Integer Visibility Bar Representations of Trees

HP Bourke

A MTH 501 – Mathematical Literature Project

Under the direction of Prof. John Caughman

In partial fulfillment of the requirements for the degree of

Masters of Science in Mathematics

Fariborz Maseeh Department of Mathematics and Statistics

Portland State University

December 8th, 2022

Abstract

In this project we consider bar visibility representations of graphs, a topic first introduced to study circuit testing [2],[6]. Specifically, we study the problem of finding a (non-unique) integer bar visibility representation of minimal width for any given tree. To find such an optimal representation, we derive both a lower bound and an upper bound on the width. In cases when these bounds are the same, we are assured of the optimality of our representation. We also study a number of related results and conjectures concerning balanced partitions of the vertices and edges of trees. We conclude with a number of directions for future research.

Contents

1. Introduction

The usage of visibility graphs in mathematics and computer science dates back to 1969, with the work of Nils Nillson [4]. Visibility graphs were originally designed for movement mapping and motion planning of autonomous robotic devices. Bar visibility graphs, which specifically represent graph vertices as horizontal line segments (bars) with vertical lines of sight among them, were introduced several years later to facilitate the study of circuit testing [2].

In this project we consider integer bar visibility graphs, in which every bar must have endpoints with integer coordinates. We study the problem of finding a (non-unique) integer bar visibility graph representation of minimal width for any given tree.

To find such an optimal representation, we derive a number of theoretical lower bounds for the width of a tree. Various lower bounds can be given, depending on a number of properties of the given tree, such as the number of vertices, the vertex degrees, and the number of leaves.

We also study several constructive upper bounds on the width of an integer bar visibility representation for a tree. We approach the construction algorithmically, and this produces a number of results that limit the optimal width in terms of various balanced partitions of the leaves of the given tree.

Our investigations lead us to several results and conjectures concerning these balanced partitions. To the extent possible, we aim for the presentation of our results to be largely self-contained, requiring only a basic familiarity with common graph theoretic notions.

The paper is organized as follows. We begin in the next section with a collection of relevant definitions and terminology about graphs and trees that will be helpful for our work. We then review some elementary results about visibility graphs and the ways in which we hope to optimize such presentations. Next we present several theoretical lower bounds for the width of a tree. Then we introduce a number of key parameters related to vertex partitions of trees. These parameters will govern the width of our constructive algorithm that produces an integer bar visibility representation for any given tree. We extract an upper bound for the width of a tree through a mathematical analysis of the algorithm. We conclude with a number of open questions and some directions for future research.

2. Definitions and Preliminaries

In this section, we begin our work with a collection of relevant definitions and terminology about graphs and trees that will be helpful to our investigations. For a more complete introduction to the basic theory of graphs, we refer the reader to the textbook by Douglas West [5].

2.1 Graphs and Subgraphs

A **graph** is an ordered pair, G = (V, E), consisting of a (finite) set V of **vertices**, and a set E of unordered pairs of vertices, which are referred to as the **edges** of G. We say a graph H = (V′, E′) is a **subgraph** of G, denoted H $\subseteq$ G, whenever V′ $\subseteq$ V and E′ $\subseteq$ E. (We note that graphs constructed in this way are commonly known as **simple graphs**, which accordingly are undirected, unweighted, and have no loops or multiple edges.)

The two vertices of any edge are referred to as its **endpoints**, and we say such a pair of vertices u and v are **adjacent**, denoted u ~ v. The set of vertices adjacent to a given vertex v are called the **neighbors** of v. The **degree** of a vertex refers to the cardinality of its set of neighbors. The graph shown in Figure 1 below has 6 vertices and 8 edges. An **arc** is defined to be an ordered pair of adjacent vertices.

For any set of vertices S $\subseteq$ V, we define the **induced subgraph G[S]** to be the subgraph of G whose vertex set is S and where, for any vertices u,v in S, if u~v in G, then u~v in G[S]. In other words, induced subgraphs of G must contain all edges of G whose endpoints are contained in S. For example, the *path* with vertices A,D,E and edges {A,D},{D,E} is not an induced subgraph in Figure 1, but the *cycle*, with edges {A,D},{D,E}, and {A,E} *is* an induced subgraph.

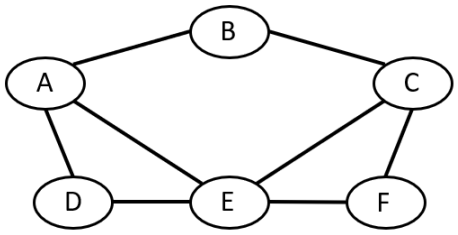

Figure 1
A simple graph with vertices A, B, C, D, E, and F

2.2 Paths and Cycles

A **path** in a graph is any sequence $v_1, v_2, \ldots, v_k$ of vertices that are distinct (no repeats) and consecutively adjacent, so that $v_i \sim v_{i+1}$ for each i ($1 \leq i < k$). The **length** of a path is one less than the number of vertices. For example, in Figure 1, the sequences D,E,F and A,B,C,F,E,D form paths of length 2 and 5, respectively. The distance between any two vertices u and v is denoted d(u,v), and equals the length of the shortest path containing u and v, or infinity if no such path exists.

We say that a graph is **connected** if, for every pair of vertices u,v, there exists a path containing u and v. The graph in Figure 1 is connected, while the graph in Figure 2 is disconnected. A **connected component** of a graph G is a subgraph which is maximal with respect to the property of being connected. In other words, a connected component is not properly contained in any other connected subgraphs of G. For example, in Figure 2, the subgraph induced on {1,2,3,4} is a connected component.

A **cycle** in a graph is similar to a path, except that we require the first and last vertices to be the same. Said another way, a cycle is any sequence $v_1, v_2, \ldots, v_k$ of distinct vertices that are cyclically adjacent, so that $v_1 \sim v_k$ and $v_i \sim v_{i+1}$ for each i ($1 \leq i < k$). For example, in Figure 1, the sequence A, B, C, F, E, D forms a cycle. A graph is said to be **acyclic** if it does not contain any cycles.

Although we most commonly view paths and cycles as subgraphs occurring within other graphs, we may also view them as families of graphs in their own right. For any positive integer n, the **path graph** on n vertices is denoted P_n, and the **cycle graph** on n vertices is denoted C_n.

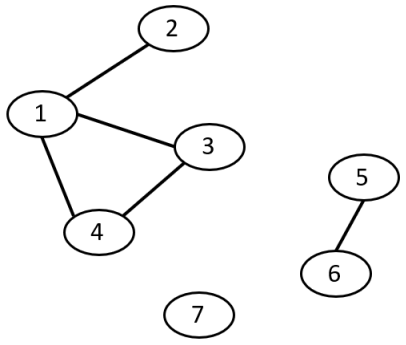

Figure 2
A graph with three connected components

2.3 Trees and Orderings

A **tree** is a graph which is both connected and acyclic. A **rooted tree** is a tree together with a distinguished vertex, called the **root**. Typically a rooted tree is presented with a planar embedding in which the root is located at the top, with its neighbors located one row below the root, and with the vertices at distance i from the root located i rows below the root. For example, Figure 3 shows a tree rooted at vertex 2. Note that in mathematics, a tree is not necessarily rooted, while in computer science, trees typically are rooted. A **leaf vertex** in a tree is a vertex with degree 1.

