Recognition Algorithms for 2-Tree Probe Interval Graphs

David Avery
Western Oregon University

Follow this and additional works at: https://digitalcommons.wou.edu/honors_theses

Part of the Computer Sciences Commons

Recommended Citation
Avery, David, "Recognition Algorithms for 2-Tree Probe Interval Graphs" (2015). Honors Senior Theses/Projects. 90.
https://digitalcommons.wou.edu/honors_theses/90

This Undergraduate Honors Thesis/Project is brought to you for free and open access by the Student Scholarship at Digital Commons@WOU. It has been accepted for inclusion in Honors Senior Theses/Projects by an authorized administrator of Digital Commons@WOU. For more information, please contact digitalcommons@wou.edu, kundas@mail.wou.edu, bakersc@mail.wou.edu.
Recognition Algorithms for 2-Tree Probe Interval Graphs

By

David J. Avery

An Honors Thesis Submitted in Partial Fulfillment
of the Requirements for Graduation from the
Western Oregon University Honors Program

Dr. Matthew Nabity,
Thesis Advisor

Dr. Gavin Keulks,
Honors Program Director

Western Oregon University
June 2015
1. Abstract

This thesis focuses on looking at a particular set of graphs and recognizing if a given graph has certain properties that would make it belong in this family, here called \textit{2-tree} Probe Interval Graphs. For these graphs, we create an algorithm to run on a coded script that recursively runs criteria through an input graph from its matrix representation to check the \textit{2-path}, and will output either a success that our graph is a \textit{2-tree} Probe Interval Graph, or failure if it is not. After the creation of this algorithm, a complexity analysis for the algorithm will be developed, as well as the implementation of different search criteria to hopefully reduce the complexity by some polynomial factor. The recognition for our set of graphs follows to the conceptual idea that triangles are built upon each other in a fashion of adding one vertex and two edges to a previous triangle in the graph. Each new triangle is added to an existing triangle and recursively builds the graph where the new vertex neighbors strictly two vertices with an existing triangle, creating a recursively defined \textit{2-path}. 
2. Introduction

We begin the first with some definitions, and the foundations of the theoretical graph model. A graph $G$ is defined as $G = (V, E)$ where $V$ is the vertex set and $E$ is the edge set from the graph. Moreover, we will be looking at simple, undirected finite graphs for our algorithms. The actual form that we will be looking for comes from the work of Brown et al. on the graph class 2-tree Probe Interval Graphs [1]. First, a graph $G$ is a Probe Interval Graph if there is a partition of $V(G)$ into $P$ and $N$ and a collection $\{I_v : v \in V(G)\}$ of closed intervals of $R$ in a one-to-one correspondence with $V(G)$ such that $uv \in E(G)$ if and only if $I_u \cap I_v \neq \emptyset$ and at least one of $u$ or $v$ belongs to $P$. The set $P$ is referred to as the probes, and the set $N$ the non-probes [1]. Although these partitions can be determined prior and used to determine the 2-tree Probe Interval Graphs, we will use a non partitioned set where we only have access to the vertices and edge sets. Probe interval graphs themselves were introduced along with the human genome project, where they were found to help with physical mapping of DNA [4, 5, 6]. We extend further to the idea of 2-tree classification for our recognition algorithms, defined recursively as

- $K_2$ is a 2-tree
- Suppose $G$ is a 2-tree; create $G'$ by adding a vertex to $G$ adjacent to both vertices of some $K_2$ of $G$

When we reference a 2-path, this is the structure that we are looking for, though in the algorithms when we construct a new vertex it is neighboring to two vertices from the previous 2-path that was found. Specifically, the 2-path is a sequence of 2 and 3-cliques, this meaning that only two distinct vertices exist in a 2-clique as the edges that are found, and that three distinct vertices exist in a 3-clique as the triangles that were found. The 2-path definition can also be likened from these alternating
and 3-cliques as they are found, for a graph $G$ with $p$ distinct 3-cliques yields $G = (e_0, t_1, e_1, t_1, \ldots, t_p, e_p)$. 
3. Procedural Overview

Before we get into the algorithms themselves, it would be worth while to explore the actual procedure that we are trying to accomplish. Below we provide a labeled input graph $G$ and adjacency matrix $M_G$. Note that $G$ has 15 vertices and that the matrix $M_G$ reflects this as well with 15 rows and columns, with the presence of an edge represented by a 1 where vertices intersect, and a 0 where there is no edge between vertices.
The first step to accomplish is to remove, or prune all vertices of degree two, degree being the number of adjacent vertices to any vertex, from $G$ in an entire sweep. While in the algorithm this can only be handled one vertex at a time, looking at $G$ and removing all degree two vertices is simple to follow, which is represented by graph $G'$ and adjacency matrix $M_{G'}$ below. Although the indexing in $M_{G'}$ doesn’t reflect the vertices that are labeled in $G'$, we will keep this labeling of vertices for easy comparison between each graph. In the actual algorithm we can’t extract this labeling, as the matrix indices would be 1 through 9, though we can still save the original index of both the vertices of degree two that are removed, as well as their neighbors upon each removal sweep.

(b) Adjacency matrix $M_G$ representation of $G$
Now that we have pruned $G$ to find $G'$, we need to perform a prune of all degree two vertices from $G'$, which we will call $G''$ and accompanying adjacency matrix $M_{G''}$. 

(c) First prune of $G$ as graph $G'$ with 9 vertices

(d) Adjacency matrix $M_{G''}$ representation of $G'$
(e) Second prune of $G$ as graph $G''$ with 6 vertices

$V_3 \quad V_5 \quad V_6 \quad V_8 \quad V_{10} \quad V_{11}$

$V_3 \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ V_5 \begin{pmatrix} 1 & 0 & 1 & 1 & 0 & 0 \\ V_6 \begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 1 \\ V_8 \begin{pmatrix} 0 & 1 & 1 & 0 & 1 & 0 \\ V_{10} \begin{pmatrix} 0 & 0 & 1 & 1 & 0 & 1 \\ V_{11} \begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 0 

(f) Adjacency matrix $M_{G''}$ representation of $G''$

After two initial sweeps of removing all degree two vertices, we are ready to check for a 2-path in the resulting graph. From the definition of the 2-tree, we first have to find a vertex of degree two to start from, here $V_3$ or $V_{11}$, assume $V_3$ is the vertex chosen. Using our 2-tree definition, it is easy to see that $V_5$ and $V_6$ are the two vertices that are adjacent to $V_3$ that make the first triangle to be saved, called $T_1$. While we save this triangle, we also need to save certain edges, on the first and last triangles there will be two edges saved. The reason that we need to save two edges is due to the fact that we need one edge to be from the starting vertex, here $V_3$, to one of its neighbors but it doesn’t matter which, here $E_1$. The second edge to save is the edge that is from two neighbors in the triangle that was just created, while also being an edge.
that is in the next triangle to be saved, which will simply be $T_2$ here. These edges will effectively branch through every vertex upon completion of the entire graph, if the graph contains a 2-path.

Once the first triangle and edges have been identified, the next vertex to find if it exists, is a vertex that hasn’t already been used in a previous triangle, and also is neighboring to exactly two vertices from the previous triangle that was found, here $V_8$. The same process of saving the accompanying triangle and edge will then occur.
Recursively continuing the same process of triangle and edge identification will continue until either we come to the last vertex that exists in the graph that hasn’t been previously used in a triangle, or the next vertex of degree two that is reached. In either case an attempt to create a triangle will be made if the vertex is neighboring to two vertices in a previous triangle, and if the vertex doesn’t meet this criteria then it simply won’t be used in a triangle. Otherwise for this example the ending result would follow.

