# Drawing String Graphs for 8-Grid Outerplanar Grid Drawings

Daniel R. Page*

Department of Computer Science, University of Manitoba

drpage@pagewizardgames.com

October 2014

**Abstract**

In this paper we present a linear-time algorithm for drawing String Graphs in the plane for a restricted type of outerplanar grid drawing called an 8-Grid Outerplanar Grid Drawing. If a Grid Drawing has edges that only intersect integer coordinates and every vertex has at most eight neighbours, then a Grid Drawing is 8-Grid. If an 8-Grid Outerplanar Grid Drawing of a graph $G$ is provided, then the algorithm draws its String Graph representation. If a drawing is given with a grid size $k \times s$, the algorithm draws a String Graph with grid size $((4k+2)r) \times ((4s+2)r)$, where $r$ is the scaling factor. We also prove the correctness of the algorithm, and propose three open problems related to 8-Grid Outerplanar Grid Drawings.



Completed: May 2013 Published: October 2014

keywords: graph drawing, string graphs, outerplanar graphs, grid drawings.

## I. Introduction

A *string graph* is a particular type of intersection graph. To begin, one must define an intersection graph:

**Definition 1** (Intersection Graph)**.** An intersection graph is an undirected graph $G = (V, E)$, such that:

- Each $v_i \in V(G)$ represents a set $S_i \in S$, where $S = \{S_1, S_2, \ldots, S_n\}$, $1 \leq i \leq n$.
- $E(G) = \{\{u, v\} \mid S_u \cap S_v \neq \varnothing\}$.

Now, a *string graph* is a specific class of intersection graphs where curves are considered in the plane.

**Definition 2** (String Graph)**.** A graph $G$ is an intersection graph called a string graph if and only if:

- When drawn in the plane, a string graph is a set of simple curves called "strings".
- At most two strings intersect at a single point when drawn in the plane.
- For each vertex $v \in V(G)$ and edge $\{u, v\} \in E(G)$, a vertex $v$ corresponds to a curve (string) and the intersection of any two curves (strings) is an edge.

*The research presented in this paper was completed as a course project for a graduate course in Graph Drawing taught by Dr. Stephane Durocher in Winter 2013.

String graphs have a numerous intractable properties that make them hard to classify, but there has been promise recently in this domain for related graph classes. Some modern results consider different graph classes and attempt to determine their equivalency to string graphs. This is in order to obtain certain tractable properties for subclasses of string graphs, or to show related graph classes inherit intractable properties for classification, if equivalent [1, 6]. Here are some interesting graph theoretic and complexity properties for string graphs:

- To determine if any graph $G$ is a string graph is decidable [8, 13], but is **NP**-complete [7, 12].
- To determine the chromatic number of a string graph is NP-hard [5].
- All planar graphs can be represented as a string graph [4, 15]. Also, not all graphs are string graphs [15].
- Scheinerman's Conjecture [14] states, "Every planar graph is the intersection graph of a set of segments in the plane". This conjecture was proven in 2009 by Chalopin and Gonçalves [2]. This implies that all planar graphs can be drawn as string graphs with at most one curve crossing per pair of curves. This result was directly proved in 2007 as well [3].

Due to the complexities related to even asking if a graph $G$ has a string graph representation, it has been difficult to capture a general algorithm for drawing string graphs. We present positive results for outerplanar graphs that have a property we will define in the next section called the 8-Grid property. We will present an algorithm for drawing outerplanar graphs that have a drawing that satisfies this property.

## II. Problem

Our problem considers a restricted case of outerplanar graphs. Let us define an outerplanar graph.

**Definition 3** (Outerplanar Graph)**.** A graph $G = (V, E)$ is outerplanar if there exists a drawing $D$ in the plane of $G$ such that two jordan arcs connecting vertices only intersect at their end points, and every vertex is incident to the outer face.

Next, we are considering the drawing convention of a grid drawing. We also will now define a grid drawing.

**Definition 4** (Grid Drawing)**.** A drawing $D$ is a grid drawing if every vertex is drawn in the plane at integer coordinates, and every edge is a straight line segment that only intersects at integer coordinates.

