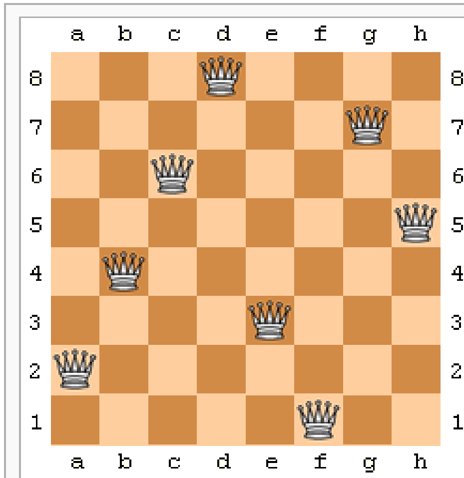
nqueens

(CP: “hello world”)

Eight queens puzzle

From Wikipedia, the free encyclopedia



One solution.

The **eight queens puzzle** is the problem of putting eight chess queens on an 8×8 chessboard such that none of them is able to capture any other using the standard chess queen's moves. The queens must be placed in such a way that no two queens would be able to attack each other. Thus, a solution requires that no two queens share the same row, column, or diagonal. The eight queens puzzle is an example of the more general ***n* queens puzzle** of placing *n* queens on an *n*×*n* chessboard, where solutions exist only for *n* = 1 or *n* ≥ 4.

Contents [hide]

History

[edit]

The puzzle was originally proposed in 1848 by the chess player Max Bezzel, and over the years, many mathematicians, including Gauss and Georg Cantor, have worked on this puzzle and its generalized n-queens problem. The first solutions were provided by Franz Nauck in 1850. Nauck also extended the puzzle to n-queens problem (on an n×n board—a chessboard of arbitrary size). In 1874, S. Gunther proposed a method of finding solutions by using determinants, and J.W.L. Glaisher refined this approach.

Eight queens puzzle - Wikipedia, the free encyclopedia - Windows Internet Explorer

http://en.wikipedia.org/wiki/Eight_queens_puzzle

History [edit]

The puzzle was originally proposed in 1848 by the chess player Max Bezzel, and over the years, many mathematicians, including Gauss and Georg Cantor, have worked on this puzzle and its generalized n-queens problem. The first solutions were provided by Franz Nauck in 1850. Nauck also extended the puzzle to n-queens problem (on an n×n board—a chessboard of arbitrary size). In 1874, S. Gunther proposed a method of finding solutions by using determinants, and J.W.L. Glaisher refined this approach.

Edsger Dijkstra used this problem in 1972 to illustrate the power of what he called structured programming. He published a highly detailed description of the development of a depth-first backtracking algorithm.[2]

This puzzle appeared in the popular early 1990s computer game *The 7th Guest*.

Constructing a solution [edit]

The problem can be quite computationally expensive as there are 4,426,165,368 (or 64!/(56!8!)) possible arrangements of eight queens on the board, but only 92 solutions. It is possible to use shortcuts that reduce computational requirements or rules of thumb that avoids brute force computational techniques. For example, just by applying a simple rule that constrains each queen to a single column (or row), though still considered brute force, it is possible to reduce the number of possibilities to just 16,777,216 (8^8) possible combinations, which is computationally manageable for n=8, but would be intractable for problems of n=1,000,000. Extremely interesting advancements for this and other toy problems is the development and application of heuristics (rules of thumb) that yield solutions to the *n* queens puzzle at an astounding fraction of the computational requirements. This heuristic solves *n* queens for any n $n \geq 4$ or $n = 1$:

1. Divide *n* by 12. Remember the remainder (*n* is 8 for the eight queens puzzle).
2. Write a list of the even numbers from 2 to *n* in order.
3. If the remainder is 3 or 9, move 2 to the end of the list.
4. Append the odd numbers from 1 to *n* in order, but, if the remainder is 8, switch pairs (i.e. 3, 1, 7, 5, 11, 9, …).
5. If the remainder is 2, switch the places of 1 and 3, then move 5 to the end of the list.
6. If the remainder is 3 or 9, move 1 and 3 to the end of the list.
7. Place the first-column queen in the row with the first number in the list, place the second-column queen in the row with the second number in the list, etc.

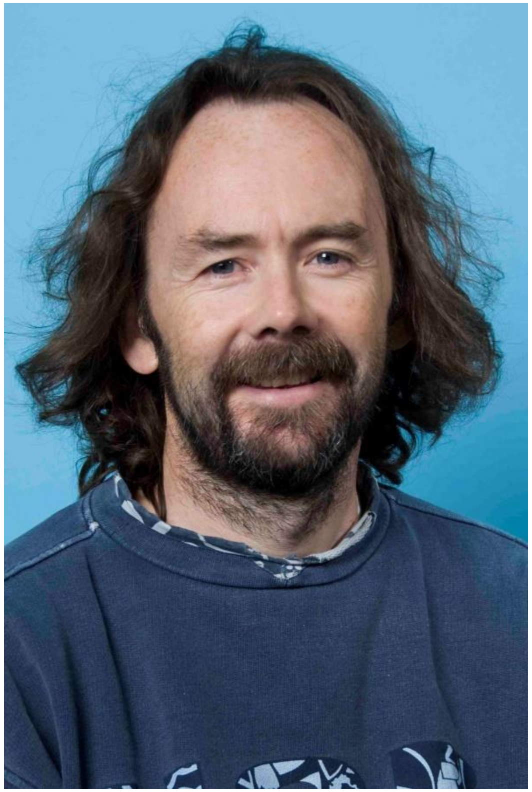
For $n = 8$ this results in the solution shown above. A few more examples follow.

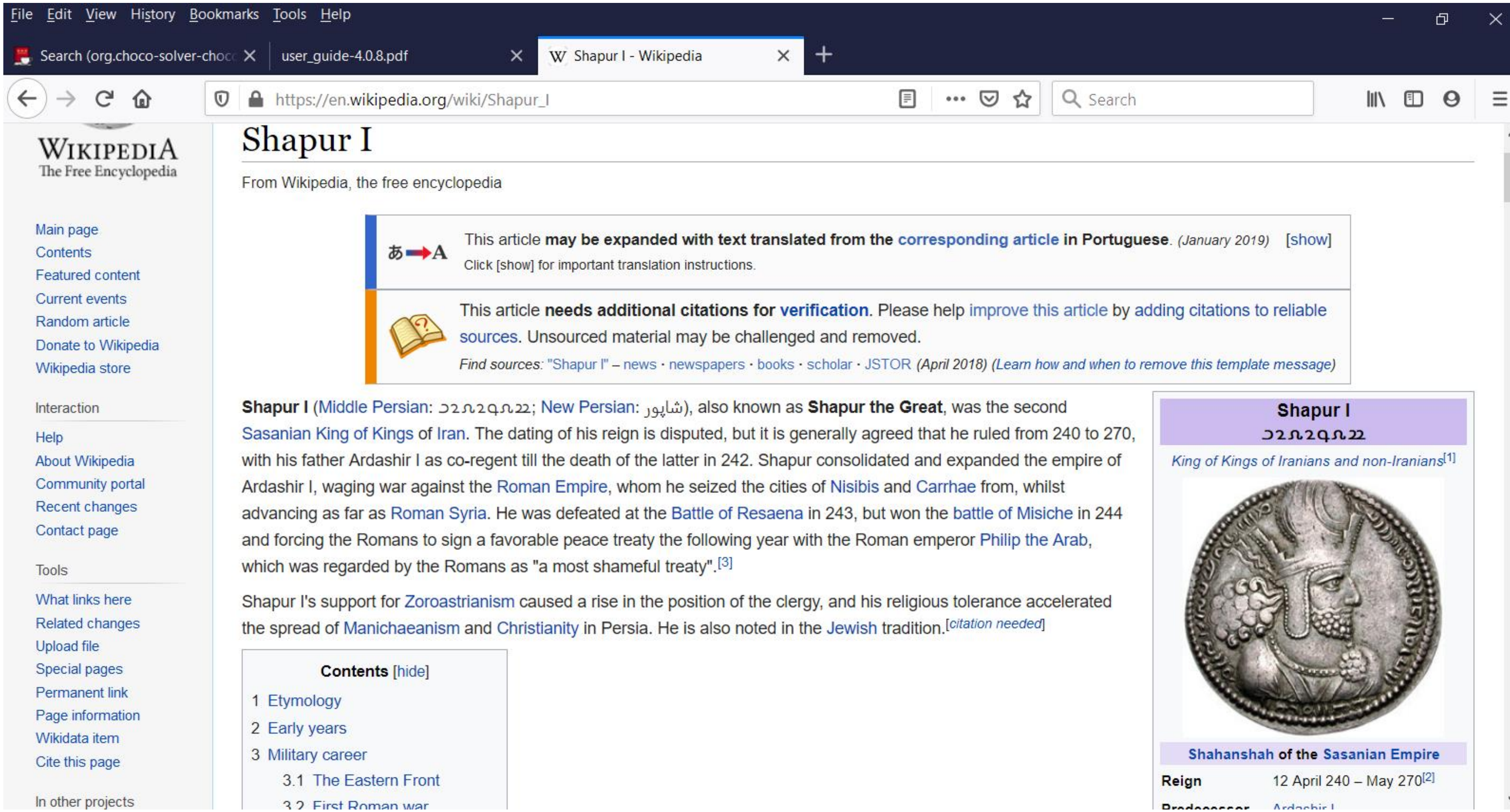
- 14 queens (remainder 2): 2, 4, 6, 8, 10, 12, 14, 3, 1, 7, 9, 11, 13, 5.

Homework hw00

```
                              m-queens
                              ---------
The famous n-queens problem is actually a simplifcation of the much more
important m-queens problem, which is originally attributed to Adardhir I the
founder of the Sassanid Empire. His mother Rodhagh challenged him to cover
his empire with the minimum number of castles so that no place was not pro-
tected by cavalry within 1 week riding. Adardhir simplifed this problem into
the m-queens problem, based on the game of chess, which was becoming popular
at the time, under the name shatranj. His son Shapur I is attributed with the
first optimal solution on a standard 8x8 chessboard. The m-queens problem,
generalizing Adardhir's problem, is to cover an n x n chess board with as few
queens as possible so that no queen can take another and no more queens can
be placed on the board without being taken.

For example given n = 5 a solution might be
q = [2, 0, 5, 3, 1]; representing the solution

     . Q . . .
     . . . . .
     . . . . Q
     . . Q . .
     Q . . . .

where 0 means no queen on that row. Note that above a 5th queen cannot be
placed on the board witjout being under attack.

and a minimal solution might be
q = [5,2,0,3,0]; representing the solution

     . . . . Q
     . Q . . .
     . . . . .
     . . Q . .
     . . . . .

That is, above we see that by placing 3 queens on the 5x5
board the queens do not attack each other and all other vacant
positions are under attack. The above solution is minimal
```

Peter Stuckey myth?





File Edit View History Bookmarks Tools Help

Search (org.choco-solver-choc | user_guide-4.0.8.pdf | Shapur I - Wikipedia

https://en.wikipedia.org/wiki/Shapur_I

WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction

Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools

What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

In other projects

Shapur I

From Wikipedia, the free encyclopedia

This article **may be expanded with text translated from the corresponding article in Portuguese**. *(January 2019)* [show]
Click [show] for important translation instructions.

This article **needs additional citations for verification**. Please help improve this article by adding citations to reliable sources. Unsourced material may be challenged and removed.
Find sources: "Shapur I" – news · newspapers · books · scholar · JSTOR *(April 2018) (Learn how and when to remove this template message)*

Shapur I (Middle Persian: 𐭱𐭧𐭯𐭥𐭧𐭩; New Persian: شاپور), also known as **Shapur the Great**, was the second Sasanian King of Kings of Iran. The dating of his reign is disputed, but it is generally agreed that he ruled from 240 to 270, with his father Ardashir I as co-regent till the death of the latter in 242. Shapur consolidated and expanded the empire of Ardashir I, waging war against the Roman Empire, whom he seized the cities of Nisibis and Carrhae from, whilst advancing as far as Roman Syria. He was defeated at the Battle of Resaena in 243, but won the battle of Misiche in 244 and forcing the Romans to sign a favorable peace treaty the following year with the Roman emperor Philip the Arab, which was regarded by the Romans as "a most shameful treaty".[3]

Shapur I's support for Zoroastrianism caused a rise in the position of the clergy, and his religious tolerance accelerated the spread of Manichaeanism and Christianity in Persia. He is also noted in the Jewish tradition.[citation needed]

Contents [hide]

1 Etymology
2 Early years
3 Military career
3.1 The Eastern Front
3.2 First Roman war

Shapur I
𐭱𐭧𐭯𐭥𐭧𐭩

King of Kings of Iranians and non-Iranians[1]

Shahanshah of the Sasanian Empire

Reign 12 April 240 – May 270[2]

Predecessor Ardashir I

Edsger W. Dijkstra - Wikipedia, the free encyclopedia - Windows Internet Explorer

http://en.wikipedia.org/wiki/Edsger_Dijkstra

article | discussion | edit this page | history

WIKIPEDIA
The Free Encyclopedia

navigation
- Main page
- Contents
- Featured content
- Current events
- Random article

search

Go Search

interaction
- About Wikipedia
- Community portal
- Recent changes
- Contact Wikipedia
- Donate to Wikipedia
- Help

toolbox
- What links here
- Related changes
- Upload file
- Special pages
- Printable version
- Permanent link
- Cite this page

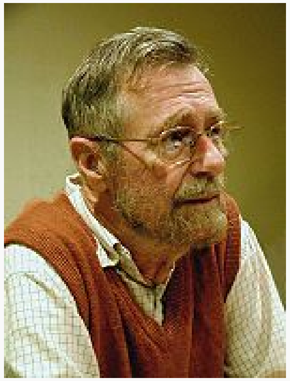
Edsger W. Dijkstra

From Wikipedia, the free encyclopedia
(Redirected from Edsger Dijkstra)

Edsger Wybe Dijkstra (May 11, 1930 – August 6, 2002; Dutch pronunciation: [ˈɛtsxər ˈwibə ˈdɛɪkstra]) was a Dutch computer scientist. He received the 1972 Turing Award for fundamental contributions to developing programming languages, and was the Schlumberger Centennial Chair of Computer Sciences at The University of Texas at Austin from 1984 until 2000.

