

**MARMARA UNIVERSITY**

**FACULTY OF ENGINEERING**

**2022 ARTIFICIAL INTELLIGENCE PROJECT REPORT**

**Students Name:** Fatih Genç *150119905 Kutay Başkurt 150119645*

**Department:** Computer Science & Engineering

**Defining Matrix:**

First of all, it was necessary to adapt the image of the board to the code. So we created a matrix. The 0's represent the outside of the board, the letter A represents that area is full, and the letter P represents that area is empty. we created the starting position of the board according to this matrix that I designed, and we created the desired final state in my matrix as we designed it. The matrix looks like below. Xs indicate full places, 0s indicate outside of the field and E indicates empty. We need to check 2 right, 2 left, 2 down and 2 up of each peg. That's why the 0's help us determine where the side pegs will move.

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 X X X 0 0 0 0

0 0 0 0 X X X 0 0 0 0

0 0 X X X X X X X 0 0

0 0 X X X E X X X 0 0

0 0 X X X X X X X 0 0

0 0 0 0 X X X 0 0 0 0

0 0 0 0 X X X 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

**Breadth-First Search:**

We have defined our Frontier List, Explored List, Memory Counter, Time Counter, Optimal Counter, Sub Optimal List , Optimal List , Temp Frontier List to control the time and our other variables to code the Breadth First Search algorithm. The information we keep in the nodes are Current Move, Past Moves, Removed Peg Number, Board. After adding the initial image of the board to our frontier, we created the loop where our search method will run. This loop will continue to work by looking at the Time Counter, Optimal Counter and Memory Counter values. If a Counter comes in a different value than requested, it will no longer enter the loop and exit. We created a for loop inside our loop that will check around each empty peg and find possible moves. After finding the possible moves, we keep the image of the board formed after the move using the New Matrix Function. We add the new move to the Parent in Past Moves value. We add the board image to our Temp Frontier using the Current Move, Past Move and Removed Peg Number files. And we delete the last added move from the parent's Past Move and repeat the same operations for other possible moves. After that, we sort the nodes in the Temp Frontier according to their Removed Peg Numbers and add the end of our Frontier. We add the first element of our Frontier to our Explored list and pop the first element of the Frontier. Before doing this, we check if the solution is Sub Optimal or Optimal. If it is Optimal, we change our Optimal Counter and ensure that it does not enter the loop again. If Sub Optimal we check if there is a Sub Optimal already added. If it is, we decide whether to add it or not according to the remaining peg numbers. If we are going to add, we delete the previous element and add our new move. After that, we reset our necessary variables and check if we have exceeded the time limit. Accordingly, we change our Time Counter. After the loop of the search method ends or exits the loop, we print the desired information on the screen. We get the length of the Explored List as the Expanded Node. For Max memory, we add the lengths of the Explored List and the Frontier List. If the Sub Optimal solution is found, we print the state of the board on the screen in an understandable way by using the print\_board function.

