Annales UMCS Informatica AI XIII, 1 (2013) 53–61
DOI: 10.2478/v10065-012-0013-3

Annales UMCS
Informatica
Lublin-Polonia
Sectio AI
http://www.annales.umcs.lublin.pl/

The N queens problem - new variants of the Wirth algorithm

Marcin Łajtar[1]

[1] *Institute of Computer Science, Lublin University of Technology, ul. Nadbystrzycka 36b, 20-618 Lublin, Poland*

Abstract – The paper presents new ways of n-queens problem solving . Briefly, this is a problem on a nxn chessboard of a set n-queens, so that any two of them are not in check. At the beginning, currently used algorithm to find solutions is discussed. Then sequentially 4 new algorithms, along with the interpretation of changes are given. The research results, including comparison, of calculation times of all algorithms together with their interpretation are discussed. Finally, conclusions are given. The results were obtained thanks to the pre-created application. Chapters except for "By filtering ver. 2" were based on the previous studies carried out during the Bachelor course [**1**].

1 Introduction

"The N-queens puzzle is the problem of putting N chess queens on an $N \times N$ chessboard such that none of them is able to capture any other using the standard chess queen's moves. The colour of the queens is meaningless in this puzzle, and any queen is assumed to be able to attack any other. Thus, a solution requires that no two queens share the same row, column, or diagonal" [**2**].

2 Backtracking and check function

Solving some algorithmic issues, we use a set of the searched patterns to reduce the number of calculations. State space pruning trees(backtracking) are used today in the problem of n queens to reduce the number of nodes. Niklaus Wirth [**3**] proposed that method. The results of this operation are presented in this paper as "Basic Backtracking". This has been the most effective type of structure to restrict the range

of research of this problem so far. A general form of the Backtracking algorithm can be presented as a simple code [**3**]:

```
public static void backtracking (node v)
{
  //code
    if (promising(v))
      if (last line) // solution is found
        save_solution();
      else
        for (every node ,,u'' child for ,,v'')
        backtracking (u);
}
```

Fig. 1 shows the search tree using the Backtracking algorithm. The symbol "x" indicates that the node is not promising and for it further calculations are no longer performed. The algorithm has completed its work upon finding the first solution, ((1,2) (2,4) (3,1) (4,3)). If we used the full tree of states to find a solution we would need for prior checking of 154 nodes, in the case of the Backtracking algorithm it was enough to check 22 nodes.

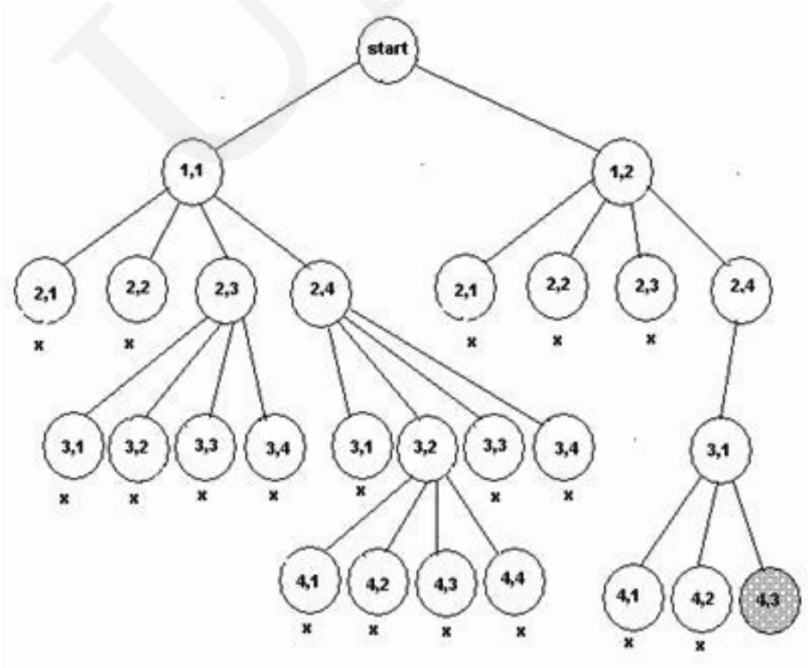

Fig. 1. Description of the return-Tree(backtracking) [**3**].

We can distinguish two types of nodes:

- Promising- -using this kind of nodes we can solve the problem
- Non-promising- broken terms of the solution. For them we do not perform further operations.

The base of algorithms is to check: "is the node promising?". It is most frequently called method and it absorbs most of the computing resources at runtime. It is based on checking “Does the Queen placed on the field "interfere" with any pre-set queen?”. It is a 100% reliable method and intuitively used.

