Modeling Time (Scheduling)

Peter Stuckey

Overview

- Scheduling problems
  - are one of the most common uses of CP in the real world
- Basic Scheduling
  - only precedence constraints
- Job Shop Scheduling
  - disjunctive global constraint
- Resource Constraint Project Scheduling
  - cumulative global constraint
- Sequence Dependent Setup Times
  - modeling order

Scheduling

- In discrete optimization
  - time is modeled by integers (not continuous)
- Time variables
  - tend to have VERY large ranges
    - e.g. start times on the minute for a 7 day schedule
  - typically only care about
    - earliest time, or
    - latest time
  - when reasoning (not about all possible times)

Overview

- Basic Scheduling
  - tasks and precedences
- Disjunctive scheduling
  - at most one task at a time
- Cumulative scheduling
  - identical resources with limited capacity
- Sequence dependent scheduling
  - modeling order between tasks

EOF

Basic Scheduling

Peter Stuckey

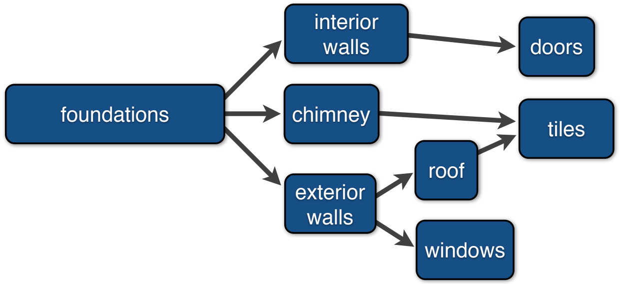
Basic Scheduling

- Scheduling is an important class of discrete optimisation problems
- Basic scheduling involves:
  - tasks with durations
  - precedences between tasks
    - one task must complete before another starts
- The aim is to schedule the tasks
  - usually to minimize the latest end time

Project Scheduling

- Building a house involves a number of tasks, and precedences where one task may not be started until another is completed. Each task has a duration. We need to determine the schedule that minimises the total time to build the house
  - Task (duration): foundations (7), interior walls (4), exterior walls (3), chimney (3), roof (2), doors (2), tiles (3), windows (3).
  - walls and chimney need foundations finished, roof and windows after exterior walls, doors after interior walls, tiles after chimney and roof

Project Scheduling



- Length indicates durations
- Arcs indicate precedences

Project Scheduling

- Data

```
int: n = 8; % no of tasks max
set of int: TASK = 1..n;
int: f = 1; int: iw =2; int: ew = 3;
int: c = 4; int: r = 5; int: d = 6;
int: t = 7; int: w = 8;
array[TASK] of int: duration =
  [7,4,3,3,2,2,3,3];
int: p = 8; % number of precedences
set of int: PREC = 1..p;
array[PREC] of TASK: pre =
  [f,f,f,ew,ew,iw,c,r];
array[PREC] of TASK: post =
  [iw,ew,c,r,w,d,t,t];
```


Project Scheduling

- Decisions

```
int: s = sum(duration);
array[TASK] of var 0..s: start;
```

- Constraints

```
forall(i in PREC)
  (start[pre[i]] + duration[pre[i]]
   <= start[post[i]]);
```

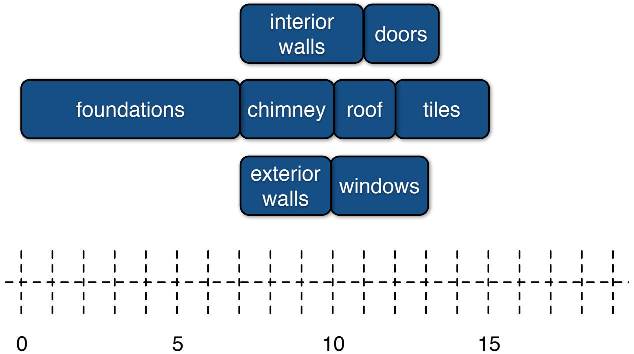
- Objective

```
var 0..s: makespan;
forall(t in TASK)
      (start[t] + duration[t] <= makespan);
solve minimize makespan;
```


Project Scheduling

- Constraints generated
  - s[f] + 7 <= s[iw]
  - s[f] + 7 <= s[ew]
  - s[f] + 7 <= s[c]
  - s[ew] + 3 <= s[r]
  - s[ew] + 3 <= s[w]
  - s[iw] + 4 <= s[d]
  - s[c] + 4 <= s[t]
  - s[r] + 2 <= s[t]
  - s[d] + 2 <= makespan
  - s[t] + 3 <= makespan
  - s[w] + 3 <= makespan

Project Scheduling Solution




```
makespan     f iw ew  c   r    d    t    w
15        = [0, 7, 7, 7, 10, 11, 12, 10]
```


Difference logic constraints

- Difference logic constraints take the form
  - $x + d \leq y$ d is constant
- Note $x + d = y \leftrightarrow x + d \leq y \wedge y + (-d) \leq x$
- A problem that is representable as a conjunction of difference logic constraints can be solved very rapidly
  - longest/shortest path problem
- But adding extra constraints means this advantage disappears
  - e.g. at most two tasks can run simultaneously

Overview

- Basic scheduling problems
  - tasks with precedences

  are a common part of many complex discrete optimisation problems
- The constraints needed to model this are a simple form of linear constraints
  - difference logic constraints
- Problems involving only these constraints can be solved very efficiently

