A CONTINUOUS PATH PLANNING AND UPDATING ALGORITHM BASED ON VORONOI DIAGRAMS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

MELIH ÖZCAN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
MECHANICAL ENGINEERING

AUGUST 2020

Approval of the thesis:

A CONTINUOUS PATH PLANNING AND UPDATING ALGORITHM BASED ON VORONOI DIAGRAMS

submitted by **MELIH ÖZCAN** in partial fulfillment of the requirements for the degree of **Master of Science in Mechanical Engineering Department, Middle East Technical University** by,

Prof. Dr. Halil Kalıpçılar
Dean, Graduate School of **Natural and Applied Sciences** ____________

Prof. Dr. M. A. Sahir Arıkan
Head of Department, **Mechanical Engineering** ____________

Assoc. Prof. Dr. Ulaş Yaman
Supervisor, **Mechanical Engineering, METU** ____________

Examining Committee Members:

Prof. Dr. Bahattin Koç
Industrial Engineering, Sabancı University ____________

Assoc. Prof. Dr. Ulaş Yaman
Mechanical Engineering, METU ____________

Assoc. Prof. Dr. Yusuf Sahillioğlu
Computer Engineering, METU ____________

Assoc. Prof. Dr. Ender Yıldırım
Mechanical Engineering, METU ____________

Assist. Prof. Dr. Ali Emre Turgut
Mechanical Engineering, METU ____________

Date:

ABSTRACT

A CONTINUOUS PATH PLANNING AND UPDATING ALGORITHM BASED ON VORONOI DIAGRAMS

Özcan, Melih
M.S., Department of Mechanical Engineering
Supervisor: Assoc. Prof. Dr. Ulaş Yaman

August 2020, 97 pages

Coverage of an area is required for a large variety of robotics and manufacturing applications, such as environment monitoring, home cleaning, search and rescue operations, machining, delivery, additive manufacturing and even for 3D terrain reconstruction. In this work, we present a highly flexible algorithm that can be used for coverage and graph traversal. In addition to being applicable to diverse types of engineering problems, proposed method is advantageous to other algorithms, as it never turns around and traverses the edge it recently traversed. Although the method takes advantage of variable-sized Voronoi cells, by which regular, irregular and complex geometries can be easily composed, it is not limited to Voronoi diagrams and can be applied for any connected graph. Furthermore, path planning algorithm can *update* the path to deal with changes in the graph. In some applications, like 3D printing, path planning must be done for many instances. However, our algorithm calculates the path at the first layer, and performs only necessary changes at the subsequent layers, instead of calculating the whole path from scratch. This update mechanism makes the method very efficient as it is demonstrated with several test cases. In addition to the path planning algorithm, a G-code file encryption method is introduced, size of G-code files can be greatly reduced. As automation and robotics integrate into numerous areas everyday, proposed methods can be useful for many applications.

Keywords: robotics, automation, additive manufacturing, 3D printing, Voronoi, Euler path, G-code

ÖZ

VORONOİ DİYAGRAMINA DAYALI BİR YÖRÜNGE PLANLAMA VE GÜNCELLEME ALGORİTMASI

Özcan, Melih
Yüksek Lisans, Makina Mühendisliği Bölümü
Tez Yöneticisi: Doç. Dr. Ulaş Yaman

Ağustos 2020 , 97 sayfa

Alan tarama işlemi robotik ve üretim alanlarında birçok uygulama için gereklidir. Çevreyi gözetlemek, ev temizliği, arama ve kurtarma operasyonları, parça işleme, teslim ve 3B arazi rekonstrüksiyonu buna örnek olarak verilebilir. Bu çalışmada, alan ve grafik tarama uygulamarında kullanılabilecek ve son derece esnek bir algoritma sunulmuştur. Farklı alanlarda birçok probleme uygulanabilir olmanın yanı sıra, algoritmanın diğer yaklaşımlardan üstün özellikleri bulunmaktadır. Örneğin, taramakta olduğu kenarı tekrar taramak yerine başka bir kenara devam etmektedir. Geliştirilen yöntem, düzenli ve düzensiz kafes yapılarını modellemek için rahatlıkla kullanılabilecek Voronoi hücrelerinden faydalanmaktadır. Öte yandan, yöntem yalnızca Voronoi grafikleriyle kısıtlı olmayıp herhangi bir bağlı grafiğe de uygulanabilir. Algoritma, yörünge planlama dışında, *yörünge güncelleme* özelliğine de sahiptir. 3B üretim gibi, onlarca katman boyunca yörünge planlamanın gerekli olacağı bir senaryoda, algoritma ilk katman için bir yörünge planı oluşturup devam eden katmanlar için bu yörüngeyi güncellemektedir, her katmanda yörüngeyi baştan planlamamaktadır. Testlerle de gösterildiği üzere, bu planlama mekanizması algoritmayı son derece verimli yapmaktadır. Yörünge planlama dışında, G-kodu dosyalarıyla aynı veriyi çok daha az bir depolama alanıyla saklayan bir yöntem sunulmuştur. Otomasyon ve robotik teknolojilerinin sayısız alanda kullanılmasıyla birlikte sunulan yöntemler birçok uygulama için kullanışlı olacaktır.

Anahtar Kelimeler: robotik, otomasyon, eklemeli imalat, 3B üretim, Voronoi, Euler yörüngesi, G-kodu

To anybody who will benefit from the information presented here

ACKNOWLEDGMENTS

No single person can change the world. Science progresses by accumulation of people's contributions, and this study is no exception.

Emerging after the invention of computers, Computational Geometry has been rapidly developed. Similarly, although it is getting more traction, Additive Manufacturing has a long way to reach its true potential. We would like to thank every person who has worked and been working to make our lives better.

Hope is the key ingredient for not giving up, during the difficult times without any visible light at the end of the tunnel, and hope comes from inspiration. I would like to acknowledge Carl Hierholzer and Mark de Berg for their great works. Finally, I hope this study brings inspiration to people, as they and many more have given me.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

ABBREVIATIONS	
2D	2 Dimensional
2.5D	2.5 Dimensional
3D	3 Dimensional
CW	Clockwise
CCW	Counterclockwise
DOF	Degree of Freedom
FDM	Fused Deposition Modeling
FFF	Fused Filament Fabrication
STL	Standard Triangle Language
BRep	Boundary Representation

CHAPTER 1

INTRODUCTION

1.1 Motivation and Problem Definition

Robotics with automation is the next big thing for the civilization. Transferring the mundane, dangerous and repetitive jobs to the robots will give humans more prosperous lives. Additionally, the jobs will be performed by a higher quality. In order to contribute to this milestone as a species, methods are developed that can be utilized in two separate, but related domains as area coverage and additive manufacturing.

There are numerous coverage applications. These are listed as below.

- Surveying drones watch their respective territories.
- Vacuum cleaners clean the rooms of an apartment.
- Lawn mowers mow the grass inside the garden.
- Autonomous farming robots seed plants, water the trees, and harvest the crops.
- CNC machines remove a layer of material, by covering the specified area.

Similarly, 3D printers print a material, again, by covering the specified area. A coverage algorithm for such applications is proposed. In some cases, coverage of a recently covered segment might not be possible, or desirable. For instance, if a farming robot is to be used, driving it forward all the time may be required. Similarly, if a drone is used for monitoring, instead of directing it to a recently monitored segment, it would be a better choice to direct it to another segment. This problem is addressed by the proposed algorithm.

Additive Manufacturing (AM) is being widely adopted by the industry and the consumers. Many people and facilities have 3D printers, which are especially useful for rapid prototyping purposes. One drawback of the AM processes is the surface quality of the finished product. Usually, a finishing operation is applied to obtain a smoother surface. However, it is believed that this issue can be mitigated by *continuous printing*. In continuous printing, the printer covers all the layer while extruding the filament. Since there are no breaks between the extrusions, the surface quality gets better and the thermal stress effects become less problematic. By the proposed algorithm, this problem is addressed, and all layer is printed continuously.

While printing an artifact, path planning must be done for all the layers. However, this is a costly operation. If a path is already available, updating it for the next layers would be a better approach. That's another problem addressed by the proposed algorithm.

Lastly, an improvement about the structure of the *G-code*, which is an industry standard, is desired to be made. The same information can be stored in a much smaller space, by which a significant storage capacity can be freed.

1.2 Proposed Methods and Models

Before the coverage to begin, the area must be discritized, for which Voronoi graphs are used. Numereous 2D and 3D cell structures can easily be constructed using them. Due to their flexibility, Voronoi graphs are utilized in various studies.

To achieve continuous printing, Eulerian graphs and cycles are constructed. An Eulerian cycle starts a path, covers all edges, and then ends at the starting position. Graphs, which have Eulerian cycles are called Eulerian graphs. Therefore, it is needed to convert a Voronoi graph to an Eulerian graph, by duplicating its edges. Actually, any connected graph can be converted into an Eulerian graph, by this approach.

1.3 Contributions and Novelties

The contributions are as follows:

- A method to convert any connected graph into an Eulerian graph is proposed.
- Given an Eulerian graph, a continuous path planning method that eliminates covering a recently traversed edge again is introduced.
- A path updating method is offered, for many instances of path planning cases, like 3D printing.
- An encryption method for G-code files is developed, resulting in a significant decrease in the storage.

1.4 The Outline of the Thesis

In Chapter 2, background information is given and relative subjects for the study are explained, and the studies that are comparable to presented work are reviewed. In Chapter 3, proposed algorithms are explained, where examples are given to make the understanding easier, and pseudocodes are also provided. In Chapter 4, implementation details are given, then the algorithms used for comparison are introduced. Afterwards, test cases and their results are covered. In Chapter 5, the conclusion for the study is drawn.

CHAPTER 2

BACKGROUND INFORMATION & LITERATURE REVIEW

In this chapter, the necessary background information is given, and critical concepts are explained. Furthermore, related work to the presented study is discussed.

2.1 Coverage Algorithms

Given an area, coverage is the application of spanning the full domain. In many real life applications, coverage is used. For instance, environment monitoring by drones, house cleaning by robotic vacuum cleaners, machining by CNC machines, farming by autonomous vehicles, etc. are all done with the help of coverage algorithms. Similarly, while 3D printing an artifact, depending on the intensity setting, a grid structure is constructed. Then, the printer head *covers* the whole layer by following the lines of the corresponding grid. Various algorithms are developed according to the application-specific requirements. The number of agents and their movement constraints (e.g. whether they're holonomic or not) are also considered.

2.2 Eulerian Cycles and Eulerian Graphs

Eulerian cycle is a path for a connected graph that starts and ends at the same vertex. It traverses every edges of the graph exactly once [2]. A graph that has an Eulerian cycle is called as an Eulerian graph. In order for an undirected graph to have an Eulerian cycle, two conditions must be satisfied:

- The graph must be connected.

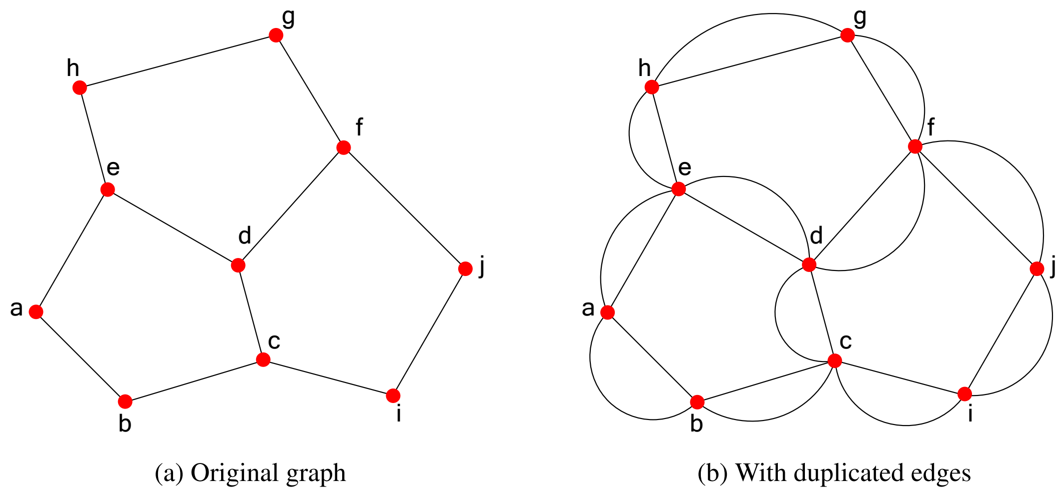


Figure 2.1: Obtaining an Eulerian cycle from a general graph.

- Every vertex must have an even number of edges.

Additionally, if the graph is directed, the number of incoming and outgoing connections must be equal for each vertex. A general graph example is given in Figure 2.1a. As seen from the figure, not all the vertices have an even number of edges. Consequently, an Eulerian cycle cannot be generated for this graph. Following the methodology of Yaman et al. [1], each edge is duplicated (Figure 2.1b). Therefore, the graph becomes Eulerian. Taking advantage of the Eulerian graph, one can start extruding the material at a vertex and end at the same place in Fused Filament Fabrication (FFF) process, eliminating fast travels.

2.3 Fast Travels and U Turns

While covering the edges of a graph, if there is no connection left at the current node although the graph is not fully covered, one needs to move into another node to continue with the coverage (Figure 2.2). That movement is called as a *fast travel*, originating from rapid positioning motion (G00) of CNC machines. On the other hand, if there are two connections between two nodes and if both of those covered one after the other, that motion is called as a *u* turn (Figure 2.2). Note that, *u turns* require a full stop. Thus, if possible, they must be eliminated for a more efficient operation.

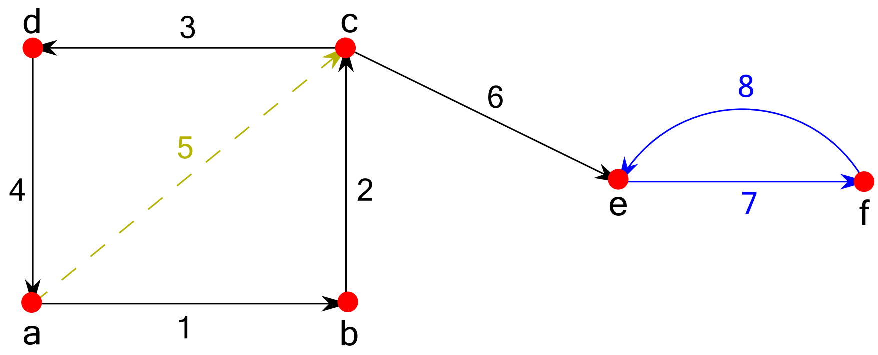


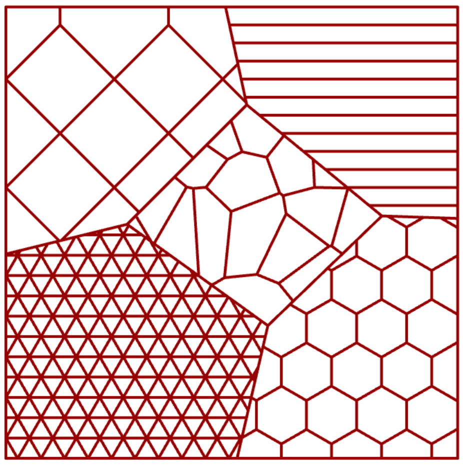
Figure 2.2: The concepts of *fast travel* (Step 5) and *u turn* (Steps 7 & 8).

2.4 Voronoi Graphs

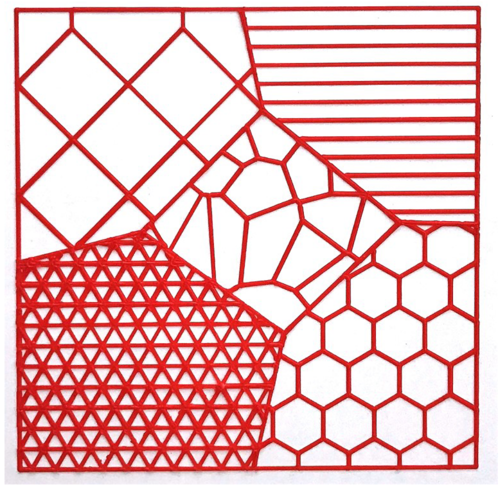
Given a list of sites, Voronoi diagrams decompose an area or a volume into smaller pieces such that each resulting Voronoi cell consists of the points that are closest to the given Voronoi site [3]. By choosing uniformly or non-uniformly generated sets of sites, diagrams with distinct patterns can be created, which makes Voronoi diagrams a useful tool for coverage. For example, rectangular, triangular, and hexagonal grid structures can be easily obtained. Using Voronoi diagrams, not only fully uniform or random structures, but also a combination of them can be produced. This is demonstrated by forming different Voronoi patterns into a square (Figure 2.3).