For our algorithm we will only be considering outerplanar graphs that have the property we will call the 8-Grid property.

**Definition 5** (8-Grid)**.** A graph $G$ with a drawing $D$ has the 8-Grid property only if it satisfies the following properties:

- every vertex is drawn at integer coordinates,
- every edge crosses on the grid at integer coordinates (every slope is of positive or negative unit size),
- every vertex has a neighbourhood of size of at most eight.

Our problem is to construct a string graph from an 8-Grid outerplanar grid drawing (8-GOGD) and a corresponding outerplanar graph $G$, if such a graph $G$ exists. Currently no general drawing algorithm exists for all string graphs, nor is there a tractable test to know if a string graph can be

drawn from a graph $G$, if $\mathbf{P} \neq \mathbf{NP}$ [7, 12]. By restricting our problem to this type of input graph and particular drawing convention we can find special cases that do have such tests of existence, and drawing algorithms we may use to construct a string graph. For instance, outerplanar graphs that have an orthogonal drawing with no edge bends, or even a rectangular drawing satisfy the 8-Grid property. Algorithms for testing and constructing such drawings do exist in linear time for outerplanar graphs [10, 11]. For example, Rahman [10] gave a method of testing if a plane graph had an orthogonal drawing with no edge bends and provided an algorithm for finding such a drawing if it existed. Such special cases among others can be used as input for our algorithm if the graph is to be sought.

## III. Algorithm

The inputs of our algorithm are an outerplanar graph $G$ with an outerplanar grid drawing $D$ that satisfies the 8-Grid property, and a scaling factor $r$ for resizing the drawing. We now will present an algorithm that takes an 8-Grid Outerplanar Graph Drawing (8-GOGD) of $G$ and constructs a string graph representation of $G$.

### I. Preliminaries

Before we describe the algorithm, we wish to describe some important notation, and concepts leading into the algorithm.

#### I.1 Assumptions

Given an 8-Grid Outerplanar Grid Drawing (8-GOGD) $D$ of a graph $G = (V, E)$, let $p(v) = (x(v), y(v))$ denote the $(x, y)$-coordinate of a vertex $v \in V(G)$ in the drawing $D$ in the plane. Also, let $e(u, v) = (p(u), p(v))$, where $\{u, v\} \in E(G)$. Assume the drawing $D$ is given as a list of vertices in a lookup-table for accessing $(x, y)$-coordinates for each vertex in $O(1)$ time. The edges $E$ of $G$ are given as an adjacency list where the list is stored as a lookup table of edge lists (in sorted order by vertex number, for every vertex in $V(G)$) where each edge list is stored as a linked list. Each of the neighbours stored in each edge list are provided in increasing order. The string graph is outputted as $n = |V(G)|$ lists. Each list is a sequence of points where convex curves are drawn between each point to form a string. Since there are $n$ lists, we obtain $n$ strings.

#### I.2 Chromatic Number of Outerplanar Graphs, and Colouring of an Outerplanar Graph

Let $\chi(G)$ denote the chromatic number of a graph $G$. For any outerplanar graph $G$, $\chi(G) \leq 3$ [9]. This means every vertex $u \in V(G)$ can be labelled $c(u) \in \{0, 1, 2\}$. Fortunately, there exists a polynomial time algorithm by Proskurowski and Syslo [9] that can colour any outerplanar graph with at most three colours. This algorithm relies on constructing a block decomposition of $G$ (bi-connected components) initially, and then performing a depth-first traversal on each face and colouring each vertex in a greedy fashion. The worst-case execution time for this algorithm depends on the block decomposition of $G$, and the depth-first traversal of the faces. For outerplanar graphs, using Tarjan's [16] algorithm for finding the bi-connected components (blocks) of $G$ yields $O(n)$ worst-case time. In the worst-case, the depth-first traversal of the faces has $O(f)$ time complexity, where $f$ is the total number of faces. Since the total number of vertices is proportional to the total number of faces by Euler's Formula, the execution in the worst-case for colouring $G$ is $O(n)$.

If we have such a colouring, every vertex $u$ is given a label $c(u)$ that is unique compared to its neighbourhood. This means every edge $\{u, v\} \in E(G)$ can represent as one of six possible cases.

