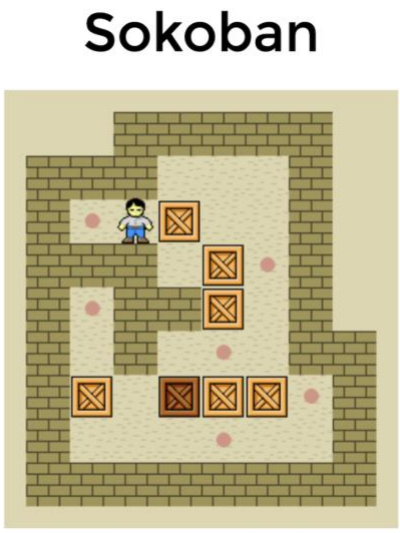
# Sokoban Project

## Group BOX, BOX!

**Prof. Harish Chandra Karnick**





| Aarush Narang | Moksha Chawla | Prerna Chakraborty | Sarthak Bhatnagar | Shashwat Prakash |
|---|---|---|---|---|
| (2010110009) | (2010110402) | (2010110485) | (2010110896) | (2010111108) |

# Contents

# The Game Description

## *1.1 Overview*

Sokoban is a seemingly simple board game where a player has to push all stones (also called crates/boxes) available on the board to final/goal positions on a board that is a maze.

## *1.2 Rules*

Each square in the grid, which is used to play the game, represents a wall or a floor. Some floor squares have boxes on them and some are designated as storage areas.

The player is limited to the grid and can only advance onto empty squares in either a horizontal or vertical direction (never through walls or boxes). The player can also enter a box to be pushed into the adjacent area. Boxes cannot be pulled or pushed against walls or other boxes. When all of the boxes are on the storage spots, the puzzle is finished.

The game rules are:

1. The player can move one square at a time in the four directions N, E, W, S.
2. The player can only push (no pulling) one stone one square at a time in the four directions N, E, W, S (no diagonal push).
3. No stone or the player can go through a wall or other stones in a move.
4. To push a stone in the relevant direction the player has to be in the appropriate immediately neighbouring square.
5. The game ends when all stones are in goal positions. A stone can occupy any goal position.

## *1.3 Scientific research*

Using the theory of computational complexity, Sokoban can be investigated. It has been established that solving Sokoban puzzles is an NP-hard task. It is PSPACE-complete and more research revealed that it was substantially harder than NP problems. Researchers studying artificial intelligence may find this interesting as well because creating a robot to move boxes in a warehouse is similar to solving the Sokoban puzzle.

Sokoban's difficulty stems from both its large search tree depth and branching factor, which is comparable to that of chess. Some levels need more than 1000 "pushes." Human players who are skilled largely rely on heuristics because they can quickly dismiss pointless or redundant lines of play and identify patterns and subgoals, which significantly reduces the amount of search.
a few Sokoban games

## 1.4 Challenges in the game

- **The size of the state domain**: The size of the domain increases exponentially since the player and the boxes could be on any legitimate floor of the grid.
- **Number of steps in solution**: A level's ideal solution could involve tens or even hundreds of steps.
- **Deadlock situations**: There may be a box or boxes that cannot be brought to their intended destination by any plausible sequence of events. These scenarios can be challenging to identify, which results in numerous unneeded operations to try to understand the situation.

The objective of the game is to push all the stones to their final positions. A solver may choose to optimize

i) the total number of moves or

ii) the number of push moves.

A push-move is one where a stone is moved.

## 1.5 Representing the game as a search problem

Formally a search problems defined as <**S**tates, **I**nitial state, **G**oal states, **A**ctions, **T**ransition Function> in our case:

- **State**: represented as the places of boxes and the player over grid, each of them can be in legal floor square.
- **Initial state**: a state where the boxes and the player are in their initial positions.
- **Goal state**: a state where all boxes are at storage locations.
- **Actions**: all possible actions that can be applied over state, in this game to move (top, down, left, right) or subset of these actions (not moving into wall).
- **Solution**: series of actions that leads the initial state to the goal state.
- **Solvable initial state**: there is a series of actions for each box to move it to storage location and there are enough storage locations for all boxes.
- **Transition Function**: the cost of each action is 1, the player makes one step each time in any direction.

