COMP30030: Introduction to Artificial Intelligence

Neil Hurley

School of Computer Science
University College Dublin
neil.hurley@ucd.ie

September 20, 2018

1 Problem Solving by Search

- Uninformed Search

Blind (Uninformed) Search

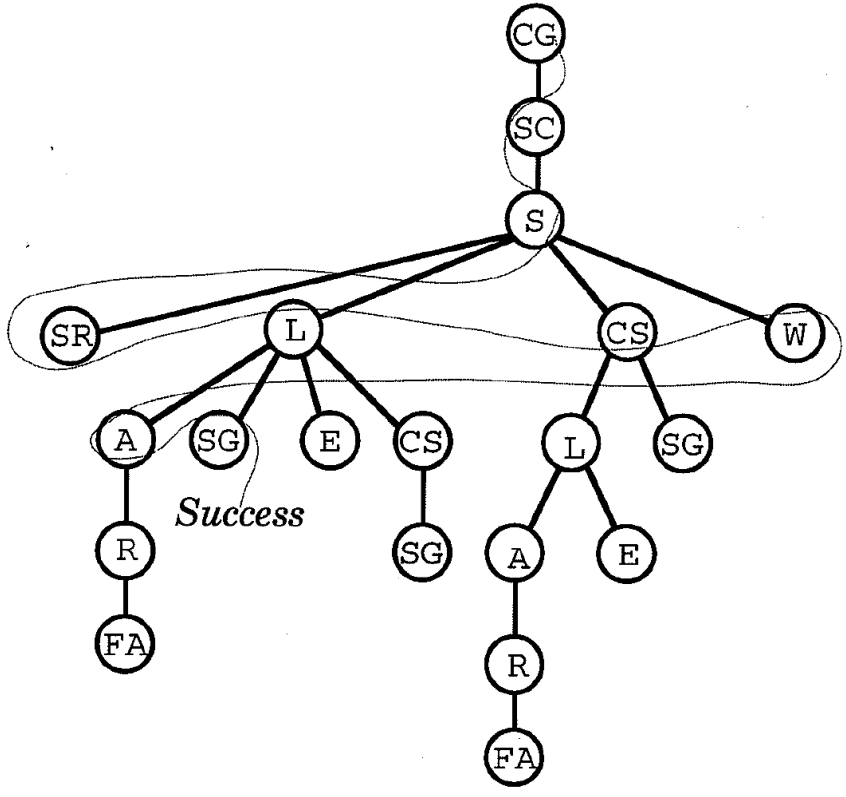
- Blind search methods have no way of judging which partial path is likely to lead to a solution and which is not so they try to **systematically follow all paths** in the hope that one will succeed.
- Blind search methods differ only in the way that they add new paths to the search queue.
  - **Depth- First Search**
  - **Breadth- First Search**

BlindSearch (initial,goal,queuing-fn)

```
Form a one element queue consisting of a
zero-length path that contains only the
initial state (the root).

Until the first path in the queue
terminates at the goal state or the queue
is empty.

     Remove the first path from the queue

     Create new paths by extending the first
     path to all successor states of its
     terminal state

     Reject all new paths with cycles

     * Add the new paths to the queue using
     to the queuing function

If the goal state is found the announce
success; otherwise announce failure
```

Breadth-First Search

- Breadth-first search traverses a tree in a left-to-right, top-to-bottom fashion. It does this by expanding all of the nodes on a particular level before moving down to the next level.
- The search is complete. It is also optimal so long as the solution is a non-decreasing function of node depth.



- **Queuing Function:** Queue new paths at end of queue. (FIFO)

Time complexity	Space complexity	Complete?	Optimal?
$O(b^d)$	$O(b^d)$	Yes	Yes

Evaluating Complexity

- **Time Complexity**
  - At level 0, we generate one node, the start node. At level 1, there are *b* successors of the start node. Each of these successors has *b* successors, so at level 2 there are b^2 nodes. Each of the nodes at level 2 has b successors, so there are b^3 nodes at level 3. If the first solution path is of length d, then the maximum number of nodes we would generate is:

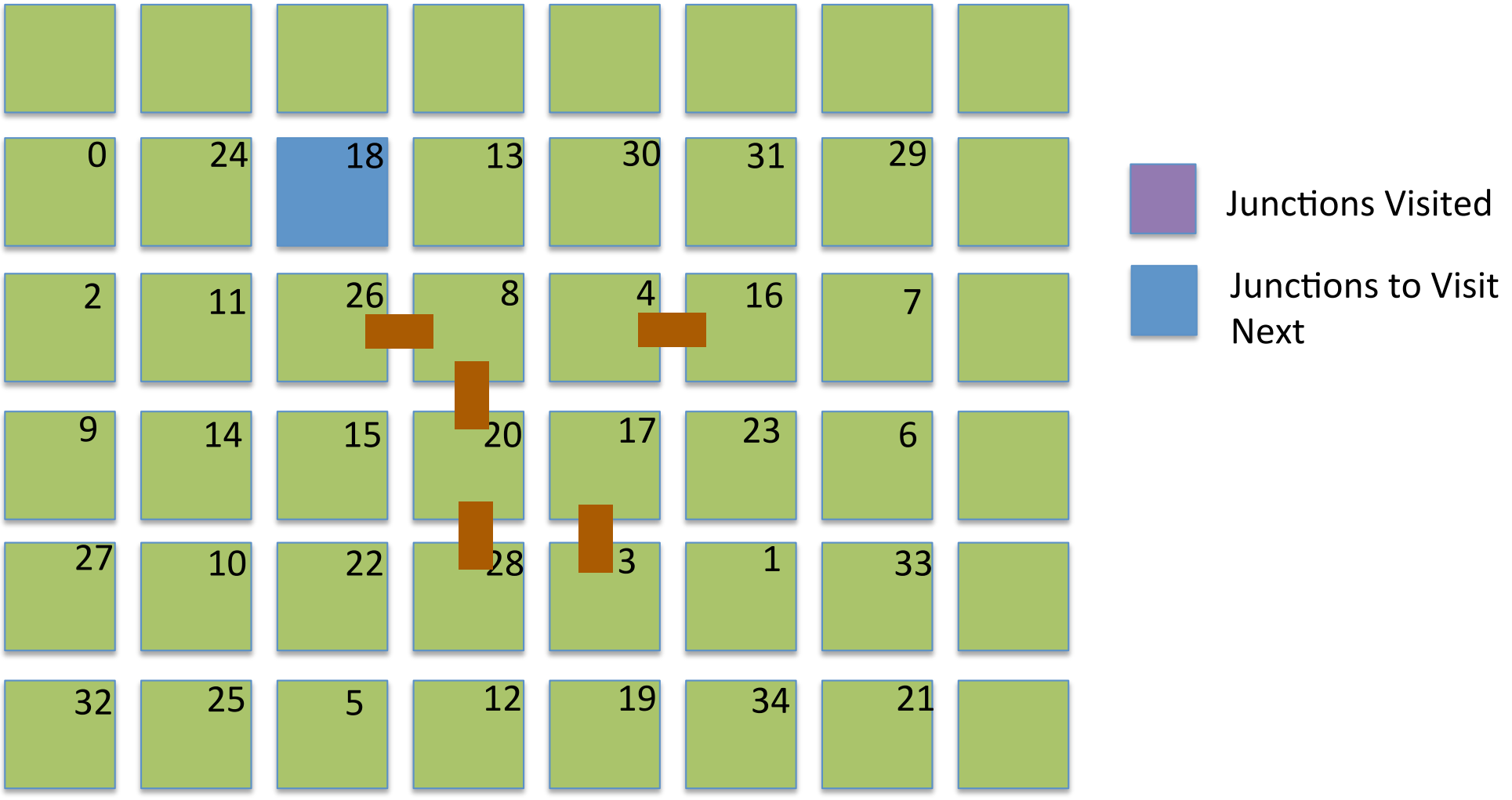
    $$1 + b + b^2 + b^3 + \ldots + b^{d-1} + b^d$$

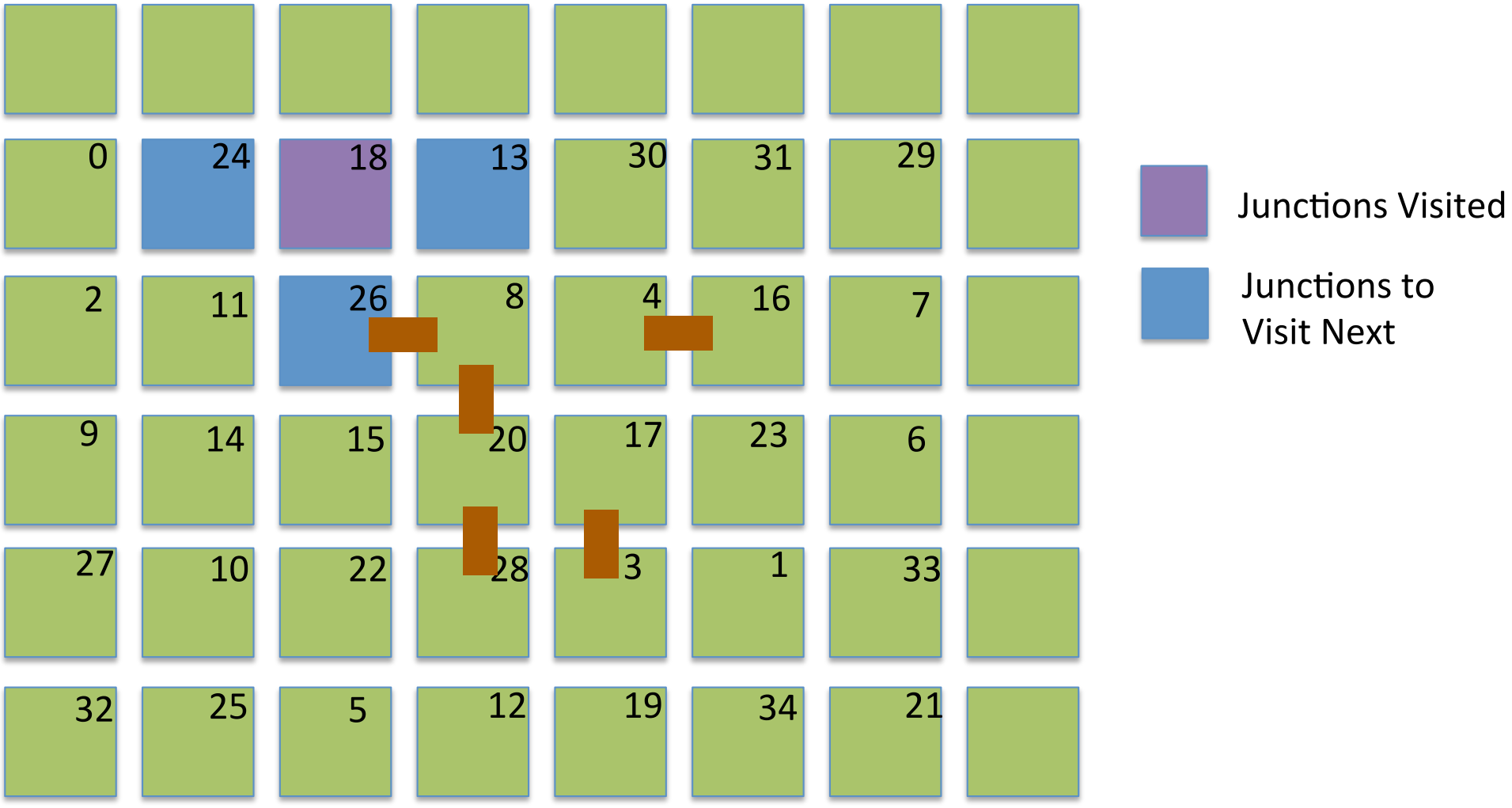
  - The worst case time complexity is therefore $O(b^d)$

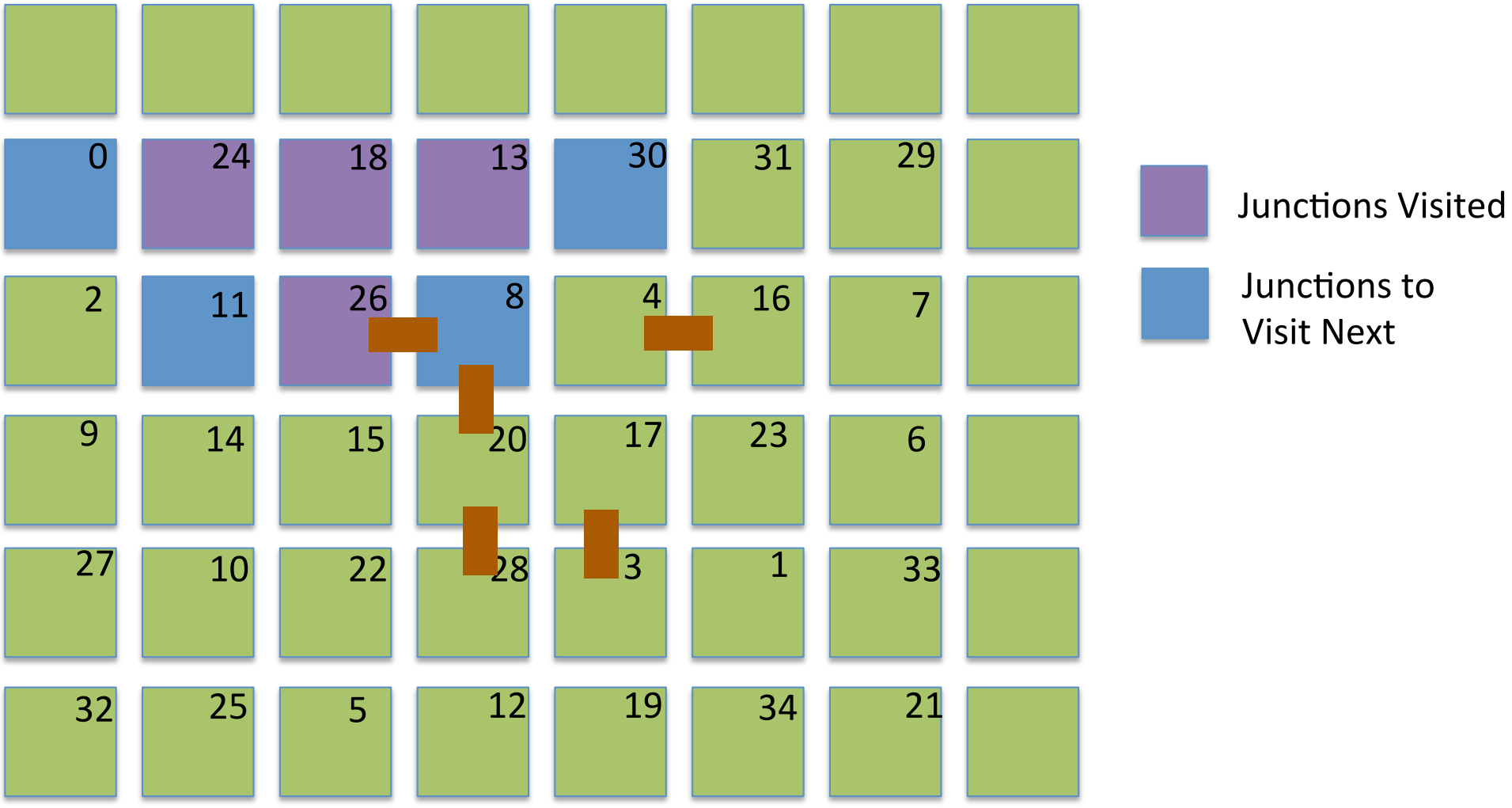
Evaluating Complexity

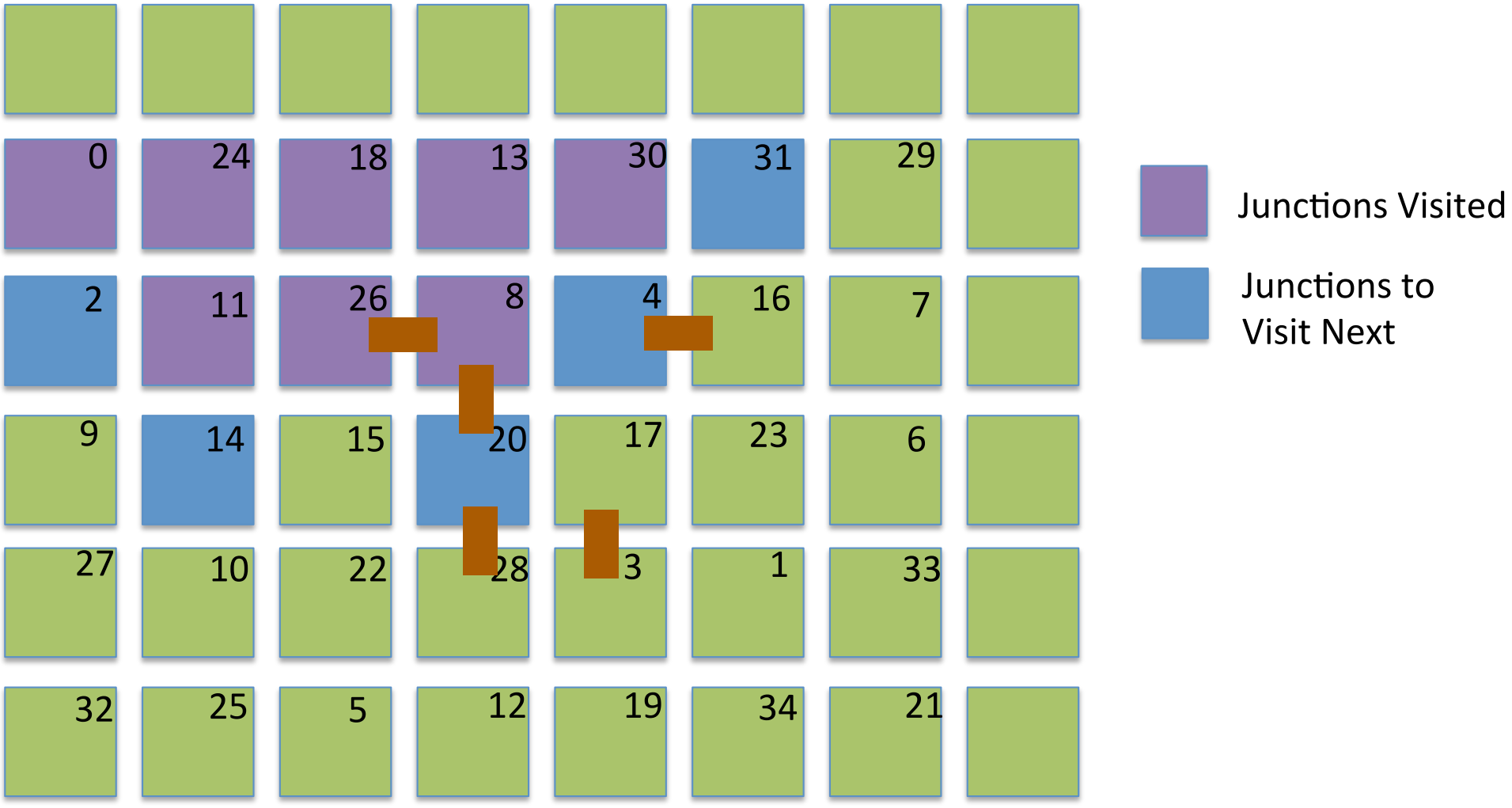
- **Space Complexity**
  - Same as time complexity. We have to maintain all of the tree that we have generated so far so that the path can be extended level by level.
  - Exponential time complexity is very bad news, but exponential space complexity is a disaster.
  - This search strategy can only be used on trivial search spaces.

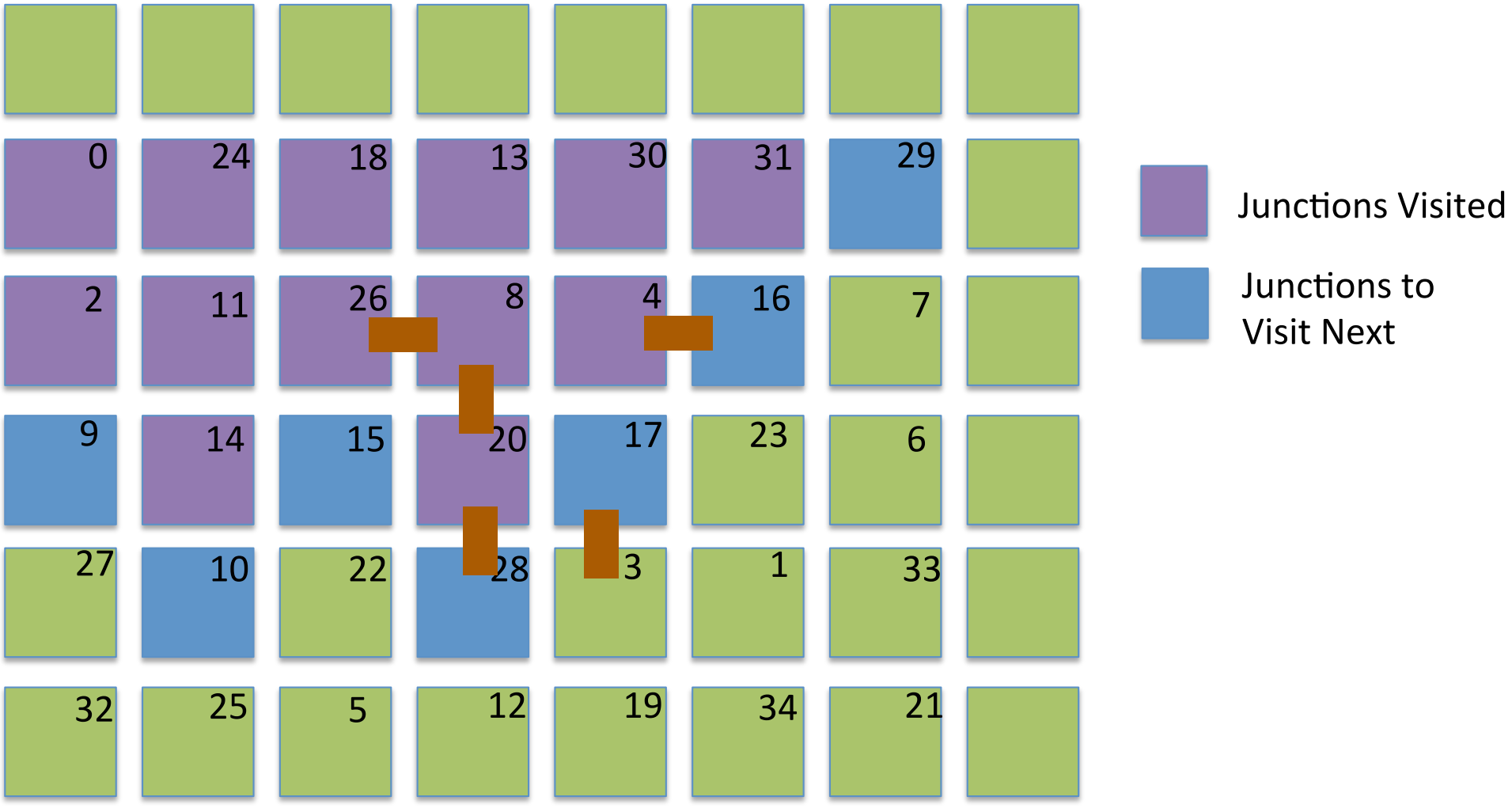
Pacman navigates from junction 18 to junction 3, using breadth first search.

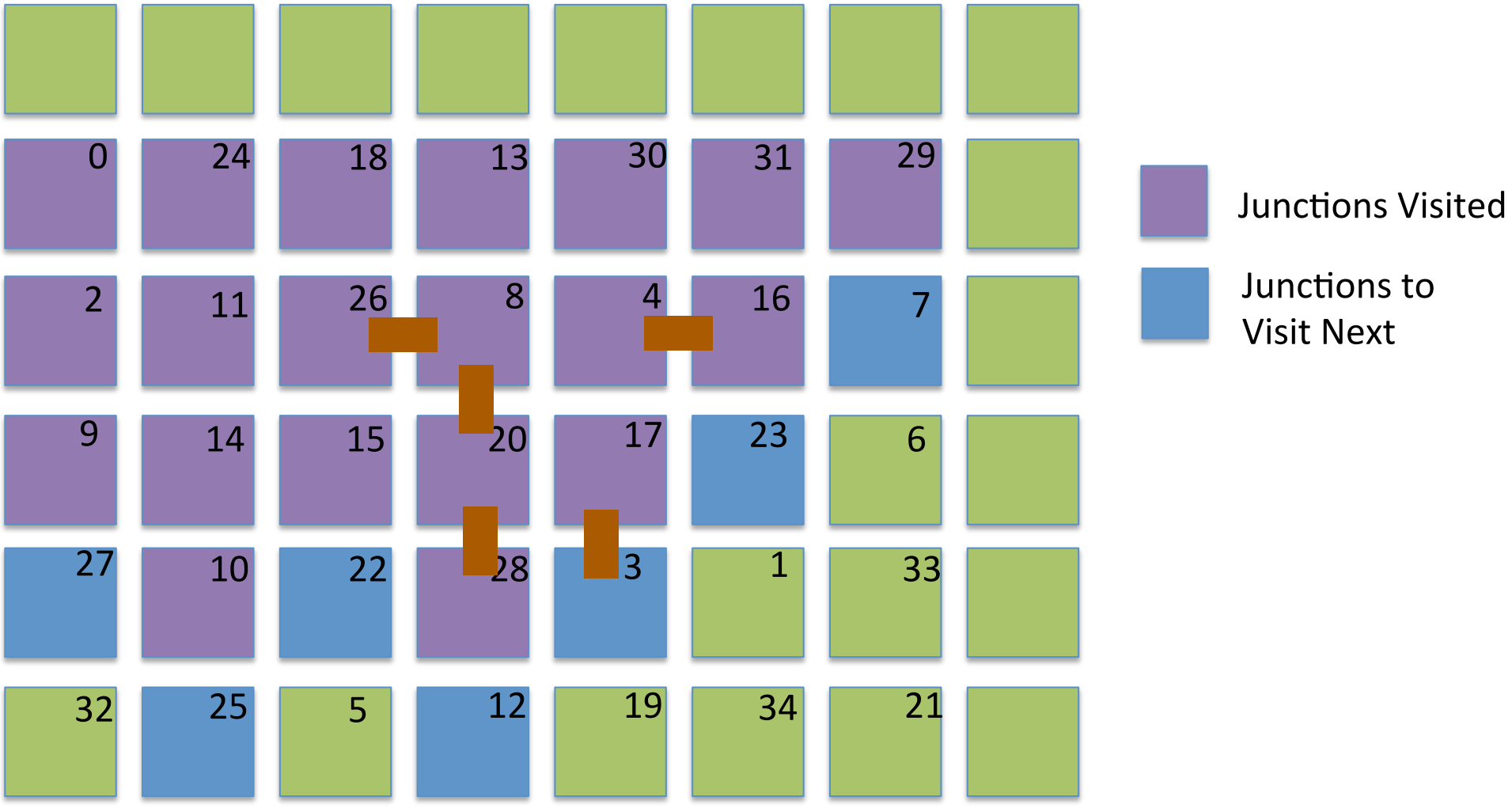

0 24 18 13 30 31 29
2 11 26 8 4 16 7
9 14 15 20 17 23 6
27 10 22 28 3 1 33
32 25 5 12 19 34 21
Junctions Visited
Junctions to Visit Next


0
24
18
13
30
31
29
2
11
26
8
4
16
7
9
14
15
20
17
23
6
27
10
22
28
3
1
33
32
25
5
12
19
34
21
Junctions Visited
Junctions to Visit Next

