Report - 1
on
Artificial Intelligence Lab Assignments

Adit Jain 201851007
IIIT Vadodara
Computer Science and Engineering
Email: 201851007@iiitvadodara.ac.in

Deep Shah 201851037
IIIT Vadodara
Computer Science and Engineering
Email: 201851037@iiitvadodara.ac.in

Devansh Agarwal 201851038
IIIT Vadodara
Computer Science and Engineering
Email: 201851038@iiitvadodara.ac.in

Kartikay Sarswat 201851057
IIIT Vadodara
Computer Science and Engineering
Email: 201851057@iiitvadodara.ac.in

Pallavi Sharma 201851079
IIIT Vadodara
Computer Science and Engineering
Email: 201851079@iiitvadodara.ac.in

Contents

***Abstract*—We have studied and analyzed various search algorithms and implemented them for several problems of Artificial Intelligence. We have learned how AI is efficient in solving such real life problems and puzzles.**

1. Introduction

In the past, the phenomenons that made life more convenient for our ancestors were termed to be "magic" and "supernatural powers". Now? It's called technology. And the "magic spells" are now termed as the complicated algorithms that get this technology running. One such technology that has occupied a prominent place in our present lives is Artificial Intelligence.

February 26, 2021

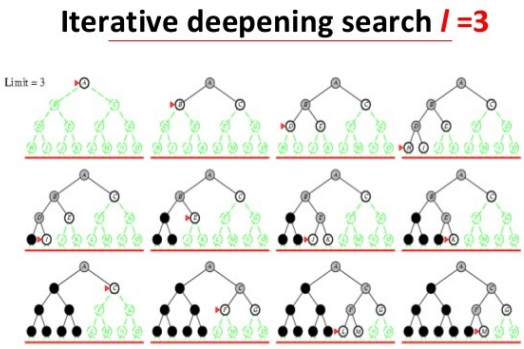
2. Assignment 1

2.1. Solution for Part A

- Step 1:- First of all we construct a tree using a dictionary of lists. Every vertex has 4 subnodes, which means every key has 4 values. Each value is an array of length 9.
- Step 2:- Once the tree is generated, we define the starting array and goal array.
- Step 3:- Then we implement the Iterative deepening search:
  - We search the highest level as we do in BFS, if we find the target node there, the search stops.
  - If we do not find the target node, we traverse towards the next level, by increasing the

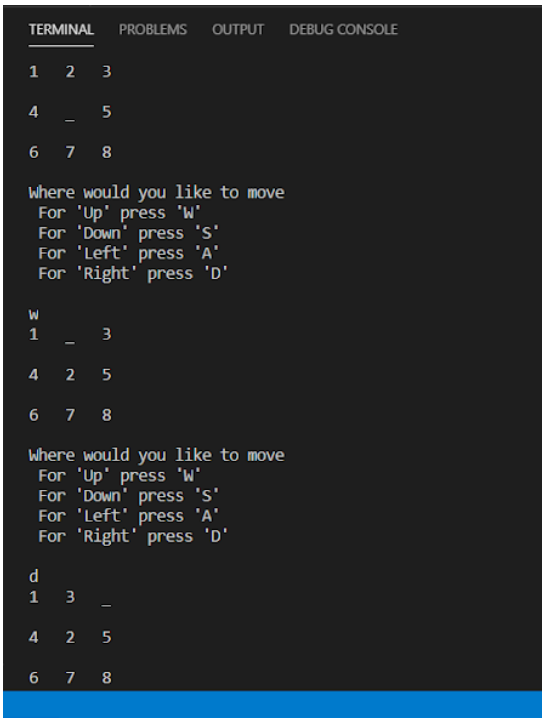
maximum depth. Now we first search the src node and then its subnodes, if we find nothing in the subnodes, then the subnodes of other nodes at previous depth are traversed.
  - The depth keeps on increasing till we find our target node.

- Advantages :- A combination of BFS and DFS which uses less memory and guarantees to find the nodes.
- Disadvantages :- We need to search nodes at the top multiple times.
- Pseudocode :
  - If target found, Return True
  - If max depth reached, Return False
  - Search recursively with maxdepth-1, with DFS till the target is achieved.
  - If the target is not found, go to the next depth.
  - If the target is found, return the source array.
  - Backtrack the array to find the path
- Flowchart and explanation :-



2.2. Solution for Part B

- The code is uploaded on the Github repo, and can be accessed by clicking here.
- It generates the environment in which we are able to operate the puzzle as desired
- We use :-
  - W :- To move the blank space upward
  - S :- To move the blank space downward
  - A :- To move the blank space left
  - D :- To move the blank space right
- The Output :-

