Snakes, and an introduction to recursive backtracking

Download the code from GitHub: **wfmag.cc/wfmag51**

Make a snake navigate around a maze using recursive backtracking. Andrew shows you how

AUTHOR
ANDREW GILLETT

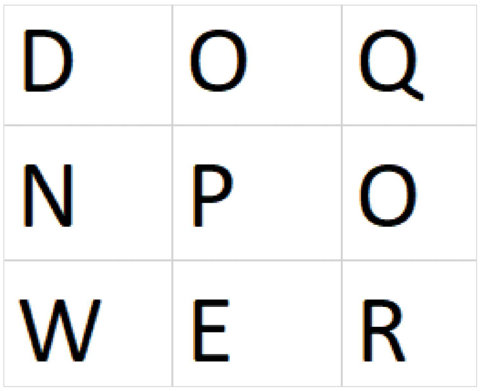
Andrew Gillett is a tutor and programmer who has worked on ten published games including *RollerCoaster Tycoon 3*, *LostWinds 2*, and *Kinectimals*.

Earlier this year I launched *Partition Sector*, my first release as a solo developer. It's a *Bomberman*-style multiplayer game with several team-based game modes, including Capture the Flag, Battle Royale and King of the Hill. During development, I thought a number of times about whether I should add computer-controlled bots to the game. However, writing decent bot artificial intelligence (AI) for this kind of game would have been a considerable challenge. Computer players would have to take many factors into account, combining the need to place bombs in suitable positions with the desire not to get blown up. For a simple *Bomberman* game it might have been doable, but the need for the AI to understand a wide range of power-ups and game modes made it unfeasible.

My plans changed into a game which is nothing like *Bomberman*, but rather more like the word game Boggle, where players must find words within a grid of randomised letters. It's fairly straightforward to write code that finds every possible word within a Boggle board.

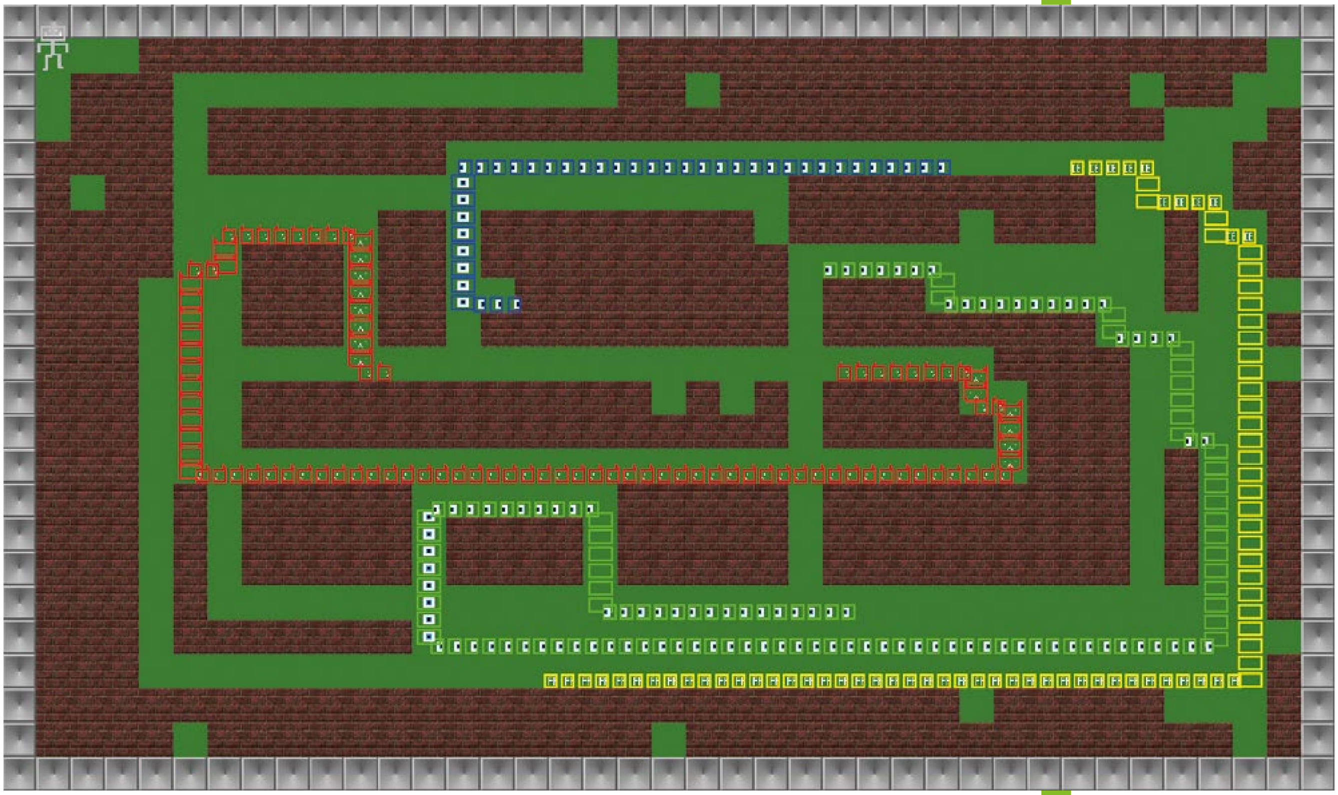
In the grid shown in **Figure 1**, various words can be made by choosing a starting letter and then making a series of moves from there. Unlike in a crossword or traditional word search puzzle, you can change direction as many times as you like in the course of making a word – the only restriction is that you can't use the same tile more than once.