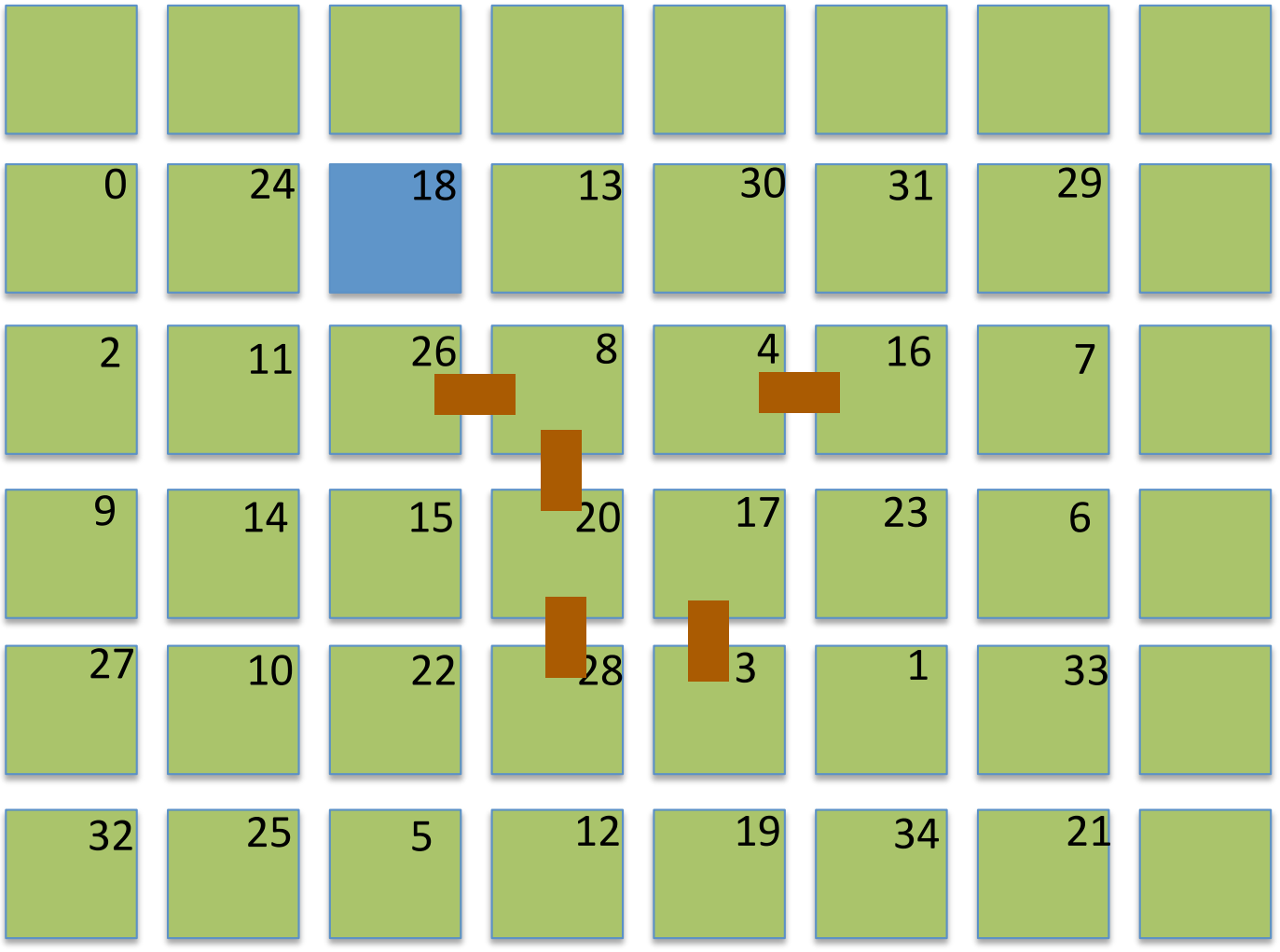

0
24
18
13
30
31
29
2
11
26
8
4
16
7
9
14
15
20
17
23
6
27
10
22
28
3
1
33
32
25
5
12
19
34
21
Junctions Visited
Junctions to Visit Next


0
24
18
13
30
31
29
2
11
26
8
4
16
7
9
14
15
20
17
23
6
27
10
22
28
3
1
33
32
25
5
12
19
34
21
Junctions Visited
Junctions to Visit Next

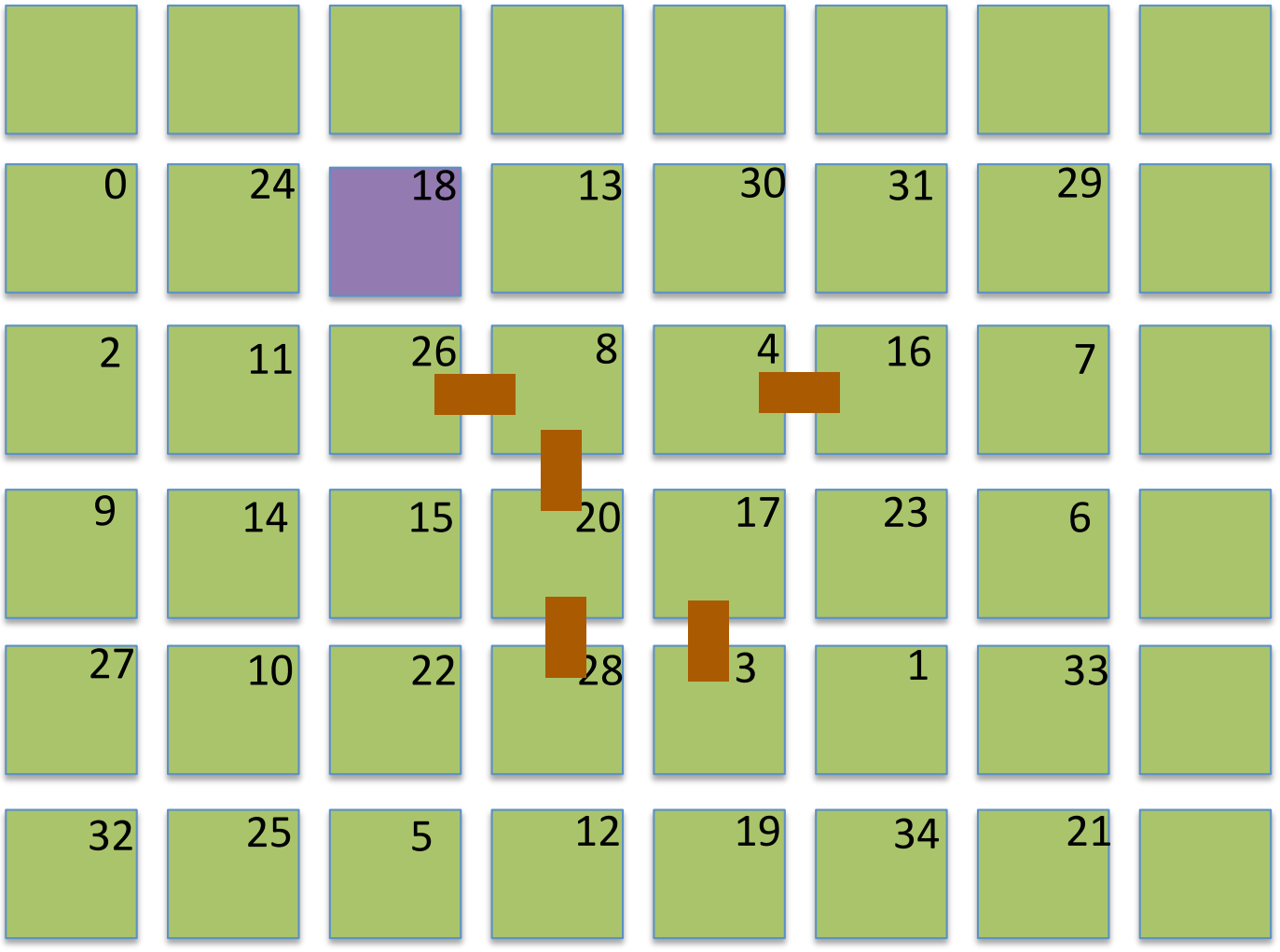

0
24
18
13
30
31
29
2
11
26
8
4
16
7
9
14
15
20
17
23
6
27
10
22
28
3
1
33
32
25
5
12
19
34
21
Junctions Visited
Junctions to Visit Next


0 24 18 13 30 31 29
2 11 26 8 4 16 7
9 14 15 20 17 23 6
27 10 22 28 3 1 33
32 25 5 12 19 34 21
Junctions Visited
Junctions to Visit Next

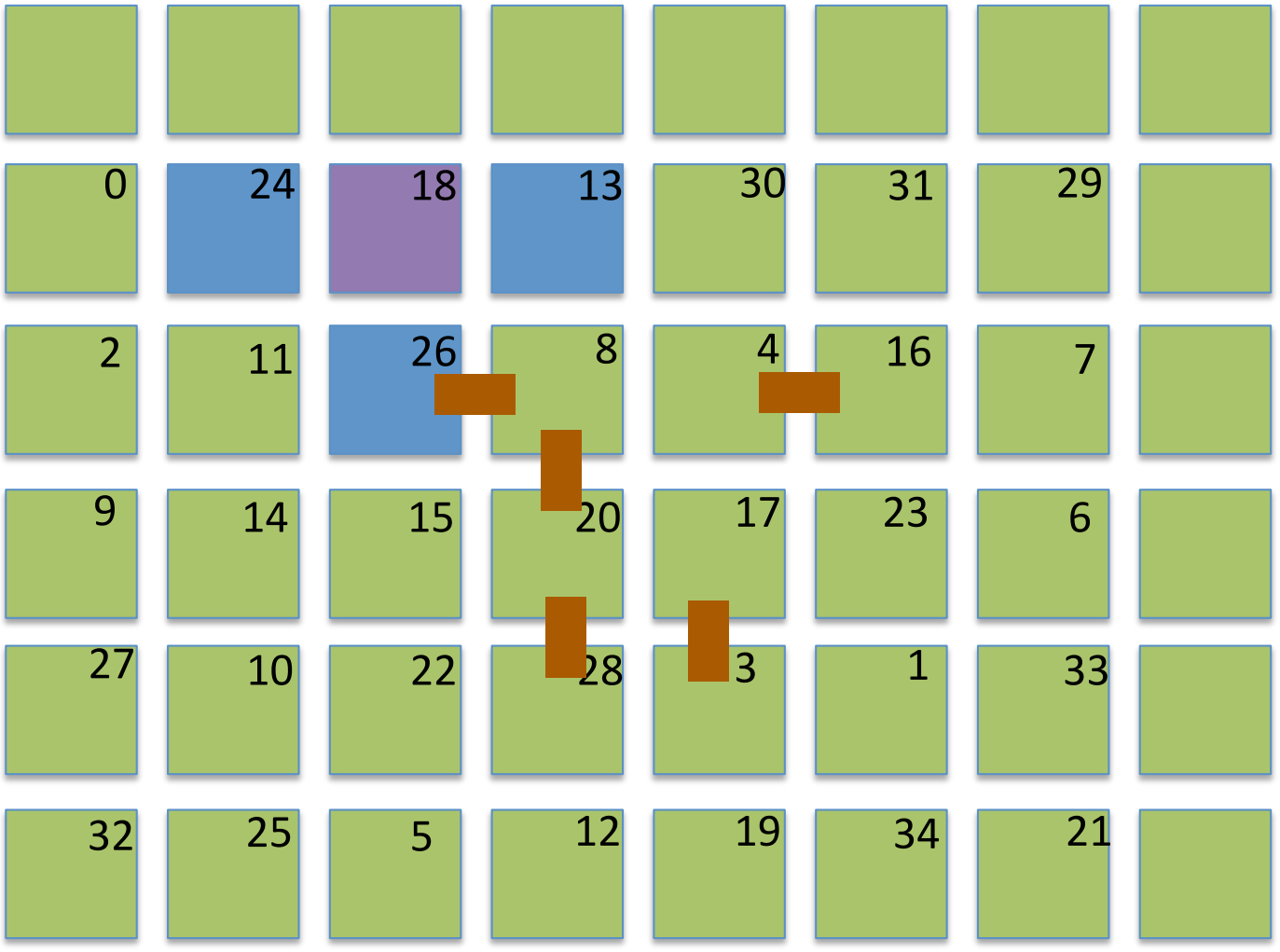
Let's see how this search is managed by a FIFO queue representing the open list of fringe nodes ...


0 24 18 13 30 31 29
2 11 26 8 4 16 7
9 14 15 20 17 23 6
27 10 22 28 3 1 33
32 25 5 12 19 34 21

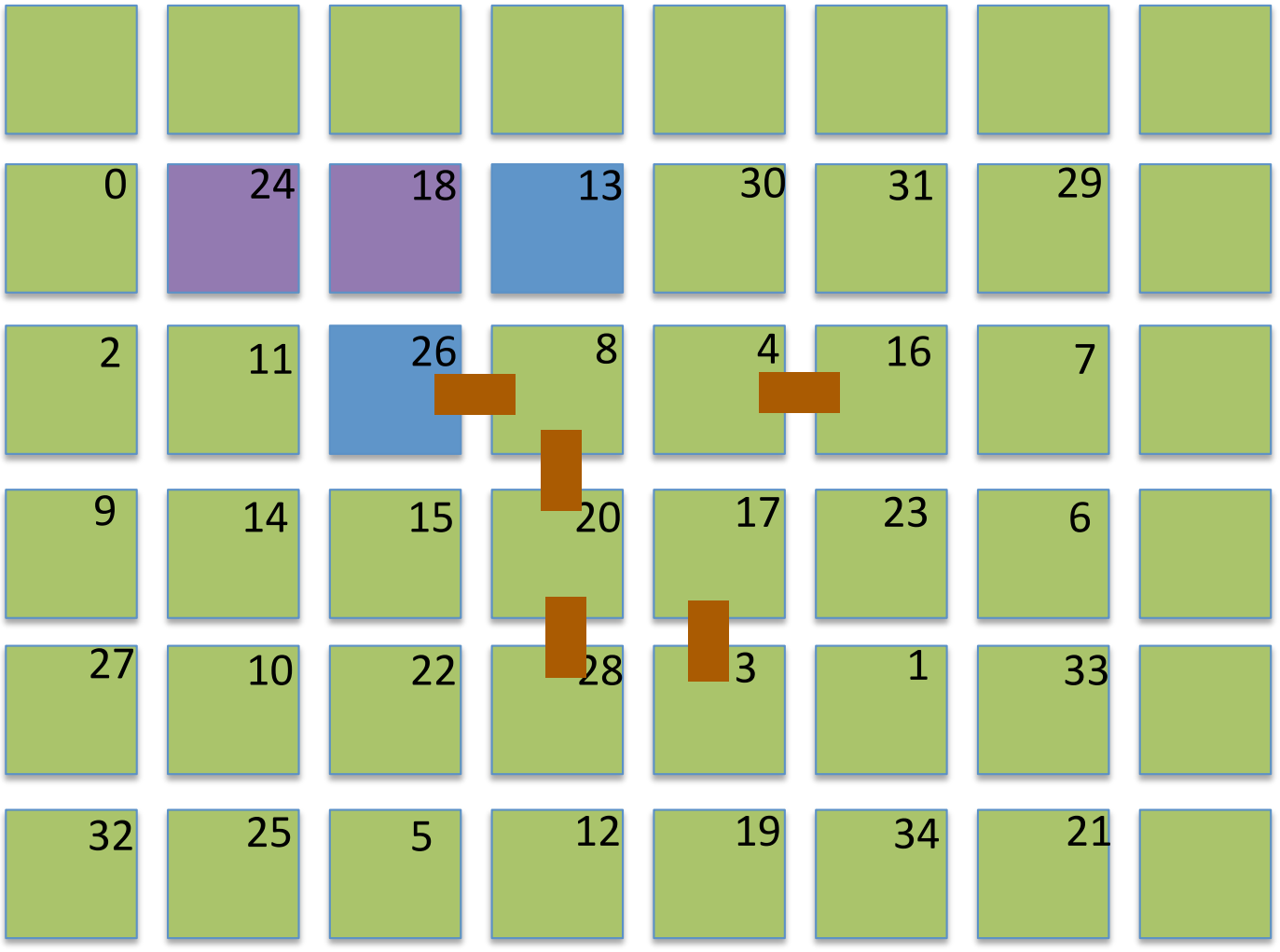
{18}


0
24
18
13
30
31
29
2
11
26
8
4
16
7
9
14
15
20
17
23
6
27
10
22
28
3
1
33
32
25
5
12
19
34
21

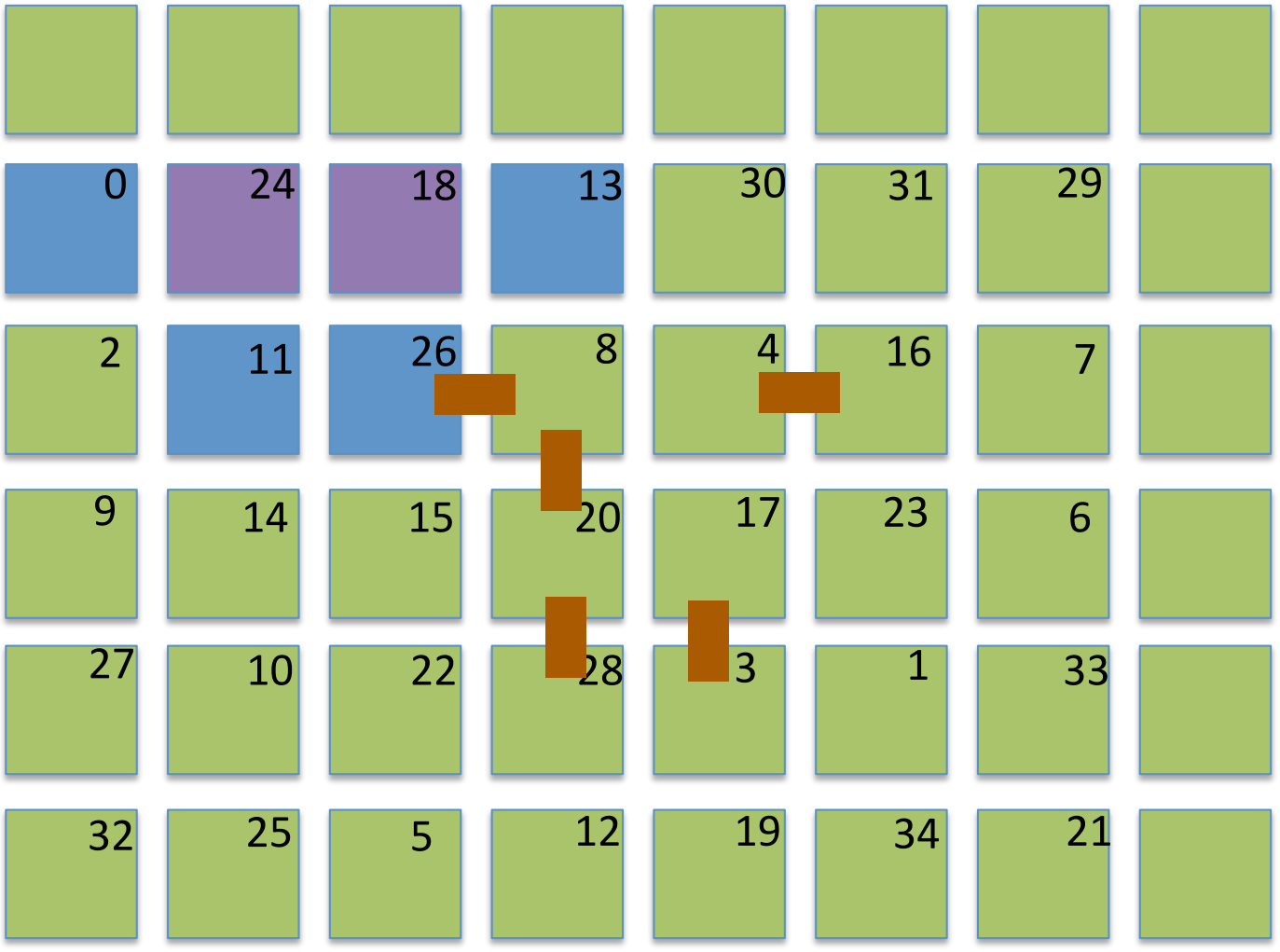
18 ← {}


0 24 18 13 30 31 29
2 11 26 8 4 16 7
9 14 15 20 17 23 6
27 10 22 28 3 1 33
32 25 5 12 19 34 21

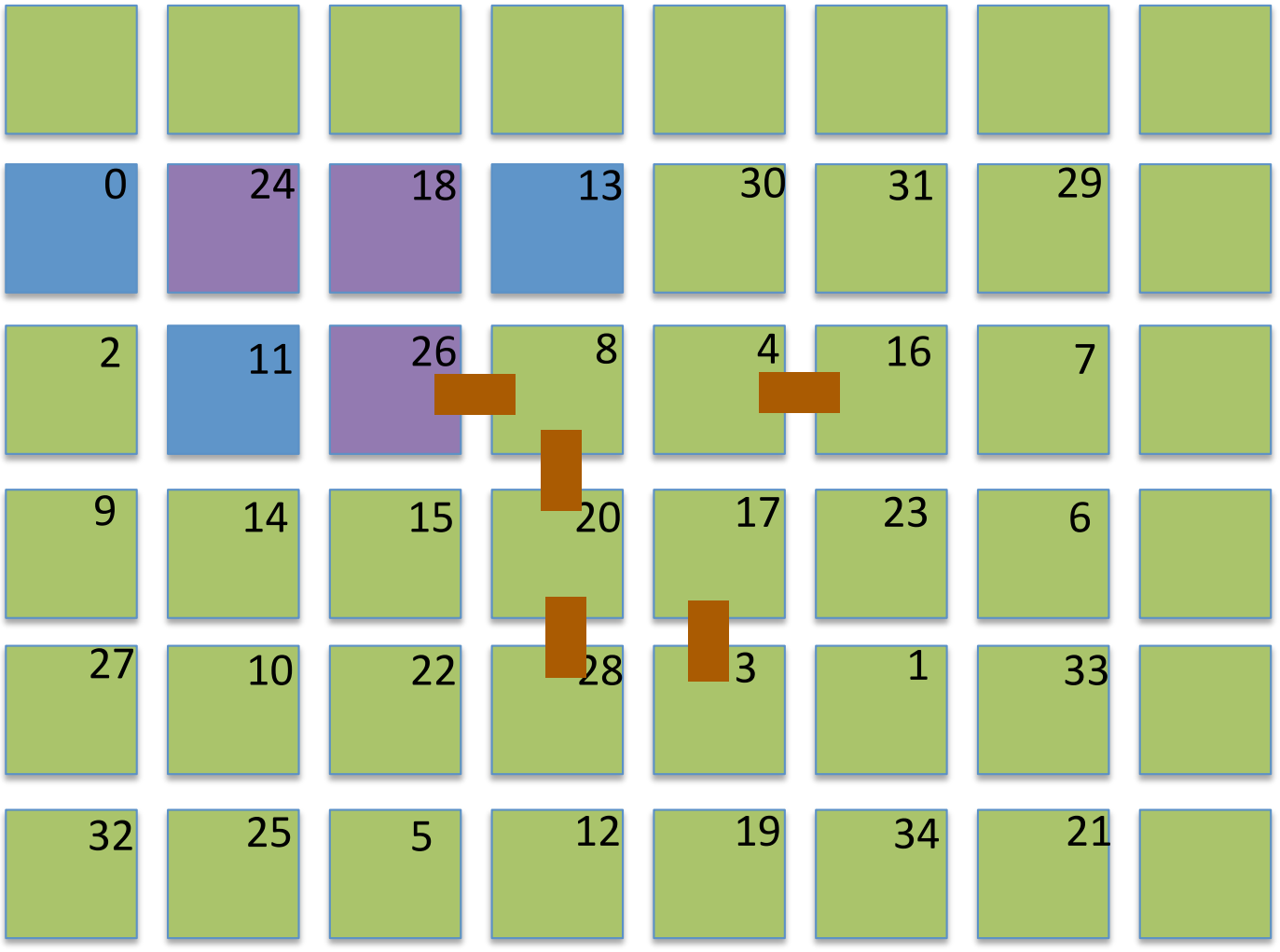
18 ← {24,26,13}


0
24
18
13
30
31
29
2
11
26
8
4
16
7
9
14
15
20
17
23
6
27
10
22
28
3
1
33
32
25
5
12
19
34
21

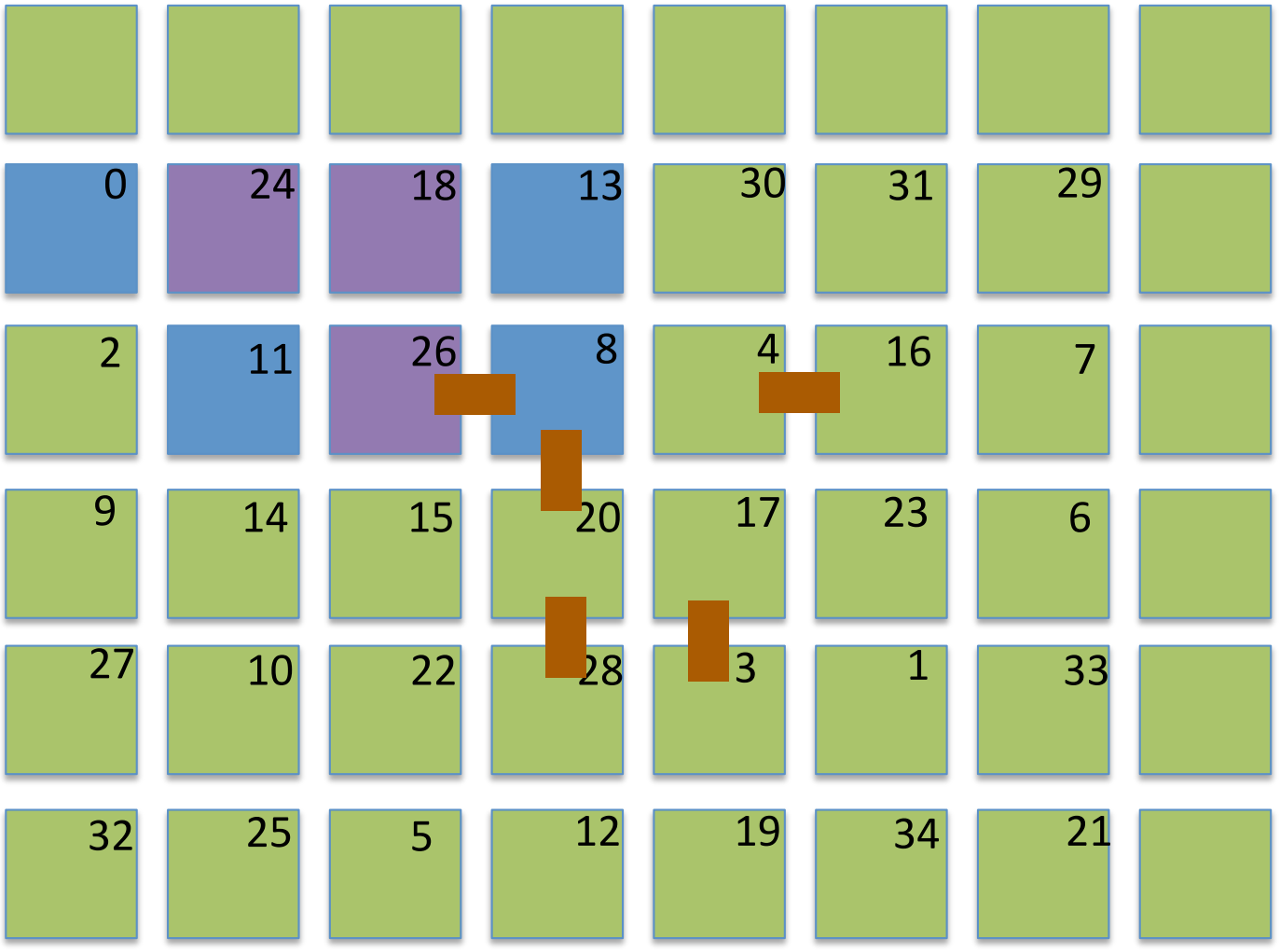
24 ← {26,13}


0 24 18 13 30 31 29
2 11 26 8 4 16 7
9 14 15 20 17 23 6
27 10 22 28 3 1 33
32 25 5 12 19 34 21

