

**INDIAN INSTITUTE OF INFORMATION TECHNOLOGY**

BHAGALPUR, Bihar

COURSE NAME:- ARTIFICIAL INTELLIGENCE

Table of Contents

[** Project Title: 3**](#_Toc57214597)

[** Acknowledgement:- 3**](#_Toc57214598)

[** Abstract:- 4**](#_Toc57214599)

[** Introduction:- 4**](#_Toc57214600)

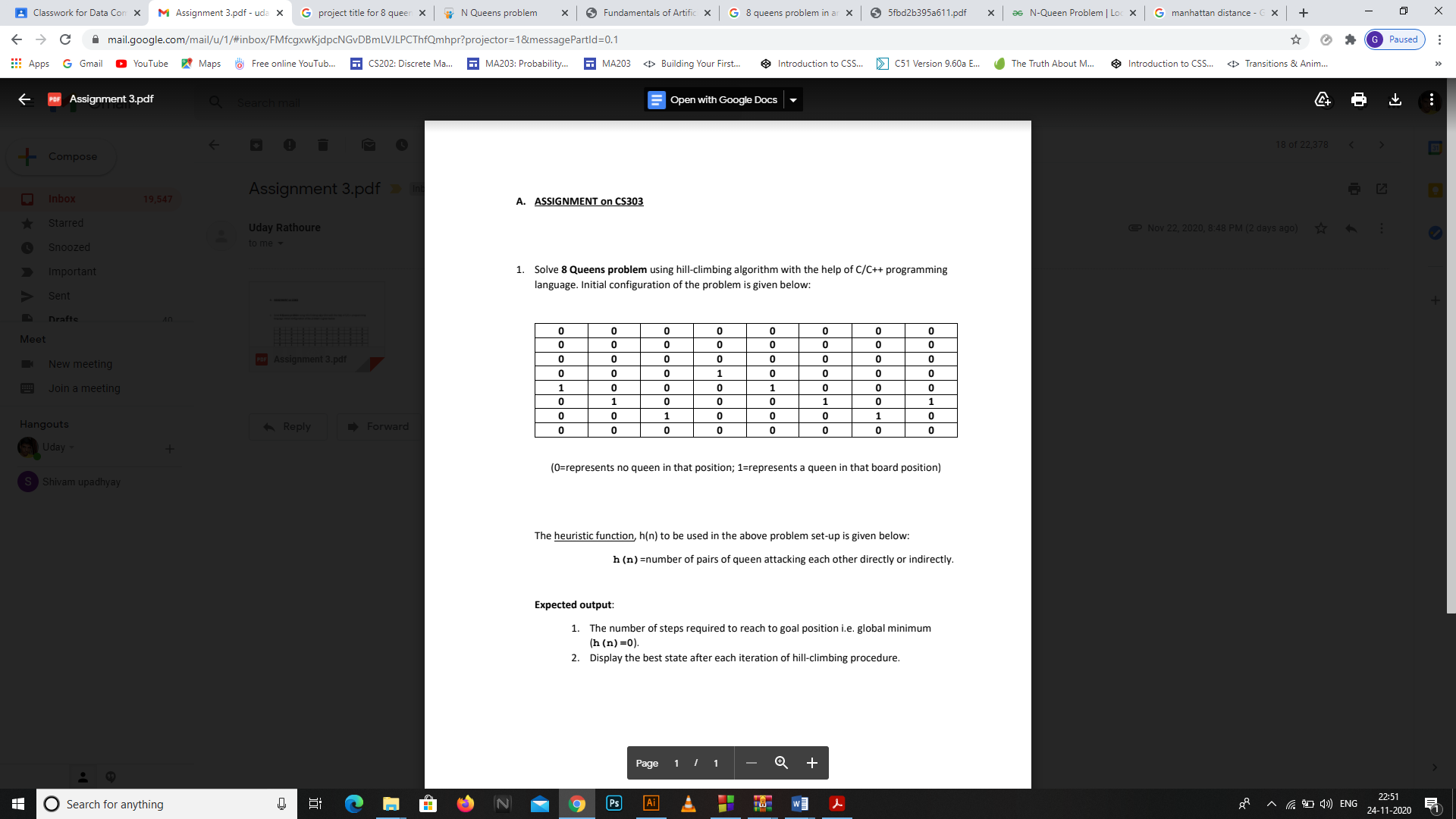
[** Algorithm:- 6**](#_Toc57214601)

[** Program code 9**](#_Toc57214602)

[** Output 16**](#_Toc57214603)

# Project Title:

Solve 8 Queens problem using hill-climbing algorithm with the help of C/C++ programming language. Initial configuration of the problem is given below:



# Acknowledgement:-

In performing our assignment, we had to take the help and guideline of some respected persons, who deserve our greatest gratitude. The completion of this assignment gives us much Pleasure. We would like to show our gratitude to Prof. Rupam Bhattacharyya, Assistant Professor, The Department of Computer Science Engineering, Indian Institute of Information Technology, Bhagalpur for giving us a good guideline for assignment throughout numerous consultations. We would also like to expand our deepest gratitude to all those who have directly and indirectly guided us in writing this assignment. In addition, a thank you to many people, especially our batch mates and team members itself, have made valuable suggestions on this proposal which gave us an inspiration to improve our assignment. We thank all the people for their help directly and indirectly to complete our assignment.

