Directed graphs

CS135 Lecture 20

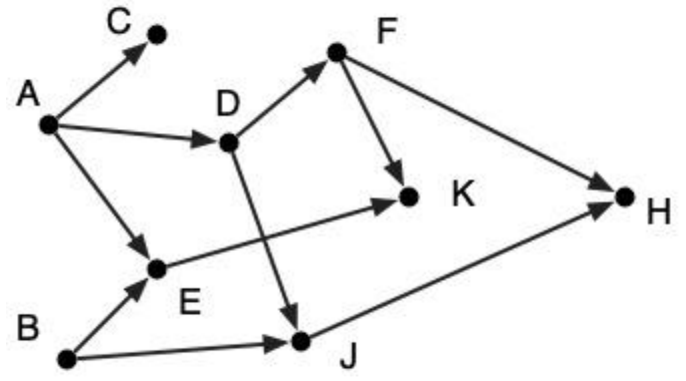
L20.0 Graphs as data abstractions

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 represent by an arrow from one node to another.

There are also undirected graphs, but CS135 doesn't use them. When we say "graph" we mean "directed graph".



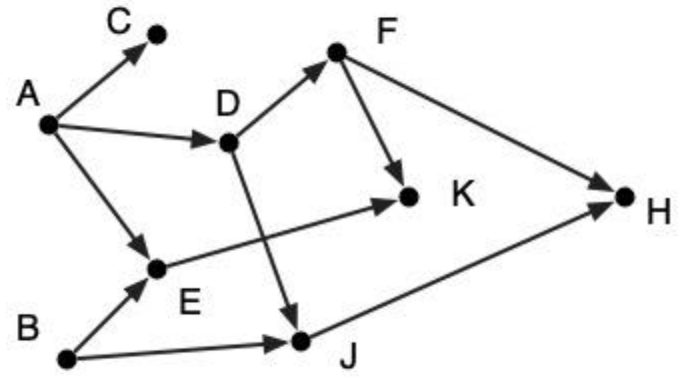
Directed graphs

We have seen such structures before.

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

Directed graphs are a general data structure capable of modeling many kinds of situations.

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



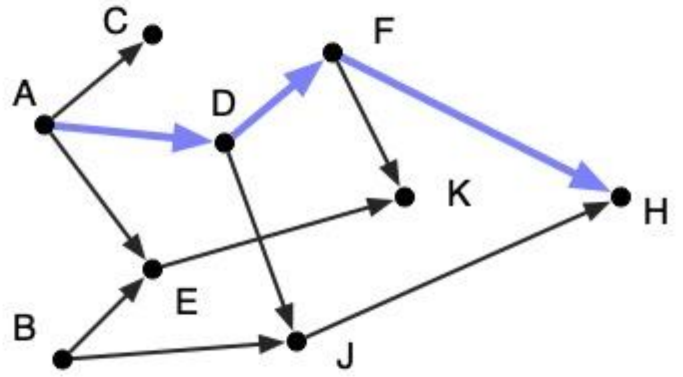
Directed graphs

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,..., v_k is a **path** of length $k - 1$ if (v_1, v_2), (v_2, v_3),..., (v_{k-1}, v_k) are all edges.

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

A directed graph without a cycle is called a **directed acyclic graph** (DAG).



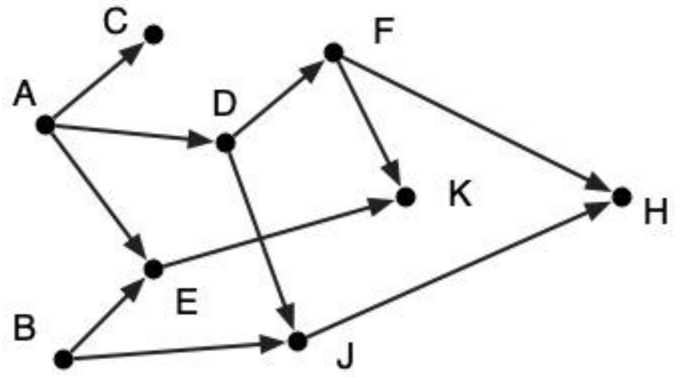
L20.1 Representing directed graphs

Adjacency lists

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

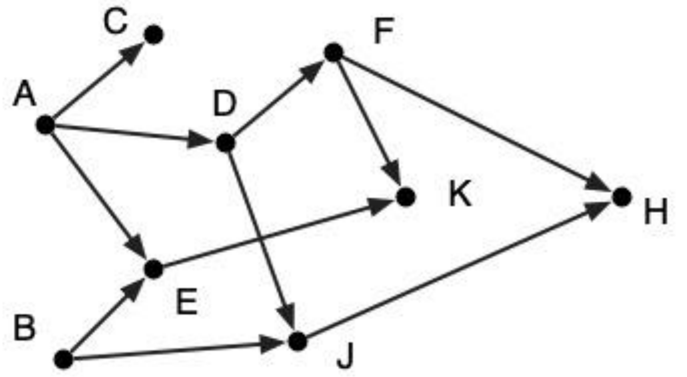
This is called an **adjacency list** representation.

