Constraint Programming

Intelligent Systems 1
BICS 2

Pierre Talbot (pierre.talbot@uni.lu)

University of Luxembourg

6 May 2020

Intelligent systems: Constraint programming

- Format: lecture + lab combined
  (Alternating lecture and exercises, . . .).
- Webpage of this lecture: hyc.io/teaching/is1.html

Menu

- Introduction
- Satisfaction problem and MiniZinc Syntax
- Constraint solving algorithm
- Optimization problem
- Modelling functions
- Global constraints
- Conclusion

Menu

- Introduction
- Satisfaction problem and MiniZinc Syntax
- Constraint solving algorithm
- Optimization problem
- Modelling functions
- Global constraints
- Conclusion

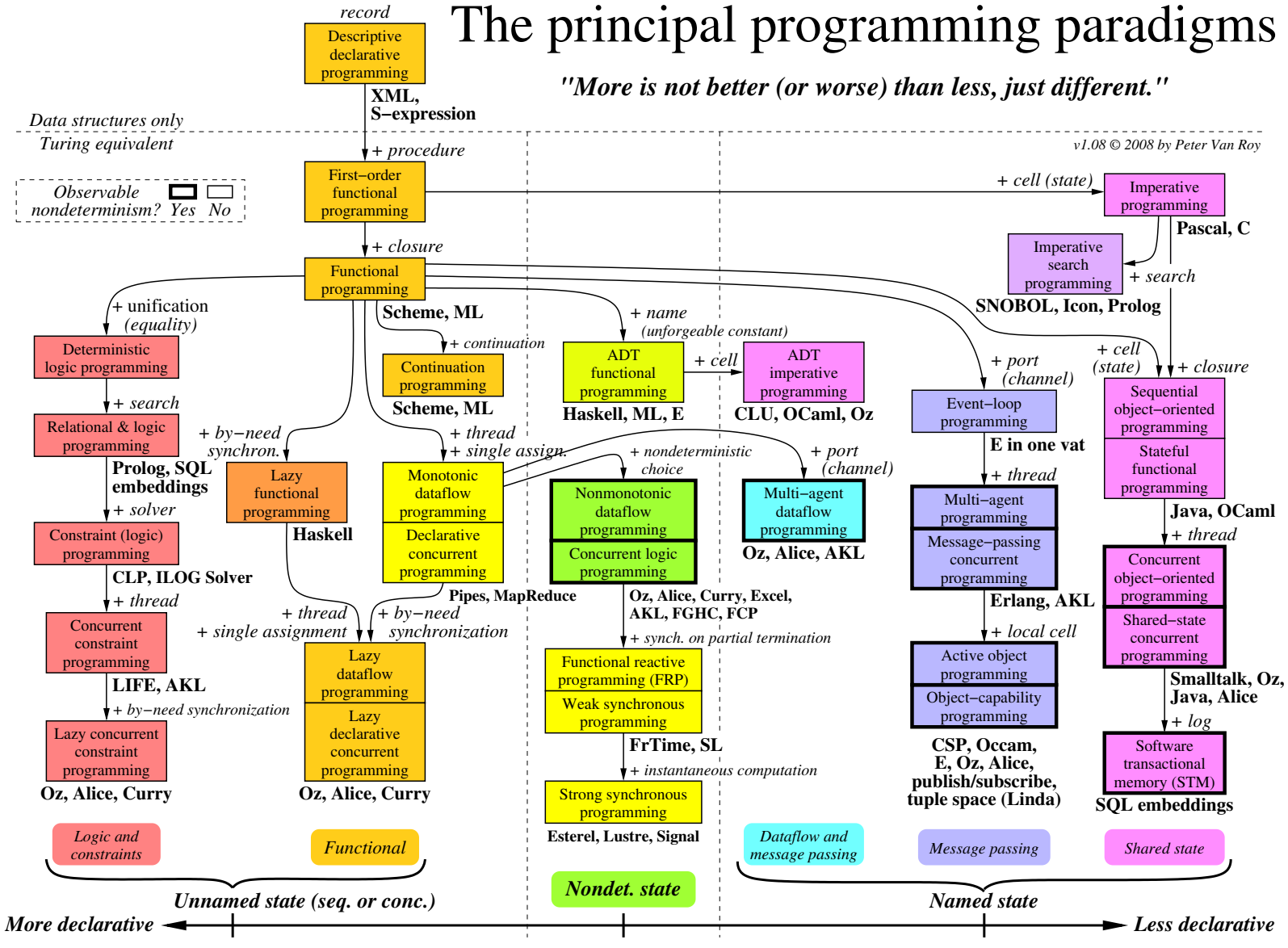
Paradigms

Programming paradigms

- You are used to the imperative paradigm (C, C++, Java, ...),
- maybe object-oriented (C++, Java, ...),
- and probably not functional (OCaml, Haskell, Scala, ...)?

These are only a few of the available paradigms!

The principal programming paradigms



https://www.info.ucl.ac.be/~pvr/paradigms.html

Expla

See "Concepts, Techniques, and Mo

The chart classifies programming pa languages (the small core language i abstractions can be defined). Kernel the creative extension principle: a ne encoded with only local transformat the same paradigm can nevertheless programmer, because they make dif programming techniques and styles

When a language is mentioned unde the language is intended (by its desig without interference from other para is a perfect fit between the language that libraries have been written in th The language's kernel language sho there is a family of related language family is mentioned to avoid clutter. not imply any kind of value judgmen

State is the ability to remember info sequence of values in time. Its expr the paradigm that contains it. We di which differ in whether the state is u nondeterministic, and sequential or c functional programming (threaded s unnamed, deterministic, and sequent declarative concurrent programming deterministic, and concurrent). Add concurrent logic programming (whi nondeterministic, and concurrent). gives message passing or shared stat and concurrent). Nondeterminism is (e.g., client/server). Named state is

Axes orthogonal to this chart are typ Typing is not completely orthogonal Aspects should be completely orthog program's specification. A domain- in any paradigm (except when the de

Metaprogramming is another way to language. The term covers many di programming, syntactic extensibility programming combined with syntac protocols and generics), to full–fled (introspection and reflection). Synta tinkering in particular are orthogona as Scheme, are flexible enough to in native fashion. This flexibility is no

Constraint programming

Today, we learn about a new programming paradigm:
Constraint programming.

- Declarative paradigm,
- Nicknamed "holy grail of computing": we declare our problem and let the computer solve it for us.

Nurse Scheduling Problem

N nurses work M days, find a planning such that:

- Each nurse does at maximum one shift per day (day shift, or evening shift, or late night shift).
- Nurses must not work two nights in a row.
- Nurses must not work the day after a night shift.
- ...

