

Klotski Puzzle and Klotski Solver

Zhifan Yang
Ocaml Project

What is Klotski

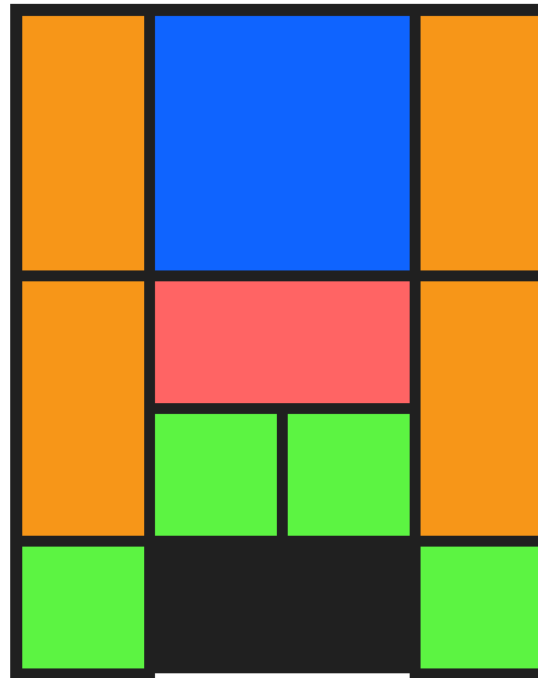
- ▶ Is a sliding block puzzle game which was invented around 20th century
- ▶ Its purpose is to move the biggest square to the exit
- ▶ Can only move blocks in vertical or horizontal direction, and cannot remove any block from the board.
- ▶ Has many variants and different openings
- ▶ Not a easy puzzle; many people give up midway



The most basic opening of Klotski

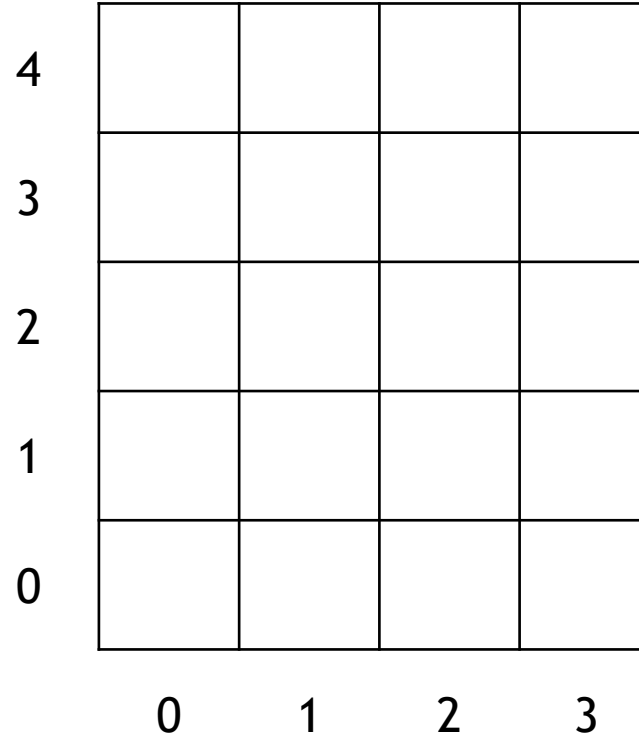
My Klotski Program

- ▶ Play Klotski with three different openings
- ▶ Get help from the computer to automatically solve the puzzle at any time; watch the solution



Representation of the game board

- ▶ The game board can be represented as a 4x5 table
- ▶ Each cell in this table can be described by its x coordinate and y coordinate



A 4x5 grid representing a game board. The grid is composed of 20 empty cells. The x-axis is labeled with 0, 1, 2, and 3 at the bottom. The y-axis is labeled with 0, 1, 2, 3, and 4 on the left side.

