Efficient Planning for Sokoban with Answer Set Programming - A Case Study

Thomas Verweyen

August 20, 2018

Summary

Abstract

Sokoban approach with ASP planning → PlainEnc, IncEnc, Optimization (?)

Introduction

What do i do? I'm working on the Sokoban problem and trying to do my best to find a fast and easy solution to it. I'm not working on any level but just the ones meeting some criteria. I'm picking qualitative and quantitative characteristics and deciding on some values to examine. My encodings are supposed to work better on some of the instances and worse on others. What rules describe those fluctuations? What assumptions can be made? Is there a trend when comparing runtimes to my qualitative and quantitative characteristics? E.g. you would expect the runtime to go up when the instance's size grows or when the amount of boxes in the level grows. In this thesis I will work on optimizing ASP encodings for the game Sokoban towards different instances. First Ill briefly explain what Sokoban is and how it is played, then Ill give a short introduction on comparable/similar/other scientific studies concerning sokoban. Then Ill go over my work, ... and finally evaluation and Discussion of Results.

What is Sokoban

Background

Sokoban is a logistic planning puzzle game from Japan. Sokoban was apparently created by Hiroyuki Imabayashi in 1981, and published by *Thinking Rabbit* in 1982. Sokoban means warehouse person in japanese. The game has very simple rules but shows to be very complex for humans and for AI. This makes Sokoban a very interesting subject for research regarding planning problems.

Rules

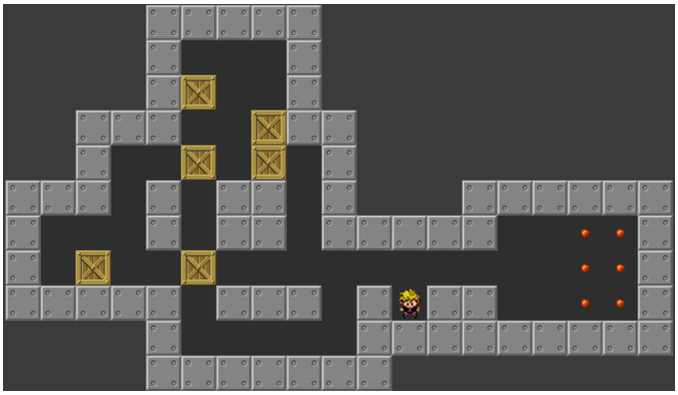

Figure 2.1: First of the Classical Sokoban Levels

Figure 2.1 illustrates the first Sokoban level of the original set of levels. The game is played on a board that can be modelled as a two-dimensional grid. The squares can either be floor or wall squares. Some floor squares are marked as goals and the same number of floor squares contain boxes. The player pushes boxes around this modelled warehouse, trying to cover the marked goal fields (targets) with one of the boxes. For that the player can move one step at a time into one of the four directions *up*, *down*, *left* and *right*. When the player moves onto a box's position the box is pushed one field into the direction of the move.

There can never be multiple boxes on one field, neither can there be a box where there is no field. A *solution* to a level is a sequence of moves that leaves a box on each target. In many of the levels created to this date, the warehouse has a lot of small passageways and very intricate designs. In this thesis I will look at more open level designs, e.g. rectangular levels. Sokoban is a very difficult planning problem for multiple reasons. There are a lot of possibilities to create a deadlock (a gamestate from which no solution can be found), the branching factor is large and the optimal solution can be very long [5].

State of the Art

Complexity: Gordon Wilfong published in 1991 that all motion planning problems with moving obstacles are NP-hard [12]. He proved this by reducing 3-Sat to such a problem. The question whether Sokoban is PSPACE-complete remained open until 1997, when Joseph C. Culberson published his paper "Sokoban is PSPACE-complete" [9]. He found a transformation algorithm to transform an LBA into a level of Sokoban, with the solution to the level having a number of steps in $\Theta(|w|+t(|w|))$, where w is the input and $t(|w|)$ is the amount of transitions the LBA made.

