SP2

Constraint Processing for Planning & Scheduling

Roman Barták
Charles University, Prague (CZ)
roman.bartak@mff.cuni.cz

What and why?

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

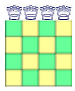
Tutorial outline

- **Constraint satisfaction in a nutshell**
  - domain filtering and local consistencies
  - search techniques
  - extensions of a basic constraint satisfaction problem
- **Constraints for planning**
  - constraint models
  - temporal reasoning
- **Constraints for scheduling**
  - a base constraint model
  - resource constraints
  - branching schemes
- **Conclusions**
  - a short survey on constraint solvers

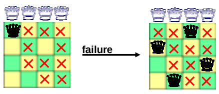
Constraint satisfaction
in a nutshell

Constraints
starting simple

Modeling (problem formulation)
- N queens problem
- **decision variables** for positions of queens in rows
  r(i) in {1,...,N}
- **constraints** describing (non-)conflicts
  $\forall i \neq j \quad r(i) \neq r(j) \ \& \ |i-j| \neq |r(i)-r(j)|$



Search and inference (propagation)
- **backtracking** (assign values and return upon failure)
- infer consequences of decisions via maintaining **consistency** of constraints



Constraint satisfaction

based on **declarative problem description** via:
- **variables with domains** (sets of possible values)
  describe **decision points** of the problem with possible **options** for the decisions
  e.g. the start time of activity with time windows
- **constraints** restricting combinations of values,
  describe arbitrary **relations** over the set of variables
  e.g. end(A) < start(B)

A **feasible solution** to a constraint satisfaction problem is a complete assignment of variables satisfying all the constraints.

An **optimal solution** to a CSP is a feasible solution minimizing/maximizing a given objective function.

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 = a value that cannot be in any solution
- This is realized via a procedure FILTER 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).
  Unsupported values are filtered out of the domain.
- A **CSP** is **arc consistent** if all the constraints are arc consistent.

Making problems AC

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

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

X in [1,..,6] Y in [1,..,6] Z in [1,..,6]	$\xrightarrow{X<Y}$	X in [1,..,5] Y in [2,..,6] Z in [1,..,6]	$\xrightarrow{Z<X-2}$	X in [4,5] Y in [2,..,6] Z in [1,2]	$\xrightarrow{X<Y}$	X in [4,5] Y in [5,6] Z in [1,2]

➪ Filtering every constraint just once is not enough!

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

Mackworth (1977)

Algorithm AC-3

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

```
procedure AC-3(V,D,C)
    Q ← C
    while non-empty Q do
        select c from Q
        D' ← c.FILTER(D)
        if any domain in D' is empty then return (fail,D')
        Q ← Q ∪ {c'∈C | ∃x∈var(c') D'x≠Dx} – {c}
        D ← D'
    end while
    return (true,D)
end AC-3
```


AC in practice

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

```
procedure AC-8(V,D,C)
    Q ← V
    while non-empty Q do
        select v from Q
        for c∈C such that v is constrained by c do
            D' ← c.FILTER(D)
            if any domain in D' is empty then return (fail,D')
            Q ← Q ∪ {u∈V | D'u≠Du}
            D ← D'
        end for
    end while
    return (true,D)
end AC-8
```

Arc-B-consistency

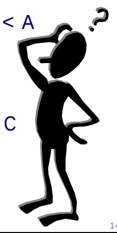
Lhomme (1993)

- Sometimes, making the problem arc-consistent is costly (for example, when domains of variables are large).
- In such a case, a weaker form of arc-consistency might be useful.

