A Tabled Prolog Program for Solving Sokoban

Neng-Fa Zhou    Agostino Dovier

Department of Computer and Information Science,
CUNY Brooklyn College & Graduate Center, USA

Department of Mathematics and Computer Science,
University of Udine, Italy

Pescara, September 2nd, 2011
Summary

1. Introduction
2. Sokoban as a Planning Problem
3. Tabling
4. Sokoban as a Prolog Program
5. Computational Results
6. Conclusions
Sokoban is a type of transport puzzle invented by Hiroyuki Imabayashi in 1980.
Published by the Japanese company Thinking Rabbit, Inc. in 1982.
Sokoban means “warehouse-keeper” (magazziniere) in Japanese.
Thinking Rabbit joined Square Co., Ltd.
Sokoban Rules
(from http://www.sokoban.jp/)
Sokoban @ work
The following sequence of 8 actions
Introduction

Macro-actions

The following sequence of 8 actions

is counted as a single (macro) action.
The genesis of this work

With Andrea Formisano and Enrico Pontelli we developed a CLP(FD) solver for the action description language $\mathcal{B}$ (and a compiler in ASP) [ICLP07–MG65]
The genesis of this work

- With Andrea Formisano and Enrico Pontelli we developed a CLP(FD) solver for the action description language $\mathcal{B}$ (and a compiler in ASP) [ICLP07–MG65]
- Neng-Fa Zhou is the main developer (actually, the father) of B-Prolog that includes a fast solver of constraints on finite domains
The genesis of this work

- With Andrea Formisano and Enrico Pontelli we developed a CLP(FD) solver for the action description language $\mathcal{B}$ (and a compiler in ASP) [ICLP07–MG65]
- Neng-Fa Zhou is the main developer (actually, the father) of B-Prolog that includes a fast solver of constraints on finite domains
- In the 2009 ASP competition we wrote with him some $\mathcal{B}$ domains that, once interpreted with the B-Prolog solver behaved very well (in particular peg-solitaire).
The genesis of this work

- With Andrea Formisano and Enrico Pontelli we developed a CLP(FD) solver for the action description language $B$ (and a compiler in ASP) [ICLP07–MG65]
- Neng-Fa Zhou is the main developer (actually, the father) of B-Prolog that includes a fast solver of constraints on finite domains
- In the 2009 ASP competition we wrote with him some $B$ domains that, once interpreted with the B-Prolog solver behaved very well (in particular peg-solitaire).
- Neng-Fa asked us to do the same for the 2011 competition.
The genesis of this work

With Andrea Formisano and Enrico Pontelli we developed a CLP(FD) solver for the action description language $\mathcal{B}$ (and a compiler in ASP) [ICLP07–MG65]

Neng-Fa Zhou is the main developer (actually, the father) of B-Prolog that includes a fast solver of constraints on finite domains.

In the 2009 ASP competition we wrote with him some $\mathcal{B}$ domains that, once interpreted with the B-Prolog solver behaved very well (in particular peg-solitaire).

Neng-Fa asked us to do the same for the 2011 competition.

This approach for Sokoban was un-successful, but this forced us to look for another declarative approach.
location(1). ... location(17).
Sokoban as a Planning Problem

Representation

location(1). ... location(17).
step(1,right,2). step(10,right,11). step(11,right,15).
step(2,left,1). step(11,left,10). step(15,left,11).
step(1,down,3). step(2,down,4).
step(3,up,1). step(4,up,2).
...
Representation
Encoding in B

Fluents

```prolog
fluent(free(L)) :- location(L).
fluent(box_in(L)) :- location(L).
fluent(sokoban_in(L)) :- location(L).
```
**Encoding in B**

**Fluents**

\[
\text{fluent} \left( \text{free}(L) \right) :-
\quad \text{location}(L).
\]

\[
\text{fluent} \left( \text{box}_\text{in}(L) \right) :-
\quad \text{location}(L).
\]

\[
\text{fluent} \left( \text{sokoban}_\text{in}(L) \right) :-
\quad \text{location}(L).
\]

\[
\text{fluent} \left( \text{reachable}(A) \right) :-
\quad \text{location}(A).
\]

box_in(9).

sokoban_in(4).

box_in(9).

sokoban_in(4).

