Hill Climbing Algorithm – N-Queen Problem

PROJECT DOCUMENTATION REPORT

PROGRAMMING PROJECT 2

ITCS 6150 - Intelligent Systems

DEPARTMENT OF COMPUTER SCIENCE

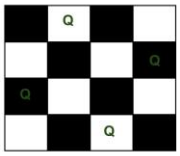
Noor Zahara

801097945

Table of Contents

PROBLEM FORMULATION

Thee N Queen is the problem of placing N chess queens on an N*N chessboard so that no two queens attack each other. For example, following is a solution for 4 Queen problem.



In this project, Hill Climbing Algorithm is implemented to solve N Queen problem.
It is an iterative algorithm that starts with an arbitrary solution to a problem, then attempts to find a better solution by making an incremental change to the solution. If the change produces a better solution, another incremental change is made to the new solution, and so on until no further improvements can be found.

Three variants of Hill climbing algorithm are implemented:

1. **Steepest-Ascent Hill Climbing -**In this algorithm, we consider all possible states from the current state and then pick the best one as successor, unlike in the simple hill climbing technique.
2. **Hill Climbing with sideways** – Here if the path cost of the parent and the child state is the same, search continues till the specified number of steps and then terminates.
3. **Random Restart hill climbing with and without sideways** – If the sideways move does not produce the goal state, the chessboard is randomly reconfigured, and the algorithm starts afresh.

A heuristic h(n) is calculated for each state which is the number of queens attacking each queen considered column wise.

PROGRAM STRUCTURE

2.1 Functions and Procedures

The code implements the variants of Hill climbing algorithm to solve N Queen problem. It takes the size of the N Queen board from the user and randomly initializes the board.

generateAdjacentNodes() : The function generates child nodes of the given parent node by moving the queens up and down column wise.
steepestAscentHillClimbSearch() : The function implements steepest ascent hill climbing algorithm.
sidewaysHillClimbSearch() : The function implements hill climbing algorithm with sideways moves upto 100 steps.
getRandomConfiguration() : The function randomly sets the initial state of the N Queen board.
checkForDuplicateStates(Node currentNode, Node parentNode) : The function checks if the current state is a duplicate of some previously generated state.
getQueenIndexToBeMoved(List<Integer> tempParentState, int colIndex) : The function returns the index of the queen that needs to be moved from the parent state to generate its child states.
calculateStateHeuristic(List<Integer> state) : The function calculates the heuristic value of a state i.e. number of conflicting queens for the given state.
getQueenIndexForTheState(List<Integer> state, int index) : The function checks if the queen is present in the given state at the given index and returns the index else returns the index of the next queen present.
checkEquality(List<T> listA, List<T> listB) : The function checks if the given 2 lists have equal elements.
findIndex(List<T> listA, T element) : The function returns the index of the given element in the given list.

2.2 Global and Local variables

Global variables:

boardSize - int

initialState - List<Integer> of size boardSize*boardSize

interationNumber – int, initialized to 300

interationLimit – int, set to 1000

queue – PriorityQueue, initialized to null

sidewaysCountLimit – int, set to 100

initialSidewaysCount - int, initialized to 0

Local variables:

inputState – Scanner object reference variable

outerIterationLoop – int, loop variable to loop through the iterations

innerIterationLoop – int, loop variable to run the algorithm for outerIterationLoop times

totalSuccessCount – int, increments for every successful goal state reached in every iteration.

totalDepthForSuccessfulState – int, stores the total depth of all the successful goal state for each iteration

totalDepthForFailureState – int, stores the total depth of all the failed goal state for each iteration

goalNode – Node, the goal node reached after the algorithm is applied.

successRate – int, gives the total success rate after the end of each iteration

averageDepthOfSuccessfulState – int, average of totalDepthForSuccessfulState

averageDepthOfFailureState – int, average of totalDepthForSuccessfulState

pathCost – int, stores the heuristic cost of each parent state

adjacentNode – Node, stores the child node generated

bestSuccessor – Node, stores the node of the best successor chosen

randomColumnList – List<Integer>, stores the random column index to set the initial state of the board

totalNumberOfRestarts – int, stores the index of the current state tile value with respect to goal state