2.5 Literature Review

Coverage algorithms are used in many applications, such as vacuum cleaning [4], painting [5], mapping [6], demining [7], lawn mowing [8], harvesting [9], window cleaning [10] and structure inspecting [11]. They can be classified according to various criteria. The ones that guarantee the complete coverage are called *complete*, whereas the others are called as *h*euristic [12]. If the area to be covered is required to be known in advance, they are called *o*ffline; and if that is not the case, they are called textit online, or *s*ensor based coverage algorithms [13]. Most algorithms are developed for the planar space, whereas some of them consider the 3D space since



(a) A Voronoi diagram with different Voronoi infill patterns



(b) 3D printed version

Figure 2.3: A heterogeneous Voronoi diagram, consisting of 5 Voronoi diagrams.

the applications dictate that. The majority of the algorithms divide the space into subregions, such as the trapezoidal decomposition [14, 15] or morse-based cell decomposition method [16]. On the other hand, grid-based algorithms represent the area as a collection of uniform grid cells, whose values can be a binary or a probability [17, 18]. Some algorithms consider multiple agents, where some of the agents are assigned to *exploration* and the remaining ones are responsible for *exploitation*, i.e., coverage [19]. Overall, many algorithms are developed and used for numerous applications.

One of the most functional and popular technologies of the present age is unmanned aerial vehicles. There are interesting coverage algorithms developed by considering their needs [20]. For instance, while covering an area, it would be advantageous to have a short path. On the other hand, the energy requirement of UAVs is much critical. Changing the direction frequently is seemed to be energy wasting. For that reason, energy-aware algorithms are developed [21, 22]. In those methods, instead of lots of sharp turns, like seen in Hilbert curves, more straight paths are preferred. Another class of algorithms aim multi-robot coverage. For instance, in decentralized algorithms, drones exchange information only with the other drones that are close to them, and every drone covers their respected area [23, 24]. Also, there are methods inspired by the nature. For instance, mimicking the ants' communication, pheromone-based algorithms are introduced, in which the area is divided into grids, and the number of visits for each grids are stored [25, 26]. As one can see, different requirements lead to various methods.

In Fused Deposition Modeling (FDM) processes, the quality of the part is determined by two critical factors that control the final shape of the product, namely *slicing* and *path planning* [27]. Some slicing approaches require a device with higher capabilities. For instance, instead of a conventional cartesian 3D printer, a robotic arm with 3 DoF or another one with 5 DoF might be necessary. After slicing is done, path planning takes place, which determines the precision of the product. Therefore, in order to have a more precise part, the boundary of the part is printed following contour-parallel lines with a relatively slower feedrate. On the other hand, the inside of the part is filled following a zigzag pattern, which allows faster printing [28]. Although this is the most common practice for the available 3D printers, it does not consider the continuous deposition issue, which is a widely studied concept in AM. To obtain a uniform material density and mechanical properties, researchers developed methods that utilize various infill patterns and path planning algorithms. Jin et al. [29] first decomposed the area into what they call simple areas that can be traversed without retraction, using direction-parallel and contour-parallel polygon decomposition methods. Then they calculated a continuous subpath for each segment, which starts and ends at the same point. In the end, they connected all the subpaths to form a single and continuous path for the whole area. Zhao et al. [30] utilized Fermat spirals. Unlike the space filling curves, such as Hilbert curves which have lots of corners and sharp turns, Fermat spirals are constructed by low curvature paths. They first decomposed the given area into subregions that can be continuously filled with a Fermat spiral. By doing so, they ensured that the start and end points of the spirals are very close to each other. After forming the paths for all subregions, they connected the paths and obtained a single continuous path. Nevertheless, as they mentioned, their method is not guaranteed to cover any arbitrary region. Shaikh at al. [31] used Hilbert curves for 3D printing. They took an area and then filled it with a Hilbert curve. Then, if there are any contours in the area, they removed the corresponding regions from the curve and printed the final shape. Nevertheless, their final product is not homogeneous in terms of infill material. Gupta et. al. [32] utilized Euler transformation to obtain a graph that is continuously printable, i.e., a graph in which every vertex has an even number of edges. As they pointed o, while traversing such a graph, one might encounter traversing a vertex many times, which they call *crossover*. They avoided crossovers by offsetting the path around the necessary vertices. Lin et. al. [33] applied a continuous path planning method for finishing processes of compound surfaces, during which many retractions are made. First, they created a curvature map. Then, they generated the cutter contact points. They used those points as cities of a TSP problem, and used LKH algorithm to traverse all of them. They argue that cutter contact point generation is a highly time consuming process since all triangles inside the object must be searched in order to find one point. Feng et. al. [34] used triply periodic minimal surfaces (TPMS) for designing porous structures. Normally, TPMS structures cannot be printed as they define open surfaces. Therefore, they provided a method to make a printable object from a TPMS structure, in which Marching Squares algorithm is utilized during slicing. Ding et al. [35] stated that although widely used for AM, the contour paths, which start from the boundary of the geometry and proceed inwards, do not guarantee filling the geometry completely. To deal with this problem, they applied the methodology developed by Kao [36], in which the medial axis transformation (MAT) is used. Given a geometry, first the skeleton is generated by medial axis transformation. Based on the skeleton, they created branches to fill the geometry, and then trimmed them. For the boundary, though, they made extra deposition, which makes the path discontinuous. Their method is developed mainly for Wire and Arc Additive Manufacturing (WAAM). Hergel at al. [37] argue that there is a fundamental difference between printing thermoplastic materials and clay. Deposited thermoplastic material solidifies rapidly after extrusion whereas the filament inside the extruder is still liquid. Therefore, moving without extrusion enables the printed part to detach from the extruder immediately and results in only minor artifacts. However, when clay is printed, it requires much longer time for solidification. Thus, if a movement without extrusion is made, the extruder will pull the printed part, causing the shape to deform. As those deformations are repeated, the failure of the shape becomes more probable. In order to cope with this, they introduced a continuous extrusion method. Firstly, they took the geometry to be printed and add support structures. Then, they calculated a continuous path for the all shape, and made optimizations for support parts in order to reduce their complexities. Their method is used for 3D printing of complex shapes with clay. Nevertheless, sometimes their part could be collapsed as there is no methodology for considering the physics and the mechanical structure of the part. In addition, their path planning takes about 10 minutes. Overall, it is a successfull step for 3D printing using clay. Kapil et al. [38] used fractal curves for hybrid machining operations with the minimum number of retractions. After filling a given area with Hilbert curves, they removed the portions of the area that must be empty from it. Then, they connected the trimmed paths and made a path with minimum number of retractions, although not guaranteed to be zero. As another methodology, they put the largest fractal curves into an area and then filled in the remaining portions with zigzag curves, which resulted in a path with no retractions. They argue that the toolpaths from fractal curves result in better heat dissipation processes for AM of the metallic objects. Nonetheless, the parts they produced don't seem precise. Also, they point out that there exists lots of sharp turns and corners in the generated paths. To conclude, although the continuous deposition methods are not perfect yet, they are getting better and surpassing the traditional coverage algorithms in the field, in their respects.

Like continuous deposition, Voronoi diagrams are widely utilized in AM for numerous reasons. Martinez et al. [39] formed Voronoi diagrams based on not Euclidean, but polyhedral distances, which have foam shapes. They were able to create Voronoi infill patterns with distinct mechanical properties by tuning the distance parameters. They could control the deformation directions of the part under a specific load and make a part flexible in some directions, whereas rigid in another direction. However, in terms of the strength, their method is less effective than plane tessellations. In another study, the same team used Voronoi diagrams to achieve different Young's modulus values, while keeping the Poisson's ratio constant [40]. After a Young's modulus value is set, Voronoi foams are generated on the fly. Therefore, there was no need for producing the full geometry of the object, like a mesh or voxels. Nevertheless, as they point out, their method cannot provide a way of Poisson's ratio's change. Tran at al. [41] utilized 2D Voronoi cells to mimic the composite structure of the natural nacre, which already has a Voronoi-shaped polygonal architecture. They formed the diagram by setting Voronoi centers at random spots, whose ranges are limited. Afterwards, they filled the boundaries between the cells with a cohesive element in their model. Similarly, they connected each layer with another layer via an adhesive element. Then, they 3D printed their designs and made compression tests on them and were able to follow the crack propagation using the Voronoi boundaries. Chen and Zhai [42] developed a path planning strategy for porous structures, which are hard to produce by conventional production techniques and advantageous due to their weight and structural properties. First, the porous geometry is sliced. At each slice, they divide the geometry into subdomains, each of which include one hole, using the generalized Voronoi diagram (GVD) and dual operation. Then, a route to connect all subdomains is computed using a genetic algorithm, and during the traversal subdomains are merged with their neighbours along the path. Afterwards, subdomains are filled using Fermat spirals, and the smoothness of the path is optimized. Like connected Fermat spirals (CFS)[30], their method generates a continuous path for the whole geometry. However, since they used a contour-parallel filling method, their parts are affected by nonhomogeneous filling, i.e., overfill and underfill issues. Pham at al. [43] mimicked the crystal structure found in metals and alloys, by creating lattice cell structures. Different lattice cells, such as face-centered-cubic and body-centered-cubic, are implemented by Voronoi diagrams. Afterwards, they oriented the cells and combine them. The boundary between the two different cell structures introduced hardening, for which they made a set of experiments. From those studies, it can be inferred that Voronoi diagrams have a great potential for AM applications.

Besides from AM, Voronoi diagrams employed in the robotics field for area decomposition and path planning applications. Generalized Voronoi Diagrams (GVD) and Generalized Voronoi Graphs (GVG) can be utilized for different cases [15]. GVD can be useful for high level planning, as they consider the two closest objects and have *m-1* dimensions for a configuration space with *m* dimensions. However, if a more detailed planning is required, GVG can be utilized, which has a single dimension and considers the closest *m* objects for the same configuration space. After the necessary Voronoi diagram is constructed, different roadmap algorithms can be used to find a path. As the dimensions of the configuration space increase, it becomes harder to construct the exact Voronoi diagram, but using sampling strategies one can still take the advantage of Voronoi diagrams without computing them explicitly. Candeloro et al. [44] use Voronoi diagrams for path planning of 3 DoF marine vehicles. After the generation of the diagram, they compute some way points from the vertices and refine them. If the vehicle is not on the right track, Fermat spirals are used to guide it back to the path, by which a curvature-continuous path is achieved. If another ship is observed, a new and smooth path can be computed on the fly. Shkolnik and Tedrake [45] developed a method for path planning of the robotic manipulators. They started with a well-studied path planning algorithm, RRT [46]. However, instead of using the configuration space, they used the task space. As they exemplified, planning in a lower dimensional projection can be quite efficient for some problems. While sampling, they took random point from the task space, and while growing the tree, instead of using the configuration space like in RRT, they still used the task space by introducing a Voronoi bias, which allows for a more direct exploration. They could successfully plan a path for a robot with 1000 links. As they explain, though, they lost completeness, since they switch to the task space. Arslan and Koditschek [47] applied an extension to GVD for path planning of multiple disk shaped robots with different sizes without collision. As they also state, restricting the robot bodies to their respective Voronoi cells is sufficient for collision avoidance [48], although it is conservative for robots with different sizes.

CHAPTER 3

CONTINUOUS PATH PLANNING ALGORITHM

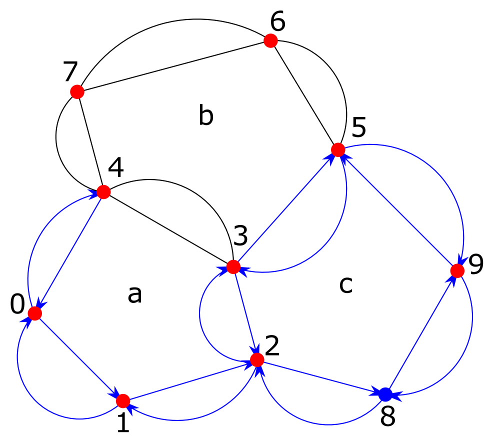
In this chapter, the path planning algorithm is explained, which is divided into two parts: *on a layer*, and *between the layers*. Additionally, a *G-code encryption* algorithm is presented after the path planning method.

3.1 Algorithm On a Layer

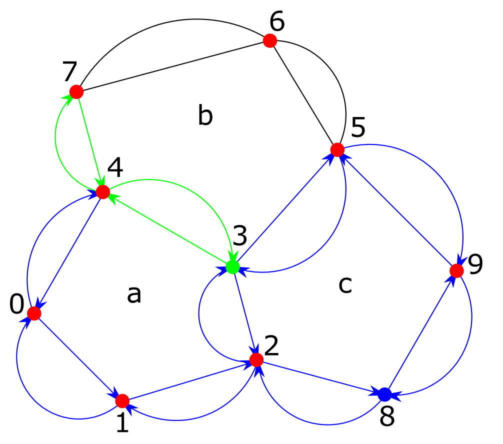
Proposed method is a variation of Hierholzer's algorithm. Therefore, first it is explained, then the method is introduced.

3.1.1 Hierholzer's Algorithm

Hierholzer's algorithm is an algorithm to extract an Eulerian cycle from an Eulerian graph [49]. Following the fact that every node has an even number of edges, starting the traversal from any node, one needs to end up at the same node. Then, another node among the path, that has connections can be chosen and a subpath can be constructed. It would end at the same node on the path. The path can be extended by inserting the subpath into it (Figure 3.1). Repeating this procedure until all the edges are traversed, the Eulerian cycle is obtained. Therefore, fast travels are eliminated. However, *u* turns are not considered.



(a) The path at the beginning of the traversal of the subpath, which is 8-2-1-0-4-0-1-2-3-5-9-8-9-5-3-2-8.



(b) The path after the subpath (3-4-7-4-3) is inserted, which is 8-2-1-0-4-0-1-2-3-5-9-8-9-5-3-4-7-4-3-2-8.

Figure 3.1: Hierholzer's algorithm.

3.1.2 Modified Hierholzer's Algorithm

Since Hierholzer's algorithm doesn't take u turns into account, new algorithm is developed, based on it. Before explaining the algorithm, why there always exists a path without u turns for the aforementioned graphs must be explained (Figure 3.2). As one can observe from the figure, when all of the cells, including boundary as a cell, is traversed in the same direction (CW in the figure), those paths can be merged such that no *u turn* will occur. Although a general Voronoi node has three neighbours, in some special cases the number of neighbours can increase. Nevertheless, for any even or odd number of neighbours, the given generalization is valid. The only inevitable u turn happens when a node has only one neighbour, as the node *f* in Figure 2.2. However, this case cannot be encountered in the presented graphs.

While the path is generated, a subpath is always inserted into the path in Hierholzer's algorithm. Therefore, if u turns are eliminated during the generation and insertion of subpaths, a path without u turns can be obtained. Modified Hierholzer's algorithm does that by traversing the edge that won't generate a u turn, and if there are more than one edges to take, the algorithm chooses the edge that has not been traversed before, if available. Note that, at the beginning all edges are duplicated and they are

Algorithm 1 Hierholzer's Algorithm

```
Input: A list of Voronoi cells, Starting node
Output: A path that traverses all of the edges twice
 1: Initialize Path as an empty list
 2: Create the Edge List that includes all connections of the Voronoi List
 3: Set the current node to the Starting node
 4: while Edge List is not empty do
 5:     Initialize SubPath as an empty list
 6:     while there exists an edge in the Edge List that starts from the current node
    do
 7:         Append the edge to the SubPath
 8:         Set the current node as the ending node of the last edge
 9:         Remove the edge from the Edge List
10:     end while
11:     Append the SubPath to the Path
12:     for each node in Path do
13:         if there exists an edge in the Edge List that starts from the node then
14:             Set that node as the current node
15:             break
16:         end if
17:     end for
18: end while
19: return Path
```

not directed, therefore an edge can be traversed in the same direction twice. Finally, the algorithm tries to insert the subpath into the path such that no u turns will occur. However, if it fails to do so, it reverses the subpath and then inserts into the path, by which it would find a path without u turns. Using this algorithm, both fast travels and u turns are eliminated.

