Homework 1. Optimization and Complexity

Daniel Fernández Martínez

April 23, 2021

Abstract

In this first homework, we are asked for analyzing different approaches for solving the N-queens and N-rooks problems. N-queen (or rooks) problem consists of placing N non-attacking queens (or rooks) on an N×N chessboard. A brute-force approach and an enhanced backtracking algorithm will be implemented to solve this task.

1 N-Queen

1.1 Does a solution exist for small N?

Let´s take as an example the case in which N = 3. For this configuration, we will try to site 3 queens on the chessboard avoiding mutual check.

A queen attacks other figure if they are placed in the same row, column or diagonal. Let´s place the first queen, for example, in the 3A square:

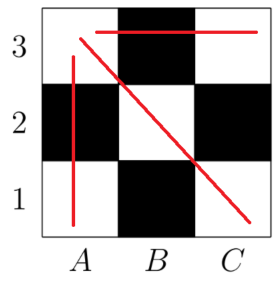

Figure 1: 1st Queen placed on A3 square

The red lines in the figure show the positions under attack. It is quite clear that the only "safe" remaining positions are B1 and C2. Now, if we place another queen on the B1 square, we obtain:

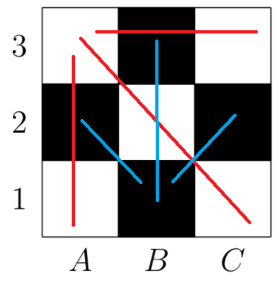

Figure 2: 2nd Queen added on B1 square

Where the blue lines show the positions under attack by the B1 queen. Consequently, in a 3x3 chessboard, we will be able to place only two queens in the best scenarios, although only one queen sited on B2 square will be enough to attack all the remaining positions.

One can conclude that there exists solution for all natural numbers N with the exception of N = 2 and N = 3.

1.2 Consider a value of N for which there exists a solution. How many queens can be sited on the chessboard?

For N = 4, which is the simplest case for which there exists a solution, the maximum number of queens that can be placed is 4, as it is shown in the following figures.

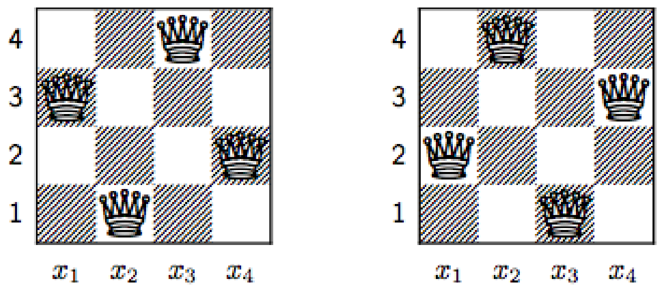

Figure 3: 4x4 Nqueen solutions

Which, in practice, are symmetric solutions, so there is actually only one “fundamental” solution.

1.3 In how many ways can N queens be sited on a NxN chessboard (allowing mutual check)?

For N = 4, we have to choose positions for 4 identical queens from 16 different possible squares. That means that there are:

$$_{16}C_4 = \frac{16!}{4! \cdot (16-4)!} = \frac{16!}{4! \cdot 12!} = \frac{16 \cdot 15 \cdot 14 \cdot 13}{4 \cdot 3 \cdot 2 \cdot 1} = 1820$$

ways to arrange the 4 queens on the chessboard.

Following the same procedure, for an 8×8 board, we have to choose positions for 8 identical queens from 64 different squares, which can be done in $_{64}C_8$ ways.

Genaralizing, there are $_{N^2}C_N$ ways to site N queens on a NxN chessboard.

1.4 Using the previous results, write a program to enumerate all the possible configuration, removing all of them in which there is at least a check. Simulate the N = 4 case.

To simplify a little bit the calculations, we can enforce one queen in each row through the use of N nested loops. That means that for N = 4, we would only have to look at $4^4 = 256$ configurations. The next code evaluates each of these 256 possible arrangements and discard all the cases in which there is at least a check.

