CZECH TECHNICAL
UNIVERSITY
IN PRAGUE

F3

Faculty of Electrical Engineering
Department of Control Engineering

Master's Thesis

Self-assembly: modelling, simulation, and planning

Bc. Lukáš Bertl
Cybernetics and Robotics: Systems and Control
bertlluk@fel.cvut.cz

May 2019
Supervisor: RNDr. Miroslav Kulich, Ph.D.

Acknowledgement / Declaration

I would like to express my gratitude to my supervisor RNDr. Miroslav Kulich, Ph.D. for a great mentorship, patience, and dedicated involvement in every step throughout the process of solving this project.

I would like to thank my girlfriend and my parents for their unlimited mental support and endless patience throughout my whole studies.

I hereby declare that I have completed this thesis with the topic ”Self-assembly: modelling, simulation, and planning” independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

In Prague, May 21, 2019

. . . .
Lukáš Bertl

Abstrakt /

Samoskládání je proces, při kterém se kolekce neuspořádaných částic samovolně orientuje do uspořádaného vzoru nebo funkční struktury bez působení vnější síly, pouze za pomoci lokálních interakcí mezi samotnými částicemi. Tato teze se zaměřuje na teorii dlaždicových samoskládacích systémů a jejich syntézu.

Nejdříve je představena oblast výzkumu věnující se dlaždičovým samoskládacím systémům, a poté jsou důkladně popsány základní typy dlaždicových skládacích systémů, kterými jsou *abstract Tile Assembly Model* (*aTAM*), *kinetic Tile Assembly Model* (*kTAM*), a *2-Handed Assembly Model* (*2HAM*). Poté jsou představeny novější modely a modely se specifickým použitím. Dále je zahrnut stručný popis původu teorie dlaždicového samoskládání společně s krátkým popisem nedávného výzkumu. Dále jsou představeny dva obecné otevřené problémy dlaždicového samoskládání s hlavním zaměřením na problém *Pattern Self-Assembly Tile Set Synthesis* (*PATS*), což je NP-těžká kombinatorická optimalizační úloha. Nakonec je ukázán algoritmus *Partition Search with Heuristics* (*PS-H*), který se používá k řešení problému PATS.

Následovně jsou demonstrovány dvě aplikace, které byly vyvinuty pro podporu výzkumu abstraktních dlaždicových skládacích modelů a syntézy množin dlaždic pro samoskládání zadaných vzorů. První aplikace je schopná simulovat aTAM a 2HAM systémy ve 2D prostoru. Druhá aplikace je řešič PATS problému, který využívá algoritmu PS-H. Pro obě aplikace jsou popsány hlavní vlastnosti a návrhová rozhodnutí, která řídila jejich vývoj.

Nakonec jsou předloženy výsledky několika experimentů. Jedna skupina experimentů byla zaměřena na ověření výpočetní náročnosti vyvinutých algoritmů pro simulátor. Druhá sada experimentů zkoumala vliv jednotlivých vlastností vzorů na vlastnosti dlaždicových systémů, které byly získány syntézou ze vzorů pomocí vyvinutého řešiče PATS problému. Bylo prokázáno, že algoritmus simulující aTAM systém má lineární časovou výpočetní náročnost, zatímco algoritmus simulující 2HAM systém má exponenciální časovou výpočetní náročnost, která navíc silně závisí na simulovaném systému. Aplikace pro řešení syntézy množiny dlaždic ze vzorů je schopna najít relativně malé řešení i pro velké zadané vzory, a to v přiměřeném čase.

Klíčová slova: Samoskládání dlaždic, Algoritmické samoskládání, Dlaždicový skládací model, Skládání vzoru, Syntéza množiny dlaždic, Wangovi dlaždice

Překlad titulu: Samoskládání: modelování, simulace a plánování

Abstract /

Self-assembly is the process in which a collection of disordered units organise themselves into ordered patterns or functional structures without any external direction, solely using local interactions among the components. This thesis focuses on the theory of tile-based self-assembly systems and their synthesis.

First, an introduction to the study field of tile-based self-assembly systems are given, followed by a thorough description of common types of tile assembly systems such as *abstract Tile Assembly Model* (*aTAM*), *kinetic Tile Assembly Model* (*kTAM*), and *2-Handed Assembly Model* (*2HAM*). After that, various recently developed models and models with specific applications are listed. A brief summary of the origins of the tile-based self-assembly is also included together with a short review of recent results. Two general open problems are presented with the main focus on the *Pattern Self-Assembly Tile Set Synthesis* (*PATS*) problem, which is NP-hard combinatorial optimisation problem. *Partition Search with Heuristics* (*PS-H*) algorithm is presented as it is used for solving the PATS problem.

Next, two applications which were developed to study the abstract tile assembly models and the synthesis of tile sets for pattern self-assembly are introduced. The first application is a simulator capable of simulating aTAM and 2HAM systems in 2D. The second application is a solver of the PATS problem based around the PS-H algorithm. Main features and design decisions are described for both applications.

Finally, results from several experiments are presented. One set of experiments were focused on verification of computation complexity of algorithms developed for the simulator, and the other set of experiments studied the influences of the properties of the pattern on the tile assembly system synthesised by our implementation of PATS problem solver. It was shown that the algorithm for simulating aTAM systems have linear computation time complexity, whereas the algorithm simulating 2HAM systems have exponential computation time complexity, which strongly varies based on the simulated system. The synthesiser application is capable of finding a relatively small solution even for quite large input patterns in reasonable amounts of time.

Keywords: Tile-based self-assembly, Algorithmic self-assembly, Tile assembly model, Pattern assembly, Tile set synthesis, Wang tiles

Contents /

Tables / Figures

Listings / Algorithms

Chapter 1 Introduction

Self-Assembly is the ultimate dream of every lazy engineer. Imagine that the only work needed to be done to assemble the desired structure or mechanism is to bring all the components into one place, mix them, and the forces of nature construct the structure without any other outside help. The attractiveness of this manufacturing vision motivated countless chemists, physicists, and engineers over the past several decades to develop systems which are able to self-assemble at various scales from the nanoscale to the macroscale. The research field of the self-assembly has grown wide, and it is full of exciting ideas.

The self-assembly is a common phenomenon in nature which is constantly ongoing around us and inside of our bodies. Self-assembly is occurring in many organic and inorganic systems as it, for example, guides the growth of crystals, or it helps to construct organic cells. And yet, the laws of physics governing its functioning has not been fully understood to this day. The self-assembly is heavily utilised in the study of nanomaterials, and it is used to construct structures at the nanoscale, where the classical top-down manufacturing process is hard to implement. However, as the bottom-up manufacturing techniques mature such as 3D additive printing, the power of self-assembly could also be harnessed in the manufacturing of the macroscale structures.

In the following text, the primary motivation of this thesis is presented, and the outline of the thesis is described then.

1.1 Motivation

This thesis contributes to the *EXPRO* project which is a joint research project of the *Intelligent and Mobile Robotics Group* (*IMR*) based in the *Czech Institute of Informatics, Robotics and Cybernetic* (*CIIRC*) and the group based in the Faculty of Civil Engineering, Czech Technical University in Prague. The goal of the EXPRO project is to research how to apply the theory of self-assembly to the manufacturing process of macroscale mechanical metamaterials.

A metamaterial is a material with properties that are not found in the naturally occurring materials [1–2]. Metamaterials usually consist of carefully engineered structures of materials with often periodically repeating building blocks. Metamaterials have a wide variety of potential applications in optics, acoustics and thermodynamics. The *mechanical metamaterial* is a more recent concept which modifies how the metamaterial structure responses to the motion, deformations, stresses and mechanical energy.

Mechanical metamaterials can also behave like machines; for example, the metamaterial can act as a hinge or lever. The main advantage of a mechanical metamaterial machine is that it consists of a single block of material, and its functionality is encoded within its unique structure. Therefore, the machine itself can be more durable, the manufacture of such a machine can use less material as the machine is designed for a particular application, and the machine does not need to be further assembled.

The EXPRO project vision is to design mechanical metamaterials which could be used in the construction of buildings. These macroscale metamaterials would be assembled from the set of modules which act as building bricks of the metamaterial. The metamaterial modules can be mass produced either using 3D printing or using a conventional method as injection moulding based on the used structural material. However, the conventional methods for combining the vast number of module units into the desired configuration are not very efficient. Therefore, the self-assembly construction method is researched in the project. This approach combines the best aspects from each field, the mechanical metamaterial used for building construction can help lower amounts of used material while its mechanical properties can be better, and the problem of creating a large metamaterial structure is solved by constructing the metamaterial from module units via self-assembly.

This thesis is primarily focused on the simulation of tile-based self-assembly systems and on the procedure of synthesising a ”bill of materials” needed for self-assembly of a given pattern of modules or tiles.

1.2 Thesis outline

The text of the thesis is structured into four main chapters besides this one. Chapter 2 contains an introduction to the theory of tile-based self-assembly systems (Section 2.1). Then the chapter presents various models of tile assembly systems where the three most common models are thoroughly described (Section 2.2). The end of the theory chapter focuses on two open problems regarding the synthesis of tile assembly systems (Section 2.3).

After that, Chapter 3 introduces two applications developed for the purposes of this thesis and describes their implementation details. The first application is a simulator of abstract tile assembly systems (Section 3.1). The second presented application is a generator of tile assembly systems which self-assemble given pattern (Section 3.2).

Chapter 4 contains demonstrations of properties of the two developed applications. Finally, Chapter 5 contains a summary of results and ideas for future work.

Chapter 2
Theory

Self-assembly is the process by which a set of simple and relatively small components spontaneously assembles by themselves into a larger and more complex formation. This process is taking place without any external controlling force, and it is driven only by interactions between the individual elements. Furthermore, the self-assembly process has the power of significant parallelism; therefore, thousands of copies of one structure can be produced simultaneously in a single test tube.

The self-assembly occurs in many natural processes such as the process of crystal formation in snowflakes, assembly of the cell membrane from lipids, self-assembly of bacteriophage virus proteins into a capsid which is used by the virus to attack other bacteria, or the formation of double helical DNA through hydrogen bonding of the individual strands [3–4]. Even the molecular self-assembly of nanoscale structures plays a role in the growth of the β-keratin structures which give geckos the ability to climb walls and adhere to ceilings and rock overhangs [5]. From the short list of examples, it can be seen that the self-assembly processes occur mainly in biological systems, but the theory which arises from the study of self-assembling systems is also utilised in the engineering field of nanotechnology and nanomanufacturing.

The conventional top-down approach of manufacturing objects from individual molecules renders very difficult and often impossible due to the need of performing the fabrication tasks in nanoscale precision. Therefore, the bottom-up approach of the self-assembly arose as a viable and convenient manufacturing alternative to construct designed complex structures from nanomaterials. The material, which is often used to study self-assembly, is the DNA molecule. Because DNA can store information, and also its physical properties, as well as methods to synthesise it, are already well-understood [6], it is the ideal material for self-assembly.

As shown by the Adleman [7], the DNA can be used to perform computations. This shows yet another advantage of self-assembly, which is the ability to perform computations. It is a field of study of its own with the goal of encoding an input of computation problem to DNA strands such that interactions between the strands produce an encoded result. With the rising demand for systematically assembling even more complex structures, for example, to build an initial base for other artificial nanostructures, the computation approach for self-assembling structures is used more in recent years.

In order to design self-assembling systems capable of automatic construction of intricate nanostructures, theoretical models have been developed. One of the most popular models is the *Tile Assembly Model* developed by Winfree, and it was formalised in his PhD thesis [8], which mainly focuses on self-assembly of DNA tiles.

The theory of the Tile Assembly Model is presented in the rest of this chapter. The origins of the model are introduced in Section 2.1 together with the short overview of the recent results of self-assembly research and with preliminaries and notation definitions used in this thesis. Section 2.2 introduces several models of Tile Assembly Systems. Finally, two main problems studied in tile self-assembly systems are explained in Section 2.3.

2.1 Self-Assembly of Tiles

In the following Section 2.1.1, a brief background to the development of tile-based self-assembly theory is presented. Section 2.1.2 contains a summary of the recent results of the research. Finally, Section 2.1.3 defines the fundamental theory of the Tile Assembly Model using mathematical formalism, which also serves as an overview of the notation used in this thesis.

2.1.1 Origin of Tile Assembly Model

The *Tile Assembly Model* developed by Winfree [8] was based on breakthroughs in synthesising DNA molecules [9] and on a generalisation of the theory of *Wang tiles* [10]. One of the aims in nanofabrication is building two-dimensional nanostructure templates for attaching other functional objects. Such templates can be assembled from molecular units consisting of DNA molecules called *DNA tiles* which have on its four sides a *sticky end* [11–13]. A Wang tile is a unit square with four colours marking its sides. A set of tiles is formed by selecting a finite and fixed number of different Wang tiles. Tiles in the chosen set cannot be rotated or reflected, and they are placed on a plane only in such a way that both sides of every two touching tiles have a matching colour. The main question about a set of Wang tiles is whether the set could cover the whole plane at least in one configuration while following the mentioned rules. The model *tile* used in Tile Assembly Models is a generalisation of the Wang tile enhanced by properties of the DNA tile. The model tile has different kinds of *glues* on its edges similarly as the Wang tile has different colours. A glue will only adhere to the same glue type, and each glue type can have various bonding strength.

Even though the similarity between *Wang tile* and *self-assembly tile* in Winfree's Tile Assembly Models can seem substantial, these two theories solve completely different questions. Whilst Wang tiling focuses on finding such a set of tiles that is able to cover an infinite tile plane entirely and whether the produced tiling have periodic pattern, the self-assembly of tiles focuses on designing such tile sets which are capable of autonomously orient themselves into desired structures, shapes or patterns with minimizing the tile set itself, minimizing the construction errors and minimizing time required for assembly. Another difference is that the process of self-assembly in the Tile Assembly Model progressing in discrete time steps when the assembly grows one tile at a time instead of instantaneous coverage of the whole plane as in Wang tiling.

Winfree developed two basic Tile Assembly Models: *the abstract Tile Assembly Model* (*aTAM*) and *the kinetic Tile Assembly Model* (*kTAM*), which are both further discussed in Sections 2.2.1 and 2.2.2, respectively. The aTAM, as the name suggests, provides an abstract high-level view on the theory on self-assembly, which ignores the laws of physics of possible real-world implementation; also, it ignores the possibility of errors. That is why aTAM is a great framework for studying mathematical properties and the possible limits of the self-assembly systems. On the other hand, the kTAM models the behaviour of the physical and chemical kinematics, and therefore, it allows to study how errors form while assembly process taking place. Also, the kTAM is used to develop mechanisms for detecting these errors, mechanisms for preventing the error from even occurring or design tile sets in such a way that the tiles can correct the errors during self-assembly.

It was proved that aTAM is *Turing complete* [8]. This fact means that the self-assembly can be algorithmically directed and so the study of aTAM systems fall into a general field of *algorithmic self-assembly* where it affects other areas of Theoretical Computer Science or Mathematics. There are several designs of aTAM tile sets that algorithmically self-assembles complex structures such are binary counters or fractal patterns known as *the Sierpinski triangle* or *the Sierpinski carpet* [8, 14–16].

All this contributed to turning the self-assembly into a quite broad and vibrant field of research which ranges from biology, over nanotechnology and mathematics to computer science and even robotics. A brief overview of the recent results is given in the following section.

2.1.2 A brief overview of recent results

An incomplete overview of the recent research results from the field of self-assembly is provided in this section. The reader is encouraged to refer to surveys [6, 17] for further details about development in the field because the articles mentioned in this section are only from research that came after these reviews. Both articles serve as excellent surveys into the area of algorithmic tile-based self-assembly. The reader also can refer to review [18] for more information about self-assembly techniques used in nanomaterial fabrication.

The research of self-assembly includes multiple disciplines. In nanofabrication research, self-assembly was used to construct optical nanomaterials [19–20], fabrication of nanostructures based on protein self-assembly [21–22], or self-assembly of nanoscale metamaterials [23].

Self-assembly techniques were used for achieving a formation of miniature robot swarms into various shapes. One approach of testing self-assembly with robots is to use robots floating in water [24–25]; other experiments were performed using kilobots [26–27].

Another direction of research aims to find limits in theoretical models of algorithmic self-assembly. The $aTAM$ model of self-assembly was proven to be *intrinsically universal* with the right system temperature [28–29], which led to attempts of proving this property also for other models [30–31]. Various tile assembly models, such is $aTAM$ or $kTAM$, are discussed in detail in Section 2.2.

There were experiments with algorithms for self-assembly of shapes at system temperature $\tau = 1$ [32], or for assembling shapes in the *staged self-assembly* model [33–34]. Algorithm for finding minimal tile sets for self-assembling square with minimal system temperature was developed in [35].

An optimisation method for designing tile sets for avoiding errors was developed in [36] while preserving method called *proofreading*, which helps with the correctness of the assembly in the $kTAM$ model.

Additionally, there were attempts to proving that the problem of self-assembling a desired multi-coloured pattern is NP-hard or NP-complete depending on the number of colours in the pattern in [37] and [38], respectively. This problem is called the *PATS Problem*, and it is further described in Section 2.3.2).

The mathematical model of the tile-based self-assembly system is defined in the next section, which also serves as an overview of the mathematical notation, which is used in this thesis.

2.1.3 Preliminaries and notation

In this section, a set of definitions is provided, which is then used to describe *Tile Self-Assembly System* (*TAS*) models. All the following TAS models can be generalised to a 3-dimensional space, but for the sake of simplicity and without loss of generality, the models will be described in 2-dimensional discrete space. The used notation is derived from those of [39–41].

Informally, each unit of an assembly is called *tile* which is a square with *glues* of various types on its every side (edge). The tile "floats" on a two-dimensional plane, and when two tiles collide, they bond together if their abutting sides have compatible glues with sufficient strength. The formal definitions are presented in the following text.

Definition 2.1. Define the set of all *unit vectors* in $\mathbb{Z}^2$ as

$$U_2 = \{(0,\, 1),\, (1,\, 0),\, (0,\, -1),\, (-1,\, 0)\}\,.$$

These vectors can be referred as their cardinal directions *North*, *East*, *South*, *West* or as N, E, S and E, respectively.

Definition 2.2. A *grid graph* is an undirected graph $G = (V,\, E)$ in which $V \subseteq \mathbb{Z}^2$ is set of all *vertices* and E is set of all *edges*. Edge is defined by an unordered pair of vertices $\{\vec{a}, \vec{b}\} \in E,\ \vec{a}, \vec{b} \in V$. In the grid graph, edges exists only between the vertices with the property $\vec{a} - \vec{b} \in U_2$. All graphs in this thesis are assumed as undirected if not specified otherwise.

Definition 2.3. Let $\mathcal{D} = \{N(x,\, y),\, E(x,\, y),\, S(x,\, y),\, W(x,\, y)\}$ be the set of four functions $\mathbb{Z}^2 \to \mathbb{Z}^2$ corresponding to shift of position $(x,\, y)$ in terms of cardinal directions $\{\mathrm{N, E, S, W}\} \in U_2$ so that position will be shifted by a unit vector in a *direction* $d \in U_2$. Let the shift in north direction be $N(x,\, y) = (x,\, y+1)$, the shift in east direction $E(x,\, y) = (x+1,\, y)$, the shift in south direction $S(x,\, y) = (x,\, y-1)$ and last the shift in west direction $W(x,\, y) = (x-1,\, y)$ and inverted functions have the same meaning as opposite cardinal direction, therefore $S(x,\, y) = N^{-1}(x,\, y)$ and $W(x,\, y) = E^{-1}(x,\, y)$.

Definition 2.4. Additionally, a notation $\mathcal{D}(x,\, y)$ can be used to represent a set of positions in the *four-neighbourhood* of position $(x,\, y)$ in a grid graph G.

Definition 2.5. Let Σ be the set of *glue types* and $s\, :\, \Sigma \times \Sigma \to \mathbb{N}$ a *glue strength* function such that $s(\sigma_1,\, \sigma_2) = s(\sigma_2,\, \sigma_1)$ for all glue types $\sigma_1,\, \sigma_2 \in \Sigma$. The strength function s returns a non-negative number which represents a *bonding strength* of the glue if it bonds with another glue. Only those glue strength functions such that $s(\sigma_1,\, \sigma_2) = 0$ if $\sigma_1 \neq \sigma_2$ are considered in this thesis, which means that only two glues with the same glue type can generate non-zero bonding strength. Each glue type can be identified using a *label*. The type of the glue type label can differ in implementation, but for simplicity, integer numbers are used as glue label in this thesis.

Definition 2.6. *Null glue* is a special glue type that has constant zero glue strength function. This means that it cannot bond with no other glue. i.e., $s(null,\, \sigma) = 0$ for all $\sigma \in \Sigma$. The *null glue* can be used to represent a missing glue or a glue with no bonding action.

Definition 2.7. *Tile type* $t \in \Sigma^4$ is defined as quadruple of glue types

$$t = (\sigma_N(t),\, \sigma_E(t),\, \sigma_S(t),\, \sigma_W(t))$$

for each side of a unit square, where $\sigma_d(t)$ is a notation for the glue type of the tile t facing the cardinal direction $d \in U_2$. *The finite set of tile types* is denoted as T. Each tile type can have a *tile label* which can be used to carry or show some useful information in a tile assembly, i.e. a value of a bit in binary counter or colour of the tile in a pattern.

The illustration of a tile used in this thesis can be seen in Figure 2.1. The tile representation consists of several parts. The square represents a body of a tile, and it can have arbitrary colour. The colour of tile body, as well as text on place of `Label`, can be used to carry a piece of information about the purpose, type or category of a tile but they have no impact on the behaviour of the tile in the self-assembly process. Small black lines on sides of the tile body represent tile glues, where the number next to each line denotes the glue type, and the number of lines on one side denotes the strength of the glue bond. In Figure 2.1, the depicted tile has a glue of type `1` with bonding strength 2 on its north side, then a glue of type `2` with strength 1 on its east side, a glue of type `3` with strength 2 on its west side and a *null glue* on its south side. The grey number in the right-bottom corner of the tile body is an *identification number* (*ID*) of the tile in a given tile set, and it is used only as a reference to one particular tile type in the tile set. The grey vector in the left-top corner of the tile body shows a position of the tile in the space of an assembly.

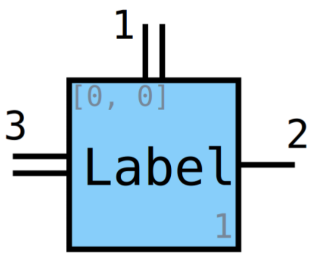


Figure 2.1. Graphical representation of a tile.

A tile with *null glues* on all its sides cannot contribute to the assembly process, hence it is not explicitly used in any tile assembly system, but it can be used to represent an empty space in the *assembly configuration* (Definition 2.8); therefore, it is referred as *empty tile*.

Definition 2.8. An *assembly* $\mathcal{A}$ (also called *configuration* in some literature) is a partial mapping from $\mathbb{Z}^2$ to $T \in \Sigma^4$ defined on at least one input. A tile t that lies on the position $\vec{a} = (x, y)$ in an assembly $\mathcal{A}$ can be denoted as $\mathcal{A}(\vec{a})$ or $\mathcal{A}(x, y)$. The points $\vec{x} \in \mathbb{Z}^2$ at which $\mathcal{A}$ is undefined are interpreted to be an empty space. Therefore, the domain of assembly dom $\mathcal{A}$ is the set of points with defined non-empty tiles. The number of points with tiles can be written as $|\text{dom}\ \mathcal{A}|$ or $|\mathcal{A}|$, and the assembly is called *finite* if $|\mathcal{A}|$ is finite. Also, an assembly can be denoted as T*-assembly* when it is needed to specify which tile set T is used to build the assembly. The assembly can be visualised as grid graph $G = (V, E)$, where vertices $V \in \mathbb{Z}^2$ are points that map to tiles and edges E map to tile glues.

Definition 2.9. Additionally, an assembly $\mathcal{A}_1$ is called *sub-assembly* of assembly $\mathcal{A}_2$, which is written as $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$, if dom $\mathcal{A}_1 \subseteq$ dom $\mathcal{A}_2$ and $\mathcal{A}_1(\vec{x}) = \mathcal{A}_2(\vec{x})$ for all points with tiles $\vec{x} \in$ dom $\mathcal{A}_1$.

Definition 2.10. The *Tile Assembly System* also referred to as *TAS* $\mathscr{T} = (T, \mathcal{S}, s, \tau)$ consists of a finite set of tile types used in the system T, a seed structure $\mathcal{S}$, a glue strength function s and a system temperature $\tau \in \mathbb{N}$. The seed $\mathcal{S}$ is used as the initial state of the system which can be used to direct the behaviour of the self-assembly system, and it can be optional or not used for some of the TAS models. The temperature

τ sets a minimal bond strength for the system so each glue bond that according to glue strength function s does not have the strength greater than or equal to τ will not be bonded. Furthermore, it can be noted that even though the number of different tile types is fixed and finite, the number of tiles (copies) of each type can be infinite in an assembly.

Definition 2.11. An assembly is called *τ-stable* if it cannot be broken up into smaller sub-assemblies without breaking glue bonds of total strength at least τ, for some $\tau \in \mathbb{N}$. Alternatively, the addition of a tile to the assembly is *τ-stable* if the tile adheres to the assembly with bond strength equal to or greater than τ.

The rule for *extending* an assembly in the self-assembly process can vary according to used model, but the following formalisation of the process can be used to show the basic concepts of the established models such as *aTAM*, *kTAM* or *2HAM* which are described in Sections 2.2.1, 2.2.2 and 2.2.3, respectively.

The self-assembly process is described as perpetual *extending* an assembly by a tile from fixed TAS until there is at least one tile that can be τ-stably added to the assembly which is formally described by the rule in the next Definition 2.12.

Definition 2.12. For two assemblies $\mathcal{A}$, $\mathcal{A}'$ for which applies $\mathcal{A} \sqsubseteq \mathcal{A}'$ and a fixed TAS $\mathscr{T} = (T, \mathcal{S}, s, \tau)$, it can be said that $\mathcal{A}'$ *extends* $\mathcal{A}$ which is written using relation as $\mathcal{A} \rightarrow_{\mathscr{T}} \mathcal{A}'$ if there exists a position $\vec{p} = (x, y) \in \mathbb{Z}^2$ and a tile $t \in T$ such that $\mathcal{A}' = \mathcal{A} \cup \{(\vec{p}, t)\}$, where the union is disjoint and

$$\sum_{D} s(\sigma_D(t),\, \sigma_{D^{-1}}(\mathcal{A}(D(x,\, y))) \geq \tau,$$

where D ranges over those positions given by direction functions in $\mathcal{D}$ for which the assembly $\mathcal{A}$ has defined tiles. This formula can be rewritten if we define the *set of defined tiles in four-neighbourhood of position* (x, y) as $\mathcal{N}(x, y) = \{n \in \mathcal{A}(\vec{x}) : \vec{x} = D(x, y),\ n \neq \textit{empty tile}, \forall D \in \mathcal{D}\}$. Now, the formula can be written as

$$\sum_{n_d \in \mathcal{N}(x,\, y),\, d \in D} s(\sigma_d(t),\, \sigma_{d^{-1}}(n_d)) \geq \tau,$$

where $n_d \in \mathcal{N}(x, y)$ denotes neighbouring tile which lies next to position (x, y) in cardinal direction d, σ_d denotes a glue type on tile side in cardinal direction d, and d^{-1} denotes a complementary cardinal direction to direction d.

Using the rewritten formula, it is easier to see that the extending tile $t \in T$ can be adjoined to an assembly $\mathcal{A}$ on position (x, y) only if t shares a common boundary with tiles that bind it into place (neighbouring tiles $\mathcal{N}(x, y)$), which are already present in the assembly $\mathcal{A}$. Also, corresponding glues of the extending tile t and its adjacent tiles have to generate total strength greater than or equal to system temperature τ. If this rule holds, a new assembly $\mathcal{A}'$ is created.

Definition 2.13. An *assembly sequence* $\vec{\mathcal{A}}$ is used to describe the steps in self-assembly process of the system $\mathscr{T}$ using the relation of extending an assembly

$$\mathcal{A}_i \rightarrow_{\mathscr{T}} \mathcal{A}_{i+1} \rightarrow_{\mathscr{T}} \mathcal{A}_{i+2} \rightarrow_{\mathscr{T}} \cdots,$$

Definition 2.14. Let $\rightarrow^*_{\mathscr{T}}$ be the reflexive transitive closure of $\rightarrow_{\mathscr{T}}$. A TAS $\mathscr{T}$ *produces* an assembly $\mathcal{A}$ if $\mathcal{A}$ is an *extension* of the seed assembly $\mathcal{S}$, that is if $\mathcal{S} \rightarrow^*_{\mathscr{T}} \mathcal{A}$. Prod $\mathscr{T}$ denotes *the set of all assemblies produced by* the TAS $\mathscr{T}$. This way, the pair (Prod $\mathscr{T}$, $\rightarrow^*_{\mathscr{T}}$) forms a partially ordered set.

Definition 2.15. A TAS $\mathscr{T}$ is *deterministic* (also called *directed*, *confluent* or *produces a unique assembly* in other literature) if the relation $\rightarrow_{\mathscr{T}}$ is directed, i.e., if for all

$\mathcal{A}$, $\mathcal{A}' \in \operatorname{Prod} \mathscr{T}$, there exists $\mathcal{A}'' \in \operatorname{Prod} \mathscr{T}$ such that $\mathcal{A} \to_{\mathscr{T}} \mathcal{A}'$ and $\mathcal{A}' \to_{\mathscr{T}} \mathcal{A}''$. Alternatively, a TAS $\mathscr{T}$ is *deterministic* if for any assembly $\mathcal{A} \in \operatorname{Prod} \mathscr{T}$ and for every position $(x, y) \in \mathbb{Z}^2$ there exists *at most one* $t \in T$ such that $\mathcal{A}$ can be extended with t at position (x, y). Also, a TAS $\mathscr{T}$ is deterministic precisely when $\operatorname{Prod} \mathscr{T}$ is a lattice.

Definition 2.16. The maximal elements in $\operatorname{Prod} \mathscr{T}$ are such assemblies $\mathcal{A}$ that can not be further extended, that is, there do not exist assemblies $\mathcal{A}'$ such that $\mathcal{A} \to_{\mathscr{T}} \mathcal{A}'$. These maximal elements are called *terminal assemblies*, and *the set of terminal assemblies of TAS* $\mathscr{T}$ is denoted as $\operatorname{Term} \mathscr{T}$.