EOF

Disjunctive Scheduling

Peter Stuckey

Scheduling Concepts (so far)

- Tasks
  - start time, duration, and end time
  - other attributes

```
array[TASK] of var int: s;
array[TASK] of var int: d;
array[TASK] of var int: e;
forall(t in TASK)(e[t] = s[t] + d[t]);
```

  - may omit end times, particular when d is fixed
- Precedences
  - one task can only start after another finishes
  - task t1 precedes t2

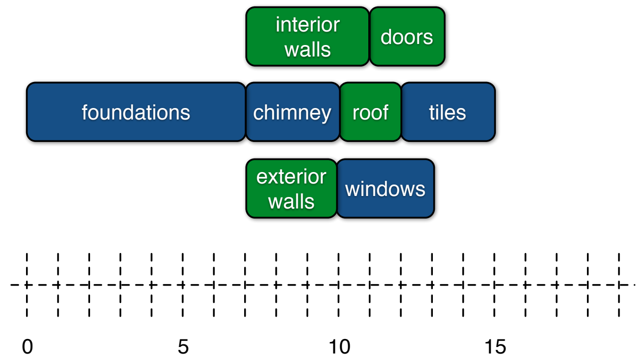
```
e[t1] <= s[t2]     (s[t1] + d[t1] <= s[t2])
```


Nonoverlap

- Consider the ProjectScheduling problem where we only have one carpenter who can undertake the walls and roof work
  - these tasks cannot overlap in duration

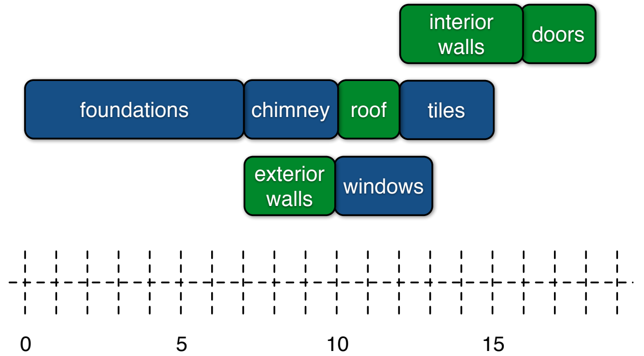
```
predicate nonoverlap(var int:s1, var int:d1,
                     var int:s2, var int:d2)=
          s1 + d1 <= s2 \/ s2 + d2 <= s1;

set of TASK: CARPENTRY = { iw, ew, r, d };
forall(t1, t2 in CARPENTRY where t1 < t2)
      (nonoverlap(start[t1],duration[t1],
                  start[t2],duration[t2]));
```


ProjectScheduling with Carpentry




```
makespan     f  iw ew  c   r   d   t   w
15        = [0,  7, 7, 7, 10, 11, 12, 10]
```

ProjectScheduling with Carpentry




```
makespan     f   iw ew  c   r    d   t    w
18         = [0, 12, 7, 7, 10, 16, 12, 10]
```


Resources

- Critical to most scheduling problems are limited resources
  - unary resource (at most one task at a time)
  - cumulative resource (a limit on the amount of resource used at any time)

Unary Resources

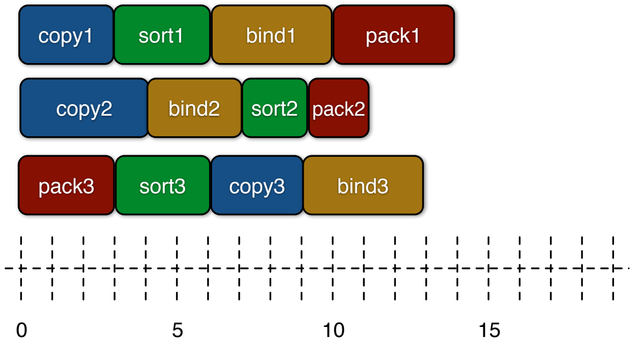
- The ProjectScheduling problem with non overlap involved a unary resource
  - number of tasks executing at one time
- Unary resources are common
  - machine
  - nurse, doctor, worker in a roster
  - track segment (one train at a time)
  - …

JobShop Scheduling

- JobShop: Given *n* jobs each made up of a sequence of *m* tasks, one each on each of *m* machines. Schedule the tasks to finish as early as possible where each machine can only run one task at a time
- Data

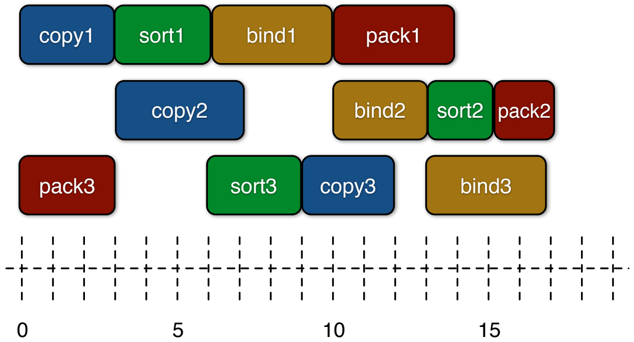
```
int: n;
set of int: JOB = 1..n;
int: m;
set of int: MACH = 1..m;
set of int: TASK = 1..m;
array[JOB,TASK] of int:  du; % length of task
array[JOB,TASK] of MACH: mc; % which machine
```

