A Tabled Prolog Program for Solving Sokoban

Neng-Fa Zhou[1] and Agostino Dovier[2]

[1] Department of Computer and Information Science,
CUNY Brooklyn College & Graduate Center, USA,
zhou@sci.brooklyn.cuny.edu
[2] Dipartimento di Matematica e Informatica,
Università di Udine, Udine, Italy,
agostino.dovier@uniud.it

Abstract. This paper presents our program in B-Prolog submitted to the third ASP solver competition for the Sokoban problem. This program, based on dynamic programming, treats Sokoban as a generalized shortest path problem. It divides a problem into independent subproblems and uses mode-directed tabling to store subproblems and their answers. This program is very simple but quite efficient. Without use of any sophisticated domain knowledge, it easily solved 11 of the 15 instances used in the competition.

1 Introduction

Sokoban is a type of transport puzzle, in which the player finds a plan for the Sokoban (means warehouse-keeper in Japanese) to push all the boxes into the designated areas. This problem has been shown to be NP-hard and has raised great interest because of its relation to robot motion planning [6]. This problem has been used as a benchmark in the Answer Set Programming competition [5, 4] and International Planning competition[3], and solutions for ASP solvers and PDDL are available. In [15], an IDA*-based program is presented with several domain-dependent enhancements.

This paper presents the program in B-Prolog, called the *BPSolver program* below, submitted to the third ASP solver competition [4]. The BPSolver program is based on the dynamic programming approach and uses mode-directed tabling [18] to store subproblems and their answers. The program was built after failed attempts to use CLP(FD) and the planning languages B [12] and BMV [8] for the problem (see Section 6). The BPSolver program is very simple (only a few lines of code) but quite efficient. In the competition, the BPSolver program solved 11 of the 15 instances of which the hardest instance took only 33 seconds, and failed to solve the remaining four instances due to lack of table space.

As far as we know, the BPSolver program is the first to apply the dynamic programming approach to the Sokoban problem. The BPSolver program treats Sokoban as a generalized shortest path problem where the locations of objects

[3] http://ipc.informatik.uni-freiburg.de/Domains

particular to a subproblem are tabled. The BPSolver program does not employ any sophisticated domain knowledge. It only checks for two simple deadlock cases: one is that a box is stuck in a corner and the other is that two boxes next to each other are stuck by a wall. With sophisticated domain knowledge, the BPSolver program is expected to perform much better.

The remainder of the paper is structured as follows: Section 2 gives a detailed description of the Sokoban problem; Section 3 introduces tabling, and in particular mode-directed tabling, as implemented in B-Prolog; Section 4 explains the BPSolver program line by line; Section 5 presents the competition results; Section 6 compares with related work and points out possible improvements; and Section 7 concludes the paper.

2 The Problem Description

The following is an adapted description of the Sokoban problem used in the ASP solver competition.[4]

Sokoban is a type of transport puzzle invented by *Hiroyuki Imabayashi* in 1980 and published by the Japanese company Thinking Rabbit, Inc. in 1982. "Sokoban" means "warehouse-keeper" in Japanese. The puzzle consists of a maze which has two types of squares: inaccessible *wall squares* and accessible *floor squares*. Several boxes are initially placed on some of the floor squares and the same number of floor squares are designated as storage squares. There is also a man (the Sokoban) whose duty is to move all the boxes to the designated storage squares. A floor square is *free* if it is not occupied by either a box or the man. The man can walk around by moving from his current position to any adjacent free floor square. He can also push a box into an adjacent free square, but in order to do so he needs to be able to get to the free square behind the box. The goal of the puzzle is to find a shortest plan to push all the boxes to the designated storage squares. To reduce the number of steps, the Sokoban moves and the successive sequence of pushes in the same direction are considered as an atomic action.

A problem instance is given by the following relations:

- `right(`L_1, L_2`)`: location L_2 is immediately to the right of location L_1.
- `top(`L_1, L_2`)`: location L_2 is immediately on the top of location L_1.
- `box(`L`)`: location L initially holds a box.
- `sokoban(`L`)`: the man is initially at location L.
- `storage(`L`)`: location L is a storage square.