averageNumberOfRestarts – int, stores average of totalNumberOfRestarts

heuristicCount – int, stores the heuristic value of each state

parentState – List<Integer>, stores the state of the parent node

depth – int, stores the depth of the node generated

adjacentNodes – List<Node> , stores all the child nodes of the given parent node

2.3 Java Code

```
package IS_Project2;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Random;
import java.util.Scanner;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

public class SteepestAscentHC {

	static int boardSize;
	static List<Integer> initialState = new
ArrayList<Integer>(boardSize*boardSize);
	static int iterationNumber = 300;
	static int iterationLimit = 1000;
	static PriorityQueue queue = null;

	public static void main(String[] args) {
		System.out.println("Enter the size of the N Queen problem : \n");
		Scanner inputState = new Scanner(System.in);
		boardSize = inputState.nextInt();
		if (boardSize < 4) {
			System.out.println("N Queen problem cannot b" + "e solved for the
size: " + boardSize);
			return;
		}
		for (int outerIterationLoop = iterationNumber; outerIterationLoop <=
iterationLimit; outerIterationLoop = outerIterationLoop + 100) {
			int totalSuccessCount = 0;
			int totalDepthForSuccessfulState = 0;
			int totalDepthForFailureState = 0;
			System.out.println("Iteration Number : " + outerIterationLoop);
			for (int innerIterationLoop = 0; innerIterationLoop <
outerIterationLoop; innerIterationLoop++) {
				initialState = getRandomConfiguration();
//				System.out.println("Initial State : \n");
//				int index = 0;
//				System.out.print("[");
//				while (index < initialState.size()) {
//					for (int i = 0; i < boardSize; i++) {
//						System.out.print(initialState.get(index) + "\t");
//						index++;
//					}
//					System.out.println("\n");
//				}
//				System.out.print("]");
```

```
 queue = new PriorityQueue();
                        Node goalNode = steepestAscentHillClimbSearch(new
Node(initialState));
                        if (goalNode.getPathCost() == 0) {
                              totalSuccessCount++;
                              totalDepthForSuccessfulState =
totalDepthForSuccessfulState + goalNode.getDepth();
                        } else {
                              totalDepthForFailureState =
totalDepthForFailureState + goalNode.getDepth();
                        }
//                  System.out.println("Goal Reached : " + isGoal);
//                  printTracePath(goalNode);
//
      System.out.println("***********************************************");
                  }
                  System.out.println("Number: " + totalSuccessCount);
                  float successRate = ((float) totalSuccessCount /
outerIterationLoop) * 100;
                  float averageDepthOfSuccessfulState = ((float)
totalDepthForSuccessfulState / totalSuccessCount);
                  float averageDepthOfFailureState = ((float)
totalDepthForFailureState / (outerIterationLoop - totalSuccessCount));
                  System.out.println("Success Rate :" + successRate);
                  System.out.println("Failure Rate:" + (100 - successRate));
                  System.out.println("Average steps - sucess:" +
averageDepthOfSuccessfulState);
                  System.out.println("Average steps - fail :" +
averageDepthOfFailureState);
                  System.out.println("************************************");
            }
      }
      /**
       * The function implements steepest ascent hill climbing algorithm by choosing
       * the best successor.
       *
       * @param initialState
       * @return
       */
      public static Node steepestAscentHillClimbSearch(Node initialState) {
            int pathCost =
initialState.calculateStateHeuristic(initialState.getState());
            initialState.setPathCost(pathCost);
            initialState.generateAdjacentNodes();
            for (int adjacentNodeindex = 0; adjacentNodeindex <
initialState.getAdjacentNodes().size(); adjacentNodeindex++) {
                  Node adjacentNode =
initialState.getAdjacentNodes().get(adjacentNodeindex);
                  queue.push(adjacentNode, adjacentNode.getPathCost());
            }
            Node bestSuccessor = (Node) queue.pop();
            if (initialState.getPathCost() <= bestSuccessor.getPathCost()) {
                  return initialState;
            }
            return steepestAscentHillClimbSearch(bestSuccessor);
      }
```

```
/**
	 * This function prints the path from the goal reached to the initial state
	 * along with the path cost
	 *
	 * @param goalNode
	 */
	public static void printTracePath(Node goalNode) {
		if (goalNode.getParentNode() != null) {
			printTracePath(goalNode.getParentNode());
		}
		List<Integer> goalState = goalNode.getState();
		int index = 0;
		System.out.println("Path Cost (h(n)):" + goalNode.getPathCost());
		System.out.println("Depth d :" + goalNode.getDepth());
		System.out.println("[");
		while (index < goalState.size()) {
			for (int i = 0; i < boardSize; i++) {
				System.out.print(goalState.get(index) + "\t");
				index++;
			}

			System.out.print("\n");
		}
		System.out.println("]");
