CSCI 422 Project Report

Andrew Gair

Objective:

The objective of this project was to implement two algorithms of my own design that solved the problems specified in "The Algorithm Design Manual" by Steven Skiena.

These algorithms were intended to work on simple undirected graphs, but my project aimed to expand the algorithms to work on simple directed graphs as well. The algorithm should be as similar as possible to its undirected counterpart, with minor variations permitted to allow correct functionality.

Algorithm Intro:

The two algorithms chosen were:

1) Detect if there is a cycle of length at least 3 in the graph.
2) Remove all nodes 'N' of degree 2 and replace them with the appropriate edges where the neighbours of N are not adjacent to each other
   For example: the node 'V' can be removed if it follows this format: (u, v) and (v, w). The new edge would then be (u, w). Supposing that 'U' and 'W' were not already adjacent before the removal.

Algorithm 1 is a modification of exercise 5-17 from the text.

Algorithm 2 is a modification of exercise 5-22 from the text.

Implementation:

In general: input format

> Each node keeps track of its neighbours through storing them into a stl::vector. These neighbours are stored in ascending order (although this doesn't end up changing anything), order is maintained when new edges are added.

Cycle Detection for undirected graphs: "Painting method"

> My implementation for cycle detection used a recursive Depth First Search and paints nodes as they are encountered. If a neighbour was already painted when a node checks it, then that means the algorithm has already been to that neighbour. If that's the case, then there is a cycle and the algorithm can return true.
> If no neighbour is painted, and this node has already checked all its neighbours, then end recursion and algorithm will continue with the node that pointed to this node.
> This means that if there is no cycle, then all nodes and all their neighbours must be checked.

Cycle Detection for directed graphs: “Old Unpainting” method

This algorithm remains the same as the “Painting method”, with a rather large addition of another neighbour vector called “pointsToMe”. This vector tracks all nodes that point to a given node. This was necessary for directed graphs as just because a node points to another, doesn’t mean that node points back.
This additional vector was used to track if the given node had a path to each node that pointed to it. This was used to make the determination of whether or not the node should keep its paint after it had checked all its neighbours. A node could only keep its paint if it had actually had such paths. This particular check was later found to be redundant and unnecessary.
Note: this method does not use the “paint-chip” as the “New Unpainting” method does. So runtime is increased by quite a lot as many nodes are checked and rechecked.

isPath():

This function took two important parameters (among others), the source and target nodes. It would check for a path from the source, to the target. It conducted a Depth First Search to achieve this result, it checked if each neighbour was in fact the target node. After checking a node, it would leave a mark, this mark signified that the node had already been checked, and no path was found. If later on during the function’s run another node points to a marked node, it will just ignore the marked node and all of the marked node’s neighbours. If there were to be a path to the target node through a marked node, the marked node would already have found it.
Thus, this function could end up checking the whole graph; that being the only way to ensure that there is absolutely no path.

Cycle Detection for directed graphs: “New Unpainting” method

This function is a large improvement in runtime over the “Old Unpainting” method. It is pointless to run isPath when trying to determine if a node should be unpainted. This is because if a cycle were to be found using a given node, it would be found before recursion ends on that node. Therefore, just before recursion ends on a node, it should be unpainted, regardless of potential paths to other nodes. It should be left with a “paint-chip” so that future nodes with this “paint-chip’ed neighbour know it can safely be avoided.
Although, in a worst case run, this algorithm still must check each node once.

Cycle Detection for directed graphs: “isPath” method

The function isPath could be used to detect cycles on its own through supplying the source node as both the source (obviously), and the target. A simple modification to the function was required to avoid automatically detecting a cycle after checking the very first neighbour.