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

# define N 4

int place_queen(int []);
void brute(int []);
int row[N];
int count1 = 0;
int count2 = 0;

int main() {
    brute(row);
    return 0;
}

void brute(int sample[]) {
    for (int i = 0; i<N; i++)
    {
        sample[0] = i;
        for (int j = 0; j<N; j++)
        {
            sample[1] = j;
            for (int k = 0; k<N; k++)
            {
                sample[2] = k;
                for (int l=0; l<N; l++)
                {
                    count1++;
                    sample[3] = l;
                    if (place_queen(sample))
                    {
                        count2++;
                        printf("Solution %d:\n", count2);
                        printf("%d%d%d%d\n\n", i+1, j+1, k+1, l+1);
                    }
                }
            }
        }
    }
    printf("\nTotal configurations: %d\n", count1);
    printf("Valid configurations: %d\n", count2);
}

int place_queen(int perm[N])
{
    int i, j;
    for (i=0; i<N; i++)
    {
        for (j=i+1; j<N; j++) {
            if ((perm[i] == perm[j]) || (abs(perm[i]-perm[j])) == abs(i-j))
                return false;
        }
    }
    return true;
}
```

The key part of this program is the `place_queen()` function that tests, given a permutation, if it is valid or not. It is straight-forward to test if they are in the same column, but testing if they share the same diagonal is a little bit more complicated. Suppose that two queens are located at (i, j) and (k, l), respectively, and both are neither in the same row nor in same column. We have to test, also, if they are in the same diagonal. The key is that two queens are on the same diagonal if and only if:

$$|j - l| = |i - k|$$

Let´s see a visual example to show this idea.

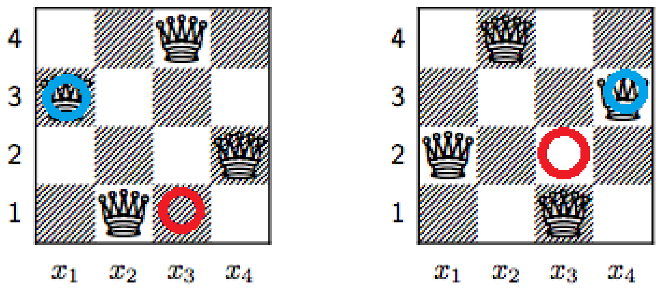

Figure 4: 4x4 Nqueen solutions

Imagine that the blue circle is Q_1 and Q_2 is the red one. On the left chessboard, Q_1 is on the $(3, 1)$ position and Q_2 on the $(1, 3)$. Visually, it is clear that they share the same diagonal. Let´s put in practice the above expression:

$$|j - l| = |1 - 3| = |i - k| = |3 - 1| = 2$$

For the right hand-side image, we get:

$$|j - l| = |4 - 3| = |i - k| = |3 - 2| = 1$$

We have validated a simple tool to test if two queens are located in the same diagonal, which has been included in the programs developed.

Using the previous code, the execution results for N=4 are displayed below:

```
Solution 1:
2413

Solution 2:
3142


Total configurations: 256
Valid configurations: 2
```

Where each number corresponds with the pair row - column in which the queens must be placed. For example, solution “2413” means that the first queen must be sited on the second row - first column square, the second queen on the fourth row - second column square, the third queen on the first row - third column square and, lastly, the fourth queen have to be placed on the third row - fourth column square.

1.5 Program the backtracking algorithm. Find all the possible solutions for the N = 4 and N = 5 cases.

Let´s briefly remind what are the principal ideas for solving the N-queen problem using backtracking.

1. Starting with a queen in the first row, first column, we search left to right for a valid position to place another queen in the next available row.
2. If we find a valid position in this row, we recursively start again on the next row.
3. If we don’t find a valid position in this row, we backtrack to the previous row.
4. When the counter size gets to N, we will have placed N queens on the board, and therefore have a solution.

The next code implements this algorithm.

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

# define N 4

void try(int);
int place_queen(int);
int column[N];
int count = 0;

int main()
{
    try(0);
    printf("\nValid configurations: %d", count);
}

void try(int n)
{
    int i, j;
    for (i=0;i<N;i++)
    {
        column[n] = i;

        if (place_queen(n)==true && n != N-1)
            try(n+1);

        if (place_queen(n) == true && n == N-1)
        {
            count++;
            printf("Solution %d:\n", count);
            for (j = 0; j<N; j++)
                printf("%d",column[j]+1);
            printf("\n\n");
        }
    }
}
```

```

int place_queen(int n)
{
    int x;

    for (x=0; x<n; x++)
        if ((column[x] == column[n]) || (abs(column[x]-column[n]) == abs(x-n)))
            return false;
    return true;
}
```

For N=4 and N=5, the results obtained are displayed below:

```
Backtracking for N = 4 Queen.

Solution 1:
2413

Solution 2:
3142


Valid configurations: 2

Backtracking for N = 5 Queen.

Solution 1:
13524

Solution 2:
14253

Solution 3:
24135

Solution 4:
25314

Solution 5:
31425

Solution 6:
35241

Solution 7:
41352

Solution 8:
42531

Solution 9:
52413

Solution 10:
53142


Valid configurations: 10
```

1.6 Use open source libraries to solve the problem. What algorithms do they use?

I only have found an open source library to solve the N-queen problem, using a constrain programming (CP) approach, the `or-tools` package in Python.