		System.out.println("**********************");
		return;
	}

	/**
	 * The function generates a random initial state.
	 *
	 * @return
	 */
	public static List<Integer> getRandomConfiguration() {
		List<Integer> randomColumnList = Arrays
				.stream(IntStream.generate(() -> new
Random().nextInt(boardSize)).limit(boardSize).toArray()).boxed()
				.collect(Collectors.toList());
		int randomListindex = 0;
		List<Integer> initialState = IntStream.of(new int[boardSize *
boardSize]).boxed().collect(Collectors.toList());
		while (randomListindex < boardSize) {
			int randomIndexTobeSet = randomColumnList.get(randomListindex);
			int tempRandomListIndex = randomListindex;
			for (int colIndex = 0; colIndex < boardSize; colIndex++) {
				tempRandomListIndex = (colIndex == 0) ?
tempRandomListIndex : tempRandomListIndex + boardSize;
				if (colIndex == randomIndexTobeSet) {
					initialState.set(tempRandomListIndex, 1);
				}
			}
			randomListindex++;
		}
		return initialState;
	}

}
```

```
package IS_Project2;

import java.util.Scanner;

public class SidewaysHC extends SteepestAscentHC {

	static int sidewaysCountLimit = 100;
	static int initialSidewaysCount = 0;
	static PriorityQueue queue = null;

	public static void main(String[] args) {
		System.out.println("Enter the size of the N Queen problem : \n");
		Scanner inputState = new Scanner(System.in);
		boardSize = inputState.nextInt();
		if (boardSize < 4) {
			System.out.println("N Queen problem cannot be solved for the
size: " + boardSize);
			return;
		}
		for (int outerIterationLoop = iterationNumber; outerIterationLoop <=
iterationLimit; outerIterationLoop = outerIterationLoop
				+ 100) {
			int totalSuccessCount = 0;
			int totalDepthForSuccessfulState = 0;
			int totalDepthForFailureState = 0;
			System.out.println("Iteration Number : " + outerIterationLoop);
			for (int innerIterationLoop = 0; innerIterationLoop <
outerIterationLoop; innerIterationLoop++) {
				initialState = getRandomConfiguration();
//				System.out.println("Initial State : \n");
//				int index = 0;
//				System.out.print("[");
//				while (index < initialState.size()) {
//					for (int i = 0; i < boardSize; i++) {
//						System.out.print(initialState.get(index) +
"\t");
//						index++;
//					}
//					System.out.println("\n");
//				}
//			System.out.print("]");

	initialSidewaysCount = 0;
				queue = new PriorityQueue();
				Node goalNode = sidewaysHillClimbSearch(new
Node(initialState));
				// System.out.println("Goal Reached :" + isGoal);
				if (goalNode.getPathCost() == 0) {
					++totalSuccessCount;
					totalDepthForSuccessfulState =
totalDepthForSuccessfulState + goalNode.getDepth();
				} else {
					totalDepthForFailureState =
totalDepthForFailureState + goalNode.getDepth();
				}
				// printTracePath(goalNode);
}
```

```
float successRate = ((float) totalSuccessCount / outerIterationLoop) * 100;
                    float averageDepthOfSuccessfulState = ((float)
totalDepthForSuccessfulState / totalSuccessCount);
                    float averageDepthOfFailureState = ((float)
totalDepthForFailureState
                                    / (outerIterationLoop - totalSuccessCount));
                    System.out.println("Success Rate :" + successRate);
                    System.out.println("Failure Rate:" + (100 - successRate));
                    System.out.println("Average steps - sucess:" +
averageDepthOfSuccessfulState);
                    System.out.println("Average steps - fail :" +
averageDepthOfFailureState);
                    System.out.println("*************************************");
             }
      }
      /**
       * The function implements hill climbing with sideways move upto 100 steps.
       *
       * @param initialState
       * @return
       */
      public static Node sidewaysHillClimbSearch(Node initialState) {
             int pathCost =
initialState.calculateStateHeuristic(initialState.getState());
             initialState.setPathCost(pathCost);
             initialState.generateAdjacentNodes();
             for (int index = 0; index < initialState.getAdjacentNodes().size();
index++) {
                    Node adjacentNode = initialState.getAdjacentNodes().get(index);
                    if (!checkForDuplicateStates(adjacentNode, initialState)) {
                           queue.push(adjacentNode, adjacentNode.getPathCost());
                    }
             }
             Node highestPriorityNode = (Node) queue.pop();
             if ((initialSidewaysCount > sidewaysCountLimit)
                           && (initialState.getPathCost() ==
highestPriorityNode.getPathCost())) {
                    return highestPriorityNode;
             }
             if ((initialSidewaysCount <= sidewaysCountLimit)
                           && (initialState.getPathCost() ==
highestPriorityNode.getPathCost())) {
                    initialSidewaysCount++;
                    return sidewaysHillClimbSearch(highestPriorityNode);
             }
             if (initialState.getPathCost() < highestPriorityNode.getPathCost()) {
                    return initialState;
             }
             initialSidewaysCount = 0;