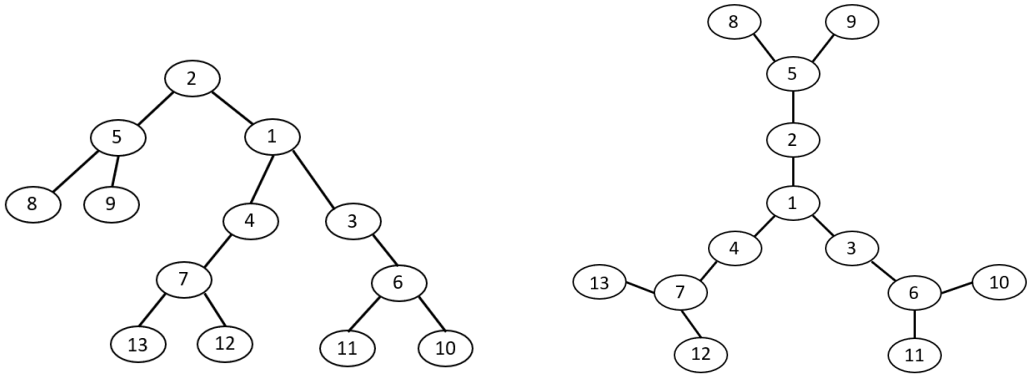

Figure 3
Left: rooted tree (with root 2)
Right: same tree, unrooted

We next consider the notions of **breadth- and depth-first orderings**, which refer to certain kinds of recursively-defined vertex orderings of a given graph.

- In a **breadth-first** ordering, we begin from an arbitrary starting vertex v and we proceed with an (arbitrary) order of its set N(v) of neighbors. Then, in turn, each set of the (new) neighbors of each u in N(v) is ordered. If the graph has multiple connected components, each component is ordered in this way. For example, the ordering:

  2, 5, 1, 8, 9, 4, 3, 7, 6, 13, 12, 11, 10

  is a valid breadth-first ordering for the rooted tree in Figure 3.
- In a **depth-first** ordering, we begin from an arbitrary starting vertex v and we proceed with an (arbitrary) order $v_1,v_2,...,v_k$ of its set N(v) of neighbors. Then, in a recursive manner, a depth-first ordering of $V-\{v,v_2,...,v_{k,}\}$is given for the set of (new) neighbors of

each v_i in N(v). For example, the ordering:

2, 5, 8, 9, 1, 4, 7, 13, 12, 3, 6, 11, 10

is a valid depth-first ordering for the rooted tree in Figure 3.

Note that these types of orderings can also be carried out for arbitrary graphs. We remark that if a given graph has multiple connected components, an arbitrary ordering of the components can be chosen, along with arbitrary starting vertices in each component. Then, in each component, a depth-first order may be given. For example, the ordering 6,5,7,2,1,3,4 is a valid breadth-first ordering for the graph in Figure 2.

In computer science, these orderings are commonly used to construct algorithms performed on a graph using a queue or stack, or, alternately, recursion. In particular, they are very effective and widely used to traverse quickly, or to search efficiently, the set of all vertices in a tree or a graph.

2.4 Visibility Graphs

Visibility graphs are generally used to model configurations consisting of multiple intervisible locations. The vertices of a visibility graph typically correspond to 2-dimensional shapes, with two vertices being adjacent to one another whenever their corresponding shapes can "see" each other, ie - if there exists at least one clear line of sight between them, free of obstructions. Conversely, we are sometimes given a graph, and we may be interested in trying to represent it by such a configuration of shapes. See Figure 4 to compare a visibility representation (on the left) with the associated graph (on the right).

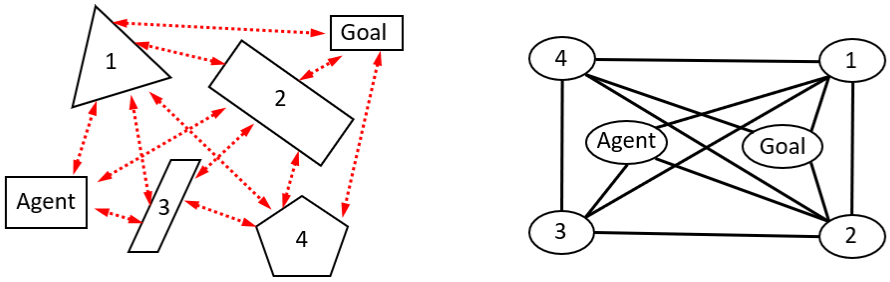

Figure 4
A visibility representation (left) of a graph (right)

A **bar visibility graph** is a visibility graph in which each vertex is represented by a horizontal line segment and all lines of sight are assumed to be vertical. An example of a bar visibility representation of a graph is given in Figure 5.

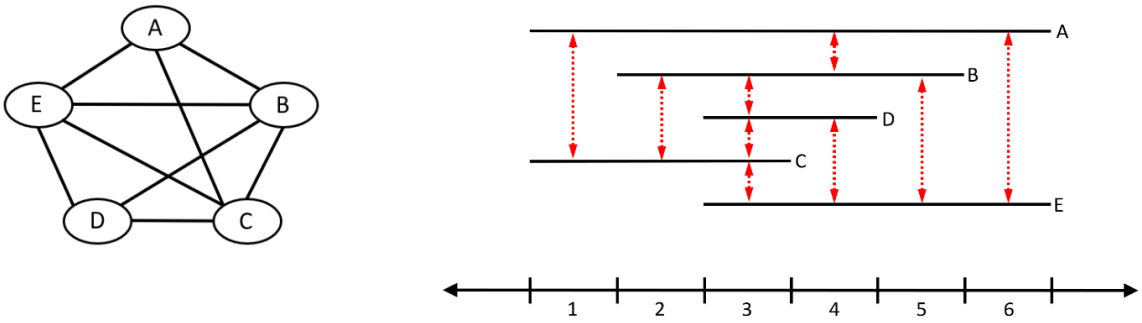

Figure 5
A graph and a bar visibility representation of width 6

An **integer bar visibility representation** of a graph G is a bar visibility representation of G in which all of the endpoints of the bars have integer coordinates. The representation in Figure 5 fits this criterion, and we will say it has width 6. In the next section we will examine this notion of width more closely.

3. The IBV-Width of a Graph

We now introduce one of the key concepts of this project. The **width** of an integer bar visibility representation refers to the maximum difference between any two x-coordinates of any bar endpoints. The IBV representation in Figure 5 is easily seen to have width 6, since the left endpoint of bar C is 6 units to the left of the right endpoint of bar E, and no two endpoints are separated by a greater horizontal distance.

The minimum width over all possible integer bar visibility representations for a given graph G will be designated as the **IBV-width** of the graph. We will denote this quantity by **width(G)** and, whenever context allows, we may refer to it simply as width.

