T.C.
MARMARA UNIVERSITY
FACULTY of ENGINEERING
COMPUTER ENGINEERING DEPARTMENT

Artificial Intelligence (CSE4082) Project - 1

Ulaş Deniz Işık – 150118887

Barış Hazar – 150118019

Implementation Details:

GameState Class:

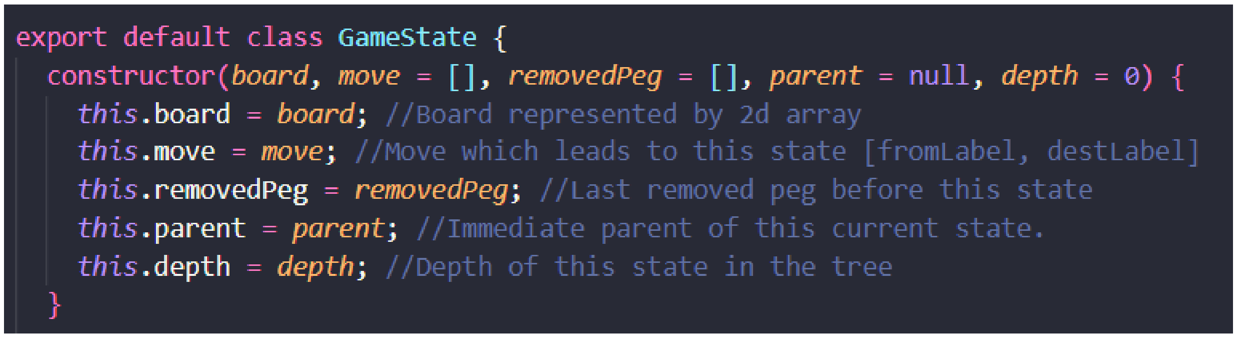
We have a GameState class which holds the current board represented by a 2-D array with 0 for empty slots, 1 for non-empty slots and -1 for invalid slots (slots at the corners).

It has a parent attribute which is a reference to another GameState instance. The reason we have parents in our data structure is that when we find a final node, we will be able to iterate all the way up to print all states until the solutions.

Also, we store the depth of each GameState variable to use in Iterative Deepening Search. (We could have also used a method which finds the depth but we preferred to use a variable instead)

Another variable is Move, it is a 2X1 array which holds the labels from and destination. For example, if our move is 14 -> 16 removing the peg 15. Then the array is [14, 16]. We use this variable for having a more user friendly output.

We also have a variable for removedPeg. It holds the slot label of the removed peg. It is necessary because we choose the least numbered removed peg in BFS DFS and Iterative Deepening Search.

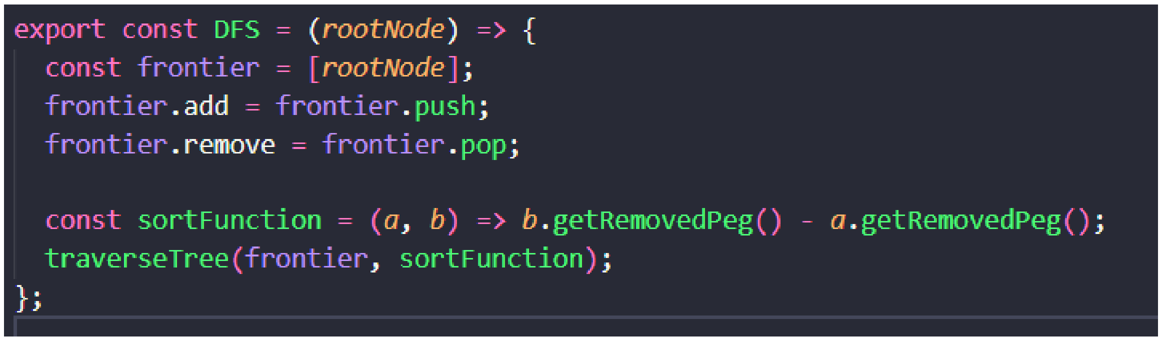


```
export default class GameState {
  constructor(board, move = [], removedPeg = [], parent = null, depth = 0) {
    this.board = board; //Board represented by 2d array
    this.move = move; //Move which leads to this state [fromLabel, destLabel]
    this.removedPeg = removedPeg; //Last removed peg before this state
    this.parent = parent; //Immediate parent of this current state.
    this.depth = depth; //Depth of this state in the tree
  }
```

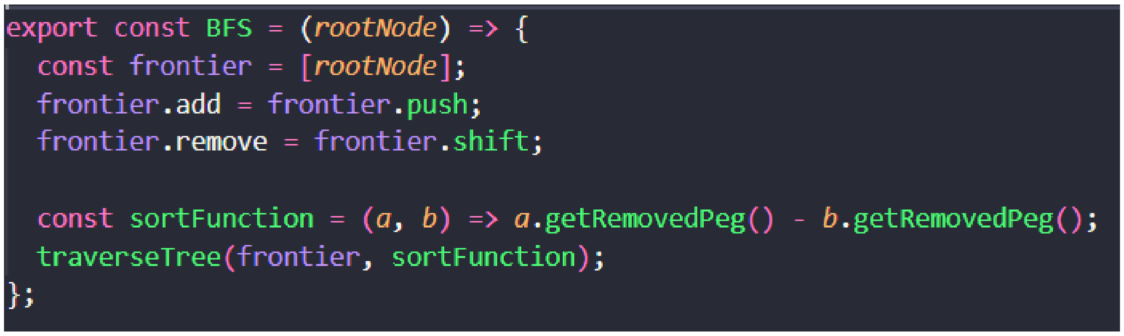
Algorithms:

We have different functions for each algorithm specified in the assignment. And each one of these functions uses an array as frontier and initializes it with the root node which is an instance of GameState given as parameter. And we assign add and remove functions to these arrays to work as queue, stack etc. For example, if we want a stack implementation, we assign add as push and remove as pop (push and pop are javascript array methods). Or similarly, if we want a stack for BFS we assign add as push(adds an item at the end) and remove as shift(removes the first node).

