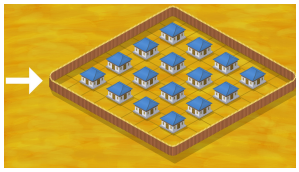
The let-in Construct

Jimmy Lee & Peter Stuckey

Raiding the Bandits' Nest

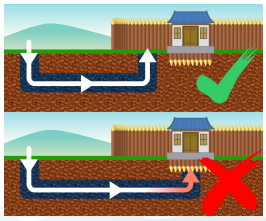


2

Raiding the Bandits' Nest

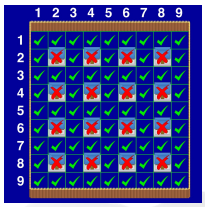
3

Digging Tunnels



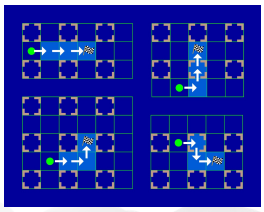
4

Diggable Spots



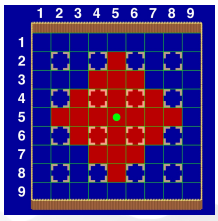
5

The Manhattan Distance



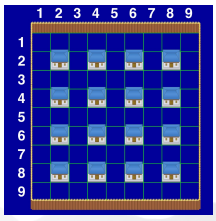
6

The Manhattan Distance



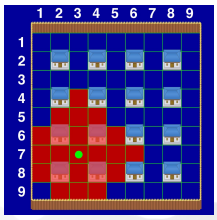
7

All Huts Covered



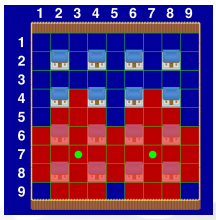
8

All Huts Covered



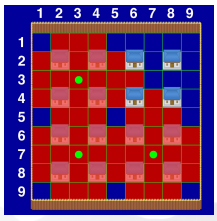
9

All Huts Covered



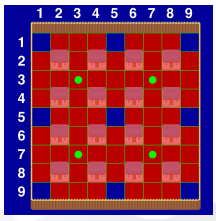
10

All Huts Covered



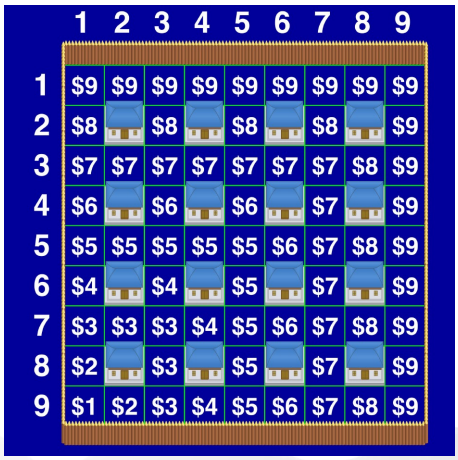
11

All Huts Covered



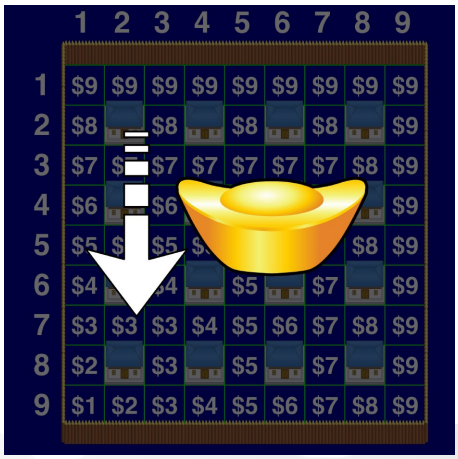
12

Cost of Digging



13

Cost of Digging



14

Data and Variables (bandits.mzn)

- Data

```
int: size; set of int: SIZE = 1..size;
int: nPts; set of int: POINT = 1..nPts;
int: mDist;

array[SIZE,SIZE] of int: cost;
```

- Variables

```
array[POINT] of var SIZE: ptR;
array[POINT] of var SIZE: ptC;
```

15

Constraints (bandits.mzn)

- All tunnel exits different

```
alldifferent(
   [(ptR[i]-1)*size + ptC[i] |
      i in POINT]
);
```

- Kind of tedious and difficult to understand.