- We say that a constraint is **arc-b-consistent** (bound consistent) if for any bound values of the variable in the constraint there exists a value for the other variable(s) in such a way that the constraint is satisfied.
  - a bound value is either a minimum or a maximum value in domain
  - domain of the variable can be represented as an interval
  - for some constraints (like A<B) it is equivalent to AC

Pitfalls of AC

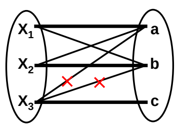
- **Disjunctive constraints**
  - A, B in {1,...,10}, $A = 1 \vee A = 2$
  - no filtering (whenever $A \neq 1$ then deduce $A = 2$ and vice versa)
- **Detection of inconsistency**
  - A, B, C in {1,...,10000000}, A < B, B < C, C < A
  - long filtering (4 seconds)
- **Weak filtering**
  - A, B in {1,2}, C in {1,2,3}, $A \neq B$, $A \neq C$, $B \neq C$
  - weak filtering (it is arc-consistent)



Inside all-different

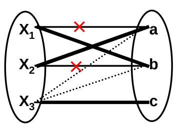
Régin (1994)

- a set of binary inequality constraints among all variables
  $X_1 \neq X_2$, $X_1 \neq X_3$, ..., $X_{k-1} \neq X_k$
- all_different$(\{X_1,\dots,X_k\}) = \{(d_1,\dots,d_k) \mid \forall i\ d_i \in D_i\ \&\ \forall i \neq j\ d_i \neq d_j\}$
- 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 ($X_1 \neq a$):

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

Meta consistency

Can we **strengthen any filtering technique**?

YES! Let us assign a value and make the rest of the problem consistent.

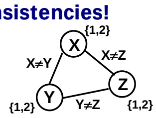
- **singleton consistency** **(Prosser et al., 2000)**
  - try each value in the domain
- **shaving**
  - try only the bound values
- **constructive disjunction**
  - propagate each constraint in disjunction separately
  - make a union of obtained restricted domains

Mackworth (1977)

Path consistency

Arc consistency does not detect all inconsistencies!

Let us look at several constraints together!



- The path (V_0,V_1,..., 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,...,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:

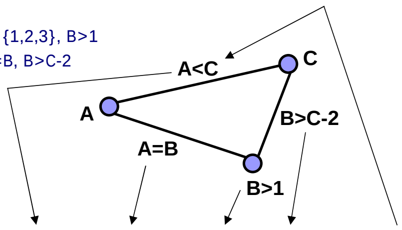
- only the **constraints between the neighboring variables** must be satisfied
- it is enough to explore **paths of length 2** (Montanary, 1974)

Path revision

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

Example:

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



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

Constraint satisfaction

Search techniques

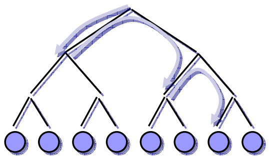
Search / Labeling

Inference 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 (**enumeration**)

In general, search algorithm resolves remaining disjunctions!

