Building and Solving Mazes ⋆, ⋆⋆

Topics

- Arrays, Records, Simple sum types
- Graphics

Introduction

In this exercise, we will build, display and solve *perfect* mazes. In a *perfect* maze, there is a unique way from entry to exit, and actually between any two points.

Constructing such a maze is done using the following, fairly simple algorithm, unrolled on a small example in Figure 1.

1. First, erect a lattice shaped juxtaposition of square rooms, each enclosed by four walls and painted in a unique color. Select an entry room and an exit one, hopefully among the outer cells so that you wont need a chopper.
2. At each step, choose a wall shared by two rooms of different colors. Pierce this wall with a sledgehammer, install a door and repaint one of the rooms in the color of the other. If this room leads to other rooms (which by induction should be of the same color), paint them as well.
3. Repeat until the maze in entirely painted in a single color.

The *perfect* condition is trivially ensured by the fact that we combine connected zones using a single door.

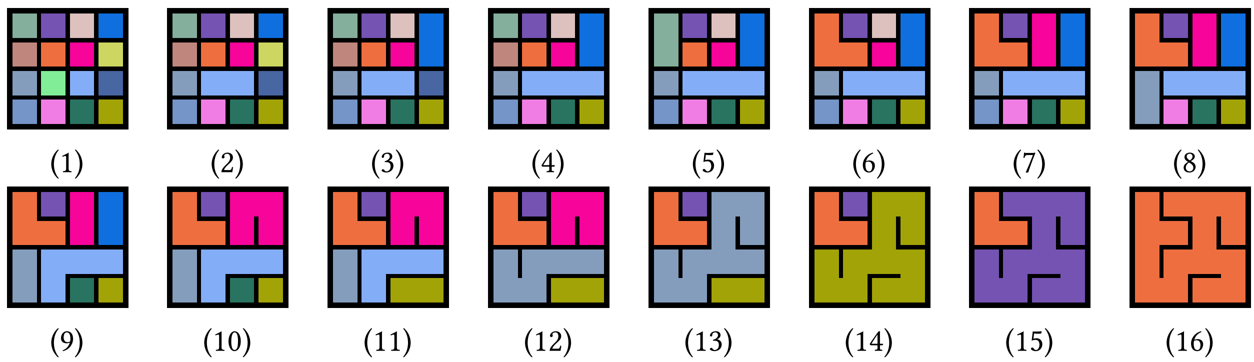

Figure 1: Steps of the maze building algorithm

Exercise 1 – Types and Primitives

Question 1.1 – We will start with simple square shaped cells, that we model by its color and the list of its opened doors. A door is modeled by its orientation. A state in the maze generation algorithm is then just a matrix of cells. A color is any enumerable and comparable type at your convenience.

We will use the following types, you are free to enrich them with any information that you consider useful to the algorithm.

```
type door = Top | Bottom | Left | Right ;;
type cell = {
  mutable id : int ;
  mutable doors : door list ;
}
type maze = {
  width : int ;
  height : int ;
  cells : cell array array ;
}
```

Question 1.2 – Write `new_maze` of *width* and *height* that builds the initial state.

Question 1.3 – Write the following primitive on doors:

- `open_door : (int * int) -> door -> unit`
- `door_opened : (int * int) -> door -> bool`
- `door_closed : (int * int) -> door -> bool`

Question 1.4 – Write `neighbour : (int * int) -> door -> (int * int)` that returns the position in the maze resulting of taking the given door in the given cell. Give also `opposite : door -> door` which returns the door by which we enter the destination room (for instance, the north door of the departure room is the south door of the destination room).

Provide a value `all_doors` listing all possible values of the `door` type (all possible doors of a room).

We defined these operations so that we can later abstract the type `door` in order to change the cell shape.

Question 1.5 – Write `choose_door : int -> int -> ((int * int) * door)` that chooses an internal door in a maze of the given width and height.

Exercise 2 – Generation

Now that we have all the primitives we need, we can write the generation algorithm in a generic manner (without referring to the door type directly).

Question 2.1 – Write `change_color : maze -> (int * int) -> color -> unit` that fills the zone that contains the given position with the given color.

Question 2.2 – Write `connect : maze -> (int * int) -> door -> unit` that breaks the wall in the given room to install the given door and paints the newly connected zones in the same color. This function must do nothing if the door has already been opened or leads to the same zone.