JobShop Example



- Rows indicate tasks in a Job
- Colors indicate different machine

JobShop Solution



- Tasks pushed later so no two of the same color are simultaneous

JobShop Variables + Constraints

- Variables

```
int: maxt = sum(j in JOB, t in TASK)(d[j,t]);
array[JOB,TASK] of var 0..maxt: s;
```

- Precedence Constraints

```
forall(j in JOB, t in 1..m-1)
      (s[j,t] + d[j,t] <= s[j,t+1]);
```

- Machine Constraints

```
forall(j1, j2 in JOB, t1, t2 in TASK where
       j1 < j2 /\ mc[j1,t1] = mc[j2,t2])
      (nonoverlap(s[j1,t1],d[j1,t1],
                  s[j2,t2],d[j2,t2]));
```


JobShop Objective

- Minimize the makespan (when the last job finishes)

```
var 0..maxt: makespan;
forall(j in JOB)
      (s[j,m] + d[j,m] <= makespan);
solve minimize makespan;
```

disjunctive

- Nonoverlap only considers two tasks at a time
  - a unary resource requires non overlap for all pairs of tasks that use it
- Disjunctive constraint
  - disjunctive(<start time array>,<duration array>)
    - ensure no two tasks in the array overlap in execution

```
predicate disjunctive(array[int] of var int:s,
                      array[int] of var int:d)=
   forall(i1,i2 in index_set(s) where i1 < i2)
         (nonoverlap(s[i1],d[i1],s[i2],d[i2]));
```


JobShop Revisited

- Replace nonoverlap with disjunctive
- We need to build the start times and durations for all jobs on a machine
  - perfect for a local variable

```
include "disjunctive.mzn";
forall(ma in MACH)
      ( let { array[int] of var int: ss =
             [ s[j,t] | j in JOB, t in TASK
                        where mc[j,t] = ma ];
             array[int] of int:     dd =
             [ d[j,t] | j in JOB, t in TASK
                   where mc[j,t] = ma ]; } in
        disjunctive(ss,dd));
```


JobShop Scheduling

- Is remarkably hard
- For some 10x10 instances from 1963
  - we did not know the optimal solution until 1989!
- There are a lot of approximation algorithms
- The online version is also heavily studied
  - where we have to schedule a job, given an existing schedule, then schedule the next job

A note about disjunctive

- In the current MiniZinc library
  - disjunctive is not included
- You can use cumulative to define it

```
include "cumulative.mzn";
predicate disjunctive(array[int] of var int:s,
                      array[int] of int: d) =
    cumulative(s,d,[1| i in index_set(s)], 1);
```

Overview

- Disjunctive scheduling
  - allows us to express that two tasks do not overlap in execution
  - without specifying the relative order
- disjunctive global constraint
  - capture a set of tasks on a unary resource
- Many classic scheduling problems
  - job shop scheduling
  - open shop scheduling

EOF

Cumulative Scheduling

Peter Stuckey

Resources

- Unary resources are unique
- Often we have multiple identical copies of a resource
  - bulldozers
  - workers (of equal capability)
  - operating theaters
  - airplane gates
- How do we model multiple identical resources?
  - assume task `t` uses `res[t]`
  - assume a limit `L` of resource at all times

Modeling Resources: Time Decomposition

- The use of the resource at each time `i` is less that the limit `L`

```
forall(i in TIME)
       (sum(t in TASK)
           (bool2int(s[t]<=i /\ s[t]+d[t]>i)
            * res[t])
        <= L);
```

- Note the expression
  - `s[t]<=i /\ s[t]+d[t]>i`
  - represents whether task `t` runs at time `i`
- Problem: size is card(TASK)*card(TIME)
  - many time periods TIME

Modeling Resources: Task Decomposition

- Note we can only only overload a resource when a task starts (otherwise no increase)
- Alternate model: only check start times

```
forall(t2 in TASK)
       (sum(t in TASK)
           (bool2int(s[t]<=s[t2]
                      /\ s[t]+d[t]> s[t2])
            * res[t])
        <= L);
```

- Can we do improve this?

Modeling Resources: Task Decomposition

- Better model: we know t2 runs at time s[t2]

```
forall(t2 in TASK)
       (sum(t in TASK where t != t2)
           (bool2int(s[t]<=s[t2]
                      /\ s[t]+d[t]> s[t2])
            * res[t])
        + res[t2] <= L);
```

- Advantage: much smaller than time decomposition card(TASK)^2
- Problem: not as much information to the solver

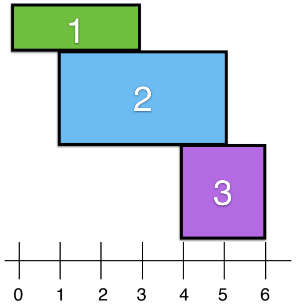
Cumulative

- The cumulative global constraint captures exactly a resource constraint
  - cumulative(<start time array>,<duration array>,
  - <resource usage array>, <limit>)
    - ensure no more than the limit of the resource is used at any time during the execution of tasks

```
predicate cumulative(array[int] of var int:s,
                     array[int] of var int:d,
                     array[int] of var int:r,
                     var int: L);
```