             return sidewaysHillClimbSearch(highestPriorityNode);
      }
```

```
/**
	 * The function checks if the current Node was already generated previously.
	 *
	 * @param currentNode
	 * @param parentNode
	 * @return
	 */
	private static boolean checkForDuplicateStates(Node currentNode, Node
parentNode) {
		if (parentNode == null) {
			return false;
		}
		if (currentNode.getState().equals(parentNode.getState())) {
			return true;
		}
		return checkForDuplicateStates(currentNode, parentNode.getParentNode());
	}

}
package IS_Project2;

import java.util.Scanner;

public class RandomRestartWithSidewaysHC extends SidewaysHC {

	/**
	 * Hill climbing with random restart is been implemented allowing sideways
move.
	 *
	 */
	public static void main(String[] args) {
		System.out.println("Enter the size of the N Queen problem : \n");
		Scanner inputState = new Scanner(System.in);
		boardSize = inputState.nextInt();
		if (boardSize < 4) {
			System.out.println("N Queen problem cannot be solved for the
size: " + boardSize);
			return;
		}
		for (int outerIterationLoop = iterationNumber; outerIterationLoop <=
iterationLimit; outerIterationLoop = outerIterationLoop
				+ 100) {
			int totalNumberOfRestarts = 0;
			int totalDepthForFailureState = 0;
			System.out.println("Iteration Number :" + outerIterationLoop);
	for (int innerIterationLoop = 0; innerIterationLoop <= outerIterationLoop;
innerIterationLoop++) {
				Node goalState = null;
				do {
					initialState = getRandomConfiguration();
					initialSidewaysCount = 0;
					queue = new PriorityQueue();
					goalState = sidewaysHillClimbSearch(new
Node(initialState));
					totalNumberOfRestarts++;
					totalDepthForFailureState =
totalDepthForFailureState + goalState.getDepth();
				} while (goalState.getPathCost() != 0);
```

```
	}
			float averageNumberOfRestarts = ((float) totalNumberOfRestarts) /
outerIterationLoop;
			float averageDepthOfFailureState = ((float)
totalDepthForFailureState / outerIterationLoop);
			System.out.println("Average number of Random restarts :" +
averageNumberOfRestarts);
			System.out.println("Average number of steps :" +
averageDepthOfFailureState);
			System.out.println("************************************");
		}
	}
}
package IS_Project2;

import java.util.Scanner;

import IS_Project1.PriorityQueue;

public class RandomRestartWithoutSidewaysHC extends SteepestAscentHC {

	/**
	 * Hill climbing with random restart is been implemented without sideways
move.
	 */
	public static void main(String[] args) {
		System.out.println("Enter the size of the N Queen problem : \n");
		Scanner inputState = new Scanner(System.in);
		boardSize = inputState.nextInt();
		if (boardSize < 4) {
			System.out.println("N Queen problem cannot be solved for the
size: " + boardSize);
			return;
		}
		for (int outerIterationLoop = iterationNumber; outerIterationLoop <=
iterationLimit; outerIterationLoop = outerIterationLoop
				+ 100) {
			int totalNumberOfRestarts = 0;
			int totalDepthForFailureState = 0;
			System.out.println("Iteration Number :" + outerIterationLoop);
			for (int innerLoop = 0; innerLoop <= outerIterationLoop;
innerLoop++) {
				Node goalState = null;
			do {
					initialState = getRandomConfiguration();
					queue = new PriorityQueue();
			goalState = steepestAscentHillClimbSearch(new
Node(initialState));
					totalNumberOfRestarts++;
	totalDepthForFailureState = totalDepthForFailureState + goalState.getDepth();
} while (goalState.getPathCost() != 0);

}
```

```
float averageNumberOfRestarts = ((float) totalNumberOfRestarts) / outerIterationLoop;
                    float averageDepthOfFailureState = ((float)
totalDepthForFailureState / outerIterationLoop);
                    System.out.println("Average number of Random restarts :" +
averageNumberOfRestarts);
                    System.out.println("Average number of steps :" +
averageDepthOfFailureState);
                    System.out.println("************************************");
                }
        }
}
package IS_Project2;

import java.util.ArrayList;
import java.util.List;

public class Node {

        private List<Integer> state;
        private int pathCost;
        private Node parentNode;
        private List<Node> adjacentNodes;
        private int depth;

        Node(List<Integer> state){
                this.state = state;
        }

        Node(Node parentNode, List<Integer> state, int pathCost, int depth) {
                this.parentNode = parentNode;
                this.state = state;
                this.pathCost = pathCost;
                this.depth = depth;
        }

        public List<Integer> getState() {
                return state;
        }

        public int getPathCost() {
                return pathCost;
        }

        public void setPathCost(int pathCost) {
                this.pathCost = pathCost;
        }

        public List<Node> getAdjacentNodes() {
                if(adjacentNodes == null) {
                        return new ArrayList<Node>();
                }
                return adjacentNodes;
        }

        public void setAdjacentNodes(List<Node> adjacentNodes) {
                this.adjacentNodes = adjacentNodes;
        }
```

```