From this example, $G''$ did indeed fit the 2-path form that we were hoping to find, as every vertex was used in a triangle at the end of recursively using the 2-tree definition. Also, along the way all of the triangle and edge references have been identified, where in the algorithm they would be saved in matrices, one for all of the triangles and one for all of the edges that were found. Lastly, from the example to understand just how the edges identified span all of the vertices in the 2-path, we will remove all of the unnecessary edges and triangle labels for illustrative purposes only. Note that the random assignment of edges in the first and last triangle would still span to all vertices if the assignment was along either $(V_3, V_6)$ and $(V_{10}, V_{11})$.
(j) Identifying spanning edges of $G''$ to all vertices
4. Initial Algorithm Idea

First, we will explore the construction of this initial algorithm idea called Path Triangulation, the first iteration of pseudocode from an original layout provided in part by Dr. Breeann Flesch, while also noting some of the shortcomings that were later resolved in Algorithm Modules. The objective here was to lay down the broad idea of how to set up the algorithm for recognizing the 2-path after all the degree two vertices had been removed through two sweeps. This initial algorithm didn’t have constraints on the input matrix, aside from the fact that it was an adjacency matrix from a bidirectional graph. The output of this algorithm is set as a logical representation of having visited all vertices in the graph, be it from sweeping over the current vertex to be checked, or as a neighbor that was used in a triangle along the algorithm sweep. The defined instance variables for this algorithm serve three distinct purposes. First, we declare a boolean array that is the same size as the order of the input matrix, where along execution of the algorithm we can mark the array location as visited when its representing vertex from the input matrix is logically visited. Then, once the sweep begins on the first vertex of degree two, we can mark it as visited, and once we come to a second degree two vertex, if at that point all array locations are marked as visited, we know we have found the desired 2-path. Second, is a triangulation matrix to hold the 3 cliques whose size is \((N - 2) \times 3\), that is there are \(N - 2\) locations for the triangles to be saved, and three array locations in each triangle to place the vertex reference, simply the vertex number. The reason that we only declare \((N - 2)\) triangle locations instead of \(N\) is for the fact that a graph of size \(N\) can only have \((N - 2)\) triangles maximally. A simple example is a triangle itself, where there would be three vertices and the single triangle. For another example, consider a square where both diagonals are present. While there is no starting vertex of degree two in this case, if another graph had such internal structure the
algorithm would iterate over it, create two instances of triangles and continue. This was a realization that the actual validation of the graph couldn’t come solely from vertex visitation, but rather edge visitation which is noted every time a triangle is created, which is handled in **Find Path Algorithm Modules** later. Lastly, we declare an edge matrix to hold the 2 cliques whose size is \((N - 1) \times 2\) for the same reason of being \((N - 1)\) as the triangulation matrix, but there is one more edge to be saved upon the last triangle that is found. Here we want to record one edge in the triangle created, thus the size 2 to hold both vertices, where after the first triangle creation the edge to be saved has the property that it shares exactly two vertices from the previous triangle that was created. In this algorithm there are two separate objectives, first to find an initial vertex to start from, and the second objective is to build the triangles and edges in execution in a recursive manner. First, we sweep over the input matrix by its rows to determine the row sum of a given vertex in the sweep. Since this comes from an adjacency matrix, a row sum of two means that the vertex has exactly two neighbors. Once such a vertex is found we simply mark it as visited, where we then find its neighbors and create our triangle and edge entries while marking each neighbor as being visited. Once completed, we need to find the next vertex to iterate over, that vertex being a neighbor to two of the vertices in the triangle created, who itself hasn’t been visited. If it is not the case that there were any such vertices we break out of execution, else we save the vertex that was found were we create triangle and edge entries based of the previous triangle created. Once this has been done we check if the vertex we are at is of degree two, and make our last edge and break from execution, else we continue to search for the next vertex. The last item in this algorithm was to check the validity of our 2-path based on the visitation of all vertices in the graph during execution, where if all of the vertices had been visited we claimed to have a 2-path.
4.1. 2 Path Design

Algorithm 4.1: Path Triangulation

**Input:** $N \times N$ Adjacency Matrix $M$ from graph of size $N$, Assume previous checks of degree sequences/size/order for validity

**Result:** Recognition of Triangulated graph or lack thereof

1. Define: Boolean Array $B(N) \times 1$ false
2. Define: Triangle Matrix $T(N - 2) \times 3$
3. Define: Edge Matrix $E(N - 2) \times 2$

for $i = 0 : N$

5. if Row sum $M[i] = 2$
6. Mark row $i$ $v_1$ visited as $B[i] = true$
7. save $i$
8. break

9. $x = 1$

while true do

11. if Row sum $M[i] = 2$ and $B[i] = true$
12. Find neighbors $j, k$ $v_2, v_3$
13. Mark row $j, k$ $v_2, v_3$ visited as $B[j], B[k] = true$
14. $t_1 = [i, i, k]$ add to $T[1]$
15. $e_0 = [i, j]$ or $[i, k]$ add to $E[0]$

else
16. Use vertices $j,k$ from $t_{x-1}$ as neighbors for $t_x = [i, j, k]$ add to $T[x]$
17. Find neighbors $j, k$ of $i$ and $t_{x-1}$, save as $e_{x-1} = [j, k]$ add to $E[x-1]$
18. if Row sum $M[i] = 2$ (Shouldn’t have concern of first degree 2) then
19. $e_x = [i, j]$ or $e_x = [i, k]$ add to $E[x]$
20. break

22. Find neighbors of $t_x$ and check $B$ for false value, if no such vertex break
23. save $i$ vertex as above
24. Mark row $i$ visited as $B[i] = true$
25. $x++$

26. If all values in $B$ are true, have found triangulated graph

What was clear during this first algorithm was that we didn’t identify first the vertices of degree two to be removed, as well as the case explained above where vertex visitation was limited in checking the validity for the 2-path form. Also, it was important to try to modulate each algorithmic aspect to its own block, such as finding the next vertex, or creating the triangles and edges.
5. Algorithm Modules

Algorithm 5.1: Algorithm Modules

Input: $N \times N$ Adjacency Matrix $M$ from graph of order $N$

Result: Recognition of 2-path or lack thereof

1. Algorithm: Initial Degree 2 Removals
2. Algorithm: Degree 2 Removed Neighbors
3. Algorithm: Initial Degree 2 Vertex
4. Algorithm: Initial Triangle and Edge Creation
5. Algorithm: Recursive Triangle and Edge Creation
6. Algorithm: New Vertex Iteration
7. Algorithm: Edge Visited Validation

Here is the overall methodology for incorporating all of the algorithm modules together to form the entire algorithm procedure. Essentially, each of these modules are taken from our running script and broken apart by their main running objective. With a proper input matrix to the initial degree removals the algorithm modules will each process their execution, where the results of one module are passed as the inputs for the next one during execution. In the case where the input matrix does not contain initial degree two vertices to remove in Algorithm: Initial Degree 2 Removals, the algorithm will not continue execution to find the underlying 2-path as it already isn’t a 2-tree. Otherwise the algorithm will execute the modules for Algorithm: Initial Degree 2 Vertex through Algorithm: New Vertex Iteration in a recursive manner until there are no more vertices to visit, or a second vertex of degree two is reached. Once the algorithm comes to a vertex of degree two anytime after Algorithm: Initial Degree 2 Vertex the algorithm will break at that point and Algorithm: Edge Visited Validation will check to see if indeed all edges in the matrix were visited during execution, and if so will confirm that we have a 2-path from our original input adjacency matrix. We will also explain each module in detail, but include high level pseudocode along with the actual module code itself.
5.1. Initial Degree 2 Removals

