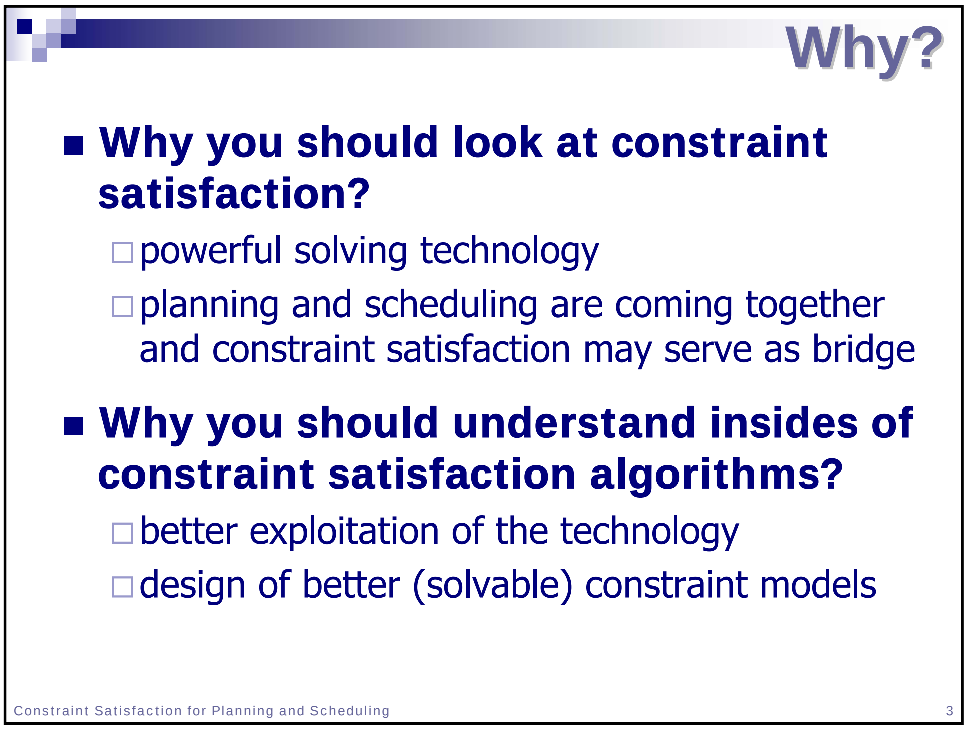
Constraint Satisfaction for Planning & Scheduling

Roman Barták
Charles University, Prague (CZ)

`roman.bartak@mff.cuni.cz`

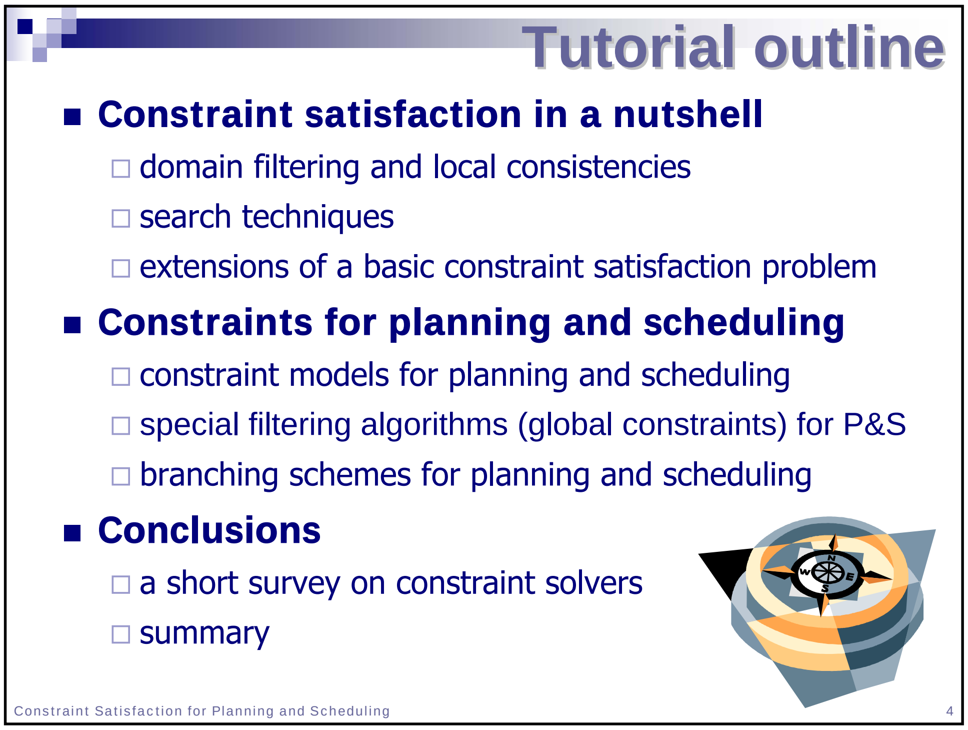
What?

- **What is the topic of the tutorial?**
  - constraint satisfaction techniques useful for P&S
- **What is constraint satisfaction?**
  - technology for modeling and solving combinatorial optimization problems
- **What is the difference from AIPS02 tutorial?**
  - focus on constraint satisfaction in general
  - more explanations but less broad



Why?

- **Why you should look at constraint satisfaction?**
  - □ powerful solving technology
  - □ planning and scheduling are coming together and constraint satisfaction may serve as bridge
- **Why you should understand insides of constraint satisfaction algorithms?**
  - □ better exploitation of the technology
  - □ design of better (solvable) constraint models



Tutorial outline

- **Constraint satisfaction in a nutshell**
  - □ domain filtering and local consistencies
  - □ search techniques
  - □ extensions of a basic constraint satisfaction problem
- **Constraints for planning and scheduling**
  - □ constraint models for planning and scheduling
  - □ special filtering algorithms (global constraints) for P&S
  - □ branching schemes for planning and scheduling
- **Conclusions**
  - □ a short survey on constraint solvers
  - □ summary

Constraint satisfaction
in a nutshell

Constraint technology

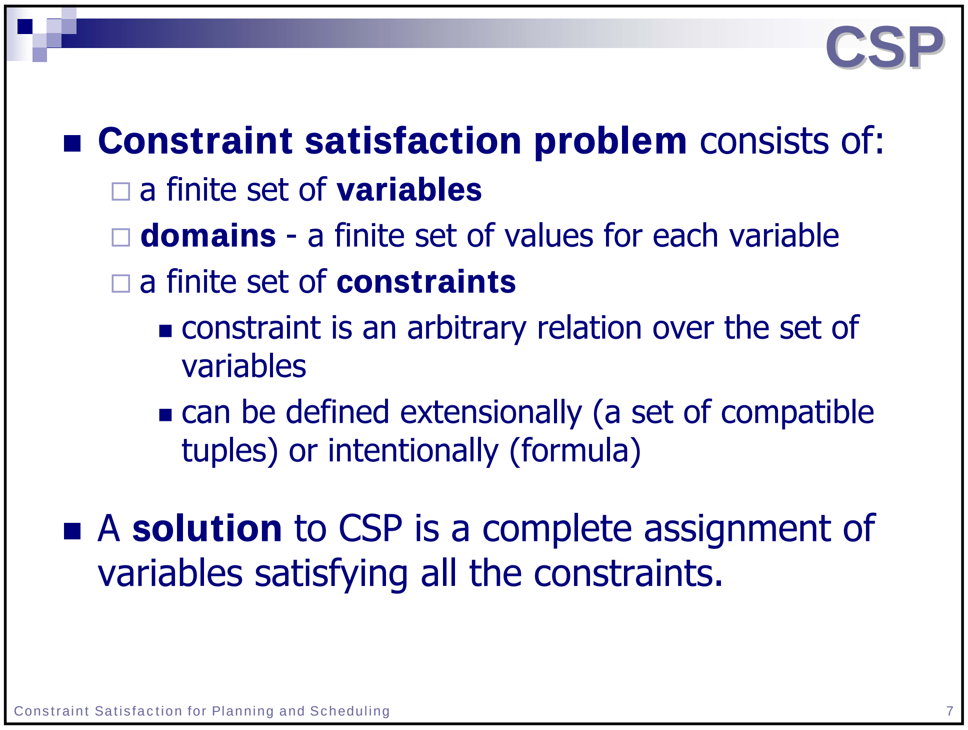
based on **declarative problem description** via:

