Graphs

Readings: Section 28

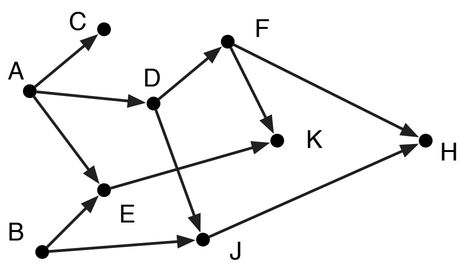
Topics:

- Directed graphs
- Representing graphs
- Finding routes
- Termination
- Improving find-route
- Making find-route more efficient

Directed graphs

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

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



We have seen such graphs before.

Evolution trees and expression trees were both directed graphs of a special type.

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 vertices $v_1, v_2, \ldots, v_k$ is a path or **route** of length $k - 1$ if (v_1, v_2), (v_2, v_3), $\ldots$, (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.

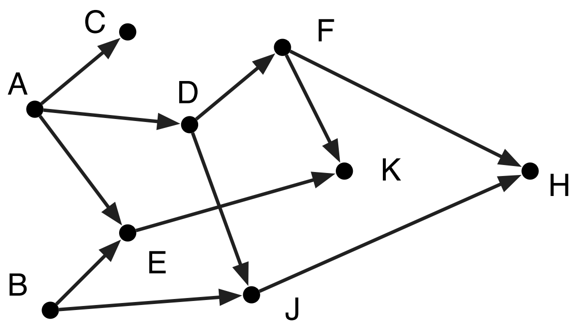
This is called the **adjacency list** representation.

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

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



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 a (listof (list Node (listof Node)))
```

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 nc
          (graph-template (rest G)) . . . )]))
```

Finding routes

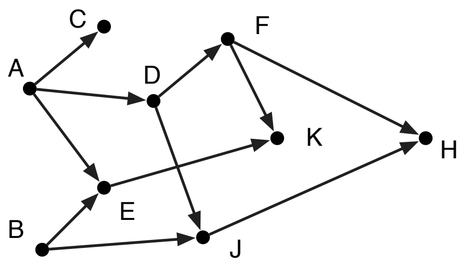
A path in a graph can be represented by the list of nodes on the path.

We wish to design a function find-route 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.

First we create an auxiliary function neighbours that consumes a node and a graph and produces the list of out-neighbours of the node.

Neighbours in our example

```
(neighbours 'A G) ⇒ (list 'C 'D 'E)
(neighbours 'H G) ⇒ empty
```



```
;; (neighbours v G) produces list of neighbours of v in G
;; neighbours: Node Graph → (listof Node)
;; requires: v is a node in G
(define (neighbours v G)
  (cond [(symbol=? v (first (first G))) (second (first G))]
        [else (neighbours v (rest G))]))
```

Cases for find-route

Structural recursion does not work for find-route; we must use generative recursion.

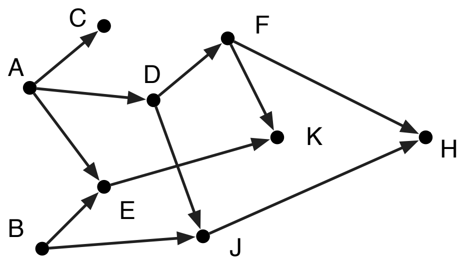
If the origin equals the destination, the path consists of just this node.

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 route from it to the destination).

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

If we knew a route 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 route from an origin to a destination.

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

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

In our example, we can see the “backtracking” since the search for a route 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 vertex (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.

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

All those out-neighbours are collected into a list associated with that node.

This suggests writing find-route/list which does this for the entire list of out-neighbours.

The function find-route/list will apply find-route to each of the nodes on that list until it finds a route to the destination.

This is the same recursive pattern that we saw in the processing of expression trees (and descendant family trees, in HtDP).

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

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

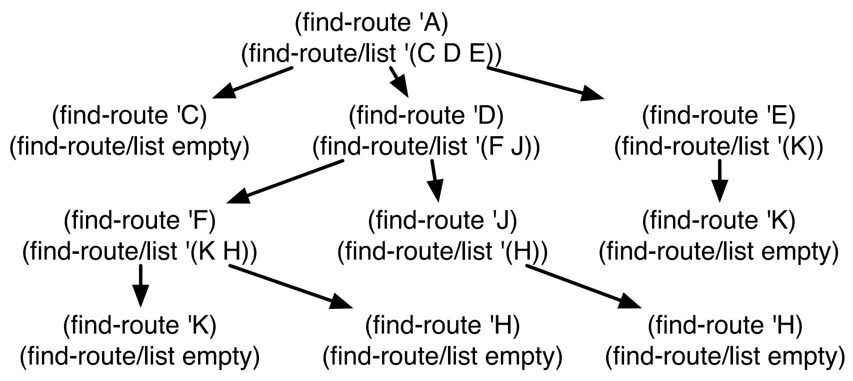
```
;; (find-route orig dest G) finds route from orig to dest in G if it exists
;; find-route: Node Node Graph → (anyof (listof Node) false)
(define (find-route orig dest G)
  (cond [(symbol=? orig dest) (list dest)]
        [else (local [(define nbrs (neighbours orig G))
                      (define route (find-route/list nbrs dest G))]
                (cond [(false? route) false]
                      [else (cons orig route)]))]))
```

```
;; (find-route/list los dest G) produces route from
;;     an element of los to dest in G, if one exists
;; find-route/list: (listof Node) Node Graph → (anyof (listof Node) false)
(define (find-route/list los dest G)
  (cond [(empty? los) false]
        [else (local [(define route (find-route (first los) dest G))]
                (cond [(false? route)
                       (find-route/list (rest los) dest G)]
                      [else route]))]))
```

If we wish to trace find-route, 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 but the computation in find-route is better visualized as a tree.

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

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



Backtracking in implicit graphs

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.

Example: nodes represent configurations of a board game (e.g. peg solitaire), edges represent legal moves.

The graph is acyclic if no configuration can occur twice in a game.

In another example, nodes could represent partial solutions of some problem (e.g. a sudoku puzzle, or the puzzle of putting eight mutually nonattacking queens on a chessboard).

Edges represent ways in which one additional element can be added to a solution.

The graph is naturally acyclic, since a new element is added with every edge.

The find-route 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 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-route (no cycles)

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

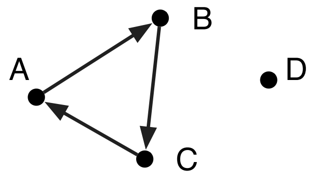
Thus, the origin will never be explored in any later call, and thus the subproblem is smaller. Eventually, we will reach a subproblem of size 0 (when all reachable nodes are treated as the origin).

Thus find-route always terminates for directed acyclic graphs.

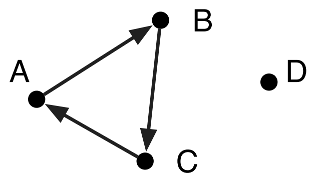
Non-termination of find-route (cycles)

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

Consider the graph '((A (B)) (B (C)) (C (A)) (D ())). What if we try to find a route from A to D?



$$\begin{tikzcd} \begin{array}{c}\text{(find-route 'A)}\\ \text{(find-route/list '(B))}\end{array} \arrow[d] \\ \begin{array}{c}\text{(find-route 'B)}\\ \text{(find-route/list '(C))}\end{array} \arrow[d] \\ \begin{array}{c}\text{(find-route 'C)}\\ \text{(find-route/list '(A))}\end{array} \arrow[d] \\ \begin{array}{c}\text{(find-route 'A)}\\ \text{(find-route/list '(B))}\\ \dots\end{array} \end{tikzcd}$$



Improving find-route

We can use accumulative recursion to solve the problem of find-route 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-route/list.

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

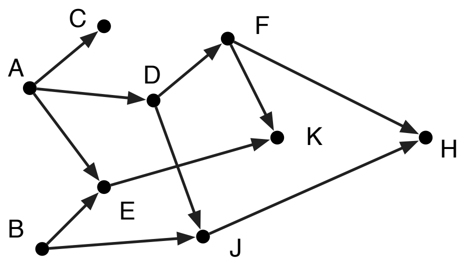
The code for find-route/list does not add anything to the accumulator (though it uses the accumulator).

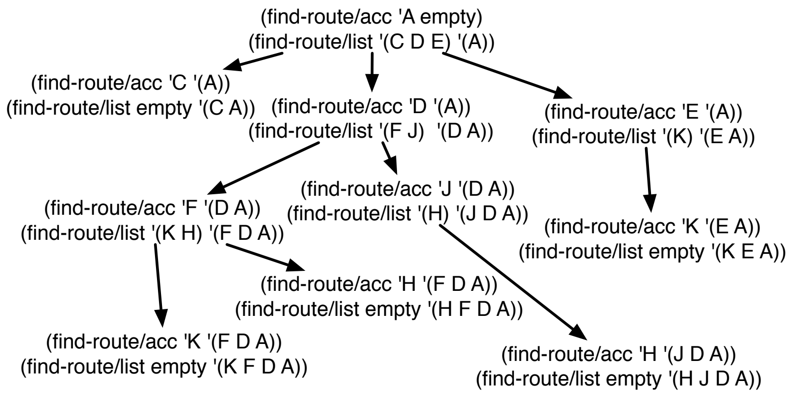
Adding to the accumulator is done in find-route/acc which applies find-route/list to the list of neighbours of some origin node.

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

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

Revisiting our example





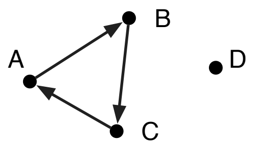
```
(find-route/acc 'A empty)
(find-route/list '(C D E) '(A))

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

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

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

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

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

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

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

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

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

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

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-route terminates.

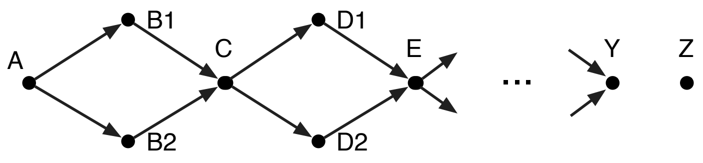


```
(find-route/acc 'A empty)
(find-route-list '(B) '(A))
            ↓
(find-route/acc 'B '(A))
(find-route-list '(C) '(B A))
            ↓
(find-route/acc 'C '(B A))
(find-route-list '(A) '(C B A))
no further recursive calls
```

In practice, we would write a wrapper function for users which would avoid their having to specify the initial value of the accumulator.

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-route 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 A to Y, all of which will be explored.

Making find-route/acc efficient

Applying find-route/acc to origin A results in find-route/list being applied to '(B1 B2), and then find-route/acc being applied to origin B1.

There is no route from B1 to Z, so this will produce false, but in the process, it will visit all the other nodes of the graph except B2 and Z.

find-route/list will then apply find-route/acc to B2, which will visit all the same nodes.

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

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 as the sentinel value. However, we must be able to distinguish this list from a successfully found route (also a list of nodes).

Remembering what the list of nodes represents

We will make a new *type* that will store both the list of nodes (either those nodes which have been visited, or the nodes along the successful path) as well as a boolean value indicating what the list of nodes represents.

```
(define-struct routepair (valid-path? nodes))
;; a RoutePair is a (make-routepair Bool (listof Node))
;; find-route/list: ... → RoutePair
(define (find-route/list los dest G visited)
  (cond [(empty? los) (make-routepair false visited)]
        [(member? (first los) visited)
         (find-route/list (rest los) dest G visited)]
        [else (local [(define route (find-route/acc (first los)
                                                    dest G visited))]
                (cond [(not (routepair-valid-path? route))
                       (find-route/list (rest los) dest G
                                        (routepair-nodes route))]
                      [else route]))]))
```

```
;; find-route/acc: Node Node Graph (listof Node)
;; → (anyof (listof Node) RoutePair)
(define (find-route/acc orig dest G visited)
   (cond [(symbol=? orig dest) (make-routepair true (list dest))]
      [else (local [(define nbrs (neighbours orig G))
                    (define route (find-route/list nbrs dest G
                                               (cons orig visited)))]
              (cond [(not (routepair-valid-path? route)) route]
                    [else (make-routepair true
                                 (cons orig
                                   (routepair-nodes route)))]))]))
```

```
;; find-route: Node Node Graph → (anyof (listof Node) false)
(define (find-route orig dest G)
  (local [(define route (find-route/acc orig dest G empty))]
    (cond [(routepair-valid-path? route) (routepair-nodes route)]
          [else false])))
```

With these changes, find-route runs much faster on the diamond graph.

In future courses we will see how to make find-route 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).

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 performance differences in the various versions of find-route.