Definition 2.17. If all *assembly sequences* $\vec{\mathcal{A}}$ in self-assembly process of TAS $\mathscr{T}$, which can be written as

$$\mathcal{S} \to_{\mathscr{T}} \mathcal{A}_1 \to_{\mathscr{T}} \mathcal{A}_2 \to_{\mathscr{T}} \cdots \to_{\mathscr{T}} \mathcal{P},$$

terminate with $\operatorname{Term} \mathscr{T} = \{\mathcal{P}\}$ for some assembly $\mathcal{P}$, it is said that $\mathscr{T}$ *uniquely produces* $\mathcal{P}$.

In general, even a deterministic TAS $\mathscr{T}$ may have a vast number of different assembly sequences leading to the production to its terminal assembly $\operatorname{Term} \mathscr{T}$. This number of possible sequences seems to make it very difficult to prove that a TAS $\mathscr{T}$ is directed. Fortunately, a property of assembly sequence called *local determinism* has been defined by Soloveichik and Winfree [42]. Soloveichik and Winfree also have proven the fact that, if TAS $\mathscr{T}$ has *any* locally deterministic assembly sequence, then the $\mathscr{T}$ is deterministic.

Definition 2.18. An assembly sequence $\vec{\mathcal{A}}$ is *locally deterministic* if

(i) each tile adjoined in $\vec{\mathcal{A}}$ binds to the existing assembly with bond strength exactly equal to τ;
(ii) if a tile at position $\vec{p}$ of type $t_0 \in T$ is deleted from result of $\vec{\mathcal{A}}$ and its neighbouring tiles which would be adjacent to tile t_0 are deleted from result of $\vec{\mathcal{A}}$, then no tile of type $t \neq t_0$, $t \in T$ can be attached to the configuration at location $\vec{p}$; and
(iii) the result of $\vec{\mathcal{A}}$ is terminal assembly $\mathcal{A}_T \in \operatorname{Term} \mathscr{T}$.

Definition 2.19. A set of points $X \subseteq \mathbb{Z}^2$ *weakly self-assembles* if there exists a TAS $\mathscr{T} = (T, \mathcal{S}, s, \tau)$ and a set of tiles $B \subseteq T$ such that all positions of tiles from set B in terminal assembly $\mathcal{A}_T$ match all the positions X, which can be written as $\mathcal{A}_T^{-1}(B) = X$, for every terminal assembly $\mathcal{A}_T \in \operatorname{Term} \mathscr{T}$. Therefore, weak self-assembly can be imagined as the creation or ”painting” of a pattern of tiles from B with some unique feature (i.e. colour or label) on a possibly larger ”canvas” of other tiles.

Definition 2.20. A set of points $X \subseteq \mathbb{Z}^2$ *strictly self-assembles* if there is a TAS $\mathscr{T}$ for which every assembly $\mathcal{A} \in \operatorname{Term} \mathscr{T}$ satisfies $\operatorname{dom} \mathcal{A} = X$. Therefore, strict self-assembly means that tiles are only placed in positions defined by the shape of X. Note that if X strictly self-assembles, then also X weakly self-assembles.

2.2 Models of Tile Assembly Systems

Since the first *Tile Assembly Model* was introduced by Winfree [8], there were developed many different models, but all are somewhat related to *the abstract Tile Assembly Model*, maybe more known as its acronym *aTAM*. Several basic tile assembly models are introduced and defined in the following text. First presented model is the aTAM in Section 2.2.1, followed by the description of *the kinetic Tile Assembly Model* commonly referred to as *kTAM* in Section 2.2.2. Then, a description of the newer *Two-Handed Assembly Model* or *2HAM* is in Section 2.2.3. *The Staged Self-Assembly* model derived from 2HAM is depicted herein in Section 2.2.4. Lastly, a short overview of model variants which were derived from the basic models is offered in Section 2.2.5. The formal definitions of models using definitions from Section 2.1.3.

2.2.1 Abstract Tile Assembly Model

The discrete mathematical model called *the abstract Tile Assembly Model* (*aTAM*) was derived from the *Wang tiling* [10] as is mentioned in Section 2.1.1. The aTAM system $\mathscr{T}$ consists of *a finite set of tiles* T, *seed structure* $\mathcal{S}$, *glue strength function* s and *system temperature* τ as defined in Section 2.1.3. Same as Wang tiling, the aTAM covers a 2D plane with *tiles*, but unlike Wang tiling, aTAM does this in an *assembly sequence* consisting of *steps*, one tile at a time. Note that the aTAM can be easily generalised for the 3D space, but it is defined only in two dimensions here for simplicity. The aTAM assembly is built from abstract non-rotatable *tiles* with *glues* as stated in Definition 2.7.

The *abstract* assembly model inherently and purposely ignores laws of physics of possible physical implementations. This abstraction of the model means that a tile cannot be wrongly attached to different tile with incompatible glues; this situation is studied using the kinetic model described in Section 2.2.2. Therefore, the purpose of the aTAM is solely theoretical. The aTAM can be used for the initial design of the tile sets so the possibility of the set to self-assemble into the desired structure can be tested. Another aTAM application is in testing the theoretical boundaries of algorithmic self-assembly and its computation power. It was proven that aTAM is *Turing complete* [8] and *intrinsically universal* for system temperature $\tau = 2$ [28]. That means that the aTAM system can perform arbitrary computation and for temperature $\tau = 2$ it can even simulate any other aTAM system with a single fixed set of tiles, where the mimicked system is encoded into seed structure.

The process of *assembly sequence* $\vec{\mathcal{A}}$ (Definition 2.13) of the aTAM system $\mathscr{T}$ is formally described in Algorithm 2.1, where i denotes the step in the assembly sequence. The assembly sequence starts with a *seed structure* $\mathcal{S}$ which describes the initial state (*assembly configuration*) of the aTAM and therefore it encodes the behaviour of the system, i.e., $\mathcal{S} = \mathcal{A}_0$ (line 1).

The subsequent progression to the next step in the assembly sequence consists of two actions. The first action is the selection of a tile from the set of tiles. The selection consists of checking each tile from the system set of tiles $t \in T$ whether it can be τ-stably attached at least at one undefined position in assembly $\mathcal{A}_i$, where $\mathcal{A}_i$ corresponds to the i-th system configuration in the assembly sequence. All such tiles are formed into *the set of all tiles with positions that can extend the assembly* $\mathcal{A}_i$ denoted as T_E^i. The set is T_E^i is constructed with the help of the set P, where P is the set all positions which are undefined in $\mathcal{A}_i$ and located in the four-neighbourhood of $\mathcal{A}_i$. One possible way how to get the set P is shown on lines 3–7 in the algorithm. The T_E^i is then filled

```
Procedure: aTAM-Assembly-Sequence

Inputs: Number of steps $n$,
        Tile Assembly System $\mathscr{T} = (T, \mathcal{S}, s, \tau)$,
        where $T$ is the set of tiles, $\mathcal{S}$ is seed structure, $s$ is the glue strength
        function, and $\tau$ is the temperature.
Outputs: Assembly in $n$-th step $\mathcal{P}_n$

1   $\mathcal{A}_0 := \mathcal{S};\ i := 0$
2   repeat
3       $P := \emptyset$
4       for each defined position $\vec{p}$ in $\mathcal{A}_i$ do
5           for each position $\vec{n}$ in $\mathcal{D}(\vec{p})$ do
6               if $\vec{n}$ is undefined in $\mathcal{A}_i$ then
7                   $P := P \cup \vec{n}$
8       $T_E^i := \emptyset$
9       for each tile $t$ in $T$ do
10          for each position $\vec{p}$ in $P$ do
11              if $(\mathcal{A}_i \cup (\vec{p}, t))$ is $\tau$-stable then
12                  $T_E^i := T_E^i \cup (\vec{p}, t)$
13      if $T_E^i = \emptyset$ then
14          $\mathcal{P}_n := \mathcal{A}_i$
14          return $\mathcal{P}_n$
15      $(t_e, \vec{p_e}) :=$ uniform_random$(T_E^i)$
16      $\mathcal{A}_{i+1} := \mathcal{A}_i \cup (\vec{p_e}, t_e)$
17      $i := i + 1$
18  until $i > n$
19  $\mathcal{P}_n := \mathcal{A}_i$
20  return $\mathcal{P}_n$
```

Algorithm 2.1. A pseudo-code description of the assembly process of the aTAM system, where $x =$ `uniform_random`(X) is a function which returns a random element x from given set X and where all elements are equally likely to be chosen.

with pairs $(\vec{p}, t)$, where $\vec{p} \in P$ is position and $t \in T$ is tile, such that t can τ-stably extend $\mathcal{A}_i$ on $\vec{p}$ (lines 8–12).

In the second action, *the random pair of position and tile* $(\vec{p_e}, t_e) \in T_E^i$ is chosen with equal probability and added to the assembly, which creates the assembly $\mathcal{A}_{i+1}$ (lines 15–17). From this point, the process is repeated from the creation of a new set of all possible extending tiles T_E^{i+1} and its positions for the newly extended assembly. The assembly process terminates when there are no tiles that could be attached to the assembly, i.e., $T_E^i = \emptyset$ (lines 13, 14).

A simple example of self-assembly of the binary counter is presented in the following text.

Example 2.1. *aTAM: Binary counter*
In Figure 2.2, a visualisation of the aTAM tile set is depicted. The tile set can be used to self-assembly a n-bit binary counter for system temperature $\tau = 2$, where n depends on the configuration of the seed structure. Note that there exist more different sets that can count in binary in various ways. Also, note that the graphical representation of a tile tends to have a grey *ID* number in its right-bottom corner, which can be used as a reference to this tile type in the discussed tile set, as it is described in Definition 2.7.

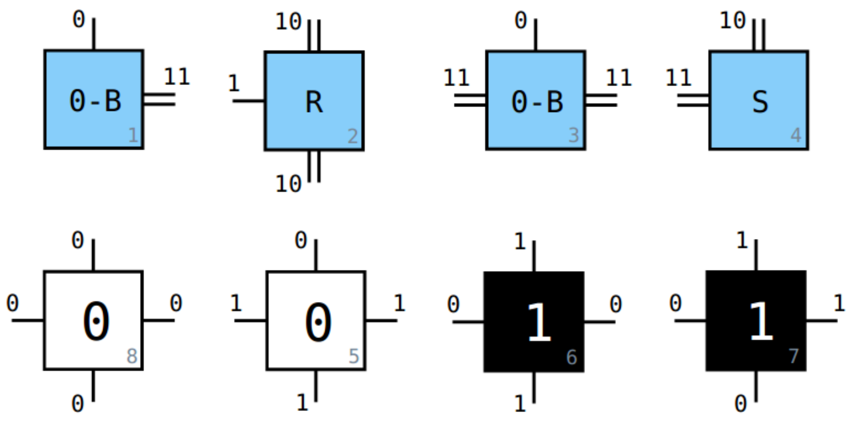


Figure 2.2. Visualisation of a possible tile set for self-assembling a n-bit binary counter in the aTAM system with temperature $\tau = 2$.

The notation $t_{ID} \in T$ is used to reference the tile with specified ID in the subscript, e.g. the tile with ID $= 4$ labelled S is referenced as t_4.

In Figure 2.2, tiles with the blue background are those which are used to generate the seed structure in the self-assembly of the n-bit counter. The tile t_4 with label S is the main seed tile, which lies in the origin of assembly configuration space. Tiles t_1, t_3 with label 0-B are used to assemble the bottom border of the assembly; hence the B in their label and they also carry the zero value of the binary counter, hence the 0. The tile t_2 labelled R is used as the right border of the counting assembly, and it is not carrying any useful information. Finally, the tiles with label 0 (t_5, t_8) or 1 (t_6, t_7) represent a value of a bit on its position.

As was mentioned above, the number of bits n of this binary counter depends on the configuration of the seed structure for this tile set. More precisely, the number of bits n is defined by the width of the bottom border of the assembly which is constructed from the tile t_1 and sequence of $n-1$ tiles t_3 which are then connected to the origin seed tile t_4. Therefore, the bottom border of the seed structure for the 3-bit counter consists of tiles $(t_1 \quad t_3 \quad t_3 \quad t_4)$, in this order as it can be seen in Figure 2.3a. It can be seen that the tile t_1 disables another tile to be attached to its west side and thus delimiting the number of bits in the counter. Note that, delimiting the number of values of the binary counter can be done similarly by defining the number of bits by encapsulating the right border in the seed structure with additional tile type same as tile t_2 but without its north glue. The definition of the maximal value of the counter was omitted in this example in order to keep the number of tiles in the set low as possible. On the contrary, note that a ∞-bit binary counter could be self-assembled if the tile t_1 were removed from the tile set.

The assembly sequence of the 3-bit binary counter is illustrated in Figure 2.3. The initial assembly $\mathcal{A}_0$ starts its sequence from the seed structure $\mathcal{A}_0 = \mathcal{S}$, which can be seen in Figure 2.3a. The green squares in the assembly visualisation indicate positions where a tile $t \in T$ could τ-stably extend the assembly. So, in the next step of assembly sequence $\mathcal{A}_1$, the tile t_7 was added on position $(-1, 1)$ relative to the origin seed tile with label S on position $(0, 0)$. It can be noticed that the right border assembled from tiles of the type t_2 can be extended in every step because the tile t_2 has on both north and south sides the same glue type 10 with strength $2 = \tau$ and the right border was not terminated similarly to the bottom border as mention before. Therefore, the tile t_2 can self-assembly long chain of same-type tiles. On the other hand, tiles that represent

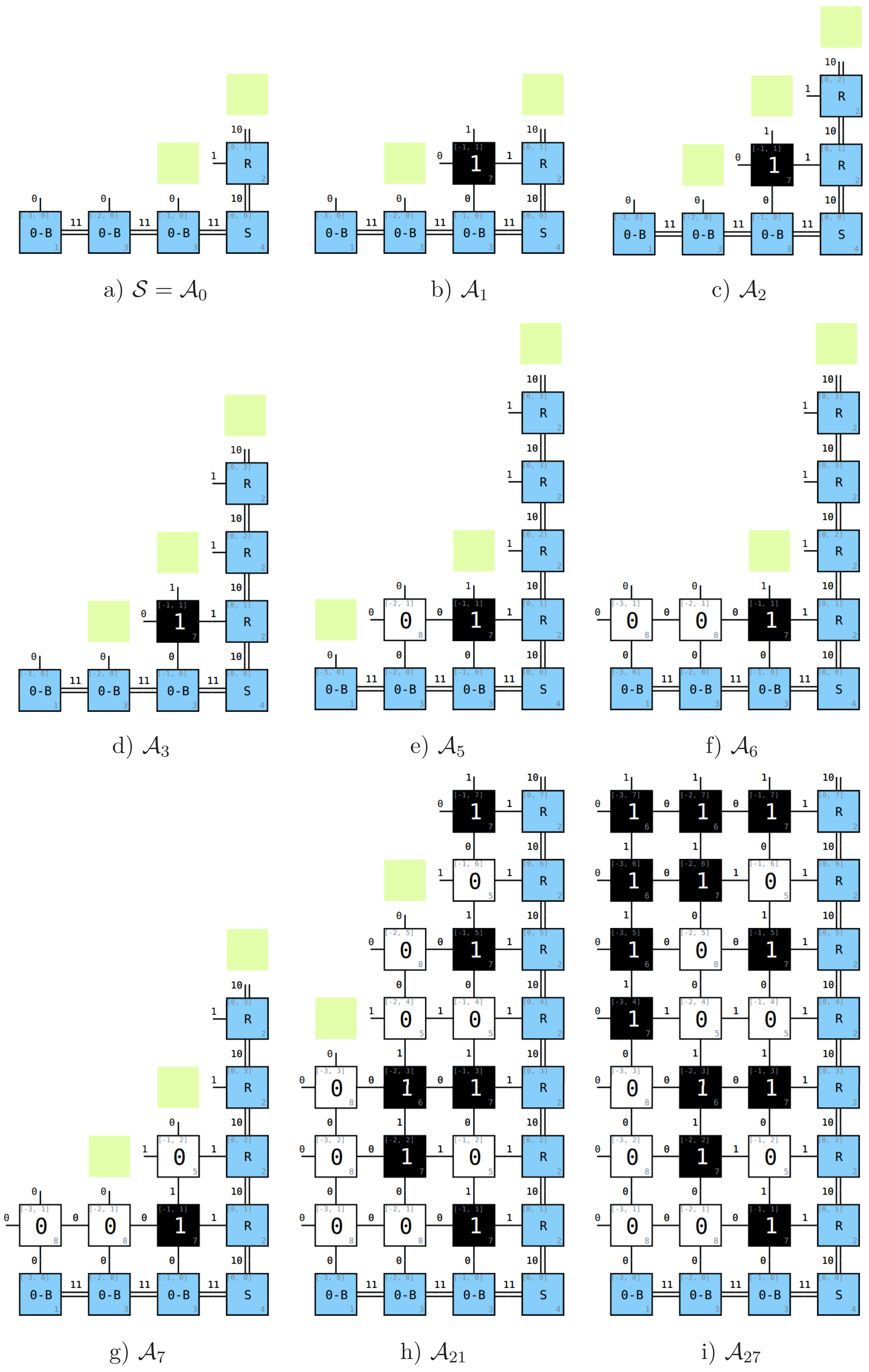


Figure 2.3. Assembly sequence of 3-bit binary counter aTAM system $\mathscr{T} = (T, \mathcal{S}, s, \tau = 2)$, where the green squares indicate a possible position where the assembly could be τ-stably extended by a tile from T and subscript i in $\mathcal{A}_i$ denotes the step of the assembly.

bit values (t_5, t_6, t_7, t_8) can attach only on those positions where they can adhere to both east and south neighbours simultaneously, because all their glues have strength 1, therefore in the system with temperature $\tau = 2$ the tiles need to *cooperate* to create sufficient bond strength. This is also behind the mechanism of limiting the counter to 3-bits because the bottom border is not defined on position $(-4, 0)$, the tiles with strength 1 cannot bond further left due to lack of glue strength which would support them from the south side, as can be shown in Figures 2.3f–i.

The right border continues with self-assembly in steps 2–4 of the assembly sequence, which is captured in Figures 2.3c–d. The tile representing bit values extended the assembly in steps 5–7 as shown in Figures 2.3e–g. One of the intermediate steps is shown in Figure 2.3h. From this progress, it can be seen that the assembly of the aTAM system is random in the sense that in each step of the assembly sequence a tile is added on random position which can τ-stably extend the assembly, but if the tile set is adequately designed, tiles eventually end up in the correct positions. The final configuration of the binary counter is given in Figure 2.3i, where it is easy to see the pattern of binary numbers in each row of the assembly.

2.2.2 Kinetic Tile Assembly Model

The kinetic Tile Assembly Model (kTAM) is a more realistic counterpart of the aTAM. As was already mentioned in Section 2.1.1, kTAM was developed together with aTAM by Winfree [8] for modelling of DNA self-assembly. The kTAM has the same general rules as the aTAM, the assembly is constructed from the same model of tiles and in sequence from given seed structure, but unlike aTAM, kTAM allows errors to occur in the assembly process. The allowance of the errors in the assembly means that tiles can adhere to incompatible glues, or tile can be randomly detached from the assembly. Thanks to this uncertainty in tile bonding kTAM functions as quite a realistic model which was used to predict results of laboratory experiments accurately, for example [43, 14]. The accuracy of kTAM helped in the development of error prevention and correction techniques, which minimised the rate of errors seen in laboratory experiments.

The formal description of the kTAM uses a notion of a *monomer* tile. Its name is derived from the designation for a molecule from which polymers are built. The monomer tile in the context of the kTAM denotes every tile from the system tile set which can be added to the assembly *frontier* (i.e. on every unoccupied position around the assembly), regardless of whether the glue type of the tile in the assembly and monomer tile match.

Definition 2.21. The rate of a particular monomer tile attaches to an assembly at a particular position is called *association rate* or *forward rate* denoted as r_f measured in Hz and defined as

$$r_f = k_f[DX] = k_f e^{-G_{mc}},$$

where k_f is the *forward rate constant* which sets the units of the time axis and it has no impact on the behaviour of the system, $[DX]$ denotes the effective concentration of the monomer tiles which aggregates entropic factors of the monomer and $G_{mc} > 0$ is the *non-dimensional entropic cost of associating to an assembly* which depends on the monomer tile concentration. If the effective concentration of the monomer tile $[DX]$ is constant, then $[DX] = e^{-G_{mc}}$.

Definition 2.22. The rate of breaking glue bonds of a tile in an assembly is called *dissociation rate* or *reverse rate* denoted as $r_{r,b}$ measured in Hz and defined as

$$r_{r,b} = k_f e^{-bG_{se}},$$

where $G_{se} > 0$ is the *non-dimensional free energy cost of breaking a single bond* and b is the number of "single-strength" matching bonds which the tile has attached.

The ratio of the concentration of monomer tiles to the strength of their individual glue bonds provides an analogue for the temperature τ aTAM parameter, which is written as

$$\tau \doteq \frac{G_{mc}}{G_{se}}.$$

Therefore, the lower the ratio of G_{mc}/G_{se}, the assembly process faster but more error-prone, similarly as temperature τ affects the aTAM system. The number of matching glue bonds that a tile has with an assembly b is used to decide whether a tile will be attached to or detached from the assembly. If b is less then τ, then the tile is more likely to be detached from the assembly.

This setting creates an opportunity for various types of errors to occur. The kTAM assembly errors can be divided into three general categories: *growth errors*, *facet errors* and *nucleation errors*. First two categories are somewhat similar. The *growth error* or sometimes called the *mismatch error* describes a situation when a tile adheres to an assembly, but one or more glues of the tile do not match with glues of tiles in the assembly, hence those glues *mismatches*. However, before the mismatched tile dissociates with the assembly, another tile with correct glues is attached to the mismatched tile and the assembly, which effectively *locks* the mismatched tile to the assembly. The *facet error* is very similar to the *growth error* except that the erroneous tile has all correct glues, but it does not have enough bond strength in them. The erroneous tile is then locked in place in the same manner as in *growth error*. The last category *nucleation error* describes a situation when two or more monomer tiles attaches together but not to the assembly. Therefore, they create another seed structure in the system without designed purpose and start construction of the unwanted assembly.

The assembly process of the kTAM system is described in Algorithm 2.2. The kTAM system $\mathscr{T}$ is specified by 6 parameters $\mathscr{T} = (T, \mathcal{S}, s, G_{mc}, G_{se}, k_f)$, where T is the set of all tiles that can occur in the assembly, $\mathcal{S}$ is the seed structure placed over the origin of the assembly plane, s is the glue strength function, G_{mc} is the entropic cost of fixing the location of a monomer tile, G_{se} is the entropic free energy cost of breaking a strength-1 bond and k_f is the forward rate constant.

At the start of the process, the assembly configuration equal to the seed $\mathcal{A}_0 = \mathcal{S}$, step and time are set to zero, i.e., $i = 0$ and $t = 0$ (line 1). After the initialisation, the self-assembly process consists of the infinite repetition of two possible actions with their probability based on the assembly in the current step $\mathcal{A}_i$ and on parameters of the system G_{mc}, G_{se}, and k_f. The first possible action is called as *"on" action*, which represents an attachment of a tile to the assembly, where all tiles from T are equally likely to be attached to the assembly to any equally possible position around the assembly. The second action is referred to as *"off" action*, which signals a detachment of a tile from the assembly, where all tiles from the assembly have an equal probability of dissociation.

The computation of *the rate of the "on" action* k_{on} is demonstrated in lines 3–5 and it can be written as

$$k_{on} = m \cdot k_f \cdot e^{-G_{mc}},$$

where m is the number of *frontier* positions of an assembly (all unoccupied positions around the perimeter of the assembly). Therefore, the rate of the "on" action is proportional to the circumference of the assembly in the current step.

```
procedure: kTAM-Assembly-Sequence

Inputs: Number of steps $n$,
        Tile Assembly System $\mathscr{T} = (T, \mathcal{S}, s, G_{mc}, G_{se}, k_f)$,
        where $T$ is the set of tiles, $\mathcal{S}$ is seed structure, $s$ is the glue strength
        function, $G_{mc}$ is the entropic cost of fixing the location of a monomer tile,
        $G_{se}$ is the free energy cost of breaking a strength-1 glue bond, and $k_f$ is
        the forward rate constant.
Outputs: Assembly in $n$-th step $\mathcal{P}_n$

$\mathcal{A}_0 := \mathcal{S};\ i := 0;\ t := 0$
repeat
    $F :=$ frontier_positions($\mathcal{A}_i$)
    $m := |F|$
    $k_{on} = mk_f e^{-G_{mc}}$
    $P := \{\mathcal{A}_i(\vec{p}) : \forall \vec{p} \in \{\mathbb{Z}^2 \setminus (0, 0)\}\}$
    $k_{off,b} := 0$
    for each position $\vec{p}$ in $P$ do
        $b_{\vec{p}} :=$ total_strength($\mathcal{A}_i(\vec{p})$)
        $k_{off,b_{\vec{p}}} = k_{off,b_{\vec{p}}} + k_f e^{-b_{\vec{p}} G_{se}}$
    $k_{off} := \sum_b k_{off,b}$
    $k_{any} := k_{on} + k_{off}$
    $\Delta t \sim \Pr(\Delta t) = k_{any} e^{k_{any} \Delta t}$
    wait $\Delta t$ seconds
    $t := t + \Delta t$
    $x := \Pr(\text{action is "on"}) \sim B(1, k_{on}/k_{any})$
    if x is true then
        $t_e :=$ uniform_random($T$)
        $\vec{p_e} :=$ uniform_random($F$)
        $\mathcal{A}_{i+1} := \mathcal{A}_i \cup (t_e, \vec{p_e})$
    else
        $P' :=$ outer_edge_positions($\mathcal{A}_i$)
        $\vec{p_d} :=$ uniform_random($P'$)
        $\mathcal{A}_{i+1} := \mathcal{A}_i \setminus \{\vec{p_d}\}$
    $i := i + 1$
until $i > n$
$\mathcal{P}_n := \mathcal{A}_i$
return $\mathcal{P}_n$
```

Algorithm 2.2. Pseudo-code description of the assembly process of the kTAM system, where function `frontier_positions`($\mathcal{A}$) returns the set of all frontier positions of the assembly $\mathcal{A}$ (unoccupied positions around the perimeter of the assembly), function `outer_edge_positions`($\mathcal{A}$) returns the set of all positions of an outer edge of the assembly $\mathcal{A}$ and function `total_strength`(t) returns the combined glue strength provided to the tile t by the assembly $\mathcal{A}_i$. Furthermore, $x =$ `uniform_random`(X) is a function which returns a random element x from given set X and where all elements are equally likely to be chosen, Pr is probability and $B(n, p)$ denotes *Binomial distribution*, where n is the number of trials and p is the probability of success.

The *rate of the "off" action* k_{off} is based on the total bond strength of each tile in the assembly, which is illustrated by iterative computation on lines `6`–`11` or it can be expressed with sums as

$$k_{off} = \sum_{b} k_{off,b}, \quad k_{off,b} = \sum_{\vec{p} \in \{\mathcal{A}_i \setminus (0,0)\} \text{ s.t. } b_{\vec{p}} = b} k_f \cdot e^{-b_{\vec{p}} G_{se}},$$

where b denotes the number of strength-1 glue bonds, therefore $b_{\vec{p}}$ is the number of matching strength-1 glue bonds of the tile placed at position $\vec{p} \in P$, and P is a set of any defined position in the current step assembly $\mathcal{A}_i$ except the seed tile at the origin of the assembly plane. Additionally, $k_{off,b}$ is the rate of the "off" action for tiles with total bond strength equal to b. Note that if the tile has matching strength-2 glue bond with its neighbouring tile, it is counted as two strength-1 bonds in b.

From these action rates, *the net rate of any action occurring* k_{any} is obtained as $k_{any} = k_{on} + k_{off}$ (line `12`). Now, the time to the next action Δt can be calculated according to *the Boltzmann distribution* as

$$\Pr(\Delta t) = k_{any} \cdot e^{k_{any} \cdot \Delta t}.$$

Given the time Δt has passed, and the next action is about to take place (lines `13`, `14`), the probability of choosing the "on" action is k_{on}/k_{any}. Whether the "on" action will be executed can be determined by generating a number x from *the Binomial distribution* $B(n, p)$ as

$$x = \Pr(\text{action is "on"}) \sim B\left(1, \frac{k_{on}}{k_{any}}\right),$$

where n is the number of trials and p is the probability of success (line `16`). If the generated number x equals to 1 (*true*), the "on" action occurs (lines `17`–`20`), otherwise the "off" action follows (lines `21`–`24`). In the next step $i+1$, all rates must be recalculated to determine the action.

2.2.3 Two-Handed Assembly Model

The *Two-Handed Assembly Model* (*2HAM*) is a generalisation of the aTAM where every two producible assemblies are allowed to combine in parallel [44–47]. The 2HAM is not limited to only one seeded assembly but models every possible assembly which can be created from the given tile set. This behaviour of the system models spontaneous nucleation of tiles in a solution where an infinite number of tiles floats and merge one to another. Note that 2HAM has many different names in the literature; for example, it is referred to as *hierarchical self-assembly*, *polyominoes*, or *recursive aTAM*.

One of the interesting properties of the 2HAM system is that it is not intrinsically universal [48] as there is always some system with temperature $\tau + 1$ which cannot be simulated by a system with temperature τ. However, it was shown in [48] that 2HAM is intrinsically universal if the higher temperature class systems are ignored. Therefore, there exists a universal tile set in a system with temperature τ, which can be used to simulate every other system in the class of systems with equal to or lower than temperature τ.

The 2HAM system is defined as ordered triplet $\mathscr{T} = (T, s, \tau)$, where parameter T is the set of tiles acting in the system, s is the glue strength function, and τ is the system temperature. The 2HAM uses so-called *supertiles* to describe its assembly sequence. A *supertile* α is positioning of ordinary tiles on the integer lattice $\mathbb{Z}^2$. Technically, the supertile has the same definition as an assembly in Definition 2.8. Therefore, the supertile describes an assembly consisting of one or more tiles and thus provides a handle for a more straightforward description of the model.

One slight difference between the usage of an assembly and a supertile is that the absolute position of singleton tiles is not considered significant in supertile, because two supertiles which differ only in the translation are considered equivalent. Therefore, only the relative configuration of tiles is sufficient to define the whole class of *similar* supertiles.

