Algorithmic Generation of DNA Self-Assembly Graphs

G. Bielefeldt, I. Horng, H. Luebsen, M. VonEschen, L. Almodóvar, A. Harsy, C. Johnson, J. Sorrells*

Abstract - With recent advances in the field of nanotechnology, there has been increasing interest in self-assembling nanostructures. These are constructed through the process of branched junction DNA molecules bonding with each other without external guidance. Using a flexible tile-based model, we represent molecules as vertices of a graph and cohesive ends of DNA strands as complementary half-edges. Due to the unpredictability of DNA self-assembly in a laboratory setting, there is a risk that undesirable products are incidentally constructed. Predicting what structures can be produced from a given list of components (referred to as a "pot of tiles") is useful but has been proven NP-hard. This research introduces an algorithm that generates and visualizes a graph realized by a given pot. For smaller cases, it also produces all possible graphs up to isomorphism. By adjusting the construction parameters, the algorithm can produce graphs of any viable order with a given collection of tile type multiplicities.

Keywords : graph theory; DNA self-assembly; flexible tile-based model; algorithm; isomorphism; combinatorics

Mathematics Subject Classification (2020) **:** 05C90; 05C85; 92E10

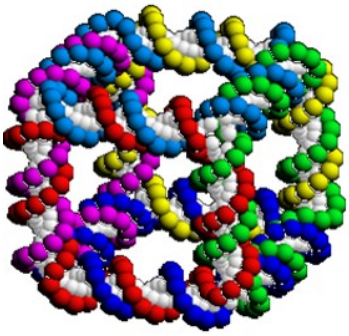
1 Introduction

With recent advances in drug delivery, biosensors, and biomolecular computing, there has been increasing interest in self-assembling nanostructures, particularly with DNA molecules [1, 2, 3]. In the late 20th century, most of these DNA nanostructures were created in laboratories with external guidance in a process called directed self-assembly [4]. However, this took significant amounts of time and resources and limited the size and quality of the desired nanostructures [5].

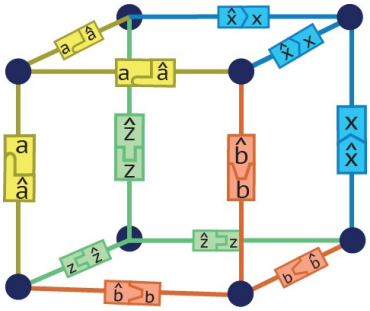
With the discovery of DNA self-assembly, these nanoparticles could be constructed through a process of branched DNA molecules spontaneously bonding with each other without external guidance. This allowed for the much-needed smaller and more complex molecules to be created, but there is still a possibility of error. With this development came the computational problem of generating and modeling these structures to predict the nanostructures that could be developed in a laboratory setting. Today, we effectively

*This work was supported by the National Science Foundation under Grant No. DMS-1929284

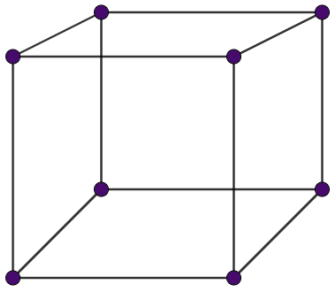
model these self-assembled structures using a flexible tile-based model. This is described in [1] as a model that uses flexible DNA molecules, called DNA tiles, so that the tiles can bend or deform during the self-assembly process (see Figure 1). Because of the unique and moldable properties of this model, we can construct complex 2D and 3D nanostructures with DNA.



A DNA cube nanostructure.[6]



A DNA cube represented by the flexible tile-based model.



A DNA cube represented as an unlabeled graph.

Figure 1: Various visual representations of a DNA cube nanostructure built with branched junction molecules.

Given components of a nanostructure, we can therefore use the flexible tile-based mathematical model to predict various structures that may result when the self-assembly process occurs. However, determining with accuracy all the possible resulting structures from a particular set of components is a time-consuming process, especially when only valid graph-theoretical structures are considered, where each edge links two distinct vertices. With a more extensive set of components, the time needed to solve this structure generation problem increases rapidly. In particular, this task is an NP-hard problem as proven in [7]. Therefore, the goal of this work is to automate this process of producing valid graphs from a given list of components.

2 Background

First, we review background on the notation and definitions of common graph theory, as well as algorithmic processes [1, 7, 8] that are used in this research.

2.1 Mathematically Modeling DNA Self-Assembly

In modeling self-assembled DNA nanostructures, as described before, we use a flexible tile-based graph-theoretical model, which represents k-armed, branched-junction DNA molecules as vertices of a graph based on research in [9]. However, for the rest of this paper, we will refer to the branches as *half-edges*. These half-edges are labeled with unhatted (ex: a) or hatted letters (ex: $\hat{a}$), which represent the unbonded strands of the DNA molecules.

In DNA self-assembly, unbonded strands can only bond with a complementary end to form a complete DNA duplex, which we label with the same letter. So, for example, a half-edge labeled a can only bond with a half-edge labeled with $\hat{a}$, in other words, the hatted letter represents the complement of the unhatted letter. *Cohesive end types* is the term used for these half-edges that can only bond with their complementary cohesive end. Each pair of complementary cohesive end types (a with $\hat{a}$, b with $\hat{b}$, etc.) is referred to as a *bond-edge type*. These pairs bond with each other to connect the vertices and form an edge in the graph-theoretical model (Figure 2). Note that these edges are assumed to be flexible, not rigid.

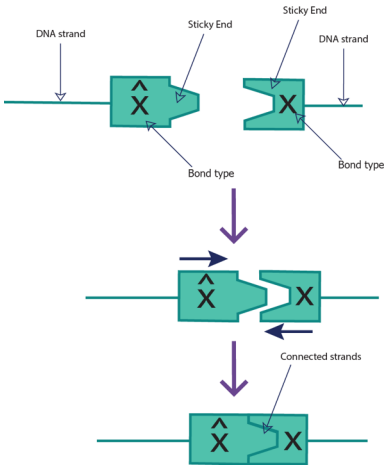


Figure 2: Complementary half-edges. [7]

Definition 2.1 We define a *tile* as a vertex with labeled half-edges. A *tile type* is a multiset of cohesive end types. Tiles are labeled according to type, as t_1, t_2, t_3, etc.

Definition 2.2 The number of half-edges stemming from a particular tile is referred to as that tile's *degree*.

Definition 2.3 We define a *pot* as a set of tile types, denoted by $P = \{t_1, t_2, ..., t_n\}$ (Figure 3). We use the notation of $\Sigma(P)$ to refer to the set of symbols from which we draw the hatted and unhatted letters. We describe the contents of a pot with $\#\Sigma(P)$ denoting the number of different bond-edge types in the pot, and $\#P$ denoting the number of distinct tile types in the pot, also known as the *size of the pot*.

A pot P *realizes* a graph G if some number of tiles of the tile types contained within it can bond with each other such that every vertex in G can be labeled with a tile type in P. Note that tile types can be used multiple times to construct a graph. If every half-edge of every tile bonds with another, that is, if every a in the graph can bond to an $\hat{a}$ and so on, then G is considered a *complete complex*. We label the edges of a complete complex with a bond-edge type, with the direction of the arrow pointing from an unhatted cohesive

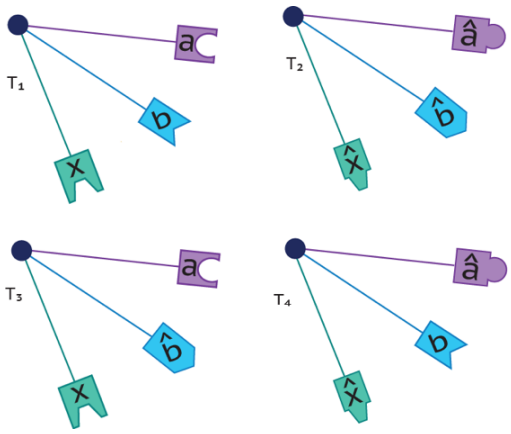


Figure 3: A pot of size 4: $P = \{\{x, b, a\}, \{\hat{x}, \hat{b}, \hat{a}\}, \{x, \hat{b}, a\}, \{\hat{x}, b, \hat{a}\}\}$ [7].

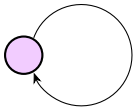


Figure 4: An example of a loop.

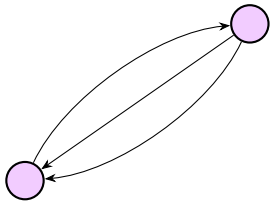


Figure 5: An example of a multi-edge.

end type to a hatted cohesive end type. We will only be considering complete complexes for the purposes of our research. The number of vertices in a realized graph G is defined as the *order* of G.

Definition 2.4 If a vertex shares an edge with itself, we refer to that edge as a *loop*.

Definition 2.5 If a vertex shares more than one edge with a separate vertex, we refer to those edges as *multi-edges*.

2.1.1 Laboratory Constraints: Scenarios 1, 2, and 3

In a laboratory setting, DNA self-assembly is largely an undirected process once it is initiated, and so there is a risk of unintended outcomes. Unbonded strands of DNA molecules have the potential to bond with any other proximal, complementary unbonded strands, whether they be from a new separate molecule, a molecule that has already been bonded with, or the same molecule if it contains both an unbonded strand and its complement [8]. As cohesive ends will only bond with their complement, it is possible for there to be a remaining half-edge that cannot find its corresponding complement. In other

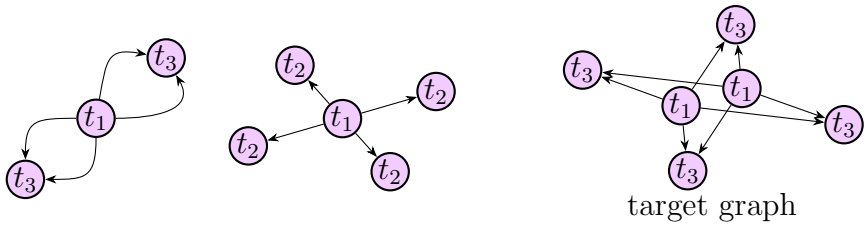


Figure 6: Acceptable graphs under Scenario 1 realized by pot $P = \{\{a^4\}, \{\hat{a}\}, \{\hat{a}^2\}\} = \{t_1, t_2, t_3\}$. Throughout this paper, arrows in figures point in the direction of unhatted towards hatted characters.

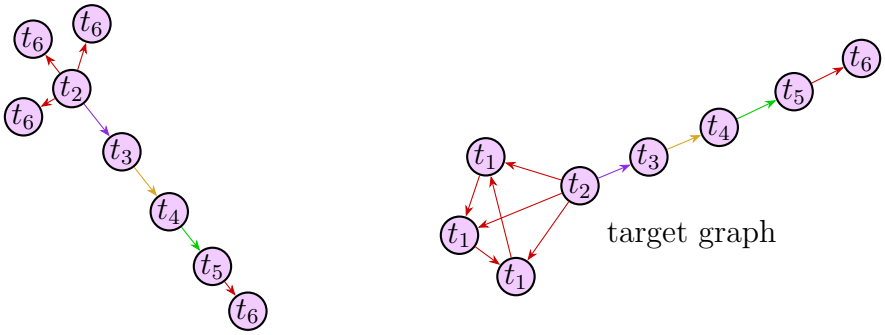


Figure 7: Acceptable graphs under Scenario 2 realized by pot $P = \{\{a, \hat{a}^2\}, \{a^3, b\}, \{\hat{b}, c\}, \{\hat{c}, d\}, \{\hat{d}, a\}, \{\hat{a}\}\} = \{t_1, t_2, t_3, t_4, t_5, t_6\}$. Throughout this paper, different colored arrows in figures represent different bond-edge types.

words, if caution is not exercised, the self-assembly process can result in the formation of smaller complexes than desired [8].