Interactive Soccer Queries

Following the score system of FIFA, answer these questions:

- Can a team still be the champion?
- Is a team is sure / has chance to qualify?
- ...

CP and MIP approaches for soccer analysis, Duque et al., 2019

Harmonization problem

Given a series of N chords, find a permutation to maximize the common notes between every two successive chords.

Musical harmonization with constraints: A survey, Pachet and Roy, 2001

Intuitions

These problems are highly *combinatorial* and generally *NP-complete*. Without optimisations, these problems are almost impossible to solve.

How does it work?

We declare a set of variables, and add constraints (or mathematical relations) on these variables. A solution is an assignment of values to variables, such that all constraints are satisfied.

$$x \in \{0, 1, 2\}, y \in \{1, 2\}, x < y$$

One possible solution is $x = 1, y = 2$.

First example: Sudoku

Can you design an algorithm that find the missing numbers? Is it hard?

Mathematical representation

Let a matrix M of size 9×9 where $1 \leq M_{l,c} \leq 9$ and l is the line and c the column, we have:

- Different lines: $\forall l, c, c',\ M_{l,c} \neq M_{l,c'}$ such that $c \neq c'$.
- Different columns: $\forall l, l', c,\ M_{l,c} \neq M_{l',c}$ such that $l \neq l'$.
- Different sub-squares:
  $\forall l, c \in \{1, 4, 7\},\ \forall sl, sc, sl', sc' \in \mathbb{Z}_3,\ M_{l+sl,c+sc} \neq M_{l+sl',c+sc'}$ such that $sl \neq sl' \wedge sc \neq sc'$.

MiniZinc representation

It is very close from the mathematical representation.
To solve: `solve satisfy;`

```
1  include "alldifferent.mzn";
2  int: N = 9;
3  array[1..N,1..N] of var 1..N: sudoku;
4  constraint forall(l in 1..N)
5    (alldifferent([sudoku[l,c] | c in 1..N]));
6  constraint forall(c in 1..N)
7    (alldifferent([sudoku[l,c] | l in 1..N]));
8  constraint forall(l, c in {1,4,7})
9    (alldifferent([sudoku[l + sl, c + sc] | sl,sc in 0..2]));
10
11 solve satisfy;
12 output [show2d(sudoku)];
```

Input and Output

Input: A Sudoku grid where missing digits are represented by _.
Output: The completed grid.

```
14 sudoku=[|
15 _, _, _, _, _, _, _, _, _|
16 _, 6, 8, 4, _, 1, _, 7, _|
17 _, _, _, _, 8, 5, _, 3, _|
18 _, 2, 6, 8, _, 9, _, 4, _|
19 _, _, 7, _, _, _, 9, _, _|
20 _, 5, _, 1, _, 6, 3, 2, _|
21 _, 4, _, 6, 1, _, _, _, _|
22 _, 3, _, 2, _, 7, 6, 9, _|
23 _, _, _, _, _, _, _, _, _|
24 |];
```

```
Compiling Sudoku.mzn
Running Sudoku.mzn
[| 5, 9, 3, 7, 6, 2, 8, 1, 4 |
   2, 6, 8, 4, 3, 1, 5, 7, 9 |
   7, 1, 4, 9, 8, 5, 2, 3, 6 |
   3, 2, 6, 8, 5, 9, 1, 4, 7 |
   1, 8, 7, 3, 2, 4, 9, 6, 5 |
   4, 5, 9, 1, 7, 6, 3, 2, 8 |
   9, 4, 2, 6, 1, 8, 7, 5, 3 |
   8, 3, 5, 2, 4, 7, 6, 9, 1 |
   6, 7, 1, 5, 9, 3, 4, 8, 2 |]
----------
Finished in 8msec
```

Live demo.

Menu

- Introduction
- Satisfaction problem and MiniZinc Syntax
- Constraint solving algorithm
- Optimization problem
- Modelling functions
- Global constraints
- Conclusion

What tools to solve a constraint problem?

- *Librairies* : GeCode (C++), Choco (Java), ...
- *Languages* : Prolog, **MiniZinc**, ...

Why MiniZinc?

- Easy: Syntax similar to mathematical representation.
- Modular: Allow to solve the model on different solvers (GeCode, Choco, ...).
- Even more modular: Allow to solve the model using different "sub-paradigms" (CP, ILP, MIP, ...). With the same model!

Declare variables

```
int: n = 9; % Parameter
var 1..9: y; % Variable
```

Two kinds of variables

- *Parameters*: The variable `n` is a parameter *fixed before execution*. It can only take one value. For instance, the size of the matrix in the Sudoku.
- *Decision variables*: The variable `y` takes a value between `1` and `9` *after the execution*. For instance, it is a cell of the Sudoku.

Types of variables

`int`, `bool`, `float`, `string`, `set` and `array`.

Add constraints

A constraint is a bidirectionnal relation among variables. Example:

```
var 1..10: x;
var 1..10: y;
int: n = 4;
constraint x = y + n;
```

If x changes, it impacts y, which change as well to satisfy the equality. Similarly, if y changes, it impacts x.

