Multiple Modeling

Peter Stuckey

Multiple Models

- Discrete optimization problems often have
  - multiple viewpoints on the same problem
- We can build two (or more) completely distinct models to solve the same problem
- We can also combine them

Overview

- Determine a function from one set to another
  - when the function is a bijection
- This is a complete matching:
  - match each *d* in DOM with a different *c* in COD
  - or equivalently match each *c* in COD with a different *d* in DOM
  - two complementary models

Complete Matching

- A bijective function has two viewpoints
- The (usual) function

```
array[DOM] of var COD: f;
```

- And the inverse function

```
array[COD] of var DOM: finv;
```

- Some constraints of the problem may be easier to express using the function, or its inverse
- Why not use both!

Matching Workers to Tasks

- What about when the number of workers and tasks are equal

```
int: n;
set of int: W = 1..n;
set of int: T = 1..n;
array[W,T] of int: profit;
```

- Original decisions: which task does a worker work on?

```
array[W] of var T: task;
```

- Inverse decisions: which worker works on a task?

```
array[T] of var W: worker;
```

Matching Workers with Tasks

- Given the profit array below

	t1	t2	t3	t4
w1	7	1	3	4
w2	8	2	5	1
w3	4	3	7	2
w4	3	1	6	3

- Which model is likely to better?
  - Original

```
alldifferent(task);
maximize sum(w in W)(profit[w,task[w]]);
```

  - Inverse

```
alldifferent(worker);
maximize sum(t in T)(profit[worker[t],t]);
```

Combined Models

- We can combine the two models
- We need to make the two functions agree

```
forall(w in W, t in T)
        (task[w] = t <->
         worker[t] = w);
```

- This is captured by the global constraint
  - `inverse(task, worker)`
  - or `inverse(worker, task)`
  - Note we can remove the `alldifferent` constraints, made redundant by `inverse`
- Why would we combine models?

inverse

- The inverse global constraint enforces that two functions are inverses of each other
  - and hence bijections
- Inverse constraint
  - inverse(<function array>,<invfunction array>)

```
predicate inverse(array[int] of var int:f,
                  array[int] of var int:if)=
  forall(i in index_set(f), j in index_set(if))
        (f[i] = j <-> if[j] = i);
```

Combined Models

- For the pure assignment problem. No!
- What about side constraints
  - w1 works on a smaller numbered task than w4
  - t3 and t4 are worked on by workers that are distance 2 apart in number
  - w1 works on t2 iff w3 works on t4
- Encode these for the two models

Combined Models

- For the pure assignment problem. No!
- What about side constraints
  - w1 works on a smaller numbered task than w4

```
task[1] < task[4];
```

  - t3 and t4 are worked on by workers that are distance 2 apart in number

```
abs(worker[3] - worker[4]) = 2;
```

  - w1 works on t2 iff w3 works on t4

```
task[1] = 2 <-> task[3] = 4;
worker[2] = 1 <-> worker[4] = 3;
```

- Encode these for the two models

Combined Models

- For some solvers the combined model may be advantageous in solving
- For example: Better search

	t1	t2	t3	t4
w1	7	1	3	4
w2	8	2	5	1
w3	4	3	7	2
w4	3	1	6	3

- Are we better searching on task or worker?
  - Searching on worker variables is better
  - The profit is correlated with task not worker!

Overview

- Multiple viewpoints of the problem leads to
  - multiple models
- Different viewpoints can express different constraints
  - more succinctly
  - better for the solvers (usually succinct is better)
- Combining the models can improve on either single model

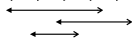
EOF

Multiple Modeling: Langfords

Peter Stuckey

Langfords Problem Example

- L(*m*,*n*): Given *m* copies of each of the numbers 1..*n* give a sequence of these numbers where there is *k* digits between every two consecutive copies of digit *k*
- For example L(2,3): 3,1,2,1,3,2



- How is this an assignment problem?
- Map DOM = 1_1, 1_2, 2_1, 2_2, 3_1, 3_2
  - the ordered copies of the digits
- to COD = 1,2,3,4,5,6
  - the position in the sequence

Langfords Example

```
int: n;
set of int: DIG = 1..n;
int: m;
set of int: COPY = 1..m;
int: l = m*n;
set of int: POS = 1..l;
array[DIG,COPY] of var POS: x;
```

- Constraints

```
forall(d in DIG, c in 1..m-1)
      (x[d,c+1] = x[d,c] + d + 1);
alldifferent([ x[d,c]
             | d in DIG, c in COPY]);
```

What is InverseLangfords?

