Constraint Programming and Scheduling

Materials from the course taught at HTWG Constanz, Germany

Hana Rudová

Faculty of Informatics, Masaryk University
Brno, Czech Republic

http://www.fi.muni.cz/~hanka

May 2009

Constraint Programming and Scheduling: Outline

- **Constraint Programming**
  - Introduction
  - Modeling
  - Propagation
  - Search
- **Constraint-based Scheduling**
  - Introduction
  - Modeling
  - Propagation
  - Search

Constraint Programming: Introduction

1. CSP
2. CP Approach
3. Complexity

Materials on Constraint Programming

- **Rina Dechter: Constraint processing.**
  Morgan Kaufmann Publishers, 2003.
  - http://www.ics.uci.edu/~dechter/books/
- **Francesca Rossi, Peter Van Beek, Toby Walsh (editors): Handbook of Constraint Programming**, Elsevier, 2006
- **Edward Tsang: Foundations of Constraint Satisfaction.**
  Academic Press, 1993.
  - full text available:
    http://cswww.essex.ac.uk/Research/CSP/edward/FCS.html
- **Roman Barták: On-line guide to constraint programming.**
  - http://ktilinux.ms.mff.cuni.cz/~bartak/constraints/
- **Constraint Programming Online**
  - http://slash.math.unipd.it/cp/index.php

Constraint

- Given
  - set of **domain variables** $Y = \{y_1, \ldots, y_k\}$
  - finite set of values (**domain**) $D = D_1 \cup \ldots \cup D_k$

  constraint c defined on Y is a subset of $D_1 \times \ldots \times D_k$ **i.e. relation**
  - it constrains values which can variables take at the same time

- Example:
  - variables: A,B
  - domains: {0,1} for A {1,2} for B
  - constraint: A #\=B or (A,B) ∈ {(0,1),(0,2),(1,2)}

- Constraint c defined on variables $y_1, \ldots y_k$ is **satisfied**,

  if $(d_1, \ldots d_k) \in c$ holds for values $d_1 \in D_1, \ldots d_k \in D_k$

- Example (continues):
  - constraint is satisfied for $(0,1), (0,2), (1,2)$
    and unsatisfied for $(1,1)$

Constraint Satisfaction Problem (CSP)

Given

- finite set of **variables** $X = \{x_1, \ldots, x_n\}$
- finite set of values (**domain**) $D = D_1 \cup \ldots \cup D_n$
- finite set of **constraints** $C = \{c_1, \ldots, c_m\}$
  - each constraint defined over subset of X

Constraint satisfaction problem (CSP) is the tripple (X, D, C)

Example:

- variables: A,B,C
- domains: {0,1} for A {1} for B {0,1,2} for C
- constraints: A #\=B, B #\=C

Solution of CSP

Example: A in 0..1, B #= 1, C in 0..2, A #\= B, B #\= C

Partial assignment of variables $(d_1, \ldots, d_k), k < n$

- some variables have assigned a value A=1, B=1

Complete assignment of variables $(d_1, \ldots, d_n)$

- all variables have assigned a value A=1, B=1, C=0

Solution of CSP

- complete assignment of variables satisfying all the constraints
- $(d_1, \ldots, d_n) \in D_1 \times \ldots \times D_n$ is **solution** of (X, D, C)
  - $x_{i_1}, \ldots x_{i_k}$ holds for each $c_i \in C$ $(d_{i_1}, \ldots d_{i_k}) \in c_i$ A=0,B=1,C=2

Constraint Optimization Problem

- **objective function** F defined on variables from X B+C
- search for **optimal solution** of F minimal: B=1, C=0

Constraint Programming (CP) Approach

Formulation of given problem using constraints: **modeling**

Solving of given formulation using

- **domain specific methods**
- **general methods**

General Methods

Example: A in 0..2, B #= 1, C in 0..2, A #\= B, B #\= C, A #\= C

Constraint propagation algorithms

- allow to remove inconsistent values from domains of variables
  - values A=1 and C=1 are removed
- simplify problem
- maintain equivalency between original and simplified problem
- for computation of **local consistency**
- approximates **global consistency**

Search algorithms

- search through solution space
  1. select value 0 for A, as a consequence C=2
  2. select value 2 for A, as a consequence C=0
- example: backtracking, branch & bound

Domain Specific Methods

Specialized algorithms

Called **constraint solvers**

Examples:

- program for solving system of linear equations
- libraries for linear programming
- graph algorithm implementing special relation/constraint

Constraint programming

- broad term including many areas
- Artificial Intelligence, Linear Algebra, Global Optimization, Linear and Integer Programming, ...

Existence of domain specific methods $\Rightarrow$ must replace general methods

- goal: find domain specific methods to replace general methods

Polynomial and NP-complete Problems

Polynomial problems

- there is a polynomial complexity algorithm solving the problem

NP-complete problem

- solvable by non-deterministic polynomial algorithm
- potential solution can be verified in polynomial time
- exponential complexity in the worst case (if P=NP does not hold)

Complexity: Polynomial Problems

Linear equalities over real numbers

- variables with domains from $\mathbb{R}$, constraints: linear equalities
- Gaussova eliminace
- polynomial complexity

Linear unequalities over real numbers

- linear programming, simplex method
- polynomial complexity is often sufficient

Complexity: NP-complete Problems

Boolean constraints

- 0/1 variables
- constraint $\equiv$ Boolean formula (conjuntion, disjunction, implication, ...)
  - example: variables A, B, C, domains 0..1
    constraints: $(A \vee B), (C \Rightarrow A)$
    CSP: $(A \vee B) \wedge (C \Rightarrow A)$
- satisfiability problem of Boolean formula (SAT problem): NP-complete
- n variables: 2^n possibilities

Constraints over finite domains

- general CSP
- satisfiability problem of general relations
- NP-complete problem
- n variables, d maximal domain size: d^n possibilities

Complexity and Completeness

Complete vs. incomplete algorithms

- complete algorithm searches through the complete solution space
- incomplete algorithm: does not search the whole solution space
  - **"don't know"** as a possible response, effectiveness could be the gain
- example:
  - incomplete polynomial algorithm for NP-complete problem

Solver complexity

- Gauss elimination (P), SAT solver (NP), general CSP solver (NP)

Constraint propagation algorithms

- mostly incomplete polynomial algorithms

Search algorithms

- complete algorithms, examples: backtracking, generate & test
- incomplete algorithms, example: randomized time limited searches

Constraint Programming: Modeling

Systems for Constraint Programming

- **ILOG, constraints in C++** 1987
  - commercial product, originally from France
  - implementation of constraints based on object oriented programming
- **Swedish Institute of Computer Science: SICStus Prolog** 1985
  - strong CLP(*FD*) library
  - commercial product, wide academic use
- **Choco, constraints in Java** 1999 (C++), 2003 (Java)
  - open-source software
  - development at: École des Mines de Nantes, Cork Constraint Computation Center, Bouygues e-lab, Amadeus SA
- For more see: **Constraint Programming Online**
  - `http://slash.math.unipd.it/cp/index.php`

Domain Representation

domain(Variables, Min, Max)

- example: domain([A,B,C], 1,3).

Variable in Min..Max

- example: A in 1..3, B in -3..10

General representation: subset of intervals of integers

- $l_{1l} \leq l_{1u} < l_{2l} \leq l_{2u} < \ldots < l_{nl} \leq l_{nu}$
- example: $1 \leq 3 < 8 \leq 10 < 20 \leq 20 < 30 \leq 33$

  corresponds to values: 1,2,3, 8,9,10, 20, 30,31,32,33

Variable in Range

- example: A in (1..3) \/ (8..10) \/ {20} \/ (30..33)

  B in (1..3) \/ (8..15) \/ (5..9) \/ {100}

  results for B: $1 \leq 3 < 5 \leq 15 < 100 \leq 100$

Arithmetic Constraints

Expr RelOp Expr

RelOp –> #=| #\=| #<| #=<| #>| #>=

- examples: A + B #=< 3
  A #\=(C - 4) * (D - 5)
  A/2 #=4

sum(Variables,RelOp,Suma)

- example: domain([A,B,C,F],1,3), sum([A,B,C], #= ,F)

scalar_product(Coeffs,Variables,RelOp,ScalarProduct)

- example: domain([A,B,C,F],1,6),
  scalar_product([1,2,3],[A,B,C], #< ,F)

Global Constraints I.

Global Constraint Catalog: 313 constraints

- maintained by Beldiceanu & Carlsson & Rampon
- `http://www.emn.fr/x-info/sdemasse/gccat`

all_different(Variables)

- each variable must take unique value among variables
- example: domain([A,B,C],1,4), all_different([A,B,C]), A #= 1, B #> 3

  results in: A=1, B=4, C in 2..3

table(Tuples, Extension)

- defines *n*-ary constraint by extension
- Extension: list of lists of integers, each of length *n*
- Tuples: list of lists of domain variables (or integers), each of length *n*
- example: table([A,B,C]], [[1,2,3],[2,3,3]])

  results in: A in 1..2, B in 2..3, C=3

  table([A,B],[C,D]], [[1,2],[2,3]])

  results in: A in 1..2, B in 2..3, C in 1..2, D in 2..3

Global Constraints II.

element(N,List,X)

- N-th element of the List equals to X
- A in 2..10, B in 1..3, element(N, [A,B], X), X #< 2.

  results in: B = 1, N = 2, X = 1, A in 2..10

global_cardinality(List, KeyCounts)

- for each member Key-Count of the list KeyCounts holds: Count members of the List equals to Key
- each Key is integer and exists at most one among keys
- A in 1..3, B in 1..3, global_cardinality([A,B], [1-N,2-2]).

  results in: A = 2, B = 2, N = 0

Cryptogram

- Assign ciphers 0, . . . 9 to letters S, E, N, D, M, O, R, Y such that

```
  SEND
+ MORE
------
 MONEY
```

  - different letters have assigned different ciphers
  - S and M are not 0
- Unique solution:

```
  9567
+ 1085
------
 10652
```

- **Variables:** S,E,N,D,M,O,R,Y
- **Domains:** domain([S,M],1,9) domain([E,N,D,O,R,Y],0,9)

Cryptogram: equality constraints

- **1 equality constraint**

```
                1000*S + 100*E + 10*N + D          SEND
+               1000*M + 100*O + 10*R + E        + MORE
#= 10000*M + 1000*O + 100*N + 10*E + Y           MONEY
```

- OR **5 equality constraints**

  use of "transfer" variables P1,P2,P3,P4 with domains [0..1]

```
     D + E #= 10*P1 + Y,
P1 + N + R #= 10*P2 + E,
P2 + E + O #= 10*P3 + N,
P3 + S + M #= 10*P4 + O
        P4 #= M
```

Cryptogram: unequality constraints

- **28 unequality constraints**

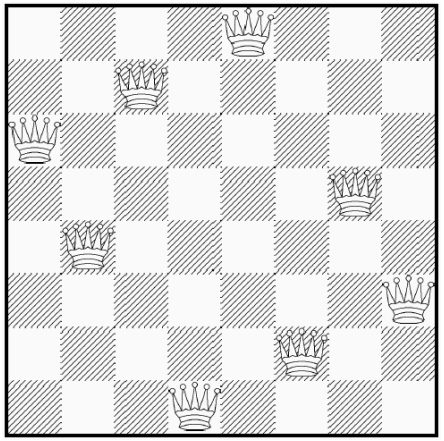
  X #\= Y for X,Y $\in$ {S,E,N,D,M,O,R,Y}, X$\prec$Y

- OR **1 unequality constraint**

  all_different([S,E,N,D,M,O,R,Y])

N Queens

Place N queens to $N \times N$ chessboard such that they do not attack each other.



- **Variables:** $X_1, \ldots, X_N$ each variable for each column
- **Domains:** domain($X_1, \ldots, X_N$],1,N)
- **Constraints:** for $i \in [1..(N-1)]$ and $j \in [(i+1)..N]$:
  - $X_i \ \#\backslash= X_j$ (rows) $\Rightarrow$ all_different $([X_1, \ldots, X_N])$
  - $X_i - X_j \ \#\backslash= i - j$ (diagonal)
    e.g. X_1=1 and X_2=2 prohibited ... diagonal through (1,1) a (2,2)
  - $X_i - X_j \ \#\backslash= j - i$ (diagonal)
    e.g. X_1=3 and X_2=2 prohibited ... diagonal through (1,3) a (2,2)

Knapsack Problem

There is a knapsack of the size M and there are N articles of given size S_i and cost C_i. Choose such articles to place into the knapsack to maximize the cost of articles in the knapsack.

- knapsack of the size M
- articles of the size $S_1, \ldots, S_N$ and cost $C_1, \ldots, C_N$

- **Variables:** $X_1, \ldots, X_N$
- **Domains:** domain([$X_1, \ldots, X_N$],0,1)
- **Constraint:**

  scalar_product([$S_1, \ldots, S_N$], [$X_1, \ldots, X_N$], #=<, M)
- **Objective function:**

  scalar_product([$C_1, \ldots, C_N$], [$X_1, \ldots, X_N$], #=, F)
- **Goal:** maximize(F)

Exercises

Define **domain variables with their domains** and write a **set of constraints** together with possible **optimization criteria** to describe the model of particular constraint satisfaction problems in the following excercises.

Cryptogram

Assign different ciphers to letters such that the equation

$$DONALD + GERALD = ROBERT$$

holds. Be careful that zero cannot be assigned to leading letters D, G, R to make the formula well readable.

Cryptogram: Solution

Variables:

- domain([O,N,A,L,E,B,T],0,9)
- domain([D,G,R],1,9]

Constraints:

- all_different([D,G,R,O,N,A,L,E,B,T])
-

```
    100000*D + 10000*O + 1000*N + 100*A + 10*L + D
+   100000*G + 10000*E + 1000*R + 100*A + 10*L + D
#=  100000*R + 10000*O + 1000*B + 100*E + 10*R + T
```

Timetable

Find time for classes of given teachers such that each teacher is expected to teach within specified interval:

teacher	min	max
Peter	3	6
Jane	3	4
Anne	2	5
Yan	2	4
Dave	3	4
Mary	1	6

All classes are taught at the same room and take one hour.

Classes of women should be taught as earlierst as possible.

Hint: all_different

Timetable: Solution

Variables for starting time of classes:

- Peter in 3..6, Jane in 3..4, Anne in 2..5,
  Yan in 2..4, Dave in 3..4, Mary in 1..6

Constraints:

- all_different([Peter, Jane, Anne, Yan, Dave, Mary])

Optimization:

- constraint: Jane + Anne + Mary #= F
- minimize(F)

Magic Sequence

A magic sequence of length n is a sequence of integers $x_0 \ . \ . \ x_{n-1}$ between 0 and $n-1$, such that for all i in 0 to $n-1$, the number i occurs exactly x_i times in the sequence.

For instance, 6,2,1,0,0,0,1,0,0,0 is a magic sequence since 0 occurs 6 times in it, 1 occurs twice, ...

Find magic sequence of the length n.

Solution:
n=0,1,2 a 5: no solution
n=3: [1,2,1,0] and [2,0,2,0]
n=4: [2,1,2,0,0]
n=6: [3,2,1,1,0,0,0]
for $n \geq 6$: [n-3,2,1,...,1,0,0,0]

Hint: global_cardinality

Magic Sequence: Solution

M=N-1 domain variables for a sequence of numbers:

- domain([X0,...XM], 0, M)

Constraints:

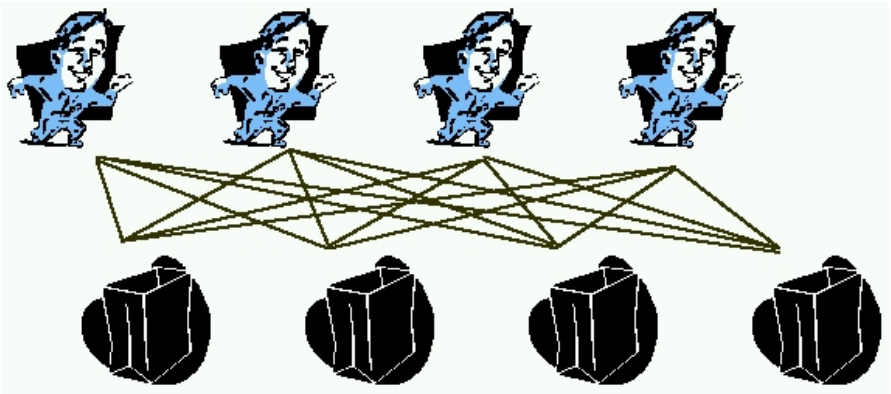
- global_cardinality([X0,X1,...,XM],[0-X0,1-X1,...,M-XM])

Additional (redundant) constraints to improve propagation:

- sum([X0,...XM],(M+1))
- scalar_product([0,1,...,M],[X0,X1,...,XM], #= ,(M+1))

Assignment Problem

Assign four workers W1,W2,W3,W4 to four products such that each worker works on one product and each product is produced by one worker. Effectiveness of production is given by the following table (e.g., worker W1 produces P1 with effectivness 7) and the total effectiveness must be 19 at least.