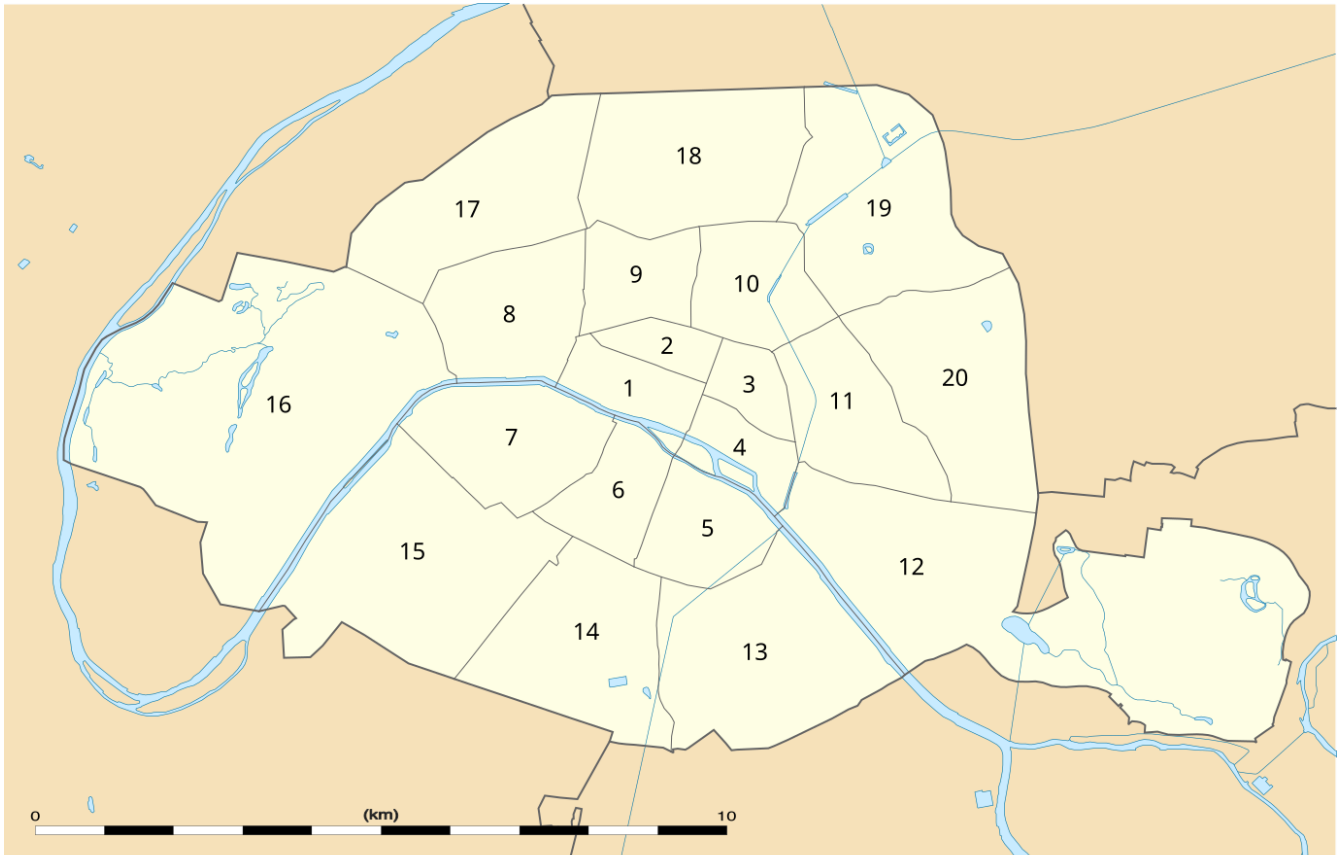
- Arithmetic constraints: `x < y` (and also `<,<=,>,>=,=`)
- Arithmetic expressions: `y - z = x + 3` (`-,+,*,/,mod,div`).
- Boolean constraints: `x != y \/ y = 0` (`\/,/\,->,<->,not`).

Example: Map coloring

Find a coloring of the following map such that:

- Consider only the districts 3, 4, 5, 11, 12.
- Use only three colors.
- Two adjacents districts do not have the same color.

Code skeleton on hyc.io/teaching/is1.html



Solution: Map coloring

```
int: nc = 3;

var 1..nc: dis3;
var 1..nc: dis4;
var 1..nc: dis5;
var 1..nc: dis11;
var 1..nc: dis12;

constraint dis3 != dis4;
constraint dis3 != dis11;
constraint dis4 != dis11;
constraint dis4 != dis12;
constraint dis4 != dis5;
constraint dis11 != dis12;

solve satisfy;
output ["dis3 = \(dis3)\t dis4 = \(dis4)\t dis5 = \(dis5)\n",
        "dis11 = \(dis11)\t dis12 = \(dis12)"];
```

Exercise: Abbot Puzzle

We distribute 100 corn boxes to 100 people such that:

- Each man recieves 3 boxes, each woman 2, and each child half a box.
- There are 5 times more women than men.
- Find a box distribution among men, women and children.

Exercise: Who cheated?

Three students are interrogated to know if someone cheated.

- A : There is one cheater.
- B : There are two cheaters.
- C : There are three cheaters.

The students cheating are always lying, and those not cheating always tell the truth. Who's cheating?

More on MiniZinc

Command line:

- `minizinc paris-color.mzn`: Find the first solution.
- `minizinc -a paris-color.mzn`: Find all solutions.

In the IDE, tick the box "Print all solutions" in the configuration panel.

MiniZinc architecture

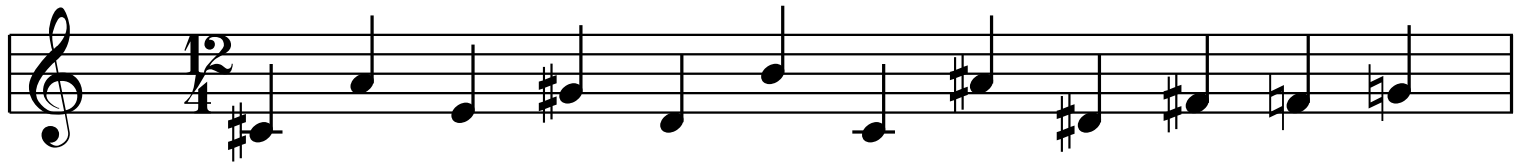
- Model (`.mzn`) + Data (`.dzn`) generate a FlatZinc (`.fzn`) given to a constraint solver for solving the problem.
- Data files set the parameters of a model (example: `nc = 2;`).
- `minizinc paris-color.mzn paris-color.dzn`
- `minizinc paris-color.mzn -D"nc = 2;"`

Menu

- Introduction
- Satisfaction problem and MiniZinc Syntax
- Constraint solving algorithm
- Optimization problem
- Modelling functions
- Global constraints
- Conclusion

Another example: All-interval series problem

For a series of 12 notes, each pitch and each interval between between two successive notes must be distinct.



Model in MiniZinc:

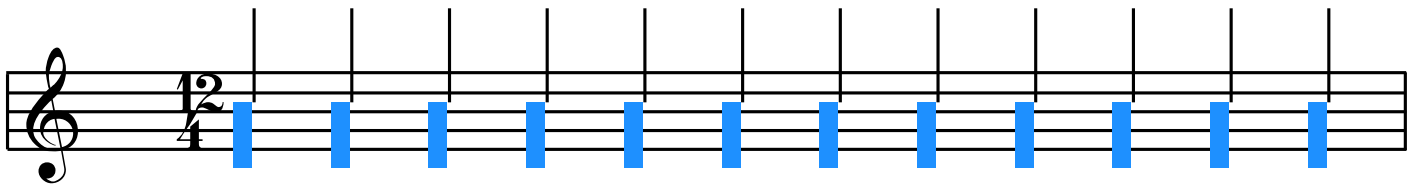
```
int: n = 12;
array[1..n] of var 1..n: pitches;
array[1..n−1] of var 1..n − 1: intervals;
constraint forall(i in 1..n − 1)
                  ( intervals [i] = abs(pitches[i+1] − pitches[i]));
constraint alldifferent ( pitches );
constraint alldifferent ( intervals );

solve satisfy;
```

How does a constraint solver work?