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

Labeling skeleton

- Search is combined with filtering techniques that prune the search space.
- **Look-ahead technique (MAC)**

```
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

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

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
  - **value ordering heuristics** 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

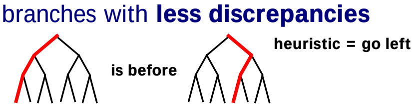
Discrepancy

= the heuristic is not followed

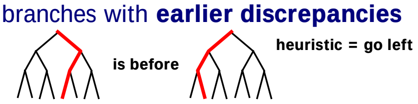
Basic principles of discrepancy search:

change the order of branches to be explored

- prefer branches with **less discrepancies**



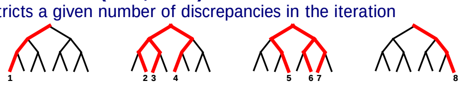
- prefer branches with **earlier discrepancies**



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



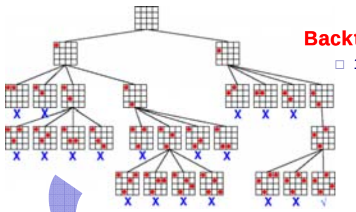
- **Depth-bounded Discrepancy Search (Walsh, 1997)**
  - restricts discrepancies till a given depth in the iteration

- …

*** heuristic = go left**

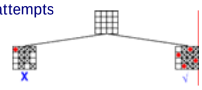
4 queens problem

CP is not (only) search!



Backtracking is not very good
- 19 attempts

MAC combining search and arc consistency
- 2 attempts



Constraint satisfaction

Extensions

Constraint optimization

- **Constraint optimization problem** (COP)
  = CSP + objective function
- Objective function is encoded in a constraint.
  - V = objective(Xs)
  - heuristics for bound estimate encoded in the filter

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) → 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 construct 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

- **Planning problem is internally dynamic**
  - actions in the plan are unknown in advance
  - ↳ a CSP is dynamic
  - Solution (Kautz & Selman, 1992):
    - **finding a plan of a given length is a static problem**
    - ↳ standard CSP is applicable there!
  - Constraint technology is frequently used to solve well-defined sub-problems such as temporal consistencies.
- **Scheduling problem is static**
  - all activities are known
  - ↳variables and constraints are known
  - ↳ standard CSP is applicable

P&S via CSP?

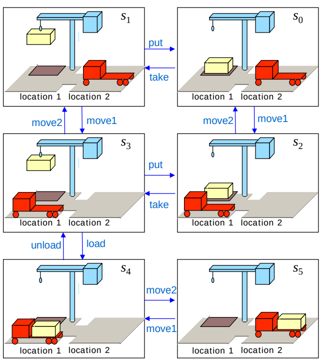
- **Exploiting state of the art constraint solvers!**
  - faster solver $\Rightarrow$ faster planner
- **Constraint model is extendable!**
  - it is possible immediately to add other variables and constraints
  - modeling numerical variables, resource and precedence constraints for planning
  - adding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

Constraints for planning

Constraint models

Planning problem

STRIPS



Propositions describe relevant features of states.
- {onground, onrobot, holding, at1, at2}

Initial state describes all initially valid propositions.
- s_0 = {onground, at2}

Goal describes propositions that must be valid in the goal state
- g = {onrobot}
- both s_4 and s_5 are goal states

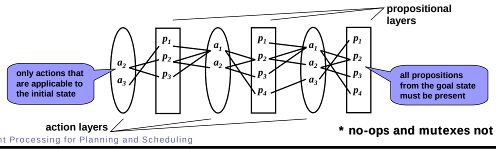
Action describes how propositions in the state are changed.
- load = (
  {holding,at1}, % precondition
  {holding}, % delete effect
  {onrobot}) % add effect

Plan is a sequence of actions transforming the initial state into a goal state.
- ⟨take,move1,load,move2⟩

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 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**.

Constraint 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**.

Constraint 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

Lopez & Bacchus (2003)