# Abstract:-

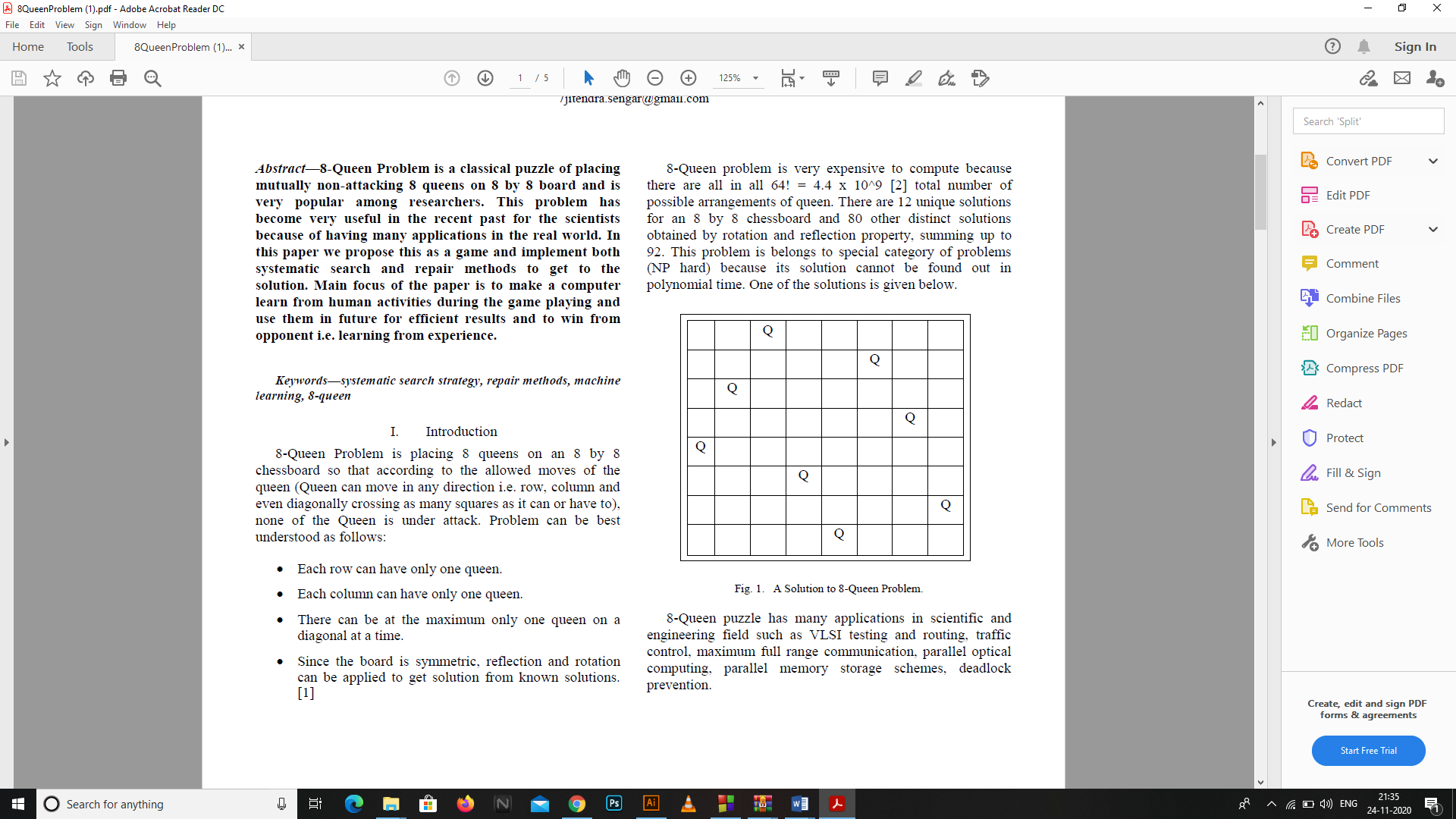
8-Queen Problem is a classical puzzle of placing mutually non-attacking 8 queens on 8 by 8 board and is very popular among researchers. This problem has become very useful in the recent past for the scientists because of having many applications in the real world. In this project we propose this as a game and implement using hill-climbing algorithm with the help of C/C++ programming language to get to the solution. The most primitive way to solve them is "Brute-Force" - that means: Simply try everything out. The n-Queen problem become intractable for large values and thus placed in NP (non-deterministic polynomial) class problem. The n-Queen problem is basically a generalized form of 8-Queen problem. The solution can very easily be extended to the generalized form of the problem for large values.

# Introduction:-

8-Queen Problem is placing 8 queens on an 8 by 8 chessboard so that according to the allowed moves of the queen(Queen can move in any direction i.e. row, column and even diagonally crossing as many squares as it can or have to),none of the Queen is under attack. Problem can be best understood as follows:

* Each row can have only one queen.
* Each column can have only one queen.
* There can be maximum only one queen on a diagonal at a time.
* Since the board is symmetric, reflection and rotation can be applied to get solution from known solutions.

