12-1-2025

Fractals, Reachability, and Computation in Models of DNA Self-Assembly and Chemical Reaction Networks

Ryan Arlie Knobel
The University of Texas Rio Grande Valley

FRACTALS, REACHABILITY, AND COMPUTATION IN MODELS OF DNA SELF-ASSEMBLY AND CHEMICAL REACTION NETWORKS

A Thesis

by

RYAN A. KNOBEL

Submitted in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE IN ENGINEERING

Major Subject: Computer Science

The University of Texas Rio Grande Valley

December 2025

FRACTALS, REACHABILITY, AND COMPUTATION IN MODELS OF DNA SELF-ASSEMBLY AND CHEMICAL REACTION NETWORKS

A Thesis
by
RYAN A. KNOBEL

COMMITTEE MEMBERS

Dr. Tim Wylie
Chair of Committee

Dr. Robert Schweller
Committee Member

Dr. Bin Fu
Committee Member

Dr. Austin Luchsinger
Committee Member

December 2025

ABSTRACT

Knobel, Ryan A., Fractals, Reachability, and Computation in Models of DNA Self-Assembly and Chemical Reaction Networks. Master of Science in Engineering (MSE), December 2025, 159 pp., 6 tables, 54 figures, 98 references.

This thesis serves as the bridge between results compiled across varying models of tile self-assembly, molecular computation, and game complexity. As such, this thesis is broken into three chapters. In the first part, we show how to generate any Discrete Self-Similar Fractal (DSSF) with a feasible generator in the seeded Tile Assembly model, a model limited to single tile attachments and pairwise state transitions. In the next part, we study models of molecular computation, where we consider the problem of reachability in Chemical Reaction Networks and similar model extensions. The final part is a game-complexity analysis of Celtic! and *k*-ago, two games which have interesting connections to tile self-assembly, knot theory, and generalized *k*-in-a-row.

DEDICATION

To my father, Roger Knobel, my mother, Mayra Knobel, and my sister, Allyson Knobel (not by choice) for their continuous love and support; to my grandparents, for the kindness and patience shown to me over the years; to my abuela, gracias por estar siempre ahí para mí.

ACKNOWLEDGMENTS

This work would not have been possible without the help of my advisors Tim Wylie, Robert Schweller, Austin Luchsinger, and Bin Fu, whom served as the gateway into the research world. Furthermore, I would also like to thank Emmett Tomai for the help over the years with funding, from teaching assistant-ships during my undergraduate degree to research assistant-ships throughout my masters, including the G.A.A.N.N grant. This enabled me to put a primary focus on research, which I could not be more grateful for.

I would also like to thank those who facilitated my jump into research. To Tim, Michael, Andrew, Elise - thank you, all of you, for providing much needed guidance (and for your much needed patience). It is why I stuck to research, and something I will not forget.

I have also been fortunate to work with the following other co-authors: Divya Bajaj, Josh Brunner, Jose-Luis Castellanos, Michael Coulombe, Erik Demaine, Jenny Diomidova, Asher Haun, Markus Hecher, Jayson Lynch, Aiden Massie, Juan Manuel Perez, Tom Peters, Rene Reyes, Marco Rodriguez, Adrian Salinas, Pablo Santos, and Ramiro Santos.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

CHAPTER I

FRACTALS IN SEEDED TILE AUTOMATA

1.1 Chapter Overview

This chapter focuses on constructing Discrete Self-Similar Fractals (DSSFs) in the seeded Tile Automata model. We start by considering a subclass of DSSFs with restricted generators. We then extend these ideas to design a single system that can build any DSSF with a feasible generator at temperature 1.

1.2 Fractals with Restricted Generators

1.2.1 Overview

This work was in collaboration with the following authors: Adrian Salinas, Robert Schweller, and Tim Wylie. This project was mostly a solo project, with help in refining definitions, figures, proofs and overall design of the paper.

1.2.2 Introduction

The essence of many organisms and processes of nature can often be described as a collection of simpler, self-organizing components working together to form more complex structures. The study of such mechanisms has resulted in numerous advances in designing artificial programmable systems that accomplish similar tasks. In [96], Winfree introduced the abstract Tile Assembly Model (aTAM), in which single non-rotating ‘tiles’ attach to growing structures. Other extensions to this model include the 2-Handed Assembly Model (2HAM) [69], where two assemblies are allowed to attach; the Signal-passing Tile Assembly model (STAM) [68], where glues can turn ‘on’ and ‘off’ and assemblies can detach; and the seeded Tile Automata Model (seeded TA) [2], where single tiles attach to a base assembly (seed) and adjacent tiles are allowed to change

states. While mostly theoretical, experiments realized in the aTAM prove the potential of these programmable systems to build complex structures [77, 78, 96].

Despite varying nuances between models, building precise shapes remains a fundamental task. In particular, one of the most well-studied problems among these models is the self-assembly of self-similar fractals. In [49, 70], it was shown that the aTAM can not strictly (without error) build certain types of self-similar fractals. However, in other models, this does not hold true. In [23], it was shown that the 2HAM can finitely self-assemble a scaled-up Sierpinski carpet, while [48] showed that the 2HAM can finitely self-assemble a larger class of discrete self-similar fractals. In [68], it was shown that the Sierpinski triangle could strictly self-assemble in the STAM if tile detachments are allowed, with [47] providing constructions for any arbitrary discrete self-similar fractal with such detachments, while without such detachments, the finite number of times a STAM tile can change state makes some fractals impossible to build. The STAM is also capable of simulating Tile Automata [18] meaning these results can be ported to the STAM, however, the simulation uses detachments, which is a known result.

In this paper, we focus on building fractals in the seeded TA model without tile detachment, a model differing from the aTAM by the ability for adjacent tiles to transition states. Particularly, we show that a special class of discrete self-similar fractals can be super-strictly built (a more restricted version of strict), leaving a full treatment for future work. Super-strict assembly of a fractal essentially requires that each stage of the fractal be built in order on the way to building the infinite fractal. We feel this is a natural property to strive for as it implies that any intermediate stage of the assembly process would represent precisely the transition between two consecutive stages of the fractal, whereas without, an intermediate assembly could potentially contain a mishmash of many different incomplete fractal stages.

1.2.3 Preliminaries

This section defines the model, discrete self-similar fractals, and strictly building shapes as defined in [2, 70].

1.2.3.1 Seeded Tile Automata. Let Σ denote a set of *states* or symbols. A tile $t = (\sigma, p)$ is a non-rotatable unit square placed at point $p \in \mathbb{Z}^2$ and has a state of $\sigma \in \Sigma$. An *affinity function* Π over a set of states Σ takes an ordered pair of states $(\sigma_1, \sigma_2) \in \Sigma \times \Sigma$ and an orientation $d \in D$, where $D = \{\perp, \vdash\}$, and outputs an element of $\mathbb{Z}^{0+}$. The orientation d is the relative position to each other with $\perp$ meaning vertical and $\vdash$ meaning horizontal, with the σ_1 being the west or north state respectively. A *transition rule* consists of two ordered pairs of states $(\sigma_1, \sigma_2), (\sigma_3, \sigma_4)$ and an orientation $d \in D$, where $D = \{\perp, \vdash\}$. This denotes that if the states (σ_1, σ_2) are next to each other in orientation d (σ_1 as the west/north state) they may be replaced by the states (σ_3, σ_4). An *assembly* A is a set of tiles with states in Σ such that for every pair of tiles $t_1 = (\sigma_1, p_1), t_2 = (\sigma_2, p_2), p1 \neq p2$. Informally, each position contains at most one tile.

Let $B_G(A)$ be the bond graph formed by taking a node for each tile in A and adding an edge between neighboring tiles $t_1 = (\sigma_1, p_1)$ and $t_2 = (\sigma_2, p_2)$ with a weight equal to $\Pi(\sigma_1, \sigma_2)$. We say an assembly A is $\tau-$stable for some $\tau \in \mathbb{Z}^0$ if the minimum cut through $B_G(A)$ is greater than or equal to τ.

A *Seeded Tile Automata* system is a $6-$tuple $\Gamma = (\Sigma, \Lambda, \Pi, \Delta, s, \tau)$ where Σ is a set of states, $\Lambda \subseteq \Sigma$ a set of initial states, Π is an affinity function, Δ is a set of transition rules, s is a stable assembly called the seed assembly, and τ is the temperature (or threshold). A tile $t = (\sigma, p)$ may attach to an assembly A at temperature τ to build an assembly $A' = A \cup t$ if A' is $\tau-$stable and $\sigma \in \Lambda$. We denote this as $A \rightarrow_{\Lambda,\tau} A'$. An assembly A can transition to an assembly A' if there exist two neighboring tiles $t_1 = (\sigma_1, p_1), t_2 = (\sigma_2, p_2) \in A$ (where t_1 is the west or north tile) such that there exists a transition rule in Δ with the first pair being (σ_1, σ_2), the second pair being some pair of states (σ_3, σ_4) such that $A' = (A \setminus \{t_1, t_2\}) \cup \{t_3 = (\sigma_3, p_1), t_4 = (\sigma_4, p_2)\}$. We denote this as $A \rightarrow_{\Delta} A'$. For this paper, we focus on systems of temperature $\tau = 1$, and all bond strengths are equal to 0 or 1.

An assembly sequence $\overrightarrow{\alpha} = \{\alpha_0, \alpha_1, \ldots\}$ in Γ is a (finite or infinite) sequence of assemblies such that each $\alpha_i \rightarrow_{\Lambda,\tau} \alpha_{i+1}$ or $\alpha_i \rightarrow_{\Delta} \alpha_{i+1}$. An assembly sub-sequence $\beta = \{\alpha'_0, \alpha'_1, \ldots\}$ in Γ is a (finite or infinite) sequence of assemblies such that for each α'_i, α'_{i+1} there exists an assembly

sequence $\overrightarrow{\alpha} = \{\alpha'_i, \ldots, \alpha'_{i+1}\}$.

We define the *shape* of an assembly A, denoted $(A)_\Lambda$, as the set of points $(A)_\Lambda = \{p | (\sigma, p) \in A\}$.

1.2.3.2 Discrete Self-Similar Fractals. Let $1 < c, d \in \mathbb{N}$ and $X \subseteq \mathbb{N}^2$. We say that X is a $(c \times d)$-discrete self-similar fractal if there is a set $G \subseteq \{0, \ldots, c-1\} \times \{0, \ldots, d-1\}$ with $(0,0) \in G$, such that $X = \bigcup_{i=1}^{\infty} X_i$, where X_i is the i^{th} *stage* of G satisfying $X_0 = \{(0,0)\}$, $X_1 = G$, and $X_{i+1} = \{(a,b) + (c^i v, d^i u) | (a,b) \in X_i, (v,u) \in G\}$. In this case, we say that G *generates* X. We say that X is a discrete self-similar fractal if it is a $(c \times d)$-discrete self-similar fractal for some $c, d \in \mathbb{N}$. A generator G is termed *feasible* if it is a connected set, and there exist (not necessarily distinct) points $(0,y), (c-1,y), (x,0), (x,d-1) \in G$, i.e., a pair of points on each opposing edge of the generator bounding box that share the same row or column. Note that the fractal generated by a generator is connected if and only if the generator is feasible. For the remainder of this paper we only consider feasible generators.

1.2.3.3 Strict and Super-strict. Let X be a discrete self-similar fractal with feasible generator G. Consider a seeded TA system $\Gamma = (\Sigma, \Lambda, \Pi, \Delta, s, \tau)$ with $(s)_\Lambda = G$, and let S denote the set of all valid assembly sequences for Γ. Γ *strictly* builds X if $\forall \overrightarrow{\alpha_i} = \{s, \alpha_1, \ldots, \alpha_i, \ldots\} \in S$, $\overrightarrow{\alpha_i}$ is infinite and $\lim_{i \to \infty} (\alpha_i)_\Lambda = X$. We further say that Γ *super-strictly* builds discrete self-similar fractal X if $\forall \overrightarrow{\alpha_i} \in S$, there exists a subsequence $\beta = \{s, \alpha'_1, \ldots\}$ of $\overrightarrow{\alpha_i}$ such that each $(\alpha'_i)_\Lambda = X_i$.

1.2.3.4 Other Definitions. Let G be a feasible generator with corresponding points $(0,y)$, $(c-1,y), (x,0), (x,d-1) \in G$, X be the discrete self-similar fractal corresponding to G, and A be an assembly such that $(A)_\Lambda = X_i$ for some $i \in \{1, 2, \ldots\}$. We denote *key positions* as four points $p_N, p_E, p_W, p_S \in (A)_\Lambda$ satisfying $p_N = (x + c^{i-1} \cdot x, d^i - 1)$, $p_E = (c^i - 1, y + d^{i-1} \cdot y)$, $p_W = (0, y + y \cdot d^{i-1})$ and $p_S = (x + c^{i-1} \cdot x, 0)$. The four tiles $t_N, t_E, t_W, t_S \in A$ with positions p_N, p_E, p_W, p_S, respectively, are called *key tiles*. We denote $t_0 \in G$ the *origin tile* if t_0 has position $(0,0)$.

Let $G_G = (V, E)$ be the embedded graph formed by adding a vertex for each point $p \in G$ and adding an edge between vertices representing neighboring points $p_1, p_2 \in G$. Let $H = \langle h_0, \ldots, h_m \rangle$

$(m = |G| - 1)$ denote a Hamiltonian path in G_G, and let vertex h_0 represent the origin of the generator, where each h_j represents $p_j = (w_j, u_j)$. Given X_i, the i^{th} stage of generator G, denote $X_i^j = \{(a + c^i w_j, b + d^i u_j) \mid (a,b) \in X_i, j \in \{0,\ldots,m\}\}$, where j is the j^{th} *step* for stage i.

Additionally, we denote a particular assembly A as A_i if $(A)_\Lambda = X_i$ and A as A_i^J if $(A)_\Lambda = \bigcup_{k=0}^{J} X_i^k$, where $J \in \{0,\ldots,m\}$. To refer to a specific sub-assembly of A_i^J corresponding to step $j \in \{0,\ldots,J\}$ for stage i, we use A_i^j, where $A_i^j \subset A_i^J$ and $(A_i^j)_\Lambda = X_i^j$.

1.2.4 Construction

Given a feasible generator G where the resulting embedded graph G_G has a Hamiltonian path, we construct a seeded TA system with seed s ($(s)_\Lambda = G$) and origin tile t_0 that super-strictly builds the corresponding fractal infinitely. To start, the number of points m in the generator excluding the origin determines the number of steps m needs to scale the assembly from stage i to stage $i+1$. We denote each point in the generator as p_j, where j represents the distance from itself and the origin p_0 following a selected Hamiltonian path P in G_G. If we let $d(p_j, p_{j-1})$ denote the relative position of p_j to p_{j-1} (north, east, west or south), then we can represent the sequence of directions the assembly will grow.

The high-level idea of the construction is as follows: given an initial assembly A^0, translate a copy of A^0 in the direction of $d(p_1, p_0)$ and denote the copy as A^1. Repeat for all A^j, copying A^j in direction $d(p_{j+1}, p_j)$ until $j = m$. Set $A' = \bigcup_{j=0}^{m} A^j$, and then continue the process with $A^0 = A'$. See Figure 1.1 for an example.

However, since the seeded TA model is limited to single attachments and transitions, a direct implementation of this high-level idea is not possible. Instead, we give each tile the responsibility of placing itself in the correct location, with the final result being a copied translation from A^j to A^{j+1}. Thus, a crucial part of our construction is the ability to store information and send signals through the assembly. This section focuses on describing each of these components more thoroughly.

1.2.4.1 Storing Information. In order to correctly copy the base assembly, every tile needs specific information. This information can implicitly be stored as the state σ of the tile.

Current State ($STATE(t)$). As each tile is responsible for placing itself in the right location at the next step, it is important to know whether each tile has either 1) not placed itself yet, 2) is currently placing itself or 3) has already placed itself. $STATE(t)$ denotes the current state of tile t. In Figure 1.2, green tiles are tiles with $STATE(t) =$ complete, red tiles are tiles with $STATE(t) =$ incomplete and the yellow tile marked T has $STATE(t) =$ waiting. Gray tiles are tiles that have been placed from the current step, so they must wait until the current step finishes.

Direction to Key Tiles ($KEY_d(t)$). With the key tile information, signals are sent in the direction of the correct key tile. For instance, if the assembly is being copied to the north at step j, signals are sent in the direction of $t_N^j \in A_t^j$. $KEY_d(t)$ denotes this direction for t. To reference all 4 key tiles, we use $KEY_{NEWS}(t)$. This is illustrated in Figure 1.5a.

Next/Previous Tiles ($NEXT(t)/PREV(t)$). This serves the purpose of knowing where each tile's neighbors are (or should be). $NEXT(t)$ denotes the direction to the 'next' tiles from t, which usually signifies which directions the signal can propagate, excluding the source direction. $PREV(t)$ denotes the direction to the previous tile from t, which usually signifies the direction a signal comes from. We use $NEXT_t(t)$ to denote the tile adjacent to t in direction $NEXT(t)$ (or similarly, the set of tiles adjacent to t for each direction in $NEXT(t)$). Similarly, $PREV_t(t)$ denotes the tile adjacent to t in direction $PREV(t)$. This is described in Figure 1.3.

The State of Neighboring Sub-assemblies ($SUB_d(t)$). This is crucial for several reasons. Firstly, this creates the order in which tiles are placed. Secondly, this makes it possible to keep track of which direction the signal is coming from, and once the tile is placed, where the signal needs to return to. $SUB_d(t)$ denotes the state of the sub-assembly (whether all tiles have been placed or not) stemming from the neighboring tile of t in direction d. To reference the state of all sub-assemblies adjacent to t, we use $SUB_{NEWS}(t)$. Additionally, we refer to a specific sub-assembly as $SUBASM_d(t)$, denoting the sub-assembly stemming from tile t in direction d. See Figure 1.5b for an example.

The Tile being Transferred ($TRANS(t)$). To distinguish between different signals, each tile keeps track of which tile the signal started from. $TRANS(t)$ is used to denote the tile that the signal is coming from.

Step. Each tile stores which step it is a part of. This allows an assembly A_i^J to know which tiles to use (tiles in A_i^j) to create sub-assembly A_i^{j+1}. Additionally, A_i^j will only send signals to $t^j_{d(p_{j+1},p_j)}$.

Terminal ($TERM(t)$). Tiles must know when they are at the end of a sub-assembly. Once a terminal tile is placed, the system knows part of the sub-assembly is complete. $TERM(t)$ is a boolean that denotes whether tile t is terminal or not. In Figure 1.3, these tiles are marked t.

Caps ($CAP_d(t)$). Copied assemblies require one extra piece of information. As the shape grows, there are points where signals can branch in multiple directions. To direct this, signals always go 'left' when there is a fork. If the sub-assembly in this direction is already constructed, a cap is placed to prevent signals from going in that direction, and it instead goes to the next path. If all paths have a cap, then it turns around and the cap is shifted to reflect that all paths are complete. Caps start from terminal tiles and gradually shift as the sub-assemblies are completed. $CAP_d(t)$ is a boolean that denotes whether tile t has a cap in direction d. Figure 1.2 shows an example of how caps are used and shifted.

1.2.4.2 Signal Passing. In addition to the stored information, it is important that tiles can communicate through signal passing. This is done via transition rules.

Tile Placement Ordering. The order in which tiles place themselves follows a 'left' first order. As the tiles place themselves and are marked complete, transition rules prompt the next tile to start placing itself.

Tile Placement Signals. When a tile t_i^j is placing itself, the signal is transmitted from t_i^j to the tile adjacent to the target position for t_i^j. Transition rules make this possible by transferring the signal between adjacent tiles. In Figure 1.2, the transmission of this placement signal is represented as the sequence of yellow tiles.

Tile Placement Completion Signals. Once the tile is placed in the correct location, a 'completion' signal gets sent back the same direction as the placement signal. Once this signal reaches the tile getting placed from A_i^j, the tile is marked as complete.

Cap Signals. When a terminal tile is placed, as the 'completion' signal gets sent back in the sub-assembly being created, a 'cap' is sent back with it to mark the sub-assembly as complete. This forces future signals to go a different path to complete a different sub-assembly.

Reset Signals. When a stage is completed, reset signals are sent to update current state, direction to key tiles, state of neighboring tiles and step, as well as removing any remaining caps. Figure 1.4 illustrates the resetting process.

Figure 1.2 details the construction outlined in the subsequent sections. The following section provides more specific details to express how the system interacts to create these fractals.

1.2.4.3 Approach. This section describes the process for taking an assembly A_i to A_{i+1}, assuming G is a feasible generator for $(A_i)_\Lambda$, $H = \langle h_0, \ldots, h_m \rangle$ is a Hamiltonian path in G_G starting from the origin where m is the number of steps, i is the stage and $t_0 \in A_i^0$ is the origin tile. Additionally, we denote t_d^j as the key tile for direction d in assembly A_i^j and $d_j = d(p_{j+1}, p_j)$. We briefly define some additional terminology:

$OPP(d)$**.** This denotes the complement of direction d, e.g., $OPP(north) = south$.

$LEFT(D)$**.** Consider $D \in \{\{N\},\{E\},\{W\},\ \{S\},\{N,E\},\{E,S\},\{N,W\},\{W,S\}\}$, where N,E,W,S represent north, east, west and south respectively. $LEFT(D)$ denotes the 'left' direction for D. This is 1) North if $D = \{N,E\}$ or $\{N\}$, 2) East if $D = \{E,S\}$ or $\{E\}$, 3) West if $D = \{N,W\}$ or $\{W\}$ and 4) South if $D = \{W,S\}$ or $\{S\}$.

Conversely, $RIGHT(D)$ denotes $NEXT(D) \setminus LEFT(D)$. For a tile t, we use $LEFT_t(D)/RIGHT_t(D)$ to denote the tile adjacent to t in direction $LEFT(D)/RIGHT(D)$, respectively.

$RESET(t)$**.** At the end of stage i, tiles must reset. This includes 1) updating the direction to the new 4 key tiles and 2) updating the 'next' direction for former key tiles (see Figure 1.6). $RESET(t)$ denotes tile t resetting, defined as follows:

- If $t = t^j_{d_j}$, where $p_j \in X_1$ is the key position for d_j, set $KEY_{d_j}(t) =$ current tile.
- For each tile $t_a \in NEXT_t(t)$ (if all t_a have reset) set each $KEY_d(t) = PREV(t)$ if $KEY_d(t_a) = d(t,t_a)$. If there is a tile such that $KEY_d(t_a) \neq d(t,t_a)$, set $KEY_d(t) = KEY_d(t_a)$. If t_a is a key tile for d, set $KEY_d(t) = d(t_a,t)$.
- Clear $TERM(t_c)$ and update $NEXT(t_c)$ if appropriate.

1.2.4.4 Algorithm. Now we describe the algorithm. For better comprehension, we describe the algorithm using sub-processes. Technical descriptions of these sub-processes are included in Section 1.2.7.

Start with $j = 0$ and let $t_c = t_0$ denote the current tile getting placed, starting with the origin tile. Let A_i^{j+1} represent the translated assembly being created at step $j+1$. The following will be repeated until $j = m$. While $SUB_{PREV(t^j_{d_j})}(t^j_{d_j})$ and $SUB_{NEXT(t^j_{d_j})}(t^j_{d_j})$ are not marked as completed:

1. Let t_a denote the tile adjacent to t_c in direction $KEY_{d_j}(t_c)$. Run $send\_placement\_signal(t_c, t^j_{OPP(d_j)}, t^j_{d_j}, t_a, d_j)$ to send a signal through the assembly to place t_c in the correct location.
2. The placement signal stops at the tile adjacent to the target position by always traversing 'left' until a tile no longer exists. Run $place\_tile(t_a, t'_c, p)$ to place the tile at this location, where t_a is the tile adjacent to position p, the target position for t_c. This places t_c in the correct location as t'_c.
3. Retrace the signal to the tile that got placed by running $send\_completion\_signal(t'_c, t_a, d_j)$. This also marks the tile as complete.
4. Mark sub-assemblies as complete if needed. Run $mark\_completed\_sub\text{-}assemblies(t_c, t_a)$ if $TERM(t_c) =$ True, where t_c is the tile that just placed itself and t_a is the tile adjacent to t_c such that $STATE(t_a) =$ complete.

5. Choose the next tile to be placed. Let t_c denote the last tile updated and C be the tiles in $NEXT_{t_c}(t_c) \cup PREV_{t_c}(t_c)$ that have a completed state. If $t_c \neq t^j_{d_j}$, repeat from (1) with the new $t_c = LEFT_{t_c}(PREV(t_c) \cup NEXT(t_c) \setminus C)$.

6. If $t_c = t^j_{d_j}$, $j \neq m$ (the stage is not yet completed) and $SUB_{PREV(t^j_{d_j})}(t^j_{d_j})$ and $SUB_{NEXT(t^j_{d_j})}(t^j_{d_j})$ are marked as completed (every tile has now been placed), run $start\_next\_step(t^{j+1}_{OPP(d_j)}, t^{j+1}_{d_{j+1}}, d_j, d_{j+1})$ to signal for the next sub-assembly to start being created. Repeat from (1) with $j = j+1$, $A^j_i = A^{j+1}_i$ and clear $TRANS(t_c)$.

7. If instead $j = m$, the initial assembly has now been up-scaled and has reached the end of stage i. To repeat this process, the assembly now has to reset. Run $reset(t^m_{OPP(d_{m-1})})$.

8. Repeat the algorithm.

A primary reason as to why this algorithm works is the existence of a Hamiltonian path in the generator, as this dictates the directions in which the fractal grows. This allows growth of the fractal for any step to only depend on the created sub-assembly from the previous step, regardless of whether or not the resulting assembly contains a Hamiltonian path or not. If a generator does not contain a Hamiltonian path, then some sub-assemblies of the fractal must be used multiple times to create copies in multiple directions, which results in synchronicity issues as multiple signals could exist in the assembly at once.

1.2.5 Results

We now show that any feasible generator G with a Hamiltonian path in G_G can be super-strictly built by a seeded TA system Γ. Let G be a feasible generator for discrete self-similar fractal X, with $H = \langle h_0, \ldots, h_m \rangle$ denoting a Hamiltonian path in G_G such that each h_j corresponds to point $p_j = (w_j, u_j) \in G$. Let $d_j = d(p_{j+1}, p_j)$, A be the current assembly starting from $A = A^J_i$ for some $J \in \{0, \ldots, m-1\}$, and $A^{j+1'}_i = A \setminus A^J_i$. We denote the copy of a tile t as t'.

Lemma 1.2.1. *Under the construction from Section 1.2.4, tile $t^j_{OPP(d_j)} \in A^j_i$ must be the first tile to place itself.*

Proof. This is due to geometry. While there may be other adjacent tiles between A^j_i and the sub-assembly being created, $A^{j+1'}_i$, $t^j_{d_j}$ is the only tile that recognizes the existence of $A^{j+1'}_i$. Thus, the only way to send a signal to $A^{j+1'}_i$ is through $t^j_{d_j}$, and the only adjacent tile to $t^j_{d_j}$ in $A^{j+1'}_i$ is $t^{j'}_{OPP(d_j)}$. □

Lemma 1.2.2. *Let $t_i = (\sigma_i, p_i) \in A^j_i$ and let $t_f = (\sigma_f, p_f) \in A^{j+1'}_i$ represent the tile adjacent to p'_i, the target location for t'_i. A signal will follow exactly 1 path from t_i to t_f.*

Proof. Signals will always follow one path in A^j_i and $A^{j+1'}_i$. We show that this signal ends at tile t_f. This can be done by comparing the order in which tiles are chosen to be placed to the direction that the signal travels.

To prove equivalence, we show that for each tile in $A^{j+1'}_i$, $PREV(t_c) \cup NEXT(t_c) \setminus d(t_p, t_c) = NEXT(t^*_c)$. We consider 2 cases:

Case 1: $t^*_c = t^{j+1}_{OPP(d_j)}$. We have that $NEXT(t^*_c) = NEXT(t^j_{OPP(d_j)}) \cup PREV(t^j_{OPP(d_j)}) \setminus OPP(d_j)$. Initially, t_p is the tile in direction $OPP(d_j)$ from t_c. This results in $PREV(t_c) \cup NEXT(t_c) \setminus d(t_p, t_c)$.

Case 2: t^*_c is any other tile. Let t^*_p denote the tile adjacent to t^*_c from which the signal came from, with t_p, t_c denoting the corresponding tiles from A^j_i. We consider 2 scenarios.

1. $t_c \in NEXT_{t_p}(t_p)$. We have that $NEXT(t^*_c) = PREV(t_c) \cup NEXT(t_c) \setminus d(t_p, t_c)$.
2. $t_c \in PREV_{t_p}(t_p)$. We have that $NEXT(t^*_c) = NEXT(t_c) \cup PREV(t_c) \setminus (d(t_p, t_c) \cup \{\textit{directions to tiles not in step } j\})$. The only time there exists a direction to a tile not in step j is when t_c is the first tile placed in step j. Since this is no longer the case for step $j+1$, we get rid of this direction from $NEXT(t^*_c)$, and since the next tile chosen to be placed from t_c does not consider this direction either, the equivalence holds.

□

Lemma 1.2.3. *Let $t_i = (\sigma_i, p_i) \in A_i^j$ and let $t_f = (\sigma_f, p_f) \in A_i^{j+1'}$ represent the tile adjacent to p_i', the target location for t_i'. Tile t_f will place tile t_i' at position $p_i' = p_i + a \cdot d_j^*$, where $a - 1 \in \mathbb{N}$ represents the distance $|p_{d_j} - p_{OPP(d_j)}|$ and $d_j^* \in \{0,1\}^2$ is a 2-D vector denoting the direction.*

Proof. Let $d_j^* = [0,1], [1,0], [-1,0], [0,-1]$ represent $d_j =$ north, east, west and south, respectively. In the case of $t_{OPP(d_j)}$' being the tile placed, the signal will stop at tile $t_{d_j}^j$ with position $p_{d_j} = p_{OPP(d_j)} + (a-1) \cdot d_j^*$. Tile $t'_{OPP(d_j)}$ is then placed at position $p'_{OPP(d_j)} = p_{d_j} + d_j^* = p_{OPP(d_j)} + a \cdot d_j^*$.

For any other tile t_i, as described in Lemma 1.2.2, we know 2 things: 1), the signal from tile t_i will stop at tile t_f adjacent to position p_i' and 2) the order in which tiles are chosen to be placed is equivalent to the direction in which signals are passed. Since the relative position of p_i' to $t'_{OPP(d_j)}$ is the same as p_i to $t_{OPP(d_j)}$ and $t'_{OPP(d_j)} = t_{OPP(d_j)} + a \cdot d_j$, it follows that $p_i' = p_i + a \cdot d_j^*$. □

Lemma 1.2.4. *Let $t_i = (\sigma_i, p_i) \in A_i^j$ and let $t_i' = (\sigma_i', p_i') \in A_i^{j+1'}$ represent the copy of t_i. A signal will follow exactly 1 path from t_i' to t_i.*

Proof. By Lemma 1.2.2, there exists one path from p_i to the tile adjacent to p_i'. Thus, when tile t_i' is placed at position p_i', the converse holds true by retracing this path. □

Theorem 1.2.5. *There is at most one tile transmitting a signal in A.*

Proof. By contradiction. Assume that there exists 2 tiles $t_1 \neq t_2$ transmitting signals through A_i^j and let $A^{j_1^*}, A^{j_2^*} \subset A_i^j$ denote 2 sets of tiles such that:

1. $\forall t_a \in A^{j_1^*}$, $STATE(t_a) =$ complete.
2. $\forall t_b \in NEXT_{t_a}(t_a) \cup PREV_{t_a}(t_a)$, $t_b \in A^{j_1^*}$, $STATE(t_b) =$ incomplete or $t_b = t_1$.

where the same applies for $A^{j_2^*}$ and t_2. We consider 2 cases:

Case 1: $A^{j_1^*} \cap A^{j_2^*} = \emptyset$. This implies $t_{OPP(d_j)}^j \in A^{j_1^*}$ or $t_{OPP(d_j)}^j \in A^{j_2^*}$, but not both. By Lemma 1.2.1, $t_{OPP(d_j)}^j$ must be the first tile placed, resulting in a contradiction.

Case 2: $A^{j_1^*} \cap A^{j_2^*} \neq \emptyset$. Consider a tile t^* such that $t_1 \in SUBASM_{d_1}(t^*)$ and $t_2 \in SUBASM_{d_2}(t^*)$. Since $SUBASM_{RIGHT(\{d_1,d_2\})}(t^*)$ must wait for $SUBASM_{LEFT(\{d_1,d_2\})}(t^*)$ to be completed, it must be that $d_1 = d_2$. This implies that $t_1 = t_2$. □

Theorem 1.2.6. *Step $j+2$ will start only when step $j+1$ is completed.*

Proof. By our construction, since $t_{d_j}^j$ is the tile communicating between A_i^j and $A_i^{j+1'}$, both $SUB_{PREV(t_{d_j})}(t_{d_j})$ and $SUB_{NEXT(t_{d_j})}(t_{d_j})$ must be marked as completed before step $j+2$ begins. This is true only when $\forall t \in A_i^j, STATE(t) = \text{complete}$. □

Lemma 1.2.7. *Let $A_i^M = \bigcup_{j=0}^{m} A_i^j$ denote the resulting assembly at step m for stage i. Every tile will reset before moving to stage $i+1$.*

Proof. This is due to our construction. A tile t will only reset when $\forall t_a \in NEXT_t(t), t_a$ is reset. The only time this is not true is when t is terminal, which marks the end of a sub-assembly. □

Lemma 1.2.8. *Let $A_i^M = \bigcup_{j=0}^{m} A_i^j$ denote the resulting assembly at step m for stage i. When t_0 resets, there will exist at most 4 key tiles and $KEY_{NEWS}(t) \forall t \in A_i^M$ is updated to point to these new key tiles.*

Proof. We first show that there will exist at most four key tiles, one for each direction d. We know that t_d must appear only in some step j. Thus, as the assembly resets, t_d^j resets as the new t_d for the up-scaled assembly, and all other $t_d^k \; \forall k \neq j \in \{0,\ldots,m\}$ are reset to normal tiles. This leaves at most four key tiles.

Next, we show that every tile will point to the direction of the new t_d's. We show this by contradiction. Assume that there exists a tile t such that $KEY_d(t) = PREV(t)$, but $t_d \in SUBASM_{NEXT(t)}(t)$. This implies that for $t_a = PREV_{t_d}(t_d)$, $KEY_d(t_a) = PREV(t_a)$, which is true only if t_d is not the key tile for direction d. Hence, $t_d \notin SUBASM_{NEXT(t)}(t)$. □

Lemma 1.2.9. *Let $A^{j*} \subseteq A_i^j$ where $\forall t \in A^{j*}$, $STATE(t) = complete$. At the end of step $j+1$, $(A_i^{j+1'})_\Lambda = (A_i^j)_\Lambda + a \cdot d_j^*$, where $a-1 \in \mathbb{N}$ represents the distance $|p_{d_j} - p_{OPP(d_j)}|$ and $d_j^* \in \{-1,0,1\}^2$ is a 2-D vector denoting the direction.*

Proof. We use Lemmas 1.2.1 and 1.2.3 to construct an inductive proof. Let $d_j^* = [0,1],[1,0],[-1,0],[0,-1]$ represent $d_j =$ north, east, west and south, respectively.

Base case. $|A_i^{j*}| = 0$, with $t_1 = t^j_{OPP(d_j)}$ being the first tile copying itself (Lemma 1.2.1). By Lemma 1.2.3, $t'_{OPP(d_j)}$ is placed at position $p'_1 = p_1 + a \cdot d_j^*$.

Inductive step. $|A_i^{j*}| = k$, with t_{k+1} being the tile copying itself. By Lemma 1.2.3, t'_{k+1} is placed at position $p'_{k+1} = p_{k+1} + a \cdot d_j^*$. It holds that $(A_i^{j+1'})_\Lambda = (A^{j*})_\Lambda + a \cdot d_j^*$. Thus, $(A_i^{j+1})_\Lambda = (A_i^j)_\Lambda + a \cdot d_j^*$. □

Theorem 1.2.10. *At the end of stage i,* $(A_i^M)_\Lambda = (A_{i+1})_\Lambda$.

Proof. Follows from Lemma 1.2.9. For each A_i^K with $K \in \{0, \ldots, m-1\}$, a new sub-assembly A_i^{k+1} is constructed such that the new assembly $A_i^{K+1} = A_i^K \cup A_i^{k+1}$ satisfying $(A_i^{K+1})_\Lambda = X_i^{K+1}$. Thus, the final assembly $A_i^M = A_i^{M-1} \cup A_i^m$ with $(A_i^M)_\Lambda = \bigcup_{j=0}^{m} X_i^j = X_{i+1} = (A_{i+1})_\Lambda$. □

Theorem 1.2.11. *Let X be a discrete self-similar fractal with feasible generator G in bounding box $c \times d$ such that G_G has a Hamiltonian path $\langle h_0, \ldots, h_m \rangle$ where h_0 represents the origin. There exists a seeded TA system Γ with $O(|G|)$ states, $O(|G|^2)$ transitions and $O(|G|^2)$ affinities that super-strictly builds X.*

Proof. We start by showing Γ strictly builds X. This follows from Theorem 1.2.10. We start with seed s, where $(s)_\Lambda = G$ and each $t_j \in s$ stores $NEXT(t_j) = d(p_{j+1}, p_j)$ (if p_{j+1} exists), $PREV(t_j) = d(p_{j-1}, p_j)$ (if p_{j-1} exists) and $TERM(t_m) =$ True. Denote the assembly as A_1. By Theorem 1.2.10, applying the construction from Section 1.2.4 yields a new assembly $A_2 = \bigcup_{j=0}^{m} A_i^j$ with shape $(A_2)_\Lambda = X_2$. Repeating this for all A_i yields $\lim_{i \to \infty} (A_i)_\Lambda = X$.

Now we show that Γ super-strictly builds X. To do so, we consider 2 cases:

1. The assembly A_i^M at the end of stage i before resetting. Leading up to this point, the order in which tiles are placed and signals are passed is deterministic. As a result, there exists 1 unique valid assembly sequence from A_i to A_i^M.

2. The assembly A_{i+1} after A_i^M resets. While there no longer exists 1 unique valid assembly sequence from A_i^M to A_{i+1}, Lemmas 1.2.7 and 1.2.8 show that every tile will reset to point to the 4 new key tiles. From (1), the rest of the local information at each tile will remain the

same. Thus, every valid assembly sequence from A_i^M to A_{i+1} starts with A_i^M and ends with A_{i+1}.

Choose $\beta = \{s, A_2^M, A_3^M, \ldots\}$ or $\beta = \{s, A_2, A_3, \ldots\}$. It follows that Γ super-strictly builds X.

Disregarding steps, the total number of ways information can be locally stored at any tile is $O(1)$ since a tile has at most 4 neighbors. However, as tiles need to distinguish between different sub-assemblies representing different steps, this results in $O(|G|)$ different states. Similarly, since transition rules and affinities use combinations of 2 states, this results in $O(|G|^2)$ transition rules and affinity rules. □

1.2.6 Conclusion

In this paper, we present a method to strictly build fractals infinitely under the assumption that the generator is feasible and contains a Hamiltonian path. This contrasts with previously known results from similar (but slightly differing) models such as the aTAM, where some fractals, such as the Sierpinski triangle, are shown to be impossible to build strictly. Additionally, we show that this class of fractals can be super-strictly built, as our construction guarantees stopping at unique intermediate assemblies for all possible assembly sequences, where each intermediate assembly represents a different stage of the fractal. However, there remains several open questions:

- Our construction strictly builds fractals infinitely with states linear in the size of the generator and transitions and affinities quadratic in the size of the generator. Is there an alternative method that reduces the state, transition and affinity counts?
- Is it possible to construct all fractals infinitely? If not, what fractals are impossible to build?
- Does there exist a seeded TA system that can strictly build any fractal infinitely?
- Our work focuses on systems with temperature 1. Is it possible to take advantage of systems with higher temperatures to strictly build these fractals more efficiently, or does higher temperatures increase the complexity of the problem?

1.2.7 Full Details for Algorithm

Below are the full details for the sub-processes used in the algorithm described in Section 1.2.4.

send_placement_signal(t_c, $t^j_{OPP(d_j)}$, $t^j_{d_j}$, t_a, d_j):

1. Set $STATE(t_c) =$ waiting.
   - If $t_c = t^j_{OPP(d_j)}$, set $NEXT(t'_c) = NEXT(t_c) \cup PREV(t_c) \setminus OPP(d_j)$, $PREV(t'_c) = OPP(d_j)$ and $TERM(t'_c) =$ False.
   - Else if $STATE(t_a) =$ complete and the number of tiles from step j in $NEXT_{t_c}(t_c) \cup PREV_{t_c}(t_c)$ is $= 1$, set $TERM(t'_c) =$ True and $PREV(t'_c) = d(t_a, t_c)$.
   - Else if $STATE(t_a) =$ complete and $t_c \in PREV_{t_a}(t_a)$, set $NEXT(t'_c) = NEXT(t_c) \cup PREV(t_c) \setminus (d(t_a,t_c) \cup \{$*directions to tiles not in step j*$\})$ and $PREV(t'_c) = d(t_a, t_c)$.
   - Else, set $NEXT(t'_c) = NEXT(t_c)$ and $PREV(t'_c) = PREV(t_c)$.
   - Set $KEY_{NEWS}(t'_c) = KEY_{NEWS}(t_c)$, $TERM(t'_c) = TERM(t_c)$ if $TERM(t'_c)$ is not defined yet, $SUB_{d(t_c,t_a)}(t_a) =$ waiting and $TRANS(t_a) = t'_c$.
   - Let $t_c = t_a$.
2. While $t_c \neq t^j_{d_j}$:
   - Set $SUB_{d(t_c,t_a)}(t_a) =$ waiting and $TRANS(t_a) = TRANS(t_c)$.
   - Let $t_c = t_a$.
3. If no tile exists adjacent to t_c in direction d_j, stop. Otherwise, set $SUB_{OPP(d_j)}(t_a) =$ waiting and $TRANS(t_a) = TRANS(t_c)$.
4. Repeat the following:
   (a) Let $t_a = LEFT_{t_c}(NEXT(t_c))$ if $!CAP_{LEFT(NEXT(t_c))}(t_c)$, else set $t_a = RIGHT_{t_c}(NEXT(t_c))$.

(b) If t_a exists, set $SUB_{d(t_c,t_a)}(t_a) =$ waiting and $TRANS(t_a) = TRANS(t_c)$. Set $t_c = t_a$ and repeat from (a).

(c) If t_a does not exist, stop.

place_tile(t_c, t_c', p):

1. Place t'_c in position p and set $SUB_{d(t_c,t'_c)}(t'_c) =$ maybe. If $TERM(t'_c)$, set $SUB_{d(t_c,t'_c)}(t'_c) =$ maybe with cap.

send_completion_signal(t_c, t_a, d_j):

1. Set $SUB_{d(t_a,t_c)}(t_c)$ to its original state, clearing $TRANS(t_c)$ and changing $SUB_d(t_a) =$ waiting to $SUB_d(t_a) =$ maybe for the direction d that the signal came from.
2. If $length(NEXT(t_c)) =$ number of caps on t_c, set $SUB_d(t_a) =$ maybe with cap and clear the cap from t_c. Otherwise, set $SUB_d(t_a) =$ maybe and leave the cap on t_c in direction $LEFT(t_c)$.
3. If $STATE(t_a) =$ waiting, set $SUB_{d(t_a,t_c)}(t_c)$ to its original state, clearing $TRANS(t_c)$ and changing $STATE(t_a) =$ waiting to $STATE(t_a) =$ complete. Otherwise, set $t_c = t_a$, let t_a be the tile adjacent to t_c from which the signal came from and repeat from (1).

mark_completed_sub-assemblies(t_c, t_a):

1. Repeat the following until t_a is not updated:

   (a) Set $SUB_{d(t_c,t_a)}(t_a) =$ complete.

   (b) If $length(SUB_{NEWS}(t_c) =$ complete$) = length(NEXT(t_c) \cup PREV(t_c))$, set $t_c = t_a$ and let t_a be the tile next to t_c with $STATE(t_a) =$ complete and $SUB_{d(t_c,t_a)} =$ incomplete.

start_next_step($t^{j+1}_{OPP(d_j)}$, $t^{j+1}_{d_{j+1}}$, d_j, d_{j+1}):

1. Let $t_c = t^{j+1}_{OPP(d_j)}$. Set $TRANS(t_c) =$ ready. While $t_c \neq t^{j+1}_{d_{j+1}}$:

   (a) Let t_a denotes the tile adjacent to t_c in direction $KEY_{d_{j+1}}(t_c)$. Set $TRANS(t_a) = TRANS(t_c)$ and clear $TRANS(t_c)$. Then let $t_c = t_a$.

reset($t^m_{OPP(d_{m-1})}$):

1. Set $TRANS(t^m_{OPP(d_{m-1})}) =$ reset. For all tiles t_a adjacent to $t_c = t^m_{OPP(d_{m-1})}$ in directions $d \in NEXT(t_c) \cup PREV(t_c)$, set $TRANS(t_a) = TRANS(t_c)$ and set $t_c = t_a$.

2. If $TERM(t_c)$, set $t_c = RESET(t_c)$. For the tile $t_a \in PREV_{t_c}(t_c)$, set $SUB_{d(t_c,t_a)}(t_a) =$ done. Add $d(t_c,t_a)$ to $NEXT(t_a)$ if not already done.

3. If $length(SUB_{NEWS}(t_c) =$ done $) = length(NEXT(t_c))$, then for the tile $t_a = PREV_{t_c}(t_c)$, set $t_c = RESET(t_c)$ and set $SUB_{d(t_c,t_a)}(t_a) =$ done.

1.3 A Complete Characterization of Fractals

1.3.1 Overview

This work was in **collaboration** with the following authors: Asher Haun, Adrian Salinas, Ramiro Santos, Robert Schweller, and Tim Wylie. This project was mostly a **solo** project, with help in refining definitions, figures, proofs and overall design of the paper.