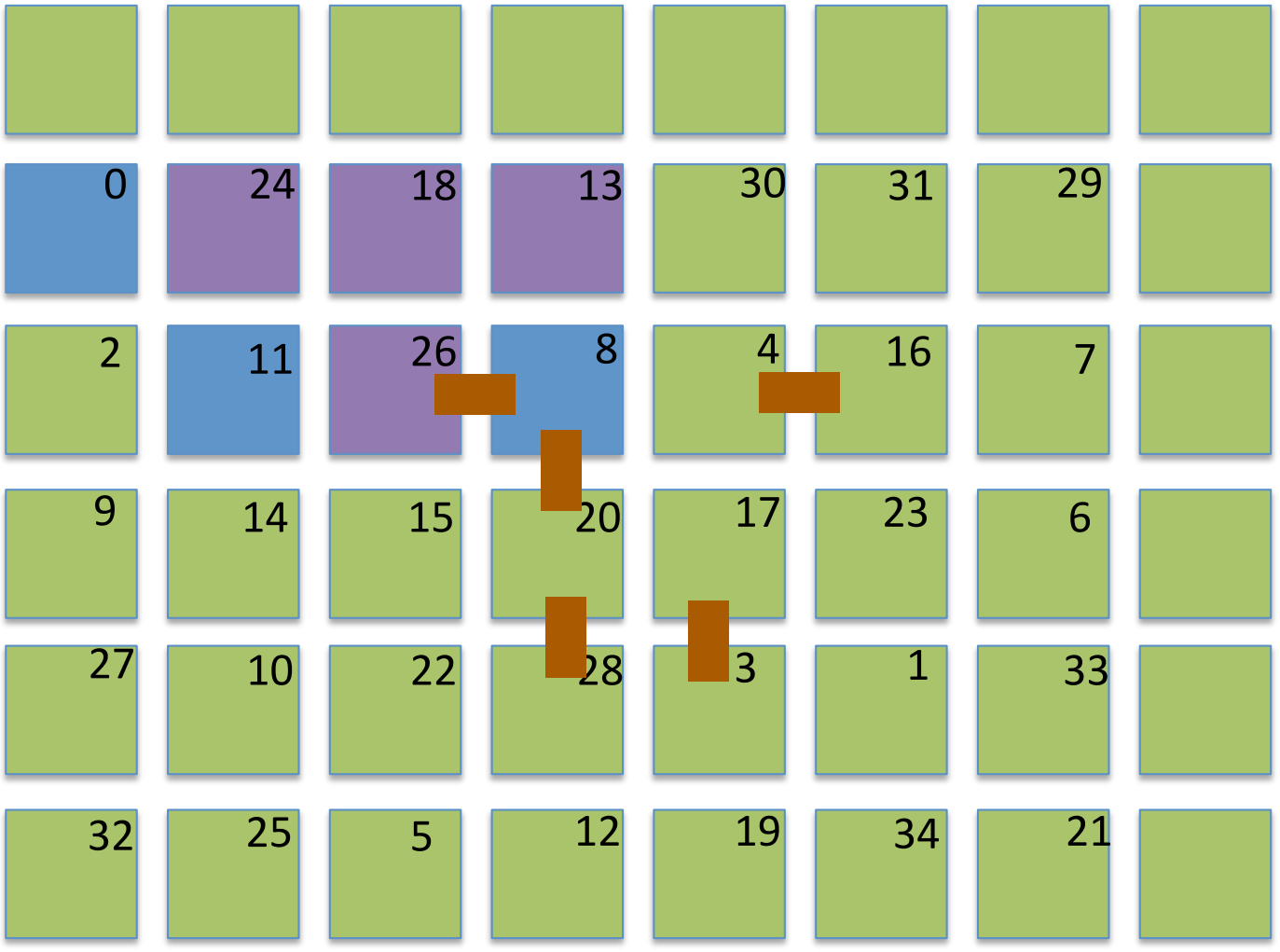
24 ← {26,13,0,11}


0 24 18 13 30 31 29
2 11 26 8 4 16 7
9 14 15 20 17 23 6
27 10 22 28 3 1 33
32 25 5 12 19 34 21

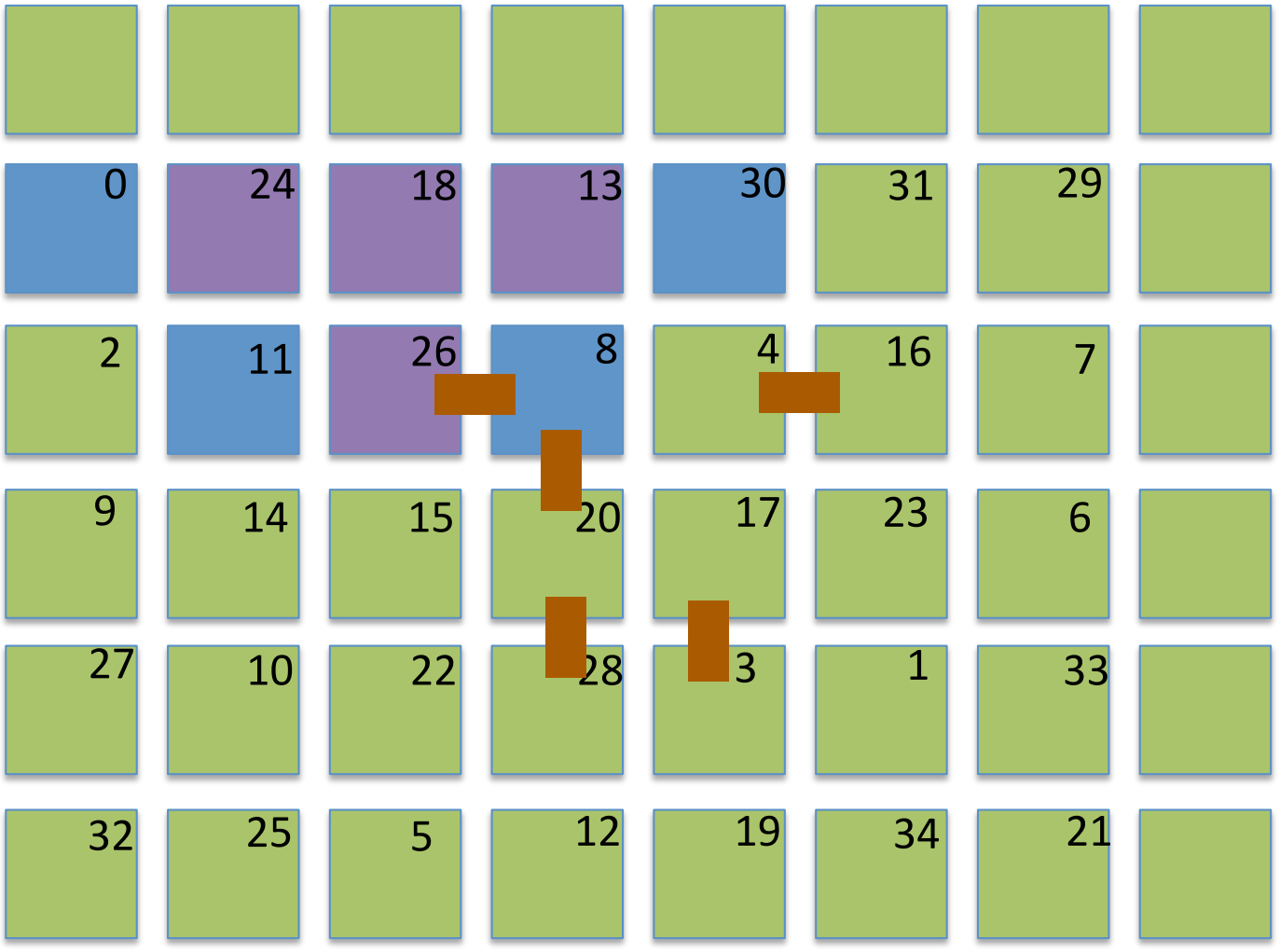
26 ← {13,0,11}


0 24 18 13 30 31 29
2 11 26 8 4 16 7
9 14 15 20 17 23 6
27 10 22 28 3 1 33
32 25 5 12 19 34 21

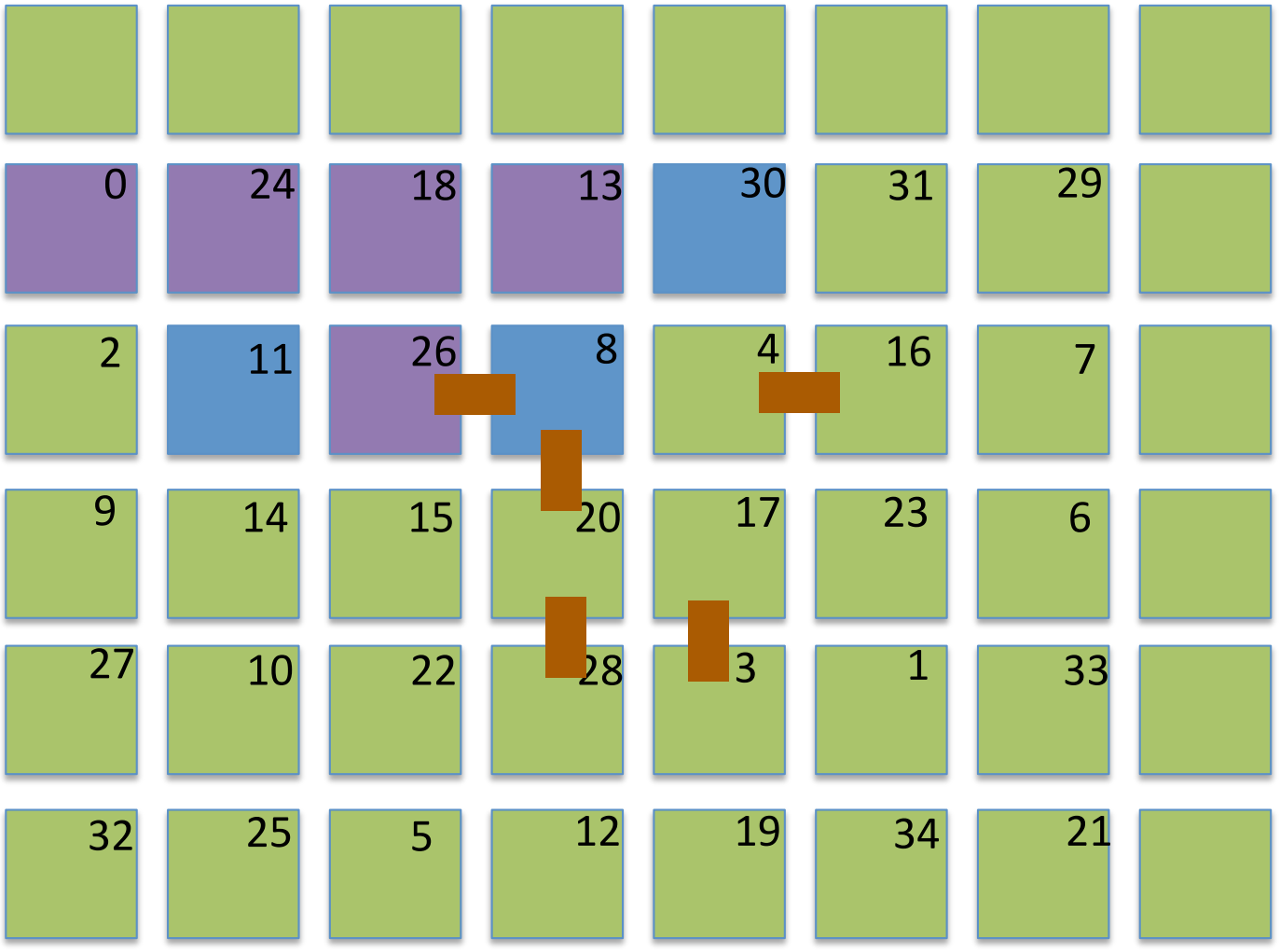
26 ← {13,0,11,8}


0 24 18 13 30 31 29
2 11 26 8 4 16 7
9 14 15 20 17 23 6
27 10 22 28 3 1 33
32 25 5 12 19 34 21

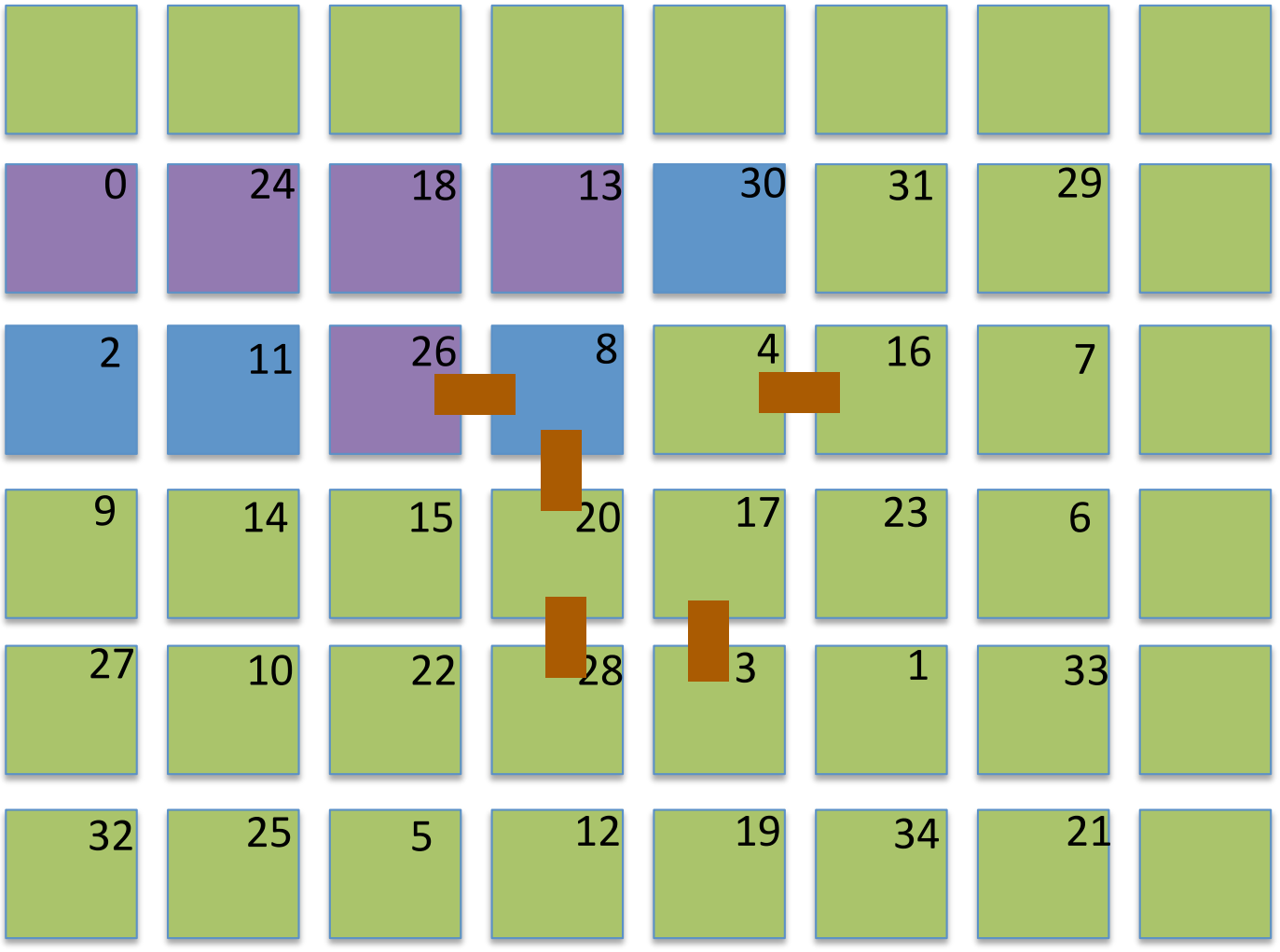
13 ← {0,11,8}


0 24 18 13 30 31 29
2 11 26 8 4 16 7
9 14 15 20 17 23 6
27 10 22 28 3 1 33
32 25 5 12 19 34 21

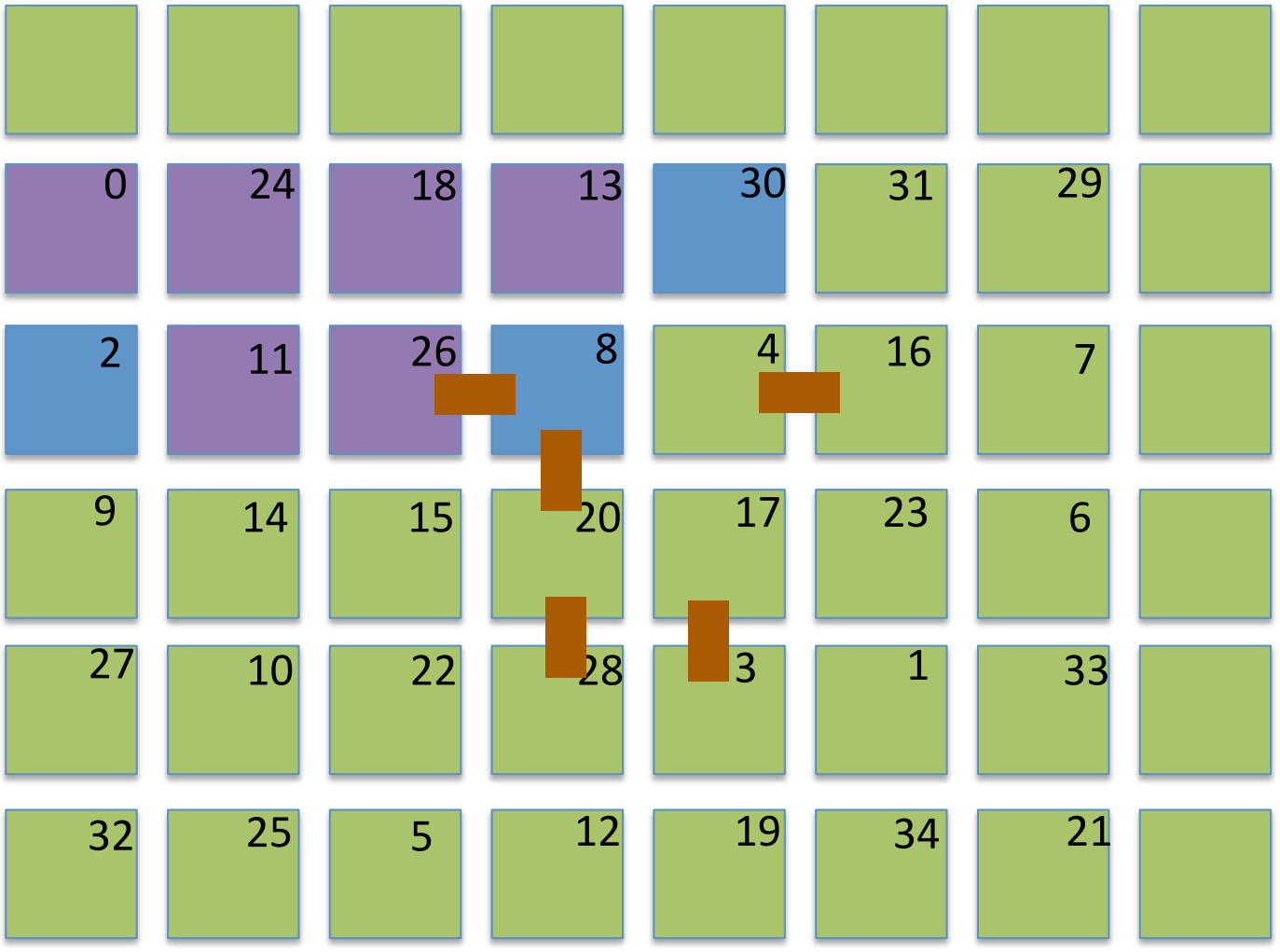
13 ← {0,11,8,30}


0 24 18 13 30 31 29
2 11 26 8 4 16 7
9 14 15 20 17 23 6
27 10 22 28 3 1 33
32 25 5 12 19 34 21

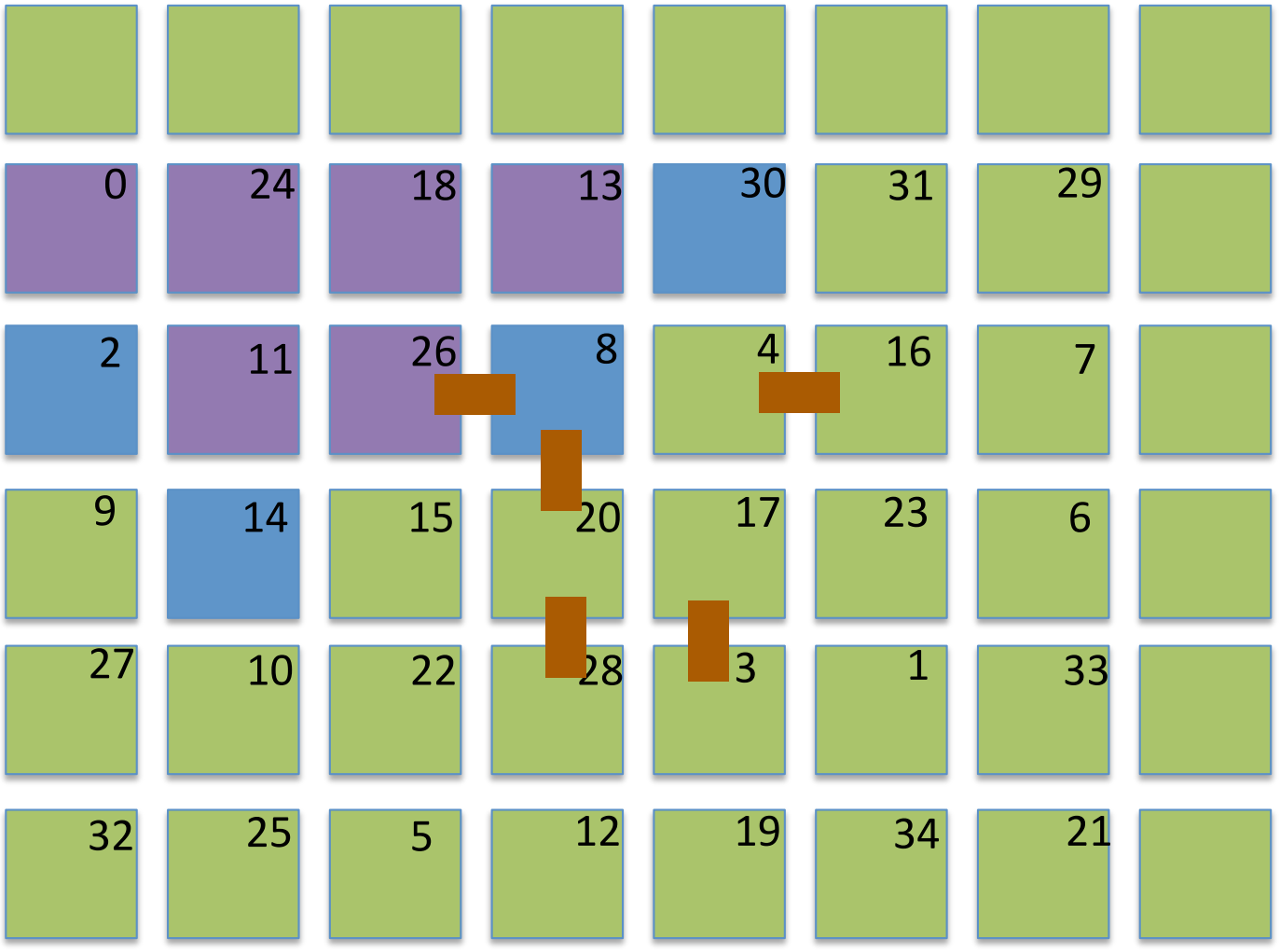
0 ← {11,8,30}


0 24 18 13 30 31 29
2 11 26 8 4 16 7
9 14 15 20 17 23 6
27 10 22 28 3 1 33
32 25 5 12 19 34 21

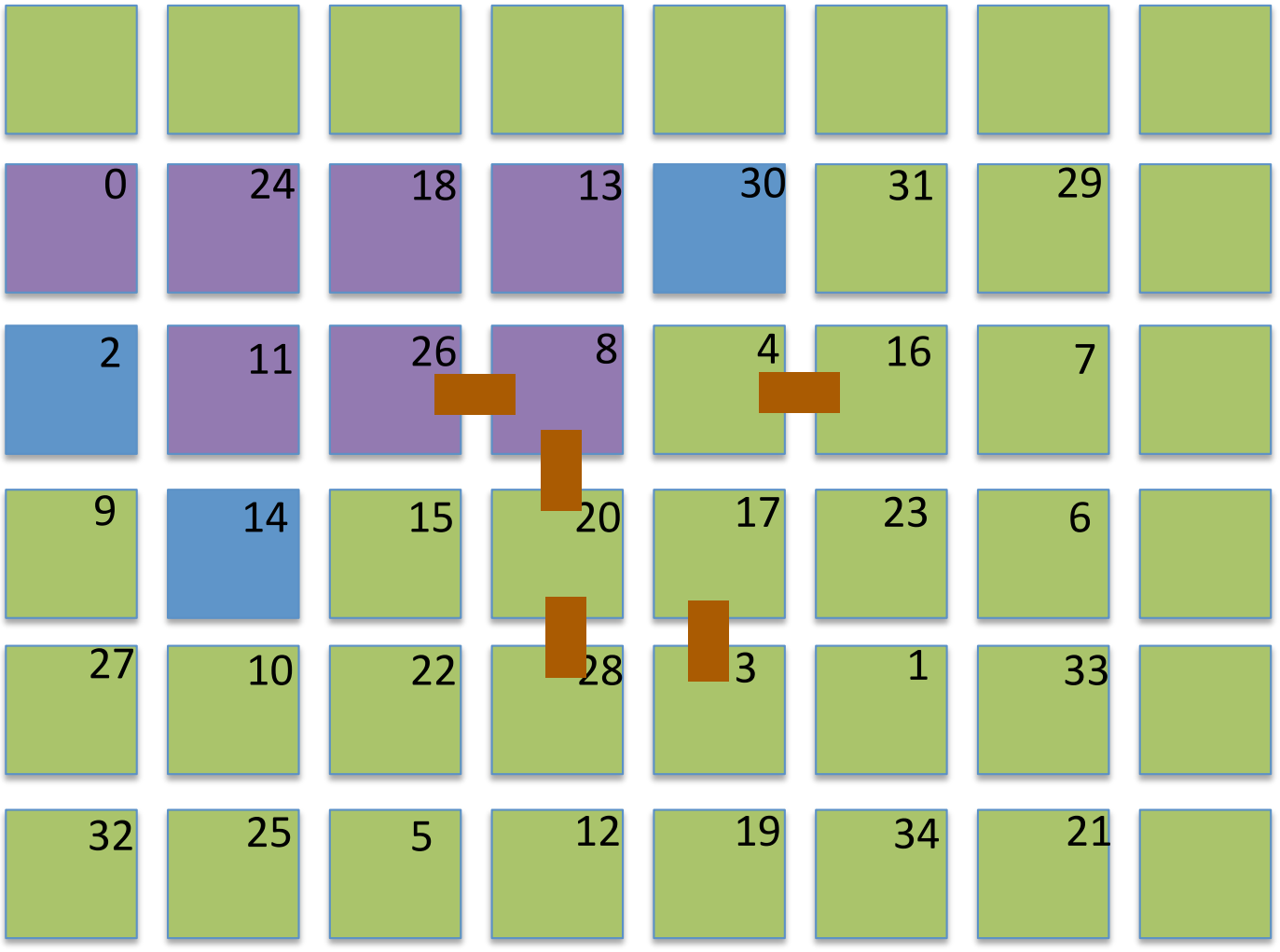
0 ← {11,8,30,2}


0 24 18 13 30 31 29
2 11 26 8 4 16 7
9 14 15 20 17 23 6
27 10 22 28 3 1 33
32 25 5 12 19 34 21