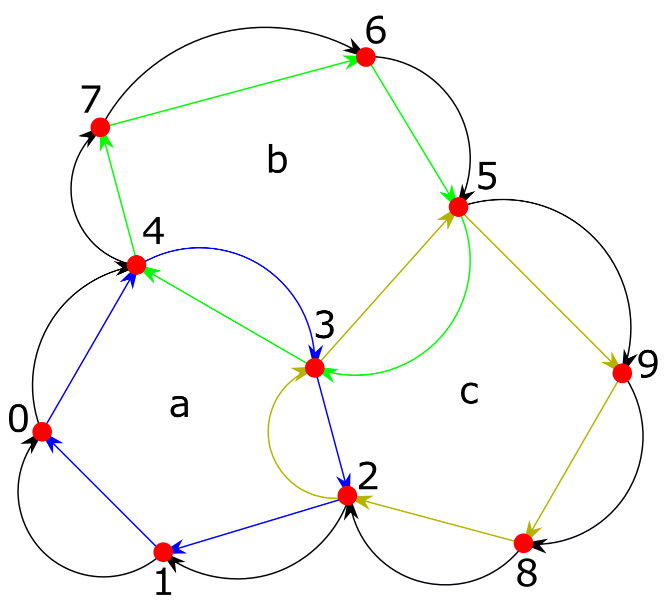


Figure 3.2: In presented graphs, it is always possible to have a *u turn free* path.

3.2 Converting a U Turn Heavy Path into a U Turn Free Path

Before the proposed algorithm for sequential layers is explained, let us discuss another algorithm, the one that makes a path with u turns into a u turn free path. This algorithm will be utilized in the next section with slight modifications. Note that, as proven in the previous section, it is always possible to construct a u turn free path in presented graphs. As it would be easier to follow, an example is provided for the explanation.

Suppose that the considered path is:

$8{-}9{-}8{-}2{-}1{-}2{-}3{-}4{-}3{-}9{-}5{-}4{-}5{-}6{-}7{-}0{-}4{-}0{-}1{-}0{-}7{-}6{-}5{-}9{-}3{-}2{-}8,$

which is u turn heavy. Since u turns are to be eliminated, the first step must be dividing the path into subpaths, wherever a u turn happens. After this divisions, the subpaths become:

$8 - 9$

$9 - 8 - 2 - 1$

$1 - 2 - 3 - 4$

Algorithm 2 Modified Hierholzer's Algorithm

```
Input: A list of Voronoi cells, Starting node
Output: A path that traverses all of the edges twice
 1: Initialize Path as an empty list
 2: Create the Edge List that includes all connections of the Voronoi List
 3: Set the current node to the Starting node
 4: while Edge List is not empty do
 5:     Initialize SubPath as an empty list
 6:     while there exists an edge in the Edge List that starts from the current vertex
    do
 7:         List available edges that won't result in u turn
 8:         if there exists an edge that never been traversed then
 9:             Append the edge to the SubPath
10:         else
11:             Append any of the edges to the SubPath
12:         end if
13:         Set the current node as the ending node of the last edge
14:         Remove the edge from the Edge List
15:     end while
16:     Append the SubPath to the Path such that no u turns will occur
17:     if failed to do so then
18:         Reverse the SubPath
19:         Append the SubPath to the Path such that no u turns will occur
20:     end if
21:     for each node in Path do
22:         if there exists an edge in the Edge List that starts from the vertex then
23:             Set that node as the current node
24:             break
25:         end if
26:     end for
27: end while
28: return Path
```

$4-3-9-5-4$

$4-5-6-7-0-4$

$4-0-1$

$1-0-7-6-5-9-3-2-8$

At this point, the subpaths can be examined. As observed, some of them start at a node, and end at another node. Let's call them *chains*. On the other hand, some of the subpaths start & end at the same node. Let's call them *loops*. Therefore, the subpaths are as follows according to the classification.

Chains:

$8-9$

$9-8-2-1$

$1-2-3-4$

$4-0-1$

$1-0-7-6-5-9-3-2-8$

Loops:

$4-3-9-5-4$

$4-5-6-7-0-4$

Chains might be added into a subpath, either from the beginning or from the end. Also, they can be reversed. For instance, the first chain, $8-9$, can be reversed and becomes $9-8$. Then, it can be added to the last chain, $1-0-7-6-5-9-3-2-8$. The resulting subpath will be $1-0-7-6-5-9-3-2-8-9$.

Loops, on the other hand, offer more flexibility, because one can change the starting and eventually ending point of them, which will be called *loop change*. For instance, the first loop, $4-3-9-5-4$ is equivalent to the following loops: $3-9-5-4-3$, $9-5-4-3-9$, $5-4-3-9-5$. Additionally, they can be reversed. Similar to the chains, they can be added to a subpath from the beginning or from the end. However, unlike the chains, they can be *inserted* into the subpaths. For instance, consider the chain $4-0-1$ and the loop $4-5-6-7-0-4$. The loop may be added to the chain from the beginning, which will result in the subpath $4-5-6-7-0-4-0-1$. Additionally, instead of doing that, one can make a *loop change* in the loop, and let it become $0-4-5-6-7-0$. Then, this loop can be inserted into the chain $4-0-1$, which will result in the subpath $4-0-4-5-6-7-0-1$. Note that, both subpaths created using the chain and the loop include 1 u turn, at $0-4-0$ and $4-0-4$ parts. In order not to have any u turn, the loop can simply be *reversed* before adding to or inserting into the chain. For the first case, the loop will become $4-0-7-6-5-4$ and the subpath will become $4-0-7-6-5-4-0-1$. For the second case, the loop and the subpath will become $0-7-6-5-4-0$ and $4-0-7-6-5-4-0-1$.

Note that, if there is a loop inside a subpath, that loop can always be reversed, no matter where it is positioned. For example, the subpath $1-2-3-4-5-6-7-4-9$ includes a loop, which is $4-5-6-7-4$. The loop can be reversed, and then the subpath becomes $1-2-3-4-7-6-5-4-9$. This strategy will be called *partial reversal*, to differentiate it from regular reversing operation, in which the subpath is fully reversed.

As the methods are described, which are, *full reversal*, *loop change*, and *partial reversal*; a general explanation of the algorithm can be given. After the path is divided into subpaths, they will be added one after the other using those three methods whenever necessary. Since there are no u turns in the subpaths, the constructed path will have no u turns either. That will be the methodology.

Going back to the example, after creating chains and the loops, the longest chain will be chosen as the *main path*, which is

$1-0-7-6-5-9-3-2-8.$

The longest chain will be deleted from the list of the chains. Now, one should enlarge the main path using chains or loops. Since loops are easier to integrate, the chains will be tried first. Looking at them, the first chain, $8-9$, can be added to the main path from the end, by which the main path becomes

$1 - 0 - 7 - 6 - 5 - 9 - 3 - 2 - 8 - 9.$

Continuing by removing $8 - 9$ from the chain list. Once again, looking at the remaining chains, $9 - 8 - 2 - 1$ cannot be added to the main path from the end as it would result in a u turn. However, this chain can be added from the beginning, then the main path becomes

$9 - 8 - 2 - 1 - 0 - 7 - 6 - 5 - 9 - 3 - 2 - 8 - 9.$

At this point, the main path becomes a *loop*. However, no *loop change* can be made for it, because it starts $(9-8)$ and ends $(9-8)$ with the same connection, and therefore any loop changes would cause a u turn (e.g. $2-1-0-7-6-5-9-3-2-8-9-8-2$). Nevertheless, if that were not the case, the loop changes could be freely made in the main path.

Looking at the remaining chains $(1 - 2 - 3 - 4\ ,\ 4 - 0 - 1)$, none of them seems to be useful. Then, the loops must be tried. The first one, $4 - 3 - 9 - 5 - 4$, can be taken, and the main path can be *inserted* into it, by which the main path becomes

$4 - 3 - 9 - 8 - 2 - 1 - 0 - 7 - 6 - 5 - 9 - 3 - 2 - 8 - 9 - 5 - 4.$

The main path is still a loop, but this time as it starts $(4 - 3)$ and ends $(5 - 4)$ with different connections, one can free to make *loop changes*. Continuing with the chains, the chain $1 - 2 - 3 - 4$ can be *reversed* and added to the main path from the end, and the result is

$4 - 3 - 9 - 8 - 2 - 1 - 0 - 7 - 6 - 5 - 9 - 3 - 2 - 8 - 9 - 5 - 4 - 3 - 2 - 1.$

Similarly, the last chain, $4 - 0 - 1$, might be reversed and added to the main path from the beginning, then the main path is

$1 - 0 - 4 - 3 - 9 - 8 - 2 - 1 - 0 - 7 - 6 - 5 - 9 - 3 - 2 - 8 - 9 - 5 - 4 - 3 - 2 - 1.$

As observed, all chains are already added to the main path, and the main path is a loop by itself. This is expected, because the (full) path must start and end at the same position, and the loops have no effect on the starting and ending positions unless they are inserted the main path into the loop. In the case of the main path is inserted into a loop, the starting and ending positions of the loop will become those of the main path. Note that, if main path can be inserted into a loop, it means that the loop can be inserted into the main path as well, it's just a matter of choice.

Finally, the only remaining loop will be inserted, $4-5-6-7-0-4$, into the main path. Therefore, the path is found as

$1-0-4-5-6-7-0-4-3-9-8-2-1-0-7-6-5-9-3-2-8-9-5-4-3-2-1.$

Applying this procedure, a path with many u turns is converted into a path with no u turns.

Algorithm 3 Path Converter - 1: Path Breaker

```
Input: A path with u turns
Output: A path without u turns
1: Initialize SubPaths as an empty list
2: for each Adjacent Connection in the Path do
3:     if a u turn exists then
4:         Divide the Path from that node
5:         Add the first part of the divided path to SubPaths
6:     end if
7: end for
```

3.3 Algorithm Between the Layers

Instead of planning the path for all of the layers, it is desired to plan it once (at the first layer), and then update it for the subsequent layers. Similar to the previous section, the algorithm will be explained using an example, and later the pseudocode will be given.

Consider a cylinder, populated by 3D Voronoi cells. Using a horizontal plane, this cylinder is sliced, and two adjacent layers are obtained (Figure 3.3). Intersection of the horizontal plane with a group of 3D Voronoi cells gives us a group of 2D Voronoi cells. Looking at the layers, it can be seen that, there are three Voronoi cells on Layer 0 (Figure 3.3a), namely, cells *a*, *b*, and *c*; and four nodes, *0*, *1*, *2*, and *3*. On the other hand, on Layer 1 (Figure 3.3b), there are four Voronoi cells, which are *a*, *b*, *c*, and *d*;

Algorithm 4 Path Converter - 2: Path Generator

```
 8: Initialize Chains and Loops as empty lists
 9: for each SubPath in the SubPaths do
10:     if SubPath starts and ends at the same node then
11:         Add SubPath to Loops
12:     else
13:         Add SubPath to Chains
14:     end if
15: end for
16: Initialize Main Path as an empty list
17: if Chains is not empty then
18:     Set the longest SubPath in Chains as Main Path
19: else
20:     Set the longest SubPath in Loops as Main Path
21: end if
22: while Chains is not empty and Loops is not empty do
23:     Initialize the boolean APPENDED as False
24:     for each SubPath in the Chains list do
25:         if SubPath or the reversed SubPath can be added to the Main Path then
26:             if Reversing is necessary then
27:                 Reverse the SubPath
28:             end if
29:             if Adding would not cause a u turn then
30:                 Add the SubPath to the Main Path
31:                 Set APPENDED as True
32:                 Remove SubPath from the Chains list
33:                 break
```

Path Converter - 2: Path Generator Part 2

```
34:             else
35:                 if Eliminating u turn by Partial loop change is possible then
36:                     Make Partial loop change in Main Path
37:                     Add the SubPath to the Main Path
38:                     Set APPENDED as True
39:                     Remove SubPath from the Chains list
40:                     break
41:                 end if
42:             end if
43:         end if
44:     end for
45:     if APPENDED is False then
46:         for each SubPath in the Loops list do
47:             if SubPath can be inserted into Main Path then
48:                 if Loop change is necessary then
49:                     Make Loop change
50:                 end if
51:                 Insert the SubPath to the Main Path
52:                 Remove SubPath from the Loops list
53:                 break
54:             end if
55:         end for
56:     end if
57: end while
58: return Main Path
```

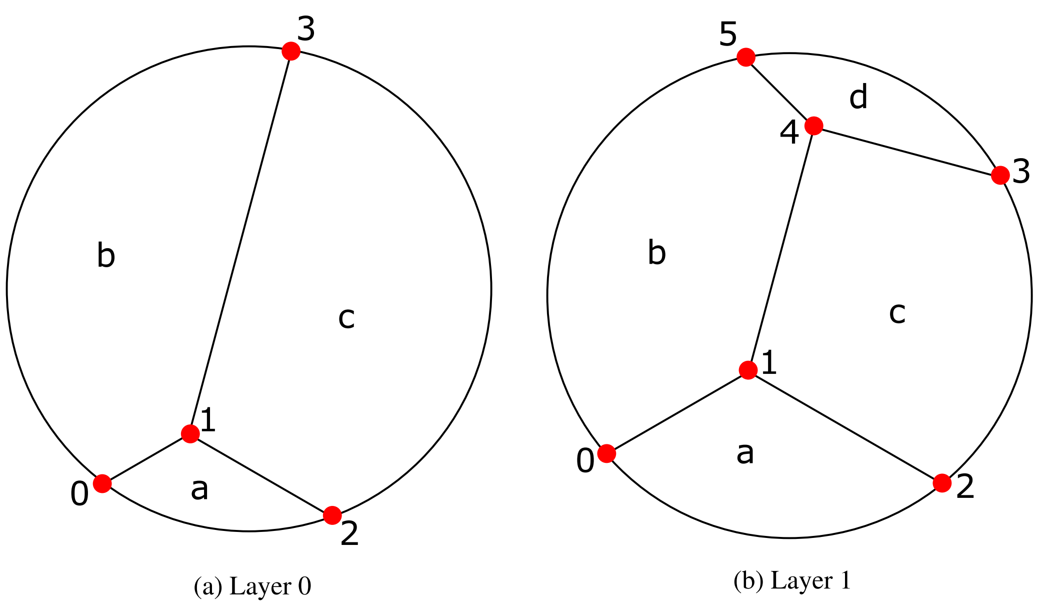


Figure 3.3: Two subsequent layers, taken from a cylinder, which is populated by 3D Voronoi cells.

and five nodes, *0*, *1*, *2*, *3*, and *4*.

The aim is to update the path of *Layer 0* for *Layer 1*. The path for Layer 0 is obtained as

$1 - 0 - 2 - 3 - 0 - 2 - 1 - 3 - 0 - 1 - 2 - 3 - 1.$

First of all, there are no u turns in this path, because it is obtained using the Modified Hierholzer's algorithm. A path that won't include any u turns for Layer 1 is to be obtained as well. Obviously, some connections of the previous path can be used in the current path, but not all of them. Also, the new connections have to be considered.

In order to meet the requirements, one must first proceed with detecting which parts of the previous path can be used. Then, those parts, i.e., subpaths, will be taken from the previous path, and the rest will be left. Afterwards, new connections will be considered. The subpaths from the previous path will be enlarged by new connections, such that no u turn takes place. Also, after the enlargement of the subpaths, if there are connections left, additional subpaths will be made from them. In the end, all subpaths will be connected to form a path. While connecting them, it must be ensured that no u turns will form, like in the previous section (Algorithm 4).

In order to understand which Voronoi cells changed, the number of nodes in each of them must be examined, which is shown in Table 3.1. The only cell, whose structure is not changed, is cell *a*.

Table 3.1: The nodes and the number of them in each Voronoi cell in Figure 3.3.

	Nodes		**# of Nodes**	
	Layer 0	**Layer 1**	**Layer 0**	**Layer 1**
a	*0*, *1*, *2*	*0*, *1*, *2*	3	3
b	*0*, *1*, *3*	*0*, *1*, *4*, *5*	3	4
c	*1*, *2*, *3*	*1*, *2*, *3*, *4*	3	4
d		*3*, *4*, *5*	0	3

As the unchanged cells are detected, one should check the nodes they include. In cell *a*, the nodes *0*, *1*, and *2* exist; which will be called *unchanged nodes*. The only *changed* node in Layer 0 is node *3*. Now, the previous path must be taken, which is $1-0-2-3-0-2-1-3-0-1-2-3-1$, and be divided into the subpaths by removing the *changed nodes*. By doing that, i.e., by removing node *3* from the path, the following subpaths are obtained:

$1-0-2$

$0-2-1$

$0-1-2$

~~-1~~

Note that, the last part, which is -1, is not a subpath, because it doesn't specify a connection. In order to make a connection, two nodes are required. Now, the subpaths from the previous path are ready, but the new connections are also needed, in order to enlarge the subpaths. To explain the subject better, the previous subpaths and the new connections are given in Figure 3.4.

