Puzzle Solving with Modern SAT Engine

The Sokoban Puzzle

Given a map in which the player, boxes, targets, walls positions are configured, the player's goal is to move the player around, pushing the boxes until all targets are covered by boxes. The player's movement is limited to horizontal and vertical on the board, one grid cell at a time. The boxes can only be pushed, not pulled.

I. BMC formulation

A BMC run of depth *T* unfolds the transition relation of sequential system *T* times, and uses an SAT solver to check the satisfiability of:

$$U = I(0) \wedge \bigwedge_{k=0}^{T-1} TR(k, k+1) \wedge G(T)$$

We then check the existence of a solution with increasing bound T.

I encode the puzzle with two types of literals: the player literal $P_{row, col, pi, time}$ and box literal $B_{row, col, bi, time}$, denoting whether player *pi* is on position *row*, *col* at time step *time* and whether player *bi* is on position *row*, *col* at time step *time*, respectively.

The rest are simple. Encode the necessary constraints and transforming them into CNF form. To alleviate solver effort, constraints coming from walls are directly used to reduce number of generated clauses, which means the solver will only be considering walkable grids throughout the solving process.

Below are the proposed constraints for this puzzle:

A. Initial State Constraint

These clauses are, for example, if player is at row *2*, col *2* initially, we get something like:

1(2, 2, 0, 0)
-2(1, 3, 0, 0)
-3(1, 4, 0, 0)
-4(1, 5, 0, 0)
-5(2, 1, 0, 0)
-6(2, 3, 0, 0)
…

Where each row represents lit-index(row, col, player-index, timestep). It is the same logic with the boxes.

B. Player Movement Constraint

1. If a grid were to be occupied in the next state by the player, the player must be in one of the adjacent grids in the current state (see Fig.2):

$P_{r,c,i,t+1}$
$\Rightarrow (P_{r-1,c,i,t} \vee P_{r+1,c,i,t} \vee P_{r,c-1,i,t} \vee P_{r,c+1,i,t})$

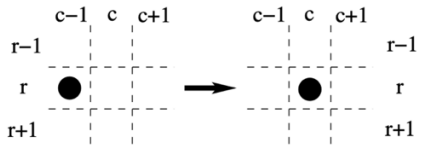

fig.2 Player moves from left to right

The constraint can be transformed into cnf easily:
For each player *i* and time step *t*:

$\neg P_{r,c,i,t+1} \vee P_{r-1,c,i,t} \vee P_{r+1,c,i,t} \vee P_{r,c-1,i,t} \vee P_{r,c+1,i,t}$

2. The player can only move to one of the adjacent grids (see Fig.3):

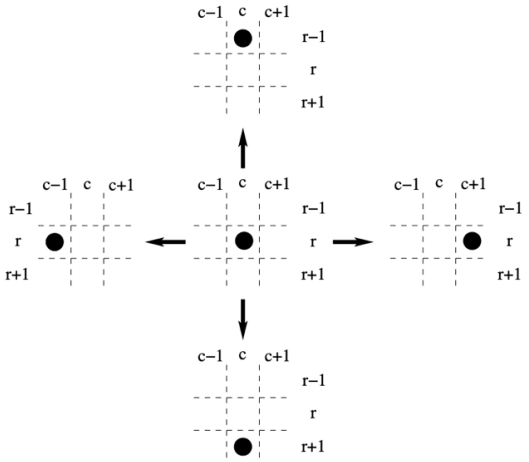

fig.3 Player can only move horizontally or vertically of one single cell

For each player *i* and time step *t*:

$P_{r,c,i,t}$
$\Rightarrow (P_{r-1,c,i,t+1} \vee P_{r+1,c,i,t+1} \vee P_{r,c-1,i,t+1} \vee P_{r,c+1,i,t+1})$

Player moving only to one single direction is ensured by *E. Player Single Placement Constraint*, where at each time frame (state), each player can only exist in

one single position. This simplifies this constraint and transforming it into cnf form is again trivial.

