Zero alloc pathfinding

@quasilyte 2023

What this talk ***is***

A dive into specialized data structures and micro-optimizations

What this talk ***is***

A dive into specialized data structures and micro-optimizations

Something that you would easily use in your job

…but ***not*** this

Agenda

1. A very short intro
2. Existing libraries
3. Why my library is so fast
4. How to overcome some of its limitations

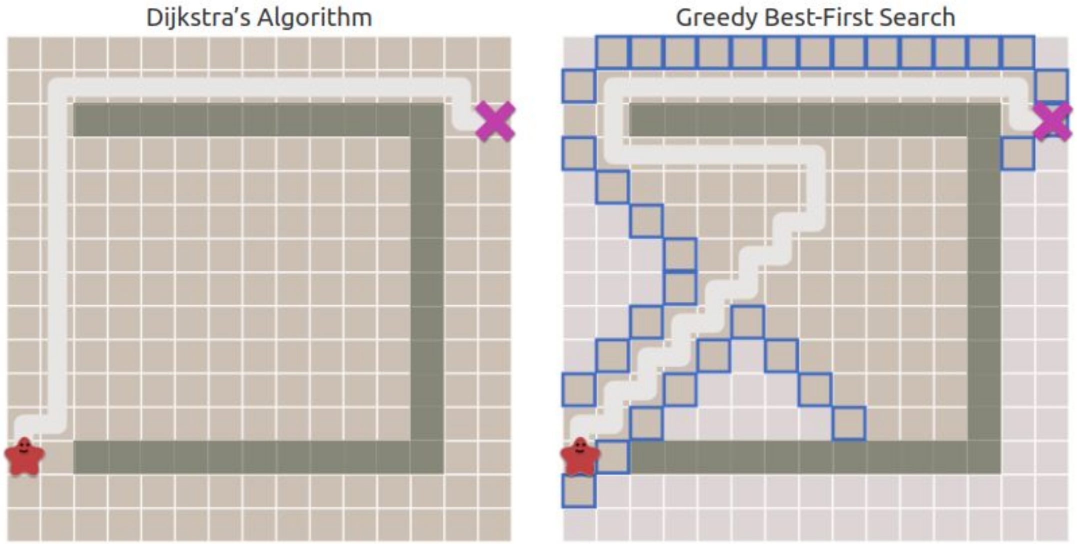


Image source: www.redblobgames.com/pathfinding/a-star

Pathfinding finds some way to get from point A to point B.

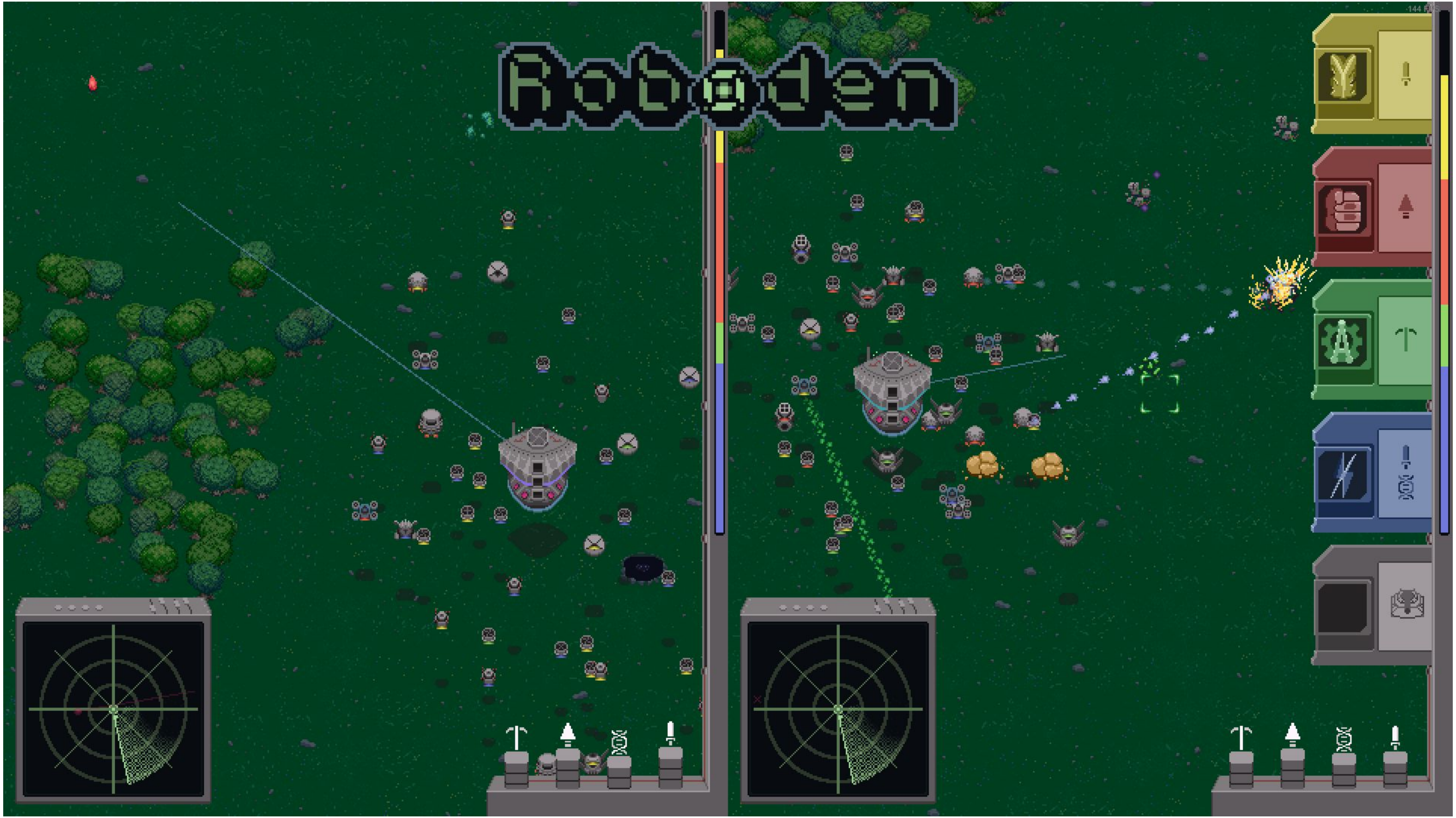
Depending on the algorithm, the paths can have different properties.

Roboden

Open source!

About me & pathfinding


Roboden


Roboden

Roboden & pathfinding

- Can have large maps (scrollable, several screens)
- Maps are generated, there are no key waypoints
- Different kinds of landscape (mountains, lava, forests, …)
- Hundreds of units that are active in the real-time
- Fixed 60 ticks per second

Pathfinding libraries

1. github.com/quasilyte/pathing (my library)
2. github.com/fzipp/astar
3. github.com/beefsack/go-astar
4. github.com/s0rg/grid
5. github.com/solarlune/paths

Let’s benchmark libraries!

Tests:

- **no_walls** - the trivial solution is the best one
- **simple_wall** - go around a simple wall
- **multi_wall** - several simple walls

50x50 grid map

Benchmark details (for nerds)

Sources: github.com/quasilyte/pathing/_bench

OS: Linux Mint 21.1

CPU: x86-64 12th Gen Intel(R) Core(TM) i5-1235U

Tools used: “go test -bench”, benchstat

Turbo boost: disabled (intel_pstate/no_turbo=1)

Go version: devel go1.21-c30faf9c54

Pathfinding benchmarks: CPU time (ns/op)

LIBRARY	no_walls	simple_wall	multi_wall
pathing	3525	2084	2688
astar			
go-astar			
grid			
paths			

Pathfinding benchmarks: CPU time (ns/op)

LIBRARY	no_walls	simple_wall	multi_wall
pathing	3525	2084	2688
astar	(x268) 948367	(x745) 1554290	(x685) 1842812
go-astar			
grid			
paths			

Read as “X times slower”

Pathfinding benchmarks: CPU time (ns/op)

LIBRARY	no_walls	simple_wall	multi_wall
pathing	3525	2084	2688
astar	(x268) 948367	(x745) 1554290	(x685) 1842812
go-astar	(x128) 453939	(x450) 939300	(x343) 1032581
grid	(x514) 1816039	(x553) 1154117	(x442) 1189989
paths	(x1868) 6588751	(x2474) 5158604	(x2274) 6114856

Pathfinding benchmarks: allocations

LIBRARY	no_walls		simple_wall		multi_wall	
pathing		0		0		0
astar	337336 B	2008	511908 B	3677	722690 B	3600
go-astar	43653 B	529	93122 B	1347	130731 B	1557
grid	996889 B	2976	551976 B	1900	740523 B	1759
paths	235168 B	7199	194768 B	6368	230416 B	7001

quasilyte/pathing

- x128-2474 times faster than the alternatives
- Does no heap allocations to build a path
- Optimized for very big grid maps (thousands of cells)
- Simple and zero-cost layers system
- Works out of the box, no need to implement an interface

Instead of talking about this…

Why other libraries are so much slower?

Instead of talking about this…

Why other libraries are so much slower?

Why pathing is so fast?

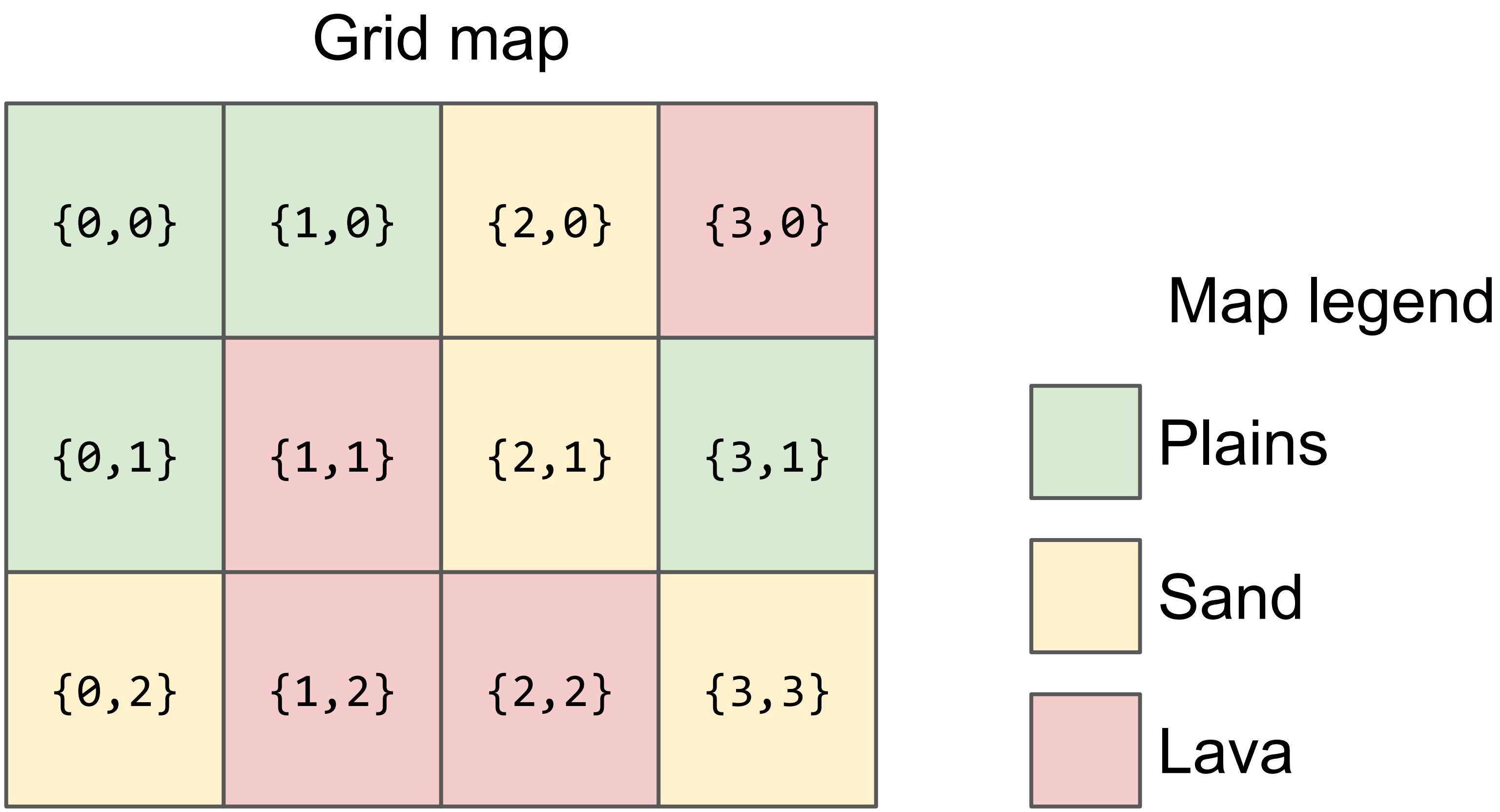
…we'll focus on this

Greedy BFS performance-critical parts

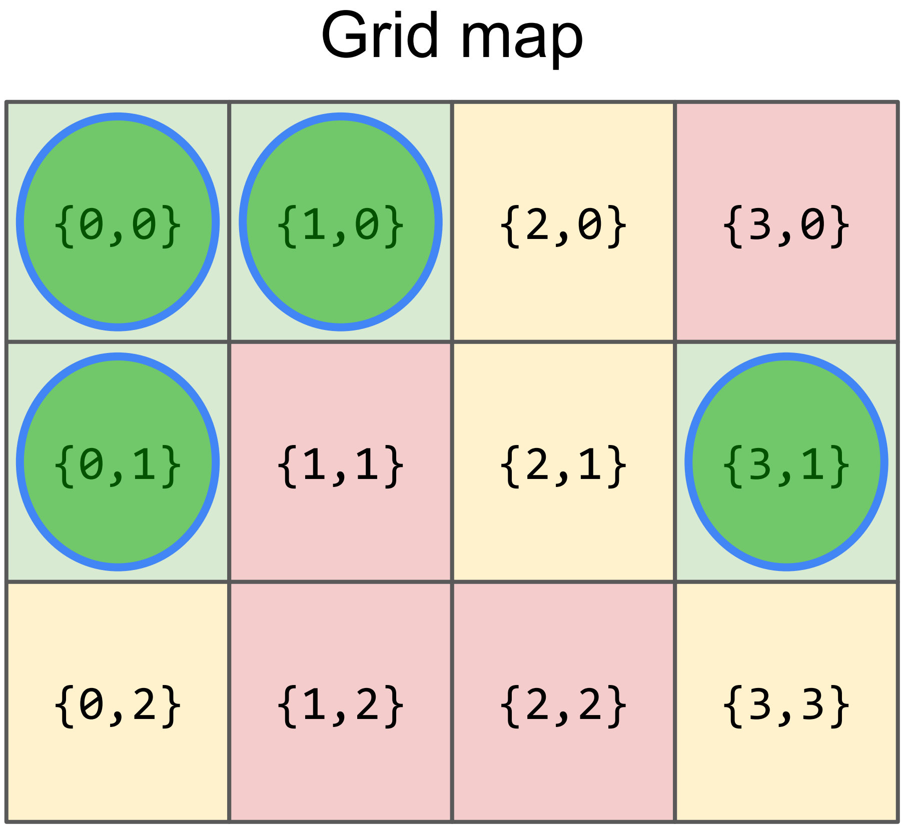
- Matrix to store cell information (the grid)
- Result path representation
- Priority queue for “frontier”
- Map to store the visited cells

Greedy BFS performance-critical parts

- Matrix to store cell information (the grid)
- Result path representation
- Priority queue for “frontier”
- Map to store the visited cells

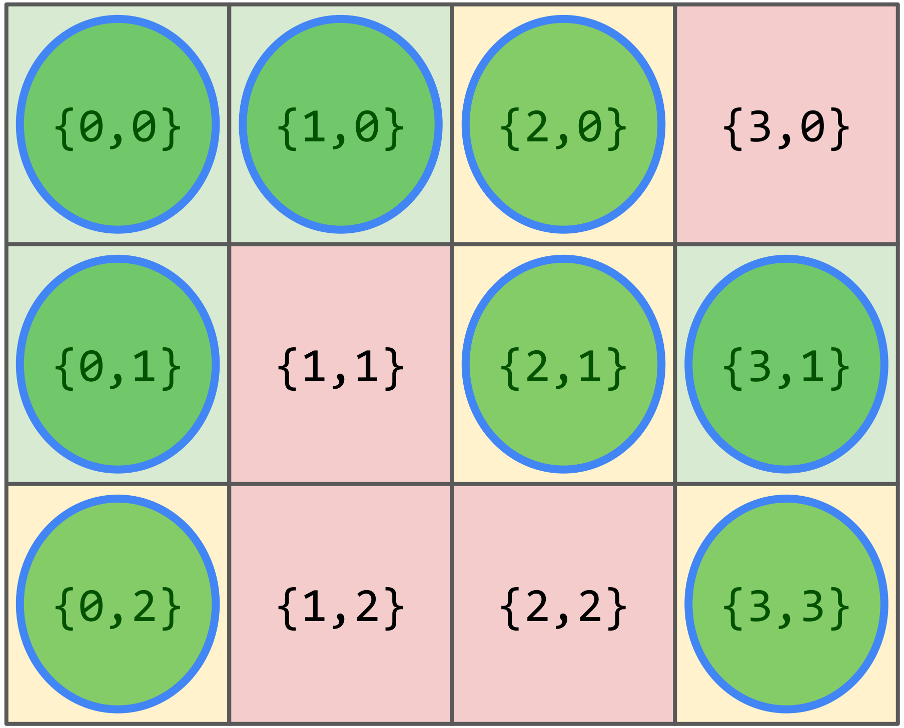


Grid map
{0,0}
{1,0}
{2,0}
{3,0}
{0,1}
{1,1}
{2,1}
{3,1}
{0,2}
{1,2}
{2,2}
{3,3}
Map legend
Plains
Sand
Lava



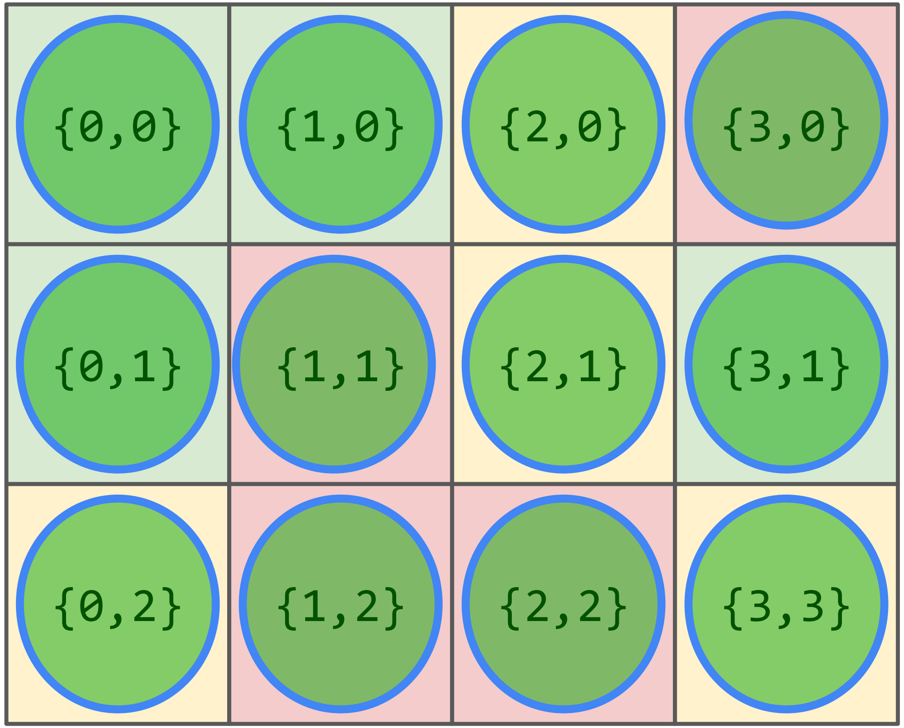
Some units can only traverse the plains.