- **variables with domains** (sets of possible values) e.g. start of activity with time windows
- **constraints** restricting combinations of variables e.g. endA < startB

constraint **optimization** via objective function
  e.g. minimize makespan

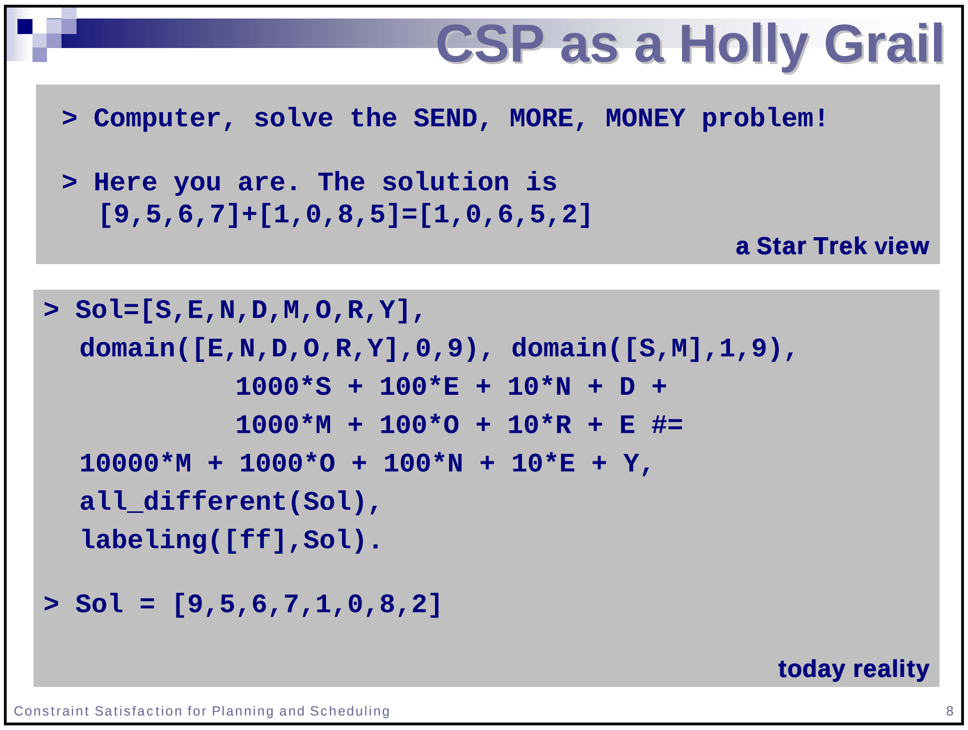
Why to use constraint technology?

- understandable
- open and extendible
- proof of concept



CSP

- **Constraint satisfaction problem** consists of:
  - a finite set of **variables**
  - **domains** - a finite set of values for each variable
  - a finite set of **constraints**
    - constraint is an arbitrary relation over the set of variables
    - can be defined extensionally (a set of compatible tuples) or intentionally (formula)
- A **solution** to CSP is a complete assignment of variables satisfying all the constraints.



CSP as a Holly Grail

```
> Computer, solve the SEND, MORE, MONEY problem!

> Here you are. The solution is
  [9,5,6,7]+[1,0,8,5]=[1,0,6,5,2]
```

a Star Trek view

```
> Sol=[S,E,N,D,M,O,R,Y],
  domain([E,N,D,O,R,Y],0,9), domain([S,M],1,9),
            1000*S + 100*E + 10*N + D +
            1000*M + 100*O + 10*R + E #=
  10000*M + 1000*O + 100*N + 10*E + Y,
  all_different(Sol),
  labeling([ff],Sol).

> Sol = [9,5,6,7,1,0,8,2]
```

today reality

Rudová (2002)

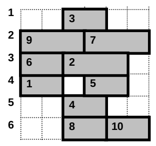
Using CSP

- To solve the problem it is enough to design a **constraint model**, that is to decide about the variables, their domains, and constraints!

Example:
Let W be the width of a rectangular area A and H be its height. The task is to place N rectangles of height 1 but of different widths into the area A in such a way that no two rectangles overlap. Let w_i be the width of the rectangle i.

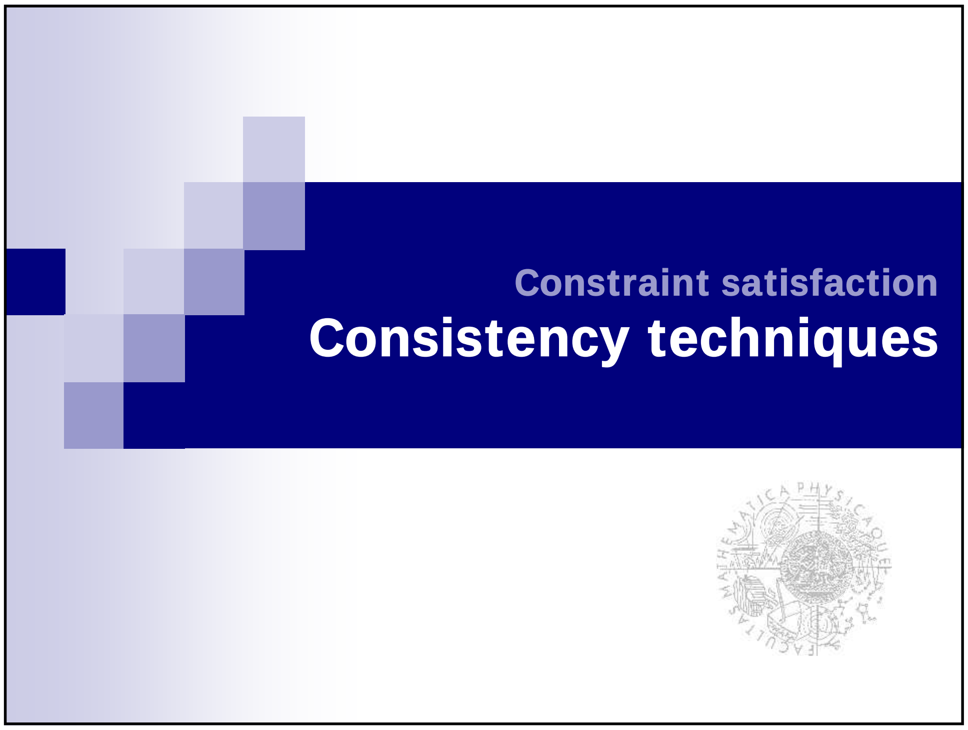
Constraint model:

- variable R_i describes the row of the rectangle i
  - R_i in {1,…,H}
- variable C_i describes the first column occupied by the rectangle i
  - C_i in {1,…,W- w_i+1}
- non-overlap constraint
  - $\forall i \neq j \quad (R_i = R_j) \Rightarrow (C_i + w_i < C_j \vee C_j + w_j < C_i)$



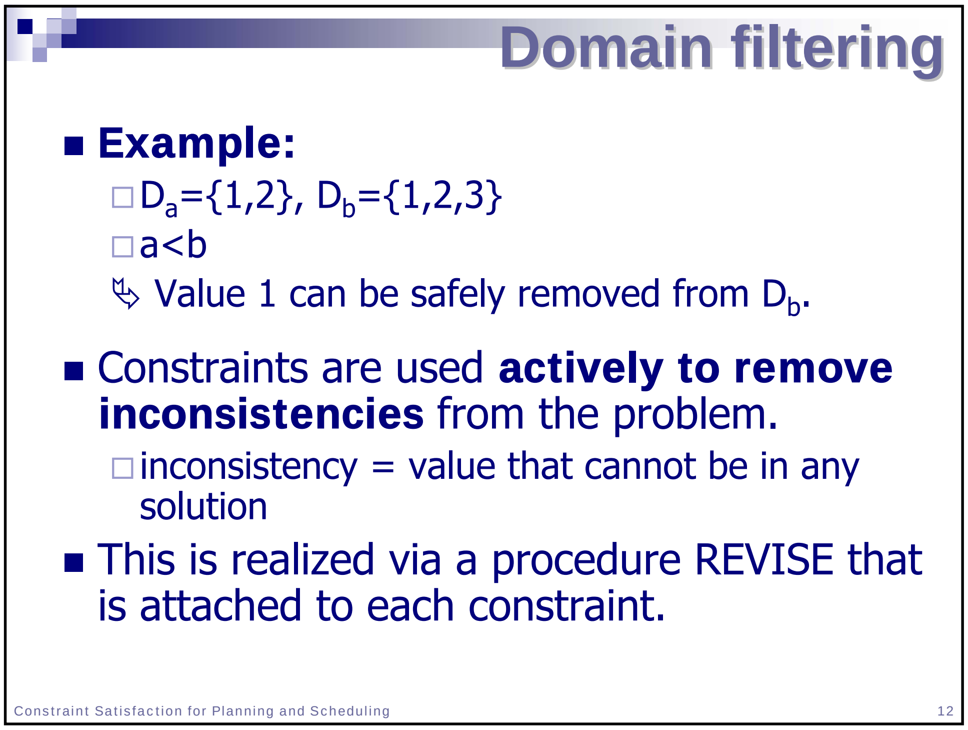
Two or more?

