Modeling Languages

Yves Deville[1], Christine Solnon[2]

(1) UCLouvain, Belgium
(2) University of Lyon, France

SLS 2009

In collaboration with Vianney le Clément[1], Jean-Noël Monette[1],
Pascal Van Hentenryck (Brown Univ.)

The context of SLS 2009

This tutorial has close relationships with the first tutorial *Computer-assisted design of high-performance algorithms* by Holger Hoos

Similar objectives

- Help the user to design an efficient algorithm
- The user focusses on higher level design issues

The main differences

- Offering a *modeling language* to the user to design problems
- Dedicated to an application domain
- Focus on *Constraint-Based approaches*
- Algorithm synthesis based on the *structure of the problem*

The context of SLS 2009

This tutorial has close relationships with the first tutorial *Computer-assisted design of high-performance algorithms* by Holger Hoos

Similar objectives

- Help the user to design an efficient algorithm
- The user focusses on higher level design issues

The main differences

- Offering a *modeling language* to the user to design problems
- Dedicated to an application domain
- Focus on *Constraint-Based approaches*
- Algorithm synthesis based on the *structure of the problem*

Overview

Outline

Constraint Satisfaction Problems

CSP

- A set of variables, defined over domains
- A set of constraints over the variables
- A solution is an assignment of values to the variables which satisfies all the constraints
- Possibility to add an objective function

Example : N-Queens

The problem

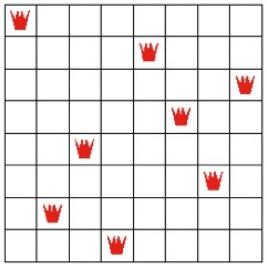
How to place 8 queens on a 8x8 board such that they do not attack each other

Variables

$X_i \in \{1, \ldots, 8\}$: position (column) of the queen on row i

Constraints

- Two queens cannot be on the same column : $X_i \neq X_j$
- Two queens cannot be on the same diagonal : $|X_i - X_j| \neq |i - j|$

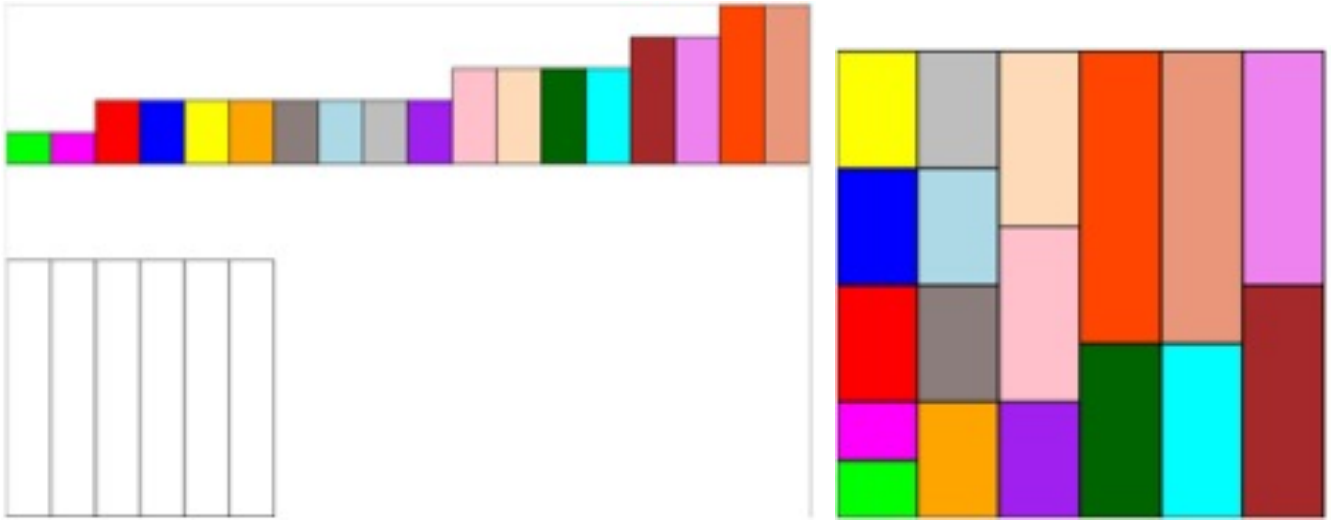


Example : Bin Packing

The problem

Given a list of objects of size $w_1, \ldots, w_n$,
m bins of capacity W
Assign each object to a bin, such that

- the capacity of the bins are not exceeded
- the number of used bins is minimal



Solving problems modeled with constraints

Program = Model + Search

Modeling

Model the problem as a CSP

- Define the constrained variables
- Specify constraints expressing relations between objects
- Specify the objective function (optimization problem)

Search

Design the search component
Depend on the chosen approach

- Local Search (incomplete, pertubative)
- ACO (incomplete, constructive)
- Constraint Programming (complete, constructive)

Outline

Comet : a language for Constraint-Based Programming

The Comet paradigm

Comet program = Model + Search

Models are expressed by means of constraints

Comet approaches

Comet includes

- Constraint-Based Local Search
- Constraint Programming
- MIP

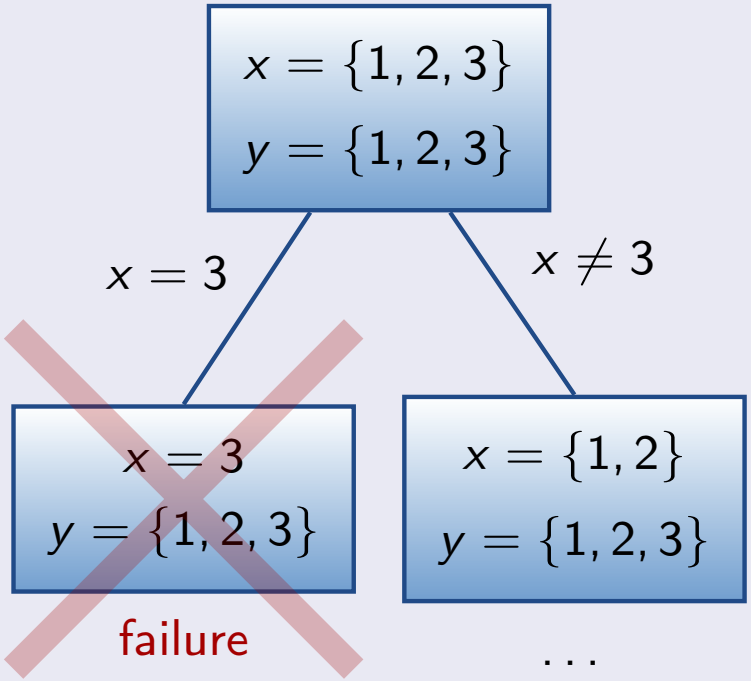
The Comet language is freely available for academics

Comet is distributed by Dynadec

CP versus CBLS with Comet

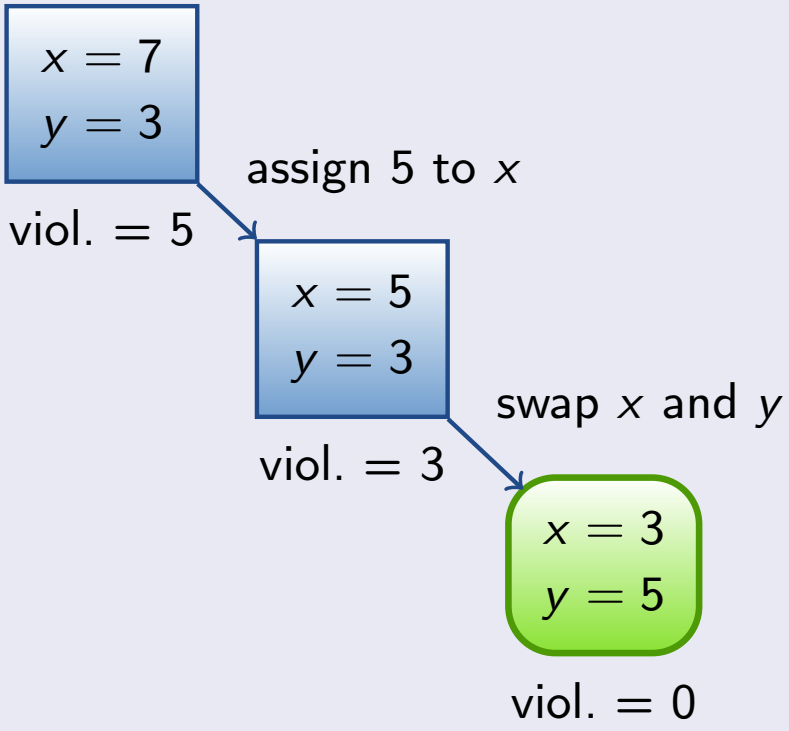
$$x < y$$

CP search



- Constraints $\longrightarrow$ propagators
- Complete

Local search (LS)



- Constraints $\longrightarrow$ violations
- Quickly find good solution

Constraint Programming with Comet

The CP search

Branch & Propagate

Branching

- Decompose into subproblems (e.g., giving a value for a variable)
- Automatic support for backtracking

Propagation

- Reduction of the search space
- Find an equivalent CSP with *smaller domains*
- Based on consistency techniques

CBLS with Comet

Constraints

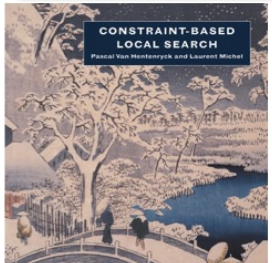
- Measure of their violation
- Differentiable objects : show how much a local move affects the violations

Objective function

- Measure of its value
- Differentiable object : show how much a local move affects the objective

Invariant

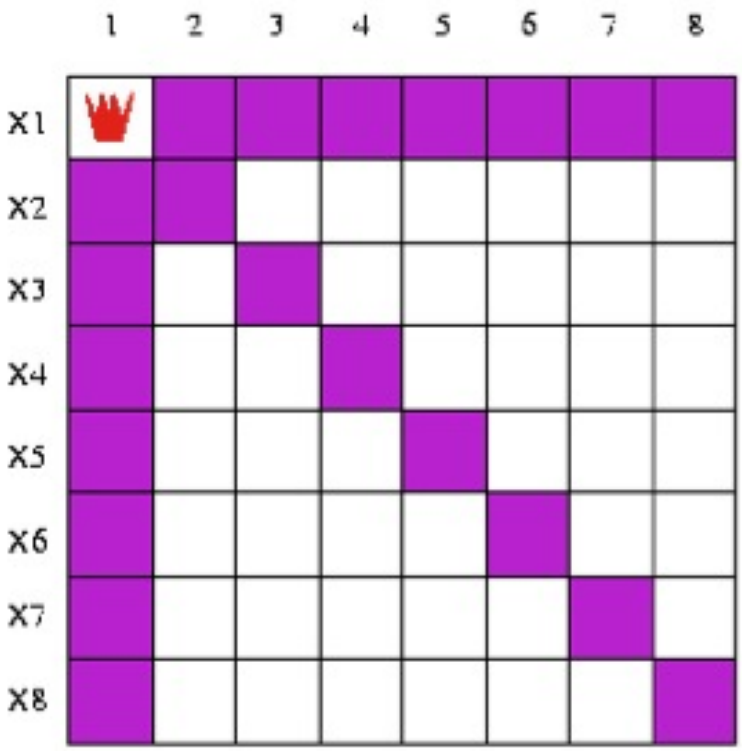
Maintain an expression incrementally

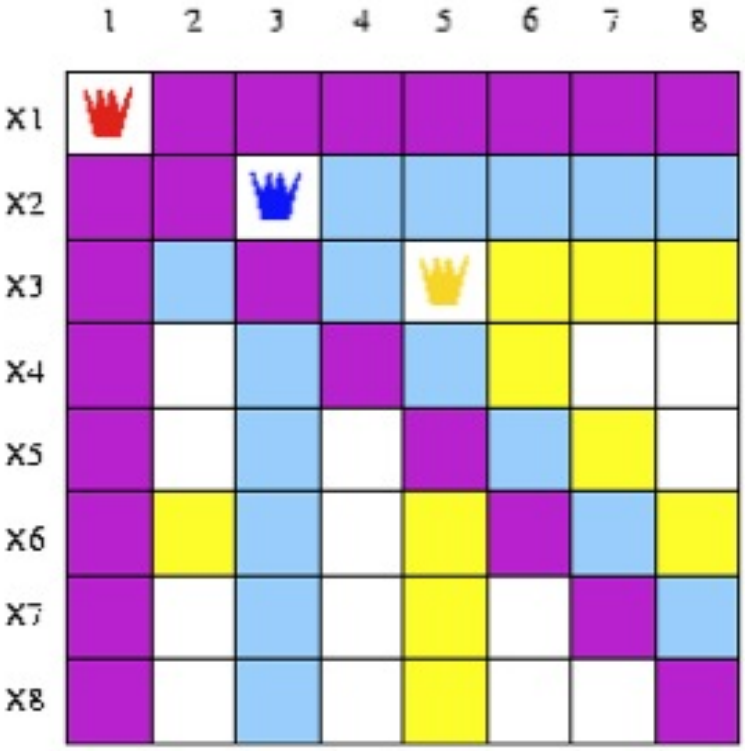


N-Queens : Comet / CP

```
import cotfd;
Solver<CP> cp();
int n = 8;
range S = 1..n;
var<CP>{int} q[i in S](cp,S);
solve<cp> {
  cp.post(alldifferent(q));
  cp.post(alldifferent(all(i in S) q[i] + i));
  cp.post(alldifferent(all(i in S) q[i] - i));
}using{
  forall(i in S : !q[i].bound())
                    by (q[i].getSize()) {
    tryall<cp>(v in S : q[i].memberOf(v))
      label(q[i],v);
  }
}
cout << q << endl;
```

