CS 154 PROJECT REPORT - **SOKOBAN**

Team : PRATYUSH AGARWAL 180050078
MEENAL PARAKH 180050058
PRANSHU S NEGI 180050076

Aim

To implement a Sokoban gaming environment with one-player, 2-player, computer vs player and propose a computer solver which will solve it using different searching algorithms (namely Breadth First Search, Depth First Search, Uniform Cost Search and Greedy Search).

About the game

The **goal** of the game is to move all the boxes to the goals no matter which goals. However, there are some constraints to the movement of boxes -

1) The player can only push the boxes.

2) The player is allowed to move only one box at a time; it cannot move two or more boxes simultaneously.

3) Neither a box or the player can cross/ go across a wall.

In order to reach the goal, following things have to be kept in mind -

a) If the player wants to move the box down, there must be a path reaching the upper side of the box and there should be a space on the lower side of the box to slide it there and same for other directions.

b) None of the boxes at any given instant can be in a **deadlock** state.

Deadlock

Deadlock means the level isn't solvable anymore, no matter what the user does. This happens when a box can't reach any of the goals no matter how many moves we make (stuck in corner for example), even if one box reaches a deadlock state, the game ends. Few examples of a deadlock state are shown in figure.

Modes of game

There are 5 modes.

1. 1 Player - In this the person set the grid practice on that grid to understand the game.
2. 2 Player - In this Player 1 sets the grid for Player 2 and vice versa. The one to complete the game first wins.

3. Computer vs player - Player gives a grid, which first the player solves and then the computer. The one to solve in less moves win.

In any of the above mode if a person wants to abandon solving the game (he is in deadlock or can't find solution, he/she can press the spacebar key to exit).

4. Computer solver (7*7)- Given a 7*7 grid ,computer solves it.
5. Computer solver(9*9)- Solves 9*9 grid. (Takes more time so more efficient for less number of boxes).

Description of our approach

Designing the **State .** We used a 2d vector to store the given maze.

We stores at each position of the grid , whether wall , box , open goal or player is there.

Further

For One-Player - The state is presented by a list containing the position of player, list of position of boxes, and the scene (which contains positions of goals and walls), respectively at the first, second and third place.

For Two-Player - A list containing five elements, position of player and list of boxes, for each player, and the scene, represents the state.

For Computer solver - Uses the same machinery as one-player but is simulated to show the result obtained from algorithms implemented. The algorithms provide solution in the form of sequence of 'u','d','l','r' (they representing the four directions), the cost will be the length of the solution. Further we show the solution graphically .

Deadlock Detection

1) We have explicitly coded for the base cases of deadlock, and then used them to detect other cases of deadlock as early as possible. The key principle used is - if all possible moves from a position reaches a deadlock position then the position is itself a deadlock position.

2) For smaller mazes, simple deadlock detector (simpleDD) can also be used. The key feature in simpleDD is that Simple deadlocks squares are precalculated at the time the level is loaded for playing.The key idea of algorithm is- a) Delete all boxes from maze and put them at goal positions. b) PULL the box from the goal square to every possible square and mark all reached squares as visited.

When deadlock is detected, the region, where it has occured, gets highlighted in blue.

Algorithms used in Computer Solver-

A. Breadth first Search

B. Depth first Search

C. Uniform Cost Search

Our code used all the three strategies , the result of which on different inputs was as follows-

BFS - It proved to be the best. Most time efficient and also gave the shortest solution. Our computer solver now uses BFS primarily with heuristics like deadlock detection to speed it up.

DFS - It gave long solution as it kept on trying one possible path until it encounters a deadlock. Also it took more time.

UCS - Similar to BFS but took longer time as priority queue was used so took time in insertion and deletion.

