Dalton Altstaetter

EE 360C

Dr. Pedro Santacruz

10/28/2014

Program 2 Pseudocode

Part 1 – Pseudocode:

How I plan to implement this program is to use adjacency lists to do a Depth First Search (DFS). To do so I will do the following:

---> Read the input and build an adjacently list. I will store these in a hashmap where the hashmap key is the base node & the list will answer which other nodes are ‘reachable’ from this base node. I will implement my list using a LinkedList representation.

---> After building my Adjacency List, I take my root node, where the message originates, and find the minimum communication time that lies within the time range. One of two things will happen:

1. It will not find a valid minimum time that lies within the time range and => the message cannot be received by the receiver node. At which time I simply output “0”.
2. It finds the valid minimum time to start at and its from here that I begin my DFS.

---> While (current time <= tf)

if (current time == tf)

-There is a path to end node. Print the number of traces & all traces from beginning node to end node to stdout and terminate program.

Else // keep searching adjacency list

If( currentTime > commTimeNextNode) // if I go backwards in time

-Remove the node from currentNodes’ adjacenct list

-Restore the outputTracesList to the last valid state

-Go back to top of Else block and keep searching adj. list

Endif

-From my current node go to the next node in my adjacency list (if it passes the above ‘ If ’ block)

-Update my currentTime and currentNode to my new values

If(nextNode == null) // there is no next node to go to

If(outputListSize == 0)

-There is no path from root node to end node. Output ‘0’

- Quit program

endif

-pop trace from outputList and set current root & time equal to the last trace added to output to restore to the last valid state

-Since I have already removed both traces from the adjacency list I won’t go down this incorrect path again & I go back to my previous node to look for any other paths to be taken from that node.

-Go back immediately to the ‘Else’ block and continue searching the adj. list from the last valid state.

Endif

// I go here only when the two ‘If ‘ blocks above evaluate to false

// Now I know the path still has traces left to check & I didn’t go

// backwards in time. This would be the normal execution path.

-Remove the paths from prevNode to currentNode in each ones’ respective adjacency list so as to ‘mark’ I’ve taken this path before

-add the trace to get to this new node to the the outputTracesList

endIf

endWhile

---> The program will now have either printed the traces or ‘0’ to indicate there was no path. What I basically did was go down each path from currNode to nextNode, remove the connection between currNode & nextNode in adj. list adding the traces to my outputList. When I run into a dead end I there is no longer a path from my previous valid state to currNode so this path will never be taken again. OutputList holds all the places I’ve gone so I restore currNode & currTime from the last trace added to outputList so that I only add back nodes that can still check other paths in the DFS. If there are two different partitions where my root node is in partition 1 and my finish node is in partition 2 then that’s where the

“If(outputListSize == 0)” block comes in and as I remove traces from my output If I find that my outputListSize == 0 then => I have gotten all the way back to my initial starting point and therefore the finish node is not reachable from my starting node.

**Runtime Complexity & Part 2**:

To Reduce implementation cost I only added nodes values to my adjacency list that correspond to the time in question and if there are nodes that communicate at time t > tf, where tf is my end communication time. I will denote edges that are within this time interval mf.

For nodes, n, and edges, m, the cost to build this adjacency list is O(n + 2mf) since I add the edges for each node ntimes and there are 2mf valid edges that I add since I double count edges since an adjacency list has the property ‘is reachable from’ and I have to do it for both nodes.

Now to go through the DFS using my adj. list is O(n+mf) as been proven in class and also can be seen by my algorithm since when I find a valid communication path I visit every edge that is valid, mf (independent of n), and I find my path on the final node. Now, you will actually get my worst runtime complexity when you have two partitions and every node except the one you’re trying to reach is in one partition and the node that you want to end at is all alone in the other i.e. where your end node is not reachable from any other node in the graph. In my implementation if this occurs I will have a runtime complexity of O(n-1+2mf) since I will reach n-1 nodes and will trace forward mf edges and then remove mf edges from my outputList until my outputListSize = 0. So my run time complexity is O(n-1+2mf) = O(n + m). Thus I have proved the correctness of my algorithm to be O(n+m).