N-Queens : Comet / CP

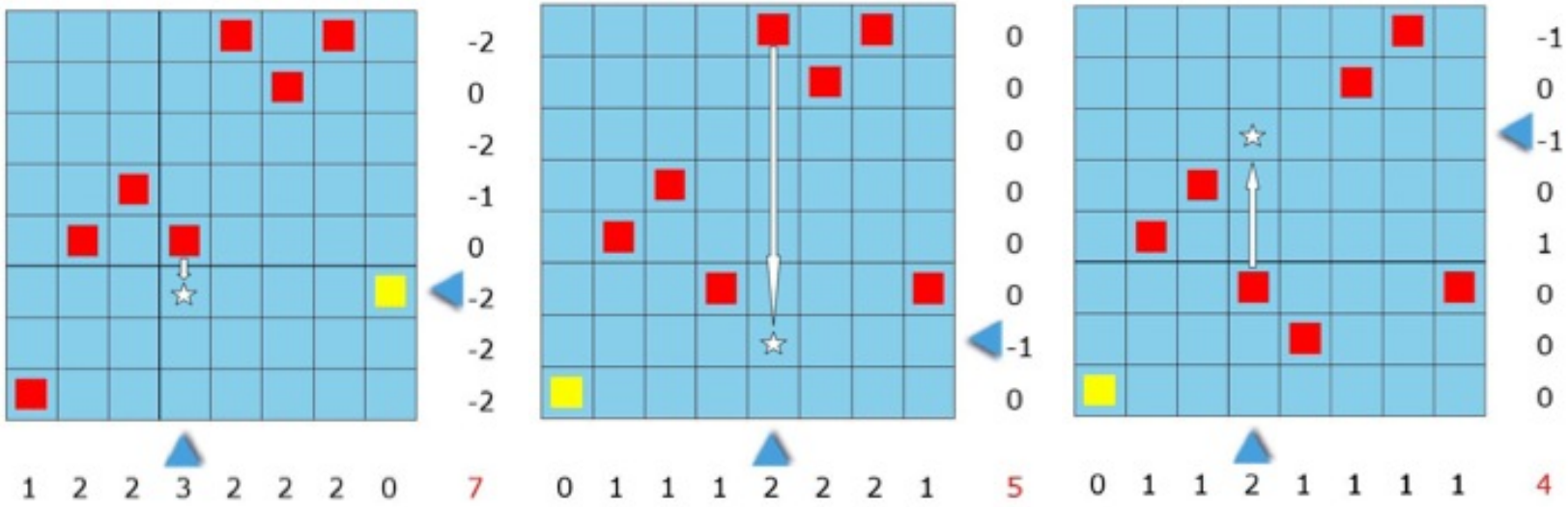




N-Queens : Comet / CBLS

```
import cotls;
Solver<LS> ls();
int n = 8;
range S = 1..n;
var{int} q[i in S](ls,S) := i;
ConstraintSystem<LS> CS(ls);
  CS.post(alldifferent(q));
  CS.post(alldifferent(all(i in S) q[i] + i));
  CS.post(alldifferent(all(i in S) q[i] - i));
ls.close();
int it = 0;
while (CS.violations() > 0 && it < 50*n) {
  selectMax(i in S)(CS.violations(q[i]))
    selectMin(v in S)(CS.getAssignDelta(q[i],v))
      q[i] := v;
  it++;
}
cout << q << endl;
```

N-Queens : Comet / CBLS



Bin-packing data in COMET

```
int n = ...; // number of bins
int m = ...; // number of objects
int W = ...; // capacity
range Rbin = 1..n; // range of the bins
range Robj = 1..m; // range of the objects
range RW = 0..W; // range of the capacity
int w[Robj] = ...; // size of each object
```

CP model in COMET

```
import cotfd;

Solver<CP> cp();
var<CP>{int} b[Robj](cp,Rbin); // bin assigned to each object
var<CP>{int} load[Rbin](cp,RW); // load of the bins
cp.close();

minimize<cp>
   max(obj in Robj) b[obj]
subject to {
   forall(j in Rbin)
      cp.post( load[j] == sum(i in Robj) (b[i] == j)*w[i] );
   cp.post( sum(j in Rbin) load[j] == sum(i in Robj) w[i] );
} using {
   label(b);
}
```

LS model in COMET

```
import cotls;

Solver<LS> ls();
var{int} b[Robj](ls,Rbin);
var{int} load[bin in Rbin](ls,RW)
      <- sum(obj in Robj) (b[obj] == bin) * w[obj];
Function<LS> objective = MinBinObjective(ls, b, load, w, W);
ls.close();

// Initialization
int curBin = Rbin.getLow();
forall (obj in Robj) {
   if (curBin < Rbin.getUp() && load[curBin] + w[obj] > W)
      curBin++;
   b[obj] := curBin;
}
```

LS model in COMET

```
// Search
int iter = 0;
while (iter < 1000) {
   iter++;
   selectMin(obj in Robj, bin in Rbin :
                load[bin] + w[obj] <= W)
         (objective.getAssignDelta(b[obj], bin)) {
      b[obj] := bin;
   }
}
```

Outline

Strength and weakness of constraint-based approaches

The holy grail of constraint programming...

The user states the problem by means of constraints
The computer solves it thanks to embeded solvers

...faced to the reality of NP-hardness

The user often has to help the computer:

- Choose the most appropriate search paradigm
  - CP when constraints are tight enough to prune efficiently
  - CBLS for looser constraints and/or optimization
- Design the "right" model that leads to an efficient search
  - CP: add redundant constraints
  - CBLS: choose appropriate invariants
- "Program" the search
  - CP: design ordering heuristics
  - CBLS: neighborhoods and strategies for escaping local optima

Strength and weakness of constraint-based approaches

The holy grail of constraint programming...

The user states the problem by means of constraints
The computer solves it thanks to embeded solvers

...faced to the reality of NP-hardness

The user often has to help the computer:

- Choose the most appropriate search paradigm
  - CP when constraints are tight enough to prune efficiently
  - CBLS for looser constraints and/or optimization
- Design the "right" model that leads to an efficient search
  - CP: add redundant constraints
  - CBLS: choose appropriate invariants
- "Program" the search
  - CP: design ordering heuristics
  - CBLS: neighborhoods and strategies for escaping local optima

Generic *versus* dedicated approaches

Dedicated approaches (operation research)

Design a customized algorithm to solve a problem: very efficient... but cannot be used to solve a slightly different problem

Constraint-based approaches

Design a solver to solve all CSPs: very generic... but not always efficient (unless the user helps the solver !)

Generic *versus* dedicated approaches

Dedicated approaches (operation research)

Design a customized algorithm to solve a problem: very efficient... but cannot be used to solve a slightly different problem

Modeling languages

Focus on an application domain and design

- a high level modeling language for this domain
- a synthesizer that generates an appropriate solver from the model

Constraint-based approaches

Design a solver to solve all CSPs: very generic... but not always efficient (unless the user helps the solver !)

Our goal

Bridge the gap between high-level modeling and efficient solving:

High-level modeling

- High level objects and constraints
- Related to an application domain
  (graph matching, scheduling, routing, line balancing, ...)

$\rightsquigarrow$ declarative modeling of problems within this domain

Efficient solving

Synthesize the appropriate search strategy:

- analyze the structure of the model
- automatically generate a customized solver
  $\rightsquigarrow$ reuse state-of-the-art approaches, combine them, ...

Characteristics of our approach

Written in Comet

- Supports both CP, CBLS, and MIP
- Object-Oriented

Easy to use as a black-box

- Easy modeling of classical problems
- May be used to model new problems

  $\rightsquigarrow$ Handling specificities through additional constraints

The box may be opened and is easily extensible

- Add new constraints
- Add new solving algorithms, heuristics

  $\rightsquigarrow$ state-of-the-art

$\rightsquigarrow$ extend the synthesizer

Outline

Graph matching problems

Why matching graphs ?

- Many applications require to measure object similarity
  $\rightsquigarrow$ Classification, Search by example, Case-based Reasoning, ...
- Graphs are often used to model objects
  $\rightsquigarrow$ Images, Molecules, Documents, Design objects, ...
- Graph similarity is measured by matching their vertices

What is a matching ?

A matching of $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ is a relation $m \subseteq V_1 \times V_2$
$\rightsquigarrow (u_1, u_2) \in m \Rightarrow$ vertex u_1 is matched to vertex u_2

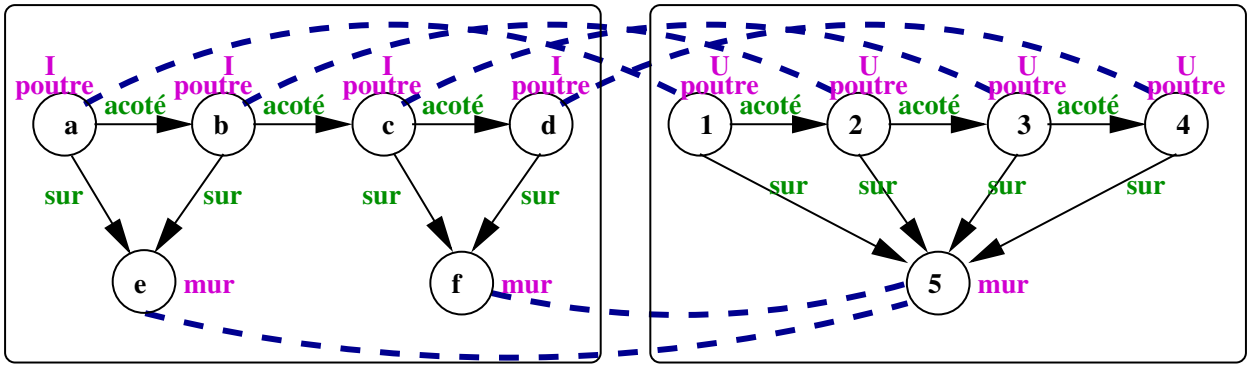
Graph matching problems

Why matching graphs ?

- Many applications require to measure object similarity
  $\rightsquigarrow$ Classification, Search by example, Case-based Reasoning, ...
- Graphs are often used to model objects
  $\rightsquigarrow$ Images, Molecules, Documents, Design objects, ...
- Graph similarity is measured by matching their vertices

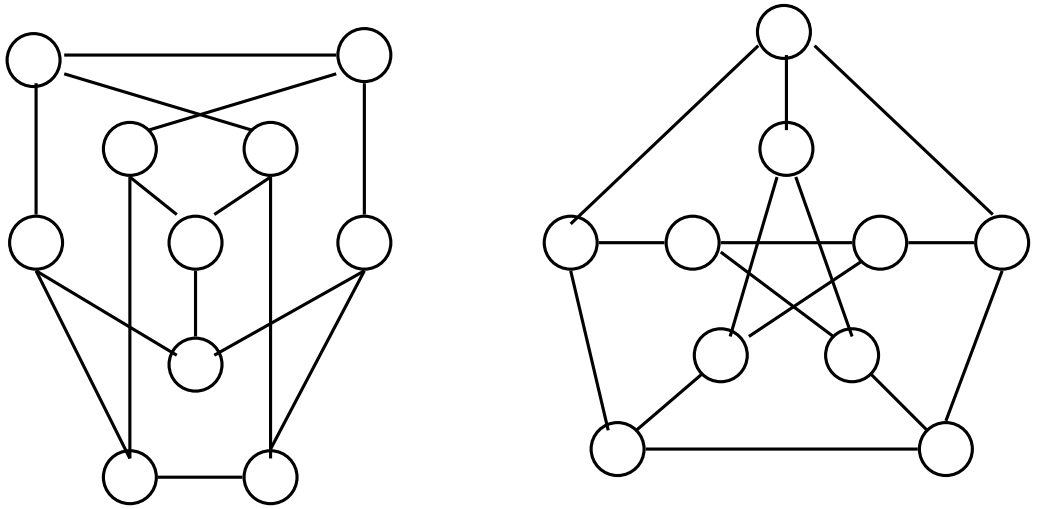
What is a matching ?

