IMPLEMENTING BIXBY AND WAGNER'S ALMOST-LINEAR-TIME ALGORITHM FOR GRAPH REALIZATION

A Thesis

Presented to the Honors Program of

Angelo State University

In Partial Fulfillment of the

Requirements for Highest University Honors

BACHELOR OF SCIENCE

by

NEIL WIEBE THIESSEN

MAY 2020

Major: Computer Science and Mathematics

IMPLEMENTING BIXBY AND WAGNER'S ALMOST-LINEAR-TIME ALGORITHM FOR GRAPH REALIZATION

by

NEIL WIEBE THIESSEN

APPROVED:

Dr. Simon Pfeil

Dr. Rob LeGrand

May 8, 2020

APPROVED:

Dr. Shirley M. Eoff
Director of the Honors Program

Dedication

Dedicated to my sister Margaret, without whose support
I could not have completed this thesis.

ACKNOWLEDGMENTS

First and foremost, I would like to thank Dr. Simon Pfeil for advising me on this project. His suggestion of the topic ultimately led me on the most difficult but fulfilling journey of my life. His continued support as I hit roadblock after roadblock in the implementation of the algorithm was invaluable, and his aid in understanding the source material was essential.

I would also like to thank Dr. Rob LeGrand for his patience and advice during the revising process. Additionally, the skills I gained by completing the honors components in his and Dr. Mark Motl's computer science courses were instrumental in successfully completing this project.

My sincerest gratitude goes to Dr. Shirley Eoff for her support not only during this project, but also during my four years as a member in the Honors Program. Her thoughtful guidance as the COVID-19 pandemic forced the suspension of in-person meetings made the situation much less stressful. To that end, her support and leadership of the Honors Program during this difficult time also deserves to be recognized.

Although not involved with this project directly, the influence of Dr. Trey Smith on my mathematical development cannot be overstated. His direction and teaching style have been greatly inspiring, and any mathematical clarity gained from this thesis should be attributed to his effective style of mathematical exposition, which I have tried to emulate.

On a more personal note, I would like to thank my family for their emotional and psychological support as this project tested my limits.

Finally, I would be remiss not to mention the friends that have supported me: Matt and Zach, with whom I have studied often; Cassandra, who has always been there for me; and my housemates Neal, Skyler, and Madde, whose support on a daily basis kept me going.

Abstract

The study of matroid theory unites topics from graph theory, linear algebra, combinatorial optimization, and many other fields. An important problem in matroid theory, called the graph realization problem, is recognizing when a given binary matroid is graphic. We present a thorough treatment of an almost-linear-time algorithm due to Bixby and Wagner that solves this problem. Along the way, we consider the important graph-theoretic concept of 2-isomorphism and the related hypopath problem. Finally, we present specific details on implementation of the algorithm, as well as explore several practical applications of the algorithm.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter I

Introduction

History

The theory of matroids was independently discovered by Takeo Nakasawa and Hassler Whitney in 1935. The work of Nakasawa has largely languished in obscurity, whereas that of Whitney sparked an entirely new field of mathematics. The name matroid is due to Whitney, who noticed the concept of independence as a common thread in his studies of linear algebra and graph theory. He then extracted those minimal conditions which gave rise to some structure of independent and dependent sets, and called the objects which arose from those conditions matroids. A crucial discovery is that not all of these objects necessarily arise from some matrix in linear algebra or graph in graph theory, implying that his idea of a matroid was not simply a consequence of previous theories, but an entirely new independent theory.

A matroid that arises from some graph is called *graphic*, and the graph that gives rise to the matroid is called a *realization* or *realizing graph*. The problem we consider is the identification and subsequent realization of graphic matroids. The algorithm we present is due to Robert Bixby and Donald Wagner, who published it as a technical report from Rice University in 1985 [3]. It first appeared as part of Wagner's Ph.D. dissertation [12], and it has also been published in the peer-reviewed journal *Mathematics of Operations Research* [2].

A key feature of this algorithm is its runtime, which is almost linear in the number of non-zero entries in the input matrix. The first polynomial-time algorithm was presented by Tutte in 1960 [11]. Various other polynomial-time algorithms have since

Journal of the American Mathematical Society been published, but of note is the almost-linear-time algorithm of Satoru Fujishige published in 1980 [7]. Fujishige's and Bixby and Wagner's algorithms were independently discovered around the same time, with preliminary results of the latter having been presented at a conference in 1979 despite its later publication date in 1985. Wagner's 1983 dissertation [12] was the first complete presentation of the algorithm, while the subsequent versions [3, 2] served to condense the work for journal publication and more widespread dissemination.

We note that work likely similar to ours has been done by Takahiro Ohto in [9], but we were unable to access this work, which we suspect has no English translation (the work is originally in Japanese). The Java implementation from Ohto's work is freely available on the internet at http://www.math.keio.ac.jp/~kakimura/GRP/, but this implementation was not useful or consulted for our implementation as it lacks comments and requires an outdated version of Java to run.

Preliminary Definitions and Notation

We will use the standard language and notation of set theory. We review some core definitions for completeness. A *set* S is an unordered collection of distinct elements. A family $\mathcal{F}$ is a set whose members are also sets. We make this distinction only for convenience, as all families are just as well described a sets. Given sets A and B, we use the following notation and terms: The *empty set* $\{\}$ is denoted $\emptyset$. The statement $a \in A$ asserts that a is an element of the set A. The notation $\{x \mid P(x)\}$ specifies a set such that each element x that satisfies the condition $P(x)$ is in the set. Set union is defined by $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$, while set intersection is defined by $A \cap B = \{x \mid x \in A \text{ and } x \in B\}$. The set $A - B$ is defined to be $\{x \mid x \in A \text{ and } x \notin B\}$. We say A is a *subset* of B, or $A \subseteq B$, if every element of A is also in B. In the case that we know $A \neq B$, we say A is a *proper subset* of B, or $A \subset B$. We define the *power set* $\mathcal{P}(A)$ of A as the family $\{X \mid X \subseteq A\}$. We denote as $|A|$ the cardinality of A.

Graphs

For our purposes, a *graph* G is an ordered pair (V, E), where V is a set of *vertices* and E is a multiset (set with repetition) of *edges*, where each edge $e \in E$ is defined by a multiset $\{u, v\}$ with $u, v \in V$ and $|e| = 2$. We call u and v the *end vertices* of e, and say that *e adjoins u* to v (or vice versa). We do not make the restriction that $u \neq v$. We say an edge and its two end vertices are *incident* to each other. We denote the vertex set of G as $V(G)$ and the edge set by $E(G)$. This definition allows for *parallel edges*, where two edges may adjoin the same two vertices, and *loops*, where an edge connects one vertex to itself. A graph with no parallel edges and no loops is a *simple graph*. The *degree* of a vertex is the number of edges incident to it.

A graph H is called a *subgraph* of another graph G if $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. The *edge-induced subgraph* $G[A]$ of a graph G with respect to an edge set $A \subset E$ is the graph with edge set A and vertex set $V = \{v \in V(G) \mid v \textit{ is an endpoint of some } e \in A\}$. A *path* is a nonempty sequence $v_0 e_1 v_1 \ldots v_{k-1} e_k v_k$ of distinct edges and vertices where each vertex or edge except v_0 is incident to its predecessor in the sequence. A *cycle* is a nonempty sequence satisfying the definition of a path, except with the restriction that $v_0 = v_k$. A loop is thus a minimal example of a cycle. A graph is *connected* if there is a path joining any two distinct vertices. A *connected component* of a graph G is a maximal connected subgraph of G. Evidently a connected graph has exactly one connected component. A graph whose edge and vertex sets are given by elements of a cycle is called a *polygon*, while a connected loopless graph on two vertices is a *bond*. A *forest* is a graph without cycles. A connected forest is a *tree*. Given

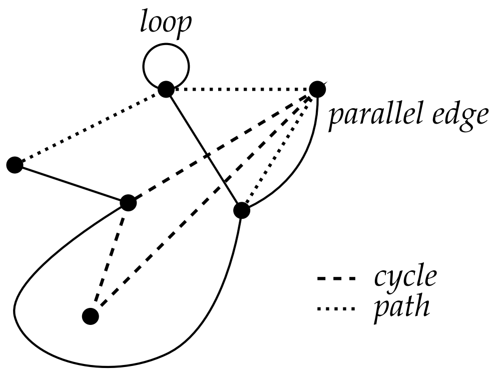


Figure 1 Example of a graph

a graph G, an important class of subgraphs of a connected graph G are the *spanning trees* of G, consisting of those subgraphs T of G that are trees and for which $V(T) = V(G)$.

A *directed graph*, or *digraph*, is a graph whose edges, now called arcs, are directed from one vertex, called the tail of the arc, to the other, called the head. The indegree and outdegree of a vertex are the number of arcs entering and number of arcs leaving a vertex, respectively. A *rooted directed tree* is a digraph where exactly one vertex, called the root, has indegree 0, and all other vertices have indegree 1. If p and c are the tail and head, respectively, of some arc of a rooted direct tree, then p is a parent of c and c is a child of p. Every vertex v in a rooted directed tree except the root has a unique parent, or predecessor, denoted $pred(v)$. Vertices of a rooted direct tree with

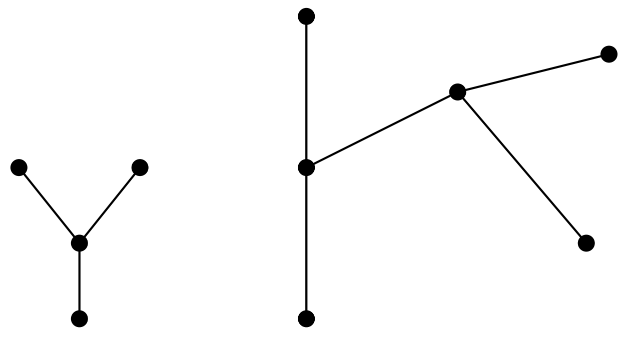


Figure 2 Example of a forest

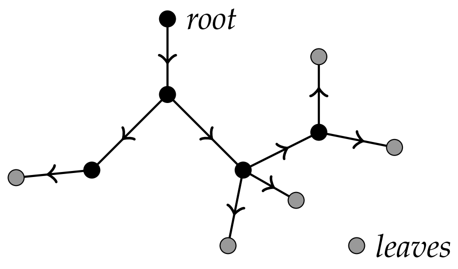


Figure 3 Example of a rooted direct tree

an outdegree of 0 are called *leaves*.

Matroids

There are many equivalent characterizations of matroids. We give three definitions relevant to our work. Proofs of their equivalence are routine and can be found in [10].

The first definition is purely matroid-theoretic, having no reference to graphs or matrices. It is the most general of the three we will present.

Definition 1 (Matroid). A *matroid* M is an ordered pair $(E, \mathcal{I})$ with E a finite set, $\mathcal{I} \subseteq \mathcal{P}(E)$ and that satisfies the following conditions:

1. $\emptyset \in \mathcal{I}$;
2. if $A \subseteq B$ and $B \in \mathcal{I}$, then $A \in \mathcal{I}$;
3. and if A and B are in $\mathcal{I}$ and $|A| < |B|$, then we can find some element e in $B - A$ such that $A \cup \{e\} \in \mathcal{I}$.

Elements of $\mathcal{I}$ are called the *independent sets* of M, and E is the *ground set* of M. Elements of $\mathcal{P}(E) - \mathcal{I}$ are called *dependent sets*.

Our next proposition derives a matroid from an arbitrary matrix. It could just as well be taken as the definition of a matroid, although this may restrict how many matroids we can construct.

First, we require several concepts from linear algebra. An $r \times c$ matrix M is a rectangular array of elements from a field $\mathbb{F}$ (typically $\mathbb{R}$ or $\mathbb{Z}$) with r rows and c columns. We label the matrix element in row i and column j of M as $m_{i,j}$. An r-vector $\mathbf{v}$ is an $r \times 1$ matrix. The r is often omitted when it is arbitrary or clear from context. A *scalar* is some element of the underlying field. Addition of two equal dimension matrices and multiplication of a scalar and a matrix are both performed element-wise. Two matrices are equal if all corresponding entries are equal. A collection of n vectors $\mathbf{v}_1, \ldots, \mathbf{v}_n$ is *linearly dependent* if there exist scalars $c_1, \ldots, c_n$, at least one of which is non-zero, such that

$$c_1\mathbf{v}_1 + \ldots + c_n\mathbf{v}_n = \mathbf{0}$$

where $\mathbf{0}$ denotes the appropriately sized vector with 0 in all entries. A collection of vectors that is not linearly dependent is *linearly independent*.

