Constraint Processing

(Version of 27 September 2004)

Constraint Satisfaction Problems (CSPs)

Variables: $X_1, X_2, \ldots, X_n$

Domains of the variables: $D_1, D_2, \ldots, D_n$

Constraints on the variables:
examples: $X_1 \neq X_3$
$3 \cdot X_1 + 4 \cdot X_2 \leq X_4$

What is a solution?

- An *assignment* to each variable of a value from its domain,
- ...such that all the constraints are satisfied.

Objective

- Find *a* solution.
- Find *all* the solutions.
- Find an *optimal* solution,
  according to some *cost expression* on the variables.

Applications

- Scheduling
- Planning
- Design
- Transport
- Logistics
- Molecular Biology
- Games
- Puzzles
- ...

Solving Methods

- *Ad hoc* programs
- Search programs
- Artificial intelligence techniques
- Mathematical programming
- Constraint programming

Complexity

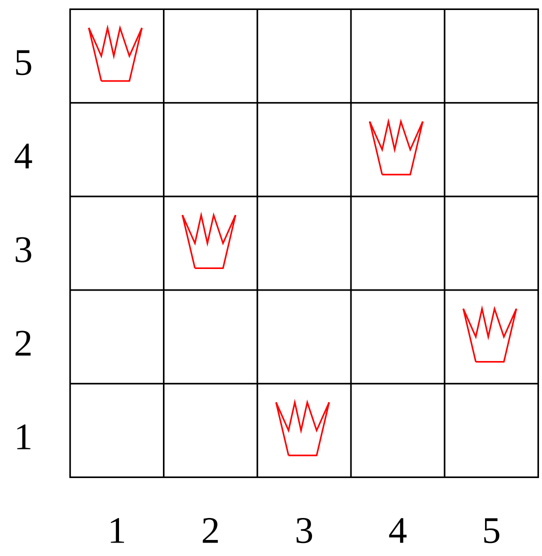
- Generally the problems are NP-complete ...
- ...with exponential complexity

Example: The n-Queens Problem

The Problem

How to place n queens on an $n \times n$ chessboard such that no queen is threatened?

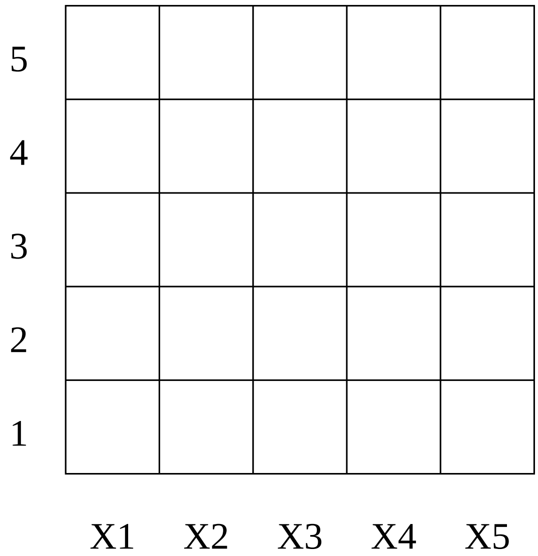
A Solution for n=5



Number of candidate solutions: $\binom{n^2}{n}$

Can we do better than that?!

The n-Queens Problem as a CSP



Variables: $X_1, X_2, \ldots, X_n$ (one variable for each column)

Domains of the variables: $D_i = \{1, 2, \ldots, n\}$ (the rows)

Constraints on the variables:

- No two queens are in the same column:
  this is impossible by the choice of the variables!
- No two queens are in the same row:

$$X_i \neq X_j, \text{ for each } i \neq j$$

- No two queens are in the same diagonal:

$$\mid X_i - X_j \mid \neq \mid i - j \mid, \text{ for each } i \neq j$$

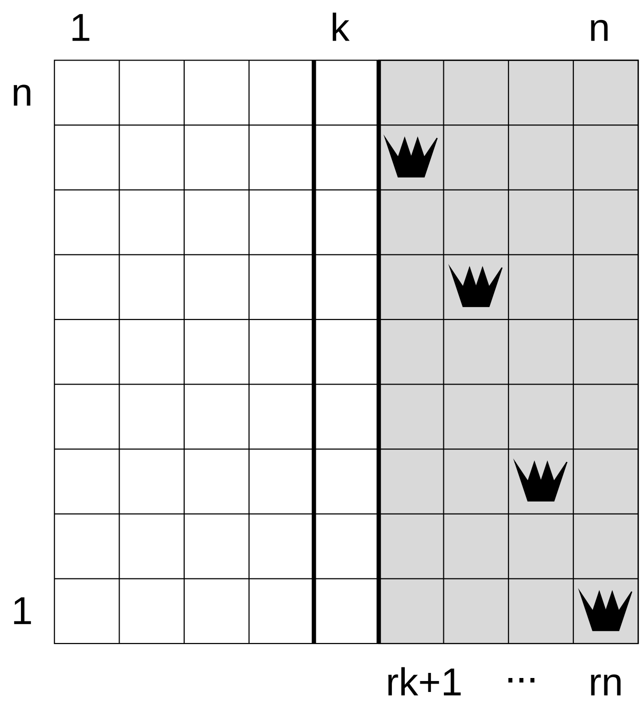
Number of candidate solutions: n^n

Can we do better than that?!

First Approach: Exhaustive Enumeration

- *Generation* of possible values of the variables.
- *Test* of the constraints.

Strategy

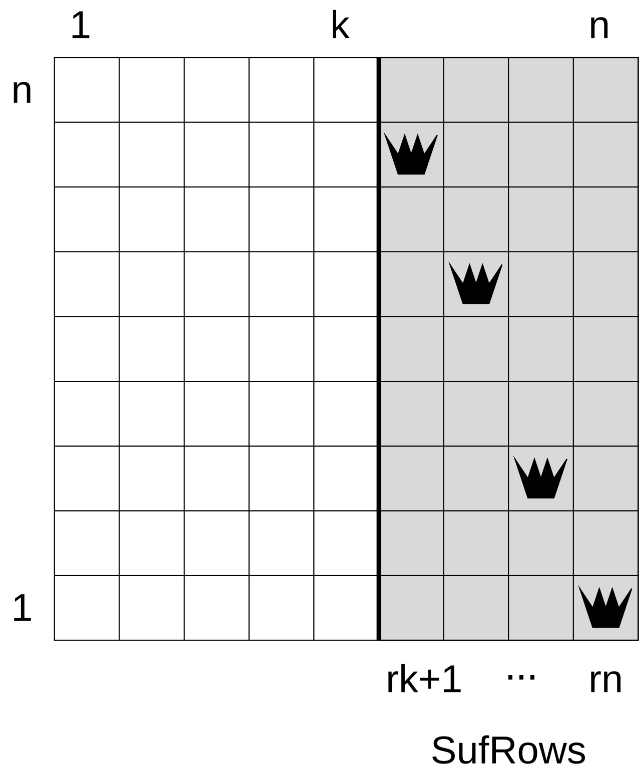


where $r_{k+1}, \ldots, r_n$ are the rows for the queens in the columns $k+1, \ldots, n$ (the "already filled" part).

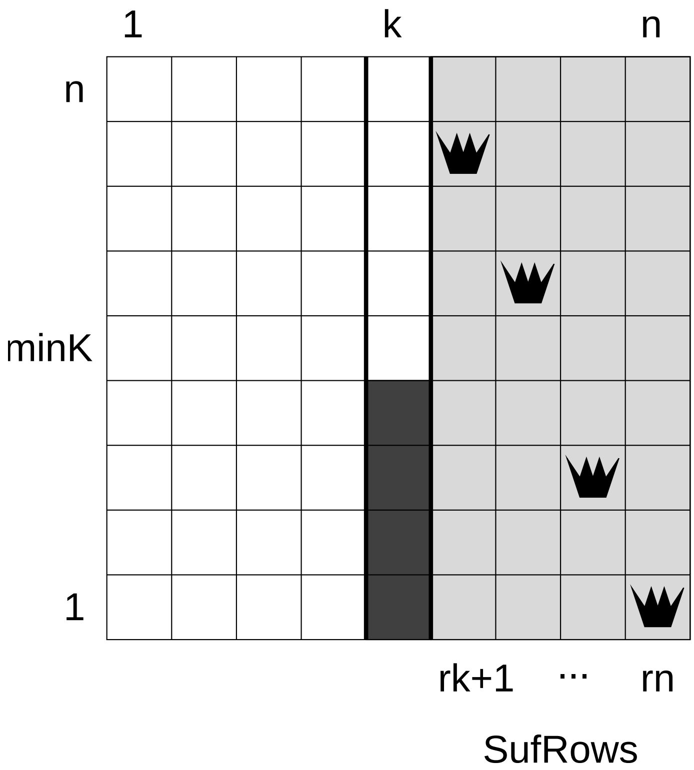
Question: Where to place a queen in column k such that it is compatible with $r_{k+1}, \ldots, r_n$?

