Preliminary assignment - Paths in a Graph

Author: Anders Nylund

February 19, 2017

Application number: UAF1703615

Abstract

This document is the answer of applicant Anders Nylund to the Preliminary assignment "Paths in a Graph". The assignment is part of the admission process to the program Computer Science at University of Helsinki.

Task 1 Solution

The method InfPath takes a graph and an integer as argument and checks if an infinite path can be found, starting from the node with the index of the passed integer. InfPath returns the value of the method FindNodesRecursive. As InfPath takes only a graph and an integer as argument, FindNodesRecursive-method needed to be created as additional arguments were necessary to make the algorithm as fast and effective as possible.

The arguments in the methods are passed by reference. Every node has an iterable list of arcs i.e nodes that it is connected to.

```
function InfPath(G, v)
  return FindNodesRecursive(G, v, [], [])
endfunction

function FindNodesRecursive(G, v, checkedNodes, deadEnds)
  if checkedNodes contains Node v
    return true
  else
    clone checkedNodes to clonedNodes
    checkedNodes.add(Node v)
    for endNode in endNodes
      if deadEnds contains endNode
        return false
      endif
      if FindNodesRecursive(G, endNode, clonedNodes, deadEnds)
        return true
      else
        deadEnds.add(endNode)
      endif
    endfor
    return false
  endif
endfunction
```

The algorithm creates a 'tree' of recursive calls. For every end node in the current node in iteration, a new call to find the end nodes of the current node in the iteration is made. This recursive calling is made as long as either an infinite path is found, or when a node that has no arcs starting from it, is found. I.e. this algorithm is an adaption of **Depth-first Search (DFS)**.

The algorithm start by calling the recursive method FindEndNodesRecursive (FENR) that takes the graph, a starting node and two empty lists as argument. These two empty lists will be used for determination if an infinite path was found and for optimizing the search time. The first call to FENR makes the 'trees' first branch. From this branch FENR will be called as many times as the current nodes has starting arcs.

First thing FENR does is return true if the 3:rd argument, that is a list of previously checked nodes, contains the current node. If not it copies the previously checked nodes, which will later be passed to all the sub-branches of the current branch. If this was not done, every branch would receive the same list of checked nodes, and the algorithm would fail.

Then FENR iterates all arcs of the current node. For each arc's end node it checks if it is a member of the dead ends list. This is a list that allows the algorithm to remember all the nodes that lead to dead ends. If the current node is in the list of dead ends the algorithm can immediately return false. If the node wasn't a dead end FENR is called recursively with the current node as the start of the new branch of the 'tree'.

The return value of FENR is evaluated, true is returned to the caller of the method. If the return value of FENR is false, the current node is added to the list of dead ends. FENR ends with returning the value false. Because if the method reached this part every recursive call has returned false, and the algorithm can be sure that no infinite path was found from this branch of the 'tree'.

The checking of dead ends significantly improves the search time of the algorithm. Without this feature the algorithm would make recursive calls on nodes, that has already been proved to not contain infinite paths. Of course in the beginning every path or branch of nodes has to be checked, but very fast the list of dead ends grows and the algorithm does not need to check the same branch more than once.

By drawing a tree of the recursive calls of the algorithm, the time complexity can be determined. Each step, by traversing the created tree pre-order, is one call to the function FENR. But by checking the dead ends of the tree this can be lowered significantly.

Task 2

The algorithm is programmed in Java. For simplicity, some of the utilities in the package **Java.util** is used. The source code and the rest of the project can be found from the following link **https://github.com/andersnylund/assignment**

For testing the algorithm a method called timeExecution() was created. This method takes the graph, a start node and the amount of test runs as argument. This way the algorithm can be timed on the same graph many times, and an average can be created from each of the searches of the test. The execution time is measured in nanoseconds.

