Program Termination Analysis and Cycle Detection

Lauren Leung

I. Introduction

Iterative algorithms have many practical applications in mathematics. Root-finding algorithms like the bisection method home in on the vicinity of a function's sign change with increased precision, and methods like Gauss-Seidel iteratively apply matrix operations to solve a linear system of equations. Although iterative algorithms are widely used in all areas of programming, their behavior is not always transparent.

As the amount of data modified within each iteration increases, it is not always clear when an iterative algorithm will end, or whether it will ever end. Consider the following example in Python posed by mathematician Byron Cook et. al in their article "Proving Program Termination,"[1] where each call to `input()` indicates the program waits for the user to enter a number of their choosing:

```
x = input()
y = input()
while (x > 0 and y > 0):
  if input() is 1:
    x = x - 1
    y = y + 1
  else:
    y = y - 1
```

Because multiple variables in the while loop are manipulated by the program in contradictory ways, it is not immediately clear whether this program will terminate or continue indefinitely. Although defining a maximum number of iterations is appropriate for programs where an *approximation* is desired over a *full computation*, there are cases when a programmer will only want to run a program if they know it will run in full. A universal method for determining program termination is therefore a desirable software design tool, and its tenability is reflected by historic discoveries in the field of computability theory.

II. The Halting Problem

Computability theory is the field of determining what a computer is capable of accomplishing when given finite time and infinite memory. Although real computers have a finite amount of memory, computability theory provides a useful upper bound on what is and is not possible for a

[1] Cook, Byron, Andreas Podelski, and Andrey Rybalchenko. "Proving Program Termination." *Communications of the ACM* 54, no. 5 (2011): 88. doi:10.1145/1941487.1941509.

physical machine. **The halting problem** is a famous problem in computability theory that invites observers to propose a universal algorithm that matches the following behavior:

> Given a program P and an input string w, return “true” if P(w) terminates, else if P(w) loops infinitely return “false”.

It is easy to demonstrate the existence of a partial solution that returns “true” if P(w) terminates. Take the following algorithm:

1. Run P(w)
2. **Stop and return “true.”**

Because each step occurs one-after-the-other rather than simultaneously, step 2 is only reached after step 1 has completed, and the algorithm will always return “true” when P(w) terminates. However, this algorithm does not return anything when P(w) loops infinitely, which makes it a partial solution only.

However, a *complete and universal* solution to the halting problem is unattainable, as can be shown through **proof by contradiction**. A complete and universal solution is first assumed to exist, and it is shown that this would lead to a contradiction. Proof by contradiction is applied to the Halting Problem in a proof by Professor Kirk Pruh of the University of Singapore:[2]

1. Assume an algorithm HALT(P, w) exists, and that it always returns either “true” or “false.”
2. Take the following method Z(w), that takes in an input w:
    a. if HALT(Z, w) is true, loop infinitely
    b. else if HALT(Z, w) is false, return immediately
3. If Z is executed with itself as input, the following cases are possible:
    a. **HALT(Z, Z) is true**: Z(Z) will halts and returns immediately. This is a contradiction.
    b. **HALT(Z, Z) is false**: Z(Z) will loops infinitely. This is also a contradiction.

Although a universal solution to the halting problem is provably impossible, many methods can prove non-termination for *certain circumstances*. One method discussed by Cook et. al emphasizes the importance of **cycle detection,** a technique for analyzing data structures called **graphs** to determine the suitability of an algorithm for the graph provided.[3] To further discuss applications for cycle analysis in program termination, additional technical terminology is needed.

---

[2] Pruhs, Kirk. "Halting Problem." National University of Singapore Computing. Accessed December 15, 2016. http://www.comp.nus.edu.sg/~cs5234/FAQ/halt.html.

[3] Cook, Byron, Andreas Podelski, and Andrey Rybalchenko.

III. Definitions

A **node** is a data structure that contains a value and a list of edges. The value contained in a node can be simple, like an integer or string, or complex, like an array containing a list of information.

An **edge** is a relation between two adjacent nodes. This relationship can represent anything, but must be consistent for all the edges in the graph.

A **graph** is a set of nodes. The set does not need to be ordered, and can be thought of as a single "bucket."