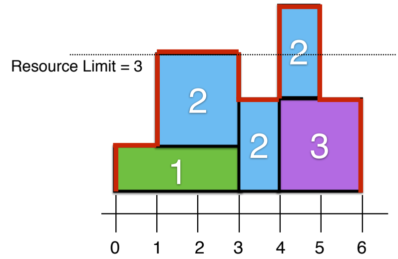
Visualizing Cumulative

- A task t is a box of length d[t] and height r[t] starting at time s[t]
  - e.g. cumulative([0,1,4],[3,4,2],[1,2,2],3)



Visualizing Cumulative

- A task t is a box of length d[t] and height r[t] starting at time s[t]
  - e.g. cumulative([0,1,4],[3,4,2],[1,2,2],3)
- They are not really boxes
- Timetable (red skyline) shows the usage

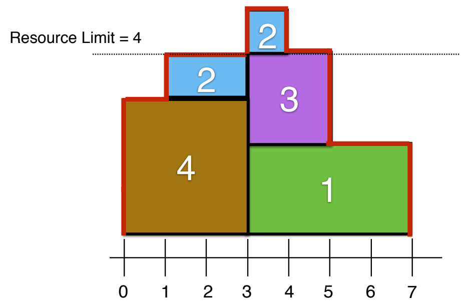


Cumulative Example

- Does the constraint below hold
  - cumulative([3,1,3,0],[4,3,2,3],[2,1,2,3],4)
- Given the cumulative constraint below does it have a solution
  - start time possibilities are given as ranges
  - cumulative([0..3,0..3,2..3,0..4],[4,3,2,3],[2,1,2,3],4)

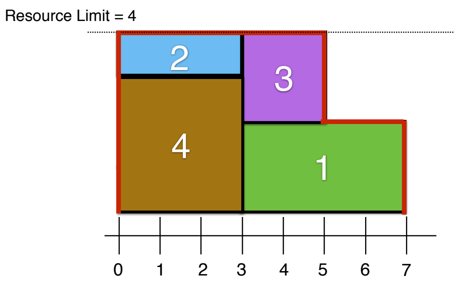
Visualizing Cumulative

- Does the constraint below hold
  - cumulative([3,1,3,0],[4,3,2,3],[2,1,2,3],4)



Visualizing Cumulative

- Given the cumulative constraint below does it have a solution
  - start time possibilities are given as ranges
  - cumulative([0..3,0..3,2..3,0..4],[4,3,2,3],[2,1,2,3],4)



Cumulative Propagators

- There is a lot of research in how to propagate cumulative constraints
  - timetable propagation
    - equivalent to the time decomposition
    - but faster than the task decomposition
  - edge finding
    - reasoning about time intervals rather than single times
  - energy based reasoning
    - more inference than edge finding, but slower
  - TTEF time table edge finding
    - a combination of timetable with some energy based reasoning
    - state of the art

Resource Constrained Project Scheduling (RCPSP)

- Given tasks $t \in TASK$
- Given precedences p ∈ PREC
  - pred[p] precedes succ[p]
- Assume resources $r \in RESOURCES$
- Each task t needs $req[r,t]$ resources during its execution
- We have a limit $L[r]$ for each resource
- Find the shortest schedule to run every task!

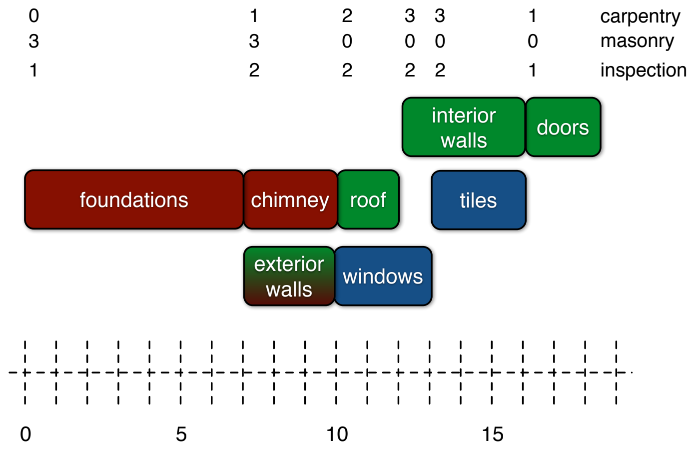
- Possibly the most studied scheduling problem

RCPSP House Building

- A more detailed version of the house building scheduling problem
- Three resources
  - carpentry
  - masonry
  - inspection

resource	f	iw	ew	c	r	d	t	w	limit
carpentry	0	3	1	0	2	1	0	0	3
masonry	3	0	2	1	0	0	0	0	3
inspection	1	1	1	1	1	1	1	1	2

RCPSP House Building Solution




```
makespan     f  iw ew  c   r   d   t   w
18        = [0, 12, 7, 7, 10, 16, 13, 10]
```


RCPSP Data

- Data

```
int: n;                       % number of tasks
set of int: TASK = 1..n;
array[TASK] of int: d;        % duration
int: m;                       % no. of resources
set of int: RESOURCE = 1..m;
array[RESOURCE] of int: L;    % resource limit
array[RESOURCE,TASK] of int: res; % usage
int: l;                       % no. of precedences
set of int: PREC = 1..l;
array[PREC,1..2] of TASK: pre; i
      % predecessor/successor pairs
int: maxt; % maximum time
set of int: TIME = 0..maxt;
```


