Planning and Scheduling Methodologies

Scheduling Algorithms

Henrique Lopes Cardoso (hlc@fe.up.pt)
April 2015
Complexity of Scheduling Problems

“intractable”, unlikely that polynomial time algorithms exist

Polynomial-time solvable

NP-hard

efficient (polynomial time) algorithms exist
Complexity of Scheduling Problems

The diagram illustrates various scheduling problems and their complexity:

- **FFc||C_max**
- **Im||C_max**
- **P2||C_max**
- **F2||C_max**
- **1||C_max**

The problems are categorized into Easy and Hard:

- Easy problems:
  - **1||C_max**
  - **1||L_max**
  - **1|prmp|L_max**

- Hard problems:
  - **FFc||C_max**
  - **Im||C_max**
  - **P2||C_max**
  - **F2||C_max**
  - **Pm||L_max**
  - **1|r_j|L_max**
  - **1|r_j, prmp|L_max**
Scheduling Algorithms

**Exact Algorithms**
Find optimal solutions.
- Branch and Bound

**Approximation Algorithms**
Find solutions guaranteed to be within a fixed percentage of the optimum.

**Heuristic Algorithms**
Do not guarantee optimal solutions. Find reasonably good solutions.

**Constructive Heuristics**
- Dispatching rules

**Improvement Heuristics**
Local search
- Simulated Annealing
- Tabu-Search
- Genetic Algorithms
- Ant Colony Optimization
Scheduling Algorithms

![Graph showing error bound over time with different techniques: Dispatching rules, Local Search, and Branch & Bound.]

- Dispatching rules
- Local Search
- Branch & Bound

Error bound
Feasible Solutions

Time
Dispatching Rules

• A dispatching rule is a function of attributes of the jobs and/or the machines
  – Job attributes: weight, processing time, due date, ...
  – Machine attributes: speed, number of jobs waiting for processing, total amount of processing waiting in queue, ...
  – Useful when attempting to find a reasonably good schedule with regard to a single objective (e.g. makespan, total completion time or maximum lateness)
  – May yield optimal schedules in some machine environments

• Dynamic vs. static rules
  – Static rules are not time dependent, but simply a function of the job and/or machine data
    • E.g. Weighted Shortest Processing Time (WSPT)
  – Dynamic rules are time dependent
    • E.g. Minimum Slack (MS): order jobs according to $\max(d_j - p_j - t, 0)$
    • The priority relationship between two jobs $j$ and $k$ may change over time
Dispatching Rules

- **Examples:**

<table>
<thead>
<tr>
<th>Rule 1</th>
<th>Rule 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIRO – <em>Service In Random Order</em></td>
<td>CP – <em>Critical Path</em></td>
</tr>
<tr>
<td>ERD – <em>Earliest Release Date first</em></td>
<td>LNS – <em>Largest Number of Successors first</em></td>
</tr>
<tr>
<td>EDD – <em>Earliest Due Date first</em></td>
<td>SST – <em>Shortest Setup Time first</em></td>
</tr>
<tr>
<td>MS – <em>Minimum Slack first</em></td>
<td>LFJ – <em>Least Flexible Job first</em></td>
</tr>
<tr>
<td>SPT – <em>Shortest Processing Time first</em></td>
<td>LAPT – <em>Longest Alternate Processing Time first</em></td>
</tr>
<tr>
<td>WSPT – <em>Weighted Shortest Processing Time first</em></td>
<td>SQ – <em>Shortest Queue first</em></td>
</tr>
<tr>
<td>LPT – <em>Longest Processing Time first</em></td>
<td>SQNO – <em>Shortest Queue at the Nest Operation first</em></td>
</tr>
</tbody>
</table>

- Some of these rules yield optimal schedules in some machine environments and are reasonable heuristics in others.
Optimal Dispatching Rules

• **WSPT** is optimal for \(1|| \sum w_j C_j\)

• **CP** is optimal for
  - \(P_m|p_j = 1, intree|C_{max}\)
  - \(P_m|p_j = 1, outtree|C_{max}\)
  - \(P_m|p_j = 1, outtree|\sum C_j\)

• **LFJ** is optimal for \(P_2|p_j = 1, M_j|C_{max}\)

• **SPT** is optimal for
  - \(1|| \sum C_j\)
  - \(P_m|| \sum C_j\)
  - \(P_m|pmtn|\sum C_j\)

• **Preemptive EDD** is optimal for \(1|r_j, pmtn|L_{max}\)

• ...

