Batch: C3	Roll No.: 121
Experiment / assignment / tutorial No. 4	

Title: Implementation of uninformed search algorithm(BFS/DFS/DLS/IDS)

Objective: Comparison and analysis of uninformed search algorithms

Expected Outcome of Experiment:

Course Outcome	After successful completion of the course students should be able to
CO 2	Analyse and solve problems for goal based agent architecture (searching and planning algorithms).

Books/ Journals/ Websites referred:

1. **"Artificial Intelligence: a Modern Approach" by Russell and Norving, Pearson education Publications**
2. **"Artificial Intelligence" By Rich and knight, Tata Mcgraw Hill Publications**
3. http://people.cs.pitt.edu/~milos/courses/cs2710/lectures/Class4.pdf
4. http://cs.williams.edu/~andrea/cs108/Lectures/InfSearch/infSearch.html
5. **http://www.cs.mcgill.ca/~dprecup/courses/AI/Lectures/ai-lecture02.pdf**
   http://homepage.cs.uiowa.edu/~hzhang/c145/notes/04a-search.pdf
6. http://wiki.answers.com/Q/Informed_search_techniques_and_uninformed_search_techniques
7. www.cs.swarthmore.edu/~eeaton/teaching/cs63/.../UninformedSearch.ppt
8. https://medium.com/nerd-for-tech/ai-search-algorithms-with-examples-54772c6d973a
9. https://www.geeksforgeeks.org/iterative-deepening-searchids-iterative-deepening-depth-first-searchiddfs/
10. https://www.geeksforgeeks.org/search-algorithms-in-ai/
11. https://www.geeksforgeeks.org/applications-of-breadth-first-traversal/

Pre Lab/ Prior Concepts: Problem solving, state-space trees, problem formulation, goal based agent architecture

Historical Profile:

The AI researchers have come up many algorithms those operate on state space tree to give the result. Goal based agent architectures solve problems through searching or planning. Depending on availability of more information other than the problem statement decides if the solution can be obtained with uninformed search or informed search.

Its fact that not all search algorithms end up in giving the optimal solution. So, it states the need to have a better and methodological approach which guarantees optimal solution.

New Concepts to be learned: Uninformed (blind) search, iterative deepening, greedy best first search, A* search

---

Uninformed searching techniques:

- Breadth first search
- Depth first search
- Iterative deepening search
- Depth limit search

Chosen Problem statement:
8 Queens Problem

States: The total number of combinations which are possible (incomplete/not-allowed as well as allowed). For example, a board wherein no queen has been placed yet, i.e., 64 empty squares, is one of the possible states which is incomplete. A board wherein two queens attack each other being in the same row, column or diagonal is also one of the possible states which is not allowed. Finally a board wherein all eight queens have been placed in a way that no two queens attack each other is one of the possible states, which is allowed. The number of incomplete as well as not allowed as well as allowed states are 8! = 40,320. The number of allowed states i.e., the number of solutions is 92.

Initial State: No queen has been placed on the board yet.

