Scheduling With Constraint Programming

Pascal Van Hentenryck
Brown University
Outline

- Motivation
- Jobshop Scheduling
- Asymmetric TSP with Time Windows
- Cumulative Scheduling

Pascal Van Hentenryck
Scheduling

- Very successful application area for CP
- Minimize project duration subject
  - Precedence constraints
  - Disjunctive constraints: no two tasks scheduled on the same machine cannot overlap in time

Pascal Van Hentenryck
Scheduling

Pascal Van Hentenryck
Vertical Extensions

- Model-based computing
- Based on concepts of
  - activities,
  - resources
  - precedence constraints
  - ...
- Encapsulates variables and global constraints
- Support search procedures

Pascal Van Hentenryck
Outline

- Motivation
- Jobshop Scheduling
- Cumulative Scheduling
Jobshop Scheduling

- Problem formulation
  - a set of tasks is given, e.g., 1..100
  - each task $t$ has a duration $d(t)$
  - each task $t$ has a machine $m(t)$ to execute
  - a set of precedence constraints $(b,a)$
    - $a$ can only start when $b$ is completed

- Goal
  - minimize the project completion time
Jobshop Scheduling

- A machine handle activities in sequence
  - Find a activity *ordering* on each machine

Pascal Van Hentenryck
Jobshop Scheduling

- Solution = a directed acyclic precedence graph

Pascal Van Hentenryck
Jobshop Scheduling

- Why is an ordering for each machine sufficient?
  - only precedence constraints left?
  - easy to solve in polynomial time
    - topological sorting (PERT)
    - transitive closure (Floyd-Warshall)
Jobshop Scheduling Model

range Jobs = 0..nbJobs-1; range Tasks = 0..nbTasks-1; range Machines = Tasks;
range Activities = 0..nbActivities+1;
int duration[Jobs,Tasks];
int machine[Jobs,Tasks];
int horizon = sum(j in Jobs,t in Tasks) duration[j,t];

Scheduler<CP> cp(horizon);
Activity<CP> a[j in Jobs,t in Tasks](cp,duration[j,t]);
Activity<CP> makespan(cp,0);

UnaryResource<CP> r[Machines](cp);

Decision Variables

Resources

Pascal Van Hentenryck
range Jobs = 0..nbJobs-1; range Tasks = 0..nbTasks-1; range Machines = Tasks;
range Activities = 0..nbActivities+1;
int duration[Jobs, Tasks];
int machine[Jobs, Tasks];
int horizon = sum(j in Jobs, t in Tasks) duration[j, t];

Scheduler<CP> cp(horizon);
Activity<CP> a[j in Jobs, t in Tasks](cp, duration[j, t]);
Activity<CP> makespan(cp, 0);

UnaryResource<CP> r[Machines](cp);
Jobshop Scheduling Model

minimize<cp>
    makespan.end()
subject to { 

    forall(j in Jobs, t in Tasks: t != Tasks.getUp())
    a[j,t].precedes(a[j,t+1]);
    forall(j in Jobs)
    a[j,Tasks.getUp()].precedes(makespan);

    forall(j in Jobs, t in Tasks)
    a[j,t].requires(r[machine[j,t]]); 
}
Jobshop Scheduling Modeling

\[ t[1] \text{ precedes } t[2]; \]
\[ \ldots \]
\[ t[99] \text{ precedes } t[100] \]

\[ \text{disjunctive}(t[1], \ldots, t[10]) \]
\[ \text{disjunctive}(t[11], \ldots, t[20]) \]
\[ \ldots \]
\[ \text{disjunctive}(t[91], \ldots, t[100]) \]

end\[t[1]\] \leq \text{start}[t[2]]
Computational Model

Constraint Store

Domain store

Pascal Van Hentenryck
Disjunctive Scheduling

- A combinatorial constraint with each machine
  - all the tasks executing on the machine
- Two tasks
  - determining feasibility
  - reducing the domain of the variables
    - bound reduction
Disjunctive Constraint

- One-Machine Feasibility
Disjunctive Constraint

- **Pruning: edge finder rules**
  - select a set $T$ of tasks and a task $i$ such that $i \notin T$
  - determine whether $i$ must start after all tasks of $T$
    - update its starting date: $E(T) + D(T)$
  - determine whether $i$ must finish before all tasks of $T$
    - update its ending date: $L(T) - D(T)$
- **The edge-finder rules can be enforced in strongly polynomial time**

Pascal Van Hentenryck
Disjunctive Constraint

- Pruning: Can $A_1$ start first?
Disjunctive Constraint

- Pruning: Can $A_1$ start first?