Composite Dispatching Rules

- **Composite** dispatching rule
  - A ranking expression that combines a number of elementary dispatching rules
  - Each dispatching rule contributes to the ranking expression using a scaling parameter (weight)
  - Useful when addressing multi-objective functions

- Example
  - $1|| \sum w_j T_j$ is NP-hard
    - B&B is prohibitively time consuming for only 30 jobs
    - Need for a reasonably good schedule with a reasonable computational effort
      - WSPT is optimal when all release and due dates are zero
      - EDD or MS are optimal when all due dates are sufficiently loose and spread out
    - Combination of WSPT and MS: *Apparent Tardiness Cost (ATC)*
Branch and Bound (B&B)

- Idea: intelligently enumerate all feasible solutions

- Build a search tree where the scheduling problem is successively reduced into smaller problems
  - Root node corresponds to the original problem to be solved
  - Each node at a level $p$ in the search tree represents a partial sequence of $p$ operations (the remaining operations are thus a sub-problem of the original problem)

- Branching
  - expand a node by extending the partial sequence of operations

- Bounding
  - **Lower bound** ($LB$): the cost for all feasible solutions that can be obtained from a node is at least $LB$
  - **Upper bound** ($UB$): cost of a known solution (the best so far)
  - If $LB \geq UB$, this partial sequence and all its subsequent descendants can be safely disregarded (i.e. this node cannot yield a better solution)
B&B

(a)

(b)

(c)

* = does not contain optimal solution
B&B – bounds calculation

• **Upper bound**
  – at the beginning, use some *heuristic* to estimate a maximum possible cost for the optimal solution
    • should be as small as possible, so that we can cut more branches
  – update with the cost of the best solution found so far

• **Lower bound**
  – Bounding function $g(P_i)$
  – Objective function $f(P_i)$: best solution going through $P_i$
    • $g(P_i) \leq f(P_i)$ for all nodes $P_i$ in the tree
    • $g(P_i) = f(P_i)$ for all leaves in the tree
    • $g(P_i) \geq g(P_j)$ if $P_j$ is the father of $P_i$ ($g$ gets closer bounds as we go on)
B&B – bounding functions

- **Bounding function choice**
  - Accurate bounds require much computation time
  - Less accurate bounds can be computed quickly

- **Lower bound**
  - Weak bounding function: cost of the partial sequence already scheduled
  - Stronger bounding function: estimate closer to the best solution of the sub-problem starting in $P_i$
    - enlarge the set of feasible solutions by leaving out some of the constraints of the original problem *(relaxation)*
      - in some cases, optimal dispatching rules can be used for relaxed problems
    - modify the objective function while ensuring that for all feasible solutions the modified function is lower than or equal to the original function
B&B – branching decisions

• Strategy for selecting the node to branch
  – Best-first search: select node with the lowest lower bound
    • may behave similar to breadth-first search ⇒ memory problems: exponential growth in the number of nodes
  – Depth-first search: select node at largest level in the search tree
    • number of sub-problems to store at the same time is bounded by the number of levels times the maximum number of children of any node
    • best-first can be used to select among nodes at the same level

• Branching rule
  – Size of each sub-problem should be smaller than the original problem
  – Generated sub-problems should be disjoint to avoid finding the same feasible solutions in different nodes of the search tree
B&B – example

- $1|r_j|L_{\text{max}}$

<table>
<thead>
<tr>
<th>jobs</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_j$</td>
<td>4</td>
<td>2</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>$r_j$</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>$d_j$</td>
<td>8</td>
<td>12</td>
<td>11</td>
<td>10</td>
</tr>
</tbody>
</table>

Lower bound generated using preemptive EDD rule
(optimal for $1|r_j, pmtn|L_{\text{max}}$, and thus a lower bound for $1|r_j|L_{\text{max}}$)

$L_{\text{max}} = L_2 = 17 - 12 = 5$
B&B – termination

- B&B
  - Time/space consuming: number of nodes to be considered is very large
  - Suitable for small problems only
  - Example: single machine with n jobs
    - from level 0: \( n \) branches to \( n \) nodes at level 1
    - from level 1: \( n-1 \) branches from each node to \( n(n-1) \) nodes at level 2
    - at level \( n \) there are \( n! \) nodes

- B&B is often terminated before optimality is reached
  - complete solution with cost \( U \)
  - lowest lower bound \( LB \) of all non-leaf nodes provides a lower bound on the optimal cost

\[
\frac{U - OPT}{OPT} \leq \frac{U - LB}{LB}
\]
Beam Search

- **Filtered beam search**
  - Limit the number of nodes that are evaluated at each level: the most *promising* nodes
  - *Beam width*: number of nodes retained at each level

