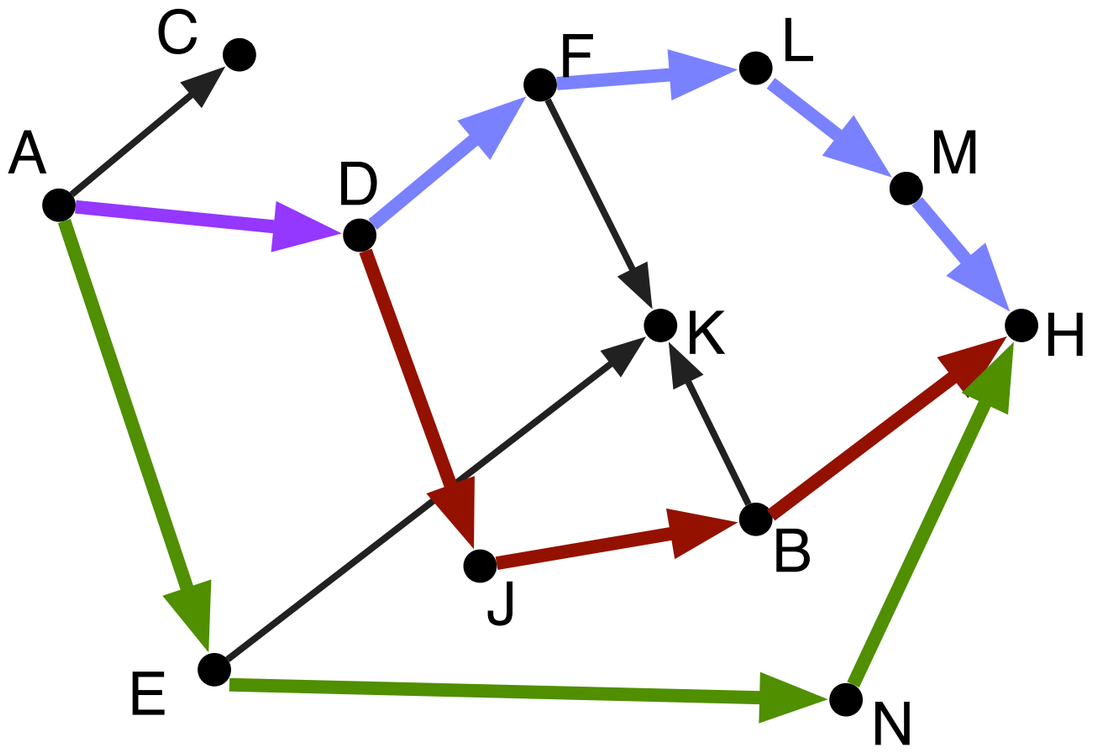
CS135 Tutorial 11

Generative Recursion, Graphs

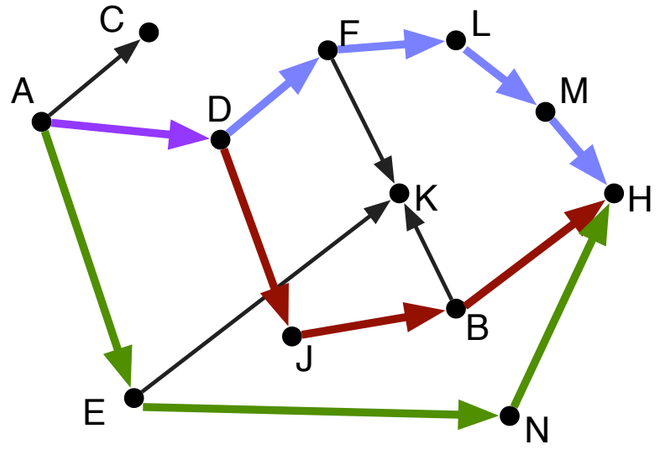
Finding Paths

- Consider this graph and finding a path from `'A` to `'H`. There are three such paths.
- Which one will `(find-path 'A 'H g2)` produce?




```
(define g2
  '((A (C D E))
    (B (H K))
    (C ())
    (D (F J))
    (E (K N))
    (F (K L))
    (H ())
    (J (B))
    (K ())
    (L (M))
    (M (H))
    (N (H))))
```

Find-Path – lecture version




```
(define (find-path orig dest g)
  (cond [(symbol=? orig dest) (list dest)]
        [else (local [(define nbrs (neighbours orig g))
                      (define ?path (find-path/list nbrs dest g))]
                (cond [(false? ?path) false]
                      [else (cons orig ?path)]))]))

(define (find-path/list nbrs dest g)
  (cond [(empty? nbrs)  false]
        [else (local [(define ?path (find-path (first nbrs) dest g))]
                (cond [(false? ?path)
                       (find-path/list (rest nbrs) dest g)]
                      [else ?path]))]))
```

```
(define g2
  '((A (C D E))
    (B (H K))
    (C ())
    (D (F J))
    (E (K N))
    (F (K L))
    (H ())
    (J (B))
    (K ())
    (L (M))
    (M (H))
    (N (H))))
```

What kind of recursion?