Shortly before his death in 2002, he received the ACM PODC Influential Paper Award in distributed computing for his work in the sub-topic of self-stabilization of program computation. This annual award was renamed the Dijkstra Prize the following year, in his honour.

Edsger Wybe Dijkstra



Born	May 11, 1930 Rotterdam, Netherlands
Died	August 6, 2002 (aged 72) Nuenen, Netherlands
Fields	Computer science
Institutions	Mathematisch Centrum Eindhoven University of Technology The University of Texas at Austin
Doctoral advisor	Adriaan van Wijngaarden
Doctoral students	Nico Habermann Martin Rem David Naumann Cornelis Hemerik Jan Tijmen Udding Johannes van de Snepscheut

Contents [hide]

Life and work [edit]

Born in Rotterdam, Netherlands, Dijkstra studied theoretical physics at Leiden University, but he quickly realized he was more interested in computer science. Originally employed by the Mathematisch Centrum in Amsterdam, he held a professorship at the Eindhoven University of Technology in the Netherlands, worked as a research fellow for Burroughs

start | Edsger ... | nqueens | emacs@... | Comma... | Microsof... | 10:58

Structured programming

From Wikipedia, the free encyclopedia

Structured programming can be seen as a subset or subdiscipline of procedural programming, one of the major programming paradigms. It is most famous for removing or reducing reliance on the GOTO statement.

Historically, several different structuring techniques or methodologies have been developed for writing structured programs. The most common are:

1. Edsger Dijkstra's structured programming, where the logic of a program is a structure composed of similar sub-structures in a limited number of ways. This reduces understanding a program to understanding each structure on its own, and in relation to that containing it, a useful separation of concerns.
2. A view derived from Dijkstra's which also advocates splitting programs into sub-sections with a single point of entry, but is strongly opposed to the concept of a single point of exit.
3. Data Structured Programming or Jackson Structured Programming, which is based on aligning data structures with program structures. This approach applied the fundamental structures proposed by Dijkstra, but as constructs that used the high-level structure of a program to be modeled on the underlying data structures being processed. There are at least 3 major approaches to data structured program design proposed by Jean-Dominique Warnier, Michael A. Jackson, and Ken Orr.

The two latter meanings for the term "structured programming" are more common, and that is what this article will discuss. Years after Dijkstra (1969), object-oriented programming (OOP) was developed to handle very large or complex programs (see below: *Object-oriented comparison*).

Contents [hide]

Programming paradigms

- Agent-oriented
- Component-based
  - Flow-based
  - Pipeline
- Concatenative
- Concurrent computing
- Declarative (contrast: Imperative)
  - Functional
    - Dataflow
      - Cell-oriented (spreadsheets)
      - Reactive
  - Graph-oriented
  - Goal-oriented
    - Constraint
      - Logic
        - Constraint logic
        - Abductive logic
        - Inductive logic
- Event-driven
  - Service-oriented
- Feature-oriented
- Function-level (contrast: Value-level)
- Imperative (contrast: Declarative)
  - Greater separation of concerns
    - Aspect-oriented

A.P.I.C. Studies in Data Processing
No. 8

STRUCTURED PROGRAMMING

O.-J. DAHL
Universitet i Oslo,
Matematisk Institutt,
Blindern, Oslo, Norway

E. W. DIJKSTRA
Department of Mathematics,
Technological University,
Eindhoven, The Netherlands

C. A. R. HOARE
Department of Computer Science,
The Queen's University of Belfast,
Belfast, Northern Ireland

1972
ACADEMIC PRESS
LONDON AND NEW YORK

CONTENTS

This section has not been included because the problem tackled in it is very exciting. On the contrary, I feel tempted to remark that the problem is perhaps too trivial to act as a good testing ground for an orderly approach to the problem of program composition. This section has been included because it contains a true eye-witness account of what happened in the classroom. It should be interpreted as a partial answer to the question that is often posed to me, viz. to what extent I can teach programming style. (I never used the "Notes on Structured Programming"—mainly addressed to myself and perhaps to my colleagues—in teaching. The classroom experiment described in this section took place at the end of a course entitled "Introduction into the Art of Programming", for which separate lecture notes—with exercises and all that—were written. As at the moment of writing the students that followed this course have still to pass their examination, it is for me still an open question how successful I have been. They liked the course, I have heard that they described my programs as "logical poems", so I have the best of hopes.)

17. The Problem Of The Eight Queens

This last section is adapted from my lecture notes "Introduction into the Art of Programming". I owe the example—as many other good ones—to Niklaus Wirth. This last section is added for two reasons.

Firstly, it is a second effort to do more justice to the process of invention. (As a matter of fact I start where the student is not familiar with the concept of backtracking and aim at discovering it as I go along.)

Secondly, and that is more important, it deals with recursion as a programming technique. In preceding sections (particularly in "On a program model") I have reviewed the subroutine concept; there it emerged as an embodiment of what I have also called "operational abstraction". In the relation between main program and subroutine we can distinguish quite clearly two different semantic levels. On the level of the main program the subroutine represents a primitive action; on that level it is used on account of "what it does for us" and on that same level it is irrelevant "how it works". On the level of the subroutine body we are concerned with how it works but can—and should—abstract from how it is used. This clear separation of the two semantic levels "what it does" and "how it works" is denied to the designer of a recursive procedure. As a result of this circumstance the design of a recursive routine requires a different mental skill, justifying the inclusion of the current section in this manuscript. The recursive procedure has to be understood and conceived on a single semantic level: as such it is more like a sequencing device, comparable to the repetitive clauses.


Letters to the editor: go to statement considered harmful - Windows Internet Explorer
http://portal.acm.org/citation.cfm?id=362929.362947
structured programming
Page
Tools
PORTAL
UK Higher Education Institutions (UK HEI): Glasgow University Library
Subscribe (Full Service) Register (Limited Service, Free) Login
Search: The ACM Digital Library The Guide
SEARCH
THE GUIDE TO COMPUTING LITERATURE
Letters to the editor: go to statement considered ha
Full text Pdf (1.07 MB)
Source Communications of the ACM archive
Volume 11 , Issue 3 (March 1968) table of contents
Pages: 147 - 148
Year of Publication: 1968
ISSN:0001-0782
Author Edsger W. Dijkstra Technological Univ., Eindhoven, The
Publisher ACM New York, NY, USA
Bibliometrics Downloads (6 Weeks): 71, Downloads (12 Month
Additional Information: references cited by index terms
Tools and Actions: Request Permissions Revie
Save this Article to a Binder Di
DOI Bookmark: Use this link to bookmark this Article: http:
What is a DOI?
REFERENCES
Note: OCR errors may be found in this Reference List extracted from the full text article. ACM has opted to expose the complete List rather than only correct and linked references.
http://delivery.acm.org/10.1145/370000/362947/p147-salton.pdf?key...
http://delivery.acm.org/10
Google
1 / 4
52.1%
Find
Letters to the Editor
Go To Statement Considered Harmful
Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24
EDITOR:
For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce. More recently I discovered why the use of the go to statement has such disastrous effects, and I became convinced that the go to statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.
My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.
My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our
dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.
Let us now consider repetition clauses (like, while B repeat A or repeat A until B). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don't wish to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as "induction" makes us well equipped to retain our intellectual grasp on the processes generated by repetition clauses. With the inclusion of the repetition clauses textual indices are no longer sufficient to describe the dynamic progress of the process. With each entry into a repetition clause, however, we can associate a so-called "dynamic index," inexorably counting the ordinal number of the corresponding current repetition. As repetition clauses (just as procedure calls) may be applied nestedly, we find that now the progress of the process can always be uniquely characterized by a (mixed) sequence of textual and/or dynamic indices.
The main point is that the values of these indices are outside programmer's control; they are generated (either by the write-up of his program or by the dynamic evolution of the process) whether he wishes or not. They provide independent coordinates in which to describe the progress of the process.
Why do we need such independent coordinates? The reason is—and this seems to be inherent to sequential processes—that we can interpret the value of a variable only with respect to the
start
3 Inte...
nqueens
emacs@...
Comma...
Microsof...
11:01

Back to nqueens

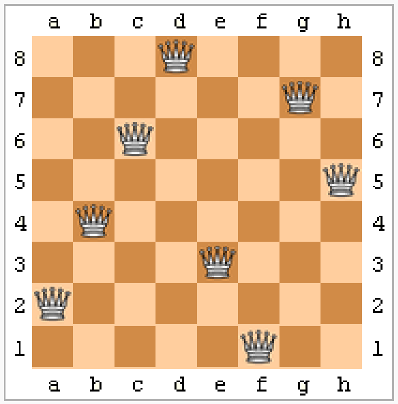
Eight queens puzzle - Wikipedia, the free encyclopedia - Windows Internet Explorer

http://en.wikipedia.org/wiki/Eight_queens_puzzle

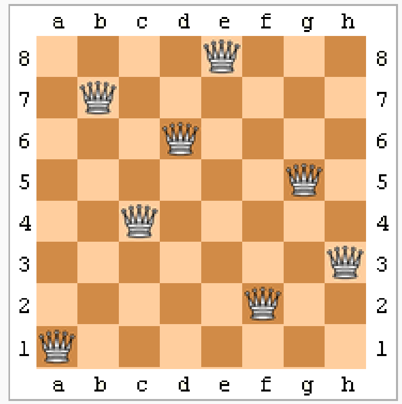
- 20 queens (remainder 8): 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 3, 1, 7, 5, 11, 9, 15, 13, 19, 17.

Solutions to the eight queens puzzle [edit]

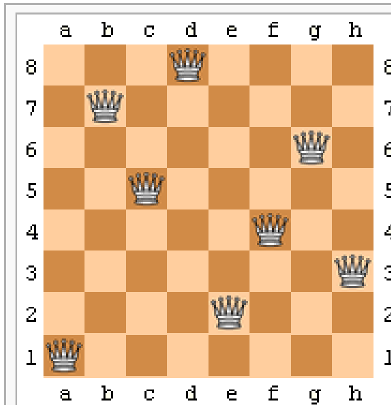
The eight queens puzzle has 92 **distinct** solutions. If solutions that differ only by symmetry operations (rotations and reflections) of the board are counted as one, the puzzle has 12 **unique** (or **fundamental**) solutions, which are presented below:



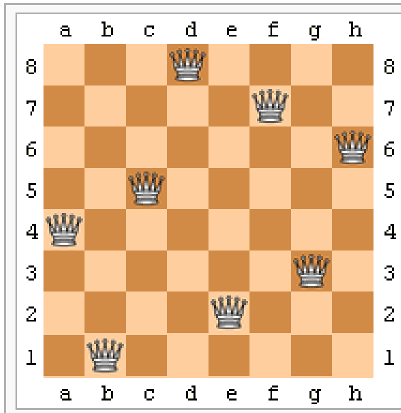
Unique solution 1



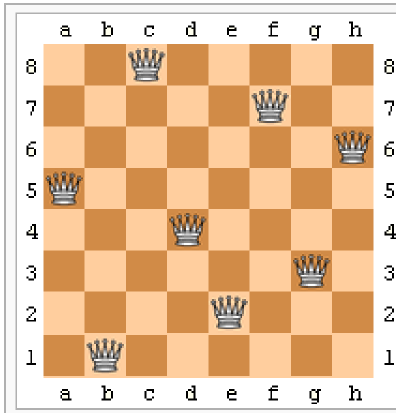
Unique solution 2



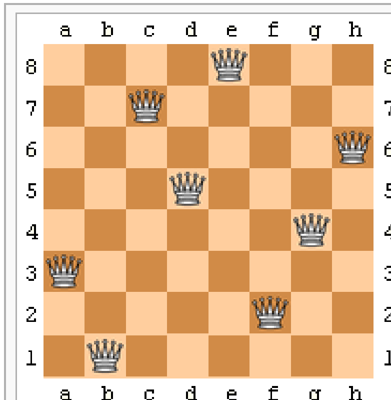
Unique solution 3



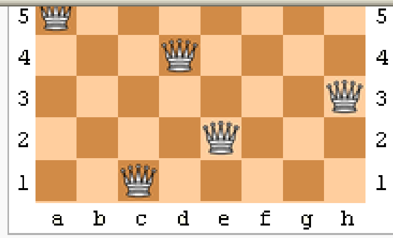
Unique solution 4



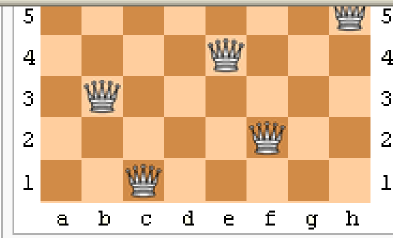
Unique solution 5



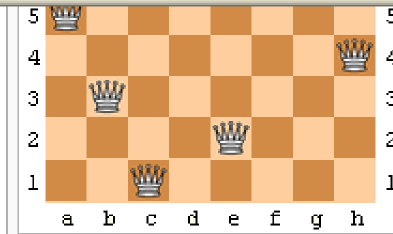
Unique solution 6



Unique solution 10



Unique solution 11



Unique solution 12

Counting solutions

[edit]

The following table gives the number of solutions for placing *n* queens on an *n* × *n* board, both unique (sequence A002562 in OEIS) and distinct (sequence A000170 in OEIS).

n:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	..	24	25	26
unique:	1	0	0	1	2	1	6	12	46	92	341	1,787	9,233	45,752	..	28,439,272,956,934	275,986,683,743,434	2,789,712,466
distinct:	1	0	0	2	10	4	40	92	352	724	2,680	14,200	73,712	365,596	..	227,514,171,973,736	2,207,893,435,808,352	22,317,699,61

Note that the six queens puzzle has fewer solutions than the five queens puzzle.

There is currently no known formula for the exact number of solutions.

Related problems

[edit]

Using pieces other than queens

For example, on an 8×8 board one can place 32 knights, or 14 bishops, 16 kings or 8 rooks, so that no two pieces attack each other. Fairy chess pieces have also been substituted for queens. In the case of knights, an easy solution is to place one on each square of a given color, since they move only to the opposite color.

Nonstandard boards

Pólya studied the *n* queens problem on a toroidal ("donut-shaped") board. In 2009 Pearson and Pearson algorithmically populated three-dimensional boards $(n \times n \times n)$ with n^2 queens, and proposed that multiples of these can yield solutions for a four-dimensional version of the puzzle.