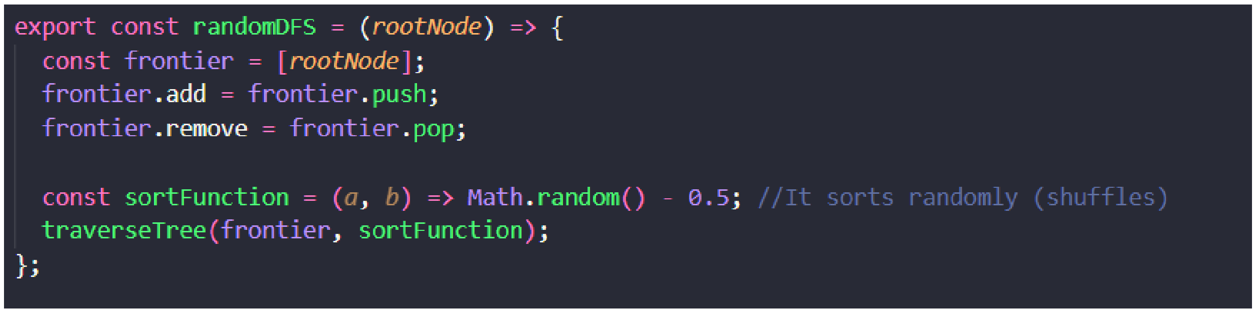
Also, we have a sorting function which defines how should the children states be sorted before added to the frontier. After we define the frontier and the sorting function we call the traverseTree function with those parameters.



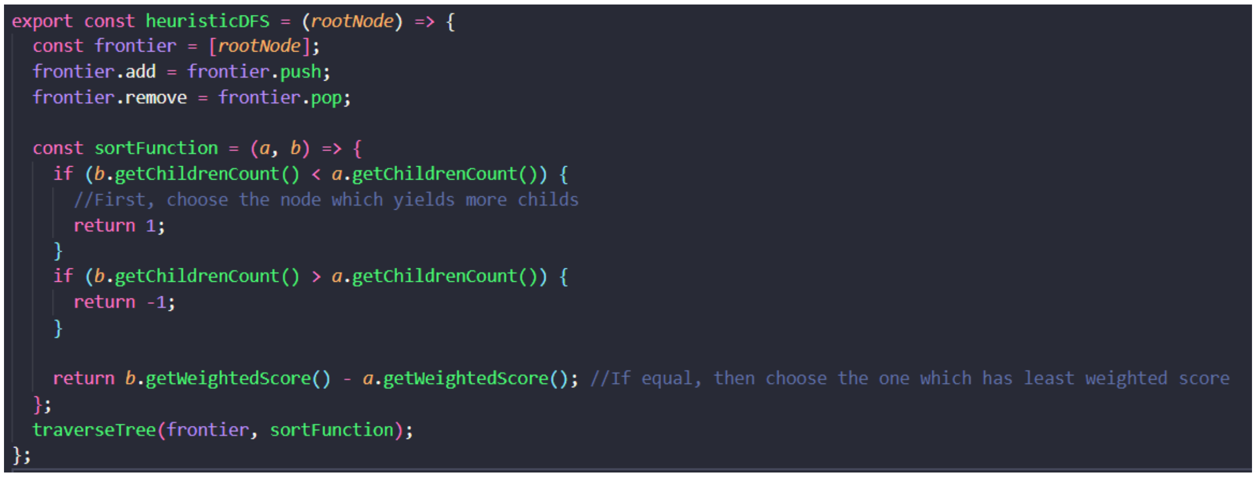
```
export const DFS = (rootNode) => {
  const frontier = [rootNode];
  frontier.add = frontier.push;
  frontier.remove = frontier.pop;

  const sortFunction = (a, b) => b.getRemovedPeg() - a.getRemovedPeg();
  traverseTree(frontier, sortFunction);
};
```



```
export const BFS = (rootNode) => {
  const frontier = [rootNode];
  frontier.add = frontier.push;
  frontier.remove = frontier.shift;

  const sortFunction = (a, b) => a.getRemovedPeg() - b.getRemovedPeg();
  traverseTree(frontier, sortFunction);
};
```



```
export const randomDFS = (rootNode) => {
  const frontier = [rootNode];
  frontier.add = frontier.push;
  frontier.remove = frontier.pop;

  const sortFunction = (a, b) => Math.random() - 0.5; //It sorts randomly (shuffles)
  traverseTree(frontier, sortFunction);
};
```



```
export const heuristicDFS = (rootNode) => {
  const frontier = [rootNode];
  frontier.add = frontier.push;
  frontier.remove = frontier.pop;

  const sortFunction = (a, b) => {
    if (b.getChildrenCount() < a.getChildrenCount()) {
      //First, choose the node which yields more childs
      return 1;
    }
    if (b.getChildrenCount() > a.getChildrenCount()) {
      return -1;
    }

    return b.getWeightedScore() - a.getWeightedScore(); //If equal, then choose the one which has least weighted score
  };
  traverseTree(frontier, sortFunction);
};
```

TraverseTree Function

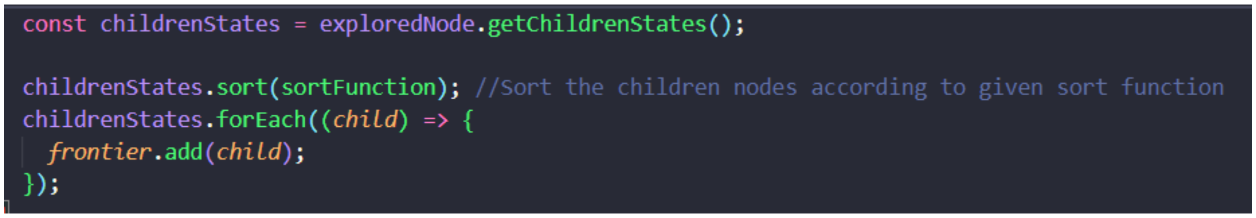
TraverseTree is the common function which all algorithms call. It takes a frontier, a sorting function (to be used for sorting children before adding to frontier). And some options to work with Iterative Deepening Search such as depth limit, isIDS etc.

