View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by DigitalCommons@CalPoly

Bipartite Graph Packing Problems

Collin Wong

June 9, 2018

1 Introduction

The overarching problem of this project was trying to find the maximal number of disjoint subgraphs of a certain type we can pack into any graph. These disjoint graphs could be of any type in the original problem. However, they were limited to be T_2 trees for my research (T_2 trees are defined in section 2.1 of the paper). In addition, most of my work was focused on packing these T_2 trees into constrained bipartite graphs (also defined in section 2.1 of the paper).

Even with these specific constraints applied to the overall problem, the project still branched into different subproblems such as packing trees into complete bipartite graphs and finding minimal and maximal bounds for packing these graphs.

2 The Initial Problem

2.1 Packing T_2 Trees into Undirected Bipartite Graphs

Suppose we have an undirected, bipartite graph G, where the vertices of G are partitioned into vertex sets S and T, and $|S| = |T| + 1$. Assume that G is a finite graph with no loops or multiple edges. We will define a T_2 tree to be a spanning tree in which every vertex in T has a degree of exactly 2. What is the maximum number of disjoint T_2 trees we can form on graph G?

2.2 Initial Notes on the Problem

From this problem, it is important to highlight important facts about these T_2 trees. Because the trees are disjoint, none of the T_2 trees drawn on a graph G can have any similar edges (in other words, each edge in G can only be used in one T_2 tree). In addition, each tree must be spanning, so every vertex in S and T must be touched by every tree. Lastly, these must be trees, so no cycles can be formed in any one T_2 tree.

2.3 Drawing Trees

Initially, it was easiest to draw complete bipartite graphs as it allowed more T_2 trees to fit within the graph. Therefore, the upper three drawings in Figure 1 show the maximum number of trees on the first three complete bipartite graphs: $K_{2,1}, K_{3,2}$, and $K_{4,3}$.

After, I attempted to draw T_2 trees that were not paths on some graphs. An example of this approach is Tree 1 in the bottom right graph of Figure 1. This showed to be much harder as forming cycles was much easier when not trying to find T_2 paths. In addition, these trees usually resulted in failures to create the maximum number of T_2 trees on the graphs.