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).



An example of an adjacency list

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



Data definitions

```
;; A Set is a (listof Sym)
;; Requires: symbols must be unique

;; A Graph is an association list
;; from nodes to their set of out-neighbours
;; A Graph is a (listof (list Sym Set))
;; Requires: keys must be unique

;; A DAG is a Graph
;; Requires: graph must be acyclic
```

Finding the out-neighbours of a node.

```
;; Look up the out-neighbours of a node
;; neighbours: Sym Graph -> Set
(define (neighbours v g)
  (cond
    [(empty? g) empty]
    [(symbol=? v (first (first g))) (second (first g))]
    [else (neighbours v (rest g))]))

(check-expect (neighbours 'D g) (list 'F 'J))
(check-expect (neighbours 'Z g) empty)
```

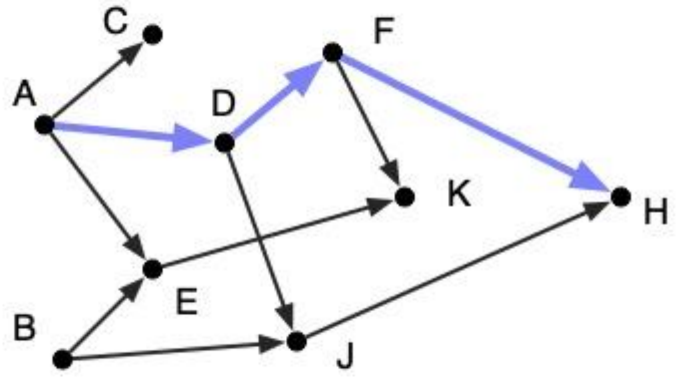
L20.2 Finding a path in a DAG

Finding a path from origin to destination nodes

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.

If no path exists, the function produces the `empty` path.



Example test cases

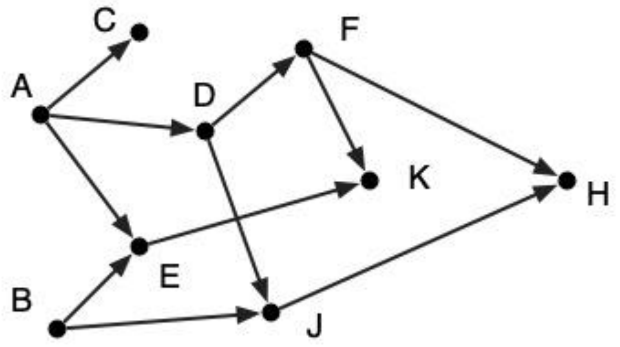
Note that there may be multiple paths between the origin and the destination. Finding any of them is acceptable.

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

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

`(find-path 'C 'H g)` ⇒ **`empty`**

`(find-path 'A 'A g)` ⇒ **`'(A)`**

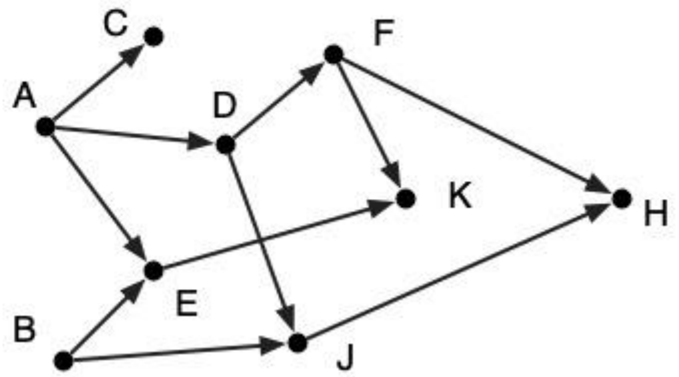


Cases for `find-path`

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.

