Iterative Deepening A*, Recursive Best First Search, and Monte-Carlo Tree Search for Sokoban Games

Dongkyu Kim / Yeojin Kim / Evan Steelee

Abstract

As the final project, we implemented IDA*, RBFS and MCTS in order to solve Sokoban game which is the well-known single player game. Sokoban game is a big challenge in artificial intelligence study due to its problem complexity. So, we had to apply heuristic functions and deadlock detectors. All algorithms solved problems, but their performances are, of course, different. The result of this experiment shows the possibility to improve algorithms further by considering specific conditions of the environment of the game.

I. Introduction

In this project, we tried to solve Sokoban game with IDA* and RBFS with Manhattan Distance and Hungarian Method, and Monte Carlo Tree Search. Automated Sokoban game solver is a big challenge in AI because it is very difficult to calculate branching factor of each state due to deadlocks. They are changing every state, but agent's options to move is not easy to guess. Thus, Sokoban is NP-hard problem, significantly much difficult than NP-Problems; it is PSAPCE-complete. In this project, we investigate MCTS can be an interesting alternative for IDA* and RBFS for a single-player game, Sokoban. In the second section, the brief introduction of Sokoban game and the implementation of algorithms and functions to solve Sokoban game. In the third section, the result of experiments will be provided, and the implication and the future work of the experiment will be discussed in Conclusion and Discussion.

II. Sokoban Game and Implementation

1. Sokoban Game

Sokoban (倉庫番 *sōko-ban*, "warehouse keeper") is a type of puzzle video game, in which the player pushes crates or boxes around in a warehouse, trying to get them to storage locations. Sokoban was created in 1981 by Hiroyuki Imabayashi, and published in December 1982 by Thinking Rabbit, a software house based in Takarazuka, Japan [1].

The game is played on a board of squares, where each square is a floor or a wall. Some floor squares contain boxes, and some floor squares are marked as storage locations. The player is confined to the board and may move horizontally or vertically onto empty squares (never through walls or boxes). The player can also move into a box, which pushes it into the square beyond. Boxes may not be pushed into other boxes or walls, and they cannot be pulled. The

number of boxes is equal to the number of storage locations. The puzzle is solved when all boxes are at storage locations [1].

Sokoban game is a single player game with fully-observable, deterministic, sequential, static, and discrete environment.

2. Implementation

The purpose of this experiment is applying Monte Carlo Tree Search(MCTS) to Sokoban game to compare the performance of Iterative-Deepening A*(IDA*) and Recursive Best First Search(RBFS) with heuristic functions to calculate the distance from the goal.

Classicial methods such as A* and IDA* are a popular and successful choice for single-player games. However, they fail without an accurate admissible evaluation function. In this project, we investigate MCTS can be an interesting alternative for IDA* and RBFS for a single-player game, Sokoban.

1) IDA*

Applying the iteration to A* search is the simplest way to reduce memory requirements. As a result, the IDA* algorithm can solve the memory issue. Instead of using depth in standard iterative deepening, IDA* uses the f-cost(g+h) as the cutoff at each iteration. The pseudocode of IDA* search algorithm is as following:

```
// Pseudocode for IDA*
function IDA*(node 'n', depth 'd', bound 'b')
{
        if (f value of node 'n' > bound 'b')
        return f value of node 'n';
        if node 'n' is goal
        return 'found';

        minimum = max value;
        for each child 'ni' in successors{
        'returned value' = IDA*('ni', 'd' + 1, b);
        if ('returned value' is 'found')
                return found;
        if ('returned value' is smaller than minimum)
                mimimum = 'returned value';
        }
        return minimum;
}
```

