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

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

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]]));
```