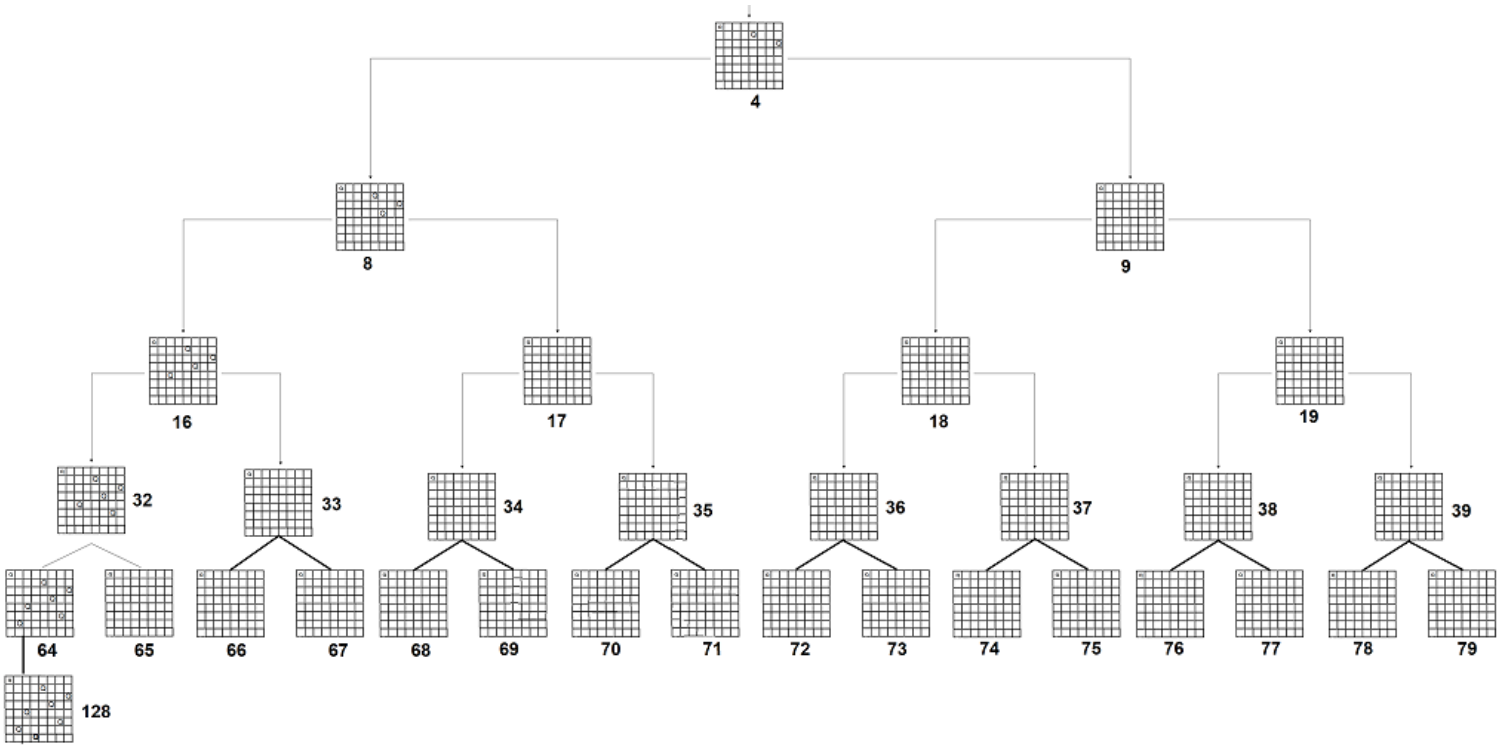
Actions: Add the next queen to the board if there is room.

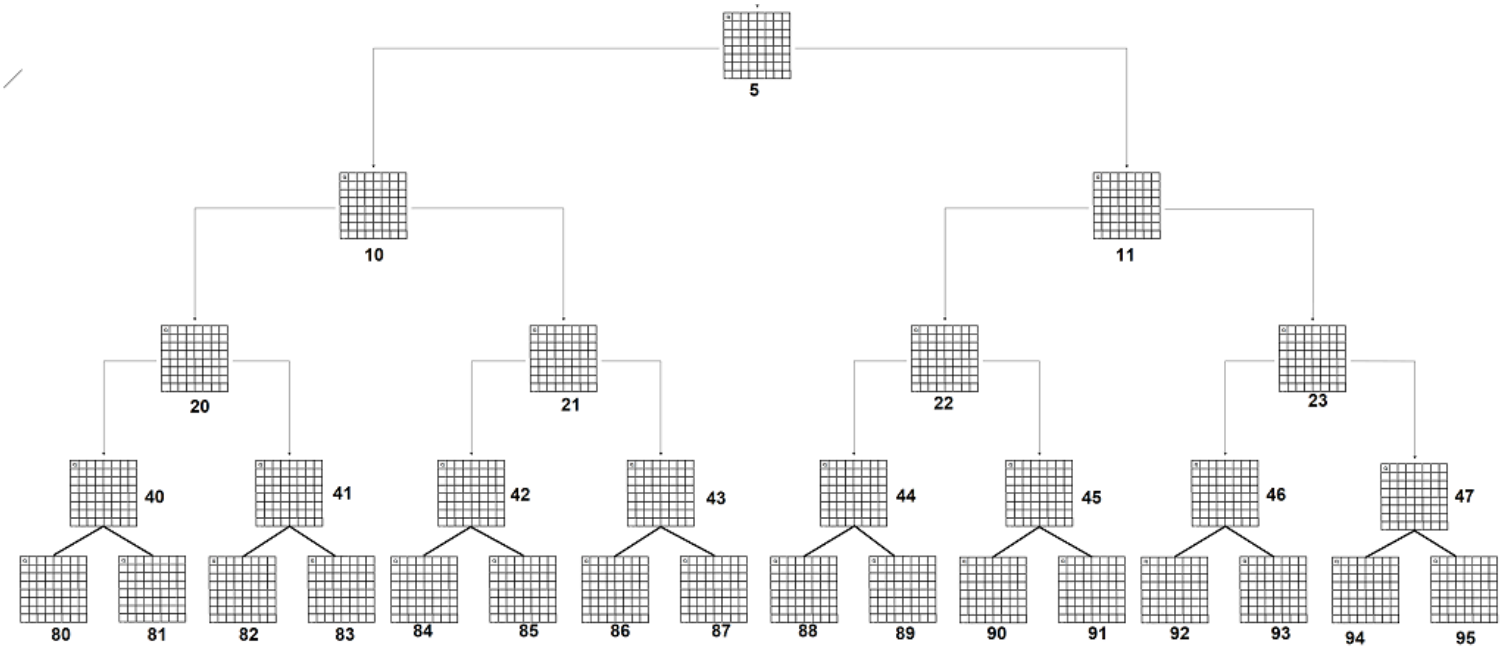
Transition Model: Returns the board with a queen added at a square in the next row such that the conditions are met.