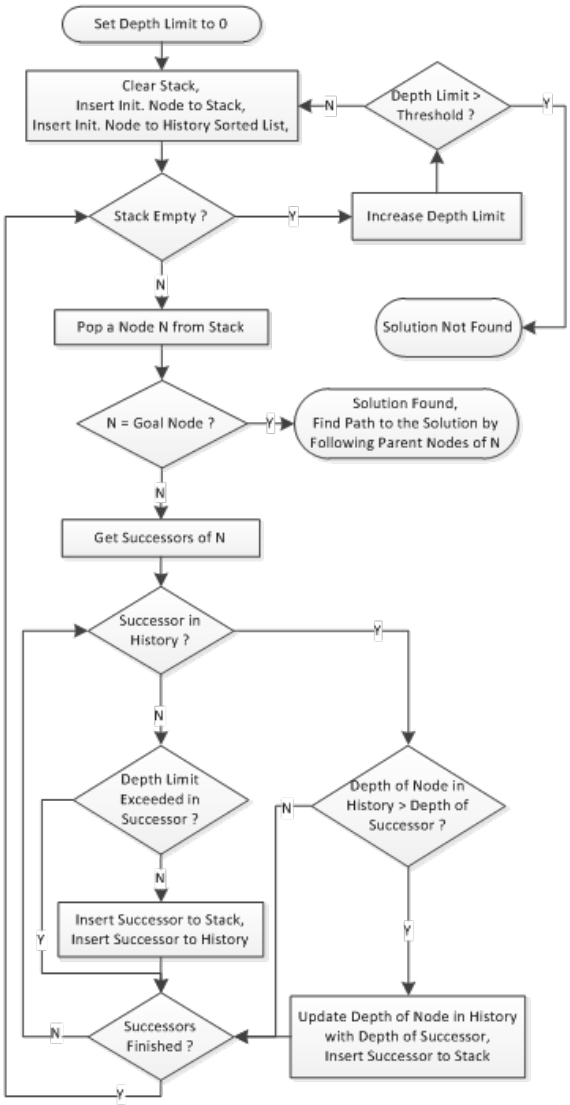


2.3. Solution for Part C

Iterative Deepening Search(IDS) is used when you have a goal directed agent in an infinite search space (or search tree). It is a hybrid of BFS and DFS. IDS combines depth-first search's space-efficiency and breadth-first search's fast search (for nodes closer to root). In this, we perform DFS up to a certain depth and keep incrementing this allowed depth. Performing DFS upto a certain allowed depth is called Depth Limited Search (DLS). So basically we do DFS in a BFS fashion.
Pseudocode:

```
IDS(tree):
    for depth = 0 to infinity:
        if (DLS(tree, depth)):
            return true
    return false
```

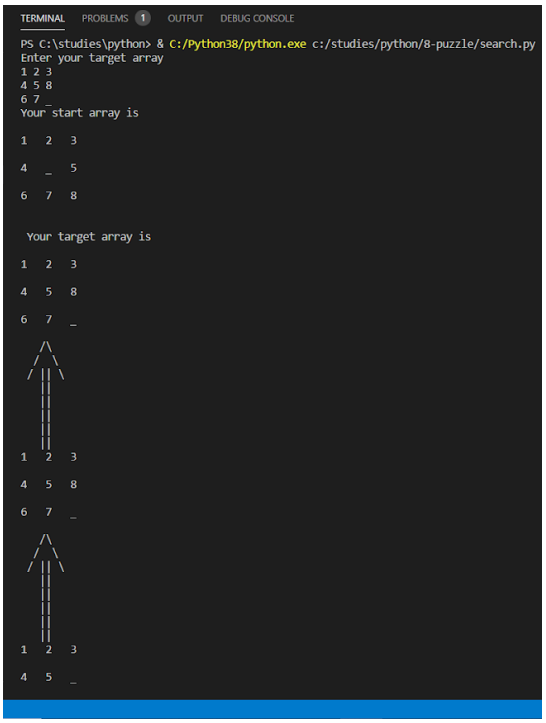
- Space complexity of BFS :- O(bd), But it ensures that we find the desired target
- Space complexity of DFS :- O(d), But it may or may not find a solution
- Therefore IDS combines both to provide a space complexity of O(bd), which is good and guarantees to find a solution.
- The time complexity of all the three are still O(bd)

