FOUNDATIONS OF AI COURSEWORK

Aneeshaa S Chowdhry

UNIVERSITY OF SOUTHAMPTON ID: 30439485

Tree Search Algorithms

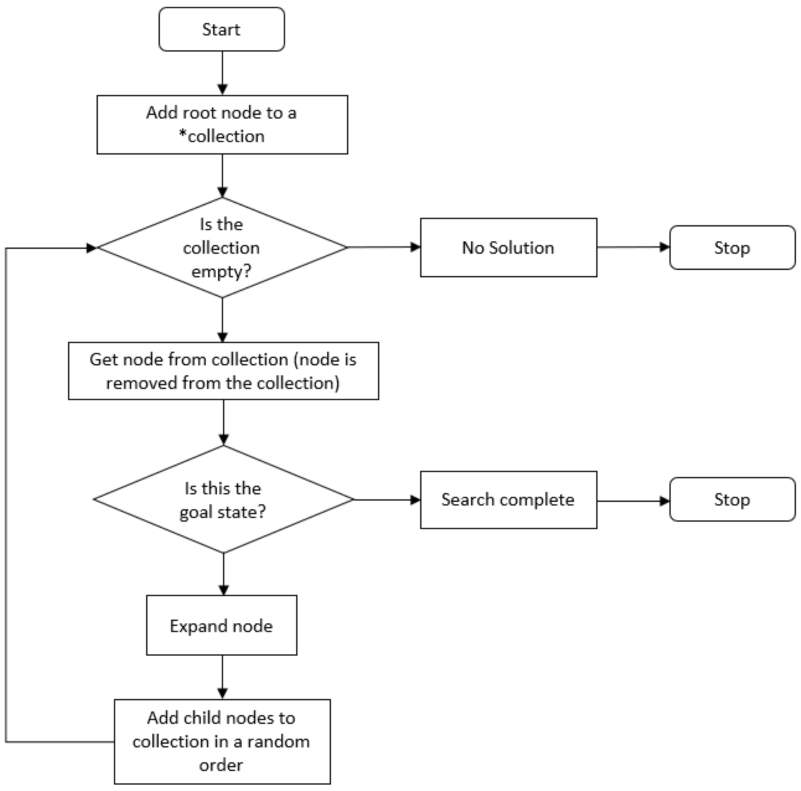
1. Abstract

This report demonstrates the approach for implementation, possible issues/enhancements of different Tree-search algorithms: Breadth-First search, Depth-First search, Iterative deepening search, and A* search in Part-I, to solve the Blocks world puzzle.

2. Approach

Although Python is closer to pseudo code, easier to read/understand, the Algorithms have been implemented in Java, because of familiarity, static typing feature, and in order to understand data structures better. For our experiments, we start at an initial position that is close to the goal state, as the programs tend to fail with Out of Memory errors for uninformed searches. However, working of the algorithm for an initial state that is more farther than the goal state is also demonstrated in subsequent sections.

The flowchart below shows the process we'll follow for implementing different search methods.
Note: '@' represents the agent.
**collection – this is a linear data structure*, that is used to store a node/current state. It differs between each search type. This is elaborated on in upcoming sections.


Start
Add root node to a *collection
Is the collection empty?
No Solution
Stop
Get node from collection (node is removed from the collection)
Is this the goal state?
Search complete
Stop
Expand node
Add child nodes to collection in a random order

2.1 Breadth-First Search

To implement this uninformed brute-force search strategy, we must provision in the above illustrated process, a way to *expand the shallowest node first*. Since, the fringe of a Breadth-First search is a FIFO(first-in first-out) queue, we leverage the *'Queue' data structure* to store nodes in the fringe. The algorithm fails with an out of memory error if the initial state is many moves away from the goal state. This is due to the drawback with the algorithm, that it stores all nodes in memory, hence *consuming excess space*. We also observe, that the algorithm takes large amount of time in comparison with other types.

2.2 Depth-First Search

Although Depth-First search algorithm is susceptible to loops, and problems where the depth of the search space is infinite, it consumes much lower space than the Breadth-First search. This is because, once a node is expanded, it can be removed from memory as all its child nodes have been expanded. The fringe in this search technique follows the Last-in First-out order. Hence, we can utilize the *'Stack' data structure to store, and expand the nodes in the fringe*.
A possible enhancement to this technique, would be to store/keep track of states that have been visited before, in order to avoid repetition, and gain immunity against loops.

2.3 Iterative Deepening Search

This variant of Depth-First search runs the Depth-First search with increasing depth limits until the goal state is found. Once again, we use the 'Stack' data structure to store nodes in the fringe. The only difference in implementation is the check for condition on 'maxDepth' or the depth bound. We start with a maxDepth =0, and repeat the search by incrementing maxDepth until we find a goal state. Each time a node is to be expanded, the depth of the node is compared with maxDepth. The node is expanded only if the depth has not yet reached maxDepth. This algorithm has similar time consumption and space consumption properties as Depth-First Search.

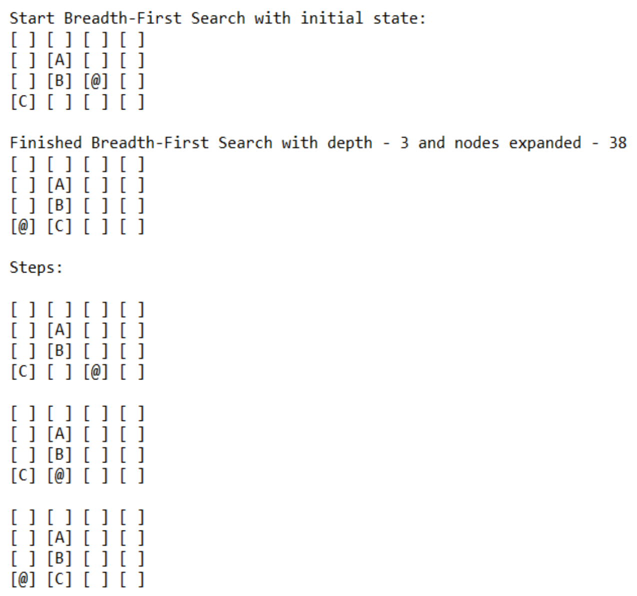
2.4 A* Heuristic Search

Before zeroing in on the heuristic function to guide the A* star search, we consider a number of *heuristic functions such as 'no. of misplaced tiles', 'Euclidean distance', and 'Manhattan Distance'.* But the Manhattan distance is a dominant, consistent, and admissible heuristic for Grid Maps that allow movement in 4 directions. In order to expand the cheapest/lowest cost node first, we utilize the *'Priority Queue'* data structure to store nodes in the fringe. This structure allows us to store Nodes with the lowest cost (calculated using cost incurred so far + Manhattan Distance), at the front of the queue.