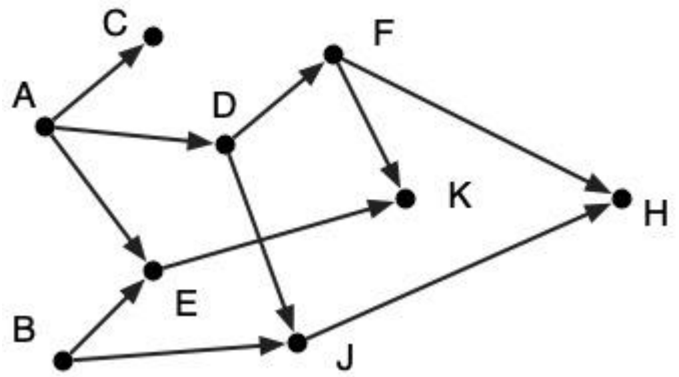
Each out-neighbour defines a subproblem (finding a path from it to the destination).



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.

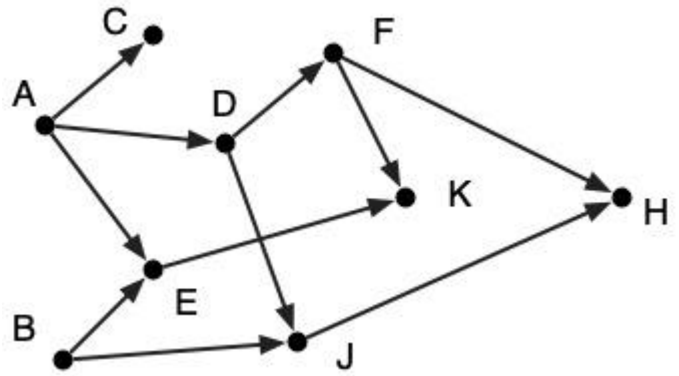


Backtracking algorithms

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.

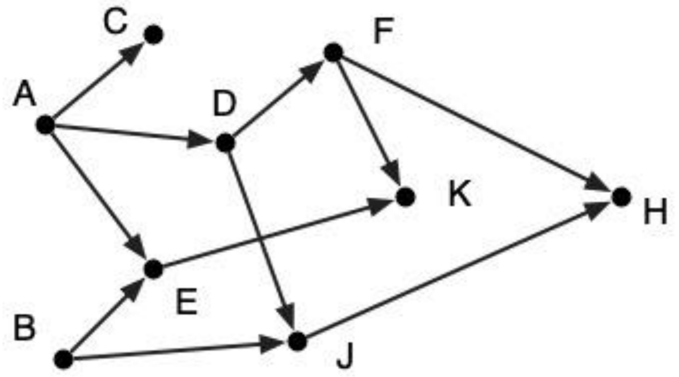


Backtracking in our example

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.

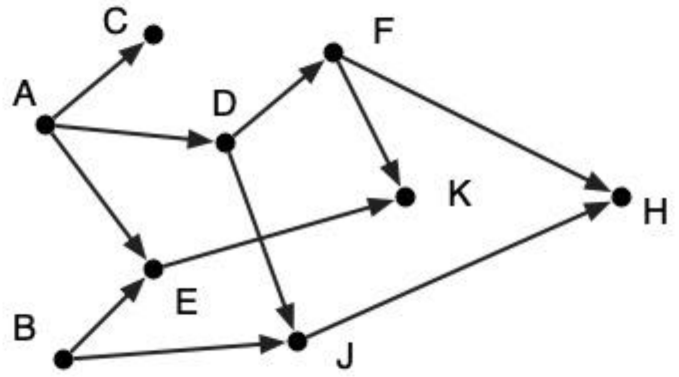


Exploring the list of out-neighbours

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 a function **`find-path/list`** that 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.

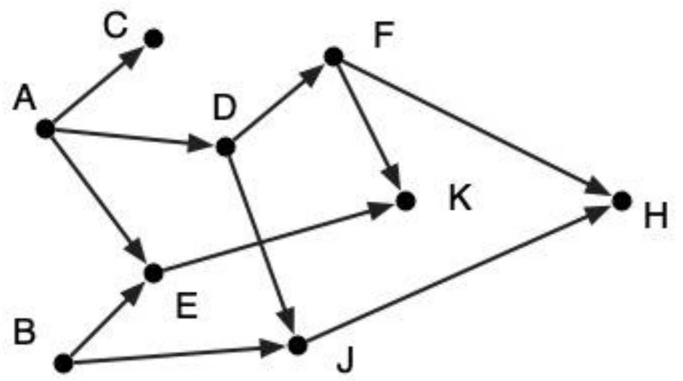


Mutual recursion

This is the same recursive pattern we saw when processing expression 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`**.