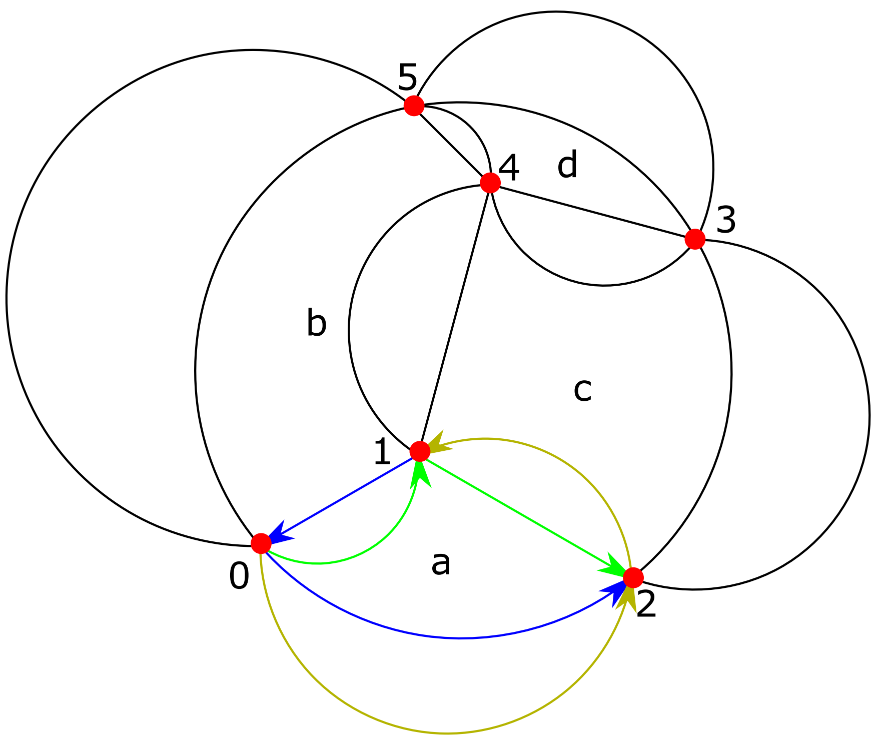


Figure 3.4: Layer 1 with duplicated connections. The subpaths from Layer 0 are shown in color, new connections are shown in black.

Using the new connections, the subpaths should be enlarged, such that no u turn occurs. Note that, subpaths must be enlarged in an expansive manner; if there are two connections with a node, but one connection with another, the one with two connections must be chosen. In other words, the bridges must not be burned. Starting with the blue subpath, $1-0-2$:

$1-0-2$

$1-0-2-3$

$1-0-2-3-4$

$1-0-2-3-4-5$

$1-0-2-3-4-5-0$

The only way to keep enlarging this subpath is to go back to node 5, but that would cause a u turn, therefore, the enlargement must be stopped. Switching to the yellow subpath, $0-2-1$:

$0 - 2 - 1$

$0 - 2 - 1 - 4$

$0 - 2 - 1 - 4 - 5$

$0 - 2 - 1 - 4 - 5 - 3$

$0 - 2 - 1 - 4 - 5 - 3 - 2$

Proceeding with the green subpath, $0 - 1 - 2$: It is not possible to enlarge it from node *2*, but it can be reversed and enlarged from node *0*:

$2 - 1 - 0$

$2 - 1 - 0 - 5$

$2 - 1 - 0 - 5 - 3$

$2 - 1 - 0 - 5 - 3 - 4$

$2 - 1 - 0 - 5 - 3 - 4 - 1$

During the enlargements, all connections are used and none is left, therefore, there is no need to create new subpaths. The enlarged subpaths are given in Figure 3.5.

The subpaths without u turns are made, and by adding them one by one, one should be able to construct the path. Checking the subpaths again, it is observed that starting with $1 - 0 - 2 - 3 - 4 - 5 - 0$, then adding $0 - 2 - 1 - 4 - 5 - 3 - 2$, and finally adding $2-1-0-5-3-4-1$, the path can be constructed, in a very straightforward fashion. Therefore, the path is obtained as

$1 - 0 - 2 - 3 - 4 - 5 - 0 - 2 - 1 - 4 - 5 - 3 - 2 - 1 - 0 - 5 - 3 - 4 - 1.$

In the example given, there was no need to use any operations from the previous section, namely *partial reversal*, *full reversal*, and *loop change*. Nevertheless, in the formal procedure the steps from the Algorithm 4 must be followed, during the path creation process from the subpaths. Utilizing the explained algorithm, the path can be updated for each layer.

Algorithm 5 Path Updater - 1: Path Breaker

Input: A path without u turns for layer *n*, lists of Voronoi cells for layers *n* & *n+1*

Output: A path without u turns for layer *n+1*

```
1: Initialize Changed Voronois as an empty list
2: Initialize Unchanged Nodes as an empty list
3: Initialize Changed Nodes as an empty list
4: for each cell in Voronoi cells list do
5:     if # of nodes of the cell is the same between layers $n$ & $n+1$ then
6:         for each node in the cell do
7:             Add node to Unchanged Nodes list
8:         end for
9:     else
10:        Add cell to Changed Voronois list
11:    end if
12: end for
13: for each cell in Changed Voronois list do
14:     for each node in cell do
15:         if node is not in Unchanged Nodes list then
16:             Add node to Changed Nodes list
17:         end if
18:     end for
19: end for
20: Initialize SubPaths as an empty list
21: for each Adjacent Connection in the previous Path do
22:     if a node from Changed Nodes list exists then
23:         Divide the Path from that node
24:         Add the first part of the divided path to SubPaths
25:     end if
26: end for
```

Algorithm 6 Path Updater - 2: SubPath Enlarger

```
27: Initialize New Connections as an empty list
28: for each cell in Changed Voronois list do
29:     if There are connections that are not already included in SubPaths then
30:         Add those connections to New Connections list
31:     end if
32: end for
33: while New Connections is not empty do
34:     Initialize the boolean APPENDED as False
35:     for each SubPath in SubPaths list do
36:         for each Connection in New Connections list do
37:             Try to enlarge SubPath, or the reversed SubPath, by adding Connec-
    tion
38:             if Enlarging is successfull then
39:                 Set APPENDED as True
40:                 Remove Connection from New Connections list
41:                 break
42:             end if
43:         end for
44:         if APPENDED is True then break
45:         end if
46:     end for
47:     if APPENDED is False then
48:         Set the first Connection in New Connections list as a SubPath
49:         Add SubPath to SubPaths list
50:         Remove Connection from New Connections list
51:     end if
52: end while
```

Algorithm 7 Path Updater - 3: Path Generator

```
53: Initialize Chains and Loops as empty lists
54: for each SubPath in the SubPaths do
55:     if SubPath starts and ends at the same node then
56:         Add SubPath to Loops
57:     else
58:         Add SubPath to Chains
59:     end if
60: end for
61: Initialize Main Path as an empty list
62: if Chains is not empty then
63:     Set the longest SubPath in Chains as Main Path
64: else
65:     Set the longest SubPath in Loops as Main Path
66: end if
67: while Chains is not empty and Loops is not empty do
68:     Initialize the boolean APPENDED as False
69:     for each SubPath in the Chains list do
70:         if SubPath or the reversed SubPath can be added to the Main Path then
71:             if Reversing is necessary then
72:                 Reverse the SubPath
73:             end if
74:             if Adding would not cause a u turn then
75:                 Add the SubPath to the Main Path
76:                 Set APPENDED as True
77:                 Remove SubPath from the Chains list
78:                 break
```

Path Updater - 3: Path Generator Part 2

```
            else
                if Eliminating u turn by Partial loop change is possible then
                    Make Partial loop change in Main Path
                    Add the SubPath to the Main Path
                    Set APPENDED as True
                    Remove SubPath from the Chains list
                    break
                end if
            end if
        end if
    end for
    if APPENDED is False then
        for each SubPath in the Loops list do
            if SubPath can be inserted into Main Path then
                if Loop change is necessary then
                    Make Loop change
                end if
                Insert the SubPath to the Main Path
                Remove SubPath from the Loops list
                break
            end if
        end for
    end if
end while
return Main Path
```

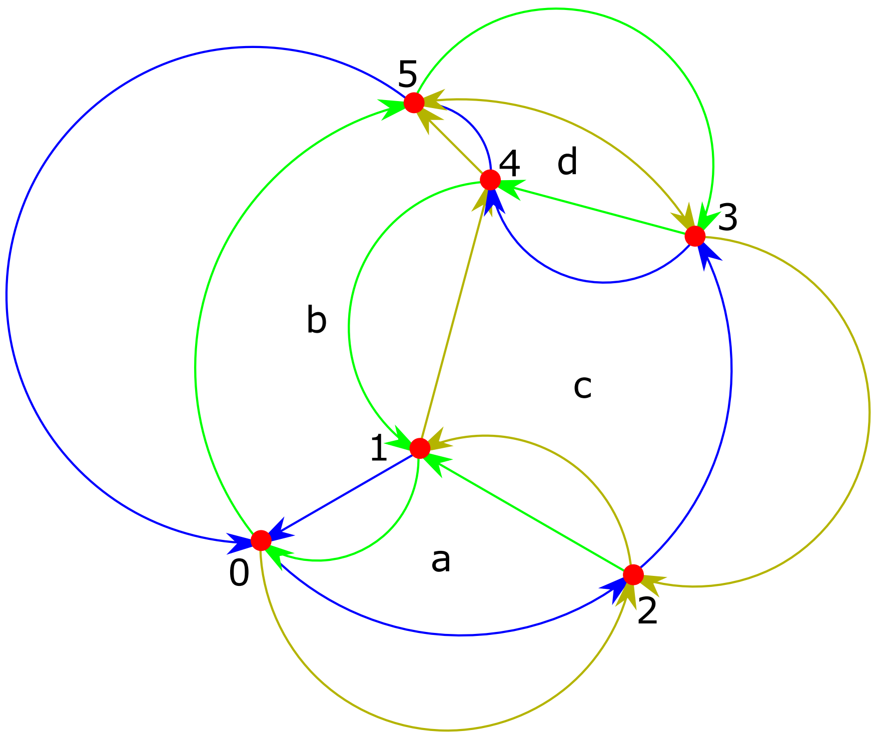


Figure 3.5: Subpaths are enlarged using available connections. Note that the green subpath is reversed during the enlargement.

3.4 G-code Encryption

Aside from the path planning algorithms, an encryption method for G-codes is developed. In a G-code file, the explicit coordinates of the visited nodes must always be written; at least the coordinates that are changing, which are usually the x and y coordinates of the nodes as the z coordinate is the same for the whole layer during 3D printing.

It is proposed to store the G-code information by using two text files: *paths.txt* & *corners.txt*.

In *paths.txt*, the path information by node numbers will be stored, which are *integers*. For instance, for a path of $1 - 2 - 0 - 2 - 1$,

1 2 0 2 1 will be written to *paths.txt*,

because the path nodes are separated using a *blank space*. Furthermore, let's say that there is a fast travel and afterwards another path traversal, and the path is $1-2-0-2-1->-3-2-3$, where $->-$ denotes the *fast travel*. In that case,

1 2 0 2 1 G3 2 3 must be written,

where *G* denotes *going to another segment*, or *fast travel*.

In the second text file, *corners.txt*, the coordinate information of the nodes will be stored. The *x* and *y* coordinates of the nodes will be written together, by separating a comma. After all nodes' *x* and *y* coordinates are written, the *z* coordinate of them will be added, which is the same for all the nodes, at a layer. Continuing with the example path, $1-2-0-2-1->-3-2-3$. Node coordinates for this example are given in Table 3.2.

Table 3.2: The coordinates of all the nodes in the example path, 1-2-0-2-1->-3-2-3.

	Coordinates		
	x	***y***	***z* (Layer Height)**
0	1.25	2.75	5.00
1	2.25	3.50	5.00
2	1.75	2.00	5.00
3	3.00	1.50	5.00

Given those coordinate values, one should write

1.25,2.75 2.25,3.50 1.75,2.00 3.00,1.50 5.00

to *corners.txt*. The node numbers, which are integers, are used as index values for *node coordinates* list. Therefore, if the largest node number in the path is *n*, the length of the node coordinates list for that path will be *n+2*: As the index starts from 0, there are *n+1* coordinate values, and at the end there is a height value. Note that, whether or not there exists a fast travel does not affect node coordinates. Another difference with the *paths.txt* is that *floats* are used, instead of integers. However, there is an exception: Assuming that the largest node number in a given path is *n*, one might encounter that some nodes that are smaller than *n* are absent in the path. Since node numbers are used as index values, values for *missing nodes* must be set as well. For those missing nodes, ***x*** will be written to *corners.txt*.

In both *paths.txt* and *corners.txt*, a layer is denoted by a *full line*. In other words, going to the *next layer* means switching to the *next line*, and then start writing to that line, for the new layer. To demonstrate all of the rules, a complete example shall be given.

Assume that, in the first layer, the generated path is $1-2-0-2-1- > -3-2-3$, for which the coordinate values were given in Table 3.2. Then, at the next layer, the first part of the path is disappeared, and the path became $3-2-3$, and the coordinate values of this path are provided at Table 3.3.

Table 3.3: The coordinates of all the nodes in the path at next layer, 3-2-3.

	Coordinates		
	x	*y*	*z* **(Layer Height)**
0	-	-	-
1	-	-	-
2	1.85	2.20	5.10
3	2.95	1.60	5.10

In that case, *paths.txt* and *corners.txt* will become:

paths.txt
```
1: 1 2 0 2 1 G3 2 3
2: 3 2 3
```

corners.txt
```
1: 1.25,2.75 2.25,3.50 1.75,2.00 3.00,1.50 5.00
2: x x 1.85,2.20 2.95,1.60 5.10
```

As it is shown, all the required information is stored in those files. If required, one can easily use a script to generate the G-code file from them.

3.4.1 Case of 2.5D Parts

If the grid structure and the boundary conditions are not changing for a material, in other words, if the x and y coordinates of the nodes are the same for subsequent layers, that advantage is taken in the encryption method. In such a case, the path will be the same, and only z coordinates of the nodes will be different.

If 2.5D printing case occurs, nothing will be written to *paths.txt*, i.e., but one must go to line afterwards. On the other hand, only the *height* value will be written to the *corners.txt*, before going to the next line.

Consider the example previously given. Let's say, the path $1-2-0-2-1->$ $-3-2-3$ is valid for three layers, with same coordinate values, except for the height. Then, assume, the second part vanishes and the path becomes $3-2-3$, and stays the same for the following two layers. In such a case, the two text files must be as such:

paths.txt

```
1: 1 2 0 2 1 G3 2 3
2:
3:
4: 3 2 3
5:
```

corners.txt

```
1: 1.25,2.75 2.25,3.50 1.75,2.00 3.00,1.50 5.00
2: 5.10
3: 5.20
4: x x 1.85,2.20 2.95,1.60 5.30
5: 5.40
```

Using this encryption method, compared to the G-code file, the size for the storage can be greatly reduced, which will be shown in the following chapter.

Algorithm 8 G-code Encryption

```
Input:  Paths List & Coordinates List for all layers
Output: paths.txt, corners.txt
 1: Initialize paths.txt and corners.txt as empty files
 2: Write the first layer's path to paths.txt
 3: Write the first layer's coordinates to corners.txt
 4: for each remaining layer in the Path List do
 5:     Go to the next line in paths.txt
 6:     Go to the next line in corners.txt
 7:     if 2.5D printing occurs (coordinates except for the height are the same) then
 8:         Write the heigth to corners.txt
 9:     else
10:         Write path to paths.txt
11:         if There are missing nodes then
12:             write x for them to corners.txt
13:         end if
14:         Write path node coordinates to corners.txt
15:         Write the heigth to corners.txt
16:     end if
17: end for
18: return paths.txt, corners.txt
```

3.5 Complexity Analysis of the Algorithms

Considering the pseudocodes provided, complexities of the algorithms are calculated. Although the implementation can greatly change how fast an algorithm runs, complexity analysis might still be used as a tool for comparison.

In this section, the main operations in each algorithm are taken into account. The complexities of these operations are determined, and finally they are summed to find the complexity of the overall algorithm.

