Solving Constraint Problems in Constraint Programming

Zeynep KIZILTAN
Department of Computer Science
University of Bologna

Email: zeynep@cs.unibo.it
What is it about?

- 10 hour lectures about the core of constraint solving in CP
  - Part I: Overview of constraint programming
  - Part II: Local consistency & constraint propagation
  - Part III: Search algorithms
  - Part IV: Advanced topics, useful pointers

- Aim:
  - Teach the basics of constraint programming.
  - Emphasize the importance of local consistency & constraint propagation & search.
  - Point out the advanced topics.
  - Inform about the literature.
Warning

- We will see how constraint programming works.
- No programming examples.
PART I: Overview of Constraint Programming
Outline

- Constraint Satisfaction Problems (CSPs)
- Constraint Programming (CP)
  - Modelling
  - Backtracking Tree Search
  - Local Consistency and Constraint Propagation
Constraints are everywhere!

- No meetings before 9am.
- No registration of marks before May 15.
- The lecture rooms have a capacity.
- Two lectures of a student cannot overlap.
- No two trains on the same track at the same time.
- Salary > 45k Euros 😊
A constraint is a restriction.
There are many real-life problems that require to give a decision in the presence of constraints:
- flight / train scheduling;
- scheduling of events in an operating system;
- staff rostering at a company;
- course time tabling at a university …

Such problems are called Constraint Satisfaction Problems (CSPs).
Sudoku: An everyday-life example
CSPs: More formally

A CSP is a triple \(<X,D,C>\) where:

- \(X\) is a set of decision variables \(\{X_1,\ldots,X_n\}\).
- \(D\) is a set of domains \(\{D_1,\ldots,D_n\}\) for \(X\):
  - \(D_i\) is a set of possible values for \(X_i\).
  - usually assume finite domain.
- \(C\) is a set of constraints \(\{C_1,\ldots,C_m\}\):
  - \(C_i\) is a relation over \(X_j,\ldots,X_k\), giving the set of combination of allowed values.
  - \(C_i \subseteq D(X_j) \times \ldots \times D(X_k)\)

A solution to a CSP is an assignment of values to the variables which satisfies all the constraints simultaneously.
CSPs: A simple example

- **Variables**
  \[ X = \{X_1, X_2, X_3\} \]

- **Domains**
  \[ D(X_1) = \{1,2\}, \quad D(X_2) = \{0,1,2,3\}, \quad D(X_3) = \{2,3\} \]

- **Constraints**
  \[ X_1 > X_2 \quad \text{and} \quad X_1 + X_2 = X_3 \quad \text{and} \quad X_1 \neq X_2 \neq X_3 \neq X_1 \]

- **Solution**
  \[ X_1 = 2, \quad X_2 = 1, \quad X_3 = 3 \quad \text{alldifferent([}X_1, X_2, X_3\text{])} \]
Sudoku: An everyday-life example

- A simple CSP
  - 9x9 variables \((X_{ij})\) with domains \(\{1,\ldots,9\}\)
  - Not-equals constraints on the rows, columns, and 3x3 boxes. E.g.,
    \[
    \text{alldifferent}([X_{11}, X_{21}, X_{31}, \ldots, X_{91}])
    \]
    \[
    \text{alldifferent}([X_{11}, X_{12}, X_{13}, \ldots, X_{19}])
    \]
    \[
    \text{alldifferent}([X_{11}, X_{21}, X_{31}, X_{12}, X_{22}, X_{32}, X_{13}, X_{23}, X_{33}])
    \]
Job-Shop Scheduling: A real-life example

- Schedule jobs, each using a resource for a period, in time D by obeying the precedence and capacity constraints.
- A very common industrial problem.
- CSP:
  - variables represent the jobs;
  - domains represent the start times;
  - constraints specify precedence and exclusivity.
CSPs

- Search space: \( D(X_1) \times D(X_2) \times \ldots \times D(X_n) \)
  - very large!
- Constraint satisfaction is NP-complete:
  - no polynomial time algorithm is known to exist!
  - I can get no satisfaction 😞
- We need general and efficient methods to solve CSPs:
  - Integer and Linear Programming (satisfying linear constraints on 0/1 variables and optimising a criterion)
  - SAT (satisfying CNF formulas on 0/1 variables)
  - ...
  - Constraint Programming

How does it exactly work?
CP Machinery

- CP is composed of two phases that are strongly interconnected:
1. The CP user models the problem as a CSP:
   - define the variables and their domains;
   - specify solutions by posting constraints on the variables:
     - off-the-shelf constraints or user-defined constraints.
   - a constraint can be thought of a reusable component with its own propagation algorithm.

  WAIT TO UNDERSTAND WHAT I MEAN 😊
Modelling

- Modelling is a critical aspect.
- Given the human understanding of a problem, we need to answer questions like:
  - which variables shall we choose?
  - which constraints shall we enforce?
  - shall we use off-the-self constraints, or define and integrate our own?
  - are some constraints redundant, therefore can be avoided?
  - are there any implied constraints?
  - among alternative models, which one shall I prefer?
A problem with a simple model

- A simple CSP
  - 9x9 variables ($X_{ij}$) with domains \{1, ..., 9\}
  - Not-equals constraints on the rows, columns, and 3x3 boxes, eg.,
    \[
    \text{alldifferent}([X_{11}, X_{21}, X_{31}, \ldots, X_{91}])
    \]
    \[
    \text{alldifferent}([X_{11}, X_{12}, X_{13}, \ldots, X_{19}])
    \]
    \[
    \text{alldifferent}([X_{11}, X_{21}, X_{31}, X_{12}, X_{22}, X_{32}, X_{13}, X_{23}, X_{33}])
    \]
A problem with a complex model

- Consider a permutation problem:
  - find a permutation of the numbers \(\{1,\ldots,n\}\) s.t. some constraints are satisfied.

- One model:
  - variables \((X_i)\) for positions, domains for numbers \(\{1,\ldots,n\}\).

- Dual model:
  - variables \((Y_i)\) for numbers \(\{1,\ldots,n\}\), domains for positions.

- Often different views allow different expression of the constraints and different implied constraints:
  - can be hard to decide which is better!

