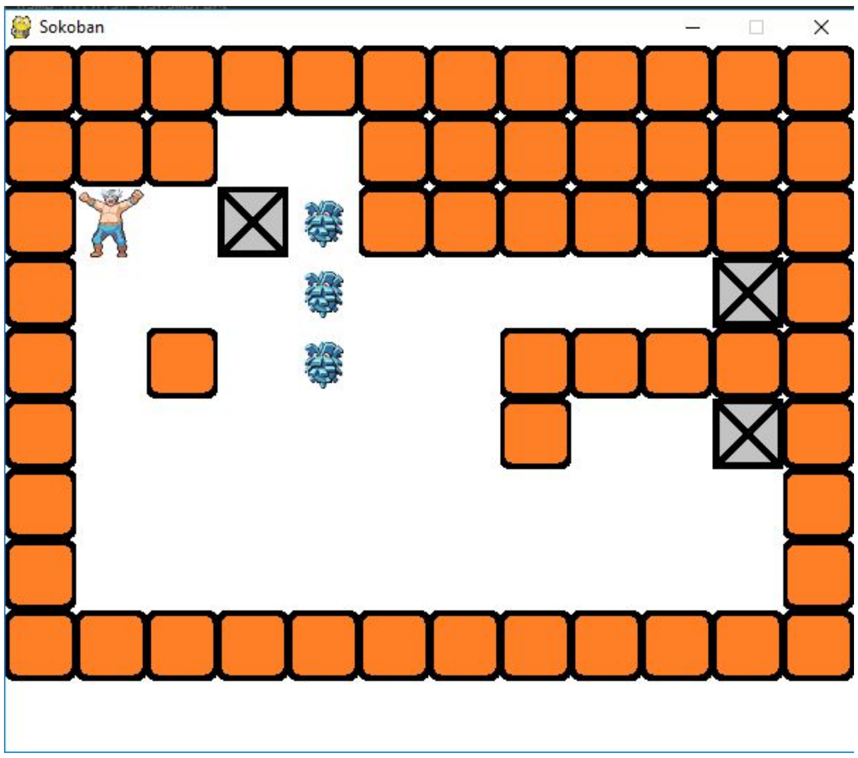
SOKOBAN SOLVING ARTIFICIAL INTELLIGENCE

TEAM MEMBERS:

● ROHAN RAJESH TALESARA	01FB16ECS309
● SAI SRIKAR KOMARAVOLU	01FB16ECS330
● SANAT S BHANDARKAR	01FB16ECS339



INTRODUCTION

Sokoban is one of the most popular thought and logic games.
The name, and the game also, comes from Japan and means "warehouse keeper".
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.
Each level has a different structure, which requires a different solution, often with a different strategy as well.
The magic of the game is that boxes, which are out of the way on one move, might be in the way on the next move.
The clearness of the rules, in combination with so many levels ranging from easy to extremely difficult, have made Sokoban become a real classic.

The rules of the game

The objective of the Sokoban game is to move objects (usually boxes) to designated locations by pushing them.
These objects are located inside a room surrounded by walls. The user controls a pusher called "Sokoban" which is said to mean something like "warehouse keeper" in Japanese.
The pusher can move up, down, left and right, but cannot pass through walls or boxes, and can only push one box at a time (never pull).
At any time, a square can only be occupied by either a wall, a box, or the pusher.

How to play Sokoban

First let's have a look at the rules of the game:

The object of Sokoban is to take some objects (usually boxes) to designated locations by pushing them.

To do this the user moves a man who we call Sokoban. Sokoban can move up, down, left and right. He can't pass through walls or boxes. He can push only one box at a time (never pull). At any time a square can only be occupied by one of a wall, box or man.

A level contains these elements:

Walls: #
Boxes: $
Goals: .
Free squares: ' '
The Sokoban / the player: @

Boxes and the Sokoban can also be located on a goal.

Boxes on goals: *
The Sokoban on a goal: +

In the following the Sokoban is called "the player" to indicate that this is the object the user can move.

Level 1

```
#########
#@  $  .#
#########
```

Using the arrow keys one has to press the "right arrow" 5 times.

Level 2

```
########
#    ###
#@$  ###
#### ###
##   ###
##    ##
#  ##. #
#      #
#####  #
########
```

Usually Sokoban levels are a lot more complicated.

Let's increase the difficulty step by step:

Level 3

```
########
#      #
#  $  .#
#@ $  .#
#  $  .#
#      #
########
```

Here the level contains 3 boxes. Nevertheless the level can easily be solved. Although every box can be pushed to each goal it doesn't matter which box is pushed to which goal. The boxes can just be pushed one after the other to any goal.

Level 4
```
########
#  #.  #
# $#   #
#  # @##
#  # $##
#    .##
########
```
This level is more complicated. The box under the player is close to a goal. Nevertheless it can't be pushed to the goal - this would result in a deadlock. This level shows some problems one has to solve before it's possible to finish the level:

1. Not every box can be pushed to every goal
2. Boxes can't just be pushed to the nearest goal in every case
3. Box pushes can create a deadlock

Level 5
```
#########
##  #   #
#.$.  $ #
# #  ## #
# @$.$. #
#########
```
To solve this level the boxes have to be pushed to their goals in a specific order. Moreover one box has to be pushed away from its goal before it can be pushed to it.