Grid map



Other units can traverse sand as well

Grid map



Flying units can probably cross lava too

- Same grid map
- Different cell interpretation

???

- Same grid map
- Different cell interpretation

Enter grid cell mappers (“layers”)

Defining the grid cell types

```go
type CellType int


const (
  CellPlain CellType = iota // 0
  CellSand                  // 1
  CellLava                  // 2
)
```

Storing the grid cell data

- We have only 3 tile types

=>

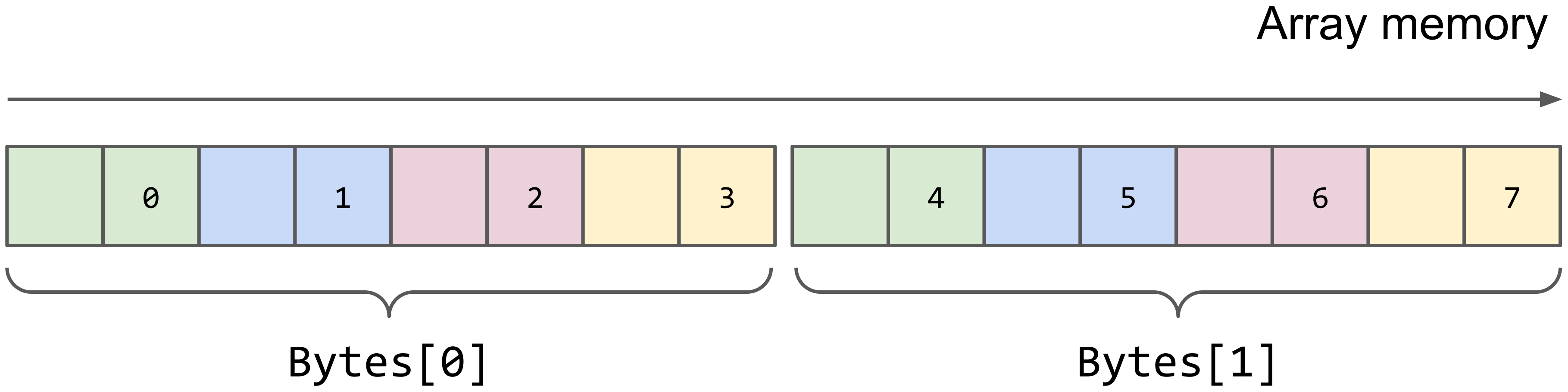
- 2 bits per cell are enough

Choosing a grid cell data size (in bits)

SIZE (bits)	VALUE RANGE	MAP SIZE (3600 CELLS)
2	0-3 (2^2)	900 bytes
3	0-7 (2^3)	1350 bytes
4	0-15 (2^4)	1800 bytes
5	0-31 (2^5)	2250 bytes
6	0-63 (2^6)	2700 bytes

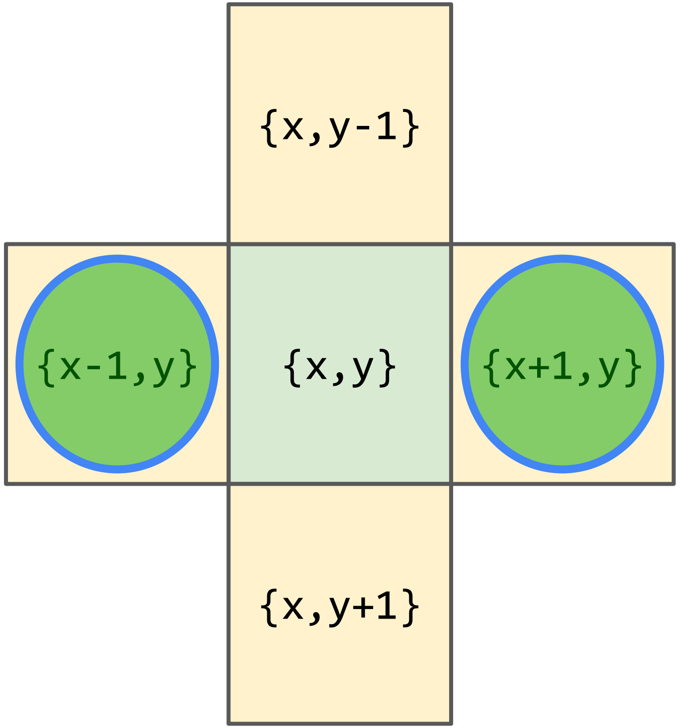
A typical cache line size is only 64 bytes

Flat array storage (2 bits / cell)



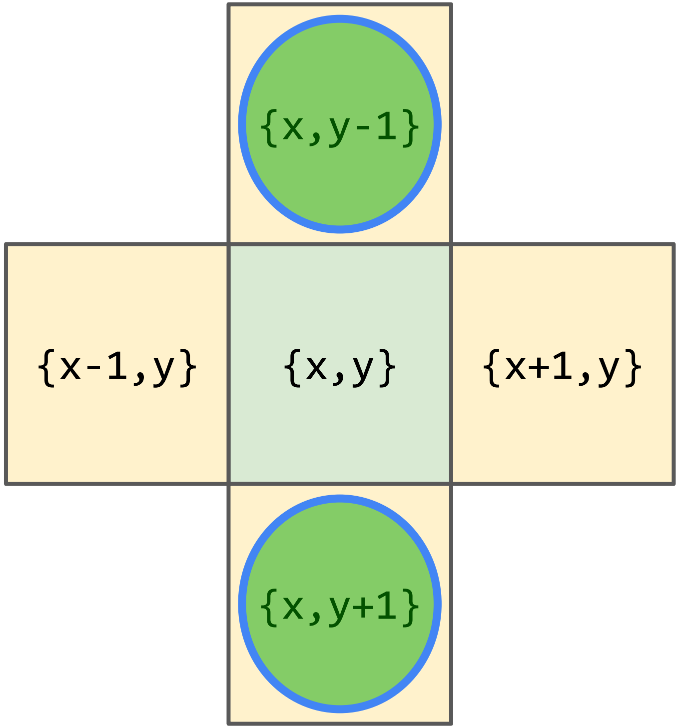

```
i := y*g.numCols + x
byteIndex := i / 4
shift := (i % 4) * 2
cell := g.bytes[byteIndex] >> shift & 0b11
```

Grid access patterns



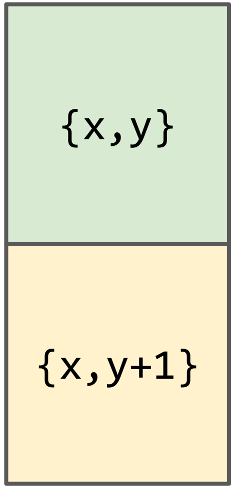
These cells may be inside the same byte => good cache locality

Grid access patterns



These cells may not even fit into the same cache line (64 byte)

Grid access patterns




```
index = numCols*y + x
```

Map width is our limiting factor

Cache lines vs flat array storage

SIZE (bits)	MAX WIDTH (cells, 32x32 pixels)
2	256 cells (8192 pixels)
3	~170 cells (5440 pixels)
4	128 cells (4096 pixels)
5	~102 cells (3264 pixels)
6	~85 cells (2720 pixels)

Querying the grid cell value

1. Get a raw value from the grid at {x,y} (0-3 for 2 bits)

Querying the grid cell value

1. Get a raw value from the grid at {x,y} (0-3 for 2 bits)
2. Map that through the cell mapper (get a 0-255 value)

Querying the grid cell value

1. Get a raw value from the grid at {x,y} (0-3 for 2 bits)
2. Map that through the cell mapper (get a 0-255 value)
3. Use that value during the pathfinding

Interpreting the cell value

- After the mapping, cell value is in [0,255] range
- 0 means that this cell can’t be traversed
- Any other value specifies the traversal cost

For the simplest cases, 0 and 1 are enough.

Defining the layer

```go
type GridLayer [4]byte

func (l GridLayer) Get(i int) byte {
  return l[i]
}
```

Declaring layers (user code)

```
var NormalLayer = pathing.GridLayer{
  CellPlain: 1,
  CellSand:  0,
  CellLava:  0,
}
```

Declaring layers (user code)

```
var FlyingLayer = pathing.GridLayer{
  CellPlain: 1,
  CellSand:  1,
  CellLava:  1,
}
```

A better way to define the layer

```go
type GridLayer uint32

func (l GridLayer) Get(i int) byte {
  return byte(l >> (uint32(i) * 8))
}
```

Greedy BFS performance-critical parts

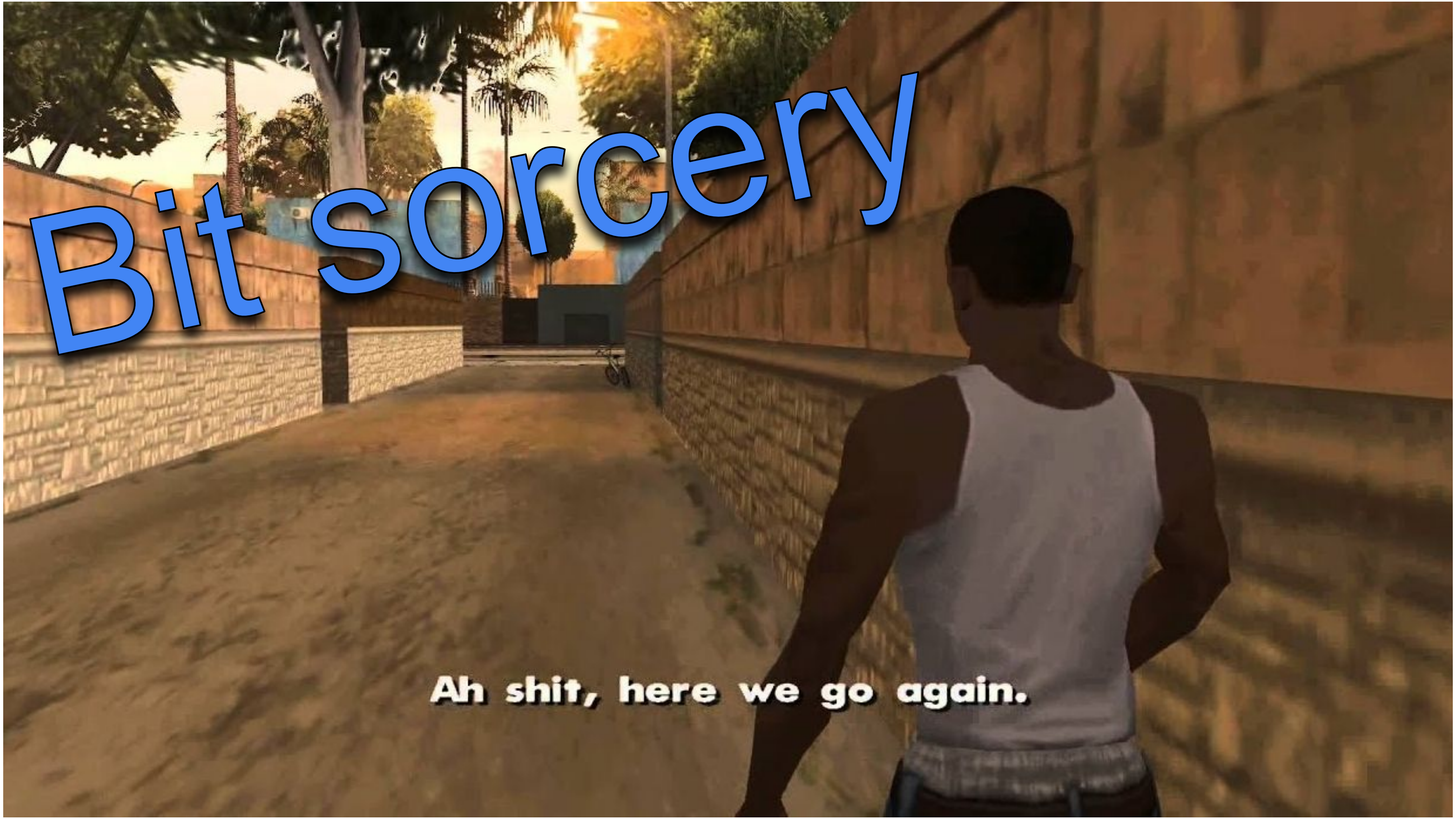
- Matrix to store cell information (the grid)
- Result path representation
- Priority queue for “frontier”
- Map to store the visited cells

How to represent the path?

Array (slice) of points

How to represent the path?

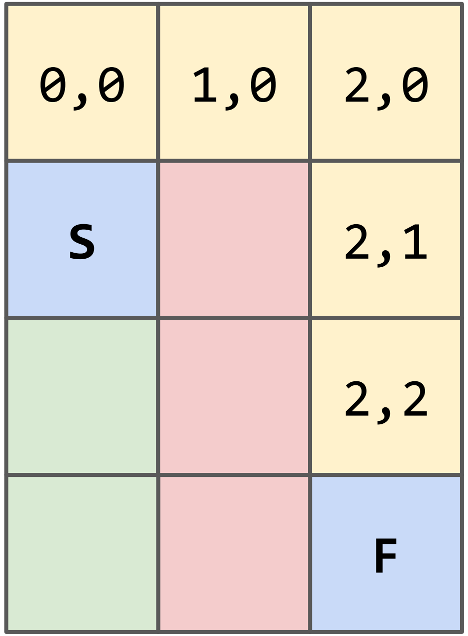
Packed deltas


Bit sorcery
Ah shit, here we go again.

A path with points

{0,0}	{1,0}	{2,0}	{2,1}	{2,2}	{2,3}

Map



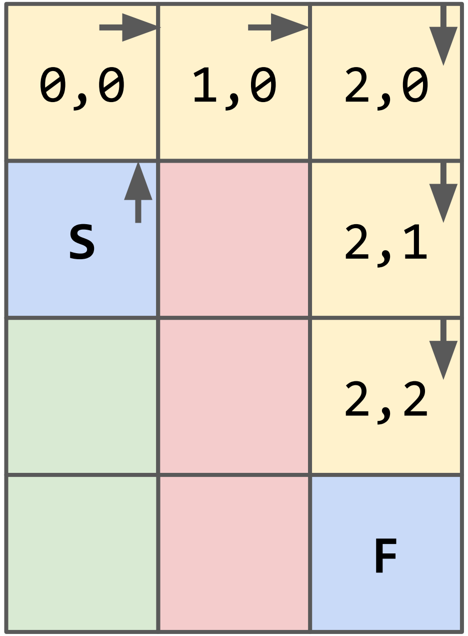
A path with points

{0,0}	{1,0}	{2,0}	{2,1}	{2,2}	{2,3}

A path with deltas (“steps”)