Question 2.3 – Write the main generation function `make_maze` of *width* and *height* that builds a maze using the previous functions. You can define any auxiliary function, for instance to implement the termination criterion.

Exercise 3 – Display

Figure 2 gives a possible display. You can add as much information or decoration as you like, and use fixed or parametric size and style as you wish.

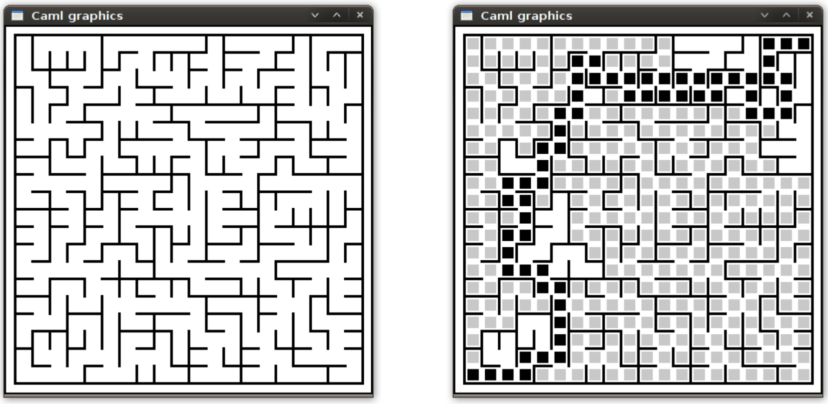

Figure 2: A generated maze and its solution

Question 3.1 – Write `draw_cell : (int * int) -> door list -> unit` that draws the walls around a cell, `mark : maze -> (int * int) -> Graphics.color -> unit` that draws a colored marker in a cell and `clear_mark : maze -> (int * int) -> unit` that removes such a mark.

Question 3.2 – Write `window_size : int -> int -> (int * int)` that computes the required display size for a maze of a given width and height.

Question 3.3 – Give `show_maze : maze -> unit` that displays a maze and wait for a key or mouse input, and try your generation algorithm implementation.

Exercise 4 – Solving

Figure 2 gives a possible illustration of the solving procedure, coloring the solution cells in black and all other cells that have been traversed in grey.

Question 4.1 – Write `solve : maze -> (int * int) -> (int * int) -> door list` that simply tries every door recursively (without going back) and returns the path from the given start to the given finish.

Question 4.2 – Write `solve_and_show : maze -> (int * int) -> (int * int) -> unit`.

Question 4.3 – Enhance `solve` so that it also returns the list of all traversed rooms and correct `solve_and_show` accordingly.

```
solve : maze -> (int * int) -> (int * int) -> door list * (int * int) list
```

Exercise 5 – Bee Hives

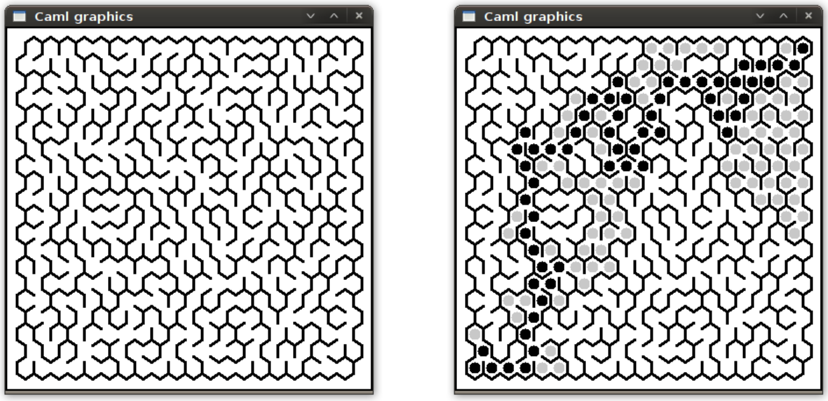

Figure 3: Bee hive shaped maze

Question 5.1 – Identify the parts to be modified.

Question 5.2 (Copy and Paste) – Duplicate the code and modify it in place

Question 5.3 (Functors) – Give another solution using modules to share everything generic.

Question 5.4 (Objets) – Give another solution based on objects.

To go further

Many extensions are possible: other cell shapes (triangles, etc.), more than one kind of cell, generation of LaTeXor SVG, interactive human solving, etc.

