On the Relationship Between Edge Connectivity and Edge-Disjoint Spanning Trees

Beatrice Aragones

December 23, 2022

Abstract

E.D. Palmer wrote a paper on the spanning tree packing number of a graph. Within it, he briefly mentioned a conjecture regarding a specific graph of 15 vertices being the smallest possible one to only have one single spanning tree despite it being a 4-regular graph. This report first introduces how Palmer's conjecture came to be developed, using previously made theorems to support it. Two different algorithms that are key to demonstrating the conjecture are analyzed next, the first being the Ford-Fulkerson algorithm and the second being the Roskind-Tarjan algorithm. In the end, Palmer's conjecture is explicitly verified using the two aforementioned algorithms as a test on various 4-regular graphs containing up to 14 vertices, as well as the special 15-vertex graph.

1 Introduction and Background Information

Connectivity is one of the most fundamental properties of graphs that people study, where many different theorems and corollaries can stem from the concept alone. In particular, edge connectivity is an important sub-topic where its applications can be seen in real life. For example, one can think of edge connectivity in maps and networks as the minimum number of roads that can be closed off for construction before it is impossible to be able to reach from one point of the map to another point. This is a critical topic, as it can be used to determine the best alternative routes one can take. Palmer's conjecture uses a theorem on edge connectivity as its foundation, which will be discussed later on.

A *graph* is made up of a set of vertices and edges, where *vertices* are points within the graph, and *edges* are lines that connect two vertices together. In network problems, these can be referred to as *nodes* and *arcs*, respectively. Edges that have a direction are *directed*, while edges that have no direction are *undirected*. An undirected edge can also be thought of as a directed edge pointing in both directions. Graphs that contain directed edges are called *directed graphs*, and graphs that contain undirected edges are called *undirected graphs*. A *complete* graph is a graph where every pair of vertices are connected by an edge. In other words, a graph is complete if every vertex has an edge that directly connects it to every other vertex. A *k-regular graph* is a graph where all vertices have the same *degree* of *k*, which is the total number of edges that are connected to a vertex.

In Figure 1, Graph 1 is an undirected, complete graph with 15 edges. Graph 2 is a directed, non-complete directed graph with 9 edges.

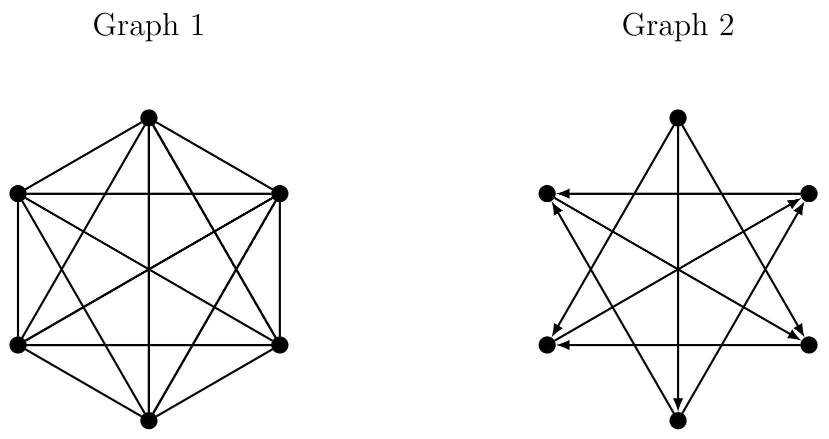

Figure 1: Two graphs containing 6 vertices each.

There are different types of subsets of graphs, called sub-graphs. One type of sub-graph is a regular graph. Another type of sub-graph is a spanning tree. A *spanning tree* is a sub-graph where all vertices have a single path that allows them to be reachable to each other without creating a cycle. A *cycle* occurs within a graph when a path starts and ends at the same vertex. As such, in order to avoid a cycle, a spanning tree will always have $V - 1$ edges, where V is the total number of vertices. Graphs also have a *spanning tree packing number*, given by the following definition:

Definition 1. The *spanning tree packing number* is the maximum number of edge-disjoint spanning trees that can be found within the graph, where different spanning trees do not share any edges.

Graph 1 of Figure 1 is a 5-regular graph, shown by the number of edges connected to a vertex in Subgraph 1. The set of edges also makes up a spanning tree. It has a spanning tree packing number of 3, as shown in Subgraph 2 of Figure 2, where each color corresponds to a different spanning tree. In Subgraph 3, the set of black edges creates a spanning tree. However, once the blue edge is included in the set, a cycle is created and is no longer a spanning tree.

