Sokoban Solver

Introduction

Sokoban is **P-space** complete problem, which means it can be solved using an amount of memory that is polynomial in space. A Sokoban level contains walls, boxes, and storage locations (**goal**). Goal of the game is to push the boxes into the storage locations. The rules for this game are very simple to understand, but in some difficult levels, it is hard for human to complete due to the large sum of calculation. Therefore, we try to implement algorithms to solve this game. The main effort is to search the space in an efficient manner.

Problem Analysis

The constrains that the player must follow –

- The player cannot move two or more boxes at a time.
- The player should avoid pushing the box into a deadlock position where a deadlock state is defined as unsolvable state.
- If the player wishes to push the box downward, there must be a path to the upper side of the box and a space on the lower side of the box.
- The final score is determined by the number of steps taken by the player. The higher the cost and thus the lower the score, the more steps the player takes.

Representation

- ### Game representation -
  We are representing walls, boxes and goals as hash set of tuples where each tuple is a coordinate, and we took the position of player a separate tuple. We also calculate static deadlock positions on the grid and store them as a hash set of tuples.

- ### State representation -
  We store the state of the game using the position of boxes, player, and the path that we took to reach this state. Since we store the states in a set during search, we had to construct a custom hash function for the State class, referenced from [2] which constructs a hash based on the normalized positions of the player and the boxes.

Algorithm and Heuristics used

We used two algorithms and 3 different heuristic –

Three Algorithms –

- A*
- Greedy
- IDA*

4 heuristics –

- Manhattan
- Euclidean
- Goal Pull
- Min-Match

Heuristics

We have started with Euclidean distance as it returns a lower approximation to the goal and acts as a good baseline to check the performance of other heuristics. Then we moved on to Manhattan which gave a better performance compared to Euclidean as it gives a closer approximation to the goal. The variants we have tried for Manhattan and Euclidean are:

- Player position to all the other box position + Box positions to all the other goal positions
- Just box positions to all the other goal positions
- Player position to the nearest box position + Box positions to all their nearest goal positions

All the variants are intuitive in the sense that we are trying to make the player reach a box and for the boxes to be pushed into goal positions. While there was a slight difference in the results on our custom dataset, there was no considerable difference to say one performed better than the other. Finally, using Manhattan could only give suboptimal results as it did not run-on bigger test cases or have been running indefinitely.

The reason they were not giving good results is that they do not consider the actual path that box has to travel to reach the node, the obstacles present and the valid moves that can be made. The next heuristic that we tried considering this limitation is the Goal-Pull heuristic referencing from [1], which calculates how far a box must be pushed from each square to reach a goal if no other boxes were present and the player could reach every part of the level. We accomplish this by using a BFS to pull a box from a goal. Then the squares are then labelled with their distance from the current goal, and we proceed by inspecting their neighboring squares. It is done for each goal. This we termed as Goal-Pull-Distance and stored it in a dictionary containing keys as goal tuples and values containing another dictionary with all the other positions as keys and the distance between that goal and position as the value. This dictionary of dictionaries is calculated right when the board is initially loaded so the values can be accessed in constant time when state changes. The final calculations involve summing up the distances from all goals to all box positions for a given state from the dictionary.

While the Goal-Pull did perform better there was not a considerate amount of improvement. The next heuristic “Min-Match” which gave us the best results so far, gives a matching set that approximates matches of goals and boxes in a greedy manner to get a lower bound for the number of moves. Formally this minimal perfect matching in a bipartite set is called the assignment problem and we have implemented the pseudo code referenced from [1] in our solver.

Deadlocks

A deadlock position is a cell where if we push a box to that cell there is no possibility of it being able to reach any goal cell. We have implemented two basic deadlock techniques to prune the search tree.

- **Corner deadlocks**: They are the corner cells, having a wall each on at-least two adjacent sides of a box.
- **Boundary deadlocks**: They are the cells that have at-least one wall touching them and hence if a box is pushed there its motion is restricted to only one direction, and if a goal cell is not present in that direction, then it is basically a dead state.