goal(box_in(3)).
Input from ASP competition

right(col4row2,col5row2). right(col5row2,col6row2).
....
Input from ASP competition

right(col4row2, col5row2). right(col5row2, col6row2).
....

box(col8row2). box(col3row4).
sokoban(col4row4).
Input from ASP competition

right(col4row2,col5row2). right(col5row2,col6row2).
...

box(col8row2). box(col3row4).
sokoban(col4row4).

solution(col3row2). solution(col2row2).
action(push(From,D,To)) :-
  location(From), location(To), neq(From,To),
  direction(D), % D = left, right, up, down
  step(_Sokoban,D,From),
  straight_connection(From,To,D,_).
**Sokoban as a Planning Problem**

**Encoding in B**

**Actions (help from Andrea Formisano)**

```
action(push(From,D,To)) :-
  location(From), location(To), neq(From,To),
  direction(D), % D = left, right, up, down
  step(_Sokoban,D,From),
  straight_connection(From,To,D,_).

executable(push(From,D,To),[sokoban_in(S0),reachable(S1),
  box_in(From) | Free_LIST ]) :-
  action(push(From,D,To)),
  location(S0),location(S1),
  step(S1,D,From),
  straight_connection(From,To,D,[From|PATH]),
  empty_path(PATH, Free_LIST ).

empty_path([],[]).
empty_path([L|R],[free(L)|S]) :-
  empty_path(R,S).
```
### Encoding in B

#### Actions Effects

causes(push(From,D,To), box_in(To), []) :- action(push(From,D,To)).

causes(push(From,D,To), neg(box_in(From)), []) :- action(push(From,D,To)).

causes(push(From,D,To), sokoban_in(S), []) :-
   action(push(From,D,To)),
   location(S), step(S,D,To).

causes(push(From,D,To), free(S), [sokoban_in(S)]) :-
   action(push(From,D,To)),
   location(S), \+ step(S,D,To).

causes(push(From,D,To), free(From), []) :-
   action(push(From,D,To)),
   \+ step(From,D,To).
caused([free(L)],neg(box_in(L))) :- location(L).

caused([free(L)],neg(sokoban_in(L))) :- location(L).

caused([sokoban_in(L)],neg(free(L))) :- location(L).

caused([sokoban_in(L)],neg(box(L))) :- location(L).

caused([sokoban_in(L1)],neg(sokoban_in(L2))) :- location(L1), location(L2), neq(L1,L2).

caused([box_in(L)],neg(free(L))) :- location(L).

caused([box_in(L)],neg(sokoban_in(L))) :- location(L).
caused([sokoban_in(A)], reachable(A)) :-
    location(A).

caused([reachable(B), free(C)], reachable(C)) :-
    location(B), location(C),
    neq(B, C),
    step(B, D, C), direction(D).
The results

- This B encoding, compiled in ASP and run using clingo run rather fast on the proposed examples.
- We later discovered that the running time is comparable to that of the direct ASP solution of the Sokoban (also run using clingo).
The results

- This B encoding, compiled in ASP and run using clingo run rather fast on the proposed examples.
- We later discovered that the running time is comparable to that of the direct ASP solution of the Sokoban (also run using clingo).
- Unfortunately, the same did not hold for the CLP(FD) encoding (even if speed was not the real problem) which was our overall goal.
Static Causal Laws as a (simple) constraint

\[ \text{push}(9, \text{left}, 3) \text{ is forbidden (12 is not reachable from 4).} \]

\[
\text{caused}([\text{sokoban_in}(A)], \text{reachable}(A)) :- \\
\quad \text{location}(A).
\]

\[
\text{caused}([\text{reachable}(B), \text{free}(C)], \text{reachable}(C)) :- \\
\quad \text{location}(B), \text{location}(C), \\
\quad \text{neq}(B,C), \\
\quad \text{step}(B,D,C), \text{direction}(D).
\]
Static Causal Laws as a (simple) constraint

push(9, left, 3) is forbidden (12 is not reachable from 4).

caused([sokoban_in(A)], reachable(A)) :-
  location(A).

caused([reachable(B), free(C)], reachable(C)) :-
  location(B), location(C),
  neq(B, C),
  step(B, D, C), direction(D).
Static Causal Laws as a (simple) constraint

push(9, left, 3) is forbidden (12 is not reachable from 4).

caused([sokoban_in(A)], reachable(A)) :-
  location(A).