1.3.2 Introduction

The past 20 years within the field of algorithmic self-assembly has led to an extensive analysis among numerous models that reflect the behavior of microscopic organisms and processes in nature. Some of these models include the abstract Tile Assembly Model (aTAM) [96] in which non-rotatable tiles attach to existing structures, the 2-Handed Assembly Model (2HAM) [69], where 2 existing structures can attach at a time, and the Signal-passing Tile Assembly model (STAM) [68] that allows for both tile detachment and dynamic behavior between tiles as the system grows. Although these models are theoretical, they are motivated by natural mechanisms. Approaching solutions from the implementation side, a lot of work has been done experimentally in not only building complex structures [77, 78, 96], but also performing non-trivial computation [38, 98]. However, from both a theoretical and experimental viewpoint, some of the most intriguing questions are those that study the geometric limitations of each model such as whether it's possible to build some shape in a given model.

For many of these models, general shape building has been resolved. For instance, in the aTAM, any finite shape may be constructed if finite scaling is allowed [84]. For specific types of shapes though, such as different classes of fractals, most models still have open questions. For infinite discrete self-similar fractals (DSSFs), some shapes are strictly buildable (without error) [10], while others exhibiting specific properties such as "pinch point" fractals are not [9, 39]. Recent work in SODA 2025 [10] has provided a complete, polynomial-time decidable categorization of which DSSFs are buildable within the aTAM.

While many DSSFs can be weakly assembled (with error) at temperature 1 in the aTAM, there does not exist a DSSF that can be strictly assembled at temperature 1 [71]. In contrast, the 2HAM and STAM models yield more positive results. For instance, the work of [48, 23] showed that in comparison to the aTAM, the 2HAM (and k-HAM) can build a larger class of discrete self-similar fractals. The work of [47] showed that any arbitrary DSSF can be strictly assembled in the STAM with detachments. However, without detachments, the limited number of times a STAM tile can change states makes some fractals impossible to build at any temperature.

Thus, there exists a middle ground between the growth only, no detachment aTAM, where a limited number of fractals are strictly buildable, in comparison to the state-changing and detachment allowing STAM, where every arbitrary DSSF is strictly buildable. Our work focuses on bridging this gap through a study of building DSSFs in the seeded Tile Automata Model (seeded TA), a model that differs from the aTAM by the ability for adjacent tiles with certain properties to change states similar to cellular automata, but does not allow the powerful operation of the detachment of placed tiles. TA can also be simulated by the STAM as shown in [19].

Most related to our work is that of [53], in which it was shown that for a special class of DSSFs with "ham-path generators", there exists a seeded TA system Γ that can strictly build the DSSF. Our work focuses on extending these results to any general DSSF in two ways. We first show that for every arbitrary fractal X, there exists a seeded TA system that strictly builds X infinitely with a finite number of states, transitions, and affinities. We then show that under these constructions, there exists a *single* seeded TA system Γ that can strictly build any DSSF infinitely at temperature

1, thus providing a full characterization for fractal generation in seeded TA. An emulator of this seeded TA system can be found at `https://github.com/rknobel1/fractals`.

1.3.3 Preliminaries

This section defines the model, discrete self-similar fractals, and strictly building shapes similar to the previous definitions in [2, 70].

1.3.3.1 Seeded Tile Automata. Let Σ denote a set of *states* or symbols. A tile $t = (\sigma, p)$ is a non-rotatable unit square placed at point $p \in \mathbb{Z}^2$ and has a state of $\sigma \in \Sigma$. An *affinity function* Π over a set of states Σ takes an ordered pair of states $(\sigma_1, \sigma_2) \in \Sigma \times \Sigma$ and an orientation $d \in D$, where $D = \{\perp, \vdash\}$, and outputs an element of $\mathbb{Z}^{0+}$. The orientation d is the relative position to each other with $\perp$ meaning vertical and $\vdash$ meaning horizontal, with the σ_1 being the west or north state respectively. A *transition rule* consists of two ordered pairs of states $(\sigma_1, \sigma_2), (\sigma_3, \sigma_4)$ and an orientation $d \in D$, where $D = \{\perp, \vdash\}$. This denotes that if the states (σ_1, σ_2) are next to each other in orientation d (σ_1 as the west/north state) they may be replaced by the states (σ_3, σ_4). An *assembly* A is a set of tiles with states in Σ such that for every pair of tiles $t_1 = (\sigma_1, p_1), t_2 = (\sigma_2, p_2), p1 \neq p2$. Informally, each position contains at most one tile.

Let $B_G(A)$ be the bond graph formed by taking a node for each tile in A and adding an edge between neighboring tiles $t_1 = (\sigma_1, p_1)$ and $t_2 = (\sigma_2, p_2)$ with a weight equal to $\Pi(\sigma_1, \sigma_2)$. We say an assembly A is $\tau-$stable for some $\tau \in \mathbb{Z}^0$ if the minimum cut through $B_G(A)$ is greater than or equal to τ.

A *Seeded Tile Automata* system is a $6-$tuple $\Gamma = (\Sigma, \Lambda, \Pi, \Delta, s, \tau)$ where Σ is a set of states, $\Lambda \subseteq \Sigma$ a set of initial states, Π is an affinity function, Δ is a set of transition rules, s is a stable assembly called the seed assembly, and τ is the temperature (or threshold). A tile $t = (\sigma, p)$ may attach to an assembly A at temperature τ to build an assembly $A' = A \cup t$ if A' is $\tau-$stable and $\sigma \in \Lambda$. We denote this as $A \rightarrow_{\Lambda,\tau} A'$. An assembly A can transition to an assembly A' if there exist two neighboring tiles $t_1 = (\sigma_1, p_1), t_2 = (\sigma_2, p_2) \in A$ (where t_1 is the west or north tile) such that there exists a transition rule in Δ with the first pair being (σ_1, σ_2), the second pair being some pair

of states (σ_3, σ_4) such that $A' = (A \setminus \{t_1, t_2\}) \cup \{t_3 = (\sigma_3, p_1), t_4 = (\sigma_4, p_2)\}$. We denote this as $A \rightarrow_\Delta A'$. For this paper, we focus on systems of temperature $\tau = 1$, and all bond strengths are equal to 0 or 1.

An assembly sequence $\overrightarrow{\alpha} = \{\alpha_0, \alpha_1, \ldots\}$ in Γ is a (finite or infinite) sequence of assemblies such that each $\alpha_i \rightarrow_{\Lambda,\tau} \alpha_{i+1}$ or $\alpha_i \rightarrow_\Delta \alpha_{i+1}$. An assembly sub-sequence $\beta = \{\alpha'_0, \alpha'_1, \ldots\}$ in Γ is a (finite or infinite) sequence of assemblies such that for each α'_i, α'_{i+1} there exists an assembly sequence $\overrightarrow{\alpha} = \{\alpha'_i, \ldots, \alpha'_{i+1}\}$.

We define the *shape* of an assembly A, denoted $(A)_\Lambda$, as the set of points $(A)_\Lambda = \{p | (\sigma, p) \in A\}$.

1.3.3.2 Discrete Self-Similar Fractals. Let $1 < c, d \in \mathbb{N}$ and $X \subseteq \mathbb{N}^2$. We say that X is a $(c \times d)$-discrete self-similar fractal if there is a set $G \subseteq \{0, \ldots, c-1\} \times \{0, \ldots, d-1\}$ with $(0,0) \in G$, such that $X = \bigcup_{i=1}^{\infty} G_i$, where G_i is the i^{th} *stage* of G satisfying $G_0 = \{(0,0)\}$, $G_1 = G$, and $G_{i+1} = \{(a,b) + (c^i v, d^i u) | (a,b) \in G_i, (v,u) \in G\}$. In this case, we say that G *generates* X (see Figure 1.7 for an example). We say that X is a discrete self-similar fractal if it is a $(c \times d)$-discrete self-similar fractal for some $c, d \in \mathbb{N}$. A generator G is termed *feasible* if it is a connected set, and there exist (not necessarily distinct) points $(0,y), (c-1,y), (x,0), (x,d-1) \in G$, i.e., a pair of points on each opposing edge of the generator bounding box that share the same row or column. Note that the fractal generated by a generator is connected if and only if the generator is feasible. For the remainder of this paper we only consider feasible generators. An example of a non-feasible generator is shown in Figure 1.8.

1.3.3.3 Strict Self-Assembly. Let X be a discrete self-similar fractal with a feasible generator G. Consider a seeded TA system $\Gamma = (\Sigma, \Lambda, \Pi, \Delta, s, \tau)$ with $(s)_\Lambda = G$, and let S denote the set of all valid assembly sequences for Γ. Γ *strictly* builds X if $\forall \overrightarrow{\alpha} = \{s, \alpha_1, \ldots, \alpha_i, \ldots\} \in S$, $\overrightarrow{\alpha}$ is infinite and $\lim_{i \rightarrow \infty} (\alpha_i)_\Lambda = X$.

1.3.3.4 Other Notation. Let A be an assembly. We denote A as A_i if $(A)_\Lambda = G_i$. We refer to a particular sub-assembly of A as $A_j^{(w,z)}$, where $(A_j^{(0,0)})_\Lambda = G_j$, if $(A_j^{(w,z)})_\Lambda = \{(a,b) + (c^j w, d^j z) \mid$

$(a,b) \in (A_j^{(0,0)})_\Lambda\}$ for $0 \leq w < c$ and $0 \leq z < d$. For convenience, we also refer to a sub-assembly with shape $(A_j^{(w,z)})_\Lambda$ as $(A_j^{(a,b)})_\Lambda + (l \cdot c^j, m \cdot d^j)$ for $l = w - a$ and $m = z - b$, or in other words, a translation of $A_j^{(a,b)}$ $l \cdot c^j$ units horizontally and $m \cdot d^j$ units vertically.

Let G be a feasible generator with points $(0,y),(c-1,y),(x,0),(x,d-1) \in G$ and X be the discrete self-similar fractal corresponding to G. We denote *key positions* for some A_i as four points $p_N, p_E, p_W, p_S \in (A_i)_\Lambda$ satisfying $p_N = (x + c^{i-1} \cdot x, d^i - 1)$, $p_E = (c^i - 1, y + d^{i-1} \cdot y)$, $p_W = (0, y + y \cdot d^{i-1})$ and $p_S = (x + c^{i-1} \cdot x, 0)$. The four tiles $t_N, t_E, t_W, t_S \in A_i$ with positions p_N, p_E, p_W, p_S, respectively, are called *key tiles*. We denote $t_0 \in G$ as the *origin tile* if t_0 has position $(0,0)$.[1]

1.3.4 High Level Overview

This section provides a high level overview of the techniques used to prove the main results of this paper, which are stated below.

Theorem 1.3.1. *For every feasible generator G, there exists a seeded TA system Γ which strictly builds the corresponding fractal with $O(1)$ states, transitions, and affinities.*

Theorem 1.3.2. *There exists a single seeded TA system Γ that, given as input any feasible generator G as a seed, will strictly build the corresponding fractal for G.*

These techniques utilize the following observation. which states that for a generator G that builds fractal X, taking any G_i as a generator still builds fractal X.

Observation 1.3.3. *Let G, G' be generators for fractals X, X' such that $G' = G_a$ for $a \in \mathbb{N}$, then X and X' are the same fractal, that is, $\lim_{i\to\infty} G_i = \lim_{j\to\infty} G'_j$.*

In particular, by starting with G, up-scaling the fractal to G_2, then setting $G = G_2$ and repeating infinitely, the fractal jumps $2i$ stages with respect to the original generator G after each iteration as shown in Figure 1.9. Thus, the procedures used throughout this paper consider the process of taking an assembly with shape G_i and up-scaling it to an assembly with shape G_{2i}. The following section provides a high-level overview of these procedures.

[1]While not all generators contain the origin (e.g., the 5-point cross), we consider generators that do have it for simplicity and note that our constructions work for any arbitrarily chosen tile as the origin tile.

1.3.4.1 The General Idea. The basic idea for scaling the generator is shown in Figure 1.10. From Observation 1.3.3, to build a desired fractal X, we start with a generator $G = G_1$, turn the fractal into G_2, then take $G = G_2$ as the new generator for X, and repeat this process infinitely. We adopt this approach as it makes the presentation more straightforward. Another important observation is that during the "up-scaling" process, in which a fractal G_i is taken to G_{2i}, each region with shape G_i in G_{2i} essentially corresponds to a *pixel* in G (in Figure 1.10b, the darker region corresponds to the circled tile from Figure 1.10a). Thus, one way of building G_{2i} is to let each region with shape G_i create copies in all cardinally adjacent directions as long as the corresponding points exist in the generator.

We incorporate these ideas in the seeded TA system through the use of *pseudo-seeds*, which are found within each region with shape G_i (see Figure 1.10b). These pseudo-seeds are used to tell each region which point in G it corresponds to, allowing it to create copies in the correct directions. Once G_i has been scaled to become G_{2i}, then the resulting fractal is treated as the new generator, and all pseudo-seeds are removed.

However, due to the non-determinism of seeded TA, this approach will lend itself to competition as shown in Figure 1.11a. To counter this, we consider the embedded graph formed by taking each point in the generator as a vertex, and edges between vertices if the corresponding points are cardinally adjacent, and derive a spanning tree (ST) of the graph. This ST dictates the order in which these regions are constructed, thus guaranteeing each region of the fractal can only be created by another specific region as shown in Figure 1.11b.

1.3.4.2 State Abstraction. A crucial part of the constructions used in this paper is the ability for tiles to explicitly encode information as the state of the tile. Thus, due to scale and ease of comprehension, this paper approaches fractal construction by considering each tile as an object (*class*) with accessible *fields*, where transition rules between tiles update the information encoded at each tile. By keeping the amount of stored information constant, it is then a matter of considering all possible valid states, transitions, and affinity rules to construct the system that strictly builds the fractal for a given generator.

1.3.4.3 Synchronization. The seeded TA model is non-deterministic in nature. However, in order to modularize the procedures needed to take an assembly from shape G_i to G_{2i}, it is important that each "module" occurs *sequentially* without interfering with other processes. This is accomplished with counters, which ensure that signals are fully propagated through an assembly, and that specific tile states are attained only at the completion of a procedure, which ensures that one procedure is completed prior to starting the next one.

1.3.4.4 The Up-scaling Procedure. The procedure for taking an assembly with shape $G_1 = G$ to G_2 is as follows. Let G be the generator, t_0 be the origin tile, and G_M a computed ST for the embedded graph as described.

1. For each cardinally adjacent point p to the origin, if an edge exists between p and the origin in G_M, create a translated copy of G in this direction.
2. Repeat (1) with the newly created regions as the new 'origins'.
3. Once no more regions can be created, consider a new ST for G_2 and set $G = G_2$.

However, since the seeded TA model is limited to single tile attachments/transitions, creating a translated copy of G does not occur simultaneously, but instead tile by tile. Thus, the procedure for creating a translated copy of G can be described as follows. Let t_p be the pseudo-seed/origin tile for an assembly with shape G.

1. Choose an adjacent tile to t_p which shares an edge in the computed ST. Let d_c denote the direction to this tile.
2. Propagate d_c to all tiles in the assembly.
3. Locate the key tile for direction $OPP(d_c)$. Denote this tile $t_{OPP(d_c)}$.
4. Create a copy of $t_{OPP(d_c)}$ and transfer it to the key tile for direction d_c.
5. Place $t_{OPP(d_c)}$ in the corresponding location and mark it (needed for resetting procedure).

6. Retrace back to the copied tile and mark as completed.

7. Choose a new tile adjacent to $t_{OPP(d_c)}$ and repeat from (4).

Once all tiles are marked as completed, then the assembly resets and chooses another direction to copy in. This copying procedure is then also applied to the translated copies and repeats until no more translated copies can be created.

1.3.4.5 The Resetting Procedure. With the assembly going from A_i to A_{2i}, the goal of the system is to now reset A_{2i} to make it the new generator. This is done in 2 steps. The first step is to mark each translated copy with a "reset ready" state, similar to what is done in (2) from Section 1.3.4.4.

Once all tiles in A_{2i} are "reset ready", then the resetting procedure commences, including resetting all tiles to their default states and deriving a new ST for A_{2i}, as described below.

1. Starting from each terminal tile t (leaf nodes in the ST), if t should be a key tile for A_{2i}, make it a key tile.

2. If t is marked, update the ST to include an edge between t and the corresponding neighboring tile.

3. Repeat from (1) with all adjacent tiles to each t that have not been reset and share an edge in the ST.

As the origin tile is reset, the new assembly A_{2i} is taken as the new generator and the up-scaling procedure commences.

1.3.4.6 The Complete Algorithm. The complete algorithm can then be seen as a combination of the procedures described above. Given a generator G, run the up-scaling procedure to reach G_2. Then, run the resetting procedure to turn G_2 into the new generator, and repeat with $G = G_2$.

A formal description of this algorithm is shown in Algorithm 1 and is based on the sub procedures detailed throughout Sections 1.3.6, 1.3.7, and 1.3.8, which include Algorithms 5 through

14. Let G be a generator for DSSF X, with S denoting the seed assembly generated from Section 1.3.6.

1.3.5 Construction Preliminaries

We start by detailing the notation used in Sections 1.3.7 and 1.3.8. With the ability for tiles to transition between multiple states, it is possible to encode information as the state of the tile and update this state as the assembly grows. Thus, for ease of understanding, we consider tiles as a *class*, with each tile t an *object* of this class with accessible *fields*. We denote accessing a specific field as *t.field_name*. We also make use of *functions* in standard form *function_name(...parameters)*.

1.3.5.1 Stored Tile Information. The following are accessible fields for a given tile t, i.e., these are the attributes of the object. Let $d \in \{N, E, W, S\}$.

- ***state.*** Takes a value of *None*, *P*, *W*, *F* with a default of *None*. These values denote whether a tile is not ready to produce (None), has not placed itself yet (Producing), is currently placing itself (Waiting), or has placed itself (Finished).
- ***copy_direction.*** Takes a value of *N*, *E*, *W*, *S*, *None* or *r* with a default of *None*. These values denote if t should be copied in the sub-assembly to the north (N), east (E), west (W), south (S), or not (None), or if the sub-assembly is being reset (r).
- ***origin_direction.*** Takes a value of *N*, *E*, *W*, *S*, *. These values denote the direction to the original seed. A value of * denotes that t is the original seed.
- ***key_d.*** Takes a value of *N*, *E*, *W*, *S*, *. These values denote the direction to key tile t_d. A value of * denotes that t is the key tile for direction d.
- ***new_key_tile.*** Takes a value of *True* or *False* with a default of *False*. These values denote if t will become a key tile for the growing assembly (True) or not (False).
- ***neighbors.*** Takes a value of the power-set $P(\{N, E, W, S\}) \setminus \emptyset$. These values denote the direction to some (but not always *all*) adjacent tiles to t. Informally, we say $t_a \in t.neighbors$ if t_a is in direction d from t, where $d \in t.neighbors$.

- ***new_neighbors.*** Takes a value of the power-set $P(\{N, E, W, S\})$ with a default of *None*. These values denote the direction to any new neighbors of t as the assembly grows.
- ***state_d.*** Takes a value of *N*, *W*, *M*, *Y*, *None*. These values denote whether the sub-assembly stemming from direction d is not complete (N), placing a tile (W), placed a tile and might now be complete (M), complete (Y), or does not exist (None).
- ***transfer.*** Points to a created tile t_c or takes a value of ?, *r*, or *None* with a default of *None*. These values denote whether the tile is passing a signal or not, or if the sub-assembly is ready to globally reset (r).
- ***terminal.*** Takes a value of *True* or *False*. These values denote if a tile is terminal (True) or not (False).
- ***caps.*** Takes a value of the power-set $P(\{N, E, W, S\})$ with a default of *None*. These values denote that the sub-assembly stemming from each direction in $t.caps$ is complete.
- ***pseudo_seed.*** Takes a value of *True* or *False* with a default of *False*. These values denote if the current tile is a pseudo-seed (True) or not (False).
- ***will_be_pseudo_seed.*** Takes a value of *True* or *False* with a default of *False*. These values denote if the current tile will be a pseudo-seed (True) or not (False) for the created assembly.
- ***original_seed.*** Takes a value of *True* or *False* with a default of *False*. These values denote if the current tile is the origin tile (True) or not (False).
- ***copied.*** Takes a value of *True* or *False* with a default of *False*. These values denote if t is a copied tile (True) or not (False).
- ***sub_assembly_copied.*** Takes a value of *True* or *False* with a default of *False*. These values denote if the sub-assembly corresponding to t is complete (True) or not (False).
- ***first_tile.*** Takes a value of *True* or *False* with a default of *False*. These values denote if t is the first tile copied for a sub-assembly (True) or not (False).

- ***can_place.*** Takes a value of *True* or *False* with a default of *False*. These values denote if t can copy itself (True) or not (False).

- ***num_times_copied.*** Takes a value $\in \{0,1,2,3,4\}$ with a default of *0*. These values denote the number of times the sub-assembly corresponding to t has been copied.

- ***counter.*** Takes a value $\in \{0,1,2,3,4\}$ or *None* with a default of *None*. This counter field is used primarily for synchronization.

- ***temp.*** Default of *None*. Used to store temporary information.

1.3.5.2 Functions. The following details certain functions that will be used.

- ***opp(d).*** Takes a direction d and outputs the 'opposite' direction $N \rightarrow S$, $E \rightarrow W$, $W \rightarrow E$, $S \rightarrow N$.

- ***retrieve_tile(t, d).*** Takes a tile t and direction d and outputs the neighboring tile in direction d.

- ***num_comp_sub-assemblies(t).*** Takes a tile t and outputs the number of $state\_d = Y$, or in other words, the number of completed sub-assemblies stemming from t.

- ***next(t).*** Takes a tile t and outputs the 'next' direction for a signal from tile t as shown in Algorithm 2.

- ***next_tile_placed(t).*** Takes a tile t with $t.state = F$ and outputs the next tile to get placed as shown in Algorithm 3.

- ***breadcrumb(t).*** Takes a tile t and outputs the direction of the 'breadcrumb' left by a signal as shown in Algorithm 4.

1.3.6 Seed Initialization

This section details how the initial seed for the seeded TA system is constructed given a feasible generator G with key positions p_N, p_E, p_W, p_S. The seed is a blank assembly A in which each point $p \in G$ contains a tile. Each tile begins with default values, and with the tile t_o located at the origin having $t_o.origin\_seed = True$. An example of an initial seed assembly is shown in Figure 1.12a.

1.3.6.1 Neighbor Initialization. The first step is to initialize the *origin_direction*, *neighbors*, *terminal*, and *state_d* fields of each tile, which is achieved by deriving a spanning tree (ST) for the seed assembly from the generator. Start by running a breadth-first search (BFS) from the original seed in which each tile is a vertex and edges exist between cardinally adjacent vertices, leaving a breadcrumb at each tile to denote the previous tile. With this information, we can now initialize the *neighbors* field for each tile t. First, for each adjacent tile t_a in direction d, if t_a is marked by t, append d to $t.neighbors$. Then, for the tile that marked t in direction d, append d to $t.neighbors$ and $t.origin\_direction$. The completion of these steps initialize the *origin_direction* and *neighbors* fields for each tile. It is then left to update the *terminal* and *state_d* fields. For each tile t and for each $d \in t.neighbors$, set $state\_d = N$. Furthermore, if $len(t.neighbors) = 1$, set $t.terminal = True$. An example of the result of this neighbor initialization is shown in Figure 1.12b.

1.3.6.2 Key Tile Initialization. The next step is to initialize each *key_d* field. This is achieved by running a BFS from the tile located at the key position for direction d on the ST derived from earlier. More formally, for each key tile t_d, start by running a BFS from t_d with a breadcrumb left at each tile denoting the direction d_p to the previous tile. Once all tiles have been marked, set each $t_d.key\_d = *$ to denote that t_d is the key tile for direction d. Then, for every other tile, set $t.key\_d = d_p$. An example of the result of the key tile initialization is shown in Figure 1.12c for the north direction.

1.3.7 Copying Procedure

This section focuses on how an assembly A_i is taken to assembly A_{2i}. Specifically, we consider the copying procedure for a sub-assembly $A_i^{(w_1,z_1)}$ that creates a new sub-assembly $A_i^{(w_2,z_2)}$. This procedure can be broken down into 4 steps: choosing a copying direction for $A_i^{(w_1,z_1)}$, copying each tile in $A_i^{(w_1,z_1)}$ to create $A_i^{(w_2,z_2)}$, readying the newly created $A_i^{(w_2,z_2)}$, and lastly resetting $A_i^{(w_1,z_1)}$. Each step heavily relies on synchronicity to ensure each step does not interfere with the others, which will be discussed within each sub-section.

1.3.7.1 Choosing a Copying Direction. Choosing a copying direction starts from the origin seed or the pseudo-seed. Let t_o denote this tile. The idea is that t_o looks at adjacent neighbors in each direction $d \in t.neighbors$. Let t_a be an adjacent tile to t_o in direction d. If $t_a.sub\_assembly\_copied = False$, then d becomes the chosen copying direction, as $A_i^{(w_1,z_1)}$ still needs to be copied in direction d. This choice is propagated to all tiles in $A_i^{(w_1,z_1)}$, using the *counter* field of each tile to ensure that all tiles receive this signal. Once this is done, then the next step begins. Algorithm 5 explicitly details how this process is undertaken, with an example of the process shown in Figure 1.13.

1.3.7.2 Copying Tiles. With $A_i^{(w_1,z_1)}$ having chosen a direction d_c to copy, the next step is to copy each of the tiles in $A_i^{(w_1,z_1)}$ to create $A_i^{(w_2,z_2)}$. Let t_{d_c} denote the key tile for $A_i^{(w_1,z_1)}$ in direction d_c. Note that all signals between $A_i^{(w_1,z_1)}$ and $A_i^{(w_2,z_2)}$ pass through t_{d_c}. Thus, the first tile that must be placed is the key tile for direction $OPP(d_c)$, which we denote as $t_{OPP(d_c)}$. We start by describing how this tile is located, then detail how and in what order each tile is copied.

Locating the First Tile to Place. The last tile to get updated is t_o. Thus, locating $t_{OPP(d_c)}$ is a matter of following the direction from each $t.key\_OPP(d_c)$ until $t_{OPP(d_c)}$ is reached. Algorithm 6 explicitly details how this process is undertaken.

Once $t_{OPP(d_c)}$ is found, $t_{OPP(d_c)}.first\_tile$ and $t_{OPP(d_c)}.can\_place$ are set to true. This distinction from other tiles is important for determining when $A_i^{(w_1,z_1)}$ has been completely copied and to start the sub-assembly copying procedure.

Copying a Tile. Copying a given tile t happens in two steps. In the first step, a copy of t is created. This copy is transferred to t_{d_c}, where it can then be transferred to $A_i^{(w_2,z_2)}$. Once the signal has reached $A_i^{(w_2,z_2)}$, the copy is transferred using the 'next' direction at each tile until a tile no longer exists where the signal needs to go. The copy is then placed as a physical tile in this location using an affinity rule. Algorithm 7 details what information is copied and stored in $t.transfer$. Algorithm 8 details how a signal is passed starting from $A_i^{(w_1,z_1)}$ and ending in $A_i^{(w_2,z_2)}$. A crucial part of this signal passing is the use of 'caps' to act as funnels in $A_i^{(w_2,z_2)}$. As parts of $A_i^{(w_2,z_2)}$ are finished, these caps can continue to be shifted until the entire sub-assembly is completely built.

In the second step, a signal is sent back to mark the tile being copied as complete, following the breadcrumb trail that is left behind. As stated, *caps* are created once terminal tiles get placed to prevent signals from entering already completed parts of the sub-assembly. Algorithm 9 explicitly details how the breadcrumb trail is retraced, as well as how caps are used and shifted. An example of how each tile is copied is shown in Figure 1.14.

Choosing the Next Tile to Place. Once the recently copied tile t has $t.state = F$, the following step is to select the next tile to get placed. The logic behind choosing the next tile varies depending on if t is a terminal tile or not. If $t.terminal = False$, then the tile selection is a matter of picking the adjacent tile t_a in direction $next\_tile\_placed(t)$. However, if $t.terminal = True$, then it is a matter of retracing steps until a tile is reached where some adjacent tiles have yet to be placed and selecting from these tiles. Algorithm 10 explicitly details how this process is undertaken, with examples of both cases shown in Figure 1.15.

1.3.7.3 Readying the Created Sub-assembly. As the last tile in $A_i^{(w_2,z_2)}$ is placed and the breadcrumb trail is retraced, $t_{OPP(d_c)}$ from $A_i^{(w_2,z_2)}$ will contain $len(t_{OPP(d_c)}.neighbors) - 1$ caps, signifying $A_i^{(w_1,z_1)}$ has been fully copied into $A_i^{(w_2,z_2)}$. It is then left to update the state of the pseudo-seed in $A_i^{(w_2,z_2)}$ so that this sub-assembly can repeat the process. Algorithm 11 explicitly details how this process is undertaken, with an example shown in Figure 1.16.

1.3.7.4 Resetting the Copied Sub-assembly. The final step is to reset the copied sub-assembly. As the last tile is marked as complete and each $t_{OPP(d_c)}.state\_d$ from $A_i^{(w_1,z_1)}$ has a

value of Y or *, this signifies the completion of the copying process. It is then left to first reset all tiles in $A_i^{(w_1,z_1)}$, then choose a new copying direction. Algorithm 12 explicitly details how this process is undertaken, with an example shown in Figure 1.17.

1.3.8 Resetting Procedure

This section details the resetting procedure once an assembly A_i has become A_{2i}. We first discuss local resetting that occurs within each sub-assembly, followed by the global reset that unifies the resulting assembly to restart the process.

1.3.8.1 Local Resetting. From Algorithm 12, each time a sub-assembly is copied, the *num_times_copied* field of the pseudo-seed/original seed is increased by one. Once this field is equal to the length of its *neighbors* field, or if the pseudo-seed/origin seed is terminal, then this sub-assembly is no longer needed to create other sub-assemblies. Thus, starting from the pseudo-seed/original seed, a signal is sent to label each tile with a 'ready to reset' state. This state is accompanied by the *counter* field of each tile being set to the length of the *neighbors* field to ensure that all tiles within each sub-assembly are marked with the resetting state. Let t_p denote the pseudo-seed seed/original seed for some sub-assembly. Algorithm 13 explicitly details how this procedure is undertaken, with an example shown in Figure 1.18.

1.3.8.2 Global Resetting. From the local resetting that occurs within each sub-assembly, tiles are marked terminal or not depending on the number of new neighbors each tile will have. As each tile is updated, global resetting occurs from the new terminal tiles. This resetting includes setting each tiles *neighbors* field to the *new_neighbors* field, as well as updating the following fields to their default values: *state*, *copy_direction*, *pseudo_seed*, *will_be_pseudo_seed*, *sub_assembly_copied*, *caps*, *copied*,
num_times_copied, *temp*, *transfer*, *new_neighbors*, *first_tile*, *can_place* and *counter*.

Additionally, the directions to the new key tiles are updated, as well as the location of these key tiles as well, which turns A_{2i} into the new generator. These directions are updated starting from the terminal tiles for each tile t based on the following logic:

1. If *t.new_key_tile* = *True*, set *t.key_d* = * for each direction which t is a key tile.

2. If *t.terminal* and *t.new_key_tile* = *False*, then each *t.key_d* field is set to *t.origin_direction*.

3. For each adjacent tile t_a not in the direction of the original seed, if t_a.*key_d* is not in the direction of t, set t.*key_d* to the direction of t_a.

4. Otherwise, t.*key_d* must be in the direction of the original seed.

Algorithm 14 explicitly details how this procedure is undertaken, with an example shown in Figure 1.19. Once the original seed is reset, then the newly created assembly A_{2i} becomes the new generator for the next step, and the entire procedure restarts. An important thing to note is each tile t is only reset if $t.transfer = r$ and if $t.counter = 0$, which ensures all tiles in the respective sub-assembly have been given this reset signal.

1.3.9 Correctness

Sections 1.3.6, 1.3.7, and 1.3.8 detail how a seeded TA system could be constructed to transform an assembly A_i into an assembly A_{2i} for a generator G. This section focuses not only on proving the correctness of the procedures used, but also on proving that these results imply the existence of a single-seeded TA system that can take any generator and infinitely build the respective fractal. Let G be a generator with key tiles t_d and original tile t_o, and let A_i be an assembly with shape X_i. Let $A_i^{(a,b)} \in A_i$ be a sub-assembly, with $A_i^{(x,y)}$ denoting a created sub-assembly from $A_i^{(a,b)}$ in direction d_c. We start by showing the procedures used in Sections 1.3.7 and 1.3.8 occur sequentially.

1.3.9.1 Correctness of Procedure Order. This section outlines the correctness of the procedure order.

Lemma 1.3.4. *Given $A_i^{(a,b)}$, which represents either the initial assembly $A_i^{(0,0)}$ or a sub-assembly after Algorithm 12 is applied. All tiles in $A_i^{(a,b)}$ are marked with a copying direction, as described in Algorithm 5, prior to initiating the copying procedure, as described in Algorithm 6.*

Proof. This is due to synchronization. Algorithm 6 takes, as input, the pseudo-seed t_p for $A_i^{(a,b)}$ and runs once $t_p.counter = len(t_p.neighbors)$. While this algorithm could technically run prior to the completion of Algorithm 5, the signal transmission from Algorithm 6 require each tile t to have a counter value equal to the length of $t.neighbors$. This is only achieved once each t propagates the copying direction to all adjacent neighbors. Thus, even if Algorithm 6 were to start early, any copying signals would still have to wait until all tiles have received a copying direction, or until the completion of Algorithm 5. □

Lemma 1.3.5. *Given $A_i^{(a,b)}$, which represents a sub-assembly prior to applying Algorithm 5, and $A_i^{(x,y)}$, which represents the created sub-assembly after applying Algorithms 5, 6, 7, 8, 9, and 10 to $A_i^{(a,b)}$. $A_i^{(x,y)}$ will start the copying procedure, as described in Algorithm 11, only once all of $A_i^{(a,b)}$ has been copied to $A_i^{(x,y)}$.*

Proof. Let t_p be the pseudo-seed for $A_i^{(x,y)}$. The copying procedure for a sub-assembly starts once $t_p.copy\_direction = '?'$, which is a result of Algorithm 11. However, Algorithm 11 is run when a special condition is met from Algorithm 8: that is, Algorithm 11 runs only when the key tile $t_{OPP(d_c)}$ for $A_i^{(x,y)}$ satisfies $len(t_{OPP(d_c)}.caps) = len(t_{OPP(d_c)}.neighbors) - 1$. By construction, this means each direction in $t_{OPP(d_c)}.neighbors$ (excluding the direction to $t_{d_c} \in A_i^{(a,b)}$) contains a cap, or in other words, all tiles from $A_i^{(a,b)}$ are copied into $A_i^{(x,y)}$. Thus, $t_p.copy\_direction = '?'$ will only occur once all of $A_i^{(a,b)}$ has been copied to $A_i^{(x,y)}$. □

Lemma 1.3.6. *Given $A_i^{(a,b)}$, which represents a sub-assembly prior to applying Algorithm 12. $A_i^{(a,b)}$ is reset only when all tiles have been copied, which occurs once Algorithm 10 can no longer be run.*

Proof. Let $t_1 \in A_i^{(a,b)}$ be the first tile placed from the copying procedure. Algorithm 12 requires, as input, $t_{OPP(d_c)} = t_1$, and runs only once $t_1.status = 'F'$ and each $t_1.state\_d = 'Y'$ or $t_1.state\_d = '*'$. This means each sub-assembly within $A_i^{(a,b)}$ stemming from t_1 in each direction d must be completely copied, otherwise $t_1.state\_d$ would take a value $\neq 'Y'$. □

Lemma 1.3.7. *Given $A_i^{(a,b)}$, which represents an assembly prior to applying Algorithm 13. Local resetting for $A_i^{(a,b)}$ occurs only when $A_i^{(a,b)}$ no longer has pending directions to copy, that is until Algorithm 5 can no longer run.*

Proof. Local resetting occurs only when the pseudo-seed t_p for a sub-assembly satisfies $t_p.num\_times\_copied = len(t_p.neighbors)$. From Algorithm 12, each time a sub-assembly has been fully copied, the *num_times_copied* field of the pseudo-seed t_p is increased by 1. Thus, $t_p.num\_times\_copied = len(t_p.neighbors)$ is satisfied only when $A_i^{(a,b)}$ no longer has pending directions to copy, in which case Algorithm 5 can no longer run. □

Lemma 1.3.8. *Given A_{2i}, which represents the assembly prior to applying Algorithm 14. Global resetting for A_{2i} occurs only when each sub-assembly $A_i^{(a,b)} \in A_{2i}$ has been locally reset, as described in Algorithm 13.*

Proof. From Algorithm 14, global resetting occurs from terminal tiles t' in which $t'.terminal = True$ and $t'.transfer = r$. Furthermore, this global reset is only transmitted to other neighboring tiles t if $t.counter = 0$ and $t.transfer = 'r'$, which implies that t has completely transmitted the reset signal to all neighboring tiles, or in other words, Algorithm 13 has run on sub-assembly $A_i^{(a,b)}$. □

Theorem 1.3.9. *Algorithms 5, 6, 7, 8, 9, 10, 11, 12, 13, and 14 from Sections 1.3.7 and 1.3.8 occur sequentially.*

Proof. These algorithms are split between 6 major procedures: choosing a copy direction (Algorithm 5), copying a sub-assembly (Algorithms 6, 7, 8, 9, 10), readying the created sub-assembly (Algorithm 11), resetting the copied sub-assembly (Algorithm 12), locally resetting a sub-assembly (Algorithm 13), and global resetting (Algorithm 14). The sequential order for each of these 6 major procedures follows from Lemmas 1.3.4, 1.3.5, 1.3.6, 1.3.7, and 1.3.8. □

1.3.9.2 Correctness of Mechanics. We now consider the correctness of the mechanics used within each procedure. We start with the following definitions, and then present the results. Intuitively, a *base-assembly* is an assembly which represents a valid output from the seed initialization

as described in Section 1.3.6. These assemblies have 4 key tiles with exactly 1 path between any 2 tiles in the assembly. Thus, by taking a base-assembly A_i to base-assembly A_{2i}, this ensures that A_{2i} can become the new 'seed' assembly for the system. A *base-sub-assembly* is essentially the same, except field values for tiles are not required to be defaulted. Examples of base-assemblies and base-sub-assemblies are shown in Figure 1.20.

Definition 1.3.1 (base-assembly)**.** *An assembly A is a base-assembly if the following hold:*

1. *There exists exactly* 4 *key tiles, one for each cardinal direction*
2. *For each direction d, in the embedded graph G_d, where each tile is a vertex and a directed edge exists from tile t to t' if $t' \in t.key\_d$, all fully traversed paths end at key tile t_d*
3. *In the embedded graph G_o, where each tile is a vertex and a directed edge exists from tile t to t' if $t' \in t.origin\_direction$, all fully traversed paths end at the original seed t_o*
4. *The embedded graph G, where each tile is a vertex and edges exist between tiles t and t' if $t \in t'.neighbors$ and $t' \in t.neighbors$, is a spanning tree*
5. *Excluding origin_direction, key_d, neighbors, state_d and terminal, all other fields for each tile are defaulted*

Definition 1.3.2 (base-sub-assembly)**.** *Let A be an assembly with $A_i \subset A$. A sub-assembly $A_i^{(x,y)}$ is a base-sub-assembly if:*

1. *There exists exactly* 4 *key tiles, one for each cardinal direction*
2. *For each direction d, in the embedded graph G_d, where each tile is a vertex and a directed edge exists from tile t to t' if $t' \in t.key\_d$, all fully traversed paths end at key tile t_d*
3. *The embedded graph G, where each tile is a vertex and edges exist between tiles t and t' if $t \in t'.neighbors$ and $t' \in t.neighbors$, is a spanning tree*
4. *There is exactly one pseudo-seed or original-seed in $A_i^{(x,y)}$*

Lemma 1.3.10. *Given $A_i^{(a,b)}$, which represents a sub-assembly prior to applying Algorithm 6, and $A_i^{(x,y)}$, which represents the created sub-assembly after applying Algorithms 5, 6, 7, 8, 9, and 10 to $A_i^{(a,b)}$. $A_i^{(a,b)}$ and $A_i^{(x,y)}$ are base-sub-assemblies.*

Proof. By construction. From Algorithm 7, each tile that is copied preserves the 4 key tiles, the directions to these key tiles, and the *t.neighbors* field. Since $A_i^{(a,b)}$ starts as a base-sub-assembly, it follows that $A_i^{(x,y)}$ is also a base-sub-assembly. Furthermore, the copying procedure does not alter any of these fields from $A_i^{(a,b)}$, so $A_i^{(a,b)}$ remains a base-sub-assembly. □

Lemma 1.3.11. *Let $t_c \in A_i^{(a,b)}$ be a tile getting copied with position (w,z), and $t_a \in t_c.neighbors$ be the tile directly adjacent to t_c with position $p = (w-j, z-k)$ and $t_c.status = F$. The copying signal is transmitted from t_c to the tile with position $p'_a = (w+j\cdot c^i - j, z+k\cdot d^i - k) \in (A_i^{(x,y)})_\Lambda \mid (j,k) \in \{(1,0),(0,1),(-1,0),(0,-1)\}$.*

Proof. From Lemma 1.3.10 and the definition of a base-sub-assembly, there exists exactly one path between any 2 pair of tiles in $A_i^{(a,b)}$. Thus, for a signal to be able to reach position $(w+j\cdot c^i, z+k\cdot d^i)$, it must first go through $t_{d_c} \in A_i^{(a,b)}$, then visit the tile at position $(w+j\cdot c^i - j, z+k\cdot d^i - k)$. We now show this is the case.

From Algorithm 8, the copying signal enters $A_i^{(x,y)}$ through t_{d_c}, then follows the *next* direction of each tile. Consider a tile $t \in A_i^{(a,b)}$ with $t.state = F$, and the copy of t, denoted $t' \in A_i^{(x,y)}$. If *num_comp_sub-assemblies(t) = len(t.neighbors)*, then each sub-assembly stemming from t has been completely copied, and thus the copy signal no longer needs to go through t'.

From Algorithm 9, these sub-assemblies in $A_i^{(x,y)}$ will have then generated caps that prevent signals from going through t'. On the other hand, if *num_comp_sub-assemblies(t)* $\neq$ *len(t.neighbors)*, then there exists at least 1 sub-assembly stemming from t that has not yet been completely copied. Thus, it is the case that $len(t'.caps) < len(t'.neighbors) - 1$, and so the copy signal will be guaranteed to go through t'. The only point at which the copy signal can no longer traverse is when a tile no longer exists in the direction it needs to travel, which is at position $p'_a = (w+j\cdot c^i - j, z+k\cdot d^i - k)$. □

Lemma 1.3.12. *Let $t \in A_i^{(a,b)}$ be a tile getting copied with position (w,z). Tile t is placed at position $p = (w + j \cdot c^i, z + k \cdot d^i) \in (A_i^{(x,y)})_\Lambda \mid (j,k) \in \{(1,0),(0,1),(-1,0),(0,-1)\}$.*

Proof. By induction. We start with the base case, or for $t = t_{OPP(d_c)}$. From Algorithm 8, the tile placement signal is transmitted to t_{d_c}. Let (w,z) denote the position of $t_{OPP(d_c)} \in A_i^{(a,b)}$. It must then be the case that t_{d_c} is located at position $p = (w + j \cdot c^i - j, z + k \cdot d^i - k)$. The new tile is then placed at position $p = (w + j \cdot c^i - j + j, z + k \cdot d^i - k + k) = (w + j \cdot c^i, z + k \cdot d^i)$.

Now the inductive step. Assume this holds for all other tiles $\in A_i^{(a,b)}$. Let $t' \in A_i^{(a,b)}$ be the most recently copied tile with position $p' = (w - j, z - k)$, with t denoting the next tile chosen to be copied with position (w,z). From Lemma 1.3.11, we get that the copying signal is sent to the tile with position $(w + j \cdot c^i - j, z + k \cdot d^i - k)$. Tile t is then placed in position $(w + j \cdot c^i - j + j, z + k \cdot d^i - k + k) = (w + j \cdot c^i, z + k \cdot d^i)$. □

Lemma 1.3.13. *Given $A_i^{(a,b)}$, which represents a sub-assembly prior to applying Algorithm 6, and $A_i^{(x,y)}$, which represents the created sub-assembly after applying Algorithms 5, 6, 7, 8, 9, and 10 to $A_i^{(a,b)}$. $A_i^{(a,b)}$ is the only sub-assembly used to create $A_i^{(x,y)}$.*

Proof. Follows from A_i being a base-assembly. Since there exists exactly one path from t_o to every other tile $t \in A_i^{(a,b)}$, each sub-assembly is created using exactly one other sub-assembly. □

Lemma 1.3.14. *Given $A_i^{(a,b)}$, which represents a sub-assembly prior to applying Algorithm 6, and $A_i^{(x,y)}$, which represents the created sub-assembly after applying Algorithms 5, 6, 7, 8, 9, and 10 to $A_i^{(a,b)}$. There is at most 1 tile being placed from $A_i^{(a,b)}$ in $A_i^{(x,y)}$ at a time.*

Proof. For a tile t to initiate the copying procedure, $t.can\_place$ must be set to *True*. Initially, all tiles in $A_i^{(a,b)}$ have this field set to *False*. From Algorithm 6, the first instance in which a tile has $t.can\_place = True$ is for $t_{OPP(d_c)}$. From Algorithm 10, it is only once the *status* field is changed to F for tiles with $t.can\_place = True$ that another tile is chosen to get placed. □

Theorem 1.3.15. *Given $A_i^{(a,b)}$, which represents a sub-assembly prior to applying Algorithm 5, and $A_i^{(x,y)}$, which represents the created sub-assembly after applying Algorithms 5, 6, 7, 8, 9, and 10 to $A_i^{(a,b)}$. $(A_i^{(x,y)})_\Lambda = (A_i^{(a,b)})_\Lambda + (j \cdot c^i, k \cdot d^i) \mid (j,k) \in \{(1,0),(0,1),(-1,0),(0,-1)\}$.*

Proof. Follows from Lemmas 1.3.12, 1.3.13 and 1.3.14. From Lemma 1.3.12, each tile in $A_i^{(a,b)}$ is placed in the correct location in $A_i^{(x,y)}$. Lemma 1.3.13 guarantees that there is no competition between $A_i^{(a,b)}$ and other neighboring sub-assemblies to construct $A_i^{(x,y)}$. Lastly, Lemma 1.3.14 guarantees that tile placement occurs synchronously. □

Lemma 1.3.16. *Given A_{2i}, which represents the assembly prior to applying Algorithm 14. Every sub-assembly $A_i^{(a,b)} \neq A_i^{(0,0)}$, where $t \in A_i$ has position (a,b), created $len(t.neighbors) - 1$ sub-assemblies.*

Proof. By construction. From Algorithm 12, as $A_i^{(a,b)}$ is reset at the end of the copying procedure, the *num_times_copied* field of the pseudo-seed t_p increases by 1. Since this value starts at 1 (as an already created sub-assembly was used to create $A_i^{(a,b)}$) and since local resetting does not occur until $t_p.num\_times\_copied = len(t_p.neighbors)$, $A_i^{(a,b)}$ creates $len(t_p.neighbors) - 1$ neighboring sub-assemblies. The exception is when $A_i^{(a,b)} = A_i^{(0,0)}$, in which case the number of created sub-assemblies is $len(t_o.neighbors)$. □

Theorem 1.3.17. *Given A_i, which represents the assembly prior to applying Algorithms 5 - 14. After global resetting, as described in Algorithm 14, the resulting assembly A is a base-assembly with $(A)_\Lambda = (A_{2i})_\Lambda$.*

Proof. From Lemma 1.3.16, we get that the total number of sub-assemblies created is $|A_i| - 1$ (excluding the initial sub-assembly $A_i^{(0,0)}$). From Theorem 1.3.15, each of these created copies is a translation of some $A_i^{(a,b)}$ in each cardinal direction d either c^i units horizontally or d_i units vertically in which $d \in t.neighbors$ for $t \in A_i$ with position (a,b). Thus, for each tile $t \in A_i^{(0,0)}$ with position (x,y), a sub-assembly $A_i^{(0,0)} + (x \cdot c^i, y \cdot d^i)$ is created. The resulting assembly A then satisfies $(A)_\Lambda = \{(a,b) + (c^i v, d^i u) | (a,b) \in (A_i)_\Lambda, (v,u) \in G_i\} = (A_{2i})_\Lambda$.