Building and Solving Mazes

OLUTIONS – SOLUTIONS – SOLUTIONS – SOLUTION

Solution to question 1.2

```
let new_maze width height =
  { width = width ;
    height = height ;
    cells =
      Array.init height
        (fun y ->
           Array.init width
             (fun x -> {
                 id = y * width + x ;
                 doors = [ ] ;
              } ))
  }
```

Solution to question 1.3

```
let open_door maze (x, y) d =
  let cell = maze.cells.(y).(x) in
    cell.doors <-
      d :: List.filter ((<>) d) cell.doors
let door_opened maze (x, y) d =
  let cell = maze.cells.(y).(x) in
    List.exists ((=) d) cell.doors
let door_closed maze (x, y) d =
  not (door_opened maze (x, y) d)
```

Solution to question 1.4

```
let all_doors =
  [ Top ; Bottom ; Left ; Right ]
let neighbour x y d =
  match d with
    | Top -> x, y + 1
    | Bottom -> x, y - 1
    | Left -> x - 1, y
    | Right -> x + 1, y
let opposite d =
  match d with
    | Top -> Bottom
    | Bottom -> Top
    | Left -> Right
    | Right -> Left
```

Solution to question 1.5

```
let rec choose_door maze =
  let y = Random.int maze.height
  and x = Random.int maze.width in
    match Random.int 4 with
      | 0 when y > 0 ->
        (x, y), Bottom
      | 1 when y < maze.height - 1 ->
        (x, y), Top
      | 2 when x > 0 ->
        (x, y), Left
      | 3 when x < maze.width - 1 ->
        (x, y), Right
      | _ ->
        choose_door maze
```

Solution to question 2.1

```
let rec change_color maze (x, y) id =
  if y >= 0 && y < maze.height && x >= 0 && x < maze.width
    && maze.cells.(y).(x).id <> id then (
      maze.cells.(y).(x).id <- id ;
      List.iter
        (fun d ->
           if door_opened maze (x, y) d then
             let x', y' = neighbour (x, y) d in
               change_id maze (x', y') id)
        all_doors
    )
```

Solution to question 2.2

```
let connect maze (x, y) d =
  let x', y' = neighbour x y d in
    if maze.cells.(y).(x).id <> maze.cells.(y').(x').id then (
      change_color maze (x', y') maze.cells.(y).(x).id ;
      open_door maze (x, y) d ;
      open_door maze (x', y') (opposite d)
    )
```

Solution to question 2.3

```
let connect n_ids maze (x, y) d =
  let x', y' = neighbour (x, y) d in
    if maze.cells.(y).(x).id
       <> maze.cells.(y').(x').id then (
      change_color maze (x', y')
        maze.cells.(y).(x).id ;
      decr n_ids ;
      open_door maze (x, y) d ;
      open_door maze (x', y') (opposite d)
    )
let make_maze width height =
  let maze = new_maze width height in
  let n_ids = ref (maze.width * maze.height) in
    while !n_ids > 1 do
```

```
      let pos, d = choose_door maze in
      connect n_ids pos d
    done ;
    maze
```

Solution to question 3.1

```
open Graphics

let size = 14 and border = 10

let mark maze (x, y) color =
  set_color color ;
  fill_rect
    (x * size + border + 3)
    (y * size + border + 3)
    (size - 6) (size - 6)
let clear_mark maze (x, y) =
  set_color white ;
  fill_rect
    (x * size + border + 3)
    (y * size + border + 3)
    (size - 6) (size - 6)
let draw_cell maze (x, y) =
  if door_closed maze (x, y) Bottom then (
    moveto (x * size + border)
           (y * size + border) ;
    rlineto size 0
  ) ;
  if door_closed maze (x, y) Left then (
    moveto (x * size + border)
           (y * size + border) ;
    rlineto 0 size
  ) ;
  if door_closed maze (x, y) Top then (
    moveto (x * size + border)
           ((y + 1) * size + border) ;
    rlineto size 0
  ) ;
  if door_closed maze (x, y) Right then (
    moveto ((x + 1) * size + border)
           (y * size + border) ;
    rlineto 0 size
  )
```

Solution to question 3.2

```
let window_size maze =
  ((maze.width * size + 2 * border),
   (maze.height * size + 2 * border))
```