NP-complete nature

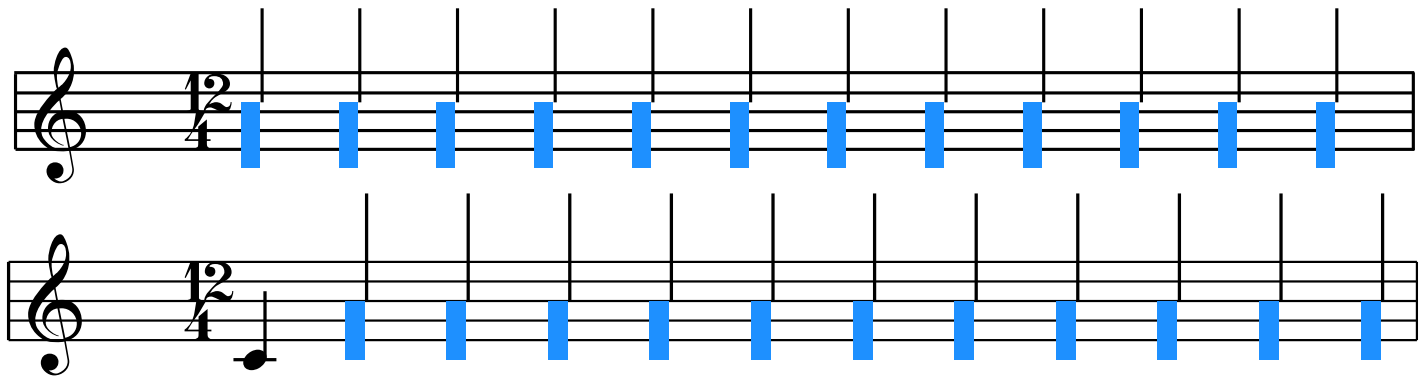
- Try all the combination until we find a solution.
- *Backtracking* algorithm building and exploring the state space.



How does a constraint solver work?

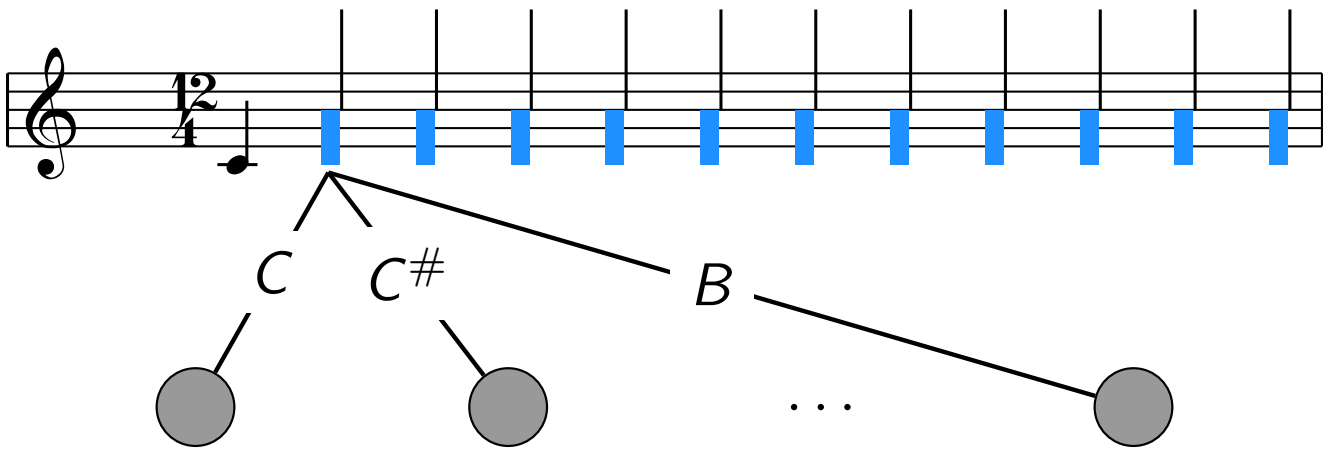
NP-complete nature

- Try all the combination until we find a solution.
- *Backtracking* algorithm building and exploring the state space.

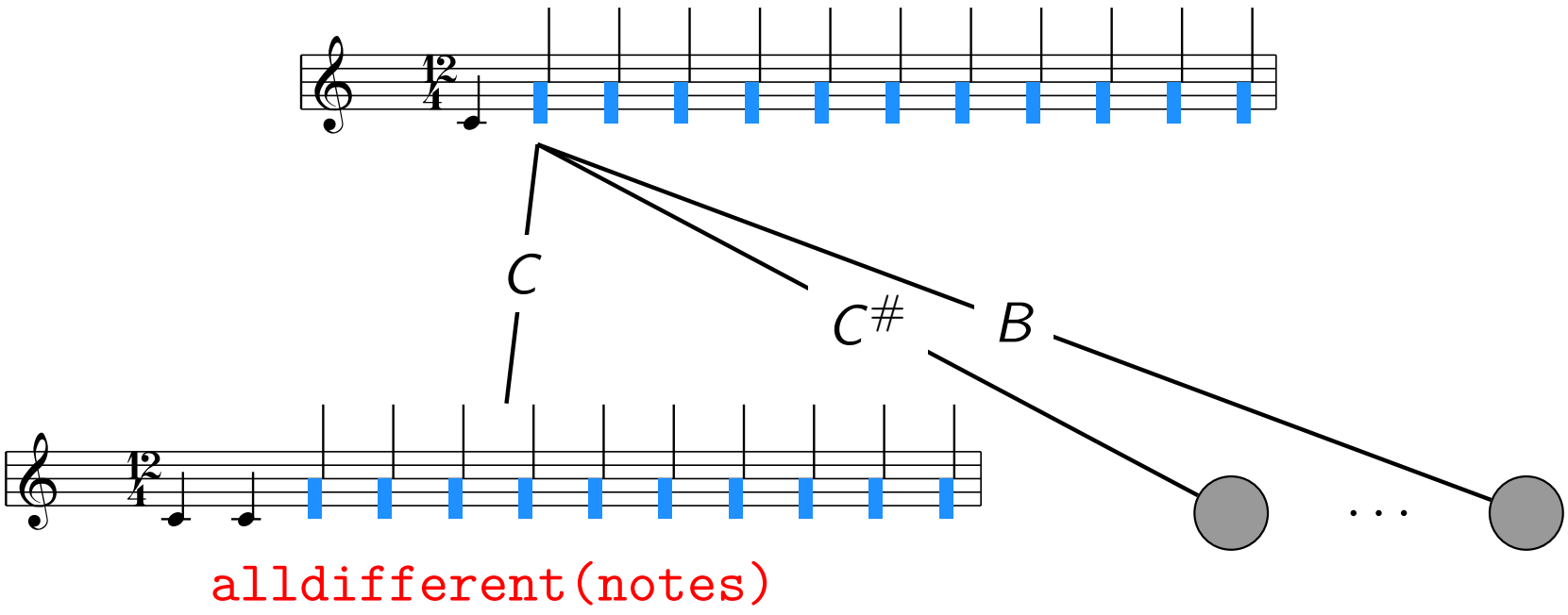


...