Level 6
```
#########
#  #   .#
#@$ $   #
# $ ##..#
#   #####
#########
```
This level is another example of the need to push a box away from its goal before the level can be solved. Additionally, one box has to be "parked" until some other boxes have reached their goals. A situation where a box must be pushed but can't be pushed immediately to its goal occurs very often in Sokoban levels. Some of them are "parking" situations. "Parking" means:

1. a box must be pushed away from its goal
2. after the box is pushed to its parking position another box has to be pushed
3. parking the box at a specific position is required to solve the level

Scientific research on Sokoban

Sokoban can be studied using the theory of computational complexity. The problem of solving Sokoban puzzles has been proven to be NP-Hard. Further work showed that it was significantly more difficult than NP problems; it is PSPACE-COMPLETE. This is also interesting for artificial intelligence researchers, because solving Sokoban can be compared to the automated planning that needs to be done by a robot that moves boxes in a warehouse.

Sokoban is difficult not only due to its branching factor(which is comparable to chess), but also its enormous search depth; some levels can be extended indefinitely, with each iteration requiring an exponentially growing number of moves and pushes. Skilled human players rely mostly on heuristics; they are usually able to quickly discard futile or redundant lines of play, and recognize patterns and subgoals, drastically cutting down on the amount of search.

IMPLEMENTATION METHODOLOGY

The problem here is a search problem, so different search algorithms were implemented and compared.

The various search algorithms used include:

1. **Depth First Search**
2. **Breadth First Search**
3. **A* Search**
4. **Uniform Cost Search**
5. **Iterative Deepening Search**

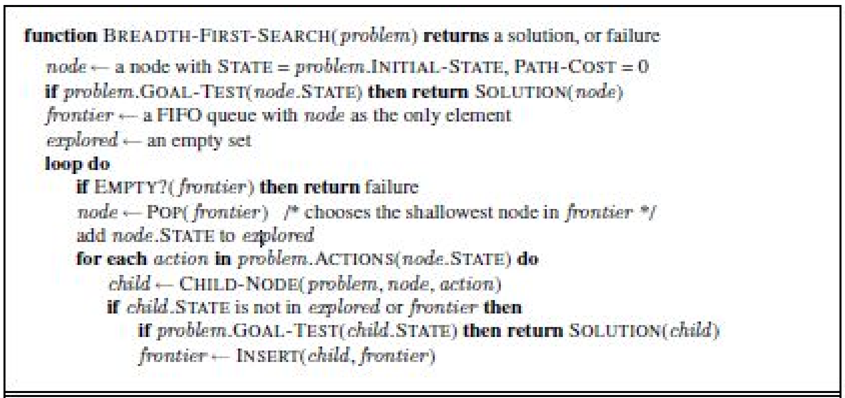
Criteria for evaluating performance of a search algorithm include:

- **Completeness**
- **Optimality**
- **Time complexity**
- **Space complexity**

Depth First Search (DFS)

Depth-first search (**DFS**) is an algorithm for traversing or searching tree or graph data structures. It is an uninformed search technique. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

Backtracking is a general algorithm for finding all (or some) solutions to some computational problems, notably constraint satisfaction problems, that incrementally builds candidates to the solutions, and abandons a candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution. The pseudo code for the DFS search algorithm is given below:



```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
      if EMPTY?(frontier) then return failure
      node ← POP(frontier)  /* chooses the shallowest node in frontier */
      add node.STATE to explored
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          if child.STATE is not in explored or frontier then
              if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
              frontier ← INSERT(child, frontier)
```

Figure 3.11 Breadth-first search on a graph.

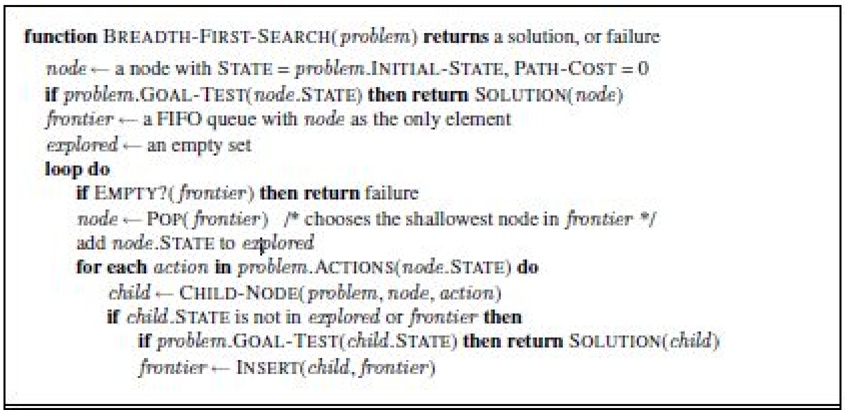
Performance of DFS:

- **Completeness :** DFS is **not complete** as it does not guarantee to find a solution if there exists one.
- **Optimality :** DFS is **not optimal** as it does not always find an optimal solution to the problem due the looping issue in graph nodes.
- **Time Complexity : O(b^m)** where b is the branching factor and m is the depth of the state space.
- **Space Complexity : O(bm)**

Breadth First Search (BFS)

Breadth-first search (**BFS**) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a 'search key'), and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level. It is an uninformed search technique which uses the opposite strategy as depth-first search, which instead explores the highest-depth nodes first before being forced to backtrack and expand shallower nodes. BFS performs better than DFS when there are potential loops between graph nodes, as it does not perform backtracking which would lead to looping within the same nodes.

The pseudo code for the BFS search algorithm is given below:



```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)  /* chooses the shallowest node in frontier */
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
           frontier ← INSERT(child, frontier)
```

Figure 3.11 Breadth-first search on a graph.