1:st test

The first test is the **a)** graph of the example in the assignment.

```java
public static void main(String[] args) {

   Graph graphA = new Graph();
   graphA.addArc(1, 2);
   graphA.addArc(1, 4);
   graphA.addArc(2, 3);
   graphA.addArc(2, 5);
   graphA.addArc(3, 6);
   graphA.addArc(4, 5);
   graphA.addArc(4, 7);
   graphA.addArc(5, 1);
   graphA.addArc(5, 3);
   graphA.addArc(5, 6);
   graphA.addArc(5, 8);
   graphA.addArc(5, 9);
   graphA.addArc(6, 9);
   graphA.addArc(7, 5);
   graphA.addArc(7, 8);
   graphA.addArc(8, 9);

   timeExecution(graphA, 1, 10000000);
}
```

By modifying my programs main() method like above, a graph with the same nodes and arcs will be created as in the assignment. The method timeExecution() will then try to find an infinite path from **graph a)**, starting from node number 1, ten million times. For each time, the elapsed time will be measured. At the end the program prints out the average time for calling the function infPath() once.

Here is the results of testing the graph 3 times with different amount of runs in each test.

```
timeExecution(graphA, 1, 10000000);

* Output *
Started timing of execution...
FOUND! There was an infinite path starting from node 1
Average search time: 914 nanoseconds
```

```
timeExecution(graphA, 1, 10000);

* Output *
Started timing of execution...
FOUND! There was an infinite path starting from node 1
Average search time: 6620 nanoseconds
```

```
timeExecution(graphA, 1, 10);

* Output *
Started timing of execution...
FOUND! There was an infinite path starting from node 1
Average search time: 211956 nanoseconds
```

From the results we can see that the time of each run increases, as we lower the amount of runs. This is probably caused by the machine that runs the program, and it's optimization of the execution.

2:nd test

The following tests are done on random graphs. The program can create random graphs with a certain amount of nodes. This allows larger and more complex graphs to truly test the algorithm. The code below demonstrates how the creation of random graphs can be used.

```
public static void main(String[] args) {
Graph graph = new Graph(10);
graph.print();
}

* Output *
1 -> 4
2 -> 4 5 8
3 -> 2 6 10
4 -> 5 6 9
5 -> 3 6
6 -> 2
```

```
7 -> 4
9 -> 6
10 -> 4 5 6
```

Here a random graph with 10 nodes is created by passing the integer 10 to the constructor of the class **Graph**. This creates 10 nodes, which then are connected randomly together. For each node, a random amount of arcs (between 0 and n, where n is the number of nodes in the graph) is drawn to random end nodes. Each time a new arc is added to the graph thorugh the method addArc(), it first checks if the arc violates the rules of the graph. Finally the graph is printed which shows every arc of the graph. The number before the arrow is the starting node and ever number after the arrow represents and end node in an arc.

A random graph with 10 000 nodes was created and the search algorithm was tested with 3 different lengths of tests.

```
public static void main(String[] args) {
Graph graph = new Graph(20000);
timeExecution(graph, 1, 10000000);
timeExecution(graph, 1, 10000);
timeExecution(graph, 1, 10);
}

* Output *
Started timing of execution...
FOUND! There was an infinite path starting from node 1
Average search time: 12172 nanoseconds
Started timing of execution...
FOUND! There was an infinite path starting from node 1
Average search time: 10841 nanoseconds
Started timing of execution...
FOUND! There was an infinite path starting from node 1
Average search time: 10146 nanoseconds
```

Time complexity and memory requirements

The time complexity and the memory requirement of the program is hard to estimate. The principle of the algorithm of the program is the same as presented in **Task 1**.

If the graph is presented as a tree, the worst-case search time of it is $O(M)$, where M is the count of nodes in the tree. This however is not the same as the count of nodes and arcs in the original graph. The checking of dead ends decreases the search time significantly too.

To represent the graph, it's nodes and arcs, the program uses Java's HashMap collection. This collection uses key-value pairs which enables easy iteration of nodes and arcs of the graph. A graph is made by adding an integer as an key to the HashMap. This represents a starting node. The value of the key is a list of integers. This is done with Java's utility ArrayList. This list consists integers that represent the end nodes that the node is connected to.

This means that the memory needed for creating a graph is directly correlated with the size of the graph. E.g for representing **graph a)** of this assignment, a HashMap with 9 key-value pairs is needed. Each key has an list which stores as many integers, as nodes it is connected too. So **graph a)** would need to store 24 integers, in addition of the overhead of the structures provided by Java.