C. Box Push Constraint

If a grid were to be occupied in the next state by a box. It'd be either the box was already there (untouched) or being pushed by some player. If it is caused by a player push, the player must be behind the box at the current state, and the player will take the box's current position at the next state (see Fig.4):

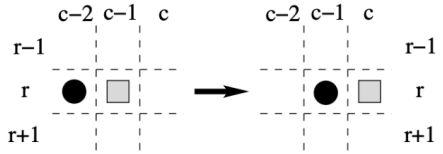

fig.4 Box and player positions for a left-to-right push move

$$
\begin{aligned}
&B_{r,c,i,t+1} \\
&\Rightarrow [B_{r,c,i,t} \vee \\
&(B_{r,c-1,i,t} \wedge P_{r,c-2,pi1,t} \wedge P_{r,c-1,pi1,t+1}) \vee \\
&(B_{r,c+1,i,t} \wedge P_{r,c+2,pi1,t} \wedge P_{r,c+1,pi1,t+1}) \vee \\
&(B_{r+1,c,i,t} \wedge P_{r+2,c,pi1,t} \wedge P_{r+1,c,pi1,t+1}) \vee \\
&(B_{r-1,c,i,t} \wedge P_{r-2,c,pi1,t} \wedge P_{r-1,c,pi1,t+1}) \vee \\
&(B_{r,c-1,i,t} \wedge P_{r,c-2,pi2,t} \wedge P_{r,c-1,pi2,t+1}) \vee \\
&\ldots]
\end{aligned}
$$

Note that in case there are multiple agents on board, iterate through all player indices from pi*1* to pi*n*. Transforming this DNF to CNF is challenging for the engines, so I translate it to CNF form myself.

To transform a DNF in the form of:

$$a + bdc + edf + \ldots$$

(i) $\neg(\neg(a + bdc + edf))$

(ii) $\neg(\neg a\ (\neg b+\neg d+\neg c)\ \ (\neg e+\neg d+\neg f))$

(iii) ¬(¬a¬b¬e+¬a¬b¬d+¬a¬b¬f+¬a¬d¬e+…)

(iv) (a+b+c)(a+b+d)(a+b+f)(a+d+e)(a+d+d)(a+d+f)(a+c+e)(a+c+d)(a+c+f)

We observe that we can generate these clauses by utilizing Cartesian Product on the sets {b,d,c}, {e,d,f}, {g,h,i},…and so on, obtaining the resulting products and adding literal "a" into each of them.

D. Player Head On Constraint

In case of a multi-agent puzzle, this constraint guarantees that no two players shall run into each other:

1. Vertical head on:

$\neg\ (P_{r,c,pi1,t} \wedge P_{r,c+1,pi1,t+1} \wedge P_{r,c+1,pi21,t} \wedge P_{r,c,pi2,t+1})$

2. Horizontal head on:

$\neg\ (P_{r,c,pi1,t} \wedge P_{r+1,c,pi1,t+1} \wedge P_{r+1,c,pi21,t} \wedge P_{r,c,pi2,t+1})$

for any pair of players {*pi1*, *pi2*}

E. Player Single Placement Constraint

F. Box Single Placement Constraint

Players and boxes should not exist in different grids at same time.

G. Player Collision Constraint

H. Box Collision Constraint

I. Box and Player Collision Constraint

Different players and boxes should not overlap with one another at all time steps.

J. Solved State Constraint

All target positions should be occupied by boxes at time step T, encoded by:

$$\bigwedge_{target} \bigvee_{box} (B_{target_r,target_c,box_{index},T})$$

II. Data Structure

For code access convenience and readability, we created three main classes: Lit, Clause, and SokobanSolver. Within the Lit class, there are operator overloading **"~"** for literal negations (simply negating an existing literal's *index* attribute, this is beneficial since it is more convenient for utilizing abc built-in Var2Lit function, for the object literal's absolute value is exactly its variable index, while its sign indicates whether the variable shall be converted into a negative or positive literal), and necessary attributes of a literal object: x coordinate, y coordinate, literal index, time step, identity (whether this literal is a box or player literal, for the future visualization).