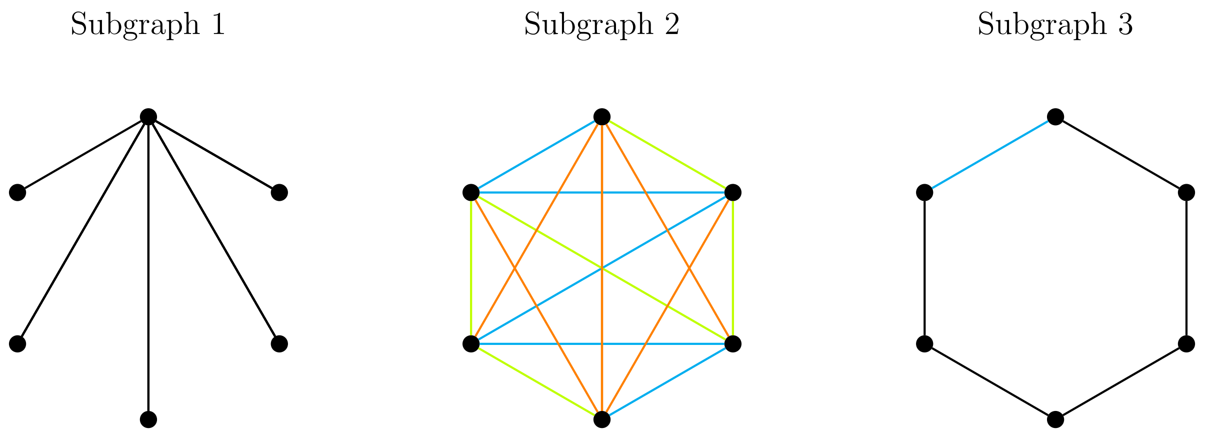

Figure 2: Three subgraphs within Graph 1 of Figure 1.

A graph is *connected* if every vertex within the graph is reachable to each other via some path. If there is at least one vertex that cannot be reachable by the other vertices, then the graph is *disconnected*. As stated previously, connectivity is one of the most foundational concepts of graph theory and can be broken down into two sub-topics: edge connectivity and vertex connectivity. For our purposes, the primary focus will be on edge connectivity. *Edge connectivity* is defined as the minimum number of edges that can be removed from a graph before it becomes disconnected. As a result, a graph is said to be *k-edge-connected* if removing any set of $k-1$ edges does not disconnect the graph.

Graph 3 is a disconnected graph due to the blue vertex not being reachable by the other 3 vertices. Graph 4, on the other hand, is connected because all vertices can reach each other via some path.

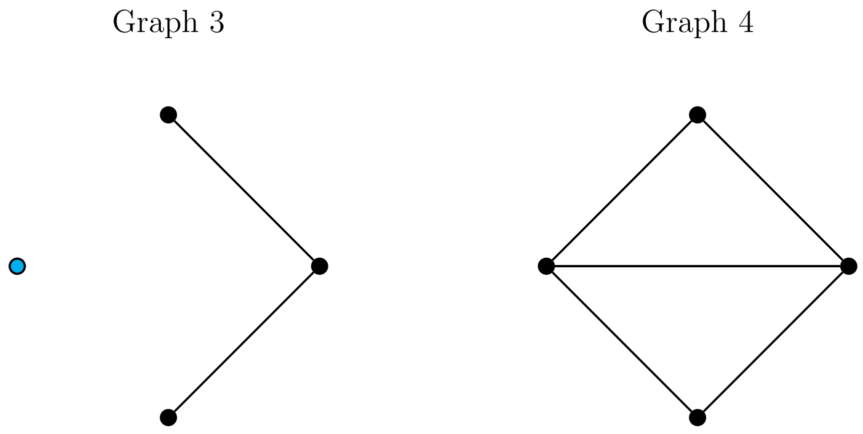

Figure 3: Two graphs containing 4 vertices each.

The graph in Figure 4 is 1-edge-connected because it takes only the removal of one edge, denoted by the blue edge, that will cause the graph to be disconnected. Edges of this nature are also called *bridges*. The graph in Figure 5 is 2-edge-connected because it takes the removal of two edges, denoted by the blue edges, that will cause the graph to be disconnected.

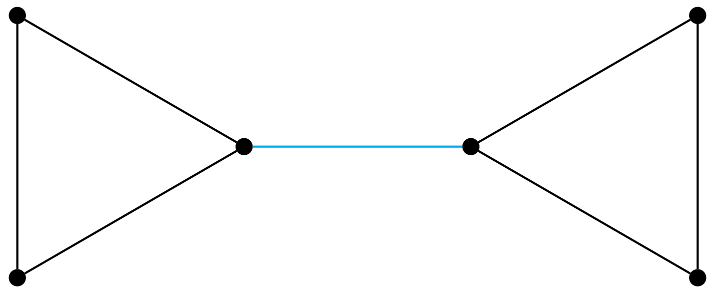

Figure 4: A graph containing 6 vertices and 7 edges.

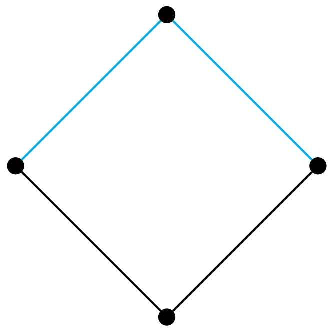

Figure 5: A graph containing 4 vertices and 4 edges.

2 Development of the Conjecture

One may question how many edge-disjoint spanning trees can be found within any given graph. The following theorem gives some insight into this problem by introducing this relationship with edge connectivity:

Theorem 1 (Menger's Theorem [2])**.** The edge connectivity of a graph is equal to the minimum number of edge-disjoint paths between two vertices.