Taking, literally, from this website:

"A CP solver works by systematically trying all possible assignments of values to the variables in a problem, to find the feasible solutions. In the 4-queens problem, the solver starts at the leftmost column and successively places one queen in each column, at a location that is not attacked by any previously placed queens."

For more information about the CP solver and the solution, visit the website. The complete code example is displayed below:

```python
# Solving N-queens problem using Google or-tools

from ortools.constraint_solver import pywrapcp


def main(N=8):
    # We create our solver.
    solver = pywrapcp.Solver("N-queens")

    # We should now declare our queens' coordinates as variables.
    # We want our variables to be in range of 0..N.
    # For that we are going to use solver.IntVar.
    # Lets create and name them q0, q1 .. q(N-1) since we have N queens.
    queens = [solver.IntVar(0, N - 1, "q%i" % i) for i in range(N)]

    # Time to add our constraints to our solver.
    # We have 2 constraints. We have to ensure that no two queens are:
    #  1- On the same column: No two values of the array can be the same.
    #  2- On the same diagonal: The values plus(and minus) the indices should
    # be all different.
    # "solver.AllDifferent" enforces a set of variables to take distinct values
    solver.Add(solver.AllDifferent(queens))
    solver.Add(solver.AllDifferent([queens[i] + i for i in range(N)]))
    solver.Add(solver.AllDifferent([queens[i] - i for i in range(N)]))

    # Using the "solver.Phase", we tell the solver what to solve.
    tree = solver.Phase(queens,
                        solver.INT_VAR_SIMPLE,
                        solver.INT_VALUE_SIMPLE)

    # After creating the search tree we can now begin our search
    solver.NewSearch(tree)
    solution_count = 0
    # We can print our solutions while iterating over them as:
    while solver.NextSolution():
        solution_count += 1
        solution = [queens[i].Value() for i in range(N)]
        print "Solution %d:" % solution_count, solution
        for i in range(N):
            for j in range(N):
                if solution[i] == j:
                    print "Q",
                else:
                    print "_",
            print
        print

    # We have reached the end of our search
    # Documentation says:
    # It is just better practice to finish the search with the method EndSearch
    solver.EndSearch()

if __name__ == '__main__':
    main(N=8)
```

2 N-Rooks

In this section, we will answer all the previous questions, but using Rooks instead of Queens.

2.1 Does a solution exist for small N?

A rook attacks other figure if they share the same row or column. So, in contrast of the N-queen problem, there exist solutions for all natural numbers. For example, for N=2, there are two possible configurations (one fundamental), in which the rooks are placed along the diagonals of the 2x2 chessboard.

2.2 Consider a value of N for which there exists a solution. How many rooks can be sited on the chessboard?

For N = 4, the maximum number of rooks that can be sited is 4.

2.3 In how many ways can N rooks be sited on a NxN chessboard (allowing mutual check)?

The previous answer for the N-queen problem can also be applied for this case, as the permutations only depends on the size of the chessboard.

2.4 Using the previous results, write a program to enumerate all the possible configuration, removing all of them in which there is at least a check. Simulate the N = 4 case.

Re-using the routine for the N-queen problem makes sense, as we only have to modify the function used for test the check condition. Specifically, we will remove the condition used for ckeck if the diagonals are safe.

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

# define N 4

int place_rook(int []);
void brute(int []);
int row[N];
int count1 = 0;
int count2 = 0;

int main() {
    brute(row);
    return 0;
}

void brute(int sample[]) {
    for (int i = 0; i<N; i++)
    {
        sample[0] = i;
        for (int j = 0; j<N; j++)
        {
            sample[1] = j;
            for (int k = 0; k<N; k++)
            {
                sample[2] = k;
                for (int l=0; l<N; l++)
                {
                    count1++;
                    sample[3] = l;
                    if (place_rook(sample))
                    {
                        count2++;
                        printf("Solution %d:\n", count2);
                        printf("%d%d%d%d\n\n", i+1, j+1, k+1, l+1);
                    }
                }
            }
        }
    }
    printf("\nTotal configurations: %d\n", count1);
    printf("Valid configurations: %d\n", count2);
}

int place_rook(int perm[N])
{
    int i, j;
    for (i=0; i<N; i++)
    {
        for (j=i+1; j<N; j++) {
            if (perm[i] == perm[j])
                return false;
        }
    }
    return true;
}
```

The execution results are displayed below. It is quite obvious that for this problem, which is less restrictive, there are more solutions than for the N-queen case.