Based on each colouring, we can identify the type of intersection between two strings that represent an edge relative to a vertex.

Let $\gamma(u, v) \in \{0, 1\}$ denote the *intersection label*, where $u, v \in V(G)$. The *intersection label* of an edge from $u$ to $v$ can be determined using a lookup table as seen in Table 1. The *intersection label* between an edge $\{u, v\}$ relative to $u$ and an edge $\{v, u\}$ relative to $v$ have complementary values, so we can construct each hook that crosses another string systematically.

| $(c(u), c(v))$ | $\gamma(u, v)$ |
|---|---|
| (0,1) | 0 |
| (0,2) | 0 |
| (1,2) | 0 |
| (1,0) | 1 |
| (2,0) | 1 |
| (2,1) | 1 |

**Table 1:** *The colouring table to compute an intersection label $\gamma(u, v)$. This value determines how two strings cross.*

### I.3 Determining Orientation of Curves

In the given graph $G$, we assuming its provided drawing $D$ has the 8-Grid property. It would be ideal to have a method to quickly determine points where we can reduce trigonometric computations. Define a labelling for each edge relative to a vertex. Each label will correspond to an angle relative to a unit circle around the given vertex. Since the drawing has the 8-Grid property, we need an entry and leave for each arc in the string to cross another vertex. This would be a total of sixteen points on the unit circle. This implictely creates twenty four labels for aesthetic reasons as if an edge doesn't exist for one direction, we would require a *dummy* point to replace where a gap would otherwise exist. This can be seen in Figure 1.

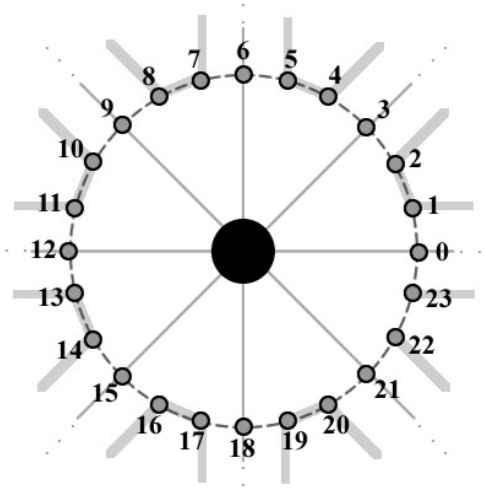


**Figure 1:** *Directional labelling of edges relative to a vertex. Every third label corresponds to the direction of an edge in each neighbourhood of a given vertex in the input drawing D.*

We can pre-compute the values for the points that form strings along the unit circle using

simple trigonometric functions. Let us construct a table that corresponds to each $x$-component and $y$-component along the unit circle for each point. Let $l(e,u)$ denote the directional label of an edge $e \in E(G)$ relative to a vertex $u \in V(G)$. In the algorithm we will refer to the lookup table seen in Table 2 as $LabelPoint$.

| $l(e,u)$ | x | y |
|---|---|---|
| 0 | $cos(0)$ | $sin(0)$ |
| 1 | $cos(\frac{2\pi}{24})$ | $sin(\frac{2\pi}{24})$ |
| 2 | $cos(\frac{4\pi}{24})$ | $sin(\frac{4\pi}{24})$ |
| $\vdots$ | $\vdots$ | $\vdots$ |
| $j$ | $cos(\frac{2j\pi}{24})$ | $sin(\frac{2j\pi}{24})$ |
| $\vdots$ | $\vdots$ | $\vdots$ |
| 23 | $cos(\frac{46\pi}{24})$ | $sin(\frac{46\pi}{24})$ |

**Table 2:** *Precomputation of $x, y$ values along the unit circle that are stored in a table called $LabelPoint$.*

Now, for each edge relative to a vertex, we can assign a *directional label*. These directional labels will be used to identify the role of each edge in structure of the string graph for each vertex. Given an edge and its corresponding location in a drawing, we can determine its directional label relative to a given vertex as follows:

**Algorithm 1:** Determining the directional label $l(e,u)$.

**Data**: Given an edge in the drawing corresponding to $\{u,v\} \in E(G)$,
$e(u,v) = (p(u),p(v)) = ((x(u),y(u)),(x(v),y(v))$.
**Result**: $l(e,u)$ for an edge $e \in E(G)$ relative to vertex $u \in V(G)$.
$label := -1$; *//Directional label initialization*
$x_0 := x(u)$; $y_0 := y(u)$; $x_1 := x(v)$; $y_1 := y(v)$;
**if** $(y_1 - y_0) > 0$ **then**
  **if** $(x_1 - x_0) > 0$ **then**
    $label := 3$;
  **else if** $(x_1 - x_0) = 0$ **then**
    $label := 6$;
  **else**
    $label := 9$;
**else if** $(y_1 - y_0) < 0$ **then**
  **if** $(x_1 - x_0) > 0$ **then**
    $label := 15$;
  **else if** $(x_1 - x_0) = 0$ **then**
    $label := 18$;
  **else**
    $label := 21$;
**else**
  **if** $(x_1 - x_0) > 0$ **then**
    $label := 0$;
  **else**
    $label := 12$;
**return** $label$; *//Directional label* $l(e,u)$

This simple algorithm considers each direction, and assigns a directional labelling based the position of a neighbouring vertex compared to the vertex being considered. Now we can determine

the orientation of curves in the string graph in such around a given vertex. Let us now define a *directional ordering* of edges relative to a vertex $u$.

**Definition 6** (Directional Label Ordering)**.** A directional label ordering of a vertex $u$ is a sequence of vertices of the neighbourhood of $u$, such that $l(e_1, u) < l(e_2, u) < \cdots < l(e_{|N(u)|}, u)$, where $N(u)$ denotes the neighbourhood of vertex $u$ and each $e_i$ is an edge connecting $u$ to each neighbour, $1 \leq i \leq |N(u)|$.

We can determine a directional label ordering of a vertex $u \in V(G)$ by applying Algorithm 1 to each edge connected to a neighbour relative to $u$, then storing each result as $(l(e, u) - 24)$ in an array called *labels*. As Algorithm 1 is being applied, store each neighbouring vertex in a parallel array called *neighbours*. Assume both arrays store eight elements, and empty elements in these integer arrays by default store the value zero. To obtain the ordering, sort *labels* in non-decreasing order and adjust the vertices stored in the parallel array *neighbours* accordingly. Upon sorting, add twenty four to all non-zero values in the array *labels* to obtain the correct directional labels. Since every neighbourhood for each vertex is at most eight elements and we are only considering one of these vertices, we can find a directional label ordering in $O(1)$ worst-case time using an augmented version of any $O(k \cdot \log(k))$ sorting algorithm. This process is described in Algorithm 2 as follows:

**Algorithm 2:** Finding the directional label ordering of a vertex $u \in V(G)$.

**Data**: Given an edge list $E$ (assumed to be a linked list) for the vertex $u$ from an Adjacency List (a lookup table of linked lists), and vertex $u \in V(G)$
**Result**: Two arrays, *neighbours* (directional label ordering of $u$) and *labels* (directional labels corresponding to each vertex in the directional label ordering).

```
int[ ]labels := new int[8]; //Directional label array
int[ ]neighbours := new int[8]; //Directional label ordering array
if (|E| > 0) then
    Link next := E.front();
    while next ≠ null do
        neighbour[i] := next.data;
        e := e(u, neighbours[i]);
        labels[i] := (Algorithm 1(e)) − 24;
        next := next.nextLink();
        i++;
    end
    Sort the array labels in non-decreasing order and simultaneously modify the corresponding
    indices of the parallel array neighbours;
    for m := 0 to 7 do
        if (labels[m] < 0) then
            labels[m] := labels[m] + 24;
        else
            labels[m] := −1;
            neighbours[m] := −1;
    end
else
    Fill arrays labels and neighbours with the value −1;
return (neighbours, labels); //Return the directional label ordering, and corresponding
directional labels
```

We can perform this procedure for each vertex. We chose to consider each vertex one-by-one so each string of a string graph can be constructed independently.

