COMP417 Artificial Intelligence
Term Project Report
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

31 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]. For the exact code implementation, please refer to the accompanying source code.

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 contains 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 = $\mathbf{1}^{n \times n}$
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 $9n$.

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 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**

---

4 Results

Table 1 contains the real execution time of our algorithms for various number of queens, whereas Figure 1 depicts the same results graphically. The results clearly show that FC-MRV approximates an exponential time complexity, which is the theoretical worst case time complexity of backtracking-based methods [5]. On the other hand, the min-conflicts and QS2 algorithms run in polynomial time, exhibiting a significantly better performance than FC-MRV for $n \geq 60$ queens. For a small number of queens, QS2 shows a worse temporal performance, probably because its time complexity constants dominate its running time. Nevertheless, the swapping mechanism of QS2 clearly outperforms the min-conflicts search for many queens. Similar curves can be seen in Table 2 and Figure 2, where the average number of operations are depicted.

Finally, Table 3 and Figure 3 contain our estimated measurements of each algorithm's memory

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 $\times n_c$
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 $\times n_c$
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 $\times n_c$
33: **end while**

---

usage. Again, min-conflicts and QS2 obviously have a linear space complexity (as expected from local search algorithms), since they only need to store the arrays C, D_{pos}, D_{neg}, R and Q, whose length linearly depends on n. In contrast, FC-MRV exhibits a huge space cost, even for smaller problem sizes, due to the rapidly increasing number of recursive calls.
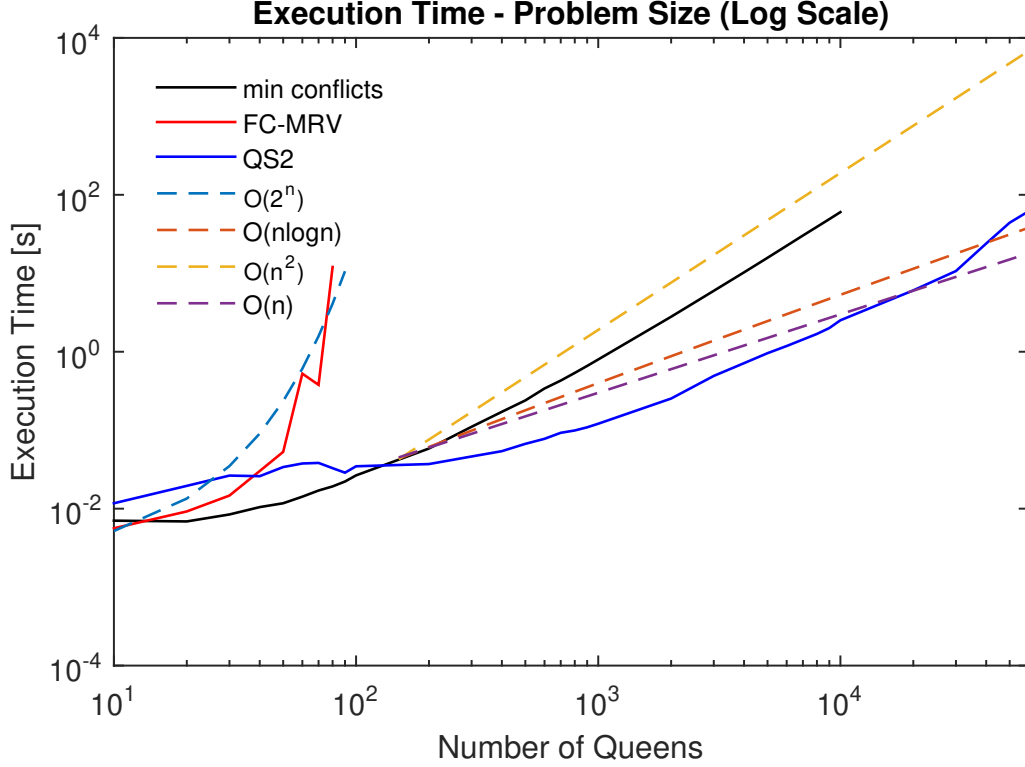

Figure 1: Real execution time as a function of n from the data of Table 1. To reveal the scaling of each algorithm's time complexity, we also draw representative exponential and polynomial functions.