3.1 Examples of Width

To further illustrate the notion of width, we consider a few more examples. As noted above, the IBV representation in Figure 5 is easily seen to have width 6. As a consequence, this establishes the fact that the graph in Figure 5 has an IBV-width that is at most 6. We note, however, that we cannot yet conclude that 6 is the actual width of this graph, since there may be other IBV-representations that achieve a smaller width.

As a second example, consider the IBV-representation in Figure 6 for the tree that was given in Figure 3.

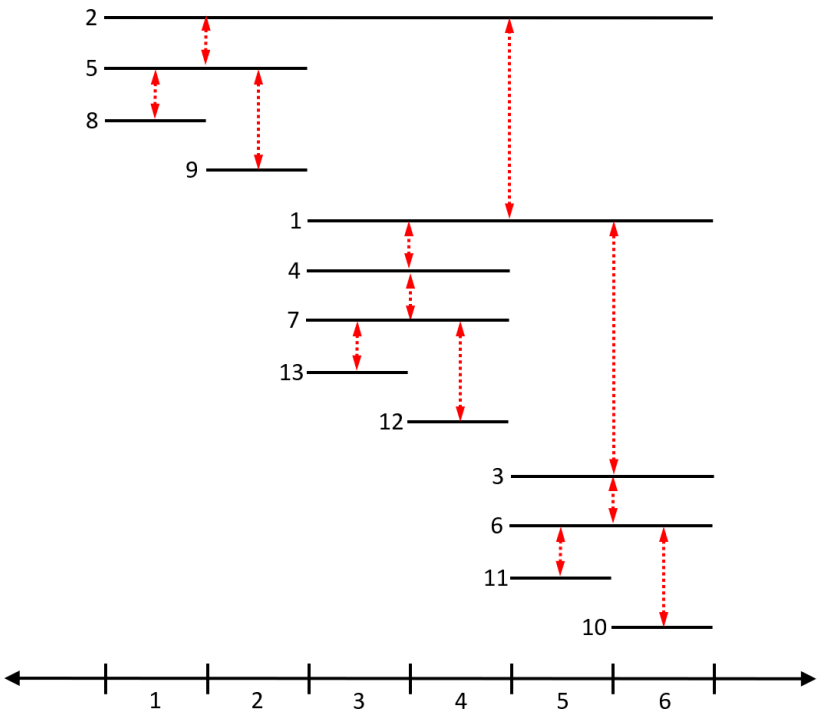

Figure 6
A width 6 IBV-representation of the tree in Figure 3

Notice that in this representation, the width equals the number of leaves of the tree. Here again, we observe that we cannot yet conclude that 6 is the actual width of this graph, since there may be other IBV-representations that achieve a smaller width.

Indeed, in Figure 7, we see that this same tree has an IBV-representation of width 4. However, if we want to prove the optimality of any such representation, we first need to establish a few results.

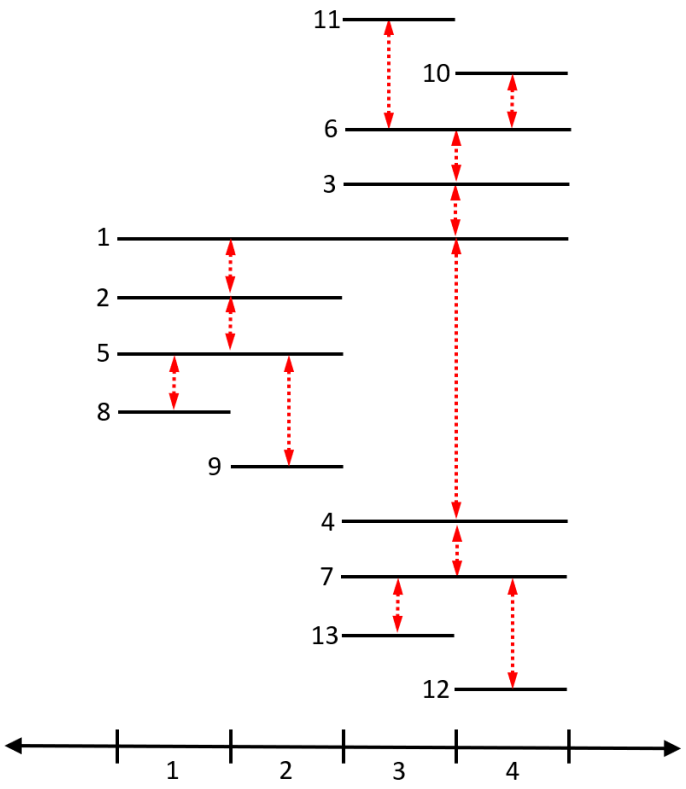

Figure 7
A width 4 IBV-representation of the tree in Figure 3

3.2 The Problem of Minimizing Width

The primary goal of this project is to find a (non-unique) *optimal* integer bar visibility graph for any given arbitrary tree, and thus to minimize the width. In order to find the optimal value, we desire to find a lower and upper bound on that value that are the same, and thereby prove that the optimal visibility graph must have a width equal to the bound.

One preliminary conjecture that we considered was the claim that the minimal width for a visibility representation of an arbitrary tree may be equal to or otherwise bounded above by L/2, half the number of leaves (rounding up in cases where the tree has an odd number of leaves). However, this preliminary conjecture was shown to be false when a counterexample was found (Figure 8 below).

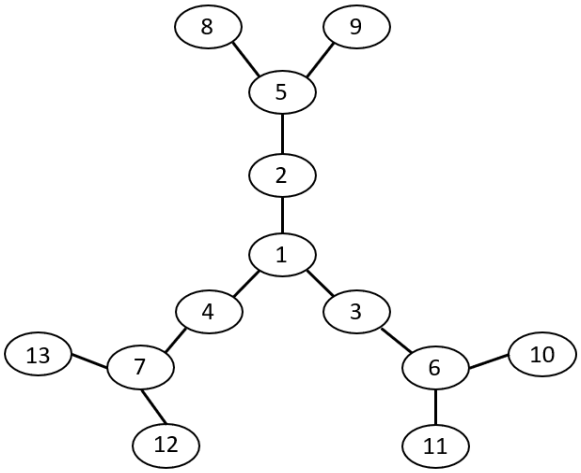

Figure 8
A tree whose width exceeds half the number of leaves

The tree depicted in Figure 8 has 6 leaves, but there is no known integer bar visibility graph for this tree which has a width less than 4. Computer searches have settled the matter definitively, so given the width 4 representation of Figure 7, we conclude that this tree has width 4, which is strictly greater than the ceiling of L/2. This demotivated our search for a proof that the width of trees could be bounded above by the ceiling of L/2, and we began our quest to find other bounds instead.

4. Lower Bounds on the Width of a Graph

In this section, we consider a number of lower bounds on the width of a graph.

4.1 Using the Maximum Degree

One factor that can affect the width of a graph is the maximum degree of its vertices. Consider the following result.

Lemma 4.1. The width of a graph must be at least half of the maximum vertex degree.