Domination

Given an *n*×*n* board, find the **domination number**, which is the minimum number of queens (or other pieces) needed to attack or occupy

The eight queens puzzle as an exercise in algorithm design [edit]

Main article: Eight queens puzzle solutions

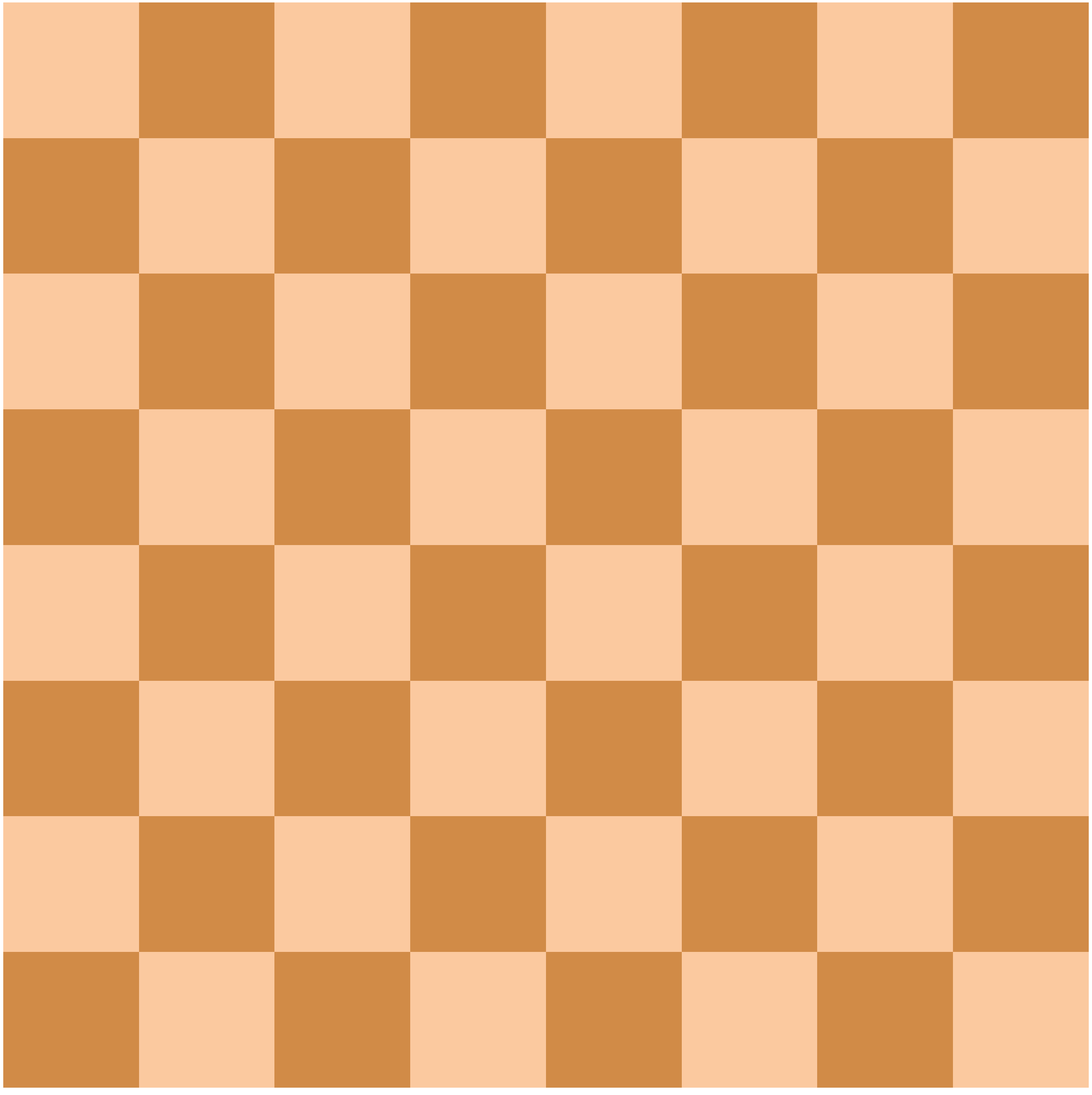
Finding all solutions to the eight queens puzzle is a good example of a simple but nontrivial problem. For this reason, it is often used as an example problem for various programming techniques, including nontraditional approaches such as constraint programming, logic programming or genetic algorithms. Most often, it is used as an example of a problem which can be solved with a recursive algorithm, by phrasing the n queens problem inductively in terms of adding a single queen to any solution to the problem of placing $n-1$ queens on an n-by-n chessboard. The induction bottoms out with the solution to the 'problem' of placing 0 queens on an n-by-n chessboard, which is the empty chessboard.

This technique is much more efficient than the naïve brute-force search algorithm, which considers all $64^8 = 2^{48}$ = 281,474,976,710,656 possible blind placements of eight queens, and then filters these to remove all placements that place two queens either on the same square (leaving only 64!/56! = 178,462,987,637,760 possible placements) or in mutually attacking positions. This very poor algorithm will, among other things, produce the same results over and over again in all the different permutations of the assignments of the eight queens, as well as repeating the same computations over and over again for the different sub-sets of each solution. A better brute-force algorithm places a single queen on each row, leading to only $8^8 = 2^{24}$ = 16,777,216 blind placements.

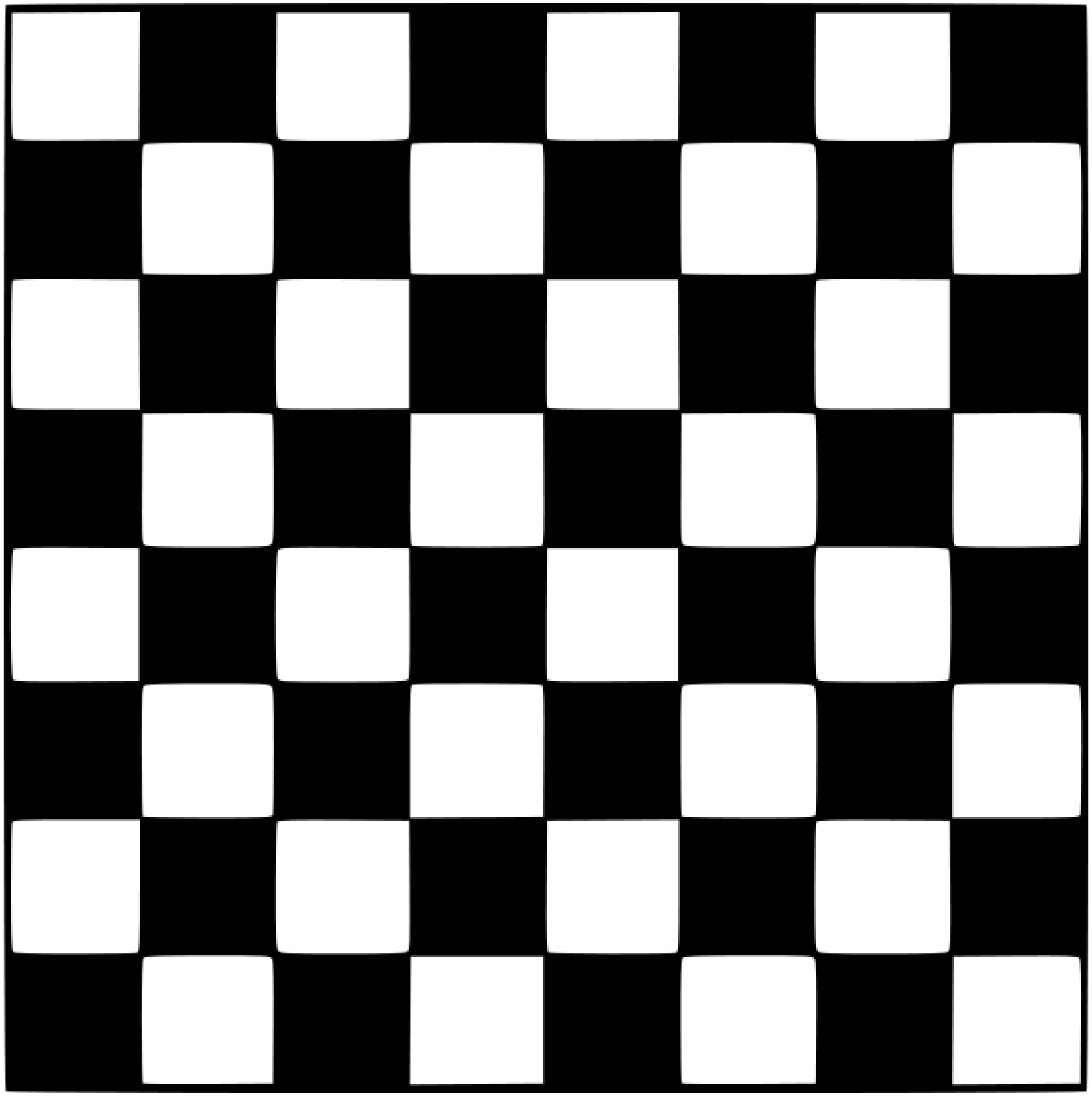
It is possible to do much better than this. One algorithm generates the permutations of the numbers 1 through 8 (of which there are 8! = 40,320), and uses the elements of each permutation as indices to place a queen on each row, guaranteeing no rook attacks. Then it rejects those boards with diagonal attacking positions. The backtracking depth-first search program, a slight improvement on the permutation method, constructs the search tree by considering one row of the board at a time, eliminating most nonsolution board positions at a very early stage in their construction. Because it rejects rook and diagonal attacks even on incomplete boards, it examines only 15,720 possible queen placements. A further improvement which examines only 5,508 possible queen placements is to combine the permutation based method with the early pruning method: the permutations are generated depth-first, and the search space is pruned if the partial permutation produces a diagonal attack. Constraint programming can also be very effective on this problem.

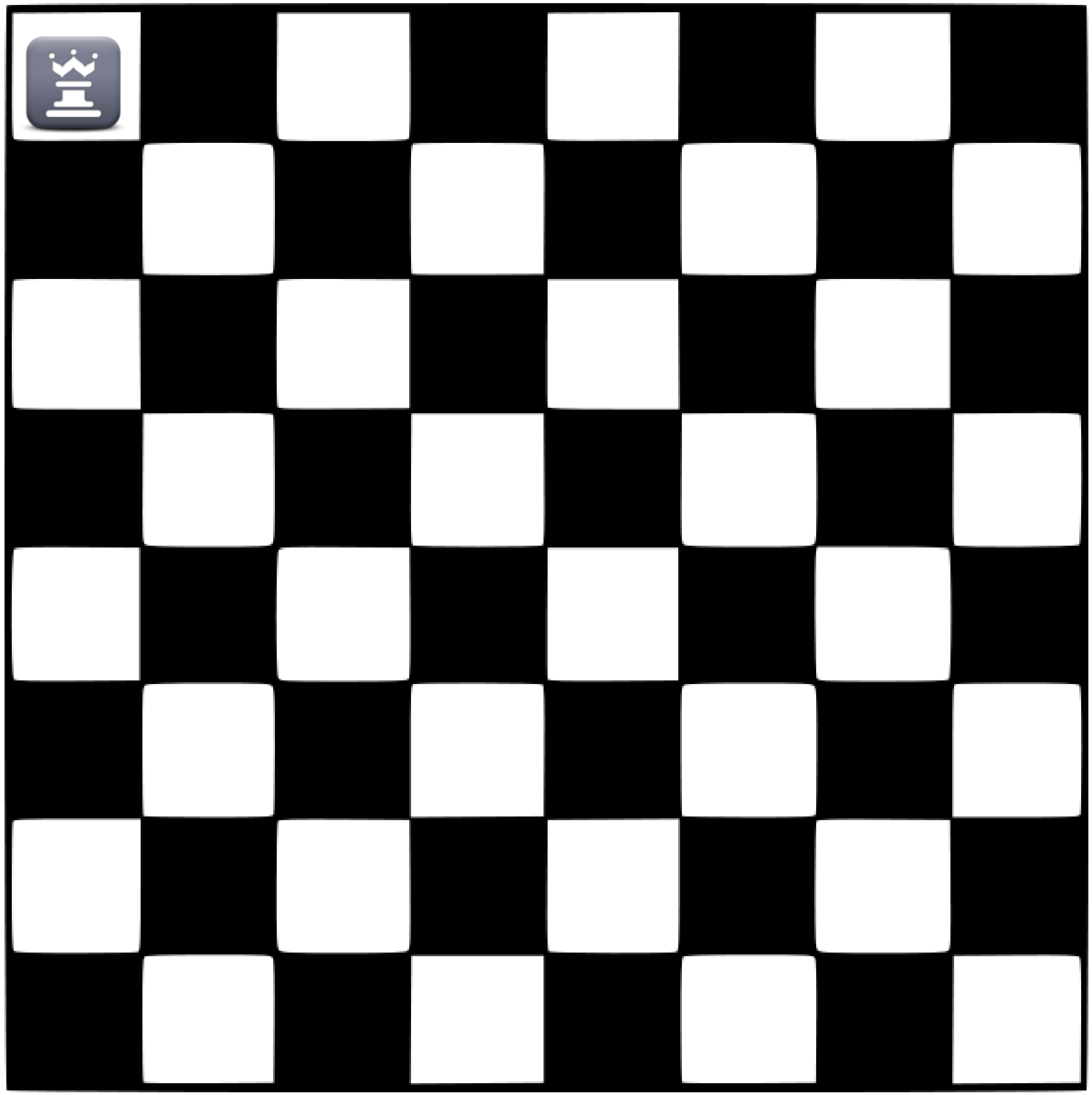
An alternative to exhaustive search is an 'iterative repair' algorithm, which typically starts with all queens on the board, for example with one queen per column. It then counts the number of conflicts (attacks), and uses a heuristic to determine how to improve the placement of the queens. The 'minimum-conflicts' heuristic — moving the piece with the largest number of conflicts to the square in the same column where the number of conflicts is smallest — is particularly effective: it solves the 1,000,000 queen problem in less than 50 steps on average. This assumes that the initial configuration is 'reasonably good' — if a million queens all start in the same row, it will obviously take at least 999,999 steps to fix it. A 'reasonably good' starting point can for instance be found by putting each queen in its own row and column such that it conflicts with the smallest number of queens already on the board.

