# Métodos de Apoio à Decisão

## Programação por restrições

João Pedro Pedroso

2023/2024

Última aula:

- Introdução à programação por restrições

Hoje:

- Programação por restrições: exemplos

# Programação por restrições

Componentes fundamentais:

- **variáveis**
- **domínios**
- **restrições**

# Programação por restrições

Restrições:

- Equações ou desigualdades (como as que vimos em otimização inteira)
- Restrições lógicas
- Restrições globais:
  - permitem especificar com um termo uma propriedade que determinadas variáveis devem possuir
  - muitas vezes, difíceis de exprimir com equações ou desigualdades
  - têm associados *solvers* específicos
  - exemplo: `AllDifferent(x1,x2,x3,x4,x5)` $\rightarrow$ as variáveis `x1,x2,x3,x4,x5` deverão todas ter valores diferentes
    - cada $x_i$ deverá ter um domínio finito
    - *e.g.*, $x_i \in \{1, \ldots, 10\}$
    - sintaxe de `AllDifferent` depende do resolutor (iremos ver exemplo com AMPL)

# AMPL + Programação por restrições

- Sintaxe para programação por restrições em AMPL:
  - Operadores lógicos:
    - `and + or + not`
    - iterados: `exists + forall`
  - Operadores conditionais:
    - `if-then + if-then-else + ==> + ==> else + <== + <==>`
  - Operadores de contagem:
    - iterados: `count + atmost + atleast + exactly + numberof`
    - coletivos: `alldiff`
- Alguns resolutores
  - IBM ILOG CP (comercial; disponível no software fornecido `option solver ilogcp;`)
  - Gecode (software livre, disponível aqui)
  - JaCoP (software livre, disponível aqui)

Nota: O sistema GLPK não suporta programação por restrições

# AMPL + Programação por restrições

Referências utilizadas:

- https://ampl.com/products/ampl/logic-and-constraint-programming-extensions/
- https://ampl.com/MEETINGS/TALKS/2012_10_Phoenix_SA15.pdf

# Logical Operators

- Unary and binary
  - *constraint-expr* `and` *constraint-expr*
  - *constraint-expr* `or` *constraint-expr*
  - `not` constraint-expr
- Iterated forms
  - `exists` { indexing } *constraint-expr*
  - `forall` { indexing } *constraint-expr*
- Implication expressions
  - `if` *constraint-expr* `then` *expr*
  - `if` *constraint-expr* `then` *expr* `else` *expr*
- Implication constraints
  - *constraint-expr* ==> *constraint-expr*
  - *constraint-expr* ==> *constraint-expr* `else` *constraint-expr*
  - *constraint-expr* <== *constraint-expr*
  - *constraint-expr* <== *constraint-expr* `else` *constraint-expr*
  - *constraint-expr* ==> *constraint-expr*
  - *constraint-expr* <==> *constraint-expr*

# Counting Operators

- Counting expressions
  - `count` { *indexing* } ( *constraint* )
  - `numberof` *num-expr* `in` ( *expr1*, *expr2*, . . . )
- Counting constraints
  - `atmost` *num-expr* { *indexing* } ( *constraint* )
  - `atleast` *num-expr* { *indexing* } ( *constraint* )
  - `exactly` *num-expr* { *indexing* } ( *constraint* )

# Global Constraints

- All-different constraint
  - `alldiff` { *indexing* } *expr*
  - `alldiff` ( *expr1*, *expr2*, . . . )
- Counting expression
  - `numberof` *const-expr* `in` ( *expr1*, *expr2*, . . . )
    - → consolidate all having same *expr-list*
- Form of expr1, expr2, . . . in list
  - *expr*
  - { *indexing* } *expr*

# Solving Sudoku

From the Wikipedia:

- Sudoku → 数独 / digit-single, originally called *Number Place*
  - French newspapers featured variations of the Sudoku puzzles in the 19th century
- Logic-based combinatorial number-placement puzzle
- Objective: fill a 9×9 grid with digits so that all of the digits from 1 to 9 are in
  - each column
  - each row
  - each of the nine 3×3 subgrids that compose the grid
    - also called "boxes", "blocks", or "regions"
- Input: partially completed grid which for a well-posed puzzle has a single solution.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | 3 | | | 7 | | | | |
| 6 | | | 1 | 9 | 5 | | | |
| | 9 | 8 | | | | | 6 | |
| 8 | | | | 6 | | | | 3 |
| 4 | | | 8 | | 3 | | | 1 |
| 7 | | | | 2 | | | | 6 |
| | 6 | | | | | 2 | 8 | |
| | | | 4 | 1 | 9 | | | 5 |
| | | | | 8 | | | 7 | 9 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | 3 | | | 7 | | | | |
| 6 | | | 1 | 9 | 5 | | | |
| | 9 | 8 | | | | | 6 | |
| 8 | | | | 6 | | | | 3 |
| 4 | | | 8 | | 3 | | | 1 |
| 7 | | | | 2 | | | | 6 |
| | 6 | | | | | 2 | 8 | |
| | | | 4 | 1 | 9 | | | 5 |
| | | | | 8 | | | 7 | 9 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | 3 | | | 7 | | | | |
| 6 | | | 1 | 9 | 5 | | | |
| | 9 | 8 | | | | | 6 | |
| 8 | | | | 6 | | | | 3 |
| 4 | | | 8 | | 3 | | | 1 |
| 7 | | | | 2 | | | | 6 |
| | 6 | | | | | 2 | 8 | 4 |
| | | | 4 | 1 | 9 | | | 5 |
| | | | | 8 | | | 7 | 9 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | 3 | | | 7 | | | | |
| 6 | | | 1 | 9 | 5 | | | |
| | 9 | 8 | | | | | 6 | |
| 8 | | | | 6 | | | | 3 |
| 4 | | | 8 | | 3 | | | 1 |
| 7 | | | | 2 | | | | 6 |
| | 6 | | | | | 2 | 8 | |
| | | | 4 | 1 | 9 | | | 5 |
| | | | | 8 | | | 7 | 9 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | 3 | 4 | 6 | 7 | 8 | 9 | 1 | 2 |
| 6 | 7 | 2 | 1 | 9 | 5 | 3 | 4 | 8 |
| 1 | 9 | 8 | 3 | 4 | 2 | 5 | 6 | 7 |
| 8 | 5 | 9 | 7 | 6 | 1 | 4 | 2 | 3 |
| 4 | 2 | 6 | 8 | 5 | 3 | 7 | 9 | 1 |
| 7 | 1 | 3 | 9 | 2 | 4 | 8 | 5 | 6 |
| 9 | 6 | 1 | 5 | 3 | 7 | 2 | 8 | 4 |
| 2 | 8 | 7 | 4 | 1 | 9 | 6 | 3 | 5 |
| 3 | 4 | 5 | 2 | 8 | 6 | 1 | 7 | 9 |

# Sukodu: modelo em otimização inteira

# Sukodu: modelo em otimização inteira

```
param givens{1..9, 1..9}, integer, >= 0, <= 9, default 0;
/* the "givens" */

var x{i in 1..9, j in 1..9, k in 1..9}, binary;
/* x[i,j,k] = 1 means cell [i,j] is assigned number k */

s.t. fa{i in 1..9, j in 1..9, k in 1..9: givens[i,j] != 0}:
     x[i,j,k] = (if givens[i,j] = k then 1 else 0);
/* assign pre-defined numbers using the "givens" */

s.t. fb{i in 1..9, j in 1..9}: sum{k in 1..9} x[i,j,k] = 1;
/* each cell must be assigned exactly one number */

s.t. fc{i in 1..9, k in 1..9}: sum{j in 1..9} x[i,j,k] = 1;
/* cells in the same row must be assigned distinct numbers */

s.t. fd{j in 1..9, k in 1..9}: sum{i in 1..9} x[i,j,k] = 1;
/* cells in the same column must be assigned distinct numbers */

s.t. fe{I in 1..9 by 3, J in 1..9 by 3, k in 1..9}:
     sum{i in I..I+2, j in J..J+2} x[i,j,k] = 1;
/* cells in the same region must be assigned distinct numbers */
```

# Sukodu: modelo em otimização inteira

```
solve;     /* there is no need for an objective function here */
for {i in 1..9}
{  for {0..0: i = 1 or i = 4 or i = 7}
      printf " +-------+-------+-------+\n";
   for {j in 1..9}
   {  for {0..0: j = 1 or j = 4 or j = 7} printf(" |");
      printf " %d", sum{k in 1..9} x[i,j,k] * k;
      for {0..0: j = 9} printf(" |\n");
   }
   for {0..0: i = 9}
      printf " +-------+-------+-------+\n";
}
data;      /* These data correspond to the example above. */
param givens : 1 2 3 4 5 6 7 8 9 :=
            1   5 3 . . 7 . . . .
            2   6 . . 1 9 5 . . .
            3   . 9 8 . . . . 6 .
            4   8 . . . 6 . . . 3
            5   4 . . 8 . 3 . . 1
            6   7 . . . 2 . . . 6
            7   . 6 . . . . . 2 8 .
            8   . . . 4 1 9 . . 5
            9   . . . . 8 . . 7 9 ;
```