- How to evaluate?
  - Careful evaluation is time consuming
  - Crude prediction is quick but may discard good solutions
  - *Filter*: evaluate crudely all the nodes, and thoroughly evaluate the nodes that pass the filter (filter width > beam width)

- **Dispatching rules** can generate schedules fast, providing good upper bounds
Beam Search – example

- $1 || \sum w_j T_j$

<table>
<thead>
<tr>
<th>jobs</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_j$</td>
<td>10</td>
<td>10</td>
<td>13</td>
<td>4</td>
</tr>
<tr>
<td>$d_j$</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>$w_j$</td>
<td>14</td>
<td>12</td>
<td>1</td>
<td>12</td>
</tr>
</tbody>
</table>
Local Search

• Iterative improvement
  – Start with a complete solution (schedule)
  – Try to find a better solution by manipulating the current solution

• Local search
  – Search in the “neighborhood” of the current solution
  – Two solutions are neighbors if one can be obtained through a well-defined modification of the other

• Design criteria
  i. Schedule representation
  ii. Neighborhood design
  iii. Search process within the neighborhood
  iv. Acceptance-rejection criteria
  v. Stopping criteria
Representation and Neighborhood

• 1||
  – Representation:
    • permutation of the $n$ jobs
  – Neighborhood:
    • single adjacent pairwise interchange
      – $n-1$ schedules in the neighborhood
    • pick arbitrary job and insert it in another position
      – $n(n-1)$ neighbors, some identical
Representation and Neighborhood

- $Jm||C_{max}$
  - Representation:
    - $m$ consecutive strings, each being a permutation of $n$ operations
  - Neighborhood:
    - **One Step Look-Back Adjacent Interchange**
      - adjacent pairwise interchange between two operations on the critical path (same machine)
      - look-back at previous operation on job whose operation now occurs first and interchange with previous operation on the same machine
    - **[One|Multi] Step Look-[Back|Ahead] Adjacent Interchange**
Search, Accept and Stop Criteria

• Search
  – Random selection
  – Select most promising neighbors

• Acceptance-rejection criteria
  – Design aspect that distinguishes a local search procedure the most
  – Deterministic vs probabilistic approaches

• Stopping criteria
  – Number of iterations
  – Convergence to local optima
Hill-Climbing

- Greedy local search

```python
function Hill-Climbing(problem) returns a state that is a local maximum
inputs: problem, a problem
local variables: current, a node
    neighbor, a node

current ← MAKE-NODE(INITIAL-STATE[problem])
loop do
    neighbor ← a highest-valued successor of current
    if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
    current ← neighbor
end
```

- Variations
  - Steepest ascent
  - Stochastic: randomly choose an uphill move (higher steepness, higher probability)
  - First-choice: generate random successors and pick when better than current state
Hill-Climbing

• Problems
  • Shoulders: allow a number of *sideway moves*
  • Local maxima: *random-restart*
    • run hill-climbing several times from different random initial states

• Approaches
Simulated Annealing

- Idea: escape local maxima by allowing some “bad” moves, while gradually decreasing their size and frequency

```python
function SIMULATED-ANNEALING(problem, schedule) returns a solution state

inputs: problem, a problem
        schedule, a mapping from time to “temperature”

local variables: current, a node
                 next, a node
                 T, a “temperature” controlling prob. of downward steps

current ← MAKE-NODE(INITIAL-STATE[problem])
for t ← 1 to ∞ do
    T ← schedule[t]
    if T = 0 then return current
    next ← a randomly selected successor of current
    ΔE ← VALUE[next] - VALUE[current]
    if ΔE > 0 then current ← next
    else current ← next only with probability e^{ΔE/T}
```

- finds global optimum if the temperature is decreased “slowly enough”
- typically the best solution found so far is stored
Tabu Search

- Acceptance/rejection criteria
  - Simulated Annealing: probabilistic
  - Tabu Search: deterministic

- Tabu-list
  - List of disallowed moves: a fixed number of recently visited solutions (FIFO)
    - sometimes operators used in generating neighbors are stored, not complete solutions
  - Avoid cycles smaller than the size of the tabu-list

