

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 is entirely painted in a single color.

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

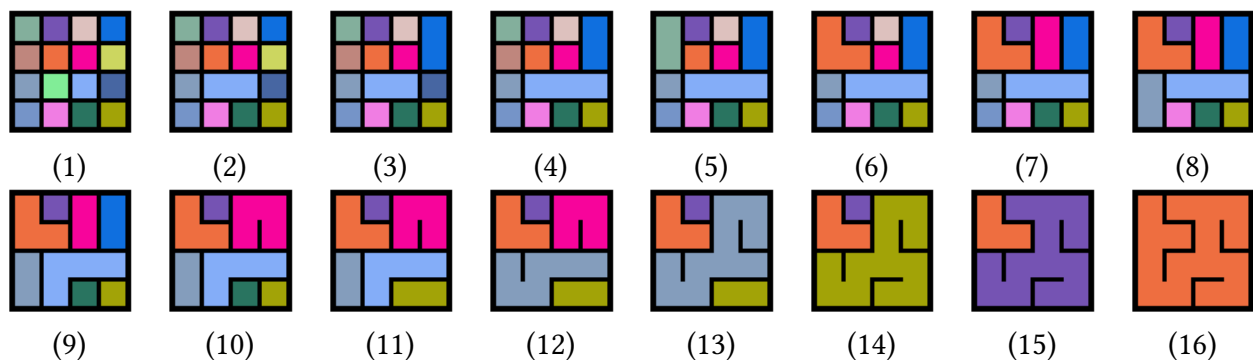


Figure 1: Steps of the maze building algorithm

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.

```
1 type door = Top | Bottom | Left | Right ;;
2 type cell = {
3   mutable id : int ;
4   mutable doors : door list ;
5 }
6 type maze = {
7   width : int ;
8   height : int ;
9   cells : cell array array ;
10 }
```

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.

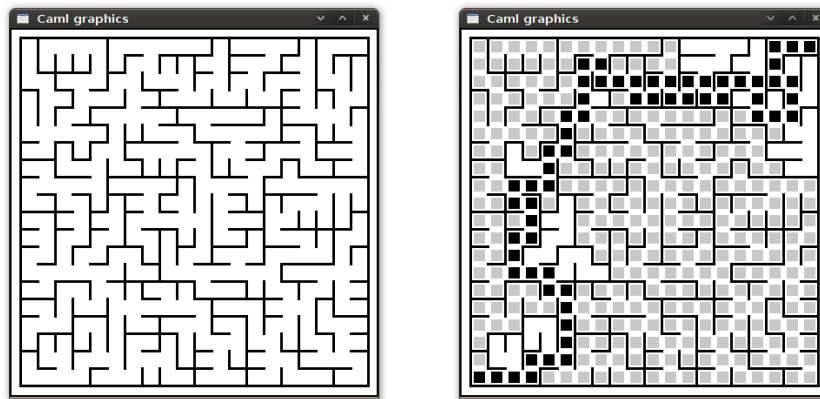


Figure 2: A generated maze and its solution

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`

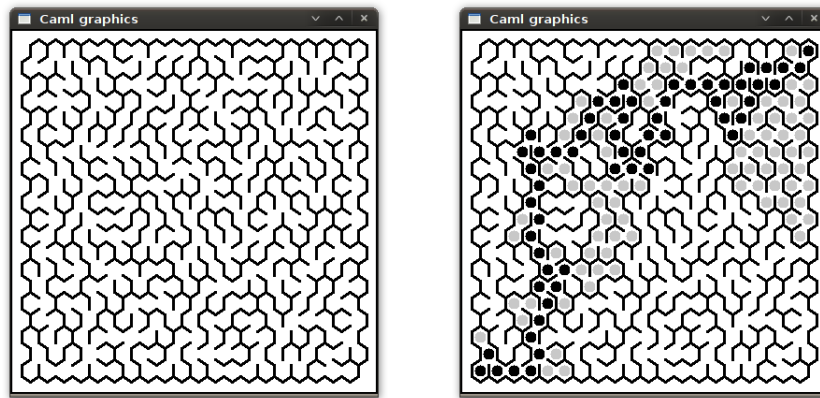


Figure 3: Bee hive shaped maze

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 \LaTeX or SVG, interactive human solving, etc.