Solution to question 3.3

```
let show_maze maze =
  let ww, wh = window_size maze in
```

```
    open_graph (Printf.sprintf " %dx%d" ww wh) ;
    for y = 0 to maze.height - 1 do
      for x = 0 to maze.width - 1 do
        draw_cell maze (x, y)
      done ;
    done ;
    ignore (wait_next_event [ Key_pressed ]) ;
    close_graph ()

let () = show_maze (make_maze 30 30)
```

Solution to exercise 4

```
let rec solve maze (x, y) (ex, ey) d =
  let rec solve (x, y) d =
    if x = ex && y = ey then
      Some []
    else
      List.fold_left
        (fun res d' ->
           match res with
           | Some res -> Some res
           | None ->
             let x', y' = neighbour (x, y) d' in
             match solve (x', y') (opposite d') with
             | Some res -> Some (d' :: res)
             | None -> None)
        None
        (List.filter ((<>) d) maze.cells.(y).(x).doors) in
  match solve (x, y) d with
  | None -> assert false
  | Some res -> res

let solve_and_show maze =
  let ww, wh = window_size maze in
  open_graph (Printf.sprintf " %dx%d" ww wh) ;
  for y = 0 to maze.height - 1 do
    for x = 0 to maze.width - 1 do
      draw_cell maze (x, y)
    done ;
  done ;
  let res = solve maze (0, 0) (maze.width - 1, maze.height - 1) Left in
  let rec draw pos = function
    | [] ->
      mark maze pos black
    | d :: ds ->
      mark maze pos black ;
      draw (neighbour pos d) ds in
  draw (0, 0) res ;
  ignore (wait_next_event [ Key_pressed ]) ;
  close_graph ()

let () = solve_and_show (make_maze 30 30) ;;
```

Solution to question 5.1

Type door.

Value all_doors.

Functions neighbour, opposite, choose_door, draw_cell, mark and window_size.

