# Using Predicates

Jimmy Lee & Peter Stuckey

## Recruiting Talents

2

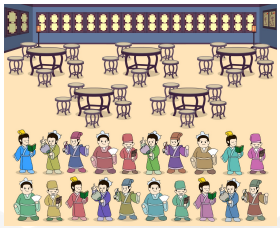
## The Scholar Feast

3

## Seating Planning

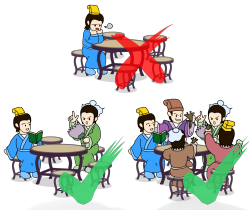
4

## Seating Planning



5

## No Lonely Tables



6

## No Enemies on the Same Table

7

## Reputation and Key Guests


8

## Reputation and Key Guests


9

## Key Guests on Separate Tables


10

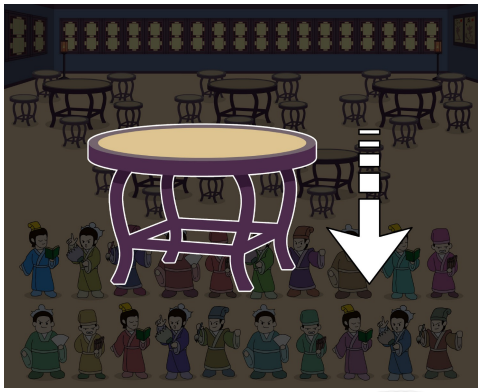
## Key Guests on Separate Tables


11

## Friends on Same Table

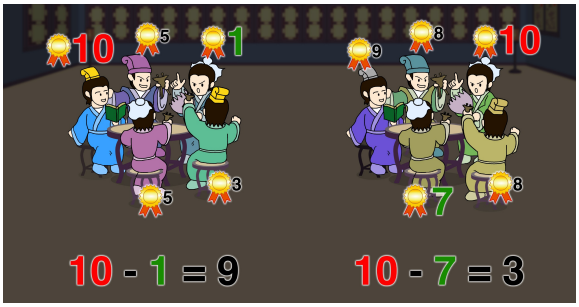
12

## Primary Objective



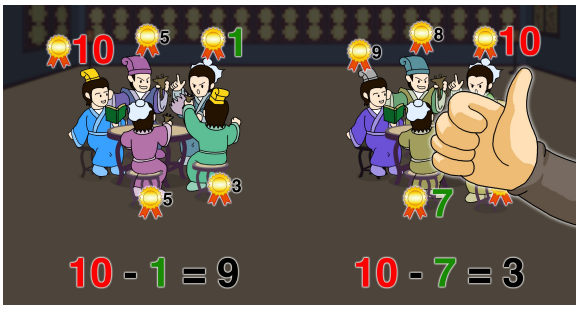
13

## Secondary Objective




14

## Secondary Objective




15

## TableSeating Problem

- The problem is to seat *n* guests at a banquet. There are *T* tables that each seat *S* people. Couples amongst the guests must not be seated at the same table. No two key guests can be at the same table. We wish to minimize the total of the table numbers of the couples.

```
n = 20;
T = 5;
S = 5;
couples = [| 1, 2 | 4, 5 | 6, 7 | 8, 10
           | 11, 12 | 13, 14 | 17, 18 |];
key = {1, 6, 7, 8};
```

16

## TableSeating Data (table_seating.mzn)

- Data

```
enum SCHOLAR;
array[SCHOLAR] of int: reputation;
int: maxreputation = max(reputation);

int: T;
set of int: TABLE = 1..T;
int: S; % tables size

array[int,1..2] of SCHOLAR: enemies;
array[int,1..2] of SCHOLAR: friends;
```

- Decisions

```
array[TABLE] of var set of SCHOLAR: table;
```

17

## TableSeating Constraints (table_seating.mzn)

- Cardinality bounds

```
forall(t in TABLE)(card(table[t]) <= S);
```

- No good to have lonely tables

```
forall(t in TABLE)(card(table[t]) != 1);
```

- Every one gets exactly one seat
  - Every one gets a seat

```
forall(p in SCHOLAR)
   (exists(t in TABLE)(p in table[t]));
```

  - No one gets more than one seat

```
forall(t1, t2 in TABLE where t1 < t2)
   (table[t1] intersect table[t2] = {});
```

