Rose-Hulman Undergraduate Mathematics Journal

Volume 20
Issue 1

Article 7

Triangle Packing on Tripartite Graphs Is Hard

Peter A. Bradshaw
University of Kansas, peter.a.bradshaw@gmail.com

Triangle Packing on Tripartite Graphs Is Hard

Cover Page Footnote
I sincerely thank Professor Jeremy Martin for being a truly inspiring educator, as well as for his continued support and guidance throughout the writing and editing process of this paper. I also thank Professor Bruce Reed for providing a reference to Mirko Morandini's article on the NP-completeness of partitioning tripartite graphs into triangles.

Rose-Hulman Undergraduate Mathematics Journal
Volume 20, Issue 1, 2019

Triangle Packing on Tripartite Graphs is Hard

By *Peter Bradshaw*

Abstract. The problem of finding a maximum matching on a bipartite graph is well-understood and can be solved using the augmenting path algorithm. However, the similar problem of finding a large set of vertex-disjoint triangles on tripartite graphs has not received much attention. In this paper, we define a set of vertex-disjoint triangles as a "tratching." The problem of finding a tratching that covers all vertices of a tripartite graph can be shown to be NP-complete using a reduction from the three-dimensional matching problem. In this paper, however, we introduce a new construction that allows us to emulate Boolean circuits using tripartite graphs in order to prove that covering a given vertex subset of a tripartite graph with a tratching is NP-hard, thereby attacking the tratching problem from a new angle.

1 Introduction

Imagine that you are organizing a concert, and you need to make a band lineup. You have a list of guitarists and a list of bassists, and you'd like to pair them up for the show. However, the rock-and-roll life is filled with drama, and some of these musicians refuse to work with each other. How many duos can you put together so that the members of each band are willing to work together?

We can tackle this question with some basic graph theory. Imagine that we have guitarists John, Janick, and Tony, and we have bassists Flea, Steve, and Geezer. Everyone is willing to cooperate, except for a few conflicts. John and Geezer don't want to play together because their styles don't match, and Tony and Flea have the same issue. And if Tony and Steve get anywhere near each other, then suffice it to say that there won't be a show anymore.

We can represent this situation with a graph. We include a vertex for each musician, and we draw an edge between any guitarist-bassist pair that is willing to be in the same band, as in Figure 1(i).

This graph can be colored with two colors such that no two adjacent vertices—that is, no two vertices that have an edge between them—have the same color. We call this

Mathematics Subject Classification. Primary: 05C70; Secondary: 05C85, 68R10, 68Q17
Keywords. triangle packing, matching, tripartite graphs, NP-complete

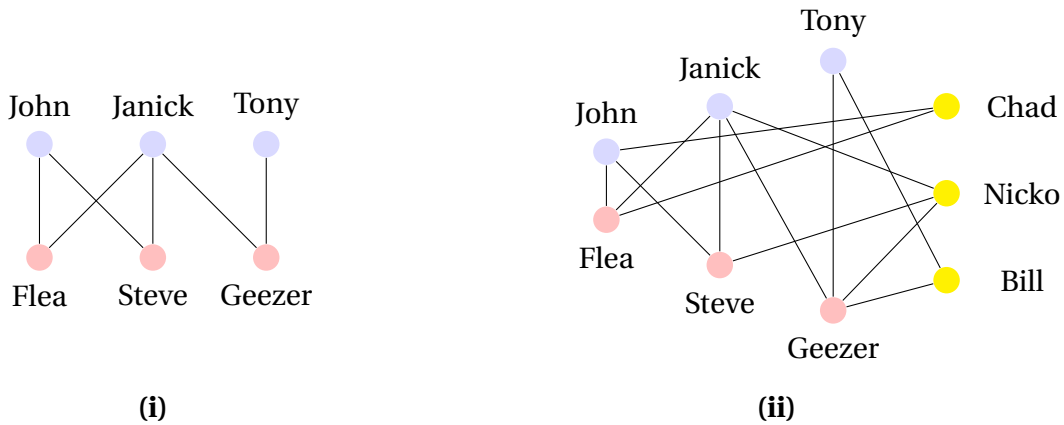

Figure 1: A bipartite graph (i) and a tripartite graph (ii). Red, blue and yellow vertices represent bassists, guitarists and drummers respectively. An edge between two vertices means that the two musicians are willing to work together.

type of graph **bipartite**. The set of red vertices and the set of blue vertices—here, the guitarists and bassists— are called the **partite sets** of the graph.

So our task of finding the greatest number of cooperating duos becomes the task of finding the largest set of nonadjacent edges on a bipartite graph. We call a set of nonadjacent edges a **matching**. That is, a matching is a set of edges that do not touch each other. It turns out that we have an efficient method for finding the maximum matching on a bipartite graph, which we will explain later. Therefore, putting together the greatest number of duo groups is a tractable problem.

Imagine again that you are organizing a concert, but this time you have guitarists, bassists, and drummers. You'd like to group these musicians into bands consisting of one guitarist, one bassist, and one drummer. However, again, some of the musicians cannot be in the same band. How many trios can you put together so that all of the members of each band are willing to work together?

Again, we can model this problem with graph theory. We'll use our previous musicians, and we'll add drummers Chad, Nicko, and Bill. Chad will only play with Flea and John; Nicko will only play with Steve, Geezer, and Janick; and Bill will only play with Geezer and Tony. Again, we can represent each musician with a vertex, and we can draw an edge between any two musicians of different instruments that are willing to work with each other, as in Figure 1(ii). Notice that this graph can be colored with three colors such that no two adjacent vertices have the same color; that is, this graph is **tripartite**.

