Introduction to Artificial Intelligence

Roman Barták

Department of Theoretical Computer Science and Mathematical Logic

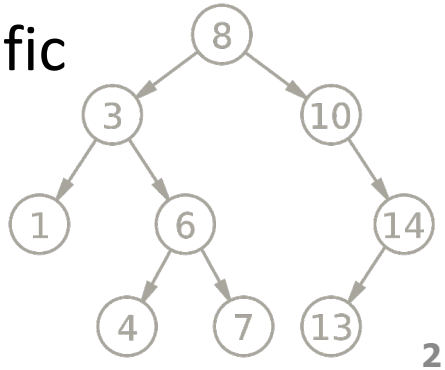
Problem solving agent is a type of goal-based agent

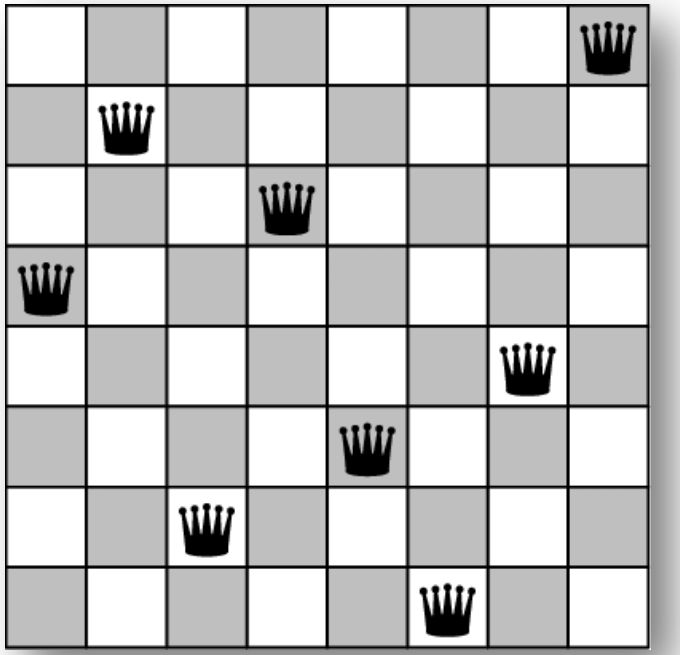
- uses **atomic representation** of states
- goal is represented by a set of **goal states**
- actions describe **transitions** between states

The task is to find a **sequence of actions** that reaches the goal state (from the initial/current) state.

Problem solving is realized via **search**:

- **tree search** vs **graph search**
- **uniformed search** (no additional information beyond problem formulation)
- **informed** (heuristic) **search** (uses problem-specific knowledge)
  - **algorithm A*:** f(n) = g(n) + h(n)





find locations of N queens on board of size N×N such that the queens do not conflict with each other

conflicts:

- same row
- same column
- same diagonal

How to model the problem?

- What is the **goal**?
- What are the **states**?
- What are the **actions**?

States = locations of queens on board

Initial state = empty board

Goal state = unknown state

but easy to recognize: N queens are on board and no conflict among them

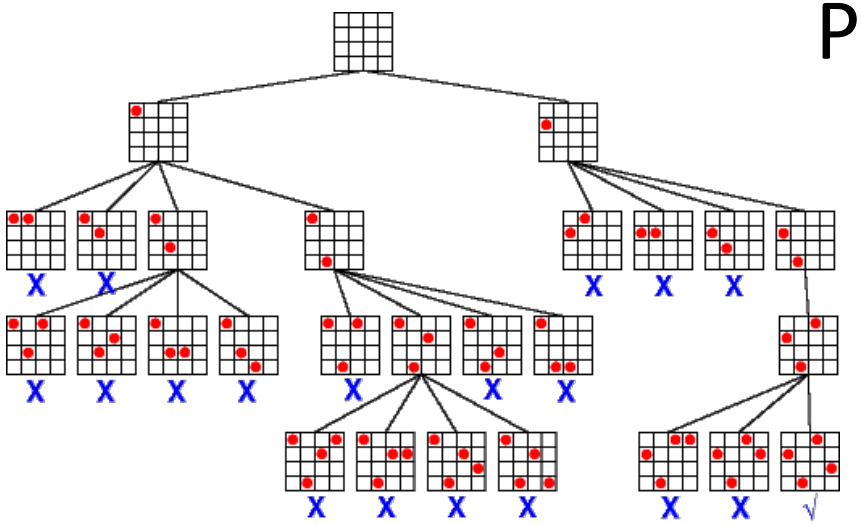
Action = put a queen to a board (such that the queen does not conflict with already placed queens)