- We need to map DIG x COPY to a single integer: $d_c = m^*(d\text{-}1) + c$
  - $1_1, 1_2, 2_1, 2_2, 3_1, 3_2 = 1, 2, 3, 4, 5, 6$

```
set of int: DIGCOP = 1..l;
array[POS] of var DIGCOP: y;
```

- How do we model the constraints?
  - $y[p] = d_c \iff x[d,c] = p$

```
forall(d in DIG, c in 1..m-1,
       p in POS)
      (y[p]  = m*(d-1)  + c <->
       y[p+d+1] = m*(d-1)  + c + 1);
```

- Terrible encoding!
  - if position p has d_c then p+d+1 has d_{c+1}

A note about the inverse constraints

```
forall(d in DIG, c in 1..m-1,
       p in POS)
      (y[p]  = m*(d-1)  + c <->
       y[p+d+1]  = m*(d-1)  + c + 1);
```

- Notice that we are accessing positions outside the y array, e.g. $l + d + 1$
- But this is correct
  - None but the last copy of a digit d_m can occur in the last d positions of the sequence
  - Relational semantics requires $y[i] = j$ evaluates to false when $i > p$

A note about the inverse constraints

- Safer to avoid out of array access
  - but clumsy in this instance

```
forall(d in DIG, c in 1..m-1,
       p in POS)
      (y[p]  = m*(d-1)  + c <->
       if p+d+1 in POS then
           y[p+d+1]  = m*(d-1)  + c + 1
       else false endif);
```

Two Distinct Models

- Langfords:
  - $x[d,c]$ = position of d_c
- InverseLangfords
  - $y[p]$ = the digit d_c in position p
- Which runs faster?
- Which one is easier to print the Langford sequence?

```
output [show((y[p]-1) div m + 1)++" "
        | p in POS ];
3 1 2 1 3 2
```

Combined Langford Model

- We can combine the models with an inverse constraint
- Omit the `alldifferent` constraints
  - they are implied by `inverse`
- Omit the inverse models encoding of the constraints
  - they are implied by the equations on $x[d,c]$
- The combined model can solve better in CP
  - searching on x and y variables simultaneously

EOF

Permutation Problems

Peter Stuckey

Permutation Problems

- An important class of matching problems are permutation problems
  - Given a set of objects OBJ place them in an order
- This is a matching of OBJ with $1..n$
  - where n is the cardinality of OBJ
- Langfords is a permutation problem

Photo Problem

- Given *n* people line them up for a photo with the most friendliness, defined as the sum of the friendliness between each pair of people adjacent in the line.

```
int:n ;
set of int: PERSON = 1..n;
set of int: POS = 1..n;
array[PERSON,PERSON] of int: friend;
```

- How should this be modelled?

PhotoProblem Model One

- Variables: the position of each person

```
array[PERSON] of var POS: x;
```

- Constraints:

```
alldifferent(x);
```

- Objective ???????

```
solve maximize …
```

- Hard to see how to express objective
- This is the wrong viewpoint

PhotoProblem Model Two

- Variables

```
array[POS] of var PERSON: y;
```

- Constraints

```
alldifferent(y);
```

- Objectives

```
solve maximize sum(i in 1..n-1)
                  (friend[y[i],y[i+1]]);
```

- Easy to express constraints, and objective!