- We can use multiple models and combine them via *channelling constraints* to keep consistency between the variables:
  - \(X_i = j \leftrightarrow Y_j = i\)
The user lets the CP technology solve the CSP:

- choose a search algorithm:
  - usually backtracking search performing a depth-first traversal of a search tree.
- integrate local consistency and propagation.
- choose heuristics for branching:
  - which variable to branch on?
  - which value to branch on?
Backtracking Search

- A possible efficient and simple method.
- Variables are instantiated sequentially.
- Whenever all the variables of a constraint is instantiated, the validity of the constraint is checked.
- If a (partial) instantiation violates a constraint, backtracking is performed to the most recently instantiated variable that still has alternative values.
- Backtracking eliminates a subspace from the cartesian product of all variable domains.
- Essentially performs a depth-first search.
Backtracking Search

- $X_1 \in \{1,2\}$  $X_2 \in \{0,1,2,3\}$  $X_3 \in \{2,3\}$
- $X_1 > X_2$ and $X_1 + X_2 = X_3$ and alldifferent([X_1, X_2, X_3])

Backtracking search

Fails 8 times!
Backtracking Search

- Backtracking suffers from thrashing 😞:
  - performs checks only with the current and past variables;
  - search keeps failing for the same reasons.

\[ X_1 \leq X_2 \]

\[ X_1 = X_3 \]
Constraint Programming

- Integrates local consistency and constraint propagation into the search.
- Consequently:
  - we can reason about the properties of constraints and their effect on their variables;
  - some values can be filtered from some domains, reducing the backtracking search space significantly!
Constraint Programming

- $X_1 \in \{1,2\}$  $X_2 \in \{0,1,2,3\}$  $X_3 \in \{2,3\}$
- $X_1 > X_2$ and $X_1 + X_2 = X_3$ and alldifferent([X_1, X_2, X_3])

Backtracking search + local consistency/propagation
Constraint Programming

- $X_1 \in \{1,2\}$, $X_2 \in \{0,1\}$, $X_3 \in \{2,3\}$
- $X_1 > X_2$ and $X_1 + X_2 = X_3$ and alldifferent([X_1, X_2, X_3])

Backtracking search + local consistency/propagation
Constraint Programming

- \( X_1 \in \{1,2\} \quad X_2 \in \{0,1,2,3\} \quad X_3 \in \{2,3\} \)
- \( X_1 > X_2 \) and \( X_1 + X_2 = X_3 \) and \( \text{alldifferent}([X_1, X_2, X_3]) \)

Backtracking search + local consistency/propagation
Constraint Programming

- $X_1 \in \{1,2\} \; X_2 \in \{0,1\} \; X_3 \in \{2,3\}$
- $X_1 > X_2$ and $X_1 + X_2 = X_3$ and alldifferent([X_1, X_2, X_3])

Backtracking search + local consistency/propagation

Fails only once!
Local consistency & Propagation

- Central to the process of solving CSPs which are inherently intractable.

Heuristics

Search

Local consistency & Propagation
CP

- Programming, in the sense of mathematical programming:
  - the user states declaratively the constraints on a set of decision variables.
  - an underlying solver solves the constraints and returns a solution.

- Programming, in the sense of computer programming:
  - the user needs to program a strategy to search for a solution
    - search algorithm, heuristics, …
  - otherwise, solving process can be inefficient.
Artificial Intelligence
Discrete Mathematics
Logic Programming
Operations Research

CP

Algorithms
Complexity Theory

Planning & Scheduling

Networks
Vehicle Routing
Configuration
Bioinformatics

…
Solve SUDOKU using CP!

http://www.cs.cornell.edu/gomes/SUDOKU/Sudoku.html

- very easy, not worth spending minutes 😊
- you can decide which newspaper provides the toughest Sudoku instances 😊
• Constraints can be embedded into:
  – logic programming (constraint logic programming)
    ● Prolog III, CLP(R), SICStus Prolog, ECLiPSe, CHIP, …
  – functional programming
    ● Oz
  – imperative programming
    ● often via a separate library
    ● IBM CP Solver, Gecode, Choco, Minion, …

NOTE: We will not commit to any CP language/library, rather use a mathematical and/or natural notation.
PART II: Local Consistency & Constraint Propagation
Local Consistency & Constraint Propagation

PART I: The user lets the CP technology solve the CSP:
- choose a search algorithm;
- design heuristics for branching;
- integrate local consistency and propagation.

What exactly are they?
How do they work?

Search → Local consistency & Propagation
Heuristics

Have central affect
Outline

- Local Consistency
  - Arc Consistency (AC)
  - Generalised Arc Consistency (GAC)
  - Bounds Consistency (BC)
  - Higher Levels of Consistency

- Constraint Propagation
  - Propagation Algorithms

- Specialised Propagation Algorithms
  - Global Constraints

- Generalised Propagation Algorithms
  - AC algorithms
Local Consistency

- Backtrack tree search aims to extend a partial instantiation of variables to a complete and consistent one.
  - The search space is too large!

- Some inconsistent partial assignments obviously cannot be completed.

- Local consistency is a form of inference which detects inconsistent partial assignments.
  - Consequently, the backtrack search commits into less inconsistent instantiations.

- Local, because we examine individual constraints.
  - Remember that global consistency is NP-complete!
Local Consistency: An example

- $D(X_1) = \{1,2\}$, $D(X_2) = \{3,4\}$, $C_1: X_1 = X_2$, $C_2: X_1 + X_2 \geq 1$
- $X_1 = 1$
- $X_1 = 2$
- $X_2 = 3$
- $X_4 = 4$
  - no need to check the individual assignments.
  - no need to check the other constraint.
  - unsatisfiability of the CSP can be inferred without having to search!

all inconsistent partial assignments wrt the constraint $X_1 = X_2$
Several Local Consistencies

- Most popular local consistencies:
  - Arc Consistency (AC)
  - Generalised Arc Consistency (GAC)
  - Bounds Consistency (BC)
- They detect inconsistent partial assignments of the form $X_i = j$, hence:
  - $j$ can be removed from $D(X_i)$ via propagation;
  - propagation can be implemented easily.
Arc Consistency (AC)

- Defined for binary constraints.
- A binary constraint $C$ is a relation on two variables $X_i$ and $X_j$, giving the set of allowed combinations of values (i.e. tuples):
  - $C \subseteq D(X_i) \times D(X_j)$
- $C$ is AC iff:
  - forall $v \in D(X_i)$, exists $w \in D(X_j)$ s.t. $(v,w) \in C$.
    - $v \in D(X_i)$ is said to have a *support* wrt the constraint $C$.
  - forall $w \in D(X_j)$, exists $v \in D(X_i)$ s.t. $(v,w) \in C$.
    - $w \in D(X_j)$ is said to have a *support* wrt the constraint $C$.
- A CSP is AC iff all its binary constraints are AC.
AC: An example

- \( D(X_1) = \{1, 2, 3\}, \ D(X_2) = \{2, 3, 4\}, \ C: X_1 = X_2 \)
- AC(C)?
  - 1 \( \in \) \( D(X_1) \) does not have a support.
  - 2 \( \in \) \( D(X_1) \) has 2 \( \in \) \( D(X_2) \) as support.
  - 3 \( \in \) \( D(X_1) \) has 3 \( \in \) \( D(X_2) \) as support.
  - 2 \( \in \) \( D(X_2) \) has 2 \( \in \) \( D(X_1) \) as support.
  - 3 \( \in \) \( D(X_2) \) has 3 \( \in \) \( D(X_1) \) as support.
  - 4 \( \in \) \( D(X_2) \) does not have a support.