Performance of BFS :

- **Completeness :** BFS is **complete** as it guarantees to find a solution if there exists one.
- **Optimality :** BFS is **not optimal** because it fails to provide an optimal solution in some cases.
- **Time Complexity : O(b^d)** where b is the branching factor and d is depth of state space.
- **Space Complexity : O(b^{d+1})**

<u>A* Search</u>

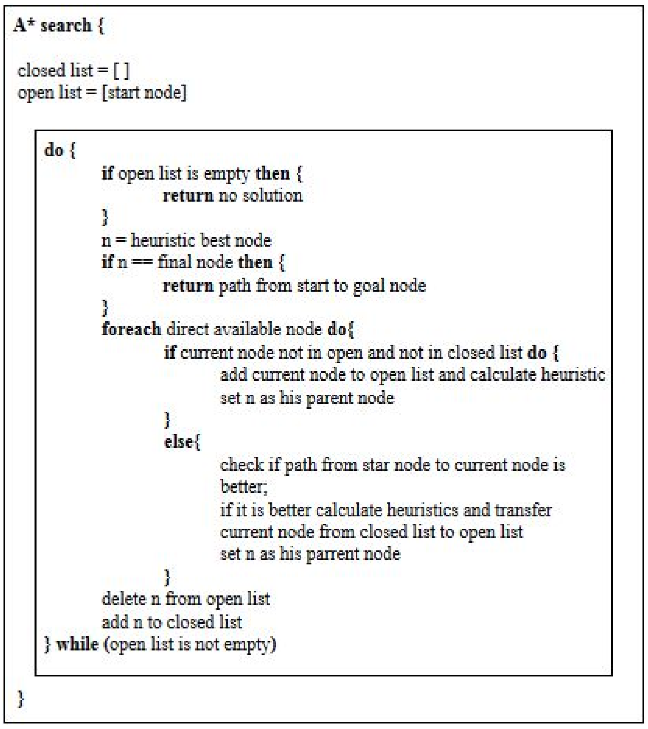
A* is a computer algorithm that is widely used in pathfinding and graph traversal, which is the process of finding a path between multiple points, called "nodes". It is an informed search technique and enjoys widespread use due to its performance and accuracy. However, in practical travel-routing systems, it is generally outperformed by algorithms which can pre-process the graph to attain better performance, although other work has found A Star to be superior to other approaches.

This algorithm was designed as an extension of the famous Dijkstra's algorithm by including a heuristic measure as a parameter to the function of a node in the graph. This heuristic can be changed based on the requirement of the problem. The heuristics we have considered in our problem include:

- Euclidean distance
- Manhattan distance
- Hungarian distance

The default chosen heuristic was euclidean distance, but performance varies based on choice of heuristic.

The pseudo code for A* search is given below:



```
A* search {

closed list = [ ]
open list = [start node]

    do {
            if open list is empty then {
                    return no solution
            }
            n = heuristic best node
            if n == final node then {
                    return path from start to goal node
            }
            foreach direct available node do{
                    if current node not in open and not in closed list do {
                            add current node to open list and calculate heuristic
                            set n as his parent node
                    }
                    else{
                            check if path from star node to current node is
                            better;
                            if it is better calculate heuristics and transfer
                            current node from closed list to open list
                            set n as his parrent node
                    }
            delete n from open list
            add n to closed list
    } while (open list is not empty)

}
```

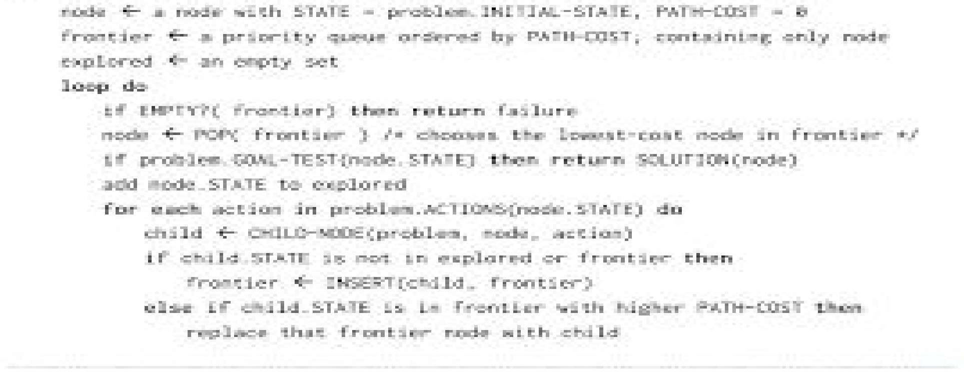
Performance of A* :

- **Completeness :** A* is **complete** as it always terminates for finite graphs.
- **Optimality :** A* is **optimal** as it always finds the optimal path from start state to goal state. This is because the OPEN frontier always contains a node n* which lies along the optimal path.
- **Time Complexity : Exponential**
- **Space Complexity : O(n)** where n is the number of nodes.

Uniform Cost Search (UCS)

Uniform Cost Search (UCS) is an uninformed search technique, where the least cost unexpanded node is expanded at each stage. It is follows a greedy approach to reaching the goal state. If the cost of each edge is the same, UCS can be effectively reduced to BFS.