Recall that a similar definition of a spanning tree packing number, given by Definition 1, was discussed earlier. Like Menger's theorem, a spanning tree packing number is a stronger way of finding the number of edge-disjoint paths within a graph. Given the similarities, one may wonder about the relationship between edge connectivity and spanning tree packing numbers. The edge connectivity of the graph can be thought of as the upper bound of its spanning tree packing number. This is due to the fact that a k-edge-connected graph does not guarantee that there will be exactly *k* edge-disjoint spanning trees. Figure 5 demonstrates this weakness, where the graph is 2-edge connected, but there can only exist one edge-disjoint spanning tree.

Menger's theorem provides an upper bound to a graph's spanning tree packing number, but what about the lower bound? The following theorem acts as a sufficient condition for a lower bound:

Theorem 2 (Kundu's Theorem [4])**.** Every $2k$-edge-connected graph has k edge-disjoint spanning trees.

Although it acts as a sufficient condition, it is not always needed to find the number of edge-disjoint spanning trees. However, Palmer made a conjecture that describes when Kundu's theorem is absolutely necessary.

The graph in Figure 6 does not contain two spanning trees within it, because one spanning tree would use 2 of the 3 highlighted edges, making it not possible for a second spanning tree to occur.

Conjecture 1 (Palmer's Conjecture [6])**.** The special 4-regular graph with 15 vertices in Figure 6 is the smallest possible 4-regular graph to have a spanning tree packing number of 1. Any 4-regular graph smaller than that will have a spanning tree packing number of 2.

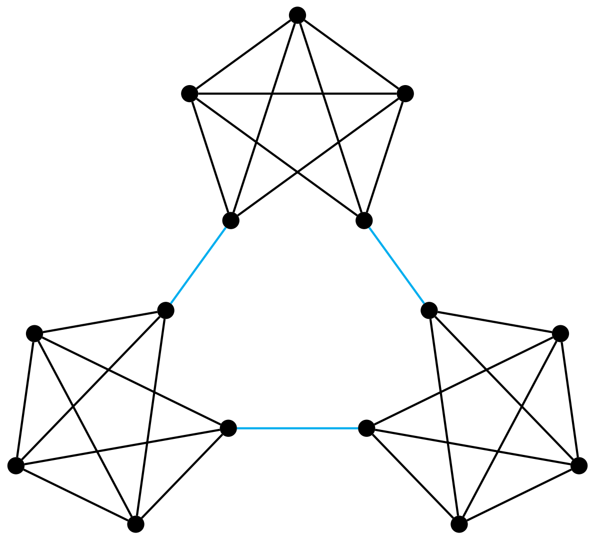

Figure 6: The special graph as described by Palmer's Conjecture.

Palmer's conjecture was verified using a specialization of graph theory, called spectral graph theory. Within this field, the relationship between linear algebra and the matrices associated with graphs is used to explore the properties within that graph. In particular, Cioaba and Wang [1] used eigenvalues to confirm Conjecture 1. Direct verification of the conjecture was done via programming in Java, which can be found here: `https://github.com/bearagones/edge-connectivity`.

3 Breadth-First Search Algorithm

To verify Palmer's conjecture, two algorithms, the *Ford-Fulkerson algorithm* and the *Roskind-Tarjan algorithm*, are used, both of which involve traversing through some graph and finding some path of edges between vertices. When searching for these paths, a breadth-first search algorithm (see Erickson [3]) is used, which is shown in the following pseudocode:

Algorithm 1 Breadth-First Search

```
function BFS(Vertex $s$)
    put $(\emptyset, s)$ in a queue
    while queue is not empty do
        take $(p, u)$ from the queue
        if $u$ is unmarked then
            mark $u$ as visited
            parent($u$) $\leftarrow p$
            for each edge $(u, v)$ do
                add $(u, v)$ to the queue
```

A starting vertex is marked as visited and added to a queue. When vertices are removed from the queue, they are marked as visited if it has not already been done, and they are set as the parent vertex. From there, the neighboring vertices are added to the queue and the algorithm repeats itself.

Figure 7 is a visualization of the algorithm, where vertex 1 is the starting parent vertex. Once vertex 1 is added to the queue and marked as visited, its neighboring vertices, 2 and 3, are then added to the queue. Vertex 2 first gets marked as visited and then becomes the next parent vertex in the queue. Its neighboring vertices are 4 and 5, which are added to the queue. Next in the queue is vertex 3 which only has one neighboring vertex, 6, which is added to the queue. Repeating the algorithm, the next vertices that are taken out of the queue in order are vertices 4, 5, and 6. Because there are no more unvisited vertices, the algorithm terminates.

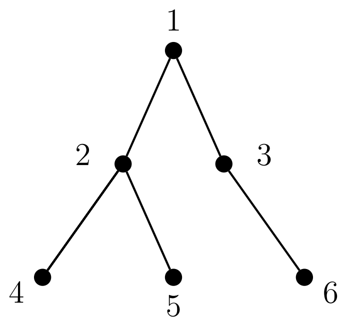

Figure 7: A graph containing 6 vertices and 5 edges.