' ' (white space) - Floor
@ - Player
+ - Player on goal
# - Wall
$ - Stone/Crate/Box
. - Goal
* - Stone/Box/Crate on Goal

The semicolon ';' is used as a comment character before the puzzle itself starts.
A move by the player is represented by u, d, l, r standing for up, down, left, right. If a move results in a push of a stone the relevant letter is capitalized i.e. U, D, L, R. So, a solution to an instance is just a string of letters from the set {u, d, l, r, U, D, L, R}.

# 2. How We Approached The Problem

## *2.1 Initial Approach*

The initial approach was to try some sort of brute force. So we tried methods like bfs and dfs to solve the puzzle states. We used simple deadlock states to check whether the box could move or not by expanding its moves. But as we know about brute force the solutions were not very efficient. Not only this, it both failed to solve the problem and took a long time to solve the ones it could.

## *2.2 Further Approach*

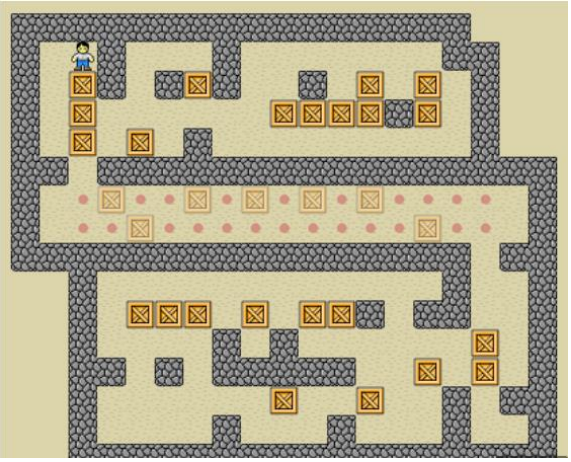
We had to make a better search algorithm to solve our puzzle. Hence we started to use A star search algorithm. It is a searching algorithm that is used to find the shortest path between an initial and a final point by calculating costs for each state by using heuristics and solving them accordingly. This was able to solve some of the simpler problems much faster, yet while printing the states we realized that our deadlock detection wasn't accomplishing much. Therefore, we implemented a better deadlock detection method by checking if a box is trapped against the wall according to rows or columns and checking whether the boxes are cornered or not, since that would be a dead end state unless a target is placed there (we can't move a box that is cornered). With this we were able to solve much more complex puzzles and this also reduced the number of states explored.

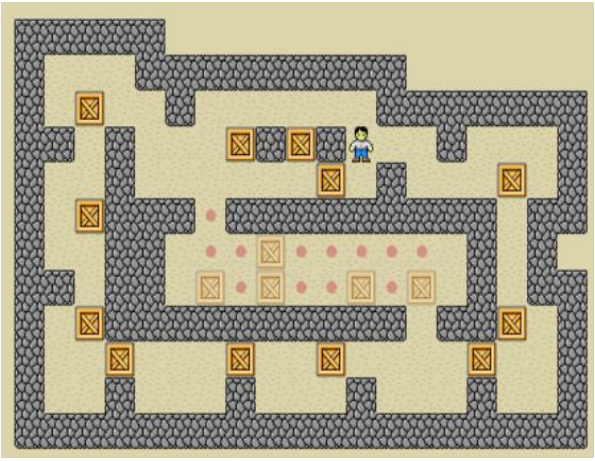
## *2.3 Deadlock detection*

All state transitions in the majority of single-agent search problems investigated maintain the problem's solvability (but not necessarily the optimality of the solution). This results from all state transitions (moves) being cancellable (there exists a move sequence which can undo a move). Sokoban has actions that cannot be reversed, like moving a stone into a corner, and these actions lead to situations where solutions are demonstrably impossible. A single action in effect can set infinity as the bottom bound for the solution's length. If the lower bound function does not reflect this, then the search will spend unnecessary effort exploring a sub-tree that has no solution. We call these states *deadlocks* because one or more stones will never be able to reach a goal.

The other approach that we had used was to find states in which columns and rows would become a trap for the player. We had used this approach to make use of these states and use them in reducing our sample size for the problem states.