Function AddPlayerLiteral creates a new literal object and stores it in the manager (an unordered map such that accessing members takes linear time) only if it cannot be accessed within the Player Literal Manager

by key {row, col, player index, time}, so does AddBoxLiteral. Hence, for any literal, only when a constraint clause containing it is created will this literal be instantiated and stored in manager.
MapInfo stores all map information by categories of "Players", "Boxes", "Walls", "Walkable", "Targets", implemented by an unordered map and information is extracted upon map loading.

III. Incremental SAT solving

We came up with two proposals. Initially, we attempt to implement incremental SAT by instantiating an ABC sat solver and out Solver only once, for each time frame, we only add newly generated clauses to the ABC sat solver, while our Solver will keep track of all prior instantiated literals, mimicking an expansion in search space, while initial state constraints will only be generated once, at the beginning of BMC search.

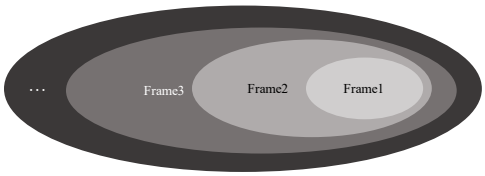

fig.6 search space expansion, where each circle represents constraints generated for that specific time frame

This approach brings about an issue, since for each time step, according to BMC algorithm, we will solve the constraints generated up to this time step, after solving and we found it UNSAT, whilst incrementing step limit by one, we will also need to remove the previous step limit's solved state constraints and replace them with solved state constraints at this particular time step limit. Since without this removal, it will guaranteed to be UNSAT due to induction. Hence, an assumption approach is adopted. For every set of newly generated solved state constraints, we inject an **assert** literal. For instance, if we have 2 targets and 2 boxes, the solved state generated up to time step limit *t* should be modified as:

$$\ldots\ldots(B_{\text{target1,box1,t-1}} + B_{\text{target1,box2,t-1}} + \mathbf{assert_{t-1}})$$
$$(B_{\text{target2,box1,t-1}} + B_{\text{target2,box2,t-1}} + \mathbf{assert_{t-1}})$$
$$(B_{\text{target1,box1,t}} + B_{\text{target1,box2,t}} + \mathbf{assert_t})$$
$$(B_{\text{target2,box1,t}} + B_{\text{target2,box2,t}} + \mathbf{assert_t})$$

This allows us to be able to 'deactivate' prior generated solved state clauses by asserting their corresponding **assert** variable to 1, 'activate' the prior generated solved state clauses by asserting their corresponding **assert** variable to 0.
The pseudo code is as follows:

```
instantiate Solver instance
instantiate Berkeley ABC sat solver pSat
Solver load map map
step = 1
while true:
    Solver set step limit to step
    Solver create the cnf constraints
    Solver write to pSat
    initialize a vector of assumptions
    for all steps s ranging from 1 to step:
        if s is current step:
            assert assert_s to false
        else    assert assert_s to true
    step++
    if (pSat solved SAT)
        print "BMC solution found!"
        access true literals
        demo steps in action
        return 0; //exit main function
```

Yet to our surprise, this approach has a worse performance in comparison to regenerating cnf constraints for each time frame (as the following pseudo code shows). We observe from running testcase1, our original incremental sat attempt results in an 10~11 seconds search time, while the modified version can accomplish the search within 6~7 seconds. Also, interestingly, despites its longer overall duration, when steps are smaller than approximately 20, the original approach offers a better performance. The modified version pseudo code is shown below:

step = 1
while **true**:
    instantiate ***Solver*** instance
    instantiate *Berkeley ABC sat solver* ***pSat***
    Solver set step limit to ***step***
    Solver load map ***map***
    Solver create the cnf constraints
    Solver write to ***pSat***
    step++
    if (***pSat*** solved SAT)
        print "BMC solution found!"
        access **true literals**
        demo steps in action
        return 0; //exit main function
    delete **pSat**