**metin içeren bir resim

Açıklama otomatik olarak oluşturuldu**

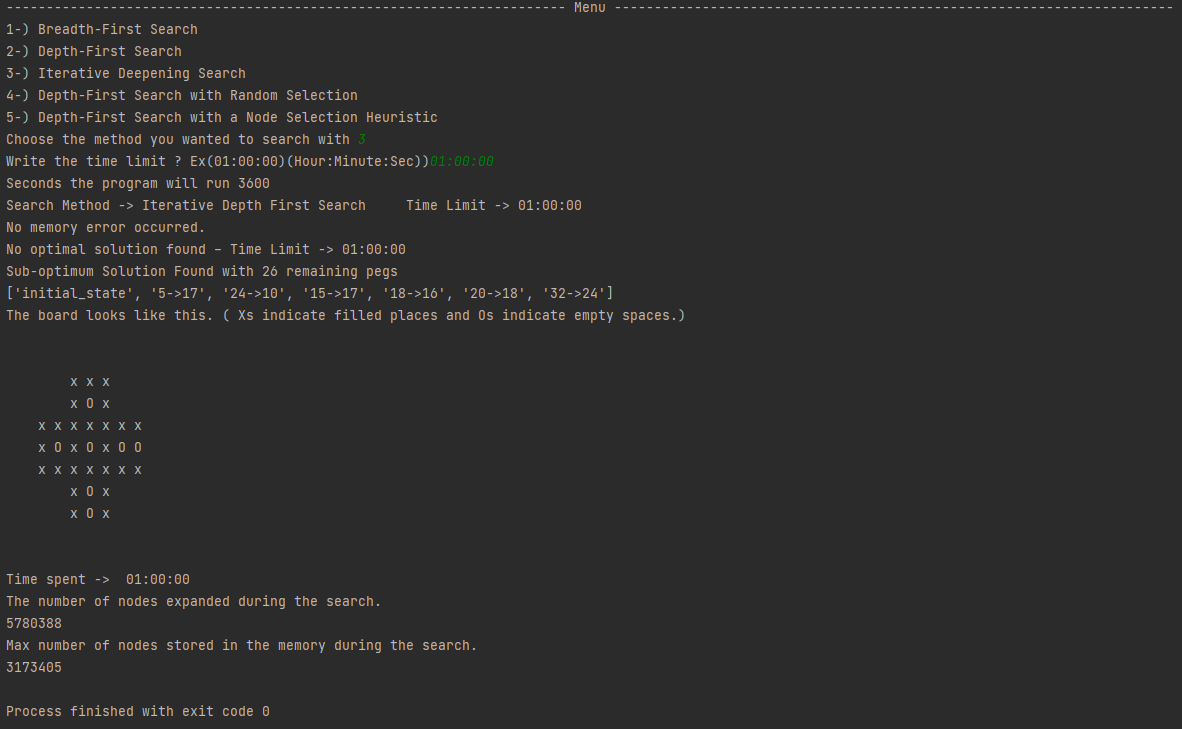
**Depth First Search:**

Here we define the variables we used in Breadth First Search while coding our Depth First Search. We check the pegs one by one. Unlike Breadth First Search, we did the following. We kept the moves that can be made according to the appearance of the board in the temp list. We ordered this list. Instead of adding it to the end of the Frontier, we add it as sorted at the beginning of the Frontier.**metin içeren bir resim