	P1	P2	P3	P4
W1	7	1	3	4
W2	8	2	5	1
W3	4	3	7	2
W4	3	1	6	3

Hint: element(W1, [7,1,3,4], EW1)

Assignment Problem: Solution

Variables for each worker representing a product to be produced by him:

- domain([W1,W2,W3,W4], 1, 4)

Constraints:

- all_different([W1,W2,W3,W4])
- element(W1,[7,1,3,4],EW1),
  element(W2,[8,2,5,1],EW2),
  element(W3,[4,3,7,2],EW3),
  element(W4,[3,1,6,3],EW4)
- EW1+EW2+EW3+EW4 #>=19

Constraint Programming: Propagation

Graph Representation of CSP

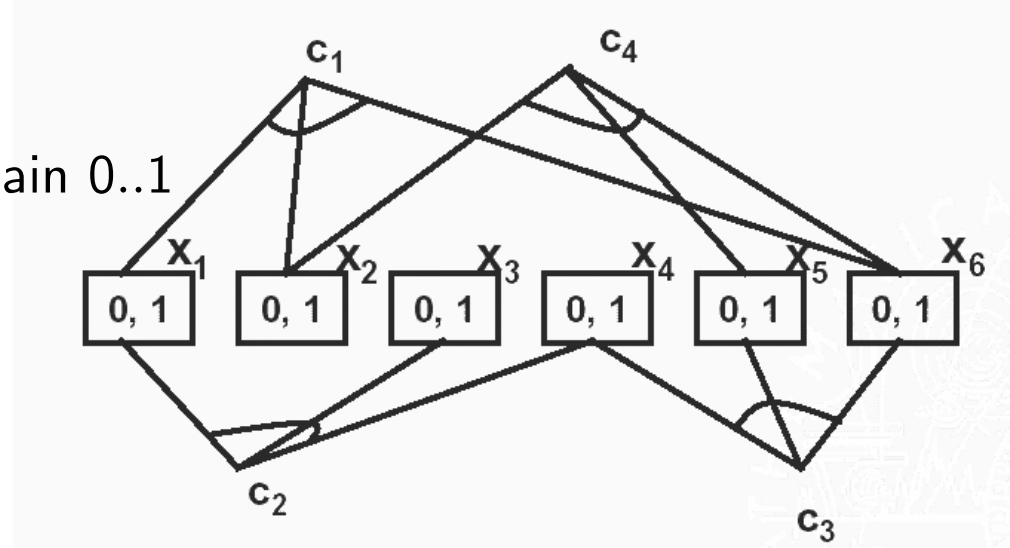
- **Constraint representation**
  - using mathematical/logical formula
  - by extension (list of compatible k-tuples, 0-1 matrix)

- **Graph:** nodes, edges (edge connects two nodes)
- **Hypergraph:** nodes, hyper-edges (hyper-edge connects set of nodes)

- CSP representation through **constraint hypergraph**
  - node = variable, hyper-edge = constraint

- Example
  - variables $X_1, \ldots, X_6$ with domain 0..1
  - constraints:

    $c_1 : X_1 + X_2 + X_6 = 1$
    $c_2 : X_1 - X_3 + X_4 = 1$
    $c_3 : X_4 + X_5 - X_6 > 0$
    $c_4 : X_2 + X_5 - X_6 = 0$



Binary CSP

Binary CSP

- CSP with binary constraints only
- unary constraints encoded into the domain

Constraint graph for binary CSP

- hypergraph not necessary
  graph sufficient (constraint defined on two nodes only)

Each CSP can be transformed to "corresponding" binary CSP

Binarization: pros and cons

- getting of unified CSP, many algorithms proposed for binary CSPs
- unfortunately significant increase of the problem size

Non-binary constraints

- more complex propagation algorithms
- semantics of constraints allows for stronger propagation
  - example: all_different vs. set of inequality constraints

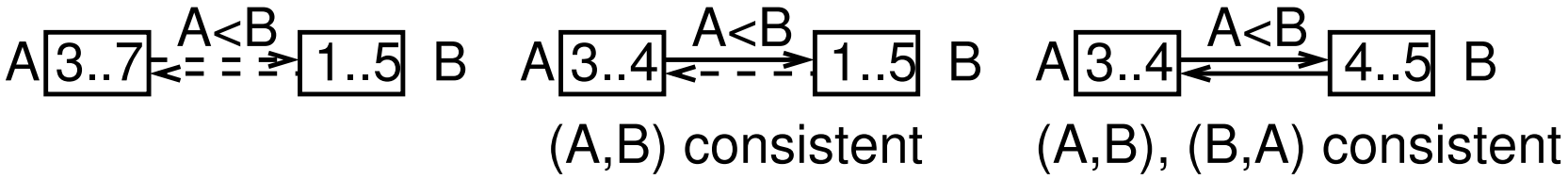
Node and Arc Consistency

Node consistency (NC)

- each value from the domain of variable V_i satisfies all unary constraints over V_i

Arc consistency (AC) for binary CSP

- **arc** (V_i, V_j) **is arc consistent**, iff there is a value y for each value x from the domain of V_i such that the assignment $V_i = x, V_j = y$ satisfies all binary constraints over V_i, V_j.
- arc consistency is **directional**
  - consistency of arc (V_i, V_j) does not guarantee consistency of (V_j, V_i)



- **CSP** is **arc consistent**, iff all arcs (in both directions) are arc consistent

Arc Revision

- How to make arc (V_i, V_j) arc consistent?
- Remove such values x from D_i that are inconsistent with current domain of V_j (there is no value y for V_j such that assignment $V_i = x$ and $V_j = y$ satisfies all binary constrants between V_i a V_j)

```
procedure revise((V_i, V_j))
Deleted := false
for ∀x in D_i do
    if there is no y ∈ D_j such that (x, y) is consistent
    then D_i := D_i − {x}
          Deleted := true
    end if
return Deleted
end revise
```

- domain([V_1, V_2],2,4), $V_1 \#< V_2$
  revise((V_1, V_2)) removes 4 from D_1, D_2 is not changed

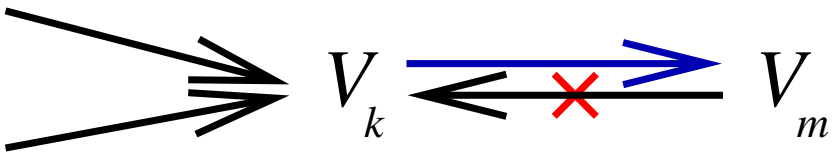
Computation of Arc Consistency

How to make CSP arc consistent?

- revisions must be repeated if the domain of some variable was changed
- more effective: revisions can be done with the help of **queue**
  - we add to queue arcs which consistency could have been violated by value removal(s)

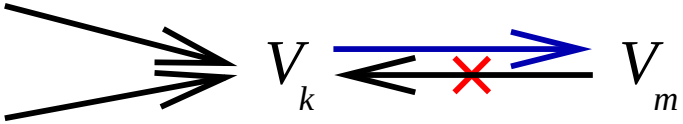
What arcs must be exactly revised after domain reduction of V_k?

- arcs (V_i, V_k) leading to variable V_k with reduced domain

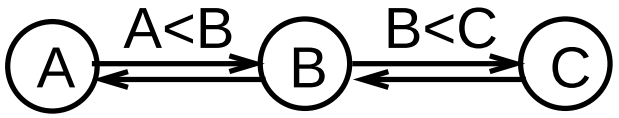


- let (V_k, V_m) caused domain reduction of V_k: then arc (V_m, V_k) leading from V_m may not be revised (change does not influence it)

AC-3 Algorithm



procedure AC-3(G)
Q := $\{(V_i, V_j) \mid (V_i, V_j) \in$ edges(G), $i \neq j\}$ % list of arcs to revise
while Q non empty do
    choose and remove (V_k, V_m) from Q
    if revise((V_k, V_m)) then % additions of arcs
        Q := Q $\cup$ $\{(V_i, V_k) \in$ edges(G), $i \neq k$, $i \neq m\}$ % still not in queue
end while
end AC-3



Example:
A<B, B<C: $(\underline{3..7},1..5,1..5) \xrightarrow{\mathtt{AB}} (3..4,\underline{1..5},1..5) \xrightarrow{\mathtt{BA}} (3..4,\underline{4..5},1..5)$
$\xrightarrow{\mathtt{BC}} (3..4,4,\underline{1..5}) \xrightarrow{\mathtt{CB}} (\underline{3..4},4,5) \xrightarrow{\mathtt{AB}} (3,4,5)$

AC-3 is the most common today but it is still not optimal!

Excercise: What will be domains of A,B,C after AC-3 for:

- domain([A,B,C],1,10), A #= B + 1, C #< B

Is it arc consistency sufficient?

AC removes many inconsistent values

- do we get solution of the problem then? NO
- do we know that solution of the problem exists? NO
  - domain([X,Y,Z],1,2), X #\= Y, Y #\= Z, Z #\= X
    - arc consistent
    - no solution exists

Why we use AC then?

- sometimes we obtain solution directly
  - some domain is emptied ⇒ no solution exists
  - all domains have one element only ⇒ we have solution
- general case: size of the solution search is decreased

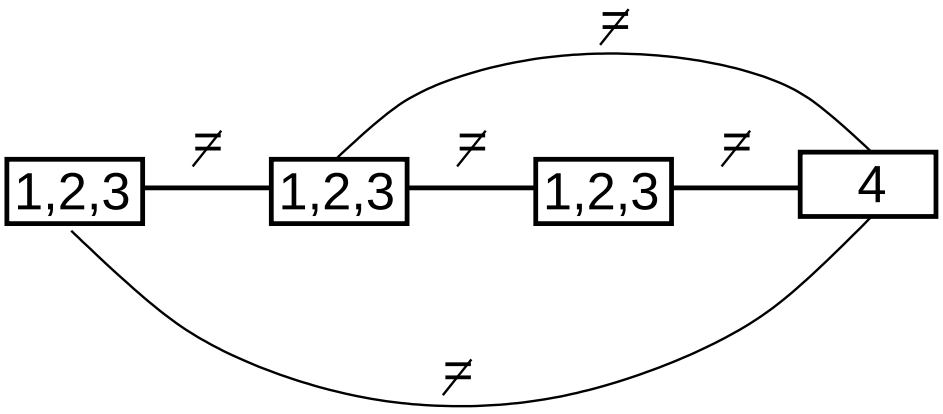
k-consistency

Have NC and AC anything in common?

- NC: consistency of one variable
- AC: consistency of two variables
- . . . an we can continue

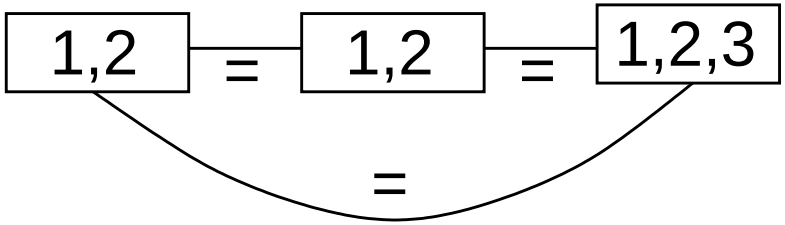
CSP is **k-consistent** iff, any consistent assignment of (k-1) variables can be extended to any k-th variable

k-consistency defined for general CSPs including non-binary constraints



4-consistent graph

Strong k-consistency



3-consistent graph

$(1,1)$ can be extended to $(1,1,1)$

$(2,2)$ can be extended to $(2,2,2)$

$(1,3)$ and $(2,3)$ are not consistent tuples (we do not extend those)

not 2-consistent

(3) cannot be extended

CSP is **strongly k-consistent** iff it is j-consistent for each j≤k

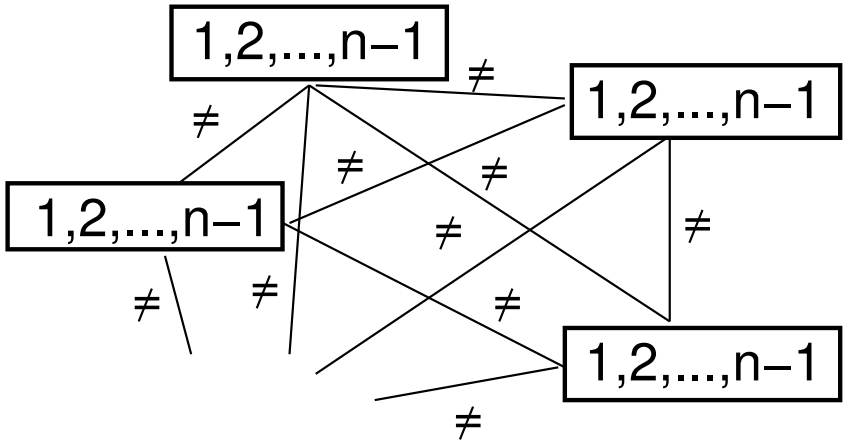
- strong k-consistency ⇒ k-consistency
- strong k-konzistence ⇒ j-consistency $\forall j \leq k$
- k-consistency ⇏ strong k-consistency

- NC = strong 1-consistency = 1-consistency
- AC = (strong) 2-consistency

Consistency for Finding of the Solution

If we have graph with n nodes, how strong consistency is necessary to obtain solution directly?

- strong n-consistency is necessary for the graph with n nodes
  - n-consistency does not suffice (see earlier example)
  - strong k-consistency for k<n does not suffice too



graph with n nodes
domains 1..(n-1)

strong k-consistent for each k<n
there is no solution yet

Strong n-consistency is necessary for the graph with n nodes

- exponential complexity!

Non-binary Constraints

k-consistency have exponential complexity, it is not used in practice

n-ary constraints are used directly

Constraint is **generally arc consistent (GAC)** iff for each variable V_i from this constraint and for each its value $x \in D_i$, there is an assignment $y \in D_j$ for each remaining variable V_j in the constraint such that it is satisfied

- A + B = C, A in 1..3, B in 2..4, C in 3..7 is GAC

Semantics of constraints is used

- special type of consistency for global constraints
  - e.g. all_different
- bounds consistency can be used
  - propagation when the smallest or largest domain value changes

One constraint may use different types of consistency

- A #\= B: arc consistency, bounds consistency

Consistency Algorithm for Non-binary Constraints

Algorithm with **queue of variables** (sometimes also called AC-8)

procedure AC-8(Q)
while Q non empty do
    choose and remove $V_j \in$ Q
    for $\forall\, c$ such that $V_j \in$ *scope*(c) do
        W := revise(V_j, c)
            // W is the set of variables with changed domain
        if $\exists\ V_i \in W$ such that $D_i = \emptyset$ then return fail
        Q := Q $\cup \{W\}$
end AC-8

- revisions are repeated untill there are domain changes
- *scope*(c): set of variables on which c is defined

Implementation

- set of **constraints to be propagated** maintained for each variable
- user defines REVISE procedures based on the constraint type

Bounds Consistency

Constraint is **bounds consistent (BC)** iff for each variable V_i from this constraint and for each its value $x \in D_i$, there is an assignment of remaining variables in the constraint such that it is satisfied and $\min(D_i) \leq y_i \leq \max(D_i)$ holds for selected assignment y_i of V_i

Bounds consistency: weaker than generalized arc consistency

Propagation when **minimal and maximal value (bounds)** changed only

Bounds consistency for inequalities

- **A #> B =>** min(A) $\geq$ min(B)+1, max(B) $\leq$ max(A)-1
- example: A in 4..10, B in 6..18, A #> B
  min(A) $\geq$ 6+1 $\Rightarrow$ A in 7..10
  max(B) $\leq$ 10-1 $\Rightarrow$ B in 6..9
- and similar for: A #< B, A #>= B, A #=< B

Bounds Consistency and Arithmetic Constraints

A #= B + C => min(A) $\geq$ min(B)+min(C), max(A) $\leq$ max(B)+max(C)
min(B) $\geq$ min(A)-max(C), max(B) $\leq$ max(A)-min(C)
min(C) $\geq$ min(A)-max(B), max(C) $\leq$ max(A)-min(B)

- change of min(A) causes the change of min(B) and min(C) only
- change of max(A) causes the change of max(B) a max(C) only, ...

Example:

A in 1..10, B in 1..10, A #= B + 2, A #> 5, A #\= 8

A #= B + 2 $\Rightarrow$ min(A) $\geq$ 1+2, max(A) $\leq$ 10+2 $\Rightarrow$ A in 3..10
$\Rightarrow$ min(B) $\geq$ 1-2, max(B) $\leq$ 10-2 $\Rightarrow$ B in 1..8
A #> 5 $\Rightarrow$ min(A) $\geq$ 6 $\Rightarrow$ A in 6..10
$\Rightarrow$ min(B) $\geq$ 6-2 $\Rightarrow$ B in 4..8 (new propagation for A #= B + 2)
A #\= 8 $\Rightarrow$ A in (6..7) \/ (9..10)
(bounds same, no propagation for A #= B + 2)

Exercises: Bounds and Arc Consistency

1. Define rules for bounds consistency of the constraints
   A #= B - C, A #>= B + C

2. What are rules for bounds consistency of the constraint X #= Y+5? How propagations are processed in the following example?
   X in 1..20, Y in 1..20, X #= Y + 5, Y #> 10.