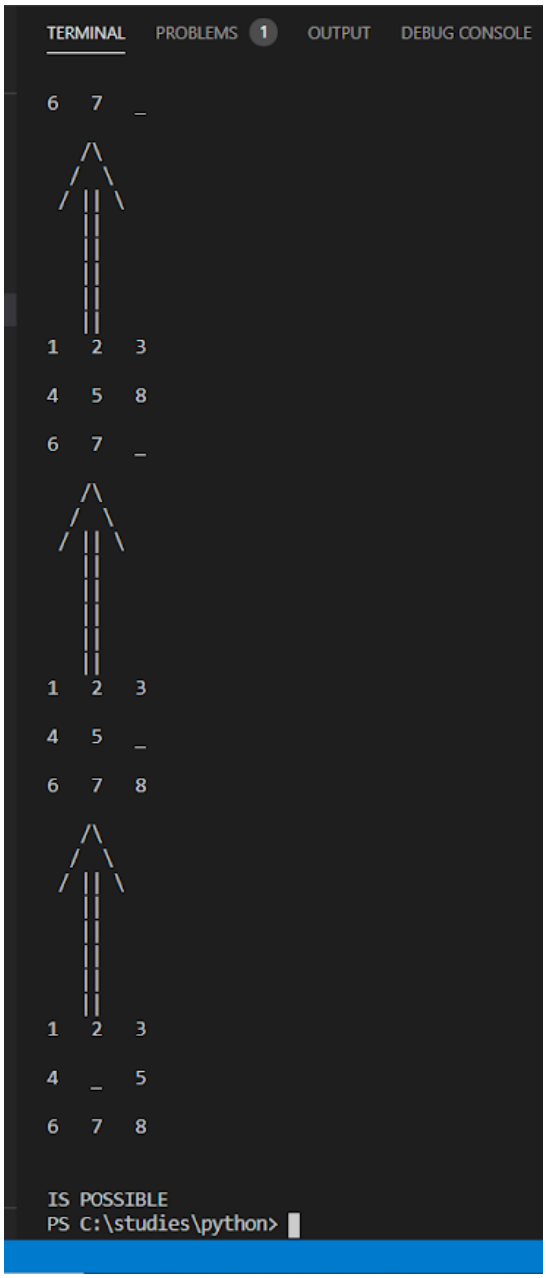


2.4. Solution for Part D

The functions applied to find and backtrack the puzzle via IDS is as follows :

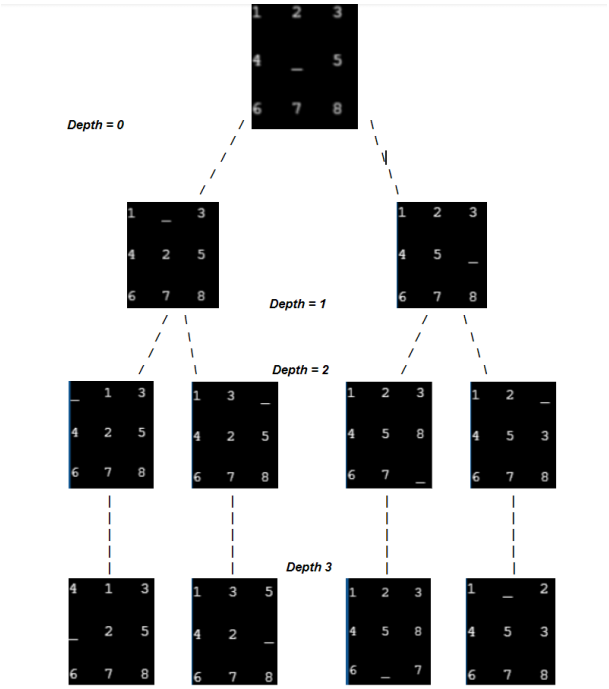
- The First Script, search.py contains the functions to design and move the 8-puzzle algorithm. Also, it takes in the goal or target and saves it as an Array.
- It constructs the complete tree for 8-puzzle -problem via creating edges. Each edge is stored as a key:value pair. To traverse through the tree, defaultdict is used.
- It initializes the search for the goal array in the array tree.
- The second script, graph.py contains the functions to find the goal and backtrack towards the starting state.
- Currently, the code is designed to operate till depth 3, which can be increased as per choice.
- The github repo can be accessed by clicking here.
- Once we run search.py, following output is obtained:-





- Hence we backtrack it towards the start state and find the path.

2.5. Solution for Part E



So on till depth "d" which can be extended to any number starting from start state until we will receive our goal state.

2.6. Solution for Part F

Here:-

b = branching factor = 4

d = depth

Memory Complexity	Time Complexity	Depth
O(bd)= O(4)	O(bd)= O(4)	1
O(bd)= O(8)	O(bd)= O(16)	2
O(bd)= O(12)	O(bd)= O(64)	3
O(bd)= O(16)	O(bd)= O(256)	4
O(bd)= O(20)	O(bd)= O(1024)	5

3. Assignment 2

3.1. Solution for Part A

The solution to the different parts of the question is as follows:-

1) The code to the priority queue can be found by clicking here.

2) One of the most famous puzzle games is marble solitaire, we have solved it via the following algorithms :

