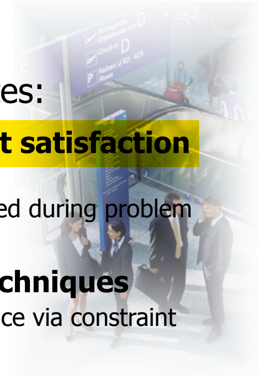
Artificial Intelligence 1

Roman Barták
Department of Theoretical Computer Science and Mathematical Logic

Constraint Satisfaction



- So far we assumed the world states as blackboxes (no internal structure was assumed) accessed via:
  - successor function
  - goal test
  - heuristic function (distance to goal)
- **Today** we will look inside the states:
  - representing problems as **constraint satisfaction problems (CSPs)**
    - state has a structure that can be exploited during problem solving
  - general **constraint satisfaction techniques**
    - depth-first search combined with inference via constraint propagation

- **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.

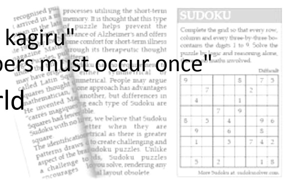
9	6	3	1	7	4	2	5	8
1	7	8	3	2	5	6	4	9
2	5	4	6	8	9	7	3	1
8	2	1	4	3	7	5	9	6
4	9	6	8	5	2	3	1	7
7	3	5	9	6	1	8	2	4
5	8	9	7	1	3	4	6	2
3	1	7	2	4	6	9	8	5
6	4	2	5	9	8	1	7	3

A bit of history

1979: first published in New York
under the name „Number Place“

1986: became popular in Japan
Sudoku – from Japanes "**Su**dji wa **doku**shin ni kagiru"
"the numbers must be single" or "the numbers must occur once"

2005: became popular in the western world



How to find out which digit to fill in?

×	×	6		1	3			
3	9	×					1	
2	1	8				4		

- Use information that each digit appears exactly once in each row and column.

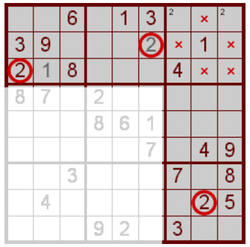
What if this is not enough?

- Look at columns

  or combine information from rows and columns

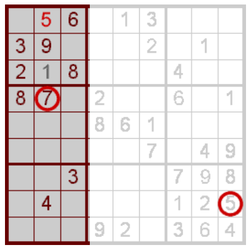
		6	×	1	3			
3	9		×	×	2		1	
2	1	8	×	×	×	4		
8	7		2					
			8	6	1			
					7		4	9
		3				7		8
	4						2	5
			9	2		3		

Sudoku – One More Step

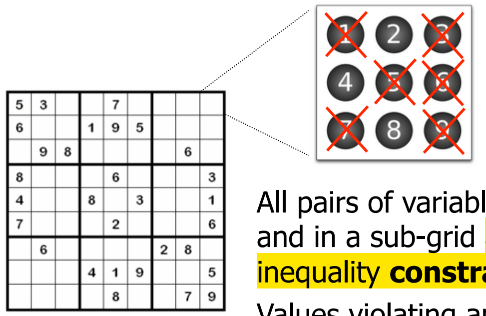


- If neither rows nor columns provide enough information, we can note allowed digits in each cell.

- The position of a digit can be inferred from positions of other digits and restrictions of Sudoku that each digit appears one in a column (row, sub-grid).



Solving Sudoku in general



Each cell can be represented as a **variable** with values taken from a **domain** {1,…,9}.

All pairs of variables in a row, in a column, and in a sub-grid are connected by inequality **constraints**.

Values violating any constraint are **filtered out**.

Such a formulation of problem is called a **constraint satisfaction problem.**

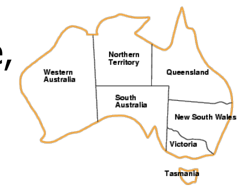
CSP

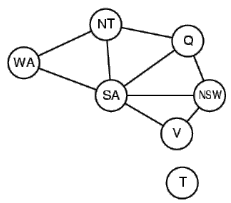
Constraint satisfaction problem consists of:

- 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
  - sometimes, there is a single common "superdomain" and domains for particular variables are defined via unary constraints
- a finite set of **constraints**
  - a constraint is a relation over a subset of variables for example rowA ≠ rowB
  - a constraint can be defined in extension (a set of tuples satisfying the constraint) or using a formula (see above)

CSP – a working example

- Find colours for countries (red, blue, green) such that no neighbours are coloured by the same colour.

