COMP 3200
Artificial Intelligence

Lecture 7
Assignment 2
Grid Pathfinding Optimizations
Bidirectional Search

New in Assignment 2

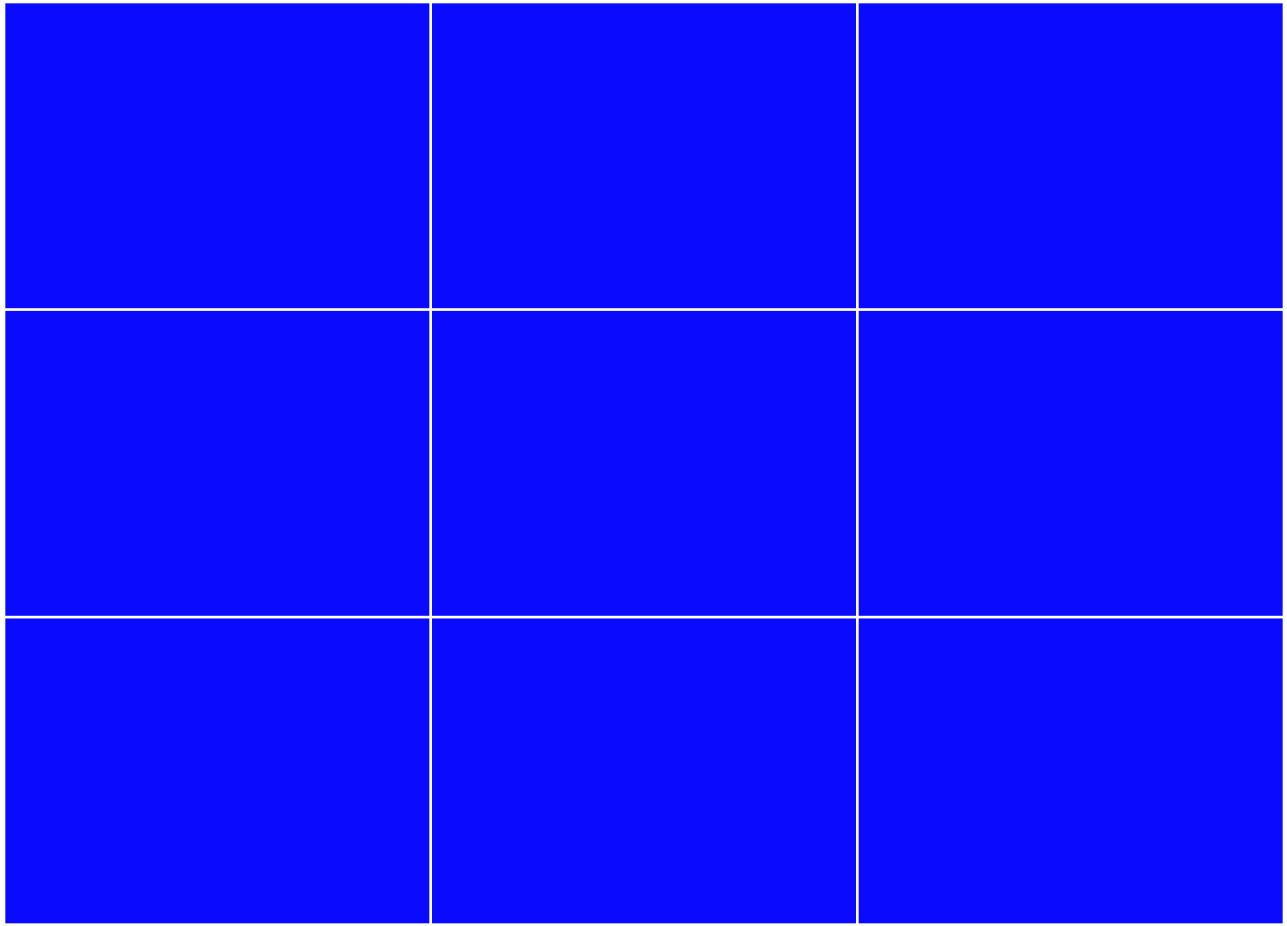
- A* Algorithm
- Multiple object sizes
- 8-directional movement
- Heuristic Functions
- Connected sectors
- Precomputing and Optimization
- 6980: Bidirectional Search

A* Graph-Search

```
1.  function AStar(problem, h)
2.       closed = {}
3.       open = {Node(problem.initial_state)}
4.       while (true)
5.           if (open.empty) return fail
6.           node = remove_min_f(open)
7.           if (node.state is goal) return solution
8.           if (node.state in closed) continue
9.           closed.add(node.state)
10.             for c in Expand(node, problem)
11.                 open.add(c)
```

Node Class

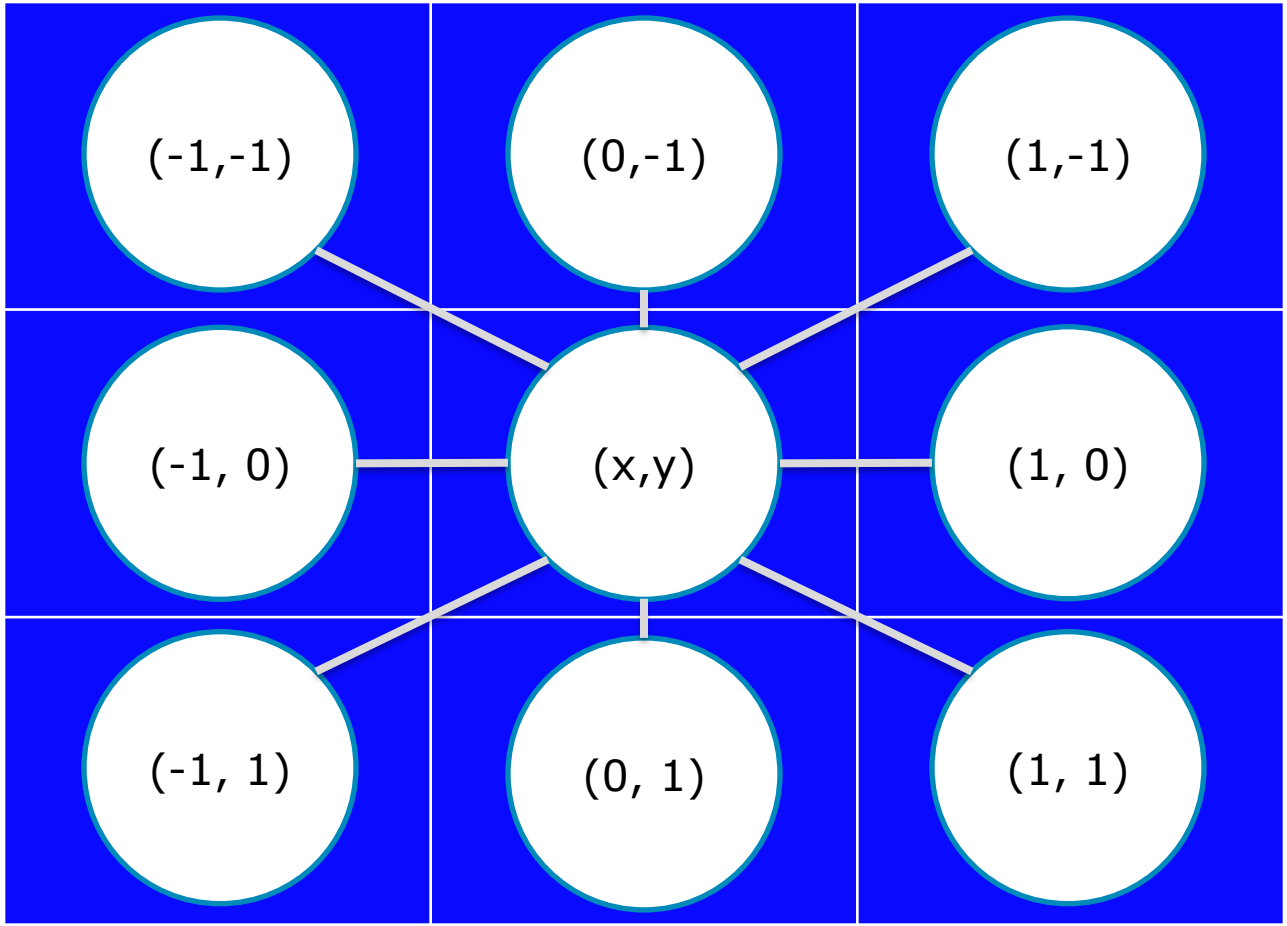
- Node has new variables
  - x, y = state of the node
  - action = action that generated node
  - parent = parent of this node
  - g = g cost of node (cost of path to node)
  - h = heuristic value of node
- NOTE: child.g = parent.g + cost(action)



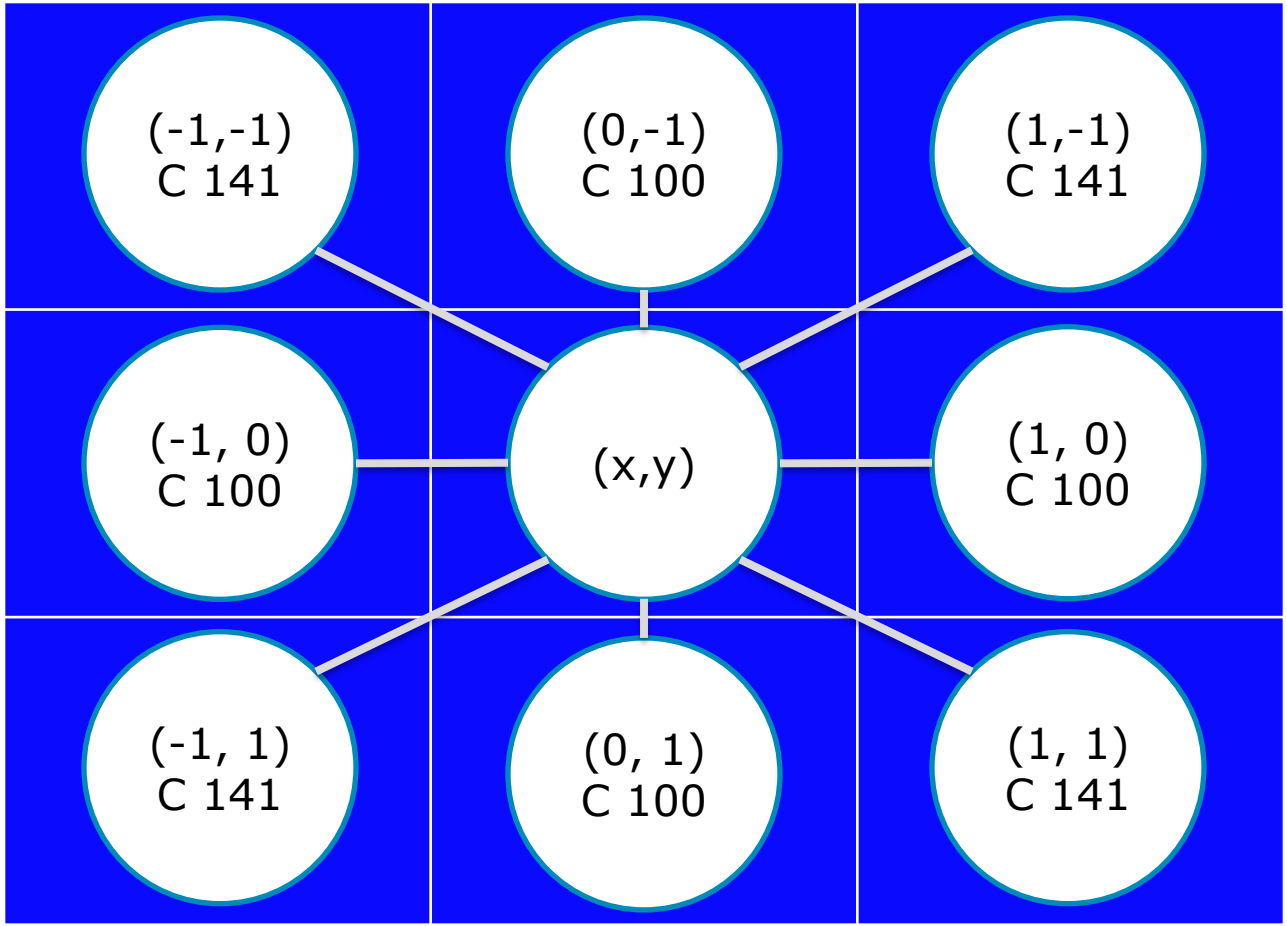
[x,y] States

0,0	1,0	2,0
0,1	1,1	2,1
0,2	1,2	2,2

Action [x,y]

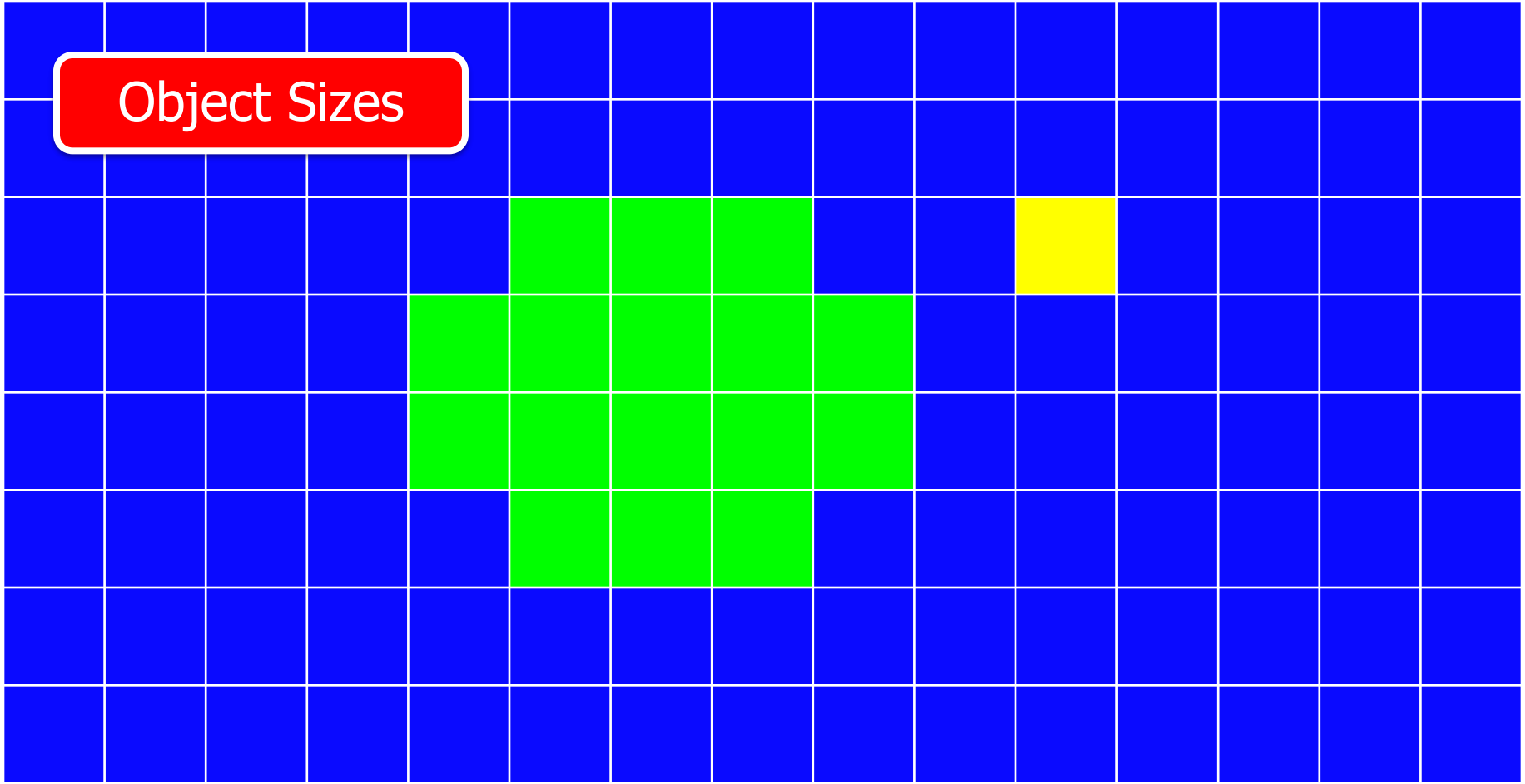


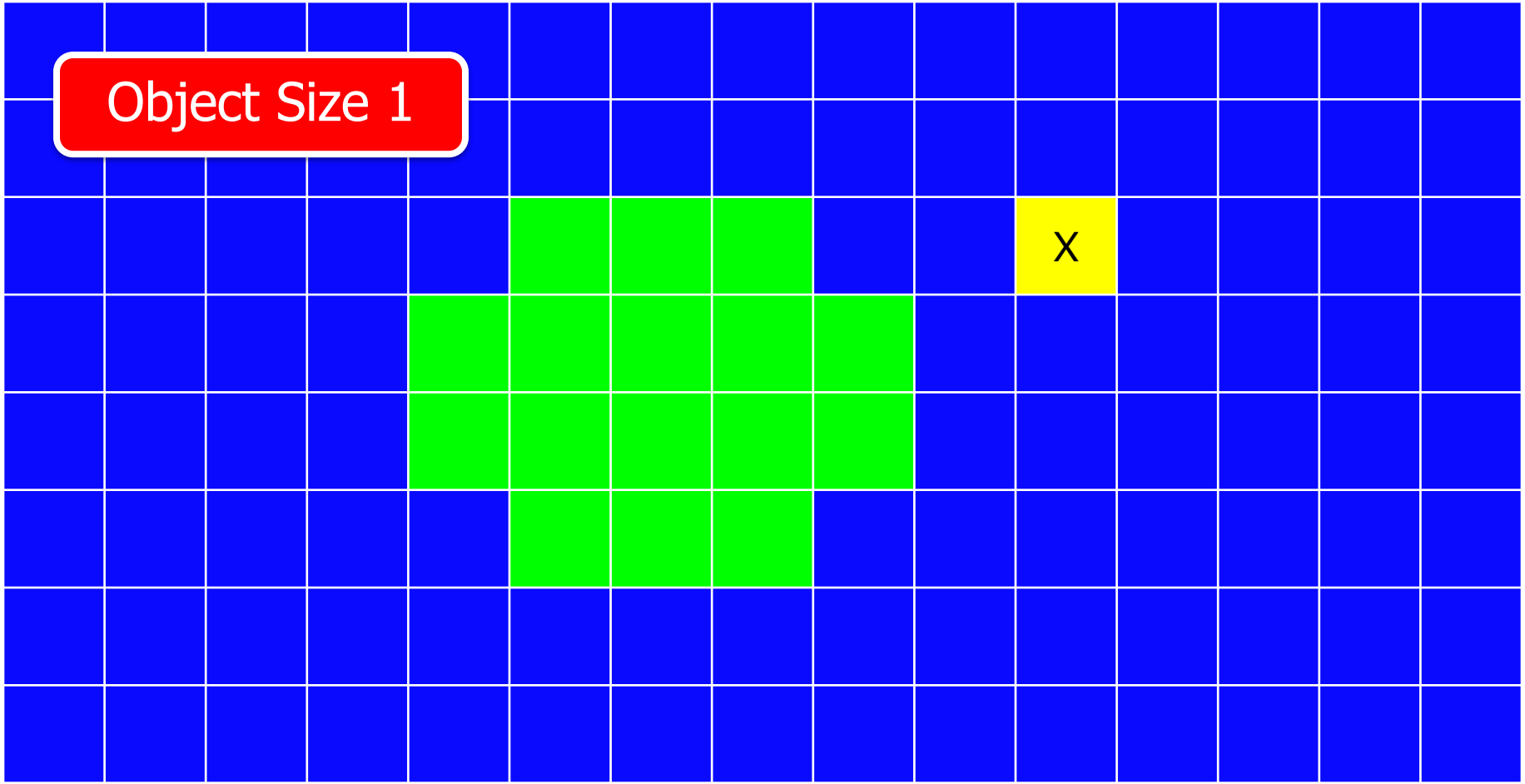
Action Costs

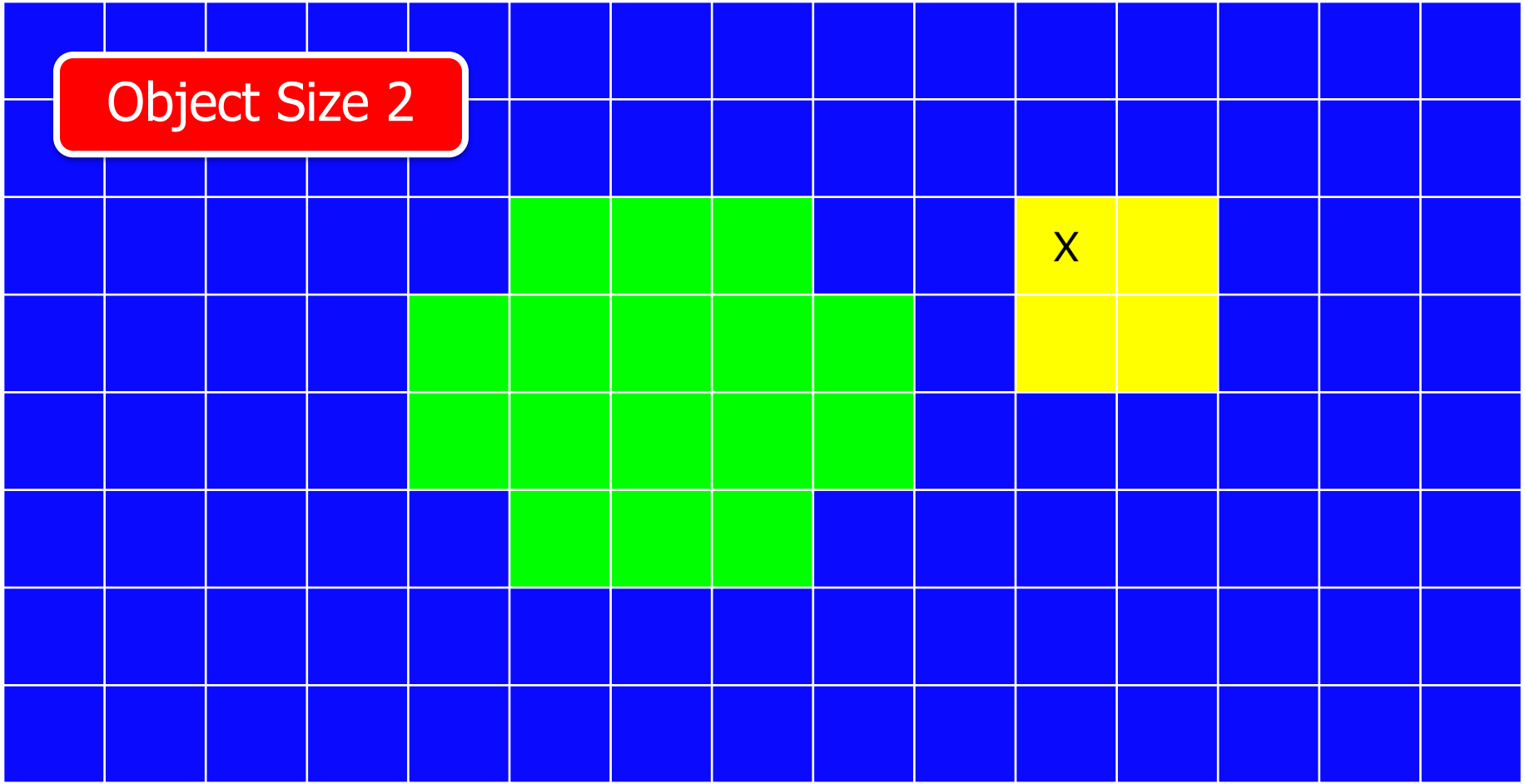


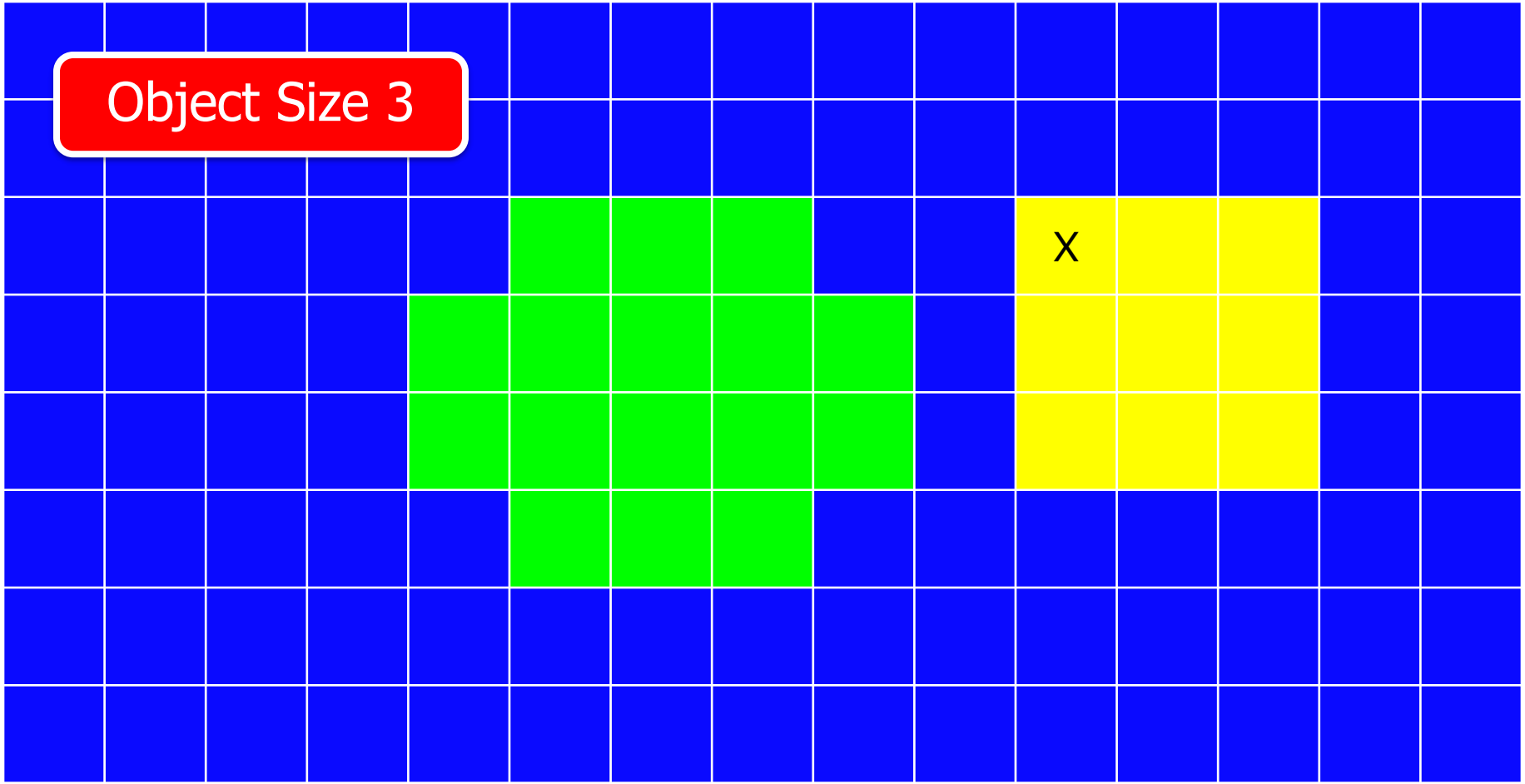
Legal Actions

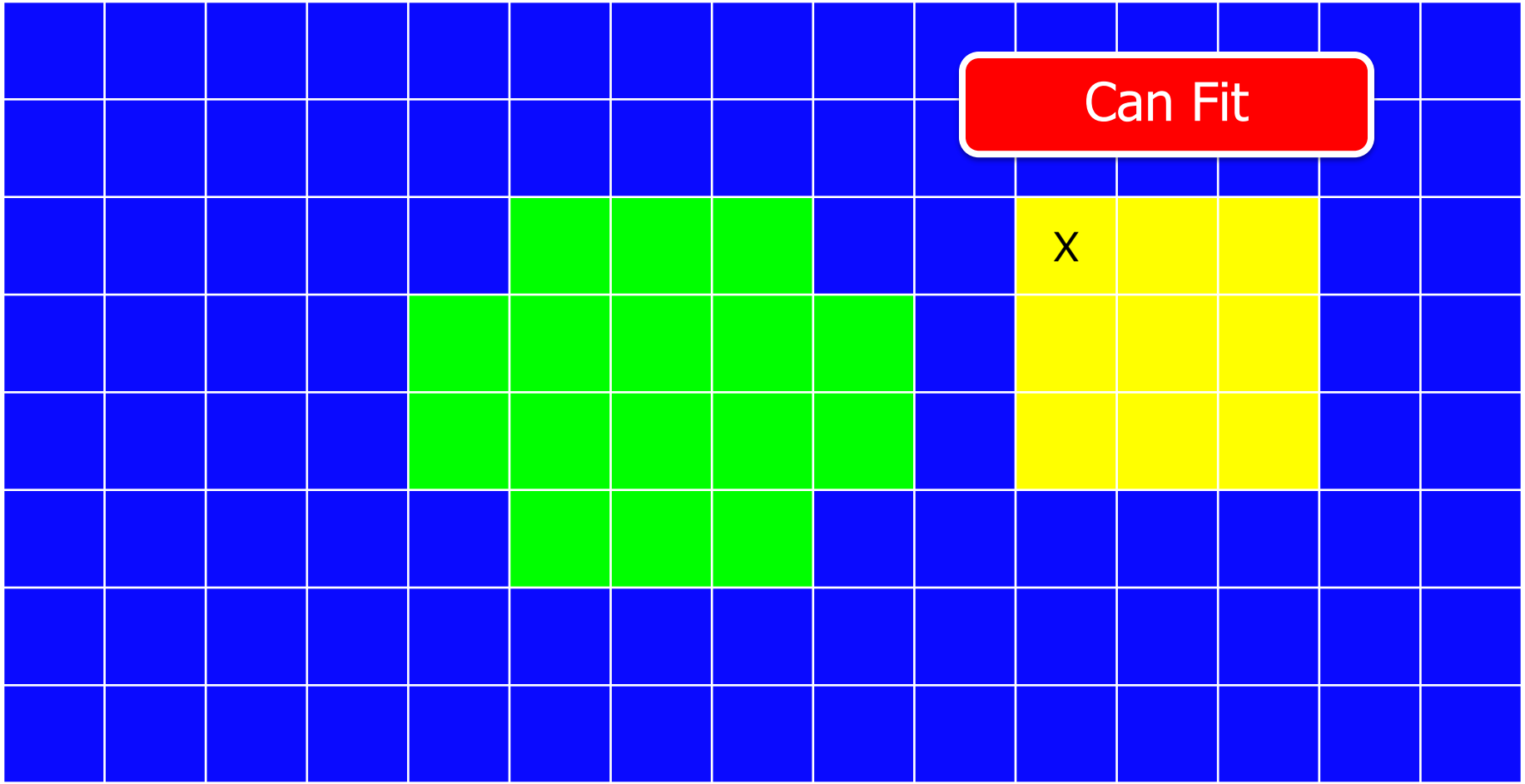
1. Object can only occupy a region if all tiles underneath are the same color
2. Object can only move to a space if all of the new region is also the same color
3. An object must be entirely within map
4. A diagonal move cannot allow the object to “jump over” a tile of another color