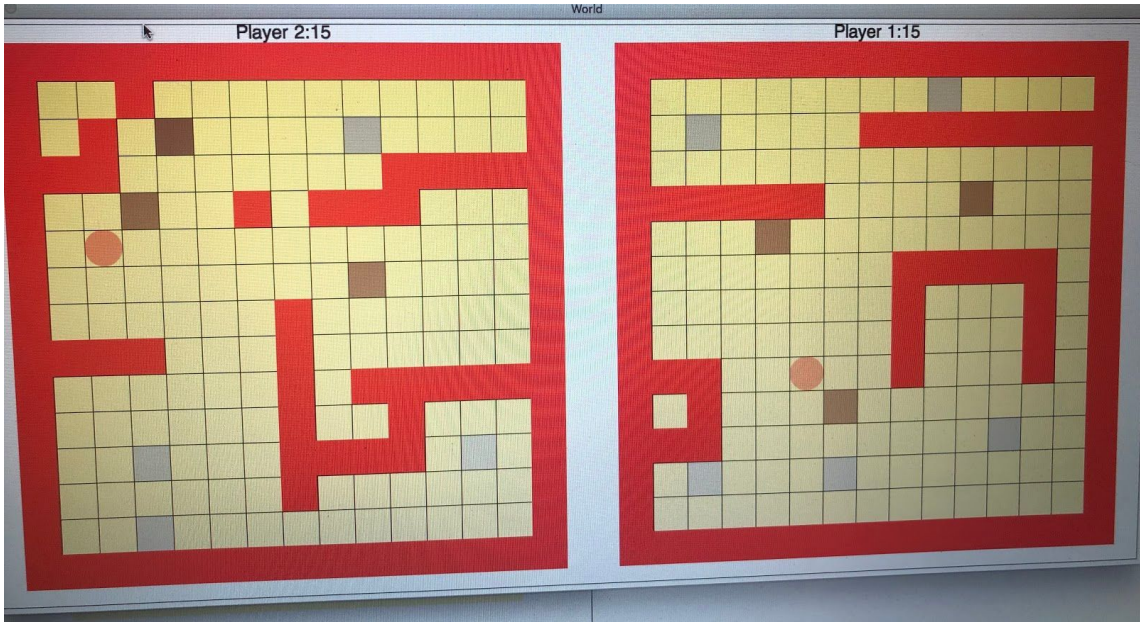
Sample input –

A map/grid surrounded by walls, having boxes and destinations placed in surrounded area. An example of how the map will look like is shown below.

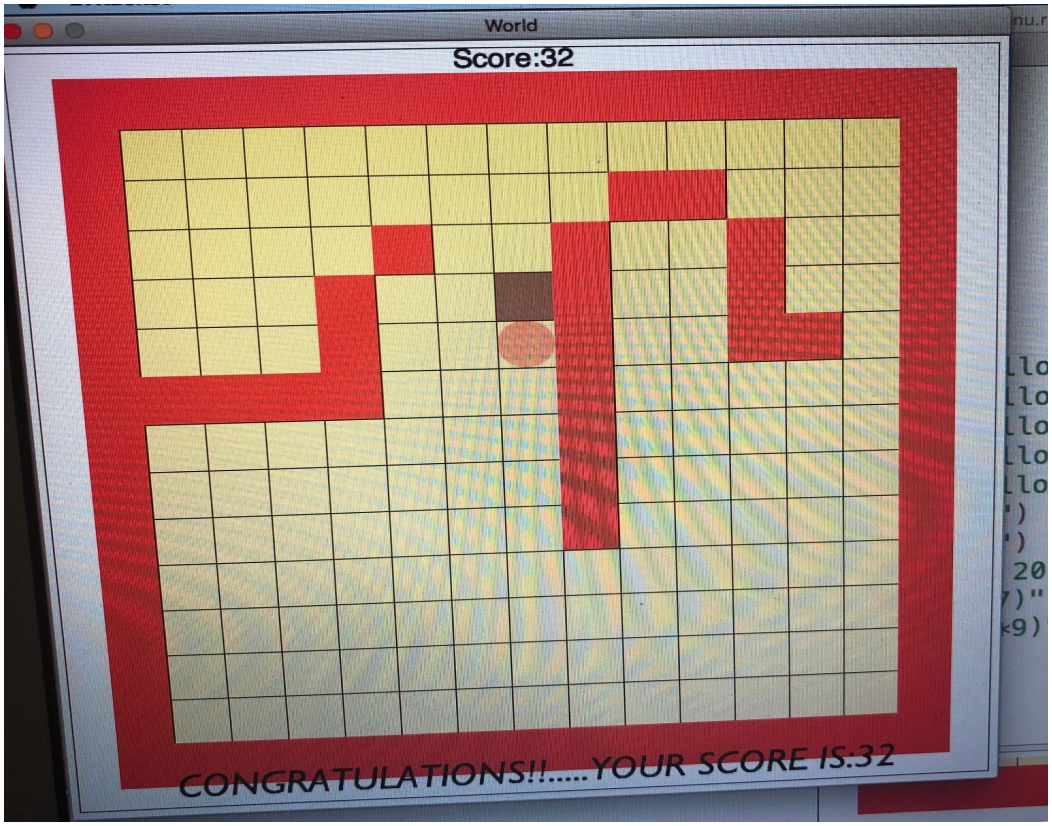
Red boxes - walls Brown Boxes : Boxes Red circle : Player