Considering this issue, when aiming to construct a target nanostructure with representative graph G of order O, there are three scenarios with various conditions and restrictions that may be considered, as described in [1]:

- *Scenario 1.* The incidental construction of a complex of order less than or equal to O is acceptable.
- *Scenario 2.* The incidental construction of a complex of greater order than O is acceptable, and the incidental construction of complexes of order O that are not isomorphic to G is acceptable.
- *Scenario 3.* The incidental construction of a complex is acceptable if and only if it satisfies either of these conditions:
  - The order of the complex is greater than O.
  - The order of the complex is O and the representative graph is isomorphic to G.

Figure 8: Acceptable graphs under Scenario 3 realized by the pot $P = \{\{a, b, c, d\}, \{\hat{a}^4\}, \{\hat{c}^4\}, \{\hat{a}, \hat{b}^2, c\}, \{a, \hat{a}, c, \hat{d}\}\} = \{t_1, t_2, t_3, t_4, t_5\}$.

Generally, complexes of a larger order than O are always allowed regardless of scenario because larger-than-minimal complexes appear infrequently [1].

Definition 2.6 The minimum number of tile types necessary to realize a target graph G under Scenario i is denoted $T_i(G)$ (see 2.1.1).

Definition 2.7 The minimum number of bond-edge types necessary to realize a target graph G under Scenario i is denoted $B_i(G)$ (see 2.1.1).

Considering these various scenarios, and their respective effects on $B_i(G)$ and $T_i(G)$, allows researchers to model DNA self-assembly depending on the desired laboratory results. More restrictive scenarios typically require more resources and precision in the lab, but offer greater accuracy and dependability. On the other hand, less restrictive scenarios often result in the construction of more unwanted DNA complexes incidentally, but are more accessible and less time-consuming to create [1]. This is evident in the results for the three scenarios in various graph families, such as book graphs [10], wheel graphs [11], and lollipop graphs [12].

The algorithm 3.1 can be used for any of the three scenarios. For Scenario 1, it can be used simply to show that a graph can indeed be realized by an inputted pot. For Scenarios 2 and 3, since the output of the algorithm shows all graphs that can be realized by the inputted pot, it can be tested whether the pot satisfies the scenario with respect to the target graph; this isomorphism problem is discussed further in Section 3.3.

2.2 Construction Matrices for Graphical Models

The following definition is from [1].

Definition 2.8 Let $P = \{t_1, t_2, t_3, ..., t_n\}$ be a pot. Let $z_{i,j} = u - v$ where u is the number of unhatted half-edges of type α_i on tile type t_j and v is the number of hatted half-edges of type $\hat{\alpha}_i$ on tile type t_j. Let m be the number of distinct bond-edge types in G, and let r_i be the proportion of tiles of type t_i used in constructing G.

Then for a complete complex, since each half-edge must be paired with its complement, we have the following system of equations:

$$\begin{aligned} z_{1,1}r_1 + z_{1,2}r_2 + ... + z_{1,n}r_n &= 0 \\ &\vdots \\ z_{m,1}r_1 + z_{m,2}r_2 + ... + z_{m,n}r_n &= 0 \\ r_1 + r_2 + ... + r_n &= 1 \end{aligned}$$

These equations can be represented as an augmented *construction matrix*, where each column corresponds to a tile t_j and each row corresponds to the bond-edge type α_i on each tile:

$$M(P) = \left(\begin{array}{cccc|c} t_1 & t_2 & ... & t_n & \\ z_{1,1} & z_{1,2} & ... & z_{1,n} & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ z_{m,1} & z_{m,2} & ... & z_{m,n} & 0 \\ 1 & 1 & ... & 1 & 1 \end{array}\right) \begin{array}{c} \\ \alpha_i \\ \vdots \\ \alpha_m \end{array} \tag{1}$$

Construction matrices, when in row-reduced echelon form, can be used to determine the minimum possible order of a graph realized by a given pot by finding the least common denominator of the values in the solution vector. The proportion of each tile type in the pot used to realize a graph is given by the respective components in the solution vector, with the first corresponding to t_1, the second t_2, and so on [7]. Finding these values is useful for determining the proportions of each tile type that are necessary to realize a graph, and to check whether graphs of smaller orders than a target order could be incidentally created from a particular pot.

Definition 2.9 Given a solution to this system of equations, we can deduce a viable *tile type multiplicity* for a pot, defined as a list of non-negative integers that determines the number of tiles to use of each tile type. If the tile type multiplicity we use is viable, we can construct a complete complex using the corresponding multiset of tiles.

We can find viable tile type multiplicities by multiplying a solution vector by a multiple of the least common denominator. When multiplying by exactly the least common denominator, the resulting list of integers is setwise coprime, and gives us a tile type multiplicity to construct a graph of minimal order for the given tile proportion.

If row reducing a construction matrix results in one or more free variables, it indicates that multiple proportions of tile types in the corresponding pot can form a complete complex [7]. Determining the minimum possible order and the proportions that are viable requires a particular process, which we describe in Section 3.2.1.

2.3 Graph Generation

Two algorithms, each introduced by Andrew Lavengood-Ryan [13], offer an approach to the problem of generating a complete complex given a pot of the form $\{\{a^m\}, \{a\}, \{\hat{a}^n\}\}$. These begin with either a path or a cycle consisting of a chain of tiles of type $\{a^m\}$ and $\{\hat{a}^n\}$ connected by a series of single bonds, leaving the other half-edges on each tile unbonded. The next steps systematically attach more bonds of these tiles to each other, as well as adding more tiles to the structure when necessary. By implementing this algorithm in Sage using a method similar to that in Lavengood-Ryan's earlier work [14], we gain insight into the process of algorithmically connecting tiles and visually representing the result. However, a limitation of Lavengood-Ryan's algorithms is that they can only generate graphs for pots with one bond-edge type. The algorithm 3.1 extends this prior work to allow for pots with more than one bond-edge type.

3 Results

3.1 Algorithm Overview

We provide an algorithm, illustrated in Figure 9, that takes a pot as input and outputs all possible non-isomorphic graphs that can be realized by that pot.

Algorithm:

In particular, the algorithm 3.1 takes the following user-specified parameters:

- A pot in the form of a string (for example, $\{\{a, \hat{b}\}, \{a, b\}, \{\hat{a}^2, \hat{b}\}, \{\hat{a}^2, b\}\}$ would be written as '$a, -b; a, b; -a2, -b; -a2, b$') with hatted letters preceded by a negative, and exponents (if applicable) listed after each letter that is being exponentiated, except for letters having an exponent of 1.

- A desired order for a target graph, which can be preceded with a negative to indicate the production of graphs of only that order (or the next smallest if the entered order is not applicable), or 0 if the user wants the program to first calculate the minimum possible order and then generate all graphs of that order.

- A number of parameters that may affect graph generation. Each parameter can be specified independently, allowing for any combination of settings:

  - `avoid_loops`
    - Prioritizes generating graphs without loops
  - `avoid_multiple_edges`
    - Prioritizes generating graphs without multiple edges
  - `ordering_to_use`
    - Specifies the tile type ordering to use (as in Section 3.2.2)
  - `order`

    - Forces the algorithm to only display graphs of a particular, inputted order (as a negative value)

- `choose_tile_ratios`
    - Allows the user to choose certain viable tile type multiplicities used in the graph generation and only generates graphs that adhere to those proportions

With these specified inputs and parameters, the algorithm first analyzes the pot (Section 3.2) to find the orders of possible graphs that can be generated and find all possible tile type multiplicities that can be used to generate a graph of the corresponding order (Section 3.2.1). Then, the algorithm uses a strategy of ordering and connecting tiles (Section 3.2.2), together with the management of multi-edges and loops (Section 3.2.3), to construct a graph. Finally, from the constructed graph, the algorithm generates various arrangements of the edges of the graph, identifying the non-isomorphic graphs using canonical labeling (Section 3.3). With some visualization techniques (Section 3.4), the algorithm finally returns all non-isomorphic graphs that can be generated from the inputted pot. In the following sections, we provide more details and an in-depth explanation of this process.

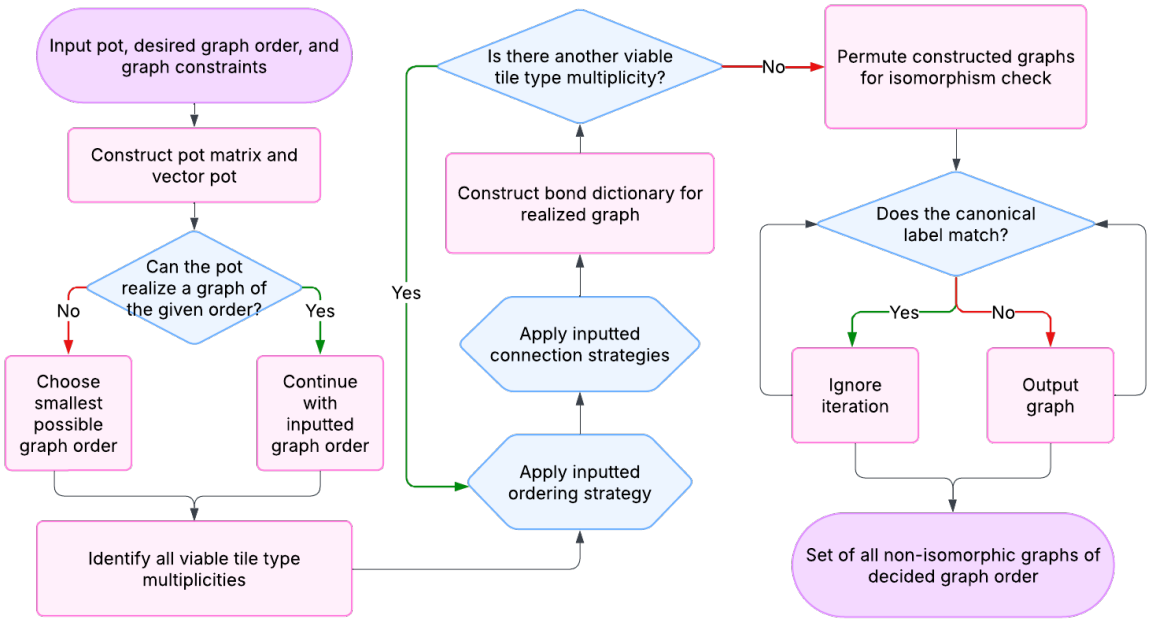


Figure 9: Flow chart overview of the algorthim

3.2 Analyzing the Pot

When given a pot, the algorithm is given the task of reading the pot and converting it into data that can be used to generate the graphs. The algorithm does this by converting the pot into a numerical representation called a pot matrix. This step is important for generating the viable tile type multiplicities.