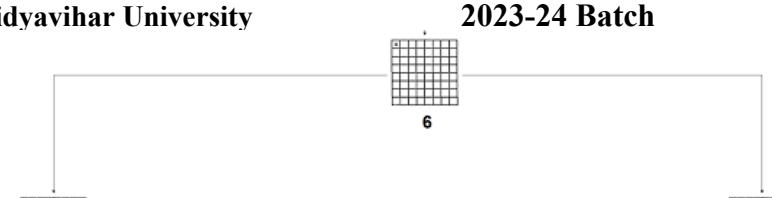
Goal test: All eight queens are placed without threatening each other.

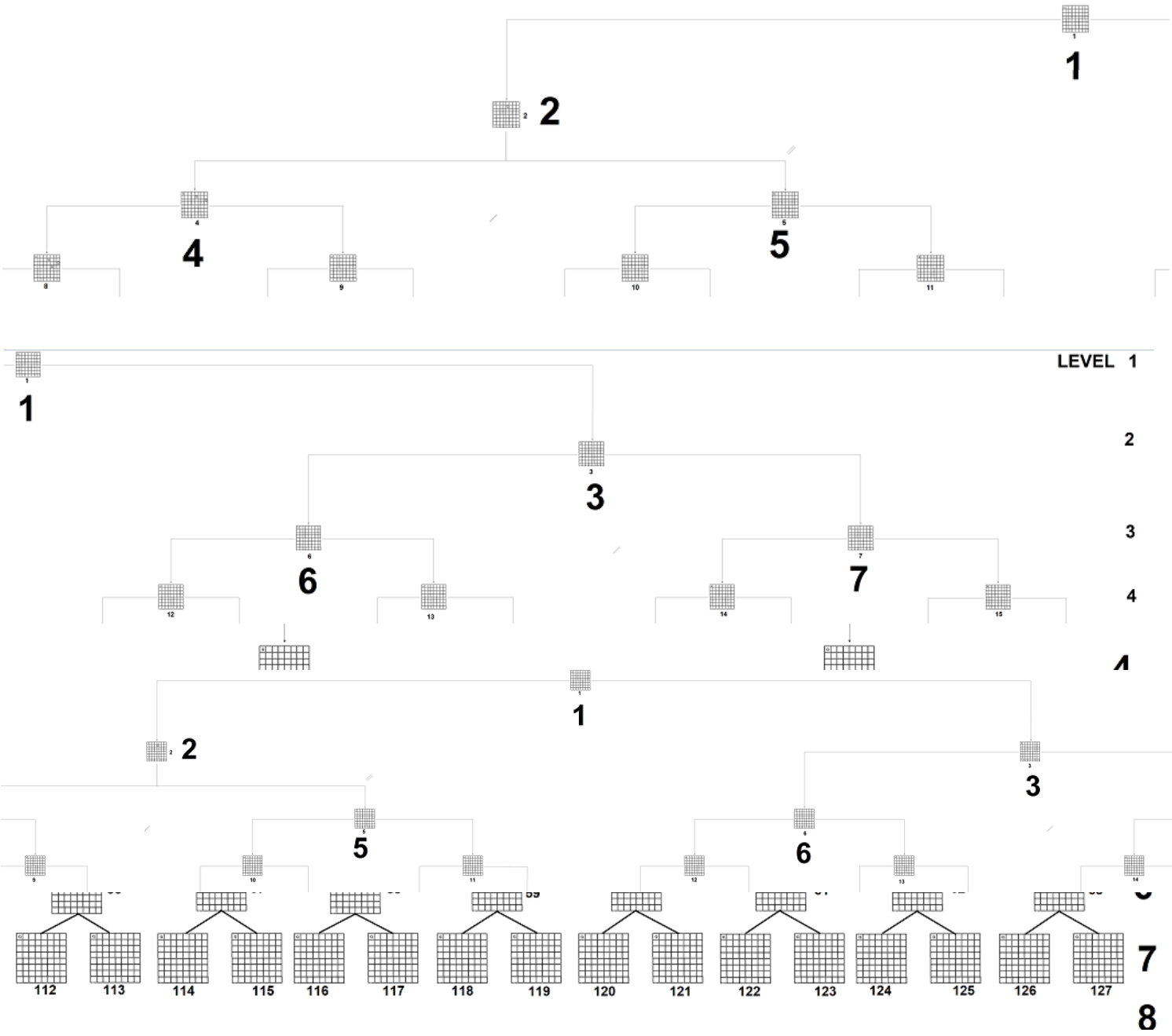
Path cost: Same (unit) path cost to go from one node to the next.

