Willy: A Sokoban Solving Agent

Luis Rei, Rui Teixeira

Faculdade de Engenharia da Universidade do Porto, R. Dr. Roberto Frias,
4200-465 Porto.
http://www.fe.up.pt/
Iuis.rei@gmail.com, http://luisrei.com
rui.teixeira.ni@gmail.com

Abstract. Sokoban is a classical, widely acclaimed, logic computer puzzle game. Despite it's simple set of rules the complexity of its mazes is overwhelming. This paper describes Willy, our first attempt to solve Sokoban puzzles, and our study of the underlying implementation difficulties of a Sokoban search agent: the very large search space and the state transitions that create unsolvable situations. Willy uses a search algorithm, IDA* and a good heuristic, Minmatching and then uses techniques to reduce the search space: removal of unsafe positions from the map, hash tables to prevent loops, macro moves to merge together multiple states into a single state and finally relevance cuts to attempt to make only relevant moves.

1 Introduction

Sokoban is a computer puzzle game in which the player pushes boxes around a maze in order to place them in designated locations. It was originally published in 1982 for the Commodore 64 and IBM-PC and has since been implemented in numerous computer platforms and video game consoles [1] .

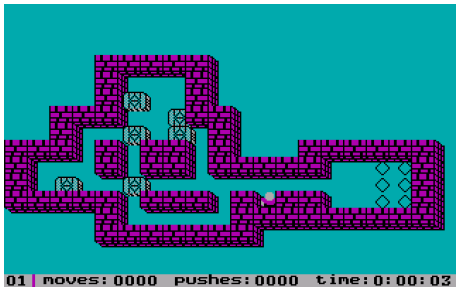


Fig. 1. Level 1 of the original PC version of Sokoban (adaptaded from [1]).

While Sokoban is just a game, it can represent a robot moving boxes in an actual warehouse and as such, it can be treated as an automated planning problem. Sokoban is an interesting problem for the field of artificial intelligence largely due to its difficulty. It has been proven NP-hard [2] and PSPACE-complete [3] . Sokoban is difficult due to its branching factor of 4 (up, down, left, right) and the huge depth of the solutions that averages 260 [4], resulting in approximately 3.4 x 10^156 states. In practice, in our test set, the median search space size is only around 10^18 [5]. Search space size alone is note necessarily a good indicator of the difficulty of a problem since it does not reflect the decision complexity [8], which in the case of Sokoban can be assumed to be high [5]. Additionally, a move may leave the puzzle in a state in which it is impossible to solve it, creating a state of deadlock.

2 Problem Formulation

In Sokoban, the player has to push boxes in a maze into goals. There are as many goals as there are boxes but all goals and all boxes are equal, in other words, there's no difference between pushing one box or the other to a given goal. The player can only push a single box at a time and is unable to pull. Under certain situations, it is possible to place the puzzle in a state in which it becomes impossible to solve. An easy example is when the player pushes a box into a corner where it becomes permanently immobilized, if that corner is not a goal, then the problem becomes unsolvable.

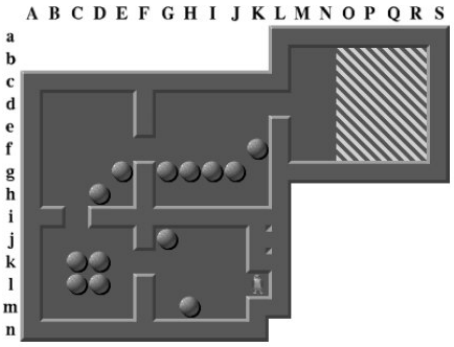


Fig. 2. Example of Deadlocks (adapted from [5]).

3 State Representation and Test Suite

The mazes and their state are represented by text characters:

- space, a free square
- '#', a wall square
- '$', a box
- '.', a goal square
- '@', the player

This representation of the mazes is the same that was used in Rolling Stone [6] and comes from XSokoban [7].

```
    #####
    #   #
    #$  #
  ###  $##
  #  $ $ #
### # ## #   ######
#   # ## #####  ..#
# $  $         ..#
##### ### #@##  ..#
    #     #########
    #######
```

Fig. 3. Representation of the initial state of Level 1 shown in Figure 1.