Grey / Light Boxes : Goals where boxes can be placed

(A snapshot of an intermediate 2 player game)



Output: For different modes of the game different outputs. (like the solution in computer solver, the winner in 2 player and computer vs player game, score in one player game)



(Output for 1 Player game is shown above)

For Human player modes (1 player and 2 player): The program will allow the player to take valid moves, will deny the invalid moves, keep updating the score with each move made, throw messages in a deadlock situation, and giving a “congratulation” message or who won message when all boxes get into goals.

For Computer Solver: The program will provide a solution to the given gaming environment by showing it graphically and also the score achieved by the computer.

Rules

Run menu.rkt . Select the mode which you want . A new window will popup . Follow the below instructions to give input.

For all modes.

1.First click on a box in the grid to place the player (circle)

2.Then **without** pressing space, select squares(using left mouse click) where boxes are to be placed one by one.Press space.

3. Now select the goals (where boxes should go) (should be >= no of boxes).Press space

4 Now add extra walls by selecting squares. Press space

For 1- player start playing by moving arrow keys (up,down,left,right). Game will finish when you solve it. If you want to exit , press space.
For 2 -player repeat steps 1-4 (first P1 selects left grid of P2 then vice versa). Now start alternate turns by arrow keys . P1 goes first for fairplay game (As P2 knows his grid so can select harder grid for P1 , hence P1 gets first chance).
For Computer vs player , do steps 1-4, then player plays. After competing or pressing space (to abandon) , computer plays.
For computer solver , give the grid as in 1-4 then wait for computer to solve and simulate the solution.

Summary

We used various things discussed in class

1. Use of macros (for , while)
2. Use of higher order functions and abstraction - Only one single function was created for BFS,DFS and UCS. We passed the functions like top,pop and push accordingly.
   Top- In BFS it was the first element , for DFS it was the last element and for UCS it was the min priority element. Pop and push also similarly defined for each.
3. Object oriented design - As we made 5 modes, so each mode has its separate variables and functions which were used depending on user's selection.

Various New things that were not discussed in class were also used

1. Racket Graphics : We used big bang to simulate all the grids and movements graphically.
2. Inbuilt racket function : Many racket functions like group by , sort etc were used after taking help from racket documentation
3. Heaps : We learnt about heaps and its implementation in Racket as we needed to implemented a priority queue
4. Graph Algorithms : We learnt BFS,DFS and UCS (form of djikstra algo) as these were crucial for our computer solver.

Limitations

Though the computer solver provides an efficient solution to the game and its performance against human players is significantly good, yet for larger grids , it takes a lot of time. As lots of states are generated to be searched for. After using hash tables and other time optimizations ,

the solver gives quick result in case of 7*7 grid . It works fine for 9*9 grids when the number of boxes are less (upto 2 or 3) but for larger number , it takes a lot of time.

