Project Report

on

HI-Q Game

Design and Analysis of Algorithms

Bachelor of Technology

in

Department of Computer Science and Engineering

By

Nihal Agarwal	2010030413
Jaideep Sharma	2010030374
T Venkata Sai Sathvik	2010030361
Shaik Abdul Shaan	2010030153

under the supervision of

Ms. P. Sree Lakshmi, MTech (Ph.D.)
Assistant Professor, CSE

Department of Computer Science and Engineering

K L University Hyderabad,

Aziz Nagar, Moinabad Road, Hyderabad – 500 075, Telangana, India.

March 2022

Declaration

The Project Report entitled "**HI-Q Game**" is a record of bonafide work of **Nihal Agarwal (2010030413), Jaideep Sharma (2010030374), T Venkata Sai Sathvik (2010030361), Shaik Abdul Shaan (2010030153)**, submitted as a requirement for the completion of the course **Design and Analysis of Algorithms** in the Department of Computer Science and Engineering to the K L University, Hyderabad. The results embodied in this report have not been copied from any other Departments/ University/ Institute.

NIHAL AGARWAL	2010030413
T VENKATA SAI SATHVIK	2010030361
JAIDEEP SHARMA	2010030374
SHAIK ABDUL SHAAN	2010030153

Certificate

This is to certify that the Project Report entitled "HI-Q Game" is being submitted by Nihal Agarwal (2010030413), Jaideep Sharma (2010030374), T Venkata Sai Sathvik (2010030361), Shaik Abdul Shaan (2010030153) , as a requirement for the completion of the course **Design and Analysis of Algorithms** in the Department of Computer Science and Engineering, K L University, Hyderabad is a record of bonafide work carried out under our guidance and supervision.

The results embodied in this report have not been copied from any other departments/ University/Institute.

Signature of the Supervisor

Ms. P. Sree Lakshmi, MTech (Ph.D.)

Assistant Professor, CSE

Signature of the HOD

Signature of the External Examine

ACKNOWLEDGEMENT

First, I would like to thank our beloved Founder and chairman, of Koneru Lakshmaiah University for giving us this opportunity to complete our project within the University in the guidance of our faculty.

I am highly indebted to our Principal, **Dr. L Koteswara Rao** who has been constantly pushing us and providing us opportunities for all the curricular activities undertaken by us.

I would like to thank my Head of department **Dr. Chiranjeevi Manike** for his exemplary guidance, monitoring and constant support through the course of the project. We thank **Ms. P. Sree Lakshmi, Assistant Professor** of our department who has supported throughout this project holding a position of supervisor.

I am extremely grateful to all the teaching and non-teaching staff of our department without whom we won't have made this project a reality. We would like to extend our sincere thanks to our parents who supported us making this project a grand success.

ABSTRACT

A HI-Q game also known as Peg Solitaire, is a board game for one player which involves movement of pegs/ marbles on a board with holes. The game is known as solitaire in Britain and as peg solitaire in US, solitaire is commonly known as patience. It is called as Brainvita in India. For solving this game there are many solutions available given by different people in different methods. The standard game fills the entire board with pegs except for the central hole. The objective is, making valid moves, to empty the entire board except for a solitary peg in the central hole.

keywords: HI-Q, Solitaire, patience, Brainvita, central hole.

TABLE OF CONTENTS

List Of Figures

1. INTRODUCTION

HI-Q game is one of the applications for ‘minimum jumps to reach the end’ algorithm. The user wants to know number of optimal jumps and the path to reach the end faster than the person he is competing with. Peg solitaire is a board game for one player. It is played on a board which has positions in a geometric arrangement. Each position can be either free or occupied. Usually, the positions are marked by indentations that can be occupied by a peg or a marble. We need to first describe the initial state of the board, then define the desired state of the board and pegs, try to find out conditions that initially fill the whole board according to the conditions and then apply an algorithm that apply transformation until desired state is reached.

2. LITERATURE SURVEY

Interactive Mathematics Miscellany and Puzzles, authored by Bogomolny, Alexander; published as Peg Solitaire and Graph Theory The main aim of the theory is to find the possible ways of moves., It shows the appropriate path while moving from one position to another position.