Specifications

function placeQueens n : int $\longrightarrow$ unit
PRE: n > 0
POST: true
SIDE-EFFECTS: display of a solution to the n-queens problem, if one exists;
otherwise, display of a message saying there is no solution.

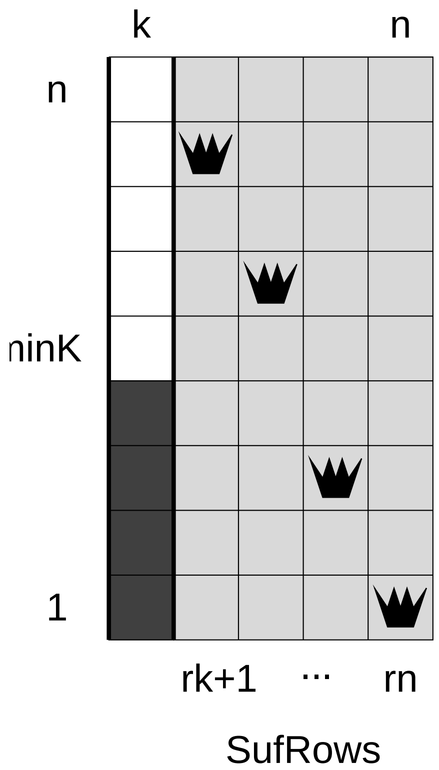


function queens n k SufRows : int $\longrightarrow$ int $\longrightarrow$ int list $\longrightarrow$ (int list $*$ bool)
PRE: $0 \leq$ k $\leq$ n > 0;
SufRows has rows of the queens in the columns k+1, . . . , n.
POST: (Rows, success), with Rows = PreRows @ SufRows,
where PreRows has rows of the queens in the columns 1, . . . , k
that are mutually compatible as well as compatible with SufRows;
if such rows exist, then success is true;
otherwise, success is false, and Rows is undetermined.



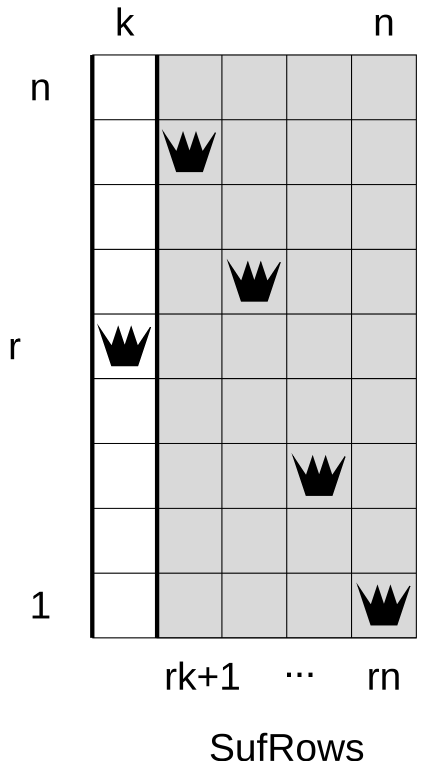
function qAux n k minK SufRows : int $\longrightarrow$ int $\longrightarrow$ int $\longrightarrow$ int list $\longrightarrow$ (int list $*$ bool)

Same as for queens, but the queen in column k must be in a row $\geq$ minK.