Finding a path from a node

```
;; Find a path from orig to dest in a DAG,
;; producing empty if no path exists
;; find-path: Sym Sym DAG -> (listof Sym)
(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 [(empty? path) empty]
                      [else (cons orig path)]))]))
```

Finding a path from a list of nodes

```
;; Find a path from a list of nodes to dest in a DAG,
;; producing empty if no path exists
;; find-path/list: (listof Sym) Sym DAG -> (listof Sym)
(define (find-path/list nbrs dest g)
  (cond [(empty? nbrs) empty]
        [else (local [(define path
                        (find-path (first nbrs) dest g))]
                (cond [(empty? path)
                       (find-path/list (rest nbrs) dest g)]
                      [else path]))]))
```

Test cases

```
(check-expect (find-path 'A 'H g) '(A D F H))
(check-expect (find-path 'D 'H g) '(D F H))
(check-expect (find-path 'C 'H g) empty)
(check-expect (find-path 'A 'A g) '(A))
```

In the first two cases, we generate one of several possible paths. Any of them is acceptable.

Termination of `find-path`

In a directed acyclic graph, any path with a given origin will recurse on its 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 contradicting our requirement that the graph is a 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).

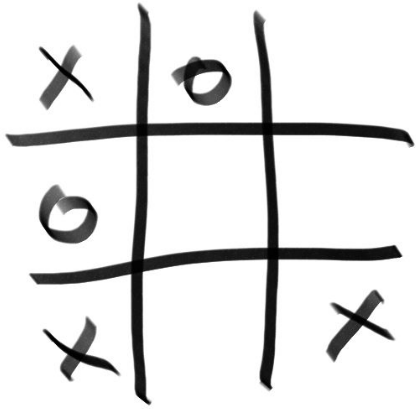
As a result, `find-path` always terminates.

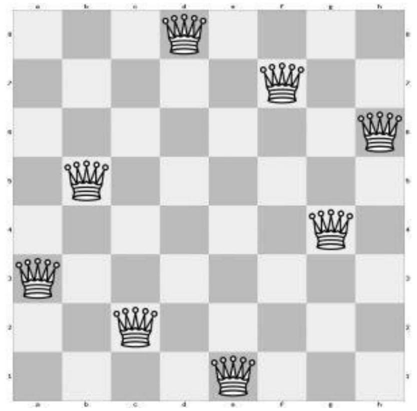
L20.3 Backtracking in implicit graphs

Backtracking in implicit graphs

The only places where real computation occurs 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”.





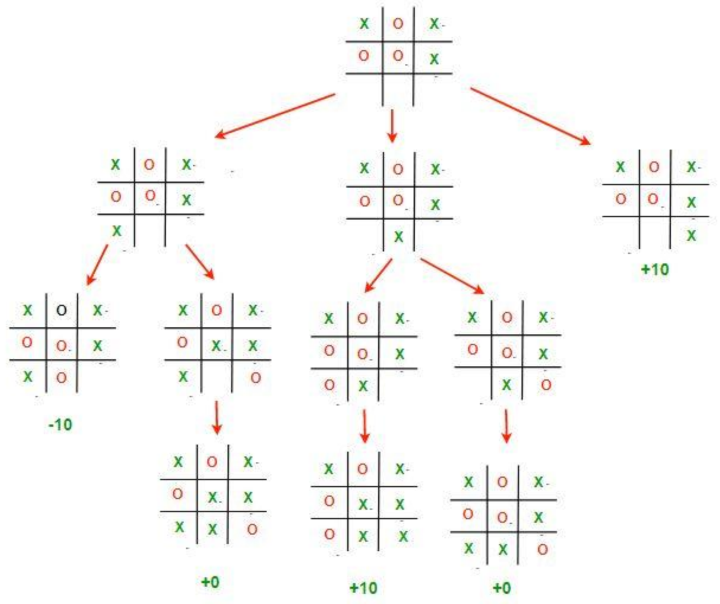
https://www.puzzleprime.com/brain-teasers/deduction/eight-queens-puzzle/

Backtracking in implicit graphs

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).



Backtracking in implicit graphs

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 some artificial intelligence programs, though they generally add heuristics to determine which neighbour to explore first, or which ones to skip because they appear unpromising.

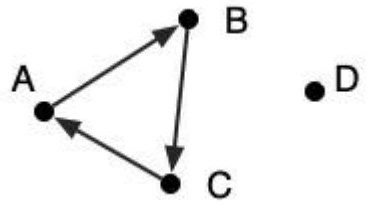
L20.4 Directed graphs with cycles

Non-termination of **`find-path`** with cycles

Suppose we try to use **`find-path`** on a directed graph with cycles.

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

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



Handling cycles

We can use an accumulator 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`**.