3. Evidence

3.1 Breadth-First Search

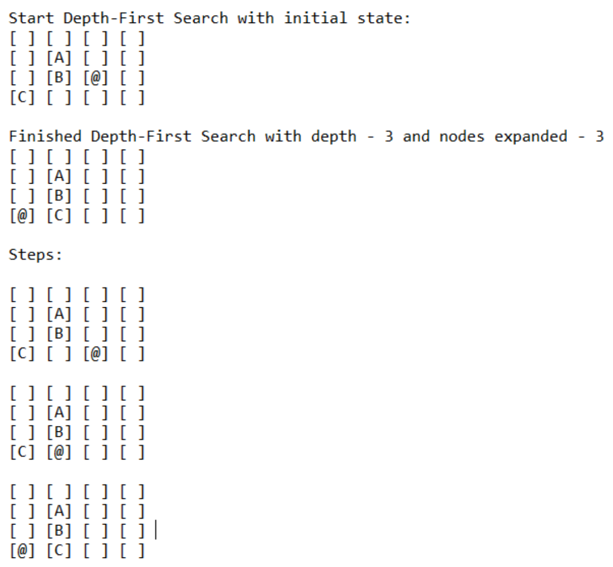
This algorithm takes a long time to reach a solution, and runs out of Heap space before it does so. This is because the algorithm is designed to store all the nodes in memory, and the number of nodes stored increases exponentially.
To demonstrate the working of this algorithm, we start with an initial state that is not too many moves far away from the goal state. Observe from the results below, the number of nodes expanded even with few misplaced blocks.



```
Start Breadth-First Search with initial state:
[ ] [ ] [ ] [ ]
[ ] [A] [ ] [ ]
[ ] [B] [@] [ ]
[C] [ ] [ ] [ ]

Finished Breadth-First Search with depth - 3 and nodes expanded - 38
[ ] [ ] [ ] [ ]
[ ] [A] [ ] [ ]
[ ] [B] [ ] [ ]
[@] [C] [ ] [ ]

Steps:

[ ] [ ] [ ] [ ]
[ ] [A] [ ] [ ]
[ ] [B] [ ] [ ]
[C] [ ] [@] [ ]

[ ] [ ] [ ] [ ]
[ ] [A] [ ] [ ]
[ ] [B] [ ] [ ]
[C] [@] [ ] [ ]

[ ] [ ] [ ] [ ]
[ ] [A] [ ] [ ]
[ ] [B] [ ] [ ]
[@] [C] [ ] [ ]
```

3.2 Depth-First Search

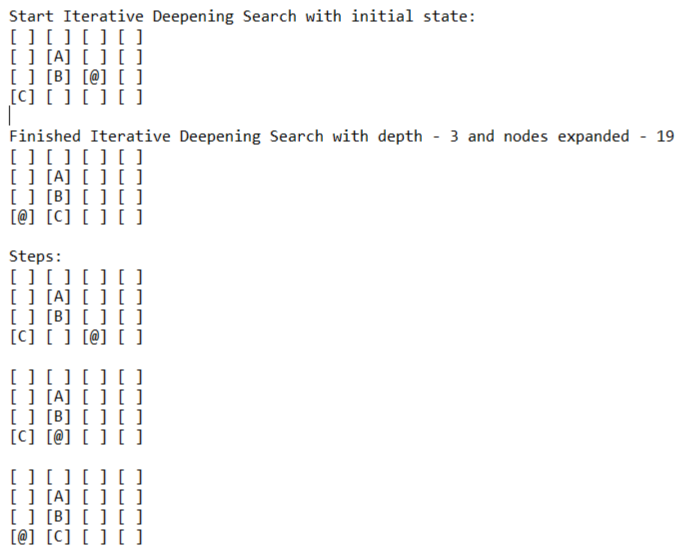
Although Depth-First Search doesn't run into memory issues for initial states that are many moves farther away from the goal state, evidence is shown below for the same initial state used earlier, just for better readability/visualization.



```
Start Depth-First Search with initial state:
[ ] [ ] [ ] [ ]
[ ] [A] [ ] [ ]
[ ] [B] [@] [ ]
[C] [ ] [ ] [ ]

Finished Depth-First Search with depth - 3 and nodes expanded - 3
[ ] [ ] [ ] [ ]
[ ] [A] [ ] [ ]
[ ] [B] [ ] [ ]
[@] [C] [ ] [ ]

Steps:

[ ] [ ] [ ] [ ]
[ ] [A] [ ] [ ]
[ ] [B] [ ] [ ]
[C] [ ] [@] [ ]

[ ] [ ] [ ] [ ]
[ ] [A] [ ] [ ]
[ ] [B] [ ] [ ]
[C] [@] [ ] [ ]

[ ] [ ] [ ] [ ]
[ ] [A] [ ] [ ]
[ ] [B] [ ] [ ]
[@] [C] [ ] [ ]
```

3.3 Iterative Deepening Search

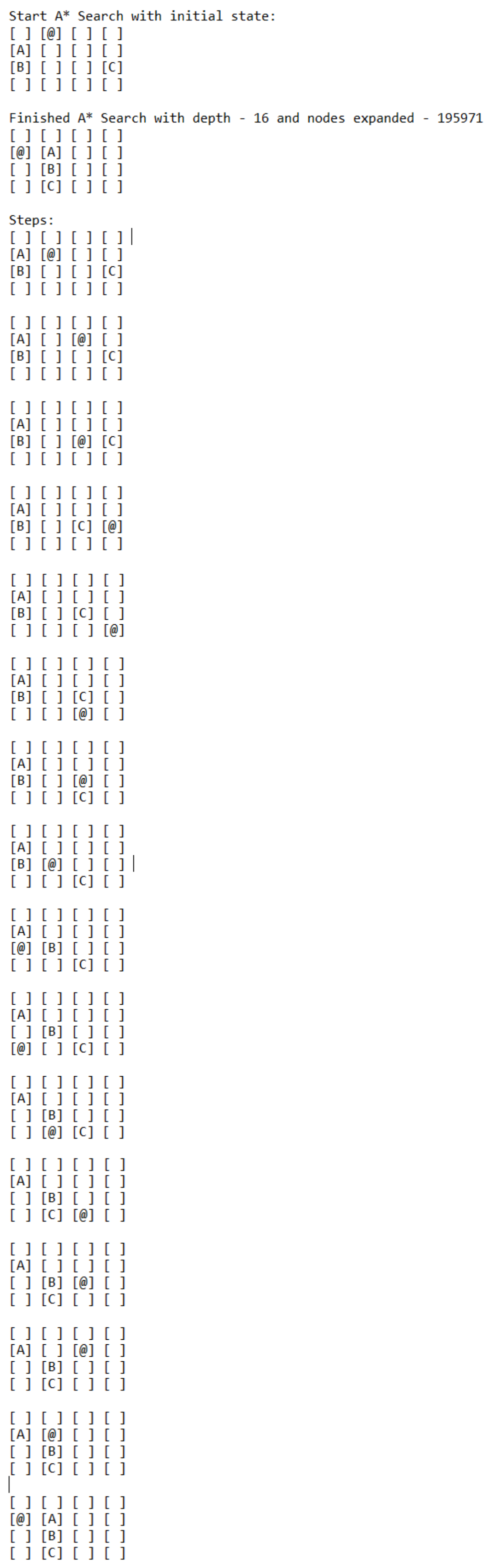
The Iterative Deepening Search does well with initial states that are farther away from goal state too. As one might expect, it doesn't take longer to complete because of its repeated traversal from the root node.



```
Start Iterative Deepening Search with initial state:
[ ] [ ] [ ] [ ]
[ ] [A] [ ] [ ]
[ ] [B] [@] [ ]
[C] [ ] [ ] [ ]

Finished Iterative Deepening Search with depth - 3 and nodes expanded - 19
[ ] [ ] [ ] [ ]
[ ] [A] [ ] [ ]
[ ] [B] [ ] [ ]
[@] [C] [ ] [ ]

Steps:
[ ] [ ] [ ] [ ]
[ ] [A] [ ] [ ]
[ ] [B] [ ] [ ]
[C] [ ] [@] [ ]

[ ] [ ] [ ] [ ]
[ ] [A] [ ] [ ]
[ ] [B] [ ] [ ]
[C] [@] [ ] [ ]

[ ] [ ] [ ] [ ]
[ ] [A] [ ] [ ]
[ ] [B] [ ] [ ]
[@] [C] [ ] [ ]
```