Definition 2.23. It is said that, two supertiles α and β are *similar* or α *is a translation of* β, which is denoted by $\alpha \simeq \beta$ if

$$\alpha \simeq \beta \iff \exists \vec{u} \in \mathbb{Z}^2 : \alpha + \vec{u} = \beta.$$

Hence the term supertile can be used to refer to the set of all translations of one particular assembly $\bar{\alpha} = \{\beta | \alpha \simeq \beta\}$.

The assembly process of 2HAM starts without a seed structure, which means that each unit tile $t \in T$ serves as a seed on its own and it is a supertile of size 1. Then, in every step of the assembly process, every *union* of two supertiles existing in the current assembly sequence step is tested whether it can τ-stably form a new supertile by translating one of the supertile, so they are not overlapping. If two supertiles do not overlap, it is said that they are *disjoint*. The definition of τ-stable connection of two supertiles is slightly different from the general definition for singleton tiles, and the definition is presented with other 2HAM terms in the following definitions.

Definition 2.24. Two supertiles α and β are called *disjoint* if $\operatorname{dom} \alpha \cap \operatorname{dom} \beta = \emptyset$.

Definition 2.25. The combination of two supertiles α and β is called a *union* denoted by $\alpha \cup \beta$, if

$$(\alpha \cup \beta)(\vec{p}) = \begin{cases} \alpha(\vec{p}) & \text{if } \alpha(\vec{p}) \text{ is defined} \\ \beta(\vec{p}) & \text{if } \beta(\vec{p}) \text{ is defined} \end{cases}, \ \forall \vec{p} \in \mathbb{Z}^2.$$

Additionally, it is said that the union is *disjoint* if both α and β are disjoint.

Definition 2.26. The supertile is *τ-stable* if every cut of its grid graph $G = (V, E)$ has weight at least τ, where vertices V of the grid graph are tiles with positions and edges E are the matching glues between the adjacent tiles with weight equal to the strength of the glue according to glue strength function s. That is another way of saying that τ-stable supertile cannot be broken into two sub-assemblies without breaking glue bonds of total strength greater than or equal to temperature τ. It can be noted that this definition enables for a mismatched tile to be added to an assembly if it receives enough bonding strength from other tiles with correct bonds. Therefore, the union of two supertiles $(\alpha \cup \beta)$ is τ-stable if there exists a translation vector $\vec{u} \in \mathbb{Z}^2$ such that $\alpha + \vec{u}$ and β are disjoint and share a common interface with glue bond strength equal to or greater than τ.

An outline of the 2HAM system assembly sequence is presented in the following text, together with an example of a simple system assembly.

In the Algorithm 2.3, it can be seen that the assembly process of the 2HAM system is initialised by promoting all singleton tiles from set T to supertiles and storing them in the set C_0 (line `1`). Then, all pairs of supertiles that can be τ-stably united are searched until there is no new supertile found. The search procedure consists of several steps.

First, every unique pair of supertiles $(\alpha, \beta) = (\beta, \alpha)$ is found (line `4`). Then, set of all translation vectors $U \subseteq \mathbb{Z}^2$ which place supertile α to be *disjoint* with supertile β while at least one tile from α is in the four-neighbourhood of at least one tile from β

```
procedure: 2HAM-Assembly-Sequence

Inputs: Number of steps $n$,
        Tile Assembly System $\mathscr{T} = (T, s, \tau)$,
        where $T$ is the set of tiles, $s$ is the glue strength function, and $\tau$ is the
        system temperature.
Outputs: Set of supertiles in $n$-th step $P_n$

1   $C_0 := \{((0, 0), t) : \forall t \in T\}$;   $i := 0$
2   repeat
3       $C' := \emptyset$
4       for each pair of supertiles $(\alpha, \beta) \in C_i$ do
5           $U :=$ disjoint_translations$(\alpha, \beta)$
6           for each translation vector $\vec{u}$ in $U$ do
7               $\gamma := (\alpha + \vec{u}) \cup \beta$
8               if ($\gamma$ is $\tau$-stable) and ($\gamma \not\simeq \varepsilon$, $\forall \varepsilon \in (C_i \cup C')$) then
9                   $C' := C' \cup \{\gamma\}$
10      if $C' = \emptyset$ then
11          $P_n := C_i$
12          return $P_n$
13      $C_{i+1} := C_i \cup C'$
14      $i := i + 1$
15  until $i > n$
16  $P_n := C_i$
17  return $P_n$
```

Algorithm 2.3. A pseudo-code description of the assembly process of the 2HAM system $\mathscr{T}$, where the function `disjoint_translations`(α, β) returns a set of all unique translation vectors U such that translated supertile $(\alpha + \vec{u})$, $\vec{u} \in U$ and supertile β are disjoint and share a common interface.

are found, i.e., α and β shares a common interface (line 5). All such translation vectors can be obtained by placing each tile on the outer edge of supertile α to every position in the frontier around the supertile β and save the difference between origins of both supertiles. Note that it does not matter whether the supertile α is translated around the supertile β or the other way around.

Next, new supertile $\gamma = (\alpha + \vec{u}) \cup \beta$, $\vec{u} \in U$ is formed (line 7). Finally, if the supertile γ is τ-stable and does not already exist in the system, it is added to the set of newly found supertiles C' in the current step (lines 8, 9). If there are no two supertiles that could be τ-stably united in the system, i.e, $C' = \emptyset$, the assembly sequence terminates (lines 10, 11), otherwise all new supertiles from C' are added to the set of producible supertiles from the former steps C_i of the assembly sequence, which creates the set of producible supertiles in the following step C_{i+1} (line 12). From this point, the search process is repeated using the newly found supertiles in the next step $i + 1$.

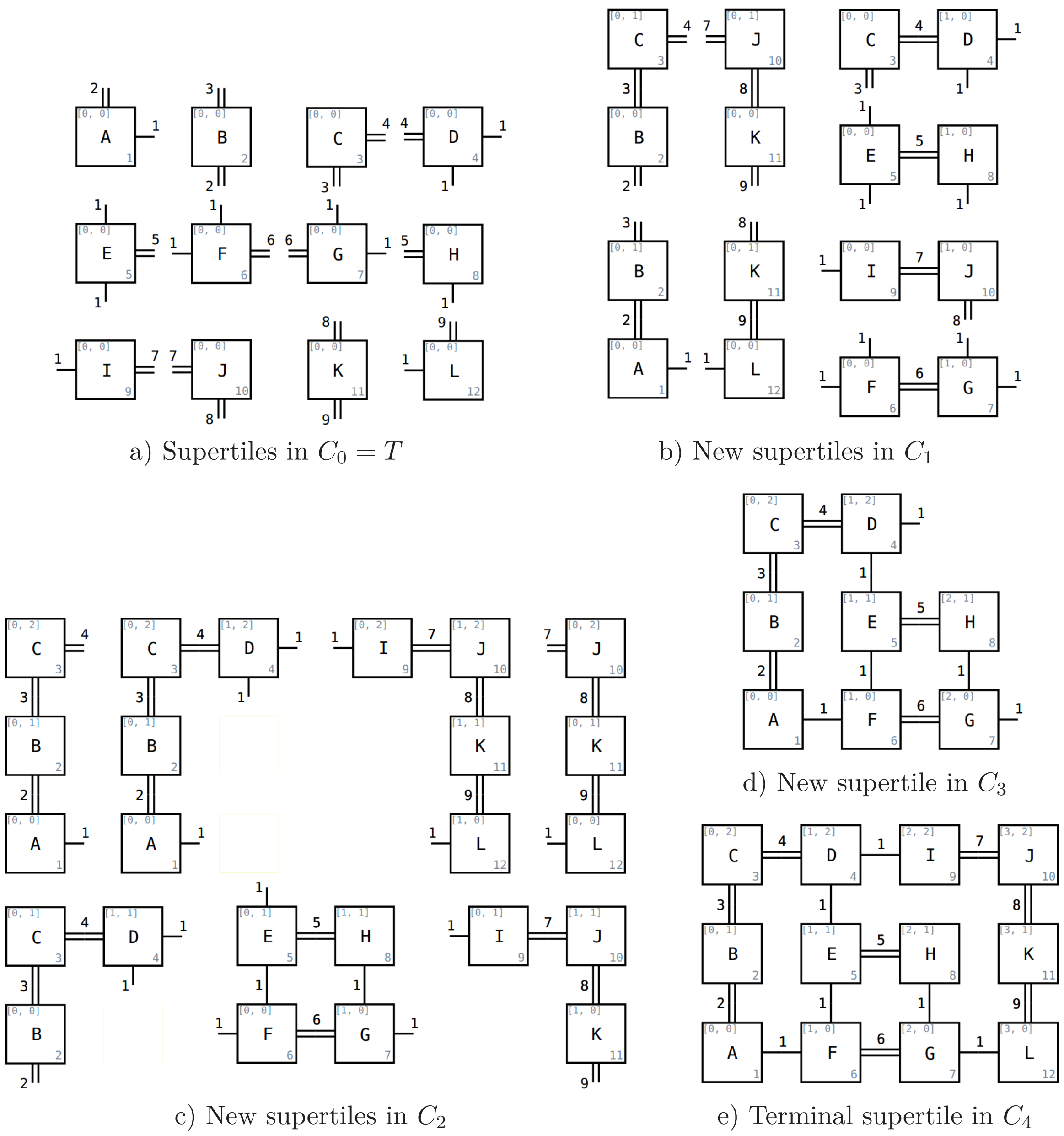

a) Supertiles in $C_0 = T$
b) New supertiles in C_1
c) New supertiles in C_2
d) New supertile in C_3
e) Terminal supertile in C_4

Figure 2.4. Assembly sequence of example 2HAM system $\mathscr{T} = (T, s, \tau = 2)$, where C_i is the set of all supertiles produced by the system from the start to the i-th step of the assembly.

Example 2.2. *2HAM: Assembly sequence*
In Figure 2.4, an example of an assembly sequence of 2HAM system $\mathscr{T} = (T, s, \tau = 2)$ is shown. The tile set $C_0 = T$ is illustrated in Figure 2.4a. In the first step, all possible τ-stable pairs of supertiles are formed, which can be seen in Figure 2.4b. Now, the pairing process is repeated using all supertiles in the system, including supertiles found in the last step. The newly found supertiles in steps 2, 3 and 4 of the assembly sequence are presented in Figures 2.4c, d, and e, respectively. The assembly sequence of this particular system terminates in step 4 with an assembly which spells out the alphabet in a "zig-zag" fashion.

2.2.4 Staged Self-Assembly Model

This section presents the basic idea behind another self-assembly model variant called *the Staged Self-Assembly* developed by Demaine [45, 30]. The Staged Self-Assembly model is a further generalisation of the 2-Handed Assembly Model. In other models, the set of tiles which are used to assemble the system is considered constant, and the whole self-assembly process happens in one configuration space or *bin*. The Staged Self-Assembly enables to self-assemble more than one system in different bins in parallel, filter their produced assemblies and use them to create new self-assembly system in the next *stage*. In every bin, the system assembly is modelled as the 2HAM system. This hierarchy of different mixing bins allows dramatically smaller tile sets to assemble unique target shapes in return for the higher complexity of handling the mixture of the bins. Additionally, this setting allows that each bin can have a different temperature τ in different stages of assembly.

It was proved that the Staged Self-Assembly along with *the Step-Wise Self-Assembly* model are both Turing universal at temperature $\tau = 1$ [49]. The mentioned Step-Wise Self-Assembly model is an older analogue of the Staged model with only one bin and additional constraints for the assembly growth similar to the aTAM [50].

A *Staged Assembly System* (*SAS*) is defined as 4-tuple of parameters $\mathscr{T} = (\{T_{i,j}\}, \{\tau_{i,j}\}, M_{r,b}, s)$, where $M_{r,b}$ is the directed mixing graph of r number of stages with b number of bins, $\{T_{i,j}\}$ is specification of sets of tiles in $i \in \langle 1; r\rangle$ stages and $j \in \langle 1; b\rangle$ bins, $\{\tau_{i,j}\}$ are values of temperature for corresponding bins, and s is the glue strength function. Intuitively, the mixing graph specifies how each collection of bins should be mixed together when transitioning from one stage to the next. An example of mixing graph with three stages and two bins can be seen in Figure 2.5, where nodes $m_{i,j}$ denote j-th bin in i-th stage of the assembly and m_* denotes the final mixing node (bin). In the example, the first stage consist of two parallel assemblies in bins $m_{1,1}$ and $m_{1,2}$ which are then mixed together in the second stage bin $m_{2,1}$. In the third stage, the produced assemblies from the second stage are putted into two separate bins $m_{3,1}$ and $m_{3,2}$, where the assembly is mixed with additional tiles. Finally, produced assemblies from the third stage are mixed in the final mixing bin m_*, where the desired assembly is constructed.

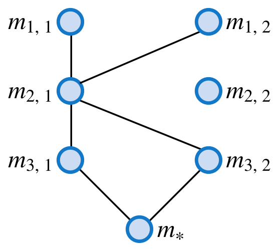


Figure 2.5. Example of a mixing graph used in the Staged Self-Assembly which has three stages and two bins.

In the assembly process, the individual bin systems are denoted as $(R_{i,j}, \tau_{i,j})$, where $R_{i,j}$ is the tile set used in the specific bin which corresponds with the node of the mixing graph $m_{i,j}$. Tile sets used in bins for the first stage are defined as $R_{1,j} = T_{1,j}$, tile sets for subsequent stages are defined as

$$R_{i,j} = \left(\bigcup_{k:(m_{i-1,k},\, m_{i,j}) \in M_{r,b}} \text{Term}\,(R_{i-1,k},\, \tau_{i-1,k}) \right) \cup T_{i,j}, \quad \text{for } i \geq 2,$$

where Term (R, τ) is the set of terminal assemblies of the bin system (R, τ) (Definition 2.16). The final tile set, where all products from the previous stages in the assembly sequence melt together, is defined as

$$R_* = \left(\bigcup_{k:(m_{r,k}, m_*) \in M_{r,b}} \text{Term}\,(R_{r,k}, \tau_{r,k}) \right).$$

Therefore, the j-th bin in i-th stage is created by mixing terminal assemblies of one or more bins from previous stage together according to the mixing graph and possibly introducing new tiles from set $T_{i,j}$ to the system. The desired assembly is then produced in the last mixing bin.

2.2.5 Overview of Other Model Variants

Basic models presented above provide an excellent foundation for theoretical and experimental research, which in turn helped to understand the powers and limits of the models further. However, these theoretical models have restrictions that do not allow for complete generality of the models in some areas of research. Therefore, many different models were developed or derived from the basic ones to capture particular aspects of self-assembly and study the behaviour of such systems, which can lead to the development of new self-assembly techniques. In this section, an incomplete list of such novel models is presented with brief high-level descriptions of ideas behind them.

One possible extension of the aTAM is to enable to set different probabilities for tiles in the system, and this is called a *probabilistic assembly* or *concentration programming*. By changing the concentration of a particular tile type in the system, the tile is more likely to be attached to the assembly more frequently. The *Probabilistic Tile Assembly Model (PTAM)* was developed in [51], and the idea of the *tile concentration programming* originated in [52].

Another class of models studied systems where the tiles used in the system are different. For example, the tile can have a different shape. Systems with triangular and hexagonal tiles were studied in [53]. Another variant of tiles which has glues attached on flexible arms was introduced in [54]. Similarly, it can be allowed for a tile to rotate during the self-assembly. The set of single rotatable tile can simulate any other aTAM system was presented in [55].

All mentioned models were more or less static in the sense that once a tile was attached to the assembly, it stayed there to the end of the assembly sequence. The following several models explore more dynamic behaviour of the system, and so they enable a tile to detach an assembly, changing global parameters of the system or changing properties of a single tile type. One way to alter the global parameter of the system is the *temperature programming* [44, 56]. As the name suggests, it enables to change system temperature during the assembly process, and it was shown that it is quite a powerful method to self-assemble arbitrary shapes. Another approach is to enable *repulsive glues* or *negative glues*, which changes the interaction between tiles in the system since the repulsive tile interaction is introduced. Systems with repulsive glues were studied in [57], and the *restricted glue Tile Assembly Model (rgTAM)* was defined in [58]. At last, another way of changing the behaviour of the system during the assembly is to let tiles to pass signals one to another and then, they change their glues based on the received signals. There are two similar models developed, the *Signal passing Tile Assembly Model (SPTAM)* introduced in [59–60], and the *Active Tile Assembly Model (ATAM)* presented in [61–62].

2.3 Tile Set Synthesis Problems

In the following text, two optimisation problems of the tile self-assembly are described. The optimisation of a tile assembly system can have different goals according to the model or the usage of the system. General minimisation criterion is the number of tiles in the system tile set. Another optimisation is a minimisation of the number of unique glues within the system. Note that the minimal number of glues in the system does not necessarily minimise the number of tiles needed for the desired assembly. Even though the smaller number of glues, the smaller the number of possible combinations which could form a tile, it does not generally result in the smaller tile set. Lastly, it is possible to optimise the time required for a tile assembly system to self-assemble by setting concentrations of each tile in the system tile set.

The problem of finding a minimal tile set to self-assemble arbitrary finite shape is presented in the following Section 2.3.1, and the problem of generating a minimal tile set to self-assembly given pattern is defined in Section 2.3.2. Please note, that the problem of shape self-assembly is not the main focus of this thesis as it is the pattern self-assembly and so the following description of the problem is rather brief, and it is included herein only for the sake of completeness.

2.3.1 The Shape Assembly Problem

The problem of synthesising minimal tile set for self-assembly of a given shape is called *The Minimum Tile Set Problem* in the literature, and it has been well studied for various tile assembly models. First, the formal description of the problem is shown, followed by a short list of published solutions.

Problem formulation

The Minimum Tile Set Problem is defined by Adleman [40] as finding the tile system with a minimum number of tile-types that can uniquely self-assembly into a given *shape*. The term *shape* is used to denote a connected grid-graph $A = (V, E)$, where positions of the graph vertices $V \in \mathbb{Z}^2$ are used to define the shape. It is said that an assembly $\mathcal{A}$ *strictly self-assembles* a shape A if $\mathcal{A}$ strictly self-assembles V according to Definition 2.20. Note that two shapes are considered equal if they can be made identical by translation.

Definition 2.27. *Minimum Tile Set Problem:* Given a shape A and a temperature τ, find TAS $\mathscr{T} = (T, \mathcal{S}, s, \tau)$ such that $|T|$ is minimal and $\mathscr{T}$ *uniquely produces* $\mathcal{P}$ which *strictly self-assembles* A.

Note that the problem does not specify the particular model of TAS $\mathscr{T}$ used for the solution, and an algorithm solving the problem for one TAS model does not transfer well to other models due to their different properties. Therefore, this problem is solved for each model separately.

Overview of the solutions

The techniques of designing tile sets for strict self-assembly of shape for the aTAM and the kTAM were developed in [12, 40, 52, 42, 58, 35, 63–65]. The synthesising techniques for 2HAM systems are exhibited in [47, 46]. The Staged Self-Assembly model has proven itself to be very powerful in assembling shapes, which was shown in [45, 30, 33].

2.3.2 The Patterned Self-Assembly Problem

The problem of designing a tile set that can self-assemble a given pattern is explored in this section. First, a description of the problem, together with a formal definition is declared. After that, a brief overview of related research, along with a short description of developed solutions is presented. In the end, one of the algorithms for solving the problem is further explained.

Problem formulation

The Pattern self-Assembly Tile set Synthesis problem also referred to as *PATS problem* was first defined by Ma and Lombardi [66]. The goal is to find a way to systematically generate a tile set which can self-assemble into a rectangle with the desired 2D *pattern*. A *pattern* can be imagined as a digital picture, where each coloured pixel has its designated position within the image. Consequently, the goal of a tile with a particular colour (or another distinct feature) is to attach itself exactly on those positions where pixels with the corresponding colour lie. The straightforward solution is to generate a unique tile type for every point (pixel position) in the pattern, but this approach is wasteful, and it would make sense only if there were no two points with the same colour in the pattern because then it would be the only solution. However, patterns with a practical purpose often have an only small number of colours relative to the number of points in the pattern. Therefore, methods for optimising the size of the tile set are needed. Furthermore, note that solutions of the PATS problem are primarily designed for usage within the aTAM or kTAM models.

It is needed to define several additional terms before the formal definition of PATS problem can be given.

Definition 2.28. *Dimensions of a pattern* are described by two natural numbers $m, n \in \mathbb{N}$, where m is the number of points in every row of the pattern (width), and n is the number of points in every column of the pattern (height).

Definition 2.29. Let a mapping from $[m] \times [n] \to [k]$ be a *k-colouring* or *k-coloured pattern* for fixed dimensions m, n, where $[m], [n] \subset \mathbb{N}$ denote sets $\{1, 2, \dots, m\}$ and $\{1, \dots, n\}$, respectively, and therefore $[m] \times [n] \subset \mathbb{Z}^2$ is the set of all points in the pattern, and $[k]$ is the set of k-number of colours used in the pattern.

Definition 2.30. *Pattern self-Assembly Tile set Synthesis (PATS) problem*: Given a *k-coloured pattern* $c : [m] \times [n] \to [k]$, find a *tile assembly system* $\mathscr{T} = (T, \mathcal{S}, s, 2)$ with the following properties

(i) The tiles in T have only glues with bonding strength 1.
(ii) The domain of the seed $\mathcal{S}$ is $(\langle 0, m\rangle \times \{0\}) \cup (\{0\} \times \langle 0, n\rangle)$ and all the terminal assemblies have the domain $\langle 0, m\rangle \times \langle 0, n\rangle$.
(iii) There exists a colouring $d : T \to [k]$ such that for each terminal assembly $\mathcal{A} \in \text{Term}\ \mathscr{T}$ it is $d(\mathcal{A}(x, y) = c(x, y)$ for all $(x, y) \in [m] \times [n]$.

Especially interesting are the *minimal solutions* for the PATS problem in terms of $|T|$. Therefore, a very reasonable assumption is that every tile of the synthesised system $t \in T$ must participate in self-assembling at least one terminal configuration of the system Term $\mathscr{T}$, because otherwise the tile t could be removed from the tile set T and the system behaviour would not change.

Note that the PATS problem is explicitly defined for a TAS $\mathscr{T}$ with temperature $\tau = 2$. The constrained temperature τ together with the first property of the PATS solving TAS $\mathscr{T}$ (Definition 2.30(i)) forces the tiles to *cooperate* during the self-assembly process. As mentioned before, this means that a tile can attach to the assembly only on such position where at least two tiles are already placed in the four-neighbourhood, because every tile in T can have only glues with maximum strength 1. This support must be provided by the seed structure $\mathcal{S}$ in order for the first tile to attach itself to the assembly. In the PATS solutions, the seed is usually a "L-shaped" structure which is described by the second property of $\mathscr{T}$ (Definition 2.30(ii)). The seed consists of $m + n + 1$ unique tiles with strength-2 glues such that the seed could uniquely self-assemble itself. Note that it does not matter whether the seed is defined as fully formed at the beginning of the assembly process or if it is defined as single tile in the origin of the assembly space, as it will self-assemble.

If both (i) and (ii) properties of the $\mathscr{T}$ are taken into account, it can be derived that every tile can be joined to the assembly on position $\vec{p} \in [m] \times [n]$ only if tiles positioned to the south $S(\vec{p})$ and the west $W(\vec{p})$ of the position are both already placed in the assembly. Therefore, the assembly is extended from the south-west corner to the north-east corner of the pattern during the self-assembly sequence. Note, that the positions of cooperating neighbours are defined by the domain of the seed $\mathcal{S}$ in (ii), for example, if the cooperating tiles should be on the north side and the west side of every attaching tile then the seed $\mathcal{S}$ should have the domain $(\langle 1, m+1\rangle \times \{m+1\}) \cup (\{0\} \times \langle 0, n\rangle)$, i.e., the seed would have an "Γ-shape".

Another reasonable assumption is to only consider the *locally deterministic* tile assembly systems according to Definition 2.18 as the set of solutions of the PATS problem because otherwise, the system cannot reliably terminate the self-assembly process with the configuration of the given pattern. Since the properties of the PATS solution system are quite specific and also the system must be deterministic, an alternative definition of system determinism if formed in the following definition.

Definition 2.31. A solution TAS $\mathscr{T} = (T, \mathcal{S}, s, \tau)$ of the PATS problem *is deterministic* precisely when for each ordered pair of glue types $(\sigma_1, \sigma_2) \in \Sigma^2$ there is at most one tile $t \in T$ such that $(\sigma_S(t) = \sigma_1) \wedge (\sigma_W(t) = \sigma_2)$.

The definition of *deterministic* $\mathscr{T}$ dramatically reduces the computation work needed for verification of solutions to the PATS problem.

Overview of the solutions

As mentioned above, the PATS problem was first defined as a combinatorial optimisation problem in [66], where the authors also proposed two greedy algorithms for finding solutions, called `PATS_Bond` and `PATS_Tile`, which encapsulates an algorithm for minimising the number of glues in the system or the number of tiles, respectively. Then, Göös and Orponen [39] continued, and they developed an exhaustive algorithm called *PS-BB* which stands for *partition-search branch-and-bound*. The PS-BB algorithm uses a *partition framework* to search for the solution in the space of partitions. Because of the exhaustive nature of the PS-BB algorithm combined with the `NP`-hardness of the PATS problem [67–68], the PS-BB algorithm renders feasible only for small patterns in size of approximately 7×7 tiles (points). Since the PS-BB algorithm searches for the minimal solution and frequently a small but not necessarily minimal solution is sufficient; therefore, a greedy heuristic version of the PS-BB algorithm was developed by Lempiäinen et at. called *PS-H* (*Partition-Search with Heuristics*) [69]. The PS-H algorithm uses a series of heuristics to guide the minimisation in the search space. Additionally, there were attempts to solve the PATS problem using the *Answer Set Programming* framework (*ASP*) in [70].

Partition search with heuristics (PS-H)

In this section, the PS-H algorithm developed in [69] is further described. As mentioned above, both PS-BB and PS-H algorithms are built on searching in the space of *partitions of sets*. A *partition of a set* X (also *partition* or *partitioning*) is a set of non-empty subsets of X such that every element $x \in X$ is in exactly one of these subsets, i.e., X is a disjoint union of the subsets. A subset of the partition can also be referred to as a *class*. Additionally, a partition of a pattern is assumed to be the partition of the set $[m] \times [n]$, where m, n are the dimensions of the pattern, and X denotes the set of all partitions of the set $[m] \times [n]$ in this section.

Definition 2.32. For two partitions P, $P' \in X$ define relation "$\sqsubseteq$" as

$$P \sqsubseteq P' \iff \forall p' \in P' : \exists p \in P : p' \subseteq p$$

and for $P \sqsubseteq P'$ is said that P' is a *refinement* of P, and conversely, P is *coarser* then P'. It can be seen that $P \sqsubseteq P'$ implies $|P| \leq |P'|$, which means that number of classes of coarser partitioning P is less than or equal to the number of classes of the more refined partitioning P'. The pair $(X, \sqsubseteq)$ together forms a partially ordered set and in fact a lattice, which is the search space of the PS-BB and PS-H algorithms.

Quite naturally, a k-coloured pattern c induces a partitioning of the pattern $P(c) = \{c^{-1}(\{i\}) \mid i \in [k]\}$, hence $|P(c)| \equiv k$. Likewise, let $P(\mathscr{T})$ denote a partition of an assembly $\mathcal{A} \in \text{Term}\ \mathscr{T}$ which is uniquely and deterministically produced by the TAS $\mathscr{T}$, and it is defined as

$$P(\mathscr{T}) = \{\mathcal{A}^{-1}(\{t\}) \mid t \in \mathcal{A}([m] \times [n])\}.$$

Thus by using the partitioning framework and the properties (i) and (ii) in Definition 2.31, a equivalence $|P(\mathscr{T})| \equiv |T|$ can be stated. Additionally, the property (iii) of $\mathscr{T}$ in Definition 2.31 can be rephrased using the partitions as

$$P(c) \sqsubseteq P(\mathscr{T}) \Rightarrow |P(c)| \leq |P(\mathscr{T})|,$$

which can be read as the terminal assembly partition $P(\mathscr{T})$ is refinement of the pattern partition $P(c)$, and therefore the minimal number of tiles in the system $|T|$ is the number of colours k in the pattern c.

Definition 2.33. A partition $P \in X$ is *constructible* if $P = P(\mathscr{T})$ for some deterministic TAS $\mathscr{T}$ according to Definition 2.31, and $\mathscr{T}$ satisfies the properties (i) and (ii) of Definition 2.30.

Now, it can be stated that an *optimal* solution of the PATS problem corresponds to a partition $P \in X$ such that P is *constructible*, $P(c) \sqsubseteq P$ and $|P|$ is minimal.

Definition 2.34. A *Most General Tile Assignment* (`MGTA`) is a mapping function $f : P \to \Sigma^4$, where P is a given partition of the set $[m] \times [n]$, such that

(i) f *is consistent*: when position in $[m] \times [n]$ is assigned with tile type according to f, any two adjacent positions must have matching glues along their abutting side.

(ii) f *is minimally constrained*: any function $g : P \to \Sigma^4$ satisfying (i) satisfies also

$$\sigma_{D_1}(f(p_1)) = \sigma_{D_2}(f(p_2)) \;\Rightarrow\; \sigma_{D_1}(g(p_1)) = \sigma_{D_2}(g(p_2)),$$

for all partition classes p_1, $p_2 \subset P$ and directions D_1, $D_2 \in \mathcal{D}$.

Since all the important definitions are now given the partition search can be further described in detail.

```
function: PS-H

Inputs: $k$-coloured pattern $c$, where
        $k$ is the number of colours in the pattern,
        $m$ is the width of the pattern,
        $n$ is the height of the pattern.
Outputs: Tile Assembly System $\mathscr{T} = (T, \mathcal{S}, s, \tau = 2)$,
        where $T$ is the set of tiles, $\mathcal{S}$ is the seed structure, $s$ is the glue strength
        function, and $\tau$ is the system temperature.

1   $P := \{\{p\} \mid p \in [m] \times [n]\}$
2   $f :=$ MGTA$(P)$
3   $H := \{\{p, q\} \mid p, q \in P, p \neq q, \exists k \in P(c) : p, q \subseteq k\}$
4   repeat
5       $K := H$
6       $K := \{\{p, q\} \in K \mid G(p, q) \geq G(u, v), \forall\{u, v\} \in K\}$
7       $K := \{\{p, q\} \in K \mid \max\{|p|, |q|\} \geq \max\{|u|, |v|\}, \forall\{u, v\} \in K\}$
8       $K := \{\{p, q\} \in K \mid \min\{|p|, |q|\} \geq \min\{|u|, |v|\}, \forall\{u, v\} \in K\}$
9       $\{a, b\} :=$ uniform_random$(K)$
10      $P', f' := P[a, b]$
11      while $P'$ *is not constructible* do
12          $U := \{\{u, v\} \mid (\sigma_S(u) = \sigma_S(v)) \wedge (\sigma_W(u) = \sigma_W(v)), \ldots$
                $\ldots u \neq v, u, v \in P'\}$
13          $\{u, v\} :=$ first$(U)$
14          $P', f' := P'[u, v]$
15      if $P' \sqsubseteq P(c)$ then
16          if $|P'| < |P|$ then
17              $P := P'; f := f'$
18      $H := H \setminus \{\{a, b\}\}$
19  until $H = \emptyset$
20  $T := \{\{f(p)\} \mid \forall p \subseteq P\}$
21  $\mathcal{S} :=$ generate_seed$(P, f)$
22  return $\mathscr{T} = (T, \mathcal{S}, s, 2)$
```

Algorithm 2.4. A pseudo-code description of the PS-H algorithm for solving PATS problem, where $x =$ `uniform_random`(X) is function which returns a random element x from a given set X and where all elements are equally likely to be chosen, and $x =$ `first`(X) is function which returns the first element x from a given set X.