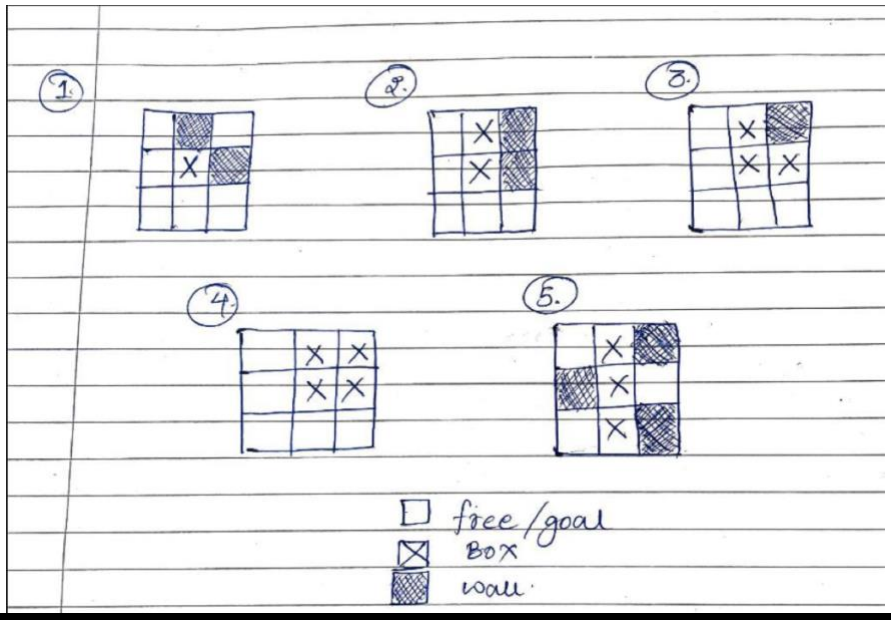
Shown below are some images depicting the same.





We have used two approaches to implement deadlock detection to make our program run faster. The first approach used was to record 5 deadlock states that could occur during the game. To alleviate this, we implemented the mirrored positions of the deadlock states that we had initially chosen as we realized that the absolute positions did not matter as much as their relative positions.

Shown below is an image depicting the 5 deadlock states that we used.



# 3. Heuristics Used

## *3.1 A* Heuristic:*

In A*, we employ a priority queue and always select the state that is "closest" to a solution rather than traversing all possible states using BFS or DFS. G+h provides "closest," and how well it works depends much on the value you pick for h. For instance, euclidean or manhattan distance can be quite helpful for issues involving moving an object along a matrix.

The Algorithm A* is BFS search algorithm that uses f to choose the next high priority state to check from the unvisited nodes. Here it's pseudo-code:

```
Astar(root) → solution

Astar(node):
    visited = set()
    toCheck = PriorityQueue()
    toCheck.push(node, 0)
    while toCheck not empty:
        currentNode = toChech.pop()
        if currentNode in visited:
            continue
        if currentNode is goal state:
            return solution
        visited.add(currentNode)
        for each node n in currentNode.expand():
            toCheck.push(n, f(n))
```

Because there are multiple methods to get to the same state in this game, we build the algorithm as a tree search that does not explore previously visited states in order to prevent loops.

The quantity of RAM that A* needs is a drawback. If we have limited memory, the search may become impossible since it must keep all the states it intends to investigate in a priority queue during the search phase.

In general, heuristic functions utilising Manhattan distance are preferable than those using Euclidean distance since the action/move is based on Manhattan distance. The best method with a consistent heuristic function is A*.

### 3.1.1 Manhattan distance

The Manhattan distance is the common heuristic for a square grid. Find the lowest cost D for going from one area to an adjacent space by looking at your cost function. To make things simple, set D to 1. On a square grid with four directions of movement, the heuristic should be D times the Manhattan distance:

```
def calc_manhattan(self, p1, p2):
    return abs(p1[0] - p2[0]) + abs(p1[1] - p2[1])
```

### 3.1.2. Euclidean distance

It uses the euclidean distance function to calculate the distance of the straight line between two points.

```
def calc_euclid(self, p1, p2):
    return math.sqrt((p1[0] - p2[0])*(p1[0] - p2[0]) + (p1[1] - p2[1])*(p1[1] - p2[1]))
```

### 3.1.3. Inadmissible Heuristic

A better and more complicated heuristic which need not always be admissible but improves the search on some of the more complicated sokoban levels most of the time. It uses dynamic programming for caching and problem initialization. Uses flood fill as well.

