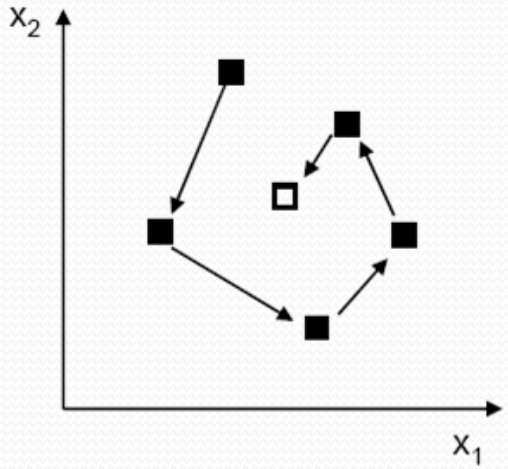
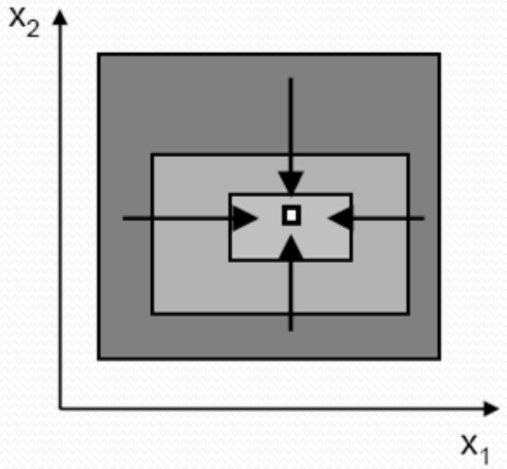
Constraints Satisfaction Problem

As opposed to constructive approaches which explore the solution domain until finding one or several solutions, one other way to find solution can be used : the domain reduction.

In this case constraints that the solution must meet are managed. Not the way to find a solution…



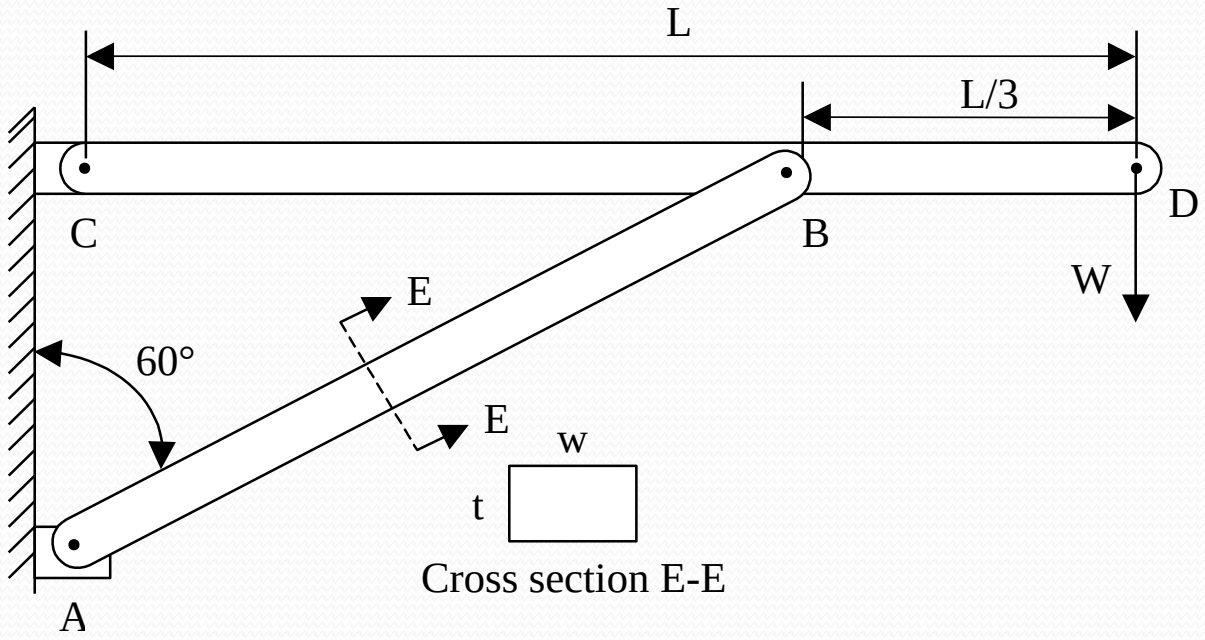
Constructive approach



Domain reduction

Constraints Satisfaction Problem

An example to illustrate this constraint based approach. The aim is to design this simple mechanical system.

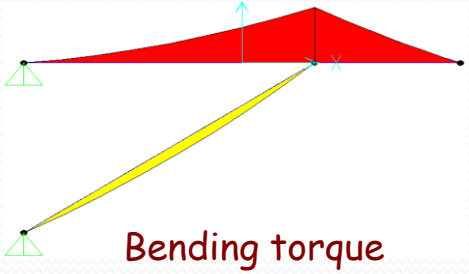


4 design parameters can be quantify by designers: **t, w_{AB}, L, W**
The aim is to find the possible values to this parameters

Constraints Satisfaction Problem

2 mechanical constraints to meet:

- The maximum bending stress σ_b apply to the beam (CD) **is limited by the permissive bending σ_r (225 MPa for steal)**.