public Node getParentNode() {
		return parentNode;
	}


	public int getDepth() {
		return depth;
	}

	/**
	 * The function generates the child nodes for the given rootnode by moving the
	 * queens up and down columnwise.
	 */
	public void generateAdjacentNodes() {
		List<Integer> parentState = this.getState();
		int boardIndex = 0;
		while(boardIndex < SteepestAscentHC.boardSize) {
			int tempIndex = boardIndex;
			List<Integer> tempParentState = new
ArrayList<Integer>(parentState);
			int tempColIndex = boardIndex;
			int currentQueenIndex =
getQueenIndexToBeMoved(tempParentState,tempColIndex);
			int depth  = this.getDepth() + 1;
			for( int colIndex = 0; colIndex < SteepestAscentHC.boardSize ;
colIndex ++) {
				tempIndex = (colIndex == 0 ) ? tempIndex : tempIndex +
SteepestAscentHC.boardSize;
				if(tempIndex != currentQueenIndex) {
					List<Integer> adjacentState = new
ArrayList<Integer>(tempParentState);
					adjacentState.set(tempIndex, 1);
					Node newNode = new Node(this,
adjacentState,calculateStateHeuristic(adjacentState),depth);
					List<Node> adjacentNodes = this.getAdjacentNodes();
					adjacentNodes.add(newNode);
					this.setAdjacentNodes(adjacentNodes);
				}
			}
			boardIndex++;
		}
	}

/**
	 * The function generates the child nodes for the given rootnode by moving the
	 * queens up and down columnwise.
	 */
	public void generateAdjacentNodes() {
		List<Integer> parentState = this.getState();
		int boardIndex = 0;
		while(boardIndex < SteepestAscentHC.boardSize) {
			int tempIndex = boardIndex;
```

```
List<Integer> tempParentState = new ArrayList<Integer>(parentState);
                    int tempColIndex = boardIndex;
                    int currentQueenIndex =
getQueenIndexToBeMoved(tempParentState,tempColIndex);
                    int depth  = this.getDepth() + 1;
                    for( int colIndex = 0; colIndex < SteepestAscentHC.boardSize ;
colIndex ++) {
                          tempIndex = (colIndex == 0 ) ? tempIndex : tempIndex +
SteepestAscentHC.boardSize;
                          if(tempIndex != currentQueenIndex) {
                                List<Integer> adjacentState = new
ArrayList<Integer>(tempParentState);
                                adjacentState.set(tempIndex, 1);
                                Node newNode = new Node(this,
adjacentState,calculateStateHeuristic(adjacentState),depth);
                                List<Node> adjacentNodes = this.getAdjacentNodes();
                                adjacentNodes.add(newNode);
                                this.setAdjacentNodes(adjacentNodes);
                          }
                    }
                    boardIndex++;
              }
        }

        /**
         * The function gives the index of the queen to be moved up and
         *  down to generate child nodes.
         *
         * @param tempParentState
         * @param colIndex
         * @return
         */
        private int getQueenIndexToBeMoved(List<Integer> tempParentState, int
colIndex) {
              int tempIndex = colIndex;
              int counter = 0;
              List<Integer> currentState = this.getState();
              while(counter < SteepestAscentHC.boardSize) {
                    if(currentState.get(tempIndex) == 1) {
                          tempParentState.set(tempIndex, 0);
                          return tempIndex;
                    }
                    tempIndex = tempIndex + SteepestAscentHC.boardSize;
                    counter++;
              }
              return -1;
        }
```

```
	/**
	 * The function generates the heuristic value for each state i.e. the number
of
	 * queens getting eachother.
	 *
	 * @param state
	 * @return
	 */
	public int calculateStateHeuristic(List<Integer> state) {
		int boardIndex = 0;
		int heuristicCount = 0;
		int boardSize = SteepestAscentHC.boardSize;
		while (boardIndex < boardSize * boardSize) {
			boardIndex = this.getQueenIndexForTheState(state, boardIndex);
			if (boardIndex < 0) {
				return heuristicCount;
			}
			int tempBoardIndex = boardIndex + 1;
			int upperDiagIndex = boardIndex;
			int lowerDiagIndex = boardIndex;
			while (((tempBoardIndex - 1) % boardSize) < (boardSize - 1)) {
				upperDiagIndex = upperDiagIndex - boardSize + 1;
				lowerDiagIndex = lowerDiagIndex + boardSize + 1;
				if (state.get(tempBoardIndex) == 1) {
					heuristicCount++;
				}
				if (upperDiagIndex > 0 && state.get(upperDiagIndex) == 1)
{
					heuristicCount++;
				}
				if (lowerDiagIndex < (boardSize * boardSize) &&
state.get(lowerDiagIndex) == 1) {
					heuristicCount++;
				}
				tempBoardIndex++;
			}
			boardIndex++;
		}
		return heuristicCount;
	}
	/**
	 * The function checks if the queen is present in the given state at the given
	 * index. If yes, returns the index else returns the index where the queen is
	 * actually present.
	 *
	 * @param state
	 * @param givenIndex
	 * @return
	 */
	private int getQueenIndexForTheState(List<Integer> state, int givenIndex) {
		int boardSize = SteepestAscentHC.boardSize;
		while (givenIndex < (boardSize * boardSize)) {
			if (state.get(givenIndex) == 1) {
				return givenIndex;
			}
			givenIndex++;
		}
		return -1;} }
```