- \( X_1 = 1 \) and \( X_2 = 4 \) are inconsistent partial assignments.
- 1 \( \in \) \( D(X_1) \) and 4 \( \in \) \( D(X_2) \) must be *removed* to achieve AC.
- \( D(X_1) = \{2, 3\}, \ D(X_2) = \{2, 3\}, \ C: X_1 = X_2 \)
  - AC(C)
Generalised Arc Consistency

- Generalisation of AC to n-ary constraints.
- A constraint $C$ is a relation on $k$ variables $X_1, \ldots, X_k$:
  - $C \subseteq D(X_1) \times \ldots \times D(X_k)$
- A support is a tuple $<d_1, \ldots, d_k> \in C$ where $d_i \in D(X_i)$.
- $C$ is GAC iff:
  - forall $X_i$ in $\{X_1, \ldots, X_k\}$, forall $v \in D(X_i)$, $v$ belongs to a support.
- AC is a special case of GAC.
- A CSP is GAC iff all its constraints are GAC.
GAC: An example

- $D(X_1) = \{1,2,3\}, D(X_2) = \{1,2\}, D(X_3) = \{1,2\}$
  - $C$: alldifferent([X_1, X_2, X_3])

- **GAC(C)?**
  - $X_1 = 1$ and $X_1 = 2$ are not supported!

- $D(X_1) = \{3\}, D(X_2) = \{1,2\}, D(X_3) = \{1,2\}$
  - $C$: $X_1 \neq X_2 \neq X_3$
  - **GAC(C)**
**Bounds Consistency (BC)**

- Defined for totally ordered (e.g., integer) domains.
- Relaxes the domain of $X_i$ from $D(X_i)$ to $[\min(X_i), \max(X_i)]$.
- **Advantages:**
  - It might be easier to look for a support in a range than in a domain;
  - Achieving BC is often cheaper than achieving GAC;
  - Achieving BC is enough to achieve GAC for monotonic constraints.
- **Disadvantage:**
  - BC might not detect all GAC inconsistencies in general.
Bounds Consistency (BC)

- A constraint $C$ is a relation on $k$ variables $X_1, \ldots, X_k$:
  - $C \subseteq D(X_1) \times \ldots \times D(X_k)$
- A *bound support* is a tuple $<d_1, \ldots, d_k> \in C$ where $d_i \in [\min(X_i) .. \max(X_i)]$.
- $C$ is BC iff:
  - forall $X_i$ in $\{X_1, \ldots, X_k\}$, $\min(X_i)$ and $\max(X_i)$ belong to a bound support.
GAC > BC: An example

- $D(X_1) = D(X_2) = \{1,2\}$, $D(X_3) = D(X_4) = \{2,3,5,6\}$, $D(X_5) = \{5\}$, $D(X_6) = \{3,4,5,6,7\}$
- $C$: `alldifferent([X_1, X_2, X_3, X_4, X_5, X_6])`
- $BC(C)$: $2 \in D(X_3)$ and $2 \in D(X_4)$ have no support.
GAC > BC: An example

- \( D(X_1) = D(X_2) = \{1,2\}, D(X_3) = D(X_4) = \{2,3,5,6\}, D(X_5) = \{5\}, D(X_6) = \{3,4,5,6,7\} \)
  
  - \( C: \text{alldifferent}([X_1, X_2, X_3, X_4, X_5, X_6]) \)

- \( \text{GAC}(C): \{2,5\} \in D(X_3), \{2,5\} \in D(X_4), \{3,5,6\} \in D(X_6) \) have no support.
GAC = BC: An example

- $D(X_1) = \{1,2,3\}$, $D(X_2) = \{1,2,3\}$, $C: X_1 < X_2$
- **BC(C):**
  - $D(X_1) = \{1,2\}$, $D(X_2) = \{2,3\}$
- **BC(C) = GAC(C):**
  - A support for $\text{min}(X_2)$ supports all the values in $D(X_2)$.
  - A support for $\text{max}(X_1)$ supports all the values in $D(X_1)$. 
Higher Levels of Consistencies

- Path consistency, k-consistencies, (i,j) consistencies, ...
- Not much used in practice:
  - detect inconsistent partial assignments with more than one <variable,value> pair.
  - cannot be enforced by removing single values from domains.
- Domain based consistencies stronger than (G)AC.
  - Singleton consistencies, triangle-based consistencies, ...
  - Becoming popular:
    - shaving in scheduling.
Outline

- **Local Consistency**
  - Arc Consistency (AC)
  - Generalised Arc Consistency (GAC)
  - Bounds Consistency (BC)
  - Higher Levels of Consistency

- **Constraint Propagation**
  - Constraint Propagation Algorithms

- **Specialised Propagation Algorithms**
  - Global Constraints

- **Generalised Propagation Algorithms**
  - AC Algorithms
Constraint Propagation

- Can appear under different names:
  - constraint relaxation
  - filtering algorithm
  - local consistency enforcing, …

- Similar concepts in other fields:
  - unit propagation in SAT.

- Local consistencies define properties that a CSP must satisfy after constraint propagation:
  - the operational behaviour is completely left open;
  - the only requirement is to achieve the required property on the CSP.
Constraint Propagation: A simple example

Input CSP: \( D(X_1) = \{1, 2\} \), \( D(X_2) = \{1, 2\} \), \( C: X_1 < X_2 \)

We can write different algorithms with different complexities to achieve the same effect.