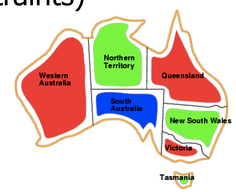




– **Constraint model**
  - variables: {WA, NT, Q, NSW, V, SA, T}
  - superdomain: {r, b, g}
  - constraints: WA ≠ NT, WA ≠ SA ...

– Can also be represented as a **constraint network** (nodes = variables, arc=constraints)

- **Problem solution**
  WA = r, NT = g, Q = r, NSW = g, V = r, SA = b, T = g



State is a partial assignment of values to variables.

A **consistent state** is an assignment that does not violate any constraint.

A complete state is a state where each variable is assigned to some value.

The goal is a complete consistent state.

Sometimes, there is an **objective function** defined over the variables that evaluates the goal states by assigning them real numbers.
Then we are looking for an **optimal goal state**, that is, a goal state with the minimal (or maximal) value of the objective function.

- So far we know various **search algorithms**, so we can apply them to CSPs too..
  - **the initial state**: an empty assignment
  - **applicable actions**: assigning a value to a certain variable such that no constraint is violated
  - **the goal**: a complete consistent assignment

Some notes:

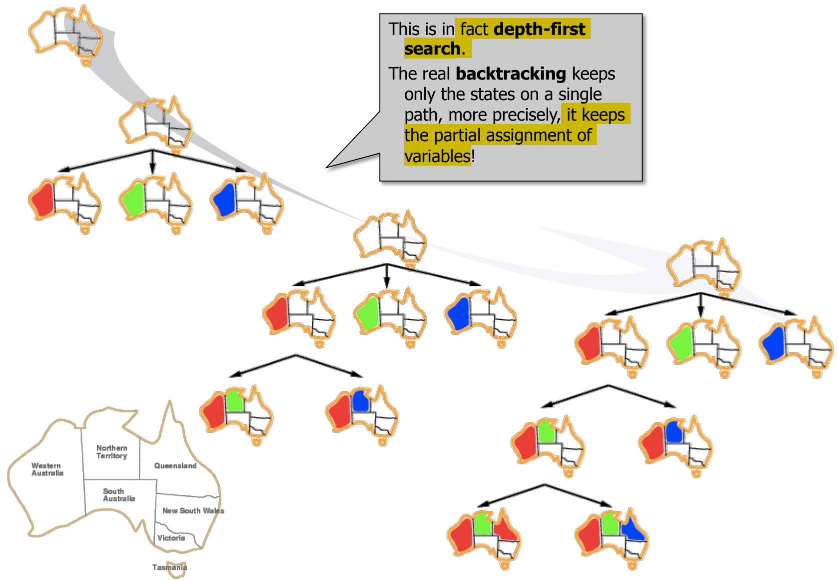
- the same solving approach for all CSPs
- the goal state is always at depth n, where n is the number of variables
  - We can use DFS even with checking cycles!
- the order of actions is not important to reach the goal (a CSP is a **commutative problem**)
  - ⟨WA=r, NT=g⟩ is the same as ⟨NT=g, WA=r⟩
  - we can also use local search techniques
- it is possible to use different branching schemes to solve CSPs, for example domain splitting

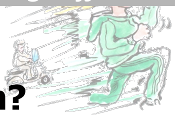
The core uninformed algorithm to solve a CSP:

- gradually assigns values to variables
- if no value can be assigned to a variable then goes back to the previous variable and tries an alternative value for that variable

```
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
```

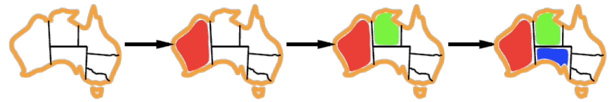





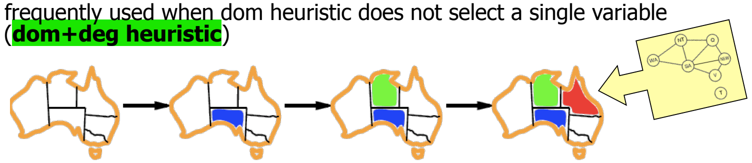
How to influence efficiency of search?