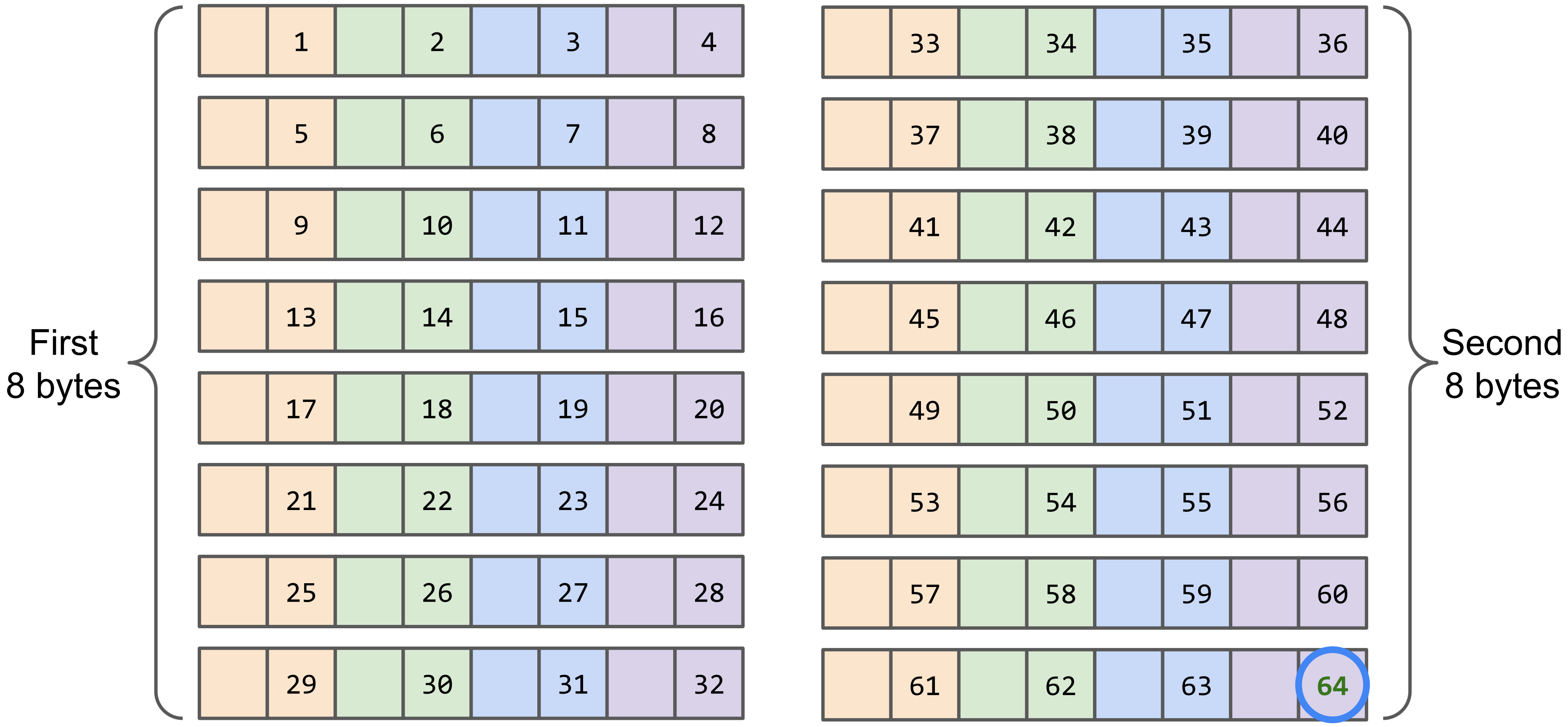
Up	Right	Right	Down	Down	Down

Map

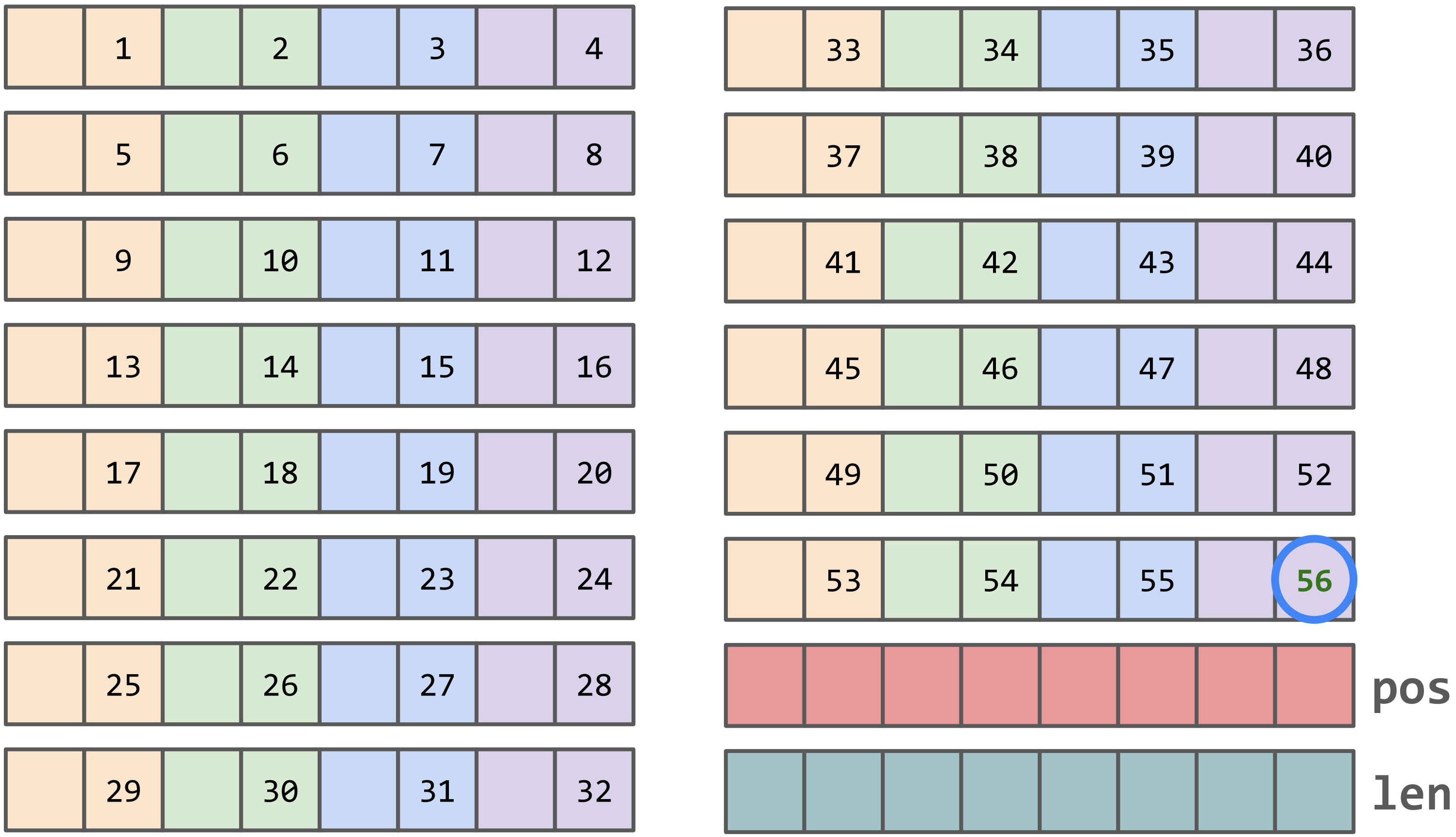


Defining cardinal directions (no diagonals for now)

```go
type Direction int
// Just 4 values, so 2 bits per direction will suffice!
const (
  DirRight Direction = iota // 0
  DirDown                   // 1
  DirLeft                   // 2
  DirUp                     // 3
)
```



We could store up to 64 “steps” in just 16 bytes (2 registers)



But we also need pos+len to iterate over the path

Defining the Path structure

```
const gridPathBytes  = (16 - 2)          // 14
const gridPathMaxLen = gridPathBytes * 4 // 56


type GridPath struct {
  bytes [gridPathBytes]byte
  len   byte
  pos   byte
}
```

Our Path type advantages

- Value semantics - just pass it without a pointer
- Copying is trivial - a pair of 64-bit MOVQ instructions
- No need to do a heap allocations
- Requires no real memory fetching, unlike dynamic arrays

Our Path type disadvantages

- Limited max path length

SIZE (bytes)	MAX PATH LENGTH (steps)
16	56
24	88
32	120

There is an advantage in keeping it under 64

Greedy BFS performance-critical parts

- Matrix to store cell information (the grid)
- Result path representation
- Priority queue for “frontier”
- Map to store the visited cells

Frontier & priority queue

- Push(coord, score)
- Pop( ) -> (coord, score)
- Reset( ) - to re-use the memory

Score is a Manhattan distance from coord to the destination.

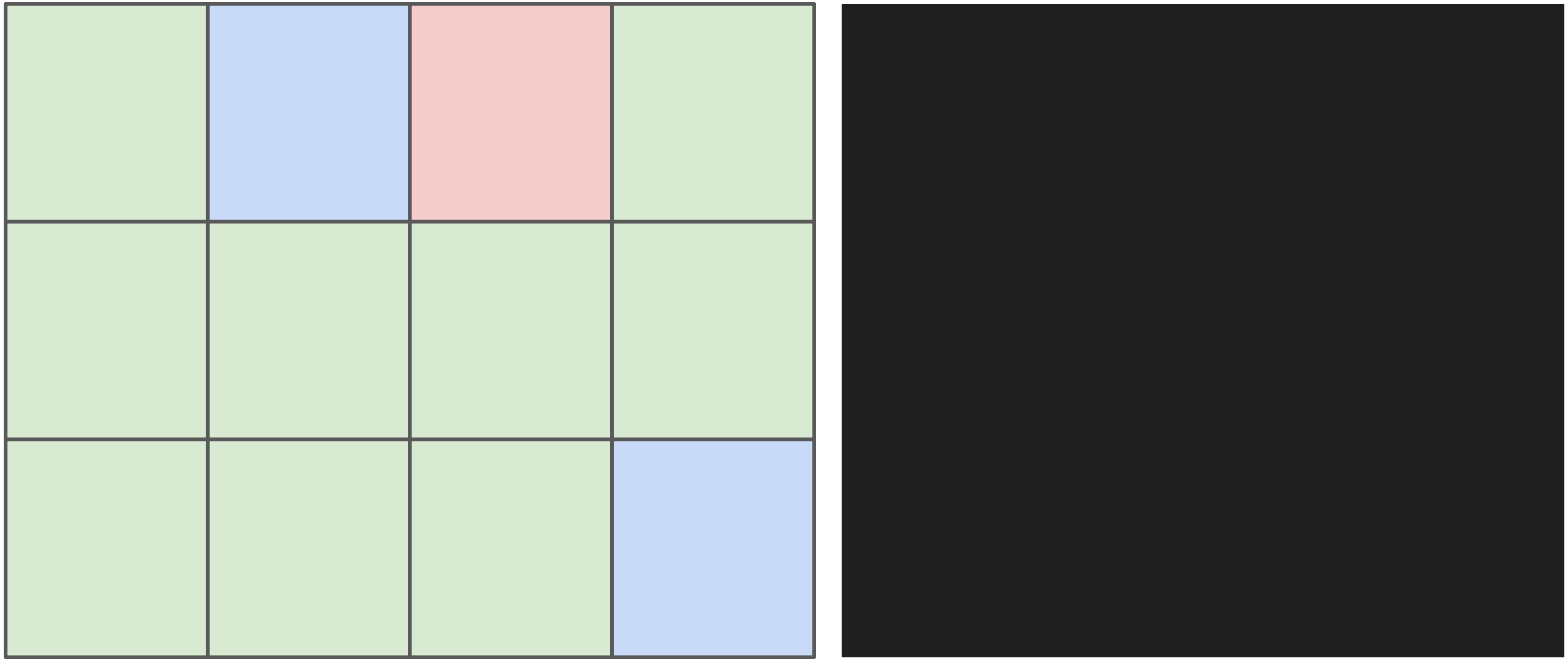
(*) The exact score calculation depends on the algorithm.

How do we use priority queue here?

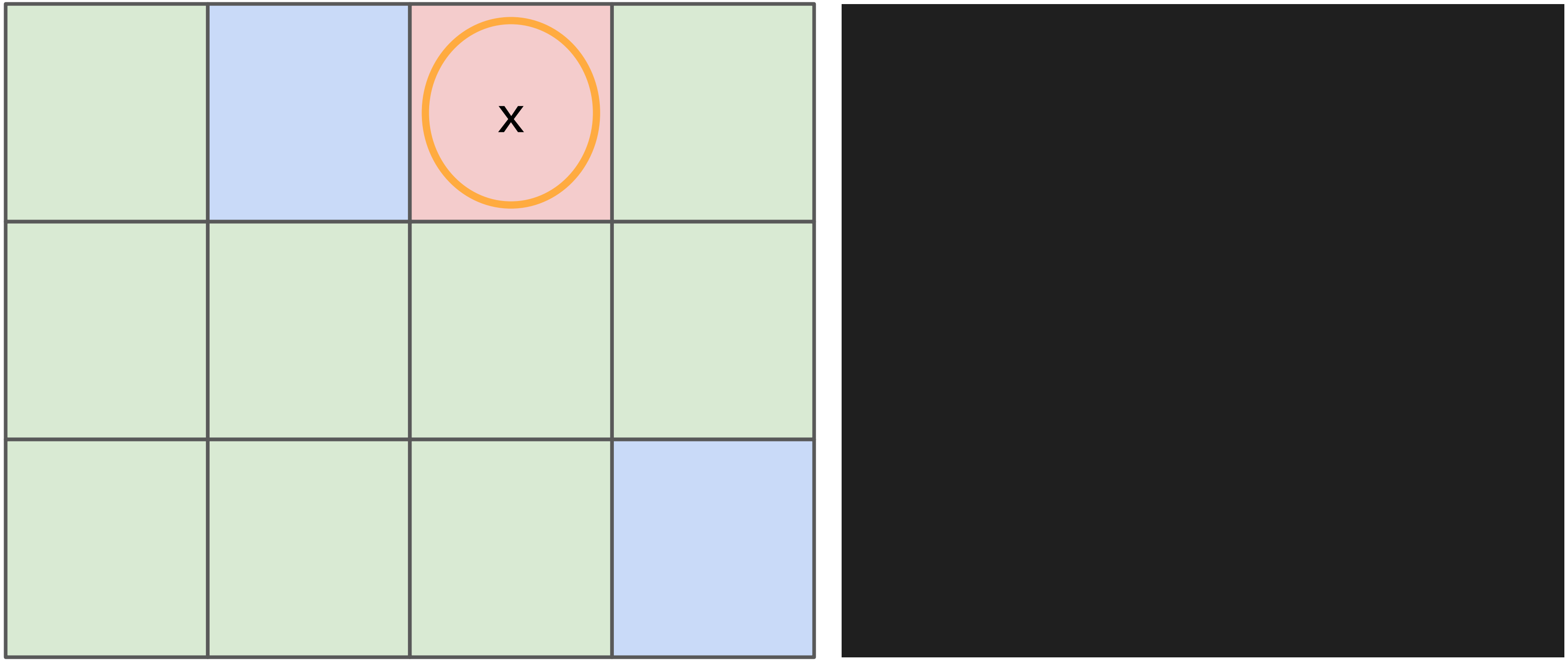
- Push all current possible moves to pqueue
- Investigate all tempting routes first

If some move brings us closer to the finish, we’ll check that route first. This is needed to reduce the average computation steps performed by the algorithm.