Internally, Willy also uses other symbols:

– '*', a box on a goal
– '!', the player on a goal
– 'x', an unsafe square
– '+', the player on an unsafe square

Less complex maps were created by reducing the number of boxes and goals in some of the maps obtained from XSokoban [7]. This was done in order to allow for quicker tests to be performed on Willy and the solutions implemented.

4 Search Algorithm

The search space of sokoban has a large search space with few goals, located deeply in the tree. Furthermore, one can have a lower-bound heuristic. These properties point to an informed search that finds sparsely distributed goals in a huge search space: Iterative Deepening A* [5].

4.1 Iterative Deepening A*

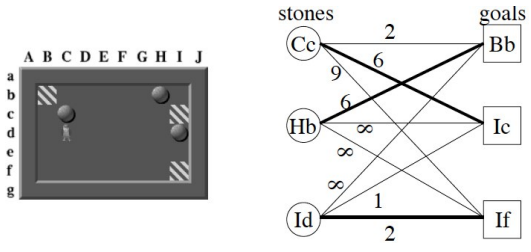
Iterative Deepening A* (IDA*) combines a depth-first iterative-deepening with the best-first heuristic search A*. At each iteration, IDA* performs a depth-first search, cutting off a branch when its total cost, $g + h$, where g is the cost to reach the current node and h is the expected cost to reach a solution (heuristic), exceeds a given threshold. This threshold starts at the estimate of the cost of the initial state, and increases for each iteration of the algorithm. At each iteration, the threshold used for the next iteration is the minimum cost of all values that exceeded the current threshold.

Not only does IDA* find a cheapest path to a solution and use far less space than A*, but it expands approximately the same number of nodes as A* in a tree search. IDA* is optimal in terms of solution cost, time, and space, over the class of admissible best-first searches on a tree [9].

4.2 Lower Bound Heuristic: Minimum Cost Matching

Willy uses the same lower bound heuristic of Rolling Stone [6] which estimates the number of box pushes needed to solve a Sokoban problem. This can be very hard to estimate accurately since puzzles can require complex maneuvers to solve such as pushing boxes through and away from goal squares to make room for other boxes.

For each box there is a minimum number of pushes required to maneuver that box into a particular goal. This minimum occurs if there are no other boxes in the maze. Since only one box can go to any one goal and all boxes must go to a goal in order to solve the puzzle, the problem is to find the assignment of boxes to goals that minimizes the sum of these distances (the total cost). This heuristic is called Minmatching [5]. The weight in the edges of the graph is the distance



stones on	goals on		
	Bb	*Ic*	*If*
Cc	2	**6**	9
Hb	**6**	∞	∞
Id	∞	1	**2**

Fig. 4. Minmatching example (adapted from [5]).

between the box (or stone) and the goal it links. If it is infinite, that means that it is impossible to push that box into that goal.

In order to calculate these values, a search has to be made for each box-goal pair in which all other boxes and goals are removed from the map. This obviously means that this is a very costly heuristic in computational terms. This cost is offset by a significant reduction in the number of nodes that have to be searched.

Willy currently uses an inferior heuristic, making a minimum cost matching using the distance between squares without accounting for walls.

5 Unsafe Positions

A state of deadlock is a state in which the puzzle is unsolvable. While certain deadlock states can be hard to detect, some are very simple. For instance, if the box is pushed into a corner, it can not be further moved. If that corner is not a goal, than the puzzle is now in an unsolvable state. Thus it is trivial and inexpensive to mark all corners without a goal as "unsafe" before the search even begins. Another observation is that all the squares in a line between two unsafe positions are also unsafe if they are positioned along a wall without goals. Since these positions remain constant and depend only on the map itself, pushing a box into these previously marked positions can be considered an illegal move and all the states that included them can be pruned from the search tree [10].

