COMP417 Artificial Intelligence
Term Project Report - Draft
Local and Constraint Satisfaction Search Algorithms for N-Queens Problem

Pantourakis Michail AM 2015030185
Proestakis Ioannis AM 2009030018
School of Electrical and Computer Engineering
Technical University of Crete

17 May 2018

1 Introduction

The n-queens problem, a combinatorial search problem, concerns the non-attacking (horizontally, vertically, and diagonally) placement of n queens on a $n \times n$ chessboard. The n-queens and similar Constraint Satisfaction Problems (CSPs) are classical examples of the limitations of simple backtracking search, with an exponential worst case time complexity that renders solving for large n impractical [1, 2]. Although many efficient heuristics have already been proposed for this problem [3–7], it still is a popular test bed for new Artificial Intelligence (AI) search problem methods. Whilst a toy problem per se, it has found some practical applications such as VLSI routing and testing, data compression, maximum full range communication and parallel optical computing [3, 4].

This problem has (at least) two variants depending on the desired number of solutions. A single solution can actually be found trivially without search, since explicit solutions exist $\forall\ n \geq 4$ [8]. On the other hand, finding all possible solutions is non-trivial. In this project we will focus on the former variant, implementing and comparing the performance of three different search algorithms: a backtracking method and two local search algorithms. We will first describe the problem mathematically and define the performance indicators for our comparison. In Section 3 we will describe the implemented algorithms, and we will subsequently illustrate and discuss their performance.

2 Problem Formulation

2.1 Mathematical Model

A naive formulation would allow any arrangement on the chessboard, resulting in a huge number of combinations (e.g. 3×10^{14} for $n = 8$). To reduce the state space, we distribute them so that each column contains only one queen. This restriction ensures that there will be no vertical conflicts, resulting in just 2057 candidate solutions (for $n = 8$) [5].

Therefore, let array $C = (c_1, ..., c_n : c_i \in \{1, ..., n\}\ \forall i \in \{1, ..., n\})$ be an n-queens placement, where i is the column number, and c_i is the row number of the i-th queen. To ensure that we have no horizontal or diagonal collisions, we need two more constraint equations. The first constraint is expressed as $c_i \neq c_j,\ \forall i \neq j$, namely no pairs of columns should be in the same row. The second constraint can be expressed as $|i - j| \neq |c_i - c_j|,\ \forall i \neq j$, namely no pair can be in the same diagonal.

Having defined the constraint and state representation formulas, the objective function that returns the total amount of direct and indirect collisions of C is given by:

$$n_c(C) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} f(c_i, c_j) \tag{1}$$

$$f(c_i, c_j) = \begin{cases} 1, & c_i = c_j \\ 1, & |i-j| = |c_i - c_j| \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

We notice that a brute-force approach in calculating the total amount of direct and indirect conflicts using Equation (1) would require $O(n^2)$ operations. However, a better approach has been formulated in Sosic and Gu [3]. Based on their work, we define the additional data structures: 1) an array D_{neg} containing the total number of queens per negative diagonal $D_{\text{neg}} = (d_{\text{neg},1}, ..., d_{\text{neg},2n-1} : d_i \in \{0, ..., n\}\ \forall i \in \{1, ..., 2n-1\})$, 2) a similarly defined array D_{pos} for the positive diagonals, 3) an array $R = (r_1, ..., r_n : r_i \in \{0, ..., n\}\ \forall i \in \{1, ..., n\})$ containing the total number of queens per row, and 4) an array $Q = (q_1, ..., q_n : q_i \in \{0, 1\}\ \forall i \in \{1, ..., n\})$ denoting whether a queen is attacked (1) or not (0).

Note that each queen belongs to a column i, a row c_i, a negative diagonal $i - c_i + n$ and a positive diagonal $i + c_i - 1$. Given the description above, we can now denote an $O(n)$ function for the number of *direct only* conflicts as:

$$n_c(D_{\text{neg}}, D_{\text{pos}}, R) = \sum_{i=1}^{2n-1} (d_{\text{neg,i}} - \min(1, d_{\text{neg,i}}) + d_{\text{pos,i}} - \min(1, d_{\text{pos,i}})) + \sum_{i=1}^{n} (r_i - \min(1, r_i)) \tag{3}$$

2.2 Performance Indicators

In order to assess the quality of our methods, we will employ three different performance indicators: 1) average execution time, 2) average memory used, and 3) average number of operations. Time and memory measurements will give an estimate of the time and space complexity of each method respectively. However, absolute time is hardware-dependent, and may even vary in the same machine due to other operating system processes. Hence, the average number of operations offers a hardware and software-independent time metric. See the next section for how operations and memory usage are estimated.