State-space tree :









Note: As it is not possible to display screenshot of the State Space Tree as a single screenshot due to its size, a separate PNG file of the State Space Tree is attached along with this write up.

Solution with of chosen algorithm on the state-space tree:

Breadth First Search (BFS)

1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 - 11 - 12 - 13 - 14 - 15 - 16 - 17 - 18 - 19 - 20 - 21 - 22 - 23 - 24 - 25 - 26 - 27 - 28 - 29 - 30 - 31 - 32 - 33 - 34 - 35 - 36 - 37 - 38 - 39 - 40 - 41 - 42 - 43 - 44 - 45 - 46 - 47 - 48 - 49 - 50 - 51 - 52 - 53 - 54 - 55 - 56 - 57 - 58 - 59 - 60 - 61 - 62 - 63 - 64 - 65 - 66 - 67 - 68 - 69 - 70 - 71 - 72 - 73 - 74 - 75 - 76 - 77 - 78 - 79 - 80 - 81 - 82 - 83 - 84 - 85 - 86 - 87 - 88 - 89 - 90 - 91 - 92 - 93 - 94 - 95 - 96 - 97 - 98 - 99 - 100 - 101 - 102 - 103 - 104 - 105 - 106 - 107 - 108 - 109 - 110 - 111 - 112 - 113 - 114 - 115 - 116 - 117 - 118 - 119 - 120 - 121 - 122 - 123 - 124 - 125 - 126 - 127 - **128**

Depth First Search (DFS)

1 - 2 - 4 - 8 - 16 - 32 - 64 - **128**

Depth Limited Search (DLS)

Level = 8

1 - 2 - 4 - 8 - 16 - 32 - 64 - **128**

Comparison of performance of uninformed Algorithm:

For N queen problem, depth search is better as it does not require as much memory as the breadth first search.

Depth First Search

```
/*************************************************************
*************************


Need to put value of N
```

```
**********************************************************
************************/

public class queen

{

   public static void main(String[] args) {

        System.out.println("Hello World");

        int N=8;

         int[][] positions = new int[N][N];



        boolean a = place(0,positions);



        //printPos(positions);

    }

     public static boolean place(int queenNo,int[][]
positions){

        boolean val=false;

                                                    for(int
row=0;row<positions[0].length;row++){//iterate row



if(checkQueen(queenNo,positions,row)==true){
```

```
                //place queen
                    positions[row][queenNo]=1; //column
is queenNo


                    if(queenNo==positions[0].length-1){
                        //end of recursion
                        printPos(positions);
                        positions[row][queenNo]=0;
                        return true;
                    }
                    //recur

if(place(queenNo+1,positions)==false){
                        //failed
                         positions[row][queenNo]=0;
                              continue; //look for next
position
                    }
                    else{
                        val = true;//valid position
```