Observations on Heuristics

We have applied the above discussed heuristics on A*, IDA*, and Greedy search algorithms. While we know that IDA* gives complete and optimal solutions, it was taking a lot of time to run on simpler test cases. A* did give better performance in time and gave optimal paths compared to Greedy. The implementations can be seen in our submitted code. Greedy worked the fastest and this is what we have shown our results on. In search code, we have used Priority Queue for storing the not visited states and a set for storing the visited states so that they will not be searched again.

Future Scope

- **Dynamic deadlocks** – As new deadlocks can be generated after every player move, we could calculate such positions beforehand and avoiding going to those cells further prunes the search tree.
- We could use python libraries like **Scipy** to calculate min-match value.
- As we referred from sources [1] Q-Learning can only solve 10*10 grid game under compute constrains so we did not explore that approach but we can use other reinforcement learning techniques which has lower time complexity.

Results obtained using greedy min-match

Testcase	**Time(ms)**	**Path**	**Nodes Expanded**
1	1068.09	uuRurrdrrUUdlllldldddrrUdlluRluuurrrrrruruLLLLLrrrrddlllldldddrrr uLdrrruLdlllluuurRlldddrruUUluRRRdLrrUUdlllldlddRdrUdrruLdllu luuurrrrrruruLLLLrrrddlllldRllddrRUUluRRRdLrrUUdlllldlddrruUluR RdlddrruLdlUUdlluuurrrdrruuruLLLrrdddlluRdrUU	6189

2	533.82	drrrrdrrddllUUruLLLulldRddrRlluuRRRdrrddllUllluurrrurDllllddrrrUdddlUrrruuLuLLLrrDDuullulldRddrRdRUUruulDlLulDrrrDDlddrUUUruLLLrrdrrddLruulldlddrUUUruLL	4520
3	64391.07	uulDDDDDurUUluurDrrrrdddllddddlllluRRluuurUluRRRRldLLulDrddlddrrruuurruuruulDDulldlllDDDuuuurrrrrdrddllldddlluRdrUUddllluRRRdrUlllluuuuurrrrrdDrdL	803538
4	1240.85	dllddddllluurRllddrrUrUUrruullDDDLddrUrrrruuuuuulllllldddRDrrruullDldRulluuurrrrrrdddddlllllUddlluuRlddrrurrrrruuuuuulllllldddrrdDLdRuuulDururrddLruulldldDrUdddllluuRRllddrrUrUddllluurRuuluuurrrrrrdddddllL	11027
5	24836.19	rurrdrrruuuLrdddlluuUddllldlluRRRRdrrruuulldDldRRdrUUdlluuuruulDDDDuuullLdlDururrrdddllldlluRRRlluururrrdddLdRuuuullldlddrRlluururrrddddRluuuullldlddrrRdrUUUddllllluururrRlllulDDDuurrrrdddllldlluRRRRdrUUddrdrruLLLullllluuurrrrDrrDDuullullldlddrrrrUdlllluuurrrrurDllldlddrrrruURuLLLrrdddllllluuuurDrrrddddrdrruLuuruLLullllldddrrrruUdddRdrUUUruLLuLLLLulDDDuurrrrrdrdddlluuUddllldlluRuuurrrRdddllLdlUUU	211913
6	2144510.83	ullluuuLUllDDuuruurDldldlldddrRRRRRRRRRRRurDllllllllluuulLuurDuullDrddLdDDllulldRRRRRRRRRRRRRRRlllllllluuuLuullDDDDuuurrdLulDDurrrdddllLLLuuuRlddlldRRRRRRRRRRRRRRdrUluRdllllllllllllluuurRdDuuurrdLulDDulldddrRRRRRRRRRRurDllllllddllllUdrrrruuluuulllllldddrRRRRRRRRRRRlllllluuullldDuullldddrRRRRRRRRRRdrUluR	8701839

References:

[1] - https://baldur.iti.kit.edu/theses/SokobanPortfolio.pdf

[2] - Albert L Zobrist. A new hashing method with application for game playing. ICCA journal, 13(2):69–73, 1970.