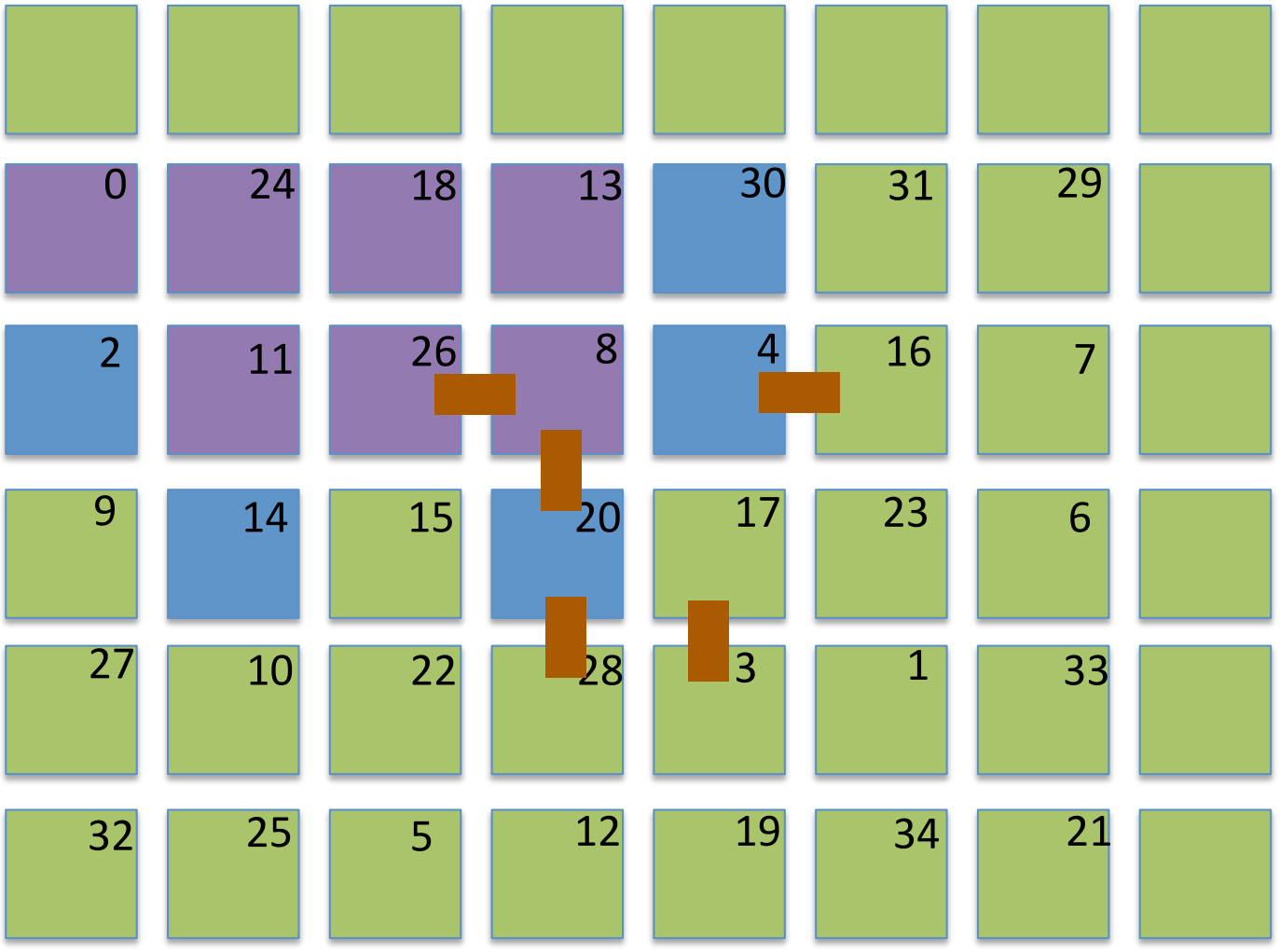
11 ← {8,30,2}


0 24 18 13 30 31 29
2 11 26 8 4 16 7
9 14 15 20 17 23 6
27 10 22 28 3 1 33
32 25 5 12 19 34 21

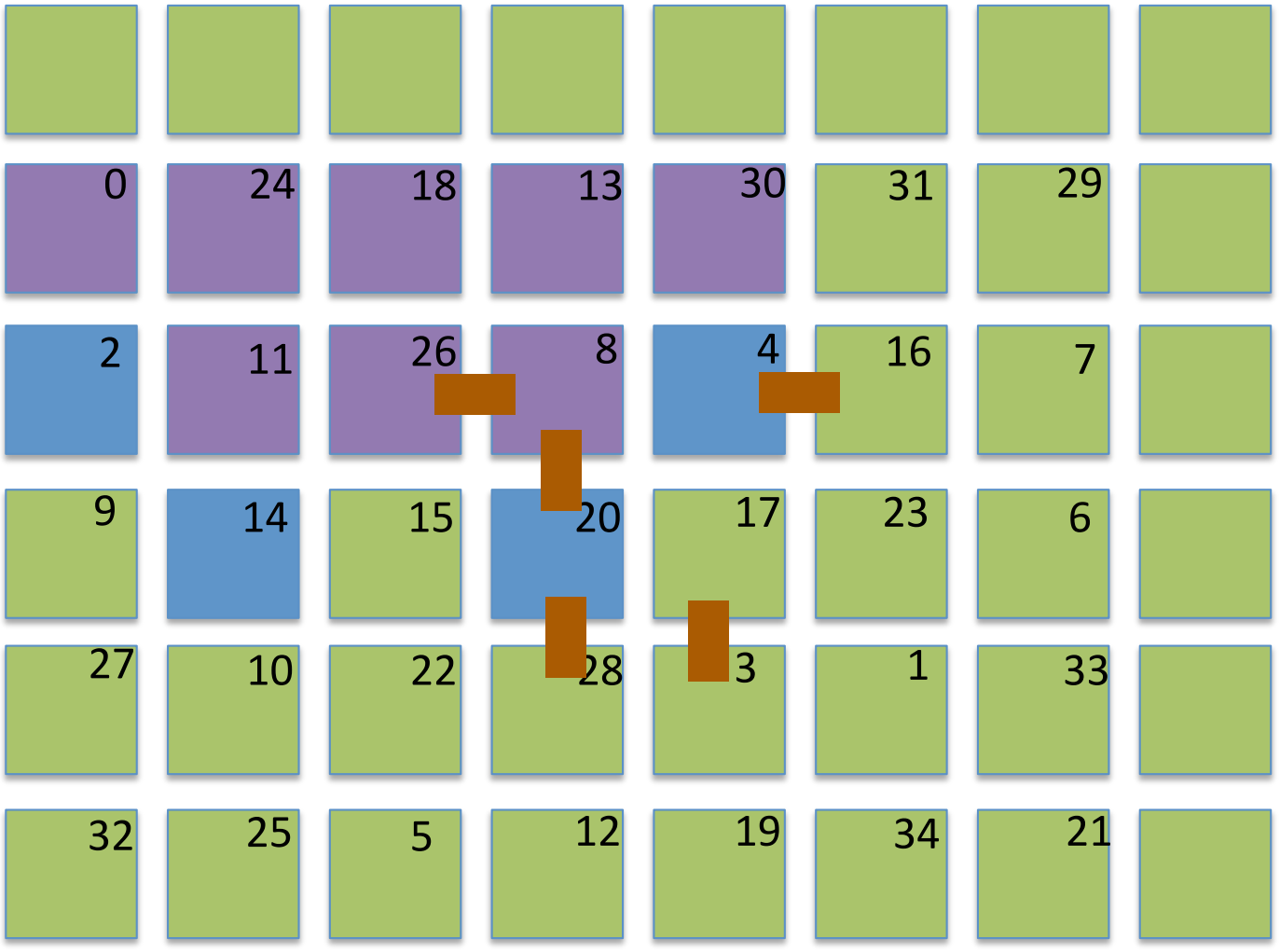
11 ← {8,30,2,14}


0 24 18 13 30 31 29
2 11 26 8 4 16 7
9 14 15 20 17 23 6
27 10 22 28 3 1 33
32 25 5 12 19 34 21

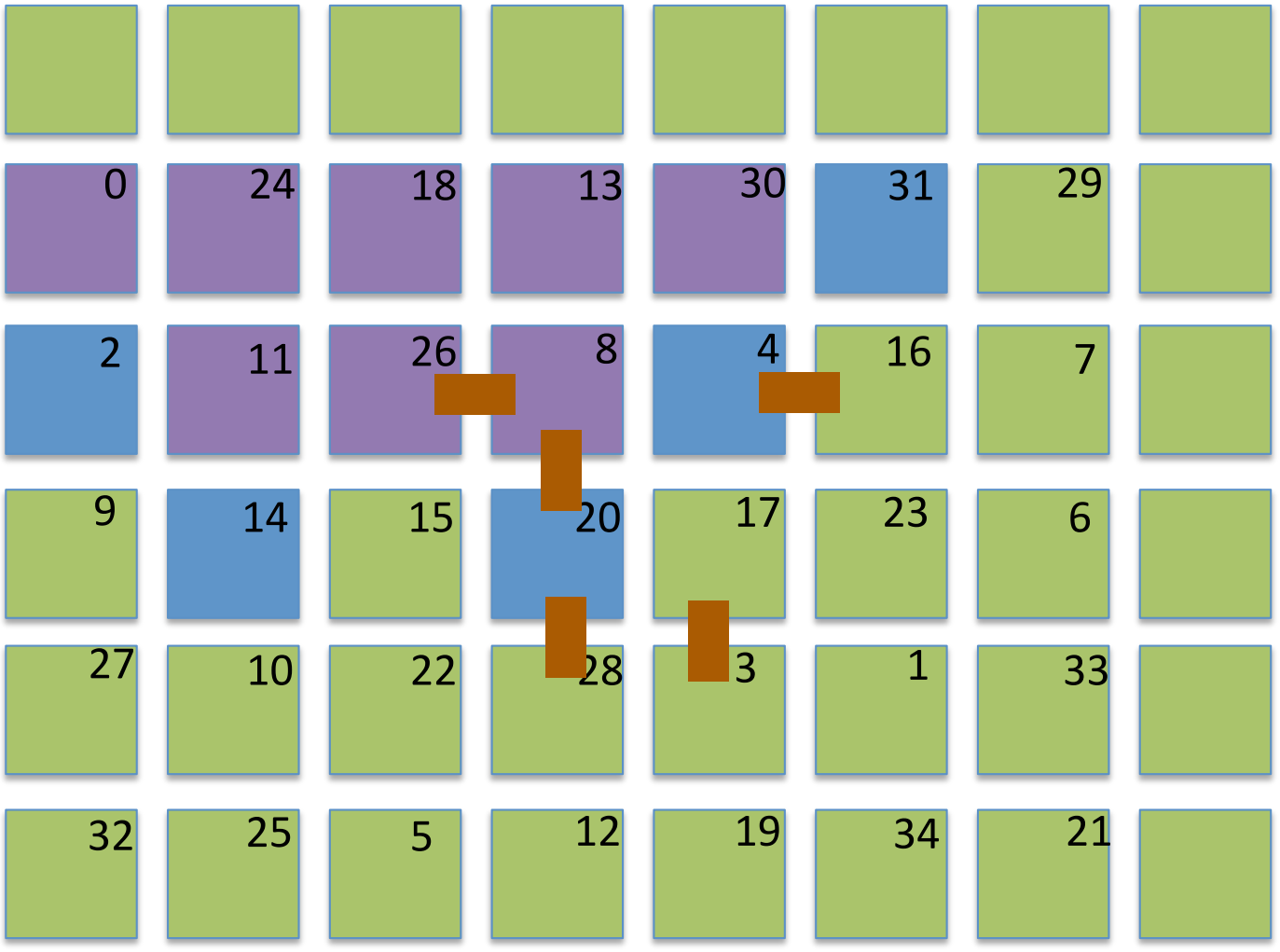
8 ← {30,2,14}


0 24 18 13 30 31 29
2 11 26 8 4 16 7
9 14 15 20 17 23 6
27 10 22 28 3 1 33
32 25 5 12 19 34 21

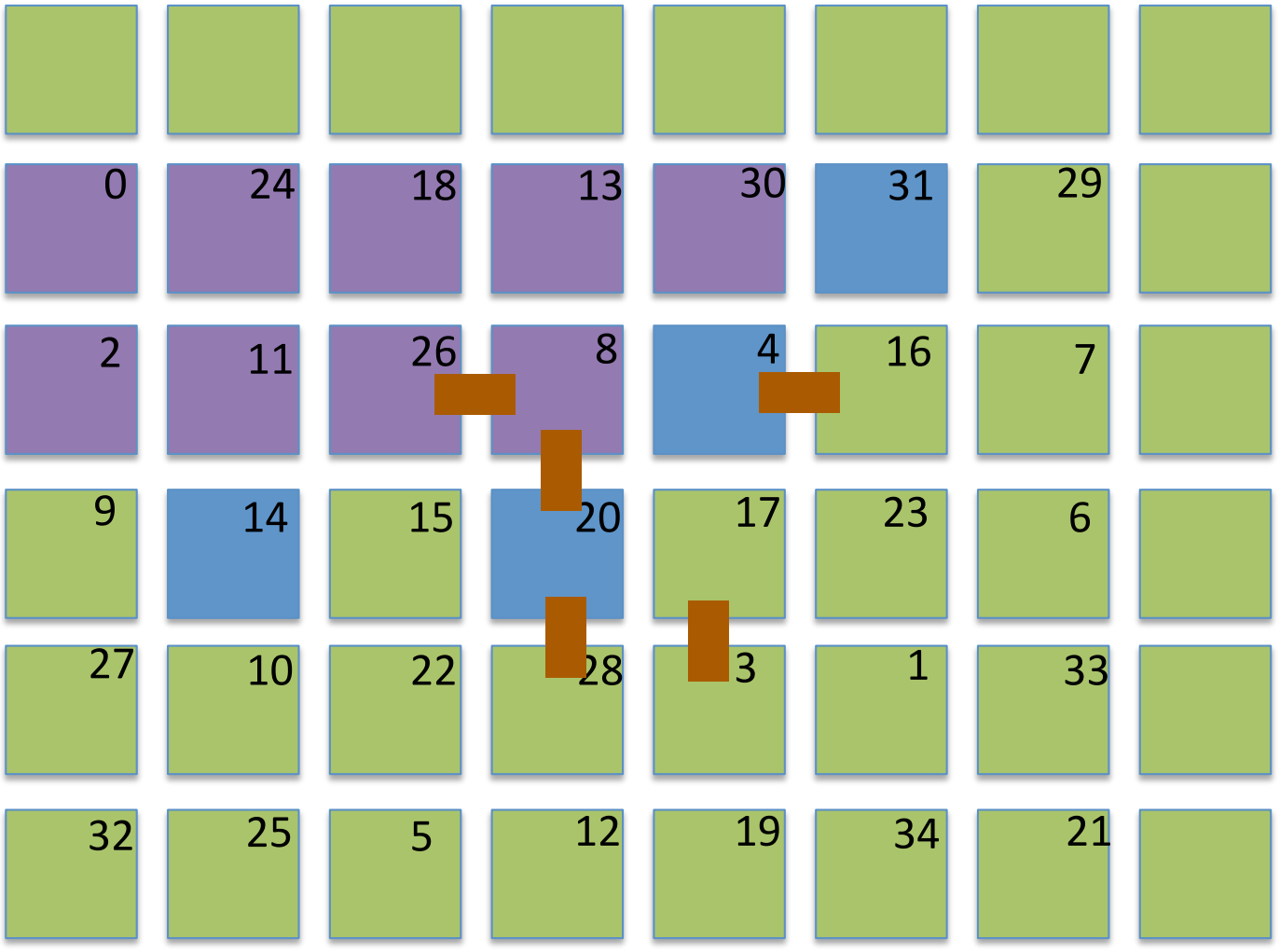
8 ← {30,2,14,20,4}


0 24 18 13 30 31 29
2 11 26 8 4 16 7
9 14 15 20 17 23 6
27 10 22 28 3 1 33
32 25 5 12 19 34 21

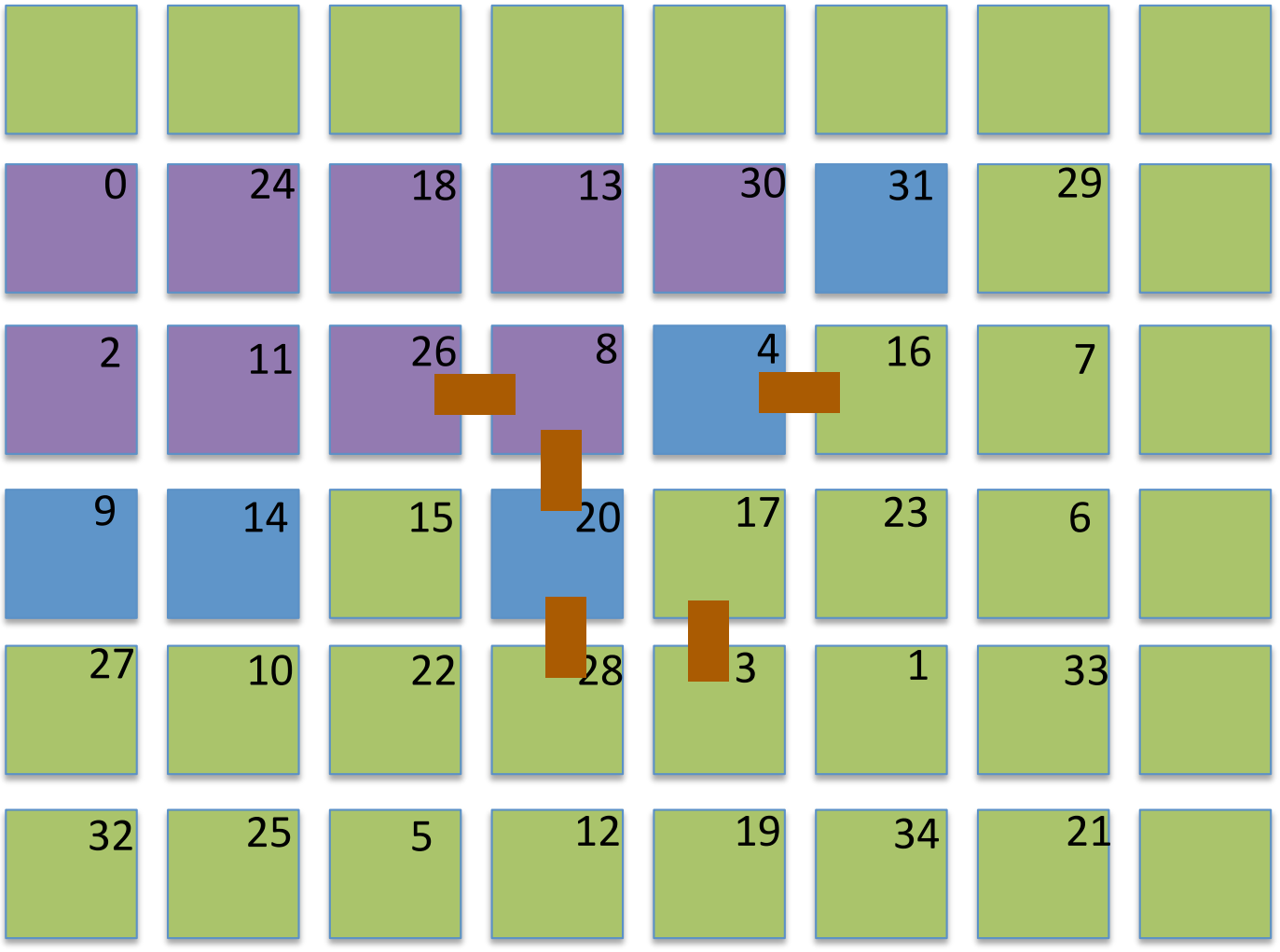
30 ← {2,14,20,4}


0 24 18 13 30 31 29
2 11 26 8 4 16 7
9 14 15 20 17 23 6
27 10 22 28 3 1 33
32 25 5 12 19 34 21

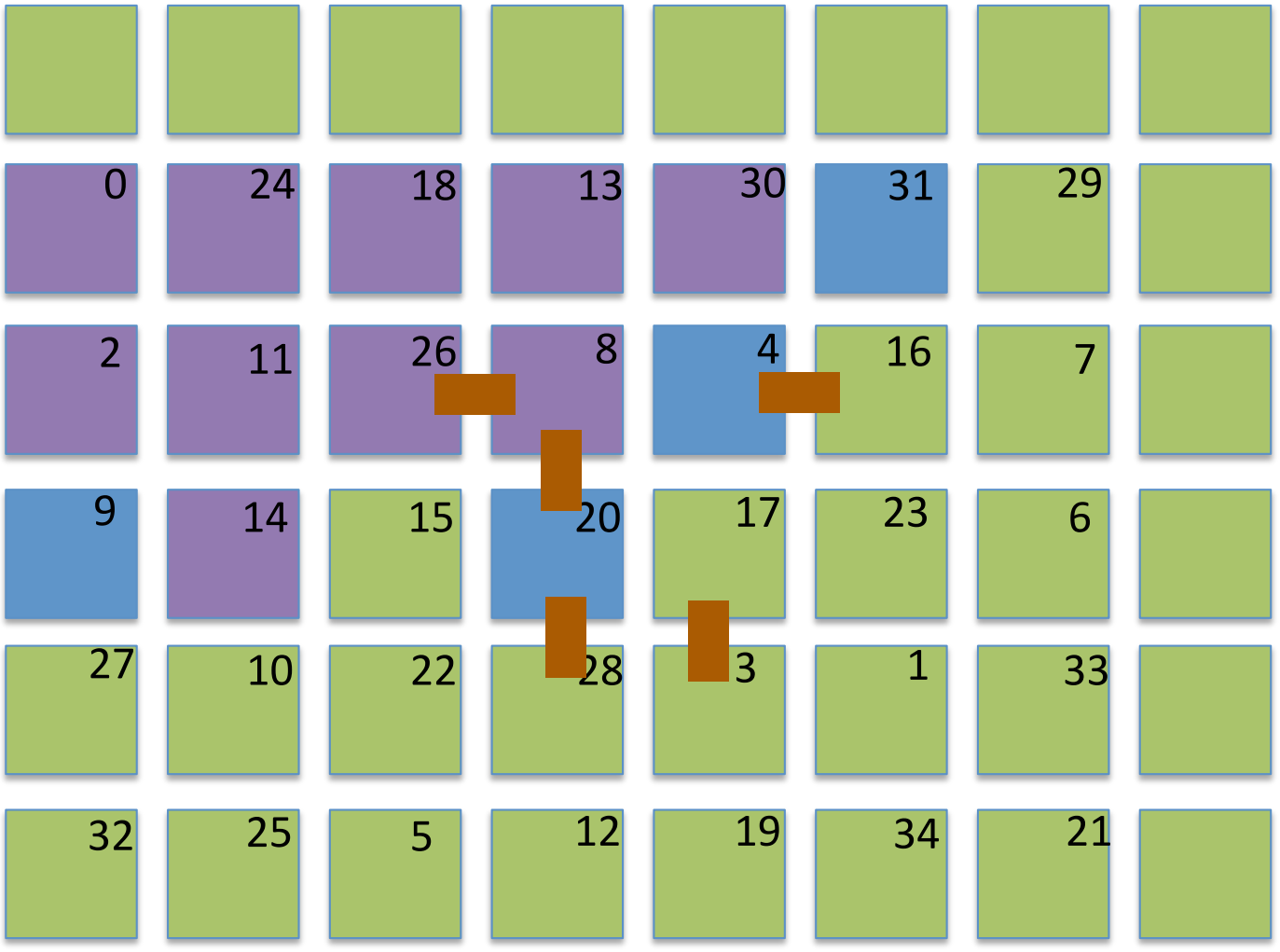
30 ← {2,14,20,4,31}


0 24 18 13 30 31 29
2 11 26 8 4 16 7
9 14 15 20 17 23 6
27 10 22 28 3 1 33
32 25 5 12 19 34 21

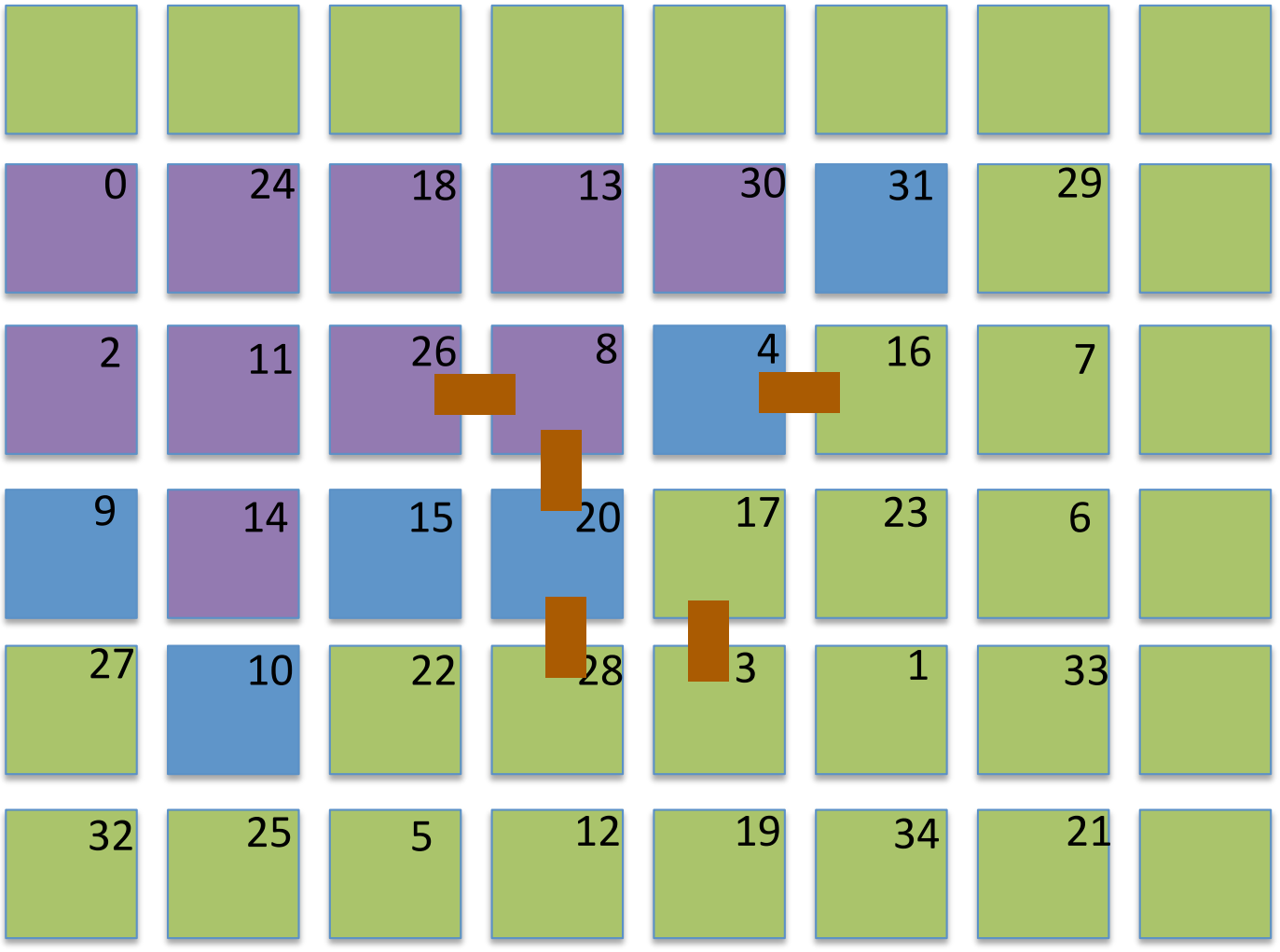
2 ← {14,20,4,31}


0 24 18 13 30 31 29
2 11 26 8 4 16 7
9 14 15 20 17 23 6
27 10 22 28 3 1 33
32 25 5 12 19 34 21