Definition 3.1 A *pot matrix* is similar to a construction matrix, with some notable differences: To construct a pot matrix for a pot $P = \{t_1, t_2, t_3, ..., t_n\}$, let $z_{i,j} = u_{i,j} - v_{i,j}$ where $u_{i,j}$ is the number of unhatted half-edges of type α_i on tile t_j and $v_{i,j}$ is the number of hatted half-edges of type α_i on tile t_j. Let m be the number of distinct bond-edge types in P.

Then we have the following system of expressions:

$$z_{1,1}r_1 + z_{1,2}r_2 + ... + z_{1,p}r_p$$

$$\vdots$$

$$z_{m,1}r_1 + z_{m,2}r_2 + ... + z_{m,p}r_p$$

These expressions can be represented as a *pot matrix*, where the typical values in the construction matrix are used, but without the last row of 1's and the last column of 0's:

$$\begin{matrix} t_1 & t_2 & ... & t_n \end{matrix} \\ \begin{pmatrix} z_{1,1} & z_{1,2} & ... & z_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ z_{m,1} & z_{m,2} & ... & z_{m,n} \end{pmatrix} \begin{matrix} a_i \\ \vdots \\ a_m \end{matrix} \tag{2}$$

The inputted pot is also translated into a quantitative format known as a *vector pot*, which is a list of lists encoding the tile types in the pot.

Definition 3.2 A *vector pot* is a quantitative representation of a pot used for graph construction and visualization. It is a 2D array that provides the number of each hatted and unhatted letter for each tile type, making the dimensions of the vector pot $2(\#\Sigma(P)) \times \#P$, where $\#\Sigma(P)$ is the number of bond-edge types and $\#P$ is the number of tile types in the pot. To construct a vector pot, let $P = \{t_1, t_2, t_3, ..., t_n\}$ be a pot and let each vector correspond to each tile type. The numbers in each vector in the vector pot are ordered by $a, \hat{a}, b, \hat{b}, etc$ and quantitatively represent how many half-edges of each cohesive end type there are in a tile type.

Example 3.3 Given the pot $P = \{\{a, \hat{a}^2\}, \{a^3, b\}, \{\hat{b}, c\}, \{\hat{c}, d\}, \{a, \hat{d}\}, \{\hat{a}\}\}$, a pot which realizes an order 8 lollipop graph, the vector pot would be:

$$[1, 2, 0, 0, 0, 0, 0, 0],$$
$$[3, 0, 1, 0, 0, 0, 0, 0],$$
$$[0, 0, 0, 1, 1, 0, 0, 0],$$
$$[0, 0, 0, 0, 0, 1, 1, 0],$$
$$[1, 0, 0, 0, 0, 0, 0, 1],$$
$$[0, 1, 0, 0, 0, 0, 0, 0]$$

This format of the pot allows the program to easily read the number of tile types and the number of each cohesive end type in each tile type. Looking at the first vector, it has a length of eight, meaning that $\#\Sigma(P) = 8/2 = 4$, and we know there is one a and two $\hat{a}$ in the first tile type. This format allows for easy access of the attributes of the pot. We use the vector pot to efficiently store and manage information about the pot in the algorithm, but it will not be used in any mathematical proofs and will only be referred to as a *vector pot*.

After the pot has been analyzed, the algorithm will have all necessary formats and information to continue to the next part of the algorithm, finding the viable tile type multiplicities for the pot using the pot matrix.

3.2.1 Finding Proportions

While the problem of determining whether there is a graph of a given order that can be realized by a given pot is known to be NP-hard, progress has been made by restricting the problem to the case where the construction matrix has two or fewer free variables [7]. For this case, the algorithm by Almodóvar et al. [7] uses the construction matrix to determine if a smaller graph or graph of the same order can be realized by a pot of tiles. When there are no free variables, the algorithm outputs the least common denominator of the solution vector, giving the minimum possible order of a realized graph. Any multiple of the least common denominator is also a viable order. When there are one or two free variables, the algorithm must explore many possible solutions, which significantly increases its time complexity [7]. For three or more free variables, the algorithm's fast-growing time complexity makes it impractical for use beyond very small examples. An implementation of this algorithm handling any number of free variables can be found in [15].

Since the situation where a pot's construction matrix has three or more free variables is common, we use SciPy to run an integer linear program to find viable tile type multiplicities. In this program, the total number of tiles used to generate the graph serves as the objective function to be minimized, and the construction matrix gives us our constraints. This part of our code was in part inspired by similar linear programs used in a related project [15].

3.2.2 Ordering Tile Types

After identifying viable tile type multiplicities, we can impose an order onto the tile types to facilitate generating possible graphs. It is important to explore different orderings because each unique arrangement of the tile types can lead to the generation of different graphs. This is valuable for our study, as it allows for the possibility to obtain non-isomorphic graphs simply from tile type re-ordering.

We define the following three possible orderings.

- degree: puts tile types in decreasing order by degree (Definition 2.2)
- diversity: puts tile types in increasing order based on the number of different cohesive end types (e.g. $(\{a\}, \{\hat{a}, b, \hat{b}\})$ has tile types of diversity 1 then diversity 3)
- lexicographic: orders tile types lexicographically, that is, t_1 comes before t_2 when t_1 has more of the first cohesive end type for which they differ. The order in which to compare the cohesive end types is given by each letter in the alphabet, followed by its corresponding hatted letter (e.g. $a, \hat{a}, b, \hat{b} \dots$).

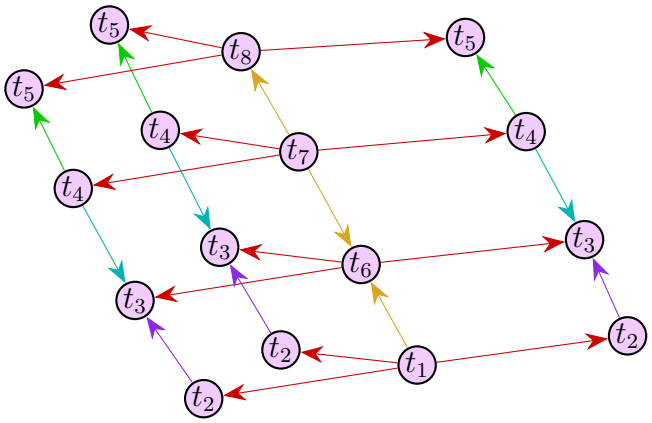
Note that if there are multiple tile types that have the same degree or diversity, then the order in which the user inputted the tile types is used as a tiebreaker in the total order. Each of these possible orderings can be used to generate a valid graph, but there is no guarantee that two orderings will result in different graphs. Below we provide some example outputs from a given pot based on the ordering imposed.

Example 3.4 Given a pot $P = \{t_1 = \{a^3, c\}, t_2 = \{\hat{a}, b\}, t_3 = \{\hat{a}, \hat{b}, \hat{e}\}, t_4 = \{\hat{a}, d, e\}, t_5 = \{\hat{a}, \hat{d}\}, t_6 = \{a^3, \hat{c}^2\}, t_7 = \{a^3, c^2\}, t_8 = \{a^3, \hat{c}\}\}$, we can see that each ordering, even when using the same strategy to connect tiles as described in 3.2.3, generates a graph that appears different, but may or may not be unique up to isomorphism.

1.) Ordering by degree, the list of tile types becomes

$$(\{a^3, \hat{c}^2\}, \{a^3, c^2\}, \{a^3, c\}, \{a^3, \hat{c}\}, \{\hat{a}, \hat{b}, \hat{e}\}, \{\hat{a}, d, e\}, \{\hat{a}, b\}, \{\hat{a}, \hat{d}\})$$

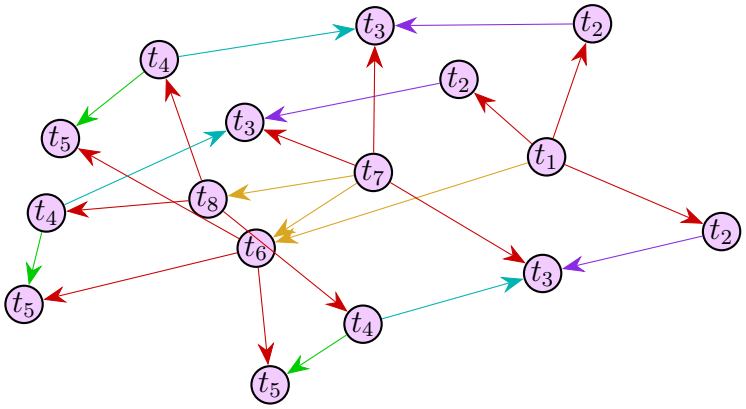
This ordering generates the following graph:



2.) Ordering by diversity, the list of tile types becomes

$$(\{a^3, c\}, \{\hat{a}, b\}, \{\hat{a}, \hat{d}\}, \{a^3, \hat{c}^2\}, \{a^3, c^2\}, \{a^3, \hat{c}\}, \{\hat{a}, \hat{b}, \hat{e}\}, \{\hat{a}, d, e\})$$

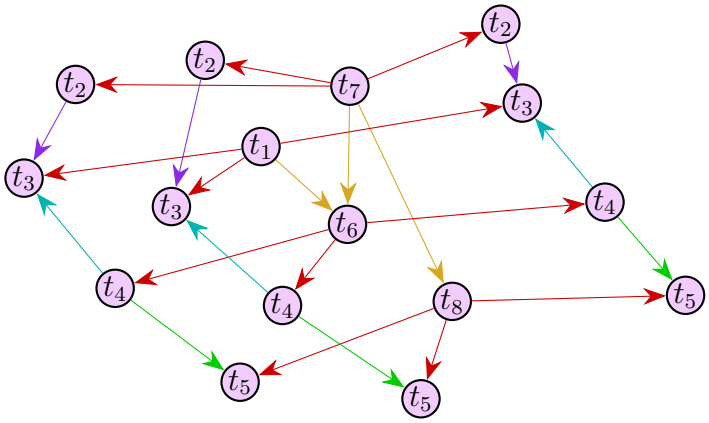
This ordering generates the following graph:



3.) Ordering lexicographically, the list of tile types becomes

$$(\{a^3, c^2\}, \{a^3, c\}, \{a^3, \hat{c}^2\}, \{a^3, \hat{c}\}, \{\hat{a}, b\}, \{\hat{a}, \hat{b}, \hat{e}\}, \{\hat{a}, d, e\}, \{\hat{a}, \hat{d}\})$$

This ordering generates the following graph:



3.2.3 Connecting Tiles

Given a tile type ordering, we propose four strategies to connect the tiles. The algorithm 3.1 can use any one of these strategies to connect the tiles.