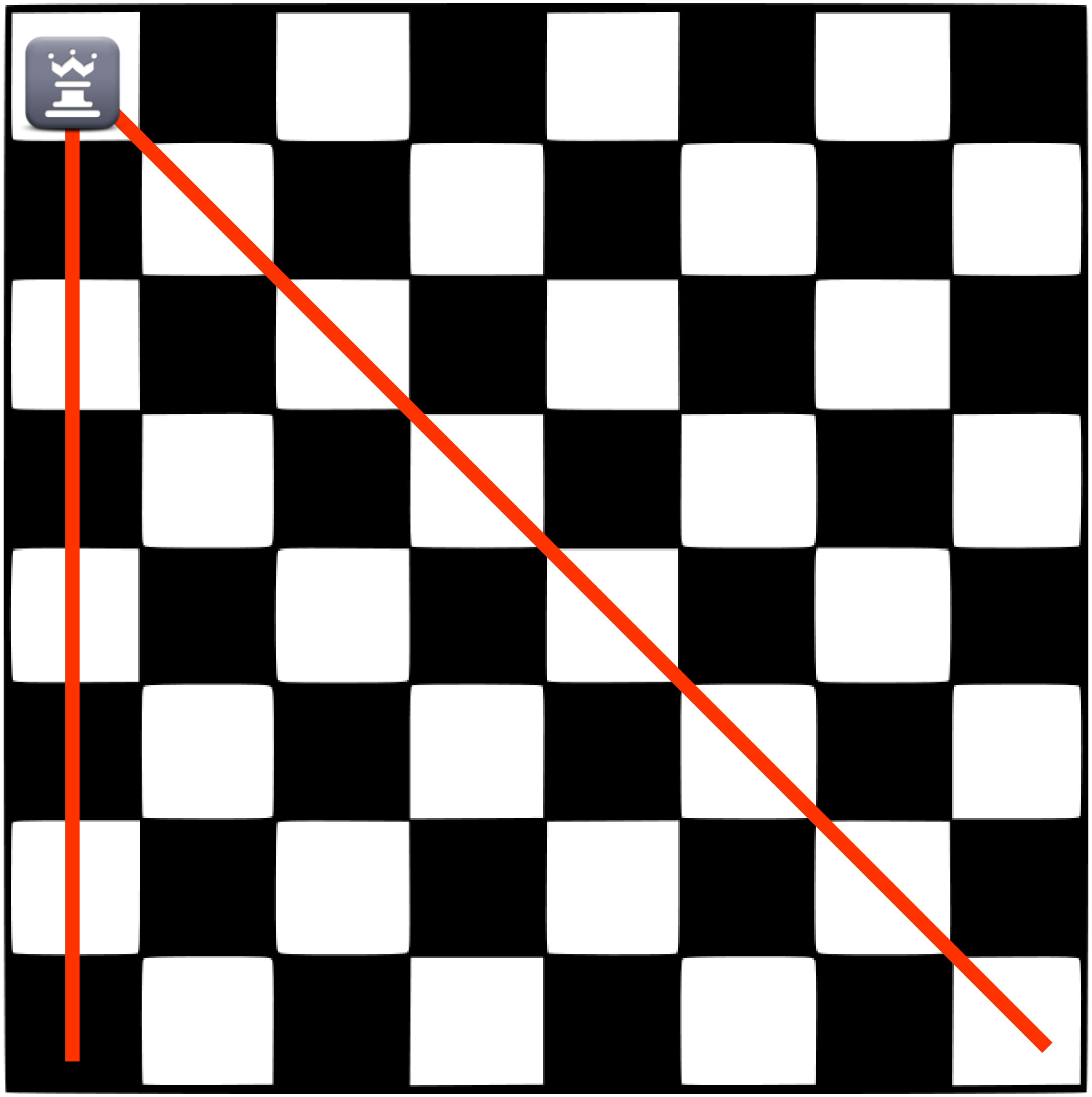
Note that 'iterative repair', unlike the 'backtracking' search outlined above, does not guarantee a solution: like all non-hillclimbing (i.e., greedy)

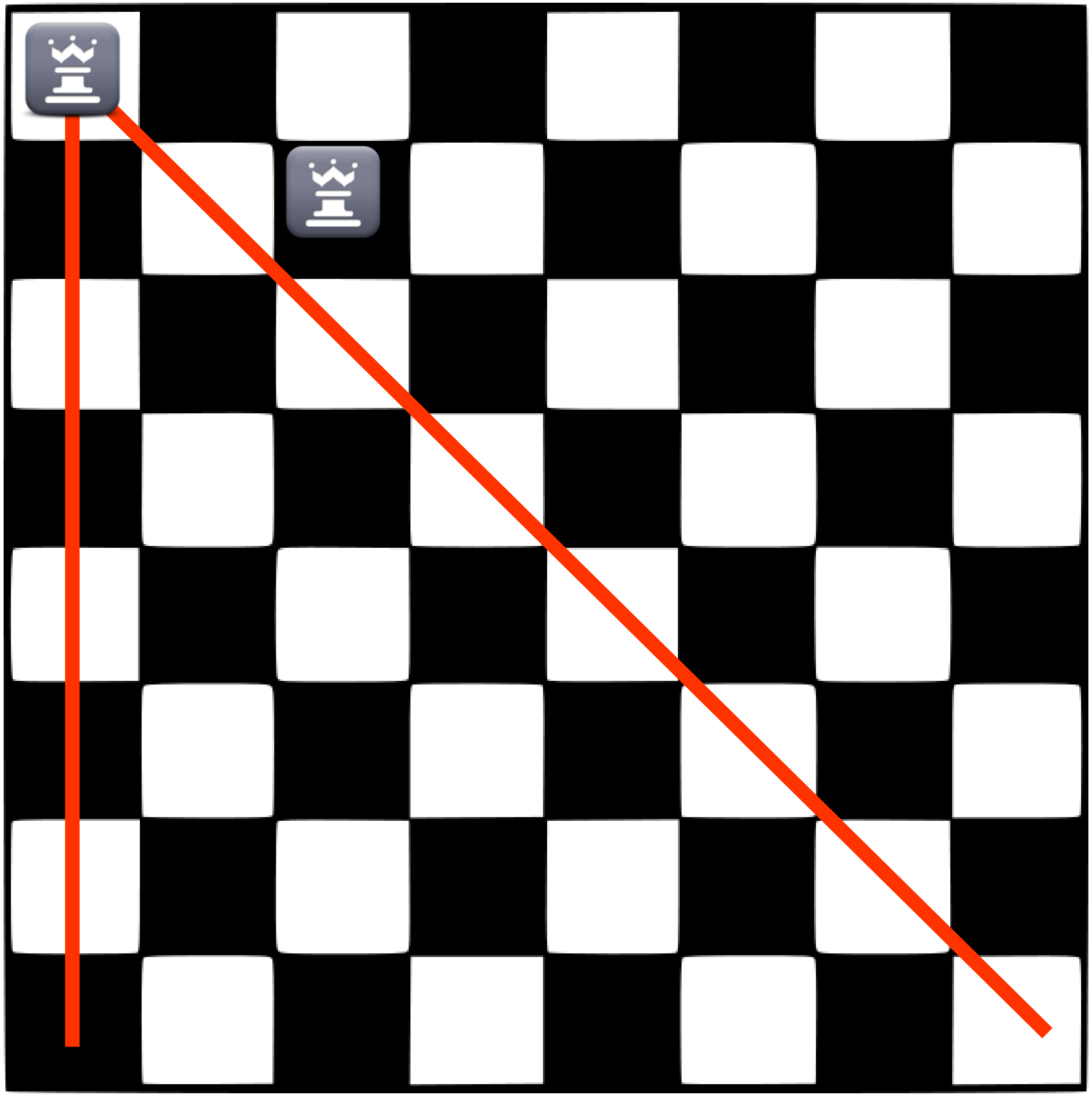


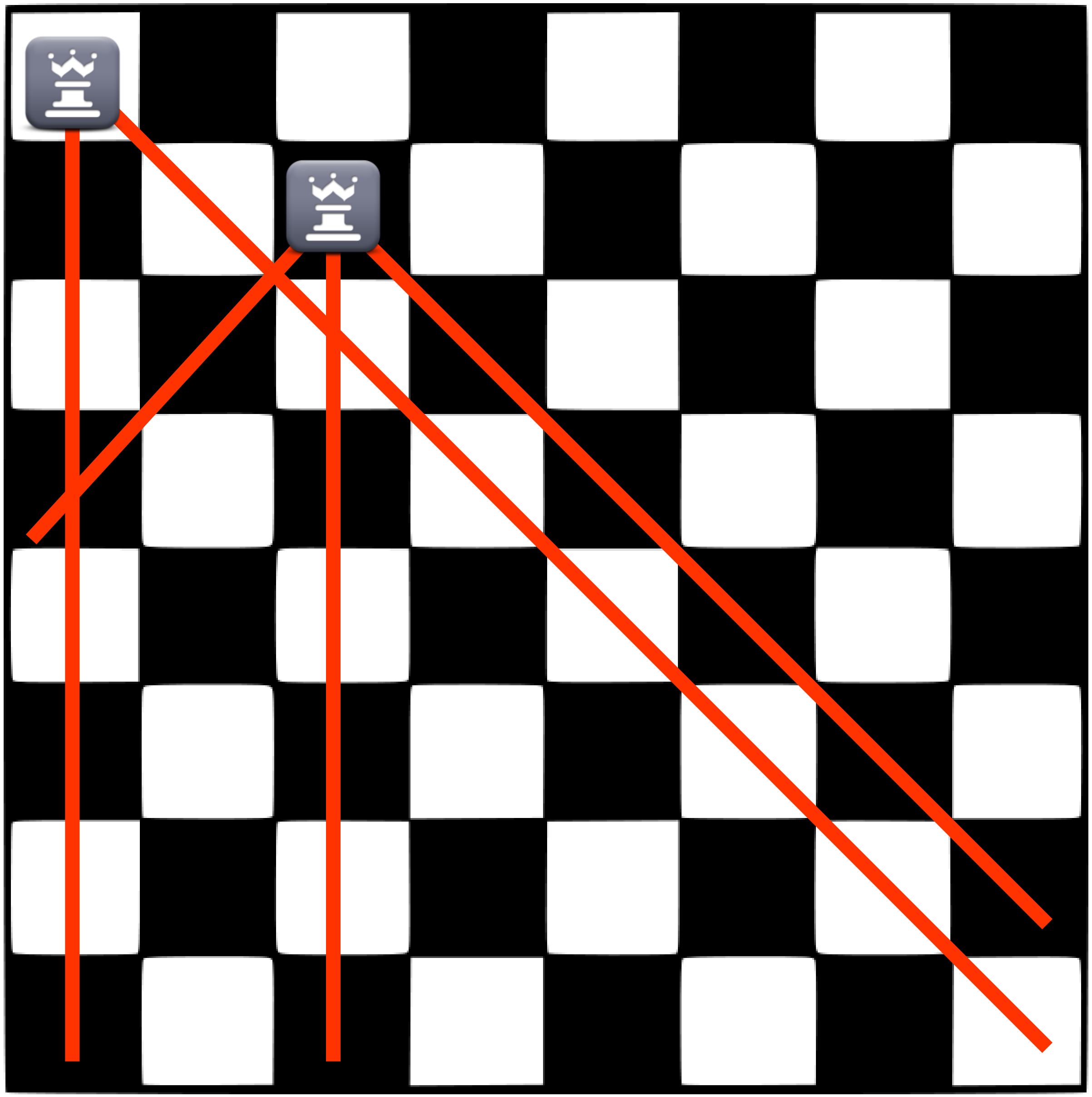
How might it go?