Better model:

queens are pre-allocated to columns and we are looking for rows only (smaller search space: N^N vs. $(N \times N)^N$)

Alternative model:

all queens are on board and we can just change their positions (local search)

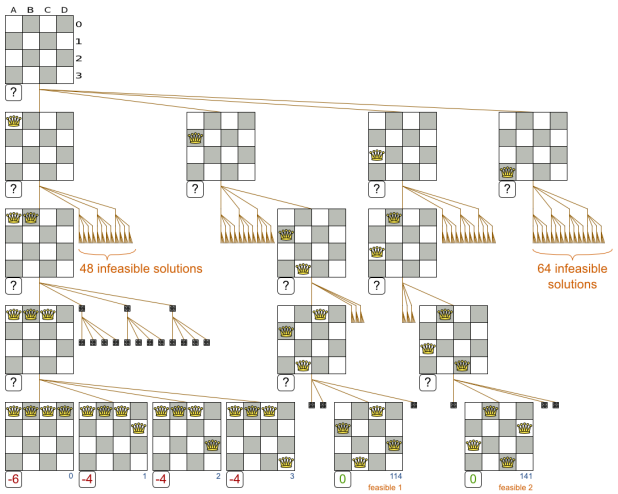


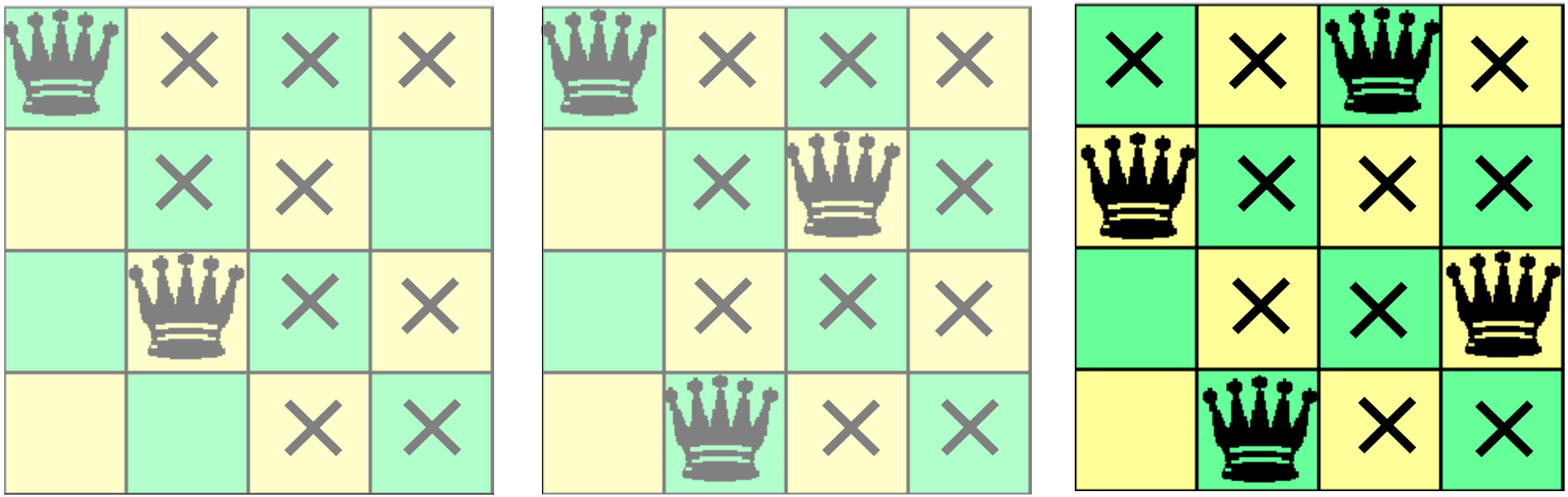
Properties:

- we know the depth where the solution lies (N)
- each branch leads to a different set of states (search nodes contain different states)

Hence, **tree search** with **depth search strategy** (backtracking) is appropriate there.

Can we do better?





Each time we assign a queen, we remove all conflicting positions for not-yet assigned queens.

This technique is called **forward checking**.

How to implement this technique for N-queens and for other problems?

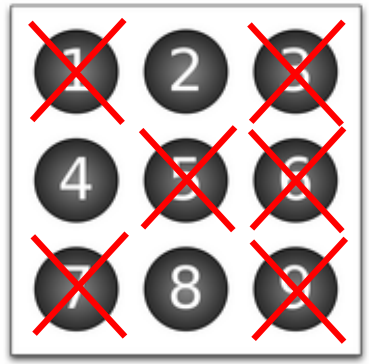
9	6	3	1	7	4	2	5	8
1	7	8	3	2	5	6	4	9
2	5	4	6	8	9	7	3	1
8	2	1	4	3	7	5	9	6
4	9	6	8	5	2	3	1	7
7	3	5	9	6	1	8	2	4
5	8	9	7	1	3	4	6	2
3	1	7	2	4	6	9	8	5
6	4	2	5	9	8	1	7	3

Logic-based puzzle, whose goal is to enter digits 1-9 in cells of 9×9 table in such a way, that no digit appears twice or more in every row, column, and 3×3 sub-grid.

Each cell is a variable with possible values from domain {1,…,9}.

Cells in rows, columns, and sub-grids should contain different values.

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



We can formulate N-queens, Sudoku and other problems using a common formalism with factored state representation.

Constraint satisfaction problem consists of:

- a finite set of **variables**
  - describe some features of the world state that we are looking for, for example position of queens at a chessboard
- **domains** – finite sets of values for each variable
  - describe "options" that are available, for example the rows for queens
- a finite set of **constraints**
  - a constraint is a **relation** over a subset of variables; constraint can be defined in extension (a set of tuples satisfying the constraint) or using a formula (rowA ≠ rowB)
  - constraint arity = the number of constrained variables

A **feasible solution** of a constraint satisfaction problem is a complete consistent assignment of values to variables.

complete = each variable has assigned a value

consistent = all constraints are satisfied

First, one needs to formulate the problem as a constraint satisfaction problem.

This is called **constraint modeling**.

Example (N-queens problem):

the core decision: each queen is pre-allocated to its own column and we are looking for its row

variables: N variables r(i) with the domain {1,…,N}

constraints: no two queens attack each other

$$\forall i \neq j \quad r(i) \neq r(j) \wedge |i-j| \neq |r(i)-r(j)|$$

Backtracking search:

- assign a value to a selected (not-yet instantiated) variable
- check constraints over already instantiated variables
- if the constraints are satisfied then continue to the next variable otherwise try a different value
- if no value can be assigned to a variable then go back to the previous variable and try an alternative value for that variable
- repeat until all variables are instantiated (and all constraints satisfied)

```
function BACKTRACKING-SEARCH( csp) returns a solution, or failure
    return RECURSIVE-BACKTRACKING({}, csp)

function RECURSIVE-BACKTRACKING( assignment,csp) returns a solution, or
failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE( Variables[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment according to Constraints[csp] then
            add { var = value } to assignment
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failue then return result
            remove { var = value } from assignment
    return failure
```

Can we use the constraints in a more active way, for example to prune inconsistent values ?

