School of Computer Science and communication

Computer Vision and Active Perception Lab

Solving Sokoban game

Urban Pettersson

Ali Ghadirzadeh

Seyed Amir Hossein Banuazizi

Project group: HOPE

Project in Artificial Intelligence (DD2380)

Examiner: Patric Jensfelt

October 2011

1. Abstract

In this report, our team work to design an intelligent solver to handle the Sokoban game is presented. We (The hope members) aimed to present a versatile solver to be able to solve as many as possible game boards by designing an intelligent agent by the use of smart search methods combined with powerful heuristic functions and deadlock checkers. We have tested our algorithm on several different boards with different levels of difficulties. Our results proved a satisfactory level of performance in terms of our success rate in solving the game boards.

2. Introduction

The best definition for Sokoban game can be found at Wikipedia as “Sokoban is a type of transport puzzle, in which the player pushes boxes around a maze, viewed from above, and tries to put them in predetermined areas. Only one box may be pushed at a time, and boxes cannot be pulled. The numbers of boxes is equal to designated locations.” [1]

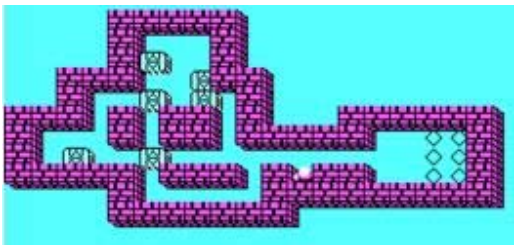


Figure 1) Sokoban game

This report describes a method for attempting to solve Sokoban puzzles by means of an efficient algorithm, a task which has been proven to be extremely difficult because of the huge search tree which is consequent of large branching factor especially with large boards with large number of boxes.

In our approach to solve this problem, we tried to implement Best First Search method which expands those states which seem to be more valuable than the others then goes through the depth levels of the chosen state to find the solution. In this way we try to lead our solver not to spend time on expanding redundant states and find the solution as soon as possible. Another thing which helps us to perform a smarter search is to prune some of the dead states before we expand them. This situation is called deadlock and happens when one or more box which are not on the goal position cannot be moved. Also we must be conscious about repeated states which can produce infinite loops if they are not taken into account. Furthermore we have exploited some facts about this game that for example the solver should

not go through the depth levels of the search tree and if such the situations happen we must cut the search and try other branches.

The rest of the paper organized as following: First in section 3, we will talk about our methods which are used to solve the problem. Then in section 4, we will mostly discuss our implementation points and results. Finally our conclusion in section 5 will end up our report.