The flood-fill is executed breath-first. This makes sure that the path that is found to a reachable tile is the shortest possible path. This path is used to apply the alteration.

Once all reachable tiles have been found, checks are performed from each reachable tile into all four directions (up, down, left and right). These checks test for the presence of a box in a neighboring tile and for an empty floor tile behind it. If this is the case, a push alteration is created. The push alteration contains the sequences of moves to the reachable tile and includes a move into the direction of the box, to push the box.

Initially: For our algorithm, we assigned a cost value to each floor position in the level. The shortest distance between this point and the closest target is represented by this cost value.

We perform a breadth-first search from each target using the flood fill algorithm, and we compute the cost (steps from target to position).Each vacant space will have the lowest possible cost to the nearest target only if this cost is less than the previous cost.Only

exploring the places that are floors, the flood fill algorithm carefully considers wall blockage when determining the best path.

Additionally, we memorize each cost (a linear combination of floor cost, box moves, and targets left) so that we can reuse it later if the box positions ever change and are in the same locations.

We return the state's cost with each call to a heuristic function. The cost is calculated by linearly combining:

Floor price: the shortest distance between a box and the closest target

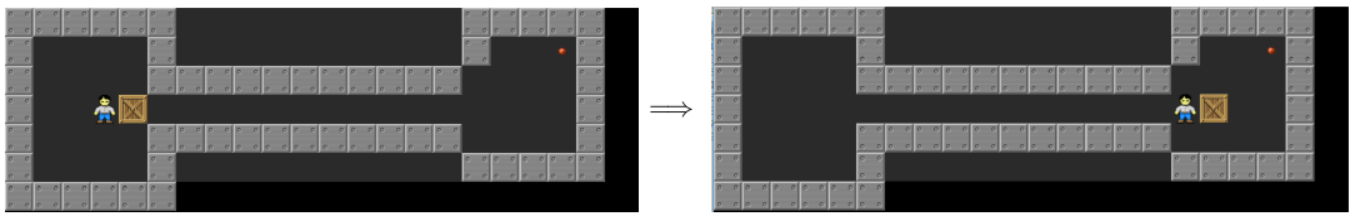
Boxes moved: boxes moved in a specific state

Targets remaining: how many targets are open and unprotected by a box

# 4. Future Improvements

## *4.1. Tunnel Macros*

A macro is a rule or pattern that specifies how a certain input sequence should be mapped to an output sequence. Macro moves, in Sokoban, consist in treating a sequence of moves as a single move. Using a marco effectively removes all the moves encompassing the macro from the search tree. Use of macro moves decreases the depth and branching factor of the search tree.



## *4.2. Looking 2 moves deep instead of one row deep*

Instead of exploring all the vertices of a particular depth before moving on to a vertex with a higher depth, it will always follow the successors of a vertex to the next depth. When the search cannot progress further down it has to backtrack to the deepest vertex that still has unexplored successors.