- **Binary constraint satisfaction**
  - only binary constraints
  - any CSP is convertible to a binary CSP
    - **dual encoding (Stergiou & Walsh, 1990)**
      swapping the role of variables and constraints
- **Boolean constraint satisfaction**
  - only Boolean (two valued) domains
  - any CSP is convertible to a Boolean CSP
    - **SAT encoding**
      Boolean variable indicates whether a given value is assigned to the variable



Constraint satisfaction

Consistency techniques



Domain filtering

- **Example:**
  - $D_a=\{1,2\}$, $D_b=\{1,2,3\}$
  - $a<b$
  - ↳ Value 1 can be safely removed from D_b.
- Constraints are used **actively to remove inconsistencies** from the problem.
  - inconsistency = value that cannot be in any solution
- This is realized via a procedure REVISE that is attached to each constraint.

Arc consistency

- We say that a constraint is **arc consistent** (AC) if for any value of the variable in the constraint there exists a value for the other variable(s) in such a way that the constraint is satisfied (we say that the value is supported).
- A **CSP is arc consistent** if all the constraints are arc consistent.

Making problems AC

- How to establish arc consistency in CSP?
- Every constraint must be revised!

Example: X in [1,..,6], Y in [1,..,6], Z in [1,..,6], X<Y, Z<X-2

$$\begin{tikzcd}
\boxed{\begin{array}{l}\text{X in [1,..,6]}\\ \text{Y in [1,..,6]}\\ \text{Z in [1,..,6]}\end{array}} \arrow[r, "X<Y"] & \boxed{\begin{array}{l}\text{X in [1,..,5]}\\ \text{Y in [2,..,6]}\\ \text{Z in [1,..,6]}\end{array}} \arrow[r, "Z<X-2"] & \boxed{\begin{array}{l}\text{X in [4,5]}\\ \text{Y in [2,..,6]}\\ \text{Z in [1,2]}\end{array}} \arrow[r, "X<Y"] & \boxed{\begin{array}{l}\text{X in [4,5]}\\ \text{Y in [5,6]}\\ \text{Z in [1,2]}\end{array}}
\end{tikzcd}$$

↳ Doing revision of every constraint just once is not enough!

- Revisions must be repeated until any domain is changed (AC-1).

Mackworth (1977)

Algorithm AC-3

- Uses a **queue of constraints** that should be revised
- When a domain of variable is changed, only the constraints over this variable are added back to the queue for re-revision.

```
procedure AC-3(V,D,C)
    $Q \leftarrow C$
    while non-empty Q do
        select c from Q
        $D' \leftarrow c.REVISE(D)$
        if any domain in D' is empty then return (fail,D')
        $Q \leftarrow Q \cup \{c' \in C \mid \exists x \in var(c')\ D'_x \neq D_x\} - \{c\}$
        $D \leftarrow D'$
    end while
    return (true,D)
end AC-3
```

AC-3 in practice

- Uses a **queue of variables** with changed domains.
  - Users may specify for each constraint when the constraint revision should be done depending on the domain change.
- The algorithm is sometimes called AC-8.

```
procedure AC-8(V,D,C)
    $Q \leftarrow V$
    while non-empty Q do
        select v from Q
        for $c \in C$ such that v is constrained by c do
            $D' \leftarrow c.REVISE(D)$
            if any domain in D' is empty then return (fail,D')
            $Q \leftarrow Q \cup \{u \in V \mid D'_u \neq D_u\}$
            $D \leftarrow D'$
        end for
    end while
    return (true,D)
end AC-8
```

Other AC algorithms

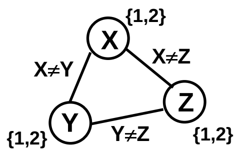
- **AC-4 (Mohr & Henderson, 1986)**
  - computes sets of supporting values
  - optimal worst-case time complexity $O(ed^2)$
- **AC-5 (Van Hentenryck, Deville, Teng, 1992)**
  - generic AC algorithm covering both AC-4 and AC-3
- **AC-6 (Bessière, 1994)**
  - improves AC-4 by remembering just one support
- **AC-7 (Bessière, Freuder, Régin, 1999)**
  - improves AC-6 by exploiting symmetry of the constraint
- **AC-2000 (Bessière & Régin, 2001)**
  - an adaptive version of AC-3 that either looks for a support or propagates deletions
- **AC-2001 (Bessière & Régin, 2001)**
  - improvement of AC-3 to get optimality (queue of variables)
- **AC-3.1 (Zhang & Yap, 2001)**
  - improvement of AC-3 to get optimality (queue of constraints)

…

Path consistency

Arc consistency does not detect all inconsistencies!

Let us look at several constraints together!



- The path $(V_0, V_1, \dots, V_m)$ is **path consistent** iff for every pair of values $x \in D_0$ a $y \in D_m$ satisfying all the binary constraints on V_0, V_m there exists an assignment of variables $V_1, \dots, V_{m-1}$ such that all the binary constraints between the neighboring variables V_i, V_{i+1} are satisfied.
- **CSP is path consistent** iff every path is consistent.

Some notes:

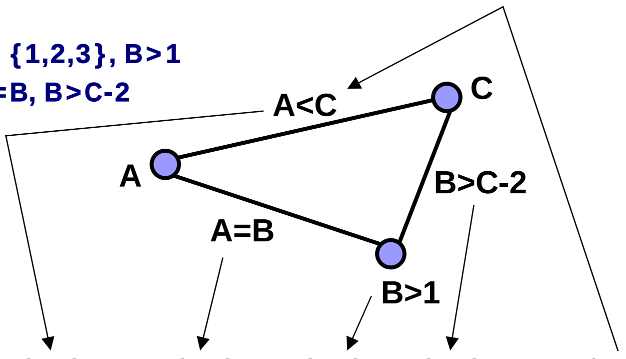
- only the **constraints between the neighboring variables** must be satisfied
- it is enough to explore **paths of length 2** (Montanary, 1974)

Path revision

Constraints represented extensionally via matrices.
Path consistency is realized via matrix operations

Example:

- **A,B,C in {1,2,3}, B>1**
- **A<C, A=B, B>C-2**



$$\begin{pmatrix}011\\001\\000\end{pmatrix} \;\&\; \begin{pmatrix}100\\010\\001\end{pmatrix} * \begin{pmatrix}000\\010\\001\end{pmatrix} * \begin{pmatrix}110\\111\\111\end{pmatrix} = \begin{pmatrix}000\\001\\000\end{pmatrix}$$

Mackworth (1977)

Algorithm PC-1

- How to make the path (i,k,j) consistent?
  - $R_{ij} \leftarrow R_{ij} \,\&\, (R_{ik} * R_{kk} * R_{kj})$
- How to make a CSP path consistent?
  - Repeated revisions of paths (of length 2) while any domain changes.

```
procedure PC-1(Vars,Constraints)
    $n \leftarrow |Vars|$, $Y^n \leftarrow$ Constraints
    repeat
        $Y^0 \leftarrow Y^n$
        for k = 1 to n do
            for i = 1 to n do
                for j = 1 to n do
                    $Y^k_{ij} \leftarrow Y^{k-1}_{ij} \,\&\, (Y^{k-1}_{ik} * Y^{k-1}_{kk} * Y^{k-1}_{kj})$
    until $Y^n = Y^0$
    Constraints $\leftarrow Y^0$
end PC-1
```

Other PC algorithms

- **PC-2 (Mackworth, 1977)**
  - revises only paths (i,k,j) for i≤j and after change, only relevant paths revised
- **PC-3 (Mohr & Henderson, 1986)**
  - based on principles of AC-4 (remembering supports), but it is not sound (it can delete a consistent value)
- **PC-4 (Han & Lee, 1988)**
  - a corrected version of PC-3 (incompatible pairs are deleted)
- **PC-5 (Singh, 1995)**
  - based on principles of AC-6 (remember just one support)

Think globally

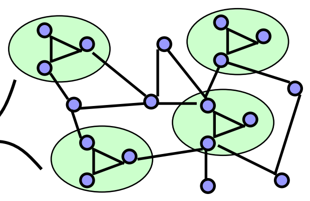
CSP describes the problem locally:

the constraints restrict small sets of variables

+ heterogeneous real-life constraints

- missing global view

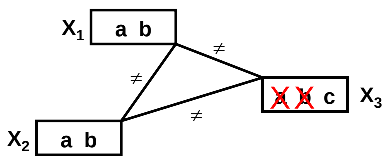
↳ weaker domain filtering



Global constraints

- **global reasoning over a local sub-problem**
- **using semantic information to improve time efficiency or pruning power**