cauused([reachable(B), free(C)], reachable(C)) :-
  location(B), location(C),
  neq(B, C),
  step(B, D, C), direction(D).
Static Causal Laws as a (simple) constraint

caus**e**d([reachable(B), free(C)], reachable(C)) :-
  location(B), location(C), neq(B, C),
  step(B, D, C), direction(D).

reachable(11) ∧ free(12) → reachable(12).
reachable(12) ∧ free(11) → reachable(11).
caused([reachable(B), free(C)], reachable(C)) :- location(B), location(C), neq(B, C), step(B, D, C), direction(D).

reachable(11) ∧ free(12) → reachable(12).
reachable(12) ∧ free(11) → reachable(11).
reachable(11) and reachable(12) both true is a solution (of the constraint).
Static Causal Laws as a (simple) constraint

caused([reachable(B), free(C)], reachable(C)) :-
    location(B), location(C), neq(B, C),
    step(B, D, C), direction(D).

reachable(11) ∧ free(12) → reachable(12).
reachable(12) ∧ free(11) → reachable(11).
reachable(11) and reachable(12) both true is a solution (of the constraint).
This (loop) problem is correctly addressed by the ASP encoding.
caused([reachable(B), free(C)], reachable(C)) :-
    location(B), location(C), neq(B, C),
    step(B, D, C), direction(D).

reachable(11) \land \overline{free(12)} \rightarrow reachable(12).
reachable(12) \land \overline{free(11)} \rightarrow reachable(11).
reachable(11) \text{ and } reachable(12) \text{ both true is a solution (of the constraint).}

This (loop) problem is correctly addressed by the ASP encoding. A correct constraint encoding would introduce too many constraints (making the CLP(FD) interpreter too slow).
A solution (in $B^{MV}$)

- Add the multivalued fluent $\text{occupation}(L)$ with three values: 0 for free, 1 for sokoban, and 2 for a box.
- For each location $L$, let $L_1, \ldots, L_{N_L}$ be the neighbor locations of $L$.
- $M$ is a “big” number.
- Post the constraints:

$$
\begin{align*}
\text{occupation}(L) = 1 & \rightarrow \text{reachable}(L) = 0 \\
\text{occupation}(L) = 2 & \rightarrow \text{reachable}(L) = M \\
\text{reachable}(L) & = \min\{\text{reachable}(L_1) + 1, \ldots, \\
& \quad \text{reachable}(L_{N_L}) + 1, M\}
\end{align*}
$$
A solution

\[
\text{reachable}(4) = 0 \\
\text{reachable}(2) = \text{reachable}(3) = \\
\text{reachable}(6) = \text{reachable}(7) = 1 \\
\text{reachable}(1) = \text{reachable}(5) = \text{reachable}(8) = 2 \\
\text{reachable}(9) = \text{reachable}(10) = \cdots = \text{reachable}(17) = 20
\]
A solution

reachable(4) = 0
reachable(2) = reachable(3) =
reachable(6) = reachable(7) = 1
reachable(1) = reachable(5) = reachable(8) = 2
reachable(9) = reachable(10) = ⋯ = reachable(17) = 20

Unfortunately, this solution generates slow code.
Tabling has become a well-known and useful feature of many Prolog systems.

The idea of tabling is to memorize answers to tabled subgoals and use the answers to resolve subsequent variant or subsumed subgoals.

This idea resembles the dynamic programming idea of reusing solutions to overlapping sub-problems.
Tabling has become a well-known and useful feature of many Prolog systems.

The idea of tabling is to memorize answers to tabled subgoals and use the answers to resolve subsequent variant or subsumed subgoals.

This idea resembles the dynamic programming idea of reusing solutions to overlapping sub-problems.

B-Prolog is a tabled Prolog system based on linear tabling, allows variant subgoals to share answers, and uses the local (lazy) strategy to return answers.
Tabling in B-Prolog

Fibonacci numbers

:-table fib/2.
fib(0, 1).
fib(1, 1).
fib(N, F):-
    N>1, N1 is N-1, N2 is N-2,
    fib(N1, F1), fib(N2, F2),
    F is F1+F2.
Tabling in B-Prolog

Fibonacci numbers

:-table fib/2.
fib(0, 1).
fib(1, 1).
fib(N, F):-  
    N>1, N1 is N-1, N2 is N-2,  
    fib(N1, F1), fib(N2, F2),  
    F is F1+F2.