Our first algorithm serves the purpose of iterating over the adjacency matrix to remove all degree two vertices through two sweeps over the entire graph. What we mean by this is that we first check all vertices in the graph and for each that is of degree two we remove it, which entails removing the neighboring edges from said vertex. This process can be referenced from Procedural Overview upon the graphs $G \to G'$ as the first sweep. Once we have done this, any neighboring vertex to a removed vertex has a new smaller degree as one of its neighbors, specifically any of the degree two vertices, were removed. We simply execute the same process once more on the current matrix before we look for an underlying 2-path, which can be referenced from Procedural Overview upon the graphs $G' \to G''$ as the second sweep. We accomplish this task by encapsulating the vertex removals within a two iteration loop, where we note the current order of the matrix that we are iterating over, here a copy of the original input matrix since we will be dynamically altering the matrix when we remove vertices. We then sweep over the copied matrix and note when we have a vertex whose degree is two, and will note the row to be removed by zero valuing both the row and column, thus removing the vertex and the edges it shares with its neighbors. We also note for future reference which vertices were removed by saving their index reference in an increasing array, so in the original graph we know which vertices would need be removed. Lastly, since we didn’t actually remove the rows previously, we need to now remove them since the order of our graph needs to reflect as such. Here we use any to remove first any zero valued rows, followed by removing any zero valued columns. Since any will return a boolean value for the existence of any non zero entries in an array, we negate its usage to assign an empty array at the now zero valued vertices.
Algorithm 5.2: Initial Degree 2 Removals

**Input:** $N \times N$ Adjacency Matrix $M$ from graph of size $N$

**Result:** Removal of all degree two vertices, with two sweeps over input matrix

1. disp('Input Matrix')
2. disp(input_matrix)
3. Define: input_matrix = A
4. Define: original_matrix = input_matrix
5. Define: degree_removals = []
6. for $j = 1 : 2$
7.     Define: $N = size(input_matrix, 2)$
8.     Define: remove_matrix = input_matrix
9.     for $i = 1 : N$
10.    if $\text{sum}(input_matrix(i,:)) == 2$
11.    remove_matrix(i,:) = zeros(1,N)
12.    remove_matrix(:,i) = zeros(1,N)
13.    degree_removals(end + 1) = i
14.     input_matrix = remove_matrix
15.    disp('Pre-removal of vertices')
16.    disp(original_matrix)
17.    Define: input_matrix(\sim\text{any}(input_matrix,2),:) = []
18.    Define: input_matrix(:,\sim\text{any}(input_matrix,1)) = []
19.    Define: str = ['Input Matrix after : ',num2str(j),': two degree removals']
20.    disp(str)
21.    disp(input_matrix)
22.    display(degree_removals)
Algorithm 5.3: Initial Degree 2 Removals (Pseudocode)

1. Printout title "Input Matrix"
2. Show printout of "input_matrix"
3. Declare a new matrix with the data from input file of matrices, call it "input_matrix"
4. Declare a copy of the input matrix, call it "original_matrix"
5. Declare an array to hold removed vertices, call it "degree_removals"
6. for Begin iterative loop, up to j steps, here \( j = 2 \) do
7. 
   Declare a number that is the same size as the number of columns in "input_matrix" which is square, call it \( N \)
8. 
   Declare a new matrix called "remove_matrix" with all of the data from "input_matrix"
9. for Begin iterative loop at \( i \), up to \( N \) steps do
10. 
    if Boolean check for the row sum of the "input_matrix" at row \( i \), if the sum is two, that is there was two neighbors continue then
11. 
    Assign the entire row of "input_matrix" at \( i \) zero values from a zero array of the same row size called "zeros(1, N)"
12. 
    Assign the entire column of "input_matrix" at \( i \) zero values from a zero array of the same row size called "zeros(1, N)"
13. 
    Assign the next value in "degree_removals" that of \( i \), that is the vertex that was removed
14. 
    Assign "input_matrix" the data of "remove_matrix" so that iteration can continue for "input_matrix" at the next step
15. Printout title "Pre-removal of vertices"
16. Show printout of "input_matrix"
17. Use any function to check "input_matrix" for 0 valued matrix rows by negating to receive which rows are actually 0 valued, where we then assign the row an empty array, effectively removing the row
18. Use any function to check "input_matrix" for 0 valued matrix columns by negating to receive which columns are actually 0 valued, where we then assign the column an empty array, effectively removing the column
19. Declare title "Input Matrix after : j : two degree removals"
20. Show printout of previous title
21. Show printout of "input_matrix"
22. Show printout of "degree_removals"

5.2. Degree 2 Removed Neighbors

In this next algorithm we have to develop a way to find the neighbors from the original matrix for each vertex that was removed previously. While this could be achieved at the same time that the vertices are removed, in an attempt to modularize each
algorithm aspect we achieve this process after finishing two sweeps of removing the
degree two vertices. The way that we accomplish this is to create a zero valued matrix
with the same number of rows as vertices that were removed in Algorithm: Initial
Degree 2 Removals with two elements to store as the neighbors of said vertex. In
execution of this algorithm, while finding any neighbors in the original matrix for a
vertex that is to be removed is quite easy, we have to ensure that upon iteration that
we are finding the correct neighbors that would appear for each degree two vertex
when we sweep over the matrix twice. Essentially, there are two methods to check
for this. First, we need to sweep over all of the removed vertices while also sweeping
over entire original matrix and check if the current vertex from the original matrix is
a neighbor to the removed vertex that were are also sweeping over. Second, we have
to ensure that the neighbors that we are selecting are both neighbors to each other
as well as the current vertex. The way that we go about this is to first check if the
current vertex is of degree two, this meaning that the vertex would be removed on
the first sweep of degree two removals and ensures that either neighbor that is found
is indeed the neighbor that we are looking for. The other case is to ensure for the
vertices that would be removed on the second sweep of degree two removals, or $G' \rightarrow G''$, is that the neighbors that we choose are again neighbors to each other, and
we can accomplish this by ensuring that we aren’t looking at a vertex that would
be removed upon the first sweep of degree two removals. Once we have found the
neighbors for each removed vertex from the first and second sweeps of degree two
removals, we have all of the neighbors that we are looking for and have them as a
reference for later.
Algorithm 5.4: Degree 2 Removed Neighbors

**Input**: Matrix with 2 sweeps of degree two vertices removed
**Result**: Recognition of neighbors for degree two vertices removed

```
define removed_neighbors = zeros(size(degree_removals,2), 2)
for m = 1 : size(degree_removals, 2)
do
    define neighbor1 = 0
    define neighbor2 = 0
    for n = 1 : size(original_matrix)
do
        if original_matrix(degree_removals(m), n) == 1 && neighbor1 == 0
            for j = 1 : size(degree_removals, 2)
do
                if sum(original_matrix(degree_removals(m),:)) == 2
                    neighbor1 = n
                    break
                else if original_matrix(degree_removals(j), n) == 1 &&
                    degree_removals(j) == degree_removals(m)
                    neighbor1 = n
                    break
                else if original_matrix(degree_removals(j), n) == 1 && j ==
                    size(degree_removals, 2)
                    neighbor1 = n
            end
        end if
        else if original_matrix(degree_removals(m), n) == 1 && neighbor2 == 0
            for j = 1 : size(degree_removals, 2)
do
                if sum(original_matrix(degree_removals(m),:)) == 2
                    neighbor2 = n
                    break
                else if original_matrix(degree_removals(j), n) == 1 &&
                    degree_removals(j) == degree_removals(m)
                    neighbor2 = n
                    break
            end
        end if
    end
    removed_neighbors(m,:) = [neighbor1, neighbor2]
display(removed_neighbors)
end
```

Find Path(input_matrix)
Algorithm 5.5: Degree 2 Removed Neighbors (Pseudocode)

1. Declare a new zero valued matrix called "removed_neighbors" with the same number of rows as vertices removed from "degree_removals" by two columns.