IV. Binary Search

To speed up the search, we integrate binary search to our BMC algorithm. Initially, it incrementally increases the step limit, using a SAT solver to check for a solution at each step, with larger increments for steps below 30 and smaller increments thereafter. Once a solution is found, it records the step limit and performs a binary search between the last checked step and the found step, iteratively narrowing down the range to determine the minimum step count that solves the puzzle. It successfully reduce the search time by half, as anticipated.

V. Experiment Results

One can compute the amount of literals easily:
# of walkable grids × (# of players + # of boxes) × (timesteps+1)
Number of literals grow linearly through timesteps.
Note that we have disregarded all Walls while encoding the constraints, reducing literal count. The following experiments are first conducted on our base case (map1.txt), which is obtained and modified from the online Sokoban game level 1, a 8*9 map with one single agent and 6 boxes on grid. As shown in *fig.5*.

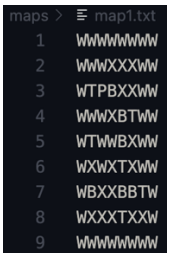

*fig.*5 testcase 1, where W for "Walls", B for "Boxes", P for "Players", and T for "Targets", X for "Unoccupied area".

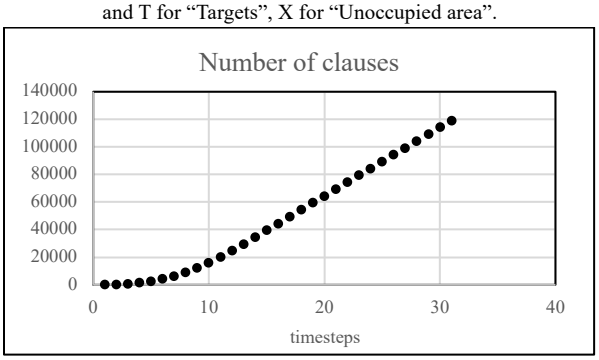

*fig.*6 growth of clauses through time

*fig.*6 suggests another potential faster approach to solving Sokoban Puzzle. Since number of clauses grows slower at small timesteps, decomposing a complex instance into multiple sub-problems may help reduce computation overhead in an SAT solver.
*fig.*7 shows how clauses and SAT solver (using miniSAT) made decisions grow through time with larger time steps.

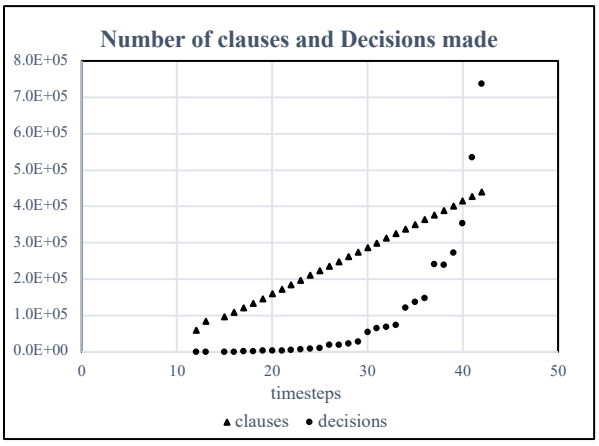

*fig.*7 growth of clauses and decisions made by solver through time for larger timesteps