4				
3				
2				
1				
0				
	0	1	2	3

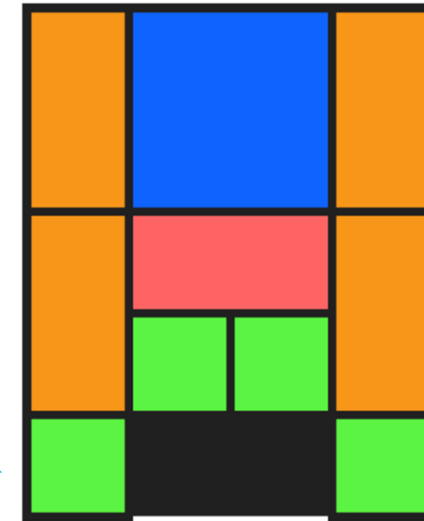
Representation of the blocks

- ▶ Total ten blocks. Assign them with a name and color

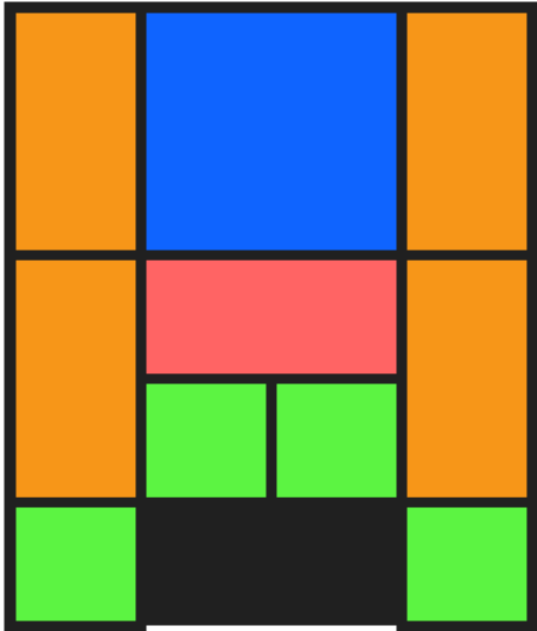
Description	Assigned Name	Assigned Color
2X2 Square	A	Blue
1X2 Rectangle	B1	Orange
1X2 Rectangle	B2	Orange
1X2 Rectangle	B3	Orange
1X2 Rectangle	B4	Orange
2X1 Rectangle	C	Red
1X1 Square	D1	Green
1X1 Square	D2	Green
1X1 Square	D3	Green
1X1 Square	D4	Green

Representation of the blocks

- ▶ Use tuple to represent each block
- ▶ The first element is a string, which is name of the block
- ▶ The second element is a variant type, which contains five integers:
 - ❑ X: x coordinate of the block
 - ❑ Y: y coordinate of the block
 - ❑ W: width of the block
 - ❑ H: height of the block
 - ❑ C: color of the block
- ▶ For example: ("D1", {x=0; y=0; w=1; h=1; c=0x5cf442})