3.Sample Outputs

1. **Steepest Ascent Hill Climbing**

Number of Iterations	Success Rate (%)	Failure Rate (%)	Avg number of steps during success	Avg number of steps during failure
300	14.6	85.3	3.7	3.09
400	19.75	80.25	4.12	3.04
500	15	85	3.98	3.07
600	15.5	84.5	4.17	3.04
700	14.85	85.14	4.02	3.09
800	14.62	85.37	3.94	3.08
900	14.55	85.44	4.06	3.10
1000	15.8	84.2	4.05	3.07

Sample Inputs

Enter the size of the N Queen problem :

8
1. Initial State :

[1 0 0 0 1 0 0 0

0 0 1 0 0 0 0 1

0 0 0 1 0 0 1 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

0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0]
Goal Reached: true
Path Cost (h(n)):10
Depth d :0
[
1 0 0 0 1 0 0 0
0 0 1 0 0 0 0 1
0 0 0 1 0 0 1 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
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
]
****

Path Cost (h(n)):6
Depth d :1
[
1 0 0 0 1 0 0 0
0 0 1 0 0 0 0 1
0 0 0 0 0 0 1 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
0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0
]
****
Path Cost (h(n)):4
Depth d :2
[
1 0 0 0 1 0 0 0
0 0 1 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0
0 1 0 0 0 1 0 0
0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0
]
****
Path Cost (h(n)):2
Depth d :3
[
1 0 0 0 1 0 0 0
0 0 1 0 0 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 0 1
0 0 0 0 0 1 0 0
0 0 0 1 0 0 0 0
]
****
Path Cost (h(n)):1
Depth d :4
[
1 0 0 0 1 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 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
]
****

Path Cost (h(n)):0
Depth d :5
[
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 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
]

2. Initial State :

[0 0 0 0 1 0 0 0

0 0 1 0 0 0 0 0

1 0 0 0 0 1 0 0

0 0 0 0 0 0 0 1

0 0 0 1 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]
Goal Reached : false
Path Cost (h(n)):3
Depth d :0
[
0 0 0 0 1 0 0 0
0 0 1 0 0 0 0 0
1 0 0 0 0 1 0 0
0 0 0 0 0 0 0 1
0 0 0 1 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
]
****
Path Cost (h(n)):2
Depth d :1
[
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 1 0 1
0 0 0 1 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
]
****

Path Cost (h(n)):1
Depth d :2
[
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 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 1 0
0 0 0 0 0 0 0 1
]
****
3. Initial State :

[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 1 0 0 1 0 0

0 0 0 0 0 0 0 0

0 1 0 0 0 0 0 0

1 0 0 0 0 0 0 0

0 0 0 0 0 0 1 0]
Goal Reached : true
Path Cost (h(n)):5
Depth d :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 1 0 0 1 0 0
0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0
1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
]
****
Path Cost (h(n)):3
Depth d :1
[
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 1 0 0 0 0 0
0 0 0 0 0 1 0 0
0 1 0 0 0 0 0 0
1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
]
****

Path Cost (h(n)):2
Depth d :2
[
0 0 0 1 1 0 0 0
0 0 0 0 0 0 0 1
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 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
]
****
Path Cost (h(n)):1
Depth d :3
[
0 0 0 1 1 0 0 0
0 0 0 0 0 0 0 1
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 1 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0
]
****
Path Cost (h(n)):0
Depth d :4
[
0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 1
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 1 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0
]
****

4. Initial State :

[1 0 0 1 1 1 0 0

0 0 0 0 0 0 1 0

0 0 0 0 0 0 0 0

0 1 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]

Goal Reached : false
Path Cost (h(n)):11
Depth d :0
[
1 0 0 1 1 1 0 0
0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0
0 1 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
]
****
Path Cost (h(n)):7
Depth d :1
[
1 0 0 1 1 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0
0 1 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 1 0 0
]
****
Path Cost (h(n)):4
Depth d :2
[
1 0 0 1 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0
0 1 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 1 0 0
]
****
Path Cost (h(n)):2
Depth d :3
[
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 1 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 1 0 0 0 0
0 0 0 0 0 1 0 0
]
****

Path Cost (h(n)):1
Depth d :4
[
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 0 0 1
0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 1 0 0 0 1 0 0
]
****

2. Hill Climbing with Sideways move

Number of Iterations	Success Rate (%)	Failure Rate (%)	Avg number of steps during success	Avg number of steps during failure
300	96.33	3.66	10.91	6.36
400	96.5	3.5	11.17	6.78
500	96.0	4.0	11.09	5.4
600	96.8	3.16	11.67	5.89
700	96.71	3.28	11.29	5.69
800	96.25	3.75	11.12	6.6
900	96.22	3.77	11.023	5.44
1000	96.0	4.0	10.82	6.8