In this setting, the wall squares are completely ignored and the adjacency relation of the floor squares is given by the `right` and `top` predicates. This input is well suited for Prolog. According to the ASP competition requirements, the output should be represented by atoms of the form `push(`$L1$`,Dir,`L_2`,Time)`, where $L1$ and L_2 are two locations, L_2 is reachable from L_1 going through the direction

[4] https://www.mat.unical.it/aspcomp2011

`Dir` (`left`, `right`, `up`, or `down`), and `Time` is an integer greater than 0 (bounded the further input predicate `step`). For each admissible value of time exactly one `push` action must occur. We also allow a slight variation of this predicate where the time information is left implicit and a consecutive sequence of push is stored in a list that, in fact, represents a plan.

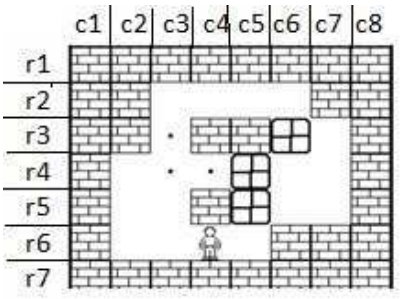

Fig. 1. A Sokoban problem.

Figure 1 shows an example problem where the storage squares have dots on them. This state is represented by the following facts:

```
top(c2r5,c2r4).        right(c3r2,c4r2).      right(c3r6,c4r6).
top(c2r6,c2r5).        right(c4r2,c5r2).      right(c4r6,c5r6).
top(c3r3,c3r2).        right(c5r2,c6r2).
top(c3r4,c3r3).        right(c6r3,c7r3).      box(c6r3).
top(c3r5,c3r4).        right(c2r4,c3r4).      box(c5r4).
top(c3r6,c3r5).        right(c3r4,c4r4).      box(c5r5).
top(c5r5,c5r4).        right(c4r4,c5r4).
top(c5r6,c5r5).        right(c5r4,c6r4).      storage(c3r3).
top(c6r3,c6r2).        right(c6r4,c7r4).      storage(c3r4).
top(c6r4,c6r3).        right(c2r5,c3r5).      storage(c4r4).
top(c6r5,c6r4).        right(c5r5,c6r5).
top(c7r4,c7r3).        right(c6r5,c7r5).      sokoban(c4r6).
top(c7r5,c7r4).        right(c2r6,c3r6).
```

The following gives a plan of 13 steps for the problem:

```
[push(c6r3,down,c6r5),  push(c5r4,left,c3r4),  push(c3r4,down,c3r5),
 push(c5r5,up,c5r4),    push(c6r5,left,c5r5),  push(c5r4,right,c6r4),
 push(c5r5,up,c5r4),    push(c6r4,up,c6r3),    push(c5r4,left,c4r4),
 push(c6r3,down,c6r4),  push(c3r5,up,c3r3),    push(c4r4,left,c3r4),
 push(c6r4,left,c4r4) ]
```

3 Tabling in B-Prolog

Tabling [17] 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 and, naturally, tabling is amenable to dynamic programming problems.

B-Prolog is a tabled Prolog system that is based on linear tabling [19], allows variant subgoals to share answers, and uses the local strategy [9] (also called lazy strategy [19]) to return answers. In B-Prolog, tabled predicates are declared explicitly by declarations in the following form:

```
:-table P1/N1,...,Pk/Nk
```

where each `Pi` ($\texttt{i} \in \{\texttt{1}, \ldots, \texttt{k}\}$) is a predicate symbol and `Ni` is an integer that denotes the arity of `Pi`.

Consider, for example, the tabled predicate computing 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 `fib(N,X)` would spawn 2^N subgoals, many of which are variants. With tabling, however, the time complexity drops to linear since the same variant subgoal is resolved only once.

For a tabled predicate, all the arguments of a tabled subgoal are used in variant checking and all answers are tabled. This table-all approach is problematic for many dynamic programming problems such as those that require computation of aggregates. Mode-directed tabling [13, 18] amounts to using table modes to instruct the system on how to table subgoals and their answers. In B-Prolog, a table mode declaration takes the following form:

```
:-table p(M1,...,Mn):C.
```

where `p/n` is a predicate symbol, `C`, called a *cardinality limit*, is an integer which limits the number of answers to be tabled for `p/n`, and each `Mi` ($\texttt{i} \in \{\texttt{1}, \ldots, \texttt{k}\}$) is a mode which can be `min`, `max`, `+`, or `-`. When `C` is 1, it can be omitted together with the preceding ':'. For each predicate, only one table mode declaration can be given. In the current implementation in B-Prolog, only one argument in a tabled predicate can have the mode `min` or `max`. Since an optimized argument can be a compound term and the built-in `@<`/2 is used to select better answers for compound terms, this restriction is not essential.

