```
 1  using System;
 2  using System.Collections.Generic;
 3  using System.Linq;
 4  using System.Text;
 5
 6  namespace NQueens
 7  {
 8      /// <summary>
 9      /// This class performs a hueristic search for the N Queens problem
10      /// </summary>
11      class InformedSearch
12      {
13          int gridSize;
14          int maxSteps;
15          int swapCounter = 0;
16          int boardCounter = 0;
17          int initialBoardCount = 3; //The generation of 3 initial boards was
              chosen after the testing shown in OptimalGenerationTester.cs. 3 came
              out better on average than any other
18          ChessBoard solution;
19          Printer printer;
20          Random rnd;
21
22          /// <summary>
23          /// Main constructor
24          /// </summary>
25          /// <param name="size">size of board to make</param>
26          /// <param name="ranSeed">seed for randomization</param>
27          public InformedSearch(int size, int ranSeed)
28          {
29              solution = HuristicSearch(size, ranSeed);
30          }
31          /// <summary>
32          /// Alternate constructor for testing, or when the number of initial
              boards needs to be changed
33          /// </summary>
34          /// <param name="size">size of board to make</param>
35          /// <param name="boardCount">number of boards to initialy push into the
               queue</param>
36          /// <param name="ranSeed">seed for randomization</param>
37          public InformedSearch(int size, int boardCount, int seed)
38          {
39              initialBoardCount = boardCount;
40              solution = HuristicSearch(size, seed);
41          }
42
43          /// <summary>
44          /// Main function. sets the base variables and calls the needed
              functions to find a solution
```

```
45          /// </summary>
46          /// <param name="size">size of board to make</param>
47          /// <param name="ranSeed">seed for random</param>
48          /// <returns>Either a solution or null</returns>
49          public ChessBoard HuristicSearch(int size, int ranSeed)
50          {
51              printer = new Printer();
52              gridSize = size;
53              maxSteps = size * size; // As explained in chapter 10, this is the
                  theoretical max it could take to find an answer
54              rnd = new Random(ranSeed);
55
56              for (int i = 0; i < maxSteps; i++)//Limits the number of generated
                  boards to maxSteps. Requiring more boards than this is
                  statistically impossible
57              {
58                  solution = MinConSearch();
59                  if (solution != null)
60                  {
61                      printer.Print(solution.board);
62                      Console.WriteLine(String.Format("\nsolution found! \nTotal
                        swaps: {0}\nBoards Generated: {1}", swapCounter,
                        boardCounter));
63                      Console.WriteLine("Seed: " + ranSeed);
64                      return solution;
65                  }
66              }
67
68              //On a board > 3, this should never be reached
69              Console.WriteLine("\nNo solution found. maxSteps insufficient");
70              return null;
71          }
72
73
74          /// <summary>
75          /// Main algorithm.
76          /// Generates a number of boards with randomly placed queens, default 3
                and queues them.
77          /// Chooses fittest board (based off the number of total hits all the
                queens on that board have)
78          /// Applies a switch based off the algorithm defined in Section 6.4 of
                Artifical Intelligence: A modern approach Third edition pg 221
79          /// </summary>
80          /// <returns>Null if failed. Proper chessboard if success</returns>
81          private ChessBoard MinConSearch()
82          {
83              Queue<ChessBoard> queue = new Queue<ChessBoard>();
84
85              for (int k = 0; k < initialBoardCount; k++)//Generate initial
```

```
                boards. Default 3
86              {
87                  ChessBoard baseBoard = new ChessBoard(gridSize);
88
89                  //Randomly fill baseboard
90                  int[] ranOrder = RanOrder(gridSize);
91                  for (int i = 0; i < baseBoard.board.Length; i++)
92                  {
93                      Coord nextQueen = new Coord(i, ranOrder[i]);
94                      baseBoard.AddQueen(nextQueen);
95                  }
96                  queue.Enqueue(baseBoard);
97
98                  boardCounter++;
99              }
100
101
102             while(queue.Count < maxSteps)
103             {
104                 queue = Prioritize(queue); //Sort
105                 ChessBoard currentBoard = queue.ElementAt(0); //Get best board
                      without removing from queue
106
107                 if (CheckSolved(currentBoard))//Check if solved
108                 {
109                     return currentBoard;
110                 }
111
112                 Coord worstQueen = FindWorstQueen(currentBoard);
113                 Coord bestSquare = FindBestSquare(currentBoard,
                      worstQueen.row);
114
115                 queue.Enqueue(MoveQueen(currentBoard, worstQueen, bestSquare));
116                 swapCounter++;
117             }
118             return null;
119         }
120
121         /// <summary>
122         /// Checks if the chessboard is solved.