Mathematics and Brainvita, authored at word press, published as Mathematics and Brainvita , It analyses the solving technique of Brainvita game. It concludes that the player has to end up in 5 positions, and then it gives number of positions the final peg ends up.

A solitaire game and its relation to a finite field, authored by N. G. de Bruijn, published as Solitaire game , it finds the optimal solution using the valid moves. It has concluded that, if we want to end the game, if we have to end up at (0,0), then the positions such as (0,0), (0,2), (2,0), (-2,0), (0,-2) should not be killed.

3. HARDWARE AND SOFTWARE REQUIREMENTS

3.1 **HARDWARE REQUIREMENTS**

1. 1. Intel(R) Core(TM) i5-10300H CPU @ 2.50GHz 2.50 GHz
2. 8.00 GB RAM or higher.
3. 64 Bit operating system or higher.
4. 1 TB Hard free drive space.

3.2 **SOFTWARE REQUIREMENTS**

1. Operating System: Windows 10
2. Web Browser: Brave/ Google Chrome
3. Java programming language
4. Java IDE for java program

4. FUNCTIONAL AND NON-FUNCTIONAL REQUIREMENTS

4.1 PEG SOLITAIRE

It is also called as solo Noble or solitaire; it is a board game for one player involving movement of pegs or marble on a board with holes. It is called as Brainvita in India. The standard game fills the entire board with pegs except the central hole. The objective is, making valid moves, to empty the entire board except for a solitary peg in the central hole.

There are two traditional boards:-

1. English solitaire board and
2. European peg solitaire board.

Fig. 4.1 Traditional Boards

4.2 PLAYING METHOD

A move is valid if it jumps a peg orthogonally over an adjacent peg into a hole two positions away and then to remove the jumped peg.

4.3 VALID MOVES

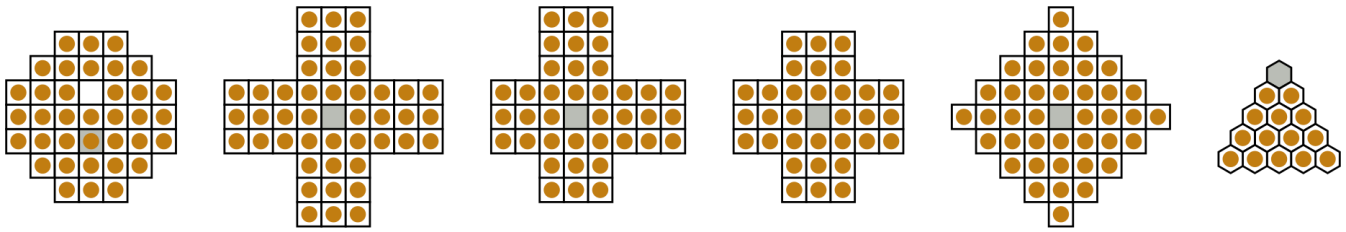
In the diagrams which follow, · indicates a peg in a hole, * emboldened indicates the peg to be moved, and **o** indicates an empty hole. A blue ¤ is the hole the current peg moved from; a red * is the final position of that peg, a red **o** is the hole of the peg that was jumped and removed.

Thus, valid moves in each of the four orthogonal directions are:

1. * · o → ¤ **o** * *Jump to right*
2. **o** · * → * **o** ¤ *Jump to left*
3. * ¤
   · → o *Jump down*
   o *
4. o *
   · → **o** *Jump up*
   * ¤

4.4 **PROJECT AREA**

The proposed solution is to find an optimal solution for solving the HI-Q game. We have to find a solution which will be following the valid moves mentioned in **4.3** . So, here we are with an optimal solution in java programming language using dynamic programming approach under memotization and tabulation methods. It uses different Collection methods for different readymade predefined classes and objects to use it for easier accessibility.



1. 2. 3. 4. 5. 6.