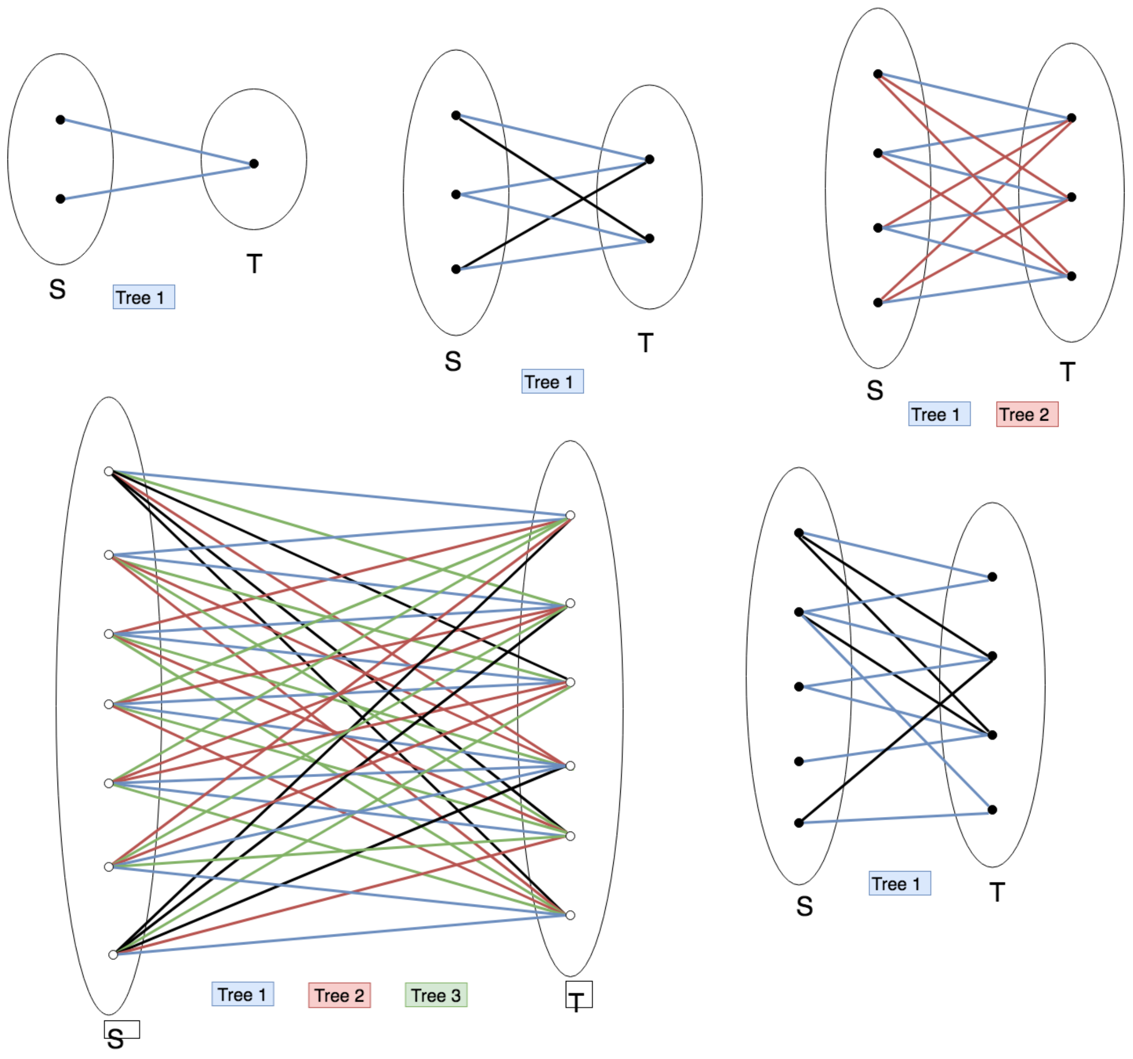

Figure 1: Initial Drawings of Packing T_2 Trees into Bipartite Graphs.

2.4 Observation from Initial Drawings

Drawing T_2 trees on different bipartite graphs gave some initial insights on certain bounds of this problem:

- By definition, Graph G must have at least (2 x |T|) edges in order to have one T_2 tree
- Because the T_2 trees must be spanning, the maximum number of T_2 trees on G ≤ min degree of all vertices in S
- Also by the definition of the T_2 tree, the maximum number of T_2 trees on G ≤ $\lfloor \frac{min\ degree\ of\ all\ vertices\ in\ T}{2} \rfloor$
- From studying the complete graphs, the maximum number of T_2 trees on G ≤ $\lfloor \frac{|S|}{2} \rfloor$

2.5 Theorem I: Minimum Edges to Form a T_2 Tree

From the initial drawings, we can see that Graph G must have at at least (2 x |T|) edges in order to form one T_2 tree.

Theorem: In an undirected, bipartite graph G, where the vertices of G are partitioned into vertex sets S and T, and |S| = |T| + 1, G must have at at least (2 x |T|) edges in order to form one T_2 tree.

Proof. Suppose we have a Bipartite Graph G = (S ∪ T, E), where |S| = |T| + 1.

By definition, every vertex in T must have a degree of exactly 2 to form one T_2 tree on G.

Therefore, there must be at least (2 x |T|) edges to form one T_2 tree on G. □

2.6 Theorem II: Maximum Edges Without Forming a $\mathbf{T_2}$ Tree

Another interesting case was to see how many edges can be added to graph G until the addition of one more would surely create a T_2 tree. For graph G, the max number of edges resulted from connecting every vertex minus one in S to every vertex in T.