A matching of $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ is a relation $m \subseteq V_1 \times V_2$
$\rightsquigarrow (u_1, u_2) \in m \Rightarrow$ vertex u_1 is matched to vertex u_2



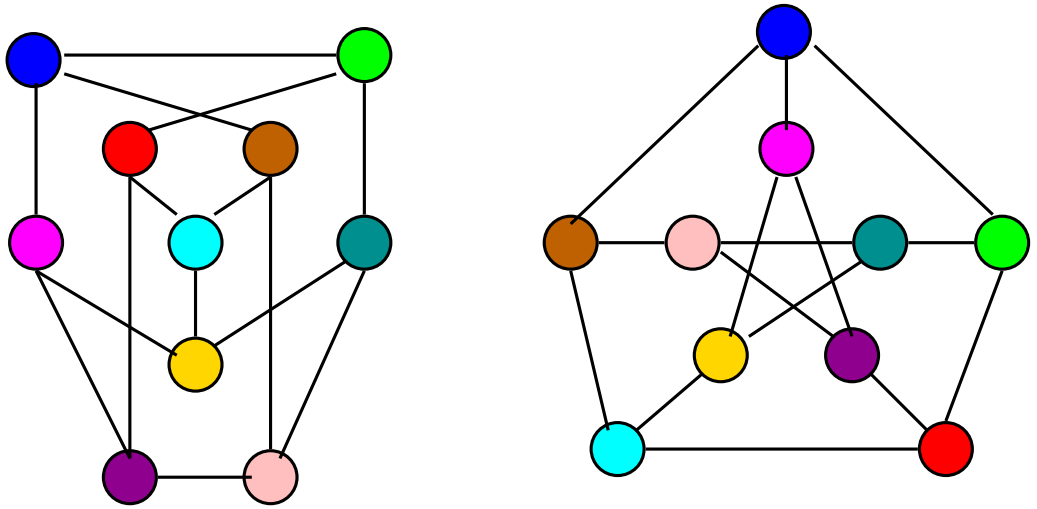
Well known examples of graph matching problems

- Graph Isomorphism $\rightsquigarrow$ decide equivalence
- Subgraph Isomorphism $\rightsquigarrow$ decide inclusion
- Maximum common subgraph $\rightsquigarrow$ Intersection
- Graph Edit Distance $\rightsquigarrow$ Best univalent matching
- Extended Graph Edit Distance $\rightsquigarrow$ Best multivalent matching



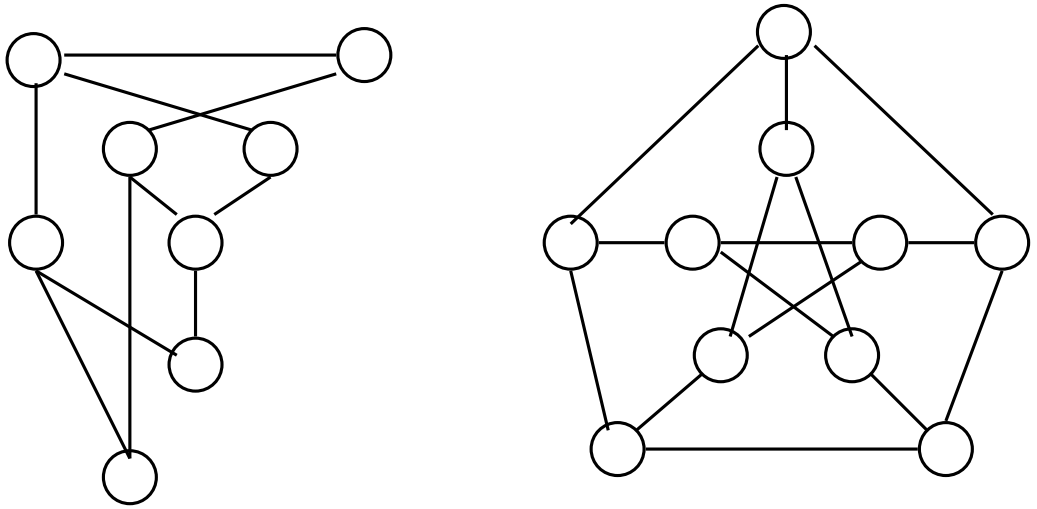
Well known examples of graph matching problems

- Graph Isomorphism $\rightsquigarrow$ decide equivalence
- Subgraph Isomorphism $\rightsquigarrow$ decide inclusion
- Maximum common subgraph $\rightsquigarrow$ Intersection
- Graph Edit Distance $\rightsquigarrow$ Best univalent matching
- Extended Graph Edit Distance $\rightsquigarrow$ Best multivalent matching



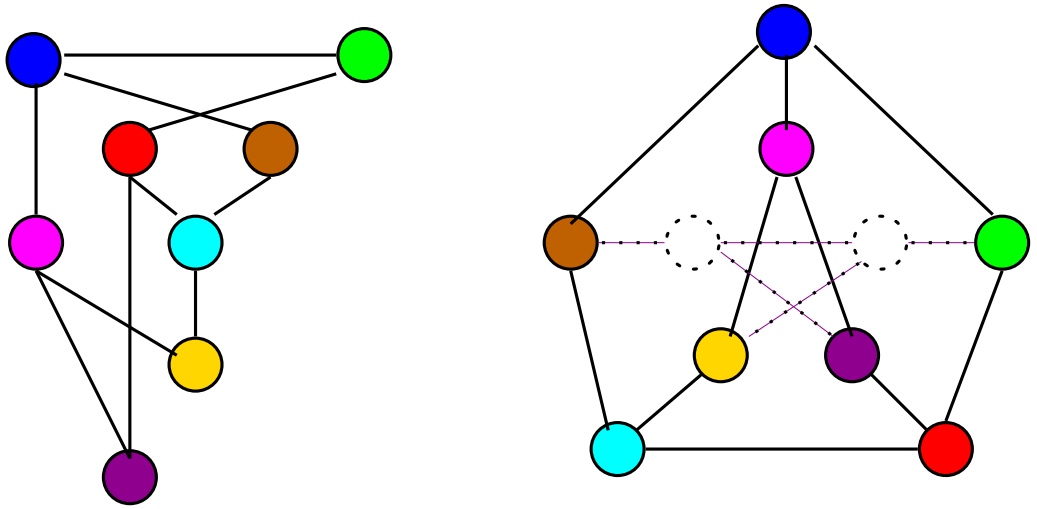
Well known examples of graph matching problems

- Graph Isomorphism $\rightsquigarrow$ decide equivalence
- Subgraph Isomorphism $\rightsquigarrow$ decide inclusion
- Maximum common subgraph $\rightsquigarrow$ Intersection
- Graph Edit Distance $\rightsquigarrow$ Best univalent matching
- Extended Graph Edit Distance $\rightsquigarrow$ Best multivalent matching



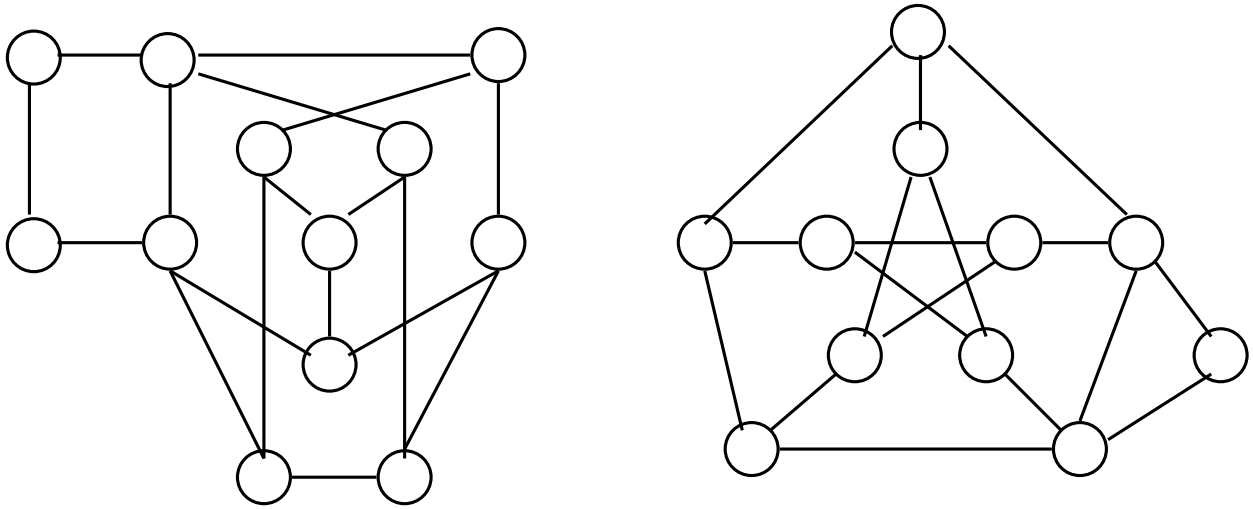
Well known examples of graph matching problems

- Graph Isomorphism $\rightsquigarrow$ decide equivalence
- Subgraph Isomorphism $\rightsquigarrow$ decide inclusion
- Maximum common subgraph $\rightsquigarrow$ Intersection
- Graph Edit Distance $\rightsquigarrow$ Best univalent matching
- Extended Graph Edit Distance $\rightsquigarrow$ Best multivalent matching



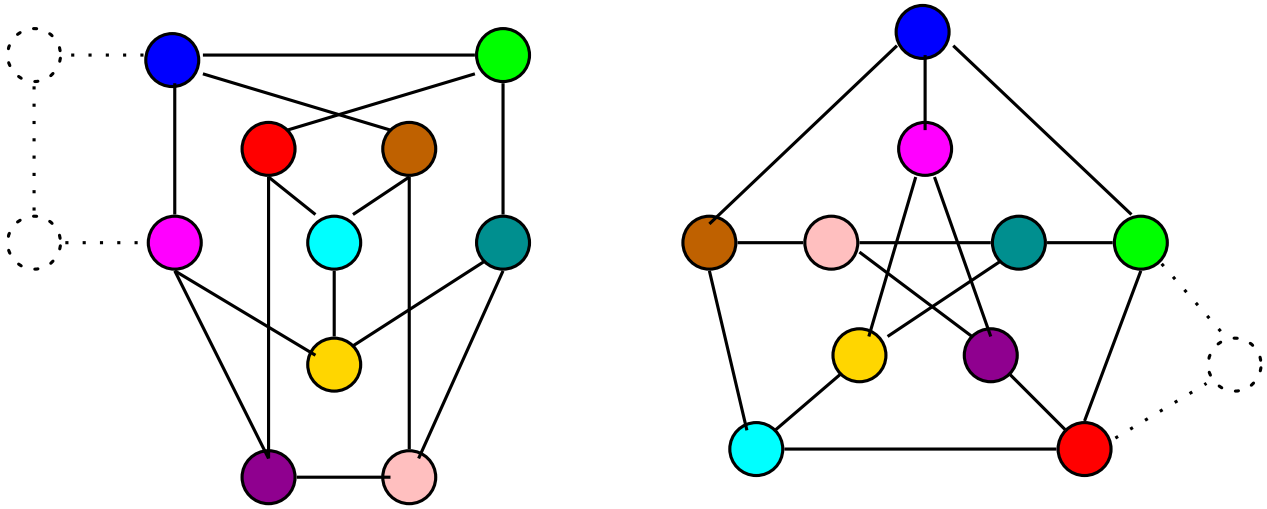
Well known examples of graph matching problems

- Graph Isomorphism $\rightsquigarrow$ decide equivalence
- Subgraph Isomorphism $\rightsquigarrow$ decide inclusion
- Maximum common subgraph $\rightsquigarrow$ Intersection
- Graph Edit Distance $\rightsquigarrow$ Best univalent matching
- Extended Graph Edit Distance $\rightsquigarrow$ Best multivalent matching



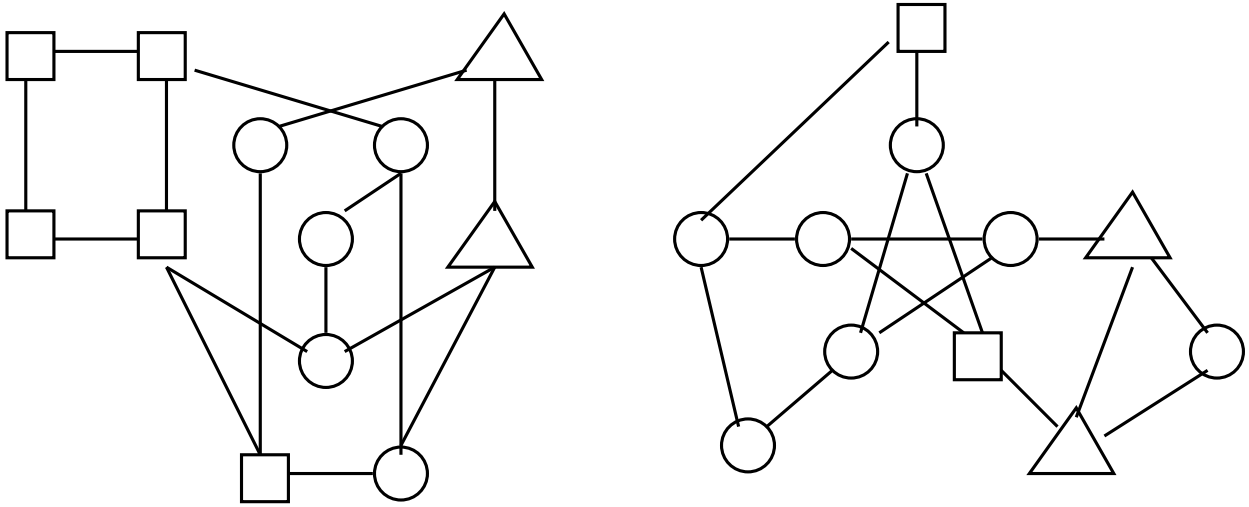
Well known examples of graph matching problems

- Graph Isomorphism $\rightsquigarrow$ decide equivalence
- Subgraph Isomorphism $\rightsquigarrow$ decide inclusion
- Maximum common subgraph $\rightsquigarrow$ Intersection
- Graph Edit Distance $\rightsquigarrow$ Best univalent matching
- Extended Graph Edit Distance $\rightsquigarrow$ Best multivalent matching