3. Solving Method

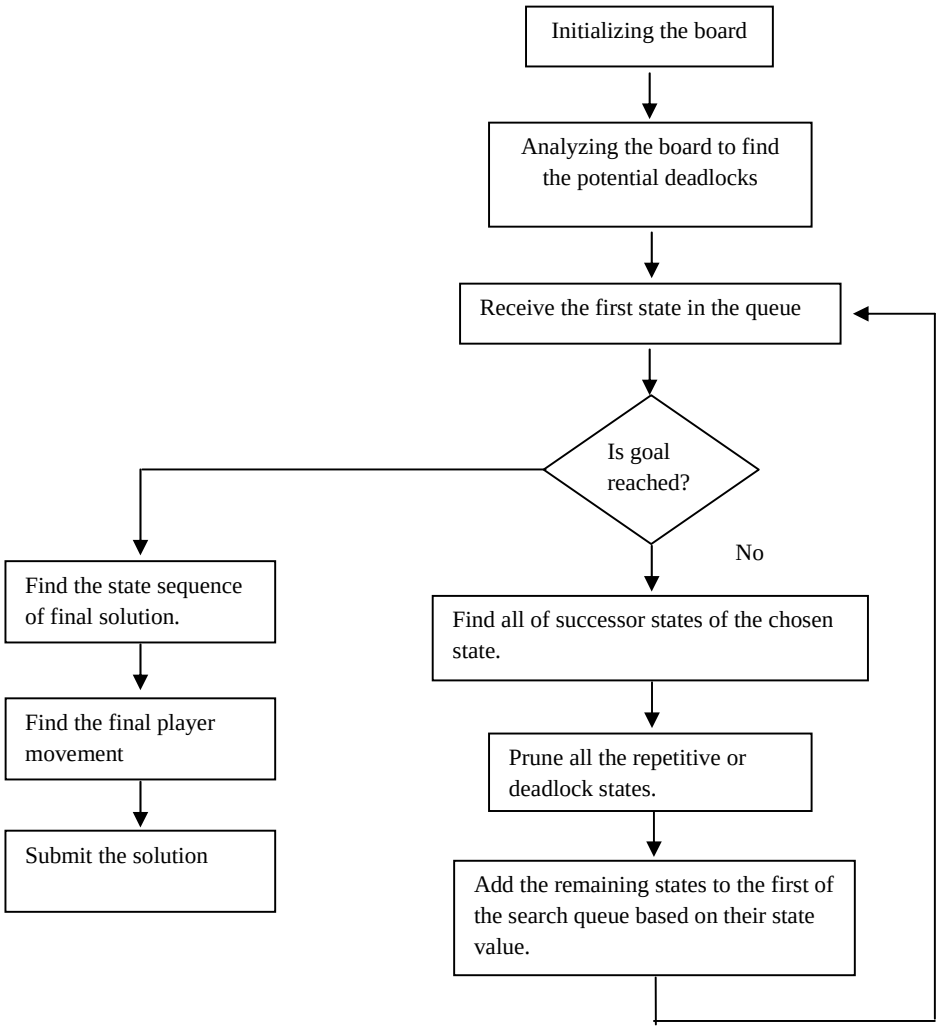
In order to find the solution of the game, we use the search methods to search through the whole states of the game to find the state sequences which lead to the final goal achievement. However as the solver proceeds, it will be enormous number of states which for the big boards with large number of boxes it is not possible to search all the states with the nowadays computers even during several days! Thus we need to use some techniques to reduce the search space to be able to find the solution in a reasonable time.

We used Best First Search method which can be assumed a special case of Depth First Search (DFS) to search through the state tree. The key point in this method is the use of heuristic function to evaluate states that is useful to choose which branch of state tree to search first.

The flowchart in the next page shows the main steps in our algorithm. In the following we will briefly described the different parts used in our flowchart.

Initializing the Board

In this step we will connect our system to the server to receive the new board as a stream of standard string input. Also it is possible to input the board manually. Then the received string is converted to our model of the Sokoban world. This model is used with the other functions to force the agent obey the game rule.



Initial Analysis of the Board

In this step the type I deadlocks are found based on a pre processing on the board. More information can be found under deadlocks topic in the following pages.

Receiving States

In our model of state tree, we have formed a linked list of the states which puts all the states in a queue. Since we are performing Depth First Search we have to put all the new states to the first of the queue and pick also from the first. (First In first out) We will talk about how to put coming states to the queue later. However the key point under this topic is the way that we pick the new state from the queue which is always the most recent state.

Goal Reached Test

This task is absolutely easy. We only need to check whether all the boxes are placed on goal positions or not.

Successor States

To construct the search tree, in each state we need to find all the possible successor states and put them in the tree branches to be in the search queue. The number of successor states depends on how many boxes exist in the board and what are all possible movements of the boxes based on the game limitations.

We have employed two types of possible successor states. In the first type which is the simpler one, we only consider one movement just for one box. But in the second type we go beyond that and consider as many as required movements for any of the boxes (Only one box can move in each turn).

For the successor states of type I, we first find all the possible next states by considering all the possible movements for all the boxes. Then prune the repetitive states or deadlocks, (It will be described with more details in the rest of the text) then all the remaining states will be evaluated and are given a value which is inversely proportional to how well is that state. The same story exists for the successor states of type II but with the difference that in this method we are trying to move each box individually to reach each of the goal positions. If it is possible then we will consider this state as one of the new successor states as well. Please note that new states which are generated in this way must also pass the deadlock and repetitive tests as well, otherwise they will be pruned.