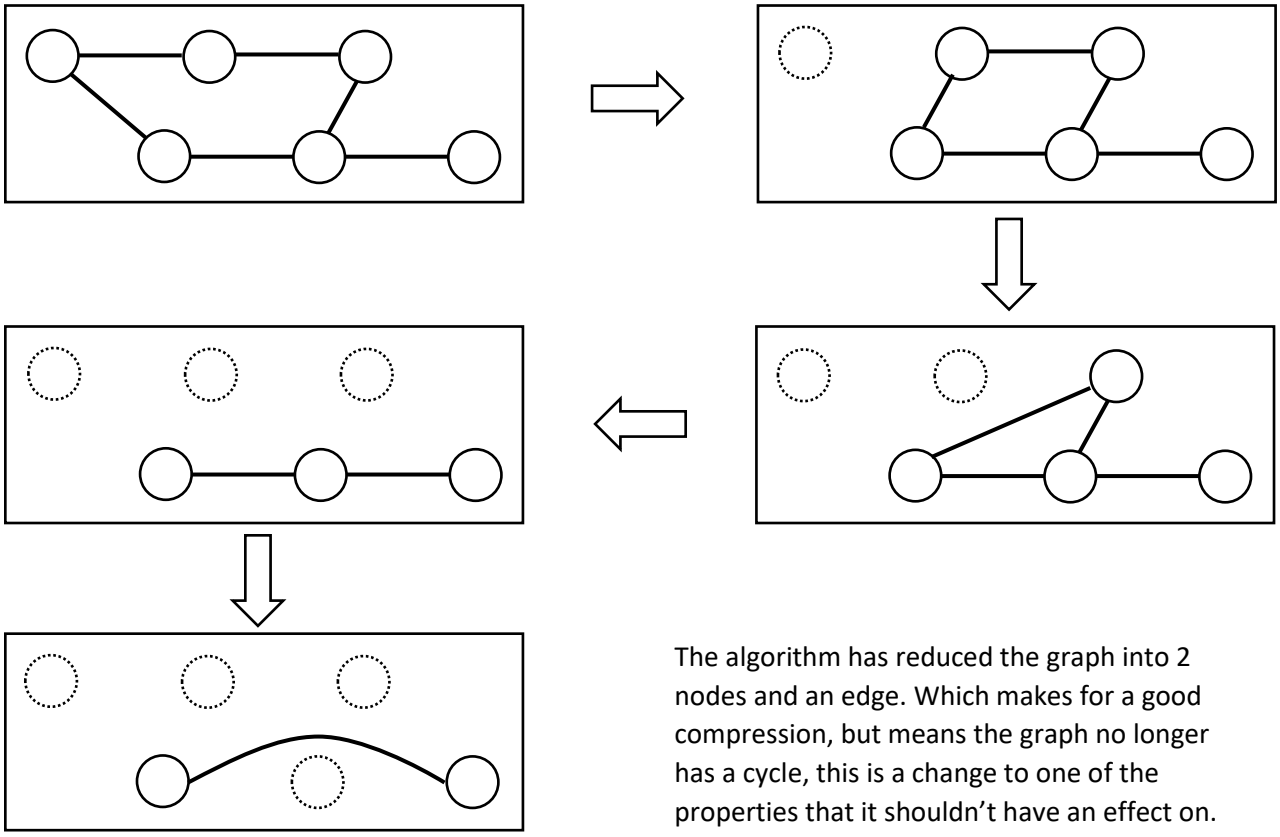
> For example: 1st node uses recursion on 2nd node, 2nd node looks at its first neighbour and finds 1st node, therefore a cycle was found. This would be a false detection of a cycle.

The small advantage of this algorithm is that it can tell if any given node is part of a cycle or not. This is due to the function only being interested in the source node, it will ignore any cycle that does not include the source node.

The large drawback of this algorithm is that it must run isPath on each and every node, therefore traversing the graph a great many times before coming to the conclusion that no cycles exist in the graph.

Node removal for undirected graphs: "Triangle Rule"

This algorithm has a lot of overhead work to it. Nodes of degree 2 are subject to removal, but there is a condition that no triangles can have their edges removed. This condition mainly applies to undirected graphs, see the example below.



The algorithm has reduced the graph into 2 nodes and an edge. Which makes for a good compression, but means the graph no longer has a cycle, this is a change to one of the properties that it shouldn't have an effect on.

Analysis:

O(n^2) for "Old Unpainting" method.
Each node has the possibility of being checked n times, while algorithm runs in O(n) time.

O(n) for "New Unpainting" method.
Each node is given a "paint-chip" preventing it from being checked for than once.

O(n) for node-removal.
Simply checks each node once, leaving linear run time.

Correctness:

Cycle detection: through “Old Unpainting Method”, “New Unpainting Method”, or “Painting Method”.

Suppose a cycle exists in a graph, and this is the only cycle in the graph.
The cycle will take the form of: $v_0v_1v_2 \dots v_{n-1}v_nv_0$. Where each v_i represents a node on the cycle, and is adjacent to both node v_{i-1} and v_{i+1} .
Note: any node on the cycle could be chosen to be the start/end node, in this case it is v_0.