- avoid multi-edges and loops
- allow multi-edges
- allow loops

- allow multi-edges and loops

Example 3.5 Given a lexicographically ordered pot $P = (\{a, \hat{a}, \hat{b}, \hat{c}\}, \{a, c\}, \{\hat{a}^2, c^4\}, \{b^2, \hat{c}\})$ with tile type multiplicity $\langle 4, 2, 1, 2\rangle$, we show the beginning steps of connecting tiles using lexicographic order and the **avoid multi-edges and loops** strategy.

First, we write a new list where each tile type is repeated according to its multiplicity, so that each tile type from the pot appears exactly as many times as indicated by its corresponding multiplicity (as shown in Table 1).

tiles:
$\{a, \hat{a}, \hat{b}, \hat{c}\}$
$\{a, \hat{a}, \hat{b}, \hat{c}\}$
$\{a, \hat{a}, \hat{b}, \hat{c}\}$
$\{a, \hat{a}, \hat{b}, \hat{c}\}$
$\{a, c\}$
$\{a, c\}$
$\{\hat{a}^2, c^4\}$
$\{b^2, \hat{c}\}$
$\{b^2, \hat{c}\}$

Table 1: List of tiles made from the pot in Example 3.5, where each tile type from the pot appears exactly as many times as indicated by its corresponding multiplicity.

We start with the first tile in our list (Table 1), $\{a, \hat{a}, \hat{b}, \hat{c}\}$. Note that the first cohesive end type is a, so as shown in Figure 10, our first step is to search for an $\hat{a}$ to bond with the a by looking through the rest of the cohesive end types in the current tile (if loops are allowed); if it does not find a match, it continues to the other tiles. In this example, we are avoiding loops, so we do not want a to bond with the $\hat{a}$ in the first tile since this would create a loop. Instead, the algorithm moves to the second tile in Table 1. Note that the first and second tiles are different tiles but of the same type, so they are both labeled as t_1 in Figure 10. We see in Step 2 of 10 that the algorithm finds an $\hat{a}$ from the second tile to bond with the a from the first tile.

Next, the algorithm continues with the first tile and determines the next half-edge that has not been bonded. In this example, Step 3 of Figure 10 shows that the next half-edge has cohesive end type $\hat{a}$.

Again, the algorithm searches for a half-edge to bond with the $\hat{a}$. However, the algorithm has already created an edge with a half-edge from the second tile. Since we are avoiding multi-edges in this example, the algorithm should not create another edge between the first and second tile. So, the algorithm continues to the third tile, which is a different tile but is still labeled t_1 because it is of the same type as the first two tiles. Step 4 of Figure 10 shows that it finds an a to bond with the $\hat{a}$ from the first tile. This process continues until the graph is realized as shown in Figure 11.

To summarize, the algorithm starts at the first tile. It goes down the list of tiles, starting at the next tile because we are avoiding loops, to try to find at most one bond per tile before moving on in order to avoid multi-edges. It continues this process until all the half-edges of the first tile are bonded. Then it goes to the second tile and repeats this process.

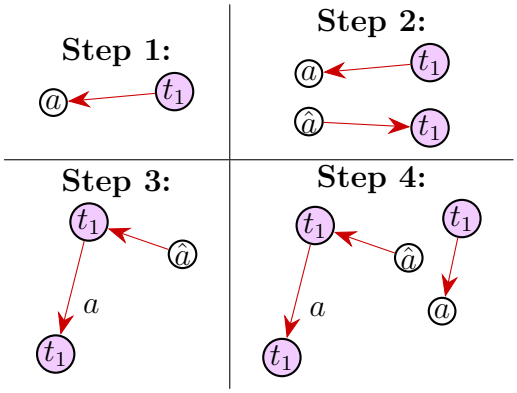


Figure 10: First 4 steps of the avoid multi-edges and loops strategy using tiles from Table 1.

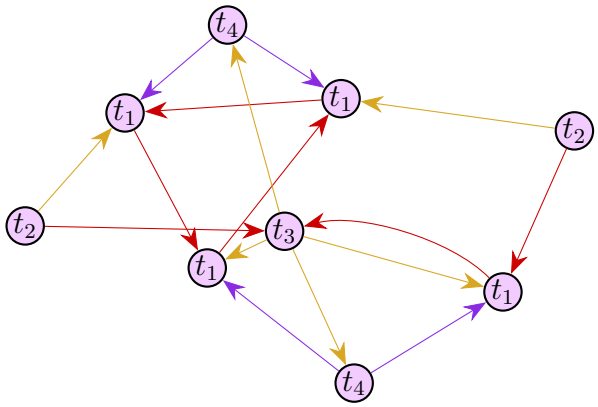


Figure 11: Graph realized by continuing the avoid multi-edges and loops strategy started in Figure 10, using tiles from List 1.

Example 3.6 Continuing with the same pot, we now show an example of connecting the tiles using the **allow multi-edges** strategy.

We start with the first tile in our list (Table 1), $\{a, \hat{a}, \hat{b}, \hat{c}\}$. Note that the first cohesive end type is a, so as shown in Figure 12, our first step is to search for an $\hat{a}$ to bond with the a. We are avoiding loops, so we do not want any of the half-edges to bond with each other, i.e. the a and $\hat{a}$ cannot bond with each other.

Now since we are allowing multi-edges, the algorithm goes to the second tile to try to find anything that can bond with the half-edges in the first tile. Like the previous example, note that the second tile is a different tile but is still labeled t_1 because it has the same tile type as the first tile. As shown in Step 2 of 12, it finds the cohesive end type $\hat{a}$ of the second tile, so that bonds with the a in the first tile.

Next, the second cohesive end type of the first tile is an $\hat{a}$. Since we are allowing multi-edges, Steps 3 and 4 of Figure 12 show that this $\hat{a}$ is allowed to bond with the a from the second tile. If multi-edges had not been allowed, then the algorithm would have tried to go to the third tile to find a half-edge to bond with the $\hat{a}$ from the first tile.

The second tile no longer has any other half-edges that can bond with the half-edges in the first tile. So, the algorithm goes to the third tile. The third tile and fourth tile do not have any half-edges that can bond with the remaining unbonded $\hat{b}$ and $\hat{c}$ in the first tile. So the algorithm goes to the fifth tile: $\{a, c\}$, which is labeled t_2 because it is a different tile type from the first four tiles. As shown in Step 4 of Figure 12, it sees that the c in the fifth tile can bond with the $\hat{c}$ in the first tile. This process continues until the graph is realized, which is shown in Figure 13.

To summarize, the algorithm starts at the first tile and goes through the rest of the list, starting at the second tile, until all the half-edges of the first tile are bonded. Starting the search at the next tile helps avoid loops. Then it moves to the second tile and goes through the rest of the list until all of its half-edges are bonded. This process makes it possible for multi-edges to exist because two half-edges of one tile could bond with two half-edges from a different single tile.

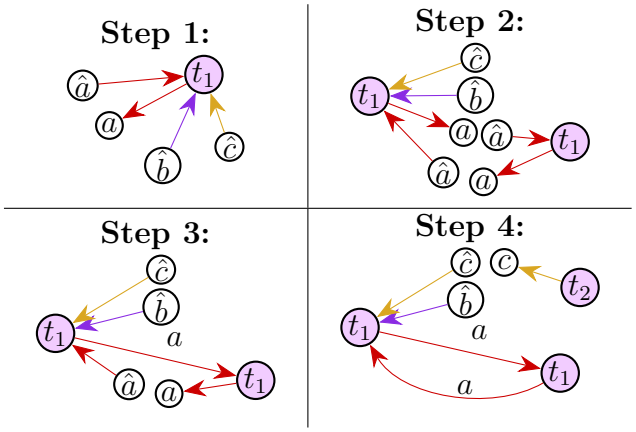


Figure 12: First 4 steps of allow multi-edge strategy using tiles from Table 1.

The **allow loops** strategy starts at the first tile and allows any of its half-edges to bond with each other. If there are half-edges that still aren't bonded, then it goes through the rest of the tile list until all the half-edges of the first tile are bonded. Note that this strategy explicitly avoids multi-edges, so during this process, a half-edge can bond with at most one half-edge of a given tile in order to avoid multi-edges. Then, after all the

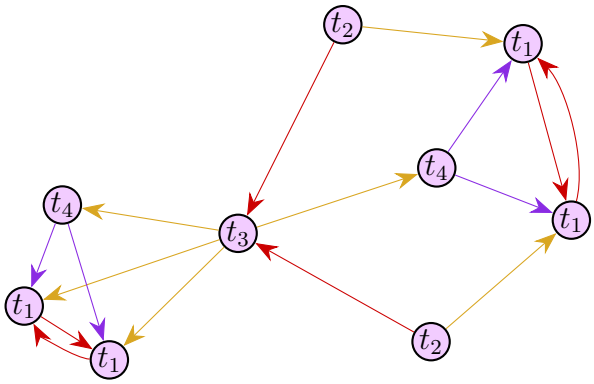


Figure 13: Graph realized by continuing the allow multi-edges strategy started in Figure 12, using tiles from Table 1.

half-edges of the first tile are bonded, it goes to the second tile, allows any of its half-edges to bond with each other, and goes through the rest of the list until all the half-edges of the second tile are bonded. This process continues. Figure 14 and Figure 15 show the first few steps and resulting graph, respectively.

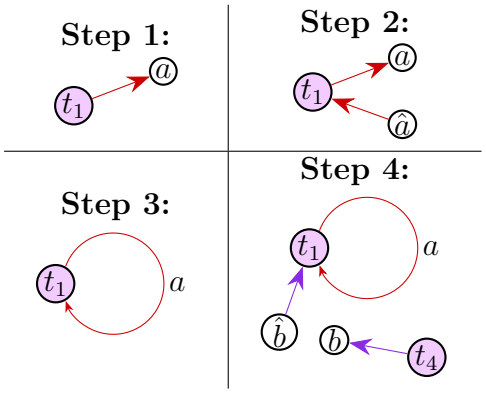


Figure 14: First 4 steps of allow loops strategy using tiles from Table 1.

The **allow multi-edges and loops** strategy starts at the first tile and allows any of its half-edges to bond with each other, allowing for loops. If there are half-edges that still aren't bonded, then it goes through the rest of the tile list until all the half-edges of the first tile are bonded. Note that two half-edges from the first tile are allowed to bond with more than one half-edge of one given tile. Then it goes to the second tile, allows any of its half-edges to bond with each other, and goes through the rest of the list until all the half-edges of the second tile are bonded while allowing for multi-edges.

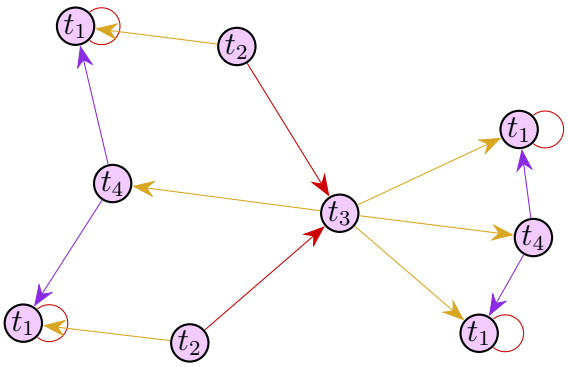


Figure 15: Graph realized by continuing the allow loops strategy started in Figure 14, using tiles from Table 1.