function newQueen n minK SufRows : int $\longrightarrow$ int $\longrightarrow$ int list $\longrightarrow$ (int $*$ bool)

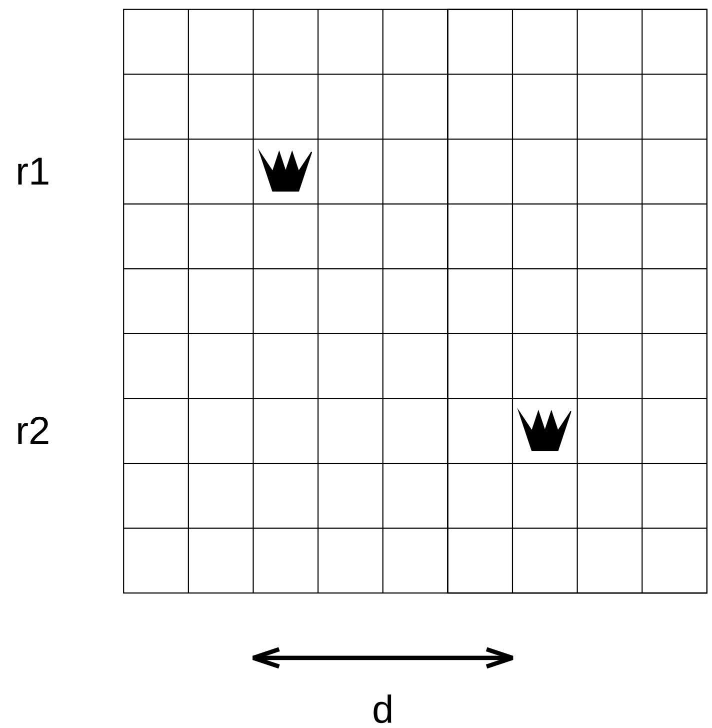
Same as for qAux, but placement of a single queen in front of SufRows.



function compK r SufRows : int $\longrightarrow$ int list $\longrightarrow$ bool

PRE: SufRows has rows of the queens in the columns k+1, . . . , n.

POST: **true** iff a queen in row r and column k is compatible with SufRows.



function compatible r1 r2 d : int $\longrightarrow$ int $\longrightarrow$ int $\longrightarrow$ bool

PRE: r1, r2, d > 0

POST: **true** iff queens in rows r1 and r2, but d columns apart, are compatible.

SML Program (`queens.sml`)

```
fun compatible r1 r2 d = r1 <> r2 andalso abs(r1−r2) <> d

fun compK r SufRows =
   let fun compKaux r d [ ] = true
         | compKaux r d (h::t) =
             (compatible r h d) andalso (compKaux r (d+1) t)
   in compKaux r 1 SufRows
   end

fun newQueen n minK SufRows =
   if minK > n then (0,false)
   else if compK minK SufRows then (minK,true)
        else newQueen n (minK+1) SufRows

fun queens n k SufRows =
   let fun qAux n k minK SufRows =
          if minK > n then ([ ],false)
          else
             let val (rowK,success) = newQueen n minK SufRows
             in if not success then ([ ],false)
                else
                let val (Rows,hurray) = queens n (k−1) (rowK::SufRows)
                in if hurray then (Rows,true)
                   else qAux n k (rowK+1) SufRows
                end
             end
   in if k=0 then (SufRows,true)
      else qAux n k 1 SufRows
   end
```

```
fun printList [ ] = print "\n"
 |  printList (x::xs) = (print (Int.toString x) ; print " " ; printList xs)

fun placeQueens n =
    let val (Rows,success) = queens n n [ ]
    in if success then (print "Solution: " ; printList Rows)
        else print "No solutions... \n"
    end
```