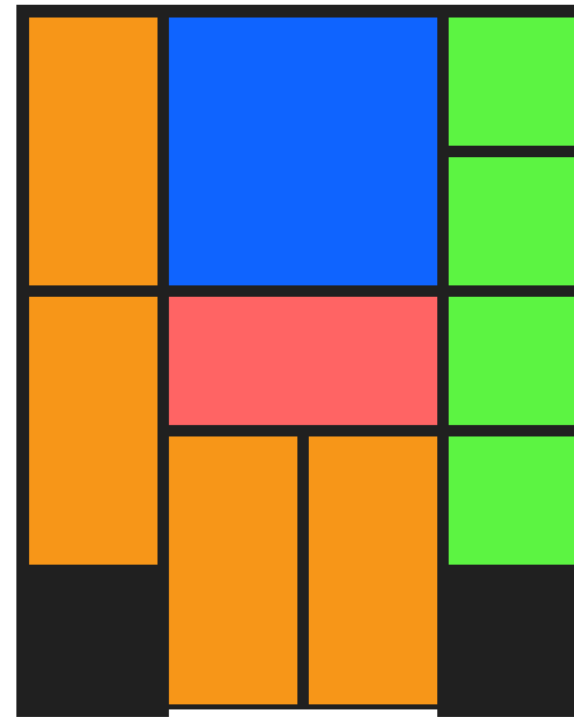
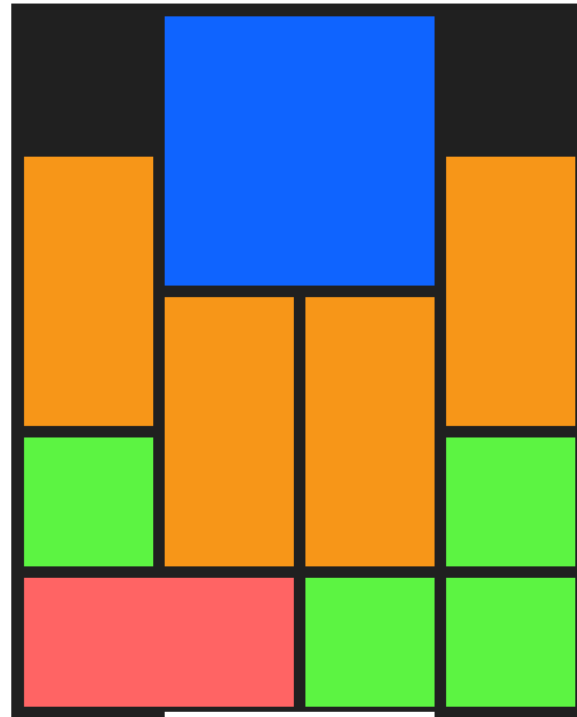
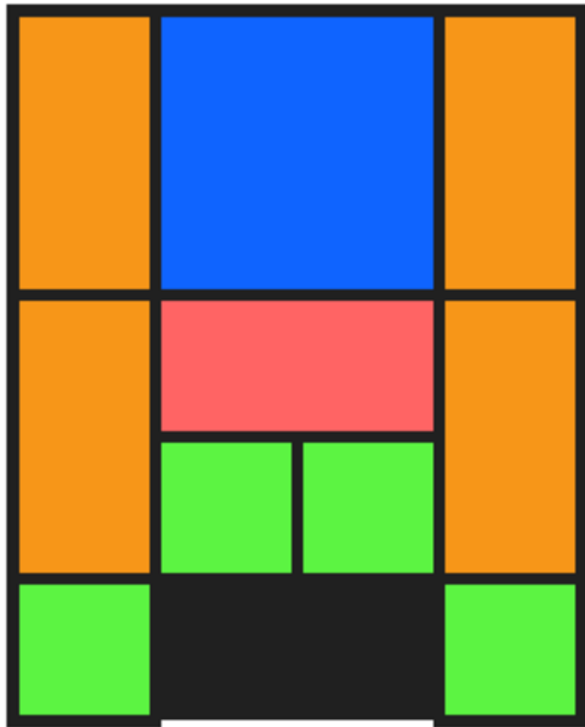
Transformation