8-Queen problem is very expensive to compute because there are all in all 64! = 4.4 x 10^9 [2] total number of possible arrangements of queen. There are 12 unique solutions for an 8 by 8 chessboard and 80 other distinct solutions obtained by rotation and reflection property, summing up to 92. This problem is belongs to special category of problems (NP hard) because its solution cannot be found out in polynomial time. One of the solutions is given below.



8-Queen puzzle has many applications in scientific and engineering field such as VLSI testing and routing, traffic control, maximum full range communication, parallel optical computing, parallel memory storage schemes, deadlock prevention. Machine learning is one of the most fascinating areas of Artificial Intelligence. It is basically the study of building computer systems which adapt and improves with its experience. Most of the problems of data acquisition are solved by machine learning.

# Algorithm:-

Hill Climbing Search

(Steepest Ascent/Descent)

• At each iteration, the hill-climbing search algorithm moves to the best successor of the current node according to an objective function.

• Best successor is the successor with best value (highest or lowest) according to an objective function.

• If no successors have better value than the current value, it returns.

• It moves in direction of uphill (hill climbing).

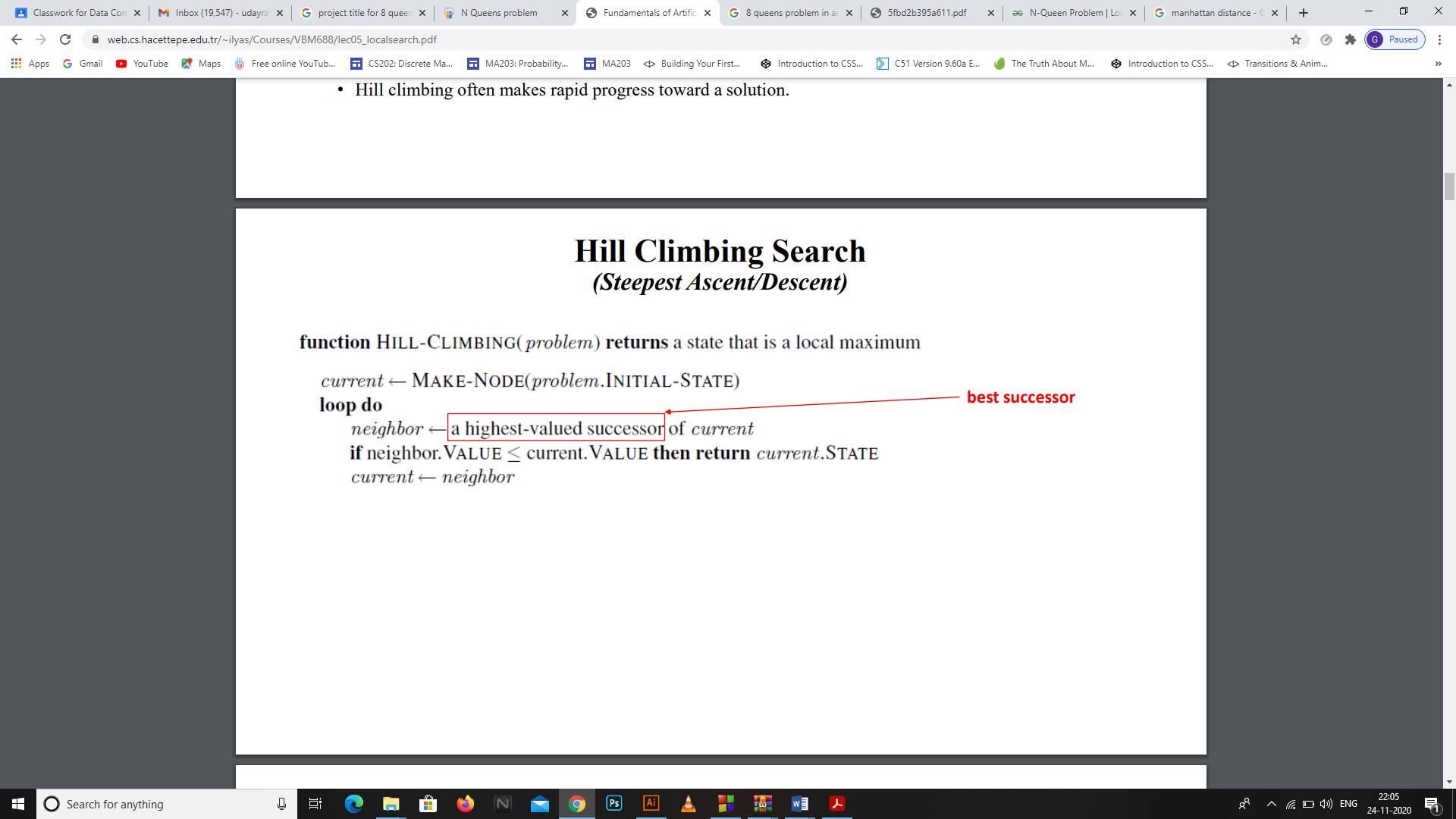
• It terminates when it reaches a “peak” where no neighbor has a higher value.