So our task of finding the greatest number of cooperating trios becomes the task of finding the largest set of vertex-disjoint triangles on a tripartite graph. This task will be

the motivation of our paper. Although the problem of finding the greatest number of cooperating trios is known to be NP-hard [5], we present new constructions that can potentially be applied to new problems. The paper will be structured as follows. In Section 2, we show an attempt to extend bipartite matching techniques to tripartite graphs. In Section 3, we build a framework to show that covering a given vertex subset of a tripartite graph with vertex-disjoint triangles is intractable. In Section 4, we use this framework to prove intractibility. Finally, Section 5 discusses remaining open questions.

2 Creating a Band Lineup

When organizing a set of duos with only guitarists and bassists, we can approach the problem by finding matchings on bipartite graphs. For matchings on bipartite graphs, we have many theorems and algorithms at our disposal, so we can solve the band lineup problem if only guitarists and bassists are involved. However, matchings don't help if we add drummers to the mix. Instead, we need to extended the definition of matching to a "three-way matching," or for short a "tratching". When we add drummers, the band lineup problem becomes much harder.

2.1 Matchings on bipartite graphs

Definition 2.1. A **graph** is a pair (V, E), where V is a set of **vertices** and E is a set of **edges**. Each edge $e \in E$ is given by an unordered pair of vertices $v, w \in V$, called its **endpoints**. We say two endpoints of a single edge are **adjacent**. For a graph G, the vertices of G are often denoted as $V(G)$, and the edges of G are often denoted as $E(G)$.

Definition 2.2. A graph G can be k**-colored** if each vertex of G can be assigned one of k colors such that no two adjacent vertices have the same color.

Definition 2.3. A graph G is **bipartite** if G can be 2-colored.

Definition 2.4. A graph G is **tripartite** if G can be 3-colored.

Definition 2.5. Let G and H be graphs such that $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. Then we say that H is a **subgraph** of G, and we use the notation $H \subseteq G$.

Definition 2.6. Let G be a graph, and let $H \subseteq G$ be a subgraph of H. If $U \subseteq V(G)$, then we say that H **covers** U if $U \subseteq V(H)$. Furthermore, for each vertex $v \in V(U)$, we say that H **covers** v.

Definition 2.7. Let G be a graph. A **matching** M on G is a set of edges in G such that no vertex in G belongs to two distinct edges in M. A matching M is a **maximum matching** if G has no matching larger than M, and M is a **perfect matching** if every vertex in G belongs to an edge in M.

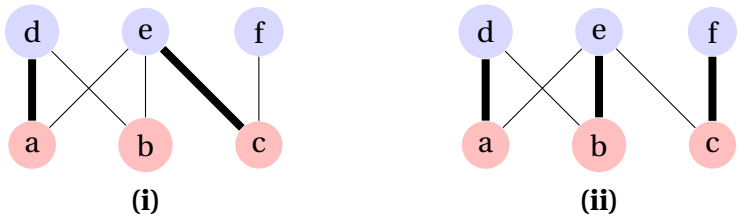

Figure 2: (i) A non-maximum matching on a bipartite graph. (ii) Increasing the size of the matching using the augmenting path algorithm.

If we only have guitarists and bassists, our goal is to create a setlist of duos that uses every musician exactly once. Reformulating this goal into a graph theory problem, we ask: if we have a bipartite graph, can we find a perfect matching?

This question can be answered simply by finding a maximum matching and checking if it covers all vertices. If we have a bipartite graph G, then we can obtain a maximum matching on G by using an efficient algorithm that increases the size of a bipartite matching until it is maximum. We demonstrate this algorithm using the bipartite graph in Figure 2(i). We begin by arbitrarily choosing any matching on the bipartite graph. We call our matching M, and we draw the edges included in M in bold. Then, after considering the path $f-c-e-b$, we see that we can obtain a larger matching by removing the edge ec from M and adding fc and eb. We show the new matching in Figure 2(ii). The reason that we can use the edges on the path $f-c-e-b$ to obtain a larger matching is that $f-c-e-b$ is an **augmenting path**, which brings us to a definition.

Definition 2.8. Let G = (V, E) be a bipartite graph. A **path** on G is a sequence of vertices $v_1v_2\dots v_k$ such that v_i is adjacent to v_{i+1} for $1 \le i \le k-1$. Let $M \subseteq E(G)$ be a matching on G. An **augmenting path** on M is a path $v_1v_2\dots v_{2n}$, $n \in \mathbb{N}$, such that $\{v_2v_3, v_4v_5, \dots, v_{2n-2}v_{2n-1}\} \subseteq M$ and such that v_1 and v_{2n} are not covered by M.

It then follows that $M \setminus \{v_2v_3, v_4v_5, \dots, v_{2n-2}v_{2n-1}\} \cup \{v_1v_2, v_3v_4, \dots, v_{2n-1}v_{2n}\}$ is a larger matching on G. (See Figure 3.)

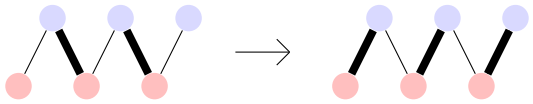

Figure 3: Augmenting Path.

Theorem 2.9. ***Berge's Lemma [3]:***
Let G *be a bipartite graph, and let* M *be a matching on* G. *Then* M *is a maximum matching if and only if* M *has no augmenting path.* □

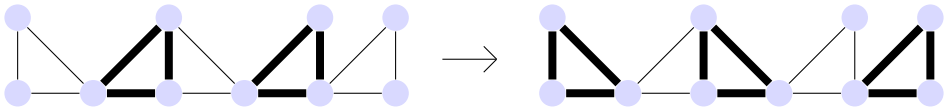

Figure 4: Using an “augmenting triangle path” to increase the size of a tratching.

So if we have a bipartite graph G, then we can use this process to find a maximum matching on G. We start with any matching M, then find all augmenting paths on M by generating a search tree starting at each vertex not covered by M, and use these augmenting paths to iteratively increase the size of M. Finally, once we see that M has no augmenting paths, we know that M is a maximum matching. This algorithm gives us a tool to efficiently solve the band lineup problem with only guitarists and bassists.

