Bu-Ali Sina University

Report

Solving Maze
(In Common Lisp)

Mehran Shahidi

supervised by
Pr. Bathaeian

July 1, 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.

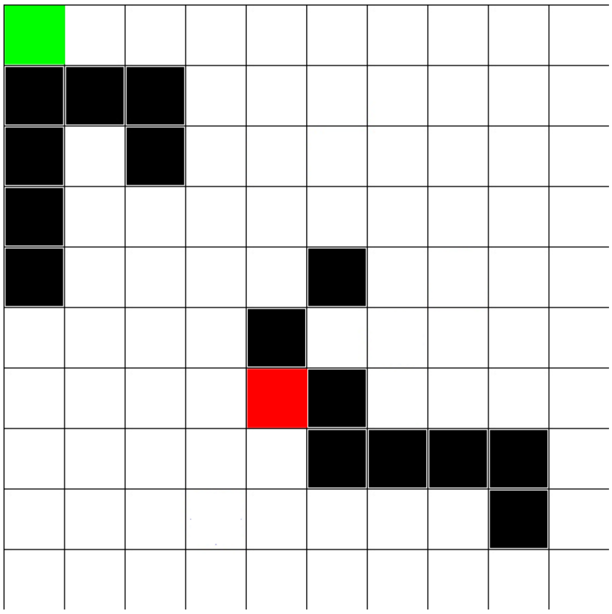

Figure 1: example of a grid map that agent need to start from green square and reach red one. black squares represent obstacles.

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. Take note, the global variables start and end represent our starting and end points.

```
(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
(setq *start* '(0 0))
(setq *end* '(3 0))
;; 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.

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

Figure 2: Example of file content

2.2 Getting accessible neighbours

Listing 2 shows a implementation of a helper function to get the accessible neighbours from a point that is represents by a list (x y). The function also takes a hash-table path to check if the neighbour is visited before or not. If the neighbour grid is in the boundry of map , 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.

```
; 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 (list (- (nth 0 index) 1) (nth 1 index)) ))
        (if (and (>= (nth 0 up) 0)                        ; if the index is in map
            (equal (aref *maze* (nth 0 up) (nth 1 up)) 0); there is no obstacle
            (null (gethash up path))                      ; it's not visited before
        )
          (push (list up 'U) neighbors)
        )
    )
    ;; check right
    (let (( right  (list (nth 0 index) (+ (nth 1 index) 1)) ))
        (if (and (< (nth 1 right) M)
            (equal (aref *maze* (nth 0 right) (nth 1 right)) 0)
            (null (gethash right path))
        )
        (push (list right 'R) neighbors)
        )
    )
    ;; check below
    (let (( down  (list (+ (nth 0 index) 1) (nth 1 index)) ))
        (if (and (<  (nth 0 down)  N)
            (equal (aref *maze* (nth 0 down) (nth 1 down)) 0)
            (null (gethash down path))
        )
        (push (list down 'D) neighbors)
        )
    )
    ;; check left
    (let (( left (list (nth 0 index) (- (nth 1 index) 1))))
        (if (and (>= (nth 1 left) 0)
            (equal (aref *maze* (nth 0 left) (nth 1 left)) 0)
            (null (gethash left path))
        )
          (push (list 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, we reach our main function for solving the maze. This function takes the map, its starting-point, and end-point. With the help of the **get-neighbors** function, it will solve the maze by returning the plan. The plan consists of the direction that the agent needs to take at each point in the path to the goal.

```
;; solve the maze by getting 2d-array as map
;; and start as (x y) that specifies the staring point
;; and end for end point
(defun solve-maze(maze start end)
    ;; path hash-table for stroing the path we take to reach to each grid
    ;; path[p1] = (p0 L) says that we reach to p1 from p0 by going to the left
    (setq path (make-hash-table :test 'equal) )
    ;; stack for visiting grid recursively
    (setq stack nil)
    (push start stack)
    ;; start
    (setf (gethash start path) (list -1 -1))
    (let ((p
    (loop
        ;; taking the first element to expand its accessible neighbours
        (setq el  (car stack))
        (setf stack (cdr stack))
        ;; the loop finish when the el is the end point
        ;; or the stack is empty and that means there is no path to goal
        (when (or (null el) (equal el end)) (return path))
        ;; geting el's neighbors
        (let ((ns (get-neighbors el path) ))
            (if (not (equal ns nil))
                (dolist (neighbor ns)
                (progn
                    ;; ading the neighbor to the stack
                    ;; and to the path
                    (push (car neighbor) stack)
                    (setf (gethash (car neighbor) path) (list el (cadr neighbor)))
                )
                )

            )

        )
    ) ))
    ;; if there is a path to end point
    ;; it we'll print the plan
    (if (gethash end path)
        (progn
            (setq  temp end)
            (setq plan nil)
            (loop (when (equal temp start) (return  plan))
                (push (cadr (gethash temp path)) plan)

                (setf temp (car (gethash temp path)))
            )
        )
        (format t "no path is found!~%")
    )
    )
)
```

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.

```
        ;; 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 ~%" *start*)
(format t "Goal:~a ~%" *end*)
(terpri)
;; solve the with starting point (0 0) and end point of (3 0)
(format t "Plan to find the goal:~% ~a" (solve-maze *maze* *start* *end*))
```

Listing 4: final part of program

Finally, you can see an example of the exection of the program in the Listing 5.

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:
 (R R D D L D L)
```