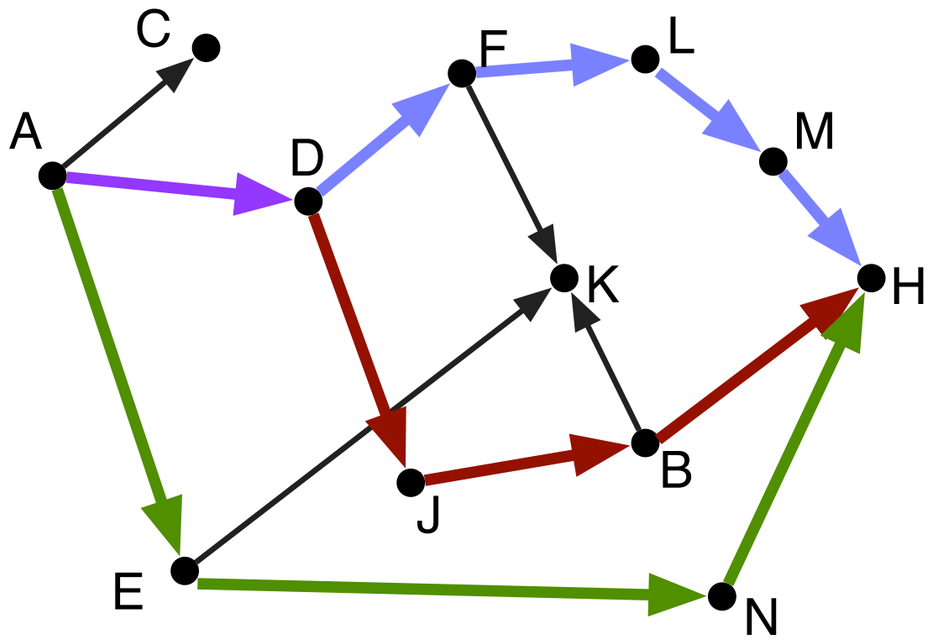
Simple: In recursive applications, args are either unchanged or one step closer to a base case in the data def.	`(define (foo x)` `  ... (foo (one-step-closer x)) ...)`

Accumulative: One or more parameters that collect information that is the answer at the end. Based on data def.	`(define (foo x acc)`
	`  (cond [(all-done? x) acc]`
	`  [else (foo (one-step-closer x)`
	`  (op acc))])) `

Mutual: Two (or more) functions call each other based on data def.	`(define (foo x) ... (bar (op1 x)) ...)` `(define (bar y) ... (foo (op2 y)) ...)`
Generative: Arguments are calculated without a data def.	`(define (foo x) ... (foo (arbitrary-calc x))...)`

Shortest Path Strategy

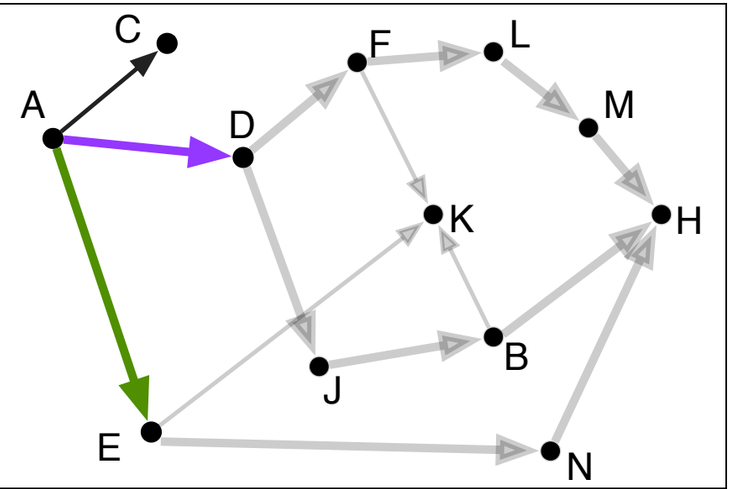
- Suppose we'd like the shortest path. That's not uncommon:
  - Google Maps wouldn't be very popular if the route from Waterloo to Toronto went via Winnipeg.
  - Booking an airline flight that took three puddle-jumpers instead of 1 direct flight.