Before that remained states can be added to our search queue, they need to pass the final test which control the solution length. It is important to know that the solution length cannot be longer than a roughly defined length. There might happen cases that solver spends a lot time to search by moving some of the boxes while leaving other boxes unchanged. Thus it will go through depth branches of the tree while the result would be just waste of time. To avoid this situation we consider a penalty term for each level that solver goes ahead. The state will be accepted or rejected based on the following condition:

$$\begin{cases} if\ state_{level} \times penalty_{cost} + state_{value} < T & Accepted \\ other\ wise & Rejected \end{cases}$$

Now both types of the successor states which survived from our conditions, are sorted based on their value and will be added to the first of our search queue.

Evaluate each state

To evaluate each state we need a measure of the distance between the boxes and the targets. We used the Manhattan distance. The core idea is to first determine which box should go to which target based on an analysis of distances. Then the Manhattan distance is calculated between each box and the target. Finally the sum of all distances is returned as the final state value. As an example, based on the following picture the distance between the box number 0 and the upper target (the red rectangles show the target positions) is equal to 5. Thus to find the value of this state, we try both conditions (first box to the first target and second box to the second target or first box to second target and second box to first target) and then find the case with minimum distance and consider it as the final value for the state.

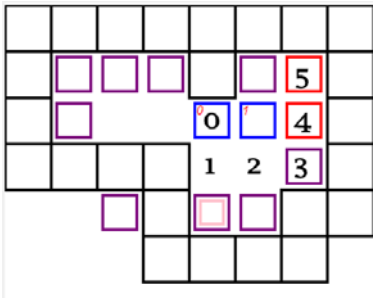


Figure 2) Manhattan distance

To be able to find the Manhattan distance we have employed the Flood Fill algorithm. The idea behind it is to release a flood stream from the target and lets it travel all the way and record the time as it reaches each cell and store it. If we just follow the time sequences backward, then we are able to find the shortest path if it exists.

To conclude the procedure used to achieve the state value consists of following two steps:

- Calculate for each box a vector of distances from each target.
- Find out the sequence of matching box/goal with the lowest value (meant as the sum of all the distances)

Deadlocks

A deadlock is a state with the configuration that makes the board unsolvable. Deadlocks are due to the limitation of just being able to push the boxes but never pull them. They are potentially very costly for the solver, because the search will be performed through the states which will not lead to a solution. These endless loops waste large amounts of resources.

We can distinguish two kinds of deadlocks. The first kind of deadlocks is related to the position of the boxes. There are positions in the board in which if any of the boxes enter these position a deadlock situation will be occurred. These positions can be found before the solver starts.

In order to find these types deadlocks, we first try to find the corners of the board. Then the path between those corners which have the same height or width are checked for continuous walls in any of the two sides. If such the wall is found and there is no goal position in the path then the whole path is marked as the deadlock type 1 and must be avoided by the solver. Following picture shows these deadlocks by purple rectangles.

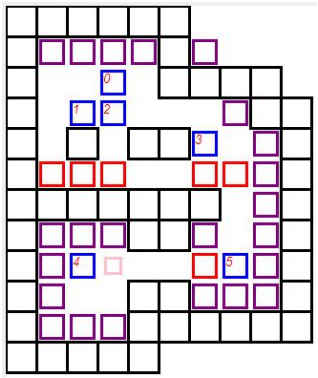


Figure 3) Deadlock founds based on initial analyze of the board shown with purple rectangles

The second type of the deadlocks happens due to bad configuration of the boxes. There exist situations in which a box will be trapped by the other boxes or the other boxes and the walls and result in a deadlock states.

The complexity involved in this problem is much more than the previous case and finding the real pattern of deadlock is much harder. It might happen situations that it seems a box is trapped by the

other boxes. However since the other boxes are free to move, this configuration is not a deadlock and must be considered in the search process.