```
Solution 1:
1234

Solution 2:
1243

Solution 3:
1324

Solution 4:
1342

Solution 5:
1423

Solution 6:
1432

Solution 7:
2134

Solution 8:
2143

Solution 9:
2314

Solution 10:
2341

Solution 11:
2413

Solution 12:
2431

Solution 13:
3124

Solution 14:
3142
```

```
Solution 15:
3214

Solution 16:
3241

Solution 17:
3412

Solution 18:
3421

Solution 19:
4123

Solution 20:
4132

Solution 21:
4213

Solution 22:
4231

Solution 23:
4312

Solution 24:
4321


Total configurations: 256
Valid configurations: 24
```

2.5 Program the backtracking algorithm. Find all the possible solutions for the N = 4 and N = 5 cases.

The same reasoning can be extended for the backtracking algorithm. The only modification made here has been the `if` condition of the `place_rook()` function.

```c
// Nrook - Backtracking. Daniel Fern ndez Mart nez

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

# define N 4

void try(int);
int place_rook(int);
int column[N];
int count = 0;

int main()
{
    printf("Backtracking for N = %d Rook.\n\n",N);
    try(0);
    printf("\nValid configurations: %d", count);
}

void try(int n)
{
    int i, j;
    for (i=0;i<N;i++)
    {
        column[n] = i;

        if (place_rook(n)==true && n != N-1)
            try(n+1); // Backtracking

        if (place_rook(n) == true && n == N-1)
        {
            count++;
            printf("Solution %d:\n", count);
            for (j = 0; j<N; j++)
                printf("%d",column[j]+1);
            printf("\n\n");
        }
    }
}

int place_rook(int n)
{
    int x;

    for (x=0; x<n; x++)
        if ((column[x] == column[n]))
            return false;
    return true;
}
```

```
Backtracking for N = 4 Rook.

Solution 1:
1234

Solution 2:
1243

Solution 3:
1324

Solution 4:
1342

Solution 5:
1423

Solution 6:
1432

Solution 7:
2134

Solution 8:
2143

Solution 9:
2314

Solution 10:
2341

Solution 11:
2413

Solution 12:
2431
```

```
Solution 13:
3124

Solution 14:
3142

Solution 15:
3214

Solution 16:
3241

Solution 17:
3412

Solution 18:
3421

Solution 19:
4123

Solution 20:
4132

Solution 21:
4213

Solution 22:
4231

Solution 23:
4312

Solution 24:
4321


Valid configurations: 24
Backtracking for N = 5 Rook.

Solution 1:
12345

Solution 2:
12354

Solution 3:
12435

Solution 4:
12453

.....

Solution 118:
54231

Solution 119:
54312
```

```
Solution 120:
54321


Valid configurations: 120
```

2.6 Use open source libraries to solve the problem. What algorithms do they use?

The `python-constraint` module module offers solvers for Constraint Satisfaction Problems (CSPs) over finite domains in simple and pure Python. In the documentation, an example for solving the 8-rooks problem using this module is shown. For more information, visit the website:

The next code implements this approach.

```python
>>> problem = Problem()
>>> numpieces = 8
>>> cols = range(numpieces)
>>> rows = range(numpieces)
>>> problem.addVariables(cols, rows)
>>> for col1 in cols:
...     for col2 in cols:
...         if col1 < col2:
...             problem.addConstraint(lambda row1, row2: row1 != row2,
...                                   (col1, col2))
>>> solutions = problem.getSolutions()
```

The solutions are displayed below:

```
>>> solutions
>>> solutions
[{0: 7, 1: 6, 2: 5, 3: 4, 4: 3, 5: 2, 6: 1, 7: 0},
 {0: 7, 1: 6, 2: 5, 3: 4, 4: 3, 5: 2, 6: 0, 7: 1},
 {0: 7, 1: 6, 2: 5, 3: 4, 4: 3, 5: 1, 6: 2, 7: 0},
 {0: 7, 1: 6, 2: 5, 3: 4, 4: 3, 5: 1, 6: 0, 7: 2},
 ...
 {0: 7, 1: 5, 2: 3, 3: 6, 4: 2, 5: 1, 6: 4, 7: 0},
 {0: 7, 1: 5, 2: 3, 3: 6, 4: 1, 5: 2, 6: 0, 7: 4},
 {0: 7, 1: 5, 2: 3, 3: 6, 4: 1, 5: 2, 6: 4, 7: 0},
 {0: 7, 1: 5, 2: 3, 3: 6, 4: 1, 5: 4, 6: 2, 7: 0},
 {0: 7, 1: 5, 2: 3, 3: 6, 4: 1, 5: 4, 6: 0, 7: 2},
 ...]
```