The pseudo code for UCS is given below :



Uniform-Cost Search

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    frontier ← a priority queue ordered by PATH-COST, containing only node
    explored ← an empty set
    loop do
        if EMPTY?( frontier) then return failure
        node ← POP( frontier ) /* chooses the lowest-cost node in frontier */
        if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                frontier ← INSERT(child, frontier)
            else if child.STATE is in frontier with higher PATH-COST then
                replace that frontier node with child
```

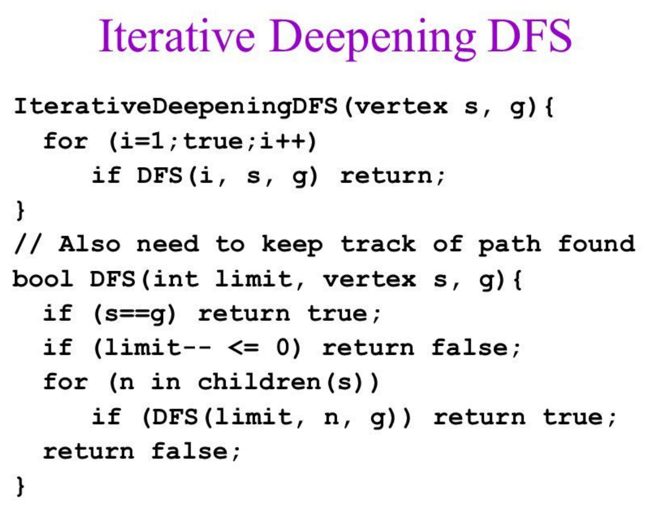
Performance of UCS :

- **Completeness :** UCS is **complete** as it guarantees to find a solution if there exists one.
- **Optimality :** UCS is **optimal** as it always finds the optimal solution from start state to goal state. This is however at the expense of exponential space and time requirements.
- **Time Complexity : $O(b^{ceil(c^*/e)})$** where c* is the total cost of the optimal solution.
- **Space Complexity : $O(b^{ceil(c^*/e)})$**

<u>Iterative Deepening Search (IDS)</u>

Iterative Deepening Search(IDS) or more specifically **Iterative Deepening Depth-First Search** is a state space/graph search strategy in which a depth-limited version of depth-first search is run repeatedly with increasing depth limits until the goal is found. IDDFS is equivalent to breadth-first search, but uses much less memory; at each iteration, it visits the nodes in the search tree in the same order as depth-first search, but the cumulative order in which nodes are first visited is effectively breadth-first.

The pseudo code for IDS is given below :



Iterative Deepening DFS

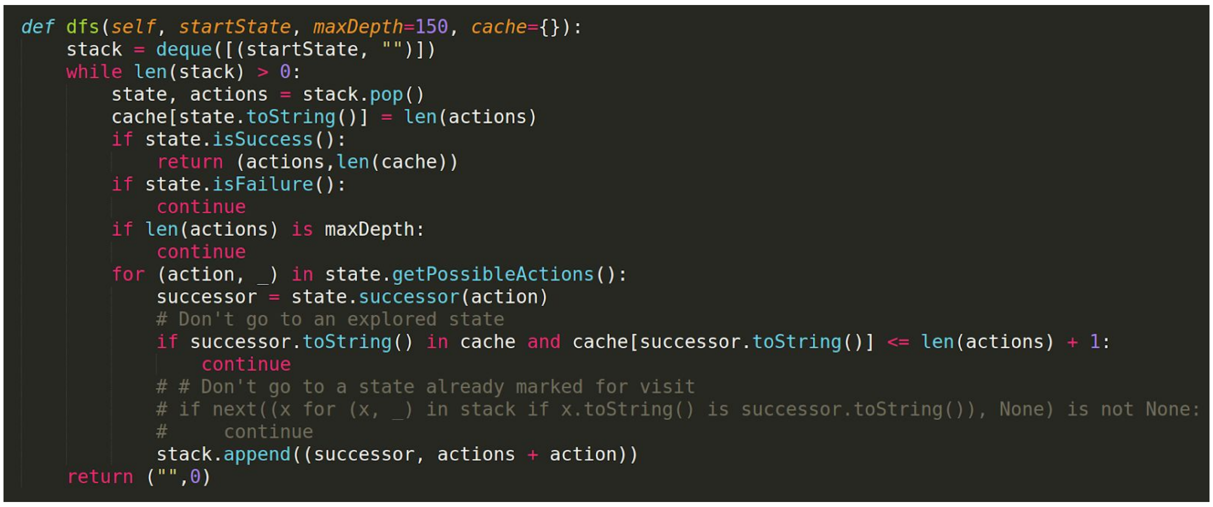
```
IterativeDeepeningDFS(vertex s, g){
  for (i=1;true;i++)
     if DFS(i, s, g) return;
}
// Also need to keep track of path found
bool DFS(int limit, vertex s, g){
  if (s==g) return true;
  if (limit-- <= 0) return false;
  for (n in children(s))
     if (DFS(limit, n, g)) return true;
  return false;
}
```

Performance of IDS :

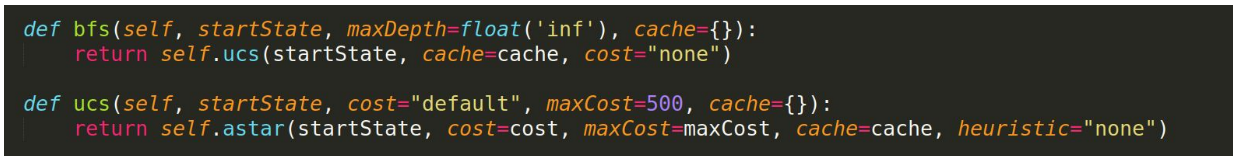
- **Completeness :** IDS is **complete** as it guarantees to find a solution if there exists one.
- **Optimality :** IDS is **optimal** because it iterates over each level of the graph ensuring that the optimal solution is obtained at some level.
- **Time Complexity : O(b^d)** where b is the branching factor and d is depth of state space.
- **Space Complexity : O(bd)**

CODE SNIPPETS

Code snippets for the search algorithms mentioned above are given below:

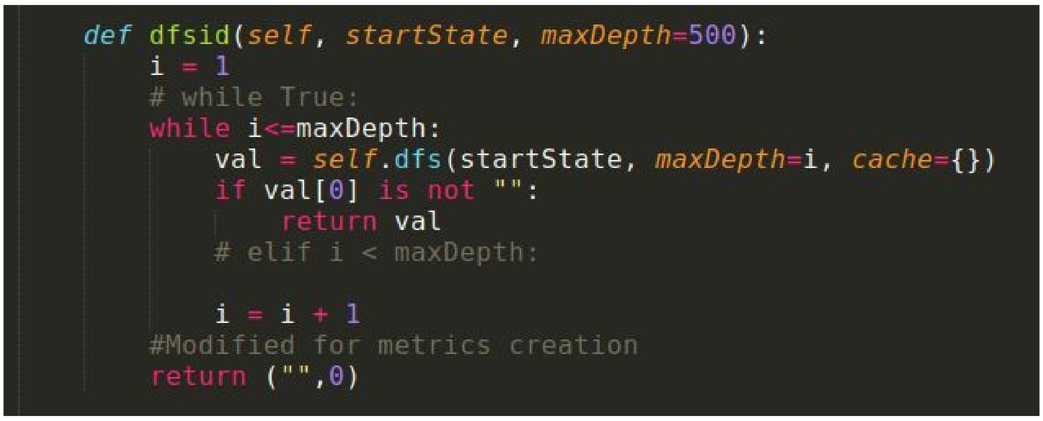