Example:

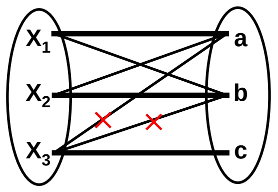


- **local (arc) consistency deduces no pruning**
- **but some values can be removed**

Régin (1994)

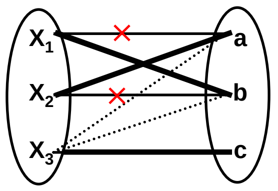
Inside all-different

- a set of binary inequality constraints among all variables
  $X_1 \neq X_2, X_1 \neq X_3, \ldots, X_{k-1} \neq X_k$
- $\text{all\_different}(\{X_1,\ldots,X_k\}) = \{(d_1,\ldots,d_k) \mid \forall i\ d_i \in D_i\ \&\ \forall i \neq j\ d_i \neq d_j\}$
- better pruning based on matching theory over bipartite graphs



Initialization:

1. compute maximum matching
2. remove all edges that do not belong to any maximum matching



Propagation of deletions (X1≠a):

1. remove discharged edges
2. compute new maximum matching
3. remove all edges that do not belong to any maximum matching

Prosser et al. (2000)

Singleton consistency

- Can we strengthen any consistency technique?
- YES! Let's assign a value and make the rest of the problem consistent.

- **CSP P is singleton A-consistent** for some notion of A-consistency iff for every value *h* of any variable *X* the problem $P_{|X=h|}$ is A-consistent.

Features:

+ we **remove only values** from variable's domain
+ **easy implementation** (meta-programming)
– could be **slow** (be careful when using SC)

Constraint satisfaction

Search techniques

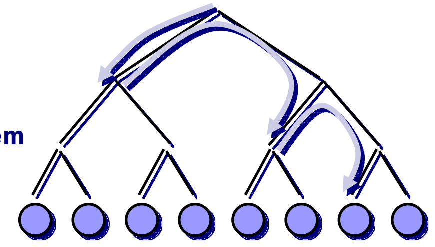
Search / Labeling

Consistency techniques are (usually) incomplete.

↳ **We need a search algorithm to resolve the rest!**

Labeling

- depth-first search
  - assign a value to the variable
  - propagate = make the problem locally consistent
  - backtrack upon failure



- X in 1..5 ≈ X=1 ∨ X=2 ∨ X=3 ∨ X=4 ∨ X=5

In general, search algorithm resolves remaining disjunctions!

- X=1 ∨ X≠1 (standard labeling)
- X<3 ∨ X≥3 (domain splitting)
- X<Y ∨ X≥Y (variable ordering)

Labeling skeleton

- Search is combined with consistency techniques that prune the search space
- **Look-ahead technique**

```
procedure labeling(V,D,C)
        if all variables from V are assigned then return V
        select not-yet assigned variable x from V
        for each value v from Dx do
                (TestOK,D') ← consistent(V,D,C∪{x=v})
                if TestOK=true then R ← labeling(V,D',C)
                if R ≠ fail then return R
        end for
        return fail
end labeling
```

Branching schemes

- **What variable should be assigned first?**
  - **first-fail principle**
    - prefer the variable whose instantiation will lead to a failure with the highest probability
    - variables with the smallest domain first
    - most constrained variables first
  - defines the **shape of the search tree**
- **What value should be tried first?**
  - **succeed-first principle**
    - prefer the values that might belong to the solution with the highest probability
    - values with more supporters in other variables
    - usually problem dependent
  - defines the **order of branches** to be explored

Systematic search

- **Chronological backtracking**
  - upon failure backtrack to last but one variable
- **Backjumping (Gaschnig, 1979)**
  - upon failure jump back to a conflicting variable
- **Dynamic backtracking (Ginsberg, 1993)**
  - upon failure un-assign only the conflicting variable
- **Backmarking (Haralick & Elliot, 1980)**
  - remember conflicts (no-goods) and use them in subsequent search

Incomplete search

- A **cutoff limit** to stop exploring a (sub-)tree
  - some branches are skipped → incomplete search
- When no solution found, **restart** with enlarged cutoff limit.
- **Bounded Backtrack Search (Harvey, 1995)**
  - restricted number of backtracks
- **Depth-bounded Backtrack Search (Cheadle et al., 2003)**
  - restricted depth where alternatives are explored
- **Iterative Broadening (Ginsberg and Harvey, 1990)**
  - restricted breadth in each node
  - still exponential!
- **Credit Search (Cheadle et al., 2003)**
  - limited credit for exploring alternatives
  - credit is split among the alternatives

Heuristics in search

- **Observation 1:**
  The **search space** for real-life problems is so **huge** that it cannot be fully explored.
- **Heuristics** - a guide of search
  - they recommend a value for assignment
  - quite often lead to a solution
- What to do upon a **failure of the heuristic**?
  - BT cares about the end of search (a bottom part of the search tree)
    so it rather repairs later assignments than the earliest ones
    thus BT assumes that the heuristic guides it well in the top part
- **Observation 2:**
  The **heuristics** are **less reliable in the earlier parts** of the search tree (as search proceeds, more information is available).
- **Observation 3:**
  The number of **heuristic violations** is usually **small.**

Discrepancies

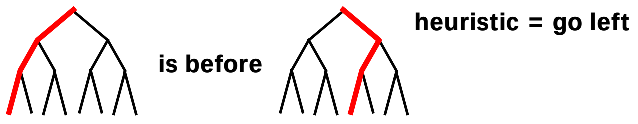
Discrepancy

= the heuristic is not followed

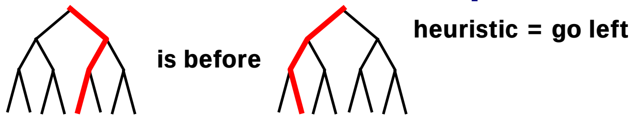
Basic principles of discrepancy search:

change the order of branches to be explored

- prefer branches with **less discrepancies**

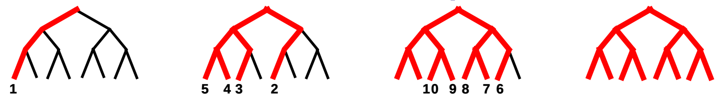


- prefer branches with **earlier discrepancies**

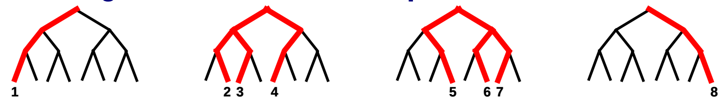


Discrepancy search

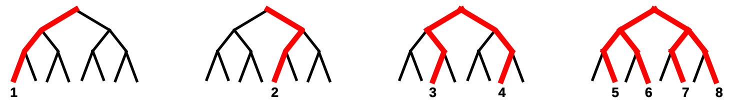
- **Limited Discrepancy Search (Harvey & Ginsberg, 1995)**
  - restricts a maximal number of discrepancies in the iteration



- **Improved LDS (Korf, 1996)**
  - restricts a given number of discrepancies in the iteration



- **Depth-bounded Discrepancy Search (Walsh, 1997)**
  - restricts discrepancies till a given depth in the iteration

