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

Thomas Verweyen

September 12, 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

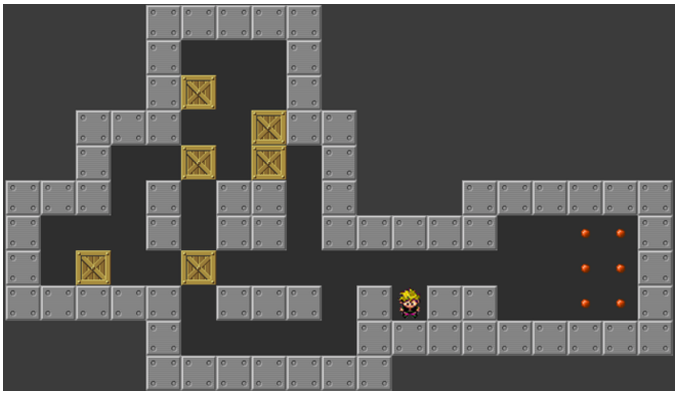

Figure 2.1: First of the Classical Sokoban Levels

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].

Related Work

Complexity: Gordon Wilfong published in 1991 that all motion planning problems with moving obstacles are NP-hard [13]. 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

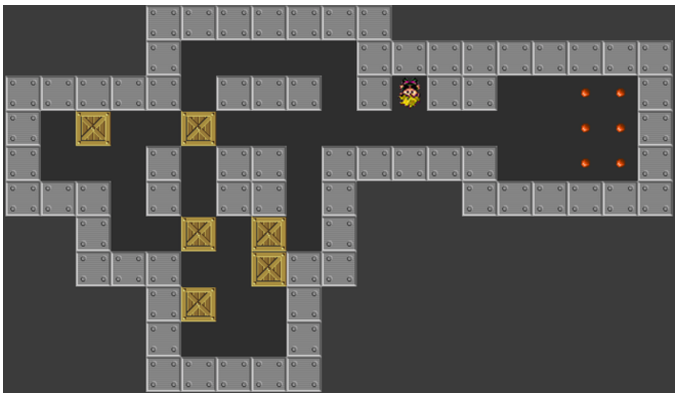

Figure 3.1: Deadlock example

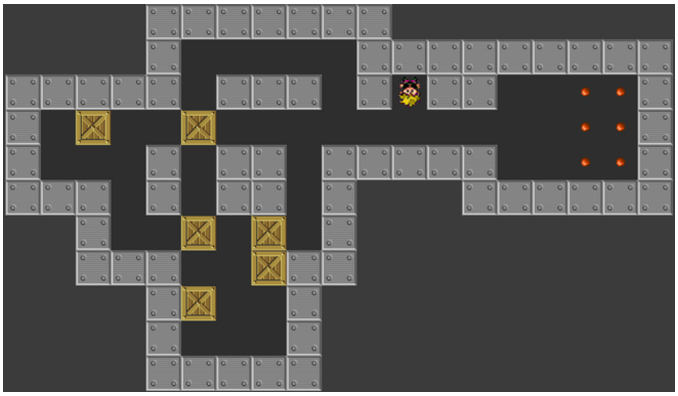

Figure 3.2: Deadlock types

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. This is used in the popular Sokoban Solver Rolling Stone which has an excellent record in approaching difficult Sokoban

problems.

My Work

In my research I focused 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 **??**. There are few common Sokoban domains that only look at levels with these qualities, so I designed a problemset myself. This meant that I had to find the horizon aswell (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

In the following, I will present the fact fromat and the first encodings I used to approach Sokoban solving in the input language of the ASP grounder gringo 3 [12].

Fact Format

```
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)).
```

Listing 4.1: Fact Format of my third level

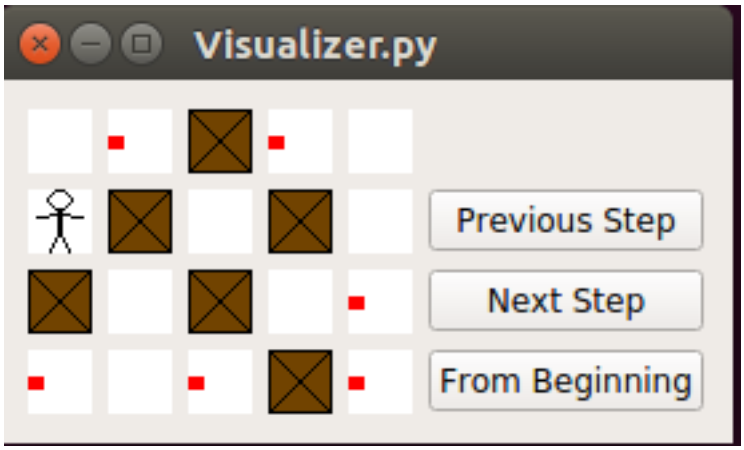

Figure 4.1: Instance 5x4 with 6 boxes

For illustration consider the level shown in Figure 4.1. The corresponding fact representation can be seen in Listing 4.1. The board size is fixed by the constants $dimX$ and $dimY$; the origin (1,1) is in the top left corner. The positions of player and boxes are stored in the at/3 predicate. The first argument represents the identifier. A zero stands for the player, while all boxes have numbers greater than zero. The second and third argument are the X and Y coordinates, signifying the initial position of the object. For instance init(at(0,1,2)) means the player is at the initial position (1,2). Because I am not exclusively looking at rectangular levels I decided to store all legal fields (meaning fields where a box or player can be placed) by using the predicate field/2. In rectangular levels the legal fields can be defined in one line. Here init(field(1..5,1..4)) (l.4) means the crossproduct of all the numbers in the pools (i.e. (1,1),..,(5,1),(1,2),..,(5,4)). The goal of the level is given in the target/2 predicate. Each coordinate given by the two arguments of a target/2 instance has to be covered by a box in the final gamestate to solve this level. Finally the constant horizon stores the information how many steps are needed to successfully satisfy the level. This horizon was determined by using the incremental encoding in section 4.1.3.

