N Queens Problem

- Samruddhi Godbole
- sgodbol1@uncc.edu
- Dhruv Dhamani
- ddhamani@uncc.edu

Table of Contents

Introduction

Positioning queens on a chess board is a classical problem in mathematics and computer science.

The Queen's Puzzle (aka the eight queens puzzle), was originally published in 1848. It involves placing eight queens on an 8x8 chess board, in such a manner that no two queens can attack each other.

For example -

```
• • Q • • • • •
• • • • • Q • •
• • • • • • • Q
Q • • • • • • •
• • • Q • • • •
• • • • • • Q •
• • • • Q • • •
• Q • • • • • •
```

Problem Formulation

The problem can be formulated in two ways -

Incremental Formulation

- **Initial State**: an empty board
- **States**: any arrangement of 0-n queens on board
- **Operations**: add a queen to any square
- **Goal State**: n queens placed on the board such that no two queens can attack each other.

Complete State Formulation

- **Initial State**: a random arrangement on n queens, with one in each column
- **States**: any arrangement of n queens, one in each column
- **Operations**: move any attacked queen to another square in the same column
- **Goal State**: n queens placed on the board such that no two queens can attack each other.

The solution follows the Complete state formulation.

Problem Structure

Global Variables

```
int n;
```

The number of queens.

```
int BoardPos[100];
```

An array to store the positions of each queen.

```
int BoardCopy[100];
```

An array to store a copy of board positions of each queen.

```
int initialPos[100];
```

An array to store the initial position of queens.

```
int Success1=0;
int Failure1=0;
```

The count of successes and failures of type 1 (steepest ascent hill climbing)

```
int Success2=0;
int Failure2=0;
```

The count of successes and failures of type 2 (hill climbing search with sideways move)

```
int Success3=0;
int Failure3=0;
```

The count of successes and failures of type 3 (random-restart hill-climbing search)

```
int Success4=0;
int Failure4=0;
```

The count of successes and failures of type 4 (random-restart hill-climbing search with sideways move)

```
int numberOfRuns=300;
```

The number of times we attempt to solve the n-queens problem, so as to evaluate average metrics.

```
int successSteps1=0;
int successSteps2=0;

int failureSteps1=0;
int failureSteps2=0;
```

The number of steps to either success or failure for type 1 and type 2 respectively.

```
int randomRestartCount=0;
int randomRestartSideMoveCount=0;
```

The count of number of average restarts needed, and the count of the number of sideways moves made.

```
int randomRestartSteps=0;
int randomRestartStepsSideMove=0;
```

To store the number of steps needed with random restart without sideways move, and with sideways move respectively.

Functions/Procedures

Heuristic calculation procedure

The cost is calculated by simply comparing the position of each queen with every other queen; if in the same row, then the queens can attack each other and the cost is incremented. Similarly, the queens can also attack each other if they're in the same diagonal and the cost is incremented in that case.

```
int calculateH()
{
    int h=0;
    for(int i=0;i<n;i++)
    {
        for(int j=i+1;j<n;j++)
        {
            if(BoardPos[i]==BoardPos[j]) //check if in the same row
                h++;
            if( (BoardPos[i]==BoardPos[j]-(j-i)) || (BoardPos[i]==BoardPos[j]+(j-
i)) ) //check for diagonal elements
                h++;
        }
    }
    return h;
}
```

Other functions/procedures

```
void printSuccessFailure()
```

Prints all the metrics asked for in the problem statement.

```
void copyArray()
```

Utility function to copy the board positions.

```
void initializeBoard()
```

Utility function to initialize the positions of the queens on the board..

```
void printArray()
```

Utility function to print the board positions of the queens.

```
void hillClimbingSteep()
```

Solver for type 1, steepest ascent hill climbing.

```
void hillClimbingSideMove()
```

Solver for type 2, hill climbing search with sideways move.

```
void randomRestart()
```

Solver for type 3, random-restart hill-climbing search.

```
void randomRestartSideMove()
```

Solver for type 4, random-restart hill-climbing search with sideways move.

```
int main()
```

Responsible for taking the input from the user (number of queens), and calling the solver functions for each type and finally displaying metrics

Execution Results

```
Enter n value:
8

==================================
Steepest-ascent
==================================
Success:75
Failure:425
Average Success steps:4
Average Failure steps:4
Success rate: 15%
Failure rate:85%

==================================
Hill climbing With sideway Moves
==================================
Success:375
Failure:125
Average Success steps:9
Average Failure steps:10
Success rate: 75%
Failure rate:25%
```

```
==================================
Random Restart
==================================
Average Restart Required:6
Average steps:4

==================================
Random Restart with Sideway Moves
==================================
Average Restart Required:1
Average steps:9
```