The mode `+` is called *input*, `-` *output*, `min` *minimized*, and `max` *maximized*. An argument with the mode `min` or `max` is called *optimized*. An optimized argument is assumed to be output. The system uses only input arguments in variant checking of tabled subgoals, ignoring all other arguments. Notice that a table mode

does not tell the instantiation state of an argument. Nevertheless, normally an input argument is ground and an output argument is a variable.

A mode declaration not only instructs the system on what arguments are used in variant checking, it also guides it in tabling answers. After an answer of a tabled subgoal is produced, the system tables it unconditionally if the cardinality limit is not reached yet. When the cardinality limit has been reached, however, the system tables the answer only if it is better than some existing answer in terms of the optimized argument. If no argument is optimized, all new answers are discarded once the cardinality limit has been reached.

Mode-directed tabling is very useful for declarative description of dynamic programming problems. The following predicate finds a path with the minimal weight between a pair of nodes in a directed graph.

```
:-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 `edge(X,Y,W)` defines a given weighted directed graph, where `W` is the weight of the edge from node `X` to node `Y`. The predicate `sp(X,Y,Path,W)` states that `Path` is a path from `X` to `Y` with the smallest weight `W`. Notice that whenever the predicate `sp/4` is called, the first two arguments are assumed to be instantiated. So for each pair of nodes, only one answer is tabled.

4 The Program

In this section, we explain the BPSolver program. The program treats the Sokoban problem as a generalized shortest path problem. For a state, if it is the goal state in which every box is in a storage location, it is done. Otherwise, the program chooses an intermediate state and splits the problem into two subproblems, one transforming the current state to the intermediate one and the other transforming the intermediate one to the goal state. All the states are tabled so that the same subproblem is solved only once.

4.1 Library and helper predicates

Before we show the program, we give the library and helper predicates used in the program. For each helper predicate written as part of the program, we give its definition.

- `member(`X, L`)`: succeeds when X is a member of the list L. It can be used to check if a given element is a member of a list and it can also be used to nondeterministically select an element from a list.

– `select(`X, L, R`)`: the same as `member(`X, L`)` except that it binds R to the rest of the list after X is selected from L.
– `neib(`$Loc1, Loc2, Dir$`)`: $Loc2$ is the next location of $Loc1$ along the direction Dir. It is defined as follows in terms of the given predicates `top/2` and `right/2`:

```
:-table neib/3.
neib(Loc1,Loc2,up):-top(Loc1,Loc2).
neib(Loc1,Loc2,down):-top(Loc2,Loc1).
neib(Loc1,Loc2,right):-right(Loc1,Loc2).
neib(Loc1,Loc2,left):-right(Loc2,Loc1).
```

The predicate is tabled for better performance.
– `insert_ordered(X,L1,L2)`: inserts X into a sorted list $L1$, resulting in a new sorted list $L2$.

```
insert_ordered(X,[],[X]).
insert_ordered(X,[Y|Ys],[X,Y|Ys]):-
    X @=<Y,!.
insert_ordered(X,[Y|Ys],[Y|Ordered]):-
    insert_ordered(X,Ys,Ordered).
```

– `goal_reached(L)`: every location in L is a storage location.

```
goal_reached([]).
goal_reached([Loc|Locs]):-
    storage(Loc),
    goal_reached(Locs).
```

This can be defined equivalently using the `foreach` construct of B-Prolog as follows:

```
goal_reached(Locs):-
    foreach(Loc in Locs, storage(Loc)).
```

– `corner(`Loc`)`: succeeds if Loc is a corner location. No box can be moved to a corner unless it is a storage square.

```
:-table corner/1.
corner(X) :- \+ noncorner(X).
noncorner(X) :- top(_,X),top(X,_).
noncorner(X) :- right(_,X),right(X,_).
```

This predicate is tabled. For the problem instance shown in Figure 1, for example, the table will contain seven possible subgoals including `corner(c2r4)` and `corner(c3r2)`.
– `stuck(`$Loc1, Loc2$`)`: two boxes in $Loc1$ and $Loc2$ constitute a deadlock if they are next to each other by a wall, unless both locations are storage squares.

