

Uninformed Search Analysis of the N-Queens Problem

Ben Sauerwine

September 20, 2003

Abstract

The purpose of this research is to explore the complexities of the N-Queens problem using uninformed searches including the average branching factor, the size of the maximum solvable puzzle, and to find interesting patterns for further research. Methods used included continued expressions, breadth-first search, and depth-first search. It was found that the models differed from the actual result by a constant factor and that there is probably no limit to the maximum-size N-Queens puzzle. Possible areas for further research are the relationship between the first lexicographic solution and the totient of n and the existence of greedy solutions other than $n = 1$ and $n = 5$.

Problem

The n-Queens problem is a classic puzzle where n queens are to be placed on an n by n "chessboard" such that no queen can attack any other queen. The goal of this research is to write a program to find solutions to the n-Queens problem, and theoretically and experimentally compute the branching factor and limits on the size of the problem.

Hypothesis

Solution Validation

The general algorithm used in my programs is to determine all places on each column where a queen could be placed then try each one in turn and repeat the procedure for the next column. This can find all possible solutions to the n-Queens problem in a finite amount of time for the following reasons:

- No two queens could ever be placed in the same column.
- The order in which the queens are placed is not important.
- The maximum depth of the problem is limited to n . In other words, after placing n queens, there are definitely no more ways to place a queen on the board. The same would be true for rooks.
- There are no cycles in the solution set. Placing a queen will never change the position of other previously placed queens.

To prove to yourself that these criteria are sufficient to prove that all solutions are found by such an algorithm, imagine a solution that was missed. This solution must have one queen in each column and the queens could have been placed in any order.

Therefore, the queens could have been placed from left to right by column and would have been found by the algorithm.

Calculating the Average Branching Factor from the Results

First, it is important to note that the average branching factor is not the same as the average of the branching factors for each node. This can be shown with a simple counterexample.

- Take for example the $n = 4$ solution to the n -Queens problem. 17 nodes are generated in 4 generations, including the initial state.

Generation				
0	1	2	3	4
		1	0	
	2	0		
	1	1	1	0
4	1	1	1	0
	2	0		
		1	0	

- The average branching factor, b , should exhibit a certain property: By the geometric series summation formula,

$$S = (1 - b^{n+1}) / (1 - b)$$

Where S will equal the total number of nodes generated after n generations. This is necessary for the branching factor to serve as a reliable estimate for the time complexity for solving the problem.

- Averaging the branching factors for the individual nodes in the first 3 generations (the fourth can be ignored since it would not be expanded even if some magic fifth generation existed) yields an average branching factor of 1.06, whereas solving the equation above for b where $n = 4$ and $S = 17$ yields 1.641, a substantially higher and more meaningful average branching factor. Therefore, the average branching factor is really a solution to the above equation for b where n is the queen count and S is the number of nodes in the search space.

Estimating the Average Branching Factor

Computing an estimate to the branching factor is a bit more difficult. Rather, it is possible to estimate the branching factor for the n -Queens problem by solving an easier problem. In these estimates, it is useful to have an approximation for $n!$ such as Stirling's Formula,

$$n! > (2 \pi n)^{1/2} (n/e)^n$$

So, in approximating the complexity in terms of total nodes generated by the n-Rooks problem, one can first use Stirling's Approximation for the desired n. Unfortunately, the branching factor in the n-Rooks problem is certainly not the same for every level, and neither Stirling's Approximation nor Factorials may be integrated or summed for each queen easily. The values to be summed follow the pattern:

$$1 + n + n(n-1) + n(n-1)(n-2) + \dots + n! + n!$$

So this is more than

$$1 + n + n(n-1) + n(n-1)(n-2) + \dots + 2n!$$

Looking at the values

$$1 + n + n(n-1) + n(n-1)(n-2) + \dots + n(n-1)(n-2)\dots(3)$$

It is apparent that in order to get another n!, all previous terms in the sum must be at least as big as the last term. However, the next smaller term is 1/3 it, the next 1/12, and so on until 1/(n)(n-1)(n-2)(n-3)...(4)(3). In fact, the sum of all terms before any subsequent turn can never quite equal the value of the next larger term. In the interest of an approximation that can be done without the use of another approximation entirely, i.e. that of the gamma function, the sum of the series is no bigger than 3n!.

