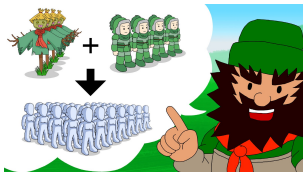
# Basic Model Improvement

Jimmy Lee & Peter Stuckey

## Small Troops

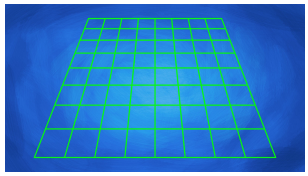


## Inserting Strawmen

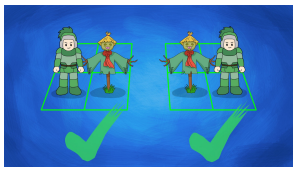




## A Grid Formation

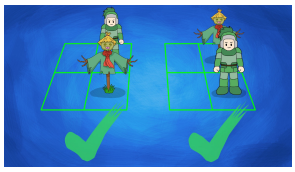


## A Real Soldier Beside Each Strawmen



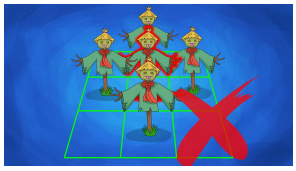
5

## A Real Soldier Beside Each Strawmen



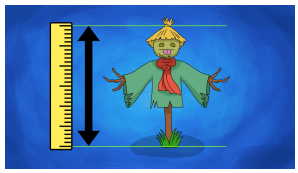
6

## A Real Soldier Beside Each Strawmen



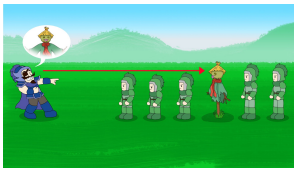
7

## Strawmen of Equal Height



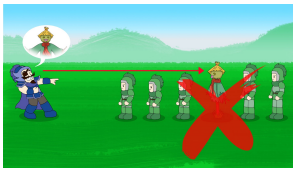
8

## Covering Up Strawmen



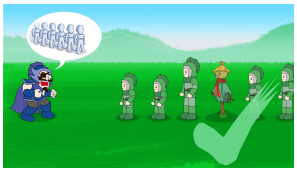
9

## Covering Up Strawmen



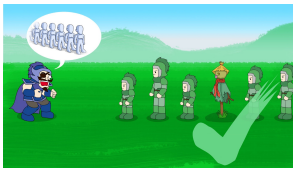
10

## Covering Up Strawmen



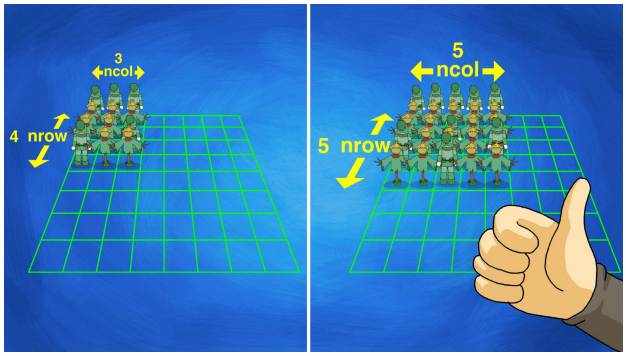
11

## Covering Up Strawmen



12

## Maximizing the StrawArmy



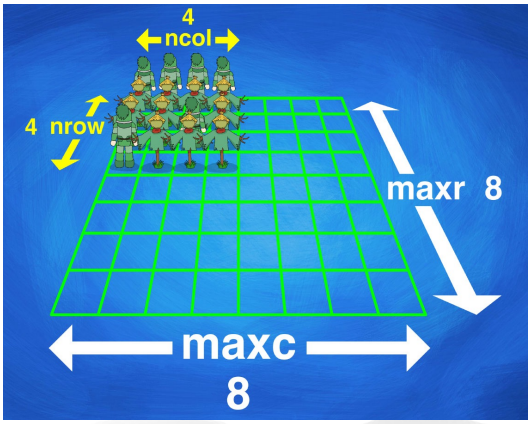

15

## Parading the Soldier

- Hearing that Guan Yu is returning, Liu Bei races out to meet him with a few troops. But Xiahou Dun is close behind. Liu Bei orders a parade of the troops adding straw soldiers to make them look larger
  - The parade fills `nrow` rows and `ncol` columns
  - All straw soldiers have height `strawheight`
  - Each straw solider has a real soldier adjacent
  - No straw soldier has only soldiers of less or equal height in front of them
  - The objective is to maximise the number of apparent soldiers (`nrow * ncol`)