```
:-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,_), \+ right(Y,_);
     \+ right(_,X), \+ right(_,Y)),!.
```

For example, for the problem instance shown in Figure 1, the subgoal `stuck(c3r2,c4r2)` succeeds and so does `stuck(c4r2,c3r2)`.

4.2 The main program

As already said, the main idea behind the main program reported in Figure 2 is to implement a tabled version of a generalization of the shortest path problem. The subgoal

```
plan_sokoban(SokobanLoc, BoxLocs, Plan, Len)
```

finds a plan `Plan` with the minimal length `Len` for the current state, where `SokobanLoc` is the location of the man and `BoxLocs` is a list of box locations. For example, for the problem instance shown in Figure 1, the subgoal would look like

```
plan_sokoban(c4r6,[c5r4,c5r5,c6r3],Plan,Len).
```

The predicate is tabled under control by the mode `plan_sokoban(+,+,-,min)`, which means that only one plan with the minimal length is tabled for each different state. The list `BoxLocs` is sorted in lexicographic order to make tabling more effective.

When the goal has been reached (`goal_reached(BoxLocs)` succeeds), an empty plan is returned. Otherwise, the second rule selects a box location `BoxLoc` from `BoxLocs` and a destination location `DestLoc` that can be reached from `BoxLoc` in the direction `Dir` (`up`, `down`, `left`, or `right`), and adds the action `push(BoxLoc,Dir,DestLoc)` into the plan. Only feasible actions are added. An action `push(BoxLoc,Dir,DestLoc)` is feasible if (1) the previous location `PrevNeibLoc` of `BoxLoc` in the direction `Dir` is free; (2) the man can walk to this location (`reachable_by_sokoban`); and (3) the location `DestLoc` is a good destination that does not result in a deadlock. The subgoal `choose_dest` non-deterministically chooses a destination `DestLoc` from the free squares ahead of `BoxLoc` in the direction `Dir`. After pushing the box at `BoxLoc` to `DestLoc`, the man moves to `NewSokobanLoc` which is the previous square of `DestLoc`.

The predicate `reachable_by_sokoban` checks if there is a path of free squares from one location to another. Again, tabling is used to prevent loops and avoid resolving the same subgoal more than once.

The predicate `good_dest` checks whether or not a location is a good destination for a box. A location `Loc` is a good destination if (1) it is not occupied

```
:-table plan_sokoban(+,+,-,min).
plan_sokoban(_SokobanLoc,BoxLocs,Plan,Len):-
    goal_reached(BoxLocs),!,
    Plan=[],Len=0.
plan_sokoban(SokobanLoc,BoxLocs,[push(BoxLoc,Dir,DestLoc)|Plan],Len):-
    select(BoxLoc,BoxLocs,BoxLocs1),
    neib(PrevNeibLoc,BoxLoc,Dir),
    \+ member(PrevNeibLoc,BoxLocs1),
    neib(BoxLoc,NextNeibLoc,Dir),
    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.