4 Ford-Fulkerson Algorithm

The Ford-Fulkerson algorithm calculates the maximum flow within a network. In a network flow, two vertices are designated as the source and the sink vertices, and each arc has some capacity to receive a certain amount of flow. The outgoing flow coming from the source node has to travel along arcs in such a way that flow is conserved at each non-sink and non-source vertex, and the flow along each arc is between 0 and the capacity of the arc. For our purposes, the Ford-Fulkerson algorithm is used to calculate the maximum number of paths that can be found between the source node and the sink node within a directed graph, so each arc will have a capacity of 1. In the following pseudocode, the implementation of the directed graph and residual graph will not be shown, but an explanation of how it is constructed will be discussed later.

Algorithm 2 Ford-Fulkerson

```
function MinimumNumberOfPaths(Graph $G$)
    $minimumPaths \leftarrow +\infty$
    create a directed graph $D$ from $G$
    for each vertex $v$ in $G$ do
        $numPaths \leftarrow$ GetNumberOfPaths$(D, 1, v)$
        $minimumPaths \leftarrow \min(minimumPaths, numPaths)$
    return $minimumPaths$

function GetNumberOfPaths(Graph $D$, Vertex $u$, Vertex $v$)
    $pathCounter \leftarrow 0$
    while true do
        create a residual graph $R$ from $D$
        $path \leftarrow$ a path between $u$ and $v$ in $R$
        if no such path exists then
            break
        AugmentFlow$(path)$
        $pathCounter \leftarrow pathCounter + 1$
    return $pathCounter$

function AugmentFlow(ArrayList $path$)
    for each edge $e$ in $path$ do
        if $e$ is a used edge with a flow of 1 then
            set $e$ as an unused edge with a flow of 0
        else set $e$ as a used edge with a flow of 1
```

To create a directed graph from the undirected graph, each undirected edge (u, v) is replaced with two directed edges pointing in opposite directions, (u, v) and (v, u). Additional vertices are added so that between two sets of vertices, there is at most only one directed edge. During the Ford-Fulkerson algorithm, an arc can have a label of 0 or 1. From the labeled directed graph, a residual graph is then created. The *residual graph* is an unlabeled directed graph on the same set of vertices, where an arc has its original orientation if its label is 0, and the reverse orientation if its label is 1. In total, the residual graph will always have $V+2E$ vertices and $4E$ edges, where V and E are the total numbers of vertices and edges from the original graph. Figure 8 exemplifies this:

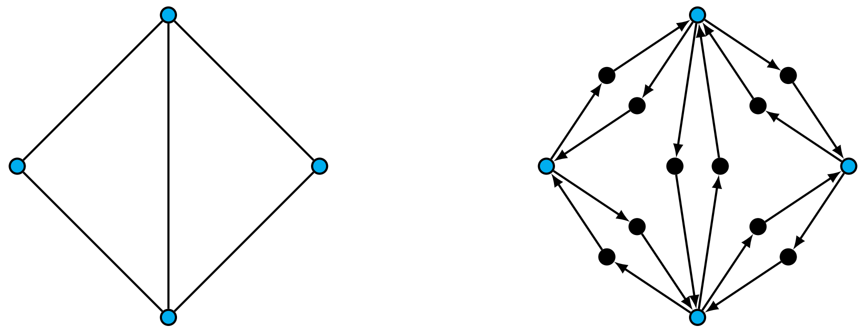

Figure 8: An undirected graph and its directed, residual graph.

After the creation of the residual graph, the algorithm tries to find a path between two vertices u and v. Each edge in the original graph has either a flow of 1 or 0 and can be viewed as an edge being either used or unused, respectively. Once a path is found, the flow is augmented such that the directed edges are flipped around to face the opposite direction, and the flow of each edge in that path is flipped between 0 and 1. The algorithm then repeats, finding alternative paths from u to v until it is no longer able to find one.

Following Figure 8, the Ford-Fulkerson algorithm can find 2 separate paths, denoted by the colored edges on the left of Figure 9, between vertices 1 and 3. After augmenting the flows of those two paths, it is no longer possible to find a path from vertex 1 to vertex 3.

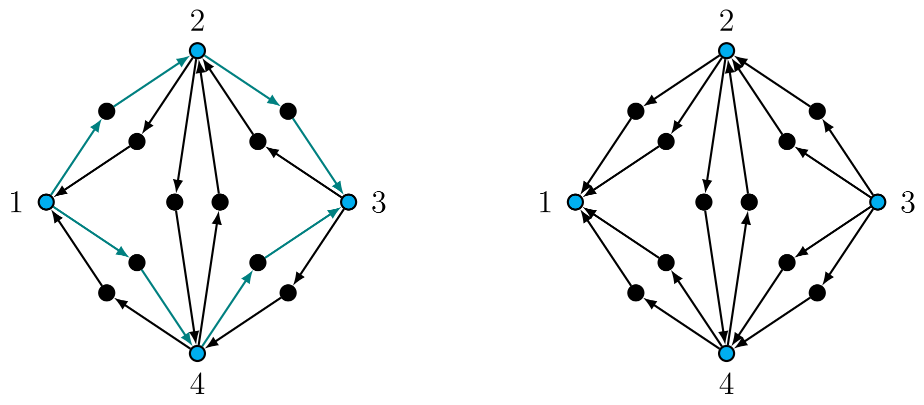

Figure 9: A residual graph with 2 feasible paths from 1 to 3 and the resulting graph after augmenting the flows.