Building and Solving Mazes

SOLUTIONS – SOLUTIONS – SOLUTIONS – SOLUTIONS

Solution to question 1.2

```
1 let new_maze width height =
2   { width = width ;
3     height = height ;
4     cells =
5       Array.init height
6         (fun y ->
7           Array.init width
8             (fun x -> {
9               id = y * width + x ;
10              doors = [ ] ;
11            } ))
12 }
```

Solution to question 1.3

```
1 let open_door maze (x, y) d =
2   let cell = maze.cells.(y).(x) in
3   cell.doors <-
4     d :: List.filter ((<>) d) cell.doors
5 let door_opened maze (x, y) d =
6   let cell = maze.cells.(y).(x) in
7   List.exists (=(d)) cell.doors
8 let door_closed maze (x, y) d =
9   not (door_opened maze (x, y) d)
```

Solution to question 1.4

```
1 let all_doors =
2   [ Top ; Bottom ; Left ; Right ]
3 let neighbour x y d =
4   match d with
5   | Top -> x, y + 1
6   | Bottom -> x, y - 1
7   | Left -> x - 1, y
8   | Right -> x + 1, y
9 let opposite d =
10  match d with
11  | Top -> Bottom
12  | Bottom -> Top
13  | Left -> Right
14  | Right -> Left
```

Solution to question 1.5

```

1 let rec choose_door maze =
2   let y = Random.int maze.height
3   and x = Random.int maze.width in
4   match Random.int 4 with
5   | 0 when y > 0 ->
6     (x, y), Bottom
7   | 1 when y < maze.height - 1 ->
8     (x, y), Top
9   | 2 when x > 0 ->
10    (x, y), Left
11   | 3 when x < maze.width - 1 ->
12    (x, y), Right
13   | _ ->
14     choose_door maze

```

Solution to question 2.1

```

1 let rec change_color maze (x, y) id =
2   if y >= 0 && y < maze.height && x >= 0 && x < maze.width
3   && maze.cells.(y).(x).id <> id then (
4     maze.cells.(y).(x).id <- id ;
5     List.iter
6       (fun d ->
7         if door_opened maze (x, y) d then
8           let x', y' = neighbour (x, y) d in
9             change_id maze (x', y') id)
10    all_doors
11  )

```

Solution to question 2.2

```

1 let connect maze (x, y) d =
2   let x', y' = neighbour x y d in
3   if maze.cells.(y).(x).id <> maze.cells.(y').(x').id then (
4     change_color maze (x', y') maze.cells.(y).(x).id ;
5     open_door maze (x, y) d ;
6     open_door maze (x', y') (opposite d)
7   )

```

Solution to question 2.3

```

1 let connect n_ids maze (x, y) d =
2   let x', y' = neighbour (x, y) d in
3   if maze.cells.(y).(x).id
4     <> maze.cells.(y').(x').id then (
5     change_color maze (x', y')
6     maze.cells.(y).(x).id ;
7     decr n_ids ;
8     open_door maze (x, y) d ;
9     open_door maze (x', y') (opposite d)
10  )
11 let make_maze width height =
12   let maze = new_maze width height in
13   let n_ids = ref (maze.width * maze.height) in
14   while !n_ids > 1 do

```

```

15     let pos, d = choose_door maze in
16     connect n_ids pos d
17 done ;
18 maze

