

Tzewen Wang
ECE578
Winter 2005

Homework #2

Source code with description

```
;/*  
; * Homework 2  
; * Tzewen Wang  
; * ECE578  
; * Winter 2005  
; *  
; * In addition to the symbols for obstacle and corridor in a labyrinth, five more  
; * symbols are used and described including those as the following:  
; * 0: corridor.  
; * 1: obstacle.  
; * 2: visited corridor.  
; * 3: robot - facing north.  
; * 4: robot - facing east.  
; * 5: robot - facing south.  
; * 6: robot - facing west.  
; *  
; * The robot in this program will try to do learning first and memorize every  
; * single turn in the final path. The algorithm assumes the learning phase is  
; * done on a high-speed computer. Once the path is found, the robot performs  
; * actual movements and makes turns at particular location given by the path.  
; * As a result, the data structure for the path is the following:  
; *  
; * ( ( X Y DIR) ( X Y DIR) .... )  
; *  
; * This project uses a simple recursive algorithm in the learning phase. Each  
; * step is a recursive call. The termination occurs on either a door or a  
; * a dead-end. A door is the outmost (edge of) labyrinth. The labyrinth  
; * is constructed with a 2D array. If a corridor appears on the edge of the  
; * labyrinth, it is concerned as a door. For example, for a 10*10 labyrinth,  
; * a corridor is at (9, 5), then (9, 5) is the door. This should work fine  
; * with multiple doors. The direction precedence to move to the next step is  
; * is the following: i) forward, ii) right, iii)left, and iv) reverse. The  
; * algorithm returns TRUE if a door is found. It returns NIL if a a dead-end  
; * is found. The backtracking is handled by language mechanism. Thus, the  
; * backtracking will not be shown explicitly during the learning process. As  
; * the robot makes turn, the coordinate and direction will be pushed into  
; * PathStack. The stack will be popped when a backtracking occurs. This  
; * algorithm operates directly on two data structure: i) a 2D array  
; * representing the labyrinth, and ii) a list of list consisting of coordinates  
; * and facing directions.  
; *  
; * One can image the execution tree is to be a quadnary tree, a node with four  
; * branches. The algorithm traverses down to the leaf where it is a "door".  
; * As traversing going, the PathStack will be expanded and shrunk. Once the door  
; * is found, the execution of the program is ended. Then, the robot is guided  
; * by the coordinates in the PathStack.  
; *  
; * An additional function is implemented to do pretty-printing. It just converts  
; * those symbols into a human readable notations.  
; *  
; * The shortest path can be obtained in the learning process with a little  
; * modification of the algorithm. As mentioned earlier, the basic idea of the  
; * algorithm is to find a door leaf on the big execution-tree. Now, with modified  
; * termination condition, the entire tree is possibly traversed. The modification  
; * is as follows: if a door is found, we put the stack aside and return a NIL  
; * instead of TRUE. In addition, a weight should be assigned with each coordinate  
; * recorded in the PathStack. The weight is the distance from start point to  
; * current location. Once the learning process is done, a number of paths  
; * will be returned and the robot can pick the lightest one.  
;*/
```

```

/**
; * learn-labyrinth (L X Y DIR)
; *
; * Walk through a labyrinth (learning phase).
; *
; * Parameters:
; *   L - labyrinth in a 2-D array.
; *   X - current position X.
; *   Y - current position Y.
; *   DIR - robot's facing directions.
; *
; * Returns T if the door is found.
; *       NIL, otherwise
; */
(defun learn-labyrinth (L X Y DIR)
  (dump-labyrinth L)
  (cond
    ; when reaching the door...
    ((or (equal X 0) (equal X (- (array-dimension L 1) 1))
         (equal Y 0) (equal Y (- (array-dimension L 0) 1))))
    T)
    ; when facing North...
    ((equal DIR 3)
     (let ((FLAG nil))
       (cond ((equal (aref L (- Y 1) X) 0)
              (setf (aref L Y X) 2)
                  (setf (aref L (- Y 1) X) 3)
                  (setf FLAG (learn-labyrinth L X (- Y 1) 3))))
             (cond ((and (null FLAG) (equal (aref L Y (+ X 1)) 0))
                    (setf (aref L Y X) 2)
                        (setf (aref L Y (+ X 1)) 4)
                        (pusht X Y 4)
                        (setf FLAG (learn-labyrinth L (+ X 1) Y 4))
                        (cond ((null FLAG) (popt))))))
             (cond ((and (null FLAG) (equal (aref L Y (- X 1)) 0))
                    (setf (aref L Y X) 2)
                        (setf (aref L Y (- X 1)) 6)
                        (pusht X Y 6)
                        (setf FLAG (learn-labyrinth L (- X 1) Y 6))
                        (cond ((null FLAG) (popt))))))
             (cond ((and (null FLAG) (equal (aref L (+ Y 1) X) 0))
                    (setf (aref L Y X) 2)
                        (setf (aref L (+ Y 1) X) 5)
                        (pusht X Y 5)
                        (setf FLAG (learn-labyrinth L X (+ Y 1) 5))
                        (cond ((null FLAG) (popt))))))
              (setf (aref L Y X) 2)
                  FLAG)
       )
     )
    ; when facing East...
    ((equal DIR 4)
     (let ((FLAG nil))
       (cond ((equal (aref L Y (+ X 1)) 0)
              (setf (aref L Y X) 2)
                  (setf (aref L Y (+ X 1)) 4)
                  (setf FLAG (learn-labyrinth L (+ X 1) Y 4))))
             (cond ((and (null FLAG) (equal (aref L (+ Y 1) X) 0))
                    (setf (aref L Y X) 2)
                        (setf (aref L (+ Y 1) X) 5)
                        (pusht X Y 5)
                        (setf FLAG (learn-labyrinth L X (+ Y 1) 5))
                        (cond ((null FLAG) (popt))))))
             (cond ((and (null FLAG) (equal (aref L (- Y 1) X) 0))
                    (setf (aref L Y X) 2)
                        (setf (aref L (- Y 1) X) 3)