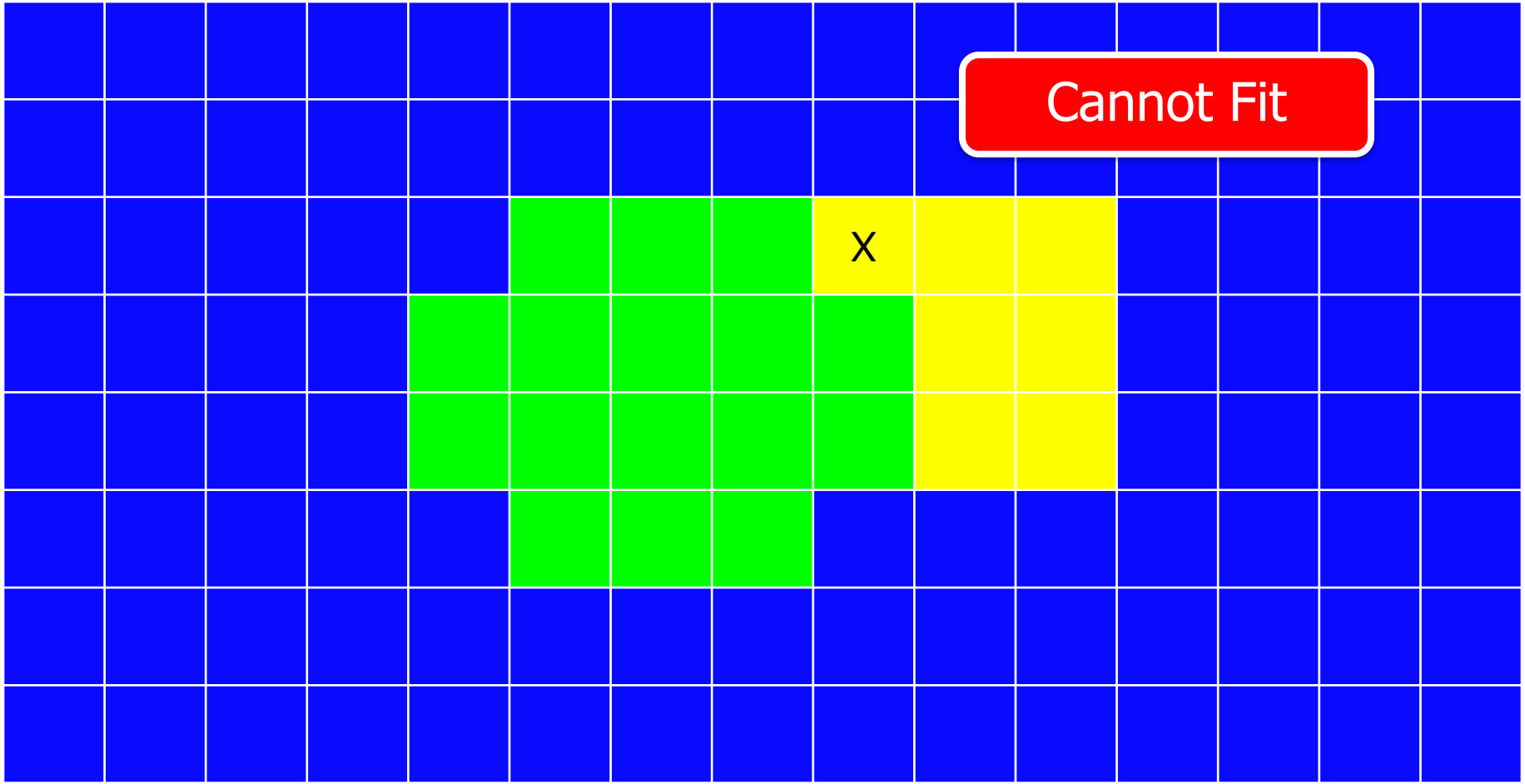

Object Sizes

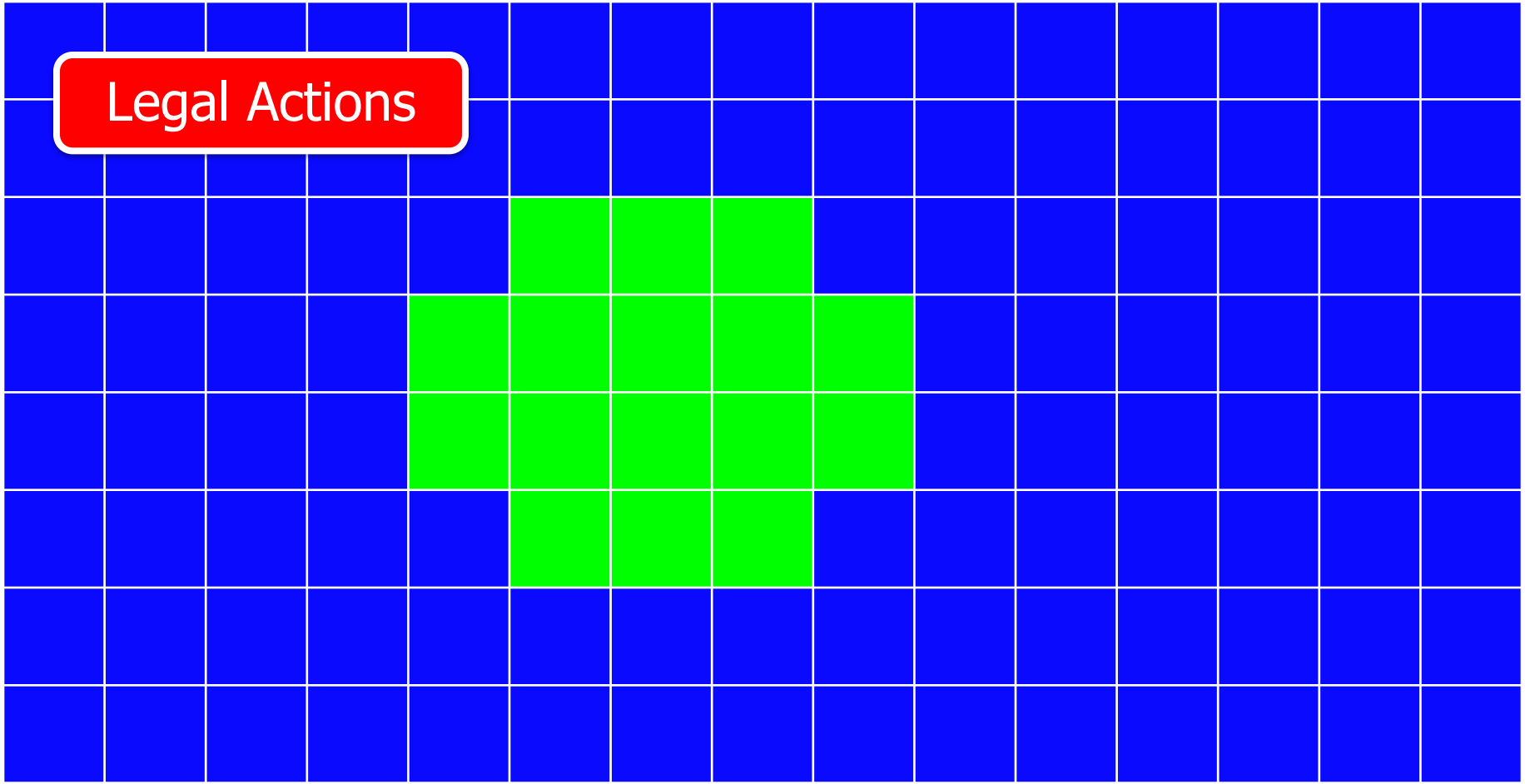

Object Size 1
X

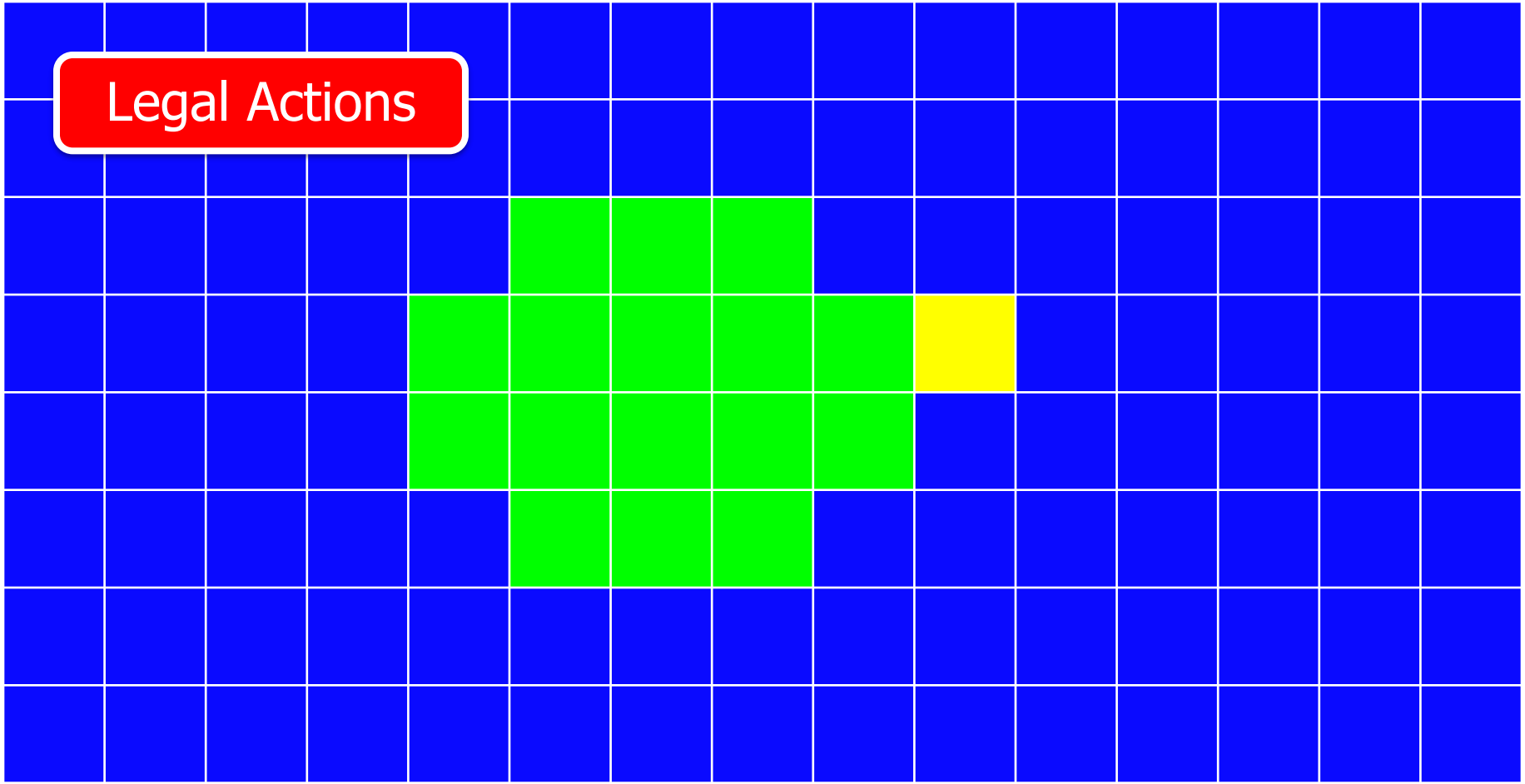

Object Size 2
X

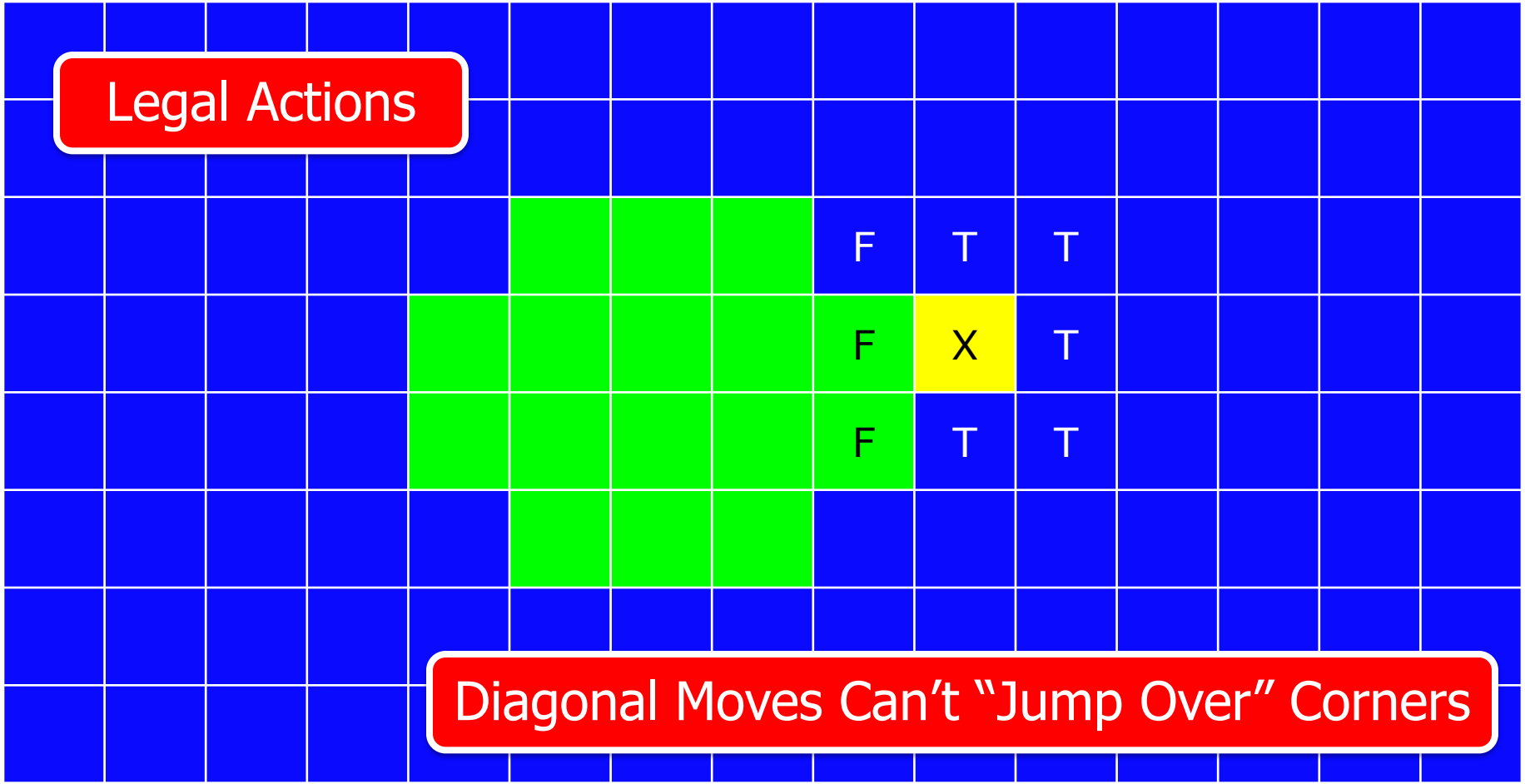

Object Size 3
X

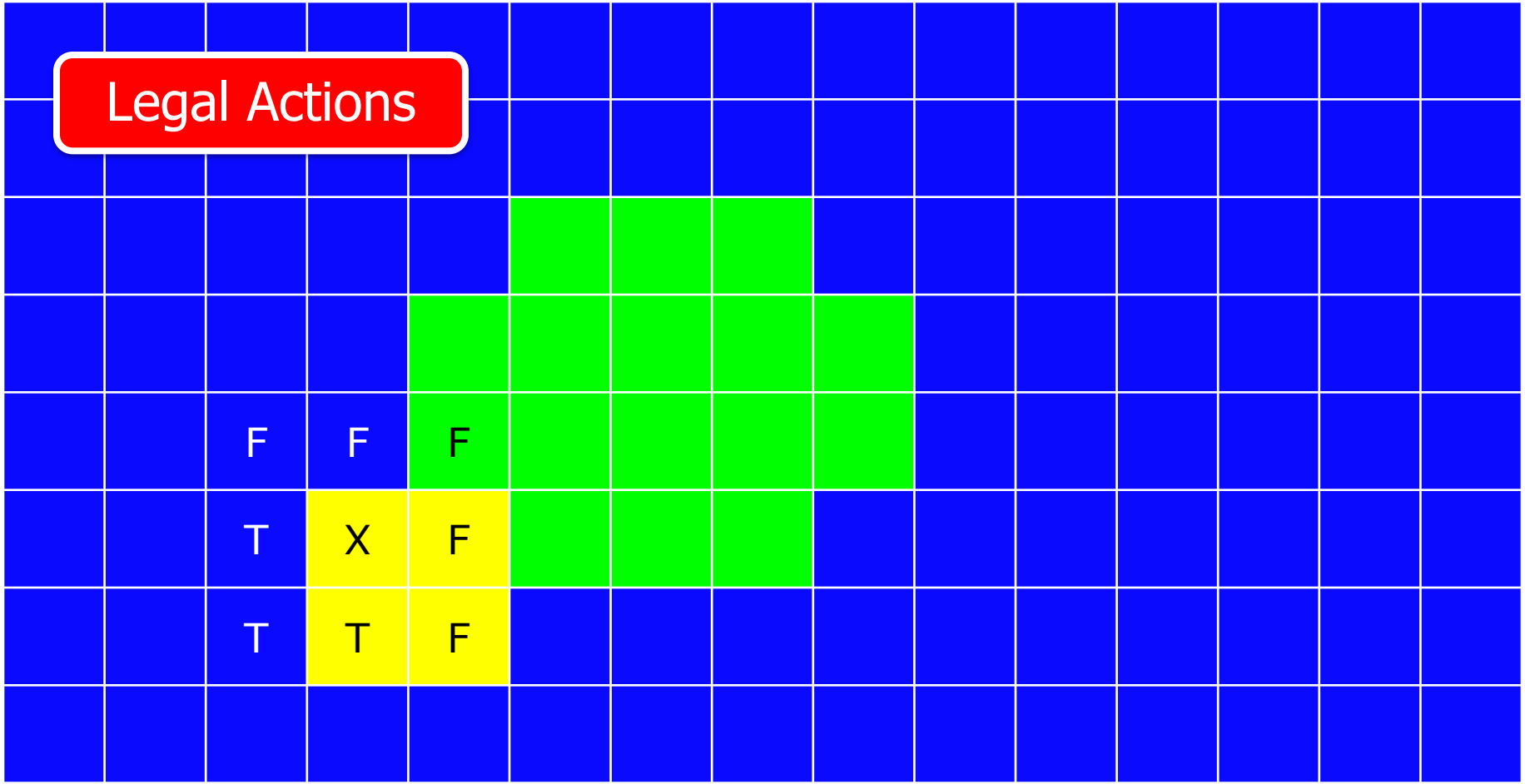

Can Fit
X

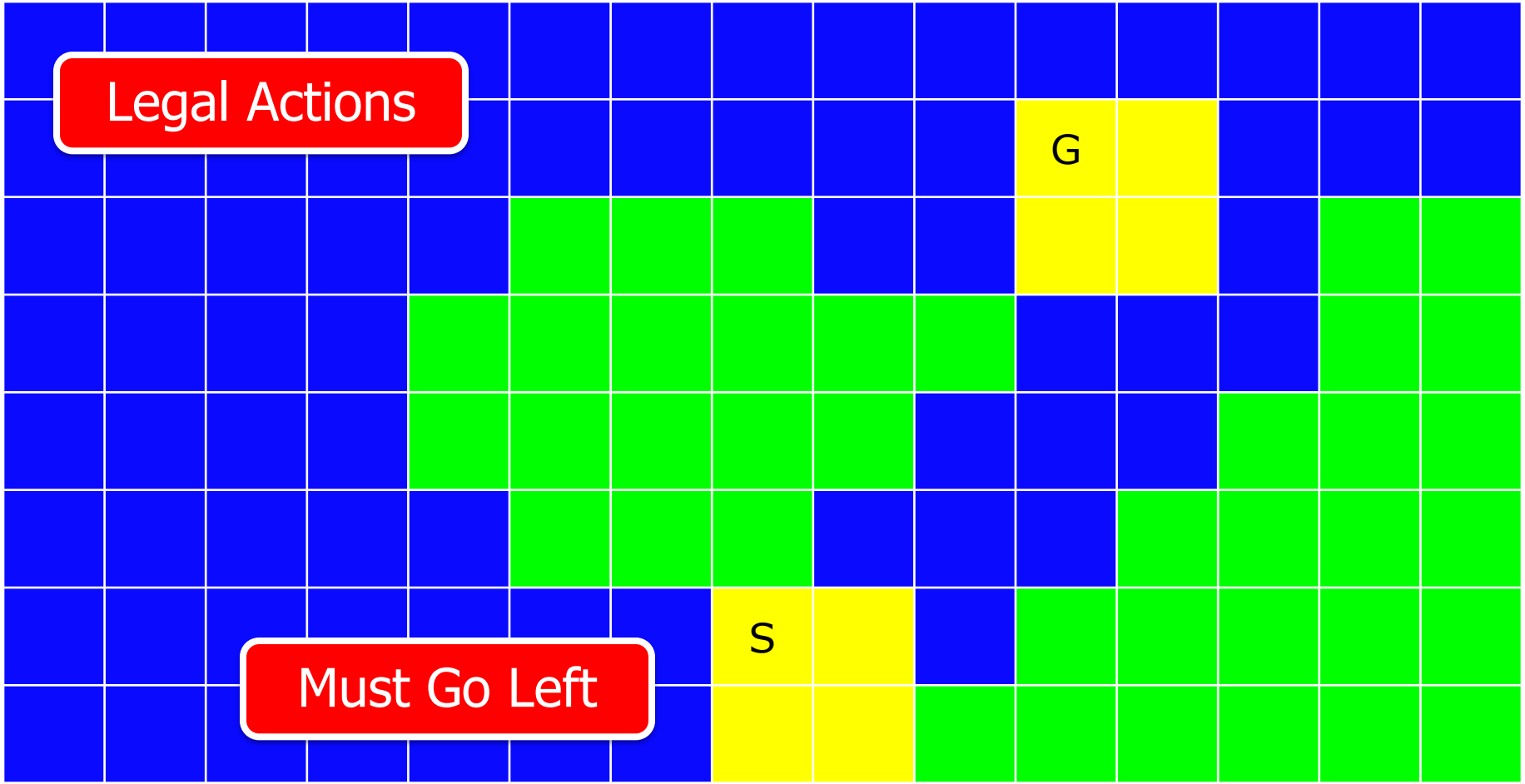

Cannot Fit
X

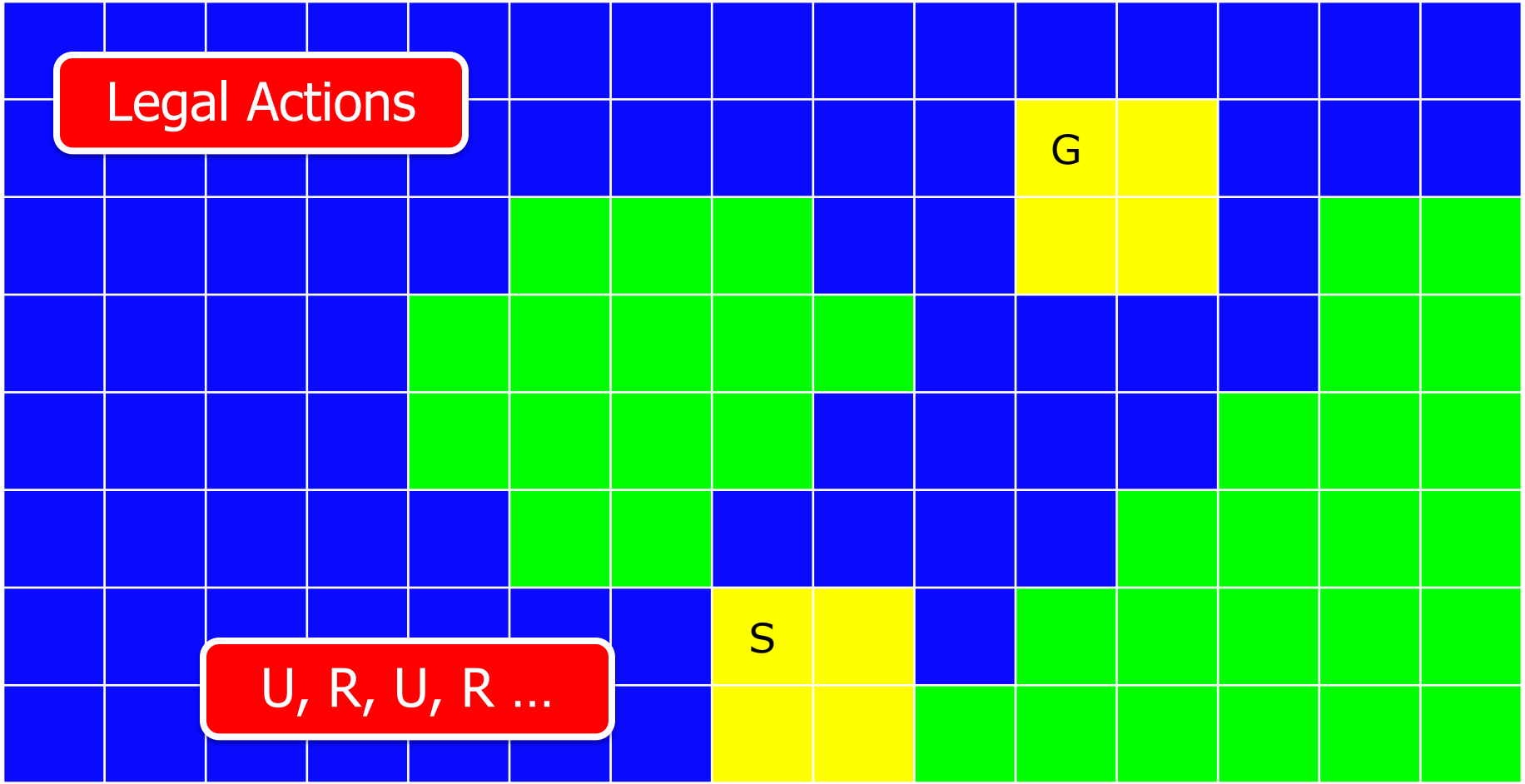

Legal Actions

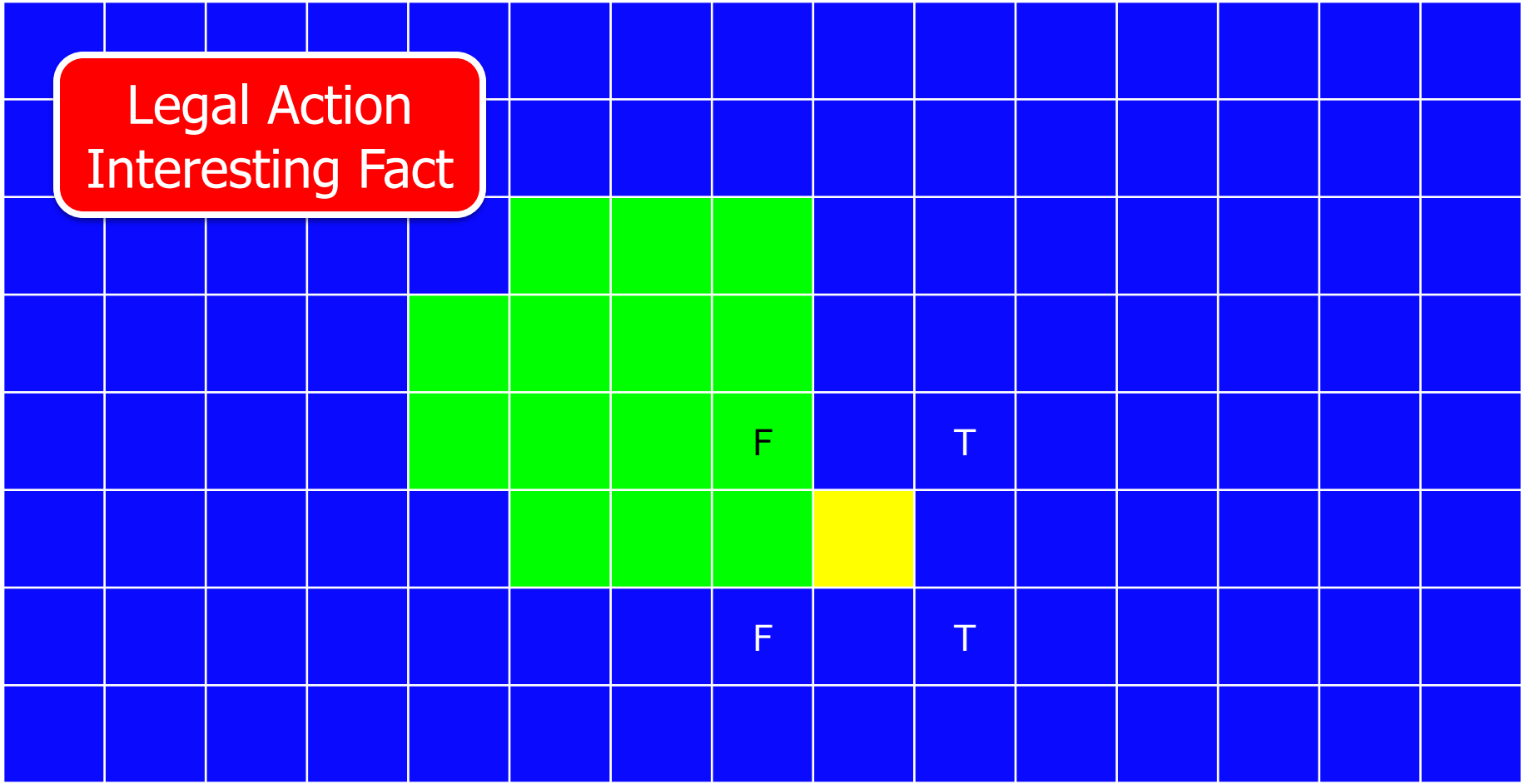

Legal Actions

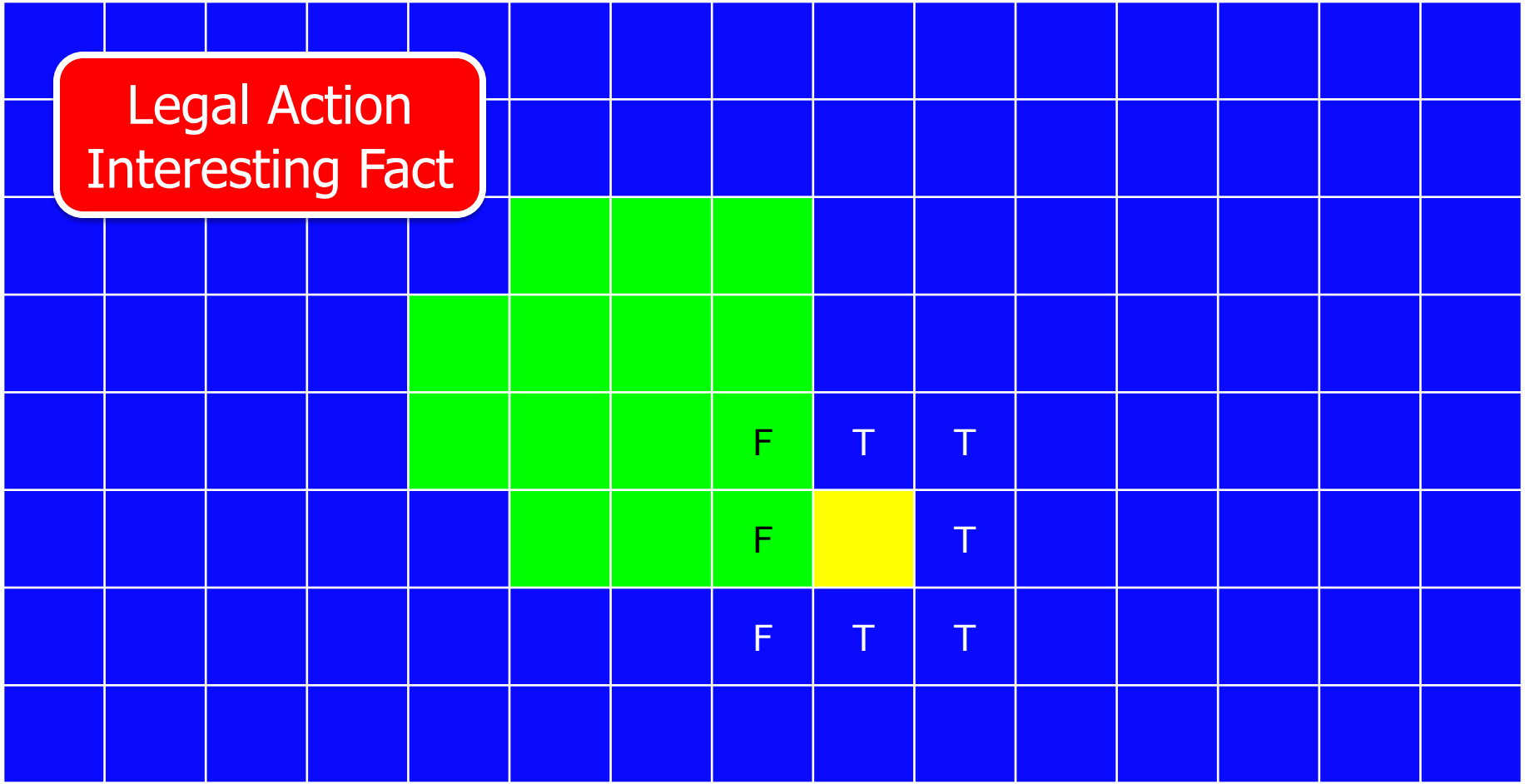

Legal Actions
F
T
T
F
X
T
F
T
T
Diagonal Moves Can’t “Jump Over” Corners

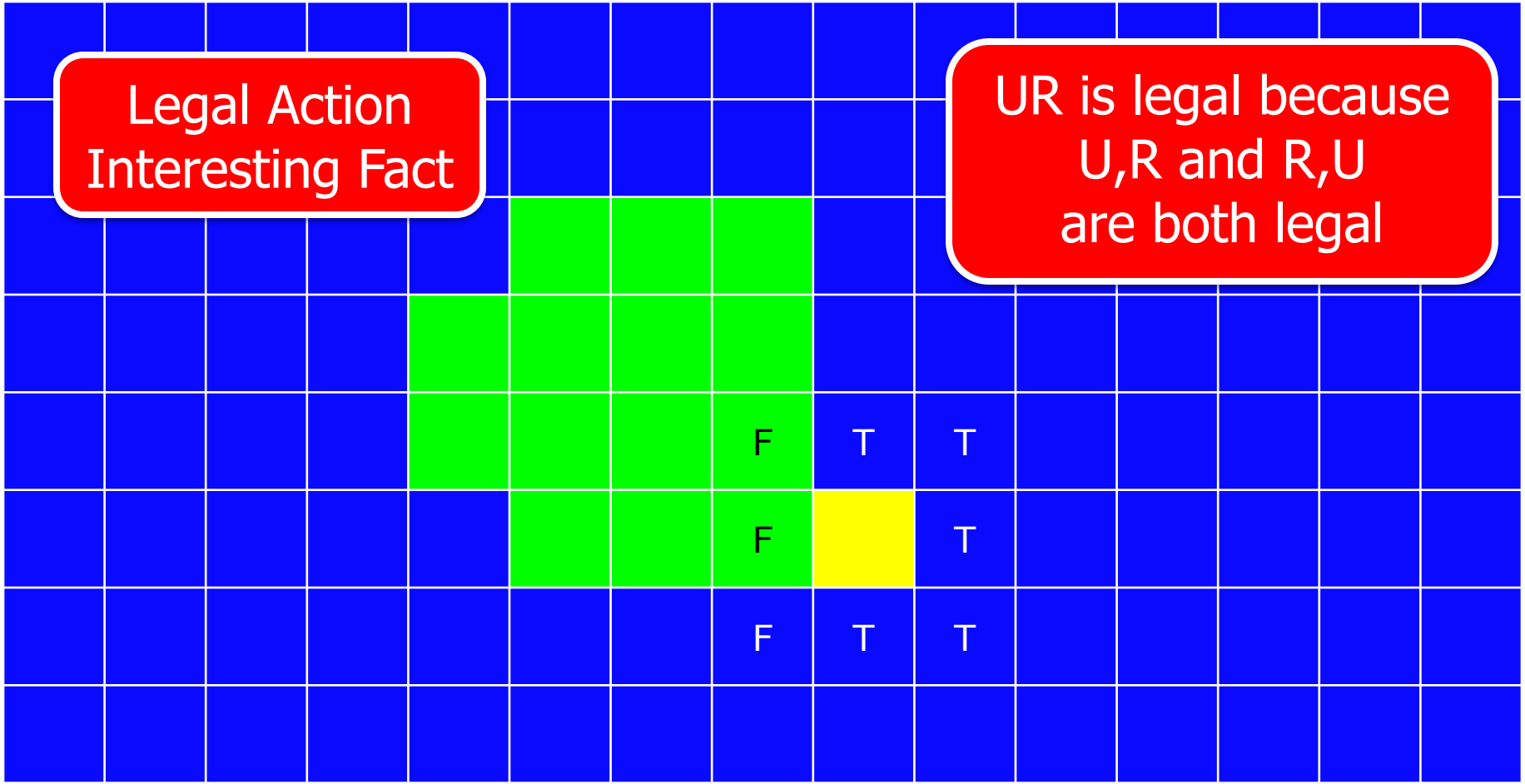

Legal Actions
F
F
F
T
X
F
T
T
F

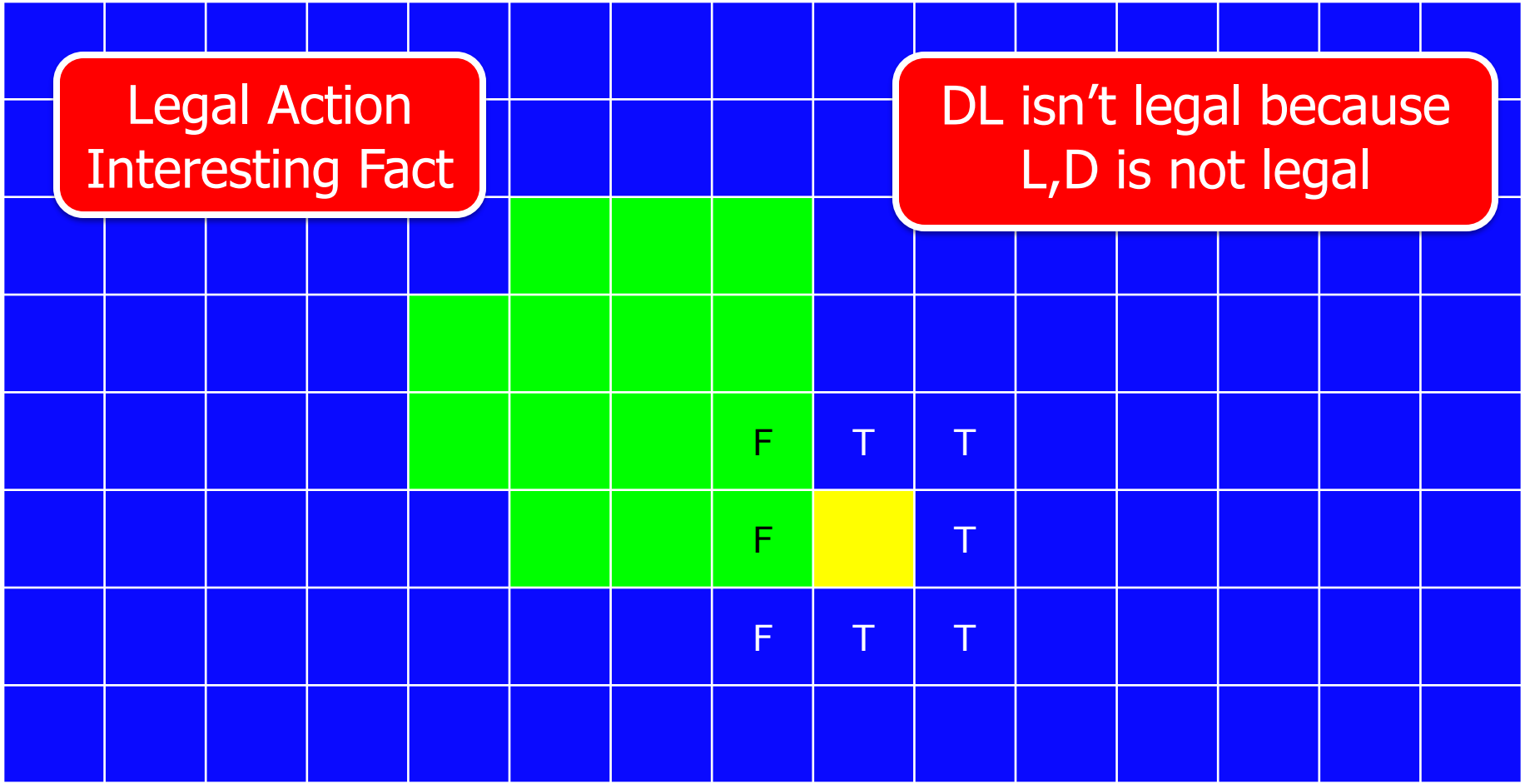

Legal Actions
G
S
Must Go Left

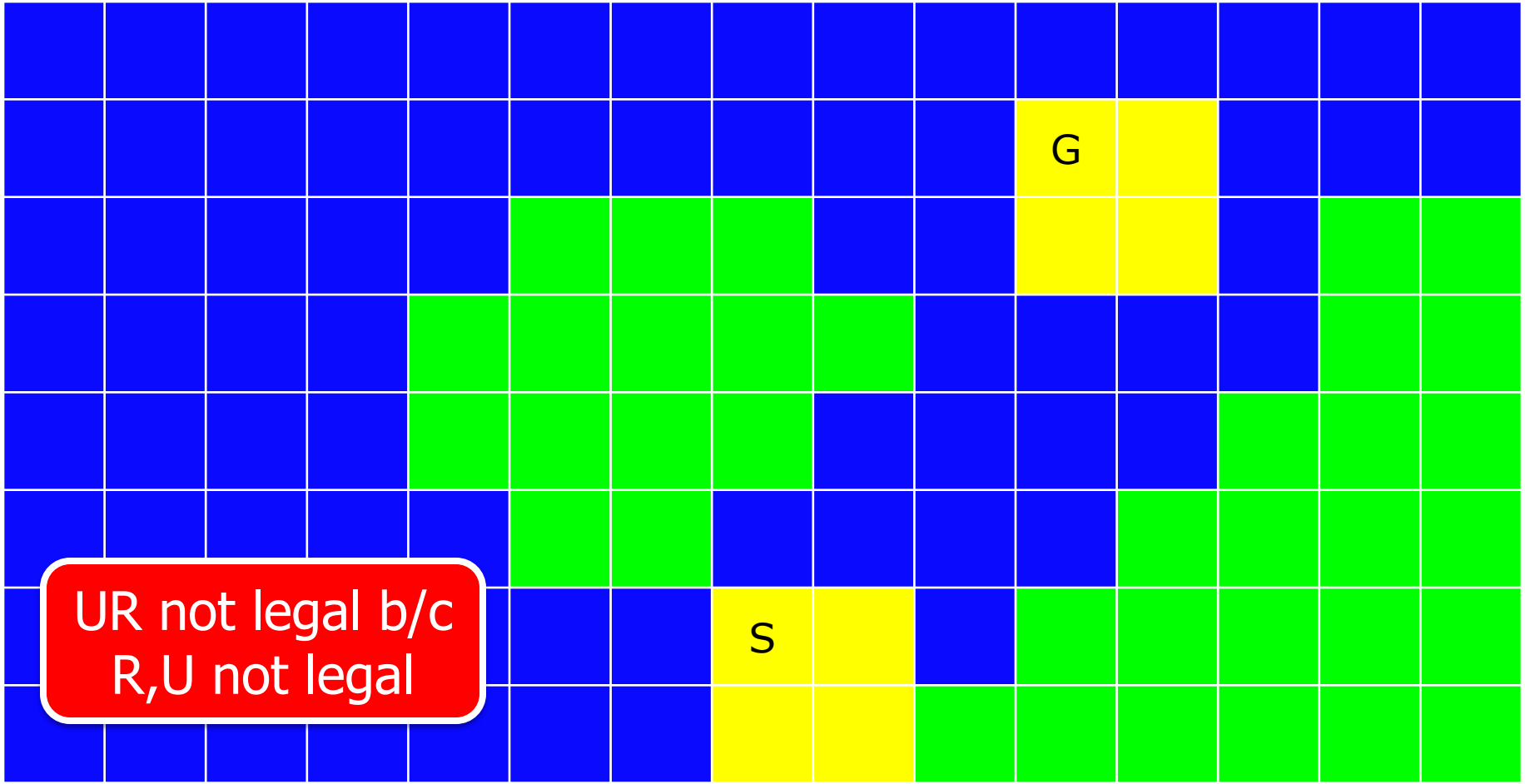

Legal Actions
G
S
U, R, U, R …

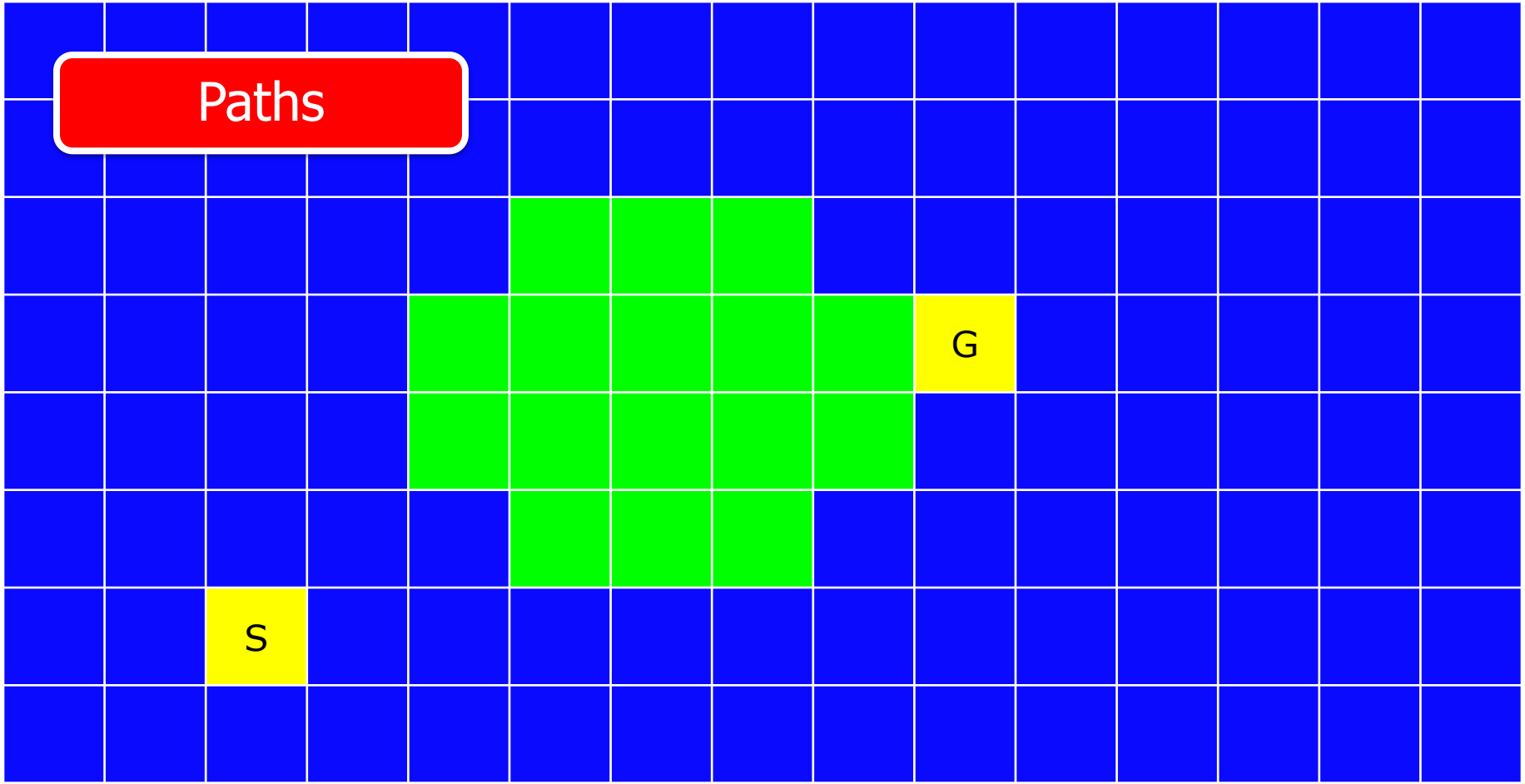

Legal Action
Interesting Fact
F
T
F
T

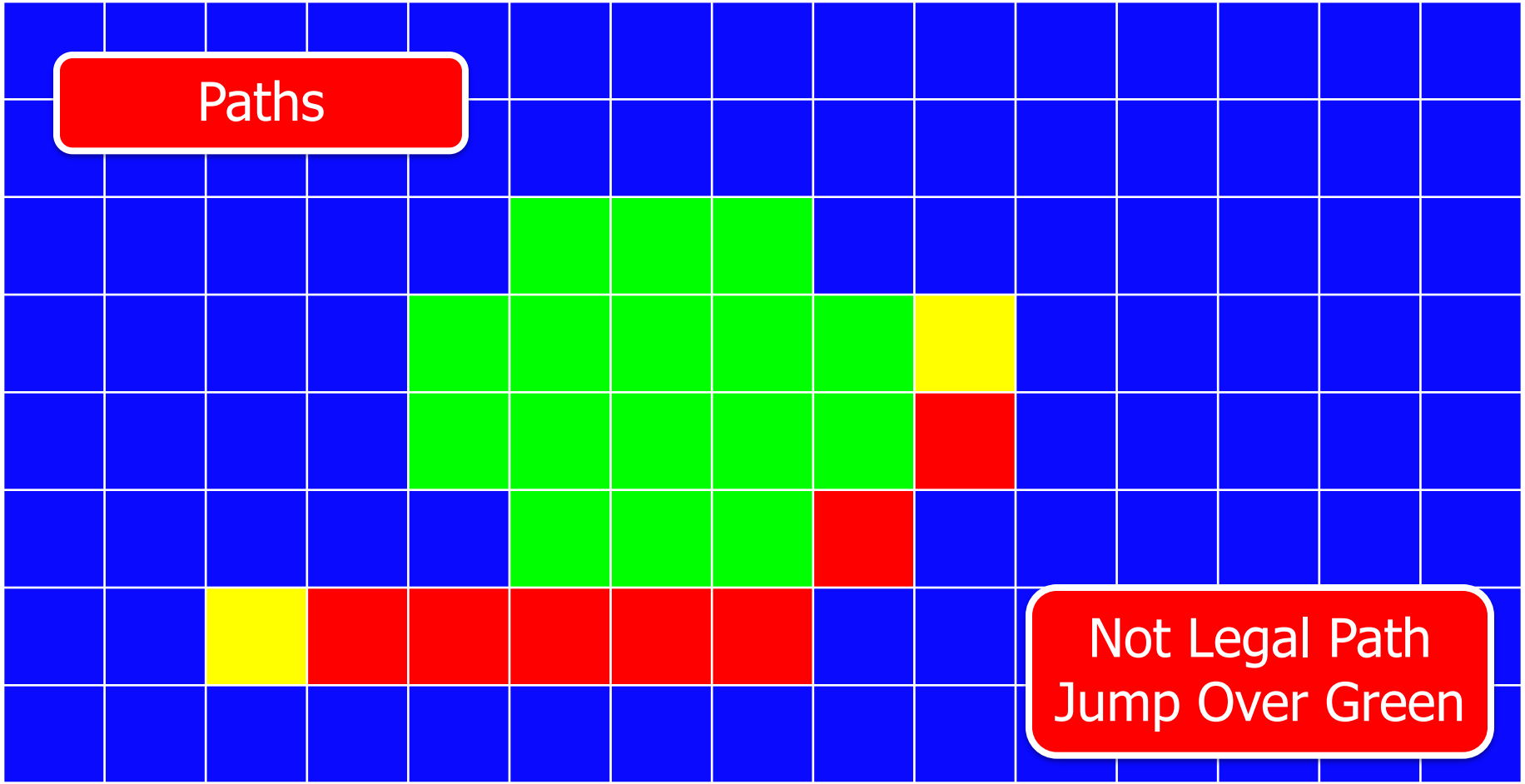

Legal Action
Interesting Fact