Example:

A in 3..7, B in 1..5 the variables' domains

A<B the constraint

– many inconsistent values can be removed

– we get A in 3..4, B in 4..5

Note: it does not mean that all the remaining combinations of the values are consistent (for example A=4, B=4 is not consistent)

How to remove the inconsistent values from the variables' domains in the constraint network?

For simplicity we will assume binary CSPs only

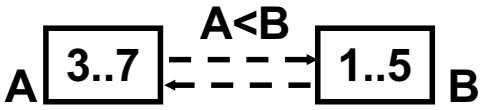
i.e. each constraint corresponds to an arc (edge) in the **constraint network**.

The arc (V_i,V_j) is **arc consistent** iff for each value x from the domain D_i there exists a value y in the domain D_j such that the assignment V_i =x a V_j = y satisfies all the binary constraints on V_i, V_j.

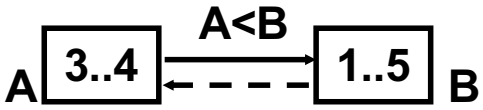
Note: The concept of arc consistency is directional, i.e., arc consistency of (V_i,V_j) does not guarantee consistency of (V_j,V_i).

CSP is **arc consistent** iff every arc (V_i,V_j) is arc consistent (in both directions).

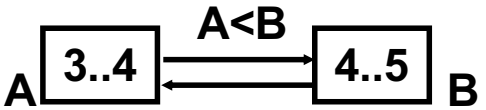
Example:



no arc is consistent



(A,B) is consistent



(A,B) and (B,A) are consistent

Algorithm AC-3

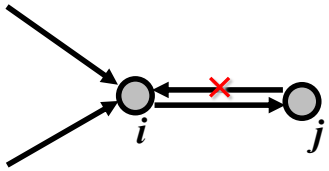
```
function AC-3( csp) returns the CSP, possibly with reduced domains
  inputs: csp, a binary CSP with variables {X_1, X_2, ..., X_n}
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (X_i, X_j) ← REMOVE-FIRST(queue)
    if RM-INCONSISTENT-VALUES(X_i, X_j) then
      for each X_k in NEIGHBORS[X_i] do
        add (X_k, X_i) to queue
```

The algorithm can be applied incrementally during search – when X is instantiated put all constraints related to X to the queue.

If the domain of variable X_i changed then verify all arcs (constraints) leading to the variable except the arc from the variable X_j.

```
function RM-INCONSISTENT-VALUES( X_i, X_j) returns true iff remove a value
  removed ← false
  for each x in DOMAIN[X_i] do
    if no value y in DOMAIN[X_j] allows (x,y) to satisfy constraint(X_i, X_j)
      then delete x from DOMAIN[X_i]; removed ← true
  return removed
```



Domain filtering for variable X_i removes values that have no support in the variable X_j, also, if any value is deleted this information is passed to the calling procedure. Knowing constraint semantics can speedup constraint checking (for example X<Y).

Time complexity of AC-3 is $O(ed^3)$, where e is the number of constraints and d is the size of domain – we need to repeatedly (ed) check the constraints (d^2). This is not optimal, we can remember the result of consistency checks - AC-4, AC-3.1, AC-2001 with time complexity $O(ed^2)$.

How to integrate arc consistency with backtracking search?

- make the problem arc consistent.
- after each assignment (during search) arc consistency is restored (by removing inconsistent values)