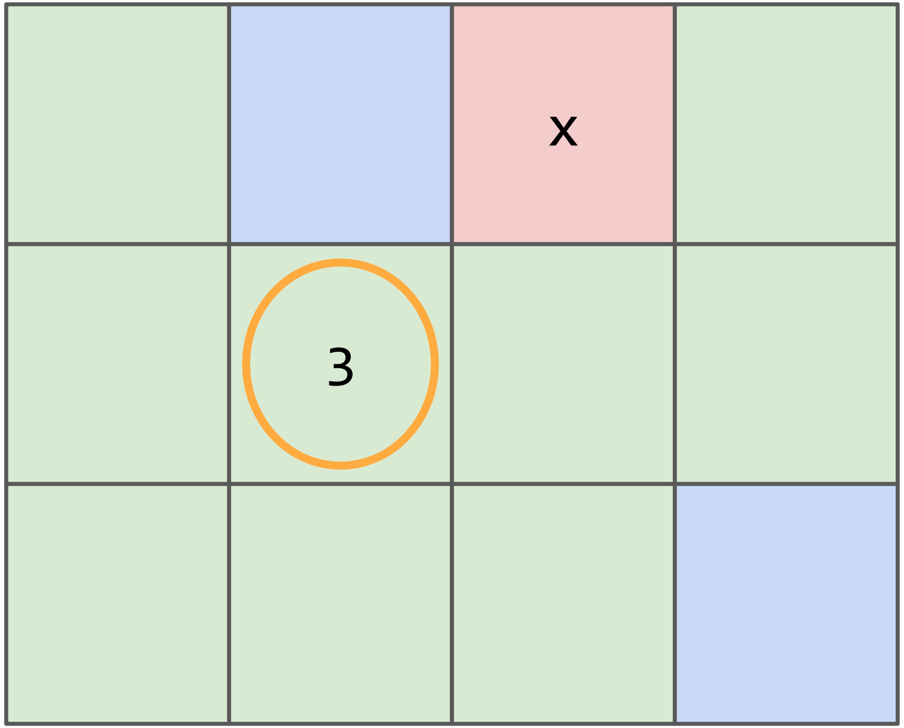
Step 0



Step 1.1

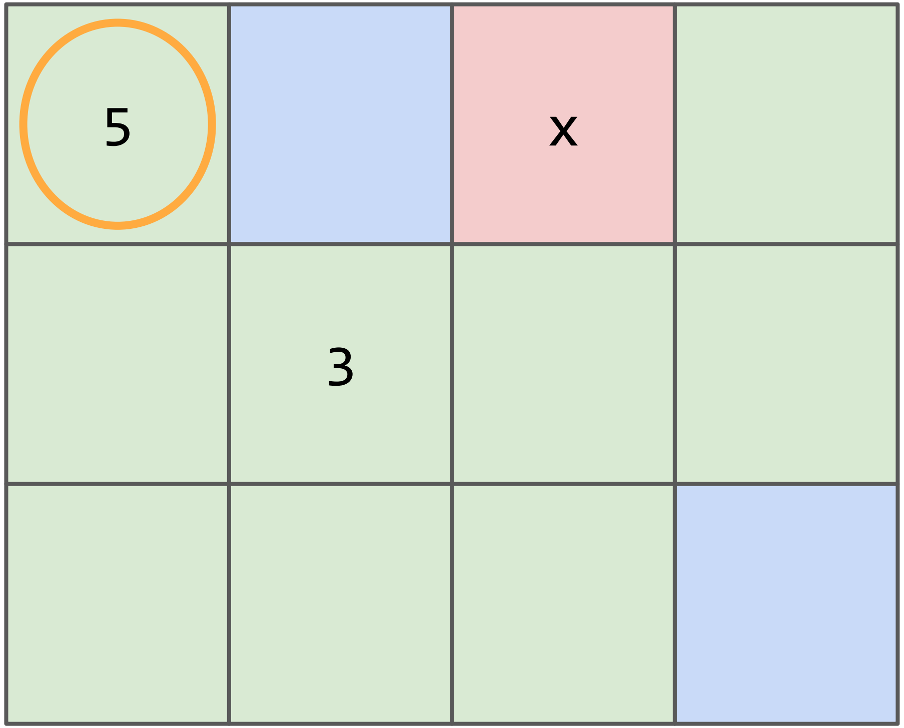


Step 1.2



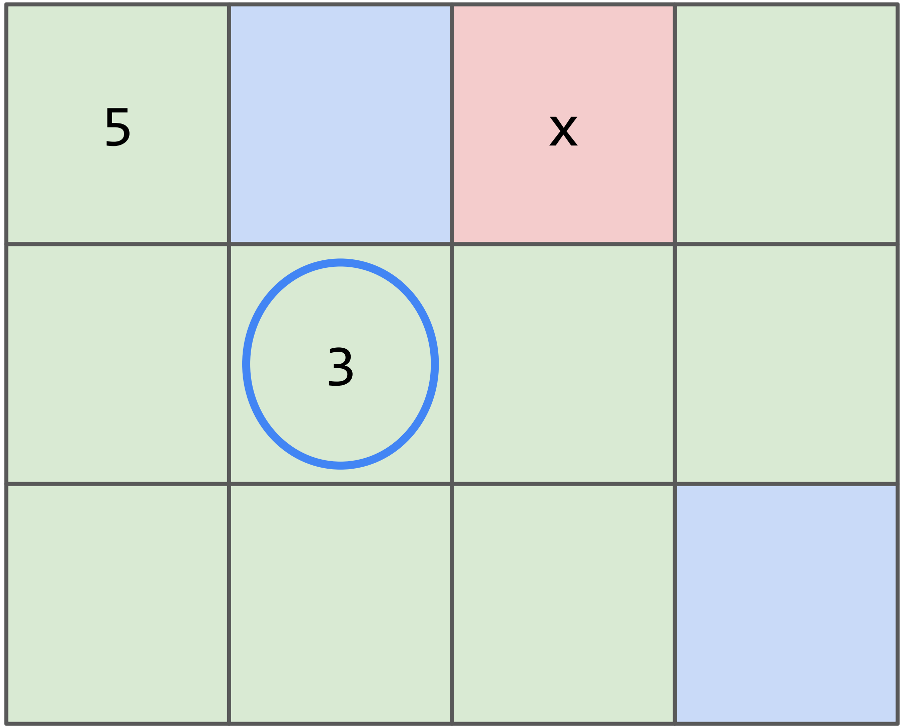

```
3: {1,1}
```

Step 1.3



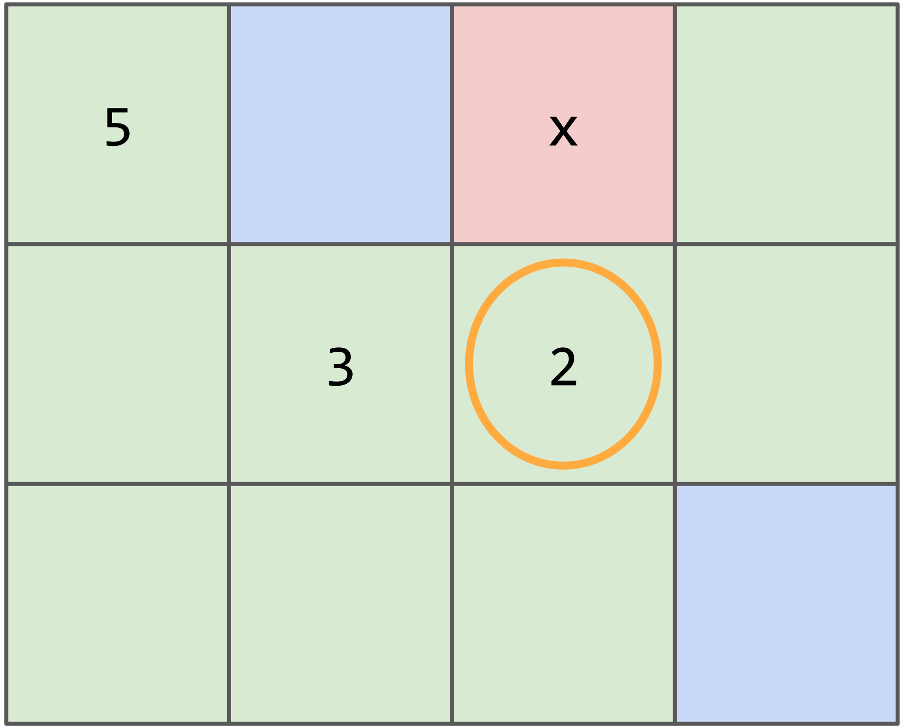

```
3: {1,1}
5: {0,0}
```

Step 2 - pop min



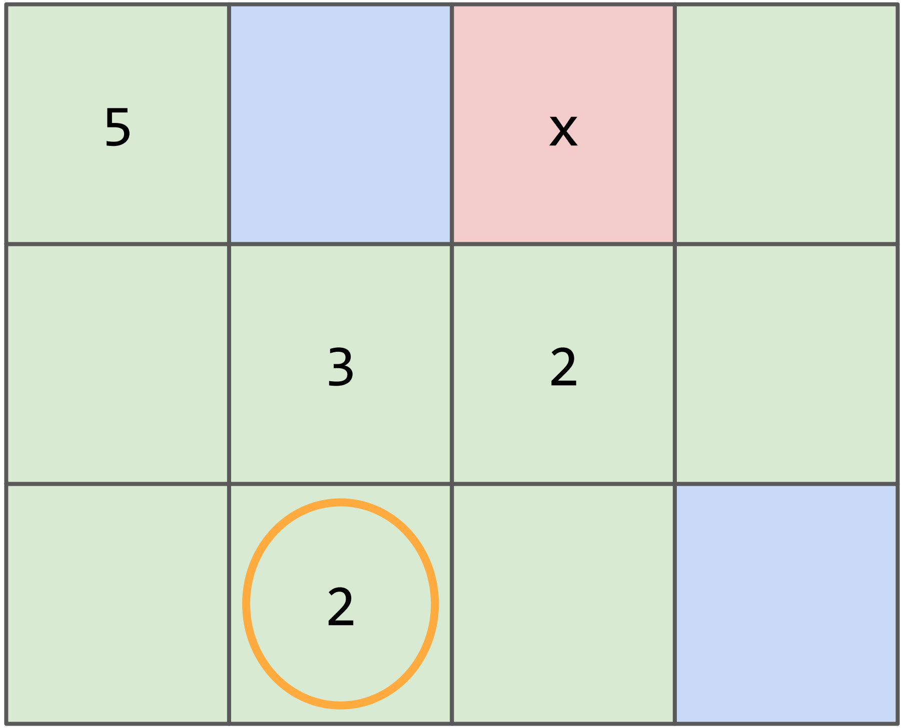

```
3: {1,1}
5: {0,0}
```

Step 2.1



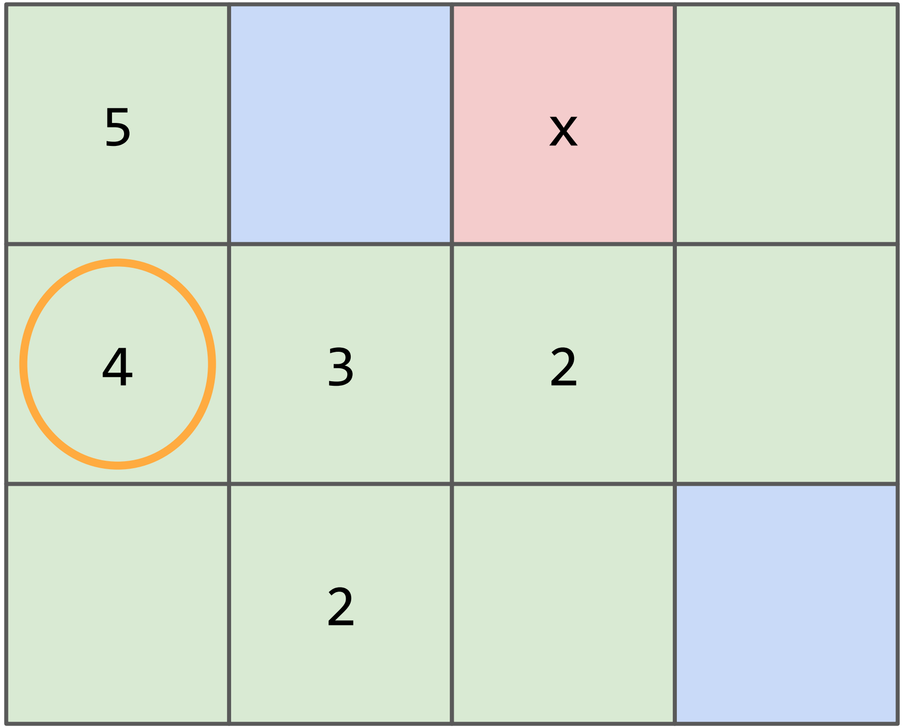

```
2: {2,1}
5: {0,0}
```

Step 2.2



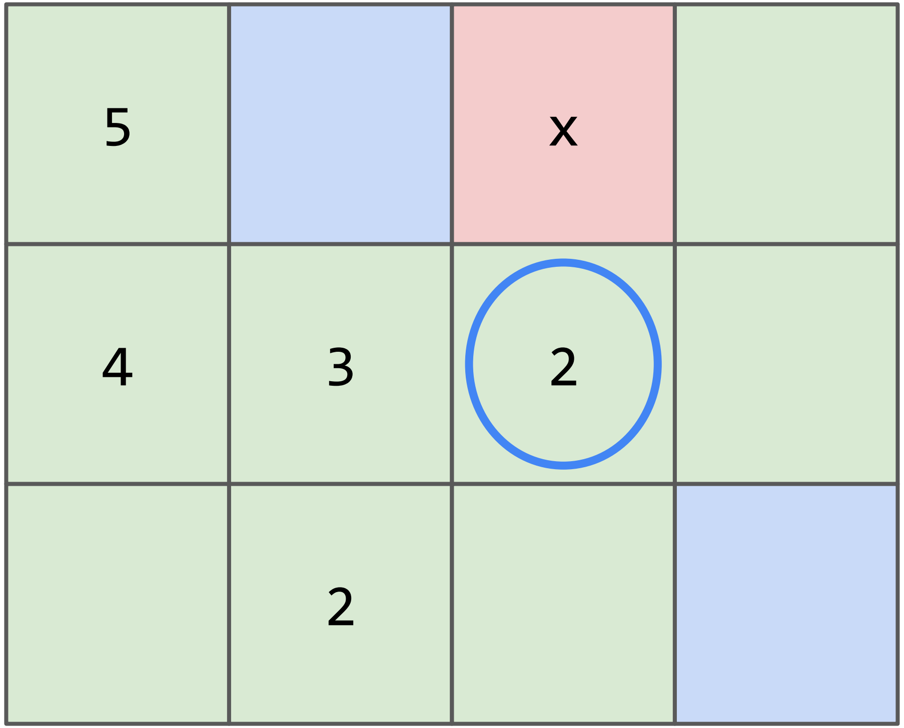

```
2: {2,1} {1,2}
5: {0,0}
```

Step 2.3



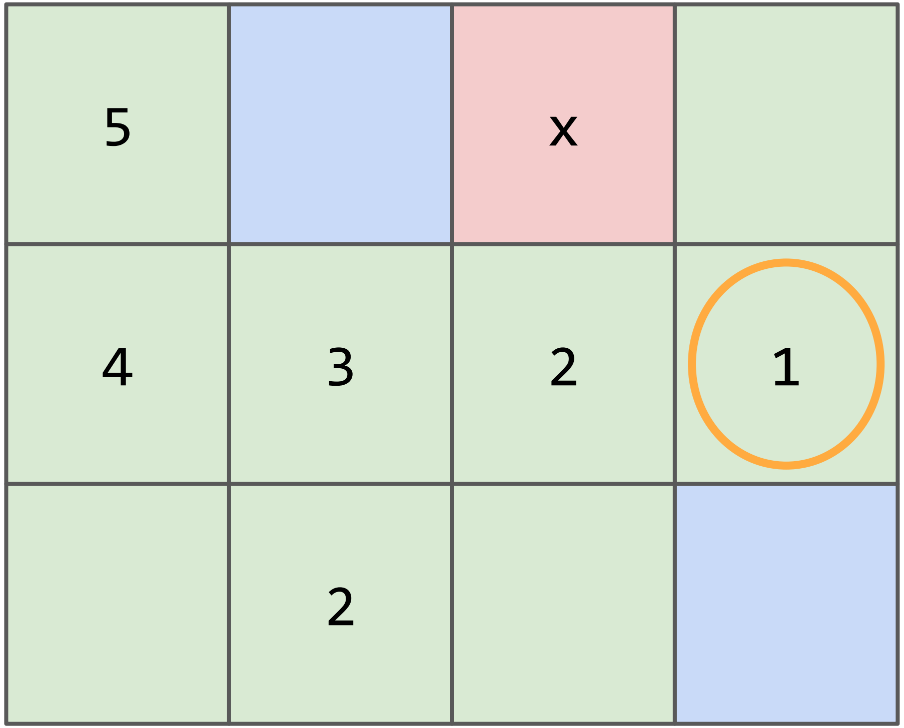

```
2: {2,1}, {1,2}
4: {0,1}
5: {0,0}
```

Step 3 - pop min



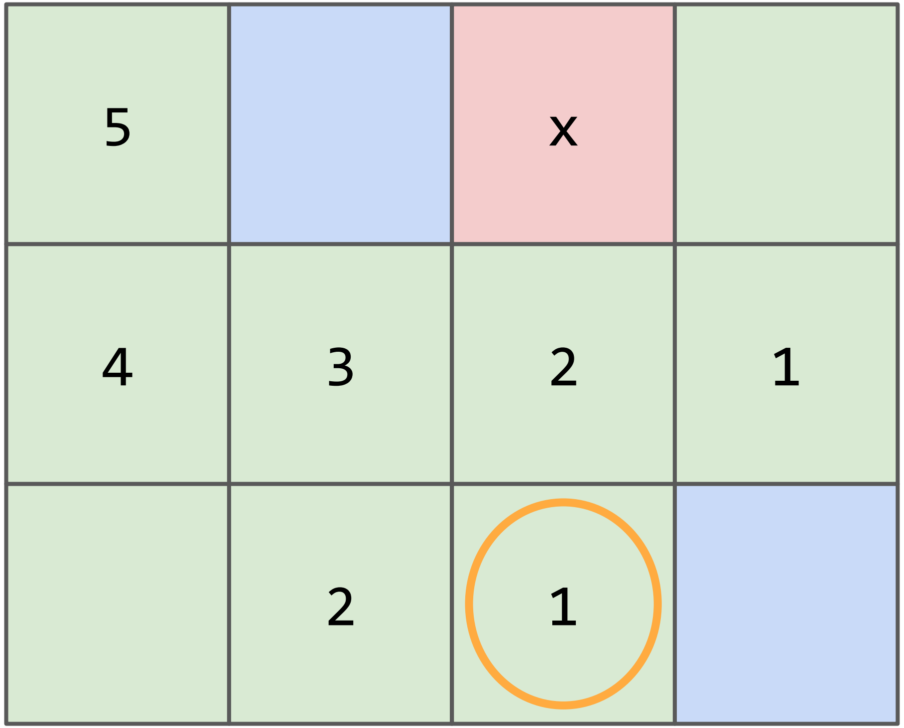

```
2: {2,1}, {1,2}
4: {0,1}
5: {0,0}
```

Step 3.1




```
1: {3,1}
2: {1,2}
4: {0,1}
5: {0,0}
```

Step 3.2




```
1: {3,1} {2,2}
2: {1,2}
4: {0,1}
5: {0,0}
```

How to implement priority queue for this?

Min heap

How to implement priority queue for this?

Bucket queue

Defining PriorityQueue

```
type PriorityQueue struct {
  buckets [64][]Coord
  mask    uint64
}
```

```go
func (q *PriorityQueue[T]) Push(priority int, value T) {
    i := uint(priority) & 0b111111
    q.buckets[i] = append(q.buckets[i], value)
    q.mask |= 1 << i
}
```

This masking removes the bound check

Actions

Queue state

```
mask: 0

buckets: <all empty>
```

Actions

```
Push(1, "foo")
```

Queue state

```
mask: 0b10
buckets: {
  1: {"foo"},
}
```

Actions

```
Push(1, "foo")
Push(4, "bar")
```

Queue state

```
mask: 0b10010
buckets: {
  1: {"foo"},
  4: {"bar"},
}
```

Actions

```
Push(1, "foo")
Push(4, "bar")
Push(1, "baz")
```

Queue state

```
mask: 0b10010
buckets: {
  1: {"foo", "baz"},
  4: {"bar"},
}
```

Unchanged!

```go
func (q *PriorityQueue[T]) Pop() T {
    i := uint(bits.TrailingZeros64(q.mask))
    if i < uint(len(q.buckets)) {
        e := q.buckets[i][len(q.buckets[i])-1]
        q.buckets[i] = q.buckets[i][:len(q.buckets[i])-1]
        if len(q.buckets[i]) == 0 {
            q.mask &^= 1 << i
        }
        return e
    }
    return T{}
}
```

TrailingZeros is basically a BSF+CMOV instructions on x86-64

Actions

```
Push(1, "foo")
Push(4, "bar")
Push(1, "baz")
```

Queue state

```
mask: 0b10010
buckets: {
  1: {"foo", "baz"},
  4: {"bar"},
}
```

Actions

```
Push(1, "foo")
Push(4, "bar")
Push(1, "baz")
Pop() // => "baz"
```

Queue state

```
mask: 0b10010
buckets: {
  1: {"foo"},
  4: {"bar"},
}
```

Unchanged!

Actions

```
Push(1, "foo")
Push(4, "bar")
Push(1, "baz")
Pop() // => "baz"
Pop() // => "foo"
```

Queue state

```
mask: 0b10000
buckets: {
  4: {"bar"},
}
```

Becomes zero!

Actions

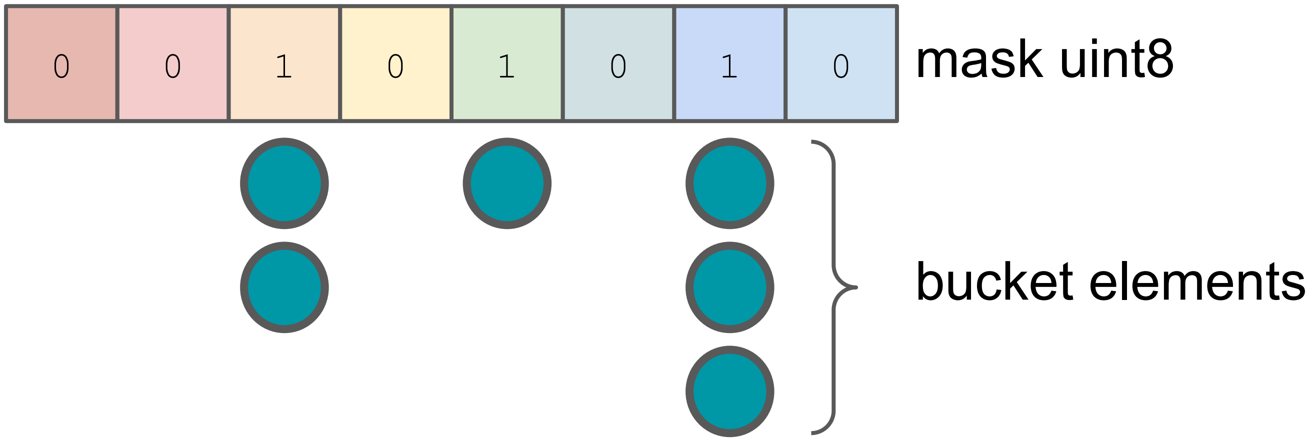
```
Push(1, "foo")
Push(4, "bar")
Push(1, "baz")
Pop() // => "baz"
Pop() // => "foo"
Pop() // => "bar"
```

Queue state

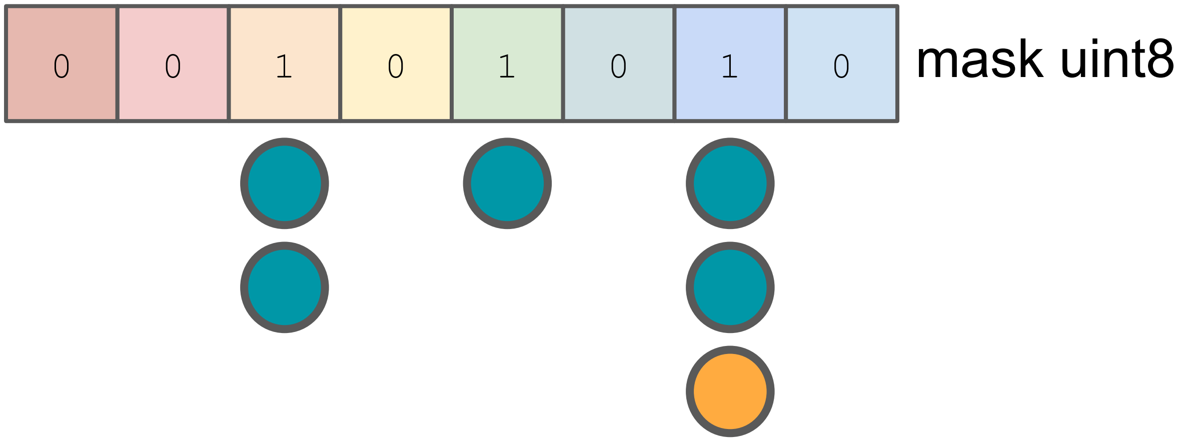
```
mask: 0b00000

buckets: <all empty>
```