```

Solution to question 3.1

```

1 open Graphics
2
3 let size = 14 and border = 10
4
5 let mark maze (x, y) color =
6   set_color color ;
7   fill_rect
8     (x * size + border + 3)
9     (y * size + border + 3)
10    (size - 6) (size - 6)
11 let clear_mark maze (x, y) =
12   set_color white ;
13   fill_rect
14     (x * size + border + 3)
15     (y * size + border + 3)
16     (size - 6) (size - 6)
17 let draw_cell maze (x, y) =
18   if door_closed maze (x, y) Bottom then (
19     moveto (x * size + border)
20             (y * size + border) ;
21     rlineto size 0
22   ) ;
23   if door_closed maze (x, y) Left then (
24     moveto (x * size + border)
25             (y * size + border) ;
26     rlineto 0 size
27   ) ;
28   if door_closed maze (x, y) Top then (
29     moveto (x * size + border)
30             ((y + 1) * size + border) ;
31     rlineto size 0
32   ) ;
33   if door_closed maze (x, y) Right then (
34     moveto ((x + 1) * size + border)
35             (y * size + border) ;
36     rlineto 0 size
37   )

```

Solution to question 3.2

```

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

```

Solution to question 3.3

```

1 let show_maze maze =
2   let ww, wh = window_size maze in

```

```

3   open_graph (Printf.sprintf "%dx%d" ww wh) ;
4   for y = 0 to maze.height - 1 do
5       for x = 0 to maze.width - 1 do
6           draw_cell maze (x, y)
7       done ;
8   done ;
9   ignore (wait_next_event [ Key_pressed ]) ;
10  close_graph ()
11
12  let () = show_maze (make_maze 30 30)

```

Solution to exercise 4

```

1  let rec solve maze (x, y) (ex, ey) d =
2      let rec solve (x, y) d =
3          if x = ex && y = ey then
4              Some []
5          else
6              List.fold_left
7                  (fun res d' ->
8                      match res with
9                          | Some res -> Some res
10                         | None ->
11                             let x', y' = neighbour (x, y) d' in
12                             match solve (x', y') (opposite d') with
13                                 | Some res -> Some (d' :: res)
14                                 | None -> None)
15                      None
16                  (List.filter ((<>) d) maze.cells.(y).(x).doors) in
17      match solve (x, y) d with
18      | None -> assert false
19      | Some res -> res
20
21  let solve_and_show maze =
22      let ww, wh = window_size maze in
23      open_graph (Printf.sprintf "%dx%d" ww wh) ;
24      for y = 0 to maze.height - 1 do
25          for x = 0 to maze.width - 1 do
26              draw_cell maze (x, y)
27          done ;
28      done ;
29      let res = solve maze (0, 0) (maze.width - 1, maze.height - 1) Left in
30      let rec draw pos = function
31          | [] ->
32              mark maze pos black
33          | d :: ds ->
34              mark maze pos black ;
35              draw (neighbour pos d) ds in
36      draw (0, 0) res ;
37      ignore (wait_next_event [ Key_pressed ]) ;
38      close_graph ()
39
40  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

```
1 let () = Random.self_init ()
2
3 type door = TopRight | BottomRight | TopLeft | BottomLeft | Left | Right
4
5 type cell = {
6   mutable color : int ;
7   mutable doors : door list ;
8 }
9
10 type maze = {
11   width : int ;
12   height : int ;
13   cells : cell array array ;
14 }
15
16 let all_doors =
17   [ TopLeft ; BottomLeft ; TopRight ; BottomRight ; Left ; Right ]
18
19 let neighbour (x, y) d =
20   match d, y mod 2 = 1 with
21   | TopLeft, false -> x - 1, y + 1
22   | TopRight, false -> x, y + 1
23   | BottomLeft, false -> x - 1, y - 1
24   | BottomRight, false -> x, y - 1
25   | TopLeft, true -> x, y + 1
26   | TopRight, true -> x + 1, y + 1
27   | BottomLeft, true -> x, y - 1
28   | BottomRight, true -> x + 1, y - 1
29   | Left, _ -> x - 1, y
30   | Right, _ -> x + 1, y
31
32 let opposite d =
33   match d with
34   | TopRight -> BottomLeft
35   | BottomRight -> TopLeft
36   | TopLeft -> BottomRight
37   | BottomLeft -> TopRight
38   | Left -> Right
39   | Right -> Left
40
41 let new_maze width height =
42   { width ; height ;
43     cells =
44       Array.init height
45         (fun y ->
46           Array.init width
47             (fun x -> {
```