2. for Begin iterative loop at m, up to the size of "degree_removals" do
   3. Declare a number called "neighbor1" zero valued.
   4. Declare a number called "neighbor2" zero valued.
   5. for Begin iterative loop at n, up to the size of "original_matrix" do
      6. if Boolean check for the existence of a neighbor at "original_matrix(column current removed vertex m, row n)" with the condition of "neighbor1" not already assigned then
         7. for Begin iterative loop at j, up to the size of "degree_removals" do
            8. if Boolean check to see if the current removed vertex "degree_removals" is of degree two then
               9. Assign neighbor1 the value of n and break
            10. else if Boolean check to see if the removed vertex "degree_removals" at j is a neighbor of the current vertex with the condition that j and m are the same removed vertex then
                11. Assign neighbor1 the value of n and break
            12. else if Boolean check to see if the removed vertex "degree_removals" at j is a neighbor of the current vertex with the condition that j is at the last possible iteration then
                13. Assign neighbor1 the value of n
         14. if Boolean check for the existence of a neighbor at "original_matrix(column current removed vertex m, row n)" with the condition of "neighbor2" not already assigned then
             15. for Begin iterative loop at j, up to the size of "degree_removals" do
                16. if Boolean check to see if the current removed vertex "degree_removals" is of degree two then
                   17. Assign neighbor2 the value of n and break
                18. else if Boolean check to see if the removed vertex "degree_removals" at j is a neighbor of the current vertex with the condition that j and m are the same removed vertex then
                   19. Assign neighbor2 the value of n and break
                20. Assign "removed_neighbors" the array value of ["neighbor1", "neighbor2"] at row m
      21. Show printout of "removed_neighbors"
   22. Begin algorithm check for 2-path in "input_matrix"
5.3. Initial Degree 2 Vertex

Now that we have removed all degree two vertices through two sweeps, we can being looking for the underlying 2-path. Just like in Algorithm: Initial Degree 2 Removals, we need to first find a vertex of degree two to begin the iteration of creating the first triangle upon its neighbors, where we can then build another vertex upon those two neighbors and so forth. Since this initial algorithm is passed in a new source file from the previous algorithms, and in the same source as the following algorithms, we will declare the instance variables to be used in later algorithms here. For the same reasons as Algorithm: Path Triangulation we have our output triangle and edge matrices, though we now have $N - 1$ edges to save, as in the first and final triangle creation we need to have a random edge between the degree two vertex and one of its neighbors, as well as the neighboring non degree two vertices. Next, we define two boolean variables, the first an array to note when we have visited a vertex during iteration, which will be used to confirm the next vertex throughout execution. Second, we declare a edge matrix, which will be built upon to create an adjacency matrix, where any edge that is used during execution can be added to this matrix, and will be the final validation against the input matrix once the 2-path search is complete.
Algorithm 5.6: Initial Degree 2 Vertex

**Input**: Adjacency matrix passed from Degree 2 removal (2 Steps)

**Result**: Vertex to start algorithm of triangle and edge matrices

1. **Define**: $N = \text{size}(\text{input\_matrix}, 2)$
2. **Define**: $\text{output\_triangle\_matrix} = \text{zeros}(N-2,3)$
3. **Define**: $\text{output\_edge\_matrix} = \text{zeros}(N-1,2)$
4. **Define**: $\text{bool\_visited\_arr} = \text{zeros}(1,N)$
5. **Define**: $\text{bool\_visited\_edges} = \text{zeros}(N,N)$
6. **Define**: $\text{vertex\_hold} = 0$
7. **for** $i = 1 : N$ **do**
8.   **if** $\text{sum}(\text{input\_matrix}(i,:)) == 2$ **then**
9.     $\text{bool\_visited\_arr}(1,i) = 1$
10.    $\text{vertex\_hold} = i$
11.   **break**

Algorithm 5.7: Initial Degree 2 Vertex (Pseudocode)

1. Declare a number the size of "input\_matrix"
2. Declare a zero valued matrix called "output\_triangle\_matrix" with $N-2$ rows and 3 columns
3. Declare a zero valued matrix called "output\_edge\_matrix" with $N-1$ rows and 2 columns
4. Declare a zero valued array called "bool\_visited\_arr" of size $N$
5. Declare a zero valued matrix called "bool\_visited\_edges" of size $N \times N$
6. Declare a number called "vertex\_hold" zero value
7. **for** Begin iterative loop at $i$, up to $N$ **do**
8.   **if** Boolean check for the row sum of the "input\_matrix" at row $i$, if the sum is two, that is there was two neighbors continue **then**
9.     Assign "bool\_visited\_arr" the value 1 at cell $i$
10.    Assign "vertex\_hold" the current value $i$
11.   **break out** of iteration, have starting vertex

5.4. Initial Triangle and Edge Creation

Once the starting degree two vertex has been found, we need to identify the neighbors for said vertex, as on this iteration these vertices create the first triangle, as well as randomly choosing one neighboring edge from the initial vertex. Here we sweep over the entire input matrix and check when we have a neighboring vertex. Once we find the first neighboring vertex we save it, and continue the sweep until we find the second neighboring vertex where one exists, and mark it as the second vertex with
the previous condition of the first neighbor already being assigned to a vertex. Once
we have all three vertices, we create our first triangle, as well randomly choosing
which edge to save between the initial vertex and one neighbor that was found. Once
we have completed saving the triangle and edge entries, we build this representation
into the edge matrix that will be used later for validation, with bidirectional edge
assignments between each vertex. At this point we can begin the search for our next
vertex in Algorithm : New Vertex Iteration, who is a neighbor of exactly two
vertices in the triangle that was just added.
Algorithm 5.8: Initial Triangle and Edge Creation

Input: Initial vertex of degree 2
Result: Creation of the first triangle and edge entries

for $i = 1 : N - 2$ do
    if $\sum(input\_matrix(vertex\_hold,:)) == 2 \&\& bool\_visited\_arr(1,vertex\_hold) == 1$ then
        neighbor1 = 0
        neighbor2 = 0
        for $j = 1 : N$ do
            if $input\_matrix(vertex\_hold, j) == 1 \&\& neighbor1 == 0$ then
                neighbor1 = j
                bool\_visited\_arr(neighbor1) = 1
            else if $input\_matrix(vertex\_hold, j) == 1$ then
                neighbor2 = j
                bool\_visited\_arr(neighbor2) = 1
            end
        end
        if $input\_matrix(neighbor1, neighbor2) == 1$ then
            output\_triangle\_matrix(i,:) = [vertex\_hold, neighbor1, neighbor2]
            x = round(rand(1))
            if $x == 0$ then
                output\_edge\_matrix(i,:) = [vertex\_hold, neighbor1]
            else if $x == 1$ then
                output\_edge\_matrix(i,:) = [vertex\_hold, neighbor2]
                bool\_visited\_edges(vertex\_hold, neighbor1) = 1
                bool\_visited\_edges(neighbor1, vertex\_hold) = 1
                bool\_visited\_edges(vertex\_hold, neighbor2) = 1
                bool\_visited\_edges(neighbor2, vertex\_hold) = 1
                bool\_visited\_edges(neighbor1, neighbor2) = 1
                bool\_visited\_edges(neighbor2, neighbor1) = 1
            else
                break
            end
        end
    end
end