# Sudoku: modelo em programação por restrições

# Sudoku: modelo em programação por restrições

```
param given {1..9, 1..9} integer, in 0..9;
        # given[i,j] > 0 is the value given for row i, col j
        # given[i,j] = 0 means no value given

var X {1..9, 1..9} integer, in 1..9;
        # x[i,j] = the number assigned to the cell in row i, col j

subj to AssignGiven {i in 1..9, j in 1..9: given[i,j] > 0}:
   X[i,j] = given[i,j];
        # assign given values

subj to Rows {i in 1..9}:
   alldiff {j in 1..9} X[i,j];
        # cells in the same row must be assigned distinct numbers

subj to Cols {j in 1..9}:
   alldiff {i in 1..9} X[i,j];
        # cells in the same column must be assigned distinct numbers

subj to Regions {I in 1..9 by 3, J in 1..9 by 3}:
   alldiff {i in I..I+2, j in J..J+2} X[i,j];
        # cells in the same region must be assigned distinct numbers
```

# Sudoku: propagação "manual"

# Sudoku: propagação "manual"

```
puzzle = [[5,3,0,0,7,0,0,0,0],
          [6,0,0,1,9,5,0,0,0],
          [0,9,8,0,0,0,0,6,0],
          [8,0,0,0,6,0,0,0,3],
          [4,0,0,8,0,3,0,0,1],
          [7,0,0,0,2,0,0,0,6],
          [0,6,0,0,0,0,2,8,0],
          [0,0,0,4,1,9,0,0,5],
          [0,0,0,0,8,0,0,7,9]]

n = len(puzzle)
L = set(range(n))
missing = [[L.copy() for i in range(n)] for j in range(n)]

sol = puzzle
dim = len(sol)
sqr = int(math.sqrt(dim))
assert sqr == math.sqrt(dim)
```

# Sudoku: propagação "manual"

```
for i in range(dim):
    for j in range(dim):
        if sol[i][j] != 0:
            missing[i][j] = None
            continue
        for k in L:
            if k in (sol[a][j] for a in range(dim)):      # check rows
                missing[i][j].discard(k)
                continue
            if k in (sol[i][a] for a in range(dim)):      # check columns
                missing[i][j].discard(k)
                continue
            # check small squares
            I = i // sqr
            J = j // sqr
            p = set()
            for i_ in range(sqr):
                for j_ in range(sqr):
                    p.add(sol[I*sqr+i_][J*sqr+j_])
            if k in p:
                missing[i][j].discard(k)
                continue
```

# Sudoku: propagação "manual"

- Depois de executar, observando `missing`

```
[None, None, {1, 2, 4}, {2, 6}, None, {2, 4, 6, 8}, {1, 4, 8, 9}, {1, 2, 4, 9}, {2,
[None, {2, 4, 7}, {2, 4, 7}, None, None, None, {3, 4, 7, 8}, {2, 3, 4}, {2, 4, 7, 8
[{1, 2}, None, None, {2, 3}, {3, 4}, {2, 4}, {1, 3, 4, 5, 7}, None, {2, 4, 7}]
[None, {1, 2, 5}, {1, 2, 5, 9}, {5, 7, 9}, None, {1, 4, 7}, {4, 5, 7, 9}, {2, 4, 5,
[None, {2, 5}, {2, 5, 6, 9}, None, {5}, None, {5, 7, 9}, {2, 5, 9}, None]
[None, {1, 5}, {1, 3, 5, 9}, {5, 9}, None, {1, 4}, {4, 5, 8, 9}, {4, 5, 9}, None]
[{1, 3, 9}, None, {1, 3, 4, 5, 7, 9}, {3, 5, 7}, {3, 5}, {7}, None, None, {4}]
[{2, 3}, {2, 7, 8}, {2, 3, 7}, None, None, None, {3, 6}, {3}, None]
[{1, 2, 3}, {1, 2, 4, 5}, {1, 2, 3, 4, 5}, {2, 3, 5, 6}, None, {2, 6}, {1, 3, 4, 6}
```

- Passos seguintes:
  - escolher uma posição com apenas *uma escolha*
  - repetir a propagação, até completar o quadro
- Em programação por restrições:
  - em geral, não há uma variável a fixar "sem ambiguidade"
  - tem de se escolher uma, "tentar" um valor possível e propagar
  - se não resultar: backtracking → escolher outro valor possível