• The algorithm does not maintain a search tree, so the data structure for the current node need only record the state and the value of the objective function.

• Hill climbing does not look ahead beyond the immediate neighbors of the current state.

• Hill climbing is sometimes called **greedy local search** because it grabs a good neighbor state without thinking ahead about where to go next.

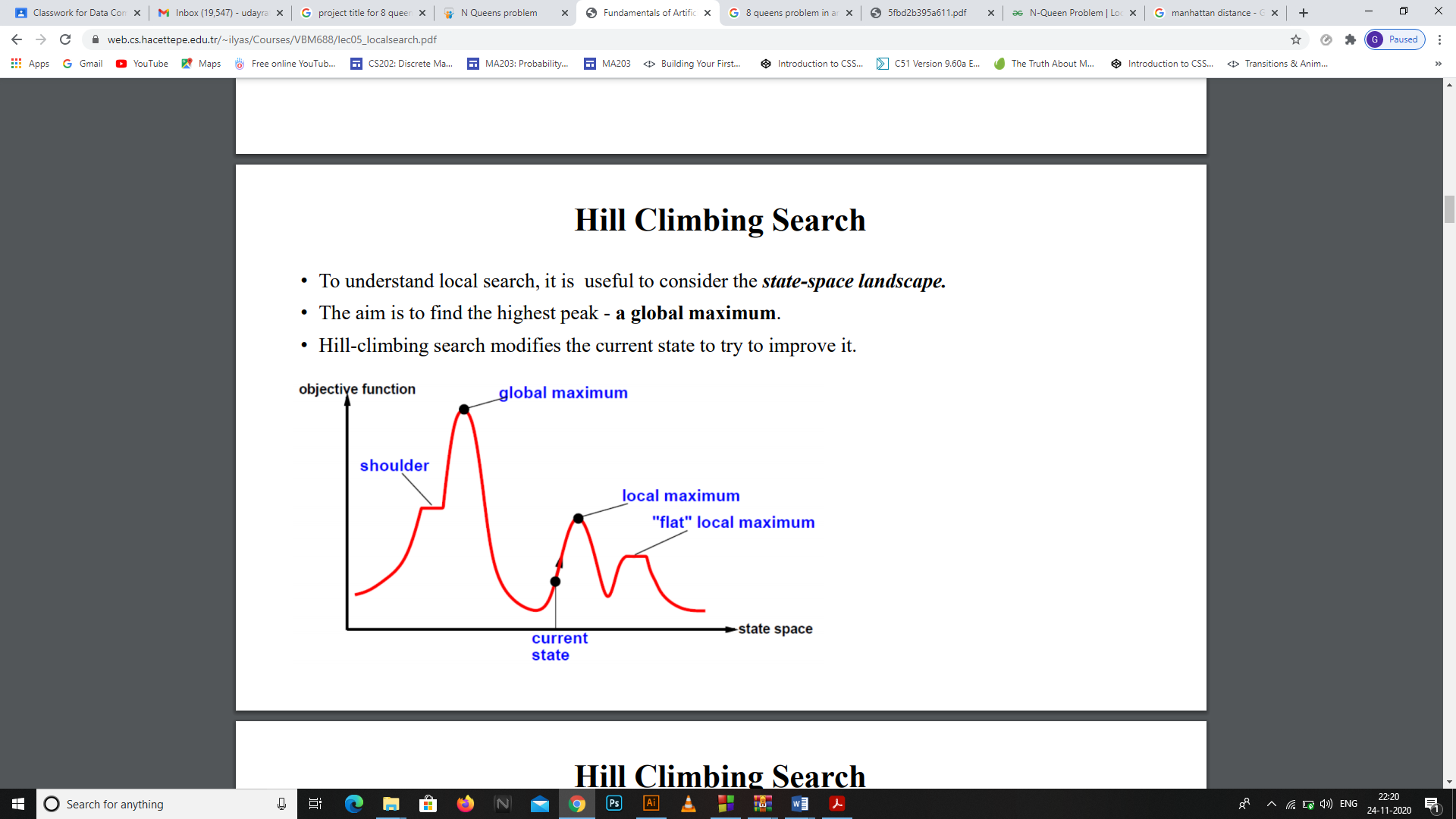
• Greedy algorithms often perform quite well and

• Hill climbing often makes rapid progress toward a solution.