We now show that A becomes a base-assembly after global resetting by considering the 5 requirements from Definition 1.3.1. From Lemma 1.3.8, we get that all tiles are globally reset. From Algorithm 14, this resetting includes setting all fields with default values to their default values, satisfying (5). The exception to this is with respect to the key tiles, where tiles with

$t.new\_key\_tile = True$ become the new key tiles for A. From Algorithm 7, this occurs only when the key tile being copied is also a pseudo-seed, which can only occur once per each direction. Thus, at most 4 new key tiles are chosen for the new assembly A, one for each direction, satisfying (1).

However, to also satisfy (2), it must be true that the *key_d* fields for each tile are updated to point to these new key tiles. From Algorithm 14, the global resetting procedure occurs starting from terminal tiles, excluding the origin seed. If a terminal tile t is not a key tile for any direction, then the key tile must exist in some sub-assembly stemming from $t.origin\_direction$. Similar logic can be applied for any tile t with $len(t.neighbors) \geq 2$; if the key tile does not exist in any neighboring sub-assemblies, then the key tile must exist in some sub-assembly in direction $t.origin\_direction$. The only time a direction is not assigned to a tile is when the key tile is found. Thus, at the end of the global resetting procedure, (2) holds.

In the case of (3), this can be shown via induction. For the base case, when no tiles have been placed yet, assembly A_i starts as a base-assembly. When the copying procedure commences and the first tile t is placed adjacent to t_d, $t.origin\_direction$ is set to the direction of t_d. Thus (3) holds for the assembly $A_i \cup t$. Assume this then holds for any number of placed tiles $A_i \cup (t_1, \ldots, t_j)$. Let t_{j+1} be the next tile placed from tile t_j. The *origin_direction* field of t_{j+1} is set to the direction of t_j. Since (3) holds for $A_i \cup (t_1, \ldots, t_j)$, it also holds for $A_i \cup (t_1, \ldots, t_j) \cup t_{j+1}$.

Showing (4) is slightly less trivial, as the *neighbors* field for some tiles is updated, resulting in a new embedded graph G. However, from Algorithm 14, this updating occurs only for tiles $t \in A_i^{(a,b)}$ and $t' \in A_i^{(x,y)}$ in which t and t' are adjacent and in which $A_i^{(a,b)}$ was used to create $A_i^{(x,y)}$. Since there is exactly one path between any 2 tiles in each $A_i^{(a,b)}$ and since each $A_i^{(x,y)}$ was created by exactly one other $A_i^{(a,b)}$, there exists exactly 1 path between any 2 tiles in A. □

1.3.10 Main Results

We now restate the main results of this paper and finish the analysis.

Theorem 1.3.1. *For every feasible generator G, there exists a seeded TA system* Γ *which strictly builds the corresponding fractal with* $O(1)$ *states, transitions, and affinities.*

Proof. By construction. Given a generator G, initialize an assembly A as described in Section 1.3.6. Then, following Sections 1.3.7 and 1.3.8 (using Algorithms 5, 6, 7, 8, 9, 10, 11, 12, 13, and 14), create system Γ which, by Theorem 1.3.17, creates a new assembly A_{2i} with $(A_{2i})_\Lambda = G_{2i}$. Since A_{2i} is a base-assembly, repeat with A_{2i} as the new generator for Γ. Doing so infinitely then strictly builds the corresponding fractal.

Since each tile can have at most 4 neighbors, the information stored within each tile must be constant. Thus, the number of transition and affinity rules must also be constant. □

Theorem 1.3.2. *There exists a single seeded TA system Γ that, given as input any feasible generator G as a seed, will strictly build the corresponding fractal for G.*

Proof. An important detail about the constructions from Sections 1.3.6, 1.3.7, and 1.3.8 is that tile placement and signal propagation are not hard-coded to the shape of the given generator. Following from Theorem 1.3.1, by considering all possible valid states, transitions, and affinity rules, there must then exist a single seeded TA system Γ which can strictly build any feasible generator at temperature 1. □

1.3.11 Conclusion

This paper extends the work of [53], showing that there not only exists a seeded TA system that can strictly build any fractal with a feasible generator, but also that there exists a *single* system that can do so at temperature 1. This approach takes advantage of the ability to encode information in the state of each tile, allowing for signal transmission and synchronization despite the non-deterministic nature of the model at hand and the geometric bottlenecks of many fractals. While this paper fully characterizes fractal generation in seeded TA, there are still some interesting open questions at hand:

- Are there more efficient ways to build such fractals at higher temperatures? In other words, is it possible to reduce the number of states, transitions, and affinities by encoding some information in the strength of each bond?

- Does there exist a natural extension to the model that can handle non-feasible generators?

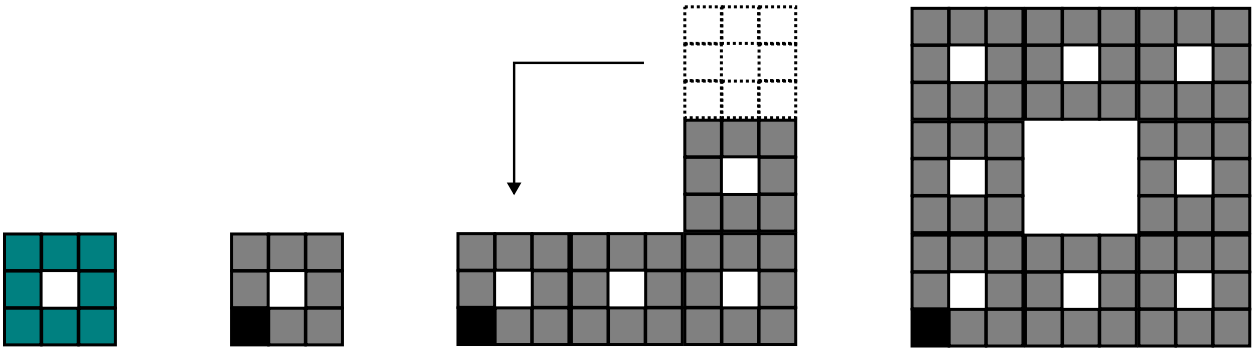


Figure 1.1: From left to right: the generator, the seed assembly (with the tile in black representing the origin tile), the assembly at the start of step 4 and the assembly at stage 2 (or the end of stage 1).

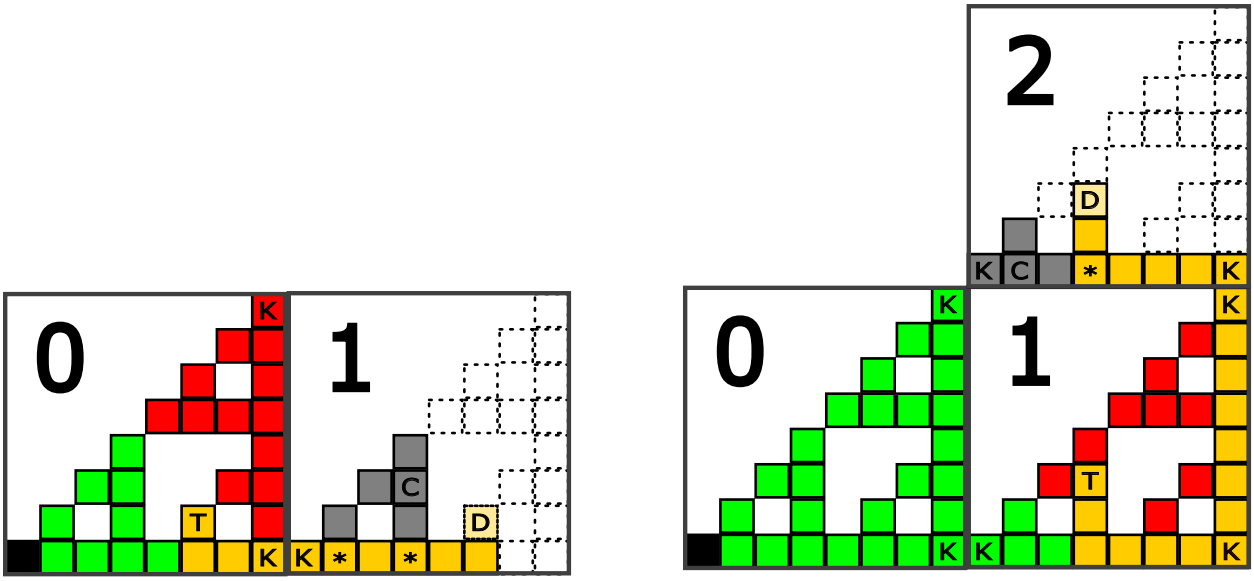


Figure 1.2: The Sierpinski triangle starting from stage 3 with $m = 2$ steps. The tile marked T is sending a signal (yellow) to place itself at position D. Tiles marked K are key tiles (origin tile is also a key tile, just not labeled). Tiles marked $*$ have a cap in the direction of the gray placed tiles. Tiles marked C had 2 caps, so the cap shifted to the tiles marked $*$.

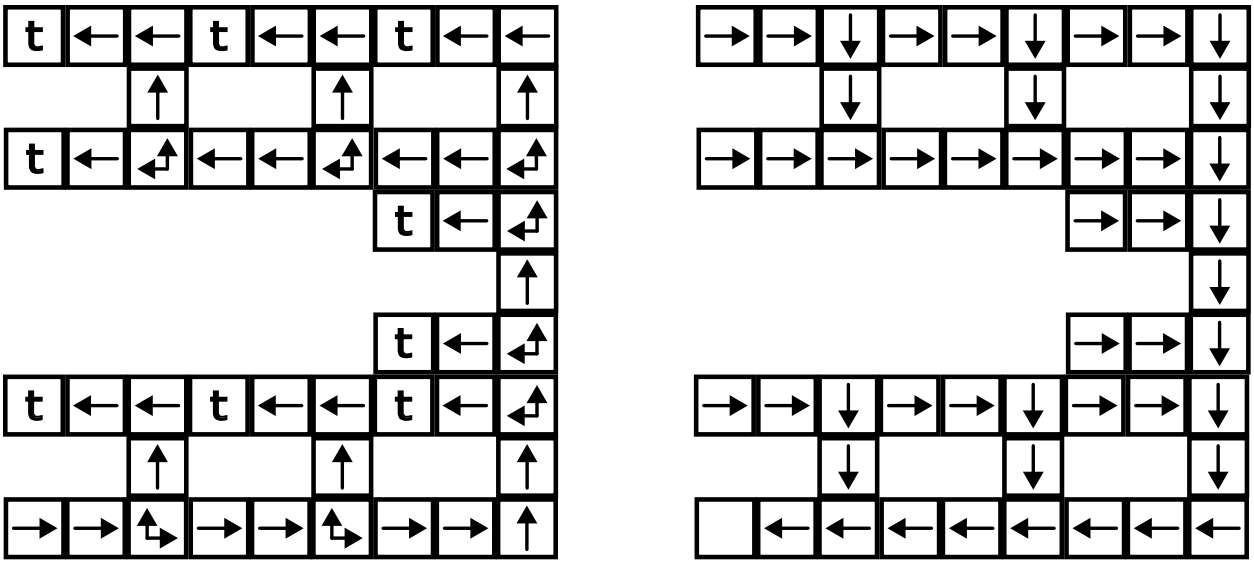


Figure 1.3: An example of the 'next' (left) and 'previous' (right) directions for each tile. Tiles marked *t* are terminal tiles, meaning they have no 'next' direction. The only tile without a 'previous' direction is the origin tile, which is the tile located at the bottom left. Note that the 'next' and 'previous' directions at each tile do not always include all adjacent tiles.

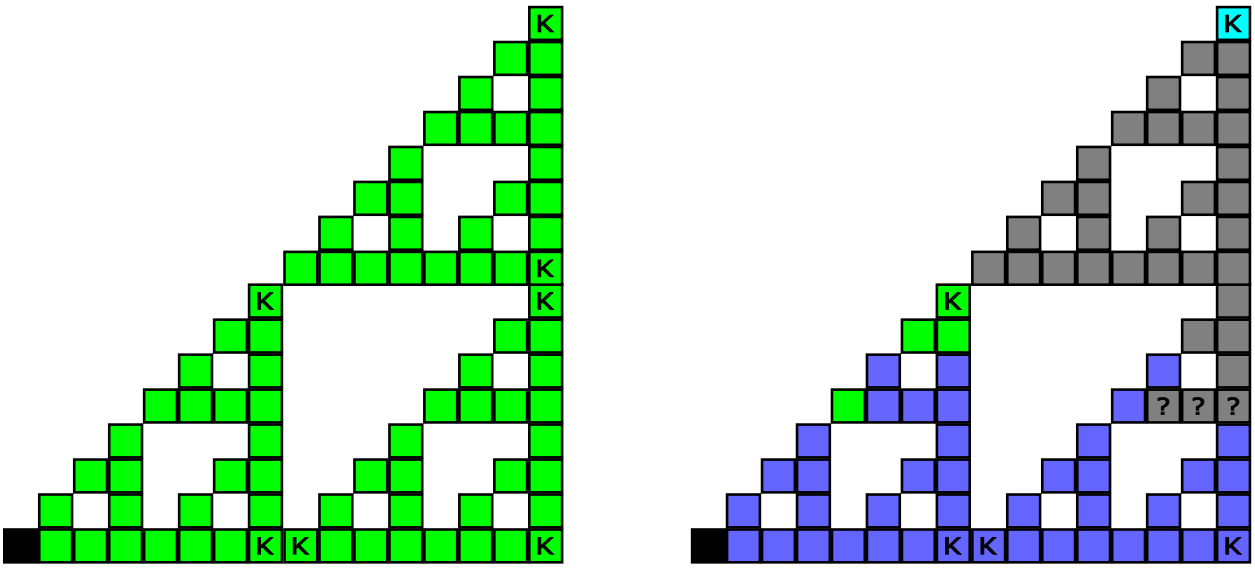
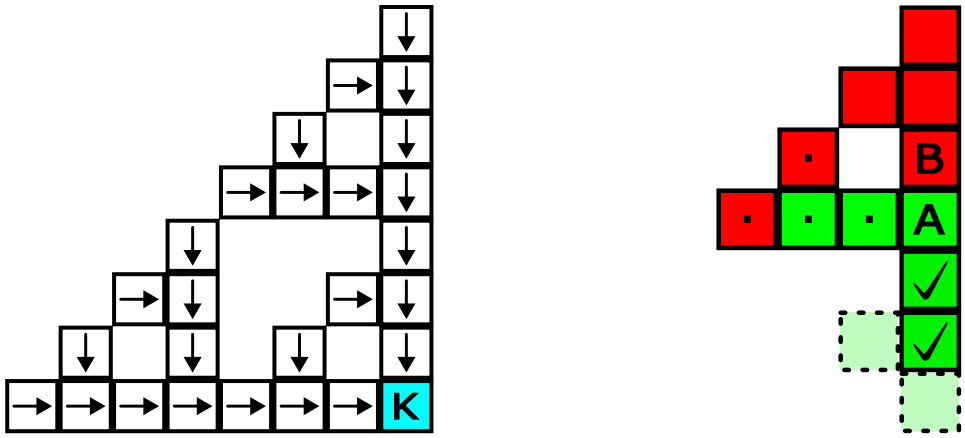


Figure 1.4: The Sierpinski triangle resetting at the end of stage 3. Tiles marked ? are waiting for the sub-assemblies adjacent to reset (the blue tiles to the north and west). Gray tiles have been reset. Blue tiles are transmitting the reset signal. Light blue tiles marked 'K' are the new key tiles for the assembly.



(a) (b)

Figure 1.5: (a) An example of the direction stored at each tile for t_E, the tile marked K. (b) An example of how sub-assemblies work. The check mark denotes the sub-assemblies to the south of tile A are completed. The sub-assemblies to the west and north of A, however, are not. As a result, $SUB_S(A) =$ complete, $SUB_W(A) =$ incomplete and $SUB_N(A) =$ incomplete. Tiles marked with . are part of $SUBASM_W(A)$. Note that B does not start placing itself until $SUB_W(A)$ and $SUB_S(A)$ are both marked completed.

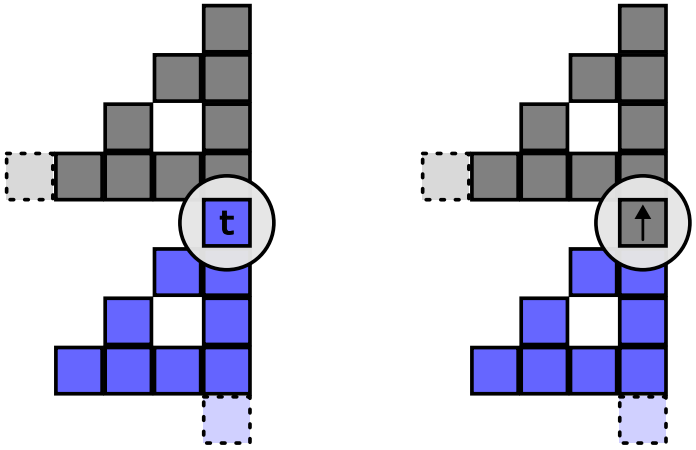


Figure 1.6: The highlighted tile is initially set as terminal. Since the tile used to be a key tile, resetting also updates the 'next' direction if appropriate.

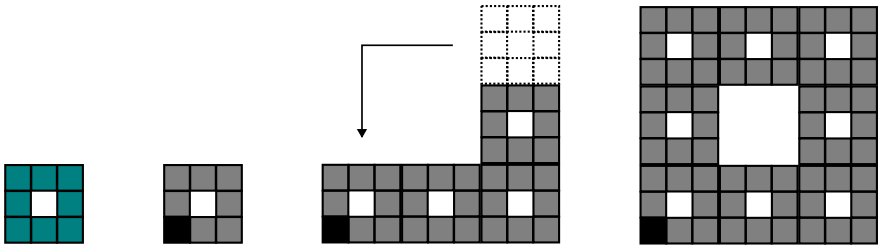


Figure 1.7: A generator for the Sierpinski square and how it's built. From left to right: a feasible generator, the fractal in stage 1, the fractal between stages 1 and 2, and the fractal in stage 2.

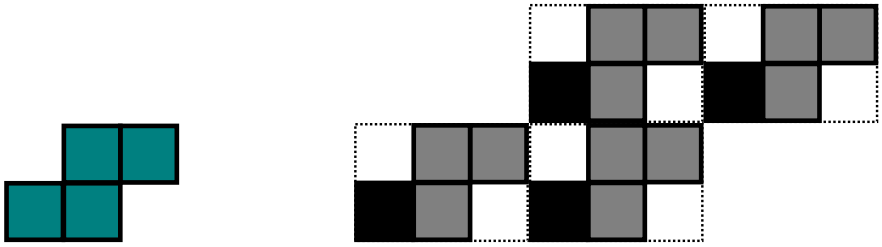


Figure 1.8: An example of a non-feasible generator (left) and the fractal in stage 2 (right). Note that the fractal is no longer connected in stage 2, as there does not exist a pair of points that lie on the left/right edges of the bounding box for the generator.

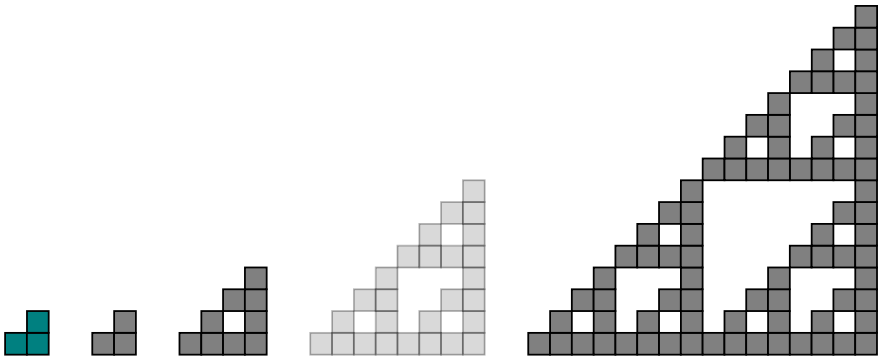
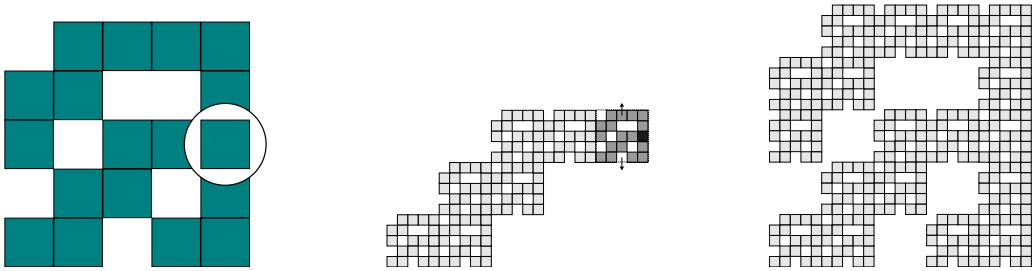
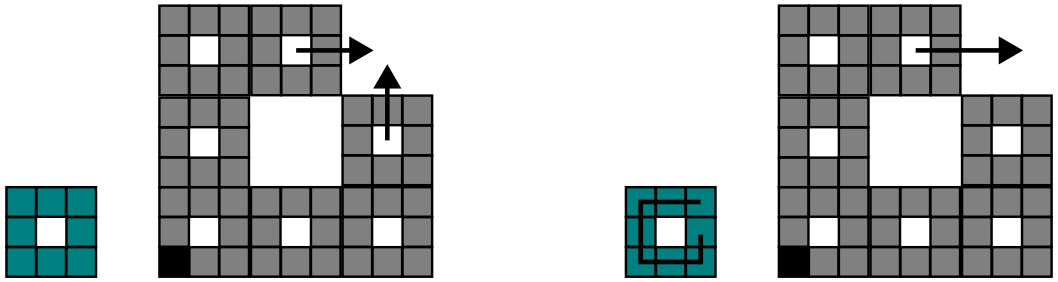


Figure 1.9: The first 4 stages of the Sierpinski triangle. The methods used throughout this paper take a given assembly A_i and transform it into a new assembly A_{2i} with respect to the generator. In this example, an assembly A_2 is taken directly to assembly A_4, passing over A_3 (shown in light gray).



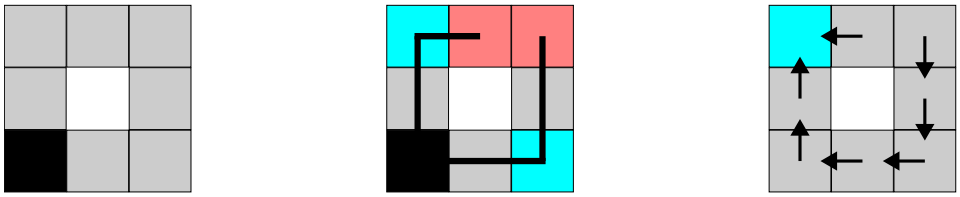
(a) (b) (c)

Figure 1.10: (a) A feasible generator. (b) The fractal between stages 1 and 2. The region shaded dark corresponds to the circled tile from (a), with the black tile denoting the pseudo-seed for this region. (c) The fractal in stage 2. This fractal becomes the new generator, and the process is repeated.



(a) (b)

Figure 1.11: (a) Competition between neighboring regions to construct the region in the top right corner. (b) With an ST of the shape (left), the region in the top right corner is now constructed solely using the top central region.



(a) (b) (c)

Figure 1.12: (a) An initial seed assembly. (b) An ST of the seed assembly from (a), with the origin tile colored black, terminal tiles colored red, and key tiles colored aqua (including the origin tile). (c) The direction from each tile to the north key tile. Note that these directions are dependent on the ST derived from (b).

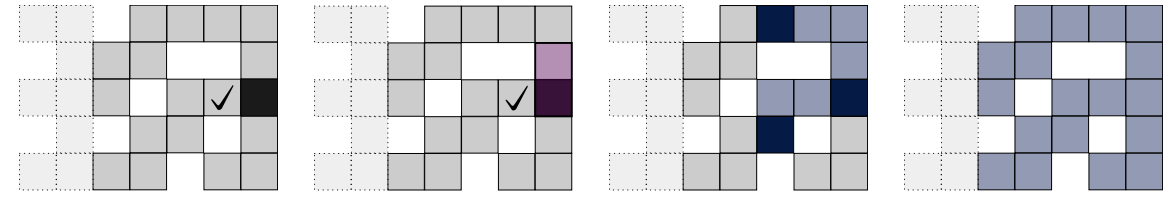


Figure 1.13: (a) A sub-assembly $A_i^{(4,2)}$ with the pseudo-seed in black. The tile with the checkmark denotes that the sub-assembly $A_i^{(3,2)}$ exists. (b) Choosing a copy direction at random from the neighbors of the pseudo-seed that do not have a check-mark. In this case, the northern direction is chosen. (c) The copy direction signal is propagated through the assembly. The dark blue tiles have not yet propagated the copy direction signal to all neighboring tiles (counter $< len(t.neighbors)$). The light blue tiles have fully propagated the signal to all neighboring tiles (counter $= len(t.neighbors)$). (d) $A_i^{(4,2)}$ at the end of the choosing-a-copy-direction procedure.

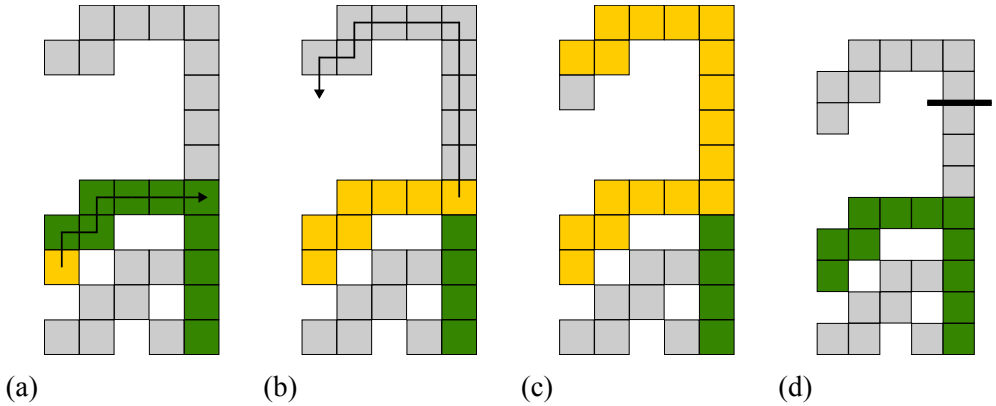


Figure 1.14: The copying procedure for a given tile. (a) A signal is passed to the key tile for the copy direction. In this case, the copy direction is north. (b) The signal then is transferred to the sub-assembly being created and follows the 'next' direction. (c) When a tile no longer exists in the 'next' direction, the tile is placed. (d) The signal is retraced and the newly placed tile is marked as completed. Since the placed tile is terminal, a cap is sent back with the signal and left at the first tile t with $len(t.caps) < len(t.neighbors) - 1$. The cap is denoted as the black line.

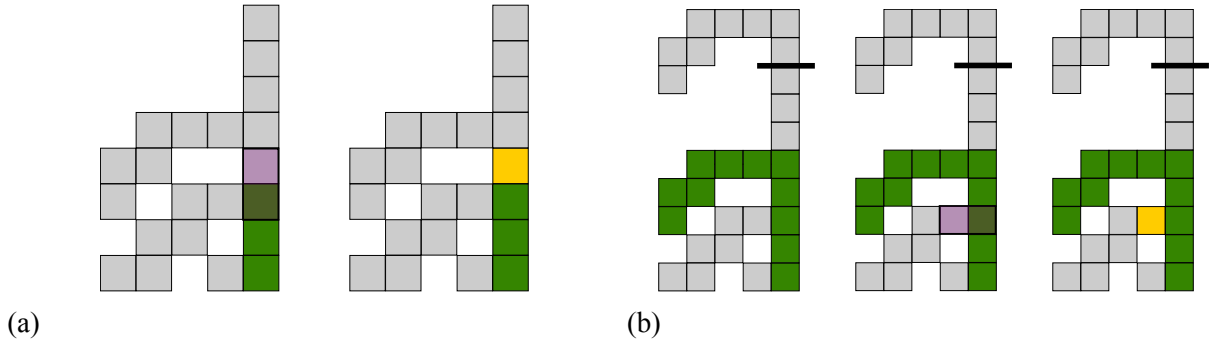


Figure 1.15: Choosing the next tile to get placed, which is based on the 'next' direction of each tile. (a) For non-terminal tiles, the next tile is chosen using the 'next' direction. (b) For terminal tiles, a signal is sent back until a tile has a neighbor that has yet to be copied. The next tile is then chosen using the 'next' direction.

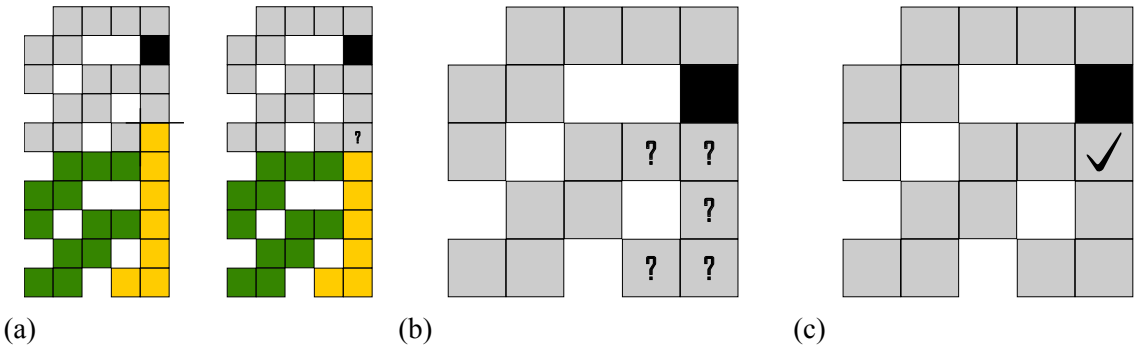


Figure 1.16: Readying the newly created sub-assembly. (a) Once all tiles have been placed, a signal is initiated to look for the pseudo-seed. (b) The signal is propagated through the sub-assembly. (c) Once the signal reaches the pseudo-seed, the choose-a-copying-direction procedure can begin.

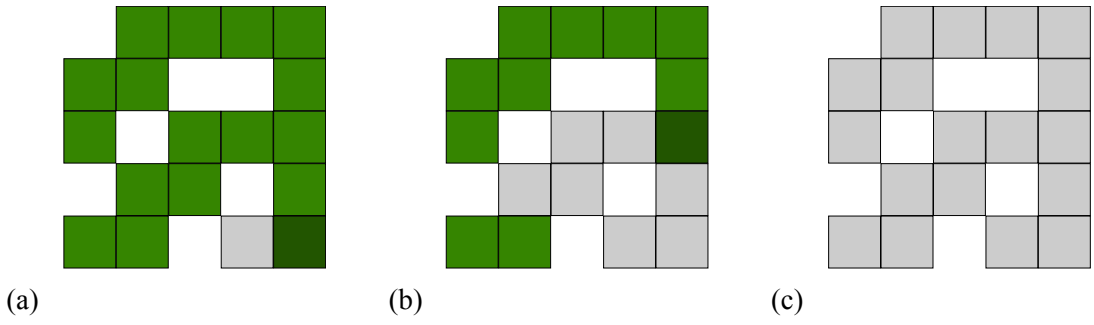
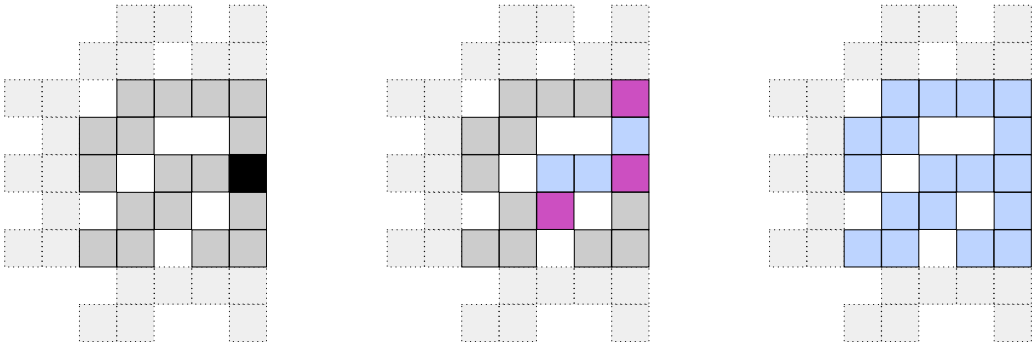
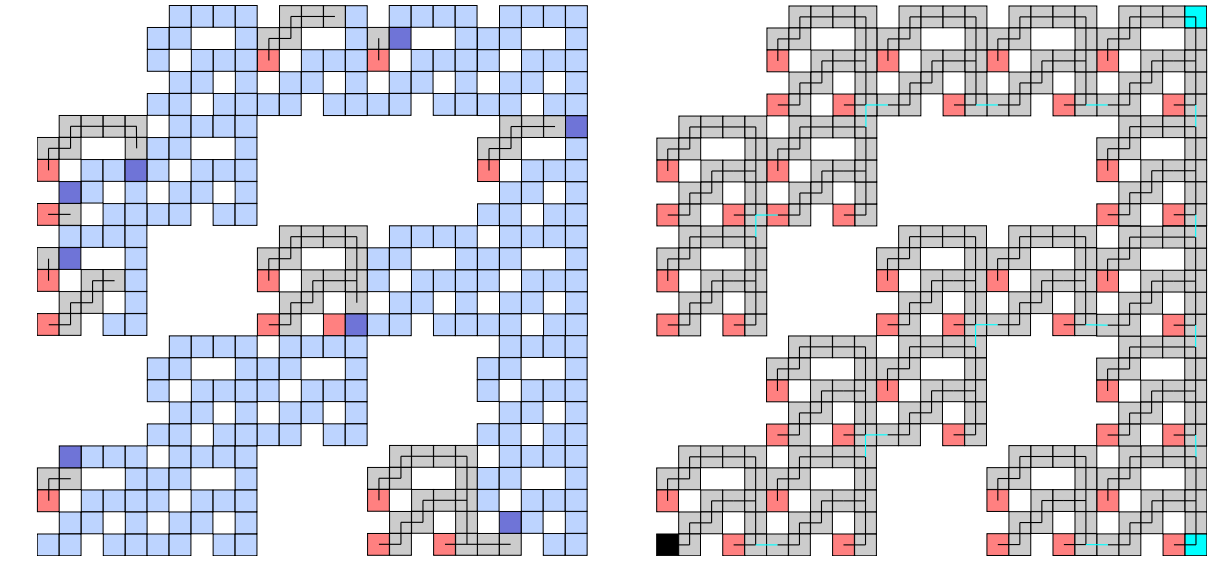


Figure 1.17: Resetting the copied sub-assembly. (a) The process is initiated, starting from the first tile that was placed (bottom right). The gray tiles denote a tile that has been reset. The dark green tile denotes not all neighboring tiles have received the reset signal ($len(t.neighbors) > t.counter$). The light green tiles have yet to be reset. (b) The signal is propagated throughout the sub-assembly. (c) The sub-assembly at the end of the resetting procedure.



(a) (b) (c)

Figure 1.18: Locally resetting a sub-assembly once it is no longer needed to create another sub-assembly. (a) The sub-assembly at the start of the local resetting procedure. The black tile denotes the pseudo-seed for this sub-assembly. (b) The local resetting signal is propagated through the sub-assembly. Light blue tiles have fully transmitted signals to neighboring tiles (counter $= 0$). Purple tiles have not fully transmitted signals to neighboring tiles (counter > 0). (c) The sub-assembly at the end of the local resetting procedure.



(a) (b)

Figure 1.19: Global resetting for the new assembly. (a) The global reset signal is initiated from the new terminal tiles (red). Dark blue tiles are tiles that are waiting for neighboring sub-assemblies to finish resetting (counter > 0). Light blue tiles do not reset until prompted by an adjacent blue tile whose counter has reached 0. Gray tiles are tiles that have been reset. (b) The assembly at the end of the global reset procedure. The aqua-colored edges denote the new edges added between sub-assemblies after resetting. The aqua tiles are the new key tiles (including the origin tile in black).

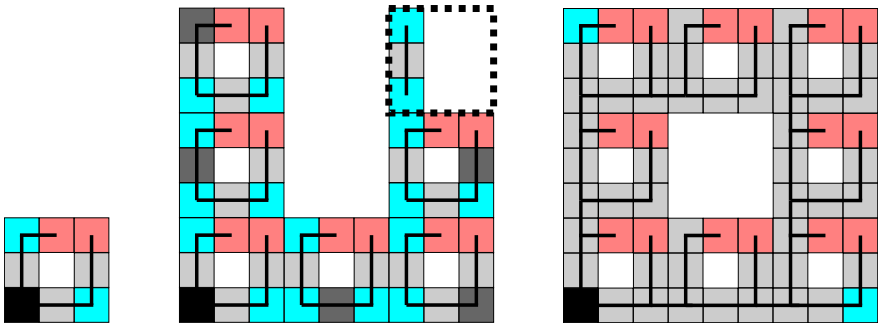


Figure 1.20: Examples of base-assemblies and base-sub-assemblies. Key tiles are colored aqua, terminal tiles are colored red, pseudo-seeds are colored dark gray and the black tile denotes the origin tile. The center assembly is not a base-assembly as it 1) does not have one key tile per direction, 2) does not have paths between all tiles and the origin tile (black), and 3) not all fields are defaulted (the top right corner is currently being copied). Additionally, the dotted region is not a base-sub-assembly, as there does not exist one key tile per direction and there is no pseudo-seed.

Algorithm 1 The full algorithm for building fractal X starting from generator G. At the end of the algorithm, the newly created assembly is treated as the new seed assembly S, and the process restarts.

Input: A generator G, seed assembly S, origin tile t_o

```
1:  procedure BuildFractal
2:      stack ← [t_o]
3:      while len(stack) > 0 do
4:          t_p ← stack.pop()
5:          ChooseCopyDirection(t_p)
6:          t ← LocateFirstTile(t_p)                ▷ Locates first tile to be copied
7:          while True do                           ▷ Copy all tiles in sub-assembly
8:              CreateCopy(t)
9:              TransferSignal(t)
10:             t' ← RetraceSignal(t)
11:             t ← LocateNextTile(t)
12:             if num_completed_sub-assemblies(t) = len(t.neighbors) then
13:                 stack.append(LocatePseudoSeed(t'))
14:                 ResetCopiedSubAssembly(t))
15:                 if t_p.num_times_copied = len(t_p.neighbors) then
16:                     MarkingSubAssembly(t_p)     ▷ No pending directions to copy
17:                 else
18:                     stack.append(t_p)           ▷ Still pending direction to copy
19:                 break
20:     GlobalResetting                 ▷ No sub-assemblies have pending directions to copy
```

Algorithm 2 The 'next' direction at a given tile t. N, E, W, S represent the 4 cardinal directions.

Input: A tile t
Output: A direction $d \in \{N, E, W, S\}$

1: **procedure** Next
2: **if** $N \notin t.caps \wedge N \in t.neighbors$ **then return** N
3: **else if** $E \notin t.caps \wedge E \in t.neighbors$ **then return** E
4: **else if** $W \notin t.caps \wedge W \in t.neighbors$ **then return** W
5: **else return** S

Algorithm 3 The 'next' tile to be placed from tile t. P denotes a 'producing' state. N, E, W, S represent the 4 cardinal directions.

Input: A tile t
Output: A tile t_a

1: **procedure** NextTilePlaced
2: **if** $retrieve\_tile(t, N).state = P \wedge N \in t.neighbors$ **then**
3: **return retrieve_tile(*t*, *N*)**
4: **else if** $retrieve\_tile(t, E).state = P \wedge E \in t.neighbors$ **then**
5: **return retrieve_tile(*t*, *E*)**
6: **else if** $retrieve\_tile(t, W).state = P \wedge W \in t.neighbors$ **then**
7: **return retrieve_tile(*t*, *W*)**
8: **else return retrieve_tile(*t*, *S*)**

Algorithm 4 The direction of the 'breadcrumb' trail at a tile t. $'W'$ and $'M'$ denote 'waiting' and 'maybe' states, respectively. N, E, W, S represent the 4 cardinal directions.

Input: A tile t
Output: A direction $d \in \{N, E, W, S\}$

1: **procedure** Breadcrumb
2: **if** $state\_N = 'W' \vee state\_N = 'M'$ **then return** N
3: **else if** $state\_E = 'W' \vee state\_E = 'M'$ **then return** E
4: **else if** $state\_W = 'W' \vee state\_W = 'M'$ **then return** W
5: **else return** S

Algorithm 5 Choosing a copying direction and propagating choice. $'?'$ denotes the pseudo-seed/origin tile can choose a copying direction.

Input: A tile t_o

```
1:  procedure ChooseCopyDirection
2:      cd ← None
3:      if t_o.original_seed or t_o.copy_direction ='?' then
4:          for d in t_o.neighbors do
5:              adj_tile ← retrieve_tile(t_o, d)
6:              if adj_tile.sub_assembly_copied = False then
7:                  cd ← d
8:                  adj_tile.sub_assembly_copied ← True
9:                  adj_tile.will_be_pseudo_seed ← True
10:                 break
11:     if cd ≠ None then
12:         stack ← [t_o]
13:         while len(stack) > 0 do
14:             t ← stack.pop()
15:             t.counter ← 0
16:             if t.terminal = True then t.counter++
17:             for d in t.neighbors do
18:                 adj_tile ← retrieve_tile(t, d)
19:                 if adj_tile.copy_direction = None then
20:                     adj_tile.copy_direction ← cd
21:                     t.counter++
22:                     stack.append(adj_tile)
```

Algorithm 6 Locating the first tile to be placed. $'?'$ denotes a 'searching' signal. $'*'$ denotes the respective tile is the key tile for some direction d_c.

Input: A tile t_o
Output: A tile t

```
1:  procedure FirstTile
2:      t ← t_o
3:      d_c ← t.copy_direction
4:      t.transfer ←'?'
5:      while t.key_OPP(d_c) ≠ '*' do
6:          adj_tile = retrieve_tile(t, t.key_OPP(d_c))
7:          t.transfer ← None
8:          adj_tile.transfer ← '?'
9:          t ← adj_tile
10:     t.first_tile ← True
11:     t.can_place ← True
12:     t.transfer ← None
13:     return t
```

Algorithm 7 Creating a copy of a tile t to be transferred to the created sub-assembly. $'*'$ denotes the respective tile is the key tile for some direction d.

Input: A tile t
Output: None

```
1:  procedure CreateCopy
2:      nt ← t                                          ▷ Creating a copy of t
3:      nt.copied ← True
4:      if t.pseudo_seed then
5:          if t.key_d = '*' then
6:              nt.new_key_tile ← True
7:              nt.sub_assembly_copied ← True
8:      else if t.origin_seed then
9:          if t.key_d = '*' then
10:             nt.new_key_tile ← True
11:             nt.sub_assembly_copied ← True
12:     else
13:         nt.new_key_tile ← False
14:     if nt.will_be_pseudo_seed then
15:         nt.pseudo_seed ← True
16:         nt.will_be_pseudo_seed ← False
17:         nt.num_times_copied += 1
18:         t.will_be_pseudo_seed ← False
19:     t.transfer ← nt
```

Algorithm 8 Transferring signal to place a tile in direction d. $'W'$ denotes a 'waiting' state. $'*'$ denotes the respective tile is the key tile for some direction d.

Input: A tile t
Output: None

```
1:  procedure TransferSignal
2:      while t.copied=False ∧ t.key_d ≠ '*' do
3:          adj_tile ← retrieve_tile(t, t.key_d)
4:          if adj_tile.counter = len(adj_tile.neighbors) then
5:              adj_tile.transfer ← t.transfer
6:              t.transfer ← None
7:              adj_tile.temp ← adj_tile.OPP(t.key_d)
8:              adj_tile.state_OPP(t.key_d) ← 'W'
9:              t ← adj_tile
10:     if retrieve_tile(t, d) = None then
11:         Place t.transfer in direction d
12:         t.new_neighbors.append(d)
13:         adj_tile ← retrieve_tile(t, d)
14:         adj_tile.origin_direction ← OPP(d)
15:         adj_tile.new_neighbors.append(OPP(d))
16:     else
17:         t ← retrieve_tile(t, d)
18:         loop
19:             adj_tile ← retrieve_tile(t, next(t))
20:             if adj_tile = None then
21:                 Place t.transfer in direction d
22:                 break
23:             else
24:                 adj_tile.transfer ← t.transfer
25:                 t.transfer ← None
26:                 adj_tile.temp ← adj_tile.state_OPP(next(t))
27:                 adj_tile.state_OPP(next(t)) ← 'W'
28:                 t ← adj_tile
```

Algorithm 9 Retracing signal to mark the tile being copied as finished. $'W'$, $'M'$, $'N'$ and $'F'$ denote 'waiting', 'maybe', 'not completed', and 'completed' states, respectively.