Algorithm : Recursive Triangle and Edge Creation
Algorithm : New Vertex Iteration
Algorithm 5.9: Initial Triangle and Edge Creation (Pseudocode)

```plaintext
for Begin iterative loop at i, up to N-2 do
  if Boolean check if the row sum of "input_matrix" at "vertex_hold" is 2
    with the condition of "bool_visited_arr" at "vertex Holds" being 1 then
      Declare a number called "neighbor1" zero valued
      Declare a number called "neighbor2" zero valued
    for Begin iterative loop at j, up to N do
      if Boolean check if "input_matrix" at row "vertexhold" column j
        is 1 with the condition of "neighbor1" not already assigned then
        Assign "neighbor1" the value j
        Assign "bool_visited_arr" at "neighbor1" value 1, visited
      else if Boolean check if "input_matrix" at row "vertexhold"
        column j is 1 previous condition implies "neighbor1" has been
        assigned then
        Assign "neighbor2" the value j
        Assign "bool_visited_arr" at "neighbor2" value 1, visited
      if Boolean check if "input_matrix" at row "neighbor1" column
        "neighbor2" is 1 that is they are both neighbors then
        Assign "output_triangle_matrix" at i the array ["vertex_hold",
        "neighbor1", "neighbor2"]
        Declare a number x the random value 0 or 1
        if Boolean check if x is 0 then
          Assign "output_edge_matrix" at i the array ["vertex_hold",
          "neighbor1"]
        else if Boolean check if x is 1 then
          Assign "output_edge_matrix" at i the array ["vertex_hold",
          "neighbor2"]
        Assign the edges between all vertices in the last triangle as
        visited, bidirectional
      else
        break
    else
      Begin algorithm check for other triangles and edges
  else
    Begin algorithm check for next vertex
```

5.5. Recursive Triangle and Edge Creation

After a new vertex has been found from Algorithm: New Vertex Iteration we can create the triangle and edge entries and save each to their matrices. Although
the desired result is the same of that in **Algorithm : Initial Triangle and Edge Creation**, we will use a simpler approach since we have the previous information that we need from the last triangle that was created. This comes both from **Algorithm : Initial Triangle and Edge Creation** and **Algorithm : New Vertex Iteration** since we already have a vertex that is a neighbor to two vertices in the previous triangle, we only need to find which two vertices from the previous triangle are neighboring to the current vertex. Note, this could certainly be accomplished at the time that the new vertex was found, but attempting to modularize we will accomplish the task in this algorithm. Thus, we first note that our current vertex is visited, and then logically find which vertices in the triangle are neighboring, and lastly will add both to the triangle, and the edge between the neighbors of the current vertex, while also building up the edge matrix to be used for validation. Upon the case that we are at the last possible vertex in the sweep, we will randomly assign an extra edge between the last vertex and a neighboring vertex, see **Algorithm : Procedural Overview** (i) for reference.
Algorithm 5.10: Recursive Triangle and Edge Creation

**Input**: Saved vertex from **Algorithm**: New Vertex Iteration which will be used to build triangles and edges, is vertex with two new neighbors

**Result**: New triangles and edges created from current vertex, are from new vertices

1 Define: bool visited_arr(1, vertex hold) = 1
2 Define: neighbor1 = 0
3 Define: neighbor2 = 0
4 if input_matrix(vertex hold, output_triangle_matrix(i-1,1)) == 1 then
5   neighbor1 = output_triangle_matrix(i-1,1)
6 if input_matrix(vertex hold, output_triangle_matrix(i-1,2)) == 1 then
7   if neighbor1 == 0 then
8     neighbor1 = output_triangle_matrix(i-1,2)
9   else
10   neighbor2 = output_triangle_matrix(i-1,2)
11 if input_matrix(vertex hold, output_triangle_matrix(i-1,3)) == 1 then
12   neighbor2 = output_triangle_matrix(i-1,3)
13 if input_matrix(neighbor1, neighbor2) == 1 then
14   output_triangle_matrix(i,:) = [vertex hold, neighbor1, neighbor2]
15   output_edge_matrix(i,:) = [neighbor1, neighbor2]
16 else
17   break
18 bool visited_edges(vertex hold, neighbor1) = 1
19 bool visited_edges(neighbor1, vertex hold) = 1
20 bool visited_edges(vertex hold, neighbor2) = 1
21 bool visited_edges(neighbor2, vertex hold) = 1
22 bool visited_edges(neighbor2, neighbor1) = 1
23 bool visited_edges(neighbor1, neighbor2) = 1
24 if i == N-2 then
25   x = round(rand(1))
26   if x == 0 then
27     output_edge_matrix(i+1,:) = [vertex hold, neighbor1]
28   else if x == 1 then
29     output_edge_matrix(i+1,:) = [vertex hold, neighbor2]
Algorithm 5.11: Recursive Triangle and Edge Creation (Pseudocode)

1. Assign "bool_visited_arr" at "vertex_hold" the value 1 that is visited
2. Declare a number called "neighbor1" zero valued
3. Declare a number called "neighbor2" zero valued
4. if Boolean check if "input_matrix" at row "vertex_hold" column from
   "output_triangle_matrix(i-1,1)" is 1 then
   5. Assign "neighbor1" the value of the first vertex of
      "output_triangle_matrix" at i-1
5. if Boolean check if "input_matrix" at row "vertex_hold" column from
   "output_triangle_matrix(i-1,2)" is 1 then
6. if Boolean check if "neighbor1" has not yet been assigned then
7.   Assign "neighbor1" the value of the second vertex of
8.   "output_triangle_matrix" at i-1
9. else
10. Assign "neighbor2" the value of the second vertex of
11. "output_triangle_matrix" at i-1
12. if Boolean check if "input_matrix" at row "vertex_hold" column from
13.   "output_triangle_matrix(i-1,3)" is 1 previous conditions imply that
14.   "neighbor1" has been assigned then
15.   Assign "neighbor2" the value of the third vertex of
16.   "output_triangle_matrix" at i-1
17. if Boolean check if "input_matrix" at row "neighbor1" column "neighbor2"
18.   is 1 that is they are both neighbors then
19.   Assign "output_triangle_matrix" at i the array ["vertex_hold",
20.   "neighbor1", "neighbor2"]
21.   Assign "output_edge_matrix" at i the array ["neighbor1", "neighbor2"]
22. else
23.   break
24. Assign the edges between all vertices in the last triangle as visited,
25. bidirectional
26. if Boolean check if i = N-2 that is the last iteration then
27. if Boolean check if x is 0 then
28.   Assign "output_edge_matrix" at i+1 the array ["vertex_hold",
29.   "neighbor1"]
30. else if Boolean check if x is 1 then
31.   Assign "output_edge_matrix" at i+1 the array ["vertex_hold",
32.   "neighbor2"]
5.6. New Vertex Iteration