$$3 (2 \pi n)^{1/2} (n/e)^n = (1 - b^{n+1}) / (1 - b)$$

However, even this equation is not solvable using simple algebra. At best, one could solve it for specific values of n and compare the result to those obtained experimentally. The best way to do this is to use a continued expression:

b is initialized with a number greater than 1.
b = 1.01

Re-evaluate b for many iterations.
b = ((b - 1) 3 (2 pi n)^{1/2} (n/e)^n + 1)^{1/(n+1)}

The result is a very good approximation of the branching factor of the n-Rooks model. To improve on this model, one can invent a new piece, the Rook-King, a piece that can attack as a King or a Rook. Assuming that at least two spaces are "determined", in that there is only one option to choose from for them, that there is no overlapping of blocked spaces by the Rook-Kings, and that the border case where only two spaces are blocked because one goes off the side are insignificant, one can estimate b using this continued expression by borrowing the calculations from the last model:

$$b = ((b - 1) n 3 (2 \pi (n - 3))^{1/2} ((n-3)/e)^{(n-3)} + 1)^{1/(n+1)}$$

Where $n > 3$

One more possible model is a probabilistic one: if this is the x th queen being placed, the probability that any previous queen might have blocked any space is

$$((n - 2) / n)^{x-1}$$

since the average number of spaces blocked per queen placed is two. Therefore, the expected value for open spaces on the x -th placement is

$$n ((n - 2) / n)^{x-1}$$

Unfortunately, the branching factor for each queen placed will be different based on how many queens were placed before it. The first step is to multiply these to determine the expected number of nodes on each level. The product of these for $x = 1$ to N is

$$((n - 2) / n)^{(1/2)(N-1)N} n^N$$

However, this value once again approximates only the number of final solutions, where the average branching factor is really the “average” number of choices on each level, which is certainly not the same. Now it is necessary to sum this for $N = 1$ to n to determine the expected number of nodes after n levels. This is a challenge that even limits are not up to. Thankfully, this value is a constant that can be calculated easily and can be represented in the earlier continued expression.