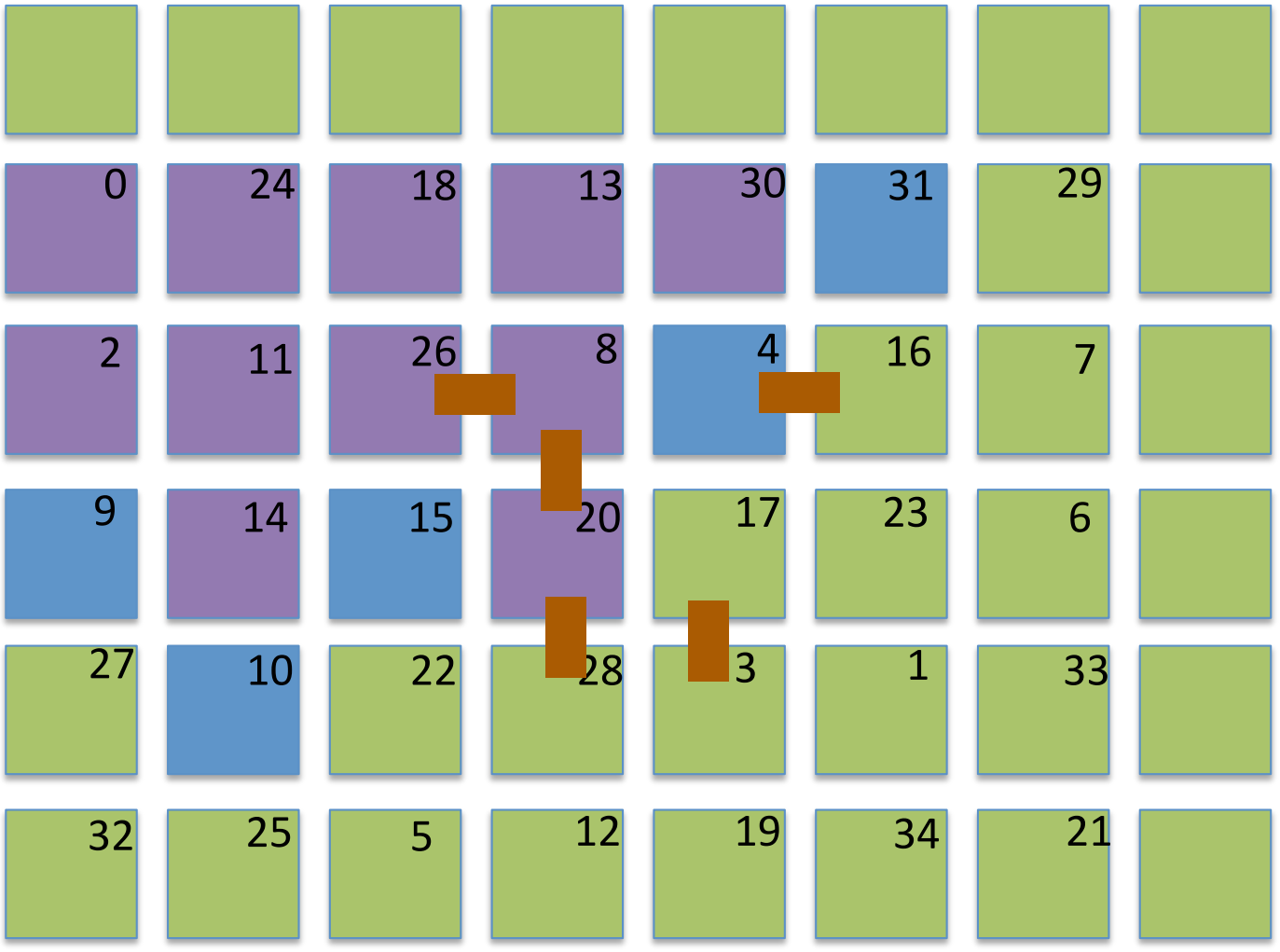
2 ← {14,20,4,31,9}


0 24 18 13 30 31 29
2 11 26 8 4 16 7
9 14 15 20 17 23 6
27 10 22 28 3 1 33
32 25 5 12 19 34 21

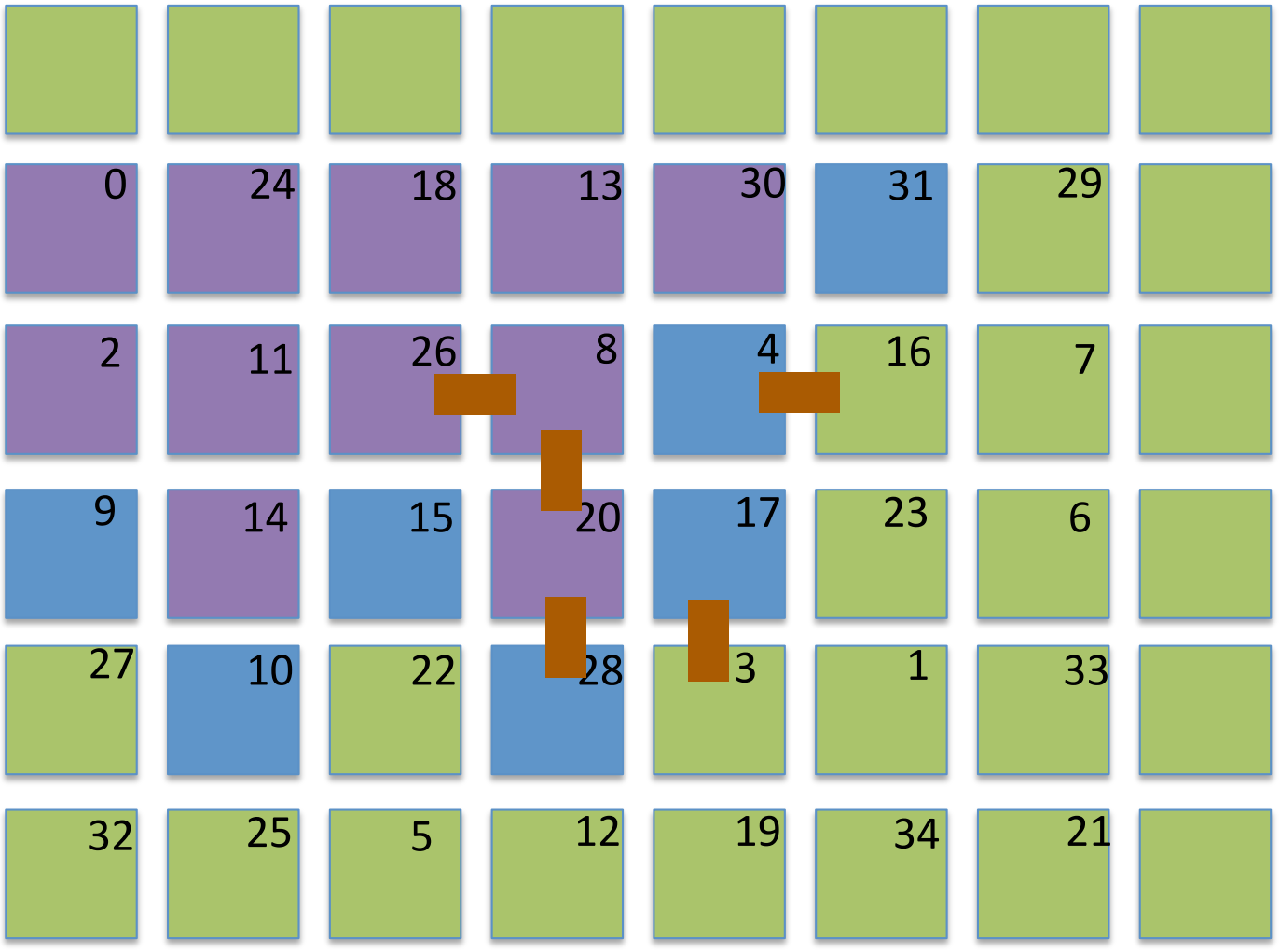
14 ← {20,4,31,9}



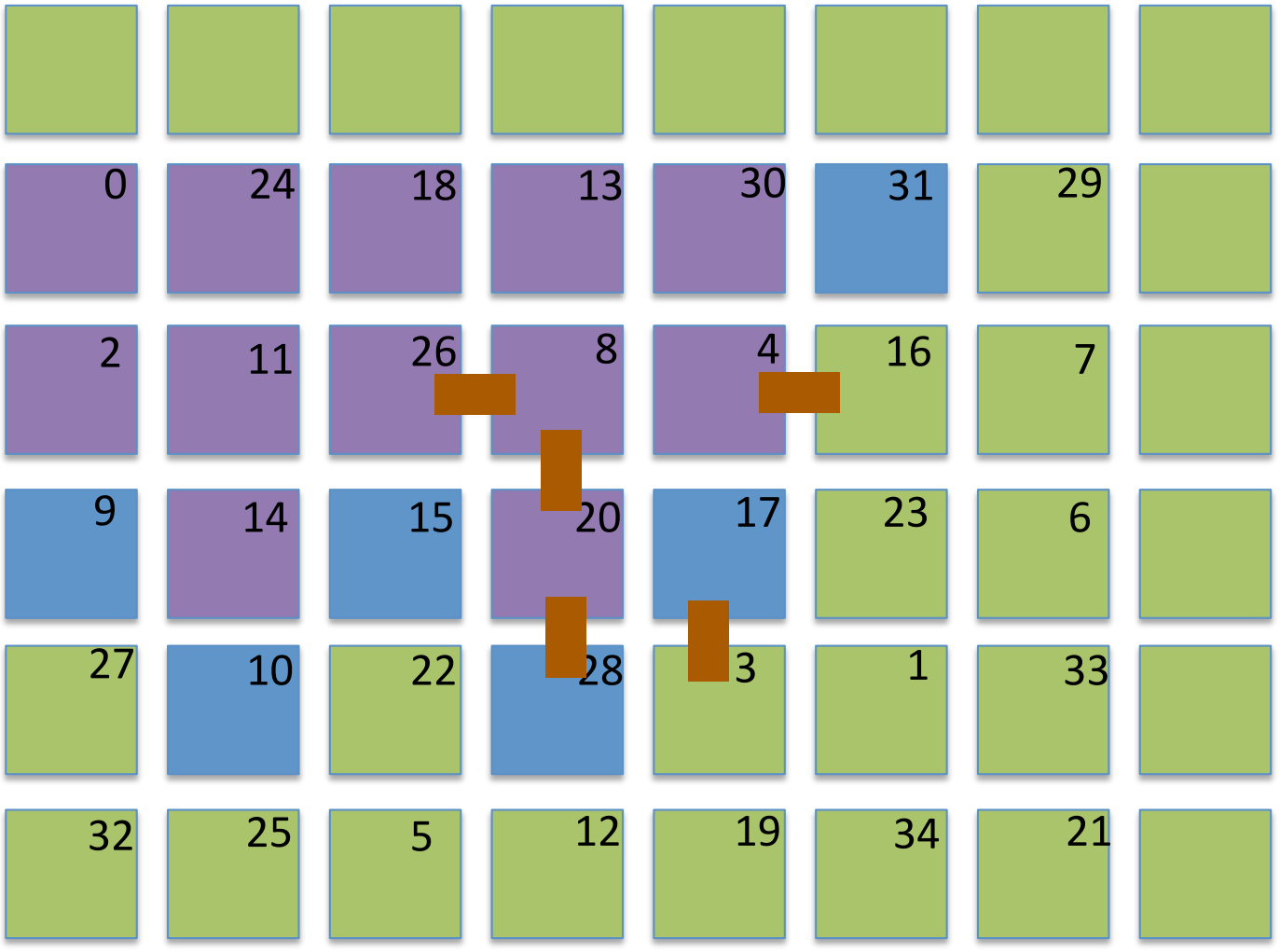
14 ← {20,4,31,9,10,15}


0 24 18 13 30 31 29
2 11 26 8 4 16 7
9 14 15 20 17 23 6
27 10 22 28 3 1 33
32 25 5 12 19 34 21