Açıklama otomatik olarak oluşturuldu**

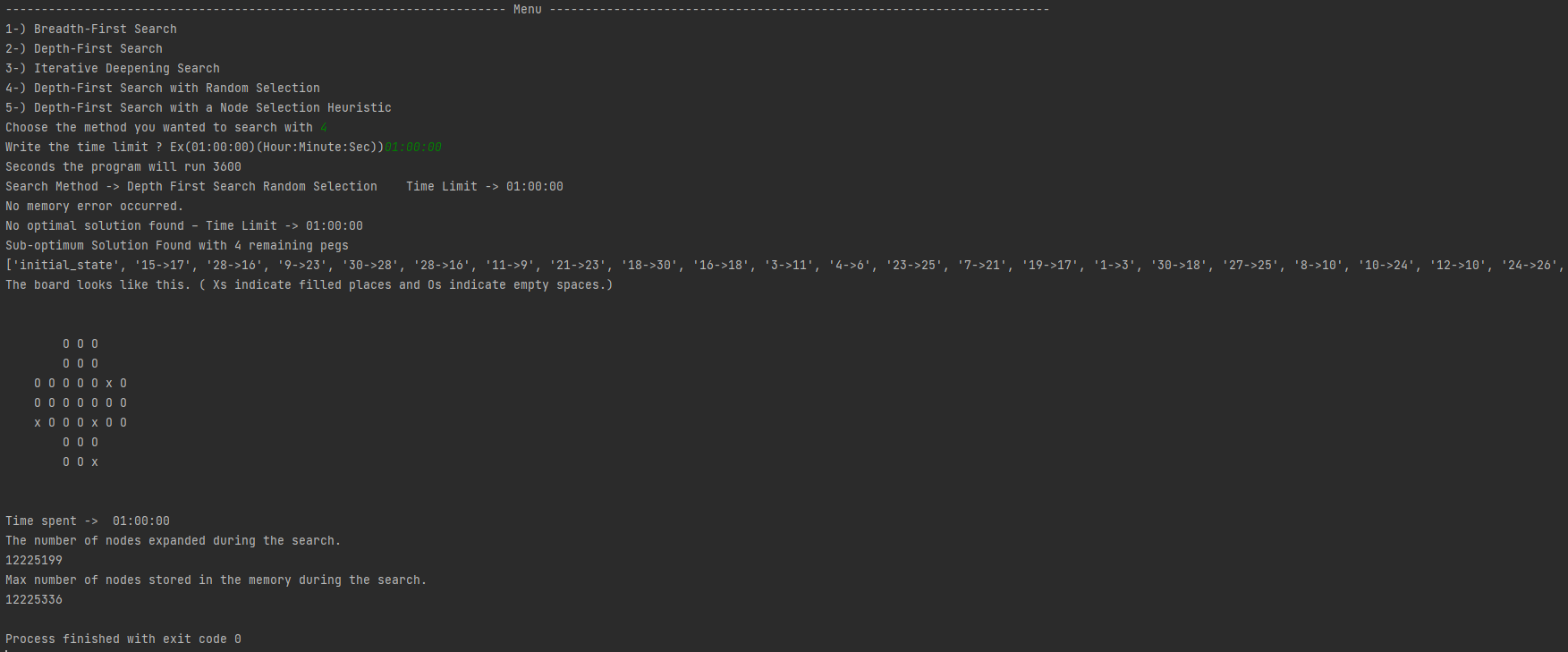
**Iterative Depth First Search:**

For Iterative Depth First Search, we use explored list, frontier list, memory counter, iterative counter, time counter, max memory number, max memory temp number, explored list item counter, optimal solutions, and sub optimal list counters. First, we used a loop to increase the Iterative counter by one. We used another loop that allows it to perform a depth first search for each iterative counter level. In this loop, we searched for possible moves of the board, as we did in Depth First Search and Breadth First Search and ranked them according to the removed peg. After that, if the level of the possible move is more than the iterative counter value, the frontier does not add the list. It just adds the parent to the expanded node and removes it from the frontier list. If it is equal to and less than the value of the iterative counter, it adds the possible moves to the beginning of the frontier. It checks the time limits and increases the explored list item counter by one. Frontier and Explored list lengths are greater than max memory temp number. We equate them to each other. When we exit this loop, we change the max memory counter value if it is less than the temp value. Then we print the necessary information on the screen.



**Depth-First Search with Random Selection:**

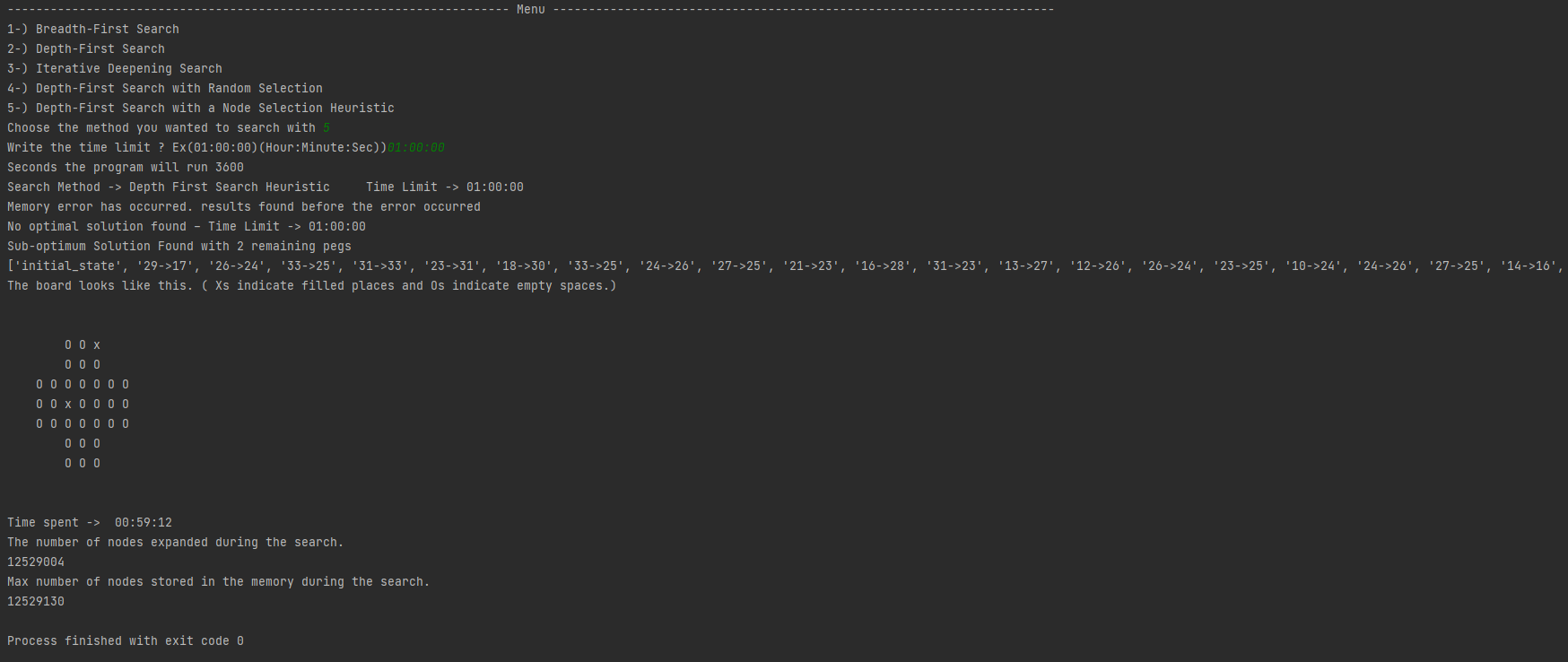
The codes of this search method are the same as Depth first search. The only difference is when sorting my temp frontier list by the number of peg's removed. We shuffle this list using Shuffle. Thus, we ensure that a random selection is made.