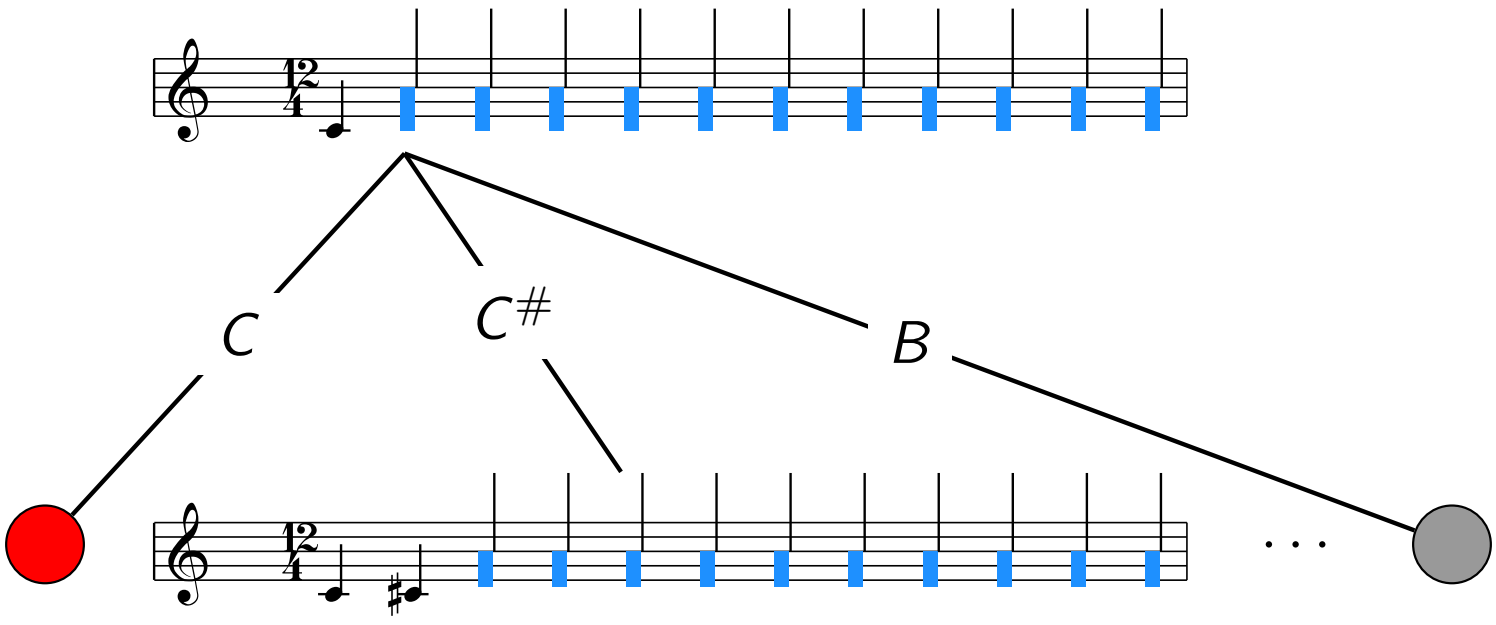
Search tree (step 1)



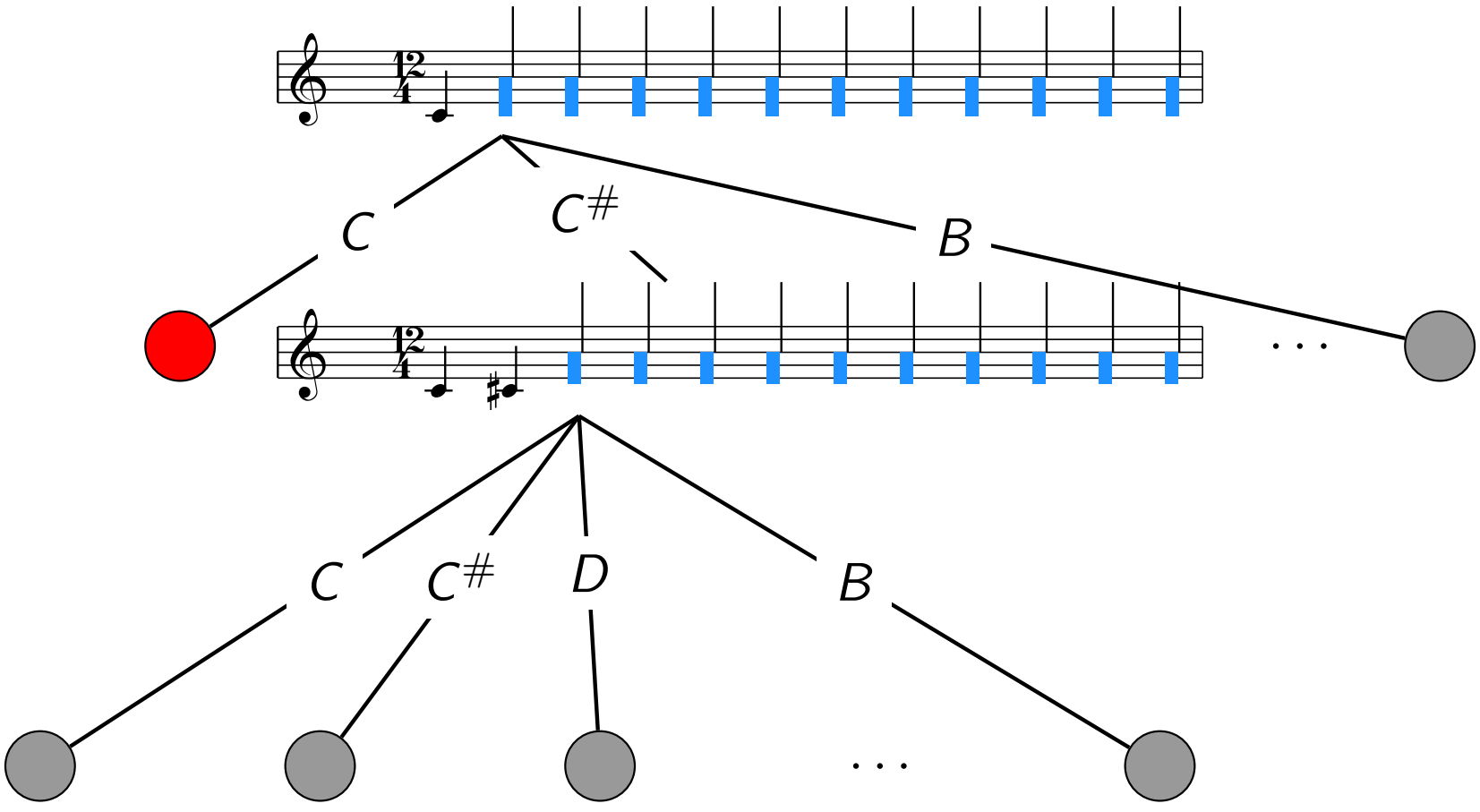
Search tree (step 2)