Proof. Suppose v is a vertex of maximum degree in a graph G. Let k denote the degree of v. Then any IBV representation must associate with v a horizontal line segment (a bar) whose endpoints

have integer coordinates. Suppose d denotes the length of the bar representing v. Since all vertices correspond to bars with integer coordinates, there are at most d lines of sight to distinct bars located downward from v and at most d lines of sight to bars located upward from v. In other words, the degree of v can be at most $2d$. Since the width of the representation must be at least as wide as the bar for v, we conclude that width(G) $\geq \lceil k/2 \rceil$. ■

We note that this bound is sharp, as evidenced by the family of graphs known as stars. A **star $K_{1,n}$** on n+1 vertices is a tree with n leaves, and a single vertex of degree n that is adjacent to them all. In Figure 9 below, we see how the bound of Lemma 4.1 is met for such graphs. Specifically, we represent vertex n+1 with a bar of length $\lceil n/2 \rceil$. Then we represent all of the leaves with bars of length 1, locating half (rounded up) of them below the bar for n+1 and the other half (rounded down) above the bar for n+1, as indicated.

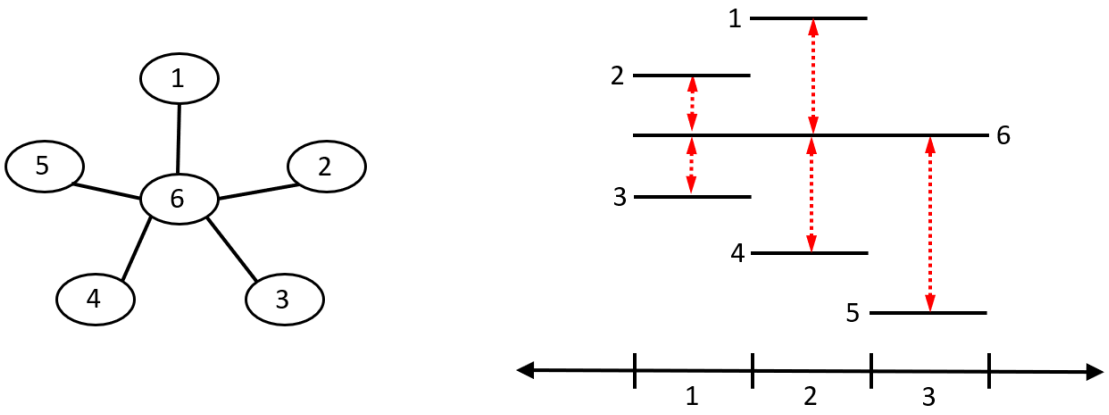

Figure 9
A star with 5 leaves and an IBV representation of width $\lceil 5/2 \rceil$

However, the bound of Lemma 4.1 can also be arbitrarily bad. A **rooted binary tree** is defined to have a root of degree 2, and every other vertex is either a leaf or a vertex of degree 3. See Figure 10 below for an example of a rooted binary tree on 13 vertices. Although such trees have maximum degree 3, they can have arbitrarily many leaves. In the next section, we will see that this means rooted binary trees exist with arbitrarily large width.

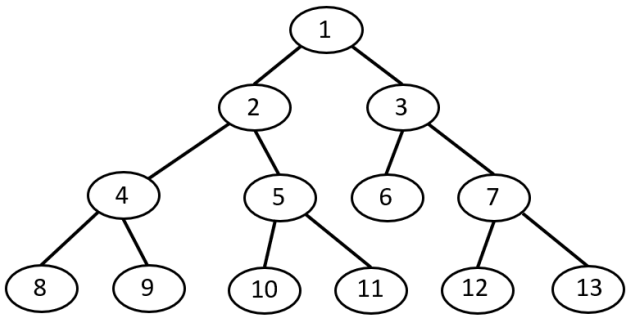

Figure 10
A rooted binary tree with 13 vertices.
This has maximum degree 3 and 7 leaves.

4.2 Using Half the Number of Leaves

Lemma 4.2. Half the number of leaves (rounded up) is a lower bound on the width of any tree.

Proof. Observe that the bar corresponding to any leaf must have length at least 1, and so it must have at least 2 lines of sight available (1 upward and 1 downward). Since, by definition, a leaf has only 1 neighbor, this implies that each bar corresponding to a leaf must have at least 1 empty line of sight. Such an empty line of sight must exit the representation through the top or bottom without meeting any other bars. But any representation of width w has at most $2w$ such lines of sight available (w along the top and w along the bottom). It follows that the number of leaves must be at most $2w$. Said another way, w must be at least half the number of leaves. ■

We note that this bound is sharp. Indeed, consider again the family of star graphs $\mathbf{K_{1,n}}$ on $n+1$ vertices that were discussed in Section 4.1 above. The construction of the IBV representation given there attains the width bound of Lemma 4.2.

We also note that the bound of Lemma 4.2 actually implies the result of Lemma 4.1, and is therefore a stronger statement. In particular, if we begin at a vertex of maximum degree, say k, and we construct a maximal path from v through any of its neighbors, these paths must terminate at leaves. So there exist at least k such paths, giving us at least k distinct leaves. It follows that half the number of leaves (rounded up) is at least as large as $k/2$ (rounded up), and therefore the bound of Lemma 4.2 is always at least as strong as the bound in Lemma 4.1.

However, as the tree in Figure 3 illustrates, this lower bound of Lemma 4.2 can also fall short of the true width, and we cannot yet rule out the possibility that there may exist infinite families of trees for which the bound becomes arbitrarily bad.

Trees that have been shown to have known bar visibility graph widths that differ from this value have been shown to be rare. Refer to the data gathered in Appendix C for details.

5. Upper Bounds on the Width of a Tree

In this section, we consider a number of upper bounds on the width of a graph.

5.1 Using the Number of Leaves

Lemma 5.1. Any rooted tree with at least two vertices and L leaves has an IBV-representation with width L, in which the root is the top bar, and where all of the leaves (distinct from the root) correspond to bars of length one located below the root bar. As a consequence, the width of any tree with at least two vertices is at most the number of leaves.

Proof. For a tree on only two vertices (and thus with two leaves), it is trivial to construct a bar visibility representation of the desired form with width $\leq L = 2$.

For our induction hypothesis, we suppose that for any tree, T, on n vertices, with L leaves, there exists a visibility graph, B, of width L.

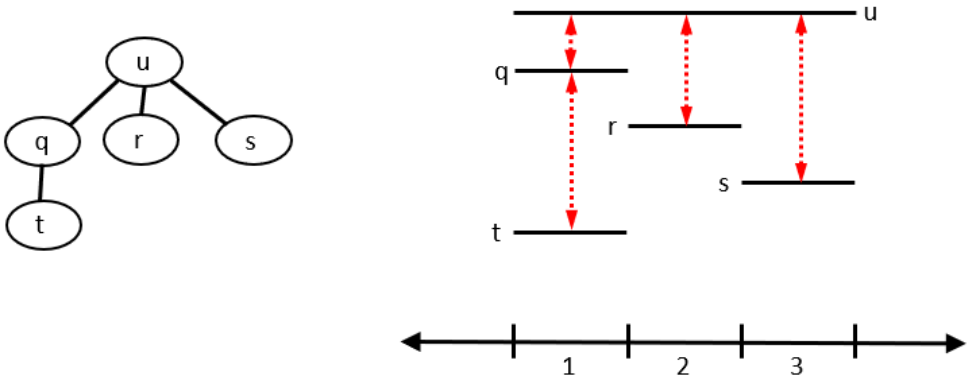

Figure 11

An arbitrary tree, T, with a construction of a bar visibility graph, B.