3.5.1 Hierholzer's Algorithm

Assuming there are *n* edges in the graph, operations and their complexities for the Hierholzer's Algorithm is given in Table 3.4.

Table 3.4: Complexity of operations for the Hierholzer's Algorithm.

Operation	**Complexity**
Appending the edges	$O(n)$
Checking if the last node has an edge	$O(n)$
If the last node has no edge, checking the path	$O(k)$

Since $k < n$, complexity of the Hierholzer's Algorithm is found as $O(n)$.

3.5.2 Modified Hierholzer's Algorithm

Assuming there are *n* edges in the graph, operations and their complexities for the Modified Hierholzer's Algorithm is given in Table 3.5.

Since $k < n$, complexity of the Modified Hierholzer's Algorithm is found as $O(n)$.

Table 3.5: Complexity of operations for the Modified Hierholzer's Algorithm.

Operation	**Complexity**
Appending the edges	$O(n)$
Checking if the last node has an edge	$O(n)$
Checking if that edge causes a u turn	$O(n)$
If the last node has no edge or u turn occurs, checking the path	$O(k)$

3.5.3 Path Converter Algorithm

This algorithm has two parts: *Path Breaker* and *Path Generator*. Assuming there are *n* edges in the graph, operations and complexities for the first part is given in Table 3.6.

Table 3.6: Complexity of operations for the Path Breaker Algorithm.

Operation	**Complexity**
Checking if a u turn exists	$O(n)$
Dividing the path at u turn locations	$O(k)$

Complexity of the Path Breaker Algorithm is found as $O(n)$, since $k < n$. For the second part, assuming there are *m* subpaths, operations and complexities are given in Table 3.7.

Table 3.7: Complexity of operations for the Path Generator Algorithm.

Operation	**Complexity**
Distributing subpaths to chains and loops	$O(m)$
Setting the longest subpath as the path	$O(m)$
Enlarging the path by adding subpaths	$O(m^2)$

Complexity of the Path Generator Algorithm is found as $O(m^2)$. By considering the two parts, complexity of the Path Converter Algorithm is found as $O(m^2 + n)$.

3.5.4 Path Updater Algorithm

This algorithm has three parts: *Path Breaker*, *Subpath Enlarger*, and *Path Generator*. Assuming there are *n* edges in the graph, operations and complexities for the first part is given in Table 3.8.

Table 3.8: Complexity of operations for the Path Breaker Algorithm.

Operation	**Complexity**
Checking if number of nodes in each cell are changed	$O(n)$
Checking if Changed nodes exist in Unchanged Nodes list	$O(n^2)$
Checking if previous path has any Changed nodes	$O(n)$
Dividing the path at Changed nodes locations	$O(k)$

Complexity of the Path Breaker Algorithm is found as $O(n^2)$. For the second part, assuming there are *n* edges, operations and complexities are given in Table 3.9.

Table 3.9: Complexity of operations for the Subpath Enlarger Algorithm.

Operation	**Complexity**
Finding new connections from Changed Voronois	$O(n)$
Appending new connections to subpaths	$O(n)$
Checking if that edge causes a u turn	$O(n)$

Complexity of the Subpath Enlarger Algorithm is found as $O(n)$. For the third part, assuming there are *m* subpaths, operations and complexities are given in Table 3.10.

Complexity of the Path Generator Algorithm is found as $O(m^2)$. By considering the three parts, complexity of the Path Updater Algorithm is found as $O(m^2 + n^2)$.

Table 3.10: Complexity of operations for the Path Generator Algorithm.

Operation	**Complexity**
Distributing subpaths to chains and loops	$O(m)$
Setting the longest subpath as the path	$O(m)$
Enlarging the path by adding subpaths	$O(m^2)$

3.5.5 G-code Encryption Algorithm

Assuming there are *n* lines in the G-code file, operations and their complexities for the G-code Encryption Algorithm is given in Table 3.11.

Table 3.11: Complexity of operations for the G-code Encryption Algorithm.

Operation	**Complexity**
Writing necessary data to *paths.txt*	$O(n)$
Writing necessary data to *corners.txt*	$O(n)$
Checking if 2.5D printing occurs at each layer	$O(n)$

Complexity of the G-code Encryption Algorithm is found as $O(n)$.

CHAPTER 4

IMPLEMENTATION & TEST CASES

In this chapter, implementation details of the developed algorithms are discussed. Afterwards, conducted tests with various shapes are given, and their results are elaborated.

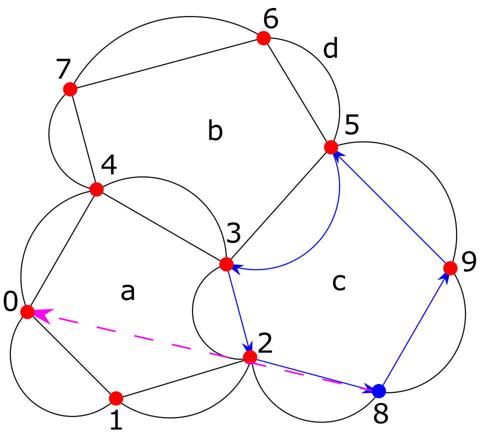
4.1 Algorithms Utilized in Test Cases for Comparison

4.1.1 Naive Algorithm

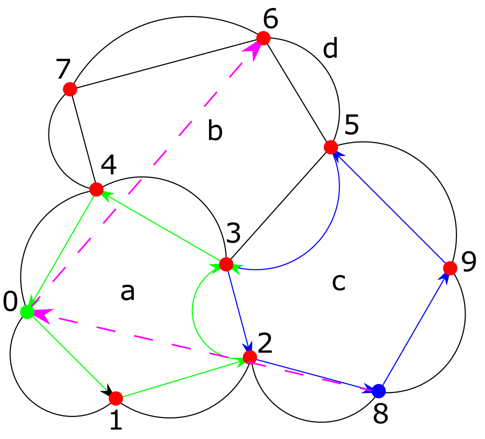
As observed from the graphs (Figure 3.1), if one traverses all of the cells once, all the edges are traversed. Naive algorithm uses this observation to operate and executes a lot of fast travel motions. After completing all edges of a cell, it goes to the starting node of the next cell and traverses all of its edges. Unless the starting nodes of the adjacent cells are the same, which is decided by the list of vertices and nodes constructed by Grasshopper, the algorithm employs fast travels (Figure 4.1). Although this algorithm is inefficient, it can be easily implemented and can be used to study the effect of fast travels.

4.1.2 Improved Naive Algorithm

At a first glance at the naive algorithm, its main inefficiency can be conceived as the unnecessary numbers of fast travels. For instance, instead of travelling from point *i* to point *a* after the first traversal (Figure 4.1a), the algorithm could start traversing the boundary, which would eliminate the fast travel. Later, it may go to the closest



(a) The path at the beginning of the traversal of cell *a*.



(b) The path at the beginning of the traversal of cell *b*.

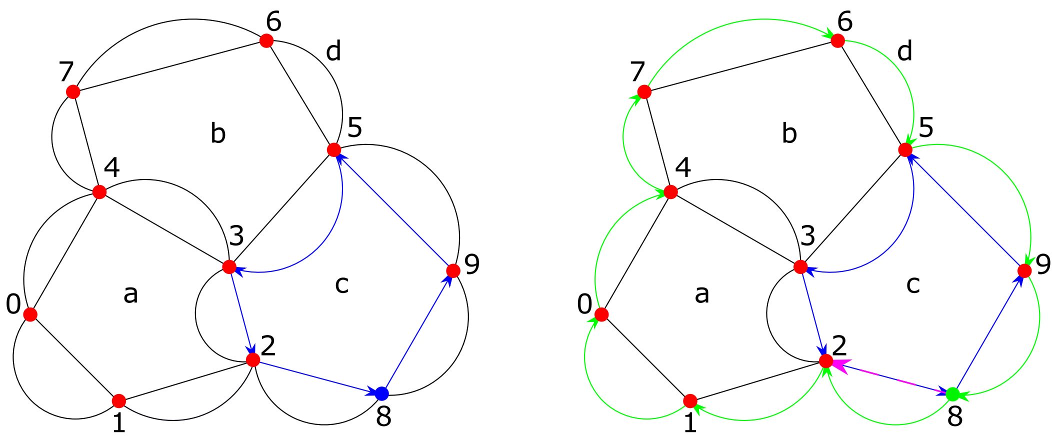
Figure 4.1: Naive algorithm, which traverses every cell one by one, without optimization.

Algorithm 9 Naive Algorithm

Input: A list of Voronoi cells, Starting node

Output: A path that traverses all of the edges twice

```
1: Initialize Path as an empty list
2: for each Voronoi cell in the Voronoi List do
3:     for each edge in the Voronoi cell do
4:         Append the edge to the Path
5:     end for
6: end for
7: return Path
```



(a) The path at the beginning of the traversal of the boundary cell (*d*).

(b) The path at the beginning of the traversal of cell *a*.

Figure 4.2: Improved naive algorithm, which traverses a cell completely, then considers the closest cell.

node of an uncompleted cell and traverse that edge. The improved naive algorithm works in that fashion by optimizing the path considering one step ahead (Figure 4.2). Even though this algorithm doesn't eliminate the fast travels in general, it's still a significant improvement over the naive algorithm.

4.1.3 Algorithm of Yaman et al.

The algorithm proposed by Yaman et al. [1] provides an Eulerian cycle for an Eulerian graph, following three simple rules. Starting from node 8 in cell *c*, path is computed by explaining the rules of the algorithm.

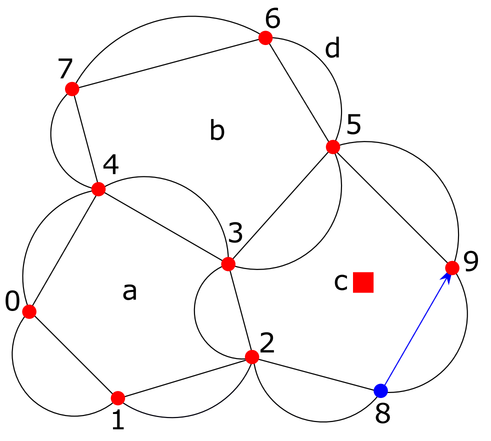
First of all, the algorithm keeps track of whether a cell is marked, i.e., any of its edges is traversed or not. At the beginning, none of the cells are marked. After the first traversal (8-9), cell *c* is marked. Beside, it keeps track of the node that has the same edge that is currently traversed, which is called as the *neighbour cell*. For instance, after the first traversal, neighbour cell is *d*, since it has the edge 9-8. The first rule of the algorithm is that if the neighbour cell is not marked yet, the edge of the neighbour cell is added to the path. Therefore, neighbour cell becomes the current cell and it is marked. At this stage, a *u* turn is observed (Figure 4.3).

Algorithm 10 Improved Naive Algorithm

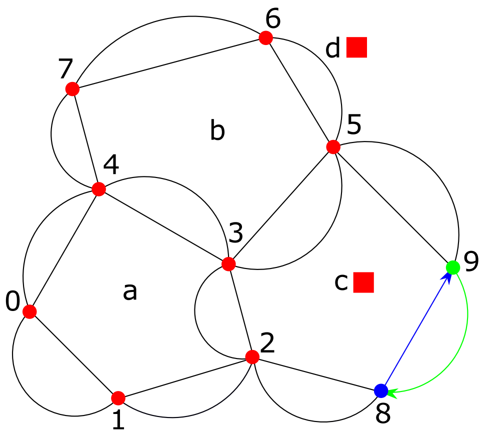
```
Input: A list of Voronoi cells, Starting node
Output: A path that traverses all of the edges twice
 1: Initialize Path as an empty list
 2: Set current node as the starting node
 3: Find a cell that includes the current node and set it to the current cell
 4: while Voronoi List is not empty do
 5:     for each edge in the current cell do
 6:         Append the edge to the Path
 7:     end for
 8:     Remove the current cell from the Voronoi List
 9:     if the current node is in the Voronoi List then
10:         Find a cell that includes the current node and set it to the current cell
11:     else
12:         Find the closest node, set it to the current node
13:         Find a cell that includes the current node and set it to the current cell
14:     end if
15: end while
16: return Path
```

If both current cell and neighbour cell are marked, and if there are connections for the current node, the algorithm chooses to stay in the current cell, which is the second rule (Figure 4.4).

The final rule of the algorithm considers the case in which all edges of the current cell are traversed & the neighbour cell is marked. In that case, if current node has connections in two cells, the algorithm chooses to go to the cell that is not a neighbour cell (Figure 4.5). Using those three rules, Euler cycle of the graph is obtained. Hence, fast travels are eliminated.

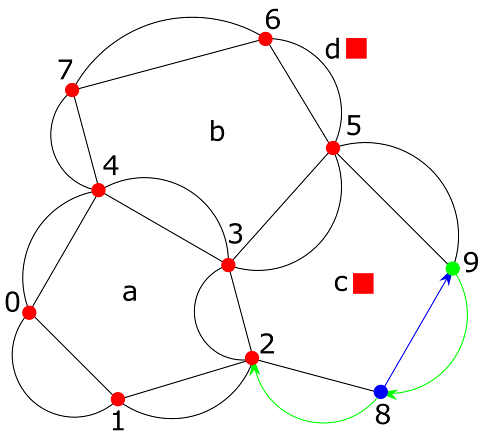


(a) After the first traversal (8-9), cell *c* is marked.

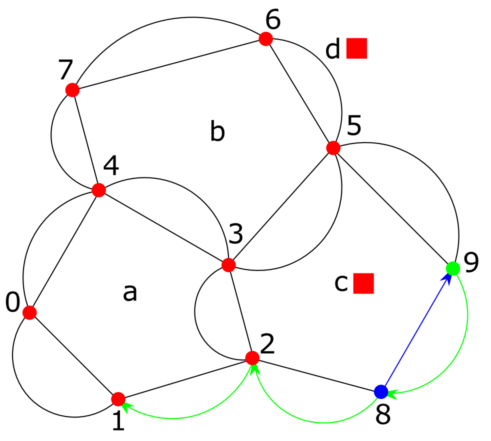


(b) Since neighbour cell (*d*) is not marked, algorithm moves into it.

Figure 4.3: The first rule of the algorithm by Yaman et al. [1].

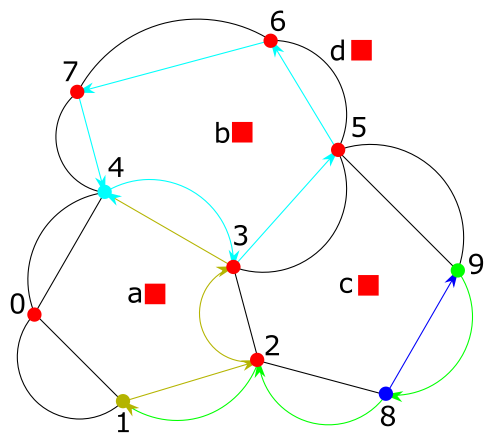


(a) After the traversal of 9-8 (Figure 4.3b), 8-2 is traversed in cell *d*.

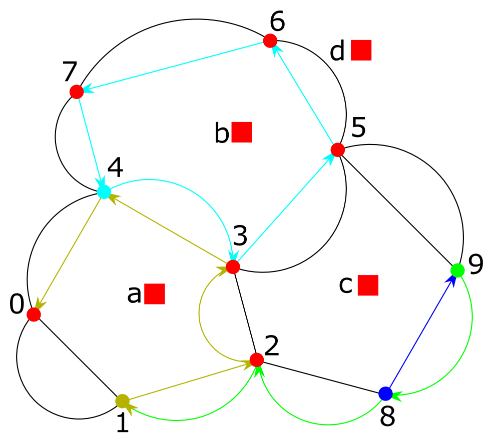


(b) After the traversal of 8-2, 2-1 is traversed in cell *d*.

Figure 4.4: The second rule of the algorithm by Yaman et al. [1].



(a) After the traversal of 7-4, no edge is left in *b*. The path so far is 8-9-8-2-1-2-3-4-3-5-6-7-4.



(b) Instead of the neighbour cell *d*, cell *a* is chosen and the edge 4-0 is traversed.

Figure 4.5: The third rule of the algorithm by Yaman et al. [1].

4.1.4 Hierholzer's Algorithm

Hierholzer's algorithm is another algorithm that is used for comparison, which was already explained in Section 3.1.1.

4.1.5 Ultimaker Cura

In order to have a comparison with the native software of the 3D printer used, Ultimaker 3 Extended, the algorithm of the printer is also utilized.