To find all possible words on a Boggle board, you can use a technique known as recursive backtracking. Imagine starting with the letter D on the top left-hand side of the board. Before we do anything else, we're going to check a dictionary to see if 'D' on its own is a word. It's not. The next step is to check a dictionary to see if there are any words that start with D. There are, of course, so we're then going to explore a series of possible words. Let's start by going to the right, so that we now have 'DO'. Obviously, that's a word, so we're off to a good start – we'll print it out.



D	O	Q
N	P	O
W	E	R

Figure 1: A Boggle grid – just starting from the first letter, we can make the words do, door, dope, don, done, and doner.



^ Four snakes in *Partition Sector*. The snakes use player head sprites for their head and body parts.

We then check to see if there are any words that start with 'DO'. There are, so we'll go further down this path, again by moving to the right, so we now have 'DOQ'. That's not a word, and there aren't any words that start with that, so there's no point in going any deeper down this rabbit hole. At this point, we backtrack, and go back to the situation where we were looking for words starting with 'DO'. We've already checked the letter Q to the right, so now we'll go clockwise and check the letter O to the bottom right. 'DOO' isn't a word, but there are words that start with that, so we'll repeat the process again – which will lead to finding the word 'DOOR'. What we have here is an algorithm that tries each possible path through the board, but skips paths that clearly aren't going to be successful.

POWER-UP

What does this have to do with a *Bomberman* game? One of the power-ups in *Partition Sector* turns the player into a snake, in the style of the old Nokia phone game. The snake can grow longer by eating brick walls, but turns back into a (temporarily stunned) normal player if it gets trapped. Each time it reaches the centre of a grid square, it can choose to go in a new direction, although it can't do a 180° turn as it would collide with its own body. I realised that I could implement snake AI using a recursive backtracking algorithm, similar to the one that solves Boggle boards. I created a new game mode named Snake Hunter, where players cooperate to destroy AI-controlled snakes. The snakes aim to grow longer by eating brick walls, and try to avoid getting trapped – defined as reaching a tile where there's nowhere to go. Players can bomb a snake to halve its length – a snake dies when its body is only four tiles long.

"My plans changed into a game which is more like the word game, Boggle"

The code below shows a simplified version of the snake AI from *Partition Sector*, recreated using Python and Pygame Zero.

```
import pgzrun
from random import randint
```