$$b = ((b - 1)(C) + 1)^{1/(n+1)}$$

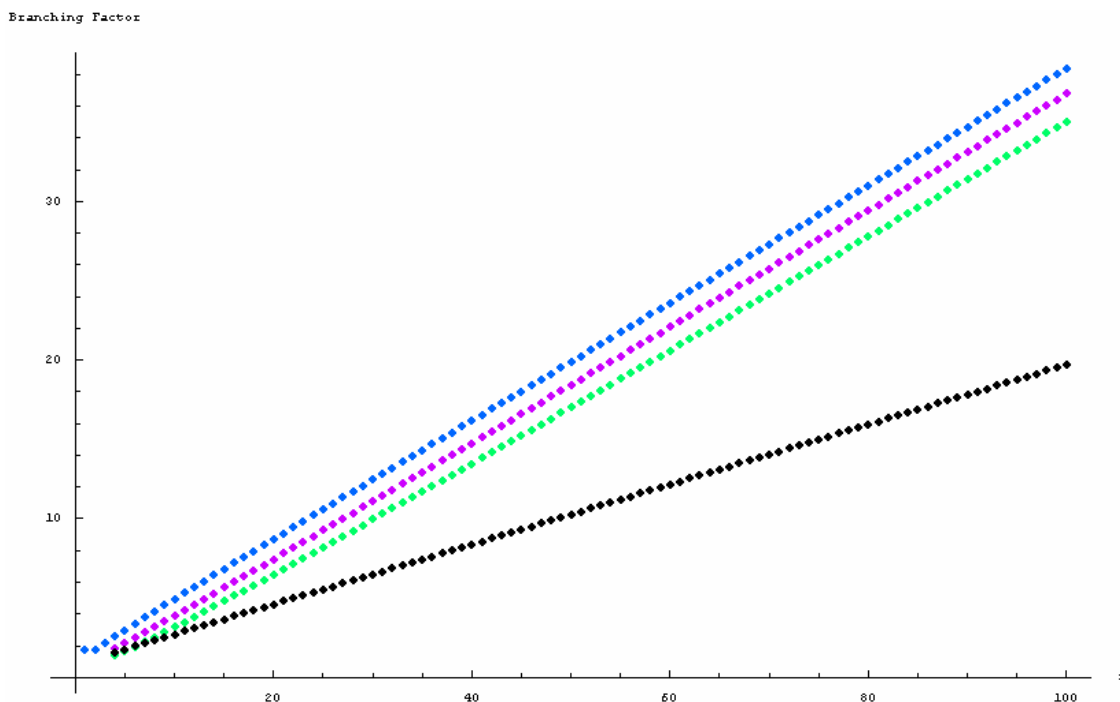
Where n cannot be 2

Results and Discussion

Branching Factor Analysis

The calculated Branching Factor grows in a roughly linear way, similar to the growth of all of the models. This is surprising since it is extremely difficult mathematically to emulate the details of the n -Queens problem such as branches that end prematurely and impossible configurations. Another serious problem with branching factor analysis entirely was that whereas the number of possible placements in the last few generations of the tree tends to taper off rather than grow as branches die off and few possible locations are left on which to place queens. Regardless of this fact, the models did very well for extreme simplifications of the problem at hand.

Branching Factor						
n-Queens	Average Change	N-Rooks	N-Rook-King	Probabilistic	Linear Regression	
1	1.000		1.766		1.000	1.019
2	1.000	0.000	1.738			1.208
3	1.278	0.278	2.139		1.389	1.397
4	1.641	0.363	2.567	1.406	1.842	1.586
5	1.926	0.285	2.982	1.639	2.208	1.775
6	2.075	0.149	3.384	1.926	2.547	1.964
7	2.268	0.194	3.778	2.228	2.882	2.153
8	2.429	0.160	4.167	2.537	3.217	2.341
9	2.585	0.156	4.554	2.851	3.557	2.530
10	2.724	0.139	4.937	3.169	3.900	2.719
11	2.870	0.146	5.319	3.491	4.247	2.908
12	3.019	0.149	5.700	3.817	4.596	3.097
13	3.165	0.146	6.080	4.145	4.949	3.286
14			6.459	4.475	5.303	3.475
15			6.837	4.808	5.659	3.664



The image above shows a graph of the different models' estimates for the branching factor versus the size, n . The top line is the n -Rooks estimate, the second line from the top is the probabilistic estimate, the bottom line is the n -Rook-King estimate, and the bottom line is the linear regression of the actual values of the branching factor. This result implies that the result might be improved by multiplying the models by a constant.

Limits to the Maximum Size of the n-Queens Problem

The only apparent limit to the size of the n-Queens problem is processor-time wise. It is certainly not well suited to a BFS solution since the results all lie at the deepest level and there are no cycles in the search space.

BFS Results in creating the first lexicographic solution

n-Queens	BFS		BFS Runtime	BFS Space
	BFS Nodes Expanded	Nodes Generated		
1	1	2	0.000	96
2	3	3	0.000	456
3	6	6	0.000	1760
4	15	17	0.000	8880
5	44	54	0.000	47832
6	149	153	0.031	245320
7	512	552	0.187	1477184
8	1965	2057	1.141	11610168
9	8042	8394	7.437	131114080
10	34815	35539	74.485	1997833560
11	Impractical	166926	Impractical	
12	Impractical	856189	Impractical	
13	Impractical	4674890	Impractical	

All indicators seem to point to there being no limit to the size of the largest solvable n-Queens puzzle.

- Test 1: Branching Factor
The branching factor and n appear to be constantly increasing, yielding a constantly increasing search space.
- Test 2: Solution Count
The number of solutions appears to be constantly increasing for $n > 4$.
- Test 3: First Solution
If the first solution were never to be found, as is the case in an unsolvable puzzle, as many DFS nodes would have to be generated as are in the search space. Based on the observed pattern, this would imply a very small of the totient of n compared to n. However, there are infinitely many prime numbers that have a very large totient compared to n. This means that their first solution would likely be found early, so at best, unsolvable values of n would likely be sporadically placed if at all.
- Test 4: Analogous Problem
Magic Squares can be used to generate n-Queens solutions, and vice-versa. Therefore, since pandiagonal magic squares of all sizes except 2 and 3 exist, so should n-Queens solutions. (Pickover 279).