Pascal Van Hentenryck
Disjunctive Constraint

- Pruning: $A_1$ must start after $A_2$ and $A_3$
Branching

- How to branch?
  - choose a machine
  - choose a task to rank first on the machine
  - on backtracking, rank not first

- Which machine?
  - tightest machine
  - e.g., the least slack

- Which task?
  - a task that can be scheduled first
  - a task which is as tight as possible

Pascal Van Hentenryck
Search Strategies

- Branching
  - specifies the tree to explore
  - does not specify how to explore it

- Search strategies
  - specifies how to explore the search tree
  - default: depth-first search
  - others:?
Limited Discrepancy search

- Assume that we have a good heuristic which make few mistakes
- Follow the heuristic (left branch)
- Trust the heuristic less and less
  - assume that it makes 1 mistake
  - then assume that it makes 2 mistakes
- The search tree is explored in waves

Pascal Van Hentenryck
Limited Discrepancy Search

Pascal Van Hentenryck
Jobshop Scheduling Search

Pascal Van Hentenryck

```plaintext
cp.setSearchController(BDSController(cp,3));
minimize<cp>
    makespan.end()
subject to {
    ...
} using {
    forall(m in Machines)  by (r[m].localSlack())
    r[m].rank();
    cp.post(makespan.end() == makespan.end().getMin());
}
```

Search

exploration

nondeterministic

ordering
The input: we are given
- a set of locations to visit
- a service time for each location
- a time window when to serve a location
- the (asymmetric) travel distance between locations

The goal: find a hamiltonian path
- satisfying the time windows
- minimizing the travel distance

Asymmetric TSP with Time Windows

Pascal Van Hentenryck
Asymmetric TSP with Time Windows
Asymmetric TSP with Time Windows

Pascal Van Hentenryck
The CP Model

Scheduler<CP> cp(0,horizon);
Activity<CP> act[i in Activities](cp,service[i],i);
UnaryResource<CP> vehicle(cp,transitionTimes);
minimize<cp>
    vehicle.getSumTransitionTimes()
subject to
    forall(i in Activities) {
        cp.post(act[i].start() >= ws[i]);
        cp.post(act[i].start() <= we[i]);
        act[i].requires(vehicle);
    }
using {
    vehicle.sequenceForward();
    forall(i in Activities) label(act[i].start());
}
Dual Modelling

- Scheduling Model
  - reasons about the start dates, time windows
  - disjunctive constraint
- Routing Model
  - reasons about the successor/predecessor
  - Hamiltonian path constraints
  - assignment constraint for the transition times
- Communication constraints

Pascal Van Hentenryck
The Assignment Constraint

\[ \text{succ}_0 \rightarrow t_{12} \rightarrow 1 \]
\[ \text{succ}_1 \rightarrow t_{23} \rightarrow 2 \]
\[ \text{succ}_2 \rightarrow 3 \]
\[ \text{succ}_3 \rightarrow 4 \]
\[ \text{succ}_4 \rightarrow t_{54} \rightarrow 5 \]

\[ \leq B \]

Pascal Van Hentenryck
Constraint Solving

Constraint Store

Domain store

Assignment

disjunctive

start

successors

path

Pascal Van Hentenryck
Large Neighborhood Search

- Combination of local search and CP
  - start with a feasible solution (CP)
  - relax part of the best solution found so far
    - select a subpath in the solution
    - select a random set of variables
  - optimize the resulting problem using CP
    - it is a very constrained combinatorial space
  - iterate the last two steps

