CSD311: Artificial Intelligence

Project Report

Team 30: SokoRamp

Prof. Harish Chandra Karnick

Hemanth Sai Boyapati - 2010110980

Saran Bodduluri - 2010110193

Mohana Siddhartha Chivukula - 2010111021

Appala Avinash - 2010110121

Katakam Saideep Reddy - 201011034

Introduction:

Sokoban is a classic puzzle game invented in Japan. The original game of SOKOBAN was written by Hiroyuki Imabayashi. Today Sokoban is one of the most popular thought and logic games. As simple as that name is, so is the idea of this game: A level represents a store room, where boxes appear to be randomly placed. You help the warehouse keeper to push the boxes around the maze of the room so that, at the end, all boxes are on marked fields. The only restrictions are that the warehouse keeper can only push a box, never pull, and that just one box may be pushed at a time.

Problem Analysis:

• Sokoban is played in an enclosed area.

• The player can only move one square at a time in the four directions N, E, W,S.

• No player or stone can go through a wall or others stones in a move.

• The game ends when all stones are in goal positions. A stone can occupy any goal position.

• There are multiple tricky states the game can enter into while doing performing a search.

Solver Ideation:

We have initially started out by constructing the board, its representations, and writing conditions for valid moves. Then we moved onto set the deadlock conditions and performing rudimentary uninformed search algorithms like DFS, BFS. Then we moved onto informed algorithms like greedy A* with different heuristics, starting with Manhattan, and Euclidean, and finally moving onto a new approach based on the goal pull distance. Then we tried to improve the running time by making use of Fibonacci heap, numpy arrays and scipy library for solving the assignment problem in the goal pull heuristic.

Data Structures used:

We used a list for taking in the levels from the given input xsb file. Each line in the given level will be represented as a string in the list. When we move into the Classes of Sokoban and State, the coordinates of all the objects, including walls, players and goals, are represented in list of tuples. While expanding the nodes in our heuristic of A*, we used a set for maintaining the visited nodes. We used a Fibonacci heap for maintain the open set of nodes to be expanded.

Goal Ordering:

One of the critical observations to be made is that it is necessary for certain levels to not stop after a box reaches a goal, it can be destined to another goal. The order in which boxes go into their goals is crucial and we made sure no error arises from the goal ordering.

Deadlock conditions used:

Deadlocks are an important aspect to be taken care of in a state space problem such as sokoban. A box is in a deadlock position if a box placed can never be pushed into the goal. The deadlock conditions we considered are as follows:

• Block is present in the corner. There is no next move possible.

• The block present in the edges. When the block is present on an edge, the only reason to be moved is to go to the end which creates the above situation.

Algorithms:

We primarily wanted tried to use various searching methods like BFS, DFS, greedy A*. A∗ starts at the goal and moves in the direction of the initial state, searching for the optimal solution. A∗ algorithm requires a heuristic that estimates the cost from the current state to the initial state. On reading the results of (Huaxuan, 2018), we found that (search algorithms like) A* is more optimal than reinforcement learning. The difference in performance only arises because of the quality of the heuristic we choose. One thing we realised is that as we try to solve harder levels, it is more about having better-designed heuristics and ways to resolve deadlocks.

We used the Euclidean, Manhattan, and Goal Pull distance heuristics. The first two only worked on simpler levels with about 2 or 3 goals. They stopped working or rather started taking too long to solve levels as they got harder. We also tried a heuristic heavily based on the paper (Froleyks, 2016) which is the goal pull heuristic. “Goal pull distance seems to be a better solution for the problem. It is calculated as follows. We calculate the distance a box has to be pushed from each square to the goal if no other boxes were present and the player could reach every part of the level. This is done by using a breadth first search algorithm to pull a box from the goal i.e., checking for each of the four cardinal directions whether a box placed on the square in that direction could be pushed onto the goal. These squares are then marked with their distance to the current goal and we continue by checking their adjacent squares. This is done for every goal. An important point to be noted is that the distance from the boxes from which the goal cannot be reached will be regarded as infinity as it is not necessary that all the boxes can contribute to the solution”.

This goal pull is the pre-computation we do before actually calculating the heuristic. The heuristic is about finding the minimal perfect matching in a given bipartite graph which is called the assignment problem. Here the two vertex sets of the bipartite graphs are the set of

goals and set of boxes coordinates with edges connecting with the weights calculated through the goal pull algorithm. For this we used a function that was available in the scipy library called linear_sum_assignment, which will essentially do the assignment for us using the Hungarian algorithm.

Code Flow:

The new_game reads the input level and identifies the walls, goals, player, empty tiles, blocks. Deadlock situations are also identified and goalpull is applied. The search object calls the required search either Astar, or DFS.

The input is in the form of an xsb file with different. After every newline, a new level is taken into the list. For every level, a new game is created from Class Sokoban and search is implemented. The state class tests the corners and boundaries and then adds the deadlocks. For every move, the valid moves are generated, and the children are generated. We can change the algorithm and heuristic according to our requirements.

Optimization of memory use:

We initially included the variables storing the walls, goals, empty tiles, and the dist_to_goal, which stores the distance of every other cell from the goal, inside the state class. And these variables were getting created for every object, and hence to optimize the memory usage, we declared them outside the State Class, as global variables, which helped very much. The space the program was taking while running was drastically reduced. Since the level 6, the hardest of all in the given test examples, is expanding enormously many nodes, the memory usage had to be optimal. For us, level 6 was running only after making this change.

Our code on test levels:

Our test cases were tested from the micro sokoban which has about 50 examples and on our brief testing with a max time limit (proceed to next test case if time exceeded) our algorithm could comfortably run about 30 levels.

	Time(sec)	Heuristic used
Case 1	1.43	Goal Pull with greedy heuristic
Case 2	1.47	Goal Pull with greedy heuristic
Case 3	15.18	Goal Pull with greedy heuristic
Case 4	6.02	Goal Pull with greedy heuristic
Case 5	60.70	Goal Pull with greedy heuristic
Case 6	477.74	Goal Pull with greedy heuristic

References

- Nils, Froleyks "Using an Algorithm Portfolio to Solve Sokoban", December 28, 2016 https://baldur.iti.kit.edu/theses/SokobanPortfolio.pdf, accessed during October 2022.
- Sokoban Wiki http://sokobano.de/wiki/index.php?title=Solver_Statistics
- X sokoban https://www.cs.cornell.edu/andru/xsokoban.html
- Stanford papers http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html
- scipy function from liearn sum assignment

https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.linear_sum_assignment.html