A constraint propagation algorithm for enforcing AC

Output CSP: \( D(X_1) = \{1\} \), \( D(X_2) = \{2\} \), \( C: X_1 < X_2 \)
Constraint Propagation Algorithms

- A constraint propagation algorithm propagates a constraint C.
  - It removes the inconsistent values from the domains of the variables of C.
  - It makes C locally consistent.
  - The level of consistency depends on C:
    - GAC might be NP-complete, BC might not be possible, …
Constraint Propagation Algorithms

- When solving a CSP with multiple constraints:
  - propagation algorithms interact;
  - a propagation algorithm can wake up an already propagated constraint to be propagated again!
  - in the end, propagation reaches a fixed-point and all constraints reach a level of consistency;
  - the whole process is referred as constraint propagation.
Constraint Propagation: An example

- $D(X_1) = D(X_2) = D(X_3) = \{1,2,3\}$
  - $C_1$: alldifferent([X_1, X_2, X_3])
  - $C_2$: $X_2 < 3$
  - $C_3$: $X_3 < 3$

- Let’s assume:
  - the order of propagation is $C_1$, $C_2$, $C_3$;
  - each algorithm maintains (G)AC.

- Propagation of $C_1$:
  - nothing happens, $C_1$ is GAC.

- Propagation of $C_2$:
  - 3 is removed from $D(X_2)$, $C_2$ is now AC.

- Propagation of $C_3$:
  - 3 is removed from $D(X_3)$, $C_3$ is now AC.

- $C_1$ is not GAC anymore, because the supports of $\{1,2\} \in D(X_1)$ in $D(X_2)$ and $D(X_3)$ are removed by the propagation of $C_2$ and $C_3$.

- Re-propagation of $C_1$:
  - 1 and 2 are removed from $D(X_1)$, $C_1$ is now AC.
Properties of Constraint Propagation Algorithms

- It is not enough to be able to remove inconsistent values from domains.
- A constraint propagation algorithm must *wake up* when necessary, otherwise may not achieve the desired local consistency property.
- Events that trigger a constraint propagation:
  - when the domain of a variable changes;
  - when a variable is assigned a value;
  - when the minimum or the maximum values of a domain changes.
Outline

- Local Consistency
  - Arc Consistency (AC)
  - Generalised Arc Consistency (GAC)
  - Bounds Consistency (BC)
  - Higher Levels of Consistency

- Constraint Propagation
  - Propagation Algorithms

- Specialised Propagation Algorithms
  - Global Constraints
    - Decompositions
    - Ad-hoc algorithms

- Generalised Propagation Algorithms
  - AC Algorithms
A constraint propagation algorithm can be general or specialised:

- general, if it is applicable to any constraint;
- specialised, if it is specific to a constraint.

Specialised algorithms:

- Disadvantage:
  - has limited use;
  - is not always easy to develop one.
- Advantages:
  - exploits the constraint semantics;
  - is potentially more efficient than a general algorithm.

Worth developing specialised algorithms for recurring constraints with a reasonable semantics.
Specialised Propagation Algorithms

- **C**: $X_1 \leq X_2$
- Observation:
  - a support of $\min(X_2)$ supports all the values in $D(X_2)$;
  - a support of $\max(X_1)$ supports all the values in $D(X_1)$.
- Propagation algorithm:
  - filter $D(X_1)$ s.t. $\max(X_1) \leq \max(X_2)$;
  - filter $D(X_2)$ s.t. $\min(X_1) \leq \min(X_2)$.
- The result is GAC (and thus BC).
Example

- \( D(X_1) = \{3, 4, 7, 8\} \), \( D(X_2) = \{1, 2, 3, 5\} \), \( C: X_1 \leq X_2 \)
Example

- $D(X_1) = \{3, 4, 7, 8\}$, $D(X_2) = \{1, 2, 3, 5\}$, $C: X_1 \leq X_2$
- Propagation:
  - filter $D(X_1)$ s.t. $\max(X_1) \leq \max(X_2)$;
Example

- $D(X_1) = \{3, 4, 7, 8\}$, $D(X_2) = \{1, 2, 3, 5\}$, $C: X_1 \leq X_2$
- Propagation:
  - filter $D(X_1)$ s.t. $\max(X_1) \leq \max(X_2)$;
Example

- \( D(X_1) = \{3, 4, 7, 8\} \), \( D(X_2) = \{1, 2, 3, 5\} \), \( C: X_1 \leq X_2 \)

- Propagation:
  - filter \( D(X_1) \) s.t. \( \max(X_1) \leq \max(X_2) \);
  - filter \( D(X_2) \) s.t. \( \min(X_1) \leq \min(X_2) \);
Example

- $D(X_1) = \{3, 4, 7, 8\}$, $D(X_2) = \{1, 2, 3, 5\}$, $C: X_1 \leq X_2$

- Propagation:
  - filter $D(X_1)$ s.t. $\max(X_1) \leq \max(X_2)$;
  - filter $D(X_2)$ s.t. $\min(X_1) \leq \min(X_2)$;
Global Constraints

- Many real-life constraints are complex and not binary.
  - Specialised algorithms are often developed for such constraints!
- A complex and n-ary constraint which encapsulates a specialised propagation algorithm is called a global constraint.
Examples of Global Constraints

- **Alldifferent** constraint:
  
  - `alldifferent([X_1, X_2, ..., X_n])` holds iff
    \[ X_i \neq X_j \text{ for } i < j \in \{1, ..., n\} \]

- useful in a variety of context
  
  - Timetabling (e.g. exams with common students must occur at different times)
  - Tournament scheduling (e.g. a team can play at most once in a week)
  - Configuration (e.g. a particular product cannot have repeating components)
  - ...
Beyond Alldifferent

- **NValue** constraint:
  - one generalisation of alldifferent
  - $\text{nvalue}([X_1, X_2, \ldots, X_n], N)$ holds iff
    \[ N = |\{X_i \mid 1 \leq i \leq n \}| \]
  - $\text{nvalue}([1, 2, 2, 1, 3], 3)$
  - alldifferent when $N = n$
  - Useful when values represent resources and we want to limit the usage of resources. E.g.,
    - Minimise the total number of resources used;
    - The total number of resources used must be between a specific interval;
    - …
Beyond Alldifferent

- **Global cardinality constraint:**
  - another generalisation of alldifferent
  - \( \text{gcc}([X_1, X_2, ..., X_n], [v_1, ..., v_m], [O_1, ..., O_m]) \) iff
    \[\forall j \in \{1, ..., m\} \quad O_j = |\{X_i | X_i = v_j, 1 \leq i \leq n\}|\]
  - \( \text{gcc}([1, 1, 3, 2, 3], [1, 2, 3, 4], [2, 1, 2, 0]) \)
  - Useful again when values represent resources
  - We can now limit the usage of each resource individually. E.g.,
    - Resource 1 can be used at most three times
    - Resource 2 can be used min 2 max 5 times
    - ...