```

48         color = y * width + x ;
49         doors = [ ] ;
50     } )) }
51
52 let open_door maze (x, y) d =
53     let cell = maze.cells.(y).(x) in
54     cell.doors <- d :: List.filter ((<>) d) cell.doors
55
56 let door_opened maze (x, y) d =
57     let cell = maze.cells.(y).(x) in
58     List.exists ((=) d) cell.doors
59
60 let door_closed maze (x, y) d =
61     not (door_opened maze (x, y) d)
62
63 let rec choose_door width height =
64     let y = Random.int height
65     and x = Random.int width in
66     match Random.int 6 with
67     | 0 when y > 0 && (x > 0 || y mod 2 = 1) ->
68         (x, y), BottomLeft
69     | 2 when y < height - 1 && (x > 0 || y mod 2 = 1) ->
70         (x, y), TopLeft
71     | 1 when y > 0 && (x < width - 1 || y mod 2 = 0) ->
72         (x, y), BottomRight
73     | 3 when y < height - 1 && (x < width - 1 || y mod 2 = 0) ->
74         (x, y), TopRight
75     | 4 when x > 0 ->
76         (x, y), Left
77     | 5 when x < width - 1 ->
78         (x, y), Right
79     | _ -> choose_door width height
80
81 let rec change_color maze (x, y) color =
82     if y >= 0 && y < maze.height
83     && x >= 0 && x < maze.width
84     && maze.cells.(y).(x).color <> color then (
85     maze.cells.(y).(x).color <- color ;
86     List.iter
87         (fun d ->
88             if door_opened maze (x, y) d then
89                 let x', y' = neighbour (x, y) d in
90                 change_color maze (x', y') color)
91     all_doors
92 )
93
94 let connect n_colors maze (x, y) d =
95     let x', y' = neighbour (x, y) d in
96     if maze.cells.(y).(x).color <> maze.cells.(y').(x').color then (
97     change_color maze (x', y') maze.cells.(y).(x).color ;
98     decr n_colors ;
99     open_door maze (x, y) d ;
100    open_door maze (x', y') (opposite d)

```

```

101 )
102
103 let make_maze width height =
104   let maze = new_maze width height in
105   let n_colors = ref (maze.width * maze.height) in
106   while !n_colors > 1 do
107     let pos, d = choose_door width height in
108     connect n_colors maze pos d
109   done ;
110   maze
111
112 open Graphics
113
114 let size = 14 and border = 10
115
116 let mark_maze (x, y) color =
117   let sx = size / 2 in
118   let sy = 2 * size / 3 in
119   let xofs = (y mod 2) * sx in
120   set_color color ;
121   fill_circle
122     (x * size + xofs + sx + border)
123     (y * size + sy + border)
124     (size / 3)
125
126 let draw_cell (x, y) doors =
127   let sx = size / 2 in
128   let sy = size / 3 in
129   let xofs = (y mod 2) * sx in
130   set_line_width 3 ;
131   set_color black ;
132   if not (List.mem TopLeft doors) then (
133     moveto (x * size + xofs + sx + border) ((y + 1) * size + sy + border) ;
134     rlineto (-sx) (-sy)
135   ) ;
136   if not (List.mem TopRight doors) then (
137     moveto (x * size + xofs + sx + border) ((y + 1) * size + sy + border) ;
138     rlineto sx (-sy)
139   ) ;
140   if not (List.mem BottomLeft doors) then (
141     moveto (x * size + xofs + sx + border) (y * size + border) ;
142     rlineto (-sx) sy
143   ) ;
144   if not (List.mem BottomRight doors) then (
145     moveto (x * size + xofs + sx + border) (y * size + border) ;
146     rlineto sx sy
147   ) ;
148   if not (List.mem Left doors) then (
149     moveto (x * size + xofs + border) (y * size + sy + border) ;
150     rlineto 0 (2 * sy)
151   ) ;
152   if not (List.mem Right doors) then (
153     moveto ((x + 1) * size + xofs + border) (y * size + sy + border) ;

```