16

The let-in construct

- The let-in construct allows new variables and parameters to be introduced and used locally at "any point" in the model
- Format

```
let {<type>:<varname> [ = <expr>] ;
          …
     <type>:<varname> [ = <expr>] [;]} in
<expr>
```

- Parameters introduced must have a defining expression, but variables do not
- New vars/pars visible **only** in remainder of let-in expression

17

Constraints (bandits.mzn)

- All tunnel exits different

```
let {array [POINT] of var 1..size*size:
   points = [(ptR[i]-1)*size + ptC[i] |
         i in POINT]} in
      alldifferent(points);
```

- Make the definition more modular and easier to understand.

18

Constraints (bandits.mzn)

- Defining whether a spot is diggable
  - Only the non-building spots

```
forall(i in POINT)
   (not((ptR[i] mod 2) = 0 /\
      (ptC[i] mod 2) = 0));
```

OR

```
forall(i in POINT)
   ((ptR[i] mod 2) = 1 \/
      (ptC[i] mod 2) = 1);
```

19

Constraints (bandits.mzn)

- All huts must be covered

```
predicate covered(var int: x, var int: y) =
   let {var POINT: i, var int: dist =
      abs(x-ptR[i]) + abs(y-ptC[i])} in
         dist <= mDist;

constraint let {
   array [1..(size div 2)] of 1..size-1:
      huts = [i*2|i in 1..(size div 2)]} in
         forall(i,j in huts)(covered(i,j));
```

- See how let-in helps modularize expressions and improves readability

20

Objective (bandits.mzn)

- Minimizing the total cost of building the tunnels

```
var int: total_cost = sum(i in POINT)
   (cost[ptR[i],ptC[i]]);
solve minimize total_cost;
```

21

Running the Model (bandits.mzn)

```
Row: [5, 7, 3]
Col: [2, 6, 6]
Cost: 18
----------
==========
```

22

Another let-in Example

- Recall the Patrol problem
- Replace

```
forall(d in DAY)
      (sum(s in SOLDIER)
         (roster[s,d] = EVE) >= l);
forall(d in DAY)
      (sum(s in SOLDIER)
         (roster[s,d] = EVE) <= u);
```

- By

```
forall(d in DAY)
  (let {var int: on = sum(s in SOLDIER)
                      (roster[s,d]=EVE)
       } in l <= on /\ on <= u);
```

- Easier than array of intermediates

23

Defining Division with Local Variable

- Suppose your solver does not support `div`
- Division using multiplication
  - non-negative only
  - $x \text{ div } y = z \leftrightarrow x = z * y + r, 0 \leq r < y$
- We need a new variable *r* for remainder
- Code

```
predicate divp(var int:x,var int: y,var int: z)
   = let {var 0..ub(y)-1: r} in
      x = z * y + r /\ r < y;
```

- What happens with division by **zero**?
- What happens with division by **negative**?

24

Predicate Assumptions Using Assert

- Our predicate assumed non-negativity
  - let's make that assumption explicit
  - using an assertion

```
predicate divp(var int:x,var int: y,var int: z)
     = assert(lb(y) >= 0,
              "divp: y must be non-negative",
              let {var 0..ub(y)-1: r} in
              x = z * y + r /\ r < y);
```

- Advantages
  - Documents the assumption
  - Prevents misuse

25

Reflection Functions

- Sometime we want to know properties of variables in the model
  - lb(x) a lower bound on all possible values of x
  - ub(x) an upper bound on all possible values
  - dom(x) a superset of all possible values of x
  - lb_array(x): lower bound on all vars in array x
  - ub_array(x): upper bound on array
- Beware these are not guaranteed to be the declared bounds

```
var -4..6: x;   var -4..-2: y;
constraint x = abs(y);
lb(x) = 0 or lb(x) = -4  or lb(x) = 2 (-4..-2)
```

26

Assertions again

- Defensive programming requires that we check assumptions in predicates.
  - `assert`(*boolexp*,*stringexp*,*exp*)
    - returns *exp* if *boolexp* holds,
    - otherwise prints *stringexp* and aborts
- For example,

```
var -4..4:x;
var -4..4:y;
constraint divp(x,y,3);
```

- Results in

```
MiniZinc: evaluation error:
dvip.mzn:9:
   in call 'divp'
dvip.mzn:2:
   in call 'assert'
  Assertion failed: divp: y must be non-negative
```

27

Summary

- Let-in expressions
  - allow introduction of parameters and variables for use locally
- Local parameters must be defined by =
- Local decisions need not be
- Help modularize expressions and improve readability of models

28

Image Credits

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