RCPSP House Data

- Example Data

```
n = 8;
d = [7,4,3,3,2,2,3,3];
m = 3;
L = [3,3,2];
res = [| 0,3,1,0,2,1,0,0
       | 3,0,2,1,0,0,0,0
       | 1,1,1,1,1,1,1,1 |];
l = 8;
pre = [| 1, 2
       | 1, 3
       | 1, 4
       | 3, 5
       | 3, 8
       | 2, 6
       | 4, 7
       | 5, 7 |];
```


RCPSP Model

- Decisions

```
array[TASK] of var TIME: s; % start time
```

- Constraints

```
forall(p in PREC)
      (s[pre[p,1]]+d[pre[p,1]] <= s[pre[p,2]]);

forall(r in RESOURCE)
   (cumulative(s,d,[res[r,t]|t in TASK],L[r]));
```

- Objective

```
solve minimize max(t in TASK)(s[t] + d[t]);
```

Overview

- Renewable capacitated resources
  - a resource capacity available over the schedule
- Time decomposition:
  - check resource usage at each time
- Task decomposition
  - check resource usage as each task starts
- `cumulative` global constraint

- RCPSP: a core scheduling problem

EOF

Sequence Dependent Scheduling

Peter Stuckey

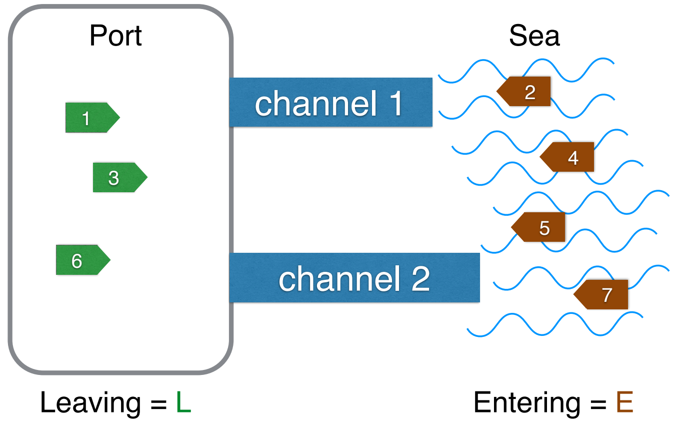
Sequence dependence

- Particularly for unary resources
- The schedule may depend on which tasks precedes another on that resource
- Examples
  - smelting: machinery must cool to perform “cold” task after “hot task”
  - embroidery: colors of thread may need changing between tasks
  - single channel: ships traveling in different directions need to wait until channel is clear
- Effectively the start time of the next task is delayed depending on the previous task

DoubleChannel

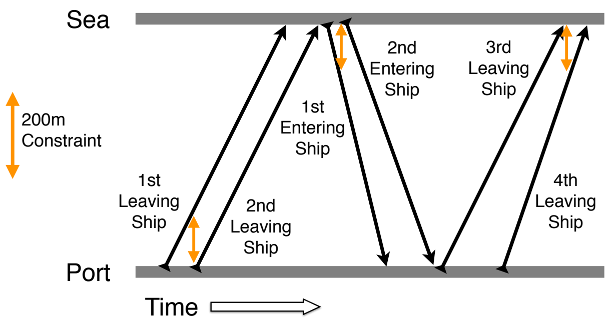
- Given a set of *E* ships entering a port, and *L* ships leaving a port, we need to choose which of two channels they use, and when they enter/leave. Each ship has a desired enter/leave time. The aim is to minimize the total difference from the desired times
- The two channels are 800m and 600m long
- Two ships cannot be closer than 200m
- An entering ship must clear the channel before a leaving ship can leave, and vice versa

DoubleChannel Problem



DoubleChannel

- The channel is a complex unary resource
- The time distance between ships is dependent on the relative direction and relative speeds



DoubleChannel

- Data

```
int: nC = 2; % number of channels
array[1..nC] of int: len = [6,8];

int: nS; % number of ships
set of int: SHIP = 1..nS;
array[SHIP] of int: speed;    % 100m time
array[SHIP] of int: desired; % desired time
int: enter = 1; int: leave = 2;
array[SHIP] of enter..leave: dirn;

int: leeway = 2;  % leeway between 2 ships
int: maxt;        % maximum time
set of int: TIME = 0..maxt;
```

DoubleChannel

- Decisions
  - add nC dummy ships to be the last ship in each channel
  - so each ship will have a next ship in its channel

```
set of int: SHIPE = 1..nS+nC; % add dummies
int: dummy = 3;
array[SHIPE] of enter..dummy: kind
          = dirn ++ [ dummy | i in 1..nC];
array[SHIPE] of var TIME: start;
array[SHIPE] of var TIME: end;
array[SHIPE] of var CHANNEL: channel;

array[SHIPE] of var SHIPE: next; % next ship
```


DoubleChannel

- Constraints
  - dummy ships are last and in a fixed channel

```
forall(s in nS + 1 .. nS + nC)
      (start[s] = maxt /\ end[s] = maxt);
forall(s in nS + 1 .. nS + nC)
      (channel[s] = s - nS);
```