Finding a path with a list of visited nodes

```
;; find-path/acc: Sym Sym Graph (listof Sym)
;;                -> (listof Sym)
(define (find-path/acc orig dest g seen)
  (cond [(symbol=? orig dest) (list dest)]
        [else (local [(define nbrs (neighbours orig g))
                      (define path
                        (find-path/list nbrs dest g
                                        (cons orig seen)))]
                (cond [(empty? path) empty]
                      [else (cons orig path)]))]))
```

Finding a path with a list of visited nodes

```
;; find-path/list: (listof Sym) Sym Graph (listof Sym)
;;                    -> (listof Sym)
(define (find-path/list nbrs dest g seen)
  (cond [(empty? nbrs) empty]
        [(in? (first nbrs) seen)
         (find-path/list (rest nbrs) dest g seen)]
        [else (local [(define path
                        (find-path/acc (first nbrs) dest g seen))]
                (cond [(empty? path)
                       (find-path/list (rest nbrs) dest g seen)]
                      [else path]))]))
```

Wrapper function

```
;; Find a path from orig to dest in a Graph,
;; producing empty if no path exists
;; find-path: Sym Sym Graph -> (listof Sym)
(define (find-path orig dest g)
  (find-path/acc orig dest g empty))

(check-expect (find-path 'A 'H g) '(A D F H))
(check-expect (find-path 'D 'H g) '(D F H))
(check-expect (find-path 'C 'H g) empty)
(check-expect (find-path 'A 'A g) '(A))
(check-expect (find-path 'A 'C cg) '(A B C))
```

The accumulator in **`find-path/list`**

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

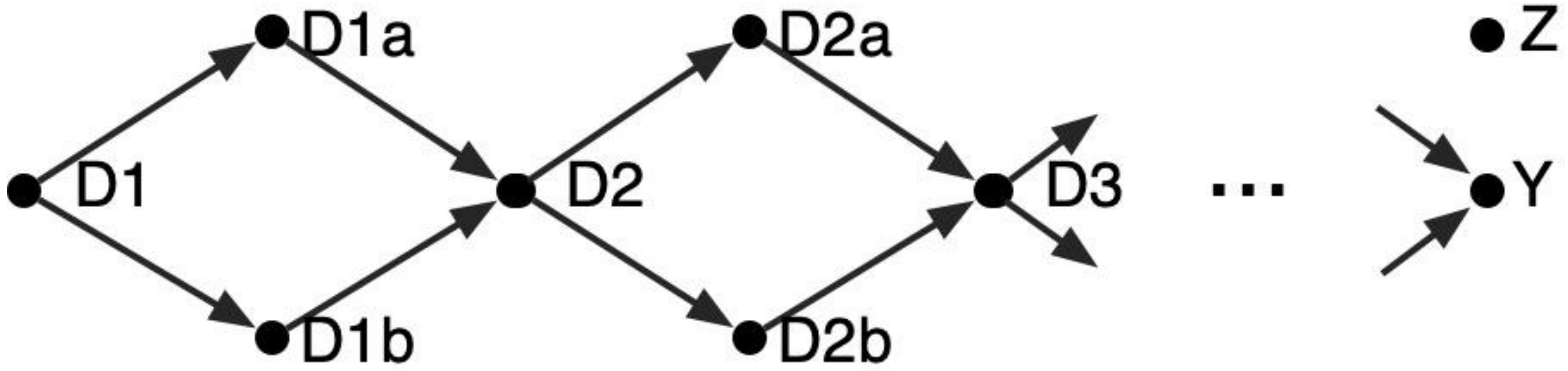
Adding to the accumulator is done in **`find-path/acc`** which applies **`find-path/list`** to the list of neighbours of an origin node.

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

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.

For example, if there are *d* diamonds in the graph below, then there are 3*d* + 2 nodes in the graph, but 2^d paths from D1 to Y, all of which will be explored.



L20 Summary

L20: You should know

- 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 directed acyclic graphs.
- You should understand and be able to implement backtracking on directed graphs with cycles using an accumulator.

L20: Allowed constructs

Newly allowed constructs:
none

Previously allowed constructs:

```
( ) [ ] + - * / = < > <= >= ; '
abs acos add1 and append asin atan boolean? build-list char?
char=? char<? check-expect check-within cond cons cons? cos
define e else empty empty? even? exp expt false filter first
foldl foldr inexact? integer? lambda length list list? local
log map max min not number? odd? or pi quotient rational?
remainder rest reverse second sin sort sqr sqrt string?
string=? string<? string->list list->string string-append
sub1 symbol? symbol=? tan third true zero?
listof Any anyof Atom Bool Char Int Nat Num Rat Str Sym
```