```

154     rlineto 0 (2 * sy)
155   )
156
157 let window_size width height =
158   ((width * size + size / 2 + 2 * border),
159    (height * size + size / 3 + 2 * border))
160
161 let rec solve maze (x, y) (ex, ey) d =
162   let rec solve (x, y) d =
163     if x = ex && y = ey then
164       Some []
165     else
166       List.fold_left
167         (fun res d' ->
168           match res with
169           | Some res -> Some res
170           | None ->
171             let x', y' = neighbour (x, y) d' in
172             match solve (x', y') (opposite d') with
173             | Some res -> Some (d' :: res)
174             | None -> None)
175         None
176         (List.filter ((<>) d) maze.cells.(y).(x).doors) in
177   match solve (x, y) d with
178   | None -> assert false
179   | Some res -> res
180
181 let solve_and_show maze =
182   let ww, wh = window_size maze.width maze.height in
183   open_graph (Printf.sprintf "%dx%d" ww wh) ;
184   for y = 0 to maze.height - 1 do
185     for x = 0 to maze.width - 1 do
186       draw_cell (x, y) maze.cells.(y).(x).doors
187     done ;
188   done ;
189   let res = solve maze (0, 0) (maze.width - 1, maze.height - 1) Left in
190   let rec draw pos = function
191     | [] ->
192       mark maze pos black
193     | d :: ds ->
194       mark maze pos black ;
195       draw (neighbour pos d) ds in
196   draw (0, 0) res ;
197   ignore (wait_next_event [ Key_pressed ]) ;
198   close_graph ()
199
200 let () = solve_and_show (make_maze 30 30)

```

Solution to question 5.3

```

1 module type SHAPE = sig
2   type door
3   val all_doors : door list
4   val neighbour : (int * int) -> door -> (int * int)

```

```

5  val opposite : door -> door
6  val choose_door : int -> int -> ((int * int) * door)
7  val draw_cell : (int * int) -> door list -> unit
8  val mark : (int * int) -> Graphics.color -> unit
9  val window_size : int -> int -> (int * int)
10 end
11
12 module Maze (Shape : SHAPE) = struct
13   include Shape
14
15   type cell = {
16     mutable color : int ;
17     mutable doors : door list ;
18   }
19
20   type maze = {
21     width : int ;
22     height : int ;
23     cells : cell array array ;
24   }
25
26   let new_maze width height =
27     { width ; height ;
28       cells =
29         Array.init height
30           (fun y ->
31             Array.init width
32               (fun x -> {
33                 color = y * width + x ;
34                 doors = [ ] ;
35               } )) }
36
37   let open_door maze (x, y) d =
38     let cell = maze.cells.(y).(x) in
39     cell.doors <- d :: List.filter ((<>) d) cell.doors
40
41   let door_opened maze (x, y) d =
42     let cell = maze.cells.(y).(x) in
43     List.exists ((=) d) cell.doors
44
45   let door_closed maze (x, y) d =
46     not (door_opened maze (x, y) d)
47
48   let rec change_color maze (x, y) color =
49     if y >= 0 && y < maze.height
50       && x >= 0 && x < maze.width
51       && maze.cells.(y).(x).color <> color then (
52     maze.cells.(y).(x).color <- color ;
53     List.iter
54       (fun d ->
55         if door_opened maze (x, y) d then
56           let x', y' = neighbour (x, y) d in
57           change_color maze (x', y') color)

```