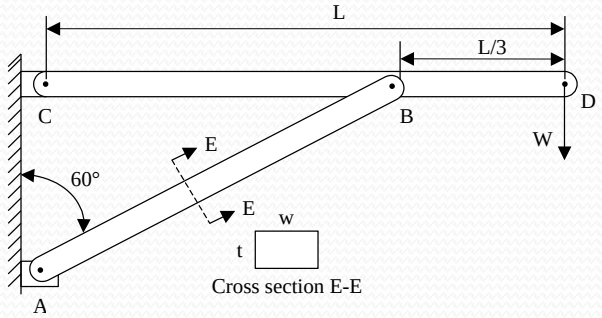


$$\sigma_b = \frac{2L\left(W + \frac{W_{CD}}{6}\right)}{w_{CD}t^2}$$

with

$$W_{CD} = \rho g w_{CD} t L$$

$$w_{CD} = w_{AB} - 0.025$$



- The compressive force $\mathbf{F_{AB}}$ apply to the beam (AB) **is limited by the buckling force limit F_b.**

$$F_{AB} = \sqrt{\left\{\frac{9}{2\sqrt{3}}\left(W + \frac{W_{CD}}{2} + \frac{W_{AB}}{3}\right)\right\}^2 + \left\{\frac{3}{2}\left(W + \frac{W_{CD}}{2}\right)\right\}^2}$$

with $\quad W_{AB} = \rho g w_{AB} t L_{AB} \quad$ and $\quad L_{AB} = \frac{4\sqrt{3}}{9} L$

$$F_b = \frac{\pi^2 E I_{AB}}{L_{AB}^2} = \frac{9\pi^2 E w_{AB} t^3}{64 L^2}$$

Constraints Satisfaction Problem

Safety index

Performance Variables: Y=(***M, s***)

Specification Constraints

$$s \geq 1.5, \quad M \leq 3200$$

System Weight

Performance Constraints

$$\begin{cases} \$s_\sigma = \dfrac{\sigma_r \left(w_{AB} - 0.025\right) t^2}{2L\left(W + \dfrac{1}{6}\rho g t L\left(w_{AB} - 0.025\right)\right)} \\ \$s_F = \dfrac{\left(\dfrac{3\pi^2 E w_{AB} t^3}{32 L^2}\right)}{\sqrt{3\left(W + \dfrac{1}{2}\rho g t L\left(w_{AB}\left(1 + \dfrac{8}{9\sqrt{3}}\right) - 0.025\right)\right)^2 + \left(W + \dfrac{1}{2}\rho g t L\left(w_{AB} - 0.025\right)\right)^2}} \\ M = \rho g t L\left(w_{AB}\left(1 + \dfrac{4\sqrt{3}}{9}\right) - 0.025\right) \\ s = \min\left(\$s_\sigma, \$s_F\right) \geq 1 \end{cases}$$

$E = 207 \cdot 10^9 \ Pa$

$\rho = 7830 \ kg/m^3$

$g = 9.81 \ m/s^2$

$\sigma_r = 225 \cdot 10^6 \ Pa$

Design Parameters: X=(***t, w_{AB}, L, W***)

Eligible Domains

$t \in [0.04, 0.10]$

$w_{AB} \in [0.04, 0.13]$

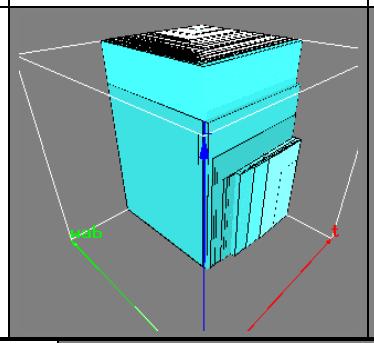
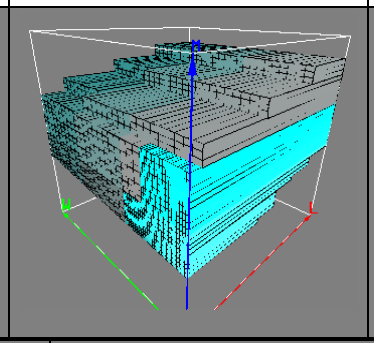
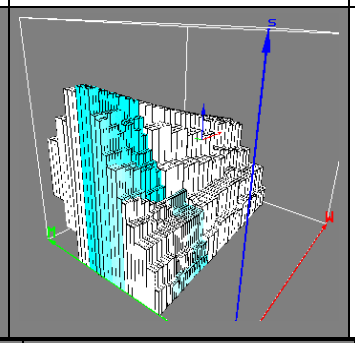
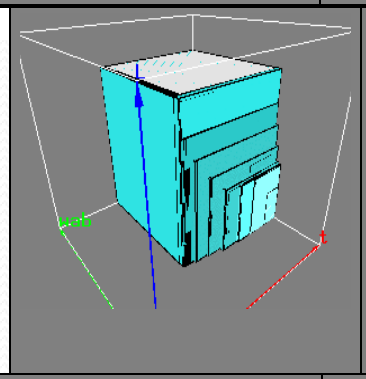
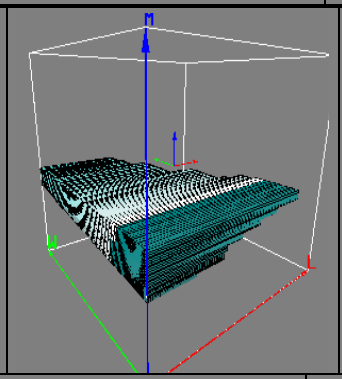
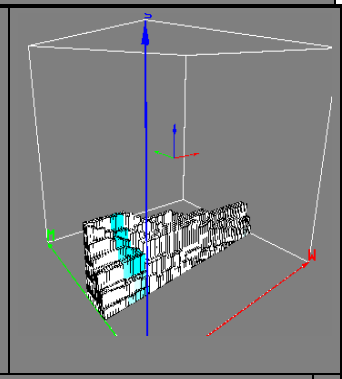
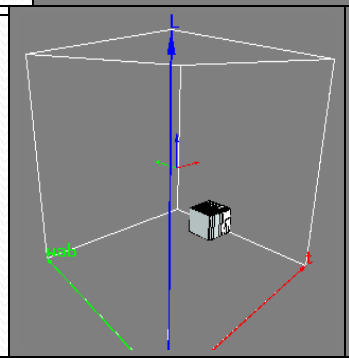
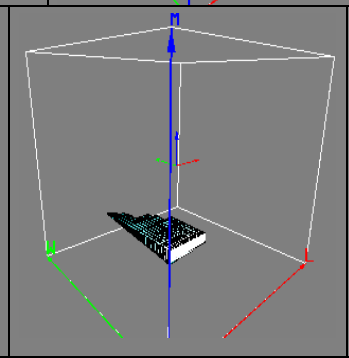
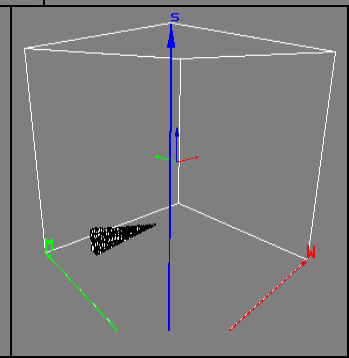
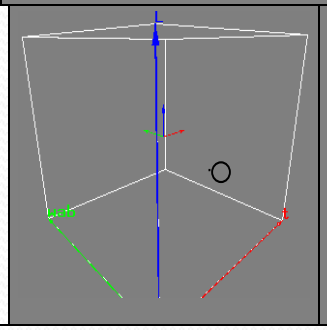
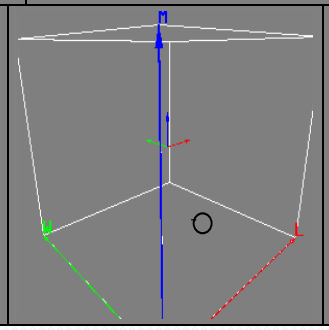
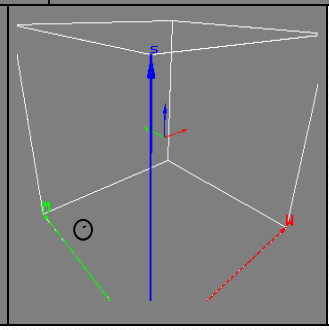
$L \in [3, 4]$