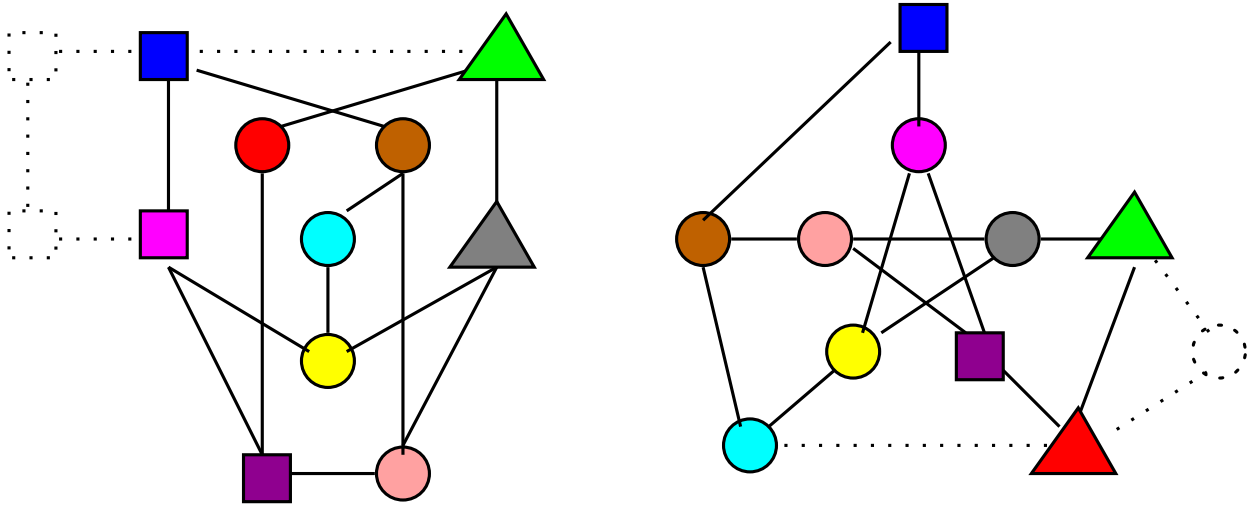
Well known examples of graph matching problems

- Graph Isomorphism $\rightsquigarrow$ decide equivalence
- Subgraph Isomorphism $\rightsquigarrow$ decide inclusion
- Maximum common subgraph $\rightsquigarrow$ Intersection
- Graph Edit Distance $\rightsquigarrow$ Best univalent matching
- Extended Graph Edit Distance $\rightsquigarrow$ Best multivalent matching



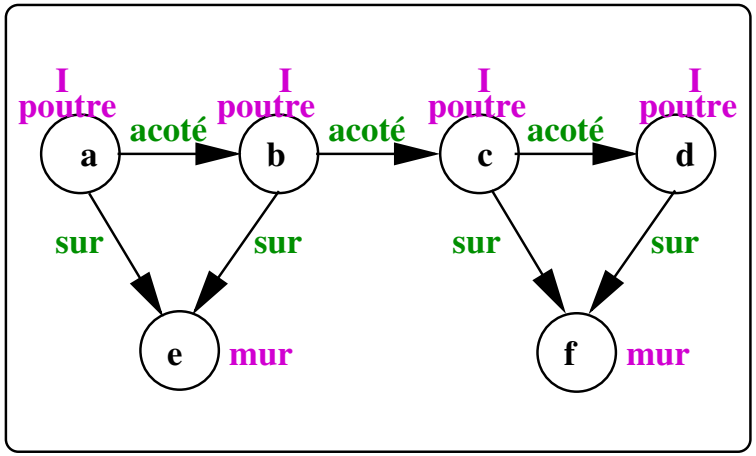
Well known examples of graph matching problems

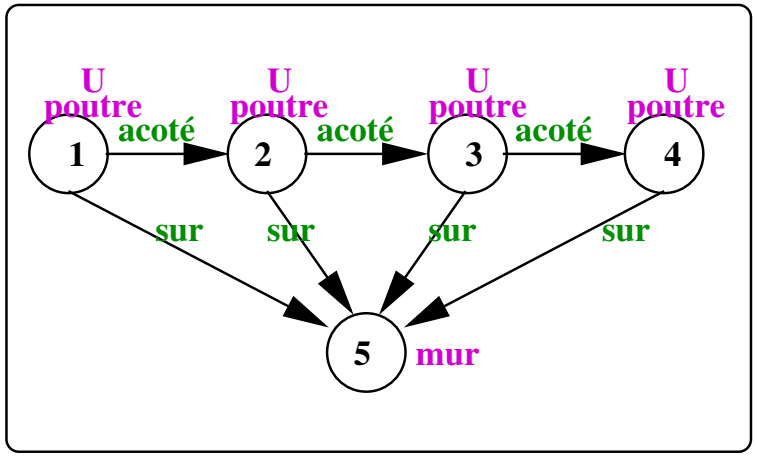
- Graph Isomorphism $\rightsquigarrow$ decide equivalence
- Subgraph Isomorphism $\rightsquigarrow$ decide inclusion
- Maximum common subgraph $\rightsquigarrow$ Intersection
- Graph Edit Distance $\rightsquigarrow$ Best univalent matching
- Extended Graph Edit Distance $\rightsquigarrow$ Best multivalent matching



Well known examples of graph matching problems

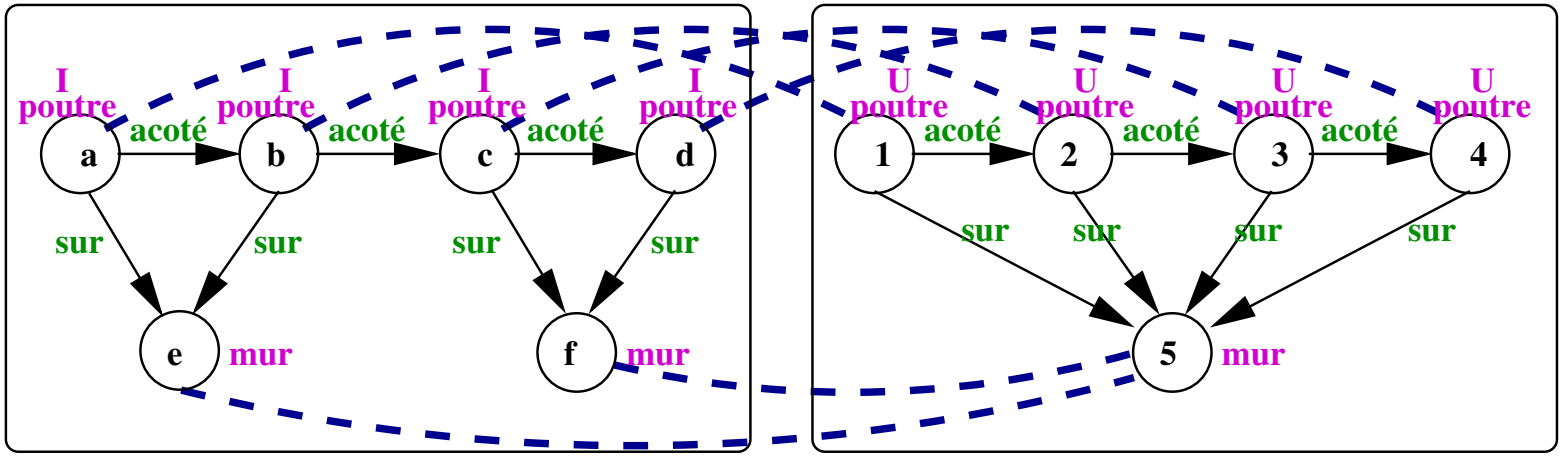
- Graph Isomorphism $\rightsquigarrow$ decide equivalence
- Subgraph Isomorphism $\rightsquigarrow$ decide inclusion
- Maximum common subgraph $\rightsquigarrow$ Intersection
- Graph Edit Distance $\rightsquigarrow$ Best univalent matching
- Extended Graph Edit Distance $\rightsquigarrow$ Best multivalent matching





Well known examples of graph matching problems

- Graph Isomorphism $\rightsquigarrow$ decide equivalence
- Subgraph Isomorphism $\rightsquigarrow$ decide inclusion
- Maximum common subgraph $\rightsquigarrow$ Intersection
- Graph Edit Distance $\rightsquigarrow$ Best univalent matching
- Extended Graph Edit Distance $\rightsquigarrow$ Best multivalent matching



Outline

Modeling graph matching by means of constraints

Constraints on the cardinality of the matching

bijective (1,1), injective (1,0..1), univalent (0..1,0..1), or multivalent (0..n,0..n)

- hard constraints: exact matchings
- soft constraints: error-tolerant matchings

Constraints on edges

- hard constraints: edges must be matched
- soft constraints: maximize the number of matched edges

Constraints on labels (in case of labeled graphs))

- hard constraints: matched components must have identical labels
- soft constraints: maximize the similarity of matched component labels

Example 1: Graph isomorphism

- Declare 2 graph objects g1 and g2 and a matching m

```
bool[,] adj1 = ...
bool[,] adj2 = ...
SimpleGraph<Mod> g1(adj1);
SimpleGraph<Mod> g2(adj2);
Matching<Mod> m(g1,g2);
```

- Post cardinality constraints on m $\rightsquigarrow$ bijective matching $(1, 1)$

```
m.post(cardMatch(g1.getAllNodes(), 1, 1));
m.post(cardMatch(g2.getAllNodes(), 1, 1));
```

- Post constraints to ensure edge matching

```
m.post(matchedToSomeEdges(g1.getAllEdges()));
m.post(matchedToSomeEdges(g2.getAllEdges()));
```

- Ask the synthesizer to create the solver... and search a solution

```
m.close();
DefaultGMSynthesizer synth();
GMSolution<Mod> sol = synth.solveMatching(m);
```

Example 1: Graph isomorphism

- Declare 2 graph objects g1 and g2 and a matching m

```
bool[,] adj1 = ...
bool[,] adj2 = ...
SimpleGraph<Mod> g1(adj1);
SimpleGraph<Mod> g2(adj2);
Matching<Mod> m(g1,g2);
```

- Post cardinality constraints on m $\rightsquigarrow$ bijective matching $(1, 1)$

```
m.post(cardMatch(g1.getAllNodes(), 1, 1));
m.post(cardMatch(g2.getAllNodes(), 1, 1));
```

- Post constraints to ensure edge matching

```
m.post(matchedToSomeEdges(g1.getAllEdges()));
m.post(matchedToSomeEdges(g2.getAllEdges()));
```

- Ask the synthesizer to create the solver... and search a solution

```
m.close();
DefaultGMSynthesizer synth();
GMSolution<Mod> sol = synth.solveMatching(m);
```

Example 1: Graph isomorphism

- Declare 2 graph objects g1 and g2 and a matching m

```
bool[,] adj1 = ...
bool[,] adj2 = ...
SimpleGraph<Mod> g1(adj1);
SimpleGraph<Mod> g2(adj2);
Matching<Mod> m(g1,g2);
```

- Post cardinality constraints on m $\rightsquigarrow$ bijective matching $(1, 1)$

```
m.post(cardMatch(g1.getAllNodes(), 1, 1));
m.post(cardMatch(g2.getAllNodes(), 1, 1));
```

- Post constraints to ensure edge matching

```
m.post(matchedToSomeEdges(g1.getAllEdges()));
m.post(matchedToSomeEdges(g2.getAllEdges()));
```

- Ask the synthesizer to create the solver... and search a solution

```
m.close();
DefaultGMSynthesizer synth();
GMSolution<Mod> sol = synth.solveMatching(m);
```

Example 1: Graph isomorphism

- Declare 2 graph objects g1 and g2 and a matching m

```
bool[,] adj1 = ...
bool[,] adj2 = ...
SimpleGraph<Mod> g1(adj1);
SimpleGraph<Mod> g2(adj2);
Matching<Mod> m(g1,g2);
```

- Post cardinality constraints on m $\rightsquigarrow$ bijective matching $(1, 1)$

```
m.post(cardMatch(g1.getAllNodes(), 1, 1));
m.post(cardMatch(g2.getAllNodes(), 1, 1));
```

- Post constraints to ensure edge matching

```
m.post(matchedToSomeEdges(g1.getAllEdges()));
m.post(matchedToSomeEdges(g2.getAllEdges()));
```

- Ask the synthesizer to create the solver... and search a solution

```
m.close();
DefaultGMSynthesizer synth();
GMSolution<Mod> sol = synth.solveMatching(m);
```

Example 2: Induced Subgraph Isomorphism

- Declare 2 graph objects g1 and g2 and a matching m

```
bool[,] adj1 = ...
bool[,] adj2 = ...
SimpleGraph<Mod> g1(adj1);
SimpleGraph<Mod> g2(adj2);
Matching<Mod> m(g1,g2);
```

- Post cardinality constraints on m $\rightsquigarrow$ injective matching $(1, 0..1)$

```
m.post(cardMatch(g1.getAllNodes(), 1, 1));
m.post(cardMatch(g2.getAllNodes(), 0, 1));
```

- Post constraints to ensure edges of G_1 to be matched

```
m.post(matchedToSomeEdges(g1.getAllEdges()));
```

- Ask the synthesizer to create the solver... and search a solution

```
m.close();
DefaultGMSynthesizer synth();
GMSolution<Mod> sol = synth.solveMatching(m);
```

Example 2: Induced Subgraph Isomorphism

- Declare 2 graph objects g1 and g2 and a matching m

```
bool[,] adj1 = ...
bool[,] adj2 = ...
SimpleGraph<Mod> g1(adj1);
SimpleGraph<Mod> g2(adj2);
Matching<Mod> m(g1,g2);
```