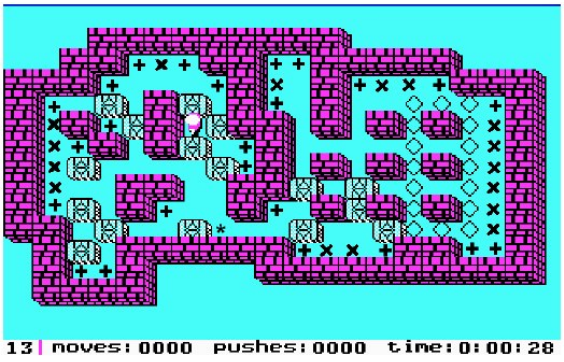


Fig. 5. Unsafe positions (adapted from [10]).

6 Hash Tables

Hash tables accomplish two different tasks: avoid cycles and duplicating work by detecting previously visited nodes [5]. Willy uses the position of the boxes and the player. If any state has the same box and player positions as another already in the hash table, it is discarded. Otherwise it is added to the table.

7 Macro Moves

A macro is a rule or pattern that specifies how a certain input sequence should be mapped to an output sequence. Macro moves, in Sokoban, consist in treating a sequence of moves as a single move. Using a macro effectively removes all the moves encompassing the macro from the search tree. Thus, the use of macro moves decreases the depth and branching factor of the search tree - the latter occurs whenever a move is incorporated in the macro.

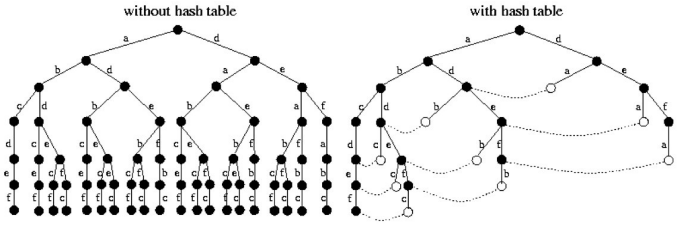


Fig. 6. Effect of using a hash table on the search tree (adapted from [4]).

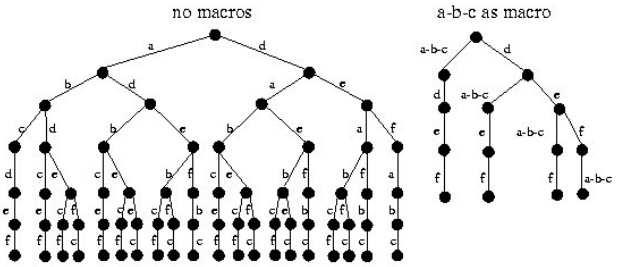


Fig. 7. Effect of using macros on the search tree (adapted from [4]).

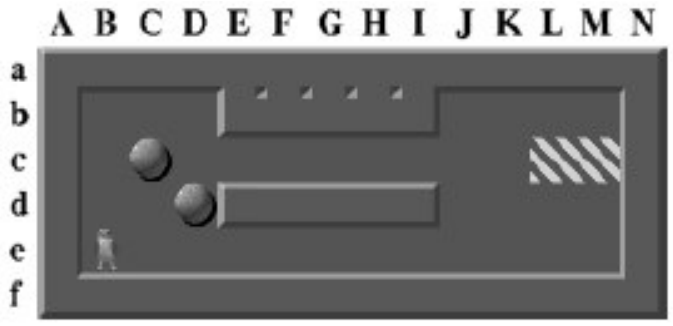


Fig. 8. Example of a tunnel (adapted from [5]).

7.1 Tunnel Macros

Tunnels are narrow, straight pathways through which only one stone may be pushed. When a box is pushed, in the example, from Dc to Ec, the push is replaced by Dc-Ic. There is no way to back out of a tunnel: either a stone is pushed out the way it came in or it is pushed from the other side if there is another path connecting the two ends of the tunnel. Because of this, pushing a second box into an already occupied tunnel generates a deadlock.

8 Relevance Cuts

Choosing a move that is relevant to the current state is a skill that is easy and natural for a human user but very complex for a computer. The search algorithm considers all legal moves and thus will consistently explore sequences of moves which a human player would never consider. Relevance cuts is a forward pruning technique that attempts to give the search algorithm the ability to choose which moves are relevant relatively to the previous moves [5]. The biggest drawback of this technique is that it may lead to non-optimal solutions. Fine tuning the aggressiveness of the cuts is therefore required.