$W \in [15000, 20000]$

Constraints

Design problem **« not too constrained »**

Design problem **« very constrained »**

Stages	$\{t, w_{AB}, L\}$ projection	$\{L, W, M\}$ projection	$\{W, M, s\}$ projection	Hull of design space
1				$t \in [0.0621, 0.1]$ $w_{AB} \in [0.0654, 0.13]$ $L \in [3, 4]$ $W \in [15000, 20000]$ $M \in [2077.9, 6300.9]$ $s \in [1, 2.567]$
2 $M \leq 3200$				$t \in [0.0621, 0.1]$ $w_{AB} \in [0.0654, 0.13]$ $L \in [3, 3.8]$ $W \in [15000, 20000]$ $M \in [2081.5, 3200]$ $s \in [1, 1.664]$
3 $s \geq 1.5$				$t \in [0.0887, 0.1]$ $w_{AB} \in [0.0859, 0.1026]$ $L \in [3, 3.150]$ $W \in [15000, 16638]$ $M \in [2926.5, 3200]$ $s \in [1.5, 1.664]$
4 $L \geq 3.14$				$t \in [0.0996, 0.1]$ $w_{AB} \in [0.0889, 0.0894]$ $L \in [3.14, 3.145]$ $W \in [15000, 15051]$ $M \in [3190.6, 3200]$ $s \in [1.5, 1.505]$

[Yannou]

Constraints Satisfaction Problem

Constraints Satisfaction Problem gathers several technical solutions (mainly and historically from AI) to solve mathematics problems by finding solutions to variables constrained to each other by mathematical relationships.

Find solutions to this kind of problem can be:

- Find all candidate solutions: this is **enumeration**
- Find the best one: this is **optimization**

[Montanari 74] was the first to define a CSP as:

- $D = \{D_1, D_2 \dots D_n\}$ a sequence of domains
- $X = \{X_1, X_2 \dots X_n\}$ a sequence composed by variables which are related to the domains previously defined: $\forall i, X_i \in D_i$
- $C = \{C_1, C_2 \dots X_m\}$ a sequence of constraints
- $R = \{R_1, R_2 \dots R_m\}$ a sequence of relationships, where a each constraint D_j is associated a relationship R_j

- A state of a CSP problem is composed by value assignment to some or all variables: $\{X_i{=}v_i, X_n{=}v_n, \dots\}$.
- An assignment which do not infringe any constraint is called legal or **consistent**
- An assignment is called **complete** if it concerns all variables
- A solution to a CSP problem is **complete and consistent** assignment

Constraints - Type

The constraints can be:

- Numerical : $y = f(x), x = y^2 + z^3$

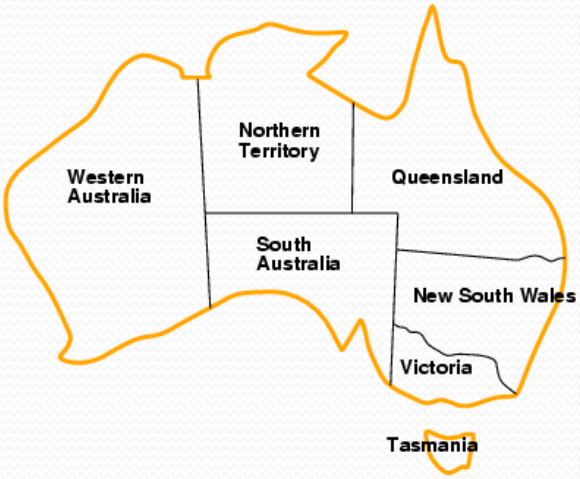
  $y > f(x), x < y^2 - 3z$
- Boolean: $(x \wedge \bar{y})(y \wedge \bar{z})$=True
- Typing: $X \in \mathbb{N}$ ... Y *is an instance of the Class Axial_Feature*
- Sets: $(x \cap y) \cup z = \{a, b, c, d\}$
- Symbols (all can be formatted with the If Then Else structure):
  - Disjunctive numerical: $z \in \{a, b, c, d\}$
  - Complex digital charts:
  - Numerical/Symbolic assignment charts:

Matériau	G	E	ν	Rm	Re	A%	ρ
Acier E24-2	80000	210000	…	340	235	…	7800
AU4G	…	…	…	…	…	…	…

  - Formal (If...Then...Else)
  - Algorithms
  - Simulations...