- ...
Symmetry Breaking Constraints

Consider the following scenario:
- \([X_1, X_2, \ldots, X_n]\) and \([Y_1, Y_2, \ldots, Y_n]\) represent the 2 day event assignments of a conference
- Each day has \(n\) slots and the days are indistinguishable
- Need to avoid symmetric assignments

Global constraints developed for this purpose are called symmetry breaking constraints.

Lexicographic ordering constraint:
- \(\text{lex}([X_1, X_2, \ldots, X_n], [Y_1, Y_2, \ldots, Y_n])\) holds iff:
  \(X_1 < Y_1\) OR \((X_1 = Y_1 \text{ AND } X_2 < Y_2)\) OR \(\ldots\)
  \((X_1 = Y_1 \text{ AND } X_2 = Y_2 \text{ AND } \ldots \text{ AND } X_n \leq Y_n)\)
- \(\text{lex}([1, 2, 4],[1, 3, 3])\)
We might sometimes want a sequence of variables obey certain patterns. E.g.,
- regulations in scheduling

A promising direction in CP is the ability of modelling problems via automata/grammar.

Global constraints developed for this purpose are called **grammar constraints**.

**Regular constraint:**
- \(\text{regular}([X_1, X_2, \ldots, X_n], A)\) holds iff \(<X_1, X_2, \ldots, X_n>\) forms a string accepted by the DFA \(A\) (which accepts a regular language).
- \(\text{regular}([a, a, b], A)\), \(\text{regular}([b], A)\), \(\text{regular}([b, c, c, c, c, c], A)\) with \(A\)
Specialised Algorithms for Global Constraints

- How do we develop specialised algorithms for global constraints?
- Two main approaches:
  - constraint decomposition
  - ad-hoc algorithm
A global constraint is decomposed into smaller and simpler constraints each which has a known propagation algorithm.

Propagating each of the constraints gives a propagation algorithm for the original global constraint.

- A very effective and efficient method for some global constraints
Decomposition of Among

- \( \text{among}([X_1, X_2, \ldots, X_n], [d_1, d_2, \ldots, d_m], N) \) holds iff
  \( N = |\{X_i \mid X_i \in \{d_1, d_2, \ldots, d_m\} 1 \leq i \leq n \}| \)

- Decomposition:
  - \( B_i \) with \( D(B_i) = \{0, 1\} \) for \( 1 \leq i \leq n \)
  - \( C_i: B_i = 1 \iff X_i \in \{d_1, d_2, \ldots, d_m\} \) for \( 1 \leq i \leq n \)
  - \( \sum_i B_i = N \)

- \( \text{AC}(C_i) \) for \( 1 \leq i \leq n \) and \( \text{BC}(\sum_i B_i = N) \) ensures GAC on among.
Decomposition of Lex

- \( \text{lex}([X_1, X_2, \ldots, X_n], [Y_1, Y_2, \ldots, Y_n]) \)
- Decomposition:
  - \( B_i \) with \( D(B_i) = \{0, 1\} \) for \( 1 \leq i \leq n+1 \) to indicate the vectors have been ordered by position \( i-1 \)
  - \( B_1 = 0 \)
  - \( C_i: (B_i = B_{i+1} = 0 \ \text{AND} \ X_i = Y_i) \ \text{OR} \ (B_i = 0 \ \text{AND} \ B_{i+1} = 1 \ \text{AND} \ X_i < Y_i) \ \text{OR} \ (B_i = B_{i+1} = 1) \) for \( 1 \leq i \leq n \)
- \( \text{GAC}(C_i) \) ensures GAC on \( \text{lex} \).
Constraint Decompositions

- May not always provide an effective propagation.
- Often GAC on the original constraint is stronger than (G)AC on the constraints in the decomposition.
- E.g., \( C: \text{alldifferent}([X_1, X_2, ..., X_n]) \)
- Decomposition following the definition:
  - \( C_{ij}: X_i \neq X_j \) for \( i < j \in \{1, ..., n\} \)
  - AC on the decomposition is weaker than GAC on \text{alldifferent}.
  - E.g., \( D(X_1) = D(X_2) = D(X_3) = \{1,2\}, \ C: \text{alldifferent}([X_1, X_2, X_3]) \)
  - \( C_{12}, C_{13}, C_{23} \) are all AC, but \( C \) is not GAC.
Constraint Decompositions

- E.g., $C: \text{lex}([X_1, X_2, \ldots, X_n], [Y_1, Y_2, \ldots, Y_n])$
- OR decomposition:
  - $X_1 < Y_1$ OR $(X_1 = Y_1 \text{ AND } X_2 < Y_2)$ OR ...
  - $(X_1 = Y_1 \text{ AND } X_2 = Y_2 \text{ AND } \ldots \text{ AND } X_n \leq Y_n)$
  - AC on the decomposition is weaker than GAC on lex.
- E.g., $D(X_1) = \{0, 1, 2\}, D(X_2) = \{0, 1\}, D(Y_1) = \{0, 1\}, D(Y_2) = \{0, 1\}$
  - $C: \text{Lex}([X_1, X_2], [Y_1, Y_2])$
  - $C$ is not GAC but the decomposition does not prune anything.
Constraint Decompositions

- AND decomposition of $\text{lex}([X_1, X_2, \ldots, X_n], [Y_1, Y_2, \ldots, Y_n])$:
  - $X_1 \leq Y_1$ AND $(X_1 = Y_1 \rightarrow X_2 \leq Y_2)$ AND ... 
    $(X_1 = Y_1$ AND $X_2 = Y_2$ AND ... $X_{n-1} = Y_{n-1} \rightarrow X_n \leq Y_n)$
  - AC on the decomposition is weaker than GAC on $\text{lex}$.
  - E.g., $D(X_1) = \{0, 1\}$, $D(X_2) = \{0, 1\}$, $D(Y_1) = \{1\}$, $D(Y_2) = \{0\}$
    $C: \text{Lex}([X_1, X_2], [Y_1, Y_2])$
  - $C$ is not GAC but the decomposition does not prune anything.
Constraint Decompositions

- Different decompositions of a constraint may be incomparable.
  - Difficult to know which one gives a better propagation for a given instance of a constraint.