16

## Modeling Trick




17

## Parade Data and Decisions (parade.mzn)

- Data

```
int: nsoldier; int: strawheight;
set of int: SOLDIER = 1..nsoldier;
set of int: SOLDIER0 = 0..nsoldier;
array[SOLDIER] of int: height;
int: maxr; % max rows
set of int: ROW = 1..maxr;
int: maxc; % max columns
set of int: COL = 1..maxc;
```

- Decisions (positions and size of parade)
  - soldier in position, or 0 = straw or empty

```
array[ROW,COL] of var SOLDIER0: x;
var ROW: nrow; var COL: ncol;
```

18

## Parade Constraints (parade.mzn)

- Each real soldier appears at most once

```
forall(r1,r2 in ROW, c1,c2 in COL where
       r1 != r2 \/ c1 != c2)
      (x[r1,c1] = 0
       \/ x[r1,c1] != x[r2,c2]);
```

- Inefficient
  - duplicates the = 0 condition many times
  - constrains each pair twice

19

## Efficient Loops

- Make sure that you don't create the same constraint multiple times
- Ensure loops are no larger than they need to be
- Add tests/constraints in loops as early as possible

20

## Parade Constraints (parade.mzn)

- Each real soldier appears at most once

```
forall(i in 1..maxr*maxc)
    (let {int: r1 = (i-1) div ncol + 1;
          int: c1 = (i-1) mod ncol + 1; } in
     x[r1,c1] = 0 \/
     forall(j in i+1..maxr*maxc)
        (let {int: r2 = (j-1) div ncol + 1;
              int: c2 = (j-1) mod ncol + 1; } in
         x[r1,c1] != x[r2,c2] ));
```

- Iterates over the “position numbers”
  - tests = 0 condition once
  - constrains each pair once
- But there is a better way!

21

## Global constraints

- Be familiar with the globals library
- Look for cases where you can use a global
- They are likely to be much more efficient
  - even if they are implemented by decomposition
  - it will be a different decomposition tailored for each solver

22

## Parade Constraints (parade.mzn)

- Each real soldier appears at most once

```
include "alldifferent_except_0.mzn";
alldifferent_except_0(
    [ x[r,c] | r in ROW, c in COL ]);
```

- A global constraint that does the job exactly
- You need to be aware of what is in the library!

23

## Parade Constraints (parade.mzn)

- Each real soldier appears at least once

```
forall(s in SOLDIER)
      (exists(r in ROW, c in COL)
            (x[r,c] = s));
```

- Inefficient
  - direct translation of the constraint
  - highly disjunctive
- Can we use a global?

24

## Parade Constraints (parade.mzn)

- Each real soldier appears at least once

```
include "global_cardinality.mzn";
global_cardinality(
       [x[r,c] | r in ROW, c in COL],
       [s | s in SOLDIER],
       [1 | s in SOLDIER]);
```

- Encodes that each soldier appears exactly once!
  - Can remove the `alldifferent_except_0`
- Is there a simpler way?

25

## Parade Constraints (parade.mzn)

- Each real soldier appears at least once

```
sum(r in ROW, c in COL)(x[r,c] != 0) = nsoldier;
```

- Count the number of soldiers appearing
  - relies on `alldifferent_except_0` to make sure all soldiers appearing are different

26

## Parade Constraints (parade.mzn)

- Each real soldier appears in the nrow*ncol area

```
forall(r in ROW, c in COL)
      (    (r > nrow -> x[r,c] = 0)
       /\ (c > ncol -> x[r,c] = 0));
```

- Force positions outside the parade to be empty

27

## Parade Constraints (parade.mzn)

- Each straw solider has a real soldier adjacent (right, left, behind, front)

```
forall(r in ROW, c in COL)
   ((r <= nrow /\ c <= ncol /\ x[r,c] = 0) ->
    (   if c < maxc then x[r,c+1] != 0 else false endif
     \/ if c > 1 then x[r,c-1] != 0 else false endif
     \/ if r < maxr then x[r+1,c] != 0 else false endif
     \/ if r > 1 then x[r-1,c] != 0 else false endif));
```