An **undirected graph** is a graph in which for every edge from $v_1 \rightarrow v_2$, there is a corresponding edge from vertex $v_2 \rightarrow v_1$. Facebook friends could be represented by an undirected graph, since once a friend request is accepted, friendship on Facebook is mutually shared between individuals.

Consider a simple graph that represents an arbitrary rectangle ABCD, in which each value represents a vertex in the rectangle and each edge represents geometric adjacency. This graph is undirected, because geometric adjacency is a symmetric relation. In Python, such a graph might look like:

```
rectangle = {'A': ['B', 'D'],
             'B': ['A', 'C'],
             'C': ['B', 'D'],
             'D': ['A', 'C']}
```

A **directed graph** is a graph in which the existence of edge $v_1 \rightarrow v_2$ does *not* imply the existence of edge $v_2 \rightarrow v_1$. Twitter followers could be represented by a directed graph, since following is a one-way relation and following someone on Twitter does not guarantee they also follow you.

Consider a simple directed graph that represents a set of integers, in which each value represents an integer and each edge represents the "greater than" relation. The following is an example of such a graph:

```
some_ints = {1: [],
             2: [1],
             4: [1, 2, 3],
             3: [1, 2]}
```

A **connected graph** is a graph in which there exists a path between any two points in the graph. Both the `rectangle` and `some_ints` are connected graphs.

The following is an example of a graph that is not connected, because there exists no path between nodes A and C:

```
not_connected = {'A': ['B'],
                 'B': [],
                 'C': ['B']}
```

A **path** is an ordered sequence of nodes $n_1, n_2, ..., n_k$ such that $2 < k$ and for every pair of nodes n_i, n_{i+1} where $1 \leq i < k$, there exists an edge from n_i to n_{i+1}.

`not_connected` contains the two paths (`A`, `B`) and (`C`, `B`).

A **directed acyclic graph** is a directed graph in which for all nodes n in the graph, there is no path from n to n.

A **linked list** is a connected directed acyclic graph in which every node contains one edge to the "next" node in the list, except for the last node in the list, which contains no edges.

The following is an example of a linked list of length four:

```
list = {'item1': ['item2'],
        'item2': ['item3'],
        'item3': ['item4'],
        'item4': []}
```

A **recursive** method is a method that calls itself. Recursive methods are useful for iteratively manipulating data when the total number of iterations needed is undefined.

The following is an example of a recursive Python program that counts the number of items in a linked list `list`:

```
list = {'item1': ['item2'],
        'item2': ['item3'],
        'item3': ['item4'],
        'item4': []}

def count(node, current_count):
  # Check if there is a 'next' node
  if (len(list[node]) == 0):
    return current_count
  else:
    # Increment current_count and call count() on 'next' node
    return count(list[node][0], current_count + 1)
```

```
print count('item1', 1)
```

Graphs in an Object Oriented Paradigm

All the preceding examples have represented nodes using simple mappings from values to lists. Such a simplification is not always sufficient, since nodes sometimes contain values that are more complex than strings or integers. In order to allow sufficient flexibility while maintaining legible code, subsequent sections will use an implementation of graphs and nodes in **Java**. This implementation will be *generic*, meaning each node can contain any type of value. The *type* of a Node's value is left undefined until the programmer specifies one, which is denoted by the letter `T`.

The following node methods are defined:

1. `public Node(T value, HashSet<Node<T>> edges)` – Given a value and a set of edges, creates and returns a new Node.
2. `public Node(T value)` – Given a value, creates and returns a new Node with an empty set of edges.
3. `public T getValue()` – For a given Node, returns the value of the Node
4. `public List<Node<T>> getEdges()` - For a given Node, returns all the edges in a list form.
5. `public void addEdge(Node<T> node)` - For a given Node, adds the argument `node` to its list of edges. This method does not have a return value.

Because a graph is defined as an unordered set of nodes, graphs are implemented using the Java data structure `HashSet<Node<T>>`.

Where appropriate, a graph represented by `HashSet<Node<T>>` may be converted to another data type, `ArrayList <Node<T>>`. This conversion places the nodes in an ordered list such that they can be iterated over easily.

Although taking these methods as given will suffice for subsequent discussion of cycle detection, implementations for these methods are included after the final section.