- Algorithm

```
s ← generateInitialSolution()
best_s ← s
tabu_list ← new Queue(size)
repeat
  s’ ← generateNonTabuNeighbor(s, tabu_list)
  tabu_list.push(s)
  best_s ← best(best_s, s’)
  s ← s’
until stopping criterion
return best_s
```
Tabu Search – example

• $1|| \sum w_j T_j$
  – Neighborhood: adjacent pairwise interchanges
  – Tabu-list: pairs of jobs that were swapped; tabu-list size = 2
  – First schedule $S_1 = 2,1,4,3 \rightarrow \sum w_j T_j = 500$

<table>
<thead>
<tr>
<th>jobs</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_j$</td>
<td>10</td>
<td>10</td>
<td>13</td>
<td>4</td>
</tr>
<tr>
<td>$d_j$</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>$w_j$</td>
<td>14</td>
<td>12</td>
<td>1</td>
<td>12</td>
</tr>
</tbody>
</table>

Neighbors of $S_1$: 1,2,4,3 (480); 2,4,1,3 (436); 2,1,3,4 (652)
Best (non-tabu) neighbor of $S_1$ is $S_2 = 2,4,1,3$
Tabu-list: $\langle (1,4) \rangle$

Neighbors of $S_2$: 4,2,1,3 (460); $\textbf{2,1,4,3 (500)}$; 2,4,3,1 (608)
Best (non-tabu) neighbor of $S_2$ is $S_3 = 4,2,1,3$
Tabu-list: $\langle (2,4), (1,4) \rangle$

Neighbors of $S_3$: $\textbf{2,4,1,3 (436)}$; 4,1,2,3 (440); 4,2,3,1 (632)
Best (non-tabu) neighbor of $S_3$ is $S_4 = 4,1,2,3$
Tabu-list: $\langle (1,2), (2,4) \rangle$

Neighbors of $S_4$: 1,4,2,3 (408); $\textbf{4,2,1,3 (460)}$; 4,1,3,2 (586)
Best (non-tabu) neighbor of $S_4$ is $S_5 = 1,4,2,3$
Tabu-list: $\langle (1,4), (1,2) \rangle$

...
Local Beam Search

• Keep $k$ states under analysis rather than just one
  – Start with $k$ randomly generated states
  – Generate all successors of all $k$ states
  – Keep the best $k$ successors

• Similar to $k$ random restarts in parallel, but with “information exchange”

• Susceptible to local optima, if the $k$ states tend to be in the same region of the state space
  – Variation: stochastic beam search, where successors are randomly chosen (as in stochastic hill-climbing)
Genetic Algorithms

- Stochastic beam search + generate successors from pairs of states

- Solutions are individuals or members of a population
  - Each individual has a fitness

- In each iteration there is a generation of individuals
  - Survivors from the previous generation
  - New individuals (offspring) generated through
    - crossover: combining parts of different (two) solutions
      - raises the level of granularity at which the search operates
    - mutation: change within the individual
      - Fittest individuals reproduce, least fit die

- Can be applied without knowing much about structural properties of the problem
Genetic Algorithms

function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual
repeat
    new-population ← empty set
    loop for i from 1 to SIZE(population) do
        x ← RANDOM-SELECTION(population, FITNESS-FN)
        y ← RANDOM-SELECTION(population, FITNESS-FN)
        child ← REPRODUCE(x, y)
        if (small random probability) then child ← MUTATE(child)
        add child to new-population
    population ← new-population
until some individual is fit enough, or enough time has elapsed
return the best individual in population, according to FITNESS-FN

function REPRODUCE(x, y) returns an individual
inputs: x, y, parent individuals
        
n ← LENGTH(x)
        c ← random number from 1 to n
return APPEND( SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
Genetic Algorithms

• Crossover helps iff substrings are meaningful components
Genetic Algorithms

- Linear Order Crossover (LOX)
  1. select at random a subsequence of jobs from one parent
  2. generate a new offspring with that subsequence in the same position
  3. discard the jobs in the subsequence from the second parent
  4. place remaining jobs in the second parent in the unfilled positions of the offspring, in the same order
Ant Colony Optimization

• Inspired by trail following behavior of ant colonies
  – Ants communicate indirectly through changes in pheromone trails
  – Trails comprise a dynamic memory structure containing information on the quality of previously obtained results
    • Pheromones indicate path quality

• General algorithm

```
initialize pheromone trails
best_s ← null
repeat
    generate l solutions by exploiting pheromone trails
    apply local search to each of the l solutions
    best_s ← best(best_s, l)
    update pheromone trail values (evaporation and ant deposit)
until stopping criterion
return best_s
```
Market/Agent-based Approaches