2.2 Tratchings on tripartite graphs

If we have guitarists, bassists, and drummers, our goal now is to create a setlist of trios that uses every musician exactly once. To formulate this problem using graph theory, we need to introduce some new definitions.

Definition 2.10. Recall that K_3 is the triangle graph, with three vertices and three edges. Let G be a graph, and let $T \subseteq G$ be a set of K_3's on G. We say that T is a **tratching** if no pair of K_3's in T shares a vertex. We say that T is a **perfect tratching** if T covers all vertices of G.

Definition 2.11. Let G be a graph, let $S \subseteq V(G)$, and let T be a tratching. We say that T is an S-**tratching** if T covers all of the vertices of S.

So in terms of graph theory, our goal is to find a perfect tratching on a tripartite graph. We could attempt to solve this problem with a similar approach to what we used for bipartite graphs and look for “augmenting triangle paths," as shown in Figure 4. If tripartite graphs are similar to bipartite graphs, then it's possible that the nonexistence of an “augmenting triangle path" indicates a maximum tratching.

Unfortunately, it isn't so simple. Consider the example in Figure 5. This graph is tripartite, and a tratching is shown in bold. If we look at the portion of the graph with labeled vertices, we can see that there are four triangles not contained in the tratching $\{abe, fgi, dhj, klm\}$ and three triangles contained in the tratching $\{bef, cdg, ijk\}$. We can increase the size of the tratching by removing $\{bef, cdg, ijk\}$ from our tratching and adding $\{abe, fgi, dhj, klm\}$. However, in this case, there is no obvious “augmenting triangle path" that tells us which triangles to add to our tratching and which triangles to remove. Thus we see that for tripartite graphs, no clear analogue of augmenting paths exists. In fact, because finding a maximum tratching is an NP-hard problem, no such efficient algorithm exists.

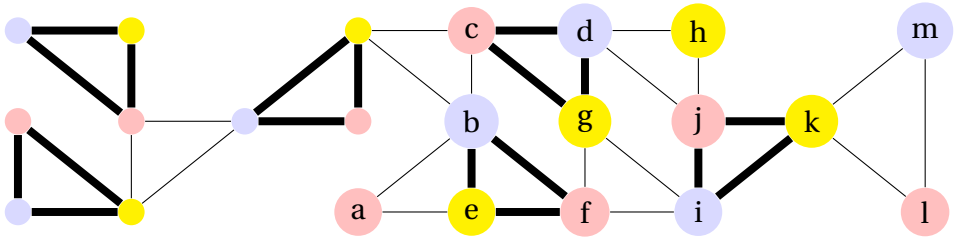

Figure 5: The tratching on this tripartite graph is not maximum. However, the tratching cannot be improved using an “Augmenting Triangle Path."

3 The Talented Musician Problem

3.1 An introduction to the problem

Imagine again that you are organizing a concert and that you have a set of available guitarists, bassists, and drummers. Again, some of the musicians are not willing to work with each other. But imagine your available guitarists are your friend, your loud upstairs neighbor, that guy who always plays on the street downtown, and John Frusciante from the Red Hot Chili Peppers. Regardless of how many bands you can put together, you would just be an irresponsible organizer if you didn't include John Frusciante in your lineup. Perhaps Neil Peart and Chad Smith are on your list of drummers. Perhaps there's a whole group of fantastic musicians, and you want to include as many of them as possible, even if some of them have to play with amateur musicians. Can we create a setlist that uses every talented musician exactly once? We will call the problem of arranging a setlist of trios that includes every talented musician from a larger set of musicians the **Talented Musician Problem**.

As before, we can transform the **Talented Musician Problem** into a graph theory problem. We construct a tripartite graph G where we represent each musician with a vertex and put an edge between each pair of musicians that are willing to work together. Then we give the name S to the set of vertices that represent talented musicians. We ask ourselves, can we find an S-tratching on G?

In the bipartite case, with only guitarists and bassists, this problem can be solved efficiently, thanks to the Hungarian Algorithm, which is an algorithm that finds a maximum weighted matching on a bipartite graph. If we have a bipartite graph G and some set $S \subseteq V(G)$, then we can first give each edge e a weight equal to the number of vertices in S adjacent to e. Then we can use the Hungarian Algorithm to find a maximum weighted matching M on G. If M has weight $|S|$, then M must cover every vertex in S. Otherwise, no matching covering S exists.

What about the tripartite case? If we have a tripartite graph G with $S \subseteq V(G)$, can we efficiently determine if an S-tratching exists? It turns out that we can't. As we will soon

see, finding an S-tratching on a tripartite graph is in a special class of problems known as NP-complete.

Definition 3.1. A **decision problem** is a problem with possible answers “yes" or “no." A decision problem is informally defined as **NP-complete** if it is difficult to solve quickly, but a “yes" answer can be verified quickly.

For example, the subset sum problem is NP-complete: given a set of integers, is there a nonempty subset that sums to zero? If we have a set of integers, it is difficult to find whether such a subset exists without resorting to brute force, but if someone gives us a subset and claims, “This subset sums to zero," then we can quickly verify whether or not the claim is correct.

It is beyond the scope of this paper to give a more rigorous definition of NP-completeness, but it should be enough for the reader to think of NP-complete as meaning “hard to solve, and quick to check." NP-complete problems appear everywhere in graph theory. Some NP-complete problems include determining if a graph is tripartite, finding an independent set of size k, finding a Hamiltonian cycle, and the Traveling Salesperson Problem. We aim to show that the **Talented Musician Problem** is also on the long list of NP-complete problems.