- C: Lex([X₁, X₂], [Y₁, Y₂])
  - D(X₁) = {0, 1}, D(X₂) = {0, 1}, D(Y₁) = {1}, D(Y₂) = {0}
    - AND decomposition is weaker than GAC on lex, whereas OR decomposition maintains GAC.
  - D(X₁) = {0, 1, 2}, D(X₂) = {0, 1}, D(Y₁) = {0, 1}, D(Y₂) = {0, 1}
    - OR decomposition is weaker than GAC on lex, whereas OR decomposition maintains GAC.
Constraint Decompositions

- Even if effective, may not always provide an efficient propagation.
- Often GAC on a constraint via a specialised algorithm is maintained faster than (G)AC on the constraints in the decomposition.
Constraint Decompositions

- **C**: Lex([X₁, X₂], [Y₁, Y₂])
  - D(X₁) = {0, 1}, D(X₂) = {0, 1}, D(Y₁) = {1}, D(Y₂) = {0}
    - AND decomposition is weaker than GAC on lex, whereas OR decomposition maintains GAC
  - D(X₁) = {0, 1, 2}, D(X₂) = {0, 1}, D(Y₁) = {0, 1}, D(Y₂) = {0, 1}
    - OR decomposition is weaker than GAC on lex, whereas OR decomposition maintains GAC

- AND or OR decompositions have complementary strengths!
  - Combining them gives us a decomposition which maintains GAC on lex.

- Too many constraints to post and propagate!
- A dedicated algorithm runs amortised in O(1).
Dedicated Algorithms

- Dedicated ad-hoc algorithms provide effective and efficient propagation.
- Often:
  - GAC is maintained in polynomial time.
  - Many more inconsistent values are detected compared to the decompositions.
Benefits of Global Constraints

- **Modelling benefits**
  - Reduce the gap between the problem statement and the model.
  - Capture recurring modelling patterns.
  - May allow the expression of constraints that are otherwise not possible to state using primitive constraints (*semantic*).

- **Solving benefits**
  - More inference in propagation (*operational*).
  - More efficient propagation (*algorithmic*).
Dedicated Algorithm for AllDifferent

- **GAC algorithm based on matching theory.**
  - Establishes a relation between the solutions of the constraint and the properties of a graph.
  - Runs in time $O(dn^{1.5})$.
- **Value graph**: bipartite graph between variables and their possible values.
- **Matching**: set of edges with no two edges having a node in common.
- **Maximal matching**: largest possible matching.
Dedicated Algorithm for Alldifferent

- An assignment of values to the variables $X_1, X_2, \ldots, X_n$ is a solution iff it corresponds to a maximal matching.
  - Edges that do not belong to a maximal matching can be deleted.

- The challenge is to compute such edges efficiently.
  - Exploit concepts like strongly connected components, alternating paths, …
Dedicated Algorithm for Alldifferent

- $D(X_1) = \{1,3\}$, $D(X_2) = \{1,3\}$, $D(X_3) = \{1,2\}$

Variable-value graph

Diagram:

- $X_1$ connected to 1
- $X_2$ connected to 2
- $X_3$ connected to 3
Dedicated Algorithm for Alldifferent

- \( D(X_1) = \{1,3\} \), \( D(X_2) = \{1,3\} \), \( D(X_3) = \{1,2\} \)

A maximal matching
Dedicated Algorithm for AllDifferent

- \( D(X_1) = \{1, 3\} \), \( D(X_2) = \{1, 3\} \), \( D(X_3) = \{1, 2\} \)

Another maximal matching

Does not belong to any maximal matching
Dedicated Algorithms

- Is it always easy to develop a dedicated algorithm for a given constraint?
- There’s no single recipe!
- A nice semantics often gives us a clue!
  - Graph Theory
  - Flow Theory
  - Combinatorics
  - Complexity Theory, ...
- GAC may as well be NP-hard!
  - In that case, algorithms which maintain weaker consistencies (like BC) are of interest.
GAC for N-value Constraint

- nvalue([X₁, X₂, …, Xₙ], N) holds iff N = |{Xᵢ | 1 ≤ i ≤ n}|
- Reduction from 3 SAT.
  - Given a Boolean formula in k variables (labelled from 1 to k) and m clauses, we construct an instance of nvalue([X₁, X₂, …, Xₖ₊ₘ], N):
    - D(Xᵢ) = {i, i'} for i ∈ {1,…, k} where Xᵢ represents the truth assignment of the SAT variables;
    - Xᵢ where i > k represents a SAT clause (disjunction of literals);
    - for a given clause like x V y’ V z, D(Xᵢ) = {x, y’, z}.
  - By construction, X₁, …, Xₖ will consume all the k distinct values.
  - When N = k, nvalue has a solution iff the original SAT problem has a satisfying assignment.
    - Otherwise we will have more than k distinct values.
    - Hence, testing a value for support is NP-complete, and enforcing GAC is NP-hard!
GAC for Nvalue Constraint

- E.g., $C_1$: (a OR b’ OR c) AND
  $C_2$: (a’ OR b OR d) AND
  $C_3$: (b’ OR c’ OR d)
- The formula has 4 variables (a, b, c, d) and 3 clauses ($C_1$, $C_2$, $C_3$).
- We construct nvalue([X_1, X_2, ..., X_7], 4) where:
  - $D(X_1) = \{a, a’\}$, $D(X_2) = \{b, b’\}$, $D(X_3) = \{c, c’\}$, $D(X_4) = \{d, d’\}$, $D(X_5) = \{a, b’, c\}$, $D(X_6) = \{a’, b, d\}$, $D(X_7) = \{b’, c’, d\}$
- An assignment to $X_1, ..., X_4$ will consume 4 distinct values.
- Not to exceed 4 distinct values, the rest of the variables must have intersecting values with $X_1, ..., X_4$.
- Such assignments will make the SAT formula TRUE.
Outline

- **Local Consistency**
  - Arc Consistency (AC)
  - Generalised Arc Consistency (GAC)
  - Bounds Consistency (BC)
  - Higher Levels of Consistency

- **Constraint Propagation**
  - Propagation Algorithms

- **Specialised Propagation Algorithms**
  - Global Constraints
    - Decompositions
    - Ad-hoc algorithms

- **Generalised Propagation Algorithms**
  - AC Algorithms
Generalised Propagation Algorithms

- Not all constraints have nice semantics we can exploit to devise an efficient specialised propagation algorithm.
- Consider a product configuration problem:
  - compatibility constraints on hardware components:
    - only certain combinations of components work together.
  - compatibility may not be a simple pairwise relationship:
    - video cards supported function of motherboard, CPU, clock speed, O/S, ...
Production Configuration Problem

- 5-ary constraint:
  - Compatible (motherboard345, intelCPU, 2GHz, 1GBRam, 80GBdrive).
  - Compatible (motherboard346, intelCPU, 3GHz, 2GBRam, 100GBdrive).
  - Compatible (motherboard346, amdCPU, 2GHz, 2GBRam, 100GBdrive).
  - ...
Crossword Puzzle

- Constraints with different arity:
  - Word₁ ([X₁,X₂,X₃])
  - Word₂ ([X₁,X₁₃,X₁₆])
  - ...