### I.4 Forming Intersections between Strings

In Section I.3, we introduced directional labels for each neighbour of a given vertex $u \in V(G)$. In our algorithm, we will construct two points for the entry and exit of a part of the string we will refer to as the *hook*. A hook of a vertex will intersect another string. It is important to note that there are at most eight hooks by the 8-Grid property. The entry and exit points of a hook correspond the coordinates found in Table 2 with the offset of the given drawing vertex for $((L-1) \bmod 24)$ and $((L+1) \bmod 24)$, where $L$ is a directional label. The order we insert points into each list corresponding to a string is vitally important. For this algorithm, if there is a directional label, we insert points as follows. We place an *entry point*, a *hook*, then an *exit point*. If a directional label doesn't exist, we can simply include dummy points in the list to maintain consistency for each string.

Next, we describe the procedure for constructing a hook. Recall in Section I.2, we built a lookup table for an intersection label $\gamma(u, v)$ (see Table 1), where $u, v \in V(G)$. The intersection label of a vertex and its neighbour corresponds to how we construct each hook. We have two different hooks corresponding to $\gamma(u, v) \in \{0, 1\}$.

**Definition 7** (Girth (of a hook in a string))**.** The girth of a hook $H$ in a string is the distance between the entry point, and exit point connected to $H$.

In the algorithm, we have a scaling factor $r$ that resizes the string graph relative to girth of each hook. We assume $r = 1$ as the algorithm progresses until the end of the algorithm. Each hook for each string in our construction has a girth of $(2 \cdot sin(\pi/24))$ as the algorithm progresses. If hooks with girth of unit size are desired upon termination, let $r = \frac{1}{2 \cdot sin(\pi/24)} \approx 3.83064878777$.

Let $h = 2 \cdot sin(pi/24)$, and $\epsilon < 1$. Consider two hook designs, these can be stored in a lookup table as seen in Table 3. In our sample example we chose $\epsilon = 1/2$ and two such hook designs will cause two strings to intersect twice, but one could create their own. A hook design should be bounded vertically between $[-h/2, h/2]$, and horizontally between $[-\epsilon/2, \epsilon/2]$.

Since we have a directional labelling for each neighbour, we can rotate each assembled hook. For a hook, compute the midpoint between two adjacent vertices $u$ and $v$ in $D$, call it $\mu = (\mu_x, \mu_y)$. Assuming we have a directional label $l(e, u)$, take each point in the table and rotate each about the origin using the lookup table $LabelPoint$. After computing this, include each rotated point $(x', y')$ in the point list as the hook from first to last as $(x' + \mu_x, y' + \mu_y)$.

## II. String Graph Drawing Algorithm from 8-GOGDs

Now, let us present our algorithm for drawing string graphs for graphs that have a drawing that satisfies the 8-Grid property.

**Algorithm 3:** String Graph Drawing Algorithm from 8-GOGDs.

**Data**: Given graph $G = (V, E)$, drawing $D$ of $G$ based on our assumptions, and a scaling factor $r \geq 1$

**Result**: A list of length $n = |V(G)|$ of linked lists that store points that can be used to construct the string graph.

*//Preliminary Construction Step*

$D :=$ *Multiply each (x,y) coordinate of each vertex in the drawing by four. Shift each vertex in the drawing by 1 in both horizontal (right) and vertical (up) directions*;

$StringGraph :=$ *Create an array of length n of linked lists*;

$\gamma :=$ *Construct lookup table for* $\gamma(u, v)$ *found in Table 1*;

$LabelPoint :=$ *Construct a lookup table corresponding to Table 2* ;

$colouring :=$ *Determine a colouring of the vertex set of G. colouring is an array of length n that labels each vertex with a label zero, one, or two* ;

*//Now, Construct each string*;

**for** $v := 0$ **to** $n$ **do**

  $(neighbours, labels) :=$ ($Algorithm$ $2(E[v], v)$);*//Obtain directional label ordering, and corresponding directional labels*

  $j := 0$; $i := 0$;

  **if** $(labels[0] \neq -1)$ **then**

    **while** $(labels[i] \neq -1) \wedge (i < 7)$ **do**

      **while** $(labels[i] - 1 > j)$ **do**

        $StringGraph[v].addFront(p(LabelPoint[j]))$; *//Dummy point*

        $j$++;

      **end**

      $L := labels[i]$;

      *//Add an entry point, the hook, then the exit point.*

      $p := (x(v) + LabelPoint[(L - 1) \bmod 24][0], y(v) + LabelPoint[(L - 1) \bmod 24][1])$;

      $StringGraph[v].addFront(p)$;

      Add *hook points* for hook $\gamma(v, neighbours[i])$ as described in Section I.4;//Construct hook for neighbour relative to $v$.