```python
def dfs(self, startState, maxDepth=150, cache={}):
    stack = deque([(startState, "")])
    while len(stack) > 0:
        state, actions = stack.pop()
        cache[state.toString()] = len(actions)
        if state.isSuccess():
            return (actions,len(cache))
        if state.isFailure():
            continue
        if len(actions) is maxDepth:
            continue
        for (action, _) in state.getPossibleActions():
            successor = state.successor(action)
            # Don't go to an explored state
            if successor.toString() in cache and cache[successor.toString()] <= len(actions) + 1:
                continue
            # # Don't go to a state already marked for visit
            # if next((x for (x, _) in stack if x.toString() is successor.toString()), None) is not None:
            #     continue
            stack.append((successor, actions + action))
    return ("",0)
```



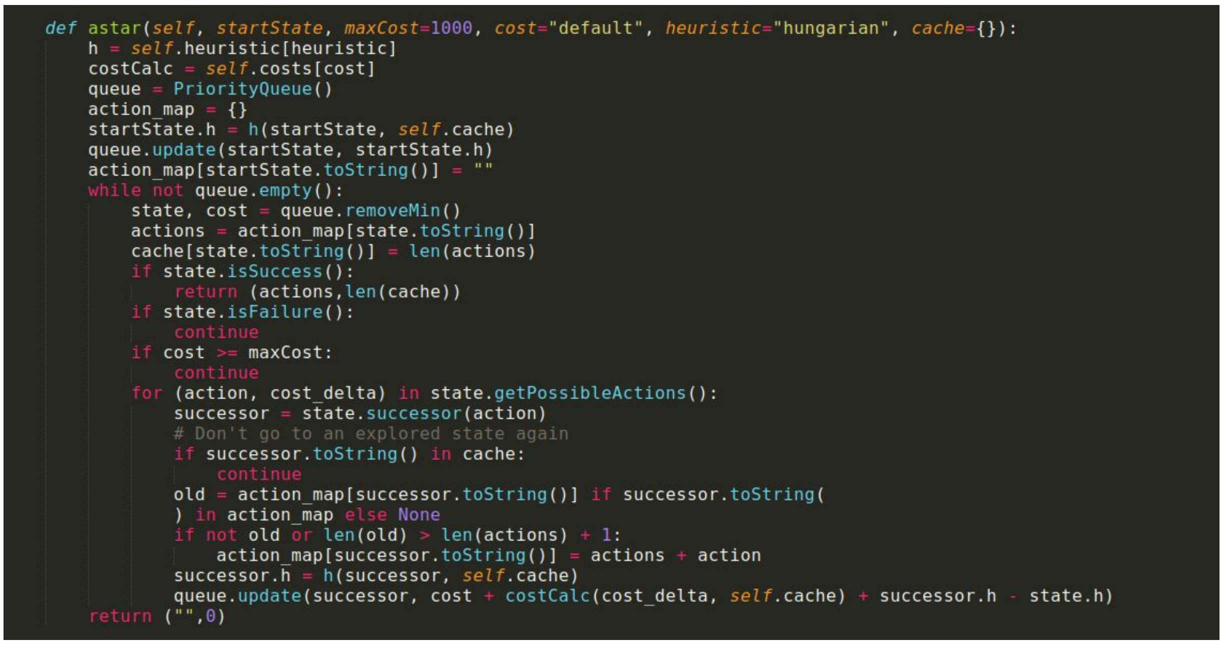
```python
def bfs(self, startState, maxDepth=float('inf'), cache={}):
    return self.ucs(startState, cache=cache, cost="none")

def ucs(self, startState, cost="default", maxCost=500, cache={}):
    return self.astar(startState, cost=cost, maxCost=maxCost, cache=cache, heuristic="none")
```