3.4 A* Search

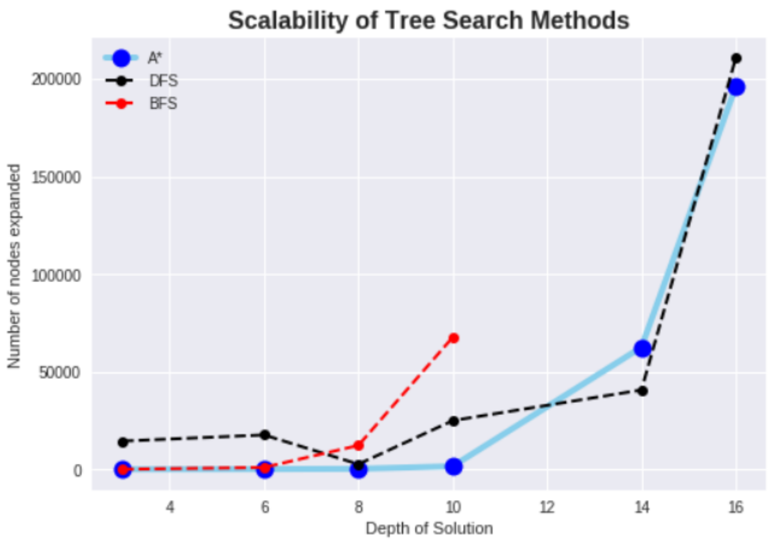
We exhibit the results for A* search for the problem when the initial/start state is farther away from the goal node than earlier, as shown below. Starting at the same initial state, if we ran Iterative deepening search, we observe that the number of nodes expanded is 37490042 which is about 191 times the number of nodes expanded by A* search.



```
Start A* Search with initial state:
[ ] [@] [ ] [ ]
[A] [ ] [ ] [ ]
[B] [ ] [ ] [C]
[ ] [ ] [ ] [ ]

Finished A* Search with depth - 16 and nodes expanded - 195971
[ ] [ ] [ ] [ ]
[@] [A] [ ] [ ]
[ ] [B] [ ] [ ]
[ ] [C] [ ] [ ]

Steps:
[ ] [ ] [ ] [ ]
[A] [@] [ ] [ ]
[B] [ ] [ ] [C]
[ ] [ ] [ ] [ ]

[ ] [ ] [ ] [ ]
[A] [ ] [@] [ ]
[B] [ ] [ ] [C]
[ ] [ ] [ ] [ ]

[ ] [ ] [ ] [ ]
[A] [ ] [ ] [ ]
[B] [ ] [@] [C]
[ ] [ ] [ ] [ ]

[ ] [ ] [ ] [ ]
[A] [ ] [ ] [ ]
[B] [ ] [C] [@]
[ ] [ ] [ ] [ ]

[ ] [ ] [ ] [ ]
[A] [ ] [ ] [ ]
[B] [ ] [C] [ ]
[ ] [ ] [ ] [@]

[ ] [ ] [ ] [ ]
[A] [ ] [ ] [ ]
[B] [ ] [C] [ ]
[ ] [ ] [@] [ ]

[ ] [ ] [ ] [ ]
[A] [ ] [ ] [ ]
[B] [ ] [@] [ ]
[ ] [ ] [C] [ ]

[ ] [ ] [ ] [ ]
[A] [ ] [ ] [ ]
[B] [@] [ ] [ ]
[ ] [ ] [C] [ ]

[ ] [ ] [ ] [ ]
[A] [ ] [ ] [ ]
[@] [B] [ ] [ ]
[ ] [ ] [C] [ ]

[ ] [ ] [ ] [ ]
[A] [ ] [ ] [ ]
[ ] [B] [ ] [ ]
[@] [ ] [C] [ ]

[ ] [ ] [ ] [ ]
[A] [ ] [ ] [ ]
[ ] [B] [ ] [ ]
[ ] [@] [C] [ ]

[ ] [ ] [ ] [ ]
[A] [ ] [ ] [ ]
[ ] [B] [ ] [ ]
[ ] [C] [@] [ ]

[ ] [ ] [ ] [ ]
[A] [ ] [ ] [ ]
[ ] [B] [@] [ ]
[ ] [C] [ ] [ ]

[ ] [ ] [ ] [ ]
[A] [ ] [@] [ ]
[ ] [B] [ ] [ ]
[ ] [C] [ ] [ ]

[ ] [ ] [ ] [ ]
[A] [@] [ ] [ ]
[ ] [B] [ ] [ ]
[ ] [C] [ ] [ ]

[ ] [ ] [ ] [ ]
[@] [A] [ ] [ ]
[ ] [B] [ ] [ ]
[ ] [C] [ ] [ ]
```

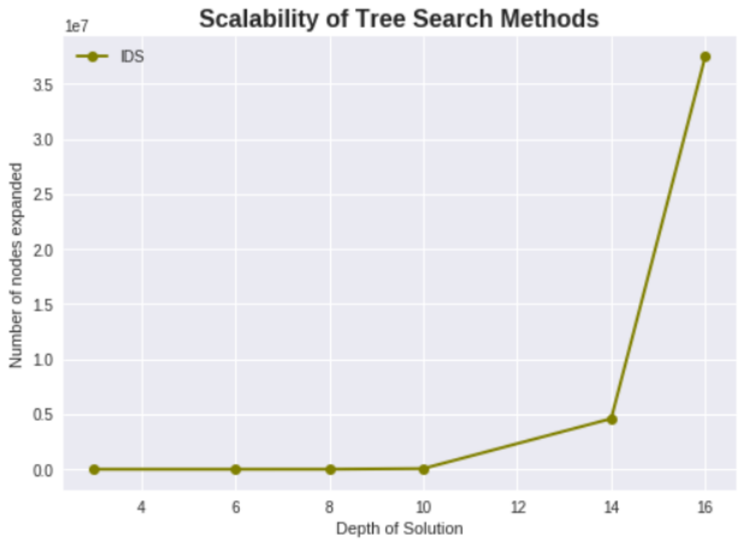
4. Scalability

In order to evaluate the scalability of the search algorithms to more complex problems/larger search spaces, we repeatedly run the algorithms for initial states with varying depth of solution/goal state in the search space.



The Iterative Deepening search results have been plotted in a separate graph due to the difference in scale.

4.1 Inference



1. The Breadth first search fails because of memory issues for initial states with solutions deeper than 10 nodes. For simpler problems as well, the number of nodes generated/stored in memory rises very quickly, in comparison with other algorithms.
2. The Depth first search performs almost as well as the A* search, but the results are inconsistent. This is because, it randomly chooses a node to expand, and it might luckily pick a node that is closer to goal state, and end up finding the solution faster.
3. The A star search exhibits more consistency/reliability in performance. However, the number of nodes expanded rises exponentially, just like the other tree search algorithms.
4. The Iterative deepening search however, had to be plot on a different graph, due to the large difference in scale. The number of nodes expanded are not 0 for depth lower than 10, but : 19, 402, 3171, 60158, 4595907, 37490042, for depths: 3,6,8,10,14,16 respectively. Once again, the number of nodes expanded for a depth of 16, is several times more than the number by A star search which expands 195971 nodes.