- **assigning the "right" values**
  - this is usually problem dependent
- **the choice of variable for assignment**
  - at the end, we need to assign values to all the variables, but the order of variables influences the size of the search tree
  - problem independent heuristics (such as first-fail)
- **early detection of "wrong" branches**
  - deducing extra information (for example via constraint propagation)
- exploiting a problem structure
  - some problems can be solved using backtrack-free search (for example tree-structured CSPs)

- **The most restricted** variable first
  - a variable with the smallest number of actions
  - i.e. variable with the smallest current domain
  - so called **dom heuristic**

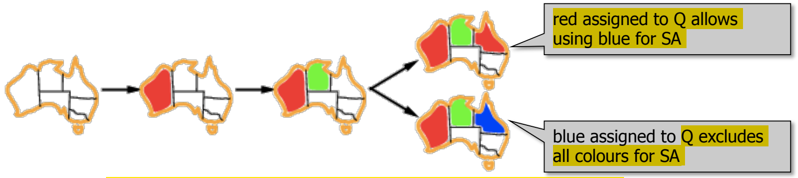


- **The most constrained** variable first
  - participates in the largest number of constraints
  - so called **deg heuristic**
  - frequently used when dom heuristic does not select a single variable (**dom+deg heuristic**)



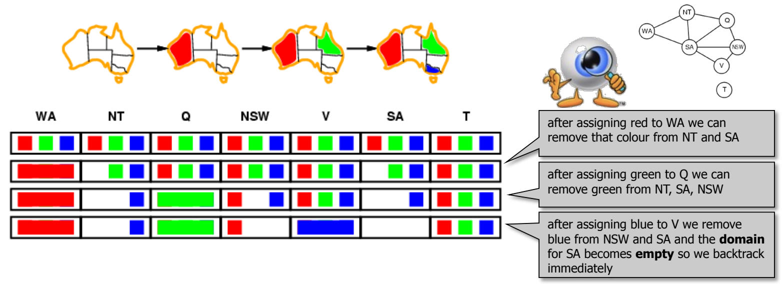
These are instances of the **fail-first principle** – assign first a variable whose assignment will probably lead to a failure.

- When selecting a value for the variable, we prefer values probably belonging to a solution – a **succeed-first principle**.
- How to recognize such a value?
  - for example a value that **restricts least the other variables** (keeps the largest flexibility in the problem)

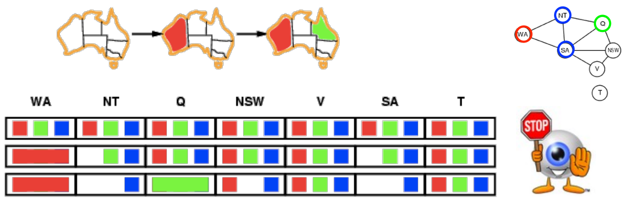


- the value can also be found by relaxing the problem, finding the solution of the relaxed problem, and using values from this solution (recall construction of heuristics)
- finding the generally best value is frequently computationally expensive and hence **problem-dependent heuristics** are usually preferred

- **Can we guess in advance that a given path does not lead to the goal?**
  - After assigning a value to the variable we can check the future constraints – constraints between the current variable and not-yet instantiated variables – **forward checking.**
  - constraint check = remove values violating the constraint

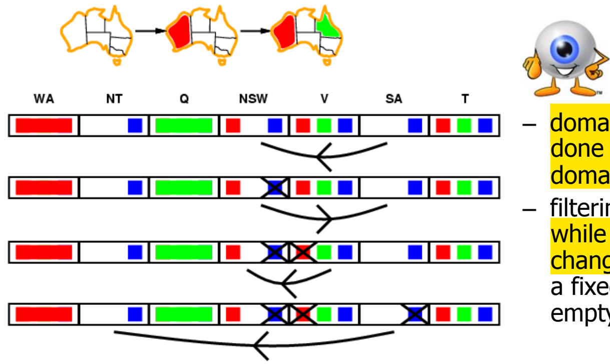


- **Can we exploit the constraints even more?**



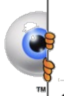
- we can check the constraints even between the future variables; then we can find that blue cannot be used for NT and SA and this is the only colour in their domains
- because the assigned value is propagated through the constraints, this method is called **constraint propagation** or **look ahead**
- this is implemented via maintaining **consistency of constraints**