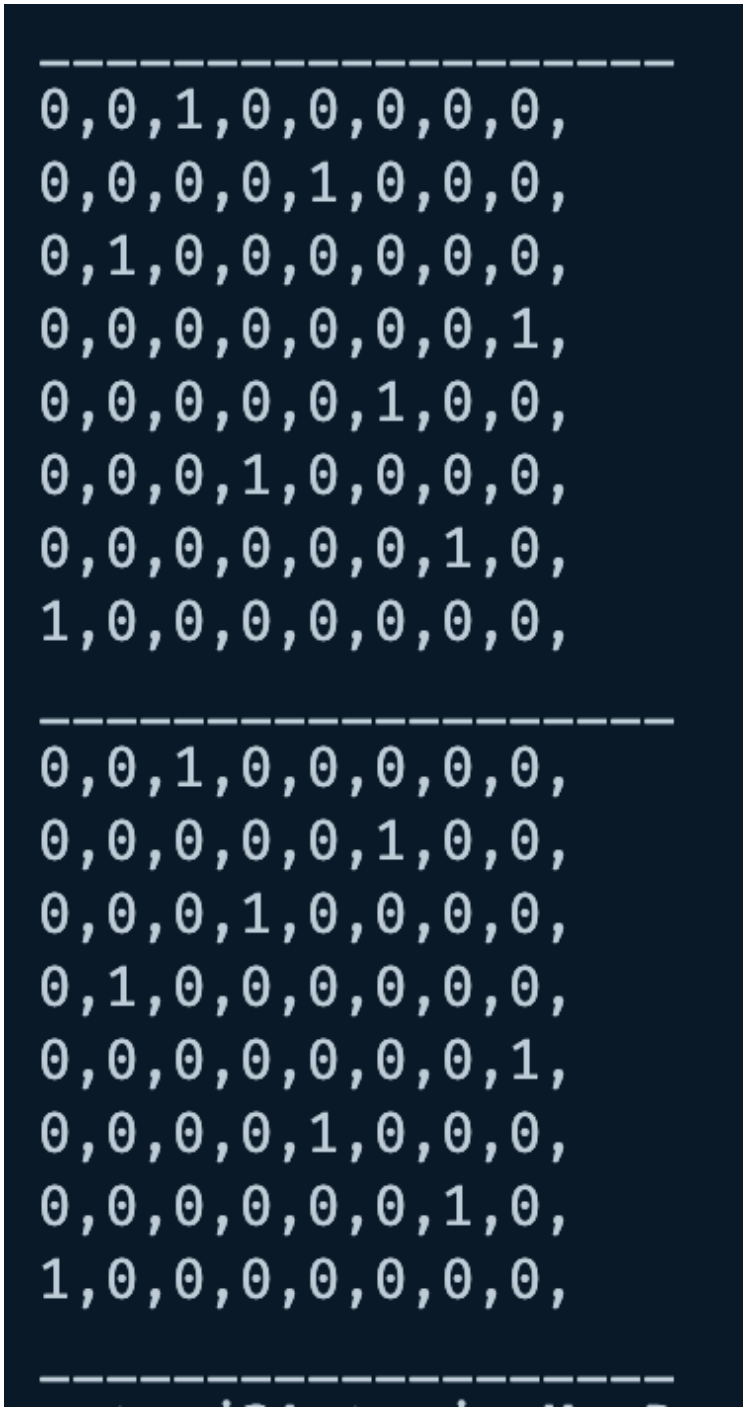
```
                        positions[row][queenNo]=0;
                        continue; //still look for next
position


                    }
                }
            }
            if(val==true){
                return true;
            }
            //backtrack
            return false;
        }
    public static boolean checkQueen(int column,int
[][] positions, int row){


if(checkRow(column,positions,row)&&checkdiaLeft(column
,positions,row)&&checkdiaRight(column,positions,row)){
                return true;
            }
```

```
        return false;



    }

    public static boolean checkRow(int column,int [][]
positions, int row){

                for(int  i=0;i<positions[0].length;i++){
//iterate columns

            if(positions[row][i]==1){


                 return false;

            }

        }

        return true;

    }

    public static boolean checkdiaLeft(int column,int
[][] positions, int row){

                for(int  i=0;i<positions[0].length;i++){
//iterate row

                for(int  j=0;j<positions[0].length;j++){
//iterate columns

           if(i-j==row-column){
```

```
            if(positions[i][j]==1 ){


              return false;

            }

          }

        }


    }

     return true;

    }


     public static boolean checkdiaRight(int column,int
[][] positions, int row){

              for(int  i=0;i<positions[0].length;i++){
//iterate row

              for(int  j=0;j<positions[0].length;j++){
//iterate columns

          if(i+j==row+column){
```

```
            if(positions[i][j]==1 ){



                return false;

             }

           }

       }



   }

    return true;

   }




   public static void printPos(int [][] positions){

               for(int  i=0;i<positions[0].length;i++){
//iterate columns

                for(int j=0;j<positions[0].length;j++){
//iterate rows

            System.out.print(positions[i][j]+",");

       }

       System.out.println();
```

```
        }

        System.out.println("_____");

    }

}
```


0,0,1,0,0,0,0,0,
0,0,0,0,1,0,0,0,
0,1,0,0,0,0,0,0,
0,0,0,0,0,0,0,1,
0,0,0,0,0,1,0,0,
0,0,0,1,0,0,0,0,
0,0,0,0,0,0,1,0,
1,0,0,0,0,0,0,0,
0,0,1,0,0,0,0,0,
0,0,0,0,0,1,0,0,
0,0,0,1,0,0,0,0,
0,1,0,0,0,0,0,0,
0,0,0,0,0,0,0,1,
0,0,0,0,1,0,0,0,
0,0,0,0,0,0,1,0,
1,0,0,0,0,0,0,0,