The algorithm would start at a node and paint it, it then looks to the next neighbour and checks for paint. If there is paint, can return true as a cycle was found. Otherwise run the algorithm on that neighbour.
Note: some of these neighbour nodes may not end up being on the cycle.

When doing this recursion, the algorithm passes along the ID of the node that just called the algorithm, this is to ensure the newly started algorithm doesn’t immediately and falsely detect a cycle if it happens to be adjacent to the calling node.

The algorithm would recursively call itself until it reaches a point where it either detects a painted neighbour, in which case it returns true and all calling functions detect this to also return true, or it encounters a node with no neighbours (or no neighbour other than the calling node). In this situation, the algorithm would return false, and the calling function would then be able to look at its next neighbour and begin this process again.

If the graph is disconnected, the algorithm need only be called on each component. This is done through placing the initial call to the algorithm within a while loop. This ensures each node is checked (at least) once.
Note: In “Old Unpainting” and “Painting” methods, it is possible some nodes will be called upon many times. In “New Unpainting” method, this is not the case. See implementation for more details.

Thus, it is not possible for the graph to “miss” any nodes, eventually each node in the graph will have to have had the algorithm called on it.

Therefore, the algorithm must have been called on v_i , a node on the cycle. Since each v_i is at least adjacent to v_{i-1} or v_{i+1} , and those nodes themselves adjacent to other nodes on the cycle, the algorithm will eventually detect an adjacent painted node, therefore returning true and detecting a cycle.

If no cycle exists, then the algorithm would have checked every node and its neighbours at least once, therefore it can confidently declare that no cycle exists as it did not find one.

Node-removal:

Nodes are either valid or invalid for removal, cannot change that status by removing another node. This means each node only needs to be checked for removal once, which is done through calling the algorithm on each node.

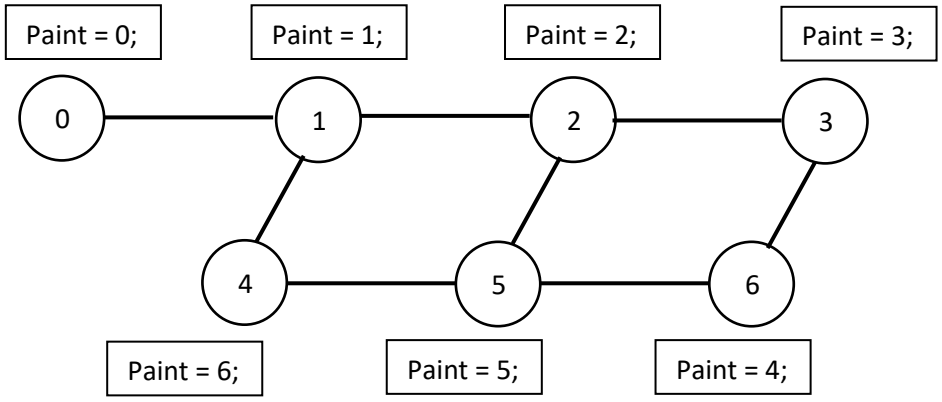
This property is because of the triangle rule explained above. A node's degree count always remains unchanged, even when a neighbour node is deleted. This means that if a node were not eligible for deletion before a neighbour was removed, it will remain not eligible after the deletion as its degree count remains unchanged.

Final thoughts:

For cycle detection, I think it would have been possible to specify the minimum length of a cycle to look for (call this value k) by using the same method as my "New Unpainting" method, but with a small modification. The modification would be in maintaining a "paint-count" for each node, and in incrementing the "paint-count" after each node is painted. Before the algorithm can conclude if a cycle has been found, would need to take current paint count and subtract the paint count of the node that would end the cycle. If that number is greater than or equal to k-1, then a cycle of length k has been found. If it's not, then just keep running the algorithm.

Suppose k = 6 for the graph here:

Recall that neighbours are incremented through in ascending order.



The algorithm would detect a cycle when it reaches node 5's neighbour, node 2.

But the paint count of node 5 minus the paint count of node 2 does not equal 5. So a cycle of specified length was not found.

Would again detect a cycle when it reaches node 4's neighbour, node 1.

The paint count of node 4 minus the paint count of node 1 is greater than 5, so a cycle of specified length is found.