 = M$)

We will make use of a notion of time and of a computation matrix to automatically devise a feasible plan. max_t must be manually provided because we are trying to model a planning problem in Minizinc and ASP, which in general is a `PSPACE-Complete` problem.

If there exists a feasible plan with length $steps \leq max\_t$, the models will provide one that ensures that A_{steps} is unitary and will try to minimize $steps$ where possible.

General details regarding the following models: during the plan generation we creation of non-contiguous plans is avoided and when a unitary matrix is found the plan is explicitly completed with “null-like“ operations until max_t is reached.

2.2 Minizinc

2.2.1 Utility predicates and functions

```
predicate is_unitary(array[int,int] of var int: A) =
  is_identity(matrix_mul(A, transpose(A)));

predicate is_identity(array[int,int] of var int: A) =
  let { int: n = max(index_set_1of2(A)) } in
  forall(i,j in 1..n where i != j)(A[i,i] = 1 /\ A[i,j
     ] = 0);

function array[int,int] of var int: get_matrix_at(
   array[int,int,int] of var int: A, var int: time) =
  let { int: n = max(index_set_1of3(A)) } in
  array2d(1..n, 1..n, [A[i,j,time] | i,j in 1..n]);

function array[int,int] of var int: transpose(array[
   int,int] of var int: A) =
  let { int: n = max(index_set_1of2(A)) } in
  array2d(1..n, 1..n, [A[j,i] | i,j in 1..n]);

function array[int,int] of var int: matrix_mul(array[
   int,int] of var int: A, array[int,int] of var int:
   B) =
  let { int: n = max(index_set_1of2(A)) } in
  array2d(1..n, 1..n, [sum(k in 1..n)(A[i,k] * B[k,j])
      | i,j in 1..n]);

function var int: determinant(array[int,int] of var
   int: A) =
  let { int: n = max(index_set_1of2(A)) } in
```

```
  if n = 1 then
    A[1,1]
  else
    sum(i,j in 1..n)((-1)^(i+j) * A[i,j] * determinant
        (sub_matrix(A, i, j)))
  endif;

function array[int,int] of var int: sub_matrix(array[
    int,int] of var int: A, int: row_ex, int: col_ex) =
  let { int: n = max(index_set_1of2(A)) } in
  array2d(1..(n-1), 1..(n-1), [A[i,j] | i,j in 1..n
      where i != row_ex /\ j != col_ex]);
```

2.2.2 V1 Model

```
include "globals.mzn";
include "utils.mzn";

% Constants
int: n;
int: max_t;

% Variables
array[1..n, 1..n] of 0..1: M;
array[1..n, 1..n, 0..max_t] of var 0..1: A;
array[1..2, 1..max_t] of var 0..n: plan;
var 0..max_t: steps;

% Initial configuration
constraint forall(i,j in 1..n)(A[i,j,0] = M[i,j]);

% Plan generation
constraint forall(t in 1..max_t)(plan[1,t] = 0 -> plan
    [2,t] = 0);
constraint forall(t in 1..max_t)(plan[1,t] != 0 ->
    plan[1,t] != plan[2,t]);

constraint forall(t in steps+1..max_t)(plan[1,t] = 0
    /\ plan[2,t] = 0);
constraint forall(t in 1..max_t-1)(plan[1,t] = 0 ->
    plan[1,t+1] = 0);

% Final goal
% constraint forall(i,j in 1..n where i != j)(sum(k in
     1..n)(A[i,k,steps] * A[i,k,steps]) = 1 /\ sum(k in
     1..n)(A[i,k,steps] * A[j,k,steps]) = 0);
```

```
% Same as above, same speed
% constraint forall(i,j in 1..n where i != j)(sum(k in
    1..n)(A[i,k,steps] * A[j,k,steps]) = 0);
% constraint forall(i in 1..n)(sum(k in 1..n)(A[i,k,
   steps] * A[i,k,steps]) = 1);

% Same as above, same speed
constraint is_unitary(get_matrix_at(A, steps));

% Evolution "split" version
% constraint forall(i,j in 1..n, t in 1..max_t)(plan
   [1,t] = i -> A[i,j,t] = (A[i,j,t-1] + A[plan[2,t],j
   ,t-1]) mod 2);
% constraint forall(i,j in 1..n, t in 1..max_t)(plan
   [1,t] != i -> A[i,j,t] = A[i,j,t-1]);

% Evolution "compound" version
constraint forall(i,j in 1..n, t in 1..max_t)(if plan
   [1,t] = i then A[i,j,t] = (A[i,j,t-1] + A[plan[2,t
   ],j,t-1]) mod 2 else A[i,j,t] = A[i,j,t-1] endif);

output ["M = \n" ++ show2d(M) ++ ";\n"];
output ["xor rows \(plan[1,t]) and \(plan[2,t])\n" ++
   "A[\(t)] = \n" ++ show2d(get_matrix_at(A, t)) ++
   ";\n" | t in 1..fix(steps)];
output ["Final matrix in \(steps) steps, is it unitary
   ? \(is_unitary(get_matrix_at(A, steps)))\n"];
output ["Plan = \n" ++ show2d(plan) ++ ";\n"];
solve %:: int_search(plan, input_order,
   indomain_random)
      minimize steps;
```