- Jobs and Machines are agents
  - Job Agent (JA)
    - Budget
  - Machine Agent (MA)
    - Fixed cost per time unit (processing or idle)

- Task allocation
  - JA sends call for bids
    - Preferred time frame: a function of final due date and estimation of processing time left
  - MA bids
    - Time to start processing the operation
    - Price: higher than fixed cost brings profit
    - Multiple bids to the same JA for different time periods/prices
    - Simultaneous bids to different JAs: time periods should not overlap
  - JA awards the operation
    - Decision based on processing time and pricing
Market/Agent-based Approaches

- Budgetary constraints
  - Budget: a function of job weight, tightness of due date, total processing time
  - Indication regarding the priority of the job

- Information infrastructure
  - Complete information vs. local information
  - Statistical information: average amount of processing, distribution of processing times and weights, due date tightness factors, ...

- Bidding and pricing rules
  - Strategies:
    - goodness of fit
    - anticipated supply/demand
    - competition
Market/Agent-based Approaches

- $Jm|r_j|\sum w_j T_j$

<table>
<thead>
<tr>
<th>job</th>
<th>$w_j$</th>
<th>$r_j$</th>
<th>$d_j$</th>
<th>machine sequence</th>
<th>processing times</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>5</td>
<td>24</td>
<td>1,2,3</td>
<td>$p_{11} = 5$, $p_{21} = 10$, $p_{31} = 4$</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>18</td>
<td>3,1,2</td>
<td>$p_{32} = 4$, $p_{12} = 5$, $p_{22} = 6$</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>0</td>
<td>16</td>
<td>3,2,1</td>
<td>$p_{33} = 5$, $p_{23} = 3$, $p_{13} = 7$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>time</th>
<th>decision data</th>
<th>allocation</th>
</tr>
</thead>
</table>
| 0    | $j_2, j_3 \mapsto m_3$
     | $w_2 = w_3$
     | $d_3 < d_2$
     | $j_3 \mapsto m_3$ |
| 5    | $j_1 \mapsto m_1$
     | $j_2 \mapsto m_3$
     | $j_3 \mapsto m_2$
     | $j_1 \mapsto m_1$
     | $j_2 \mapsto m_3$
     | $j_3 \mapsto m_2$ |
| 10   | $j_1 \mapsto m_2$
     | $j_2, j_3 \mapsto m_1$
     | $\text{slack}_2 = -3$
     | $\text{slack}_3 = -1$
     | $j_1 \mapsto m_2$
     | $j_2 \mapsto m_1$ |
Market/Agent-based Approaches

• In simple settings, e.g. deterministic and with identical machines
  – If JAs send out call for bids only for the next operation, bidding/pricing rules are comparable to dispatching rules
  – Optimization techniques will probably perform better

• Make more sense in complex scheduling environments
  – Simultaneous calls for bids for a number of operations
    • Jobs may not be ready at the time of a commitment
  – Different machines: bid according to speed
    • Take into account competitors’ speeds
  – Distributed scheduling in environments with various forms of randomness
    • Optimization algorithms are hard to develop in these settings
Constraint Programming (CP)

• Constraint Satisfaction Problem (CSP)
  – \( \langle X, D, C \rangle \)
    • \( X \) is a set of variables
    • \( D \) contains the domains for variables in \( X \)
    • \( C \) is a set of constraints, limiting the values that variables in \( X \) may simultaneously have

• Solution of a CSP
  – Instantiation of each variable \( x_i \) in \( X \) with a value \( v_i \) in \( D_{x_i} \) such that every constraint in \( C \) is satisfied

• Goal
  – Determine if there is a solution
  – Find any solution
  – Find every solution
  – Find the optimal solution
CP – solving a CSP

• Steps
  1. Declare variables and domains
     • Model the problem at stake
  2. Impose constraints
     • Constrain the search space by defining and propagating constraints
  3. Search for solutions
     • Generate solutions using search through the remaining space

• Search will depend on the goal
  – If any solution is enough, search with backtracking
  – If optimal solution is sought, use branch and bound
CP – propagating constraints

- **Consistency**
  - eliminate redundant values from the variables domains (i.e., values that cannot be used in any solution)