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 1 { move(DX,DY,T) : dir(DX,DY) } 1 :- time(T), T>0.
7
8 at(0,X+DX,Y+DY,T) :- at(0,X,Y,T-1), move(DX,DY,T), time(T).
```

```
 9 at(N,X+DX,Y+DY,T) :- at(N,X,Y,T-1), at(0,X,Y,T), move(DX,DY,T),
10                     time(T), N>0.
11 at(N,X,Y,T) :- at(N,X,Y,T-1), not at(0,X,Y,T), N>0, time(T).
12
13 :- not init(field(X,Y)), at(_,X,Y,_).
14 :- at(N,X,Y,T), at(M,X,Y,T), N>M, time(T).
15
16 box(X,Y) :- at(N,X,Y,horizon), N>0.
17 :- init(target(X,Y)), not box(X,Y).
18
19 #show at/4.
```

Listing 4.2: Plain Encoding

The plain encoding seen in listing 4.2 uses a strategy common in ASP planning. A choice rule picks the moves (l.6) and the succeeding states are derived from those by using effect and frame axioms (ll.8-11). In the first few lines basic information is created, namely a $time/1$ predicate for each timestep (l.1), a $dir/2$ predicate for each of the four possible directions to move in (l.3) and an $at/4$ predicate to set the initial positions of objects as their position at time T=0. Additionally the encoding needs integrity constraints to ensure that the plans found are legal. There is one in line 13 that ensures that no object can move off the playable area and another in line 14 that prevents two object from being at the some position at the same time. In the lines 16 and 17 the goal condition is defined. The $box/2$ predicate is only created at the last timestep to specify each box position (X,Y). The constraint in line 17 stipulates that all targets have to have a corresponding box predicate with the same coordinates. The added directive **#show at/4.** in line 21 allows for using clingos output as input to my Visualizer *SokoViz*. This encoding allows to check for a plan of length equal to the horizon constant. This is of course not always optimal, e.g. when one wants to know a shortest plan to solve a level. To achieve an encoding that is able to, I will take advantage of incremental solving. *clingo* provides an easy to use built-in incremental solving and grounding mode. An encoding using the inc-mode can be seen in listing **??**. It is very similar to the plain encoding from listing 4.2 in having the same structure of a choice rule picking the moves and effect and frame aioms finding the successive state. This encoding is pslit into three parts though, which is critical for using the inc-mode. The #program base part has only information that is created in the very beginning and will be unchanged throughout the whole grounding and solving process. This infomation is called #static. The #program step(t) part is the part accumulating knowledge, holding rules that will be collected and used for each step t and all following steps. The last part #program check(t) holds volatile rules which will be checked once and discarded for the next timestep. Those rules are used to check for success and are no help to solving following steps. There is a minor change in the goal defining lines. In the plain encoding the horizon can be used to check efficiently only for the last timestep. In the incremental encoding I carry the time in the $box/3$ prediate to ensure that the constraint in line 21 only affects the target at time t.

Incremental Encoding

```
#include <incmode>.

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

#program step(t).
1 { move(DX,DY,t) : dir(DX,DY) } 1.

at(0,X+DX,Y+DY,t) :- at(0,X,Y,t-1), move(DX,DY,t).
at(N,X+DX,Y+DY,t) :- at(N,X,Y,t-1), at(0,X,Y,t),
                     move(DX,DY,t), N>0.
at(N,X,Y,t) :- at(N,X,Y,t-1), not at(0,X,Y,t), N>0.

:- at(_,X,Y,_), not init(field(X,Y)).
:- at(N,X,Y,t), at(M,X,Y,t), N>M.

#program check(t).
#external query(t).
box(X,Y,t) :- at(N,X,Y,t), N>0, query(t).
:- not box(X,Y,t), init(target(X,Y)), query(t).

#show at/4.
```

Listing 4.3: Incremental Encoding

Advanced Encoding

```

dirPos(1;-1).
:- 4 #sum {1,1 : at(A,X+DX,Y,t), not init(target(X+DX,Y)), A>0 ;
               1,1 : not init(field(X+DX,Y)) ;
           1,2 : at(B,X,Y+DY,t), not init(target(X,Y+DY)), B>0 ;
               1,2 : not init(field(X,Y+DY)) ;
           1,3 : at(C,X+DX,Y+DY,t), not init(target(X+DX,Y+DY)), C>0 ;
               1,3 : not init(field(X+DX,Y+DY)) ;
           1,4 : at(D,X,Y,t), not init(target(X,Y)), D>0 },
           dirPos(DX), dirPos(DY), init(field(X,Y)).
```

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] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Marius Lindauer, Max Ostrowski, Javier Romero, Torsten Schaub, Sven Thiele, *Potassco User Guide*, Second Edition, 2015.

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