Interesting Patterns for Further Research

One interesting pattern in the results of the experiment involved the distribution of “lucky” solutions where the DFS algorithm found its solution unusually fast. The first step in analyzing this pattern further was to find the distribution of “greedy” solutions where the first solution tried was correct. Looking through the list of a few showed 1 and 5 as greedy solutions, but did others exist?

n-Queens	First Lexicographic Solution
1	{1}
2	No Solution Exists
3	No Solution Exists
4	{2 4 1 3}
5	{1 3 5 2 4}
6	{2 4 6 1 3 5}
7	{1 3 5 7 2 4 6}
8	{1 5 8 6 3 7 2 4}
9	{1 3 6 8 2 4 9 7 5}
10	{1 3 6 8 10 5 9 2 4 7}
11	{1 3 5 7 9 11 2 4 6 8 10}
12	{1 3 5 8 10 12 6 11 2 7 9 4}
13	{1 3 5 2 9 12 10 13 4 6 8 11 7}
14	{1 3 5 7 12 10 13 4 14 9 2 6 8 11}
15	{1 3 5 2 10 12 14 4 13 9 6 15 7 11 8}
16	{1 3 5 2 13 9 14 12 15 6 16 7 4 11 8 10}
17	{1 3 5 2 8 11 15 7 16 14 17 4 6 9 12 10 13}
18	{1 3 5 2 8 15 12 16 13 17 6 18 7 4 11 9 14 10}
19	{1 3 5 2 4 9 13 15 17 19 7 16 18 11 6 8 10 12 14}
20	{1 3 5 2 4 13 15 12 18 20 17 9 16 19 8 10 7 14 6 11}
21	{1 3 5 2 4 9 11 15 21 18 20 17 19 7 12 10 8 6 14 16 13}
22	{1 3 5 2 4 10 14 17 20 13 19 22 18 8 21 12 9 6 16 7 11 15}
23	{1 3 5 2 4 9 11 13 18 20 22 19 21 10 8 6 23 7 16 12 15 17 14}
24	{1 3 5 2 4 9 11 14 18 22 19 23 20 24 10 21 6 8 12 16 13 7 17 15}
25	{1 3 5 2 4 9 11 13 15 19 21 24 20 25 23 6 8 10 7 14 16 18 12 17 22}

Upon running the test program in the Appendix that kept increasing n and trying the greediest solution possible for 7 hours and 45 minutes, no more greedy solutions were found. This does not necessarily mean that no more greedy solutions exists, but it does support the idea that there may be no more. The most likely cause of this becomes obvious upon examining the placement of pieces in these greedy solutions: a diagonal exists bisecting the square from the 5-greedy solution. All of the pieces below this diagonal are placed in typically long sets that, within the set, are placed with each being two squares lower than the last. This means that for every piece that cannot be placed above the diagonal, the size of the square must increase by at least two. The number of

“misses” rapidly accumulates and results in a seemingly irreversible and growing discrepancy between the square’s height and width.

Equipped with the hypothesis that no more greedy solutions exist, but with evidence that DFS finds a solution faster for some numbers, especially odd numbers and primes, faster than other numbers, an attempt was made to link the results to Euler’s totient function.

n-Queens	DFS Nodes Expanded	DFS Nodes Generated	DFS Runtime (seconds)	DFS Space (bytes)	Solution Count	Totient
1	1	2	0.000	104	1	
2	3	3	0.000	464	0	1
3	6	6	0.000	1752	0	2
4	8	11	0.000	4480	2	2
5	5	12	0.000	4448	10	4
6	31	40	0.000	40264	4	2
7	9	23	0.000	16184	40	6
8	113	125	0.031	273464	92	4
9	41	61	0.031	117272	352	6
10	102	125	0.063	372000	724	4
11	52	84	0.032	215688	2680	10
12	261	296	0.343	1396256	14200	4
13	111	155	0.110	669480	73712	12
14	1899	1945	2.688	14292328	Impractical	6
15	1359	1415	1.969	11597656	Impractical	8
16	10052	10113	17.656	101103416	Impractical	8
17	5374	5450	10.157	59347104	Impractical	16
18	41299	41378	91.219	533121856	Impractical	6
19	2545	2643	6.204	35374232	Impractical	18
20	199635	199734	559.563	3227919552	Impractical	8
21	8562	8677		Impractical		14
22	1737188	1737309		Impractical		10
23	25428	25567		Impractical		22
24	411608	411756		Impractical		8
25	48683	48859		Impractical		20

The “DFS Nodes Expanded” column hits minima and maxima that correspond inversely with the values of Euler’s Totient function. Unfortunately, it is difficult to gauge just how “lucky” DFS got from these results only, as it is impossible to say from these results only how many nth-level branches were explored versus the total number of solutions and the total number of nth-level nodes. Given that researchers have been able to create n-Queens solutions from n by n magic squares and vice-versa, however, this result is not terribly shocking (Pickover 283).