4.2 Complexity Analysis of the Algorithms

Considering the pseudocodes provided, complexities of the algorithms are calculated, as done in Section 3.5.

Firstly, the main operations in each algorithm are elaborated. Then, the complexities of these operations are determined, and finally they are summed to find the complexity of the overall algorithm.

Algorithm 11 Algorithm of Yaman et al.

```
Input: A list of Voronoi cells, Starting node
Output: A path that traverses all of the edges twice
 1: Initialize Path as an empty list
 2: Create the Edge List that includes all connections of the Voronoi List
 3: Create the Marked List as an empty list
 4: Find an edge that starts with the Starting node
 5: Set the current edge to the edge
 6: Append the current edge to the Path
 7: Set the current cell to the corresponding cell
 8: Append the current cell to Marked List
 9: Remove the current edge from the Edge List
10: while Edge List is not empty do
11:     Find the Neighbour cell
12:     if Neighbour cell is not marked then
13:         Set the current cell to the Neighbour cell
14:     else if there exists an edge into another cell then
15:         Set the current cell to the another cell
16:     else
17:         Current cell is unchanged
18:     end if
19:     Set the current edge to the edge that belongs to the current cell and starting
        with the current node
20:     Append the current edge to the Path
21:     Remove the current edge from the Edge List
22:     Append the current cell to the Marked List
23: end while
24: return Path
```

4.2.1 Naive Algorithm

Assuming there are *n* edges in the graph, operations and their complexities for the Naive Algorithm is given in Table 4.1.

Table 4.1: Complexity of operations for the Naive Algorithm.

Operation	**Complexity**
Appending the edges	$O(n)$

Complexity of the Naive Algorithm is found as $O(n)$.

4.2.2 Improved Naive Algorithm

Assuming there are *n* edges in the graph, operations and their complexities for the Improved Naive Algorithm is given in Table 4.2.

Table 4.2: Complexity of operations for the Improved Naive Algorithm.

Operation	**Complexity**
Appending the edges	$O(n)$
Checking if the current node is in Voronoi list	$O(n)$
Finding the closest node	$O(n)$
Finding a cell that includes the closest node	$O(n)$

Complexity of the Improved Naive Algorithm is found as $O(n)$.

4.2.3 Algorithm of Yaman et al.

Assuming there are *n* edges in the graph, operations and their complexities for the Algorithm of Yaman et al. [1] is given in Table 4.3.

Complexity of the Algorithm of Yaman et al. [1] is found as $O(n)$.

Table 4.3: Complexity of operations for the Algorithm of Yaman et al. [1].

Operation	**Complexity**
Appending the edges	$O(n)$
Setting the current cell	$O(n)$
Finding the neighbour cell	$O(2n)$
Appending cell to the marked list	$O(n)$
Decicing on the next node	$O(n)$

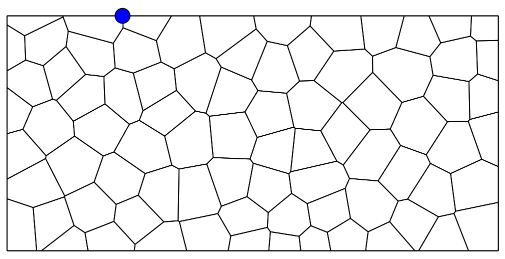
4.3 Implementation

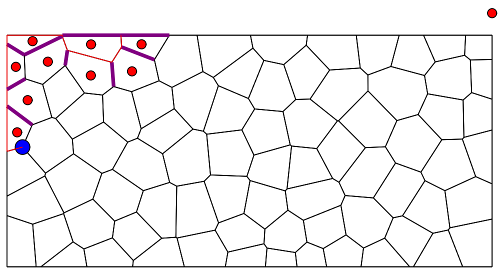
Rhino3D and Grasshopper3D are used for the generation of 3D Voronoi cells after object files are imported into them. Slicing operations and algorithm implementations are done using the GhPython module of Grasshopper3D. Furthermore, G-code files, and the text files that include the nodes to traverse and the coordinates of the nodes are generated within GhPython modules.

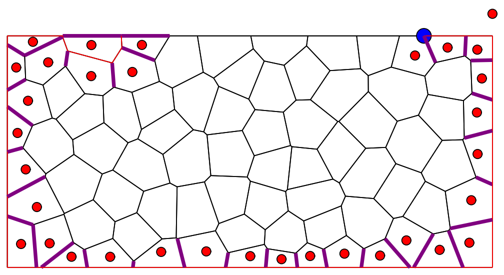
First, the object file is imported. Then, they are sliced by specified vertical distances from the bottom to the top. At each layer, the intersection of the 3D Voronoi cells and the horizontal plane is obtained, which is a 2D Voronoi graph. Afterwards, using the discussed algorithms, paths are generated. Finally, G-code and text files are updated for that layer. This operation is done for all of the layers.

4.4 Simulation

For a better understanding of the algorithms, simulations are performed using Tkinter module of Python. Snapshots of the simulation for a scene with 80 Voronoi cells are provided in Figure 4.6. Note that the algorithm used in this simulation is the one proposed by Yaman et al. [1]. Therefore, the cells are sequentially marked by the algorithm during the traversal and a large number of *u* turns are present. Note that, marked cells are shown by red dots, whereas the first traversal is shown by a *thin* red,


(a) At the beginning, no cells are marked.


(b) As the cells are traversed, they are being marked.


(c) The algorithm spreads throughout all the cells.

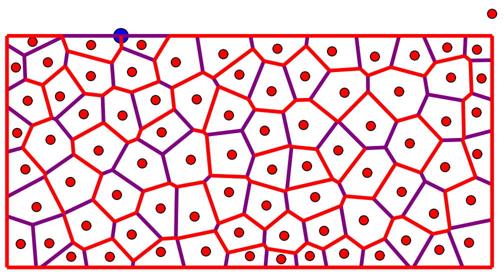
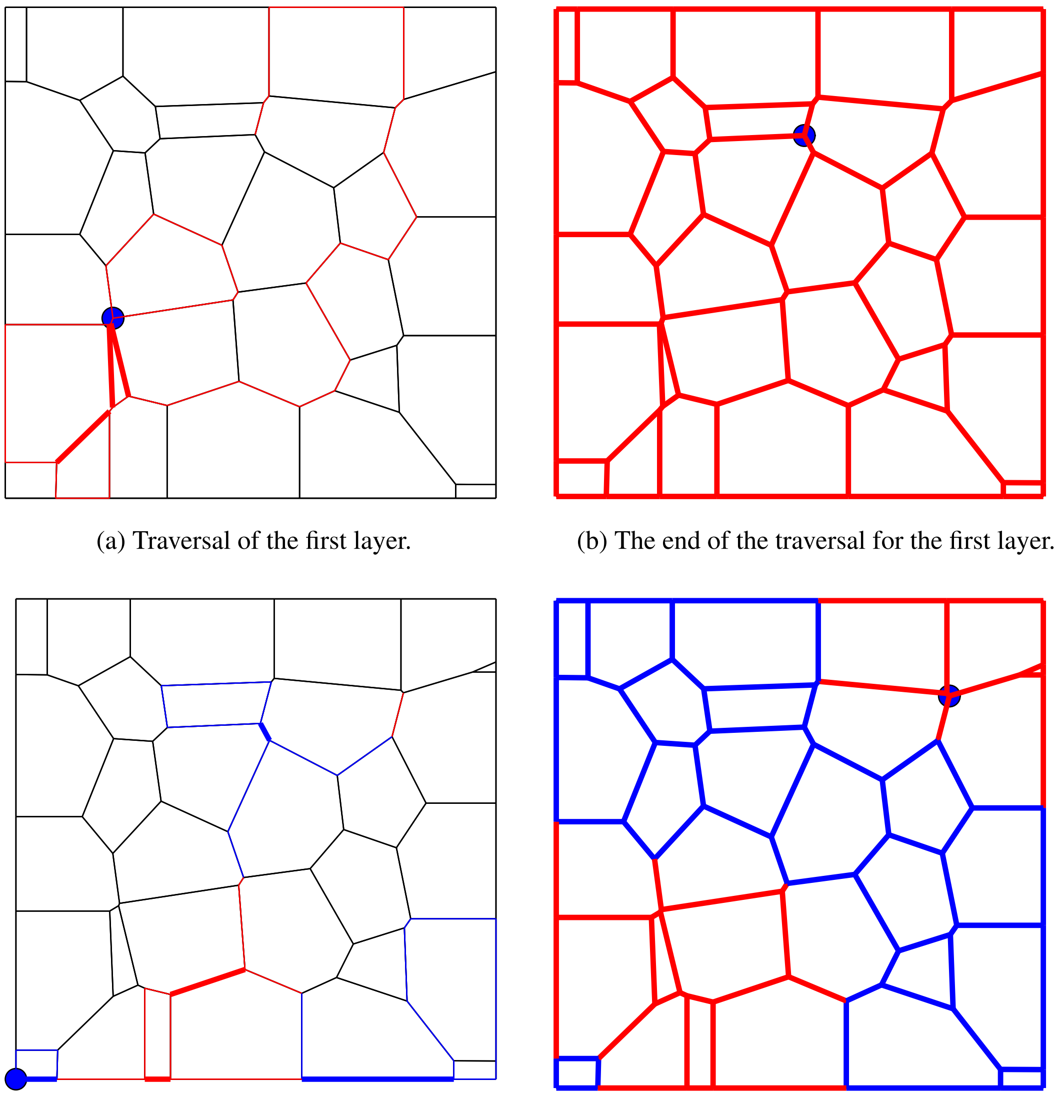

(d) Traversal is ended at the starting node.

Figure 4.6: Simulation for a graph with 80 cells.

and second traversal is shown by either a *thick* red (if there is no *u turn*) or purple (if a *u turn* exists) line.

Another simulation is performed for the proposed algorithm. A cube is first filled with 100 random Voronoi cells (Section 4.5.1.1), then two layers that are 0.5 mm apart are considered. Since the Modified Hierholzer algorithm makes no *u turns*, no purple line is seen on the first layer. For the second layer, due to the proposed algorithm, the path is not calculated from the beginning, but by updating the path of the previous layer. In the simulation, the edges that are taken from the previous path are shown in blue, whereas new connections, i.e., the edges belonging to the modified Voronoi cells are shown in red.

The simulation includes all off the algorithms except the Hierholzer's algorithm, since the modified version is already provided. Additionally, sample environments with different complexities are given. Three supplementary videos are also available. The first one explains all of the algorithms in a relatively simple environment, which can



(a) Traversal of the first layer.

(b) The end of the traversal for the first layer.

(c) Traversal of the second layer.

(d) The end of the traversal for the second layer.

Figure 4.7: Simulation for two sequential layers of a cube filled with 100 Voronoi cells.

be found at `https://youtu.be/C4R8sq5XM1k`. The second one compares the algorithm proposed by Yaman et al. [1] and the Modified Hierholzer's algorithm in an environment with 80 Voronoi cells, which is available at `https://youtu.be/zPRsC1rkaNw`. The last one exemplifies the proposed algorithm, whose screenshots are given in Figure 4.7. It can be seen at `https://youtu.be/YChOYG3NCjA`.

4.5 Test Cases

In all of the test prints, the objects are first filled with randomly placed 3D Voronoi cells, and then with cube-shaped Voronoi cells. The latter results in a square type of grid structure on each layer. This structure is useful for comparison with the native Ultimaker software (Cura). Furthermore, in a grid structure some part of the path from the previous layer might be used in the next layer, which would be advantageous for the proposed method. Algorithms are compared by their execution times and the size of the G-code files. Additionally, the sizes of the G-code files are compared with those of the files generated by the algorithm, namely *corners.txt* and *paths.txt*.

For both randomly-shaped and cube-shaped Voronoi cells, two models (sparse and dense Voronoi distribution) are generated. Results of these cases are tabulated and elaborated in the below subsections of the chapter.

4.5.1 Cube

In this case, a cube is studied. Size of the cube is 30 mm. The layer thickness is set to be 0.1 mm, which results in 300 layers. Since it is an 2.5D object, encryption can effectively be made, especially for grid-type structures.

4.5.1.1 Cube with 100 Random 3D Voronoi Cells

In this case, 100 randomly distributed 3D Voronoi cells are generated inside the cube (Figure 4.8). The results are given in Table 4.4. Without 2.5D encryption, the sum of *corners.txt* and *paths.txt* becomes 407 KB, which enables 73.8% reduction compared to the G-code with a size of 1556 KB. With 2.5D encryption, the reduction becomes 84.2%. Proposed algorithm takes 9.1% less time compared to the Modified Hierholzer's algorithm.

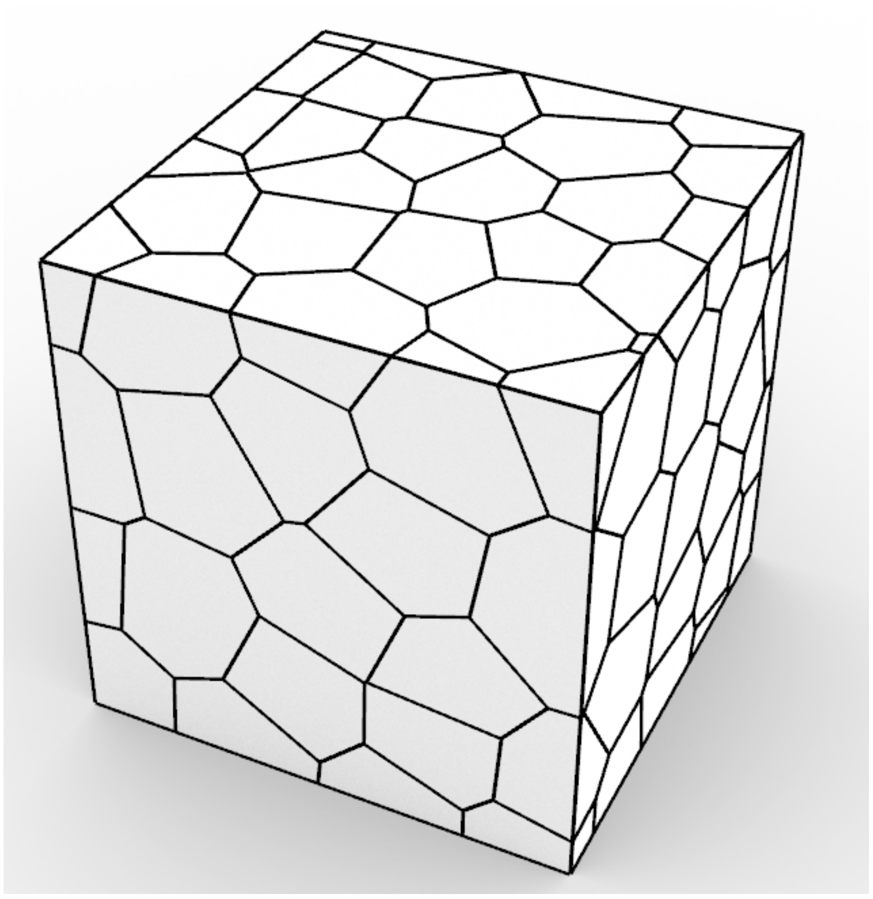


Figure 4.8: The cube with 100 randomly placed 3D Voronoi cells.

4.5.1.2 Cube with 400 Random 3D Voronoi Cells

In this case, 400 randomly placed 3D Voronoi cells are generated inside the cube (Figure 4.9). The results are given in Table 4.5. Without 2.5D encryption, the sum of *corners.txt* and *paths.txt* becomes 1047 KB, which enables 72.7% reduction compared to the G-code file with a size of 3840 KB. With 2.5D encryption, the reduction becomes 73.6%. Proposed algorithm takes 10.3% less time compared to the Modified Hierholzer's algorithm.

Table 4.4: The results for the case of cube with 100 Voronoi cells.