```

```

        (pusht X Y 3)
        (setf FLAG (learn-labyrinth L X (- Y 1) 3))
        (cond ((null FLAG) (popt))))))
(cond ((and (null FLAG) (equal (aref L Y (- X 1)) 0))
      (setf (aref L Y X) 2)
      (setf (aref L Y (- X 1)) 6)
      (pusht X Y 6)
      (setf FLAG (learn-labyrinth L (- X 1) Y 6))
      (cond ((null FLAG) (popt))))))
(setf (aref L Y X) 2)
FLAG
)
)
; when facing South...
((equal DIR 5)
 (let ((FLAG nil))
   (cond ((equal (aref L (+ Y 1) X) 0)
         (setf (aref L Y X) 2)
         (setf (aref L (+ Y 1) X) 5)
         (setf FLAG (learn-labyrinth L X (+ Y 1) 5))))
   (cond ((and (null FLAG) (equal (aref L Y (- X 1)) 0))
         (setf (aref L Y X) 2)
         (setf (aref L Y (- X 1)) 6)
         (pusht X Y 6)
         (setf FLAG (learn-labyrinth L (- X 1) Y 6))
         (cond ((null FLAG) (popt))))))
   (cond ((and (null FLAG) (equal (aref L Y (+ X 1)) 0))
         (setf (aref L Y X) 2)
         (setf (aref L Y (+ X 1)) 4)
         (pusht X Y 4)
         (setf FLAG (learn-labyrinth L (+ X 1) Y 4))
         (cond ((null FLAG) (popt))))))
   (cond ((and (null FLAG) (equal (aref L (- Y 1) X) 0))
         (setf (aref L Y X) 2)
         (setf (aref L (- Y 1) X) 3)
         (pusht X Y 3)
         (setf FLAG (learn-labyrinth L X (- Y 1) 3))
         (cond ((null FLAG) (popt))))))
   (setf (aref L Y X) 2)
   FLAG
  )
 )
; when facing West...
((equal DIR 6)
 (let ((FLAG nil))
   (cond ((equal (aref L Y (- X 1)) 0)
         (setf (aref L Y X) 2)
         (setf (aref L Y (- X 1)) 6)
         (setf FLAG (learn-labyrinth L (- X 1) Y 6))))
   (cond ((and (null FLAG) (equal (aref L (- Y 1) X) 0))
         (setf (aref L Y X) 2)
         (setf (aref L (- Y 1) X) 3)
         (pusht X Y 3)
         (setf FLAG (learn-labyrinth L X (- Y 1) 3))
         (cond ((null FLAG) (popt))))))
   (cond ((and (null FLAG) (equal (aref L (+ Y 1) X) 0))
         (setf (aref L Y X) 2)
         (setf (aref L (+ Y 1) X) 5)
         (pusht X Y 5)
         (setf FLAG (learn-labyrinth L X (+ Y 1) 5))
         (cond ((null FLAG) (popt))))))
   (cond ((and (null FLAG) (equal (aref L Y (+ X 1)) 0))
         (setf (aref L Y X) 2)
         (setf (aref L Y (+ X 1)) 4)
         (pusht X Y 4)
         (setf FLAG (learn-labyrinth L (+ X 1) Y 4))
         (cond ((null FLAG) (popt))))))
   (setf (aref L Y X) 2)
  )
 )