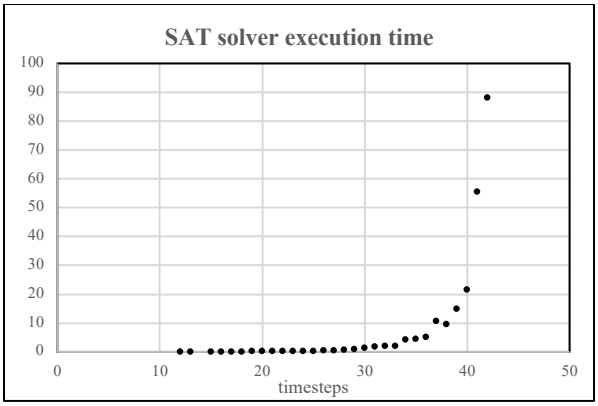

fig.8 SAT solver execution time to timesteps

Decisions made by SAT solver grows exponentially as BMC unfolds. This is inevitable in all three of our means. This suggests that SAT solvers might not be the best solution for Sokoban solving.

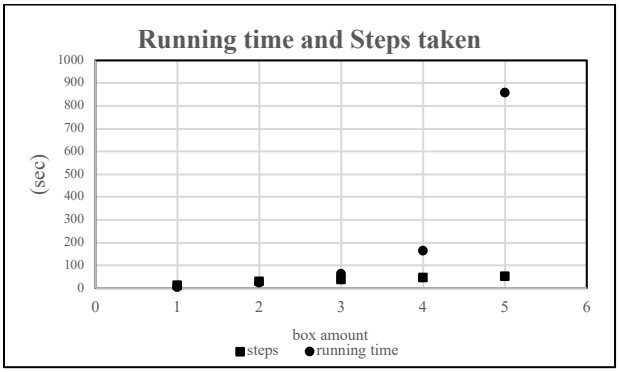

fig.9 Running time comparing to steps taken with respect to different box amount (modified from map5)

We can see that even though steps taken doesn't vary that much as box amount grows, running time grows substantially.

Here, we showcase the execution time comparisons among incremental SAT (ISAT), non-incremental SAT (NISAT), and non-incremental SAT with binary search (NISAT+BS).

map(min. step, box#, player#)	ISAT	NISAT	NISAT+BS
map1 (31, 6, 1)	10	6	4
map1MA (16, 6, 2)	8	6	4
map5 (51, 5, 1)	X	965	629
map10 (23, 2, 1)	0	0	0
map11 (31, 2, 1)	2	1	0
map12 (37, 3, 1)	10	3	1
map13 (70, 4, 1)	341	149	67

table 1 BMC search duration comparisons

VI. Appendix

fig.10 Screenshot of the intermediate step for testcase 1 (map1.txt)

fig.11 Screenshot of the solved state for testcase 1 (map1.txt)

fig.12 Screenshot of the intermediate step for multi-player version of testcase 1 (map1MA.txt)

fig.13 Screenshot of the solved state for multi-player version of testcase 1 (map1MA.txt)

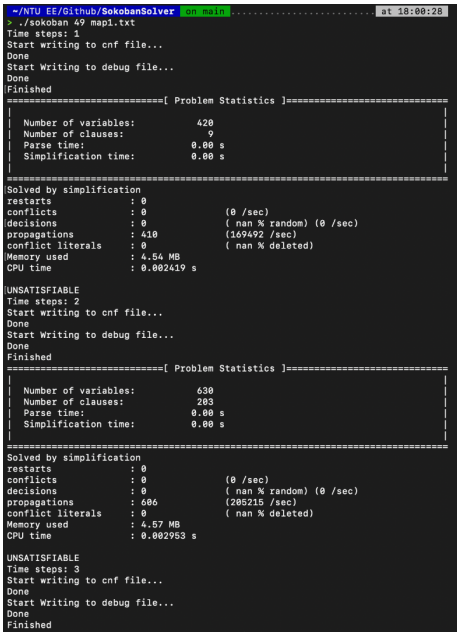

fig.14 it is observable that the constraints are quite complete (to some extent) since for the first 12 steps for testcase1, the solution is found by simplification (0 restarts, 0 conflicts)