This suffices to present our first characterization of matroids based on matrices.

Proposition 1 (Matrices give rise to matroids)**.** *Given an $m \times n$ matrix A, the ordered pair $M[A] = (E, \mathcal{I})$ where*

1. *E is the set of column labels of A,*
2. *and $\mathcal{I}$ is the set of all subsets of E whose corresponding group of column vectors is linearly independent*

satisfies the conditions of definition 1.

There is some ambiguity in the previous proposition, as the notion of a matrix and linear independence requires an underlying field. Any field can be chosen, but, unless otherwise noted, we assume the field in question is $\mathbb{F}_2$, the finite field on two elements, with the usual elements 0 and 1. The field $\mathbb{F}_2$ is characterized by the following tables:

$+$	0	1
0	0	1
1	1	0

$\cdot$	0	1
0	0	0
1	0	1

Example 1. *Given the matrix*

$$A = \begin{array}{c} \begin{matrix} a & b & c & d & e \end{matrix} \\ \begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \end{array}$$

we can compute the corresponding matroid, $M[A]$. We see immediately that $E = \{a, b, c, d, e\}$. To find $\mathcal{I}$, we must consider all combinations of columns, and choose only those that are linearly independent. Upon inspection, we find that

$$\begin{aligned} \mathcal{I} = \Big\{ &\emptyset, \{a\}, \{b\}, \{c\}, \{d\}, \\ &\{a,b\}, \{a,c\}, \{a,d\}, \{b,c\}, \{b,d\}, \{c,d\}, \\ &\{a,b,c\}, \{a,c,d\}, \{b,c,d\} \Big\}. \end{aligned}$$

The $2^{|E|} - |\mathcal{I}| = 18$ dependent sets are any containing e, any with cardinality of 4 or higher, and the set $\{a, b, d\}$.

A minimal dependent set, or *circuit*, is a dependent set whose proper subsets are all independent. For our matroid, the set of circuits is

$$\mathcal{C} = \Big\{\{e\}, \{a, b, d\}\Big\}.$$

The set of circuits, along with a ground set E, provides enough information to uniquely identify our matroid; they are the smallest sets that will ruin the independence of a set containing them. Thus, we can find $\mathcal{I}$ from $\mathcal{C}$ by choosing all subsets of E not a superset of some element of $\mathcal{C}$. We make this remark to avoid listing another equivalent definition of a matroid, one in terms of circuits.

Another distinguished collection of sets that characterizes a matroid is its set of maximal independent sets, each of which is called a *basis* or *base* for the matroid. By maximal we mean that any proper superset of a basis B is dependent. For our matroid, we find that the set of bases is

$$\mathcal{B} = \Big\{\{a, b, c\}, \{a, c, d\}, \{b, c, d\}\Big\}.$$

The set of bases along with a ground set E also uniquely define a matroid. We can obtain $\mathcal{I}$ from a set of bases $\mathcal{B}$ by choosing all subsets of E that are also a subset of some element of $\mathcal{B}$.

Our final characterization of matroids is one in terms of graphs.

Proposition 2 (Graphs give rise to matroids)**.** *Let G be a graph with edge set E. Then the set of edge sets of cycles of G are the circuits of a matroid on E denoted $M(G)$. This matroid is called the cycle matroid of G.*

The name circuit is motivated by this connection to cycles in graphs. We also note that if our graph is connected, the set of spanning trees of G corresponds directly with the set of bases of $M(G)$.

Example 2. *Consider the graph G displayed in figure 4. The only cycles are* $\{e\}$ *and* $\{a, b, d\}$. *Since circuits (along with the same ground set) uniquely determine matroids, the matroid* $M(G)$ *is exactly* $M[A]$ *found in example 1.*

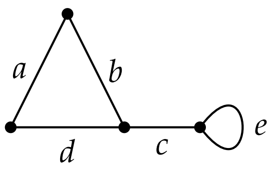


Figure 4 A graph with two cycles

A matroid M for which a graph G exists such that $M \cong M(G)$ is call *graphic*. A matroid M for which a matrix A over $\mathbb{F}_2$ exists such that $M \cong M[A]$ is called *binary*. The set of graphic matroids is a proper subset of the set of binary matroids [10].

Chapter II

The Graph Realization Problem

The Problem

The overarching problem we consider is this: given a binary matroid M, is there a graph G such that $M \cong M(G)$? If so, can we construct such a graph algorithmically? We call this the graph realization problem (GRP).

Recall that a key connection between a graph and its cycle matroid is the correspondence between the cycles and spanning trees of the graph and circuits and bases of the matroid. Seeing that we can construct a matroid given the cycles (or spanning trees) of a graph, a natural attempt at solving GRP would then be to attempt the reverse: given the circuits (or bases) of a matroid, create a graph with those circuits as cycles.

In order to facilitate the discussion going back and forth from graphs to matroids, we define several terms and an important matrix representation.

We require the following lemma:

Lemma 3. *Let B be a basis of a matroid M. If $e \in E(M) - B$, then $B \cup \{e\}$ contains a unique circuit $C(e, B)$.*

We will call this the *fundamental circuit of e with respect to B.* In general, a basis and all of its associated fundamental circuits does not give enough information to specify a matroid. However, it is enough if the matroid is binary [10]. Recalling that our statement of GRP above assumes a binary matroid as input, we will henceforth assume that all matroids under consideration are binary. Thus, we can represent them by giving a basis along with its fundamental circuits. In graph-theoretic terms, this is akin to specifying a family of graphs by giving a spanning tree and all of its cycles. As opposed to the case in a binary matroid, this information does not specify a single graph. It might specify multiple graphs, exactly one graph, or none at all.

Given a matroid M and some basis B, let $r = |B|$ and $c = |E(M) - B|$. Thus we can represent M (and a family of graphs G such that $M(G) = M$) by an $r \times c$ matrix N over $\mathbb{F}_2$. Label rows with elements of B and columns with elements of $E(M) - B$, and let each entry $n_{j,k}$ be defined by

$$n_{j,k} = \begin{cases} 1 & \text{if } j \in C(k, B) \\ 0 & \text{otherwise} \end{cases}$$

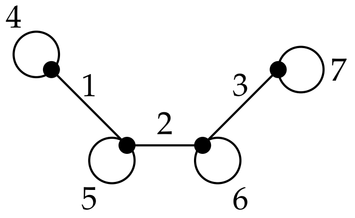
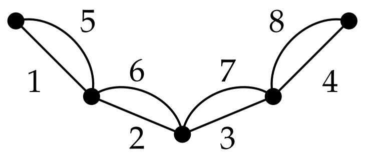
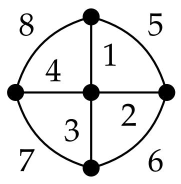
We call this a *fundamental matrix* for M. Our matroid from Example 1 can thus be represented by

$$N = \begin{array}{c|cc|} & b & e \\ a & 1 & 0 \\ d & 1 & 0 \\ c & 0 & 0 \end{array}$$

Notice the e column consists of all 0s. This is because e was a one element circuit. Notice also that the c row is also all 0. This corresponds with the fact that it is not a part of any circuit.

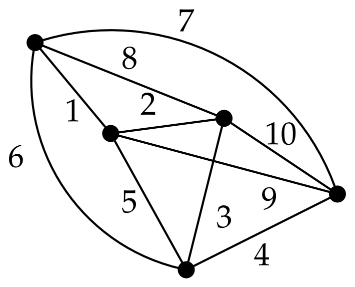
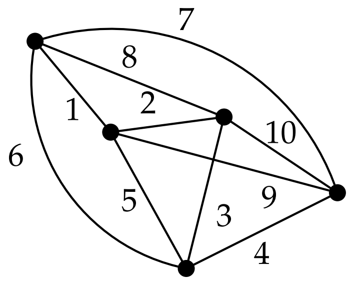
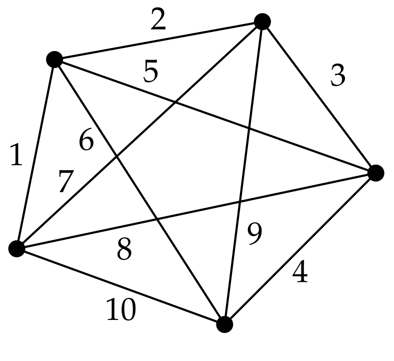
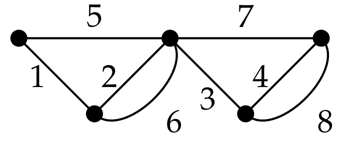
The graph realization problem can thus be restated in terms of this matrix. Given a fundamental matrix of a matroid, find a graph G (if it exists) such that the row labels form a spanning tree, and the column labels along with the row labels of any ones in that column form cycles of the graph. It is not required that these be the only cycles of G. If this graph exists, we call the matrix M *graphic*.

Several simple cases can be dealt with immediately. We consider specific examples, but each example could be easily extended to an arbitrary matrix of a similar form.

Fundamental Matrix	Realizing Graph
$M_1 = \begin{array}{c\|cccc} & 4 & 5 & 6 & 7 \\ \hline 1 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 \end{array}$	
$M_2 = \begin{array}{c\|cccc} & 5 & 6 & 7 & 8 \\ \hline 1 & 1 & 0 & 0 & 0 \\ 2 & 0 & 1 & 0 & 0 \\ 3 & 0 & 0 & 1 & 0 \\ 4 & 0 & 0 & 0 & 1 \end{array}$	
$M_3 = \begin{array}{c\|cccc} & 5 & 6 & 7 & 8 \\ \hline 1 & 1 & 0 & 0 & 1 \\ 2 & 1 & 1 & 0 & 0 \\ 3 & 0 & 1 & 1 & 0 \\ 4 & 0 & 0 & 1 & 1 \end{array}$	

Several things are clear from these examples. A zero rows corresponds with a tree edge not a part of any cycle, and a zero column corresponds with a loop. Zero rows and columns are thus trivial, and we can safely excise them from our matrix, attempt to realize the smaller matrix, and then add the appropriate elements to our graph. Henceforth, we will not consider fundamental matrices with any zero rows or columns.

The next table gives several more matrices and their realization, with the goal of developing an intuition regarding what scenarios might occur to make this problem difficult. Notice that the matrices M_4 and M_5 have the same realizing graph, which illustrates the nonuniqueness of this representation.

Fundamental Matrix	Realizing Graph
$M_4 = \begin{array}{c\|cccccc} & 5 & 6 & 7 & 8 & 9 & 10 \\ \hline 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 2 & 1 & 1 & 1 & 1 & 1 & 0 \\ 3 & 1 & 1 & 1 & 0 & 1 & 1 \\ 4 & 0 & 0 & 1 & 0 & 1 & 1 \end{array}$	
$M_5 = \begin{array}{c\|cccccc} & 1 & 3 & 4 & 6 & 8 & 9 \\ \hline 2 & 1 & 1 & 1 & 1 & 0 & 1 \\ 5 & 0 & 1 & 1 & 1 & 0 & 0 \\ 7 & 1 & 0 & 0 & 1 & 1 & 0 \\ 10 & 1 & 0 & 1 & 1 & 1 & 1 \end{array}$	
$M_6 = \begin{array}{c\|cccccc} & 5 & 6 & 7 & 8 & 9 & 10 \\ \hline 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 2 & 1 & 1 & 1 & 1 & 0 & 1 \\ 3 & 1 & 1 & 0 & 1 & 1 & 1 \\ 4 & 0 & 1 & 0 & 0 & 1 & 1 \end{array}$	
$M_7 = \begin{array}{c\|cccc} & 5 & 6 & 7 & 8 \\ \hline 1 & 1 & 0 & 0 & 0 \\ 2 & 1 & 1 & 0 & 0 \\ 3 & 0 & 0 & 1 & 0 \\ 4 & 0 & 0 & 1 & 1 \end{array}$	

Evidently, the graph becomes more difficult to realize as the number of "interactions" between the columns, or cycles, increases. By an interaction we mean a column sharing a one in the same row as another column. To capture the degree to which the columns of a matrix interact, we will construct a secondary graph that shows this information.