The algorithm itself uses this representation of the graph to search for an infinite path. In addition it uses two ArrayList's of integers for storing the previously checked nodes and for the dead ends found during the search. These two ArrayList's however differ from each other. As presented in task 1 of this assignment, the algorithm creates a new branch on the 'tree' when searching for an infinite path. The ArrayList of dead ends is the same for each branch created. But the ArrayList of checked nodes is unique to every branch of the 'tree'. This means that for each level in the 'tree', the list of checked nodes grows with one.

Source code

Main.java

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

public class Main {

   public static void main(String[] args) {
      Graph graph = new Graph(10);
      timeExecution(graph, 1, 10000000);
   }

   private static void timeExecution(Graph graph, int startNode, int
       testRuns) {
      System.out.println("Started timing of execution...");

      List<Long> runTimes = new ArrayList<>();

      for(int i = 0; i < testRuns; i++) {
         long startTime = System.nanoTime();
         graph.infPath(startNode);
         long stopTime = System.nanoTime();
         runTimes.add(stopTime-startTime);
      }

      boolean found = graph.infPath(startNode);
      if(found) {
         System.out.println("FOUND! There was an infinite path starting
             from node " + startNode);
      } else {
         System.out.println("Not found. There was not an infinite path
             starting from node " + startNode);
      }

      long average = averageExecutionTime(runTimes);

      System.out.println("Average search time: " + average + "
           nanoseconds");
   }

   private static long averageExecutionTime(List<Long> runs) {
      long sum = 0;
      for(long value : runs) {
         sum = sum + value;
      }
      return sum/runs.size();
   }
}
```

Graph.java

```java
import java.util.ArrayList;
import java.util.List;
import java.util.HashMap;
import java.util.Map;
import java.util.Random;
import java.util.TreeSet;
import java.util.Set;

public class Graph {

   private Map<Integer, Set<Integer>> arcs = new HashMap<>();
   private Random rand = new Random();

   public Graph() {}

   public Graph(int n) {
      for(int i = 0; i < n; i++) {
         for(int j = 0; j < rand.nextInt(n); j++) {
            addArc(i+1, rand.nextInt(n)+1);
         }
      }
   }

   public void addArc(int startIndex, int endIndex) {

      // No arcs to itself
      if(startIndex == endIndex) {
         return;
      }

      // Must be a directed graph
      if(arcs.get(endIndex) != null) {
         if(arcs.get(endIndex).contains(startIndex)) {
            return;
         }
      }

      Set<Integer> endNodes = arcs.get(startIndex);
      if (endNodes == null) {
         endNodes = new TreeSet<>();
      }

      endNodes.add(endIndex);
      arcs.put(startIndex, endNodes);
   }
```

```
  public void print() {
    for(Map.Entry<Integer,Set<Integer>> entry : arcs.entrySet()) {
      System.out.print(entry.getKey() + " -> ");
      for(Integer value : entry.getValue()) {
        System.out.print(value + " ");
      }
      System.out.println();
    }
  }

  public boolean infPath(int startNode) {
  return findEndNodesRecursive(startNode, new ArrayList<>(), new
      ArrayList<>());
  }

  private boolean findEndNodesRecursive (int startNode, List<Integer>
      checkedNodes, List<Integer> deadEnds) {
    if (checkedNodes.contains(startNode)) {
      return true;
    }

    // Create a copy of the currently checked nodes
    // Java passes arguments by reference
    List<Integer> copiedNodes = new ArrayList<>();
    for(Integer node : checkedNodes) {
      copiedNodes.add(node);
    }

    copiedNodes.add(startNode);

    // TODO resolve if checking for deadEnds is efficient or not

    // If startNode has arcs
    if (arcs.containsKey(startNode)) {
      // Iterate all arcs
      for(Integer endNode : arcs.get(startNode)) {
        if(deadEnds.contains(endNode)) {
          return false;
        }
        if (findEndNodesRecursive(endNode, copiedNodes, deadEnds)) {
        return true;
        } else {
          deadEnds.add(endNode);
        }
      }
    }
    return false;
  }
}
```