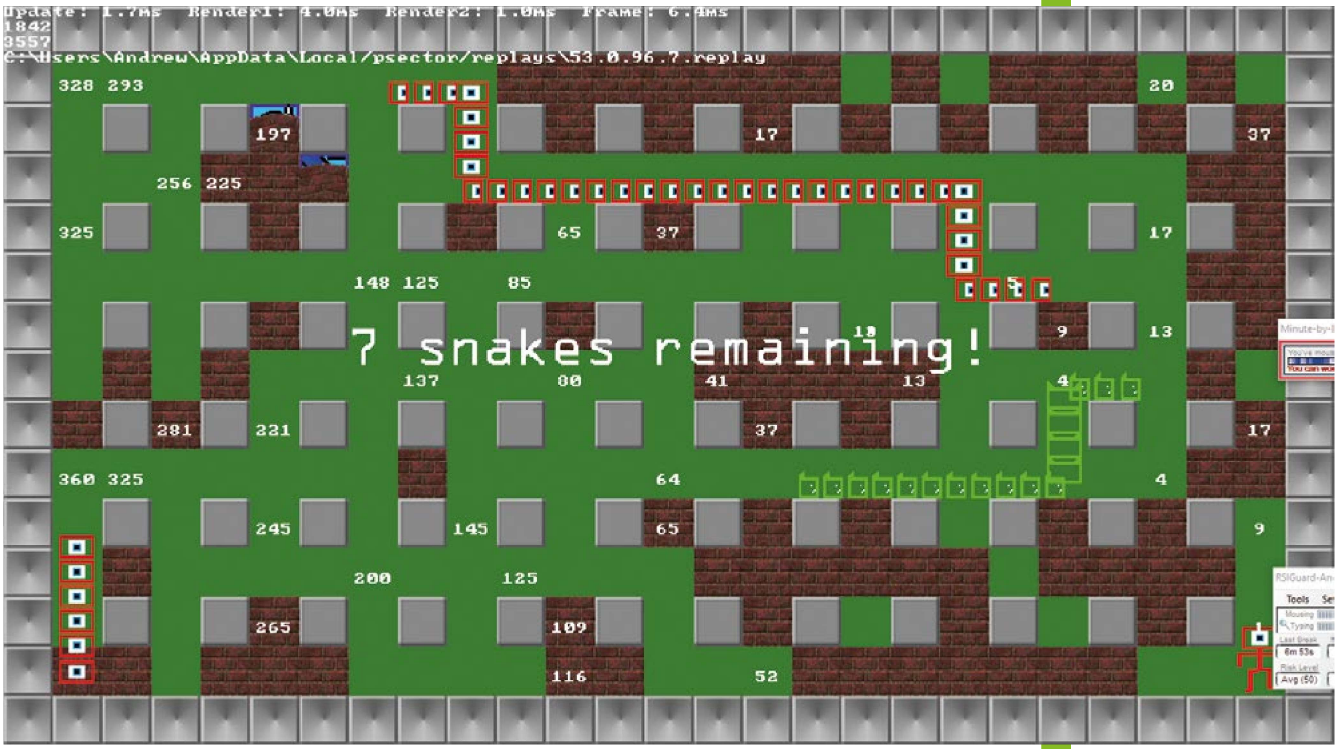
FUNCTION FUN

A recursive function is a function that calls itself. Each time the function calls itself, another instance of that function is added to an area of memory known as the call stack. Each instance of the function has its own separate copy of its local variables. A recursive function always needs what's known as a base case, where it stops calling itself and instead returns a value – the lack of a base case leads to the recursive equivalent of an infinite loop, which will ultimately result in a stack overflow error.

```
WIDTH,HEIGHT = 1200,448
MAX_DEPTH = 20
GRID_SQ_SIZE = 64
TILE_SPRITES = {'X': 'gridblock', '.': 'wall'}
DIRECTIONS = ((1, 0), (-1, 0), (0, 1), (0, -1))

GRID = ['XXXXXXXXXXXXXXXXXXX',
        'X..  .   ..       X',
        'X.X X.X X X X X X X',
        'X .. ..  . ..     X',
        'X X X X X X X X X X',
        'X  .   . ..   .   X',
        'XXXXXXXXXXXXXXXXXXX']

def snake_step_score(score, depth, snake):
   head_pos = snake[0]
   square = GRID[head_pos[1]][head_pos[0]]
   if square == ' ':
       snake = snake[:-1]
   if square == 'X' or head_pos in snake[1:]:
      return score/2
   elif square == '.':
      score += 2
   else:
      score += 1

   if depth <= 0:
      return score

   best_score = 0
   for dir in DIRECTIONS:
      new_head_pos = (head_pos[0]+dir[0],
                      head_pos[1]+dir[1])
      result = snake_step_score(score, depth-1,
                                [new_head_pos]+snake)
      best_score = max(result, best_score)

   return best_score

snake = [(1,1)]

def change_grid_pos(row,col,char):
   grid_row = GRID[row]
   new_row = grid_row[:col] + char + grid_
row[col+1:]
   GRID[row] = new_row

def update_snake():
   global snake
   head_pos = snake[0]
   best_score, best_snake = 0, None
   new_pos_eats_wall = False
   for dir in DIRECTIONS:
      new_head_pos = (head_pos[0] + dir[0],
                      head_pos[1] + dir[1])
      new_snake = [new_head_pos] + snake
      result = snake_step_score(0, MAX_DEPTH,
new_snake)
      if result > best_score:
         best_score = result
         best_snake = new_snake
         new_pos_eats_wall = GRID[new_head_
pos[1]][new_head_pos[0]] == '.'

   if best_snake != None:
      snake = best_snake
      if not new_pos_eats_wall:
         snake = snake[:-1]
      else:
         new_head_pos = snake[0]
         change_grid_pos(new_head_pos[1],
                     new_head_pos[0], ' ')

def update():
   update_snake()

def draw():
   screen.clear()
   for row in range(len(GRID)):
      for col in range(len(GRID[row])):
         square = GRID[row][col]
         if square in TILE_SPRITES:
            x = col * GRID_SQ_SIZE
            y = row * GRID_SQ_SIZE
            screen.blit(TILE_SPRITES[square], (x,
y))
         pos = (col, row)
         if pos in snake:
            if pos == snake[0]:
               image = 'snakehead'
            else:
               image = 'snakebody2'
            screen.blit(image, (col * GRID_SQ_SIZE,
                         row * GRID_SQ_SIZE))

pgzrun.go()
```

SNAKE PASS

In this code, a snake is defined as a list of pairs of grid coordinates. So for example, [(1,1),(2,1),(3,1)] represents a snake with its head at (1,1), and two tail parts in the squares to the right. On each update, the snake looks at the available exits from its current head position, and essentially imagines a series of paths it could take from each exit. Each possible path



```
Update: 1.7ms  Render1: 4.0ms  Render2: 1.0ms  Frame: 6.4ms
1842
3557
C:\Users\Andrew\AppData\Local/psector/replays\53.0.96.7.replay
```

A screenshot from *Partition Sector*'s development – the numbers shown in certain squares are path scores calculated in the snake AI code.

has an associated score – a high-scoring path is one where the snake can travel for a long time without reaching a dead end. Bonus points are given for paths that include edible brick walls. The snake will choose the exit which includes the highest-scoring path.

The key function in assessing the paths is **`snake_step_score`**. This recursive function receives three values. **`score`** is the current score for the path currently being assessed. **`depth`** indicates how many more steps ahead we will look before returning. Without this, the program would try to look an infinite number of steps ahead! **`snake`** is the state of the snake on the current path.

When **`update_snake`** is called, we loop through the four possible directions in which we could go. For each direction, we determine what the new head position would be, and create the list **`new_snake`**, which consists of the existing snake but with the new head position at the front. At this stage **`new_snake`** assumes that a wall will be eaten, so will be one unit longer than before – whether or not a wall is eaten, and what to do about it, is dealt with later.

We then call **`snake_step_score`**. This will eventually return the best possible score from going in the current direction. We give it an initial score of zero, the constant **`MAX_DEPTH`** and **`new_snake`**.

`snake_step_score` starts by checking to see if the grid square under the snake's head is an empty space – i.e. no edible wall or solid wall. If it is an empty space, that means the snake isn't going to grow in length, so we need to remove the last tail piece to account for the fact that a new head piece was added at the front. The next check is to see whether the square is blocked. That could be due to a solid wall, or it could be because part of the snake's tail is in that square. In that case, it's not a path we can go down →

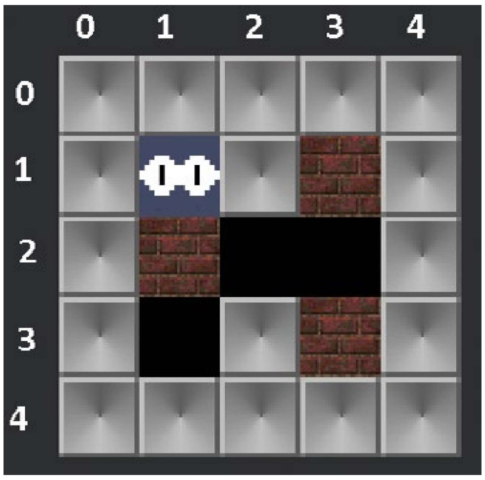


Figure 2: A small level with a snake at position (1,1).