Peg solitaire game board shapes:

1. French (European) style, 37 holes, 17th century.
2. J. C. Wiegleb, 1779, Germany, 45 holes.
3. Asymmetrical 3-3-2-2 as described by George Bell, 20th century.
4. English style (standard), 33 holes.
5. Diamond, 41 holes.
6. Triangular, 15 holes.

Grey = the hole for the survivor.

5. PROPOSED SYSTEM

5.1 pegSolataire ALGORITHM

we consider a puzzle commonly known as HI-Q game in this project. The board of the puzzle commonly consists of 33 points arranged as in the picture-1 with all, but one point placed with tokens. These special points, the center of the board, is named the end point or goal. A move consists of jumping one token over an adjacent one onto an empty point. When a token is jumped it is removed from the board. The objective of the puzzle is to make the board to a position with just one token on the goal. Thus, a sequence of jumps which leads the board consisting of no tokens, but the goal will be an answer for the puzzle. The puzzle will be consisting of the initial position on the extended board of the size n*n, and a goal is also given on which only one token will finally be placed. We show that the problem to determine whether there is an answer for a given generalized HI-Q is NP-Complete.

Algorithm **pegSolataire**{

if there is only n=1 peg left,

if it is in center, return indicating we found a solution, otherwise, we did not;

if this board is flagged as congruent to one that has been determined to not lead a solution,

return indicating, we did not find one, otherwise,

for each possible jump,

modify the board to account

for the jump, and call the backtracking algorithm recursively using memotization technique,

where if the algorithm indicates success, record the jump and

return indicating, we found a solution,

otherwise, reset the board to the state it was in prior to making a jump;

and the loop will only finish

if none of the possible jumps led to a solution, so flag that

this board has been found not to find a solution and

return indicating, we did not find one.

}

6. IMPLEMENTATION

6.1 **ARRAYLIST IN JAVA**

Java **ArrayList** class uses a dynamic array for storing the elements. It is like an array, but there is no size limit. We can add or remove elements anytime. So, it is much more flexible than the traditional array. It is found in the java.util package.

In an ArrayList the elements can be repeated, it uses list interface by which we can use all the methods of list interface. It also maintains the insertion order internally.

6.2 **ARRAY IN JAVA**

Java array is an object which contains elements of a similar data type. Additionally, the elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.

ADVANTAGES

1. **Code Optimization:** It makes the code optimized, we can retrieve or sort the data efficiently.
2. **Random access:** We can get any data located at an index position.

DISADVANTAGES

1. **Size Limit:** We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.

6.3 **COLLECTIONS IN JAVA**

The Collection in Java is a framework that provides an architecture to store and manipulate a4 group of objects. **Java Collections** can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

6.4 **CODE EXPLANATION**

This method contains three attributes called as from, hole and to. From attribute shows the hole from which the marble is jumping, Hole attribute shows from which marble is the marble jumping and to attribute shows to which hole the marble is jumping. The toString method prints the Possibilities of all the marbles jumping from one hole to another hole till end result. The compareto method compares the existing move grid value with next move grid value for better and faster optimal solution in a best time complexity. It will compare one hole with another hole for the marble to jump. Then we have arraylist which is a 2d array of the input from user as a grid. We have an array of 1d array for the moveslist and arraylist of 2d arraylist to store the Unsuccessful moves of the grid which is already a 2d array.

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;

public class Pegsolataire {
  class Move implements Comparable<Move>{
    int from;   // from where the marble is being moved
    int hole; // to which hole its being moved
    int to;    // to which hole its being moved
    Move(int from, int hole, int to){
      // constructor
      this.from = from;
      this.hole = hole;
      this.to = to;
    }
    // to print it we can use toString method
    public String toString() {
      return ("("+ this.from + "," + this.hole + "," + this.to + ")");
    }
    @Override
    public int compareTo(Move m) {
      return Integer.valueOf(this.from).compareTo(Integer.valueOf(m.from));   // comparing the values and sorting in ascedning order
    }
  }
  ArrayList<ArrayList<Integer>> grid;   // we store the array of the arrays for the board in the form of 1's and -1's
  ArrayList<Move> movesList;        // movesList stores from one location to another location.
  ArrayList<ArrayList<ArrayList<Integer>>> unsuccessfullGrid;   // we are going to store the grids which has arraylist of arraylist inside an another arraylist
```

Then we have pegSolataire constructor which initialises the grid which the input is passed from the main. The printOutput method calls the toString method above to print the possibilities of the hole from which it is jumping to the hole which it is jumping. The displayGrid method calls the forEach method for each input given from the user and if its -1 then the - is put and if it's not -1 then the toString method is called for the grid occupancy.

