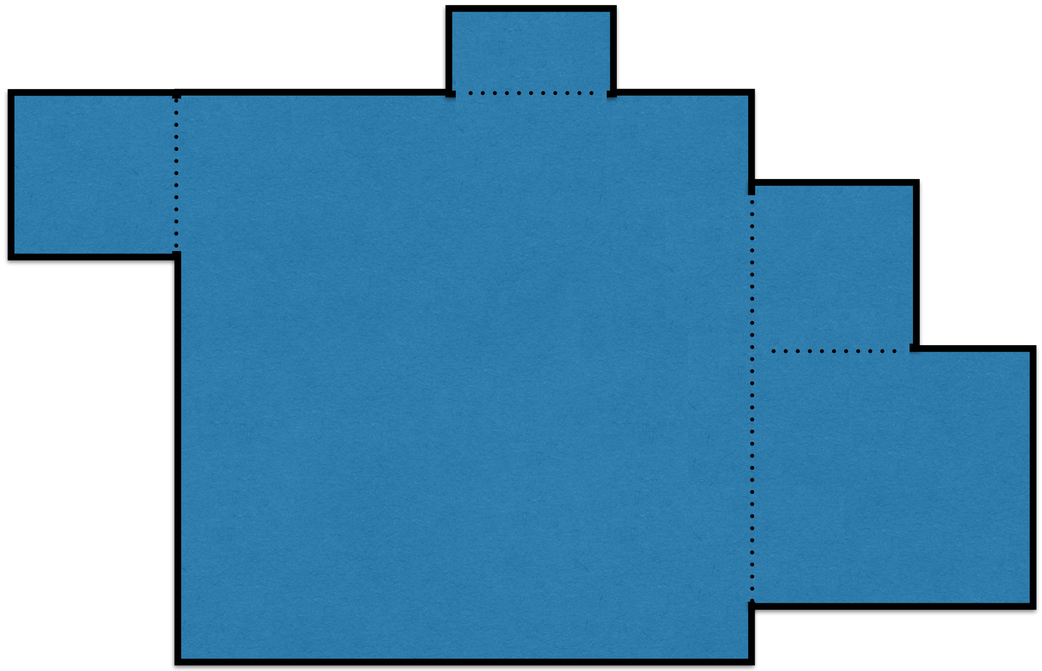
# Rectilinear Packing with Rotation

Peter Stuckey & Guido Tack

MONASH University

# A Shape Usually Has 4 Orientations



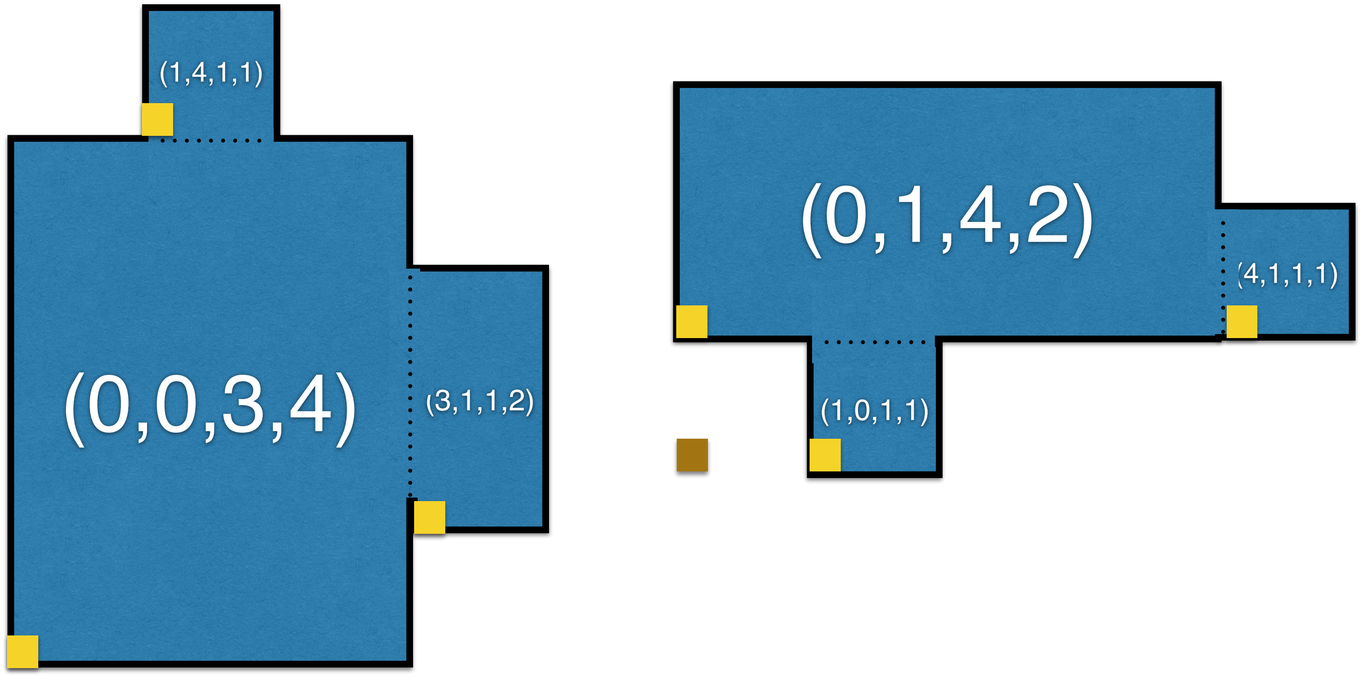
# Orientation Exceptions

- A plain **rectangle** has only **two** orientations
- A **square** only has a **single** orientation

- There can be application-specific restrictions