3. What is the difference between bounds and arc consistency? Show it on the example.

4. How arc consistency is achieved in the following example?
   domain([X,Y,Z],1,5]), X #< Y, Z #= Y+1

Constraint Programming: Search

Search & Consistency

Constraint satisfaction through **search** of the solution space

- constraints used passively as a test
- assign values and try what happens
- examples: **backtracking**, **generate & test** (trivial)
- complete methods (either solution is find or inconsistency is proved)
- too slow (exponential): search through "clearly" bad assignments

Consistency/propagation techniques

- allow to remove inconsistent values from domains
- incomplete methods (some inconsistent values still in domains)
- relatively fast (polynomial)

Combination of both methods used

- subsequent assignment of values to variablse
- after assignment, inconsistent values removed by consistency techniques

Depth First Search (DFS)

Depth first search of the solution space:
base search algorithm for CSPs

Two phases of the search

- **forward phase:** variables subsequently selected, partial assignment extended by assignment of consistent value (if exists) to another variable
  - after value selection, consistency tests are processed
- **backward phase**: if there is no consistent value for current variable, algorithm backtracks to earlier assigned value

Types of variables

- **past**: already selected variable (have assigned value)
- **current**: currently selected variable to be assigned a value
- **future**: variables which will be selected in the future

Core search procedure: DFS

Variables numbered for simplicity, assignment processed in given order

Initial call: labeling(G,1)

```
procedure labeling(G,a)
if a > |edges(G)| then return nodes(G)
for ∀x ∈ D_a do
   if consistent(G,a) then   % consistent(G,a) replaced by FC(G,a), LA(G,a), ...
      R := labeling(G,a + 1)
      if R ≠ fail then return R
return fail
end labeling
```

R: assignment of variables or fail

Procedures consistent(G,i) will be described for binary constraints only

Backtracking (BT)

Backtracking verifies consistency of constraints leading from past variables to current variable at each step

Backtracking mintains consistency of constraints

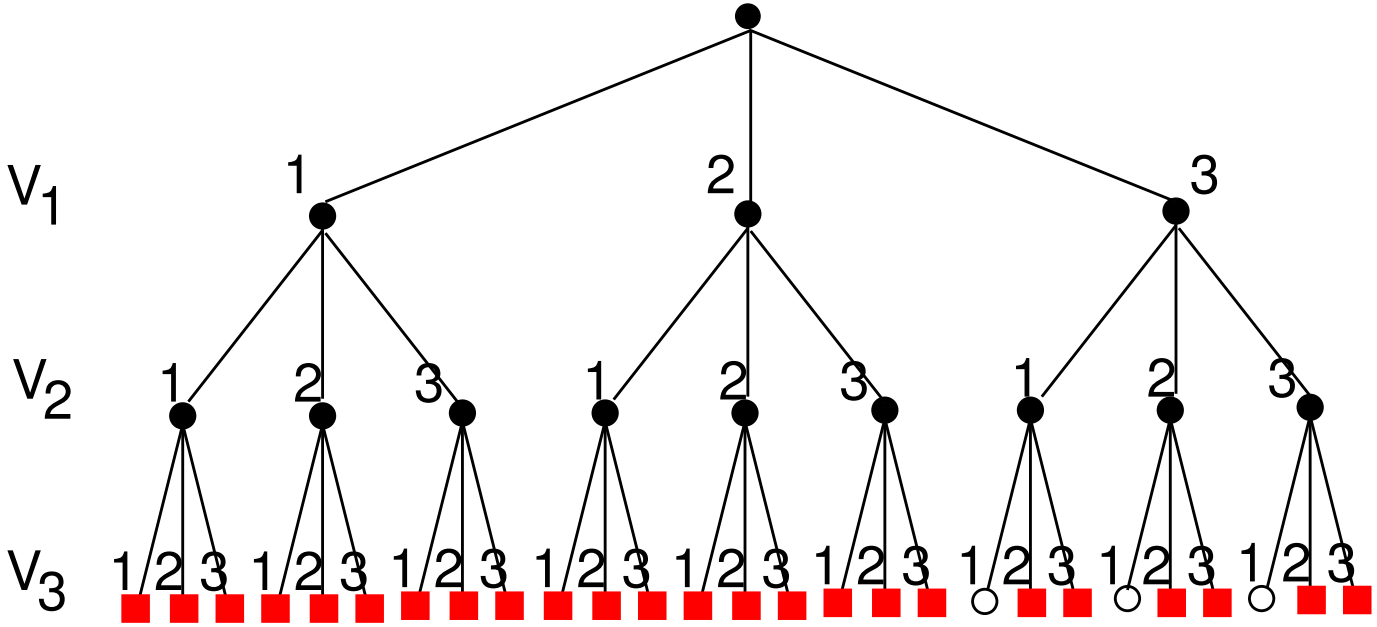
- on all past variables
- on past and current variable

procedure BT(G,a)
Q:=$\{(V_i, V_a) \in$ edges(G), $i < a\}$ % edges from past to current variable
Consistent := true
while non empty Q $\wedge$ Consistent do
    choose and remove any edge (V_k, V_m) from Q
    Consistent := not revise(V_k, V_m) % if value is removed, domain is empty!
return Consistent
end BT

Example: Backtracking

Constraints: V_1, V_2, V_3 in $1 \ldots 3$, $\quad V_1\# = 3 \times V_3$

Solution space:



- red boxes: failed attempt for assignment, no solution
- empty circles: solution found
- black circles: inner node representing partial assignment

Forward Checking (FC)

FC is an extension of backtracking

In addition, FC maintains consistency between current and future variables

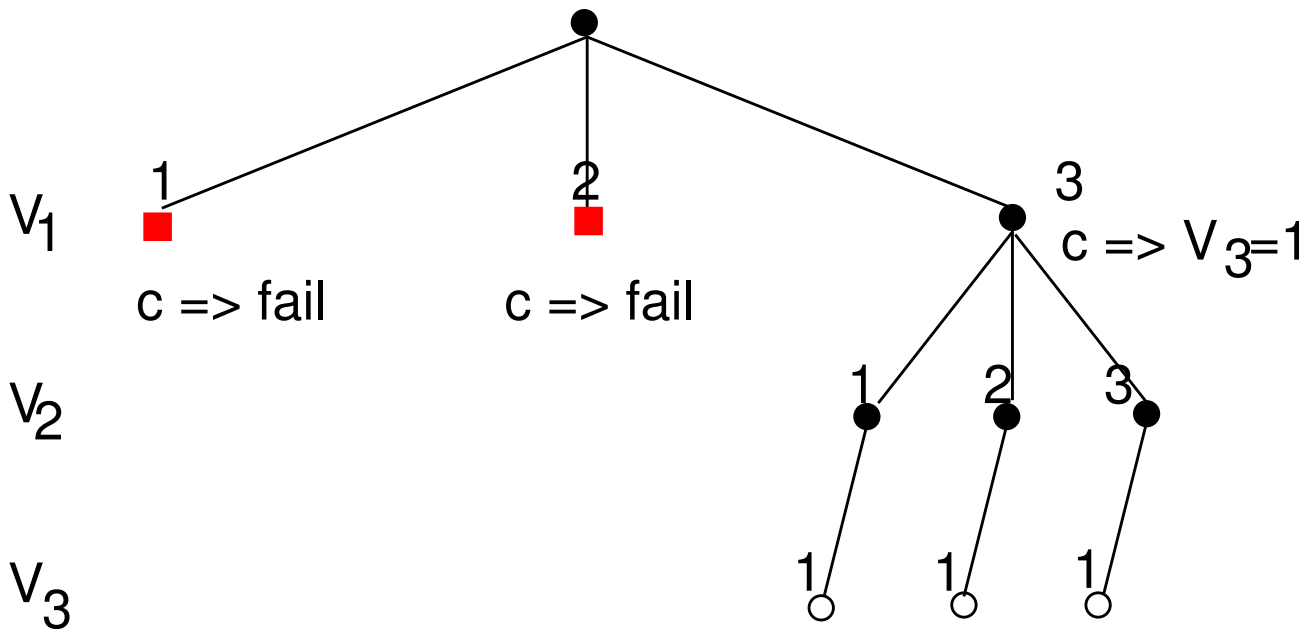
procedure FC(G,a)
Q:=$\{(V_i, V_a) \in$ edges(G), $i > a\}$
% addition of arcs from future to current variable
Consistent := true
while non empty Q $\wedge$ Consistent do
    choose and remove any edge (V_k, V_m) from Q
    if revise((V_k, V_m)) then
        Consistent := $(|D_k| > 0)$ % empty domain means fail
return Consistent
end FC

Edges from past to current variables is not necessary to test

Example: Forward Checking

Constraints: V_1, V_2, V_3 in $1 \ldots 3$, $\quad c : V_1\# = 3 \times V_3$

Solution space:



Looking Ahead (LA)

LA is an extension of FC, LA maintains arc consistency
In addition, LA maintains consistency between all future variables

procedure LA(G,a)
Q := $\{(V_i, V_a) \in$ edges(G), $i > a\}$ % start with edges leading to a
Consistent := true
while non empty Q $\wedge$ Consistent do
  choose and remove any edge (V_k, V_m) from Q
  if revise((V_k, V_m)) then
    Q := Q $\cup \{(V_i, V_k) | (V_i, V_k) \in$ **edges(G),** $i \neq k, i \neq m, i > a\}$
    Consistent := $(|D_k| > 0)$
return Consistent
end LA

- edges from past variables to current variable are not necessary to test again
- this LA procedure is based on AC-3, other AC algorithms can be also applied

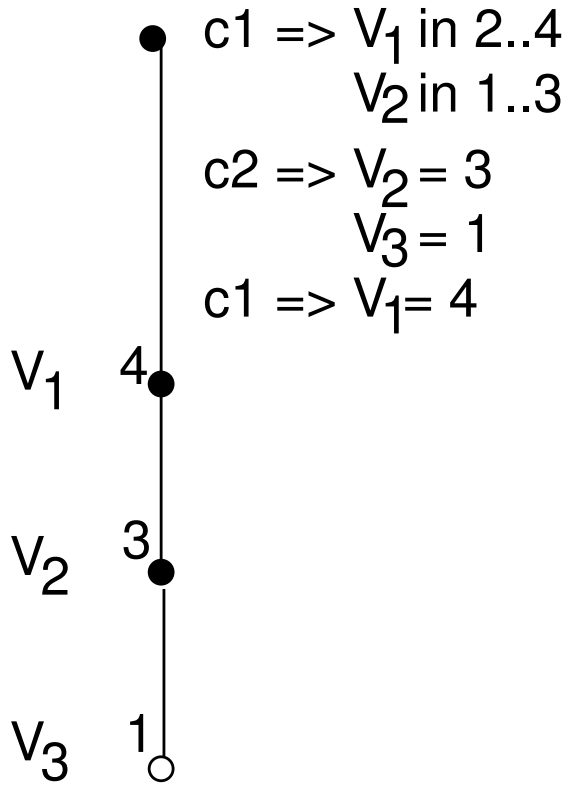
LA maintains arc consistency: since LA(G,a) applies AC-3,
initial consistency must be computed by AC-3 before search starts

Example: Looking Ahead (with AC-3)

Constraints: V_1, V_2, V_3 in $1 \ldots 4$, $c1 : V_1\# > V_2$, $c2 : V_2\# = 3 \times V_3$

Solution space:

- **initial consistency is computed (by AC-3 algorithm) before search**



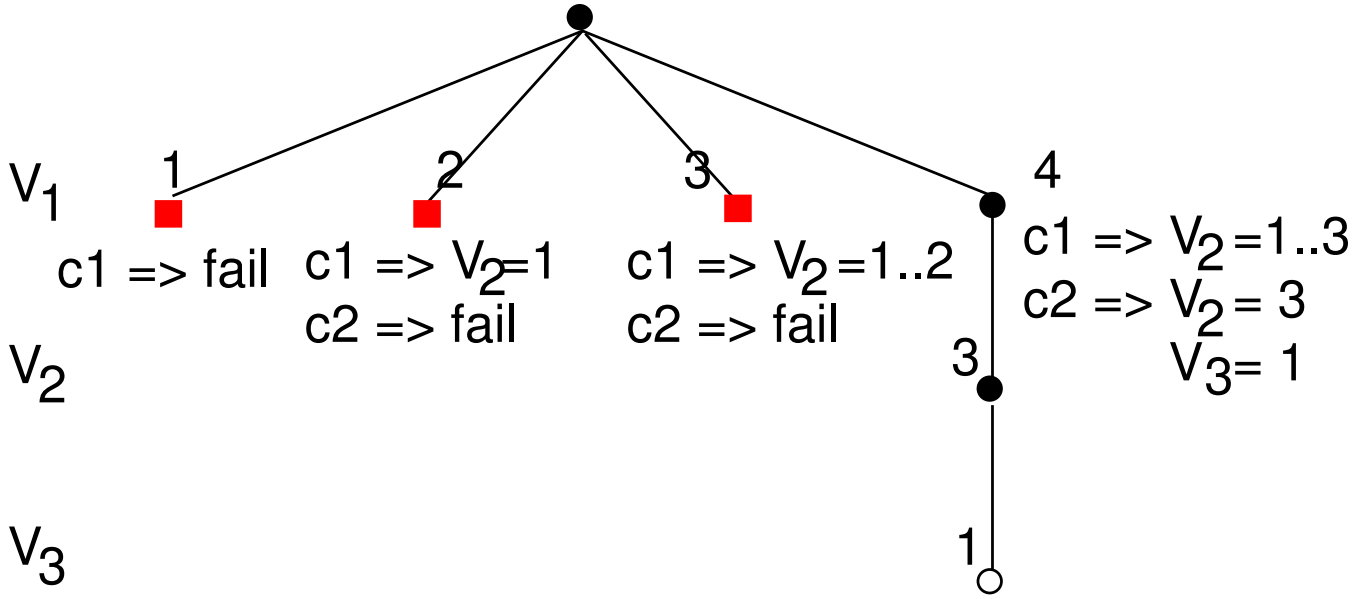
Example: Looking Ahead with AC-1

Constraints: V_1, V_2, V_3 in $1 \ldots 4$, $c1 : V_1\# > V_2$, $c2 : V_2\# = 3 \times V_3$

Solution space (when AC-1 is used instead of AC-3):

- **initial consistency before search is not computed**
- AC-1 algorithm repeats revisions of all arcs in cycles unless domains of all variables are stable (do not change)

$\Rightarrow$ AC-1 makes the problem arc consistent as soon as the value of current variable is assigned

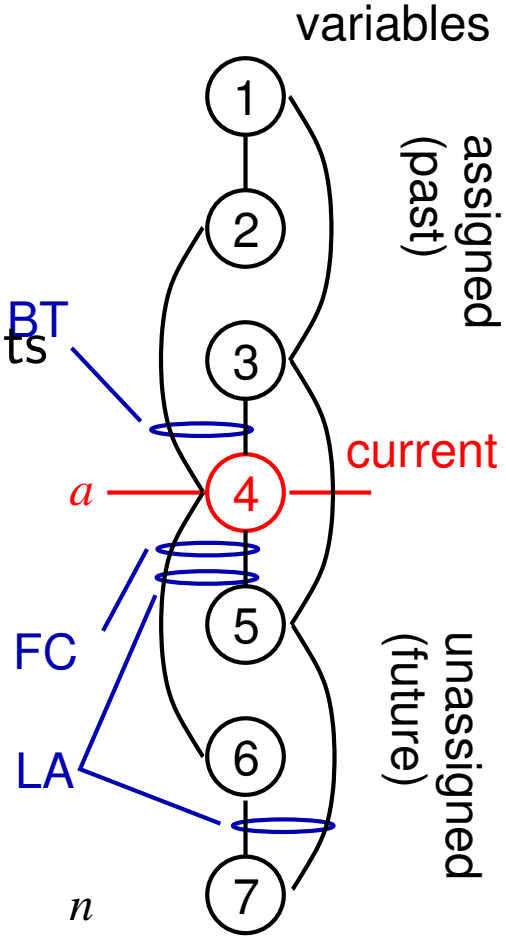


Summary of Algorithms

Backtracking (BT) maintains at step a constraints
$c(V_1, V_a), \ldots, c(V_{a-1}, V_a)$
from past variables to current variable

Forward checking (FC) maintains at step a constraints
$c(V_{a+1}, V_a), \ldots, c(V_n, V_a)$
from future variables to current variable

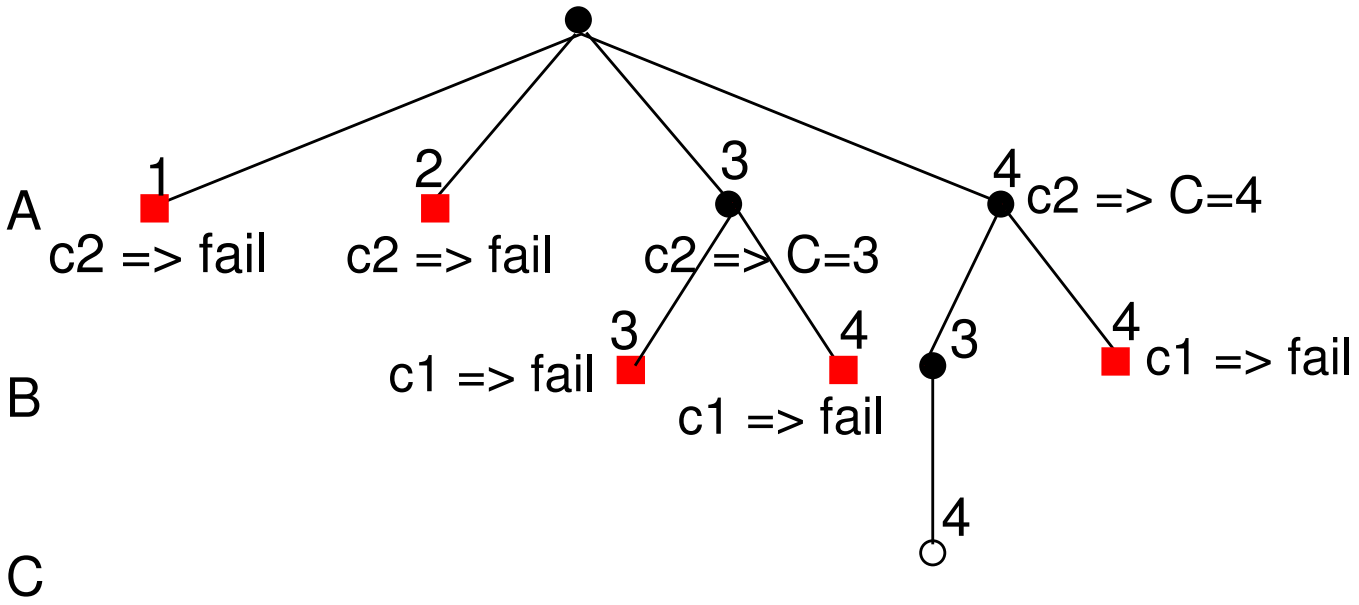
Looking Ahead (LA) maintains at step a constraints
$\forall l (a \leq l \leq n), \forall k (a \leq k \leq n), k \neq l : c(V_k, V_l)$
from future variables to current variable
between future variables



Exercise 1.