Approaches: There are loads of different researches regarding approaches to Sokoban. I decided which to show. Botea et al. found out that heuristics, while significantly improving chess AI, do not seem to help for planning the soko puzzle [5]. Planning would be a better approach. So to get there and create good plans like humans would, the solver will start meta-reasoning by recognizing rooms and tunnels. Then using those rooms and tunnels to create a plan where first a way is found to bring the crates from the room where theyre at to the target room. Then another plan is made to organize crates in a room.

Macromoves: Macromoves are a planning strategy that is so successful it is implemented in several good Sokoban solvers. The best example would be Rolling Stone, an excellent Sokoban solver, with a lot of optimizations implemented. One of the optimizations is the possibility to create macromoves for tunnels to reduce all states in the tunnel to one or two, reducing the search space severely.

Deadlock Detection: Deadlocks are states from which no solution can be found. A trivial example for a deadlock (see Figure 3.1) is a box pushed into a corner with no target under it. The box cannot be pulled, so it cannot be recovered from the corner, but to find a solution all boxes have to be covering a target. Detecting deadlocks the moment they appear is a very powerful optimization, for the reason that huge parts of the search tree can be eliminated early on, without further looking into them. The problem with this strategy is that deadlocks can be very hard to detect, e.g. when they consist of multiple boxes blocking each other in some kind of tunnelsystem. Deadlocks are easier to detect and remove from the search tree when the domain is restricted. When

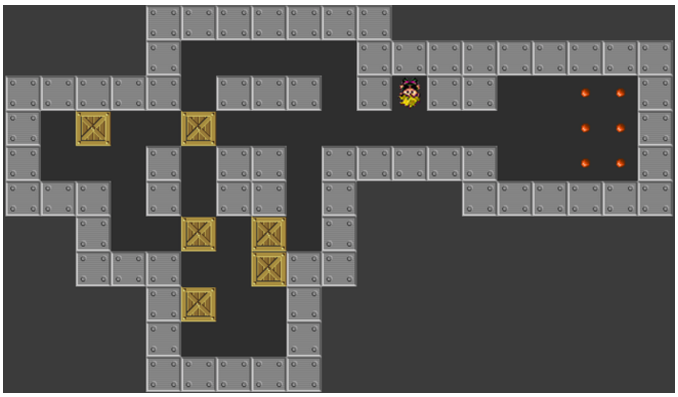

Figure 3.1: Deadlock example

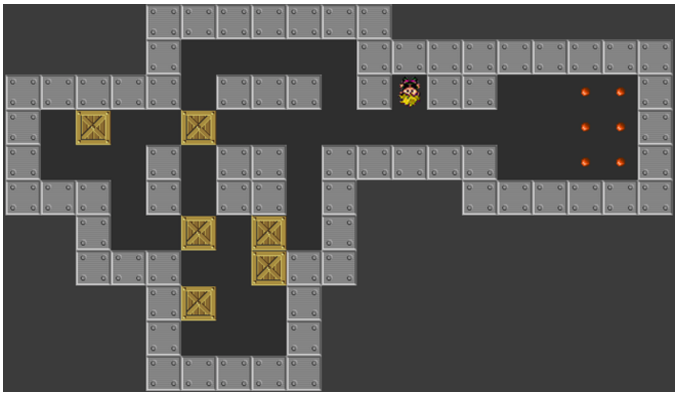

Figure 3.2: Deadlock types

looking at levels with a rectangular shape there are four common deadlocks, cp. Fig. 3.2.

Relevance Cutting: A basic AI to solve a planning problem like Sokoban will look at all possible ways to move around the level, but some of those are dubious but legal or just plain irrelevant. In a lot of cases those dubious ways do not lead to an optimal solution, but of course a problem can be constructed where the optimal solution can only be found when looking at those dubious cases. So Junghanns and Schaeffer tried to teach their AI meta-reasoning to give it the ability to know when to look at those dubious ways and when to ignore them. This they did by creating an influence measurement and only taking moves that are influenced by prior moves, to prevent actions that are disembodied from all prior moves (and seem irrelevant).

