Csci422 Project Outline

Andrew Gair

Program Description:

Read input graph from a file passed to the program through the command line. Program will run two algorithms, one for cycle detection and another for removing select vertices of degree 2. Modified graph will be output to a file and terminal window will display if a cycle exists.

Data format:

Data will be read into the program as adjacency lists through an input file (figure 'a' below). The input file will follow this format:

The first character will be either a 'D', or a 'U'. Representing the fact that the following data is either a directed graph or an undirected graph.

Following that, the file will contain the total number of vertices in the graph, this value will be known as 'n'. Similar to what was done for the Poset labs, this value will be used as the end of line character.

Each line after the two previous will represent a vertex in the graph, starting with vertex 0 and working up to vertex n-1. These lines will start with the vertex number they represent and follow with a space-separated list of all neighbours for the given vertex. This list is to be sorted in ascending order. To denote the end of the list the value 'n' is to be used, 'n' also denotes the end of the file when found as the only entry on the line.

Data Storage:

Data will be stored in an array of vectors. The array itself represents a given vertex (figure 'b'), and the vector it contains is the list of all adjacent vertices (figure 'c').

When the program starts it will open the file specified in the command line and read supplied data into the array of vectors.

Example input:

U
5
0 1 2 5
1 0 4 5
2 0 4 5
3 2 5
4 1 2 5
5

(a)

Bolded items stored into nodeList[0 through 4]

U
5
0 1 2 5
1 0 4 5
2 0 4 5
3 2 5
4 1 2 5
5

(b)

Bolded items stored into nodeList[0].nodeVector[]

U
5
0 **1 2 5**
1 0 4 5
2 0 4 5
3 2 5
4 1 2 5
5

(c)

Algorithms:

There are two algorithms that are to be implemented:

1) Determine if a graph contains a cycle of length at least 3. This should be completed in O(|v| * |e|) time.
2) Remove each vertex of degree 2 from a graph by replacing the edges (u, v) and (v, w) with the edge (u, w). Removing vertices may end up creating a new vertex of degree 2 which must then be removed. This should be completed in O(n) time.

These algorithms are preliminary as they have not yet been tested, they serve only as guides for what the final algorithm may look like. They focus only on simple undirected graphs for now.

Algorithm #1) Cycle detection

Assumption: Do not care where the cycle occurs, just that there is a cycle.

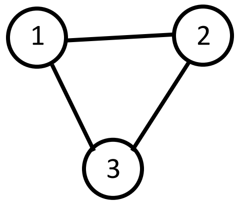
```
bool cycle(int previousNode, int currentNode, int targetNode, int indexer){
//--If algorithm ever finds a cycle with length at least 3
    if(nodeList[currentNode].nodeVector[indexer] != previousNode AND
    nodeList[currentNode].nodeVector[indexer] == targetNode){
        return true
    }
//--Increment through the neighbours of currentNode
//--Ignores previous node so to avoid traversing back to startNode right away
    else if(indexer != n AND nodeList[currentNode].nodeVector[indexer] !=
                                                previousNode){

        return cycle(previousNode, currentNode, targetNode, indexer++)
    }
//--Increment down the array list and reset indexer
    else if(currentNode != n){
        return cycle(currentNode, currentNode++, targetNode, 0)
    }
    else{
        return false
    }
}
```

Time complexity: O(|n| * |e|)
For each node, this function checks each possible edge.

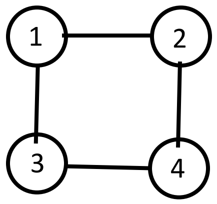
Algorithm #2) Removing vertices of degree 2 (where possible)

Assumption: If (u, w) path already exists, then vertex v will not be removed (fig 2.1).



No vertex meets requirement for removal and replacement as a path between all vertices already exists.