```python
    def dfsid(self, startState, maxDepth=500):
        i = 1
        # while True:
        while i<=maxDepth:
            val = self.dfs(startState, maxDepth=i, cache={})
            if val[0] is not "":
                return val
            # elif i < maxDepth:

            i = i + 1
        #Modified for metrics creation
        return ("",0)
```

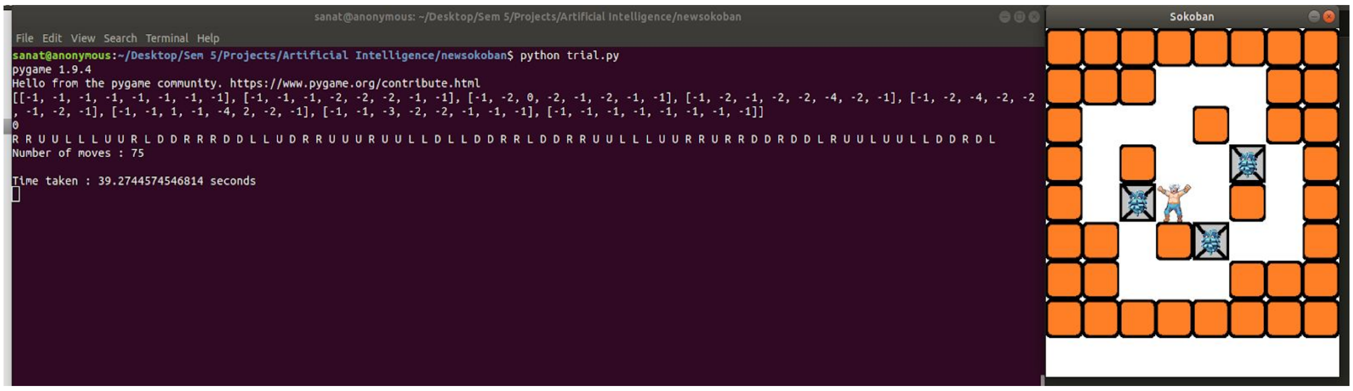


```python
    def astar(self, startState, maxCost=1000, cost="default", heuristic="hungarian", cache={}):
        h = self.heuristic[heuristic]
        costCalc = self.costs[cost]
        queue = PriorityQueue()
        action_map = {}
        startState.h = h(startState, self.cache)
        queue.update(startState, startState.h)
        action_map[startState.toString()] = ""
        while not queue.empty():
            state, cost = queue.removeMin()
            actions = action_map[state.toString()]
            cache[state.toString()] = len(actions)
            if state.isSuccess():
                return (actions,len(cache))
            if state.isFailure():
                continue
            if cost >= maxCost:
                continue
            for (action, cost_delta) in state.getPossibleActions():
                successor = state.successor(action)
                # Don't go to an explored state again
                if successor.toString() in cache:
                    continue
                old = action_map[successor.toString()] if successor.toString(
                ) in action_map else None
                if not old or len(old) > len(actions) + 1:
                    action_map[successor.toString()] = actions + action
                successor.h = h(successor, self.cache)
                queue.update(successor, cost + costCalc(cost_delta, self.cache) + successor.h - state.h)
        return ("",0)
```

RESULTS

The game was played using different search algorithms and the outputs for level 1 are as follows:

A* Search

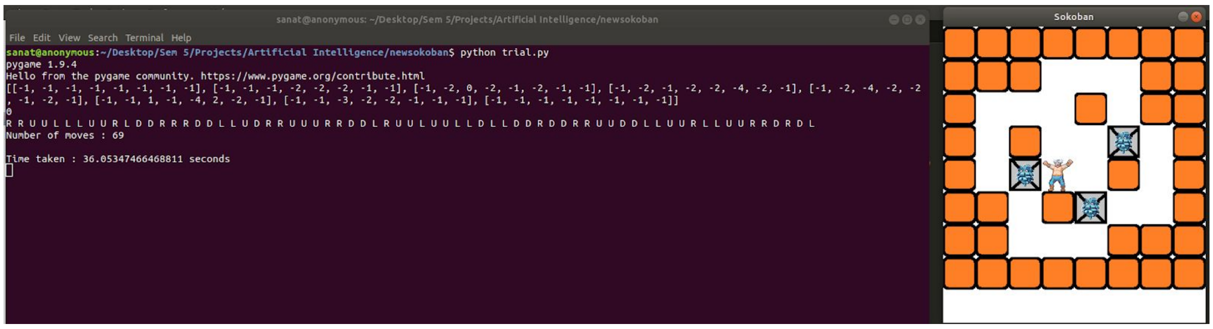


```
sanat@anonymous:~/Desktop/Sem 5/Projects/Artificial Intelligence/newsokoban$ python trial.py
pygame 1.9.4
Hello from the pygame community. https://www.pygame.org/contribute.html
[[-1, -1, -1, -1, -1, -1, -1, -1], [-1, -1, -1, -2, -2, -2, -1, -1], [-1, -2, 0, -2, -1, -2, -1, -1], [-1, -2, -1, -2, -2, -4, -2, -1], [-1, -2, -4, -2, -2
, -1, -2, -1], [-1, -1, 1, -1, -4, 2, -2, -1], [-1, -1, -3, -2, -2, -1, -1, -1], [-1, -1, -1, -1, -1, -1, -1, -1]]
0
R R U U L L L U U R L D D R R R D D L L U D R R U U U R U U L L D L L D D R R L D D R R U U L L L U U R R U R R D D R D D L R U U L U U L L D D R D L
Number of moves : 75