Let M be an $r \times c$ matrix over $\mathbb{F}_2$ with no all-zero rows or columns. Let R and C be the row and column indices, respectively. Define a graph G with vertex set $R \cup C$. For each $m_{j,k}$ that is nonzero, add an edge connecting j to k. The connected components of this graph now capture exactly which columns interact with one another. We call the columns and rows corresponding to each connected component of G a *block* of M. Further, we call M nonseparable if it has only one block.

Figures 1-3 contain the corresponding auxiliary graphs for the example matrices M_2, M_3, and M_7.

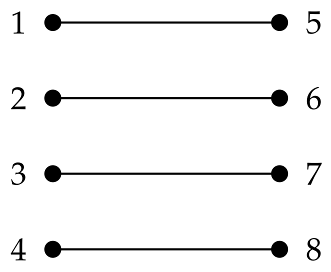


Figure 1 Auxiliary Graph for M_2

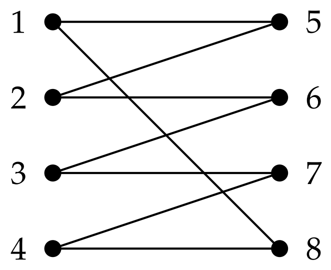


Figure 2 Auxiliary Graph for M_3

Because there is no interaction between blocks of a matrix, we can effectively re-

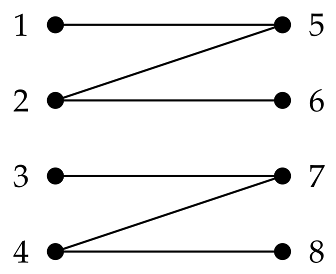


Figure 3 Auxiliary Graph for M_7

duce the problem into realizing each of the blocks of our matrix. Once we have realized each block, we can simply identify their realizing graphs together at one vertex to gain a connected realization for M.

We have reduced the problem somewhat, but have made little progress on how to realize a block other than to do so manually through trial and error. In the next section, we will introduce several graph-theoretic concepts that will shed light on that problem.

2-isomorphism

The key to effectively constructing graphs with a given set of cycles lies in the theory of *2-isomorphisms* described first by Whitney. Our treatment of the subject is derived from [10] and [3]. We require a preliminary definition. A k-separation of a connected graph is a partition $\{E_1, E_2\}$ of $E(G)$ such that

$$|E_1|, |E_2| \geq k$$

$$|V(G[E_1]) \cap V(G[E_2])| = k$$

That is, a partition of the edge set of a graph into two sets is a k-separation if each of the two sets has at least k edges, and the intersection of the node sets of the two subgraphs induced by the partition has exactly k vertices in it. Intuitively, a k-separation is thought of as a set of k vertices which would leave G disconnected if deleted. Further, we say a graph is n-connected if it has no k-separation for positive $k < n$. The notion of graph connectedness defined earlier corresponds with 1-connectedness as defined here. Graphs that have a 1-separation are *separable*, while graphs that do not are *nonseparable*. By definition, all 2-connected graphs are thus nonseparable, and all nonseparable graphs are connected.

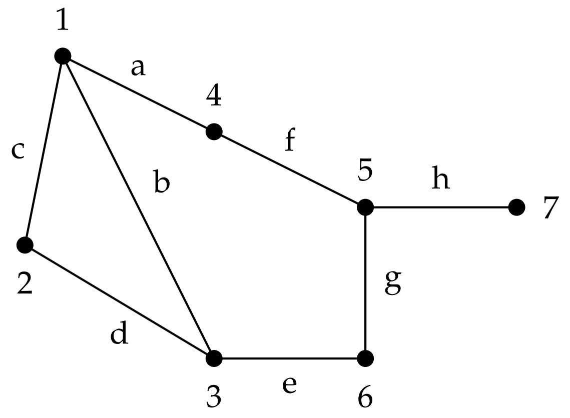


Figure 4 A graph with several 2-separations

The graph in Figure 4 has several 2-separations; for instance, the edge sets (along with their complements) $\{c, d\}$, $\{a, c, d\}$, and $\{a, c, d, f\}$ all define valid 2-separations. The graph is not nonseparable, as the edge set $\{h\}$, along with its complement, define a 1-separation.

It is important to keep the notions of nonseparable matrix and nonseparable graph distinct. They are related, but not equivalent, terms.

Suppose a graph G has a 2-separation $\{E_1, E_2\}$, with u and v the two vertices in $V(G[E_1]) \cap V(G[E_2])$. Then the process of switching u and v in every edge incident to them in $G[E_1]$ is called a *twisting*.

Figure 5 shows an example of a twisting, while Figure 6 shows a different repre-

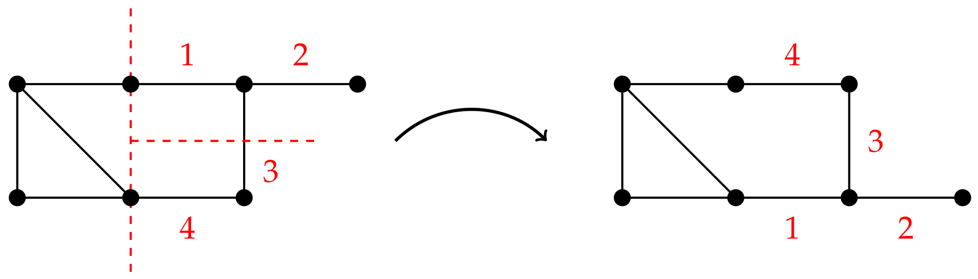


Figure 5 Illustration of a twisting

sentation of the same twisting.

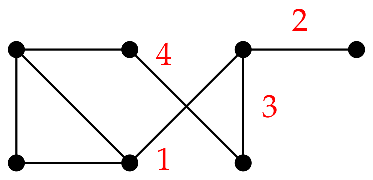


Figure 6 Alternative realization of the twisting in Figure 5

Intuitively, this involves cutting the graph at u and v, flipping the graph on one side of our cut about the axis perpendicular to the cut, and then rejoining both sides again. Two graphs G and H are *2-isomorphic* if H can be obtained from G by a sequence of twistings. The process of twisting a nonseparable graph has the important property that it will not modify the cycles of the underlying graph. A twisting might reorder the way certain cycles are put together, but it will not change the actual edge makeup of the cycle. Thus, it follows that graphs that are 2-isomorphic have the same cycles. The converse is also true, and this result is due to Whitney.

Theorem 4 (Whitney's 2-Isomorphism Theorem)**.** *Let G and H be two nonseparable graphs with the same edge set. Then G and H are 2-isomorphic if and only if they have the same cycles.*

This leads directly to a useful corollary connecting this theorem to matroids:

Corollary 5. *Let G and H be 2-isomorphic graphs. Then $M(G) \cong M(H)$.*

The converse is also true, but requires a slight extension of the definition of 2-isomorphism from nonseparable graphs to arbitrary, potentially disconnected, graphs.

With this new tool, we are now poised to develop a theoretical solution to the problem.

The Hypopath Problem

Recalling that we previously reduced the problem to considering each block of a matrix separately, let M be a nonseparable $r \times c$ matrix over $\mathbb{F}_2$. Let M_k be the matrix consisting of the first k columns of M with any zero rows deleted. We say M is *totally nonseparable* if each M_k is nonseparable for $1 \leq k \leq c$. Note that it is always possible to permute the columns of a nonseparable matrix into a totally nonseparable matrix since we know that every column interacts with at least one other column. To find such a totally nonseparable matrix M' for a given nonseparable matrix M, we can start with any column of M as M'_1. For each subsequent column, choose some column of M that has not yet been added and that interacts with some column already added. The result will be totally nonseparable. We will thus assume from now on that M is totally nonseparable.

We define two useful sets related to each column of a given fundamental matrix M. We denote the fundamental cycle encoded by the kth column by C_k. Each C_k will thus consist of the column label k along with the row labels of nonzero elements in the column. We define $P_k = C_k \cap \left(\bigcup_{j<k} C_j \right)$. Intuitively, each P_k defines which previous edges the kth column interacts with. Because M is totally nonseparable, we are guaranteed that $P_k \neq \emptyset$.

We also introduce the term *hypopath* to denote any set of edges of a graph G that is a path in some 2-isomorphic copy of G. A hypopath is thus a potential path of G, a set of edges that can be made a path without altering the cycles of G.

A theorem due to Löfgren provides us with an outline for a potential solution to our problem [8].

Theorem 6. *Let M be a totally nonseparable $r \times c$ matrix over $\mathbb{F}_2$. If M_k is realized by the graph G_k for some $1 \leq k < c$, then M_{k+1} is graphic if and only if P_{k+1} is a hypopath of G_k.*

Clearly M_1 is graphic, as we can always realize it as either a bond (if there is exactly one one in the first column) or a polygon (all other cases). As a general step, assume M_k is graphic with realization G_k. If P_{k+1} is not a hypopath, M is not graphic by Theorem 1.3.1. Supposing P_{k+1} is a hypopath of G_k, perform a series of twistings on G_k to turn P_{k+1} into a path of G'_k. Finally, add the edges in $C_{k+1} - P_{k+1}$ to G'_k as a path incident only to the end vertices of the path P_{k+1} to obtain a realization for M_{k+1}.

The only nontrivial step of this procedure is determining whether a set of edges P is a hypopath in a graph G and producing the 2-isomorphic graph G' with P as a path.

Given an arbitrary graph, this might seem quite daunting. There are several observations that make this problem seem more tractable.

1. Our graph is finite, so the number of 2-separations must also be finite.

2. The operation of twisting is its own inverse. That is, twisting across the same 2-separation twice has no effect.

3. We do not need to consider twistings about any vertices not incident to an edge of our desired path.

4. If we can recognize some subgraphs as being 3-connected, then we might be able to say immediately that our path is *not* a hypopath.

A naive solution to the hypopath problem could thus be achieved in time $O(|P| \cdot 2^n)$ where n is the number of 2-separations in our graph and P is the set of edges of our desired path. To do this, we simply inspect the graphs resulting from every combination of twistings that can be performed on the graph to see if our desired path is a hypopath. The base of two in the exponential derives from the fact that each 2-separation need only be considered in either its natural or twisted state. The factor of $|P|$ is the time taken to check whether P is a path for each combination of twistings. Adding a check for 3-connected subgraphs might make the algorithm faster in certain cases, but would not affect the worst case when the graph has no such subgraphs.

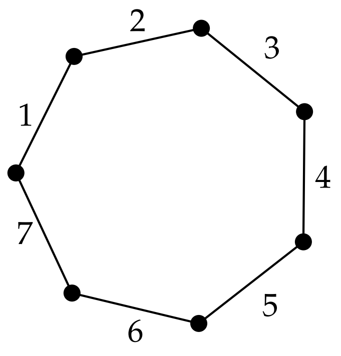


Figure 7 A polygon

The simplest example of a worst case graph for our algorithm is a polygon (a graph consisting of a single cycle), because every pair of two vertices with at least 2 edges between them define a potential twisting point. Suppose we wish to make $\{1, 4, 6\}$ in Figure 7 into a path. We could perform the sequence of twistings given in Figure 8 to achieve this. It is clear from this example that any proper subset of the edges of a polygon are a hypopath of the polygon.

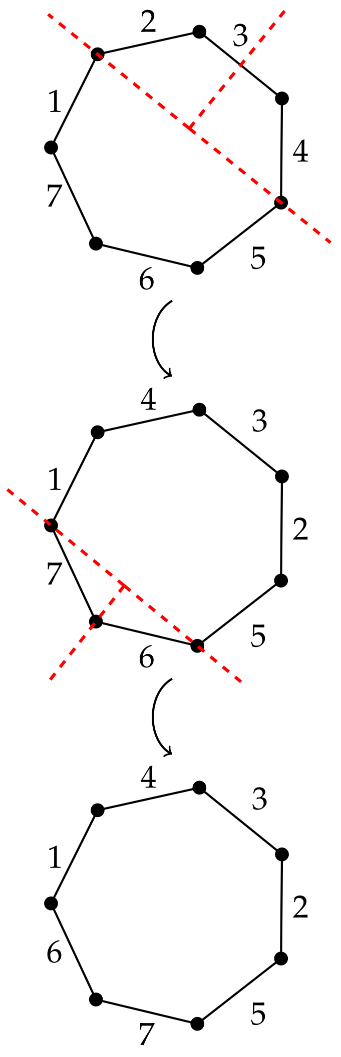


Figure 8 Reordering the edges of a polygon via a sequence of twistings

Corollary 7. *Let G be a polygon and $P \subset E(G)$ be a set of edges. Then P is a hypopath of G.*

Proof. This is a direct consequence of Whitney's 2-Isomorphism Theorem. In particular, the graph G' with the same edge set as G but with edge incidences updated to make P a path still has as its only cycle the entire edge set. So by Whitney's 2-Isomorphism Theorem, the graphs are 2-isomorphic and P is a hypopath of G $\square$

This allows us to bypass even considering the twistings of a polygon; we can simply relink the edges into whatever configuration we please.