- Post cardinality constraints on m $\rightsquigarrow$ injective matching $(1, 0..1)$

```
m.post(cardMatch(g1.getAllNodes(), 1, 1));
m.post(cardMatch(g2.getAllNodes(), 0, 1));
```

- Post constraints to ensure edges of G_1 to be matched

```
m.post(matchedToSomeEdges(g1.getAllEdges()));
```

- Ask the synthesizer to create the solver... and search a solution

```
m.close();
DefaultGMSynthesizer synth();
GMSolution<Mod> sol = synth.solveMatching(m);
```

Example 2: Induced Subgraph Isomorphism

- Declare 2 graph objects g1 and g2 and a matching m

```
bool[,] adj1 = ...
bool[,] adj2 = ...
SimpleGraph<Mod> g1(adj1);
SimpleGraph<Mod> g2(adj2);
Matching<Mod> m(g1,g2);
```

- Post cardinality constraints on m $\rightsquigarrow$ injective matching $(1, 0..1)$

```
m.post(cardMatch(g1.getAllNodes(), 1, 1));
m.post(cardMatch(g2.getAllNodes(), 0, 1));
```

- Post constraints to ensure edges of G_1 to be matched

```
m.post(matchedToSomeEdges(g1.getAllEdges()));
```

- Ask the synthesizer to create the solver... and search a solution

```
m.close();
DefaultGMSynthesizer synth();
GMSolution<Mod> sol = synth.solveMatching(m);
```

Example 2: Induced Subgraph Isomorphism

- Declare 2 graph objects g1 and g2 and a matching m

```
bool[,] adj1 = ...
bool[,] adj2 = ...
SimpleGraph<Mod> g1(adj1);
SimpleGraph<Mod> g2(adj2);
Matching<Mod> m(g1,g2);
```

- Post cardinality constraints on m $\rightsquigarrow$ injective matching $(1, 0..1)$

```
m.post(cardMatch(g1.getAllNodes(), 1, 1));
m.post(cardMatch(g2.getAllNodes(), 0, 1));
```

- Post constraints to ensure edges of G_1 to be matched

```
m.post(matchedToSomeEdges(g1.getAllEdges()));
```

- Ask the synthesizer to create the solver... and search a solution

```
m.close();
DefaultGMSynthesizer synth();
GMSolution<Mod> sol = synth.solveMatching(m);
```

Example 3: Largest Common Induced Subgraph

- Declare 2 graph objects g1 and g2 and a matching m

```
bool[,] adj1 = ...
bool[,] adj2 = ...
SimpleGraph<Mod> g1(adj1);
SimpleGraph<Mod> g2(adj2);
Matching<Mod> m(g1,g2);
```

- Post cardinality constraints on m $\rightsquigarrow$ univalent matching $(0..1, 0..1)$

```
m.post(cardMatch(g1.getAllNodes(), 0, 1));
m.post(cardMatch(g2.getAllNodes(), 0, 1));
```

- Post a soft constraint to maximize the number of matched vertices

```
m.softpost(minMatch(g1.getAllNodes(), 1), 1)
```

- Post constraints to ensure edge matching

```
m.post(matchedToAllEdges(g1.getAllEdges()));
m.post(matchedToAllEdges(g2.getAllEdges()));
```

- Ask the synthesizer to create the solver... and search a solution

```
m.close(); DefaultGMSynthesizer synth();
GMSolution<Mod> sol = synth.solveMatching(m);
```

Example 3: Largest Common Induced Subgraph

- Declare 2 graph objects g1 and g2 and a matching m

```
bool[,] adj1 = ...
bool[,] adj2 = ...
SimpleGraph<Mod> g1(adj1);
SimpleGraph<Mod> g2(adj2);
Matching<Mod> m(g1,g2);
```

- Post cardinality constraints on m $\rightsquigarrow$ univalent matching $(0..1, 0..1)$

```
m.post(cardMatch(g1.getAllNodes(), 0, 1));
m.post(cardMatch(g2.getAllNodes(), 0, 1));
```

- Post a soft constraint to maximize the number of matched vertices

```
m.softpost(minMatch(g1.getAllNodes(), 1), 1)
```

- Post constraints to ensure edge matching

```
m.post(matchedToAllEdges(g1.getAllEdges()));
m.post(matchedToAllEdges(g2.getAllEdges()));
```

- Ask the synthesizer to create the solver... and search a solution

```
m.close(); DefaultGMSynthesizer synth();
GMSolution<Mod> sol = synth.solveMatching(m);
```

Example 3: Largest Common Induced Subgraph

- Declare 2 graph objects `g1` and `g2` and a matching `m`

```
bool[,] adj1 = ...
bool[,] adj2 = ...
SimpleGraph<Mod> g1(adj1);
SimpleGraph<Mod> g2(adj2);
Matching<Mod> m(g1,g2);
```

- Post cardinality constraints on `m` $\rightsquigarrow$ univalent matching $(0..1, 0..1)$

```
m.post(cardMatch(g1.getAllNodes(), 0, 1));
m.post(cardMatch(g2.getAllNodes(), 0, 1));
```

- Post a soft constraint to maximize the number of matched vertices

```
m.softpost(minMatch(g1.getAllNodes(), 1), 1)
```

- Post constraints to ensure edge matching

```
m.post(matchedToAllEdges(g1.getAllEdges()));
m.post(matchedToAllEdges(g2.getAllEdges()));
```

- Ask the synthesizer to create the solver... and search a solution

```
m.close(); DefaultGMSynthesizer synth();
GMSolution<Mod> sol = synth.solveMatching(m);
```

Example 3: Largest Common Induced Subgraph

- Declare 2 graph objects g1 and g2 and a matching m

```
bool[,] adj1 = ...
bool[,] adj2 = ...
SimpleGraph<Mod> g1(adj1);
SimpleGraph<Mod> g2(adj2);
Matching<Mod> m(g1,g2);
```

- Post cardinality constraints on m $\rightsquigarrow$ univalent matching $(0..1, 0..1)$

```
m.post(cardMatch(g1.getAllNodes(), 0, 1));
m.post(cardMatch(g2.getAllNodes(), 0, 1));
```

- Post a soft constraint to maximize the number of matched vertices

```
m.softpost(minMatch(g1.getAllNodes(), 1), 1)
```

- Post constraints to ensure edge matching

```
m.post(matchedToAllEdges(g1.getAllEdges()));
m.post(matchedToAllEdges(g2.getAllEdges()));
```

- Ask the synthesizer to create the solver... and search a solution

```
m.close(); DefaultGMSynthesizer synth();
GMSolution<Mod> sol = synth.solveMatching(m);
```

Example 3: Largest Common Induced Subgraph

- Declare 2 graph objects `g1` and `g2` and a matching `m`

```
bool[,] adj1 = ...
bool[,] adj2 = ...
SimpleGraph<Mod> g1(adj1);
SimpleGraph<Mod> g2(adj2);
Matching<Mod> m(g1,g2);
```

- Post cardinality constraints on `m` $\rightsquigarrow$ univalent matching $(0..1, 0..1)$

```
m.post(cardMatch(g1.getAllNodes(), 0, 1));
m.post(cardMatch(g2.getAllNodes(), 0, 1));
```

- Post a soft constraint to maximize the number of matched vertices

```
m.softpost(minMatch(g1.getAllNodes(), 1), 1)
```

- Post constraints to ensure edge matching

```
m.post(matchedToAllEdges(g1.getAllEdges()));
m.post(matchedToAllEdges(g2.getAllEdges()));
```

- Ask the synthesizer to create the solver... and search a solution

```
m.close(); DefaultGMSynthesizer synth();
GMSolution<Mod> sol = synth.solveMatching(m);
```

Outline

Synthesizing a solver for graph matching problems (1/3)

Warning: Ongoing research with a very first prototype
$\rightsquigarrow$ many improvements are still to be done !

Canonical form of modeling constraints

Aggregate all modeling constraints of a same type

- Cardinality (MinMatch, MaxMatch, CardMatch, ...)
- Edge matching (MatchedToSomeEdges, MatchedToAllEdges, ...)
- Label matching (MatchAllNodeLabels, MatchAllEdgeLabels, ...)

$\rightsquigarrow$ Derive characteristics from the canonical model

Choose a search approach

- CP if no soft constraints and MaxCard ≤ 1 for all vertices of a graph
  $\rightsquigarrow$ Maintaining Arc Consistency
- CBLS otherwise
  $\rightsquigarrow$ Tabu search

Synthesizing a solver for graph matching problems (2/3)

Creation of low level variables

Associate a variable with every vertex of both graphs

- Domains are defined wrt cardinality constraints

MinMatch	MaxMatch	Type	Domain
1	1	**int**	N
0	1	**int**	$N \cup \{\perp\}$
Otherwise		set	2^N

- Ensure symmetry (X_u matched to $v \Rightarrow X_v$ matched to u):
  - CP $\rightsquigarrow$ Channeling constraints
  - CBLS $\rightsquigarrow$ invariants

Synthesizing a solver for graph matching problems (3/3)

Post the canonical constraints

- CP (hard constraints only)
  - Cardinality constraints
    $\rightsquigarrow$ Partly handled by variable domains
    $\rightsquigarrow$ Global allDiff for injective and bijective matchings
  - Edge constraints $\rightsquigarrow$ binary constraints
  - Label constraints on nodes $\rightsquigarrow$ variable domains
  - Label constraints on edges $\rightsquigarrow$ binary constraints
- CBLS (hard and soft constraints)
  - Cardinality $\rightsquigarrow$ neighborhood if hard; invariants if soft
  - Edge $\rightsquigarrow$ invariants
  - Node labels $\rightsquigarrow$ neighborhood if hard; invariants if soft
  - Edge labels $\rightsquigarrow$ invariants

Outline

(Preliminary) Experimental Results (1/2)

$\mathcal{SI} \rightsquigarrow$ Subgraph Isomorphism

	#Nodes	Synthesizer/CP					vf2 [Cordella et al. 99]				
		5%	10%	20%	33%	50%	5%	10%	20%	33%	50%
$\mathcal{SI}$	100	0.8	0.5	0.7	0.1	0.2	0.0	0.0	0.0	2.0	0.0
	500	19.3	4.7	10.5	15.8	30.7	0.1	0.1	246.7	192.3	–
	1000	30.6	595.8	119.0	152.3	–	86.7	–	–	–	–

- Vf2 better for small instances
- Synthesizer outperforms vf2 for larger instances
- Additional constraint improves the search process

(Preliminary) Experimental Results (1/2)

$\mathcal{SI} \rightsquigarrow$ Subgraph Isomorphism
$\mathcal{SI}+ \rightsquigarrow$ Subgraph Isomorphism + additional distance constraint

	#Nodes	Synthesizer/CP					vf2 [Cordella et al. 99]				
		5%	10%	20%	33%	50%	5%	10%	20%	33%	50%
$\mathcal{SI}$	100	0.8	0.5	0.7	0.1	0.2	0.0	0.0	0.0	2.0	0.0
	500	19.3	4.7	10.5	15.8	30.7	0.1	0.1	246.7	192.3	–
	1000	30.6	595.8	119.0	152.3	–	86.7	–	–	–	–
$\mathcal{SI}+$	100	0.3	0.1	0.1	0.1	0.2					
	500	3.0	4.4	9.5	16.9	28.9					
	1000	16.1	47.8	82.5	148.0	–					

- Vf2 better for small instances
- Synthesizer outperforms vf2 for larger instances
- Additional constraint improves the search process

(Preliminary) Experimental Results (2/2)

Maximum common subgraph $\rightsquigarrow$ CBLS

#nodes	time		iterations		edges%	
25	8.5	(2.5)	7768.1	(2301.3)	48.3	(1.1)
50	33.9	(10.7)	8023.8	(2543.3)	40.2	(0.5)
100	141.5	(46.4)	8398.4	(2755.0)	34.5	(0.2)

- First results to assess feasibility
- Complete approaches cannot handle these instances
- We haven't (yet) compared these results with other approaches

Further works on modeling for graph matching

- Improve the analysis of the matching characteristics
  $\rightsquigarrow$ identify sub-problems that are "easy" to solve
- Integrate dedicated filtering algorithms $\rightsquigarrow$ CP
  - Iterative partitionning for graph isomorphism (Nauty)
  - Iterative labeling for subgraph iso. (Zampelli et al 2009)
- Integrate reactive search and other meta-heuristics for CBLS
  $\rightsquigarrow$ Parameter tuning... !
- Combine CP and CBLS