3.2 Building a framework for NP-completeness

How will we prove that the **Talented Musician Problem** is NP-complete? The way that we show a new problem is NP-complete is by showing that the problem is at least as hard as a problem that is already known to be NP-complete, and then by showing that a “yes" answer can be verified quickly. In this case, we will take a problem called **1-IN-3 SAT**, which is known to be NP-complete, and we will show that finding an S-tratching on a tripartite graph is at least as hard as **1-IN-3 SAT**. Then we will show that verifying an S tratching can be done quickly. By doing so, we will prove that the **Talented Musician Problem** is NP-complete.

Definition 3.2. Let $x_1, \ldots, x_m$ be Boolean variables; that is, each can take a value of either TRUE or FALSE. A **clause** C_i is a set of up to three literals $(a_1, ..., a_k)$, $k \leq 3$, where each literal is either a variable x_j or its negation $\neg x_j$. A clause C_i is **satisfied** if and only if exactly one literal in C_i is true. An instance of the **1-IN-3 SAT problem** is a finite set of clauses $\{C_1, \ldots, C_n\}$. A solution to the problem is an assignment of Boolean values to the variables such that all clauses are satisfied.

We see that **1-IN-3 SAT** is a decision problem, because it either “yes," has a solution, or “no," does not have a solution.
Example: The following is an example of an instance of **1-IN-3 SAT**.
We have variables $\{x_1, x_2, x_3, x_4\}$.

We have clauses

$$C_1 = (x_1, x_2, x_4), \quad C_2 = (\neg x_1, \neg x_2), \quad C_3 = (x_1, \neg x_2, x_3), \quad C_4 = (\neg x_1, \neg x_2, x_4).$$

We ask if there is a truth assignment to our set of variables that satisfies each clause. In this instance there is. We set

$$x_1 = \text{FALSE}, \quad x_2 = \text{TRUE}, \quad x_3 = \text{TRUE}, \quad x_4 = \text{FALSE}.$$

The following table shows that each clause contains exactly one true literal and is therefore satisfied.

C_1		C_2		C_3		C_4	
x_1	F			x_1	F	$\neg x_1$	T
x_2	T	$\neg x_1$	T	$\neg x_2$	F	$\neg x_2$	F
x_4	F	$\neg x_2$	F	x_3	T	x_4	F

Thus in this instance of **1-IN-3 SAT**, we have an assignment of variables that satisfies every clause. So this instance of **1-IN-3 SAT** has an answer "yes."

Determining if there is a variable truth assignment that satisfies every clause in an instance of **1-IN-3 SAT** is NP-complete. This is a consequence of Schaefer's dichotomy theorem [6, Theorem 2.1].

We will create a logical circuit to model an instance of **1-IN-3 SAT**. That is, for any instance P of **1-IN-3 SAT**, we will create a logical circuit that represents the information contained in P. Consider, for example, the instance of **1-IN-3 SAT** that contains two clauses: (x, y), and $(x, \neg y, z)$. This instance has three variables, so we build a circuit with three literal pairs at the top: $(x, \neg x)$, $(y, \neg y)$, and $(z, \neg z)$, as in Figure 6 (i). This instance has two clauses, so we put two clauses at the bottom of our circuit. Then we draw a line from the x literal to every clause that contains the literal x; we draw line from y to every clause that contains the literal y; we draw a line from $\neg y$ to every clause that contains the literal $\neg y$; and we repeat this process for each literal that appears in a clause. Finally, we simulate assigning a truth value to each variable by transmitting a signal from each true literal to all clauses connected to that literal. We say that our variable assignment satisfies this instance of **1-IN-3 SAT** if and only if each clause receives a signal from exactly one literal. In Figure 6 (i), we show that the assignment x = FALSE, y = TRUE, z = TRUE sends exactly one signal to each clause and thereby satisfies this instance of **1-IN-3 SAT**.

In Figure 6 (ii), we show another circuit built to represent the instance of **1-IN-3 SAT** with clauses $(\neg x, \neg y, z)$, $(\neg x, w)$, and $(\neg w)$. We show that an assignment of x = FALSE, y = TRUE, z = TRUE, w = FALSE does not satisfy this instance, as this assignment causes the clause $(\neg x, \neg y, z)$ to receive two signals. Changing the value of z to FALSE, however, would cause each clause to receive exactly one signal and hence would satisfy this instance of **1-IN-3 SAT**.

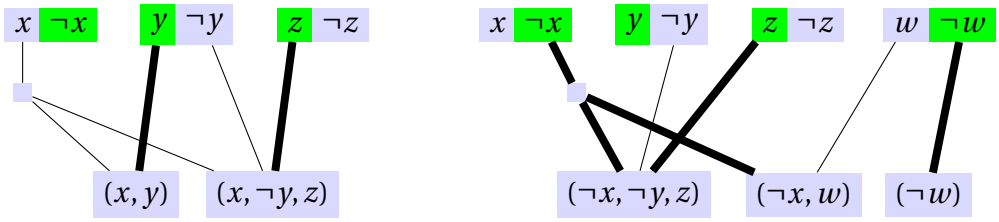

Figure 6: Two instances of **1-IN-3 SAT**, each represented by a circuit. In (i), the assignment x =FALSE, y = TRUE, and z = TRUE satisfies all clauses. In (ii), the assignment x =FALSE, y = TRUE, and z = FALSE, w = FALSE satisfies all clauses.

We can build a circuit for any such instance of **1-IN-3 SAT** using four specific gadgets: the variable gadget, wire gadget, fork gadget, and clause gadget. These gadgets are sketched in Figure 7.

Variable gadget

- The variable gadget represents a variable x.
- The variable gadget has exactly two **literal gadgets** x and $\neg x$.
- The variable gadget allows us to choose exactly one of its literal gadgets. The literal gadget that is chosen is **positive**; the other one is **negative**.

