Bu-Ali Sina University

Report

Solving Maze
(In Common Lisp)

Mehran Shahidi

supervised by
Pr. Bathaeian

July 16, 2020

Summary

In this Project we are going to implement Maze solver in Common Lisp programming language. For more information about codes visit this repository:

https://github.com/m3hransh/solve-maze

Contents

1 Defining Problem

In the Maze problem, our goal is to start from a point and reach some goal point bypassing obstacles through the way. To represent the maze map, we are using a grid map. Grid maps represent every point that agents can be by a square with some specific color and obstacles with another color.

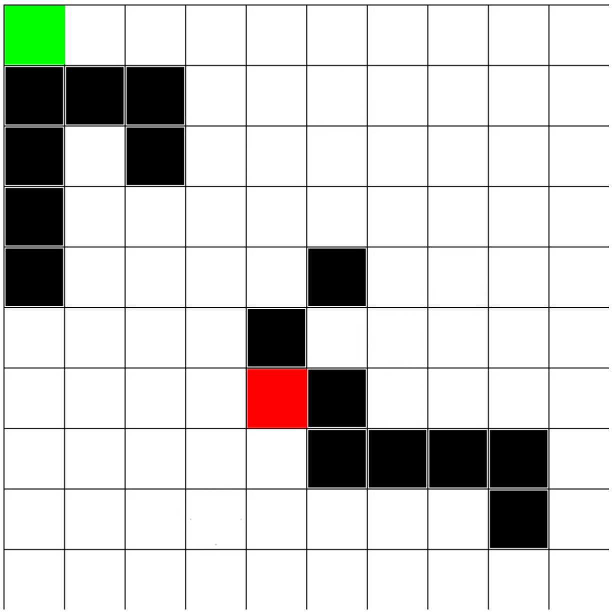

Figure 1: example of a grid map that agent need to start from green square and reach red one. black squares represent obstacles.

2 Implementing Maze Solver in Lisp

For the implementation, we are using **Common Lisp**. Common Lisp (CL) is a dialect of the Lisp programming language, published in ANSI standard. To implement the maze we need to represent our map. To do so, we will use a simple binary array in Lisp. The *t*s are points that agent can take as a path, and *Nil*s are obstacles. The code is shown in the Listing 1. *solve-maze* is the main function that use other helper functions. At first, it will check if the point that we are in is the end point or not and then try to go further by checking it's neighbors and solve them recursively. it also remembers the points it is visiting through the way by putting them in visiting list and directoins it took from the starting point in the plan list.

```
(defun list-member (x lst)
    (cond ((null lst) nil)
        ((equal x (car lst)) t)
        (t (list-member x (cdr lst))))
)
;; check if the index is in the boundary of the map
;; or is not a obstacle
;; or is not visited before
(defun accessible(index map visited)
    (cond ((or (< (car index) 0) (< (cdr index) 0)
                (>= (car index) (car (array-dimensions map)))
                (>= (cdr index) (car (cdr (array-dimensions map))))
                (null (aref map (car index) (cdr index)))
                (list-member index visited)
            ) nil)
            (t t))
)
;; has list the list of neighbors as parameter that will check them one by one
;; if one neighbor is accessible but has no path to end
;; then call check the next neighbor by adding visited points of previous neighbor
(defun check-neighbors(map neighbors end plan visited )
```

```lisp
    (cond ((null neighbors) (cons nil visited))
        ((accessible (car (car neighbors)) map visited )
            ((lambda (next)
                (cond ((null (car next)) (check-neighbors map (cdr neighbors) end plan (cdr next)))
                    (t next))
                ) (solve-maze map (car (car neighbors)) end (append plan (list (cdr (car neighbors)))) (append visited (list (car (car neighbors))))))
        )
        (t (check-neighbors map (cdr neighbors) end plan visited))
    )
)
;; main function for solving the maz
;; it takes map a 2-D array of the maz
;; current point that we are in at the moment
;; that at the start is starting point
;; and end that is the goal
;; plan is used to add the plan as we visit points
;; visited is a list of all visited point till now
(defun solve-maze (map start end plan visited)
    ;; check if the current point is end point
    (cond ((equal start end) plan)
    ;; checks all neighbors using check-neighbors function
        (t (check-neighbors map (list
                                (cons (cons (car start) (- (cdr start) 1)) 'L)
                                (cons (cons (- (car start) 1) (cdr start)) 'U)
                                (cons (cons (car start) (+ (cdr start) 1)) 'R)
                                (cons (cons (+ (car start) 1) (cdr start)) 'D)
                                ) end plan visited)

        )
    )
)
;; map:
;; 0 0 0 0 0
;; 1 1 0 1 0
;; 1 0 0 1 0
;; 0 0 1 0 0
;; map represented as true and false (cons 0 0) is the start
;; (cons 3 0) is the end
;; the plan has S element at first
;; the visited list has the start point at the beginning
(format t "map:~%0 0 0 0 0 ~%1 1 0 1 0 ~%1 0 0 1 0 ~%0 0 1 0 0 ~%~%start :(0 0)~%end (3 0)~%~%")
(write (solve-maze  (make-array '(4 5)
    :initial-contents '((t t t t t)
                        (nil nil t nil t)
                        (nil t t nil t)
                        (t t nil t t))) (cons 0 0) (cons 3 0) (list 'S) (list (cons 0 0))))
```

Listing 1: Getting map from a file and represent it as 2D-array

Figure 2 shows an example, how we can define our file content.

$$
\begin{array}{ccccc}
0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 0 \\
1 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 1 & 0
\end{array}
$$

Figure 2: Example of file content

2.1 Running the program

Listing 2 is the execution of the program. Last line shows the plan that is needed to take to solve the maze.

Listing 2: Exectuion of the program

```
map:
0 0 0 0 0
1 1 0 1 0
1 0 0 1 0
0 0 1 0 0

start :(0 0)
end (3 0)

(S R R D D L D L)
```