- **Constraint propagation** should ensure *node, arc and path* consistency

\[
\begin{align*}
\text{x} & \quad \text{x} + 2 < \text{y} & \quad \text{y} \\
\{1..5\} & \quad \quad \{1..5\}
\end{align*}
\]

\[
\begin{align*}
\text{x} & \quad \text{x} + 2 < \text{y} & \quad \text{y} \\
\{1,2\} & \quad \quad \{1..5\}
\end{align*}
\]

\[
\begin{align*}
\text{x} & \quad \text{x} + 2 < \text{y} & \quad \text{y} \\
\{1,2\} & \quad \quad \{4,5\}
\end{align*}
\]
CP – propagating constraints

<table>
<thead>
<tr>
<th>Task</th>
<th>Duration</th>
<th>Precedes</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
<td>B, C</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>D</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variable</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>{0}</td>
</tr>
<tr>
<td>A</td>
<td>{0..11}</td>
</tr>
<tr>
<td>B</td>
<td>{0..11}</td>
</tr>
<tr>
<td>C</td>
<td>{0..11}</td>
</tr>
<tr>
<td>D</td>
<td>{0..11}</td>
</tr>
<tr>
<td>End</td>
<td>{0..11}</td>
</tr>
</tbody>
</table>

```
Start {0} A {0..2} B {3..7} C {3..5} D {7..9} End {9..11}
```
Constraint Logic Programming

• Takes advantage of the declarativeness of logic programming
  – Reduced development time, efficiency, clarity

• CLP(FD)
  – Finite domains
  – Useful for modeling discrete optimization/decision problems: scheduling, planning, packing, timetabling, ...
Constraints (SICStus)

- Arithmetic constraints
  - #=, #\=, #>, #<, #>=, #<=
  - sum/3, scalar_product/4/5, minimum/2, maximum/2

- Membership constraints
  - domain/3, in, in_set

- Propositional constraints
  - #\, #\/, #\^, #\, #>=, #<=, #<=>

- Combinatorial constraints
  - global_cardinality/2/3, element/3, table/2/3, case/3/4, all_different/1/2, all_distinct/1/2, assignment/2/3, sorting/3, circuit/1/2, cumulative/1/2, cumulatives/2/3, ...
CLP – example (SICStus)

- $\|C_{max}$, resources limited to 13

<table>
<thead>
<tr>
<th>Task</th>
<th>Duration</th>
<th>Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>T2</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>T3</td>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td>T4</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>T5</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>T6</td>
<td>18</td>
<td>1</td>
</tr>
<tr>
<td>T7</td>
<td>4</td>
<td>11</td>
</tr>
</tbody>
</table>

variables and domains

constraints

search

schedule(Ss, End) :-
Ss = [S1,S2,S3,S4,S5,S6,S7],
Es = [E1,E2,E3,E4,E5,E6,E7],
Tasks = [task(S1, 16, E1, 2, 1),
        task(S2, 6, E2, 9, 2),
        task(S3, 13, E3, 3, 3),
        task(S4, 7, E4, 7, 4),
        task(S5, 5, E5, 10, 5),
        task(S6, 18, E6, 1, 6),
        task(S7, 4, E7, 11, 7)],
domain(Ss, 1, 30),
domain(Es, 1, 50),
domain([End], 1, 50),
maximum(End, Es),
cumulative(Tasks, [limit(13)]),
append(Ss, [End], Vars),
labeling([minimize(End)], Vars).
CLP – example (SICStus)

- $\|C_{max}\$

<table>
<thead>
<tr>
<th>Task</th>
<th>Duration</th>
<th>Resources</th>
<th>Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>16</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>T2</td>
<td>6</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>T3</td>
<td>13</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>T4</td>
<td>7</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>T5</td>
<td>5</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>T6</td>
<td>18</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>T7</td>
<td>4</td>
<td>11</td>
<td>1</td>
</tr>
</tbody>
</table>

schedule(Ss, End) :-

Ss = [S1,S2,S3,S4,S5,S6,S7],
Es = [E1,E2,E3,E4,E5,E6,E7],
Tasks = [task(S1, 16, E1, 2, 1),
         task(S2, 6, E2, 9, 2),
         task(S3, 13, E3, 3, 1),
         task(S4, 7, E4, 7, 2),
         task(S5, 5, E5, 10, 1),
         task(S6, 18, E6, 1, 2),
         task(S7, 4, E7, 11, 1)],
Machines = [machine(1,12), machine(2,10)],
domain(Ss, 1, 30),
domain(Es, 1, 50),
domain([End], 1, 50),
maximum(End, Es),
cumulatives(Tasks, Machines, [bound(upper)]),
append(Ss, [End], Vars),
labeling([minimize(End)], Vars).