- No simple way to decide acceptable words other than to put them in a table.
A generic propagation algorithm.
- Enforces GAC on an n-ary constraint given by:
  - a set of allowed tuples;
  - a set of disallowed tuples;
  - a predicate answering if a constraint is satisfied or not.
- Sometimes called the “table” constraint:
  - user supplies table of acceptable values.

Complexity: $O(ed^n)$ time
Hence, $n$ cannot be too large!
- Many solvers limits it to 3 or so.
Arc Consistency Algorithms

- Generic AC algorithms with different complexities and advantages:
  - AC3
  - AC4
  - AC6
  - AC2001
  - ...

AC-3

- **Idea:**
  - Revise \((X_i, C)\): removes unsupported values of \(X_i\) and returns TRUE.
  - Place each \((X_i, C)\) where \(X_i\) participates to \(C\) and its domain is potentially not AC, in a queue \(Q\);
  - While \(Q\) is not empty:
    - Select and remove \((X_i, C)\) from \(Q\);
    - If \(\text{revise}(X_i, C)\) then
      - If \(\text{D}(X_i) = \{\}\) then return FALSE;
      - else place \(\{(X_j, C') \mid X_i, X_j \text{ participate in some } C'\}\) into \(Q\).
AC-3

- AC-3 achieves AC on binary CSPs in $O(ed^3)$ time and $O(e)$ space.
  - Time complexity is not optimal 😞
  - Revise does not remember anything about past computations and re-does unnecessary work.
AC-3

(X, C₁) is put in Q

only check of X ← 3 was necessary!
AC-4

- Stores max. amount of info in a preprocessing step so as to avoid redoing the same constraints checks.

- Idea:
  - Start with an empty queue Q.
  - Maintain counter\([X_i, v_j, X_k]\) where \(X_i, X_k\) participate in a constraint \(C_{ik}\) and \(v_j \in D(X_i)\)
    - Stores the number of supports for \(X_i \leftarrow v_j\) on \(C_{ik}\).
  - Place all supports of \(X_i \leftarrow v_j\) (in all constraints) in a list \(S[X_i, v_j]\).
AC-4

- **Initialisation:**
  - All possible constraint checks are performed.
  - Each time a support for $X_i \leftarrow v_j$ is found, the corresponding counters and lists are updated.
  - Each time a support for $X_i \leftarrow v_j$ is not found, remove $v_j$ from $D(X_i)$ and place $(X_i, v_j)$ in $Q$ for future propagation.
  - If $D(X_i) = \{ \}$ then return FALSE.
AC-4

- Propagation:
  - While Q is not empty:
    - Select and remove \((X_i, v_j)\) from Q;
    - For each \((X_k, v_t)\) in \(S[X_i, v_j]\)
      - If \(v_t \in D(X_k)\) then
        - decrement counter\([X_k, v_t, X_i]\)
        - If counter\([X_k, v_t, X_i]\] = 0 then
          - Remove \(v_t\) from \(D(X_k)\); add \((X_k, v_t)\) to Q
          - If \(D(X_k) = \{\}\) then return FALSE.
AC-4

No additional constraint check!

(y,3) is put in Q

\[
\begin{align*}
\text{counter}[x,1,y] &= 4 & \text{counter}[y,1,x] &= 1 & \text{counter}[y,1,z] &= 1 \\
\text{counter}[x,2,y] &= 3 & \text{counter}[y,2,x] &= 2 & \text{counter}[y,2,z] &= 1 \\
\text{counter}[x,3,y] &= 2 & \text{counter}[y,3,x] &= 3 & \text{counter}[y,3,z] &= 0 \\
\text{counter}[x,4,y] &= 1 & \text{counter}[y,4,x] &= 4 & \text{counter}[z,3,y] &= 3
\end{align*}
\]

\[
\begin{align*}
S[x,1] &= \{(y,1),(y,2),(y,3),(y,4)\} \\
S[x,2] &= \{(y,2),(y,3),(y,4)\} \\
S[x,3] &= \{(y,3),(y,4)\} \\
S[x,4] &= \{(y,4)\} \\
S[y,1] &= \{(x,1),(z,3)\} \\
S[y,2] &= \{(x,1),(x,2),(z,3)\} \\
S[y,3] &= \{(x,1),(x,2),(x,3)\} \\
S[y,4] &= \{(x,1),(x,2),(x,3),(x,4),(z,3)\} \\
S[z,3] &= \{(y,1),(y,2),(y,4)\}
\end{align*}
\]
AC-4

- AC-3 achieves AC on binary CSPs in $O(ed^2)$ time and $O(ed^2)$ space.
  - Time complexity is optimal 😊
  - Space complexity is not optimal 😞

- AC-6 and AC-2001 achieve AC on binary CSPs in $O(ed^2)$ time and $O(ed)$ space.
  - Time complexity is optimal 😊
  - Space complexity is optimal 😊
PART IV: Search Algorithms
Outline

● Depth-first Search Algorithms
  – Chronological Backtracking
  – Conflict Directed Backjumping
  – Dynamic Backtracking
  – Branching Strategies
  – Heuristics

● Best-First Search Algorithms
  – Limited Discrepancy Search
Depth-first Search Algorithms

- Backtracking tree search algorithms essentially perform depth-first traversal of a search tree.
  - Every node represents a decision made on a variable.
  - At each node:
    - check every completely assigned constraint;
    - If consistent continue down in the tree;
    - otherwise prune the underlying subtrees and backtrack to an uninstantiated variable that still has alternative values.
Chronological Backtracking

- Backtracks to the most recent variable.
Chronological Backtracking

- Suffers from trashing.
  - The same failure can be remade an exponential number of times.
Non-Chronological Backtracking

- Backtrack on a culprit variable.
- E.g.,
  - Backtracking to $X_5$ is pointless.
  - Better to backtrack on $X_4$. 
Conflict Sets

- CS($X_k$): assigned variables in conflict with some value of $X_k$. 
Conflict Directed Backjumping

- Backtracks to the last variable in the conflict set.
- Intermediate decisions are removed.
No-goods

- Subset of incompatible assignments.
- E.g., map colouring problem.
  - $X_1$, $X_2$, $X_3$ are adjacent with $D = \{1, 2\}$.
  - $(X_1 = a \text{ and } X_3 = a)$ or equivalently $(X_1 = a \rightarrow X_3 \neq a)$ is a no-good.
- No-good resolution:
  - $X_1 = a \rightarrow X_3 \neq a$
  - $X_2 = b \rightarrow X_3 \neq b$
Dynamic Backtracking

- One no-good for each incompatible value is maintained.
  - Empty domain: new no-good by no-good resolution.
  - Backtrack to the variable in the right hand side of the no-good.
Dynamic Backtracking

- Backtracks to the last decision responsible for the dead-end.
- Intermediate decisions are not removed.
Branching Strategies

- The method of extending a node in the search tree.
  - Usually consists of posting a unary constraint on a chosen variable $X_i$.
  - $X_i$ & the ordering of the branches are chosen by the heuristics.