IV. Cycle Detection

Motivating Example

Consider the following example, where a method `countNodesInList` returns the number of nodes in a Linked List graph:

```
/*
 * Given a start node, return the number of nodes in the
```

```
 * linked list
 */
public static int countNodesInList(Node<T> start) {

  int count = 0;
  List<Node<T>> next = start.getEdges();

  do {
    /* get node pointed to by current node */
    Node<T> current_node = next.get(0);
    next = current_node.getEdges();
    count = count + 1;
  } while (next.size() > 0);

  return count;
}
```

This program will terminate for all cases where `start` belongs to a valid linked list, since a linked list is defined as a finite, connected, and acyclic graph. Because there is only one path between any two nodes in an acyclic graph, each node can only be reached once, and the while loop will iterate a finite number of times. However, consider what will happen when `countNodesInList` is run on the following input:

```
Node<Integer> first = new Node<Integer>(1);
Node<Integer> second = new Node<Integer>(2);
first.addEdge(second);
Node<Integer> third = new Node<Integer>(3);
second.addEdge(third);
third.addEdge(first);
countNodesInList(first);
```

When the while loop inside `countNodesInList` reaches the node `third`, the value of `next.get(0)` will point back to the node `first,` and the loop will repeat indefinitely. It is clear from this example that assuming a well-formed and acyclical input is insufficient.

Similar issues arise from algorithms that take any directed graph as input. Consider the following recursive algorithm `generateNetwork`, which creates a graph of `user`'s network based off of everyone who `user` has followed, and everyone who has been followed by people `user` has followed, and so forth. For each outbound follow request belonging to `user`, the algorithm first adds `user` to their own network, then uses recursive calls to `generateNetwork` to retrieve the networks of everyone `user` follows:

```
/*
 * Given a user, generate a graph of that user's
 * network, based on outbound follow requests
 */
```

```
public static int generateNetwork(Node<User> user, HashSet<Node<User>> network)
{
  network.add(user); /* add the current user to their own network */
  /* get outbound follow requests for current user */
  List<Node<User>> outbound_follows = user.getEdges();

  for (Node<User> acquaintance : outbound_follows) {
    HashSet<Node<User>> acquaintance_network = generateNetwork(acquaintance,
network);
    network.addAll(acquaintance_network);
  }
  return network;
}
```

While the program `countNodesInList` terminated whenever `start` belonged to a valid linked list, there are many cases where `user` belongs to a valid directed graph and `generateNetwork` does not terminate. Consider the following example:

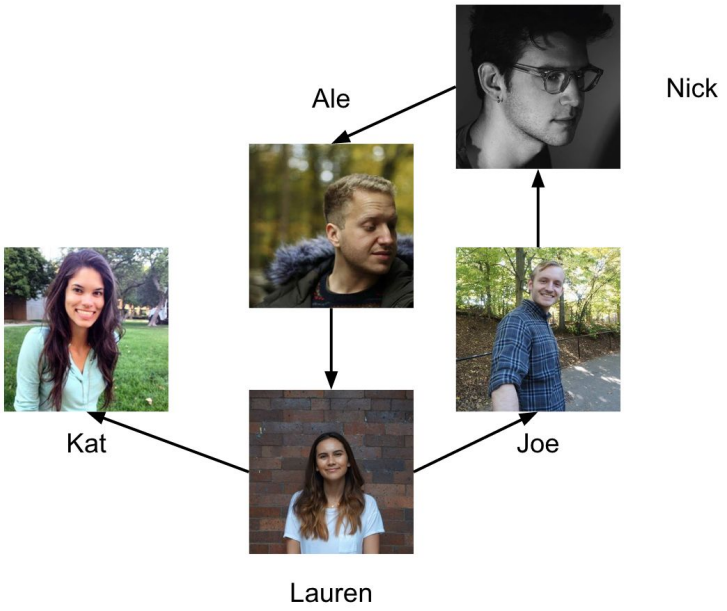
```
HashSet<Node<User>> graph = new HashSet<Node<User>>();
Node<User> lauren = new Node<User>("Lauren");
Node<User> kat = new Node<User>("Katrina");
Node<User> ale = new Node<User>("Alejandro");
Node<User> joe = new Node<User>("Joe");
Node<User> nick = new Node<User>("Nick");

lauren.addEdge(joe);
lauren.addEdge(kat);
ale.addEdge(lauren);
joe.addEdge(nick);
nick.addEdge(ale);

graph.add(lauren);
graph.add(lauren);
graph.add(joe);
graph.add(nick);
graph.add(ale);
graph.add(kat);

HashSet<Node<User>> network = new HashSet<Node<User>>();
generateNetwork(graph, network);
```