But what happens if the chessboard is larger? It turns out that the checking-algorithm has significant influence on the runtime of the base method. So far the literature lacks interesting solutions. New ways of solving this problem and comparing it with the classical checking function are proposed in this work.

The general code of checking function is presented below:

```
  Global variables:
int queens[]-array of pre-set queens;
for example:  queens[0]= column value of the queen set in the first
row,
queens[1]= column value of queen set in the second row, etc.
Public boolean CheckNode(int row, int column)// with input parameters
of queen position
{
//code
boolean promising= true;
While(it has non check Queens and promising value is true) //comparing
with every pre- //set queen
{
if(the same column, or the same diagonal)
{promising=false;} //queen conflict with the pre-set queen;
}
return promising; //output value of promising
}
```

Table 1 shows the number of the call function CheckNode for 4-14 size chessboards and for their every row. Depending on which row function is called, it has a different number of pre-set queens to check. For example, for the first (‘0’) row, it has 0 to check because it is zero pre-set queen. For row ‘1’ it has 1 queen to check, for row ‘n’ it has ‘n’ queens to check. Of course it is not necessary to check ‘n’ nods if we find before that the checking node is not-promising. The average number of comparisons shows that this number is smaller than half of size, but this depends on it. It is very important information, because it also shows that the bigger chessboard is, the more time is needed to execute the function CheckNode. We do not know how many

solutions bigger chesboards have, but we can estimate the upper limit, as Craig Letavec and John Ruggiero [**4**] did.

Table 1. Checked nodes.

Dimension Row\	4	5	6	7	8	9	10	11	12	13	14
0	4	5	6	7	8	9	10	11	12	13	14
1	16	25	36	49	64	81	100	121	144	169	196
2	24	60	120	210	336	504	720	990	1320	1716	2184
3	16	70	216	532	1120	2106	3640	5896	9072	13390	19096
4		60	276	980	2752	6588	14000	27148	48960	83252	134848
5			240	1148	4544	14526	39160	93412	202224	404300	756952
6				658	4400	20628	75520	234982	634272	1530022	3380776
7					2496	18342	96320	409728	1441248	4355130	11690784
8						9594	78280	485628	2343240	9200074	30966152
9							40400	382514	2672640	14125384	61487832
10								166276	1931568	14973114	88522448
11									819168	10574824	90606208
12										4553926	63166908
13											27167000
Sum	60	220	894	3584	15720	72378	348150	1806706	10103868	59815314	377901398
Number of comparisons	84	405	2016	9297	46752	243009	1297558	7416541	45396914	292182579	1995957888
Avarage number of comparisons per node	1,4	1,84	2,225	2,594	2,974	3,3575	3,72701	4,105007	4,4930233	4,88474538	5,2816896

3 Symmetry

Chessboard is an object of many axes of symmetry. By using this property a lot of solutions could be obtained after the reflection, or rotation. However, we should consider how this fact can speed up the algorithm? Here some problems appear. Solutions "reflected" in most cases will not give us knowledge about the point from which to begin calculations, or upon which end. Implementation of even formation of such "settings" will lead up to duplicate results. So we can not approach this fact too greedily. The advantages coming from this relation are significant. Based on the possession of the vertical axis of symmetry we call a function executing only half of the nodes. The other solutions are symmetrical. This approach allows us to "miss" half a tree of states, which will reduce the calculation time. This fact is used in all next algorithms.

4 Backtracking ver.1

As already mentioned in Section 2 check function is not a very effective algorithm. Along with moving deep into tree, longer operations of comparison are obtained. The

main idea for speeding up the program was to create a method wich will partially or fully replace the function of checking. That was the idea of creating a boolean array specifying the column already used. Every time we "put" the Queen, the value "false" is set in the proper place. That structure is used as a filter that stops executing if the column is already used. In this way adding a few instructions largely eliminates the need for call check function. In the objects such as table, a problem with the transmission of the subsequent recursive cycles appears. It is a very large number of calls, and copying arrays dramatically increase the demand for memory. The solution is to create a two dimensional array, instead of one. The second dimension describes the row for which data is stored. Then the recursive call takes and overwrites only the appropriate line.

```
  Global variables:
Int dimension;//chess board dimension
Boolean columns[dimension][dimension];//and filled by proper values

  Public static void backtrackingVer1(int row)
{
//copy values columns[row-1][] to columns[row]
for(int i=0; i<dimension; i++)//''i'' describes column
{
    if(columns[row][i]=='true')// column was not used
    {
      if (CheckNode(i)=true)//call check function
      {
        columns[row][i]=false;//column is used now
        if (last line) // solution is found
           {save_solution(); }
        else { backtrackingVer1 (row+1); }
      }
    }
}
}
```