4.2 Conclusion

Concretely, the A* search algorithm consistently gives us good results, followed by Depth First search, Iterative Deepening search, and lastly, Breadth First search. While running experiments, it is also observed that the Iterative deepening search consumes more time in comparison with A* search. Breadth First search is unusable for larger/complex problems. Also, Depth First search gives inconsistent results, leaving us to only hope that it luckily chooses an optimal path to expand. Hence, A* search is a more reliable solution to problems.

5. Extras and Limitations

Addressing the limitations first, we must acknowledge that our code has the grid size, and block positions hardcoded. This could be modified to induce flexibility in code, by reading input from user. Also, even if PriorityQueue used in code orders nodes in ascending order of cost, including a tie-breaker would help expand the right nodes, and find the solution faster. But this has not been implemented.

Although grid size is hardcoded, code can run with larger grid sizes by changing the size/block position values.

6. References

[1] R. Graham H. McCabe S. Sheridan "Pathfinding in Computer Games" The ITB Journal vol. 4 no. 2 2003.

[2] I. Millington dan J. Funge Artificial Intelligence for Games Oxford UK: Elsevier 2009.

[3] Z. Xu M. Van Doren A Museum Visitors Guide with the A* Pathfinding Algorithm 2011

[4] Artificial intelligence—a modern approach by Stuart Russell and Peter Norvig, Prentice Hall.

7. Appendix

Program has been written by taking inspiration from the sources above(idea of using priority queue for A start search. Parts downloaded have not been included below; game board display, calculation of Manhattan distance, check for invalid move, and ideas for exception handling.

Main Class:

```
import java.util.*;

/* this Class has the main method to run the different search algorithms */

public class Search {

	public static void main(String[] args) {

		//feed in info on gridSize(4x4), position of block A, position of 
block B, position of block C, position of  Agent
		// passed into state class - initial state information
		// making sure block positions are close to Goal state for BFS, else 
it throws out of memory error
		State start = new State(new int[] {4,4}, new int[] {1,2}, new int[] 
{1,3}, new int[] {4, 3}, new int[] {2, 1});
		State goal = new State(null, new int[] {2,2}, new int[] {2,3}, new 
int[] {2, 4}, null);

		Node node = new Node(start);
```

```
            Search search = new Search();
//          Runs search methods
//          search.BFS(node, goal);
            search.DFS(node, goal);
//          search.IDS(node, goal);
//          search.aStar(node, goal);
      }

      //Depth-First Search

      public void DFS(Node root, State goal)
      {
            int nodesExpanded = 0;
            // Stack is a Last In First Out (LIFO) data structure, to keep track
of DFS frontier
            Stack<Node> stack = new Stack<>();

            // root is the initial or start state
            System.out.println("Start Depth-First Search with initial state:\n"
+ root.getState());
            stack.add(root);          // to expand root
            while(!stack.isEmpty())
            {
                  ArrayList<Node> successors = new ArrayList<>();

                  //get node from stack. last node is popped or read first, it
is a stack data structure
                  Node current = stack.pop();

                  //Check if we're at the goal state
                  if(Goal_test(current.getState(), goal))
                  {
                        ArrayList<Node> steps = current.sequence(current);
                        System.out.println("Finished Depth-First Search with
depth - " + current.getCost() + " and nodes expanded - " + nodesExpanded + "\n" +
current.getState() + "\nSteps:\n");
                        /*
                        for(Node step : steps) {
                              System.out.println(step.getState());
                        }
                        */
                        break;
                  }
                  //Expands Node if solution not found. Branches are going to be
all possible moves from current state
                  nodesExpanded++;
                  successors = Add_child_nodes(current);
                  //Add successors to the stack (randomly for DFS)
                  Collections.shuffle(successors);
                  for(Node child : successors) {
                        if(child != null) {
                              stack.add(child);
                        }
                  }
            }
      }

      //Breadth-First Search
```

```
	public void BFS(Node root, State goal) {
		int nodesExpanded = 0;
		// queue to keep track of frontier
		Queue<Node> queue = new LinkedList<Node>();

		// root is nothing but the initial or start state
		System.out.println("Start Breadth-First Search with initial 
state:\n" + root.getState());
		queue.add(root);
		while(!queue.isEmpty()) {
			ArrayList<Node> successors = new ArrayList<>();
			//Remove queue head
			Node current = queue.remove();
			//Check if the solution is found
			if(Goal_test(current.getState(), goal)) {
				ArrayList<Node> steps = current.sequence(current);
				System.out.println("Finished Breadth-First Search with 
depth - " + current.getCost() + " and nodes expanded - " + nodesExpanded + "\n" + 
current.getState() + "\nSteps:\n");
				/*
				for(Node step : steps) {
					System.out.println(step.getState());
				}
				*/
				break;
			}
			//Expands Node if solution not found
			nodesExpanded++;
			successors = Add_child_nodes(current);

			//Add successors to the queue
			for(Node child : successors)
			{
				if(child != null) {
					queue.add(child);
				}
			}
		}
	}

	//Iterative Deepening Search

	public void IDS(Node root, State goal) {
		//Initially set maxDepth to 0
		int maxDepth = 0, nodesExpanded = 0;
		// stack to keep track of frontier
		Stack<Node> stack = new Stack<>();

		// root is nothing but the initial or start state
		System.out.println("Start Iterative Deepening Search with initial 
state:\n" + root.getState());
		stack.add(root);
		while(!stack.isEmpty()) {
			ArrayList<Node> successors = new ArrayList<>();
			//Pop from top of the stack
			Node current = stack.pop();
			//Checks if the popped Node is a solution
			if(Goal_test(current.getState(), goal)) {
```

```
                            ArrayList<Node> steps = current.sequence(current);
                            System.out.println("Finished Iterative Deepening Search 
with depth - " + current.getCost() + " and nodes expanded - " + nodesExpanded + 
"\n" + current.getState() + "\nSteps:");
                            /*
                            for(Node step : steps) {
                                  System.out.println(step.getState());
                            }
                            */
                            break;
                     //Expands Node if the depth of the Node is less than the 
maxDepth
                     } else if(current.getLevel() < maxDepth)
                     {
                            nodesExpanded++;
                            successors = Add_child_nodes(current);
                            for(Node child : successors) {
                                  if(child != null) {
                                         stack.add(child);
                                  }
                            }
                     }
                     //If stack size = 0, meaning that there is no solution for the 
current maxDepth, increase value of maxDepth by 1
                     //and adds the root to the stack so the search can be started 
again with a higher maximum depth
                     if(stack.size() == 0)
                     {
                            stack.push(root);
                            maxDepth++;
                     }

              }
       }


       // A* Search

       public void aStar(Node root, State goal)
       {
              int nodesExpanded = 0;
              // PriorityQueue to keep track of frontier, and select cheapest node 
to expand first
              PriorityQueue<Node> pri_queue = new PriorityQueue<>();

              // root is nothing but the initial or start state
              System.out.println("Start A* Search with initial state:\n" + 
root.getState());
              pri_queue.add(root);
              while(!pri_queue.isEmpty())
              {
                     ArrayList<Node> successors = new ArrayList<>();
                     //Remove queue head
                     Node current = pri_queue.poll();
                     //Check if head of the queue is the solution
                     if(Goal_test(current.getState(), goal))
                     {
                            ArrayList<Node> steps = current.sequence(current);
```

```
                        System.out.println("Finished A* Search with depth - " +
current.getCost() + " and nodes expanded - " + nodesExpanded + "\n" +
current.getState() + "\nSteps:");
                        /*
                        for(Node step : steps) {
                              System.out.println(step.getState());
                        }
                        */
                        break;
                  }
                  //Expands Node if solution not found
                  nodesExpanded++;
                  successors = Add_child_nodes(current);

                  //Add successors to the queue
                  for(Node child : successors)
                  {
                        if(child != null)
                        {
                              child.calculateCostUsingHeuristic(goal);
                              pri_queue.add(child);
                        }
                  }
            }
      }

      // Perform goal test- to see if desired end state is reached

      private boolean Goal_test(State current, State goal) {
            if(Arrays.equals(current.getPositionA(), goal.getPositionA()) &&
Arrays.equals(current.getPositionB(), goal.getPositionB()) &&
Arrays.equals(current.getPositionC(), goal.getPositionC()))
            {
                  return true;
            } else
            {
                  return false;
            }
      }

      public ArrayList<Node> Add_child_nodes(Node current_node)
      {
            ArrayList<Node> child_nodes = new ArrayList<>();
            Move m = new Move();
            child_nodes.add(m.moveUp(current_node));
            child_nodes.add(m.moveDown(current_node));
            child_nodes.add(m.moveLeft(current_node));
            child_nodes.add(m.moveRight(current_node));
            return child_nodes;
      }
}
```