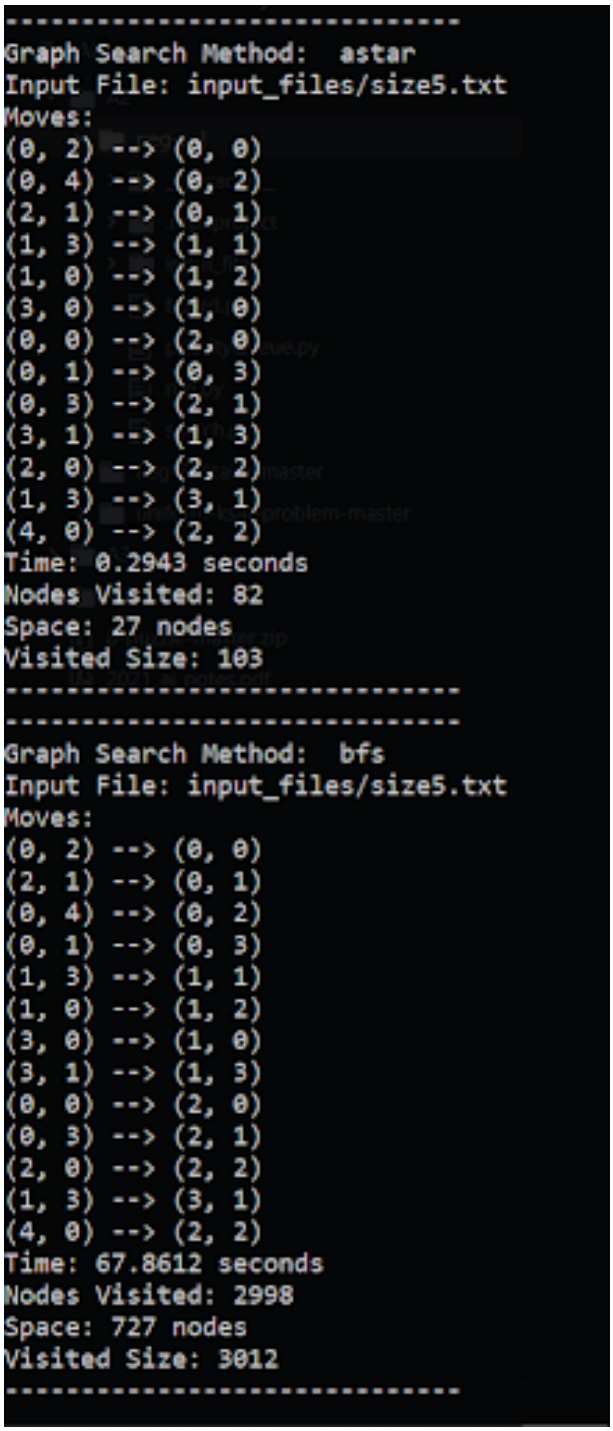
- priority queue based search considering path cost
- best first search algorithm,
- A*

Following is the code for the algorithms.
We need heuristic functions to provide a direction for our algorithms. Two different heuristic functions can be used :-

- **Distances from the center :-** Manhattan distance, Exponential distance and Corrected exponential.
  - Manhattan distance :- This heuristic value is calculated as the sum of distances of each peg from the center of the board, for example, if there are 3 pegs left far away from the center, they will have a much higher heuristic value than 7 pegs very close to the center and to each other as they can be reduced further by continuing the solution.
  - The distance of each peg is calculated and summed up to provide heuristic value of the state. The algorithms use such values to move ahead of their state.
  - The Manhattan distance can be converted to exponential distance.
  - Exponential Distance :- The procedure to find heuristic value of a state is similar, but with a larger bias towards the center. If H,V are the horizontal and vertical distances from the center respectively, then the heuristic's value is 2max(H,V).
  - The exponential Distance can further be converted to corrected exponential to provide better heuristical values for a given state.
  - Corrected Exponential :- This again is similar to the previous one, but the distances of some of the positions close to the center were altered a bit to perform better (in the original board).

- **Connected Components:-**
  - The idea is to prefer states in which no two concentrated groups of pegs are so far away from each other that they will never meet. If they are not far away, the heuristic simply gives the number of pegs minus one (this is always a lower bound, and if the state is solvable it is also tight). If they are, the state is not solvable and it returns infinity. Since in either case it returns a lower bound, it is also admissible.
  - For example, the heuristic value of a state with two pegs away from each other are far more than the heuristic of 2 pegs near each other. So the heuristic tries to minimize the distance between the pegs.
  - Although the heuristic is not as good as the corrected exponential, it still provides a good way to rank the states for the algorithms.

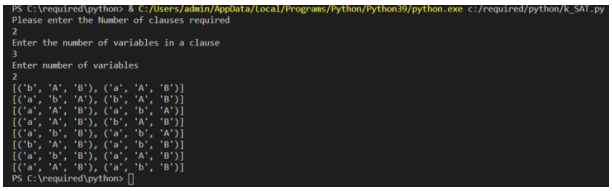
3) and 4) The Github repo can be accessed by clicking here.



5) Among various search algorithms, A* shows significantly better performance. A* search algorithm takes about 0.3 seconds to solve marble solitaire, whereas Breadth-first search takes about 65 seconds for the same, which is very high. Performance of Iterative Deepening Search is in between these two.

3.2. Solution for Part B

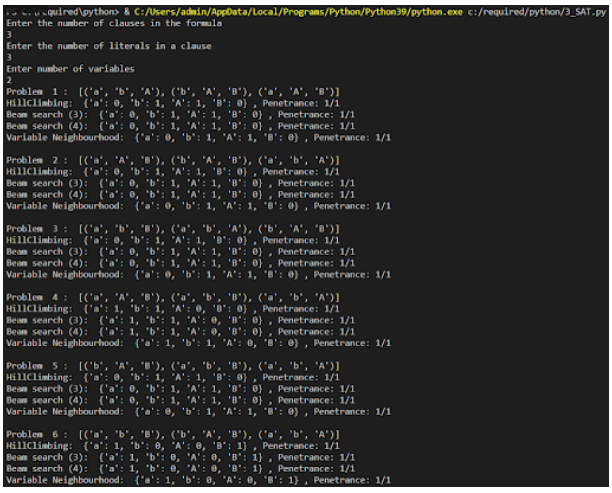
The Github repo contaning the solution code can be found here.



3.3. Solution for Part C

Github repo contaning the code can be accessed by clicking here.