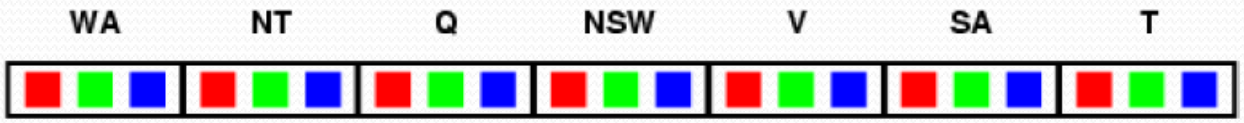
CSP - Discrete Example

A famous example used to illustrate the CSP is the coloring of the Australian states problem:



The aim of this problem is to color each state in a different color from its neighbor. The colors are limited to three: **blue**, **red** and **green**.

- The variables are: $V = \{ WA, NT, Q, NSW, V, SA, T \}$
- The domain of each variable is one composed by the three color: $\{R, G, B\}$



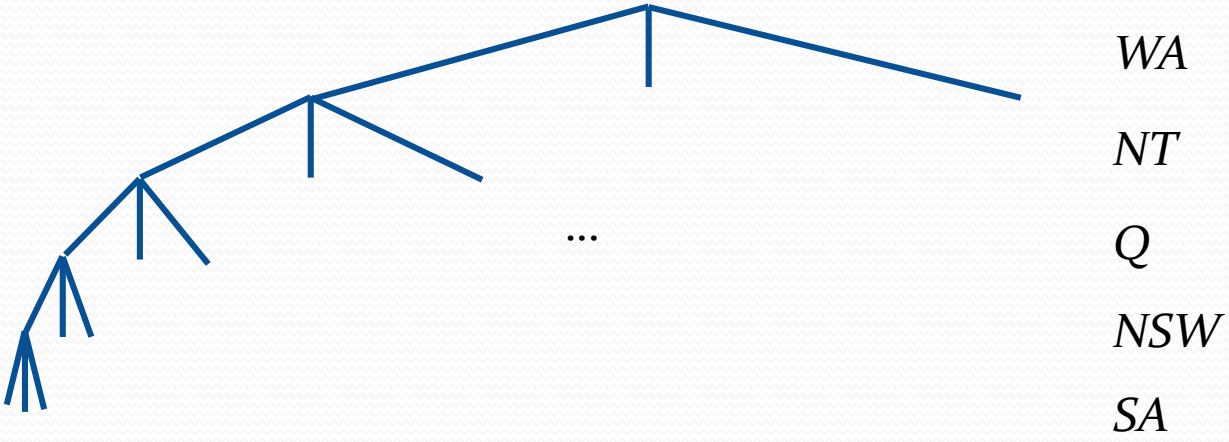
- Constraints: Two neighboring states must have different colors: $WA \neq NT$ ... $NT \neq Q$

CSP - Generate and Test

The first way to perform this solution research is to generate all solutions and to verify if each solution is consistent. This way is:

- Easy to perform since it only needs to define tree structure and for each variable value, to add nodes.
- It becomes totally useless when the combinatory is huge => Process time too long, or impossible.

In the case of the Australian coloring, the number of branches are: $3^6 = 729$



This solution cannot be use in this context of manufacturing process plan selection which is highly combinatorial. An other solution, which prevent the calculation of inconsistent branches as soon as possible must be explore.

CSP - Test and Generate (Back Tracking)

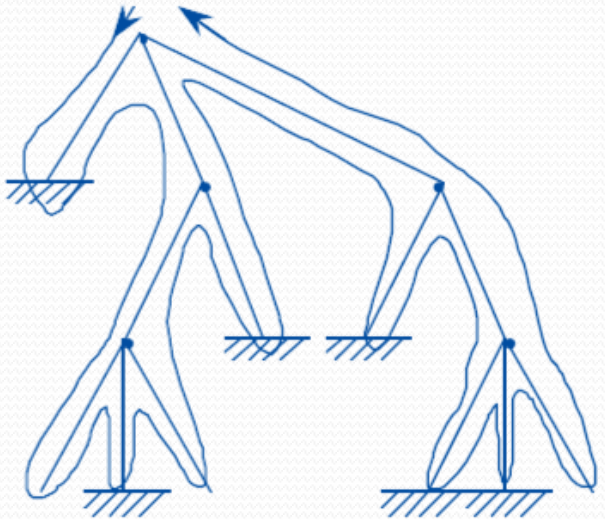
This approach is based on several principles:

- Variable assignment is commutative:

  The assignment order has no importance

  So, WA=Red then NT=green is exactly the same then NT=green then WA=red

- Consequently only one variable can be considered at each node of the research tree
- The algorithm aims to perform deep search by assigning only one value for each variable



Tree exploration by backtracking strategy

CSP - Test and Generate (Back Tracking)