Theorem: In an undirected, bipartite graph G, where the vertices of G are partitioned into vertex sets S and T, and |S| = |T| + 1, the maximum number of edges in G without forming a T_2 tree is $|T|^2$ edges.

Proof. Suppose we want to find the maximum number of edges without forming a T_2 tree on a graph where n = |S| and m = |T|.

We can create a complete bipartite graph $K_{m,m}$ = (S ∪ T, E) and add a vertex into S, s_d, so that each vertex in T is connected to each vertex in S$\backslash s_d$. We will call this graph G.

In graph G, there is no possible way to form a T_2 tree because T_2 trees must be spanning and there are no edges incident to s_d.

In addition, we cannot add any more edges to S$\backslash s_d$, because it was created by the complete graph $K_{m,m}$.

Therefore, the only way to add an edge to G is with an edge from s_d to a vertex in T.

Claim: Adding one more edge to the G will always create a T_2 tree.

Proof:

Label the vertices in S: $s_1, s_2, s_3, \ldots, s_n$

Label the vertices in T: $t_1, t_2, t_3, \ldots, t_{n-1}$

We will say there are edges from every vertex in T to every vertex in S except s_n (In this proof of the claim, s_n refers to the vertex s_d in the encasing proof). Therefore, we can only add an edge from s_n to any vertex in T.

If we add an edge from s_n to t_x, where x is any integer from 1 to (n-1), we can use this formula to find a T_2 tree: $s_n t_x s_1 t_{x+1} s_2 \ldots s_{n-2} tx - 1 s_{n-1}$

Therefore, there will always be a T_2 tree in the graph after adding one more edge to G. ■

Therefore, there can be no more edges added to G without forming a T_2 tree on G.

Therefore, the maximum number of edges that we can have in a graph without forming a T_2 tree is $|T|^2$. □

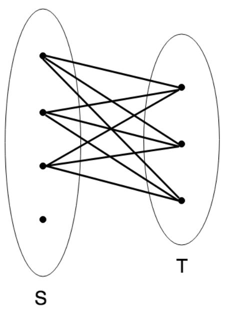

Figure 2: Graph with the max number of edges without forming a T_2 Tree.

3 The Complete Bipartite Graph: $\mathbf{K_{n,m}}$

3.1 Investigation of $\mathbf{K_{n,m}}$

The Complete Bipartite Graph $K_{n,m}$ is the graph G = (S ∪ T, E), where n = |S| and m = |T|, n = m + 1, and E contains an edge from every vertex in S to every vertex in T. The complete bipartite

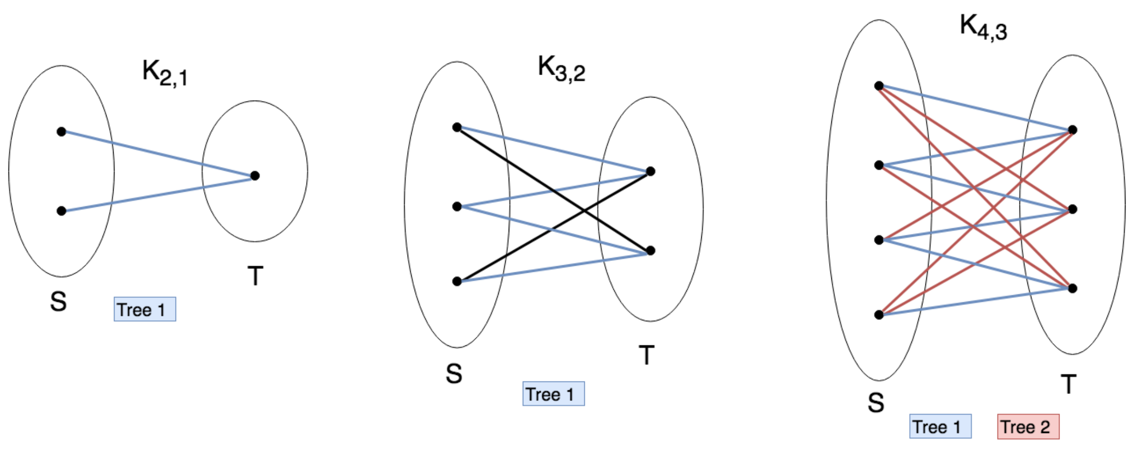

Figure 3: T_2 trees on $K_{2,1}$, $K_{3,2}$, and $K_{4,3}$.