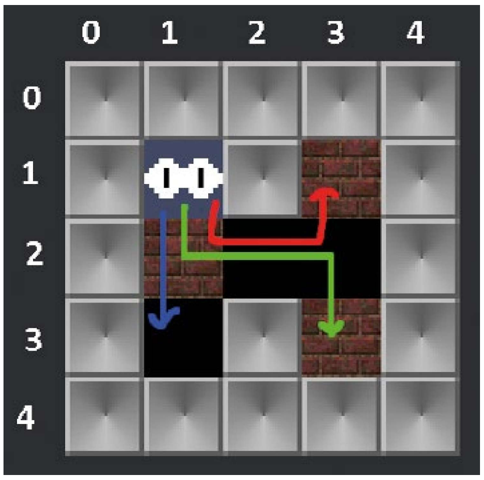


Figure 3: The coloured arrows show the possible paths through our simple maze.

any further, so we return the current score divided by 100. As for why, I'll explain that later. If there isn't a blockage, we add either 1 or 2 to the current score depending on whether we're looking at an empty space or an edible wall. Each step along the path is good, but an edible wall is considered to be better.

The next step is to check to see if **depth** has reached one. If so, we have searched as far as we're allowed to down this path, and we haven't hit a dead end, so we'll just return the current score.

The next part is very similar to the direction-checking code in **update_snake**. For each of the four possible directions, we make a recursive call to **snake_step_score**. We pass in the current score, **depth-1** and a new snake list which consists of the existing snake with the new head position added at the front. The process described above repeats for each step of each possible path. At the end of **snake_step_score** we return the best score returned from the recursive calls.

Before we look at a concrete example, I'll address why hitting an obstacle causes the code to return the current score divided by 100, rather than zero. Hitting an obstacle is bad, so we want to return a very low score, but we still want to favour a long path which ends in a dead end, over a short path which ends in a dead end.

Let's run through an example scenario. **Figure 2** (overleaf) shows a very small level, with a snake at position (1,1). The grey blocks, represented in **GRID** by 'X', are impassable. The brick walls, represented by a dot character, can be eaten. The empty squares are stored as spaces.

In **update_snake**, we call **snake_step_score** for each direction, starting with right and left. Because there are impassable walls in each of those directions, **snake_step_score** returns zero. That's also the case for up, which is checked last. When we check down, however, things get more interesting. Grid position (1,2) contains an edible wall, so **score** goes up by two. We then get to the recursive stage. We first check the path to the right of (1,2), using a recursive call to **snake_step_score**. (2,2) is an empty square – as the snake won't be eating anything here, we score one point and remove the last entry from the **snake** list, to balance out the effect of having added the new head position to the front of the list. It's important to note here that the **snake** variable in **snake_step_score** is not the same as the **snake** global variable. The latter is the current, 'official' snake, which is updated once per frame based on the chosen direction, and displayed on the screen – whereas each recursive instance of **snake_step_score** has its own **snake**, which is the snake being assessed at the current point along a possible future path.

From (2,2), up and down are impassable walls and going left would mean colliding with the body of the snake on the current path. So the next square along this path is (3,2). Again, we score one point and remove the last tail piece. From here, going left or right result in immediate collisions, so the only valid moves are up or down. Each of these squares is a dead end containing an edible wall, so both will score two points.