All bits are 0

How Pop( ) calculates the bucket in O(1)



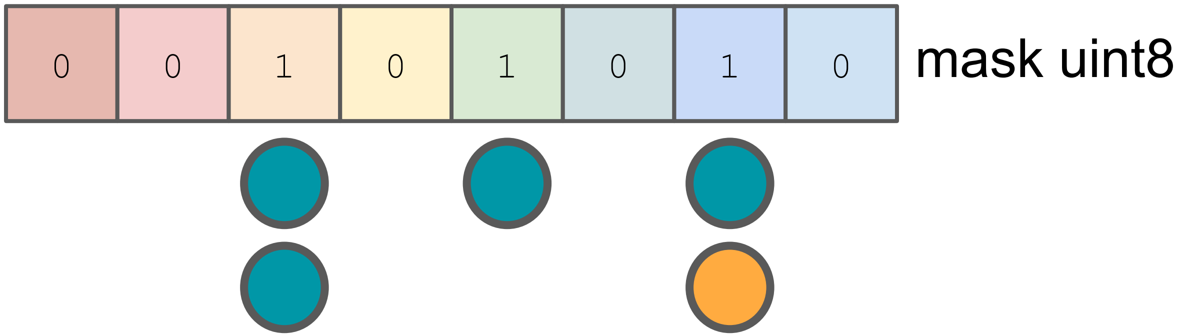
How Pop( ) calculates the bucket in O(1)



Pop#1

`bits.TrailingZeros(0b00101010) => 1`

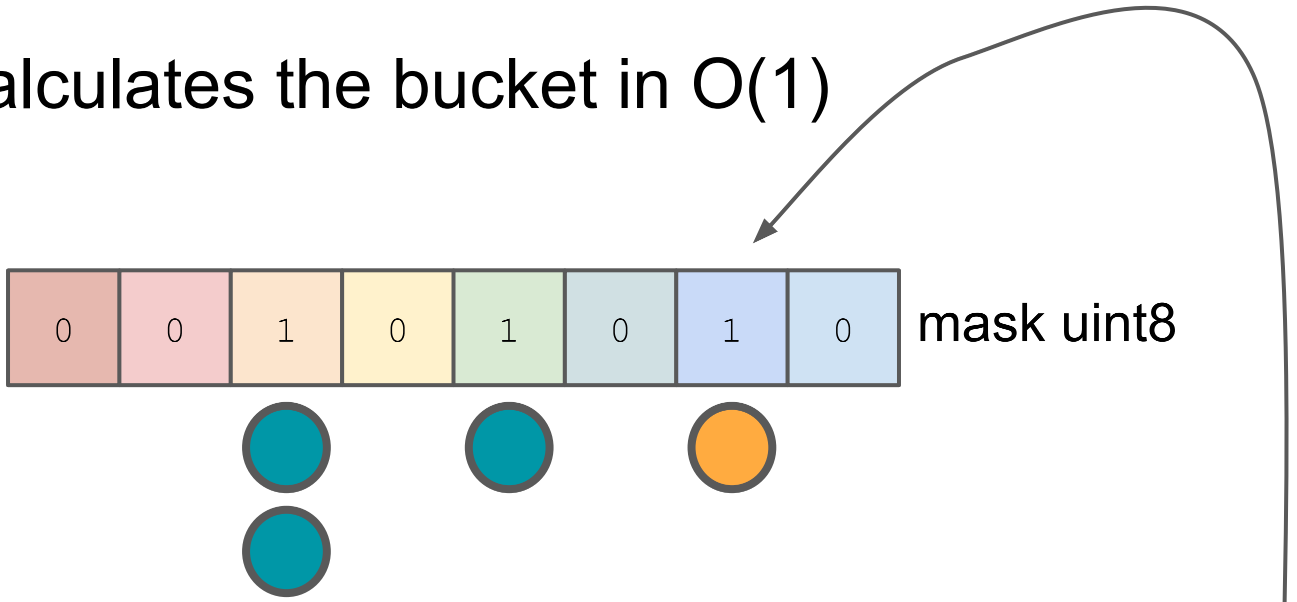
How Pop( ) calculates the bucket in O(1)



Pop#2

`bits.TrailingZeros(0b00101010) => 1`

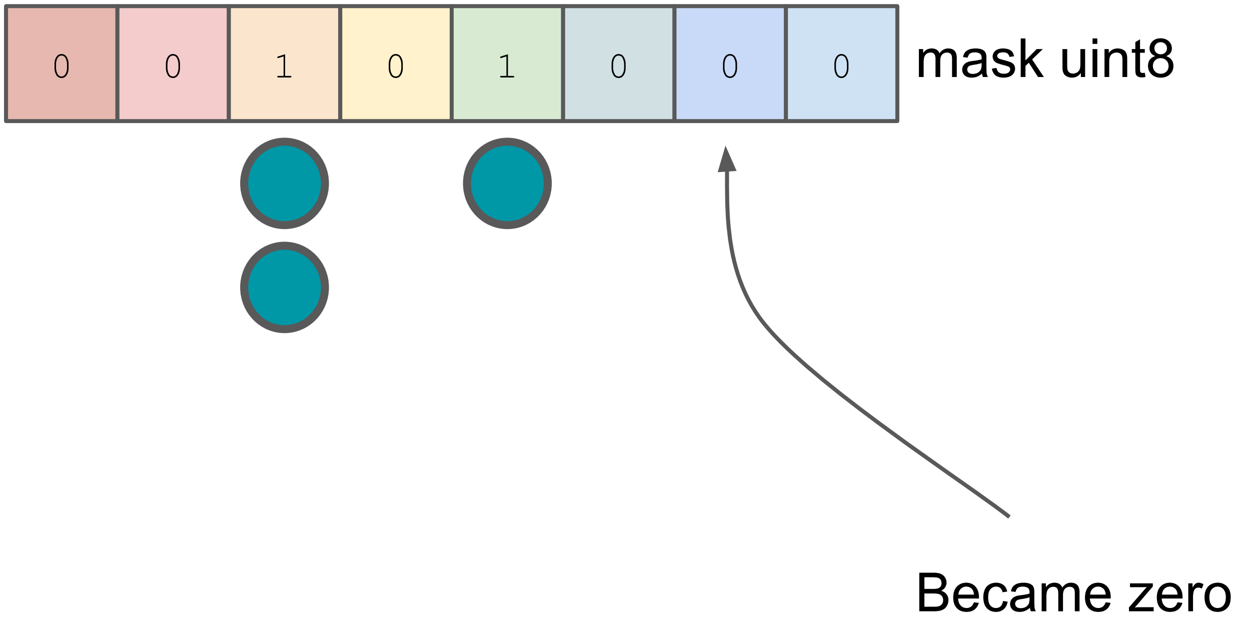
How Pop( ) calculates the bucket in O(1)



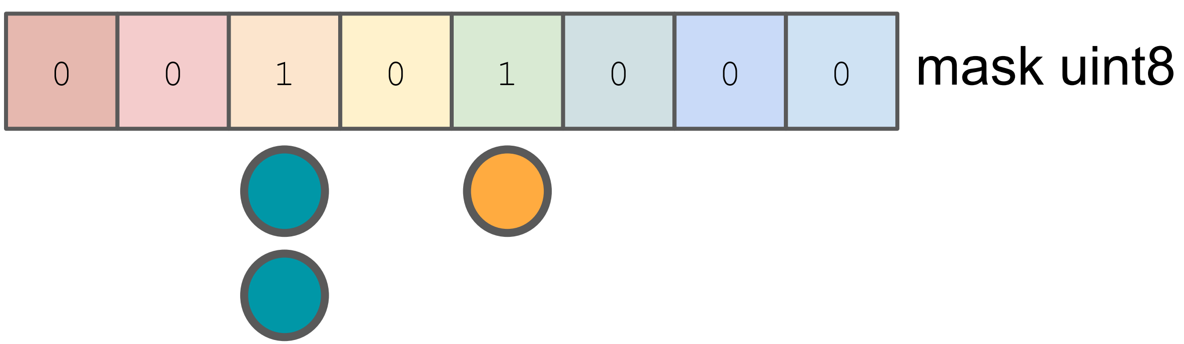
Pop#3

`bits.TrailingZeros(0b00101010) => 1`

How Pop( ) calculates the bucket in O(1)



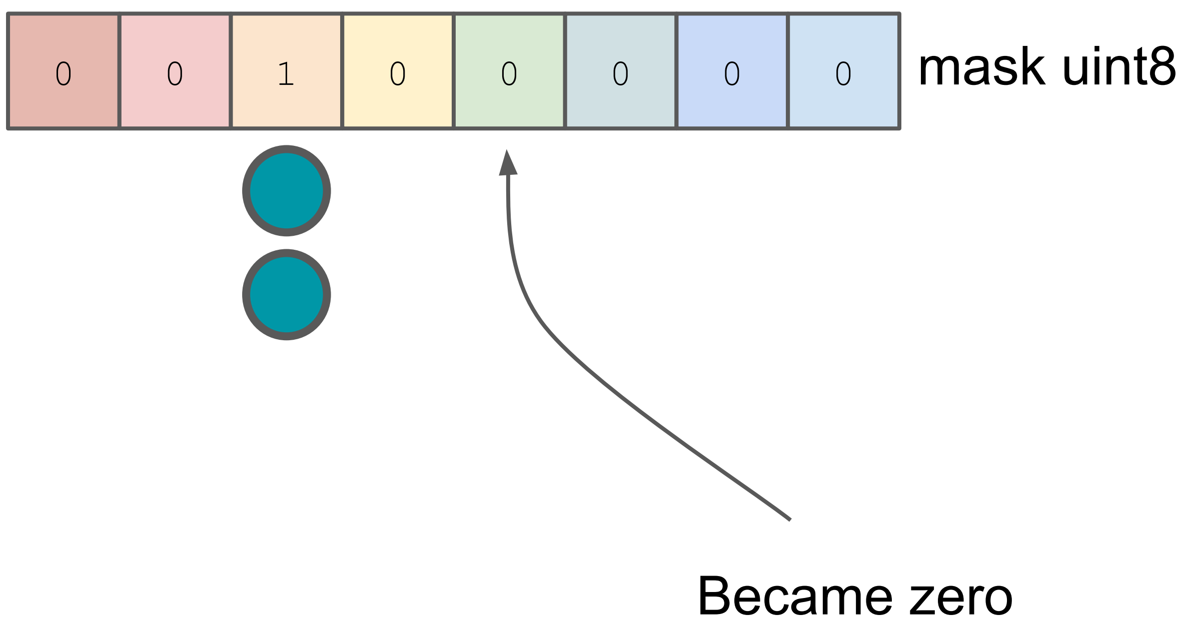
How Pop( ) calculates the bucket in O(1)



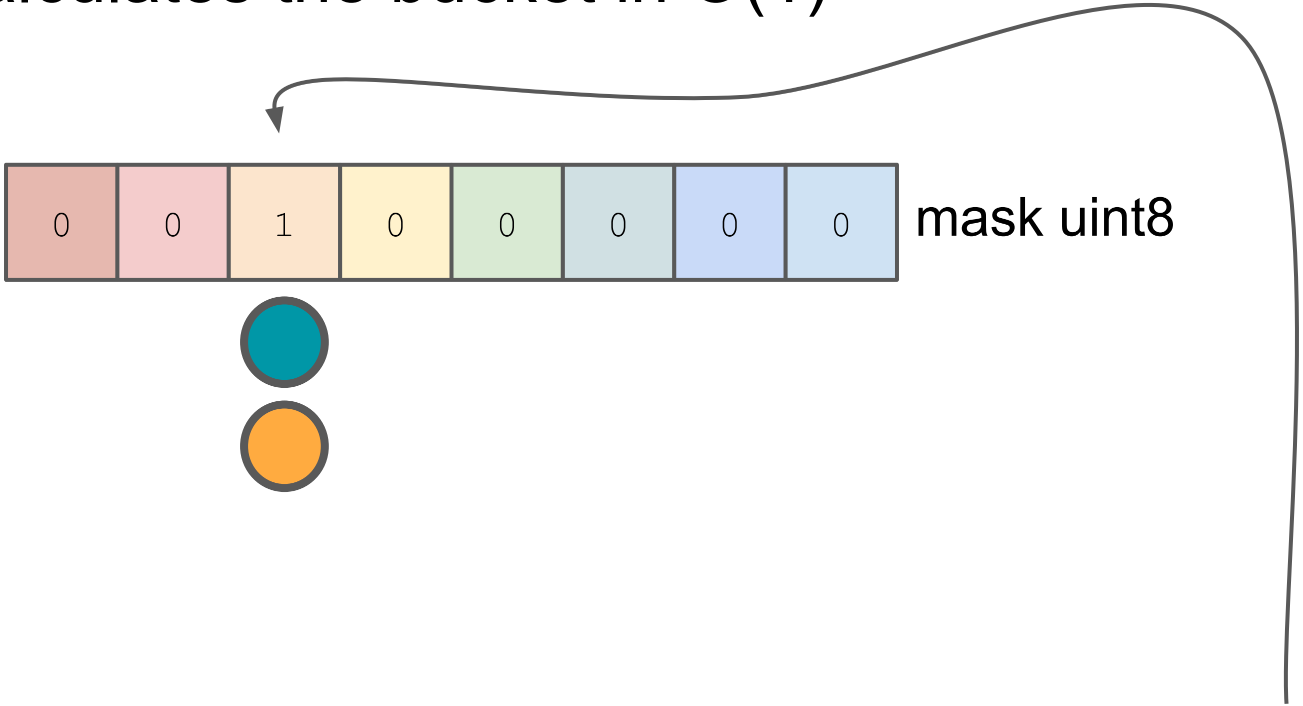
Pop#4

`bits.TrailingZeros(0b00101000) => 3`

How Pop( ) calculates the bucket in O(1)



How Pop( ) calculates the bucket in O(1)



Pop#5

`bits.TrailingZeros(0b00100000) => 5`

```go
func (q *PriorityQueue[T]) Reset() {
  offset := uint(bits.TrailingZeros64(q.mask))
  q.mask >>= offset
  i := offset
  for q.mask != 0 {
     if i < uint(len(q.buckets)) {
        q.buckets[i] = q.buckets[i][:0]
     }
     q.mask >>= 1
     i++
  }
  q.mask = 0
}
```

100% memory re-use

Bucket64 properties

- Push( ) - O(1)
- Pop( ) - O(1)
- Reset( ) - O(1)*

Reset is constant to the number of buckets.

Note that we're using the mask to skip reslicing batches of buckets, so it's quite fast.


Friendship ended with
container/heap
Now
bits
is my
best friend

Greedy BFS performance-critical parts

- Matrix to store cell information (the grid)
- Result path representation
- Priority queue for “frontier”
- Map to store the visited cells

How to implement “visited set”?

map[coord]data

How to implement "visited set"?

[ ]data

How to implement “visited set”?

[ ]data

2D (or 1D) array properties

- Get(coord) - O(1)
- Set(coord, value) - O(1)
- Reset( ) - O(n)

Re-using a big array is an expensive memset(0).

It makes them impractical for us.

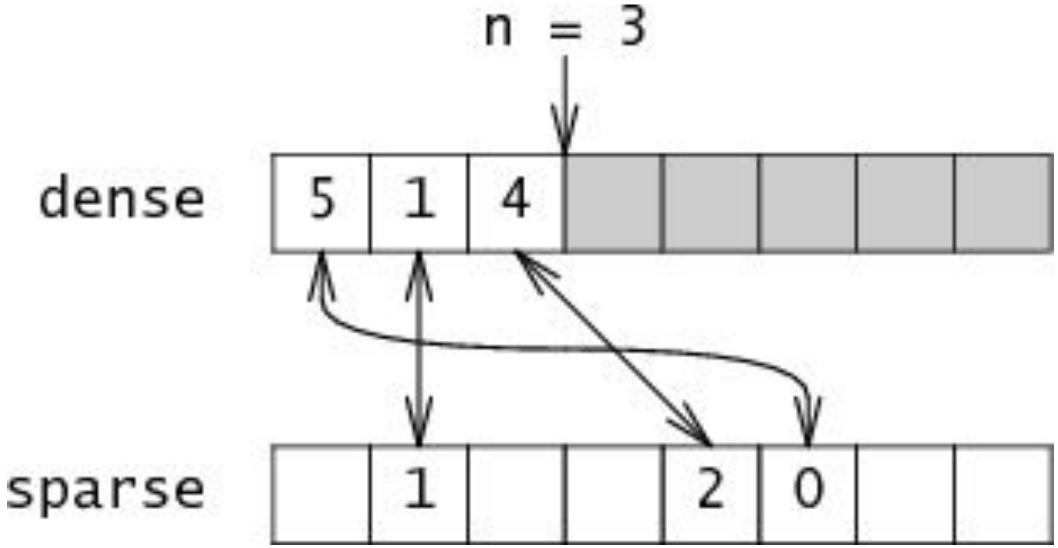
CoordMap benchmarks, size=96*96 (ns/op)

CONTAINER	SET	GET	RESET
array	10	4	7200

Reset( ) for an array can be quite slow

Sparse map to the rescue!

https://research.swtch.com/sparse



Sparse map properties

- Get(coord) - O(1)
- Set(coord, value) - O(1)
- Reset( ) - O(1)