From either of the algorithms Algorithm: Initial Triangle and Edge Creation and Algorithm: Recursive Triangle and Edge Creation they will need the next vertex in iteration to continue from, which occurs once for the initial algorithm and as many times as needed for the recursive algorithm. Since both of the previous algorithms have their output as a new triangle created, we need to verify through our input matrix if an unvisited vertex is a neighbor to exactly two vertices from the previous triangle that was created. We can accomplish this with logical expressions explained below that cover all the possible cases of neighboring vertices to the triangle. As we cannot be assured the the last two entries in the triangle are the neighbors of the new vertex, we need to check all vertices in the triangle essentially together. We first check for the first neighbor of the new vertex, for this is certainly one of the neighbors. Second, we check against the other two vertices simultaneously to see if the new vertex is only neighboring to one of the remaining vertices in the triangle. Here, in the case that there is no such vertex, since this algorithm is encapsulated within a loop, no vertex will be visited on the sweep, thus nullifying the existence of a 2-path.
Algorithm 5.12: New Vertex Iteration

**Input:** Previous triangle created from Algorithm: Initial Triangle and Edge Creation or Algorithm: Recursive Triangle and Edge Creation

**Result:** An unvisited vertex who is a neighbor to exactly two vertices from the previous triangle created

```plaintext
1 for \( j = 1 : N \) do
  2     if bool_visited_arr(1,j) == 1 then
  3         continue
  4     if input_matrix(j, output_triangle_matrix(i,1)) == 1 then
  5         if input_matrix(j, output_triangle_matrix(i,2)) == 1 &&
            input_matrix(j, output_triangle_matrix(i,3)) == 0 then
  6           vertex_hold = j
  7         else if input_matrix(j, output_triangle_matrix(i,2)) == 0 &&
            input_matrix(j, output_triangle_matrix(i,3)) == 1 then
  8           vertex_hold = j
  9     else if input_matrix(j, output_triangle_matrix(i,2)) == 1 then
 10        if input_matrix(j, output_triangle_matrix(i,1)) == 1 &&
            input_matrix(j, output_triangle_matrix(i,3)) == 0 then
 11           vertex_hold = j
 12        else if input_matrix(j, output_triangle_matrix(i,1)) == 0 &&
            input_matrix(j, output_triangle_matrix(i,3)) == 1 then
 13           vertex_hold = j
 14     else if input_matrix(j, output_triangle_matrix(i,3)) == 1 then
 15        if input_matrix(j, output_triangle_matrix(i,1)) == 1 &&
            input_matrix(j, output_triangle_matrix(i,2)) == 0 then
 16           vertex_hold = j
 17        else if input_matrix(j, output_triangle_matrix(i,1)) == 0 &&
            input_matrix(j, output_triangle_matrix(i,2)) == 1 then
 18           vertex_hold = j
```
Algorithm 5.13: New Vertex Iteration (Pseudocode)

1. for Begin iterative loop at \( j \), up to \( N \) do
2.   if Boolean check if "bool_visited_arr" at \( j \) is 1 that is visited then
3.     continue
4.   if Boolean check if "input_matrix" at row \( j \) column from the first vertex at \( i \) in "output_triangle_matrix" is 1 that is they are neighbors then
5.     if Boolean check if "input_matrix" at row \( j \) column from the second vertex at \( i \) in "output_triangle_matrix" is 1 that is they are neighbors with the condition that \( j \) is also not a neighbor of the third vertex at \( i \) in "output_triangle_matrix" then
6.       Assign "vertex_hold" the value of \( j \)
7.     else if Boolean check if "input_matrix" at row \( j \) column from the second vertex at \( i \) in "output_triangle_matrix" is 0 that is they are not neighbors with the condition that \( j \) is also a neighbor of the third vertex at \( i \) in "output_triangle_matrix" then
8.       Assign "vertex_hold" the value of \( j \)
9.   else if Boolean check if "input_matrix" at row \( j \) column from the second vertex at \( i \) in "output_triangle_matrix" is 1 that is they are neighbors then
10.  if Boolean check if "input_matrix" at row \( j \) column from the first vertex at \( i \) in "output_triangle_matrix" is 1 that is they are neighbors with the condition that \( j \) is also not a neighbor of the third vertex at \( i \) in "output_triangle_matrix" then
11.    Assign "vertex_hold" the value of \( j \)
12.  else if Boolean check if "input_matrix" at row \( j \) column from the first vertex at \( i \) in "output_triangle_matrix" is 0 that is they are not neighbors with the condition that \( j \) is also a neighbor of the third vertex at \( i \) in "output_triangle_matrix" then
13.    Assign "vertex_hold" the value of \( j \)
14.  else if Boolean check if "input_matrix" at row \( j \) column from the third vertex at \( i \) in "output_triangle_matrix" is 1 that is they are neighbors then
15.  if Boolean check if "input_matrix" at row \( j \) column from the first vertex at \( i \) in "output_triangle_matrix" is 1 that is they are neighbors with the condition that \( j \) is also not a neighbor of the second vertex at \( i \) in "output_triangle_matrix" then
16.    Assign "vertex_hold" the value of \( j \)
17.  else if Boolean check if "input_matrix" at row \( j \) column from the first vertex at \( i \) in "output_triangle_matrix" is 0 that is they are not neighbors with the condition that \( j \) is also a neighbor of the second vertex at \( i \) in "output_triangle_matrix" then
18.    Assign "vertex_hold" the value of \( j \)
5.7. Edge Visited Validation

The final step in processing the 2-path is to check if all the edges were used during the algorithm sweep. We have to do this instead of looking solely at the visitation of vertices due to the degree of each vertex. Since during the sweep the only concern is looking for a new vertex who is a neighbor to two vertices in the previous triangle, we cannot account for extra edges in this new vertex that may be connected to another vertices that wouldn’t fit the recursive 2-path definition. Although the vertex itself would be visited, we need to effectively assure that each edge from each new vertex is created only from two vertices in the previous triangle. For this reason, we create a new matrix that assures a 2-path, and allow the final validation to come from equality to the input matrix. This is quickly handled a matrix equality check, which will return true when both given matrix inputs are indeed equal, in both the row and column sense. If it is the case that both matrices are equal then we have assured that a 2-path was found, otherwise there were extra edges in the graph that invalidated such form.

Algorithm 5.14: Edge Visited Validation

<table>
<thead>
<tr>
<th>Input:</th>
<th>Matrix representation of all the edges which were visited during execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result:</td>
<td>Final result of input graph having 2-path or not, with edge and triangle matrices created</td>
</tr>
</tbody>
</table>

```
1 if isequal(bool_visited_edges, input_matrix) then
2   Printout title "Found a 2-path!"
3 else
4   Printout title "Not a 2-path, edges not used."
```

Algorithm 5.15: Edge Visited Validation (Pseudocode)

```
1 if Boolean check if "bool_visited_edges" and "input_matrix" are equal using built in function isequal then
2   Printout title "Found a 2-path!"
3 else
4   Printout title "Not a 2-path, edges not used."
```
6. Runtime Complexity