We want to show that if we insert a new vertex into the tree, then we can construct a bar visibility graph for our modified tree T on $n+1$ vertices that will have a width of at most equal to the number of leaves.

Case 1: If we insert an additional vertex, v, as a leaf, then v will be adjacent to exactly one other vertex, u, in the tree. Consider the bar, b_u, which represents vertex u in the visibility graph for T. We may need to extend b_u by a single unit to make room for b_v, the bar representing the new vertex, v, if there are no "slots" available such that the bar b_u has no visible vertices in any of its columns in either direction. In doing so, we may also need to adjust the other bars in the graph to the left or right to make room for the extension, but the relationships between the other bars will be preserved.

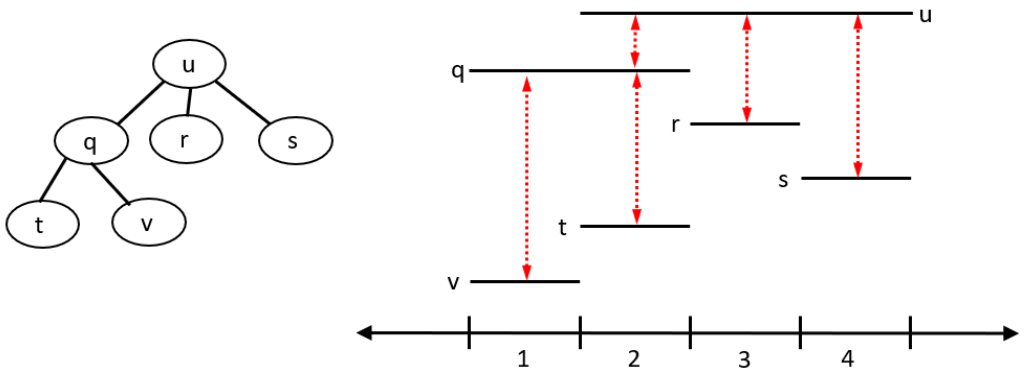

Figure 12

The modified tree T, after insertion of v, and its modified bar visibility representation

Case 2: If, instead, we were to insert the additional vertex, v, as the root of the tree, making the previous root, u, the child of v, we need merely to prepend a bar, b_v, of the same width and in the same columns as the bar, b_u, representing its previous root, u, to the visibility graph, which will not augment its width at all.

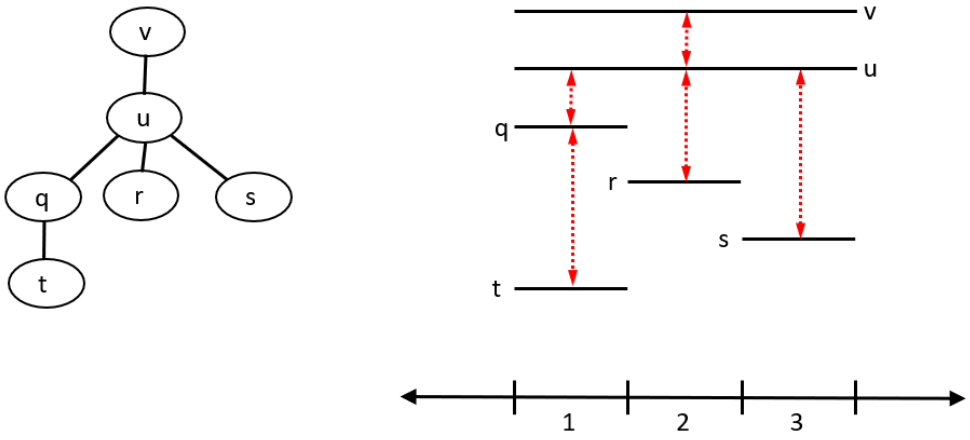

Figure 13

The modified tree *T*, after inserting *v* as root, and its modified bar visibility representation

Case 3: Consider the case in which the inserted vertex, *v*, is neither a root nor a leaf. In this case, the new vertex, *v*, takes the place of an existing vertex, *u*, in the graph, with *u* now being the child of *v*, and the previous parent of *u* being the parent of *v*. We may use a very similar method to that of Case 2. Insert above b_u a new bar, b_v, of the same width and occupying the same columns as b_u. This operation will not modify the width. Therefore, by mathematical induction, a bar visibility graph of width *L* can be constructed for any arbitrary tree, *T*. ■

Although the bound given above is easily implemented, it is wasteful to place all leaf intervals below the root interval. Indeed, this upper bound can be shown to be arbitrarily inefficient.

To illustrate this point, we (yet again) consider the family of star graphs, defined in Section 4.1 above. For any star graph, $K_{1,2n}$ recall that we can easily construct a bar visibility graph of width *n* by merely partitioning the 2*n* leaves of $K_{1,2n}$ into two sets, V_1 and V_2, of equal cardinality, *n*, and orienting all of the single-unit width bars corresponding to the vertices in V_1 on one side of the bar representing the central vertex, *c*, and orienting all of the single-unit width bars corresponding to the vertices in V_2 on the opposing side.

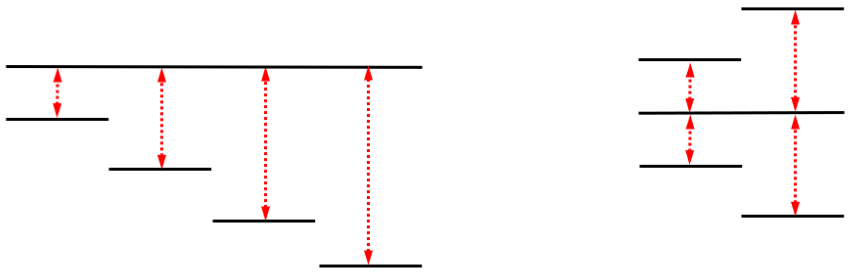

Figure 14
Left: a sub-optimal representation for $K_{1,4}$ of width 4 using the method in Lemma 5.1.
Right: an optimal representation for $K_{1,4}$ of width 2, as described above.

Thus the optimal width for a bar visibility graph for any star graph $K_{1,2n}$ cannot be larger than n, however, our upper bound from Lemma 5.1 gives a representation of width $2n$.

5.2 Using the Minimum Edge Profile

Given any tree T and arc (u,v), we define the **leaf degree,** denoted $\boldsymbol{l(u,v)}$, to be the number of leaves w such that the (unique) u-w path contains v. For example, in Figure 15, consider the arc (3,12). The leaf degree for this arc is 2, because there are exactly two leaves (vertices 4 and 10) that are connected to vertex 3 by paths that contain vertex 12.