Source Code

```
#include <iostream>
#include <stdlib.h>
#include<math.h>
using namespace std;

//Global variables:

    int n;
    int BoardPos[100];
    int BoardCopy[100];
    int initialPos[100];

    // for steepest-ascent
    int Success1=0;
    int Failure1=0;

    // for with side Moves
    int Success2=0;
    int Failure2=0;

    // randomRestart
    int Success3=0;
    int Failure3=0;

    //randomRestart with SideMove
    int Success4=0;
    int Failure4=0;

    int previousSuccess1=0;
    int previousSuccess2=0;

    int numberOfRuns=500;

    int successSteps1=0;
    int successSteps2=0;
```

```
    int failureSteps1=0;
    int failureSteps2=0;

    int randomRestartCount=0;
    int randomRestartSideMoveCount=0;

    int randomRestartSteps=0;
    int randomRestartStepsSideMove=0;

    int numberOfCalls3=0;
    int numberOfCalls4=0;

    int boardMoves[1000][8];
    int boardIndex=0;

void copyArray()
{
    for(int i=0;i<n;i++)
        BoardPos[i]=BoardCopy[i];
}
void initializeBoard()
{
    for(int i=0;i<n;i++)
    {
        BoardCopy[i]=initialPos[i];
        BoardPos[i]=initialPos[i];
    }
}
void printArray()
{
    cout<<"\n\n";
    for(int i=0;i<n;i++)
        cout<<BoardPos[i]<<" " ;
}

int calculateH()
{
    int h=0;
    for(int i=0;i<n;i++)
    {
        for(int j=i+1;j<n;j++)
        {
            if(BoardPos[i]==BoardPos[j]) //check if in the same row
                h++;
            if( (BoardPos[i]==BoardPos[j]-(j-i)) || (BoardPos[i]==BoardPos[j]+(j-
i)) ) //check for diagonal elements
                h++;
        }
    }
    return h;
}

void printSuccessFailure()
{
```

```
    printf("\n==================================");
    printf("\nSteepest-ascent");
    printf("\n==================================");
    printf("\nSuccess:%i \nFailure:%i ",Success1,Failure1);
    cout<<"\nAverage Success steps:"<<(int)
(float)successSteps1/Success1<<"\nAverage Failure steps:"<<(int)
(float)failureSteps1/Failure1;
    cout<<"\nSuccess rate: "<< (float) Success1/numberOfRuns *100<<"% \nFailure
rate:"<<(float) Failure1/numberOfRuns *100<<"%";


    printf("\n\n==================================");
    printf("\nHill climbing With sideway Moves");
    printf("\n==================================");
    printf("\nSuccess:%i \nFailure:%i ",Success2,Failure2);
    if(Failure2==0)
        cout<<"\nAverage Success steps:"<<(int)
(float)successSteps2/Success2<<"\nAverage Failure steps:"<<0;
    else
        cout<<"\nAverage Success steps:"<<(int)
(float)successSteps2/Success2<<"\nAverage Failure steps:"<<(int)
(float)failureSteps2/Failure2;
    cout<<"\nSuccess rate: "<< (float) Success2/numberOfRuns *100<<"% \nFailure
rate:"<<(float) Failure2/numberOfRuns *100<<"%";


    printf("\n\n==================================");
    printf("\nRandom Restart");
    printf("\n==================================");
    //printf("\nSuccess:%i \nFailure:%i ",Success3,Failure3);
    cout<<"\nAverage Restart Required:"<<(int)
(float)randomRestartCount/numberOfRuns;
    cout<<"\nAverage steps:"<<(int)(float)randomRestartSteps/numberOfCalls3;
    //cout<<"\nSuccess rate: "<< (float) Success3/numberOfRuns *100<<"% \nFailure
rate:"<<(float) Failure3/numberOfRuns *100<<"%";

    printf("\n\n==================================");
    printf("\nRandom Restart with Sideway Moves");
    printf("\n==================================");
    //printf("\nSuccess:%i \nFailure:%i ",Success4,Failure4);
    if(numberOfRuns==0)
        cout<<"\nAverage Restart Required:"<<1;//(int)
(float)randomRestartSideMoveCount/numberOfRuns;
    else
        cout<<"\nAverage Restart Required:"
<<ceil((float)randomRestartSideMoveCount/numberOfRuns);
    if(numberOfCalls4==0)
        cout<<"\nAverage steps:"<<randomRestartStepsSideMove;//(int)
(float)randomRestartStepsSideMove/numberOfCalls4;
    else
        cout<<"\nAverage steps:"<<(int)
(float)randomRestartStepsSideMove/numberOfCalls4;
    //cout<<"\nSuccess rate: "<< (float) Success4/numberOfRuns *100<<"% \nFailure
```

```
rate:"<<(float) Failure4/numberOfRuns *100<<"%";




}

void hillClimbingSteep()    // Consider ONLY the best move     //Example : [2 1 3
0]- failure  [2 0 3 0]-success
{
    int h=calculateH();
    int minH=h;
    //cout<<"\ninitial H val: "<<h;

    int nextPossibleMove[n];
    int k=12;
    int terminate=0;
    int loop=0;
    while(h>0 && terminate==0) //1 while loop generates all the possible