      $p := (x(v) + LabelPoint[(L + 1) \bmod 24][0], y(v) + LabelPoint[(L + 1) \bmod 24][1])$;

      $StringGraph[v].addFront(p)$;

      $j := labels[i] + 2$;

      $i$++;

    **end**

  **else**

    **while** $(j < 24)$ **do**

      $StringGraph[v].addFront(p(LabelPoint[j]))$; *//Dummy point*

      $j$++;

    **end**

**end**

**if** $(r > 1)$ **then**

  Shift every point down and left by one unit, then multiply each component by $r$. Upon doing this, shift each point back by $r$;

**return** *StringGraph*; *//The points that form the string graph*

| $m$ | $(x, y)$ *if* $\gamma(v, label[i]) = 0$ |
|---|---|
| 1 | $(-\epsilon/2, -h/2)$ |
| 2 | $(0, -h/2)$ |
| 3 | $(\epsilon/4, -h/4)$ |
| 4 | $(0, 0)$ |
| 5 | $(-\epsilon/2, 0)$ |
| 6 | $(-\epsilon/2, h/2)$ |

| $m$ | $(x, y)$ *if* $\gamma(v, label[i]) = 1$ |
|---|---|
| 1 | $(-\epsilon/2, -h/2)$ |
| 2 | $(0, -h/2)$ |
| 3 | $(\epsilon/4, -h/4)$ |
| 4 | $(\epsilon/4, 0)$ |
| 5 | $(0, h/4)$ |
| 6 | $(-\epsilon/2, h/4)$ |
| 7 | $(-\epsilon/2, h/2)$ |

**Table 3:** *Let $\epsilon = 1/2$. Example designs for each hook. After, each point is rotated about the origin using LabelPoint and shifted by the midpoint $\mu$.*

## III. Drawing curves with the points to form strings

Consider the points returned by the algorithm in each list contained in each position of the array. To produce the string graph, it is sufficient to connect the points for each list with line segments. That is, if the list $l_j$ contains points $p_{j,1}, p_{j,2}, \ldots, p_{j,|l_1|}$, then we will connect the points $p_{j,q}$ and $p_{j,q+1}$ with a line segment, where $1 \leq q < |l_j|$. Each list produces an open polygonal chain of line segments that induces a string in the string graph as we will prove later in Section 3.5. Also it is important to note that simple convex curves may be used to connect points if they are of sufficient length to not induce an unwanted intersection. An unwanted intersection is where a string intersects itself, or intersects improperly with another string.

## IV. Drawing Size

**Theorem 1.** *Upon termination, the size of the drawing of the string graph by the algorithm is of grid size $((4k + 2)r) \times ((4s + 2)r)$.*

*Proof.* Consider the drawing $D$ given as an input. Assume $D$ occupies a grid of size of $k \times s$, where $k$ is the length of the drawing, and $s$ is the width of the drawing. The algorithm begins by multiplying every point by four, then shifting $D$ by one unit to the right and one unit up These two steps alone cause the input drawing to have size $(4k + 1) \times (4s + 1)$. When we obtain the points before re-scaling, the furthest distance the rightmost or topmost points can be from a vertex point in $D$ is one. Upon constructing the points of the string graph, the new drawing $S$ of the string graph before we re-scale is of size $(4k + 2) \times (4s + 2)$. If the scaling factor equals 1, we return $S$. If $r > 1$, the drawing $S$ is shifted left by one and down by one, then every point is multiplied by $r$. After this, the points are all shifted up by $r$, and to the right by $r$. This means we have a drawing of a string graph on a grid of size $(r(4k + 2 - 1) + r) \times (r(4s + 2 - 1) + r)$. Therefore, we have a drawing of size $((4k + 2)r) \times ((4s + 2)r)$. □