1. Write solution space for constraints
A in 1..4, B in 3..4, C in 3..4, c1: B #< C, c2: A #= C
when using forward checking and ordering of variables A,B,C. Explain what types of propagation happens in each node.

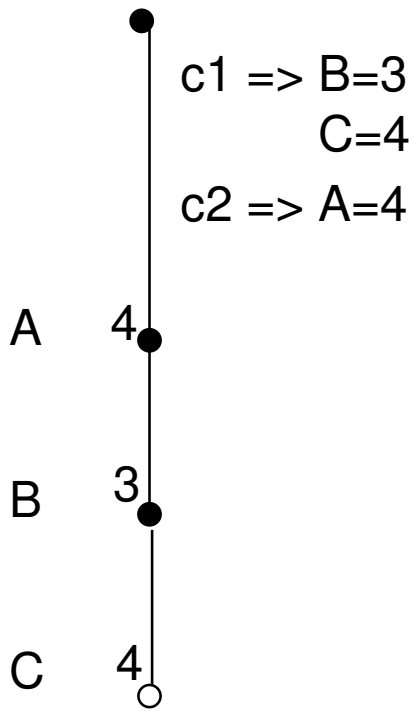
Solution:



Exercise 2.

2. Write solution space for constraints
A in 1..4, B in 3..4, C in 3..4, c1: B #< C, c2: A #= C
when using looking ahead and ordering of variables A,B,C. Explain what types of propagation happens in each node.

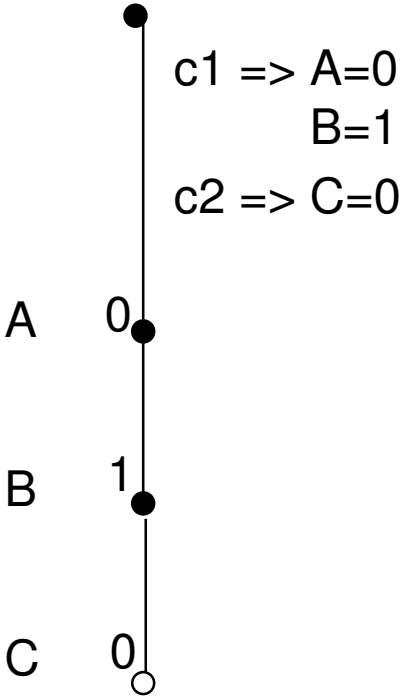
Solution: (initial propagation computed by AC-3)

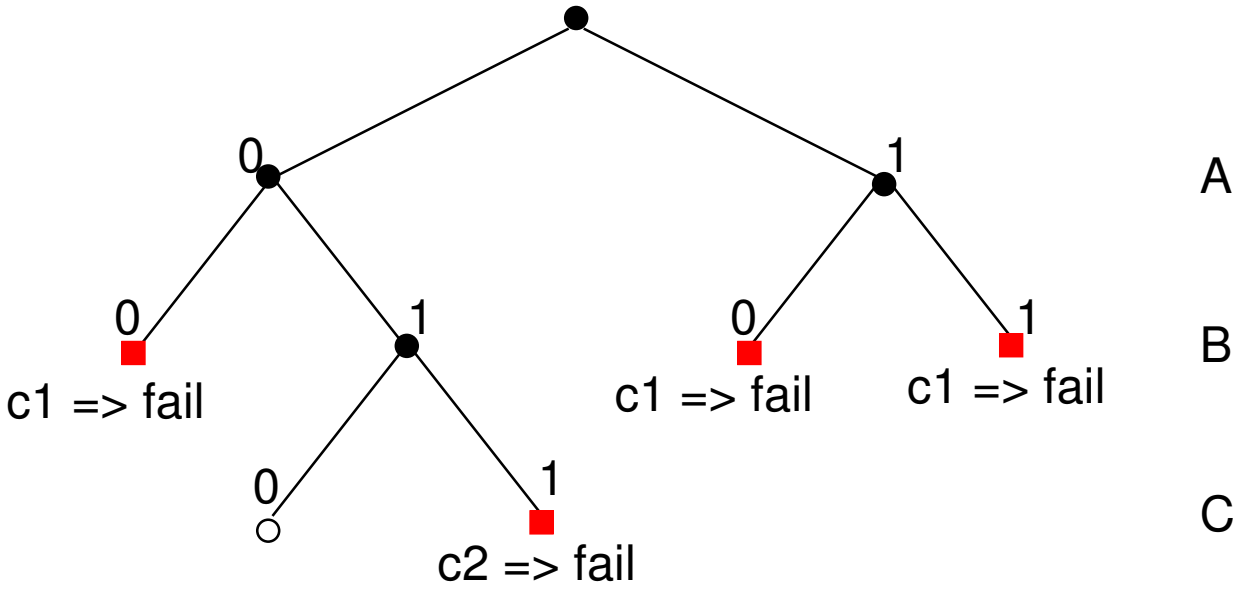


Exercises 3. and 4.

3. Write and compare solution spaces for constraints domain([A,B,C],0,1), c1: A #= B-1, c2: C #= A*A when using backtracking and looking ahead. Explain what types of propagation happens in each node.

Solution:





4. Present on some example differences between forward checking and looking ahead.

Constraint-based Scheduling: Introduction

Materials on Constraint-based Scheduling (CBS)

Some topics are described in

- Philippe Baptiste, Philippe Laborie, Claude Le Pape, Wim Nuijten: Constraint-based Scheduling and Planning. Chapter 22 in Handbook of Constraint programming, pages 761-799, Elsevier, 2006.
- Roman Barták: Filtering Techniques in Planning and Scheduling, ICAPS 2006, June 6-10, 2006, Cumbria, England

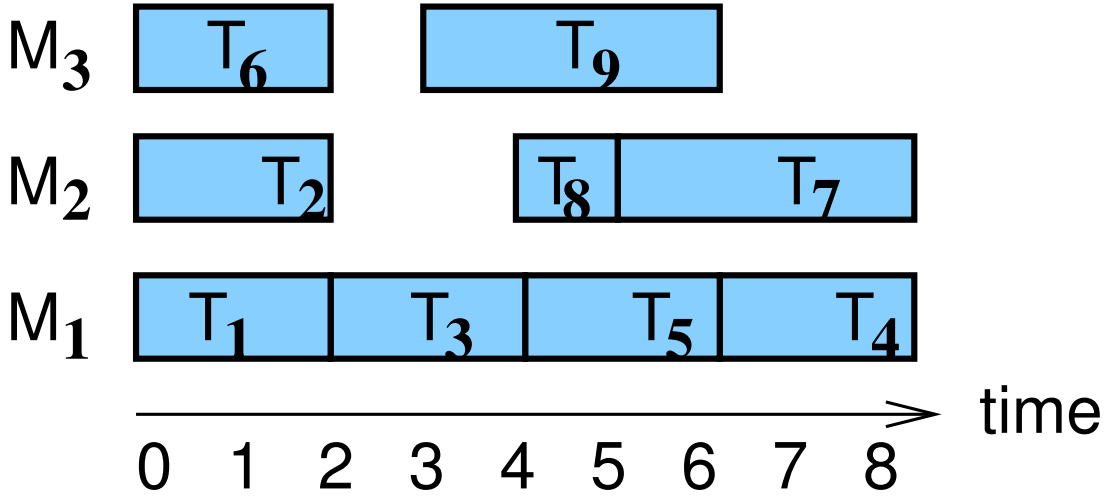
  `http://www.plg.inf.uc3m.es/icaps06/preprints/i06-tu2-allpapers.pdf`
- Philippe Baptiste, Claude Le Pape, Wim Nuijten: Constraint-based Scheduling, Kluwer Academic Publishers, 2001.

Scheduling

- **Scheduling**
  optimal resource allocation of a given set of activities in time
  - resource or machine
  - activity or task

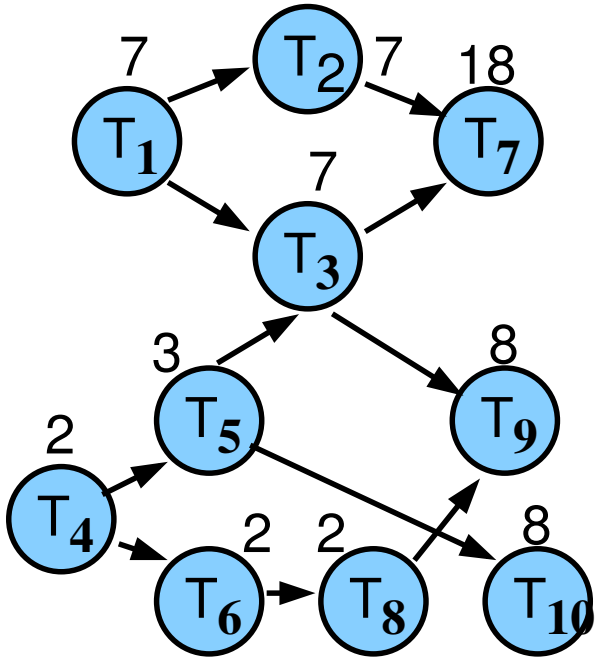
- Machine $M_j, j = 1, \ldots, 3$ Task $T_i, i = 1, \ldots, 9$

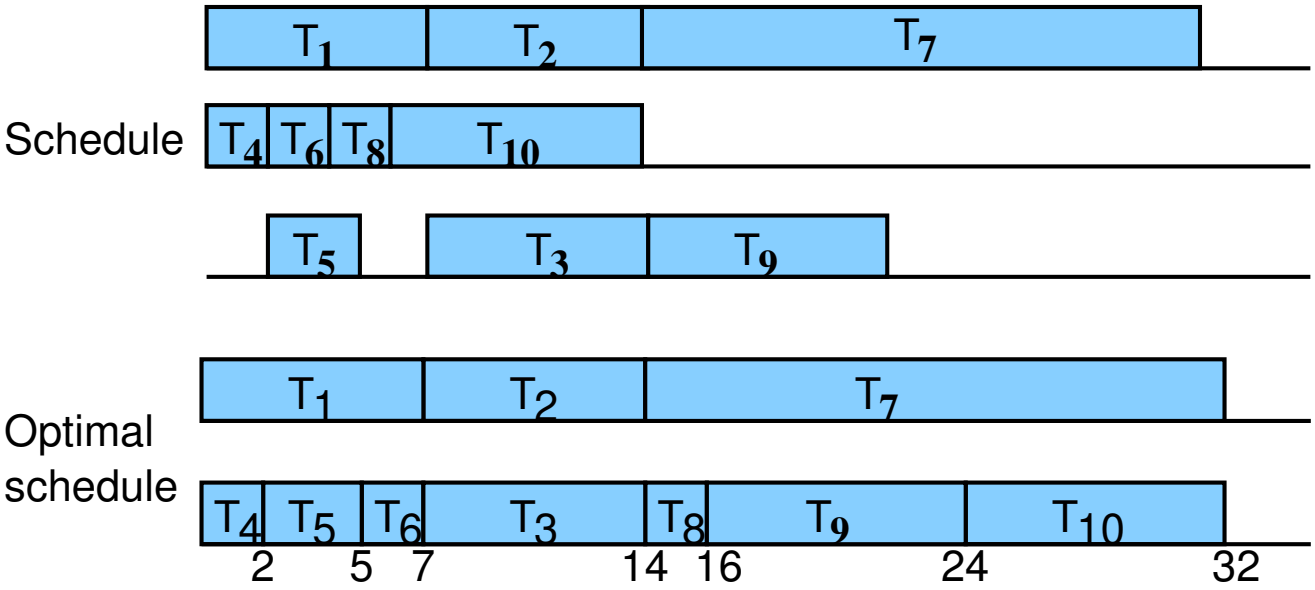
Machine–oriented Gantt chart



Example: Bicycle Assembly

- 3 workers who can perform tasks
- 10 tasks with its own duration
- Precedence constraints ($T_i \ll T_j$)
  - activity must be processed before other activity
- No preemption
  - activity cannot be interrupted during processing





Example: Classroom allocation

- One day seminar with several courses to be presented in several available rooms
- 8:00am – 4:00pm (periods 1,2,. . . ,8)
- **14 courses** (A,B, . . . N)
  each course has several meetings with pre-assigned time periods
- **5 rooms** (1,2,3,4,5) . . . resources
- **Find suitable room for each meeting**

Demands:

Course	A	B	C	D	E	F	G	H	I	J	K	L	M	N
Periods	2	8	4 5	1 2	3 4 5	6	7 8	2 3	1 2	5	6 7	3 4	8	2 3 4

Solution = Schedule/Timetable:

Periods	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Room 1	D	D		C	C	F		B
Room 2	I	I	E	E	E		G	G
Room 3		H	H		J	K	K	
Room 4		N	N	N				M
Room 5		A	L	L				

Scheduling problems

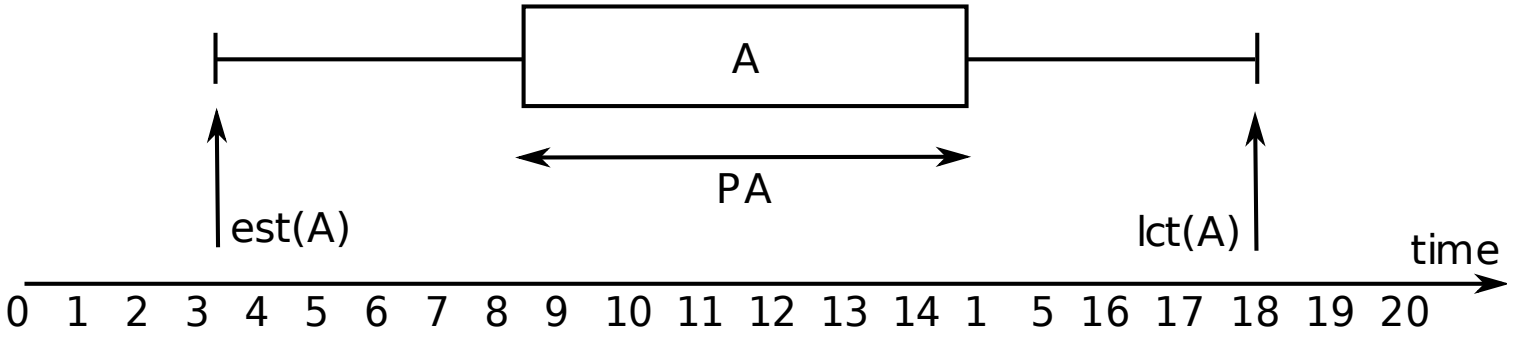
- **Project planning and scheduling**
  - software project planning
- **Machine scheduling**
  - allocation of jobs to computational resources
- **Scheduling of flexible assembly systems**
  - car production
- **Employees scheduling**
  - nurse rostering
- **Transport scheduling**
  - gate assignment for flights
- **Sports scheduling**
  - schedule for NHL
- **Educational timetabling**
  - timetables at school
- ...