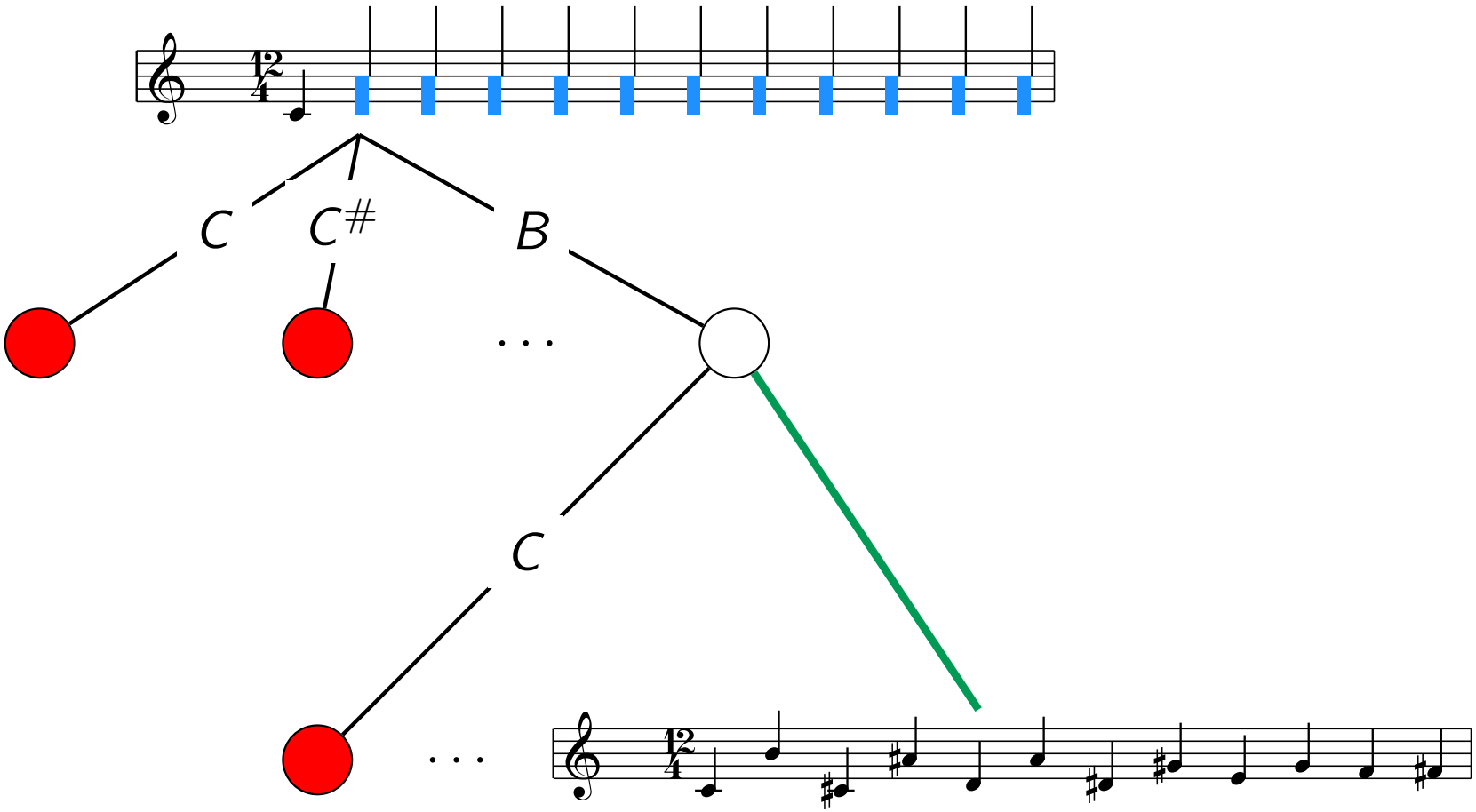
Search tree (step 3)



Search tree (step 4)



Search tree (step 5)



Solving algorithm

All constraint programming solvers are based on this base backtracking algorithm, but contain many additional optimizations.

Optimizations

- Propagation consists in removing inconsistent values in every node.
- Learning schemes in order to avoid making the "same mistake" again (e.g. CDCL in SAT solver).
- Symmetry breaking.
- Various search strategies.
- ...

Menu

- Introduction
- Satisfaction problem and MiniZinc Syntax
- Constraint solving algorithm
- Optimization problem
- Modelling functions
- Global constraints
- Conclusion

Optimization problem

- We wish to find a solution maximizing or minimizing a value among all solutions satisfying the constraints.
- We replace `solve satisfy;` by:
  1. `solve maximize x+y;` to maximize the expression $x + y$.
  2. `solve minimize a;` to minimize the variable a.
- Very common in "real life", we often want to minimize the cost, or maximize productivity, ...

Revisiting map coloring

We want to minimize the printing cost.

- We have three couleurs: white, black and red.
- Printing a district in white costs 1€, in black 2€, and in red 3€.
- What is the coloring with the minimal cost?

Solution: Revisiting map coloring

- Trick: the number of the color is also its cost.
- We have to give an upper bound to the cost.
- For efficiency, you should try to *always give an upper bound to variables*.

```
var 1..nc*5: cost = arr3 + arr4 + arr5 + arr11 + arr12;

solve minimize cost;
```

What to do if the costs are: white (1€), black (1€) and red (4€)?

Menu

Assignment problem

Given objects in a domain, find the matching objects in a codomain.

Examples

- Given the domain of districts $\{1, .., 5\}$, find its matching colors in the codomain $\{1, .., 3\}$.
- Given the domain of colors $\{1, .., 3\}$, find its matching costs in the codomain $\{1, 4\}$.

Set

Reminder

- Parameters (`int: x = 1;`) are instantiated in the beginning.
- Decision variables (`var 1..5: vx;`) are instantiated as the solving process progresses: 1..5 represents the set of *possible values*.