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

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

choose_dest(Loc,NextLoc,_Dir,Dest,NewSokobanLoc,_BoxLocs):-
    Dest=NextLoc, NewSokobanLoc=BoxLoc.
choose_dest(Loc,NextLoc,Dir,Dest,NewSokobanLoc,BoxLocs):-
    neib(NextLoc,NextNextLoc,Dir),
    good_dest(NextNextLoc,BoxLocs),
    choose_dest(NextLoc,NextNextLoc,Dir,Dest,NewSokobanLoc,BoxLocs).
```

Fig. 2. The main program

by any box; (2) it is not a corner unless it is a storage square; and (3) moving a box to `Loc` does not result in a deadlock. As mentioned above, two boxes next to each other by a wall constitute a deadlock unless the two locations are storage squares. There are more sophisticated deadlock cases that involve more than two locations [15], but these cases are not considered here.

5 The Competition Results

Sokoban was one of the benchmarks of the 2011 ASP competition [4]. The main scope of the competition is to challenge different solvers on declarative encodings. In particular, in the *System Track* different ASP solvers were required to run on a proposed encoding in Answer Set Programming (pure declarative code, no optimization). In the next Section we will briefly sketch this modeling. It is a decision version of the problem where a plan of a given length is looked for. The allowed actions are `push` of a block in the four directions. Moreover, the move of the Sokoban for reaching (if possible) a block is supposed to happen instantaneously immediately before the successive push move. Most of the ASP solvers behave quite well on the proposed instances. It must be observed, however, that the instances were not so large (the more difficult were of 6 boxes/20 moves). In the *Model and Solve* competition, competitors were allowed to encode directly the problem allowing some domain information. In this case a minimum length plan is looked for. Most of the submitted programs are variants of the one proposed for the System Track; for this approach, best performances have been obtained by the family of Clasp solvers [10] (`http://potassco.sourceforge.net/`) and by EZCSP [1] (`http://marcy.cjb.net/ezcsp/index.html`).

The BPSolver program is available at: `www.sci.brooklyn.cuny.edu/~zhou/asp11/` Table 1 gives the CPU times of the actual runs in the third ASP solver competition. In comparison, the result of Clasp, the solver that won this benchmark, is also shown. For the solved instances, BPSolver is actually a little faster than Clasp on average. BPSolver failed to solve four of the instances due to lack of table space.

6 Related Work

The Sokoban problem is a typical planning problem where a set of admissible *actions* might affect the value of some *fluents* that globally determine a *state*. This kind of problems are naturally encoded using Action Description Languages such as STRIPS, B, and PDDL. Before starting the encoding one needs to carefully choose the atomic actions allowed for the Sokoban and their duration.

The simplest choice (finest granularity) is that at each time step the Sokoban is allowed to do a single `move`, or a single `push` of a box, in one of the four direction `up`, `down`, `left`, and `right`; the duration of the move is 1. This is the basic encoding of the Sokoban problem (see, e.g. `http://ipc.informatik.uni-freiburg.de/Domains`); it is simple and elegant and the non deterministic branching in the search is limited to 4. Any state can be represented by 3ℓ

Table 1. Competition results (CPU time, seconds).

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

fluents of the form `free(L)`, `box_in(L)`, `sokoban_in(L)`, where L is one of the ℓ admitted cells. The actions affect these fluents. However, with this encoding, a lot of steps are needed either to push a block without changing directions or to reach the next block to be pushed. This increases the number of steps necessary for the plan and, since the size of the search tree grows exponentially in this number, it is rather difficult to solve non-trivial instances.

As already said, the granularity chosen for the ASP competition is coarser. As soon as there is a path from the Sokoban position to the desired side of a block, the move action is left implicit (it takes zero time). Just a unique family of `push` actions are admitted, parametric on the starting `From` and arrival `To` points of the block (aligned in a given direction `D`). A push of any number of steps in the same direction is viewed as an atomic action. If, on the one side, this allows to dramatically decrease the number of (macro) actions needed for executing the plan, on the other side, it generates new problems. The first is that the branching is now increased. The second, more subtle, is that the modeling language needs to be able to deal with a dynamic notion of reachability.

Basically, for stating that the action `push(From,D,To)` be executable, we need to require that: `box_in(From)`, that all the cells `L` between `From` and `To` are `free`, and, morevoer, that the Sokoban can reach the cell adjacent to `From` in the direction `D`, external to the segment [`From`,`To`]. Let us call `S1` this cell, we need to require that `reachable(S1)` (namely that the cell is currently reachable from the Sokoban).

We need therefore to introduce ℓ additional Boolean fluents `reachable(L)` and to deal with them. This can be done using *static causal laws* (or rules) that are not allowed in all Action Description Languages. We should write two rules of the form (using the syntax of the language B):

```
sokoban_in(A) caused reachable(A).

