Cycle detection

Goldsmiths Computing

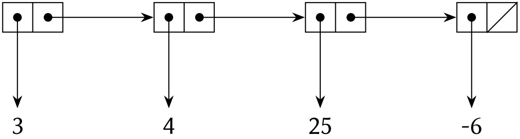
Motivation

Did your SLList code suffer from baffling infinite loops at any point?

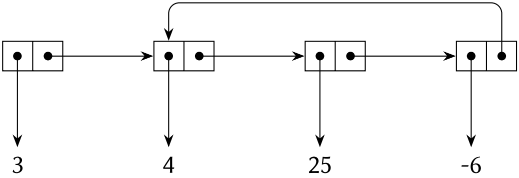
Definition

Cycle detection algorithms detect whether there are loops in a graph, graph, or repeated values in a function with same domain and range, and where and how long those loops are.

Linked list implementation



SET-REST!(REST(REST(REST(list))), REST(list))



Naïve circularity detection

Algorithm: at each step, check all previous steps for repeat

Helper function

```
function IS-IN?(sequence,node,end)
    j ← 0
    x ← sequence
    while j < end do
        if x = node then
            return true
        end if
        x ← REST(x)
        j ← j + 1
    end while
    return false
end function
```

Naïve circularity detection

Algorithm: at each step, check all previous steps for repeat

Main function

Require: sequence :: list
  node ← REST(sequence)
  end ← 0
  while node ≠ NIL **do**
    end ← end + 1
    if IS-IN?(sequence,node,end) **then**
      return true
    end if
    node ← REST(node)
  end while
  return false

Naïve circularity detection

Algorithm: at each step, check all previous steps for repeat

Complexity analysis

Space

No extra space required

$$\Longrightarrow \Theta(1)$$

Time

$$T_k = T_{k-1} + \Theta(k)$$

If there is no cycle, the algorithm traverses the entire list, checking an increasing amount of the entire list each time

$$\Longrightarrow \Theta(N^2)$$

If there is a cycle, the algorithm stops at the first repeated node after once round the cycle

$$\Longrightarrow \Theta((j+l)^2)$$

Less naïve circularity detection

Algorithm: at each step, check all previous steps for repeat

Hash-table memory

Require: sequence :: list
  table ← new Hashtable
  node ← sequence
  while node ≠ NIL **do**
    if node ∈ table **then**
      return true
    end if
    table[node] ← true
  end while
  return false

Less naïve circularity detection

Algorithm: at each step, check all previous steps for repeat

Complexity analysis

Space

Hash table with N entries required

$$\implies \Theta(N) \text{ extra space}$$

Time

$$T_k = T_{k-1} + \Theta(1)$$

If there is no cycle, the algorithm traverses the entire list, doing a constant-time lookup each time

$$\implies \Theta(N)$$

If there is a cycle, the algorithm stops at the first repeated node

$$\implies \Theta(j + l)$$

(assumes hash-table lookup is $\Theta(1)$)

Hare and tortoise

Also known as Floyd's cycle-finding algorithm

Key insight

for circularity of length l beginning at position j

$$L[k + nl] = L[k]$$

for all $k > j, n \geq 0$.

... or in words

If two nodes at different positions in the list are identical, the difference in positions is an integer multiple of the circularity length.

Converse

If there is a circularity and two iterators are each within it, incrementing the *difference* between two list iterators by 1 will always lead to the two iterators arriving at the same list node.

Hare and tortoise

Also known as Floyd's cycle-finding algorithm

Algorithm

Require: sequence :: list
  tortoise ← REST(sequence)
  hare ← REST(tortoise)
  while hare ≠ NIL **do**
    if hare = tortoise **then**
      return true
    end if
    tortoise ← REST(tortoise)
    hare ← REST(REST(hare))
  end while
  return false

Hare and tortoise

Also known as Floyd's cycle-finding algorithm

Complexity analysis

Space

No extra space needed

$$\Longrightarrow \Theta(1)$$

Time

If there is no cycle, the hare traverses the list; in that time, the tortoise traverses half the list:

$$\Longrightarrow \Theta(N)$$

If there *is* a cycle, the hare and tortoise meet no more than l steps after the tortoise is in the cycle

$$\Longrightarrow \Theta(j + l)$$

Additional information from algorithm

Position of first repeat

1. reset tortoise to the head of the list
2. move hare and tortoise one step at a time (same speed)
3. count steps until hare and tortoise are equal

Length of circularity

1. hold tortoise still
2. move hare one step at a time
3. count steps until hare and tortoise are equal again