Depth Limited Search

```
/**********************************************************
************************

Need to put value of N

Depth limited search

**********************************************************
************************/

public class DLS_queen

{

public static void main(String[] args) {

System.out.println("Hello World");

int N=4;

int[][] positions = new int[N][N];


boolean a = place(0,positions,5);


//printPos(positions);

}
```

```
public static boolean place(int queenNo,int[][] positions,int depthLimit){
  if(depthLimit==0){
       return false;
  }
  boolean val=false;
                                                    for(int row=0;row<positions[0].length;row++){//iterate row

      if(checkQueen(queenNo,positions,row)==true){
          //place queen
               positions[row][queenNo]=1; //column is queenNo

             if(queenNo==positions[0].length-1){
                 //end of recursion
                      printPos(positions);
                      positions[row][queenNo]=0;
                 return true;
             }
```

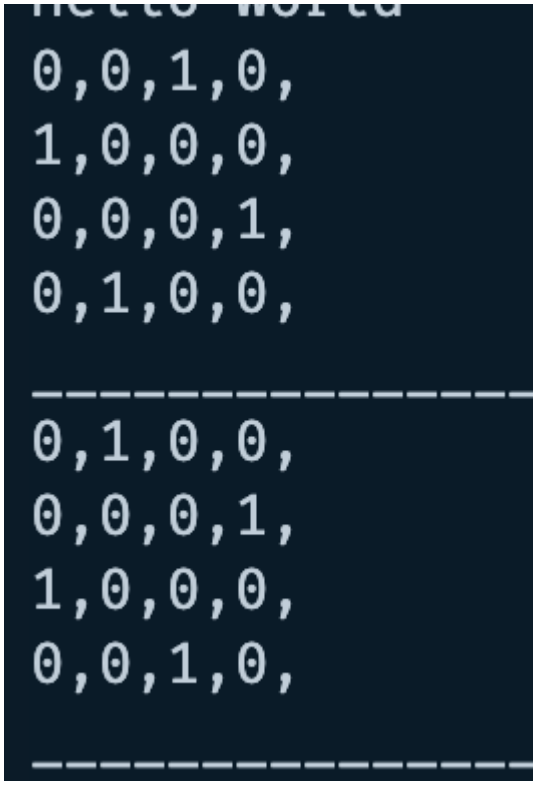
```
                //recur

if(place(queenNo+1,positions,depthLimit-1)==false){

                    //failed

                     positions[row][queenNo]=0;

                     continue; //look for next position

                }

                else{

                          val = true;//valid position

                          positions[row][queenNo]=0;

                          continue; //still look for next
position



                }

        }

   }

         if(val==true){

              return true;

         }

   //backtrack
```

```
  return false;

}

public static boolean checkQueen(int column,int [][]
positions, int row){



if(checkRow(column,positions,row)&&checkdiaLeft(column
,positions,row)&&checkdiaRight(column,positions,row)){

      return true;

  }

  return false;



}

public static boolean checkRow(int column,int [][]
positions, int row){

             for(int  i=0;i<positions[0].length;i++){
//iterate columns

          if(positions[row][i]==1){


              return false;

          }
```

```
        }

        return true;

    }

    public static boolean checkdiaLeft(int column,int
[][] positions, int row){

                for(int  i=0;i<positions[0].length;i++){
//iterate row

                for(int  j=0;j<positions[0].length;j++){
//iterate columns

           if(i-j==row-column){


              if(positions[i][j]==1 ){


                 return false;

            }

           }

        }


    }

     return true;
```

```
    }


    public static boolean checkdiaRight(int column,int
[][] positions, int row){

                for(int  i=0;i<positions[0].length;i++){
//iterate row

                for(int  j=0;j<positions[0].length;j++){
//iterate columns

           if(i+j==row+column){


              if(positions[i][j]==1 ){


                return false;

             }

           }

        }


    }

     return true;

    }
```

```
   public static void printPos(int [][] positions){

              for(int  i=0;i<positions[0].length;i++){
//iterate columns

               for(int j=0;j<positions[0].length;j++){
//iterate rows

           System.out.print(positions[i][j]+",");

       }

       System.out.println();

       }

       System.out.println("_____________________");

   }

}
```