3 Methods

We selected three algorithms to implement in MATLAB: 1) forward checking with minimum remaining values (FC-MRV) algorithm as a CSP solver [5], 2) the min-conflict algorithm as a CSP/local search method [5], and 3) QS2, a swapping local search algorithm [3]. We chose these algorithms since previous results have illustrated their good performance, especially in comparison to simpler backtracking-based methods [3, 5].

3.1 FC-MRV Algorithm

Algorithm 1 contains the pseudo-code of our FC-MRV recursive implementation in MATLAB. The initial call is performed by Algorithm2. The successor state is determined by the recursive calls of FC-MRV. More specifically, the MRV method (line 8) chooses the unassigned column with the minimum number of remaining valid rows (choosing randomly if there are multiple). All valid rows are then checked one-by-one in lines 9-14. For each row, a new recursive call updates the current domain of the remaining unassigned columns (lines 1-5). The current domain is stored in a $n \times n$ matrix, which represents the board and containts 1 for valid positions and 0 for invalid ones. This cycle of recursions continues until either a complete solution is found, or until a column does not have any remaining valid positions left (which means that the previously selected row is not part of a valid configuration).

Performance Estimation: In order to estimate the number of operations and the memory used we make the following assumptions: 1) Each call of find(), UpdateDomain(), MRV() and initialization

of solution array C costs n operations. 2) Initialization of current domain matrix costs n^2 operations. 3) Current domain matrix costs n^2 arbitrary memory units, which is replicated for each recursive call. 4) Each recursion keeps two solution arrays (the old and new one) costing $2n$. 5) An array that keeps the unassigned queens costs n.

Algorithm 1 FC-MRV(currDomain, solution, row, column)

```
1: for all i : unassigned columns do
2:     UpdateDomain(currDomain, row, column, i) using Equation 2
3:     if no available rows left for column i then return empty solution
4:     end if
5: end for
6: if all columns have been assigned with a row then return solution
7: end if
8: newColumn = MRV(currDomain, solution)
9: for all newRow : available rows for newColumn do
10:    solution[newColumn] = newRow
11:    newSolution = FC-MRV(currDomain, solution, newRow, newColumn)
12:    if newSolution not empty then return newSolution
13:    end if
14: end for
15: return empty solution
```

Algorithm 2 FC-MRV-MAIN(n)

```
1: Initialize empty solution
2: Initialize currDomain = 1^{n×n}
3: Choose a random column j
4: for all i : rows do
5:     solution[j] = i
6:     newSolution = FC-MRV(currDomain, solution, i, j)
7:     if newSolution not empty then return newSolution
8:     end if
9: end for
```

3.2 Min-Conflicts Algorithm

Algorithm 3 contains the pseudo-code of our min-conflicts implementation in MATLAB. The successor operations of this method are in lines 14-15, where local changes upon the current state happen by randomly choosing an attacked queen and moving her to the row with the lowest number of conflicts. The above procedure is repeated until a maximum number of steps is reached or a solution is found. The outer while loop guarantees completeness.

Performance Estimation: In order to estimate the number of operations and the memory used we make the following assumptions: 1) Each call of checkDiagonals(), countDiagConflicts(), findAttackedQueens(), find(), min() and randperm() cost n operations. 2) Initialization of R and array that keeps conflicts per row (line 12) cost n operations. 3) D_{pos} and D_{neg} cost $2n$ arbitrary memory units, whereas the rest of the arrays cost n units. With these assumptions the space complexity is linear, estimated as $9n$.