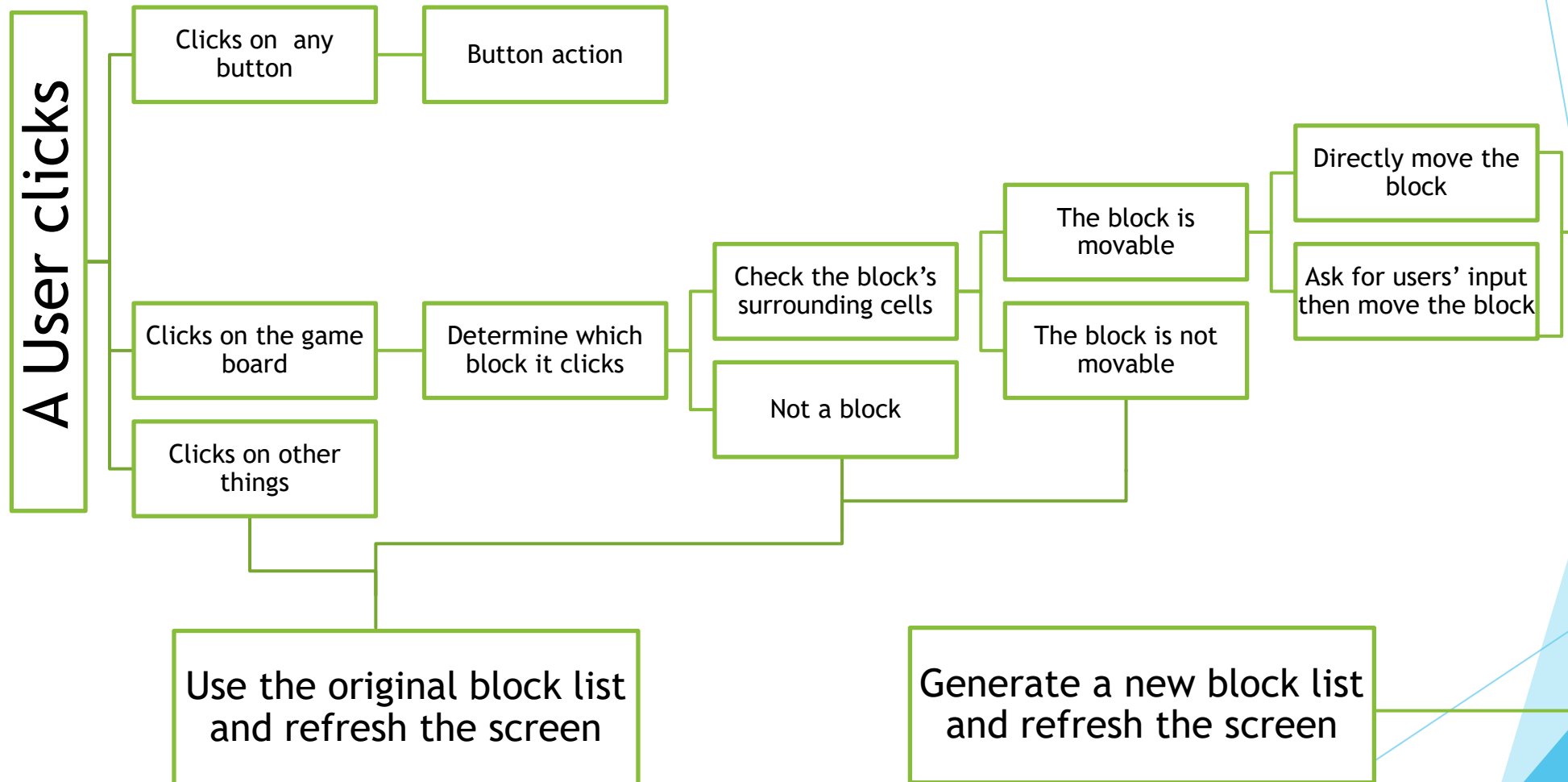
```
[("D1",{x=0; y=0;w=1;h=1;c=0x5cf442 (*green*)});  
 ("D2",{x=3; y=0;w=1;h=1;c=0x5cf442 (*green*)});  
 ("B1",{x=0; y=1;w=1;h=2;c=0xf79618 (*orange*)});  
 ("D3",{x=1; y=1;w=1;h=1;c=0x5cf442 (*green*)});  
 ("D4",{x=2; y=1;w=1;h=1;c=0x5cf442 (*green*)});  
 ("B2",{x=3; y=1;w=1;h=2;c=0xf79618 (*orange*)});  
 ("C", {x=1; y=2;w=2;h=1;c=0xff6464 (*red*)});  
 ("B3",{x=0; y=3;w=1;h=2;c=0xf79618 (*orange*)});  
 ("A", {x=1; y=3;w=2;h=2;c=0x0f64ff (*blue*)});  
 ("B4",{x=3; y=3;w=1;h=2;c=0xf79618 (*orange*)})]
```

For some blocks which occupy more than one cell, the x and y indicate the lower-left corner of the block. However, the program still knows they occupy more than one cell, because their width and/or height are larger than 1.