To understand local search, it is useful to consider the **state-space landscape.**

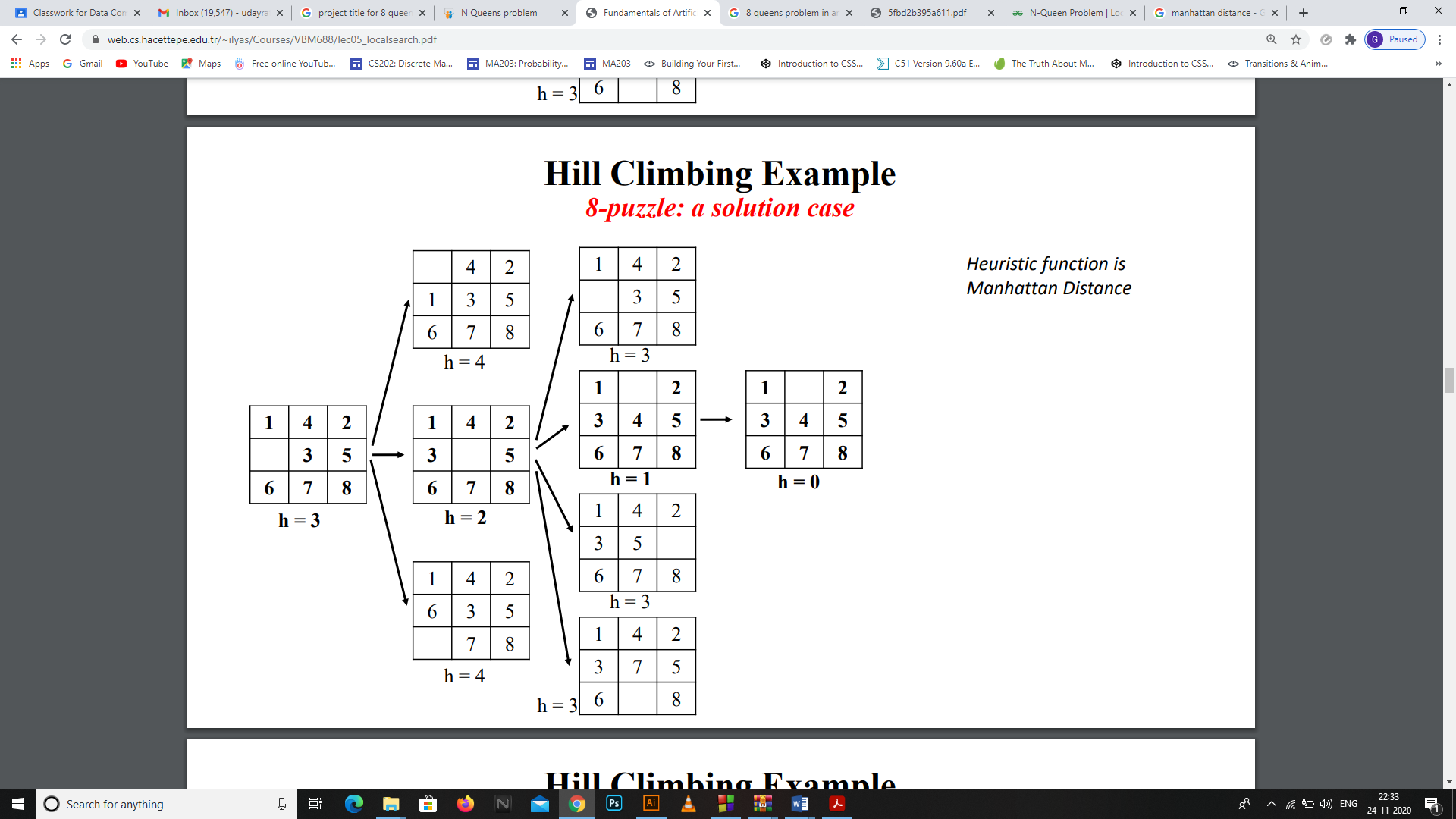
* The aim is to find the highest peak - **a global maximum.**
* Hill-climbing search modifies the current state to try to improve it.
* A local maximum is a peak that is higher than each of its neighboring states but lower than the global maximum.
  + Hill-climbing algorithms that reach the vicinity of a local maximum will be drawn upward toward the peak but will then be stuck with nowhere else to go.
* A **plateau** is a **flat area** of the state-space landscape. It can be a **flat local maximum,** from which no uphill exit exists, or a shoulder, from which progress is possible.
* A hill-climbing search might get lost on the plateau.

• Random sideways moves can escape from shoulders but they loop forever on flat maxima.