Without tabling, the subgoal \texttt{fib}(N, X) would spawn \(2^N\) subgoals, many of which are variants.
Tabling in B-Prolog

Fibonacci numbers

:-table fib/2.
fib(0, 1).
fib(1, 1).
fib(N, F):-
    N>1, N1 is N-1, N2 is N-2,
    fib(N1, F1),   fib(N2, F2),
    F is F1+F2.

Without tabling, the subgoal \( \text{fib}(N, X) \) would spawn \( 2^N \) subgoals, many of which are variants.

With tabling, the time complexity drops to linear since the same variant subgoal is resolved only once.
Tabling in B-Prolog

Modes

- B-Prolog allows **Mode-directed tabling**
  
  ```prolog
  :-table p(M1, ..., Mn):C.
  ```

- **C** (optional), the *cardinality limit*, bounds the number of answers to be tabled for *p*.

- Each *Mi* is a **mode**:
  - **+** (input $\rightarrow$ usually ground)
  - **−** (output $\rightarrow$ usually a variable)
  - **min** or **max** (optimized $\rightarrow$ output)

- Only one argument in a tabled predicate can have the mode **min** or **max**.
Tabling in B-Prolog

Shortest path ($sp$) in a weighted directed graph $X \xrightarrow{W} Y$

```
:-table sp(+,+,-,min).
sp(X,Y,[[X,Y]],W) :-
    edge(X,Y,W).
sp(X,Y,[[X,Z]|Path],W) :-
    edge(X,Z,W1),
    sp(Z,Y,Path,W2),
    W is W1+W2.
```

The predicate $sp(X,Y,Path,W)$ states that $Path$ is a path from $X$ to $Y$ with the smallest weight $W$. 
Tabling in B-Prolog

Shortest path \( (sp) \) in a weighted directed graph \( X \xrightarrow{w} Y \)

\[
:\text{-table } sp(+,+,+-,\text{min}).
\]

\[
sp(X,Y,[(X,Y)],W) :-
\]
\[
\text{edge}(X,Y,W).
\]

\[
sp(X,Y,[(X,Z)|Path],W) :-
\]
\[
\text{edge}(X,Z,W1),
\]
\[
sp(Z,Y,Path,W2),
\]
\[
W \text{ is } W1+W2.
\]

The predicate \( sp(X, Y, \text{Path}, W) \) states that \( \text{Path} \) is a path from \( X \) to \( Y \) with the smallest weight \( W \).

For each pair of nodes, only one (shortest) answer is tabled!
Neng-Fa’s implementation of the Sokoban program is based on the just seen tabled definition of shortest path

The overall code is very short and simple (as one might expect from Prolog programming)

A little (actually very little) domain knowledge is added
The Sokoban program

:-table plan_sokoban(+,+,-,min).
plan_sokoban(_,BoxLocs,Plan,Len):-goal_reached(BoxLocs),!, Plan=[],Len=0.
plan_sokoban(SokobanLoc,BoxLocs,[push(BoxLoc,Dir,DestLoc)|Plan],Len):-
    select(BoxLoc,BoxLocs,BoxLocs1),
    step(PrevNeibLoc,Dir,BoxLoc),
    \+ member(PrevNeibLoc,BoxLocs1),
    step(BoxLoc,Dir,NextNeibLoc),
    good_dest(NextNeibLoc,BoxLocs1),
    reachable_by_sokoban(SokobanLoc,PrevNeibLoc,BoxLocs),
    choose_dest(BoxLoc,NextNeibLoc,Dir,DestLoc,NewSokobanLoc,BoxLocs1),
    insert_ordered(DestLoc,BoxLocs1,NewBoxLocs),
    plan_sokoban(NewSokobanLoc,NewBoxLocs,Plan,Len1),
    Len is Len1+1.
The Sokoban program

:-table reachable_by_sokoban/3.
reachable_by_sokoban(Loc,Loc,_BoxLocs).
reachable_by_sokoban(Loc1,Loc2,BoxLocs):-
    step(Loc1,_,Loc3),
    \+ member(Loc3,BoxLocs),
reachable_by_sokoban(Loc3,Loc2,BoxLocs).