My Work

In my research I focussed on simple approaches in very specific domains, to be able to compare the efficiency of different optimizations in those domains. The domains I researched are rectangular halls and slightly more crooked levels, for reference see Figure **??**. Those maps had to be selfmade, so I had to find the horizon myself (by using incremental encodings). The basic optimizations are some approaches to deadlock detection, basic move-evaluation (don't take moves back), splitting x and y to reduce groundable variables, and i tried for lexordering the ways to reduce multiple grounded ways from A to B.

Encodings

Fact Format

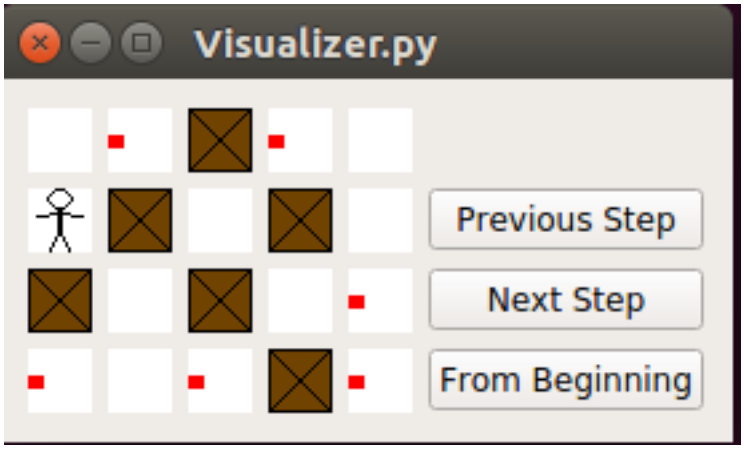

Figure 4.1: Instance 5x4 with 6 boxes

```
1 #const horizon = 23.
2 #const dimX = 5.
3 #const dimY = 4.
4 init(field(1..5,1..4)).
```

```
 5 init(at(0,1,2)).
 6 init(at(1,1,3)).
 7 init(at(2,2,2)).
 8 init(at(3,3,1)).
 9 init(at(4,3,3)).
10 init(at(5,4,2)).
11 init(at(6,4,4)).
12 init(target(1,4)).
13 init(target(2,1)).
14 init(target(3,4)).
15 init(target(4,1)).
16 init(target(5,3)).
17 init(target(5,4)).
```

In the code

Plain Encoding

```
 1 time(0..horizon).
 2
 3 dir(1,0;0,1;-1,0;0,-1).
 4 at(N,X,Y,0) :- init(at(N,X,Y)).
 5
 6 % pick one direction to move in
 7 1 { move(DX,DY,T) : dir(DX,DY) } 1 :- time(T).
 8 % move the player
 9 at(0,X+DX,Y+DY,T) :- at(0,X,Y,T-1), move(DX,DY,T), time(T).
10 % move box, the player collides with
11 at(N,X+DX,Y+DY,T) :- at(N,X,Y,T-1), at(0,X,Y,T), move(DX,DY,T),
12                       time(T), N>0.
13 % leave boxes that aren't touched at their place with t+1
14 at(N,X,Y,T) :- at(N,X,Y,T-1), not at(0,X,Y,T), N>0, time(T).
15 % there can't be boxes, players, where there is no field
16 :- not init(field(X,Y)), at(_,X,Y,_).
17 % there can't be two boxes/players, at the same spot
18 :- at(N,X,Y,T), at(M,X,Y,T), N>M, time(T).
19
20 uncvrd(X,Y,T) :- init(target(X,Y)), not at(_,X,Y,T), time(T).
21 goal(X,Y,T) :- init(target(X,Y)), not uncvrd(X,Y,T),
22                 not at(0,X,Y,T), time(T).
23 :- not goal(X,Y,horizon), init(target(X,Y)).
24
25 #show at/4.
```

The first try to construct a naive encoding to solve sokoban-instances yielded a solution that works in a common way in ASP planning. In line 4 we initialize the information from the level instance with the time 0. In line 7 we state that there is one move at minimum and at maximum for each timestep. This makes the ASP planner ground all sequences of moves, pick one for every timestep and infer every successive state from the moves it picks. In lines 9-13 we calculate the successor state from the move picked in line 7. Line 9 calculates the new player-position, line 11 calculates the new position of the box in the players way and line 13 calculates the new position of boxes not touched by the player.