Wire gadget

- The wire gadget has two ends: a **base** and a **tail**.
- The wire gadget connects its base to a literal gadget.
- If we connect the base of a wire gadget to a positive (resp., negative) literal, then we say that the tail of the wire gadget is positive (resp., negative).

Fork gadget

- The fork gadget has three ends: one **base** and two **children**.
- The fork gadget can connect its base to a literal gadget.
- If we connect a fork gadget's base to a positive (resp., negative) literal gadget, then we say that each child is positive (resp., negative).
- We can connect the base of one fork gadget to the child of another fork gadget to emulate a fork gadget with more than two children.

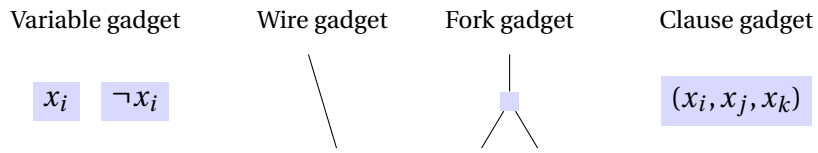

Figure 7: Gadgets.

Clause gadget

- The clause gadget represents a clause with literals $x_1, ..., x_k$, where $k \leq 3$.
- The clause gadget can connect to both wire gadgets and children of fork gadgets.
- For each literal x_i represented by the clause gadget, the clause gadget connects to exactly one wire gadget or one child of a fork gadget. The gadget to which the clause gadget connects must carry a value from the literal gadget x_i.
- We say that a clause gadget is **satisfied** if and only if it connects to exactly one positive gadget.

We can use these gadgets to build a circuit to emulate any instance P of **1-IN-3 SAT**.

- For every pair of literals x and $\neg x$ in P, we create a variable gadget with literal gadgets x' and $\neg x'$.
- For each clause C in P, we create a clause gadget C$'$.
- For each literal x_i that appears in clause C, we create a wire pathway from the literal gadget x_i' to C$'$. To create pathways from one literal gadget to multiple clause gadgets, we use fork gadgets.

An instance of **1-IN-3 SAT** has a “yes" answer if and only if all of the clause gadgets in its corresponding circuit can be satisfied. This technique of building a Boolean circuit out of gadgets in order to prove the complexity of a problem is a standard one. Proving that our circuits are equivalent to instances of **1-IN-3 SAT** is straightforward, because our gadgets have been specifically engineered to represent the **1-IN-3 SAT** problem.

Theorem 3.3. *An instance* P *of **1-IN-3 SAT** has a “yes" answer if and only if all of the clause gadgets in its corresponding circuit can be satisfied.*

Proof. Suppose P has an answer of “yes". Then for each clause C in P, C has exactly one literal whose value is true. Therefore, by construction, the clause gadget C$'$ that represents C in the corresponding circuit is connected to exactly one positive gadget is

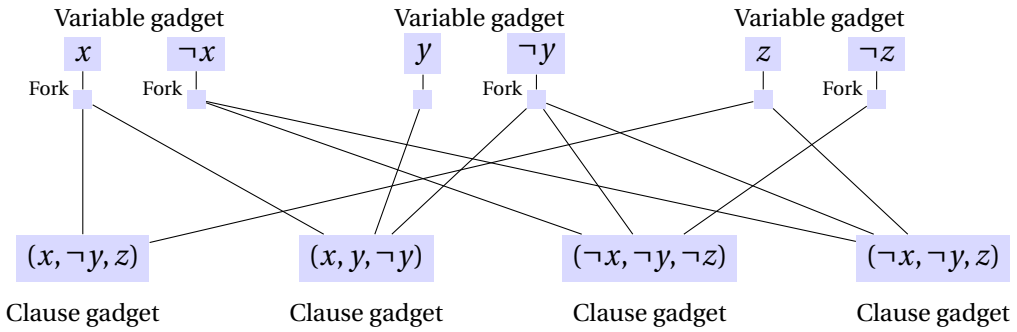

Figure 8: An instance of **1-IN-3 SAT** with four clauses. Each literal gadget is connected by wires and forks to all clause gadgets containing that literal.

therefore satisfied. Since all clause gadgets represent a clause in P, all clause gadgets in the circuit are satisfied.

Suppose on the other hand that each clause gadget C′ representing clause C in the circuit is satisfied. Then some choice of literal gadgets $x'_{a1}, ..., x'_{an}$ sends exactly one positive value to each clause gadget. Since each clause gadget C′ receives values from literal gadgets $x'_1, ..., x'_k$ if and only if clause C contains literals $x_1, ..., x_k$, then assigning the value TRUE to literals $x_{a1}, ..., x_{an}$ should cause each clause C to contain exactly one true literal and therefore be satisfied. Thus P has an answer of “yes." □

4 Reducing 1-IN-3 SAT to the Talented Musician Problem

We now show that finding an S-tratching on a tripartite graph is at least as difficult as **1-IN-3 SAT**. To accomplish this, we will show that for any instance P of **1-IN-3 SAT**, we can construct a tripartite graph G with S ⊆ V(G) such that an S-tratching exists on G if and only if all of the clauses of P can be satisfied. This means that an algorithm that efficiently finds S-tratchings on tripartite graphs can also efficiently solve **1-IN-3 SAT**, implying that finding an S-tratching on a tripartite graph must be at least as hard as **1-IN-3 SAT**.

We construct G and S by using tripartite graphs to emulate each gadget used in the circuit that models P. In other words, we will realize the gadgets of Figure 7 using small graphs that can be connected together so that circuits such as that of Figure 8 become schematics for large tripartite graphs. The end result will be that we can reduce the **Talented Musician Problem** from **1-IN-3 SAT**.

In the upcoming figures showing the tripartite graph gadgets, vertices in the set S will be called “starred" and marked with asterisks. The starred vertices represent the talented guitarists, bassists, and drummers, while the unstarred vertices represent the

untalented musicians. In our construction, we will attempt to find an S-tratching on one of our large constructed tripartite graphs by adding vertex-disjoint triangles to a set called T with the goal that T should cover all starred vertices.