  - e.g., objects must be able to stand stably on their own

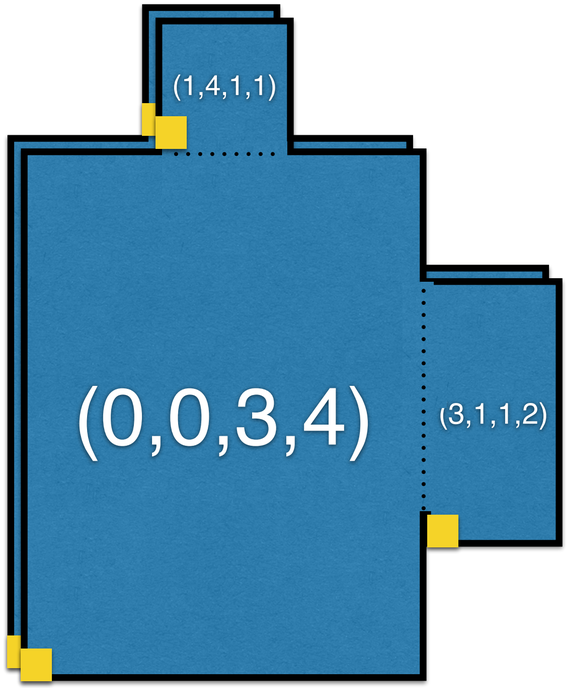
# Representing Block Shapes

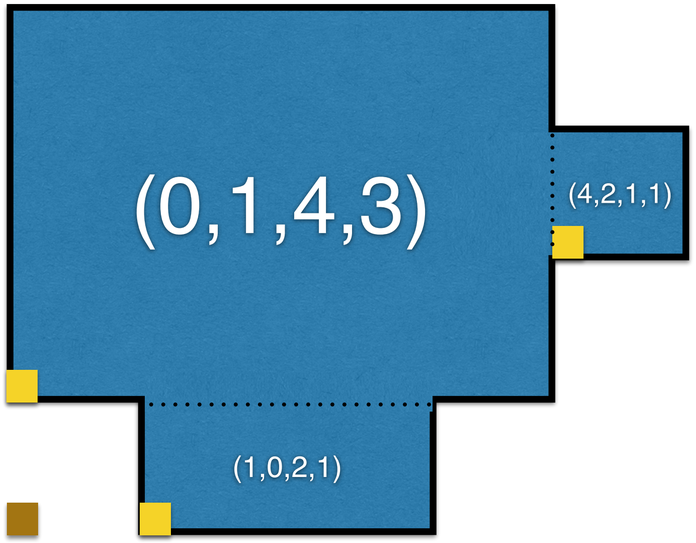
- Rectangles at offset to shape bottom left
  - $(x_{offset}, y_{offset}, x_{size}, y_{size})$
- The offsets are **different** even when the orientation is different (more later)