After each iteration of the algorithm, the flow is increased by 1, and by its termination, the final value is the maximum flow. As such, the total maximum flow is equal to the total number of edge-disjoint paths. By Menger's theorem, this is equal to the graph's edge connectivity.

5 Roskind-Tarjan Algorithm

The Roskind-Tarjan algorithm [7] finds a set of k edge-disjoint spanning trees in an undirected graph that has the minimum total cost. For our purposes, the algorithm is instead used to find if it is possible to find exactly two edge-disjoint spanning trees in a graph. The Roskind-Tarjan algorithm consists of processing edges from the graph by trying to add them to one of two forests, where a *forest* is an acyclic, but not necessarily connected, graph. In other words, a forest is a collection of trees. If a cycle is created by adding an edge to a forest, the other edges within that cycle are called the *fundamental cycle* of that edge. In this case, the Roskind-Tarjan algorithm tries to find a sequence of edge swaps so that the edge can be added without the creation of a cycle.

Algorithm 3 Roskind-Tarjan

```
function HasTwoSpanningTrees(Graph $G$)
    create two empty forests $F_1$ and $F_2$ on the same set of vertices as $G$
    for each edge $e$ in G do
        $D \leftarrow$ CreateDirectedGraph($e$, $F_1$, $F_2$)
        $shortestPath \leftarrow$ FindShortestPath($e$, $D$)
        if $shortestPath$ is not null then
            for each vertex $v$ in $shortestPath$ do
                Exchange($v$, $F_1$, $F_2$)
    return $F_1$ is a tree AND $F_2$ is a tree

function CreateDirectedGraph(Edge $e$, Forest $F_1$, Forest $F_2$)
    $E_{forests} \leftarrow$ total number of edges from $F_1$ and $F_2$
    $D \leftarrow$ an empty graph with $E_{forests} + 1$ vertices ▷ vertices represent edges in the forests and $e$
    for each vertex in $D$ do
        if $u$ is part of the fundamental cycle of $v$ when added to the opposite forest then
            insert a directed edge $(u, v)$
    return $D$

function FindShortestPath(Edge $e$, Graph $D$)
    $V_{sink} \leftarrow$ vertices that only have incoming edges in $D$
    return shortest path from $e$ to $V_{sink}$

function Exchange(Vertex $u$, Forest $F_1$, Forest $F_2$)
    if $u$ is in $F_1$ then
        move $u$ to $F_2$
    else move $u$ to $F_1$
```

Figure 10 contains a forest with 5 vertices and a set of edges that makes up a tree: $(1,3)$, $(1,4)$, $(2,3)$, and $(2,5)$. When trying to insert the green edge, $(2,5)$, into the forest, a cycle is created. The resulting fundamental cycle is the set of blue edges.

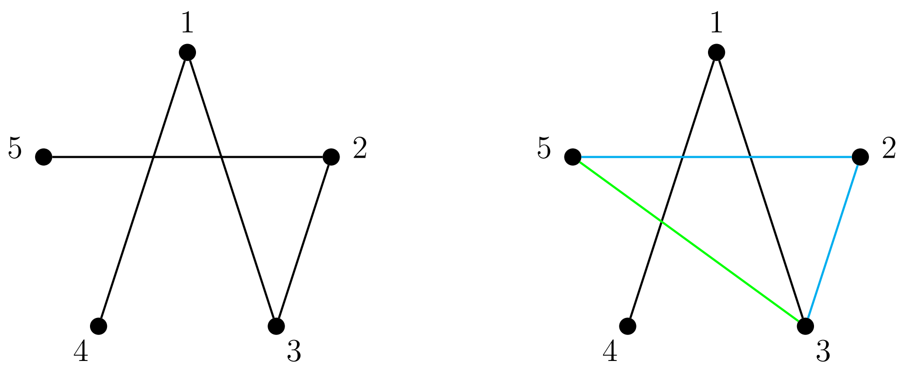

Figure 10: The fundamental cycle of $(3,5)$.

From the pseudocode, a directed graph is created where vertices represent the edges within the two forests as well as the edge e that is trying to be added, and directed edges represent if the destination vertex is part of the fundamental cycle of the origin vertex when added to the opposite forest (or F_1, in the case of the edge e). After the conversion of the forests to the directed graph, sink vertices are identified as the vertices that do not have any outgoing edges. Once these vertices are identified, the shortest path between the vertex representing the edge to be added and the sink vertex is found. Figure 11, using the forest from Figure 10, demonstrates this procedure.