2) RBFS

RBFS is a simple recursive algorithm that improves upon heuristic search by reducing the memory requirement. RBFS uses only linear space and it attempts to mimic the operation of standard best-first search. Its structure is similar to recursive depth-first search but it doesn't continue indefinitely down the current path, the f_limit variable is used to keep track of the f-value of the best alternative path available from an ancestor of the current node. RBFS remembers the f-value of the best leaf in the forgotten subtree and can decide whether it is worth re-expanding the tree later. However, RBFS still suffers from excessive node regeneration. The pseudocode of the RBFS algorithm is as following:

```
// Pseudocode for RBFS
function RBFS( node n, depth, f-limit)
{
        if ( node n is goal)
        solution  node n;
        else
        successors = expand(n);

        if (successor is empty)
        return max value;

        for each child 'ni' in successors{
        if (F('ni') is larget than f - limit of 'n')
                F('ni') = max(F('ni'), F('ni'))
        else
                F('ni') = f value of 'ni'
        }

        sort(successors, successors + size()
        n1 = best;
        n2 = alternative;

        while (f value of 'n1' < f - lilmit and F(n1) < max value) {
        n1 = RBFS(n1, min(f - limit, F(n2)));
        sort(successors, successors+size());
        n1 = best;
        n2 = alternative;
        }

        return F(n1);
}
```

3) Heuristic Functions

We implemented two heuristic functions, Manhattan-Distance, and Hungarian Method, for Iterative-Deepening A* and Recursive Best First Search.

(a) Manhattan Distance

We use the fundamental distance heuristic, Manhattan Distance, which is based on the grid-like street geography. In this heuristic, we obtain the Manhattan distance from each box positions to the nearest storage position and sum the total distance. The minimum distance box must move cannot be Manhattan Distance. This heuristic is non-overestimating. But, the limitation of this method is that two boxes can be allocated to the one storage. As shown in Figure 1 below, the box at (7,1) and the box at (8,3) are matched to the same storage at (8,2).

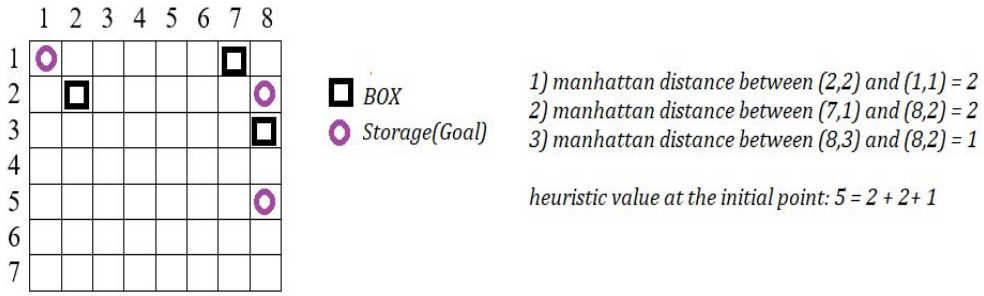

Figure 1. Manhattan Distance Heuristic Value

(b) Hungarian Method

As an alternative to the above assignment problem, we introduce the Hungarian method. The Hungarian method is a combinatorial optimization algorithm that solves the assignment problem in polynomial time and which anticipated later primal-dual methods. It is in O(N^3), where N is the number of boxes to be assigned. This method is fundamentally based on the Greedy algorithm.

4) MCTS

MCTS is a method for making optimal decisions in artificial intelligence (AI) problems, typically move planning in combinatorial games. It combines the generality of random simulation with the precision of tree search.

```
def UCT(rootstate, itermax, simple_dead_pos):

    root_node = MCTS_node(this_board=rootstate)

    for i in range(itermax):
        node = root_node

        # select
        while node.untried_moves == []:
            node = node.select_child()

        # expand
        if node.untried_moves != []:
            move = random.choice(node.untried_moves)
            node = node.add_child(move, curr_state.current_board,
simple_dead_pos)

        # play out
        while random_child != []:
```

```
            random_child =
random.choice(state_calculation.getMoves(state_calculation.current_board,
simple_dead_pos))

        # backpropagate
        while node != None :
            if simple_deadlock(node.current_board, simple_dead_pos):
                node.win += -3.0
                node.t = node.visits
            elif board.freeze_deadlock(node.current_board):
                node.win += -1.0
                node.t = node.visits
            else:
                node.win += 0.5
                node.t = node.visits
            node = node.parent

    return sorted(root_node.children, key=lambda c: c.visits)[-1]
```

Pseudocode for MCTS

(a) A variation in the selection strategy for Single-Play Sokoban

We require a modified UCT equation for single player from the original version of the two-player games. In the equation (1) below, the first two terms come from the original two-player games. The first term is exploitation term, which simply means the average win rate. In two-player games, the results of a game is denoted by loss, draw, or win, i.e., {-1, 0, 1}. In one play game, an arbitrary score can be achieved. In our project, we give weighted win value. When the node is reached to simple deadlock states, we give the node '-3' as the reward. When the node is reached to freeze deadlock states, we give the node '-1' as the reward. When no deadlock is detected, we give the node '0.5' as the reward. The second term is exploration term and it goes larger the less frequently a node is selected. In the third term, $\sum v_i^2$ is the sum of squared rewards obtained by rollouts that have transited through node n_i and $n_i \times (\frac{v_i}{n_i})^2$ correspond to the expected squared reward.