- **D-way branching:**
  - One branch is generated for each $v_j \in D(X_i)$ by $X_i \leftarrow v_j$.

- **2-way branching:**
  - 2 branches are generated for each $v_j \in D(X_i)$ by $X_i \leftarrow v_j$ and $X_i \leftarrow \setminus v_j$.

- **Domain splitting:**
  - k branches are generated by $X_i \in D_j$ where $D_1 \cdots D_k$ are partitions of $D_i$. 
Variable and Value Ordering Heuristics

- Guide the search.
- Problem specific vs generic heuristics.
- Static Heuristics:
  - a variable is associated with each level;
  - branches are generated in the same order all over the tree;
  - calculated once and for all before search starts, hence cheap to evaluate.
Variable and Value Ordering Heuristics

- **Dynamic Heuristics:**
  - at any node, any variable & branch can be considered;
  - decided dynamically during search, hence costly;
  - takes into account the current state of the search tree.
Variable Ordering Heuristics

- Fail-first principle: to succeed, try first where you are most likely to fail.
- Min domain (dom):
  - choose next the variable with minimum domain.
- Most constrained (deg):
  - choose next the variable involved in most number of constraints.
- Combinations
  - dom + deg; dom / deg
Value Ordering Heuristics

- Succeed-first principle: choose next the value most likely to be part of a solution.
  - Approximating the number of solutions.
  - Looking at the remaining domain sizes when a value is assigned to a variable.
Problems with Depth-first Search

- The branches out of a node, ordered by a value ordering heuristic, are explored in left-to-right order, the left-most branch being the most promising.
- For many problems, heuristics are more accurate at deep nodes.
- Depth-first search:
  - puts tremendous burden on the heuristics early in the search and light burden deep in the search;
  - consequently mistakes made near the root of the tree can be costly to correct.
- Best-first search strategy is of interest.
Limited Discrepancy Search

- A discrepancy is the case where the search does not follow the value ordering heuristic and thus does not take the left-most branch out of a node.

- LDS:
  - Trusts the value ordering heuristic and gives priority to the left branches.
  - Iteratively searches the tree by increasing number of discrepancies, preferring discrepancies that occur near the root of the tree.
Limited Discrepancy Search

- The search recovers from mistakes made early in the search.

Figure 1: Paths with 0, 1, 2, and 3 Discrepancies in a Depth 3 Binary Tree
PART IV: Some Useful Pointers about CP
(Incomplete) List of Advanced Topics

- Modelling
- Global constraints, propagation algorithms
- Search algorithms
- Heuristics
- Symmetry breaking
- Optimisation
- Local search
- Soft constraints, preferences
- Temporal constraints
- Quantified constraints
- Continuous constraints
- Planning and scheduling
- SAT
- Complexity and tractability
- Uncertainty
- Robustness
- Structured domains
- Randomisation
- Hybrid systems
- Applications
- Constraint systems
- No good learning
- Explanations
- Visualisation
Literature

- **Books**
  - *Handbook of Constraint Programming*
    F. Rossi, P. van Beek, T. Walsh (eds), Elsevier Science, 2006.

Some online chapters:
- Chapter 1 - **Introduction**
- Chapter 3 - **Constraint Propagation**
- Chapter 6 - **Global Constraints**
- Chapter 10 - **Symmetry in CP**
- Chapter 11 - **Modelling**
Literature

- **Books**
  - Constraint Logic Programming Using Eclipse
  - Principles of Constraint Programming
  - Constraint Processing
  - Constraint-based Local Search
  - The OPL Optimization Programming Languages
Literature

- **People**
  - Barbara Smith
    - Modelling, symmetry breaking, search heuristics
    - Tutorials and book chapter
  - Christian Bessiere
    - Constraint propagation
    - Global constraints
      - Nvalue constraint
    - Book chapter
  - Jean-Charles Regin
    - Global constraints
      - Alldifferent, global cardinality, cardinality matrix
  - Toby Walsh
    - Modelling, symmetry breaking, global constraints
    - Various tutorials
Literature

- Journals
  - Constraints
  - Artificial Intelligence
  - Journal of Artificial Intelligence Research
  - Journal of Heuristics
  - Intelligenza Artificiale (AI*IA)
  - Informs Journal on Computing
  - Annals of Mathematics and Artificial Intelligence
Literature

**Conferences**

- Principles and Practice of Constraint Programming (CP)
  http://www.cs.ualberta.ca/~ai/cp/
- Integration of AI and OR Techniques in CP (CP-AI-OR)
  http://www.cs.cornell.edu/~vanhoeve/cpaior/
- National Conference on AI (AAAI)
  http://www.aaai.org
- International Joint Conference on Artificial Intelligence (IJCAI)
  http://www.ijcai.org
- European Conference on Artificial Intelligence (ECAI)
  http://www.eccai.org
- International Symposium on Practical Aspects of Declarative Languages (PADL)
  http://www.informatik.uni-trier.de/~ley/db/conf/padl/index.html
Literature

- **Schools and Tutorials**
  - ACP summer schools:
  - AI conference tutorials (IJCAI’09, 07, 05, ECAI’04 …).
  - CP conference tutorials.
  - CP-AI-OR master classes.
Literature

- Solvers & Languages
  - Choco (http://choco.sourceforge.net/)
  - Comet (http://www.comet-online.org/)
  - Eclipse (http://eclipse.crosscoreop.com/)
  - FaCiLe (http://www.recherche.enac.fr/opti/facile/)
  - Gecode (http://www.gecode.org/)
  - IBM ILOG Solver (http://www-01.ibm.com/software/websphere/products/optimization/)
  - Koalog Constraint Solver (http://www.gecode.org/)
  - Minion (http://minion.sourceforge.net/)
  - OPL (http://www.ilog.com/products/oplstudio/)
  - Sicstus Prolog (http://www.sics.se/isl/sicstuswww/site/index.html)