The nodes in this example correspond to the following chart:



Because there exists a cycle from `lauren→joe→nick→ale→lauren` , the algorithm will recursively call `generateNetwork` on these nodes infinitely.

`generateNetwork` will terminate for *some* directed graphs, so long as there is no cycle in the graph. When this is the case, every path in the graph contains each node at most once. Because there is a finite number of unique nodes in the graph, every path in the graph that corresponds to a recursive call will terminate and return eventually.

In the first example, the programmer may deem it reasonable for `countNodesInList` to stop and throw an error when a cyclical linked list is provided. In the second example, it may be more likely that the input node is part of a cyclical graph. When this is the case, rather than stop and throw an error, the programmer may wish to call a different algorithm that is equipped to handle cycles. This may be especially desirable in the case that handling cyclical input is slower, less accurate, or consumes more memory.

Floyd's Algorithm for Cycle Detection

Floyd's Algorithm, also known as the **Tortoise and the Hare Algorithm**, is a cycle-finding algorithm first outlined by computer scientist Donald E. Knuth in 1967.[4] The algorithm is based on a simple observation: if two objects move with constant velocity v_1, v_2 where $v_1 > v_2$ and are traveling down a straight path, the faster object will pass the slower object exactly once before reaching the end of the path. However, if the two objects are traveling down a cyclical path, the

[4] Knuth, Donald E. *The Art of Computer Programming, Vol. II: Seminumerical Algorithms*. Addison-Wesley, 1967.

faster object will pass the slower object many times. The algorithm's second name comes from the imagining of this faster object as a "hare," and the slower object as a "tortoise."

Although a number of cycle detection algorithms exist, this paper focuses on Floyd's Algorithm. Later sections will discuss the time and space tradeoffs of Floyd's algorithm, which will reveal it to be an efficient choice for linked list analysis.

Floyd's Algorithm Implementation

Consider Floyd's algorithm on a linked list, where the algorithm must determine if the linked list is well-formed and acyclical or if the linked list contains a cycle. Recall that in a linked list, it is assumed that each node has only one edge that points to the next item in the list. For this example, the algorithm proceeds as follows:

1. Given a list of Nodes, make the "turtle" the first Node in the list, and make the "hare" the next node after "turtle."
2. If either the turtle cannot move one node forward or the hare cannot move two nodes forward, **stop and return "false."**
3. If the turtle and the hare are set to the same node, a cycle has been found. **Stop and return "true"**
4. Set "turtle" to the next node after "turtle," and set "hare" to the next node after the next node after "hare."
5. Repeat step 2 with the new values of "turtle" and "hare."

The following `containsCycle` method is specific to linked lists, and makes use of Floyd's algorithm. `containsCycle` performs step 1, then calls the iterative method `turtleAndHareList` to perform steps 2-4.

```
/*
 * Helper method that implements Floyd's algorithm, assuming a linked list
 */
private static <T> boolean turtleAndHareList(Node<T> turtle, Node<T> hare) {
    List<Node<T>> turtle_edges = turtle.getEdges();
    List<Node<T>> hare_edges = hare.getEdges();
    List<Node<T>> hare_next_edges = hare_edges;

    while (turtle_edges.size() != 0 && hare_next_edges.size() != 0) {
        /* If the current node has no edges, we have reached the end */
        if (turtle == hare) {
            return true;
        }
        turtle = turtle_edges.get(0);
        hare = hare_next_edges.get(0);

        turtle_edges = turtle.getEdges();
```

```
            hare_edges = hare.getEdges();
            if (hare_edges.size() == 0) {
                return false;
            }
            else {
                hare_next_edges = hare_edges.get(0).getEdges();
            }
        }
        return false;
    }

    /*
     * Iterate over all the Nodes in a graph, and run turtleAndHare on them.
     */
    public static <T> boolean containsCycle(HashSet<Node<T>> graph) {
        ArrayList <Node<T>> nodes = new ArrayList<Node<T>>(graph);

        /* Call turtleAndHare for each Node in the graph */
        for (Node<T> node : nodes) {
            /* set turtle to the current node */
            List<Node<T>> edges = node.getEdges();

            if (edges.size() == 0) {
                /* If the current node has no edges, Floyd's cannot be run. skip it
    */
                continue;
            }
            else {
                Node<T> next_node = edges.get(0);
                if (turtleAndHareList(node, next_node)) {
                        return true;
            }
        }
        /* turtleAndHare has returned false for all nodes in the graph */
        return false;
    }
```