Scheduling as a CSP: time assignment

Activity A is an entity occupying some space (resources) and time

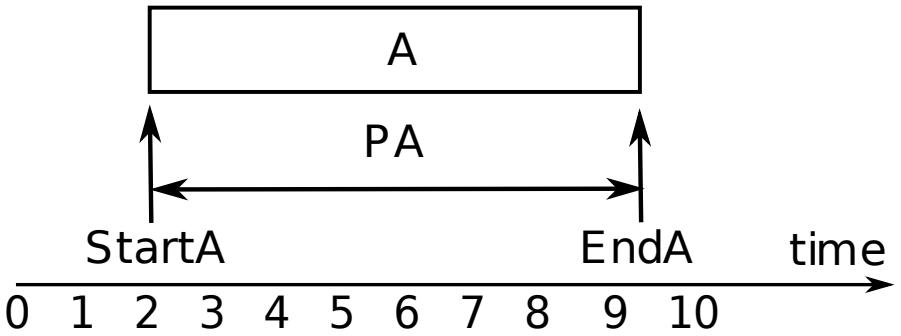
Variables and their domains for each activity for **time assignment**

- **StartA**: start time of the activity
  - activity cannot start before its **release date**
  - **est(A)** = min(StartA), earliest start time
- **EndA**: completion time of the activity
  - activity must finish before the **deadline**
  - **lct(A)** = max(EndA), latest completion time
- **PA**: processing time (duration) of the activity
  - StartA = {est(A), . . . , (lct(A)-PA)}
  - EndA = {(est(A)+PA), . . . , lct(A)}

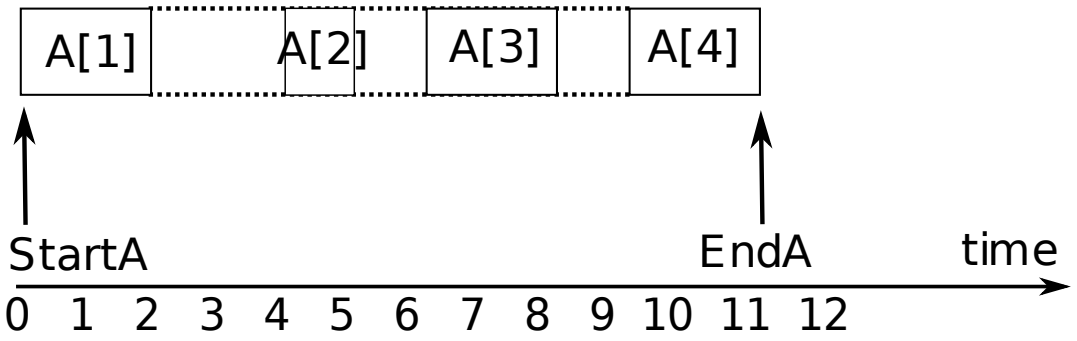


Scheduling as a CSP: basic constraints I.

- **Non-preemptive** activity: **no interruption** during processing
  - StartA + PA #= EndA



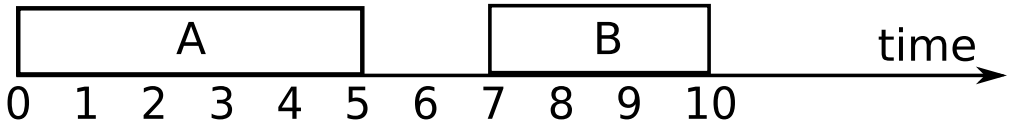
- **Preemptible activity**: **can be interrupted** during its processing
  - StartA + PA #=< EndA



PA #= PA[1] + PA[2] + PA[3] + PA[4]

Scheduling as a CSP: basic constraints II.

- **Sequencing A«B** of activities A,B
  (also: **precedence constraint** between activities A,B)
  - EndA #=< StartB



- **Disjunctive constraint**: non-overlapping of activities A, B
  - non-preemptive activities
  - A«B or B«A
  - (EndA #=< StartB) #\/ (EndB #=< StartA)
  - related with the idea of unary resource

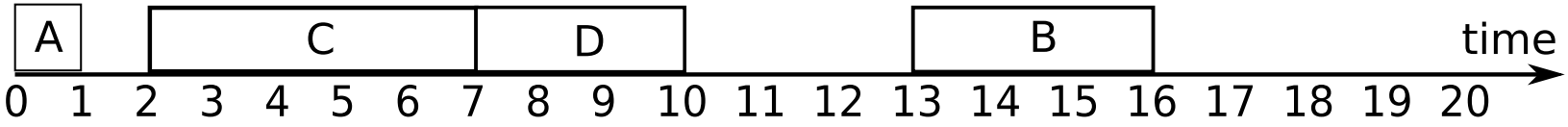
Scheduling as a CSP: resource allocation

Domain variables for resources

- **CapA**: requested capacity of the resource
  - unary resources
  - cumulative resources
  - producible/consumable resources
- **ResourceA**: alternative resources for A

Unary (disjunctive) resources

- Each activity requests
  unary capacity of the resource: CapA=1
- **Single activity can be processed at given time**
- Any two non-preemptive activities are related
  by the **disjunctive constraint A«B or B«A**



- Example: one machine with jobs running on it

- **cumulative([task(StartA1,PA1,EndA1,1,A1),**
  ..., task(StartAn,PAn,EndAn,1,An)],Options)

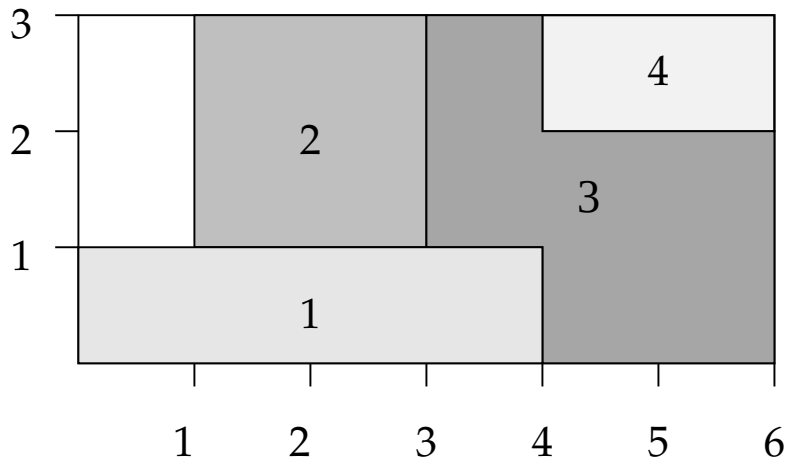
  A1, ..., An: activity indentifiers

  Options: options for different propagation algorithms

  4th parameter of the task = 1: unit consumption of the resource

Cumulative (discrete) resources

- Each **activity uses some capacity** of the resource **CapA**
- Several activities can be **processed in parallel** if a resource capacity is not exceeded

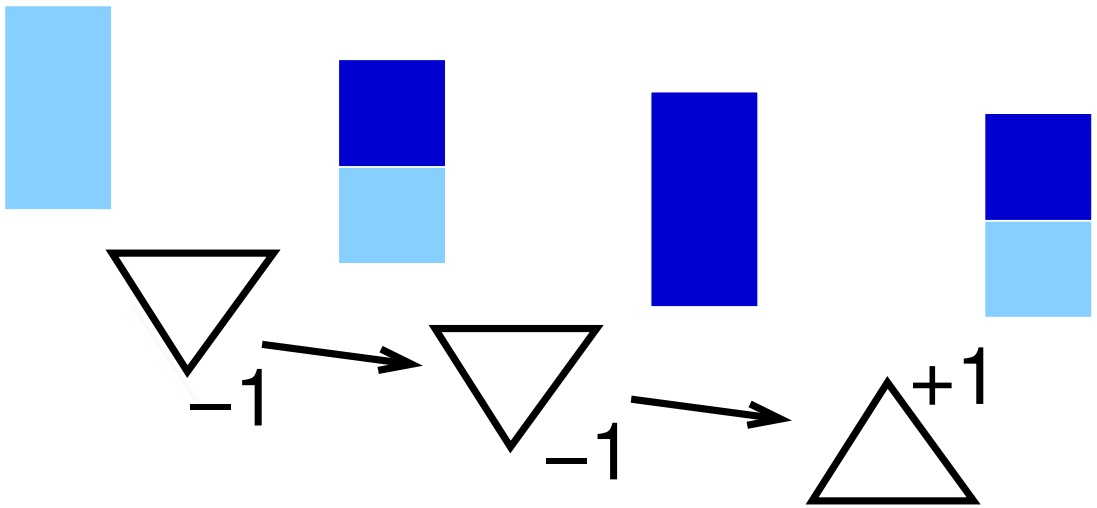


- Example: multi-processor computer with parallel jobs
- **cumulative([task(StartA1,PA1,EndA1,CapA1,A1), ...,**

  task(StartAn,PAn,EndAn,CapAn,An)],[limit(L)|Options])
  A1, ..., An: activity indentifiers
  limit(L): available capacity of the resource is **L**
  Options: options for different propagation algorithms

Producible/consumable resources

- Resource = **reservoir**
- **Activity consumes some quantity** of the resource **CapA<0** or **activity produces some quantity** of the resource **CapA>0**
- Minimal capacity is requested (consumption)
  and maximal capacity cannot be exceeded (production)



- Example: inventory for some products, activities producing them and activities using them in other production

Alternative resources

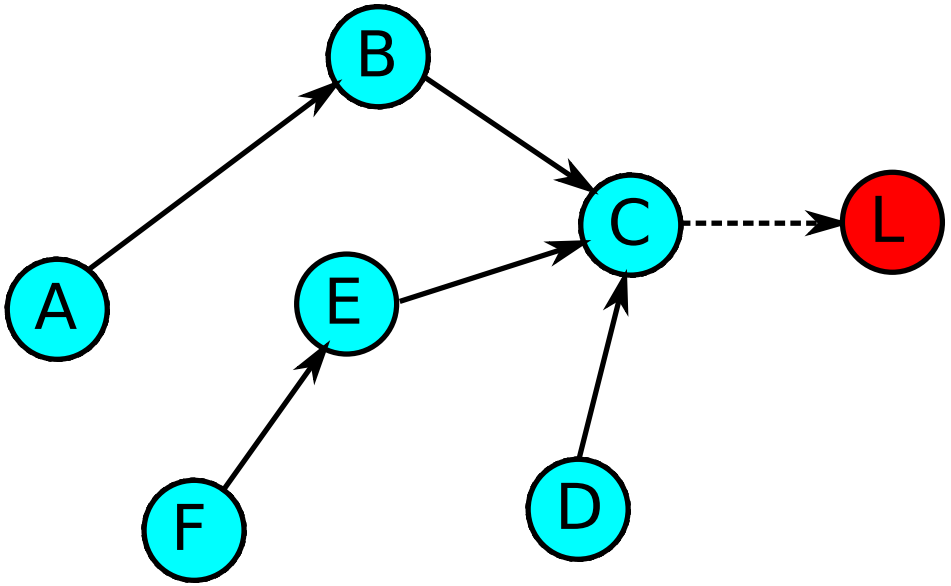
- Activity can be processed on a set of alternative resources
  - defined by the domain variable **ResourceA**
- One of them is selected for the activity
- Alternative unary resources
  - activity can be processed on any of the unary resources
  - can be modeled as one cumulative resource
    with resource capacity corresponding to the number of alternative unary resources
    - suitable for symmetric unary resources
- Example: any of the persons can process set of tasks

Optimization

Various criteria and objective function

Common criteria: **makespan**

- completion time of the last activity
- modeling
  - introduced a new additional activity L, PL=0
  - added precedence constraint
    for each activity T with no successor: T≪L



  - makespan = StartL

Constraint-based Scheduling: Modeling

21. **Machine Scheduling**
    - Machine scheduling with unary resource
    - Scheduling with cumulative resource
    - Job-shop problem
22. **Timetabling**
23. **Employees Scheduling**
24. **Exercises**
    - Operator Scheduling
    - Meeting Scheduling
    - Scheduling of Computational Jobs
    - Room Assignment

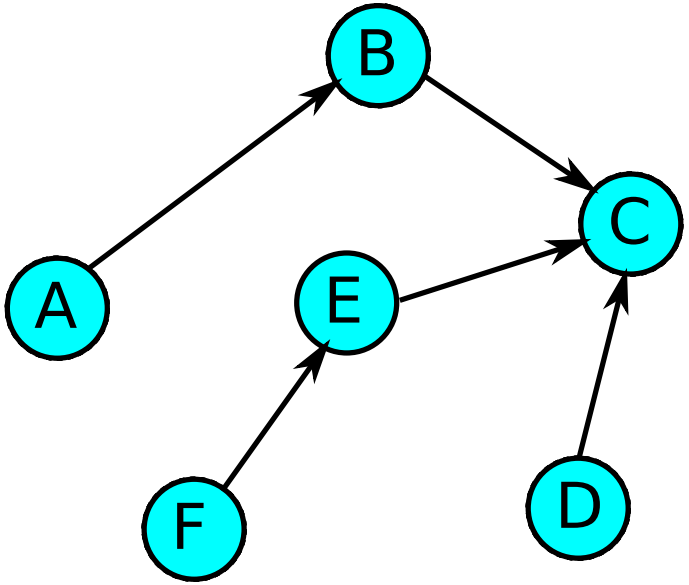
Machine scheduling with unary resource: problem & example

Problem: Create a schedule for several tasks with

- earliest (est) and latest completion time (lct)
- processing time (P)
- precedence constraints given by the graph

on machine of unit capacity such that the makespan is minimized

Task T	est(T)	lct(T)	PT
A	0	10	2
B	0	15	3
C	5	25	4
D	0	20	1
E	10	25	5
F	0	5	3



Machine scheduling with unary resource: variables

- Start time variables StartT for each task T
- lst(T) = lct(T) − PT

  StartT in est(T)..lst(T)

- Example:

Task T	est(T)	lct(T)	PT
A	0	10	2
B	0	15	3
C	5	25	4
D	0	20	1
E	10	25	5
F	0	5	3

StartA in 0..8, StartB in 0..12, StartC in 5..21,

StartD in 0..19, StartE in 10..20, StartF in 0..2

Machine scheduling with unary resource: constraints

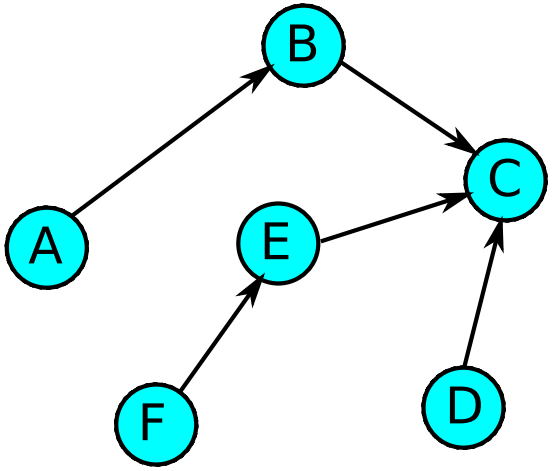
Precedence constraints for each tasks T1«T2

- StartT1 + PT1 #=< StartT2
- StartA+2 #=< StartB, StartB+3 #=< StartC,
  StartF+3 #=< StartE, StartE+5 #=< StartC, StartD+1 #=< StartC

Unary resource for all tasks T given by

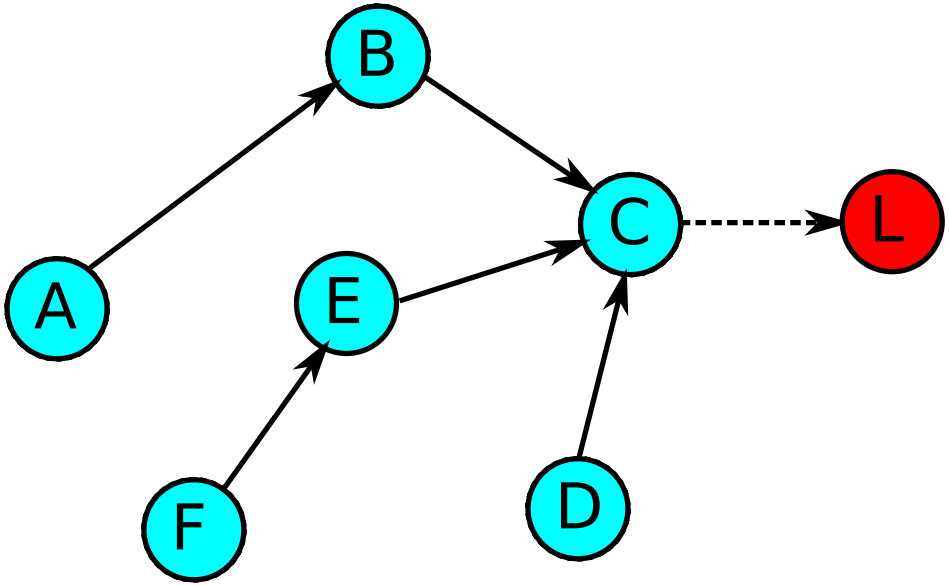
- start time variables StartT
- end time variables EndT #= StartT+PT
  cumulative([task(StartA,2,EndA,1,A), .., task(StartF,3,EndF,1,F)],Options)

Task T	est(T)	lct(T)	PT
A	0	10	2
B	0	15	3
C	5	25	4
D	0	20	1
E	10	25	5
F	0	5	3



Scheduling with unary resource: optimization

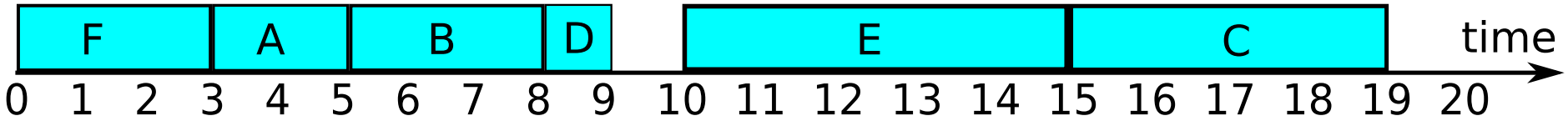
- New task L with PL=0 added
- Precedence constraints
  between L and tasks with no successor added
- Example: StartC+4 #=< StartL



