Preliminary assignment - Paths in a Graph

Author: Anders Nylund

February 16, 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 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.

```
1 function InfPath(G, v)
2   return FindNodesRecursive(G, v, [], [])
3 endfunction
4
5 function FindNodesRecursive(G, v, checkedNodes, deadEnds)
6   if checkedNodes contains Node v
7     return true
8   else
9     clone checkedNodes to clonedNodes
10    checkedNodes.add(Node v)
11    for endNode in endNodes
12      if deadEnds contains endNode
13        return false
14      endif
15      if FindNodesRecursive(G, endNode, clonedNodes, deadEnds)
16        return true
17      else
18        deadEnds.add(endNode)
19      endif
20    endfor
21    return false
22  endif
23 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.

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.

$$\sum_{i=1}^{n} 2^{-}$$

i.e. $O(n^2)$

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.

```
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);
.
.
.
Started timing of execution...
FOUND! There was an infinite path
Average search time: 914 nanoseconds
```

```
timeExecution(graphA,1, 10000);
.
.
.
Started timing of execution...
FOUND! There was an infinite path
Average search time: 6620 nanoseconds
```

```
timeExecution(graphA, 1, 1);
.
.
.
Started timing of execution...
FOUND! There was an infinite path
Average search time: 535626 nanoseconds
```

2:nd test