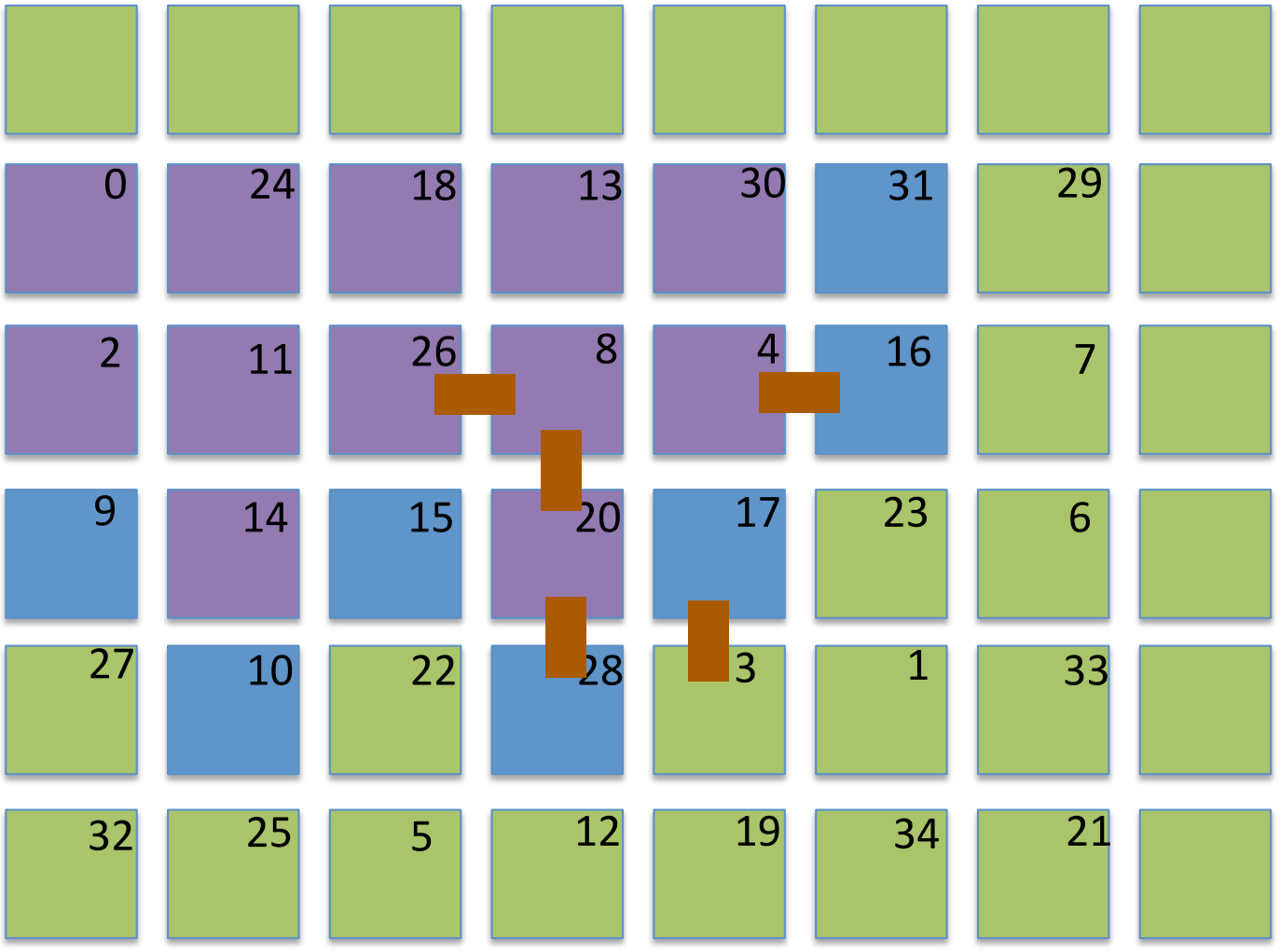
20 ← {4,31,9,10,15}



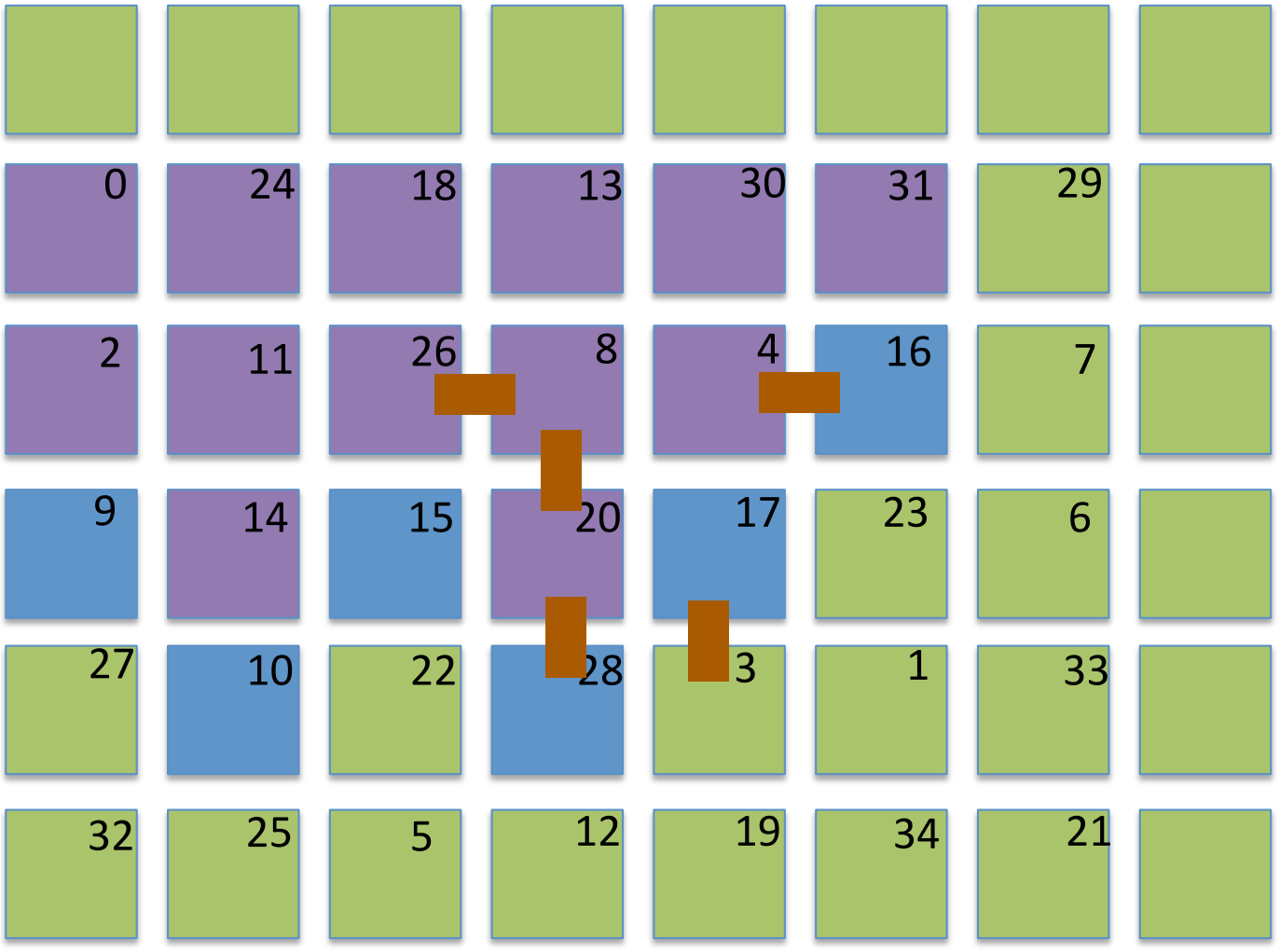
20 ← {4,31,9,10,15,17,28}



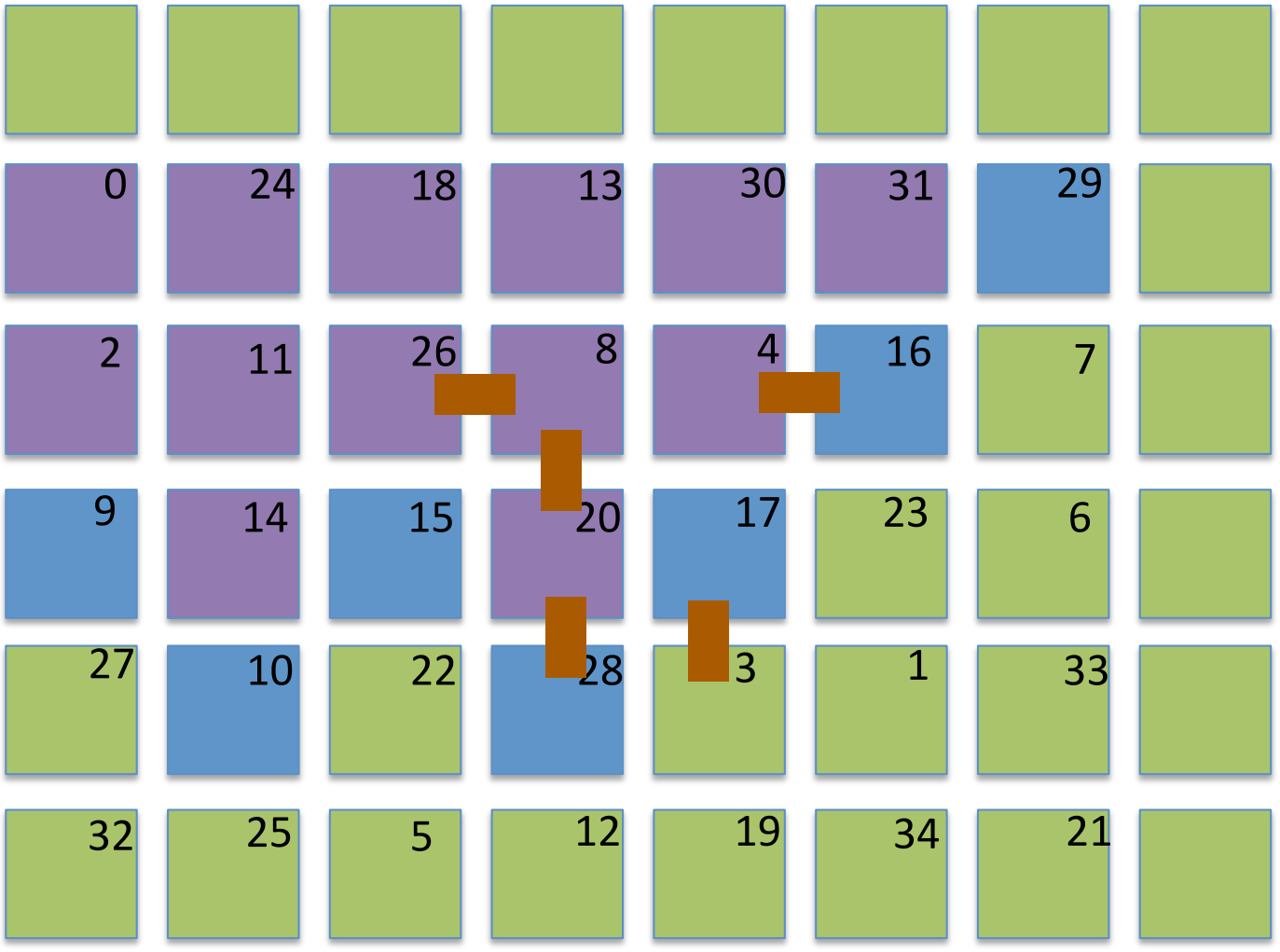
4 ← {31,9,10,15,17,28}



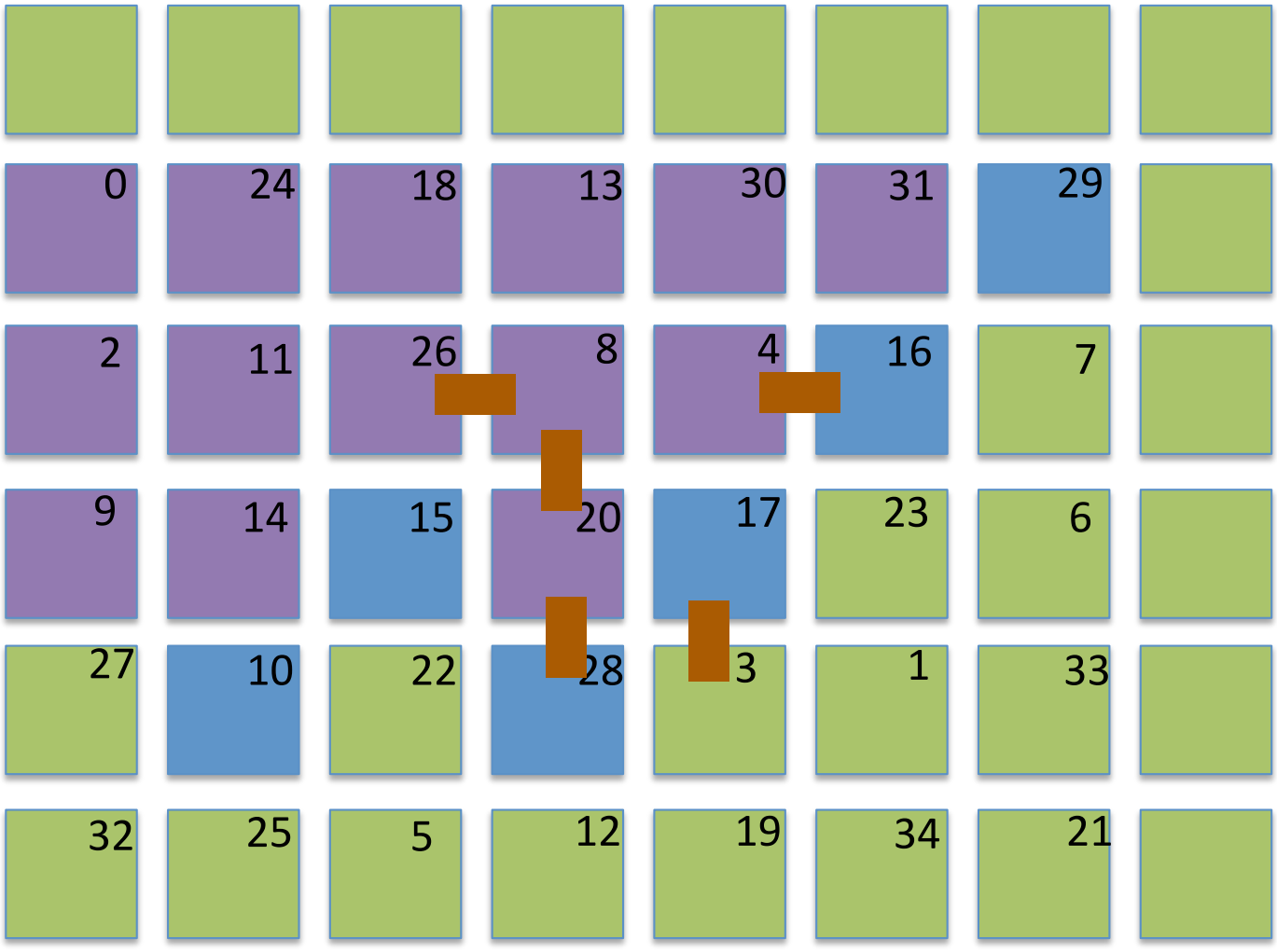
4 ← {31,9,10,15,17,28,16}



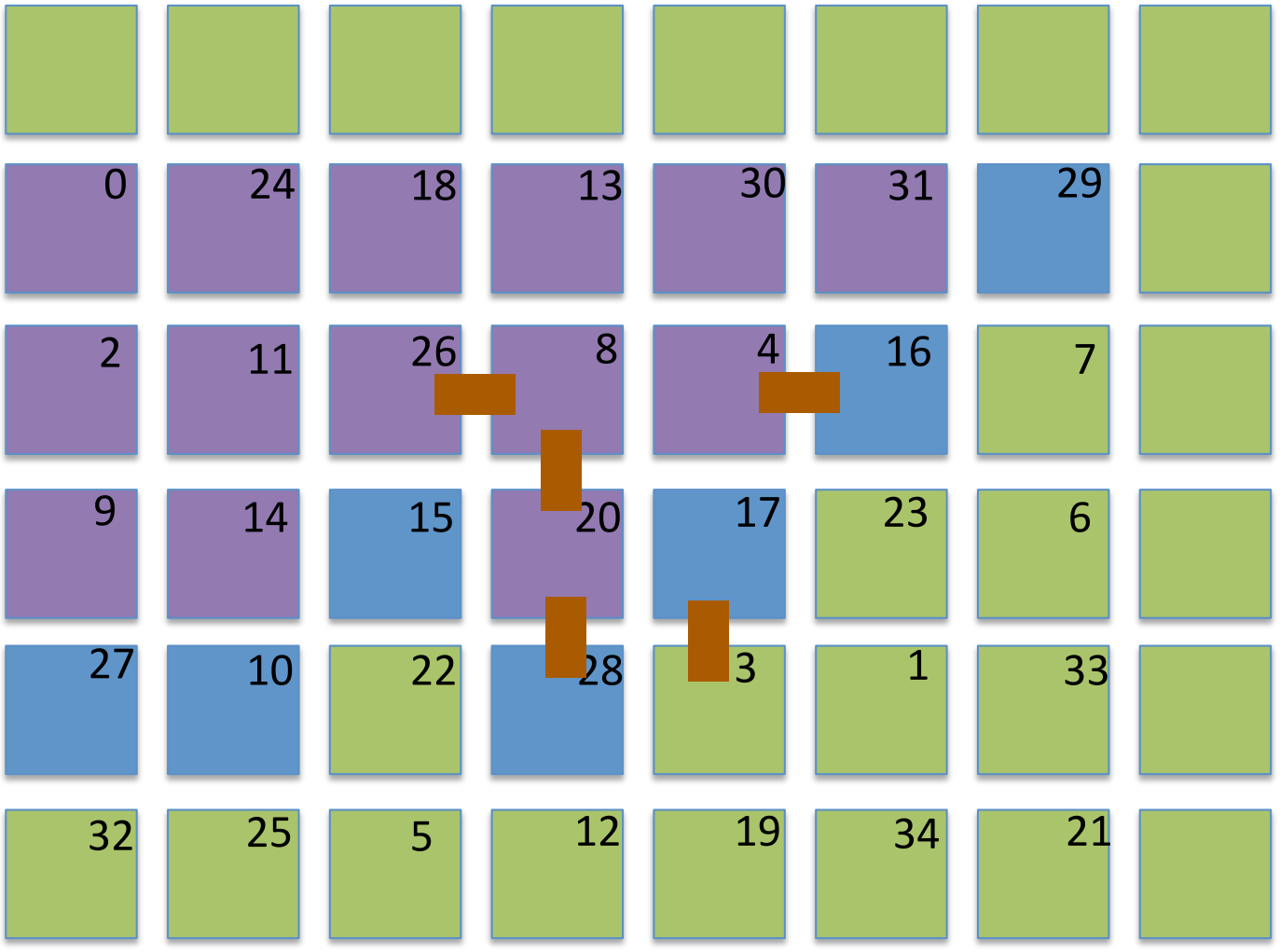
31 ← {9,10,15,17,28,16}



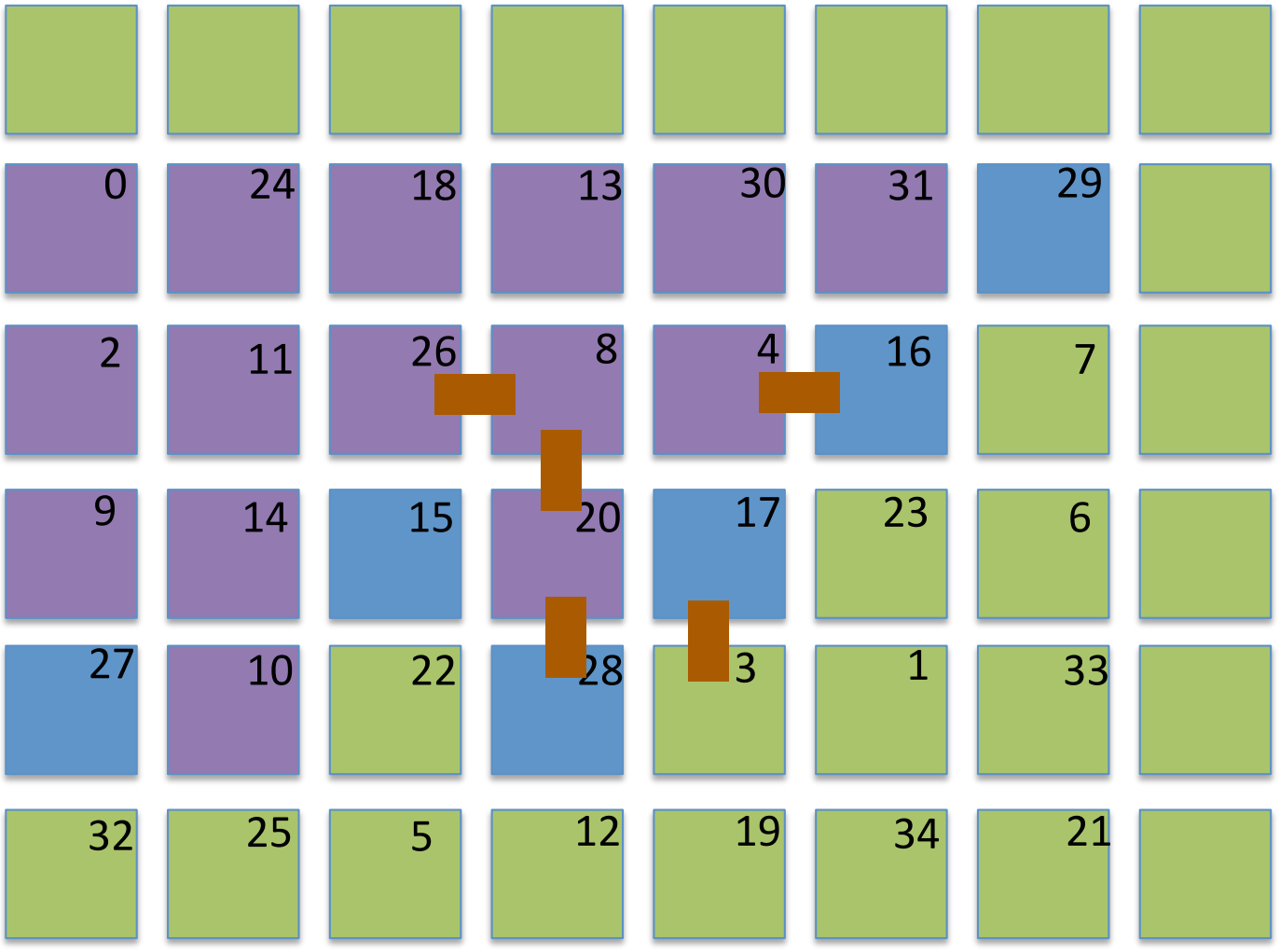
31 ← {9,10,15,17,28,16,29}



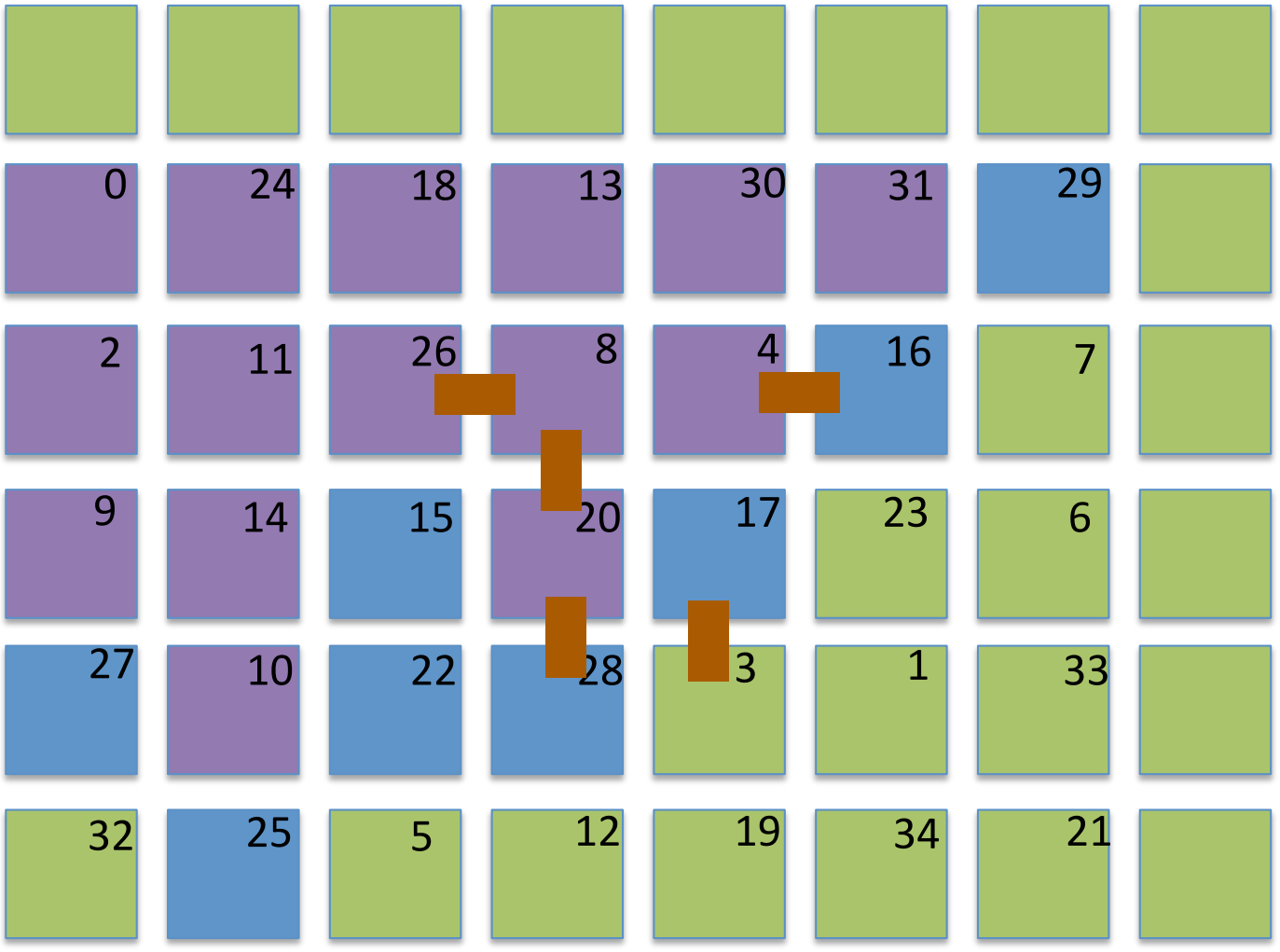
9 ← {10,15,17,28,16,29}



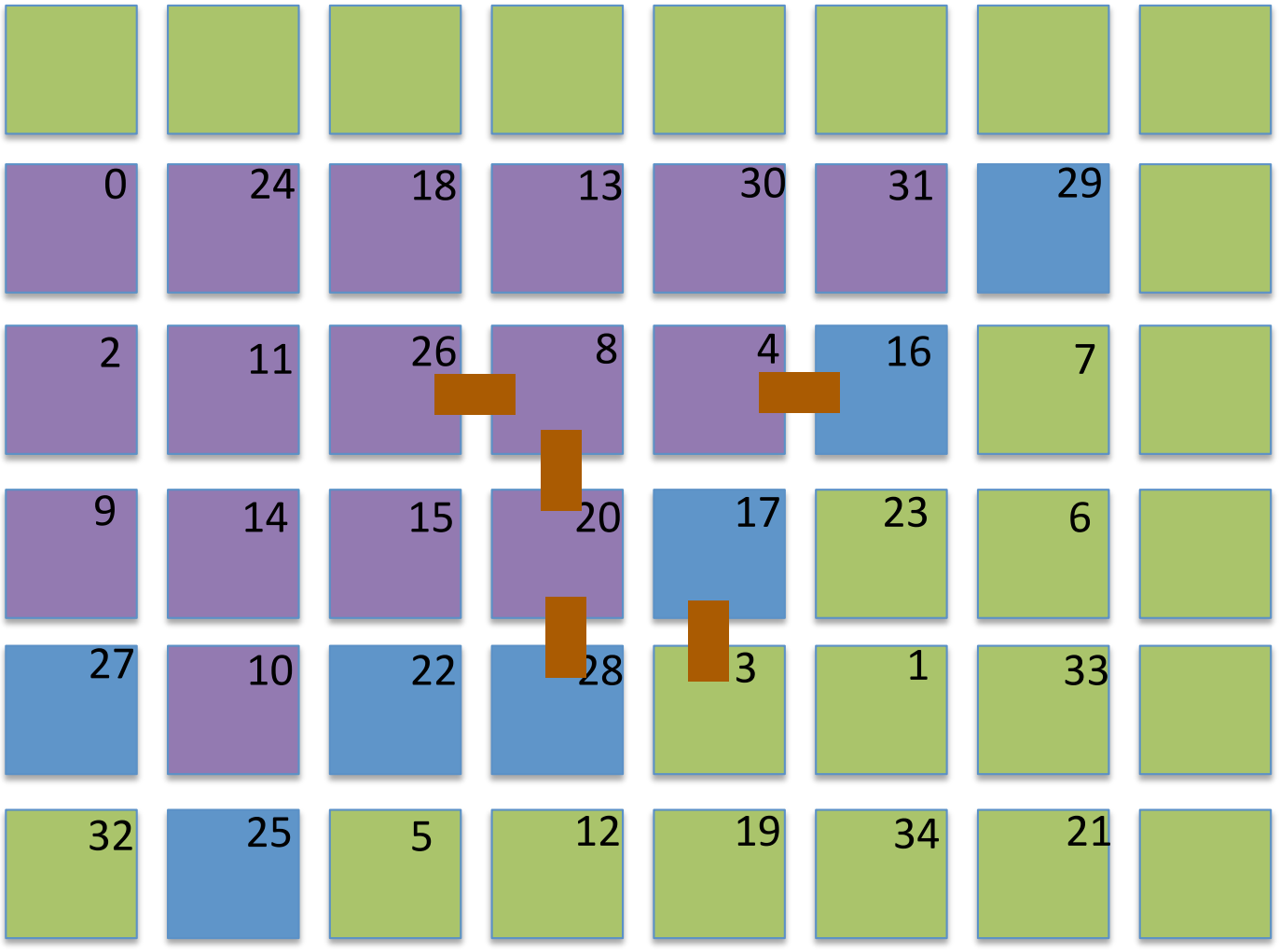
9 ← {10,15,17,28,16,29,27}



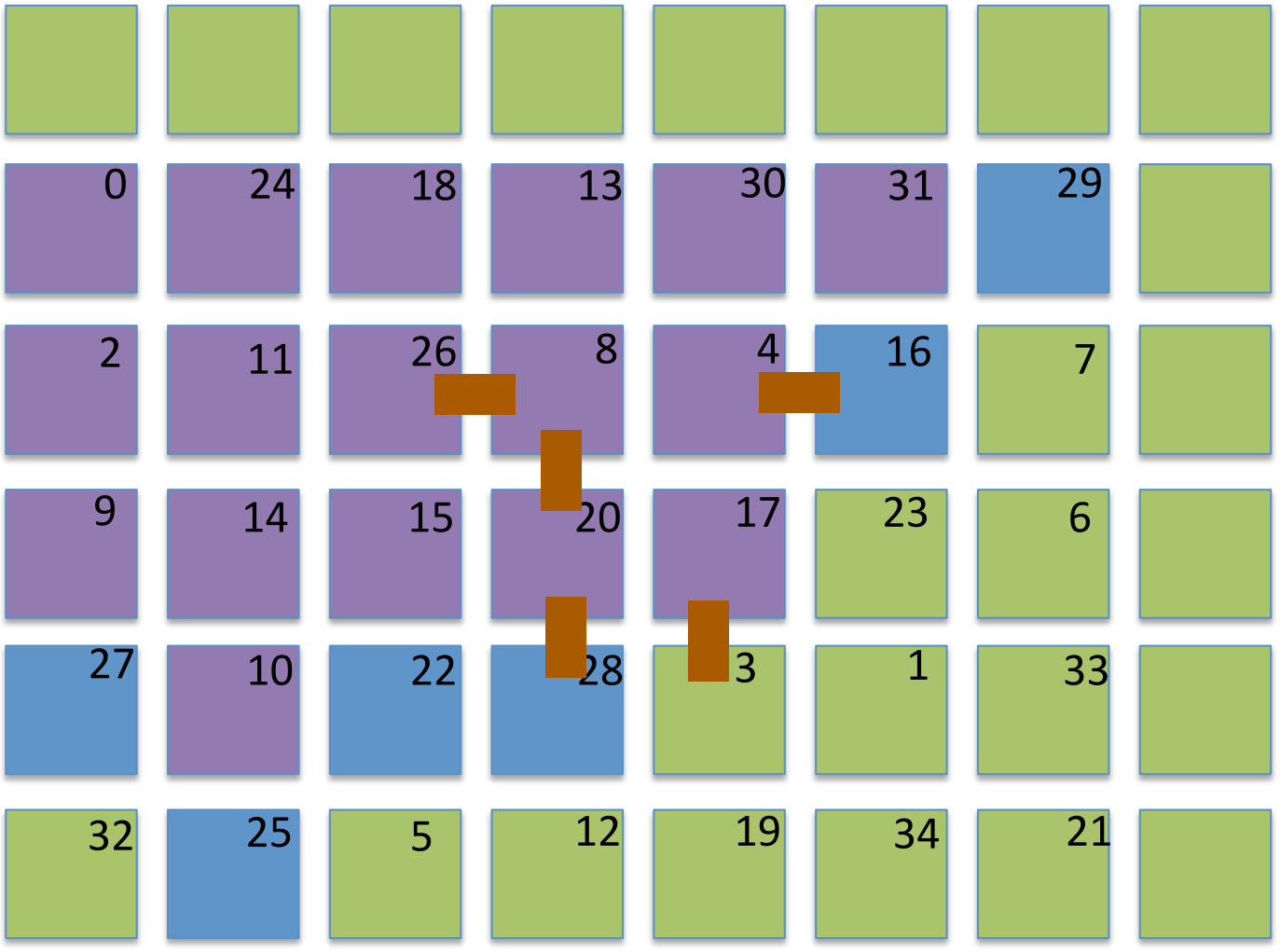
10 ← {15,17,28,16,29,27}



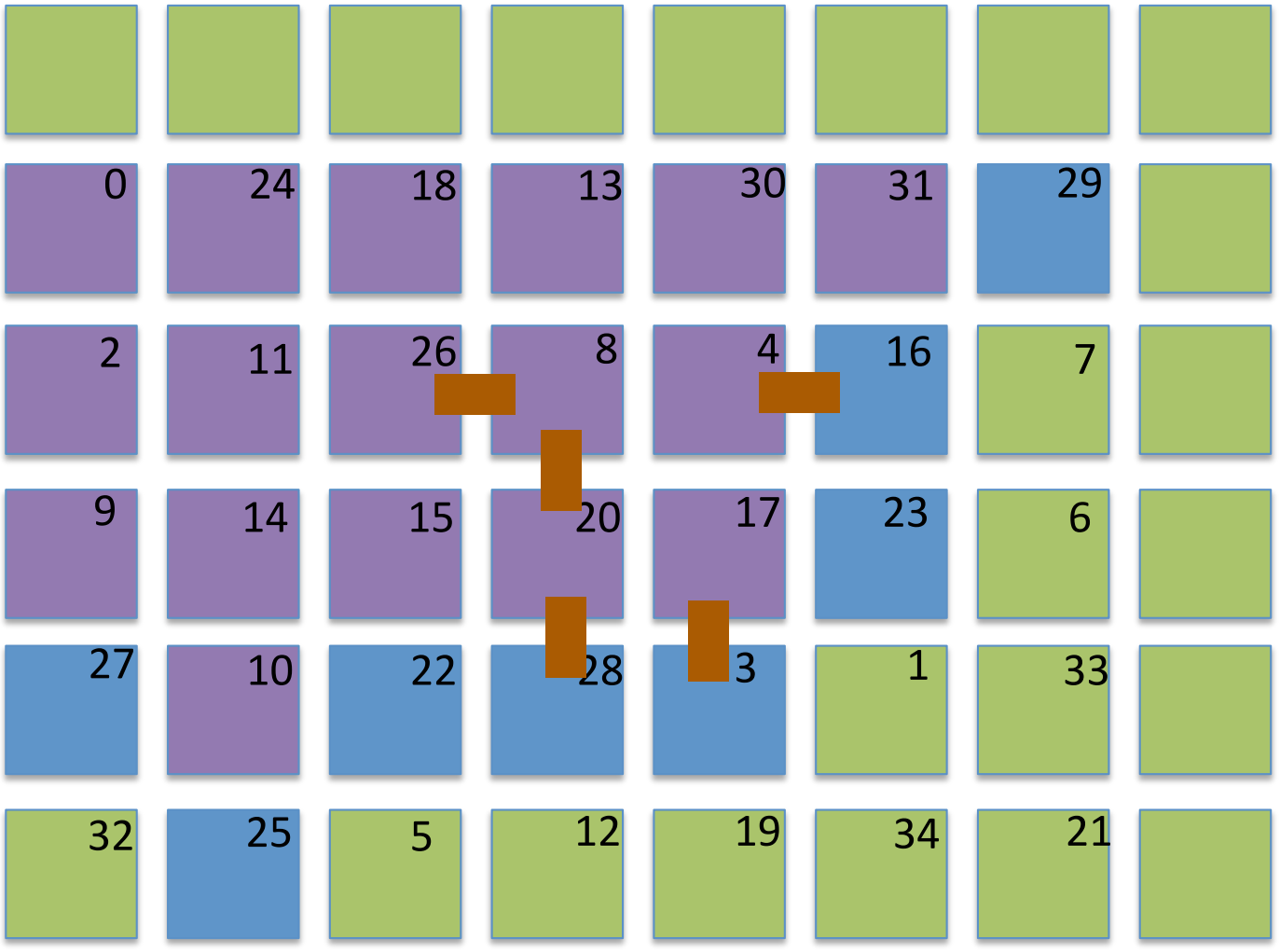
10 ← {15,17,28,16,29,27,22,25}



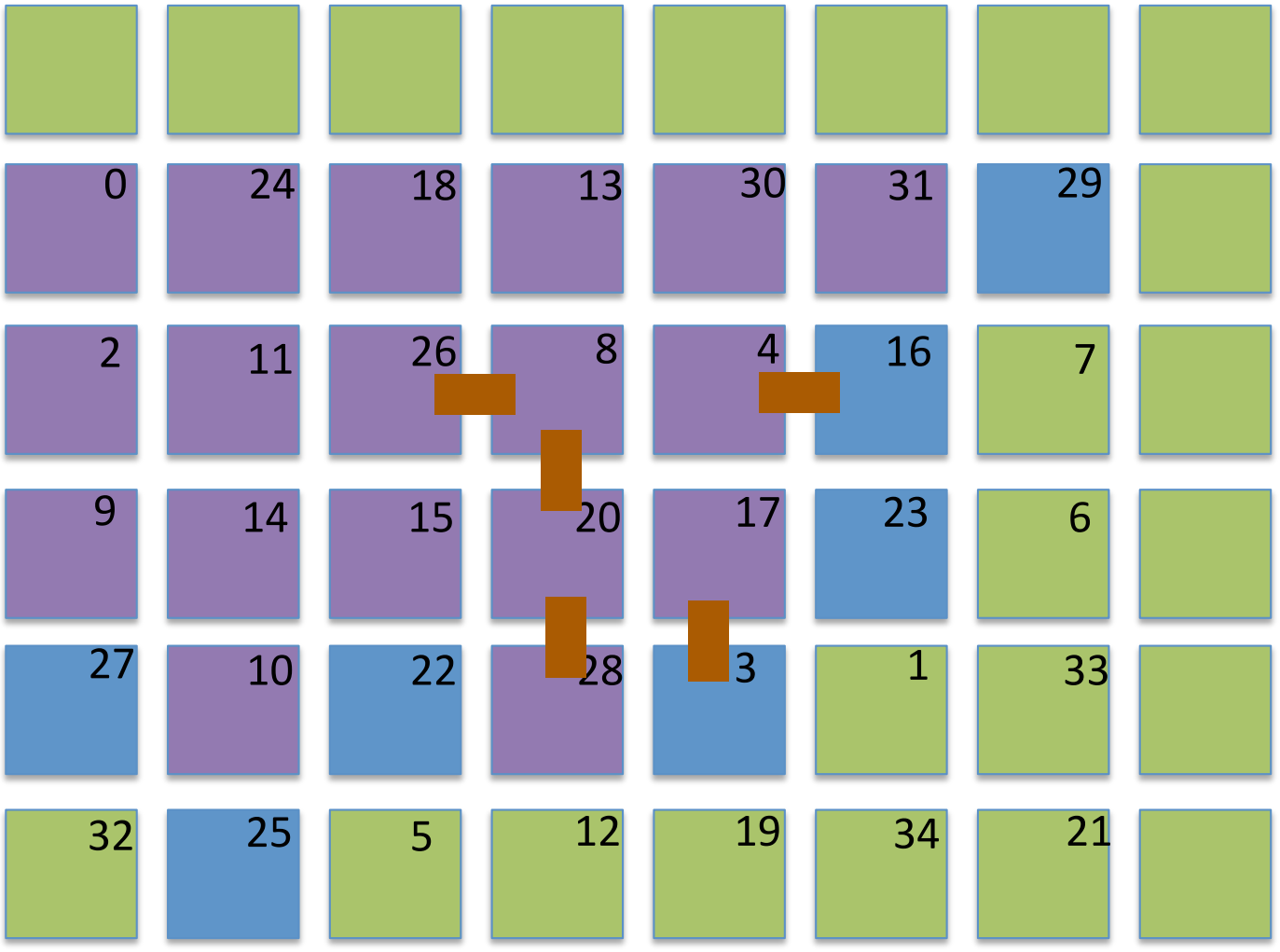
15 ← {17,28,16,29,27,22,25}



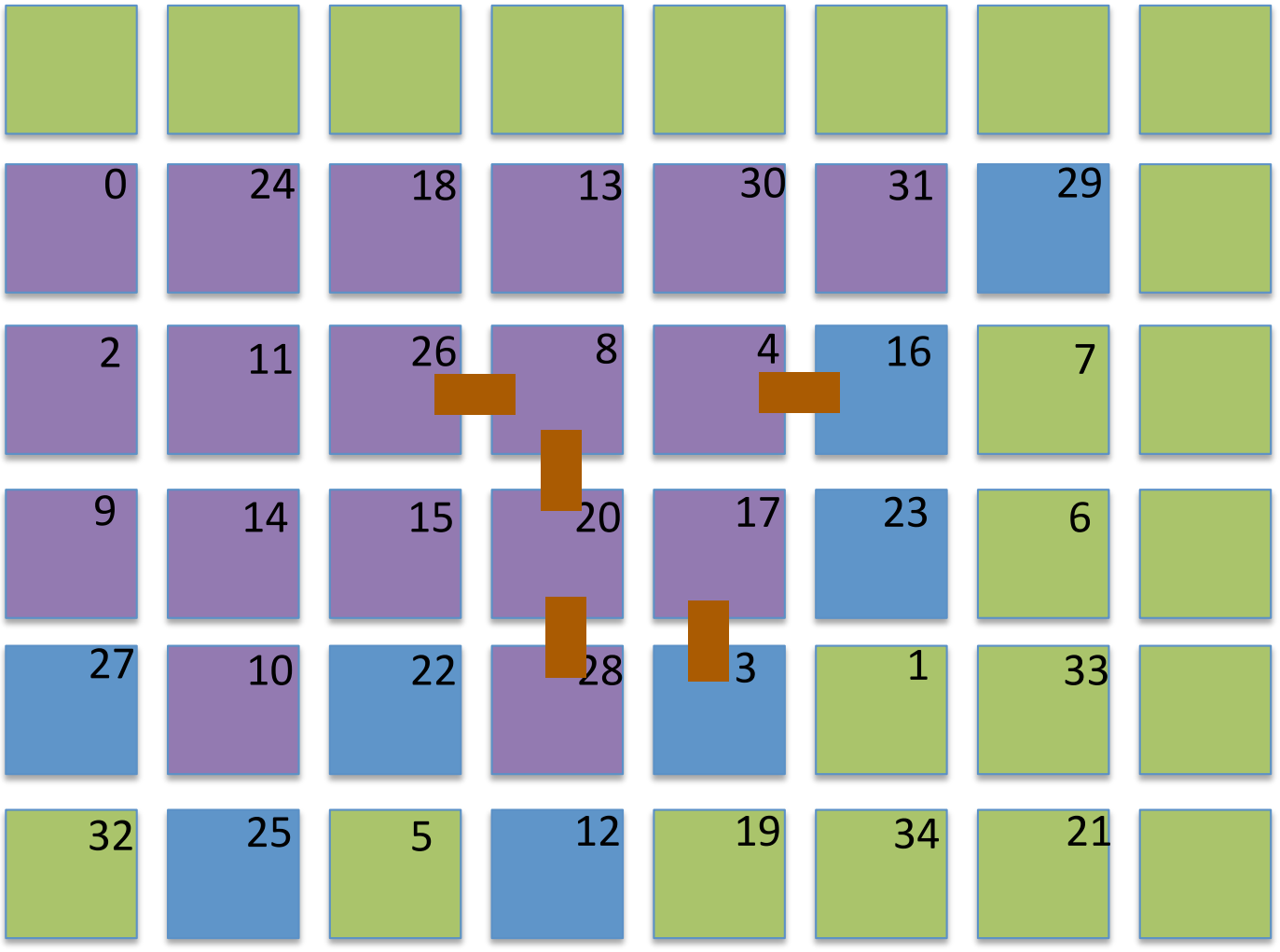
17 ← {28,16,29,27,22,25}



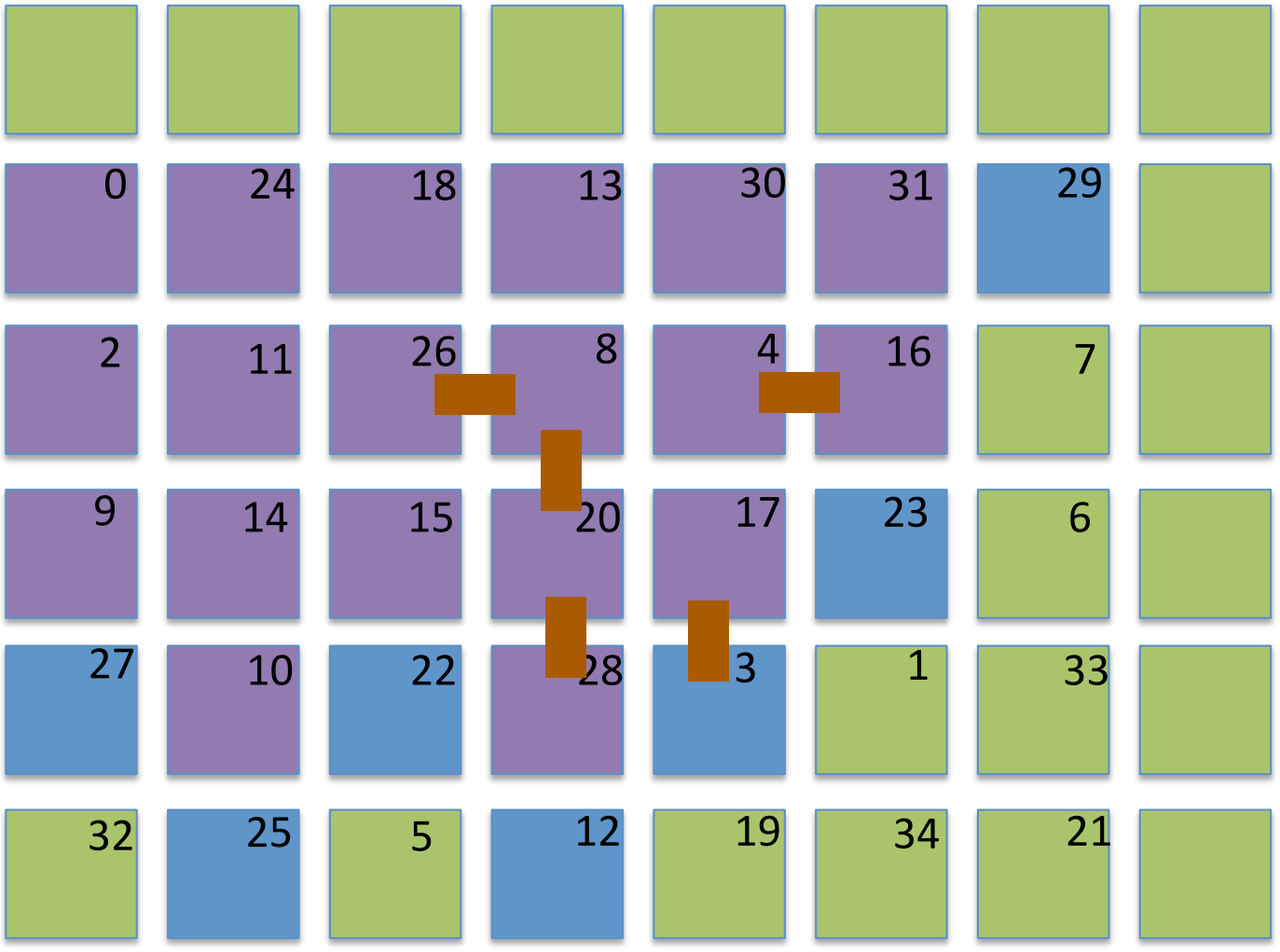
17 ← {28,16,29,27,22,25,3,23}



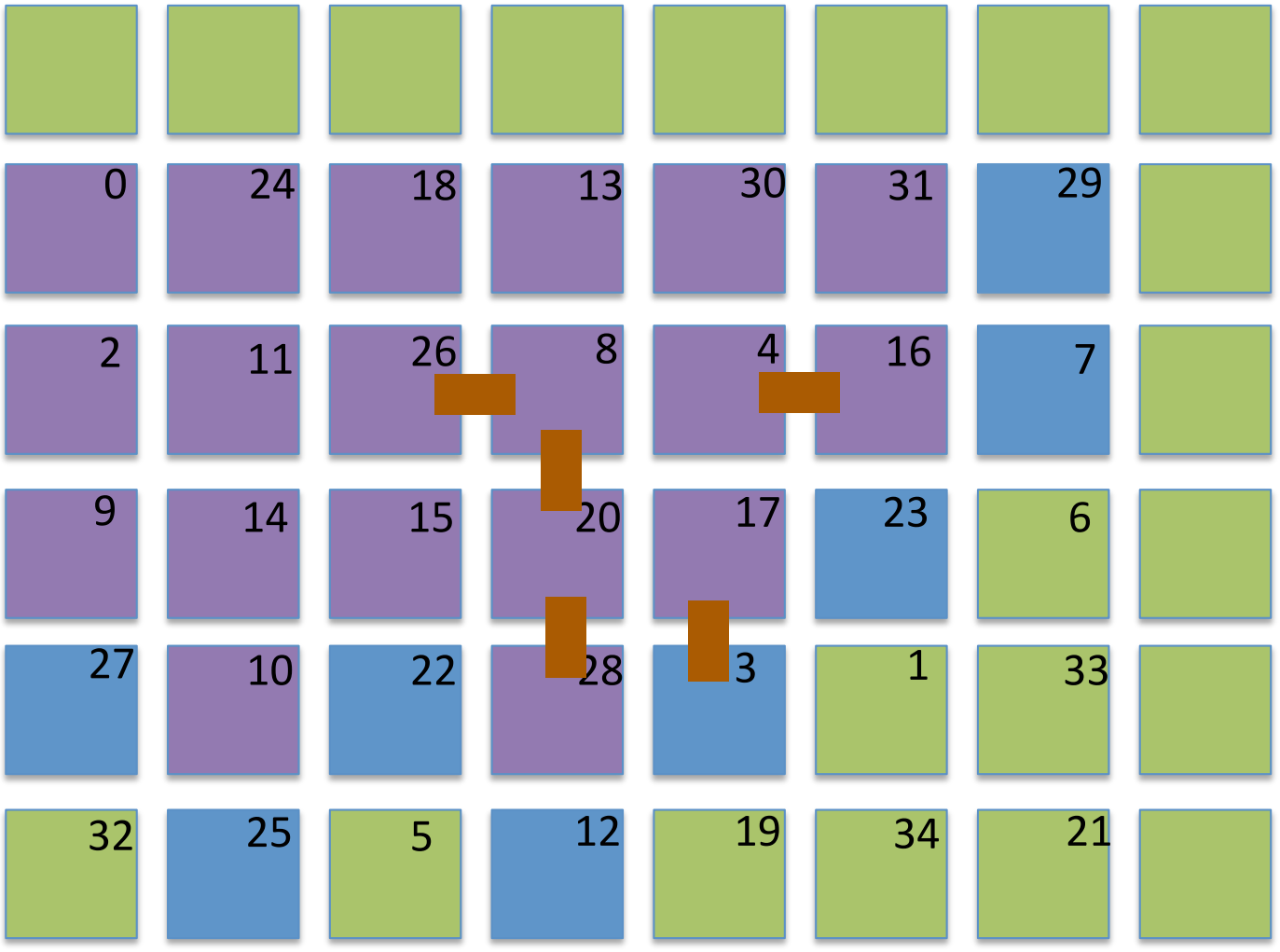
28 ← {16,29,27,22,25,3,23}



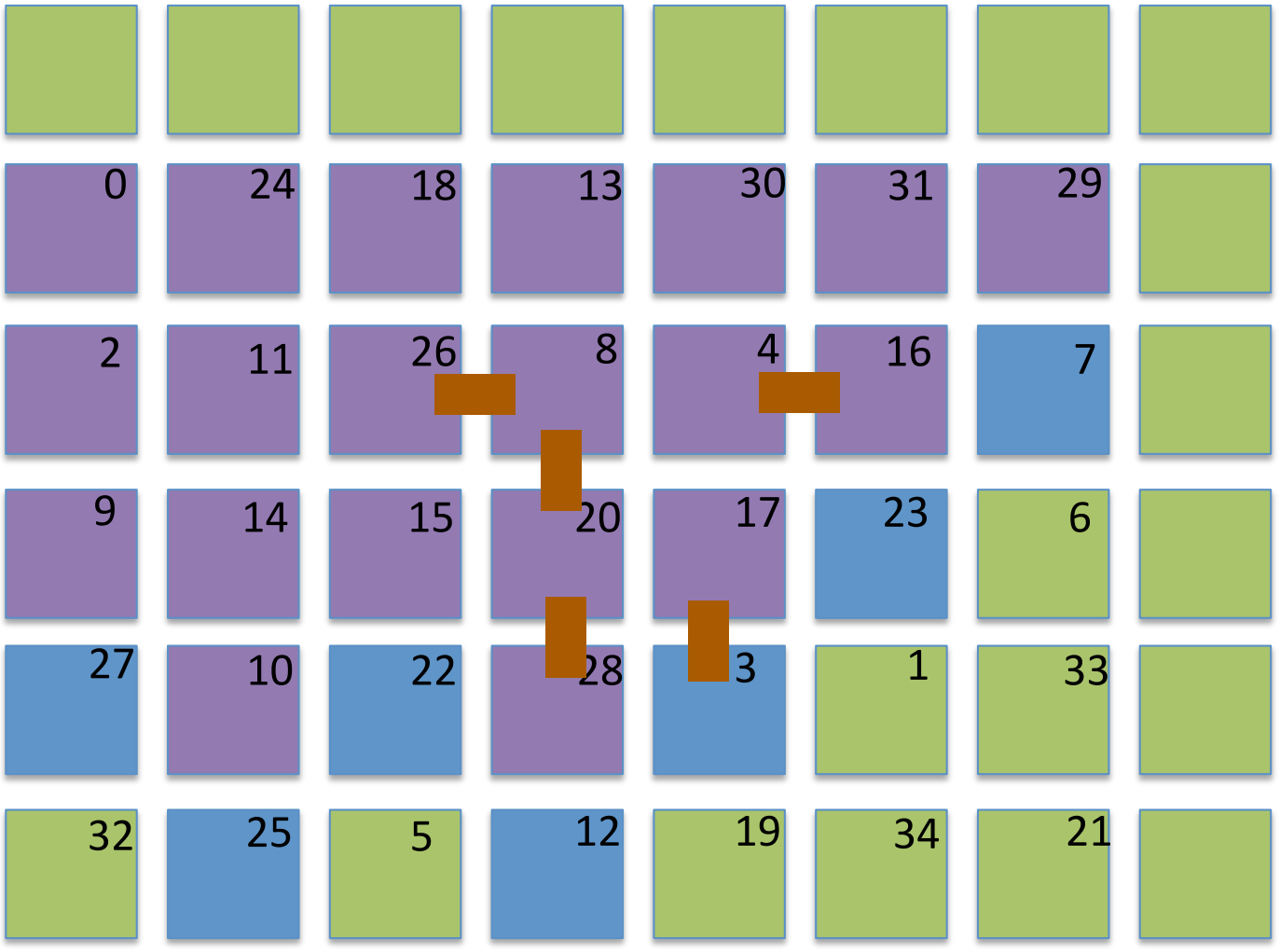
28 ← {16,29,27,22,25,3,23,12}



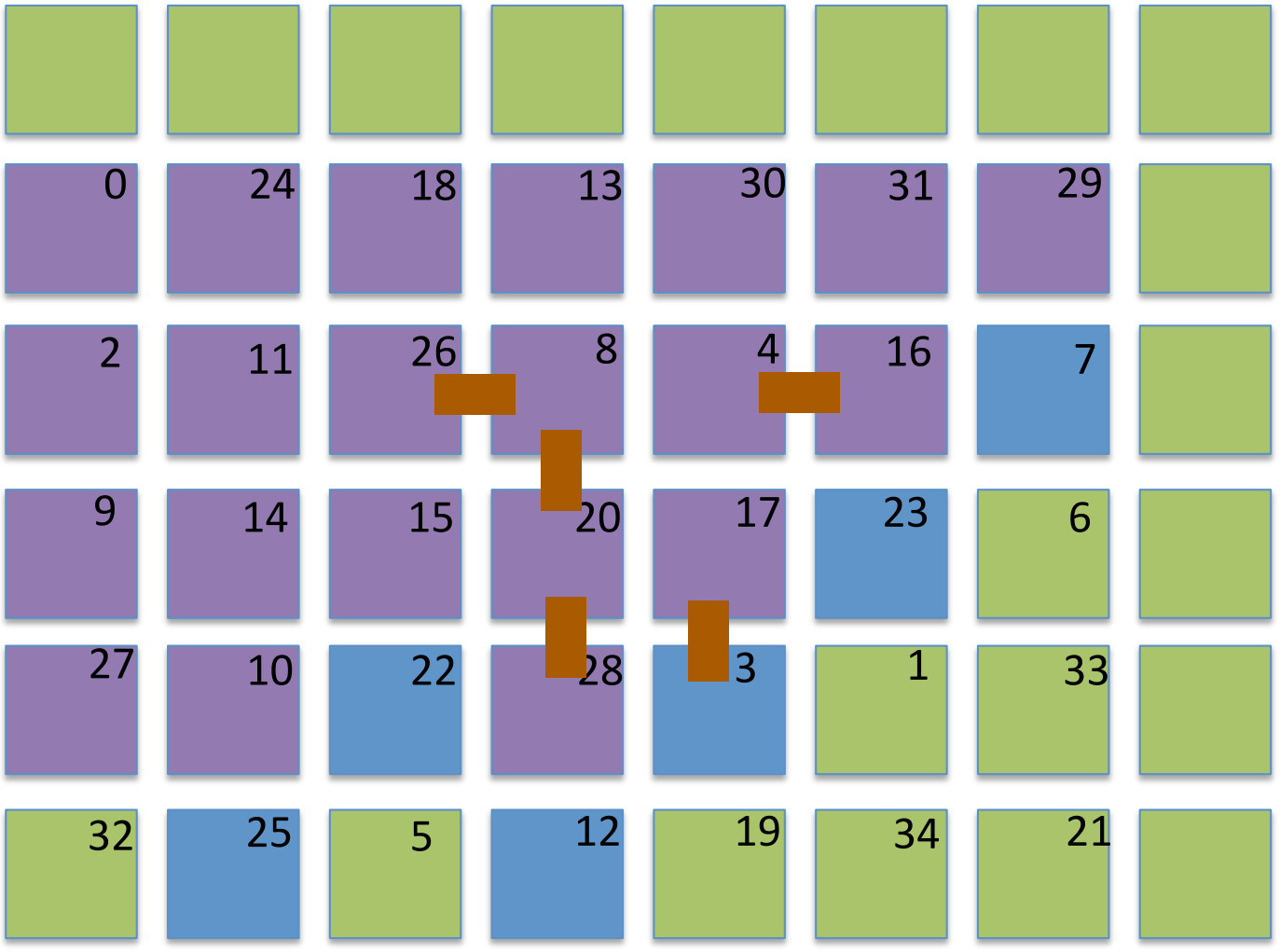
16 ← {29,27,22,25,3,23,12}



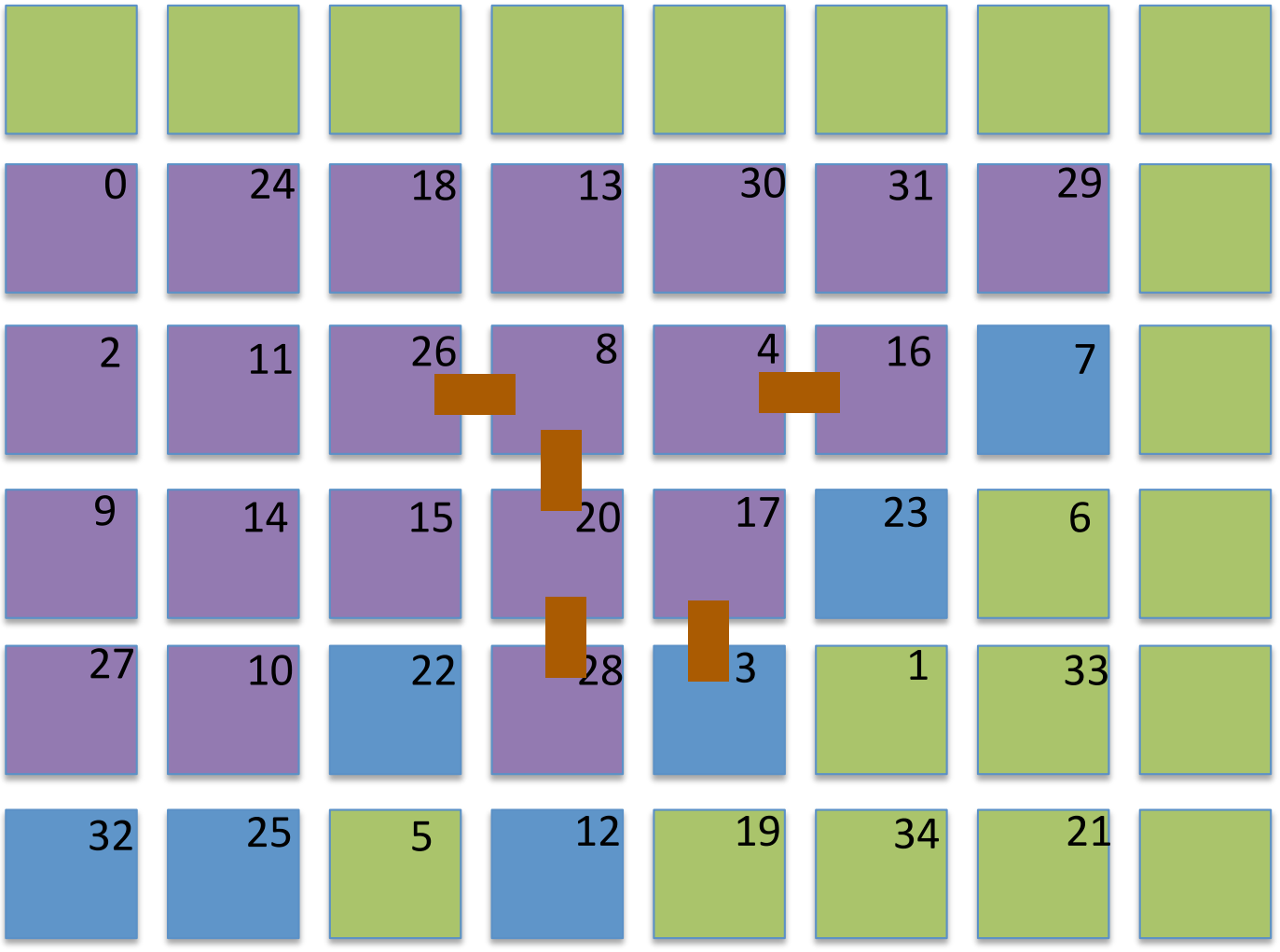
16 ← {29,27,22,25,3,23,12,7}



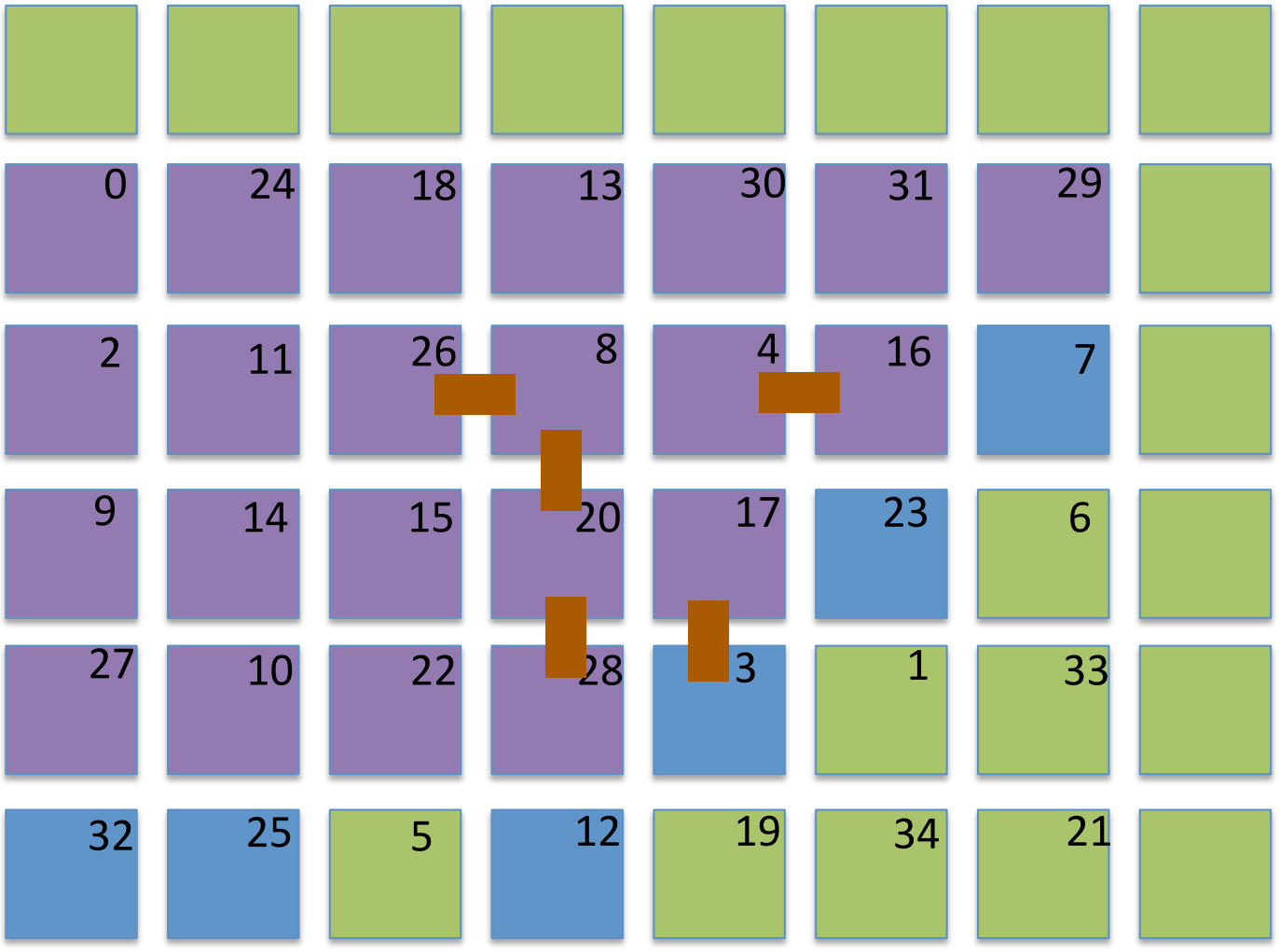
29 ← {27,22,25,3,23,12,7}



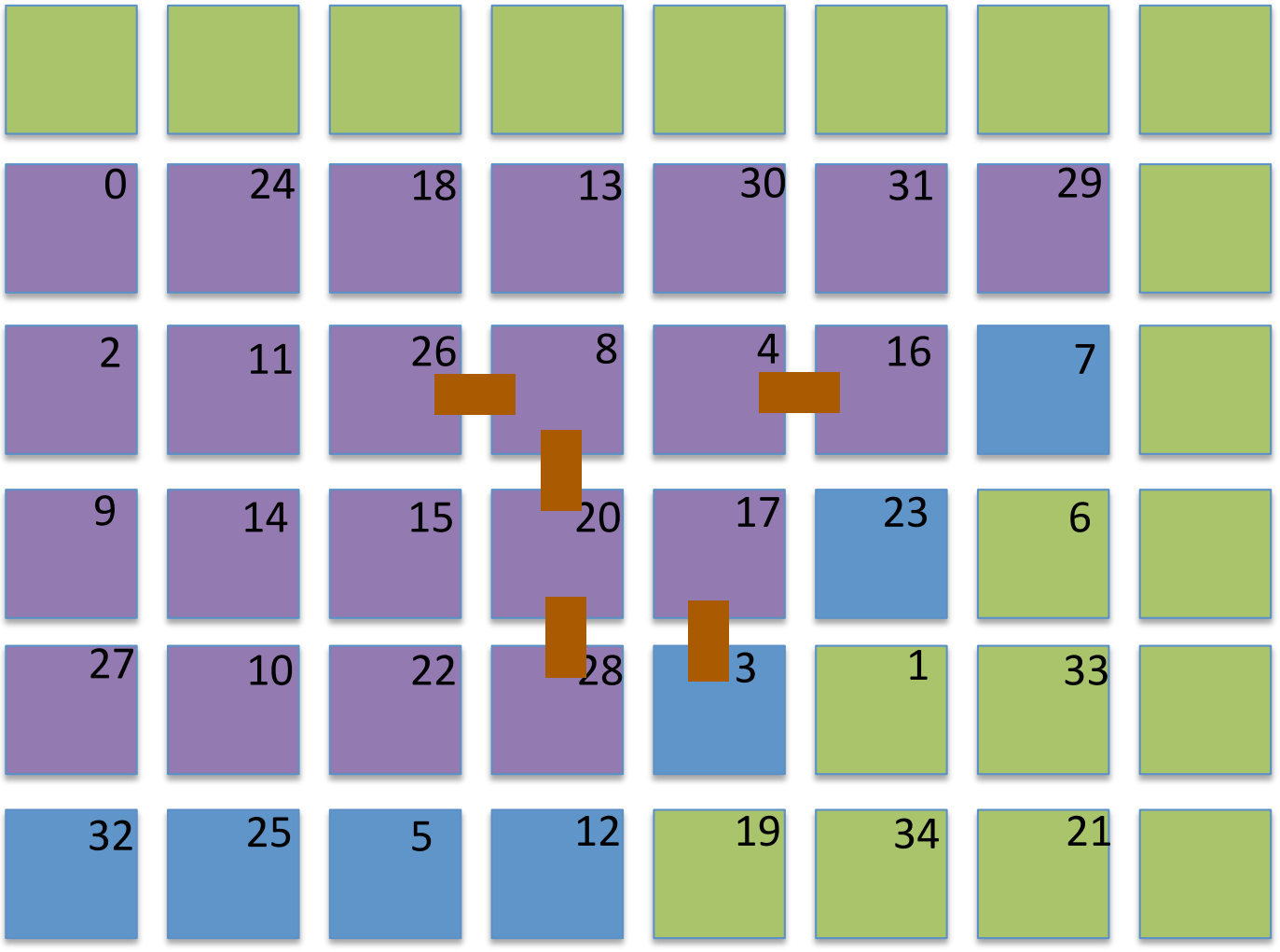
27 ← {22,25,3,23,12,7}



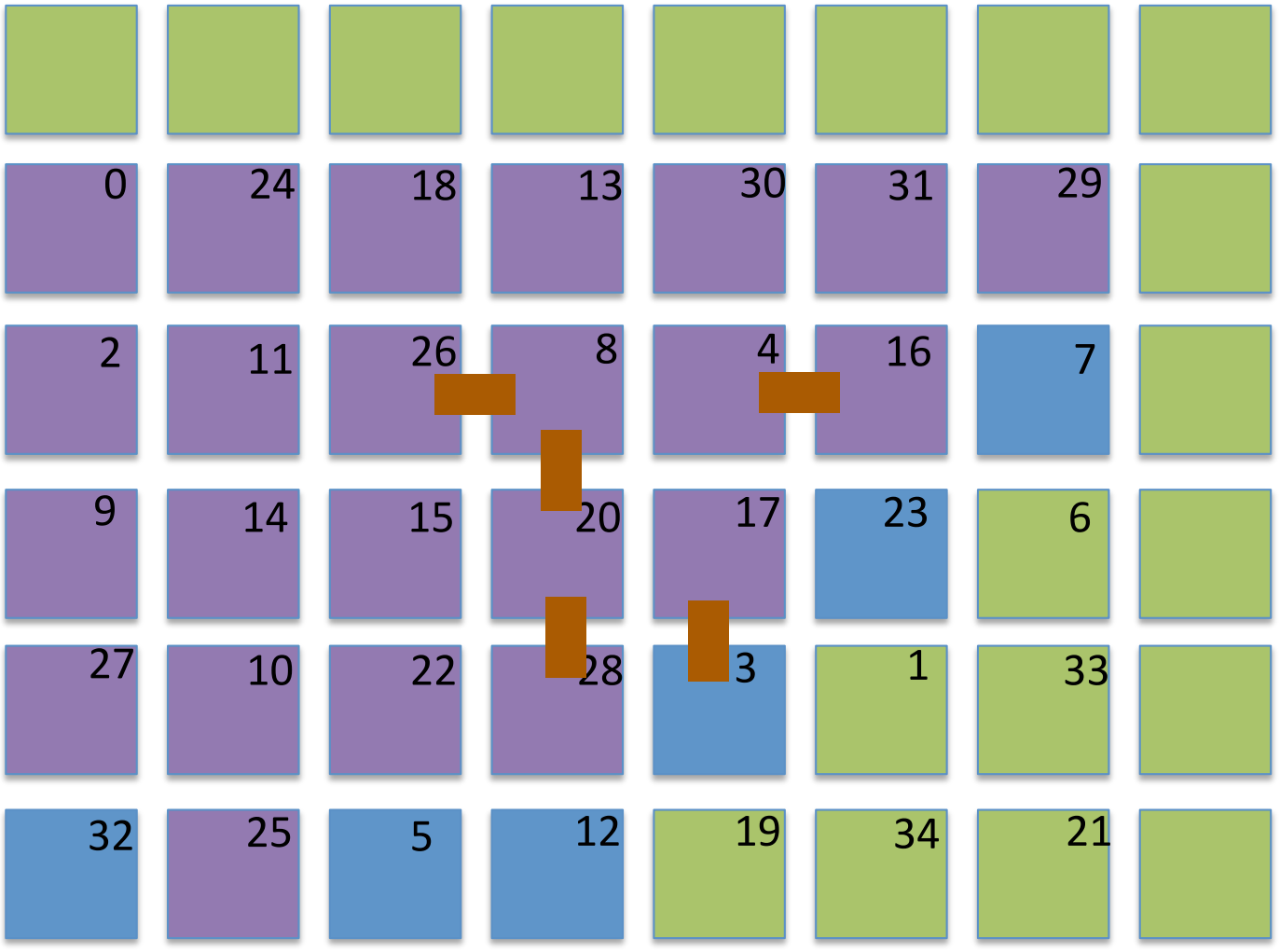
27 ← {22,25,3,23,12,7,32}



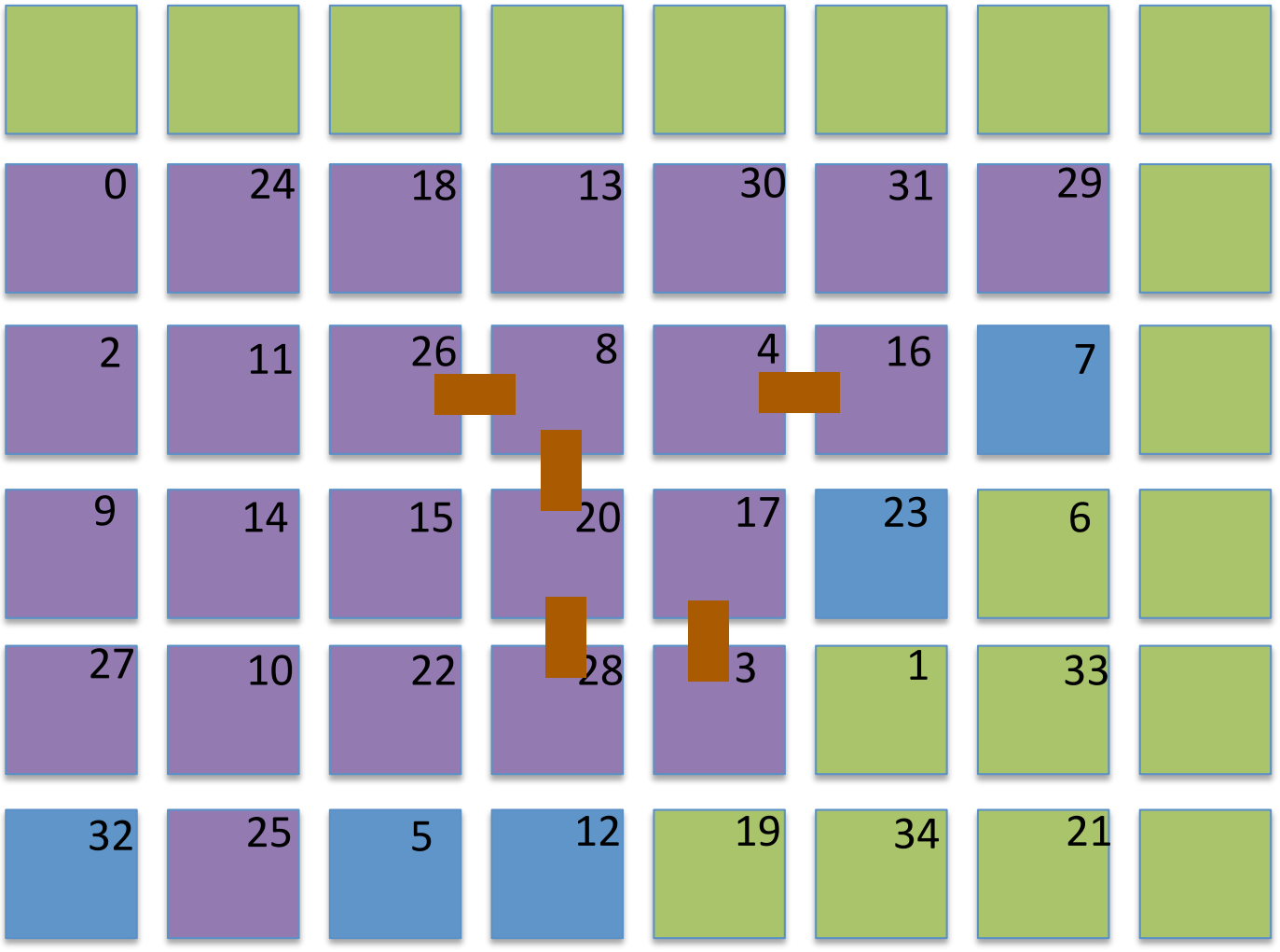
22 ← {25,3,23,12,7,32}



22 ← {25,3,23,12,7,32,5}



25 ← {3,23,12,7,32,5}


0 24 18 13 30 31 29
2 11 26 8 4 16 7
9 14 15 20 17 23 6
27 10 22 28 3 1 33
32 25 5 12 19 34 21

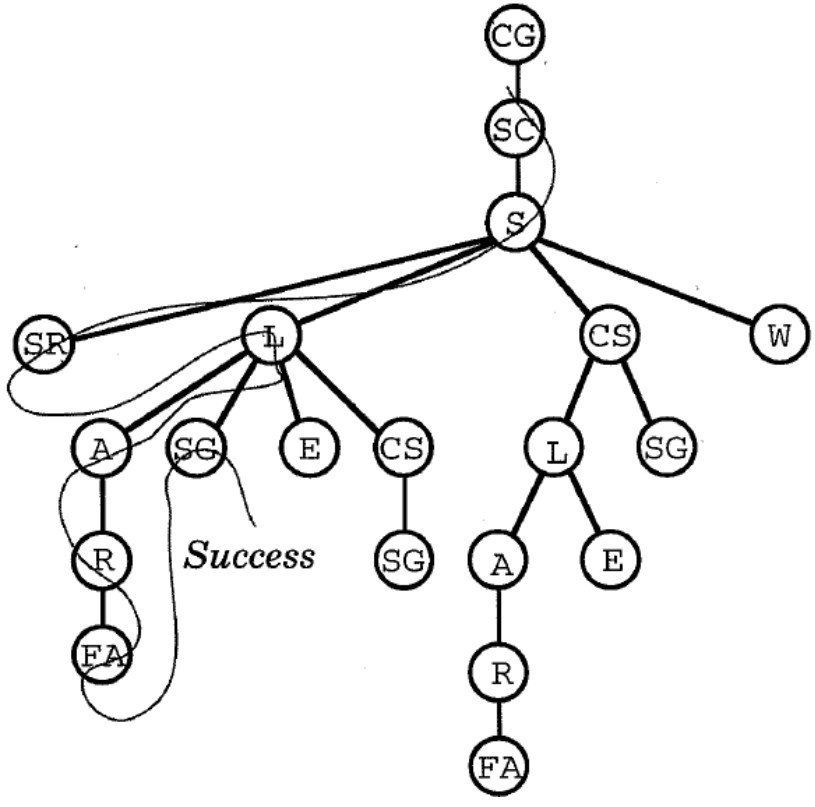
3 ← {23,12,7,32,5}

BFS with Path Costs

- Pure BFS using a FIFO queue pays no attention to the weight of a link
  - Okay for the Farmer Jones problem or the 8-puzzle problem
  - Not good for a map navigation problem with weighted paths.
- To modify BFS to cope with weighted graph, we should, at each step, select the node with the smallest accumulated path cost.
- It stays optimal, and complete, so long as costs are positive.
- But, now we require a different type of queue – a priority queue to manage the node selection strategy.

Depth-First Search

- Depth-first search traverses a search tree in a top-to-bottom, left to right fashion. It does this by expanding the left-most open node and working downward from that node until a leaf node is encountered.
- If no solution is found on this path then search continues from next lowest, left-most, unexpanded (closed) node.

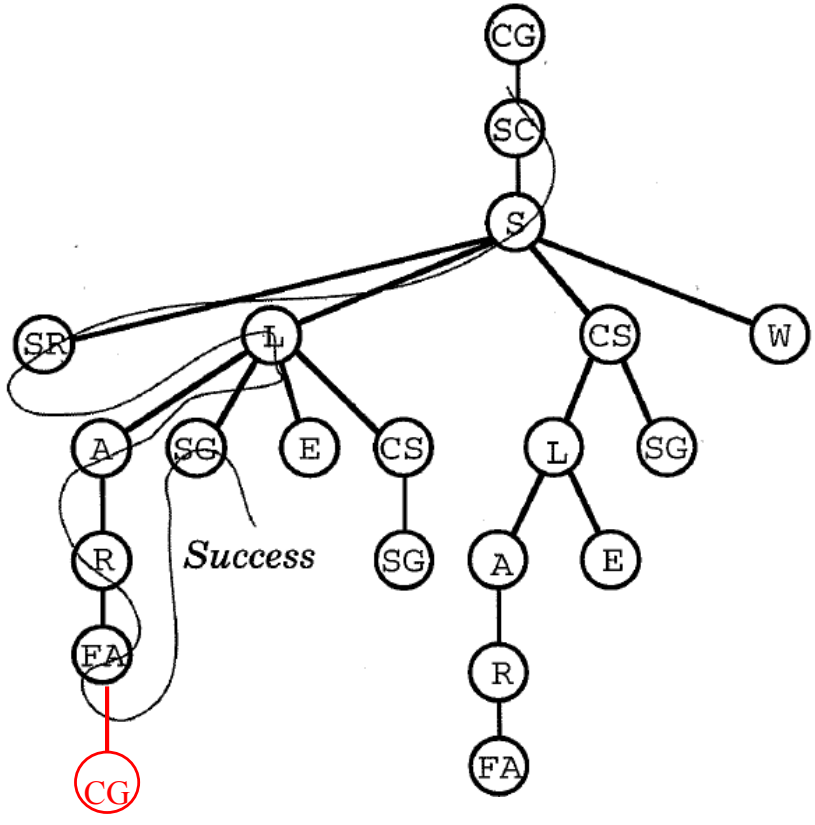


- **Queuing Function:** Queue new paths at start of queue. (LIFO)

Time complexity	Space complexity	Complete?	Optimal?
$O(b^d)$	O(bd)	No	No

Depth-First Search-Complete?

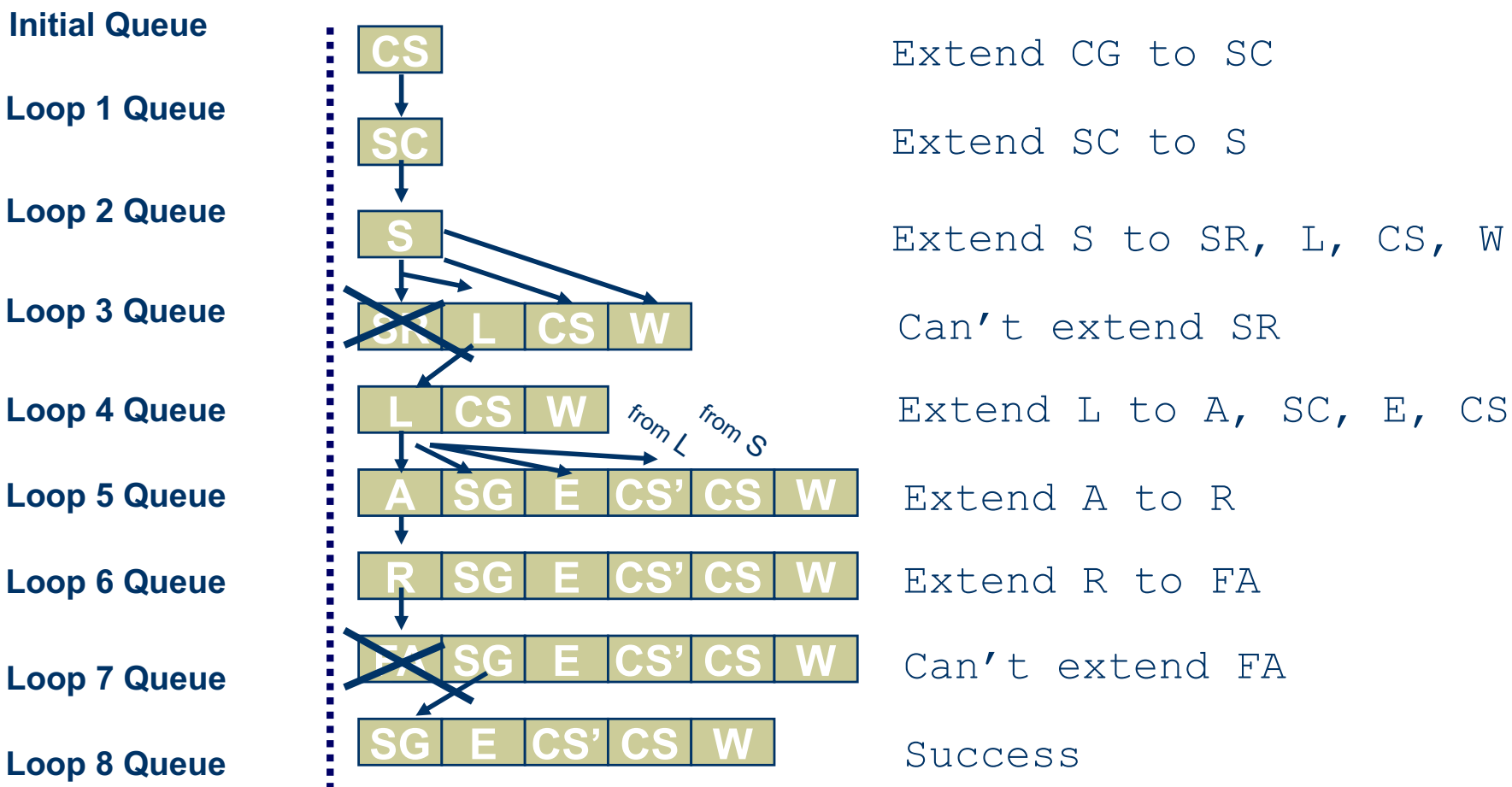
- If carrying out tree search, then we do not keep track of expanded nodes in a closed-list.
- Hence, if a path loops back to the same node, DFS may enter an infinite loop.
- So, DFS is not complete for tree search, in general.
- If the search space is finite and we can avoid expanding nodes more than once, then eventually DFS will explore the entire search space, and will find a solution if one exists.



Suppose path leads back to initial state.

Depth-First Search

- Here is a trace of the depth-first procedure on the UCD route finding problem.



Evaluating Complexity

- **Time Complexity**
  - Assume that the tree is finite and has a maximum depth *m*. In the worst case (where *d*=*m* and the goal node is the last node on level *m*), DFS will visit all nodes in the tree. That is $O(b^m)$.

$$1 + b + b^2 + b^3 + \ldots + b^m$$

  - The best-case time complexity is just *d+1* (i.e., *O(d)*)