graph $K_{n,m}$ is an important special case because the graph has every possible edge connecting its vertices, while keeping the bipartite and no multiple edges or loops conditions. Investigation of $K_{n,m}$, showed that it was easiest to form T_2 trees on the complete bipartite graph with paths. However, it is important to note that is may not be the case for every possible bipartite graph.

In addition, the complete graph $K_{n,m}$ was useful to research because it helped form an upper bound on the maximum number of T_2 trees on all bipartite graphs, where n = m + 1. In Figure 3 , I have drawn the maximum number of T_2 trees on $K_{2,1}$, $K_{3,2}$, and $K_{4,3}$. However, after continuing to draw $K_{5,4}$, $K_{6,5}$, and $K_{7,6}$, it became obvious that each graph had $\lfloor \frac{n}{2} \rfloor$ disjoint T_2 trees on it. Therefore, I conjectured that this special graph $K_{n,m}$ must have $\lfloor \frac{n}{2} \rfloor$ disjoint T_2 paths on the graph (paths are trees by definition).

3.2 Theorem III: Packing Maximal $\mathbf{T_2}$ trees on $\mathbf{K_{n,m}}$

Theorem: A complete bipartite graph G = (S ∪ T, E), where |S| = |T| + 1, must have $\lfloor \frac{|S|}{2} \rfloor$ disjoint T_2 paths on the graph.
Input: A complete bipartite graph G = (S ∪ T, E), where |S| = n, |T| = n - 1, and n > 1. Goal: $\lfloor \frac{n}{2} \rfloor$ disjoint T_2 paths

Proof.
Label the vertices in S: $s_1, s_2, s_3, \ldots, s_n$
Label the vertices in T: $t_1, t_2, t_3, \ldots, t_{n-1}$
Paths:
First : $s_1 t_1 s_2 t_2 s_3 t_3 \ldots s_{n-1} t_{n-1} s_n$
Second : $s_3 t_1 s_4 t_2 s_5 t_3 \ldots s_n t_{n-2} s_1 t_{n-1} s_2$
...
General: $s_k t_1 s_{k+1} t_2 s_{k+2} t_3 \ldots s_{k-2} t_{n-1} s_{k-1}$

Case 1: n is even
Last : $s_{n-1} t_1 s_n t_2 s_1 t_3 \ldots s_{n-3} t_{n-1} s_{n-2}$
These are $\frac{n}{2}$ disjoint T_2 paths on G. Because all edges are used, there cannot be more T_2 trees/paths.

Case 2: n is odd
Last : $s_{n-2} t_1 s_{n-1} t_2 s_n t_3 \ldots s_{n-4} t_{n-1} s_{n-3}$
These are $\frac{n-1}{2}$ or $\lfloor \frac{n}{2} \rfloor$ disjoint T_2 paths on G.
The graph G initially has |S| x |T|, or n(n - 1), edges.
We know each disjoint T_2 tree uses 2|T|, or 2(n - 1), edges of G.
We have created $\frac{n-1}{2}$ disjoint T_2 paths on G, so we have used:

$$[\frac{n-1}{2}\ \ trees] * [2(n-1)\ \ \frac{edges}{tree}] = (n-1)^2\ \ edges \tag{1}$$

We can use this to determine how many edges remaining in graph G.

$$n(n-1) - (n-1)^2 = (n-1)\ \ edges\ \ remaining \tag{2}$$

We are left with (n - 1), or |T|, edges left in G, and we know we need at least 2(n - 1), or 2|T|, edges to form another T_2 path/tree.

