Bu-Ali Sina University

Report

Solving Maze
(In Common Lisp)

Mehran Shahidi

supervised by
Pr. Bathaeian

July 12, 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.

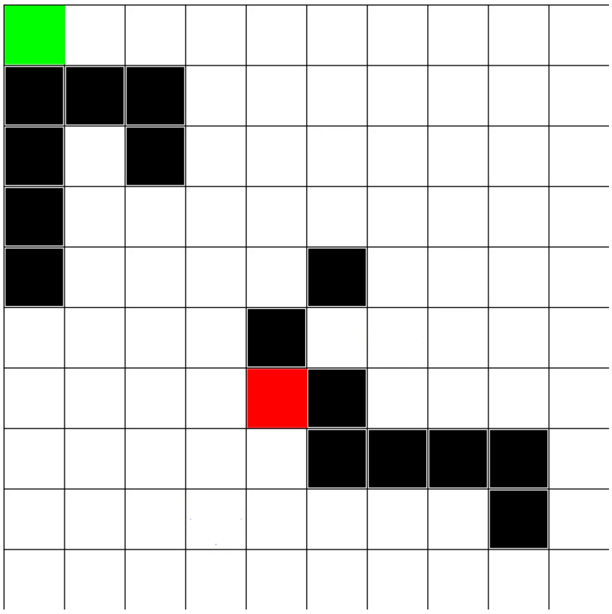

Figure 1: example of a grid map that agent need to start from green square and reach red one. black squares represent obstacles.

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.

2.1 Map representation

For map representation, a simple binary matrix is enough. The 0s are points that agents can take as a path, and 1s are obstacles. For convenience, the program in Listing 1 is written to get the map from a file and change it to a 2D-array.

```lisp
(defun delimiterp (c) (or (char= c #\Space) (char= c #\,)))
(defun my-split (string &key (delimiterp #'delimiterp))
  (loop :for beg = (position-if-not delimiterp string)
    :then (position-if-not delimiterp string :start (1+ end))
    :for end = (and beg (position-if delimiterp string :start beg))
    :when beg :collect (subseq string beg end)
    :while end))
;; *maze* variable for storing our map az matrix
(setq *maze* nil)
;;define start and end points here

;; reading from the fiel maze.txt
(let ((in (open "maze.txt" :if-does-not-exist nil)))
   (when in
      (loop for line = (read-line in nil)

      while line do (push  (mapcar #'parse-integer (my-split line)) *maze*))
      (close in))
   )
)
;; function for converting 2d list to 2d-array
(defun list-to-2d-array (list)
```

```
  (make-array (list (length list)
                    (length (first list)))
              :initial-contents list))
;; shape size of the maze
(defconstant N (length *maze*))
(defconstant M (length (first *maze*)))
;; reverse tge *maze* and convert to the array
(setf *maze* (list-to-2d-array (reverse *maze*)))
```

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.

$$\begin{matrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \end{matrix}$$

Figure 2: Example of file content

2.2 Getting accessible neighbours

Listing 2 shows a implementation of a helper function to get the accessible neighbours from the index that is represented as a cons cell (x.y). The function also takes a list path to check if the neighbour is visited before or not. If the neighbour grid is in the boundry of map (*map* is a global variable), is not a obstacle and is not visted before, it will be added to the list of the accessible neighbours. And it will be returned by the function.

```
(defun check-path(x y)
    (cond ((atom y) nil)
    ((equal x (caar y)) x)
    (t (check-path x (cdr y)))
    )
)


;; helper function for finding adjacency grid that we can go from index
;; with respect to the path that we've visted
(defun get-neighbors(index path)
    (let ((neighbors nil))
    ;; check up
    (let (( up (cons (- (car index) 1) (cdr index)) ))
        (if (and (>= (car up) 0)                        ; if the index is in map
            (equal (aref *maze* (car up) (cdr up)) 0); there is no obstacle
            (null (check-path up path))                      ; it's not visited before
        )
          (push (cons up 'U) neighbors)
        )
    )
    ;; check right
    (let (( right  (cons (car index) (+ (cdr index) 1)) ))
        (if (and (< (cdr right) M)
            (equal (aref *maze* (car right) (cdr right)) 0)
            (null (check-path right path))
        )
        (push (cons right 'R) neighbors)
        )
    )
    ;; check below
    (let (( down  (cons (+ (car index) 1) (cdr index)) ))
        (if (and (<  (car down)  N)
            (equal (aref *maze* (car down) (cdr down)) 0)
            (null (check-path down path))
        )
        (push (cons down 'D) neighbors)
        )
    )
    ;; check left
```

```
    (let (( left (cons (car index) (- (cdr index) 1))))
        (if (and (>= (cdr left) 0)
            (equal (aref *maze* (car left) (cdr left)) 0)
            (null (check-path left path))
        )
          (push (cons left 'L) neighbors)
        )
    )
    (return-from get-neighbors neighbors);return all possible neighbor accessible from index
    )
)
```