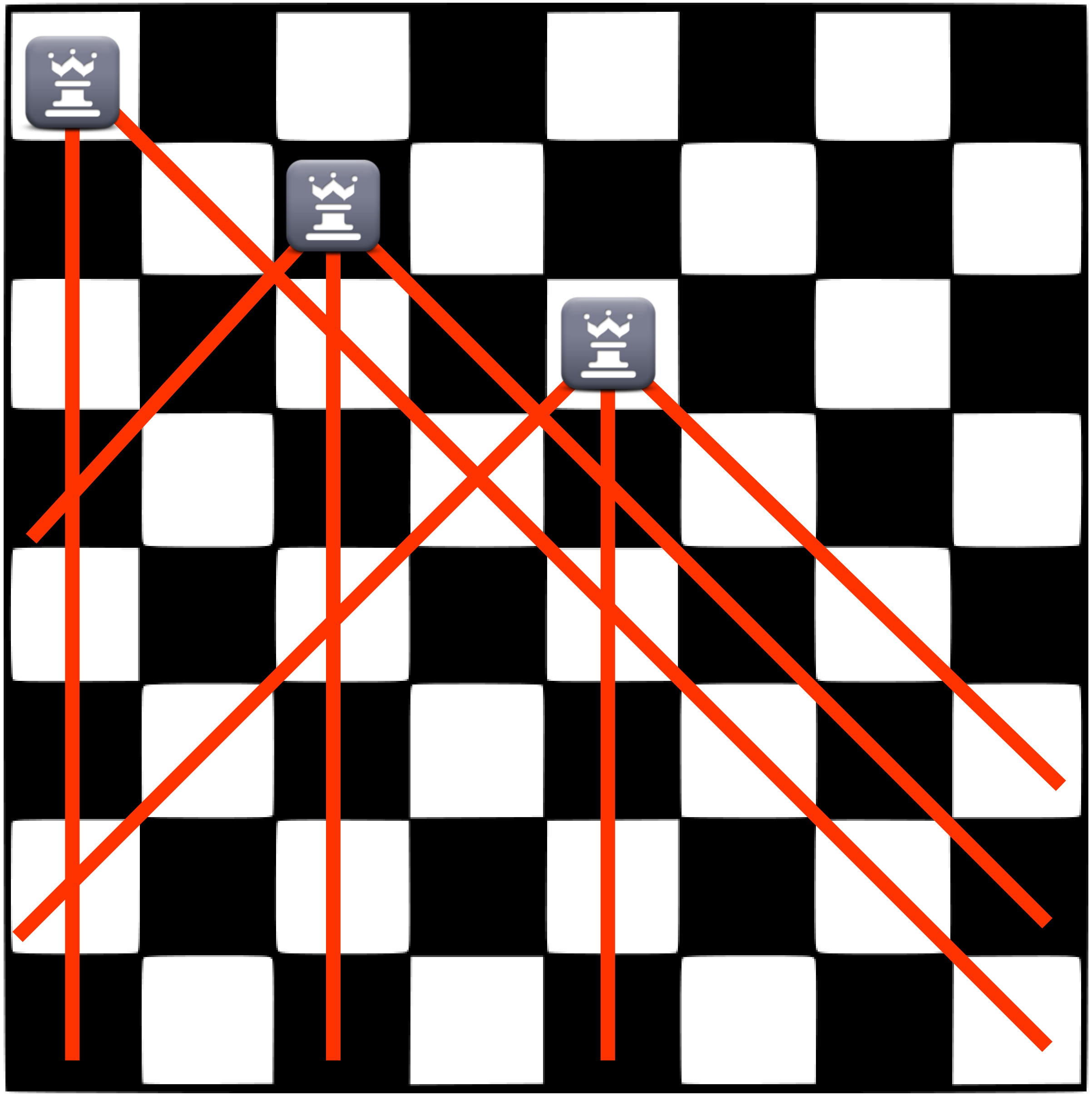


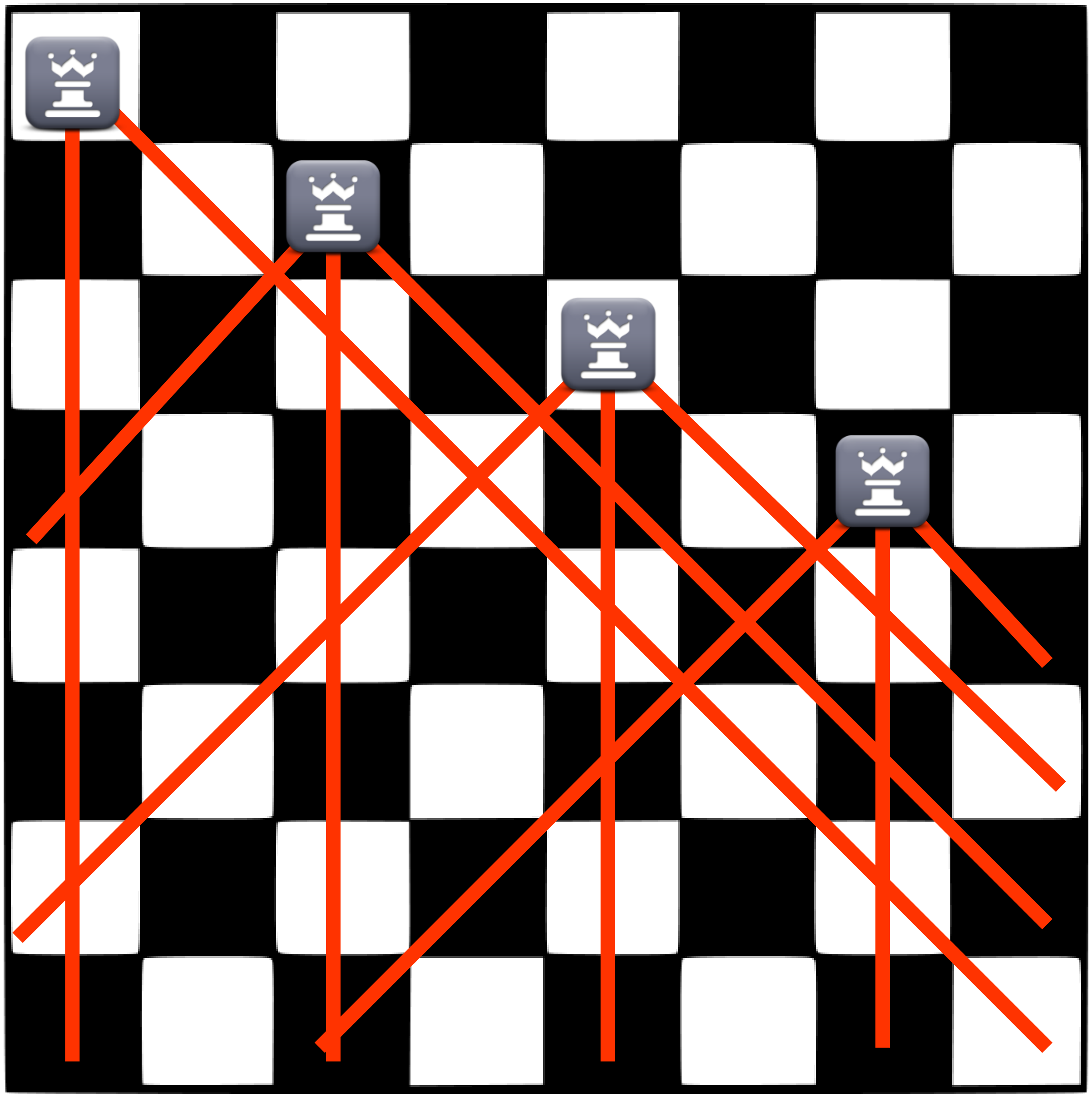


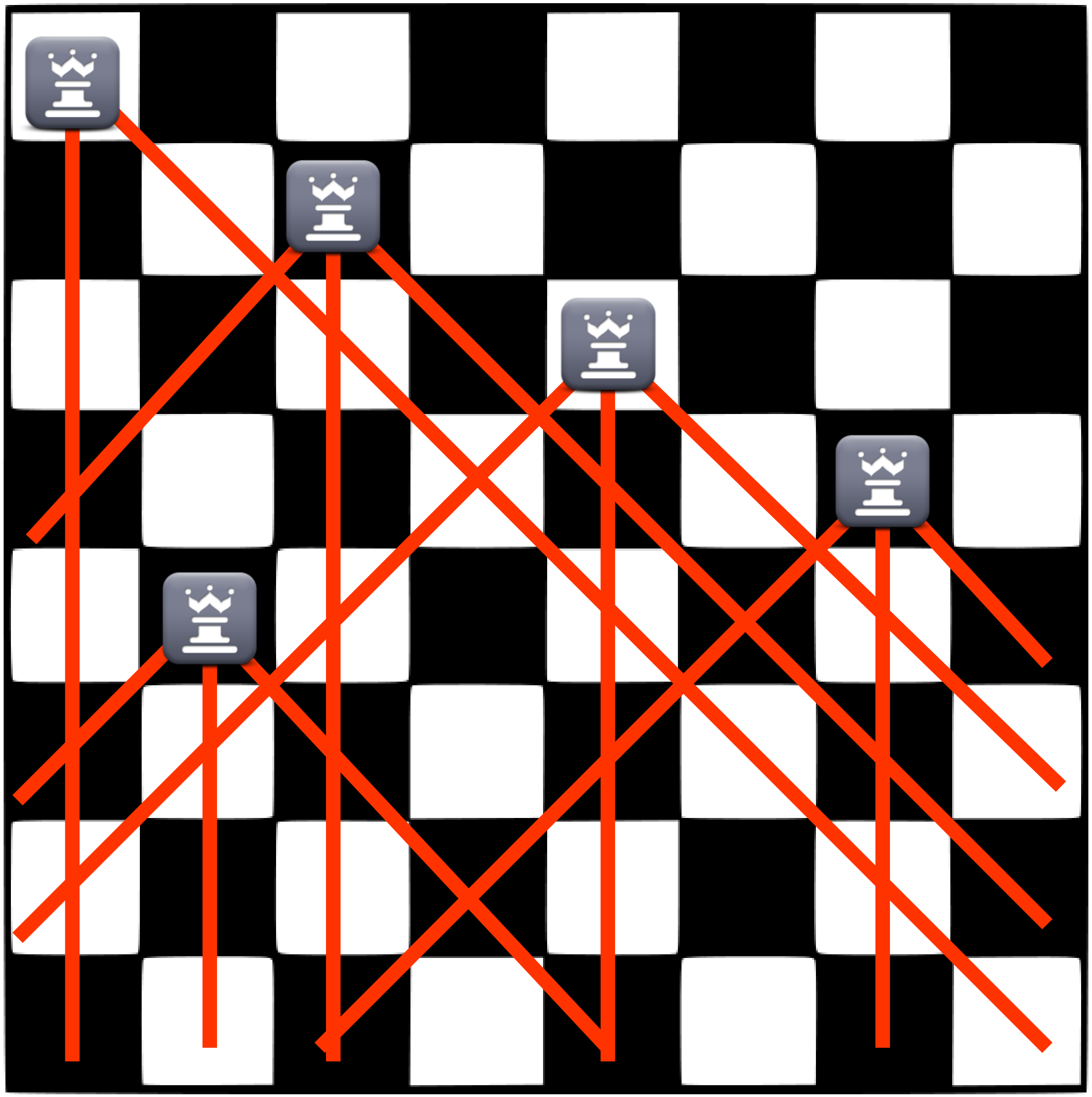


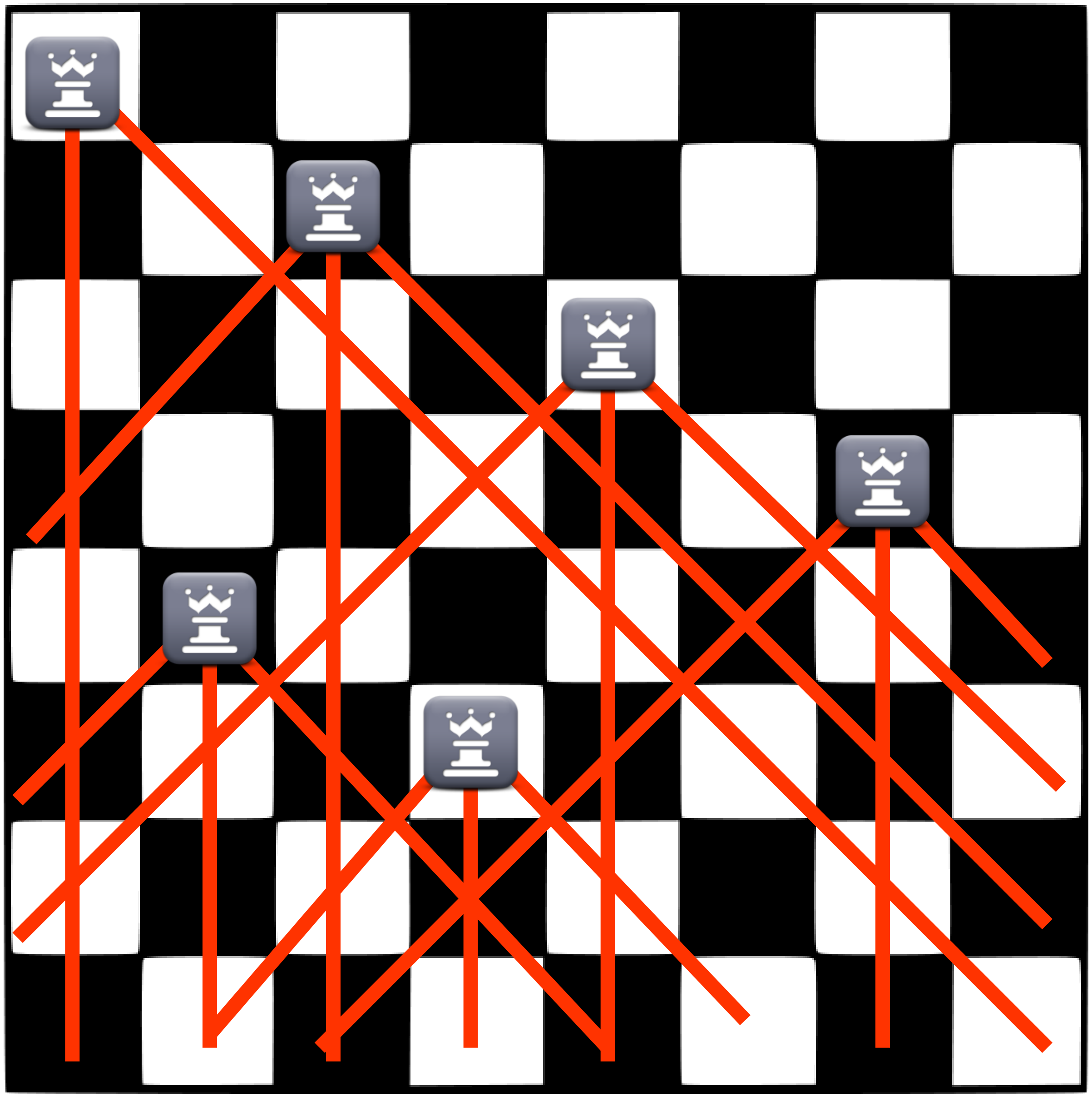


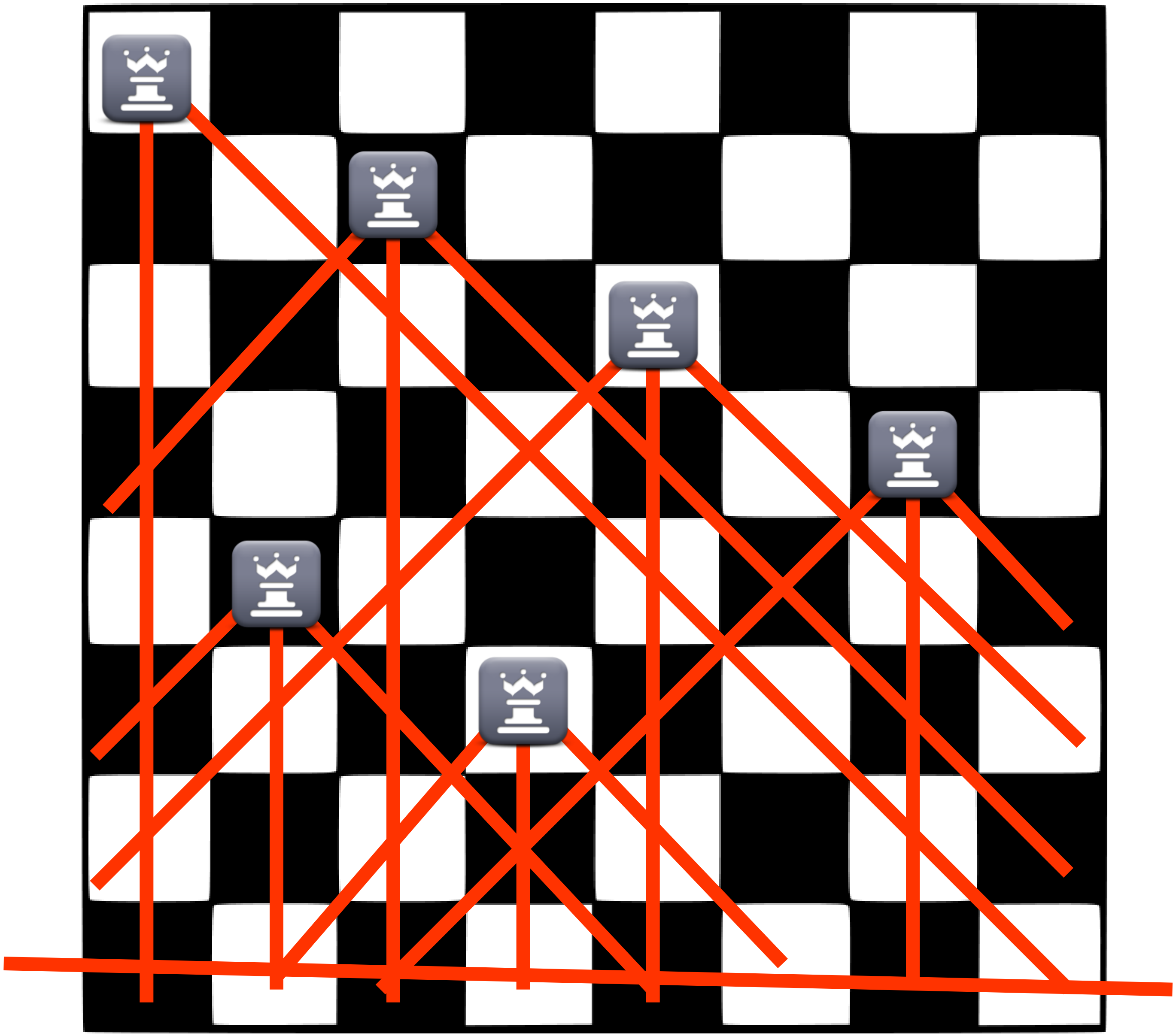




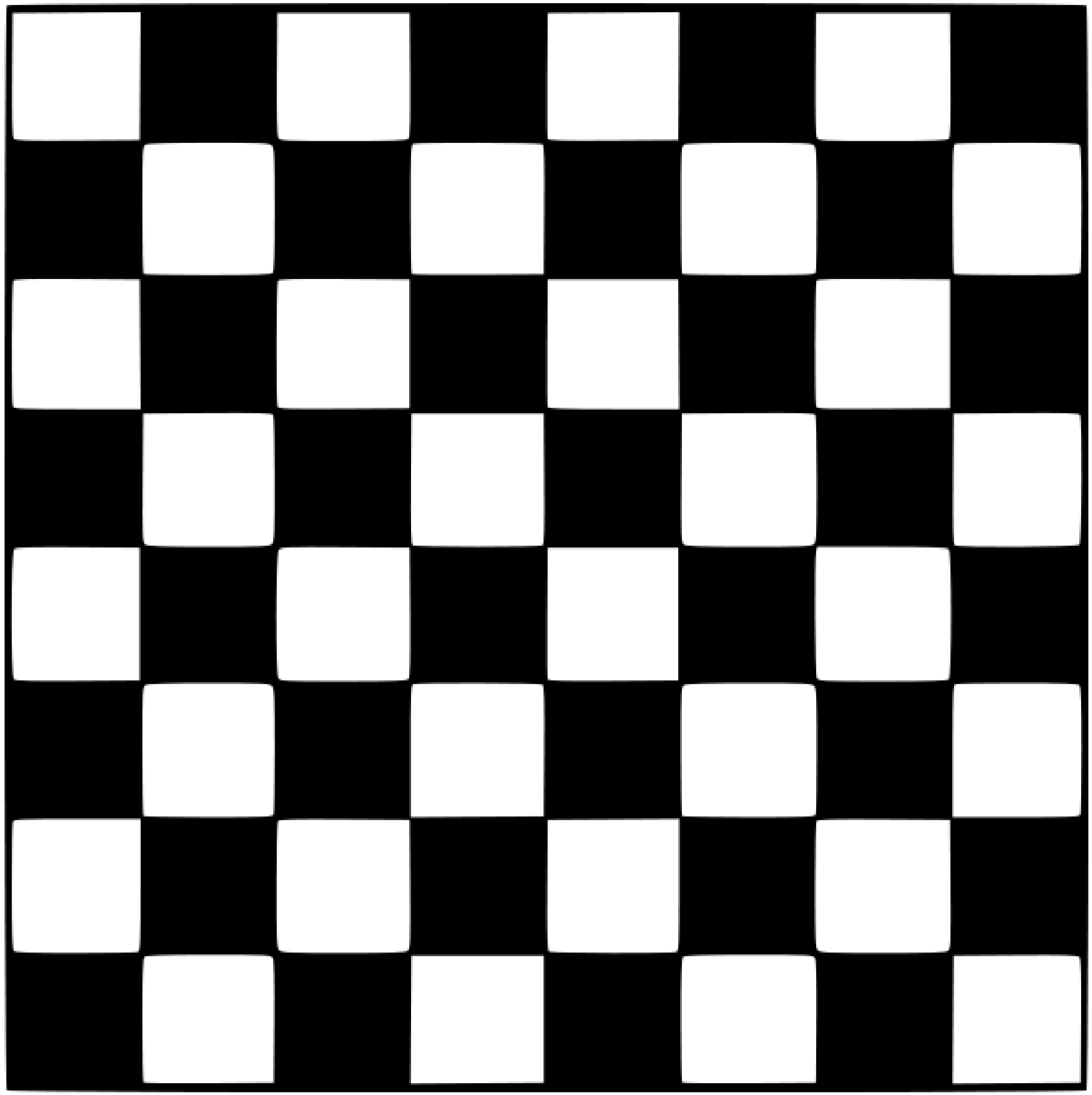


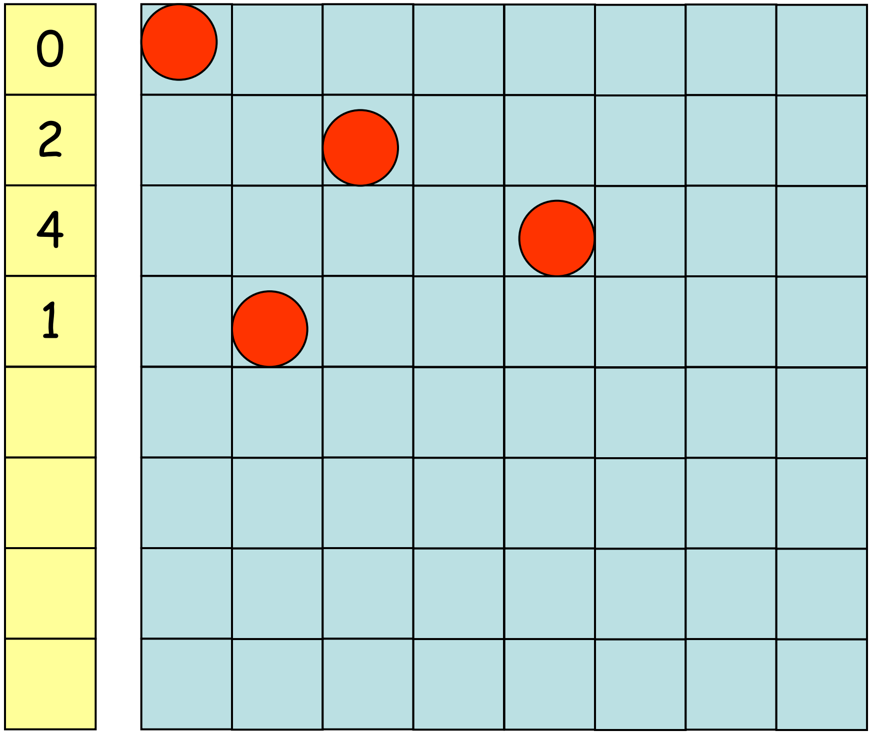




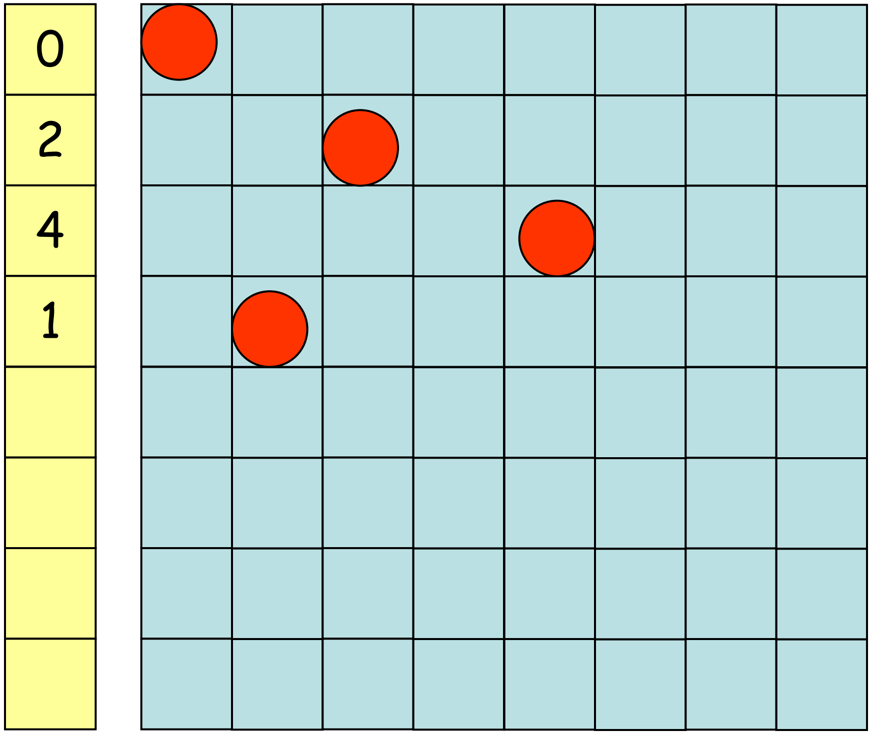


FAIL





Representation/model
8 constrained integer variables
Domains [0..7]
A variable represents a row
v[i] = j ↔ queen on row i, column j



Representation/model

Constraints

no column attacks

$v[i] \neq v[j]$ for all $i \neq j$

no diagonal attacks

$|v[i] - v[j]| \neq |i - j|$

NQueens0

emacs@BYRON

File Edit Options Buffers Tools Java Help

```java
import java.io.*;
import java.util.*;
import org.chocosolver.solver.Model;
import org.chocosolver.solver.Solver;
import org.chocosolver.solver.variables.IntVar;
import org.chocosolver.solver.constraints.IIntConstraintFactory.*;

public class NQueens0 {

    public static void main(String[] args) {

        int n           = Integer.parseInt(args[0]);
        Model model     = new Model("nqueens");
        Solver solver = model.getSolver();
        IntVar[] q      = model.intVarArray("queen",n,0,n-1);

        //
        // columns constraint
        //
        for (int i=0;i<n-1;i++)