## V. Correctness

**Theorem 2.** *When the algorithm terminates, the points returned in each list in the array form each string inducing a string graph representation of the given simple outerplanar graph $G$ with an 8-GOGD $D$.*

*Proof.* To begin, the algorithm resizes the drawing $D$ so for two vertices $v_1$ and $v_2$, $d(p(v_1), p(v_2)) \geq 4$, where $d$ denotes the Euclidean distance between two vertices in the drawing. Since the hooks can be customized, we assume the two hooks intersect at most twice. First, we must prove that each list of points when lines are drawn between each point $p_i$ and $p_j$ form a string, where $i < j$.

**Lemma 1.** In the $i^{th}$ list produced by the algorithm $p_{i,1}, p_{i,2}, \ldots, p_{i,|l_i|}$, the polygonal chain is simple (doesn't intersect itself), where the size of the list is $|l_i|$.

*Proof.* By the directional label ordering, we consider points in counter-clockwise order. Since each hook in the open polygonal chain are determined to be points perpendicular to the entry and exit points for a hook, so no two hooks on the same chain can intersect itself. So, the only place where two lines could form an intersection is where exit points meet with entry points between hooks in the directional label ordering.

By definition, each directional label for entry and exit points are assigned a position on a unit circle. To cross, the algorithm would formed a line by assigning a point that is clockwise on the unit circle. The algorithm assigns a unique ordering determined by the directional label ordering, so this is not possible. Therefore, the polygonal chain formed by connecting lines between each point in the $i^{th}$ list does not intersect itself. □

By Lemma 1, each list $j$ has points $p_{j,1}, p_{j,2}, \ldots, p_{j,|l_j|}$ that form a polygonal chain when lines are drawn between each point. This open polygonal chain forms a *string*. For every dummy point whereever an edge doesn't exist in $G$, entry point, and exit point, $p_z$ in a string corresponding to $v_i$, $d(p_z, p(v_i)) = 1$. We know $d(p(v_i), \mu) \geq 2$ for any two vertices in $D$ when resized, so the only place two strings could intersect are at the hooks of a string, where $\mu$ denotes the midpoint between any vertex in the drawing $D$ and any neighbour of $v_i$. In the algorithm, the hooks cross near the midpoint $\mu$ between two vertices in the drawing $D$ that correspond to the vertices in $G$. By the 8-Grid property, hooks intersect at complementary directions perpendicular to each hook's entry and exit points. The hooks correspond to the edges of $G$, all the strings together by definition form a string graph.

Therefore, upon termination the points returned in each list in the array form each string inducing a string graph representation of $G$.

□

## IV. Open Problems

In this section we present three open problems related to this topic.

### I. Drawing algorithm for outerplanar grid drawings that satisfy the 8-Grid property

Currently we have efficient algorithms for special cases of outerplanar graphs that have outerplanar grid drawings that satisfy the 8-Grid property. Through our exploration of the literature we have not been able to locate a general algorithm for any outerplanar graph $G$. This would be ideal to have for any outerplanar graph $G$ if a grid drawing $D$ that satisfies the 8-Grid property exists. If such an algorithm were to exist, our algorithm could be augmented to require only an outerplanar graph $G$, and the scaling factor $r$ as inputs.

### II. Is there a linear-time algorithm for drawing 1-STRING representations of 8-GOGDs?

Another open problem relates to planar graphs being in 1-STRING [3]. As stated previously, this means all plane drawings can be drawn as a string graph that has at most one crossing between strings. Is there a polynomial-time algorithm for 8-GOGDs that produces string graphs with at most one crossing between each string?