3.3 Isomorphism

Given a pot and a list of viable tile type multiplicities, the algorithm 3.1 uses a tile type ordering and a strategy for connecting tiles to realize a possible graph. However, there are multiple choices for the ordering and for the connecting strategy. While these differing choices may sometimes give a unique graph in each instance, it is not guaranteed that each differing choice will produce a unique graph up to isomorphism. Further, it is not guaranteed that we will obtain all possible graphs that can be realized by the pot by varying the connection strategies. We would like to ensure that we can obtain all possible unique graphs, which is essential for determining whether a pot satisfies Scenario 2 or 3. Recall from Section 2.1.1, Scenario 2 allows for any graph of the same order as the target graph, and Scenario 3 only allows for the target graph and graphs of higher order. Thus, to cover all cases, we enumerate all possible unique realized graphs of a given order up to isomorphism. In particular, in Section 3.3.1, we employ canonical labeling to establish a consistent method for encoding graphs and determining the non-isomorphic ones. Before identifying non-isomorphic graphs, we first generate all possible graphs by considering edge permutations, as discussed in Section 3.3.2. The following sections detail the process of generating all potential graphs by permuting edges and then using canonical labeling to systematically identify the non-isomorphic ones.

Definition 3.7 An *isomorphism* between two graphs G and H is a bijection $f\colon V(G) \to V(H)$ such that $(u, v) \in E(G)$ if and only if $(f(u), f(v)) \in E(H)$ for all $u, v \in V(G)$. In other words, there exists a map between the vertices in graph G and those in graph H such that two vertices are connected by an edge in G if and only if their corresponding vertices are connected by an edge in graph H. When two graphs G and H are isomorphic, we denote this by $G \cong H$ [16].

3.3.1 Canonical Labeling

Given two graphs, we can determine if they are isomorphic by looking at their canonical graphs. First, we define a *partition* as a coloring of vertices that allows us to partition the set of vertices into disjoint subsets. An *equitable partition* is a coloring of the graph such that vertices of the same color (partition) are adjacent to the same number of vertices of each color, and is done in a way that uses the least number of colors [17]. This process of subdividing cells (sets of vertices of the same color) to get to the equitable partition is known as *partition refinement* [17].

This first equitable partition, which we call our initial refined coloring, now serves as the root of a search tree (as shown in Figure 16). The root node has associated with it an empty sequence. The children of each node in the tree append onto the sequence a vertex from a specified cell given by a target cell selector function. The selector function uses a further refinement of the coloring on each child node to determine the next target cell [17]. The final sequence given by each leaf in the tree determines a labeling of the graph. Given this search tree, one can define an order on the labeled graphs (e.g., lexicographic), and the greatest labeled graph is called the *canonical label*. For any graph G, its canonical graph is denoted by $c(G)$ [17].

Theorem 3.8 *Two graphs are isomorphic, that is $G_1 \cong G_2$, if and only if their canonical graphs are identical, that is $c(G_1) = c(G_2)$. [17, 18]*

3.3.2 Permutations of Graphs

Before identifying the non-isomorphic graphs, we must first generate all possible graphs from a given pot. Once all the graphs are generated, we can apply Theorem 3.8 to identify which graphs are non-isomorphic.

To make the graph generation process more efficient, we introduce a *bond dictionary* 3.9 that stores and tracks edges and vertices, allowing the algorithm to access this information quickly and easily.

Definition 3.9 We can represent a graph G with a *bond dictionary*. This bond dictionary reports the edges that correspond to each bond-edge type that is used. The ordering of the vertex indices within each edge represents the direction in which the unhatted half-edge points to the hatted half-edge. In a theoretical sense, each set of edges of the same bond-edge type is unordered, but in the algorithm their order emerges from the order in which they are constructed by the steps in 3.2.3. This order remains fixed, since the edges are stored in an array.

Example 3.10 Given a pot $P = \{\{a, \hat{b}\}, \{a, b\}, \{\hat{a}^2, \hat{b}\}, \{\hat{a}^2, b\}\}$, one possible graph G that is realized has the bond dictionary

$$\text{dict}(G) = \{a\colon [1,2], [3,2], [4,5], [6,5], b\colon [4,1], [2,5], [6,3]\},$$

as shown in Figure 17. In the figure we can see that the edges $[1,2], [3,2], [4,5], [6,5]$ have bond-edge type a, and edges $[4,1], [2,5], [6,3]$ have bond-edge type b.

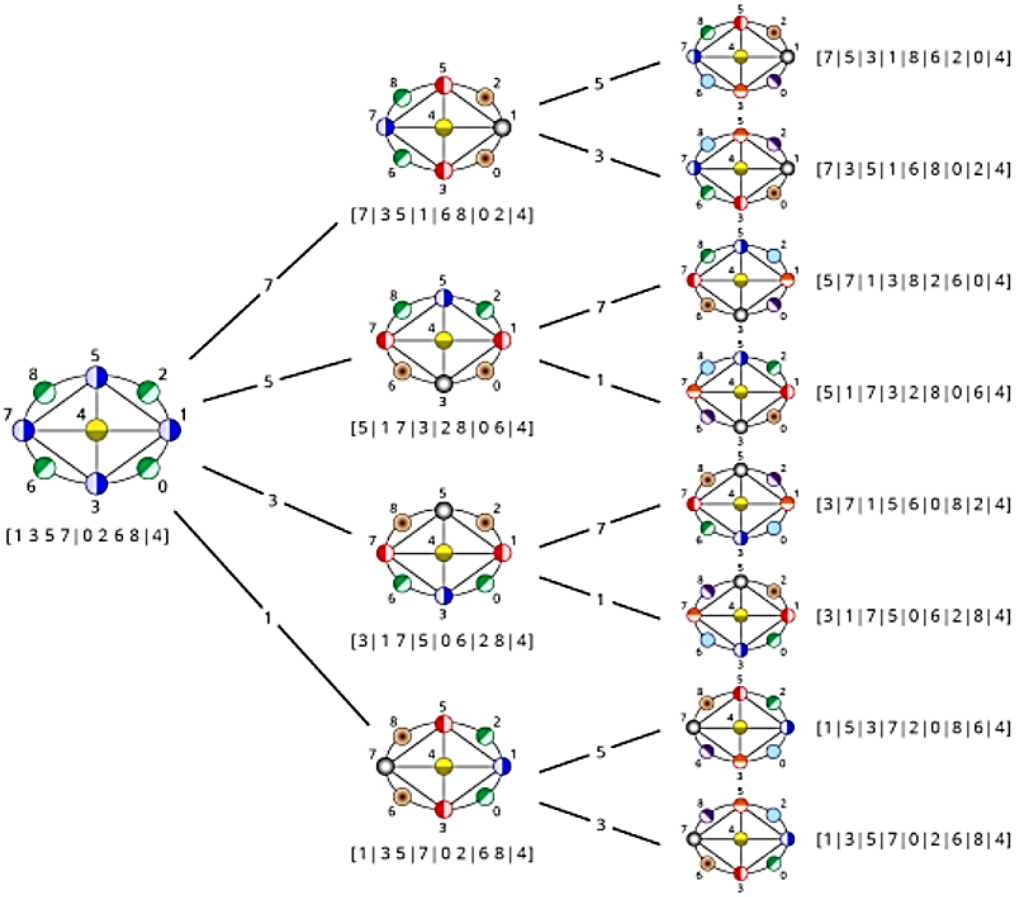


Figure 16: Search tree used to find the canonical labeling of a graph [17].

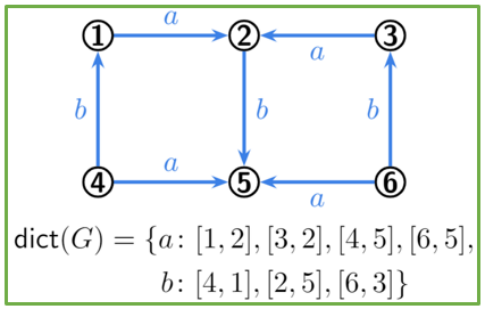


Figure 17: Example of the bond dictionary dict(G) of a possible graph G realized by pot $P = \{\{a, \hat{b}\}, \{a, b\}, \{\hat{a}^2, \hat{b}\}, \{\hat{a}^2, b\}\}$.

Given a pot and a viable tile type multiplicity determined by the process outlined in Section 3.2.1, we can construct one possible graph as we described in Sections 3.2.2 and 3.2.3. This graph can then be represented using a bond dictionary, which allows us to

explore all possible permutations of edge swaps on the graph. We consider three methods for obtaining these permutations.

Method 1: Brute Force Edge Swapping Algorithm

Given the bond dictionary of one possible graph realized by a pot, since the number of edges and their bond-edge type remain fixed, we can find all other possible graphs realized by the given pot with the same tile type multiplicity by permuting the edges. This process can then be repeated for all other viable tile type multiplicities.

To see how we can encode edge swapping, let's use Example 3.10. To understand *swaps*, let's focus our attention on edges of one bond-edge type by determining possible swaps of edges of type b. First, consider swapping edges $[4,1]$ and $[2,5]$. We switch their second components to get $[4,5]$ and $[2,1]$. This gives us the graph G' with bond dictionary

$$\text{dict}(G') = \{a\colon [1,2],[3,2],[4,5],[6,5], b\colon [4,5],[2,1],[6,3]\}.$$

This first process is shown in Figure 18. Next, consider swapping the edges $[2,1]$ and $[6,3]$ to get $[2,3]$ and $[6,1]$. This gives us graph G'' with bond dictionary

$$\text{dict}(G'') = \{a\colon [1,2],[3,2],[4,5],[6,5], b\colon [4,5],[2,3],[6,1]\},$$

shown in Figure 19.

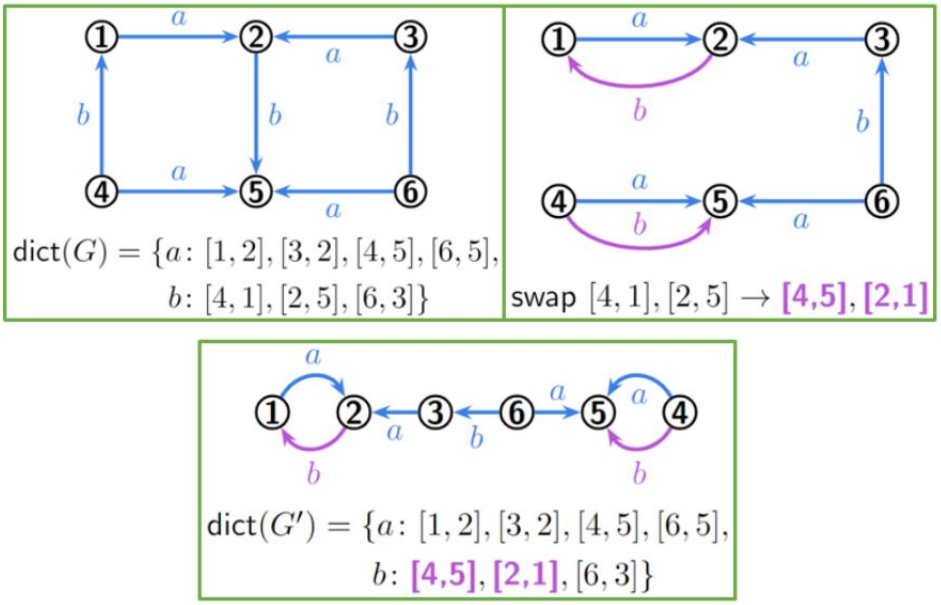


Figure 18: Showing the procedure of starting at graph G and its bond dictionary from Figure 17, then swapping edges $[4,1]$ and $[2,5]$, then the swap resulting in graph G' which its bond dictionary $\text{dict}(G')$.

