

CSC324 Tutorial, Jan 28 2004

A1 discussion

Important Note: there have been some minor typos in the handout please go to the website (click refresh bottom) and re-download the handout.

- Question?

Solving a maze:

Given an $m \times n$ maze, return the list of all *unique paths* from the start point to the end point.

Rules:

- A path must begin with the given *starting* point and end at the given *end* point.
- A path cannot pass through a solid block!
- a path cannot cross itself, i.e. no block in the path is visited more than once.
- The list of paths you return must be unique, i.e. each path is mentioned once!

Example:

In the following maze, the start point is the block $(4,2)$ and the end point is the block $(0,2)$, and we have shown solid blocks with an **X**.

	0	1	2	3	4	5
0		X	e			
1				X		X
2		X		X		
3						
4	X		s	X		

Note: the top left block is addressed $(0,0)$, and block (x,y) corresponds to row x and column y .

What are the valid paths from $(4,2)$ to $(0,2)$?

Representing a maze as a nested list in scheme:

	0	1	2	3	4	5
0		X	e			
1				X		X
2		X		X		
3						
4	X		s	X		

```
(define maze1
  '( (0 1 0 0 0 0) ;this is row 0
      (0 0 0 1 0 1) ;this is row 1
      (0 1 0 1 0 0) ;this is row 2
      (0 0 0 0 0 0) ;this is row 3
      (1 0 0 1 0 0) ;this is row 4
    )
)
```

Note: start and end point are not specified in the maze itself! We specify them when calling the maze-solver as follows:

```
(maze-solver maze1 '(4 2) '(0 2))
```

Important Note: your code will be auto-marked. So, make sure to define the procedure exactly as specified. So, you must define the procedure **maze-solver** that accepts 3 arguments as follows:

1. a maze (i.e. a list of list of 0/1),
2. a start point (i.e. a list of two numbers specifying the coordinates of *start*),
3. an end point (i.e. a list of two numbers specifying the coordinates of *end*).

Assume (as *preconditions*) that the input arguments are meaningful and correct. So, you DONOT need to check the following conditions in your program:

- the maze is a valid maze with at least one square (it exactly specifies an $m \times n$ matrix filled with 0 and 1)
- the *start* and *end* coordinates are valid and within range (i.e. $0 \leq x \leq m-1$ and $0 \leq y \leq n-1$). And their corresponding block is **free** (i.e. 0).
- The *start* and *end* coordinates are not the same.

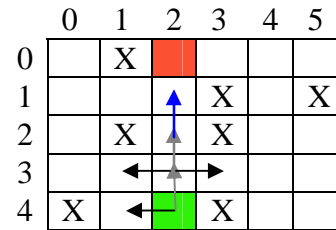
Now, let's try to solve this maze. We start at (4,2), we cannot go DOWN or RIGHT, why?. We can only go UP or LEFT.

	0	1	2	3	4	5
0		X				
1				X		X
2		X		X		
3						
4	X			X		

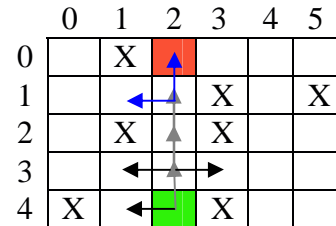
Let's try UP first. Now, we are at (3,2). We can go UP, LEFT, RIGHT but not DOWN (why?)

	0	1	2	3	4	5
0		X				
1				X		X
2		X		X		
3						
4	X			X		

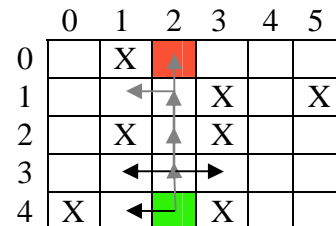
Let's try UP first. So, we are at (2,2) now. We can only go UP (why?):



So, now we are at (1,2). We can go only LEFT and UP:



Let's try UP first. There you go! We got to the end (the end point was (0,2)). So, we return this path. What about the other paths. We have to go back to the **last point** and try other possibilities from there. The last time, we were at (1,2) and tried UP, we go back there and try LEFT this time:



So, we are at (1,1). If we continue from here, you can see there would be no valid path to (0,2). Note that you cannot visit a square more than once!!

So, let's go back. What was our last choice? Oh, we were at (2,2) and tried UP. But that was the only choice. So, let's go back. We were at (3,2) and we tried UP. Ok, we haven't tried LEFT and RIGHT yet. Let's try RIGHT:

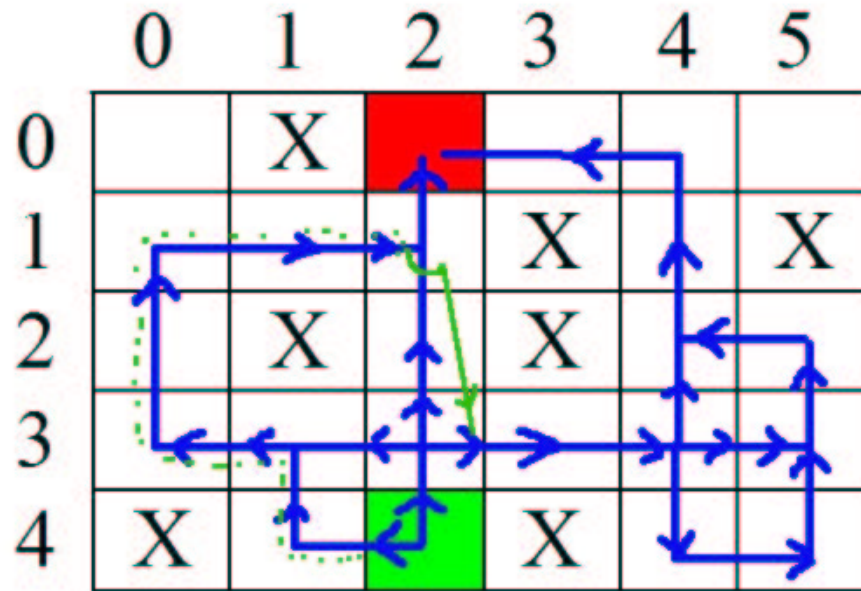
	0	1	2	3	4	5
0		X				
1				X		X
2		X	↑	X		
3		←	↑	→		
4	X	←		X		

So, we are at (3,3) and we can only go RIGHT to (3,4). Let's go there. There, we have three choices, UP,RIGHT, DOWN:

	0	1	2	3	4	5
0		X				
1				X		X
2		X	↑	X	↑	
3		←	↑	→	→	→
4	X	←		X	↑	↓

And so on.....

Here is the complete trial:



(((4 2) (3 2) (2 2) (1 2) (0 2))
 ((4 2) (3 2) (3 3) (3 4) (2 4) (1 4) (0 4) (0 3) (0 2))
 ((4 2) (3 2) (3 3) (3 4) (3 5) (2 5) (2 4) (1 4) (0 4) (0 3) (0 2))
 ((4 2) (3 2) (3 3) (3 4) (4 4) (4 5) (3 5) (2 5) (2 4) (1 4) (0 4) (0 3) (0 2))
 ((4 2) (3 2) (3 1) (3 0) (2 0) (1 0) (1 1) (1 2) (0 2))
 ((4 2) (4 1) (3 1) (3 0) (2 0) (1 0) (1 1) (1 2) (0 2))
 ((4 2) (4 1) (3 1) (3 0) (2 0) (1 0) (1 1) (1 2) (2 2) (3 2) (3 3))
)

It looks scary! How do we implement this in scheme!

Do not panic. It's much easier than you think! Start with writing simple small procedures like the followings:

- Given a maze what are its dimensions?
- Is a move valid? What do you need for this? The maze the current position, **the current path** (to avoid visiting a square twice). Watch out for **boundaries!**
- If a move is valid then you need to try it (going to the new position updating the path ...)
-
-

OK, but how do we **search** all these paths without any loop?

Think recursively! It's much much harder to implement this without recursion!!

- Assume you are at a point and have a partial path from start point to there what are all the possible ways we can continue from here to the end?
- What are the base cases?

If you think this way, believe me, the search can be implemented in just a couple of lines!

Special note:

- For electronic submission only submit **one file** called **a1.scm** (all letters in lower-case). If the auto-marker cannot file this file your correctness mark will be 0!
- The auto-marker will load this file and will call the procedure **maze-solver** with 3 parameters as specified. We could have our own mazes with different start and end points.
- Do not print any extra information. Your procedure just returns a list of unique paths (if none exists it returns empty list).

Question?

Let's write a simple *scheme* procedure that given a position and a maze, returns the content of the square at that position (remember coordinates start at $(0,0)$).

Idea: if the x coordinate is 0, it means we don't need to consider the whole maze, we only need to look at the row 0 and find its y^{th} element. But if $x > 0$ then we can discard the first row, and look for position $((x-1),y)$ in the maze without the first row. So, we have:

; returns the value in the maze corresponding to position *pos*.

; Precondition: *pos* is a valid position in the maze.

```
(define (getMazeVal maze pos)
  (if (zero? (car pos)) ;if pos in the row 0. This is the base-case.
      (getItem (car maze) (cadr pos)) ;base-case: calling helper function getItem
      (getMazeVal (cdr maze) (list (- (car pos) 1) (cadr pos)))) ;else: recursive call
  )
)
```

Where *getItem* is a helper function that given a list and a number i , returns the i th element of the list (note that index starts at 0). For example `(getItem '(0 9 -5 2 7 3) 2)` returns -5.

- **A Simple exercise:** write the *getItem* procedure.
- In our implementation We didn't need to worry if *pos* was out of range, because the precondition says *pos* must be in the range. What happens if someone calls *getMazeVal* with a position out of range, or even a negative position like $(-10, 3)$?

- Using built-ins *let* and *let** make your program simpler and more readable (and sometimes more efficient). You are strongly encouraged to learn them and use them in this assignment.

The syntax of *let* is as follows:

```
(let (listOf YourDefinitions) body)
```

And here is the above procedure using *let*:

```
; returns the value in the maze corresponding to position pos.
; Precondition: pos is a valid position in the maze.
```

```
(define (getMazeVal maze pos)
  (let ( (x (car pos)) ; here we put our local definitions
        (y (cadr pos))
        )
    ;the following is the body of let
    (if (zero? x) ;if pos in the row 0. This is the base-case.
        (getItem (car maze) y) ;base-case: calling helper function getItem
        (getMazeVal (cdr maze) (list (- x 1) y)) ;else: recursive call
    )
  )
)
```