Conclusions

The algorithm supplied did indeed find all of the solutions to the n-Queens problem for a supplied n. The branching factor of the actual problem was not as meaningful as possible because the search space tapered rapidly as the last several queens were placed. However, the models predicted that the branching factor would grow in a linear fashion with respect to n, and it did seem to converge quickly to a slope of 0.148. All of the models' speculated growth factors overestimated, roughly, by constant factors, suggesting that the simplification models captured many of the most important aspects of the n-Queens problem's complexity. Furthermore, the evidence suggested that there is no unsolvable n-Queens problem with n greater than 3.

Appendix: Code Reference

Modify

Replace occurrences of mtarget in msearch with mreplace.

If msearch is equal to the mtarget, replace it with the mreplace.

If this is an atom return this.

If this is a list, return modify car with modify cdr at the end.

```
(defun MODIFY (mtarget msearch mreplace)
  (cond
    (
      (equal msearch mtarget)
      mreplace
    )
    (
      (atom msearch)
      msearch
    )
    (
      (listp msearch)
      (append
        (list (MODIFY mtarget (car msearch) mreplace))
        (MODIFY mtarget (cdr msearch) mreplace)
      )
    )
  )
)
```

```
[85]> (modify 'a '((b (c a) d) x a (a (u v))) '(w w))
((B (C (W W)) D) X (W W) ((W W) (U V)))
```

```
[92]> (modify '(u) '((a (u v)) ((u)) (y (u) v)) 'v)
((A (U V)) (V) (Y V V))
```

```
[93]> (modify '(3 4) '((1 8)(2 4)(3 7)(4 3)(5 6)(6 2)(7
5)(8 1)) '(0 0))
((1 8) (2 4) (3 7) (4 3) (5 6) (6 2) (7 5) (8 1))
```

```
[99]> (modify '(who feeds) '((who feeds) keys to the goose
who feeds me) '(I feed))
((I FEED) KEYS TO THE GOOSE WHO FEEDS ME)
```

```
[104]> (modify '(lots (and (lots (of (money)))))) '(they pay
me (lots (and (lots (of (money)))))) and I like it) 'nil)
(THEY PAY ME NIL AND I LIKE IT)
```

Threat?

This will determine if a piece placed at this row at the last column of list would threaten any of the preceding pieces assuming that they are ordered by column with the value in the list being row. It loops through the list in reverse, checking at each column to see whether the spaces threatened by this queen hit an existing queen.

```
(defun Threat? (row collist)
  (let ((posn 0) (thrt nil))
    (dolist
      (itm (reverse collist))
      (if
        (or
          (= itm row)
          (= (abs (- itm row)) (setf posn (+ posn 1))))
        (setf thrt (or thrt t))
        )
      )
    thrt
  )
)
```

Valid-board?

Determine if a board state is valid. If the board is empty, it's valid. If not, it's valid if the piece on the far right doesn't threaten any pieces on the left and if the portion of the board to the left is also valid.

```
(defun Valid-Board? (board)
  (cond
    ((null board)
     t)
    (t
     (and
      (not
       (threat?
        (car (reverse board))
        (reverse (cdr (reverse board))))))
      (Valid-Board? (reverse (cdr (reverse board))))))
  )
)
```

```
[25]> (valid-board? '(5 1 4 2 7 3 8 6))
```

```
NIL
```

```
[26]> (valid-board? '(6 4 7 1 8 2 5 3))
```

```
T
```

```
[27]> (valid-board? '(1 5 2 6 3 7 4 8))
```

```
NIL
```

```
[28]> (valid-board? '(7 2 5 8 6 4 3 1))
```

```
NIL
```

Expand

Add all valid board states stemming from this board state, assuming that we are solving the n-queens problem for the n given. Loop from 1 to n, adding all valid sub-boards to a list.

```
(defun Expand (board n)
  (do
    ((i 1 (+ i 1)) (subboards '()))
    ((> i n) subboards)
    (if
     (valid-board? (append board (list i)))
     (setf subboards
           (append subboards (list (append board (list i))))
          )
     )
    )
  )
)
```