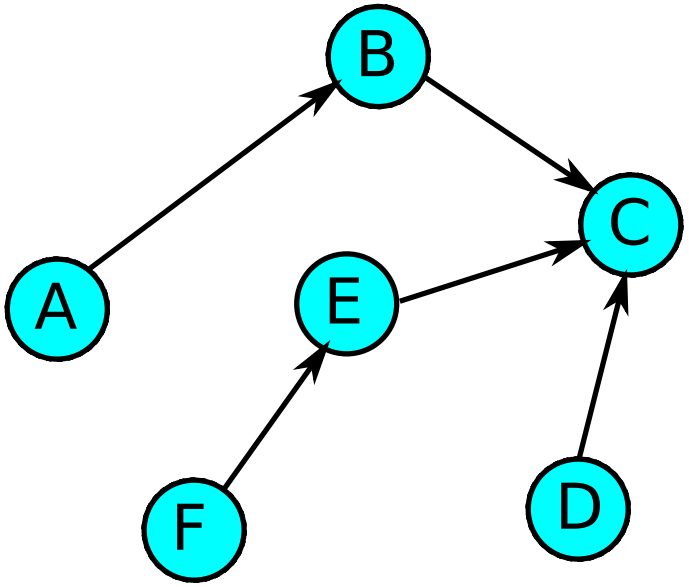
- minimize(Makespan) = minimize(StartL)

Scheduling with unary resource: solution

Solution:



Task T	est(T)	lct(T)	PT
A	0	10	2
B	0	15	3
C	5	25	4
D	0	20	1
E	10	25	5
F	0	5	3



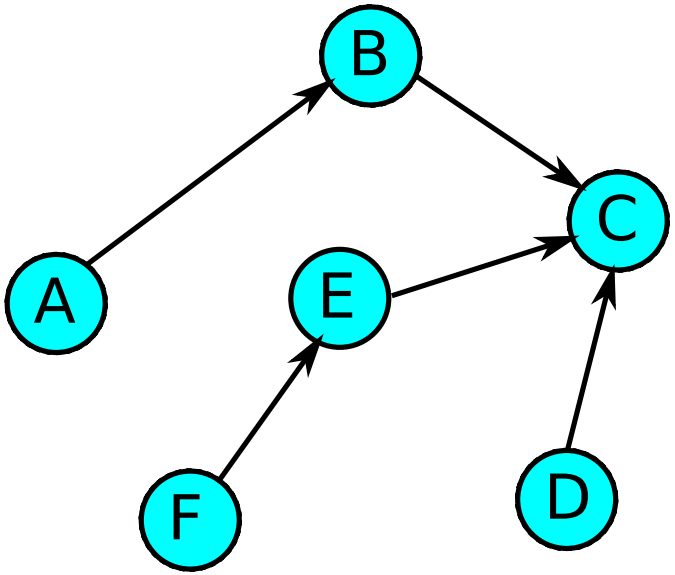
Scheduling with cumulative resource: problem & example

Problem: Create a schedule for several tasks with

- earliest (est) and latest completion time (lct)
- processing time (p)
- **requested capacity of the resource (cap)**
- precedence constraints given by the graph

on **machine of capacity 3** such that the makespan is minimized

Task T	est(T)	lct(T)	PT	**CapT**
A	0	10	2	**1**
B	0	15	3	**2**
C	5	25	4	**2**
D	0	20	1	**3**
E	10	25	5	**2**
F	0	5	3	**2**



Scheduling with cumulative resource: modeling

Same model as for scheduling with unary resource with

- unary resource replaced by cumulative resource

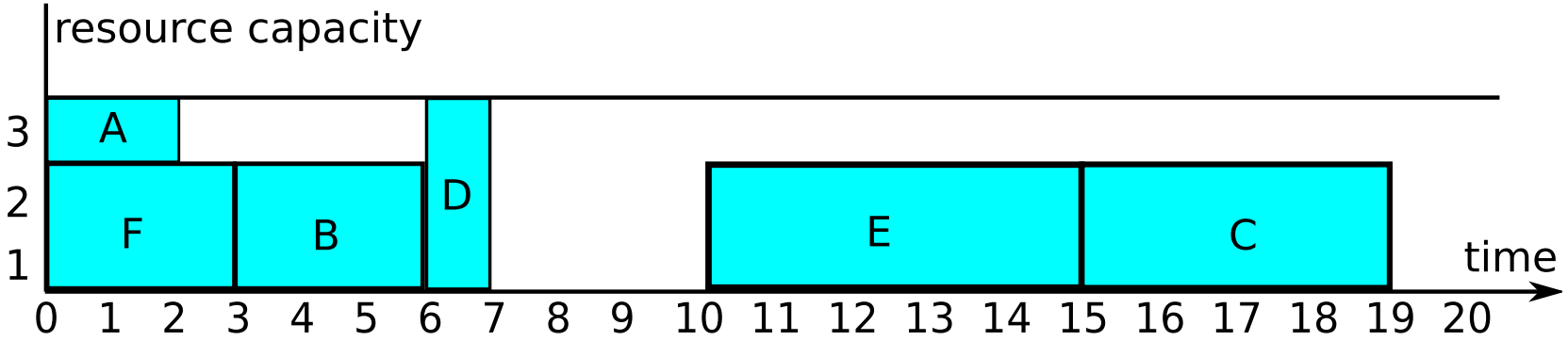
Cumulative resource for all tasks T given by

- start time variables StartT
- duration PT
- requested capacity of the resource
- example: cumulative([task(StartA,2,EndA,1,A), ..,
  task(StartF,3,EndF,2,F)],[limit(3)|Options])

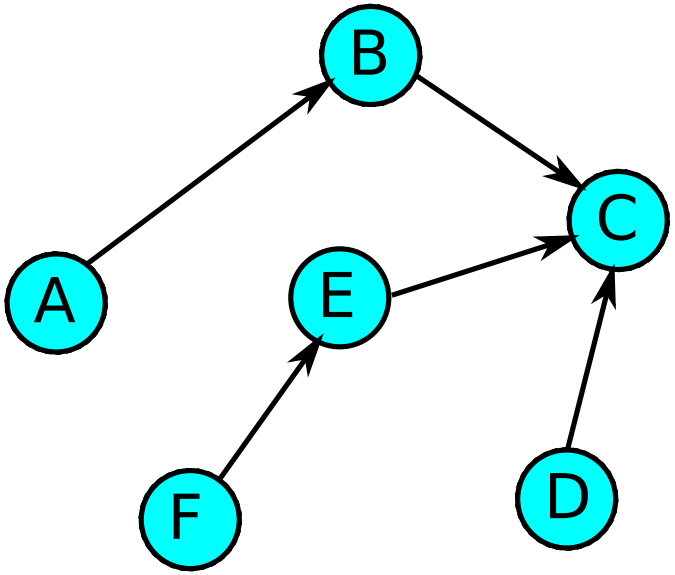
Task T	est(T)	lct(T)	PT	**CapT**
A	0	10	2	**1**
B	0	15	3	**2**
C	5	25	4	**2**
D	0	20	1	**3**
E	10	25	5	**2**
F	0	5	3	**2**

Scheduling with cumulative resource: solution

Solution:



Task T	est(T)	lct(T)	PT	**CapT**
A	0	10	2	**1**
B	0	15	3	**2**
C	5	25	4	**2**
D	0	20	1	**3**
E	10	25	5	**2**
F	0	5	3	**2**



Job-shop problem: problem

Create a schedule for several tasks such that

- each task consists of several jobs
- ordering of jobs for each task is fixed
- jobs of each tasks are processed on different dedicated machine
- machines have unit capacity
- makespan is minimized

Job-shop problem: example

- Machines: M1, M2, M3
- Tasks T1, T2 and T3 with jobs noted by (machine,task)

  T1: (3,1)«(2,1)«(1,1)

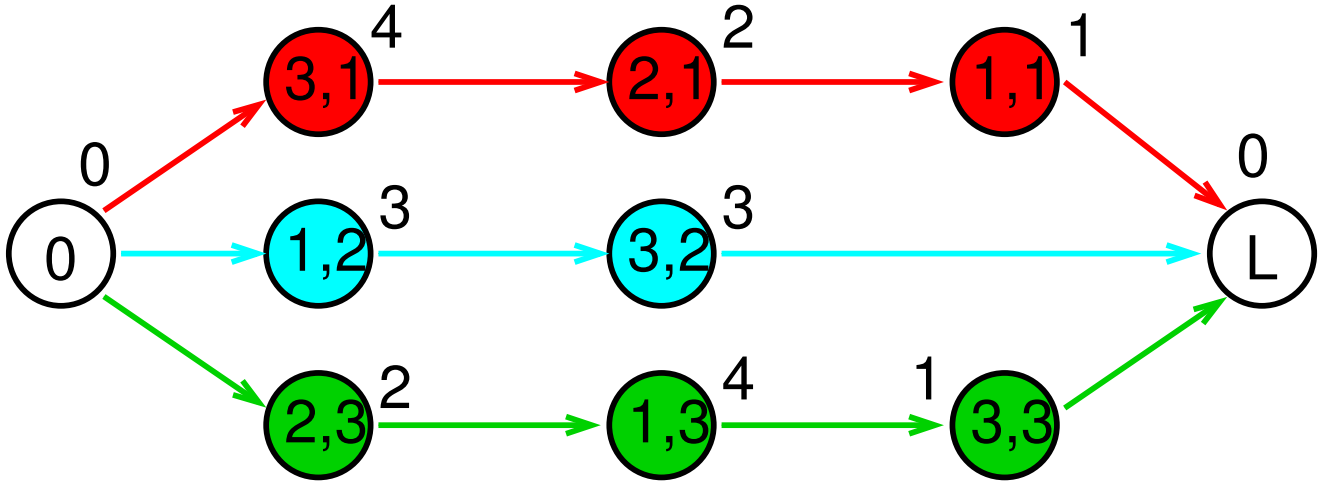
  T2: (1,2)«(3,2)

  T3: (2,3)«(1,3)«(3,3)
- Processing times:

  P31=4,P21=2,P11=1

  P12=3,P32=3

  P23=2,P13=4,P33=1



Additional first and last activities O and L with P0 = PL = 0

Job-shop problem: modeling

Variables and domains

- StartIJ start time variables for J task running on machine I

Constraints

- ordering of jobs modeled through precedence constraints

  Start31+P31 #=< Start21, Start21+P21 #=< Start11, ...
- unary resource constraint for each machine I
  with jobs (I,J) for all tasks J

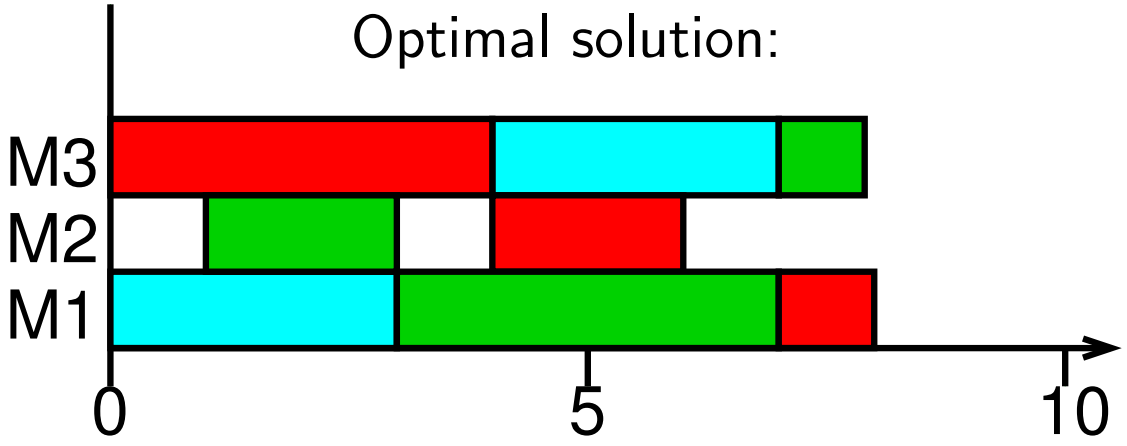
  1st machine: cumulative([task(Start11,1,End11,1,11), task(Start12,3,End12,1,12), task((Start13,4,End13,1,13)],Options),

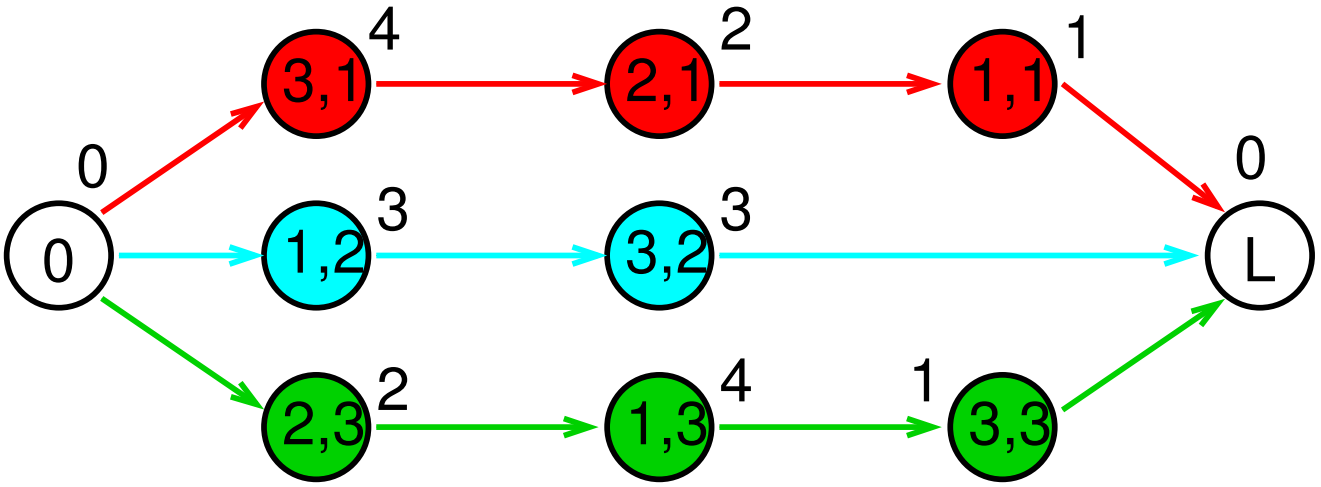
  2nd machine: cumulative(...), 3d machine: cumulative(...)

Optimization

- precedence constraints: Start11+1 #< StartL, Start32+3 #< StartL, Start33+1 #< StartL
- minimize(Makespan) = minimize(StartL)

Job-shop problem: solution





Class Timetabling: problem

Create schedule for N periods for classes with

- given duration
- given teacher
- given number of students
- prohibited time periods

Several classrooms (M) with specified number of seats are given.

There are sets of classes creating a curriculum

- no overlaps within curricula allowed

Class Timetabling: variables and domains

- **Class represents activity with given duration**
- Start time variables for each class **StartA**
  - StartA in 0..(N-1) (N number of periods)
  - for each prohibited time period i of the class A:
    StartA #\= ProhibitedAi

- **Classroom represents resource**
- **Classrooms are ordered by the number of seats**
  - smallest classroom = 0
  - largest classroom = M-1 (M number of rooms)
- Classroom variable for each class **ResourceA**
  - ResourceA in K..(M-1) such that K is the smallest classroom where the class fits by the number of students
  - example
    - 4 classrooms with sizes 20, 20, 40, 80 corresponding to 0,1,2,3
    - class A wants a room of the size 20: ResourceA in 0..3
    - class B wants a room of the size 40: ResourceB in 2..3
    - class C wants a room of the size 80: ResourceC = 3

Class Timetabling: resource constraints

Teacher represents a unary resource

- classes of one teacher cannot overlap
- all classes of each teacher are constrained by unary resource constraint
- classes are represented with their **StartA and PA** variables
- EndA #= StartA+PA
- for each teacher I and all his classes I1,...:
  cumulative([task(StartI1,PI1,EndI1,1,I1), ...], Options)

Curriculum represents a unary resource

- classes of one curriculum cannot overlap
- classes of one curriculum define one unary resource constraint
- classes are represented with their **StartA and PA** variables
- EndA #= StartA+PA
- for each curriculum J and all its classes J1,...:
  cumulative([task(StartJ1,PI1,EndJ1,1,J1), ...], Options)

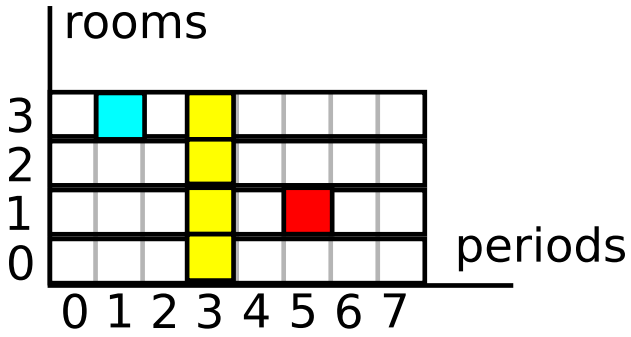
Class Timetabling: time and classrooms

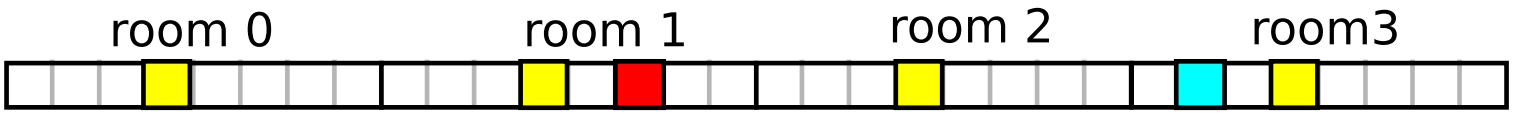
Constraint:

- **Each time at most one course must be taught at any classroom**

All classrooms together represents one unary resource

- all classes request this resource
- each class is encoded by activity with the starting time
  StartResourceA #= StartA + ResourceA * N
  and duration PA





For all classes A1,...:
cumulative([task(StartResourceA1,PA1,(StartResourceA1+PA1),1,A1),...],Options)

Employees Scheduling (rostering): problem

Create a one week schedule for employees working on shifts with

- several shift types
- minimal and maximal number of employees per shift
- minimal and maximal number of shift types per employee
- minimal and maximal number of working shifts per employee
- cost for each shift type to be paid to emploee working on it
- minimal cost

Employees scheduling: example

- Employees Peter, Paul, Mary, Jane, Keith, Alex, Anne
- One week schedule, i.e. 7 days
- Shift types: M morning, A afternoon, N night
- Each shift – M:3 employees, A:2-3 employees, N:1-2 employees
- Each employee – M: 2-3 shifts, A:1-3 shifts, N: 1-2 shifts
- Working shifts: 4-6
- Cost – CostM=10, CostA=11, CostN=13

Schedule	Mo	Tue	Wed	Thu	...
Peter	M	M	N	-	
Paul	A	A	-	M	
Mary	N	-	M	A	
...					