```
 Pegsolataire(ArrayList<ArrayList<Integer>> grid){
  // constructor function for the initialise the pegSolataire function
  this.grid = grid;          // it initialises the grid given by the hardcoded
  movesList = new ArrayList<>();          // initialises the new array list for moveslist
  unsuccessfullGrid = new ArrayList<>();
 }
 // we will write the method to print all the moves it has been performed when called.
 private void printOutput() {
  for (Move move: movesList) {
    System.out.println(move.toString());     // we had already created the move toString to print
output
   }
 }
  private void displayGrid() {
  //to print the entire grid
  for (ArrayList<Integer> line : grid) {
   for (int i: line) {
    if (i == -1 ) {
     System.out.print("- ");      // whenever there is no grid value print the  -
    }
    else {
     System.out.print(Integer.toString(i)+ " ");
    }
   }
   System.out.println();
  }
 }
```

The makeMove method takes the input as move and calls its own class Move and it gets the input from the column as it divides the move input/7 to indicate the row number, it gets the row number from get method and then sets the pointer to the column number by modulo division by 7 and sets the value to 1. And remaining rows are set to 0. As it is described from the hole which it is jumping is set to 0 and the hole from which it is jumping is set to 0 and the hole to which it is jumping is set as 1. The undoMove method also does the same thing as makeMove method but undoes the false moves in the from hole, in hole and to hole method and removes the added move in the movesList.

```
private void makeMove(Move move) {
   grid.get(move.from/7).set(move.from % 7, 0);    // we will get the number from the row number using the /7 and the column number using the %
   grid.get(move.hole/7).set(move.hole % 7, 0);
   grid.get(move.to/7).set(move.to % 7, 1);
   movesList.add(move);  // we add it to the arraylist of the moveList
  }
  private void UndoMove(Move move) {
   grid.get(move.from/7).set(move.from % 7, 1);    // we will undo the marbel jumped on
   grid.get(move.hole/7).set(move.hole % 7, 1);
   grid.get(move.to/7).set(move.to % 7, 0);
   movesList.remove(movesList.size()-1);  // we will remove the last number from the grid. similar to pop
  }
```

The deepCopy method takes the arraylist of arraylist as an input by initialising the return statement as arraylist of arraylist in the methods. It calls each input from the main method after computing the answer and then adds the remaining grid to the unsuccessful grid and also copies each iteration into the new grid as a copy, because whenever other methods call the deepCopy method it can have a look at the previous iteration and then complete the possible move based on it. The getCount method gets the count of number of 1 by pointing at the row then at the column and increases the counter based on it as to verify the method in the end of having only one possible 1 in the end. The solve method is the heart of the problem to verify the problem, if there is unsuccessful grid present same as the input then the input might be wrong, or the computation is not done. If the count is 1 and the middle row has 1 as the output after computation, then it will call the display and print methods or else it will again sort all the moves in the moveslist and call the compute possibilites method to verify the method once again which calls the undoMove and makeMove method.