Pascal Van Hentenryck
Asymmetric TSP with Time Windows

Pascal Van Hentenryck
Asymmetric TSP with Time Windows

Pascal Van Hentenryck
### Experimental Results

<table>
<thead>
<tr>
<th></th>
<th>BK</th>
<th>300s</th>
<th>600s</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>386</td>
<td>386</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>492</td>
<td>487</td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>488</td>
<td>484</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>414</td>
<td>414</td>
<td></td>
</tr>
<tr>
<td>67</td>
<td>1048</td>
<td>1048</td>
<td></td>
</tr>
<tr>
<td>86</td>
<td>1052</td>
<td>1051</td>
<td></td>
</tr>
<tr>
<td>92</td>
<td>1111</td>
<td>1093</td>
<td></td>
</tr>
<tr>
<td>125</td>
<td>1410</td>
<td>1409</td>
<td></td>
</tr>
<tr>
<td>132</td>
<td>1400</td>
<td>1382</td>
<td></td>
</tr>
<tr>
<td>152</td>
<td>1792</td>
<td>1783</td>
<td></td>
</tr>
<tr>
<td>172</td>
<td>1897</td>
<td>1870</td>
<td>1799</td>
</tr>
<tr>
<td>193</td>
<td>2452</td>
<td>2433</td>
<td></td>
</tr>
<tr>
<td>201</td>
<td>2296</td>
<td>2234</td>
<td></td>
</tr>
<tr>
<td>233</td>
<td>2786</td>
<td>2683</td>
<td></td>
</tr>
</tbody>
</table>

- Ascheuer, Fischetti, Grötschel, 2001
- Industrial application in robotic
- 5 hours of CPU Time

Pascal Van Hentenryck
Cumulative Scheduling

Problem formulation
- A set of tasks is given, e.g., 1..100
- Each task $t$ has a duration $d(t)$
- Each task $t$ has a machine $m(t)$ to execute
- A set of precedence constraints $(b,a)$
  - $a$ can only start when $b$ is completed
- Each machine has a capacity, i.e., a maximum number of activities that can be executed simultaneously

Goal
- Minimize the project completion time
int capacity = 8; int nbTasks = 34;
range Tasks = 1..nbTasks;
int duration[Tasks] = …
int totalDuration = sum(t in Tasks) duration[t];
int demand[Tasks] = …
tuple P { int before; int after; } 
set{P} setOfPrecedences = …

Scheduler<CP> cp(totalDuration);
Activity<CP> a[t in Tasks](cp,duration[t]);
DiscreteResource<CP> d(cp,capacity);
Activity<CP> makespan(cp,0);
minimize<cp>
  makespan.end()
subject to {
  forall(t in Tasks)
    a[t].precedes(makespan);
  forall(p in setOfPrecedences)
    a[p.before].precedes(a[p.after]);
  forall(t in Tasks)
    a[t].requires(d,demand[t]);
} using {
  setTimes(a);
}
Cumulative Scheduling Search

- **Basic ideas**
  - Not sufficient to order the tasks
  - Must choose starting times for the tasks

- **Value/Variable Search**
  - Choose the earliest time at which an activity can be scheduled
  - Nondeterministically choose an activity to start there
  - May use dominance rules to decide which activities to consider
Large Neighborhood Search

- Key idea: partial order schedule

  - First step
    - relax a number of activities
    - remove them from the schedule

  - Second step
    - use the resources to impose new precedence constraints
    - do not fix variables to their values
Cumulative Scheduling

```
minimize<cp>
    makespan.end()
subject to { ...}
using setTimes(a);
onRestart {
    Solution s = cp.getSolution();
    if (s!=null) {
        set{Activity<CP>} R();
        forall(a in Activities)
            if (distr.get() <= Pr)
                R.insert(a);
        cp.relaxPOS(s,R);
    }
}
```
The Trolley Problem