Algorithm	**Time [s]**	**Size [KB]**			**Fast Travels**	**U Turns**
		G-code	**corners**	**paths**		
Naive	17.347	1797			8465	7
Improved Naive	23.323	1776			8178	133
Hierholzer's	20.982	1556			300	375
Modified Hierholzer's	21.651	1556			300	0
Yaman et. al.	29.643	1556			300	5036
Proposed (w/o 2.5D Enc.)	19.688		247	160	300	0
Proposed (w/ 2.5D Enc.)	19.688		149	97	300	0
Ultimaker Cura	NA					

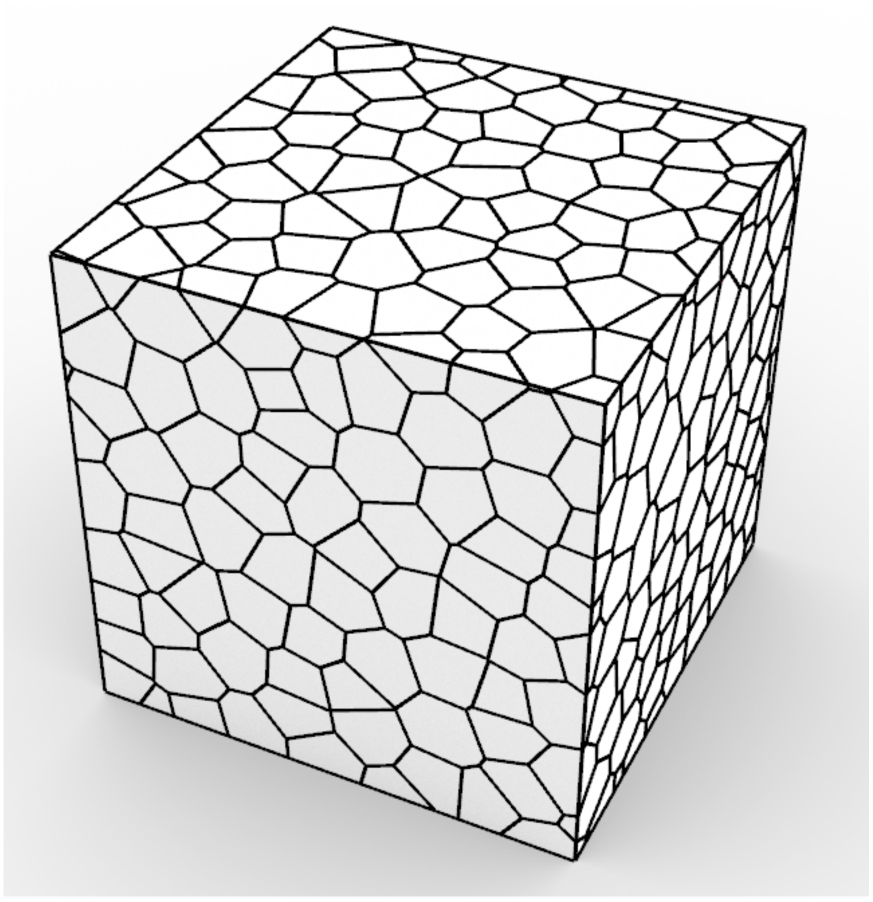


Figure 4.9: The cube with 400 randomly placed 3D Voronoi cells.

Table 4.5: The results for the case of cube with 400 Voronoi cells.

Algorithm	Time [s]	Size [KB]			Fast Travels	U Turns
		G-code	corners	paths		
Naive	44.361	4519			21237	27
Improved Naive	75.205	4495			20642	191
Hierholzer's	60.664	3840			300	1204
Modified Hierholzer's	63.777	3840			300	0
Yaman et. al.	118.103	3840			300	12933
Proposed (w/o 2.5D Enc.)	57.238		616	431	300	0
Proposed (w/ 2.5D Enc.)	57.238		598	417	300	0
Ultimaker Cura	NA					

4.5.1.3 Cube with 64 Square Voronoi Cells on XY-Plane

In this case, a square grid structure is obtained, which is equivalent to 64 vertical Voronoi cells (Figure 4.10). The results are given in Table 4.6. Without 2.5D encryption, the sum of *corners.txt* and *paths.txt* becomes 486 KB, which enables 79.5% reduction compared to the G-code file with a size of 2374 KB. With 2.5D encryption, the reduction becomes 99.6%. This result is expected, since the object is a 2.5D shape. Proposed algorithm takes 23.6% less time compared to the Modified Hierholzer's algorithm.

4.5.1.4 Cube with 441 Square Voronoi Cells on XY-Plane

In this case, a square grid structure is obtained, which is equivalent to 441 vertical Voronoi cells (Figure 4.11). The results are given in Table 4.7. Without 2.5D encryption, the sum of *corners.txt* and *paths.txt* becomes 3750 KB, which enables 77.0% reduction compared to the G-code file with a size of 16238 KB. With 2.5D encryption, the reduction becomes 99.7%. Proposed algorithm takes 58.6% less time compared to the Modified Hierholzer's algorithm.

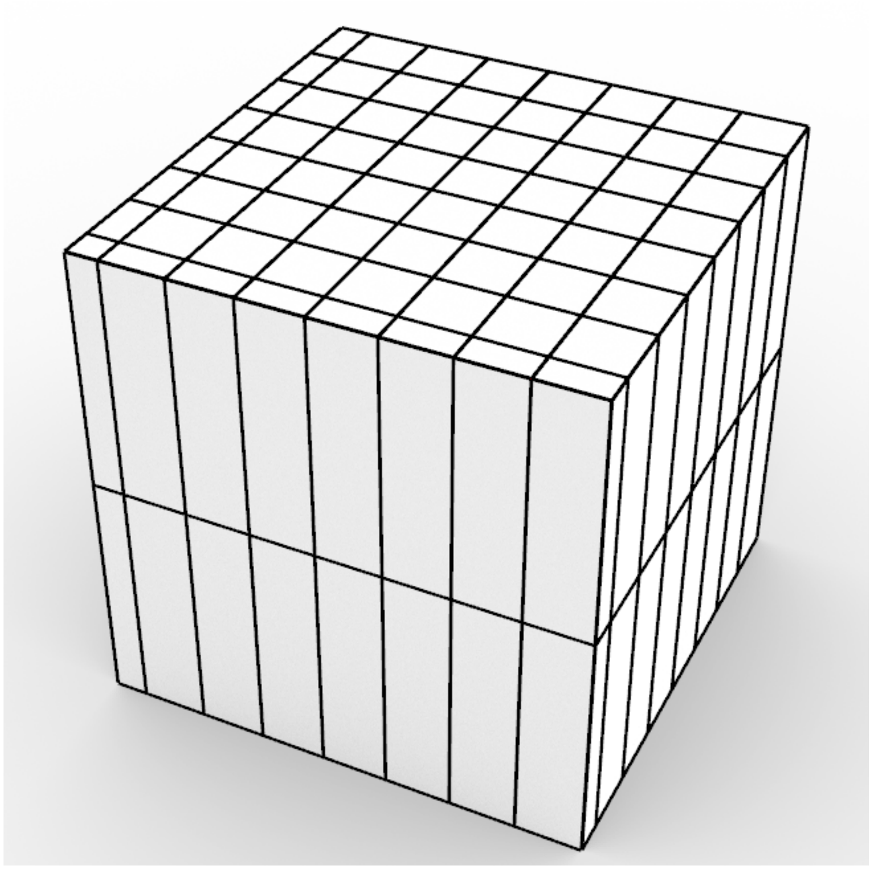


Figure 4.10: The cube with 64 Voronoi cells at each layer.

Table 4.6: The results for the case of cube with 64 Voronoi cells.

Algorithm	Time [s]	Size [KB]			Fast Travels	U Turns
		G-code	corners	paths		
Naive	25.218	2936			17361	12
Improved Naive	40.118	2780			14111	1248
Hierholzer's	32.388	2374			300	218
Modified Hierholzer's	32.826	2374			300	0
Yaman et. al.	58.369	2374			300	11685
Proposed (w/o 2.5D Enc.)	25.091		240	246	300	0
Proposed (w/ 2.5D Enc.)	25.091		5	4	300	0
Ultimaker Cura	2.430	1062			5107	0

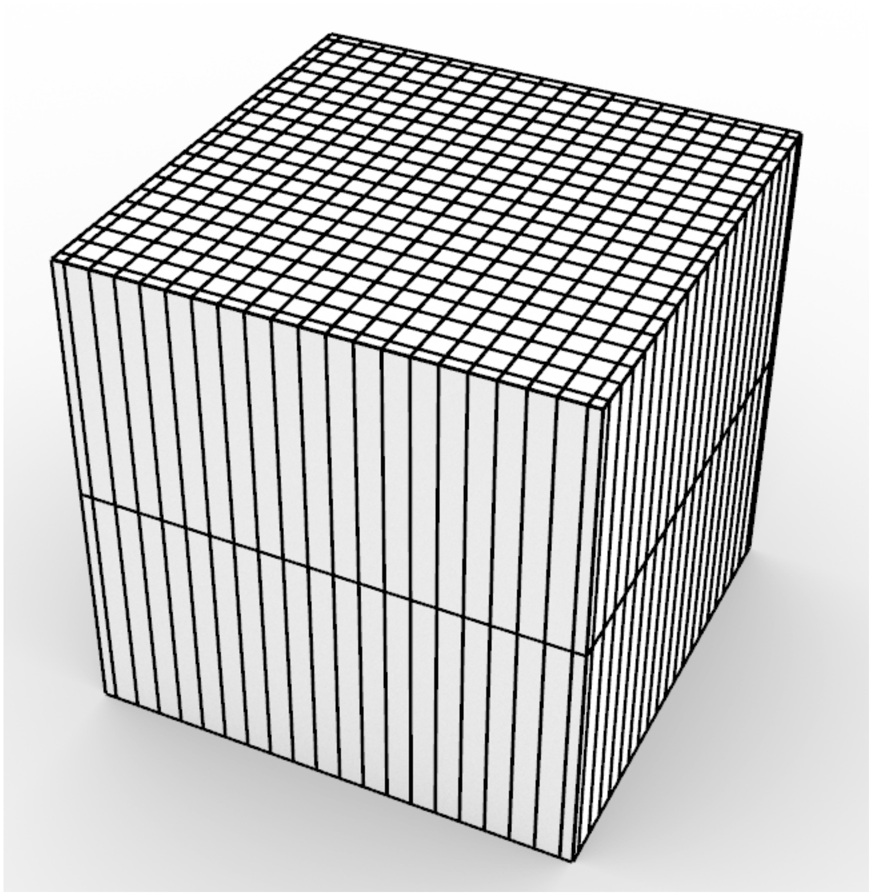


Figure 4.11: The cube with 441 Voronoi cells at each layer.

Table 4.7: The results for the case of cube with 441 Voronoi cells.

Algorithm	**Time [s]**	**Size [KB]**			**Fast Travels**	**U Turns**
		G-code	**corners**	**paths**		
Naive	168.247	21188			117113	38
Improved Naive	661.472	21025			113396	5097
Hierholzer's	431.660	16238			300	996
Modified Hierholzer's	452.468	16238			300	0
Yaman et. al.	1639.285	16238			300	83154
Proposed (w/o 2.5D Enc.)	187.233		1684	2056	300	0
Proposed (w/ 2.5D Enc.)	187.233		19	22	300	0
Ultimaker Cura	2.560	1771			5107	0

4.5.2 Sphere

In this case, another solid object, a sphere is utilized. Radius of the sphere is 20 mm. 398 layers are obtained after slicing with a layer thickness of 0.1 mm. For the naive algorithm, a different intersection method is applied, thus it takes longer than usual.

The boundary of the sphere is changing continuously throughout the layers. Therefore, little 2.5D encryption can be made.

4.5.2.1 Sphere with Randomly Placed 100 3D Voronoi Cells

In this case, 100 randomly placed 3D Voronoi cells are generated inside the sphere (Figure 4.12). The results are given in Table 4.8. Without 2.5D encryption, the sum of *corners.txt* and *paths.txt* becomes 1401 KB, which enables 70.3% reduction compared to the G-code file with a size of 4719 KB. With 2.5D encryption, the reduction becomes 70.4%. Proposed algorithm takes 1.9% less time compared to the Modified Hierholzer's algorithm.

Table 4.8: The results for the case of sphere with 100 Voronoi cells.

Algorithm	**Time [s]**	**Size [KB]**			**Fast Travels**	**U Turns**
		G-code	**corners**	**paths**		
Naive	141.550	5319			14409	24
Improved Naive	143.156	5379			13698	274
Hierholzer's	103.460	4719			398	617
Modified Hierholzer's	104.161	4719			398	0
Yaman et. al.	154.178	4719			398	9166
Proposed (w/o 2.5D Enc.)	102.194		889	512	301	0
Proposed (w/ 2.5D Enc.)	102.194		886	510	301	0
Ultimaker Cura	NA					

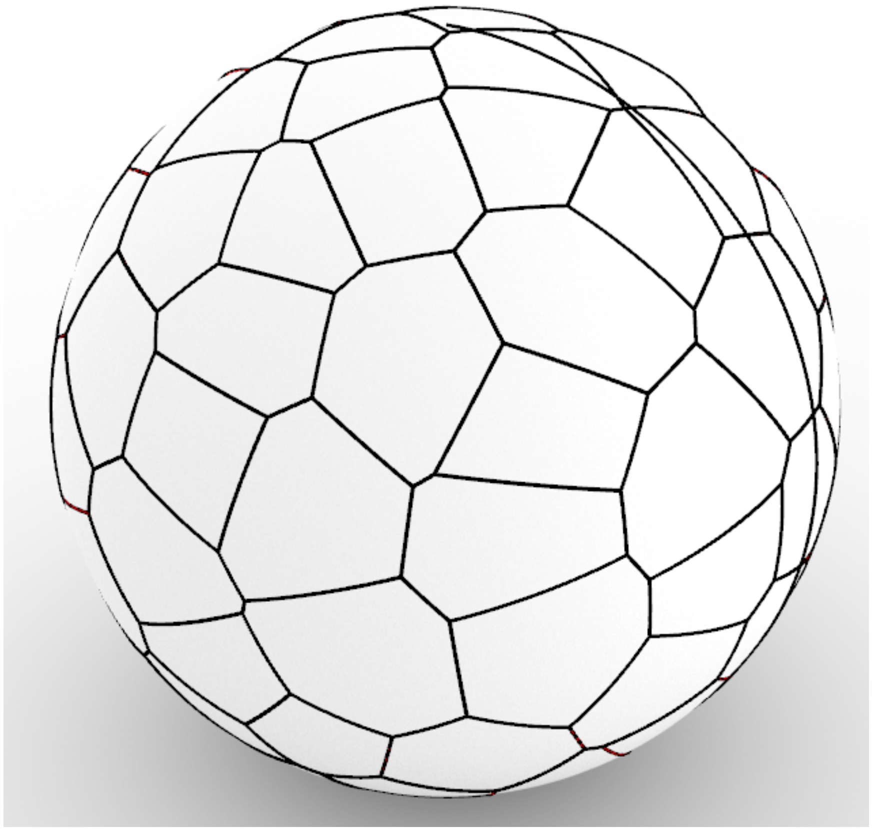


Figure 4.12: The sphere with 100 randomly placed 3D Voronoi cells.

4.5.2.2 Sphere with Randomly Placed 400 3D Voronoi Cells

In this case, 400 randomly placed 3D Voronoi cells are generated inside the sphere (Figure 4.13). The results are given in Table 4.9. Without 2.5D encryption, the sum of *corners.txt* and *paths.txt* becomes 3090 KB, which enables 71.1% reduction compared to the G-code file with a size of 10700 KB. For this case, no 2.5D encryption can be applied. Proposed algorithm takes 25.1% less time compared to the Modified Hierholzer's algorithm.