```
  private ArrayList<Move> computePossibilities() {
   // return the possibilities the list of moves
   ArrayList<Move> possibilites = new ArrayList<>();
   // we need to check the marbles that are two lengths difference or available to left right or top or bottom
   for(int i = 0 ; i<grid.size(); i++) {
    for (int j = 0; j<grid.get(i).size(); j++) {
     if (grid.get(i).get(j) == 0) {
      if ((i-2) >= 0) {
       if ((grid.get(i-2).get(j) == 1) && (grid.get(i-1).get(j) == 1)) {
        possibilites.add(new Move((i-2)*7+j,((i-1)*7+j), (i*7+j)));
       }
      }
```

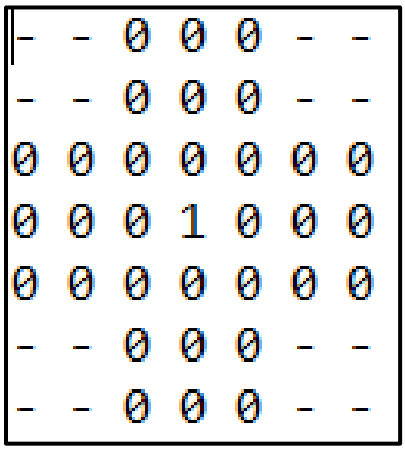
```
      if ((j-2) >= 0) {
        if ((grid.get(i).get(j-2) == 1) && (grid.get(i).get(j-1) ==1))
        {
          possibilites.add(new Move(i*7 + j-2 , i*7 + j-1, i*7 +j));
        }
       }
       if (i+2 <= 6){
        if ((grid.get(i+2).get(j) == 1) && (grid.get(i+1).get(j) == 1)) {
          possibilites.add(new Move((i+2)*7+j,(i+1)*7+j, i*7+j));
        }
       }
      if (j+2 <= 6) {
        if((grid.get(i).get(j+2) == 1) && grid.get(i).get(j+1) == 1) {
          possibilites.add(new Move(i*7+j+2, i*7+j+1, i*7+j));
        }
       }
      }
    }
   }
   return possibilites;
}
private ArrayList<ArrayList<Integer>> deepCopy(ArrayList<ArrayList<Integer>> input){
          ArrayList<ArrayList<Integer>> newGrid=new ArrayList<>();
          for (ArrayList<Integer> line:input)
          {
                    ArrayList<Integer> cpLine=new ArrayList<>();
                    for(Integer i:line) {
                              cpLine.add(Integer.valueOf(i));
                    }
                    newGrid.add(cpLine);
          }
          return newGrid;
}
private int getCount() {
  int count = 0;
  for (int i =0 ; i<grid.size();i++) {
    for(int j =0 ; j<grid.get(i).size(); j++) {
      if (grid.get(i).get(j) == 1) {
        count ++;
      }
    }
  }
  return count;
}
```

```
public boolean solve() {
   if (unsuccessfullGrid.contains(grid)) {
     return false;
    }
   if (getCount() == 1 && grid.get(3).get(3) == 1) {
     displayGrid();
     printOutput();
     return true;
    }
   else {
     ArrayList<Move> moves  = computePossibilities();
     Collections.sort(moves);

     for (Move move: moves) {
       makeMove(move);
       if (solve())  {
         return true;
       }
       else {
         UndoMove(move);