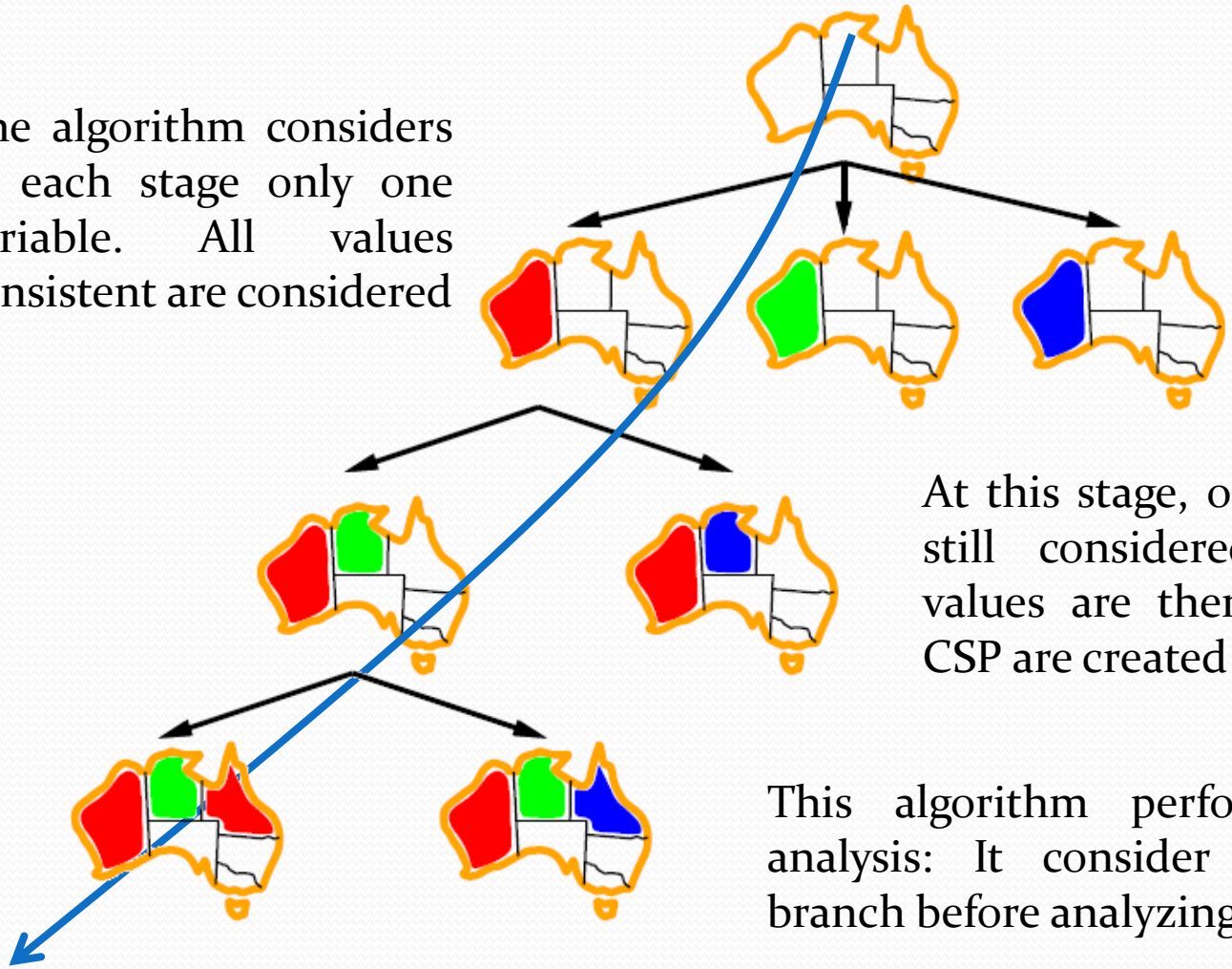
function BACKTRACKING-SEARCH(*csp*) **return** a solution or failure
    return BACKTRACK(*{}* , *csp*)

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

CSP - Test and Generate (Back Tracking)

Applied to the coloring problem, this algorithm works in this way:

The algorithm considers at each stage only one variable. All values consistent are considered



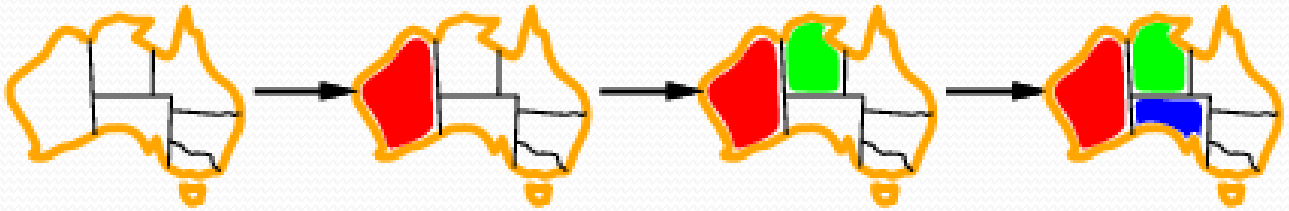
At this stage, only one variable is still considered. Its consistent values are then selected: 2 new CSP are created

This algorithm performs deep analysis: It consider only one branch before analyzing one other

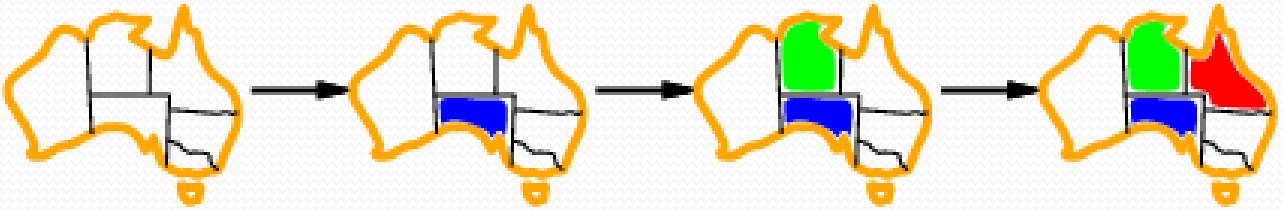
CSP - Back Tracking - Improvements

Some improvements can be proposed to limit the exploration of the tree (constraints verifications) and so the calculation time needed to find solution space:

- By selecting intelligently the variable to manage (SELECT-UNASSIGNED-VARIABLE)
  - Take into consideration variables having the shortest domain

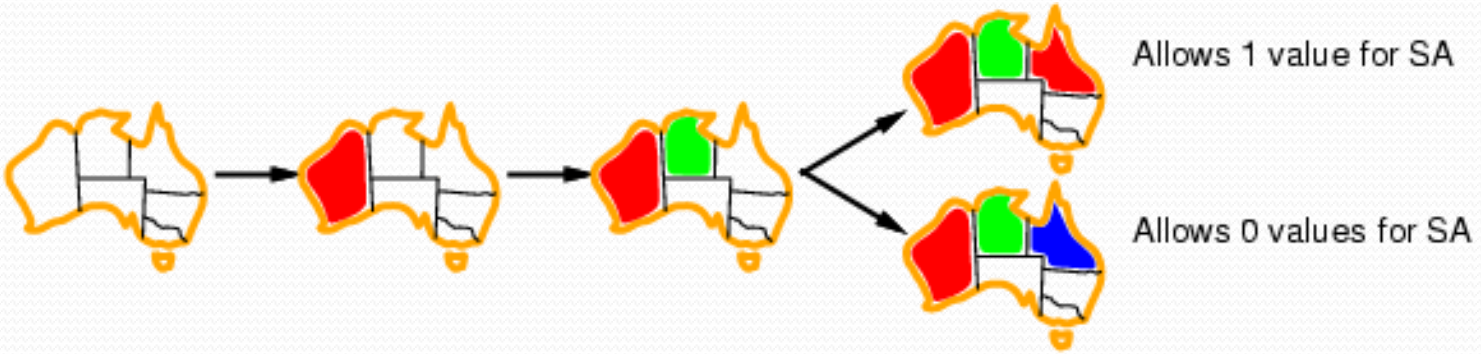


  - Take into consideration variables in relationships with more constraints



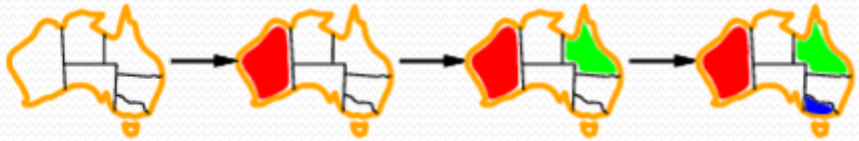
CSP - Back Tracking - Improvements

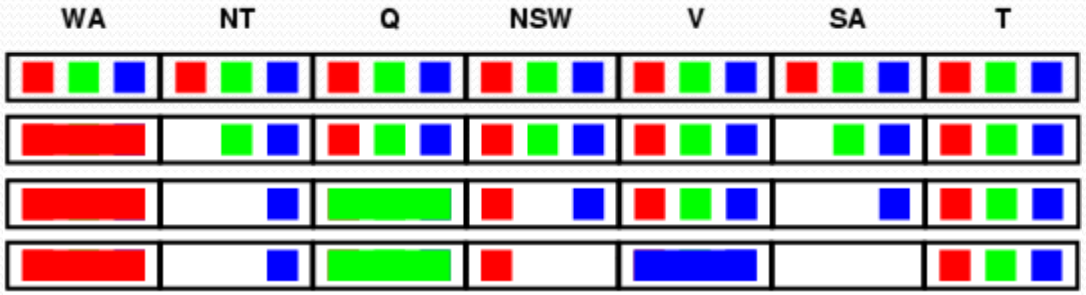
- By ordering the value to give to each variable (ORDER-DOMAIN-VALUES)
  - The aim is to select the less constraining value in order to prevent from limiting the values available for the variables not yet valued



CSP - Back Tracking - Improvements

- By detecting an issue as soon as possible this idea is called **forward-checking**:
  - When a variable is assigned, the algorithm have to check the consistency of all variables in relationships (constraint relations) with it.
  - Pay attention to the fact that if the value is assigned to the variable called X, it checks only the consistency with variables Y since there is a constraint linking them. However, if Y is modified by this action, the algorithm do not take care of the impacts of this modification (and so the other variables linked to Y) of Y!





CSP - Exercise

We want to use Back Tracking to solve this particular type of CSP. This problem is a cryptarithmetic puzzle, a mathematical puzzle in which each letter represents one digit (for example, if X=3, then XX=33).

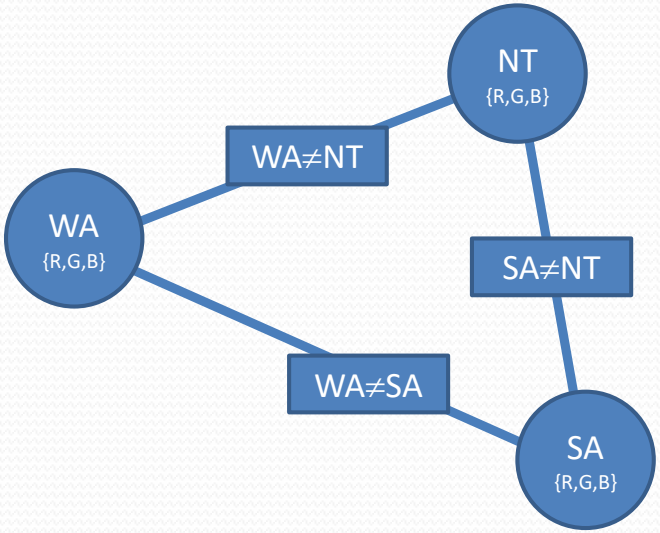
The aim is to find the value of each letter. A digit is represented by only one letter (If X=3, Y cannot be 3). And the first letter cannot be zero (Given the value ZW, Z cannot be zero).