It first removes the node in the frontier (the order is determined by add and remove function of the frontier as can be seen from above figures). After it removes the top node, it checks if it is a solution (it is game over). If it is and its depth is higher than the previous best solution, we update the best solution so far to the newly explored node.

After that we check if the time limit is reached or the solution is the optimal solution, if it is we exit the loop, if it is not, we continue to the normal execution.

At the end of this checking phase, we generate the children states of the newly explored node by calling its method getChildrenStates() which returns all the possible children states of the given node. These generated states are also an instance of the GameState class.

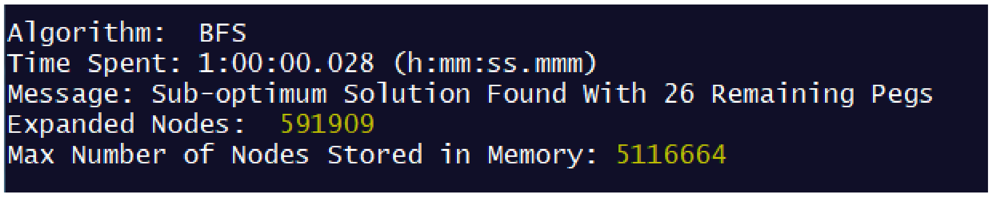
After generating the children, we sort these children with the given sorting function and add all of them to the frontier and continue to the loop.



```
const childrenStates = exploredNode.getChildrenStates();

childrenStates.sort(sortFunction); //Sort the children nodes according to given sort function
childrenStates.forEach((child) => {
  frontier.add(child);
});
```

Algorithm Outputs

a)Breadth-first Search



```
Algorithm:  BFS
Time Spent: 1:00:00.028 (h:mm:ss.mmm)
Message: Sub-optimum Solution Found With 26 Remaining Pegs
Expanded Nodes:  591909
Max Number of Nodes Stored in Memory: 5116664
```

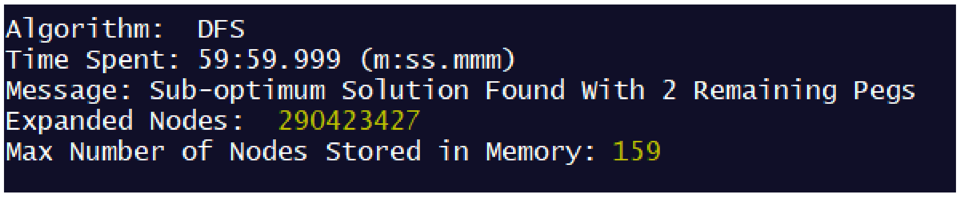
NOTE: Below is output to the terminal and since it is a very long output, we copy pasted it instead of taking a screenshot. If you want, you can test our code.

```
=== Board States Until the Solution. ===

Move #1: 29 => 17

    1 1 1
    1 1 1
1 1 1 1 1 1 1
1 1 1 0 1 1 1
1 1 1 1 1 1 1
    1 1 1
    1 1 1
```

```
Move #2: 10 => 24

    1 1 1
    1 1 1
1 1 1 1 1 1 1
1 1 1 1 1 1 1
1 1 1 0 1 1 1
    1 0 1
    1 1 1



Move #3: 19 => 17

    1 1 1
    1 1 1
1 1 1 0 1 1 1
1 1 1 0 1 1 1
1 1 1 1 1 1 1
    1 0 1
    1 1 1



Move #4: 16 => 18

    1 1 1
    1 1 1
1 1 1 0 1 1 1
1 1 1 1 0 0 1
1 1 1 1 1 1 1
    1 0 1
    1 1 1



Move #5: 14 => 16

    1 1 1
    1 1 1
1 1 1 0 1 1 1
1 1 0 0 1 0 1
1 1 1 1 1 1 1
    1 0 1
    1 1 1
```

```
Move #6: 2 => 10

    1 1 1
    1 1 1
1 1 1 0 1 1 1
0 0 1 0 1 0 1
1 1 1 1 1 1 1
    1 0 1
    1 1 1



    1 0 1
    1 0 1
1 1 1 1 1 1 1
0 0 1 0 1 0 1
1 1 1 1 1 1 1
    1 0 1
    1 1 1
```