Input: A tile t
Output: A tile t_p

```
1:  procedure RetraceSignal
2:      t.transfer ← None
3:      t_p ← None
4:      prev_tile ← retrieve_tile(t, breadcrumb(t))
5:      if len(t.caps) = t.neighbors - 1 ∧ prev_tile.copy_direction = d then t_p ← t
6:      while prev_tile.status ≠ 'W' do
7:          t.state_breadcrumb(t) ← t.temp
8:          prev_tile.state_breadcrumb(prev_tile) ← 'M'
9:          t.temp ← None
10:         if len(t.caps) = len(t.neighbors) - 1 then
11:             if prev_tile.copy_direction = d then
12:                 Place t.transfer in direction d
13:             else
14:                 prev_tile.caps.append(OPP(breadcrumb(t)))
15:         t ← prev_tile
16:     t.state_breadcrumb(t) ← 'N'
17:     prev_tile.status ← 'F'
18:     return t_p
```

Algorithm 10 Locating the next tile to be placed. $'Y'$ denotes a 'complete' state.

Input: A tile t
Output: A tile t_a

```
1:  procedure LocateNextTile
2:      if t.terminal then
3:          adj_tile ← retrieve_tile(t, t.neighbors)
4:          adj_tile.state_OPP(t.neighbors) ← 'Y'
5:          t ← adj_tile
6:          while num_comp_sub-assemblies(t) = len(t.neighbors) do
7:              for n in t.neighbors do
8:                  adj_tile ← retrieve_tile(t, n)
9:                  if num_comp_sub-assemblies(adj_tile) < len(t.neighbors) then
10:                     adj_tile.state_OPP(n) ← 'Y'
11:                     t ← adj_tile
12:                     break
13:     adj_tile ← retrieve_tile(t, next_tile_placed(t))
14:     adj_tile.state_OPP(next_tile_placed(t)) ← 'Y'
15:     adj_tile.can_place ← True
16:     t_a ← t
17:     return t_a
```

Algorithm 11 Locating the pseudo-seed of the created sub-assembly. $'?'$ denotes a 'searching' signal.

Input: A tile $t_{OPP(d_c)}$
Output: A tile t_p

```
1:  procedure LocatePseudoSeed
2:      stack ← [t_OPP(d_c)]
3:      while len(stack) > 0 do
4:          t ← stack.pop()
5:          if t.pseudo_seed then
6:              t_p ← t
7:              return t_p
8:          t.copy_direction ← '?'
9:          for d in t.neighbors do
10:             adj_tile ← retrieve_tile(t, d)
11:             if adj_tile.copy_direction ≠ '?' then stack.append(adj_tile)
```

Algorithm 12 Resetting the copied sub-assembly. $'r'$ denotes a 'reset-ready' state.

Input: A tile $t_{OPP(d_c)}$
Output: None

```
1:  procedure ResetCopiedSubAssembly
2:      stack ← [t_OPP(d_c)]
3:      while len(stack) > 0 do
4:          t ← stack.pop()
5:          t.copy_direction ← 'r'
6:          for d in t.neighbors do
7:              adj_tile ← retrieve_tile(t, d)
8:              if adj_tile.copy_direction ≠ 'r' then
9:                  stack.append(adj_tile)
10:                 t.counter++
11:         t.status ← None
12:         t.caps ← None
13:         t.temp ← None
14:         t.transfer ← None
15:         t.can_place ← False
16:         t.first_tile ← False
17:         if t.pseudo_seed ∨ t.original_seed then t.num_times_copied++
```

Algorithm 13 Marking a sub-assembly with 'ready to reset' states. $'r'$ denotes a 'reset-ready' state.

Input: A tile t_p
Output: None

```
1:  procedure MarkingSubAssembly
2:      stack ← [t_p]
3:      while len(stack) > 0 do
4:          t ← stack.pop()
5:          t.transfer ← 'r'
6:          t.counter ← len(t.neighbors) - 1
7:          if t.pseudo_seed then t.counter++
8:          if len(t.new_neighbors) > 1 then
9:              t.terminal ← False
10:         else
11:             t.terminal ← True
12:             t.key_d ← None
13:         for n in t.neighbors do
14:             adj_tile ← retrieve_tile(t, n)
15:             if adj_tile.transfer ≠ 'r' then
16:                 stack.append(adj_tile)
17:                 t.counter−−
```

Algorithm 14 Global Resetting from Terminal Tiles. $'N'$ denotes a not completed state. $'r'$ denotes a 'reset-ready' state.

```
1:  procedure GlobalResetting
2:      stack ← [all terminal tiles]
3:      while len(stack) > 0 do
4:          t ← stack.pop()
5:          t.counter−−
6:          if t.counter = 1 ∧ t.transfer = 'r' then
7:              reset(t)
8:              for n in t.neighbors do
9:                  adj_tile ← retrieve_tile(t, n)
10:                 if t.terminal ∧ t.new_key_tile then
11:                     if t.key_d then adj_tile.key_d ← OPP(n)
12:                 if adj_tile.transfer = 'r' then stack.append(adj_tile)
13:                 t.state_d ← 'N'
14:             t.transfer ← None
```

CHAPTER II

REACHABILITY IN CHEMICAL REACTION NETWORKS

2.1 Chapter Overview

This section focuses on the reachability problem in Chemical Reaction Networks (CRNs), which asks: given some initial configuration of species and a target configuration, is there a sequence of rules that can be applied to go from the initial configuration to the target configuration? As this problem has extensively been studied since the 1960's, we consider reachability in variations of CRNs, including void CRNS (CRNs where rules always delete species, never produce) and Surface CRNs (species are placed at some fixed point on a grid).

2.2 Deletion-Only Chemical Reactions Networks

2.2.1 Overview

This work was in collaboration with the following authors: Bin Fu, Timothy Gomez, Austin Luchsinger, Aiden Massie, Marco Rodriguez, Adrian Salinas, Robert Schweller, and Tim Wylie. My main contribution to this paper is the $(k, k-1)$ result for general (basic) CRNs and a review of the $(2,0)$ and $(2,1)$ result for general CRNs.

2.2.2 Introduction

2.2.2.1 Background. In molecular programming, Chemical Reaction Networks [7, 8] have become a staple model for abstracting molecular interactions. The model consists of a set of chemical species (formal symbols) as well as a set of reactions that dictate how these species interact. As an example, the reaction $A + B \rightarrow C + D$ describes chemical species A and B reacting to form species C and D. While chemical kinetics are commonly modeled continuously using ordinary differential equations, this approximation breaks down for systems with relatively small volumes where species

are present in very low amounts. Such systems are better modeled as discrete Chemical Reaction Networks (which we will hereby be referring to simply as CRNs), where the system state consists of non-negative integer counts of each species and state transitions occur stochastically as a continuous time Markov process [40]. It turns out that this model of chemistry has deep connections to several other well-studied mathematical objects. In fact, CRNs are equivalent [29] to Vector Addition Systems (VASs) [52] and Petri-Nets [72] which were independently introduced to represent and analyze concurrent and distributed processes. This underlying object also appears in the form of commutative semigroups when only fully reversible reactions are considered [54, 62], and as Population Protocols [6] when reactions must have exactly two input and two output elements.

Perhaps the most fundamental problem within these models is that of *reachability* which asks: given an initial configuration and target configuration for a particular system, can the target configuration be reached from the initial configuration by following some sequence of legal transitions [57, 61, 79]? Although interest in this problem dates back to the 1970s and 1980s, it was only recently resolved with a flurry of new results over the past few years [31, 32, 33, 55, 56], which conclude that reachability is Ackermann-complete for these models. Over the course of this 40+ year-long quest to determine the complexity of reachability, scientists also began to discover the centrality of this problem. Several other important problems from seemingly unrelated areas have been reduced to reachability, from system liveness to language emptiness problems and more [43, 81], emphasizing the significance of succinctly classifying reachability.

Despite the closure of this problem for general CRNs, and in fact due to how complicated these systems can be, there is a natural motivation to explore reachability for more restricted systems. Many papers have emerged that investigate reachability in restricted versions of the model [12, 21, 50, 81, 89], but one of the most elementary restrictions is that of [3] and [4]. There, the authors consider deletion-only systems (called *void rules*) where reactions only ever consume species and reduce the size of the system. The interaction rules for these systems can be thought to convert some chemical reactants into inert waste species, or cause system agents to "go offline" and become inactive. Similar work has also investigated reachability in a slightly different version

of size-reducing chemical reaction networks whose stoichiometric matrices are totally unimodular [85, 86]. This limited class of systems permits tractable and intractable problems, placing it along an interesting complexity boundary.

2.2.2.2 Related Work. In [3], the authors studied reachability for this restricted class of CRNs that use (deletion-only) *void rules* (e.g., $A+B \to A$ or $A+B \to \emptyset$). Under this restriction, they prove NP-completeness for reachability using void rules of size (3,0) (3 reactants, no products) and provide a polynomial time algorithm for reachability using void rules of size (2,0) (2 reactants, no products) for unary inputs. In this paper, we continue this line of work by considering reachability in CRNs with void rules of varying numbers of reactants and products. Since void rules arguably constitute the simplest type of reaction, one may wonder what computationally interesting behavior can be achieved with them — and this exact idea is investigated in [4].

The authors of [4] and [5] combine void rules with an experimentally motivated model extension called *step* CRNs. Ideally, experimental scientists use the CRN model to design molecular systems and predict the molecular interactions and possible end-states of those systems. However, contrary to the CRN model in which all the molecules are ready to interact from the start, many experimental designs rely on a progressive addition of molecules that allow some interactions to occur first before moving forward with the experiment. Furthermore, the ability of reactions to add molecules that interact with the rest of the system is precisely what makes experimental applications of theoretical concepts difficult to realize with minimal error at a practical scale [83, 93]. Thus, the step CRN model was introduced which incorporates a sequence of discrete steps, where, at each step, new species are added to the existing CRN and react until no further reactions can occur. This augmentation more closely reflects a laboratory setting in which chemicals are incrementally added in a test tube and left to interact. The work of [4] and [5] shows that, surprisingly, extremely simple void rules are capable of simulating threshold formulas and threshold circuits when steps are added. In this paper, we show that adding even a single step drastically changes the computational complexity of the reachability problem, moving from P (and even NL) to NP-complete in many cases.

2.2.2.3 Our Contributions and Paper Organization. In this paper we present several results for the various types of deletion-only interaction rules. We show that bimolecular deletion-only systems are solvable in polynomial time, with or without catalysts for basic (*single-step*) CRNs, while they are NP-complete even with one additional step. On the other hand, we show that larger void rules (with $k \geq 3$ reactants and $g \leq k-2$ products) are NP-complete even for basic CRNs unless all but one reactant is a catalyst, in which case we show the problem is in P. These results are outlined in Table 2.1, and we visualize the complexity landscape in Figures 2.1a, 2.1b, and 2.1c. When taken together, these results yield the following theorems, which provide a nearly complete characterization of reachability with void rules in CRNs with and without steps.

Theorem 2.2.1. *Reachability for basic CRNs uniformly using size* $(k \geq 3, g \leq k-2)$ *void rules is NP-complete, and is in P when uniformly using any other size void rule.*

Theorem 2.2.2. *Reachability for 2-step CRNs with only void rules that use exactly one reactant is in P, and is NP-complete otherwise.*

Theorem 2.2.3. *Reachability for basic CRNs that use a combination of void rule types is in P for combinations of (2,0) + (2,1) rules as well as* $(k, k-1)^+$ *rules, and is NP-complete for any combination that uses* $(k \geq 3, g \leq k-2)$ *rules.*

We begin the paper by defining void rules, Step Chemical Reaction Networks, and the reachability problem in Section 2.2.3. Sections 2.2.4-2.2.7 establish the complexity of reachability based on the size of rules used by a given system: Section 2.2.4 shows membership in NP for all deletion-only systems. Section 2.2.5 considers bimolecular rule sets that are either all catalytic or non-catalytic, and shows membership in P in either case. It also shows that, in contrast, the problem becomes NP-complete with the inclusion of a second step. Section 2.2.6 expands this to bimolecular systems with mixed catalytic and non-catalytic rules, and Section 2.2.7 considers larger size rules which are polynomially solvable if all but a single reactant serve as catalysts, and NP-complete otherwise.

2.2.3 Preliminaries

2.2.3.1 Chemical Reaction Networks. Let $\Lambda = \{\lambda_1, \lambda_2, \ldots, \lambda_{|\Lambda|}\}$ denote some ordered alphabet of *species*. A configuration $\vec{C} \in \mathbb{N}^{\Lambda}$ is a length-$|\Lambda|$ vector of non-negative integers where $\vec{C}[i]$ denotes the number of copies of species λ_i. For a species $\lambda_i \in \Lambda$, we denote the configuration consisting of a single copy of λ_i and no other species as $\vec{\lambda}_i$. A *rule* or *reaction* is represented as an ordered pair $\gamma = (\vec{R}, \vec{P}) \in \mathbb{N}^{\Lambda} \times \mathbb{N}^{\Lambda}$. $\vec{R}$ contains the minimum counts of each *reactant* species necessary for reaction γ to occur, where reactant species are either *consumed* by the rule in some count or leveraged as *catalysts* (not consumed); in some cases a combination of the two. The *product* vector $\vec{P}$ has the count of each species *produced* by the *application* of rule γ, effectively replacing vector $\vec{R}$. The species corresponding to the non-zero elements of $\vec{R}$ and $\vec{P}$ are termed *reactants* and *products* of γ, respectively.

The *application* vector of γ is $\vec{P} - \vec{R}$, which shows the net change in species counts after applying rule γ once. For a configuration $\vec{C}$ and rule γ, we say γ is applicable to $\vec{C}$ if $\vec{C}[i] \geq \vec{R}[i]$ for all $i \in \Lambda$, and we define the *application* of γ to $\vec{C}$ as the configuration $\vec{C}' = \vec{C} + \vec{P} - \vec{R}$. For a set of rules Γ, a configuration $\vec{C}$, and rule $\gamma \in \Gamma$ applicable to $\vec{C}$ that produces $\vec{C}' = \vec{C} + \vec{P} - \vec{R}$, we say $\vec{C} \rightarrow^1_{\Gamma} \vec{C}'$, a relation denoting that $\vec{C}$ can transition to $\vec{C}'$ by way of a single rule application from Γ. We further use the notation $\vec{C} \rightarrow^*_{\Gamma} \vec{C}'$ to signify the transitive closure of $\rightarrow^1_{\Gamma}$ and say $\vec{C}'$ is *reachable* from $\vec{C}$ under Γ, i.e., $\vec{C}'$ can be reached by applying a sequence of applicable rules from Γ to initial configuration $\vec{C}$. Here, we use the following notation to depict a rule $(\vec{R}, \vec{P})$: $\vec{R}[1]\lambda_1 + \cdots + \vec{R}[|\Lambda|]\lambda_{|\Lambda|} \rightarrow \vec{P}[1]\lambda_1 + \cdots + \vec{P}[|\Lambda|]\lambda_{|\Lambda|}$. For example, a rule turning two copies of species H and one copy of species O into one copy of species W would be written as $2H + O \rightarrow W$.

Definition 2.2.1 (Discrete Chemical Reaction Network)**.** *A discrete chemical reaction network (CRN) $\mathscr{C}$ is an ordered pair (Λ, Γ) where Λ is an ordered alphabet of species, and Γ is a set of rules over Λ.*

A configuration is called *terminal* with respect to a CRN (Λ, Γ) if no rule γ can be applied to it. An initial configuration $\vec{A}$ and CRN (Λ, Γ) is said to be *bounded* if a terminal configuration

is guaranteed to be reached within some finite number of rule applications starting from $\overrightarrow{A}$. We denote the set of reachable configurations of a CRN as $REACH_{\overrightarrow{A},\Lambda,\Gamma}$. We define the subset of reachable configurations that are terminal as $TERM_{\overrightarrow{A},\Lambda,\Gamma}$.

2.2.3.2 Void Rules. A *void* rule is any rule that does not create any new copies of any species types (only deletes). Thus, a void rule either has no products or has products that are a subset of its reactants (in which case these products are termed *catalysts*). The formal definitions of void rules and rule size are as follows.

Definition 2.2.2 (Void rules)**.** *A rule* $(\overrightarrow{R}, \overrightarrow{P})$ *is a* void *rule if* $\overrightarrow{P} - \overrightarrow{R}$ *has no positive entries and at least one negative entry. There are two classes of void rules, catalytic and true void. In catalytic void rules, one or more reactants remain and one or more reactant is deleted after the rule is applied. In true void rules, there are no products remaining.*

Definition 2.2.3. *The* size/volume *of a configuration* $\overrightarrow{C}$ *is* $\texttt{volume}(\overrightarrow{C}) = \sum \overrightarrow{C}[i]$. *When* $\overrightarrow{C}$ *is an initial configuration of a CRN, we refer to* $\Sigma_{\overrightarrow{C}} = \texttt{volume}(\overrightarrow{C})$.

Definition 2.2.4 (size-(i,j) rules)**.** *A rule* $(\overrightarrow{R}, \overrightarrow{P})$ *is said to be a size-*(i,j) *rule if* $(i,j) = (\texttt{volume}(\overrightarrow{R}), \texttt{volume}(\overrightarrow{P}))$. *A reaction is trimolecular if* $i = 3$*, bimolecular if* $i = 2$*, and unimolecular if* $i = 1$.

2.2.3.3 Step CRNs. A step CRN is an augmentation of a basic CRN in which a sequence of additional copies of some system species are added after a terminal configuration is reached. Formally, a step CRN of k steps is an ordered pair $((\Lambda,\Gamma),(\overrightarrow{S}_0, \overrightarrow{S}_1, \overrightarrow{S}_2, \ldots \overrightarrow{S}_{k-1}))$, where the first element of the pair is a normal CRN (Λ,Γ), and the second is a sequence of length-$|\Lambda|$ vectors of non-negative integers denoting how many copies of each species type to add after each step. We define a *step-configuration* $\overrightarrow{C}_i$ for a step CRN as a valid configuration $\overrightarrow{C}$ over (Λ,Γ) along with an integer $i \in \{0,\ldots,k-1\}$ that denotes the configuration's step. We denote the initial volume of step CRN as $\Sigma_{\overrightarrow{S}_0} = \texttt{volume}(\overrightarrow{S}_0)$ and total volume as $\Sigma_T = \sum_{i=0}^{k-1} \texttt{volume}(\Sigma_{\overrightarrow{S}_i})$. Figure 2.2 illustrates a simple step CRN system.

Given a step CRN, we define the set of reachable configurations after each sequential step. To start off, let REACH_1 be the set of reachable configurations of (Λ, Γ) with initial configuration $\overrightarrow{S}_0$, which we refer to as the set of configurations reachable *after step 1*. Let TERM_1 be the subset of configurations in REACH_1 that are terminal. Note that after a single step we have a normal CRN, i.e., 1-step CRNs are just normal CRNs with initial configuration $\overrightarrow{S}_0$. For the second step, we consider any configuration in TERM_1 combined with $\overrightarrow{S}_1$ as a possible starting configuration and define REACH_2 to be the union of all reachable configurations from each possible starting configuration attained by adding $\overrightarrow{S}_1$ to a configuration in TERM_1. We then define TERM_2 as the subset of configurations in REACH_2 that are terminal. Similarly, define REACH_i to be the union of all reachable sets attained by using initial configuration $\overrightarrow{S}_{i-1}$ plus any element of TERM_{i-1}, and let TERM_i denote the subset of these configurations that are terminal. The set of reachable configurations for a k-step CRN is the set REACH_k, and the set of terminal configurations is TERM_k. A classical CRN can be represented as a step CRN with $k = 1$ steps and an initial configuration of $\overrightarrow{A} = \overrightarrow{S}_0$.

Note that our definitions assume only the terminal configurations of a given step are passed on to seed the subsequent step. This makes sense if we assume we are dealing with *bounded* systems, as this represents simply waiting long enough for all configurations to reach a terminal state before proceeding to the next step. In this paper, we only consider bounded void rule systems; we leave more general definitions to be discussed in future work.

2.2.3.4 Reachability. The computational problem studied in this paper is *reachability*. Informally, reachability asks if a given initial configuration $\overrightarrow{A}$ can be turned into a target configuration $\overrightarrow{B}$ by applying a sequence of rules from the given CRN $\mathscr{C}$. The precise problem statement is as follows.

Definition 2.2.5 (Reachability Problem)**.** *Given an initial configuration $\overrightarrow{A}$, a destination (target) configuration $\overrightarrow{B}$, and a step CRN $\mathscr{C}_{\mathscr{S}} = ((\Lambda,\Gamma),(\overrightarrow{S}_0 = \overrightarrow{A}, \overrightarrow{S}_1, \overrightarrow{S}_2, \ldots \overrightarrow{S}_{k-1}))$, determine if $\overrightarrow{B} \in REACH_k$, i.e., is configuration $\overrightarrow{B}$ reachable for the given step CRN. In the case of basic CRNs, this simplifies to: given configurations $\overrightarrow{A}$ and $\overrightarrow{B}$, and basic CRN $\mathscr{C} = (\Lambda,\Gamma)$, determine if $\overrightarrow{B} \in REACH_{\overrightarrow{A},\Lambda,\Gamma}$.*

2.2.4 Membership in NP for Void Rule Systems

We initiate our study of deletion-only systems by observing that reachability stays within the class NP with only void rules, even with step-CRNs. This is straightforward to see in the case of polynomial bounded volume (unary encoded species counts) as each rule reduces the system volume by at least 1. But in the case of binary encoded species counts, the argument is more subtle as such deletion sequences could be exponentially long. To deal with this issue, we use the following *rearrangement* lemma that states that any order of void rule applications can be rearranged such that all applications of a given rule occur in a contiguous sequence. Given this lemma, any sequence of void rule applications can be rearranged into a sequence that can be encoded and verified in polynomial time.

Lemma 2.2.4 (Rearrangement Lemma)**.** *For any sequence of applicable void rules A, there exists a sequence B that is a permutation of A such that all applications of a given rule type occur contiguously.*

Proof. Consider a sequence of applicable void rules A that is not contiguous. Suppose rule x occurs at positions i and j in A, $i < j-1$, and there is at least one non-x rule in between them. Construct a new sequence A' by shifting the x rule at position i up to position $j-1$, and shifting all rules in between down one position. This new sequence must be applicable as the only rule that moved to a higher index in the sequence is of type x, and we know that x is still applicable at position $j-1$ since x is known to be applicable at position j. As this swapping preserves the applicability of the sequence while reducing the number of non-contiguous blocks of one rule type in the sequence, we can repeat this process of swapping rule positions until the sequence is contiguous. □

This lemma implies the existence of a polynomial-time verifiable certificate for 'yes' instances of the reachability problem for step-CRNs, giving us membership in NP.

Theorem 2.2.5. *The reachability problem for step-CRNs with void rules is in NP.*

Proof. As a certificate, we utilize a contiguous sequence of applicable rules for each step of the CRN, which must exist by Lemma 2.2.4. This sequence, while potentially exponential in length, can be encoded with a sequence of rule types accompanied by a count on the number of applications of each rule type. The result of such a sequence can be computed in polynomial time and therefore can serve as a certificate for the reachability problem. □

2.2.5 Bimolecular Rules of Uniform-type: With or Without Catalysts

In this section, we focus on bimolecular systems with either all size-$(2,0)$ rules (non-catalytic) or $(2,1)$ rules (catalytic). Recently, size-$(2,0)$ rule reachability in a single step was proven to be polynomial [3]. For size-$(2,1)$ 1-step systems, we present polynomial-time algorithms for reachability. In contrast, we show that in either scenario the problem becomes NP-complete with the addition of a second step. Later in Section 2.2.6, we consider the scenario of CRNs that use both $(2,0)$ and $(2,1)$ rules together.

2.2.5.1 Bimolecular Void Rules Without a Catalyst: $(\mathbf{2},\mathbf{0})$. In [3], the authors proved that reachability in a CRN system with only size-$(2,0)$ rules is in P by reducing from the perfect *b*-matching problem, which is a generalization of matching. This takes the form of a traditional matching when all *b*-values are 1, and an uncapacitated *b*-matching occurs when all edge capacities are assigned $u(e) = \infty$.

Theorem 2.2.6. *Reachability for basic CRNs with binary encoded species with only rules of size (2, 0) is solvable in* $O(|\Lambda|^2 \log(|\Lambda|)(|\Gamma| + |\Lambda| \log(|\Lambda|)))$ *time [3].*

We now look at the problem with only one additional step, and show that it becomes NP-complete by reducing from the graph 3-colorability (3-COL) problem. Given an instance $\langle G \rangle$, where $G = (V,E)$ is an undirected graph, 3-COL asks if each vertex of G can be assigned one of

three colors such that no adjacent vertices share the same color. An instance of 3-COL $\langle G \rangle$ can be converted into a (2,0) 2-step CRN $\mathscr{C}_{\mathscr{S}}$ as follows.

Species. For each vertex $v \in V$, we create the species v, v_R, v_G, and v_B. v_C represents an assignment of color $C \in \{R, G, B\}$ to vertex v; the v species will be used to represent assigning only one color to vertex v through specific reactions. We also create the species X to verify that a corresponding color assignment of G in $\mathscr{C}_{\mathscr{S}}$ has no adjacent vertices that share a color.

Steps and Rules. In step one (or $\overrightarrow{S}_0$), for each $v \in V$, we add two copies of v and one copy of v_R, v_G, and v_B. We also create the *assignment* rule $v + v_C \to \emptyset$ for each $C \in \{R, G, B\}$. Two of the three assignment rules created for v are applied to its respective species copies, consuming all v copies and two of the three copies of v_C. The remaining v_C copy then corresponds to assigning vertex v the color C. Additionally, for each edge $(i, j) \in E$ and $C \in \{R, G, B\}$, we create the *edge* rule $i_C + j_C \to \emptyset$. If the remaining copy of both i_C and j_C share a color C, they will be deleted by one of the edge rules.

In the second step ($\overrightarrow{S}_1$), we introduce $|V|$ copies of the species X. We also construct the *verification* rule $v_C + X \to \emptyset$ for each $v \in V$ and $C \in \{R, G, B\}$. All existing copies of v_C will be consumed by a X species. Thus, any remaining copies of X in the terminal configuration indicates that some v_C copies were deleted by an edge rule.

Theorem 2.2.7. *Reachability for 2-step CRNs with only rules of size-*$(2,0)$ *is NP-complete, even for unary encoded species counts.*

Proof. We reduce from the graph 3-colorability problem. Given an instance of 3-COL $\langle G \rangle$, we convert G into a 2-step $(2,0)$ CRN $\mathscr{C}_{\mathscr{S}}$, following the construction outlined above, and set $\overrightarrow{A} = \overrightarrow{S}_0$ and $\overrightarrow{B}$ to the *empty configuration* $\overrightarrow{0}$.

Forward Direction. Assume there exists a color assignment in G where no adjacent vertices share a color. By the construction of $\mathscr{C}_{\mathscr{S}}$, a sequence of assignment rules can be applied in $\overrightarrow{A}$ that results in a configuration of one copy of v_C for each vertex that matches the color assignment in G. Denote this new configuration $\overrightarrow{S}'_0$. Since no adjacent vertices share a color in G, no edge rule will be applied in $\overrightarrow{S}'_0$, keeping the count of the v_C copies to $|V|$. $\mathscr{C}_{\mathscr{S}}$ then transitions to $\overrightarrow{S_1}$, introducing

the $|V|$ X copies. Since $|V|$ V_C copies were preserved, all v_C and X copies are deleted by verification reactions to reach the final configuration $\overrightarrow{0} = \overrightarrow{B}$.

Reverse Direction. Assume there exists a sequence of applicable rules in $\mathscr{C}_{\mathscr{S}}$ that reaches $\overrightarrow{B}$ from $\overrightarrow{A}$. First, removing all v species can only be accomplished by applying a sequence of assignment rules. The resulting configuration $\overrightarrow{S}'_0$ is $|V|$ v_C copies. Assume no edge rule can be applied in $\overrightarrow{S}'_0$. By the construction of $\mathscr{C}_{\mathscr{S}}$, this implies that the matching color assignment in G also does not have adjacent vertices sharing colors. We then add $|V|$ X copies at the second step, resulting in the deletion of all v_C and X copies in the system by the verification rules, resulting in a final terminal configuration of $\overrightarrow{0}$. If an edge rule was applied in $\overrightarrow{S}'_0$, at least two v_C copies were removed, causing the final count of X to be greater than 0. Note that applying an edge rule *before* an assignment rule also guarantees $\overrightarrow{0}$ cannot be reached, as either 1) the assignment rules for the affected v_C copies are then applied, removing those copies and consequentially preventing some X copies from being deleted, or 2) more edge rules are applied on the affected v_C copies, which prevents all v copies from being deleted. Therefore, the only way for $\mathscr{C}_{\mathscr{S}}$ to reach $\overrightarrow{0} = \overrightarrow{B}$ is for a color assignment to exist on all vertices in G where no adjacent vertices share a color.

Theorem 2.2.5 shows reachability with void step CRN systems to be in NP. □

2.2.5.2 Bimolecular Void Rules with Catalyst: $(\mathbf{2},\mathbf{1})$. In this section, we first show that reachability for size-$(2,1)$ void rules resides in the class *NL*.

Lemma 2.2.8. *Let the* implication *graph G of a CRN* (Λ,Γ) *with size-*$(2,1)$ *void rules be the graph where each node is a species* λ_i *and each reaction* $\lambda_i + \lambda_j \to \lambda_j$ *implies a directed edge from* λ_i *to* λ_j*. A configuration* $\overrightarrow{B}$ *is reachable from* $\overrightarrow{A}$ *if and only if for each species* λ_i *there exists a path to a node* λ_r *that holds one of the following properties:*

1. $\overrightarrow{A}[r] = \overrightarrow{B}[r] > 0$,
2. $\lambda_r + \lambda_j \to \lambda_j \in \Gamma$ *where* $\overrightarrow{B}[j] \geq 1$, *or*
3. $\overrightarrow{B}[r] \geq 1$ *and* $\lambda_r + \lambda_r \to \lambda_r \in \Gamma$

Proof. We will refer to a node which satisfies one of these conditions as a *root node*. A path from species λ_i to a root node λ_r means that we can delete enough copies of λ_i to reach the target configuration. We will prove this recursively, inducing over the length of the path from λ_i to λ_r, to create a reaction sequence through this process with our base case being the end of the sequence.

For our base case, any node $\lambda_i = \lambda_r$ can reach the target amount if it satisfies a condition in the Lemma statement. In Case 1, the number of species in the starting configuration is already the target amount so the claim is trivially true. In Case 2, we may use copies of the species λ_j to delete λ_i using the leftover species in the target configuration. In Case 3, the species may delete itself to reach the target amount.

For our inductive case, assume that there exists a reachable configuration such that any species λ_k with a shortest path of length $\leq l$ to a node λ_r can be reduced to the target amount $\overrightarrow{B}[k]$. For a species λ_i with a shortest path of length $l+1$, there exists an edge to a species λ_k with length l. We can use the reaction $\lambda_i + \lambda_k \rightarrow \lambda_k$ to decrease λ_i to $\overrightarrow{B}[i]$ copies at the start of the current sequence. It remains to prove that removing these copies does not affect anything later in the sequence. If the closest node λ_r falls under case 1 or 3 then removing λ_i does not affect it. If the closest node is case 2 and $\lambda_i = \lambda_j$, the species λ_i is the one used to remove λ_r; thus, the condition $\overrightarrow{B}[j] \geq 1$ means that we leave at least a single copy in the configuration that can be used to delete λ_r.

If a node does not have a path to a root node, then it either does not have any outgoing edges, or all of its outgoing edges are part of some cycle where all nodes along each cycle have a target count of 0. As a result, if the node does not fall into case 1, there is no way of reducing the respective species to its target count without leaving some other species unsatisfied. □

Theorem 2.2.9. *Reachability for basic CRNs with size (2, 1) void rules is in NL.*

Proof. We will show that we can decide whether a node has no path to a root in log space; thus reachability is in coNL = NL. We non-deterministically check a species λ_i, then for every node λ_j reachable from λ_i in the implication graph, we check if λ_j satisfies any of the conditions in Lemma 2.2.8. If we do not find such a node λ_j, then we reject.

If any node does not have a path to a root node, then some branch of this algorithm will reject. Checking each path can be done in NL as this is a directed graph. Checking if a node is a root node can be done in log space as it only involves edge queries and queries to the target configuration. □

We now show that adding an extra step turns the problem NP-complete, as with $(2,0)$ CRN systems. Here, we reduce from the classic 3SAT problem. Given an instance of 3SAT $\langle\Phi\rangle$, we construct a $(2,1)$ 2-step CRN $\mathscr{C}_{\mathscr{S}}$ as follows.

Species. For each variable x_i, we create a pair of species T_i and F_i, which represents assigning x_i a value of true or false, respectively. Additionally, for each clause c_j, we create the species C_j. The presence of a copy of C_j in a configuration of $\mathscr{C}_{\mathscr{S}}$ indicates c_j has yet to be satisfied by one of its assigned variables. Finally, we create the species X for "clean-up" procedures.

Steps and Rules. In the first step ($\overrightarrow{S}_0$), for each variable x_i, we introduce one copy of T_i and F_i. We also create a pair of *assignment* rules, one to represent assigning true to x_i ($T_i + F_i \to T_i$) and one for assigning false ($T_i + F_i \to F_i$). One of two assignment reactions will be applied to the copies of T_i and F_i; the non-deleted copy represents assigning x_i the corresponding boolean value.

In the second step ($\overrightarrow{S}_1$), we add a single copy of a clause species C_j for each clause c_j and a single copy of X. Additionally, given a clause c_j and a variable of the clause $x_k \in (x_a, x_b, x_c)$, 3 separate *verification* rules created of the form $C_j + T_k \to T_k$ (for non-negated variables) or $C_j + F_k \to F_k$ (for negated variables). If T_k/F_k is still present in $\mathscr{C}_{\mathscr{S}}$, then C_j will be consumed by that species. Finally, for each variable x_i, we create the *cleaning* rules $X + T_i \to X$ and $X + F_i \to X$ to consume all present copies of T_i and F_i. Any C_j species remaining in $\mathscr{C}_{\mathscr{S}}$ after the application of the cleaning rules indicates that it could not be deleted by a verification rule.

Theorem 2.2.10. *Reachability for 2-step CRNs with only rules of size (2, 1) is NP-complete, even for unary encoded species counts.*

Proof. We reduce from 3SAT. Given an instance of 3SAT $\langle\Phi\rangle$, we convert Φ into a 2-step $(2,1)$ CRN $\mathscr{C}_{\mathscr{S}}$ via the reduction from above. Let $\overrightarrow{A} = \overrightarrow{S}_0$ and $\overrightarrow{B}$ be a single copy of X ($\vec{X}$).

Forward Direction. Assume there exists an assignment of variables that satisfies Φ to true. A sequence of assignment reactions can be then performed in $\overrightarrow{S}_0$ that results in a configuration with only one T_i/F_i copy for each variable that matches the variable assignment. In the second step, by the construction of $\mathscr{C}_{\mathscr{S}}$, since the assignment satisfies Φ, each introduced copy of C_j can be deleted with a verification reaction. Finally, the added X copy deletes all remaining literal species. The final configuration of $\mathscr{C}_{\mathscr{S}}$ is then $\vec{X}$.

Reverse Direction. Assume there exists a sequence of applicable rules in $\mathscr{C}_{\mathscr{S}}$ that reaches $\overrightarrow{B}$ from $\overrightarrow{A}$. First, a sequence of assignment reactions can be performed in $\overrightarrow{S}_0$ that consumes one of the T_i and F_i copy for each variable. We then add one copy of each C_j species and one copy of X in the second step. Assume all C_j copies can be consumed by a verification reaction. By the construction of $\mathscr{C}_{\mathscr{S}}$, this implies that there exists an assignment of variables in Φ that evaluates the formula to true. The X copy can delete the remaining X_i/F_i copies with cleanup rules to reach the final configuration $\vec{X}$. If a C_j copy could not be removed, this implies that the variable assignment couldn't satisfy the corresponding clause. Therefore, the only way for $\mathscr{C}_{\mathscr{S}}$ to reach $\vec{X} = \overrightarrow{B}$ is for an assignment of variables in Φ to exist that satisfies all clauses of the formula.

Theorem 2.2.5 shows reachability with void step CRN systems to be in NP. □

2.2.6 Bimolecular Rules of Mixed Type: (2,0) and (2,1)

In this section, we show that the reachability problem for general single-step bimolecular void rules systems that include a mix of non-catalytic $(2,0)$ rules and catalytic $(2,1)$ rules is in P. We show this via a reduction to the perfect b-matching problem in an undirected graph.

Definition 2.2.6 (Perfect b-matching Problem)**.** *Given a graph $G = (V,E)$, $u : e \in E \to \mathbb{N} \cup \{\infty\}$ to be edge capacities, and $b : v \to \mathbb{N}$ to be the number of matchings a vertex can take, does there exists an assignment to the edges $f : e \to \mathbb{N}$ such that $f(e) \leq u(e)$ and $\sum_{e \in \delta(v)} f(e) = b(v)$ for all $v \in V$?*

Unlike reducing from reachability to perfect b-matching with only $(2,0)$ rules in [3], the inclusion of catalyst rules requires a substantially more involved reduction. We therefore begin

with a brief overview of our reduction and then describe each step in greater detail. Finally, we follow with a proof of correctness and a runtime analysis for the entire result.

2.2.6.1 Overview. Our reduction transforms an instance of CRN reachability $\langle \mathscr{C}, \overrightarrow{A}, \overrightarrow{B} \rangle$ into an instance of the perfect b-matching problem. The key idea is to identify which species involved in catalytic $(2,1)$ rules can be fully deleted using just catalytic rules and which must be further deleted using non-catalytic $(2,0)$ rules. If this distinction were known in advance, it would be straightforward to construct a corresponding graph and solve reachability via a matching instance.

To infer this structure, we first construct a directed graph T, which we refer to as the *catalytic deletion graph*, where each vertex corresponds to a species λ_i and each directed edge (λ_i, λ_j) represents a $(2,1)$ rule $\lambda_i + \lambda_j \to \lambda_j$ that catalytically deletes λ_i using λ_j. We then compute the strongly connected components (SCC's) of T and build a condensation graph H whose nodes each represent a component of T. The structure of H allows us to identify which catalytic species *must* be completely deleted via a non-catalytic $(2,0)$ rule.

Using this information, we construct an undirected graph G whose nodes effectively represent species that must be deleted via $(2,0)$ or $(2,1)$ rules and whose edges represent those corresponding rules. We then formulate a perfect b-matching instance on G where a perfect matching exists exactly when there exists a valid sequence of $(2,0)$ and $(2,1)$ rules that complements the catalytic deletions to reach the target configuration $\overrightarrow{B}$.

2.2.6.2 Creating Catalytic Deletion Graph T. Given a CRN $\mathscr{C} = (\Lambda, \Gamma)$, we construct the directed graph $T = (V, E)$ as follows. For each catalyst void rule $\lambda_i + \lambda_j \to \lambda_j \in \Gamma$, if $\overrightarrow{A}[i], \overrightarrow{A}[j] > 0$, we create vertices λ_i and λ_j and the directed edge (λ_i, λ_j).

The intuition of T is that an edge from λ_i to λ_j represents the species λ_i being deleted by the catalyst species λ_j. It then follows that a species whose respective vertex in T has an out-degree of 0 can only be deleted by a $(2,0)$ rule. We label these vertices as *mandatory vertices*. We also consider cycles in T in which each vertex (species) only has an out-going edge to another vertex in the cycle. If the count of all represented species in the cycle in $\overrightarrow{B}$ is zero, then regardless of the

application of rules corresponding to the edges of the cycle, there is guaranteed to be at least one remaining species left that can only be completely removed by a $(2,0)$ rule. We label these cycles as *mandatory cycles*.

Definition 2.2.7 (Mandatory Vertices)**.** *A vertex in T with an out-degree of* 0*.*

Definition 2.2.8 (Mandatory Cycles)**.** *A cycle in T in which each vertex 1) only has an out-going edge to another vertex in the cycle, and 2) has a corresponding species count of* 0 *in* $\overrightarrow{B}$*.*

2.2.6.3 Creating SCC Condensation Graph H. Given a directed graph $T = (V,E)$, we construct the directed graph $H = (V',E')$ as follows. First, we run Tarjan's Strongly Connected Component Algorithm on T, which returns a partition of T's vertices of strongly connected components $C = \{c_1, c_2, \cdots, c_n\}$ [88]. For each component $c_i \in C$, we create the vertex c_i. For each directed edge from c_i to another component c_j, we create the directed edge (c_i, c_j).

Observation 2.2.11. *A vertex in H represents a mandatory vertex if it is not a condensed component of T and it has an out-degree of 0.*

Observation 2.2.12. *A vertex in H represents a mandatory cycle if it is a condensed component of T in which all corresponding species have final counts of 0, and it has an out-degree of 0.*

2.2.6.4 Creating b-matching Instance Graph G. Given a CRN $\mathscr{C} = (\Lambda, \Gamma)$, configurations $\overrightarrow{A}$ and $\overrightarrow{B}$, and directed graphs $T = (V,E)$ and $H = (V',E')$, we create an instance of the perfect b-matching problem with the graph $G = (V'',E'')$ as follows. Let the *difference configuration* $\overrightarrow{D} = \overrightarrow{A} - \overrightarrow{B}$.

Creating V'' and $b(\cdot)$. For each species $\lambda_i \in \Lambda$, if $\overrightarrow{D}[i] > 0$, we create the vertices λ_{i1} and λ_{i2} and set both $b(\lambda_{i1})$ and $b(\lambda_{i2})$ to $\overrightarrow{D}[i]$. These vertices represent the number of copies of λ_i that must be removed from $\overrightarrow{A}$ by the void rules. Additionally, if $\overrightarrow{B}[i] > 0$, we create the vertices $\overline{s_{i1}}$ and $\overline{s_{i2}}$ and set both $b(\overline{s_{i1}})$ and $b(\overline{s_{i2}})$ to $\overrightarrow{B}[i]$. These vertices exist just to "set aside" the final configuration for matchings, hence the bar labels.

We now consider species that can be deleted by catalyst void rules. For each catalytic rule $\lambda_i + \lambda_j \to \lambda_j \in \Gamma$, if $\overrightarrow{A}[i], \overrightarrow{A}[j] > 0$ *and* its corresponding edge in T is not part of a cycle, we create

the vertices λ_{i1}' and λ_{i2}', if not already created, and set both $b(v_{i,1}')$ and $b(v_{i,2}')$ to $\overrightarrow{D}[i]$. Additionally, given a vertex of H $c_i \in V'$, if c_i represents a condensed cycle $\{\lambda_1, \cdots, \lambda_n\}$, we create the vertices c_{i1}' and c_{i2}'. If the cycle is mandatory, we assign $b(c_{i1}')$ and $b(c_{i2}')$ the value $(\sum_{\lambda_i \in c_i} \overrightarrow{D}[i]) - 1$; else they are assigned $(\sum_{\lambda_i'' \in c_i} \overrightarrow{D}[i])$, where λ_i'' is a *non-mandatory* vertex of c_i. These vertices represent a choice to delete a species λ_i using a catalytic species.

Let the vertices with b-values from $\overrightarrow{D}$ be the sub-graph G_D, and the vertices with b-values from $\overrightarrow{B}$ be the sub-graph G_B.

Creating E'' and $u(\cdot)$. For each $(2,0)$ rule $\lambda_i + \lambda_j \to \emptyset \in \Gamma$, if the vertices for both species were created in G, we create the edges $(\lambda_{i1}, \lambda_{j1})$ and $(\lambda_{i2}, \lambda_{j2})$. Performing a matching on these edges corresponds to deleting λ_i and λ_j by a $(2,0)$ rule.

For each $(2,1)$ rule $\lambda_i + \lambda_j \to \lambda_j \in \Gamma$, if λ_{i1}' and λ_{i1}' were created in G and the rule is not part of a cycle in T, we create the edges $(\lambda_{i1}, \lambda_{i1}')$ and $(\lambda_{i2}, \lambda_{i2}')$. For each vertex of T that represents a condensed cycle $c_i = \{\lambda_1, \cdots, \lambda_n\}$, if the cycle is mandatory, we create the edges (λ_{k1}, c_{i1}') and (λ_{k2}, c_{i2}') for all $s_k \in c_i$. Otherwise, we only create (λ_{k1}, c_{i1}') and (λ_{k2}, c_{i2}') for the *non-mandatory* vertices of c_i. Matching the edges represents deleting a species λ_i with a catalyst rule.

We finally create the following edges: for all $\bar{s}_{i1}$ and $\bar{s}_{i2}$ vertices, create the edge $(\bar{s}_{i1}, \bar{s}_{i2})$, for all λ_{i1}' and λ_{i2}' vertices, create the edge $(\lambda_{i1}', \lambda_{i2}')$, and for all c_{i1}' and c_{i2}' vertices, create the edge (c_{i1}', c_{i2}'). Matching on these edges does not represent a rule application, but rather ensures a perfect b-matching can be performed on these vertices even if they were not perfectly matched by other (2,1) edges. For all edges $e \in E''$, assign $u(e) = \infty$.

2.2.6.5 Result. The overall effect is that G has a perfect b-matching exactly when configuration $\overrightarrow{B}$ is reachable from configuration $\overrightarrow{A}$ which yields our main theorem from this section.

Theorem 2.2.13. *Reachability in void rule systems with (2,0) and (2,1) rules is solvable in $O(|\Lambda|^2 \log(|\Lambda|)(|\Gamma| + |\Lambda| \log(|\Lambda|)))$ time.*

2.2.7 Larger Void Rules

Our next results involve CRNs with reactions that require more than two reactants. If a system's rules have all but one reactant serving as catalysts (i.e., $(k,k-1)$ void rules), then reachability remains polynomial-time solvable. In contrast, reachability for systems with any other form of void rule (with 3 or more reactants) becomes NP-complete.