By this point in these two paths, illustrated in red and green in **Figure 3**, the total score will be six. But the game doesn't yet know that these are dead ends – that happens when we do the next recursive calls to **snake_step_score**, which will find that there are no valid moves from these squares. That means the final returned value from both paths ends up as 0.06, as we divide the score by 100. That's still a better score than the blue path, which will be assessed next, as the algorithm backtracks to (1,2). The blue path will return 0.03 – two points for a brick wall, one point for an empty square, then divide by 100 because it's a dead end. The end result of all this, back in **update_snake**, is that the snake head will move down by one square, from (1,1) to (1,2), because down returned a score of 0.06 whereas up, left, and right, being impassable walls, returned zero. Because the new square contains a wall, we won't remove the tail piece, so the snake now has a length of two. We replace the wall with an empty square using **`change_grid_pos`**. On the next frame, the whole process is repeated, except this time the starting point is a snake with its head at (1,2) and its tail at (1,1). This time the decision will be to go to the right, because that's a better prospect than going down.

THE DOWNSIDE

The main downside of this algorithm is that it can get massively CPU-intensive. In the example above, all paths quickly ended in a dead end. On a larger level, or one which contained places where the snake could go round in a loop indefinitely, we need to put a limit on how many steps ahead we look. This is the purpose of the **`MAX_DEPTH`** constant. On anything other than a very small level, making changes to this will have a big effect on performance. If it's too high, situations where the snake has many possible paths in front of it could cause the game to freeze up for seconds at a time, as the code may have to evaluate tens or hundreds of thousands of possible paths. On the other hand, if **`MAX_DEPTH`** is too low, the snake won't be looking very far ahead and may end up going down a path which ends in a dead end.

For the snake code in *Partition Sector*, I had to make some sacrifices for the sake of performance. Although the game is written in C++, which performs a lot better than Python, the snake AI code needs to run quickly enough to allow the game to maintain 60 frames per second. One sacrifice I had to make is that unlike the Python version of the code, the original snake AI doesn't take into account the movement of the snake's tail as it checks each possible path. Taking this into account means having to make copy of the entire snake list for each recursive call to **`snake_step_score`**. This had too much of a performance penalty to make it feasible. Unlike the Python code, the C++ version allows more than one CPU core to be used – for example, one core can be checking the paths down from the current head square, while another can be simultaneously checking the paths above the current square. The C++ version also dynamically varies **`MAX_DEPTH`** based on how long the previous snake update took, so it can adapt to a variety of level sizes and CPU speeds.

"Hitting an obstacle is bad, so we want to return a very low score"

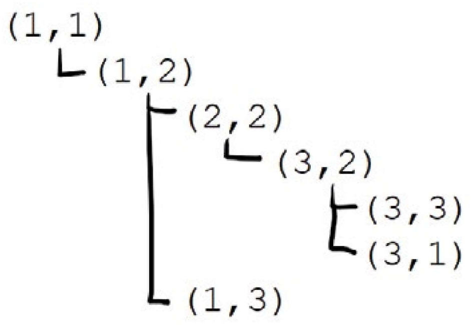
This kind of algorithm can result in emergent behaviour, where the complexity involved makes it very difficult for a programmer to predict how it will behave in practice. During development of the Python version, I occasionally came across a situation where the snake would fail to go for certain edible walls, even though the scoring system incentivises it to do so, and it wasn't in any danger of getting trapped. When the snake reached a particular point on the grid near these walls, it had a choice of going either down, which would lead it to eat the walls, or left, which would take it past them. The snake always went left. I eventually realised why – when it went left, the snake could see that one of the possible paths along this direction would lead it to eat the walls later on. Therefore, both the left and down directions scored the same. Because the code in **`update_snake`** assesses the horizontal directions before the vertical ones, it ends up favouring left over down when those directions give the same score. It just so happened that in this situation, the end result was that it would never eat those walls. One way of fixing this would be to give a higher score to paths which lead to eating walls sooner rather than later.

You can download the full code and image files at GitHub (**wfmag.cc/wfmag51**), which includes additional comments, plus some extra code in **`update`** which randomly adds edible walls to the level over time.

***Partition Sector* is out now on Steam.**

FURTHER READING

For more on the subject of AI and pathfinding, check out Paul Roberts' feature on the subject, An introduction to pathfinding in games, in issue 48's Toolbox. You can purchase a copy or download a PDF from **wfmag.cc/48**



The call tree for the example paths in Figure 2, showing how `snake_score_step` calls itself recursively for each square along a path. Squares which cannot be moved into are not shown.