Recognizing that we now have two classes of graph whose hypopaths are easy to recognize (polygons and 3-connected graphs), our next goal will be to describe a decomposition of graphs into smaller, easier-to-manage graphs.

Decomposition of Graphs

The theory described in this section was first developed by Tutte in [5], and refined for use in the context of this problem by Bixby and Wagner. We present here a complete overview of only the structure of a graph decomposition, and not the process for decomposing a graph to fit into such a structure. As the algorithm we are working towards is constructive, with each step building on the previous step, we will never have to actually decompose a graph.

An oriented decomposition is a triple $(\mathcal{T}, D, F)$ consisting of a rooted directed tree $\mathcal{T}$, a finite set of nonseparable graphs D, and a set of orientation functions F.

In particular, let D be a finite set of nonseparable graphs and let $\mathcal{T}$ be a digraph with vertex set D. Any two members of D can have at most one edge and no vertices in common. We note that this not possible under our definition of a graph, as an edge is determined by its end vertices. Thus, one edge could not have different end vertices in each of the graphs it appears in. We resolve this by saying that common edges are designated as such without actually being strictly equal. We also adopt the convention of calling any common edges by one name, and will also identify the common edge of G and H as the one element in the set $E(G) \cap E(H)$. These remarks are purely a technicality and do not have any real impact on our exposition. The root of $\mathcal{T}$ is chosen so that it has at least one edge not in any other member of D. Two vertices of $\mathcal{T}$ are connected by an arc if they share a common edge. The direction of each arc is uniquely determined by the characteristics of a rooted directed tree: the root has indegree zero, and every other vertex has indegree one. If there are multiple vertices that could be validly designated as the root, any one can be chosen. This choice will never have to be made in the algorithm by design.

Given two members G and H in D with $E(G) \cap E(H) = \{e\}$, we call the common edge e a *marker* or *virtual* edge. If $H = pred(G)$ (that is, H is the parent of G), then e is called a child marker of H and called the parent marker of G. In the latter context it is denoted $pm(G)$. We denote as $root(\mathcal{T})$ and $root(D)$ the root of $\mathcal{T}$. To make our pm function well-defined over D, we assign $pm(root(\mathcal{T}))$ to be any edge of the root that is not a marker edge (earlier we ensured the root must have such an edge).

For every member G of D except the root, we require an orientation function $f_G \in F$. Each of these functions is defined as follows: Let H be a non-root member of $\mathcal{T}$, and let K be its parent. Then H and K share the edge $e = pm(H)$. Let u_1 and u_2 be the end vertices of e in H and let v_1 and v_2 be the end vertices of e in K. Each orientation function is then defined as a bijection $f_H : \{u_1, u_2\} \to \{v_1, v_2\}$. Notice that we can have either $f_H = \{u_1 \to v_1, u_2 \to v_2\}$ or $f_H = \{u_1 \to v_2, u_2 \to v_1\}$. The operation of twisting over a 2-separation now corresponds to switching this orientation function between the two of its possible states.

The shared marker edges and orientation functions provide the information for how $\mathcal{T}$ can be merged into a single undirected graph made up of members of D modulo their marker edges. This merging will be described in detail later.

Finally, we specify several conditions to make this general oriented decomposition more manageable for our purposes:

1. Every member of D has at least three edges.
2. Every member of D is either a bond, polygon, or is 3-connected.
3. Only bond members of D have edges parallel to their parent marker.
4. Polygons cannot be adjacent to polygons in $\mathcal{T}$.
5. Bonds cannot be adjacent to bonds in $\mathcal{T}$.

An oriented decomposition satisfying these five conditions is called a *t-decomposition*. Instead of referring to a t-decomposition as a triple $(\mathcal{T}, D, F)$, we will refer to it as simply D. The existence of $\mathcal{T}$ and F, in their proper forms, are assumed. Figure 9 gives an example of a t-decomposition. We make one allowance in the case of 3-connected members: a member having only trivial 2-isomorphisms, as is the case when two vertices are connected by more than one edge, is still designated as 3-connected.

Given two members of D that are adjacent in $\mathcal{T}$, we can define a merge operation. To do so, let K be the parent, H be the child, and e be the marker edge they share in common. To merge them, identify the ends u_1, u_2 of e in H with the ends v_1, v_2 of e in K according to the orientation function $f_H \in F$. Finally, delete e from the resulting graph, which we denote $\mathit{merge}(H, K)$. We can then replace H and K in D with $\mathit{merge}(H, K)$, and update the arcs of $\mathcal{T}$ to gain a new decomposition. Note that

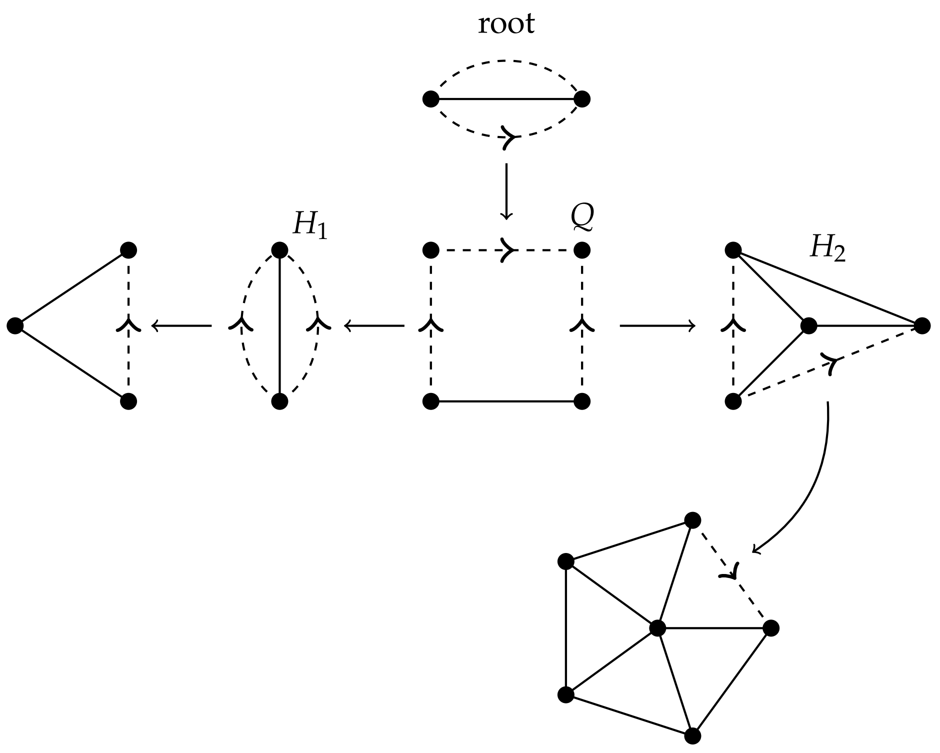


Figure 9 A t-decomposition

the class of t-decompositions is not closed under merging, and care must be taken to ensure the result remains one.

If we go through all pairs of elements and merge them (in any order), we obtain a merged graph denoted $merge(D)$ or $merge(\mathcal{T})$. Figure 10 gives an example of the result of this operation on the graph from Figure 9.

Given a member Q of D, we denote by $mst_F(Q)$ the result of recursively merging the children of Q according to the orientation functions in F. When the identity of F is clear, we will abbreviate $mst_F(Q)$ as $mst(Q)$. By this notation, $merge(D) = mst(root(D))$. We choose mst to stand for merged subtree. Given a member Q with children $H_1, \ldots, H_n$, we denote the set $\{mst_F(H_1), \ldots, mst_F(H_n)\}$ as the *complete children* of Q with respect to F.

Figures 11 and 12 show the complete children of Q in the t-decomposition given

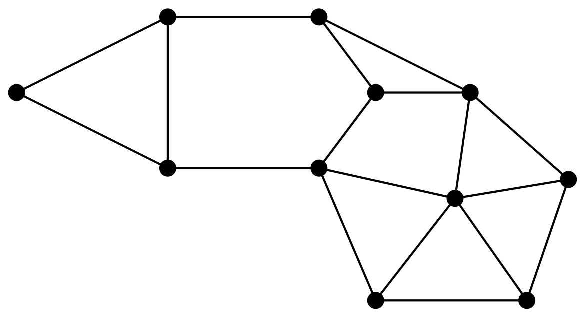


Figure 10 The merged graph of Figure 9

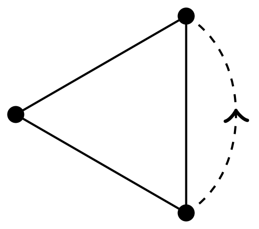


Figure 11 $mst(H_1)$

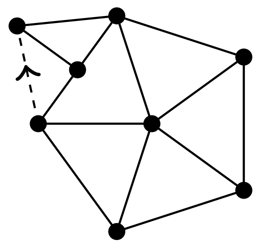


Figure 12 $mst(H_2)$

in Figure 9.

We are now prepared to recast our problem once more, this time in terms of t-decompositions.

Given a t-decomposition D and a set of edges $P \in E(m(D))$, find a t-decomposition D', if one exists, such that $merge(D')$ has the same cycles as (is 2-isomorphic to) $merge(D)$ and P is a path of $merge(D')$.

Within a t-decomposition D, we define a *restricted t-decomposition* $(\widehat{\mathcal{T}}, \hat{D}, \hat{F})$ with respect to a set of edges P to be the minimal decomposition that contains all members of D containing some edge in P. This restricted t-decomposition might contain members of D that do not meet P. This is because we require that the restriction remain a decomposition. The original orientation functions in F related to each member of $\hat{D}$ will make up $\hat{F}$, the orientation functions of our restricted decomposition. The arcs of $\mathcal{T}$ between members of $\hat{D}$ are used to form $\widehat{\mathcal{T}}$.

Classifying Members of a Decomposition with Respect to a Path

Given a graph G, an edge $m \in E(G)$, and a nonempty set of edges $X \subseteq E(G) - \{m\}$, we define the arrangement $A(G, X, m)$ as follows:

$$A(G,X,m) = \begin{cases} 1, & \text{if } X \cup \{m\} \text{ is a cycle;} \\ 2, & \text{if } X \text{ is a path with } m \text{ incident to one end-node and one internal node;} \\ 3, & \text{if } X \cup \{m\} \text{ is a path with } m \text{ an end-edge;} \\ 4, & \text{if } X \cup \{m\} \text{ is a path with } m \text{ not an end-edge;} \\ 5, & \text{otherwise.} \end{cases}$$

The distinguished edge m will usually be the parent marker of the graph G within a t-decomposition. The set X will usually be all the edges of a desired hypopath P that are also in $E(G)$. We define an auxiliary type function by

$$T(G, X, m) = \min\{A(G', X, m) \mid G' \textit{ is 2-isomorphic to } G\}$$

The function T thus gives us the best (where a lower number arrangement is considered better than a higher one) possible arrangement a given triple can satisfy under a sequence of twistings.

We say (H, X, m) is *good* if $T(H, X, m) < 5$ and $A(H, X, m) = T(H, X, m)$. The latter condition means that not only is it possible to twist the graph to get an arrangement less than five, but the graph, without any twistings, is actually in a best possible state. Where clear from context, we will often denote $T(H, X, m)$ simply as $T(H)$.

We provide some examples for clarity in Figures 13-17. These examples will be helpful as a canonical way to imagine a graph of each type. The set X consists in each case of the bold red edges, while m is the dashed edge.