Node Class:

```
import java.util.ArrayList;

import java.util.Collections;
```

```
// Node class implements the basic structure of a node in the tree which contains the configuration
(state), the parent, the level the node is on and the cost to get to the Node

public class Node implements Comparable<Node> {
        private int level, cost;
        private State state;
        private Node parent = null;

// Creating initial state or root node of a tree. Parent will be null , node level to 0 and the cost to 0

        public Node(State state) {
                this.state = state;
                this.parent = null;
                this.level = 0;
                this.cost = 0;
        }

        // Creating a Node with a state, parent and level

        public Node(State state, Node parent, int level, int cost) {
                this.state = state;   //state in the puzzle
                this.parent = parent; //Parent of the Node
                this.level = level; //Level of the Node
                this.cost = level; //Cost to get the Node
        }

        // Current level of the Node

        public int getLevel() {
                return level;
        }
```

```
	// Cost of the Node

	public int getCost() {
		return cost;
	}

	// State of the puzzle in the Node

	public State getState() {
		return state;
	}

// Calculates the cost using the Manhattan Distance heuristic between the state of the Node and the
final State and the current cost of the Node

	public void calculateCostUsingHeuristic(State finalState) {
		int distanceFromGoalA = Math.abs(state.getPositionA()[0] -
finalState.getPositionA()[0]) + Math.abs(state.getPositionA()[1] - finalState.getPositionA()[1]);
		int distanceFromGoalB = Math.abs(state.getPositionB()[0] -
finalState.getPositionB()[0]) + Math.abs(state.getPositionB()[1] - finalState.getPositionB()[1]);
		int distanceFromGoalC = Math.abs(state.getPositionC()[0] -
finalState.getPositionC()[0]) + Math.abs(state.getPositionC()[1] - finalState.getPositionC()[1]);
		cost = cost + distanceFromGoalA + distanceFromGoalB + distanceFromGoalC;
	}


// Returns path from root to a given Node

	public ArrayList<Node> sequence(Node nodeToFindPath) {
		ArrayList<Node> sequence = new ArrayList<>();
		Node node = nodeToFindPath;
		while (node.parent != null) {
			sequence.add(node);
			node = node.parent;
		}
		Collections.reverse(sequence);
		return sequence;
	}

//Compares the cost of 2 nodes (used by priority queue for A*)

	@Override
	public int compareTo(Node n) {
		int cost = this.cost - n.getCost();
		//Dealing with nodes that have the same heuristic
		if(cost == 0) {
			if(this.getLevel() > n.getLevel()) {
				return -1;
			} else if (this.getLevel() == n.getLevel()){
				return 0;
			} else {
				return 1;
			}
		} else {
			return cost;
		}
	}
}
```

Move Class:

```
import java.util.Arrays;
// Class to do the basic movement of up, down, left and right for any given node
public class Move {
	private Node doMovement(Node n, int[] newAgentPosition) {
		Node x = null;
		if(Arrays.equals(n.getState().getPositionA(), newAgentPosition)) {
			State s = new State(n.getState().getGridSize(),
n.getState().getPositionAgent(), n.getState().getPositionB(),
n.getState().getPositionC(), newAgentPosition, n.getState().getBlockedTiles());
			x = new Node(s, n, n.getLevel() + 1, n.getCost() + 1);
		} else if(Arrays.equals(n.getState().getPositionB(),
newAgentPosition)) {
			State s = new State(n.getState().getGridSize(),
n.getState().getPositionA(), n.getState().getPositionAgent(),
n.getState().getPositionC(), newAgentPosition, n.getState().getBlockedTiles());
			x = new Node(s, n, n.getLevel() + 1, n.getCost() + 1);
		} else if(Arrays.equals(n.getState().getPositionC(),
newAgentPosition)) {
			State s = new State(n.getState().getGridSize(),
n.getState().getPositionA(), n.getState().getPositionB(),
n.getState().getPositionAgent(), newAgentPosition,
n.getState().getBlockedTiles());
			x = new Node(s, n, n.getLevel() + 1, n.getCost() + 1);
		} else {
			State s = new State(n.getState().getGridSize(),
n.getState().getPositionA(), n.getState().getPositionB(),
n.getState().getPositionC(), newAgentPosition, n.getState().getBlockedTiles());
			x = new Node(s, n, n.getLevel() + 1, n.getCost() + 1);
		}
		return x;
	}


	// Move Left. Current state is fed to the method,upon which the movement
needs to be done
	// -1 is to simulate leftward movement. New Node returned as result of the
movement

	public Node moveLeft(Node n) {
		if(n.getState().getPositionAgent()[0] - 1 >= 1 &&
!checkInterference(n.getState().getPositionAgent()[0] - 1,
n.getState().getPositionAgent()[1], n.getState().getBlockedTiles())) {
			return doMovement(n, new int[]
{n.getState().getPositionAgent()[0] - 1, n.getState().getPositionAgent()[1]});
		} else {
			return null;
		}
	}


	// Move Down. Current state is fed to the method,upon which the movement
needs to be done
	// +1 is to simulate leftward movement. New Node returned as result of the
movement

	public Node moveDown(Node n) {
		if(n.getState().getPositionAgent()[1] + 1 <=
n.getState().getGridSize()[1] &&
!checkInterference(n.getState().getPositionAgent()[0],
n.getState().getPositionAgent()[1] + 1, n.getState().getBlockedTiles())) {
			return doMovement(n, new int[]
{n.getState().getPositionAgent()[0], n.getState().getPositionAgent()[1] + 1});
		} else {
			return null;
		}
	}


	// Move Right. Current state is fed to the method,upon which the movement
needs to be done
	// +1 is to simulate leftward movement. New Node returned as result of the
movement

	public Node moveRight(Node n) {
		if(n.getState().getPositionAgent()[0] + 1 <=
n.getState().getGridSize()[0] &&
!checkInterference(n.getState().getPositionAgent()[0] + 1,
n.getState().getPositionAgent()[1], n.getState().getBlockedTiles())) {
			return doMovement(n, new int[]
{n.getState().getPositionAgent()[0] + 1, n.getState().getPositionAgent()[1]});
		} else {
			return null;
		}
	}

// Movement of blocks

	public Node moveUp(Node n) {
		if(n.getState().getPositionAgent()[1] - 1 >= 1 &&
!checkInterference(n.getState().getPositionAgent()[0],
n.getState().getPositionAgent()[1] - 1, n.getState().getBlockedTiles())) {
			return doMovement(n, new int[]
{n.getState().getPositionAgent()[0], n.getState().getPositionAgent()[1] - 1});
		} else {
			return null;
		}
	}

//Checks if a coordinate interferes with the blocked tiles

	private boolean checkInterference(int x, int y, int[][] unreachableCoords)
{
		boolean flag= false;
		if(unreachableCoords != null) {
			for(int[] coord : unreachableCoords) {
				if(x == coord[0] && y == coord[1]) {
					flag = true;
					break;
				}
			}
		}
		return flag;
	}
}
```