5 Backtracking ver. 2

Another creative intention was the rejection of having to check each of the nodes whose column index has been previously used. Such nodes are by definition non-promising and there is no need to check them. In the previous algorithm, the filter "stops" calculation. In this case they will be skipped in the declaration of the loop already. Originally the solution was to be a queue of integers that stores information about the columns which can still be used. However, this approach is loaded with

many problems in transfer of data to subsequent recursive calls. The final result is an array of integers with the same parameters. For the next row the size is successively decremented. The transmission array is performed using the lists whose indices indicate rows of the chessboard.

6 By filtering ver. 1

Looking at the backtracking algorithm ver1 we can see that we base on exclusion from the set of possible fields whose columns were previously used. By setting the queen in a certain place in accordance with the solution rules, apart from column it is possible to eliminate also the diagonal. Row is not checked, because every time we pass to the next. Following the clue of elimination of columns and diagonals, we can see that each field can be described using these parameters. The number of columns is equal to the "dimension" and the number of diagonals is equal to "2* dimension-1". If we save already used columns and diagonals in the Boolean arrays, we can completely replace the need of recall checking function. This approach significantly speeds up calculations in deeper nodes. Unlike the checking function, the operation for each node in this case takes the same amount of time.

7 By filtering ver. 2

Considering executing a program, we reach the conclusion that during the recursive call it is a jump instruction with putting appropriate values on the stack. Upon returning, these values are collected from stack and the next lines of code are executing. With this idea before entering into the tree, we can remove the columns and diagonals that have been already used, and after back instruction (while returning the parent node) to the set the appropriate values to "true"(add them back). For the program with such assumptions, we need three arrays defining the columns and diagonals used, but they will not be 2-dimensional just one-dimensional. Now copying parent arrays to child will not be necessary. This will save us a lot of operations. Operation on a critical loop in this case will be composed of 3 operations on bits for non-promising nodes, and for promising of 9 operations on bits. Execution of these operations takes much less time than copy operations on arrays or making integer operations.

8 Research results

All algorithms are based on the principle of cutting the tree of states. This method allows to significantly reduce the number of nodes to check. Precise data can be found in the work of Professor Teslera [**5**]. The "backtracking ver. 2" also limits the range, excluding the calculation nodes, whose columns have been already used. In this way the number of checked nodes is much smaller. In comparison with the other functions,

the number of checked nodes is 2-times smaller for a 4×4 chessboard and nearly 4-times smaller for a 14×14 chessboard.

The application created for research has been done in the Java platform. This is not the language which should be used if we consider calculation speed. However, for comparison it is enough to make the implementation within the same tehnology using the same rules. Persons who would like to accelerate the program will find necessary information in the conclusions.

The program includes the method with different schemes of search for solutions. Before calling any of them the counter is activated, and it stops after completion of calculations. It should be added that time has relative and absolute errors. The calculations were performed on the Intel Core 2 Duo T5550 1.83 GHz CPU and the operating system MS Windows XP. Computation time for each call was different. The reason for that might be for example the occurrence of interrupts from the system during the computation. These errors were relatively large for the small chessboards. For larger size chessboards calculation errors, became less significant. Larger 10-18 chessboards will be used in presenting results. All included algorithms are built on the symmetry relation.

Table 2. Comparison of results.

Dimension:	Times in ms:				
	Basic backtracking	Backtracking ver.1	Backtracking ver.2	By filtering ver.1	By filtering ver.2
10	16.896095	12.255002	18.970754	12.783468	2.730235
11	97.865232	66.790704	86.597412	69.778515	15.013920
12	329.048245	185.5165947	257.401989	164.302819	69.790155
13	1 850.414229	1 223.275025	1 746.307980	1 141.714659	533.501883
14	11 573.996974	6 973.351548	9 483.632231	6 059.041489	2 623.219152
15	86 234.007953	42 064.092224	57 809.804598	35 432.127826	18 264.930599
16	600 858.600464	285 466.713228	374 769.852314	224 476.190130	117 607.564090
17	4 966 228.215597	2 309 856.942991	3 083 093.004278	1 817 708.698731	879 178.312327
18	38 898 236.537884	16 686 192.199287	21 352 901.948153	13 522 508.749855	5 861 311.315265

In the next table, the information about relative profit by using the algorithms can be found. It is presented as the ratio of runtime basic function (Basic backtracking) to the selected method.