4.1 Realizing gadgets as explicit graphs

In this section, A, B, and C refer to triangles.

1. The variable gadget. The realization of the variable gadget is shown in Figure 9.

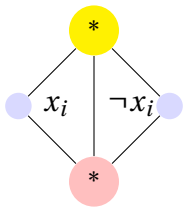

Figure 9: Variable gadget.

Observe that:

- The two literal gadgets in the variable gadget correspond to the triangles marked x_i and $\neg x_i$.
- We choose a literal gadget by adding its corresponding triangle to T.

In order to cover the starred vertices in the variable gadget, we must add at least one triangle to T. To guarantee that the triangles are disjoint, we must add no more than one triangle to T. Therefore, the variable gadget forces us to add exactly one triangle to T, and it therefore forces us to choose exactly one literal gadget. Thus the variable gadget functions as intended.

2. The wire gadget. The realization of the wire gadget is shown in Figure 10.

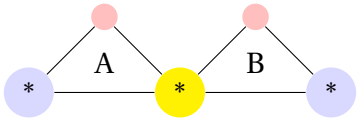

Figure 10: Wire gadget.

Observe that:

- The blue vertex of A represents the base of the wire gadget, and the blue vertex of B represents the tail.
- We attach the base of the wire gadget to a literal gadget by identifying the base of the wire gadget with the blue vertex of the literal gadget.
- If the triangle B is in T, then we say that the wire gadget is positive.
- If the triangle B is not in T, then we say that the wire gadget is negative.

If we attach the base of a wire gadget to a positive literal gadget by identification of blue vertices, then triangle A cannot be included in T without overlapping the triangle from the positive literal gadget. To cover the wire's yellow vertex with a disjoint triangle, we must then include the triangle B in our set T. In this way, if a wire gadget is connected to a positive literal gadget, then the wire gadget becomes positive, as in Figure 11.

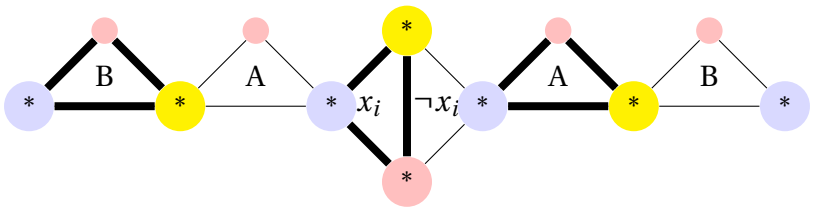

Figure 11: Two wire gadgets attached to a variable gadget. The wire on the left is positive, while the wire on the right is negative.

If we attach the base of a wire gadget to a negative literal gadget by identification of blue vertices, then we must include the triangle A in our set T; otherwise, the base of the wire gadget would not be covered by T. Thus we cannot include triangle B in our set T, because then two triangles $A, B \in T$ would intersect at the yellow vertex. In this way, if a wire gadget is attached to a negative literal gadget, then the wire gadget becomes negative, as in Figure 11.

Thus a wire gadget is positive if and only if its base is attached to a positive literal gadget as described above, and therefore the wire gadget functions as intended.

3. The fork gadget. The realization of the fork gadget is shown in Figure 12.

Observe that:

- The base of the fork gadget corresponds to the blue vertex of A.
- The children of the fork gadget correspond to the blue vertices of B and C.
- We attach the base of the fork gadget to a literal gadget by identifying the base of the fork gadget with the blue vertex of the literal gadget.

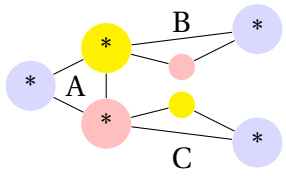

Figure 12: Fork gadget. A, B, and C are the triangles of the gadget.

- If the triangles B and C are in T, then we say that each child is positive; otherwise, each child is negative.

By reasoning similar to that used in the wire gadget, each child of the fork gadget is positive if and only if the base of the fork gadget is attached, by identification of blue vertices, to a literal gadget with a positive value, as in Figure 13. Therefore, the fork gadget functions as intended.

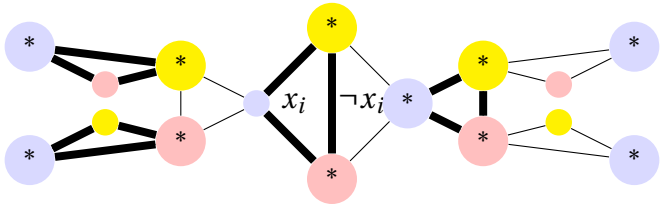

Figure 13: Two fork gadgets attached to a variable gadget. The children of each fork gadget carry the same value as the literal gadget to which the base of the fork is attached.

4. The clause gadget. The realization of the clause gadget for the clause x_i, x_j, x_k is shown in Figure 14. A clause with fewer variables can be realized by deleting one or more connections from x_i, x_j, and x_k.

Observe that:

- A clause gadget is constructed with a single blue vertex v in S.
- For each literal x_i in the clause, we identify with v exactly one wire gadget tail that carries a positivity value from the literal gadget representing x_i.
- If no such wire gadget tail is available, then we identify with v exactly one fork gadget child that carries a value from the literal gadget representing x_i.
- We say that a clause gadget is satisfied if it is covered by T.

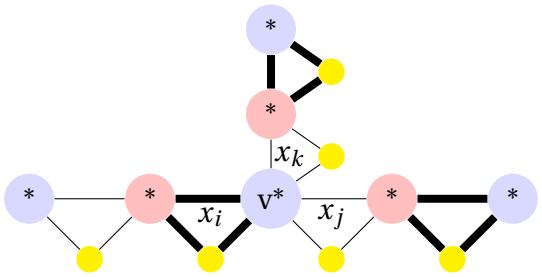

Figure 14: The clause gadget for the clause (x_i, x_j, x_k). The clause gadget's vertex is labelled v. In this figure, we have assigned x_i = TRUE, x_j = FALSE, x_k = FALSE.