For all n $>$ 1, 2(n - 1) $>$ (n - 1).

Therefore, there cannot be more T_2 trees/paths.

□

3.3 Failures on packing the Maximal Number of $\mathbf{T_2}$ trees on $\mathbf{K_{n,m}}$

While attempting to draw T_2 trees on $K_{n,m}$, there were many failures, where failures are defined as drawing less than $\lfloor \frac{|S|}{2} \rfloor$ disjoint T_2 trees on the graph. These failures resulted from starting on a non-minimal degree vertex of S when drawing the T_2 paths, avoiding paths altogether, and creating cycles inside the T_2 trees.

If S contains an odd number of vertices, then we can still create $\lfloor \frac{|S|}{2} \rfloor$ disjoint T_2 paths on the graph while starting on non-minimal edges of the graph. However, when S has an even number of vertices, each T_2 path must start on minimal degree vertex of S.

In addition, attempting to use non-paths in the $K_{n,m}$ graph resulted in failures. For example, creating a T_2 tree that contains a star tree (with one vertex in S as the center vertex and all other vertices in T) will result in a failure (proved in Conjecture II). This example is shown in Figure 5.

The most common error was creating a cycle while trying to create a T_2 tree when not using the Construction in Conjecture I for finding T_2 trees on $K_{n,m}$. It could also be possible to use Depth First Search in order to find a path to fix this problem.

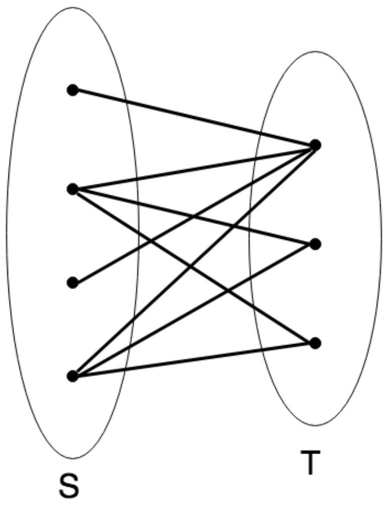

Figure 4: Example of a graph where you cannot create a T_2 tree.

3.4 Theorem IV: Disjoint Star Trees

Theorem: A graph can never have two disjoint spanning trees, one of which is a star.

Proof. Suppose we have a graph G = (V, E) that has no multiple edges.

Suppose graph G contains a star tree, S, that has a center vertex v_c and edges E_S.

Since v_c is adjacent to every other vertex in G, G$\backslash E_S$ leaves v_c as an isolated vertex. Therefore, there cannot be another spanning tree in G$\backslash E_S$ as v_c is an isolated vertex. Therefore, a graph with a star spanning tree cannot have another disjoint spanning tree.