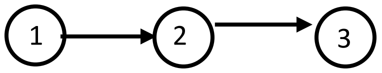
Fig 2.1



Any vertex qualifies for removal and replacement, but only one can be chosen before ending up in the situation shown in fig 2.1

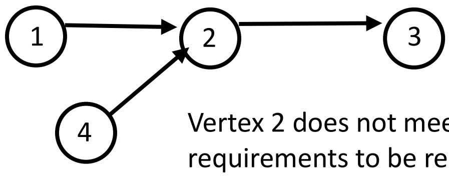
Fig 2.2

Note: for directed graphs this algorithm needs to check both the outgoing and incoming degree. Needs exactly 1 incoming edge and exactly 1 outgoing edge in order to meet requirements for removal and replacement (fig 2.3).



Vertex 2 meets the requirements to be removed and replaced in a directed graph.

Fig 2.3



Vertex 2 does not meets the requirements to be removed and replaced in a directed graph.

Fig 2.4

```
//--Run the algorithm
for(each node){
        removeReplace(node)
}

//--function explanations are provided on the next page
void removeReplace(currentNode){
        if(nodeList[currentNode].degree == 2){
                if(isPath(nodeList[currentNode].nodeVector[0],
                   nodeList[currentNode].nodeVector[1] == false){
                        createPath(nodeList[currentNode].nodeVector[0],
                                   nodeList[currentNode].nodeVector[1])
                        deleteNode(nodeList[currentNode])
                }
        }
}
```

Time complexity: not O(n) as was desired.

For each node, this function needs to compute degree (which could take a while depending on connectivity of the graph), but if degree is larger than 2 can stop checking immediately as the node fails to meet requirements to be removed. Thus this check can be run with only 2 comparisons.

isPath checks the first node's adjacency list for an instance of the second node. The algorithm returns true if a path is found, false otherwise. This could potentially take 'n-1' comparisons (potentially a node is connected to every other node). This is one reason linear run-time is not achieved.
For directed graphs the direction of edges will modify which of the two nodes should go in the first slot or second. For undirected graphs the choice is irrelevant as they are both needed to list each other as neighbours anyways.

createPath takes 2 nodes as input, it will modify the data structure to include a new edge. This will not take constant time as the data structure is in sorted order. Adding a new edge could take 'n-1' time to find insertion position (linear search handled by "find" function from vector STL). This is one reason linear run-time is not achieved.

deleteNode takes a node as input, it will remove this node from the data structure. nodeList[node] will be set to -1 to signify the node was removed. This will take constant time, the corresponding nodeVector will be removed at this point also, freeing up memory.

Concerns and Comments:

- Running time for createPath seems like a bottleneck for the algorithm. I'm wondering if sorting nodeVector is worth the time, given that it's not relevant for algorithm #2 to have a sorted list of adjacent nodes. I'd be curious to see how sorted or unsorted lists influence the performance of both algorithms. I assume it would have no effect.

- As far as reading data from the file goes, I'm thinking nodeVector should not be reading 'n' into its adjacency list as I demonstrated in figure 'c' on page 1. I can instead test if a vector is at the end of its list by using the .end() function from C++'s vector STL.

- Removing a node by replacing its entry in nodeList with a -1 seems lazy, but if I were to actually remove the item I would be required to re-sort nodeList as the elements within nodeList get their 'names' based on their positions. So the element in nodeList[3] is representing the node 'named' 3. This is relevant to any nodes that are adjacent to 3 as they would place the number '3' in their adjacency list.

  This matters quite a bit if the algorithm were to completely remove an element from nodeList and then "condense" nodeArray. Say if node 1 were removed, then node 3 would be shifted into the number 2 slot, therefore causing all sorts of trouble with its neighbours as they would still list '3' as a neighbour, but nodeList[3] represents the 4th node at this point, not the 3rd.

- I'm expecting directed graphs to be quite a challenge for algorithm #2, but I don't foresee algorithm #1 changing all that much to incorporate directed graphs if it already works for undirected graphs.