The heuristic functions that can be chosen are “Number of clauses which are true” and “Random Walk”. In the first one, the heuristic is calculated by checking the number of clauses that are True. On other hand, RandomWalk algorithm first randomly selects a clause that is not satisfied with the CNF, then randomly selects a flip in the clause. We have used the prior in our implementation since it is more efficient and easier to implement.



```
Problem  7 :  [('b', 'A', 'B'), ('a', 'b', 'B'), ('a', 'A', 'B')]
HillClimbing:  {'a': 0, 'b': 0, 'A': 1, 'B': 1} , Penetrance: 1/1
Beam search (3):  {'a': 0, 'b': 0, 'A': 1, 'B': 1} , Penetrance: 1/1
Beam search (4):  {'a': 0, 'b': 0, 'A': 1, 'B': 1} , Penetrance: 1/1
Variable Neighbourhood:  {'a': 0, 'b': 0, 'A': 1, 'B': 1} , Penetrance: 1/1

Problem  8 :  [('a', 'b', 'A'), ('b', 'A', 'B'), ('a', 'b', 'B')]
HillClimbing:  {'a': 1, 'b': 1, 'A': 0, 'B': 0} , Penetrance: 1/1
Beam search (3):  {'a': 1, 'b': 1, 'A': 0, 'B': 0} , Penetrance: 1/1
Beam search (4):  {'a': 1, 'b': 1, 'A': 0, 'B': 0} , Penetrance: 1/1
Variable Neighbourhood:  {'a': 1, 'b': 1, 'A': 0, 'B': 0} , Penetrance: 1/1

Problem  9 :  [('a', 'b', 'A'), ('a', 'b', 'B'), ('a', 'A', 'B')]
HillClimbing:  {'a': 1, 'b': 1, 'A': 0, 'B': 0} , Penetrance: 1/1
Beam search (3):  {'a': 1, 'b': 1, 'A': 0, 'B': 0} , Penetrance: 1/1
Beam search (4):  {'a': 1, 'b': 1, 'A': 0, 'B': 0} , Penetrance: 1/1
Variable Neighbourhood:  {'a': 1, 'b': 1, 'A': 0, 'B': 0} , Penetrance: 1/1

Problem  10 :  [('a', 'A', 'B'), ('a', 'b', 'B'), ('b', 'A', 'B')]
HillClimbing:  {'a': 0, 'b': 1, 'A': 1, 'B': 0} , Penetrance: 1/1
Beam search (3):  {'a': 0, 'b': 1, 'A': 1, 'B': 0} , Penetrance: 1/1
Beam search (4):  {'a': 0, 'b': 1, 'A': 1, 'B': 0} , Penetrance: 1/1
Variable Neighbourhood:  {'a': 0, 'b': 1, 'A': 1, 'B': 0} , Penetrance: 1/1
```

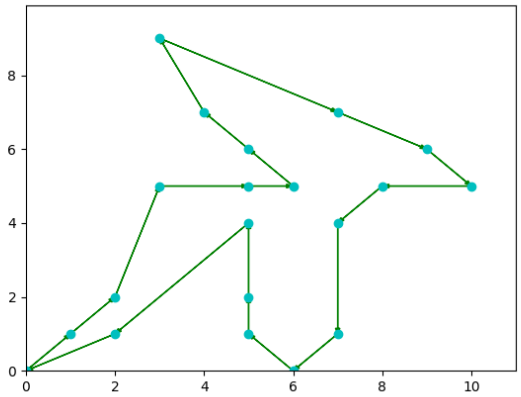
4. Assignment 3

The Github repo containing the code can be access by clicking here.

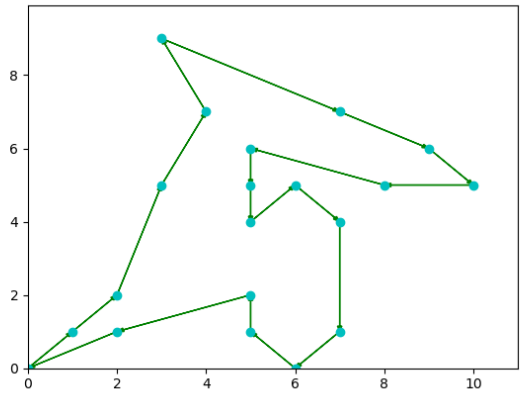
Comparison of Travel Cost:

No. of Cities	Our Code Output	Optimal Value
20(Rajasthan)	41	41
131	890	564
237	1792	1019
343	2015	1368
379	1986	1332
380	3840	1621