Set

- `set of int: DIS = 1..5;` represents the integer set $\{1, .., 5\}$.
- `var DIS: dis;?`

Set

Reminder

- Parameters (`int: x = 1;`) are instantiated in the beginning.
- Decision variables (`var 1..5: vx;`) are instantiated as the solving process progresses: 1..5 represents the set of *possible values*.

Set

- `set of int: DIS = 1..5;` represents the integer set $\{1, .., 5\}$.
- `var DIS: dis;`? `DIS` stays a set (even after solving!) but `dis` eventually becomes an `int`.
- `var set of int: disS = 1..5;` ?

Set

Reminder

- Parameters (`int: x = 1;`) are instantiated in the beginning.
- Decision variables (`var 1..5: vx;`) are instantiated as the solving process progresses: 1..5 represents the set of *possible values*.

Set

- `set of int: DIS = 1..5;` represents the integer set $\{1, .., 5\}$.
- `var DIS: dis;`? `DIS` stays a set (even after solving!) but `dis` eventually becomes an `int`.
- `var set of int: disS = 1..5;` ? The final type of `disS` is a set, its value belongs to the powerset $\mathcal{P}(\{1, 2, 3, 4, 5\})$.

Arrays

- Indices of an array are represented by a set of integers.
- Matching between DOM and CODOM can be static (cf. `price`).

DOM districts $\{1, .., 5\}$ to CODOM colors $\{1, .., 3\}$.

```
int: dis3 = 1; int: dis4 = 2;
set of int: DIS = 1..5;
set of int: COLOR = 1..3;
set of int: PRICE = {1, 4};
array[DIS] of var COLOR: color;
array[COLOR] of PRICE: price = [1, 1, 4];

constraint color[dis3] != color[dis4];
...
```

How to use arrays in constraints?

Generators

Whenever you have arrays in a programming language, you also have constructs to iterate over these:

- *Fold*: Traverse an array to build an unique object (e.g. sum of an array).

```
array[1..n] of var 1..c: cost;
var 1..n*c: total = sum(cost);
```

- *Map*: Traverse an array to build another array (e.g. multiply all elements by 2).

```
array[1..n] of var 1..c*2: double_cost =
  [cost[i] * 2 | i in 1..n];
```

Generator Map/Filter (list comprehension)

Given an array of elements, for each element satisfying some criterions (filters), we apply an operation and return the new array:

```
% neighbors[i,j] is the score that i gives to j.
array[1..n, 1..n] of 0..p: neighbors;
% We compute the score of everybody, we use a nested list comprehension.
array[1..n] of 0..p*n: score =
  [sum([neighbors[i,j] | j in 1..n where i != j])
   | i in 1..n];
```

Fold generator

Given an array of elements, return a result.
We can compute the global score of all people:

```
var 0..p*n*n: global_score =
  sum([neighbors[i,j] | i,j in 1..n where i != j]);
```

Syntactic sugar

Instead of `sum([e | g])` we can write `sum(g)(e)`:

```
var 0..p*n*n: global_score =
  sum(i,j in 1..n where i != j)
     (neighbors[i,j]);
```

Other generators

- Similarly to `sum(array)`, we have `product(array)`.
- To generate a conjunction of constraints, we use `forall(array)`:

```
int: n = 3;
constraint forall(i,j in 1..n where i < j)
                    (t[i] != t[j]);
```

Importantly, the forall is unfolded at *compile-time* into:

```
constraint t[1] != t[2] /\ t[1] != t[3] /\ t[2] != t[3];
```

Map coloring (with arrays)

Rewrite the map coloring optimization problem with arrays, starting from the following example:

```
int: dis3 = 1; int: dis4 = 2;
set of int: DIS = 1..5;
set of int: COLOR = 1..3;
set of int: PRICE = {1, 4};
array[DIS] of var COLOR: color;
array[COLOR] of PRICE: price = [1, 1, 4];

constraint color[dis3] != color[dis4];
...
```

Menu

- Introduction
- Satisfaction problem and MiniZinc Syntax
- Constraint solving algorithm
- Optimization problem
- Modelling functions
- Global constraints
- Conclusion

Sub-structure of a problem

Many problems share identical sub-structure, for instance “all the elements of this array must be distinct”.

Global constraints

- N-ary constraints capturing a particular sub-problem.
- Reusable abstraction across problems.
- They can be implemented very efficiently using dedicated algorithms.

Alldifferent constraint

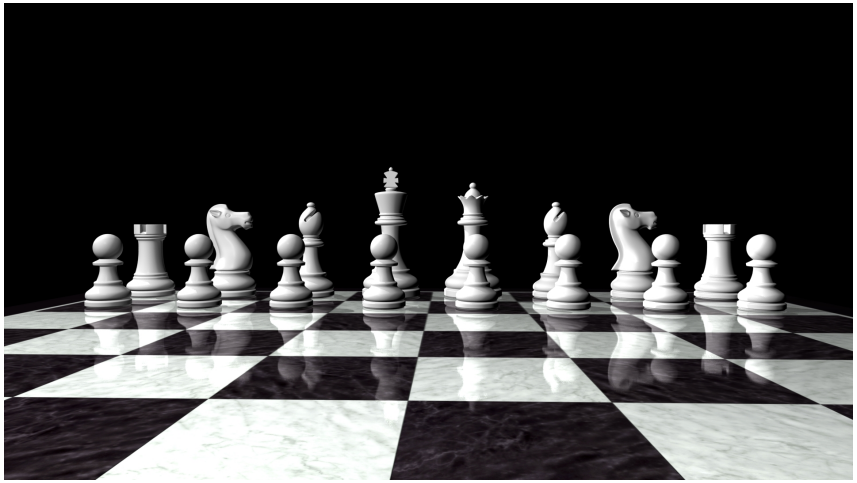
The predicate `alldifferent(array)` is bundled in the MiniZinc standard library. To use it, we must include the corresponding library.

```
include " alldifferent .mzn";
int: N = 9;
array[1..N,1..N] of var 1..N: sudoku;
constraint forall(l in 1..N)( alldifferent ([sudoku[l,c] | c in 1..N]));
```

Note: We can use `include "globals.mzn"` to include all the available global constraints at once.

N-Queens problem

Given a chess board of size $N * N$, place on each line a queen such that no queen can attack another one (line, column, the two diagonals).
Test with $N = 35$, does global constraints help to reduce the execution time?