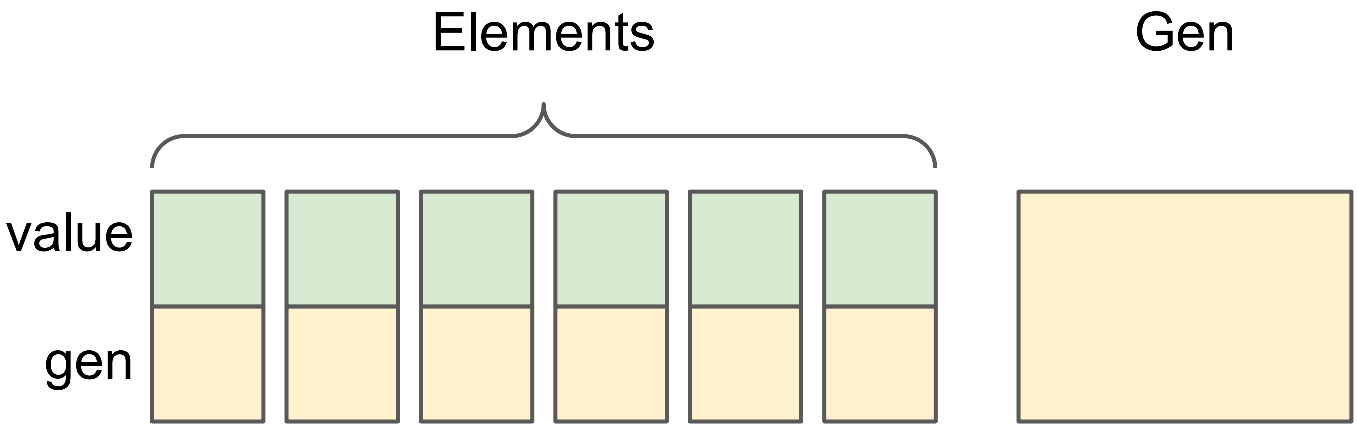
CoordMap benchmarks, size=96*96 (ns/op)

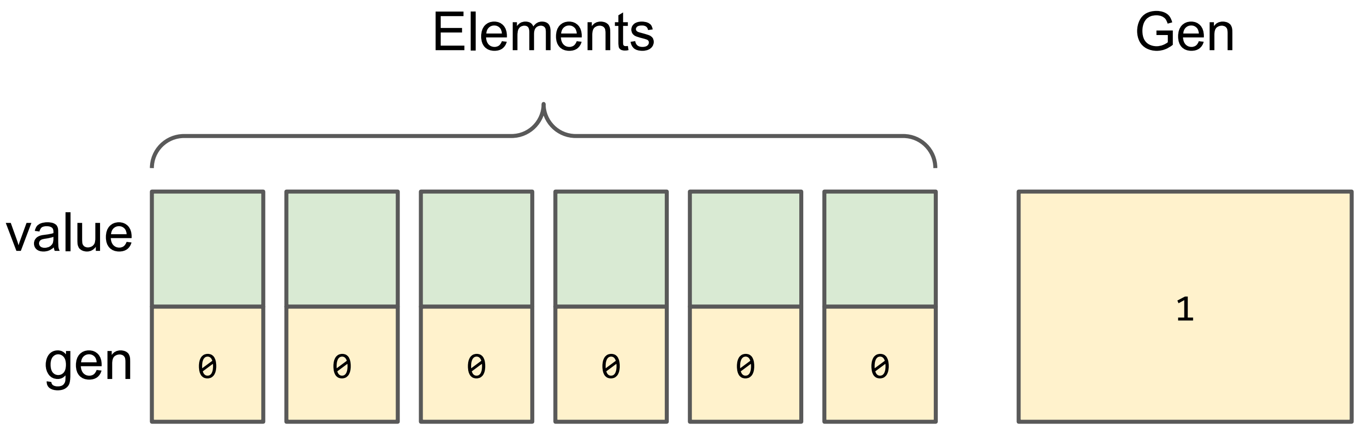
CONTAINER	SET	GET	RESET
array	10	4	7200
sparse-dense	(+25) 35	(+13) 17	1

sparse-dense Set( ) & Get( ) have some extra overhead

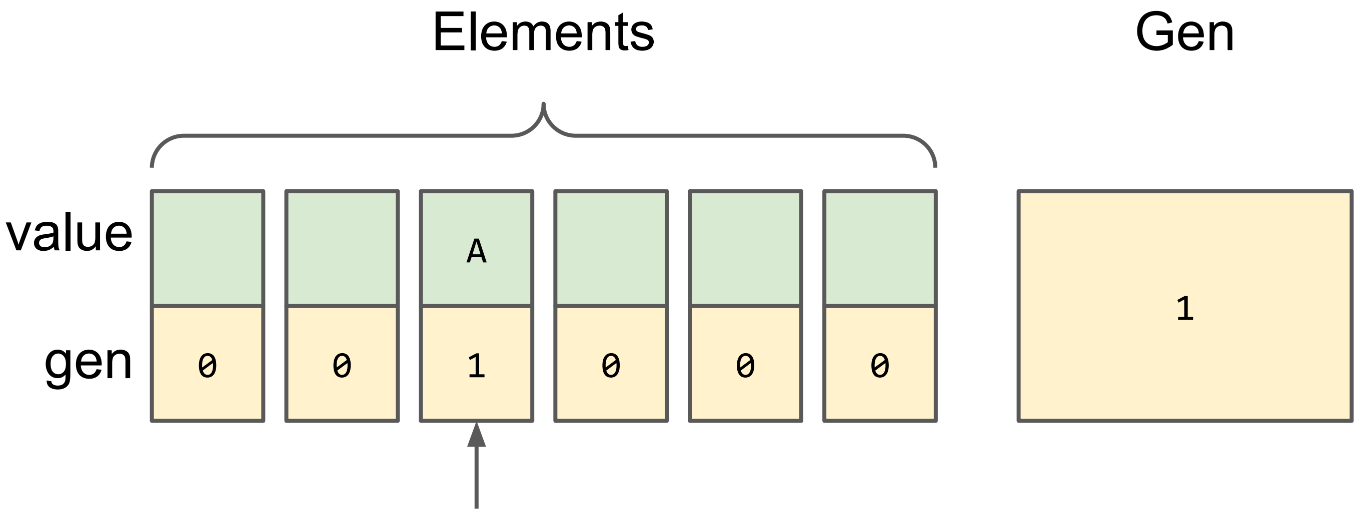
Generations-based map to the rescue!

https://quasilyte.dev/blog/post/gen-map/

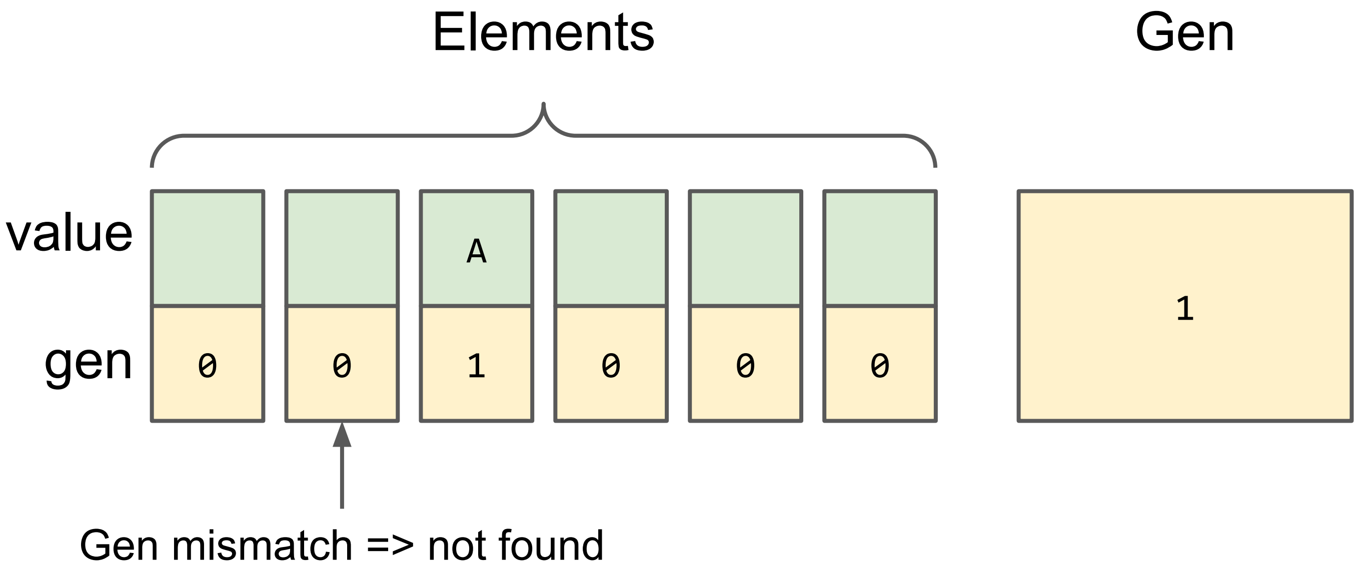

Elements
Gen
value
gen



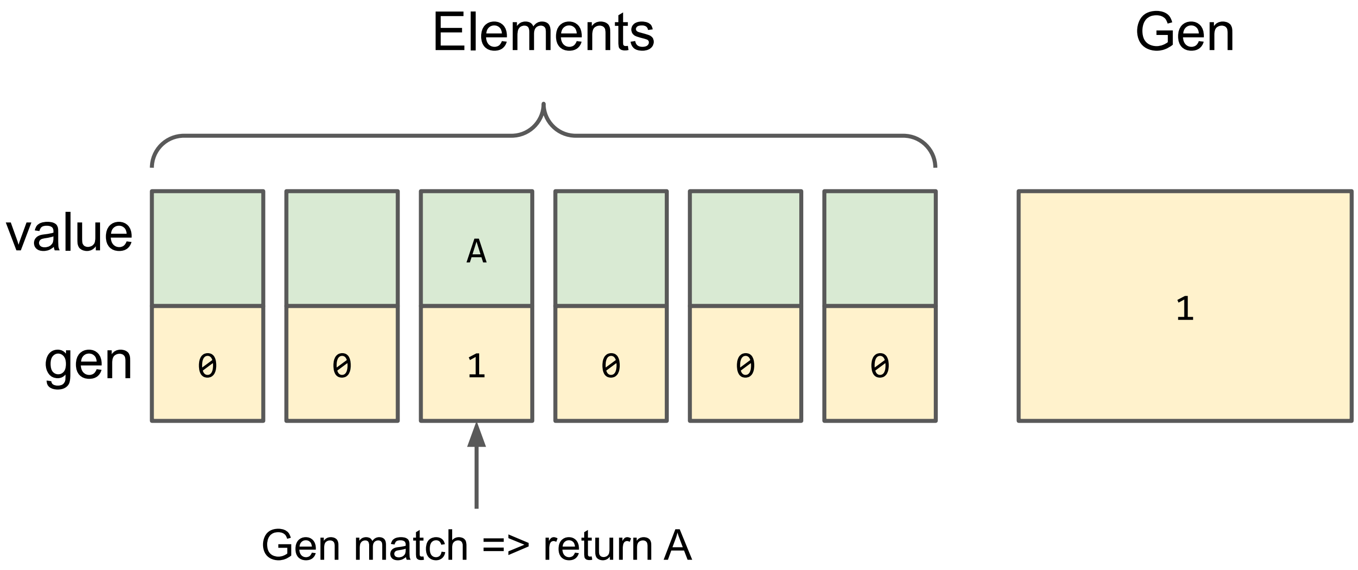
The zero-value container gen is 1, element gens are 0



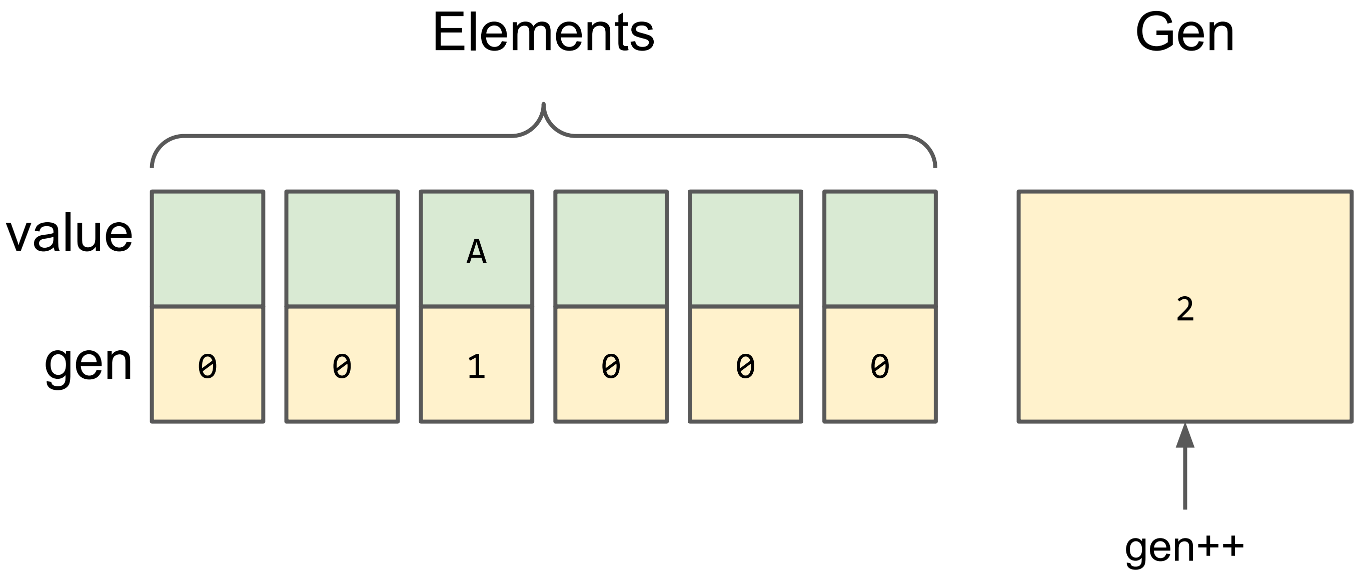
Set( ) assigns the value & updates the gen counter



Get( ) compares the elem and container gen values



Get( ) compares the elem and container gen values



No memory reset is needed, Get( ) will work correctly right away

Generation map properties

- Get(coord) - O(1)
- Set(coord, value) - O(1)
- Reset( ) - O(1)

minimal overhead

CoordMap benchmarks, size=96*96 (ns/op)

CONTAINER	SET		GET		RESET
array		10		4	7200
sparse-dense	(+25)	35	(+13)	17	1
gen-map	(+4)	14	(+6)	10	1

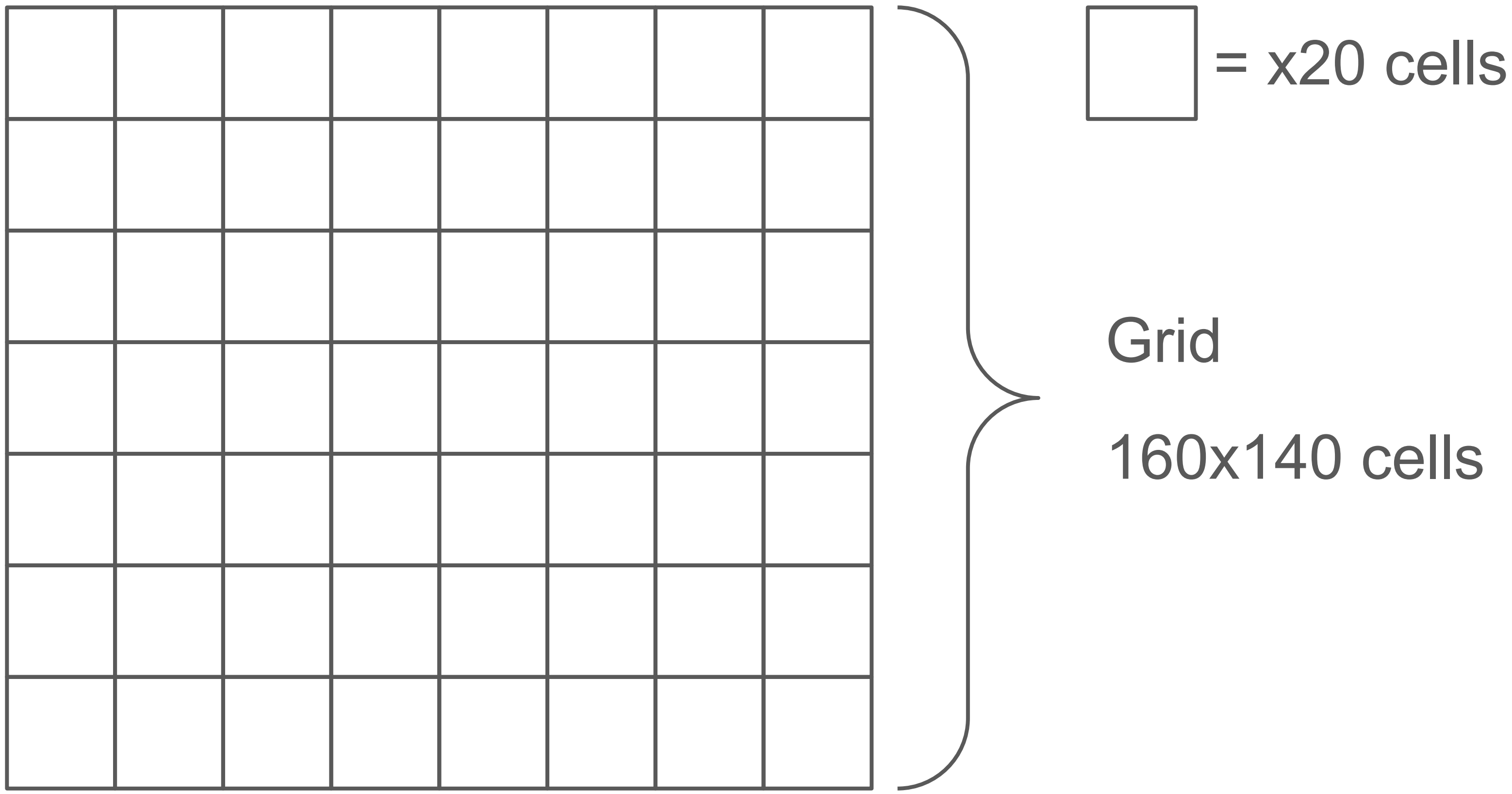
Generations-based map is ~2 times faster than sparse-dense map

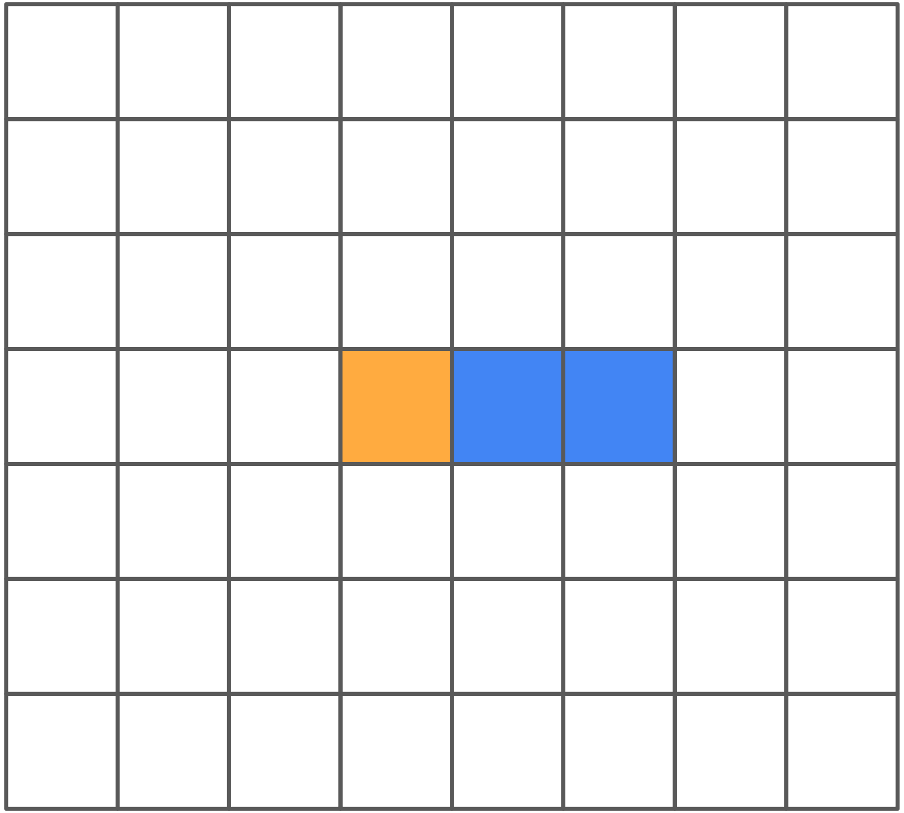
CoordMap size estimations

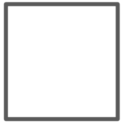
- Grid stores 1 cell for 2 bits
- CoordMap needs ~8 bytes per cell

We’re paying x32 times more memory for the CoordMap!