9 Conclusions

As follows from the research currently used algorithms for counting the solutions of N-queens problem are not the most efficient. They are based on the "checking method" which with subsequent calls into the depths of tree works longer. Partial or total exclusion of calling this method made it possible to speed up calculations in a non-linear way, as evidenced by the increase in the value of the ratio in the next board sizes .

Table 3. Comparison of results - rounded times.

Dimension:	Times:				
	Basic backtracking	Backtracking ver.1	Backtracking ver.2	By filtering ver.1	By filtering ver.2
10	<1s	<1s	<1s	<1s	<1s
11	<1s	<1s	<1s	<1s	<1s
12	<1s	<1s	<1s	<1s	<1s
13	2s	1s	2s	1s	<1s
14	12s	7s	9s	6s	3s
15	1min26s	42s	58s	35s	18s
16	10min	4min45s	6min15s	3min44s	1min58s
17	1h22min46s	38min30s	51min23s	30min18s	14min39s
18	10h48min18s	4h38min6s	5h55min53s	3h45min23s	1h37min41s

Table 4. Comparison of results - ratio Basic backtracking to other algorithms.

Dimension:	Ratio				
	Basic backtracking	Backtracking ver.1	Backtracking ver.2	By filtering ver.1	By filtering ver.2
10	1.000000	1.378710	0.890639	1.321714	6.188513
11	1.000000	1.465252	1.130117	1.402512	6.518300
12	1.000000	1.773686	1.278344	2.002694	4.714823
13	1.000000	1.512672	1.059615	1.620733	3.468431
14	1.000000	1.659747	1.220418	1.910203	4.412135
15	1.000000	2.050062	1.491685	2.433780	4.721289
16	1.000000	2.104829	1.603274	2.676714	5.109013
17	1.000000	2.150015	1.610794	2.732136	5.648716
18	1.000000	2.331163	1.821684	2.876555	6.636439

To speed up the program there should be used a less developed language such as C, or even implementations on the orders of microprocessor. The style of writing is also very important. Using references, operations on bits, etc. can significantly speed up the program as shown in the work by Jeffs Somers [**6**]. It is worth reading the article, which offers a multi-threaded solution of that problem [**7**]. If we are not interested in finding all the solutions, but only one we should use the heuristic methods. They are applied usually when we have to deal with the cases where it is hard to perform accurate calculations. As presented by the Year Sosic and Jun Gu [**8**], a solution for really huge chessboards can be found doing fewer operations than presented by Peter Alfeld [**9**].One should get familiar with the works [**10**], [**11**], [**12**] and [**2**], which contribute largely to the problem.

The study describes the patterns of four new methods for finding solutions:

- Backtracking ver. 1
- Backtracking ver. 2
- By filtering ver. 1
- By filtering ver. 2

Each of them brings something new into the problem. Backtracking ver. 2 even reduces the number of tested nodes. It is possible that implementation of this method in a different environment with the addition of diagonal filter would provide even better results than the algorithm "Filtering ver.1" or even "Filtering ver. 2". But the main discovery is to show other ways to achieve the objective.

Acknowledgement

I would like to express my gratitude to Professor S. Grzegórski for his supervision and helpful remarks.

References

[1] Łajtar M., Implementacja i porównanie wybranych metod w problemie N-królowych, BSc Thesis supervision Grzegórski S., Lublin University of Technology (2011).
[2] http://www.etsi.org/plugtests/grid/Document/N-QUEENS-CHALLENGE-2007-v4.pdf (01.10.2012).
[3] Wirth N., Program Development by Refinement, Communication of the ACM (1971).
[4] Letavec C. Ruggiero J., The n-Queens Problem, INFORMS Transcations on Education (2002).
[5] Tesler G., n-Queens, Math 188 (2001).
[6] http://jsomers.com/nqueen_demo/nqueens.html (01.10.2012).
[7] Hać M., Brzuszek M, Równoległe rozwiązanie problemu N-królowych z wykorzystaniem standardu OPENMP, Scientific Bulletin of Chełm 1 (2008).
[8] Rok S., Jun G., Polynomial Time Algorithms for the N-Queen Problem, ACM SIGART (1990).
[9] Alfeld P., The N by N Queens Problem, Univerity of Utah (1997).
[10] Chatham R. D., Reflections on the N + k Queens Problem, Integre Technical Publishing (2009).
[11] http://www.academic.marist.edu/∼jzbv/algorithms/Backtracking.htm (01.10.2012).
[12] http://proactive.inria.fr/index.php?page=nqueens25 (01.10.2012).