F
T
T
F
T
F
T
T

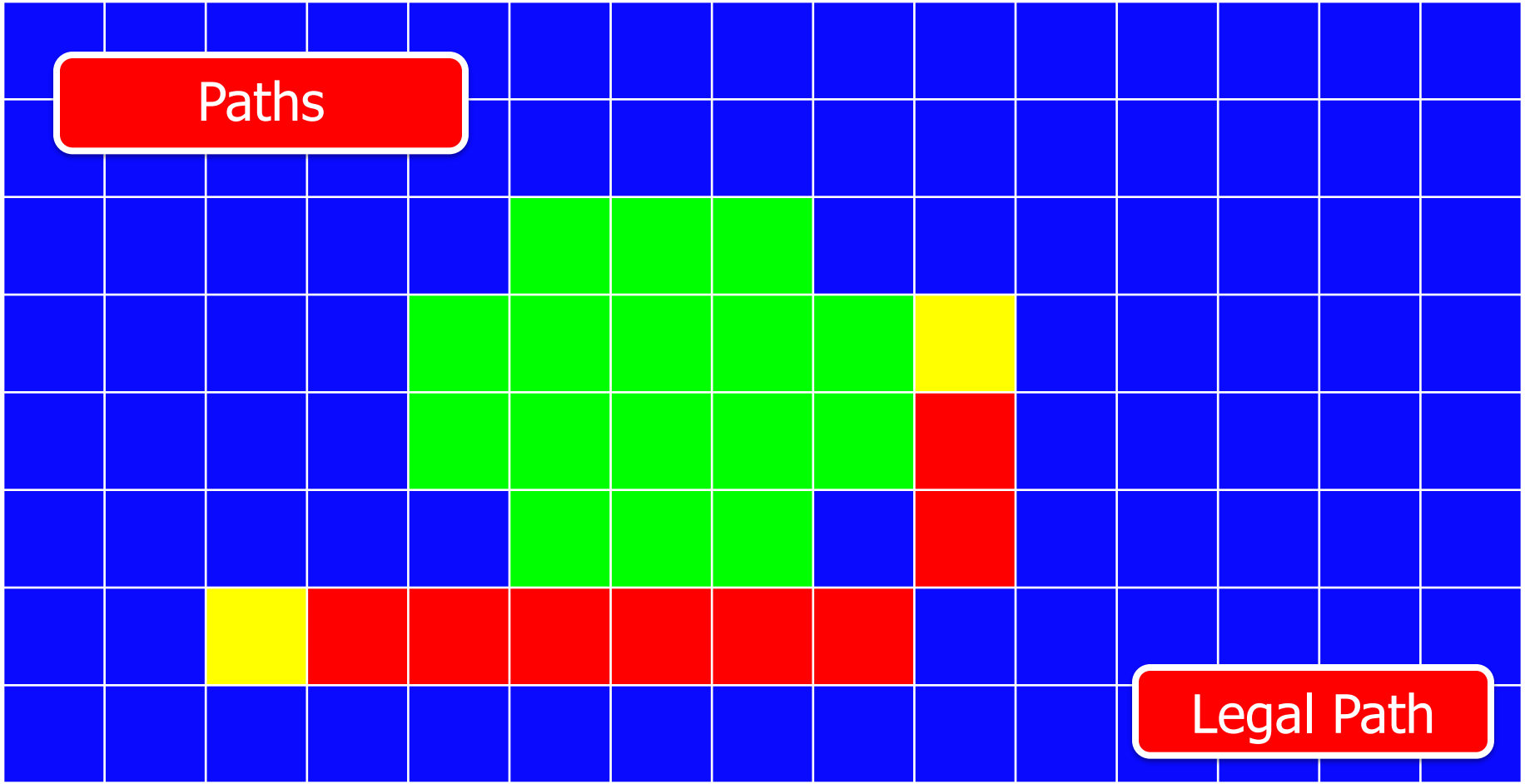

Legal Action
Interesting Fact
UR is legal because
U,R and R,U
are both legal
F
T
T
F
T
F
T
T

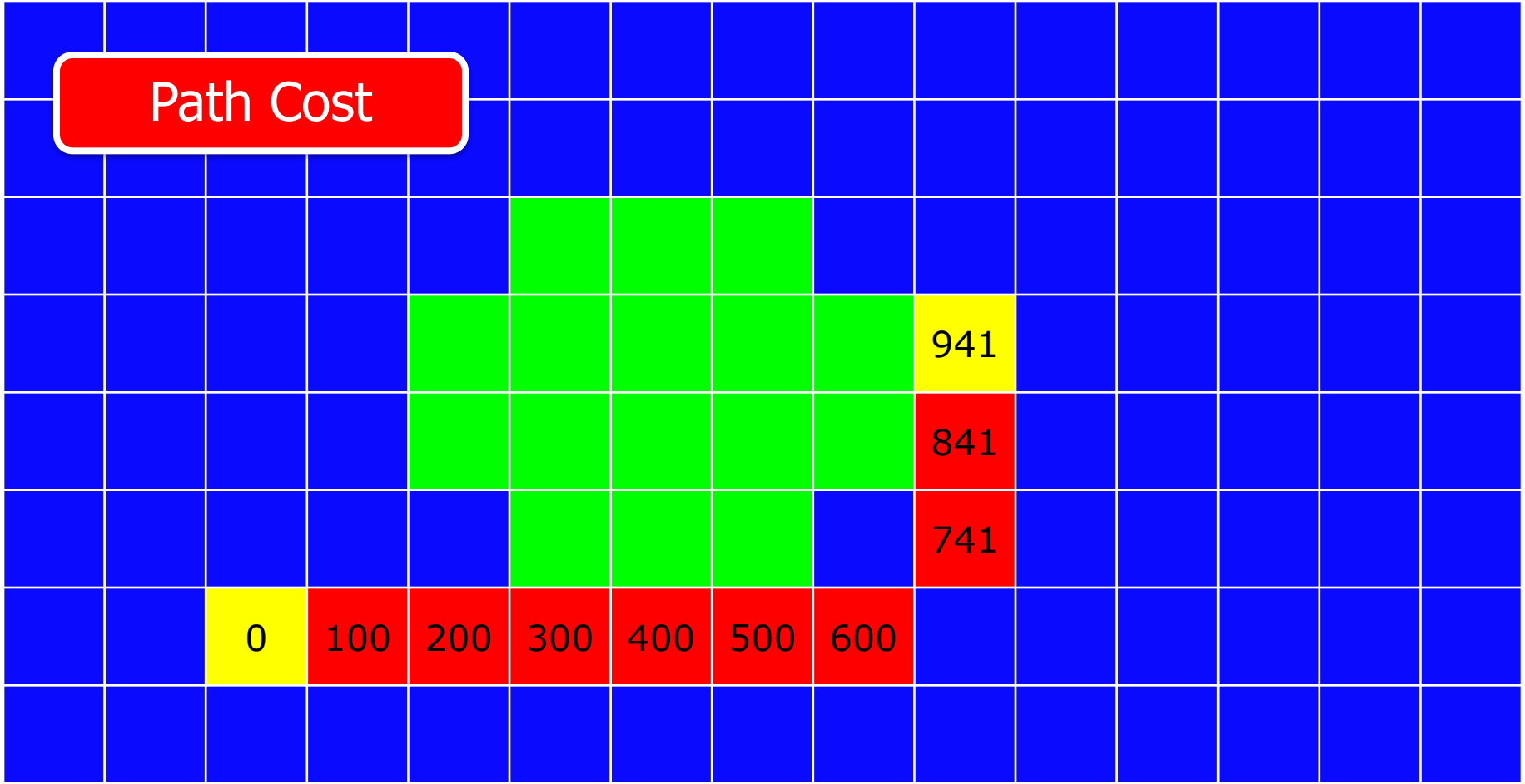

Legal Action
Interesting Fact
DL isn't legal because
L,D is not legal
F
T
T
F
T
F
T
T


G
S
UR not legal b/c
R,U not legal


Paths
G
S

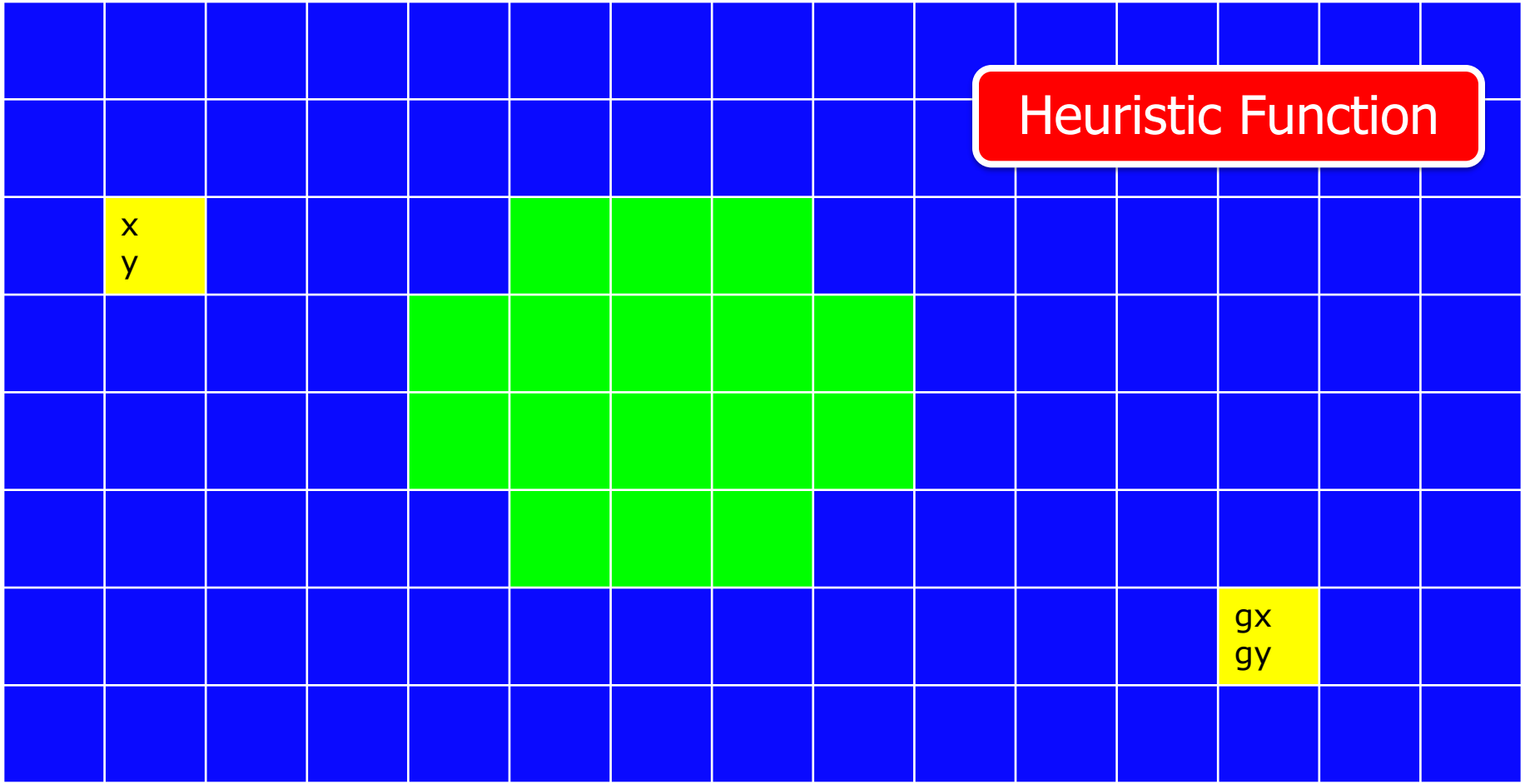

Paths
Not Legal Path
Jump Over Green

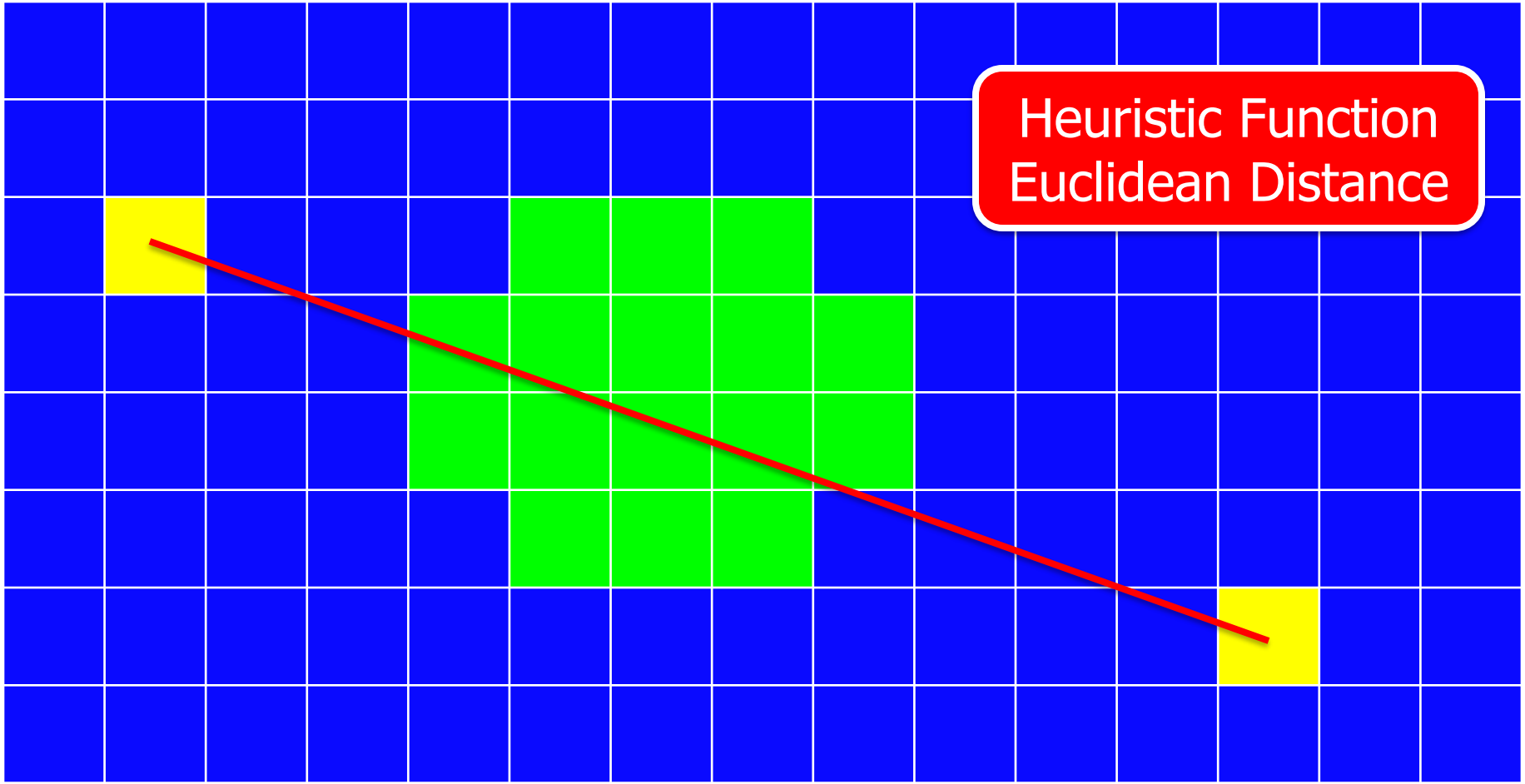

Paths
Legal Path

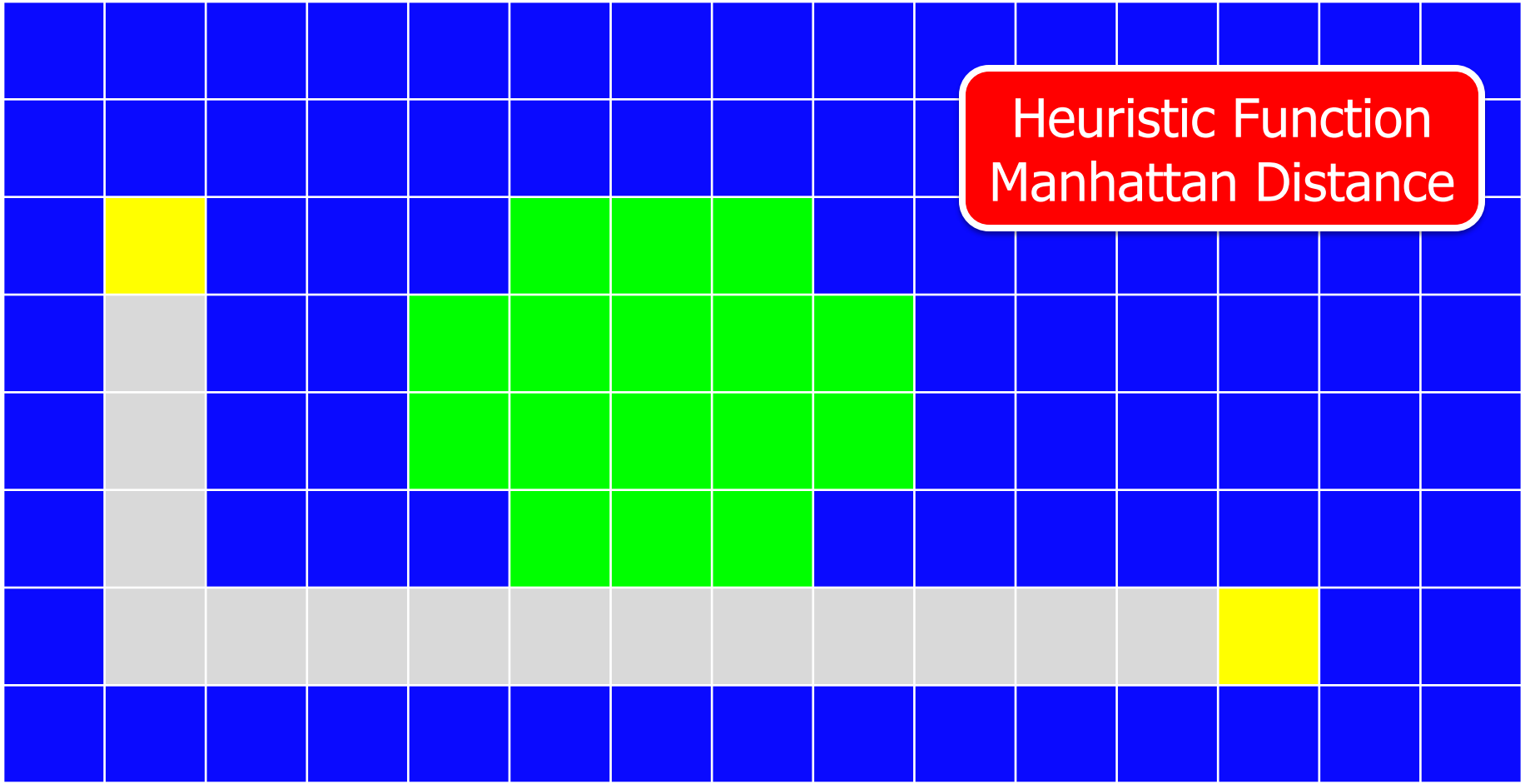

Path Cost
941
841
741
0
100
200
300
400
500
600

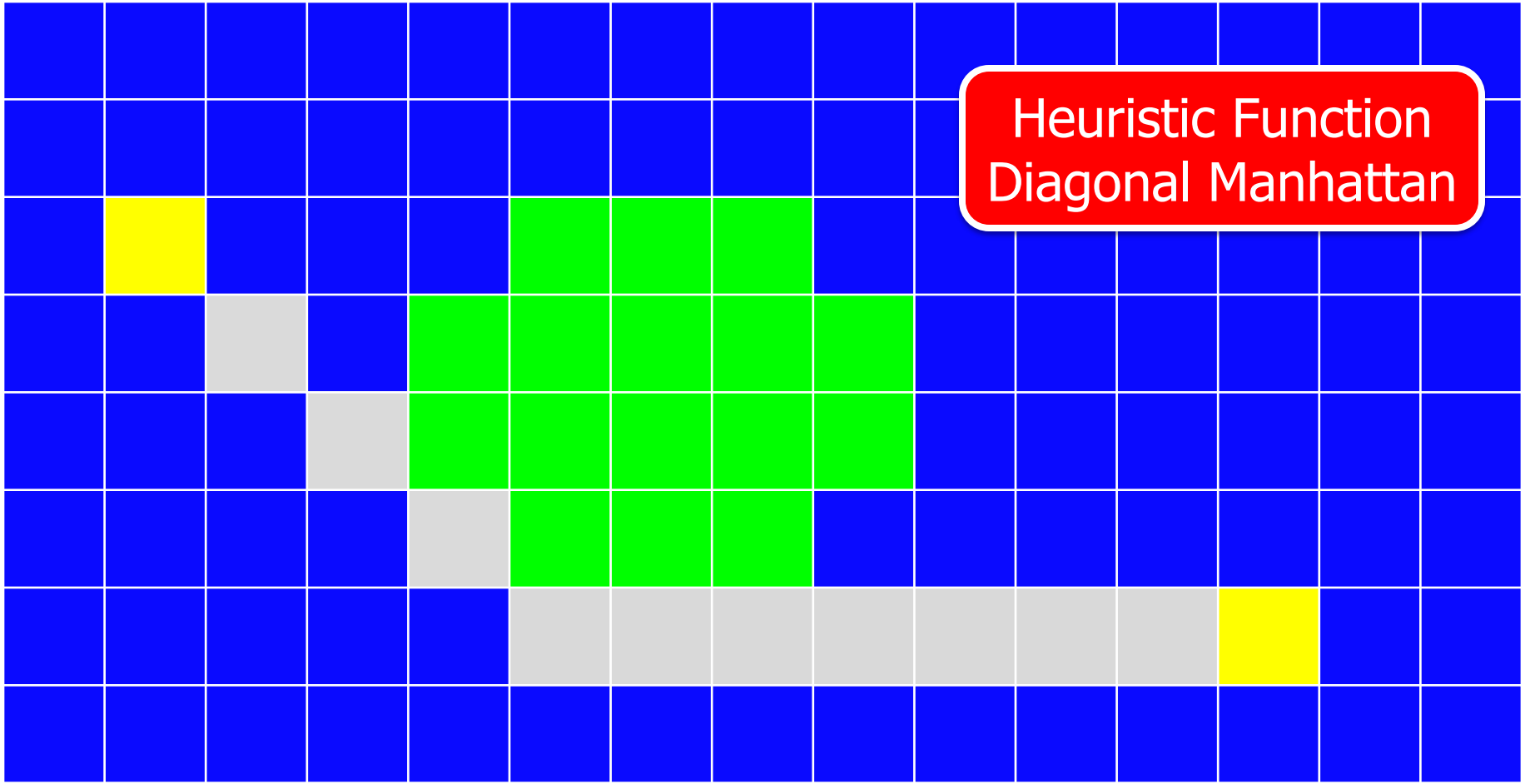
Heuristic Functions

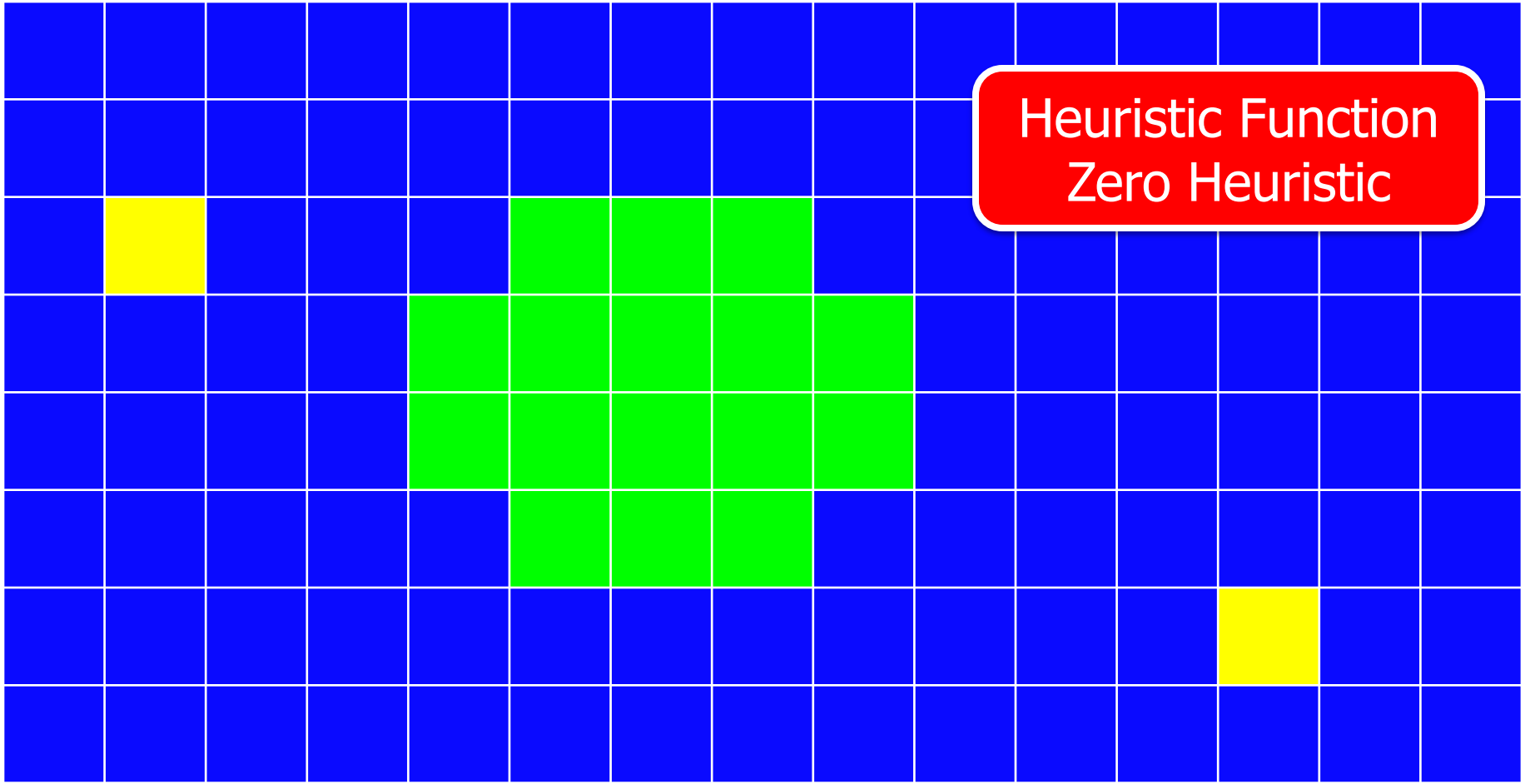
- Guess how much distance remaining
- A2: estimateCost(x, y, gx, gy)
- Four different heuristics
  - ‘dist’ - Euclidean 2D Distance (Pythagorus)
  - ‘card’ – Cardinal Manhattan (4 direction)
  - ‘diag’ – Diagonal Manhattan (8 direction)
  - ‘zero’ – Return Zero (literally)