```

58     all_doors
59 )
60
61 let connect n_colors maze (x, y) d =
62   let x', y' = neighbour (x, y) d in
63   if maze.cells.(y).(x).color <> maze.cells.(y').(x').color then (
64     change_color maze (x', y') maze.cells.(y).(x).color ;
65     decr n_colors ;
66     open_door maze (x, y) d ;
67     open_door maze (x', y') (opposite d)
68   )
69
70 let make_maze width height =
71   let maze = new_maze width height in
72   let n_colors = ref (maze.width * maze.height) in
73   while !n_colors > 1 do
74     let pos, d = choose_door width height in
75     connect n_colors maze pos d
76   done ;
77   maze
78
79 open Graphics
80
81 let rec solve maze (x, y) (ex, ey) d =
82   let rec solve (x, y) d =
83     if x = ex && y = ey then
84       Some []
85     else
86       List.fold_left
87         (fun res d' ->
88           match res with
89             | Some res -> Some res
90             | None ->
91               let x', y' = neighbour (x, y) d' in
92               match solve (x', y') (opposite d') with
93                 | Some res -> Some (d' :: res)
94                 | None -> None)
95         None
96         (List.filter ((<>) d) maze.cells.(y).(x).doors) in
97   match solve (x, y) d with
98   | None -> assert false
99   | Some res -> res
100
101 open Graphics
102
103 let solve_and_show maze =
104   let ww, wh = window_size maze.width maze.height in
105   open_graph (Printf.sprintf "_%dx%d" ww wh) ;
106   for y = 0 to maze.height - 1 do
107     for x = 0 to maze.width - 1 do
108       draw_cell (x, y) maze.cells.(y).(x).doors
109     done ;
110   done ;

```

```

111   let res =
112       solve maze (0, 0)
113           (maze.width - 1, maze.height - 1)
114           (List.hd all_doors) in
115   let rec draw pos = function
116       | [] ->
117           mark pos black
118       | d :: ds ->
119           mark pos black ;
120           draw (neighbour pos d) ds in
121   draw (0, 0) res ;
122   ignore (wait_next_event [ Key_pressed ]) ;
123   close_graph ()
124 end
125
126 module Square_shape = struct
127   let () = Random.self_init ()
128
129   type door = Left | Right | Top | Bottom
130
131   let all_doors =
132       [ Left ; Right ; Top ; Bottom ]
133
134   let neighbour (x, y) d =
135       match d with
136       | Top -> x, y + 1
137       | Bottom -> x, y - 1
138       | Left -> x - 1, y
139       | Right -> x + 1, y
140
141   let opposite d =
142       match d with
143       | Top -> Bottom
144       | Bottom -> Top
145       | Left -> Right
146       | Right -> Left
147
148   let rec choose_door width height =
149       let y = Random.int height
150       and x = Random.int width in
151       match Random.int 4 with
152       | 0 when y > 0 ->
153           (x, y), Bottom
154       | 1 when y < height - 1 ->
155           (x, y), Top
156       | 2 when x > 0 ->
157           (x, y), Left
158       | 3 when x < width - 1 ->
159           (x, y), Right
160       | _ -> choose_door width height
161
162   open Graphics
163

```

```

164 let size = 14 and border = 10
165
166 let mark (x, y) color =
167     set_color color ;
168     fill_rect
169         (x * size + border + 3)
170         (y * size + border + 3)
171         (size - 6) (size - 6)
172
173 let draw_cell (x, y) doors =
174     set_line_width 3 ;
175     set_color black ;
176     if not (List.mem Bottom doors) then (
177         moveto (x * size + border) (y * size + border) ;
178         rlineto size 0
179     ) ;
180     if not (List.mem Left doors) then (
181         moveto (x * size + border) (y * size + border) ;
182         rlineto 0 size
183     ) ;
184     if not (List.mem Top doors) then (
185         moveto (x * size + border) ((y + 1) * size + border) ;
186         rlineto size 0
187     ) ;
188     if not (List.mem Right doors) then (
189         moveto ((x + 1) * size + border) (y * size + border) ;
190         rlineto 0 size
191     )
192
193 let window_size width height =
194     ((width * size + 2 * border),
195      (height * size + 2 * border))
196
197 end
198
199 module Beehive_shape = struct
200     type door = BottomLeft | Left | Right | TopRight | BottomRight | TopLeft
201
202     let all_doors =
203         [ BottomRight ; Left ; Right ; TopLeft ; BottomLeft ; TopRight ]
204
205     let neighbour (x, y) d =
206         match d, y mod 2 = 1 with
207         | TopLeft, false -> x - 1, y + 1
208         | TopRight, false -> x, y + 1
209         | BottomLeft, false -> x - 1, y - 1
210         | BottomRight, false -> x, y - 1
211         | TopLeft, true -> x, y + 1
212         | TopRight, true -> x + 1, y + 1
213         | BottomLeft, true -> x, y - 1
214         | BottomRight, true -> x + 1, y - 1
215         | Left, _ -> x - 1, y
216         | Right, _ -> x + 1, y

```