Three Different Openings

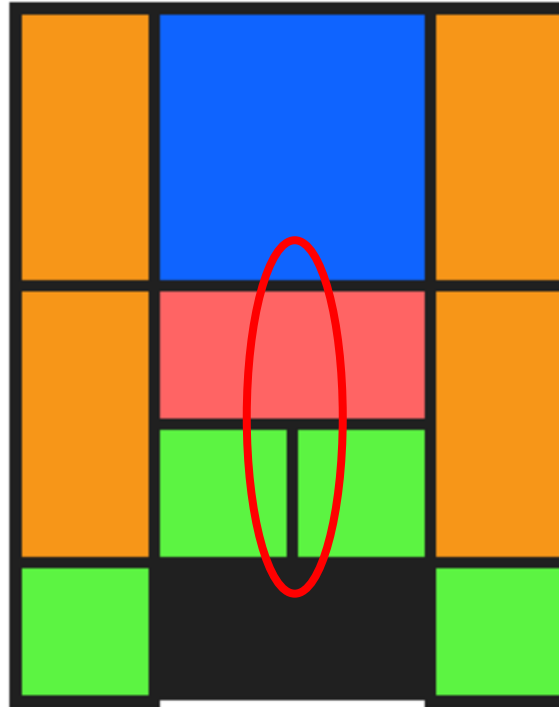


User Action



A Special Case

- ▶ If the mouse clicks within this kind of area, the program will think it does not click on any block
- ▶ However, this is not correct, sometimes.
- ▶ Add a check function
 - If the mouse clicks at cross section, horizontal aisle or vertical aisle, the program will check the neighboring cells. If they belong to the same block, the mouse actually hits the block!