Due to the way we have constructed wire gadgets and fork gadgets, a wire or fork gadget H that attaches to a clause gadget by identification of blue vertices is positive if and only if the triangle in H adjacent to the clause gadget is included in T. Therefore, if a clause gadget represents a clause with two positive literals, then two gadgets H_1 and H_2 will both require triangles that cover v to be included in T. However, these triangles from H_1 and H_2 intersect at v, so therefore they cannot both be included in T. So if we are to avoid this contradiction, then a clause gadget cannot be satisfied if it represents a clause with two positive literals.

On the other hand, if a clause gadget represents no positive literals, then no triangle adjacent to v can be included in T. Then v is not covered by T, and the clause gadget is not satisfied.

So a clause gadget is satisfied if and only if it represents a clause with exactly one true literal, as in Figure 14. Therefore, the clause gadget functions as intended.

4.2 The reduction

Every gadget in a **1-IN-3 SAT** circuit for a **1-IN-3 SAT** problem P can be recreated using a tripartite graph with specific vertex subsets that are to be covered by a tratching. Therefore, by piecing together our gadget graphs, we can create a tripartite graph G and $S \subseteq G$ such that G has S-tratching if and only if P can be satisfied.

Additionally, such a graph G is tripartite as constructed. To see this, we color each gadget with three colors such that any vertex that might be identified with another vertex is colored blue, and thus when we identify gadgets at blue vertices, we do not to disrupt the coloring of any gadget. Therefore any graph that we create with these gadgets is tripartite.

Figures 15 and 16 show the circuits and the tripartite graphs corresponding to two instances of **1-IN-3 SAT**, one satisfiable and one unsatisfiable.

Theorem 4.1. *The **Talented Musician Problem** is NP-complete. That is, if* G *is a tripartite*

graph and $S \subseteq V(G)$*, then the problem of determining the existence of an* S*-tratching on* G *is NP-complete.*

Proof. Let P be an instance of **1-IN-3 SAT**. Then we can represent P using gadgets as previously described by building a logical circuit. Then we build a tripartite graph G with $S \subseteq V(G)$ by piecing together graphs that emulate the gadgets in the circuit representing P. By our construction of these gadget graphs, G has an S-tratching if and only if P has a variable assignment that satisfies all of its clauses. As such, the problem of finding an S-tratching on a tripartite graph is at least as hard as **1-IN-3 SAT**.

Furthermore, one can quickly verify whether or not a given tratching covers S, so a "yes" answer to the problem of determining the existence of an S-tratching can be verified quickly. Therefore, the problem of finding an S-tratching on a tripartite graph is NP-complete. □

This technique of constructing gadgets and building a circuit in order to prove a problem is NP-complete is extremely common. It can be used to prove NP-completeness for many other problems, such as Yosenabe [4], Zig-Zag Numberlink [1], Super Mario Bros., Legend of Zelda, Metroid, and PushPush [2].

So it turns out that the **Talented Musician Problem** is NP-complete and therefore likely not a tractable problem. If we had an algorithm that, given any input, could efficiently create a band lineup that uses all talented musicians, then this algorithm would have to be able to efficiently solve every instance of **1-IN-3 SAT**. Complexity theorists do not believe that NP-complete problems like **1-IN-3 SAT** can be efficiently solved, so it follows that the **Talented Musician Problem** is also likely not to have an efficient solution.

4.3 Generalization with more musicians

One could generalize the **Talented Musician Problem** by adding more musicians to each band, such as vocalists or keyboard players. If each band requires r musicians and we have r types of musicians available, then a cooperative band could be represented not by a triangle, but by a K_r, the complete graph on r vertices. This generalized problem would be equivalent to determining the existence a set T of vertex-disjoint K_r's on an r-colored graph such that T covers a given $S \subseteq V(G)$. Even in doing so, however, the problem of finding a setlist that includes every talented musician would remain NP-complete for all $r \geq 3$.

If each band requires r musicians, we again can make gadgets to represent the parts of a **1-IN-3 SAT** circuit. For each gadget, we can add unstarred vertices and edges to each triangle to change the triangle into a K_r. In Figure 17, we show how a variable gadget and fork gadget are modified for the case when $r = 5$. After adding these new vertices, each gadget still functions as before, and the gadgets can still be identified at their blue vertices, so the resulting graph is r-colorable. Thus, for any instance P of **1-IN-3 SAT**,