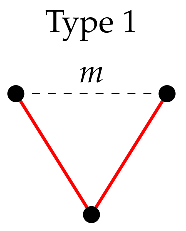


Figure 13 A representative graph of type 1

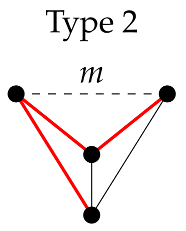


Figure 14 A representative graph of type 2

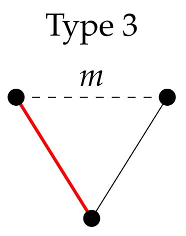


Figure 15 A representative graph of type 3

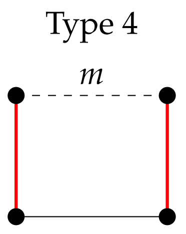


Figure 16 A representative graph of type 4

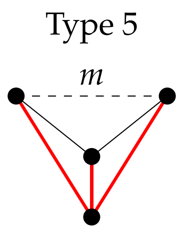


Figure 17 A representative graph of type 5

Because all members of a t-decomposition are either polygons, bonds, or 3-connected graphs, there are some observations about the possible types of our members:

- A bond B will always be type 1 as long as X is nonempty, X is a subset of $E(B) - \{m\}$, $|E(B) \cap X| = 1$, and $m \in E(B)$.
- A polygon can be in any arrangement except 2. The type of a polygon, however, can only be 1, 3, or 5 because any arrangement that would be type 4 can always be reordered to be type 3.

- A 3-connected graph G can be of any type. Moreover, $T(G) = A(G)$ because 3-connected graphs have no 2-isomorphisms.

The type function will later be evaluated on merged subtrees of a t-decomposition, so these remarks are only valuable when evaluating a leaf member of a t-decomposition. The point on polygons, for example, does not hold if we are considering a merged subtree whose root is a polygon. In a polygon with exactly one type-4 child (with all other children of type 1) whose non-marker edges are path edges, the type of the merged subtree would be 4. The implementation of $A(G, X, m)$ is fairly routine, and will not be covered.

Having introduced a theoretical framework to work in, we are ready to present Bixby and Wagner's algorithm.

Chapter III

Bixby and Wagner's almost-linear-time solution

An outline of the algorithm

The main algorithm is essentially a direct translation of the Löfgren procedure described earlier. It references two procedures, which perform the majority of the calculations. These will be treated in separate sections later.

1. HYPOPATH takes a t-decomposition D and a set of edges P and determines whether P is a hypopath of $merge(D)$. If it is, the procedure also updates polygon members and orientation functions of D such that P is a path of $merge(D)$. Note that $merge(D)$ is never actually calculated in this step.

2. UPDATE takes a t-decomposition, a set of edges C, and a set of edges P, and returns a new t-decomposition with C as a cycle. Note that only the edges $C - P$ are added in this step, as P is already guaranteed to be a path at this point.

Algorithm 1 Is-Graphic

```
Input: An $r \times c$ totally nonseparable matrix over $\mathbb{F}_2$.
Output: A graph that realizes $M$, or the conclusion that $M$ is not graphic.
 1: procedure IS-GRAPHIC($M$)
 2:     if $M_1$ has more than one 1 in it then
 3:         Prepend a new column with a single 1 to $M$, while ensuring that the result
    is still totally nonseparable
 4:     end if
 5:     Let $G_1$ be a bond with two edges that realizes $M_1$
 6:     if $c = 1$ then
 7:         return $G_1$
 8:     end if
 9:     Let $D_1$ be a new decomposition with one member $G_1$, and define $pm(G_1)$ to
    be the non-tree edge of $G_1$
10:     $j \leftarrow 2$
11:     repeat
12:         $P \leftarrow P_j$
13:         $D \leftarrow$ HYPOPATH($D_{j-1}$, P)
14:         if $P$ is not a hypopath then
15:             return False
16:         end if
17:         $C \leftarrow C_j$
18:         $D_j \leftarrow$ UPDATE(D, C, P)
19:         $j \leftarrow j + 1$
20:     until j = c
21:     return $merge(D_{j-1})$
22: end procedure
```

The HYPOPATH Procedure

HYPOPATH takes a t-decomposition D and a set of edges P and determines whether P is a hypopath of $merge(D)$. If it is, the procedure also updates polygon members and orientation functions of D such that P is a path of $merge(D)$. Note that $merge(D)$ is never actually calculated in this step.

The theory of typing with respect to certain edges is fundamental to this procedure. To start, given a t-decomposition D and a set of edges $P \in E(merge(D))$, we calculate the restricted t-decomposition $\hat{D}$ with respect to P by adding to $\hat{D}$ all members of D containing some edge of P and then adding all members from D needed to make $\hat{D}$ a valid t-decomposition. This construction is detailed in Algorithm 11.

Our aim is then, by starting at the leaves of $\hat{D}$, to find the type of every complete child of $root(\hat{D})$. This is achieved by the procedure TYPING. Once the types (and updated orientations) of all complete children of the root are known, it is straightforward to check whether the path is a hypopath.

We make several remarks about the characteristics of certain types when viewed in the context of a t-decomposition. Correctness of these remarks can be verified by the reader, or found in [3].

Given a member Q of a restricted t-decomposition $\hat{D}$ with respect to P, suppose we are determining the type of $K = mst(Q)$, the merged subtree rooted at Q. We assume that the types of the complete children $H_1, \ldots, H_n$ are known. A node incident to an edge in X is called an end-node of X if it is adjacent to exactly one edge of X and not incident to any parent markers.

If $T(H_i) = 1$, then H_i contains no end-nodes of X. If $T(H_i) = 2$ or $T(H_i) = 3$, then H_i contains exactly one end-node of X. If $T(H_i) = 4$, then H_i contains exactly two end-nodes of X. The importance of these end-nodes is that, after any polygons involved have been relinked, we can be sure that an end-node of X is essentially fixed (*i.e.*, it must be an end-node of our hypopath).

Because a path must have exactly two end-nodes, there are some cases here that can be easily discarded (*i.e.*, X cannot be a hypopath).

If more than two of the H_i are type 2 or 3, or more than one is type 4, X cannot be a hypopath. Moreover, if one complete child is type 4, all other complete children of Q must be of type 1 for X to be a possible hypopath. The type of K can be at least 4 in this case. The orientation of the type-4 complete child does not matter.

If exactly two complete children are type 2 or 3, then all other complete children must be of type 1. The type of K can be at least 4 in this case. There are four possible combinations of orientations to consider (each type-2 or -3 complete child has 2 orientations) in determining the type of K.

If exactly one complete child is type 2 or 3 and all others are type 1, the type of K can be 2, 3, 4, or 5. To determine the minimum type of K, two orientations for the type-2 or -3 child must be examined.

If all complete children are type 1, K can be of any type. The orientations of any type-1 children does not matter.

We have been referencing the type function T, as opposed to the arrangement function A. This means we must consider all 2-isomorphic copies of the graph we are trying to type. However, because we are inside a t-decomposition, only polygon members have nontrivial 2-isomorphisms. Polygons will be dealt with by relinking, or reordering, their edges.

In order to define a procedure to relink polygons, we introduce several new sets. Let Q be a polygon and $H_1, \ldots H_n$ be the complete children of Q. Define $m_i = pm(H_i)$, $X = P \cap E(K)$ (the path edges within the subtree we are considering), and $W_Q = (P \cap E(Q)) \cup \{m_1, \ldots, m_n\}$. We let $Z \subseteq W_Q$ be the set of child markers of Q corresponding to any non-type-1 children. By our previous remarks, we may assume $|Z| \leq 2$. Depending on the value of $|Z|$, we assume $Z = \emptyset$, $Z = \{m_1\}$, or $Z = \{m_1, m_2\}$. This serves only to name possible members of Z for reference in the algorithm.

The considerations are slightly different depending on whether the polygon we are relinking is the root of a restricted t-decomposition or not, so we give two slightly different procedures.

Algorithm 2 Relink-Non-Root

```
Input: $Q$ is a polygon, $W_Q \subseteq E(Q)$, and $Z \subseteq W_Q$ with $|Z| \leq 2$.
Output: A properly updated $Q$.
 1: procedure RELINK-NON-ROOT($Q, W_Q, Z$)
 2:     $m \leftarrow pm(Q)$
 3:     Reorder the edges of $Q$ so that $W_Q - Z$ is a path with one end of $W_Q - Z$ incident to $m$
 4:     if $m_1$ is defined then
 5:         Ensure that $m_1$ is incident to the other end of $W_Q - Z$
 6:     end if
 7:     if $m_2$ is defined then
 8:         Ensure that $m_2$ is incident to the node of $m$ not incident to $W_Q - Z$
 9:     end if
10:     return $Q$
11: end procedure
```

In order to traverse the tree of a t-decomposition D, we define a depth partition $\pi = (\pi_0, \ldots, \pi_s)$ of D where $Q \in \pi_i$ if the unique path from Q to $root(D)$ has i arcs. This means $\pi_0 = \{root(D)\}$ and π_s consists of all leaves of our decomposition that are of equal and maximum distance away from the root. It is not necessarily the case that all leaves will be in π_s, only that all members of π_s are leaves. We are now prepared to state TYPING. The procedures RULE1-RULE3 will be defined later.

Algorithm 3 Typing

Input: A restricted t-decomposition $\hat{D}$ with respect to a path $P \subseteq merge(\hat{D})$, and a depth partition π of $\hat{D}$

Output: An updated $\hat{D}$ such that each complete child of $root(\hat{D})$ is good and has a known type, or the conclusion that P is not a hypopath of $merge(\hat{D})$.

```
1:  procedure TYPING($\hat{D}, P, \pi$)
2:      Let $s$ be the index of the deepest level in $\pi$
3:      for all $Q \in \pi_s$ do
4:          $X \leftarrow P \cap E(Q)$
5:          if $Q$ is a polygon then
6:              $Q \leftarrow$ RELINK-NON-ROOT($Q, X, \emptyset$)
7:          end if
8:          $Q.type \leftarrow A(Q, X, pm(Q))$
9:          RULE1(Q)
10:         RULE2(Q)
11:     end for
12:     $i \leftarrow s - 1$
13:     while $i > 0$ do
14:         for all $Q \in \pi_i$ do
15:             Let $H_1, \ldots, H_n$ be the representative graphs of the appropriate types
    for each of the children of $Q$
16:             if $Q$ is a polygon then
17:                 $Q \leftarrow$ RELINK-NON-ROOT($Q, W_Q, Z$)
18:             end if
19:             if more than two of the $H_i$ are not type 1 then
20:                 return False
21:             end if
22:             Find, if possible, orientations $F'$ such that $mst_{F'}(Q)$ is good.
23:             Let $T$ be the minimum type achievable for $mst(Q)$.
24:             Update $\hat{D}$ with the orientations that achieved the minimum type.
25:             if T = 5 then
26:                 return False
27:             end if
28:             $Q.type \leftarrow T$
29:             RULE1(Q)
30:             RULE2(Q)
31:             RULE3(Q)
32:         end for
33:         $i \leftarrow i - 1$
34:     end while
35:     return $\hat{D}$
36: end procedure
```

After TYPING has been applied, we can effectively know what $merge(\hat{D})$ will look like on a macro level, without having had to compute any actual merged graphs. At this point, we do not care about $T(root(\hat{D}))$, but rather wish to make the final determination of whether or not P can be made into a path. For example, a type of 5 is preferable to a type of 4, because a type of 4 would mean our path is split by a marker edge which is no longer acceptable. Many acceptable arrangements for our path would be classified as a type of 5 because we do not require that our path have any vertices incident to the parent marker of $root(\hat{D})$. With this in mind, we will not calculate the type of $root(\hat{D})$, and so we require a slightly modified relinking procedure for the case that $root(\hat{D})$ is a polygon.

Algorithm 4 Relink-Root

```
Input: $Q$ is a polygon, $W_Q \subseteq E(Q)$, and $Z \subseteq W_Q$ with $|Z| \leq 2$.
Output: A properly updated $Q$.
 1: procedure RELINK-ROOT($Q, W_Q, Z$)
 2:     Reorder the edges of $Q$ so that $W_Q - Z$ is a path.
 3:     if $m_1$ is defined then
 4:         Ensure that $m_1$ is incident to the one end of $W_Q - Z$
 5:     end if
 6:     if $m_2$ is defined then
 7:         Ensure that $m_2$ is incident to the other end of $W_Q - Z$
 8:     end if
 9:     return $Q$
10: end procedure
```

We are now prepared to state HYPOPATH.