choose_dest(Loc,NextLoc,_Dir,Dest,NewSokobanLoc,_BoxLocs):-
    Dest=NextLoc, NewSokobanLoc=BoxLoc.
choose_dest(Loc,NextLoc,Dir,Dest,NewSokobanLoc,BoxLocs):-
    step(NextLoc,Dir,NextNextLoc),
good_dest(NextNextLoc,BoxLocs),
choose_dest(NextLoc,NextNextLoc,Dir,
    Dest,NewSokobanLoc,BoxLocs).
The Sokoban program

Domain Knowledge

good_dest(Loc, BoxLocs):-
  \+ member(Loc, BoxLocs),
  (corner(Loc) -> storage(Loc); true),
  foreach(BoxLoc in BoxLocs, \+ stuck(BoxLoc, Loc)).
:-table stuck/2.

stuck(X, Y):- (right(X, Y); right(Y, X)),
  \+ storage(X); \+ storage(Y),
  \+ top(X, _), \+ top(Y, _);
  \+ top(_, X), \+ top(_, Y)), !.

stuck(X, Y):- (top(X, Y); top(Y, X)),
  \+ storage(X); \+ storage(Y),
  \+ right(X, _), \+ rights(Y, _);
  \+ right(_, X), \+ right(_, Y)), !.

Two boxes constitute a deadlock if they are next to each other and both adjacent to a wall, unless both their locations are storage squares.
## Competition results

### CPU time, seconds

<table>
<thead>
<tr>
<th>Instance</th>
<th>BPSolver</th>
<th>Clasp</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-sokoban-optimization-0-0.asp</td>
<td>0.58</td>
<td>0.06</td>
</tr>
<tr>
<td>13-sokoban-optimization-0-0.asp</td>
<td>0.06</td>
<td>0.74</td>
</tr>
<tr>
<td>18-sokoban-optimization-0-0.asp</td>
<td>0.00</td>
<td>9.80</td>
</tr>
<tr>
<td>20-sokoban-optimization-0-0.asp</td>
<td>33.57</td>
<td>13.24</td>
</tr>
<tr>
<td>24-sokoban-optimization-0-0.asp</td>
<td>2.66</td>
<td>3.52</td>
</tr>
<tr>
<td>27-sokoban-optimization-0-0.asp</td>
<td>0.78</td>
<td>1.16</td>
</tr>
<tr>
<td>29-sokoban-optimization-0-0.asp</td>
<td>0.78</td>
<td>2.92</td>
</tr>
<tr>
<td>33-sokoban-optimization-0-0.asp</td>
<td>1.96</td>
<td>26.74</td>
</tr>
<tr>
<td>37-sokoban-optimization-0-0.asp</td>
<td>0.38</td>
<td>8.52</td>
</tr>
<tr>
<td>4-sokoban-optimization-0-0.asp</td>
<td>Mem Out</td>
<td>0.62</td>
</tr>
<tr>
<td>43-sokoban-optimization-0-0.asp</td>
<td>Mem Out</td>
<td>35.67</td>
</tr>
<tr>
<td>45-sokoban-optimization-0-0.asp</td>
<td>Mem Out</td>
<td>9.30</td>
</tr>
<tr>
<td>47-sokoban-optimization-0-0.asp</td>
<td>Mem Out</td>
<td>18.66</td>
</tr>
<tr>
<td>5-sokoban-optimization-0-0.asp</td>
<td>0.00</td>
<td>0.16</td>
</tr>
<tr>
<td>9-sokoban-optimization-0-0.asp</td>
<td>0.00</td>
<td>2.12</td>
</tr>
</tbody>
</table>
Conclusions

We have played with the Sokoban game using several Logic Programming tools.
Conclusions

- We have played with the Sokoban game using several Logic Programming tools.
- It was funny.
Conclusions

- We have played with the Sokoban game using several Logic Programming tools
- It was funny
- Direct ASP (or B translated to ASP) works
- B interpreted by CLP(FD) does not work correctly (but we are now developing and exploiting a special reachability global constraint)
Conclusions

- We have played with the Sokoban game using several Logic Programming tools.
- It was funny.
- Direct ASP (or B translated to ASP) works.
- B interpreted by CLP(FD) does not work correctly (but we are now developing and exploiting a special reachability global constraint).
- Tabled B-Prolog works (even if there are still some memory problems to cope with).
- Adding knowledge, of course, helps.