    - if the first path we explore is the one that contains the shallowest goal, then we walk straight down to the goal, with no time spent on fruitless paths.

Evaluating Complexity

- **Space Complexity**
  - DFS has excellent space complexity. It needs to store the start node. Then at level 1 it stores *b* successors of the start node. Then at level 2 it stores *b* successors of one of these nodes. At level three the *b* successors of one of the level 2 nodes are stored. This goes on, in the worst case, until the path can be extended no further (level *m*).
  - When it moves onto another path, the space that was being used for the path that it can extend no further can be reclaimed and reused. Hence, the worst case space complexity is *O(bm).* It's space requirement is linear.

A Time/Space Comparison

- **Breadth-First Search**
  - **Assuming…**
    - *b* = 10,
    - 10,000 nodes/sec;
    - 1000 byte/node

DEPTH	NODES	TIME	MEMORY
2	1100	.11 seconds	1 megabyte
4	111,100	11 seconds	106 megabytes
6	10^7	19 seconds	10 gigabytes
8	10^9	31 hours	1 terabyte
10	10^{11}	129 days	101 terabytes
12	10^{13}	35 years	10 petabytes
14	10^{15}	3,523 years	1 exabyte

- **Depth-First Search**
  - **Assuming…**
    - ***That nodes at the same depth as the goal have no successors…***
    - ***118 kilobytes instead of 10 petabytes at depth d = 12***
    - ***A factor of 10 billion times less space***

When to Use Which…

- **Depth-first search** is a good idea if you are confident that all partial paths either reach dead ends or become complete solutions at reasonable depths.

- In contrast **breadth-first search** can work well when this is not the case, even in trees that are infinitely deep. But breadth first methods are wasteful if all goal nodes are reached at more or less the same depths or if the branching factor is large (in fact even moderate branching factors result in exponential growth).

Random Queuing

- If you have no idea about the topology of the search tree – for example, if you do not know the branching factor or if you cannot rule out the possibility of unmanageably deep solutions paths – then the idea of **random queuing** offers a pragmatic *middle-ground* between depth-first and breadth-first methods.

- In this type of search paths are not added at the start or end of the queue. Instead they are inserted in a random queue position; this is often referred to as *non-deterministic search*. The effect is that search tree nodes are expanded at random. **The search is equally likely to deepen as it is to broaden.**

- **Queuing Function:**
  - Queue new paths at random positions in queue.

Depth-Limited Search

- This search technique avoids the pitfalls of depth-first search by imposing a limit to prevent search from going beyond a certain depth.
- For example, in the UCD problem we know that no solution will need more than 11 steps because there are only 12 locations. So we can use 11 as a limit and then we need not consider the problem of following infinitely deep paths.
- We can implement this type of search in a number of ways. One is to modify our operator descriptions to take account of the limit: -

Move(x,y) iff Edge(x,y) and Depth < l (where l refers to depth limit)

Depth-Limited Search

- If a solution exists within the specified depth limit, then it will be found.
- However, since operating in DFS mode, a found solution is not necessarily **optimal**.
- **Complexity viewpoint**: similar to standard DFS, but with the new depth limit playing the role of the maximum tree depth.

Time complexity	Space complexity	Complete?	Optimal?
$O(b^l)$	$O(bl)$	Yes	No

The Best of Both Worlds!