```

217
218 let opposite d =
219     match d with
220     | TopRight -> BottomLeft
221     | BottomRight -> TopLeft
222     | TopLeft -> BottomRight
223     | BottomLeft -> TopRight
224     | Left -> Right
225     | Right -> Left
226
227 let rec choose_door width height =
228     let y = Random.int height
229     and x = Random.int width in
230     match Random.int 6 with
231     | 0 when y > 0 && (x > 0 || y mod 2 = 1) ->
232         (x, y), BottomLeft
233     | 2 when y < height - 1 && (x > 0 || y mod 2 = 1) ->
234         (x, y), TopLeft
235     | 1 when y > 0 && (x < width - 1 || y mod 2 = 0) ->
236         (x, y), BottomRight
237     | 3 when y < height - 1 && (x < width - 1 || y mod 2 = 0) ->
238         (x, y), TopRight
239     | 4 when x > 0 ->
240         (x, y), Left
241     | 5 when x < width - 1 ->
242         (x, y), Right
243     | _ -> choose_door width height
244
245
246 open Graphics
247
248 let size = 14 and border = 10
249
250 let mark (x, y) color =
251     let sx = size / 2 in
252     let sy = 2 * size / 3 in
253     let xofs = (y mod 2) * sx in
254     set_color color ;
255     fill_circle
256         (x * size + xofs + sx + border)
257         (y * size + sy + border)
258         (size / 3)
259
260 let draw_cell (x, y) doors =
261     let sx = size / 2 in
262     let sy = size / 3 in
263     let xofs = (y mod 2) * sx in
264     set_line_width 3 ;
265     set_color black ;
266     if not (List.mem TopLeft doors) then (
267         moveto (x * size + xofs + sx + border) ((y + 1) * size + sy + border) ;
268         rlineto (-sx) (-sy)
269     ) ;

```

```

270   if not (List.mem TopRight doors) then (
271       moveto (x * size + xofs + sx + border) ((y + 1) * size + sy + border) ;
272       rlineto sx (-sy)
273   ) ;
274   if not (List.mem BottomLeft doors) then (
275       moveto (x * size + xofs + sx + border) (y * size + border) ;
276       rlineto (-sx) sy
277   ) ;
278   if not (List.mem BottomRight doors) then (
279       moveto (x * size + xofs + sx + border) (y * size + border) ;
280       rlineto sx sy
281   ) ;
282   if not (List.mem Left doors) then (
283       moveto (x * size + xofs + border) (y * size + sy + border) ;
284       rlineto 0 (2 * sy)
285   ) ;
286   if not (List.mem Right doors) then (
287       moveto ((x + 1) * size + xofs + border) (y * size + sy + border) ;
288       rlineto 0 (2 * sy)
289   )
290
291   let window_size width height =
292       ((width * size + size / 2 + 2 * border),
293        (height * size + size / 3 + 2 * border))
294
295 end
296
297 let () =
298     Random.self_init () ;
299     let module Square = Maze (Square_shape) in
300     Square.(solve_and_show (make_maze 30 30)) ;
301     let module Beehive = Maze (Beehive_shape) in
302     Beehive.(solve_and_show (make_maze 30 30))

```