2.2.3 V2 Model

```
include "globals.mzn";
include "utils.mzn";

% Constants
int: n;
int: max_t;

% Variables
array[1..n, 1..n] of 0..1: M;
array[1..n, 1..n, 0..max_t] of var 0..1: A;
array[1..2, 1..max_t] of var 0..n: plan;
```

```
var 0..max_t: steps;

% Initial configuration
constraint forall(i,j in 1..n)(A[i,j,0] = M[i,j]);

% Plan generation
constraint forall(t in 1..max_t)(plan[1,t] = 0 -> plan
   [2,t] = 0);
constraint forall(t in 1..max_t)(plan[1,t] != 0 ->
   plan[1,t] != plan[2,t]);

constraint forall(t in steps+1..max_t)(plan[1,t] = 0
   /\ plan[2,t] = 0);
constraint forall(t in 1..max_t-1)(plan[1,t] = 0 ->
   plan[1,t+1] = 0);

% Final goal
constraint forall(i in 1..n)(count([A[i,j,steps] | j
   in 1..n], 1) = 1);
constraint forall(j in 1..n)(count([A[i,j,steps] | i
   in 1..n], 1) = 1);

% Evolution "split" version
% constraint forall(i,j in 1..n, t in 1..max_t)(plan
   [1,t] = i -> A[i,j,t] = (A[i,j,t-1] + A[plan[2,t],j
   ,t-1]) mod 2);
% constraint forall(i,j in 1..n, t in 1..max_t)(plan
   [1,t] != i -> A[i,j,t] = A[i,j,t-1]);

% Evolution "compound" version
constraint forall(i,j in 1..n, t in 1..max_t)(if plan
   [1,t] = i then A[i,j,t] = (A[i,j,t-1] + A[plan[2,t
   ],j,t-1]) mod 2 else A[i,j,t] = A[i,j,t-1] endif);

output ["M = \n" ++ show2d(M) ++ ";\n"];
output ["xor rows \(plan[1,t]) and \(plan[2,t])\n" ++
   "A[\(t)] = \n" ++ show2d(get_matrix_at(A, t)) ++
   ";\n" | t in 1..fix(steps)];
output ["Final matrix in \(steps) steps, is it unitary
   ? \(is_unitary(get_matrix_at(A, steps)))\n"];
output ["Plan = \n" ++ show2d(plan) ++ ";\n"];
solve %:: int_search(plan, input_order,
   indomain_random)
      minimize steps;
```

2.3 ASP

2.3.1 V1 Model

```
% Constants
time(1..max_t).
val(0). val(1).
coord(1..n).

% Initial state
a(X,Y,V,0) :- coord(X), coord(Y), val(V), m(X,Y,V).

% Non-deterministic choices
0 { xor(I,J,T) : coord(I), coord(J), I != J } 1 :-
    time(T).
0 { a(X,Y,V,T) : val(V) } 1 :- coord(X), coord(Y),
    time(T).

% Contiguous plan
:- time(T), time(T-1), xor(_,_,T), not xor(_,_,T-1).

% Evolution propagation
affected(I,T) :- coord(I), coord(J), time(T), xor(I,J,
    T).

a(X,Y,S,T) :-   coord(X), coord(Y), coord(Z), val(S),
    val(V1), val(V2), time(T), steps(L),
                affected(X,T), xor(X,Z,T), a(X,Y,V1,T
                    -1), a(Z,Y,V2,T-1), S = (V1 + V2) \
                     2, T <= L.
a(X,Y,V,T) :-   coord(X), coord(Y), val(V), time(T),
    steps(L),
                not affected(X,T), a(X,Y,V,T-1), T <=
                   L.
:- coord(X), coord(Y), val(V), time(T), steps(L), a(X,
    Y,V,T), T > L.

% Plan length
steps(L) :- L = #count { T : coord(I), coord(J), time(
    T), xor(I,J,T) }.

% Matrix for unitary check
c(I,J,S) :- coord(I), coord(J), steps(L),
            S = #sum { V1 * V2, K : coord(K), val(V1),
                val(V2), a(I,K,V1,L), a(J,K,V2,L) }.
```

```
% Final goal
goal(X,Y) :- coord(X), coord(Y), c(X,Y,S), X != Y, S =
    0.
goal(X,X) :- coord(X), c(X,X,S), S = 1.
:- coord(X), coord(Y), not goal(X,Y).

#minimize { L : steps(L) }.

% Output
% #show a/4.
#show xor/3.
#show steps/1.

% Debug
% #show affected/2.
% #show c/3.
% #show goal/2.
```

