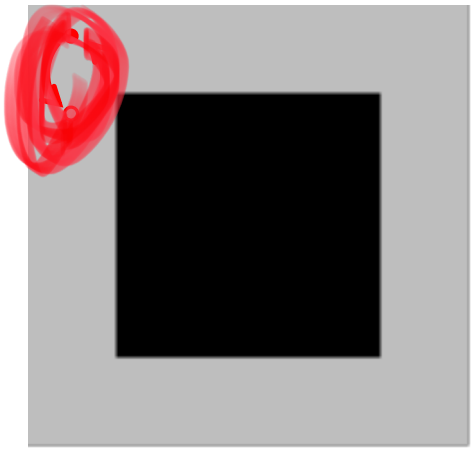
Lecture 21
mazes and graphs



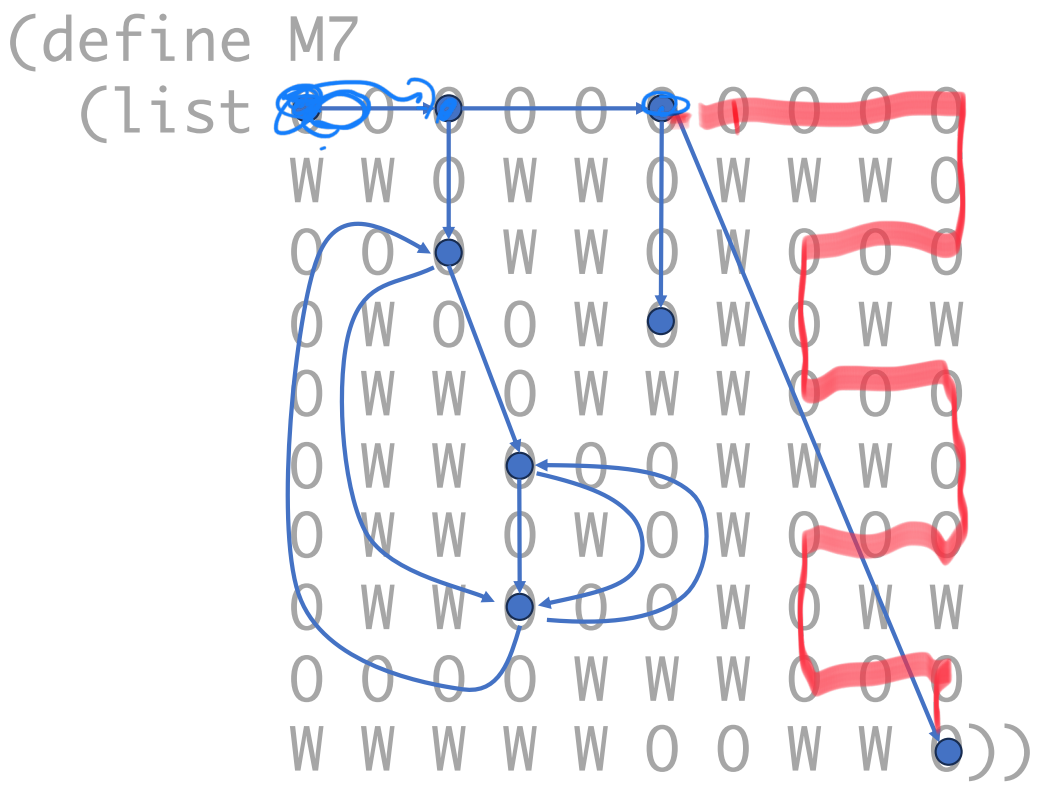
cycles!
directed
graph

start end

```
(define M7
  (list O O O O O O O O O O
        W W O W W O W W W O
        O O O W O W O O O
        O W O O W O W O W W
        O W W O W W W O O O
        O W W O W W W W W O
        O W W O W O W O O O
        O W W O O O W O W W
        O O O O W W W O O O
        W W W W W O O W W O))
```

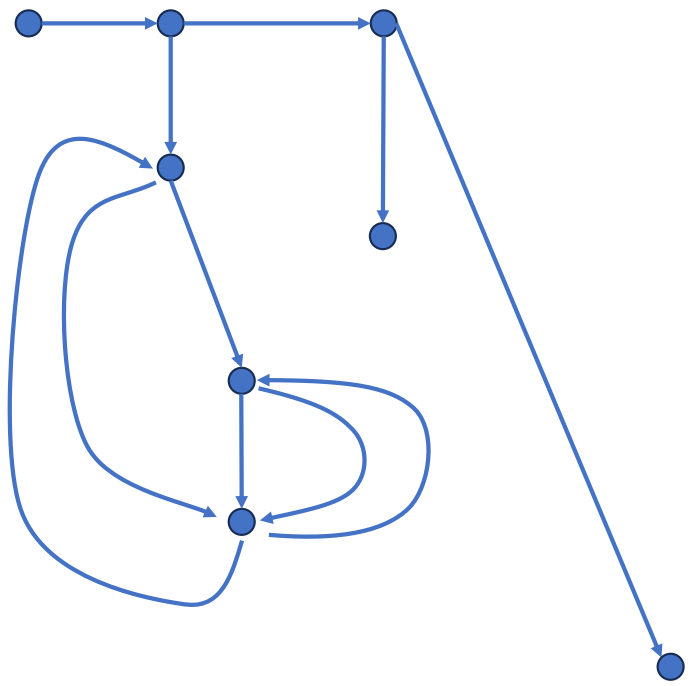
The fine grained maze structure is pretty dense. Not complex really, but just dense.

But we can simplify…



The fine grained maze structure is pretty dense. Not complex really, but just dense.