We represent any number of swaps of edges of the same bond-edge type using a *single bond-edge type permutation*, where the value of the element at each position in the permutation represents which new target vertex the edge of each source points to.

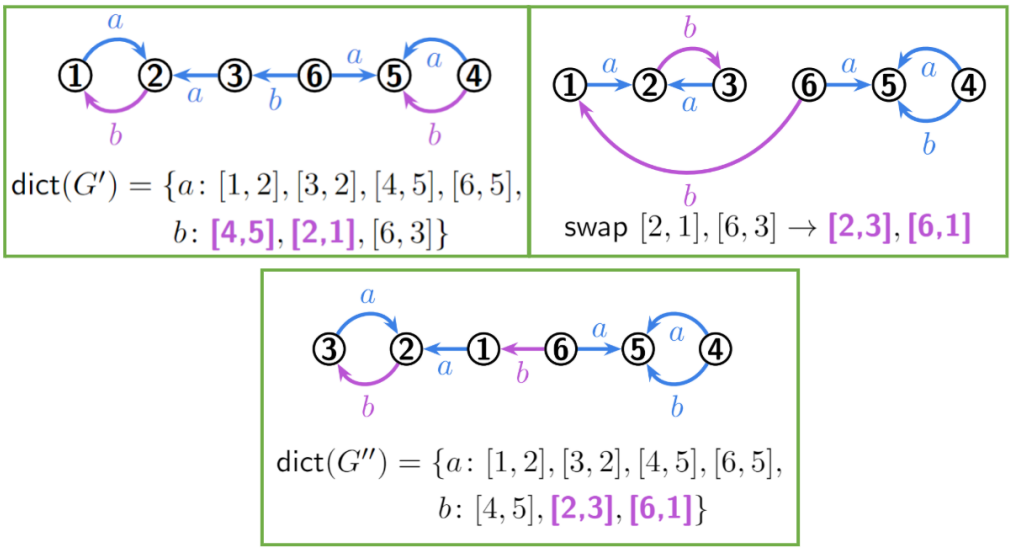


Figure 19: Showing the procedure of starting at graph G' and its bond dictionary from Figure 18, then swapping edges $[2, 1]$ and $[6, 3]$, then this swap resulting in graph G'' which its bond dictionary dict (G'').

Following our examples in Figures 18 and 19, the composition of these two swaps can be represented as the permutation $(2, 3, 1)$, since the source of the original 1^{st} edge is now connected to the target of the original 2^{nd} edge, the source of the original 2^{nd} edge is now connected to the target of the original 3^{rd} edge, and the source of the original 3^{rd} edge is now connected to the target of the original 1^{st} edge. This process is shown in Figure 20.

For all edges of a given bond-edge type, we can consider all possible single bond-edge type permutations. Then the Cartesian product of all sets of single bond-edge type permutations gives us all possible edge swappings of our original graph. Since we can enumerate the swappings in this way, we can simply count them to find an upper bound on the number of resulting non-isomorphic graphs.

Proposition 3.11 *Given a graph G realized by the ordered pot $P = (t_1, t_2, \ldots, t_n)$ and a tile type multiplicity $r = (r_1, r_2, ..., r_n)$, if the pot has bond-edge types $\Sigma(P) = \{a, b, c, \ldots\}$, we have an upper bound on the number of graphs up to isomorphism that can be realized by P with r:*

$$\textit{number of possible graphs} \leq \prod_{\alpha \in \Sigma(P)} e_\alpha(G)!$$

where $e_\alpha(G)$ is the total number of edges of type α in G.

Proof. In the worst case, each distinct permutation of edge swaps produces a unique graph up to isomorphism. We consider this case to determine the upper bound on the number of unique graphs.

Given any bond-edge type $\alpha \in \Sigma(P)$, there are $e_\alpha(G)$ edges of type α in G, and thus there are $e_\alpha(G)$ edges of type α in the bond dictionary of G. Then the permutations

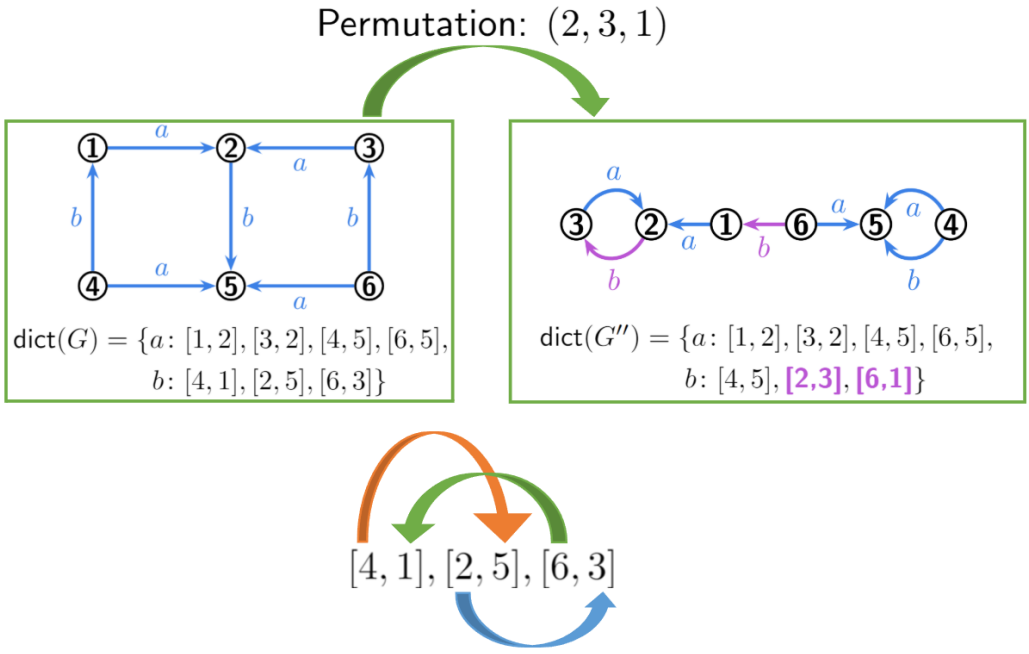


Figure 20: The process of one swap illustrated in Figure 18 followed by another swap illustrated illustrated in Figure 19 can be represented as a permutation $(2, 3, 1)$.

of these edges are represented by permutations of length $e_\alpha(G)$. Thus, there are $e_\alpha(G)!$ distinct permutations of edges of type α. Then the number of elements of the Cartesian product of these sets of single bond-edge type permutations is given by the product of the number of elements of each set. □

Example 3.12 Consider the pot $P = (\{a, b\}, \{a, \hat{b}\}, \{\hat{a}^2, b\}, \{\hat{a}^2, \hat{b}\})$ with tile type multiplicity $r = (2, 2, 1, 1)$. Let G be the graph from Example 3.10 which can result from P with r. Then

$$\begin{aligned} \text{number of possible graphs} &\leq (e_a(G)!)(e_b(G)!) \\ &= (4!)(3!) \\ &= 144. \end{aligned}$$

Method 2: Accounting for Redundancy

Method 1 may involve redundant edge swaps, such as swapping two edges that share a source, which results in the same graph. By considering the number of bond-edges of the same type on each tile, we can reduce the bound of Method 1.

Proposition 3.13 *Given a graph G realized by the ordered pot $P = (t_1, t_2, \ldots, t_n)$ with n tiles and the proportion of tiles $r = (r_1, r_2, ..., r_n)$, if the pot has bond-edge types $\Sigma(P) = \{a, b, c, \ldots\}$, we have*

$$\textit{number of possible graphs} \leq \prod_{\alpha \in \Sigma(P)} \frac{e_\alpha(G)!}{deg_\alpha(v_1)!deg_\alpha(v_2)! \cdots deg_\alpha(v_k)!}$$

where the v_i's are the vertices of G, and $\deg_\alpha(v_i)$ is the number of bond-edges of type α with v_i as their source.

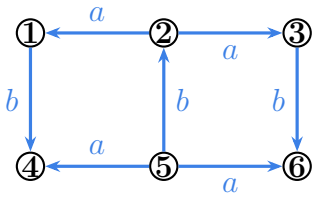
Proof. Given a graph G, consider a vertex v of G, with tile type $t = \{\alpha_1^{d_1}, \hat{\alpha}_1^{d_1'} \ldots, \alpha_n^{d_n}, \hat{\alpha}_n^{d_n'}\}$. Fixing an i, we have $\deg_{\alpha_i}(v) = d_i$. Let $e_i = e_{\alpha_i}(G)$. Then in $\mathrm{dict}(G)$, there are e_i edges under the α_i heading that can be swapped with each other. Out of these e_i edges, there are d_i edges all stemming from the same vertex v. Swapping any of these d_i edges with each other leaves exactly the same graph. Thus, we will want to consider any single bond-edge type permutations that differ only by swaps of these edges to be equivalent. From this, we change the count in our upper bound by dividing out by all the ways these edges can be swapped. This amounts to the number of permutations of those edges, which there are $d_i!$ of. Repeating this for each vertex gives the final formula. □

Example 3.14 Consider the pot $P = (\{\hat{a}, b\}, \{\hat{a}, \hat{b}\}, \{a^2, b\}, \{a^2, \hat{b}\})$ with tile type multiplicity $r = (2, 2, 1, 1)$. Then given a resulting graph G, shown in Figure 21, the number of possible realized graphs is bounded by

$$\left(\frac{e_a(G)!}{\deg_a(v_2)!\deg_a(v_5)!}\right)\left(\frac{e_b(G)!}{\deg_b(v_1)!\deg_b(v_3)!\deg_b(v_5)!}\right) = \left(\frac{4!}{(2!)(2!)}\right)\left(\frac{3!}{(1!)(1!)(1!)}\right) = 36.$$

This improved bound allows us to check a smaller total number of edge swaps. However, to actually construct those edge swaps in the algorithm, we need a different process. To not generate any redundant permutations, we designate a canonical form for a single bond-edge type permutation, as in the next example.

Example 3.15 Consider the same pot $P = (\{\hat{a}, b\}, \{\hat{a}, \hat{b}\}, \{a^2, b\}, \{a^2, \hat{b}\})$ with tile type multiplicity $r = (2, 2, 1, 1)$. This pot, with r, realizes the graph G, depicted in Figure 21.