Algorithm 5 HYPOPATH

```
Input: A $t$-decomposition $D$ and a set of edges $P \in E(merge(D))$
Output: An updated $t$-decomposition $D$ such that $P$ is a path of $merge(D)$, or the
    conclusion that $P$ is not a hypopath of $merge(D)$
 1: procedure HYPOPATH($D, P$)
 2:     Compute the restricted $t$-decomposition $\hat{D}$ of $D$ with respect to $P$.
 3:     if $|\hat{D}| > 1$ then
 4:         Compute the depth partition $\pi$ of $\hat{D}$
 5:         $\hat{D} \leftarrow$ TYPING($\hat{D}, P, \pi$)
 6:         if $\hat{D} =$ False then
 7:             return False
 8:         end if
 9:         $Q \leftarrow root(\hat{D})$
10:     else
11:         $Q \leftarrow root(\hat{D})$
12:     end if
13:     Let $H_1, \ldots, H_n$ be the representative graphs of the appropriate types for each
    of the children of $Q$
14:     if more than two of the $H_i$ are not type 1 then
15:         return False
16:     end if
17:     if $Q$ is a polygon then
18:         $Q \leftarrow$ RELINK-ROOT($Q, W_Q, Z$)
19:     end if
20:     Find, if possible, orientations $F'$ such that $P$ is a path of $mst_{F'}(Q)$.
21:     if $P$ cannot be made a path then
22:         return False
23:     end if
24:     RULE4(Q)
25:     RULE5(Q)
26:     Update $D$ with the orientations that make $P$ a path.
27:     return $D$
28: end procedure
```

We note that if $|\hat{D}| = 1$, line 20 reduces to checking if P is a path of Q because there are no complete children to consider.

The Update Procedure

In implementing the Is-Graphic procedure, there are two basic directions we could go after determining that a given path is a hypopath. The first option is to compute the merged graph $merge(D)$ and add the remaining edges of our cycle so that they form a cycle together with P (which is by now a path of $merge(D)$). In order to run Hypopath again, however, we require a t-decomposition of $merge(D)$ with these new edges added. This would require an algorithm to compute a t-decomposition given an arbitrary graph. The other, more efficient, option is to update D with the extra edges of our cycle and proceed with another call to Hypopath from there.

We will take the latter approach, but note that updating a t-decomposition, and having it remain a t-decomposition, must be done carefully.

Assume once more that D is a t-decomposition, and that P is a path of $merge(D)$.

The general update procedure can be described succinctly. Let K_1 and K_2 be the members of D containing the end-nodes of P, and let $u_1 \in V(K_1)$ and $u_2 \in V(K_2)$ be the end-nodes of P. If $K_1 = K_2$ (that is, all of our path resides in one member), the update changes D minimally. However, in the case that $K_1 \neq K_2$, more substantial changes must occur. Let R be the unique path in $\mathcal{T}$ between K_1 and K_2. Our goal is to merge R into one 3-connected member of D, essentially putting us back in the case that $K_1 = K_2$. To do this succesfully, we must first break apart any polygons in R such that their inclusion in $merge(R)$ will not cause $merge(R)$ to have a 2-separation (and thus not be 3-connected). This procedure of breaking up a polygon will be called splitting and is described later.

Before we can consider typing, we must first address how K_1, K_2, u_1, and u_2 are found. The following five procedures, previously referenced in Typing and Hypopath, achieve this. Note that the value of K_1, K_2, u_1, and u_2 are not returned by any of the rules. Rather, we assume for simplicity that these variables are available globally and reset after each call to UPDATE.

Algorithm 6 Rules for selecting K_1, K_2, u_1, and u_2

```
 1: procedure RULE1(H)
 2:     if $H$ has no children or all children of $H$ are type 1 then
 3:         if $H.type = 2$ or 3 then
 4:             if $K_1$ has not been defined then
 5:                 $K_1 \leftarrow H$
 6:                 Let $u_1$ be the end-node of $P$ not incident to $pm(H)$
 7:             else
 8:                 $K_2 \leftarrow H$
 9:                 Let $u_2$ be the end-node of $P$ not incident to $pm(H)$
10:             end if
11:         end if
12:     end if
13: end procedure
14: procedure RULE2(H)
15:     if $H$ has no children or all children of $H$ are type 1 then
16:         if $H.type = 4$ then
17:             $K_1, K_2 \leftarrow H$
18:             Let $u_1$ and $u_2$ be the end-nodes of $P$ not incident to $pm(H)$
19:         end if
20:     end if
21: end procedure
22: procedure RULE3(H)
23:     if $H$ has one type 2 or 3 child, and all others are type 1 then
24:         if $H.type = 4$ then
25:             Let $D$ be the unique path between $K_1$ and $H$
26:             Let $K_2$ be the vertex of $D$ closest to $K_1$ that contains the same end-node
    of $P$ as $H$
27:             Let $u_2$ be the end-node of $P$ in $K_2$
28:         end if
29:     end if
30: end procedure
```

Algorithm 7 Rules for selecting K_1, K_2, u_1, and u_2, continued

```
31: procedure RULE4(H)
32:     if $K_1$ has not been defined then
33:         $K_1, K_2 \leftarrow H$
34:         Let $u_1, u_2$ be the end-nodes of $P$ not incident to any marker edges in $H$
35:     else
36:         Let $D$ be the unique path between $K_1$ and $H$
37:         Let $K_2$ be the vertex of $D$ closest to $K_1$ that contains the same end-node of
    $P$ as $H$
38:         Let $u_2$ be the end-node of $P$ in $K_2$
39:     end if
40: end procedure
41: procedure RULE5(H)
42:     if $K_1$ and $K_2$ have been defined and $K_1 = K_2$ then
43:         if the ends of $P$ are the ends of $pm(K_1)$ and $K_1$ is not a bond then
44:             $K_1, K_2 \leftarrow p(K_1)$
45:         else if the ends of $P$ are the ends of another marker $m_i$ of $K_1$ and $K_1$ is a
    polygon then
46:             if the marker edge $m_i$ is also in a bond $B$ then
47:                 $K_1, K_2 \leftarrow B$
48:             end if
49:         end if
50:     end if
51: end procedure
```

We note that although we have defined five types and five rules for determining K_1, K_2, there is no connection between the types and the rules. RULE1 and RULE2 handle the simple case where the end-node(s) are within (not adjacent to a parent marker of) a single member. In the case that we discover end-node(s) of our path to be incident to a parent marker, RULE3, RULE4, and RULE5 aim to minimize the length of R by picking the end-node in the member closest to K_1. Essentially, we wish to minimize the amount of our t-decomposition that we will have to modify during UPDATE. Picking these members in this way is also key to ensuring that $merge(R)$ will be 3-connected after taking the appropriate steps in UPDATE.

Now that we have identified which members contain the end-nodes of P, we must describe a method for splitting up a polygon that would be "caught up" by our cycle.

Algorithm 8 Splitting a polygon

```
1: procedure SPLIT($D^*, Q, L, R$)
Input: $Q$ is a polygon and $L \subset E(Q)$
2:     if $|E(L)| > 2$ then
3:         Let $f'$ be a new edge
4:         Let $P_1, P_2$ be two polygons formed by adding $f'$ to the paths $L$ and $S - L$, respectively
5:         Replace $S$ in $D^*$ with $P_1$ and $P_2$
6:         Replace $S$ in $R$ by the polygon formed from $S - L$
7:     end if
8: end procedure
```

Finally, we conclude with the actual UPDATE procedure. Details on implementation will be given in the next chapter.

Algorithm 9 Updating a t-decomposition

```
Input: A $t$-decomposition $D$, a path $P$ of $merge(D)$, and desired cycle $C$ such that
    $C \cap E(merge(D)) = P$ and $C - P \neq \emptyset$
Output: A $t$-decomposition of the graph obtained from $merge(D)$ by adding the
    edges of $C - P$ so that $C$ is a cycle and $C - P$ is incident to $merge(D)$ at exactly
    two nodes.
 1: procedure UPDATE($D, P, C$)
 2:     $D^* \leftarrow D$
 3:     if $|C - P| = 1$ then
 4:         $\{f\} \leftarrow C - P$
 5:     else
 6:         Let $f$ be a new edge
 7:         Form a polygon with edge-set $\{f\} \cup (C - P)$ and add this polygon to $D^*$
 8:     end if
 9:     if $K_1 = K_2$ then
10:         if $K_1$ is not a polygon then
11:             Join $u_1$ and $u_2$ by $f$ in $K_1$
12:         else if $K_1$ is a polygon and $u_1, u_2$ are adjacent then
13:             Let $f'$ be the edge joining $u_1$ and $u_2$ in $K_1$
14:             Let $f''$ be a new edge
15:             Replace $f'$ with $f''$ in $K_1$
16:             Add a bond with edge set $\{f, f', f''\}$ to $D^*$
17:         else                              ▷ $K_1$ is a polygon with $u_1, u_2$ not adjacent
18:             Let $L_1$ and $L_2$ be the two paths joining $u_1$ and $u_2$ in $K_1$
19:             Let $f_1, f_2$ be new edges.
20:             Let $P_1$ be a new polygon created by joining the ends of $L_1$ with $f_1$
21:             Let $P_2$ be a new polygon created by joining the ends of $L_2$ with $f_3$
22:             Replace $K_1$ in $D^*$ with $P_1$ and $P_2$
23:             Add a bond with edge-set $\{f, f_1, f_2\}$ to $D^*$
24:         end if
25:     else                                                      ▷ $K_1 \neq K_2$
26:         Let $R$ be the unique path $K_1 = J_1, \ldots, J_{s+1} = K_2$ joining $K_1$ and $K_2$ in $\mathcal{T}$.
27:         Let $\{m_j\} = E(J_j) \cap E(J_{j+1})$ for $1 \leq j \leq s$
28:         for $j \leftarrow 1, s$ do
29:             if ($J_j$ is 3-connected and $\{m_{j-1}, m_j\}$ is a cycle of $J_j$) or ($J_j$ is a bond on at
    least 4 edges and $p(J_j) \notin R$) then
30:                 Let $f'$ be a new edge
```

Algorithm 10 Updating a t-decomposition, continued

```
31:             Let $J_j$ be $J_j$ with $\{m_{j-1}, m_j\}$ deleted and $f'$ added in its place
32:             Let $B$ be a bond with edge-set $\{m_{j-1}, m_j\}$
33:             Replace $J_j$ in $R$ with $B$
34:             Replace $J_j$ in $D^*$ with $J_j'$ and $B$
35:         end if
36:     end for
37:     if $K_1$ is a polygon then
38:         Let $L_1, L_2$ be the two paths joining $m_1$ and $u_1$
39:         SPLIT($D^*, K_1, L_1, R$)
40:         SPLIT($D^*, K_1, L_2, R$)
41:     end if
42:     if $K_2$ is a polygon then
43:         Let $L_1, L_2$ be the two paths joining $m_s$ and $u_2$
44:         SPLIT($D^*, K_2, L_1, R$)
45:         SPLIT($D^*, K_2, L_2, R$)
46:     end if
47:     for all $J_j \in \{J_2, \ldots, J_s\}$ do
48:         if $J_j$ is a polygon then
49:             Let $L_1, L_2$ be the two components of $J_j - \{m_{j-1}, m_j\}$
50:             SPLIT($D^*, J_j, L_1, R$)
51:             SPLIT($D^*, J_j, L_2, R$)
52:         end if
53:     end for
54:     $G \leftarrow merge(R)$
55:     Join $u_1$ and $u_2$ in $G$ with $f$
56:     Delete all nodes of $R$ from $D^*$ and add $G$
57: end if
58: return $D^*$
59: end procedure
```

Chapter IV

Discussion

Data Structures

As with the previously presented algorithms, we will also reiterate the data structures required for their implementation. We hope in doing so to further illuminate the original paper [3].

Each of the columns and rows of the input matrix M is assumed to have a unique name. A natural choice of names is the integers from 1 to $r + c$. These names will be used as the edge names of the final graph. Marker edges are also named, and each marker edge (except $pm(root(D))$) as presented in the algorithm will have two names, one for each of the two members it appears in. Edge names must be unique across the whole t-decomposition, so an integer counter starting at $r + c + 1$ provides an easy source of names. Every member of D is given a unique name. Vertices of members of D are also given unique names, and a similar strategy to edge naming may be used.

For each edge, we store a pointer to the member wherein it appears. We also store the name of each of its two end vertices. For marker edges, we designate one vertex as $+$ and one as $-$. This encodes the orientation functions of our t-decompositions. For each member, we store a pointer to its predecessor, an integer representing its designation, the name of its parent marker, and the name of the child marker in its predecessor that this member corresponds to. For polygon members, the edge set is stored as a doubly linked list, allowing edges to be reordered efficiently. In the special case that a member is a bond and its parent is a polygon, we also store with the parent polygon which child marker(s) correspond to bonds. This allows RULE5 to be executed efficiently. The size of polygon members H of D may be stored and updated to allow for quick calculation of $A(H)$, but this is not required.