- It is often difficult to find a good search limit to use with depth-limiting methods. Manages to combine DFS and BFS.
- In general, iterative deepening is the preferred search method when there is a **large search space** and **the depth of the solution is not known**.
- Iterative deepening search enjoys the linear memory requirements of depth first search while guaranteeing that a target/goal node of minimal depth will be found (if a goal can be found at all).
- Successive depth-first searches are conducted – each with depth bounds increasing by 1 – until success is achieved.

Iterative Deepening

- **Tournament chess** is played under a strict time control, and a program must make decisions about how much time to use for each move. Most chess programs do not set out to search to a fixed depth, but use a technique called **iterative deepening**.

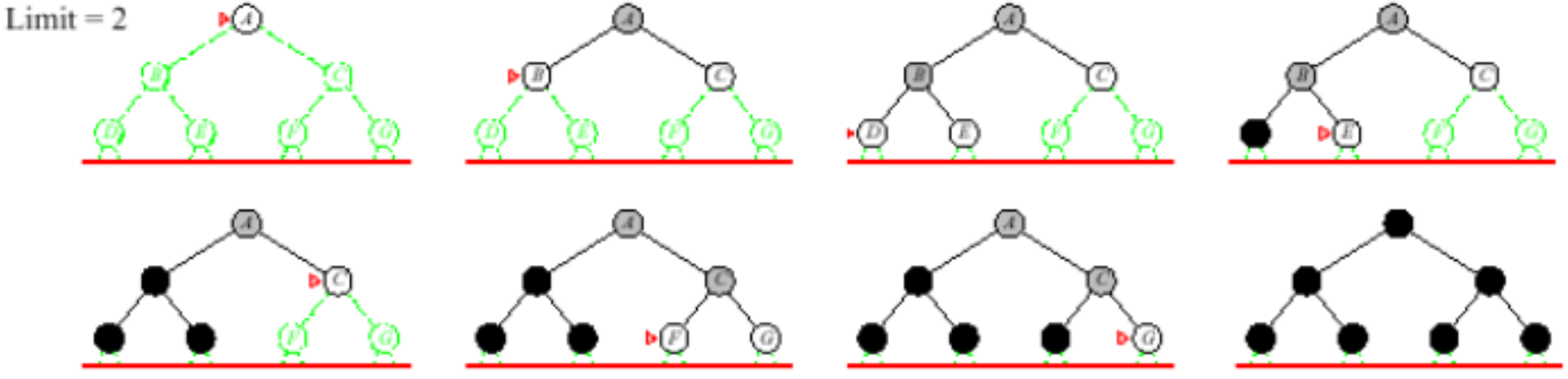
- This means a program does a depth two search, then a depth three search, then a depth four search, and so on until the allotted time has run out.

- When the time is up, the program returns its current best guess at the move to make.

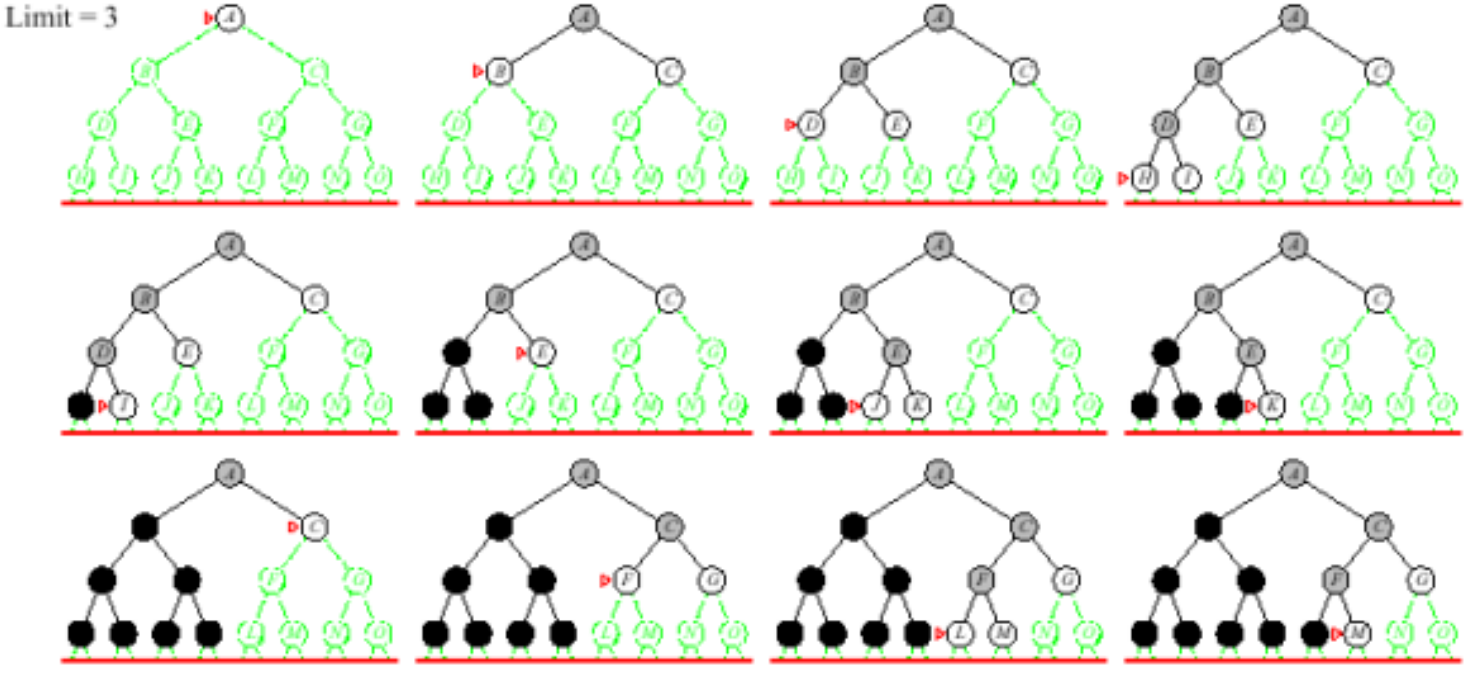
Iterative Deepening

Limit = 1

Limit = 2



Iterative Deepening



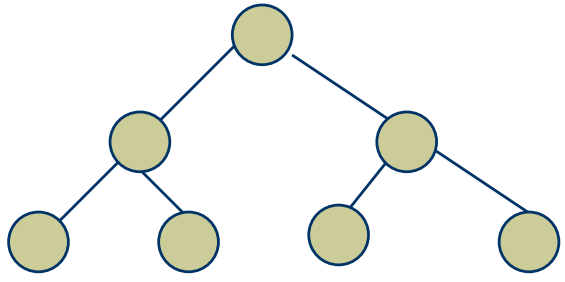
Time complexity	Space complexity	Complete?	Optimal?
$O(b^d)$	O(bd)	Yes	Yes

Iterative Deepening

- The number of nodes expanded by a breadth-first search could be as high as:

  $1 + b + b^2 + \ldots + b^d$ expansions $= (b^{d+1} - 1)/(b- 1)$

- That is, one expansion for the root, b for the level a nodes, b^2 for the level 3 nodes and so on…



$b = 2$
$d = 3$

- For if **b=10, d=5** that equates to **111,111** expansions!

Iterative Deepening

- To calculate the number of nodes expanded by iterative deepening we first note that the number of nodes expanded by a complete depth-first search down to a depth *m* is the same:
  - **i.e., $1 + b + b^2 + \ldots + b^d$ expansions**

- Worst case, iterative-deepening search for a goal at depth *d* has to conduct separate, complete depth-first searches for depths up to *d*.

- Thus, the number of expansions is computed as follows …

$(L+1)1 + (L)b + (L-1)b^2 + \ldots + b^L$ expansions

{123,456 if b=10, L=5; an increase of only 11% over BFS}

Repeated States

- In general, problems with repeated states have *infinite* search trees.
- If repeated states are taken care of (that is: they do not happen), most of these search trees become finite: they are bounded by the size of the state space.
- E.g. the 8-puzzle has ~181k reachable states. That is the maximum size of the search tree if no state is visited twice.
- If you manage to not visit states multiple times, your branching factor will also go down.
-

Repeated States

- However, to avoid visiting a state twice, you have to keep track of it..
- This may be feasible for the 8-puzzle (you can keep 181k states in memory, and even access them quickly), but it isn't for larger problems.
- What if the search space contains 10^{15} states? In the worst case scenario (you find the solution after visiting them ~all), you would need to keep them all in memory..

Repeated States

- Even for those problems where you can keep track of all states, there will be an overhead because you need to check each state against the visited list before adding it to the fringe.
- In some cases you can implement a quick hash. But the larger the state space the harder.
- Reduced visits are traded off against lengthier times for expansion.
- Note that keeping the closed-list may well destroy the linear memory requirements of DFS and DFS-ID.
- Careful consideration of trade-offs is required on a problem-by problem basis