We have considered both types of deadlock checker and our solver avoids searching the states with such the configurations.

Infinite Loops

During the tree search, if the solver visits an already encountered state, it is probable that the solver will enter an infinite loop. To avoid this problem, the solver needs to take note of every state it visited. For each generated new state, the solver looks up in its list whether it already visited this state or not. If such is the case, the solver prunes the state and therefore avoids entering the infinite loop. It is worth mentioning that we have implemented a bash table in order to reduce time complexity of searching through generated states.

4. Implementations and results

We have implemented two version of the work, one in C# and the other in Java. The aim for C# implementation was that we were mostly expert in coding in C# and it was much easier to design a visual output by the use of Microsoft Visual Studio 2008. On the other side, C# codes could be translated into Java quiet fast. Thus we could do our initial implementations in C# and then translate everything into Java. Our Software in C# is depicted in the following picture.

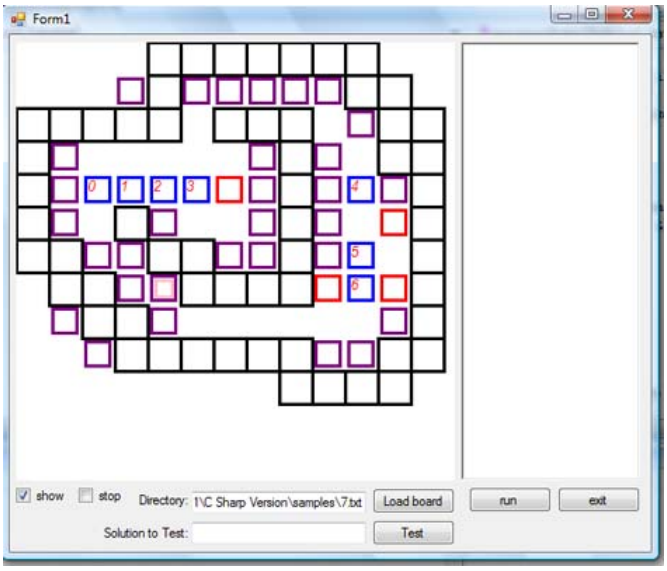


Figure 4) Our Designed Software

In our Java implementations, we have used 5 different classes as following:

Table 1 Java Classes

Class	Description
Client	Receives the boards, runs the main loop of the program and sends the output.
Sokoban	Most of the solver is implemented in this class
Flood Fill	Runs the flood fill algorithm when is necessary
State	State structure
Position	Position structure

In order to be able to present our results, we provide the actual time which is required to solve the boards in the server provided for the course ("130.237.218.85", 7777). Following table summarize our results:

Table 2 Performances

Board	Required Time(s)	Board	Required Time
1	0	8	>60
2	1	9	>60
3	2	10	>60
4	54	11	>60
5	>60		
6	9		
7	47		

As it can be inferred about our result, our solver could solve more than half of the boards in server where are actually with different level of difficulties.

5. Conclusions and further work

In this work an implementation of the Sokoban was presented. The main steps of the algorithm are described and as our results showed, we can say our solver is able to solve enough boards with a certain level of difficulties in a suitable time. It can be said this project is absolutely a great example of the search methods introduced in AI course.

The other aspect of this work was the ability to work in team. Time and resource management were the two other most important lessons which we learnt during this project.

For the future works, we can combine the methods in a more efficient way to be able to increase the performance of the solver. As an example, successor states type 2 were really helpful in some boards but were just waste of the time in another boards. If we could implement solver in a way that initially our system could run with successor states type 2, but after some seconds, if the system did not reach the solution, ignore them and just search based on successor type 1, then we believe that our solver could be improved at least to be able to solve 15% more boards. (With this combination we were able to solve board 5 and 11 in less than 5 seconds in result section)

6. References

1. http://en.wikipedia.org/wiki/Sokoban

2. Hordern, Edward (1986). Sliding Piece Puzzles (Recreations in Mathematics, No 4). p. 161