The PS-H algorithm is described in Algorithm 2.4. The algorithm starts with the initial partition P which is the most refined partition of the set $[m] \times [n] \in \mathbb{Z}^2$, i.e., every point in the discrete rectangle is in its own class $p \in P$, where m, n are the dimensions of the input k-coloured pattern c (line 1). Then, the *Most General Tile Assignment* function f is created using the initial partition P which provides a mapping between the partition P and a collection of tile types (line 2), and it is defined as follows.

So, the algorithm starts by setting a unique tile on each point in the pattern such that their glues match with the adjacent tiles, and this is stored as a partitioning P and *tile map* function f which maps the classes in P to tile types. The goal of the rest of the algorithm is to *merge* tile types in order to get the optimal (minimal) solution. It only makes sense to merge such tile types that have the same colour in the desired pattern c, because otherwise, it would be impossible to assemble the pattern. Therefore, the next step is to form the set of all unordered pairs of positions where both elements of the pair have the same colour in the desired pattern, and this set is denoted as H on line 3 in the algorithm. Each pair of partition classes in H represents a candidate partitioning which is created by merging two elements in the pair to a single class and thus reducing the tile assembly system by one tile type. If the candidate partition is constructible, it is one of the valid solutions of the problem. The algorithm merges the pairs of tile types to search the solution-space guided by a series of heuristics until it exhausts the set H.

The process of selecting a class pair for merging starts by filtering the set H with heuristics into the set K (lines 5–8). The goal of the heuristics is to pick such pairs that have the highest probability of forming a valid constructible solution. The first heuristic on line 6 selects such pairs of tile types that have the most common glues, which is denoted by the function $G : P \times P \to \{0, 1, 2, 3, 4\}$ defined as

$$G(p, q) = \sum_{D \in \mathcal{D}} g\left(\sigma_D(f(p)), \sigma_D(f(q)\right),$$

where the function g is defined as

$$g(\sigma_1, \sigma_2) = \begin{cases} 1, & \text{for } \sigma_1 = \sigma_2 \\ 0, & \text{otherwise} \end{cases}, \ \sigma_1, \sigma_2 \in \Sigma,$$

p, q are classes in partition P and the function f maps partition classes to tile types. The following two heuristics on lines 7 and 8 picks the pairs of classes with the most positions in the partitioned set.

From the filtered set K, a random pair of partition classes is then selected which is then merged, and the newly formed partition P' is tested after that (lines 9, 10). The merging of two classes a, b in a partition P is denoted by $P[a, b]$. This merger must also be reflected in the tile mapping function f, where at least two glue types are merged into one, this is indicated with f' on lines 10 and 14. Given a partition $P \in X$ and a tile map function $f : P \to \Sigma^4$, a new tile map f' is obtained from f by merging glues a and b as

$$\sigma_D(f'(p)) = \begin{cases} a, & \text{if } \sigma_D(f(p)) = b \\ \sigma_D(f(p)), & \text{otherwise} \end{cases}, \ \forall (p, D) \in P \times \mathcal{D}.$$

After that, the merged partition P' is tested whether it is *constructible* with the following outcomes.

- If P' *is constructible*, then it is tested whether it is a *refinement* of the target pattern $P(c)$:
  - If $P' \not\sqsubseteq P(c)$, then it is abandoned as it cannot provides a solution to the problem.
  - If $P' \sqsubseteq P(c)$ and $|P'| \leq |P|$ it is set as the current best solution together with tile mapping function (lines 15–17).
- If P' *is not constructible*, then there must exists two partition classes $p_1, p_2 \in P'$, $p_1 \neq p_2$ such that tile mapping function f' gives $\sigma_S(f'(p_1)) = \sigma_S(f'(p_2))$ and $\sigma_W(f(p_1)) = \sigma_W(f(p_2))$. Therefore, those two classes are merged, and the newly formed partition is tested again (lines 11–14).

Lastly, no matter what the outcome of merging operation was, the pair of classes a, b is removed from the set H (line 18) and the heuristic selection of pair and merging classes afterwards is repeated until the set H is not empty (line 19).

In the end, the tile set T is generated from the best-found partition P and tile mapping function f (line `20`), followed by the generation of the seed structure, which is denoted by function `generate_seed`(P, f) on line `21` in the algorithm and thus completing the solution TAS $\mathscr{T}$.

The description of the implementation of the PS-H algorithm using C++ programming language is presented in Section 3.2.

Chapter 3 Implementation

In this chapter, an implementation description of the original software for *Tile Assembly System* simulation and synthesis is presented. The implementation of the simulator is described in the following Section 3.1, and the implementation of tile assembly system synthesiser is outlined in Section 3.2.

3.1 MuTATOR – The Tile Assembly Simulator

MuTATOR software is introduced in this section. The acronym stands for the *Meta-Material Tile self-Assembly sysTem simulatOR*, where the mention of meta-materials comes from one of the main goals of the *EXPRO* project to which this thesis contributes. One of the aims of the project is the research of self-assembly of a macro-scale structure which forms meta-material. The meaning of the rest of the name acronym is straightforward; MuTATOR is an application for simulation of tile assembly systems.

For comparison purposes, a brief overview of the previous work in the field of tile-based self-assembly simulators follows. There exist at least another three simulator implementations (known to the author) – *Xgrow*, *ISU TAS* and *PyTAS*.

The first, *Xgrow* [71] was developed by the DNA and Natural Algorithms Group, led by Winfree, at the California Institute of Technology. Xgrow is the oldest of the three simulators as the first version was released in 2003. Xgrow is written in the C programming language for the X Windows environment, hence the ”X” in its name. Xgrow can simulate *aTAM*, and *kTAM* systems in 2D and it provides a wide variety of control parameters which enable modification of the tile self-assembly system simulation. Furthermore, Xgrow even allows altering the simulation parameters dynamically during the assembly growth. Xgrow provides powerful high-level insights into the working mechanisms of tile assembly systems and allows researchers to study the outcomes of the interplay of various parameter settings. Besides its age, Xgrow does not provide much debugging or inspection tools, which was the primary motivation for developing a newer simulator.

The *Iowa State University Tile Assembly Simulator* (*ISU TAS*) was developed by Patitz [72–73] at the Iowa State University Laboratory for Molecular Programming. Its first version was released in 2009, and the development stopped at the first half of 2018. ISU TAS is written in C++ programming language, and it provides both 2D and 3D simulation of *aTAM*, *kTAM* and *2HAM* systems. ISU TAS also includes a graphical tile set editor which provides an easy and convenient way of designing a tile assembly system. The simulator enables detailed configurations of the simulated environment, including the specification of concentrations of tile types. Although the ISU TAS is an open source application, it is relatively mature software package consisting of many modules, and thus it renders any modifications more difficult than the development of new simulator precisely suited for requirements of the project.

The last simulator discussed here is the *PyTAS* (*Python-based Tile Assembly Simulator*) which is being developed by a team headed by Patitz [74], and currently, it is available in an early-beta release from early 2018. Patitz stated [74] that the motivation behind the development of the new Python-based simulator is that Python scripts can more easily be updated, and the simulator can run on multiple platforms while Python provides highly optimised 3D rendering engine. PyTAS can simulate only the *aTAM* but in both 2D and 3D. Besides, PyTAS delivers a similar look and functions as ISU TAS. Even though the PyTAS could be more easily modified then ISU TAS, its existence was not known in the time when the development of the new simulator started; hence, it would be impossible to use it.

Simulator	aTAM	kTAM	2HAM
Xgrow	2D	2D	×
ISU TAS	2D&3D	2D&3D	2D&3D?
PyTAS	2D&3D	×	×
MuTATOR	2D	×	2D

Table 3.1. Comparison of supported models of tile-based self-assembly systems by simulators.

Another reason for the development of the new simulator is that the simulator can be fitted to requirements of the EXPRO research project. The research studies ways how to assemble 3D-printed cubes with highly complex inside structure. By assembly of different types of those cubes into a designed pattern, the assembled material gains various mechanical properties due to the structure of the pattern. Thus, such assembly constitutes a material called *meta-material* (or *mechanical metamaterial*). The 3D-printed cubes are the smallest units, building blocks, of the meta-material structure; therefore, they are naturally represented as tiles in the tile assembly system due to their similar roles. The development of the new simulator allows for total control over the implementation of assembly models, and so it further helps with the research.

In the following text, a specification guiding the design of the simulator is demonstrated along with examples of simulator usage in Section 3.1.1. A detailed description of the program architecture is presented after that in Section 3.1.2.

3.1.1 Features

Currently, MuTATOR can simulate the self-assembly process of *aTAM* and *2HAM* systems in 2D. However, it was developed with a possible extension to 3D in mind. MuTATOR is written using the C++14 modern specification of C++ programming language [75–76]. The *graphical user interface* (*GUI*) was built on the *GTKmm* library which provides C++ *application programming interface* (*API*) of the *GTK project* [77–78] which is part of the *GNOME project*. The GTK framework offers a rich variety of graphical widgets which have native Linux-based-system look and feel. Therefore, the application has familiar features and intuitive controls for the user. In addition, the application utilises the *spdlog* library [79] for logging of the program output. The example of MuTATOR graphical interface can be seen in Figure 3.1.

The primary task of the simulator is the validation of designed or synthesised tile assembly systems. Moreover, the simulator enables that a simulation can be reversed, which can be used for investigation of the assembly. Therefore, the interface of the simulator offers several modes of simulation such as step-by-step simulation, animated simulation and maximum-speed simulation. Because the theoretical models of self-assembly systems are discrete, the simulation progresses in a sequence of steps. Thus, the step-by-step simulation is the standard way of simulating the aTAM system, where

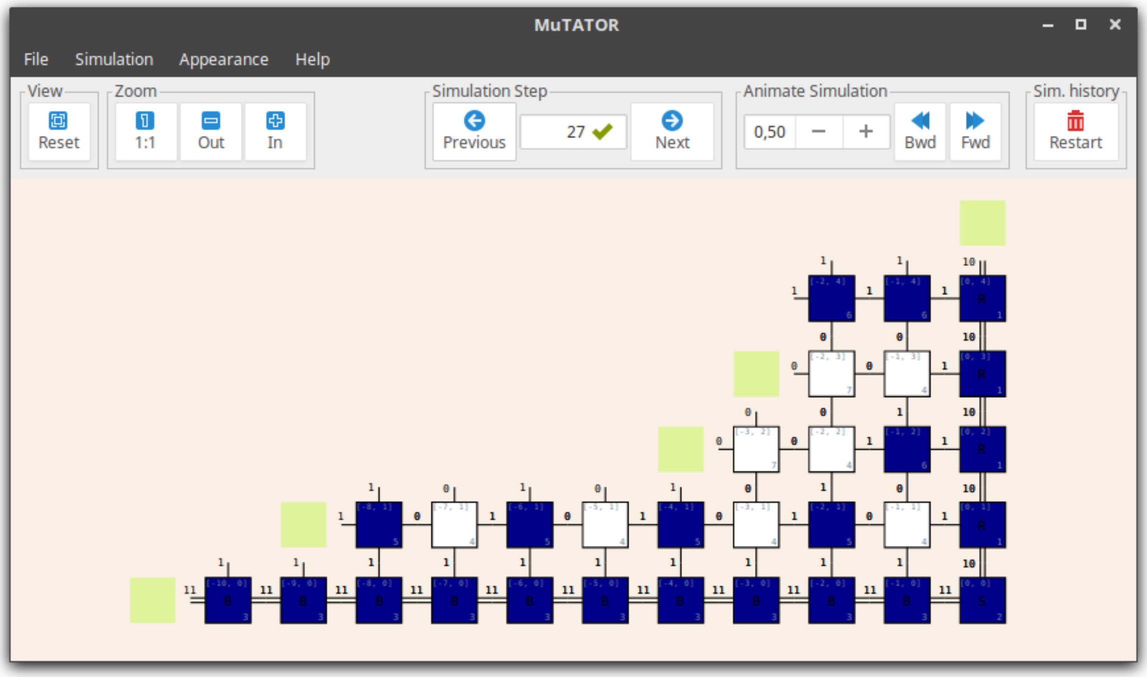


Figure 3.1. Example of MuTATOR window layout. The window is divided into two parts, the top part contains all program and simulation controls, and the bottom part is used for assembly state visualisation.

the user controls the simulation step by incrementing it or decrementing it via designated buttons. The animated simulation mode increments the system step in a periodic manner, where the user can specify the length of the time period. Lastly, the maximum-speed simulation mode enables the user to enter a specific step, and the simulator will calculate all preceding steps without any delays, so the only speed-limiting factor is the computation power of the machine.

The main difference between the animated and the maximum-speed mode is that the maximum-speed mode does not redraw the simulated assembly after each added or removed tile, which saves a decent amount of computations. Therefore, the maximum-speed mode acts as a convenient way to validate tile assembly systems. For example, the user expects that their aTAM system will terminate in step 100, so the user starts the maximum-speed mode by entering the termination step. Then, the simulator rapidly gives feedback to the user that the simulation terminated, say, on step 42. The user can manually back-track the assembly process to investigate the cause of the system early termination afterwards.

The *Tile Assembly System* for simulation is specified in a file using *JSON* notation [80–81]. The data in JSON format provides good readability of the data for a human while the data are also easily modifiable. For JSON related operations, the simulator uses a small library developed by the author in previous projects. The structure of the TAS file is based on the theoretical models of TAS discussed in Chapter 2. Therefore, the TAS file has to contain a specification of the system temperature and the set of tiles acting in the assembly. Additionally, the seed structure must be specified for the aTAM simulation. MuTATOR allows only positive integer system temperature.

The set of system tiles is specified as a vector of elements in the JSON, where each tile element has an optional field for its four glues and one additional optional field which encapsulates arbitrary user-specified data. One possible example of tile in JSON

```
{
   "north": {"id":0, "strength":1},
   "east":  {"id":1, "strength":1},
   "south": {"id":1, "strength":1},
   "west":  {"id":0, "strength":1},
   "additional_info": {"fill-color":"White"}
}
```

Listing 3.1. Example of a tile represented in JSON format.

representation is in Listing 3.1. The additional field in JSON tile representation is primarily used to set visual representation of the tile such as its colour or label. The glue element consists of the positive integer ID number, which determines the type of the glue and the positive integer bond strength. The glue strength function is not explicitly stated as in Definition 2.10 because it is determined from the implementation of glues implicitly and it behaves as in Definition 2.5, i.e., the bond strength between two glues is generated if and only if both glues have the same type (ID), otherwise the bond has zero strength. Furthermore, the seed structure for the aTAM is specified as a pair of position and tile elements, and the set of tiles used in seed structure is then added to the rest of specified tiles, and therefore the seed tiles can be reused (and often they are) in the assembly.

Moreover, MuTATOR offers two other options to load a TAS for simulation besides loading a file. Given the string nature of JSON data representation, the TAS in JSON serialised string can be directly given to the simulator as program argument via the command line, or it can be sent using the standard input. Reading the TAS representation from the standard input enables pipelining several commands in the command line. Thus, redirecting the TAS generated by other application to the simulator where it can be validated, and then, if the TAS produces the desired assembly, it can be saved. This feature has proven itself useful in the debugging process of the TAS synthesiser discussed in Section 3.2.

One of the requirements of the project was to include an option to display the internal structure of the 3D-printed cubes from which the meta-material is built. The cut through the body of the cube is called a *mesh* in this thesis. The mesh of each 3D-printed cube type is used as the graphical representation of the tile with a special flag in the simulator, which results in an assembly visualisation as can be seen in Figure 3.2.

3.1.2 Application architecture

The *Object-Oriented Programming* (*OOP*) paradigm is heavily utilised in the design of MuTATOR. Therefore, most of the application source code is assigned to classes, which abstracts corresponding concepts or objects. The simulator classes can be divided into two general groups. One group of classes defines the GUI of the application, and it handles runtime commands from the user. The other group incorporates various simulator models and provides a common interface for them. These two groups somewhat represent a *frontend* and a *backend* of the application, respectively. The frontend enables the user to interact with the application, and it calls functions provided by the API of the backend. The backend is independent of the frontend and provides all needed computations; however, without the frontend, the results of the computations cannot be presented to the user. Therefore, the general goal of application architecture is to have independent backend and frontend, and the number of needed connections between the frontend and backend should be kept as low as possible.

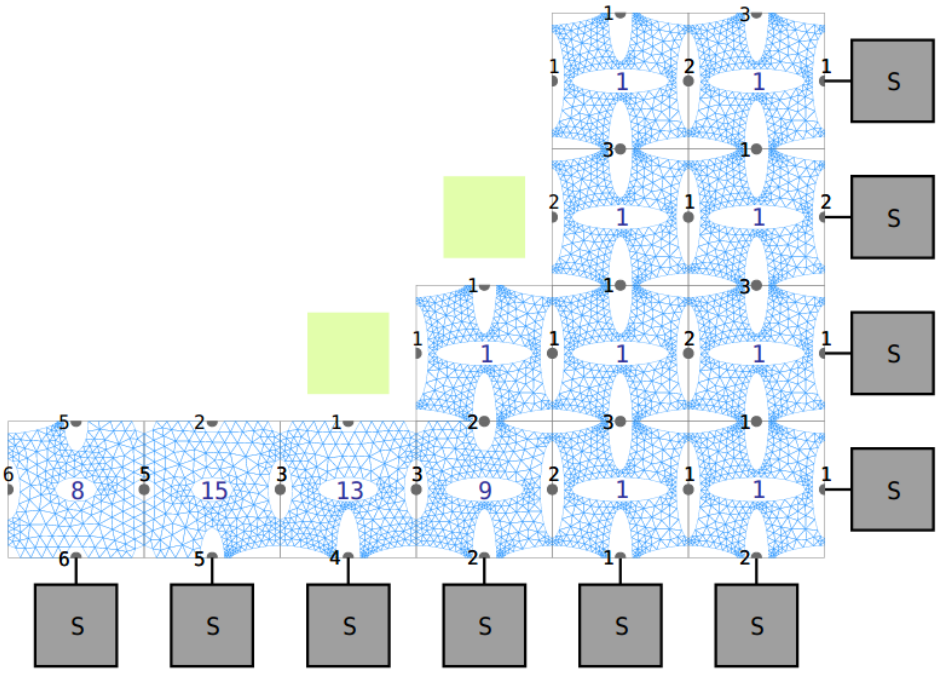


Figure 3.2. Example of meta-material mesh visualisation in assembly simulation.

Note that there are many more classes used in the implementation of the simulator especially in the backend, but they are not explicitly mentioned in this section because they do not carry a significant functionality, and they serve primarily as data structures. Those classes implement representations of theoretical concepts which are defined in the previous chapter, for example, representation of glue, tile, tile assembly system or assembly.

Connections between classes in the application are indicated in Figure 3.3. In MuTATOR, the main class which connects the backend with the frontend is called `Application`. Note that the vast majority of classes presented here are nested within the `mutator` namespace, but the namespace is not explicitly stated for the sake of simplicity in this thesis[1]. The `Application` class is derived from the same-named class from the Gtkmm library, and so it provides an interface for signals or callbacks. The signals are used to propagate asynchronous commands from the user in frontend to the simulator in the backend, and thus the application interacts with the user.

The `ApplicationWindow` class lies in the centre of the application frontend, and it provides the look of the GUI window with which the user interacts (Figure 3.1). The layout of the window was designed using the Glade tool, which is an application from the GNOME project. Glade is a ”what you see is what you get” (WYSIWYG) rapid development tool for easy design of user interfaces for an application using the GTK framework. Using Glade, the whole layout of the application is defined in one XML file, and thus the GUI definitions do not occupy space in the source files. This approach, in turn, improves the clarity of the source code, while it also allows for easy modifications of the layout. The `ApplicationWindow` then defines the reactions for the possible user inputs (actions), such are clicks on buttons or changes in entry boxes, and if needed it sends signals to the backend through the `Application` class.

[1] A *namespace* is a C++ language feature which helps to ensure that no identifier (name) collision occurs in the source code, i.e., classes `a::Foo` and `b::Foo` are considered different. Therefore, the technically correct name of the class is `mutator::Application`.

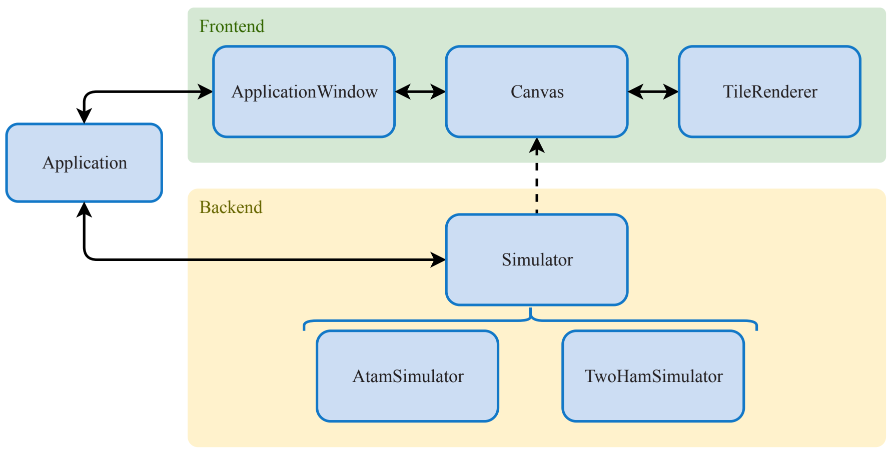


Figure 3.3. Dataflow diagram of the essential classes in MuTATOR application, where the dotted line between `Canvas` and `Simulator` means that the `Canvas` does not own the simulator instance, and thus it can only read the status of the simulator, and the bracket indicates which classes are derived from the `Simulator` class.

Possibly the most visible frontend feature is the `Canvas` class. The `Canvas` class is used for drawing the state of the simulation, and also it provides functions for view modifications so that the user can pan and zoom the view by the mouse or by corresponding key-strokes or buttons. The `Canvas` class is the only other class apart from the `Application` class, which has a direct connection to the application backend. In more detail, the `Canvas` class gets a pointer directly to the current state of the simulated assembly upon the simulator creation so it can display the assembly state as the system evolves in the simulation. Note that this is a read-only connection, which enables the `Canvas` to retrieve the state of the simulation asynchronously to the simulator. The `Canvas` uses the `TileRenderer` class for drawing the individual tiles in the assembly. The `TileRenderer` class fully defines the graphical representation of the tile, and it can provide various methods for altering the tile look in the simulator. Besides, the `TileRenderer` class provides an interface for an alternative renderer which can be added in the future (possibly for 3D rendering).

As can be seen in Figure 3.3, the backend of the application is essentially the `Simulator` class. The `Simulator` class is an abstract class that means that it defines a necessary interface which must be implemented by its deriving classes, as hinted in the figure. This interface is used by the other classes in the application to communicate with the simulator. Therefore, it unites the control of all implemented models of the simulation, such as aTAM and 2HAM in case of MuTATOR. Also, it provides common simulator framework and thus an easy way for adding a new simulation model.

In order to enable the option to reverse the simulation to the user and allow the user to examine previous steps in the simulation, each class implementing the `Simulator` interface must keep a history of calculated steps. Keeping the simulation history brings several advantages. It avoids the recalculation of previously calculated steps, so if the user wants to watch the progress of assembly again from the beginning, the simulator repeats the saved sequence of steps. The history is especially crucial in aTAM simulation due to its stochastic nature. Because aTAM selects a random tile from the set of possible additions to the assembly in every step, the simulation would not be consistent without the simulation history. If the user would revert one simulation step and then proceeded again to the next step, it could have a different outcome.

Some of the interesting algorithms used in the simulator implementation are further discussed in Appendix Section D.2.

3.2 MuTAGEN – Tile Assembly Synthesizer

In this section, MuTAGEN application is described. MuTAGEN acronym stands for the *Meta-Material Tile self-Assembly system GENerator*, and it is a program for solving the *PATS* problem defined in Section 2.3.2.

There is only one published PATS problem solver (known to the author) called 2PATS developed by a team led by Kari [82–83]. This solver is available in two versions, the first version is written in C++ programming language and the second version intended for distributed computing is written in Haskell programming language. The main disadvantage of both versions of the 2PATS program is that, as its name suggests, it only solves binary patterns, i.e., patterns with only two colours. Because it was needed to solve patterns of at least 16 colours for the EXPRO project, the MuTAGEN application was developed.

In the rest of this section, features of the application are presented in Section 3.2.1, and the architecture of the application is further discussed in Section 3.2.2.

3.2.1 Features

MuTAGEN synthesiser is a console application, which means it has no graphic user interface. It is written in the modern specification of C++ programming language like MuTATOR. Even though MuTAGEN is a console application, it uses *GTKmm* library [77–78] because it offers useful functions for processing bitmap image files and functions for testing file existence on a hard disk. Because MuTATOR also utilises the GTKmm library, the usage of the library in MuTAGEN does not introduce new project dependency. Moreover, MuTAGEN also uses the *spdlog* library [79] for logging and control over the output in the console.

The only goal of MuTAGEN is to solve the PATS problem such that the user inputs a pattern and the application returns the aTAM tile assembly system file in JSON representation compatible with MuTATOR simulator which was described in Section 3.1.1. An illustration of MuTAGEN workflow is in Figure 3.4.

For solving the tile set synthesis problem, MuTAGEN uses an implementation of *PS-H* (*Partition-Search with Heuristics*) algorithm developed by Lempiäinen et at. [69] described in Section 2.3.2. Because the PS-H is a *greedy* algorithm, it searches for the solution based on local optimum given by the heuristic, it generally does not always find the globally optimal solution, but it gives a somewhat optimal solution much faster than complete space search methods. Therefore, in order to find the best solution, MuTAGEN runs the PS-H algorithm multiple times and then returns the best solution found. Found solutions are lexicographically ordered by the number of different glue types needed and then by the number of unique tiles required for the self-assembly of the given pattern, i.e., a solution with a lower number of glue types is prefered over the solution with fewer unique tiles but with a greater number of glue types.

Intuitively, the goal is to get the given pattern "drawn" (recreated) by the self-assembly process. Therefore, the input pattern can be specified by either a bitmap image file or a text file. For usage of the image file, each colour represents a tile type with a unique feature and the position of the colour (pixel) in the image determines the desired position of a tile in the pattern. Similarly, in the text file pattern, the tile types are specified by integer numbers separated by spaces to denote rows of the pattern.

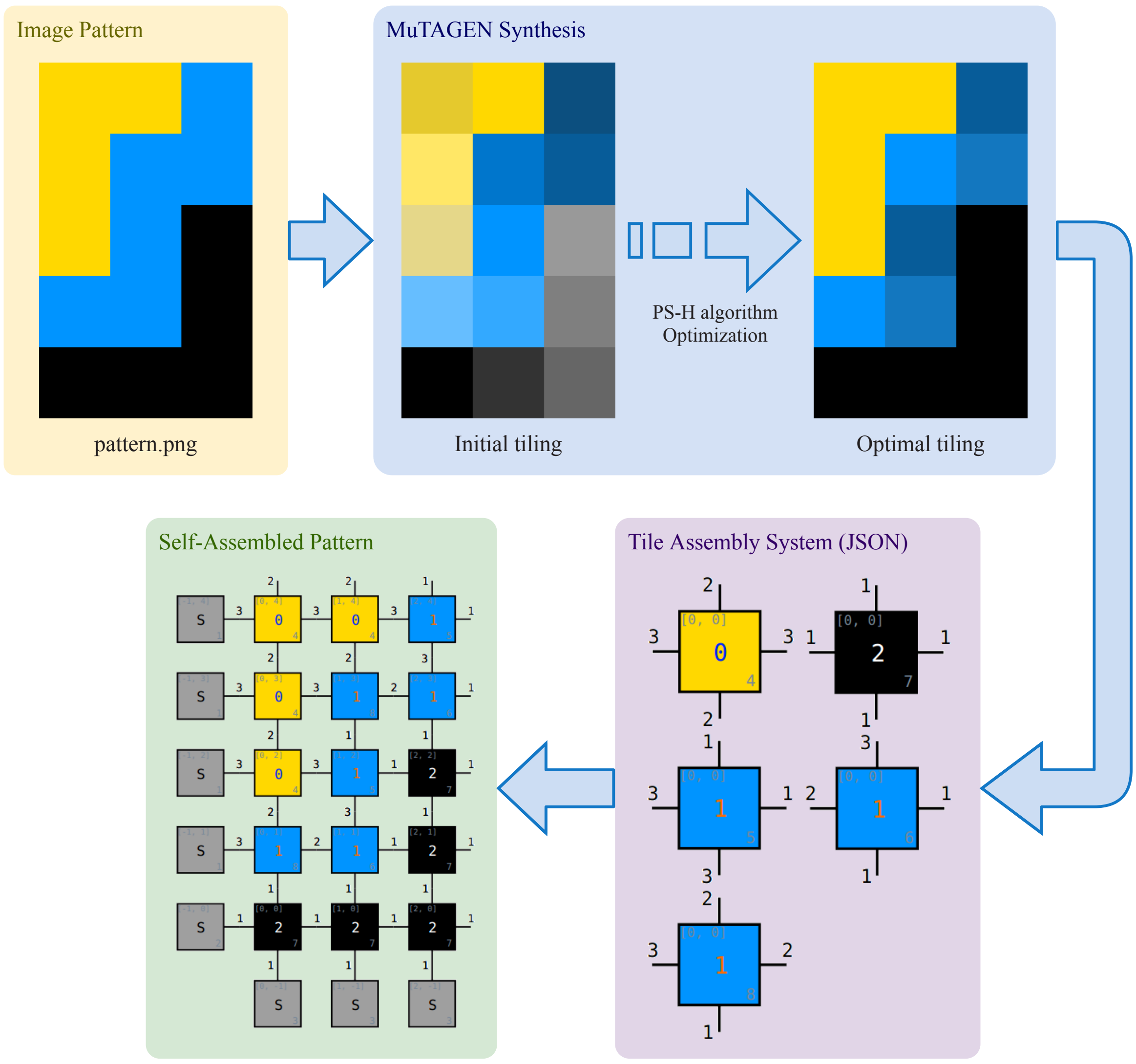


Figure 3.4. Example of MuTAGEN workflow. An image pattern is given to MuTAGEN, which returns the most optimal solution found as a JSON file specifying the Tile Assembly System which self-assembles into the desired pattern. Different colour shades in the tiling examples represent different tile types.