            for (int j=i+1;j<n;j++)
                model.arithm(q[i],"!=",q[j]).post();

        //
        // diagonal constraint
        //
        for (int i=0;i<n-1;i++)
            for (int j=i+1;j<n;j++)
                model.distance(q[i],q[j],"!=",Math.abs(i-j)).post();

        boolean solved = solver.solve();

        if (solved){
```

-\--- NQueens0.java 8% L38 (Java/l Abbrev)

NQueens0

```java
import java.io.*;
import java.util.*;
import org.chocosolver.solver.Model;
import org.chocosolver.solver.Solver;
import org.chocosolver.solver.variables.IntVar;
import org.chocosolver.solver.constraints.IIntConstraintFactory.*;
```

NQueens0

```
int n          = Integer.parseInt(args[0]);
Model model    = new Model("nqueens");
Solver solver  = model.getSolver();
IntVar[] q     = model.intVarArray("queen",n,0,n-1);
```

NQueens0

```
//
// columns constraint
//
for (int i=0;i<n-1;i++)
    for (int j=i+1;j<n;j++)
        model.arithm(q[i],"!=",q[j]).post();
```

NQueens0

```
//
// diagonal constraint
//
for (int i=0;i<n-1;i++)
    for (int j=i+1;j<n;j++)
        model.distance(q[i],q[j],"!=",Math.abs(i-j)).post();
```

NQueens0

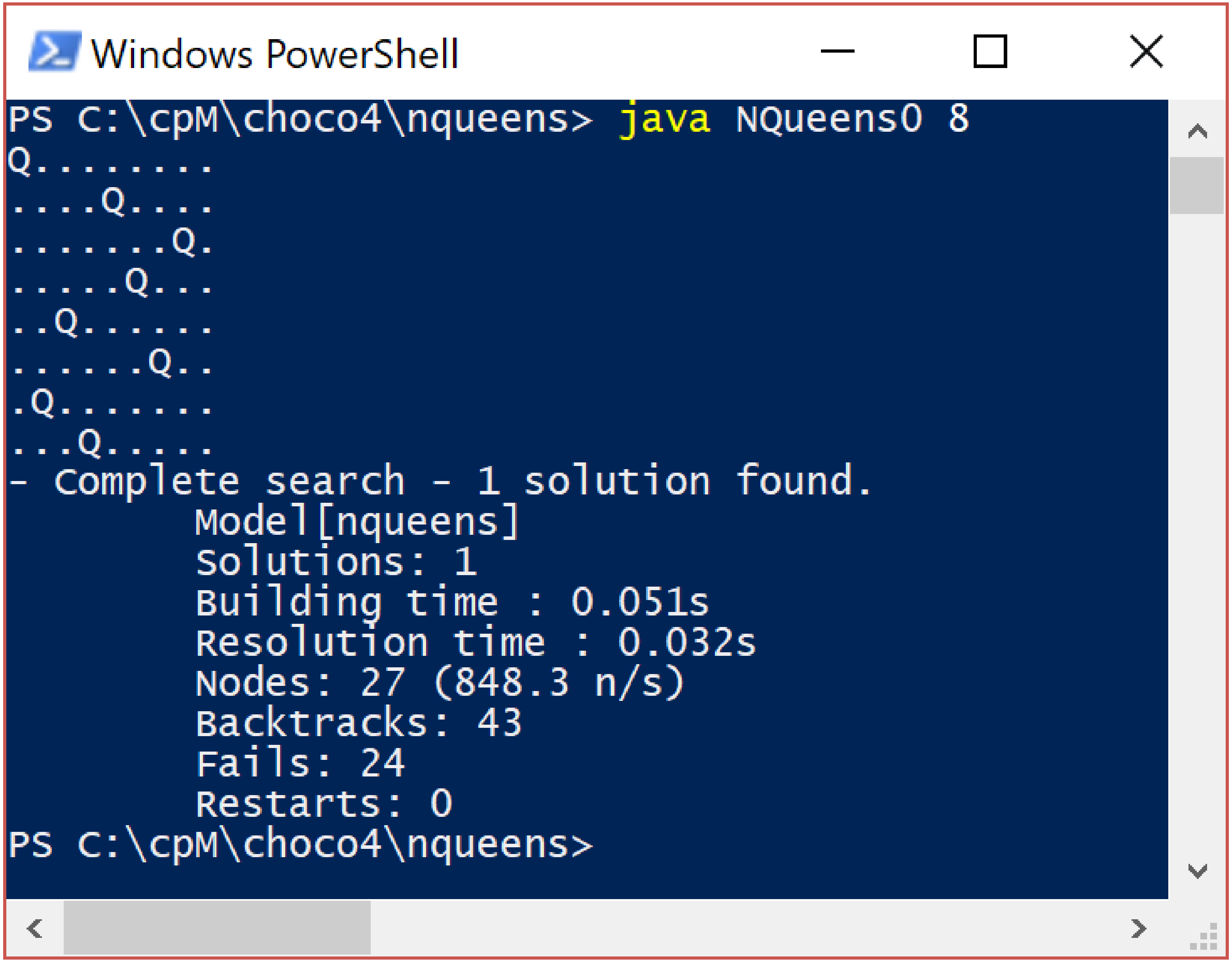
```
solver.setSearch(Search.inputOrderLBSearch(q)); // simplest
boolean solved = solver.solve();
```