Boolean model
constraints

- **precondition constraints**
  - $A_{i,m+1} \Rightarrow P_{j,m}$
  - p_j is a precondition of action a_i
- **next state constraints**
  - $P_{i,m} \Leftrightarrow (\vee_{p_i \in add(a_j)} A_{j,m-1}) \vee (P_{i,m-2} \,\&\, (\wedge_{p_i \in del(a_j)} \neg A_{j,m-1})))$
  - p_j 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**
  - $\neg A_{i,m} \vee \neg A_{j,m}$ for mutex between actions a_i and a_j at layer m
  - $\neg P_{i,n} \vee \neg P_{j,n}$ for mutex between propositions p_i and p_j at layer n
- **goals**
  - $P_{i,k}$=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

Constraints for planning

Temporal reasoning

Foundations

What is time?

The mathematical structure of time is generally a set with **transitive and asymmetric ordering operation**.

The set can be continuous (reals) or discrete (integers).

The planning/scheduling systems need to **maintain consistent information about time relations.**

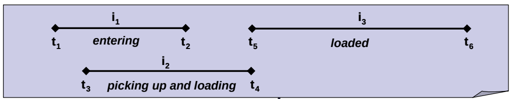
We can see time relations:

- **qualitatively**
  - relative ordering (A finished before B)
  - typical for modeling causal relations in planning
- **quantitatively**
  - absolute position in time (A started at time 0)
  - typical for modeling exact timing in scheduling

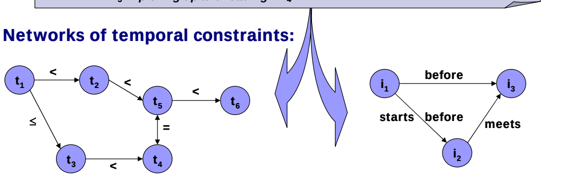
Ghallab et al. (2004)

Qualitative approach
example

- Robot starts entering a loading zone at time t_1 and stops there at time t_2.
- Crane starts picking up a container at t_3 and finishes putting it down at t_4.
- At t_5 the container is loaded onto the robot and stays there until time t_6.



Networks of temporal constraints:



Qualitative approach
formally

When **modeling time** we are interested in:

- **temporal references** (when something happened or hold)
  - **time points** (instants) when a state is changed **instant** is a variable over the real numbers
  - **time periods** (intervals) when some proposition is true **interval** is a pair of variables (x,y) over the real numbers, such that x<y
- **temporal relations** between temporal references
  - **ordering** of temporal references

Vilain & Kautz (1986)

Point algebra

symbolic calculus modeling relations between instants
without necessarily ordering them or allocating to exact times

There are three possible **primitive relations** between instants t_1 and t_2:

- $[t_1 < t_2]$, $[t_1 > t_2]$, $[t_1 = t_2]$

- A set of primitives, meaning a disjunction of primitives, can describe any (even incomplete) relation between instants:
  - R = { {}, {<}, {=}, {>}, {<,=}, {>,=}, {<,>}, {<,=,>} }
    - {} means failure
    - {<,=,>} means that no ordering information is available
  - useful operations on R:
    - **set operations** $\cap$ (conjunction), $\cup$ (disjunction)
    - **composition operation •** ($[t_1 < t_2]$ and $[t_2 =< t_3]$ gives $[t_1 < t_3]$ **)**

Consistency:

- The **PA network** consisting of instants and relations between them is **consistent** when it is possible to assign a real number to each instant in such a way that all the relations between instants are satisfied.
- To make the PA network consistent it is enough to make its transitive closure, for example using techniques of **path consistency**.
  - $[t_1\ r\ t_2]$ and $[t_1\ q\ t_3]$ and $[t_3\ s\ t_2]$ gives $[t_1\ r \cap (q \bullet s)\ t_2]$

Allen (1983)

Interval algebra

symbolic calculus modeling relations between intervals
(interval is defined by a pair of instants i^- and i^+, $[i^- < i^+]$)

There are thirteen primitives:

x **b**efore y	$x^+ < y^-$	
x **m**eets y	$x^+ = y^-$	
x **o**verlaps y	$x^- < y^- < x^+$ & $x^+ < y^+$	
x **s**tarts y	$x^- = y^-$ & $x^+ < y^+$	
x **d**uring y	$y^- < x^-$ & $x^+ < y^+$	
x **f**inishes y	$y^- < x^-$ & $x^+ = y^+$	
x **e**quals y	$x^- = y^-$ & $x^+ = y^+$	
b', m', o', s', d', f'	symmetrical relations	

Consistency:

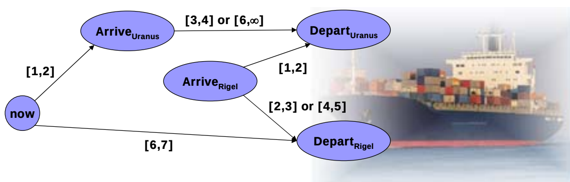
- The **IA network** is **consistent** when it is possible to assign real numbers to x_i^-, x_i^+ of each interval x_i in such a way that all the relations between intervals are satisfied.
- Consistency-checking problem for IA networks is an NP-complete problem.

Ghallab et al. (2004)

Qualitative approach

example

- Two ships, Uranus and Rigel, are directing towards a dock.
- The Uranus arrival is expected within one or two days.
- Uranus will leave either with a light cargo (then it must stay in the dock for three to four days) or with a full load (then it must stay in the dock at least six days).
- Rigel can be serviced either on an express dock (then it will stay there for two to three days) or on a normal dock (then it must stay in the dock for four to five days).
- Uranus has to depart one to two days after the arrival of Rigel.
- Rigel has to depart six to seven days from now.



Qualitative approach
formally

- The basic temporal primitives are again **time points**, but now the relations are numerical.
- Simple **temporal constraints** for instants t_i and t_j:
  - unary: $a_i \le t_i \le b_i$
  - binary: $a_{ij} \le t_i - t_j \le b_{ij}$,

  where a_i, b_i, a_{ij}, b_{ij} are (real) constants

Notes:
- Unary relation can be converted to a binary one, if we use some fix origin reference point t_0.
- $[a_{ij},b_{ij}]$ denotes a constraint between instants t_i a t_j.
- It is possible to use disjunction of simple temporal constraints.

Dechter et al. (1991)

STN

Simple Temporal Network (STN)
- only simple temporal constraints $r_{ij}= [a_{ij},b_{ij}]$ are used
- **operations**:
  - composition: $r_{ij} \bullet r_{jk} = [a_{ij}+a_{jk}, b_{ij}+b_{jk}]$
  - intersection: $r_{ij} \cap r'_{ij} = [\max\{a_{ij},a'_{ij}\}, \min\{b_{ij},b'_{ij}\}]$
- **STN** is **consistent** if there is an assignment of values to instants satisfying all the temporal constraints.
- **Path consistency** is a complete technique making STN consistent (all inconsistent values are filtered out, one iteration is enough). Another option is using all-pairs minimal distance **Floyd-Warshall algorithm**.

Algorithms

- **Path consistency**
  - finds a transitive closure of binary relations r
  - one iteration is enough for STN (in general, it is iterated until any domain changes)
  - works incrementally

one iteration for STN
```
PC(X,C)
    for each k : 1 ≤ k ≤ n do
        for each pair i,j : 1 ≤ i < j ≤ n, i ≠ k, j ≠ k do
            r_ij ← r_ij ∩ [r_ik • r_kj]
            if r_ij = ∅ then exit(inconsistent)
end
```

general
```
PC(C)
    until stabilization of all constraints in C do
        for each k : 1 ≤ k ≤ n do
            for each pair i,j : 1 ≤ i < j ≤ n, i ≠ k, j ≠ k do
                c_ij ← c_ij ∩ [c_ik • c_kj]
                if c_ij = ∅ then exit(inconsistent)
end
```

- **Floyd-Warshall algorithm**
  - finds minimal distances between all pairs of nodes
  - First, the temporal network is converted into a directed graph
    - there is an arc from i to j with distance b_{ij}
    - there is an arc from j to i with distance $-a_{ij}$.
  - STN is consistent iff there are no negative cycles in the graph, that is, $d(i,i) \ge 0$