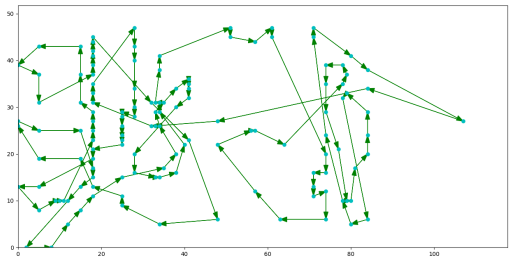
No. of Cities:- 20
Our Output:-



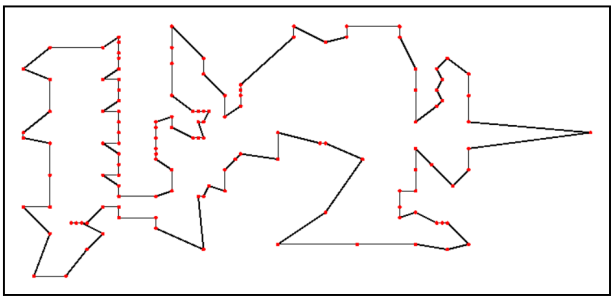
Optimal Output:-



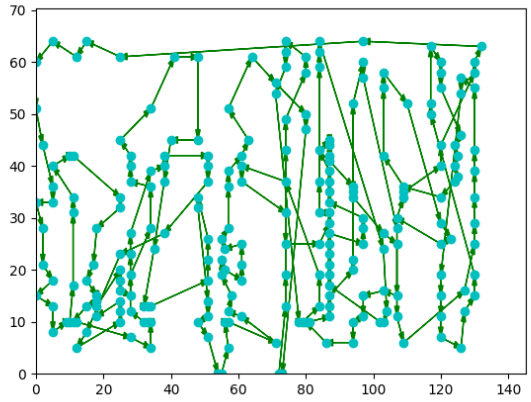
No. of Cities:- 131
Our Output:-



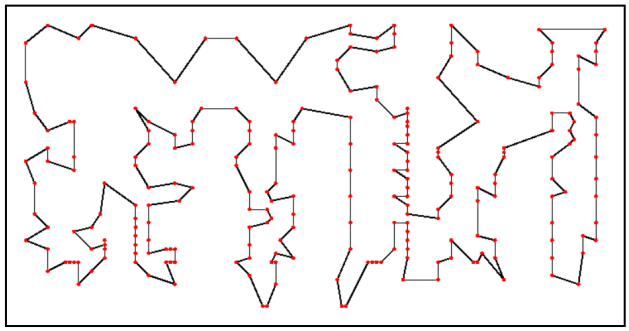
Optimal Output:-



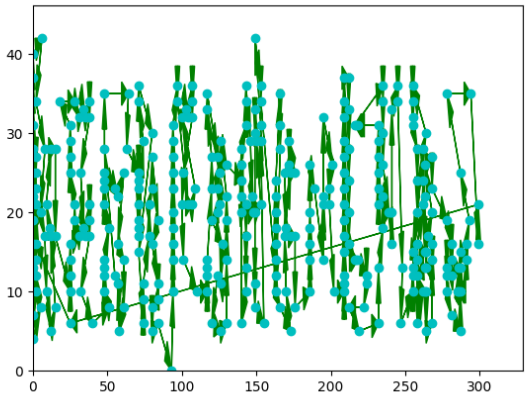
No. of Cities:- 237
Our Output:-



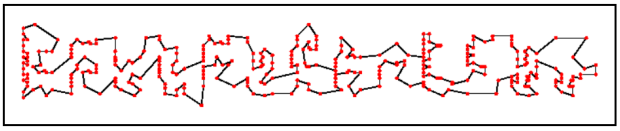
Optimal Output:-



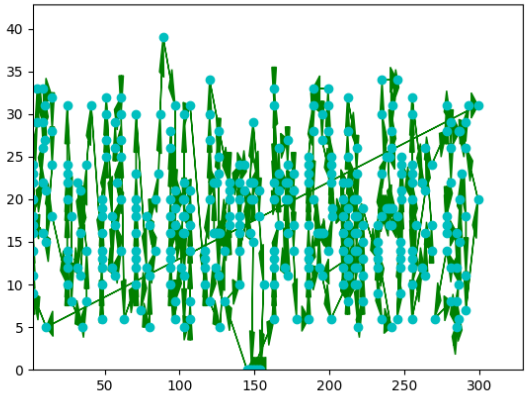
No. of Cities:- 343
Our Output:-



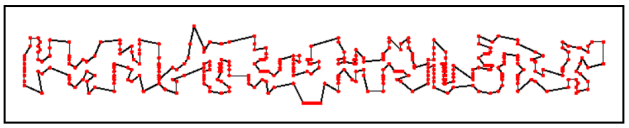
Optimal Output:-



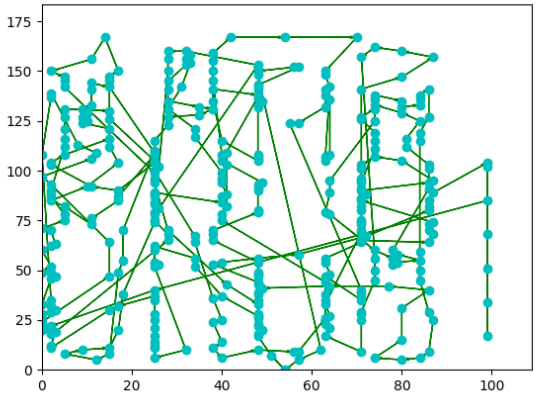
No. of Cities:- 379
Our Output:-



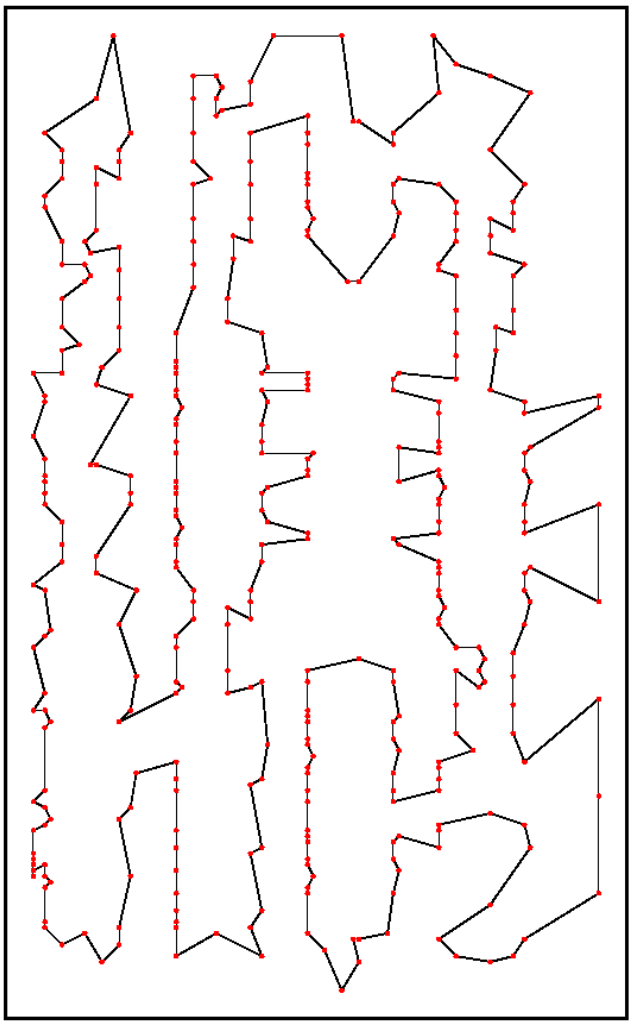
Optimal Output:-



No. of Cities:- 380
Our Output:-



Optimal Output:-



5. Assignment 4

5.1. Solution for Part A

The size of the game tree for Noughts and Crosses:-
Assuming that the game will end once all the positions have been filled meaning the players will keep on playing even after victory, its size becomes