b) Depth-First Search



```
Algorithm:  DFS
Time Spent: 59:59.999 (m:ss.mmm)
Message: Sub-optimum Solution Found With 2 Remaining Pegs
Expanded Nodes:  290423427
Max Number of Nodes Stored in Memory: 159
```

```
=== Board States Until the Solution. ===

Move #1: 5 => 17

    1 1 1
    1 1 1
1 1 1 1 1 1 1
1 1 1 0 1 1 1
1 1 1 1 1 1 1
    1 1 1
    1 1 1
```

```
Move #2: 8 => 10

    1 1 1
    1 0 1
1 1 1 0 1 1 1
1 1 1 1 1 1 1
1 1 1 1 1 1 1
    1 1 1
    1 1 1



Move #3: 1 => 9

    1 1 1
    1 0 1
1 0 0 1 1 1 1
1 1 1 1 1 1 1
1 1 1 1 1 1 1
    1 1 1
    1 1 1



Move #4: 3 => 1

    0 1 1
    0 0 1
1 0 1 1 1 1 1
1 1 1 1 1 1 1
1 1 1 1 1 1 1
    1 1 1
    1 1 1



Move #5: 11 => 3

    1 0 0
    0 0 1
1 0 1 1 1 1 1
1 1 1 1 1 1 1
1 1 1 1 1 1 1
    1 1 1
    1 1 1
```

```
Move #6: 16 => 4

    1 0 1
    0 0 0
1 0 1 1 0 1 1
1 1 1 1 1 1 1
1 1 1 1 1 1 1
    1 1 1
    1 1 1



Move #7: 1 => 9

    1 0 1
    1 0 0
1 0 0 1 0 1 1
1 1 0 1 1 1 1
1 1 1 1 1 1 1
    1 1 1
    1 1 1



Move #8: 10 => 8

    0 0 1
    0 0 0
1 0 1 1 0 1 1
1 1 0 1 1 1 1
1 1 1 1 1 1 1
    1 1 1
    1 1 1



Move #9: 7 => 9

    0 0 1
    0 0 0
1 1 0 0 0 1 1
1 1 0 1 1 1 1
1 1 1 1 1 1 1
    1 1 1
    1 1 1
```

```
Move #10: 13 => 11

    0 0 1
    0 0 0
0 0 1 0 0 1 1
1 1 0 1 1 1 1
1 1 1 1 1 1 1
    1 1 1
    1 1 1


Move #11: 18 => 6

    0 0 1
    0 0 0
0 0 1 0 1 0 0
1 1 0 1 1 1 1
1 1 1 1 1 1 1
    1 1 1
    1 1 1


Move #12: 3 => 11

    0 0 1
    0 0 1
0 0 1 0 0 0 0
1 1 0 1 0 1 1
1 1 1 1 1 1 1
    1 1 1
    1 1 1


Move #13: 21 => 7

    0 0 0
    0 0 0
0 0 1 0 1 0 0
1 1 0 1 0 1 1
1 1 1 1 1 1 1
    1 1 1
    1 1 1
```

```
Move #14: 22 => 8

    0 0 0
    0 0 0
1 0 1 0 1 0 0
0 1 0 1 0 1 1
0 1 1 1 1 1 1
    1 1 1
    1 1 1



Move #15: 8 => 10

    0 0 0
    0 0 0
1 1 1 0 1 0 0
0 0 0 1 0 1 1
0 0 1 1 1 1 1
    1 1 1
    1 1 1



Move #16: 24 => 22

    0 0 0
    0 0 0
1 0 0 1 1 0 0
0 0 0 1 0 1 1
0 0 1 1 1 1 1
    1 1 1
    1 1 1



Move #17: 11 => 9

    0 0 0
    0 0 0
1 0 0 1 1 0 0
0 0 0 1 0 1 1
0 1 0 0 1 1 1
    1 1 1
    1 1 1
```

```
Move #18: 31 => 23

    0 0 0
    0 0 0
1 0 1 0 0 0 0
0 0 0 1 0 1 1
0 1 0 0 1 1 1
    1 1 1
    1 1 1



Move #19: 30 => 18

    0 0 0
    0 0 0
1 0 1 0 0 0 0
0 0 0 1 0 1 1
0 1 1 0 1 1 1
    0 1 1
    0 1 1



Move #20: 22 => 24

    0 0 0
    0 0 0
1 0 1 0 0 0 0
0 0 0 1 1 1 1
0 1 1 0 0 1 1
    0 1 0
    0 1 1



Move #21: 18 => 16

    0 0 0
    0 0 0
1 0 1 0 0 0 0
0 0 0 1 1 1 1
0 0 0 1 0 1 1
    0 1 0
    0 1 1
```

```
Move #22: 33 => 31

        0 0 0
        0 0 0
    1 0 1 0 0 0 0
    0 0 1 0 0 1 1
    0 0 0 1 0 1 1
        0 1 0
        0 1 1
```

```
Move #23: 27 => 25

        0 0 0
        0 0 0
    1 0 1 0 0 0 0
    0 0 1 0 0 1 1
    0 0 0 1 0 1 1
        0 1 0
        1 0 0
```

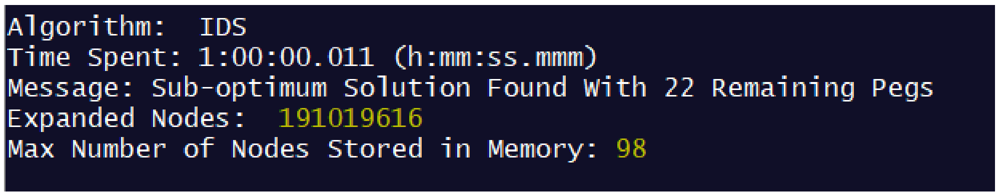
```
Move #24: 20 => 18

        0 0 0
        0 0 0
    1 0 1 0 0 0 0
    0 0 1 0 0 1 1
    0 0 0 1 1 0 0
        0 1 0
        1 0 0
```

```
Move #25: 18 => 30

        0 0 0
        0 0 0
    1 0 1 0 0 0 0
    0 0 1 0 1 0 0
    0 0 0 1 1 0 0
        0 1 0
        1 0 0
```

```
Move #26: 30 => 28

    0 0 0
    0 0 0
1 0 1 0 0 0 0
0 0 1 0 0 0 0
0 0 0 1 0 0 0
    0 1 1
    1 0 0



Move #27: 31 => 23

    0 0 0
    0 0 0
1 0 1 0 0 0 0
0 0 1 0 0 0 0
0 0 0 1 0 0 0
    1 0 0
    1 0 0



Move #28: 24 => 22

    0 0 0
    0 0 0
1 0 1 0 0 0 0
0 0 1 0 0 0 0
0 0 1 1 0 0 0
    0 0 0
    0 0 0



Move #29: 9 => 23

    0 0 0
    0 0 0
1 0 1 0 0 0 0
0 0 1 0 0 0 0
0 1 0 0 0 0 0
    0 0 0
    0 0 0
```

```
Move #30: 22 => 24

    0 0 0
    0 0 0
1 0 0 0 0 0 0
0 0 0 0 0 0 0
0 1 1 0 0 0 0
    0 0 0
    0 0 0



    0 0 0
    0 0 0
1 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 1 0 0 0
    0 0 0
    0 0 0
```

