Sokoban Solver

Rasoul Mojtahedzadeh, rasoulm@kth.se
Soheil Damangir, damangir@kth.se
Saad Ullah Akram, saadua@kth.se
Xu Yuzhe, yuzhe@kth.se

***Abstract*— In this article the method implemented to solve the famous Japanese game Sokoban (pusher) is described. This is the project of the course Artificial Intelligence DD2380. The problem was divided into two milestones, Midterm and Final requirements. The algorithm, implementation, problems and issues, and finally advantages and disadvantages are described.**

I. INTRODUCTION

Sokoban is a type of one-player puzzle, in which the player should control the sokoban to push boxes in 2D maze, and try to put them into some requirement positions, goal positions. The sokoban could only push only one box at a time and not be able to pull a box. There is same number of goal positions in the maze as the boxes, see figure 1.

Solving Sokoban puzzles has been proved to be NP-hard and PSPACE-complete. It is difficult to solve not only due to its branching factor but also its enormous search tree depth. One of the main problems in solving a Sokoban game with artificial intelligence is that wrong moves may place boxes in deadlock condition. Even one box is in deadlock condition, it is impossible to move all boxes into goal positions, because the deadlock condition is an unrecoverable position form where a box cannot be pushed to previous position, for example a corner of the maze. Another problem is that it is hard to design a proper heuristic function to do a relevant move. In this article, it tries to find an efficient solver which can solve Sokoban problem in finite steps automatically using a single-agent search algorithm.

II. MIDTERM MILESTONE

1. Algorithm

In Sokoban, the aim is to place all boxes into the goal positions. At any time, the player only can control Sokoban to move in four different directions, Up, Down, Left and Right. So the solution of the game is a sequence of the moves which pushes all boxes to the goals. The simple solver could return a list of moves using breath-first or depth-first search which focuses on motions of Sokoban.

To improve the performance, we add two important strategies to reduce the computational complexity:

A. Avoid repeated states

B. Avoid deadlock conditions

Strategy A will prevent search tree from infinity depth. The repeated states can be easily detected and avoid by comparing present state with that states stored in a history record. The second strategy can prune the search tree by eliminating deadlock positions from the search.

Note that it is same difficult to detect all possible deadlock position as to find the solution to the game. However, some of the deadlock positions can be formulated simply. In the midterm, it just labels the corners of maze as the deadlock position and prevents sokoban from pushing the boxes into these deadlocks.

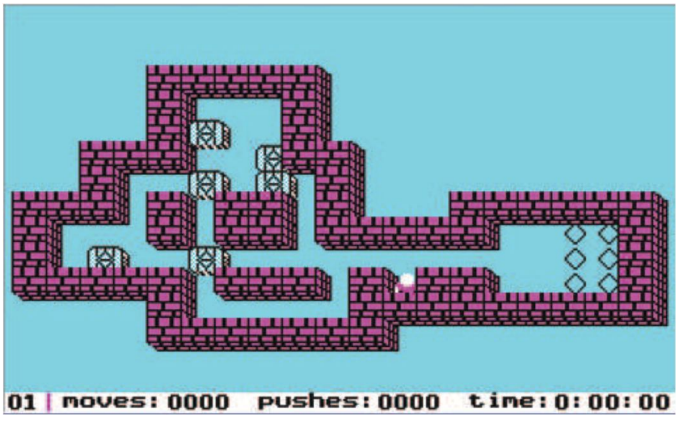

Figure 1: A Sokoban sample board.

2. Problems

Firstly, the simple solver can just handle some simplest Sokoban games because of the vast dimensions of the tree. For example the solution sequence of a media difficult game may contain about hundreds moves. It means that a complete search in a tree with branching factor of about 4 to a depth of more than 100 is certainly out of the question even avoiding the deadlock in searching.

Secondly, the size of the history recodes of the previous states would increase when the complex of the maze increases.

Worse, simple deadlock detecting just can prevent sokoban from moving into simplest deadlock positions. There are many other static deadlocks formed by well and more dynamic deadlock conditions which formed by the boxes in Sokoban games. The simple detector cannot detect them and prevent sokoban from pushing boxes into them.