This exercise aims to solve this cryptarithmetic puzzle:

$$\begin{array}{r} \,T\;W\;O \\ +\;T\;W\;O \\ \hline F\;O\;U\;R \end{array}$$

CSP - Exercise

Perform the modeling (as defined mathematically by Montanari) of this CSP problem.



Deduce from this mathematical modeling, the constraints network where each variable and their relationships (the constraints linking them) are drawn.

Considering an excerpt of the Australia map coloring problem detailed during the course (by only regarding WA, NT and SA variables), this network is given in next.

By using the improved *Back Tracking* algorithm (which improves the selection of the variable to consider and its assignment), build the tree this algorithm explores, with this new constraint: **O=8**. Identify the solutions of this CSP.

CSP - Go further ?!

To train yourself, you can try to solve these problems:

- The previous problem without the constraint which regards the assignment of the variable O (O=8). To minimize the depth of the backtracking, remember that O is an even number!

- This other classic and famous cryptarithmetic problem:

$$\begin{array}{cccccc} & S & E & N & D \\ + & M & O & R & E \\ \hline M & O & N & E & Y \end{array}$$

Several software are available to solve CSP (free or expensive):

- ECLIPSe (free): http://eclipseclp.org/index.html
- MiniZinc (free): http://www.minizinc.org/
- Python Constraint (free): http://labix.org/python-constraint
- Google or-Tool (free) : https://code.google.com/p/or-tools/
- IBM ILOG CP (expensive…): http://www.ibm.com/
- …

Need more exercises ?

Try to solve these famous constraint satisfaction problems:

- Sudoku (give as an input the sudoku to solve)
- N queens problem
- One of the problems proposed in http://www.puzzlor.com/ website
- The CSP exercises of the previous final tests GTL S43b
- …

What is MiniZinc?

MiniZinc is a free and open-source constraint **modeling language**. It's not a solver.

MiniZinc can be used to model constraint satisfaction and optimization problems in a high-level, **solver-independent way**. Consequently you only have to learn one programming language: it's compatible and understood by a wide range of solvers: from free solvers bundled with MiniZinc to more costly solutions…

The software runs both on Windows (32 and 64bits), Linux (32 and 64bits) and MAC OS X. Its last version was update in September 2017.

To download this IDE: http://www.minizinc.org/index.html

CSP - Reminder

As explained during the lecture, [Montanari 74] defined a CSP mathematically as:

- $X = \{X_1, X_2 \dots X_n\}$ a sequence composed by **variables** which are related to their domains : $\forall i, X_i \in D_i$
- $D = \{D_1, D_2 \dots D_n\}$ a sequence of **domains** (continuous or discrete)
- $R = \{R_1, R_2 \dots R_m\}$ a sequence of **relationships**, where a each constraint C_j is associated a relationship R_j
- $C = \{C_1, C_2 \dots X_m\}$ a sequence of **constraints**

In MiniZinc IDE:

- Variables and their domains are defined in only one programming line
- Constraints and relationships are merged into only one programming line

How to define a Variable and its domain?

In Minizinc the syntax to define a variable is:

var <*type* or *domain*>: *nameofvariable* [=⟨expression⟩];

- the type can be **int**, **float, bool** (not string)
- the domain is expressed by putting upper and lower boundaries separated by two points

Examples:

- **var** 1..9: T;
- **var float**: L;

Some parameters (that are not to consider in the solving process, such as temporary parameters or constant) can be define in this way:

<*type*>: *nameoftheparameter* [=⟨expression⟩];

How to define a Constraint ?

To define a constraint in Minizinc, the syntax is:

constraint ⟨*Boolean expression*⟩

To express this Boolean relationships, the relational operators can be used:

- Equal: = or == (if you prefer Python's style ☺)
- Not equal: !=
- Strictly less (or greater) than: < (or >)
- Less (or grater) than or equal to: <= (or >=)

Define the type of problem

Since MiniZinc is design to model both CSP and optimization problems, you have to precise in which case your are with the instruction **solve**:

- **solve satisfy;** if you want to find a solution to a CSP problem (if you want to see all solution, modify the MiniZinc preferences)
- **solve maximize** *<Arithmetic expression to maximize>;*
- **solve minimize** *< Arithmetic expression to minimize>;*

This expression can use any type of arithmetic operators such as:

- For **integer**: addition (+), subtraction (-), multiplication (*), integer division (div) and integer modulus (mod)
- For **float**: addition (+), subtraction (-), multiplication (*) and division (/), conversion of int to float (Int2float), absolute value (abs), square root(sqrt), natural logarithm (ln), logarithm base 2 (log2),exponentiation of e(exp), sinus (sin), cosinus (cos), tangent (tan) and power (pow)
- For **Boolean**: and (/\), or (\/), only-if (<-), implies (->), if-and-only-if (<->) and negation (not) => These elements can be useful to combine constraints!

How to display the results ?