2.2.7.1 Mostly-Catalytic Large Void Rules of Mixed-type: $(\mathbf{k},\mathbf{k-1})^{+}$. We provide a polynomial-time dynamic programming algorithm to decide reachability for CRNs that use mostly-catalytic void rules of the form $(k,k-1)$. We further argue that reachability remains in P, even for CRNs that use a combination of various size $(k,k-1)$. For simplicity, we refer to void rules of sizes $(k_1,k_1-1),\ldots,(k_b,k_b-1)$, where all $k_i \in \mathbb{N}$, as $(k,k-1)^{+}$, meaning there is one or more rule of this type.

Lemma 2.2.14. *Reachability for basic CRNs with only void rules of size $(k,k-1)$ requires at most $|\Lambda|$ distinct rules.*

Proof. For simplicity, let $n = |\Lambda|$. Assume there exists a sequence of reactions $a_1 r_1, \ldots, a_{n+1} r_{n+1}$ for a CRN $\mathscr{C}$ with set of species Λ and set of rules Γ that takes some initial configuration $\overrightarrow{A}$ to configuration $\overrightarrow{B}$, where $a_1,\ldots,a_{n+1}$ are positive integers (denoting how many times to apply each rule) and $r_1,\ldots,r_{n+1}$ are rules in Γ. There must then exist some species s that gets consumed by 2 rules r_i and r_j, where $i < j$. Let s^i, s^x and s^f denote the initial, intermediate and final counts of species s, where r_i reduces s from s^i to s^x and r_j reduces s from s^x to s^f. Since $s^i > s^x$, any rule r_l, where $l > i$, that uses s with count s^x can also use s with count s^i. Thus, rule r_i is not needed. □

Theorem 2.2.15. *Reachability for basic CRNs with only void rules of size $(k,k-1)$ is solvable in $O(|\Lambda|^2|\Gamma|)$.*

Proof. Let Γ denote the set of rules. We can use a dynamic programming approach to solve the problem. Construct an $|\Lambda| \times (|\Lambda|+1)$ table $D(s,j)$ of boolean entries, where each row represents a different species. Reduce the count of each species to $max(k,s^f)$, where s^f represents the final

count of species s, if there exists a rule that can do so. Starting from the first column $j = 0$, place a 1 if the respective species is already in its final count. Then, for each entry $D(s, j)$, place a 1 if $D(s, j-1)$ is a 1 or if there exists a reaction $\gamma \in \Gamma$ that reduces s to its final count, where all the reactants of γ have either reached their final counts or will not prevent the reaction from occurring once they do. If column $|\Lambda| + 1$ contains all 1's, then reachability is possible. Otherwise, it is not.

By Lemma 2.2.14, a solution to the problem requires at most $|\Lambda|$ unique reactions, where each reaction directly reduces each species from its initial counts to its final counts. Thus, finding a solution to the problem takes at most $|\Lambda|$ steps since we are implicitly selecting at least one reaction per column. This results in $|\Lambda| + 1$ columns, with the first column representing the initial configuration.

Since every rule is a catalytic void rule, there must exist an ordering of reactions such that some reactions can occur first without impeding other species from getting reduced to their final counts. A bottom-up approach can be used to find this ordering, starting with the reactions that can be put off until later and working up to the reactions that must occur first. In table D, this ordering is implicitly represented between columns, with the reactions between the rightmost columns being the ones we do first. Any species with final count greater than k is reduced before the algorithm is run if there exists an applicable rule, as reducing this species count does not prevent any other rule from occurring. Filling D takes at most $O(|\Lambda|^2|\Gamma|)$ steps. Hence, reachability is solvable in polynomial time. □

Theorem 2.2.16. *Reachability for basic CRNs with only void rules of size $(k, k-1)^+$ is solvable in $O(|\Lambda|^2|\Gamma|)$.*

Proof. This follows from Theorem 2.2.15. Since we are considering a rule set Γ where the size of each rule $\gamma \in \Gamma$ is $\leq k$, reducing each initial species count to $\max(k, s^f)$ guarantees that any r that needs s will be able to occur. The algorithm remains the same. □

Corollary 2.2.17. *Reachability for 2-step CRNs with only void rules of size* $(k, k-1)^+$ *is NP-complete, even for unary encoded species counts.*

Proof. Follows from Theorem 2.2.10. □

2.2.7.2 Large Void Rules of Uniform-type: $(\mathbf{k \geq 3, g \leq k-2})$**.** Here we show that reachability for CRNs using any void rules with at least three reactants, and which remove at least two species, becomes NP-complete. To achieve this result, we show that reachability is NP-complete for CRNs using only $(3,1)$ void rules via a reduction from the Hamiltonian path problem for directed graphs. Since it was previously shown that reachability is NP-complete for CRNs using only $(3,0)$ rules [3], this implies Corollary 2.2.19.

We reduce from the Hamiltonian path problem in directed graphs. For simplicity, we reduce from the variation where each vertex has max in-degree and out-degree of two, which is still NP-complete [74]. Given an instance of the problem $\langle G, s, t \rangle$, we outline the species and the reactions for the reduction, and show correctness in the proof.

Species. We create a species for every path through a vertex (unless it starts at t or goes back to s). For example, looking at Figure 2.3, for vertex b, we create two species $S_{a,b,c}$ and $S_{a,b,t}$, where the notation denotes the vertex it came from, the vertex itself, and the vertex it goes to. With a maximum in/out degree of two, for each vertex, there are at most 4 paths through the vertex. In general, for each vertex $j \in V$, we create the species $S_{i,j,k}$ where $(i,j),(j,k) \in E$ and $k \neq s$, $i \neq t$. For s, t, we just have S_s and S_t. We also create a species P_i for each $i \in V$ where $i \neq s$. Thus, we have $n-1$ of these species.

Reactions. There are two main types of reactions: those that pick which path through a vertex and those that walk the Hamiltonian path.

- Since there are at most 4 species per vertex, we will create rules that create a tournament to choose one of the species for each vertex. Assuming 3 species for ease of explanation, we create the rules $S_1 + S_2 + S_3 \to S_1$, $S_1 + S_2 + S_3 \to S_2$, and $S_1 + S_2 + S_3 \to S_3$. With only two species, we can create a dummy species that is not used as a catalyst. With 4

species, we augment the case of 3 species with another reaction that must occur with a dummy species (S_d) and the previous choice. For each choice S_i with $i \in \{1,2,3\}$, we create the rules $S_i + S_4 + S_d \to S_i$ and $S_i + S_4 + S_d \to S_4$.

- We create rules that walk the Hamiltonian path with valid connecting species and a 'fuel' species (P_i) so that we can only visit a vertex one time. We create each of the rules $S_{i,j,k} + S_{j,k,l} + P_k \to S_{j,k,l}$ where $i,j,k,l \in V$ and $i \neq t, l \neq s$. This rule indicates a walk from j to k along a valid edge and removes the P_k token to mark the vertex as visited.

Example. For Figure 2.3, we give the full reduction as follows. The final configuration $\vec{S}_t$ (just a single copy of S_t) is only reachable if there is a Hamiltonian path.

- The main species are S_s, $S_{s,a,b}$, $S_{s,a,c}$, $S_{a,b,c}$, $S_{a,b,t}$, $S_{a,c,d}$, $S_{b,c,d}$, $S_{c,d,t}$, S_t, P_a, P_b, P_c, P_d, and P_t. We also create a single copy each of dummy species D_a, D_b, D_c.
- The tournament reactions for each vertex are (d has only one path)
  - a) $S_{s,a,b} + S_{s,a,c} + D_a \to S_{s,a,b}$ and $S_{s,a,b} + S_{s,a,c} + D_a \to S_{s,a,c}$,
  - b) $S_{a,b,c} + S_{a,b,t} + D_b \to S_{a,b,c}$ and $S_{a,b,c} + S_{a,b,t} + D_b \to S_{a,b,t}$,
  - c) $S_{a,c,d} + S_{b,c,d} + D_c \to S_{a,c,d}$ and $S_{a,c,d} + S_{b,c,d} + D_c \to S_{b,c,d}$.
- The walking reactions for each vertex are
  - a) $S_s + S_{s,a,b} + P_a \to S_{s,a,b}$ and $S_s + S_{s,a,c} + P_a \to S_{s,a,c}$,
  - b) $S_{s,a,b} + S_{a,b,c} + P_b \to S_{a,b,c}$ and $S_{s,a,b} + S_{a,b,t} + P_b \to S_{a,b,t}$,
  - c) $S_{s,a,c} + S_{a,c,d} + P_c \to S_{a,c,d}$ and $S_{a,b,c} + S_{b,c,d} + P_c \to S_{b,c,d}$,
  - d) $S_{a,c,d} + S_{c,d,t} + P_d \to S_{c,d,t}$ and $S_{b,c,d} + S_{c,d,t} + P_d \to S_{c,d,t}$,
  - t) $S_{a,b,t} + S_t + P_t \to S_t$ and $S_{c,d,t} + S_t + P_t \to S_t$.

Theorem 2.2.18. *Reachability for CRNs with only void rules of size* $(3,1)$ *is NP-complete, even for unary encoded species counts.*

Proof. We reduce from the directed Hamiltonian path problem. Given an instance $\langle G,s,t\rangle$, we convert this to an instance of the reachability problem, as outlined above, for CRN $\mathscr{C}$ and target configuration $\vec{S}_t$. We show that H is true iff the configuration $\vec{S}_t$ is reachable in $\mathscr{C}$.

Forward Direction. Assume there exists a Hamiltonian path in G from s to t. Then it is possible that the tournament for every vertex correctly produces the species that represents the Hamiltonian path through the vertex. If this does occur, all species have been removed from the system except $|V|$ S species for the path and $|V|-1$ P species. Then, the walking reactions can occur successively by destroying the previous path vertex and the 'fuel' species, which will only leave one copy of S_t.

Reverse Direction. Assume there exists a sequence of applicable rules in $\mathscr{C}_{\mathscr{S}}$ that reaches $\overrightarrow{B}$ from $\overrightarrow{A}$. The only way to remove an S species is through the tournament and the walking reactions. The tournament will always leave at least one S for each vertex, meaning the walking reactions must be used to delete the other $|V|-1$. Along with this, the P vertices ensure that each vertex can only be visited once. Thus, the walk can not occur before the tournament and take multiple paths to reach a vertex. Thus, reaching a configuration with only S_t ensures that a walk through the graph occurred starting at S_s, ending at S_t, and that every vertex was visited.

All void CRN systems are in NP with the certificate being the sequence of rules to apply, and the number of times to apply them [3]. □

Corollary 2.2.19. *Reachability for CRNs with only void rules of size* $(k \geq 3, g \leq k-2)$ *(*$k,g \in \mathbb{N}$*) is NP-complete, even for unary encoded species counts.*

Proof. For $g \geq 1$, this follows from Theorem 2.2.18. For $g = 0$, this follows from the fact that reachability for $(3,0)$ rules is NP-complete [3]. □

2.2.8 Primary Results

We restate the primary results formally with the corresponding proofs. Although not discussed, for completeness, we also include the following lemma.

Lemma 2.2.20. *Reachability for step CRNs (including basic CRNs) uniformly using size* $(1,0)$ *void rules is in P.*

Proof. Simply decrease each species to the desired count. Since each step must become terminal, all species in rules will be removed before the subsequent step. Thus, there must exist a step that adds counts greater than or equal to the target counts. Then treat that step as a basic CRN. If no such step exists, the target configuration is not reachable. □

The collection of results presented, as a whole, yields the following main theorems of this work that characterize void rules within CRNs and step CRNs.

Theorem 2.2.1. *Reachability for basic CRNs uniformly using size* $(k \geq 3, g \leq k-2)$ *void rules is NP-complete, and is in P when uniformly using any other size void rule.*

Proof. This follows from [3], Theorems 2.2.9, 2.2.15, Corollary 2.2.19, and Lemma 2.2.20. □

Theorem 2.2.2. *Reachability for 2-step CRNs with only void rules that use exactly one reactant is in P, and is NP-complete otherwise.*

Proof. This follows from [3], Theorems 2.2.7, 2.2.10, Corollaries 2.2.17, 2.2.19, and Lemma 2.2.20. □

Theorem 2.2.3. *Reachability for basic CRNs that use a combination of void rule types is in P for combinations of (2,0) + (2,1) rules as well as* $(k,k-1)^+$ *rules, and is NP-complete for any combination that uses* $(k \geq 3, g \leq k-2)$ *rules.*

Proof. This follows from [3], Theorems 2.2.13, 2.2.16, and 2.2.19. □

2.2.9 Conclusion and Future Work

This paper presents a nearly complete classification of the computational complexity of reachability for CRNs and step CRNs that consist of deletion-only rules. We provide polynomial-time algorithms for most combinations of void rules in basic CRNs and show NP-completeness for rules of size greater than $(k, g \leq k-2)$ for $k \geq 3$. Additionally, we prove that with the addition of a single step, these problems become NP-complete. We include some natural open directions to explore:

- **Mixed-size Void Rule Systems.** What is the complexity of reachability when you consider void rules of size $(2,0)$ and $(k,k-1)$ together? This combination of void rule types is the missing piece that would complete the picture for the entire complexity landscape of void rule reachability.

- **Staged CRNs.** The step CRN model augments the basic CRN model with steps that add species once reactions are completed. A generalization of this model could have multiple "stages", where CRNs are left to react and the results of these stages are combined. How do stages affect reachability?

- **Model Variants.** What is the complexity of reachability in deletion-only extensions of CRNs, petri-nets, and vector addition systems?

2.3 Surface Chemical Reaction Networks

2.3.1 Overview

This work was in collaboration with the following authors: Robert M. Alaniz, Michael Coulombe, Erik D. Demaine, Bin Fu, Timothy Gomez, Elise Grizzell, Andrew Rodriguez, Robert Schweller, and Tim Wylie. My main contribution to this paper is the 1-burnout result for $1 \times n$ surfaces and for planar/general surfaces.

2.3.2 Introduction

A prominent area of research in molecular computation, Chemical Reaction Networks (CRNs), study well-mixed solutions of molecules. Limited by the inherent lack of geometry, the model has important restrictions on its computational power, including no proven capability of error-free computation of logarithm [26] or Turing universality [83]. Specifically, CRNs are capable of computing all semilinear functions [25]. The introduction of a surface and, by extension, geometry, with abstract Surface Chemical Reaction Networks (sCRNs) removes these limitations, and thus has increased computational power. Molecular computing on a surface is an increasingly popular direction in both experimental [24, 91] and theoretical [34, 65] research.

In this paper, we explore a restricted version of the powerful surface CRN model, where each molecule in the system can only change in a reaction a set number of times. We refer to this constraint as *burnout*. Bounding the number of state changes leads to polynomial-time and XP algorithms for many reconfiguration problems that are otherwise PSPACE-complete.

Motivations for the study of problems with burnout include examples such as optimizing limited lifetime biomolecules or modeling redox reactions in which the electron transfer from one chemical species to another increases the cost of further reaction beyond what any other current or future neighbors could afford.

2.3.2.1 Previous Work. Surface Chemical Reaction Networks (sCRNs) were introduced in [75] with a simulator provided in [27]. These papers show various constructions such as Boolean circuits and a Cellular Automata simulation.

Another restricted version of sCRNs uses only swap reactions, in which the two species only change position, Example: $A + B \rightarrow B + A$. In [15], the authors show swap reactions are capable of feed-forward computation and provide an analysis of thermodynamic properties of the circuit. Recently, [1] showed that reconfiguration is PSPACE-complete for swap surface CRNs with only 4 species and 3 reactions, and in P with any system of fewer species or reactions. This work also introduces k-burnout surface CRNs and show two important results: that 1-reconfiguration

(whether a single cell can change) is NP-complete with 1-burnout and that general reconfiguration is NP-complete with 2-burnout. Burnout is similar to the freezing concept from Cellular Automata [41, 42, 90] and Tile Automata [22], but while freezing is defined as having an ordering on states or a tile never revisiting a state, burnout is a constraint where a cell never reacts more than a fixed number of times. Thus, returning to a previous state is possible, unlike the freezing restrictions.

1D Cellular Automata are capable of Turing computation from [28]. P-completeness of prediction, is this cell in state at time step less than t, for Cellular Automata Rule 110 was shown in [67], implying it is also capable of efficient computation. This problem is also P-complete for a number of Freezing CAs in 2D, while it is always in NL for Freezing 1D CAs [42]. This work also gives a 1D freezing CA, which is Turing universal.

2.3.2.2 Our Contributions. This work investigates the reconfiguration problem for linear surface CRNs with k-burnout. Our results are outlined in Table 2.2. We begin in Section 2.3.4, where we present a polynomial-time algorithm for 1D 1-burnout. We then increase the burnout number to investigate 1D 2-burnout systems and prove that this is still in P. Following this, we show that for the case of any fixed $k = \mathcal{O}(1)$, there exists an algorithm that has a polynomial runtime. In the terms of parameterized complexity classes, this is the class XP, also known as slice-wise polynomial [30]. We then present an NP-completeness proof for when the burnout is a unary input. This result contrasts PSPACE-completeness known when the burnout is unbounded or exponentially high [75].

After $1 \times n$ lines, we begin investigating 1-burnout in 2D systems in Section 2.3.6. We start with the problem of reconfiguration, where we only have non-catalytic rules. We then show that on an arbitrary graph and with all types of rules, the reconfiguration problem is NP-complete. Finally, we study the problem of 1-reconfiguration for bounded-height surfaces, presenting an XP algorithm parameterized by height.

2.3.3 Preliminaries

A brief overview of the model and relevant problems.

2.3.3.1 Surface, Cells and Species. A *surface* for a CRN Γ is an undirected graph G of large size n. The vertices of the surface are also referred to as *cells*. Many of our results deal with $1 \times n$ grid graphs, or *linear* surfaces.

The state of a vertex is representative of a molecular *species* in the system. Chemical *reactions* considered here are bimolecular, as in they occur between two species in neighboring vertices. A rule denoting that neighboring species A and B may react to become C and D is written as $A + B \to C + D$. This is a *non-catalytic* rule, as both species change. In a *catalytic* reaction, only one of the two species will change, e.g. $C + D \to C + B$, the other used as a catalyst.

A *surface Chemical Reaction Network (sCRN)* consists of a surface, a set of molecular species S, and a set of reaction rules R. A *configuration* is a mapping from each vertex to a species from the set S.

2.3.3.2 Reachable Configurations. For two configurations I, T, we write $I \to_\Gamma^1 T$ if there exists a $r \in R$ such that performing reaction r on a pair of species in I yields the configuration T. Let $I \to_\Gamma T$ be the transitive closure of $I \to_\Gamma^1 T$, including loops from each configuration to itself. Let $\Pi(\Gamma, I)$ be the set of all configurations T where $I \to_\Gamma T$ is true.

2.3.3.3 Burnout. A limit on the number of changes that can occur in any vertex v_i. In systems that allow catalytic reactions, after this limit has been reached, while v_i will not change again, neighboring species may still use the species in that cell as a catalyst.

2.3.3.4 Reconfiguration Problem. Given an sCRN Γ and two configurations I and T, is $T \in \Pi(\Gamma, I)$?

2.3.3.5 1-Reconfiguration Problem. Given an sCRN Γ, configuration I, vertex v, and species s, does there exist a $T \in \Pi(\Gamma, I)$ such that T has species s at vertex v?

2.3.4 Algorithms for Constant Burnout

We show that reconfiguration of a linear surface is solvable in polynomial-time when the burnout is one or two.

2.3.4.1 1-Burnout Linear Surfaces. In the case of 1-burnout with a $1 \times n$ line, the problem of reconfiguration is solvable in linear time with respect to n and the size of the rule set. As an observation, there are at most six reactions for any vertex, v_i, on a linear surface since a vertex has at most two neighbors. These reactions include the following:

- A left reaction, where vertex v_i reacts with vertex v_{i-1} and both vertices reach their final states.
- A left catalytic reaction, where vertex v_i reacts with vertex v_{i-1} in its initial state to transition vertex v_i to its final state without changing v_{i-1}.
- A left final-catalytic reaction (or left final), where vertex v_i reacts with vertex v_{i-1} in its final state to transition vertex v_i to its final state without changing v_{i-1}.
- A right reaction, where vertex v_i reacts with vertex v_{i+1} and both vertices reach their final states.
- A right catalytic reaction, where vertex v_i reacts with vertex v_{i+1} in its initial state to transition vertex v_i to its final state without changing v_{i+1}.
- A right final-catalytic reaction (or right final), where vertex v_i reacts with vertex v_{i+1} in its final state to transition vertex v_i to its final state without changing v_{i+1}.

Additionally, we also consider when a vertex is in its final state. An example system and sequence of reactions can be found in Figures 2.5 and 2.6.

We construct a $7 \times n$ table (Example in Table 2.3), where each row represents one of the possible reactions, including no reaction, and each column represents the starting configuration's vertices from left to right. For each entry in the table, we see if the reaction exists for that vertex and if the vertex reaches its final state. If both cases are satisfied, place a 1 in the corresponding

row, otherwise, place a 0. After all cells are evaluated, we construct a directed graph with edges being directed from column i to column $i+1$ with the following properties for each row entry in column i:

- In final state: edge to every row in column $i+1$ with a 1 except left reaction.
- Left final: edge to every row in column $i+1$ with a 1 except left reaction.
- Left catalytic: edge to every row in column $i+1$ with a 1 except left reaction.
- Left reaction: edge to every row in column $i+1$ with a 1 except left reaction.
- Right final: edge to every row in column $i+1$ with a 1 except left reaction and left final.
- Right catalytic: edge to every row in column $i+1$ with a 1 except left reaction and left catalytic.
- Right reaction: edge only to the row corresponding to left reaction in column $i+1$ if there is a 1.

These edges ensure that no matter which reaction is chosen for a vertex represented by column i, the reaction chosen for the column $i+1$ vertex will be able to perform its reaction either before or after the previous reaction.

Once these edges are defined for every column, the problem is then finding a path from s to t, where s is a vertex that has directed edges to each entry in column 1 and t is a vertex that can be reached from each entry in column n (see Table 2.3 and Figure 2.4 for reference). Any path represents a set of rules that can be assigned an ordering to reconfigure all vertices to their final states.

Theorem 2.3.1. *Reconfiguration in 1-burnout for* $1 \times n$ *lines is solvable in* $O(n+|R|)$ *time.*

Proof. We provide proof by induction for the previously described algorithm that solves reconfiguration in $1 \times n$ surfaces. This proof guarantees that any solution from this algorithm constitutes a set of reactions that can be reordered to successfully reconfigure a given initial configuration to its final configuration.

Base case: $n = 2$. Let v_i be the leftmost vertex. Since this vertex does not have a neighbor to its left, there are only 4 reactions we need to consider for this vertex:

1. In final state: vertex v_{i+1} must be in its final state or a left catalytic or left final reaction.
2. Right catalytic: vertex v_{i+1} must be in its final state or a left final reaction.
3. Right final: vertex v_{i+1} must be in its final state or a left catalytic reaction.
4. Right reaction: vertex v_{i+1} must be a left reaction.

If two such reactions exist for each vertex, then a path exists from s to t visiting the vertices in the table that correspond to each reaction. Otherwise, no such path would exist.

Inductive step: let $n = k$. Assume that there is a set of k reactions for vertices $v_1, \ldots, v_k$ that can be reordered to transition all k vertices to their final states. In order for the reaction chosen for vertex v_{k+1} to be valid, it must not interfere with the k^{th} reaction corresponding to vertex v_k. Consider two cases:

1. Vertex v_k is currently in its final state or reacts with its left neighbor v_{k-1}. Vertex v_{k+1} is never used, so as long as v_{k+1} does not perform a left reaction with v_k, it will not interfere with the k^{th} reaction.
2. Vertex v_k reacts with vertex v_{k+1}. Consider 3 possible reactions for v_k: (1) Right reaction: the only valid reaction for v_{k+1} is a left reaction, (2) Right catalytic: except left or left catalytic, all reactions are valid for v_{k+1}, and (3) Right final: except left or left final, all reactions are valid for v_{k+1}.

If we think of v_k as being column i and v_{k+1} as being column $i+1$, edges are defined from i to $i+1$ in a way that avoid these conflicting reactions. Any other reaction that is chosen for v_{i+1} can always perform its reaction before or after v_i performs its reaction. As a result, any path up to column $i+1$ would represent a set of reactions that can be reordered to transition these $k+1$ vertices to their final states.

Given the initial and final configurations, it takes $O(n)$ time to compare the states. Constructing the table takes $O(|R|)$ time. The path finding algorithm runs in $O(V+E) = O(n+E)$

time. However, the number of edges is a constant factor of the number of vertices, whereas $|R|$ might be exponential in n. Thus, the final runtime for the algorithm is $O(n+|R|)$. □

2.3.4.2 2-Burnout Linear Surfaces. This section considers the problem of reachability on linear surfaces with 2 burnout.

Theorem 2.3.2. *Reconfiguration of a $1 \times n$ line for surface CRNs with 2-burnout is solvable in $O(n \cdot |S|^2 \cdot |R|^4)$ time.*

Proof. Since we are considering 2-burnout, every cell can only change species twice. This is a cell starting with the initial species, possibly changing to an intermediate species, then finally changing to the target species. It is then possible to track all the possible transitions of a cell in a polynomial sized table. We define the table D with each entry $D(x,s,r_1,r_2)$ being a Boolean indicating if the cells at indices $0,1,\ldots,x$ can reach their target species using reactions r_1 and r_2 on x, and using s as intermediate species for cell x. (Note, r_1 and s may be null if the cell only reacts once to reach the target species.) The reactions are specific with which neighbor the cell reacts with, left or right. This results in $\mathcal{O}(n \cdot |S| \cdot |R|^2)$ cells of the table.

To compute each entry $D(x,s,r_1,r_2)$, we check if r_1 and r_2 are consistent with cell $x-1$. Meaning, if r_1 reacts with the left neighbor, some entry $D(x-1,s',r_1,r_3)$ or $D(x-1,s',r_3,r_1)$ for any s,r_3 must be true. If r_1 is a catalytic reaction, then the species in cell $x-1$ does not change and must be the initial species, intermediate species, or the target species.

We must also be careful with the ordering of the reactions. If r_1 or r_2 reacts with the intermediate species s' of the $(x-1)$th cell, then r_1 must be the second reaction for $x-1$. The run time to compute each cell of the table is $\mathcal{O}(|S| \cdot |R|^2)$.

If any $D(n-1,s,r_1,r_2)$ is true, then the answer to reconfiguration is true. □

2.3.4.3 Constant Burnout. In this section, we consider the problem of reconfiguration for a surface CRN with n cells with at most k-burnouts on a $1 \times n$ board.

Theorem 2.3.3. *There is an $O(n^{1+k\log h})$ time algorithm for k-burnout degree-h 1D surface CRN reconfiguration, where each species is in at most h rules.*

Proof. We have a divide-and-conquer approach in our algorithm. A brute force method is used to enumerate all the possible transitions for the median position. The problem is split into two independent problems that can be solved independently.

Let p be the position of the median in a 1D surface CRN. We enumerate all the possible ways to burn out the position p at most k times. Since each species is in at most h rules, we have at most h^k combinations about the list of transitions involved by position p. Let $T(n)$ be the running time to solve the reconfiguration problem. We have the recursion $T(n) = h^k(2T(\frac{n}{2}))$. It brings a solution with $T(n) = h^{k\log n} \cdot n = n^{1+k\log h}$. □

2.3.5 Non-constant Burnout on a Line

Here, we show that reconfiguration with k-burnout, where k is part of the input, is NP-hard. Without burnout (no bound on state changes), reconfiguration of a $1 \times n$ line is PSPACE-complete [75], but even with a burnout k given in binary, the problem may not be in the class NP since $O(kn)$ possible reactions could occur, which is exponential in $\log k$. This motivates looking at bounds on state changes that are polynomial in n and further motivates the other algorithms in the paper.

Reduction. We reduce from Vertex Cover (VC) by enumerating all vertices and using them as states on a $1 \times n$ line. A state "walks" back and forth choosing a vertex to add to the cover and crossing off instances it finds. Given a graph $G = (V,E)$ where $V = \{1,2,\ldots,n\}$ and an edge $e \in E$ is defined as $e = \{v_i, v_j\}$ for $v_i, v_j \in V$ and $i \neq j$. An edge is listed as two states: 34 meaning an edge between vertices v_3 and v_4. Between any two edges we include a spacing state $-$.

Create the line representing the graph with edges in any order: $BS_0 - e_1 - e_2 - \cdots - e_m - E$, where the B state indicates the beginning of the line, E is the end of the line, and S_0 is a special state indicating no vertices are in the vertex cover. Example: $BS_0 - 34 - 13 - 21 - 14 - E$.

Basically, each edge independently and nondeterministically picks the vertex to cover it with both possible rules. Create rules for all $v_i, v_j \in V$ as $v_i + v_j \to v_i' + x$ and $v_i + v_j \to x + v_j'$ where x is an ignored state and the prime state is the chosen vertex for that edge. The spacing states ensure edges do not affect each other. Example: $3 + 4 \to 3' + x$ and $3 + 4 \to x + 4'$.

The S counting state sweeps back and forth k times to choose a vertex to add to the cover and ignores the other states. The S state takes the first picked vertex and removes all duplicates of it while remembering the count. There is a state S_{count}^{vertex} that exists for each vertex and count up to k. Thus, the rules $S_i + v'_j \to S_{i+1}^j + x$ are added for each vertex and count up to k. Example: $S_0 + 3' \to S_1^3 + x$ is used if v_3 is the first vertex added.

Once a vertex transitions to an S^i state, it ignores everything but v'_i states. Meaning it only swaps states, or "walks" in that direction. Thus, all rules $S^i + A \to A + S^i$ is added for any state X that is not v_i, B, or E. For v_i, $S_c^i + v'_i \to S_c^i + x$.

When a S_c^j vertex is next to the B or E states, it can transition to S_c. The rules $B + S_c^j \to B + S_c$ and $E + S_c^j \to E + S_c$ exist for all vertices v_j. This means we have removed all instances of the chosen vertex and can pick a new vertex for the cover.

This requires $O(kn)$ states to handle counting for each vertex. If k is odd, the final configuration, given a k VC exists, is $B - xx - xx - xx - \cdots - S_k E$. If k is even, then the final configuration is $BS_k - xx - xx - xx - \cdots - E$. S_k can not interact with anything. This requires $k+1$ burnout.

Theorem 2.3.4. *Reconfiguration of a $1 \times n$ configuration in sCRNs with k-burnout is NP-hard, even when $k < n$, and NP-complete as long as k is polynomial in n.*

Proof. Given a VC with graph $G = (V, E)$ and $k \in \mathbb{N}$, we create a surface CRN system with configuration C and rules R as described above. We define the output configuration D based on the number of edges and parity of k as described. G has a VC of size k if and only if C can reach configuration D with burnout $k+1$. Note that $k \leq n$ as input from VC, so the number of states and rules in the reduction is polynomial.

Given that the graph G has a k vertex cover, in the sCRN system, the only transitions possible at first are for each edge to pick a vertex to cover it. Then the counting state walks across, increases the count and selects the vertex from the first edge, and that state continues walking and removes any other instance of that vertex. In the best case, all locations but the first and last have changed twice. If this continues, and it always adds the correct vertices, then after k passes only x's are left.

S_k does not interact with anything, so nothing else transitions. The k passes and the initial choice requires $k+1$ burnout.

If the sCRN system ends in the output configuration with x's on every edge state, which can only occur if the k passes chose vertices that appeared in the other edges and were crossed out. Thus, every edge correctly chose the right vertex to cover it so that only k different vertices were used. □

2.3.6 Extension to 2D Graphs

As an extension to the 1D case, we now consider reconfiguration and 1-reconfiguration for 2D surfaces. In the case of reconfiguration, we study a restricted version of the problem where all reactions are non-catalytic.

Theorem 2.3.5. *Reconfiguration in 1-burnout for a planar graph $G = (V,E)$ is solvable in $O(|V|^{1.5} + |R|)$ time if every reaction is non-catalytic.*

Proof. Given a planar graph $G = (V,E)$, construct a subgraph G' from G such that there is an edge between pairs of vertices if there exists a non-catalytic reaction that transitions both vertices to their final states. Run maximum matching on G'. If all vertices are either matched or in their final state, then reconfiguration is possible. Otherwise, reconfiguration is not possible.

Since non-catalytic reactions transition both vertices to their final states, a vertex must be involved in at most one reaction. Edges represent these non-catalytic reactions between two vertices. As a result, limiting a vertex to one reaction is the equivalent of matching each vertex in G' to at most one other vertex it shares an edge with, which is a perfect matching problem. For planar graphs, this can be solved using a maximum matching algorithm. If any unmatched vertex is not in its final state, then reconfiguration is not possible because this vertex is unable to react.

Constructing G' takes $O(V + |R|)$ time. Running the maximum matching algorithm takes $O(V^{1.5})$ time. A last check of G' for any unmatched vertices that are not in their final state takes $O(V)$ time. Therefore, the runtime is $O(V^{1.5} + |R|)$. □

Corollary 2.3.6. *Reconfiguration in 1-burnout for general graphs is solvable in $O(V^4 + |R|)$ time if every reaction is non-catalytic, where V is the number of vertices.*

Proof. Proof follows from Theorem 2.3.5. Maximum matching on general graphs runs in $O(V^4)$ time. □

2.3.6.1 Arbitrary Graphs with 1-Burnout. We now consider surface CRNs that allow catalytic as well as non-catalytic rules. With this additional rule type, we prove the problem of reconfiguration is NP-complete on an arbitrary graph with 1-burnout.

Theorem 2.3.7. *Reconfiguration with* 1*-burnout of an arbitrary surface in surface CRNs is NP-complete.*

Proof. We reduce from the dominating set problem to sCRN reconfiguration with 1-burnout. Let $G = (V, E)$ be an arbitrary graph and k be an integer parameter. We need to decide if graph G has a dominating set of size k. Note that a subset $U \subseteq V$ is a dominating set of G if each vertex $v \in V - U$ has $(u, v) \in E$ for some $u \in U$ (vertex u dominates v).

Let $v_1, \cdots, v_n$ be the n vertices of G. We design a surface CRN system. For each edge (v_i, v_j) in E, create two rules $v_i + v_j \to v_i + v'_j$ and $v_i + v_j \to v'_i + v_j$. We introduce k additional species $u_1, \cdots, u_k$. The target configuration is to let each v_i enter v'_i for $i = 1, \cdots, n$ and each u_t enter u'_t. We set up the rules $u_t + v_j \to u'_t + v'_j$ for all $t \leq k$ and all $j \leq n$.

If graph G has a dominating set of size k, the target configuration is reachable. Assume that $v_{i_1}, \cdots, v_{i_k}$ dominate all the vertices in the graph G. For each v_j with $j \in \{1, \cdots, n\} - \{i_1, \cdots, i_k\}$, it can be transformed into v'_j by a rule $v_{i_s} + v_j \to v_{i_s} + v'_j$. Each v_{i_s} can enter v'_{i_s} by a rule $u_s + v_{i_s} \to u'_s + v'_{i_s}$. Here, the burnout is 1. Similarly, if the target configuration is reachable, there is a dominating set of size k. If the target configuration is reachable, we have at most $v_{i_1}, \cdots, v_{i_h}$ with $(h \leq k)$ such that each v_{i_r} enters v'_{i_r} via the type of rule $u_t + v_{i_r} \to u'_t + v'_{i_r}$ as there is only one burnout for each v_i and u_j. Clearly, $v_{i_1}, \cdots, v_{i_h}$ dominate all the other vertices in the graph G.

This is a polynomial-time reduction and membership is known from [1]. □

2.3.6.2 1-Burnout 1-Reconfiguration. We finish by considering the problem of 1-Reconfiguration on 2 dimensional surfaces with 1-burnout.

Theorem 2.3.8. 1-*Reconfiguration in* 1-*burnout of a* $w \times n$ *rectangle for surface CRNs is solvable in* $O\left(n \cdot (|S||R|)^{2w} \cdot f(w)\right)$ *time.*

Proof. We use a dynamic programming approach similar to that in Theorem 2.3.2, defining a table D with Boolean entries $D(x, \vec{s}, \vec{r}, \pi)$, where x is a column index, $\vec{s} = [s_1, s_2, \ldots, s_w]$, $\vec{r} = [r_1, r_2, \ldots, r_w]$, and π a permutation of $[1, w]$. Each $s_y \in S$ is a potential final species of cell (x, y), which changes from its initial species into (x, y) due to reaction $r_y \in R$, and π gives the order in which the reactions occur. As before, r_y specifies which of its up-to-four neighboring cells participated in the reaction, and s_y and r_y may be null if the cell never changes species.

Since only one cell (x_t, y_t) of the target configuration is fixed, the top-level of the dynamic program will be column x_t, and it will symmetrically recurse outwards in both directions, with base-cases at both ends. So, for $x < x_t$ $D(x, \vec{s}, \vec{r}, \pi)$ is true if the cells in columns $0, 1, \ldots, x$ can reach a target configuration in which column x reaches species $\vec{s}$ using reactions $\vec{r}$ occurring in order π, for $x > x_t$ we consider columns $x, x+1, \ldots, n-1$ instead, and for $x = x_t$ we consider the entire surface.

To compute $D(x, \vec{s}, \vec{r}, \pi)$, say when $x < x_t$, we search for a smaller subproblem $D(x-1, \vec{s}', \vec{r}', \pi')$ which has value true and $(\vec{r}', \vec{r})$ together are a chain of reactions that actually transform columns $x-1$ and x into species $(\vec{s}', \vec{s})$ from their initial species, given that they must occur in relative orders π' and π. Specifically, we can search each possible interleaving of $\pi(\vec{r})$ and $\pi'(\vec{r}')$, simulate the reactions in that order, and verify that the reactions within these two columns can actually be performed and do result in $(\vec{s}', \vec{s})$. Notably, for reactions between columns $x-2$ and $x-1$, we do not need to validate the species in column $x-2$ because the smaller subproblem already performed that validation, and for reactions between columns x and $x+1$, the species in column $x+1$ are assumed to be validated later in a larger subproblem. For $x > x_t$, the recursion is symmetric.

For top-level subproblems $D(x_t, \vec{s}, \vec{r}, \pi)$, we only consider $\vec{s}$ that include the fixed target species s_{y_t}, and we search for both $D(x_t - 1, \vec{s}', \vec{r}', \pi')$ and $D(x_t + 1, \vec{s}'', \vec{r}'', \pi'')$ and validate between

all three columns $x_t - 1, x_t, x_t + 1$ in a similar manner. If any $D(x_t, \vec{s}, \vec{r}, \pi)$ is true, then the answer to 1-reconfiguration is true.

The size of D is $O(n \cdot |S|^w \cdot (4|R|)^w \cdot w!)$. Computing each entry involves checking $O(|S|^w \cdot (4|R|)^w \cdot w!)$ subproblems, and each check considers $\binom{2w}{w}$ interleavings of orderings and runs an simulation taking $O(w)$ time. Combined, the total time is $O\left(n \cdot (|S||R|)^{2w} \cdot f(w)\right)$ for a function f only depending on w. Therefore, for constant w, this is polynomial time. □

2.3.7 Conclusion

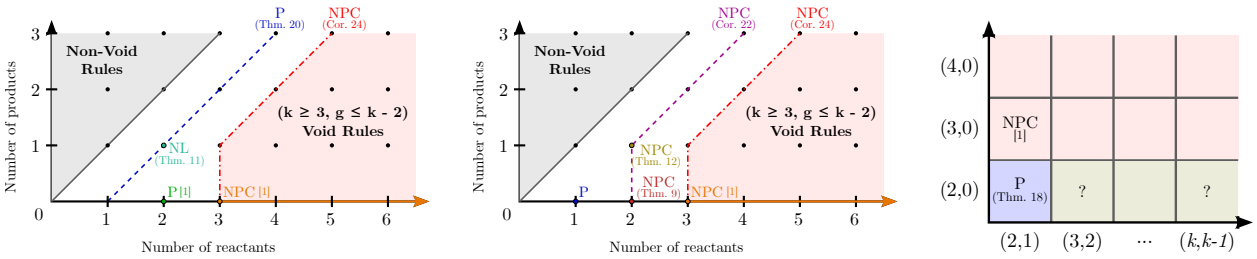
In this paper, we have shown that the reconfiguration problem on $1 \times n$ surface CRNs with k-burnout is in P when $k = 1$ or $k = 2$. To show this, we have given algorithms that output a sequence of reactions to achieve the given configuration. Further, we show that for any $k = \mathcal{O}(1)$, there exists an algorithm that has a polynomial runtime in k. To conclude our investigation of 1-Dimensional surface CRNs, we prove that when the burnout number, k, is part of the input (in unary), the problem of reconfiguration is NP-complete.

Following by exploring 2-Dimensional surface CRNs and showing that a restricted case of 1-burnout reconfiguration can be seen as perfect matching, showing this case of the problem to still be in P. Finishing with a proof that the problem of 1-Reconfiguration in 1-burnout can be solved in polynomial time on a $w \times n$ rectangle when w is constant. Some of the open questions are then:

- What is the lower bound for a given k-burnout?
- In a rectangle/grid graph, what is the lower/upper bound for k-burnout?
- Most of our complexity is in terms of the size of the surface. Are there interesting results looking at the complexity of other aspects of an sCRN such as states, rules, and burnout?
- We have a direct NP-complete reduction but does there exist an L-reduction for some inapproximability result?

Table 2.1: Summary of reachability results. The notation $(k,k-1)^+$ means one or more void rules of the form $(k,k-1)$ for $k \geq 2$. The notation $(k \geq 3, g \leq k-2)$ means void rules with at least three reactants that consume at least two species. The notation $\rightarrow$ NPC signifies that the step CRN result follows directly from the basic CRN case.

Reachability Results				
	Basic CRNs (1-step)		**2-Step CRNs**	
Void Rules	**Complexity**	**Ref.**	**Complexity**	**Ref.**
$(2,1)$	NL	Thm. 2.2.9	NPC	Thm. 2.2.10
$(k,k-1)^+$	$O(\lvert\Lambda\rvert^2\lvert\Gamma\rvert)$	Thm. 2.2.16	NPC	Cor. 2.2.17
$(2,0)$	$O(\lvert\Lambda\rvert^2\lvert\Gamma\rvert\log(\lvert\Lambda\rvert))$	[3]	NPC	Thm. 2.2.7
$(2,0)$, $(2,1)$	$O(\lvert\Lambda\rvert^2\lvert\Gamma\rvert\log(\lvert\Lambda\rvert))$	Thm. 2.2.13	NPC	Thm. 2.2.7
$(k \geq 3, g \leq k-2)$	NPC	Cor. 2.2.19	$\rightarrow$ NPC	-



(a) Void Rules in CRNs (1-step) (b) Void Rules with 2-step CRNs (c) Mixing Void Rules

Figure 2.1: Visual representations showing how the results from Table 2.1 fit together. (a) A plot depicting the complete characterization of reachability complexity for basic CRNs with uniform-type void rules. (b) A plot depicting the complete characterization of uniform-type step results with only a single additional step. (c) Void rule systems with mixed-type rules in basic CRNs.

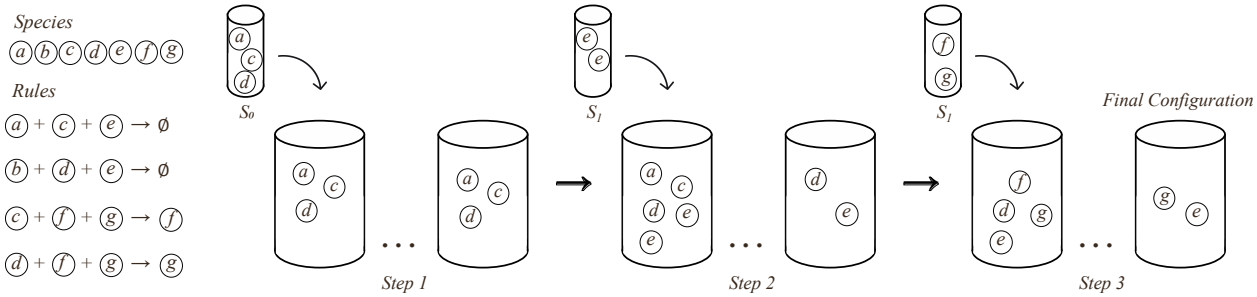


Figure 2.2: An example step CRN system. The test tubes show the species added at each step and the system with those elements added. The CRN species and void rule-set are shown on the left.

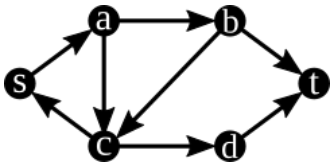


Figure 2.3: Directed graph used in the example $(3,1)$ reduction in Section 2.2.7.2.

Table 2.2: Comparison of reconfiguration results. For a CRN system, R is the set of rules and S is the set of species. V is the set of vertices for the graph defining the shape. **Non-catalytic rules only. ‡These results are for the problem of 1-Reconfiguration.

Shape	**Burnout**	**Result**	**Theorem**
$1 \times n$	1	$O(n + \|R\|)$	Thm. 2.3.1
$1 \times n$	2	$O(n \cdot \|S\|^2 \cdot \|R\|^4)$	Thm. 2.3.2
$1 \times n$	$O(1)$	P for $O(1)$ degree	Thm. 2.3.3
$1 \times n$	k (unary)	NP-complete	Thm. 2.3.4
$1 \times n$	Unbounded	PSPACE-complete	[75]
Planar	1	$O(\|V\|^{1.5} + \|R\|)$	Thm. 2.3.5**
General	1	NP-complete	Thm. 2.3.7
$m \times n$	1	NP-complete	[1]‡
$m \times n$	1	FPT in m	Thm. 2.3.8‡

Table 2.3: Turning the example system from Figure 2.5 into a table of reactions.