LineTSP Problem

- Given a set of cities on a line, and a set of precedences amongst the cities, visit each city in turn starting from position 0 to satisfy the precedences and minimize the total distance travelled.

```
int: n; % number of cities
set of int: CITY = 1..n;
set of int: POS = 1..n;
array[CITY] of int: pos; % position of city
int: m; % number of precedences
set of int: PREC = 1..m;
array[PREC] of CITY: left;
array[PREC] of CITY: right;
```

LineTSP Model

- Decisions
  - order: the posn of each city in the permutation
  - city: the city at each position

```
array[CITY] of var POS: order;
array[POS] of var CITY: city;
```

- Constraints (inverse and precedences)

```
inverse(order,city);
forall(i in PREC)
      (order[left[i]] < order[right[i]]);
```

- Objective

```
solve minimize sum(i in 1..n-1)
     (abs(coord[city[i]] - coord[city[i+1]]));
```

Overview

- Permutation problems
  - always have two viewpoints
- Choose the viewpoint that is:
  - easy to express constraints and objective
- Or choose both viewpoints and add
  - inverse constraint

EOF

More Multiple Models

Peter Stuckey

More Multiple Models

- If there are two ways to represent a decision we can have multiple models
- Remember 01 Knapsack

```
array[OBJ] of var 0..1: x
var set of OBJ: s
```

- Channel

```
forall(o in OBJ)
      (x[i] = bool2int(o in s));
```

- Is there an advantage!
  - NO

Channeling Set Representations

- Fixed cardinality set representations
  - set variable

    ```
    var set of OBJ: s;
    card(s) = u;
    ```

  - array of values

    ```
    array[1..u] of OBJ: x;
    forall(i in 1..u-1)(x[i] > x[i+1]);
    ```

- Channeling

```
forall(o in OBJ)
      (o in s -> exists(i in 1..u)(x[i] = o));
forall(i in 1..u)(x[i] in s);
```

Can Set Channelling be Useful?

- CrazySets Problem
- Choose *m* subsets of 1..*n* in order each of size *c* such that the intersection of each three sets is empty, and the least element of each set is less than the least element of next, and the second least element of each set is less than the second least element of the next set, similarly for the third to *c*th element.

```
int: n; % maximum value
set of int: NUMBER = 1..n;
int: c; % cardinality of sets
int: m; % number of sets
```

CrazySets

- For example given

```
n = 10;
c = 3;
m = 4;
```

- one solution is
  - { 1, 3, 4 },
  - { 2, 4, 6 },
  - { 3, 5, 8 },
  - { 5, 8, 9 }
- Note how
  - each column of elements is increasing
  - intersection of any 3 sets is empty

CrazySets with Set Variables

- Variables

```
array[1..m] of var set of NUMBER: s;
```

- Constraints: empty intersection of triples

```
forall(i,j,k in 1..m where i < j /\ j < k)
     ( s[i] intersect s[j] intersect s[k] = {} )
```

- Very hard to represent the ordering

CrazySets with Arrays

- Variables

```
array[1..m,1..c] of var NUMBER: x;
forall(i in 1..m, j in 1..c-1)
      ( x[i,j] < x[i,j+1] );
```

- Constraints: Order of elements

```
forall(i in 1..m-1, j in 1..c)
      ( x[i,j] < x[i+1,j] );
```

- Hard to represent the intersection

CrazySets combined

- Channel

```
forall(i in 1..m, o in NUMBER)
      (o in s[i]
       -> exists(j in 1..c)(x[i,j] = o));
forall(i in 1..m, j in 1..c)(x[i,j] in s[i]);
```

- Constraints
  - non-intersection from s variables
  - ordering from x variables.

CrazySets Better Model

- In fact we can represent the intersection constraint using the array model
- If the intersection of any 3 sets is empty, then no element appears more than twice
- This is a cardinality constraint, over all elements in the arrays x
- Add this to array model

```
global_cardinality_low_up_closed(
    [ x[i,j] | i in 1..m, j in 1..c ],
    [ i | i in 1..n ],     % for each number
    [ 0 | i in 1..n ],     % at least 0
    [ 2 | i in 1..n ]);    % at most 2
```

Combined Models

- For CrazySets we did find a way, but variations would be difficult
- e.g. the cardinality of the intersection of any three sets can be at most 1

```
forall(i,j,k in 1..m where i < j /\ j < k)
 (1 >= card(s[i] intersect s[j] intersect s[k]));
```

- Hard to see how to model this using arrays

Overview

- The best variable representation of a decision
  - differs for different constraints
- Combined models allow
  - using the best representation for each constraint
  - ensuring multiple representations agree
- Worthwhile when
  - difference in solving for representations is large
  - cost of making representations agree is not too big

EOF