Note that the number of tiles in the solution of the pattern is always greater than or equal to the number of colours in the given pattern and thus one particular colour in the pattern can be represented by multiple tile types with different glue configurations in the solution. Additionally, it is very advantageous to use lossless image formats, e.g., PNG or BMP, when specifying a pattern using an image file, as compression methods of lossy image formats, e.g., JPEG, cause a change of colour for some pixels in the pattern which consequently devalues the pattern.

The application provides several options to customise the behaviour of the searching. The user can specify the number of concurrent computation threads. By default, the number of used threads is set to the number of processor cores of the computer, which should ensure the optimal performance, and it significantly reduces the computation time. The user can specify the number of runs of the searching algorithm, which increases the probability of finding a more optimal solution. Because the time needed for one run of the searching algorithm might not be known, the user can specify a desired time of the optimisation. Furthermore, the user can specify both the number of optimisation runs and time of the optimisation, and the optimisation terminates on whichever event occurs first.

The application also provides several options to modify the output solution. The user can select the position (orientation) of the ”L-shaped” seed structure which can be set to any corner of the pattern. The application allows the user to set the type of the seed structure either to ”self-assembling” type or ”fixed” type. The ”self-assembling” type is the standard seed structure which can self-assemble from a single origin seed tile and then support the pattern assembly. On the other hand, the ”fixed” type seed structure is generated as already assembled in the TAS file where the individual seed tiles do not have glues between themselves. Therefore, the ”fixed” type of seed structure simulates an external template in the system which cannot self-assemble, but it provides the needed support for the pattern assembly.

Additionally, the application offers an option to remove tile types which correspond to a given colour in pattern from the solution. This option can further reduce the number of tile types of solution or removes filler tiles; however, it is hazardous as it can destroy the self-assembling property of the solution.

Finally, the application provides an option to load all previously mentioned options from a text file which can be easily created and modified by the user and thus it can be used for re-running the search process with the same conditions or sharing the configuration between users.

3.2.2 Application architecture

Like in MuTATOR, the *Object-Oriented Programming* (*OOP*) approach is also used in the design on MuTAGEN application. Both applications share a common library of classes representing entities used in the tile assembly systems such as representations of glue, tile, or tile assembly system. In the MuTAGEN architecture, the main class is `TasSynthesizer`, which encapsulates the logic for parsing the given pattern, optimisation process and implementation of the PS-H algorithm.

The PS-H algorithm performs searching in the space of partitions and thus to facilitate the abstractions, the `TasSynthesizer` utilises a *class template* `Partition`. The *class template* is a C++ language feature which allows for defining a family of classes with the same functions performed on different types of data. The typical usage of class templates is for defining data *containers* such as arrays, maps, queues or linked lists which allows to store any data type in them and perform actions typical for the corresponding container. A partition is also a *container* because it can store a configuration of different data types. For example, a k-coloured input pattern is a partition of colours, and a solution to the PATS problem is effectively a partition of tiles.

In the application, there are defined two additional specialisations of the `Partition` class template – the `PatternPartition` class and the `TilePartition` class. The specialised classes retain the same functionality as the `Partition` class but define additional functions which are specific for the particular specialisation. The class inheritance is illustrated by an inheritance diagram in Figure 3.5.

As the name suggests, the `PatternPartition` class is an internal representation of the desired pattern. It consolidates functions for parsing supported image files and text files which define the pattern. Therefore, the class unifies the representation of a pattern which is used to guide the optimisation process.

The other specialisation, `TilePartition`, represents a partitioning of tiles and provides additional functions for merging tile types, functions for testing whether the tiling

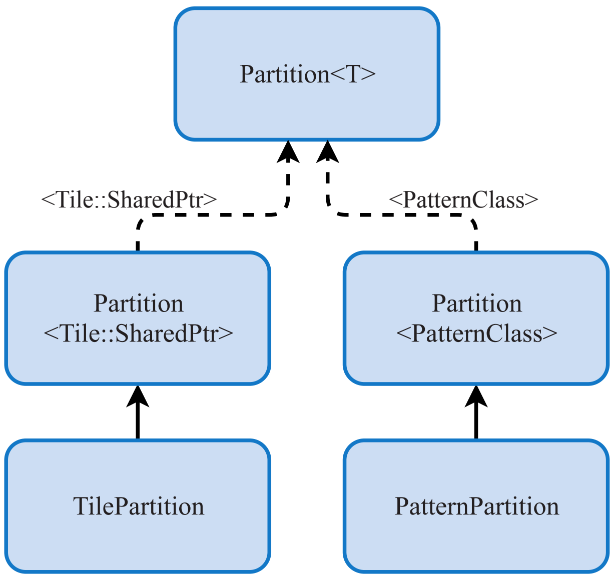


Figure 3.5. Class inheritance diagram of specialised classes which represents pattern partitioning and tile partitioning, where dotted lines indicate classes created from a class template, and full lines indicate inheritance of classes.

is *constructible* or functions for generating seed structure for the tiling. The construction of an instance of the `TilePartition` class implements the $MGTA$ function described in Definition 2.34. The `TasSynthesizer` class utilises the `TilePartition` class to perform the search operations on it, and the `TilePartition` also stores the ongoing solution of the problem during the optimisation process.

A couple of examples of the synthesiser application implementation are further described in Appendix Section D.3.

Chapter 4
Experiments

This chapter discusses the results of experiments, which were conducted to validate properties of the simulator implementation and the synthesiser implementation. Experiments were performed on a single PC with parameters shown in Table 4.1. All measured values were captures from the inside of implemented applications.

Parameter	Value
Operating System	Linux Mint 18.2 (64-bit)
Processor Type	Intel® Core™ i3-4010U
Processor Frequency	1.70 GHz
Number of Processor Cores	2
Number of Concurrent Threads	4
Processor Cache Size	3 MB
Memory Size	8 GB
Memory Type	DDR3L
Memory Frequency	1333 MHz

Table 4.1. Specification of machine used for experiments data measuring.

In the rest of this chapter, the results of experiments which show the performance of MuTATOR application are described in Section 4.1. Also, the results of experiments which were performed on MuTAGEN application to verify the influence of synthesised pattern on the resulting tile assembly system are presented in Section 4.2.

4.1 MuTATOR Experiments

This section discusses how properties of the simulated system influence the time needed for computing the simulation. The first two experiments presented here were conducted on the aTAM simulator, and the results of the experiments are shown in Section 4.1.1. Next, the results obtained by the simulation of various 2HAM systems are described in Section 4.1.2.

4.1.1 Simulating aTAM

It is always advantageous to know how the computation time is affected by the size of the simulation. For the case of aTAM simulation in MuTATOR application, a series of computation time measurements were captured in order to validate the computation time dependence on the number of performed simulation steps. The captured data are illustrated by a graph in Figure 4.1 together with a table containing a set of basic statistics about a selected data in Table 4.2.

There were performed 200 measurements of the computation time which the simulator needed for simulating the given number of simulation steps from the initialised state for each number of steps. As a representative model, the TAS assembling the

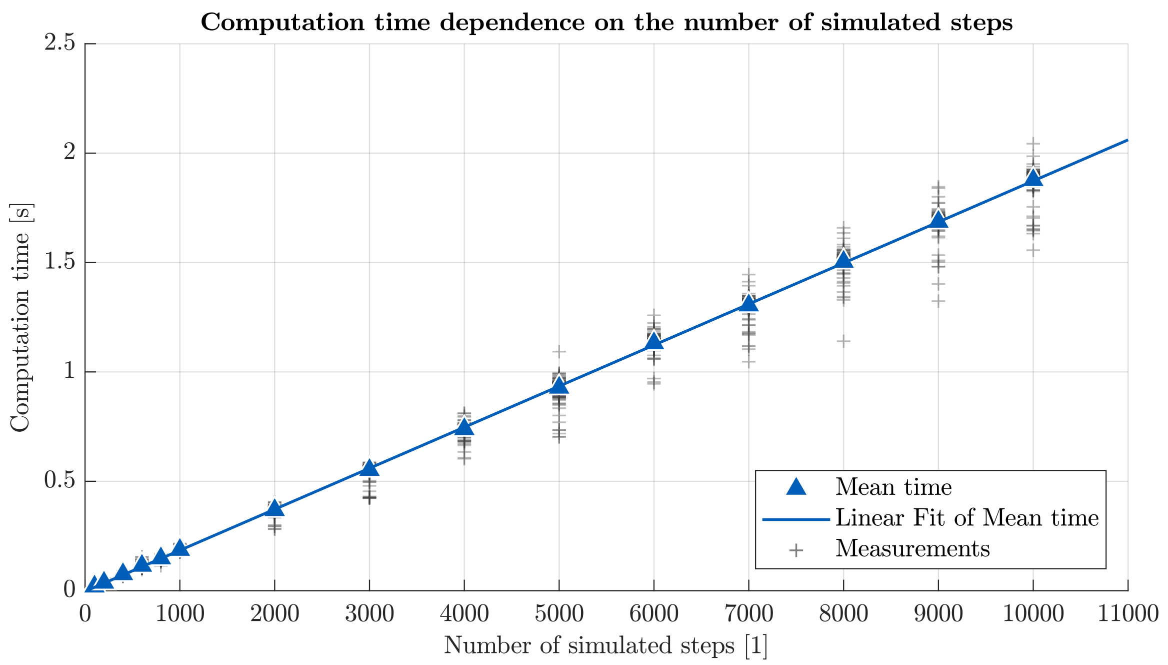


Figure 4.1. Computation time dependence on the number of simulated steps using the aTAM system, which assembles *Sierpinski triangle* fractal pattern. The blue triangle is the mean value of 200 measurements, which are denoted by grey crosses. The blue line is a linear trend fitted to mean values.

Num. of steps [1]	Mean [s]	Minimum [s]	Maximum [s]	Std. deviation [s]
200	0,0365	0,0328	0,0569	0,0023
1 000	0,1864	0,1773	0,2158	0,0050
2 000	0,3683	0,2815	0,4063	0,0157
4 000	0,7386	0,6032	0,8115	0,0246
6 000	1,1310	0,9458	1,2585	0,0323
8 000	1,5033	1,1401	1,6588	0,0448
10 000	1,8767	1,5563	2,0432	0,0550

Table 4.2. Basic statistics of the computation time needed for simulating selected number of steps in aTAM based on 200 measurements.

Sierpinski triangle fractal pattern was selected as it does not terminate; thus, it is not limited by the number of steps. As can be seen in the graph, the computation time can be approximated as a linear function of the number of simulated steps. This result agrees with expectations, as the simulator uses an algorithm which updates only the local neighbourhood of the last added tile in the assembly after initialisation. Therefore, the algorithm has a constant maximum number of operations in each step.

As can be seen in Table 4.2, the standard deviation of measured data is relatively small as the data are considerably consistent. The leading cause of differences between measured times is the random nature of aTAM system which causes that every simulation sequence is slightly different as in every step, there is a different number of available positions, where tiles can attach. Therefore, the computation time can vary.

Additionally, an experiment was conducted to determine the computation time of a large number of steps using the same tile assembly system. The resulting computation times are 19,09 s for 100 000 steps, 113,94 s for 500 000 steps and 198,68 s for 1 000 000 simulated steps. These measurements are consistent with the estimated linear trend from the graph in Figure 4.1.

Moreover, MuTATOR application offers an option to the user, which enables periodic redrawing of the simulated assembly during more extensive simulations. If the periodic redrawing is enabled, the simulation time dependence on the number of simulated steps will change to exponential function as the redrawing of the assembly is more demanding as the number of tiles in the assembly grows. During the redrawing of a large number of tiles, the computation time is heavily influenced by the computation power available in the used computer. Therefore, measurements which were captured with the redrawing enabled are not presented here as they have small information value, and they depend on the used computer machine.

The following experiment investigates how the number of tiles in the simulated system affects the computation time of the simulation. The measured data are depicted in a graph in Figure 4.2.

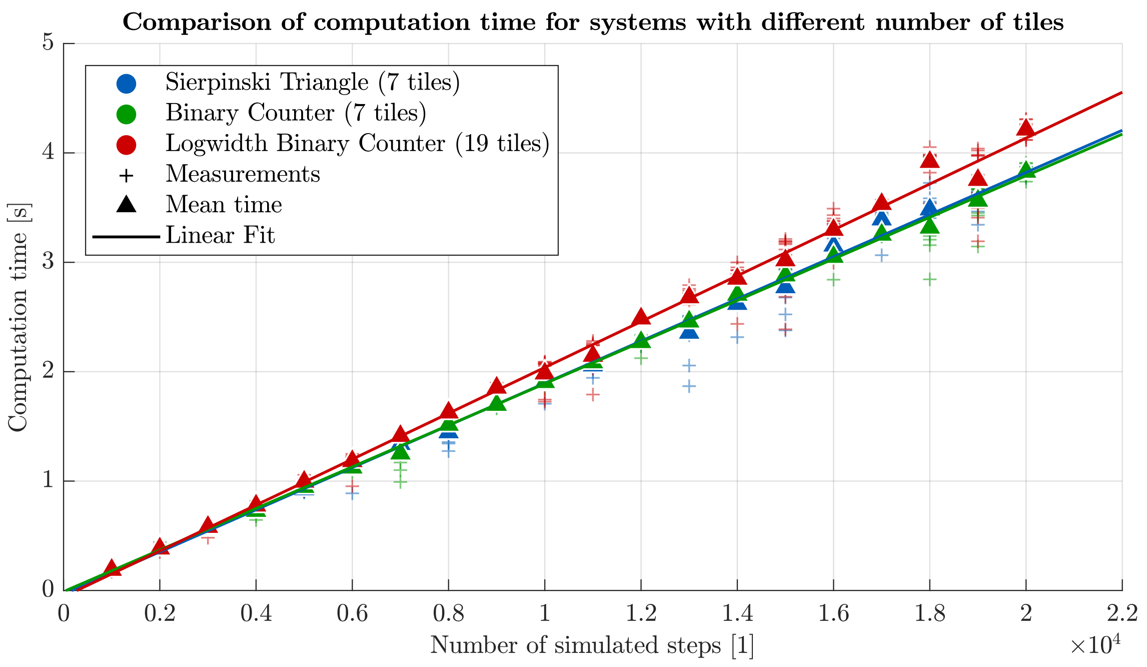


Figure 4.2. Comparison of computation time for aTAM systems with different number of tiles. Trends are fitted with linear functions.

There were performed ten sets of measurements on three systems – *Sierpinski Triangle*, *Binary Counter* and *Logwidth Binary Counter*, which are denoted by blue, green and red colour in the graph, respectively. The system referred to as *Logwidth Binary Counter*, assembles a binary counter with no additional supports unlike its alternative counterpart *Binary Counter*. The Logwidth Binary Counter consists of 19 unique tile types, but it is more tile efficient in the process of assembling the binary counter pattern than the other binary counter system. The remaining two systems consist of 7 tiles.

In the graph, it can be seen that the two systems with seven tiles have both very similar computation times, whereas the Logwidth Binary Counter has slightly longer computation times. The longer computation time is caused by the fact that more tiles must be checked whether they can be placed on the given location in the assembly in each simulation step.

4.1.2 Simulating 2HAM

The simulation of 2HAM systems is quite computationally demanding, which can be demonstrated by the measurements shown in the following graphs. A series of measurements on various 2HAM systems were conducted to examine the dependencies between the computation time and properties of the given system. The results of experiments are presented in Figures 4.3, 4.4 and 4.5. For experiments, five systems were used.

The system denoted as *ABC* in graphs is the system which assembles a subset of the alphabet, and it is used in the example from Section 2.2.3. It is the only system from the five tested systems which terminates. The system *Infinite Line* assembles into an infinitely long line of tiles with a height of two tiles. The system *Random Integer* system generates random strings of bits in the 2HAM simulation, and thus it generates random binary representations of integer numbers. The last two systems *Logwidth Binary Counter* and *Sierpinski Triangle* are the same systems used for the aTAM experiments described above.

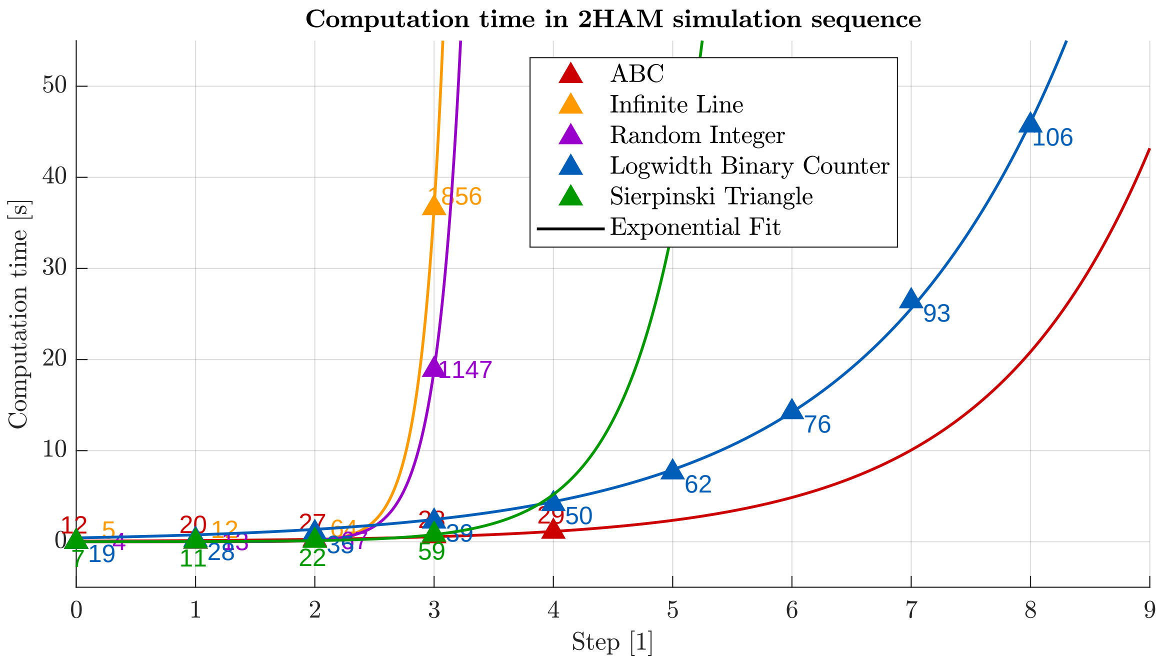


Figure 4.3. Computation time of the simulation sequence of several 2HAM systems. Numbers above data points denote the number of supertiles in the corresponding system in particular simulation step. Trends are fitted with exponential functions.

In the Figure 4.3, the explosion of needed computation time for 2HAM systems can be observed. The exponential growth of computation time is particularly prominent for the Infinity Line and the Random Integer systems. The computation difficulty is driven by two factors, which are the number of supertiles in the system and the number of available positions of each supertile to which other supertile can be adjoined. The number of supertiles in the system is annotated around each data point in the graph in Figure 4.3, or it can be observed in a separate graph in Figure 4.4.

It can be seen that systems with relatively slow supertile growth in the simulation sequence, such are the Logwidth Binary Counter and the ABC systems, can be simulated in reasonable amounts of time. It can be seen that the increase of the computation time of two consecutive steps in the Logwidth Binary Counter system allowed its simulation until the eighth step where the influence of the second factor took over. On the

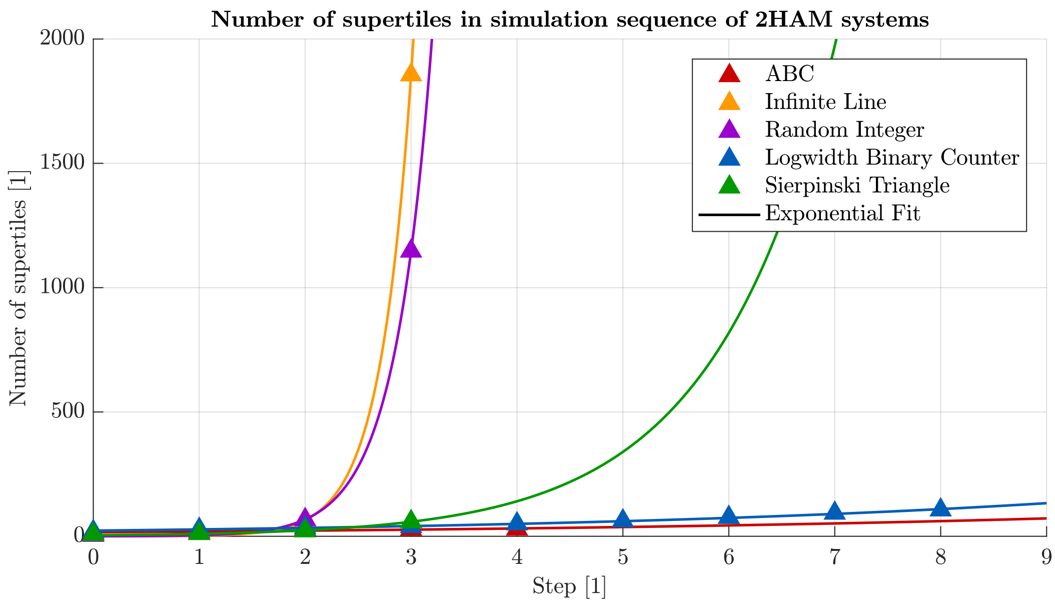


Figure 4.4. Number of supertiles in simulation sequence of several 2HAM systems. Trends are fitted with exponential functions.

other hand, the systems such as the Infinite Line and the Random Integer, rendered exceptionally difficult to simulate past the third step as the number of supertiles in the system exploded from 67 in the second step to 1 147 in the third step for the case of the Random Integer system, and from 64 to 1 856 for the case of the Random Integer system.

The dependency between the number of supertiles and the computation time is illustrated in a graph in Figure 4.5.

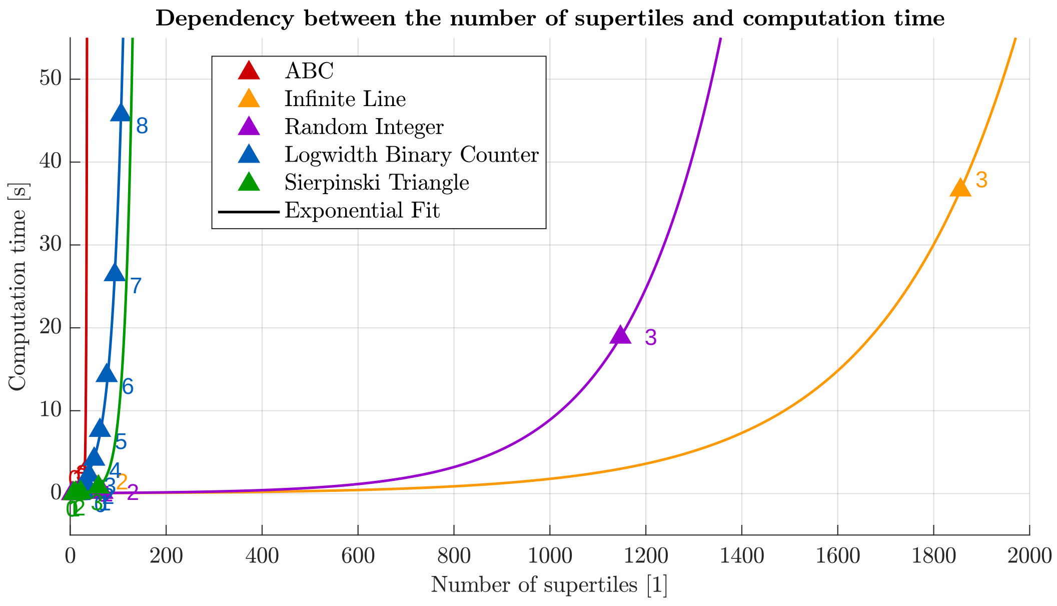


Figure 4.5. Computation time dependence on the number of supertiles in the 2HAM systems. Numbers above data points denote the corresponding simulation step. Trends are fitted with exponential functions.

4.2 MuTAGEN Experiments

The properties of synthesis of tile assembly systems using MuTAGEN application are demonstrated with several experiments in the following text. The experiments are primarily focused on how properties of the desired pattern influence the synthesised tile assembly system. The main properties of the pattern are the number of colours it contains k, its size and whether it is symmetric, or it contains some form of repeating pattern. The properties of the synthesis process, which can affect the solution is the number of the optimisation cycles, the position of the seed structure. Also, the computation time of the synthesis process is measured. The synthesised system if evaluated by the number of its tiles $|T|$ and by the number of its glues $|\Sigma|$.

First, the influence of the number of colours in the pattern is examined in Section 4.2.1. Next, synthesised systems from the set of patterns are compared to their counterparts with known optimal or analytic solutions in Section 4.2.2. Lastly, the influence of a seed position on a synthesised system is presented in Section 4.2.3.

4.2.1 Synthesising TAS

In this experiment, the effects of different patterns on the computation time of synthetic solutions are demonstrated.

To determine how the number of colours in the pattern k and the size of the pattern influences the synthesis time, a set of random patterns was generated. The largest form of each random k-coloured pattern is shown in Figure 4.6. The set of different sized patterns was generated from the largest pattern by iteratively cropping the bottom row and the right column of pixels. This procedure yielded a series of square patterns with a side of size in the range from 5 to 16 points. The random patterns were chosen to decrease any influence of the symmetry in the pattern on the synthesised systems. The presented results correspond to the most optimal solutions obtained after 24 optimisation cycles.

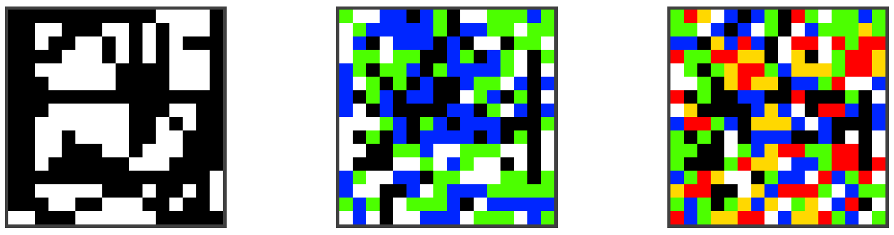

a) $k = 2$ b) $k = 4$ c) $k = 6$

Figure 4.6. Random k-coloured patterns of size 16×16 used for testing of synthesis properties.

The dependence between the number of colours in the pattern and the number of points in the system is illustrated in Figure 4.7. The synthesis time of the pattern with six colours was faster than for patterns with two or four colours, which can be surprising. The faster synthesis of the more-coloured pattern is caused by the fact that the used algorithm tries to merge pairs of tiles with the same colour in the pattern. A less-coloured pattern has more unordered pairs of points with the same colour, and therefore, the optimisation algorithm performs more computations.

However, unsurprisingly, the more-coloured pattern generally yields tile assembly systems with more tile types and more glues needed for the self-assembly of the pattern. The number of tiles and the number of glues for different sized random k-coloured

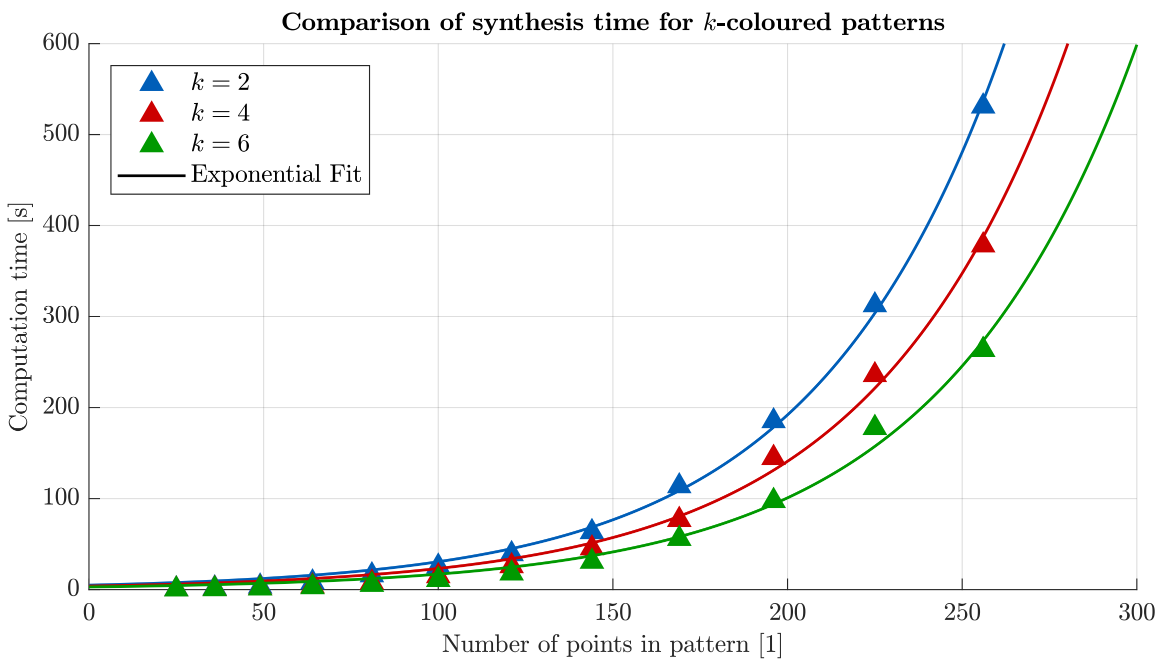


Figure 4.7. Comparison of synthesis time dependences on the number of points in random k-coloured patterns (Figure 4.6). Trends are fitted with exponential functions.