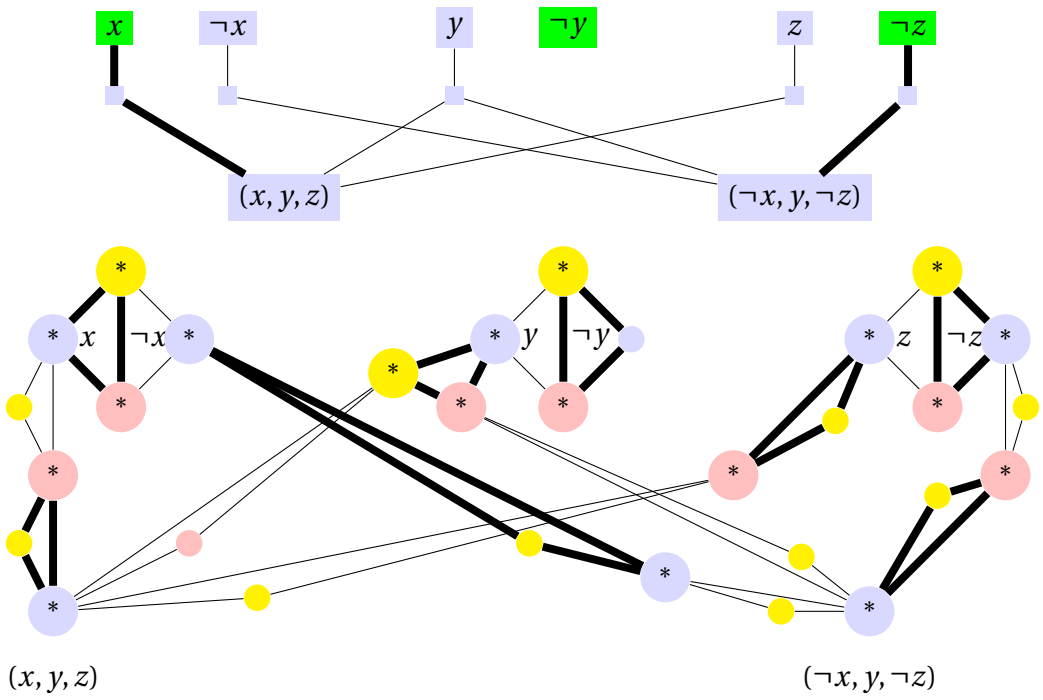

Figure 15: The boolean circuit (top) and tripartite graph (bottom) corresponding to the **1-IN-3-SAT** problem $\{(x, y, z), (\neg x, y, \neg z)\}$. This problem has the solution $(x, y, z) =$ (TRUE, FALSE, FALSE). Similarly, we can find an S-tratching by choosing literal gadgets x, $\neg y$, and $\neg z$.

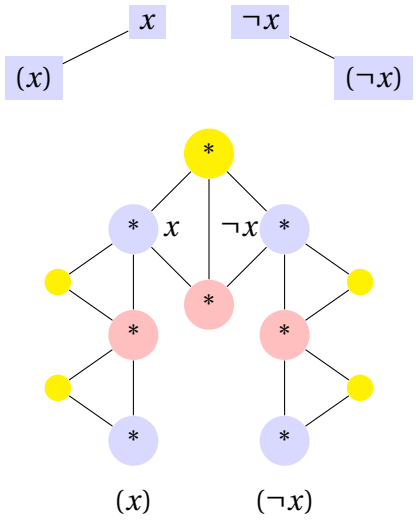

Figure 16: The boolean circuit (top) and tripartite graph (bottom) corresponding to the **1-IN-3-SAT** problem $\{(x), (\neg x)\}$. This instance of **1-IN-3 SAT** is unsatisfiable, and the tripartite graph has no S-tratching. Just as one cannot choose both x and $\neg x$, choosing one literal gadget causes the clause gadget attached to the other literal gadget not to be covered by T.

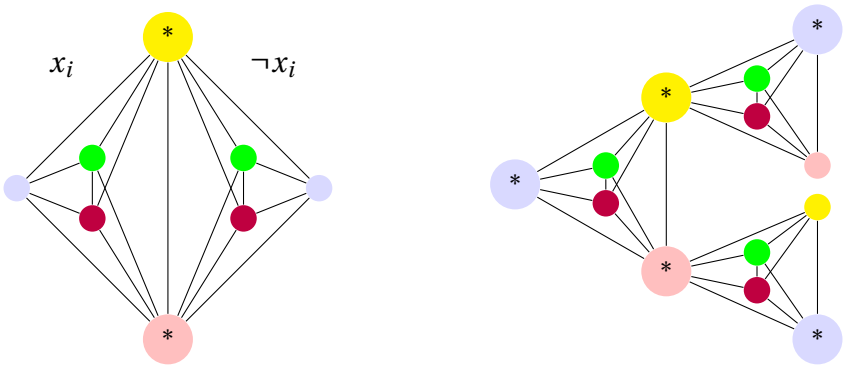

Figure 17: A 5-colored variable gadget and a 5-colored fork gadget. Each triangle from the original gadget is changed into a K_5.

we can create an r-colorable graph G with $S \subseteq V(G)$ such that S can be covered by a set of vertex disjoint K_r's if and only if P has a solution. Therefore, the **Talented Musician Problem** remains NP-complete even when more musicians are added to each band.

5 Conclusion

While the problem of determining the existence of a perfect tratching can be proven to be NP-complete using a reduction from three-dimensional matchings [5], our proof shows how to emulate boolean circuits by piecing together tripartite graphs that are constructed to function like Boolean gadgets. It would be intersting to see what other results can be proven using similar constructions.

References

[1] Aaron Adcock, Erik D. Demaine, Martin L. Demaine, Michael P. O'Brien, Felix Reidl, Fernando Sánchez Villaamil, and Blair D. Sullivan. Zig-Zag Numberlink is NP-complete. *J. Inform. Processing*, 23(3):239–245, 2015.

[2] Greg Aloupis, Erik D. Demaine, Alan Guo, and Giovanni Viglietta. Classic Nintendo games are (computationally) hard. *Theoret. Comput. Sci.*, 586:135–160, 2015.

[3] Claude Berge. Two theorems in graph theory. *Proc. Nat. Acad. Sci. U.S.A.*, 43:842–844, 1957.

[4] Chuzo Iwamoto. Yosenabe is NP-complete. *J. Inform. Processing*, 22(1):40–43, 2014.

[5] Mirko Morandini. NP-complete problem: Partition into triangles. Technical report, University of Verona, Verona, Italy, 2003-2004.

[6] Thomas J. Schaefer. The complexity of satisfiability problems. In *Conference Record of the Tenth Annual ACM Symposium on Theory of Computing (San Diego, Calif., 1978)*, pages 216–226. ACM, New York, 1978.

Peter Bradshaw
University of Kansas
peter.a.bradshaw@gmail.com