$\mathrm{dict}(G) = \{a\colon [2,1], [5,4], [2,3], [5,6],$
$b\colon [1,4], [3,6], [5,2]\}$

Figure 21: One possible graph realized by pot $P = \{\{\hat{a}, \hat{b}\}, \{\hat{a}, b\}, \{a^2, \hat{b}\}, \{a^2, b\}\}$

Focusing on the bond-edge type a as an example, we assign an index to each edge under the a heading in $\mathrm{dict}(G)$. The index 1 refers to $[2, 1]$, the index 2 refers to $[5, 4]$, the index 3 refers to $[2, 3]$, and the index 4 refers to $[5, 6]$. Using those indices, we can note that

the permutations $(1,3,2,4)$, $(2,3,1,4)$, $(1,4,2,3)$, and $(2,4,1,3)$ are all equivalent, since they each change the set of edges to $\{[2,1],[5,3],[2,4],[5,6]\}$. To determine a canonical permutation, we choose the permutation with "increasing" indices, which in this case is $(1,3,2,4)$. Each sublist of indices in this permutation corresponding to the same source vertex is increasing. For example, the sublist $(1,2)$ corresponding to source vertex 2 is increasing. This sublist contains the indices of the a-type edges whose original target vertices become the new targets for a-type edges stemming from source vertex 2.

To enumerate all canonical permutations, first we enumerate all the possible increasing sublists corresponding to source vertex 2: $(1,2)$, $(1,3)$, $(1,4)$, $(2,3)$, $(2,4)$, and $(3,4)$. This leaves us with a list of incomplete permutations: $(1,\_,2,\_)$, $(1,\_,3,\_)$, $(1,\_,4,\_)$, $(2,\_,3,\_)$, $(2,\_,4,\_)$, and $(3,\_,4,\_)$. Now each sublist of blanks corresponds to source vertex 5. We fill in each sublist of blanks with all possible increasing sublists, noting for each permutation the indices that were already used for source vertex 2. In this case there is only one possible increasing sublist to fill in the blanks with for each permutation, so the final list of a-type permutations we need to check is $(1,3,2,4)$, $(1,2,3,4)$, $(1,2,4,3)$, $(2,1,3,4)$, $(2,1,4,3)$, and $(3,1,4,2)$.

Now in the general case, given a bond dictionary $\mathrm{dict}(G)$ for any realized graph G, we can assign to each edge of a given bond-edge type α_i an index from the set $[e_i] = \{1,\dots,e_i\}$. Let v be any source vertex for at least one α_i-type edge in G, and let $I_v \subseteq [e_i]$ be the set of indices corresponding to the edges with v as their source. In the algorithm, we construct canonical single bond-edge type permutations $p\colon [e_i] \to [e_i]$ piecewise using smaller maps $p_v\colon I_v \to [e_i]$ derived from each vertex. A permutation $p\colon [e_i] \to [e_i]$ is in canonical form if each $p_v\colon I_v \to [e_i]$ has the property that for any $m,n \in I_v$, if $m > n$, then $p_v(m) > p_v(n)$.

To determine the simpler maps, we impose any arbitrary order on the relevant vertices $v_1,\dots,v_k$, since the generation of each map will depend on the previous ones. Starting with v_1, we can choose any map $p_{v_1}\colon I_{v_1} \to [e_i]$ such that for any $m,n \in I_{v_1}$, if $m > n$, then $p_{v_1}(m) > p_{v_1}(n)$. The number of choices for p_{v_1} is given by

$$\binom{e_i}{|I_{v_1}|} = \binom{e_i}{\deg_{\alpha_i}(v_1)}.$$

Now to construct each subsequent map $p_{v_j}\colon I_{v_j} \to [e_i]$, since some elements of $[e_i]$ are already mapped to by p_{v_1} or other previous maps, we first construct a map $\widetilde{p}_{v_j}\colon I_{v_j} \to [e_i]\setminus \mathrm{im}(p_{v_1}) \setminus \cdots \setminus \mathrm{im}(p_{v_{j-1}})$, where $\setminus$ is the set difference and $\mathrm{im}(f)$ is the image of f. We again restrict the choice of $\widetilde{p}_{v_j}$ so that for any $m,n \in I_{v_j}$, if $m > n$, then $\widetilde{p}_{v_j}(m) > \widetilde{p}_{v_j}(n)$. Then each p_{v_j} is defined by $p_{v_j}(n) = \widetilde{p}_{v_j}(n)$. Thus the number of choices for p_{v_j} is given by

$$\binom{e_i - |I_{v_1}| - \cdots - |I_{v_{j-1}}|}{|I_{v_j}|} = \binom{e_i - \deg_{\alpha_i}(v_1) - \cdots - \deg_{\alpha_i}(v_{j-1})}{\deg_{\alpha_i}(v_j)}.$$

Once a choice for each map is made, the full permutation $p\colon [e_i] \to [e_i]$ is defined by

$$p(n) = \begin{cases} p_{v_1}(n) & \text{if } n \in I_{v_1}, \\ \vdots & \vdots \\ p_{v_k}(n) & \text{if } n \in I_{v_k}. \end{cases}$$

The number of choices for p is given by the product of the number of choices for each p_{v_j}. Repeating this process for each bond-edge type, the number of edge swaps this procedure generates is then

$$\prod_{\alpha \in \Sigma(P)} \binom{e_\alpha(G)}{\deg_\alpha(v_1)} \binom{e_\alpha(G) - \deg_\alpha(v_1)}{\deg_\alpha(v_2)} \cdots \binom{e_\alpha(G) - \deg_\alpha(v_1) - \cdots - \deg_\alpha(v_{k-1})}{\deg_\alpha(v_k)}.$$

This expression is equal to the upper bound shown in 3.13, which can be seen by expanding each binomial coefficient.

Method 3: Using Both Sides

After accounting for redundant permutations arising from edges that share a common source, an additional optimization can be applied. Similar to Method 2, we can also account for redundant permutations of edges that share a target rather than a source. In this case, the right component of each edge remains fixed while the left components are analyzed for potential swaps. We call the permutations of Method 2 *left permutations*, which leave the left components fixed and change which right component they are paired with, and we call the permutations of this new variant form *right permutations*, which leave the right components fixed and change which left component they are paired with.

As shown in Method 2, the left permutations alone, when applied to our starting graph as swaps, produce all possible non-isomorphic graphs. Similarly, the right permutations alone also cover all possible non-isomorphic graphs. Ultimately, left and right permutations encode the same information, that of which source half-edge to pair with which target half-edge. We can translate any given right permutation into a left permutation. After doing this, we can see that there is sometimes further redundancy.

Proposition 3.16 *Given a bond dictionary, for any bond-edge type α, let L be the set of canonical left single bond-edge type permutations as in Method 2, and let R be the similar set of canonical right single bond-edge type permutations. Let R' be the set of permutations of R re-encoded as canonical left permutations. Then $L \cap R'$ contains all necessary single bond-edge type permutations to construct all possible non-isomorphic graphs from the same pot and tile type multiplicity.*

Example 3.17 Consider the pot $P = (\{a^2\}, \{a, \hat{a}\}, \{\hat{a}^2\})$ with tile type multiplicity $r = (1, 2, 1)$. This realizes a graph with bond dictionary $\{a\colon [1,2], [1,3], [2,4], [3,4]\}$.

By Method 2, we find L and R. Consider the example right permutation $p = (2, 4, 1, 3)$. We can visualize this permutation using a 2×4 matrix:

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 1 & 3 \end{pmatrix}.$$

Each column says that p maps the top number to the bottom number. In this context, each column says that we take the right component of the edge whose index is the top number, and pair it with the left component of the edge whose index is the bottom number. When permuting the columns, these pairings remain fixed, so this matrix encodes the same map:

$$\begin{pmatrix} 3 & 1 & 4 & 2 \\ 1 & 2 & 3 & 4 \end{pmatrix}.$$

This alternate matrix was obtained by permuting the columns of the first matrix so that the bottom row is ascending. This process is equivalent to applying the permutation p^{-1} to the columns of the matrix, since it undoes the order of the bottom row given by p. Now the top row is given by $p^{-1} = (3, 1, 4, 2)$. This permutation, when used as a left permutation on the bond dictionary, is equivalent to p used as a right permutation, since each column of the matrix says that we take the left component of the edge whose index is the bottom number, and pair it with the right component of the edge whose index is the top number.

The permutation p^{-1} as a left permutation swaps the edges of the graph in the same way as p as a right permutation, but p^{-1} might not be in canonical form. So as a final step, we sort each sublist of p^{-1} with indices corresponding to edges stemming from the same vertex. For our example $p^{-1} = (3, 1, 4, 2)$, we see in the bond dictionary that edges 1 and 2 have the same left component. So we sort the sublist $(3, 1)$ in p^{-1}, giving us $(1, 3)$. Thus the canonical form of this left permutation is $(1, 3, 4, 2)$. After taking each right permutation, finding the inverse, and putting it into canonical form, we are left with these sets.

L	R	R'
$(1, 2, 3, 4)$	$(3, 4, 1, 2)$	$(3, 4, 1, 2)$
$(1, 2, 4, 3)$	$(4, 3, 1, 2)$	$(3, 4, 2, 1)$
$(1, 3, 2, 4)$	$(2, 4, 1, 3)$	$(1, 3, 4, 2)$
$(1, 3, 4, 2)$	$(4, 2, 1, 3)$	$(2, 3, 4, 1)$
$(1, 4, 2, 3)$	$(2, 3, 1, 4)$	$(1, 3, 2, 4)$
$(1, 4, 3, 2)$	$(3, 2, 1, 4)$	$(2, 3, 1, 4)$
$(2, 3, 1, 4)$	$(1, 4, 2, 3)$	$(1, 3, 4, 2)$
$(2, 3, 4, 1)$	$(4, 1, 2, 3)$	$(2, 3, 4, 1)$
$(2, 4, 1, 3)$	$(1, 3, 2, 4)$	$(1, 3, 2, 4)$
$(2, 4, 3, 1)$	$(3, 1, 2, 4)$	$(2, 3, 1, 4)$
$(3, 4, 1, 2)$	$(1, 2, 3, 4)$	$(1, 2, 3, 4)$
$(3, 4, 2, 1)$	$(2, 1, 3, 4)$	$(1, 2, 3, 4)$

From this we have

$$L \cap R' = \{(1, 2, 3, 4), (1, 3, 2, 4), (1, 3, 4, 2), (2, 3, 1, 4), (2, 3, 4, 1), (3, 4, 1, 2), (3, 4, 2, 1)\},$$

so for this pot and tile type multiplicity, those are the only edge permutations on the graph we need to check.

Proof. We proceed by contradiction. Given a bond dictionary, suppose there is some left permutation $p \notin L \cap R'$ which when applied to the edges of type α produces a graph not isomorphic to any graph produced by permutations in $L \cap R'$. Then either $p \notin L$ or $p \notin R'$. If $p \notin L$, then this contradicts that the permutations of L cover all necessary cases as shown in Method 2. If $p \notin R'$, then this contradicts that the permutations of R cover all necessary cases, since R and R' make the same swaps. □

As of the time of writing, the problem of counting the number of permutations in $L \cap R'$ remains open.