# Representing Orientation

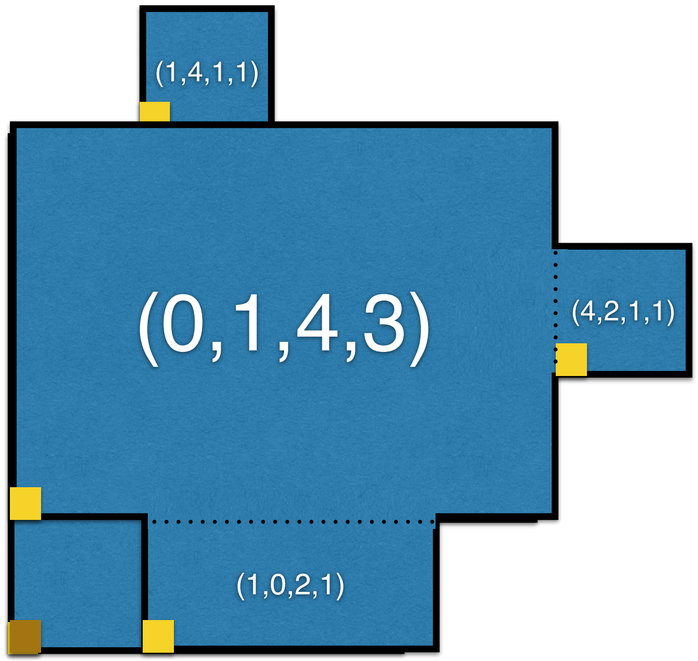
- A new orientation is **no different** from a new block shape
- New origin and new offsets



# Representing Block Shapes

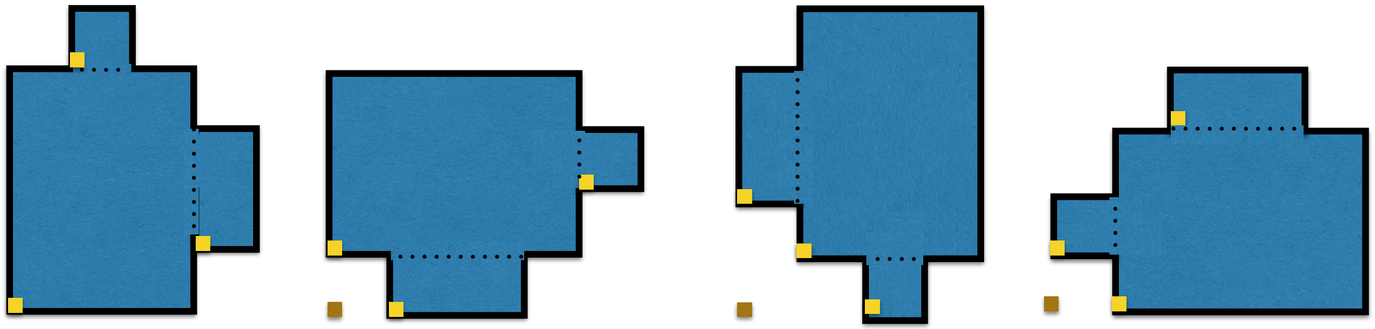
- Represent the component rectangles with offsets
- A block (of a specific orientation) is a set of rectangles with offsets
- Component rectangles can be shared if possible
- E.g. a list of components
  - 1: 0,0,3,4
  - 2: 0,1,4,3
  - 3: 1,4,1,1
  - 4: 3,1,1,2
  - 5: 4,2,1,1
  - 6: 1,0,2,1




representation {1,2,3,4,5,6}

# Representing a Shape

- An orientation is a set of components
- A shape consists of 4 (or less) orientations



- A shape can thus be represented by a **list** (**array**) of sets (of offsets)
- Forbidden/duplicated orientations can be represented by the **empty set**