0,0,1,0,
1,0,0,0,
0,0,0,1,
0,1,0,0,
0,1,0,0,
0,0,0,1,
1,0,0,0,
0,0,1,0,

Breadth First Search

```
/*************************************************************
************************

Need to put value of N
Depth first search

*************************************************************
************************/
import java.util.HashSet;
import java.util.List;
import java.util.ArrayList;

public class BFS_queen
{
public static void main(String[] args) {
System.out.println("Hello World");
int N=8;
int[][] initpositions = new int[N][N];
boolean [] terminated = new boolean[N];
ArrayList<int[][]>      listOfPositions      =      new
ArrayList<>();
ArrayList<int[][]>     newConfigurations      =      new
ArrayList<>();
listOfPositions.add(initpositions);
for(int i=0;i<4;i++){
for (int[][] positions : listOfPositions) {
```

```
for(int queen=i;queen<N;queen++){
   for(int row=0;row<N;row++){//iterate row
 if( checkQueen(queen,positions,row)==true){
     int [] [] positions_cpy = deepCopy(positions);
    positions_cpy[row][queen]=1; //column is queenNo
    // printPos(positions_cpy);
    newConfigurations.add(positions_cpy);
 }
   }
}
}
listOfPositions.clear();
listOfPositions.addAll(newConfigurations);
newConfigurations.clear();

}
        // Convert the ArrayList to a Set to remove
duplicates
                HashSet<int[][]>  uniqueSet  =  new
HashSet<>(listOfPositions);

       // Convert the Set back to an ArrayList
               ArrayList<int[][]>  uniqueList  =  new
ArrayList<>(uniqueSet);

for (int[][] positions : uniqueList) {
      printPos(positions);
   }
```

```
}

public static boolean checkQueen(int column,int [][]
positions, int row){


if(checkRow(column,positions,row)&&checkdiaLeft(column
,positions,row)&&checkdiaRight(column,positions,row)&&
checkCol(column,positions,row)){
      return true;
  }
  return false;

}
  // Helper method to create a deep copy of a 2D array
   private static int[][] deepCopy(int[][] original) {
        int rows = original.length;
        int cols = original[0].length;
        int[][] copy = new int[rows][cols];

        for (int i = 0; i < rows; i++) {
              System.arraycopy(original[i], 0, copy[i],
0, cols);
        }

        return copy;
   }

public static boolean checkRow(int column,int [][]
positions, int row){
```

```
                for(int  i=0;i<positions[0].length;i++){
//iterate columns
            if(positions[row][i]==1){

                return false;
            }
        }
        return true;
    }

    public static boolean checkCol(int column,int [][]
positions, int row){
                for(int  i=0;i<positions[0].length;i++){
//iterate row
            if(positions[i][column]==1){



                return false;
            }
        }
        return true;
    }
    public static boolean checkdiaLeft(int column,int
[][] positions, int row){
                for(int  i=0;i<positions[0].length;i++){
//iterate row
                for(int  j=0;j<positions[0].length;j++){
//iterate columns
           if(i-j==row-column){
```

```
            if(positions[i][j]==1 ){

              return false;
           }
          }
       }

   }
    return true;
   }

    public static boolean checkdiaRight(int column,int
[][] positions, int row){
              for(int  i=0;i<positions[0].length;i++){
//iterate row
              for(int  j=0;j<positions[0].length;j++){
//iterate columns
          if(i+j==row+column){

            if(positions[i][j]==1 ){

              return false;
           }
          }
       }

   }
    return true;
   }
```

```
   public static void printPos(int [][] positions){
                for(int  i=0;i<positions[0].length;i++){
//iterate columns
                 for(int j=0;j<positions[0].length;j++){
//iterate rows
             System.out.print(positions[i][j]+",");
        }
        System.out.println();
        }
        System.out.println("_____________________");
    }
}
```

```
__________________
0,0,0,0,0,0,1,0,
0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,1,
0,0,0,0,0,0,0,0,
0,0,0,0,1,0,0,0,
1,0,0,0,0,0,0,0,

__________________
0,0,0,1,0,0,0,0,
0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,
0,0,1,0,0,0,0,0,
0,0,0,0,0,0,0,0,
0,0,0,0,0,1,0,0,
1,0,0,0,0,0,0,0,

__________________
0,0,0,0,0,1,0,0,
0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,
0,1,0,0,0,0,0,0,
0,0,0,1,0,0,0,0,
0,0,0,0,0,0,1,0,

__________________
0,0,0,0,0,0,0,0,
0,0,0,1,0,0,0,0,
0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,1,
0,0,0,0,0,1,0,0,
0,1,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,
```

Post Lab Objective questions

1. Which search algorithm imposes a fixed depth limit on nodes?

a. Depth-limited search
b. Depth-first search
c. Iterative Deepening search
d. Only (a) and (b)
e. Only (a), (b) and (c).

Answer: a. Depth-limited search (DLS)

2. Optimality of BFS is

a. When all step costs are equal
b. When all step costs are unequal
c. When there is less number of nodes
d. Both a & c

Answer: d. Both a & c

Post Lab Subjective Questions:

1. Mention the criteria for the evaluation of search Algorithm.

Ans.