□

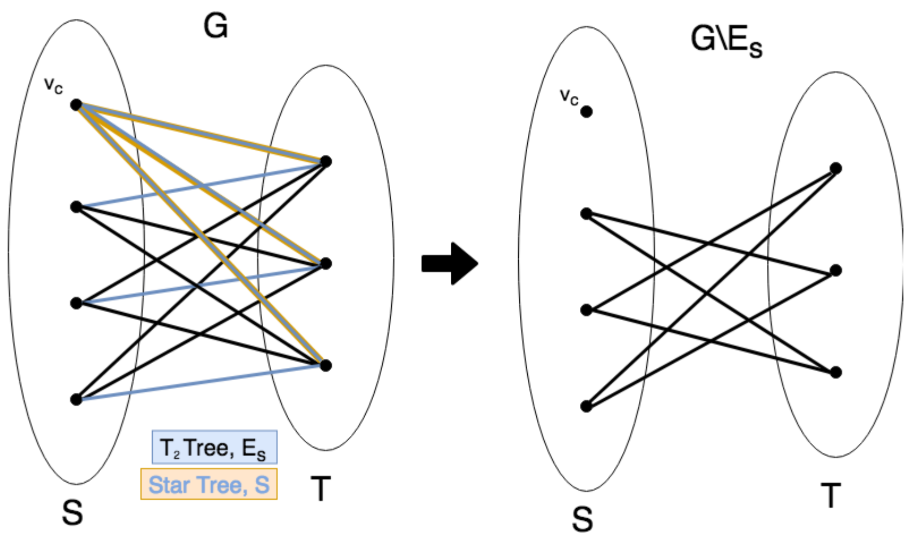

Figure 5: An graphical representation of Conjecture II.

4 Research on Related Problems

4.1 Lovász on Kneser's Conjecture

In the paper *Kneser's Conjecture, Chromatic Number, and Homotopy*, Lovász attempts to generalize Kneser's Conjecture. "If the simplicial complex formed by the neighborhoods of points of a graph is (k-2)-connected, then the graph is not k-colorable" [1]. This conjecture was used to prove this theorem: "If we split the n-subsets of a (2n+k) element subset, one of the classes will contain two disjoint n-subsets" [1]. In these definitions, it is still unclear what a simplicial complex is, but an n-subset is a subset of exactly n distinct elements of the set.

As an example of this, suppose n = 2 and k = 1, and the set is 1, 2, 3, 4, 5. Therefore, the 2-subsets are: (1,2), (1,3), (1,4), (1,5), (2,3), (2,4), (2,5), (3,4), (3,5), (4,5). We will separate classes into subsets that do not contain the element 1 and subsets that do contain the element 1.

The two classes are:
Class 1 (that contains 1) : (1,2), (1,3), (1,4), (1,5)
Class 2 (does not contain 1): (2,3), (2,4), (2,5), (3,4), (3,5), (4,5)

As shown, there are actually three pairs of two disjoint n-subsets in Class 2: (2,3)/(4,5), (2,4)/(3,5), and (3,4)/(2,5). Moreover, Lovász includes another theorem that can be deduced from the first theorem: "The chromatic number of Kneser's graph $KG_{n,k}$ is k+1" [Lov77].
In Figure 6, we can see the previous example in graph form. This shows the 2 classes from the earlier example. In addition, it shows the minimum coloring of the graph is 3 (red, yellow, and blue vertices). Therefore, the chromatic number of $KG_{2,1}$ is 3.

Unfortunately, after reading this paper, it was decided that it had no applications to the graph packing problems I was working on.