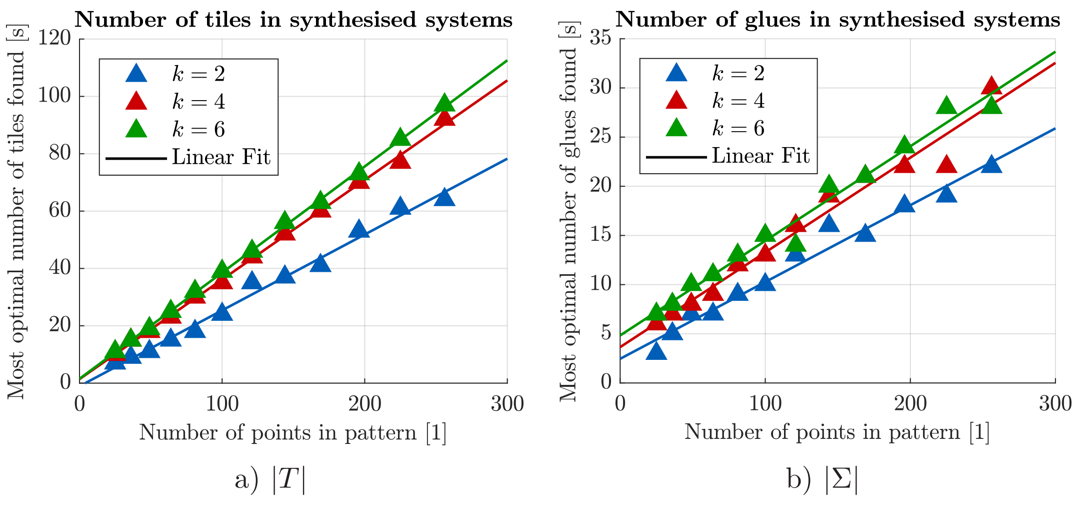


Figure 4.8. Graphs comparing dependences between the number of tiles or glues in synthesised systems and the number of points in random k-coloured patterns (Figure 4.6). Trends are fitted with linear functions.

patterns are illustrated in graphs in Figure 4.8. It can be seen that the number of tiles in the system is approximately a third of the number of points in the pattern in the case of a random pattern. Similarly, the number of glues needed in the synthesised system can be approximated as a linear function in the number of points in a random pattern.

Besides the random patterns, a small experiment was performed using larger image patterns, which are shown in Figure 4.9. The image patterns were synthesised using 12 optimisation cycles. The properties of the image patterns can be found in Table 4.3 together with properties of the most optimal solutions found and the time needed for the synthesis. There was performed one additional synthesis of a 2-coloured pattern of size 48×36 depicting the CTU logo, but the synthesiser has not even once completed

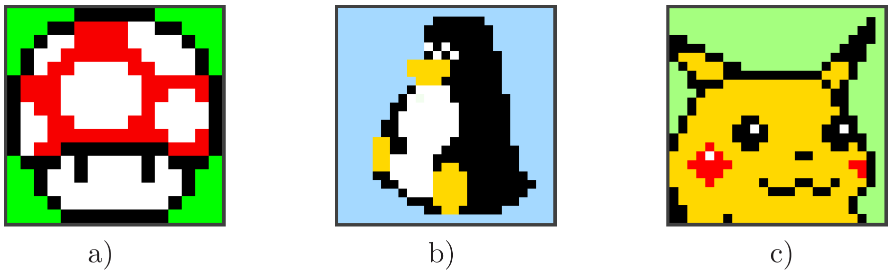


Figure 4.9. Larger image patterns used for testing TAS synthesis.

Figure	Size	k	$\lvert T\rvert$	$\lvert\Sigma\rvert$	t [s]
4.9 a	16×16	4	85	29	103,43
4.9 b	25×25	4	97	32	1477,86
4.9 c	24×24	5	114	36	1729,42

Table 4.3. Comparison of properties of synthesised solutions for larger image patterns, where k is the number of colours in the pattern, $|T|$ is the number of tiles found, $|\Sigma|$ is the number of glues, and t is the time of synthesis. Results are the best solutions found after 12 optimization cycles.

the optimisation cycle in 12 hours of running time. Together with the data in the table, this demonstrates how rapidly is the computation time increasing in response to relatively small changes in the pattern size.

4.2.2 Comparison of synthesised and analytic solutions

The goal of this experiment is to compare the known optimal solutions of self-assembly of some pattern to the synthesised solutions from the given pattern.

Note that the actual tiles in the systems are not tested whether they are identical, as only the number of tiles and glues are used for the comparison. This experiment merely tries to confirm that the synthesiser can found the optimal solution in terms of tile and glue counts. The synthesised system is generally not expected to have the same tiles as its analytic counterpart, because the synthesised solution is always finite in the number of points in the pattern and it relies on the configuration of the seed structure. Therefore, the tiles contributing to the seed structure are not counted.

In the Figure 4.10, the three patterns which were used in this experiment are displayed. The first pattern is the *Chessboard*, which can be self-assembled using two tiles with two glues. The second used pattern is the *Sierpinski Triangle*, which was used for testing the simulator. The Sierpinski Triangle system contains four non-seed tiles with two glues. Last used system is the *Binary Counter* which was also used in experiments with the simulator. The Binary Counter also contains four tiles with two types of glue. The properties of the systems can be found in Table 4.4.

These patterns were given to the synthesiser in the size of 16×16 points, and the presented results were the most optimal found after 24 optimisation cycles. As it can be seen in the table, synthesised solutions of the Chessboard pattern and the Sierpinski Triangle fractal both yielded solutions with the same number of non-seed tiles with the same number of glues as their analytic counterparts. However, the solution of the Binary Counter pattern was considerably worse. This experiment demonstrated that the pattern synthesis could result in a system with the optimal theoretic number of tiles and glues, but there are patterns for which the synthesised solution is always suboptimal.

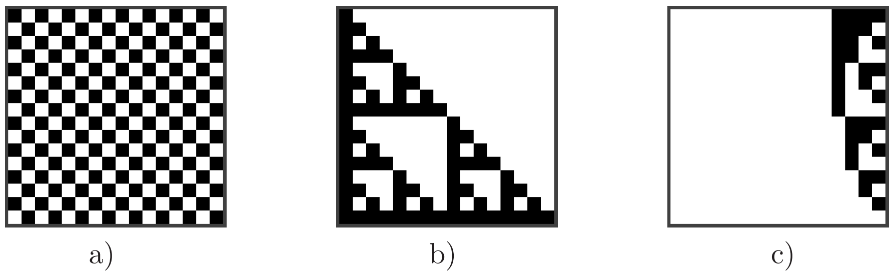


Figure 4.10. Image patterns used for comparing analytic and synthetic solutions.

System	Figure	Analytic solution $\|T\|$	$\|\Sigma\|$	Synthetic solution $\|T\|$	$\|\Sigma\|$	t [s]	Optimal?
Chessboard pattern	4.10 a	2	2	2	2	107,82	True
Sierpinski triangle	4.10 b	4	2	4	2	133,77	True
Binary counter	4.10 c	4	2	11	5	174,04	False

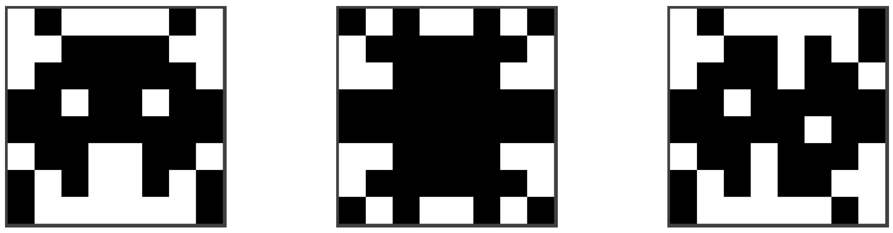
Table 4.4. Comparison of properties of analytic and synthetic solutions of patterns, where $|T|$ is the number of tiles found, $|\Sigma|$ is the number of glues, and t is the time of synthesis. Presented results are the best solutions found after 24 optimization cycles. The size of 2-coloured patterns used for synthesis was 16×16.

4.2.3 Influence of seed position on the solution

The last experiment tries to verify that the seed position affects the synthesised system. As was mentioned in the definition of the PATS problem in Section 2.3.2, the seed structure for a synthesised system covers two adjacent sides of the pattern assembly. MuTAGEN application offers the user an option to choose from which sides the seed supports the assembly. The idea behind this experiment is whether the position of the seed can influence the properties of the synthesised system.

A set of patterns of size 8×8 was generated. Each pattern has one type of symmetry. Example patterns are presented in Figure 4.11. The vertically symmetric pattern is expected to yield equal solutions for a seed located either on the bottom or the top of the assembly and no matter if it lies on the left side or the right side of the assembly. Note that the pattern with horizontal symmetry is identical as the pattern with vertical symmetry rotated by 90 degrees. Thus, it is not included here as it would yield similar results as the pattern with a vertical symmetry. The second class of patterns are the patterns with both vertical and horizontal symmetry, which is referred to as *full* symmetry for simplicity. The pattern with full symmetry is expected to yield equal solutions no matter on seed position. The last group of patterns used for this experiment are the patterns with diagonal symmetry. The pattern with diagonal symmetry is expected to yield equal solutions for diagonally opposed seed positions as the pattern is the same relative to the seed structure.

Table 4.5 contains the properties of the most optimal solutions found after 24 optimisation cycles. In each row, most optimal solutions for a given seed position are highlighted. As can be seen, for each type of pattern symmetry, there are systems which match the mentioned expectations, but also there are systems with behaviour opposing the expectations. It can be said that the seed position slightly affects the properties of the synthesised system, but it is not entirely clear from the obtained data on how the seed position affects the properties of the synthesised system.



a) Vertical b) Full c) Diagonal

Figure 4.11. Examples of symmetric patterns used for testing the influence of seed position to synthesised TAS.

	Left-Bottom		Left-Top		Right-Top		Right-Bottom	
Symmetry	$\|T\|$	$\|\Sigma\|$	$\|T\|$	$\|\Sigma\|$	$\|T\|$	$\|\Sigma\|$	$\|T\|$	$\|\Sigma\|$
Vertical	19	9	19	9	**17**	**7**	17	8
	16	**7**	**16**	**7**	16	9	16	8
	18	7	19	9	18	7	**16**	**8**
	17	6	17	7	17	9	**15**	**8**
	17	**8**	18	8	20	8	19	9
Full	18	7	18	7	18	7	**16**	**8**
	15	**7**	16	6	**15**	**7**	**15**	**7**
	12	**5**	**12**	**5**	12	7	15	7
	18	8	18	8	**15**	**8**	18	7
	15	5	**14**	**7**	17	8	16	7
Diagonal	**17**	**9**	18	7	18	9	18	9
	16	8	18	7	**14**	**8**	17	8
	17	**9**	18	9	**17**	**9**	18	8
	19	9	18	8	**16**	**8**	18	8
	17	7	19	8	18	9	**15**	**8**

Table 4.5. Comparison of the most optimal solutions to symmetric patterns found depending on the position of seed structure, where $|T|$ is the number of tiles found, and $|\Sigma|$ is the number of glues. Presented results are the best solutions found after 24 optimization cycles. The size of 2-coloured patterns used for synthesis was 8×8.

Chapter 5
Conclusions

The theory of tile-based self-assembly systems has been presented, and the basic concepts of the theory were thoroughly discussed in the theory chapter. Two applications were successfully developed to their first usable versions using theoretical knowledge. The MuTATOR simulator is an entirely original design, and it is capable of simulating two types of abstract tile assembly systems. MuTATOR has been experimentally verified that the developed algorithms are able to simulate an aTAM model with linear computation complexity. Also, it has been shown that a 2HAM model is simulated with exponential computation complexity.

The second developed application can be used to generate tile assembly systems which self-assemble into the desired pattern. The algorithm used by MuTAGEN application is an implementation of the Partition Search with Heuristics algorithm for solving the PATS problem. It has been experimentally demonstrated that the synthesiser is able to find a nearly optimal solution in reasonable amounts of time with the relation to the size of the desired pattern. The influence of particular properties of a pattern used for the tile assembly system synthesis have been experimentally tested, and the result presented. Additionally, the application offers a variety of options which can be used to modify the synthesised system.

Both applications were developed with specific requirements of the EXPRO project in mind, and both applications are intended to be used for research purposes. The MuTATOR application could be used as a framework for developing novel tile assembly models which would model some aspects of tile-based self-assembly of macroscale systems.

5.1 Ideas for future enhancements

Future work could include further optimisation of simulation algorithms, especially the simulation of 2HAM. In the current state, the 2HAM simulator works only in the single computation thread. By using the multi-threaded solution, the 2HAM simulation could be several times faster.

There exists an option in MuTAGEN application to perform a batch of measurements to obtain a set of durations of a simulation to a given step. However, the batch processing cannot run in headless mode in the current state, i.e., running the application without the GUI. Implementation of headless mode to MuTAGEN would make it easier to use the application in scripts, and thus, the results of simulations could be processed in another program.

Another field of improvement could be the development of novel better heuristics or meta-heuristics for the MuTAGEN application. The new heuristic could help direct the solution to be better suited for macroscale self-assembly. The primary problem of macroscale self-assembly is the limited number of manufacturable types of mechanism which would mimic the behaviour of a glue in the abstract self-assembly models, whereas, in the nanoscale, a large number of glue types can be easily generated.

Therefore, methods for compensating the effects of the self-assembly of systems with an insufficient number of glues could be developed and tested within the MuTATOR framework. The methods could be based on concentration programming models, where the concentrations of different tile types in the system would be changed during the assembly process. Another method could utilise passing signals between the tiles or perhaps globally controlling which glues are active and inactive during the given time in the assembly process as this approach would be more suitable for macroscale implementation by utilising electronics. The method could incorporate aspects of the staged assembly model which would improve the ability to control the self-assembly process, but it would make the process more complicated.

These ideas could be implemented and tested in the MuTATOR or MuTAGEN applications which could help the research of the macroscale tile-based self-assembly systems.

References

[1] Bertoldi, Katia, Vincenzo Vitelli, Johan Christensen, and Martin van Hecke. Flexible mechanical metamaterials. *Nature Reviews Materials*. Nature Publishing Group, 2017, Vol. 2, No. 11, pp. 17066. Available from DOI 10.1038/natrevmats.2017.66.

[2] Ion, Alexandra, Johannes Frohnhofen, Ludwig Wall, Robert Kovacs, Mirela Alistar, Jack Lindsay, Pedro Lopes, Hsiang-Ting Chen, and Patrick Baudisch. Metamaterial Mechanisms. In: *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*. New York, NY, USA: ACM, 2016. pp. 529–539. UIST '16. ISBN 978-1-4503-4189-9. Available from DOI 10.1145/2984511.2984540. Available from `http://doi.acm.org/10.1145/2984511.2984540`.

[3] Philp, Douglas, and J. Fraser Stoddart. Self-Assembly in Natural and Unnatural Systems. *Angewandte Chemie International Edition in English*. 1996, Vol. 35, No. 11, pp. 1154-1196. Available from DOI 10.1002/anie.199611541. Available from `https://onlinelibrary.wiley.com/doi/abs/10.1002/anie.199611541`.

[4] Whitesides, George M., and Bartosz Grzybowski. Self-Assembly at All Scales. *Science*. American Association for the Advancement of Science, 2002, Vol. 295, No. 5564, pp. 2418–2421. ISSN 0036-8075. Available from DOI 10.1126/science.1070821. Available from `https://science.sciencemag.org/content/295/5564/2418`.

[5] Santos, Daniel, Matthew Spenko, Aaron Parness, and Mark Cutkosky. Directional adhesion for climbing: theoretical and practical considerations. *Journal of Adhesion Science and Technology*. Taylor & Francis, aug, 2007, Vol. 21, No. 12-13, pp. 1317–1341. Available from DOI 10.1163/156856107782328399.

[6] Doty, David. Theory of Algorithmic Self-assembly. *Commun. ACM*. New York, NY, USA: ACM, dec, 2012, Vol. 55, No. 12, pp. 78–88. ISSN 0001-0782. Available from DOI 10.1145/2380656.2380675.

[7] Adleman, LM. Molecular computation of solutions to combinatorial problems. *Science*. American Association for the Advancement of Science, nov, 1994, Vol. 266, No. 5187, pp. 1021–1024. ISSN 0036-8075. Available from DOI 10.1126/science.7973651. Available from `https://science.sciencemag.org/content/266/5187/1021`.

[8] Winfree, Erik. *Algorithmic self-assembly of DNA*. California Institute of Technology, may, 1998. Ph.D. Thesis. Available from `http://resolver.caltech.edu/CaltechETD:etd-05192003-110022`.

[9] Seeman, Nadrian C. Nucleic acid junctions and lattices. *Journal of Theoretical Biology*. nov, 1982, Vol. 99, No. 2, pp. 237–247. ISSN 0022-5193. Available from DOI 10.1016/0022-5193(82)90002-9. Available from `http://www.sciencedirect.com/science/article/pii/0022519382900029`.

[10] Wang, H. Proving theorems by pattern recognition – II. *The Bell System Technical Journal*. jan, 1961, Vol. 40, No. 1, pp. 1–41. ISSN 0005-8580. Available from DOI 10.1002/j.1538-7305.1961.tb03975.x.

[11] Winfree, Erik, Furong Liu, Lisa A Wenzler, and Nadrian C Seeman. Design and self-assembly of two-dimensional DNA crystals. *Nature*. Nature Publishing Group, aug, 1998, Vol. 394, No. 6693, pp. 539–544. Available from DOI 10.1038/28998.

[12] Rothemund, Paul W. K., and Erik Winfree. The Program-size Complexity of Self-assembled Squares. In: *Proceedings of the Thirty-second Annual ACM Symposium on Theory of Computing*. New York, NY, USA: ACM, 2000. pp. 459–468. STOC '00. ISBN 1-58113-184-4. Available from DOI 10.1145/335305.335358.

[13] Rothemund, Paul WK. Folding DNA to create nanoscale shapes and patterns. *Nature*. Nature Publishing Group, mar, 2006, Vol. 440, No. 7082, pp. 297. Available from DOI 10.1038/nature04586.

[14] Rothemund, Paul W. K, Nick Papadakis, and Erik Winfree. Algorithmic Self-Assembly of DNA Sierpinski Triangles. *PLOS Biology*. Public Library of Science, dec, 2004, Vol. 2, No. 12. Available from DOI 10.1371/journal.pbio.0020424.

[15] Chen, Holin, Ashish Goel, Chris Luhrs, and Erik Winfree. Self-assembling tile systems that heal from small fragments. *Presented at the thirteenth International meeting on DNA based computers (DNA)*. 2007. Available from
`https://web.stanford.edu/~ashishg/papers/selfhealing2.pdf`.

[16] Kautz, Steven M, and James I Lathrop. Self-assembly of the Discrete Sierpinski carpet and Related Fractals. *DNA Computing and Molecular Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, jan, 2009, pp. 78–87. ISSN 1611-3349. Available from DOI 10.1007/978-3-642-10604-0_8.

[17] Patitz, Matthew J. An introduction to tile-based self-assembly and a survey of recent results. *Natural Computing*. Springer Netherlands, 2014, Vol. 13, No. 2, pp. 195–224. ISSN 15729796. Available from DOI 10.1007/s11047-013-9379-4.

[18] Thorkelsson, Kari, Peter Bai, and Ting Xu. Self-assembly and applications of anisotropic nanomaterials: A review. *Nano Today*. Elsevier Ltd, 2015, Vol. 10, No. 1, pp. 48–66. ISSN 1878044X. Available from
`https://www.sciencedirect.com/science/article/pii/S1748013214001613`.

[19] Baron, Alexandre, Ashod Aradian, Virginie Ponsinet, and Philippe Barois. Self-assembled optical metamaterials. *Optics and Laser Technology*. Elsevier, 2016, Vol. 82, pp. 94–100. ISSN 00303992. Available from
`https://www.sciencedirect.com/science/article/pii/S0030399216301153`.

[20] Gomez-Graña, Sergio, Aurélie Le Beulze, Mona Treguer-Delapierre, Stéphane Mornet, Etienne Duguet, Eftychia Grana, Eric Cloutet, Georges Hadziioannou, Jacques Leng, Jean-Baptiste Salmon, Vasyl G. Kravets, Alexander N. Grigorenko, Naga A. Peyyety, Virginie Ponsinet, Philippe Richetti, Alexandre Baron, Daniel Torrent, and Philippe Barois. Hierarchical self-assembly of nanoparticles for optical metamaterials. 2016, pp. 1–21. Available from
`http://arxiv.org/abs/1606.02105`.

[21] McManus, Jennifer J., Patrick Charbonneau, Emanuela Zaccarelli, and Neer Asherie. The physics of protein self-assembly. *Current Opinion in Colloid and Interface Science*. The Authors, 2016, Vol. 22, pp. 73–79. ISSN 1359-0294. Available

from DOI 10.1016/j.cocis.2016.02.011. Available from
https://www.sciencedirect.com/science/article/pii/S1359029416300292.

[22] Sun, Hongcheng, Quan Luo, Chunxi Hou, and Junqiu Liu. Nanostructures based on protein self-assembly: From hierarchical construction to bioinspired materials. *Nano Today*. Elsevier Ltd, 2017, Vol. 14, pp. 16–41. ISSN 1878044X. Available from DOI 10.1016/j.nantod.2017.04.006. Available from
https://www.sciencedirect.com/science/article/pii/S1748013216305436.

[23] Levchenko, Igor, Kateryna Bazaka, Michael Keidar, Shuyan Xu, and Jinghua Fang. Hierarchical Multicomponent Inorganic Metamaterials: Intrinsically Driven Self-Assembly at the Nanoscale. *Advanced Materials*. Wiley-Blackwell, 2018, Vol. 30, No. 2, pp. 1702226. ISSN 09359648. Available from DOI 10.1002/adma.201702226.

[24] Haghighat, Bahar, Brice Platerrier, Loic Waegeli, and Alcherio Martinoli. Synthesizing rulesets for programmable robotic self-assembly: A case study using floating miniaturized robots. In: Mauro Dorigo, Marcoand Birattari, Xiaodong Li, Manuel López-Ibáñez, Kazuhiro Ohkura, Carlo Pinciroli, and Thomas Stützle, eds. *International Conference on Swarm Intelligence*. Cham: Springer International Publishing, 2016. pp. 197–209. ISBN 978-3-319-44427-7. Available from DOI 10.1007/978-3-319-44427-7_17.

[25] Haghighat, Bahar, and Alcherio Martinoli. Automatic synthesis of rulesets for programmable stochastic self-assembly of rotationally symmetric robotic modules. *Swarm Intelligence*. Springer US, 2017, Vol. 11, No. 3-4, pp. 243–270. ISSN 19353820. Available from DOI 10.1007/s11721-017-0139-4.

[26] Rubenstein, Michael, Alejandro Cornejo, and Radhika Nagpal. Programmable self-assembly in a thousand-robot swarm. *Science*. American Association for the Advancement of Science, 2014, Vol. 345, No. 6198, pp. 795–799. ISSN 0036-8075. Available from DOI 10.1126/science.1254295. Available from
http://www.ncbi.nlm.nih.gov/pubmed/25124435.

[27] Gauci, Melvin, Radhika Nagpal, and Michael Rubenstein. Programmable Self-disassembly for Shape Formation in Large-Scale Robot Collectives. In: Roderich Gross, Andreas Kolling, Spring Berman, Emilio Frazzoli, Alcherio Martinoli, Fumitoshi Matsuno, and Melvin Gauci, eds. *Distributed Autonomous Robotic Systems: The 13th International Symposium*. Cham: Springer International Publishing, 2018. pp. 573–586. Available from DOI 10.1007/978-3-319-73008-0_40.

[28] Doty, David, Jack H Lutz, Matthew J Patitz, Scott M Summers, and Damien Woods. Intrinsic universality in self-assembly. In: *Proceedings of the 27th international symposium on theoretical aspects of computer science*. 2009. pp. 275–286. Available from
http://arxiv.org/abs/1001.0208v2.

[29] Doty, D., J. H. Lutz, M. J. Patitz, R. T. Schweller, S. M. Summers, and D. Woods. The Tile Assembly Model is Intrinsically Universal. In: *2012 IEEE 53rd Annual Symposium on Foundations of Computer Science*. 2012. pp. 302–310. ISSN 0272-5428. Available from DOI 10.1109/FOCS.2012.76.

[30] Demaine, Erik D., Sarah Eisenstat, Mashhood Ishaque, and Andrew Winslow. One-dimensional staged self-assembly. *Natural Computing*. Springer Netherlands, 2013, Vol. 12, No. 2, pp. 247–258. ISSN 15677818. Available from DOI 10.1007/s11047-012-9359-0.

[31] Woods, Damien. Intrinsic universality and the computational power of self-assembly. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*. 2015, Vol. 373, No. 2046. ISSN 1364503X. Available from DOI 10.1098/rsta.2014.0214.

[32] Furcy, David, and Scott M. Summers. Optimal Self-Assembly of Finite Shapes at Temperature 1 in 3D. *Algorithmica*. Springer US, 2018, Vol. 80, No. 6, pp. 1909–1963. ISSN 14320541. Available from DOI 10.1007/s00453-016-0260-6.

[33] Demaine, Erik D., Sándor P. Fekete, Christian Scheffer, and Arne Schmidt. New geometric algorithms for fully connected staged self-assembly. *Theoretical Computer Science*. Elsevier B.V., 2017, Vol. 671, pp. 4–18. ISSN 03043975. Available from DOI 10.1016/j.tcs.2016.11.020. Available from
https://www.sciencedirect.com/science/article/pii/S030439751630679X?via%3Dihub.

[34] Chalk, Cameron, Eric Martinez, Robert Schweller, Luis Vega, Andrew Winslow, and Tim Wylie. Optimal Staged Self-Assembly of General Shapes. *Algorithmica*. Springer US, 2018, Vol. 80, No. 4, pp. 1383–1409. ISSN 14320541. Available from DOI 10.1007/s00453-017-0318-0.

[35] Chen, Ho Lin, David Doty, and Shinnosuke Seki. Program Size and Temperature in Self-Assembly. *Algorithmica*. Springer US, 2015, Vol. 72, No. 3, pp. 884–899. ISSN 14320541. Available from DOI 10.1007/s00453-014-9879-3.

[36] Evans, Constantine G., and Erik Winfree. Optimizing Tile Set Size While Preserving Proofreading with a DNA Self-assembly Compiler. In: David Doty, and Hendrik Dietz, eds. *DNA Computing and Molecular Programming: 24th International Conference*. Jinan, China: Springer International Publishing, 2018. pp. 37–54. ISBN 978-3-030-00030-1. Available from DOI 10.1007/978-3-030-00030-1_3.

[37] Johnsen, Aleck C., Ming-Yang Kao, and Shinnosuke Seki. Computing Minimum Tile Sets to Self-Assemble Colors Patterns. *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2014, pp. 699–710. ISSN 1611-3349. Available from DOI 10.1007/978-3-642-45030-3_65. Available from
http://arxiv.org/abs/1404.2962.

[38] Seki, Shinnosuke, and Andrew Winslow. The Complexity of Fixed-Height Patterned Tile Self-Assembly. 2016, pp. 1–22. Available from
http://arxiv.org/abs/1604.07190.

[39] Göös, Mika, and Pekka Orponen. Synthesizing Minimal Tile Sets for Patterned DNA Self-Assembly. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2009, Vol. 6518 LNCS, pp. 71–82. ISSN 03029743. Available from DOI 10.1007/978-3-642-18305-8_7. Available from
http://arxiv.org/abs/0911.2924.

[40] Adleman, Len, Qi Cheng, Ashish Goel, Ming-Deh Huang, David Kempe, Pablo Moisset de Espanés, and Paul Wilhelm Karl Rothemund. Combinatorial optimization problems in self-assembly. In: *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing*. New York, NY, USA: ACM Press, 2002. pp. 23–32. STOC '02. ISBN 1-58113-495-9. Available from DOI 10.1145/509907.509913.

[41] Lathrop, James I., Jack H. Lutz, and Scott M. Summers. Strict self-assembly of discrete Sierpinski triangles. *Theoretical Computer Science*. Elsevier B.V.,

2009, Vol. 410, No. 4-5, pp. 384–405. ISSN 03043975. Available from DOI 10.1016/j.tcs.2008.09.062. Available from
`https://www.sciencedirect.com/science/article/pii/S030439750800724X`.

[42] Soloveichik, David, and Erik Winfree. Complexity of Self-Assembled Shapes. *SIAM Journal on Computing*. jan, 2007, Vol. 36, No. 6, pp. 1544–1569. ISSN 0097-5397. Available from DOI 10.1137/S0097539704446712.

[43] Chen, Ho-Lin, Rebecca Schulman, Ashish Goel, and Erik Winfree. Reducing Facet Nucleation during Algorithmic Self-Assembly. *Nano Letters*. 2007, Vol. 7, No. 9, pp. 2913-2919. Available from DOI 10.1021/nl070793o. Available from
`https://pubs.acs.org/doi/abs/10.1021/nl070793o`. PMID: 17718529.

[44] Aggarwal, Gagan, Qi Cheng, Michael H. Goldwasser, Ming-Yang Kao, Pablo Moisset de Espanes, Schweller, Robert T., Pablo Moisset de Espanes, and Robert T Schweller. Complexities for Generalized Models of Self-Assembly. *SIAM Journal on Computing*. Society for Industrial and Applied Mathematics, 2005, Vol. 34, No. 6, pp. 1493–1515. ISSN 0097-5397. Available from DOI 10.1137/S0097539704445202.

[45] Demaine, Erik D., Martin L. Demaine, Sándor P. Fekete, Mashhood Ishaque, Eynat Rafalin, Robert T. Schweller, and Diane L. Souvaine. Staged self-assembly: Nanomanufacture of arbitrary shapes with O(1) glues. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Springer Netherlands, 2008, Vol. 4848 LNCS, No. 3, pp. 1–14. ISSN 03029743. Available from DOI 10.1007/978-3-540-77962-9_1.

[46] Chen, Ho-Lin, and David Doty. Parallelism and Time in Hierarchical Self-assembly. In: *Proceedings of the Twenty-third Annual ACM-SIAM Symposium on Discrete Algorithms*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2012. pp. 1163–1182. SODA '12. Available from
`http://dl.acm.org/citation.cfm?id=2095116.2095208`.

[47] Cannon, Sarah, Erik D. Demaine, Martin L. Demaine, Sarah Eisenstat, Matthew J. Patitz, Robert Schweller, Scott M. Summers, and Andrew Winslow. Two Hands Are Better Than One (up to constant factors). *Stacs 2013*. 2012, No. January. ISSN 01616382. Available from DOI 10.4230/LIPIcs.STACS.2013.172. Available from
`http://arxiv.org/abs/1201.1650`.