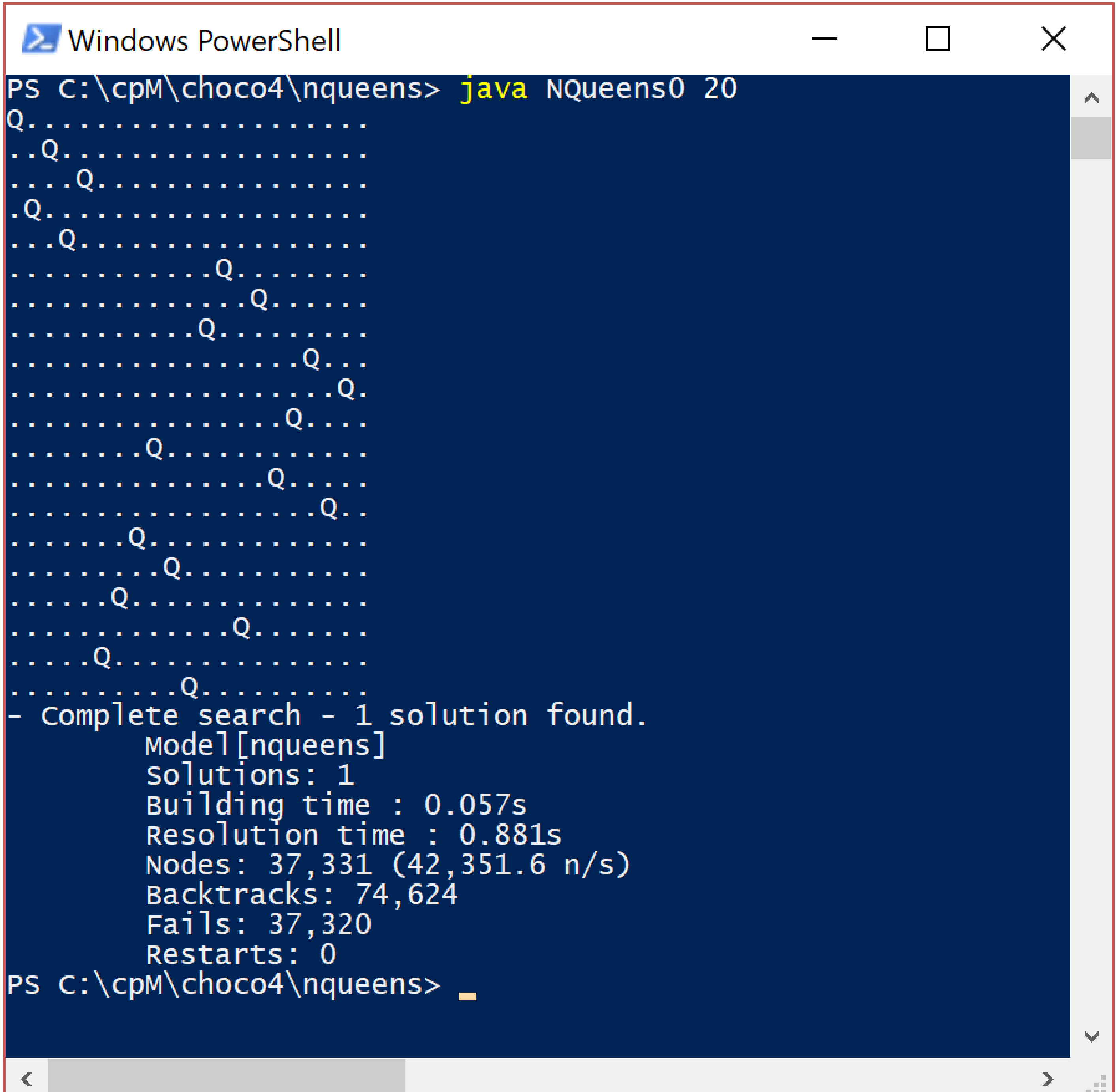
NQueens0

```
if (solved){
    for (int i=0;i<n;i++){
        for (int j=0;j<q[i].getValue();j++) System.out.print(".");
        System.out.print("Q");
        for (int j=q[i].getValue();j<n;j++) System.out.print(".");
        System.out.println();
    }
}
System.out.println(solver.getMeasures());
```

Windows PowerShell

```
PS C:\cpM\choco4\nqueens> java NQueens0 4
.Q..
...Q
Q...
..Q.
- Complete search - 1 solution found.
        Model[nqueens]
        Solutions: 1
        Building time : 0.050s
        Resolution time : 0.030s
        Nodes: 4 (131.3 n/s)
        Backtracks: 3
        Fails: 2
        Restarts: 0
PS C:\cpM\choco4\nqueens> _
```


Windows PowerShell
PS C:\cpM\choco4\nqueens> java NQueens0 8
Q.......
....Q...
.......Q.
.....Q...
..Q.....
......Q..
.Q.......
...Q.....
- Complete search - 1 solution found.
Model[nqueens]
Solutions: 1
Building time : 0.051s
Resolution time : 0.032s
Nodes: 27 (848.3 n/s)
Backtracks: 43
Fails: 24
Restarts: 0
PS C:\cpM\choco4\nqueens>


Windows PowerShell
PS C:\cpM\choco4\nqueens> java NQueens0 20
Q...................
..Q.................
....Q...............
.Q..................
...Q................
............Q.......
..............Q.....
...........Q........
.................Q..
...................Q
................Q...
........Q...........
...............Q....
..................Q.
.......Q............
.........Q..........
......Q.............
.............Q......
.....Q..............
..........Q.........
- Complete search - 1 solution found.
Model[nqueens]
Solutions: 1
Building time : 0.057s
Resolution time : 0.881s
Nodes: 37,331 (42,351.6 n/s)
Backtracks: 74,624
Fails: 37,320
Restarts: 0
PS C:\cpM\choco4\nqueens>

NQueens1

```
solver.setSearch(Search.minDomLBSearch(q)); // fail-first
boolean solved = solver.solve();
```

NQueens1

```
PS C:\cpM\choco4\nqueens> java NQueens1 20
Q...................
..Q.................
....Q...............
.............Q......
................Q...
...Q................
...............Q....
......Q.............
...........Q........
.................Q..
..............Q.....
..................Q.
.....Q..............
.........Q..........
...................Q
..........Q.........
.......Q............
.Q..................
............Q.......
........Q...........
- Complete search - 1 solution found.
        Model[nqueens]
        Solutions: 1
        Building time : 0.053s
        Resolution time : 0.075s
        Nodes: 44 (582.8 n/s)
        Backtracks: 60
        Fails: 33
        Restarts: 0
PS C:\cpM\choco4\nqueens> t
```

```
int solutions = 0;
while (solver.solve()) solutions++;

System.out.println("solutions: "+ solutions);
System.out.println(solver.getMeasures());
```

NQueens3

Windows PowerShell

```
PS C:\cpM\choco4\nqueens> java NQueens3 8
solutions: 92
- Complete search - 92 solution(s) found.
        Model[nqueens]
        Solutions: 92
        Building time : 0.057s
        Resolution time : 0.089s
        Nodes: 455 (5,096.8 n/s)
        Backtracks: 727
        Fails: 272
        Restarts: 0
PS C:\cpM\choco4\nqueens> java NQueens3 12
solutions: 14200
- Complete search - 14,200 solution(s) found.
        Model[nqueens]
        Solutions: 14,200
        Building time : 0.058s
        Resolution time : 2.017s
        Nodes: 116,973 (58,003.8 n/s)
        Backtracks: 205,547
        Fails: 88,574
        Restarts: 0
PS C:\cpM\choco4\nqueens> t
```

Version 3.2 (2-core)

N:	<------ N-Queens Solutions Total	-----> Unique	<---- time ----> days hh:mm:ss.--
5:	10	2	0.00
6:	4	1	0.00
7:	40	6	0.00
8:	92	12	0.00
9:	352	46	0.00
10:	724	92	0.00
11:	2680	341	0.00
12:	14200	1787	0.00
13:	73712	9233	0.02
14:	365596	45752	0.05
15:	2279184	285053	0.22
16:	14772512	1846955	1.47
17:	95815104	11977939	9.42
18:	666090624	83263591	1:11.21
19:	4968057848	621012754	8:32.54
20:	39029188884	4878666808	1:10:55.48
21:	314666222712	39333324973	9:24:40.50

AMD Athlon(tm) Dual Core Processor 5050e 2.60 GHz
Microsoft Visual C++ 2008 Express Edition with SP1
Windows SDK for Windows Server 2008 and .NET

```
//
// columns constraint ... allDiff!
//
model.allDifferent(q).post();
```

```
//
// columns constraint ... allDiff!
//
model.allDifferent(q).post();
```

NQueens 1

N = 200

Windows PowerShell

```
...................................Q..........
..................................
.............................................Q
..................................
..............................................
..................................
- Complete search - 1 solution found.
        Model[nqueens]
        Solutions: 1
        Building time : 0.174s
        Resolution time : 56.920s
        Nodes: 147,014 (2,582.8 n/s)
        Backtracks: 293,662
        Fails: 146,838
        Restarts: 0
PS C:\cpM\choco4\nqueens>
```

NQueens 2

N = 200

Windows PowerShell

```
..................................
.............................................Q
..................................
..............................................
..................................
- Complete search - 1 solution found.
        Model[nqueens]
        Solutions: 1
        Building time : 0.139s
        Resolution time : 2.742s
        Nodes: 306 (111.6 n/s)
        Backtracks: 250
        Fails: 128
        Restarts: 0
PS C:\cpM\choco4\nqueens> _
```

Harder questions

Could we have a different representation?

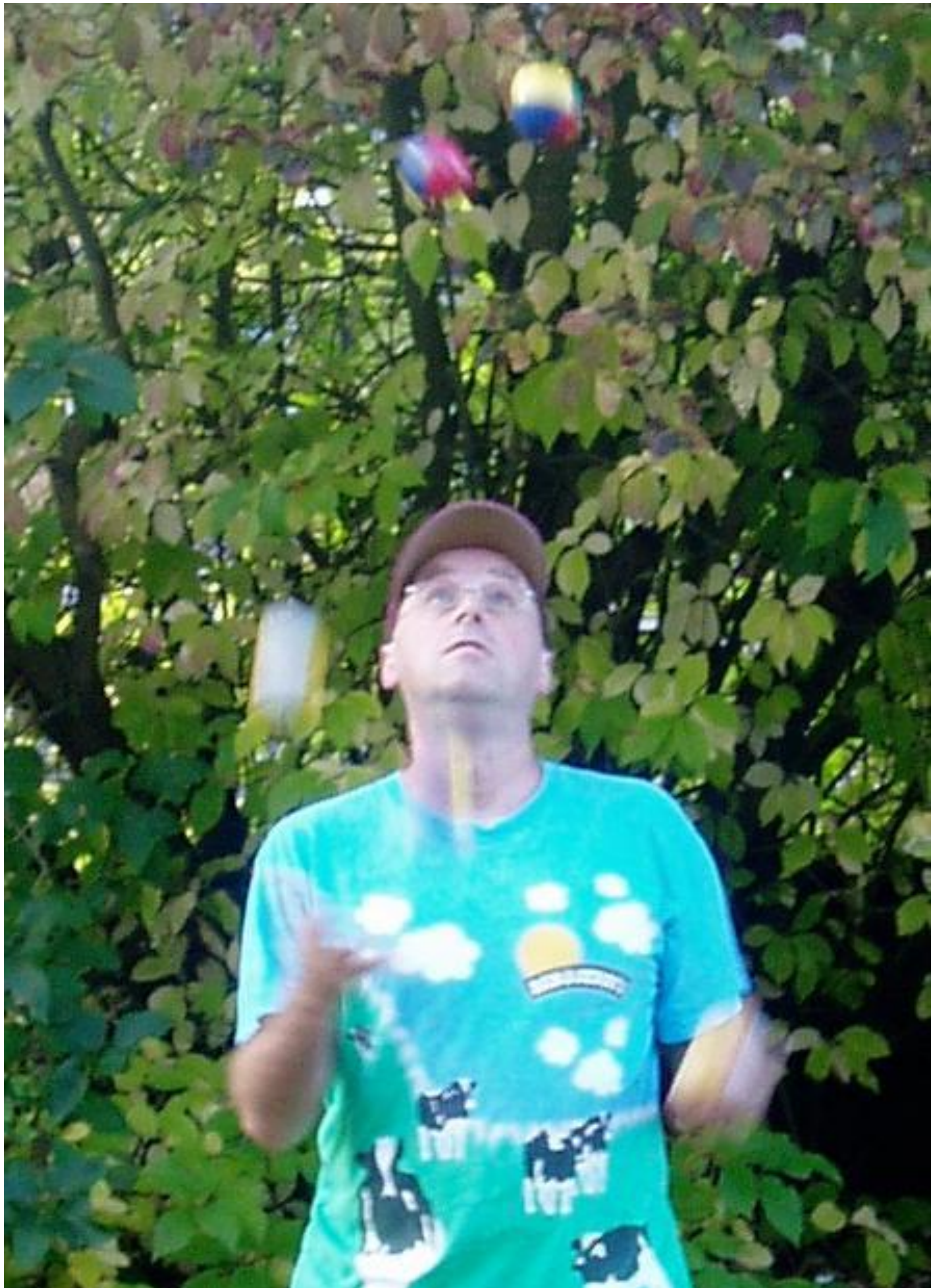
Could we solve it a different way (other than systematic search) ?