123         /// This is done by iterating over every square checking it for fail
              conditions
124         /// </summary>
125         /// <param name="currentBoard">Board to check</param>
126         /// <returns>Whether the board is solved</returns>
127         private bool CheckSolved(ChessBoard currentBoard)
128         {
129             if(currentBoard.GetNumQueens() != gridSize)
130             {
```

```
131                 return false;
132             }
133             for (int k = 0; k < currentBoard.board.Length; k++)
134             {
135                 for (int j = 0; j < currentBoard.board.Length; j++)
136                 {
137                     if (currentBoard.board[k][j].isQueen && currentBoard.board
                        [k][j].GetNumHits() > 0)
138                     {
139                         return false;
140                     }
141                 }
142             }
143             return true;
144         }
145 
146 
147         /// <summary>
148         /// Sorts a  queue of chessboards from best to worst in terms of each
              boards queenHits
149         /// </summary>
150         /// <param name="queue">Queue to be sorted</param>
151         /// <returns>Sorted queue</returns>
152         private Queue<ChessBoard> Prioritize(Queue<ChessBoard> queue)
153         {
154             return new Queue<ChessBoard>(queue.OrderBy(board =>
                    board.queenHits));
155         }
156 
157         /// <summary>
158         /// Finds the coordinate of the 'best' square in the row of the given
              coord. The best square has the least hits
159         /// </summary>
160         /// <param name="currentBoard">Board on which to perform the search</
              param>
161         /// <param name="row">Row on which to search</param>
162         /// <returns>Coordinate where it would be best to place the next
              queen</returns>
163         private Coord FindBestSquare(ChessBoard currentBoard, int row)
164         {
165             List<Coord> bestSquares = new List<Coord>();
166             int benchMark = int.MaxValue;
167             for (int j = 0; j < currentBoard.board.Length; j++)
168             {
169                 if (!currentBoard.board[row][j].isQueen)
170                 {
171                     if (currentBoard.board[row][j].GetNumHits() < benchMark) //
                        Square is Better than best square
172                     {
```

```
173                         benchMark = currentBoard.board[row][j].GetNumHits();
174                         bestSquares.Clear();
175                         bestSquares.Add(new Coord(row,j));
176                     }
177                     else if (currentBoard.board[row][j].GetNumHits() ==
                        benchMark) //Square is as good as best square
178                     {
179                         bestSquares.Add(new Coord(row, j));
180                     }
181                 }
182 
183             }
184 
185             //return random element in list
186             return bestSquares[rnd.Next(bestSquares.Count)];
187         }
188 
189         /// <summary>
190         /// Finds the coordinate of the 'worst' queen on the board, meaning the
               queen with the most hits.
191         /// If there is a tie, choose a random queen with the same number of
              hits
192         /// </summary>
193         /// <param name="currentBoard">Board on which to perform the search</
              param>
194         /// <returns>Coord of the worst queen</returns>
195         public Coord FindWorstQueen(ChessBoard currentBoard)
196         {
197             List<Coord> worstQueens = new List<Coord>();
198             int benchMark = 0;
199             for (int k = 0; k < currentBoard.board.Length; k++)
200             {
201                 for (int j = 0; j < currentBoard.board.Length; j++)
202                 {
203                     if (currentBoard.board[k][j].isQueen)
204                     {
205                         if (currentBoard.board[k][j].GetNumHits() >
                        benchMark) //Square is WORSE than worst queen
206                         {
207                             benchMark = currentBoard.board[k][j].GetNumHits();
208                             worstQueens.Clear();
209                             worstQueens.Add(new Coord(k,j));
210                         }
211                         else if (currentBoard.board[k][j].GetNumHits() ==
                        benchMark) //Square is as bad as worst queen
212                         {
213                             worstQueens.Add(new Coord(k, j));
214                         }
215                     }
```

```
216
217                 }
218             }
219             //return random element in list
220             return worstQueens[rnd.Next(worstQueens.Count)];
221         }
222
223         /// <summary>
224         /// Moves queen from the worstQueen square to the bestSquare square
225         /// </summary>
226         /// <param name="currentBoard">Chessboard onwhich to apply the move</
            param>
227         /// <param name="worstQueen">Square containing the worst placed queen</
            param>
228         /// <param name="bestSquare">Square with the least hits in the same
            row</param>
229         /// <returns></returns>
230         public ChessBoard MoveQueen(ChessBoard currentBoard, Coord worstQueen,
            Coord bestSquare)
231         {
232             currentBoard.RemoveQueen(worstQueen);
233
234             currentBoard.AddQueen(bestSquare);
235
236             return currentBoard;
237         }
238
239         /// <summary>
240         /// Returns an array of random integers wherein no integer repeats.
241         /// Shuffle alogorthm is bassed of Knuth shuffle
242         /// </summary>
243         /// <param name="size">Desired size of list</param>
244         /// <returns>Ranomized array of </returns>
245         public int[] RanOrder(int size)
246         {
247             List<int> list = new List<int>();
248
249             for (int i = 0; i < size; i++) //Initialize
250             {
251                 list.Add(i);
252             }
253             for (int i = 0; i < list.Count; i++) //Randomize
254             {
255                 int rnd = this.rnd.Next(i, list.Count);
256                 int t = list[i];
257                 list[i] = list[rnd];
258                 list[rnd] = t;
259             }
260             return list.ToArray();
```

```
261         }
262
263
264         public ChessBoard GetSolution() { return solution; }
265
266         public int GetNumSwaps() { return swapCounter; }
267     }
268 }
269
```