3.4 Graph Visualization

During the graph construction process, using an alternative representation of tiles and their connections, similar to the approach in [14], allows for visualization of the graph at each intermediate step. Given our tile list, for each tile we create a star graph with the central vertex labeled with the tile type and the outer vertices labeled with bond-edge types.

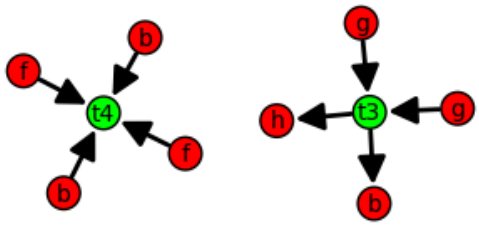


Figure 22: Disjoint union of tiles $t_4 = \{\hat{b}^2, \hat{f}^2\}$ and $t_3 = \{b, \hat{g}^2, h\}$. In the next step, a vertex labeled b on t_4 will be merged with the one on t_3.

When connections between a new tile and the current graph are identified, we take the disjoint union of the current graph and the new tile, bringing them into the same graph if they are not already. We then merge the vertices where the tiles bond. This approach was used in earlier implementations of the algorithm. In the current version, we instead encode each tile as in the vector pot and represent bonds as they are identified by putting them into a bond dictionary. Computationally, the old process runs very similar to the bond dictionary process, but is slightly slower due to the disjoint union and vertex merging functions, which are circumvented in the bond dictionary process.

Once a graph is represented as a Sage graph type, we can optimize the placements of the vertices by running Sage's spring algorithm with a very high number of iterations, and adjust manually by editing their positions if we wish. The Sage display of graphs is fairly limited, so we also created a function that takes in the graph and outputs a string of TikZ code for use in LaTeX diagrams. Using TikZ this string can be manually edited, but there are also a number of input options in the function.

3.5 Algorithm Example

In order to test the algorithm 3.1, we use various pots of known graphs to confirm that all non-isomorphic graphs were generated. As an example, given the pot

$$P = \{\{a, \hat{b}\}, \{\hat{a}^2, b\}, \{a, b\}, \{\hat{a}^2, \hat{b}\}\},$$

the algorithm finds that the smallest possible graph using this pot has order 6. Using each of the tile type multiplicities for a graph of order 6, the algorithm chooses an appropriate strategy for tile type ordering and connecting tiles, manages multi-edges and loops, and generates a graph as a bond dictionary. Using this initial graph, it then generates all possible graphs using canonical edge permutations and applies canonical labeling in order to identify the non-isomorphic graphs. Finally, it outputs a total of 5 non-isomorphic unique graphs that can be realized by the pot P, as shown in Figure 23. The code for the algorithm 3.1 can be found on GitHub [19].

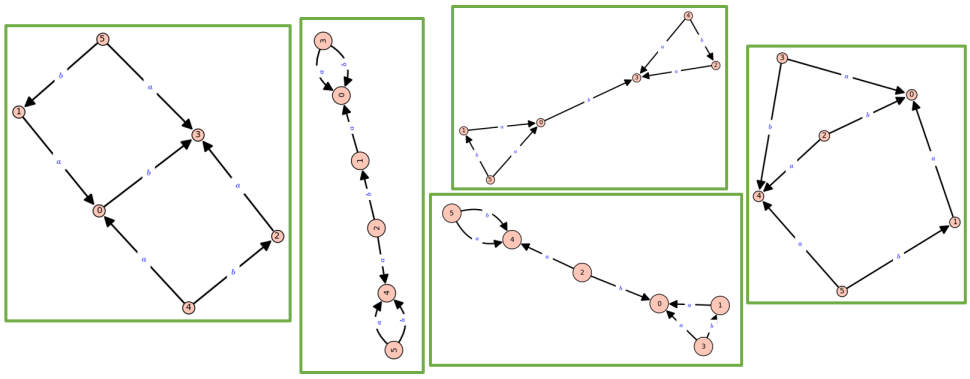


Figure 23: Five non-isomorphic graphs realized by pot $P = \{\{a, \hat{b}\}, \{\hat{a}^2, b\}, \{a, b\}, \{\hat{a}^2, \hat{b}\}\}$.

4 Conclusion and Discussion

The algorithm 3.1 successfully generates all non-isomorphic graphs that can be realized by a given pot of tiles. Given a pot of tiles, the algorithm 3.1 first analyzes the pot in order to find the smallest order (or user specified order) of a graph that can be realized, and its corresponding tile type multiplicities. Using a strategy for tile type ordering and connecting tiles, which may be specified by the user, the algorithm constructs a graph. Various edge permutations are then generated in order to identify all possible non-isomorphic graphs. Using visualization techniques, the algorithm successfully visually outputs all non-isomorphic graphs that can be realized by the inputted pot.

Significantly expanding on previous work which could only generate graphs using pots with one bond-edge type [14], the algorithm 3.1 can not only generate graphs using pots with two or more bond-edge types, but also all non-isomorphic graphs. Although, due to computer memory issues, the algorithm 3.1 is unable to handle certain complex

pots. For example, the pot $\{\{a^3, b\}, \{\hat{a}^2, b, \hat{b}\}, \{b, \hat{b}^3\}\}$ runs perfectly well, constructing 840 graphs and reducing them down to 53 unique graphs up to isomorphism in less than a minute. But the pot $\{\{a^2, b\}, \{\hat{a}, c^2\}, \{\hat{b}^2, \hat{d}\}, \{c, \hat{c}^2\}, \{a, c, d\}\}$ requires the construction of over 10^{20} edge swap permutations, so the algorithm crashes and is unable to reach the stage where it identifies any unique graphs up to isomorphism. This disparity is typical, with very few cases in the middle where the algorithm might terminate after an hour or so. Therefore, determining upper bounds on variables such as the number of bond-edge types and tile types that can be feasibly handled by the algorithm 3.1, optimizing the edge swap permutations to generate all possible valid graphs, and more efficiently solving the isomorphism problem remain problems worth exploring.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. DMS-1929284 while the authors were in residence at the Institute for Computational and Experimental Research in Mathematics in Providence, RI, during the Summer@ICERM program. This material is also based upon work supported by the National Science Foundation under Grant No. 2015553. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. We would like to thank the reviewers for their thoughtful feedback and suggestions, which have greatly improved the quality of our manuscript.

References

[1] J. Ellis-Monaghan, G. Pangborn, L. Beaudin, D. Miller, N. Bruno, A. Hashimoto, "Minimal tile and bond-edge types for self-assembling dna graphs." In: *Discrete and Topological Models in Molecular Biology*, Springer, 2014.

[2] G. Rinaldi, Nanoscience and technology: a collection of reviews from nature journals, *Assembly Automation*, **30, no. 2** (2010).

[3] H. Gu, J. Chao, S.-J. Xiao, N. C. Seeman, A proximity-based programmable dna nanoscale assembly line, *Nature*, **465, no. 7295** (2010), 202–205.

[4] P. Kumar, Directed self-assembly: Expectations and achievements, *Nanoscale Research Letters*, **5, no. 9** (2010), 1367–1376.

[5] W. e. a. Liu, Concepts and application of dna origami and dna self-assembly: A systematic review, *Applied Bionics and Biomechanics*, **2021** (2021).

[6] N. C. Seeman, H. F. Sleiman, Dna nanotechnology, *Nature Reviews Materials*, **3, no. 1** (2017), 1–23.

[7] L. Almodóvar, J. Ellis-Monaghan, A. Harsy, C. Johnson, J. Sorrells, Computational complexity and pragmatic solutions for flexible tile based dna self-assembly, available online at the URL: `https://arxiv.org/pdf/2108.00035`

[8] J. Ellis-Monaghan, N. Jonoska, G. Pangborn, Tile-based dna nanostructures: mathematical design and problem encoding, *Algebraic and Combinatorial Computational Biology*, (2019), 35–60.

[9] N. Jonoska, G. L. McColm, A. Staninska, "Spectrum of a pot for dna complexes," In: *DNA Computing: 12th International Meeting on DNA Computing, DNA12, Seoul, Korea, June 5-9, 2006, Revised Selected Papers 12*, Springer, 2006.

[10] E. Redmon, M. Mena, M. Vesta, A. R. Cortes, L. Gernes, S. Merheb, N. Soto, C. Stimpert, A. Harsy, Optimal tilings of bipartite graphs using self-assembling dna, *PUMP Journal*, **6** (2023), 124–150.

[11] G. Lopez, C. Johnson, Self-assembling dna complexes with a wheel graph structure, available online at the URL: `https://arxiv.org/pdf/2302.13014`

[12] C. Griffin, J. Sorrells, Tile-based modeling of dna self-assembly for two graph families with appended paths, *Involve*, **16, no. 1** (2023), 69–106.

[13] C. Johnson, A. Lavengood-Ryan, Analysis and algorithmic construction of self-assembled dna complexes, (2023).

[14] A. T. Lavengood-Ryan, Dna complexes of one bond-edge type, (2020).

[15] J. Ashworth, L. Grossmann, F. Navarro, L. Almodovar, A. Harsy, C. Johnson, J. Sorrells, Algorithmic pot generation: Algorithms for the flexible-tile model of dna self-assembly, (2024).

[16] D. B. West, Introduction to graph theory, (2001).

[17] B. D. McKay, A. Piperno, Practical graph isomorphism, ii, *Journal of Symbolic Computation*, **60** (2014), 94–112.

[18] R. A. Beezer, C. Godsil, Explorations in algebraic graph theory with sage, (2015).

[19] I. Horng, M. VonEschen, H. Luebsen, and G. Bielefeldt, DNASelfAssembly PotToGraphs, available online at the URL: `https://github.com/irishorng/DNASelfAssembly_PotToGraphs`

Grace Bielefeldt
St. Olaf College
1520 St Olaf Ave
Northfield, MN 55057
E-mail: `bielef1@stolaf.edu`

Iris Horng
University of Pennsylvania
3451 Walnut Street
Philadelphia, PA, 19104
E-mail: `ihorng@sas.upenn.edu`

Holly Luebsen
University of Texas at Austin
2515 Speedway
Austin, TX 78712
E-mail: `holly@luebsen.com`

Mitchell VonEschen
Lawrence University
711 East Boldt Way
Appleton, WI 54911
E-mail: `mjvville@gmail.com`

Leyda Almodóvar Velazquez
Stonehill College
320 Washington St.
North Easton, MA 02357
E-mail: lalmodovarvel@stonehill.edu

Amanda Harsy Ramsay
Lewis University
One University Parkway
Romeoville, IL 60446
E-mail: harsyram@lewisu.edu

Cory Johnson
California State University, San Bernardino
5500 University Pkwy
San Bernardino, CA 92407
E-mail: corrine.johnson@csusb.edu

Jessica Sorrells
Converse University
580 E Main St.
Spartanburg, SC 29302
E-mail: jessica.sorrells@converse.edu

Received: September 24, 2024 **Accepted**: May 2, 2025
Communicated by Laura Smith Chowdhury