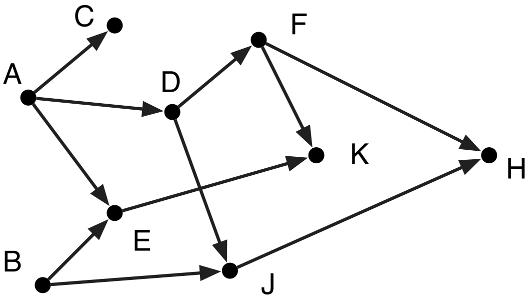
18: Graphs

Directed graphs

A **directed graph** consists of a collection of **nodes** (also called **vertices**) together with a collection of **edges**.

An edge is an ordered pair of nodes, which we can represent by an arrow from one node to another.



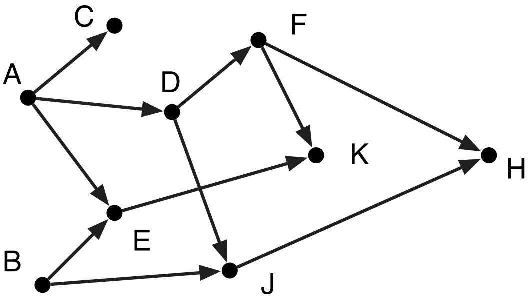
> Directed graphs

We have seen such graphs before.

Binary trees and expression trees were both directed graphs of a special type where an edge represented a parent-child relationship.

Graphs are a general data structure that can model many situations.

Computations on graphs form an important part of the computer science toolkit.



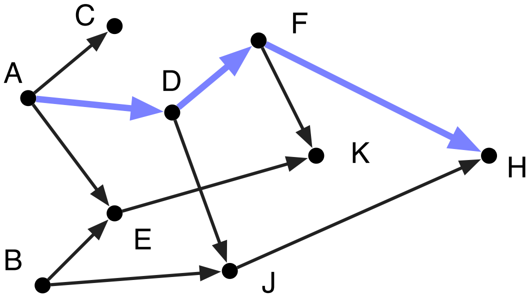
> Graph terminology

Given an edge (v, w), we say that w is an **out-neighbour** of v, and v is an **in-neighbour** of w.

A sequence of nodes $v_1, v_2, \ldots, v_k$ is a **path** or **route** of length $k-1$ if (v_1, v_2), (v_2, v_3), …, (v_{k-1}, v_k) are all edges.

If $v_1 = v_k$, this is called a **cycle**.

Directed graphs without cycles are called **DAG**s (**directed acyclic graphs**).



Representing graphs

We can represent a node by a symbol (its name), and associate with each node a list of its out-neighbours.

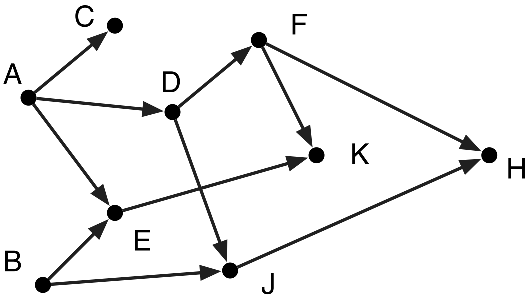
This is called the **adjacency list** representation.

More specifically, a graph is a list of pairs, each pair consisting of a symbol (the node's name) and a list of symbols (the names of the node's out-neighbours).

This is very similar to a parent node with a list of children.

> Our example as data

```
(define g
  '((A (C D E))
    (B (E J))
    (C ())
    (D (F J))
    (E (K))
    (F (K H))
    (H ())
    (J (H))
    (K ()))
  )
```



Recall that `'(A (B C))` is a more compact way of writing `(list 'A (list 'B 'C))`. See M14-25 for a review of quoted lists.

> Data definitions

To make our contracts more descriptive, we will define a `Node` and a `Graph` as follows:

```
;; A Node is a Sym

;; A Graph is one of:
;; * empty
;; * (cons (list v (list w_1 ... w_n)) g)
;;   where g is a Graph
;;         v, w_1, ... w_n are Nodes
;;         v is the in-neighbour to w_1 ... w_n in the Graph
;;         v does not appear as an in-neighbour in g
```

> The template for graphs

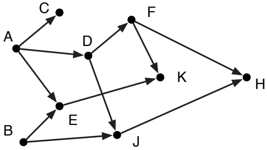
```
;; graph-template: Graph → Any
(define (graph-template g)
  (cond
    [(empty? g) ...]
    [(cons? g)
     (... (first (first g))             ; first node in graph list
          (listof-node-template
               (second (first g)))      ; list of adjacent nodes
          (graph-template (rest g)))]))
```

> `neighbours`

We can use the graph template to write a function that produces the out-neighbours of a node. We'll need this function in just a moment.

```
;; (neighbours v g) produces list of neighbours of v in g
;; Examples:
(check-expect (neighbours 'D g) (list 'F 'J))
(check-expect (neighbours 'Z g) false)

;; neighbours: Node Graph → (anyof (listof Node) false)
;; Requires:  v is a node in g
(define (neighbours v g)
  (cond
    [(empty? g) false]
    [(symbol=? v (first (first g))) (second (first g))]
    [else (neighbours v (rest g))]))
```



Ex. 1

Write `(count-out-neighbours g)` which consumes a `Graph` and produces a `(listof Nat)` indicating how many out-neighbours each `Node` in `g` has.

For example, with the sample graph

```
(check-expect (count-out-neighbours g)
              (list 3 2 0 2 1 2 0 1 0))
```

Hint: `map` and `length` will be useful.

Ex. 2

Write a function `(count-in-neighbours g)` which consumes a `Graph` and produces a `(listof Nat)` indicating how many in-neighbours each `node` has.

For example, with the sample graph

```
(check-expect (count-in-neighbours g)
              (list 0 0 1 1 2 1 2 2 2))
```

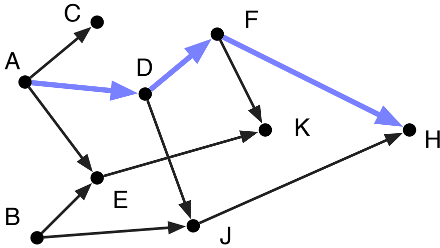
Hint: `filter`, `map`, `length` and `member?` will be useful.

Hint: for each `Node`, we need to count how many nodes in `g` have that node as an out-neighbour.

Finding paths

A path in a graph can be represented by an ordered list of the nodes on the path.

We wish to design a function `find-path` that consumes a graph plus origin and destination nodes, and produces a path from the origin to the destination, or `false` if no such path exists.



```
(find-path 'A 'H g) ⇒ (list 'A 'D 'F 'H) or (list 'A 'D 'J 'H)

(find-path 'D 'H g) ⇒ (list 'D 'F 'H) or (list 'D 'J 'H)

(find-path 'C 'H g) ⇒ false

(find-path 'A 'A g) ⇒ (list 'A)
```

> Cases for `find-path`

Simple recursion does not work for `find-path`; we must use generative recursion.

If the origin equals the destination, the path consists of just this node.

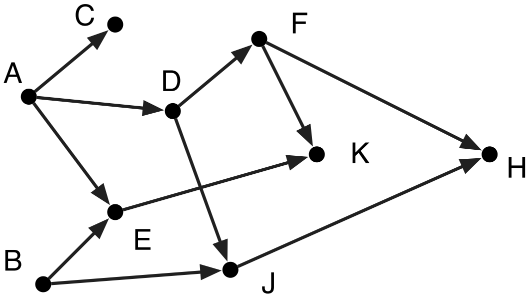
Otherwise, if there is a path, the second node on that path must be an out-neighbour of the origin node.

Each out-neighbour defines a subproblem (finding a path from it to the destination).

> Building a path from a solved sub-problem

In our example, any path from A to H must pass through C, D, or E.

If we knew a path from C to H, or from D to H, or from E to H, we could create one from A to H.



> Backtracking algorithms

Backtracking algorithms try to find a path from an origin to a destination.

If the initial attempt does not work, such an algorithm “backtracks” and tries another choice.

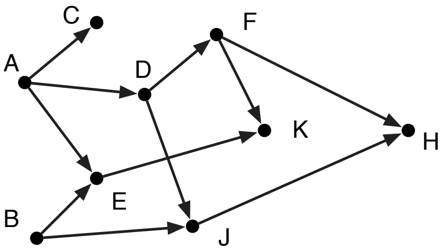
Eventually, either a path is found, or all possibilities are exhausted, meaning there is no path.

» Backtracking in our example

In our example, we can see the "backtracking" since the search for a path from A to H can be seen as moving forward in the graph looking for H.

If this search fails (for example, at C), then the algorithm "backs up" to the previous node (A) and tries the next neighbour (D).

If we find a path from D to H, we can just add A to the beginning of this path.



> Exploring the list of out-neighbours

We need to apply `find-path` on each of the out-neighbours of a given node.

The `neighbours` function gives us a list of all the out-neighbours associated with that node.

This suggests writing `find-path/list` which consumes a list of nodes and will apply `find-path` to each one until it either finds a path to the destination or exhausts the list.

» Mutual recursion

This is the same recursive pattern that we saw in the processing of expression trees and evolutionary trees.

For expression trees, we had two mutually recursive functions, `eval` and `apply`.

Here, we have two mutually recursive functions, `find-path` and `find-path/list`.

> `find-path`

```
;; (find-path orig dest g) finds path from orig to dest in g if it exists
;; find-path: Node Node Graph → (anyof (ne-listof Node) false)
(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)]))]))
```

We're using `?path` to mean it might hold a path or it might not.

`false?` produces `true` if its argument is the value `false`.

> find-path/list

```
;; (find-path/list nbrs dest g) produces path from
;;     an element of nbrs to dest in g, if one exists
;; find-path/list: (listof Node) Node Graph → (anyof (ne-listof Node) false)
(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]))]))
```

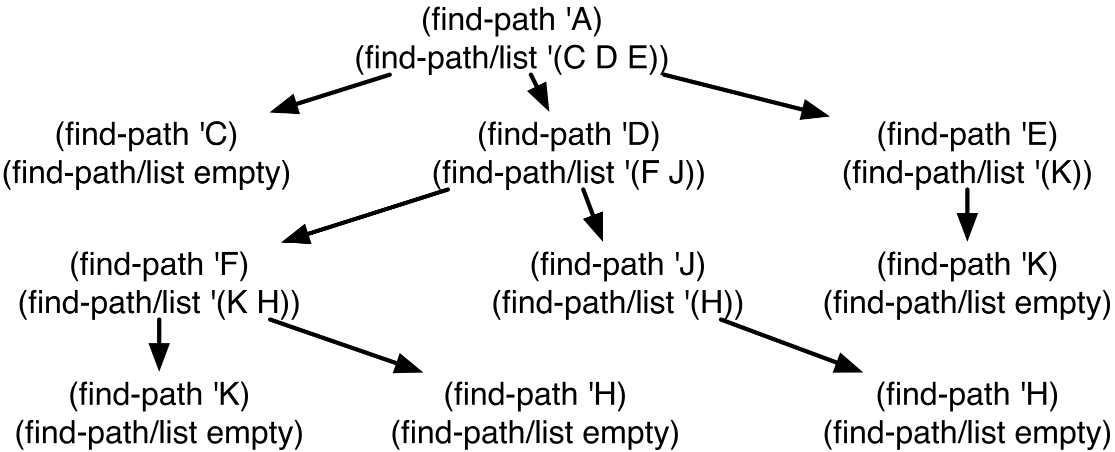
> Tracing `(find-path 'A 'B g)` (1/2)

If we wish to trace `find-path`, trying to do a linear trace would be very long, both in terms of steps and the size of each step. Our traces also are listed as a linear sequence of steps, but the computation in `find-path` is better visualized as a tree.

We will use an alternate visualization of the potential computation (which could be shortened if a path is found).

The next slide contains the trace tree. We have omitted the arguments `dest` and `g` which never change.

» Tracing `(find-path 'A 'B g)` (2/2)



(find-path 'A)
(find-path/list '(C D E))

(find-path 'C)
(find-path/list empty)

(find-path 'D)
(find-path/list '(F J))

(find-path 'E)
(find-path/list '(K))

(find-path 'F)
(find-path/list '(K H))

(find-path 'J)
(find-path/list '(H))

(find-path 'K)
(find-path/list empty)

(find-path 'K)
(find-path/list empty)

(find-path 'H)
(find-path/list empty)

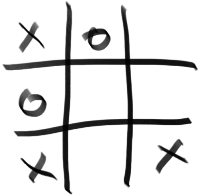
(find-path 'H)
(find-path/list empty)

> Backtracking in implicit graphs (1/3)

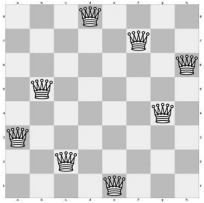
The only places where real computation is done on the graph is in comparing the origin to the destination and in the `neighbours` function.

Backtracking can be used without having the entire graph available if the neighbours can be derived from a “configuration”.

Board games:



Puzzles:



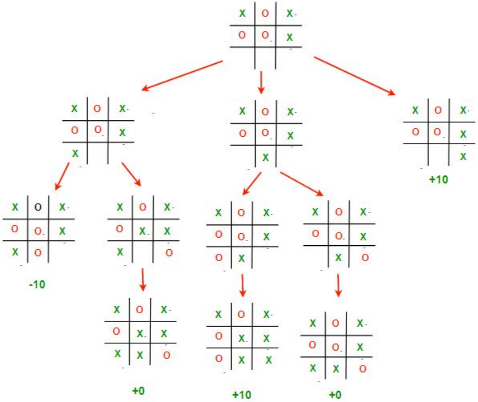
https://www.puzzleprime.com/brain-teasers/deduction/eight-queens-puzzle/

» Backtracking in implicit graphs (2/3)

Nodes typically represent configurations: (e.g. X's and O's played so far)

Edges represent ways in which one configuration becomes another: (e.g. the next player places an X or O)

The graph is acyclic if no configuration can occur twice in a game. This happens naturally when edges represent additions (tic-tac-toe, 8-queens, Sudoku).



https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-3-tic-tac-toe-ai-finding-optimal-move/

» Backtracking in implicit graphs (3/3)

The `find-path` functions for implicit backtracking look very similar to those we have developed.

The `neighbours` function must now generate the set of neighbours of a node based on some description of that node (e.g. the placement of pieces in a game).

This allows backtracking in situations where it would be inefficient to generate and store the entire graph as data.

Backtracking in implicit graphs forms the basis of many artificial intelligence programs, though they generally add heuristics to determine which neighbour to explore first, or which ones to skip because they appear unpromising.

Termination of `find-path` (no cycles)

In a directed acyclic graph, any path with a given origin will recurse on its (finite number) of neighbours by way of `find-path/list`. The origin will never appear in this call or any subsequent calls to `find-path`: if it did, we would have a cycle in our DAG.

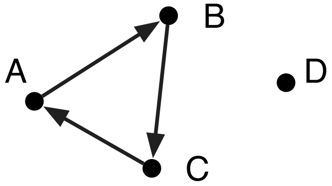
Thus, the origin will never be explored in any later call, and thus the subproblem is smaller. Eventually, we will reach a subproblem of size 0 (when all reachable nodes are treated as the origin).

Thus `find-path` always terminates for directed acyclic graphs.

Non-termination of `find-path` (cycles)

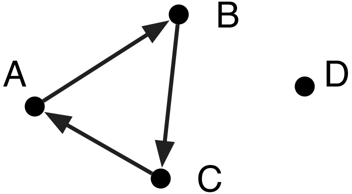
It is possible that `find-path` may not terminate if there is a cycle in the graph.

Consider the graph . What if we try to find a path from A to D in this graph?



```
'((A (B))
  (B (C))
  (C (A))
  (D ()))
```

> Non-termination of `find-path` (cycles)



(find-path 'A)
(find-path/list (list ‘B))
↓
(find-path 'B)
(find-path/list (list 'C))
↓
(find-path 'C)
(find-path/list (list 'A))
↓
(find-path 'A)
(find-path/list (list 'B))
...

Paths v2: Handling cycles

We can use accumulative recursion to solve the problem of `find-path` possibly not terminating if there are cycles in the graph.

To make backtracking work in the presence of cycles, we need a way of remembering what nodes have been visited (along a given path).

Our accumulator will be a list of visited nodes.

We must avoid visiting a node twice.

The simplest way to do this is to add a check in `find-path/list`.

> find-path/list

```
;; find-path/list: (listof Node) Node Graph (listof Node) →
;;                 (anyof (listof Node) false)
(define (find-path/list nbrs dest g visited)
  (cond [(empty? nbrs) false]
        [(member?  (first nbrs) visited)
         (find-path/list (rest nbrs) dest g visited)]
        [else (local [(define ?path (find-path/acc (first nbrs)
                                                  dest g visited))]
                (cond [(false? ?path)
                       (find-path/list (rest nbrs) dest g visited)]
                      [else ?path]))]))
```

> `find-path/list`'s accumulator

The code for `find-path/list` does not add anything to the accumulator (though it uses the accumulator).

Adding to the accumulator is done in `find-path/acc` which applies `find-path/list` to the list of neighbours of some origin node.

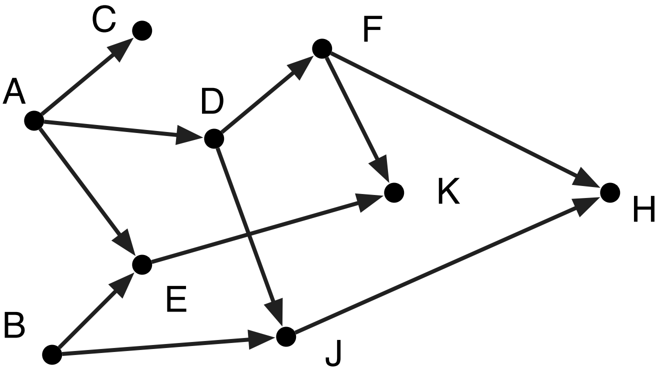
That origin node must be added to the accumulator passed as an argument to `find-path/list`.

> find-path/acc

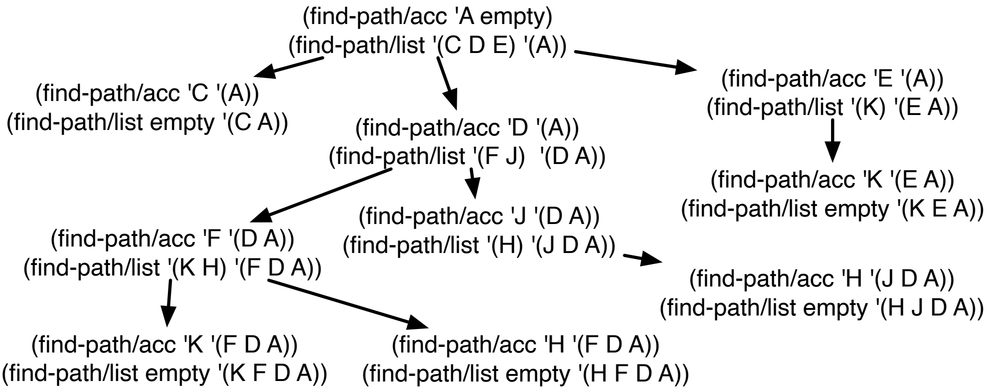
```
;; find-path/acc: Node Node Graph (listof Node) →
;;                (anyof (listof Node) false)
(define (find-path/acc orig dest g visited)
  (cond [(symbol=? orig dest) (list dest)]
        [else (local [(define nbrs (neighbours orig g))
                      (define ?path (find-path/list nbrs dest g
                                                    (cons orig visited)))]
           (cond [(false? ?path) false]
                 [else (cons orig ?path)]))]))


(define (find-path orig dest g)        ;; new wrapper function
  (find-path/acc orig dest g empty))
```

> Tracing our examples (1/4)



> Tracing our examples (2/4)



(find-path/acc 'A empty)
(find-path/list '(C D E) '(A))

(find-path/acc 'C '(A))
(find-path/list empty '(C A))

(find-path/acc 'D '(A))
(find-path/list '(F J) '(D A))

(find-path/acc 'E '(A))
(find-path/list '(K) '(E A))

(find-path/acc 'K '(E A))
(find-path/list empty '(K E A))

(find-path/acc 'F '(D A))
(find-path/list '(K H) '(F D A))

(find-path/acc 'J '(D A))
(find-path/list '(H) '(J D A))

(find-path/acc 'H '(J D A))
(find-path/list empty '(H J D A))

(find-path/acc 'K '(F D A))
(find-path/list empty '(K F D A))

(find-path/acc 'H '(F D A))
(find-path/list empty '(H F D A))

Note that the value of the accumulator in `find-path/list` is always the reverse of the path from A to the current origin (first argument).

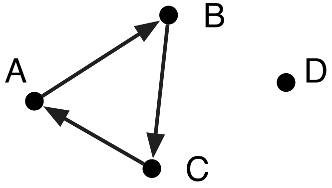
> Tracing our examples (3/4)

This example has no cycles, so the trace only convinces us that we haven't broken the function on acyclic graphs, and shows us how the accumulator is working.

But it also works on graphs with cycles.

The accumulator ensures that the depth of recursion is no greater than the number of nodes in the graph, so `find-path` terminates.

Tracing our examples (4/4)



(find-path/acc 'A empty)
(find-path/list '(B) (list 'A))

↓

(find-path/acc 'B (list 'A))
(find-path/list '(C) (list 'B 'A))

↓

(find-path/acc 'C (list 'B 'A))
(find-path/list '(A) (list 'C 'B 'A))

no further calls to find-path/acc

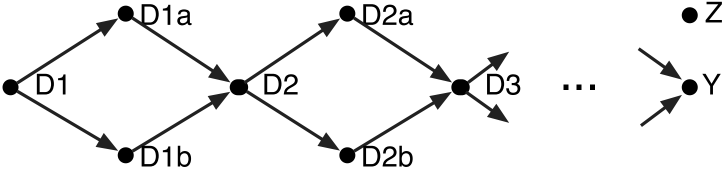
> Cycles is solved, but...

Backtracking now works on graphs with cycles, but it can be inefficient, even if the graph has no cycles.

If there is no path from the origin to the destination, then `find-path` will explore every path from the origin, and there could be an exponential number of them.

Paths v3: Efficiency

If there is no path from the origin to the destination, then `find-path` will explore every path from the origin, and there could be an exponential number of them.



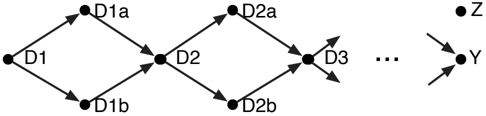
If there are d diamonds, then there are $3d + 2$ nodes in the graph, but 2^d paths from `D1` to `Y`, all of which will be explored.

> Understanding the problem (1/2)

Applying `find-path/acc` to origin `D1` results in `find-path/list` being applied to `(list 'D1a 'D1b)`, and then `find-path/acc` being applied to origin `D1a`.

There is no path from `D1a` to `Z`, so this will produce `false`, but in the process, it will visit all the other nodes of the graph except `D1b` and `Z`.

`find-path/list` will then apply `find-path/acc` to `D1b`, which will visit all the same nodes again.

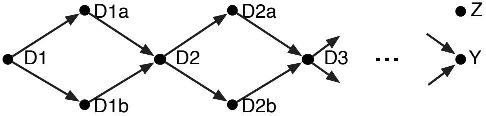


> Understanding the problem (2/2)

When `find-path/list` is applied to the list of nodes `nbrs`, it first applies `find-path/acc` to `(first nbrs)` and then, if that fails, it applies itself to `(rest nbrs)`.

To avoid revisiting nodes, the failed computation should pass the list of nodes it has seen on to the next computation.

It will do this by returning the list of visited nodes instead of `false` when it fails to find a path. However, we must be able to distinguish this list from a successfully found path (also a list of nodes).



> Remembering what the list of nodes represents

We will encapsulate each kind of list in its own structure. We can then easily use the structure predicates (`success?` and `failure?`) to check whether the list of nodes represents a path (success) or visited nodes (failure).

```
(define-struct success (path))
;; A Success is a (make-success (listof Node))

(define-struct failure (visited))
;; A Failure is a (make-failure (listof Node))

;; A Result is (anyof Success Failure)
```

> find-path/list

```
;; find-path/list: (listof Node) Node Graph (listof Node) → Result
(define (find-path/list nbrs dest g visited)
  (cond [(empty? nbrs) (make-failure visited)]
        [(member? (first nbrs) visited)
         (find-path/list (rest nbrs) dest g visited)]
        [else (local [(define result (find-path/acc (first nbrs)
                                                    dest g visited))]
                (cond [(failure? result)
                       (find-path/list (rest nbrs) dest g
                                       (failure-visited result))]
                      [(success?  result) result]))]))
```

`?path` is renamed `result` for clarity.

```
;; find-path/acc: Node Node Graph (listof Node) → Result
(define (find-path/acc orig dest g visited)
  (cond [(symbol=? orig dest) (make-success (list dest))]
        [else (local [(define nbrs (neighbours orig g))
                      (define result (find-path/list nbrs dest g
                                                    (cons orig visited)))]
                (cond [(failure?  result) result]
                      [(success?  result)
                        (make-success (cons orig
                                           (success-path result)))]))]))
```

`?path` is renamed `result` for clarity.

> `find-path`

```
;; find-path: Node Node Graph → (anyof (listof Node) false)
(define (find-path orig dest g)
  (local [(define result (find-path/acc orig dest g empty))]
    (cond [(success?  result) (success-path result)]
          [(failure?  result) false])))
```

> `find-path` v3 summary

With these changes, `find-path` runs *much faster* on the diamond graph.

In future courses we will see how to make `find-path` even more efficient and how to formalize our analyses.

Knowledge of efficient algorithms, and the data structures that they utilize, is an essential part of being able to deal with large amounts of real-world data.

These topics are studied in CS 240 and CS 341 (for majors) and CS 234 (for non-majors).

Ex. 3

Write a function `k-path-length` which consumes a symbol `start` corresponding to a node, a number `k`, and a graph. If there is a path with `k` or more edges originating from `start` that does not repeat any nodes, the function produces one such path. Otherwise the function produces `false`.

Ex. 4

Write a function, `make-diamond-graph`, which consumes *n* and produces a `Graph` with *n* diamonds. You can make a symbol to identify a node with

```
;; mk-node: Nat Str -> Sym
(define (mk-node n suffix)
  (string->symbol (string-append "D" (number->string n) suffix)))
```

Note the use of `string->symbol` which we are **not** including as one of the “permitted functions” on the last slide!

Ex. 5

Write a function, `graph-complement`, that consumes a graph and produces its complement. The complement of a graph g is a graph g' such that for each pair of nodes u and v, (u, v) is an edge in g' if and only if it is not an edge in g. Assume that neither graph has edges from a node to itself. For example, the complement of `simple-graph` is `complement-graph`:

```
(define simple-graph
  '((a (i j k))
    (j ())
    (k (a j))
    (i (j))))
```

```
(define complement-graph
  '((a ())
    (j (i a k))
    (k (i))
    (i (a k))))
```

Use explicit recursion. Encapsulate helper functions using **`local`**.

Ex. 6

Write a function, `graph-complement/alf`, which is the same as `graph-complement` except that it is implemented using higher order functions, not explicit recursion.

Goals of this module

- You should understand directed graphs and their representation in Racket.
- You should be able to write functions which consume graphs and compute desired values.
- You should understand and be able to implement backtracking on explicit and implicit graphs.
- You should understand the problems that the second and third versions of `find-path` address and how they solve those problems.

Summary: built-in functions

The following functions and special forms have been introduced in this module:

`false? member?`

You should complete all exercises and assignments using only these and the functions and special forms introduced in earlier modules. The complete list is:

`* + - ... / < <= = > >= abs add1` **`and`** `append boolean? build-list ceiling char-alphabetic? char-downcase char-lower-case? char-numeric? char-upcase char-upper-case? char-whitespace? char<=? char<? char=? char>=? char>? char? check-error check-expect check-within` **`cond`** `cons cons? cos` **`define define-struct`** `define/trace e eighth` **`else`** `empty? equal? error even? exp expt false? fifth filter first floor foldl foldr fourth integer?` **`lambda`** `length list list->string list?` **`local`** `log map max member? min modulo negative? not number->string number? odd?` **`or`** `pi positive? quicksort quotient remainder rest reverse round second seventh sgn sin sixth sqr sqrt string->list string-append string-downcase string-length string-lower-case? string-numeric? string-upcase string-upper-case? string<=? string<? string=? string>=? string>? string? sub1 substring symbol=? symbol? tan third zero?`