Reaction Type	🟡	⚪	🔵	⚪
In Final State	-	-	-	-
Left	-	1	-	-
Left Catalytic	-	-	-	1
Left Final	-	-	-	-
Right	1	-	-	-
Right Catalytic	-	-	-	-
Right Final	-	-	1	-

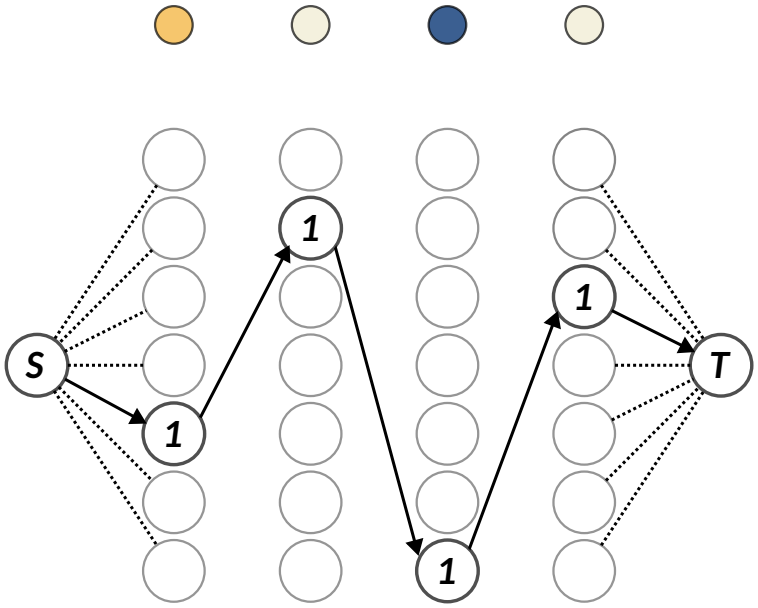
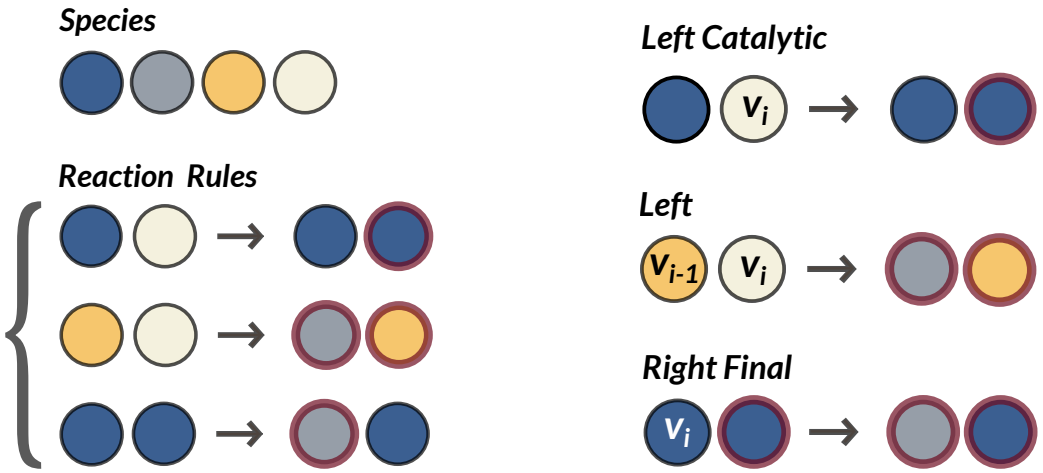


Figure 2.4: Table 2.3 as a graph.



(a) (b)

Figure 2.5: (a) An example sCRN system with 4 species, three rules, and 1 burnout. (b) Rule types used in Figure 2 example. *Note: The red ring outline shows whether the vertex has been "burned out." There is no effect on the reaction rule itself.*

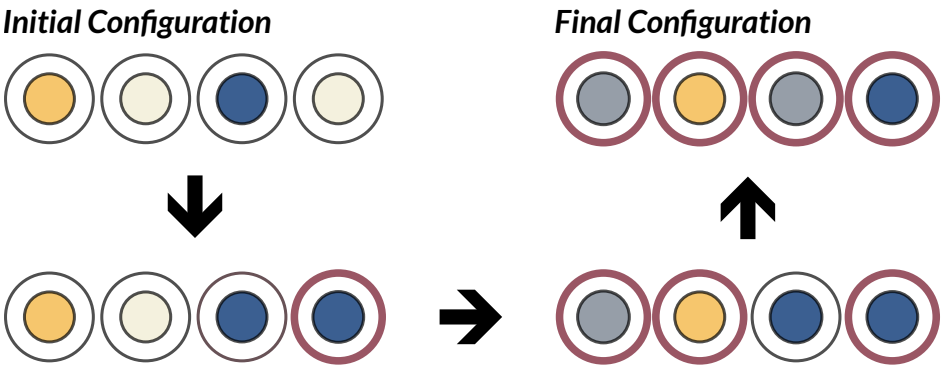


Figure 2.6: A possible sequence of reactions for the system described in Figure 2.5

CHAPTER III

GAME COMPLEXITY

3.1 Chapter Overview

This section considers the complexity of 2 different games in both 1-player and 2-player settings. In Celtic!, we show that between 0, 1, and 2-player variations of the original game, the complexity of the problem jumps in completeness for the classes P, NP, and PSPACE, which holds interesting implications for related models such as tile assembly and knot theory. We then consider k-ago, a generalization of k-in-a-row games, and show a distinction between different board sizes and values of k that result in both polynomial time solvable solutions and NP-completeness for determining a solution to the game.

3.2 Celtic!

3.2.1 Overview

This work is in collaboration with Divya Bajaj, Juan Manuel Perez, Rene Reyes, Ramiro Santos, and Tim Wylie. My contributions to this work include the introduction, the 1-player results and figures, preliminaries and definitions, and the proof of a guaranteed draw for boards of odd dimension.

3.2.2 Introduction

Square edge-matching puzzles generally employ the use of (possibly rotatable) tiles, with each side given a specific label that dictates how they may be used in conjunction with other tiles. These types of puzzles have existed for centuries, and are often deceptively difficult to solve. For instance, the Eternity II puzzle [95], introduced in 2007, offered $\$2,000,000$ for the first complete solution. While no restrictions were placed on the methods used, no complete solution has yet to

be found, with the closest solutions falling short a few pieces. The work of [35] shows that edge-matching puzzles, including MacMahon Squares [60], Scramble Squares, and TetraVex [87], are NP-complete and are equivalent to other types of puzzles such as jigsaw and polyomino packing puzzles.

The puzzles often remain NP-hard even for small instances. In [36], the authors show that even for a 1-by-*n* puzzle, unsigned edge-matching with rotation is still NP-hard and fixed parameter tractable in the number of unique labels. This work was later improved in [14], where it was shown that for both signed and unsigned edges, 1-by-*n* puzzles are NP-hard, even to approximate. Naturally, variations of these types of puzzles have been considered, such as having additional inequality constraints between adjacent tiles, triangular edge matchings, and when no target shape is specified [13].

In general, edge-matching puzzles are undecidable with unlimited pieces [11]. More abstractly, generalized edge-matching of squares (without rotation) is closely related to computational complexity as exhibited through Wang tiles [94], and more recently, self-assembly models such as the abstract Tile Assembly Model (aTAM) [97], which are both Turing Universal even with small tile sets.

Our work focuses on a game with similar edge-matching mechanics: Celtic! [17]. Celtic! is a 2-Player (Blue/Red) tile placement game where each player attempts to complete knots by alternatively placing pieces onto a fixed size board. Each piece can be thought of as a rotatable square with a 'closed' or 'open' label on each side, similar to that of unsigned edge-matching with two labels. The goal of the game is alternate playing pieces and completing knots. The player who has the most number of pieces of their color in a knot (over all knots) wins the game. See Figure 3.1 for an example of a completed game with scoring.

The pieces of Celtic! use crossing paths, which makes it a type of connection game where the goal is to build connecting paths. Path based pieces and mechanics show up in several games such as Tsuro [64], Squiggle [63], Travel [45], Kaliko/Psyche-Paths/Cram [80], and many others. Please see [16] for an overview of connection-based (and edge-matching) games.

3.2.3 Our Contributions

Table 3.1 overviews the main results. We characterize the game of Celtic! for 0, 1, and 2 player games based on the types of pieces.

We consider the complexity of three variations of the board game. To start, we look at a 1-player variation of Celtic! in which some pieces initially exist on the board and the goal of the player is to form a single closed knot of some length in k moves. We show that for some pieces, this is NP-complete in Section 3.2.5. We show that it is PSPACE-complete to determine if there is a 1^{st}-player win in a 2-player game (Section 3.2.6). In Section 3.2.7, we analyze a 0-player version with deterministic placement and show that making a closed knot of some length is P-complete.

3.2.4 Preliminaries

This section considers the mechanics of Celtic! along with formal definitions.

3.2.4.1 Game Mechanics. We first detail some game mechanics from the original game.

Pieces. The game pieces consist of 5 distinct types of pieces, with 5 copies of each type (25 pieces in total), with path segments (Figure 3.2a). The Blue and Red players each own 10 pieces (2 of each type with their background color), and they share 5 pieces with a **white** background.

Initial Configuration and Rules. The piece is placed to begin. Players take turns placing a piece adjacent to already existing pieces to form knots; each piece must be placed within the playing area such that no knot is prevented from closing (Fig. 3.2b). In the standard game, it must remain in a 5×5 square.

3.2.4.2 Definitions. We now provide formal definitions for the general game and problems afterwards.

Let $D(p_1, p_2)$ be the ℓ_1-norm (Manhattan distance) between points p_1, p_2. We say 2 cells $[x_1, y_1], [x_2, y_2]$ are **adjacent** if $D((x_1, y_1), (x_2, y_2)) = 1$.

Definition 3.2.1 (Board and cells)**.** *A **board** B is a fixed $w \times h$ lattice surface composed of **cells**, with each cell $[x, y] \in B$ corresponding to a location in row x and column y, and $[0, 0]$ corresponding to the bottom-leftmost cell.*

Definition 3.2.2 (Path-segment)**.** *A path-segment is a pair of connected edge points on a piece that represent a continuous path from one point to the other.*

Definition 3.2.3 (Piece)**.** *A **piece** $p \in \{$  *is a rotatable* 4*-sided unit square, where each side is labeled **open** or **close** denoting the type of side that may or may not attach on this side. Each open side has at most two path-segment points (over and under). Label two points on each side* $\langle 1,2,3,4,5,6,7,8 \rangle$ *going counterclockwise around the square (Figure 3.2a) as all path-segment points, Celtic! uses pieces (ignoring rotation) with path-segments (21)*  *, (25, 61)* *, (27, 81)* *, (27, 41, 83)* *, (27, 41, 63, 85)* *. We use the convention that each path-segment has the over number followed by the under number, and we use this rotation as the canonical orientation.*

Definition 3.2.4 (Matching and Paths)**.** *Two pieces p_1, p_2 are **matching** if they are in adjacent cells and the touching sides are both labeled open. A **path** is made of a sequence of path-segments from pieces. A path between p_1, p_2 conceptually connects the internal path-segment i_1o_1 from p_1 to path-segment i_2o_2 on p_2, and is defined as $P = \langle \ldots, i_1o_1, i_2o_2, \ldots \rangle$, where o_1, i_2 lie on the touching sides of p_1, p_2, and $o_1 \mod 2 = i_2 \mod 2$.*

Definition 3.2.5 (Valid piece)**.** *A piece in cell $[x_1, y_1]$ is said to be **valid** if:*

1. *$\exists [x_i, y_j]$ for $(i,j) \in \{(2,1),(1,2),(0,1),(1,0)\}$ such that $[x_1,y_1],[x_i,y_j]$ contain matching pieces, and*
2. *All other adjacent cells $[x_i, y_j]$ for $(i,j) \in \{(2,1),(1,2),(0,1),(1,0)\}$ satisfy (a) $[x_1,y_1],[x_i,y_j]$ contain matching pieces, (b) the touching sides of the pieces in $[x_1,y_1]$ and $[x_i,y_j]$ are both labeled closed, or (c) $[x_i,y_j]$ is empty.*

Definition 3.2.6 (Knot)**.** *A **knot** is a continuous path $P = \langle i_1o_1, \ldots, i_ko_k \rangle$ through path-segments on valid pieces $\langle p_1, \ldots, p_k \rangle$ where each path piece p_a identifies a corresponding input and output path-segment i_ao_a on a piece such that*

1. *each p_a matches with p_{a+1} for $1 \le a \le k-1$ and p_k matches with p_1, and*
2. *each o_a continues through i_{a+1} for $1 \le a \le k-1$ and o_k continues to i_1.*

Note that multiple path-segments through the same piece may exist. The length (score) of a knot is not the number of path-segments in pieces it crosses, but the number of ***distinct*** *pieces (or grid locations) the total path crosses. This definition ignores the topological properties of a knot in relation to other knots.*

3.2.4.3 Problem Definitions. This paper looks at 3 variations of Celtic!: a 0-Player simulation (Fig. 3.20), a 1-Player puzzle (Fig. 3.3), and a 2-Player game (Fig. 3.1).

Definition 3.2.7 (0-Player Celtic!)**.** *A* 0*-player Celtic! simulation is defined as:*

INPUT: *A* $w \times h$ *board partially filled with pieces* $\in$ {  ,  ,  ,  ,  }*, a* *piece as a starter piece and a positive integer L.*

OUTPUT: *Is it possible to place the starter piece in a cell* $[0,i]$ *rotated* 180 *degrees for some integer* $0 < i < w$ *and create a knot of length* $\geq L$*?*

Definition 3.2.8 (1-Player Celtic!)**.** *A* 1*-Player Celtic! puzzle is defined as:*

INPUT: *A* $w \times h$ *board partially filled with pieces, a multi-set S of pieces (with* $k = |S|$*) and positive integer L where* $k \leq L \leq k + L \leq wh$*.*

OUTPUT: *Can k valid pieces (from S) be placed such that a closed knot of length* $\geq L$ *is formed?*

Definition 3.2.9 (2-Player Celtic!)**.** *A* 2*-Player Celtic! game is defined as:*

INPUT: *A* $w \times h$ *board with a white* *piece in cell* $[i,j]$*, where* $1 \leq i \leq w$ *and* $1 \leq j \leq h$*, multi-sets* S_R*,* S_B*, and* S_W *of red, blue, and white pieces, respectively.*

MOVES: *Players take turns placing a valid move using one of their colored pieces or a white piece. Once no valid moves exist, each player chooses a completed knot and receives a score. The score* M_R, M_B *for each player is a positive integer denoting the number of pieces of their color from their chosen knot.*

OUTPUT: *Whether there is a Blue player win, a Red player win, or a draw based on* $\max(M_R, M_B)$*.*

3.2.4.4 Constraint Logic. The 0-player and 2-player reductions make use of constraint logic (CL) [46]. Problems in CL are based on a constraint graph where each weighted directed edge has value 2 (blue) or 1 (red), and every vertex has a minimum inflow constraint of 2. The game is based on flipping edges of the graph, where an edge can be flipped only if the minimum inflow constraints are maintained. In [46], they show that all graphs in CL can be reduced to a few simple gadgets, each of degree 3, that are equivalent for the games.

The 0-player gadgets for Bounded Deterministic Constraint Logic (BDCL), Figure 3.1c, require that the graph is non-planar and every edge may only be flipped once. They consist of an AND, FANOUT, and OR. BDCL is P-complete.

The Bounded 2-player Constraint Logic (B2CL) gadgets (Figure 3.1d) have assigned edges (white is 1^{st} player and black is 2^{nd}) and may only be flipped once by the owner. They consist of an AND, FANOUT, OR, CHOICE, and VARIABLE. B2CL is PSPACE-complete even for planar graphs. Note that the 2^{nd} player can only move in the variable gadget.

3.2.5 Generalized 1-player Celtic!

This section looks at generalized 1-Player Celtic! played on an $O(n) \times O(n)$ board, where the goal is to build a closed knot of length $\geq L$ in k moves. We start with polynomial time algorithms when restricted to placing  , , or  pieces. We then show a general framework for the NP-hardness reductions, which is used in the final section to show NP-completeness when restricted to either placing or , as well as for other piece combinations.

3.2.5.1 Polynomial Cases. This section considers the polynomial time solvable cases in generalized 1-player Celtic!.

Theorem 3.2.1. *Building a closed knot of length $\geq L$ with k moves by placing*  *pieces in Generalized 1-Player Celtic!, where the initial board configuration can contain* , ,  , , *and*  *pieces, is solvable in $O(n^2)$ time.*

Proof. For a given board B, we create an $n \times n$ array D, where each $D(i, j) = 0$ if $[i, j] \in B$ is empty, and $D(i, j) = 1$ if $[i, j] \in B$ is non-empty. Then, starting from $i = 0$ and $j = 0$, check if $D(i, j) = 1$.

If yes, run a breadth-first search (BFS) from $D(i,j)$, visiting any cardinally adjacent cell $D(i',j')$ if the pieces in locations $[i,j]$ and $[i',j']$ are matching. For each visited piece, add the number of open (unmatched) sides to some variable c. Consider all visited pieces as part of knot K. If $c \leq k$ and if $|K|+c \geq L$, then we can close K with the c moves. If all c moves are valid, return K as the chosen knot. Otherwise, we disregard the knot by changing the value of each visited cell to 0 and continue searching.

Since the  piece can only close a knot, there must already exist a non-closed knot K of length $\geq L-c$, where c denotes the number of pieces needed to close K. Thus, calculating $|K|$ and c for each existing knot in B determines if a knot of the correct length exists that can be closed. As each cell in D is visited at most twice (a 2^{nd} time to set the value to 0), the resulting runtime is then $O(n^2)$. □

Theorem 3.2.2. *Building a closed knot of length $\geq L$ with k moves by placing*  *pieces in Generalized 1-Player Celtic!, where the initial board configuration can contain*  , , , , *and* *pieces, is solvable in $O(n^2)$ time.*

Proof. The algorithm is similar to that of Theorem 3.2.1. We slightly modify table D to account for the fact that a piece can have up to 4 different paths which include point 1, 3, 5, or 7. Enumerate these points $0,1,2,3$. Create an $n \times n \times 4$ array D, with each $D(i,j,h) = 1$ if $[i,j] \in B$ is open in direction h, and 0 otherwise. Starting from i, j, h initially set to 0, for each $D(i,j,h) = 1$, run a breadth-first search (BFS) from $D(i,j,h)$, visiting adjacent pieces if they are matching along the path that includes h. For each visited piece, consider each cardinally adjacent cell $D(x,y,z)$ such that $[i,j]$ is open in the direction of $[x,y]$ and the paths including h and z can be matched. If $|y-j| = 1$ (the open side is north/south), set $D(x,y,z) = 1$ and place a piece in cell $[x,y]$, and if $|x-i| = 1$ (the open side is east/west), set $D(x,y,z) = 1$ and place a piece in cell $[x,y]$. For each and played, add 1 to some variable c. Then, visit any cardinally adjacent cell $D(i',j',z')$ if the pieces in locations $[i,j]$ and $[i',j']$ are matching (including the recently placed pieces) along the paths including z and z'. Once no more moves can be played, consider all visited pieces as part of knot K. If $c \leq k$ and if $|K| \geq L$, then we can create the knot K with c moves. If all c moves are valid

and if K is closed, return K as the chosen knot. Otherwise, we disregard the knot by changing the value of each visited cell to 0, remove the placed and pieces, and continue searching.

Since the piece is unable to close a knot on its own, any existing non-closed knot $K' \in B$ will only be part of a solution K if a sequence of pieces can be placed that result in the formation of a closed knot. This implies that each solution found along some given path will be unique. As a proof, assume the contrary: there exists solutions $K_1 \neq K_2$ with $K_1 \cap K_2 \neq \emptyset$. Let $p \in K_1 \cap K_2$ be a piece in cell $[x,y]$ and $p' \notin K_1 \cap K_2$ be a matching piece in cell $[x',y']$. Since p and p' are matching, p must have an open side in the direction of p'. Assume $p' \in K_1$. Since K_2 is a solution, a piece $z \neq p'$ must exist in $[x',y']$. However, as the player can only place pieces, it must be the case that $z = p'$. A piece then does not exist in $[x',y']$, implying K_2 is not a solution.

For each existing knot in B, a potential solution is completely grown out until no more pieces can be placed. Each cell in D is visited at most 2 times, resulting in a runtime of $O(n^2)$. □

Theorem 3.2.3. *Building a closed knot of length $\geq L$ with k moves by placing pieces in Generalized 1-Player Celtic!, where the initial board configuration can contain , , , , and pieces, is solvable in $O(n^2)$ time.*

Proof. Follows from Theorem 3.2.2, placing the piece instead of the and pieces for the algorithm. □

3.2.5.2 General Framework for Hardness Reductions. We outline the framework used for the hardness reductions. For each result, we reduce from directed, planar Hamiltonian cycle with max degree 3 [73]. At a high-level, given a directed, planar graph, we transform it into an equivalent rectilinearly embedded graph G_e [58]. We space out the vertices in G_e so each edge may be transformed to have roughly the same length (Figure 3.4). Denote this new graph G. We then construct the Celtic! board used for the reduction. We replace edges with and pieces, leaving a 7×7 empty region centered at each vertex, which is where the vertex gadget will be placed. The gadget consists of a piece with other pieces placed depending on the allowable pieces, and the main objective being that all incoming edges lie either to the left or right of the central vertex

piece, and all outgoing edges lie on the other side. This forces the player to place pieces that choose exactly one incoming edge and one outgoing edge per vertex gadget. If these choices are consistent across all vertices, then a knot of length L is formed corresponding to a Hamiltonian cycle in the original graph G_e, otherwise, the knot is not formed.

For each reduction, we give a vertex gadget with the different combinations of input and output on different sides for the construction using the pieces for that problem. Depending on the reduction, some additional pieces are needed to aid with making a choice between the incoming and outgoing edges at each vertex piece. Although not all cases are given, a few examples are shown in Figure 3.5.

We now formally detail how an initial Celtic! board is created from a given directed, planar graph. We start with some useful definitions and Lemmas.

Definition 3.2.10 (Edge)**.** *An edge for a rectilinearly embedded graph is defined as an ordered sequence* $e = \langle p_0, p_1, \ldots, p_{i-1}, p_i \rangle$*, where* p_0, p_i *are the integer coordinates of vertices* v_1, v_i *respectively and each* p_j *for* $1 \leq j \leq i-1$ *is the integer coordinate of a bend. Thus, an edge is drawn as the union of line segments* $\overline{p_j p_{j-1}} \forall 1 \leq j \leq i$*.*

Definition 3.2.11 (Edge length)**.** *The length of an edge* $e = \langle p_0, \ldots, p_i \rangle$ *for* $i \in \mathbb{N}$ *is defined as* $len(e) = \sum_{j=1}^{i} ||\overline{p_j p_{j-1}}||$*.*

Definition 3.2.12 (Horizontal Stretch)**.** *A horizontal stretch of a rectilinearly embedded graph G by some constant* $k \in \mathbb{N}$ *is defined as:*

1. $\forall v_{(x,y)} \in V$*, let* $(x,y) = (kx,y)$
2. $\forall e = \langle (x_1,y_1), \ldots, (x_j,y_j) \rangle \in E$*, with* $j \in \mathbb{N}$*, let* $e = \langle (kx_1,y_1), \ldots, (kx_j,y_j) \rangle$

Definition 3.2.13 (Vertical Stretch)**.** *A vertical stretch of a rectilinearly embedded graph G by some constant* $k \in \mathbb{N}$ *is defined as:*

1. $\forall v_{(x,y)} \in V$, *let* $(x,y) = (x,ky)$

2. $\forall e = \langle (x_1,y_1),\ldots,(x_j,y_j)\rangle \in E$, *with* $j \in \mathbb{N}$, *let* $e = \langle (x_1,ky_1),\ldots,(x_j,ky_j)\rangle$

Lemma 3.2.4. *Every planar graph with max degree* 4 *has a rectilinear embedding with at most* 2 *bends per edge such that:*

1. $\forall e \in E$, $d \leq len(e) < d+3$, *where* $d = \min_{e\in E} len(e)$

2. *The area is bounded by* $O(|V|^3) \times O(|V|^3)$

Proof. From [58], we get that any planar graph with max degree 4 has an equivalent rectilinearly embedded graph which contains at most 2 bends per edge and area at worst $(|V|+1)\times(|V|+1)$. Let G_e be the corresponding embedded graph. Start by vertically and horizontally stretching G_e by a factor of $|V|^2$. The resulting graph then has area at worst $|V|^2\cdot(|V|+1)\times|V|^2\cdot(|V|+1)$. Denote this graph G.

We then 'pump' each edge using the space created by the vertical and horizontal stretches as shown in Figure 3.6. Let e_{min}, e_{max} denote the edges with the smallest/largest lengths in G_e, so $len(e_{min}) = 1$ and $len(e_{max}) = 3\cdot(|V|+1)$. After stretching G_e by $|V|^2$, $len(e_{min}) = |V|^2$ and $len(e_{max}) = 3|V|^2\cdot(|V|+1)$. From pumping, $len(e_{min})$ can grow by $2\cdot(\frac{(|V|^2-2)}{2}\cdot\frac{(|V|^2-4)}{2}) = |V|^4-6|V|^2+8$ units. Thus, every edge can be sufficiently pumped to be of length $> len(e_{max})$. Furthermore, each time an edge is pumped, it grows either 3 units to start the pumping, or 2 units to continue. In the worst case, an edge of length $3|V|^2\cdot(|V|+1)-1$ will grow to be of length $3|V|^2\cdot(|V|+1)+2 = len(e_{max})+2$.

Since all edges can grow to be larger than length $len(e_{max})$, it follows that $\forall e \in E$, $d \leq len(e) < d+3$, where $d = len(e_{max})$. □

We now detail how the initial Celtic! board is created, which can be described as follows. Given a directed, planar graph with max degree 3, consider its rectilinear embedding $G = (V,E)$ in which $\forall e \in E$, $d \leq len(e) < d+3$, with d denoting the largest non-pumped edge (Lemma 3.2.4). We create the following initial configuration on a $(3|V|^2\cdot(|V|+1))\times(3|V|^2\cdot(|V|+1))$ board:

1. For each edge $e = \langle p_1, \ldots, p_i \rangle \in E$ with $i \in \mathbb{N}$: for all points (a,b) which lie on a line segment $l = \overline{(\mathrm{p_{j-1}p_j})} \forall 3 \leq j \leq i-1$, excluding the endpoints:

   (a) Place  in cell $[a,b]$ if the line segment is vertical; otherwise, place  in cell $[a,b]$ if the line segment is horizontal

2. For each edge $e = \langle p_1, \ldots, p_i \rangle \in E$ with $i \in \mathbb{N}$: for all $p_j = (x_j, y_j)$ with $3 \leq j \leq i-2$, consider the line segments $l_1 = \overline{\mathrm{p_{j-1}p_j}}$ and $l_2 = \overline{\mathrm{p_jp_{j+1}}}$:

   (a) If $(x_j - 1, y_j) \in l_1$ and $(x_j, y_j - 1) \in l_2$ (or vice versa), place 

   (b) If $(x_j, y_j - 1) \in l_1$ and $(x_j + 1, y_j) \in l_2$ (or vice versa), place 

   (c) If $(x_j + 1, y_j) \in l_1$ and $(x_j, y_j + 1) \in l_2$ (or vice versa), place 

   (d) If $(x_j, y_j + 1) \in l_1$ and $(x_j - 1, y_j) \in l_2$ (or vice versa), place 

3. For each edge $e = \langle p_1, \ldots, p_i \rangle \in E$ with $i \in \mathbb{N}$: for $p_2 = (x_2, y_2), p_{i-1} = (x_{i-1}, y_{i-1}) \in e$, place  in cells $[x_2, y_2]$ and $[x_{i-1}, y_{i-1}]$

4. For each vertex $v_{(x,y)} \in V$, remove all pieces in cells $[r,c]$ s.t $x-3 \leq r \leq x+3$ and $y-3 \leq c \leq y+3$.

3.2.5.3 Hardness Results. Using the framework discussed earlier, we now show NP-completeness for different piece combinations.

Lemma 3.2.5. *Generalized 1-Player Celtic! is in NP.*

Proof. For an $O(n) \times O(n)$ board, the maximum number of locations to play a move is $O(n^2)$. Thus, with $k \leq L \leq n^2$, it takes polynomial time to verify a length L knot formed in k moves. □ □

Theorem 3.2.6. *Building a closed knot of length $\geq L$ with k moves by placing*  *pieces in Generalized 1-Player Celtic!, where the initial board configuration can contain* *and* *pieces, is NP-complete.*

Proof. Start with the initial board described in the framework subsection. We construct 24 vertex gadgets, one for each possible input/output edge combination as shown in Figure 3.7, with a filled gadget shown in Figure 3.8. Let $L = |V| \cdot (d+3)$, where d is the length of the largest non-pumped edge. A Hamiltonian cycle exists if and only if it is possible to create a knot of size $\geq L$ in $k = 3n$ moves given $3n$  pieces.

Forwards Direction. There exists a Hamiltonian cycle. For vertex $v_{(x,y)}$ with $deg^-(v_{(x,y)}) = 2$, use 3  pieces to connect the  piece in the center of the vertex gadget to the respective 'input' edge. The same applies in the case of $deg^+(v_{(x,y)}) = 2$. With n vertices, this results in $3n$ total pieces that must be played. Due to the limited space around each central  piece, exactly 1 input and 1 output edge are chosen. If the output choice of one vertex does not match the respective vertices input, then a knot is not formed. From Lemma 3.2.4, each edge $e \in E$ is guaranteed to be of at least length d and within 5 units of each other. Thus, the resulting knot formed by connecting each vertex to the adjacent edges is of length $\geq |V| \cdot d + 3|V| = |V| \cdot (d+3)$. Any other knot which does not use an 'input' and 'output' edge for each vertex will have length $< L$.

Backwards Direction. A knot of length $\geq L$ can be formed in $k = 3n$ moves. For each edge 'chosen' in the configuration, traverse this edge in the given graph. Since the input and output edges are aligned to be on opposite sides, when a choice is made, exactly 1 input and 1 output edge is chosen. As a result, forming the knot of length $\geq L$ is only possible if every 'vertex' in the initial configuration is connected to an adjacent 'edge' on both the input and output locations.

From Lemma 3.2.4, the given graph is transformed into one that allows for all edge lengths to be no more than 3 units greater than the length of the largest non-pumped edge d. Thus, taking $L = |V| \cdot (d+3)$ guarantees a knot of at least L length can be formed if there exists a Hamiltonian cycle.

It then follows from Lemma 3.2.5 and the above that building a closed knot of length $\geq L$ by playing k pieces in Generalized 1-Player Celtic!, where the initial board configuration can contain  and pieces, is NP-complete. □

Theorem 3.2.7. *Building a closed knot of length $\geq L$ with k moves by placing  pieces in Generalized 1-Player Celtic!, where the initial board configuration can contain  ,  , and  pieces, is NP-complete.*

Proof. Start with the initial board described in the framework subsection. We construct 24 vertex gadgets, one for each possible input/output edge combination as shown in Figure 3.9, with a filled gadget shown in Figure 3.10. Let $L = |V| \cdot (d+4)$, where d is the length of the largest non-pumped edge. A Hamiltonian cycle exists if and only if it is possible to create a knot of size $\geq L$ in $k = 2n$ moves given $2n$ pieces. The proof follows from Theorem 3.2.6.

It then follows from Lemma 3.2.5 and the above that building a closed knot of length $\geq L$ by playing k  pieces in Generalized 1-Player Celtic!, where the initial board configuration can contain  ,  , and  pieces, is NP-complete. □ □

Theorem 3.2.8. *Building a closed knot of length $\geq L$ with k moves by placing  and  pieces in Generalized 1-Player Celtic!, where the initial board configuration can contain  ,  , and  pieces, is NP-complete.*

Proof. Start with the initial board described in the framework subsection. We construct 24 vertex gadgets, one for each possible input/output edge combination as shown in Figure 3.11, with a filled gadget shown in Figure 3.12. Let $L = |V| \cdot (d+9)$, where d is the length of the largest non-pumped edge. A Hamiltonian cycle exists if and only if it is possible to create a knot of size $\geq L$ in $k = 8n$ moves given $5n$ pieces and $3n$ pieces. The proof follows from Theorem 3.2.6.

It then follows from Lemma 3.2.5 and the above that building a closed knot of length $\geq L$ by playing k  and  pieces in Generalized 1-Player Celtic!, where the initial board configuration contains  ,  , and  pieces, is NP-complete. □ □

3.2.6 Constraint-Graph Reduction for 2-Player

We now analyze generalized 2-player Celtic! played on an $O(n) \times O(n)$ board where both Blue and Red players have a multi-set of pieces.

Lemma 3.2.9. *Given an $O(n) \times O(n)$ board configuration where both Blue and Red player have a multi-set of pieces containing pieces* [ ,  ,  ,]. *Deciding whether there is a sequence of moves to force a Blue player (player 1) win starting at a given board position is PSPACE-hard.*

Proof. To show that determining if Blue player has a forced win is PSPACE-hard, we reduce Bounded 2-player Constraint Logic, where a sequence of moves in the 2-Player Celtic! game represents the flipping of an edge in the Constraint Graph whose configuration represents the constraint-satisfaction problem of 2CL. The problem is defined as: Does the Blue player have a forced win?

We reduce Bounded 2CL by designing the VARIABLE, AND, OR, CHOICE and FANOUT gadgets for our game that are joined to create a constraint graph representing the sequence of moves. There is a Winning gadget that the Blue player can use to finish the knot. We show that the constraint graph accepts the configuration when the Blue player wins.

3.2.6.1 VARIABLE Gadget. (Figure 3.13a) We need to show that this gadget satisfies the same constraints as that of a variable vertex in the Bounded 2CL constraint graph. If the Red player moves first, it closes the knot using the piece  preventing Blue player from continuing an edge to other gadgets. But if the Blue player moves first, it can place a piece  that restricts the red player to only continue the knot to other gadgets.

3.2.6.2 WIRE Gadget. A wire is a walled path of width 1 that connect gadgets.

3.2.6.3 WINNING Gadget. (Figure 3.13b) The winning gadget is used by the blue player to close the knot once they reach it. In all other gadgets, the players continue with the knot, and they finally lead into the winning gadget. Here only the blue player is able to close the knot using the  as shown in Figure 3.13c.

3.2.6.4 OR Gadget. (Figures 3.14a, 3.14b) The construction of this OR gadget in our game satisfies the same constraints as a Bounded 2CL OR vertex. For the player to continue with a valid knot, the player can connect a knot from the right side using the  or from the left side using the  . Another knot can later be used to close the opposite Input side using  .

3.2.6.5 CHOICE Gadget. (Figure 3.14c) The construction of this gadget satisfies the same constraints as that of a CHOICE vertex in a Bounded 2CL where a sequence of moves mark whether the knot can be extended towards left or right. For the player to continue with a valid knot, the player can use either  or  . Depending on the piece used, the knot will then continue right or left. No knot can be completed from the opposite side.

3.2.6.6 AND Gadget. (Figure 3.15a) We now show that the construction of this gadget satisfies the same constraints as that of the AND vertex in a Bounded 2CL constraint. For the player to continue with a valid knot, the player can use both  and  . If the player only connects a knot from one side, the knot will not be closed. Hence, the pieces in the knot will not count towards the players.

3.2.6.7 FANOUT Gadget. (Figure 3.15c) This gadget satisfies the same constraints as that of the FANOUT vertex in a Bounded 2CL constraint graph. The construction is quite similar to the AND gadget where the Output edge is now the Input edge. If the player is able to connect to the Input edge, then the player can complete the knot by connecting on both sides using  and  .

Since the Red player only has  , they can use it in the Variable gadget to block, or continue with existing knots in all the gadgets. The Blue player makes moves in the Variable gadgets. If the Blue player is able to assign the Variables to true (place in the Variable gadget), they can then satisfy other constraints by continuing knots in other gadgets. Once the knot reaches the Winning gadget, the Blue player can close it, thus ending and winning the game. If both players make alternate moves and Blue player makes the first move, the winning gadget will ensure that the Blue player is the last player to make a move. Thus, Blue has at least one piece more than Red, making them the winner.

The construction of these gadgets satisfies all constraints as that of vertices in a constraint graph of a Bounded 2CL. We know that finding the variables assignments for the constraint logic problem for a Bounded 2CL is PSPACE-complete. Such variable assignments correspond to the Blue and Red player making moves in the variable gadgets, setting them to true or false. Therefore

deciding if the Blue player can make certain moves in the variable gadgets such that the Blue player always wins is PSPACE-hard. □ □

Lemma 3.2.10. *Given an $O(n) \times O(n)$ board configuration where both Blue and Red player have a multi-set of pieces containing pieces* []. *Deciding whether there is a sequence of moves the Blue player can make that results in a forced win starting at a given board position is in PSPACE.*

Theorem 3.2.11. *Given an $O(n) \times O(n)$ board configuration where both Blue and Red player have a multi-set of pieces containing pieces* [ ]. *Deciding if the given board configuration is a 1^{st} player win is PSPACE-complete.*

3.2.7 Constraint Logic Reduction for 0-Player

This section considers a 0-player variation of Celtic! with rule constraints to allow for automation, and shows that 0-player Celtic! is P-complete.

3.2.7.1 0-Player Decisions. We define placement rules for an automated 0-player to make decisions.

- Structural Pieces - Existing white pieces used as walls or requiring specific pieces to traverse.
- Signal Pieces - The automated blue pieces that propagate through the board.
- Close South Rule - A  piece is always placed to close a southward path unless there is another adjacent cell that requires a different piece (Fig. 3.17a).
- 3-Open Piece Rule - A  piece is placed when 3 sides are possible (Fig. 3.17b).
- 4-Open Piece Rule - A is placed when 4 open sides are possible, which also allows for crossovers in the graph (Fig. 3.17c).

3.2.7.2 0-Player Bounded Deterministic Constraint Logic Gadgets. Bounded DCL is P-complete [46] since we can use it to simulate monotone boolean circuits, which only requires us to simulate simple AND, OR and FANOUT gadgets. We reduce from Bounded DCL to 0-Player Celtic! simulation. We show the construction of the AND, OR and FANOUT in Figure 3.19.

3.2.7.3 WIRE Gadget. A wire is a walled (by structural pieces) path of width 1 that connects gadgets.

3.2.7.4 OR Gadget. The OR gadget has two inputs and one output. If a signal piece propagates through the left input, by the *4-Open Sided Rule*, it will propagate vertically through the 4 open-sided structural piece (Figure 3.19). As it propagates north, it splits into 2 signals: one of them continues as the output of the gadget, the other takes a turn left. By the *Close South Movement Rule* and *4-Open Sided Rule* it will then connect with the 4 open-sided piece and continue horizontally until it closes going southward through the other input path. Similarly, if the signal piece propagates on the right input, the signal traverses the  structural piece horizontally, continuing north with its output, and eventually propagating south vertically through the  piece and closes the other input. If another signal tries to propagate through the opposite side it will immediately close, but the signal has already propagated from the first input.

3.2.7.5 AND Gadget. By the *3-Open Sided Rule*, a signal piece coming from either input cannot continue through the output unless the automated player has connected signal pieces to both parallel sides of the structural piece (Figure 3.19).

3.2.7.6 FANOUT Gadget. By the *3-Open Sided Rule*, a signal piece propagating through the perpendicular open side of the  piece allows the automated player to split the signal and propagate through both outputs (Figure 3.19).

3.2.7.7 Simulating DCL with Gadgets. Figure 3.20 demonstrates how the gadgets on a board can simulate bounded DCL. The goal is to attempt to reach the output from the top AND gadget. Notice it can only reach the output by playing in the middle starting position where the signal splits in the FANOUT gadget, through the 2 OR gadgets, and then into the final AND gadget. The goal of a 0-Player Celtic! simulation is to generate a knot of size $\geq L$, so you can extend the output of the last top gadget to be longer than any possible knot before that, and let L equal to that. Thus, the only knot that can achieve this is the knot that can output through the last gadget.

Theorem 3.2.12. *0-Player Celtic! is P-complete.*

3.2.8 Generalized Celtic! with Boards of Odd Dimension

This section considers generalized Celtic! played on boards of odd dimension, where the count of each  ,  ,  ,  and  white pieces are even and the set of pieces each player gets is equal. This includes when the white pieces are excluded altogether, or if the number of white pieces are assumed infinite. We show that through a *strategy-stealing* argument, where player 2 takes advantage of the games symmetry to prevent player 1 from having a winning strategy, the game is guaranteed to be a draw. Let S_B, S_R and S_W be multi-sets of blue, red and white pieces, respectively.

Theorem 3.2.13. *Celtic! played on an $n \times n$ board, where n is odd, is a guaranteed draw when $S_B = S_R$ and the count of each*  ,  ,  ,  ,  $\in S_W$ *is even.*

Proof. We employ *strategy-stealing* that takes advantage of the symmetric nature of the game to show that player 2 has a way to guarantee a draw. Assume the first player places a valid piece p in cell $[i, j]$ for $0 \leq i, j < n$. Player 2 then 'steals' the strategy of player 1 and places the same piece p rotated 180 degrees in cell $[w, z]$ with $w = |n-1-i|$ and $z = |n-1-j|$. The resulting board with no valid moves left is then symmetric w.r.t the origin, since we are assuming $S_B = S_A$. Let $K = \{[x_1, y_1], \ldots, [x_i, y_i]\}$ be a knot with length $|K|$. Consider 2 cases, where $[x_{n/2}, y_{n/2}]$ represents the center of the board:

Case 1: $[x_{n/2}, y_{n/2}] \in K$. Let $K_L, K_R \subset K$ be 2 subsets of K denoting the 2 sub-knots formed by cutting K at cell $[x_{n/2}, y_{n/2}]$. Assume K_L contains P_B blue pieces and P_R red pieces such that $P_B \geq P_R$. Thus, K_L has a surplus of $P_B - P_R$ blue pieces. Each piece in cell $[a, b] \in K_L$ is mapped to the same piece of the opposite color in cell $[a', b'] = [|n-1-a|, |n-1-b|]$, with $[a', b'] \in K_R$. From here we get K_R has a surplus of $P_R - P_B$ red pieces. The total number of blue and red pieces in K is then equivalent.

Case 2: $[x_{n/2}, y_{n/2}] \notin K$. Let $K' = \{[|n-1-x_a|, |n-1-y_a|] \mid \forall (x_a, y_a) \in K\}$. Assume K contains a surplus of $P_B - P_R$ blue pieces. K' then must have a surplus of $P_R - P_B$ red pieces. Thus, K or K' give both players the same advantage. □

3.2.9 Conclusion

This work characterizes and analyzes the complexity of 3 variations of Celtic!: The 0, 1, and 2 player variants depending on the pieces allowed. In the 1-Player version, we show NP-completeness or membership in P depending on the pieces allowed. For the 2-player variation, deciding if there is a first-player win is PSPACE-complete. We show that a 0-Player variation with forced play, determining if a closed knot of some length can be formed after an initial starter piece is P-complete. This work naturally leads to some open questions, such as those in the table and below.

- What is the complexity with different pieces for Red or Blue player?
- What is the complexity of the different games given a board of fixed dimension? What restricted board instances are polynomial time solvable?
- What is the complexity of the two open cases in Table 3.1?
- We show that for any general board of odd dimension with even numbers of each piece, the game always ends in a draw. Without these restrictions, is the standard game a first-player win?
- What if we look at pattern complexity based on the knots? General patterns could be answered similar to the PATS (patterned self-assembly tile set synthesis) problem [59], but if we consider the topology of the knots, the pattern is no longer based solely on the tile-type, or even color, at some location.

3.3 Generalized k-in-a-row Matching with k-ago

3.3.1 Overview

This work is in collaboration with Juan Manuel Perez, Kwabena Aboagye-Otchere, and Aiden Massie. My contributions to this work include the results, a writeup of the definitions, and the figures.

3.3.2 Introduction

Connect 4[37], Go-moku[76], Candy Crush[44], and tic-tac-toe[20] are some of the most popular puzzle games today, despite being introduced several decades prior. Among games such as these lies a common theme: the use of *k*-in-a-row to accomplish some objective. As such, extensive work has focused not only on studying the complexity of each individual game, but also the more general *k*-in-a-row mechanic [51].

For instance, Edelkamp and Kissmann[37] showed that the complexity of *k*-in-a-row in Connect Four, including its state space and termination conditions, can be represented by polynomial-sized binary decision diagrams (BDD). The set of reachable states can be efficiently captured with polynomial-sized BDDs under optimal variable ordering. However, the termination criterion, when a player achieves *k*-in-a-row, where $k = 4$, leads to an exponential growth in the size of the BDD, depending on the order. This complexity arises from the need to evaluate **all** possible configurations of *k*-aligned pieces across rows, columns, and diagonals, which exponentially increases the number of required BDD nodes.

The exploration of *k*-in-a-row mechanics in Connect Four by Edelkamp and Kissmann highlights the general applicability of their findings in games like Tic-Tac-Toe and Gomoku in which increasing board dimensions and *k*-values present an exponential complexity in data representation and problem-solving. In Stefan Reisch's work on Gomoku, detailed in "Gomoku is PSPACE-Complete", the focus is shifted to the computational challenge of determining whether a player has a winning strategy. Reisch demonstrates that Gomoku's decision problem is PSPACE-complete, relying on polynomially bounded game durations and reducing it from PSPACE-complete games such as Generalized Geography.

These findings on the computational intricacies of *k*-in-a-row games, including Connect Four and similar games such as Gomoku, translate to Stefan Reisch's work on Gomoku. In "Gobang ist PSPACE-vollständig", Reisch explores the decision problem of determining whether a player has a winning strategy in Gomoku (referred to as Gobang in the paper)[76], a 5-in-a-row game, and demonstrates that it is PSPACE-complete. By reducing the problem from Generalized Geography,

Reisch establishes that Gomoku belongs to the same complexity class. Moreover, the findings are generalized to k-Gomoku ($k > 5$), further revealing how the computational depth of k-in-a-row games scales with board size and k-values.

Another example is shown by Hamilton, Nguyen, and Roughan[44], in which the complexity of counting stable configurations in the mobile game, Candy Crush, can be understood with the use of hypergraph colouring. By framing the game mechanics as a problem of poor proper colorings in a k-uniform hypergraph, the authors developed a Fully Polynomial Randomized Approximation Scheme (FPRAS) to estimate the size of the game state space efficiently. Additionally, while it was claimed that the exact computation of the state space is computationally intractable, the use of a multilevel splitting algorithm and the Moser-Tardos technique allows for probabilistic estimation of configurations where no matchings occur, that is, "stable" Candy Crush boards. This approach demonstrated scalability for large grids and provided insights into broader applications for hypergraph colouring problems, while also exposing limitations in cases involving fewer colors or constraints closer to the original game settings.

