Directed graphs
Finding routes
Termination
Improving find-route

Graphs
Graphs

• Readings: Section 28
Directed graphs
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 \(v_1, v_2, ..., v_k\) is a path or route of length \(k - 1\) if \((v_1, v_2), (v_2, v_2), ..., (v_{k-1}, v_k)\) are all edges.

If \(v_1 = v_k\), this is called a cycle.

Directed graphs without cycles are called DAGs (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)))
CQ 1

For other types of graphs like evolution and binary search trees, we represented the whole structure by a single “root” node. What is the main reason that we do not use this kind of representation for graphs?

A. We could, but it would be less efficient.
B. We could, but it would require structures instead of lists.
C. Sometimes there is no “root node” from which all others can be reached.
D. There is always a “root node”, but it might be difficult to find it.
E. There could be cycles in the graph.
CQ 1

For other types of graphs like evolution and binary search trees, we represented the whole structure by a single “root” node. What is the main reason that we do not use this kind of representation for graphs?

A. We could, but it would be less efficient.
B. We could, but it would require structures instead of lists.
C. Sometimes there is no “root node” from which all others can be reached.
D. There is always a “root node”, but it might be difficult to find it.
E. There could be cycles in the graph.
The template for graphs

;; A Node is a Sym

;; A Graph is a (listof (list Node (listof Node)))

;; graph-template: Graph -> Any
(define (graph-template G)
  (cond
    [(empty? G) ...]
    [(cons? G) (...;
      ; first node in graph list
      ...(first (first G))
      ; list of adjacent nodes
      ...(listof-node-template (second (first G)))
      (graph-template (rest G)))]))
Finding routes
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) => '(C D E)
(neighbours 'H G) => empty
(define (neighbours v G)
  (cond
   [(symbol=? v (first (first G))) (second (first G))]
   [else (neighbours v (rest G))])))
Cases for find-route

Simple 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.
Cases for find-route

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

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

Each out-neighbour defines a sub-problem (finding a route from it to the destination).

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.
Backtracking algorithms

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.
Backtracking algorithms

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.
Backtracking algorithms

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.
Backtracking algorithms

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.