Heuristic Function
x
y
gx
gy


Heuristic Function
Euclidean Distance

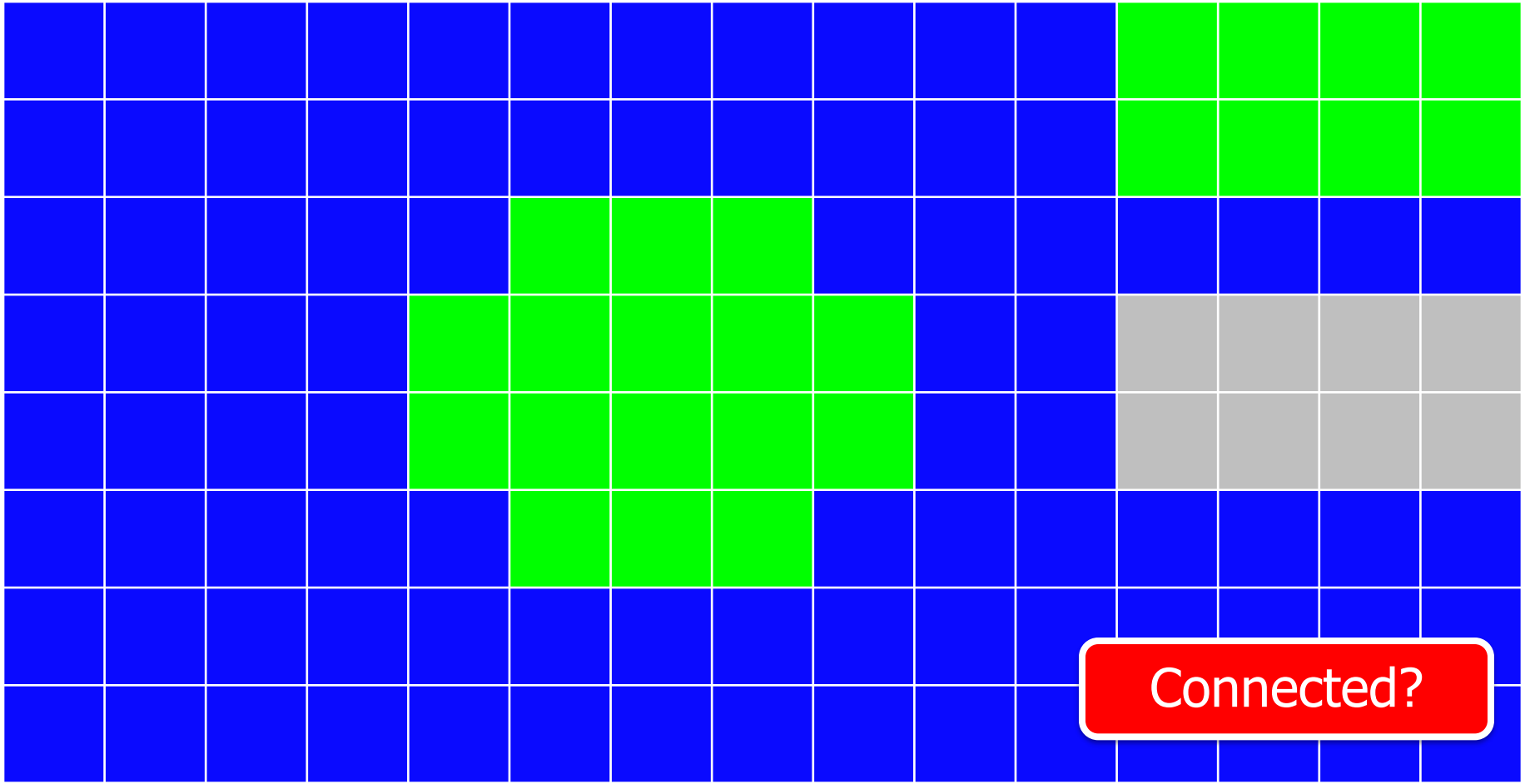

Heuristic Function
Manhattan Distance

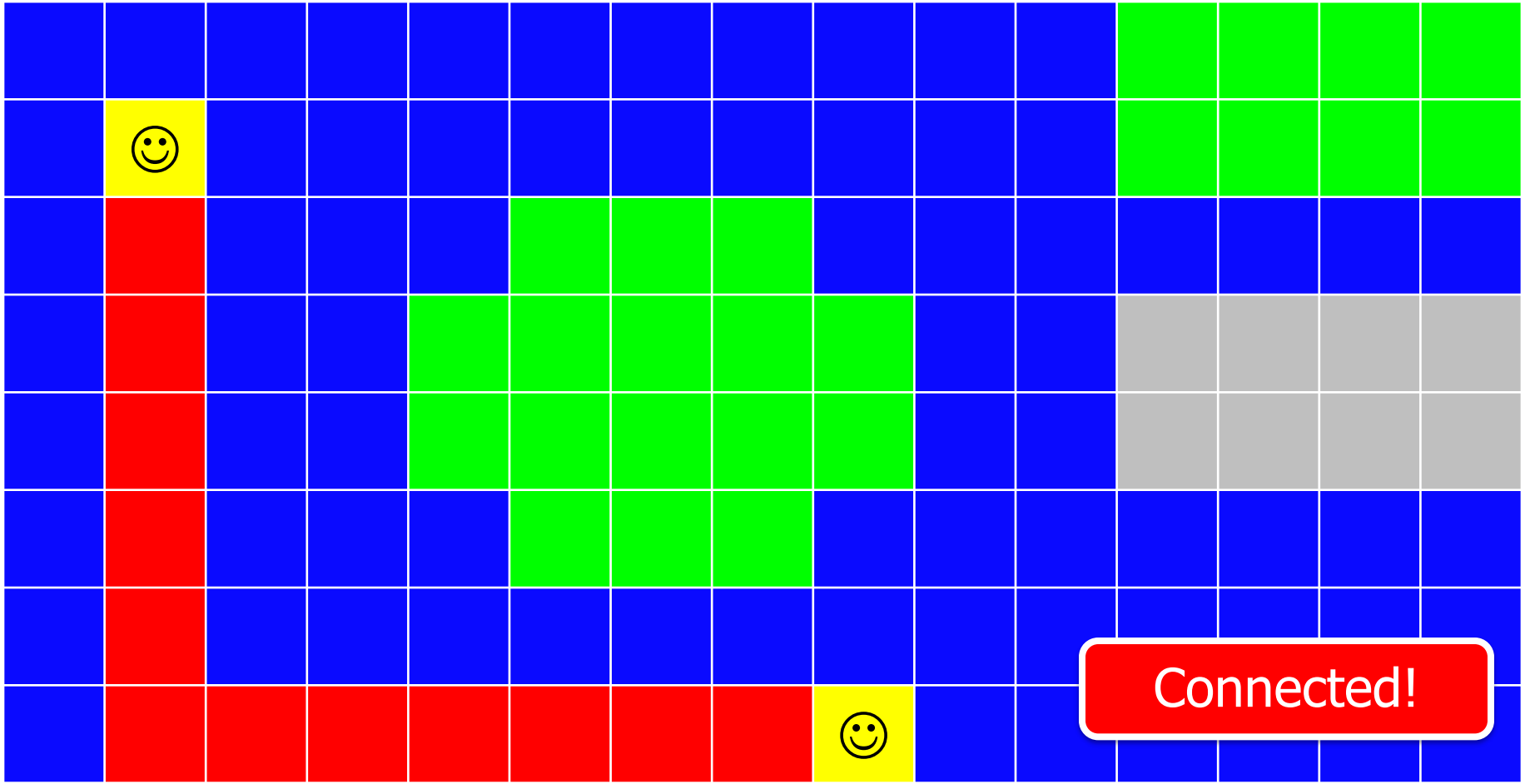

Heuristic Function
Diagonal Manhattan

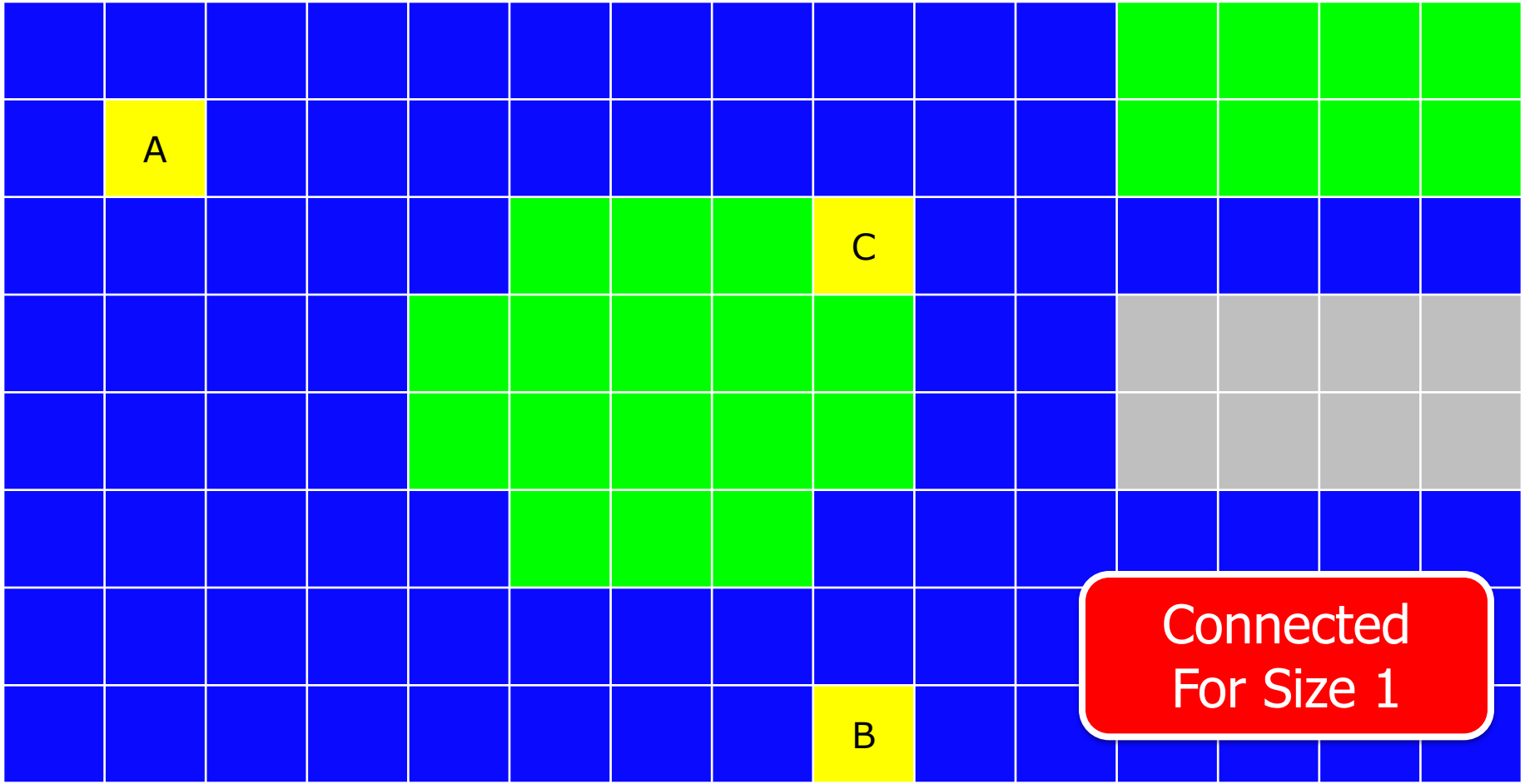

Heuristic Function
Zero Heuristic

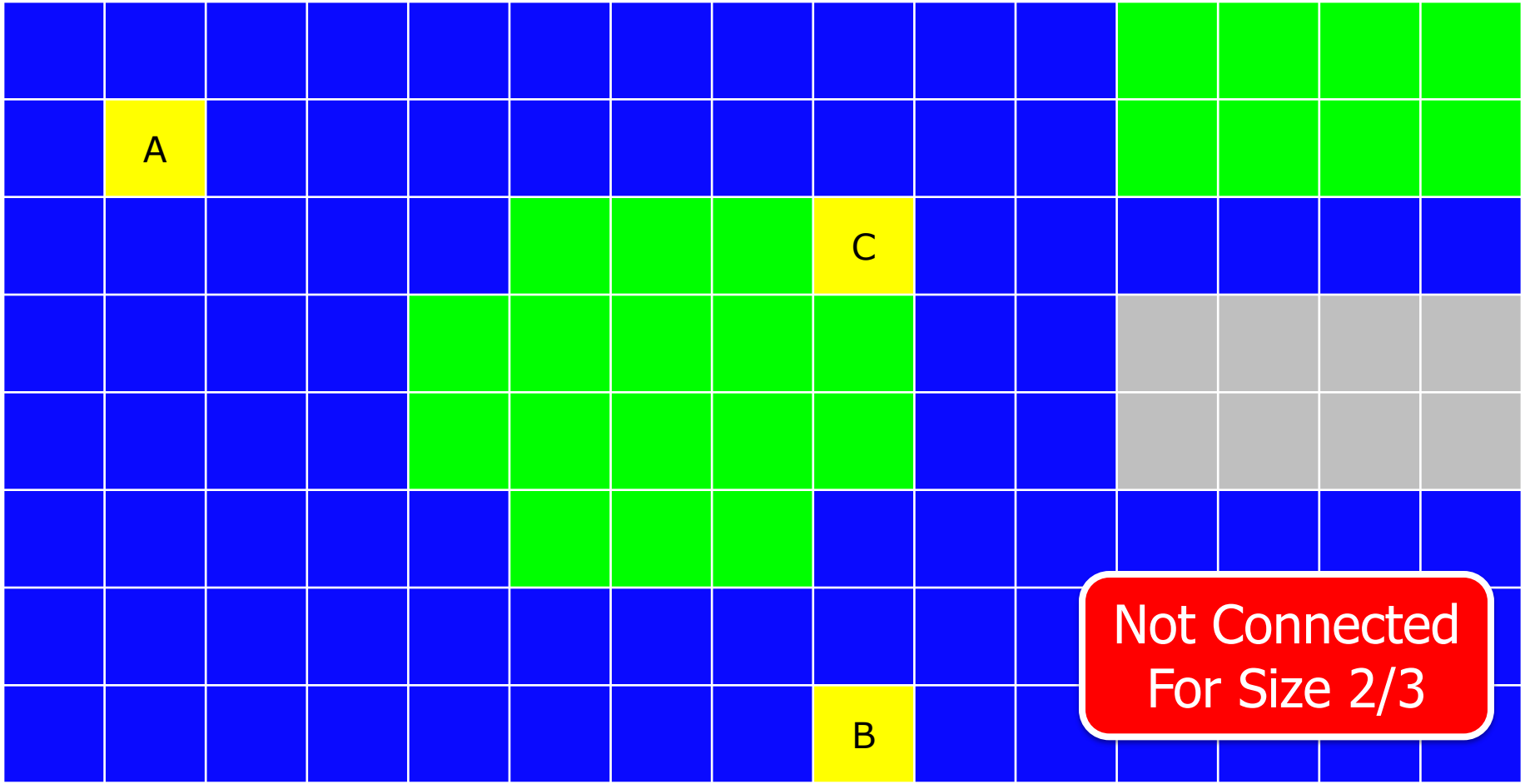
Connected States

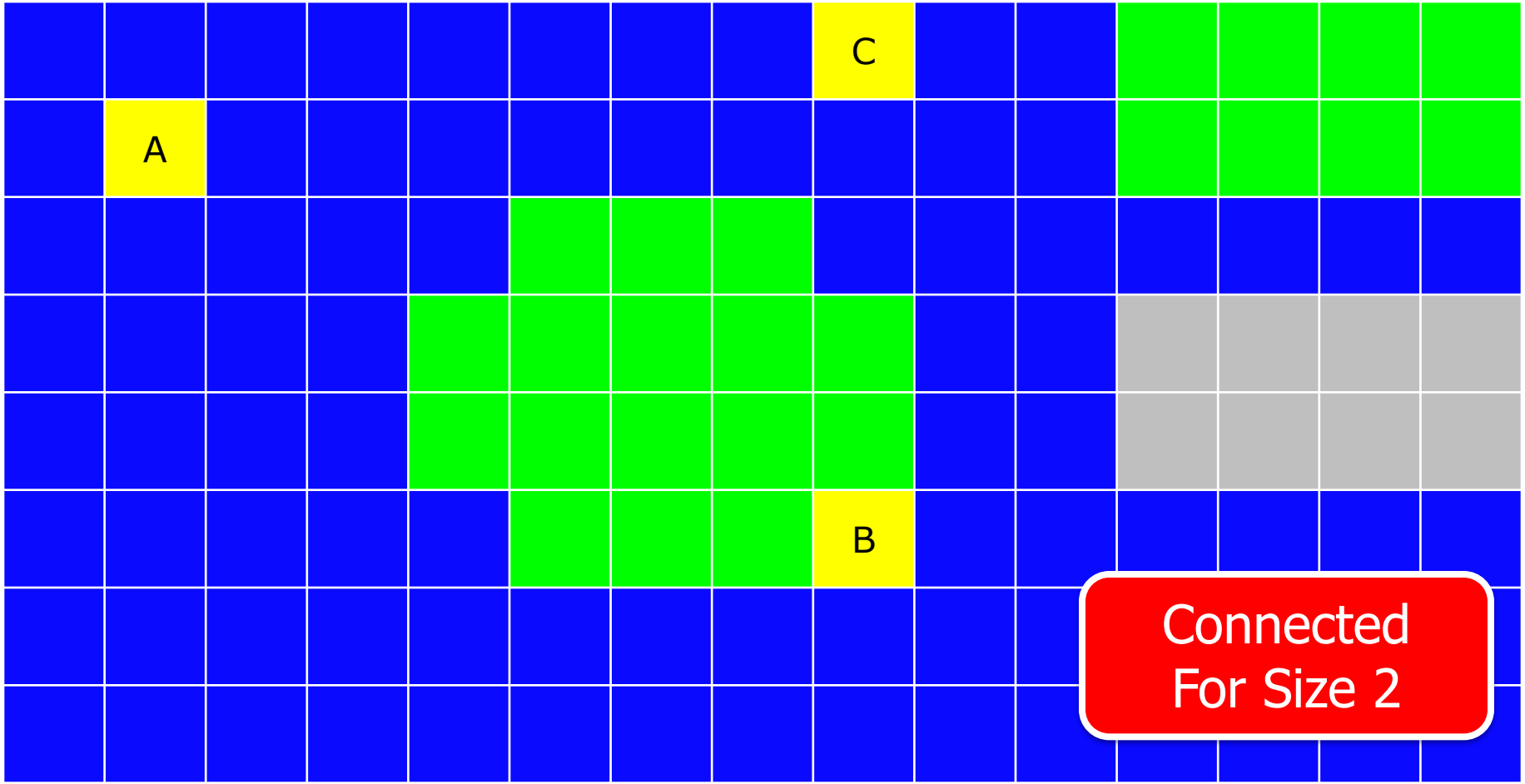
- States are considered ‘connected’ for an object of a given size if a legal path exists between them, no matter how long
- A ‘connected sector’ is the entire set of states that are all connected to each other
- A2: Holding right mouse button highlights all states connected to the clicked state
- Connectivity are transitive: if A is connected to B, and B is connected to C, A is connected to C

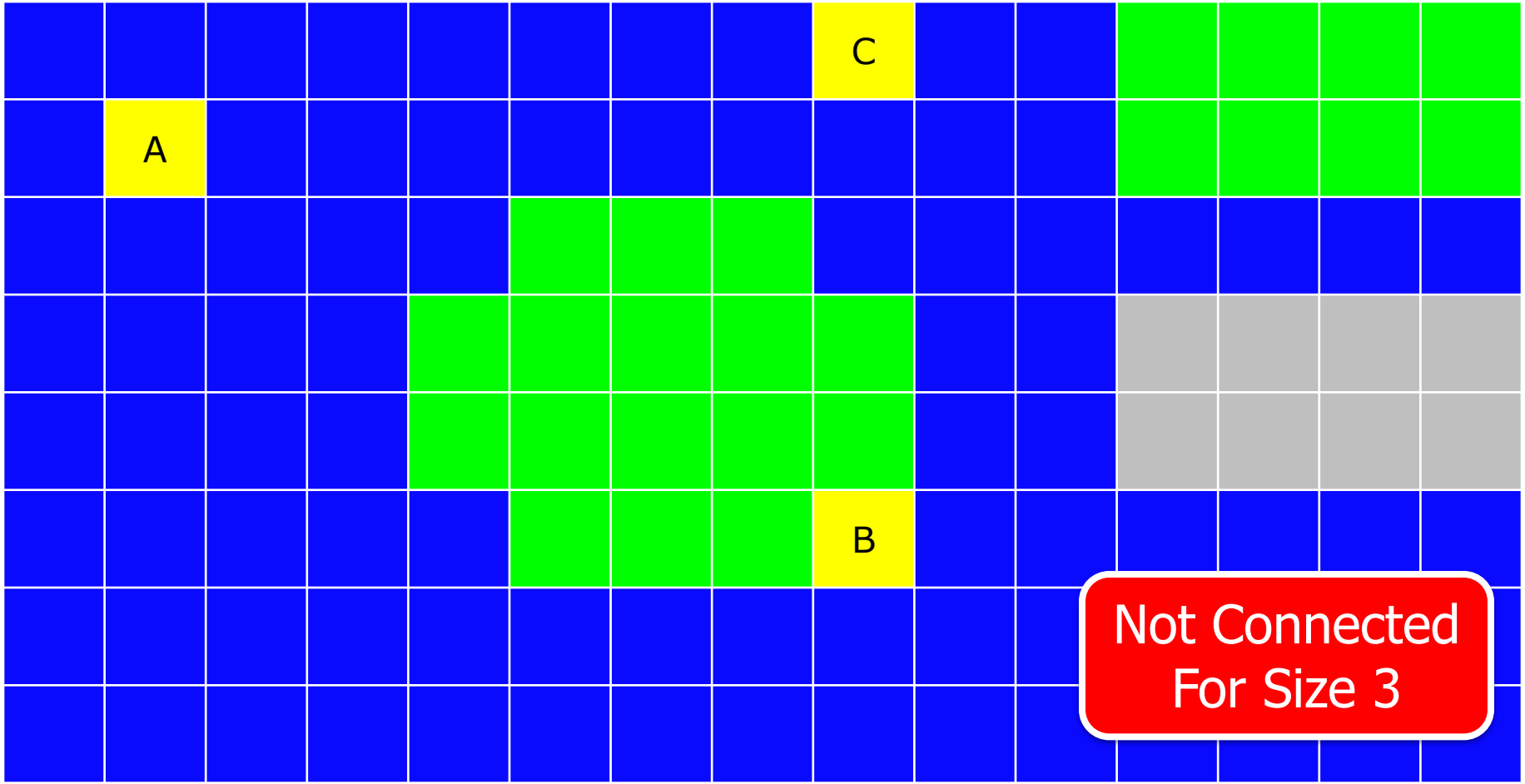

Connected?

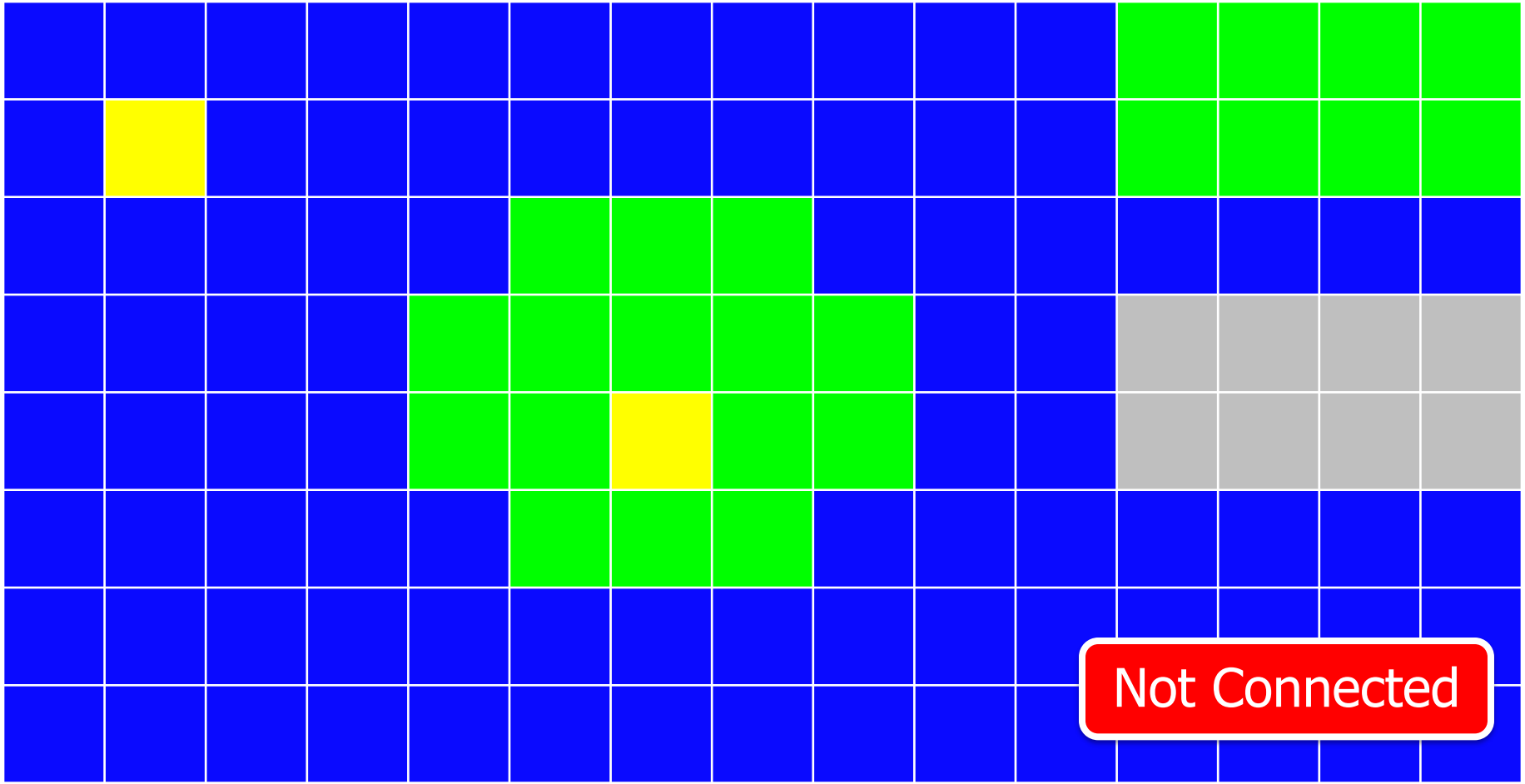

Connected!