Another aspect to the thesis is to determine the runtime complexity of the entire algorithm. Like other probe interval graphs from the works of Busch et al. and McConnell et al., although linear time complexity algorithms have been found, we expect to find a polynomial factor complexity, with improvements reducing the polynomial factor to $O(N^2)$ if possible [2, 3]. However, the work upon linear time recognition occurs from partitioned sets of probes and non probes, where non partitioned sets were found to be bounded by $O(N^2)$, remembering that our examination occurs for non partitioned sets. This runtime complexity analysis will be in terms of Big Oh as a worst case analysis, which guarantees a upper bound on the algorithm. We accomplish this by looking at each algorithm individually and determine the complexity of each, and note the complexity for the entire algorithm where encapsulated algorithms exist. To make this process simple, we will look at a line by line examination and expression for each algorithm and determine individual element complexities and subsequent algorithm complexity. Lastly, for each recursive polynomial time elements, we will express by asymptotic computational complexity, as inputs whose size is smaller than the constant factor associated with each order are too small to be of concern, thus only conserving the complexity of the highest order. Since the modules use similar functionality, aside from those that use built in functionality that is provided, we will examine the possible complexities that we encounter here, rather than repetitively expressing them. First, beginning at the simplest complexity we have the constant time expressions. These expressions deal with single value assignments, like integer creation, or boolean checks of validity for a set expression, expressions whose execution is set for any given input size, here the size of the adjacency matrix. For our algorithm, constant time expressions are mostly integer assignments, like that of setting the new vertex at each iteration, or the boolean checks for the neighbor of a given triangle where we are comparing a specific matrix value lookup to another integer.
Second, the linear time complexity expressions, are dependent on the given input size, here the order of the graph, and grow at a linear rate with respect to said size. Expressions such as array declaration and sweeps over the all the vertices fall in this category for our algorithms. The interesting case here, however, is the complexity for array declaration of a set length. Although for example when our algorithm assigns a new triangle, there are always three elements to add, that is one array of size three that one may expect to execute in constant time. The problem with this assumption is that upon compilation, we can’t be certain how this is handled. Thus, to assure a upper bound we bound each of these expressions in linear time. For the highest run-time complexity order, we have polynomial $O(N^2)$ time, which for a single expression in our algorithm comes down to matrix declaration and sweeps. The reasoning for this is due to the fact that depending on the compiler, essentially it is stored as an array, whose entry is itself another array, thus the $N \times N$ size and subsequent matrix entries. Finally, there is also the occurrence of encapsulation of one complexity order within another. For example, a constant time expression that is encapsulated within a linear expression, itself will execute $N \times 1$, or $N$ total steps. Thus, where ever there is one complexity order within another, we need to be careful to express the entire expression with the factoring of all complexity orders together.
6.1. Initial Degree 2 Removals (Complexity)

**Algorithm 6.1:** Initial Degree 2 Removals (Complexity)

1. $O(N^2)$ matrix declaration
2. $O(N^2)$ matrix declaration
3. $O(N)$ array declaration
4. for $O(2)$ iteration of 2 steps do
   5. $O(1)$ integer declaration
   6. $O(N^2)$ matrix declaration
   7. for $O(N)$ iteration of $N$ steps do
      8. if $O(N)$ boolean iteration of $N$ steps then
         9. $O(N)$ array declaration
         10. $O(N)$ array declaration
         11. $O(1)$ integer declaration
      12. $O(N^2)$ matrix declaration
   13. $O(N^2)$ array declaration encapsulated in $N$ steps
   14. $O(N^2)$ array declaration encapsulated in $N$ steps

Line 7 block $N(N^2 + 3N + 1) = (N^3 + 3N^2 + N)$
Line 4 block $2(N^3 + 3N^2 + N + N^2 + 1) = 2N^3 + 8N^2 + 2N + 2$
Total $2N^3 + 8N^2 + 2N + 2 + 4N^2 + N = 2N^3 + 12N^2 + 3N + 2$

**Algorithm :** Initial Degree 2 Removals $O(N^3)$
6.2. Degree 2 Removed Neighbors (Complexity)

Algorithm 6.2: Degree 2 Removed Neighbors

1. \( O(N^2) \) matrix declaration
2. for \( O(N) \) iteration of \( N \) steps do
   3. \( O(1) \) integer declaration
   4. \( O(1) \) integer declaration
   5. for \( O(N) \) iteration of \( N \) steps do
      6. if \( O(2) \) boolean checks at matrix index then
         7. for \( O(N) \) iteration of \( N \) steps do
            8. if \( O(N) \) boolean iteration of \( N \) steps then
               9. \( O(1) \) integer declaration
            10. else if \( O(2) \) boolean checks at matrix index then
                11. \( O(1) \) integer declaration
            12. else if \( O(2) \) boolean checks at matrix index then
                13. \( O(1) \) integer declaration
         14. else if \( O(2) \) boolean checks at matrix index then
            15. for \( O(N) \) iteration of \( N \) steps do
               16. if \( O(N) \) boolean iteration of \( N \) steps then
                  17. \( O(1) \) integer declaration
               18. else if \( O(2) \) boolean checks at matrix index then
                  19. \( O(1) \) integer declaration
         20. \( O(N) \) array declaration

Line 7 block \( N(N + 7) = N^2 + 7N \)
Line 15 block \( N(N + 4) = N^2 + 4N \)
Line 5 block \( N(N^2 + 7N + 2 + N^2 + 4N + 2) = 2N^3 + 11N^2 + 4N \)
Line 2 block \( N(2N^3 + 11N^2 + 4N + 2 + N) = 2N^4 + 11N^3 + 5N^2 + 2N \)
Total \( N + 2N^4 + 11N^3 + 5N^2 + 2N = 2N^4 + 11N^3 + 5N^2 + 3N \)
Algorithm: Degree 2 Removed Neighbors \( O(N^4) \)
6.3. Initial Degree 2 Vertex (Complexity)

<table>
<thead>
<tr>
<th>Algorithm 6.3: Initial Degree 2 Vertex (Complexity)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. $O(1)$ integer declaration</td>
</tr>
<tr>
<td>2. $O(N^2)$ matrix declaration</td>
</tr>
<tr>
<td>3. $O(N^2)$ matrix declaration</td>
</tr>
<tr>
<td>4. $O(N)$ array declaration</td>
</tr>
<tr>
<td>5. $O(N^2)$ matrix declaration</td>
</tr>
<tr>
<td>6. $O(1)$ integer declaration</td>
</tr>
<tr>
<td>7. for $O(N)$ iteration of $N$ steps do</td>
</tr>
<tr>
<td>8. if $O(N)$ boolean iteration of $N$ steps then</td>
</tr>
<tr>
<td>9. $O(1)$ integer declaration on array</td>
</tr>
<tr>
<td>10. $O(1)$ integer declaration</td>
</tr>
</tbody>
</table>

Line 7 block $N(N + 2) = N^2 + 2N$
Total $3N^2 + N + 2 + N^2 + 2N = 4N^2 + 3N + 2$

Algorithm: Initial Degree 2 Vertex $O(N^2)$
6.4. Initial Triangle and Edge Creation (Complexity)

**Algorithm 6.4: Initial Triangle and Edge Creation**

1. for $O(N - 2)$ iteration of $N$-2 steps do
2.   if $O(N + 1)$ boolean iteration of $N$ steps then
3.     $O(1)$ integer declaration
4.     $O(1)$ integer declaration
5.   for $O(N)$ iteration of $N$ steps do
6.     if $O(2)$ boolean checks at matrix index then
7.       $O(1)$ integer declaration
8.       $O(1)$ integer declaration
9.     else if $O(1)$ boolean checks at matrix index then
10.    $O(1)$ integer declaration
11.   $O(1)$ integer declaration on array
12.   if $O(1)$ boolean checks at matrix index then
13.    $O(N)$ array declaration in matrix
14.    $O(1)$ integer declaration
15.   if $O(1)$ boolean check then
16.    $O(N)$ array declaration in matrix
17.   else if $O(1)$ boolean check then
18.    $O(N)$ array declaration in matrix
19.    $O(1)$ integer declaration on array
20.    $O(1)$ integer declaration on array
21.    $O(1)$ integer declaration on array
22.    $O(1)$ integer declaration on array
23.    $O(1)$ integer declaration on array
24.    $O(1)$ integer declaration on array
25.   else
26.   Algorithm: Recursive Triangle and Edge Creation
27.   Algorithm: New Vertex Iteration