- Relationship between start and end

```
forall(s in SHIP)(end[s] = start[s] +
                  len[channel[s]]*speed[s]);
```

- The next ships are all different

```
alldifferent(next);
```

- Note that next of dummy ship is the first in channel

DoubleChannel

- Reasoning about the channel
  - once we know the next ship its reasonably simple
- If the next ship is in the opposite direction
  - it can only start once we end
- If the next ship is in the same direction
  - it must start after we travel 200m

- Is that enough?
  - NO, its not allowed to "catch up"
  - it must end after we travel 200m past our end

DoubleChannel

- Relationship between a ship and its next ship
  - the start and end time are constrained

```
forall(s in SHIP)
          % ships of opposite dirn
      (if kind[s] + kind[next[s]] = 3 then
          end[s] <= start[next[s]]
       else % same dirn
start[s]+speed[s]*leeway <= start[next[s]] /\
  end[s]+speed[s]*leeway <= end[next[s]]
       endif);
```

  - they are in the same channel

```
forall(s in SHIP)
      (channel[next[s]] = channel[s]);
```

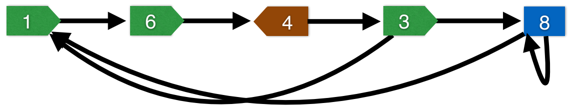
DoubleChannel

- Objective

```
solve minimize sum(s in SHIP)
               (abs(start[s] - desired[s]));
```


Subtleties of the Model

- Is the `alldifferent` constraint enough
  - for example this satisfies the constraint



- Yes since start times of the ships increase
  - except the dummy ship
- But for other similar problems we will need
  - the `circuit` global constraint

Order Dependent Setup Times

- If there is order dependent setup times or costs
  - model the next task explicitly
  - add constraints to ensure the setup time or cost is paid
- Examples are:
  - direction change in channel
  - mode change in machine shop scheduling
  - cool down time for smelting jobs at different temperatures

Overview

- Complex scheduling applications
  - have interdependencies between a task and the task that follows it
- We need to model for each task
  - which task is next, and
  - how that constrains the schedule

EOF

Packing

Peter Stuckey

Overview

- Packing problems
  - are another common uses of CP in the real world
  - come in lots of varieties
- 2D packing
  - `diffn` global constraint
  - redundant `cumulative` constraints
- Other packing constraints
  - `geost`
- Bin packing

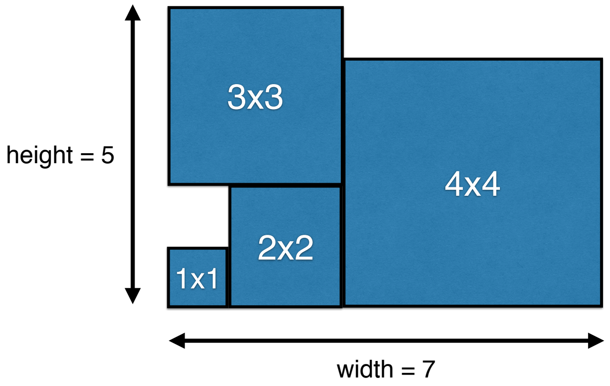
Square Packing

- Try to pack the squares 1x1, 2x2, .., *n*x*n*, into a rectangle of smallest area.

```
int: n; % number of squares
set of int: SQUARE = 1..n;
int: maxl = sum(i in SQUARE)(i);
int: mina = sum(i in SQUARE)(i*i);
var n..maxl: height;
var n..maxl: width;
var mina .. n*maxl: area = height*width;
array[SQUARE] of var 0..maxl: x;
array[SQUARE] of var 0..maxl: y;
```

- Note tight bounds on variables

Square Packing



area = 5x7 = 35

Square Packing

- Squares fit in the rectangle

```
forall(s in SQUARE)(x[s] + s <= width);
forall(s in SQUARE)(y[s] + s <= height);
```

- Squares do not overlap

```
forall(s1, s2 in SQUARE where s1 < s2)
      ( x[s1] + s1 <= x[s2] \/
        x[s2] + s2 <= x[s1] \/
        y[s1] + s1 <= y[s2] \/
        y[s2] + s2 <= y[s1] );
```

- Objective

```
solve minimize area;
```


Square Packing (Search)

- Search strategy should concentrate on smallest possible rectangles

```
solve :: int_search([area,height,width],
     input_order, indomain_min, complete)
minimize area;
```

- Packing, often useful to pack the biggest objects first

```
array[1..2*n] of var 0..maxl: vs =
 [ if i mod 2 = 1 then x[n+1 - i div 2]
   else y[n+1 - i div 2] endif
 | i in 2..2*n+1 ];
int_search(vs, input_order, indomain_min,
complete)
```


diffn

- The diffn global constraint captures exactly 2d non overlap (it should be called diff2)
  - diffn($[x_1, \dots, x_n]$, $[y_1, \dots, y_n]$,
  - $[dx_1, \dots, dx_n]$, $[dy_1, \dots, dy_n]$)
    - ensure no two objects at positions (x_i,y_i) with dimensions (dx_i,dy_i) overlap.

```
predicate diffn(array[int] of var int: x,
                array[int] of var int: y,
                array[int] of var int: dx,
                array[int] of var int: dy);
```