Can we still have an ~unlimited Grid size after that?

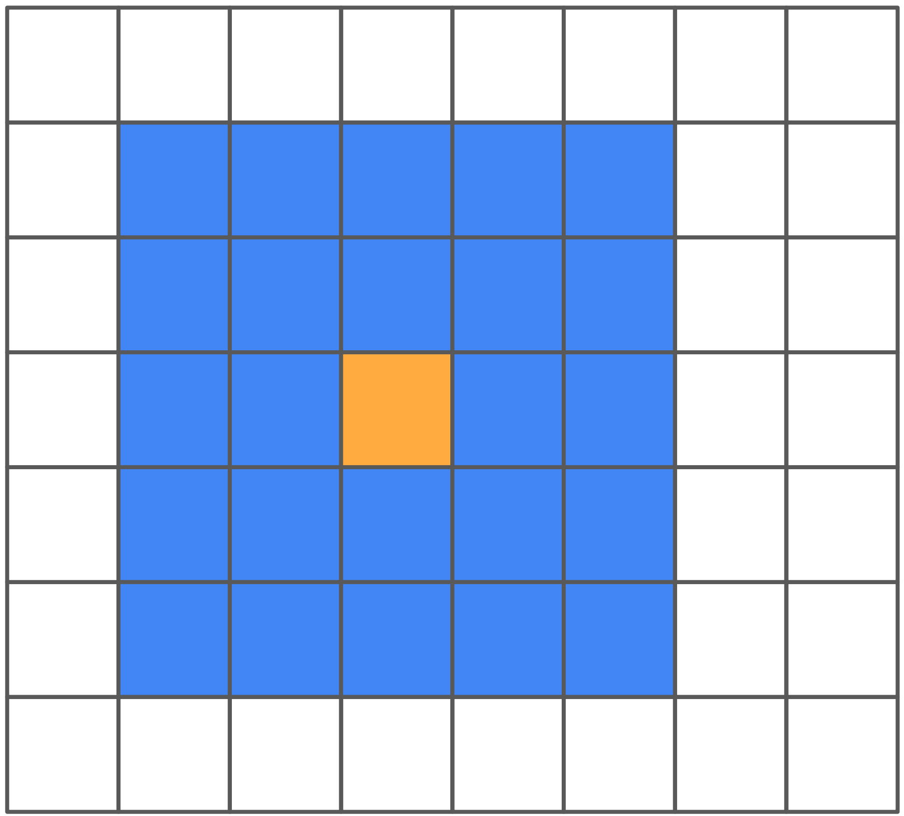

= x20 cells
Grid
160x140 cells




= x20 cells

Max path length

(showed as 60 cells)



= x20 cells

Max search zone

(it’s less than Grid)

CoordMap max size

- Max search area is N=((56*2)^2)+2 => 12546
- The max size is 12546 * 8 => 100368 bytes (0,1 MB)

After that, you can increase the Grid size for as much as you want; the CoordMap won’t get bigger.

Defining coord map element

```go
type mapElem struct {
  value uint8  // Direction stored as uint8
  gen   uint32 // Generation counter
}


// Sizeof(mapElem) => 8 (due to padding)
```

Defining coord map element

```go
type mapElem struct {
  value uint8  // Direction stored as uint8
  gen   uint16 // Generation counter
}

// Sizeof(mapElem) => 4 (1 byte wasted)
```

Handling the overflow

```go
type mapElem struct {
  value uint8  // Direction stored as uint8
  gen   uint8  // Generation counter
}


// Sizeof(mapElem) => 2 (no bytes wasted)
```

Defining coord map element

```go
func (m *CoordMap) Reset() {
    // Use a constant to match the gen type.
    if m.gen == math.MaxUint16 {
        m.clear() // A real memclr; happens very rarely
    } else {
        m.gen++
    }
}
```

Defining coord map element

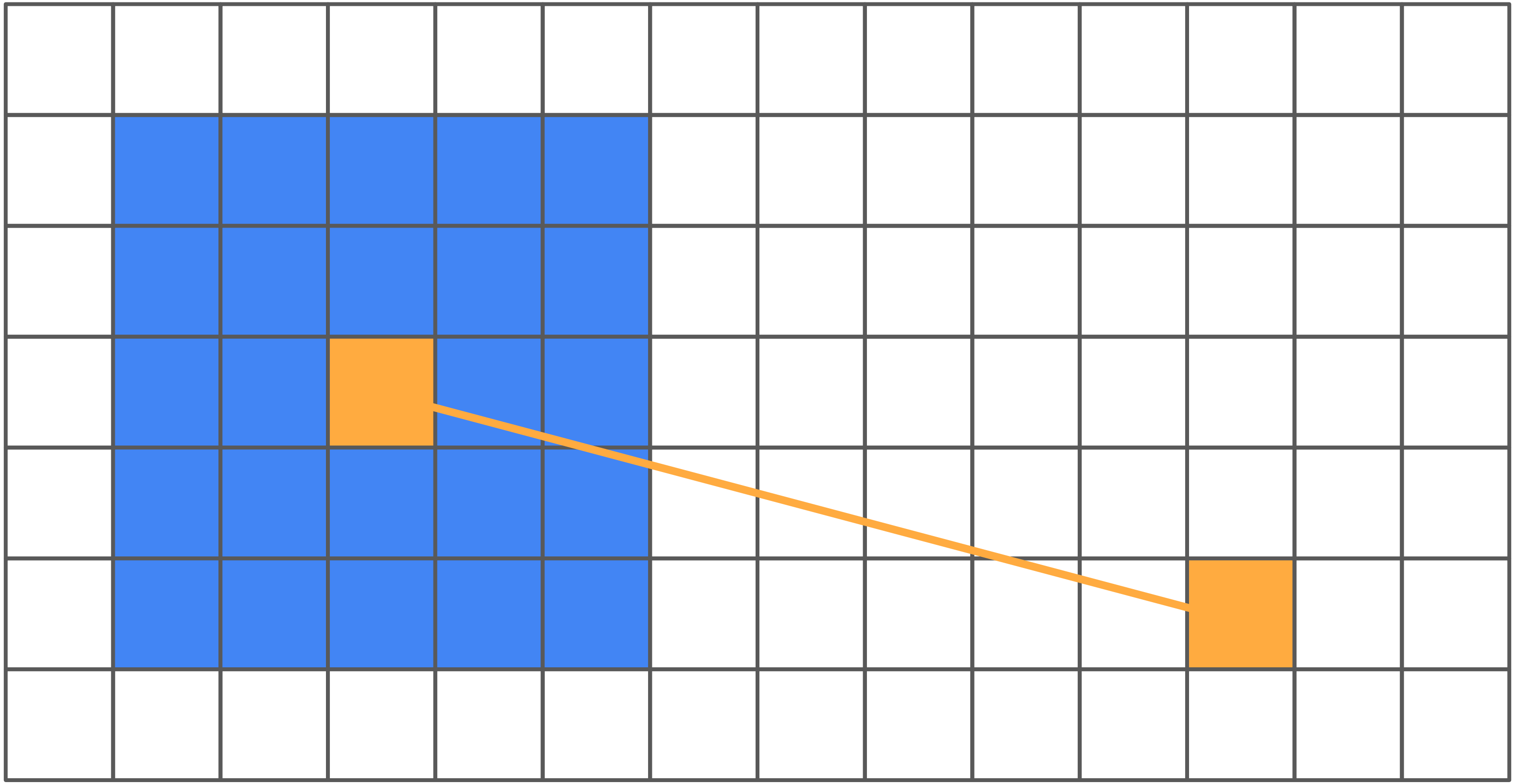
```go
func (m *CoordMap) clear() {
    m.gen = 1
    clear(m.elems) // Sets elem.gen to 0
}
```

Gen counter size

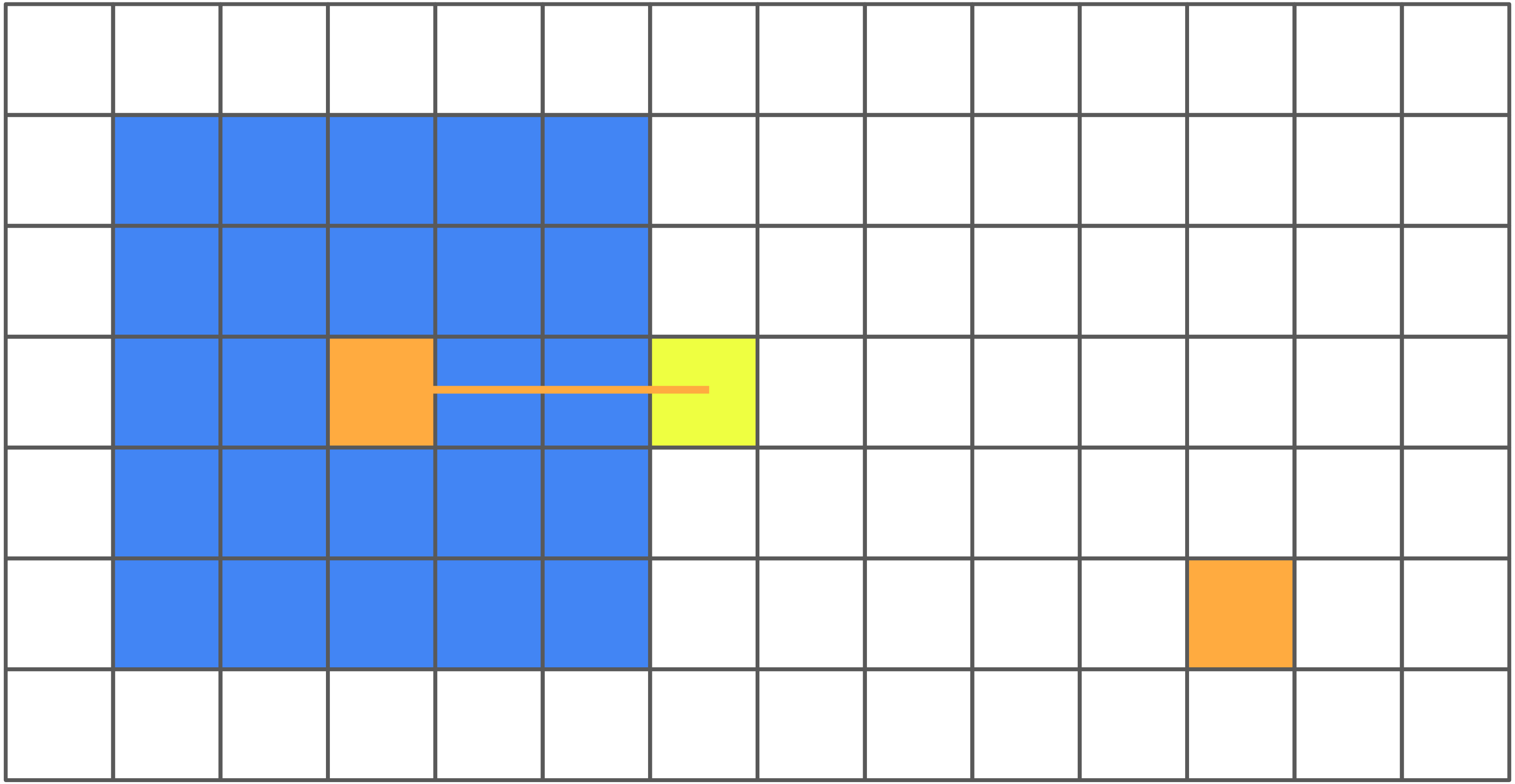
Smaller counter => less memory overhead per element, but real memory clears will happen more often.

Counters like uint32-uint64 make real memory clears almost impossible, but they will consume more memory per element.

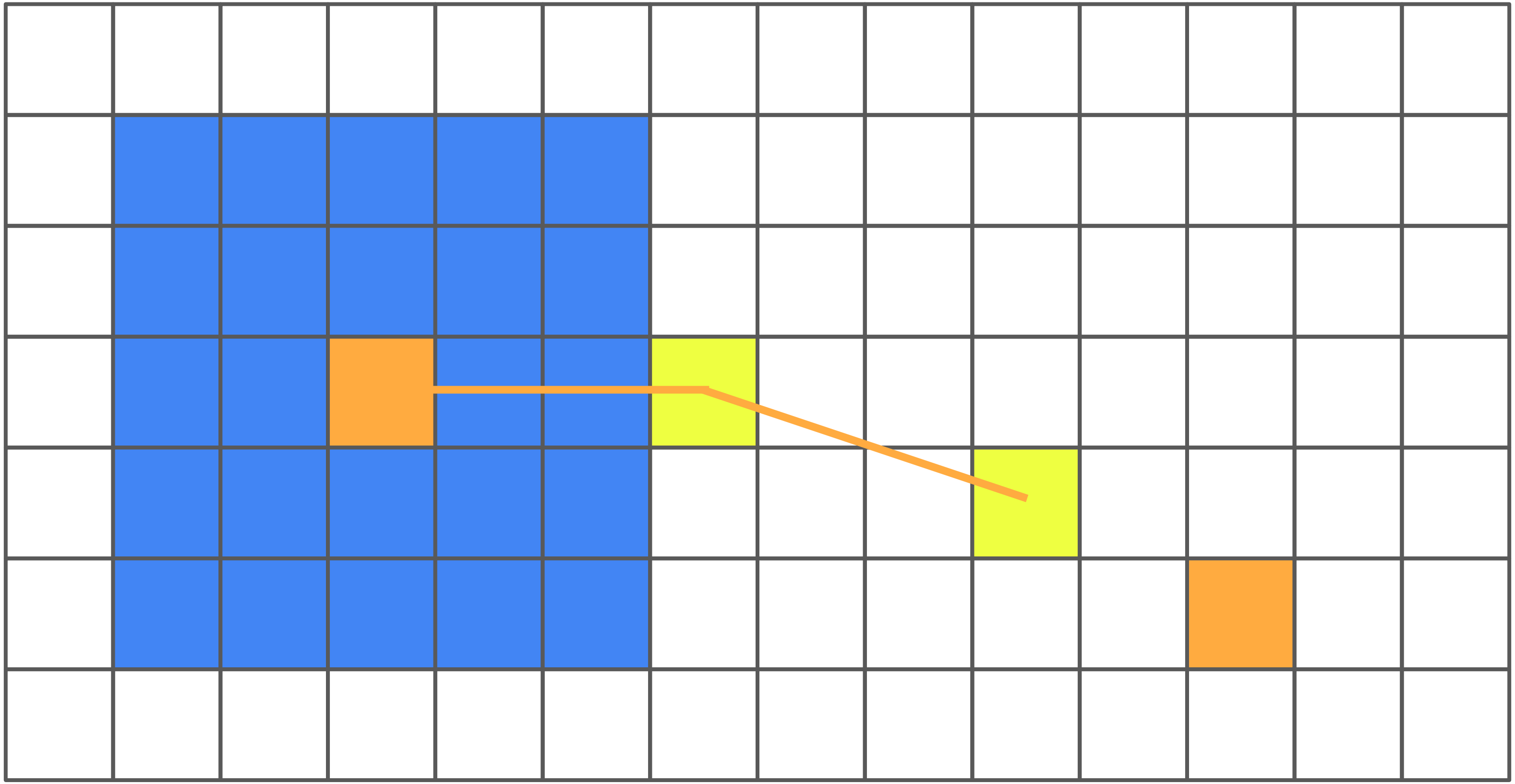
Overcoming the limitations



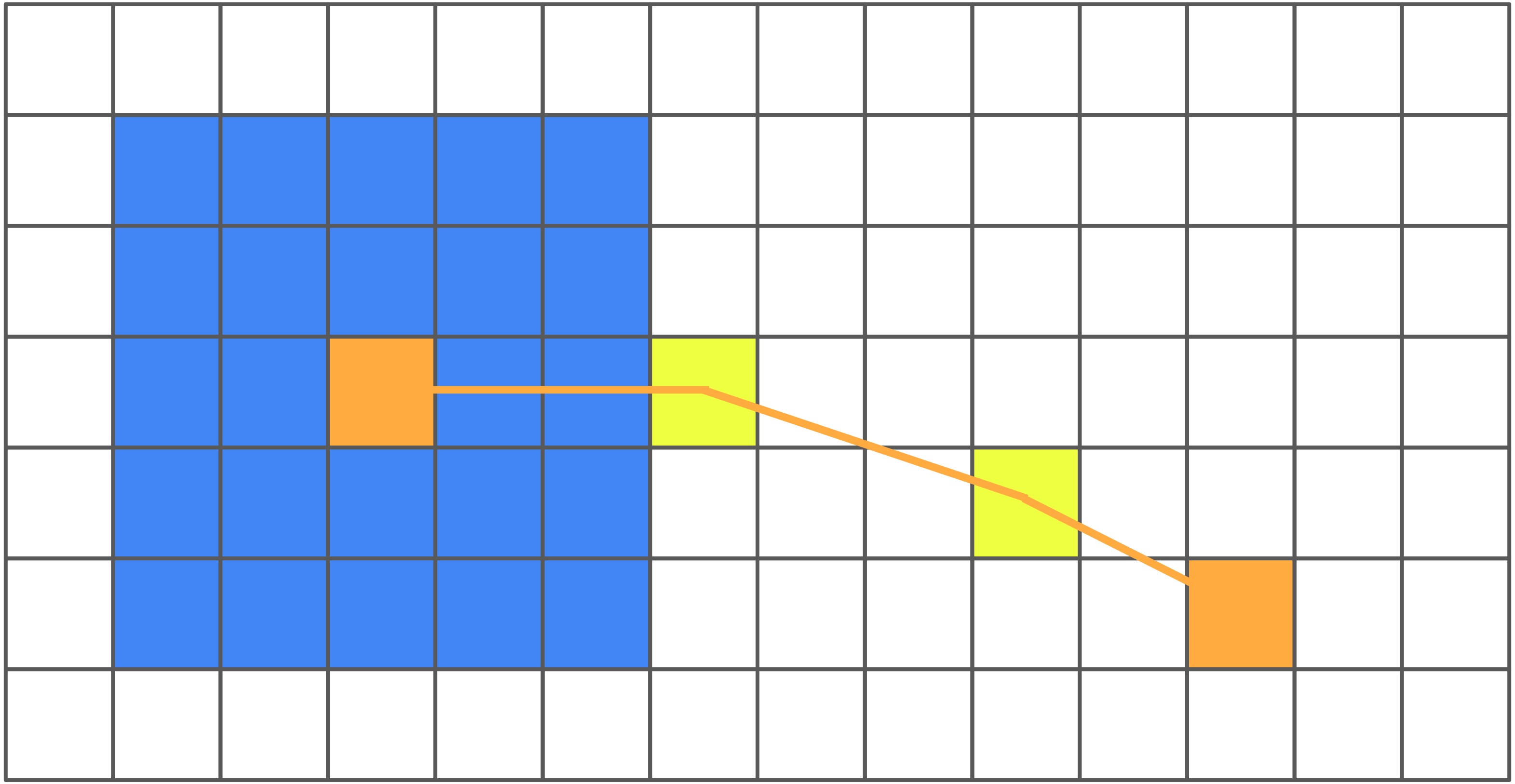
What if you need a path longer than 54 cells?