N-Queens_Single_Solution

Find all solutions to the n-queens problem given stemming from the given starting state with a BFS queue. Also return the number of nodes expanded and the number of nodes generated. To do so, check the first node on the queue to see if it's a solution, and if not, expand it and place the newly generated nodes on the back of the queue.

```
(defun n-Queens_Single_Solution (state n)
  (do ((nodelist state) (expanded 0))
    (
     (or (null nodelist) (= (list-length (car nodelist)) n))
     (append
      (list (car nodelist))
      (list expanded)
      (list (+ expanded (list-length nodelist)))
      )
     )
    (setf nodelist
          (append (cdr nodelist) (expand (car nodelist) n))
          )
    (setf expanded (+ expanded 1))
    )
  )
)
```

N-Queens_Single_Solution_DFS

Similar to the BFS code, but rather than placing the newly generated nodes on the front of the stack.

```
(defun n-Queens_Single_Solution_DFS (state n)
  (do ((nodelist state) (expanded 0))
      (
        (or (null nodelist) (= (list-length (car nodelist)) n))
        (append
         (list (car nodelist))
         (list expanded)
         (list (+ expanded (list-length nodelist))))
        )
      )
    (setf nodelist
      (append (expand (car nodelist) n) (cdr nodelist)))
      )
    (setf expanded (+ expanded 1))
    )
  )
```

N-Queens_Solution_Count

Count the number of solutions to the N-Queens problem with n queens and an arbitrary start state. This is similar to the single solution BFS, except it continues when a solution is found, tallying solutions until all paths have been exhausted.

```
(defun n-Queens_Solution_Count (state n)
  (do ((nodelist state) (expanded 0) (solutions 0))
      (
        (null nodelist)
        (append (list solutions) (list expanded))
        )
      (if (= (list-length (car nodelist)) n)
          (setf solutions (+ solutions 1))
          )
      (setf nodelist
        (append (cdr nodelist)
                 (expand (car nodelist) n))
        )
      (setf expanded (+ expanded 1))
      )
  )
```

N-Queens_Solution_Count_DFS

This is identical to the N-Queens_Solution_Count, except it uses a DFS stack instead of a BFS queue. The difference is that the newly generated nodes are inserted earlier into the open list.

```
(defun n-Queens_Solution_Count_DFS (state n)
  (do ((nodelist state) (expanded 0) (solutions 0))
      (
        (null nodelist)
        (append (list solutions) (list expanded))
      )
    (if (= (list-length (car nodelist)) n)
        (setf solutions (+ solutions 1))
      )
    (setf nodelist
      (expand (car nodelist) n)
      (append (cdr nodelist))
    )
    (setf expanded (+ expanded 1))
  )
)
```

Greedy Solution Experiment

This experiment is designed to try placing Queens across the columns, placing each queen in the first available row and outputting each n-Queens “greedy” solution found. After 7 hours and 45 minutes of runtime, it finds only 1 and 5. This supports the idea that no more greedy solutions exist.

```

#include <iostream>
#include <cmath>
#include <vector>

using namespace std;

void main()
{
vector<int> queens;
int maxqueen;
int a;
int b;

queens.resize(1);
maxqueen = 0;

while (true)
{
    queens.resize(queens.size() + 1);

    for (a=1;;a++)
    {
        for (b=2;b<queens.size();b++)
            if ((a==queens[queens.size()-b]) ||
                (abs(a - queens[queens.size()-b])==b-1))
                break;

        if (b==queens.size())
            break;
    }

    if (a>maxqueen)
        maxqueen = a;

    queens[queens.size()-1] = a;

    if (maxqueen==queens.size()-1)
        cout<<maxqueen<<endl;
}
}

```

Works Referenced

Graham, Paul. ANSI Common Lisp. Prentice Hall: Upper Saddle River, New Jersey. 1996.

Schroeder, M R. Number Theory in Science and Communication. Springer-Verlag: Germany. 1986.

Works Cited

Pickover, Clifford A. The Zen of Magic Squares, Circles and Stars. Princeton University Press: Princeton, New Jersey. 2002.