Solution to question 5.2

```
let () = Random.self_init ()

type door = TopRight | BottomRight | TopLeft | BottomLeft | Left | Right

type cell = {
  mutable color : int ;
  mutable doors : door list ;
}

type maze = {
  width : int ;
  height : int ;
  cells : cell array array ;
}

let all_doors =
  [ TopLeft ; BottomLeft ; TopRight ; BottomRight ; Left ; Right ]

let neighbour (x, y) d =
  match d, y mod 2 = 1 with
    | TopLeft, false -> x - 1, y + 1
    | TopRight, false -> x, y + 1
    | BottomLeft, false -> x - 1, y - 1
    | BottomRight, false -> x, y - 1
    | TopLeft, true -> x, y + 1
    | TopRight, true -> x + 1, y + 1
    | BottomLeft, true -> x, y - 1
    | BottomRight, true -> x + 1, y - 1
    | Left, _ -> x - 1, y
    | Right, _ -> x + 1, y

let opposite d =
  match d with
    | TopRight -> BottomLeft
    | BottomRight -> TopLeft
    | TopLeft -> BottomRight
    | BottomLeft -> TopRight
    | Left -> Right
    | Right -> Left

let new_maze width height =
  { width ; height ;
    cells =
      Array.init height
        (fun y ->
           Array.init width
             (fun x -> {
```

```
                    color = y * width + x ;
                    doors = [ ] ;
                  } )) }

let open_door maze (x, y) d =
  let cell = maze.cells.(y).(x) in
    cell.doors <- d :: List.filter ((<>) d) cell.doors

let door_opened maze (x, y) d =
  let cell = maze.cells.(y).(x) in
    List.exists ((=) d) cell.doors

let door_closed maze (x, y) d =
  not (door_opened maze (x, y) d)

let rec choose_door width height =
  let y = Random.int height
  and x = Random.int width in
    match Random.int 6 with
      | 0 when y > 0 && (x > 0 || y mod 2 = 1) ->
        (x, y), BottomLeft
      | 2 when y < height - 1 && (x > 0 || y mod 2 = 1) ->
        (x, y), TopLeft
      | 1 when y > 0 && (x < width - 1 || y mod 2 = 0) ->
          (x, y), BottomRight
      | 3 when y < height - 1 && (x < width - 1 || y mod 2 = 0) ->
        (x, y), TopRight
      | 4 when x > 0 ->
          (x, y), Left
      | 5 when x < width - 1 ->
          (x, y), Right
      | _ -> choose_door width height

let rec change_color maze (x, y) color =
  if y >= 0 && y < maze.height
     && x >= 0 && x < maze.width
     && maze.cells.(y).(x).color <> color then (
    maze.cells.(y).(x).color <- color ;
    List.iter
      (fun d ->
         if door_opened maze (x, y) d then
           let x', y' = neighbour (x, y) d in
           change_color maze (x', y') color)
      all_doors
  )

let connect n_colors maze (x, y) d =
  let x', y' = neighbour (x, y) d in
  if maze.cells.(y).(x).color <> maze.cells.(y').(x').color then (
    change_color maze (x', y') maze.cells.(y).(x).color ;
    decr n_colors ;
    open_door maze (x, y) d ;
    open_door maze (x', y') (opposite d)
```

```
  )

let make_maze width height =
  let maze = new_maze width height in
  let n_colors = ref (maze.width * maze.height) in
  while !n_colors > 1 do
    let pos, d = choose_door width height in
    connect n_colors maze pos d
  done ;
  maze

open Graphics

let size = 14 and border = 10

let mark maze (x, y) color =
  let sx = size / 2 in
  let sy = 2 * size / 3 in
  let xofs = (y mod 2) * sx in
    set_color color ;
    fill_circle
      (x * size + xofs + sx + border)
      (y * size + sy + border)
      (size / 3)

let draw_cell (x, y) doors =
  let sx = size / 2 in
  let sy = size / 3 in
  let xofs = (y mod 2) * sx in
    set_line_width 3 ;
    set_color black ;
    if not (List.mem TopLeft doors) then (
      moveto (x * size + xofs + sx + border) ((y + 1) * size + sy + border) ;
      rlineto (-sx) (-sy)
    ) ;
    if not (List.mem TopRight doors) then (
      moveto (x * size + xofs + sx + border) ((y + 1) * size + sy + border) ;
      rlineto sx (-sy)
    ) ;
    if not (List.mem BottomLeft doors) then (
      moveto (x * size + xofs + sx + border) (y * size + border) ;
      rlineto (-sx) sy
    ) ;
    if not (List.mem BottomRight doors) then (
      moveto (x * size + xofs + sx + border) (y * size + border) ;
      rlineto sx sy
    ) ;
    if not (List.mem Left doors) then (
      moveto (x * size + xofs + border) (y * size + sy + border) ;
      rlineto 0 (2 * sy)
    ) ;
    if not (List.mem Right doors) then (
      moveto ((x + 1) * size + xofs + border) (y * size + sy + border) ;
```

```
      rlineto 0 (2 * sy)
    )

let window_size width height =
  ((width * size + size / 2 + 2 * border),
   (height * size + size / 3 + 2 * border))

let rec solve maze (x, y) (ex, ey) d =
  let rec solve (x, y) d =
    if x = ex && y = ey then
      Some []
    else
      List.fold_left
        (fun res d' ->
           match res with
           | Some res -> Some res
           | None ->
             let x', y' = neighbour (x, y) d' in
             match solve (x', y') (opposite d') with
             | Some res -> Some (d' :: res)
             | None -> None)
        None
        (List.filter ((<>) d) maze.cells.(y).(x).doors) in
  match solve (x, y) d with
  | None -> assert false
  | Some res -> res

let solve_and_show maze =
  let ww, wh = window_size maze.width maze.height in
  open_graph (Printf.sprintf "␣%dx%d" ww wh) ;
  for y = 0 to maze.height - 1 do
    for x = 0 to maze.width - 1 do
      draw_cell (x, y) maze.cells.(y).(x).doors
    done ;
  done ;
  let res = solve maze (0, 0) (maze.width - 1, maze.height - 1) Left in
  let rec draw pos = function
    | [] ->
      mark maze pos black
    | d :: ds ->
      mark maze pos black ;
      draw (neighbour pos d) ds in
  draw (0, 0) res ;
  ignore (wait_next_event [ Key_pressed ]) ;
  close_graph ()

let () = solve_and_show (make_maze 30 30)
```