Line 5 block $N(7) = 7N$
Line 1 block $(N - 2)(N + 3 + 7N + 3N + 10) = 11N^2 - 11N - 26$

Initial Triangle and Edge Creation $O(N^2)$
6.5. Recursive Triangle and Edge Creation (Complexity)

**Algorithm 6.5:** Recursive Triangle and Edge Creation (Complexity)

1. $O(1)$ integer declaration on array
2. $O(1)$ integer declaration
3. $O(1)$ integer declaration
4. if $O(1)$ boolean checks at matrix index then
   5. $O(1)$ integer declaration
5. if $O(1)$ boolean checks at matrix index then
   6. $O(1)$ integer declaration
   7. if $O(1)$ boolean check then
      8. $O(1)$ integer declaration
   9. else
   10. $O(1)$ integer declaration
6. if $O(1)$ boolean checks at matrix index then
7. $O(1)$ integer declaration
8. if $O(1)$ boolean checks at matrix index then
9. $O(N)$ array declaration
10. $O(N)$ array declaration
11. else
12. $O(1)$ integer declaration on array
13. $O(1)$ integer declaration on array
14. $O(1)$ integer declaration on array
15. $O(1)$ integer declaration on array
16. $O(1)$ integer declaration on array
17. if $O(1)$ boolean check then
18. $O(1)$ integer declaration
19. if $O(1)$ boolean check then
20. $O(N)$ array declaration
21. else if $O(1)$ boolean check then
22. $O(N)$ array declaration

Total $4N + 22$

Recursive Triangle and Edge Creation $O(N)$
6.6. New Vertex Iteration (Complexity)

**Algorithm 6.6: New Vertex Iteration (Complexity)**

```
1 for \(O(N)\) iteration of \(N\) steps do
2   if \(O(1)\) boolean checks at array index then
3     if \(O(1)\) boolean checks at matrix index then
4       if \(O(2)\) boolean checks at matrix index then
5         \(O(1)\) integer declaration
6       else if \(O(1)\) boolean checks at matrix index then
7         \(O(1)\) integer declaration
8     else if \(O(1)\) boolean checks at matrix index then
9       if \(O(2)\) boolean checks at matrix index then
10      \(O(1)\) integer declaration
11     else if \(O(1)\) boolean checks at matrix index then
12      \(O(1)\) integer declaration
13     else if \(O(1)\) boolean checks at matrix index then
14       if \(O(2)\) boolean checks at matrix index then
15         \(O(1)\) integer declaration
16       else if \(O(1)\) boolean checks at matrix index then
17         \(O(1)\) integer declaration
```

Line 1 block \(N(22) = 22N\)
Total = \(22N\)
**New Vertex Iteration \(O(N)\)**

6.7. Edge Visited Validation (Complexity)

**Algorithm 6.7: Edge Visited Validation (Complexity)**

```
1 if \(O(N^2)\) boolean check on matrix entries then
2 else
```

Total \(N^2\)
**Edge Visited Validation \(O(N^2)\)**
6.8. Algorithm Modules (Complexity)

Algorithm 6.8: Algorithm Modules (Complexity)

1. Algorithm: Initial degree removals $O(N^3)$
2. Algorithm: Neighbors of degree two removed $O(N^4)$
3. Algorithm: Initial vertex degree 2 search $O(N^2)$
4. Algorithm: Initial Triangle and Edge Creation $O(N^3)$
5. Algorithm: Recursive Triangle and Edge Creation $O(N)$
6. Algorithm: New vertex iteration $O(N)$
7. Algorithm: Edge visited validation $O(N^2)$

Total $(2N^3 + 12N^2 + 3N + 2) + (2N^4 + 11N^3 + 5N^2 + 3N) + (4N^2 + 3N + 2) + (N - 2)(4N + 22 + 22N + 11N + 13) + N^2 = 2N^4 + 13N^3 + 59N^2 + 34N - 120$

Find Path Algorithm Modules $O(N^4)$

7. Conclusion

What we have found for our algorithms is that a complexity analysis bounds at a polynomial factor of $O(N^4)$. This was a polynomial factor that we were expecting to find from the onset, but looking at the algorithms there are aspects that can likely be reduced to a polynomial factor of $O(N^3)$, perhaps $O(N^2)$. First, we previously explained how finding the neighbors of the degree two removals can be handled in $O(N^2)$ time, but the modularization to separate each algorithmic result from another deemed this to be an issue. Another enhancement in the initial process of removing all degree two vertices from $O(N^3)$ to $O(N^2)$ is likely to be accomplished if we were able to remove vertices without having to copy the matrix at each sweep. The reasoning that we copy the matrices is for the fact that we can easily assure that we are only removing the degree two vertices that exist for the entire matrix at each iteration, without the problem of removing a vertex first and then causing a subsequent vertex to have a degree of two by the fact that its degree two neighbor was previously removed. Later iterations of the algorithm may be able to address these issues, but for this project we can be happy that we first have created an algorithm that can find a 2-path from a given input matrix. We can also be happy that we developed edge
and triangle matrices to be checked for the Probe Interval Form, while extensively looking at the underlying 2-path that exists, and finding a polynomial upper bound runtime for all algorithm modules.

To overview, it is important to clearly restate the purpose of this project, as well as the contributions that I made, along with future contributions that are still needed to accomplish the 2-tree Probe Interval Graph recognition. First, the primary goal of this project was to create a standalone script that could sweep over the interpretation of an input graph from its adjacency matrix, and determine if the graph contains an underlying 2-path. Before the 2-path itself can be found, the graph needs to have two prunes of all degree two vertices, which can later be used to validate the Probe Interval form. If the prune is successful, the remaining graph has to be processed to match the recursive 2-tree definition, again which we call in execution a 2-path, which comes from triangle and spanning edge identifications. Once the entire graph has been swept through this process, if it did indeed contain a 2-path, then the output triangles and edges can be used with the initial degree two removals to check for the entire 2-tree Probe Interval form. The second purpose of this project was to determine the complexity of the entire algorithm that was created. The complexity analysis that was chosen was a Big Oh upper bound analysis, that under any input graph we can have a precise upper count of the number of steps that the algorithm would take, as well as a factor that bounds with respect to the order of the graph, which was found to be $O(N^4)$. These were my primary contributions to the project, where the complexity analysis was mostly my own idea as well as the process of determining the complexity factor, and also designing/creating the actual algorithm itself. However, an overview of pseudocode of how to accomplish the task was provided in part by the works of Brown, Flesch, and Lundgren [1], along with contributions from my advisor Dr. Matthew Nabity. The actual question of developing an algorithm to determine if an input graph is of the 2-tree Probe Interval form is still open and ready to be
accomplished using the results of this work, with the remaining work requiring the actual Probe Interval Graph to be further explored. Also, the algorithm developed here can itself see improvements in complexity, with real world timings from varying graphs to be explored to see just how complexity improvements can affect the runtime for the algorithm.
References