To ease the reading of the resulting complete assignment of each decision variable, you can use the following instruction:

output[<list of strings expressions (separate by a comma)>];

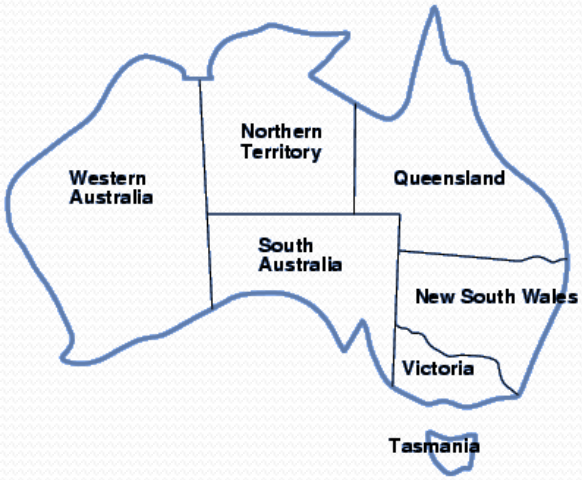
If the data you want to show is not a string, use the function **show**() to perform the type conversion.

To structure the results, you can use particular string expression:

- "**\n**" to add a line break
- "**\t**" to add tabulation

Exercise #1 - Color

Solve the Australia coloring problem by using MiniZinc and print the result.



Tip: Don't use color domains as string expression, but as integer to ease the definition of constraints and their variables.

Exercise #2 - Puzzle

Solve the cryptarithmetic puzzle we partially solved during the course:

$$\begin{array}{cccc} & T & W & O \\ + & T & W & O \\ \hline F & O & U & R \end{array}$$

Compare the computer time required to solve this problem by modifying the way to model constraint: the one with remainders and the one without.

Tip: To add **alldifferent(*<list of variable separate by comma>*)** function, you have to include this instruction to MiniZinc. To do so, add this programming line to the head of your program:

include "alldifferent.mzn";

To go further, solve **SEND + MORE = MONEY** cryptarithmetic problem.

Conditional expression

It is possible in MiniZinc to use conditional expression to differently apply constraints or quantify parameters regarding a set of conditions.

This structure follows this syntax:

if <*Boolean Expression*> **then** <*Expression*> **else** <*Expression*> **endif**;

Two examples:

- int: r = if z==0 then 0 else z endif;
- constraint if z>0 then z<12 else true endif;

Arrays

It is possible to define arrays (matrix, lists…) in MiniZinc by using this programming syntax:

array[<indexset1>,…,<indexsetn>] **of** [**var**] <type>**:** NameVariable;

Example of definition:

- array[1..10] of string: townsname;
- array[2,3] of var float: consumption;

Example of writing:

- consumption[1,2] = 1.2;
- consumption = [| 0.1,0.2,0.3,| 1.1,1.2,1.3 |];
- consumption = array2d(1..2, 1..3, [0.1,0.2,0.3,1.1,1.2,1.3])

row divider

Function transforming a list into an n-dimension array

Constraints on array

To apply constraints on all (or selected) elements of an array, use the **forall** function:

forall ([<Expression> | <Generator> **where** <Boolean Expression>])

forall (<Generator> **where** <Boolean Expression>) (<Expression>)

Examples (they are similar):

- forall([a[j] != a[i] | i,j in 1..3 where i != j]);
- forall (i,j in 1..3 where i != j) (a[j] != a[i]);

forall function can be used to apply constrain on a set of elements or to generate a list of values (to check the maximum value for instance).

Aggregation functions

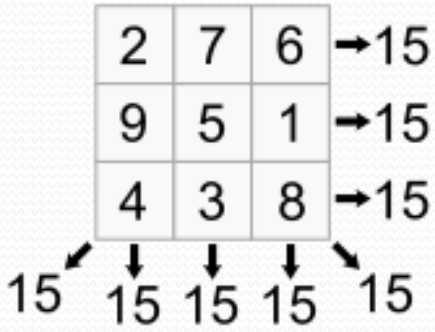
In addition to forall instruction, several other aggregation functions are available in MiniZinc. These functions give one output considering a set of values (arrays, lists) given as inputs:

- For **arithmetic** values:
  - **sum**() and **product**()
  - **min**() and **max**()
- For **Boolean** values:
  - **forall**(): Checks that every values is True
  - **exists**(): Check that at least one value is True

The way to use these aggregation functions is the same than the one explained in the previous slide.

Exercise #3 - Magic Square

A magic square is a n × n square grid filled with distinct positive integers in the range 1 , 2 , . . . , n^2 such that each cell contains a different integer and the sum of the integers in each row, column and diagonal is equal. This sum is called the magic constant or magic sum of the magic square.



Work to do: Design a Minizinc program able to generate a magic square of any size (quantify by the user).

Tip: The Magic sum is defined mathematically as: **$n.(n^2+1)/2$**. Try to solve this problem with and after, without this magic sum equation.

Exercise #4 - Best burger ever ? 1/2

As the owner of a fast food restaurant with declining sales, your customers are looking for something new and exciting on the menu. Your market research indicates that they want a burger that is loaded with everything as long as it meets certain health requirements. Money is no object to them.

The ingredient list in the table (next slide) shows what is available to include on the burger. You must include at least one of each item and no more than five of each item. You must use whole items (for example, no half servings of cheese). The final burger must contain less than 3000 mg of sodium, less than 150 grams of fat, and less than 3000 calories.

To maintain certain taste quality standards you'll need to keep the servings of ketchup and lettuce the same. Also, you'll need to keep the servings of pickles and tomatoes the same.

Question: What is the most expensive burger you can make?

Exercise #4 - Best burger ever ? 2/2

Item	Sodium (mg)	Fat (g)	Calories	Cost (cents)
Beef Patty	50	17	220	25
Bun	330	9	260	15
Cheese	310	6	70	10
Onions	1	2	10	9
Pickles	260	0	5	3
Lettuce	3	0	4	4
Ketchup	160	0	20	2
Tomato	3	0	9	4