Additionally we need some integrity constraints to prevent illegal states. The integrity constraint in line 15 ensures that at no time a box or player has a position outside of the playing field. The integrity constraint in line 17 ensures that at no time boxes or the player share one field. In the lines 19-21 we define our win-condition to be that all targets in the instance have to be covered by boxes. The Plain Encoding has the predicates, time/1, dir/2, at/4, move/3, uncovered/3, goal/3 The encoding works in a common way in ASP planning, by guessing the sequence of actions and inferring the successor states by effect axioms. There is a time/1 predicate for every timestep and a at/4 predicate for every box and the player at every timestep.

Incremental Encoding

```
1 #include <incmode>.
2
3 #program base.
4 dir( 1,0 ; 0,1 ; -1,0 ; 0,-1 ).
5 at(N,X,Y,0) :- init(at(N,X,Y)).
6
7 #program step(t).
8 % pick one direction to move in
9 1 { move(DX,DY,t) : dir(DX,DY) } 1.
10 % move the player
11 at(0,X+DX,Y+DY,t) :- at(0,X,Y,t-1), move(DX,DY,t), init(field(X+DX,Y+DY)).
12 % move box, the player collides with
13 at(N,X+DX,Y+DY,t) :- at(N,X,Y,t-1), at(0,X,Y,t), move(DX,DY,t), N>0.
14 % leave boxes that aren't touched at their place with t+1
15 at(N,X,Y,t) :- at(N,X,Y,t-1), not at(0,X,Y,t), N>0.
16 % there can't be boxes, players, where there is no field
17 :- at(_,X,Y,_), not init(field(X,Y)).
18 % there can't be two boxes/players, at the same spot
19 :- at(N,X,Y,t), at(M,X,Y,t), N>M.
20
21 #program check(t).
22 #external query(t).
23 uncovered(X,Y,t) :- init(target(X,Y)), not at(_,X,Y,t), query(t).
24 goal(X,Y,t) :- init(target(X,Y)), not uncovered(X,Y,t), not at(0,X,Y,t), query(t).
25 :- not goal(X,Y,t), init(target(X,Y)), query(t).
26
27 #show at/4.
```

The incremental encoding is similar to the plain encoding. The difference is the partition of the encoding into three parts: the #program base in lines 3-5 which is independent of the step parameter t, the #program step in lines 7-19 which is the cumulative part, collecting knowledge and the #program check in lines 21-25 which is specific for each value of t (STEP ALSO IS THOUGH). This allows for gradually processing each time step and accumulating knowledge that persists even if the current step is not satisfiable.