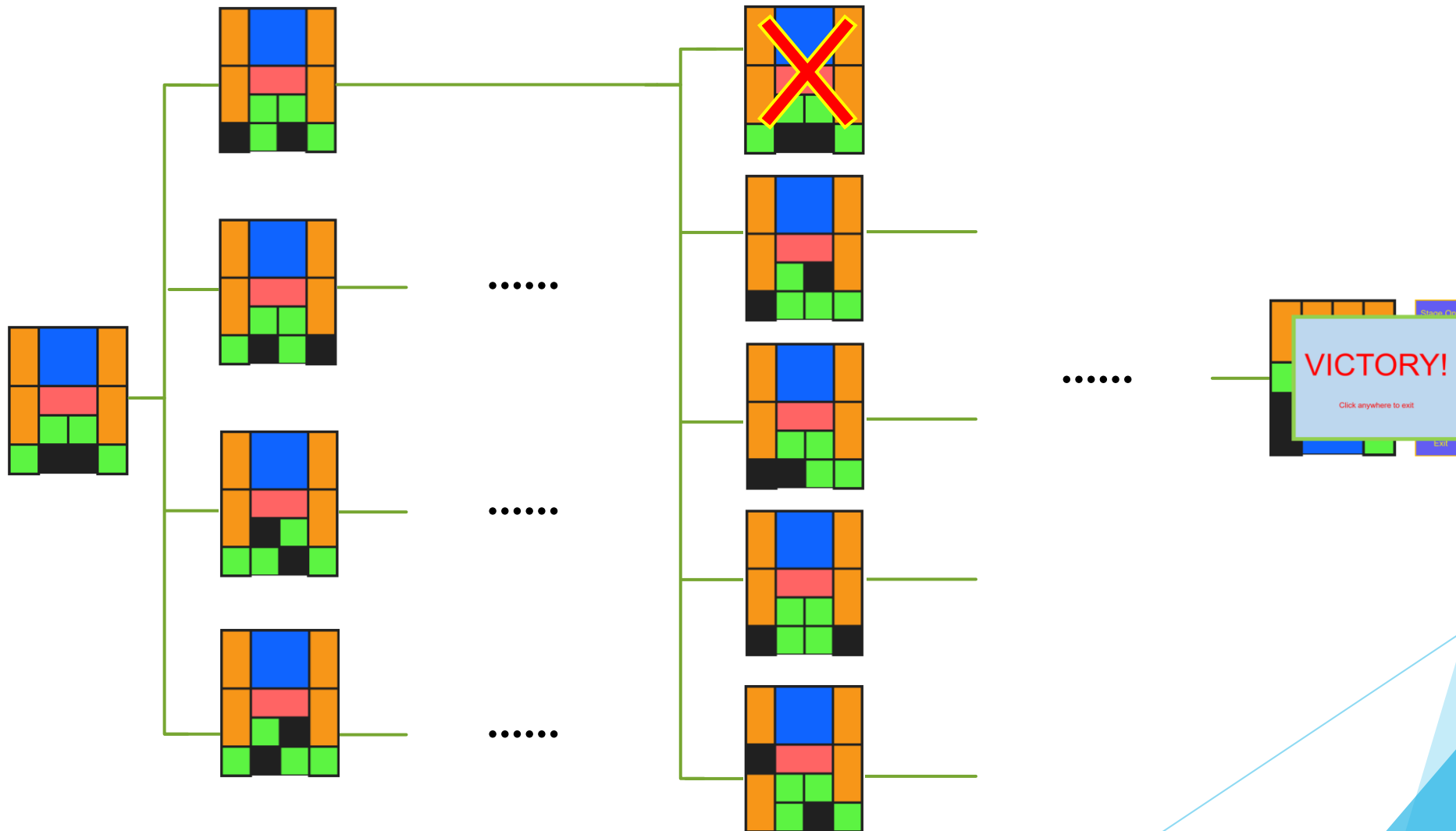


Victory Check

- ▶ Right after the screen is refresh, the program will check if the block list is in “victory state”, i.e. Block A's $x = 1$ and $y=0$
- ▶ If not, wait for user's input
- ▶ If it is, generate a victory screen



Auto Solving Function



Backtrack and Show the Solution

- ▶ After the solution tree is generated, backtracking the solution tree to generate a path, which is our solution
- ▶ Display solution through a simple animation

Auto Solving Comparison

```
[("D1",{x=0; y=0;w=1;h=1;c=0x5cf442 (*green*)));  
 ("D2",{x=3; y=0;w=1;h=1;c=0x5cf442 (*green*)));  
 ("B1",{x=0; y=1;w=1;h=2;c=0xf79618 (*orange*)));  
 ("D3",{x=1; y=1;w=1;h=1;c=0x5cf442 (*green*)));  
 ("D4",{x=2; y=1;w=1;h=1;c=0x5cf442 (*green*)));  
 ("B2",{x=3; y=1;w=1;h=2;c=0xf79618 (*orange*)));  
 ("C", {x=1; y=2;w=2;h=1;c=0xff6464 (*red*)));  
 ("B3",{x=0; y=3;w=1;h=2;c=0xf79618 (*orange*)));  
 ("A", {x=1; y=3;w=2;h=2;c=0xf64ff (*blue*)));  
 ("B4",{x=3; y=3;w=1;h=2;c=0xf79618 (*orange*)))]
```

```
[("D1",{x=0; y=0;w=1;h=1;c=0x5cf442 (*green*)));  
 ("D2",{x=3; y=0;w=1;h=1;c=0x5cf442 (*green*)));  
 ("B1",{x=3; y=1;w=1;h=2;c=0xf79618 (*orange*)));  
 ("D3",{x=1; y=1;w=1;h=1;c=0x5cf442 (*green*)));  
 ("D4",{x=2; y=1;w=1;h=1;c=0x5cf442 (*green*)));  
 ("B2",{x=0; y=1;w=1;h=2;c=0xf79618 (*orange*)));  
 ("C", {x=1; y=2;w=2;h=1;c=0xff6464 (*red*)));  
 ("B3",{x=0; y=3;w=1;h=2;c=0xf79618 (*orange*)));  
 ("A", {x=1; y=3;w=2;h=2;c=0xf64ff (*blue*)));  
 ("B4",{x=3; y=3;w=1;h=2;c=0xf79618 (*orange*)))]
```

- ▶ Actually they are the same, but the program will treat them different
- ▶ Need to override Pervasives.compare
- ▶ If this number is the same, then it is the same

```
| "A" -> convertlist (4*100+position.x*10+position.y)  
| "B" -> convertlist (3*100+position.x*10+position.y)  
| "C" -> convertlist (2*100+position.x*10+position.y)  
| "D" -> convertlist (1*100+position.x*10+position.y)
```