State Class:

```
// State class created to implement the basic structure of a State of the Puzzle
```

```
public class State {
	private int[] gridSize, positionA, positionB, positionC, positionAgent;
	private int[][] blockedTiles;

	// Create a State without blocked tiles

	public State(int[] gridSize, int[] positionA, int[] positionB, int[]
positionC, int[] positionAgent) {
		this.gridSize = gridSize;
		this.positionA = positionA;
		this.positionB = positionB;
		this.positionC = positionC;
		this.positionAgent = positionAgent;
	}

	// Create a State without blocked tiles

	public State(int[] gridSize, int[] positionA, int[] positionB, int[]
positionC, int[] positionAgent, int[]...blockedTiles) {
		this.gridSize = gridSize;
		this.positionA = positionA;
		this.positionB = positionB;
		this.positionC = positionC;
		this.positionAgent = positionAgent;
		this.blockedTiles = blockedTiles;
	}


	public int[] getGridSize() {
		return gridSize;
	}


	public int[] getPositionA() {
		return positionA;
	}


	public int[] getPositionB() {
		return positionB;
	}


	public int[] getPositionC() {
		return positionC;
	}


	public int[] getPositionAgent() {
		return positionAgent;
	}


	public int[][] getBlockedTiles() {
		return blockedTiles;
	}

// Code to print node in a grid
```

```
        public String toString() {
               String output = "";

          for (int y = 1; y <= gridSize[0]; y++) { //columns =4
              for (int x = 1; x <= gridSize[1]; x++) { //rows=4
                  if(x == positionA[0] && y == positionA[1]) {
                      output += "[A] ";
                  } else if(x == positionB[0] && y == positionB[1]) {
                      output += "[B] ";
                  } else if(x == positionC[0] && y == positionC[1]) {
                      output += "[C] ";
                  } else if(x == positionAgent[0] && y == positionAgent[1]) {
                      output += "[@] ";
                  } else {
                      boolean flag = false;
                      if(blockedTiles != null) {
                              for(int[] coord : blockedTiles) {
                                      if(coord[0] == x && coord[1] == y) {
                                              output += "[-] ";
                                              flag = true;
                                              break;
                                      }
                              }
                      }
                      if(!flag) {
                              output += "[ ] ";
                      }
                  }
              }
              output += "\n";
          }
          return output;
        }
}
```

Game Simulation: Tic-Tac-Toe

Aneeshaa S Chowdhry
University of Southampton
ID: 30439485
asc1n18@soton.ac.uk

ABSTRACT

In this paper, we describe the process of creating a computer program that can play Tic-Tac-Toe, as well as, or better than humans. One means of achieving this is to use an Adversarial search algorithm: Minmax, to find the best possible moves a computer can make to win against its opponent. We describe the program flow, and algorithm in subsequent sections.

1. INTRODUCTION

For humans, games are a means of recreation. However, for AI researchers trying to simulate game-playing with computer programs, games are non-trivial problems. Games often have multiple players, taking actions alternatively. In search problems with no adversary, such as: planning, scheduling activities, the solution is the method of finding the goal state; heuristics can find an optimal solution.

On the other hand, for game-playing, we need to take into account an adversary. Here solution is a strategy that specifies best possible move for every move of opponent. Also, we evaluate a state, based on the goodness of the position on game board.

Let us attempt to simulate a simple, and familiar game: Tic-Tac-Toe. This is an ideal starting point for novice AI programmers, as it has few board positions, and finishes in minimal time.

As simple as the game seems, it requires a player to lookahead an opponent's move. Anticipating opponent's move, after our current move, will ensure we don't lose. We can think of the game as tree with the root node as the initial state of the board. The successors will be the possible moves/states of the board. Each time the opponent makes a move, the tree is expanded to cover all possibilities of the outcome of the game. Once the tree is constructed, we score each possible move, and select the best one that maximizes our chance of winning.

2. APPROACH

To achieve simulation of intelligent Tic-Tac-Toe game play, we use a decision making algorithm called Minimax. This algorithm maximizes player's chances of winning , and minimizes opponent's chances of winning.

Rules to implement

First, we will attempt to highlight some basic rules we think of while playing other humans:

1. Search for a possible winning move, and make the move
2. Block opponent's possible winning move

Additionally, since we aim to create an unbeatable program that at the most, ends up in a draw against an opponent:

3. Try to make a move that creates a fork(two ways of winning)
4. Block opponent from creating a fork

Initial state

We represent board positions by numbers between 1-9: as shown:

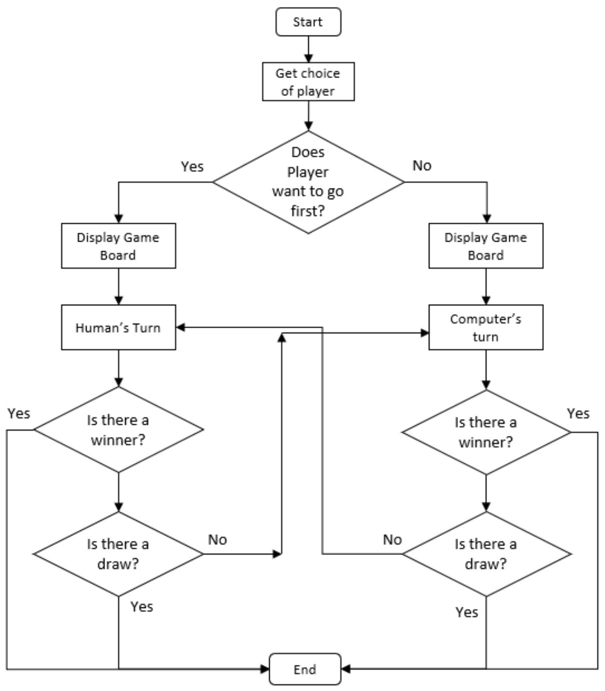
1	2	3
4	5	6
7	8	9

The human player can make his move by entering a desired board position. The initial state would be a board game with empty values in all 9 cells.

```
---------------
|   ||   ||   |
---------------
|   ||   ||   |
---------------
|   ||   ||   |
---------------
```

The Game Loop

The program flow, and logic is simplified for easier visualization and understanding in the below flow chart:



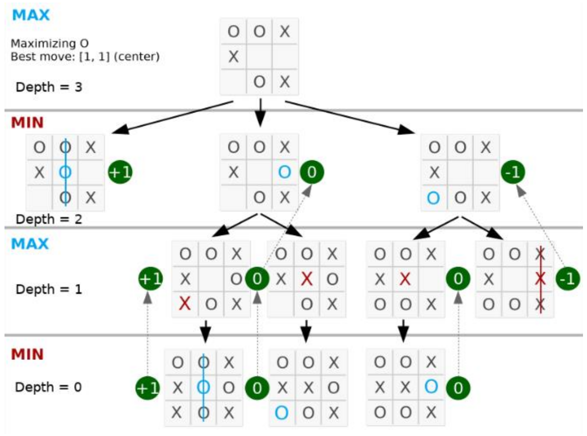
1. ***First to Play?*** Get user's input, on whether he/she would like to play first, and their choice between naught and cross. Computer's choice would be the opposite.