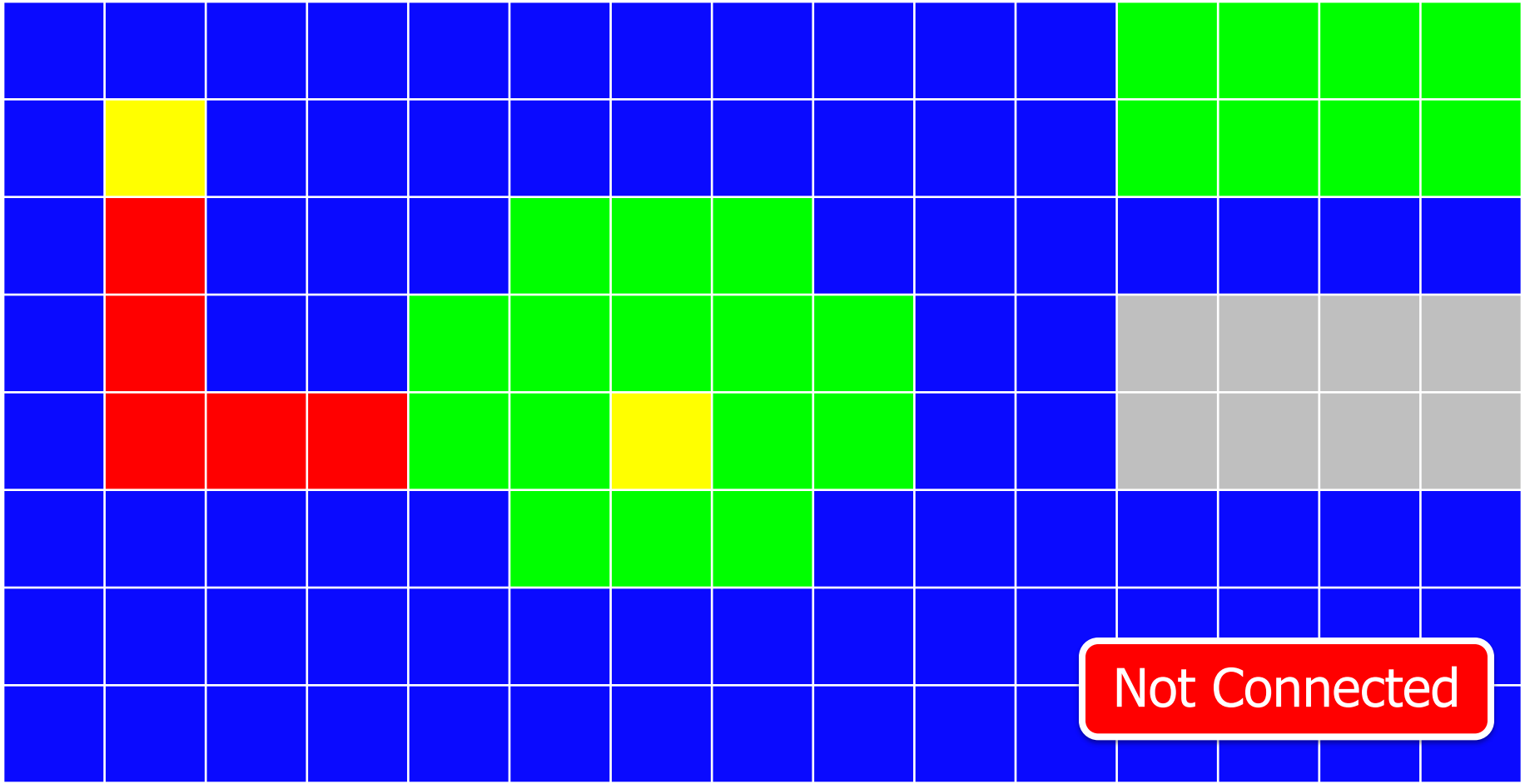

A
C
B
Connected
For Size 1

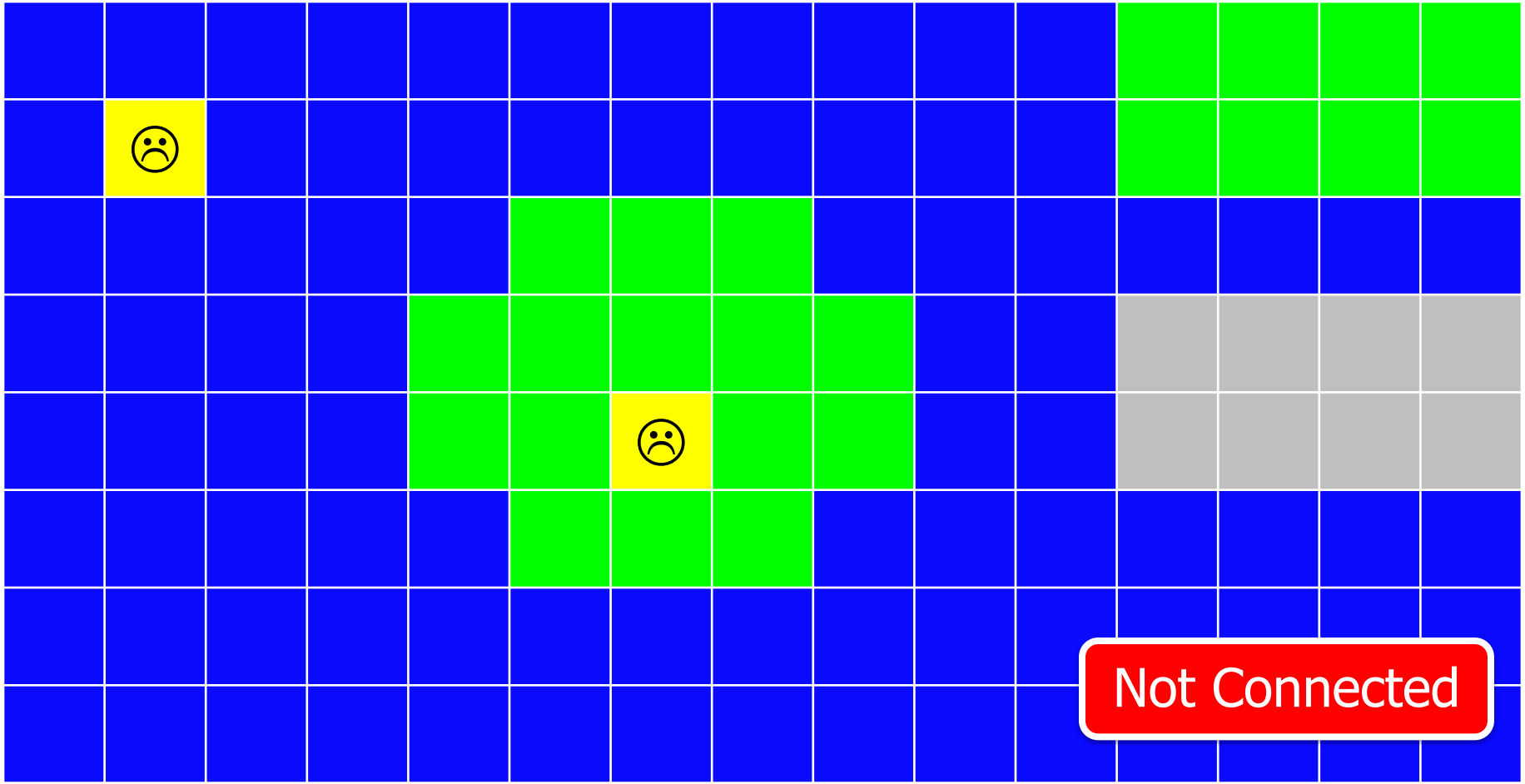

A
C
B
Not Connected
For Size 2/3

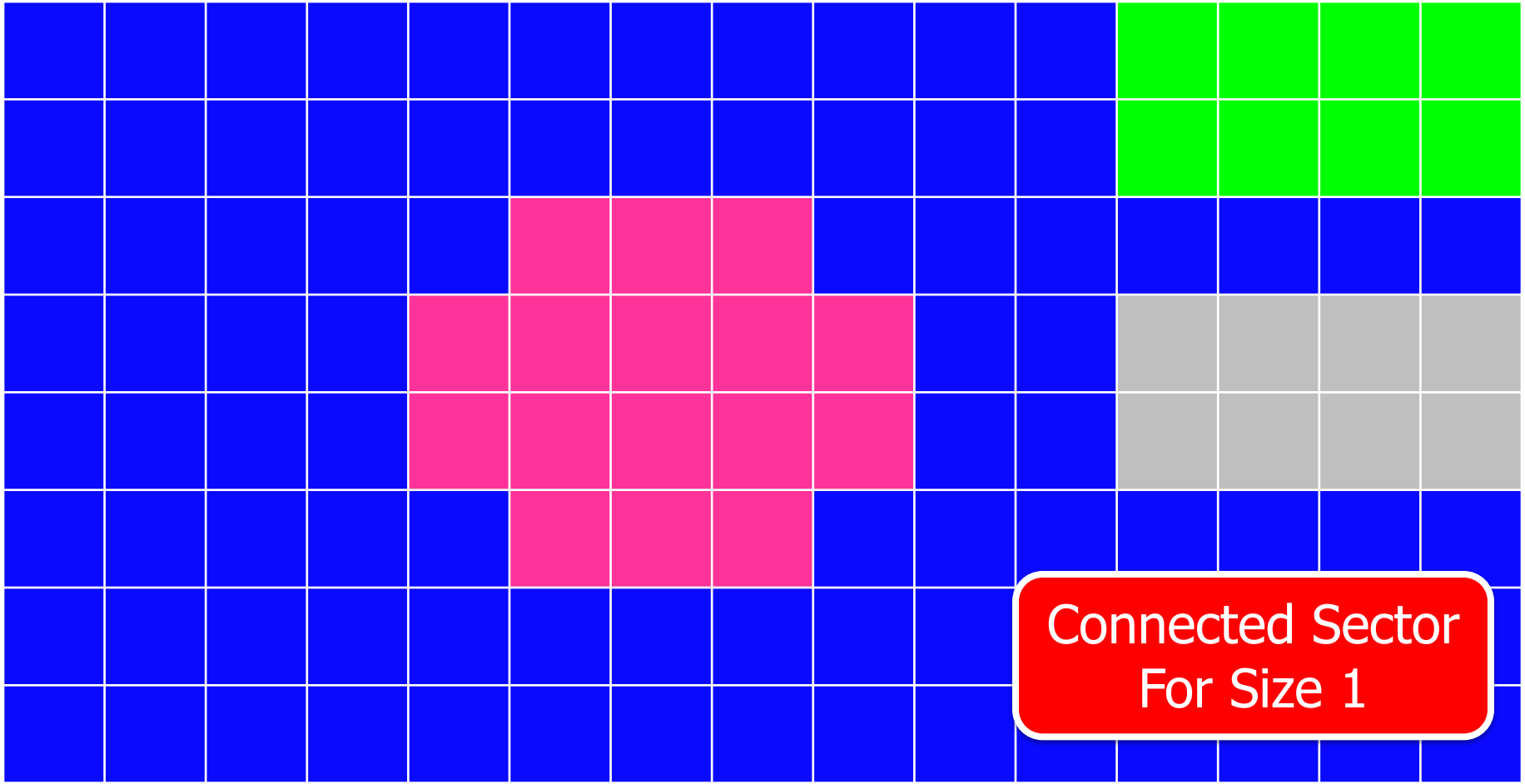

C
A
B
Connected
For Size 2

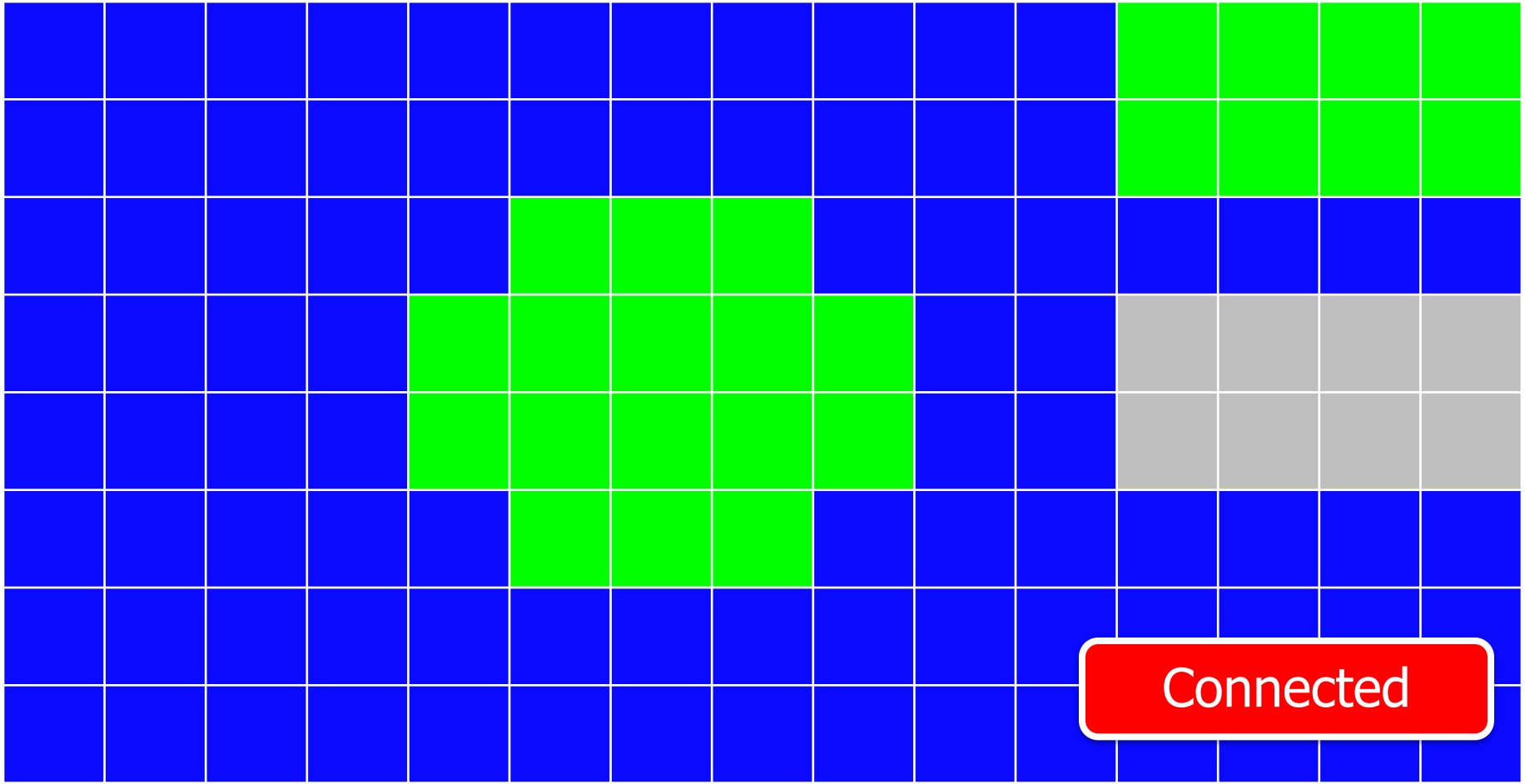

C
A
B
Not Connected
For Size 3


Not Connected


Not Connected

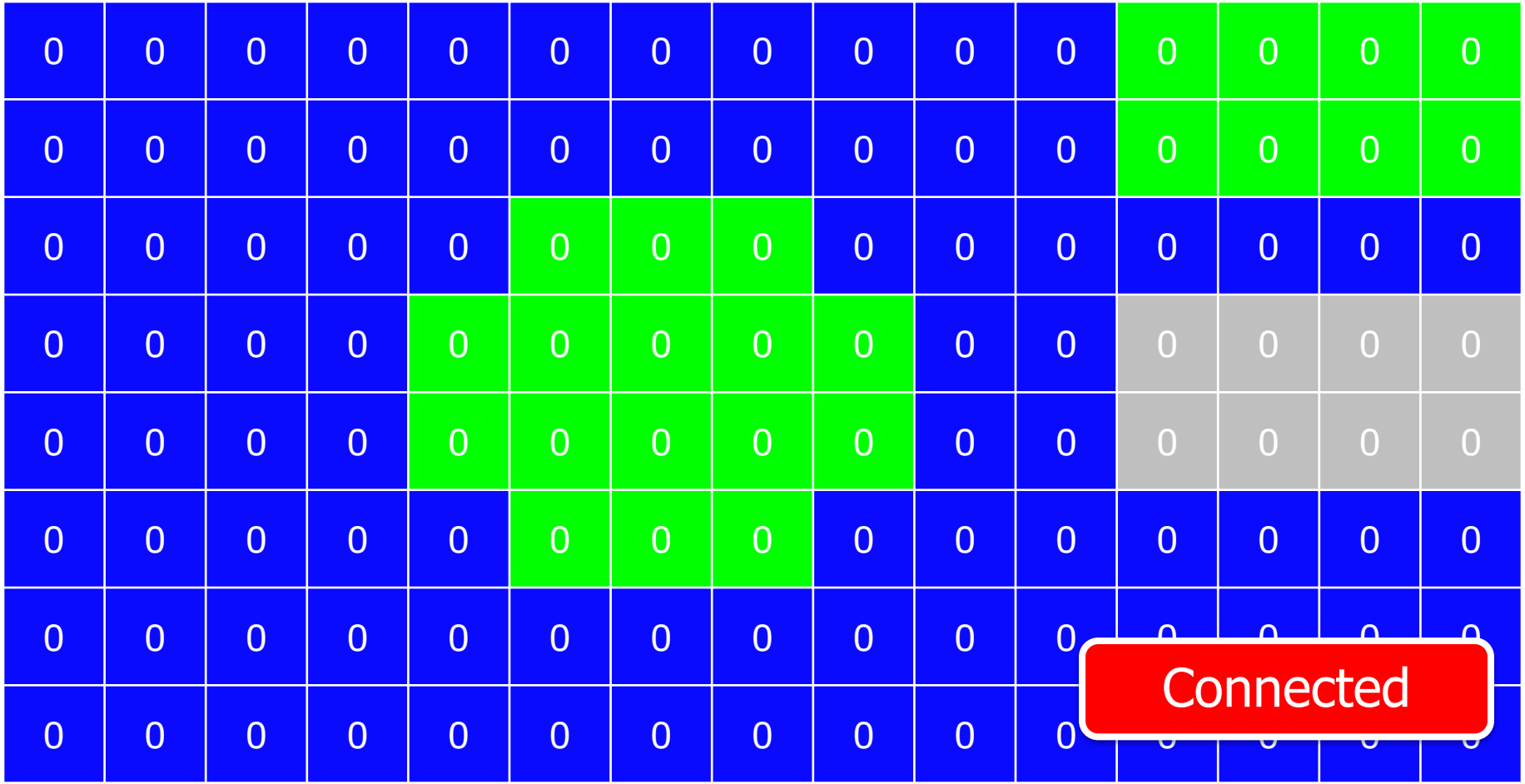

Not Connected

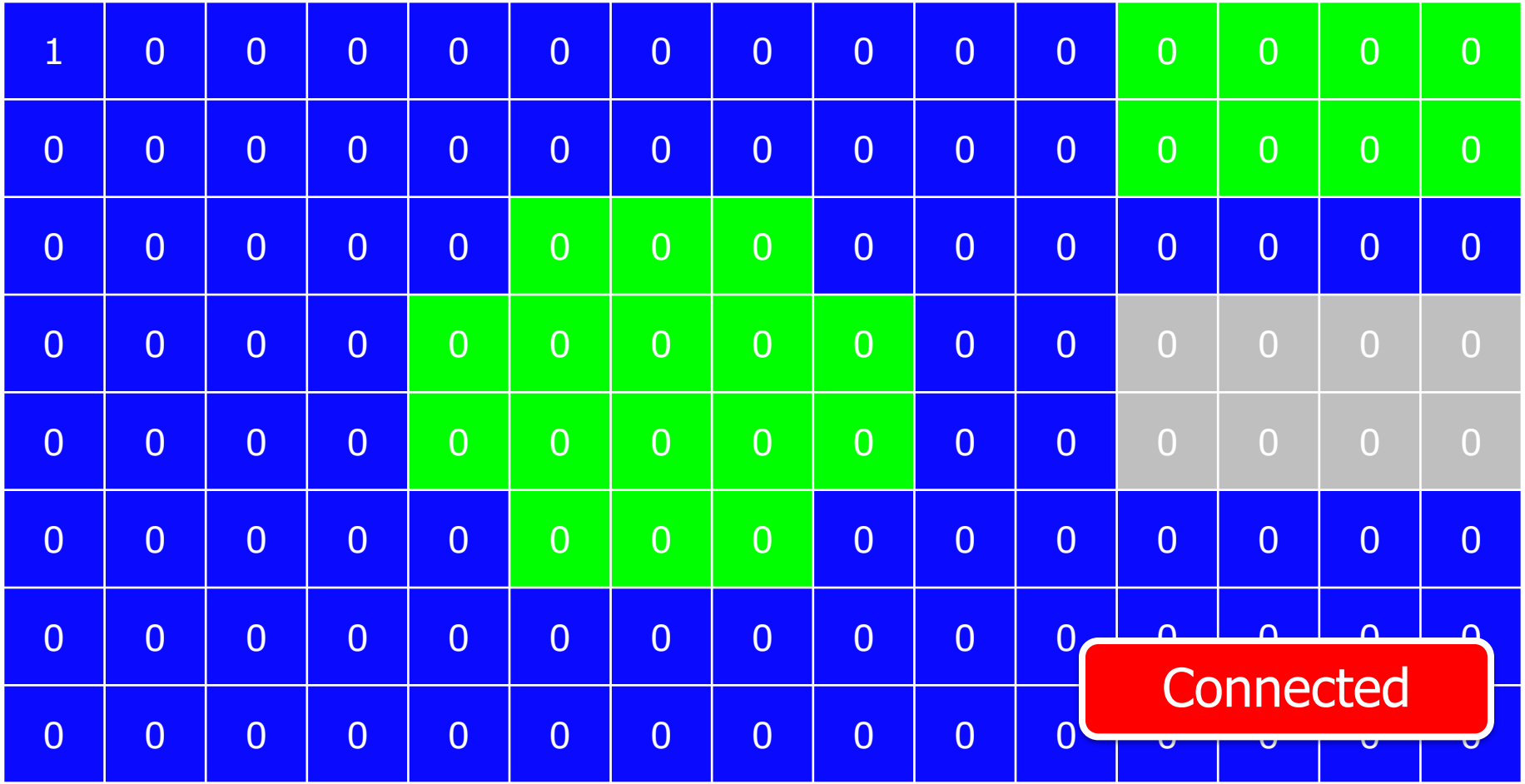

Connected Sector
For Size 1

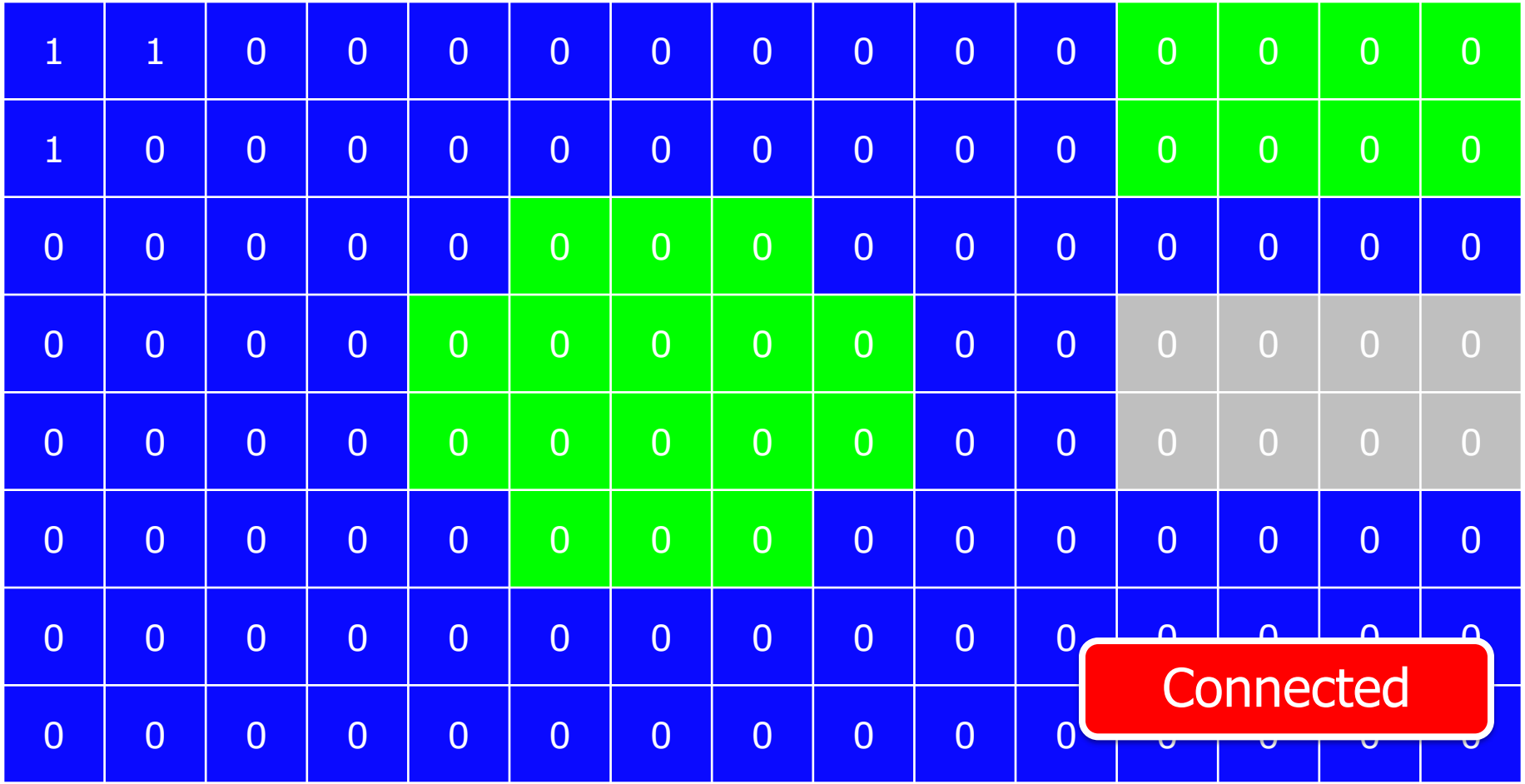

Connected

Computing Connectivity

- A diagonal move is only legal if both surrounding cardinal actions is also legal
- This means that a path with diagonal moves only exists if another path with only cardinal actions also exists (explained in optimization section)
- Therefore, cardinal connectivity is the same as diagonal connectivity, and easier to compute
- Let’s use BFS!


0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0
Connected


1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Connected

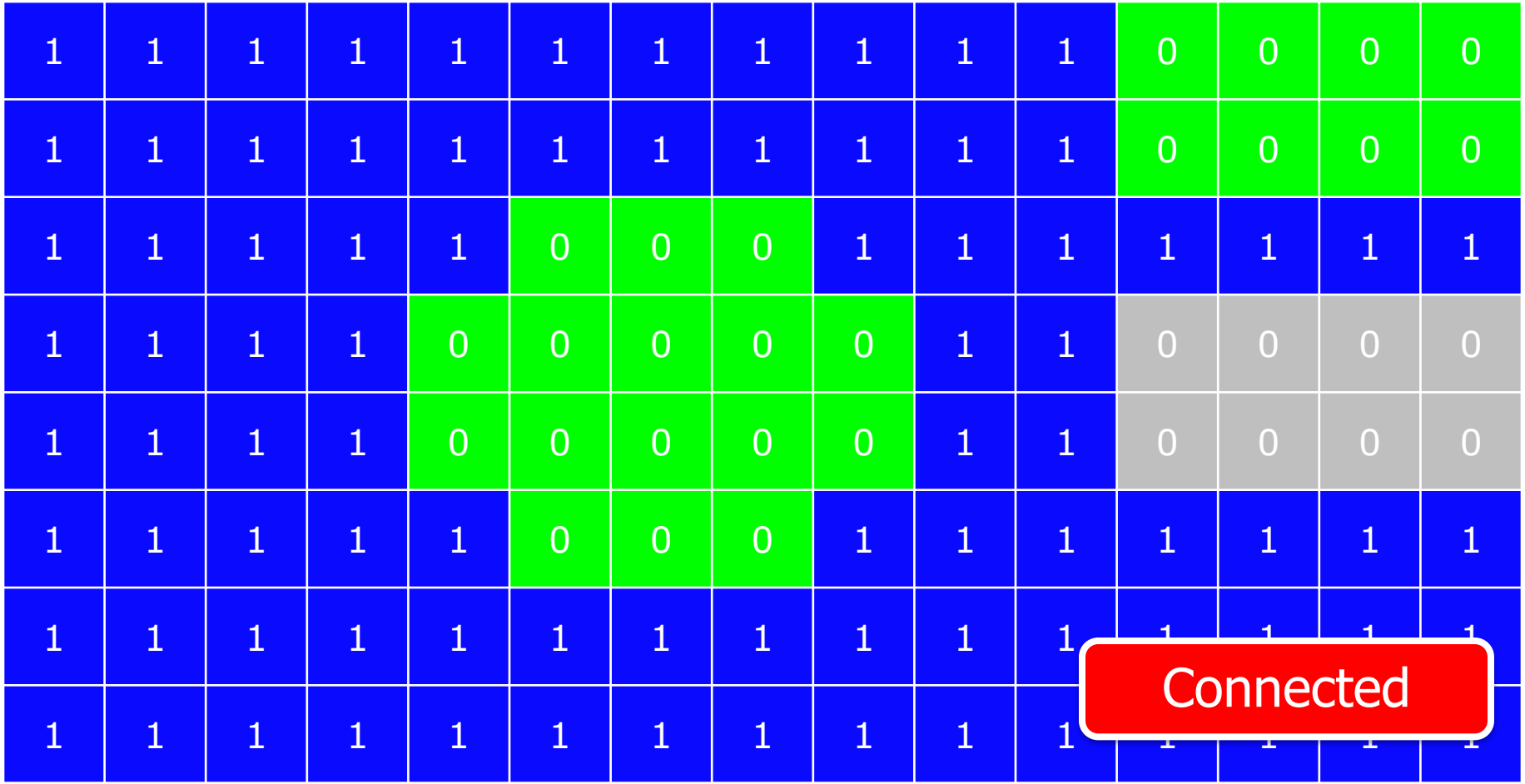

1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Connected

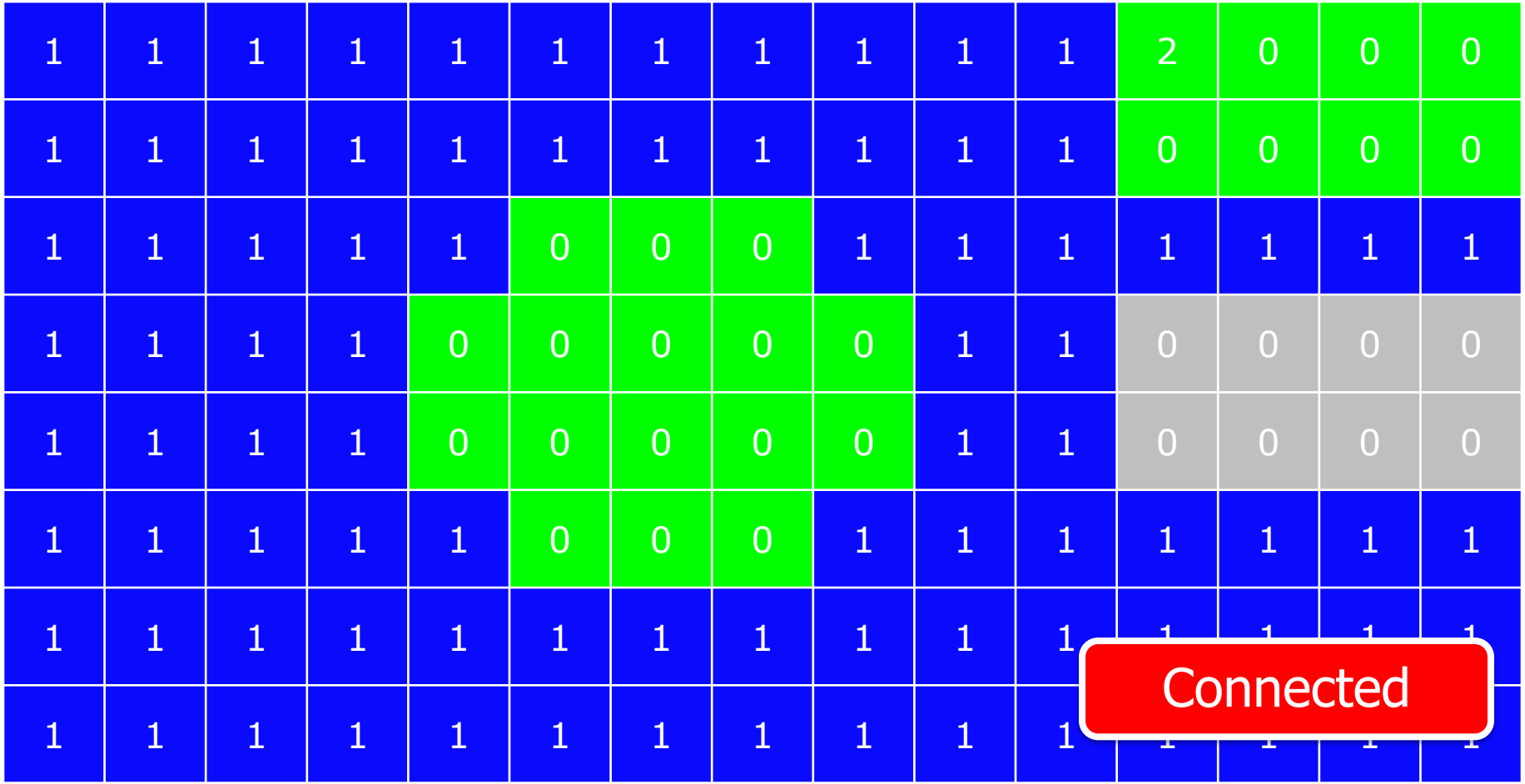
1	1	1	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0

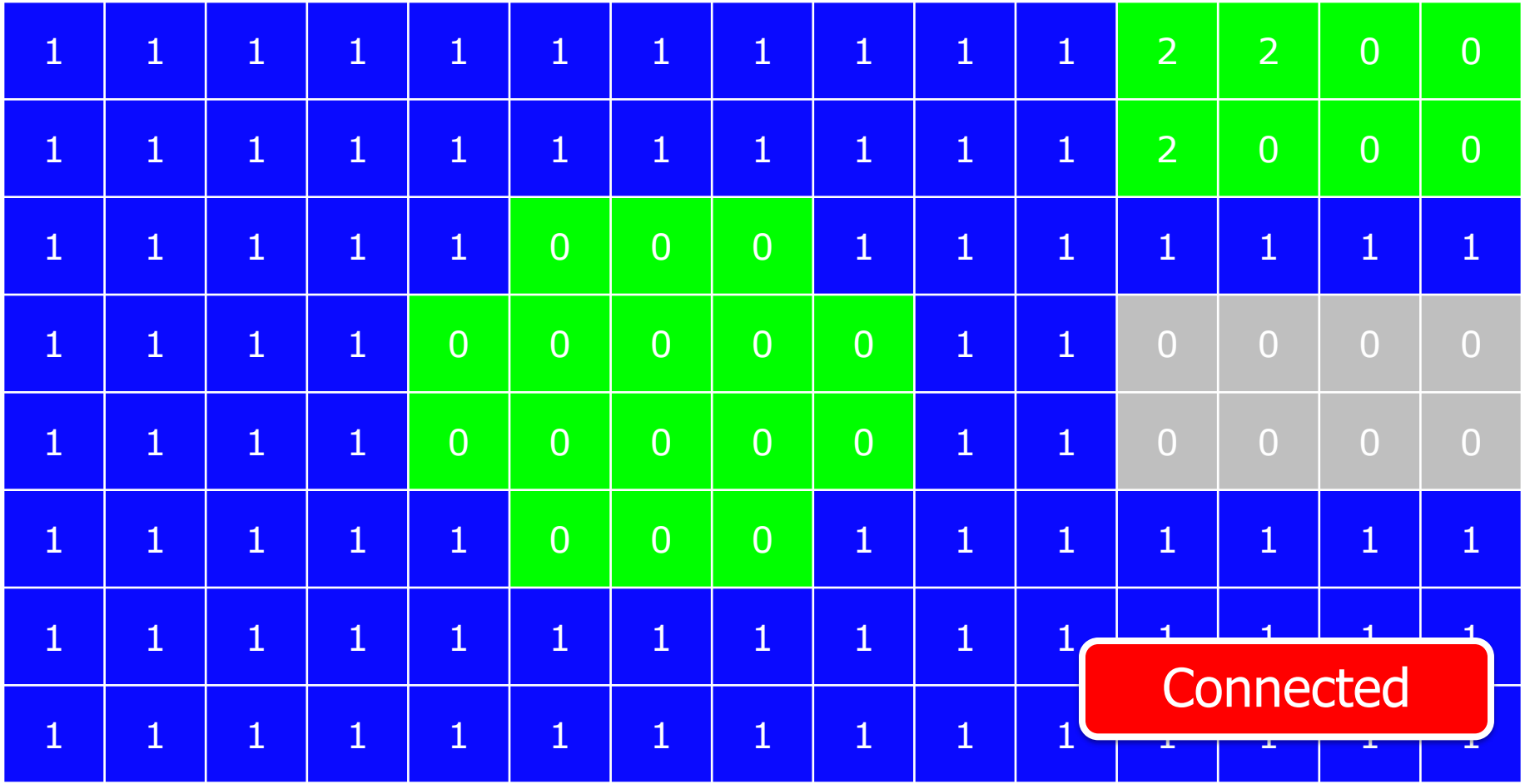
Connected

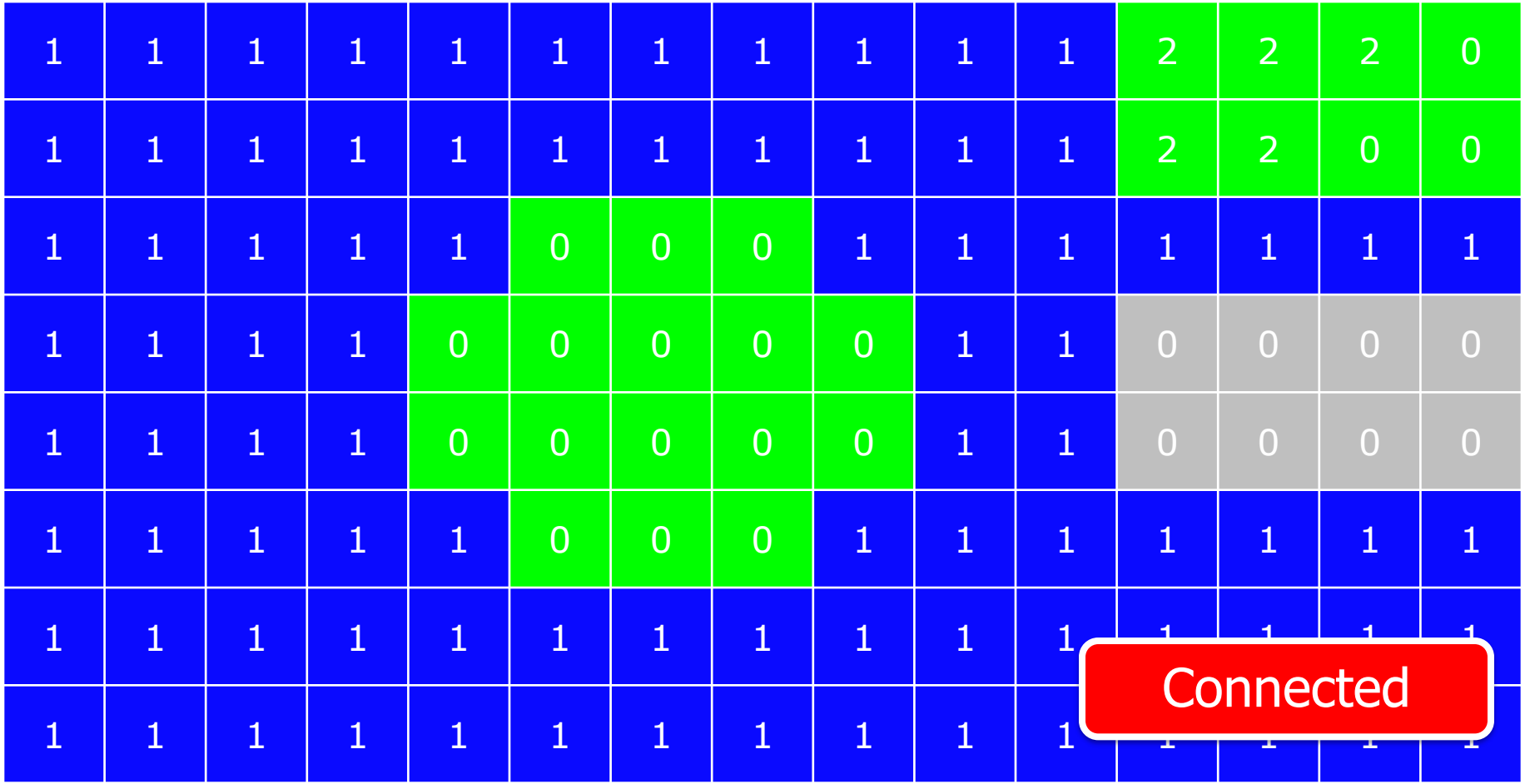
1	1	1	1	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Connected


1 1 1 1 1 1 1 1 1 1 1 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 0 0 0 0
1 1 1 1 1 0 0 0 1 1 1 1 1 1 1
1 1 1 1 0 0 0 0 0 1 1 0 0 0 0
1 1 1 1 0 0 0 0 0 1 1 0 0 0 0
1 1 1 1 1 0 0 0 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1
Connected

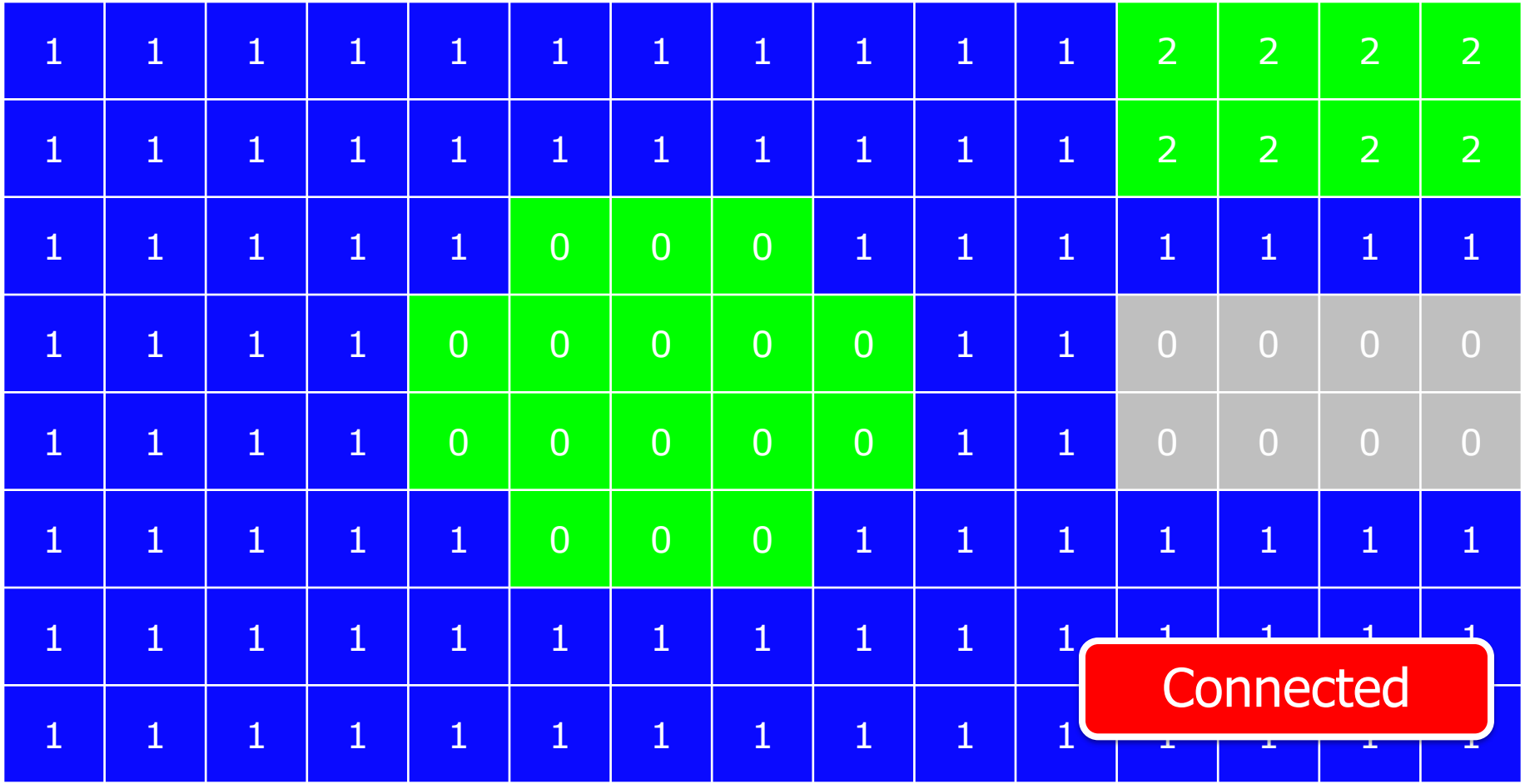

1 1 1 1 1 1 1 1 1 1 1 2 0 0 0
1 1 1 1 1 1 1 1 1 1 1 0 0 0 0
1 1 1 1 1 0 0 0 1 1 1 1 1 1 1
1 1 1 1 0 0 0 0 0 1 1 0 0 0 0
1 1 1 1 0 0 0 0 0 1 1 0 0 0 0
1 1 1 1 1 0 0 0 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1
Connected


1 1 1 1 1 1 1 1 1 1 1 2 2 0 0
1 1 1 1 1 1 1 1 1 1 1 2 0 0 0
1 1 1 1 1 0 0 0 1 1 1 1 1 1 1
1 1 1 1 0 0 0 0 0 1 1 0 0 0 0
1 1 1 1 0 0 0 0 0 1 1 0 0 0 0
1 1 1 1 1 0 0 0 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1
Connected


1 1 1 1 1 1 1 1 1 1 1 2 2 2 0
1 1 1 1 1 1 1 1 1 1 1 2 2 0 0
1 1 1 1 1 0 0 0 1 1 1 1 1 1 1
1 1 1 1 0 0 0 0 0 1 1 0 0 0 0
1 1 1 1 0 0 0 0 0 1 1 0 0 0 0
1 1 1 1 1 0 0 0 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1
Connected

1	1	1	1	1	1	1	1	1	1	1	2	2	2	2
1	1	1	1	1	1	1	1	1	1	1	2	2	2	0
1	1	1	1	1	0	0	0	1	1	1	1	1	1	1
1	1	1	1	0	0	0	0	0	1	1	0	0	0	0
1	1	1	1	0	0	0	0	0	1	1	0	0	0	0
1	1	1	1	1	0	0	0	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Connected

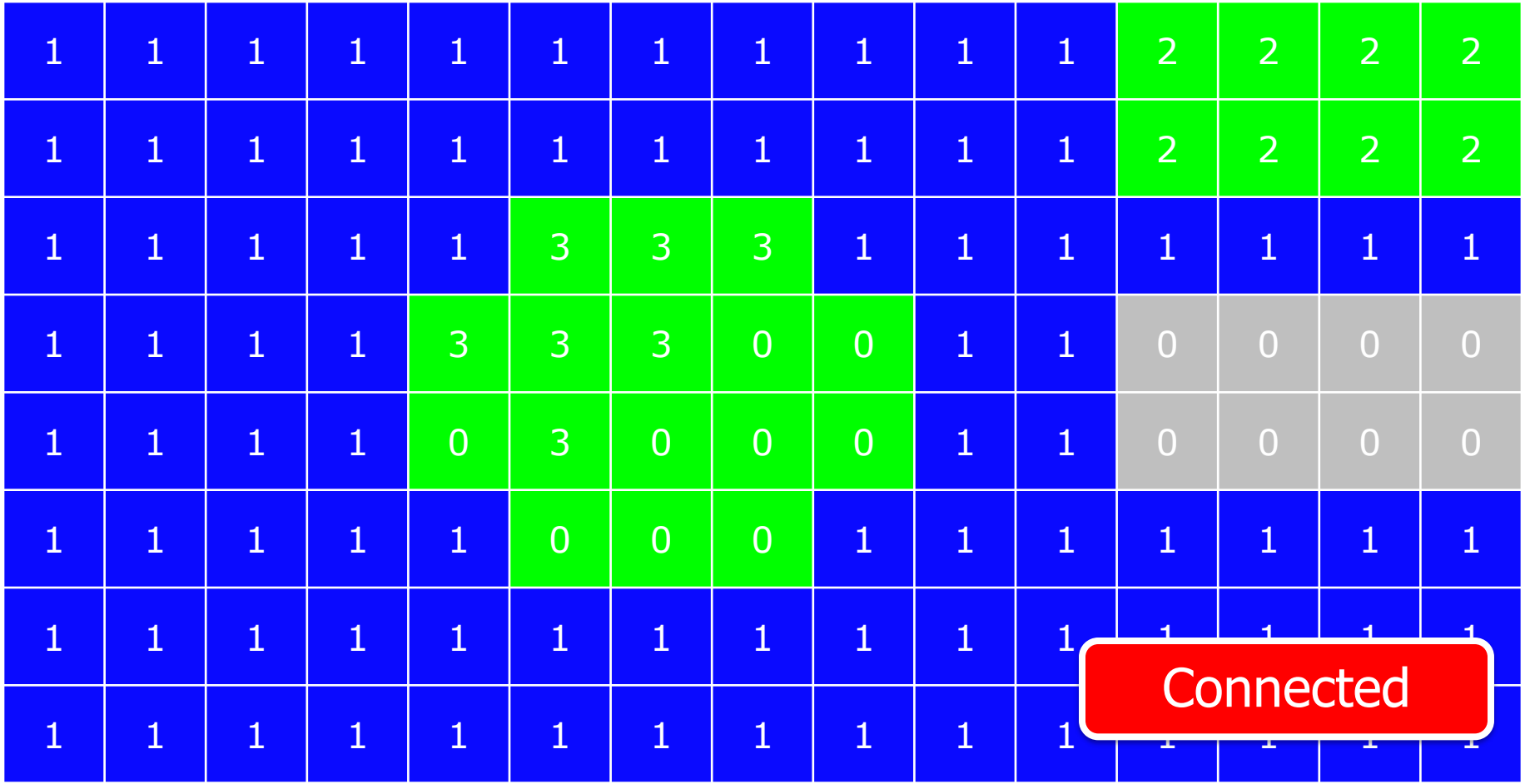

1 1 1 1 1 1 1 1 1 1 1 2 2 2 2
1 1 1 1 1 1 1 1 1 1 1 2 2 2 2
1 1 1 1 1 0 0 0 1 1 1 1 1 1 1
1 1 1 1 0 0 0 0 0 1 1 0 0 0 0
1 1 1 1 0 0 0 0 0 1 1 0 0 0 0
1 1 1 1 1 0 0 0 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1
Connected