```
Floyd-Warshall(X, E)
    for each i and j in X do
        if (i,j) ∈ E then d(i,j) ← l_ij else d(i,j) ← ∞
        d(i,i) ← 0
    for each i,j,k in X do
        d(i,j) ← min{d(i,j), d(i,k) + d(k,j)}
end
```

TCSP

Temporal Constraint Network (TCSP)

- It is possible to use **disjunctions of simple temporal constraints**.
- Operations • and ∩ are being done over the sets of intervals.
- **TCSP** is **consistent** if there is an assignment of values to instants satisfying all the temporal constraints.
- Path consistency does not guarantee in general the consistency of the TCSP network!
- A straightforward **approach** (constructive disjunction):
  - decompose the temporal network into several STNs by choosing one disjunct for each constraint
  - solve obtained STN separately (find the minimal network)
  - combine the result with the union of the minimal intervals

Constraints for scheduling

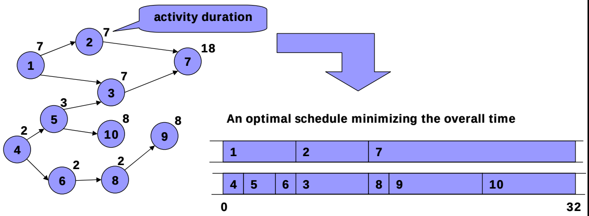
Base constraint model

Scheduling problem

Scheduling deals with **optimal resource allocation** of a given set of activities **in time**.

Example (two workers building a bicycle):

- Activities have a fixed duration, cannot be interrupted and the precedence constraints must be satisfied



Scheduling model

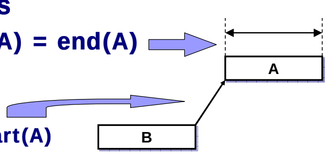
- **Scheduling problem** is static so it can be directly **encoded as a CSP**.
- Constraint technology is used for **full scheduling**.

Constraint 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 (∨ resource(A) ≠ resource(B))
    - ↳ **end(A)** ≤ **start(B)** ∨ **end(B)** ≤ **start(A)**

Constraints for scheduling
Resource constraints

Resources

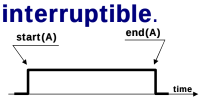
- **Resources are used in slightly different meanings in planning and scheduling!**
- **scheduling**
  - resource
    = a **machine** (space) for processing the activity
- **planning**
  - resource
    = consumed/produced **material** by the activity
  - resource in the scheduling sense is often handled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

Resource types

- **unary (disjunctive) resource**
  - a single activity can be processed at given time
- **cumulative (discrete) 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



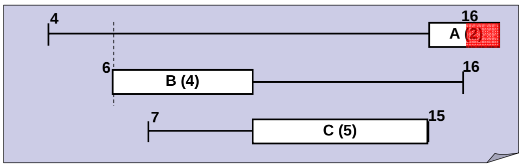
  - **interruptible** (preemptible) activity can be interrupted by another activity

  Note:
  There exists variants of below presented filtering algorithms for interruptible activities.
- A simple model with **disjunctive constraints**
  - $A \ll B \lor B \ll A$
    ↳ $\text{end}(A) \leq \text{start}(B) \lor \text{end}(B) \leq \text{start}(A)$

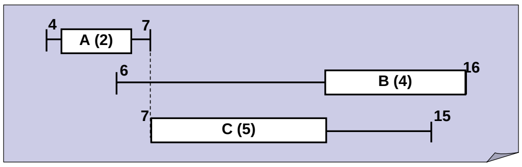
Baptiste & Le Pape (1996)

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

The rules:

- **$p(\Omega \cup \{A\}) > lct(\Omega \cup \{A\}) - est(\Omega) \Rightarrow A \ll \Omega$**
- **$p(\Omega \cup \{A\}) > lct(\Omega) - est(\Omega \cup \{A\}) \Rightarrow \Omega \ll A$**
- **$A \ll \Omega \Rightarrow end(A) \leq \min\{\, lct(\Omega') - p(\Omega') \mid \Omega' \subseteq \Omega \,\}$**
