Images haven't loaded yet. Please exit printing, wait for images to load, and try to print again.

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 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 completeness, there are more efficient algorithms to get out of a maze). 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 realize that is not going to be a valid solution, we then stop that computing 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 realize

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.

. . .

Quite a while ago I've been gifted one of those puzzles based on shaped pieces (*à la* tetris) that has to be framed in form of a square or a rectangle, like this:

after tweaking with it for a while I couldn't come up with a solution, so I decided to write a program to solve the puzzle for me.

I've chosen the Go language and the Gotk3 project (a binding to GTK3 libraries) to write a simple GUI application that -given a puzzle in input- uses backtracking to find all the possible solutions.

The main idea of the algorithm is this: we start with an empty frame and then try to place the first piece; since the canvas is empty, it will for sure fit into it; we recursively try to place the second piece (not overlapping the first), and then the third and so on, until either it finds a piece that cannot be placed into the canvas, or there are no more pieces to place. In

the first case, we have to go back from that branch of execution (we have to *backtrack*) because it makes no sense going on trying to place the remaining pieces if that one cannot be placed (there's no valid solution without that piece); in case of no more pieces to place, that means we found a solution, so we can add it to the set of solutions and go on finding other ones.

. . .

Let's now consider the very nature of this puzzle: the pieces can be rotated and flipped, so for every piece we have to try all its possible rotations. Given that, here's the solver function (a lot of details like data structures and other functions are omitted, but the sense should be clear):

```
func solvePuzzle(puzzle *Puzzle, remainingPieces []Piece) {

  // this is the base case of recursion
  if len(remainingPieces) == 0 {
    addSolution()
    return
  }
```

```
// loops over the remaining pieces
  for piece := range remainingPieces {

    // considers all possible rotations of this piece
    for rot := range piece.Rotations {

      // tries every cell of the grid (limited to the
      // positions where the piece is not outside the
      // boundaries of the frame)
      for j := 0; j <= len(puzzle.Grid[0])-len(rot[0]); j++
{
        for i := 0; i <= len(puzzle.Grid)-len(rot); i++ {

          // if the cell is empty and the piece doesn't
          // overlap with other pieces
          if puzzle.Grid[i][j] == EMPTY &&
            pieceFits(rot, i, j, puzzle.Grid) {

            // adds the piece to the grid
            updatedGrid := addShapeToGrid(rot, i, j,
puzzle)
            puzzle.Grid = updatedGrid


            // removes this piece from the remaining ones
            removePieceFromRemaining(remainingPieces,
piece)
```

```
            // recursively calls this function
            solvePuzzle(puzzle, remainingPieces)

            // after having tried all this branch, remove
this
            // piece and goes on with the next in the loop
            updatedGrid = removeShapeFromGrid(updatedGrid)
            puzzle.Grid = updatedGrid
            addToRemainingPieces(piece)
          }
        }
      }
    }
  }
}
```

If you want to see the real implementation, head to the Github repository: https://github.com/andreaiacono/GoShapesPuzzle.

Considering a 5x6 model, like this one:

the execution time is not exciting: on my notebook it took 1h18m31s. Really too much. It takes so long because the algorithm is placing the pieces in every possible position, even where it make no sense to do it. For example, the algorithm places the pieces in this way:

Those little empty spots will never be filled

Of course those 1-cell and 2-cells empty spaces will never be filled because we don't have any piece small enough to fit into them in this model, and thus the whole branch of computation will eventually fail. So, it would be nice to cut it as soon as we realize that there's an empty space smaller than the smaller of the remaining pieces to place.

This can be achieved adding this check:

```
func hasLeftUnfillableAreas(grid Grid,
     shape Shape,
     i, j int,
     minPieceSize int) bool {
```

```
  gridCopy := addShapeToGrid(shape, i, j, grid)
  var min = math.MaxInt32
  for i := 0; i < len(gridCopy); i++ {
    for j := 0; j < len(gridCopy[0]); j++ {
       if gridCopy[i][j] == EMPTY {
          var area = getAreaSize(&gridCopy, i, j)
          if min > area {
             min = area
          }
       }
    }
  }
  return min < minPieceSize
}
```

```
func getAreaSize(grid *Grid, x, y int) int {
   (*grid)[x][y] = FLOOD_FILL_VALUE
   size := 1

   if y > 0 && (*grid)[x][y-1] == EMPTY {
      size += getAreaSize(grid, x, y-1)
   }
   if x > 0 && (*grid)[x-1][y] == EMPTY {
      size += getAreaSize(grid, x-1, y)
   }
   if x < len((*grid))-1 && (*grid)[x+1][y] == EMPTY {
      size += getAreaSize(grid, x+1, y)
   }
   if y < len((*grid)[0])-1 && (*grid)[x][y+1] == EMPTY {
      size += getAreaSize(grid, x, y+1)
   }

   return size
}
```

First we place into the grid the piece we are examining now, and then we compute the size of every empty area (using a floodfill like algorithm). At the end of the function, we just return if the minimum empty area is smaller than the smaller remaining piece. So if this function returns *true* that means that this branch of computation will never arrive to a solution, and hence we can cut it.

What we've done is to add some extra computation (to find the minimum empty space size) in order to avoid following a branch that will never arrive to a solution; more in general, it depends on the problem we're trying to solve if it makes sense to add the extra computation or not

because it could be something that worsen the general performance of the algorithm.

In our case this extra computation resulted in a total computation time cut from 1h18m31s to 6m19s: a 12.5x increment in performance!

But we can improve the performance further. If we look at the main loop of the solver, we realize that the same configuration is computed multiple times. Let's suppose that the solver starts placing the piece no. 1 in (0,0) and then the piece no.2 in (3,0); when the branch of piece no.1 as the first piece will be over, the solver will start placing piece no.2, and after a while it will place it in (3,0), and the following step will be to place piece no.1 in (0,0) since it's empty. So we are computing again the same state, and recursively, all the states following this.

To avoid this, I created a map that maps a string representation of the grid to a boolean (I would have created a Set with another language, but Go doesn't have it) and this code to check it:

```
func checkAndUpdateVisitedState(grid Grid) bool {

    gridString := fmt.Sprintf("%s", grid)
    _, isPresent := visited[gridString]

    if !isPresent {
       visited[gridString] = true
    }
    return isPresent
}
```

So, every time the solver wants to place a piece, it first checks if it already did it before, and if it did, it just skips this state, otherwise it saves the new state into the map and goes on with that branch. Thanks to this optimization, the total computation time dropped from 6m19s to 1m44: another 3.5x performance increment!

So, from the first implementation we had a 43x performance increase!

If you're interested in seeing the complete source code and run it, you can find it on github: https://github.com/andreaiacono/GoShapesPuzzle.