Finally, in "Tic-Tac-Toe on a Finite Plane" by Maureen T. Carroll and Steven T. Dougherty [20], it was shown that extending the classic k-in-a-row mechanic of tic-tac-toe to affine and projective planes adds significant geometric and strategic complexity. In these variants, winning lines are not limited to straight rows, columns, or diagonals but include configurations prescribed by finite geometries, such as lines from mutually orthogonal Latin squares. The paper explores how the structure of these planes influences gameplay, highlighting that the outcomes depend on the interaction between the number of points per line (k) and the total number of points (n^2). For larger planes, strategies shift from traditional moves to a geometric approach that prioritizes blocking winning lines. In particular, on projective planes of the order $n \geq 3$, the second player can guarantee a draw by employing weight-based strategies to block critical configurations.

In light of this extensive research across several different yet related k-in-a-row games, our work looks to further contribute to what is currently known. We focus on k-ago, a game which relies on k-in-a-row mechanics to fill in the board, or further, turn all pieces gray. Specifically, we

can break our work into 4 sections. In Section 3.3.4, we consider a restricted version of k-ago in which the 2-dimensional board is fixed in at least 1 dimension, providing several polynomial time algorithms for both versions of the 1-Player puzzle. In Section 3.3.5, we consider the generalized version of both puzzles, in which the problem becomes NP-complete for $k \geq 3$. These results are outlined in Table 3.2. In Section 3.3.6, we look at a 2-Player version of k-ago where each player attempts to place a majority of their pieces on the board, using k-in-a-row matchings to attain an advantage. Under these assumptions, we show that for boards of odd dimension, the first player has a guaranteed win, while for boards of size 4×4, the result is a guaranteed draw. Finally, in Section 3.3.7, we show that for an optimization version of TAPG k-ago, namely Maximal Gray k-ago, each k admits a k-approximation algorithm where in general the problem is NP-complete for $k \geq 3$.

3.3.3 Preliminaries

This section provides definitions and preliminary information for the results listed in Sections 3.3.4, 3.3.5 and 3.3.6.

3.3.3.1 Board and Cells. A *board* B is a fixed $w \times h$ lattice surface composed of *cells*, with each cell $c_{(x,y)} \in B$ corresponding to a playable location in row x and column y, and $c_{(0,0)}$ corresponding to the bottom left most cell.

3.3.3.2 Pieces. Pieces consist of 2 sides: one *dark*, and one *light*. Pieces initially placed on the board may either be dark or light; however, all pieces played by a player must be dark.

3.3.3.3 1-Player k-ago. The game of *1-Player k-ago* is played on a board B of size $w \times h$. All pieces are the same, and contain a *black (dark)* and a *gray (light)* side. The board starts partially filled with gray and black pieces, with the rest of the pieces given to the player. At each turn, the player may place 1 black piece in any empty cell. If placing this piece results in k-in-a-row of black pieces, the k pieces are flipped to gray and the player gets another move. If more than one k-in-a-row matching exists, the player gets to choose *one* k-in-a-row matching and flips the k pieces that are part of the chosen matching.

In *Fill-in-the-Board (FITB) k*-ago, the player wins by completely filling out the board, regardless of the orientation of the pieces. In *Turn-All-Pieces-Gray (TAPG) k*-ago, the player must additionally flip all pieces to gray. Example FITB/TAPG 3-ago games are shown in Figures 3.21 and 3.22, respectively.

3.3.3.4 2-Player *k*-ago. The game of *2-Player k-ago* is played on a board B of size $w \times h$. Pieces now consist of 2 different colors: a *blue* color for player 1 and a *red* color for player 2. Each piece consists of 2 sides: a *dark blue/dark red* side, and a *light blue/light red* side. The board starts empty, with each player given the same number of pieces. The players take turns placing a piece of their color on the board in any empty cell. If placing this piece results in *k*-in-a-row of their respective *dark* color, the player can then choose to flip the *k* pieces to their respective *light* color and get another move. If more than one *k*-in-a-row matching exists, the corresponding player gets to choose *one k*-in-a-row matching and flips the *k* pieces that are part of the chosen matching. The game is won by whichever player places the most number of blue/red pieces on the board.

The following are formal definitions of each problem we consider:

Definition 3.3.1 (1-Player TAPG *k*-ago)**.** *The 1-Player TAPG k-ago puzzle is defined as follows:*

INPUT: *A* $w \times h$ *board partially filled with black and gray pieces, and an integer* $2 \leq k \leq \max(w,h)$.

OUTPUT: *If there exists a sequence of k-in-a-row moves that turns all pieces gray.*

Definition 3.3.2 (1-Player FITB *k*-ago)**.** *The 1-Player FITB k-ago puzzle is defined as follows:*

INPUT: *A* $w \times h$ *board partially filled with black and gray pieces, and an integer* $2 \leq k \leq \max(w,h)$.

OUTPUT: *If there exists a sequence of k-in-a-row moves that fills the entire board.*

Definition 3.3.3 (2-Player *k*-ago)**.** *2-Player k-ago is defined as follows:*

INPUT: *A* $w \times h$ *empty board and an integer* $2 \leq k \leq \max(w,h)$.

MOVES: *Players alternate turns placing a dark piece of their respective color. If k-in-a-row of their dark color is formed, the k pieces are flipped to their light and the player gets another*

turn. The score of each player S_1, S_2 is determined as the number of blue/red pieces on the board, regardless of orientation.

OUTPUT: *Whether player 1 wins, player 2 wins, or the result is a draw based on* $\max(S_1, S_2)$.

3.3.4 Restricted k-ago

We start by considering polynomial time algorithms for restricted versions of k-ago, fixing the height of the board by a constant.

Observation 3.3.1. *TAPG k-ago played on a $O(1) \times O(1)$ board is solveable in polynomial time.*

Proof. Let $c > k$ be a constant denoting the width and height of the board. The maximum number of empty locations permissible is $\frac{c^2}{k}$, with $4k$ total ways for each location to be matched. Thus, a brute force solution checking all possible matchings runs in $4k^{\frac{c^2}{k}}$.

While not surprising, we include for completeness. □

Observation 3.3.2. *FITB k-ago played on a $O(1) \times O(1)$ board is solveable in polynomial time.*

Theorem 3.3.3. *TAPG k-ago played on a $1 \times n$ board is solveable in $O(k \cdot n)$ time.*

Proof. We use dynamic programming. Let D be a $1 \times n$ array, where each $D(j) = -1$ if $C_{(0,j)}$ is empty, $D(j) = 0$ if $C_{(0,j)}$ contains a gray piece, and $D(j) = 1$ if $C_{(0,j)}$ contains a black piece. Run Algorithm 15. If all cells contain values of 0, then all pieces can be turned gray and the board is completely filled. Otherwise, it is not possible to turn all pieces gray and fill the board.

The intuition for the algorithm is as follows. Start from the left most cell and create a sliding window of size k. If there is an empty cell, and the remaining $k-1$ pieces are black, flip them to gray (including the empty cell). Shift the sliding window over to the right 1 unit. At the end of this process, the entries in D tell us whether or not all pieces can be flipped to gray.

We now include a more formal proof of the algorithm. Consider the left most cell $c_{(0,l)}$ such that $c_{(0,l)}$ is empty. Let $c_{(0,r)}$, with $r > l$, denote an empty cell to the right of $c_{(0,l)}$. Assume that there exists a cell $c_{(0,b)}$ containing a black piece satisfying $b < l < r$ and $b \geq r - k$. Since $c_{(0,l)}$ is

empty, a piece played at $c_{(0,r)}$ can never flip the black piece in $c_{(0,b)}$. This would require a black piece to already exist in $c_{(0,l)}$, but this implies a piece is played and never flipped, resulting in the game ending. Thus, $c_{(0,b)}$ must be flipped by the played piece in $c_{(0,l)}$. It is then clear that for any cell, starting from left to right, it is sufficient and necessary to take the left most sliding window to find each matching.

Each pass of the sliding window takes $O(k)$ time to compare all the elements. With n total elements and with $k \leq n$, the runtime is then $O(k \cdot n)$. □

Theorem 3.3.4. *FITB k-ago played on a* $1 \times n$ *board is solveable in* $O(k \cdot n)$ *time.*

Proof. The algorithm is similar to that of Theorem 3.3.3. At the end, instead of checking D for values of 0, check that all values in D are non-negative. If so, then the board can be completely filled. Otherwise it is not possible. □

Theorem 3.3.5. *TAPG k-ago played on a* $w \times n$ *board is solveable in* $O(w \cdot k \cdot n)$ *time, where* $1 \leq w < k$.

Proof. Since there are $w < k$ rows, matchings can only occur horizontally. Thus, by running the algorithm from Theorem 3.3.3 on each row, we get a solution to the problem. □

Theorem 3.3.6. *FITB k-ago played on a* $w \times n$ *board is solveable in* $O(w \cdot k \cdot n)$ *time, where* $1 \leq w < k$.

Proof. Follows from Theorems 3.3.4 and 3.3.5. □

3.3.5 Generalized k-ago

We now consider a generalized version of k-ago, which is played on an $O(n) \times O(n)$ board. We start by providing a polynomial time algorithm for $k = 2$, followed by NP-completeness for the case of $k = 3$.

Theorem 3.3.7. *Generalized TAPG 2-ago is solveable in $O(n^2)$ time.*

Proof. Let L contain the set of cells on the board that are empty and let R contain the set of cells on the board occupied by a black piece. An edge exists between any $l \in L$, $r \in R$ if $l_{(i+c_1, j+c_2)} = r_{(x,y)} \mid c_1, c_2 \in \{-1, 0, 1\}$. In other words, an edge exists between l and r if the 2 cells are directly adjacent to one another.

If there exists a matching such that all cells from L and R are matched, then it is possible to completely fill the board and turn all pieces gray by using the matching. Otherwise, it is not. This problem is maximum bipartite matching, which is solveable in $O(\sqrt{V} \cdot E)$ time. The number of edges is at most $8n$, and the number of vertices is bounded by the size of the board, or n^2. It follows that the runtime is $O(n^2)$. □

Theorem 3.3.8. *Generalized FITB 2-ago is solveable in $O(n^2)$ time.*

Proof. Follows from Theorem 3.3.7. The difference is a matching only needs to ensure that all cells from L are matched to something in R. If a matching exists, then it is possible to completely fill the board. Otherwise, it is not. □

Lemma 3.3.9. *Generalized TAPG/FITB k-ago $\in NP$.*

Proof. With a board size of $O(n) \times O(n)$, the maximum number of pieces playable is $O(n^2)$. Thus, verifying a solution to the problem can be done in polynomial time. It follows that Generalized TAPG/FITB k-ago $\in NP$. □

Lemma 3.3.10. *Every planar graph with max degree 4 has a rectilinear embedding with at most 2 bends per edge such that the resulting area is at worst $(|V|+1) \times (|V|+1)$ [58].*

Theorem 3.3.11. *Generalized TAPG 3-ago is NP-complete.*

Proof. To show NP-hardness, we reduce from directed, planar Hamiltonian cycle with max degree 3 [73]. We start by transforming the given graph into one that will make the reduction easier, then introduce the corresponding vertex and wire gadgets.

Graph Transformation. Let $D(p_1, p_2)$ be the Euclidean distance between points p_1 and p_2. Consider the equivalent planar, rectilinearly embedded graph $G = (V, E)$ for a given directed, planar graph of max degree 3 (Lemma 3.3.10), with each $v_{(x,y)} \in V$ located at coordinate (x, y) and each $e = \langle p_1, b_1, b_2, p_2 \rangle \in E$, where p_1, p_2 are the integer coordinates of vertices $v_1, v_2 \in V$ and b_1, b_2 are the integer coordinates of the at most 2 bends of e such that $D(p_1, b_1) < D(p_1, b_2)$. We start by scaling G by a factor of 3.

1. $\forall v_{(x,y)} \in V$, set $(x, y) = (3x, 3y)$
2. $\forall e = \langle (x_1, y_1), \ldots, (x_i, y_i) \rangle \in E$, with $2 \leq i \leq 4$, set $e = \langle (3x_1, 3y_1), \ldots, (3x_i, 3y_i) \rangle$

This step is important, as it ensures that there is sufficient space between adjacent edges and vertices. We then modify the graph slightly for the reduction. Let vertex $v_{(x_1,y_1)} \in V$ where $deg^+(v_{(x_1,y_1)}) = 1$. Denote the output edge as $e_O = \langle (x_1, y_1), \ldots, (x_i, y_i) \rangle$, with $2 \leq i \leq 8$. Consider 4 cases:

1. $y_1 - y_2 < 0$ (output is to the north). Adjust the edges as shown in Figure 3.23a.
2. $x_1 - x_2 < 0$ (output is to the east). Adjust the edges as shown in Figure 3.23b.
3. $x_1 - x_2 > 0$ (output is to the west). Mirror the adjustments along the y axis from Case 2.
4. $y_1 - y_2 > 0$ (output is to the south). Mirror the adjustments along the x axis from Case 1.

Denote the 2 input edges as $e_I^1 = \langle (w_i, z_i), \ldots, (x_1, y_1) \rangle$ and $e_I^2 = \langle (f_i, g_i), \ldots, (x_1, y_1) \rangle$. The new edges which include $v_{(x_1,y_1)}$ become $e_O = \langle (x_1, y_1), a_1, \ldots, a_j, \ldots, (x_i, y_i) \rangle$, $e_I^1 = \langle (w_i, z_i), \ldots, b_1, \ldots, b_k, (x_1, y_1) \rangle$ and $e_I^2 = \langle (f_i, g_i), \ldots, c_1, \ldots, c_l, (x_1, y_1) \rangle$ for $j, k, l \in \{3, 4\}$ (3-4 bends are added per edge).

A similar rearrangement is made if $deg^-(v_{(x_1,y_1)}) = 1$, treating this edge as the output edge from the steps described above. The importance of the rearrangement is that the incoming edges and outgoing edges are split into 2 separate regions. Note that the resulting graph after edge adjustments

is no longer a rectilinear embedding, and each edge now contains up to 8 new bends; however, this does not affect the reduction, as we mostly care for planarity.

The final step is to scale the resulting modified graph G again by a factor of 3. This is due to the fact that in our reduction, wires are composed of groups of 3 elements, which requires graph components to be spaced out by 3 units.

We now introduce the 2 gadgets which will be used in the reduction.

Vertex Gadget. The vertex gadget consists of 'input' and 'output' locations. These locations are adjacent to wires that represent incoming and outgoing edges, respectively. When a piece is placed in the 'output' location, a choice is made between the 2 wires in which one is 'selected' and one is 'not selected'. These choices are then propagated along the wire to the 'input' location of the receiving vertex gadgets. Examples of vertex gadgets with 2 outputs are shown in Figure 3.24a. If the vertex has only 1 output edge, the gadget is slightly modified, as shown in Figure 3.24b.

Wire Gadget. The wire gadget consists of a repeated sequence of 2 adjacent black pieces followed by 1 empty location. Signals are passed depending on which black pieces are chosen by the empty location to flip with. This is shown in Figure 3.25.

Reduction. We now present the reduction, with Figures 3.26a, 3.26b and 3.27 showing concrete examples of this procedure. Given a directed, planar graph with max degree 3, follow the transformations described in the *Graph Transformation* section to construct the equivalent directed, rectilinearly embedded graph $G = (V, E)$. Then on a $9(|V|+1) \times 9(|V|+1)$ board:

1. $\forall v_{(x,y)} \in V$, replace $\{\{c_{(x-3,y-3)}, \ldots, c_{(x+3,y-3)}\} \times \{c_{(x-3,y+3)}, c_{(x+3,y+3)}\}\}$ with the vertex gadget from Figure 3.24a if $deg^+(v_{(x,y)}) = 2$, otherwise replace with the vertex gadget from Figure 3.24b.

2. $\forall e = \langle p_1, \ldots, p_i \rangle \in E$ with $8 \leq i \leq 12$: for all points (a,b) which lie on a line segment $\overline{\mathrm{p_{j-1}, p_j}} \forall 1 < j < i$, leave cell $c_{(a,b)}$ empty if $(a+b) \pmod 3 = 0$, otherwise place a black piece.

Note that the line segments going directly into/out of each vertex are ignored when placing wires. This is because the vertex gadgets overlap with these locations; once each vertex gadget is placed, the wires are connected to the 'input' and 'output' locations for the vertex.

We now consider correctness. That is, we show that G contains a Hamiltonian cycle if and only if all pieces can be turned gray.

If there exists a Hamiltonian cycle, we take each edge (v_i, v_j) and assign each 'output' location for v_i to the corresponding wire leading to the 'input' location for v_j. For the unused edge (v_i, v_u), all black pieces are flipped leading up to the 'input' location for v_u, without placing a piece in this location. Thus, in order for all the pieces to be able to flip, the signal from the other 'input' location is forced to be 'selected'; otherwise, one empty location will be left with no adjacent black pieces, or conversely, 2 black pieces will be left unflipped.

If all pieces can be turned gray, then there exists a Hamiltonian cycle. For each 'output' location, take the chosen wire to be the respective edge traversal in the graph. Since incoming/outgoing edges are separated into 2 regions, and each 'input'/'output' location can only flip once, this forces each vertex gadget to select exactly 1 incoming and 1 outgoing edge, and conversely, not select the other edges.

From Lemma 3.3.10, the rectilinear embedding of a planar graph with n vertices is contained within an area of $(n+1) \times (n+1)$. We scale the board by 9, resulting in an $O(n) \times O(n)$ size board for the reduction. It follows from Lemma 3.3.9 and the above that generalized TAPG 3-ago is NP-complete. □

Theorem 3.3.12. *Generalized FITB 3-ago is NP-complete.*

Proof. We modify the reduction from Theorem 3.3.11 by adding a single isolated location surrounded by gray pieces. Thus, the only way to completely fill in the board is by first turning all the pieces gray within the $9(|V|+1) \times 9(|V|+1)$ region, then placing the final piece in this isolated location. NP-completeness then follows from Lemma 3.3.9 and Theorem 3.3.11. □

Corollary 3.3.13. *Generalized TAPG 1-Player k-ago is NP-complete for $k \geq 3$.*

Corollary 3.3.14. *Generalized FITB 1-Player k-ago is NP-complete for $k \geq 3$.*

3.3.6 2-Player k-ago

This section considers 2-Player *k*-ago. We start by showing that for boards of odd dimension, 2-Player *k*-ago is a first player win. This uses ideas similar to John Nash's proof of a first player win in the game Hex [66].

Lemma 3.3.15. *When played optimally,* ***one*** *of the following holds for 2-Player k-ago played on an $n \times n$ board:*

1. *Player 1 has a winning strategy*
2. *Player 2 has a winning strategy*
3. *If (1) and (2) are not true, then both players can force at least a draw*

Proof. This is a result of Zermelo's Theorem [82]. 2-Player *k*-ago is played on a finite board, each player takes turns alternating moves and no information is hidden from the opposing player. □

Lemma 3.3.16. *Consider an $n \times n$ board B in which player 1 is the Blue player and player 2 is the Red player. Let p denote an extra, arbitrarily placed blue piece. Piece p is never a disadvantage for the Blue player.*

Proof. We consider 2 cases. The first case is that *p* is part of the winning strategy for player 1. Thus, on the player 1 turn where *p* should be played, player 1 then places a piece elsewhere. However, if *p* is not part of the winning strategy, *p* can simply be ignored. □

Lemma 3.3.17. *Player 2 does not have a winning strategy.*

Proof. We use a *strategy-stealing* argument. Assume that player 2 has a winning strategy. Player 1 then arbitrarily places a blue piece, effectively making player 2 the first player. Player 1 then 'steals' player 2's winning strategy. From Lemma 3.3.16, the arbitrary piece played by player 1 is never a disadvantage. Thus, player 1 and player 2 both have a winning strategy - contradiction. □

Lemma 3.3.18. *2-Player k-ago on an $n \times n$ board, where n is odd, is never a draw.*

Proof. With n being odd, the total number of pieces playable n^2 must also be odd. It follows that the number of pieces played by player 1 and player 2 will not be equal. □

Theorem 3.3.19. *2-Player k-ago on an $n \times n$ board, where n is odd, is a first player win.*

Proof. From Lemma 3.3.17, we get that player 2 does not have a winning strategy. From Lemma 3.3.18, any 2-Player k-ago game played on a board with odd dimension will never end in a draw. Thus, (1) must hold from Lemma 3.3.15: player 1 must have a winning strategy. □

While these results generalize for any k-ago games played on boards of odd dimension, here we show an instance where 2-Player k-ago played on boards of size 4×4 is guaranteed to be a draw when played optimally.

Lemma 3.3.20. *$m,n,4$-games are a draw for $m,n \leq 5$ [92].*

Theorem 3.3.21. *2-Player k-ago played on a 4×4 board is a guaranteed draw for $k \geq 3$.*

Proof. Consider the more trivial case, when $k \geq 4$. As no k-in-a-row matchings can occur with $k > 4$ and with an even number of pieces playable, the game is a guaranteed draw. Furthermore, when $k = 4$, this is also fairly trivial. From Lemma 3.3.17, we get that player 2 does not have a winning strategy. From Lemma 3.3.20, we get that $4,4,4$-games are a guaranteed draw. As a result, when played optimally, player 2 plays for the draw by preventing player 1 from achieving a 4-in-a-row matching.

The less trivial case is when $k = 3$. It is known that $m,n,3$-games are a first player win for $m \geq 3, n \geq 4$, which implies that a drawing strategy for player 2 *must* include one 3-in-a-row matching, as player 1 can *always* achieve a 3-in-a-row matching. We present player 2's strategy as follows. Note that due to the boards symmetry, the strategy can be rotated to accommodate for other first move starting positions.

Case 1: Player 1 plays anywhere along the edge. Player 2 places a central piece in $c_{(2,1)}$. Regardless of where player 1 then plays, player 2 can then force a matching within 2 moves. Consider an arbitrary move by player 1. If the 2 blue pieces lie on opposite edges of the board or do

not threaten a 3-in-a-row, player 2 places a red piece in any other central location where the line segment formed between the 2 red pieces passes through 2 empty cells. Otherwise, consider the line segment formed between the 2 blue pieces. Player 2 places a red piece on this line segment if it passes through a central cell, otherwise in a location such that the line segment formed between the 2 red pieces is parallel to the line segment formed between the 2 blue pieces.

Case 2: This case is slightly trickier. Player 1 plays a central blue piece in $c_{(2,1)}$. Player 2 plays a red piece in $c_{(3,1)}$. Player 1 is then forced to play a blue piece in either $c_{(3,0)}$ or $c_{(3,2)}$, otherwise player 2 can force a 3-in-a-row matching in row 3. In either case, player 2 blocks off the 3-in-a-row of blue pieces formed. This gives player 1 a playable location which guarantees a matching for player 1. Assume player 1 plays at some arbitrary location, which may or may not be the location that guarantees a 3-in-a-row of blue pieces. Player 2 essentially ignores the player 1 move and sets up their own forced 3-in-a-row matching either in $c_{(1,1)}$ or $c_{(2,2)}$.

In both cases, player 2 is then guaranteed *one* matching along with player 1, and the resulting empty cells no longer admit 3-in-a-row matchings for either player. Since each player has the same number of matchings, the number of blue and red pieces is equal, resulting in a draw. □

3.3.7 Approximation Algorithm

This section focuses on an optimization variation of TAPG k-ago known as Maximal Gray k-ago. The mechanics remain the same as the original k-ago games; however, the goal now is to maximize the number of pieces that are flipped to gray. We show that while NP-complete in the general case, each k admits a k-approximation algorithm.

Definition 3.3.4 (Maximal Gray k-ago)**.** *The maximal gray k-ago puzzle is defined as follows:*

INPUT: *An $n \times n$ board partially filled with black pieces and an integer $2 \leq k \leq n$*

OUTPUT: *An integer* $0 \leq P \leq n^2$*, where P is the maximum number of pieces that can be turned to gray*

Corollary 3.3.22. *Maximal Gray k-ago is NP-complete for* $k \geq 3$.

Proof. Follows from Lemma 3.3.9 and Theorem 3.3.11. A Hamiltonian cycle exists if the entire board can be turned gray, i.e the maximum $P = n^2$. □

Theorem 3.3.23. *Maximal Gray k-ago for* $k \geq 3$ *admits a k-approximation algorithm.*

Proof. For a 2-dimensional table of size $n \times n$ with entry $[r,c]$ referring to the row and column index, respectively, we refer to a 'top left to bottom right' pass as starting from a cell in row $0 \cup$ column 0, traversing to cell $[r+1, c-1]$ from current cell $[r,c]$, and ending in a cell in column $n-1$ $\cup$ row $n-1$. Similarly, we refer to a 'bottom left to top right' pass as starting from a cell in column $0 \cup$ row $n-1$, traversing to cell $[r+1, c+1]$ from current cell $[r,c]$, and ending in a cell in row 0 $\cup$ column $n-1$.

The algorithm uses Algorithm 15 as a subroutine in 4 passes:

1. For each row, from left to right
2. For each column, from top to bottom
3. For each diagonal, from top left to bottom right
4. For each diagonal, from bottom left to top right

Intuitively, we start with as many horizontal matchings as possible, then move on to all vertical matchings, and so on. The direction of each pass is important, as we consider all possible directions in which matchings can occur. Thus, by the end of the 4 passes, although we may not encounter an optimal solution, the solution found will be maximal in the sense that it will not admit any additional matchings.

Let P_{OPT} denote the optimal solution and P_A denote the output from the approximation algorithm. Consider a horizontal sequence of $k-1$ black pieces followed by an empty location.

Assume that in the optimal solution, the $k-1$ black pieces are matched vertically to a different empty location, and the empty location is matched vertically to other adjacent black pieces. Thus, for this incorrect matching, $(k-1)(k)$ pieces are left unflipped (shown in Figure 3.28). By tiling this pattern onto an $n \times n$ board, we get the total number of pieces flipped to be:

$$P_A = k \cdot \frac{n}{k} \cdot \frac{n}{k} = \frac{n^2}{k} \tag{3.1}$$

with the optimal solution as:

$$P_{OPT} = k^2 \cdot \frac{n}{k} \cdot \frac{n}{k} = n^2 \tag{3.2}$$

It follows that:

$$\alpha = \frac{P_{OPT}}{P_A} = \frac{n^2}{\frac{n^2}{k}} = k \tag{3.3}$$

□

3.3.8 Conclusion

K-in-a-row mechanics, popular in 1-player puzzle games, serve as the foundation of our examination on the board game *k*-ago. The paper expounds on two variations - Fill in the Board k-ago (FITB *k*-ago) and Turn All Pieces Gray *k*-ago (TAPG *k*-ago). For restricted board configurations, polynomial-time algorithms are developed, while generalized versions of the game are shown to be NP-complete for $k \geq 3$. Furthermore, we show that a 2-player version of *k*-ago is a guaranteed first player win when playing on boards of odd dimension, whereas a guaranteed draw when playing on boards of size 4×4. We finally conclude by considering an optimization variation of *k*-ago in which the number of gray pieces is maximized. Under these assumptions, we show that there exists a *k*-approximation algorithm for each *k*-ago with $k \geq 3$. These results highlight the computational depth of *k*-in-a-row mechanics and contribute to the broader understanding of complexity in combinatorial games.

While our work extensively covers several variations of k-ago, there are still several interesting questions to consider:

- What is the complexity of deciding a first player win given a board configuration in the 2-Player variation of k-ago? We conjecture that for $k \geq 5$, the problem is PSPACE-complete following similar logic to other 5-in-a-row game complexity proofs.

- We show that while for any odd dimension board 2-Player k-ago is a first player win, boards of size 4×4 is a guaranteed draw. Does this extend to any arbitrarily sized board of even dimension?

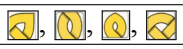
- Does the inclusion of multi-colored pieces affect the results in 1-Player k-ago? What about if parts of the board could rotate? Or if pieces were never played, but matchings occurred through rotation only: is turning all pieces gray still NP-complete?

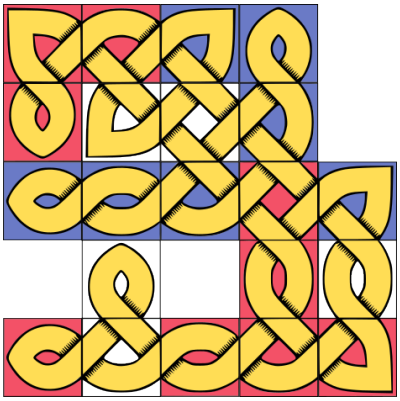
- Is there some $w \times n$ board, with $w = O(1)$, in which the game is NP-complete for any $w_2 \geq w$ but polynomial otherwise?

- Does there exist a FPT algorithm if we fix the board size? What about with the number of moves?

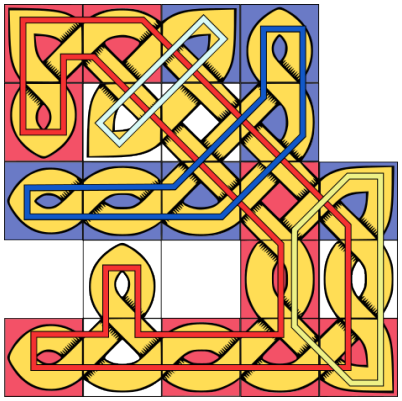
- Is a better approximation factor α for the Maximal Gray k-ago puzzle achieveable? As a first step, a better factor may be attained simply by dynamically ordering the direction of the 'passes' of the algorithm to better fit the given board, rather than keeping a static ordering.

Table 3.1: Summary of results for different Celtic! variations. All reductions use a $O(n) \times O(n)$ size board.

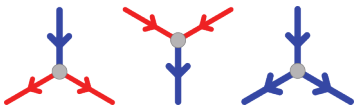
Version	Player Pieces	Board Pieces	Complexity	Theorem
0-Player		all	P-complete	Thm. 3.2.12
1-Player		all	$O(n^2)$	Thm. 3.2.1
		all	$O(n^2)$	Thm. 3.2.2
		all	$O(n^2)$	Thm. 3.2.3
			NP-complete	Thm. 3.2.6
			NP-complete	Thm. 3.2.7
		all	OPEN	-
		all	OPEN	-
			NP-complete	Thm. 3.2.8
2-Player			PSPACE-complete	Thm. 3.2.11



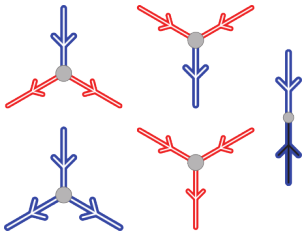
(a) Completed Game



(b) Completed Knots

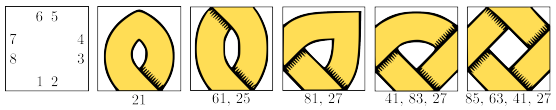


(c) Bounded DCL



(d) Bounded 2CL

Figure 3.1: (a) Final Celtic! board where all knots are closed and no valid moves exist. (b) Celtic! board displaying completed knots. The knot score for Red (R) and Blue (B) players based on the knot. Red knot: R(9) - B(2), Blue knot: R(1) - B(5), Yellow knot: R(4) - B(1), and White knot: R(0) - B(1). Given these knots, the best score for Red is 9 and 5 points for Blue. (c) Gadgets for 0-player non-planar Bounded Deterministic CL. (d) Gadgets for Bounded 2-player CL.



(a) Piece paths

(b) Invalid vs. Valid

Figure 3.2: (a) The 5 types of pieces in the game as well as their path connections (in this orientation). Each player (red and blue) has two copies of each type and there is a shared set of white-backed ones. (b) Invalid vs. valid placement.

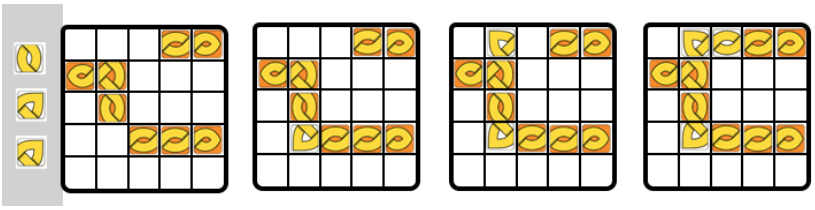
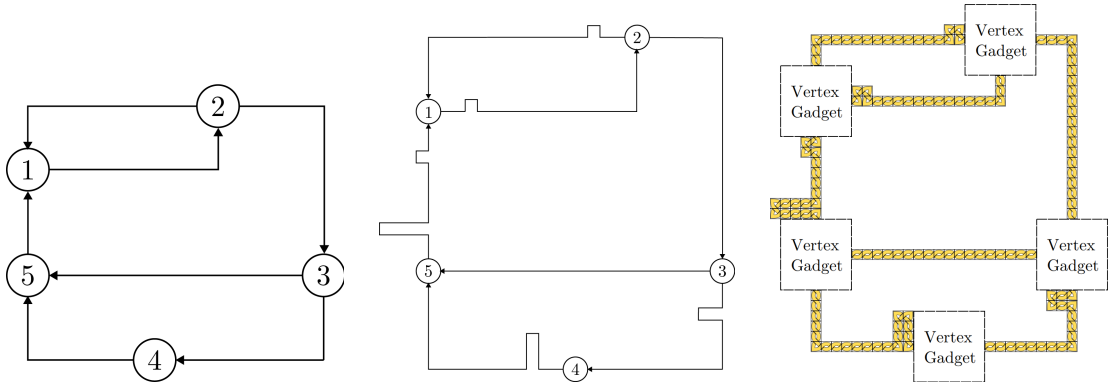
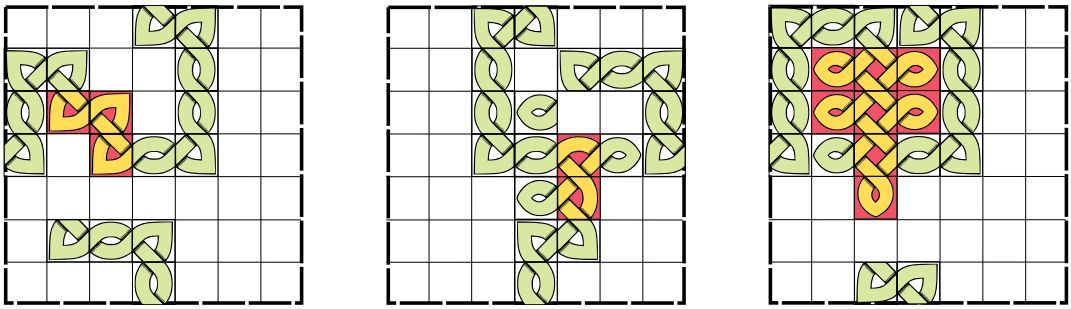


Figure 3.3: Example of an initial game board configuration for generalized 1-Player Celtic! and a sequence of valid moves that form a closed knot of length $|K| = 11$ in $k = 3$ moves given 2 pieces and 1 piece.



(a) (b) (c)

Figure 3.4: The 3-step process for creating the initial board for the NP-complete reductions. (a) Example directed, planar graph with max degree 3. (b) The graph is scaled by a factor of $|V|^2$, then transformed to have edges of roughly equal lengths using the free space around each vertex. The edges within 3 units of the vertex are left untouched. (c) The initial board created for the graph. The vertex gadget changes per reduction depending on the pieces used.



(a) (b) (c)

Figure 3.5: Vertex gadgets for the (a) piece selecting the western edge, (b) selecting the southern edge, and (c) and selecting the western edge.

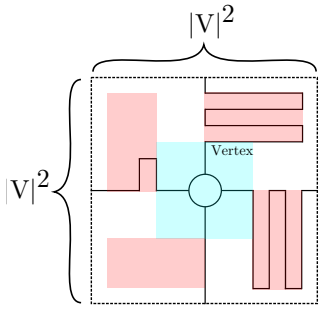


Figure 3.6: Pumping the edge lengths of a degree 4 vertex in a graph that has been scaled by a factor of $|V|^2$. Each edge can be 'pumped' using the free space in the area shaded red which has area $\frac{|V|^2-2}{2} \times \frac{|V|^2-4}{2}$. The area shaded aqua is left untouched to be replaced by vertex gadgets in later reductions.

Table 3.2: Summary of results for 1-Player TAPG/FITB *k*-ago.

TAPG/FITB *k*-ago				
Board Size	**k**	**Complexity**	**TAPG**	**FITB**
$1 \times n$	any	$\min(O(k \cdot n), O(n^2))$	Thm. 3.3.3	Thm. 3.3.4
$w \times n$	$> w$	$\min(O(w \cdot k \cdot n), O(n^3))$	Thm. 3.3.5	Thm. 3.3.6
$O(n) \times O(n)$	2	$O(n^2)$	Thm. 3.3.7	Thm. 3.3.8
$O(n) \times O(n)$	≥ 3	NP-complete	Thm. 3.3.11	Thm. 3.3.12

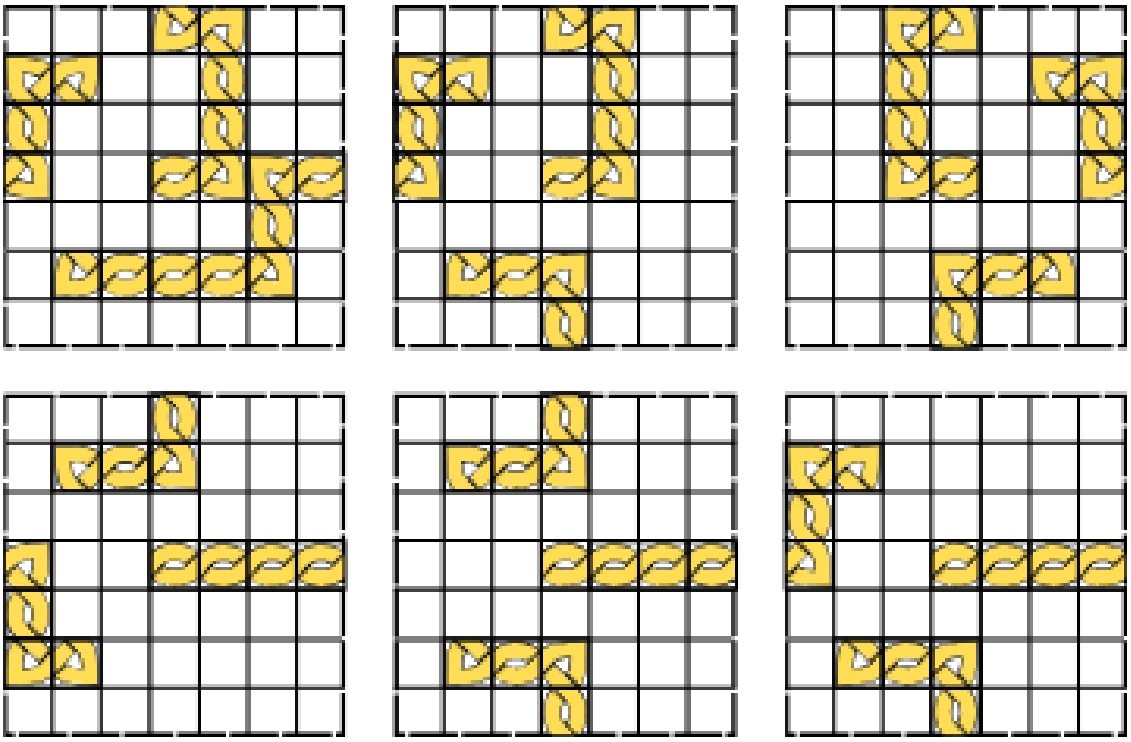
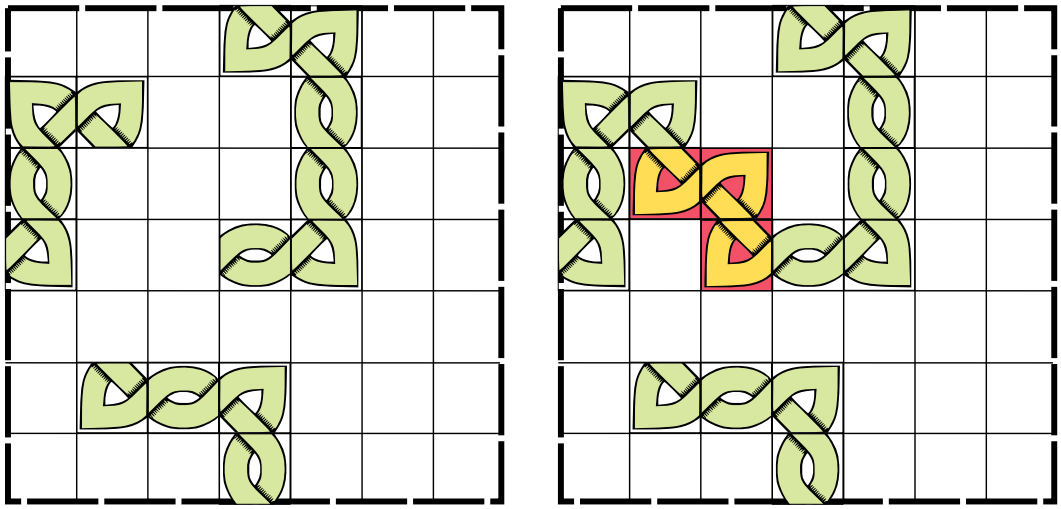


Figure 3.7: Vertex gadgets for generalized 1-Player Celtic! with only placing pieces: vertices with northern output edge and east/west input edges (top left), northern output edge and west/south input edges (top), northern output edge and east/south input edges (top right), eastern output edge and north/west input edges (bottom left), eastern output edge and north/south input edges (bottom), and eastern output edge and west/south input edges (bottom right). West and south output edges are achieved through 180 degree rotation of the east and north output gadgets, respectively. For vertices with 1 input edge and 2 output edges, the input edge is treated as the output edge in the vertex gadgets.

Table 3.3: Summary of results for 2-Player k-ago.

2-Player k-ago			
Board Size	**k**	**Result**	**Theorem**
Odd Dimension	any	First Player Win	Thm. 3.3.19
4×4	≥ 3	Draw	Thm. 3.3.21



(a) (b)

Figure 3.8: (a) A vertex gadget for the piece. (b) Selection of the western edge.

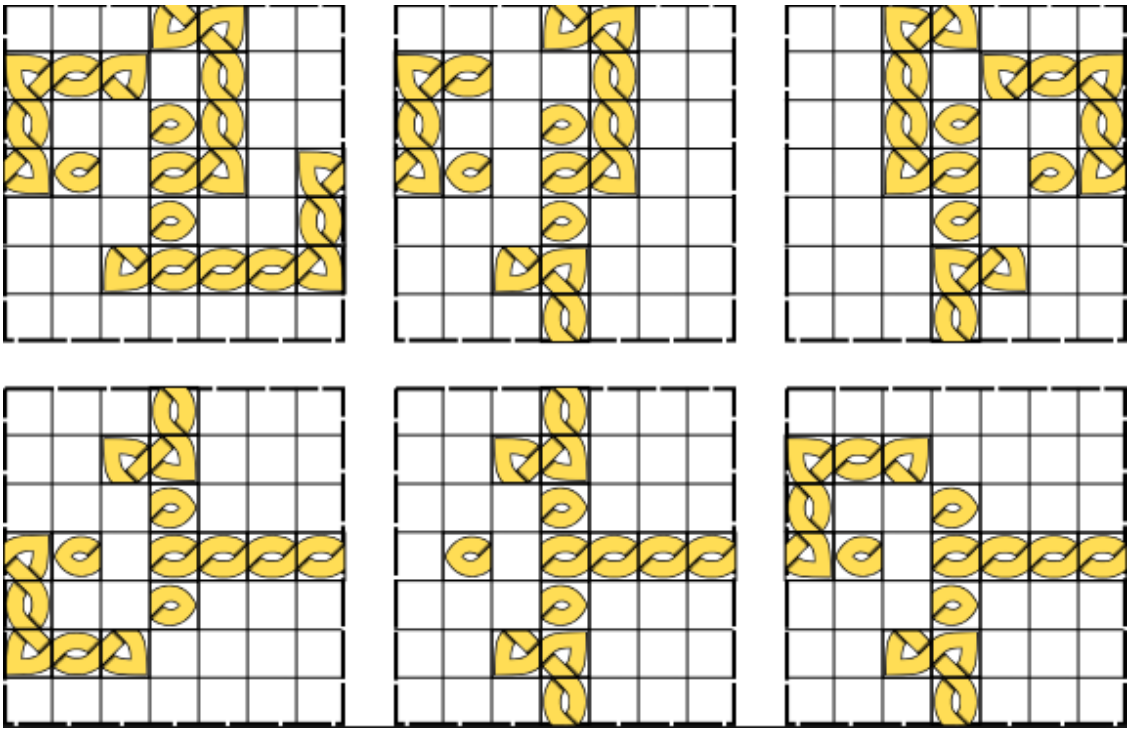
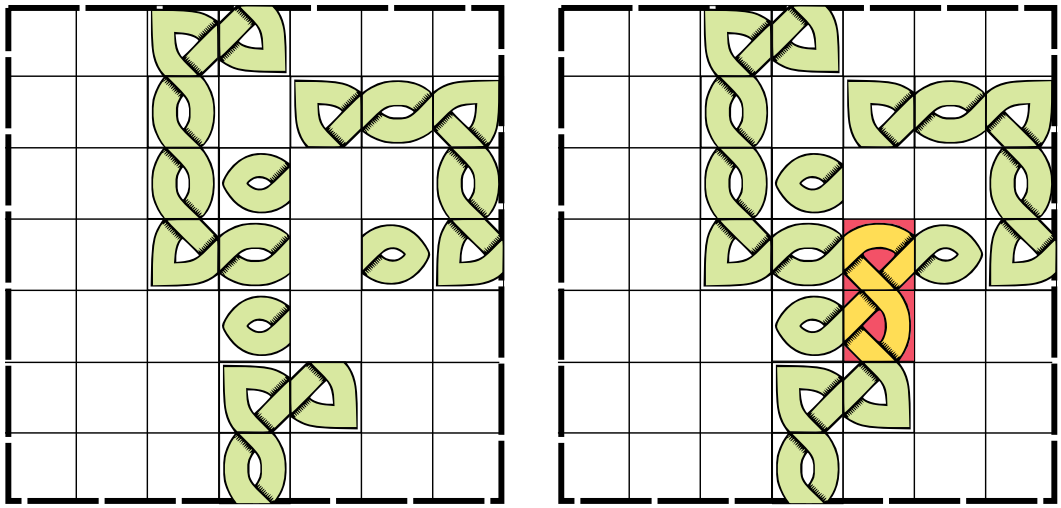


Figure 3.9: Vertex gadgets for generalized 1-Player Celtic! with only placing pieces: vertices with northern output edge and east/west input edges (top left), northern output edge and west/south input edges (top), northern output edge and east/south input edges (top right), eastern output edge and north/west input edges (bottom left), eastern output edge and north/south input edges (bottom), and eastern output edge and west/south input edges (bottom right). West and south output edges are achieved through 180 degree rotation of the east and north output gadgets, respectively. For vertices with 1 input edge and 2 output edges, the input edge is treated as the output edge in the vertex gadgets.