```

```

        FLAG
      )
    )
  )
)

;/**
; * walk-labyrinth (L X Y DIR)
; *
; * Walks through a labyrinth by a given path.
; * (This is not a argument-safe nor path-safe funciton).
; *
; * Parameters:
; * L - the labyrinth in a 2D array.
; * X - current position X.
; * Y - current position Y.
; * DIR - robot's facing direction.
; * PATH - learned path.
; *
; * Returns T if success.
;*/
(defun walk-labyrinth (L X Y DIR PATH)
  (dump-labyrinth L)
  (cond
    ; when reaching the door...
    ((or (equal X 0) (equal X (- (array-dimension L 1) 1))
         (equal Y 0) (equal Y (- (array-dimension L 0) 1)))
     T)
    ; when reaching a turn point...
    ((and (not (null PATH)) (equal (first (first PATH)) X) (equal (second (first PATH))
Y))
     (setf (aref L Y X) (third (first PATH)))
     (walk-labyrinth L X Y (third (first PATH)) (rest PATH)))
    ; when facing North...
    ((equal DIR 3)
     (setf (aref L (- Y 1) X) 3)
     (setf (aref L Y X) 2)
     (walk-labyrinth L X (- Y 1) 3 PATH))
    ; when facing East...
    ((equal DIR 4)
     (setf (aref L Y (+ X 1)) 4)
     (setf (aref L Y X) 2)
     (walk-labyrinth L (+ X 1) Y 4 PATH))
    ; when facing South...
    ((equal DIR 5)
     (setf (aref L (+ Y 1) X ) 5)
     (setf (aref L Y X) 2)
     (walk-labyrinth L X (+ Y 1) 5 PATH))
    ; when facing West...
    ((equal DIR 6)
     (setf (aref L Y (- X 1)) 6)
     (setf (aref L Y X) 2)
     (walk-labyrinth L (- X 1) Y 6 PATH))
    (T (print "ERROR in walk-labyrinth!"))
  )
)

;/**
; * Dumps a labyrinth in human readable notations.
; * (This is not a argument-safe function).
; *
; * L - the labyrinth to be dumped.
; *
; * Returns T if success.