3. Pseudo Codes

function Sokoban-Solver (maze) **returns** seq
  inputs: maze
  static: seq, an move sequence, initially empty
    rept_state, the collection of all previous mazes
    maze, the maze of the game, including positions of boxes, sokoban and goal positions
  if goal-test(maze) is false **then do**
    if lock_test(maze) is false **and** rept_test(maze) is false **then do**
      maze ←do_move(maze, move)
      seq ←add_moves(seq,move)
      rept_state←add_state(rept_state,maze)
    else
      rept_state←add_state(rept_stae,maze)
      seq,maze←pre_moves(seq,maze)
    end
  end
return seq

The goal_test function tests whether all the boxes in the goal positions. And the lock_test function tests whether any box in the deadlock position, while the rept_test function tests if the present maze already exists in storage of the states.

The do_move function returns the new maze after the sokoban moving one step.

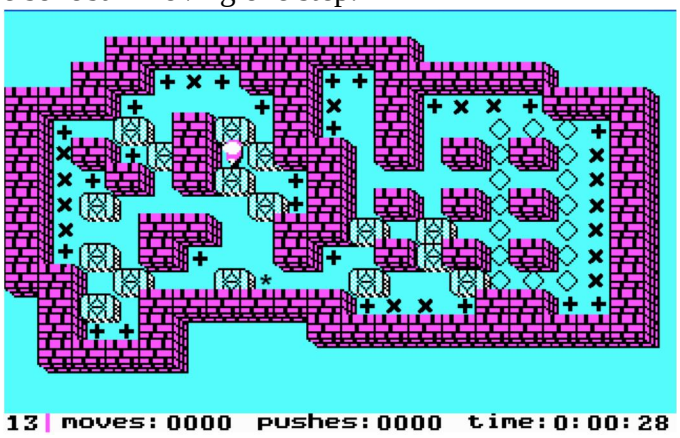

Figure 2: Static deadlocks are marked with (x, +)

4. Pros and Cons

The simple solver can solve the simplest Sokoban game within 1 minute. But it might takes years to solve more complex game.

III. Final Milestone

1. Algorithm

We tried to solve the problem by two different approaches. Although the first method seemed to be attractive to follow, and partially implementation of this method managed to solve some boards at once, but unfortunately we encountered some run-time errors which needed more time to debug. The second approach was an implementation of so-called A-star search algorithm with reverse solving of the problem. It means that we put the boxes in the goals and made Sokoban to pull the boxes to the initial positions instead of pushing them. The major advantage of pulling is to face any dynamic deadlocks position much sooner than solving the problem by pushing.

In the first approach, all goals get a priority number. This priorities are calculated just once because all the walls and the goals positions are static and don't change during the game. To this, all boxes except one box are deleted, and then all two combinations of goals are considered. One of these two goals becomes a wall and Sokoban tries to push the box to another goal and vice versa. If there is one box which can be put in the goal, this goal has no higher priority in respect with another one. If

not, the other goal has higher priority than this one. This procedure is done for all other boxes and combinations of goals. Finally, we will have an array of goal positions and their related priority. Then, the algorithm tries to push boxes to the goal with highest priority. We call unoccupied squares as *static deadlocks* if Sokoban cannot push a box from that square to any goal position. These are static deadlocks, because the walls and goal position would not change during the game. See figure 2.

In second strategy, a multi-agent realization of the problem was considered. The algorithm generates the box-moves instead of Sokoban possible moves and searches on all possible box-moves that Sokoban can reach at *Proper Square* to push or pull the box. Proper squares are those in which Sokoban can push or pull the boxes. For example the left reachable square of a box that Sokoban should first go to, and then push the box to the right is a proper square. These squares are determined by some implementation of the *flood-fill algorithm*. It is originally developed to determine which parts of a bitmap to fill with color in Paint programs.

One major advantage of solving the problem is that the depth of search to find the goal terminal will be reduced considerably, whereas the branching factor will increase from 3 (for Sokoban) to number of possible box moves.

Because Sokoban can walk in four directions if the destination is not occupied by a box or wall, hence the flood-fill algorithm generates squares that Sokoban can reach by only walking in the given board level.

The search to find a solution is based on famous A-star search algorithm. The node-cost $g(n)$ is the number of displacement of the box (and not Sokoban). The architecture of multi-agent reduces the search space by considering all possible Sokoban walking moves in a level, as one state. This is the first advantage of the multi-agent architecture.

A heuristic function $h(n)$ estimates the cost from the current node n to the goal terminal. To this, we used two different approaches. At first, the simple chess-board distance between boxes and goals was calculated:

$$d_{ij} = \left|x_b^i - x_g^j\right| + \left|y_b^i - y_g^j\right|$$

Where x_b^i indicates the x-position of the box number i. The minimum distance is the scale used for this heuristic function, see figure 3.