2. ***Human Turn:*** We prompt the human player to make a move. Let us assume the Human player chooses to go first, the Game board is displayed, and player makes a move by entering a number between 1 to 9, indicating the position of this move.
3. ***Evaluate a board state***: If the computer wins, the score of the board state is +1, if the human wins, the score of the board state is -1, else if there is a draw, score is 0. To check if there is a winner, we search for [X X X] or [O O O] in 3 rows, 3 columns, and two diagonals. If we indeed have a winner, or a draw, the Game ends.
4. ***The computer's turn:*** If the computer is making the first move, i.e. if the board is currently empty, pick a random position to make a move.
   Otherwise, we call the minimax algorithm recursively to expand the tree, and then evaluate the board state at the leaf node, to decide if a particular move will result in a win. This is run repeatedly for as long as there are empty cells/possible positions for the computer/human to make a move alternatively.
5. ***Minimax Algorithm***: The Minimax algorithm used in our program is simplified for explanation here. Inputs to the program are player(+1 for computer, -1 for human), board or state of the game board, and depth is the index of the node in the Game tree. Based on the score returned, the computer makes a move on the board.

```
minimax(player, board, depth)

    if(game over in current board position)
        return winner
    children = all legal moves for player from this board
    if(computer's turn)
        return maximal score of calling minimax on all the children
    else (human's turn)
        return minimal score of calling minimax on all the children
```

If there is not already a winner, minimax will go through each possible child, and (by recursively calling itself) evaluate each possible move. Then, the best possible move will be chosen, where best is the move leading to the board with the most positive score for the computer, and the board with the most negative score for human player.

The minimax method in our code is modified to return best possible position(x, y co-ordinates on the Game board), and score. An illustration of the process[1] is shown below.



6. Exception handling has been included in various parts of code to ensure smoother operation. For example, when the user enters board position to make a move, we check for its validity. i.e. if he has entered position of an empty cell, or the range of number(all positions are between 0 and 9).

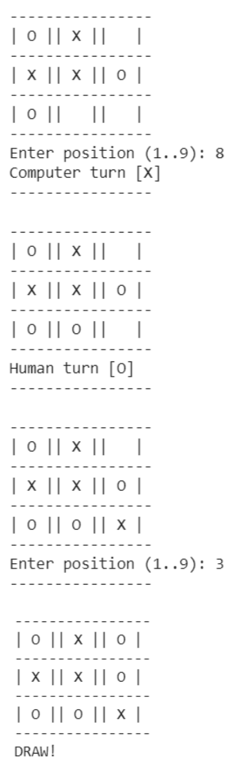
3. EVIDENCE

In order to test our program's intelligence, we play against it a number of times. It is observed that, the games mostly end in a draw, or a win for the computer. Results of one such Game is shown below.

The User, on this instance, chooses to play a naught, and chooses to go first. Consequently, the computer marks the board with crosses.

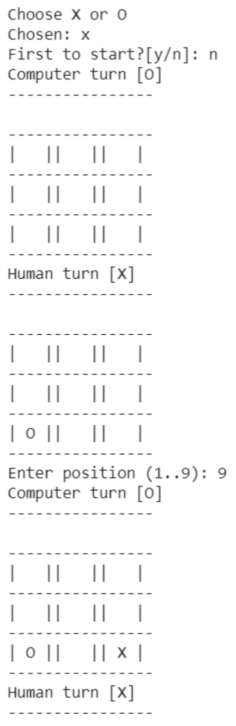
Case 1: Evidence of game that ends in a draw.

```
Choose X or O
Chosen: O
First to start?[y/n]: Y
Human turn [O]
---------------

---------------
|   ||   ||   |
---------------
|   ||   ||   |
---------------
|   ||   ||   |
---------------
Enter position (1..9): 1
Computer turn [X]
---------------

---------------
| O ||   ||   |
---------------
|   ||   ||   |
---------------
|   ||   ||   |
---------------
Human turn [O]
---------------

---------------
| O ||   ||   |
---------------
|   || X ||   |
---------------
|   ||   ||   |
---------------
Enter position (1..9): 7

Computer turn [X]
---------------

---------------
| O ||   ||   |
---------------
|   || X ||   |
---------------
| O ||   ||   |
---------------
Human turn [O]
---------------

---------------
| O ||   ||   |
---------------
| X || X ||   |
---------------
| O ||   ||   |
---------------
Enter position (1..9): 6
Computer turn [X]
---------------

---------------
| O ||   ||   |
---------------
| X || X || O |
---------------
| O ||   ||   |
---------------
Human turn [O]
---------------
```

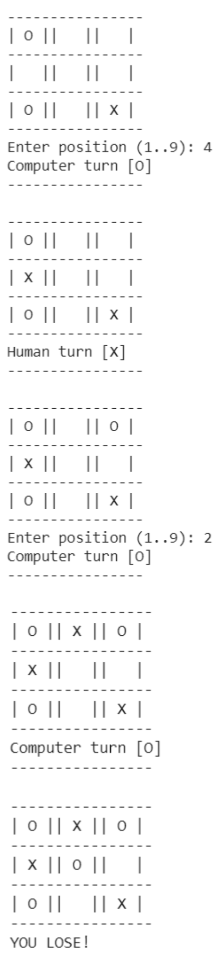


```
----------------
| O || X ||   |
----------------
| X || X || O |
----------------
| O ||   ||   |
----------------
Enter position (1..9): 8
Computer turn [X]
----------------

----------------
| O || X ||   |
----------------
| X || X || O |
----------------
| O || O ||   |
----------------
Human turn [O]
----------------

----------------
| O || X ||   |
----------------
| X || X || O |
----------------
| O || O || X |
----------------
Enter position (1..9): 3
----------------

----------------
| O || X || O |
----------------
| X || X || O |
----------------
| O || O || X |
----------------
DRAW!
```

Case 2: Evidence of game, where human player loses



```
Choose X or O
Chosen: x
First to start?[y/n]: n
Computer turn [O]
----------------

----------------
|   ||   ||   |
----------------
|   ||   ||   |
----------------
|   ||   ||   |
----------------
Human turn [X]
----------------

----------------
|   ||   ||   |
----------------
|   ||   ||   |
----------------
| O ||   ||   |
----------------
Enter position (1..9): 9
Computer turn [O]
----------------

----------------
|   ||   ||   |
----------------
|   ||   ||   |
----------------
| O ||   || X |
----------------
Human turn [X]
----------------
```



```
----------------
| O ||   ||   |
----------------
|   ||   ||   |
----------------
| O ||   || X |
----------------
Enter position (1..9): 4
Computer turn [O]
----------------

----------------
| O ||   ||   |
----------------
| X ||   ||   |
----------------
| O ||   || X |
----------------
Human turn [X]
----------------

----------------
| O ||   || O |
----------------
| X ||   ||   |
----------------
| O ||   || X |
----------------
Enter position (1..9): 2
Computer turn [O]
----------------

----------------
| O || X || O |
----------------
| X ||   ||   |
----------------
| O ||   || X |
----------------
Computer turn [O]
----------------

----------------
| O || X || O |
----------------
| X || O ||   |
----------------
| O ||   || X |
----------------
YOU LOSE!
```

4. SCALABILITY

The Minimax algorithm performs complete depth-first exploration of the game tree. This means, the Time complexity and Space complexity of this algorithm resemble Depth-First search algorithm.