Using Floyd's Algorithm to detect cycles in graphs where the max number of edges for any node $E_{max} > 1$ requires additional recursive calls. Each time the current turtle or hare node has multiple edges, the algorithm must test every possible path for a cycle. This is done in a triple-nested for-loop, which generates every possible value for the hare (there are two loops to reflect the fact that the hare moves two spaces forward) and every possible value for the turtle, and recursively calls `turtleAndHare` on each combination. The method `containsCycle` is also redefined to accept a parameter `is_list`, which can be set to either "true" or "false" - if `is_list` is set to "true," `containsCycle` will call `turtleAndHareList`, and if `is_list` is "false," `containsCycle` will call `turtleAndHare`.

```
    /*
```

```
 * Recursive helper method that implements Floyd's algorithm
 */
private static <T> boolean turtleAndHare(Node<T> turtle, Node<T> hare) {
    if (turtle == hare) {
        /* Turtle has caught up to Hare. A cycle was found */
        return true;
    }

    List<Node<T>> turtle_edges = turtle.getEdges();
    List<Node<T>> hare_edges = hare.getEdges();

    /* Triple-nested for-loop (ouch) */
    for (Node<T> hare_neighbor : hare_edges) {
        List<Node<T>> hare_neighbor_edges = hare_neighbor.getEdges();
        for (Node<T> turtle_next : turtle_edges) {
            for (Node<T> hare_next : hare_neighbor_edges) {
                if (turtleAndHare(turtle_next, hare_next)) {
                    return true;
                }
            }
        }
    }
    return false;
}

/*
 * Iterate over all the Nodes in a graph, and run turtleAndHare on them.
 * is_list = true when the input Graph is to be used as a linked list
 */
public static <T> boolean containsCycle(Graph<T> g, boolean is_list) {
    List <Node<T>> nodes = g.getNodes();

    /* Call turtleAndHare for each Node in the graph */
    for (Node<T> node : nodes) {
        /* set turtle to the current node */
        List<Node<T>> edges = node.getEdges();

        if (edges.size() == 0) {
            /* If the current node has no edges, Floyd's cannot be run. skip it
*/
            continue;
        }

        else {
            /* Check each edge in the node */
            for (Node<T> edge : edges) {
                Node<T> next_node = edge;
                if (is_list && turtleAndHareList(node, next_node)) {
                    return true;
                }
```

```
                else if (turtleAndHare(node, next_node)) {
                    return true;
                }
            }
        }
    }
    /* turtleAndHare has returned false for all nodes in the graph */
    return false;
}
```

Time complexity of Floyd's Algorithm

First, the runtime of Floyd's Algorithm for a linked list is analyzed:

- Take λ to be the length of the cycle.
- Take μ to be the number of nodes between the initial node n_0 and the first node in the cycle, such that $\mu = n - \lambda$. The amount of time the turtle takes to arrive at the first node in the cycle is $O(\mu)$.
- Take i to be the number of iterations elapsed in the algorithm. At each iteration, the turtle has proceeded i nodes forward and the hare has proceeded $2i$ nodes forward.
- When the hare has caught up with the turtle, both the turtle and hare are inside the cycle. Define the turtle's position as $i = \mu + a\lambda$ and the hare's position as $2i = \mu + b\lambda$, where a and b represent the number of times the turtle and hare went around the cycle.
- Subtracting the two equations yields $i = (b - a)\lambda$. Since i, λ must both be integers, $b - a$ must also be an integer.
- The amount of time the hare takes to catch up to the turtle once both are within the cycle is $O((b - a)\lambda) = O(\lambda)$.

