Depth-First and Breadth-First Search

CS 5010 Program Design Paradigms
“Bootcamp”
Lesson 9.2
Introduction

• In this lesson, we'll return to the problem of searching in a graph.
• When we're searching for all the nodes reachable from a given node, the order in which we search doesn't matter— we have to search everything anyway.
• But if we're searching for a specific node or set of nodes, then the order may make a big difference in running time.
Outline

• In this lesson, we'll write two variations on searching in a graph:
  – depth-first search
  – breadth-first search

• We'll see how the invariants help us keep track of the different variables in our calls.
We'll start with `path?` from 08-3-reachability.rkt

```scheme
(define (path? graph src tgt)
  (local
    ((define (reachable-from? newest nodes)
        ;; RETURNS: true iff there is a path from src to tgt in graph
        ;; INVARIANT: newest is a subset of nodes
        ;; AND:
        ;; (there is a path from src to tgt in graph)
        ;; iff (there is a path from newest to tgt)
        ;; STRATEGY: generative recursion
        ;; HALTING MEASURE: the number of graph nodes _not_ in 'nodes'
        (cond
          [(member tgt newest) true]
          [else (local
                     ((define candidates (set-diff
                                          (all-successors newest graph)
                                          nodes)))
                      (cond
                        [(empty? candidates) false]
                        [else (reachable-from? candidates
                                  (append candidates nodes))])))])
    (reachable-from? (list src) (list src))))
```
Our first step is to break out the inner function

• This makes the inner function a little less scary
• We will have to pass more arguments, so the purpose statement will get a little larger.
• But don't worry, we haven't really changed anything.
Contract and Purpose Statement

;; ListOfNodes ListOfNodes Node Graph -> Boolean
;; GIVEN:
;; 1. The list 'nodes' of all the nodes we've seen
;; 2. The list of nodes whose successors we haven't taken
;; 3. The target node 'tgt' that we are trying to reach
;; 4. The graph we are searching
;; RETURNS: Is tgt reachable from any of the nodes in
;;   'nodes'?
;; INVARIENT: newest is a subset of nodes
;; AND:
;;  (there is a path from src to tgt in graph)
;;  iff (there is a path from newest to tgt)
;; HALTING MEASURE: the number of graph nodes _not_ in
;;   'nodes'
reachable-from?

(define (reachable-from? newest nodes tgt graph)
  (cond
    [(member tgt newest) true]
    [else (local
            ((define candidates (set-diff
                                 (all-successors newest graph)
                                  nodes)))
              (cond
                [(empty? candidates) false]
                [else (reachable-from?
                        candidates
                        (append candidates nodes)
                        tgt
                        graph)]))])))
Defining **path?** in terms of **reachable-from?**

;; Strategy: Function composition
(define (path?.v1 graph src tgt)
  (reachable-from?
    (list src) (list src) tgt graph))
Refining reachable-from?

• In order to control the order in which nodes are explored, we'll stop using *(all-successors newest)* and take the successors of each node in *newest* one at a time.

• We'll call our new function *reachable-from-dfs?* . (We'll explain the name later)

• What are the possibilities?
Possibilities #1-2

• **tgt** is already in **newest**. In that case we've found the node that we're looking for, and the answer is **true**.

• **newest** is empty. In that case, there are no nodes left to explore, so the answer must be **false**.
What else could happen?

- Otherwise, we'll let candidates be the successors of (first newest) that are not already in nodes. (set-diff (successors (first newest) graph) nodes)
- This guarantees that none of the nodes in candidates are already in nodes.
- And since newest is a subset of nodes, it means that none of the nodes in candidates are in newest, either.
- Now what?
Possibility #3

• **candidates** is empty
  – in that case, we know that **tgt** is not reachable from **(first newest)**.
  – so if **tgt** is reachable from **newest**, it must be reachable from **(rest newest)**.

• So we add a cond-line that says

  ```lisp
  [(empty? candidates)
   (reachable-from-dfs?
    (rest newest) nodes tgt graph)]
  ```
Possibility #4

- **candidates** is non-empty.
- So we need to add **candidates** to our list **newest** of nodes to explore.
- We also need to remove **(first newest)**, since we've explored it.
- We also need to add **candidates** to **nodes**, in order to maintain the invariant that **newest** is a subset of **nodes**.
Possibility #4 (cont'd)

• So our cond line will be:

```
[else
  (reachable-from-dfs?
    (append candidates (rest newest))
    (append candidates nodes)
    tgt
    graph)]
```

Get the next value of `newest` by removing `(first newest)` and adding `candidates`.

Add candidates to nodes to maintain the invariant.
(define (path-dfs? graph src tgt)
  (reachable-from-dfs?
   (list src) (list src) tgt graph))
Why did we call this dfs?

• We add the newly-discovered nodes candidates to the front of the list of nodes to be explored.
• So the nodes that we just discovered get explored first.
• This is called depth-first search.
Let's see this in action

• Here is a tree, with the nodes numbered in the order this function will discover them.
Alas, this is only “almost” DFS

The order in which this algorithm finds the nodes

Real depth-first search would find the node labelled 3 here as the left son of 2, not as the third son of 1. See 09-3a-reachability.rkt, which contains a detailed discussion of dfs.
Breadth-First search

• The other possibility is to put the new nodes at the END of the worklist (the list **newest**)
• This explores nodes strictly in the order of their distance from the starting nodes.
• This is called **breadth-first search**.
reachable-from-bfs?

(define (reachable-from-bfs? newest nodes tgt graph)
  (cond
    [(member tgt newest) true]
    [(empty? newest) false]
    [else (local
          ((define candidates (set-diff
                                  (successors (first newest) graph)
                                  nodes)))
            (cond
              [(empty? candidates)
                (reachable-from-dfs?
                 (rest newest)
                 nodes tgt graph)]
              [else (reachable-from-dfs?
                     (append (rest newest) candidates)
                     (append candidates nodes) tgt graph)]))]))

Only difference: put the candidates at the END of the list to be explored
The same tree, in bfs order
What if there were cycles?

This edge creates a cycle

NO PROBLEM: When we discover 2 in (successors 6), it will already be in nodes, so it will be filtered out by the set-diff.

The halting measure assures us that the number of nodes not in nodes is strictly decreasing, so there can't possibly be an infinite loop.
Choosing between bfs and dfs

• If you know that the solution is close to the root, then bfs is better.
• If your tree is very broad, then maybe dfs is better.
• Go look at an algorithms book for more examples.
Variations

• If you knew more about your problem, you could call a function to choose the order in which to explore your nodes

• So you might write something like:

```lisp
(reachable-from-bfs?
 (reorder-candidates
  (rest newest) candidates)
 (append candidates nodes) tgt
 graph)
```
Variations (2)

• If you had a set of targets instead of a single target, you could return the target you actually found.
• Or you could keep track not just of the nodes you've reached, but the path you took to get to each of them.
Summary

• We've written two variations on searching in a graph:
  – depth-first search
  – breadth-first search

• We've seen how the invariants help us keep track of the different variables in our calls.