# Block Packing Data

- Given `n` blocks defined by fixed shapes, **each with possible rotations**. Place the shapes in a rectangle of height `h` so they don't overlap with the width `l` used minimized

```
enum Block;
enum Comp;
enum Dims = {XOff, YOff, XSize, YSize};
array[Comp, Dims] of int: dims;
enum Rot = R(1..4); % 4 possible rotations
array[Block,Rot] of set of Comp: shape;

int: h;    % height of rectangle
int: maxw; % maximum width of rectangle

array[Block] of var 0..maxw: x;
array[Block] of var 0..h: y;
```

# Block Packing Data

```
Block = B(1..5);  Comp = C(1..45); h = 9; maxw = 16;
%                  (xoffset,yoffset,xsize,ysize)
dims = [| 1,0,1,1 | 0,1,5,2 | 0,3,1,1 | 1,0,2,5
        | 3,4,1,1 | 3,1,1,1 | 0,0,1,1 | 3,3,1,1
        | 4,0,1,1 | 0,0,4,4 | 1,4,3,1 | 4,2,2,2
        | 0,2,4,4 | 4,2,1,3 | 2,0,2,2 | 1,0,4,4
        | 0,1,1,3 | 1,4,2,2 | 2,1,4,4 | 2,0,3,1
        | 0,1,2,2 | 1,5,1,2 | 1,0,3,5 | 4,1,1,4
        | 5,3,2,1 | 0,1,5,3 | 1,0,4,1 | 3,0,3,1
        | 0,1,2,1 | 2,1,5,3 | 2,4,4,1 | 4,3,1,3
        | 3,0,1,2 | 1,2,3,5 | 0,2,1,4 | 0,0,1,3
        | 1,0,3,4 | 0,3,3,1 | 0,0,4,3 | 1,0,3,1
        | 0,1,4,3 | 3,1,1,3 | 0,0,3,4 | 0,0,5,4
        | 0,0,4,5 |];
```

# Block Packing Data

```
shape =
[| {C(1),C(2),C(3)}, {C(3),C(4),C(5)},
   {C(6),C(4),C(7)}, {C(8),C(2),C(9)}
 | {C(10),C(11),C(12)}, {C(13),C(14),C(15)},
   {C(16),C(17),C(18)}, {C(19),C(20),C(21)}
 | {C(17),C(22),C(23),C(24)}, {C(11),C(25),C(26),C(27)},
   {C(28),C(29),C(30),C(31)}, {C(32),C(33),C(34),C(35)}
 | {C(44)}, {C(45)}, {}, {}
 | {C(36),C(37)}, {C(38),C(39)},
   {C(40),C(41)}, {C(42),C(43)}
|];
```

# Block Packing Decisions + Objective

- For each block
  - `x` position of its base
  - `y` position of its base
  - **which orientation is used**

```
array[Block] of var 0..maxw: x;
array[Block] of var 0..h: y;
array[Block] of var Rot: rot;

var 0..maxw: w; % width of rectangle used


solve minimize w;
```

# Block Packing Constraints

- **Disallow disabled rotations**

```
forall(i in Block)(shape[i,rot[i]] != {});
```

- Each rectangle component must fit within the full area

```
forall(i in Block)(
  forall(r in shape[i,rot[i]])(
    x[i]+d[r,XOff]+d[r,XSize] <= l /\
    y[i]+d[r,YOff]+d[r,YSize] <= h
  )
);
```