successors of the current node and chooses the best one
    {
        loop++;
        int flag=0;
        for(int i=0;i<n;i++)//for each col
        {
            for(int j=0;j<n;j++)// each row of col
            {

                if(j!=BoardPos[i])
                {
                    BoardPos[i]=j;
                    //printArray();
                    h=calculateH(); //update H value
                    //cout<<"\n"<<h;

                    if(h<minH)
                    {
                        minH=h;
                        flag=1;
                        for(int i=0;i<n;i++)
                            nextPossibleMove[i]=BoardPos[i];
                    }
                }
                copyArray();
            }
            //printf("\n==========================");
        }

        //printf("\n==========================");
        if(flag==0)//(minH<h || minH==h)
        {
            terminate=1;
            //printf("\nhere");
        }
```

```
        else
        {
            //printf("\nmin H val:%i",minH);
            //printf("\nNext move: ");
            //for(int i=0;i<n;i++)
                //cout<<nextPossibleMove[i]<<" " ;

            //change current BoardPos to best move
            h=minH;
            for(int i=0;i<n;i++)
            {
                BoardPos[i]=nextPossibleMove[i];
                BoardCopy[i]=nextPossibleMove[i];
            }
            k--;
        }
    }
    if(terminate==1)
    {
        //printf("\nfailed to find solution");
        failureSteps1+=loop;
        Failure1++;
    }
    else if(h==0)
    {
        //printf("\nSuccess!");
        successSteps1+=loop;
        Success1++;
    }
}

void hillClimbingSideMove()  //consider equal or less than H value with Max
allowed side Moves:100
{
    int h=calculateH();
    int minH=h;
    //cout<<"\ninitial H val: "<<h;

    int nextPossibleMove[n];
    int k=12;
    int sidewaysCount=0;
    int loop=0;

    int previousH=h;
    int previousSidewayH=0;
    int flag1=0;
    int terminate=0;
    while(h>0 && sidewaysCount<100 && terminate==0) //Allow Max of 100 side Moves
    {
        int flag2=0; //check if successor available
        loop++;
        int flag=0;
        for(int i=0;i<n;i++)//for each col
        {
```

```
            for(int j=0;j<n;j++)// each row of col
            {

                if(j!=BoardPos[i])
                {
                    BoardPos[i]=j;
                    //printArray();
                    h=calculateH(); //update H value
                    //cout<<"\n"<<h;

                    if(h<minH)
                    {
                        minH=h;
                        flag=1;
                        flag2=1;
                        for(int i=0;i<n;i++)
                            nextPossibleMove[i]=BoardPos[i];
                    }
                    else if(flag==0 && h==previousH)//h==minH)
                    {
                        flag1=0; //int
                        for(int k=0;k<boardIndex;k++) //check if current board
position already created?
                        {
                            int count=0;
                            for(int m=0;m<8;m++)
                            {
                                if(boardMoves[k][m]==BoardPos[m])
                                    count++;
                            }
                            if(count==8)
                            {
                                //printf("\n flag1=1");
                                flag1=1;
                            }
                        }

                        if(flag1==0) //proceed only if current board position is
new
                        {
                            flag2=1;
                            //printf("\n inside if");
                            minH=h;
                            for(int i=0;i<n;i++)
                                nextPossibleMove[i]=BoardPos[i];
                        }
                    }
                }
                copyArray();
            }
            //printf("\n==========================");
        }

        //printf("\n==========================");
```

```
        //printf("\nmin H val:%i",minH);
        if(flag2==0)
        {
            terminate=1;
        }
        //printf("\nNext move: ");
        //for(int i=0;i<n;i++)
            //cout<<nextPossibleMove[i]<<" " ;