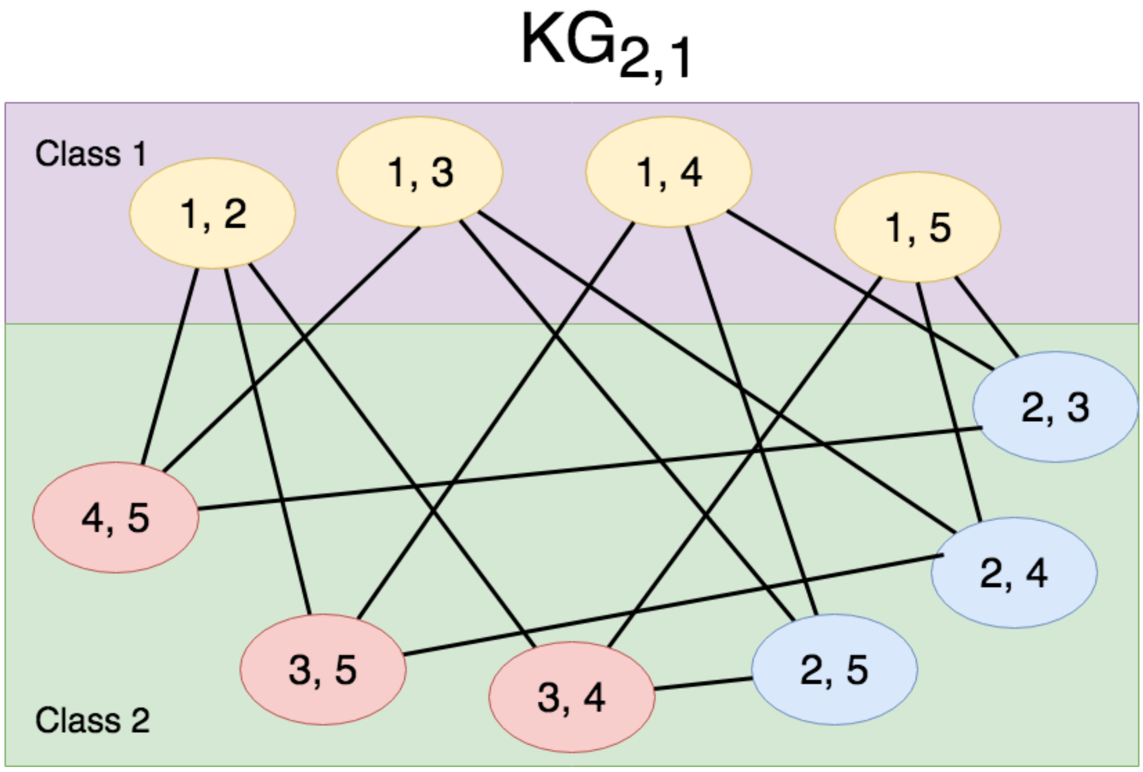

Figure 6: Kneser's graph $KG_{2,1}$ (n = 2, k = 1).

4.2 Lovász on Konig's Theorem

In his other paper, *A Generalization of Konig's Theorem*, Lovász gives a generalization of Konig's Theorem which was originally stated for bipartite graphs. The theorem states "the maximum number of (pairwise) independent edges of a bipartite graph G equals to the minimum number of vertices covering all the edges of G" [Lov70].

Lovász uses many variables and functions without clear explanations in this paper, so it was difficult to understand this paper. Nevertheless, it was also decided that this edge/vertex cover theorem was also not helpful for bipartite graph packing.

4.3 Lovász on Covering Graphs

In the paper *On Covering of Graphs*, Lovász discusses how we can minimally cover graphs given certain constraints. In this paper, he considers undirected graphs without multiple edges. The investigation begins with covering graphs with a minimal number of paths and circuits. He gives a theorem that says "A graph of n vertices can be covered by $\leq \lfloor \frac{n}{2} \rfloor$ disjoint paths and [cycles]" [Lov68]. He also goes on to give another theorem about complete graphs that says "a graph of n vertices can be covered by $\lfloor \frac{n^2}{2} \rfloor$ complete graphs" [Lov68].

Although this paper initially seemed related to the problem, it only talked about minimal coverings instead of maximal coverings which would relate to our problem of packing the maximal amount of subgraphs into a graph.

5 The Chinese Postman Problem

5.1 Definitions

In his paper *Matching, Euler Tours and the Chinese Postman*, Edmonds discusses the interesting problem of the Chinese postman, as well as its subproblems of finding matchings and Euler tours. In order to understand the problem, it is important to define these five terms:

- "A *simple tour* of G is a [cycle] which contains every edge e at least once" [Edm73].

- "A *postman tour* of G is a [cycle] which contains every edge at least once" [Edm73].

- "An *Euler tour* of G is a [cycle] which contains every edge exactly once" [Edm73].

- "The *length* c_e of an edge e, [the weight of the edge], is assumed to be a non-negative number" [Edm73].
- "The *length of a tour* is $\sum_{i=1}^{l} c_{e_i}$, where l is the number of edges in the tour" [Edm73].