2.3.2 V2 Model

```
% Constants
time(1..max_t).
val(0). val(1).
coord(1..n).

% Initial state
a(X,Y,V,0) :- coord(X), coord(Y), val(V), m(X,Y,V).

% Non-deterministic choices
0 { xor(I,J,T) : coord(I), coord(J), I != J } 1 :-
   time(T).
0 { a(X,Y,V,T) : val(V) } 1 :- coord(X), coord(Y),
   time(T).

% Contiguous plan
:- time(T), time(T-1), xor(_,_,T), not xor(_,_,T-1).

% Evolution propagation
affected(I,T) :- coord(I), coord(J), time(T), xor(I,J,
   T).

a(X,Y,S,T) :-   coord(X), coord(Y), coord(Z), val(S),
   val(V1), val(V2), time(T), steps(L),
                affected(X,T), xor(X,Z,T), a(X,Y,V1,T
                   -1), a(Z,Y,V2,T-1), S = (V1 + V2) \
                    2, T <= L.
```

```
a(X,Y,V,T) :-    coord(X), coord(Y), val(V), time(T),
   steps(L),
                 not affected(X,T), a(X,Y,V,T-1), T <=
                    L.
:- coord(X), coord(Y), val(V), time(T), steps(L), a(X,
   Y,V,T), T > L.

% Plan length
steps(L) :- L = #count { T : coord(I), coord(J), time(
   T), xor(I,J,T) }.

% Final goal
row_goal(X) :- coord(X), steps(L), S = #sum { V, Y :
   coord(Y), val(V), a(X,Y,V,L) }, S = 1.
col_goal(Y) :- coord(Y), steps(L), S = #sum { V, X :
   coord(X), val(V), a(X,Y,V,L) }, S = 1.
:- coord(X), not row_goal(X).
:- coord(Y), not col_goal(Y).

#minimize { L : steps(L) }.

% Output
% #show a/4.
#show xor/3.
#show steps/1.

% Debug
% #show affected/2.
% #show row_goal/1.
% #show col_goal/1.
```

3 Benchmarks

All instances are made of matrixes with a determinant $\neq 0$, to ensure that a plan always exists. The difficulty of each instance is determined with respect of the performance of the default Minzinc model_v1. The time-out timer is set at 300s (5 minutes). The minizinc solver configuration can be found in `Minizinc/solver_config.mpc`.
Tested on a Ryzen 5 5600 CPU with 16GB DDR4 3200Mhz CL16 RAM, Minizinc v2.8.2, Clingo v5.4.1.

3.1 Minizinc

Table 1: Minizinc model_v1

Instance	**Steps**	**v1_default (s)**	**Steps**	**v1_input-random (s)**
01.1	5	3.888	5	0.266
01.2	6	5.708	6	3.211
01.3	4	5.645	4	0.029
01.4	5	5.258	5	0.035
02.1	5	161.538	5	1.8
02.2	7	46.187	7	5.828
02.3	6	90.387	6	2.812
02.4	6	145.02	6	1.831
03.1	UNKNOWN	TIME-OUT	UNKNOWN	TIME-OUT
03.2	UNKNOWN	TIME-OUT	8	25.575
03.3	UNKNOWN	TIME-OUT	UNKNOWN	TIME-OUT
03.4	UNKNOWN	TIME-OUT	UNKNOWN	TIME-OUT

Table 2: Minizinc model_v2

Instance	**Steps**	**v2_default (s)**	**Steps**	**v2_input-random (s)**
01.1	5	3.23	5	0.222
01.2	6	6.71	6	2.694
01.3	4	5.412	4	0.026
01.4	5	7.145	5	0.03
02.1	5	154.51	5	0.976
02.2	UNKNOWN	TIME-OUT	7	5.064
02.3	6	111.277	6	2.445
02.4	6	149.594	6	1.593
03.1	UNKNOWN	TIME-OUT	UNKNOWN	TIME-OUT
03.2	UNKNOWN	TIME-OUT	8	19.996
03.3	UNKNOWN	TIME-OUT	UNKNOWN	TIME-OUT
03.4	UNKNOWN	TIME-OUT	UNKNOWN	TIME-OUT

3.2 ASP models

Table 3: ASP

Instance	Steps	v1_default (s)	Steps	v2_default (s)
01.1	5	0.213	5	0.109
01.2	6	0.506	6	0.364
01.3	4	0.077	4	0.083
01.4	5	0.154	5	0.109
02.1	5	0.204	5	0.273
02.2	7	1.126	7	1.051
02.3	6	1.106	6	0.355
02.4	6	0.234	6	0.405
03.1	UNSATISFIABLE	5.73	UNSATISFIABLE	8.204
03.2	8	3.44	8	3.025
03.3	UNKNOWN	TIME-OUT	UNKNOWN	TIME-OUT
03.4	UNSATISFIABLE	148.527	UNSATISFIABLE	111.03