UCT variation in the selection strategy is shown below [2]:

$$SP-UCT = \frac{v_i}{n_i} + C\times\sqrt{\frac{2\,lnn_p}{n_i}} + \sqrt{\frac{\sum v_i^2 - n_i\times\left(\frac{v_i}{n_i}\right)^2 + D}{n_i}} \quad \text{--- Equation(1)}$$

,where v_i : the reward value for child node i

n_i : the number of times child node i visited(simulated)

n_p : the number of times parent node p visited(simulated)
C: a constant for exploration to adjust the amount of exploration performed
D: a constant to ensure that rarely explored nodes are considered uncertain

(b) Deadlock Situations

We find that the agent can never push the box to the storage position, in which a box in a level is called dead. We call the grid position deadlock square. This means that the level is not solvable anymore, no matter what the agent does. The only way to solve this level is to undo a movement. There are several deadlock types which are already known. In our implementation, we tried to solve two deadlock type, simple deadlock and freeze deadlock.

i. Simple Deadlock

In Figure 2, if the box is positioned at the darker shaded grid, the box cannot get out of the position and can never reach the storage position.

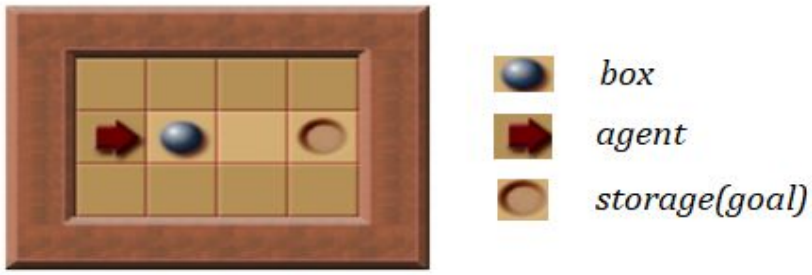

Figure 2. Simple Deadlock [3]

We solve this deadlock by introducing the simple deadlock function. Within the simple deadlock, we check if there is a corner deadlock and edge deadlock. We can compute the simple deadlock squares(dead position) from the original board and we have the deadlock square information as a hash table.

```
def corner_deadlock(board):

  dead_position = []

  for row in range(1, len(board)):
    for column in range(1, len(board[row])):
      if board[row][column] == "3":
        if check corner square of 'board' :
          dead_position.append(row)
          dead_position.append(column)

  return dead_position
```

Figure 3. Pesudocode for Corner Deadlock

```
def edge_deadlock(board):

    dead_position = []

    for row in range(1, len(board)):
        for column in range(1, len(board[row])):
            if edge deadlock state of 'board' :
                dead_position.append(row)
                dead_position.append(column)

    return dead_position
```

Figure 4. Pesudocode for Edge Deadlock

ii. Freeze Deadlock

Sometimes, boxes become immovable because of consecutive boxes or protruded walls. We check this deadlock type at every step.

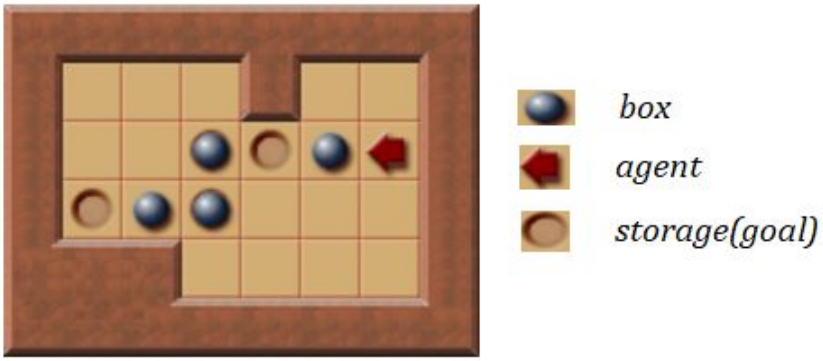

Figure 5. Freeze Deadlock [3]