        //update sidewaysCount if sideWay move was allowed
        if(flag==0 && minH==previousH)
        {
            if(previousSidewayH==minH) //update sidewaysCount for that particular
H value
                sidewaysCount++;
            else
            {
                previousSidewayH=minH;  //for new H value update previousSidewayH
and initialize sidewaysCount to 1
                sidewaysCount=1;
            }
        }
        //change current BoardPos to best move
        h=minH;
        previousH=minH;
        for(int i=0;i<n;i++)
        {
            BoardPos[i]=nextPossibleMove[i];
            BoardCopy[i]=nextPossibleMove[i];
            boardMoves[boardIndex][i]=nextPossibleMove[i];
        }
        boardIndex++;

        k--;
    }
    if(h==0)
    {
        //printf("\nSuccess!");
        successSteps2+=loop;
        Success2++;
    }
    else
    {
        //printf("\nFailed to find solution!");
        failureSteps2+=loop;
        Failure2++;
    }
    //printf("\n\n sidewaysCount:%i",sidewaysCount);
}

void randomRestart()
{
    if(Success1>previousSuccess1)
        Success3++;
```

```
    else
    {
        int currentSuccess1=Success1;
        int currentFailure1=Failure1;
        int currentSucessSteps1=successSteps1;
        int currentFailureSteps1=failureSteps1;
        while(Success1<=previousSuccess1)
        {
            randomRestartCount++;
            for(int i=0;i<n;i++)
            {
                BoardPos[i]=rand() % n;
                BoardCopy[i]=BoardPos[i];
                initialPos[i]=BoardPos[i];
            }
            hillClimbingSteep();
            numberOfCalls3++;
        } // end while

        randomRestartSteps+=(successSteps1-currentSucessSteps1)+(failureSteps1-
currentFailureSteps1);//(Success1-currentSuccess1)+(Failure1-currentFailure1);
        Success1=currentSuccess1;
        Failure1=currentFailure1;
        successSteps1=currentSucessSteps1;
        failureSteps1=currentFailureSteps1;
        Success3++;
    } //end else
    //printf("\nRandom restasrt Success!");
}

void randomRestartSideMove()
{
    if(Success2>previousSuccess2)
        Success4++;
    else
    {
        int currentSuccess2=Success2;
        int currentFailure2=Failure2;
        int currentSucessSteps2 =successSteps2;
        int currentFailureSteps2=failureSteps2;
        while(Success2<=previousSuccess2)
        {
            randomRestartSideMoveCount++;
            for(int i=0;i<n;i++)
            {
                BoardPos[i]=rand() % n;
                BoardCopy[i]=BoardPos[i];
                initialPos[i]=BoardPos[i];
            }
            hillClimbingSideMove();
            numberOfCalls4++;
        } // end while

        randomRestartStepsSideMove+=(successSteps2-currentSucessSteps2)+
```

```
(failureSteps2-currentFailureSteps2);//(Success2-currentSuccess2)+(Failure2-
currentFailure2);
        Success2=currentSuccess2;
        Failure2=currentFailure2;
        successSteps2=currentSucessSteps2;
        failureSteps2=currentFailureSteps2;
        Success4++;
    } //end else
    //printf("\nRandom restasrt with side Moves Success!");
}

int main()
{
    //cout<<"Hello World";
    cout<<"\n"<<"Enter n value:"<<"\n";
    std::cin >>n ;
    //cout<<"\n"<<"Enter initial board position:"<<"\n";
    //for(int i=0;i<n;i++)
        //std::cin >>BoardPos[i] ;

    //    Start iteration: 500
    for(int itr=0;itr<numberOfRuns;itr++)
    {
        boardIndex=0;
        for(int i=0;i<n;i++)
        {
            BoardPos[i]=rand() % n;
            BoardCopy[i]=BoardPos[i];
            initialPos[i]=BoardPos[i];
            boardMoves[boardIndex][i]=BoardPos[i];
        }
        boardIndex++;
        previousSuccess1=Success1;
        previousSuccess2=Success2;

        //printf("\nInitial board position: ");
        //printArray();

        //Call steepest-ascent hill climbing
        hillClimbingSteep();

        //printf("\n\n\n");
        initializeBoard();

        //Call Hill-climbing search with sideways move
        hillClimbingSideMove();

        //printf("\n\n\n");
        initializeBoard();

        //Call Random-restart  hill-climbing WITHOUT sideway move
        randomRestart();

        //printf("\n\n\n");
```

```
        initializeBoard();


        //Call Random-restart  hill-climbing with sideway move
        randomRestartSideMove();
    } //end For
    printSuccessFailure();
    return 0;
}
```