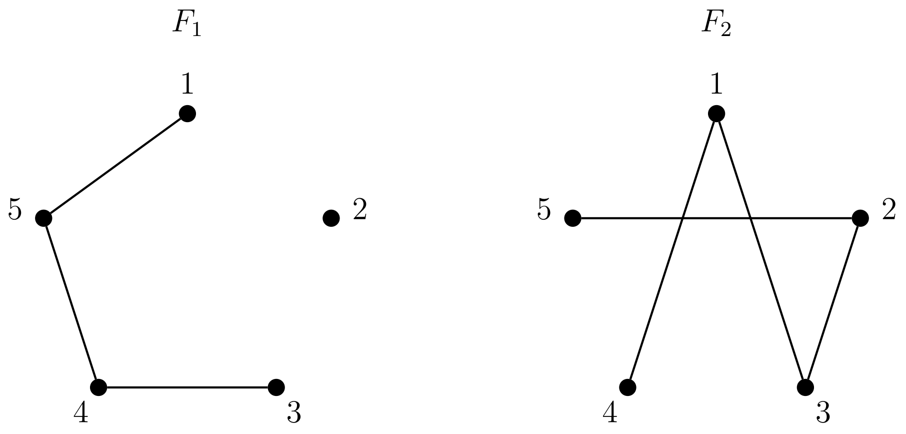

Figure 11: Two forests with 5 vertices each.

From Figure 11, assume that the edge that is trying to be added is the edge $(3,5)$. Notice that when trying to add the edge in either forest, a cycle is created, and thus a swap sequence must occur. In the directed graph, the vertices of F_1 are duplicated on the right side for clarity, and the graph should be read from left to right.

Within Figure 12, the edges from F_1 are denoted as the black vertices, and the edges from F_2 are denoted as the white and green vertices. The edge that we are trying to add to the forest is the blue vertex. An example of adding directed edges is as follows: the edge $(1,5)$ is currently located in F_1. If we were to add the edge to F_2, it can be seen that a cycle is created within that forest, with the edges $(1,3)$, $(2,3)$, and $(2,5)$ within that cycle. As such, there are three directed edges pointing towards the three vertices representing those edges, with the vertex $(1,5)$ as the origin. This is repeated for every edge in both forests.

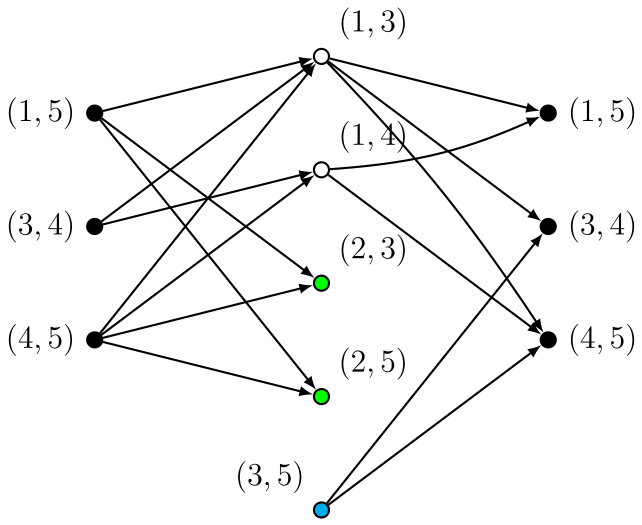

Figure 12: The directed graph of Figure 11.

After the directed graph has been made, note that the vertices $(2,3)$ and $(2,5)$ are the only two vertices that do not have any outgoing edges. These are the sink vertices. Starting from the vertex $(3,5)$, the shortest path that can be found between that and the sink vertices is the path $(3,5) \rightarrow (4,5) \rightarrow (2,5)$. From this path, it is implied that the two edges can exchange forests (i.e. $(4,5)$ is moved to F_2 and $(2,5)$ is moved to F_1), and the edge $(3,5)$ can be added to F_1. This is indeed a valid augmenting swap sequence, as shown by the updated forests in Figure 13.

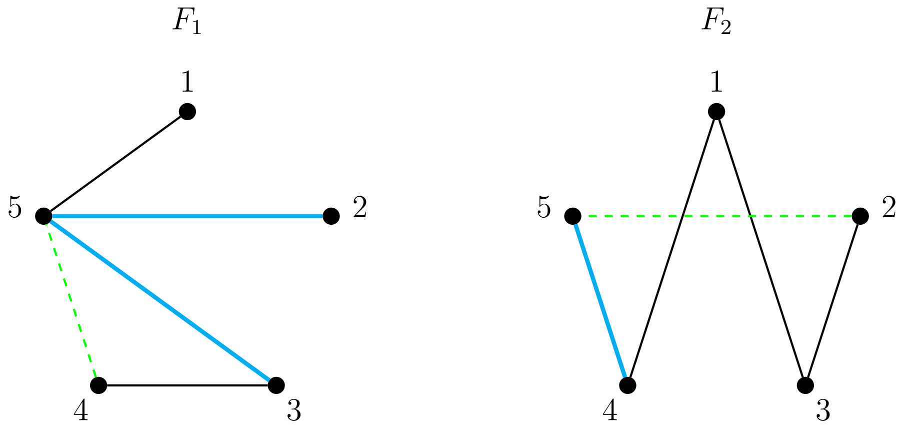

Figure 13: The updated forests with the added edge $(3,5)$

6 Verification of the Conjecture

To verify Palmer's conjecture, a combination of the Ford-Fulkerson algorithm and the Roskind-Tarjan algorithm was implemented to test 4-regular graphs on the basis of Kundu's theorem. The following algorithm was tested on approximately 100,000 4-regular graphs with 14 or fewer vertices in a single .txt file that was generated by the `nauty` library [5].

