```
########################################ASTAR
ALGORITHM#########################################################################
import heapq
import time
import sys

def ValidMove(board, x, y):
    """
     :param board: NxN board
     :param x: x position
     :param y: y position
     :return: return 2d board
     """
    return 0 <= x < len(board) and 0 <= y < len(board[0])


def ManhattanDistance(pos1, pos2):
    """
    :param a: Any position
    :param b: Any position
    :return: Absolute distance between 2 co-ordinates
    """
    return abs(pos1[0] - pos2[0]) + abs(pos1[1] - pos2[1])


def Heuristic(board,robot, boxes, storagepace):
    """
      :param board: NxN Board
      :param robot: Robot position
      :param boxes: Boxes position
      :param storagepace: storagspace  positions
      :return: total distance
      """
    total_dist = 0
    for box in boxes:
        min_dist = min(ManhattanDistance(box, target) for target in storagepace)
        total_dist += min_dist
    return total_dist + ManhattanDistance(robot, boxes[0])


def ShowBoard(board):
    """
    :param board: NxN board
    :return: Print board on console
    """
    for row in board:
        print("".join(row))
    print()


def Successor(board, robot, boxes, storage):
    """
     :param board: NxN board
     :param robot: Location of robot
     :param boxes: Location of boxes
     :param storage: Location of the storage
     :return:Show next board state
     """
    print("Pukoban Using A Star Algorithm")
    directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]

    def is_goal(boxes):
        return sorted(boxes) == sorted(storage)

    def apply_move(entity, direction):
        return (entity[0] + direction[0], entity[1] + direction[1])
```

```
    def valid_move(entity, direction):
        new_entity = apply_move(entity, direction)
        if not ValidMove(board, *new_entity) or board[new_entity[0]][new_entity[1]] == 'O':
            return False
        return True

    start_state = (robot, tuple(sorted(boxes)))
    visited = set()
    priority_queue = [(0, start_state)]

    while priority_queue:
        f_cost, (robot, boxes) = heapq.heappop(priority_queue)

        if is_goal(boxes):
            return boxes

        if (robot, boxes) in visited:
            continue

        visited.add((robot, boxes))

        solution_board = [list(row) for row in board]
        for box in boxes:
            x, y = box
            solution_board[x][y] = 'B'
        solution_board[robot[0]][robot[1]] = 'R'
        ShowBoard(solution_board)

        for direction in directions:
            new_robot = apply_move(robot, direction)
            if not ValidMove(board, *new_robot) or board[new_robot[0]][new_robot[1]] == 'O':
                continue

            if (new_robot, boxes) not in visited:
                heapq.heappush(priority_queue, (f_cost + 1 + Heuristic(board, new_robot,
boxes, storage),
                                                (new_robot, boxes)))

            for box_index, box in enumerate(boxes):
                if box == new_robot:
                    new_box = apply_move(box, direction)
                    if valid_move(box, direction) and (
                    new_box, boxes[:box_index] + (new_box,) + boxes[box_index + 1:]) not in
visited:
                        heapq.heappush(priority_queue, (f_cost + 1 + Heuristic(board,
new_robot, boxes, storage),
                                                        (new_robot,
                                                         boxes[:box_index] + (new_box,) +
boxes[box_index + 1:])))
    return None


def Initstate():
    """
    :return: Initial state of the board
    """
    print('Initial Board\n')
    file_obj = open(sys.argv[1])
    rows = file_obj.readlines()
    for line in rows:
        if line != "":
            board.append(line.strip())
        else:
            break
    return board
```

```
def GetStoragePos(board):
    """
    :param board: NxN board
    :return: Positions of the Storages
    """
    return {(x, y) for x in range(len(board)) for y in range(len(board[0])) if board[x][y] ==
'S'}


def GetRobotPos(board):
    """
    :param board: NxN board
    :return: Positions of the Robot
    """
    return [(x, y) for x in range(len(board)) for y in range(len(board[0])) if board[x][y] ==
'R'][0]


def GetBoxPos(board):
    """
    :param board:NxN board
    :return: Positions of the boxes
    """
    return tuple((x, y) for x in range(len(board)) for y in range(len(board[0])) if board[x]
[y] == 'B')



# Driver Code
board = []
board = Initstate()

storage = GetStoragePos(board)
robot = GetRobotPos(board)
boxes = GetBoxPos(board)

start_time = time.time()


result = Successor(board, robot, boxes, storage)

if result:
    print("End state can be found:")
    solution_board = [list(row) for row in board]
    for box in result:
        x, y = box
        solution_board[x][y] = 'B'
    print("\n".join(["".join(row) for row in solution_board]))

else:
    print("End state cannot be found:")

print(f'Total execution time is {time.time() - start_time}')
```