Algorithm 3 MIN-CONFLICTS(n)

```
1: Set a random permutation as initial solution C
2: Calculate D_pos, D_neg and R for initial solution
3: Calculate n_c using Equation 3
4: Calculate attacked queens in Q
5: steps = 0
6: Define constants maxSteps
7: while n_c > 0 do
8:     while steps < maxSteps do
9:         steps++
10:        Choose an attacked Queen i randomly
11:        for all row : 1 to n do
12:            Calculate conflicts per row
13:        end for
14:        Choose the row that minimizes the conflicts (choose randomly if there are more than one)
15:        Perform change updating C, D_pos, D_neg and R
16:        if n_c = 0 then
17:            return solution C
18:        else
19:            Recalculate attacked queens in Q
20:        end if
21:    end while
22:    Set a random permutation as initial solution C
23:    Calculate D_pos, D_neg and R for initial solution
24:    Calculate n_c using Equation 3
25:    Calculate attacked queens in Q
26:    steps = 0
27: end while
```

3.3 QS2: Swapping Queens

Algorithm 4 contains the pseudo-code of our min-conflicts implementation in MATLAB. The successor operations take place in lines 13-14. For each attacked queen, we randomly choose another queen and we check whether swapping their rows will decrease n_c. If it does, then the state is updated, performing the swap. The limit of line 18 is used as a way of preventing breaking the for loop prematurely and recalculating the attacked queens unnecessarily [3]. Again the outer while loop guarantees that an optimal solution will be returned.

Performance Estimation: In order to estimate the number of operations and the memory used we make the following assumptions: 1) Each call of checkDiagonals(), countDiagConflicts(), findAttackedQueens(), find(), randperm() cost n operations. 2) For each iteration of for loop in line 10, we add a single operation. 3) D_{pos} and D_{neg} cost $2n$ arbitrary memory units, whereas the rest of the arrays cost n units. With these assumptions the space complexity is linear, estimated as $7n$.

Algorithm 4 QS2(n)

```
1:  Set a random permutation as initial solution C
2:  Calculate D_pos and D_neg for initial solution
3:  Calculate n_c using Equation 3
4:  Calculate attacked queens in Q
5:  steps = 0
6:  Define constants maxSteps and C1
7:  limit = C1 ×n_c
8:  while n_c > 0 do
9:      while steps < maxSteps do
10:         for all i : attacked Queens do
11:             steps++
12:             Choose another column j
13:             if swapping rows between column i and column j decreases conflicts then
14:                 Perform swap updating C, D_pos and D_neg
15:                 if n_c = 0 then
16:                     return solution C
17:                 else
18:                     if n_c < limit then
19:                         limit = C1 ×n_c
20:                         Recalculate attacked queens in Q
21:                         Break loop
22:                     end if
23:                 end if
24:             end if
25:         end for
26:     end while
27:     Set a random permutation as initial solution C
28:     Calculate D_pos and D_neg for initial solution
29:     Calculate n_c using Equation 3
30:     Calculate attacked queens in Q
31:     steps = 0
32:     limit = C1 ×n_c
33: end while
```

References

[1] R. Sosic and J. Gu. "A Polynomial Time Algorithm for the N-Queens Problem". In: *SIGART Bull.* 1.3 (Oct. 1990), pp. 7–11. ISSN: 0163-5719.

[2] H. S. Stone and J. M. Stone. "Efficient search techniques - an empirical study of the N-Queens problem". In: *IBM Journal of Research and Development* 31.4 (1987), pp. 464–474.

[3] R. Sosic and J. Gu. "Fast search algorithms for the n-queens problem". In: *IEEE Transactions on Systems, Man, and Cybernetics* 21.6 (1991), pp. 1572–1576.

[4] X. Hu, R. C. Eberhart, and Y. Shi. "Swarm intelligence for permutation optimization: a case study of n-queens problem". In: *Swarm intelligence symposium, 2003. SIS'03. Proceedings of the 2003 IEEE*. IEEE. 2003, pp. 243–246.

[5] S. J. Russell, P. Norvig, et al. *Artificial intelligence*. Prentice Hall/Pearson Education, 2003.

[6] M. R. Engelhardt. "A group-based search for solutions of the n-queens problem". In: *Discrete Mathematics* 307.21 (2007), pp. 2535–2551. ISSN: 0012-365X.

[7] K. Agarwal, A. Sinha, and M. H. Bindu. "A novel hybrid approach to N-queen problem". In: *Advances in Computer Science, Engineering & Applications*. Springer, 2012, pp. 519–527.

[8] B. Bernhardsson. "Explicit solutions to the N-queens problem for all N". In: *ACM SIGART Bulletin* 2.2 (1991), p. 7.