reachable(B) and free(C) caused reachable(C) if adjacent(B,C).
```

The semantics of an Action Description Language in presence of static causal laws becomes complex [12, 8] and is related to the notion of Answer Set [11]. As shown in detail in [7] B programs can be either

- interpreted using constraint (logic) programming, or
- automatically translated in Answer Set Programming and then solved using an ASP solver.

The former encoding is also studied in a slight different context, by other authors (e.g., [2]); however, static laws are not considered. The encoding implemented in [7] deals with static causal laws, but the proposed implementation does not ensure correctness for some classes of static causal laws. Other encodings are viable, but they would introduce too many constraints. Intuitively, this happens when rules introduce loops of implications between literals. In the case of Sokoban, simultaneous un-justified changes of fluent values might satisfy the constraints, the Sokoban can reach unreachable cells, and not-allowed push moves can be executed. The same problem was pointed out in [16] where authors translated a ground ASP Program into a SAT encoding. In presence of such kind of loops, solutions of the SAT formula obtained are not admissible answer sets. The problem can be avoided introducing the so-called *loop-formulas* but their number can grow exponentially. Unfortunately, the above definition of reachability as static causal laws introduces these undesirable loops and therefore a CLP(FD) approach for solving it in this way (e.g., using B-Prolog) is not feasible.

As far as the latter approach is concerned, it works correctly on a modeling in B based on the ideas above.[5] It is well-known that the main problem of Answer Set Programming is the size of the ground version of the program that is computed in the first stage of the solution's search. We experimentally observe that this size is bigger (typically, twice) than the size of the ASP program written by the Clasp group that won the competition. A direct encoding of this problem in ASP, of course, produces clever code. Let us say some words about this program. First of all, it focuses on 2ℓ Boolean fluents `box` and `sokoban`, repeated for each time step (the fluent free is left implicit). The reachability relation (called `route`) is encoded directly in ASP (in a similar way as done above in B) and it is parametric on time steps. Push move is split into `push_from`, `push`, and `push_to`. This allows us to reduce the grounding. The `push_from` relation is defined to be a function w.r.t. the time step (and defined only if the goal has not yet been reached):