Pascal Van Hentenryck
The Trolley Problem

- Model the trolley as a state resource
  - the state represents the location of the trolley
  - all activities use the trolley except the actual processing on the machine
The Trolley Problem (I)

```pascal
enum Jobs = {j1,j2,j3,j4,j5,j6};
enum Tasks =
  {loadA,unload1,process1,load1,unload2,process2,load2,unloadS};
enum Locations = {m1,m2,m3,areaA,areaS};

Locations location[Jobs,Tasks];
int duration[Jobs,Tasks];

Scheduler<CP> cp(horizon);
StateResource<CP> trolley(cp,Locations);
UnaryResource<CP> machine[Locations](cp);
Activity<CP> act[j in Jobs,t in Tasks](cp,duration[j,t],location[j,t]);
Activity<CP> makespan(cp,0);```

Pascal Van Hentenryck
The Trolley Problem (I)

```pascal
minimize<cp> makespan.end()
subject to {
    forall(j in Jobs, t1 in Tasks, t2 in Tasks: t1 < t2)
        act[j,t1].precedes(act[j,t2]);
    forall(j in Jobs) {
        act[j,process1].requires(machine[job[j].machine1]);
        act[j,process2].requires(machine[job[j].machine2]);
    }
    forall(j in Jobs, t in Tasks: t != process1 && t != process2)
        act[j,t].requires(trolley,location[j,t]);
    forall(j in Jobs)
        act[j,unloadS].precedes(makespan);
}
using {
    setTimes(all(j in Jobs, t in Tasks) act[j,t]);
    label(makespan.start());
}
```

Pascal Van Hentenryck
The Trolley Problem (II)

- Adding transition times between the machines

```plaintext
int tt[Locations,Locations] = [
    [ 0, 50, 60, 50, 90 ],
    [ 50, 0, 60, 90, 50 ],
    [ 60, 60, 0, 80, 80 ],
    [ 50, 90, 80, 0, 120 ],
    [ 90, 50, 80, 120, 0 ]
];
```
The Trolley Problem (II)

```c
enum Jobs = {j1,j2,j3,j4,j5,j6};
enum Tasks =
    {loadA, unload1, process1, load1, unload2, process2, load2, unloadS};
enum Locations = {m1,m2,m3,areaA,areaS};
Locations location[Jobs,Tasks];
int duration[Jobs,Tasks];

Scheduler<CP> cp(horizon);
StateResource<CP> trolley(cp, Locations, tt);
UnaryResource<CP> machine[Locations](cp);
Activity<CP> act[j in Jobs, t in Jobs](cp, duration[j, t], location[j, t]);
Activity<CP> makespan(cp, 0);

Pascal Van Hentenryck
```
The Trolley Problem (III)

- Adding a capacity on the Trolley
  - Use activities to track when a job uses the trolley
  - Use a discrete resource

- The trolley is modeled by two resources
  - a state resource to denote its location
  - a discrete resource to denote its load

- Synchronization constraints

Pascal Van Hentenryck
The Trolley Problem (III)

- Resources + Activities

```cpp
eenum TrolleyTasks = {onTrolleyA1, onTrolley12, onTrolley2S};
Scheduler<CP> cp(horizon);
UnaryResource<CP> machine[Locations](cp);
StateResource<CP> trolley(cp, Locations);
DiscreteResource<CP> trolleyCapacity(cp, 3);
Activity<CP> act[j in Jobs, t in Jobs]
    (cp, duration[j, t], location[j, t]);
Activity<CP> tact[j in Jobs, t in TrolleyTasks]
    (cp, 2*loadDuration..horizon);
Activity<CP> makespan(cp, 0);
```

Pascal Van Hentenryck
The Trolley Problem (III)

Specifying the trolley activities

forall(j in Jobs) {
    cp.post(tact[j,onTrolleyA1].start() == act[j,loadA].start());
    cp.post(tact[j,onTrolleyA1].end() == act[j,unload1].end());
    cp.post(tact[j,onTrolley12].start() == act[j,load1].start());
    cp.post(tact[j,onTrolley12].end() == act[j,unload2].end());
    cp.post(tact[j,onTrolley2S].start() == act[j,load2].start());
    cp.post(tact[j,onTrolley2S].end() == act[j,unloadS].end());
}
forall(j in Jobs, t in TrolleyTasks)
    tact[j,t].requires(trolleyCapacity,1);