Employees scheduling: variables & domains

Matrix of variables corresponding to shifts of employees:

- PeterMo, PeterTue, . . . , PeterSun,
  PaulMo, PaulTue, . . . , PaulSun,
  MaryMo, MaryTue, . . . , MarySun,
  . . .

Domains:

- new shift type F (free) added to record free time shifts
- M, A, N, F corresponds to 1,2,3,4
- domain of variables corresponds to 1..4

Global cardinality constraint (revision)

Global constraint **global_cardinality(List, KeyCounts)**

- List: list of domain variables
- KeyCounts: list of Key-Count tuples with
  - Key: unique integer in the list of keys
  - Count: domain variable (or natural number)
- **Each Key is contained in List with cardinality Count**

Minimal and maximal number constraints I.

Schedule	Mo	Tue	Wed	Thu	...
Peter	M	M	N	F	...
Paul	A	A	F	M	
Mary	N	F	M	A	
...	...				

Minimal and maximal number of employees per shift

- new domain variables representing these numbers
  M1 in MinM1..MaxM1, A1 in MinA1..MaxM1, N1 in MinN1..MaxN2
- global cardinality constraint for each day
  global_cardinality([PeterMo,PaulMo,MaryMo,...], [1-M1,2-A1,3-N1])

Minimal and maximal number of shift types per employee

- new domain variables representing these numbers

  M2 in MinM2..MaxM2, A2 in MinA2..MaxM2, N2 in MinN2..MaxN2
- global cardinality constraint for each employee
  global_cardinality([PeterMo,PeterTue...,PeterSun], [1-M2,2-A2,3-N2])

Minimal and maximal number constraints II.

Minimal (MinW) and maximal number (MaxW) of working shifts per employee

- MinF = Days-MaxW . . . minimal number of free time shifts
- MaxF = Days-MinW . . . maximal number of free time shifts
- example
  - Days=7, MaxW=6 ⇒ MinF=1
  - Days=7, MinW=4 ⇒ MaxF=3
- new domain variable F in MinF..MaxF
- global_cardinality([PeterMo,PeterTue,. . . ,PeterSun], [4-F])
- better to addd to "Minimal and maximal number of shift types per employee" global cardinality constraint

  global_cardinality([PeterMo,PeterTue,. . . ,PeterSun], [1-M2,2-A2,3-N2,4-F])

Cost minimization

Schedule	Mo	Tue	Wed	Thu	...
Peter	M	M	N	F	...
Paul	A	A	F	M	
Mary	N	F	M	A	
...	...				

- M1, A1, N1 represent number of particular shifts on Monday
- All M1 variables for particular days can be summarized into M and same for A1 and A, N1 and N
  sum([M1Mon, M1Tue, ..., M1Sun], #= ,CostM), ...
- Total schedule cost corresponds to
  Cost #= M*CostM + A*CostA + N*CostN

Alternatively

- M2, A2, N2 represent number of particular shifts for Peter
- All M2 variables for all employees can be summarized into M

... and same for A2 and A, N2 and N

Exercises

Define **domain variables with their domains** and write a **set of constraints** together with possible **optimization criteria** to describe the model of particular constraint satisfaction problems in the following excercises.

Operator Scheduling

There are N tasks and M operators. Tasks should be scheduled within 4 weeks, each weeks starts on Monday and ends on Friday. Each task I has specified constant number of days DI to be processed. Find starting day of all tasks with given prerequisities:

1. Each day can be processed Limit tasks at most.
2. Each operator J has a list of tasks J1,...,JKJ to work on. In addition, each operator can process only one task at any time.
3. Each task I has specified weak WI to be completed at the latest.
4. Last task N must processed after completion of three first tasks 1,2,3.

Operator Scheduling: Solution

Variables for starting times:

- domain([T1,...,TN],1,20)

Constraints:

1. cumulative([task(T1,D1,(T1+D1),1,1),...,task(TN,DN,(TN+DN),1,N],
   [limit(Limit)|Options])
2. M constraints:
   cumulative([task(TJ1,DJ1,(JT1+JD1),1,1),...,
   task(TJKJ,DJKJ,(TKJ+DKJ),1,KJ], Options)
3. N constraints: TI+DI #=< WI*5
4. T1+D1 #=< TN, T2+D2 #=< TN, T3+D3 #=< TN

Meeting Scheduling

Find time and room for N meetings and K persons. Meetings will run from 8:00 to 17:00 and each meeting J has specified its constant duration DJ.

1. Meetings will take place in M rooms. One meeting can place at each time and room at most.
2. Earch person I will attend 4 meeting I1, I2, I3, I4.
3. Meeting A must take before meeting B.
4. Meeting C must take after meeting D.
5. There are no meetings during lunch time from 12:00 to 13:00.

Meeting Scheduling: Solution

Variables for starting time and room:

- domain([T1,...,TN], 0, 8)
  0 corresponds to 8:00, 8 corresponds to 16:00
- domain([R1,...,RN], 0, M-1)

Constraints:

1. variable for each meeting I: TimeRoomI #= RI*9 + TI
   cumulative([task(TimeRoom1,D1,TimeRoom1+D1,1,1),...,
   task(TimeRoomN,DN,TimeRoomN+DN,1,N)], Options)
2. K constraints:
   cumulative([task(TI1,DI1,TI1+DI1,1,I1),...,
   task(TI4,DI4,TI4+DI4,1,I4)], Options)
3. TA+DA #=< TB
4. TD+DD #=< TC
5. N constraints: TI #\= 4 (4 corresponds to 12:00)

Scheduling of Computational Jobs

There are N jobs to be processed on M processor cluster. Find starting time of all jobs given duration of the job I equal to DI and required number of processors by the job RI.

1. Each processor can process one job at any time.
2. Each job J requires memory EJ specified in GB. Any time the total allocated memory must be smaller or equal to 128GB available on the cluster.
3. Some jobs needs output of other jobs, so that they needs to be processed after completion of given jobs. Specifically, job U uses output of jobs V and job V uses output of W. In addition, job R uses output of S and T.

Also completion time of all jobs must be minimized.

Scheduling of Computational Jobs: Solution

Variables for starting time:

- sum([D1,...,DN], #= ,Sum), domain([T1,...,TN],0,(Sum-1))
  starting time of all jobs will be certainly smaller than sum of their durations

Constraints:

1. cumulative([task(T1,D1,(T1+D1),R1,1),...,task(TN,DN,(TN+DN),RN,N],
   [limit(M)|Options])
2. cumulative([task(T1,D1,(T1+D1),E1,1),...,task(TN,DN,(TN+DN),EN,N],
   [limit(128)|Options])
3. TU #>= TV+DV, TV #>= TW+DW
   TR #>= TS+DS, TR #>= TT+DT

Optimization:

- N constraints: End #>= TI+DI
- minimize(End)

Room Assignment

Find rooms for 6 courses A,B,C,D,E,F taking from 9 am to 4 am. Each course consists of several lectures and each lecture takes one hour. Each lecture has specified its time (starting time of all lectures for each course is given in the table). All lectures of one course must be at the same room. There are three rooms available and one lecture can be at any room at most.

Course/Time period	9 am	10 am	11 am	12 am	1 pm	2 pm	3 pm
A		+	+	+			
B		+	+				
C	+	+	+				
D				+	+	+	
E				+	+		
F					+	+	+

Room Assignment: Solution

Domain variables:

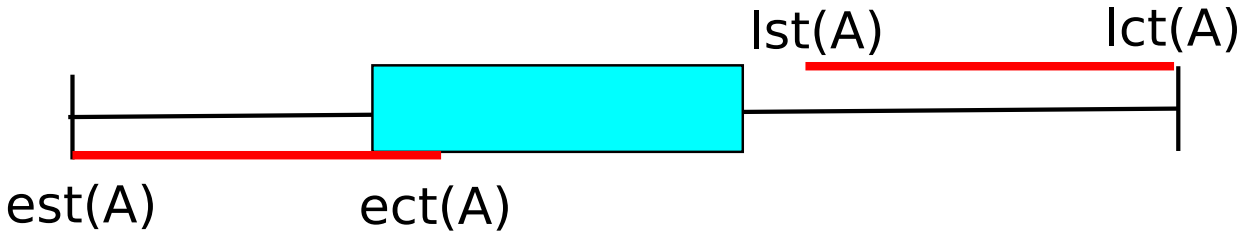
- domain([RoomA,RoomB,RoomC,RoomD,RoomE,RoomF],1,3)

Constraints:

- 10 am: all_different([RoomA,RoomB,RoomC])
- 11 am: all_different([RoomA,RoomB,RoomC])
  (not necessary, same as for 10 am)
- 12 am: all_different([RoomA,RoomD,RoomE])
- 1 pm: all_different([RoomD,RoomE,RoomF])
- 2 pm: RoomD #\= RoomF (not necessary, included in 1 pm)

Constraint-based Scheduling: Propagation

Notation



- start(A) domain variable for starting time of activity A
- end(A) domain variable for completion time of activity A
- p(A) domain variable for processing time of activity A

- est(A) earliest start time of activity A
- ect(A) earliest completion time of activity A
- lst(A) latest start time of activity A
- lct(A) latest completion time of activity A

- Ω is the set of activities
- $p(\Omega) = \sum_{A \in \Omega} p(A)$
- $est(\Omega) = \min\{est(A) \mid A \in \Omega\}$
- $lct(\Omega) = \max\{lct(A) \mid A \in \Omega\}$

Unary Resource for Unit Time Activities: all_different

teacher	min	max
X1	3	6
X2	**3**	**4**
X3	2	5
X4	**2**	**4**
X5	**3**	**4**
X6	1	6

U = {X2, X4, X5}, dom(U) = {2, 3, 4}:
{2,3,4} impossible for X1, X3, X6 $\Rightarrow$
X1 in 5..6, X3 = 5, X6 in {1} \/ (5..6)

Consistency:
$\forall \{X_1, \ldots, X_k\} \subset V : card\{D_1 \cup \cdots \cup D_k\} \geq k$

Inference rule

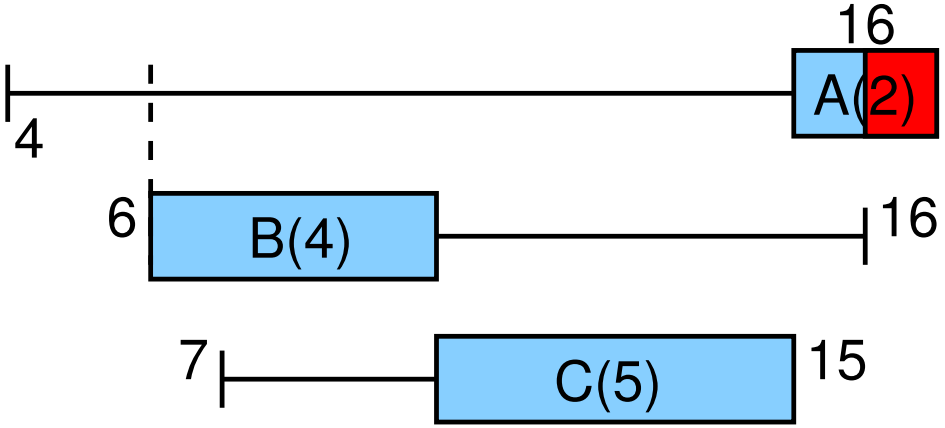
- $U = \{X_1, \ldots, X_k\}$, $dom(U) = \{D_1 \cup \cdots \cup D_k\}$
- $card(U) = card(dom(U)) \Rightarrow \forall v \in dom(U), \forall X \in (V - U), X \neq v$
- values in $dom(U)$ unavailable for other variables

Complexity

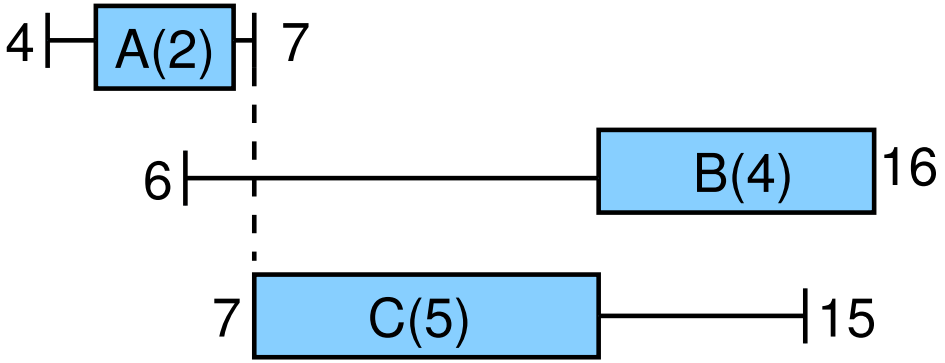
- $O(2^n)$: search through all subsets of the set of n variables
- $O(n \log n)$: changes of bounds propagated only (1998)

Edge finding: example

- What happens if activity A is not processed first?

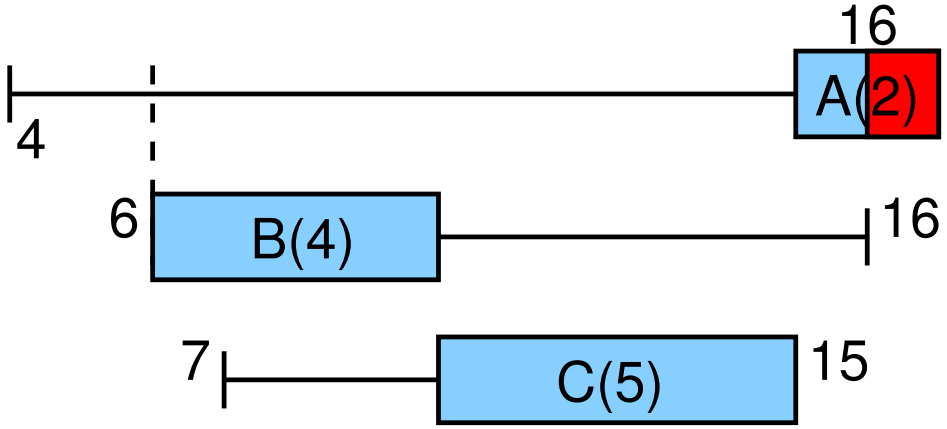


- Not enough time for A,B, and C and thus A must be first!