Sample Inputs

Enter the size of the N Queen problem :

8

1. Initial State :

[0 0 0 0 0 1 0 1

0 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 1 0

1 0 0 0 0 0 0 0

0 1 0 0 1 0 0 0]

Goal Reached :true

Path Cost (h(n)):7
Depth d :0
[
0 0 0 0 0 1 0 1
0 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 1 0
1 0 0 0 0 0 0 0
0 1 0 0 1 0 0 0
]
****
Path Cost (h(n)):5
Depth d :1
[
0 0 0 0 0 1 0 1
0 0 0 1 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 0 1 0
1 0 0 0 0 0 0 0
0 1 0 0 1 0 0 0
]
****
Path Cost (h(n)):4
Depth d :2
[
0 0 0 0 0 0 0 1
0 0 0 1 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 1 0
1 0 0 0 0 0 0 0
0 1 0 0 1 0 0 0
]
****
Path Cost (h(n)):3
Depth d :3
[
0 0 0 0 0 0 0 1
0 0 0 1 0 0 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 0 0 0 0 1 0 0
1 0 0 0 0 0 0 0
0 1 0 0 1 0 0 0
]
****

Path Cost (h(n)):2
Depth d :4
[
0 0 0 0 0 0 0 1
0 0 0 1 0 0 0 0
0 1 0 0 0 0 0 0
0 0 1 0 0 0 1 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 1 0 0 0
]
****
Path Cost (h(n)):1
Depth d :5
[
0 0 0 0 0 0 0 1
0 0 0 1 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 1 0 0
1 0 1 0 0 0 0 0
0 0 0 0 1 0 0 0
]
****
Path Cost (h(n)):1
Depth d :6
[
0 0 0 0 0 0 0 1
0 0 0 1 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
1 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0
]
****
Path Cost (h(n)):1
Depth d :7
[
0 0 0 0 0 0 0 1
0 0 0 1 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
1 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0
]
****

Path Cost (h(n)):1
Depth d :8
[
0 0 0 0 0 0 0 0
0 0 0 1 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
1 0 0 0 0 1 0 0
0 0 0 0 0 0 0 1
0 0 0 0 1 0 0 0
]
****
Path Cost (h(n)):1
Depth d :9
[
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 1 0
0 0 1 0 0 0 0 0
1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1
0 0 0 0 1 0 0 0
]
****
Path Cost (h(n)):1
Depth d :10
[
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 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 1
1 0 0 0 1 0 0 0
]
****
Path Cost (h(n)):1
Depth d :11
[
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 1 0
0 0 1 0 1 0 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
]
****

Path Cost (h(n)):1
Depth d :12
[
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 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 1
1 0 0 0 0 0 0 0
]
****
Path Cost (h(n)):1
Depth d :13
[
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 1 0
0 0 0 0 1 0 0 0
0 0 1 0 0 0 0 0
1 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0
]
****
Path Cost (h(n)):1
Depth d :14
[
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 1 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 0 1
]
****
Path Cost (h(n)):1
Depth d :15
[
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 1 1
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 0 0
]
****

Path Cost (h(n)):1
Depth d :16
[
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 1 0 0 0 0 0
1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
]
****
Path Cost (h(n)):1
Depth d :17
[
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 1 0 0 0 1 0
1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
]
****
Path Cost (h(n)):0
Depth d :18
[
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
0 0 1 0 0 0 0 0
]
****

2. Initial State :

[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 0 0 0 1 0

0 0 0 1 0 0 0 1

0 1 1 0 1 0 0 0]

Goal Reached :false

Path Cost (h(n)):10
Depth d :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 0 0 0 1 0
0 0 0 1 0 0 0 1
0 1 1 0 1 0 0 0
]
****
Path Cost (h(n)):6
Depth d :1
[
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 0
0 0 0 0 0 0 0 0
1 0 0 0 0 0 1 0
0 0 0 1 0 0 0 1
0 1 1 0 0 0 0 0
]
****
Path Cost (h(n)):3
Depth d :2
[
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 0
0 0 1 0 0 0 0 0
1 0 0 0 0 0 1 0
0 0 0 1 0 0 0 1
0 1 0 0 0 0 0 0
]
****
Path Cost (h(n)):2
Depth d :3
[
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 0
0 0 1 0 0 0 0 0
1 0 0 0 0 0 1 0
0 0 0 1 0 0 0 0
0 1 0 0 0 0 0 1
]
****

Path Cost (h(n)):1
Depth d :4
[
0 0 0 0 1 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 1 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 1 0 0 0 0
0 1 0 0 0 0 0 1
]
****
Path Cost (h(n)):1
Depth d :5
[
0 0 0 0 1 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 1
0 0 1 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 1 0 0 0 0
0 1 0 0 0 0 0 0
]
****
Path Cost (h(n)):1
Depth d :6
[
0 0 0 0 1 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 1 0 0 0 0 0
0 0 0 0 0 0 1 1
0 0 0 1 0 0 0 0
0 1 0 0 0 0 0 0
]
****
3.Initial State :