Solution to question 5.3

```
module type SHAPE = sig
  type door
  val all_doors : door list
  val neighbour : (int * int) -> door -> (int * int)
```

```
  val opposite : door -> door
  val choose_door : int -> int -> ((int * int) * door)
  val draw_cell : (int * int) -> door list -> unit
  val mark : (int * int) -> Graphics.color -> unit
  val window_size : int -> int -> (int * int)
end

module Maze (Shape : SHAPE) = struct
  include Shape

  type cell = {
    mutable color : int ;
    mutable doors : door list ;
  }

  type maze = {
    width : int ;
    height : int ;
    cells : cell array array ;
  }

  let new_maze width height =
    { width ; height ;
      cells =
        Array.init height
          (fun y ->
             Array.init width
               (fun x -> {
                    color = y * width + x ;
                    doors = [ ] ;
                  } )) }

  let open_door maze (x, y) d =
    let cell = maze.cells.(y).(x) in
    cell.doors <- d :: List.filter ((<>) d) cell.doors

  let door_opened maze (x, y) d =
    let cell = maze.cells.(y).(x) in
    List.exists ((=) d) cell.doors

  let door_closed maze (x, y) d =
    not (door_opened maze (x, y) d)

  let rec change_color maze (x, y) color =
    if y >= 0 && y < maze.height
       && x >= 0 && x < maze.width
       && maze.cells.(y).(x).color <> color then (
      maze.cells.(y).(x).color <- color ;
      List.iter
        (fun d ->
           if door_opened maze (x, y) d then
             let x', y' = neighbour (x, y) d in
             change_color maze (x', y') color)
```

```
      all_doors
  )

let connect n_colors maze (x, y) d =
  let x', y' = neighbour (x, y) d in
  if maze.cells.(y).(x).color <> maze.cells.(y').(x').color then (
    change_color maze (x', y') maze.cells.(y).(x).color ;
    decr n_colors ;
    open_door maze (x, y) d ;
    open_door maze (x', y') (opposite d)
  )

let make_maze width height =
  let maze = new_maze width height in
  let n_colors = ref (maze.width * maze.height) in
  while !n_colors > 1 do
    let pos, d = choose_door width height in
    connect n_colors maze pos d
  done ;
  maze

open Graphics

let rec solve maze (x, y) (ex, ey) d =
  let rec solve (x, y) d =
    if x = ex && y = ey then
      Some []
    else
      List.fold_left
        (fun res d' ->
           match res with
           | Some res -> Some res
           | None ->
             let x', y' = neighbour (x, y) d' in
             match solve (x', y') (opposite d') with
             | Some res -> Some (d' :: res)
             | None -> None)
        None
        (List.filter ((<>) d) maze.cells.(y).(x).doors) in
  match solve (x, y) d with
  | None -> assert false
  | Some res -> res

open Graphics

let solve_and_show maze =
  let ww, wh = window_size maze.width maze.height in
  open_graph (Printf.sprintf "␣%dx%d" ww wh) ;
  for y = 0 to maze.height - 1 do
    for x = 0 to maze.width - 1 do
      draw_cell (x, y) maze.cells.(y).(x).doors
    done ;
  done ;
```

```
    let res =
      solve maze (0, 0)
        (maze.width - 1, maze.height - 1)
        (List.hd all_doors) in
    let rec draw pos = function
      | [] ->
        mark pos black
      | d :: ds ->
        mark pos black ;
        draw (neighbour pos d) ds in
    draw (0, 0) res ;
    ignore (wait_next_event [ Key_pressed ]) ;
    close_graph ()
end

module Square_shape = struct
  let () = Random.self_init ()

  type door = Left | Right | Top | Bottom

  let all_doors =
    [ Left ; Right ; Top ; Bottom ]

  let neighbour (x, y) d =
    match d with
    | Top -> x, y + 1
    | Bottom -> x, y - 1
    | Left -> x - 1, y
    | Right -> x + 1, y

  let opposite d =
    match d with
    | Top -> Bottom
    | Bottom -> Top
    | Left -> Right
    | Right -> Left

  let rec choose_door width height =
    let y = Random.int height
    and x = Random.int width in
    match Random.int 4 with
    | 0 when y > 0 ->
      (x, y), Bottom
    | 1 when y < height - 1 ->
      (x, y), Top
    | 2 when x > 0 ->
      (x, y), Left
    | 3 when x < width - 1 ->
      (x, y), Right
    | _ -> choose_door width height

  open Graphics