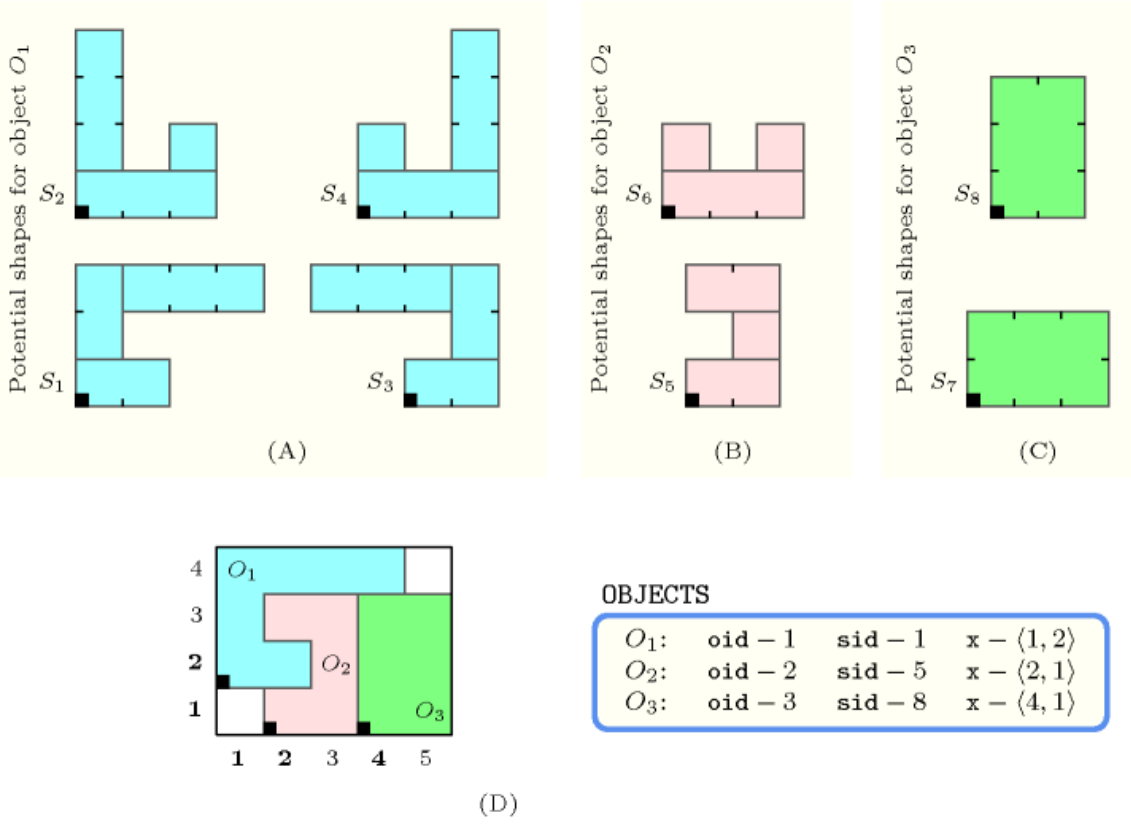


Packing Problem

- Loading a truck with packages while balancing the weight.
- Optimal plan of an appartment to maximize the number of rooms (of a minimal size).
- Electronic design automation: packing of blocks into a circuit layout.

Geost

These problems have a common sub-structure captured by the global constraint `Geost(k, Objects, SBoxes)`.



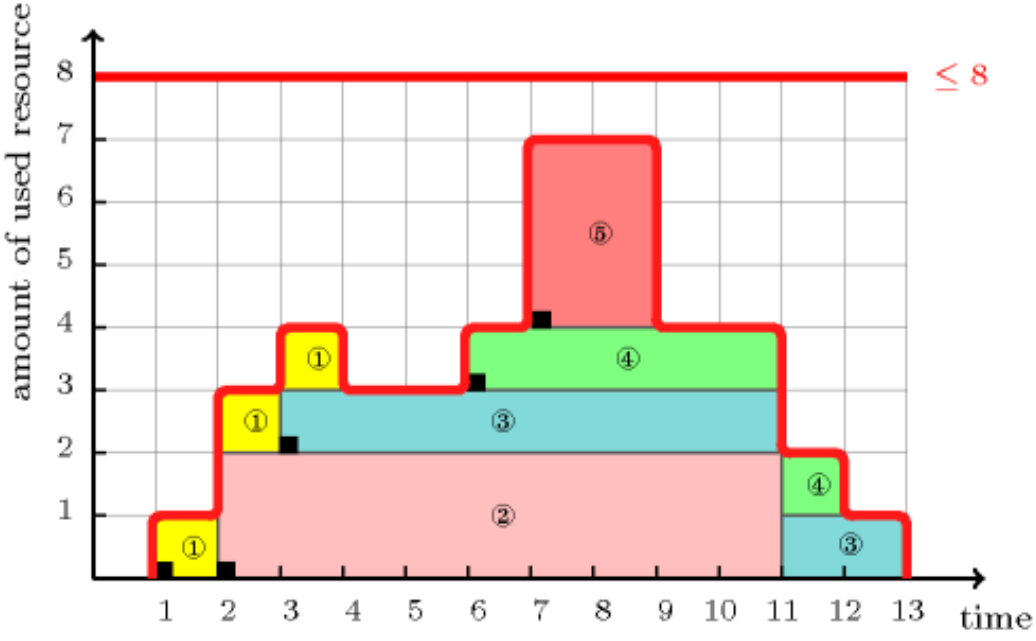
Job scheduling under resources

- Course planning where teachers have preferences.
- Maximize productivity in a factory.
- ...

Cumulative

```
predicate cumulative(array [int] of var int: s,
                     array [int] of var int: d,
                     array [int] of var int: r,
                     var int: b)
```

Given a set of tasks, a task i starts at time $s[i]$ for a duration of $d[i]$, and use $r[i]$ resources. The capacity of resources per instant is b (on the diagram, $b = 8$).



Global constraint catalog

There are many global constraints (> 400), a part of it is being referenced on `http://sofdem.github.io`.

- Don't panick! It's mostly variants of a smaller pool of "main" global constraint.

Menu

- Introduction
- Satisfaction problem and MiniZinc Syntax
- Constraint solving algorithm
- Optimization problem
- Modelling functions
- Global constraints
- Conclusion

Conclusion

The paradigm of constraint programming is suited to solve combinatorial problem.

- We declare the problem, and let the system solve it for us.
- We focus on the problem, rather than the resolution method.
- Numerous solving algorithms exist depending on the problem (local search, MIP, SMT...).

This is only an introduction...

- Graph modelling, string constraints, ...
- Symmetry breaking
- Search strategy: IDDFS, LDS, ...
- Parallelization (EPS, ...)

To go further

The following online classes and papers are nice places to start:

- Coursera – Modeling Discrete Optimization (P. Stuckey, J. Lee)
- Coursera – Discrete Optimization (P. Van Hentenryck): More general course on various paradigms (CP, MIP, local search).
- Guido Tack. Constraint Propagation – Models, Techniques, Implementation
- If you are interested to know more, don't hesitate to contact me, and share your progresses ;-)