### III. Can be do better than $O(k|V(G)|)$ time for $2k$-GOGDs?

Lastly, consider if we loosen the property that edges must meet along the integer grid at integer coordinates and instead state more generally that we can have at most $2k$ neighbours, where $k \geq 2$. Consider the general case for if we uniformly spread the directions of edges where edges leave every

vertex at integer coordinates only at angles $1\pi/k, 2\pi/k, 3\pi/k, \ldots, 2(k-1)\pi/k, 2\pi$. We define this to be the $2k$-Grid property. It would seem easy to extend the results we obtained to gain an algorithm for $2k$-GOGDs, as our algorithm for 8-GOGDs is $k = 4$. We claim that if one were to fix any $k$, it would seem we could obtain an algorithm of our same running time. Say $k$ is variable, can we do better than $O(k|V(G)|)$ in the worst-case for this algorithmic problem?

## V. Conclusions

In this paper we have presented a linear-time algorithm for drawing string graphs from an outerplanar grid drawing that satisfies the 8-Grid property. The worst case time-complexity of this algorithm is $O(|V(G)|)$. This algorithm can produce a resizable drawing using different values for the *scaling* variable $r \geq 1$, and as we have proved produces a drawing of grid size $((4k+2)r) \times ((4s+2)r)$, where the input drawing is of size $r \times s$.

This algorithm offers additional advantages. The *hooks* that form intersections between each string can be a pair of customized sets of points. Another advantage of this algorithm is that this procedure is trivial to parallelize due to each string being able to be drawn in the plane without requiring knowledge from other strings during execution after the precomputation phase. There are two primary drawbacks of this algorithm. First, there are that some instances that may produce additional empty space that was found in the original drawing $D$. Also the string graph construction the algorithm considers is based heavily on the original drawing $D$.

With additional problems introduced, we encourage other researchers to explore graph drawing algorithms for drawing string graphs for other specific classes of graphs or drawing conventions.

## References

[1] A. Asinowski, E. Cohen, M. Golumbic, V. Limouzy, M. Lipshteyn, and M. Stern. Vertex intersection graphs of paths on a grid. *Journal of Graph Algorithms and Applications*, 16(2):129–150, 2012.

[2] J. Chalopin and D. Gonçalves. Every planar graph is the intersection graph of segments in the plane. In *Proceedings of the 41st annual ACM symposium on Theory of computing*, pages 631–638. ACM, 2009.

[3] J. Chalopin, D. Gonçalves, and P. Ochem. Planar graphs are in 1-STRING. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 609-617. Society for Industrial and Applied Mathematics, 2007.

[4] J. Chalopin, D. Gonçalves, and P. Ochem. Planar graphs have 1-string representations. *Discrete & Computational Geometry*, 43(3):626–647, 2010.

[5] G. Ehrlich, S. Even, and R. Tarjan. Intersection graphs of curves in the plane. *Journal of Combinatorial Theory, Series B*, 21(1):8–20, 1976.

[6] J. Fox and J. Pach. String graphs and incomparability graphs. *Advances in Mathematics*, 230(3):1381–1401, 2012.

[7] J. Kratochvíl. String graphs II: Recognizing string graphs is NP-hard. *Journal of Combinatorial Theory, Series B*, 52(1):67–78, 1991.

[8] J. Pach and G. Tóth. Recognizing string graphs is decidable. In *Graph Drawing*, pages 479–482. Springer, 2002.

[9] A. Proskurowski and M. Syslo. Efficient vertex- and edge-coloring of outerplanar graphs. *SIAM Journal on Algebraic Discrete Methods*, 7(1):131–136, 1986.

[10] M. Rahman and T. Nishizeki. Orthogonal drawings of plane graphs without bends. *Journal of Graph Algorithms and Applications*, 7(4):335–362, 2003.

[11] M. Rahman, T. Nishizeki, and S. Ghosh. Rectangular drawings of planar graphs. *Journal of Algorithms*, 50(1):62–78, 2004.

[12] M. Schaefer, E. Sedgwick, and D. Štefankovič. Recognizing string graphs in NP. *Journal of Computer and System Sciences*, 67(2):365–380, 2003.

[13] M. Schaefer and D. Stefankovic. Decidability of string graphs. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 241–246. ACM, 2001.

[14] E. Scheinerman. *Intersection classes and multiple intersection parameters of graphs*. PhD thesis, Princeton University, 1984.

[15] F. Sinden. Topology of thin film RC-circuits. *Bell System Tech. J*, 45:1639–1662, 1966.

[16] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.