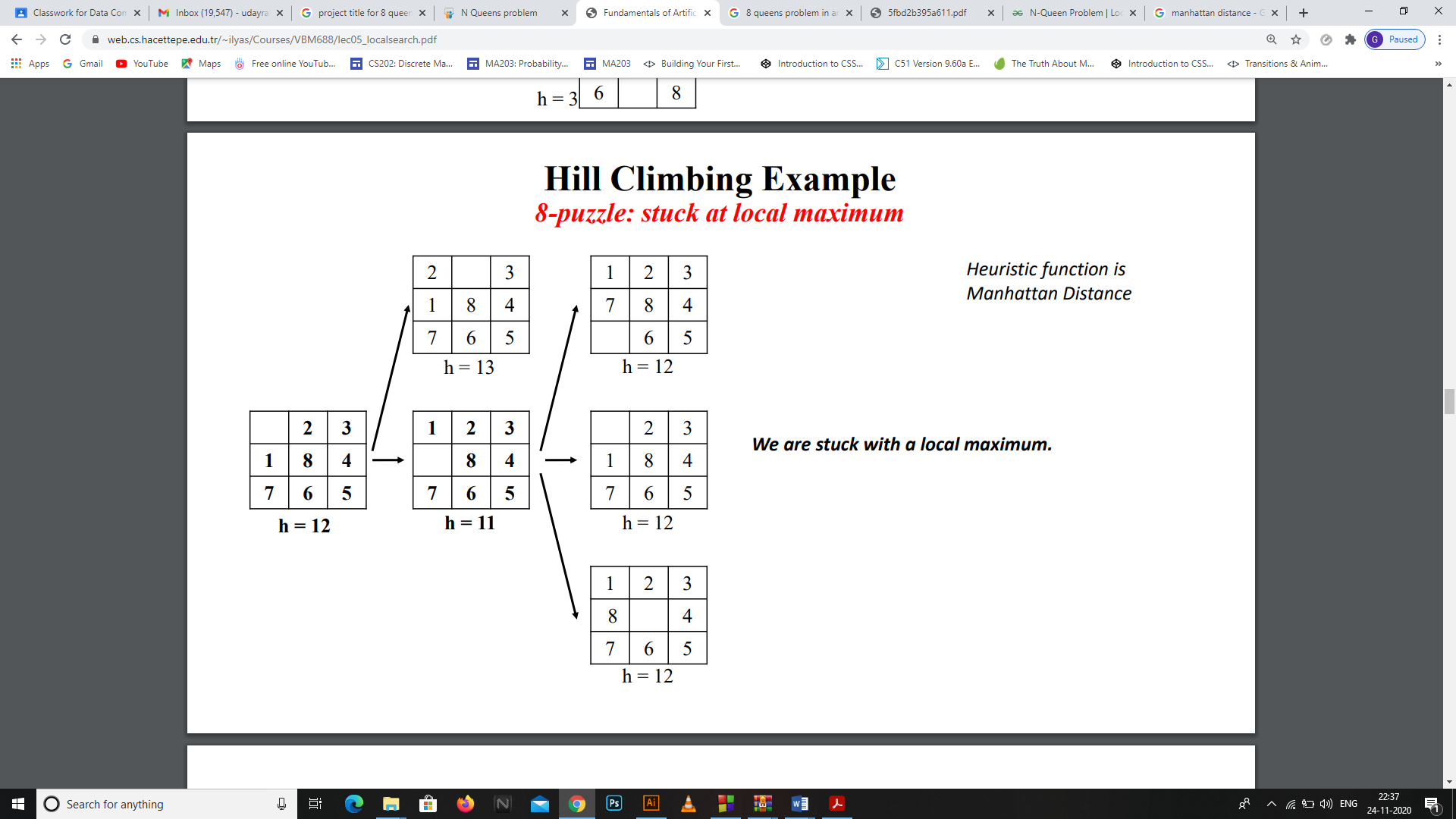
**Hill Climbing Example**

**8-puzzle: a solution case**



**Hill Climbing Example**

**8-puzzle: stuck at local maximum**



# Program code

// C++ implementation of the

// above approach

#include <iostream>

#include <math.h>

#define N 8

**using** **namespace** std**;**

// A utility function that prints

// the 2D array "board".

void printBoard**(**int board**[][**N**])**

**{**

**for** **(**int i **=** 0**;** i **<** N**;** i**++)** **{**

cout **<<** " "**;**

**for** **(**int j **=** 0**;** j **<** N**;** j**++)** **{**

cout **<<** board**[**i**][**j**]** **<<** " "**;**

**}**

cout **<<** "\n"**;**

**}**

**}**

// A utility function that prints

// the array "state".

void printState**(**int**\*** state**)**

**{**

**for** **(**int i **=** 0**;** i **<** N**;** i**++)** **{**

cout **<<** " " **<<** state**[**i**]** **<<** " "**;**

**}**

cout **<<** endl**;**

**}**

// A utility function that compares

// two arrays, state1 and state2 and

// returns true if equal

// and false otherwise.

bool compareStates**(**int**\*** state1**,**

int**\*** state2**)**

**{**

**for** **(**int i **=** 0**;** i **<** N**;** i**++)** **{**

**if** **(**state1**[**i**]** **!=** state2**[**i**])** **{**

**return** **false;**

**}**

**}**

**return** **true;**

**}**

// A utility function that fills

// the 2D array "board" with

// values "value"

void fill**(**int board**[][**N**],** int value**)**

**{**

**for** **(**int i **=** 0**;** i **<** N**;** i**++)** **{**

**for** **(**int j **=** 0**;** j **<** N**;** j**++)** **{**

board**[**i**][**j**]** **=** value**;**

**}**

**}**

**}**

// This function calculates the

// objective value of the

// state(queens attacking each other)

// using the board by the

// following logic.

int calculateHuristic**(**int board**[][**N**],**

int**\*** state**)**

**{**

// For each queen in a column, we check

// for other queens falling in the line

// of our current queen and if found,

// any, then we increment the variable

// attacking count.