The ability to merge graphs efficiently is essential in achieving the almost-linear-time bound of the algorithm. This can be done by using the standard UNION-FIND data structure, which operates as follow:

A UNION-FIND data structure U consists of a set of names, each having an associated parent name and rank, upon which three operations are defined:

1. MAKE-SET(x), which adds x to U with x as its own parent;

2. FIND(x), which recursively calls FIND($parent(x)$) until reaching a value y where $y = parent(y)$. It then sets $parent(x) = y$ and returns y. This has the effect of compressing the path, making subsequent calls faster; and

3. UNION(x, y), which first calls FIND(y) and FIND(y). Without loss of generality, assume the rank of x is less than that of y. We then assign $parent(y) \leftarrow parent(x)$. If the ranks are equal, we do the same but increase the rank of x by 1.

By performing path compression and performing unions according to rank, a sequence of m UNION-FIND operations on an underlying set of n elements can be performed in time $O(m\alpha(n))$. The function $\alpha(n)$ refers to the inverse Ackermann function, an extremely slowly growing function. In particular, it has a value less than 5 for any remotely feasible value of n. A sequence of n UNION-FIND operations is thus considered to run in almost linear time.

There are two types of data that will be stored in this data structure: the vertices (which are themselves graphs) of $\mathcal{T}$, and the vertices of the members of D. In addition to the procedure outlined above, our purposes require several constant-time additions to UNION.

When storing the vertices of $\mathcal{T}$, the UNION operation corresponds to the merging of two adjacent graphs. When storing the vertices of members of D, the UNION operation corresponds to identifying two vertices as one.

When identifying vertices inside a member of a t-decomposition, the UNION operation needs no modification. When two graphs x and y are being merged by calling UNION(x, y), the UNION procedure is modified to also add all nodes of one graph into the other (along with their associated rank and parent arrays), perform UNION on similarly signed vertices of the parent marker and child marker to be merged, and delete the appropriate parent and child marker edges from the graph.

Implementation Details

The procedures for calculating both the restricted t-decomposition and the depth partition in the first part of HYPOPATH require a careful implementation to maintain the overall running time of the algorithm. We found that a naive implementation of these can easily make the runtime quadratic. A general strategy to implement line 2 of HYPOPATH is as follows:

Algorithm 11 Calculate Restricted t-decomposition

```
1: $L, T \leftarrow \{\}$
2: **for all** $e \in P$ **do**
3:     Add the name of the member of $D$ containing $e$ to $L$
4: **end for**
5: Let $u \in L$
6: Add $u$ to $T$
7: **while** $L - T \neq \emptyset$ **do**
8:     Let $v \in L - T$
9:     Add all members on the path from $v$ to the nearest member of $T$ to $T$
10: **end while**
11: Add all members of $T$ to a new $t$-decomposition $\hat{D}$
12: $root(\hat{D}) \leftarrow$ the member of $T$ whose predecessor is not in $T$
```

Finding the path on line 9 can be achieved by building up two paths, one from u and one from v, by following predecessors. Alternating back and forth between the u-path and the v-path, continue adding predecessors until the two paths intersect or we hit some other member of T. The latter case is crucial to ensure the optimal runtime of the algorithm. Always continuing until the paths from u and from v intersect works, but leads to a worst-case runtime of $O(|P|^2)$ in the case where u is the root and the members of L are all on the same path.

Algorithm 12 Calculate Depth Partition

```
1: $d_{root(\hat{D})} \leftarrow 0$
2: $\pi_0 = \{root(\hat{D})\}$
3: Let $N$ be the set of all non-root members of $\hat{D}$
4: while $N \neq \emptyset$ do
5:     Let $u \in N$
6:     Let $W$ be an ordered list of the members on the path between $u$ and the near-
       est member $v$ of $\hat{D}$ whose depth $d_v$ is known
7:     $i \leftarrow 1$
8:     for $w_i \in reversed(W)$ do
9:         $k \leftarrow d_v + i$
10:        $d_{w_i} \leftarrow k$
11:        Add $w_i$ to $\pi_k$
12:        Remove $w_i$ from $N$
13:        $i \leftarrow i + 1$
14:    end for
15: end while
```

The procedure to implement line 6 is only a slight modification of that for line 9 of Algorithm 11. In particular, we build up a list of predecessors of w, and stop when we hit a member whose depth is known. We will always encounter such a member because the depth of the root is known to be 0. The path W is assumed to include w as its first member, but not to include the member whose depth is known as its last member. We remark that the only reason we compute the depth partition separately from the restricted t-decomposition is that calculating the depth partition in this way requires knowing $root(\hat{D})$, which we only know after finishing Algorithm 11.

These implementations are based on the algorithms in Wagner's dissertation [12].

The lines of TYPING and HYPOPATH that require one to find the best possible orientation for the non-type-1 children requires some explanation, as it is not entirely obvious. Recall what information is known when we are typing the merged subtree rooted at Q: we know the types of all children of Q and we know that Q has been relinked appropriately. We also know that there can be at most two non-type-1 children and hence there are only a maximum of four orientation combinations to consider. So for each of the orientation combinations, we can construct a graph Q' by adding to Q the non-marker edges of the canonical representations of the child types according to the orientations F' we are considering. If we let P_t be the temporary path edges added in this step, then the type of $mst_{F'}(Q)$ is given by $A(Q', P_t \cup (P \cap E(Q')), pm(Q))$. We inspect (by deleting and readding temporary edges) each combination of orientations and choose the one with the lowest A-value. The temporary edges are then deleted. Each line where orientations are calculated thus expands to a maximum of 4 calls to A and $O(1)$ temporary edge insertions and deletions.

Time Complexity

We will not formally derive the time bound of the algorithm, but we will give an outline of the highlights. The complete derivation, along with a bound on space used, can be found in [3].

Most of the steps involved will require the use of FIND operations. We will assume for this analysis that such operations can be done in constant time, keeping in mind that our final bound must then include an additional factor of $\alpha(n)$.

We recall an important definition. For a member $Q \in \hat{D}$ with complete children $H_1, \ldots, H_n$, we define $W_Q = (P \cap E(Q)) \cup \{m_1, \ldots, m_n\}$, where m_i is the parent marker of H_i. Although we might sometimes be able to specify slightly tighter bounds, we will aim to, where possible, bound runtimes of certain procedure by $|W_Q|$.

We first address TYPING. Line 2 can clearly be accomplished in time $O(|\pi|)$, and $|\pi|$ is itself bounded by $O(|\hat{D}|)$. The calculation of X on line 4 can be done in time $O(|P|)$. We will henceforth not mention simple lines similar to this. The application of RELINK-NON-ROOT can be done in time $O(|W_Q|)$ since $X = W_Q$ because Q has no children if Q is in π_s. Similarly, the calculation of A on line 8 takes $O(|W_Q|)$ time. The first loop thus completes in time

$$O\Big(\sum_{Q \in \pi_s} |W_Q|\Big).$$

Turning to the for loop beginning on line 14, the number of representative graphs considered is bounded by the number of children of Q, which is bounded by $|W_Q|$. Each representative graph is composed of a constant number of edges and vertices, so the line takes time $O(|W_Q|)$. As before, relinking also takes time $O(|W_Q|)$. Finding the minimum orientation requires, by our previous discussion, at most four calls to A, with each call taking time $O(|W_Q|)$. Updating the orientations involves changing the signs of at most two parent markers and can be done in constant time. One iteration of the for loop thus completes in time

$$O\Big(\sum_{Q \in \pi_i} |W_Q|\Big).$$

The while loop beginning on line 13 has the effect of performing the inner loop once for every level of the depth partition except π_0 and π_s. Together with the first part of TYPING, we will thus do $O(|W_Q|)$ work for each member of $\hat{D}$ except the root. We now reference a calculation in [3]:

$$\sum_{Q \in \hat{D}} |W_Q| = |P| + |\hat{D}| - 1$$

This ensures that one call to TYPING can be done in time $O(|P| + |\hat{D}|)$.

The only aspect of HYPOPATH that differs from TYPING is the calculation of the restricted t-decomposition and the depth partition. In particular, the lines from line 13 onward are identical in time complexity to one iteration of the inner for loop of TYPING.

So, we need only address the time complexity of Algorithms 11 and 12. The first for loop in Algorithm 11 clearly takes time $O(|P|)$. A careful look at the while loop starting on line 7 shows that each member of the not-yet-constructed $\hat{D}$ will be considered at most once, as each will either appear as u, v, or one of the members found on the path calculated on line 9. The while loop thus takes time $O(|\hat{D}|)$. Altogether, finding the restricted t-decomposition takes time $O(|P| + |\hat{D}|)$.

Algorithm 12 is essentially the same, but with some constant-time calculations added within the while loop and with the absence of the initial step that took $O(|P|)$ time in Algorithm 11. Calculating the depth partition can thus be done in time $O(\hat{D})$.

Combining all of this, we conclude that one application of HYPOPATH can be completed in time $O(|P| + |\hat{D}|)$.

We now consider the applications of RULE1-RULE5 in TYPING and HYPOPATH. RULE1 and RULE2 are constant time assuming we have earlier stored the end-nodes u_1 and/or u_2 during a previous call to A. RULE3 has the potential to take $O(\hat{D})$ time due to the calculation of a path in $\hat{D}$. However, we are guaranteed that the conditions of RULE3 will be satisfied at most once during one call to TYPING, because otherwise we would conclude that P is not a hypopath. So, even though RULE3 is applied within a loop, we are guaranteed that its total contribution will be at most an added $O(\hat{D})$ time, which does not change the time complexity previously found for TYPING. RULE4 might also take $O(\hat{D})$ time, but it doesn't appear inside a loop (and even if it did, its conditions will only ever be satisfied once per call to HYPOPATH). RULE5 easily runs in constant time. Altogether, the application of these rules does not change the overall time complexity.

We now consider UPDATE. Lines 3-8 can be done in time $O(|C|)$. We now consider the first case ($K_1 = K_2$). The only potentially non-constant operation occurs in the calculation of the two paths on line 18. Because K_1 is a polygon whose edges are stored in a doubly linked list, we need only check the relative directions (within the doubly linked list) of the four edges incident to u_1 and u_2 to determine the information needed for the subsequent steps. Because we are splitting a polygon in two, only one will keep its old name. We will thus have to update the edge locations of all edges in one of the polygons P_1 or P_2. Notice however that one of these polygons is guaranteed to be made up only of edges of P along with child markers that correspond to complete children containing edges of P. Thus, by choosing this polygon to update, we can bound the time required by $O(|P|)$.

The case that $K_1 \neq K_2$ is more complex. Finding the path R on line 26 takes time $O(|R|)$. This is not a particularly illuminating bound, and we could just as well bound it by $O(|D^*|)$. The loop on line 28 is complicated but performs only a constant number of operations per iteration, and thus completes in time $O(|R|)$ overall. Each application of SPLIT is essentially the same as the procedure encountered earlier on lines 18-23, so $O(|P|)$ work is needed for all of the calls to SPLIT combined, because the exact edges of P involved in one application of SPLIT will not be involved in another call to SPLIT. Finally, the calculation of G on line 54 requires $O(|R|)$ calls to UNION and FIND.

Altogether, one call to UPDATE thus takes time $O(|R| + |C|)$.

It is clear that $|P|$ and $|C|$ are bounded by the number of ones in a given column, and so if we perform $O(|P| + |C|)$ operations for each column, the total time taken throughout IS-GRAPHIC will be $O(n)$, where n is the number of ones in the entire matrix.

It is less clear that the $|\hat{D}|$ and $|R|$ terms are similarly bounded. To see this, we introduce without proof two theorems from [3].

In these theorems, r and c are the number of rows and columns, respectively, of the input matrix and n is the number of nonzero entries in the input matrix. Further, we denote as $\hat{D}_i$ the restricted t-decomposition calculated for the ith column of the input matrix and denote as R_i the path calculated in line 26 of UPDATE as called for the ith column of the input matrix. There is not necessarily an R_i for each column.

Theorem 8. *For $r \geq 3$,*

$$\sum_{\text{all } R_i} |R_i| \leq 6r - 12$$

Theorem 9.

$$\sum_{i=1}^{c} |\hat{D}_i| \leq 2n + c + 6r - 12$$

From this we can deduce that the final time complexity for IS-GRAPHIC is $O(n + c + r)$. Because we have specified that our input matrix have no zero columns or rows, the c and r terms are themselves bounded by n and are redundant. Reintroducing the α term, we have that one application of IS-GRAPHIC runs in time $O(n\alpha(n))$.