Listing 2: Finding accessible neighbours from a specific point

2.3 Solving the Maze

Now, let's consider our main function for solving the maze . This function takes starting-point, end-point , and the path list that we visisted till now. Starting point shows the grid that we are currently in. The solve-maze is a recursive function. First it checks if the current grid is goal or not, then if is not, use the search-neighbors function, that in turn run solve-maze on each neighbor that is accessible from the current grid and by putting those neighbors as the current grid solve the maze recursively. The function search-neighbors will return nil if none of the neighbors has a path to the goal.

```
; run solve-maze function on others neighbors to find path to the goal point
(defun search-neighbors(neighbors start end path solve-maze)
        (cond ((null (car neighbors)) nil) ;if there is no neighbor it returns nil
            (t ((lambda (x) ; to check if the first neighbor give the answer if not search other neighbors
                    (cond ((null x) (search-neighbors (cdr neighbors) start end path solve-maze))
                        (t x) )
                ;; calling the solve-maze with start point of the neighbor
                )(solve-maze (caar neighbors) end
                    (append path (list (car neighbors)))
                    ; and new path that consist of previous
                 )  ; start and it's direction to the new start
                )
                )
        )
)

; print the plan by taking the path as input
(defun plan-for-path(x)
    (cond ((null x) nil)
        ;; creating a list of directions of the path elements
        (t (cons (cdar x) (plan-for-path (cdr x)) )

        )
)
)

(defun solve-maze(start end path)
    (cond ((equal start end) (plan-for-path path))
            (t (search-neighbors (get-neighbors start path) start end path #' solve-maze)
            )
    )
)
```

Listing 3: main function for solving the maze

2.4 Running the program

Now that the main function is completed we can run our program on some inputs. The code in Listing 4 shows how with the help of the **solve-maze** and map, starting-point and end-point we can write our final part of our program.

```lisp
    ;; printing the map
(format t "Map:~%")
(dotimes (x 4)
    (dotimes(y 5)
        (format t "~a " (aref *maze* x y))
    )
    (format t "~%")
)
(terpri)
(format t "start:~a ~%" (cons 0 0))
(format t "Goal:~a ~%" (cons 3 0))
(terpri)
;; solve the maze with starting point (0 0) and end point of (3 0)
(format t "Plan to find the goal:~% ~a" (solve-maze  (cons 0 0) (cons 3 0) (list (
    cons (cons 0 0) 'S)) ))
```

Listing 4: final part of program

Finally, you can see an example of the exection of the program in the Listing 6.

Listing 5: Exectuion of the program

```
$ cat maze.txt
0 0 0 0 0
1 1 0 1 0
1 0 0 1 0
0 0 1 1 0
$ clisp maze_solver.lisp
Map:
0 0 0 0 0
1 1 0 1 0
1 0 0 1 0
0 0 1 1 0

start:(0 . 0)
Goal:(3 . 0)

Plan to find the goal:
 (S R R D D L D L)
```

Listing 6: Exectuion of the program on a map with no existing path to end

```
$ cat maze.txt
0 0 0 0 0
1 1 1 1 0
1 0 0 1 0
0 0 1 1 0
$ clisp maze_solver.lisp
Map:
0 0 0 0 0
1 1 1 1 0
1 0 0 1 0
0 0 1 1 0

start:(0 . 0)
Goal:(3 . 0)

Plan to find the goal:
 Nil
```