Time taken : 39.2744574546814 seconds
```

- Number of moves : 75
- Time taken : 39.27 seconds
- Moves sequence : R R U U L L L U U R L D D R R R D D L L U D R R U U U R U U L L D L L D D R R L D D R R U U L L L U U R R U R R D D R D D L R U U L U U L L D D R D L

BFS Search

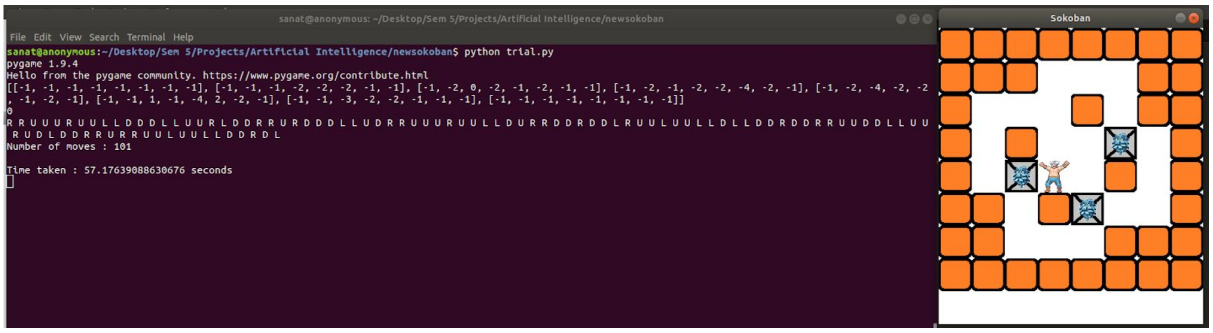


```
sanat@anonymous:~/Desktop/Sem 5/Projects/Artificial Intelligence/newsokoban$ python trial.py
pygame 1.9.4
Hello from the pygame community. https://www.pygame.org/contribute.html
[[-1, -1, -1, -1, -1, -1, -1, -1], [-1, -1, -1, -2, -2, -2, -1, -1], [-1, -2, 0, -2, -1, -2, -1, -1], [-1, -2, -1, -2, -2, -4, -2, -1], [-1, -2, -4, -2, -2
, -1, -2, -1], [-1, -1, 1, -1, -4, 2, -2, -1], [-1, -1, -3, -2, -2, -1, -1, -1], [-1, -1, -1, -1, -1, -1, -1, -1]]
0
R R U U L L L U U R L D D R R R D D L L U D R R U U U R R D D L R U U L U U L L D L L D D R D D R R U U D D L L U U R L L U U R R D R D L
Number of moves : 69

Time taken : 36.05347466468811 seconds
```

- Number of moves : 69
- Time taken : 36.05 seconds
- Moves sequence : R R U U L L L U U R L D D R R R D D L L U D R R U U U R R D D L R U U L U U L L D L L D D R D D R R U U D D L L U U R L L U U R R D R D L

DFS Search

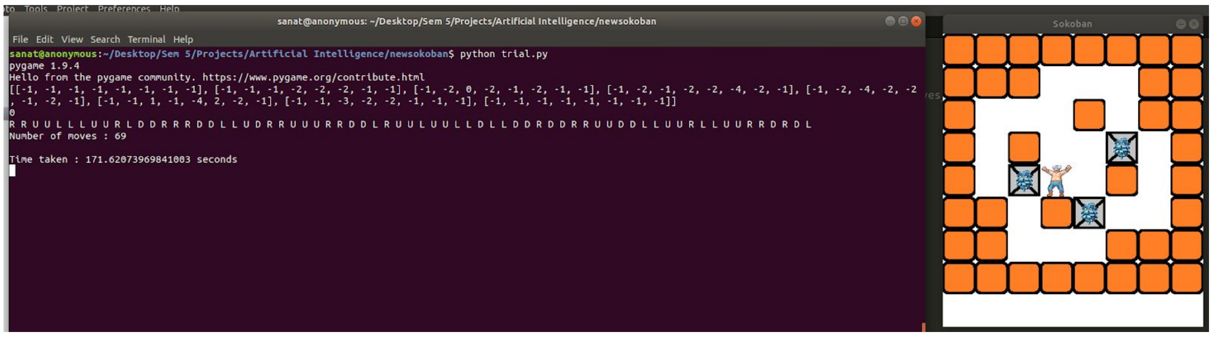

sanat@anonymous: ~/Desktop/Sem 5/Projects/Artificial Intelligence/newsokoban
File Edit View Search Terminal Help
Sokoban

```
sanat@anonymous:~/Desktop/Sem 5/Projects/Artificial Intelligence/newsokoban$ python trial.py
pygame 1.9.4
Hello from the pygame community. https://www.pygame.org/contribute.html
[[-1, -1, -1, -1, -1, -1, -1, -1], [-1, -1, -1, -2, -2, -2, -1, -1], [-1, -2, 0, -2, -1, -2, -1, -1], [-1, -2, -1, -2, -2, -4, -2, -1], [-1, -2, -4, -2, -2
, -1, -2, -1], [-1, -1, 1, -1, -4, 2, -2, -1], [-1, -1, -3, -2, -2, -1, -1, -1], [-1, -1, -1, -1, -1, -1, -1, -1]]
0
R R U U U R U U L L D D D L L U U R L D D R R U R D D D L L U D R R U U U R U U L L D U R R D D R D D L R U U L U U L L D L L D D R D D R R U U D D L L U U
 R U D L D D R R U R R U U L U U L L D D R D L
Number of moves : 101