5 Conclusions

In this report we implemented in MATLAB three different methods for solving the n-queens problem. Our results clearly illustrated the superior performance of two local search algorithms (min-conflicts and QS2) against a typical backtracking-based method (FC-MRV). Both local search algorithms started from a random state, ensuring that no two queens can be found in the same column and row. A better initialization may have been a partial queen placement with no conflicts as shown in Sosic and Gu [3]. Finally, more general methods, such as bioinspired algorithms and neural networks, can also be used with success [9].
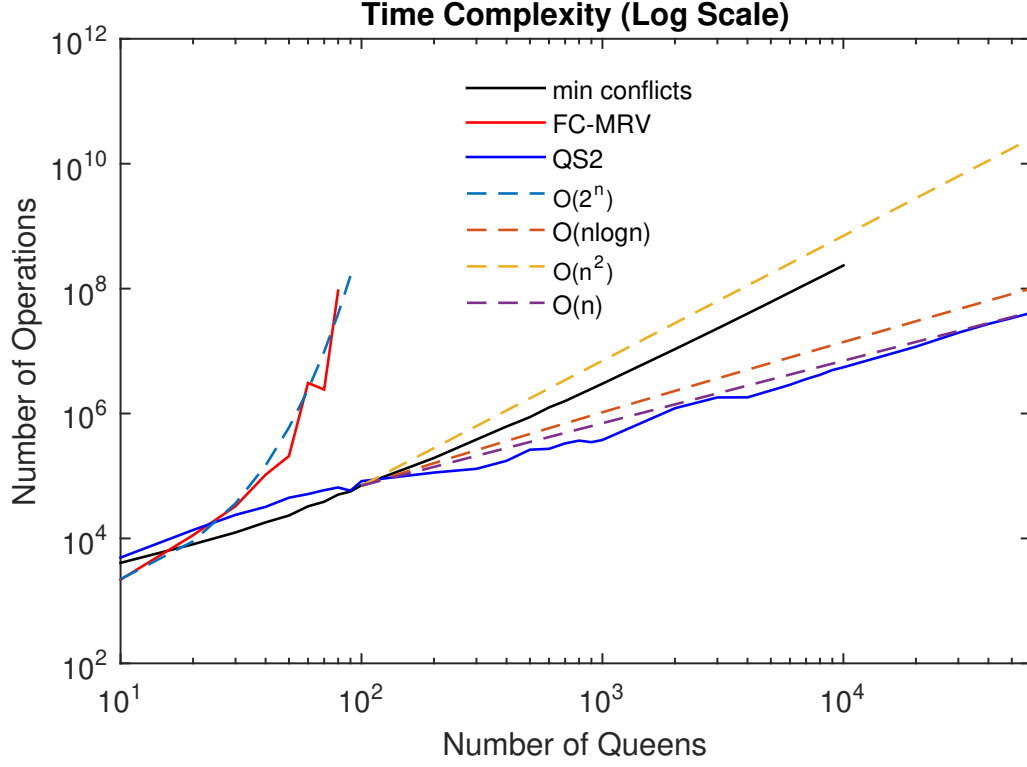
Figure 2: Number of operations as a function of n from the data of Table 2. To reveal the scaling of each algorithm's time complexity, we also draw representative exponential and polynomial functions.
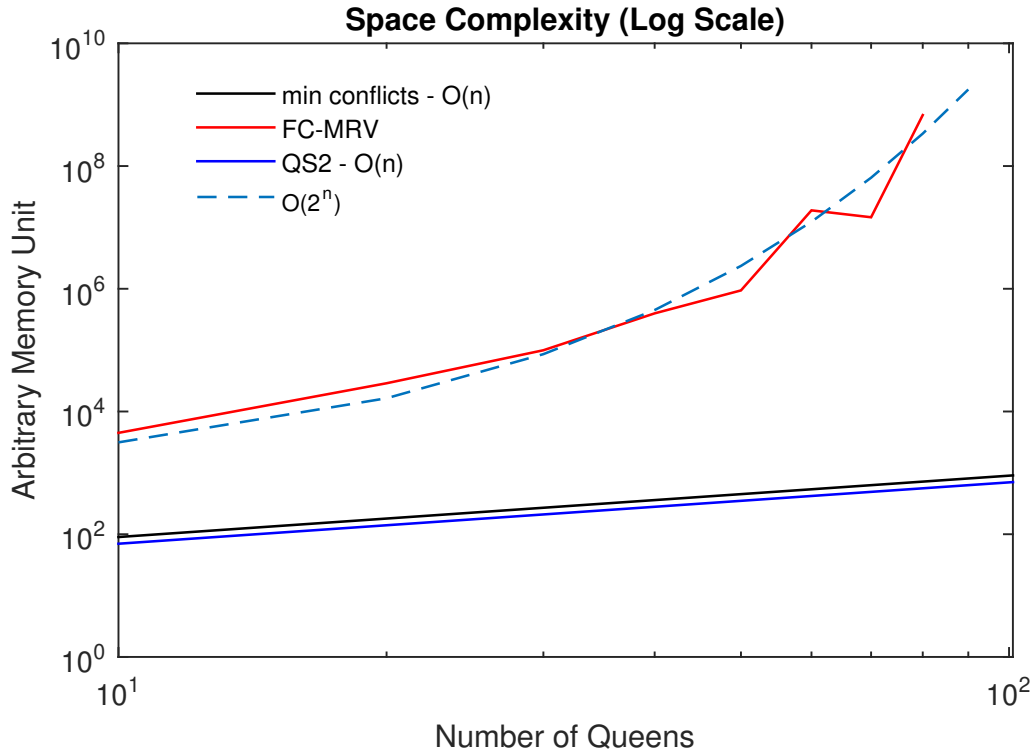