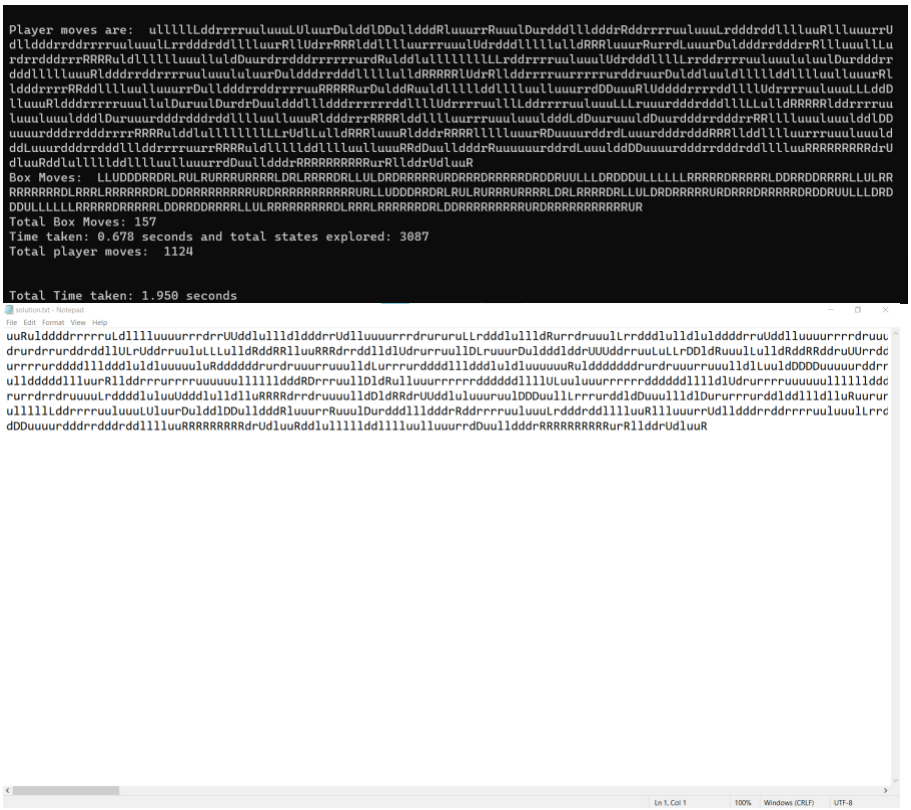
### *4.3. Looking at area instead of specific path*

As the ending of any series of player moves is with a push move, the entire accessible area by a player in a state can be considered as the player having any position in it. Hence, removing the player from the board and only knowing what the accessible area was would reduce the number of different states possible exponentially.

To implement this, we tried to reduce the map to a 2D matrix with 1s representing the accessible area by the player in that state which was found using a flood fill algorithm. However, the comparison of states that were helpful became a problem and the solver couldn't figure out which position of the player would be optimal. The code for this is under the function get_area() which has been commented out.

## 5. Conclusion

We have successfully implemented Sokoban Solver using A star search algorithm. We were able to run all test cases from the text file testExamples.xsb and output both the play moves and push moves in our solution.txt file. It works in under 3 seconds as well. By the help of better deadlock implementation, and maybe better heuristics we can solve the puzzles more efficiently.