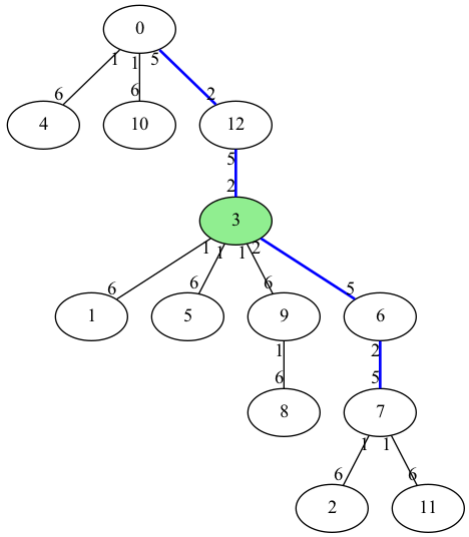

Figure 15

A tree with the leaf degrees labeled for each edge

The **profile of an edge** is the larger of the two leaf valances associated with the two arcs that correspond to that edge. For example, the profile of the edge {3,12} in Figure 15 is 5, since 5 is the larger of the two leaf degrees $l(3,12)=2$ and $l(12,3)=5$.

We define the **edge profile of a tree** to be the minimum profile for all edges in the tree. For example, the edge profile of the tree in Figure 15 is 5, and the edges with the minimum profile are indicated in blue.

The subtree induced by the set of edges with minimum profile shall be called the **edge visibility center.** The edge visibility center of the tree in Figure 15 consists of the path induced on vertices 0,12,3,6,7.

Lemma 5.3. The width of a tree with at least 2 vertices cannot exceed the edge profile.

Proof. Suppose T is a tree with at least 2 vertices and edge profile k. Then there exists at least one edge $e = \{u,v\}$ with profile k. Associated with this edge e is at least one arc, say (u,v), for which $l(u,v) = k$. It follows from the definition that $l(v,u) \leq k$.

Note that T-$\{e\}$ has exactly 2 connected components T_u and T_v, where T_u is a tree rooted at u, and T_v is a tree rooted at v. Also observe that T_u has exactly $l(u,v) = k$ non-root leaves, and T_v has exactly $l(v,u) \leq k$ non-root leaves.

So the construction of Lemma 5.1 gives us two IBV representations, one for T_u of width k with the bar for u on top, and the other for T_v of width $\leq k$ with the bar for v on top. By flipping the representation for T_v upside-down and locating it above the representation for T_u (see Figure 16), we obtain a representation for T of width k. ■

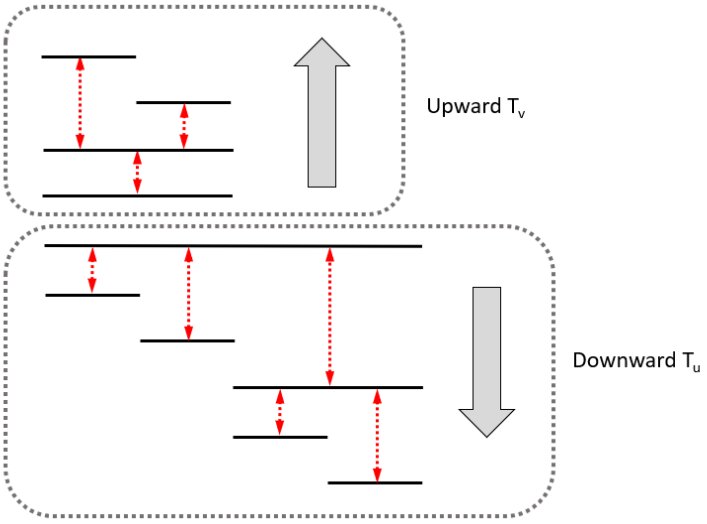

Figure 16

Two rooted trees combined along an edge

But the bound of Lemma 5.3 can also be quite bad. For example, with stars, all of the edges actually share the minimum profile. For these trees, unfortunately this technique places all but one leaf on the same side of the bar representing the central vertex. In this case, it is natural to consider splitting the representation around a vertex rather than an edge. This idea is explored in the next section.

5.3 Using the Minimum Vertex Profile

To define the **profile of a (non-leaf) vertex**, u, we must consider all possible partitions $N=A \cup B$ of the set N of neighbors of u into two disjoint non-empty subsets A and B. Define the **leaf sums** to be the quantities:

$$\mathbf{\Sigma}\mathrm{A} = \sum_{a \in A} l(u,a) \quad \text{and} \quad \mathbf{\Sigma}\mathrm{B} = \sum_{b \in B} l(u,b).$$

Among all such partitions, we seek to minimize the larger of ΣA and ΣB. The minimum of this maximum over all such partitions is denoted by **profile(u)**. In other words,

$$\text{profile}(u) = \min_{A,B \subseteq N} \{\max\{\Sigma A, \Sigma B\} : N = A \cup B\}$$

We note that the profile is attained for a partition in which the two leaf sums are as close to being equal as possible. For example, the profile(0)=5 in Figure 15, since ΣA=2 and ΣB=5, where A={4,10} and B={12}. Moreover, profile(3)=4, which is attained by the partition A={1,12} and B={5,6,9}, where we have ΣA=3 and ΣB=4.

We define the **vertex profile of a tree** to be the minimum profile for all vertices in the tree. For example, the vertex profile of the tree in Figure 15 is 4, which is attained by vertex 3 and the partition described above. Note that other partitions around vertex 3 also achieve this same value.

We will call the subgraph induced by the set of vertices that have the minimum profile the **vertex visibility center.** The vertex visibility center of the tree in Figure 15 contains only vertex 3.

Lemma 5.4. The width of a tree with at least 3 vertices cannot exceed the vertex profile.

Proof. Suppose T is a tree with at least 3 vertices and vertex profile k. Then there exists at least one non-leaf vertex u with profile k. Associated with this vertex u is a partition A,B of its neighbors where $\Sigma A \leq \Sigma B = k$.

Note that T-A has exactly $|A|$+1 connected components, one of which is a tree T_B rooted at u. Similarly, T-B has exactly $|B|$+1 connected components, one of which is a tree T_A rooted at u. Observe that trees T_A and T_B share only vertex u, and T is the union of T_A and T_B. Also observe that T_B has exactly k non-root leaves, while T_A has at most k non-root leaves.