```

```
  let size = 14 and border = 10

  let mark (x, y) color =
    set_color color ;
    fill_rect
      (x * size + border + 3)
      (y * size + border + 3)
      (size - 6) (size - 6)

  let draw_cell (x, y) doors =
    set_line_width 3 ;
    set_color black ;
    if not (List.mem Bottom doors) then (
      moveto (x * size + border) (y * size + border) ;
      rlineto size 0
    ) ;
    if not (List.mem Left doors) then (
      moveto (x * size + border) (y * size + border) ;
      rlineto 0 size
    ) ;
    if not (List.mem Top doors) then (
      moveto (x * size + border) ((y + 1) * size + border) ;
      rlineto size 0
    ) ;
    if not (List.mem Right doors) then (
      moveto ((x + 1) * size + border) (y * size + border) ;
      rlineto 0 size
    )

  let window_size width height =
    ((width * size + 2 * border),
     (height * size + 2 * border))

end

module Beehive_shape = struct
  type door = BottomLeft | Left | Right | TopRight | BottomRight | TopLeft

  let all_doors =
    [ BottomRight ; Left ; Right ; TopLeft ; BottomLeft ; TopRight ]

  let neighbour (x, y) d =
    match d, y mod 2 = 1 with
    | TopLeft, false -> x - 1, y + 1
    | TopRight, false -> x, y + 1
    | BottomLeft, false -> x - 1, y - 1
    | BottomRight, false -> x, y - 1
    | TopLeft, true -> x, y + 1
    | TopRight, true -> x + 1, y + 1
    | BottomLeft, true -> x, y - 1
    | BottomRight, true -> x + 1, y - 1
    | Left, _ -> x - 1, y
    | Right, _ -> x + 1, y
```

```
let opposite d =
  match d with
  | TopRight -> BottomLeft
  | BottomRight -> TopLeft
  | TopLeft -> BottomRight
  | BottomLeft -> TopRight
  | Left -> Right
  | Right -> Left

let rec choose_door width height =
  let y = Random.int height
  and x = Random.int width in
  match Random.int 6 with
  | 0 when y > 0 && (x > 0 || y mod 2 = 1) ->
    (x, y), BottomLeft
  | 2 when y < height - 1 && (x > 0 || y mod 2 = 1) ->
    (x, y), TopLeft
  | 1 when y > 0 && (x < width - 1 || y mod 2 = 0) ->
    (x, y), BottomRight
  | 3 when y < height - 1 && (x < width - 1 || y mod 2 = 0) ->
    (x, y), TopRight
  | 4 when x > 0 ->
    (x, y), Left
  | 5 when x < width - 1 ->
    (x, y), Right
  | _ -> choose_door width height


open Graphics

let size = 14 and border = 10

let mark (x, y) color =
  let sx = size / 2 in
  let sy = 2 * size / 3 in
  let xofs = (y mod 2) * sx in
  set_color color ;
  fill_circle
    (x * size + xofs + sx + border)
    (y * size + sy + border)
    (size / 3)

let draw_cell (x, y) doors =
  let sx = size / 2 in
  let sy = size / 3 in
  let xofs = (y mod 2) * sx in
  set_line_width 3 ;
  set_color black ;
  if not (List.mem TopLeft doors) then (
    moveto (x * size + xofs + sx + border) ((y + 1) * size + sy + border) ;
    rlineto (-sx) (-sy)
  ) ;
```

```
    if not (List.mem TopRight doors) then (
      moveto (x * size + xofs + sx + border) ((y + 1) * size + sy + border) ;
      rlineto sx (-sy)
    ) ;
    if not (List.mem BottomLeft doors) then (
      moveto (x * size + xofs + sx + border) (y * size + border) ;
      rlineto (-sx) sy
    ) ;
    if not (List.mem BottomRight doors) then (
      moveto (x * size + xofs + sx + border) (y * size + border) ;
      rlineto sx sy
    ) ;
    if not (List.mem Left doors) then (
      moveto (x * size + xofs + border) (y * size + sy + border) ;
      rlineto 0 (2 * sy)
    ) ;
    if not (List.mem Right doors) then (
      moveto ((x + 1) * size + xofs + border) (y * size + sy + border) ;
      rlineto 0 (2 * sy)
    )

  let window_size width height =
    ((width * size + size / 2 + 2 * border),
     (height * size + size / 3 + 2 * border))

end

let () =
  Random.self_init () ;
  let module Square = Maze (Square_shape) in
  Square.(solve_and_show (make_maze 30 30)) ;
  let module Beehive = Maze (Beehive_shape) in
  Beehive.(solve_and_show (make_maze 30 30))
```