- **$\Omega \ll A \Rightarrow start(A) \geq \max\{\, est(\Omega') + p(\Omega') \mid \Omega' \subseteq \Omega \,\}$**

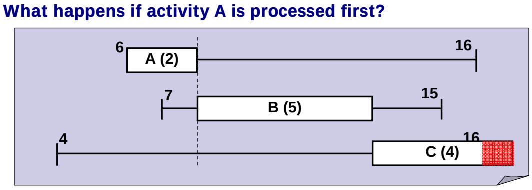
In practice:

- there are $n.2^n$ pairs (A,Ω) to consider (too many!)
- instead of Ω use so called **task intervals** [X,Y]
  $\{C \mid est(X) \leq est(C) \wedge lct(C) \leq lct(Y)\}$
  - time complexity $O(n^3)$, frequently used incremental algorithm
- there are also $O(n^2)$ and $O(n.\log n)$ algorithms

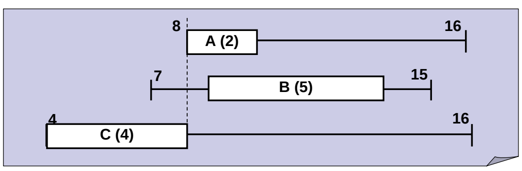
Torres & Lopez (2000)

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:

$p(\Omega \cup \{A\}) > lct(\Omega) - est(A) \Rightarrow \neg A \ll \Omega$

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

Not-last (symmetrical) rules:

$p(\Omega \cup \{A\}) > lct(A) - est(\Omega) \Rightarrow \neg \Omega \ll A$

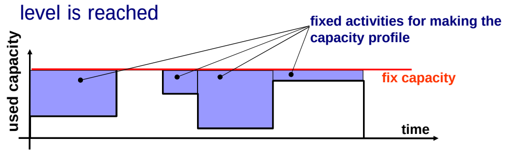
$\neg \Omega \ll A \Rightarrow end(A) \leq \max\{ lst(B) \mid B \in \Omega \}$

In practice:

- can be implemented with time complexity $O(n^2)$ and space complexity $O(n)$

Cumulative resources

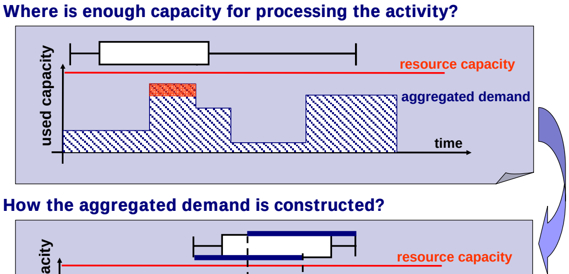
- Each **activity uses some capacity** of the resource – **cap(A)**.
- Activities can be **processed in parallel** if a resource capacity is not exceeded.
- Resource capacity **may vary in time**
  - modeled via fix capacity over time and fixed activities consuming the resource until the requested capacity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



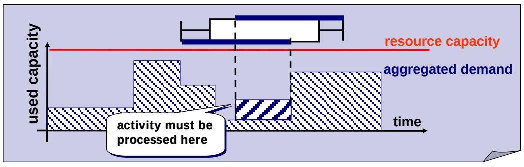
Baptiste et al. (2001)

Aggregated demands

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 exceeded at any time point?*

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

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

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

cap=1
in unary resource

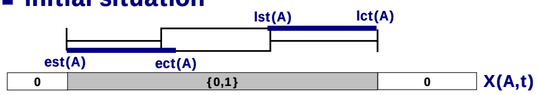
$$\forall t,i \ \ start(A_i) \le t < end(A_i) \Leftrightarrow X(A_i,t)$$

*** discrete time is expected**

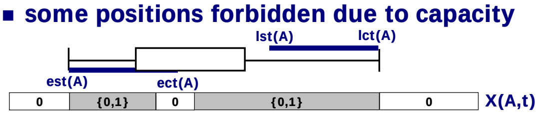
Timetable constraint

filtering example

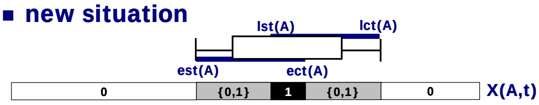
- **initial situation**



- **some positions forbidden due to capacity**



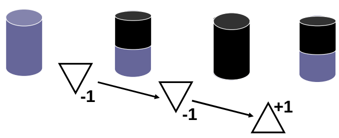
- **new situation**



Reservoirs

Producible/consumable resource

- Each event describes how much it increases or decreases the level of the resource.



- Cumulative resource can be seen as a special case of producible/consumable resource (reservoirs).
  - Each activity consists of consumption event at the start and production event at the end.

Relative ordering

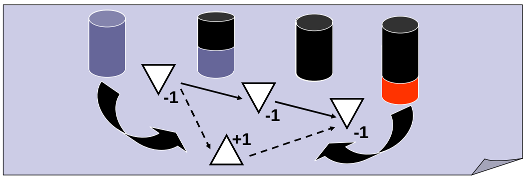
When time is relative (ordering of activities)

then edge-finding and aggregated demand deduce nothing

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

Example:

Reservoir: events consume and supply items



Cesta & Stella (1997)

Resource profiles

- Event 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 happens
  - events known to be before A are assumed together with the production events that can be before A