// Number of queens attacking each other,

// initially zero.

int attacking **=** 0**;**

// Variables to index a particular

// row and column on board.

int row**,** col**;**

**for** **(**int i **=** 0**;** i **<** N**;** i**++)** **{**

// At each column 'i', the queen is

// placed at row 'state[i]', by the

// definition of our state.

// To the left of same row

// (row remains constant

// and col decreases)

row **=** state**[**i**],** col **=** i **-** 1**;**

**while** **(**col **>=** 0

**&&** board**[**row**][**col**]** **!=** 1**)** **{**

col**--;**

**}**

**if** **(**col **>=** 0

**&&** board**[**row**][**col**]** **==** 1**)** **{**

attacking**++;**

**}**

// To the right of same row

// (row remains constant

// and col increases)

row **=** state**[**i**],** col **=** i **+** 1**;**

**while** **(**col **<** N

**&&** board**[**row**][**col**]** **!=** 1**)** **{**

col**++;**

**}**

**if** **(**col **<** N

**&&** board**[**row**][**col**]** **==** 1**)** **{**

attacking**++;**

**}**

// Diagonally to the left up

// (row and col simoultaneously

// decrease)

row **=** state**[**i**]** **-** 1**,** col **=** i **-** 1**;**

**while** **(**col **>=** 0 **&&** row **>=** 0

**&&** board**[**row**][**col**]** **!=** 1**)** **{**

col**--;**

row**--;**

**}**

**if** **(**col **>=** 0 **&&** row **>=** 0

**&&** board**[**row**][**col**]** **==** 1**)** **{**

attacking**++;**

**}**

// Diagonally to the right down

// (row and col simoultaneously

// increase)

row **=** state**[**i**]** **+** 1**,** col **=** i **+** 1**;**

**while** **(**col **<** N **&&** row **<** N

**&&** board**[**row**][**col**]** **!=** 1**)** **{**

col**++;**

row**++;**

**}**

**if** **(**col **<** N **&&** row **<** N

**&&** board**[**row**][**col**]** **==** 1**)** **{**

attacking**++;**

**}**

// Diagonally to the left down

// (col decreases and row

// increases)

row **=** state**[**i**]** **+** 1**,** col **=** i **-** 1**;**

**while** **(**col **>=** 0 **&&** row **<** N

**&&** board**[**row**][**col**]** **!=** 1**)** **{**

col**--;**

row**++;**

**}**

**if** **(**col **>=** 0 **&&** row **<** N

**&&** board**[**row**][**col**]** **==** 1**)** **{**

attacking**++;**

**}**

// Diagonally to the right up

// (col increases and row

// decreases)

row **=** state**[**i**]** **-** 1**,** col **=** i **+** 1**;**

**while** **(**col **<** N **&&** row **>=** 0

**&&** board**[**row**][**col**]** **!=** 1**)** **{**

col**++;**

row**--;**

**}**

**if** **(**col **<** N **&&** row **>=** 0

**&&** board**[**row**][**col**]** **==** 1**)** **{**

attacking**++;**

**}**

**}**

// Return pairs.

**return** **(**int**)(**attacking **/** 2**);**

**}**

// A utility function that

// generates a board configuration

// given the state.

void generateBoard**(**int board**[][**N**],**

int**\*** state**)**

**{**

fill**(**board**,** 0**);**

**for** **(**int i **=** 0**;** i **<** N**;** i**++)** **{**

board**[**state**[**i**]][**i**]** **=** 1**;**

**}**

**}**

// A utility function that copies

// contents of state2 to state1.

void copyState**(**int**\*** state1**,** int**\*** state2**)**

**{**

**for** **(**int i **=** 0**;** i **<** N**;** i**++)** **{**

state1**[**i**]** **=** state2**[**i**];**

**}**

**}**

// This function gets the neighbour

// of the current state having

// the least objective value

// amongst all neighbours as

// well as the current state.

void getNeighbour**(**int board**[][**N**],**

int**\*** state**)**

**{**

// Declaring and initializing the

// optimal board and state with

// the current board and the state

// as the starting point.

int opBoard**[**N**][**N**];**

int opState**[**N**];**

copyState**(**opState**,**

state**);**

generateBoard**(**opBoard**,**

opState**);**

// Initializing the optimal

// objective value

int opHuristic

**=** calculateHuristic**(**opBoard**,**

opState**);**

// Declaring and initializing

// the temporary board and

// state for the purpose

// of computation.

int NeighbourBoard**[**N**][**N**];**

int NeighbourState**[**N**];**

copyState**(**NeighbourState**,**

state**);**

generateBoard**(**NeighbourBoard**,**

NeighbourState**);**

// Iterating through all

// possible neighbours

// of the board.

**for** **(**int i **=** 0**;** i **<** N**;** i**++)** **{**

**for** **(**int j **=** 0**;** j **<** N**;** j**++)** **{**

// Condition for skipping the

// current state

**if** **(**j **!=** state**[**i**])** **{**

// Initializing temporary

// neighbour with the

// current neighbour.