[48] Demaine, Erik D., Matthew J. Patitz, Trent A. Rogers, Robert T. Schweller, Scott M. Summers, and Damien Woods. The Two-Handed Tile Assembly Model is not Intrinsically Universal. *Algorithmica*. feb, 2016, Vol. 74, No. 2, pp. 812–850. ISSN 1432-0541. Available from DOI 10.1007/s00453-015-9976-y.

[49] Behsaz, Bahar, Ján Maňuch, and Ladislav Stacho. Turing Universality of Step-Wise and Stage Assembly at Temperature 1. In: Darko Stefanovic, and Andrew Turberfield, eds. *DNA Computing and Molecular Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. pp. 1–11. ISBN 978-3-642-32208-2. Available from
`https://link.springer.com/chapter/10.1007/978-3-642-32208-2_1`.

[50] Reif, John H. Local Parallel Biomolecular Computation. In: *Proceedings of DNA Based Computers III, DIMACS*. American Mathematical Society, 1999. pp. 217–254. Available from
`https://users.cs.duke.edu/~reif/paper/DNAassembly/Assembly.pdf`.

[51] CHANDRAN, Harish, Nikhil GOPALKRISHNAN, and John REIF. The Tile Complexity of Linear Assemblies. In: Susanne ALBERS, Alberto MARCHETTI-SPACCAMELA, Yossi MATIAS, Sotiris NIKOLETSEAS, and Wolfgang THOMAS, eds. *Automata, Languages and Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. pp. 235–253. ISBN 978-3-642-02927-1. Available from
`https://link.springer.com/chapter/10.1007/978-3-642-02927-1_21`.

[52] BECKER, Florent, Éric RÉMILA, and Ivan RAPAPORT. Self-assemblying Classes of Shapes with a Minimum Number of Tiles, and in Optimal Time. In: *FSTTCS*. 2006. pp. 45–56. LNCS. Available from DOI 10.1007/11944836_7. Available from
`https://hal.archives-ouvertes.fr/hal-00460570`.

[53] KARI, Lila, Shinnosuke SEKI, and Zhi XU. Triangular and Hexagonal Tile Self-assembly Systems. In: *Proceedings of the 2012 International Conference on Theoretical Computer Science: Computation, Physics and Beyond*. Berlin, Heidelberg: Springer-Verlag, 2012. pp. 357–375. WTCS'12. ISBN 978-3-642-27653-8. Available from DOI 10.1007/978-3-642-27654-5_28. Available from
`https://link.springer.com/chapter/10.1007/978-3-642-27654-5_28`.

[54] JONOSKA, Natasa, Stephen A. KARL, and Masahico SAITO. Three dimensional DNA structures in computing. *Biosystems*. 1999, Vol. 52, No. 1, pp. 143 - 153. ISSN 0303-2647. Available from DOI 10.1016/S0303-2647(99)00041-6. Available from
`http://www.sciencedirect.com/science/article/pii/S0303264799000416`.

[55] DEMAINE, Erik D., Martin L. DEMAINE, Sándor P. FEKETE, Matthew J. PATITZ, Robert T. SCHWELLER, Andrew WINSLOW, and Damien WOODS. One Tile to Rule Them All: Simulating Any Tile Assembly System with a Single Universal Tile. In: Javier ESPARZA, Pierre FRAIGNIAUD, Thore HUSFELDT, and Elias KOUTSOUPIAS, eds. *Automata, Languages, and Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014. pp. 368–379. ISBN 978-3-662-43948-7. Available from
`https://link.springer.com/chapter/10.1007%2F978-3-662-43948-7_31`.

[56] SUMMERS, Scott M. Reducing Tile Complexity for the Self-assembly of Scaled Shapes Through Temperature Programming. *Algorithmica*. Jun, 2012, Vol. 63, No. 1, pp. 117–136. ISSN 1432-0541. Available from DOI 10.1007/s00453-011-9522-5.

[57] DOTY, David, Lila KARI, and Benoît MASSON. Negative Interactions in Irreversible Self-assembly. *Algorithmica*. May, 2013, Vol. 66, No. 1, pp. 153–172. ISSN 1432-0541. Available from DOI 10.1007/s00453-012-9631-9. Available from
`https://link.springer.com/article/10.1007/s00453-012-9631-9`.

[58] PATITZ, Matthew J., Robert T. SCHWELLER, and Scott M. SUMMERS. Exact Shapes and Turing Universality at Temperature 1 with a Single Negative Glue. In: Luca CARDELLI, and William SHIH, eds. *DNA Computing and Molecular Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. pp. 175–189. ISBN 978-3-642-23638-9. Available from
`https://link.springer.com/chapter/10.1007%2F978-3-642-23638-9_15`.

[59] PADILLA, Jennifer E., Wenyan LIU, and Nadrian C. SEEMAN. Hierarchical self assembly of patterns from the Robinson tilings: DNA tile design in an enhanced Tile Assembly Model. *Natural Computing*. Springer Netherlands, 2012, Vol. 11, No. 2, pp. 323–338. ISSN 15677818. Available from DOI 10.1007/s11047-011-9268-7.

[60] Padilla, Jennifer E., Matthew J. Patitz, Raul Pena, Robert T. Schweller, Nadrian C. Seeman, Robert Sheline, Scott M. Summers, and Xingsi Zhong. Asynchronous Signal Passing for Tile Self-assembly: Fuel Efficient Computation and Efficient Assembly of Shapes. In: Giancarlo Mauri, Alberto Dennunzio, Luca Manzoni, and Antonio E. Porreca, eds. *Unconventional Computation and Natural Computation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. pp. 174–185. ISBN 978-3-642-39074-6. Available from `https://link.springer.com/chapter/10.1007/978-3-642-39074-6_17`.

[61] Jonoska, Nataša, and Daria Karpenko. Active Tile Self-assembly, Self-similar Structures and Recursion. 11, 2012. Available from `https://arxiv.org/pdf/1211.3085.pdf`.

[62] Jonoska, Nataša, and Daria Karpenko. Active Tile Self-Assembly, Part 2: Self-Similar Structures and Structural Recursion. *International Journal of Foundations of Computer Science*. 2014, Vol. 25, No. 02, pp. 165-194. Available from DOI 10.1142/S0129054114500099.

[63] Adleman, Leonard, Qi Cheng, Ashish Goel, and Ming-Deh Huang. Running time and program size for self-assembled squares. *Proceedings of the thirty-third annual ACM symposium on Theory of computing - STOC '01*. New York, New York, USA: ACM Press, 2003, pp. 740–748. Available from DOI 10.1145/380752.380881. Available from `http://portal.acm.org/citation.cfm?doid=380752.380881`.

[64] Becker, Florent. Pictures worth a thousand tiles, a geometrical programming language for self-assembly. *Theoretical Computer Science*. Elsevier B.V., 2009, Vol. 410, No. 16, pp. 1495–1515. ISSN 03043975. Available from DOI 10.1016/j.tcs.2008.12.011.

[65] Becker, Florent, Éric Rémila, and Nicolas Schabanel. Time optimal self-assembly for 2D and 3D shapes: The case of squares and cubes. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Springer, Berlin, Heidelberg, 2009, Vol. 5347 LNCS, pp. 144–155. ISSN 03029743. Available from DOI 10.1007/978-3-642-03076-5_12.

[66] Ma, Xiaojun, and Fabrizio Lombardi. Synthesis of tile sets for DNA self-assembly. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. may, 2008, Vol. 27, No. 5, pp. 963–967. ISSN 02780070. Available from DOI 10.1109/TCAD.2008.917973. Available from `http://ieeexplore.ieee.org/document/4492836/`.

[67] Seki, Shinnosuke. Combinatorial optimization in pattern assembly (Extended Abstract). *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Springer, Berlin, Heidelberg, 2013, Vol. 7956 LNCS, No. 902184, pp. 220–231. ISSN 03029743. Available from DOI 10.1007/978-3-642-39074-6-21.

[68] Czeizler, Eugen, and Alexandru Popa. Synthesizing minimal tile sets for complex patterns in the framework of patterned DNA self-assembly. *Theoretical Computer Science*. Springer, Berlin, Heidelberg, 2013, Vol. 499, pp. 23–37. ISSN 03043975. Available from DOI 10.1016/j.tcs.2013.05.009.

[69] Lempiäinen, Tuomo, Eugen Czeizler, and Pekka Orponen. Synthesizing Small and Reliable Tile Sets for Patterned DNA Self-assembly. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lec-*

ture Notes in Bioinformatics). Springer, Berlin, Heidelberg, 2011, Vol. 6937 LNCS, pp. 145–159. ISSN 03029743. Available from DOI 10.1007/978-3-642-23638-9_13.

[70] Göös, Mika, Tuomo Lempiäinen, Eugen Czeizler, and Pekka Orponen. Search methods for tile sets in patterned DNA self-assembly. *Journal of Computer and System Sciences*. Elsevier Inc., 2014, Vol. 80, No. 1, pp. 297–319. ISSN 00220000. Available from DOI 10.1016/j.jcss.2013.08.003.

[71] Winfree, Erik, Rebecca Schulman, and Constantine Evans. *Xgrow Simulator* [online]. 2003. [Accessed 2019-05-07]. Available from `http://www.dna.caltech.edu/Xgrow/`.

[72] Patitz, Matthew J. Simulation of self-assembly in the abstract tile assembly model with ISU TAS. *In: FNANO*. 2009, pp. 56–69. Available from `http://arxiv.org/abs/1101.5151`.

[73] Patitz, Matthew, Trent Rogers, and Michael Sharp. *ISU TAS* [online]. 2018. [Accessed 2019-05-07]. Available from `http://self-assembly.net/wiki/index.php?title=ISU_TAS&oldid=1489`.

[74] Patitz, Matthew, Trent Rogers, and Michael Sharp. *PyTAS* [online]. 2018. [Accessed 2019-05-07]. Available from `http://self-assembly.net/wiki/index.php?title=PyTAS&oldid=1490`.

[75] Meyers, Scott. *Effective modern C++*. 1st ed. Sebastopol, California: O'Reilly Media, 2014. ISBN 978-1-491-90399-5.

[76] *Cppreference.com* [online]. 2000. [Accessed 2019-05-07]. Available from `https://en.cppreference.com/`.

[77] Cumming, Murray, Bernhard Rieder, Jonathon Jongsma, Ole Laursen, Marko Anastasov, Daniel Elstner, Chris Vine, David King, Pedro Ferreira, and Kjell Ahlstedt. *Programming with gtkmm 3* [online]. 2005. [Accessed 2019-05-07]. Available from `https://developer.gnome.org/gtkmm-tutorial/3.24/`.

[78] The GNOME Project. *Gtkmm: gtkmm Reference Manual* [online]. 2005. [Accessed 2019-05-07]. Available from `https://developer.gnome.org/gtkmm/3.18/`.

[79] gabime. *spdlog* [online]. 2016. [Accessed 2019-05-07]. Available from `https://github.com/gabime/spdlog`.

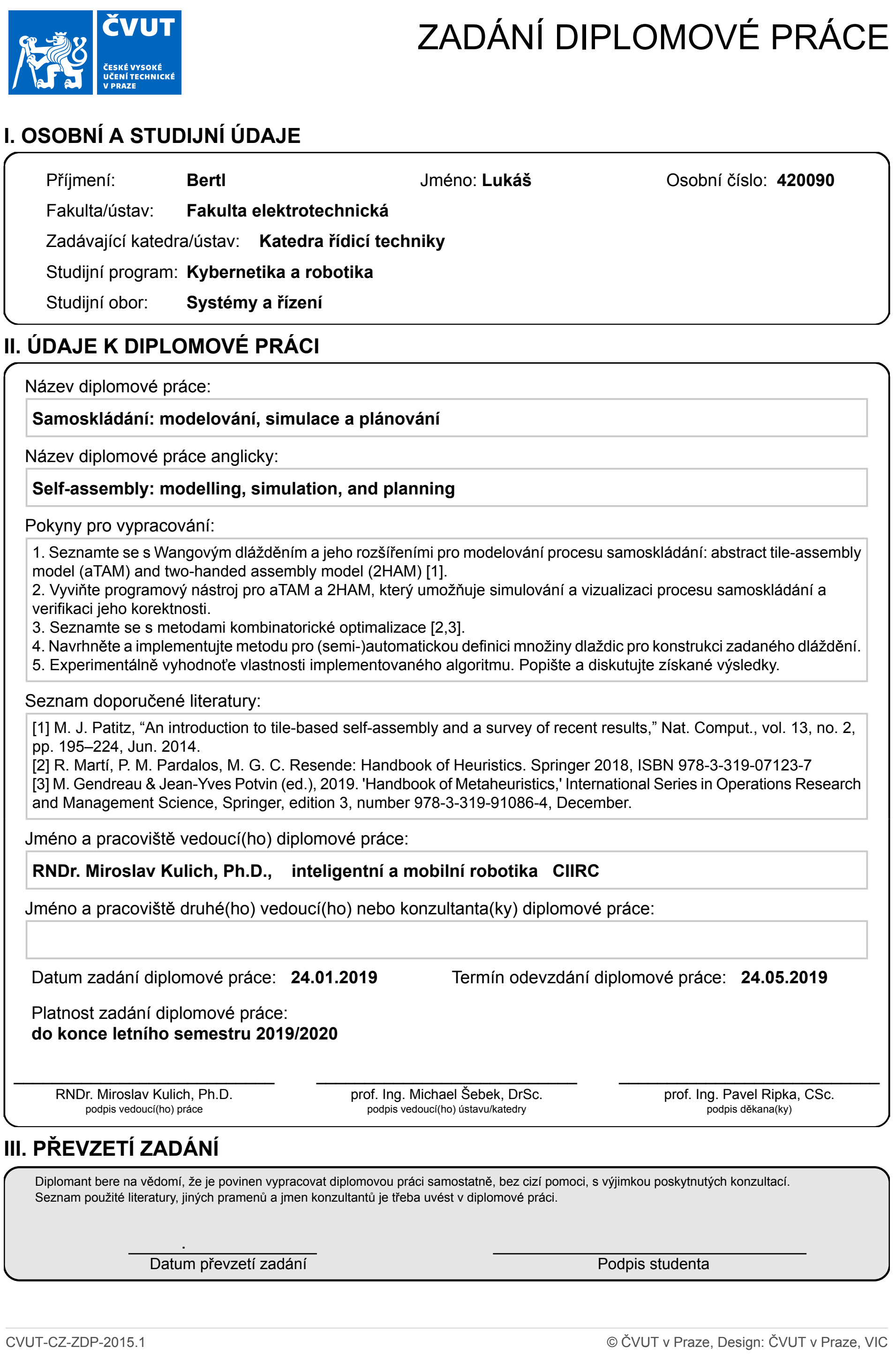
[80] Crockford, Douglas. *JSON* [online]. 1999. [Accessed 2019-05-07]. Available from `http://json.org/`.

[81] Standard ECMA-404. *The JSON data interchange format* [online]. 2017. [Accessed 2019-05-07]. Available from `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf`.

[82] Kari, Lila, Steffen Kopecki, Pierre-Étienne Meunier, Matthew J. Patitz, and Shinnosuke Seki. Binary Pattern Tile Set Synthesis Is NP-Hard. *Algorithmica*. Springer Nature, apr, 2016, Vol. 78, No. 1, pp. 1–46. ISSN 1432-0541. Available from DOI 10.1007/s00453-016-0154-7. Available from `https://link.springer.com/article/10.1007%2Fs00453-016-0154-7`.

[83] Kopecki, Steffen. *2PATS* [online]. 2014. [Accessed 2019-05-12]. Available from `http://self-assembly.net/wiki/index.php?title=2PATS-tileset-search`.

Appendix A
Specification



ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení:	**Bertl**	Jméno: **Lukáš**	Osobní číslo: **420090**
Fakulta/ústav:	**Fakulta elektrotechnická**		
Zadávající katedra/ústav:	**Katedra řídicí techniky**		
Studijní program:	**Kybernetika a robotika**		
Studijní obor:	**Systémy a řízení**		

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Samoskládání: modelování, simulace a plánování

Název diplomové práce anglicky:

Self-assembly: modelling, simulation, and planning

Pokyny pro vypracování:

1. Seznamte se s Wangovým dlážděním a jeho rozšířeními pro modelování procesu samoskládání: abstract tile-assembly model (aTAM) and two-handed assembly model (2HAM) [1].
2. Vyviňte programový nástroj pro aTAM a 2HAM, který umožňuje simulování a vizualizaci procesu samoskládání a verifikaci jeho korektnosti.
3. Seznamte se s metodami kombinatorické optimalizace [2,3].
4. Navrhněte a implementujte metodu pro (semi-)automatickou definici množiny dlaždic pro konstrukci zadaného dláždění.
5. Experimentálně vyhodnoťte vlastnosti implementovaného algoritmu. Popište a diskutujte získané výsledky.

Seznam doporučené literatury:

[1] M. J. Patitz, “An introduction to tile-based self-assembly and a survey of recent results,” Nat. Comput., vol. 13, no. 2, pp. 195–224, Jun. 2014.
[2] R. Martí, P. M. Pardalos, M. G. C. Resende: Handbook of Heuristics. Springer 2018, ISBN 978-3-319-07123-7
[3] M. Gendreau & Jean-Yves Potvin (ed.), 2019. 'Handbook of Metaheuristics,' International Series in Operations Research and Management Science, Springer, edition 3, number 978-3-319-91086-4, December.

Jméno a pracoviště vedoucí(ho) diplomové práce:

RNDr. Miroslav Kulich, Ph.D., inteligentní a mobilní robotika CIIRC

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **24.01.2019** Termín odevzdání diplomové práce: **24.05.2019**

Platnost zadání diplomové práce:
do konce letního semestru 2019/2020

RNDr. Miroslav Kulich, Ph.D.	prof. Ing. Michael Šebek, DrSc.	prof. Ing. Pavel Ripka, CSc.
podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

.
Datum převzetí zadání Podpis studenta

CVUT-CZ-ZDP-2015.1 © ČVUT v Praze, Design: ČVUT v Praze, VIC

Appendix B
Glossary of Abbreviations

API	Application Programming Interface
aTAM	abstract Tile Assembly Model
ATAM	Active Tile Assembly Model
CIIRC	Czech Institute of Informatics, Robotics and Cybernetics
CTU	Czech Technical University
DNA	Deoxyribonucleic acid
GIMP	GNU Image Manipulation Program
GNOME	GNU Network Object Model Environment
GNU	GNU's Not Unix
GTK	GIMP Toolkit
GUI	Graphical User Interface
IMR	Intelligent and Mobile Robotics group
ISU TAS	Iowa State University Tile Assembly Simulator
kTAM	kinetic Tile Assembly Model
MuTAGEN	the Meta-Material Tile Assembly system GENerator
MuTATOR	the Meta-Material Tile Assembly sysTem simulatOR
OOP	Object-Oriented Programming
PATS	Patterned self-Assembly Tile set Synthesis problem
PTAM	Probabilistic Tile Assembly Model
PyTAS	Python-based Tile Assembly Simulator
rgTAM	restricted glue Tile Assembly Model
SAS	Staged self-Assembly System model
STAM	Signal passing Tile Assembly Model
TAS	Tile Assembly System
WYSIWYG	What You See Is What You Get
XML	eXtensible Markup Language
2HAM	Two-Handed Assembly Model

Appendix C
Contents of the CD

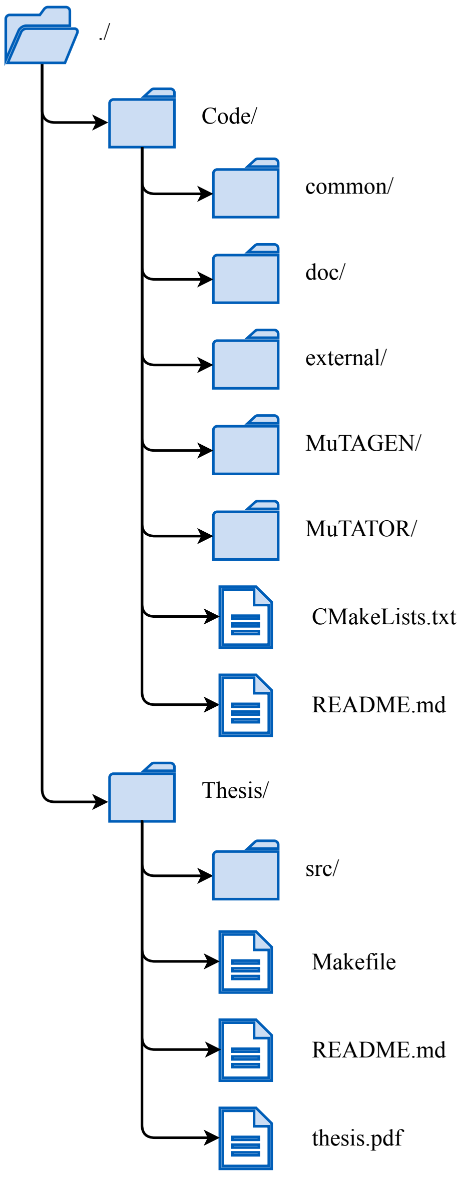


Figure C.1. Diagram showing the directory structure of the CD.

The diagram in Figure C.1 shows the structure of contents of the enclosed CD. There is a `README.md` file for each part of the project which informs about contents of the directory where is located.

The `./Code/` directory contains source files of the MuTATOR and MuTAGEN applications in `./Code/MuTATOR/` and `./Code/MuTAGEN/` directories, respectively.

The `./Thesis/` directory contains this thesis in the file named `thesis.pdf`, and source files of the thesis are located in the `./Thesis/src/` directory.

Appendix D
Code listings

This chapter includes several examples of our implementation of data structures and functions in MuTATOR and MuTAGEN applications.

D.1 MuTATOR and MuTAGEN Common Data Structures

D.1.1 Implementation of Tile

In this section, the data structure of `Tile` class is presented. The basic idea behind the `Tile` class implementation can be seen in Listing D.2.

```
 1  class Tile
 2  {
 3  public:
 4      enum class Side {
 5          North, East, South, West
 6      };
 7
 8      static constexpr std::array<Tile::Side, 4> sides = {
 9          Tile::Side::North,
10          Tile::Side::East,
11          Tile::Side::South,
12          Tile::Side::West
13      };
14
15  ...
16
17  private:
18      Glue m_north;
19      Glue m_east;
20      Glue m_south;
21      Glue m_west;
22      imr::JSON::Dictionary m_additional_info;
23  };
```

Listing D.2. Implementation of the `Tile` class data structure.

The implementation of the `Tile` class corresponds to the definition of the tile in the theory of tile-based self-assembling systems. Therefore, the `Tile` class consists of four `Glue` classes which represents glues for each side of the tile (lines 18–21). The `m_additional_info` member variable is used for storing additional information and configurations of the tile object such as its colour or label in JSON format (line 22).

Note the definition of enum class `Side` and the corresponding static array `sides` which are both often utilized in algorithms for accessing to glue on the given tile side or for easy iteration (lines 4–6 and 8–13).

```
1   class TileConfiguration
2   {
3   public:
4   ...
5
6   private:
7       TileAssemblySystem const m_tas;
8       std::unordered_map<Tile::Position, TileID> m_tile_map;
9       std::unordered_set<Tile::Position> m_perimeter;
10      std::unordered_set<Tile::Position>
11          m_perimeter_with_enough_bond_strength;
12      Boundary m_boundary;
13  };
```

Listing D.3. Implementation of the `TileConfiguration` class data structure.

D.1.2 Implementation of Tile Configuration

The `TileConfiguration` class representing an assembly of tiles in the MuTATOR simulator is shown in this section. The member variables of the `TileConfiguration` class are presented in Listing D.3.

The implementation of `TileConfiguraion` class is based on the theoretic definition of an assembly of tile assembly system. Therefore, the class contains member `m_tas` which encapsulates the tile assembly system used in the simulation. The member variable `m_tile_map` stores the configuration of tiles in the assembly space using mapping between 2D position vectors and `TileID` which are used to reference particular instance of `Tile` class owned by the `m_tas` variable (lines 7, 8).

For acceleration of various algorithms, the `TileConfiguration` class stores current configuration of its perimeter and perimeter positions with available bonding strength greater than system temperature, in `m_perimeter` and `m_perimeter_with_enough_bond_strength` variables, respectively (lines 9, 10). The `m_boundary` variable stores the positional limits of the assembly (line 11).

D.2 MuTATOR Algorithm Implementations

In this section, a couple of examples from the simulator implementation are described.

D.2.1 aTAM: Algorithm for finding the set of tiles to extend the assembly

The first presented example is a function which finds for the set all tiles which can extend the simulated assembly in the current step. One possible implementation can be seen in Listing D.4.

Please note that the code was reformated to fit the page, and comments were removed from it. The algorithm uses the current state of the simulation, and thus, it does not explicitly state any function inputs. However, the algorithm uses an instance of class `TileConfiguration` called `m_configuration`, which stores the current state of the simulated aTAM assembly.

First, the output vector of actions `possible_actions` is initialised (line 4). The action type `AtamSimulator::Action` consists of a pair of a position in the assembly and internal ID of a tile. Then, each position on the assembly space named `position` which lies in the *perimeter* of the assembly is checked in the loop (lines 6–8). The *perimeter*

```
1  std::vector<AtamSimulator::Action>
2      AtamSimulator::find_all_possible_actions() const
3  {
4      auto possible_actions = std::vector<AtamSimulator::Action>{};
5
6      for( auto const & position :
7           m_configuration.get_perimeter_with_enough_bond_strength()
8      ) {
9          auto candidate_tile_sides = std::array<Glue, 4>{};
10         for( auto const side : Tile::sides ) {
11             auto const side_i = static_cast<size_t>( side );
12             auto const neighbour_tile = m_configuration.get_tile_on(
13                 position + unit_vector( side )
14             );
15
16             candidate_tile_sides[ side_i ] =
17                 neighbour_tile[ complement(side) ];
18         }
19
20         auto const candidate_tile = Tile{
21             candidate_tile_sides[0], candidate_tile_sides[1],
22             candidate_tile_sides[2], candidate_tile_sides[3]
23         };
24
25         auto const compatible_tiles =
26             m_configuration.get_tas()
27             .find_tiles_with_matching_not_null_glues( candidate_tile );
28
29         for( auto const tile_id : compatible_tiles ) {
30             if( m_configuration.can_be_placed_on( position, tile_id ) ) {
31                 possible_actions.emplace_back( position, tile_id );
32             }
33         }
34     }
35
36     return possible_actions;
37 }
```

Listing D.4. Implementation of aTAM function which returns a set of all possible pairs of a tile and position suitable for addition to the simulated assembly.

of the assembly refers to the set of unoccupied positions directly neighbouring to the occupied positions in the assembly in this thesis. Additionally, the function used in line 7 filters the set of perimeter positions and returns only those positions where the combined glue strength of adjoined tiles is greater than or equal to system temperature. This filtration further reduces the number of cycles needed for the function to return.

An array of four instances of `Glue` class named `candidate_tile_sides` is initialised, which is used to hold glues of an ideal (most likely non-existent) tile that could be attached at `position` (line 9). The ideal candidate tile is generated by copying glues of the adjacent tiles and assigning them to complement sides of the tile (lines 10–18), and then an instance of `Tile` class named `candidate_tile` is constructed (lines 20–23).

The candidate tile is then used for finding a set of all tiles in the system with the same non-null glue sides as the candidate tile denoted as `compatible_tiles` (lines 25–27).

Finally, if there are any tiles in the `compatible_tiles` set, then each tile is tested whether it has sufficient glue strength to be placed on `position`. If the tile does have enough strength, it is paired up with the `position` and added to `possible_actions` vector (lines 29–33). This process is subsequently repeated for another position in the assembly perimeter. In the end, all suitable pairs of positions and tiles are found and returned (line 36).

In the actual implementation, the function is not used to find suitable tiles in every step of simulation as it is used only for the very first step. Because each found action stays valid for next steps until its corresponding position gets occupied, it is sufficient to search for new actions only in the local neighbourhood of the last attached tile using a similar function which does not search the whole assembly perimeter. The computation of the simulation step is significantly accelerated by saving all found actions and by updating them based only on the local changes in the assembly.

D.2.2 2HAM: Simulation step algorithm

The next algorithm implemented in the simulator and described here is a function which finds all possible combinations of supertiles in the current step of 2HAM simulation. One possible implementation is presented in Listing D.5.

Like the previously presented function, also this function does not have explicitly declared input, but it is using the internal representation of 2HAM simulation state. The 2HAM simulation state is defined as a vector of supertiles which are implemented using the `TileConfiguration` class. Note that a single instance of the same class is used to represent the state of the aTAM simulation, but there is a whole vector of those instances in the 2HAM simulation. One action of 2HAM simulation is the unification of two supertiles. In order to unite two supertiles, one of them has to be translated to ensure that the supertiles are disjoint. Therefore, in the implementation, the simulator action `TwoHamSimulator::ActionTuple` is defined as a triplet of two supertiles and translation. In each step of 2HAM simulation, there can be multiple of those actions which will be stored in set `next_actions` of type `TwoHamSimulator::Actions` (line 3).

In order to find all supertiles that can be united in the given step, every unordered pair of supertiles must be checked. Each pair of supertiles denoted as `configuration1`, and `configuration2` is constructed by two nested for-cycles (lines 4–8). The function then tries every translation of `configuration1` so that it is disjoint with `configuration2` and they share a common interface. Each translation is calculated as the difference of a perimeter position of `configuration1` and a frontier position of `configuration2`, where the set of frontier positions consist of the occupied positions on the outer edge of the configuration (lines 10–17). Next, `configuration2` is copied and translated using the calculated translation (lines 19–22). Then, the translated supertile is tested whether it is disjoint with `configuration1` and whether both supertiles generates enough bonding strength (lines 24–26). If the translated supertile can be united with `configuration1`, a simulator action is constructed, storing the found supertile (lines 27, 28).