# Propagação

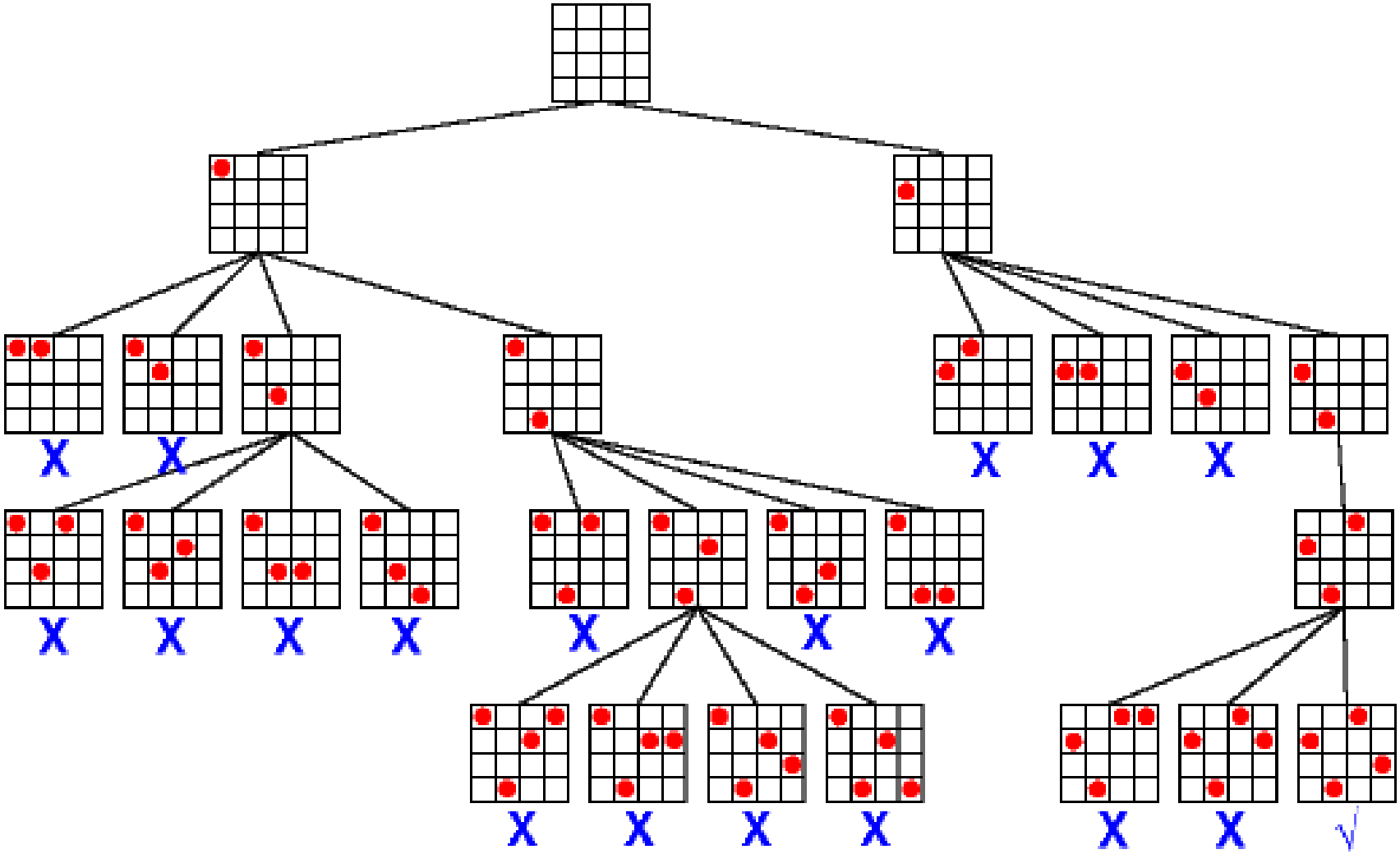
Sem alterar soluções do problema:

- reduzir domínio de certas variáveis
- "deduzir":
  - novos limites mais fortes para certas restrições
  - novas restrições
  - exemplo:
    $6x_1 + 3x_2 + 5x_3 + 2x_4 \leq 10, \quad x_i \in \{0, 1\}$
    $\Rightarrow x_1 + x_2 + x_4 \leq 2$

