Searching in a Graph

CS 5010 Program Design Paradigms
“Bootcamp”
Lesson 8.4
Many problems in computer science involve directed graphs.

Generative recursion is an essential tool for computing on graphs.

In this lesson we will design a program for an important problem on graphs, using generative recursion.

The algorithm we will develop has many other applications.
Learning Objectives

• At the end of this lesson you should be able to:
  – explain what a directed graph is, and it means for one node to be reachable from another
  – explain what a closure problem is
  – explain the worklist algorithm
  – write similar programs for searching in graphs.
What's a graph?

• You should be familiar with the notion of a graph from your previous courses.
• A graph consists of some nodes and some edges.
• We will be dealing with directed graphs, in which each edge has a direction. We will indicate the direction with an arrow
A Graph

nodes: A, B, C, etc.
edges:
  (A,B), (A,C),(A,D), etc.
successors of a node

The successors of a node are the nodes that it can get to by following one edge.

\[(\text{successors A}) = \{B, C, D\}\]
\[(\text{successors D}) = \{C, F\}\]
successors of a set of nodes

all-successors of a set of nodes are all the successors of any of the nodes in the set

(all-successors {}) = {}

(all-successors {A,D}) = {B,C,D,F}
A path is a sequence of nodes that are connected by edges. Notice that the node A by itself is a path, since there are no edges to check. On the other hand, (A,A) is not a path, since there is no edge from A to itself.

**Paths in a Graph**

- **Paths:**
  - (A, C, E)
  - (B, C, E, G)
  - (A, D, C, E)
  - (A)

- **Non-paths:**
  - (D, A)
  - (A, C, G)
  - (A, C, D, E)
  - (A, A)
This graph has a \textit{cycle}: a path from the node B to itself. Graphs without cycles are said to be \textit{acyclic}.

For this lesson, our graphs are allowed to have cycles.
Reachability

One node is reachable from another if there is a path from the one node to the other.

Nodes reachable from D: \{B,C,D,E,F,G\}
Not reachable: \{A\}

D is reachable from itself by a path of length 0, but not by any other path.
Another classic application of generative recursion

reachables :

    Graph SetOfNode -> SetOfNode

Returns the set of nodes in the given graph that is reachable from the given set of nodes.
Definition

A node $t$ is reachable from a node $s$ iff either

1. $t = s$

2. there is some node $s'$ such that
   1. $s'$ is a successor of $s$
   2. $t$ is reachable from $s'$
What does this definition tell us?

• If S is a set of nodes, then (reachables S) has the property that:
  – IF node n is in (all-successors (reachables S))
  – THEN n is already in (reachables S).

• Why? Because if n is a successor of a node reachable from S, then n is itself reachable from S.
Another way of looking at this:

- If $S$ is a set of nodes, then
  - $(\text{reachables } S)$ is the smallest set of nodes such that $(\text{all-successors } S)$ is a subset of $S$. 
Growing (reachables $S$): not done yet

not closed under successors: more reachables to be found!
Growing (reachables S): done!

(reachables S): closed under successors
Closure problems

• This is called a "closure problem": we want to find the smallest set $S$ which is closed under some operation

• In this case, we want to find the smallest set that contains our starting set of nodes, and which is closed under all-successors.
Assumptions

• We assume we've got data definitions for Node and Graph, and functions
  – node=? : Node Node -> Boolean
  – successors :
    Node Graph -> SetOfNode
  – all-successors :
    SetOfNode Graph -> SetOfNode
• We also assume that our graph is finite.
Initial Solution

;; reachables: SetOfNode Graph -> SetOfNode
;; GIVEN: A set of nodes in a graph
;; RETURNS: the set of nodes reachable from the starting nodes
;; STRATEGY: Generative Recursion
;; TERMINATION ARGUMENT: At every recursive call, the set of
;; nodes NOT in 'nodes' decreases.
(define (reachables nodes graph)
  (local
   ((define candidates (all-successors nodes graph)))
   (cond
    [(subset? candidates nodes) nodes]
    [else (reachables
             (set-union candidates nodes) graph)])))

if 'nodes' is closed under all-successors, then we're done

Otherwise, add the candidates to the nodes, and try again
Problem with this algorithm