C) Iterative Deepening Search



```
Algorithm:  IDS
Time Spent: 1:00:00.011 (h:mm:ss.mmm)
Message: Sub-optimum Solution Found With 22 Remaining Pegs
Expanded Nodes:  191019616
Max Number of Nodes Stored in Memory: 98
```

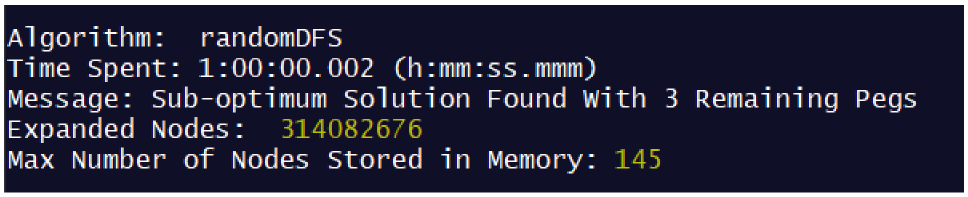
```
=== Board States Until the Solution. ===

Move #1: 19 => 17

    1 1 1
    1 1 1
1 1 1 1 1 1 1
1 1 1 0 1 1 1
1 1 1 1 1 1 1
    1 1 1
    1 1 1
```

```
Move #2: 30 => 18

    1 1 1
    1 1 1
1 1 1 1 1 1 1
1 1 1 1 0 0 1
1 1 1 1 1 1 1
    1 1 1
    1 1 1



Move #3: 17 => 19

    1 1 1
    1 1 1
1 1 1 1 1 1 1
1 1 1 1 1 0 1
1 1 1 1 0 1 1
    1 1 0
    1 1 1



Move #4: 20 => 18

    1 1 1
    1 1 1
1 1 1 1 1 1 1
1 1 1 0 0 1 1
1 1 1 1 0 1 1
    1 1 0
    1 1 1



Move #5: 11 => 25

    1 1 1
    1 1 1
1 1 1 1 1 1 1
1 1 1 0 1 0 0
1 1 1 1 0 1 1
    1 1 0
    1 1 1
```

```
Move #6: 15 => 17

        1 1 1
        1 1 1
    1 1 1 1 0 1 1
    1 1 1 0 0 0 0
    1 1 1 1 1 1 1
        1 1 0
        1 1 1



Move #7: 28 => 16

        1 1 1
        1 1 1
    1 1 1 1 0 1 1
    1 0 0 1 0 0 0
    1 1 1 1 1 1 1
        1 1 0
        1 1 1



Move #8: 9 => 23

        1 1 1
        1 1 1
    1 1 1 1 0 1 1
    1 0 1 1 0 0 0
    1 1 0 1 1 1 1
        0 1 0
        1 1 1



Move #9: 3 => 11

        1 1 1
        1 1 1
    1 1 0 1 0 1 1
    1 0 0 1 0 0 0
    1 1 1 1 1 1 1
        0 1 0
        1 1 1
```

```
Move #10: 1 => 9

    1 1 0
    1 1 0
1 1 0 1 1 1 1
1 0 0 1 0 0 0
1 1 1 1 1 1 1
    0 1 0
    1 1 1



    0 1 0
    0 1 0
1 1 1 1 1 1 1
1 0 0 1 0 0 0
1 1 1 1 1 1 1
    0 1 0
    1 1 1
```