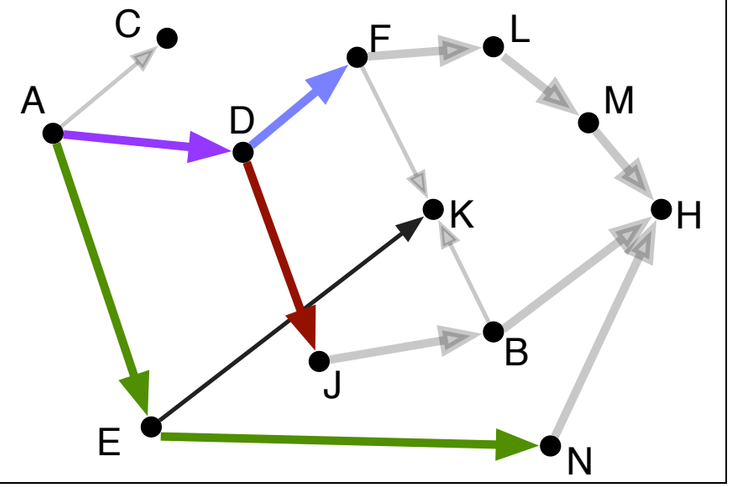
- How could we go about finding the shortest path?
  - Assume we have an acyclic graph to keep it simpler.

Strategy 2: Extend all paths by 1 until…

```
(extend-paths '((A)) g2)
=> (list (list 'C 'A)
         (list 'D 'A)
         (list 'E 'A))
```

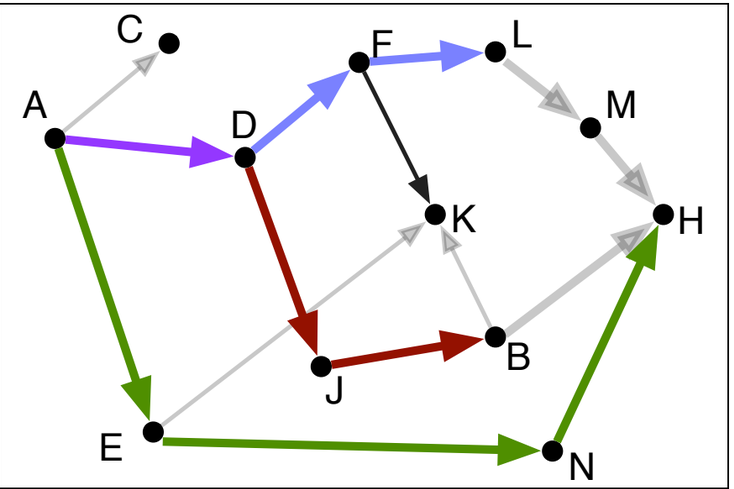



```
(extend-paths
   (extend-paths '((A)) g2) g2)
=> (list (list 'K 'E 'A)
         (list 'N 'E 'A)
         (list 'F 'D 'A)
         (list 'J 'D 'A))
```

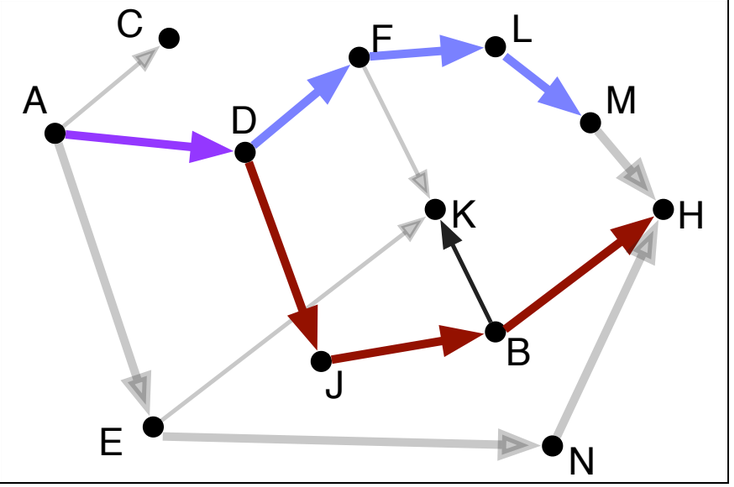


Strategy 2: Extend all paths by 1 until…

```
(extend-paths
   (extend-paths
      (extend-paths '((A)) g2) g2) g2)
=> (list (list 'B 'J 'D 'A)
         (list 'K 'F 'D 'A)
         (list 'L 'F 'D 'A)
         (list 'H 'N 'E 'A))
```




```
(extend-paths
   (extend-paths
      (extend-paths
         (extend-paths '((A)) g2) g2) g2) g2)
=> (list (list 'M 'L 'F 'D 'A)
         (list 'H 'B 'J 'D 'A)
         (list 'K 'B 'J 'D 'A))
```



Data Definitions

```
;; A Path is a (ne-listof Node)
;; Paths are a (listof Path)
```

Strategy 1: Find all paths; take the shortest

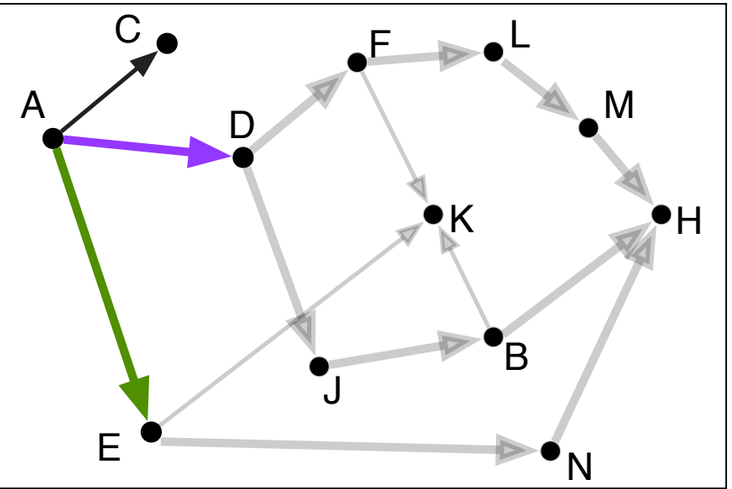
```
;; (findall-path orig dest g) finds all paths from orig to dest in g
;; find-path: Node Node Graph -> Paths
(define (findall-paths orig dest g)
  (cond [(symbol=? orig dest) (list (list dest))]
        [else (local [(define nbrs (neighbours orig g))
                      (define paths (findall-paths/list nbrs dest g))]
                (map (lambda (p) (cons orig p)) paths))]))

;; (findall-paths/list nbrs dest g) produces all the paths from
;;     nbrs to dest in g
;; find-path/list: (listof Node) Node Graph -> Paths
(define (findall-paths/list nbrs dest g)
  (cond [(empty? nbrs)  empty]
        [else (append (findall-paths (first nbrs) dest g)
                      (findall-paths/list (rest nbrs) dest g))]))
```

Strategy 1: Find all paths; take the shortest

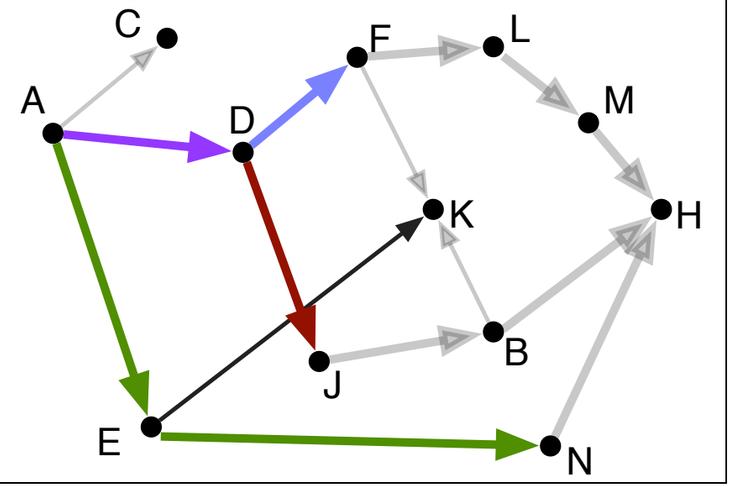
```
;; (shortest orig dest g) finds the shortest path from orig to
;; dest in g; empty if no path exists.
;; shortest: Node Node Graph -> (anyof Path false)
(define (shortest-path orig dest g)
  (local [(define paths (findall-paths orig dest g))]
    (cond [(empty? paths) false]
          [else (foldl (lambda (p rror) (cond [(< (length p) (length rror)) p]
                                              [else rror]))
                       (first paths) (rest paths))])))
```

Strategy 2: Extend all paths by 1 until…

```
(extend-paths '((A)) g2)
=> (list (list 'C 'A)
         (list 'D 'A)
         (list 'E 'A))
```



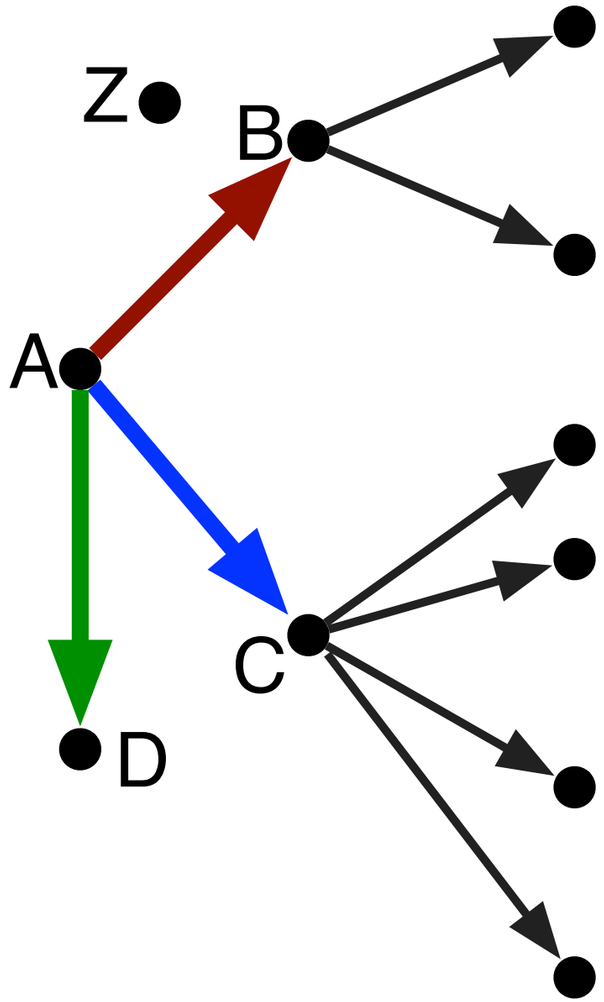

```
(extend-paths (list (list 'C 'A)
                    (list 'D 'A)
                    (list 'E 'A)) g2)
=> (list (list 'K 'E 'A)
         (list 'N 'E 'A)
         (list 'F 'D 'A)
         (list 'J 'D 'A))
```



Strategy 2: Extend all paths by 1 until…

```
(extend-paths (list (list 'B 'A)
                    (list 'C 'A)
                    (list 'D 'A)) g)
```

produces how many paths?



Strategy 2: Extend paths

```
;; (extend-paths paths dest g) extends each path in paths with the
;; neighbours of the first node in the path.
;; extend-paths: Paths Graph -> Paths
(define (extend-paths paths g)
  (local [;; Extend one path
          ;; extend-one-path: Path -> Paths
          (define (extend-one-path path)
            (local [(define nbrs (neighbours (first path) g))]
              (map (lambda (n) (cons n path)) nbrs)))

          ;; Add the results of extending one path to
          ;; the list we're building up.
          (define (handle-one-path path rror)
            (append (extend-one-path path) rror))]

    (foldl handle-one-path empty paths)))
```

Strategy 2: A useful helper function

```
;; (find pred? lst) finds the first element in lst that satisfies pred?.
;; find: (X -> Bool) (listof X) -> (anyof false X)
(define (find pred? lst)
  (cond [(empty? lst) false]
        [(pred? (first lst)) (first lst)]
        [else (find pred? (rest lst))]))
```

Strategy 2: Keep on extending

```
;; (shortest-path orig dest g) finds the shortest path from orig to dest
;; in g, or false if no such path exists.
;; shortest-path: Node Node Graph -> (anyof Path false)
(define (shortest-path orig dest g)
  (local [; repeatedly extend each path with its neighbours until
          ; a path to dest is found or paths becomes empty.
          (define (repeat paths)
            (local [(define ?path
                      (find (lambda (p) (symbol=? (first p) dest)) paths))]
              (cond [(cons? ?path) (reverse ?path)]
                    [(empty? paths) false]
                    [else (repeat (extend-paths paths g))])))]

    (repeat (list (list orig)))))
```

Summary

- Lots of algorithms benefit from a shortest-path
- The second approach (extending all paths one step at a time) is the more common approach and is more efficient.
- The “AI” for a game (for example) uses a similar approach because you often can’t search all the way to the end of the game.

We saw a couple of places where higher-order functions could be used easily.