So the construction of Lemma 5.1 gives us two IBV representations, one for T_B of width k with the bar for u on top, and the other for T_A of width $\leq k$ with the bar for u on top. By flipping the representation for T_A upside-down, deleting the bar for u, and locating it above the representation for T_B (see Figure 17), we obtain a representation for T of width k. ■

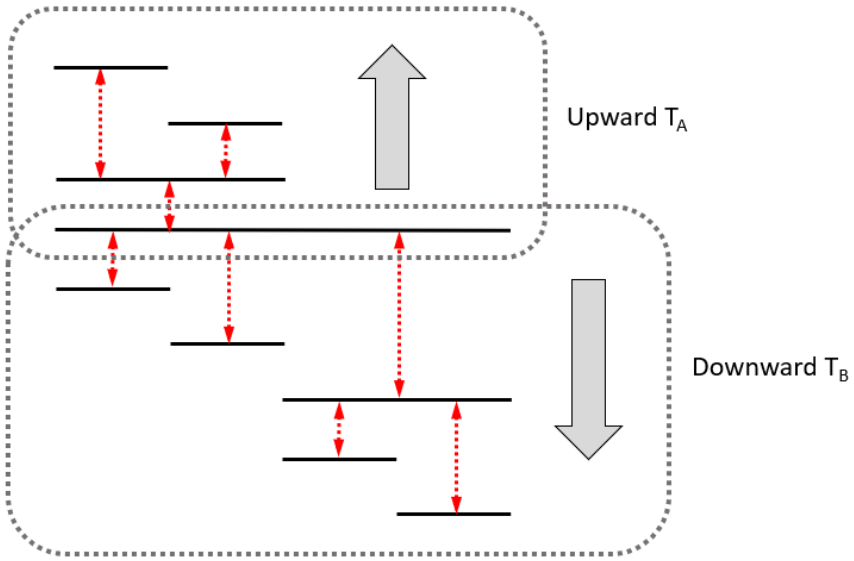

Figure 17

Two rooted trees joined at a shared vertex

Computational experimentation suggests that the vertex profile bound performs quite well in practice (refer to Appendix C for data illustrating the quality of this bound). That said, we conjecture that this bound can also be made arbitrarily bad, using trees with 3-fold symmetry similar to the one depicted in Figure 8.

Nonetheless, we are able to prove that the vertex profile bound of Lemma 5.4 is always at least as good as the edge profile bound of Lemma 5.3. In particular, the following holds.

Lemma 5.5. For any tree T with at least 3 vertices, the vertex profile does not exceed the edge profile.

Proof. Let T be any tree with at least 3 vertices and suppose T has vertex profile x and edge profile y. By the definition of edge profile, there must exist at least one arc (u,v) with leaf degree y. But now we can partition the neighbors of u so that $B = \{v\}$. This implies that u has a vertex profile less than or equal to y. It follows that the vertex profile of T is also less than or equal to y, as desired. ■

We know of no trees that require more than one extra unit of width above the lower bound given in Lemma 4.2. So it is possible, with this lack of counterexamples, to hazard a conjecture such as the following.

Conjecture 5.5. The width of any tree with L leaves is at most 1+ceiling(L/2).

6. Multiset Partitioning Problem

The algorithm described in this project for finding the vertex profile of a tree includes, as a subtask, an instance of a famous NP-hard problem known as the "partition problem". In the partition problem, we are given a multiset of numbers and tasked with finding a partition into two sub-multisets whose element sums are as close to equal as possible. This deceptively difficult problem is sometimes referred to in the mathematics community as "the easiest hard problem" (Hayes).

In the algorithm for finding the optimal width of a visibility bar graph described in the section above, a variant of the partition problem is required to partition the multiset of leaf degrees such that the difference of their sums are minimized, i.e., the nearest possible partition. For further reading about the partition problem, we refer the interested reader to *Multi-Way Number Partitioning* (Korf).

A non-optimal version of the nearest-partition algorithm is implemented in the current Python code described in the proceeding section at the time of this writing. However, since the size of the multi-sets being partitioned is reasonably bounded in practice, the code runs in real-time quickly despite the high asymptotic time complexity.

An interesting by-product of our data is the suggestion that our specific instance of the partition problem is not at all difficult to solve quickly. In particular, it appears to be the case that the vertices of minimum vertex profile can be greedily found in any given tree T by following a simple heuristic. Beginning at any vertex u of T, greedily choose the arc (u,x) that has the greatest leaf degree and walk along that arc from u to x. Repeat until no arc from the current vertex can increase the most recent leaf degree. It seems that this always locates the vertices of minimum profile in T, and a similar effect seems to work for locating the edges of minimum profile. This leads us to the following conjecture.

Conjecture 7.1. The greedy algorithm described above always identifies the visibility centers of any given tree.

The interested reader may enjoy experimenting with this algorithm on the labeled tree below.

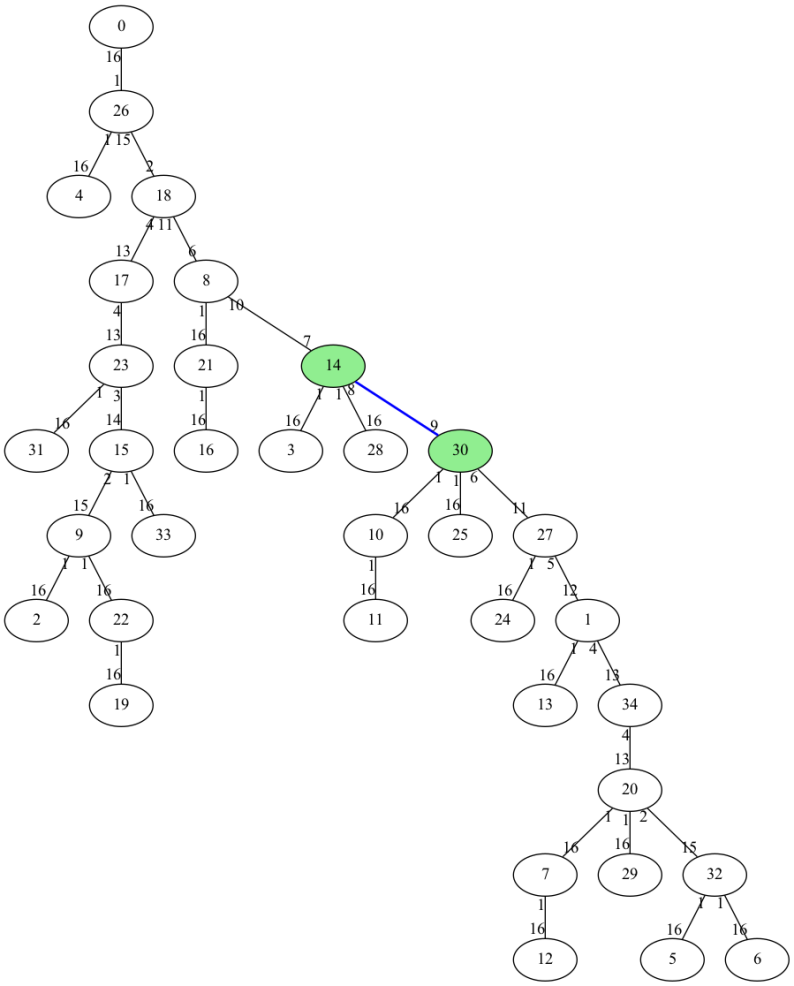

Figure 18

An example output from the code

We conclude this section with a number of further observations/conjectures relating to the two kinds of centers we have studied.

- The subgraph of a tree induced by the least profile vertices is connected.
- The subgraph of a tree induced by the least profile edges is connected.