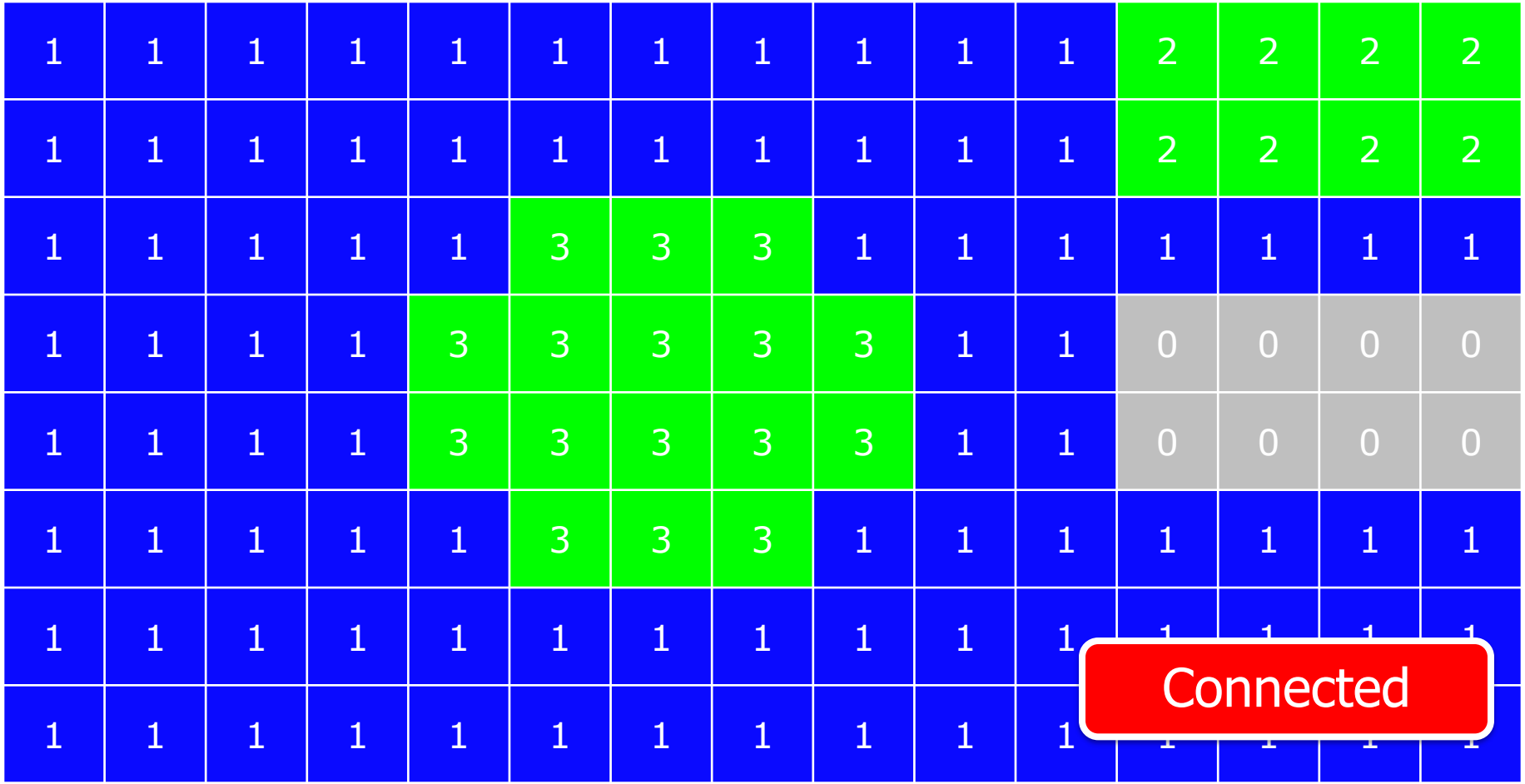
1	1	1	1	1	1	1	1	1	1	1	2	2	2	2
1	1	1	1	1	1	1	1	1	1	1	2	2	2	2
1	1	1	1	1	3	0	0	1	1	1	1	1	1	1
1	1	1	1	0	0	0	0	0	1	1	0	0	0	0
1	1	1	1	0	0	0	0	0	1	1	0	0	0	0
1	1	1	1	1	0	0	0	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Connected

1	1	1	1	1	1	1	1	1	1	1	2	2	2	2
1	1	1	1	1	1	1	1	1	1	1	2	2	2	2
1	1	1	1	1	3	3	0	1	1	1	1	1	1	1
1	1	1	1	0	3	0	0	0	1	1	0	0	0	0
1	1	1	1	0	0	0	0	0	1	1	0	0	0	0
1	1	1	1	1	0	0	0	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

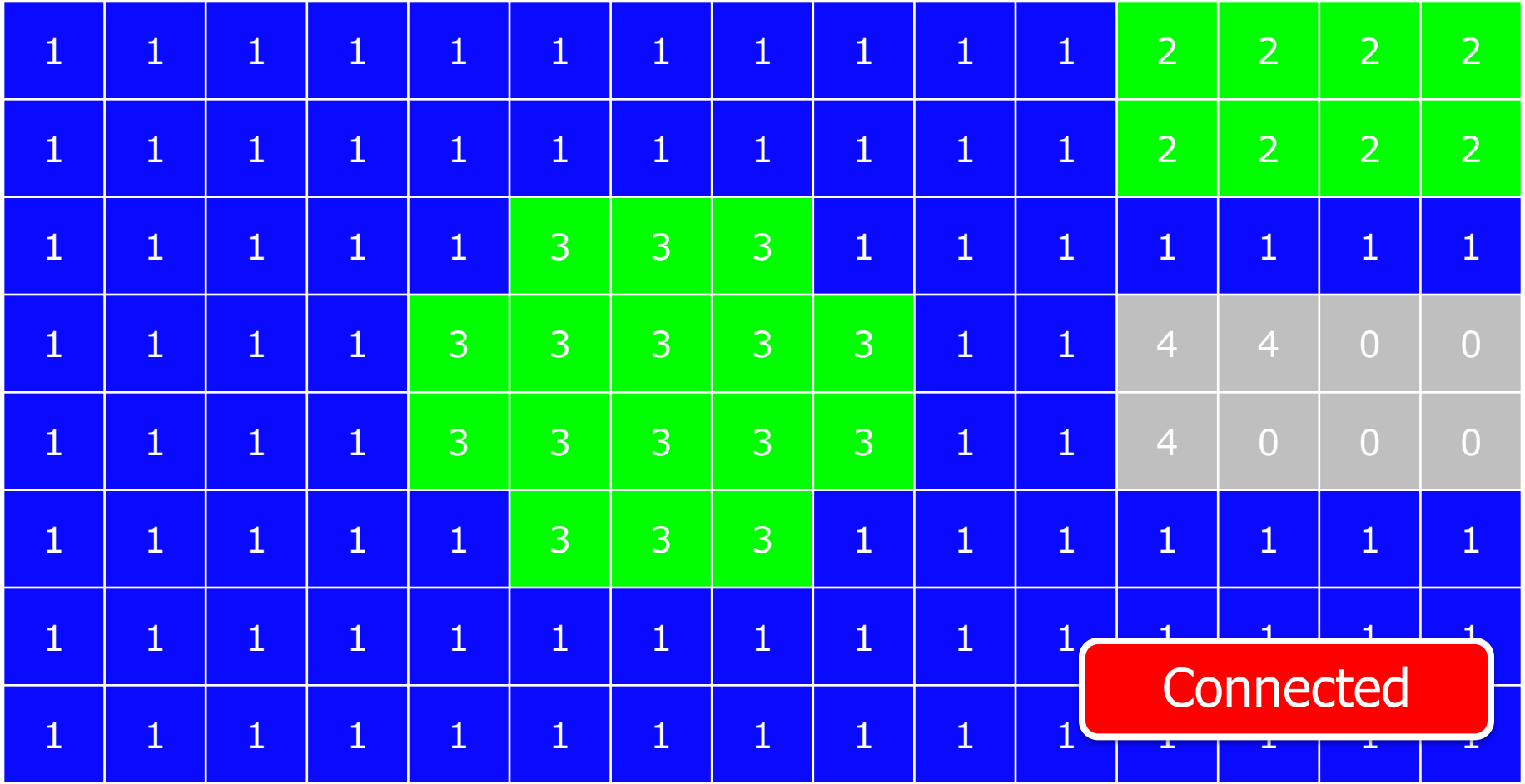
Connected

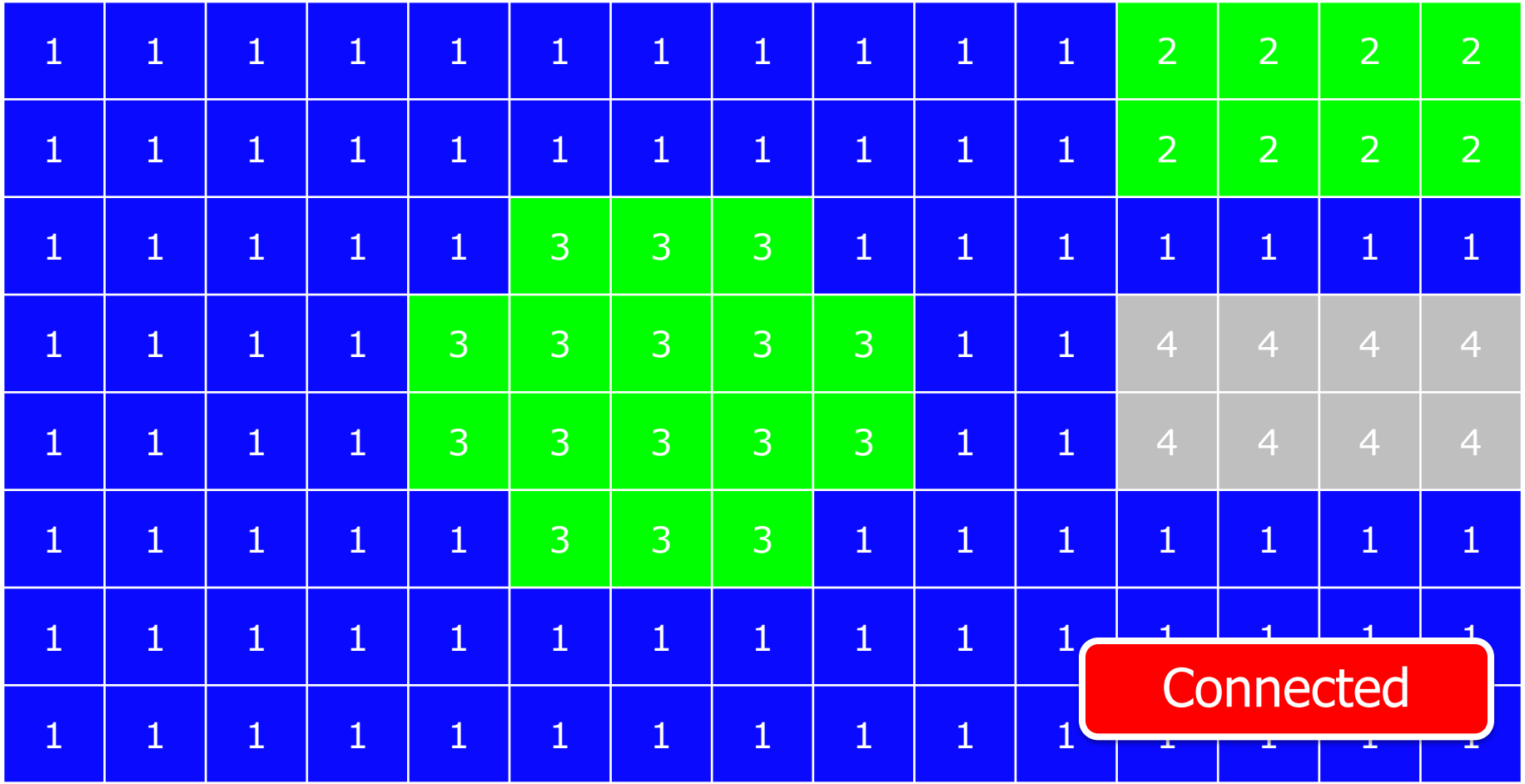

1 1 1 1 1 1 1 1 1 1 1 2 2 2 2
1 1 1 1 1 1 1 1 1 1 1 2 2 2 2
1 1 1 1 1 3 3 3 1 1 1 1 1 1 1
1 1 1 1 3 3 3 0 0 1 1 0 0 0 0
1 1 1 1 0 3 0 0 0 1 1 0 0 0 0
1 1 1 1 1 0 0 0 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1
Connected


1 1 1 1 1 1 1 1 1 1 1 2 2 2 2
1 1 1 1 1 1 1 1 1 1 1 2 2 2 2
1 1 1 1 1 3 3 3 1 1 1 1 1 1 1
1 1 1 1 3 3 3 3 3 1 1 0 0 0 0
1 1 1 1 3 3 3 3 3 1 1 0 0 0 0
1 1 1 1 1 3 3 3 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1
Connected

1	1	1	1	1	1	1	1	1	1	1	2	2	2	2
1	1	1	1	1	1	1	1	1	1	1	2	2	2	2
1	1	1	1	1	3	3	3	1	1	1	1	1	1	1
1	1	1	1	3	3	3	3	3	1	1	4	0	0	0
1	1	1	1	3	3	3	3	3	1	1	0	0	0	0
1	1	1	1	1	3	3	3	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Connected


1 1 1 1 1 1 1 1 1 1 1 2 2 2 2
1 1 1 1 1 1 1 1 1 1 1 2 2 2 2
1 1 1 1 1 3 3 3 1 1 1 1 1 1 1
1 1 1 1 3 3 3 3 3 1 1 4 4 0 0
1 1 1 1 3 3 3 3 3 1 1 4 0 0 0
1 1 1 1 1 3 3 3 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1
Connected


1 1 1 1 1 1 1 1 1 1 1 2 2 2 2
1 1 1 1 1 1 1 1 1 1 1 2 2 2 2
1 1 1 1 1 3 3 3 1 1 1 1 1 1 1
1 1 1 1 3 3 3 3 3 1 1 4 4 4 4
1 1 1 1 3 3 3 3 3 1 1 4 4 4 4
1 1 1 1 1 3 3 3 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1
Connected

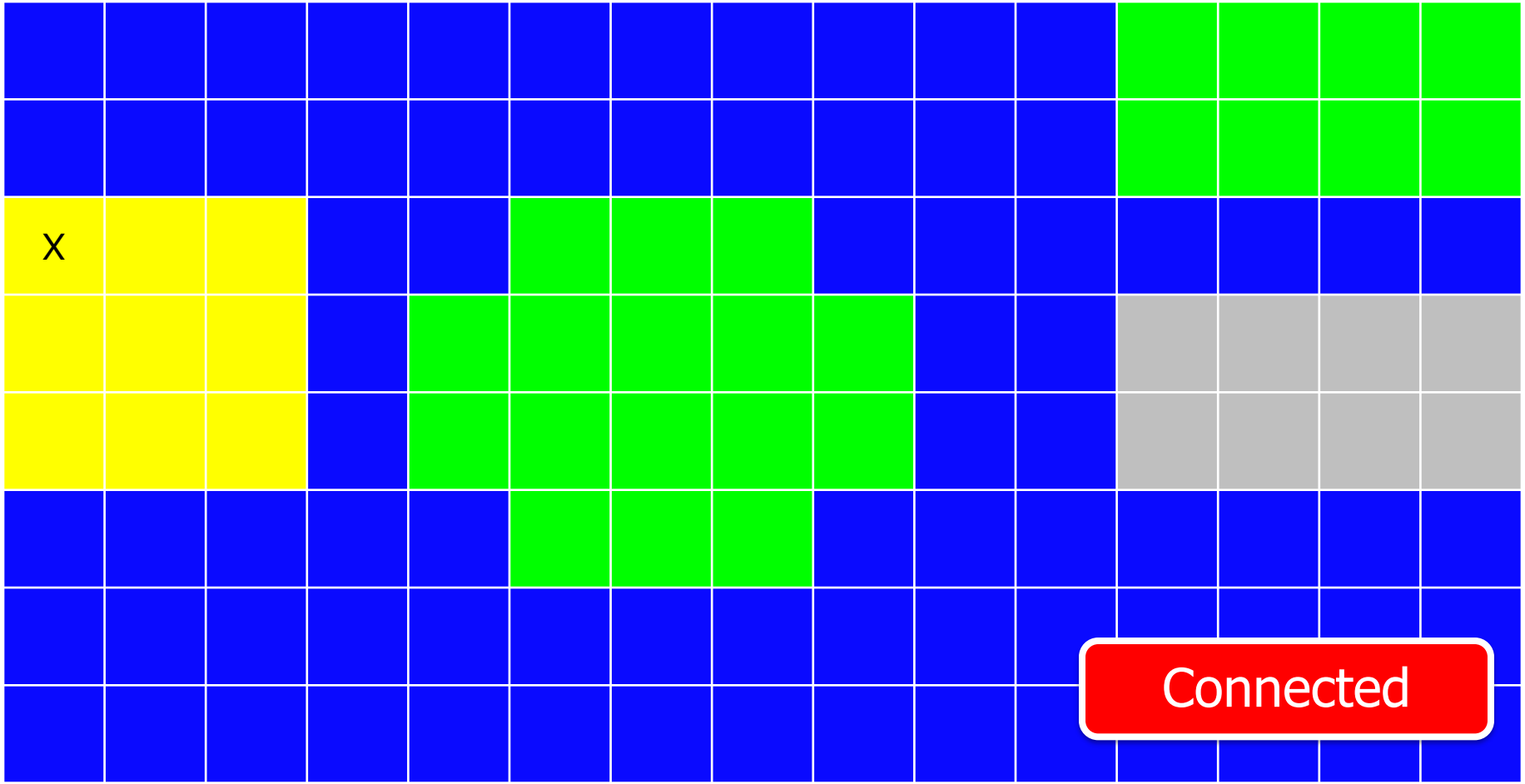
Computing Connectivity

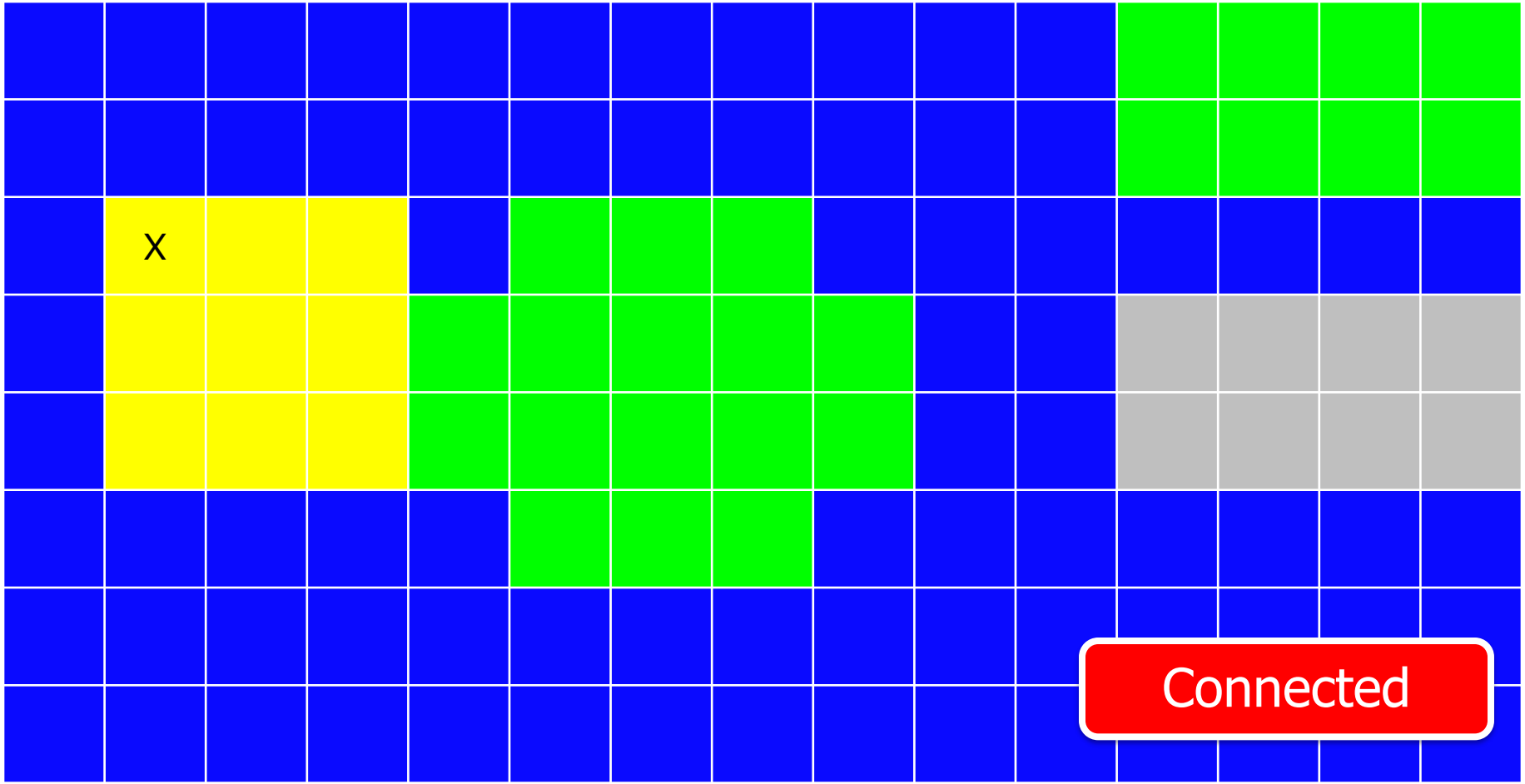
```
1.  function computeSectors()
2.       sectors = Grid(mapWidth, mapHeight, 0)
3.       sectorNumber = 0
4.          for (x=0; x<mapWidth; x++)
5.              for (y=0; y<mapHeight; y++)
6.                  if (sectors[x][y] != 0 || !canFit(x,y,s)) continue;
7.                  sectorNumber += 1
8.                  CardinalBFS(x, y, sectorNumber);
```

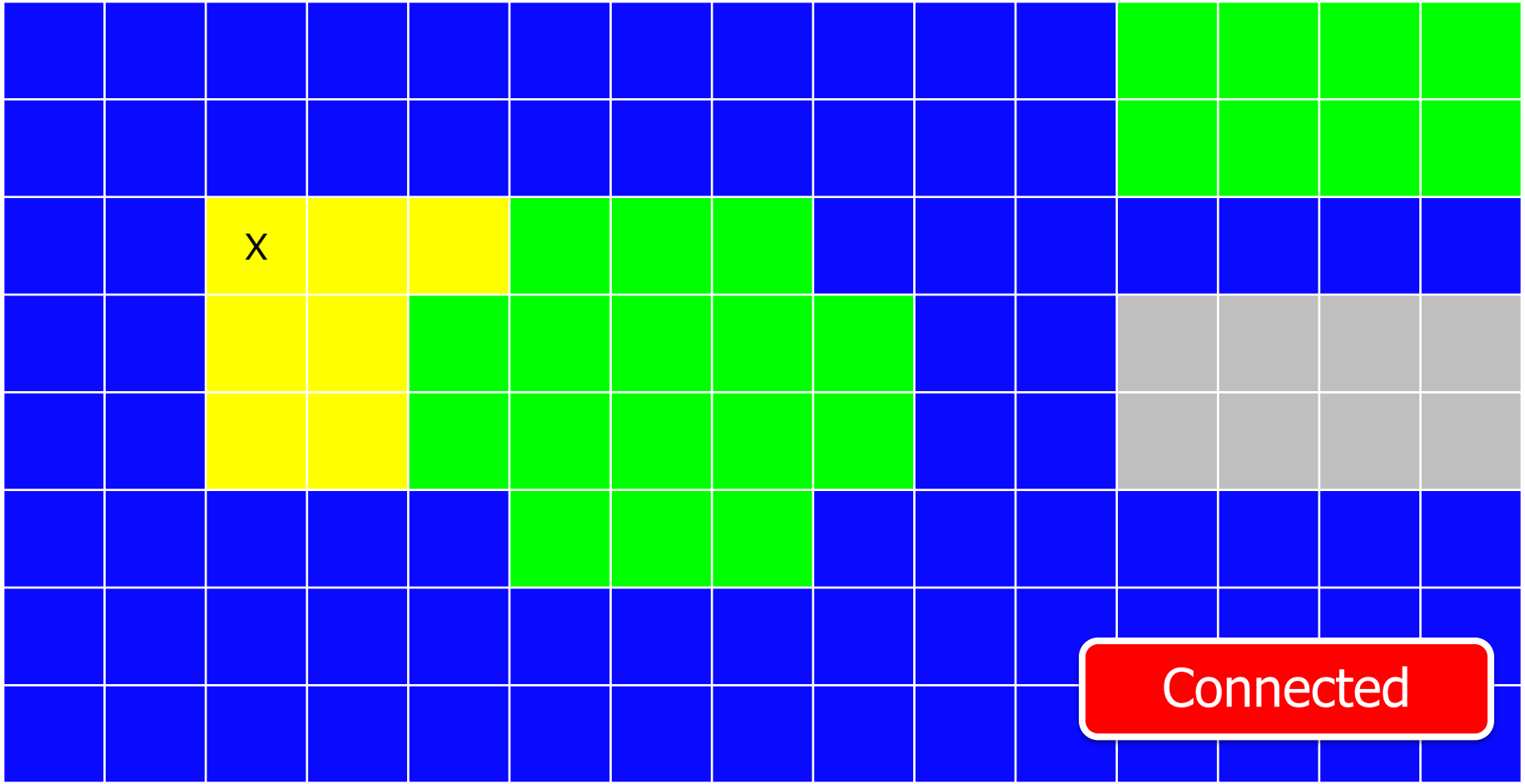
```
1.  function isConnected(x1,y1,x2,y2)
2.          return sectors[x1][y1] == sectors[x2][y2]
```

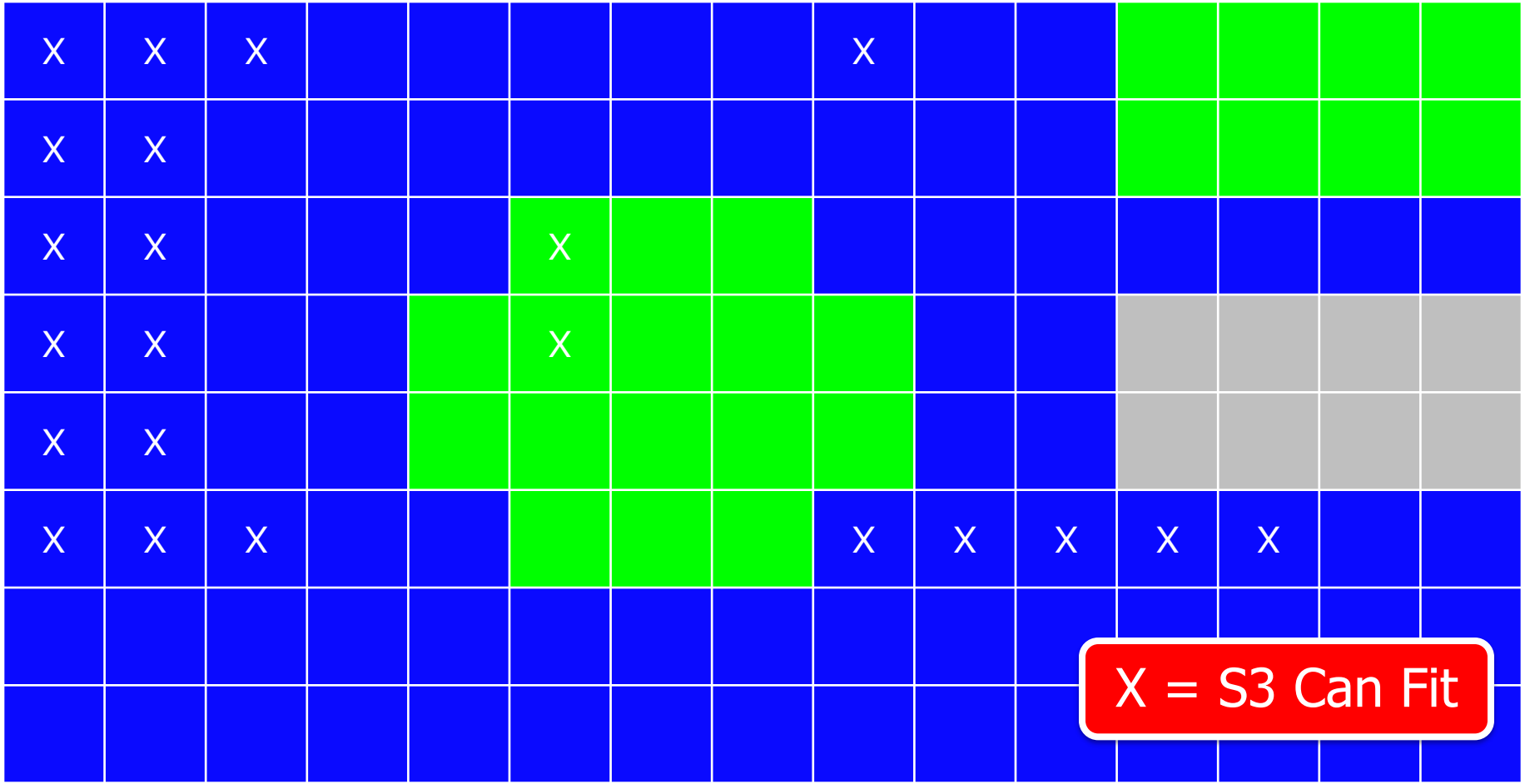
Connectivity Check

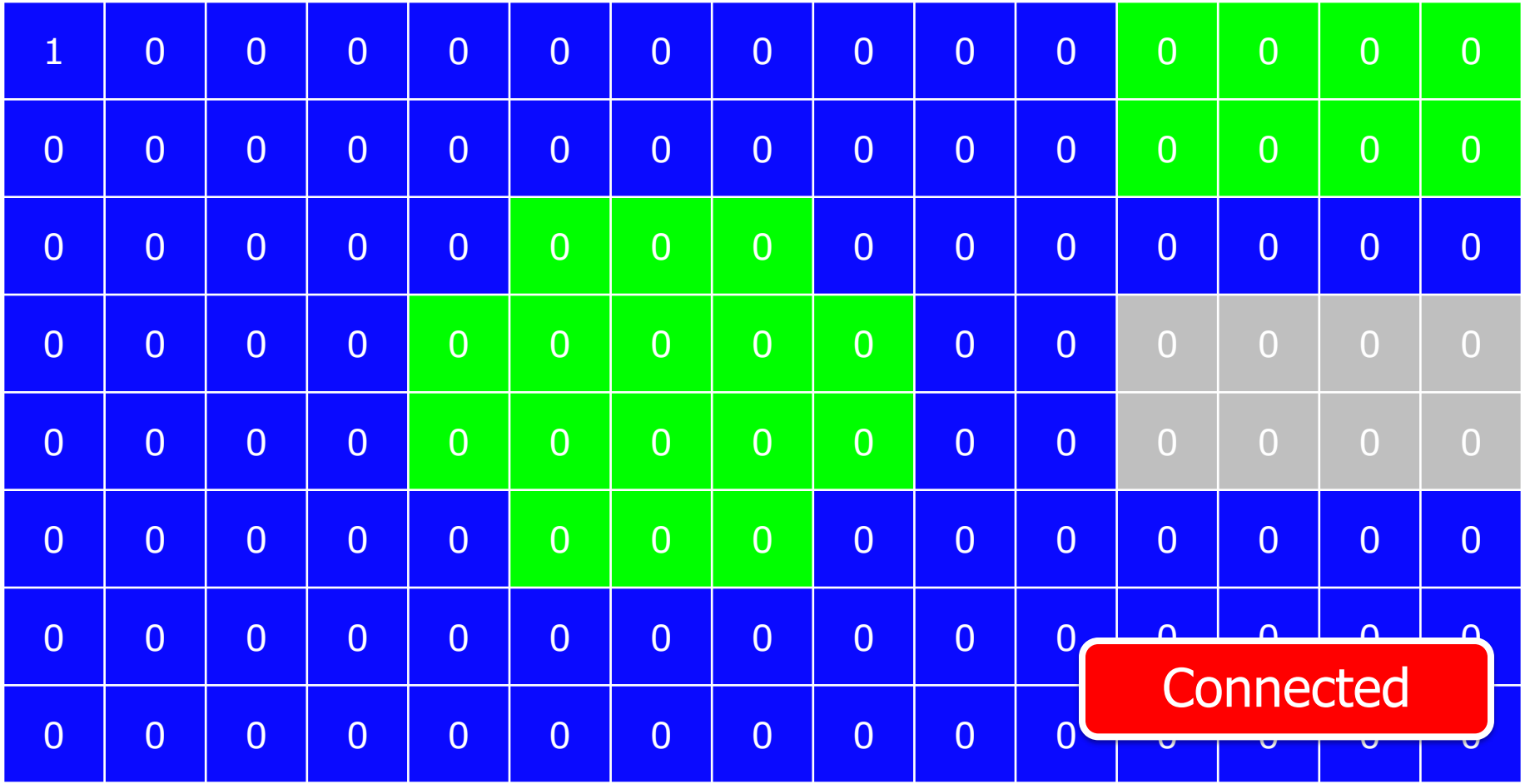
- Two tiles are connected if their precomputed sector number is the same
- We can use this as a pre-check to see if a path is possible before computing it
  - If start and goal are not connected, then no path can exist between them!
- Let’s look at bigger object sizes

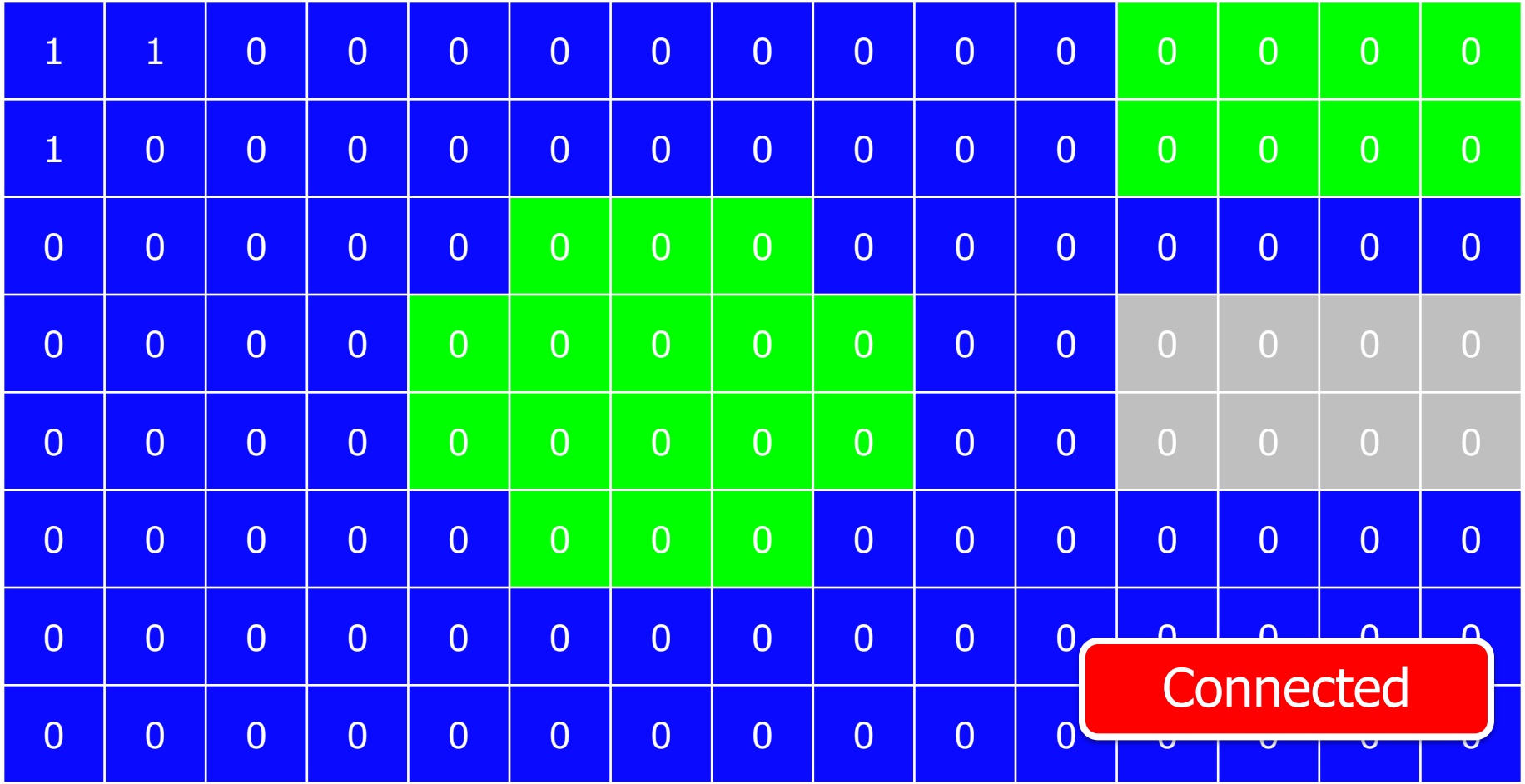

X
Connected


X
Connected


X
Connected


X = S3 Can Fit


1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Connected

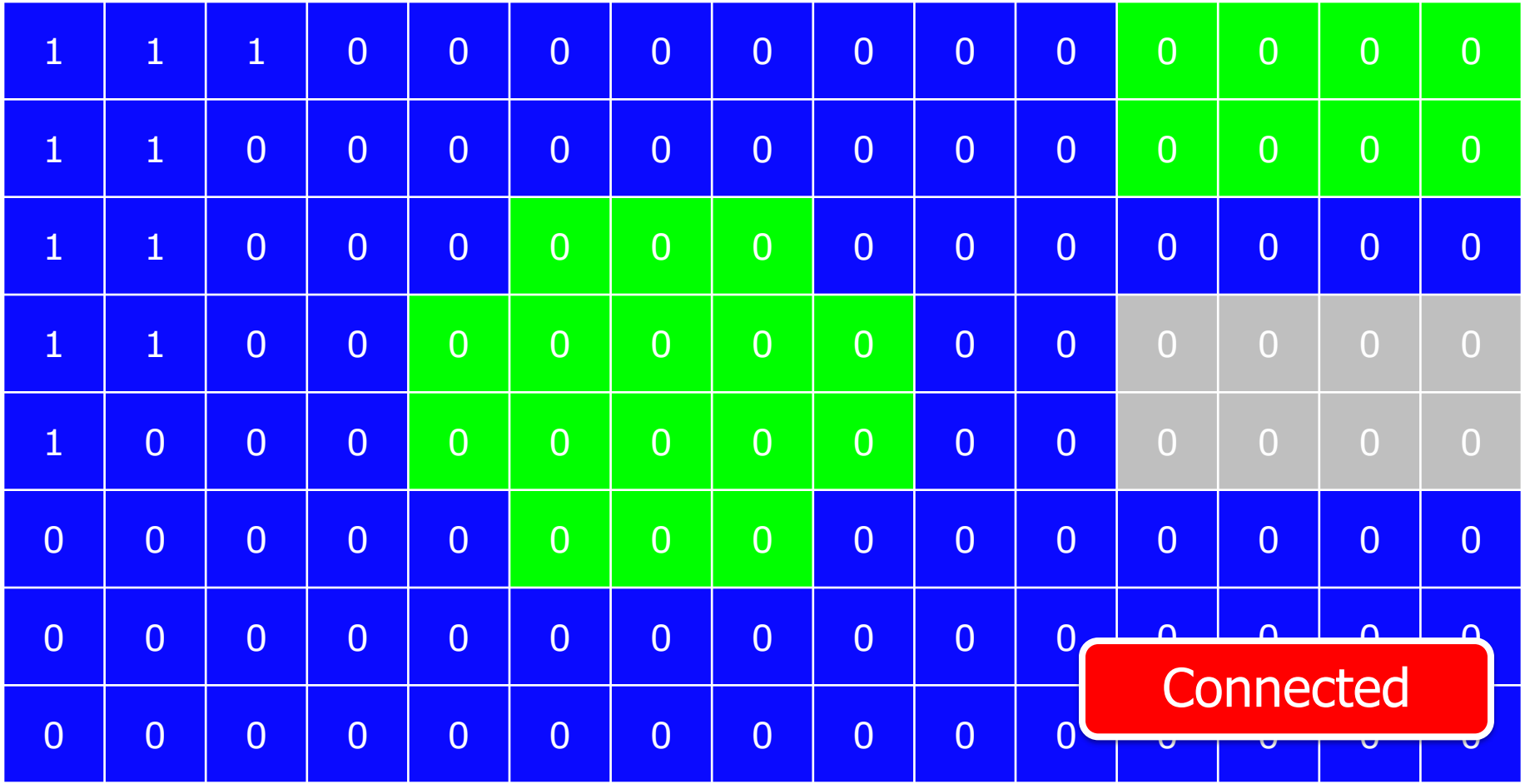

1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Connected

1	1	1	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0

Connected

1	1	1	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0

Connected


1 1 1 0 0 0 0 0 0 0 0 0 0 0
1 1 0 0 0 0 0 0 0 0 0 0 0 0
1 1 0 0 0 0 0 0 0 0 0 0 0 0
1 1 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
Connected