Figure 3: Memory usage as a function of n from the data of Table 3. To reveal the scaling of FC-MRV we also draw a representative exponential function. The min-conflicts and QS2 curves are linear as explained in Section 3.

Number of Queens n	Time (s)		
	FC-MRV	Min-conflicts	QS2
10	$5.625 \cdot 10^{-3}$	$7.031 \cdot 10^{-3}$	$1.172 \cdot 10^{-2}$
20	$9.219 \cdot 10^{-3}$	$6.875 \cdot 10^{-3}$	$1.953 \cdot 10^{-2}$
30	$1.469 \cdot 10^{-2}$	$8.438 \cdot 10^{-3}$	$2.641 \cdot 10^{-2}$
40	$3.016 \cdot 10^{-2}$	$1.047 \cdot 10^{-2}$	$2.594 \cdot 10^{-2}$
50	$5.297 \cdot 10^{-2}$	$1.172 \cdot 10^{-2}$	$3.391 \cdot 10^{-2}$
60	$5.259 \cdot 10^{-1}$	$1.422 \cdot 10^{-2}$	$3.766 \cdot 10^{-2}$
70	$3.783 \cdot 10^{-1}$	$1.703 \cdot 10^{-2}$	$3.828 \cdot 10^{-2}$
80	$1.203 \cdot 10^{1}$	$1.922 \cdot 10^{-2}$	$3.281 \cdot 10^{-2}$
90	-	$2.219 \cdot 10^{-2}$	$2.875 \cdot 10^{-2}$
100	-	$2.656 \cdot 10^{-2}$	$3.469 \cdot 10^{-2}$
200	-	$5.844 \cdot 10^{-2}$	$3.703 \cdot 10^{-2}$
300	-	$1.106 \cdot 10^{-1}$	$4.625 \cdot 10^{-2}$
400	-	$1.716 \cdot 10^{-1}$	$5.406 \cdot 10^{-2}$
500	-	$2.397 \cdot 10^{-1}$	$6.703 \cdot 10^{-2}$
600	-	$3.400 \cdot 10^{-1}$	$7.766 \cdot 10^{-2}$
700	-	$4.291 \cdot 10^{-1}$	$9.266 \cdot 10^{-2}$
800	-	$5.373 \cdot 10^{-1}$	$9.922 \cdot 10^{-2}$
900	-	$6.597 \cdot 10^{-1}$	$1.089 \cdot 10^{-1}$
1000	-	$7.973 \cdot 10^{-1}$	$1.209 \cdot 10^{-1}$
2000	-	$2.793 \cdot 10^{0}$	$2.533 \cdot 10^{-1}$
3000	-	$5.979 \cdot 10^{0}$	$4.919 \cdot 10^{-1}$
4000	-	$1.028 \cdot 10^{1}$	$7.152 \cdot 10^{-1}$
5000	-	$1.575 \cdot 10^{1}$	$9.583 \cdot 10^{-1}$
6000	-	$2.244 \cdot 10^{1}$	$1.183 \cdot 10^{0}$
7000	-	$3.026 \cdot 10^{1}$	$1.430 \cdot 10^{0}$
8000	-	$3.907 \cdot 10^{1}$	$1.687 \cdot 10^{0}$
9000	-	$4.900 \cdot 10^{1}$	$2.004 \cdot 10^{0}$
10000	-	$6.019 \cdot 10^{1}$	$2.517 \cdot 10^{0}$
20000	-	-	$6.058 \cdot 10^{0}$
30000	-	-	$1.069 \cdot 10^{1}$
40000	-	-	$2.396 \cdot 10^{1}$
50000	-	-	$4.413 \cdot 10^{1}$
60000	-	-	$6.193 \cdot 10^{1}$