Time taken : 57.17639088630676 seconds
```

- Number of moves : 101
- Time taken : 57.17 seconds
- Moves sequence : R R U U U R U U L L D D D L L U U R L D D R R U R D D D L L U D R R U U U R U U L L D U R R D D R D D L R U U L U U L L D L L D D R D D R R U U D D L L U U R U D L D D R R U R R U U L U U L L D D R D L

IDS Search

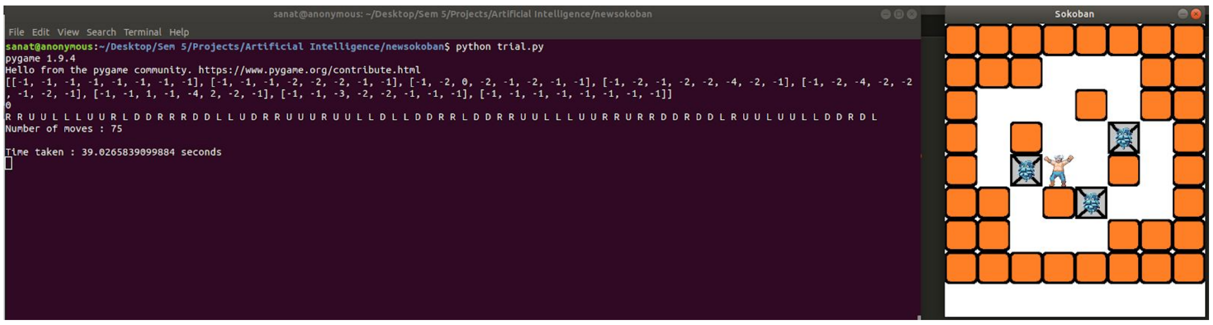

sanat@anonymous: ~/Desktop/Sem 5/Projects/Artificial Intelligence/newsokoban
File Edit View Search Terminal Help
Sokoban

```
sanat@anonymous:~/Desktop/Sem 5/Projects/Artificial Intelligence/newsokoban$ python trial.py
pygame 1.9.4
Hello from the pygame community. https://www.pygame.org/contribute.html
[[-1, -1, -1, -1, -1, -1, -1, -1], [-1, -1, -1, -2, -2, -2, -1, -1], [-1, -2, 0, -2, -1, -2, -1, -1], [-1, -2, -1, -2, -2, -4, -2, -1], [-1, -2, -4, -2, -2
, -1, -2, -1], [-1, -1, 1, -1, -4, 2, -2, -1], [-1, -1, -3, -2, -2, -1, -1, -1], [-1, -1, -1, -1, -1, -1, -1, -1]]
0
R R U U L L L U U R L D D R R R D D L L U D R R U U U R R D D L R U U L U U L L D L L D D R D D R R U U D D L L U U R L L U U R R D R D L
Number of moves : 69

Time taken : 171.62073969841003 seconds
```

- Number of moves : 69
- Time taken : 171.62 seconds
- Moves sequence : R R U U L L L U U R L D D R R R D D L L U D R R U U U R R D D L R U U L U U L L D L L D D R D D R R U U D D L L U U R L L U U R R D R D L

UCS Search



```
sanat@anonymous: ~/Desktop/Sem 5/Projects/Artificial Intelligence/newsokoban
File  Edit  View  Search  Terminal  Help
sanat@anonymous:~/Desktop/Sem 5/Projects/Artificial Intelligence/newsokoban$ python trial.py
pygame 1.9.4
Hello from the pygame community. https://www.pygame.org/contribute.html
[[-1, -1, -1, -1, -1, -1, -1, -1], [-1, -1, -1, -2, -2, -2, -1, -1], [-1, -2, 0, -2, -1, -2, -1, -1], [-1, -2, -1, -2, -2, -4, -2, -1], [-1, -2, -4, -2, -2
, -1, -2, -1], [-1, -1, 1, -1, -4, 2, -2, -1], [-1, -1, -3, -2, -2, -1, -1, -1], [-1, -1, -1, -1, -1, -1, -1, -1]]
0
R R U U L L L U U R L D D R R R D D L L U D R R U U U R U U L L D L L D D R R L D D R R U U L L L U U R R U R R D D R D D L R U U L U U L L D D R D L
Number of moves : 75

Time taken : 39.0265839099884 seconds
```

- Number of moves : 75
- Time taken : 39.02 seconds
- Moves sequence : R R U U L L L U U R L D D R R R D D L L U D R R U U U R U U L L D L L D D R R L D D R R U U L L L U U R R U R R D D R D D L R U U L U U L L D D R D L

CONCLUSIONS

Thus, the mentioned search algorithms were compared in terms of number of moves taken to solve a level and the time taken to do so. The following observations apply to our game and are not generalized.

- From the above results, it can be observed that BFS performed the best in terms of number of moves needed to solve the level. However, on further testing it was found that A* was the best algorithm overall for our problem statement.
- DFS performed the worst in the chosen level and overall due to repeated moves cycles.
- IDS solved the chosen level in the least number of moves, but the search time was the highest due to iteration over each depth of states.
- UCS and A* performed almost identically for this level, both in terms of time taken and number of moves. However, in other levels it was observed that the greedy approach taken by UCS increased number of moves.