d) Depth-First Search with Random Selection



```
Algorithm:  randomDFS
Time Spent: 1:00:00.002 (h:mm:ss.mmm)
Message: Sub-optimum Solution Found With 3 Remaining Pegs
Expanded Nodes:  314082676
Max Number of Nodes Stored in Memory: 145
```

```
=== Board States Until the Solution. ===

Move #1: 19 => 17

    1 1 1
    1 1 1
1 1 1 1 1 1 1
1 1 1 0 1 1 1
1 1 1 1 1 1 1
    1 1 1
    1 1 1
```

```
Move #2: 6 => 18

    1 1 1
    1 1 1
1 1 1 1 1 1 1
1 1 1 1 0 0 1
1 1 1 1 1 1 1
    1 1 1
    1 1 1



Move #3: 25 => 11

    1 1 1
    1 1 0
1 1 1 1 0 1 1
1 1 1 1 1 0 1
1 1 1 1 1 1 1
    1 1 1
    1 1 1



Move #4: 33 => 25

    1 1 1
    1 1 0
1 1 1 1 1 1 1
1 1 1 1 0 0 1
1 1 1 1 0 1 1
    1 1 1
    1 1 1



Move #5: 31 => 33

    1 1 1
    1 1 0
1 1 1 1 1 1 1
1 1 1 1 0 0 1
1 1 1 1 1 1 1
    1 1 0
    1 1 0
```

```
Move #6: 4 => 6

    1 1 1
    1 1 0
1 1 1 1 1 1 1
1 1 1 1 0 0 1
1 1 1 1 1 1 1
    1 1 0
    0 0 1



Move #7: 24 => 32

    1 1 1
    0 0 1
1 1 1 1 1 1 1
1 1 1 1 0 0 1
1 1 1 1 1 1 1
    1 1 0
    0 0 1



Move #8: 23 => 31

    1 1 1
    0 0 1
1 1 1 1 1 1 1
1 1 1 1 0 0 1
1 1 1 0 1 1 1
    1 0 0
    0 1 1



Move #9: 21 => 23

    1 1 1
    0 0 1
1 1 1 1 1 1 1
1 1 1 1 0 0 1
1 1 0 0 1 1 1
    0 0 0
    1 1 1
```

```
Move #10: 17 => 5

    1 1 1
    0 0 1
1 1 1 1 1 1 1
1 1 1 1 0 0 1
0 0 1 0 1 1 1
    0 0 0
    1 1 1



Move #11: 2 => 10

    1 1 1
    0 1 1
1 1 1 0 1 1 1
1 1 1 0 0 0 1
0 0 1 0 1 1 1
    0 0 0
    1 1 1



Move #12: 16 => 4

    1 0 1
    0 0 1
1 1 1 1 1 1 1
1 1 1 0 0 0 1
0 0 1 0 1 1 1
    0 0 0
    1 1 1



Move #13: 7 => 9

    1 0 1
    1 0 1
1 1 0 1 1 1 1
1 1 0 0 0 0 1
0 0 1 0 1 1 1
    0 0 0
    1 1 1
```

```
Move #14: 6 => 18

    1 0 1
    1 0 1
0 0 1 1 1 1 1
1 1 0 0 0 0 1
0 0 1 0 1 1 1
    0 0 0
    1 1 1



Move #15: 13 => 11

    1 0 1
    1 0 0
0 0 1 1 0 1 1
1 1 0 0 1 0 1
0 0 1 0 1 1 1
    0 0 0
    1 1 1



Move #16: 10 => 8

    1 0 1
    1 0 0
0 0 1 1 1 0 0
1 1 0 0 1 0 1
0 0 1 0 1 1 1
    0 0 0
    1 1 1



Move #17: 14 => 16

    1 0 1
    1 0 0
0 1 0 0 1 0 0
1 1 0 0 1 0 1
0 0 1 0 1 1 1
    0 0 0
    1 1 1
```

```
Move #18: 26 => 24

    1 0 1
    1 0 0
0 1 0 0 1 0 0
0 0 1 0 1 0 1
0 0 1 0 1 1 1
    0 0 0
    1 1 1



Move #19: 18 => 6

    1 0 1
    1 0 0
0 1 0 0 1 0 0
0 0 1 0 1 0 1
0 0 1 1 0 0 1
    0 0 0
    1 1 1



Move #20: 3 => 11

    1 0 1
    1 0 1
0 1 0 0 0 0 0
0 0 1 0 0 0 1
0 0 1 1 0 0 1
    0 0 0
    1 1 1



Move #21: 1 => 9

    1 0 0
    1 0 0
0 1 0 0 1 0 0
0 0 1 0 0 0 1
0 0 1 1 0 0 1
    0 0 0
    1 1 1
```

```
Move #22: 8 => 10

    0 0 0
    0 0 0
0 1 1 0 1 0 0
0 0 1 0 0 0 1
0 0 1 1 0 0 1
    0 0 0
    1 1 1



Move #23: 16 => 28

    0 0 0
    0 0 0
0 0 0 1 1 0 0
0 0 1 0 0 0 1
0 0 1 1 0 0 1
    0 0 0
    1 1 1



Move #24: 10 => 12

    0 0 0
    0 0 0
0 0 0 1 1 0 0
0 0 0 0 0 0 1
0 0 0 1 0 0 1
    1 0 0
    1 1 1



Move #25: 27 => 13

    0 0 0
    0 0 0
0 0 0 0 0 1 0
0 0 0 0 0 0 1
0 0 0 1 0 0 1
    1 0 0
    1 1 1
```

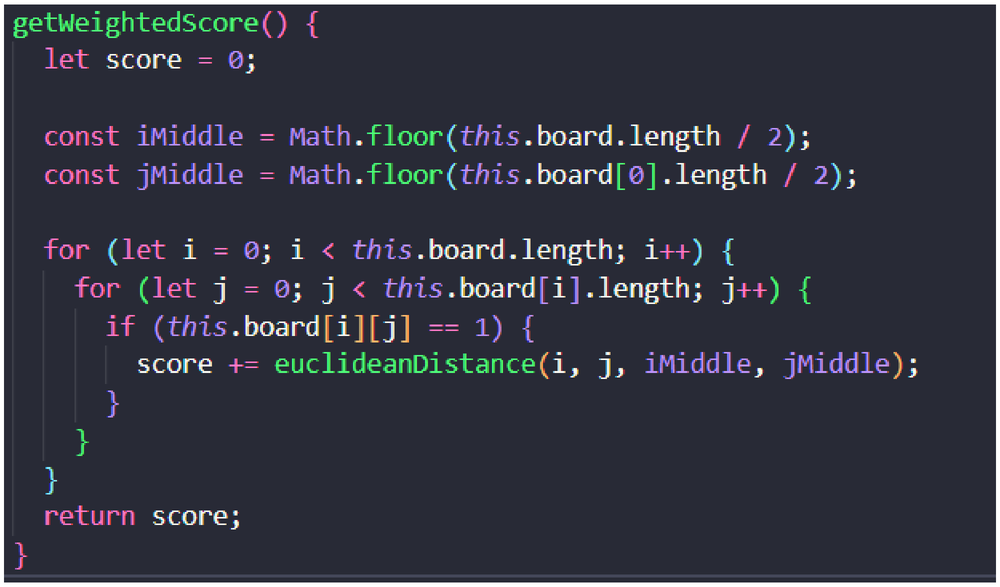
```
Move #26: 13 => 11

        0 0 0
        0 0 0
0 0 0 0 0 1 1
0 0 0 0 0 0 0
0 0 0 1 0 0 0
        1 0 0
        1 1 1


Move #27: 31 => 23

        0 0 0
        0 0 0
0 0 0 0 1 0 0
0 0 0 0 0 0 0
0 0 0 1 0 0 0
        1 0 0
        1 1 1


Move #28: 23 => 25

        0 0 0
        0 0 0
0 0 0 0 1 0 0
0 0 0 0 0 0 0
0 0 1 1 0 0 0
        0 0 0
        0 1 1


Move #29: 33 => 31

        0 0 0
        0 0 0
0 0 0 0 1 0 0
0 0 0 0 0 0 0
0 0 0 0 1 0 0
        0 0 0
        0 1 1


        0 0 0
        0 0 0
0 0 0 0 1 0 0
0 0 0 0 0 0 0
0 0 0 0 1 0 0
        0 0 0
        1 0 0
```