Advanced Encoding

```
time(0..horizon).

dir(1,0;0,1;-1,0;0,-1).
at(N,X,Y,0) :- init(at(N,X,Y)).

forbidden(X,Y) :- init(field(X,Y)), not init(target(X,Y)),
3 #sum {3 : not init(field(X+DX,Y+DY)), not init(field(X+DY,Y+DX)), dir(DX,DY);
 3 : not init(field(X+DX,Y+DY)), not init(field(X-DY,Y-DX)), dir(DX,DY);
 1,1 : forbidden(X+DX,Y+DY), dir(DX,DY); 1,2 : forbidden(X+DX,Y+DY), dir(DX,DY), forbidden(X-D
 1,3 : not init(field(X+DX,Y+DY)), dir(DX,DY);
 1,4 : not init(field(X+DX,Y+DY)), not init(field(X-DX,Y-DY)), dir(DX,DY)}.

% pick one direction to move in
1 { move(DX,DY,T) : dir(DX,DY) } 1 :- time(T).
% move the player
at(0,X+DX,Y+DY,T) :- at(0,X,Y,T-1), move(DX,DY,T), time(T).
% move box, the player collides with
at(N+1,X+DX,Y+DY,T) :- at(N+1,X,Y,T-1), at(0,X-DX,Y-DY,t), at(0,X,Y,T), move(DX,DY,T), time(T)
% leave boxes that aren't touched at their place with t+1
at(N,X,Y,T) :- at(N,X,Y,T-1), not at(0,X,Y,T), N>0, time(T).
% don't take a move back on the very next step
:- move(DX,DY,T), move(-DX,-DY,T-1), at(0,X,Y,T), not at(_,X-DX,Y-DY,T-1), time(T).
% there can't be boxes, players, where there is no field
:- not init(field(X,Y)), at(_,X,Y,_).
% there can't be two boxes/players, at the same spot
:- at(N,X,Y,T), at(M,X,Y,T), N>M, time(T).
% don't push a box onto a forbidden field
:- at(N,X,Y,T), forbidden(X,Y), N>0, T>0, not at(N,X,Y,T-1).
dirPos(1;-1).
:- 3 #sum {1,1 : at(A,X+DX,Y,T), A>0 ; 1,1 : not init(field(X+DX,Y)) ;
           1,2 : at(B,X,Y+DY,T), B>0 ; 1,2 : not init(field(X,Y+DY)) ;
           1,3 : at(C,X+DX,Y+DY,T), C>0 ; 1,2 : not init(field(X+DX,Y+DY)) },
           dirPos(DX), dirPos(DY), at(N,X,Y,T), N>0, time(T).

uncovered(X,Y,T) :- init(target(X,Y)), not at(_,X,Y,T), time(T).
goal(X,Y,T) :- init(target(X,Y)), not uncovered(X,Y,T), not at(0,X,Y,T), time(T).
:- not goal(X,Y,horizon), init(target(X,Y)).

#show at/4.
```

Evaluation

Experiments and Results

Discussion of Results

compare chapter State of the Art with chapter My Work [1]

Bibliography

[1] Vladimir Lifschitz, *Answer Set Planning*, ICLP, 1999.

[2] Vladimir Lifschitz, *Answer set programming and plan generation*, Artificial Intelligence, 2002, Vol. 138, p.39-54.

[3] Vladimir Lifschitz, *What is Answer Set Programming?*, AAAI, 2008.

[4] Martin Gebser, Holger Jost, Roland Kaminski, Philipp Obermeier, Orkunt Sabuncu, Torsten Schaub, Marius Thomas Lindauer, *Ricochet Robots: A Transverse ASP Benchmark*, LPMNR, 2013.

[5] Adi Botea, Martin Mller, Jonathan Schaeffer, *Using Abstraction for Planning in Sokoban*, Computers and Games, 2002.

[6] Alan M. Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel, Toby Walsh, *Propagation algorithms for lexicographic ordering constraints*, Artificial Intelligence, 2006, Vol. 170, p.803-834.

[7] Dorit Dor, Uri Zwick, *SOKOBAN and other motion planning problems*, Computational Geometry, 1999, Vol. 13, p.215-228.

[8] Nils Christian Froleyks and Tomáš Balyo, *Using an Algorithm Portfolio to Solve Sokoban*, SOCS, 2017.

[9] Joseph C. Culberson, *Sokoban is PSPACE-complete*, 1997.

[10] Andreas Junghanns, Jonathan Schaeffer, *Sokoban: Improving the Search with Relevance Cuts*, Journal of Theoretical Computing Science, 1999, Vol. 252, p.1-2.

[11] Brahim Hnich, Zeynep Kiziltan, Toby Walsh, *Combining Symmetry Breaking with Other Constraints: Lexicographic Ordering with Sums*, ISAIM, 2004.

[12] Gordon Wilfong, *Motion planning in the presence of movable obstacles*, Annals of Mathematics and Artificial Intelligence, 1991.