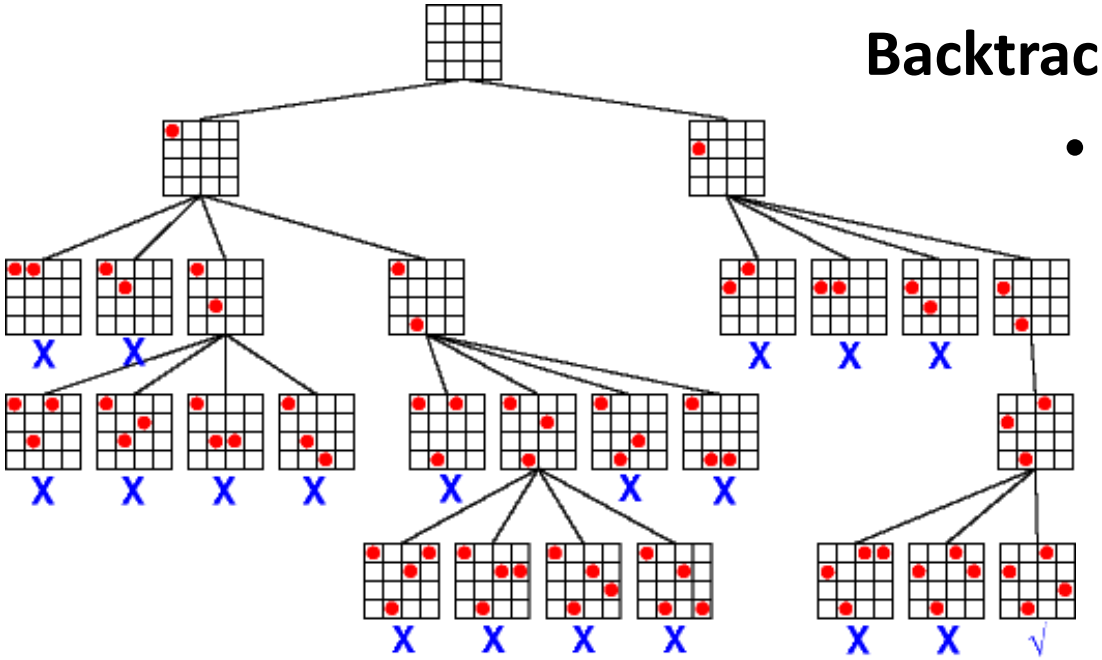
This technique is known as **look ahead** or **constraint propagation** or **maintaining arc consistency**.

What is the difference from forward checking?

- FC only checks constraints containing currently instantiate variable
- LA checks all constraints (and hence removes more inconsistencies)