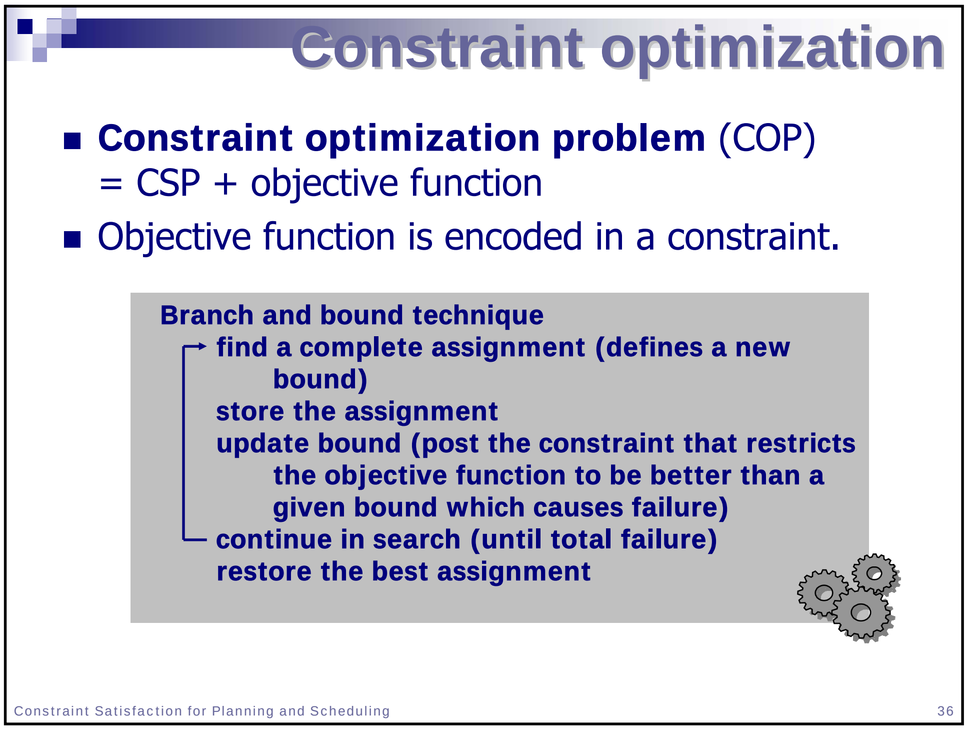


- **...**

*** heuristic = go left**

Constraint satisfaction

Extensions



Constraint optimization

- **Constraint optimization problem** (COP)
  = CSP + objective function
- Objective function is encoded in a constraint.

```
Branch and bound technique
  ┌→ find a complete assignment (defines a new
  │        bound)
  │  store the assignment
  │  update bound (post the constraint that restricts
  │        the objective function to be better than a
  │        given bound which causes failure)
  └─ continue in search (until total failure)
     restore the best assignment
```

Soft problems

- **Hard constraints** express restrictions.
- **Soft constraints** express preferences.
- Maximizing the number of satisfied soft constraints
- Can be solved via **constraint optimization**
  - Soft constraints are encoded into an objective function
- Special frameworks for soft constraints
  - **Constraint hierarchies** (Borning et al., 1987)
    - symbolic preferences assigned to constraints
  - **Semiring-based CSP** (Bistarelli, Montanary, and Rossi, 1997)
    - semiring values assigned to tuples (how well/badly a tuple satisfies the constraint)
    - soft constraint propagation

Dynamic problems

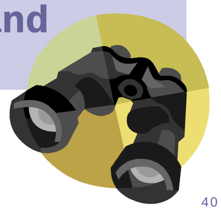
- **Internal dynamics (Mittal & Falkenhainer, 1990)**
  - planning, configuration
  - variables can be active or inactive, only active variables are instantiated
  - **activation (conditional) constraints**
    - cond(x_1,…, x_n) $\rightarrow$ activate(x_j)
  - solved like a standard CSP (a special value in the domain to denote inactive variables)
- **External dynamics (Dechter & Dechter, 1988)**
  - on-line systems
  - **sequence of static CSPs**, where each CSP is a result of the addition or retraction of a constraint in the preceding problem
  - Solving techniques:
    - reusing solutions
    - **maintaining dynamic consistency** (DnAC-4, DnAC-6, AC|DC).

Constraints for planning and scheduling

Terminology

"The **planning task** is to find out a sequence of actions that will transfer the initial state of the world into a state where the desired goal is satisfied"

"The **scheduling task** is to allocate known activities to available resources and time respecting capacity, precedence (and other) constraints"



Constraints and P&S

- **Scheduling problem is static**

  all activities are known

  ↳variables and constraints are know

  ↳ standard CSP is applicable

- **Planning problem is internally dynamic**

  activities are unknown in advance

  ↳ variables describing activities are unknown

  Solution (Kautz & Selman, 1992):

  - **finding a plan of a given length is a static problem**
  - standard CSP is applicable there!

P&S via CSP?

- **Exploiting state of the art constraint solvers!**
  - □ faster solver ⇒ faster planner
- **Constraint model is extendable!**
  - □ it is possible immediately to add other variables and constraints
  - □ modeling numerical variables, resource and precedence constraints for planning
  - □ adding side constraints to base scheduling models
- **Scheduling algorithms encoded in the filtering algorithms for constraints!**
  - □ fast algorithms accessible to constraint models

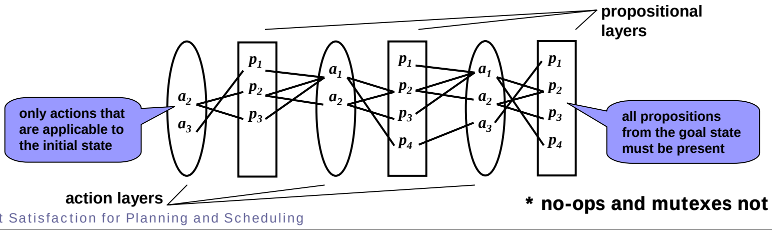
Constraints in P&S

Constraint models

Blum & Furst (1997)

Planning graph

- **Planning graph** is a layered graph representing STRIPS-like plans of a given length.
- **nodes** = propositions + actions
- Interchanging **propositional and action layers**
  - action is connected to its **preconditions** in the previous layer and to its **add effects** in the next layer
  - **delete effect** is modeled via **action mutex** (actions cannot deleting and adding the same effect cannot be active at the same layer)
  - **propositional mutexes** generated from action mutexes
  - **no-op actions** (same pre-condition as the add effect)



* no-ops and mutexes not displayed

Do & Kambhampati (2000)

Activity-based model

- **Planning graph** of a given length is a static structure that can be **encoded as a CSP**.
- Constraint technology is used for **plan extraction.**

CSP model:

- **Variables**
  - propositional nodes $P_{j,m}$ (proposition p_j in layer m)
  - only propositional layers are indexed
- **Domain**
  - activities that has a given proposition as an add effect
  - $\perp$ for inactive proposition
- **Constraints**
  - connect add effects with preconditions
  - mutexes

Do & Kambhampati (2000)

Activity-based model

Constraints

$\mathbf{P_{4,m} = \mathit{a} \Rightarrow P_{1,m-1} \neq \perp \;\&\; P_{2,m-1} \neq \perp \;\&\; P_{3,m-1} \neq \perp}$

- action *a* has preconditions p_1, p_2, p_3 and an add effect p_4
- the constraint is added for every add effect of *a*

$\mathbf{P_{i,m} = \perp \vee P_{j,m} = \perp}$

- propositional mutex between propositions p_i and p_j

$\mathbf{P_{i,m} \neq \mathit{a} \vee P_{j,m} \neq \mathit{b}}$

- actions *a* and *b* are marked mutex and p_i is added by *a* and p_j is added by *b*

$\mathbf{P_{i,k} \neq \perp}$

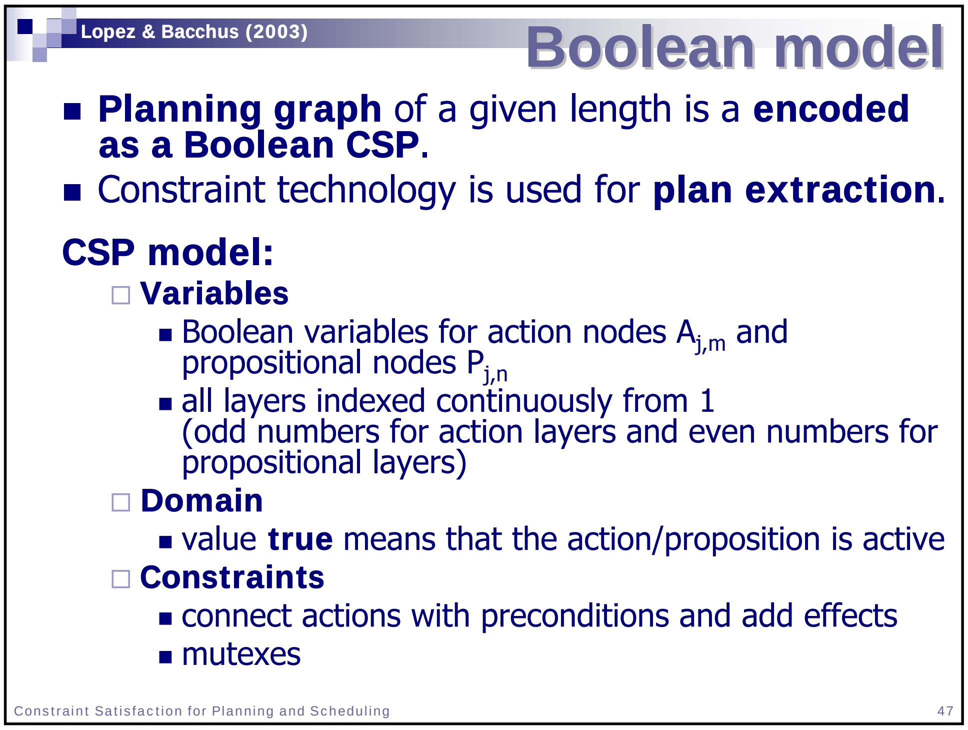
- p_i is a goal proposition and k is the index of the last layer

no parallel actions

- maximally one action is assigned to variables in each layer

no void layers

- at least one action different from a no-op action is assigned to variables in a given layer