- Squares do not overlap

```
array[SQUARE] of int: size = [ i| i in SQUARE];
diffn(x,y,size,size);
```

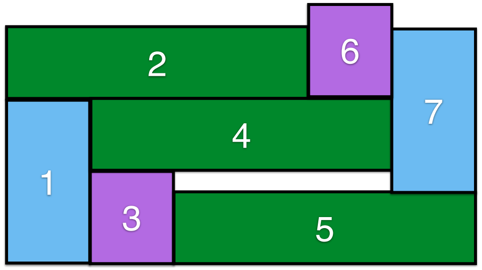
Packing and Cumulative

- If there is a packing
  - then the cumulative constraint must hold!
- We can add redundant cumulative constraints to packing problems
  - improves propagation (and hence solving)

```
cumulative(x,size,size,height);
cumulative(y,size,size,width);
```


Packing and Cumulative

- In general
  - cumulative constraints do not enforce packing
  - even when the the x positions are fixed



Overview

- Packing problems
  - are complex discrete optimization problems
- `diffn` encodes 2D non-overlap
- `disjunctive` encodes 1D non-overlap
- `cumulative` constraints are redundant for packing
  - but useful for improving solving

EOF

Carpet Cutting

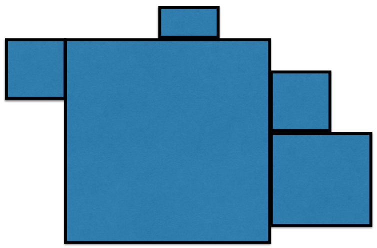
Peter Stuckey

Carpet Cutting

- Carpeting a new house
  - measure each room size and shape
  - cut the carpets out of a roll of carpet of fixed width
  - lay the carpet
- Using the least length of the roll means
  - less wastage
  - more profit for the carpeting company
- Complexities
  - carpet direction
  - stairs, filler carpets, weave constraints

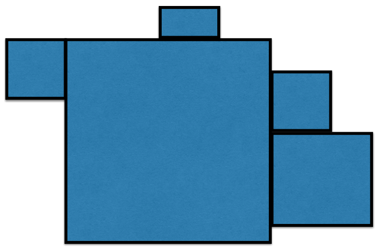
Room Shapes

- We assume rooms are rectilinear
- Carpets are made up of rectangles
- Example
  - a complex room shape of 5 rectangles



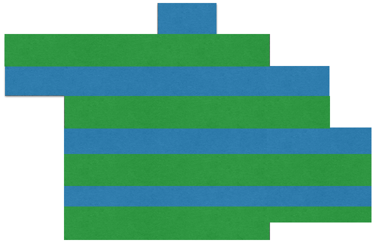
Carpet Orientation

- On a uniform unpatterned carpet we can cut a room in 4 different ways



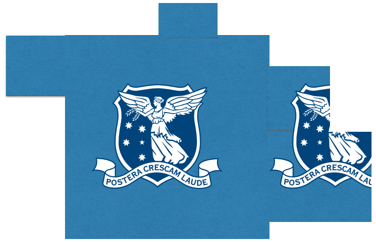
Carpet Orientation

- On a horizontally striped carpet we can cut a room in 2 different ways



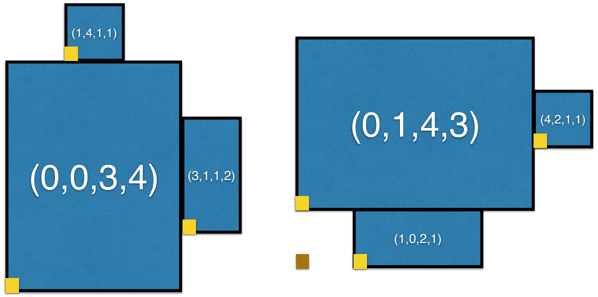
Carpet Orientation

- On a non-symmetric patterned carpet we can cut a room in only 1 way



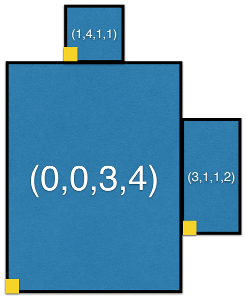
Representing Room Shapes

- Rectangles at offset to shape bottom left
- (x offset, y offset, x size, y size)
- Each rotation is different!



Carpet representation

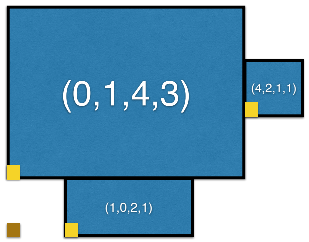
- A layout (rotation) for a carpet
  - set of rectangles and offsets
- Number rectangle and offsets
  - use set to define layouts
- E.g.
  - 1: 0,0,3,4
  - 2: 0,1,4,3
  - 3: 1,4,1,1
  - 4: 3,1,1,2
  - 5: 4,2,1,1
  - 6: 1,0,2,1



representation {1,3,4}

Carpet representation

- A layout (rotation) for a carpet
  - set of rectangles and offsets
- Number rectangle and offsets
  - use set to define layouts
- E.g.
  - 1: 0,0,3,4
  - 2: 0,1,4,3
  - 3: 1,4,1,1
  - 4: 3,1,1,2
  - 5: 4,2,1,1
  - 6: 1,0,2,1



representation {2,5,6}

Carpet Cutting

- Given *n* rooms defined by fixed shapes, each with a possible rotations. Place the shapes on a carpet of height *h* so they don't overlap in minimizing the length *l* used.

```
int: n; % number of rooms
set of int: ROOM = 1..n;
int: m; % number of rectangle/offsets
set of int: ROFF = 1..m;
array[ROFF,1..4] of int: d; % defns
set of int: ROT = 1..4;
array[ROOM,ROT] of set of ROFF: shape;
int: h; % height of roll
int: maxl; % maximum length of roll
```