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return BACKTRACK({}, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment then
      add {var = value} to assignment
      inferences ← INFERENCE(csp, var, value)
      if inferences ≠ failure then
        add inferences to assignment
        result ← BACKTRACK(assignment, csp)
        if result ≠ failure then
          return result
    remove {var = value} and inferences from assignment
  return failure
```

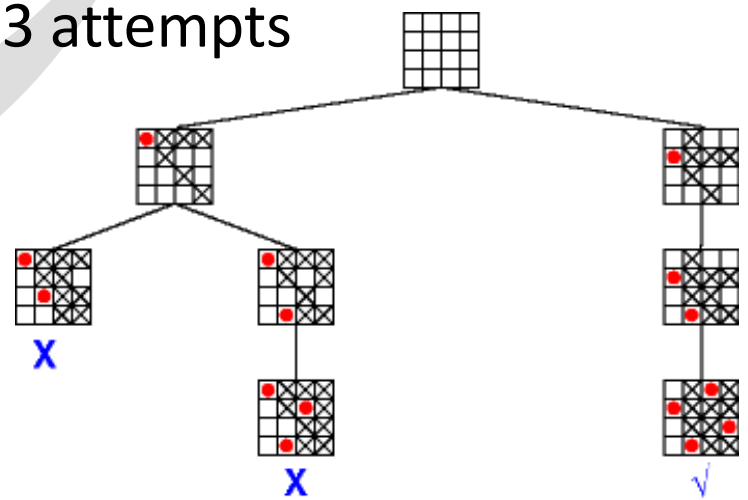


Backtracking is not very good

- 19 attempts

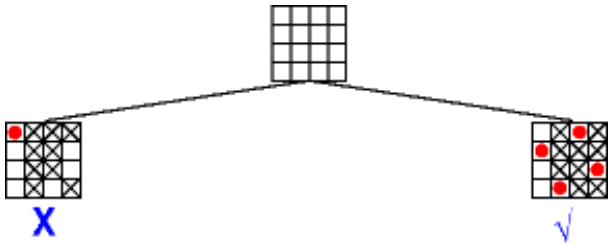
Forward checking is better

3 attempts



And the winner is **Look Ahead**

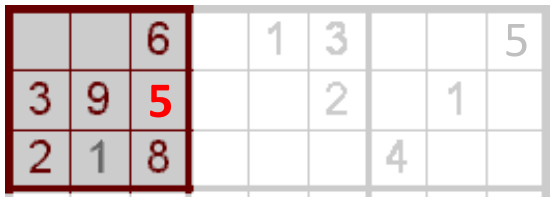
2 attempts

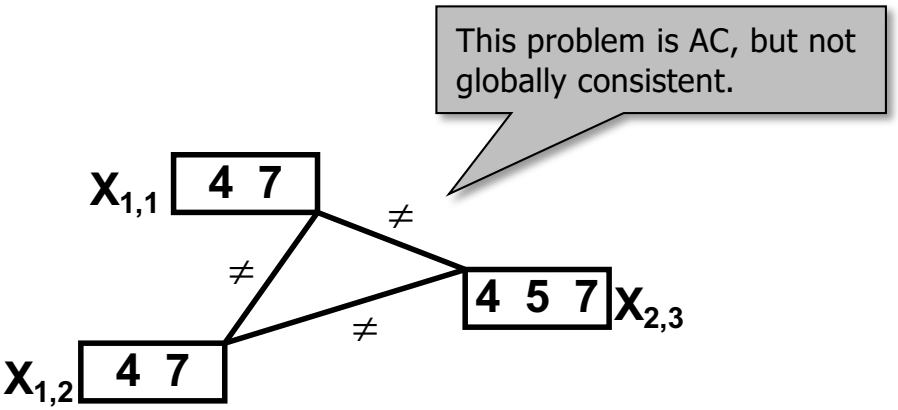


Arc consistency is a form of **local consistency.**

Arc consistency removes values (locally) violating some constraints but **does not guarantee global consistency**.

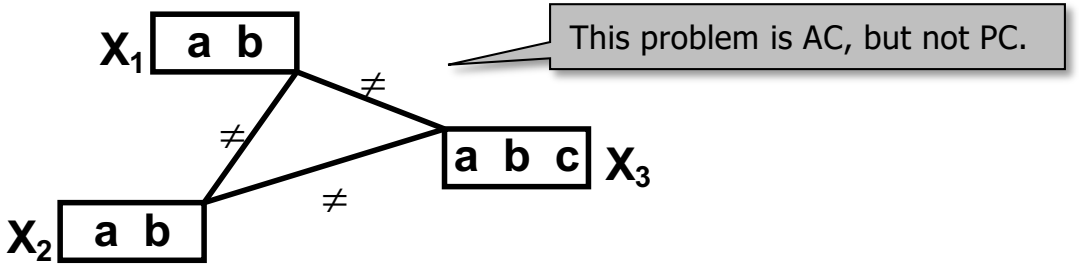
Example (back to Sudoku):





We can generally define **k-consistency**, as the consistency check where for a consistent assignment of (k-1) variables we require a consistent value in one more given variable.

- arc consistency (AC) = 2-consistency
- path consistency (PC) = 3-consistency



Theorem: If the problem is i-consistent ∀i=1,..,n (n is the number of variables), then we can solve it in a backtrack-free way.

- DFS can always find a value consistent with the assignment of previous variables

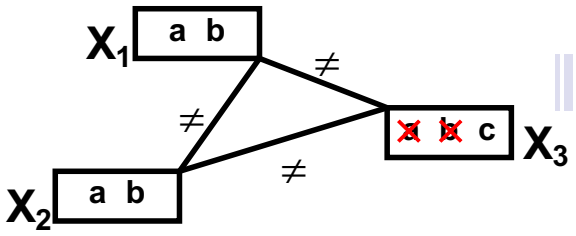
Unfortunately, the time complexity of k-consistency is exponential in k.

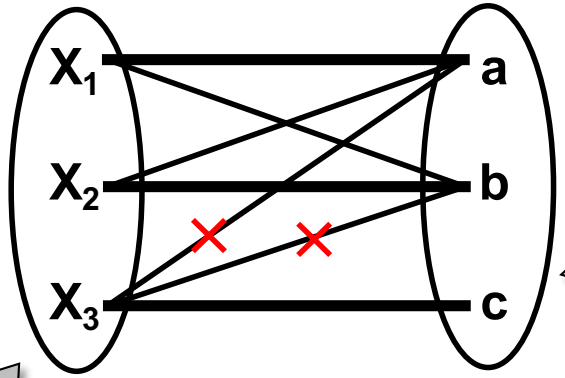
Instead of stronger consistency techniques (expensive) usually **global constraints** are used – a global constraint encapsulates a sub-problem with a specific structure that can be exploited in the ad-hoc domain filtering procedure.

Example:

global constraint **all_different({X_1,…, X_k})**