**Depth-First Search with a Node Selection Heuristic:**

When determining our strategy, we considered what we did in the previous stages. In the first 3 methods, our strategy was to rank from smallest to largest. For this method, we made the Heuristic selection in order from largest to smallest. Instead of making the number of the peg removed while making a move, we sorted it from biggest to smallest. We reversed the list by taking the code of the list that we had previously ordered from smallest to biggest. We have determined a Heuristic that is the exact opposite of the one used in the first three methods, and we have continued our operations accordingly.

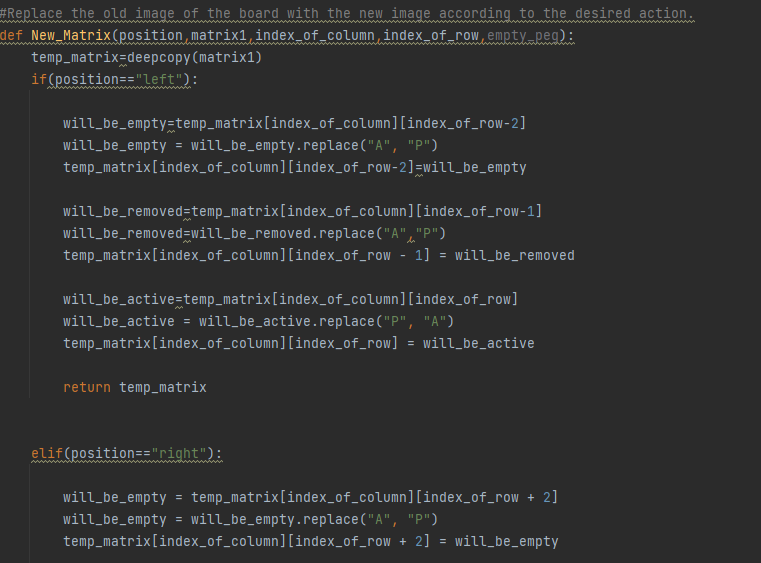




**Other Functions:**

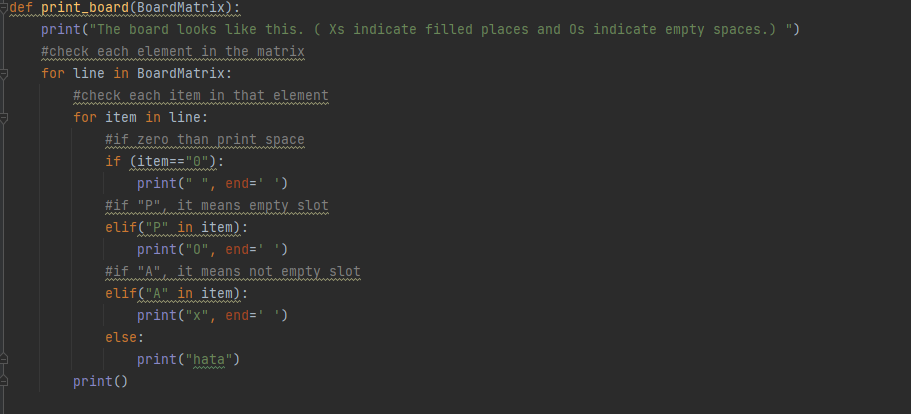
**New\_Matrix:**

As you can see in the picture, this function saves the changes made on the board in a matrix after finding the possible moves and gives us the new state of the board according to the parameters used.

****

**Print\_Board:**

This function has been created to print the image of the board in the code on the screen in a clear way.

****

**Time\_Operations:**

This function converts the time limit format entered by the user to seconds and sends it to the search functions.

**metin içeren bir resim

Açıklama otomatik olarak oluşturuldu**

**Menu:**

The function, which is partially visible in the photo, is made to guide the user and to choose the desired search method. In addition, it asks the user for the time limit. If the user does not enter the information in the desired format, the user requests the information to be entered again. When the user enters it correctly, the desired search methods start to work.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu