CS 60

Currying and Graph Algorithms

September 25, 2001

Review: Curried Functions

- Given the definition

```
scaleBy(Y) = (X) => Y*X;
```

we have

```
map(scaleBy(5), [1,2,3]) ==> [5,10,15]
scaleby(4)(7) => 28
```

- We could have defined

```
scale(Y, L) = map(scaleBy(Y), L);
```

Review: Curried Functions

```
scaleBy(Y) = (X) => Y*X;
mapster(F) = (L) => map(F, L);
```

```
map(even,[1,2,3,4]) ==> [0,1,0,1]
((L) => map(even,L))([1,2,3,4]) ==> [0,1,0,1]
(mapster(even))([1,2,3,4]) ==> [0,1,0,1]

((X) => 4*X)(7) ==> 28
(scaleby(4))(7) ==> 28
map(scaleBy(5), [1,2,3]) ==> [5,10,15]
```

A Useful Example

- An **association list** is a representation of a function (with a finite domain):

```
[[“Jan”,31],[“Feb”,28],[“Mar”,31],[“Apr”,30]]
```

- But we cannot simply **apply** this list as a function; we have to use `assoc` instead:

assoc(“Feb”, *... list above ...*) ==> [“Feb”, 28]

Making Fun of Association Lists

- We want a function
  `makeFun(Alist)`
  that converts an association list `Alist` to an ordinary function that can be applied, e.g.,

```
f = makeFun([[“Jan”,31],[“Feb”,28],[“Mar”,31]]);
f(“Feb”) ==> 28
```

Defining `makeFun`

- How to define `makeFun`?

Composing Functions

- The composition of two functions,
  f: T → U
  g: S → T
  is the function, call it h for now, such that
  h: S → U
  and for every x in S, h(x) = f(g(x))
- The composition is sometimes written using an operator o:
  f o g

Composing Functions

- If anonymous functions are supported, we don’t need the “call it h” aspect:

```
compose(f,g) = (x) => f(g(x))
```

or, equivalently,

```
compose(f,g)(x) = f(g(x))
```

Why `compose`?

- Composing can make code more efficient in some cases

```
map(f, map(g, L)) ==
```

Generating Curried Functions

- Define
  `curry(f)(X)(Y) = f(X, Y)`
- If `f` is a two-argument function then `curry(f)` is a curried version of this function.
  - Computes the same result as the given `f`, but `curry(f)` expects its argument to be given in separate applications rather than simultaneously.
  - `curry(f)(X)` is like “`f(X,•)`” in engineering or mathematics books.
- Example: `scale(K,L) = map(curry(*)(K),L);`

  the 2-argument multiply function

Searching a Graph

- Suppose we want to find whether there is a node with a certain property `P` reachable from a node in a graph.
- Rather than assume a specific representation such as an arc-list, we'll use **abstraction**:
- Assume we have a function `targets` such that
  `targets(Graph, Node)`
  is the list of targets of the node.

Finding a Node in a Graph

- Want to define
  `findNode(Graph, Node, P)`
  where `Node` is the starting node, `P` is a predicate on nodes.)
- Return value:
  - If a node is found, return a list containing just that node.
  - If no such node is found, return the empty list.

Simplifying the Problem

```
findNode(Graph, Node, P) =>
     P(Node) ? [Node];

findNode(Graph, Node, P) =>
     findFromList(Graph,
                  targets(Graph, Node),
                  P);
```

Starting From a List of Nodes

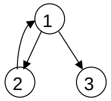
```
findFromList(Graph, [], P) => [ ];

findFromList(Graph, [Target|Targets], P) =>
     Found = findNode(Graph, Target, P),
     (Found != []) ?
         Found :
         findFromList(Graph, Targets, P);
```

Mutual Recursion

- The relationship between `findNode` and `findFromList` is that of **mutual recursion**.
  - `findNode` delegates work to `findFromList`
  - `findFromList` delegates work to `findNode`
- This approach seems natural in this problem.

Correctness

- The previous solution will work if the graph is acyclic.
  - If not acyclic, it may work in some cases, and loop infinitely in others.
- So it doesn't *really* work.
- How can we fix this?



Handling the Cyclic Case

- If our graph is **finite**, infinite looping can only occur when the **same node** recurs on a path.
- By **keeping track of the nodes on the path from the starting point**, we can check whether a node recurs.

`findNode` revised

```
findNode(Graph, Node, P) =>
     findNode(Graph, Node, P, []);

findNode(Graph, Node, P, Seen) =>
     P(Node) ? [Node];

findNode(Graph, Node, P, Seen) =>
     member(Node, Seen) ? [];

findNode(Graph, Node, P, Seen) =>
     findFromList(Graph,
                  targets(Graph, Node),
                  P, [Node|Seen]);
```

`findFromList` revised

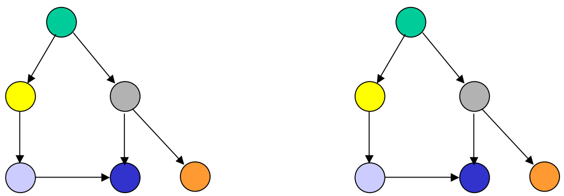
```
findFromList(Graph, [], P, Seen) => [ ];

findFromList(Graph,[Target|Targets],P,Seen) =>
     Found = findNode(Graph,Target,P,Seen),
     (Found != []) ?
        Found :
        findFromList(Graph, Targets, P,
                     [Target | Seen]);
```

- We include `Target` in the list above so that we never do a recursive `findNode` from a node more than once.
- However, we may still search from the same node more than once. Why? Is this preventable?

Varieties of Search

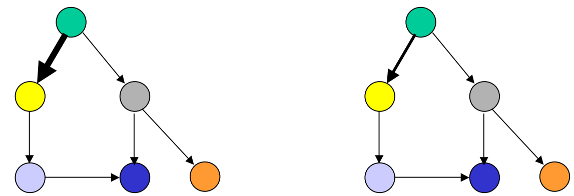
- The version of find presented previously is called **depth-first** search.
- The other prevalent form of find is **breadth-first** search.

Search Orders



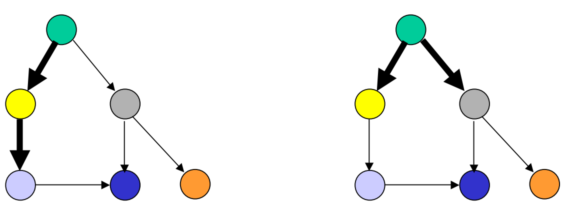
depth-first: **breadth-first:**

Search Orders



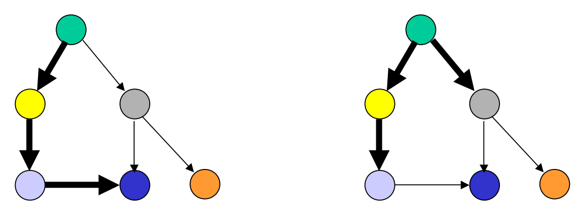
depth-first: **breadth-first:**

Search Orders



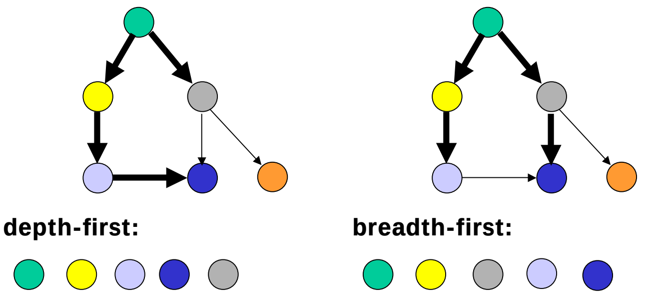
depth-first: **breadth-first:**

Search Orders

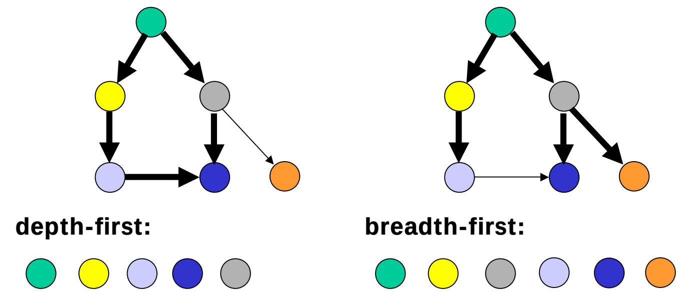


depth-first: **breadth-first:**

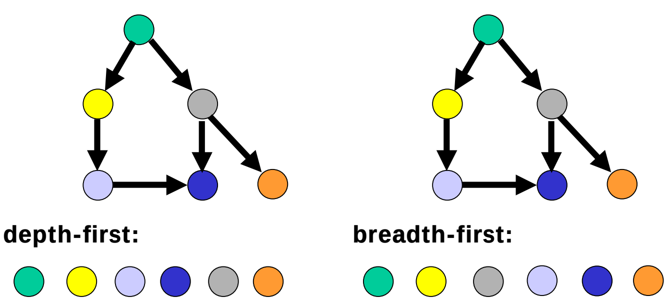
Search Orders



Search Orders



Search Orders



Comparative Strengths

- Breadth-first has the advantage of finding the ***shortest*** path to a node with the desired property.
- Depth-first is easier to implement and is more space-efficient.