- encapsulates a set of binary inequalities $X_1 \neq X_2$, $X_1 \neq X_3$, …, $X_{k-1} \neq X_k$
- **all_different({X_1,…, X_k})** = {(d_1,…, d_k) | $\forall i \ d_i \in D_i$ & $\forall i \neq j \ d_i \neq d_j$}
- the filtering procedure is based on matching in bipartite graphs





1. find a **maximal matching**
2. **remove arcs** that are not part of any maximal matching
3. **remove** corresponding **values**

Bipartite graph

- variables on one side, values on the other side
- arcs connect a variable with values in its domain

The backtracking search algorithm instantiates variables in some order and assigns values in some order.

Which variable and value order should be used?

Variable ordering

Fail-first principle: assign first a variable whose assignment will probably lead to a failure

- **dom heuristic**: variable with the smallest domain first
- **deg heuristic**: variable participating in the largest number of constraint first

Value ordering

Succeed-first principle: value belonging to the solution first

How to recognize such a value?

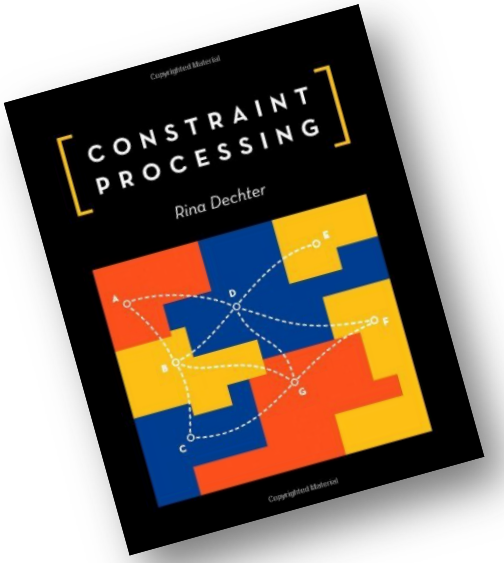
- for example a value that **restricts least the other variables** (keeps the largest flexibility in the problem)
- finding the generally best value is frequently computationally expensive and hence **problem-dependent heuristics** are more frequently used

Constraint Programming is a declarative approach to (combinatorial) problem solving.

- construct a **model** (variables, domains, constraints)
- use a general constraint **solver**
  - combination of **search** (backtracking) and **inference** (domain pruning)
  - **arc consistency** and **global constraints** are the most widely used inference techniques

For more information
course Constraint Programming

- winter term
- http://ktiml.mff.cuni.cz/~bartak/podminky/



© 2020 Roman Barták

Department of Theoretical Computer Science and Mathematical Logic
bartak@ktiml.mff.cuni.cz