# *Backtracking* (retrocesso)

- algoritmo geral para encontrar soluções que satisfazem um conjunto de restrições
- instancia incrementalmente soluções candidatas
  - enumeração recursiva, variável a variável
- quando verifica que a candidata não pode ser completada: backtracking
  - retrocesso para o último valor "tentado"
  - tentativa do valor seguinte do domínio dessa variável
    - se não existir: retroceder para a variável anterior
    - ...
- corresponde a atravessar uma árvore de enumeração em profundidade
  - filhos de nós inviáveis não são enumerados

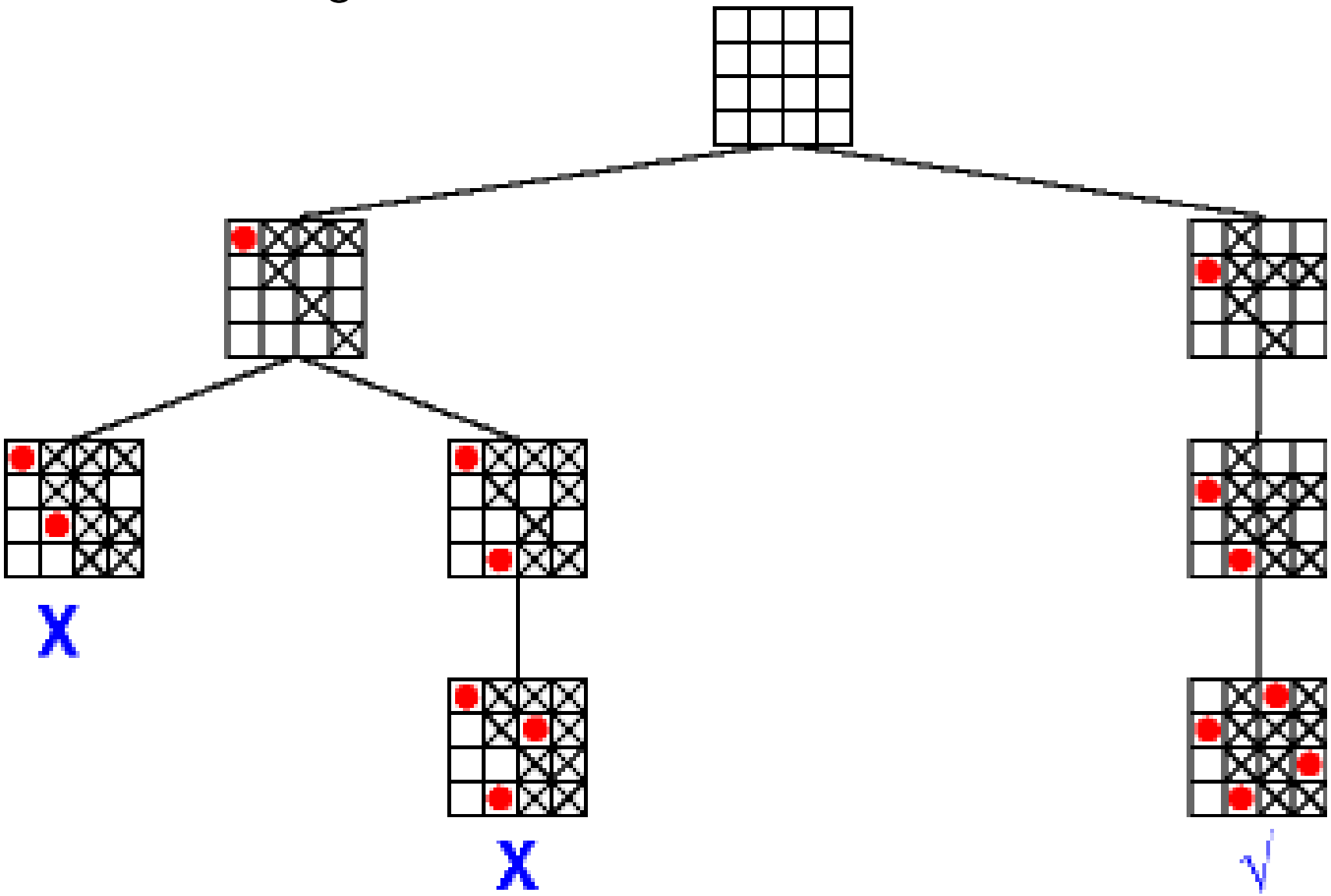
# Backtracking



source: "Guide to Constraint Programming"
https://ktiml.mff.cuni.cz/~bartak/constraints/index.html

# Backtracking + Propagation

## Forward Checking



# Próxima aula

- Programação por restrições: exemplos