But we can simplify…



The fine grained maze structure is pretty dense. Not complex really, but just dense.

But we can simplify…

two functions

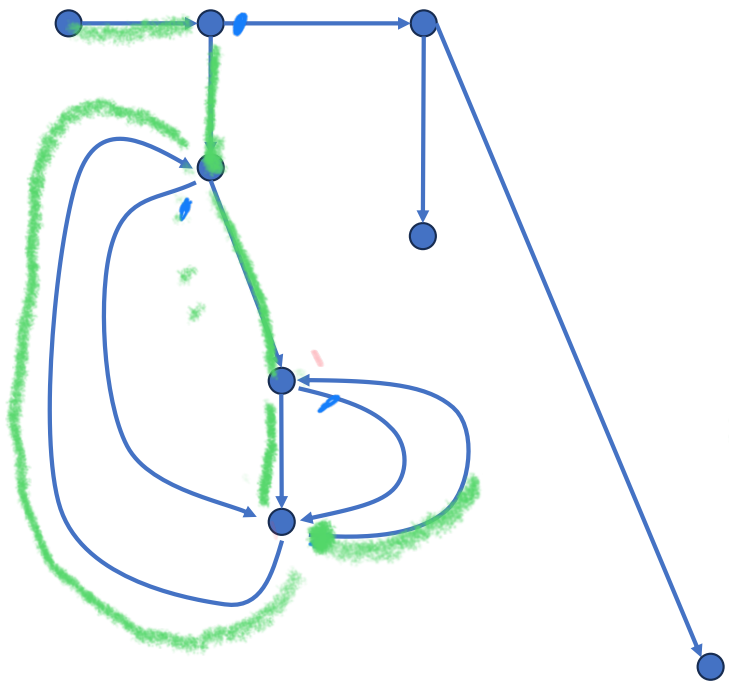
- solveable-no-revisits
  - given maze m, is it possible to go from 0,0 to lower right
  - function must not visit any position more than once
  - (it can come to the position, but not pass through it)

  - A) ordinary recursion B) tail recursion

- distance-from
  - given maze m and positions and and b, if it is possible, starting at 0,0 to first reach a, and then reach b, produce distance from a to b

  - A) ordinary recursion B) tail recursion

normal
rec. path
visited TR



solvable-no-revisits?

graph: yes → path or visited required

TR

…

above
left siblings

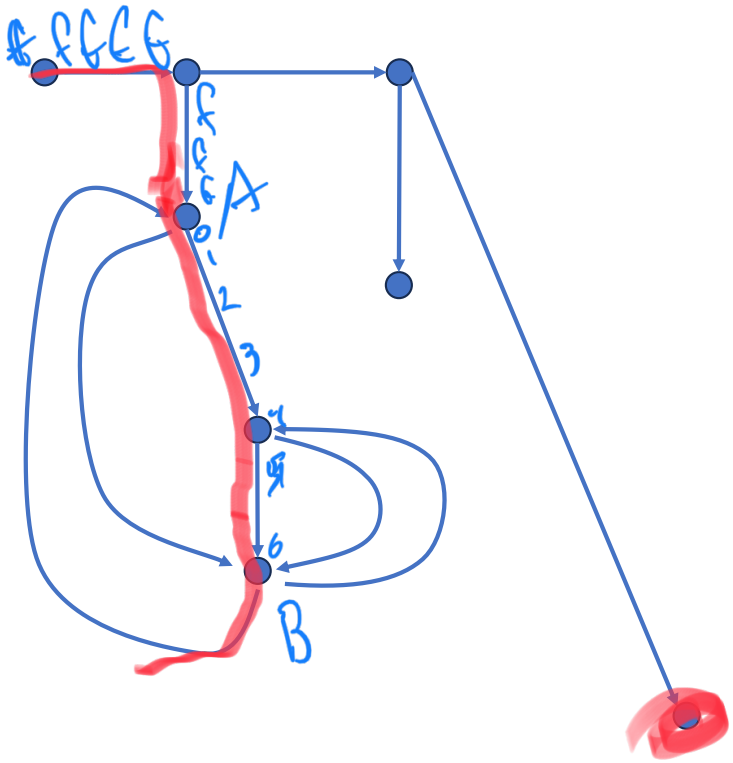
```
(define (solvable-no-revisits? m)
  (local [(define R (sqrt (length m)))

          ;; trivial:
          ;; reduction:
          ;; argument:

          (define (fn-for-p p p-wl visited)
            (cond [(solved? p) true]
                  [(member? p visited) (fn-for-lop p-wl visited)]
                  [else
                   (fn-for-lop (append (next-ps p) p-wl) (cons p visited))]))

          (define (fn-for-lop p-wl visited)
            (cond [(empty? p-wl) false]
                  [else
                   (fn-for-p (first p-wl) (rest p-wl) visited)]))
```

some path in data not TR



distance-from

graph: yes → path or visited required

…

normal rec. dist above

```
(define (distance-from m start end)
  (local [(define R (sqrt (length m)))

          ;; trivial:
          ;; reduction:
          ;; argument:

          (define (fn-for-p p path dist)
            (cond [(equal? p end)   (distance-add1 dist)]
                 ;[(solved? p)      false]
                  [(member? p path) false]
                  [else
                   (if (equal? p start)
                       (fn-for-lop (next-ps p) (cons p path) 0)
                       (fn-for-lop (next-ps p)
                                   (cons p path)
                                   (distance-add1 dist)))]))

          (define (fn-for-lop lop path dist)
            (cond [(empty? lop) false]
                  [else
                   (local [(define try (fn-for-p (first lop) path dist))]
                     (if (not (false? try))
                         try
                         (fn-for-lop (rest lop) path dist)))]))
```