5.2 The Problem

The Chinese postman problem is to find the *shortest* postman tour in a undirected, connected graph, where *shortest* does not mean the least number of edges, but rather the shortest length of the tour. While this definition may seem like it is really one problem, it is actually almost always going to be two different problems.

If the graph already contains an Euler tour, then we will just need to find the Euler tour. This is because if the graph contains an Euler tour, then that tour is the shortest postman tour as no edges will have to be repeated in the tour and each edge is included at least once. In order to determine this, there is a theorem that says: *if every vertex in graph G has an even degree, then G for sure has a Euler Tour.* If this is true, we will say the graph is graph G' and move onto *5.4 Finding The Euler Tour.*

However, if the graph does not contain a Euler tour, we will have to add edges to the graph in order to create this Euler Tour, as well as finding the Euler tour.

5.3 Matching Problem

Suppose we have a graph G = (V,E), which does not contain an Euler tour. We will be adding edges to graph G to create graph G'. We will do this by implementing a Matching Algorithm:

1. Add each odd-degree vertex in G to G_p.

2. Find the shortest path between every odd-degree vertex in G (There are O($|V|^3$) time algorithms for this), and add an edge in G_p between every edge. Each edge in G_p will have a weight of the length of the shortest path in G of the two adjacent vertices to the edge.

3. Find an *optimum 1-matching*, M: a set of edges with the least possible total weight where every vertex only touches an edge once.

4. For each edge in M, m, add an edge into G with the same weight as each edge in the shortest path corresponding to the edge m. We will call this new graph G'.

In this algorithm, there must be a 1-matching because of the theorem that says: *in a connected graph, there must be an even number of odd degree vertices.* Because G was connected and added only its odd degree vertices to G_p, G_p must have an even number of vertices. In addition, G_p will always be the complete graph because each vertex has an edge to every other vertex in G_p. Therefore, there must always be a 1-matching in G_p.

5.4 Finding the Euler Tour

Once you have graph G', a graph with an Euler tour, we will need an algorithm to actually find a series of edges to follow to find an Euler tour. Edmonds calls this an End-Pairing Algorithm, Algorithm 1. This algorithm will return the tour in the form of a list of edges of the tour. Keep in mind that this is not a unique Euler Tour, as there can be many Euler tours. However, we now have the shortest Postman tour of graph G'.

5.5 Applications to the Problem

Although this problem does not pack graphs, the matching algorithm relates to our problem of packing because it is packing as many edges in a graph with the constraint that each vertex can only touch one edge. However, this matching problem is different because we are adding in edges into a graph whereas our problem does not allow adding edges into the graph.

Algorithm 1 End-Pairing Algorithm

```
Input: Graph G' = (V', E') that contains an Euler Tour
Output: A list of edges of a Euler Tour of G'
function PAIR-ENDS(G' = (V', E'))
    tour = ∅
    mark all edges in E' as unused
    do
        if tour == ∅ then
            $v_o$ = any vertex in V' connected to an unused edge in E'
        else
            $v_o$ = any vertex in V' connected to an unused edge in E' and connected to an edge in the tour
        end if
        $v = v_o$
        currentTour = ∅
        do
            $e$ = an unused edge in E' incident to v
            mark e as used
            append e to currentTour
            $v$ = the other vertex incident to $e$ that is not the current $v$
        while v is incident to an unused edge in E'
        if tour == ∅ then
            tour = currentTour
        else
            interject currentTour into tour between two consecutive edges in tour that are both incident to $v_0$
        end if
    while There exists an unused edge in E'
    return tour
end function
```

References

[Edm73] Jack Edmonds. Matching, euler tours and the chinese postman. 1973. Waterloo, Canada.

[Lov68] Lubromir Lovász. On covering of graphs. 1968. Budapest, Hungary.

[Lov70] Lubromir Lovász. A generalization of konig's theorm. 1970. Budapest, Hungary.

[Lov77] Lubromir Lovász. Kneser's conjecture, chromatic number, and homotopy. 1977. Szeged, Hungary.