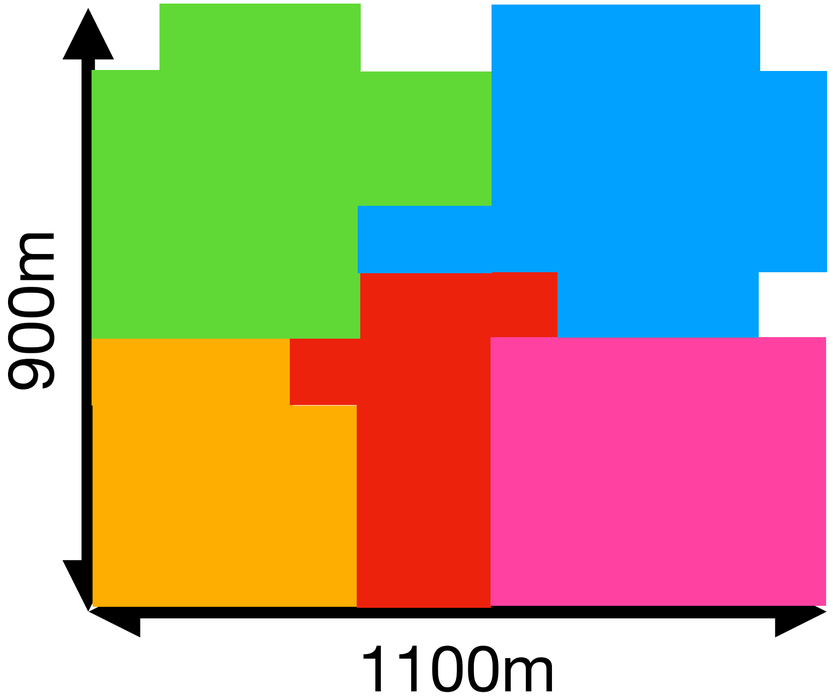
# Block Packing Constraints

- Rectangle/offsets don't overlap

```
forall(i,j in Block where i < j)(
forall(r1 in shape[i,rot[i]],
       r2 in shape[j,rot[j]])(
   x[i]+dims[r1,XOff]+dims[r1,XSize]
                  <= x[j]+dims[r2,XOff]
\/ x[j]+dims[r2,XOff]+dims[r2,XSize]
                  <= x[i]+dims[r1,XOff]
\/ y[i]+dims[r1,YOff]+dims[r1,YSize]
                  <= y[j]+dims[r2,YOff]
\/ y[j]+dims[r2,YOff]+dims[r2,YSize]
                  <= y[i]+dims[r1,YOff]
  )
);
```

# Solving the Model

```
w = 11;
x = [3, 0, 4, 6, 0];
y = [0, 4, 4, 0, 0];
rot = [R(2), R(1), R(3), R(1), R(2)];
----------
==========
Finished in 1m 30s
```



# The `geost` Global Costraint

- The `geost` global constraint enforces non-overlap of objects, like `diffn`, but can take rotations into account
  - objects may have multiple possible shapes
  - each shape is a set of offset rectangles

# The geost Global Constraint

```
predicate geost_bb(
% number of dimensions
  int: k,
% size & offsets: row = comp, col = dim
  array[int,int] of int: rect_size,
  array[int,int] of int: rect_offset,
% shape definitions (sets of components)
  array[int] of set of int: shape,
% position of each object
  array[int,int] of var int: x,
% kind (shape) of each object
  array[int] of var int: kind,
% lower and upper bounds on each dimension
  array[int] of var int: l,
  array[int] of var int: u
)
```

# The geost Global Constraint

- All sizes and offsets are given as 2D arrays

```
array[Comp, 1..2] of int: rect_size =
  array2d(Comp, 1..2, dims[..,XSize..YSize]);

array[Comp, 1..2] of int: rect_offset =
  array2d(Comp, 1..2, dims[..,XOff..YOff]),

array[Block, 1..2] of var int: coord =
  [ (b,i): if i = 1 then x[b] else y[b] endif
    | b in Block, i in 1..2];
```

# The `geost` Global Constraint

- All shapes are given together in a single array (without disallowed shapes)

```
array[int] of set of Comp: geost_shapes
= [ s | s in array1d(shape) where s != {} ]
```

- We can channel the choice of these shapes back to rotations

```
array[Block] of var int: kind = [
  (b-1)*4 + rot[b] % translated index
     % num. of prev. disallowed shapes
  - (count([shape[p,r] = {} | p in ..enum_prev(b),
     r in Rot]) default 0)  | b in Block
];
```

# The geost Global Constraint

- We can then use the geost constraint

```
include “geost.mzn”
constraint geost_bb(
   2,
   rect_size,
   rect_offset,
   geost_shapes,
   coord,
   kind,
   [0,0],
   [w,h],
)
```

# Summary

- Complex packing problems
  - make shapes from components
  - ensure components don't overlap
  - rotations/orientations add to the complexity of the problem
- Globals
  - `diffn_k` (for k dimensional packing)
  - `geost` (for flexible k dimensional packing)
- In practice most packing is 2D or 3D

# EOF