NeighbourState**[**i**]** **=** j**;**

NeighbourBoard**[**NeighbourState**[**i**]][**i**]**

**=** 1**;**

NeighbourBoard**[**state**[**i**]][**i**]**

**=** 0**;**

// Calculating the objective

// value of the neighbour.

int temp

**=** calculateHuristic**(**

NeighbourBoard**,**

NeighbourState**);**

// Comparing temporary and optimal

// neighbour objectives and if

// temporary is less than optimal

// then updating accordingly.

**if** **(**temp **<=** opHuristic**)** **{**

opHuristic **=** temp**;**

copyState**(**opState**,**

NeighbourState**);**

generateBoard**(**opBoard**,**

opState**);**

**}**

// Going back to the original

// configuration for the next

// iteration.

NeighbourBoard**[**NeighbourState**[**i**]][**i**]**

**=** 0**;**

NeighbourState**[**i**]** **=** state**[**i**];**

NeighbourBoard**[**state**[**i**]][**i**]** **=** 1**;**

**}**

**}**

**}**

// Copying the optimal board and

// state thus found to the current

// board and, state since c++ doesn't

// allow returning multiple values.

copyState**(**state**,** opState**);**

fill**(**board**,** 0**);**

generateBoard**(**board**,** state**);**

**}**

void hillClimbing**(**int board**[][**N**],**

int**\*** state**)**

**{**

// Declaring and initializing the

// neighbour board and state with

// the current board and the state

// as the starting point.

int neighbourBoard**[**N**][**N**]** **=** **{};**

int neighbourState**[**N**];**

int count**=**0**;**

copyState**(**neighbourState**,** state**);**

generateBoard**(**neighbourBoard**,**

neighbourState**);**

**do** **{**

// Copying the neighbour board and

// state to the current board and

// state, since a neighbour

// becomes current after the jump.

copyState**(**state**,** neighbourState**);**

generateBoard**(**board**,** state**);**

// Getting the optimal neighbour

getNeighbour**(**neighbourBoard**,**

neighbourState**);**

int temp**=** calculateHuristic**(**neighbourBoard**,**neighbourState**);**

cout**<<**"\n"**;**

printBoard**(**neighbourBoard**);**

cout**<<**"no of pairs of queens attacking each other in upper state are : "**<<**temp**<<**endl**;**

count**++;**

**if** **(**compareStates**(**state**,**

neighbourState**))** **{**

// If neighbour and current are

// equal then no optimal neighbour

// exists and therefore output the

// result and break the loop.

// printBoard(board);

cout**<<**"number of steps taken are : "**<<**count**<<**endl**;**

**break;**

**}**

**else** **if** **(**calculateHuristic**(**board**,**

state**)**

**==** calculateHuristic**(**

neighbourBoard**,**

neighbourState**))** **{**

// If neighbour and current are

// not equal but their objectives

// are equal then we are either

// approaching a shoulder or a

// local optimum, in any case,

// jump to a random neighbour

// to escape it.

// Random neighbour

neighbourState**[**rand**()** **%** N**]**

**=** rand**()** **%** N**;**

generateBoard**(**neighbourBoard**,**

neighbourState**);**

**}**

**}** **while** **(true);**

**}**

// Driver code

int main**()**

**{**

int state**[**N**]** **=** **{};**

int board**[**N**][**N**]** **=** **{};**

cout**<<**"enter the initial postions\n"**;**

int temp**;**

**for(**int i**=**0**;**i**<**N**;**i**++)**

**{**

**for(**int j**=**0**;**j**<**N**;**j**++)**

**{**

cin**>>**temp**;**

board**[**i**][**j**]=**temp**;**

**if(**temp**==**1**)**

state**[**j**]=**i**;**

**}**

**}**

hillClimbing**(**board**,** state**);**

**return** 0**;**

**}**

# OUTPUT

enter the initial postions

1 1 1 1 1 1 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 0 0 0 0 0

0 0 0 0 0 0 0 0

0 0 0 0 0 0 1 0

no of pairs of queens attacking each other in upper state are : 6

1 1 1 1 1 1 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 0 0 0 0 0

0 0 0 0 0 0 1 0

no of pairs of queens attacking each other in upper state are : 5

1 1 1 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 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

no of pairs of queens attacking each other in upper state are : 4

1 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 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 1 0

no of pairs of queens attacking each other in upper state are : 3

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 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 1 0 0 0

0 0 0 0 0 0 1 0

no of pairs of queens attacking each other in upper state are : 2

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

1 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 1 0

no of pairs of queens attacking each other in upper state are : 1

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 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 1 0

no of pairs of queens attacking each other in upper state are : 1

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 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 1 0

no of pairs of queens attacking each other in upper state are : 1

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

1 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 1 0

no of pairs of queens attacking each other in upper state are : 1

0 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

1 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 1 0

no of pairs of queens attacking each other in upper state are : 1

0 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

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

no of pairs of queens attacking each other in upper state are : 1

0 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

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

no of pairs of queens attacking each other in upper state are : 1

0 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

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

no of pairs of queens attacking each other in upper state are : 0

0 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

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

no of pairs of queens attacking each other in upper state are : 0

number of steps taken are : 14