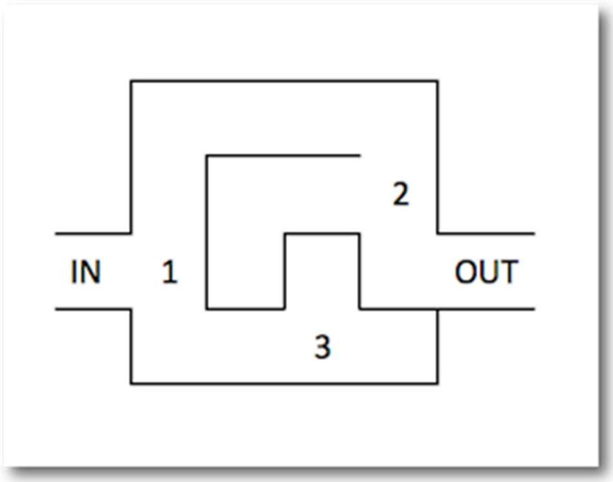
Backtracking explained

Backtracking is one of my favourite algorithms because of its simplicity and elegance; it doesn't always have great performance, but the branch cutting part is really exciting and gives you the idea of progress in performance while you code.

But let's first start with a simple explanation. According to Wikipedia:
Backtracking is a general algorithm for finding all (or some) solutions to some computational problems, that incrementally builds candidates to the solutions, and abandons each partial candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution.

Once you already have used backtracking, it's a pretty straightforward definition, but I realise that when you read it for the first time is not that clear (or—at least—it wasn't to me).

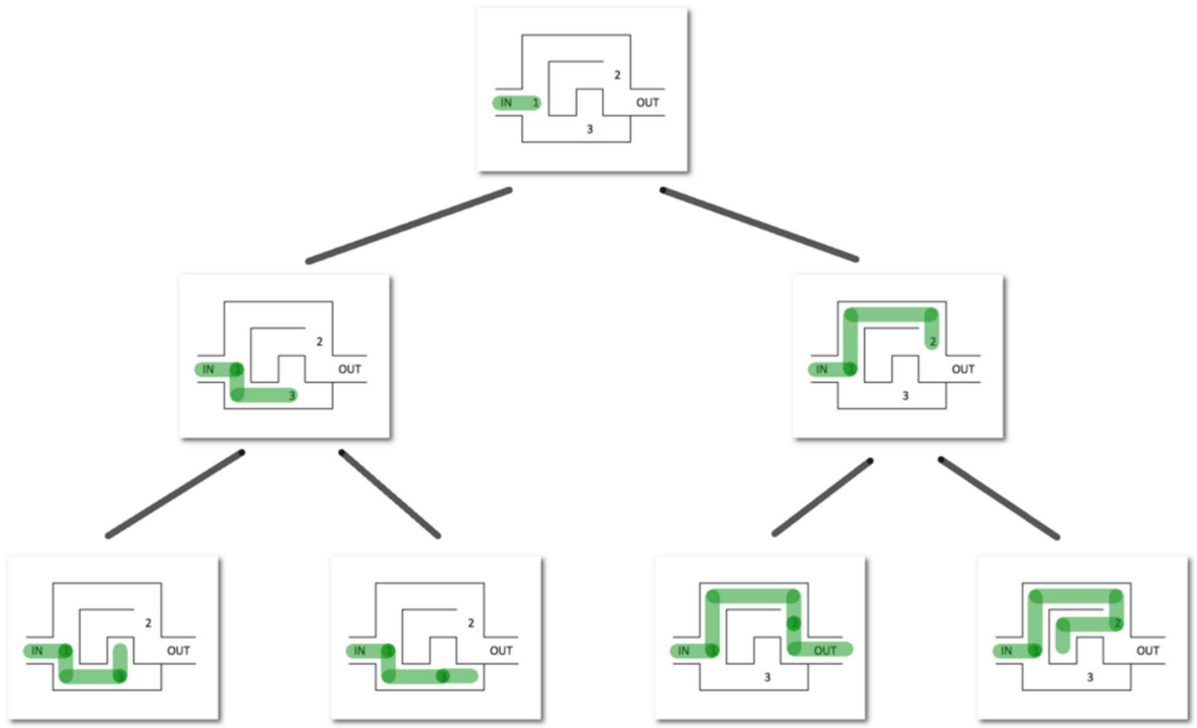
A little example could help us. Imagine to have a maze and you want to find if it has an exit (for sake of precision, algorithms to get out of a maze using graphs are more efficient than backtracking). This is the maze:



A simple maze with only three junctions

where we have labeled the junctions as 1, 2 and 3.

If we want to check every possible path in the maze, we can have a look at the tree of paths, split for every junctions stop:



All the possible paths of the maze

Let’s see a pseudo code for traversing this maze and checking if there’s an exit:

```
function backtrack(junction):

  if is_exit:
    return true
  for each direction of junction:
    if backtrack(next_junction):
      return true

  return false
```

If we apply this pseudo code to the maze we saw above, we’ll see these calls:

```
- at junction 1 chooses down        (possible values: [down, up])
    - at junction 3 chooses right   (possible values: [right,
up])
         no junctions/exit          (return false)
    - at junction 3 chooses up      (possible values: [right,
up])
         no junctions/exit          (return false)
```

```
- at junction 1 chooses **up**        (possible values: [down, up])
    - at junction 2 chooses **down**   (possible values: [down,
left])
         the exit was found!       (*return true*)
```

Please note that every time a line is indented, it means that there was a recursive call. So, when a `no junctions/exit` is found, the function returns a `false` value and goes back to the caller, that resumes to loop on the possible paths starting from the junction. If the loop arrives to the end, that means that from that junction on there's no exit, and so it returns `false`.

The idea is that we can build a solution step by step using recursion; if during the process we realise that is not going to be a valid solution, then we stop computing that solution and we return back to the step before (*backtrack*). In the case of the maze, when we are in a dead-end we are forced to backtrack, but there are other cases in which we could realise that we're heading to a non valid (or not good) solution before having reached it. And that's exactly what we're going to see now.