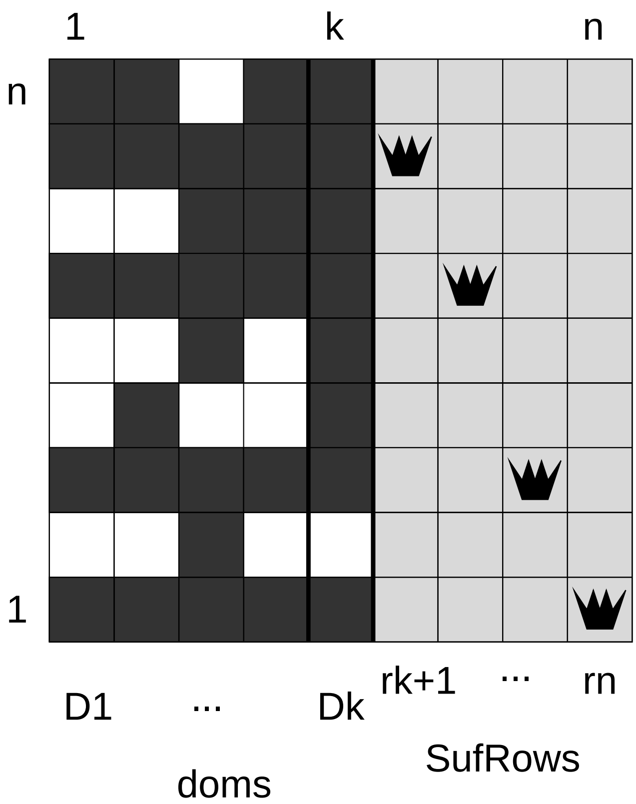
Analysis

- Exploration of *many* possibilities.
- Very *late* detection of deadends.
- Exponential complexity.

Second Approach: Domain Reduction

Strategy

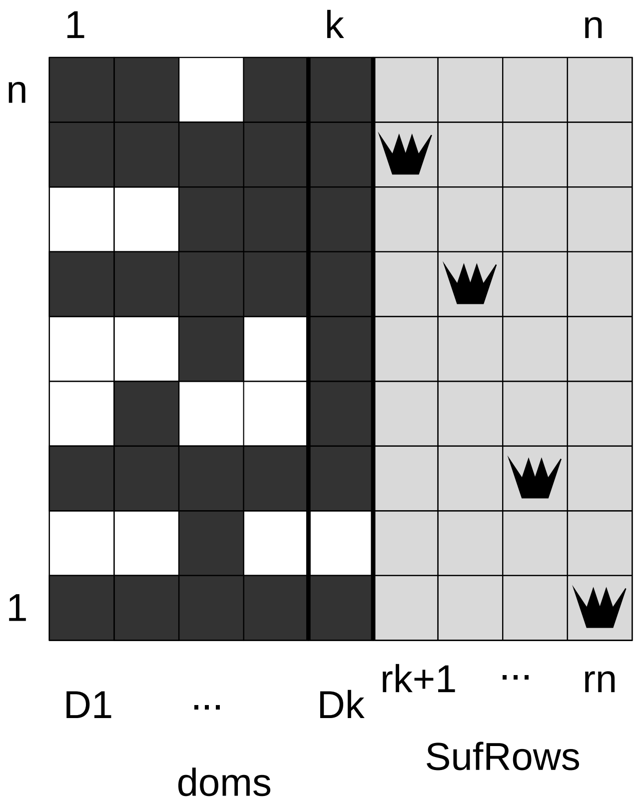
Maintain for each variable X_i the domain D_i containing the values (row numbers) that are still possible for the queen in column i.



Search effort:
for instance, for $n = 10$:
only $4,066$ ($\ll 10^{10}$) backtracks to find all the 724 solutions!

Can we do better than that?!
Yes, by exploiting the symmetries of the chessboard!

Specification



function qDom k SufRows Doms : int ⟶ int list ⟶ int list list ⟶ (int list ∗ bool)
PRE: SufRows has rows of the queens in the columns k+1, . . . , n ;
Doms = [D_k, . . . , D_1], where D_i has the row numbers
that are compatible with SufRows for a queen in column i.
POST: (Rows, success), with Rows = PreRows @ SufRows,
where PreRows has rows from Doms of the queens in the columns 1, . . . , k
that are mutually compatible as well as compatible with SufRows;
if such rows exist, then success is true;
otherwise, success is false, and Rows is undetermined.

SML Program

- Extension of the auxiliary problems of the first approach.
- Integration of the domains: see the (on-line) code.