(a) (b)

Figure 3.10: (a) A vertex gadget for the piece. (b) Selection of the southern edge.

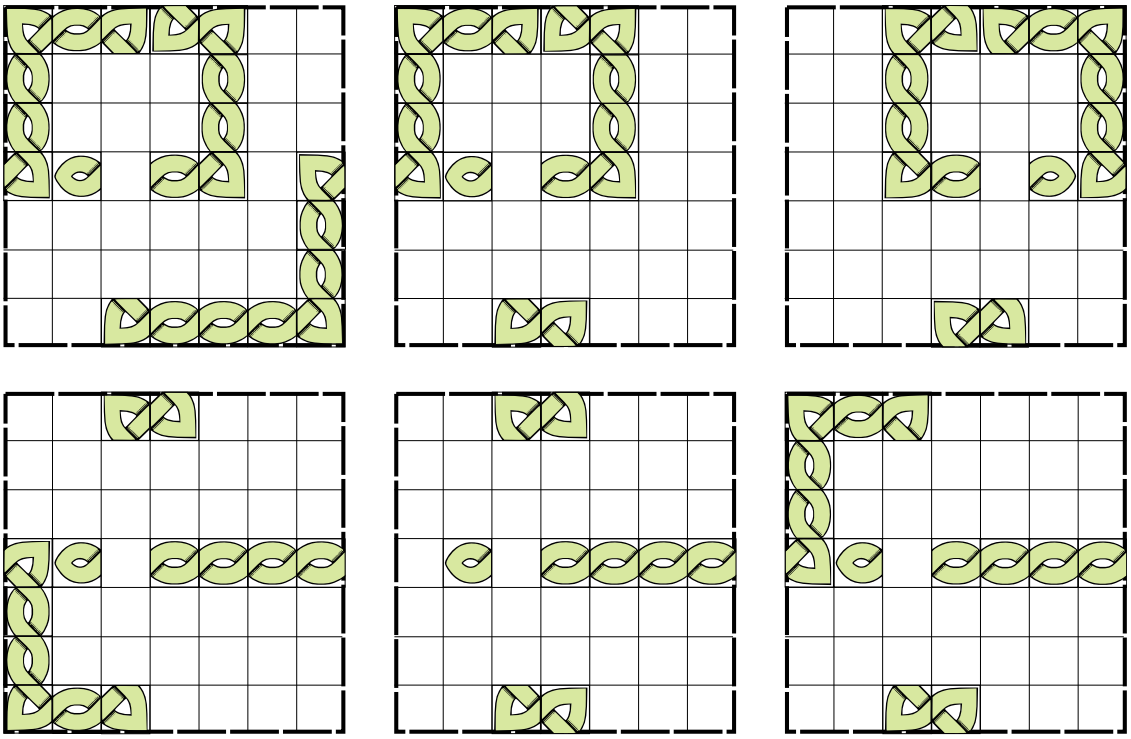
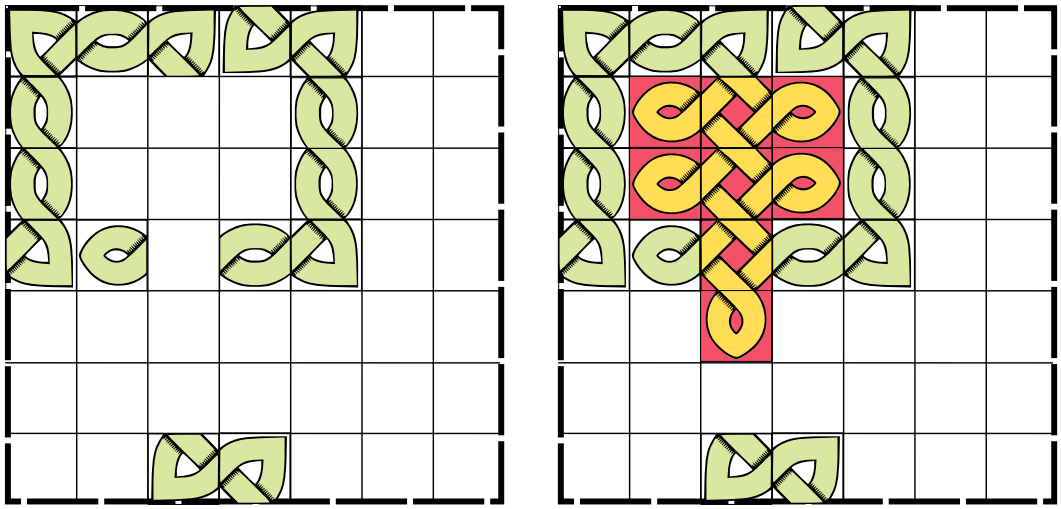


Figure 3.11: Vertex gadgets for generalized 1-Player Celtic! with only placing  and  pieces: vertices with northern output edge and east/west input edges (top left), northern output edge and west/south input edges (top), northern output edge and east/south input edges (top right), eastern output edge and north/west input edges (bottom left), eastern output edge and north/south input edges (bottom), and eastern output edge and west/south input edges (bottom right). West and south output edges are achieved through 180 degree rotation of the east and north output gadgets, respectively. For vertices with 1 input edge and 2 output edges, the input edge is treated as the output edge in the vertex gadgets.

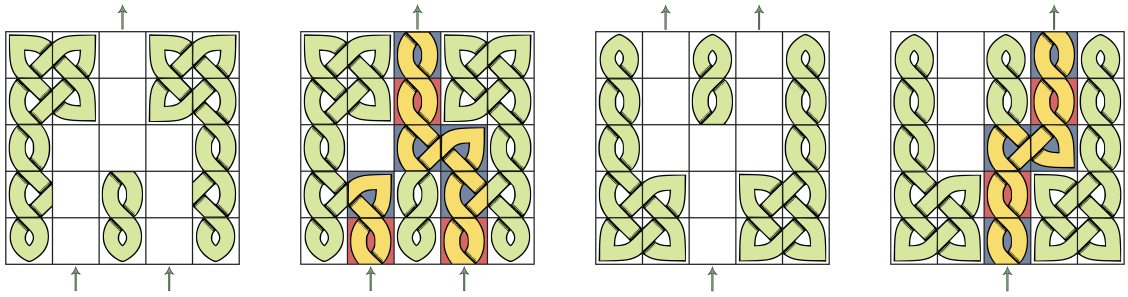


(a) (b)

Figure 3.12: (a) A vertex gadget for the and pieces. (b) Selection of the western edge.

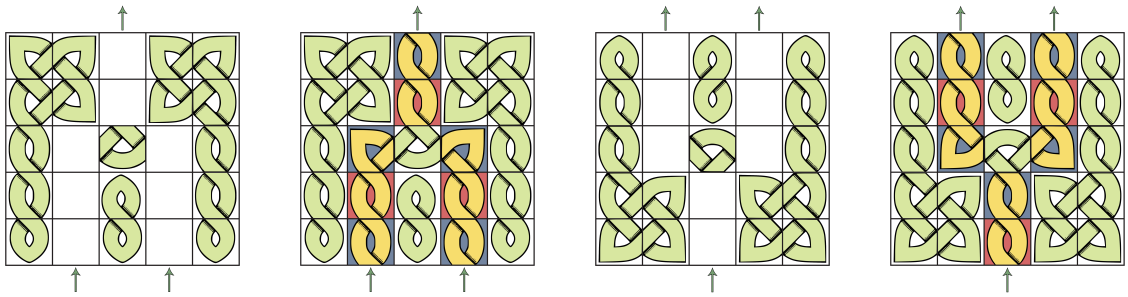
(a) Variable Gadget (b) Win Gadget (c) Win Example

Figure 3.13: (a) VARIABLE Gadget in the 2-player game representing the VARIABLE vertex in a Bounded 2CL. If Red player moves first, the knot is closed with , preventing Blue player from continuing an edge to other gadgets. If Blue player moves first, is placed, which restricts the red player to only continue the knot to other gadgets. (b) In the WINNING Gadget, only the Blue player is allowed to finally close the knot. Once the blue player closes the knot, the knot can no longer be extended. (c) Example moves in the WINNING gadget. The red player can only continue in the gadget, while blue player makes the last move.



(a) OR Gadget (b) OR Example (c) CHOICE (d) CHOICE Ex.

Figure 3.14: (a) OR Gadget in the 2-player game representing the OR vertex in a Bounded 2CL. For the player to continue with a valid knot, the player can connect a knot from the right side using the or from the left side using the . The player can then complete the knot from the opposite side as well. (b) An example of a sequence of moves in the OR gadget when the knot from the right side is continued. (c) CHOICE Gadget in the 2-player game representing the CHOICE vertex in a Bounded 2CL. For the player to continue a valid knot, the player can use either or . Depending on the piece, the knot continues right or left. No knot can be completed from the opposite side. (d) Moves in the CHOICE gadget continuing the knot to the right side.



(a) AND Gadget (b) AND Example (c) FANOUT (d) FANOUT Ex.

Figure 3.15: (a) AND Gadget in the 2-player game representing the AND vertex in a Bounded 2CL. For the player to continue with a valid knot, the player can use both and . If the player only connects a knot from one side, the knot will not be closed. Hence, the pieces in the knot will not count towards the players. (b) An example of moves in the AND gadget. To complete the knot the player has to connect from both sides. (c) FANOUT Gadget in the 2-player game representing the FANOUT vertex in a Bounded 2CL. The player can complete the knot by connecting on both sides using and . (d) An example of moves in the FANOUT gadget. The player is able to continue knots on both sides.

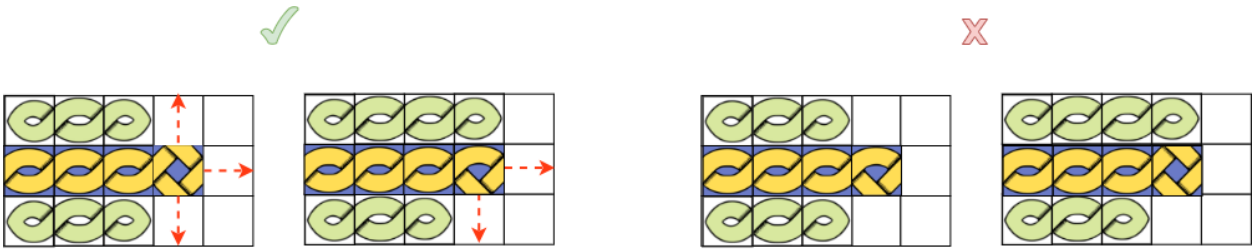


Figure 3.16: Valid (left) and invalid (right) moves. In the invalid moves, the left has an empty cell on the north side, so the player must place a piece. In the other example, there is an adjacent piece to the north with a closed side, thus the played piece needs a closed side on its north.

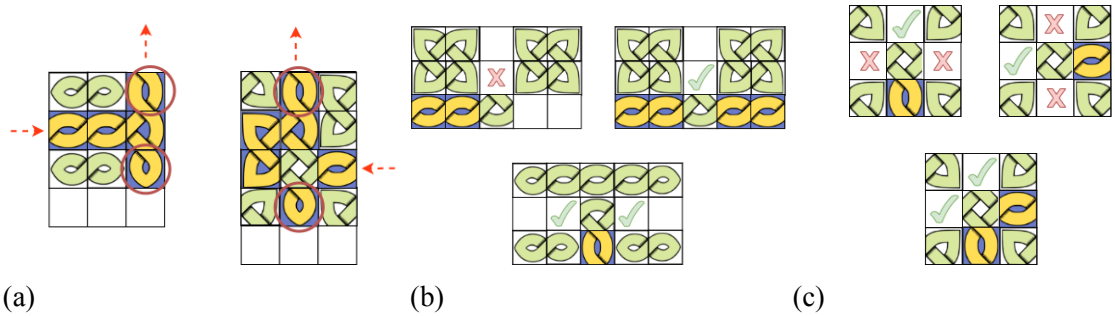


Figure 3.17: (a) An example of the Close South Movements rule. The red arrows show the direction that the player is placing the pieces. As it continues north, any movement south will result in a piece to close that path. (b) The conditions needed in order to continue through a piece. (c) The conditions necessary to continue through a piece.

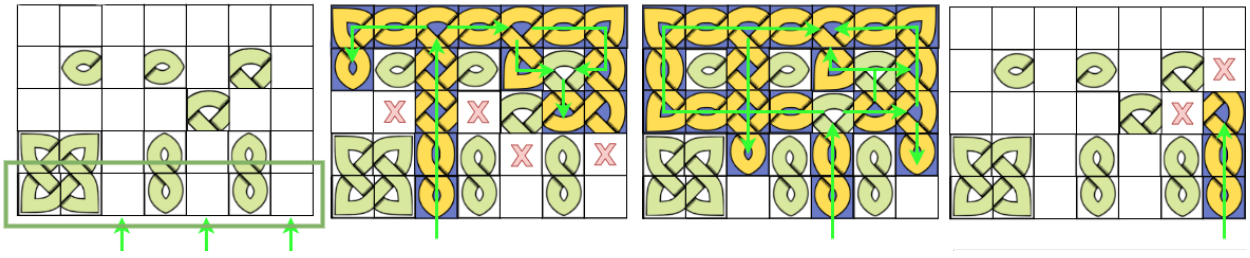


Figure 3.18: Example 0-player simulations of a board, depending on where the starter piece is placed. Here, the longest knot is created by placing it in the middle. If the goal is to get a knot ≥ 5, the player can place the starter piece in the left or middle starting location. The green arrows show the path the simulation takes and the red X's are the directions it can not continue through.

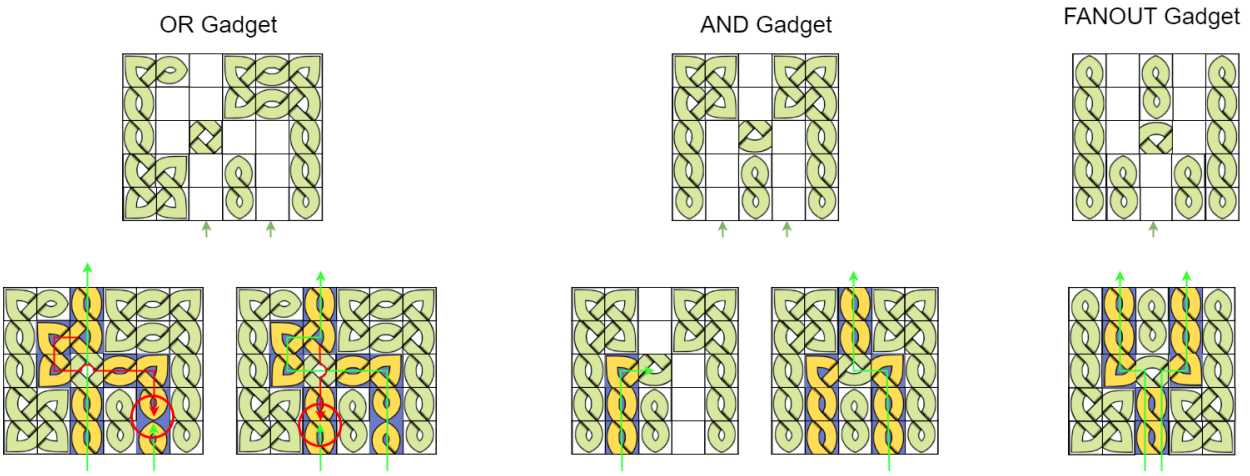


Figure 3.19: Examples of an OR, AND, and FANOUT gadget, and how they are traversed. The green arrows show inputs propagating inward, red arrows depict southward propagation getting closed. In the OR gadget, one input causes the other input path to close, and any other signal coming in will close. The AND requires both inputs in order to continue outwards. FANOUT splits into 2.

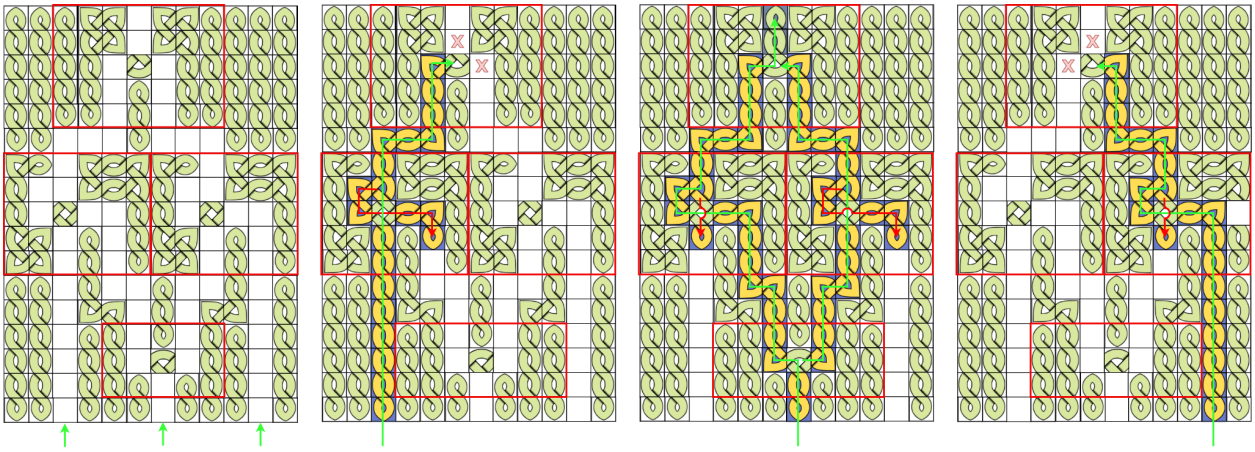


Figure 3.20: Example of the 0-Player Celtic! reduction from DCL. The red squares highlight the gadgets locations. On top we have an AND with inputs coming from the outputs of the middle OR gadgets. At the bottom, we can place our starter piece as input into one of the ORs, or we can input it into the FANOUT. Here, only the middle starting position will reach the opposite side of the board.

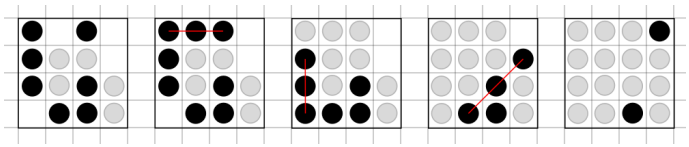


Figure 3.21: Example of a valid sequence of moves resulting in a win for 1-Player FITB 3-ago. Note that this sequence of moves does not result in a win for the TAPG variation, since 2 black pieces remain.

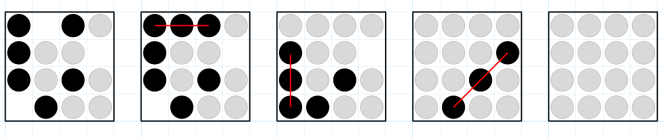
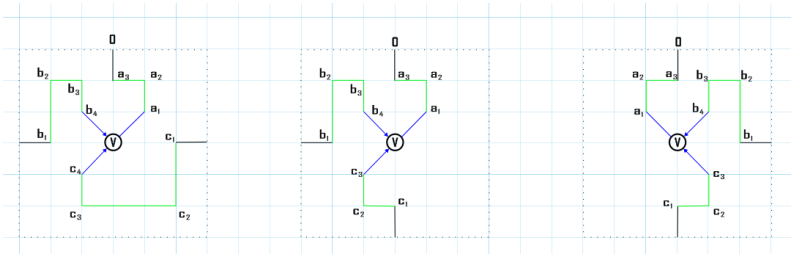


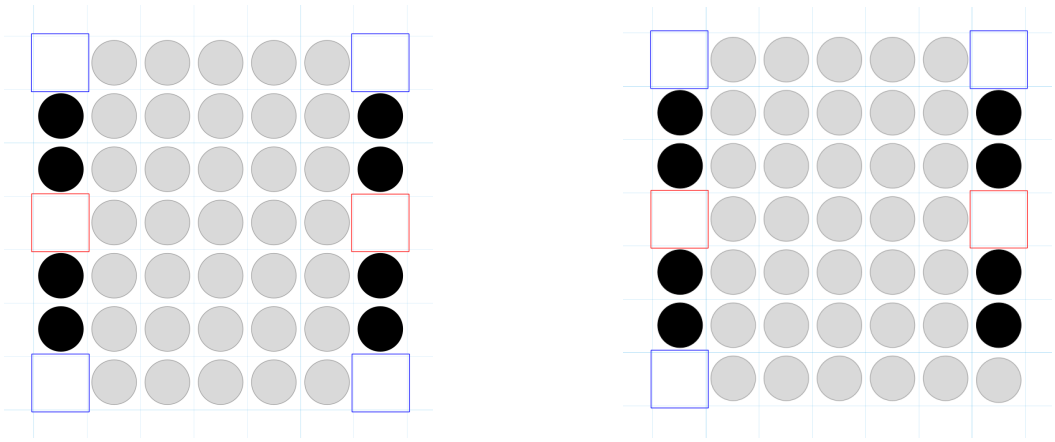
Figure 3.22: Example of a valid sequence of moves resulting in a win for 1-Player TAPG 3-ago. Note that this sequence of moves also results in a win for the FITB variation.



(a)

(b)

Figure 3.23: Edge rearrangement with northern output (a) and eastern output (b). Each a_j, b_k, c_l correspond to the integer coordinates of the new bends for the outgoing and 2 incoming edges, respectively.



(a) Vertex Gadget with 2 outputs.

(b) Vertex Gadget with 1 output.

Figure 3.24: Vertex gadgets, with the left red square representing the 'input' and the right red square representing the 'output' (however, the gadgets can be flipped/rotated as needed). An edge is 'chosen' in the red square by selecting the 2 black pieces from the other *non-chosen* wire to flip with. The blue squares connect to the respective edge wires. Note that in the case of only 1 output edge, one of the empty locations on the 'output' side is replaced with a gray piece.

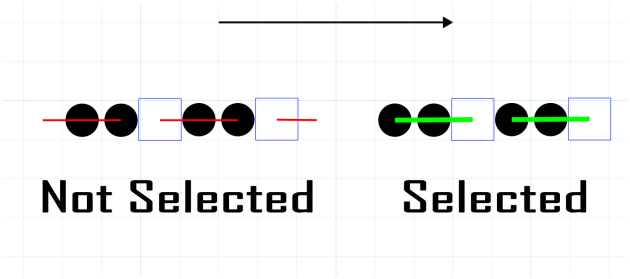
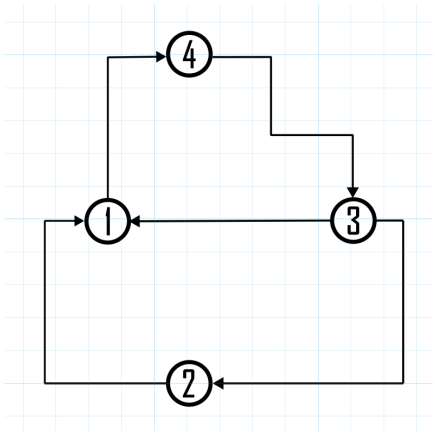
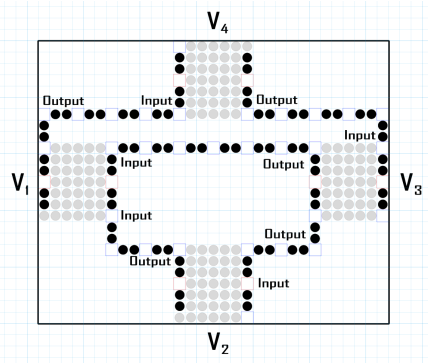


Figure 3.25: The wire for a given directed edge. The choice of the adjacent black pieces determines whether the corresponding edge was 'selected' or 'not selected'.



(a) Example graph G.



(b) The initial board configuration for G.

Figure 3.26: An example reduction from Hamiltonian cycle on a directed, rectilinearly embedded graph G to TAPG 3-ago. Empty cells without a bordering blue/red square are implied to be gray.

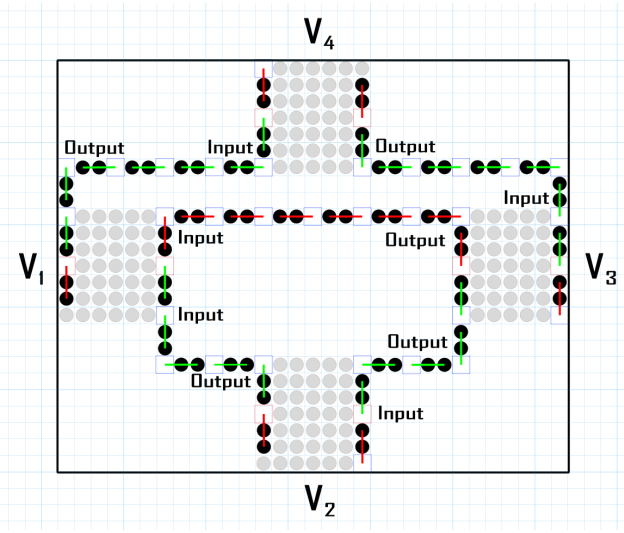


Figure 3.27: The corresponding sequence of moves to turn all pieces gray from Figure 3.26b. Green matchings denote the edge was selected, whereas red matchings denote the edge was not selected.

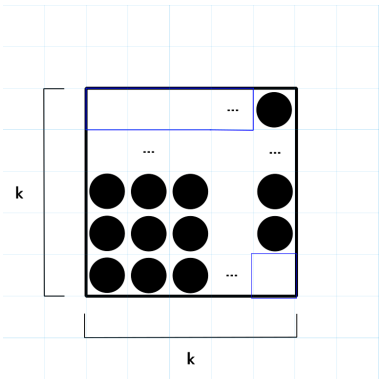


Figure 3.28: A pattern for which the approximation algorithm incorrectly matches the bottom k elements, leaving the rest of the $k \times k$ area unmatched.

Algorithm 15 TAPG k-ago with $1 \times n$ board

```
1:  // Iterate over entire array
2:  j ← 0
3:  while j < n do
4:      // Compare all elements in sliding window
5:      update ← True
6:      emptyCells ← 0
7:      for idx in j, ..., j + k do
8:          // If more than 1 empty cell in sliding window, flip not possible
9:          if D[idx] = −1 then
10:             if emptyCells > 0 then
11:                 update ← False
12:             emptyCells += 1
13:         // If there is a gray cell, flip not possible
14:         if D[idx] = 0 then
15:             update ← False
16:
17:     // If flip possible, turn all pieces to gray in the sliding window
18:     if update then
19:         for idx in j, ..., j + k do
20:             D[idx] ← 0
21:     j ← j + 1
22:
23: // Check that all pieces have turned gray
24: j ← 0
25: while j < n do
26:     if D[j]! = 0 then
27:         return False
28:     j ← j + 1
29: return True
```

REFERENCES

[1] R. M. Alaniz, J. Brunner, M. Coulombe, E. D. Demaine, Y. Diomidov, T. Gomez, E. Grizzell, R. Knobel, J. Lynch, A. Rodriguez, R. Schweller, and T. Wylie, *Complexity of reconfiguration in surface chemical reaction networks*, in Proc. of the 29th International Conference on DNA Computing and Molecular Programming, DNA'23, 2023. To appear.

[2] R. M. Alaniz, D. Caballero, S. C. Cirlos, T. Gomez, E. Grizzell, A. Rodriguez, R. Schweller, A. Tenorio, and T. Wylie, *Building squares with optimal state complexity in restricted active self-assembly*, Journal of Computer and System Sciences, 138 (2023), p. 103462.

[3] R. M. Alaniz, B. Fu, T. Gomez, E. Grizzell, A. Rodriguez, R. Schweller, and T. Wylie, *Reachability in restricted chemical reaction networks*, 2022.

[4] R. Anderson, A. Avila, B. Fu, T. Gomez, E. Grizzell, A. Massie, G. Mukhopadhyay, A. Salinas, R. Schweller, E. Tomai, and T. Wylie, *Computing threshold circuits with void reactions in step chemical reaction networks*, in 10th conference on Machines, Computations and Universality (MCU 2024), 2024.

[5] R. Anderson, B. Fu, A. Massie, G. Mukhopadhyay, A. Salinas, R. Schweller, E. Tomai, and T. Wylie, *Computing threshold circuits with bimolecular void reactions in step chemical reaction networks*, in International Conference on Unconventional Computation and Natural Computation, Springer, 2024, pp. 253–268.

[6] D. Angluin, J. Aspnes, Z. Diamadi, M. J. Fischer, and R. Peralta, *Computation in networks of passively mobile finite-state sensors*, Distributed Computing, 18 (2006), p. 235–253.

[7] R. Aris, *Prolegomena to the rational analysis of systems of chemical reactions*, Archive for Rational Mechanics and Analysis, 19 (1965), pp. 81–99.

[8] ——, *Prolegomena to the rational analysis of systems of chemical reactions ii. some addenda*, Archive for Rational Mechanics and Analysis, 27 (1968), pp. 356–364.

[9] K. Barth, D. Furcy, S. M. Summers, and P. Totzke, *Scaled tree fractals do not strictly self-assemble*, in Unconventional Computation and Natural Computation: 13th International Conference, UCNC 2014, London, ON, Canada, July 14-18, 2014, Proceedings 13, Springer, 2014, pp. 27–39.

[10] F. Becker, D. Hader, and M. J. Patitz, *Strict self-assembly of discrete self-similar fractals in the abstract tile-assembly model*, in Proceedings of the ACM-SIAM Symposium on Discrete Algorithms, 2025.

[11] R. L. Berger, *The undecidability of the domino problem*, Memoirs of the American Mathematical Society, (1966), pp. 0–0.

[12] M. Blondin, M. Englert, A. Finkel, S. Göller, C. Haase, R. Lazić, P. Mckenzie, and P. Totzke, *The reachability problem for two-dimensional vector addition systems with states*, Journal of the ACM (JACM), 68 (2021), pp. 1–43.

[13] J. Bosboom, C. Chen, L. Chung, S. Compton, M. Coulombe, E. D. Demaine, M. L. Demaine, I. T. F. A. Filho, D. Hendrickson, A. Hesterberg, C. Hsu, W. Hu, O. Korten, Z. Luo, and L. Zhang, *Edge matching with inequalities, triangles, unknown shape, and two players*, 2020. arXiv:2002.03887.

[14] J. Bosboom, E. D. Demaine, M. L. Demaine, A. Hesterberg, P. Manurangsi, and A. Yodpinyanee, *Even* $1 \times n$ *edge-matching and jigsaw puzzles are really hard*, 2016. arXiv:1701.00146.

[15] T. Brailovskaya, G. Gowri, S. Yu, and E. Winfree, *Reversible computation using swap reactions on a surface*, in Proc. of the International Conference on DNA Computing and Molecular Programming, DNA'19, Springer, 2019, pp. 174–196.

[16] C. Browne, *Connection Games: Variations on a Theme*, A K Peters, Ltd., 2005.

[17] ——, *Celtic!* [Board Game], 2009.

[18] A. A. Cantu, A. Luchsinger, R. Schweller, and T. Wylie, *Signal Passing Self-Assembly Simulates Tile Automata*, in 31st International Symposium on Algorithms and Computation (ISAAC 2020), Y. Cao, S.-W. Cheng, and M. Li, eds., vol. 181 of Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl, Germany, 2020, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, pp. 53:1–53:17.

[19] ——, *Signal Passing Self-Assembly Simulates Tile Automata*, in 31st International Symposium on Algorithms and Computation, vol. 181 of ISAAC'20, 2020, pp. 53:1–53:17.

[20] M. T. Carroll and S. T. Dougherty, *Tic-tac-toe on a finite plane*, Mathematics Magazine, 77 (2004), pp. 260–274.

[21] A. Case, J. H. Lutz, and D. M. Stull, *Reachability problems for continuous chemical reaction networks*, Natural Computing, 17 (2018), pp. 223–230.

[22] C. Chalk, A. Luchsinger, E. Martinez, R. Schweller, A. Winslow, and T. Wylie, *Freezing simulates non-freezing tile automata*, in DNA Computing and Molecular Programming: 24th International Conference, DNA 24, Jinan, China, October 8–12, 2018, Proceedings 24, Springer, 2018, pp. 155–172.

[23] C. T. Chalk, D. A. Fernandez, A. Huerta, M. A. Maldonado, R. T. Schweller, and L. Sweet, *Strict self-assembly of fractals using multiple hands*, Algorithmica, 76 (2016), p. 195–224.

[24] G. Chatterjee, N. Dalchau, R. A. Muscat, A. Phillips, and G. Seelig, *A spatially localized architecture for fast and modular DNA computing*, Nature nanotechnology, 12 (2017), pp. 920–927.

[25] H.-L. Chen, D. Doty, and D. Soloveichik, *Deterministic function computation with chemical reaction networks*, Natural computing, 13 (2014), pp. 517–534.

[26] C. T. Chou, *Chemical reaction networks for computing logarithm*, Synthetic Biology, 2 (2017), p. ysx002.

[27] S. Clamons, L. Qian, and E. Winfree, *Programming and simulating chemical reaction networks on a surface*, Journal of the Royal Society Interface, 17 (2020), p. 20190790.

[28] M. Cook et al., *Universality in elementary cellular automata*, Complex systems, 15 (2004), pp. 1–40.

[29] M. Cook, D. Soloveichik, E. Winfree, and J. Bruck, *Programmability of Chemical Reaction Networks*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 543–584.

[30] M. Cygan, F. V. Fomin, Ł. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk, and S. Saurabh, *Parameterized algorithms*, vol. 5, Springer, 2015.

[31] W. Czerwiński, S. Lasota, R. Lazić, J. Leroux, and F. Mazowiecki, *The reachability problem for Petri nets is not elementary*, Journal of the ACM (JACM), 68 (2020), pp. 1–28.

[32] W. Czerwiński, S. Lasota, and Ł. Orlikowski, *Improved Lower Bounds for Reachability in Vector Addition Systems*, in 48th International Colloquium on Automata, Languages, and Programming (ICALP 2021), vol. 198 of Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl, Germany, 2021, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, pp. 128:1–128:15.

[33] W. Czerwiński and Ł. Orlikowski, *Reachability in vector addition systems is Ackermann-complete*, in 62nd Annual Symposium on Foundations of Computer Science, FOCS'21, IEEE, 2021, pp. 1229–1240.

[34] F. Dannenberg, M. Kwiatkowska, C. Thachuk, and A. J. Turberfield, *DNA walker circuits: computational potential, design, and verification*, Natural Computing, 14 (2015), pp. 195–211.

[35] E. D. Demaine and M. L. Demaine, *Jigsaw puzzles, edge matching, and polyomino packing: Connections and complexity*, Graphs and Combinatorics, 23 (2007), pp. 195–208.

[36] M. Ebbesen, P. Fischer, and C. Witt, *Edge-matching problems with rotations*, 2017. arXiv:1703.09421.

[37] S. Edelkamp and P. Kissmann, *On the complexity of bdds for state space search: A case study in connect four*, Proceedings of the AAAI Conference on Artificial Intelligence, 25 (2011), pp. 18–23.

[38] C. G. Evans, *Crystals that count! Physical principles and experimental investigations of DNA tile self-assembly*, PhD thesis, California Institute of Technology, 2014.

[39] D. Furcy and S. M. Summers, *Scaled pier fractals do not strictly self-assemble*, Natural Computing, 16 (2017), pp. 317–338.

[40] D. T. Gillespie, *Stochastic simulation of chemical kinetics*, Annu. Rev. Phys. Chem., 58 (2007), pp. 35–55.

[41] E. Goles, D. Maldonado, P. Montealegre, and M. Ríos-Wilson, *On the complexity of asynchronous freezing cellular automata*, Information and Computation, 281 (2021), p. 104764.

[42] E. Goles, N. Ollinger, and G. Theyssier, *Introducing freezing cellular automata*, in Cellular Automata and Discrete Complex Systems, 21st International Workshop (AUTOMATA 2015), vol. 24, 2015, pp. 65–73.

[43] M. H. T. Hack, *Decidability questions for Petri Nets.*, PhD thesis, Massachusetts Institute of Technology, 1976.

[44] A. Hamilton, G. T. Nguyen, and M. Roughan, *Counting candy crush configurations*, Discrete Applied Mathematics, 295 (2021), pp. 47–56.

[45] F. W. Hardesty and I. W. Schenck, *Competitive road building and travel game*. US grant 3309092A, 1967.

[46] R. A. Hearn and E. D. Demaine, *Games, puzzles, and computation*, CRC Press, 2009.

[47] J. Hendricks, M. Olsen, M. J. Patitz, T. A. Rogers, and H. Thomas, *Hierarchical self-assembly of fractals with signal-passing tiles*, Natural computing, 17 (2018), pp. 47–65.

[48] J. Hendricks and J. Opseth, *Self-assembly of 4-sided fractals in the two-handed tile assembly model*, in Unconventional Computation and Natural Computation, M. J. Patitz and M. Stannett, eds., Cham, 2017, Springer International Publishing, pp. 113–128.

[49] J. Hendricks, J. Opseth, M. J. Patitz, and S. M. Summers, *Hierarchical growth is necessary and (sometimes) sufficient to self-assemble discrete self-similar fractals*, Natural computing, 13 (2020), pp. 357–374.

[50] J. Hopcroft and J.-J. Pansiot, *On the reachability problem for 5-dimensional vector addition systems*, Theoretical Computer Science, 8 (1979), pp. 135–159.

[51] M. Y. Hsieh and S.-C. Tsai, *On the fairness and complexity of generalized k-in-a-row games*, Theoretical Computer Science, 385 (2007), pp. 88–100.

[52] R. M. Karp and R. E. Miller, *Parallel program schemata*, Journal of Computer and System Sciences, 3 (1969), pp. 147–195.

[53] R. Knobel, A. Salinas, R. Schweller, and T. Wylie, *Building discrete self-similar fractals in seeded tile automata*, CCCG, 2024.

[54] U. Koppenhagen and E. W. Mayr, *Optimal algorithms for the coverability, the subword, the containment, and the equivalence problems for commutative semigroups*, Information and Computation, 158 (2000), pp. 98–124.

[55] J. Leroux, *The reachability problem for Petri nets is not primitive recursive*, in 62nd Annual Symposium on Foundations of Computer Science, FOCS'21, IEEE, 2021.

[56] J. Leroux and S. Schmitz, *Reachability in vector addition systems is primitive-recursive in fixed dimension*, in 2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), IEEE, 2019, pp. 1–13.

[57] R. J. Lipton, *The reachability problem requires exponential space*, Tech. Rep. 62, Yale University, 1976.

[58] Y. Liu, A. Morgana, and B. Simeone, *A linear algorithm for 2-bend embeddings of planar graphs in the two-dimensional grid*, Discret. Appl. Math., 81 (1998), pp. 69–91.

[59] X. Ma and F. Lombardi, *Synthesis of tile sets for dna self-assembly*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 27 (2008), pp. 963–967.

[60] P. A. MacMahon, *New mathematical pastimes*, Cambridge, University Press, 1921.

[61] E. W. Mayr, *An algorithm for the general Petri net reachability problem*, in Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing, STOC '81, New York, NY, USA, 1981, Association for Computing Machinery, p. 238–246.

[62] E. W. Mayr and A. R. Meyer, *The complexity of the word problems for commutative semigroups and polynomial ideals*, Advances in mathematics, 46 (1982), pp. 305–329.

[63] T. McMurchie, *Squiggle Game*. US grant 4180271A, 1979.

[64] ——, *Tsuro*. Calliope Games, 2005.

[65] R. A. Muscat, K. Strauss, L. Ceze, and G. Seelig, *DNA-based molecular architecture with spatially localized components*, ACM SIGARCH Computer Architecture News, 41 (2013), pp. 177–188.

[66] J. F. Nash, *Some games and machines for playing them*, (1952).

[67] T. Neary and D. Woods, *P-completeness of cellular automaton rule 110*, in Automata, Languages and Programming: 33rd International Colloquium, ICALP 2006, Venice, Italy, July 10-14, 2006, Proceedings, Part I 33, Springer, 2006, pp. 132–143.

[68] J. E. Padilla, M. J. Patitz, R. T. Schweller, N. C. Seeman, S. M. Summers, and X. Zhong, *Asynchronous signal passing for tile self-assembly: Fuel efficient computation and efficient assembly of shapes*, International Journal of Foundations of Computer Science, 25 (2014), pp. 459–488.

[69] M. J. Patitz, *An introduction to tile-based self-assembly*, in Unconventional Computation and Natural Computation, J. Durand-Lose and N. Jonoska, eds., Berlin, Heidelberg, 2012, Springer Berlin Heidelberg, pp. 34–62.

[70] M. J. Patitz and S. M. Summers, *Self-assembly of discrete self-similar fractals*, Natural computing, 9 (2010), pp. 135–172.

[71] M. J. Patitz and S. M. Summers, *Self-assembly of discrete self-similar fractals*, Natural Computing, 9 (2010), pp. 135–172.

[72] C. A. Petri, *Kommunikation mit Automaten*, PhD thesis, Rheinisch-Westfälischen Institutes für Instrumentelle Mathematik an der Universität Bonn, 1962.

[73] J. Plesn´ik, *The np-completeness of the hamiltonian cycle problem in planar diagraphs with degree bound two*, Information Processing Letters, 8 (1979), pp. 199–201.

[74] J. Plesník, *The np-completeness of the hamiltonian cycle problem in planar diagraphs with degree bound two*, Information Processing Letters, 8 (1979), pp. 199–201.

[75] L. Qian and E. Winfree, *Parallel and scalable computation and spatial dynamics with DNA-based chemical reaction networks on a surface*, in DNA Computing and Molecular Programming: 20th International Conference, DNA 20, Kyoto, Japan, September 22-26, 2014. Proceedings, vol. 8727, Springer, 2014, p. 114.

[76] S. Reisch, *Gobang ist pspace-vollständig*, Acta Informatica, 13 (1980), pp. 59–66.

[77] P. W. K. Rothemund, *Theory and experiments in algorithmic self -assembly*, PhD thesis, 2001.

[78] P. W. K. Rothemund, N. Papadakis, and E. Winfree, *Algorithmic self-assembly of dna sierpinski triangles*, PLOS Biology, 2 (2004), p. null.

[79] G. S. Sacerdote and R. L. Tenney, *The decidability of the reachability problem for vector addition systems (preliminary version)*, in Proceedings of the ninth annual ACM symposium on Theory of computing, 1977, pp. 61–76.

[80] C. Schensted and C. Titus, *Kalico (Psyche-Paths)*. Funtastic, Kadon, 1968.

[81] S. Schmitz, *The complexity of reachability in vector addition systems*, ACM SigLog News, (2016).

[82] U. Schwalbe and P. Walker, *Zermelo and the early history of game theory*, Games and economic behavior, 34 (2001), pp. 123–137.

[83] D. Soloveichik, M. Cook, E. Winfree, and J. Bruck, *Computation with finite stochastic chemical reaction networks*, natural computing, 7 (2008), pp. 615–633.

[84] D. Soloveichik and E. Winfree, *Complexity of self-assembled shapes*, SIAM Journal on Computing, 36 (2007), pp. 1544–1569.

[85] G. Szlobodnyik and G. Szederkényi, *Polynomial time reachability analysis in discrete state chemical reaction networks obeying conservation laws*, MATCH-Communications in Mathematical and in Computer Chemistry, 89 (2023), pp. 175–196.

[86] G. Szlobodnyik, G. Szederkényi, and M. D. Johnston, *Reachability analysis of subconservative discrete chemical reaction networks*, MATCH-Communications in Mathematical and in Computer Chemistry, 81 (2019), pp. 705–736.

[87] Y. Takenaga and T. Walsh, *Tetravex is np-complete*, Information Processing Letters, 99 (2006), pp. 171–174.

[88] R. Tarjan, *Depth-first search and linear graph algorithms*, SIAM journal on computing, 1 (1972), pp. 146–160.

[89] C. Thachuk and A. Condon, *Space and energy efficient computation with dna strand displacement systems*, in International Workshop on DNA-Based Computers, 2012.

[90] G. Theyssier and N. Ollinger, *Freezing, bounded-change and convergent cellular automata*, Discrete Mathematics & Theoretical Computer Science, 24 (2022).

[91] A. J. Thubagere, W. Li, R. F. Johnson, Z. Chen, S. Doroudi, Y. L. Lee, G. Izatt, S. Wittman, N. Srinivas, D. Woods, et al., *A cargo-sorting DNA robot*, Science, 357 (2017), p. eaan6558.

[92] J. W. Uiterwijk, *Solving strong and weak 4-in-a-row*, in 2019 IEEE Conference on Games (CoG), 2019, pp. 1–8.

[93] B. Wang, C. Thachuk, A. D. Ellington, E. Winfree, and D. Soloveichik, *Effective design principles for leakless strand displacement systems*, Proceedings of the National Academy of Sciences, 115 (2018), pp. E12182–E12191.

[94] H. Wang, *Proving theorems by pattern recognition — ii*, The Bell System Technical Journal, 40 (1961), pp. 1–41.

[95] Wikipedia, *Eternity ii*. en.wikipedia.org/wiki/Eternity_II_puzzle, December 2024.

[96] E. Winfree, *Algorithmic self-assembly of DNA*, PhD thesis, 1998.

[97] E. Winfree, *Algorithmic Self-Assembly of DNA*, PhD thesis, California Institute of Technology, June 1998.

[98] D. Woods, D. Doty, C. Myhrvold, J. Hui, F. Zhou, P. Yin, and E. Winfree, *Diverse and robust molecular algorithms using reprogrammable dna self-assembly*, Nature, 567 (2019), pp. 366–372.

VITA

Ryan Arlie Knobel, born and raised in the Rio Grande Valley, attained his Bachelors of Computer Science with a Minor in Statistics at the University of Texas Rio Grande Valley (UTRGV) in December of 2023. It was in his junior year where he first learned about research and joined the Algorithmic Self-Assembly Research Group (ASARG). Ryan continued his education at UTRGV, working under ASARG until the completion of his Masters of Science in Engineering degree in December of 2025.

Ryan can be contacted at r.knobel27@yahoo.com.