1. **Completeness:** Ability to find a solution when there is a solution (The algorithm should not miss any node before reaching any goal node/end of the state space).
2. **Optimality:** Ability to find the highest quality solution (The algorithm should find the goal nodes in shallower level first).
3. **Time complexity:** How long the algorithm takes to process the nodes. (The execution time of the algorithm/ CPU time)
4. **Space complexity:** How much memory is used by the algorithm to store nodes.
5. **Admissibility:** A search algorithm is admissible if it never overestimates the cost of reaching the goal state.
6. **Consistency**: A search algorithm is consistent if the estimated cost of reaching the goal state from a given state is less than or equal to the sum of the cost of reaching a neighboring state and the estimated cost of reaching the goal state from that neighboring state.

2. State the properties of BFS, DFS, DLS and IDS

Ans.

1. **Breadth-First Search (BFS)**:
   - Completeness: BFS is complete if the branching factor is finite.
   - Optimality: BFS is optimal if the cost of the path is a non-decreasing function of the depth of the node.

- Time complexity: The time complexity of BFS is O(b^d), where b is the branching factor and d is the depth of the shallowest goal node.
- Space complexity: The space complexity of BFS is O(b^d), where b is the branching factor and d is the depth of the shallowest goal node.
- Admissibility: BFS is admissible if the cost of the path is non-decreasing function of the depth of the node.
- Consistency: BFS is consistent if the cost of the path is non-decreasing function of the depth of the node.

2. **Depth-First Search (DFS)**:
   - Completeness: DFS is not complete if the search space is infinite or the depth of the goal node is infinite.
   - Optimality: DFS is not optimal if the cost of the path is not a non-decreasing function of the depth of the node.
   - Time complexity: The time complexity of DFS is O(b^m), where b is the branching factor and m is the maximum depth of the search tree.
   - Space complexity: The space complexity of DFS is O(bm), where b is the branching factor and m is the maximum depth of the search tree.
   - Admissibility: DFS is not admissible if the cost of the path is not a non-decreasing function of the depth of the node.
   - Consistency: DFS is not consistent if the cost of the path is not a non-decreasing function of the depth of the node.
3. **Depth-Limited Search (DLS)**:
   - Completeness: DLS is complete if the depth limit is greater than or equal to the depth of the shallowest goal node.
   - Optimality: DLS is not optimal if the cost of the path is not a non-decreasing function of the depth of the node.
   - Time complexity: The time complexity of DLS is O(b^l), where b is the branching factor and l is the depth limit.
   - Space complexity: The space complexity of DLS is O(bl), where b is the branching factor and l is the depth limit.
   - Admissibility: DLS is not admissible if the cost of the path is not a non-decreasing function of the depth of the node.
   - Consistency: DLS is not consistent if the cost of the path is not a non-decreasing function of the depth of the node.
4. **Iterative Deepening Search (IDS)**:
   - Completeness: IDS is complete if the branching factor is finite and the depth limit is infinite.
   - Optimality: IDS is optimal if the cost of the path is a non-decreasing function of the depth of the node.
   - Time complexity: The time complexity of IDS is O(b^d), where b is the branching factor and d is the depth of the shallowest goal node.

- Space complexity: The space complexity of IDS is O(bd), where b is the branching factor and d is the depth of the shallowest goal node.
- Admissibility: IDS is admissible if the cost of the path is non-decreasing function of the depth of the node.
- Consistency: IDS is consistent if the cost of the path is non-decreasing function of the depth of the node.

3. Explain why BFS is worst approach when the branching factor and solution depth in state-space tree is large (value =10 or more)

Ans.

Breadth-First Search (BFS) is a complete and optimal search algorithm, but it is not always the best choice for searching large state spaces. The main drawback of BFS is its memory requirement. Since BFS stores all the nodes at a given depth before moving on to the next level, it requires a lot of memory when the branching factor and solution depth in the state-space tree are large. In such cases, the space complexity of BFS becomes O(b^d), where b is the branching factor and d is the depth of the shallowest goal node . This can cause the algorithm to run out of memory and crash.

In contrast, Depth-First Search (DFS) has a much lower memory requirement than BFS, as it only stores the nodes on the current path from the root to the current node . However, DFS is not complete or optimal in all cases . Depth-Limited Search (DLS) and Iterative Deepening Search (IDS) are two variants of DFS that address some of its limitations .

To elaborate further, let's consider an example where we have a state-space tree with a branching factor of 10 and a solution depth of 10. In this case, the number of nodes at the last level of the tree would be 10^10, which is a very large number . If we use BFS to search this tree, we would need to store all these nodes in memory before we can move on to the next level. This would require a huge amount of memory, which may not be feasible on most computers. In contrast, DFS would only store the nodes on the current path from the root to the current node, which would require much less memory.

Therefore, BFS is not the best approach when the branching factor and solution depth in the state-space tree are large. DFS, DLS, and IDS are better alternatives in such cases, as they have lower memory requirements and can still find a solution to the problem