• We keep looking at the same nodes over and over again:
  – we always say *(all-successors nodes)*, but we've seen most of those nodes before.
A Better Idea: keep track of which nodes are new

only need to explore nodes in this region— all others are accounted for.
Do this with an extra argument and an invariant

newest = the most recently added elements of 'nodes'
Version with invariant

;; reachables1 : SetOfNode SetOfNode Graph
;; GIVEN: two sets of nodes, 'nodes' and 'newest' in a graph
;; WHERE: newest is a subset of nodes
;; AND: newest is the most recently added set of nodes
;; RETURNS: the set of nodes reachable from 'nodes'.
;; STRATEGY: Generative Recursion
;; TERMINATION ARGUMENT: At every recursive call, the set of nodes NOT
;; in 'nodes' decreases.
(define (reachables1 nodes newest graph)
  (local
    ((define candidates (set-minus
                        (all-successors newest graph) nodes)))
     (cond
      [(empty? candidates) nodes]
      [else (reachables1
              (append candidates nodes) candidates graph)]))))

Since candidates is disjoint from nodes, we can replace the set-union with append.
Initializing the invariant

;;; we initialize newest to nodes since
;;; initially all the nodes are new.

(define (reachables nodes graph)
  (reachables1 nodes nodes nodes graph))
This is called the "worklist" algorithm

• It is used in many applications
  – in compiler analysis
  – in AI (theorem proving, etc.)
You could use this to define \texttt{path}?

\texttt{;; path? : Graph Node Node -> Boolean
;; Returns true if there is a path in graph
;; from src to tgt
;; STRATEGY: function composition
(define (path? graph src tgt)
  (member tgt (reachables (list src) graph)))}
But for that, you don't need to build the whole set

(define (path? graph src tgt)
  (local
    ((define (reachable-from? newest nodes)
      ;; Returns true if there is a path in graph from src to tgt
      ;; INVARIANT: newest is a subset of nodes
      ;; AND:
      ;; (there is a path from src to tgt in graph)
      ;; iff (there is a path from some node in 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))))

Look carefully at this invariant.

Just watch for tgt to show up in newest

Why is the invariant true again at this call?

Can you check to see that the invariant is true at this call?
What if my function doesn't always halt?

You must deliver a termination argument for each function that uses generative recursion. This has one of two forms:

1. The function produces a solution for all problems because [show halting measure] is always non-negative and gets smaller at every recursive call

2. The function does not terminate on some input problems, example: [put here a specific example that does not terminate].
Another topic: changing the data representation

;; reachables: SetOfNode Graph -> SetOfNode
(define (reachables nodes graph)
  (local
    ((define candidates (all-successors nodes graph)))
    (cond
      [(subset? candidates nodes) nodes]
      [else (reachables (set-union candidates nodes) graph)])))

Notice that the only thing we do with graph is to pass it to all-successors.
So let’s pass in the graph’s all-successors function

;; reachables: SetOfNode (SetOfNode -> SetOfNode)
;; -> SetOfNode
(define (reachables nodes all-successors-fn)
  (local
    ((define candidates (all-successors-fn nodes)))
    (cond
      [(subset? candidates nodes) nodes]
      [else (reachables
        (set-union candidates nodes)
        all-successors-fn)])))
How do you build an all-successors-fn?

;;; You could do it from a data structure:

;;; Graph -> (SetOfNode -> SetOfNode)
(define (all-successors-fn g)
  (lambda (nodes)
    (all-successors nodes g))))
Or you could avoid building the data structure entirely

• Just define a successors function, and then define all-successors by HOFC

• Good thing to do if your graph is very large—e.g. Rubik’s cube.
Example of an “implicit graph”

;;; Int -> SetOfInt
Returns the list n’s successors in the implicit graph.
;;; For this graph, this is always a set (no repetitions)
(define (successors1 n)
  (if (<= n 0)
    empty
    (local
      ((define n1 (quotient n 3)))
      (list n1 (+ n1 5))))))
;; all-successors1 : SetOfInt -> SetOfInt
Returns the set of all successors of ns in the implicit graph represented by ns.
(define (all-successors1 ns)
  (unionall (map successors1 ns)))

Here’s a function you could pass to reachables.
Summary

• We've applied Generative Recursion to an important problem: graph reachability
• We considered the functions we needed to write on graphs in order to choose our representation(s).
• We used generative recursion with a list of subproblems
• We used list abstractions to make our program easier to write
Learning Objectives

• You should now be able to:
  – explain what a directed graph is, and it means for one node to be reachable from another
  – explain how the function for reachability works.
  – explain what a closure problem is
  – explain the worklist algorithm
  – write similar programs for searching in graphs.