e) Depth-First Search with a Node Selection Heuristic

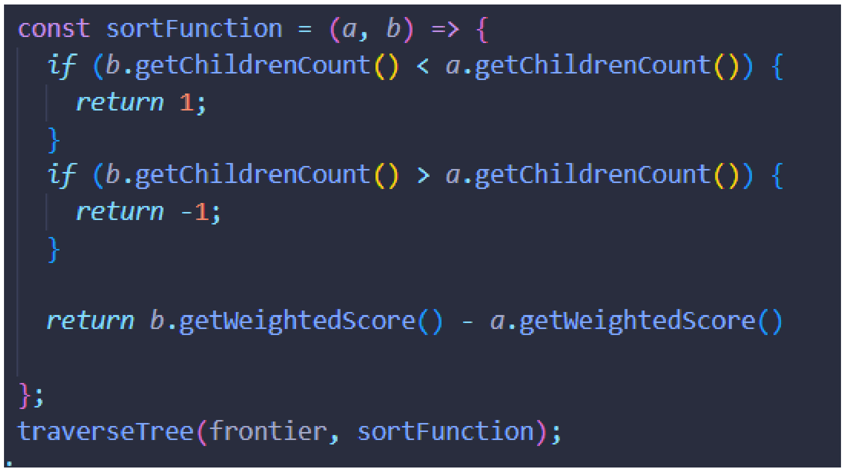
Heuristic Function:

In our heuristic function, we first choose the node which yields more children states. And if they are equal, we choose the node which has the least **weighted score**.

Weighted score is the sum of all the Euclidean distances between each peg and the center. In this way, we get an optimal Solution in less than 1 second by expanding 19057 nodes and storing maximum of 298 nodes in the memory.

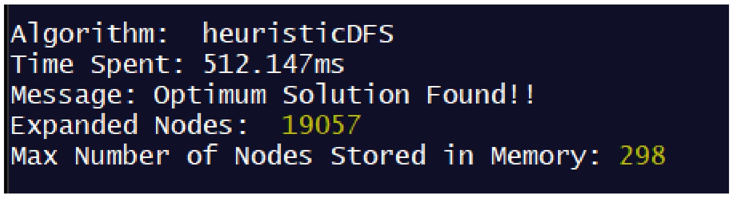


```
getWeightedScore() {
  let score = 0;

  const iMiddle = Math.floor(this.board.length / 2);
  const jMiddle = Math.floor(this.board[0].length / 2);

  for (let i = 0; i < this.board.length; i++) {
    for (let j = 0; j < this.board[i].length; j++) {
      if (this.board[i][j] == 1) {
        score += euclideanDistance(i, j, iMiddle, jMiddle);
      }
    }
  }
  return score;
}
```



```
const sortFunction = (a, b) => {
  if (b.getChildrenCount() < a.getChildrenCount()) {
    return 1;
  }
  if (b.getChildrenCount() > a.getChildrenCount()) {
    return -1;
  }

  return b.getWeightedScore() - a.getWeightedScore()

};
traverseTree(frontier, sortFunction);
```