1	1	1	0	0	0	0	0	4	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	2	0	0	0	0	0	0	0	0	0
1	1	0	0	0	2	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	3	3	3	3	3	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Connected

Connected Sector (sizes)

- Your Search_Student object will not re-construct if a different size agent is selected
- You must compute connectivity of all sizes in computeSectors() when your object constructs
- I recommend:
  - this.sectors[x][y][s] = sector [x,y] at size s
  - 3D array, or 2D array of arrays [s1, s2, s3]
  - In practice I use maxSize+1 length this.sectors[x][y] = [0, s1, s2, s3] so [1] = size 1

Speed Optimizations

- The following optimizations will not affect the number of nodes generated, but will dramatically speed up running time
- None of these optimizations are REQUIRED for getting full assignment marks
- Some of them MAY be required to get smooth paths while dragging mouse

Connected Sector Optimization

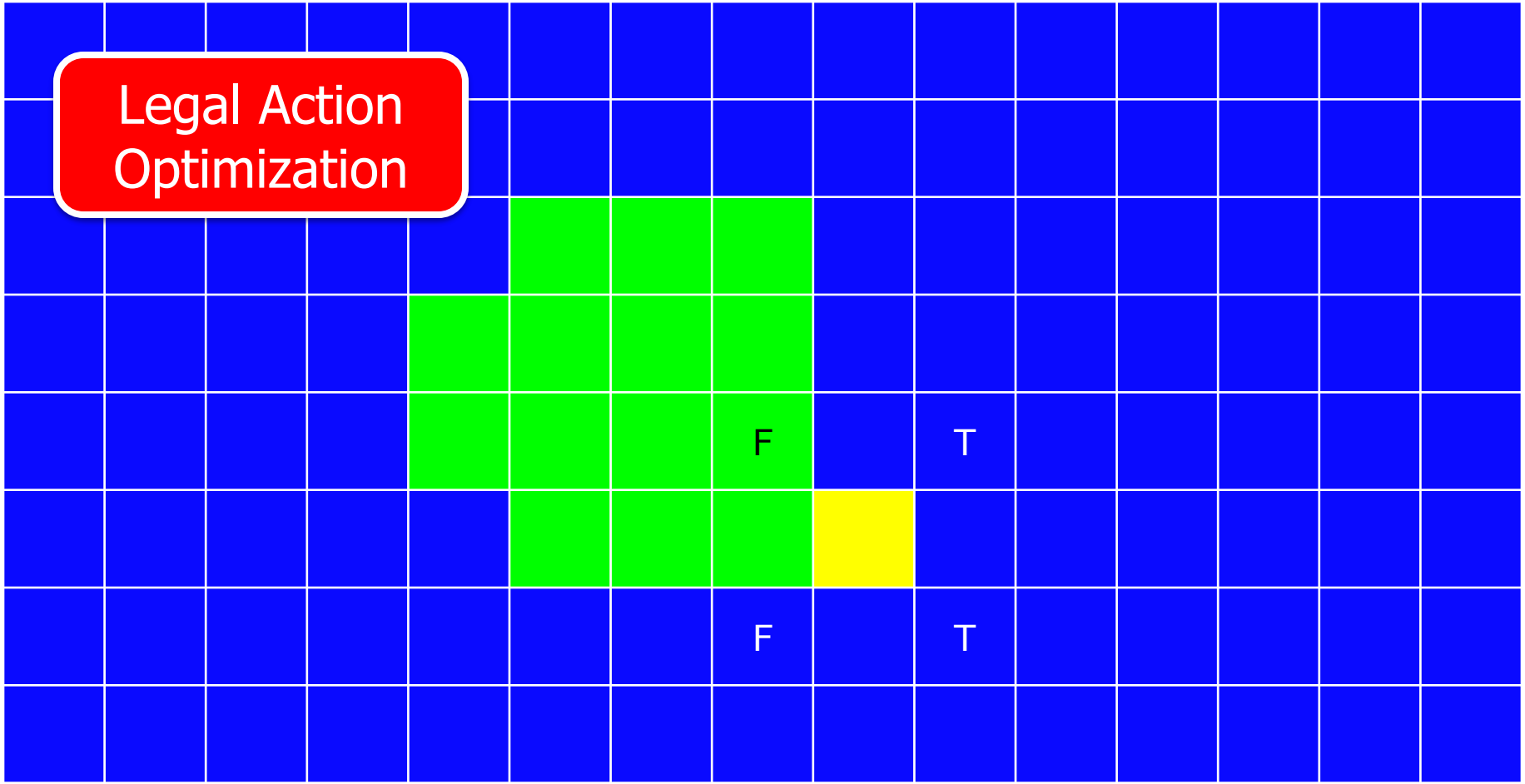
- For larger object sizes, we may have some tiles that have a 0 value
- This means that the object cannot fit
- By precomputing these values, we can use them later instead of doing the full object size check to see if all of the underlying tiles have the same value

1	1	1	0	0	0	0	0	4	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	2	0	0	0	0	0	0	0	0	0
1	1	0	0	0	2	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	3	3	3	3	3	0	0
0	0	0	0	0	0	0	0	0	0	0				
0	0	0	0	0	0	0	0	0	0	0				

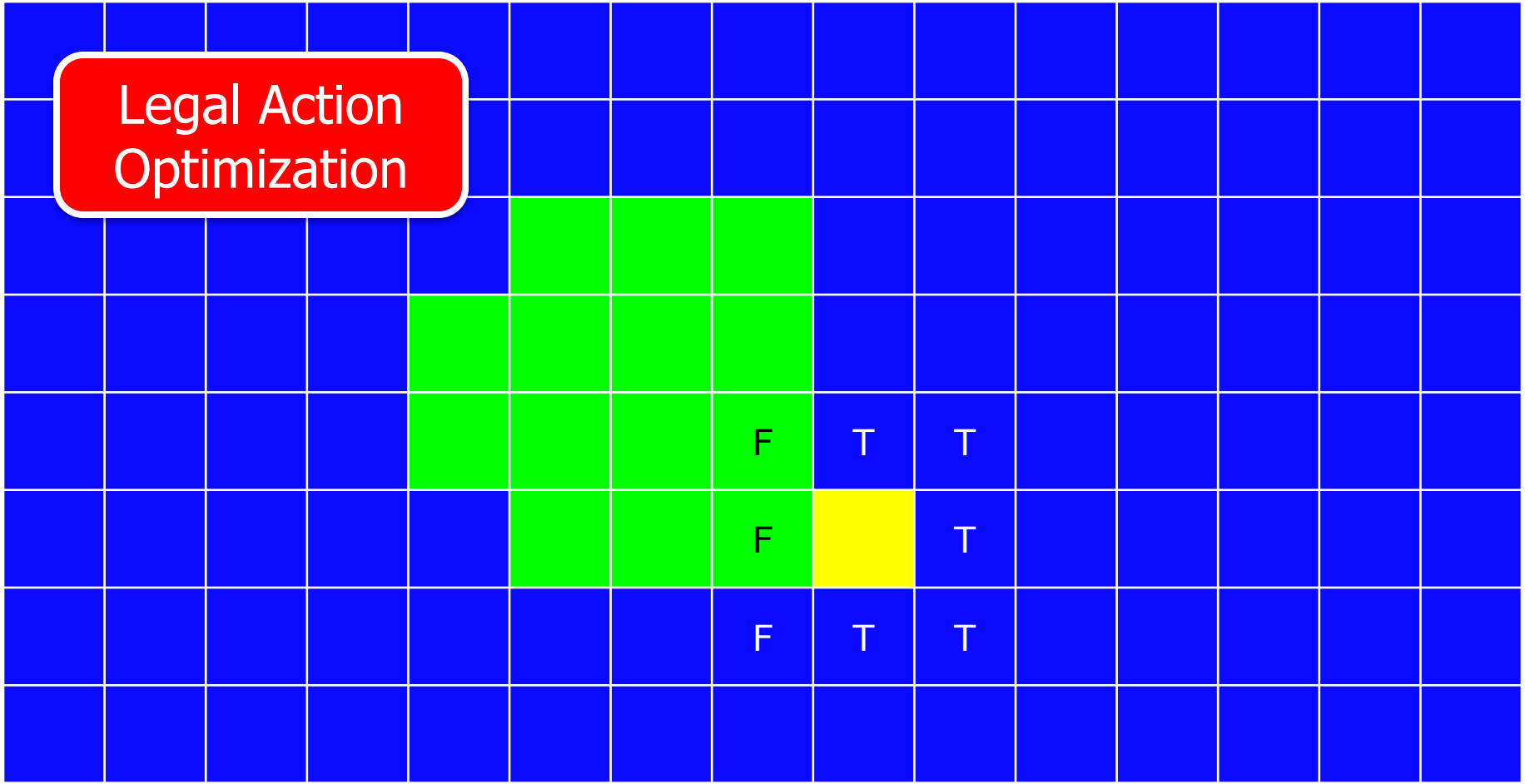
Connected (3)
0 = Can't Fit

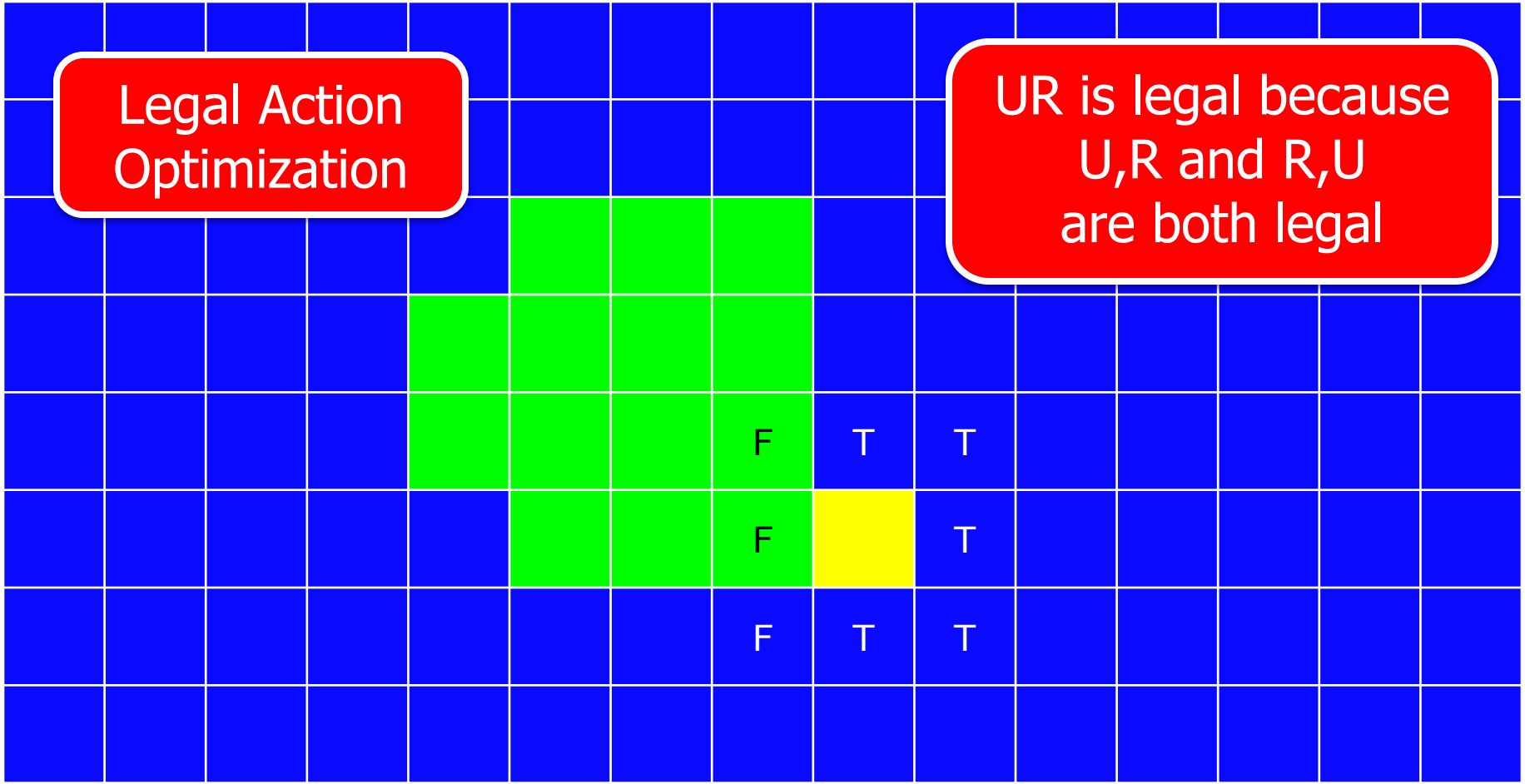
Legal Action Optimization

- The rule of ‘no jumping over diagonals’ actually gives us an interesting property
- A diagonal move of XY is only legal if both X,Y and Y,X are legal as well
- This property holds for all object sizes


Legal Action Optimization
F
T
F
T

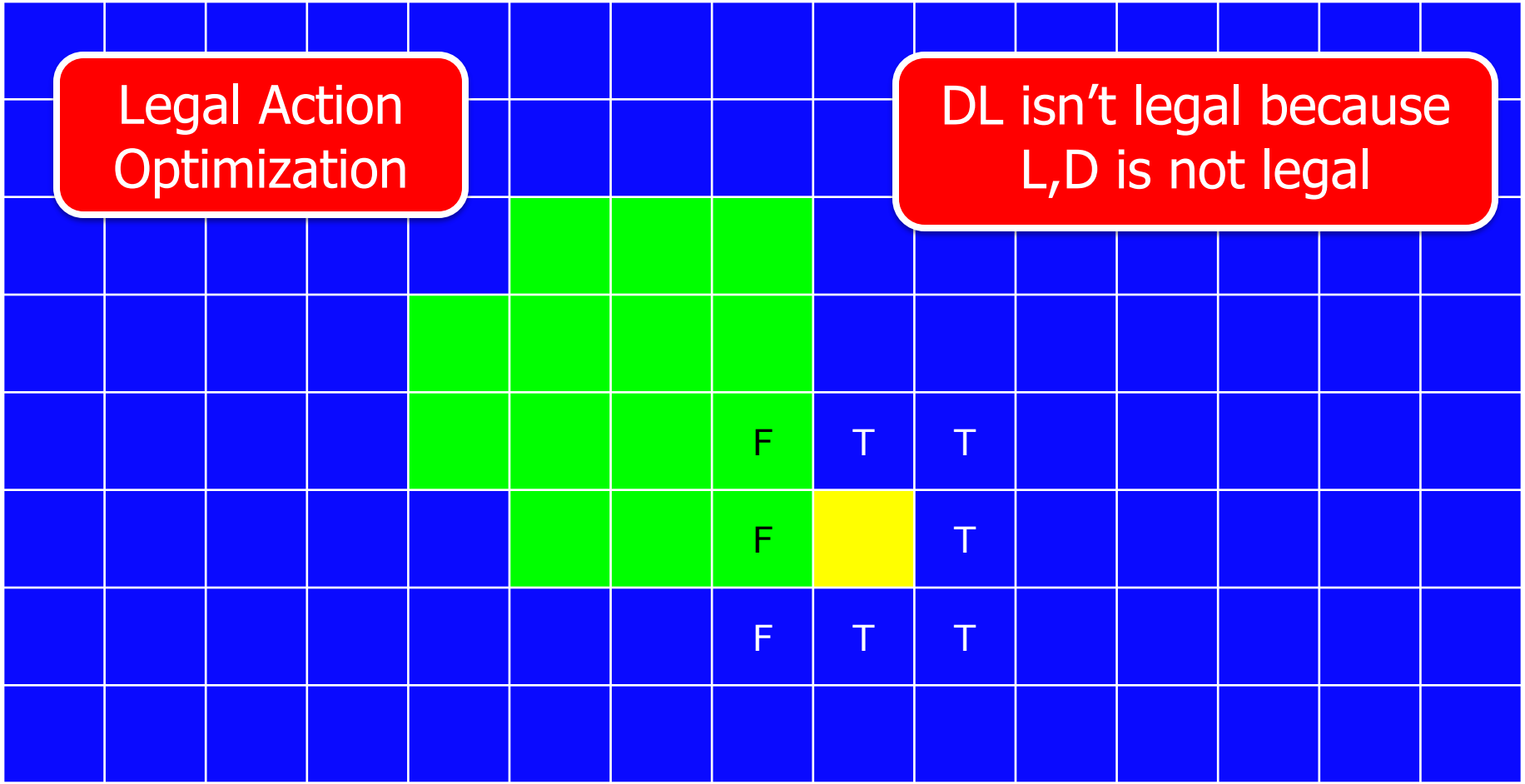
Legal Action Optimization

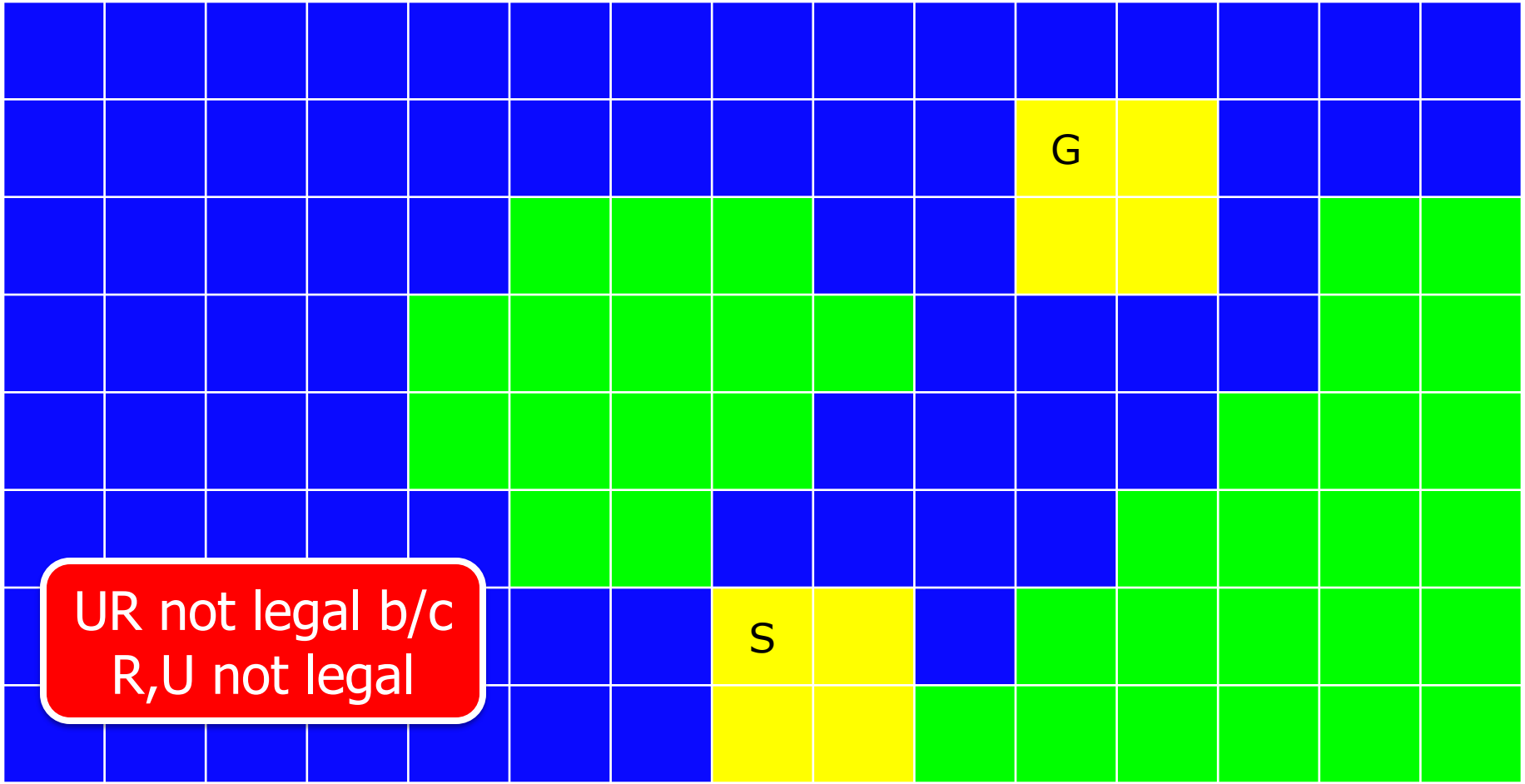



Legal Action Optimization
UR is legal because U,R and R,U are both legal
F
T
T
F
T
F
T
T

Legal Action Optimization

DL isn't legal because L,D is not legal



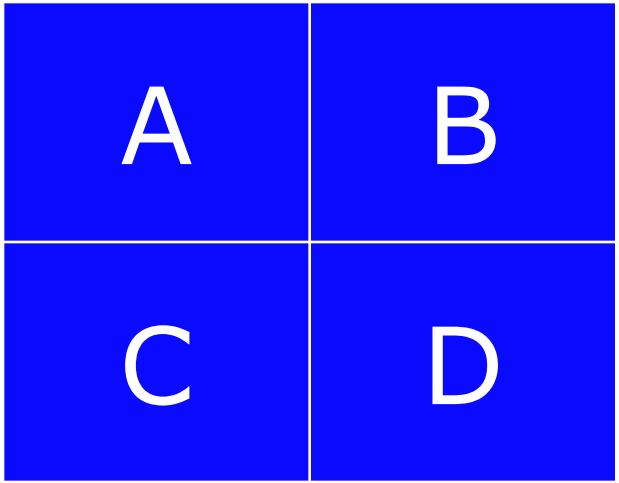

G
S
UR not legal b/c
R,U not legal

isLegalAction Optimization

- The most commonly called function in A2 is probably isLegalAction, since it is called when every child node generated
- We can use the precomputed connectivity sectors to check for legal cardinal actions, since actions are legal only if the source and destination tiles are connected

isLegalAction Optimization

- Diagonal action AD is legal only if AB, BD, AC, CD are also legal
- AB is legal if A,B connected
  - Same for all cardinal actions UDLR
- AB, BD, AC, CD are legal if each of A,B,C,D are all connected
- If ABCD all connected then AD legal
- Connectivity is transitive:
  - If AB and BD connected, AD connected
  - So if AB, AC, AD connected, AD is legal



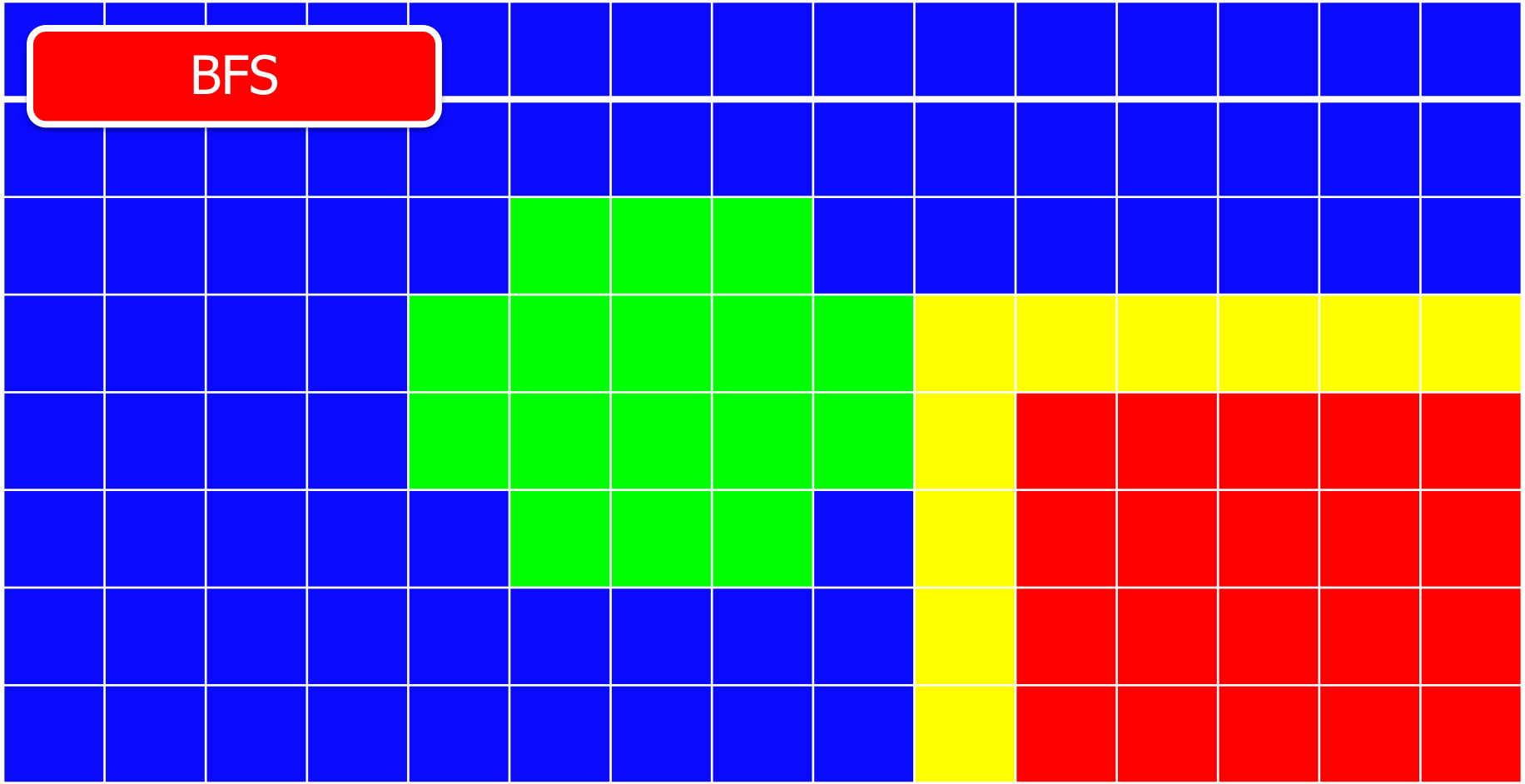
Legal Action Optimization++^{2}

- Our environment is static
- Tiles will have the same legal actions for a given object size every time you visit
- You can precompute all legal actions for all tiles and later just iterate over the list of legal actions rather than generate them
- This will result in a lot of time saved

[0,1]	Precomputed Legal Action Arrays	[0,1]
[0,-1], [1,0], [0,1], [1,1]	[-1,0], [1,0] [0, 1], [-1,1] [1, 1]	[-1,0], [0,-1] [0,1], [-1,1]
[0,-1], [1, 0] [1, -1]	[-1,0], [1, 0] [0, -1], [-1, -1] [1, -1]	[-1, 0], [0, -1] [-1, -1]

Closed List Optimization

- Closed list stores unique states
- Each state on the map will either be in the closed list or not in the closed list
- Store a 2D array of true/false the same size as the map which denotes whether or not that state is in the closed list
- Closed list membership now constant time


BFS

Open List Optimization

- You can use an Array for the open list and iterate through it for min_f
- Or you can use the included BinaryHeap data structure, which will be much faster
- `let open = new BinaryHeap(sortFunction)`
  - `open.push(node)` – adds node
  - `open.pop()` – return and remove min node
  - `open.peek()` – return min node (don't remove)

BFS Performance Notes

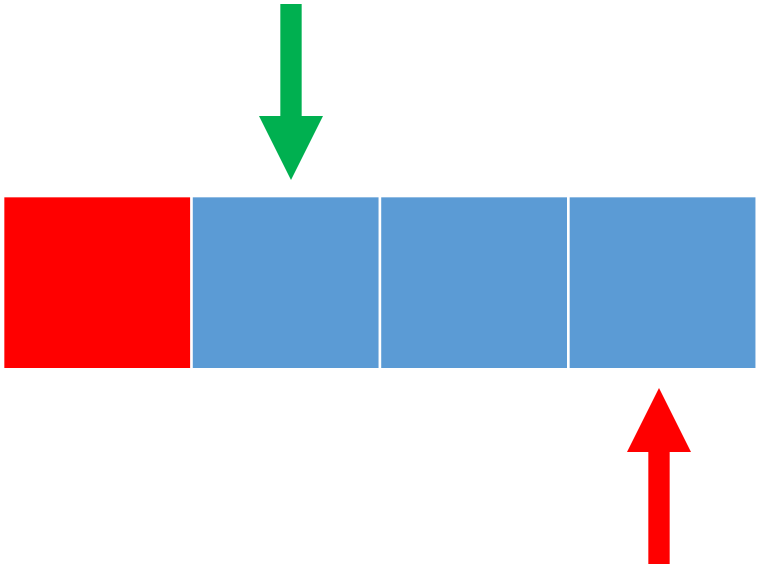
- Actually implementing BFS iteratively with a Queue results in reduced performance
- Vector-implemented queue needs to left shift all elements to pop from the front, which can be very slow in practice
- Performing look-up in a closed list can also be slow if implemented poorly

BFS Optimizations (Avoid Queue)

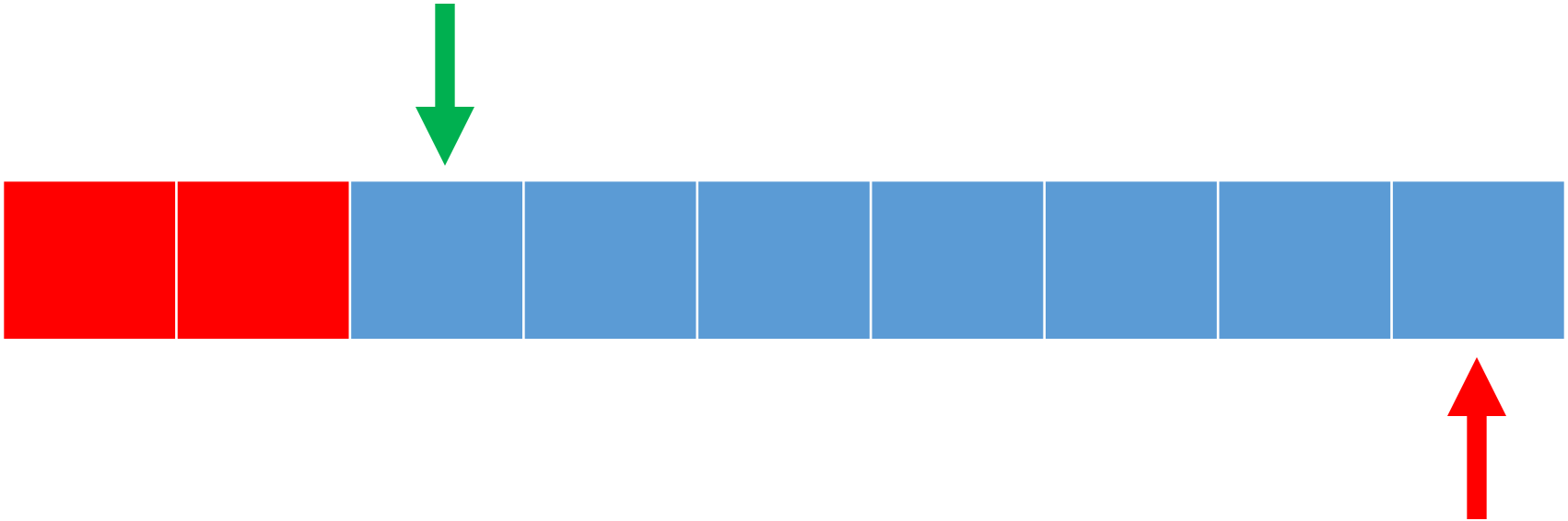
- We will simulate a Queue using a vector
- To insert into the Queue, push the state into the back of the vector as normal
- Instead of removing from the front of the Queue, simply advance an index pointer
- Instead of checking if the Queue is empty, check if the pointer is pointing to the end