[0 0 0 0 1 0 0 1
1 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 1 0 0 0 0
0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0]
Goal Reached :true

Path Cost (h(n)):7
Depth d :0
[
0 0 0 0 1 0 0 1
1 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 1 0 0 0 0
0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0
]
****
Path Cost (h(n)):4
Depth d :1
[
0 0 0 0 1 0 0 0
1 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 1
0 0 0 1 0 0 0 0
0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0
]
****
Path Cost (h(n)):3
Depth d :2
[
0 0 0 0 1 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 1
0 0 0 1 0 0 0 0
0 1 0 0 0 0 0 0
0 0 0 0 0 0 1 0
]
****
Path Cost (h(n)):3
Depth d :3
[
0 0 0 0 1 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 0 0 0 0 1
0 0 0 1 0 0 0 0
0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0
]
****
Path Cost (h(n)):2
Depth d :4
[
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 0 0 0 0 1
0 0 0 1 0 0 0 0
0 1 0 0 0 0 0 0
0 0 0 0 1 0 0 0
]
****

Path Cost (h(n)):2
Depth d :5
[
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 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 1 0 0 0
]
****
Path Cost (h(n)):2
Depth d :6
[
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 1
0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 1 0 0 0 0 0 0
0 0 0 0 1 0 0 0
]
****
Path Cost (h(n)):0
Depth d :7
[
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 0 0 1
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 0 0
]
****
4. Initial State :

[0 1 0 0 0 0 0 0

0 0 1 0 0 0 0 0

0 0 0 0 0 0 1 0

0 0 0 0 1 0 0 0

1 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 1 0 0]

Goal Reached :true

Path Cost (h(n)):7
Depth d :0
[
0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0
1 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 1 0 0
]
****
Path Cost (h(n)):4
Depth d :1
[
0 1 0 0 0 0 0 0
0 0 1 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 1
0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0
]
****
Path Cost (h(n)):3
Depth d :2
[
0 1 0 0 0 0 0 0
0 0 1 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 1
0 0 0 0 1 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0
]
****
Path Cost (h(n)):2
Depth d :3
[
0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 1 0 0 0 0
1 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 0 0 0 1 0 0
]
****

Path Cost (h(n)):1
Depth d :4
[
0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 1 0 0 0 0
1 0 0 0 0 0 0 1
0 0 0 0 1 0 0 0
0 1 0 0 0 0 0 0
0 0 0 0 0 1 0 0
]
****
Path Cost (h(n)):1
Depth d :5
[
0 0 0 0 0 0 0 1
0 0 1 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 1 0 0 0 0
1 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 0 0 1 0 0
]
****
Path Cost (h(n)):1
Depth d :6
[
0 0 0 0 0 0 0 1
0 0 1 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 1 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 0 0 0 1 0 0
]
****
Path Cost (h(n)):1
Depth d :7
[
0 0 0 0 0 0 0 1
0 0 1 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 1 0 0 0 0
0 1 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 0 0 1 0 0
]
****

Path Cost (h(n)):1
Depth d :8
[
0 0 0 0 0 0 0 1
0 0 1 0 0 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 1 0 0 0
1 0 0 0 0 0 0 0
0 0 0 1 0 1 0 0
]
****
Path Cost (h(n)):1
Depth d :9
[
0 0 0 0 0 0 0 1
0 0 1 0 0 0 0 0
0 0 0 0 0 0 1 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
1 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0
]
****
Path Cost (h(n)):1
Depth d :10
[
0 0 0 0 0 0 0 1
0 0 1 0 0 0 0 0
0 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
1 0 0 0 0 0 1 0
0 0 0 1 0 0 0 0
]
****
Path Cost (h(n)):0
Depth d :11
[
0 0 0 0 0 0 0 1
0 0 1 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 1 0
0 0 0 1 0 0 0 0
]
****

3.Random Restart Hill Climbing Search without sideways move

Number of Iterations	Average number of Random Restarts	Average number of steps
300	6.39	20.44
400	7.17	22.93
500	6.61	21.21
600	6.58	21.23
700	6.54	20.92
800	6.94	22.20
900	6.90	22.16
1000	6.30	20.40

4.Random Restart Hill Climbing Search with sideways move

Number of Iterations	Average number of Random Restarts	Average number of steps
300	1.056	11.16
400	1.03	11.34
500	1.038	11.078
600	1.035	10.65
700	1.044	11.3
800	1.033	11.12
900	1.046	10.89
1000	1.049	11.49

3. References:

1) www.stackoverflow.com
2) www.geeksforgeeks.com
3) https://en.wikipedia.org/wiki/Hill_climbing