18

## More TableSeating Constraints (table_seating.mzn)

- Not-at-the-same-table is reused

```
predicate not_same_table(SCHOLAR: p1, SCHOLAR: p2) =
   forall(t in TABLE)
      (not ({p1,p2} subset table[t]));
```

- Enemies not at the same table

```
forall(c in index_set_1of2(enemies))
   (not_same_table(enemies[c,1],enemies[c,2]));
```

- Key guests not at the same table

```
constraint forall(p1,p2 in SCHOLAR where p1 < p2
      /\ reputation[p1] = 10 /\ reputation[p2] = 10)
   (not_same_table(p1,p2));
```

- Friends at the same table

```
forall(c in index_set_1of2(friends))
   (not(not_same_table(friends[c,1],
      friends[c,2])));
```

19

## index_set

- The `index_set` function returns the range of an array
  - `array[5..9] of int: x;`
  - `index_set(x) = 5..9`
- There are versions for 2d, 3d, … ,6d arrays, e.g.
  - `array[3..7,2..8] of int: y;`
  - `index_set_1of2(y) = 3..7`
  - `index_set_2of2(y) = 2..8`
  - `array[-1..1,2..3,0..1] of int: z;`
  - `index_set_2of3(z) = 2..3`

20

## Primary Objective (table_seating.mzn)

- Use as few tables as possible

```
var int: obj1;
constraint obj1 = sum(t in TABLE)
   (card(table[t]) > 0);
```

21

## Secondary Objective (table_seating.mzn)

- The secondary objective function is large and complex
- Use local variables to break up large exprs

```
var int: obj2;
constraint obj2 = sum(t in TABLE)
   (let {var int: minRep = min(
         [reputation[p]*(p in table[t]) +
            maxreputation*(1-(p in table[t])) |
            p in SCHOLAR]);
         var int: maxRep = max(
         [reputation[p]*(p in table[t]) |
            p in SCHOLAR]);}
   in if minRep = maxreputation then 0
      else maxRep - minRep endif);
```

22

## Multi-Objectives (table_seating.mzn)

- We use `obj1` as the major comparator, and `obj2` is used only when `obj1` is the same
- Estimate the possible values of `obj1` and `obj2`
- Multiply `obj1` by a big enough value so that it dominates `obj2`

```
solve minimize (obj1*100 + obj2);
```

23

## Solving the Model

- The model cannot return any solution **after 10 mins**
- Is there a better model?
- Can we use techniques we learned before?
  - The decisions are "wrong"
    - decide for each person which table
    - channeling
  - Use global_cardinality for counting
  - Remove value symmetry

24

## TableSeating Improved (table_seating_imp.mzn)

- Decide a table for each person and channel

```
array[SCHOLAR] of var TABLE: seat;
```

  - The new viewpoint ensures **automatically** that every person gets exactly one seat at a table
  - The existence and set intersection constraints can be removed
- Make the predicate not_same_table simpler

```
predicate not_same_table(SCHOLAR:p1, SCHOLAR:p2) =
                         seat[p1] != seat[p2];
```

- The following constraints remain unchanged
  - Enemies not at the same table
  - Key guests not at the same table
  - Friends at the same table

25

## Using Predicates

- Predicates have the advantage of
  - providing a layer of abstraction
- Changing the model is simpler
  - when we only have to change the definition of a predicate

26

## TableSeating Improved (table_seating_imp.mzn)

- Use global_cardinality for counting if possible

```
global_cardinality_low_up(seat, [t|t in TABLE],
                [0|t in TABLE], [S|t in TABLE]);
```

  - The card() constraint can be **removed**

- The table numbers don't matter! **Remove** the value symmetry

```
value_precede_chain([t | t in 1..T], seat);
```

27

## Running the Model

- The improved model is now solved in less than 1 second

```
Table 1: {S1, S5, S13, S18, S19}
Table 2: {S2, S4, S6, S12, S16}
Table 3: {S3, S7, S9, S10, S17}
Table 4: {S8, S11, S14, S15, S20}
Table 5: {}
Obj1: 4 and Obj2: 23
----------
==========
```

28

## Summary

- Predicates
  - provide “macros” for constraints
- Local variables
  - provide names for expressions
- Both can be useful
  - even if we only use the macro/name once
  - at least for model understanding
- Predicates also introduce reflection abilities
  - index set functions: `index_set`
  - variable bounds lookup

29

## Image Credits

All graphics by Marti Wong, ©The Chinese University of Hong Kong and The University of Melbourne 2016

30