Path Accumulators

When trying to keep track of a node from a root (or other node), keeping track of the paths taken prevents the same path being taken (looping). Think of these as bungee cords that come from a root and try to find a route.

- Original recursion - current branch of the data is in the past
- Prevents cycles (A leads to B and B leads to A)

```
(local [(define R (sqrt (length m)))

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

      (define (fn-for-p p path passed-start?)
        (cond [(equal? p end)   (add1 (position-of start path))]
              ;if end is reached, then find the distance between start/path
              ;[(solved? p)      false]
              [(member? p path) false] ;termination condition/result
              [else
               (if (equal? p start)
                   (fn-for-lop (next-ps p) (cons p path) true)
                   ; if the starting node has been found, pass true as acc
                   (fn-for-lop (next-ps p) (cons p path) passed-start?))]))
                   ; if the starting node is yet to be found pass acc
      (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)))]))

      ;; CONSTRAINT: p is in lop
      (define (position-of p lop)
        (cond [(empty? p) (error "p was not in lop")]
              [else
               (if (equal? p (first lop))
                   0
                   (add1 (position-of p (rest lop))))]))

   ...

(fn-for-p (make-pos 0 0) empty false)))
```

(Full Code)

```
(local [(define R (sqrt (length m)))

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

      (define (fn-for-p p path passed-start?)
        (cond [(equal? p end)   (add1 (position-of start path))]
              ;if end is reached, then find the distance between start/path
              ;[(solved? p)      false]
              [(member? p path) false] ;termination condition/result
              [else
               (if (equal? p start)
                   (fn-for-lop (next-ps p) (cons p path) true)
                   ; if the starting node has been found, pass true as acc
                   (fn-for-lop (next-ps p) (cons p path) passed-start?))]))
                   ; if the starting node is yet to be found pass acc
      (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)))]]))

        ;; CONSTRAINT: p is in lop
        (define (position-of p lop)
          (cond [(empty? p) (error "p was not in lop")]
                [else
                 (if (equal? p (first lop))
                     0
                     (add1 (position-of p (rest lop))))]))

        ;; Pos -> Boolean
        ;; produce true if pos is at the lower right
        (define (solved? p)
          (and (= (pos-x p) (sub1 R))
               (= (pos-y p) (sub1 R))))


        ;; Pos -> (listof Pos)
        ;; produce next possible positions based on maze geometry
        (define (next-ps p)
          (local [(define x (pos-x p))
                  (define y (pos-y p))]
            (filter (lambda (p1)
                      (and (<= 0 (pos-x p1) (sub1 R))  ;legal x
                           (<= 0 (pos-y p1) (sub1 R))  ;legal y
                           (open? (maze-ref m p1))))   ;open?
                    (list (make-pos x (sub1 y))        ;up
                          (make-pos x (add1 y))        ;down
                          (make-pos (sub1 x) y)        ;left
                          (make-pos (add1 x) y)))))    ;right

        ;; Maze Pos -> Boolean
        ;; produce contents of maze at location p
        ;; assume p is within bounds of maze
        (define (maze-ref m p)
          (list-ref m (+ (pos-x p) (* R (pos-y p)))))]

  (fn-for-p (make-pos 0 0) empty false)))
```

Visited Accumulators

When trying to keep track of all the nodes that have been checked, visited prevents ... . This is similar to spray paint where if a node has already been "visited," the program proceeds to the next node. This prevents

- Tail recursion - every node visited in the computation
- Prevents cycles and joins (two paths to the same node)

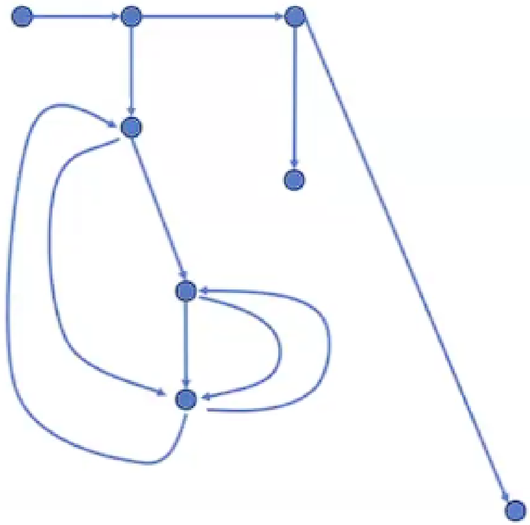
```
(define (solvable-no-revisits? m)
  (local [(define R (sqrt (length m)))

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

        (define (fn-for-p p p-wl visited)
          (cond [(solved? p) true] ;is condition met?
                [(member? p visited) (fn-for-lop p-wl visited)]
                ;if path has been visited, skip by "ignoring" current p by:
                ; 1. not adding p to visited accumulator
                ; 2. not adding p's children to worklist accumulator
                [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)]))
        ; tail recursive to record all nodes visited.

        ...1

  (fn-for-p (make-pos 0 0) empty empty)))
```