Carpet Cutting Data

- Sample data, three rooms with two configurations, {} indicates non-configuration

```
n = 3; m = 7;
d = [| 0,0,3,4  % (xoffset,yoffset,xsize,ysize)
    | 0,1,4,3
    | 1,4,1,1
    | 3,1,1,2
    | 4,2,1,1
    | 1,0,2,1
    | 0,0,4,3 |];
shape = [| {1,3,4}, {2,5,6}, {}, {}
         | {1,3,4}, {2,5,6}, {}, {}
         | {1}, {7}, {}, {}  |];
h = 7; maxl = 12;
```


Carpet Cutting Decisions + Objective

- For each object
  - x position of its base
  - y position of its base
  - which shape is used

```
array[ROOM] of var 0..maxl: x;
array[ROOM] of var 0..h: y;
array[ROOM] of var ROT: rot;

var 0..maxl: l; % length of carpet used

solve minimize l;
```

Carpet Cutting Constraints

- Disallow non-configurations

```
forall(i in ROOM)(shape[i,rot[i]] != {});
```

- For each rectangle/offset in each object
  - it fits within the carpet area

```
forall(i in ROOM)(forall(r in ROFF)
      (r in shape[i,rot[i]] ->
        (x[i] + d[r,1] + d[r,3] <= l /\
         y[i] + d[r,2] + d[r,4] <= h)));
```

  - Can it stick out the bottom or left?
  - No, since offsets are positive

Carpet Cutting Constraints

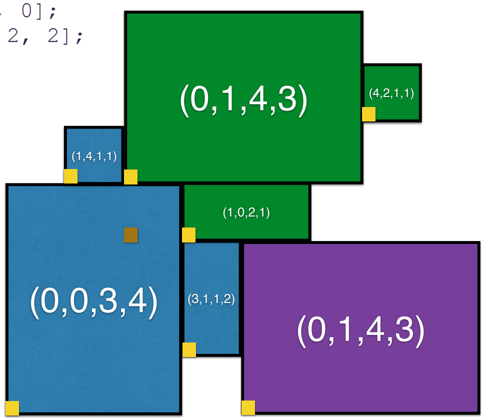
- Rectangle/offsets don't overlap

```
forall(i,j in ROOM where i < j)
   (forall(r1,r2 in ROFF)
      (r1 in shape[i,rot[i]] /\
       r2 in shape[j,rot[j]] ->
(x[i] + d[r1,1] + d[r1,3] <= x[j] + d[r2,1]
                          \/
 x[j] + d[r2,1] + d[r2,3] <= x[i] + d[r1,1]
                          \/
 y[i] + d[r1,2] + d[r1,4] <= y[j] + d[r2,2]
                          \/
y[j] + d[r2,2] + d[r2,4] <= y[i] + d[r1,2])
   ));
```


Carpet Cutting Example Solution

- With the tiny example we get

```
l = 8;
x = [0, 2, 4];
y = [0, 3, 0];
rot = [1, 2, 2];
```



Packing Globals

- `diffn` is extensible to *k* dimensions
  - in MiniZinc `diffn_k`
- The `geost` global constraint enforces non-overlap of objects
  - objects may have multiple possible shapes
  - each shape is a set of offset rectangles

geost Global Constraint

```
predicate geost_bb(int: k,
                   array[int,int] of int: rect_size,
                   array[int,int] of int: rect_offset,
                   array[int] of set of int: shape,
                   array[int,int] of var int: x,
                   array[int] of var int: kind,
                   array[int] of var int: l,
                   array[int] of var int: u)
```

- Arguments
  - k = number of dimensions
  - rectangle sizes: row = rectangle,col = dimension
  - rectangle offsets: row = rect, col = dim
  - shape definitions (sets of rectangle/offsets)
  - position of each object
  - kind (shape) of each object
  - lower and upper bounds on each dimension

Example geost Constraint

- one constraint (almost)

```
predicate geost_bb(2,
       [| 3,4 | 4,3 | 1,1 | 1,2 | 1,1 | 2,1 | 4,3 |],
       [| 0,0 | 0,1 | 1,4 | 3,1 | 4,2 | 1,0 | 0,0 |],
       [ {1,3,4}, {2,5,6}, {1}, {7} ],
       [| x[1],y[1] | x[2],y[2] | x[3],y[3] |],
       kind,
       [ 0,0 ],
       [ 1,h ]);

kind[1] in {1,2};
kind[2] in {1,2};
kind[3] in {3,4};
```


Overview

- Complex packing problems
  - make shapes from components
  - ensure components don't overlap
- Globals
  - `diffn_k` (for *k* dimensional packing)
  - `geost` (for flexible *k* dimensional packing)
- In practice most packing is 2D or 3D

EOF