Outline

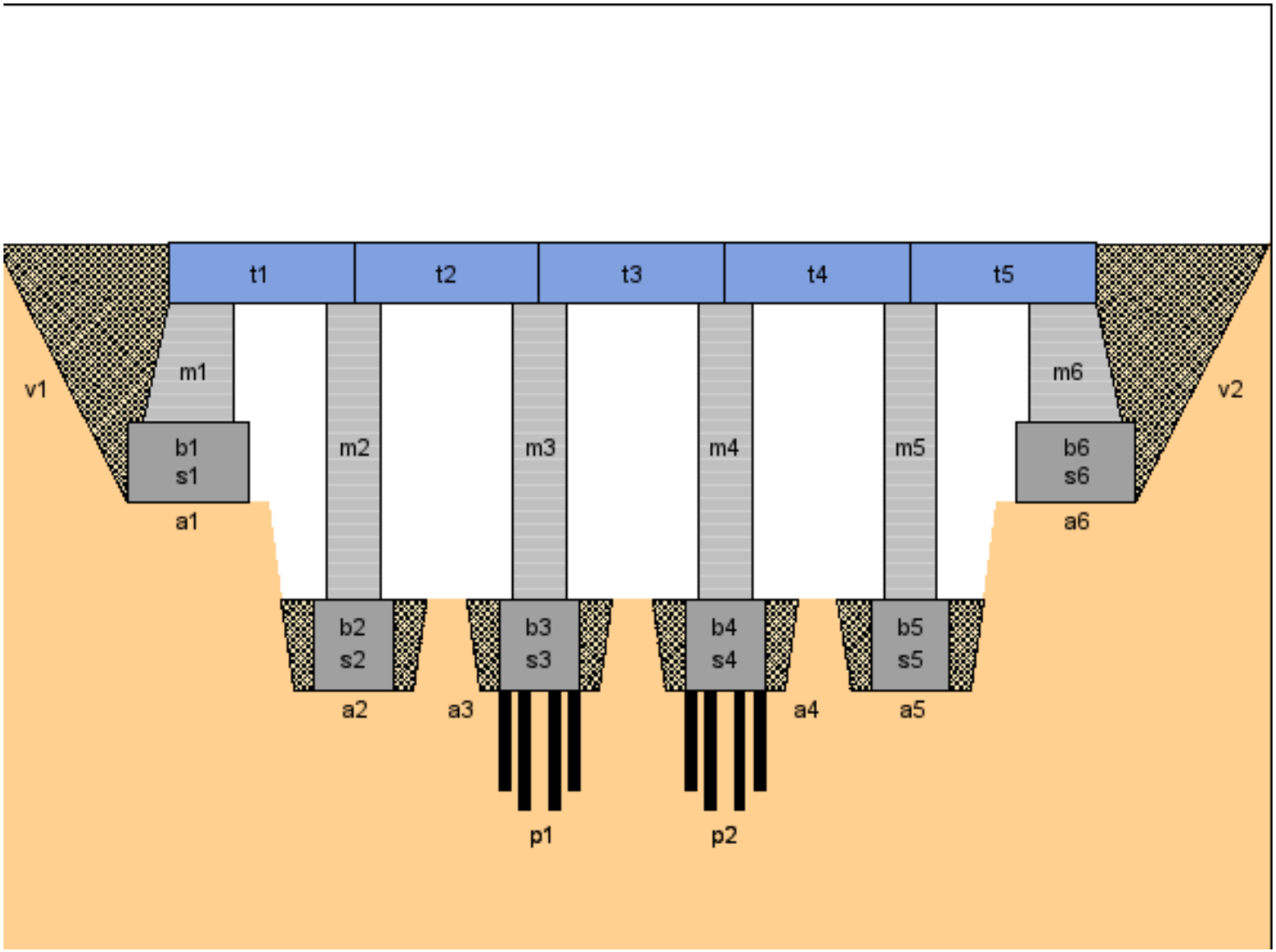
Scheduling

The goal is to allocate scarce resources to a set of activities over time.

Scheduling is everywhere

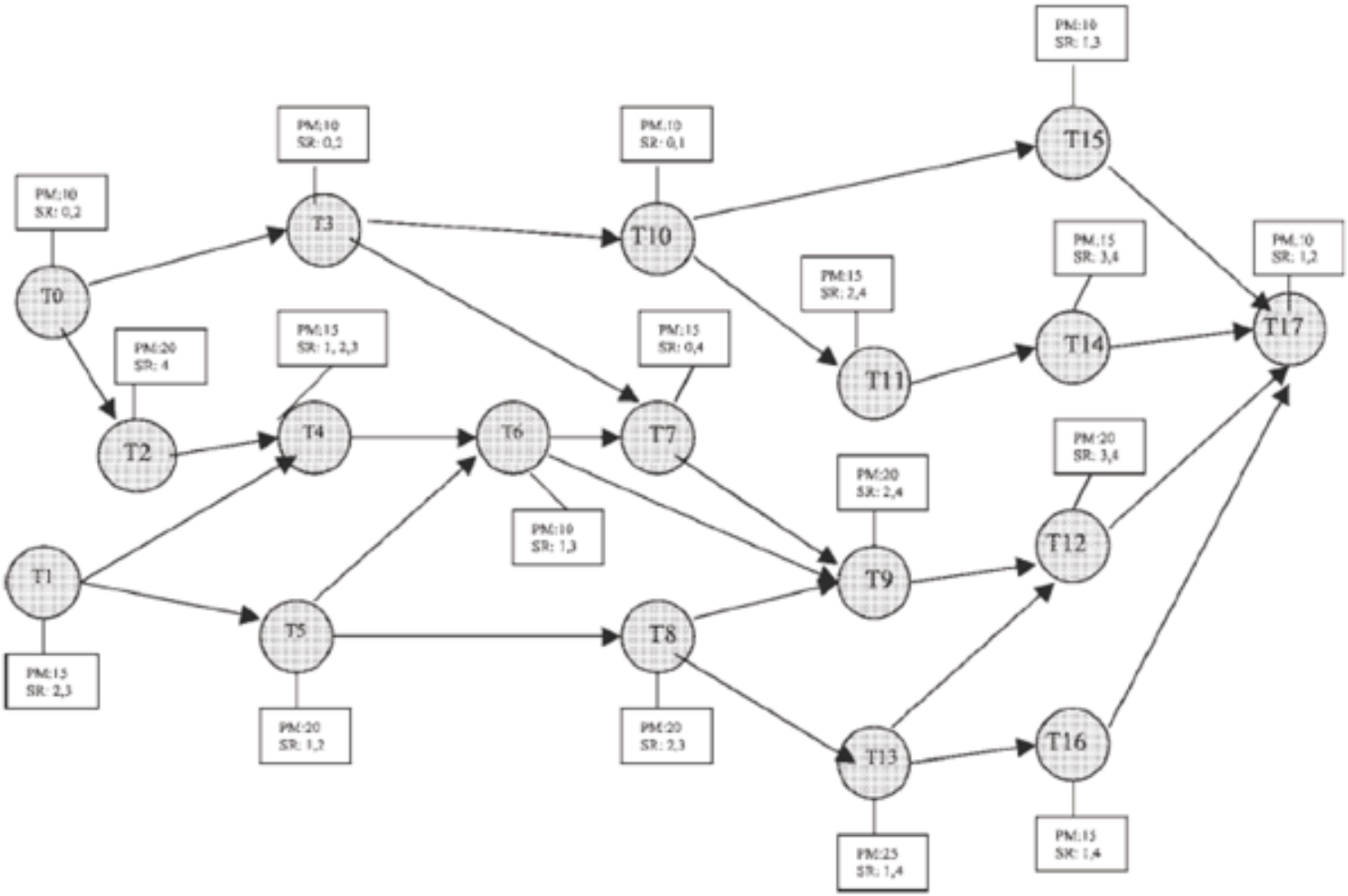
- Products Manufacturing
- Construction Planning
- Code Optimization in Compilers
- Project Management (Pharmaceutic Industry, for instance)
- Trains and Airplanes Scheduling
- Closely Related to Timetabling, Vehicle Routing, Planning...

Construction Scheduling



Airport Scheduling

Project Scheduling



Scheduling

There exists a lot of variations

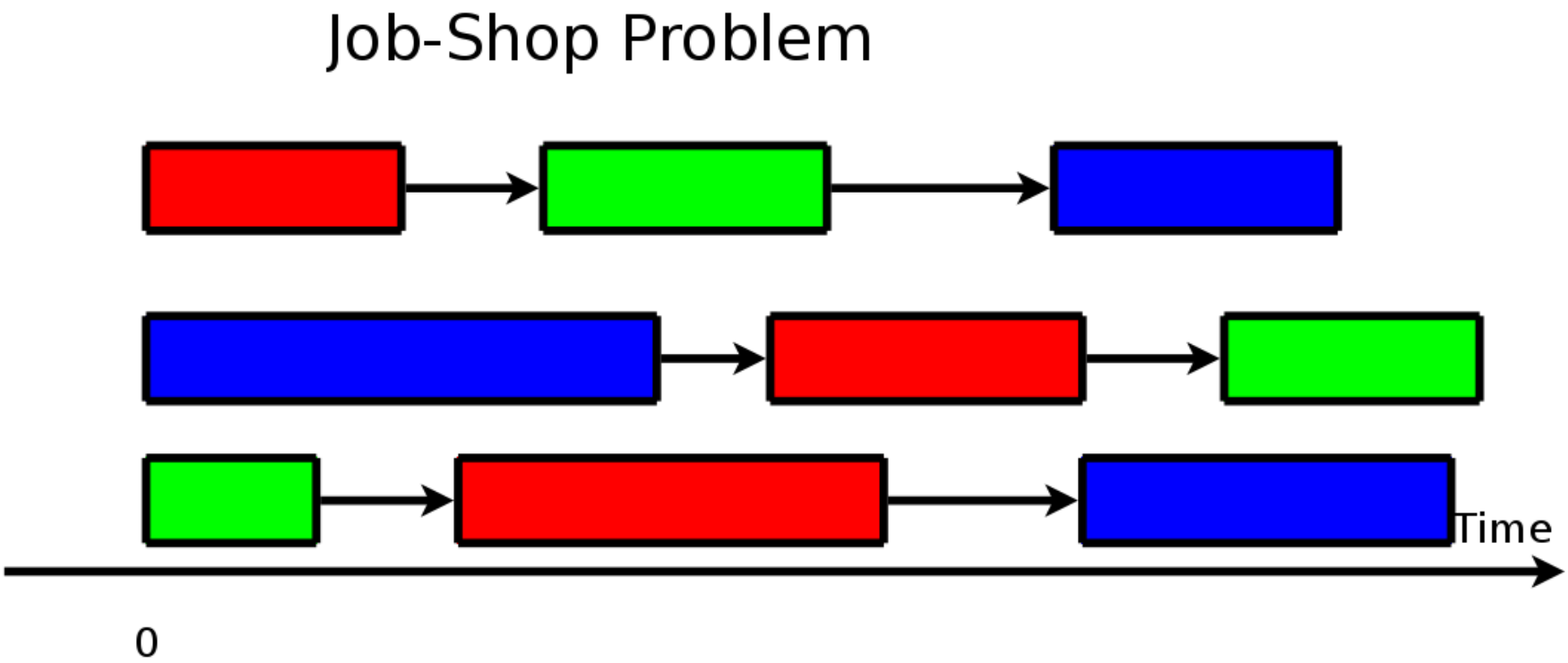
- Models for activities
  - Preemption, Jobs, ...
- Models for resources
  - Cumulative, Machines, Reservoirs, ...
- Constraints
  - Precedences, Max-Slack, ...
- Objective functions
  - Makespan, Sum of Tardiness, ...

Examples of Scheduling Problems

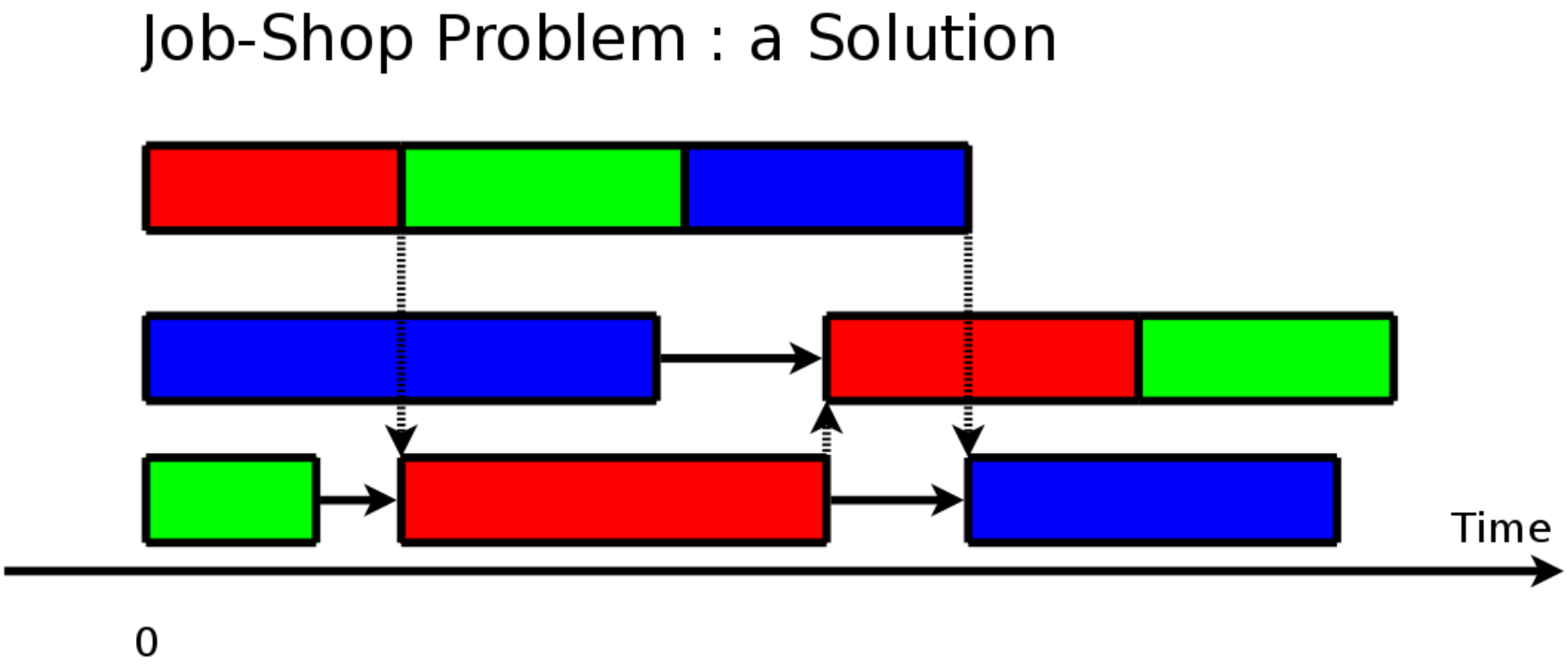
- Job-Shop (Makespan, Tardiness, Earliness-Tardiness)
- Open-Shop
- Cumulative Job-Shop
- RCPSP, RCPSP/max
- MMRCPSP, MMRCPSP/max
- Trolley Problem
- MascLib (NCOS and NCGS classes)