Output:



```
Algorithm:  heuristicDFS
Time Spent: 512.147ms
Message: Optimum Solution Found!!
Expanded Nodes:  19057
Max Number of Nodes Stored in Memory: 298
```

```
=== Board States Until the Solution. ===

Move #1: 29 => 17

    1 1 1
    1 1 1
1 1 1 1 1 1 1
1 1 1 0 1 1 1
1 1 1 1 1 1 1
    1 1 1
    1 1 1



Move #2: 26 => 24

    1 1 1
    1 1 1
1 1 1 1 1 1 1
1 1 1 1 1 1 1
1 1 1 0 1 1 1
    1 0 1
    1 1 1
```

```
Move #3: 11 => 25

    1 1 1
    1 1 1
1 1 1 1 1 1 1
1 1 1 1 1 1 1
1 1 1 1 0 0 1
    1 0 1
    1 1 1



Move #4: 9 => 11

    1 1 1
    1 1 1
1 1 1 1 0 1 1
1 1 1 1 0 1 1
1 1 1 1 1 0 1
    1 0 1
    1 1 1



Move #5: 23 => 9

    1 1 1
    1 1 1
1 1 0 0 1 1 1
1 1 1 1 0 1 1
1 1 1 1 1 0 1
    1 0 1
    1 1 1



Move #6: 14 => 16

    1 1 1
    1 1 1
1 1 1 0 1 1 1
1 1 0 1 0 1 1
1 1 0 1 1 0 1
    1 0 1
    1 1 1
```

```
Move #7: 31 => 23

    1 1 1
    1 1 1
1 1 1 0 1 1 1
0 0 1 1 0 1 1
1 1 0 1 1 0 1
    1 0 1
    1 1 1



Move #8: 33 => 31

    1 1 1
    1 1 1
1 1 1 0 1 1 1
0 0 1 1 0 1 1
1 1 1 1 1 0 1
    0 0 1
    0 1 1



Move #9: 16 => 28

    1 1 1
    1 1 1
1 1 1 0 1 1 1
0 0 1 1 0 1 1
1 1 1 1 1 0 1
    0 0 1
    1 0 0



Move #10: 4 => 16

    1 1 1
    1 1 1
1 1 1 0 1 1 1
0 0 0 1 0 1 1
1 1 0 1 1 0 1
    1 0 1
    1 0 0
```

```
Move #11: 7 => 9

    1 1 1
    0 1 1
1 1 0 0 1 1 1
0 0 1 1 0 1 1
1 1 0 1 1 0 1
    1 0 1
    1 0 0



Move #12: 2 => 10

    1 1 1
    0 1 1
0 0 1 0 1 1 1
0 0 1 1 0 1 1
1 1 0 1 1 0 1
    1 0 1
    1 0 0



Move #13: 6 => 18

    1 0 1
    0 0 1
0 0 1 1 1 1 1
0 0 1 1 0 1 1
1 1 0 1 1 0 1
    1 0 1
    1 0 0



Move #14: 13 => 11

    1 0 1
    0 0 0
0 0 1 1 0 1 1
0 0 1 1 1 1 1
1 1 0 1 1 0 1
    1 0 1
    1 0 0
```

```
Move #15: 18 => 6

    1 0 1
    0 0 0
0 0 1 1 1 0 0
0 0 1 1 1 1 1
1 1 0 1 1 0 1
    1 0 1
    1 0 0
```

```
Move #16: 3 => 11

    1 0 1
    0 0 1
0 0 1 1 0 0 0
0 0 1 1 0 1 1
1 1 0 1 1 0 1
    1 0 1
    1 0 0
```

```
Move #17: 10 => 12

    1 0 0
    0 0 0
0 0 1 1 1 0 0
0 0 1 1 0 1 1
1 1 0 1 1 0 1
    1 0 1
    1 0 0
```

```
Move #18: 27 => 13

    1 0 0
    0 0 0
0 0 1 0 0 1 0
0 0 1 1 0 1 1
1 1 0 1 1 0 1
    1 0 1
    1 0 0
```

```
Move #19: 13 => 11

    1 0 0
    0 0 0
0 0 1 0 0 1 1
0 0 1 1 0 1 0
1 1 0 1 1 0 0
    1 0 1
    1 0 0



Move #20: 30 => 18

    1 0 0
    0 0 0
0 0 1 0 1 0 0
0 0 1 1 0 1 0
1 1 0 1 1 0 0
    1 0 1
    1 0 0



Move #21: 24 => 10

    1 0 0
    0 0 0
0 0 1 0 1 0 0
0 0 1 1 1 1 0
1 1 0 1 0 0 0
    1 0 0
    1 0 0



Move #22: 31 => 23

    1 0 0
    0 0 0
0 0 1 1 1 0 0
0 0 1 0 1 1 0
1 1 0 0 0 0 0
    1 0 0
    1 0 0
```

```
Move #23: 16 => 28

    1 0 0
    0 0 0
0 0 1 1 1 0 0
0 0 1 0 1 1 0
1 1 1 0 0 0 0
    0 0 0
    0 0 0



Move #24: 21 => 23

    1 0 0
    0 0 0
0 0 1 1 1 0 0
0 0 0 0 1 1 0
1 1 0 0 0 0 0
    1 0 0
    0 0 0


Move #25: 28 => 16

    1 0 0
    0 0 0
0 0 1 1 1 0 0
0 0 0 0 1 1 0
0 0 1 0 0 0 0
    1 0 0
    0 0 0



Move #26: 16 => 4

    1 0 0
    0 0 0
0 0 1 1 1 0 0
0 0 1 0 1 1 0
0 0 0 0 0 0 0
    0 0 0
    0 0 0
```

```
Move #27: 1 => 9

        1 0 0
        1 0 0
    0 0 0 1 1 0 0
    0 0 0 0 1 1 0
    0 0 0 0 0 0 0
        0 0 0
        0 0 0
```

```
Move #28: 18 => 6

        0 0 0
        0 0 0
    0 0 1 1 1 0 0
    0 0 0 0 1 1 0
    0 0 0 0 0 0 0
        0 0 0
        0 0 0
```

```
Move #29: 9 => 11

        0 0 0
        0 0 1
    0 0 1 1 0 0 0
    0 0 0 0 0 1 0
    0 0 0 0 0 0 0
        0 0 0
        0 0 0
```

```
Move #30: 6 => 18

        0 0 0
        0 0 1
    0 0 0 0 1 0 0
    0 0 0 0 0 1 0
    0 0 0 0 0 0 0
        0 0 0
        0 0 0
```

```
Move #31: 19 => 17

    0 0 0
    0 0 0
0 0 0 0 0 0 0
0 0 0 0 1 1 0
0 0 0 0 0 0 0
    0 0 0
    0 0 0


    0 0 0
    0 0 0
0 0 0 0 0 0 0
0 0 0 1 0 0 0
0 0 0 0 0 0 0
    0 0 0
    0 0 0
```

Note: If you want to run this program, you need to have Node.js installed on your computer. To run the program after you install node.js, go to the project directory and run first **“npm install”** and after you install the dependencies run **“node src/index.js”.** But when we run our JavaScript code for the Breadth-first Search algorithm, we run the file with the following command:

node max_old_space_size=16384 src/index.js

This will give the nodejs app a 16gb memory heap size. If we don’t do this, it will give a “heap out of memory” error (Only for BFS) and since we want the program to run for an hour, we give larger heap memory.

Conclusion

For the BFS algorithm, we expected that frontier will allocate too much space because it will only pop 1 node from the beginning of the queue and then push the children of it. Since generally the number of children of each node are higher than 1, in every step we add more nodes than we expand. Thus, the number of nodes in the frontier will increase every second. This was our assumption, and it turned out to be true. We got a “heap out of memory” error when we didn’t enlarge the size of the heap memory by running the program with a specific command.

The other assumption we made is that since the goal state has only 1 remaining peg, we thought that in one hour, BFS algorithm could not even reach to the depth level of 15 or 20. Because we assumed that the branch factor would be high and at the depth level of 15 or 20, there would be too many nodes to look and store in the frontier. We both thought that it wouldn't find the goal state and give an error and we have learned that we were right about our assumptions. BFS couldn't even reach to the depth level 7 in our experiment. In the end, BFS found a sub-optimal solution at the depth level of 6 (with 26 remaining pegs of course), which is the worst end possible.

The DFS algorithm allocated very little space and didn't give any heap memory error. It found a sub-optimal solution with 2 remaining pegs. It extended hundreds of millions of nodes. We assumed that it would find much better solutions than the BFS algorithm since it right away looks deeper levels and deeper levels means fewer remaining pegs. It gave the output as we expected. The difference between the BFS and randomized BFS was very small (as small as 1 remaining peg difference). IDS founds solutions that is closer to the of the BFS, but it allocates very small memory compared to the BFS. This was also something we expected to happen.