Table 1: Execution time of all algorithms for n-queens problem. For each problem size $n = \{10, 20, ..., 100, 200, ..., 1000, 2000, ..., 10000, ..., 60000\}$ results are averaged over 100 tries. FC-MRV and min-conflicts ran for size of up to 80 and 10000 queens respectively, due to their large running times.

Number of Queens n	Number of Operations		
	FC-MRV	Min-conflicts	QS2
10	2190	4062	4934
20	11260	8069	13632
30	32809	12452	23830
40	104653	18076	32047
50	208325	23275	44853
60	3089588	32664	51337
70	2419038	38651	59200
80	92664845	50340	65395
90	-	56345	58184
100	-	71095	82639
200	-	194120	113384
300	-	383505	130177
400	-	619140	174827
500	-	875625	263529
600	-	1250970	272058
700	-	1585535	332380
800	-	2013400	368229
900	-	2485260	347673
1000	-	3004600	376858
2000	-	10733900	1215070
3000	-	22950600	1809304
4000	-	39726200	1814601
5000	-	60948250	2336182
6000	-	86990400	2883814
7000	-	117443900	3560520
8000	-	151767200	4162563
9000	-	191103300	4962452
10000	-	234658500	5496021
20000	-	-	11752792
30000	-	-	19648046
40000	-	-	27203392
50000	-	-	33736673
60000	-	-	40844200

Table 2: Number of operations of all algorithms for n-queens problem. For each problem size $n = \{10, 20, ..., 100, 200, ..., 1000, 2000, ..., 10000, ..., 60000\}$ results are averaged over 100 tries. FC-MRV and min-conflicts ran for size of up to 80 and 10000 queens respectively, due to their large running times. See Section 3 for how the number of operations is estimated in each method.

Number of Queens n	Memory (abritrary units)		
	FC-MRV	Min-conflicts	QS2
10	4486	90	70
20	28853	180	140
30	99574	270	210
40	397412	360	280
50	940120	450	350
60	18963989	540	420
70	14634236	630	490
80	676954812	720	560
90	-	810	630
100	-	900	700
200	-	1800	1400
300	-	2700	2100
400	-	3600	2800
500	-	4500	3500
600	-	5400	4200
700	-	6300	4900
800	-	7200	5600
900	-	8100	6300
1000	-	9000	7000
2000	-	18000	14000
3000	-	27000	21000
4000	-	36000	28000
5000	-	45000	35000
6000	-	54000	42000
7000	-	63000	49000
8000	-	72000	56000
9000	-	81000	63000
10000	-	90000	70000
20000	-	-	140000
30000	-	-	210000
40000	-	-	280000
50000	-	-	350000
60000	-	-	420000

Table 3: Memory usage of all algorithms for n-queens problem. For each problem size $n = \{10, 20, ..., 100, 200, ..., 1000, 2000, ..., 10000, ..., 60000\}$ results are averaged over 100 tries. FC-MRV and min-conflicts ran for size of up to 80 and 10000 queens respectively, due to their large running times. See Section 3 for how memory usage is quantified in each method.

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 and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 2002.

[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.

[9] M. G. Kaosar, M. Shorfuzzaman, and S. Ahmed. "A Novel Approach to Solving N-Queens Problem". In: *Proceedings of the 8th World Multi conference on Systemic, Cybernetics and Informatics (SCI 2004), Orlando, Florida, USA.* 2004, pp. 1–5.