Solution.txt

## *6. Test Cases*

```
PS C:\Users\shash\Desktop\College\Project\AI_Sokoban\Final try> python .\main.py
Starting level  1
#########
#...  . #
######  #
#     $#
# $.   #
# # ####
#@$$ $ #
#      #
#######

Player moves are:  uuRuldddrrrrruLdllluuuurrrdrrUUddlulldldddrrUdlluuuurrrdrururuLLrdddlullldRurrdruuulLrrdddlulldl
uldddrruUddlluuuurrrdruuullLLrrrdddlulldldddrruuUdddrruLdlUUddlluuuurRRRdrUUruLLLLrrrdddlulldldddrruuUdddlluRdrUU
ddluluuurRRRdrUUruLLLrrdddlulldldddruruUdddluluuurRRRdLurdrUU
Box Moves:  RLUUULLRLULLULUURRRUULLLLURUURRRUULLLURRRLUURLUUULLRLULLULUURRRUULLLLURUURRRUULLLURRRLUU
Total Box Moves: 44
Time taken: 0.057 seconds and total states explored: 396
Total player moves:  279
```

```
Starting level  2
########
#@  #  #
# ..   ##
##.##   #
 #  $$# #
 ###$   #
   #  ###
   ####

Player moves are:  drurdrrurddrddllULrUddrruuluLLLulldRddRRlluuRRRdrrddlldlUdrurruullDLruuurDuldddlddrUUUddrruuLuLLrDD
ldRuuulLulldRddRRddruUUrrddLLuuruulDlLulDurdrrurddrddlluUddrruuluLLLulldRRRRddlllUdrrrurrddlldlUdruuuullulldRddrRddrur
ruullulLrrurDuldllulDurdrrdDlddrUUUddrruuLuLLLrrurddrddlluUddrruuluLL
Box Moves:  ULULLLRRRRRRUDLDUUULLLDDRLRRRUULLDLDULLLRRRRUURRLDDDUUULLLLULLULULLLRRRRRRUDLDUUULLLDDRLRRRUULLDLDULLLRRRR
UURRLDDDUUULLLLULL
Total Box Moves: 62
Time taken: 0.072 seconds and total states explored: 569
Total player moves:  286
```

```
Starting level  3
   ###
  ## # ####
 ##  ###  #
## $      #
#   @$ #  #
### $###  #
  #  #..  #
 ## ##.# ##
 #      ##
 #     ##
 #######

Player moves are:  urrrrurdddd llldddluldluuuuuluRddddddrurdruuurruulldLurrrurddddlllddduldluuuuuuRuldddddddrurdruuur
ruuulldlLuuldDDDDuuuuurddrrurrurddddlllddduldlluRRdluuuuuuurdRdluuldddrUdluuRurRRlllulddddddrrruuurruuruulDurddddll
ldddlluluuuuuuurdrrdLululddddruUddlddrrruuurruruuuldDuurddddlllddduluuuuuRRRRldlululddddddddrrruuurrruulDuruulDDuurdd
ddLLrruuuuldduuurddddLuuulldlululddddddddruRdrU
Box Moves:  RLRLDDDDRRRURRRDLUDRRRRDDDLLDLRURLRLDDDDRRRURRRDLUDRRRRDDDLLDLRU
Total Box Moves: 32
Time taken: 0.431 seconds and total states explored: 3262
Total player moves:  382
```

```
Starting level  4
 #########
 #       #
 # ##### #
 # #   # #
 # $.#@# #
###$.  # #
#  $.### #
# #$.    #
#    #####
######

Player moves are:  ullddddddlllluurRlldddrrrurrrrruuuuuulllllllddRDrrruullDldRullluuurrrrrrddddddllllULuuluuurrrrrrddddddll
llldlUdrurrrruuuuuulllllllddrrrDullluuurrrrrrddddddlllldluUddrurrrruuuuuulllllldddrrdDldRuuuurrddLruulldlDDrUdddllluuRRll
ddrrruUddlllluurruulluuurrrrrrddddddllLrrruuuuuulllllldddrddllddrrUdlluurR
Box Moves:  RRDDRULUDUDRLDDURRULURRRDDRULUDUDRLDDURRULUR
Total Box Moves: 22
Time taken: 0.158 seconds and total states explored: 971
Total player moves:  289
```

```
Starting level  5
 ########
 #  ##  #
 #.$   .###
 # .## $  #
##$### #  #
# $    # ##
# @ #.    #
#######   #
      #####

Player moves are:  rurrdrrdruuuuLrddddluluuUdddlulldlluRRRRdrrdruuuulldDldRRdrUUddluluuuruulDDDuullLrrrurddldDuuullldl
DururrrurrddldddllldlluruurrrurrddlddldRuuuulllldlddRRldluuururrrurddrDuluuldllldldddrurRdrUUUdddlulldluuururrRddddlulld
luuuruulDDDuuurdrrrddddlulldlluRuuuurdrrrddddRluuuulllulddddRRRdrrdrruLLLUlllldluuuuurdrrrdrrdDuulluurDulddRuuldlllluldd