- The vertex and edge visibility centers are nested (the vertex center is a subgraph of the edge center).
- There exist trees for which not every vertex in the edge visibility center belongs to the vertex visibility center. (See Figure 15 above!)

7. Conclusion and Directions for Further Research

In this project, we explored integer bar visibility representations of trees. We were motivated to try to minimize the width of such representations, which led us to seek upper and lower bounds, as well as to the formulation of algorithms to find representations efficiently. We introduced a number of parameters describing some basic ways of splitting the leaves of a tree into various subsets. Using these parameters, we were able to establish a number of bounds that constrained the width as desired. Related to these results, we observed a number of properties enjoyed by the subgraphs induced on vertices of minimum profile. We collected several of these observations as conjectures and we plan to continue researching these topics in the future.

References

[1] Hayes, Brian (March–April 2002), "The Easiest Hard Problem" (PDF), *American Scientist*, Sigma Xi, The Scientific Research Society, vol. 90, no. 2, pp. 113–117.

[2] Garey, M.R., Johnson, D.S., So, H.C.,“An application of graph coloring to printed circuit testing”. *IEEE Trans. Circuits and Systems* CAS-23(**10**), 591–599 (1976).

[3] Korf, Richard E. (2009). *Multi-Way Number Partitioning*(PDF).

[4] Lozano-Pérez, Tomás; Wesley, Michael A. (1979), "An algorithm for planning collision-free paths among polyhedral obstacles", *Communications of the ACM*, **22** (10): 560–570.

[5] West, Douglas Brent. *Introduction to Graph Theory*. Pearson, 2018.

[6] Wismath, S.K.: Characterizing bar line-of-sight graphs. In: Proceedings of the First Symposium of Computational Geometry, pp. 147–152. ACM, New York (1985).

Appendix A. Algorithms to Find Visibility Centers

A breadth- or depth-first search is performed on a tree from any arbitrary vertex to determine its number of leaves.

A second breadth- or depth-first search is performed beginning at an arbitrary leaf, and the leaf valances are determined for each vertex-edge pair in the tree, whilst keeping track of the minimum vertex profile found.

Appendix B. Python Code

All code written for this project is available on Github:

https://github.com/Haleyo/OptimalVisibilityGraphs/tree/main/gen_vis_graphs

There are two projects contained in this repository, referred to as **gen_trees** and **gen_vis_graphs**. The **gen_vis_graphs** project is of greater interest.

This code relies on the library **networkx**, an open source library for working with graphs and trees. The code uses **networkx** to generate an arbitrary tree on n vertices.

This project also uses **graphviz**, a library used for visualizing graphs.

The code in **novel_algorithm.py** is called from the main file, and it finds the leaf degree for each vertex-edge pair in the tree, and calculates the profile of each vertex and edge.

The output of the script is the printed profile of the tree, as well as an image like Figure 19 below, which labels the leaf degree for each vertex-edge pair. The resulting image also has the vertex visibility center highlighted in green, while the edge visibility center is highlighted in blue.

Appendix C. Data on Tree Profile

The following table documents experimental data gathered by running the main script in the **gen_vis_graphs** project with the inputs from the first and second columns.

The "% min prof" (percent minimum profile) columns represent the percentage of trees generated with minimum vertex profiles within the range specified. For example, in row 1, we see that 1,000,000 random trees were generated on 7 vertices, and 100% of the trees generated had a minimum vertex profile that was less than 1 plus half the number of leaves ($L/2 + 1$).

Note: do not run the code on 1,000,000 trees without first commenting out the call to the "create_image" function in main.py, otherwise you will get 1,000,000 images.

num_nodes	num_trees	% min prof < $L/2 + 1$	% min prof: $L/2 + 1$	% min prof: $L/2 + 1.5$	% min prof: $L/2 + 2$
7	1,000,000	100%	-	-	-
8	1,000,000	100%	-	-	-
9	1,000,000	100%	-	-	-
10	1,000,000	99.9273%	0.0727%	-	-
11	1,000,000	99.6804%	0.3196%	-	-
12	1,000,000	99.2704%	0.7296%	-	-
13	1,000,000	98.7642%	1.2357%	0.0001%	-
14	1,000,000	98.2528%	1.7454%	0.0018%	-
15	1,000,000	97.7688%	2.2178%	0.0134%	-
16	1,000,000	97.3434%	2.6235%	0.0331%	-
17	3,000,000	96.8511%	3.0648%	0.0841%	-
18	3,000,000	96.3418%	3.4958%	0.1623%	3.33×10^{-5} %
19	3,000,000	95.8420%	3.8850%	0.2726%	0.0004%
20	3,000,000	95.2664%	4.3186%	0.4135%	0.0014%

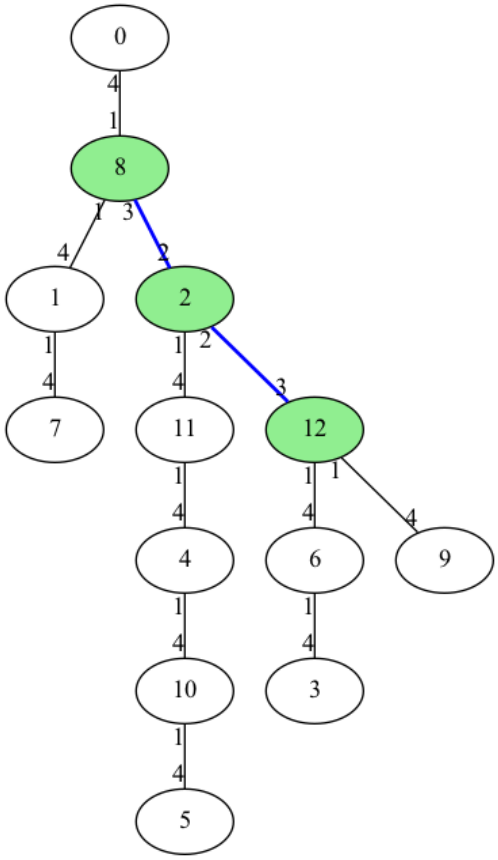

Figure 19 - an example output of the code

Requirements for running the script include **python3**, **networkx** library and the **graphviz** library.

To run the script, once all requirements are installed, simply navigate to the **gen_vis_graphs** folder and run the command:

python3 main.py --num_nodes=n --num_trees=m

Here you may replace the inputs *n* and *m* with the number of vertices in your tree and the number of trees respectively. **Care should be taken to not run the code with inputs that are too large.** For every tree generated, an image will be saved, and, given that the algorithm runs in non-polynomial time, the time and space complexity is necessarily unwieldy. It is recommended that you run this code to start with 10 vertices and only 1 tree, e.g.,

python3 main.py --num_nodes=10 --num_trees=1

and increment slowly. If the inputs are forgotten or entered incorrectly, a helpful output will be printed. The script may also be run with the "--help" flag, e..g,

python3 main.py –help

for a reminder.

Once run, the code will generate a pseudorandom tree on *n* vertices, and will execute the algorithm described on this tree to determine its visibility centers and profiles. The output will be found in the **TreeImages** folder in the same directory.