- Note the use of if-then-else to avoid out of bounds

28

## Parade Constraints (parade.mzn)

- No straw soldier has only soldiers of less or equal height in front of it

```
not exists(r1 in ROW, c in COL)
    ((r1 <= nrow /\ c <= ncol /\ x[r1,c] = 0) ->
      forall(r2 in ROW)(r2 < r1 ->
                (x[r2,c] = 0 \/
                 height[x[r2,c]] <= strawheight))
     );
```

- LHS of -> implication tests for straw soldier
- RHS checks all soldiers in front are no taller
- **BUT** the check `r2 < r1` is fixed

29

## Parade Constraints (parade.mzn)

- No straw soldier has only soldiers of less or equal height in front of it

```
not exists(r1 in ROW, c in COL)
    ((r1 <= nrow /\ c <= ncol /\ x[r1,c] = 0) ->
      forall(r2 in 1..r1-1)
            (x[r2,c] = 0 \/
             height[x[r2,c]] <= strawheight)
     );
```

- Better but inefficient
  - Disjunction in inner loop

30

## Auxiliary Data

- If a model is easier to state with some auxiliary data
  - create the auxiliary data (once)
  - create simpler constraints for the solver using it
  - automatic CSE (Common Subexpression Elimination) might not find this case
- Notice if the model is really dependent on something not given in the data!

31

## Parade Constraints (parade.mzn)

- No straw soldier has only soldiers of less or equal height in front of it

```
array[SOLDIER0] of int: hx =
      array1d(SOLDIER0,[strawheight]++height);
not exists(r1 in ROW, c in COL)
    ((r1 <= nrow /\ c <= ncol /\ x[r1,c] = 0) ->
      forall(r2 in 1..r1-1)
            (hx[x[r2,c]] <= strawheight)
     );
```

- Inefficient
  - Negation is terribly hard for solvers

32

## Things to Avoid

- negation `not C`
  - solvers definitely do not like negation
  - replace negation with negated form:
    - `not x = y` becomes `x != y`
- disjunctions `C1 \/ C2`
  - they will be necessary
  - try to make them as simple as possible
- implications `C1 -> C2`
  - make C1 especially simple
    - it's in a negated context
- existential loops `exists([ .. ])`

33

## Parade Constraints (parade.mzn)

- **Every straw soldier has a taller soldier in front**

```
array[SOLDIER0] of int: hx =
      array1d(SOLDIER0,[strawheight]++height);
forall(r1 in ROW, c in COL)
    ((r1 <= nrow /\ c <= ncol /\ x[r1,c] = 0) /\
      exists(r2 in 1..r1-1)
            (hx[x[r2,c]] > strawheight)
     );
```

- Push negations inside quantifiers
  - forall is much more effective than exists

34

## Parade Objective (parade.mzn)

- Maximize the number of apparent soldiers

```
solve maximize nrow*ncol;
```

- Solving the model

```
 7  9 13 12 10  8  9  .
 .  .  .  .  .  .  .  .
 . 11  .  .  .  8  .  .
 .  .  .  3  .  .  .  .
 4  .  .  .  .  .  6  .
 .  .  6  .  7  .  .  .
 .  .  .  .  .  .  .  .
 .  .  .  .  .  .  .  .
nrow = 6;
ncol = 7;
```

35

## Parade Objective (parade.mzn)

- Maximize the number of apparent soldiers

```
solve maximize nrow*ncol;
```

- Solving the model

```
 7  9 13 12 10  8  9  .
 .  .  .  .  .  .  .  .
 . 11  .  .  .  8  .  .
 .  .  .  3  .  .  .  .
 4  .  .  .  .  .  6  .
 .  .  6  .  7  .  .  .
 .  .  .  .  .  .  .  .
 .  .  .  .  .  .  .  .
nrow = 6;
ncol = 7;
```

36

## Models and Solvers

- Remember that constraint solvers take
  - a set of variables and
  - a conjunction of constraints
- And return an answer
- They are designed to handle problems in this form
- Models can look very different to this
- Models that are simple conjunctions are easier to solve
  - other constructs need to be translated
  - they are more difficult for solvers

37

## Size of Models

- The larger the model in general the more difficult to solve
- Make use of global constraints
- Beware sometimes the global will be decomposed without you noticing!
- Run `minizinc -k` (keep files)
- Examine the size of the FlatZinc (`.fzn`) file

38

## Summary

- There are many ways to model a problem, and the difference between a good model and a bad model can be exponential
- Avoid negation, disjunction, exists, bool2int
- Size of models matter
- Calculate useful intermediates for reuse
  - whether fixed or variable
- Look for other ways to model the constraints

39

## Image Credits

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

40