dddrurrrUUruLLLrrurddlddlllldluuuuurDuldddddrurrrdrdrruLuuruLLuuldlllulddddrurrrdRdrUUUdddluluuUdddrdruuuruLLuLLLLulDD
DuuurdrrrurddrddddluluuddlllldlluRuuuurdrrRdddllLdlUUU
Box Moves:  LURRRRDRRUUDDDLDDRRRRDRUUURDDDRRRRRLLLUDDRUULLLDLLLRUUUULLLLLLDDDURRLUUULURRRRDRRUUDDDLDDRRRRDRUUURDDDRRRR
RLLLUDDRUULLLDLLLRUUUULLLLLLDDDURRLUUU
Total Box Moves: 72
Time taken: 0.607 seconds and total states explored: 4733
Total player moves:  507
```

```
 Starting level  6
    #####
    #   #
    #$  #
  ###  $##
  #  $ $ #
### # ## #   ######
#   # ## #####  ..#
# $  $         ..#
##### ### #@##  ..#
    #     #########
    #######

 Player moves are:  ullllLddrrrruuluuuLULuurDulddlDDullddddRluuurrRuuulDurdddlllddddrRddrrrruuluuuLrddddrddllllluuRlllluuur
 rUdlldddrrddrrrruuluuulLrrdddrddllllluurRllUdrrRRRldddllllluurrruuulUdrddddllllulldRRRluuurRurrdLuuurDulddrrdddrrRllluuu
 llLurdrrddrrrRRRRuldlllllluuulluldDuurdrrddddrrrrrrurdRuldddlulllllllLLrddrrrruuluuulUdrddddllllLrrddrrrruuluuululuulDu
 rddddrrddddllllluuuRlddddrrddrrrruuluuuululuurDuldddrrddddllllluulldRRRRRlUdrRlldddrrrruurrrrrurddruurDuldddluuldlllllddllllluu
 lluuurRllddddrrrrRRddllllluullluuurrDullddddrrddrrrruuRRRRRurDuldduuuldlllllddllllluullluuurrdDDuuuRlUddddrrrrddllllUdrrrruu
 luuuLLLddDlluuuRlddddrrrrruuullulDuruulDurdrDuuldddllldddrrrrrrddllllUdrrrruullLddrrrruuluuuLLLruuurddddrddlllLLlulldRR
 RRRlddrrrruuluuuluuuldddlDuruuurddddrdddrddllllluullluuuRlddddrrrRRRRlddllllluurrruuuluuuldddLdDuuuruuuldDuurddddrrddddrrRRlll
 luuuluuuuldddlDDuuuurddddrrddddrrrrRRRRuldddlullllllllLLrUdlLulldRRRluuuRlddddrRRRRllllluuurRDuuuurddrdLuuurddddrdddRRRlldddll
 llluurrruuuluuuuldddLuuurddddrrddddllldddrrrruurrRRRRuldlllllddllllluulluuuRRdDuulldddrRuuuuuurddrdLuuuldd DDuuuurddddrrddddrdd
 llllluuRRRRRRRRRdrUdluuRddlullllldddllllluullluurrdDuulldddrRRRRRRRRRRurRlldrUdluuR
 Box Moves:  LLUDDDRRDRLRULRURRRURRRRLDRLRRRRDRLLULDRDRRRRRURDRRRDRRRRRDRDDRUULLLDRDDDULLLLLLRRRRRDRRRRLDDRRDDRRRRLLUL
 RRRRRRRRRDLRRRLRRRRRRDRLDDRRRRRRRRRRURDRRRRRRRRRRRURLLUDDDRRDRLRULRURRRURRRRLDRLRRRRDRLLULDRDRRRRRURDRRRDRRRRRDRDDRUULL
 LDRDDDULLLLLLRRRRRDRRRRRLDDRRDDRRRRLLULRRRRRRRRRDLRRRLRRRRRRDRLDDRRRRRRRRRRURDRRRRRRRRRRRUR
 Total Box Moves: 157
 Time taken: 0.643 seconds and total states explored: 3087
 Total player moves:  1124
```

## 6. References


1. https://timallanwheeler.com/blog/2022/01/19/basic-search-algorithms-on-sokoban/
2. https://baldur.iti.kit.edu/theses/SokobanPortfolio.pdf
3. https://assets.ctfassets.net/r26fkm24j6bh/4cK9QOfIHKt3SBe2q3dXG0/3d6bf0dce06964 1deb4d88d2ce62e2df/masterthesis_simonkarman.pdf
4. https://www.geeksforgeeks.org/python-map-function-count-total-set-bits-numbers-1-n/
5. https://baldur.iti.kit.edu/theses/SokobanPortfolio.pdf
6. https://worksheets.codalab.org/worksheets/0x2412ae8944eb449db74ce9bc0b9463fe/
7. https://www.cs.huji.ac.il/w~ai/projects/2012/SokobanWP/files/report.pdf
8. https://www.ri.cmu.edu/pub_files/2011/3/fahrpaper.pdf
9. http://sokobano.de/wiki/index.php?title=Solver_Statistics
10. https://www.cs.cornell.edu/andru/xsokoban.html
11. http://www.game-sokoban.com/index.php?mode=catalog