       }
     }
    }
   if (!unsuccessfullGrid.contains(grid)){
     unsuccessfullGrid.add(deepCopy(grid));
   }
   return false;
  }
  public static void main(String args[]) {
    ArrayList<ArrayList<Integer>> grid = new ArrayList<>();
    Integer[] line1 = {-1,-1,1,1,1,-1,-1};
    Integer[] line2 = {-1,-1,1,1,1,-1,-1};
    Integer[] line3 = {1,1,1,1,1,1,1};
    Integer[] line4 = {1,1,1,0,1,1,1};
    Integer[] line5 = {1,1,1,1,1,1,1};
    Integer[] line6 = {-1,-1,1,1,1,-1,-1};
    Integer[] line7 = {-1,-1,1,1,1,-1,-1};
    grid.add(new ArrayList<Integer>(Arrays.asList(line1)));
    grid.add(new ArrayList<Integer>(Arrays.asList(line2)));
    grid.add(new ArrayList<Integer>(Arrays.asList(line3)));
    grid.add(new ArrayList<Integer>(Arrays.asList(line4)));
    grid.add(new ArrayList<Integer>(Arrays.asList(line5)));
    grid.add(new ArrayList<Integer>(Arrays.asList(line6)));
    grid.add(new ArrayList<Integer>(Arrays.asList(line7)));
    new Pegsolataire(grid).solve();
  }
}
```

7. RESULTS DISCUSSION



0 → Empty Hole

1 → Filled Hole (here center)

━ → No use (array completion)

Fig 7.1. The board after moves

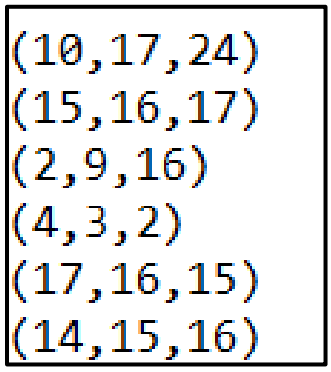


Fig 7.2. Moves-1

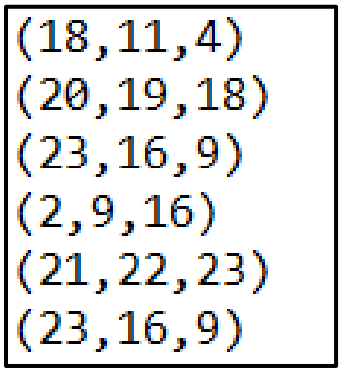


Fig 7.3. Moves-2

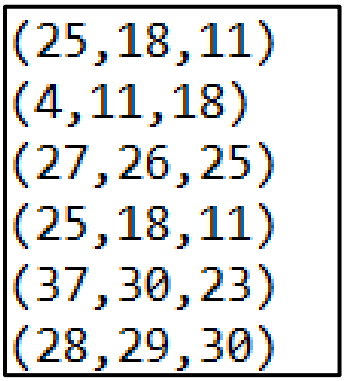


Fig 7.4. Moves-3

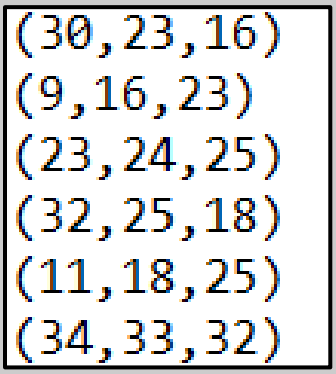


Fig 7.5. Moves-4

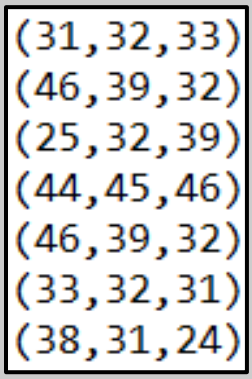


Fig 7.6. Moves-5

8. CONCLUSION AND FUTURE WORK

8.1 **CONCLUSION**

In the proposed system, we are generating an optimal solution using memotization and dynamic approach. the main outcome of the problem is getting a filled hole at the center using the marbles by following the valid moves as mentioned above. In order to get the output, we have used arraylist in java, collections and arrays in java. The output generated shows the design of board after solving the game, and also shows the moves made by the user in order to get the desired solution.

8.2 **FUTURE WORK**

In future, the code can be changed to a User Interface model where the player can easily move the marbles on a digital board. And get the desired solution.

9. REFERENCES

1. K. Atkinson and A. Bogomolny, The discrete Galerkin method for integral equations, Math of Comp 48 (1987),595-616 and S11-S15.
2. E. R. Berlekamp, J. H. Conway, R. K. Guy, Winning Ways for Your Mathematical Plays, v2, Academic Press, 1982.
3. A. Bialostocki, An Application of Elementary Group Theory to Central Solitaire, The College Mathematics Journal, v 29, n 3, May 1998, 208-212.
4. White Pixels (24 October 2017), Peg Solitaire: Easy to remember symmetrical solution (video), Youtube, archived from the original on 2021-12-11
5. Play Multiple Versions of Peg Solitaire including English, European, Triangular, Hexagonal, Propeller, Minimum, 4Holes, 5Holes, Easy Pinwheel, Banzai7, Megaphone, Owl, Star and Arrow at pegsolitaire.org