The technique obtained good results in Rolling Stone however in the current implementation of Willy, the cost of the pre-computation was exceedingly high and RC were disabled until further improvements can be made.

9 Results and Conclusions

Map/Technique	IDA+Un+HT+Mac	IDA+HT+Mac	IDA+HT	IDA	IDA+Un+HT	IDA+Un+Mac	IDA+Un
Simplified1.1s	8	11	14	21	10	12	15
Simplified2.1s	39	63	63	96	39	57	57
Simplified3.1s	6	7	7	13	6	12	12
Simplified5.1s	43	69	69	88	43	55	55
Tunnel.1s	4	6	12	16	8	0.5	12

Fig. 9. Time in seconds it took to solve each map with different techniques.

Using only IDA* and a common heuristic it would take days to solve a Sokoban puzzle. Domain specific knowledge is vital. Each technique added allowed Willy to find the solution faster, the most significant overall improvement was adding Hash Tables.

Furthermore, the most powerful search enhancements can be linked to specific knowledge about the particular problem instance. For example, a puzzle with a long tunnels connecting the boxes to the goals will benefit heavily from tunnel macros. More specific optimizations can improve performance in certain maps or even in single maps.

10 Future Improvements

Further improvement of the lower bound heuristic in Willy seems to be the most important task ahead judging from the results obtained in Rolling Stone [5]. Fixing Relevance Cuts would probably also yield slightly better results. There are also several techniques that are planned for future versions of Willy.

10.1 Goal Macros

Goal macros are similar to tunnel macros. In most maps, all goals are located together, separate from the rest of the map, and in many of those maps there is a single entrance into that area or a small number of entrances. In those cases, once a box is pushed into the entrance, a macro can be activated to push the box into the correct goal [5]. The correct goal can be pre-computed before the search begins. Unlike tunnel macros, goal macros may result in non-optimal solutions.

10.2 Goal Cuts

If a move can be made that results in placing a box into a goal macro, it should be made instead of the alternatives which are deleted from the move list.

10.3 Move Ordering

Each successive iteration of IDA* takes longer as it expands more nodes than the previous iteration. Therefore the last iteration is potentially the longest. The objective of Move Ordering is to visit the "good" (according to some measure) successors first [11]. If it works, it means that the solution is found on the left side of the tree (the first nodes visited at that depth) more often, reducing the time it takes to complete the last iteration.

10.4 Backward Search

Backward search (or reversed search) consists in starting at the solution and searching until the initial state is found. It has been proven to be better in certain maps [5] [10] however there remains the problem of knowing in which maps it would be better. A possible solutions would be starting a Backward search and stopping it after a pre-determined number of nodes searched without finding a solution. Another solution that could be investigated would be to attempt to use learning methods or simply heuristics to identify which maps would benefit from using backward search instead of forward search.

References

1. Wikipedia Sokoban Last visited on July 21 2009
2. M. Fryers and M.T. Greene Sokoban, Eureka 54, 1995
3. J. C. Culberso Sokoban is PSPACE-complete, Technical Report TR 97-02, Dept. of Computing Science, University of Alberta, 1997
4. A. Junghanns and J. Schaeffer Pushing the Limits: New Developments in Single-Agent Search, GPW'99, October 16, 1999
5. A. Jungmans Pushing the limits: New developments in single-agent search, PhD Thesis, Uni-versity of Alberta, 1999
6. Rolling Stone http://www.cs.ualberta.ca/ games/Sokoban/program.html Last visited on July 23 2009
7. XSokoban http://www.cs.cornell.edu/andru/xsokoban.html Last visited on July 23 2009
8. V. Allis Searching for Solutions in Games and Artificial Intelligence, PhD Thesis, University of Limburg, 1994
9. R.E. Korf Depth-first Iterative-Deepening: An Optimal Admissible Tree Search, Artificial Intelligence, 27(1):97-109, 1985
10. F. Tales Sokoban: Reversed Solving, Leiden Institute of Advanced Computer Science (LIACS), Leiden University, June 2008
11. A. Reinfeld and T.A. Marsland Enhanced iterative-deepening search, IEEE Transactions on Pattern Analysis and Machine Intelligence, 16(7):701 710, July 1994