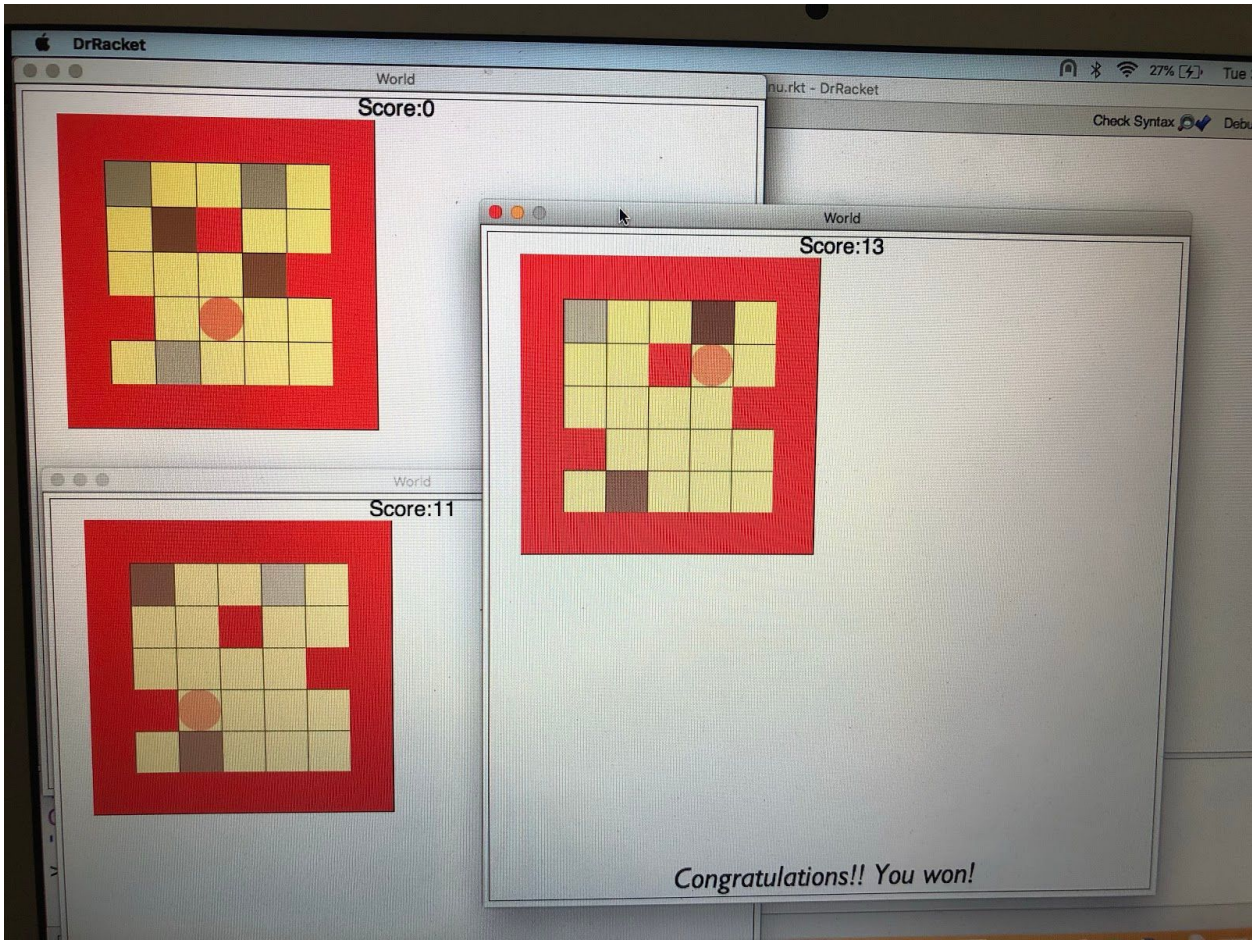
For 1 player and 2 player game , we have kept a large grid , so when players give each other grid , we can't say for surely whether a solution will exist or not as computer solver is slow for large grids (15*15). So that has to be kept in mind by the players.

Some deadlocks are detected only when they reach the base cases and so even if the game is in deadlock position, player has to keep playing to reach the base case of deadlock. Also, there may arise cases in which deadlock cannot be detected at all.
Also, simpleDD, although evaluated only one time at the beginning, can take a lot of time for mazes scarcely filled with boxes and goals and walls.

Interesting fact

In many cases the solver beats the player as it finds an almost shortest path. But still our algo is not bruteforce and doesn't give the exact smallest solution each time(To save up on computation time , as examining all states increases time and probability heuristics suggest high chance of win if we cut off some states).
One particular case when our solver lost to one of our friend is attached below.

Solver got a score of 13 while our friend solved it in 11 moves.
The initial grid , player's score and solver score is shown.


DrRacket
World
Score:0
World
Score:11
World
Score:13
Congratulations!! You won!
Check Syntax