- each constraint is used to filter out values that violate the constraint
- usually implemented in a directional way – remove values from the domain of X that have no support (a consistent value) in the domain of Y for the binary constraint (X,Y); of course do it also in the reverse direction



- domain filtering in X is done each time the domain of Y changes
- filtering is repeated while the domains are changing until reaching a fixed point or emptying some domain

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

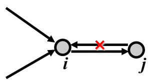


```
function AC-3( csp) returns the CSP, possibly with reduced domains
   inputs: csp, a binary CSP with variables {X1, X2, ..., Xn}
   local variables: queue, a queue of arcs, initially all the arcs in csp

   while queue is not empty do
      (Xi, Xj) ← REMOVE-FIRST(queue)
      if RM-INCONSISTENT-VALUES(Xi, Xj) then
         for each Xk in NEIGHBORS[Xi] do
            add (Xk, Xi) to queue

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

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.

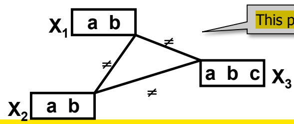


Domain filtering for variable X_i removes values that have no support in the variable X_j, also, if any value is deleted this information is passed to the calling procedure. Knowing constraint semantics can speedup constraint checking (for example X<Y).

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 $O(ed^2)$.

- 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

This problem is AC, but not PC.



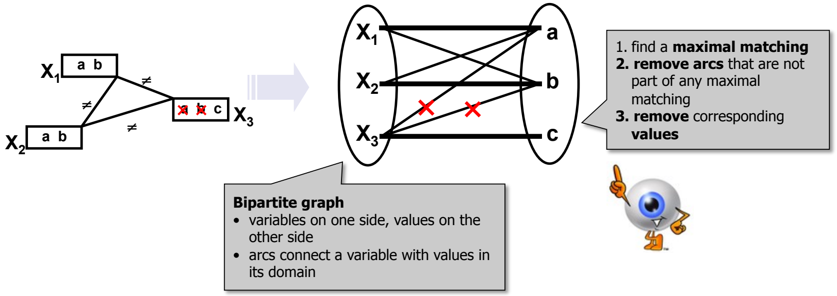
- If the problem is i-consistent ∀i=1,...,n (n is the number of variables), then we can solve it in a backtrack-free way.
  - DFS can always find a value consistent with the assignment of previous variables
- Unfortunately, the time complexity of k-consistency is exponential in k.

- 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-doc domain filtering procedure.

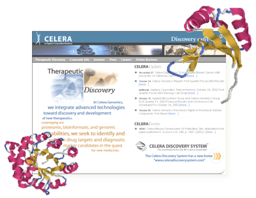
Example:

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



- **A declarative approach** to problem solving
  - construct a **model** (variables, domains, constraints)
  - use a **general constraint solver**
- Possible extensions
  - **optimisation problems**
    - applying branch-and-bound
  - **soft constraints**
    - the constraint describes a preference rather than a restriction
    - optimisation methods are applied there
- Other solving approaches
  - **local search** (the path to the goal is not important)
  - integer programming (for linear constraints)



Bioinformatics

- DNA sequencing
- determining 3D structures of proteins

Planning

- autonomous action planning for space probes (Deep Space 1)

Manufacturing scheduling

- savings after applying CSP: US$ 0.2-1 milion per day

- **Constraint Solvers**
  - SICStus Prolog (available in labs)
  - ECLiPSe (Open Source, http://eclipse.crosscoreop.com/)
  - GECODE (Open Source C++, http://www.gecode.org/)
  - Choco (Open Source Java, http://www.emn.fr/z-info/choco-solver/)
  - …
- Cource **Constraint Programming**
  - also taught in English
  - http://ktiml.mff.cuni.cz/~bartak/podminky/