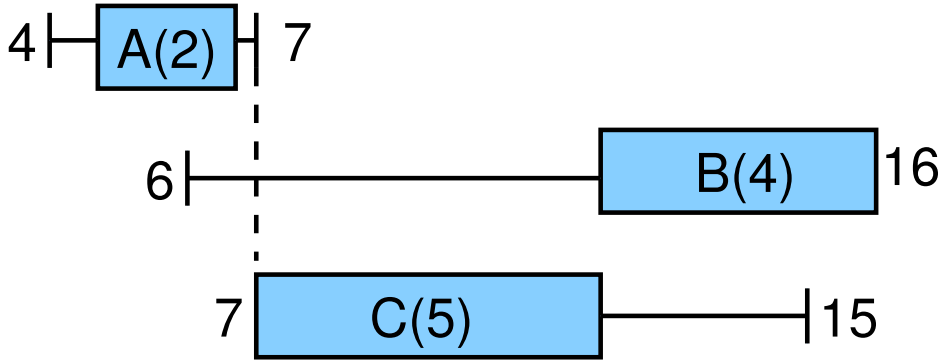


Edge finding: example with filtering rules

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



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



Edge finding: all filtering rules

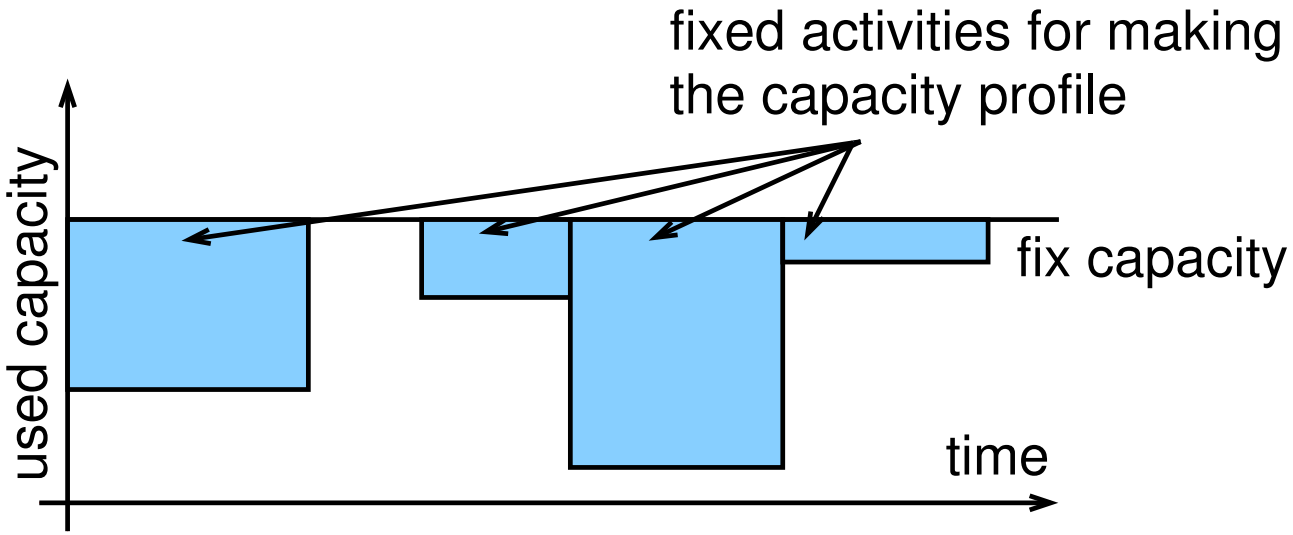
- **Edge-finding rules**
  - $p(\Omega \cup \{A\}) > lct(\Omega \cup \{A\}) - est(\Omega)$
    $\Rightarrow A << \Omega$
  - $A << \Omega \Rightarrow$
    $end(A) \leq \min\{lct(\Omega') - p(\Omega') \mid \Omega' \subseteq \Omega\}$

- **Edge-finding (symmetrical) rules**
  - similar rules for $\Omega << A$ and start(A)

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

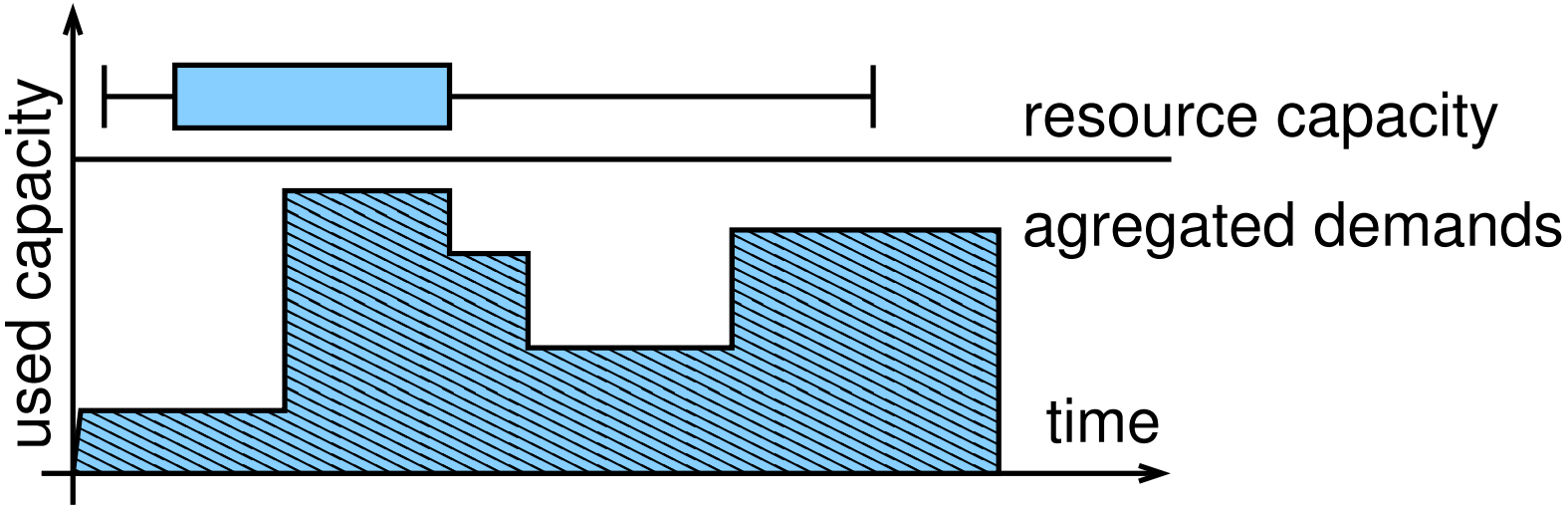
Cumulative resources

- 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

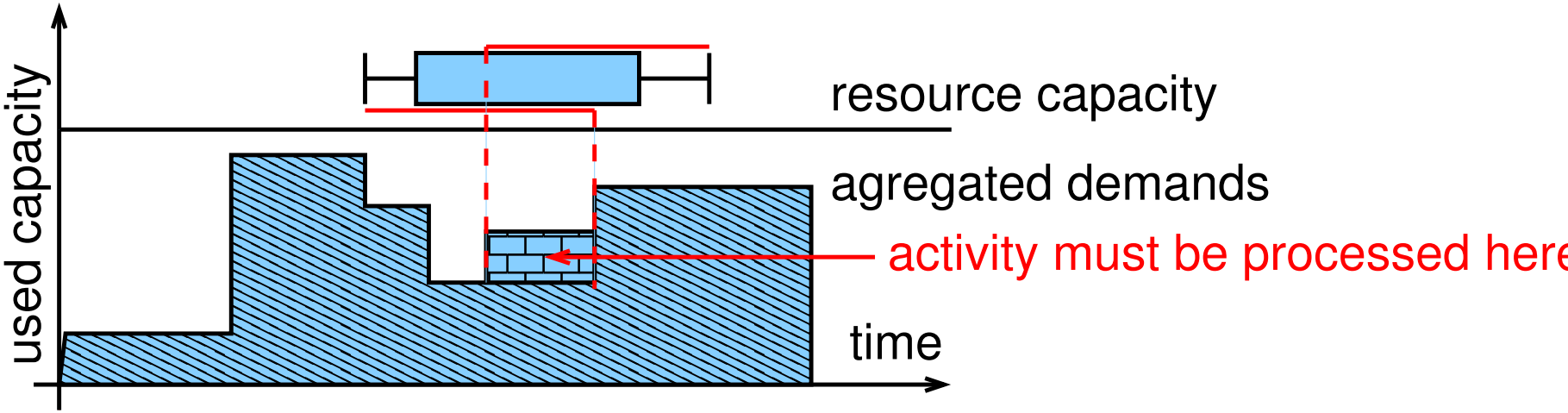


Aggregated demands

- Where is enough capacity for processing the activity?



- How aggregated demand is constructed?



Timetable constraint

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

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

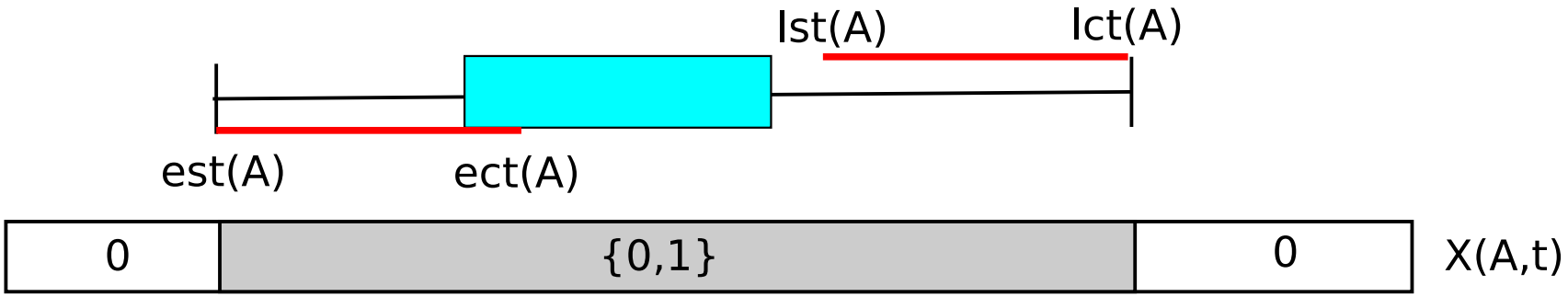
- **Timetable** for activity A is a set of Boolean domain variables $X(A, t)$ indicating whether A is processed in time t

$$\forall t \sum_{A_i} X(A_i, t) * cap(A_i) \leq MaxCapacity$$

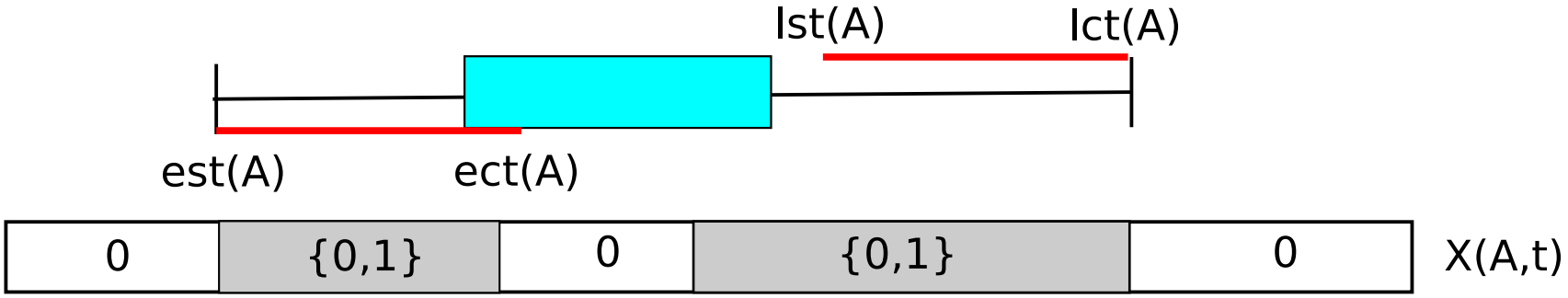
$$\forall t, i \; start(A_i) \leq t < end(A_i) \Leftrightarrow X(A_i, t)$$

Timetable constraint: filtering example

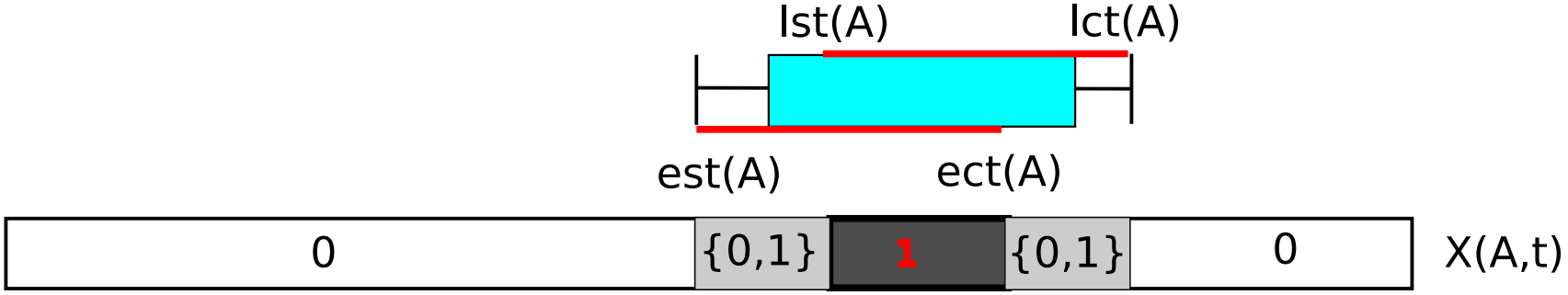
Initial situation



Some positions forbidden due to capacity



New situation



Constraint-based Scheduling: Search Strategies

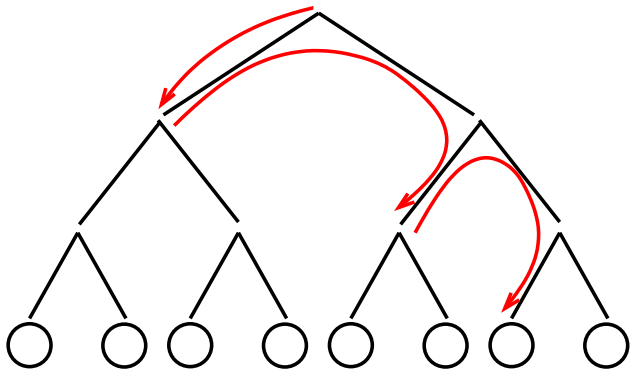
Search (revision)

Constraint propagation techniques are (usually) incomplete

$\Rightarrow$ search algorithm is needed to solve the "rest"

Labeling

- depth-first search (DFS/BT)
  - assign value to variable
  - propagate = make the problem locally consistent
  - backtrack in case of failure



- X in 1..5 $\equiv$ X=1 $\vee$ X=2 $\vee$ X=3 $\vee$ X=4 $\vee$ X=5

Generally search algorithm solves remaining disjunctions

- $X=1 \vee X\neq 1$ standard assignment
- $X<3 \vee X\geq 3$ domain splitting
- $X<Y \vee X\geq Y$ variable ordering (scheduling: tasks ordering)

Search: variable ordering

Which variable should be assigned first?

First-fail principle

- prefer variable with the hardest assignment
- for example variable with the smallest domain:
  domain can be emptied easily
- or variable with the most constraints:
  assignment of other variables constrain and
  make the domain smaller easily

Variable ordering defines **shape of the search tree**

- selection of variable with small domain size:
  small branching on this level . . . . . more options left for later
- selection of variable with large domain size:
  large branching on this level . . . . . less options left for later

Search: value ordering

What value should be chosen first?

Succeed-first principle

- prefers values with probably belongs to the solution
- for example the values with
  most supports in neighbouring variables
- this heuristic is usually problem specific

Value ordering defines the **order how the branches are explored**

Branching schemes in scheduling

Branching=resolving disjunctions

Traditional scheduling approaches

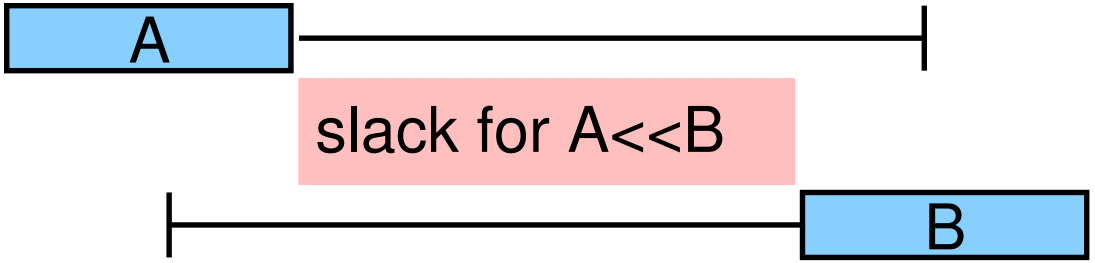
- take the **critical decisions first**
  - resolve bottlenecks, ...
  - defines the **shape** of the search treee
  - recall the **first-fail** principle
- prefer an **alternative leaving more flexibility**
  - defines **order** of branches to be explored
  - recall **succeed-first** principle

How to describe criticality and flexibility formally?

Slack

- **Slack** is a formal description of flexibility
- Slack for **a given order of two activities**

  "free time for shifting the activities"



$$slack(A << B) = \max(end(B)) - min(start(A)) - p(A) - p(B))$$

- Slack for **two activities** (without any ordering)

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

- Slack for **a group of activities**

  $$slack(\Omega) = \max(end(\Omega)) - min(start(\Omega)) - p(\Omega)$$

Order branching

$A << B \quad \vee \quad \neg A << B$

What activities A, B should be ordered first?

- the most critical pair (first-fail)
- **the pair with the minimal slack**$(\{A, B\})$

What order of activities A and B should be selected?

- the most flexible order
- the order with **with the maximal slack**$(A??B)$

$O(n^2)$ choice points

First/last branching

$(A << \Omega \vee \neg A << \Omega) \quad \vee \quad (\Omega << A \vee \neg\Omega << A)$

Should we look for the first or last activity?

- look to the set of possible candidates for first activity and to the set of possible candidates for last activities
- select a **smaller set** from these (first-fail)
  - smaller number of candidates means that it is harder to find a suitable candidate

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 acticity 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