  orp(A) = InitLevel + prod(A) + $\Sigma_{B«A}$ **prod(B) +** $\Sigma_{B?A \wedge prod(B)>0}$ **prod(B)**
- **pessimistic resource profile (prp)**
  - minimal possible level of the resource when A happens
  - events known to be before A are assumed together with the consumption events that can be before A

  prp(A) = InitLevel + prod(A) + $\Sigma_{B«A}$ **prod(B) +** $\Sigma_{B?A \wedge prod(B)<0}$ **prod(B)**

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

Cesta & Stella (1997)

orp filtering

- **orp(A) < MinLevel ⇒ 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) –** $\Sigma_{B«C \wedge C?A \wedge prod(C)>0}$ **prod(C) < MinLevel ⇒ 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 ⇒ 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) –** $\sum_{B«C \wedge C?A \wedge prod(C)<0}$ **prod(C) > MaxLevel ⇒ B«A**

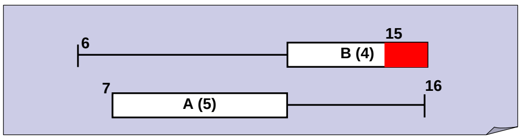
  for any B such that B?A and prod(B)<0
  - □ "if consumption in B is planned after A and the maximal required level in the resource is exceeded then B must be before A"

Vilím (2002)

Detectable precedence

from time windows to ordering

What happens if activity A is processed before B?



- □ Restricted time windows can be used to deduce new precedence relations.
- □ est(A)+p(A)+p(B) > lct(B) ⇒ B « A

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 resource(A)
    - deleting the resource from the domain of variable resource(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.

Alternative resources
filtering details

- Let A_u be the duplicate activity of A allocated to resource $u \in res(A)$.

$u \in resource(A) \Rightarrow start(A) \leq start(A_u)$
$u \in resource(A) \Rightarrow end(A_u) \leq end(A)$
$start(A) \geq min\{start(A_u) : u \in resource(A)\}$
$end(A) \leq max\{end(A_u) : u \in resource(A)\}$
failure related to $A_u \Rightarrow resource(A) \setminus \{u\}$

Actually, it is maintaining a constructive disjunction between the alternative activities.

Constraints for scheduling
Search strategies

Branching schemes

Branching = resolving disjunctions
Traditional scheduling approaches:

- take the **critical decisions first**
  - resolve bottlenecks …
  - defines the shape of the search tree
  - recall the **fail-first** 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?

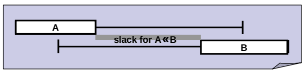
Smith and Cheng (1993)

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<<Ω $\vee$ $\neg$A<<Ω) or (Ω<<A $\vee$ $\neg$ Ω<<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

eclipse.crosscoreop.com

- 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 optimization)
- 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 CP

www.ilog.com/products/cp

- the largest family of optimization 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

Constraint Processing for Planning and Scheduling 88

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**

Constraint Processing for Planning and Scheduling 89

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**

Constraint Processing for Planning and Scheduling 90