Outline

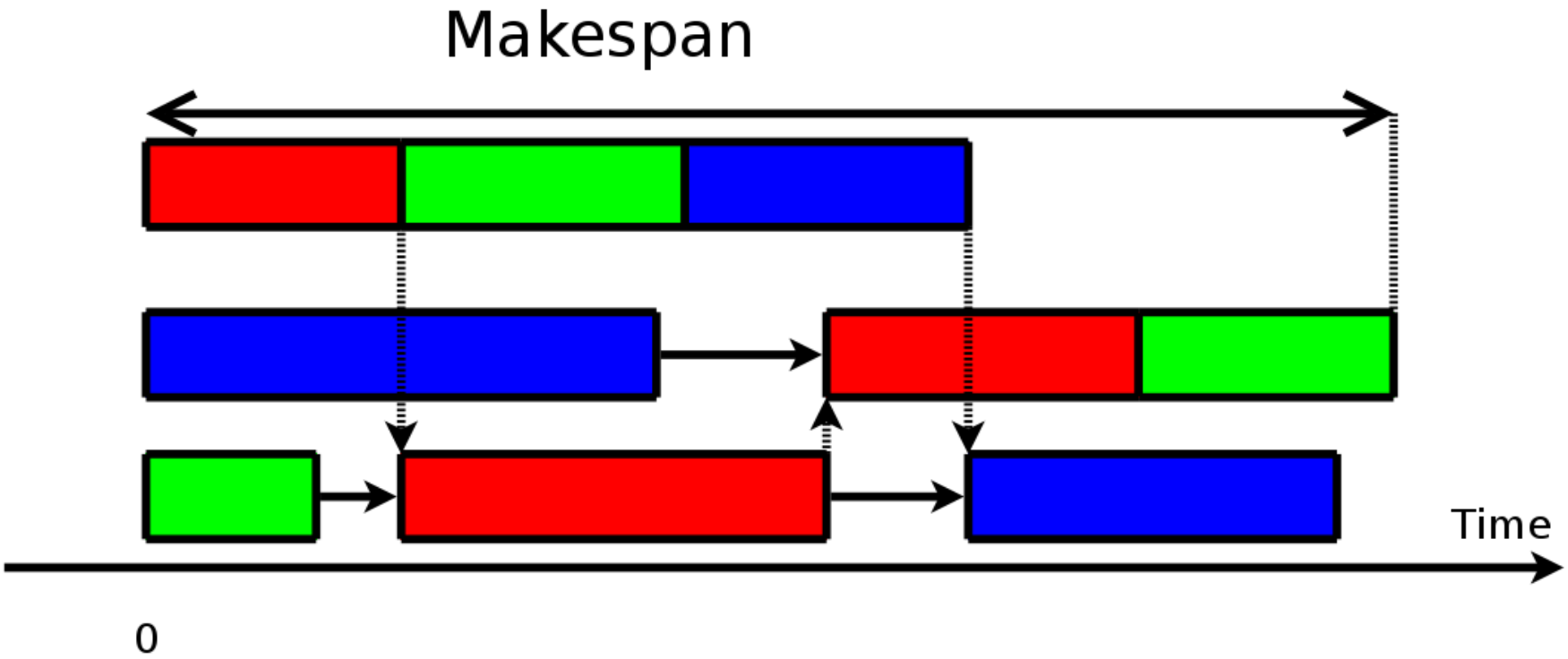
Job-Shop Scheduling Problem



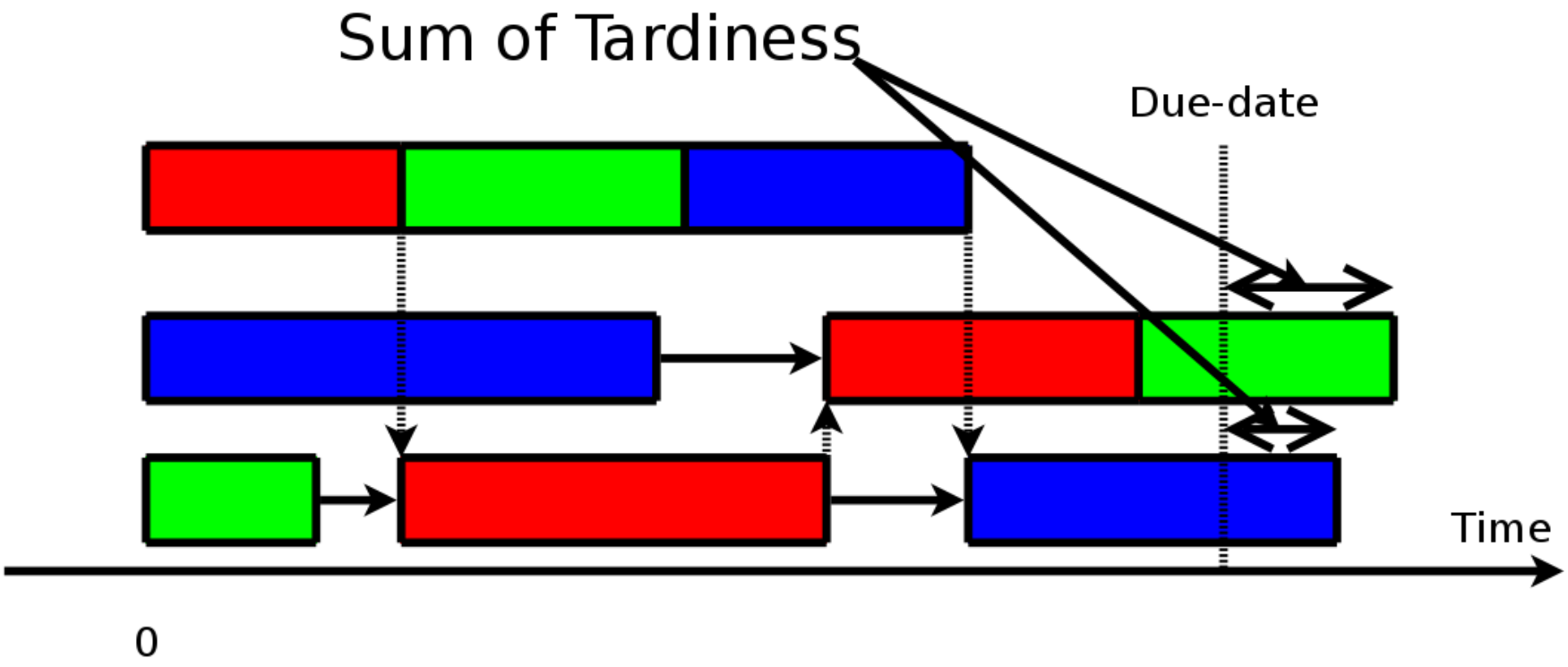
Job-Shop Scheduling Problem



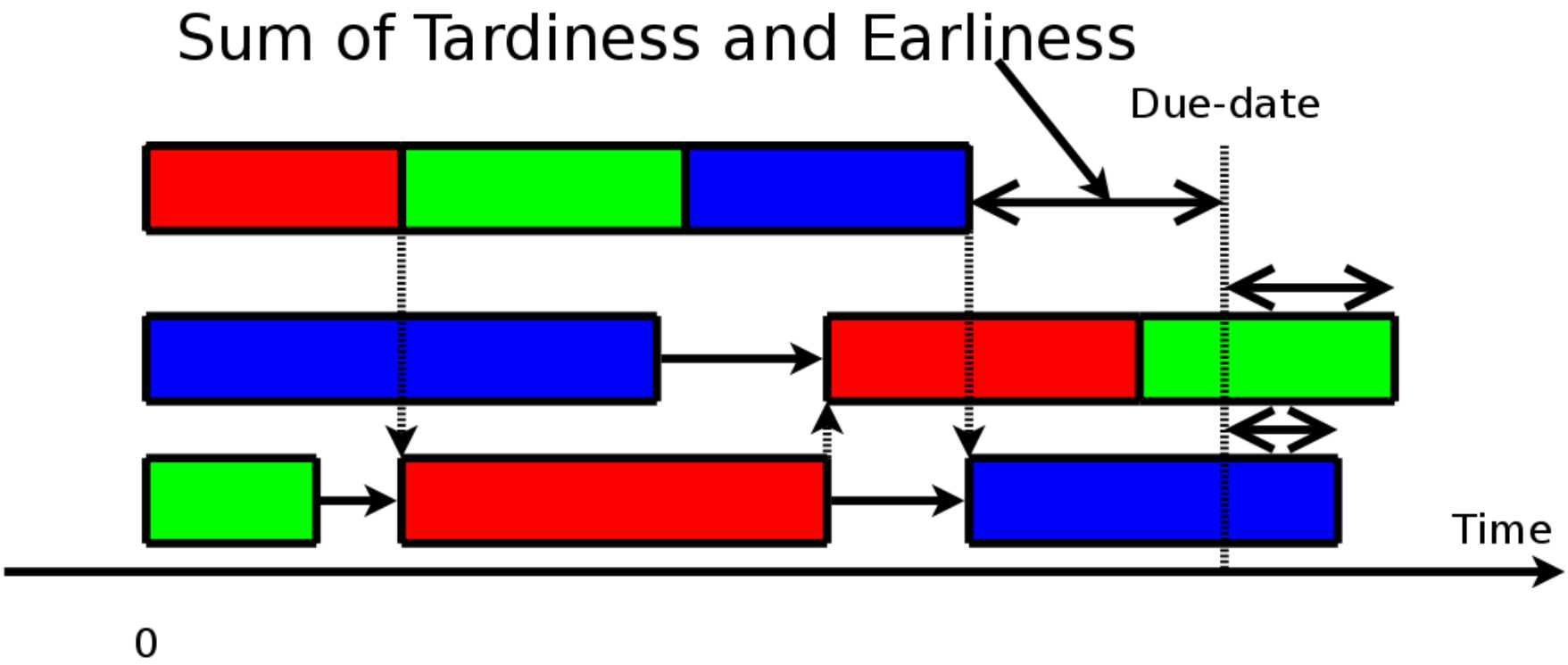
Job-Shop : Objectives



Job-Shop : Objectives



Job-Shop : Objectives



Modeling : Job-Shop

```
range jobs = 1..nbjobs;
range machines = 0..nbmachines-1;
range tasks = 1..nbjobs*nbmachines;
int proc[tasks];
int mach[tasks];
int job[jobs,machines];

Schedule<Mod> s();
Job<Mod> J[i in jobs](s,"J"+i);
Machine<Mod> M[i in machines](s,"M"+i);
Activity<Mod> A[i in tasks](s,proc[i],"A"+i);
forall(i in tasks)
  A[i].requires(M[mach[i]]);
forall(i in jobs)
  J[i].containsInSequence(
          all(j in machines)A[job[i,j]]);
s.minimizeObj(makespanOf(s));
```

Solving

```
GreedyTabuSynthesizer synth();
// CPSynthesizer synth();
Solution<Mod> sol = synth.solve(s);
sol.printSolution();
```

```
Classifier Set-Up
Models[JobShopWithMakespan,CumulativeJobShopWithMakespan,CumulativeJobShop

...

930.000000
Time = 12568
```

Modeling : RCPSP

```
range tasks;
range resources;
int proc[tasks]; //Processing Times
int cap[resources]; //Capacities
int succ[][tasks]; //Successors
int req[tasks,resources]; //Requirements
Schedule<Mod> s();
Activity<Mod> A[i in tasks](s, proc[i], "J");
Resource<Mod> R[i in resources](s, cap[i], "R");
forall(i in tasks){
  forall(j in succ[i].getRange())
    A[i].precedes(A[succ[i][j]]);
  forall(j in resources)
    if(req[i,j]!=0)A[i].requires(R[j],req[i,j]);
}
s.minimizeObj(Tardiness<Mod>(s,A[sink],duedate)*tardCost);
```

Available Abstractions

Description	Classes
Schedule	`Schedule`
Activities	`Activity`
	`MultiModeActivity`
Jobs	`Job`
Resources	`Resource`
	`Machine`
	`Reservoir`
	`StateResource`
Objectives	`ScheduleObjective`
	`CompletionTime, PiecewiseLinearFunction`
	`Tardiness, Earliness, Lateness, UnitCost`
	`AbsenceCost, AlternativeCost`
	`MultObjective, ShiftObjective`
	`SumObjective, MaxObjective`

Outline

Context

- Scheduling is a large domain of research and application for optimization techniques.
- Among the techniques: Constraint Programming, Local Search, Integer Programming, Genetic Algorithms, Greedy Algorithms
- Most algorithms are specific to a restricted class of problems. A lot of parameters must be tuned.
- It may be hard to recognize problems, find the most appropriate algorithm and code it.