Lopez & Bacchus (2003)

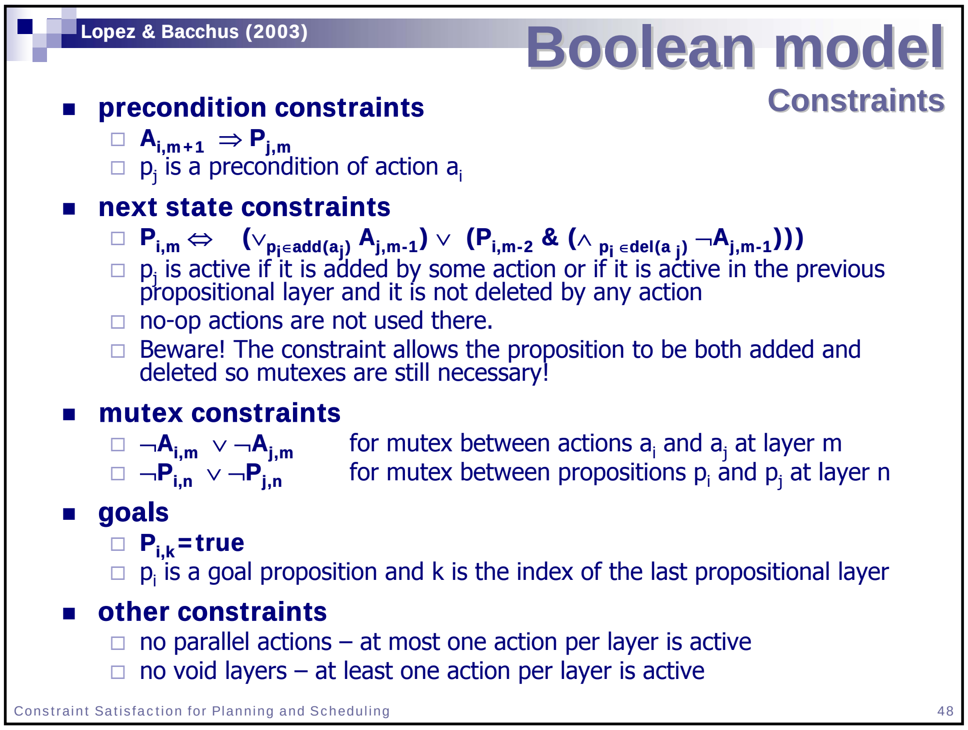
Boolean model

- **Planning graph** of a given length is a **encoded as a Boolean CSP.**
- Constraint technology is used for **plan extraction.**

CSP model:

- **Variables**
  - Boolean variables for action nodes $A_{j,m}$ and propositional nodes $P_{j,n}$
  - all layers indexed continuously from 1 (odd numbers for action layers and even numbers for propositional layers)
- **Domain**
  - value **true** means that the action/proposition is active
- **Constraints**
  - connect actions with preconditions and add effects
  - mutexes

Constraint Satisfaction for Planning and Scheduling 47


Lopez & Bacchus (2003)

Boolean model

Constraints

- **precondition constraints**
  - $\mathbf{A_{i,m+1} \Rightarrow P_{j,m}}$
  - p_j is a precondition of action a_i
- **next state constraints**
  - $\mathbf{P_{i,m} \Leftrightarrow (\vee_{p_i \in add(a_j)} A_{j,m-1}) \vee (P_{i,m-2} \,\&\, (\wedge_{p_i \in del(a_j)} \neg A_{j,m-1}))}$
  - p_i is active if it is added by some action or if it is active in the previous propositional layer and it is not deleted by any action
  - no-op actions are not used there.
  - Beware! The constraint allows the proposition to be both added and deleted so mutexes are still necessary!
- **mutex constraints**
  - $\mathbf{\neg A_{i,m} \vee \neg A_{j,m}}$ for mutex between actions a_i and a_j at layer m
  - $\mathbf{\neg P_{i,n} \vee \neg P_{j,n}}$ for mutex between propositions p_i and p_j at layer n
- **goals**
  - $\mathbf{P_{i,k}=true}$
  - p_i is a goal proposition and k is the index of the last propositional layer
- **other constraints**
  - no parallel actions – at most one action per layer is active
  - no void layers – at least one action per layer is active

Constraint Satisfaction for Planning and Scheduling 48

Scheduling model

- **Scheduling problem** is static so it can be directly **encoded as a CSP.**
- Constraint technology is used for **full scheduling**.

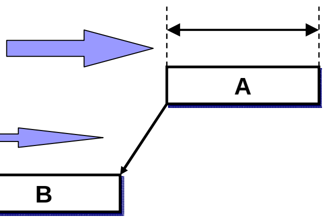
CSP model:

- **Variables**
  - position of activity A in time and space
  - time allocation: **start(A), [p(A), end(A)]**
  - resource allocation: **resource(A)**
- **Domain**
  - **ready times** and **deadlines** for the time variables
  - **alternative resources** for the resource variables
- **Constraints**
  - sequencing and resource capacities

Scheduling model

Constraints

- **Time relations**
  - **start(A)+p(A)=end(A)**
  - sequencing
    - B<<A
    - ↳ **end(B)≤start(A)**



- **Resource capacity constraints**
  - unary resource (activities cannot overlap)
    - A<<B ∨ B<<A
    - ↳ **end(A)≤start(B) ∨ end(B)≤start(A)**

Constraints in P&S

Filtering algorithms

Resources

- **Resources are used in slightly different meanings in planning and scheduling!**
- **scheduling**
  - resource
    = a **machine** (space) for processing the activity
- **planning**
  - resource
    = consumed/produced **material** by the activity
  - resource in the scheduling sence is often handled via logical precondition (e.g. hand is free)

Resource types

- **Unary resource**
  - a single activity can be processed at given time
- **Cumulative resource**
  - several activities can be processed in parallel if capacity is not exceeded.
- **Producible/consumable resource**
  - activity consumes/produces some quantity of the resource
  - minimal capacity is requested (consumption) and maximal capacity cannot be exceeded (production)

Unary resources

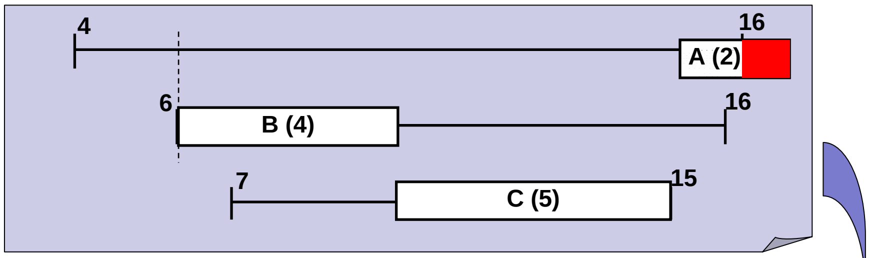
- Activities cannot overlap
- We assume that activities are uninterruptible
  - uninterruptible activity occupies the resource from its start till its completion
- A simple model with disjunctive constraints
  - **A<<B $\vee$ B<<A**

    end(A)$\leq$start(B) $\vee$ end(B)$\leq$start(A)

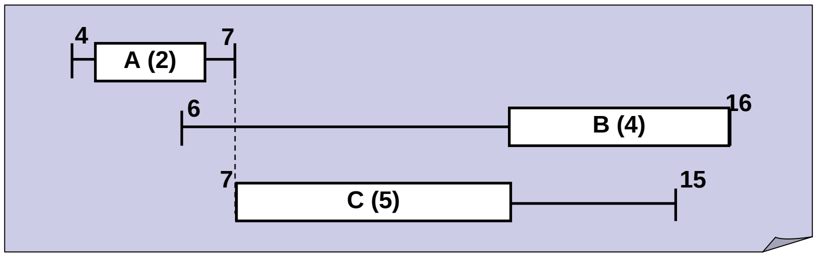
Baptiste & Le Pape (1996)

Edge finding

What happens if activity A is not processed first?



Not enough time for A, B, and C and thus A must be first!



