```csharp
using System.Collections.Generic;


namespace Assign_1
{
    /** Matthew Alunni
    *  5865647
    *  COSC 3P71
    *  Assignment 1 **/



    /** this class is for information about queens **/
    public class Position
    {
        // the cost of finding a specific colution
        public int Cost { get; set; }

        public int Line { get; set; }
        public int Row { get; set; }
        public Position Parent { get; set; }

        public Position(int Line, int Row, Position Parent)
        {
            this.Line = Line;
            this.Row = Row;
            this.Parent = Parent;
        }

        /** this method finds a solution by checking if a nearby queen has threats, then if a
         * solution is reached, it adds it to a list of solutions**/
        public void FindSolution(List<Solution> solutions,  int numberOfQueens, int cost)
        {
            System.Diagnostics.Debug.WriteLine(cost);

            Cost ++ ;

            if (Line == numberOfQueens) // last line (=number of queens) reached: solution
            {
                if (solutions == null)
                {
                    solutions = new List<Solution>();
                }

                var solution = new Solution
                {
                    Position = this
                };

                // calculate heuristic cost
                // the solution cost is the sum os the cost of each position found
                var pos = this;
                while (pos.Parent != null)
                {
                    solution.Cost += pos.Cost;
                    pos = pos.Parent;
                }

                solutions.Add(solution);
                return;
            }
            else
            {
                for (var r = 0; r < numberOfQueens; r++) // try all rows in next line
                {
                    // check threats for all queens in previous lines
                    var queenAbove = this;
                    while (!HasVerticalThreat(queenAbove, r)                          // vertical threat?
                           && !HasDiagonalLeftThreat(queenAbove, r)                   // diagonal threat left?
                           && !HasDiagonalRightThreat(queenAbove, r))                 // diagonal threat right?
                    {
                        queenAbove = queenAbove.Parent;                                // repeat check for all queens in previous lines
                    }

                    if (queenAbove.Line == 0)                                          // back to first queen - no threat found
                    {
                        new Position(Line + 1, r, this).FindSolution(solutions, numberOfQueens, Cost); // put queen on next line
                    }
                }
            }
        }

        /** this method checks if the queen at position has a vertical threat**/
        public bool HasVerticalThreat(Position queen, int row)
        {

            if (queen.Row >= 0 && row != queen.Row) // First row is Ok and different row is Ok
            {

                return false;
            }
            else
            {
                return true;
            }
        }


        /** this method checks if the queen at position has a diagonal threat**/
        public bool HasDiagonalLeftThreat(Position queen, int row)
        {

            if (row - queen.Row != Line + 1 - queen.Line)
            {

                return false;
            }
            else
            {
                return true;
            }
        }


        /** this method checks if the queen at position has a diagonal threat**/
        public bool HasDiagonalRightThreat(Position queen, int row)
        {

            if (queen.Row - row != Line + 1 - queen.Line)
            {

                return false;
            }
            else
            {
                Cost++;
                return true;
            }
        }
    }
}

```