III. Results

Below are the results of cases with 2 to 4 boxes.

```
# -1 == Wall (nothing moves past!)
# 0 == Goal (nothing on it at all)
# 1 == Goal (agent occupying goal square)
# 2 == Goal (box occupying goal square)
# 3 == Box (a sad, lonely box)
# 4 == Agent (it's us!)
# 5 == Empty (square with nothing but potential)
```

1. The number of steps of 2 Box Case

-1	-1	-1	-1	-1	-1	-1
-1	5	-1	5	5	-1	-1
-1	5	5	3	0	-1	-1
-1	5	5	5	5	5	-1
-1	5	0	3	4	5	-1
-1	-1	5	5	5	5	-1
-1	-1	-1	-1	-1	-1	-1

	IDA*	RBFS	MCTS (Iteration)
Manhattan	284	250	Max: 6338 (249) Min: 229 (6 Avg.: 1974 (71)
Hungarian Method	290	260	

2. The number of steps of 3 Box Case

-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	5	5	-1	-1
-1	-1	5	5	0	-1	-1
-1	5	5	3	5	5	-1
-1	5	2	5	2	5	-1
-1	-1	-1	5	5	4	-1
-1	-1	-1	-1	-1	-1	-1

	IDA*	RBFS	MCTS (Iteration)
Manhattan	525	491	Max: 47532 (274) Min: 40 (2) Avg.: 3485 (132)
Hungarian Method	490	457	

3. The number of steps of 4 Box Case

-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	5	5	-1	-1
-1	-1	5	5	0	-1	-1
-1	5	5	3	5	5	-1
-1	5	2	5	2	5	-1
-1	-1	-1	5	5	4	-1
-1	-1	-1	-1	-1	-1	-1

	IDA*	RBFS	MCTS (Iteration)
Manhattan	11719	14655	Max: 36980 (1404) Min: 233 (13) Avg.: 12364 (464)
Hungarian Method	3089	3974	

IV. Conclusion and Discussion

When it comes to IDA* and RBFS, our results scale as expected with more complicated boards. IDA* seems to require more steps than RBFS for 2 and 3 boxes cases, but RBFS requires more steps in 4 boxes case. With respect to the heuristic algorithm, the Hungarian Method performs better as a more accurate analysis of how close a board is to being a finished puzzle. The heuristic evaluations are difficult for the game since moves can be made where the heuristic evaluations from the boxes do not change, like when an agent moves from one empty square to another empty square.

MCTS requires more steps, but the term, step, in MCTS is including steps of simulation until it reaches a leaf node. Therefore, we need to see the number of iteration to the root node for the performance measurement as well. In respect to it, we can expect the result of MCTS to be the alternative of IDA* and RBFS.

Nevertheless, it still has the unexplored area for improvement. In this project, we have not applied deadlock detector to IDA* and RBFS. We believe deadlock detector help to avoid dead position and reduce search steps efficiently with heuristic functions, and it will improve the performance of IDA* and RBFS. Also, SP-UTC can be tuned with calculating the location of boxes. Bidirectional Search also can be applied because an agent holds the information of goal

locations calculated by heuristics. Of course, more accurate deadlock detector will absolutely improve overall performances.

References

[1] “Sokoban,” *Wikipedia*, 02-Mar-2019. [Online]. Available: https://en.wikipedia.org/wiki/Sokoban. [Accessed: 16-Mar-2019].

[2] M. P. D. Schadd, M. H. M. Winands, H. J. V. D. Herik, G. M. J.-B. Chaslot, and J. W. H. M. Uiterwijk, “Single-Player Monte-Carlo Tree Search,” Computers and Games Lecture Notes in Computer Science, pp. 1–12, 2008.

[3] “Sokoban Wiki,” *Sokoban Wiki RSS*. [Online]. Available: http://sokobano.de/wiki/index.php?title=Main_Page. [Accessed: 16-Mar-2019].