Algorithm 4 Palmer's Verification

```
function VERIFICATION(graph $G$)
    if minimumNumberOfPaths(graph $G$) is not equal to 4 then
        if hasTwoSpanningTrees(graph $G$) is false then
            return false
    return true
```

The pseudocode first uses the Ford-Fulkerson algorithm to verify that the graph is 4-edge-connected. If it is, then by Kundu's theorem, it is guaranteed that there will be two edge-disjoint spanning trees. On the other hand, if it is not 4-edge-connected, then it is still possible to find two edge-disjoint spanning trees. If this is the case, then the Roskind-Tarjan algorithm is used next to explicitly verify that there are two edge-disjoint spanning trees that can be found. If the two trees are not present, then the verification fails.

The following figures are examples of graphs generated by the program that passes the verification. In Figure 14, the graph is 4-edge-connected because it will always take the removal of 4 edges at minimum to disconnect the graph.

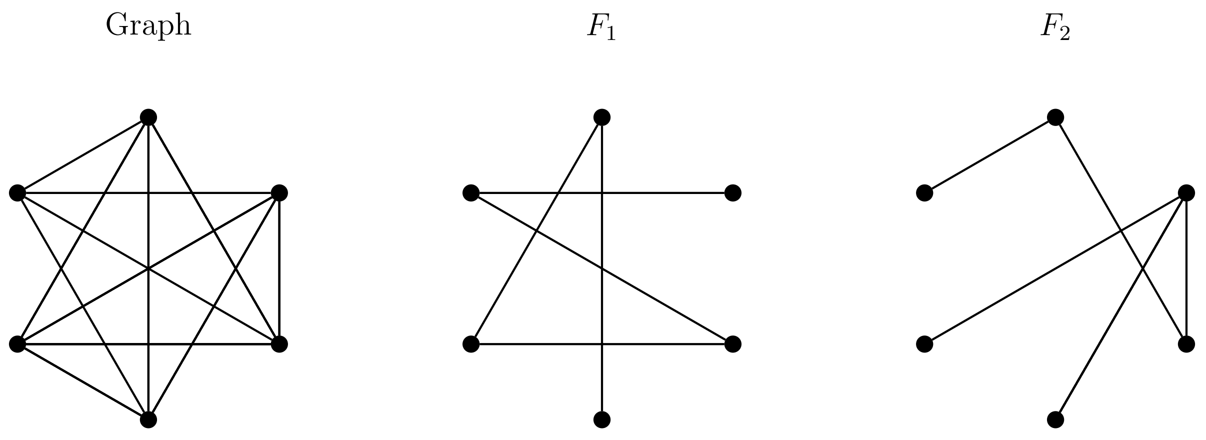

Figure 14: A graph with 6 vertices that is 4-edge-connected and has two edge-disjoint spanning trees.

In Figure 15, one may initially think that the following graph is 4-edge-connected. However, when applying the Ford-Fulkerson algorithm, it is found that there can only be 2 edge-disjoint paths between vertex 1 and vertex 2 before they become unreachable to each other, which are denoted by the blue and green edges. This implies that removing one blue edge and one green edge disconnects the graph, and thus the graph is 2-edge-connected. However, by the Roskind-Tarjan algorithm, two edge-disjoint spanning trees can still be found and the verification still holds.

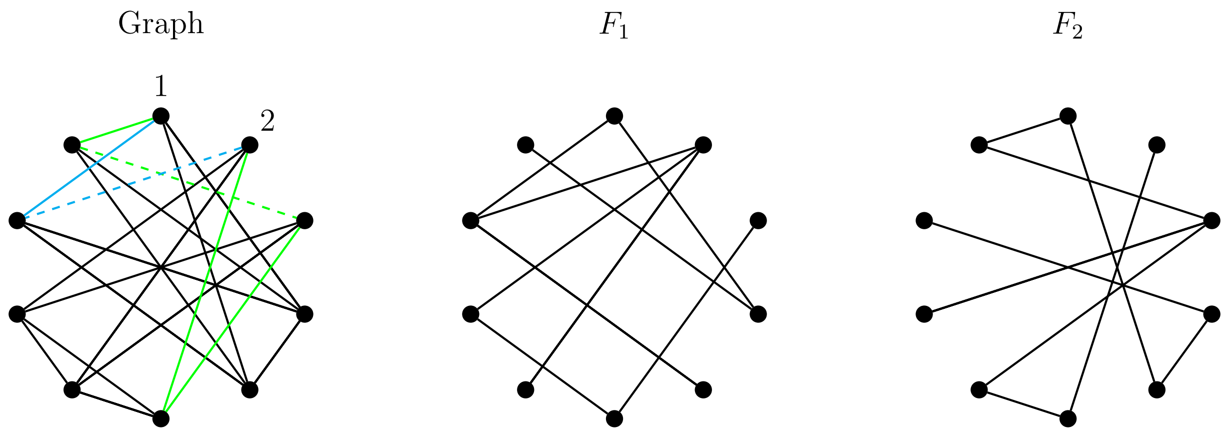

Figure 15: A graph with 10 vertices that is not 4-edge-connected but still has two edge-disjoint spanning trees.