Although this method managed to solve 60 percent of the boards, but the convergence time was not efficient. It should be mentioned that the heuristic function must be an underestimate of cost to reach to the goal terminal.

The second design to calculate the heuristic value was based on a *reachability matrix*. It is a three-dimensional matrix which relates each square in the board to each goal. To simplify the notation, we show each square coordinate (x_i, y_i) with the point P_i,

$$\begin{bmatrix} r_{11} & \cdots & r_{1m} \\ \vdots & \ddots & \vdots \\ r_{N1} & \cdots & r_{Nm} \end{bmatrix}$$

where r_{ij} illustrates the reachability of point P_i to the goal $g_j$as well as the distance between them. This distance is calculated by removing all boxes from the board, therefore it is an underestimation of the cost. To end, minimizing this cost (distance) is the principle of A-star search used in the solver.

One improvement in the search algorithm was to freeze a box when it reaches at a goal and open a new node in the search tree at this point. Although it increases the branching factor by one, but it is more probable to find the goal terminal in the children of this new opened node. The reason is that in the goal terminal, each box is located to one goal position which intuitively led us to use it in the search. On the other hand, the nodes which are searched from this frozen node are part of the search tree which will not be visited again. Hence, although it seems to increase the search space, but in the worst-case it is equal to adhere to the original A-star algorithm.

To detect repeated states (board levels), a hash table based on DJB2 algorithm was implemented. This method originally was not proper for Sokoban

problem mostly because the difference between two boards can be just displacement of one box. Therefore, some modification to the hashing algorithm was performed.

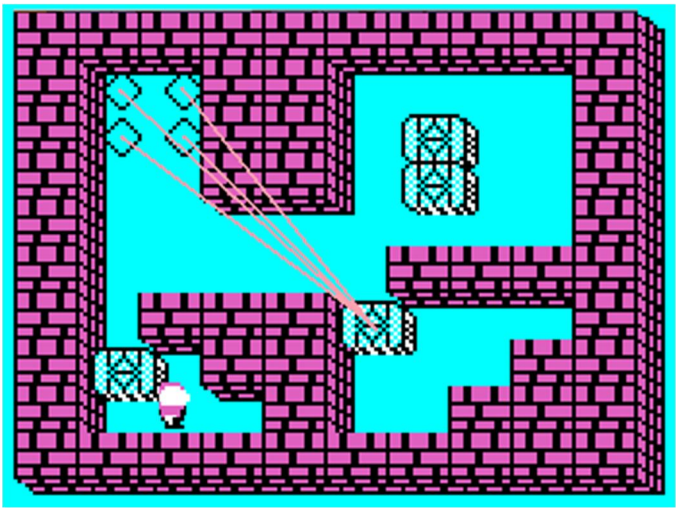

Figure 3: Average distance of the box to goals.

2. Problems

One problem with reverse solving of the problem (pulling) is that Sokoban might be able to find a solution to pull all the boxes to their initial positions, but if Sokoban cannot find a way to back to its initial position, this is a wrong solution! It means that although Sokoban managed to pull all the boxes to the goals, but it is also trapped among boxes and probably walls.

3. Pros and Cons

The program can find the solution for boards with equally distributed goal priorities quite well. The A-star search with some optimization in queue lists is the heart of the program which was implemented well. Another profit of the program is the reverse solving (pulling) approach which prevents dealing with dynamic deadlocks. Using flood-fill algorithm to find the Sokoban walking moves towards the boxes, is another strong point of the method to reduce the search time.

On the other side, the program suffers from a simple heuristic function which is not optimum for all boards.

IV. Conclusion

Although the rules of the game seem to be simple and static property of board (walls and goals) are tempting at the beginning, but finding a Sokoban solver algorithm to solve all the boards in reasonable time is a hard problem. Especially trying to solve the problem in forward mode (pushing) require to solve the problem "*How to detect dynamic deadlocks?*" which is not easier than the problem itself. However, manipulating different methods we learned in Artificial Intelligence to solve this problem showed that designing an intelligent agent could be very difficult.

References

[1] Frank Takes, "*Sokoban: Reversed Solving*", Leiden Institute of Advanced Computer Science (LIACS), Leiden University, June 20, 2008.