(Full code)

```
(local [(define R (sqrt (length m)))

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

        (define (fn-for-p p p-wl visited)
          (cond [(solved? p) true] ;is condition met?
                [(member? p visited) (fn-for-lop p-wl visited)]
                ;if path has been visited, skip by "ignoring" current p by:
                ; 1. not adding p to visited accumulator
                ; 2. not adding p's children to worklist accumulator
                [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)]))
        ; tail recursive to record all nodes visited.


        ;; Pos -> Boolean
        ;; produce true if pos is at the lower right
        (define (solved? p)
          (and (= (pos-x p) (sub1 R))
               (= (pos-y p) (sub1 R))))


        ;; Pos -> (listof Pos)
        ;; produce next possible positions based on maze geometry
        (define (next-ps p)
          (local [(define x (pos-x p))
                  (define y (pos-y p))]
            (filter (lambda (p1)
                      (and (<= 0 (pos-x p1) (sub1 R))  ;legal x
                           (<= 0 (pos-y p1) (sub1 R))  ;legal y
                           (open? (maze-ref m p1))))   ;open?
                    (list (make-pos x (sub1 y))        ;up
                          (make-pos x (add1 y))        ;down
                          (make-pos (sub1 x) y)        ;left
                          (make-pos (add1 x) y)))))    ;right

        ;; Maze Pos -> Boolean
        ;; produce contents of maze at location p
        ;; assume p is within bounds of maze
        (define (maze-ref m p)
          (list-ref m (+ (pos-x p) (* R (pos-y p)))))]

  (fn-for-p (make-pos 0 0) empty empty)))
```

BFS vs. DFS

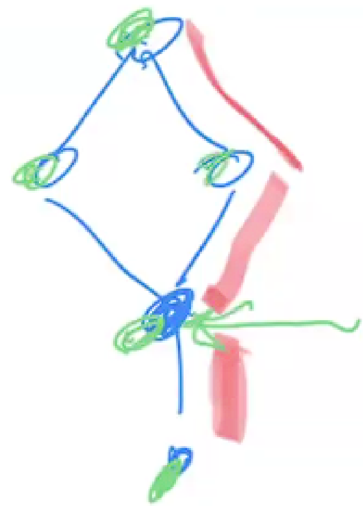


solvable-no-revisits?

graph: yes → path or visited required

must be TR

...



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

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

          ;; visited is (listof Pos)
          ;; every position passed through so far in the tail recursion

          (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)]))
```

```
;; have we passed through start in the current branch

(define (fn-for-p p path passed-start?)
  (cond [(and passed-start? (equal? p end))
         (add1 (position-in start path))]
        [(member? p path) false] ;ordinary recursion fail
        [else
         (fn-for-lop (next-ps p)
                     (cons p path)
                     (if (equal? p start)
                         true
                         passed-start?))]))

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

```
;; CONSTRAINT p is in path
(define (position-in p path)
  (cond [(empty? path) "can't happen!"]
        [else
         (if (equal? (first path) p)
             0
             (add1 (position-in p (rest path))))]))
```

6'0

```
;; Question 2:
;;
;; I know that:
;;
;; A. working through practice problems on my own is the best way to
;;    prepare for the final
;; B. WATCHING video is nowhere near as effective as playing a little,
;;    stopping, trying to get ahead, restarting and comparing what I
;;    did to the video
;; C. Office hours are the best place to work through practice problems
;;    because I can get hints from course staff rather than looking at solution
;; D. Using the problem bank is a good way to assess what material I already
;;    have mastered and what material I should be working on
;; D2. The exam will have more than one problem, spending too much mental energy
;;    preparing for tandem worklist graph problems isn't a good use of my time
;;    until I am already comfortable with everything up to that point
;; E. All of the above
```

Everything but big band,
Unlikely for htdf module 1 problem (6 and on)

What have you learned ... about design?

- Figuring out what you actually want is half the battle
  - signature
  - purpose
  - examples (wrapped in check-expect)

  - information examples
  - interpretation

What have you learned ... about design?

- then the structure of the solution
  - template origins
  - accumulator types and invariants

- and the details
  - fill in ... according to all above
  - debug

`All 5 tests pass!`

What have you learned ... about design?

- but sometimes

```
50% of  50%   Submitted tests: correct - all submitted test pass.
 0% of  50%   Additional tests: incorrect - 3 autograder internal additional tests failed.
```

  - despite your best efforts
  - what you end up with is not what you really wanted
  - go back and systematically revise the design, and learn from that error

Problems that build off of PSET 9-11