In the rest of the cycle, the found unification is checked whether it was not already found in previous simulation steps, and if it was not, it is added to the `next_actions` set (lines 30–46). This procedure is then repeated for each frontier position of `configuration2`, for each perimeter position of `configuration1` and each unordered pair of supertiles in the current step of the simulation. Thus the function finds the set

```
TwoHamSimulator::Actions TwoHamSimulator::simulate_next_production()
{
    auto next_actions = TwoHamSimulator::Actions{};
    for( auto id1 = 0u; id1 < m_configurations.size(); id1++ ) {
        auto const & configuration1 = m_configurations.at( id1 ).tiles;
        for( auto id2 = id1; id2 < m_configurations.size(); id2++ ) {
            auto const & configuration2 =
                m_configurations.at( id2 ).tiles;

            for( auto const & perimeter_position :
                 configuration1.get_perimeter()
            ) {
                for( auto const & frontier_position :
                     configuration2.find_frontier_of_tiles()
                ) {
                    auto const translation =
                        perimeter_position - frontier_position;

                    auto translated_configuration =
                        TileConfiguration{ configuration2 };

                    translated_configuration.translate( translation );

                    if( configuration1.can_be_united_with(
                            translated_configuration )
                    ) {
                        auto const action_tuple =
                            std::make_tuple( id1, id2, translation );

                        auto action_already_exists = false;
                        for( auto const & actions_in_step :
                             m_production_history
                        ) {
                            for( auto const & history_action :
                                 actions_in_step
                            ) {
                                if( action_tuple == history_action ) {
                                    action_already_exists = true;
                                    break;
                                }
                            }
                            if( action_already_exists ) { break; }
                        }
                        if( !action_already_exists ) {
                            next_actions.emplace( action_tuple );
                        }
                    }
            }   }
    }   }
    return next_actions;
}
```

Listing D.5. Implementation of 2HAM simulator function which finds the set of all possible combinations of supertiles for the next simulation state.

of all possible unifications of supertiles and returns it (line 50). Note that the function has six nested for-cycles at one point, which makes this function quite computation demanding. One possible optimisation for the future implementation would be utilising a multi-thread programming and testing each pair of supertiles concurrently.

D.3 MuTAGEN Algorithm Implementations

In this section, two algorithms implemented in MuTAGEN application are further discussed in detail.

D.3.1 Partition-Search with Heuristics (PS-H)

In the following Listings D.6, D.7, it can be seen the implementation of the PS-H algorithm introduced in Section 2.3.2. Please note that the code is separated into two listings despite the code was stripped of comments and reformated in order to fit it on the page.

In the first part of the PS-H algorithm (Listing D.6), can be seen that the function takes two inputs, a tile partition `a_tiling` and a pattern `a_pattern`, and the function returns an optimised tile partition. This configuration allows for entirely or partly suboptimal tile pattern `a_tiling` to be optimised.

First, the inputted tiling is copied to the new instance `tiling`, which is safe to modify (line 5). The set of all unordered pairs of tiles in `tiling` where both elements have the same colour in `a_pattern` is found and stored as `same_color_class_pairs` (lines 6, 7). The set stores all possible pairs of partition classes which can be merged together such that the pattern is not damaged. The algorithm then repeats the following process until `same_color_class_pairs` set is depleted.

The set of the same coloured pairs is filtered (ordered) using the trio of heuristics denoted as `H1`, `H2`, and `H3` in the code. Because the limited space of the page and the implementation of each heuristic is very similar, only the `H1` heuristic is explicitly stated in the code. The first heuristic `H1` filters those pairs of tiles which share the maximum number of glues between themselves. The heuristic is implemented in the following fashion. First, an empty set of pairs `common_glues_pairs` is initialised together with a counter of the maximum number of glues shared between the tiles in the pair `max_num_of_common_glues` (lines 11–13). Then, it is obtained the number of common glues `num_of_common_glues` for each pair of tiles in `same_color_class_pairs` (lines 15–17). If `num_of_common_glues` is greater than the current maximum, the `common_glues_pairs` set is emptied, and the new maximum is set (lines 18–21). Now if the `num_of_common_glues` is equal to the `max_num_of_common_glues`, the pair is added to the `common_glues_pairs` set (lines 22–24).

The other two heuristics `H2` and `H3` indicated in lines 26 and 27, respectively, are implemented similarly as `H1` where each heuristic takes the set of pairs filtered by the previous heuristic, and it performs additional filtering. The `H2` heuristic prioritises pairs where the tile type with *greater* number of occurrences in the current tiling is maximal, and `H3` prioritises pairs where the tile type with *smaller* number of occurrences in the current tiling is also maximal. Formal definitions of each heuristic are described in Algorithm 2.4.

Next, step is to randomly select one pair from the remaining pairs which were filtered by the heuristics. Each pair of tiles have a uniform probability of selection (lines 29–40). The randomly selected pair is named `pair` (line 42). Next, the tile pattern `tiling` is copied together with the selected pair and `same_color_class_pairs` so it can be

```cpp
TilePartition TasSynthesizer::optimize_tiling(
    TilePartition const & a_tiling,
    PatternPartition const & a_pattern
) {
    auto tiling = TilePartition::clone( a_tiling );
    auto same_color_class_pairs =
        tiling.find_tile_class_pairs_of_same_color( a_pattern );

    while( !same_color_class_pairs.empty() ) {
        // H1: common_glues_pairs
        auto common_glues_pairs =
            std::unordered_set<TilePartition::PairOfTiles>{};
        auto max_num_of_common_glues = 0u;
        for( auto const & pair : same_color_class_pairs ) {
            auto const num_of_common_glues = number_of_common_glues(
                *pair.first, *pair.second
            );
            if( num_of_common_glues > max_num_of_common_glues ) {
                common_glues_pairs.clear();
                max_num_of_common_glues = num_of_common_glues;
            }
            if( num_of_common_glues >= max_num_of_common_glues ) {
                common_glues_pairs.insert( pair );
            }
        }
        // H2: maximum_point_count_of_larger_class_in_pairs
        // H3: maximum_point_count_of_smaller_class_in_pairs

        auto const selection_vector = std::vector<
          std::reference_wrapper<const TilePartition::PairOfTiles>>(
            std::cbegin( maximum_point_count_of_smaller_class_in_pairs ),
            std::cend( maximum_point_count_of_smaller_class_in_pairs )
        );

        std::random_device rd;
        auto generator = std::mt19937{ rd() };
        auto random_index_distribution =
            std::uniform_int_distribution<size_t>{
                0, selection_vector.size()-1 };
        auto const random_index = random_index_distribution( generator );

        auto const & pair = selection_vector.at( random_index ).get();

        auto copied_objects = TilePartition::clone_with_pair_and_set(
            tiling,
            pair,
            same_color_class_pairs
        );
```

Listing D.6. An implementation of the PS-H algorithm written using C++ in MuTAGEN application (1/2).

tested whether merging the tile types in `pair` yields better (more optimal) tiling without the risk of losing the current solution (lines `44–48`). The implementation continues in Listing D.7.

Copied intances of `tiling`, `pair` and `same_color_class_pairs` are retrieved and named `visiting_tiling`, `first_merging_pair` and `visiting_same_color_pairs`, respectively. The `visiting_tiling` is used to check whether the tiling is still constructable after merging the tile types of `first_merging_pair` (lines `49–51`).

After that, tile types in `merging_pair` are merged together in `visiting_tiling`, and then, it is tested whether the tiling is still constructible using the function `is_constructable()`. The function returns a tuple with a boolean value containing the result of the test and another pair of tile types to be merged, which prevents the tiling from being constructible. The merging is repeated until the `visiting_tiling` is either found to be constructible or every tile pair, which prevents the constructability of the tiling is merged (lines `56–64`).

If the `visiting_tiling` is not found constructible, it is abandoned, and the merging pair, which create it is removed from the `same_color_class_pairs` set (lines `90–92`). Otherwise, the `visiting_tiling` is tested whether it is a refinement of the pattern, and it not the main loop is broken because any subsequent merges cannot improve the current tiling (lines `67–71`). If the merged tiling is both constructible, and it is a refinement of the pattern, it is set as the new best-found solution (lines `73`, `74`).

Before the start of the next loop, the `same_color_class_pairs` set is recreated from the `visiting_same_color_pairs` by filtering out pairs of tile types where the tile type which was removed from the tiling in the merging process is not included (lines `76–88`).

After depleting the set of same coloured pairs, the tiling is additionally checked whether it contains duplicate tile types and these types are eventually merged (line `94`). The last step is a consolidation of used glue types which is done by the function `reassign_glue_ids()` (line `95`). The `reassign_glue_ids()` function can further decrease the number of the needed glues in the tiling, which is further described in the following text. In the end, the function returns the most optimal found tiling (line `96`).

D.3.2 Glue type reduction function

The last example from MuTAGEN implementation is the function which is used to reassign glue types of tiles in the `TilePartition` class. The function code is presented in Listing D.8.

After generating the tile partition using the PS-H algorithm described above, the individual tiles in the partition end up with sparsely distributed glue type identification numbers (IDs) due to the fact that merging of the tile types in PS-H algorithm takes place in no particular order. For example, there exists glue with type ID 3 and glue with type ID 6, but there are no glues with type IDs 1, 2, 4, 5, 7, and so forth in the tiling. This function ensures that the glue type IDs are reassigned beginning from the ID 1 to the number of unique glues in the system.

Because of the requirements placed on the tile assembly system which solves the partitioning problem, the self-assembly process sequence of the system is determined by the combination of south and west glues in the individual tile type (in the case of the origin of seed structure is placed in the south-west corner). Additionally, thanks to the way how the MGTA function assigns glue type IDs in the start of the system synthesis, the set of all glue type IDs which are located on the south side of the tiles $\sigma_S(T)$ is disjoint with the set of glues on the west side of the tiles $\sigma_W(T)$, where T

```cpp
        auto visiting_tiling           = std::get<0>( copied_objects );
        auto first_merging_pair        = std::get<1>( copied_objects );
        auto visiting_same_color_pairs = std::get<2>( copied_objects );

        auto tiling_is_constructable = false;
        auto merging_pair = first_merging_pair;

        do {
            visiting_tiling.merge_tiles( merging_pair );
            std::tie( tiling_is_constructable, merging_pair ) =
                visiting_tiling.is_constructable();
        } while(
            (!tiling_is_constructable)
            && (merging_pair.first != nullptr)
            && (merging_pair.second != nullptr)
        );

        if( tiling_is_constructable ) {
            auto const tiling_is_refinement_of_pattern =
                visiting_tiling.is_refinement_of( a_pattern );
            if( !tiling_is_refinement_of_pattern ) {
                break;
            }

            tiling = visiting_tiling;
            visiting_same_color_pairs.erase( first_merging_pair );

            same_color_class_pairs.clear();
            for( auto const & p : visiting_same_color_pairs ) {
                if( tiling.contains_class( p.first )
                    && tiling.contains_class( p.second )
                    && ( p.first != p.second )
                    && ( same_color_class_pairs.count( p ) < 1 )
                    && ( same_color_class_pairs.count(
                           TilePartition::PairOfTiles{p.second, p.first}
                         ) < 1
                       )
                ) {
                    same_color_class_pairs.insert( p );
                }
            }
        } else {
            same_color_class_pairs.erase( pair );
        }
    }
    tiling.merge_duplicate_tiles();
    tiling.reassign_glue_ids();
    return tiling;
}
```

Listing D.7. An implementation of the PS-H algorithm written using C++ in MuTAGEN application (2/2).

```cpp
void TilePartition::reassign_glue_ids()
{
    for( auto const & tile : get_classes() ) {
        for( auto && side : Tile::sides ) {
            auto & glue = tile->operator[](side);
            glue.set_id( -glue.get_id() );
        }
    }

    auto vertical_glue_id_counter = 1;
    auto horizontal_glue_id_counter = 1;
    auto vertical_glue_id_map = std::unordered_map<Glue, Glue>{};
    auto horizontal_glue_id_map = std::unordered_map<Glue, Glue>{};

    for( auto const & tile : get_classes() ) {
        auto const & north_glue = tile->get_north();
        if( vertical_glue_id_map.count( north_glue ) < 1 ) {
            vertical_glue_id_map[ north_glue ] =
                Glue{ vertical_glue_id_counter++, 1u };
        }
        auto const & south_glue = tile->get_south();
        if( vertical_glue_id_map.count( south_glue ) < 1 ) {
            vertical_glue_id_map[ south_glue ] =
                Glue{ vertical_glue_id_counter++, 1u };
        }
        auto const & east_glue = tile->get_east();
        if( horizontal_glue_id_map.count( east_glue ) < 1 ) {
            horizontal_glue_id_map[ east_glue ] =
                Glue{ horizontal_glue_id_counter++, 1u };
        }
        auto const & west_glue = tile->get_west();
        if( horizontal_glue_id_map.count( west_glue ) < 1 ) {
            horizontal_glue_id_map[ west_glue ] =
                Glue{ horizontal_glue_id_counter++, 1u };
        }
    }

    for( auto const & p : vertical_glue_id_map ) {
        replace_glue( p.first, p.second );
    }
    for( auto const & p : horizontal_glue_id_map ) {
        replace_glue( p.first, p.second );
    }
}
```

Listing D.8. Implementation of function which consolidates the glue types within the set of tiles and it can further reduce the number of glues in the solution system after synthesising the solution using the PS-H algorithm.

denotes the set of all tiles in the system. This relation can be written as

$$\sigma_S(T) \cap \sigma_W(T) = \emptyset.$$

Finally, because the pair of south and west glues $\sigma_S(t)$ and $\sigma_W(t)$, respectively, which determines the behaviour of the tile t in the system is *ordered*, the glue IDs used in the south glues can be reused in the west glues. Therefore, the number of needed glues in the system can be reduced to half.

The function is implemented using the code in Listing D.8. It is easy to see that the function does not take any input, and it is applied to the instance of the `TilePartition` class. First, all glues are assigned with negative IDs, which prevents possible ID collisions during the process of reassigning the IDs (lines 3–8). Then, ID counters and glue ID maps are initialised for glues in the horizontal direction (east, west) and for glues in the vertical direction (north, south) separately (lines 10–13). Next, for each tile type in the system, the following process is repeated. Glue ID of each tile side is tested whether it is already in the corresponding map. If the glue ID has been mapped, the algorithm continues to the next glue or tile. Otherwise, the glue is assigned the next ID from the corresponding counter and the new mapping is added to the corresponding map (lines 16–35). In the end, the new glue IDs is applied to the tiles in the system using the maps (lines 38–43).

Appendix E
User Manuals

E.1 MuTATOR User Manual

The Meta-Material Tile self-Assembly sysTem simulatOR

E.1.1 Dependencies

- CMake 3.10 or newer (`https://cmake.org/`)
- GTKmm 3.0 or newer (libgtkmm-3.0-dev) (`https://www.gtkmm.org/`)
- Cairomm 1.0 or newer (`https://www.cairographics.org/`) (should be included in GTKmm)
- spdlog (`https://github.com/gabime/spdlog`) (included in `./external/`)
- Doxygen (`http://www.doxygen.org/`) with graphviz module (`https://www.graphviz.org/`) (optional, for building documentation)

MuTATOR was tested and developed with GCC 7.3.0, GTKmm 3.12.0, Cairomm 1.12.0, Cairo 1.14.6 and Doxygen 1.8.11.

E.1.2 How to compile

For compiling MuTATOR project, it is needed to use standard CMake compilation procedure from the project root directory.

Type the following commands from the project root directory:

```
mkdir build && cd build
cmake ..
make
```

To clean the generated files, type the following commands:

```
cd build
make clean
cd ..
rm -r build
rm -r bin
```

E.1.3 How to run

Application executable is located in `./bin/` directory. Type the following line to run the application:

```
cd ./bin
./mutator [options] <arg-tas>
```

Description of program arguments:

Argument	Position	Description
`<arg-tas>`	1st	Tile Assembly System (TAS) JSON string of the system to be simulated. More about TAS JSON representation can be found in Section E.1.4.

Description of program options:

Option	Description
`-h`, `-help`, `--help`, `--usage`	Prints the program usage instructions and exits.
`-v`, `--version`	Prints the program version information and exits.
`--verbose`, `--debug`	Prints out debug messages and more verbose messages about program status.
`--tas=<path_string>`, `--tas-path=<path_string>`, `--input=<path_string>`	Specifies path to the Tile Assembly System (TAS) JSON file that will be loaded after program start. More about TAS JSON representation can be found in Section E.1.4
`--`, `--stdin`	Specifies that the TAS JSON should be read from the standard input. Useful for piping output from MuTAGEN directly to MuTATOR. More about TAS JSON representation can be found in Section E.1.4.
`--module-meshes-dir=<path_string>`, `--module-meshes-path=<path_string>`	Specifies path to directory with alternative definitions of tile module meshes. The directory must contain files to all modules that are files named `tileXX.txt` where `XX` is `01` to `16`. If the path is not valid or the directory does not contain all module mesh definitions, the default meshes will be used. More about Module Mesh Definition files can be found in the Section E.1.5

Examples of usage:

Run MuTATOR without any options:

```
./mutator
```

Specifying a path to a TAS file to be opened at the startup:

```
./mutator --tas="../tas/binary_counter.json"
```

Directly input a TAS JSON as a string argument to the program:

```
./mutator "{ ...<tas_json>... }"
```

Specifying to load TAS JSON from standard input or pipe:

```
./mutator --
```

Specifying to load TAS JSON directly from MuTAGEN using a pipe:

```
./mutagen ../patterns/pattern.png | ./mutator --
```

Specifying directory with alternative module meshes definitions:

```
./mutator --module-meshes-dir="../alternative_meshes/"
```

Specifying to display more verbose messages and debug messages:

```
./mutator --verbose
```

Showing program usage instructions:

```
./mutator --help
```

E.1.4 Tile Assembly System: JSON Representation

The JSON file structure is described using the following definitions.

Basic Definitions

Tile Assembly System (*TAS*) is represented by two required elements – *tiles* and *temperature* and two optional elements – *seed* and *simulator*.

Tiles are simply a set of all tiles which can occur in the system. A single tile is defined by four *glues* on each side of the tile (in the 2D case). Glue is defined by its strength and identifier.

Temperature is an integer constant greater than or equal to 1 and defines the minimum strength of tile *glues* which is needed for the formation of tile bond.

Seed structure is only used in aTAM simulation, and the seed structure must be specified in order to simulate an aTAM model. The seed represents the initial tile structure from which the system grows. Seed can be single tile or array of multiple tiles with their positions. Simulator element is used to specify preferred simulator for the given file.

TAS JSON structure

Root elements of the file can be seen in the following commented pseudo-JSON code.

```
{
    "temperature": <positive_integer>,
    "tiles": [...],
    "seed": [...] | {...},              // optional
    "simulator": <id_string>            // optional
}
```

Temperature element

Temperature `temperature` is an unsigned integer greater or equal to `1`. The temperature must always be specified.

Simulator element

String `simulator` is optional and can be used to specify the preferred simulator to simulate the given TAS model.

The recognised (case insensitive) strings are the following:

- `aTAM`: the aTAM simulation model
- `2HAM`: the 2HAM simulation model

Tile element

Tile object is specified by the following elements:

```
{
    "north": {
        "id": 1,
        "strength: 2,
    },
    "east": {
        "id": 2,
        "strength: 2,
    },
    "south": {
        "id": 5,
        "strength: 2,
    },
    "west": {
        "id": 4,
        "strength: 1,
    },
    "additional_info": {...}
}
```

Each glue element **north**, **east**, **south** and **west** is optional and can be skipped. The skipped glue is represented by the *null glue*, which means that the tile cannot bond on this side.

Each specified glue object must have both its elements **id** and **strength**, which are both integers.

Last element **additional_info** is also optional and carry information about tile visualization and user information.

The **additional_info** can contain the following elements:

- `label`: String that will be shown on tile (string, default:)
- `label-size`: Size of the label (float, default: 8.0)
- `label-color`: Color of the label (X11 color string, default: `black`)
- `line-width`: Width of the tile border line (float, default: `1.0`)
- `line-color`: Color of the tile border line (X11 color string, default: `black`)
- `fill-color`: Color of the tile body (X11 color string, default: `white`)

Seed element

Seed element is optional in general but please note, that it is required for the aTAM simulation. The *seed* can be a single object representing a single seed tile, for example

```
{
    ...
    "seed": <tile_element>,
    ...
}
```

or it can be array of tiles with positions, as in the following example

```
{
    ...
    "seed": [
            {
                "position": {
                    "x": 1,
                    "y": 0
                },
                "tile": <tile_element>
            }, {
                "position": {
                    "x": 1,
                    "y": 1
                },
                "tile": <tile_element>
            }
        ],
        ...
    }
```

For structure of the Tile object (`<tile_element>`), see the section above.

Object `position` has two number elements `x` and `y`, which specifying the position of the seed tile `tile` in the seed structure. This object must specify both its elements.

Tiles element

The element `tiles` consists from array of Tile objects (`<tile_element>`), see the section above.

```
    {
        ...
        "tiles": [
             <tile_element>,
             ...
             <tile_element>
        ],
        ...
    }
```

E.1.5 Module Mesh Definition File Structure

The text file that defines a module mesh must have the following format:

```
    <number of vertices N>
    <number of triangle elements M>
                        // empty line
    <x> <y> <z>         // vertex ID: 1
    <x> <y> <z>         // vertex ID: 2
    ...                 // other vertecies
    <x> <y> <z>         // vertex ID: N
                        // empty line
    <ID> <ID> <ID>      // 1st element
    ...                 // other elements
    <ID> <ID> <ID>      // M-th element
                        // end of file
```

Where the `<number of vertices N>` is a positive non-zero integer number that defines how many mesh nodes will be specified, `<number of triangle elements M>` is positive non-zero integer number that defines how many triangle elements will create module mesh.

Each vertex is defined as 3D point using floating-point numbers `<x>` `<y>` `<z>` but only `<x>` and `<y>` are actually used for drawing the module mesh. The line on which the vertex definition lies represents `ID` of the vertex. Vertex `ID` are numbered from `1`.

Each element is defined on one line, and it defined using at least three vertex `IDs` which will be drawn as a triangle in module mesh. The vertex `IDs` are ignored from the fourth `ID` onwards in the element definition; it means that only the first three `IDs` are used in drawing.

Example of Module Mesh Definition file:

NOTE: The following example was shortened using ”. . . ”.

```
1400
618

-1.400000e-01 -5.000000e-01 0.000000e+00
-9.000000e-02 -5.000000e-01 0.000000e+00
9.000000e-02 -5.000000e-01 0.000000e+00
...
-2.481333e-01 -2.736187e-01 0.000000e+00
-2.577768e-01 -2.474160e-01 0.000000e+00
-2.371039e-01 2.841897e-01 0.000000e+00

5 21 165 396 557 556
21 6 166 397 559 558
7 22 167 398 561 560
...
385 386 387 1389 1391 1400
387 388 385 1392 1393 1400
389 390 391 1394 1399 1397
```

E.2 MuTAGEN User Manual

The Meta-Material Tile self-Assembly system GENerator

E.2.1 Dependencies

- CMake 3.10 or newer (`https://cmake.org/`)
- GTKmm 3.0 or newer (libgtkmm-3.0-dev) (`https://www.gtkmm.org/`)
- Cairomm 1.0 or newer (`https://www.cairographics.org/`) (should be included in GTKmm)
- spdlog (`https://github.com/gabime/spdlog`) (included in `./external/`)
- Doxygen (`http://www.doxygen.org/`) with graphviz module (`https://www.graphviz.org/`) (optional, for building documentation)

MuTAGEN was tested and developed with GCC 7.3.0, GTKmm 3.12.0, Cairomm 1.12.0, Cairo 1.14.6 and Doxygen 1.8.11.

E.2.2 How to compile

For compiling the whole project, you can use the standard CMake compilation procedure. Type the following commands from the project root directory:

```
mkdir build && cd build
cmake ..
make
```

For cleaning the generated files, type the following commands:

```
cd build
make clean
cd ..
rm -r build
rm -r bin
```

E.2.3 How to run

Application executable is located in `./bin/` directory. Type the following line to run the application:

```
cd ./bin
./mutagen [options] <arg-input> <arg-output>
```

Description of program arguments:

Argument	Position	Description
`<arg-input>`	1st	Path to an image file or text file that specifies the required pattern of tiles. Also, it can be specified using options `--input` or `--pattern`. Lossless image formats are preferred (e.g. BMP, PNG). All supported input file formats are listed below, or they can be shown using `./mutator --help-formats` option or below in Section E.2.4.
`<arg-output>`	2nd	Path to output file or directory, where the resulting TAS will be stored. Also, it can be specified using options `--output` or `--tas`. If not specified, left empty or "`--`" the resulting TAS will be printed to standard output.

Description of program options:

Option	Description
`--configuration=<path_string>`, `--config=<path_string>`	Specifies a path to configuration file which can be used to specify other program options and flags. Option values that are specified by the user are preferred over option values specified by the configuration file. More about configuration file format can be found in Section E.2.5.
`-h`, `-help`, `--help`, `--usage`	Prints the program usage instructions and exits.
`--help-formats`, `--list-formats`	Prints all supported image and text formats of pattern input file.
`-v`, `--version`	Prints the program version information and exits.
`--verbose`, `--debug`	Prints out more verbose messages about program status. *WARNING: Cannot be used while piping output TAS to another program, for example, MuTATOR!*
`--print-tas`, `--print-output`	Prints out the output TAS even if the output file path is specified.
`--input=<path_string>`, `--pattern=<path_string>`	Path to a image file or text file that specifies the required pattern of tiles. Lossless image formats are preferred (e.g. BMP, PNG). All supported input file formats can be shown using `--help-formats` option or below in Section E.2.4.
`--output=<path_string>`, `--tas=<path_string>`	Path to output file, where the resulting TAS will be stored. If not specified, left empty or "--" the resulting TAS will be outputted to standard output. Default: `--`
`--cycles=<positive_integer>`	Number of optimization cycles. Default: 1
`--threads=<positive_integer>`	Number of used concurrent threads in optimization. The default value corresponds to the number of processor cores in the system.
`--timeout-min=<positive_integer>`, `--minutes=<positive_integer>`	Optimization timeout time in minutes. After the timeout elapses, no more optimization tasks are spawned. NOTE: The value of this option is combined with timeout in seconds option `--timeout-sec`. Default: 0 minutes
`--timeout-sec=<positive_integer>`, `--seconds=<positive_integer>`	Optimization timeout time in seconds. After the timeout elapses, no more optimization tasks are spawned. NOTE: The value of this option is combined with timeout in minutes option `--timeout-min`. Default: 0 seconds

Option	Description
`--seed-fixed`, `--fixed`	Specifies that seed structure will be generated as fixed, that means it cannot be self-assembled. This is the default behaviour.
`--seed-assembling`, `--assembling`	Specifies that seed structure will be generated as self-assembling and the whole assembly will start from single seed tile.
`--seed-left`, `--left`	Specifies that one of the seed borders should lie on the left of the pattern. Default seed position is left and bottom.
`--seed-right`, `--right`	Specifies that one of the seed borders should lie on the right of the pattern. Default seed position is left and bottom.
`--seed-top`, `--top`	Specifies that one of the seed borders should lie on the top of the pattern. Default seed position is left and bottom.
`--seed-bottom`, `--bottom`	Specifies that one of the seed borders should lie on the bottom of the pattern. Default seed position is left and bottom.
`--filter-ids=<integer_vector>`, `--ignore-ids=<integer_vector>`	List of integer labels of tiles that will be excluded from tile set when generating TAS. The list must consist of only integer labels and must be enclosed within square brackets `[]`, and values in the list must be separated by a comma. Make sure that the whole list is enclosed with quotation marks when using spaces inside the list. Example: `--filter-ids=̈[1, 2, 3]̈` or `--filter-ids=[1,2,3]`. Default: `[]`
`--enable-meshes`, `--show-meshes`	Specifies that tiles should be rendered with module meshes in MuTATOR.
`--mesh-mode`	Does the same thing as specifying the following options `--enable-meshes --ignore-ids=[0]`. WARNING: Be careful when using this option, ignoring tile with label `0` may result in unconstructable TAS.

E.2.4 Supported input formats:

The first program argument or option `--input`, `--pattern` must be specified as the path to file that describes the desired pattern. This input file must have one of the formats in the following lists and its standard extensions.

Supported image formats:

Label	Name
GdkPixdata	GdkPixdata
icns	MacOS X icon
gif	GIF
xpm	XPM
xbm	XBM
png	PNG
pnm	PNM/PBM/PGM/PPM
ani	Windows animated cursor
tga	Targa
ico	Windows icon
qtif	QuickTime
bmp	BMP
jpeg	JPEG
wmf	Windows Metafile
tiff	TIFF
svg	Scalable Vector Graphics

Supported text formats:

Text file with extensions `*.txt`, `*.pattern`, or `*.mutagen`, where positive integer values separated by tabulator in row, represents partitioning of the pattern and new-line starts next row. Text pattern is rotated 90 degrees counter clock-wise in the final assembly.

Examples:

Specifying pattern image and output TAS to standard output (implicitly) using options:

```
./mutagen --input="../patterns/image.png"
```

Specifying pattern image and output TAS to standard output (explicitly) using options. Note that options can have arbitrary order, but the program arguments must be in a defined order:

```
./mutagen --output="--" --input="../patterns/image.png"
```

Specifying pattern text file and saving output TAS to file using options with verbose output:

```
./mutagen --verbose --pattern="../patterns/file01.txt"
--tas="tas01.json"
```

Specifying pattern image file and saving output TAS to file using program arguments and also preview the output TAS in the console:

```
./mutagen --print-output ../patterns/pattern.bmp pattern_output.json
```

Specifying pattern text file and saving best output TAS from 10 optimisation cycles:

```
./mutagen --cycles=10 ../patterns/pattern.txt ./pattern_tas.json
```

Showing program usage instructions:

```
./mutagen --help
```

Showing all supported input pattern file formats.:

```
./mutagen --help-formats
```

E.2.5 Description of the Configuration file

Program options can be provided by the configuration text file with the following rules:

- Line Comments are initiated by `#`
- Empty lines are ignored
- One Option specification per line
- Option specification consists of Identifier and Value and they are separated by `=`
- Identifier is the same as program option described above in Section E.2.3
- Whitespace before Identifier, whitespace between Identifier and `=` and between `=` and Value and whitespace after Value is ignored
- Option identifier can omit the initial `-` or `--` from its name
- Mentioned Flag is assumed as `true` if it is not specified otherwise, to disable flag simply do not list it or set it as `false` if you want to be explicit
- Value itself must not contain whitespace, otherwise it must be enclosed with ”

Example of the Configuration file:

```
#
# MuTAGEN Example Configuration
#

# Timeout options
cycles      = 24
timeout-min = 1

# Seed position and type
seed-bottom            # flags are automatically assumed as ’true’
seed-left
seed-fixed = true      # explicitly enabled flag

# Show Mesh on tiles is disabled
enable-meshes = false  # explicitly disabled flag

# Do not ignore tiles
ignore-ids = []

# Verbose output is disabled
verbose = false
```