Hence, the time elapsed or number of nodes generated increases exponentially, with depth: $O(b^d)$. However, the space consumption varies linearly: $O(bd)$. For Tic-Tac-Toe, there are 3^9 = 19,683 possible states.

Therefore, Minimax algorithm sans alpha-beta pruning does not scale well for games with larger boards/search spaces. For Chess, the branching factor is 35, and the number of moves by both players is 100. So, the search tree is as large as approx.. 10^{154}. Unless we have a mechanism to prune search tree, this would be impractical to implement.

In conclusion, Minimax algorithm works well for smaller games such as Nim, Tic-Tac-Toe, Five-in-a-row, etc.

5. LIMITATIONS

As mentioned in the Scalability section, the game has 19,683 possible states. Since, the code written doesn't include provision to prune the search tree, the algorithm is inefficient, and incapable of scaling for larger problems.

Also, the program doesn’t include a strategy or heuristic to give preference to forking positions. The program has been written in python in order to gain exposure to the language.

6. REFERENCES

[1] George T. Heineman; Gary Pollice; Stanley Selkow. Algorithms in a nutshell. O'Reilly, 2009.

[2] Brendan O’Donoghue, B., "Min-Max Approximate Dynamic Programming", IEEE Multi-Conference on Systems and Control, pp. 28-30, 2011.

[3] Rivest, R.L, "Game Tree Searching by Min / Max Approximation", Artificial Intelligence 12(1), pp. 77-96, 1988.

[4] Chen, J.; Wu, I.; Tseng, W.; Lin, B.; and Chang, C. “Job-level alpha-beta search”, IEEE Transactions on Computational Intelligence and AI in Games, 2014.

[5] S. Gal, “A discrete search game”, SIAM Journal of Applied Mathematics 27(4), pp. 641-648, 1974

[6] https://www.ntu.edu.sg/home/ehchua/programming/java/JavaGame_TicTacToe_AI.html

[7] https://en.wikipedia.org/wiki/Minimax

7. APPENDIX

Program has been written by taking inspiration from the sources above. Parts downloaded have not been included below; parts such as accepting user input, game board display, check for winner, and ideas for exception handling.

```
#!/usr/bin/env python3
from math import inf as infinity
from random import choice
import platform
import time
from os import system

HUMAN = -1
AI = +1
board = [
    [0, 0, 0],
    [0, 0, 0],
    [0, 0, 0],
]

# Main function that calls all functions
def main():
    clear_console() # function written to clear console
    human_choice = '' # X or O
    ai_choice = '' # X or O
    first = ''  # if human is the first

    # Human chooses X or O to play
    while human_choice != 'O' and human_choice != 'X':
        try:
            print('')
            human_choice = input('Choose X or O\nChosen: ').upper()
        except KeyboardInterrupt:
            print('Bye')
            exit()
        except:
            print('Invalid choice')

    # Setting computer's choice
    if human_choice == 'X':
        ai_choice = 'O'
    else:
        ai_choice = 'X'

    # Human may start first
    clear_console()
    while first != 'Y' and first != 'N':
        try:
            first = input('First to start?[y/n]: ').upper()
        except KeyboardInterrupt:
            print('Bye')
            exit()
        except:
            print('Invalid choice')

    # Main loop of this game
    while len(empty_cells(board)) > 0 and not check_if_game_over(board): # while moves remain and no one has won
        if first == 'N':
            ai_turn(ai_choice, human_choice) # computer makes move if human chose to not play first
            first = ''

        human_turn(ai_choice, human_choice)
        ai_turn(ai_choice, human_choice)

    # Game over message
    if check_for_winner(board, HUMAN):
        clear_console()
        print('Human turn [{}]'.format(human_choice))
        display_board(board, ai_choice, human_choice)
        print('YOU WIN!')
    elif check_for_winner(board, AI):
        clear_console()
        print('Computer turn [{}]'.format(ai_choice))
        display_board(board, ai_choice, human_choice)
        print('YOU LOSE!')
    else:
        clear_console()
        display_board(board, ai_choice, human_choice)
        print('DRAW!')

    exit()
```

# up to human, to make a valid move

```
def human_turn(ai_choice, human_choice):

    depth = len(empty_cells(board))
    #print(depth)
    if depth == 0 or check_if_game_over(board):
        return

    # Dictionary of valid moves
    move = -1
    moves = {
        1: [0, 0], 2: [0, 1], 3: [0, 2],
        4: [1, 0], 5: [1, 1], 6: [1, 2],
        7: [2, 0], 8: [2, 1], 9: [2, 2],
    }

    clear_console()
    print('Human turn [{}]'.format(human_choice))
    display_board(board, ai_choice, human_choice)

    while (move < 1 or move > 9):
        try:
            move = int(input('Enter position (1..9): '))
            board_position = moves[move]
            try_move = set_move(board_position[0], board_position[1], HUMAN)

            if try_move == False:
                print('Invalid move')
                move = -1
        except KeyboardInterrupt:
            print('Bye')
            exit()
        except:
            print('Invalid choice')
```

# For the first move, choose random coordinate. Else call minimax function

```
def ai_turn(ai_choice, human_choice):

    depth = len(empty_cells(board))
    if depth == 0 or check_if_game_over(board): # if no more moves left or any player won, return to main
        return

    clear_console()
    print('Computer turn [{}]'.format(ai_choice))
    display_board(board, ai_choice, human_choice)

    if depth == 9:  #for ai's first move, choose any position randomly
        x = choice([0, 1, 2])
        y = choice([0, 1, 2])
    else:
        move = minimax(board, depth, AI)
        x, y = move[0], move[1]

    set_move(x, y, AI)
    time.sleep(1)
```

# recursively call minimax, to evaluate all possible moves, and #make the best move for winning position

```
def minimax(state, depth, player):
    if player == AI:
      # initialize coordinates of best move, and score of best move
        best = [-1, -1, -infinity] # -infiniy because, we need AI to choose max value
    else:
      # initialize coordinates of best move, and score of best move
        best = [-1, -1, +infinity] # +infiniy because, human will choose min value for AI

    if depth == 0 or check_if_game_over(state): # are there no empty cells remaining? or is there already a winner?
        score = calc_score(state)
        return [-1, -1, score]
    for cell in empty_cells(state): # for the remaining empty cells, check for best possible move
        x, y = cell[0], cell[1] # take first available empty cell
        state[x][y] = player   # mark as +1 if AI's turn, -1 if human's turn
        score = minimax(state, depth - 1, -player) #-player, because players turns alternate. This is where/how we expand the tree( recursively calling minimax)
        state[x][y] = 0 # wipe out changes, make cell empty again
        score[0], score[1] = x, y # get calculated score

        if player == AI:
            if score[2] > best[2]:
                best = score  # max value
        else:
            if score[2] < best[2]:
                best = score  # min value for human player

    return best
```

# Function to calculate score or evaluation of state.

```
def calc_score(state):
    if check_for_winner(state, AI):
        score = +1
    elif check_for_winner(state, HUMAN):
        score = -1
    else:
        score = 0

    return score
```