We note that the inverse Ackerman function is technically a function of two variables $\alpha(m, n)$ with $m \geq n$, and the full statement of the runtime is $O(n\alpha(n, r))$. However, because the inverse Ackerman function grows so extraordinarily slowly, the choice of inputs is often abbreviated.

Motivation and Conclusion

The motivation behind Bixby and Wagner's algorithm is its potential in solving certain problems in linear programming. In general, a linear programming problem (called a *linear program*) involves minimizing or maximizing a linear *objective function* according to a set of linear equations or inequalities called *constraints*. A wide variety of problems in economics and business can be stated as linear programs. Being able to solve a linear program with a large number of variables in a tractable amount of time has thus been the subject of much research.

The most general algorithm widely used to solve linear programs is the *simplex method* of Dantzig. Although often much better in practice, the simplex method has been shown to be exponential in the worst case [4]. There are a large number of other algorithms aimed at solving linear programs of specific forms.

One particular structure that can arise is called a *network flow problem*. In particular, network flow problems can be solved much more efficiently than linear programs in general. Work by Bixby and Cunningham in [1] described an algorithm for converting a linear program, if possible, into a network flow problem. The crucial discovery was that this conversion problem is equivalent to the problem of realizing a binary matroid as graphic. With the almost-linear-time algorithms of Fujishige and of Bixby and Wagner, this conversion problem can now be solved extremely efficiently.

More recently, this algorithm has found use in the *perfect phylogeny haplotype problem* (PPH): given a set of genotypes, M, find a set of explaining haplotypes, M', which defines a perfect phylogeny. This problem, it turns out, can be reduced in polynomial time to an instance of the graph realization problem [6]. We will not attempt to expand on the details of the PPH problem as that is far beyond the scope of this paper. We mention it in brief, however, to illustrate the far-reaching impact that the solution to a seemingly esoteric problem in mathematics can have.

An attractive attribute of this algorithm is that the final t-decomposition also gives an easy way to enumerate all possible (nonseparable) solutions to the problem. To do this, we enumerate all possible combinations of orientations and polygon relinkings. In doing so, we obtain all nonseparable graphs which provide a solution to our problem. This turns out to be useful in the case of the PPH problem.

Finally, we found that accessible implementation details beyond those provided in [3] were virtually nonexistent. Our motivation was thus to (1) describe the graph realization problem in a way that didn't require extensive mathematical training, (2) present Bixby and Wagner's algorithm as clearly as possible, (3) thoroughly document the practical considerations involved in writing an implementation of the algorithm, (4) write an implementation of the algorithm, and (5) contribute our implementation to the SageMath project. This written thesis serves to fulfill points (1)-(3), while point (4) was mostly done before writing to ensure any interpretations presented were correct. Fulfillment of (5) will follow the submission of this thesis, and further information along this line can be found at https://trac.sagemath.org/ticket/20834.

REFERENCES

[1] Robert E. Bixby and William H. Cunningham. "Converting Linear Programs to Network Problems". In: *Mathematics of Operations Research* 5.3 (1980), pp. 321–357. ISSN: 0364765X, 15265471. URL: http://www.jstor.org/stable/3689441.

[2] Robert E. Bixby and Donald K. Wagner. "An Almost Linear-Time Algorithm for Graph Realization". In: *Mathematics of Operations Research* 13.1 (1988), pp. 99–123. DOI: 10.1287/moor.13.1.99. URL: https://doi.org/10.1287/moor.13.1.99.

[3] Robert E Bixby and Donald K Wagner. *An Almost Linear-Time Algorithm for Graph Realization*. Tech. rep. 85-2. Rice University, 1985.

[4] National Research Council. *Probability and Algorithms*. Washington, DC: The National Academies Press, 1992. ISBN: 978-0-309-04776-0. DOI: 10.17226/2026. URL: https://www.nap.edu/catalog/2026/probability-and-algorithms.

[5] William H Cunningham and Jack Edmonds. "A Combinatorial Decomposition Theory". In: *Canadian Journal of Mathematics* 32.3 (1980), pp. 734–765.

[6] Zhihong Ding, Vladimir Filkov, and Dan Gusfield. "A Linear-Time Algorithm for the Perfect Phylogeny Haplotyping (PPH) Problem". In: *Research in Computational Molecular Biology*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 585–600. ISBN: 978-3-540-31950-4.

[7] Satoru Fujishige. *An Almost-Linear-Time Algorithm for Solving the Graph Realization Problem*. Tech. rep. Kyoto University, 1980.

[8] Lara Löfgren. "Irredundant Boolean Branch-Networks". In: *"IRE Transactions on Information Theory"* 5.5 (1959), pp. 158–175.

[9] Takahiro Ohto. “An Experimental Analysis of the Bixby-Wagner Algorithm for Graph Realization Problems”. In: *Research Report of the Special Interest Group for Algorithms of the Information Processing Society of Japan* 84 (May 2002), pp. 1–8. ISSN: 09196072. URL: https://ci.nii.ac.jp/naid/110002812484/en/.

[10] James Oxley. *Matroid Theory*. 2nd ed. Oxford University Press, 2011.

[11] W. T. Tutte. “An Algorithm for Determining Whether a Given Binary Matroid is Graphic”. In: *Proceedings of the American Mathematical Society* 11.6 (1960), pp. 905–917. ISSN: 00029939, 10886826. URL: http://www.jstor.org/stable/2034435.

[12] Donald Wagner. “AN ALMOST LINEAR-TIME GRAPH REALIZATION ALGORITHM”. PhD thesis. Northwestern University, 1983. URL: http://search.proquest.com/docview/303187154/.

Appendix A

Experimental Validation of Runtime

We did not have time to perform a thorough runtime evaluation of our implementation. We did however run our algorithm on upper triangular matrices of the form

$$M = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

as a preliminary confirmation of runtime. We ran the algorithm for matrices of size 100×100 up to $1,500 \times 1,500$. From 100 to $1,000$ we proceeded in increments of 50, and from $1,000$ to $1,500$ in increments of 100. The following chart and table detail our results. The number of nonzero entries in an $N \times N$ matrix of this form was found to be $\frac{N(N+1)}{2}$.

Data was gathered using the `%timeit` magic command available in the SageMath interactive console. The exact code used to test an $N \times N$ matrix is as follows:

```
sage: n = N
....: M = []
....: for i in range(0, n):
....:     M.append([])
....:     for j in range(0, n):
....:         if i <= j:
```

```
....:             M[i].append(1)
....:         else:
....:             M[i].append(0)
....: M_matrix = matrix(M)
....: %timeit graph = realize_and_merge(M_matrix)
```

The function `realize_and_merge` is a direct implementation of IS-GRAPHIC as presented in this paper.

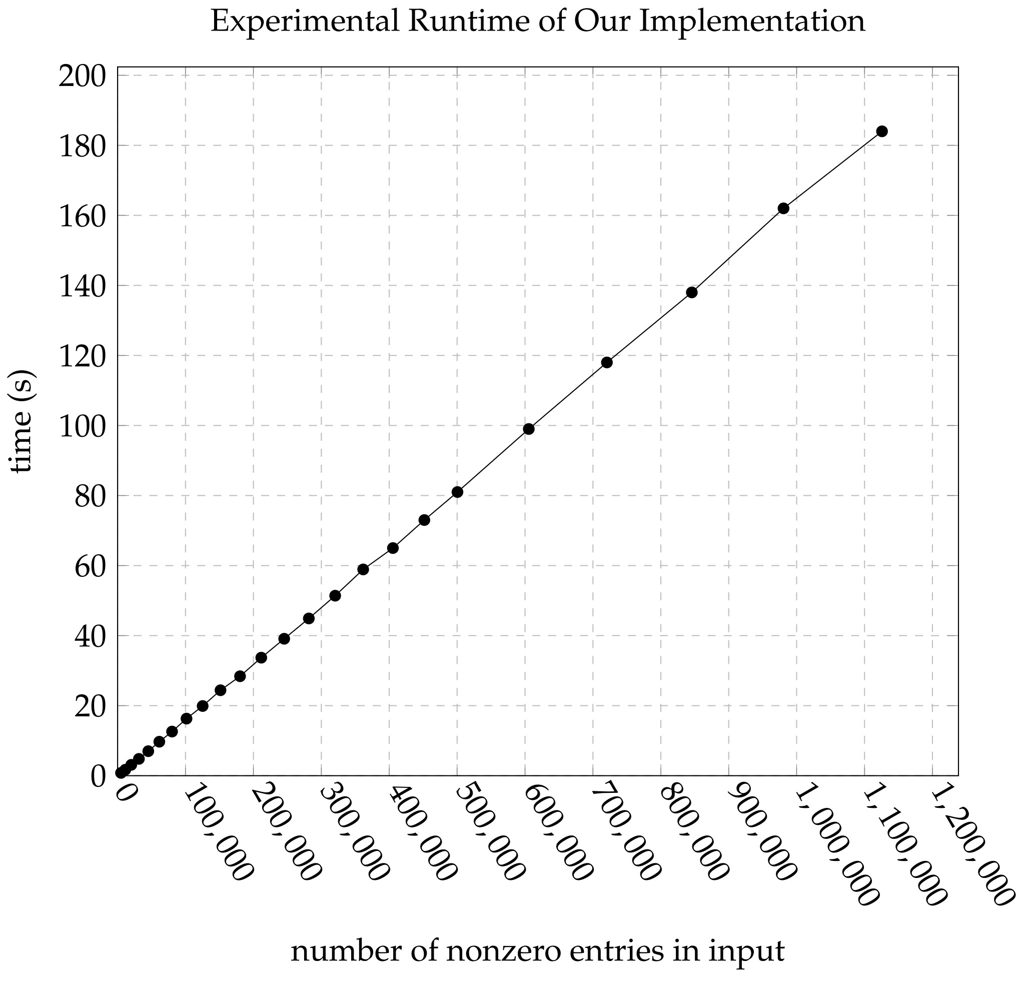


Table A.1 Experimental Runtime

Matrix size	Nonzero entries	Time (s)	Unit Time (ms)
100	5,050	0.8	0.150
150	11,325	1.7	0.152
200	20,100	3.1	0.153
250	31,375	4.8	0.153
300	45,150	7.0	0.154
350	61,425	9.7	0.158
400	80,200	12.6	0.157
450	101,475	16.3	0.161
500	125,250	19.9	0.159
550	151,525	24.4	0.161
600	180,300	28.4	0.158
650	211,575	33.7	0.159
700	245,350	39.1	0.159
750	281,625	44.9	0.159
800	320,400	51.4	0.160
850	361,675	58.9	0.163
900	405,450	65	0.160
950	451,725	73	0.162
1000	500,500	81	0.162
1100	605,550	99	0.163
1200	720,600	118	0.164
1300	845,650	138	0.163
1400	980,700	162	0.165
1500	1,125,750	184	0.163

Appendix B

Terminology Changed from that of Bixby and Wagner

In some situations we saw fit to slightly alter the notation from that of [3]. In order to make it easier to cross reference, we provide a short table of alternate terms.

Term used in [3]	Term we use
arborescence	rooted directed tree
prime	3-connected
node	vertex
subgraph reversal	twisting
RELINK1	RELINK-NON-ROOT
RELINK2	RELINK-ROOT
SQUEEZE	SPLIT
R1-R5	RULE1-RULE5
reduced t-decomposition	restricted t-decomposition
$m(D)$	$merge(D)$
$p(Q)$	$pred(Q)$
$\{0,1\}$-matrix	matrix over $\mathbb{F}_2$
$Q_f[H_1,\ldots,H_n]$	$mst_f(Q)$

Biography

Neil Wiebe Thiessen was born and raised in Seminole, TX. During his time at Angelo State University, Neil has been a member of the Honors Program and the university chapter of the Mathematical Association of America (MAA). He will graduate with his Bachelor of Science in Mathematics and Computer Science in May of 2020. He completed a research project in general topology under Dr. Trey Smith in the fall semester of 2019. Neil has been the recipient of two faculty-mentored undergraduate research grants, and was nominated for the Angelo State University Presidential Award by the computer science department in April of 2020.

Neil can be reached via email at neilthiessen1@gmail.com.