At most 9! = 3,62,880 nodes

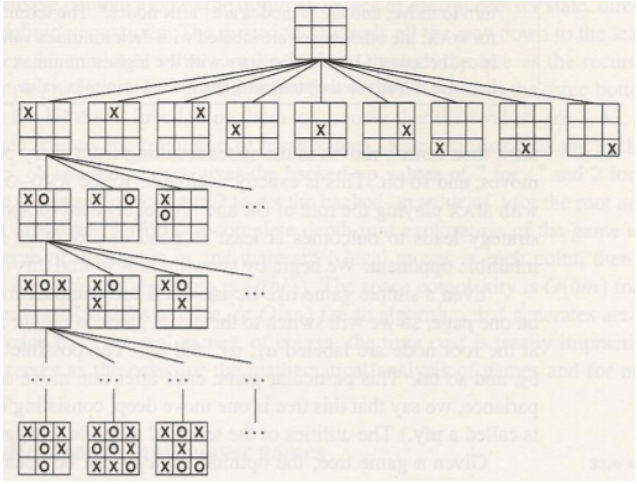
Also, if the game ends early, for example at the 5^{th} move(Best case) with 3X's and 2O's,
The size becomes:-

9*8*7*6*5 = 15,120 nodes

For the complete game tree, fewer than 3,63,000 nodes are feasible. This figure can be significantly reduced if board symmetries are taken into account, in essence that board reflection and rotation are considered the same; it makes one game the copy of another.

After removing these cases from boards, we are left with 26,830 nodes.

Now, each position can either be a Nought, a Cross or empty
Hence size: 3^9 = 19,683
This includes the cases where the board has say 4 X's and 2 O's which is an invalid board. So after removing these cases we are left with 5,478 nodes.
This also includes the symmetrical cases mentioned above. Upon removing them, there are only **765 distinct legal boards**.



5.2. Solution for Part B

The result of the game of Nim depends upon 2 factors:

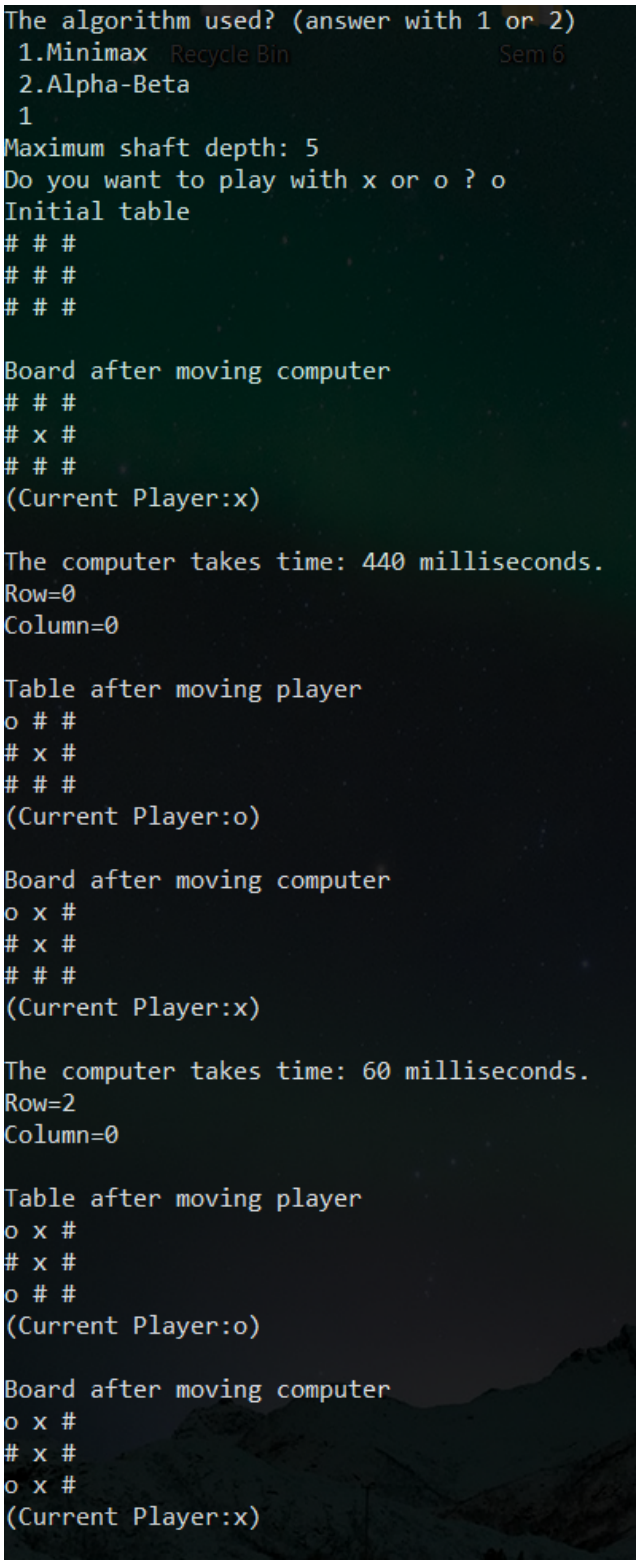
- The player who begins the game
- The initial configuration of the game

One can actually predict the winner of the game by just looking at the initial configuration of the game. This is possible by looking at the Nim-Sum at the beginning of the game. Nim Sum is the cumulative XOR of coins in each pile at any point in the game.
Nim Sum states that if the XOR value reaches 0 at any point in the game, player-1 will lose indefinitely.

5.3. Solution for Part C

The repo contaning the code to the solution can be accessed by clicking here.

Output is attached alongside this column.



5.4. Solution for Part D

Using recurrence relation under perfect ordering of leaf nodes, first consider recursive equations which give the number of states to be considered . From the equations, we will determine a rough bound on the branching factor. And next will conclude about Time complexity.

let S(m) be the minimum number of states to be considered m ply from a given state when we need to know the exact value of the state.
let R(m) be the minimum number of states to be considered m ply from a given state when we need to know a bound on the state's value.
As usual, let "b" be the branching factor and "m" as depth.

$$S(m) = S(m-1) + (b-1)R(m-1)$$

the exact value of one child and bounds on the rest.
Now,

$$R(m) = S(m-1)$$

i.e., the exact value of one child

BASE CASE: S(0) = 1 and R(0) = 1.
If we take our branching factor(b) as 3 and depth(m) as 3.
SO, we get

$$S(3) = b^2 + b - 1 => 11$$

Now, expand recursive function:

$$\begin{aligned} S(m) &= S(m-1) + (b-1)R(m-1) \\ &= (S(m-2) + (b-1)R(m-2)) + \\ &\quad (b-1)S(m-2) \\ &= bS(m-2) + (b-1)R(m-2) \\ &= bS(m-2) + (b-1)S(m-3) \end{aligned}$$

As we know, S(m-2) > S(m-3)
So, S(m) < (2b - 1) S(m - 2)

$$< 2bS(m-2)$$

The branching factor every two levels is less than 2b, which means the effective branching factor is less than $\sqrt{2b}$

we derive,

$$S(m) <= (\sqrt{2b})^m$$

$$<= (b)^{m/2}$$

So, from above we conclude that the time complexity for Alpha-beta pruning is O($b^{m/2}$).

6. Conclusion

Hence, we see that Artificial Intelligence contributes to our daily lives and solves real life problems with ease. The problems solved above are classical puzzles and real life problems, and it is shown that AI is great at solving such instances. Because of this, AI has now become a prominent part of our Life. Development in the same field would lead us to achieve great heights. Therefore,we must try to contribute in the field as much as we can.

Acknowledgment

We have taken efforts in this project, however it would not have been possible if not for Mr. Pratik Shah sir, we sincerely express our gratitude and thank you for the moral and educational support.

We would also like to acknowledge different people who have assisted in our project at various times. The internet also provided massive help for the completion of the assignment.

Last but not the least, We would like to thank our Entire team for providing great collaboration and coordination for the completion of the assignment.

References

[1] A search algorithm

[2] A* Search Algorithm

[3] Boolean Satisfiability Problem

[4] Alpha Beta Pruning

[5] Travelling Salesman

[6] Minimax

[7] Best Search BFS

[8] A distributed implementation of simulated annealing for the travelling salesman problem