Available Synthesizers

Description	Classes
Synthesizers	`ScheduleSynthesizer`
	`CPSynthesizer`
	`TSSynthesizer`
	`SASynthesizer`
	`GreedySynthesizer`
	`SequenceSynthesizer`
	`ScheduleAnimator`
Solutions	`Solution`

Represented problems

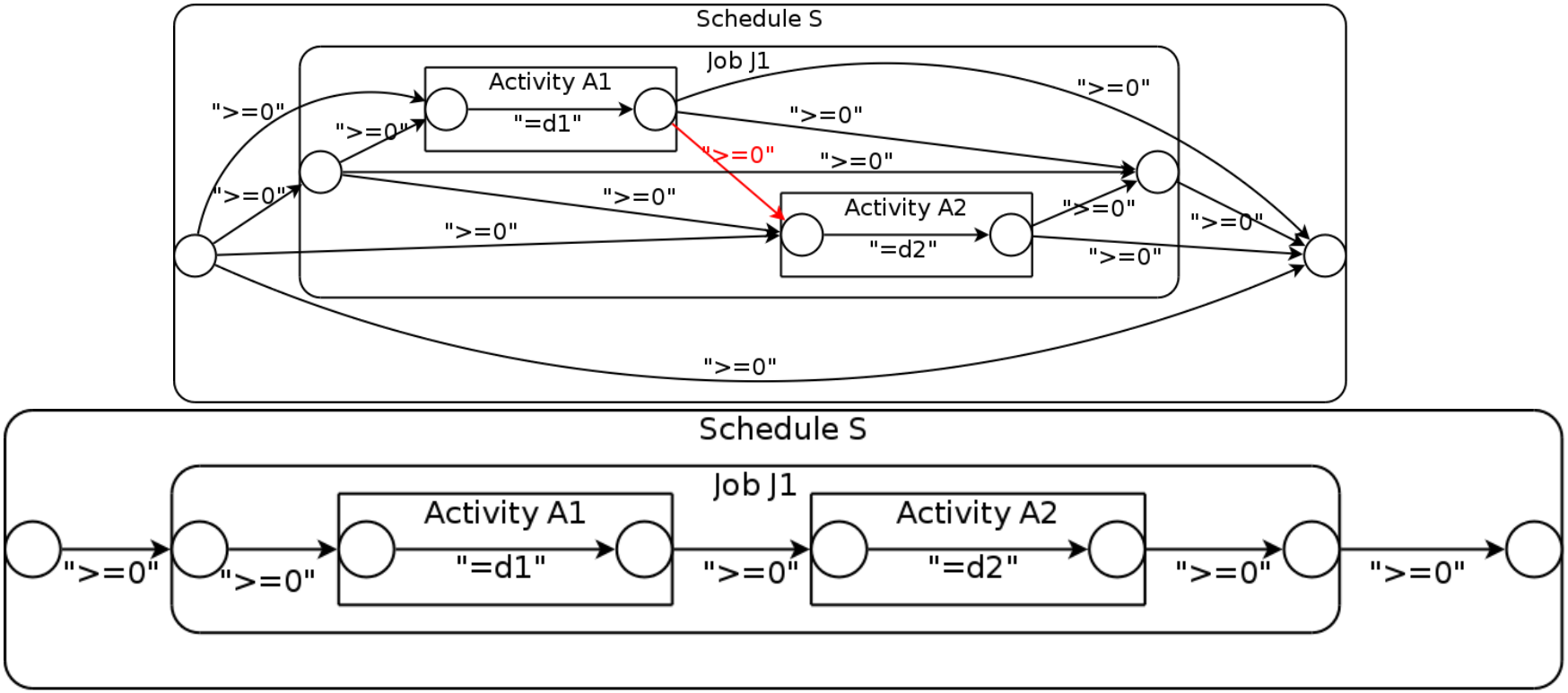
- Job-Shop (Makespan, Tardiness, Earliness-Tardiness)
- Open-Shop
- Cumulative Job-Shop
- RCPSP, RCPSP/max
- MMRCPSP, MMRCPSP/max
- Trolley Problem
- MascLib (NCOS and NCGS classes)

Internal Representation

- **Canonical** : Allow different models of the same problem to be classified in the same way
- **Homogeneous** : Ease the analysis and the information retrieval
- **Structured** : To keep the structure of the problem also helps in the analysis

Representation : Precedences

- Simplification of the precedences



Classification of problems

- Goal : Classify the model in one of the classes of problems.
- Based on characteristics.
- A simple "constraint" imposes a value for the characteristic. Its value for a model can be true or false.
- More complex constraints are build as boolean formulas of constraints (using negation, disjunction and conjunction).
- A class of problem is represented by a boolean formula.

Model Example : Job-Shop & RCPSP

Characteristic	Type	JSP	RCPSP
Unit Processing Time	boolean	–	–
Fixed Processing Time	boolean	true	true
Preemption Allowed	enum	never	never
Common Release Dates	boolean	true	true
Common Deadlines	boolean	–	–
Deadlines Exist	boolean	false	false
Form of the Precedence Graph	enum	chains	DAG
Delay between Activities	boolean	false	false
No wait between Activities	boolean	false	false
Jobs inside Jobs	boolean	false	–
Number Of State Resources	integer	0	0
Maximum Capacity	integer	1	–
All Capacities are Equal	boolean	true	–

Model Example : Job-Shop & RCPSP

Characteristic	Type	JSP	RCPSP
Reservoir Consumption	boolean	false	false
Reservoir Production	boolean	false	false
Setup Times	boolean	false	false
Disjunctive Requirements	boolean	false	false
All Activities in Jobs	boolean	true	false
Nb of Multi-Mode Activities	integer	0	0
Sum Of Requirements	integer	1	–
Objective Type	enum	minimize	minimize
Objective Form	enum	maximum	total
Objective Components	enum	completion time	lateness
Objective Scope	enum	all activities	one activity
All Due-Dates are equal	enum	–	–

Synthesis

- In input, we have a model and its classification
- The user (optionally) specifies a technology by choosing a synthesizer
- Each synthesizer associates a solving strategy to problem classes
- The synthesizer instantiates the strategy

Search

- Greedy Search
- Local Search
- Constraint Programming
- Linear Programming
- Large Neighborhood Search
- Hybrids : sequence, parallelization, master-slave combinations...

AEON is an open system

Extension Mechanisms

- Modelling Abstractions : Requires a lot of work
- Problem Characteristics : Requires to modify several classes
- Problem Classes : Write a XML file
- Synthesizers : Write a subclass of `ScheduleSynthesizer`
- Strategy : Write a subclass of `Strategy`

Outline

Experiments

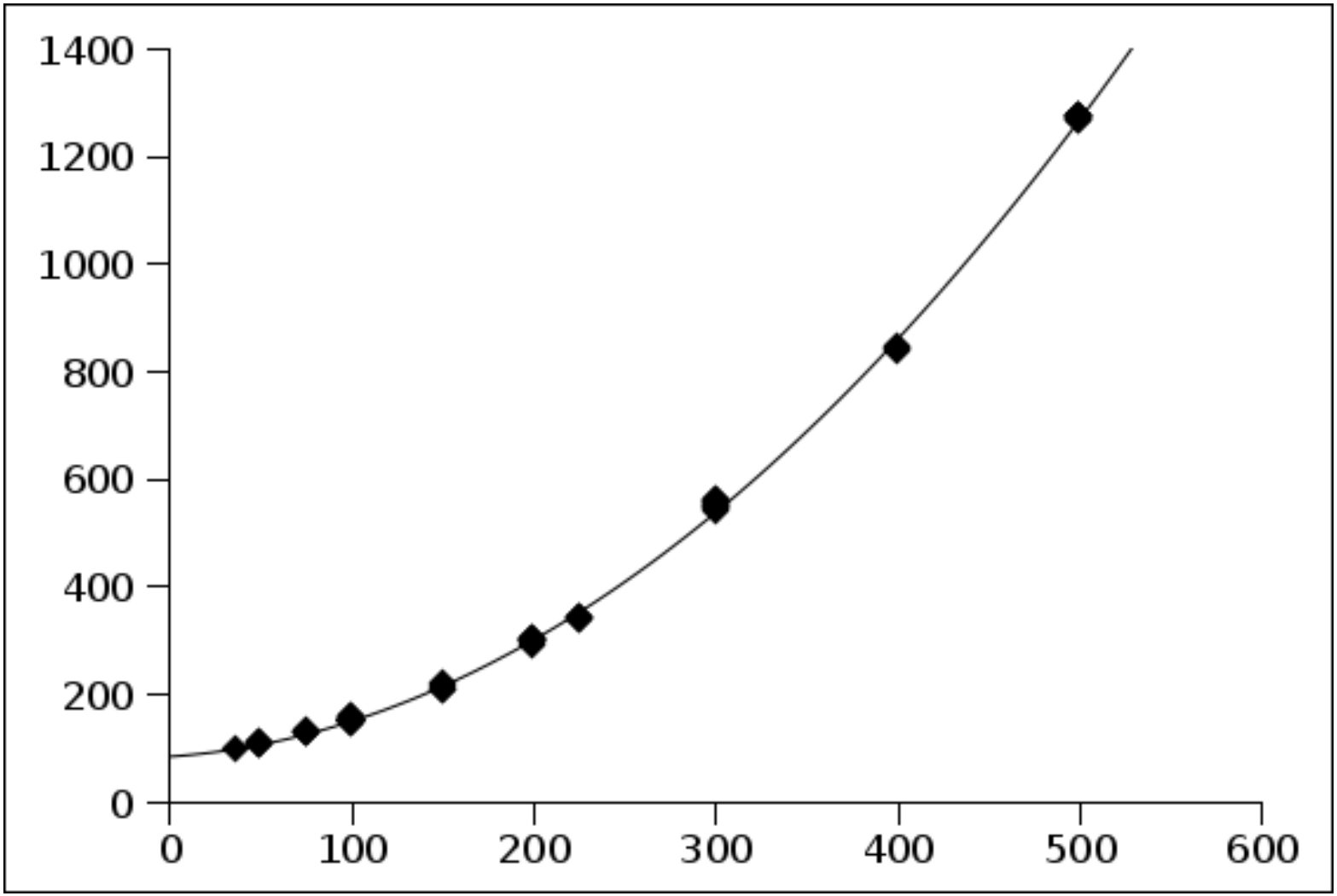
- Goal : assess the practibility of the approach
- Settings : 3 benchmarks:
  - Job-Shop Problem with makespan minimization (JSP)
  - Open-Shop Problem with makespan minimization (OSP)
  - Job-Shop with weighted tardiness minimization (JSPWT)
- Compare three synthesizers together with a specific algorithm:
  - LS (Tabu Search or Simulated Annealing)
  - CP
  - Sequence of LS and CP
  - Reference, a specific algorithm : state of the art algorithm coded in Comet
- Evaluation :
  - MRE (Mean Relative Error) $= 100 * (UB - Opt) / Opt$
  - Time to best found solution

Experiments: Results

Problem	#Inst.	Average MRE			
		Ref.	LS	CP/LNS	LS+CP
JSP	78	2.08	2.09	54.40	2.03
OSP	80	1.68	1.70	1.58/0.01	0.85
JSPTW	22	4.28	3.87	97.88	4.14

Problem	#Inst.	Average running time to best solution			
		Ref.	LS	CP/LNS	LS+CP
JSP	78	2.6	3.1	4.4(30)	3.4(52)
OSP	80	24.1	25.0	8.0(49)/ <120	30.2(50)
JSPTW	22	24.4	24.3	-(0)	-(0)

Experiments: Overhead of AEON



Outline

Related Work (1/2)

Constraint-Based modeling systems for specific domains

e.g. scheduling OPL, ILOG Scheduler, COMET

- Feature high-level abstractions
- Map them directly to the structure used in the search
- Still necessary to write one's own search algorithm

Synthesizing Algorithms from High-Level Models

[Van Hentenryck and Michel, 2007]

- Not specific to a domain application
- Limited to LS
- Analysis of the structure to create neighborhoods and searches

Related Work (2/2)

Adaptive Searches

- Self-adapting Large Neighborhood Search [Laborie and Godard, 2007]
- Impact-based search strategies [Refalo, 2004]
- Reactive search, ...

$\rightsquigarrow$ should be integrated in our systems

Engineering SLS

- Parameter optimization
- Instance-based algorithm selection
- Algorithm portfolios, ...

$\rightsquigarrow$ could be integrated in our systems

Further Work (1/2)

From prototype to robust systems

- Better detection of problem properties
- Hybrid approaches; combination of solvers
- Validation through real world applications

Extensions of the systems

- Adaptive search techniques
- Computer-assisted design techniques

Further Work (2/2)

New application domains

- Routing problems
- Graph partitionning
- Line balancing
- ...

Integration of other search paradigms

- Ant Colony Optimization
- Genetic algorithm
- ...

Modeling Languages

Yves Deville[1], Christine Solnon[2]

(1) UCLouvain, Belgium
(2) University of Lyon, France

SLS 2009

In collaboration with Vianney le Clément[1], Jean-Noël Monette[1],
Pascal Van Hentenryck (Brown Univ.)