When verifying the special graph described by Palmer in Figure 6, it is found that the graph is not 4-edge-connected. This is due to the fact that removing any two of the blue edges in Figure 6 disconnects the graph. Only one edge-disjoint spanning tree can be found because any spanning tree would use at least two of the blue edges as shown in Figure 16.

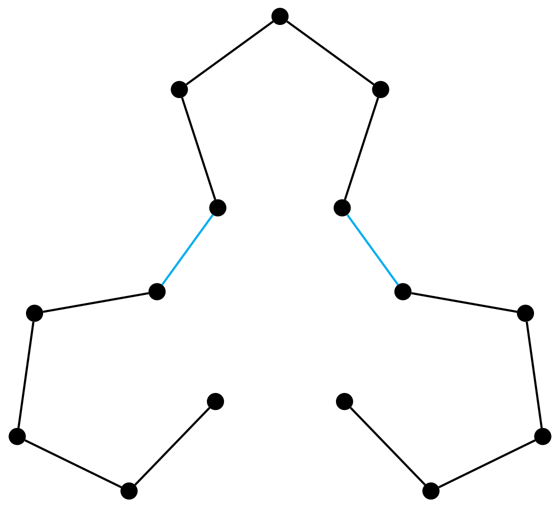

Figure 16: The special graph with 15 vertices is 2-edge-connected but only has 1 edge-disjoint spanning tree.

7 Conclusion

Kundu's Theorem states that every $2k$-edge-connected graph has k edge-disjoint spanning trees. For our purposes, testing was done on the cases where $k = 2$. Following this theorem, Palmer made a conjecture that a special 4-regular graph containing 15 vertices is the smallest possible graph that only contains one edge-disjoint spanning tree, and any 4-regular graph smaller than that will have two trees. To test the validity of the conjecture, a combination of the Ford-Fulkerson algorithm and the Roskind-Tarjan algorithm was implemented to test thousands of graphs, in addition to the aforementioned special graph on the basis of Kundu's theorem. The validation consisted of first checking if the graph was 4-edge-connected using the Ford-Fulkerson algorithm. If it was, then it was guaranteed to contain two edge-disjoint spanning trees. If, however, it was not 4-edge-connected, the algorithm then checks if there are two trees using the Roskind-Tarjan algorithm. When testing 4-regular graphs with 14 vertices or less, all graphs were verified to have 2 edge-disjoint spanning trees. When testing the special graph, on the other hand, the verification failed because it only contained one spanning tree, which corroborates Palmer's conjecture.

Despite using the Roskind-Tarjan algorithm as part of the verification process of the conjecture, a full implementation was unable to be done, which involves graph partitioning and clustering of vertices. Rather, a less-efficient implementation was done, causing the algorithm to be more computationally expensive. The code can be optimized further so that the algorithm takes less time to verify the thousands of graphs. Further investigation can be done to test varying cases of Kundu's theorem, and verify what the smallest possible graphs are, similar to what Palmer conjectured.

References

[1] Sebastian M. Cioabă and Wiseley Wong. "Edge-disjoint spanning trees and eigenvalues of regular graphs". In: *Linear Algebra and its Applications* 437.2 (2012), pp. 630–647. ISSN: 0024-3795. DOI: `https://doi.org/10.1016/j.laa.2012.03.013`. URL: `https://www.sciencedirect.com/science/article/pii/S0024379512002133`.

[2] Reinhard Diestel. *Graph theory*. Fifth. Vol. 173. Graduate Texts in Mathematics. Paperback edition of [MR3644391]. Springer, Berlin, 2018, pp. xviii+428. ISBN: 978-3-662-57560-4.

[3] Jeff Erickson. *Algorithms*. eng. 1st paperback edition. Erscheinungsort nicht ermittelbar: Selbstverlag, 13. ISBN: 9781792644832.

[4] Sukhamay Kundu. "Bounds on the number of disjoint spanning trees". In: *Journal of Combinatorial Theory, Series B* 17.2 (1974), pp. 199–203. ISSN: 0095-8956. DOI: `https://doi.org/10.1016/0095-8956(74)90087-2`. URL: `https://www.sciencedirect.com/science/article/pii/0095895674900872`.

[5] Brendan D McKay and Adolfo Piperno. "nauty and Traces user's guide (version 2.6)". In: *Computer Science Department, Australian National University, Canberra, Australia* (2016).

[6] E.M. Palmer. “On the spanning tree packing number of a graph: a survey”. In: *Discrete Mathematics* 230.1 (2001). Catlin, pp. 13–21. ISSN: 0012-365X. DOI: https://doi.org/10.1016/S0012-365X(00)00066-2. URL: https://www.sciencedirect.com/science/article/pii/S0012365X00000662.

[7] James Roskind and Robert E. Tarjan. “A Note on Finding Minimum-Cost Edge-Disjoint Spanning Trees”. In: *Mathematics of Operations Research* 10.4 (1985), pp. 701–708. ISSN: 0364765X, 15265471. URL: http://www.jstor.org/stable/3689437 (visited on 12/08/2022).