Build multiple paths!



Build multiple paths!



Build multiple paths!

More than 4 tile types in a game?

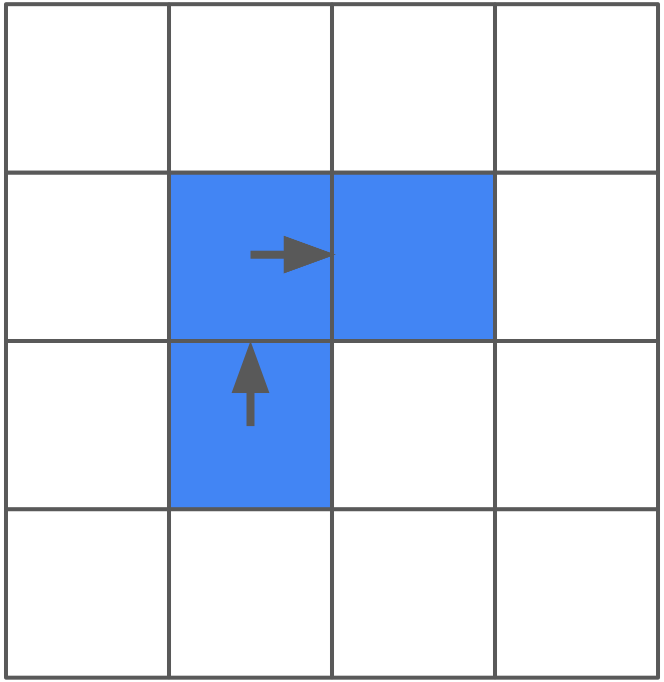
Use per-biome layer sets!

Jungle biome

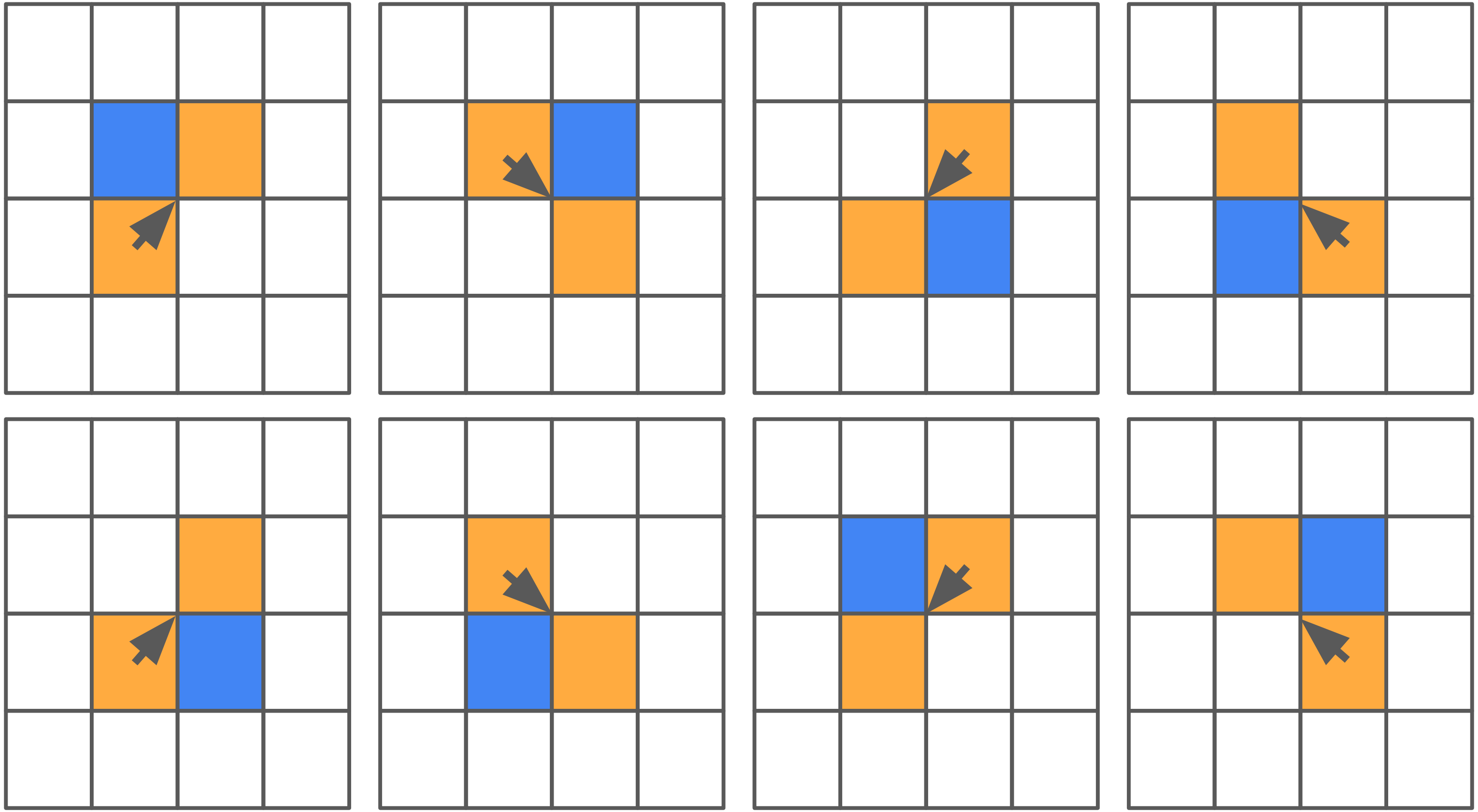
0	Plains
1	Forest
2	Water
3	Mountains

Inferno biome

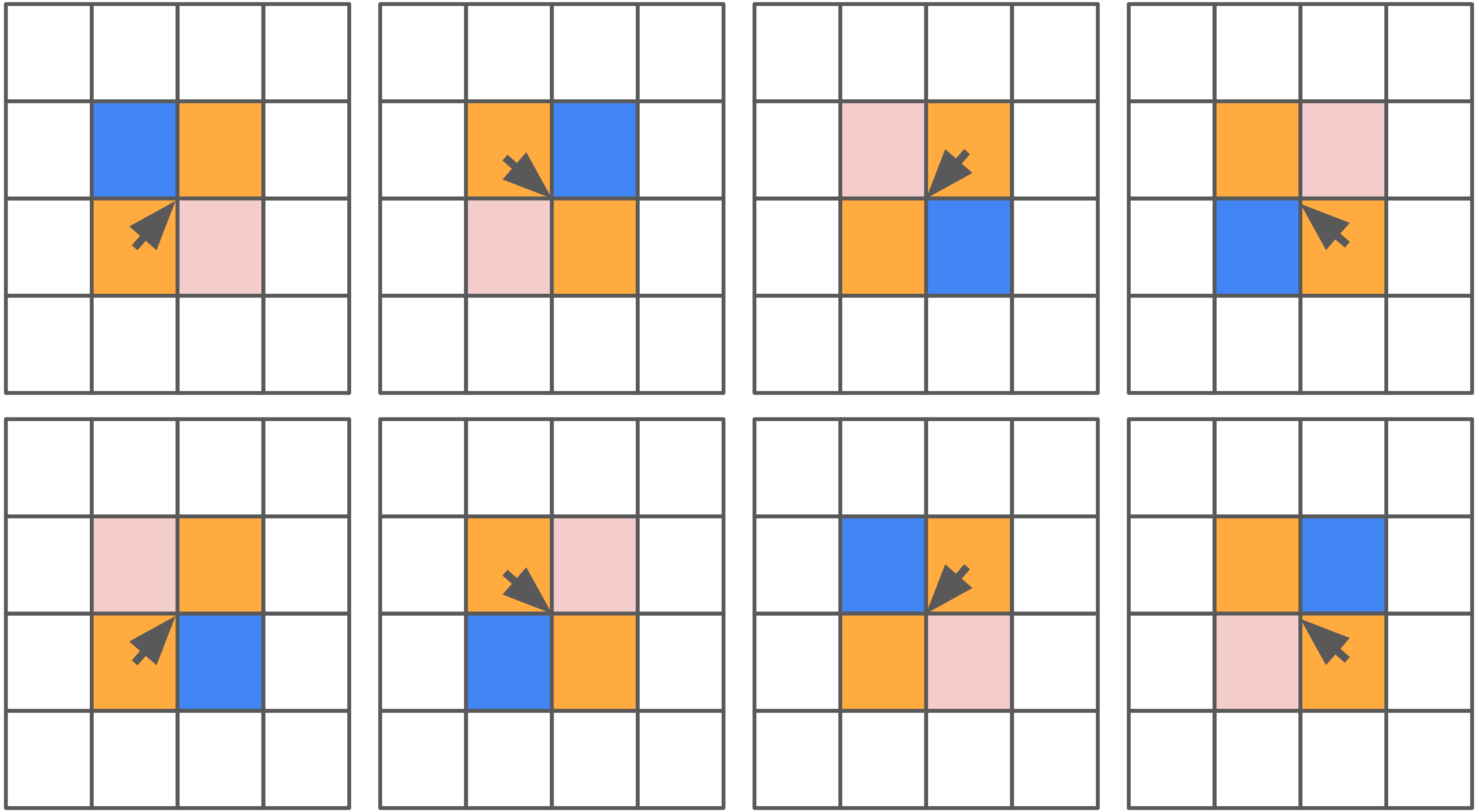
0	Sand
1	Lava
2	Volcano
3	Mountains



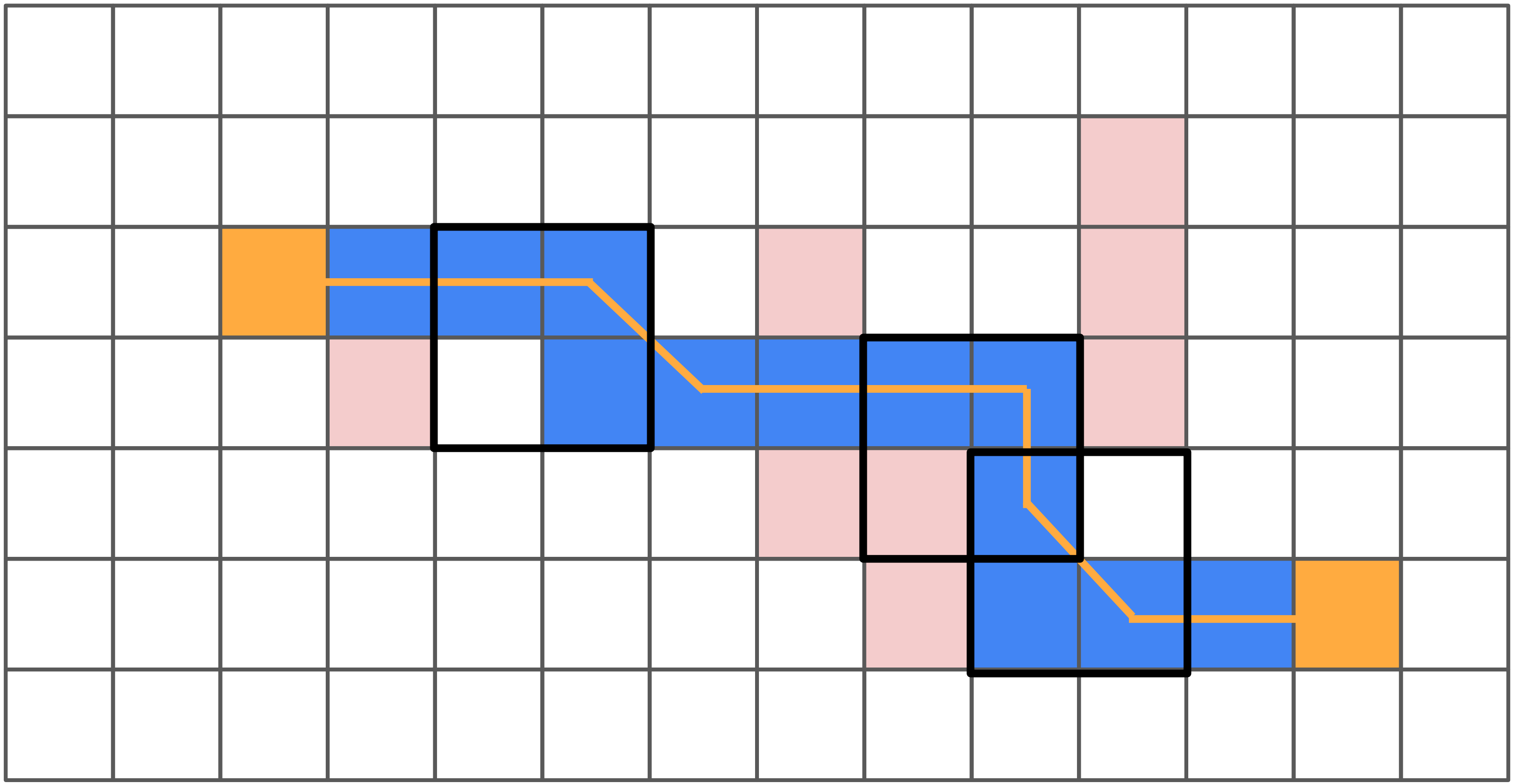
When you see an L-shaped turn, a diagonal move is possible
(hint: use a Peek2 function)



If both adjacent cells are free, do a diagonal move



If one of them is not free, don’t do a diagonal move



Diagonal moves in action

The closing notes

Zero allocs?

- Paths are just an on-stack value (16 bytes)

Zero allocs?

- Paths are just an on-stack value (16 bytes)
- Unlike builtin maps, sparse maps allows 100% re-use

Zero allocs?

- Paths are just an on-stack value (16 bytes)
- Unlike builtin maps, sparse maps allows 100% re-use
- Priority queue is also re-use friendly

A* vs Greedy BFS

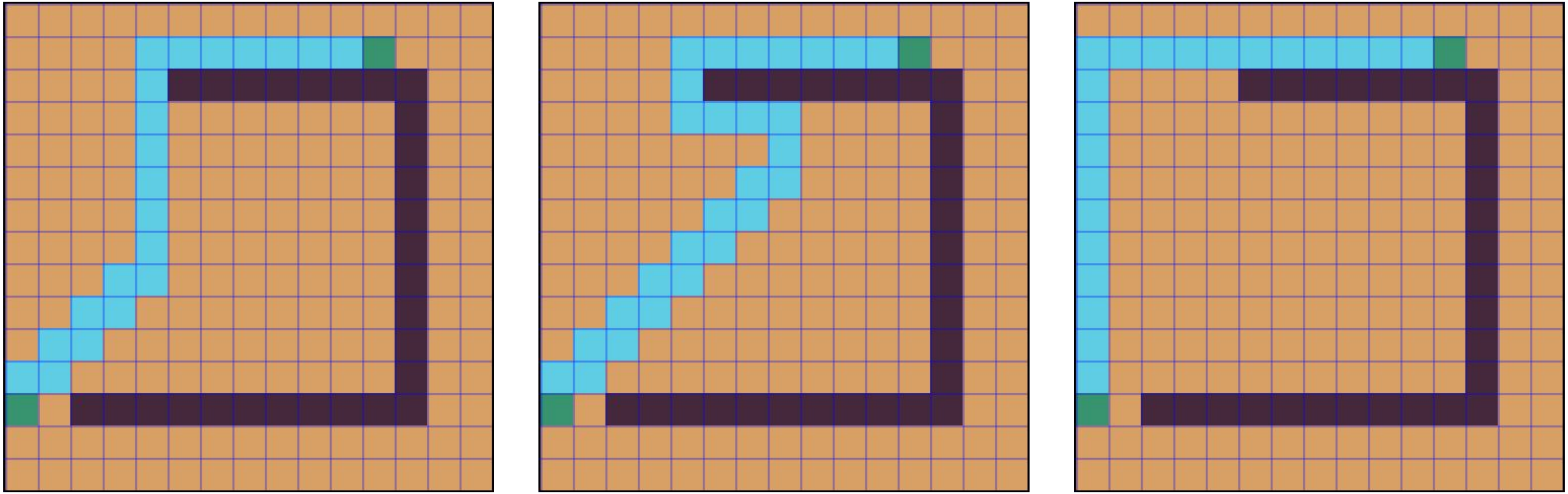
A* pros:

- Optimal paths
- Always finds the path

A* cons:

- More expensive to compute
- Requires more memory

A* (21) | Average BFS (27) | My BFS (21)



A* vs Greedy BFS performance

LIBRARY	no_walls	simple_wall	multi_wall
pathing/bfs	3525	2084	2688
pathing/astar	20140	3415	13310

Useful links

- My library source code
- Sparse set/map explained
- Generations map explained
- Great A* and greedy BFS introduction
- Morton space-filling curve
- Roboden game source code
- Awesome Ebitengine list

Useful links (2)

- [Factorio path finding details]
- [How to get success in life]

Zero alloc pathfinding

@quasilyte 2023

Roboden game
Free & Source-Available

“GOOD GAEM” - quasilyte