Baptiste & Le Pape (1996)

Edge finding rules

- **Some definitions:**

  $p(\Omega) = \sum_{A \in \Omega} p(A)$, where Ω is a set of activities

  $\min(start(\Omega)) = \min_{A \in \Omega} \{start(A)\}$

- **The rules:**

  $\min(start(\Omega)) + p(\Omega) + p(A) > \max(end(\Omega \cup \{A\})) \Rightarrow A \ll \Omega$

  $A \ll \Omega \Rightarrow end(A) \leq \min\{ \max(end(\Omega')) - p(\Omega') \mid \Omega' \subseteq \Omega \}$

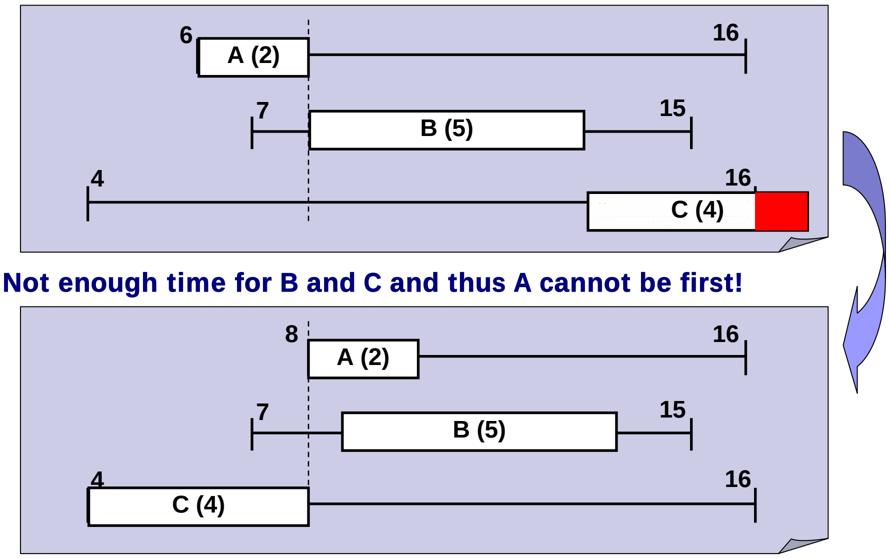
- **In practice:**
  - instead of Ω use so called **task intervals** [A,B]

    $\{C \mid \min(start(A)) \leq \min(start(C))\ \&\ \max(end(C)) \leq \max(end(B))\}$
  - time complexity O(n^3)

Torres & Lopez (2000)

Not-first/not-last

What happens if activity A is processed first?



Not enough time for B and C and thus A cannot be first!

Torres & Lopez (2000)

Not-first/not-last rules

- **Not-first rules:**

  $\min(start(A)) + p(\Omega) + p(A) > \max(end(\Omega)) \Rightarrow \neg A \ll \Omega$

  $\neg A \ll \Omega \Rightarrow start(A) \geq \min\{ end(B) \mid B \in \Omega \}$

- **Not-last (symmetrical) rules:**

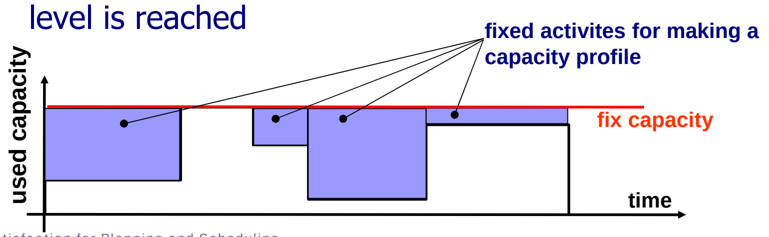
  $\min(start(\Omega)) + p(\Omega) + p(A) > \max(end(A)) \Rightarrow \neg \Omega \ll A$

  $\neg \Omega \ll A \Rightarrow end(A) \leq \max\{ start(B) \mid B \in \Omega \}$

- **In practice:**
  - it is possible to use selected sets Ω only
  - time complexity $O(n^2)$

Cumulative resources

- Each **activity uses some capacity** of the resource – **cap(A)**.
- Activities can be **processed in parallel** if a resource capacity is not exceeded.
- Resource capacity **may vary in time**
  - modeled via fix capacity over time and fixed activities consuming the resource until the requested capacity level is reached

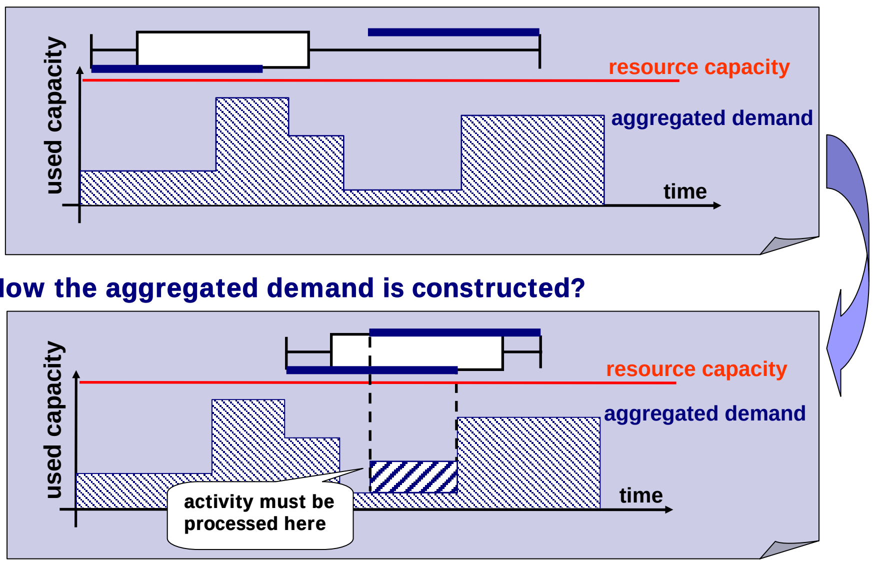


Aggregated demands

Baptiste et al. (2001)

Where is enough capacity for processing the activity?

How the aggregated demand is constructed?



Baptiste et al. (2001)

Timetable constraint

- How to ensure that capacity is not exceed at any time point?*

$$\forall t \sum_{start(A_i) \le t \le end(A_i)} cap(A_i) \le MaxCapacity$$

- **Timetable** for the activity A is a set of Boolean variables **X(A,t)** indicating whether A is processed in time t.

$$\forall t \sum_{A_i} X(A_i,t) \cdot cap(A_i) \le MaxCapacity$$

$$\forall t,i \quad start(A_i) \le t \le end(A_i) \Leftrightarrow X(A_i,t)$$

*** discrete time is expected**

Alternative resources

- **How to model alternative resources for a given activity?**
- Use a **duplicate activity** for each resource.
  - duplicate activity participates in a respective resource constraint but does not restrict other activities there
    - failure means removing the resource from the domain of variable res(A)
    - deleting the resource from the domain of variable res(A) means „deleting" the respective duplicate activity
  - original activity participates in precedence constraints (e.g. within a job)
  - restricted times of duplicate activities are propagated to the original activity and vice versa.

Relative ordering

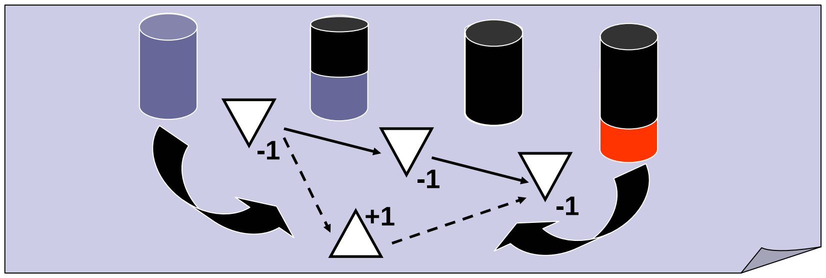
When time is relative (ordering of activities)

then edge-finding and aggregated demand deduce nothing

We can still use information about ordering of activities and resource production/consumption!