Does the problem get harder or easier as it gets larger?

What happens if we start with some queens already on the board?

Harder questions

Could we view nqueens differently, maybe as a permutation problem?



File Edit View History Bookmarks Tools Help

Search (org.choco-solver-choc | user_guide-4.0.8.pdf | Shapur I - Wikipedia | n-Queens Completion is NP-Co

https://blogs.cs.st-andrews.ac.uk/csblog/2017/08/31/n-queens-completion-is-n

Search

University of St Andrews

School of Computer Science

Computer Science Blog

About | School of Computer Science | University of St Andrews

n-Queens Completion is NP-Complete

Peter Nightingale and Ian Gent at Falkland Palace, Wednesday, 17 August 2017. ©Stuart Nicol Photography, 2017

Ian Gent, Christopher Jefferson and Peter Nightingale have shown that a classic chess puzzle is NP-Complete. Their paper "Complexity of *n*-Queens Completion" was published in the Journal of Artificial Intelligence Research on August 30.

The *n*-Queens puzzle is a classic chess problem: given a chessboard of size *n* by *n*, can you place *n* queens so that no two queens attack each other? That is, can you place the queens with no two queens are on the same row, column, or diagonal? The *n*-Queens puzzle has long been known to be simple to solve: you can solve the problem for all *n* except 2 and 3, and solutions for all other *n* can be described in a few lines. This very

St Andrews / CS

News and events from the School of Computer Science at the University of St Andrews.

Search …

Tags

THE COURIER.CO.UK

Challenge to end stalemate over $1 million maths puzzle

by Aileen Robertson August 31 2017, 12.43pm

File Edit View History Bookmarks Tools Help

Search (org.choco-solver-choco | user_guide-4.0.8.pdf | Shapur I - Wikipedia | Challenge to end stalemate ov | Millennium Problems | Clay Ma

https://www.claymath.org/millennium-problems

Search

CMI

ABOUT PROGRAMS MILLENNIUM PROBLEMS PEOPLE PUBLICATIONS EVENTS EUCLID

Millennium Problems

Yang–Mills and Mass Gap

Experiment and computer simulations suggest the existence of a "mass gap" in the solution to the quantum versions of the Yang-Mills equations. But no proof of this property is known.

Riemann Hypothesis

The prime number theorem determines the average distribution of the primes. The Riemann hypothesis tells us about the deviation from the average. Formulated in Riemann's 1859 paper, it asserts that all the 'non-obvious' zeros of the zeta function are complex numbers with real part 1/2.

P vs NP Problem

If it is easy to check that a solution to a problem is correct, is it also easy to solve the problem? This is the essence of the P vs NP question. Typical of the NP problems is that of the Hamiltonian Path Problem: given N cities to visit, how can one do this without visiting a city twice? If you give me a solution, I can easily check that it is correct. But I cannot so easily find a solution.

Does the nqueens get easier as n increases?

File Edit View History Bookmarks Tools Help

Search (org.choco-sol | user_guide-4.0.8.pdf | Shapur I - Wikipedia | Challenge to end stal | Millennium Problems | 1990-Solving Large-S

www.aaai.org/Papers/AAAI/1990/AAAI90-003.pdf

1 of 8 Automatic Zoom

Solving Large-Scale Constraint Satisfaction and Scheduling Problems Using a Heuristic Repair Method

Steven Minton[1] **Mark D. Johnston**[2] **Andrew B. Philips**[1] **Philip Laird**[3]

[1]Sterling Federal Systems
NASA Ames Research Center
Mail Stop: 244-17
Moffett Field, CA 94035

[2]Space Telescope Science Institute
3700 San Martin Drive
Baltimore, MD 21218

[3]AI Research Branch
NASA Ames Research Center
Mail Stop: 244-17
Moffett Field, CA 94035

Abstract

This paper describes a simple heuristic method for solving large-scale constraint satisfaction and scheduling problems. Given an initial assignment for the variables in a problem, the method operates by searching though the space of possible re-

that minimizes the number of outstanding constraint violations.

The work described in this paper was inspired by a surprisingly effective neural network developed by Adorf and Johnston for scheduling the use of the Hubble Space Telescope[2,13]. Our heuristic CSP method was distilled from an analysis of the network, and


File Edit View History Bookmarks Tools Help

Search (org.chocc | user_guide-4.0.8.pdf | Shapur I - Wikipe | Challenge to end | Millennium Probl | Min-conflicts

https://en.wikipedia.org

Search

Not logged in Talk Contributions Create account Log in

WIKIPEDIA
The Free Encyclopedia

Article Talk | Read Edit View history | Search Wikipedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction

Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools

What links here

Min-conflicts algorithm

From Wikipedia, the free encyclopedia

In computer science, the **min conflicts algorithm** is a search algorithm or heuristic method to solve constraint satisfaction problems (CSP).

Given an initial assignment of values to all the variables of a CSP, the algorithm randomly selects a variable from the set of variables with conflicts violating one or more constraints of the CSP.[1] Then it assigns to this variable the value that minimizes the number of conflicts. If there is more than one value with a minimum number of conflicts, it chooses one randomly. This process of random variable selection and min-conflict value assignment is iterated until a solution is found or a pre-selected maximum number of iterations is reached.

Because a CSP can be interpreted as a local search problem when all the variables have an assigned value (called a complete state), the min conflicts algorithm can be seen as a repair heuristic[2] that chooses the state with the minimum number of conflicts.

Contents [hide]

Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export

Download as PDF
Printable version

Languages

Add links

Algorithm [edit]

```
algorithm MIN-CONFLICTS is
    input: csp, A constraint satisfaction problem.
           max_steps, The number of steps allowed before giving up.
           current_state, An initial assignment of values for the variables in the csp.
    output: A solution set of values for the variable or failure.

    for i ← 1 to max_steps do
        if current_state is a solution of csp then
            return current_state
        set var ← a randomly chosen variable from the set of conflicted variables CONFLICTED[csp]
        set value ← the value v for var that minimizes CONFLICTS(var,v,current_state,csp)
        set var ← value in current_state

    return failure
```

Although not specified in the algorithm, a good initial assignment can be critical for quickly approaching a solution. Use a greedy algorithm with some level of randomness and allow variable assignment to break constraints when no other assignment will suffice. The randomness helps min-conflicts avoid local minima created by the greedy algorithm's initial assignment. In fact, Constraint Satisfaction Problems that respond best to a min-conflicts solution do well where a greedy algorithm almost solves the problem. Map coloring problems do poorly with Greedy Algorithm as well as Min-Conflicts. Sub areas of the map tend to hold their colors stable and min conflicts cannot hill climb to break out of the local minimum. The CONFLICTS function counts the number of constraints violated by a particular object, given that the state of the rest of the assignment is known.

File Edit View History Bookmarks Tools Help

Search (org.choco-sol | user_guide-4.0.8.pdf | Shapur I - Wikipedia | Challenge to end stal | Millennium Problems | Min-conflicts algorith

https://en.wikipedia.org/wiki/Min-conflicts_algorithm

Search

Problems that respond best to a min-conflicts solution do well where a greedy algorithm almost solves the problem. Map coloring problems do poorly with Greedy Algorithm as well as Min-Conflicts. Sub areas of the map tend to hold their colors stable and min conflicts cannot hill climb to break out of the local minimum. The `CONFLICTS` function counts the number of constraints violated by a particular object, given that the state of the rest of the assignment is known.

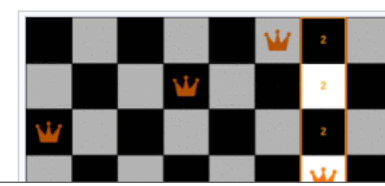
History [edit]

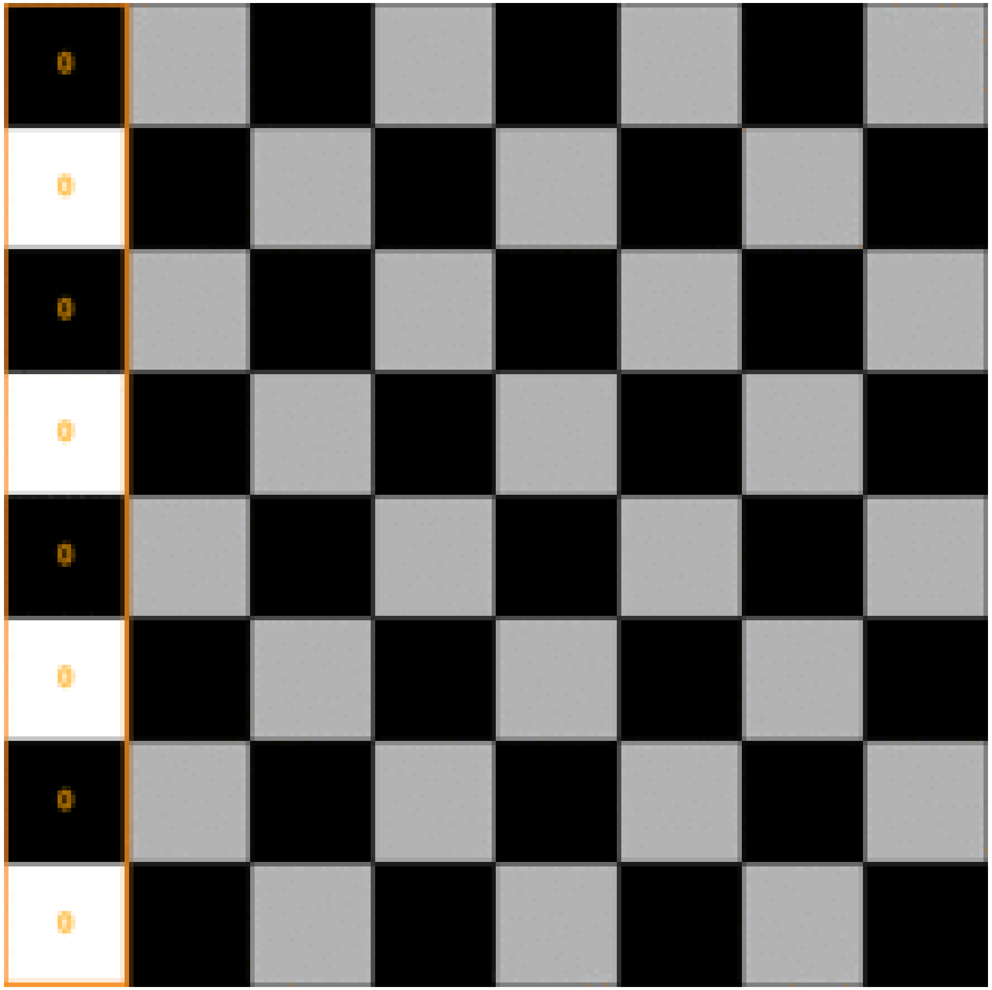
Although Artificial Intelligence and Discrete Optimization had known and reasoned about Constraint Satisfaction Problems for many years, it was not until the early 1990s that this process for solving large CSPs had been codified in algorithmic form. Early on, Mark Johnston of the Space Telescope Science Institute looked for a method to schedule astronomical observations on the Hubble Space Telescope. In collaboration with Hans-Martin Adorf of the Space Telescope European Coordinating Facility, he created a neural network capable of solving a toy *n*-queens problem (for 1024 queens).[3][4] Steven Minton and Andy Philips analyzed the neural network algorithm and separated it into two phases: (1) an initial assignment using a greedy algorithm and (2) a conflict minimization phases (later to be called "min-conflicts"). A paper was written and presented at AAAI-90; Philip Laird provided the mathematical analysis of the algorithm.

Subsequently, Mark Johnston and the STScI staff used min-conflicts to schedule astronomers' observation time on the Hubble Space Telescope.

Example [edit]

Min-Conflicts solves the *N*-Queens Problem by randomly selecting a column from the chess board for queen reassignment. The algorithm searches each potential move for the number of conflicts (number of attacking queens), shown in each square. The algorithm moves the queen to the square with the minimum number of conflicts, breaking ties randomly. Note that the number of conflicts is generated by each new direction that a queen can attack from. If two queens would




0
0
0
0
0
0
0
0

Again … does the nqueens get easier as n increases?

Strategy	$n = 10^1$	$n = 10^2$	$n = 10^3$	$n = 10^4$	$n = 10^5$	$n = 10^6$
Basic Backtrack†	53.8	4473 (70%)	88650 (13%)	*	*	*
Most Constrained Backtrack†	17.4	687 (96%)	22150 (81%)	*	*	*
MinConflicts Hill-Climbing‡	57.0	55.6	48.8	48.5	52.8	48.3
MinConflicts Backtrack‡	46.8	25.0	30.7	27.5	27.8	26.4

† = number of backtracks, ‡ = number of repairs
* = exceeded computational resources (100 runs required > 12 hours on a SPARCstation1)

Table 1: Number of Backtracks/Repairs for N-Queens Algorithms

be instantiated in linear time to yield a solution[1]. Nevertheless, the problem has remained relatively "hard" for heuristic search methods. Several studies of the n-queens problem [21,8,14] have compared heuristic backtracking methods such as search rearrangement backtracking (e.g., most-constrained first), forward checking, dependency-directed backtracking, etc. However, no previously identified heuristic search method has been able to consistently solve problems involving hundreds of queens within a reasonable time limit.

On the n-queens problem, Adorf and Johnston [2] reported that the probability of the GDS network converging increases with the size of the problem. For

better than the network because the hill-climbing program's preprocessing phase invariably produces an initial assignment that is "close" to a solution, in that the number of conflicting queens in the initial assignment grows extremely slowly (from a mean of 3.1 for $n = 10$ to a mean of 12.8 for $n = 10^6$). Once this difference was eliminated, by starting the network in an initial state produced by our preprocessing algorithm, the network and the hill-climbing program performed quite similarly. We note, however, that the network requires $O(n^2)$ space, as compared to the $O(n)$ space required by the hill-climbing program, which prevented us from running very large problems on the network.

Table 1 compares the efficiency of our hill-climbing stop