```

```

;*/
(defun dump-labyrinth (L)
  (dotimes (Y (array-dimension L 0))
    (dotimes (X (array-dimension L 1))
      (write-char (cond ((equal (aref L Y X) 0) #\space)
                       ((equal (aref L Y X) 1) #\*)
                       ((equal (aref L Y X) 2) #\.)
                       ((equal (aref L Y X) 3) #\^)
                       ((equal (aref L Y X) 4) #\>)
                       ((equal (aref L Y X) 5) #\|)
                       ((equal (aref L Y X) 6) #\<))
                )
            )
      (write-char #\newline)
    )
  (write-char #\newline)
)

;/**
; * pusht (X Y DIR)
; *
; * Push a turn into path list.
; *
; * Parameters:
; * X, Y - coordinates.
; * DIR - robot's facing direction.
; *
; * Returns a list of coordinates on the stack.
;*/
(defun pusht (X Y DIR)
  (cond ((atom PATH) (setf PATH (list (list X Y DIR))))
        ((and (equal X (first (first (reverse PATH))))
              (equal Y (first (second (reverse PATH))))
              (popt))
         (setf PATH (append PATH (list (list X Y DIR)))))
        (T (setf PATH (append PATH (list (list X Y DIR)))))
  )
)

;/**
; * pop ()
; *
; * Pop a turn into path list.
; *
; * Parameters:
; * X, Y - coordinates.
; * DIR - robot's facing direction.
; *
; * Returns a list of coordinates on the stack.
;*/
(defun pop ()
  (cond ((equal (length PATH) 1) (setf nil))
        (T (setf PATH (reverse (rest (reverse PATH)))))
  )
)

;/**
; * copy (S)
; *
; * Clone an arbitrary structure.
; *
; * Parameters:

```

```

; * S - The structure to be replicated.
; *
; * Returns a clone copy of the structure.
;*/
(defun copy (S)
  (cond ((atom S) S)
        (T (cons (copy (car S)) (copy (cdr S))))
  )
)

;/**
; * Main program.
; * Find a way to get out of the labyrinth.
;*/
(defun main ()
  (setf PATH nil)
  (setf LAB (make-array `(10 10)
    :initial-contents
    `((1 1 1 1 1 1 1 1 1 1)
      (1 0 0 0 0 0 0 0 0 1)
      (1 0 1 0 1 0 0 1 0 1)
      (1 0 1 0 1 1 1 1 0 1)
      (1 5 1 0 0 0 0 0 0 1)
      (1 0 1 0 1 1 1 1 0 0)
      (1 0 1 0 0 0 0 0 0 1)
      (1 0 1 0 1 1 1 1 0 1)
      (1 0 1 0 0 0 0 0 0 1)
      (1 1 1 1 1 1 1 1 1 1))))
  (write-char #\newline)
  (print "Learning phase")
  (write-char #\newline)
  (learn-labyrinth (copy LAB) 1 4 5)
  ; In case of COPY does not work. A manual copy instead.
  (setf LAB (make-array `(10 10)
    :initial-contents
    `((1 1 1 1 1 1 1 1 1 1)
      (1 0 0 0 0 0 0 0 0 1)
      (1 0 1 0 1 0 0 1 0 1)
      (1 0 1 0 1 1 1 1 0 1)
      (1 5 1 0 0 0 0 0 0 1)
      (1 0 1 0 1 1 1 1 0 0)
      (1 0 1 0 0 0 0 0 0 1)
      (1 0 1 0 1 1 1 1 0 1)
      (1 0 1 0 0 0 0 0 0 1)
      (1 1 1 1 1 1 1 1 1 1))))
  (write-char #\newline)
  (print "Solving phase")
  (write-char #\newline)
  (walk-labyrinth LAB 1 4 5 PATH)
)

; Main program invocation.
(main)

```

Program output

(next page)


```
*****
*.....*
*.*.*.*
*.*****|*
*.*      *
* * ****
* *      *
* * ****
* *      *
* *      *
*****
```

```
*****
*.....*
*.*.*.*
*.*****.*
*.*      |*
* * ****
* *      *
* * ****
* *      *
* *      *
*****
```

```
*****
*.....*
*.*.*.*
*.*****.*
*.*      .*
*.*      |*
* * ****
* *      *
* * ****
* *      *
* *      *
*****
```

```
*****
*.....*
*.*.*.*
*.*****.*
*.*      .*
*.*      .*
* * ****>*
* *      *
* * ****
* *      *
* *      *
*****
```

```
*****
*.....*
*.*.*.*
*.*****.*
*.*      .*
*.*      .*
* * ****>*
* *      *
* * ****
* *      *
* *      *
*****
```