The overall time complexity of Floyd's Algorithm for a linked list is therefore $O(\lambda + \mu) = O(n)$.

Recall that Floyd's Algorithm for a graph attempts Floyd's Algorithm on every possible path until there are no possible paths left, or a cycle is found. An upper bound on Floyd's Algorithm for general directed graphs can be obtained by maximizing the following two quantities:

1. The length of the cycle λ
2. The distance of the path between the starting node n_0 and any node inside the cycle, μ
3. The number of non-cyclical paths the algorithm checks at each iteration, before continuing towards the cycle

Assume the input graph contains n nodes. Take λ as a constant representing the maximal-length cycle in the graph, where $\lambda \leq n$. The maximal value for μ is therefore $n - \lambda$, since this initial path cannot traverse any node in the cycle.

The maximum number of paths in the graph that do not contain a cycle is calculated as follows:

- Take the μ nodes in the graph that are outside the cycle.
- Assume that any subset of these nodes represents a valid path. There are 2^{μ} or $2^{n-\lambda}$ such subsets.

Assume that for every iteration of the algorithm, each of these non-cyclical paths is traversed before the turtle and hare proceed towards the path containing a cycle. An upper bound on Floyd's Algorithm for a general directed graph is therefore $O((2^{n-\lambda}) \cdot [(n - \lambda) + (\lambda)])$, or $O(2^{n-\lambda} \cdot n)$.

Space complexity of Floyd's Algorithm

Floyd's Algorithm for linked lists does not store any data besides the turtle node, the hare node, and both nodes' edges. This requires a constant amount of memory, making the space complexity for linked lists $O(1)$.

Floyd's Algorithm for general directed graphs requires a constant amount of memory for each recursive call that has not yet returned. The maximum number of recursive calls that have not returned occurs when the turtle and hare have made it through the cycle some number of times, and may be in the process of exploring a path outside the cycle. This is equal to a space complexity of $O(2\mu + k\lambda)$ for some constant k, or $O(\mu + \lambda) = O(n)$.

V. Conclusion

Drawing upon the partial solution to the Halting Problem introduced in Section II, cycle detection can be used to provide an additional layer of analysis for algorithms that manipulate directed acyclical graphs:

1. **If containsCycle(G), stop and return "false"**
2. Run P(w)
3. **Stop and return "true."**

While this is one small step towards a tool that forecasts program termination, termination analysis tools like AProVE have received accolades for using a wide range techniques to analyze program termination, which include cycle detection.[5] Although devising one-off solutions for every type of infinite loop may be implausible, the efficacy of these tools suggests that programmers may soon be able to rely on software that is a close second.

---

[5] Giesl, Jurgen, Rene Thiemann, Peter Schneider-Kamp, and Stephan Falke. "Automated Termination Proofs with AProVE." *J Autom Reasoning*, October 5, 2016, 210-20. Accessed December 15, 2016. doi:10.1007/978-3-540-25979-4_15.

```
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;

public abstract class AbstractNode<T, K> {

    /*
     * These are set to private, since the value and edges are
     * only accessed and modified through helper methods
     */
    protected T value;
    protected HashSet<K> edges;

    /*
     * Initializes a Node, given a generically typed value
     * and a set of edges
     */
    public AbstractNode(T value, HashSet<K> edges) {
        this.value = value;
        this.edges = edges;
    }

    /*
     * Initializes a Node, given a generically typed value only
     */
    public AbstractNode(T value) {
        this.value = value;
        HashSet<K> empty_edges = new HashSet<K>();
        this.edges = empty_edges;
    }

    /*
     * Return the value of a node.
     */
    public T getValue() {
        return value;
    }

    /*
     * Return a list containing all the node's edges.
     */
    public List<K> getEdges() {
        List<K> edges_list = new ArrayList<K>(edges);
        return edges_list;
    }

    /*
     * Add a new adjacency to the node
     */
    public void addEdge(K node) {
```

```
            edges.add(node);
        }
    }
```