Example:

Reservoir: activities consume and supply items



Cesta & Stella (1997)

Resource profiles

- Activity A „produces" **prod(A)** quantity:
  - positive number means **production**
  - negative number means **consumption**
- **Optimistic resource profile (orp)**
  - maximal possible level of the resource when A is processed
  - activities known to be before A are assumed together with the production activities that can be before A

$orp(A) = InitLevel + prod(A) + \sum_{B<<A} prod(B) + \sum_{B??A\ \&\ prod(B)>0} prod(B)$

- **Pesimistic resource profile (prp)**
  - minimal possible level of the resource when A is processed
  - activities known to be before A are assumed together with the consumption activities that can be before A

$prp(A) = InitLevel + prod(A) + \sum_{B<<A} prod(B) + \sum_{B??A\ \&\ prod(B)<0} prod(B)$

***B??A means that order of A and B is unknown yet**

Cesta & Stella (1997)

orp filtering

- $orp(A) < MinLevel \Rightarrow fail$
  - "despite the fact that all production is planned before A, the minimal required level in the resource is not reached"
- $orp(A) - prod(B) - \sum_{B<<C\ \&\ C??A\ \&\ prod(C)>0} prod(C) < MinLevel \Rightarrow B<<A$,
  for any B such that B??A and *prod*(B)>0
  - "if production in B is planned after A and the minimal required level in the resource is not reached then B must be before A"

Cesta & Stella (1997)

prp filtering

- $prp(A) > MaxLevel \Rightarrow \text{fail}$
  - "despite the fact that all consumption is planned before A, the maximal required level (resource capacity) in the resource is exceeded"
- $prp(A) - prod(B) - \sum_{B<<C\ \&\ C??A\ \&\ prod(C)<0} prod(C) > MaxLevel \Rightarrow B<<A$,
  for any B such that B??A and *prod*(B)<0
  - "if consumption in B is planned after A and the maximal required level in the resource is exceeded then B must be before A"

Temporal problems

Simple temporal relations:

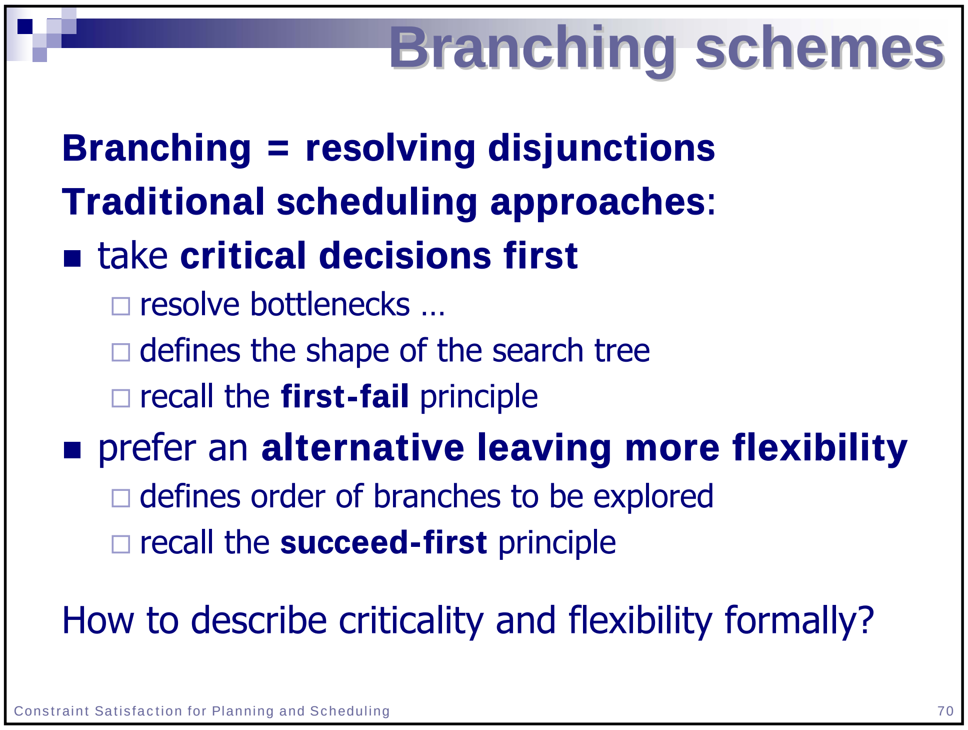
- *time(B)-time(A)* $\leq d_{A,B}$
- *time(A)-time(B)* $\leq d_{B,A}$

Solvable in polynomial time (Dechter et al., 1991)

- **Floyd-Warshall‘ algorithm** for computing shortest paths
- A special version of **path consistency**
  - sometimes encoded in a global constraint

Constraints in P&S

Search strategies



Branching schemes

Branching = resolving disjunctions

Traditional scheduling approaches:

- take **critical decisions first**
  - resolve bottlenecks …
  - defines the shape of the search tree
  - recall the **first-fail** principle
- prefer an **alternative leaving more flexibility**
  - defines order of branches to be explored
  - recall the **succeed-first** principle

How to describe criticality and flexibility formally?

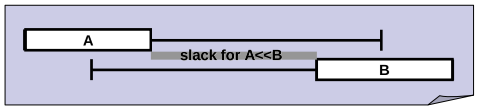
Smith and Cheng (1993)

Slack

Slack is a formal description of flexibility

- Slack for **a given order of two activities**

  „free time for shifting the activities"



  slack(A<<B) = max(end(B))-min(start(A))-p({A,B})

- Slack for **two activities**

  slack({A,B}) = max{slack(A<<B),slack(B<<A)}

- Slack for **a group of activities**

  slack(Ω) = max(end(Ω)) - min(start(Ω)) - p(Ω)

Smith and Cheng (1993)

Order branching

A<<B $\vee$ $\neg$**A<<B**

- Which activities should be ordered first?
  - □ the most critical pair (first-fail)
  - □ the pair with **the minimal slack({A,B})**
- What order should be selected?
  - □ the most flexible order (succeed-first)
  - □ the order with **the maximal slack(A??B)**
- O(n^2) choice points

Baptiste et al. (1995)

First/last branching

$(A \ll \Omega \vee \neg A \ll \Omega)$ or $(\Omega \ll A \vee \neg \Omega \ll A)$

- Should we look for first or last activity?
  - select a **smaller set** among possible first or possible last activities (first-fail)
- What activity should be selected?
  - If first activity is being selected then the activity with the **smallest min(start(A))** is preferred.
  - If last activity is being selected then the activity with the **largest max(end(A))** is preferred.
- O(n) choice points

Resource slack

- **Resource slack** is defined as a slack of the set of activities processed by the resource.
- **How to use a resource slack?**
  - choosing a resource on which **the activities will be ordered** first
    - resource with a minimal slack (**bottleneck**) preferred
  - choosing a resource on which the **activity will be allocated**
    - resource with a maximal slack (**flexibility**) preferred

Conclusions

Constraint solvers

- It is not necessary to program all the presented techniques from scratch!
- Use existing constraint solvers (packages)!
  - provide **implementation of data structures** for modelling variables' domains and constraints
  - provide a basic **consistency framework** (AC-3)
  - provide **filtering algorithms** for many constraints (including global constraints)
  - provide basic **search strategies**
  - usually **extendible** (new filtering algorithms, new search strategies)

SICStus Prolog

www.sics.se/sicstus

- a strong Prolog system with libraries for solving constraints (FD, Boolean, Real)
- arithmetical, logical, and some global constraints
  - an interface for defining new filtering algorithms
- depth-first search with customizable value and variable selection (also optimization)
  - it is possible to use Prolog backtracking
- **support for scheduling**
  - constraints for **unary** and **cumulative** resources
  - first/last **branching scheme**

ECLiPSe

www.icparc.ic.ac.uk/eclipse

- a Prolog system with libraries for solving constraints (FD, Real, Sets)
- integration with OR packages (CPLEX, XPRESS-MP)
- arithmetical, logical, and some global constraints
  - an interface for defining new filtering algorithms
- Prolog depth-first search (also optimisation)
- a repair library for implementing local search techniques
- **support for scheduling**
  - constraints for **unary** and **cumulative** resources
  - **„probing“** using a linear solver
  - Gantt chart and network viewers

CHIP

`www.cosytec.com`

- a constraint solver in C with Prolog as a host language, also available as C and C++ libraries
- popularized the concept of global constraints
  - different, order, resource, tour, dependency
- it is hard to go beyond the existing constraints
- **support for scheduling**
  - constraints for **unary** and **cumulative** resources
  - a **precedence** constraint (several cumulatives with the precedence graph)

ILOG

`www.ilog.com/products`

- the largest family of optimisation products as C++ (Java) libraries
- ILOG Solver provides basic constraint satisfaction functionality
- **ILOG Scheduler** is an add-on to the Solver with classes for scheduling objects
  - activities
  - **state, cumulative, unary, energetic** resources; **reservoirs**
    - alternative resources
  - **resource**, **precedence**, and **bound** constraints

Mozart

www.mozart-oz.org

- a self contained development platform based on the Oz language
- mixing logic, constraint, object-oriented, concurrent, and multi-paradigm programming
- **support for scheduling**
  - constraints for **unary** and **cumulative** resources
  - first/last **branching scheme**
  - **search visualization**

Summary

Basic constraint satisfaction framework:

- **local consistency** connecting filtering algorithms for individual constraints
- **search** resolves remaining disjunctions

Problem solving:

- **declarative modeling** of problems as a CSP
- **dedicated algorithms** encoded in constraints
- special **search strategies**