```
1 { push_from(L,D,S): loc(L): direction(D) } 1 :-
                                          step(S), not goal(S).
```

Namely, for each step `S` that in which the goal has not yet been reached, just one move is allowed (one location and one direction are selected). Constraints

[5] The encoding is available in `http://www.dimi.uniud.it/dovier/CLPASP/BBMVLAST`.

are added to ensure action executability. If `push_from` is enabled, then the length of the move is non deterministically chosen and the consequent effects on fluents are determined. Constraints are also added to eliminate useless push moves; this reduces either the size of the corresponding ground program or the search space. As already said, the size of the ground program and the time needed to compute it are strictly less than those needed for the program automatically obtained from B. However, the running time after grounding (both in Clasp) are comparable on the instances of the ASP competition.

In AI literature, Andreas Junghanns and Jonathan Schaeffer in [15] pointed out that the Sokoban problem is interesting for several reasons: in general it is difficult to find a tight lower bound for the number of moves, there is the problem of a deadlock (e.g. when a box is pushed to a corner), and, moreover, the branching factor is very high (considering macro moves). The same authors then published some improving solutions to the problem in the context of single agent planning, summarized in a paper with Adi Botea and Martin Müller [3]. In particular, they exploited an abstraction based on tunnels and rooms of the Sokoban warehouse that allowed to obtain good performances. In [14] the authors show how to develop a domain-independent heuristics for cost-optimal planning. They apply this idea to the Sokoban, and test a STRIPS encoding of the Sokoban on a collection, called "microban", developed by David W. Skinner and available from `http://users.bentonrea.com/~sasquatch/sokoban/`. The STRIPS encoding used is based on the finest granularity approach (simple move), but reachability and other techniques are used as heuristics for sequences of atomic moves. They choose a collection of moderate instances and they are able to solve the 70% of them. Interestingly, they are able to find plans of length 290 (atomic actions) on instance 140 in half of an hour of computation.

Apart from academic contributions to this challenging puzzle we would like to point out two working solvers available on the web:

- Sokoban Puzzle Solver `http://codecola.net/sps/sps.htm`, developed (in 2003–2005) by Faris Serdar Taşel is basically a generate and test solver for Sokoban. An executable file for Windows is downloadable, but extra details on the implementation are not available. In spite of its apparent simplicity, it solves in acceptable time most the instances available from that web page.
- Sokoban Automatic Solver `http://www.ic-net.or.jp/home/takaken/e/soko/index.html`, developed (in 2003–2008) by Ken'ichiro Takahashi is another solver for Windows. It finds solutions that are not ensured to be optimal. It allows two options: (1) brute force (generate and test) and (2) using analysis. The second options allows faster executions but the author gives no idea on how this analysis is performed.

7 Concluding Remarks

This paper has presented the BPSolver program for solving the Sokoban problem. This program has demonstrated for the first time that dynamic programming is

a viable approach to the problem and mode-directed tabling is effective. Without using sophisticated domain knowledge, this program is able to solve some interesting instances that our other program in B, interpreted using CLP(FD) have failed to solve. As shown in the competition results, this program is as competitive as the Clasp program for the instances that are not so memory demanding.

The BPSolver program basically explores all possible states including states that can never occur in an optimal solution: a way for improving it is to consider more deadlock states such as those involving multiple blocks [15] to be filtered out. Moreover, some domain knowledge such as the topological information should be exploited to reduce the graph. Lastly, heuristics should be employed to select a box to move and a destination to move the box to. Ideally, a path that leads to a goal state should be explored as early as possible.

We believe that reasonable sized planning problems can benefit of the same technique presented.

Acknowledgement

We really thank Andrea Formisano for his wise advice in the B encoding of the Sokoban. Neng-Fa Zhou was supported in part by NSF (No.1018006). Agostino Dovier is partially supported by INdAM-GNCS 2011 and PRIN 20089M932N.

References

1. Marcello Balduccini. Splitting a cr-prolog program. In *LPNMR*, pages 17–29, 2009.
2. R. Barták and D. Toropila. Reformulating constraint models for classical planning. In David Wilson and H. Chad Lane, editors, *FLAIRS'08: Twenty-First International Florida Artificial Intelligence Research Society Conference*, pages 525–530. AAAI Press, 2008.
3. Adi Botea, Martin Müller, and Jonathan Schaeffer. Using abstraction for planning in sokoban. In *Computers and Games*, pages 360–375, 2002.
4. Francesco Calimeri, Giovambattista Ianni, Francesco Ricca, Mario Alviano, Annamaria Bria, Gelsomina Catalano, Susanna Cozza, Wolfgang Faber, Onofrio Febbraro, Nicola Leone, Marco Manna, Alessandra Martello, Claudio Panetta, Simona Perri, Kristian Reale, Maria Carmela Santoro, Marco Sirianni, Giorgio Terracina, and Pierfrancesco Veltri. The third answer set programming competition. In *LPNMR*, pages 388–403, 2011.
5. Marc Denecker, Joost Vennekens, Stephen Bond, Martin Gebser, and Miroslaw Truszczynski. The second answer set programming competition. In *LPNMR*, pages 637–654, 2009.
6. Dorit Dor and Uri Zwick. Sokoban and other motion planning problems. *Computational Geometry: Theory and Applications*, 13:215–228, 1995.
7. Agostino Dovier, Andrea Formisano, and Enrico Pontelli. Perspectives on Logic-based Approaches for Reasoning About Actions and Change. In *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning, Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday*, volume 6565 of *LNCS*, pages 259–279. Springer, 2011.

8. Agostino Dovier, Andrea Formisano, and Enrico Pontelli. Multivalued action languages with constraints in CLP(FD). *TPLP*, 10(2):167–235, 2010.
9. Juliana Freire, Terrance Swift, and David Scott Warren. Beyond depth-first: Improving tabled logic programs through alternative scheduling strategies. *Journal of Functional and Logic Programming*, 1998.
10. Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Sven Thiele. Engineering an incremental asp solver. In *ICLP*, pages 190–205, 2008.
11. Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proceedings of the Joint International Conference and Symposium on Logic Programming (JICSLP)*, pages 1070–1080, 1988.
12. Michael Gelfond and Vladimir Lifschitz. Action languages. *Electron. Trans. Artif. Intell.*, 2:193–210, 1998.
13. Hai-Feng Guo and Gopal Gupta. Simplifying dynamic programming via mode-directed tabling. *Softw., Pract. Exper.*, 38(1):75–94, 2008.
14. Patrik Haslum, Adi Botea, Malte Helmert, Blai Bonet, and Sven Koenig. Domain-independent construction of pattern database heuristics for cost-optimal planning. In *AAAI*, pages 1007–1012, 2007.
15. Andreas Junghanns and Jonathan Schaeffer. Sokoban: Enhancing general single-agent search methods using domain knowledge. *Artif. Intell.*, 129(1-2):219–251, 2001.
16. Fangzhen Lin and Yuting Zhao. ASSAT: Computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157(1–2):115–137, 2004.
17. David Scott Warren. Memoing for logic programs. *Comm. of the ACM, Special Section on Logic Programming*, 35:93–111, 1992.
18. Neng-Fa Zhou, Yoshitaka Kameya, and Taisuke Sato. Mode-directed tabling for dynamic programming, machine learning, and constraint solving. In *ICTAI*, pages 213–218, 2010.
19. Neng-Fa Zhou, Taisuke Sato, and Yi-Dong Shen. Linear tabling strategies and optimizations. *TPLP*, 8(1):81–109, 2008.