BFS Vector as Queue

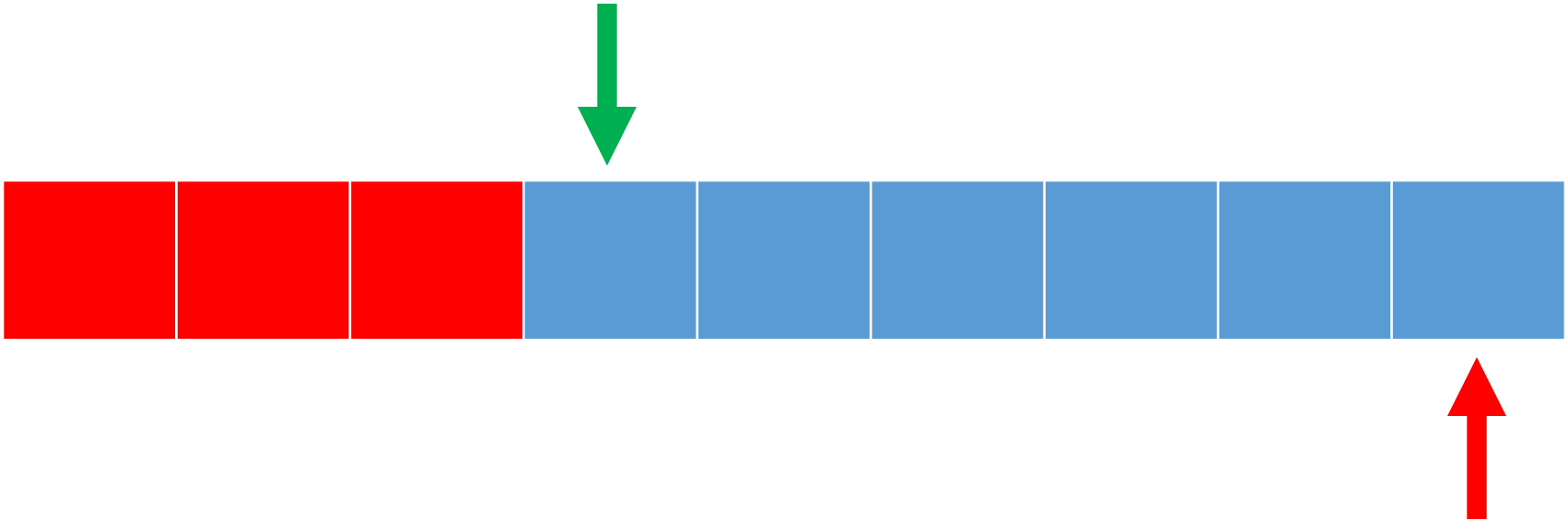
BFS Vector as Queue



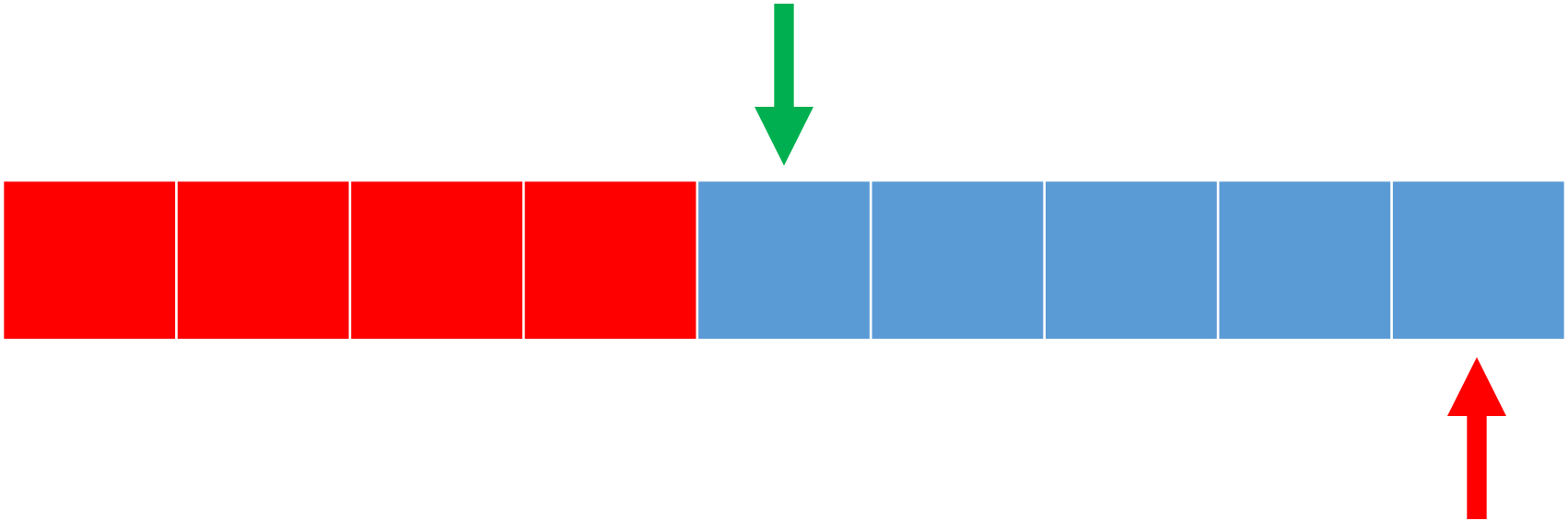
BFS Vector as Queue



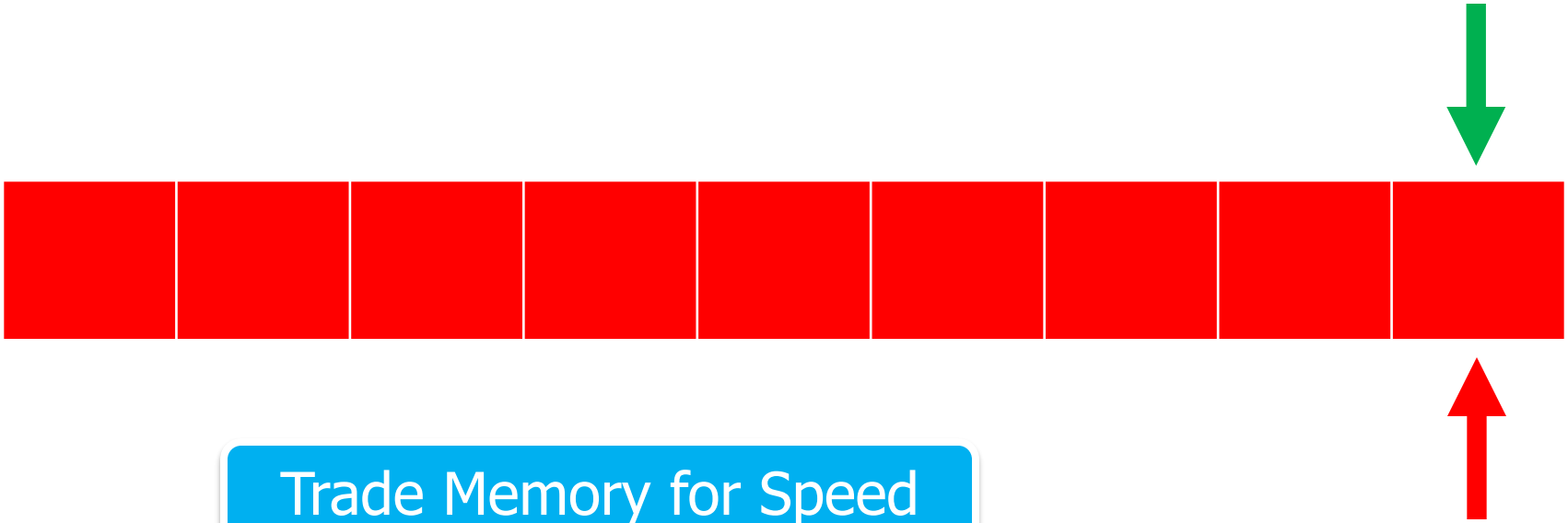
BFS Vector as Queue



BFS Vector as Queue



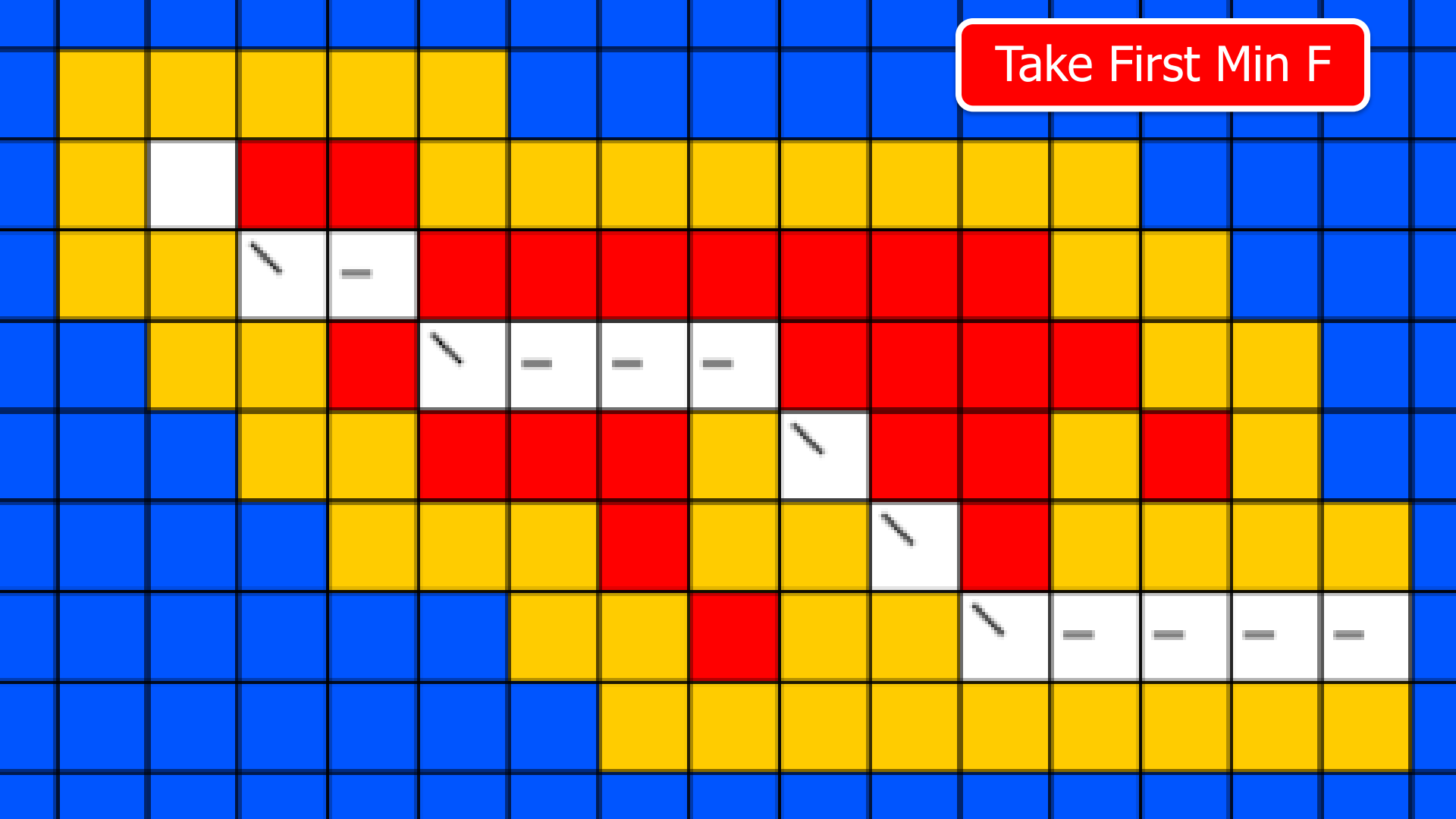
BFS Vector as Queue

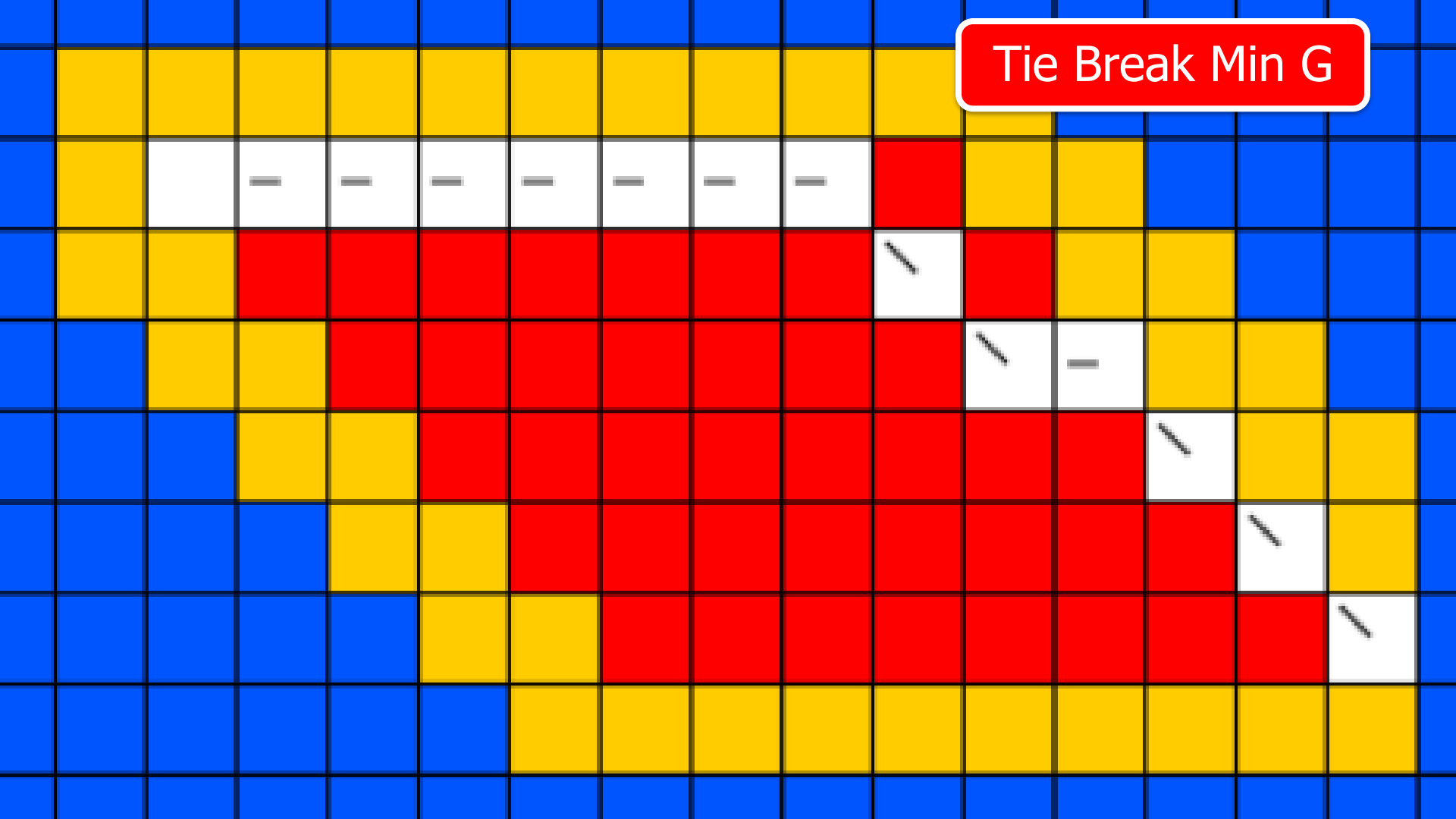


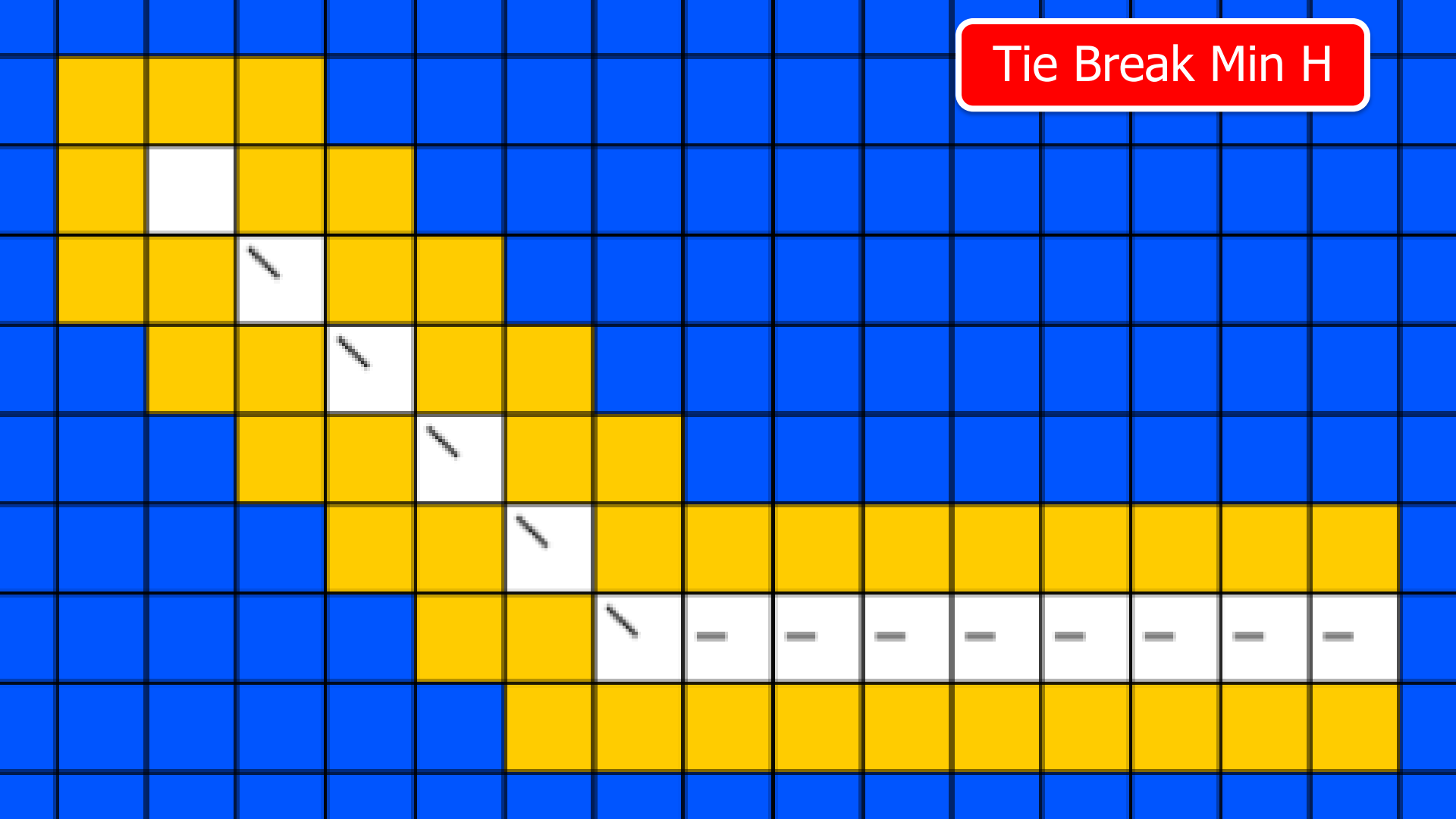
Trade Memory for Speed

Tie-Breaking F Values

- Many nodes in the open list can contain identical values for f = g + h
- How you ‘tie break’ these can have different effects on how your paths look
- Can also affect node expansions
- Consider:
  - Take first? What about G and H?


Take First Min F


Tie Break Min G


Tie Break Min H

6980: Bidirectional Search

- So far we have looked at finding a path from a start location to a goal location
- What about the other way around?
- What about both?
- Bidirectional search searches from start to goal, and from goal to start
- Search stops when the graphs intersect

Bidirectional Search

- Intuitively, we can visualize why bidirectional search may save computation
- Search algorithms work outward from a start state, creating a radius of expansion
- Which has larger total area:
  - One circle with diameter D
  - Two circles each with diameter D/2
  - These diameters are the exponent!
  - Remember, Time complexity = $O(b^d)$

Bidirectional Search

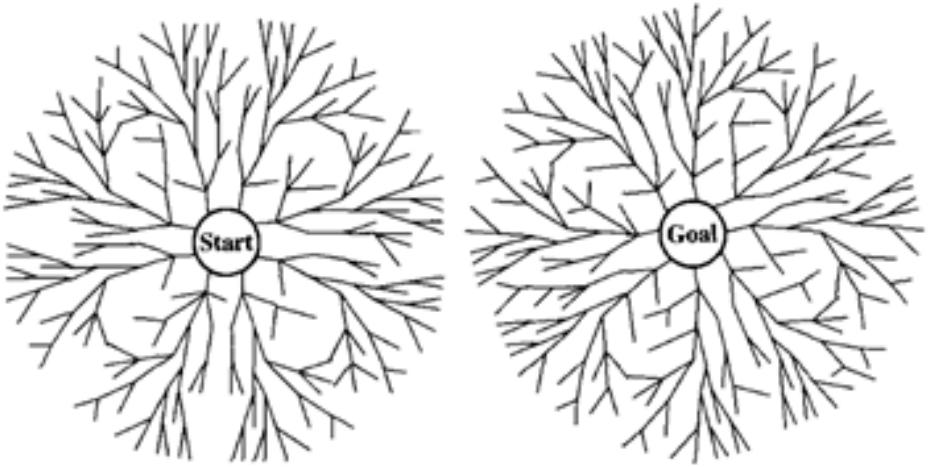
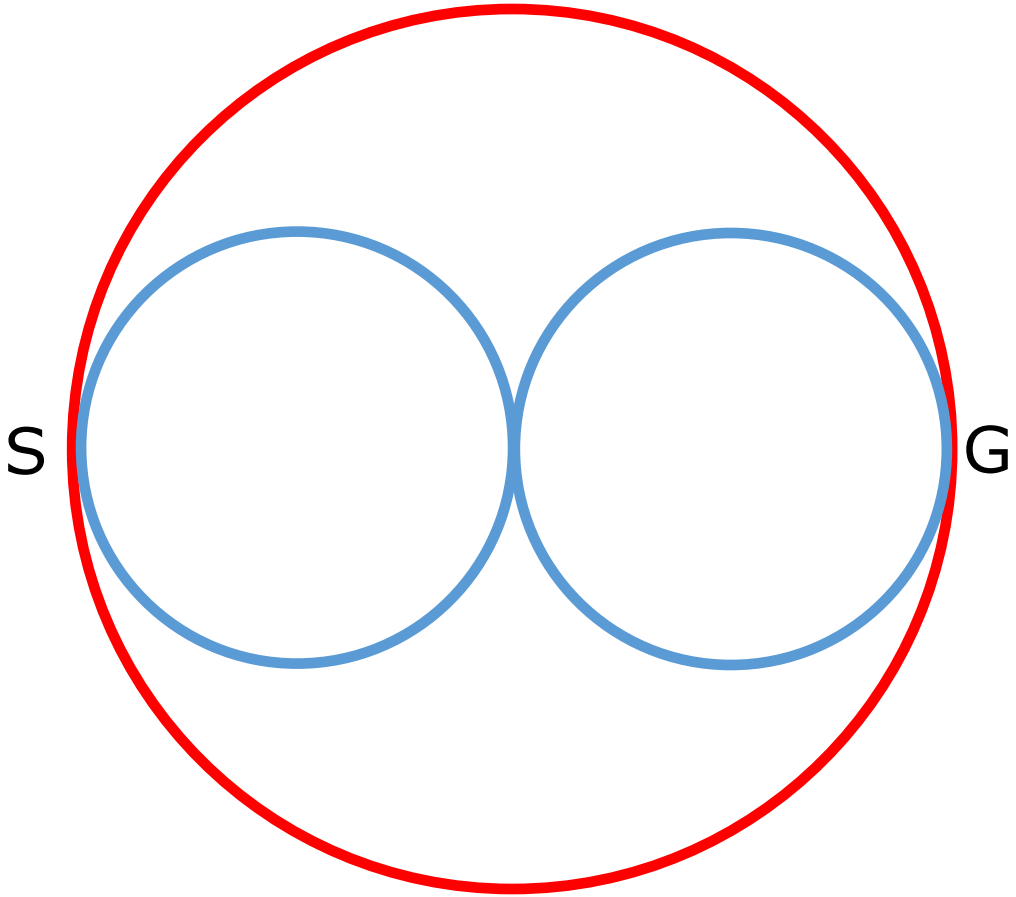
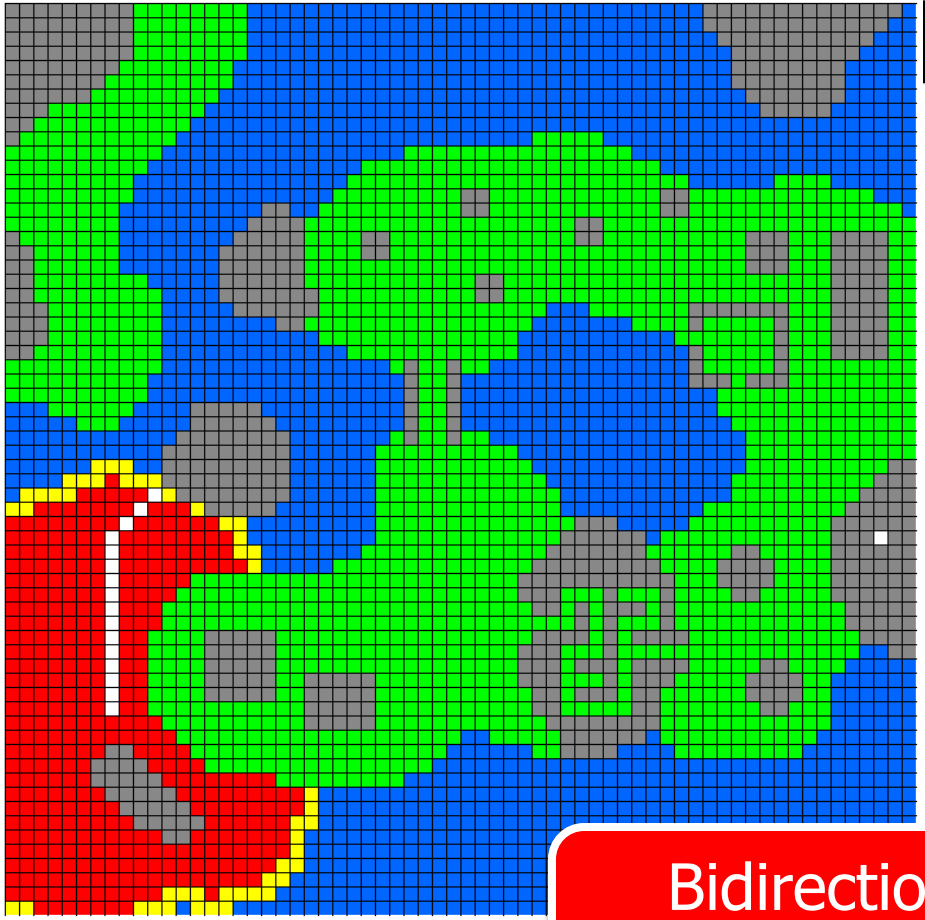


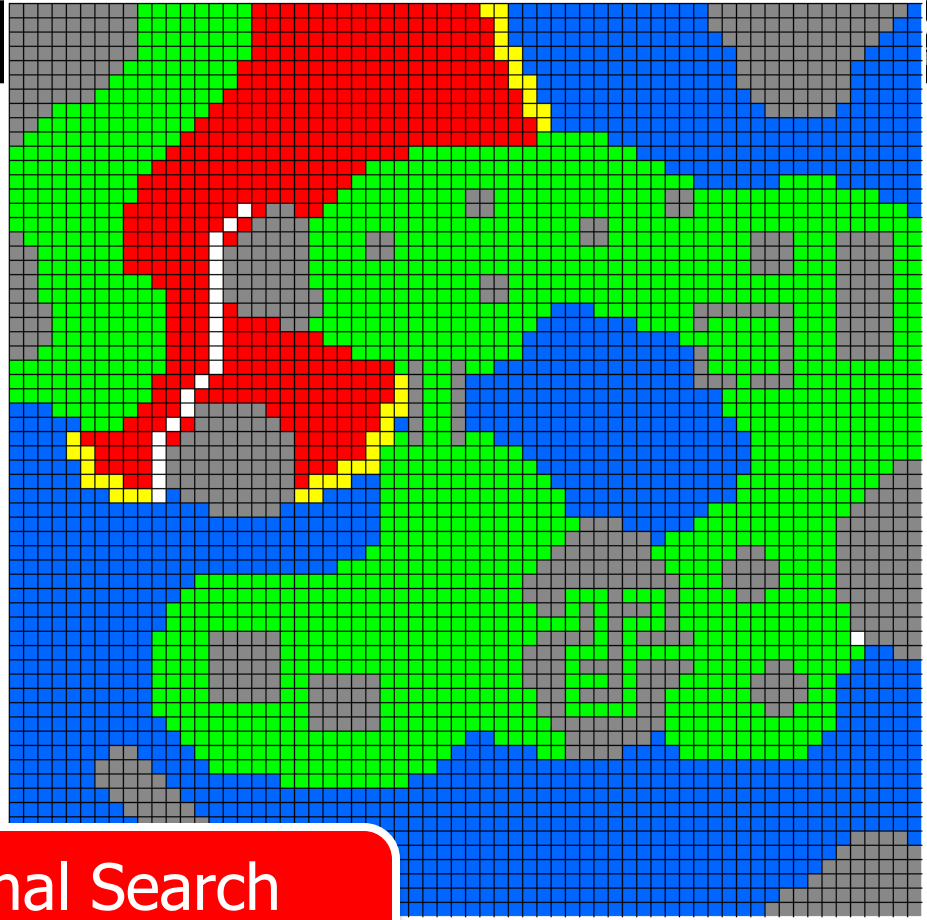
Figure 3.16 A schematic view of a bidirectional search that is about to succeed, when a branch from the start node meets a branch from the goal node.

Bidirectional Search

- Area of circle = Expanded
- Sum of small circle area less than single big circle



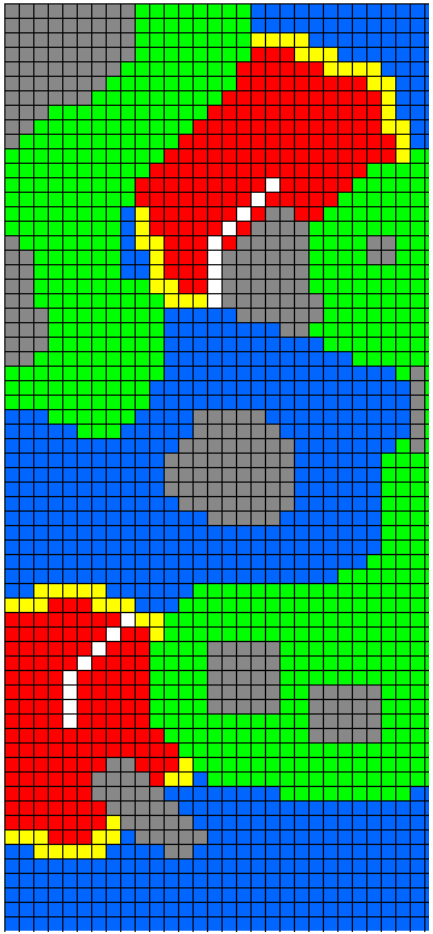




Bidirectional Search
Can Expand Fewer Nodes

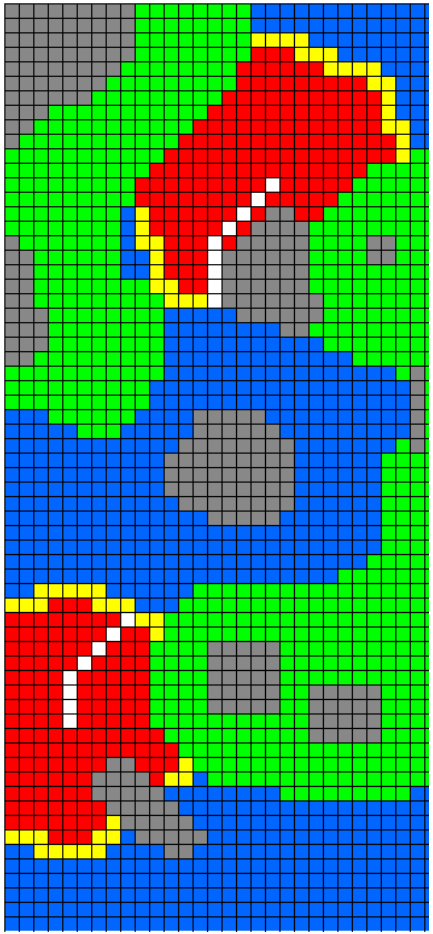
Bidirectional Search

- Algorithm modifications
  - May use two open/closed lists
  - Add start node to its open list
  - Add goal node to its closed list
  - Heuristic is ‘backward’ for 2nd list
- Can you use just one of each list?
  - Add start and goal node both to open
  - Choose min f from the single open
  - Be careful of heuristic direction!



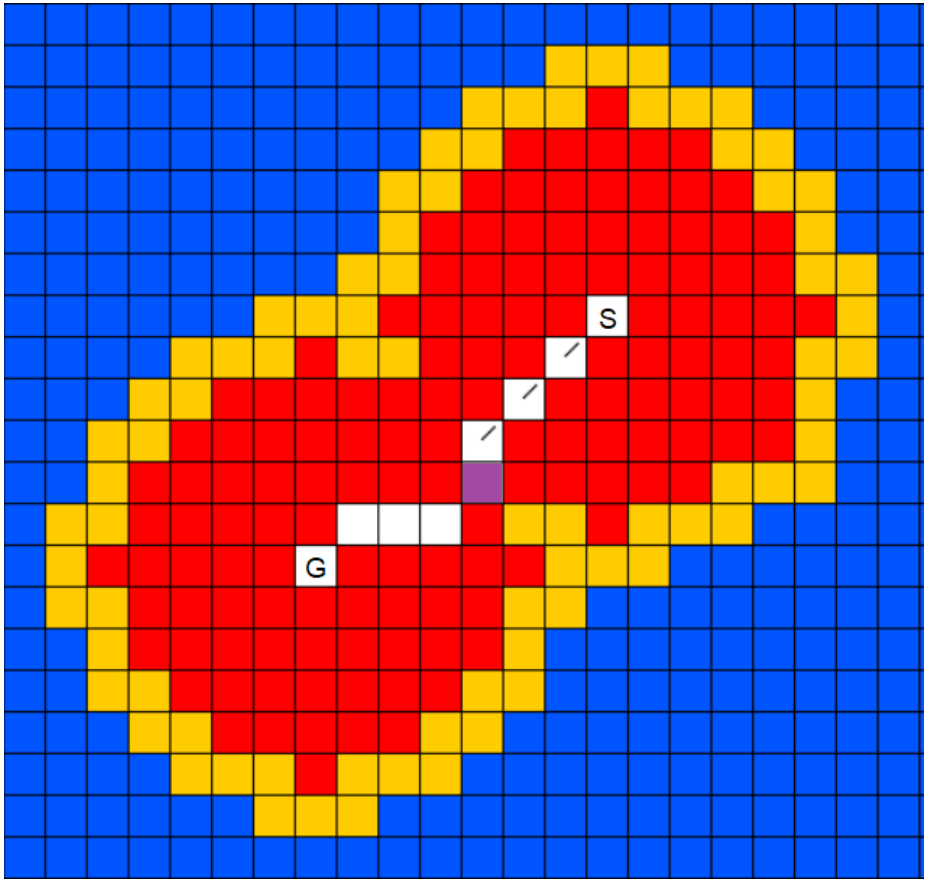
Bidirectional Search

- New ‘goal test’ condition
  - When closed lists intersect?
  - When open lists intersect?
  - Try these out in the assignment!
- Node expansion options
  - Alternate min f from both
  - Expand single min f from all
  - ‘Bounding’ strategies



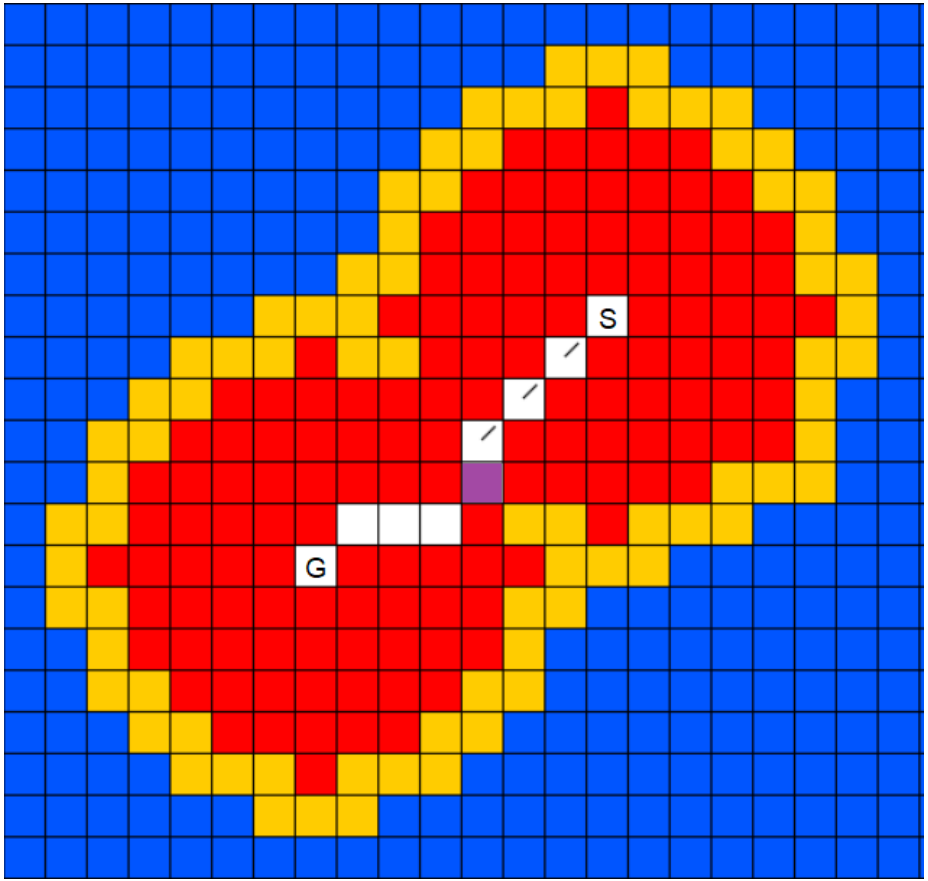
BDS Optimality

- Can we ensure optimality?
- 2 shortest paths to common state may not be the overall shortest path between start and goal
- If we stop at the first solution we find, it may not be an optimal one



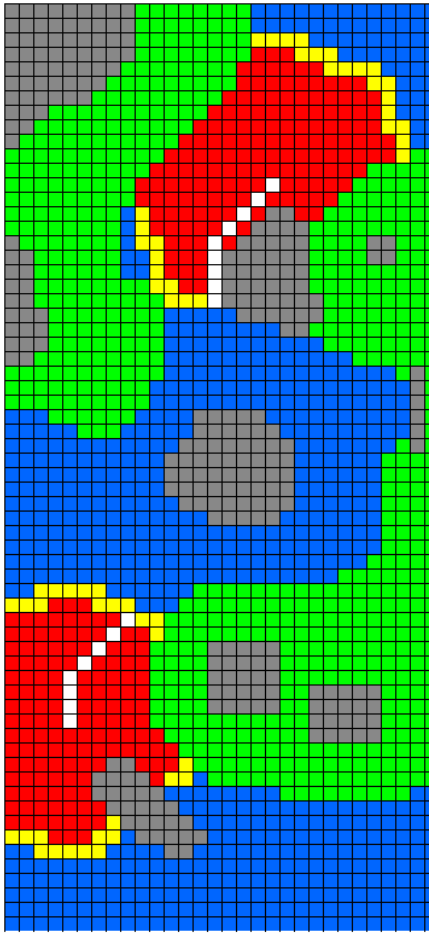
BDS Optimality

- For A*, we can be optimal
- Set initial solution cost to infinity (none found yet)
- When you find a new better solution, set cost
- Stop search when next node's f cost greater than cost of best solution so far
- In practice doesn't save time often, but is optimal



Bidirectional Search

- A2 Requirements
  - Use two open / closed lists
  - Add start to ‘start’ open list, goal to ‘goal’ open list
  - Return contents of both open/closed for vis
  - Animated search should see some nodes being selected from both lists during search
  - Path should be formed when both lists ‘meet’
  - Should show a path if one is it’s possible
  - Optimal path / tests not necessary for most marks
  - Default tests pass = 2%



Bidirectional Search Paper

- A Brief History and Recent Achievements in Bidirectional Search
  - Nathan Sturtevant, Ariel Felner
  - https://webdocs.cs.ualberta.ca/~nathanst/papers/Sturtevant18smt.pdf
- This will NOT be tested for the exams, just for your information / interest

Bidirectional Search Demo

http://qiao.github.io/PathFinding.js/visual/