4.5.2.3 Sphere with 81 Square Voronoi Cells on XY-Plane

In this case, a square grid structure is created, which is equivalent to 81 Voronoi cells at each layer (Figure 4.14). Note that, not all of the grid cells intersect the object at each layer. The results are given in Table 4.10. Without 2.5D encryption, the

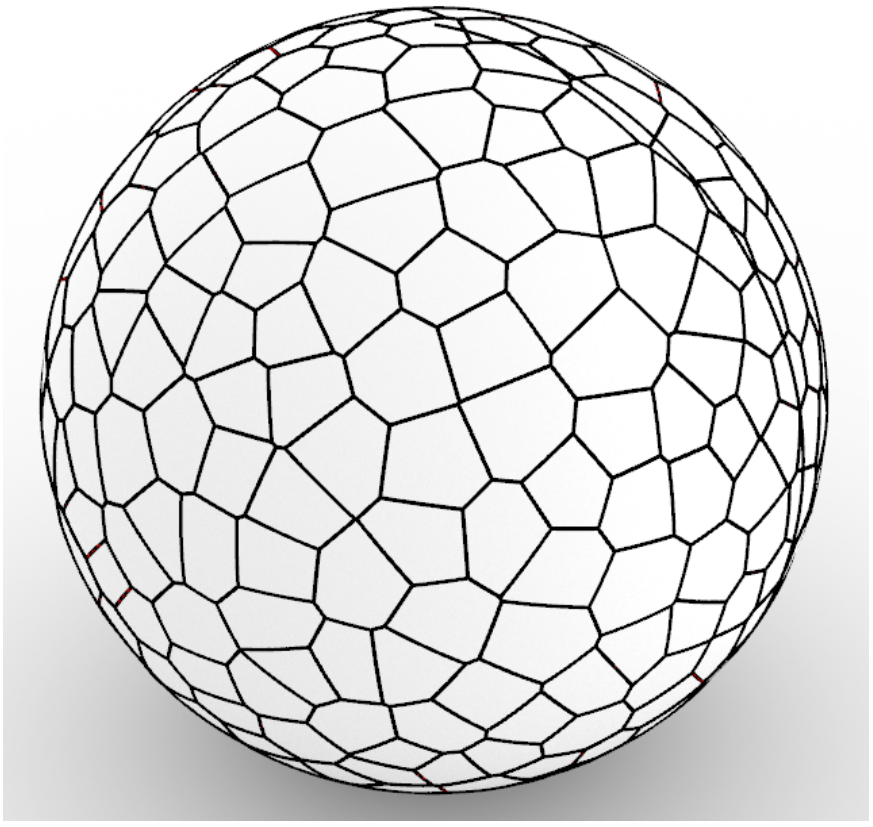


Figure 4.13: The sphere with 400 randomly placed 3D Voronoi cells.

Table 4.9: The results for the case of sphere with 400 Voronoi cells.

Algorithm	**Time [s]**	**Size [KB]**			**Fast Travels**	**U Turns**
		G-code	**corners**	**paths**		
Naive	249.917	12847			47827	4
Improved Naive	545.098	12981			46471	431
Hierholzer's	278.849	10700			398	3204
Modified Hierholzer's	282.767	10700			398	0
Yaman et. al.	645.060	10700			398	28494
Proposed (w/o 2.5D Enc.)	211.797		1833	1257	301	0
Proposed (w/ 2.5D Enc.)	211.797		1833	1257	301	0
Ultimaker Cura	NA					

sum of *corners.txt* and *paths.txt* becomes 1302 KB, which enables 71.2% reduction compared to the G-code file with a size of 4524 KB. For this case, no 2.5D encryption can be applied. Proposed algorithm takes 7.7% less time compared to the Modified Hierholzer's algorithm.

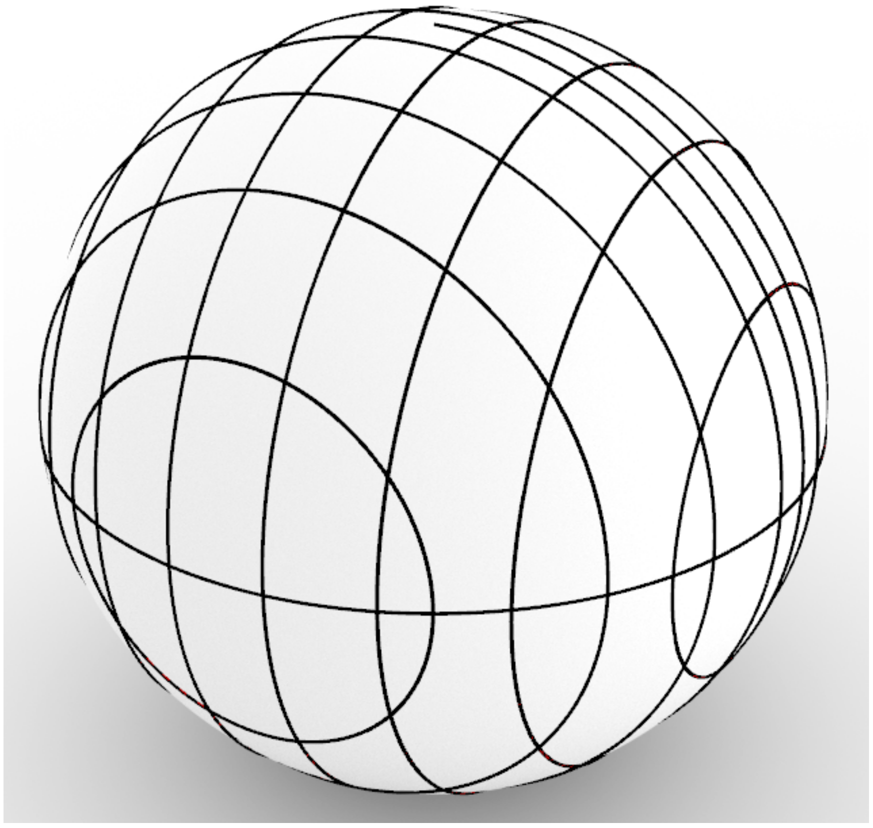


Figure 4.14: The sphere with 81 Voronoi cells at each layer.

Table 4.10: The results for the case of sphere with 81 Voronoi cells.

Algorithm	**Time [s]**	**Size [KB]**			**Fast Travels**	**U Turns**
		G-code	**corners**	**paths**		
Naive	135.905	5175			18369	16
Improved Naive	167.718	5094			12728	1861
Hierholzer's	97.862	4524			398	221
Modified Hierholzer's	100.098	4524			398	0
Yaman et. al.	143.734	4524			398	12555
Proposed (w/o 2.5D Enc.)	92.377		786	516	398	0
Proposed (w/ 2.5D Enc.)	92.377		786	516	398	0
Ultimaker Cura	4.360	7748			13317	0

4.5.2.4 Sphere with 441 Square Voronoi Cells on XY-Plane

In this case, a square grid structure is created, which is equivalent to 441 Voronoi cells at each layer (Figure 4.15). Note that, not all of the grid cells intersect the object at each layer. The results are given in Table 4.11. Without 2.5D encryption, the sum of *corners.txt* and *paths.txt* becomes 3379 KB, which enables 74.3% reduction compared to the G-code file with a size of 13157 KB. For this case, no 2.5D encryption can be applied. Proposed algorithm takes 35.6% less time compared to the Modified Hierholzer's algorithm.

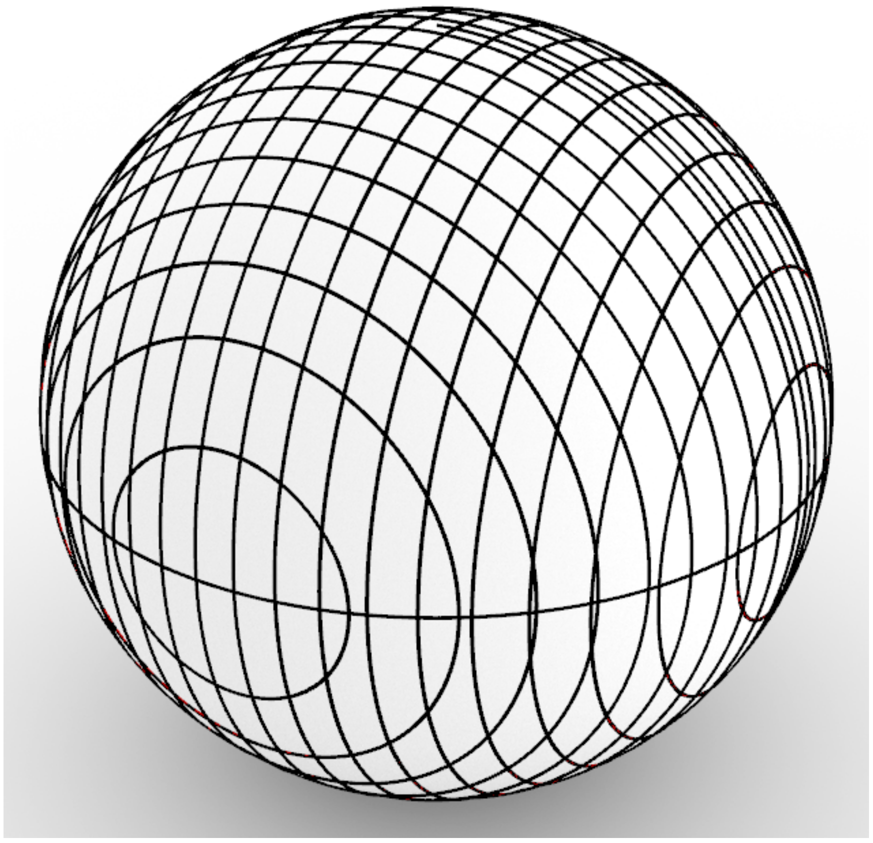


Figure 4.15: The sphere with 441 Voronoi cells at each layer.

4.5.3 Custom Shape

In this case, a 2.5D shape with a hole at the center is designed. There are 350 layers when the layer thickness is set to 0.1 mm. Effective 2.5D encryption is expected for grid-type structures.

Table 4.11: The results for the case of sphere with 441 Voronoi cells.

Algorithm	**Time [s]**	**Size [KB]**			**Fast Travels**	**U Turns**
		G-code	**corners**	**paths**		
Naive	245.852	16737			88536	34
Improved Naive	1010.237	16043			62768	9021
Hierholzer's	383.372	13157			398	439
Modified Hierholzer's	392.131	13157			398	0
Yaman et. al.	1063.742	13157			398	60067
Proposed (w/o 2.5D Enc.)	252.690		1620	1759	398	0
Proposed (w/ 2.5D Enc.)	252.690		1620	1759	398	0
Ultimaker Cura	5.260	8573			13514	0

4.5.3.1 Custom Shape with Randomly Placed 100 3D Voronoi Cells

In this case, 100 randomly placed 3D Voronoi cells are generated inside the custom shape (Figure 4.16). The results are given in Table 4.12. Without 2.5D encryption, the sum of *corners.txt* and *paths.txt* becomes 790 KB, which enables 71.9% reduction compared to the G-code file with a size of 2810 KB. With 2.5D encryption, the reduction becomes 76.2%. Proposed algorithm takes 2.4% less time compared to the Modified Hierholzer's algorithm.

4.5.3.2 Custom Shape with Randomly Placed 400 3D Voronoi Cells

In this case, 400 randomly placed 3D Voronoi cells are generated inside the custom shape (Figure 4.17). The results are given in Table 4.13. Without 2.5D encryption, the sum of *corners.txt* and *paths.txt* becomes 1478 KB, which enables 71.4% reduction compared to the G-code file with a size of 5161 KB. With 2.5D encryption, the reduction becomes 71.8%. Proposed algorithm takes 5.7% more time compared to the Modified Hierholzer's algorithm.

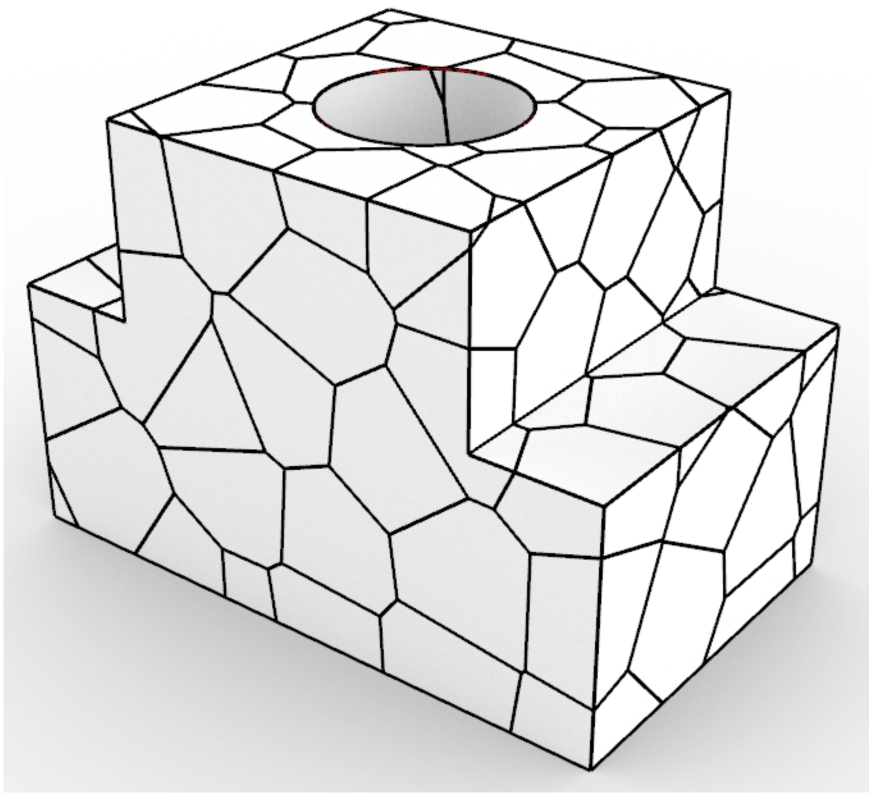


Figure 4.16: Custom shape with 100 randomly placed 3D Voronoi cells.

Table 4.12: The results for the case of custom shape with 100 Voronoi cells.

Algorithm	**Time [s]**	**Size [KB]**			**Fast Travels**	**U Turns**
		G-code	**corners**	**paths**		
Naive	49.489	3066			10019	1
Improved Naive	64.718	3052			9643	138
Hierholzer's	43.820	2810			350	397
Modified Hierholzer's	44.338	2810			350	0
Yaman et. al.	61.602	2810			350	6715
Proposed (w/o 2.5D Enc.)	45.385		507	283	350	0
Proposed (w/ 2.5D Enc.)	45.385		430	240	350	0
Ultimaker Cura	NA					

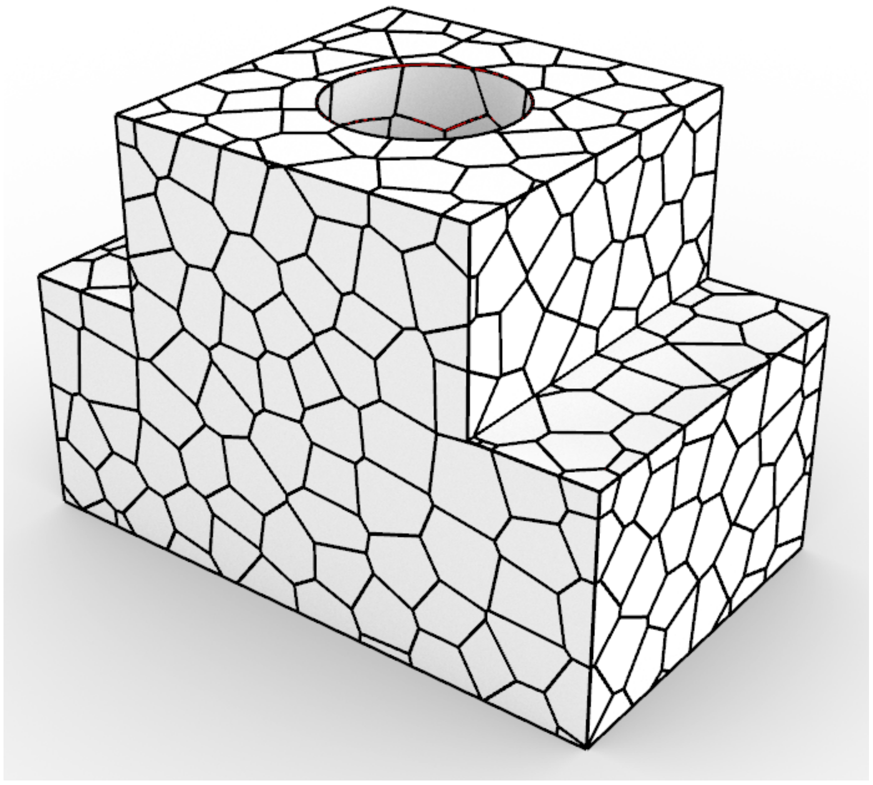


Figure 4.17: Custom shape with 400 randomly placed 3D Voronoi cells.

Table 4.13: The results for the case of custom shape with 400 Voronoi cells.

Algorithm	Time [s]	Size [KB]			Fast Travels	U Turns
		G-code	corners	paths		
Naive	81.175	5835			23281	0
Improved Naive	132.205	5835			22890	144
Hierholzer's	94.478	5161			350	1155
Modified Hierholzer's	97.060	5161			350	0
Yaman et. al.	164.759	5161			350	14227
Proposed (w/o 2.5D Enc.)	102.572		898	580	350	0
Proposed (w/ 2.5D Enc.)	102.572		884	572	350	0
Ultimaker Cura	NA					

4.5.3.3 Custom Shape with 104 Square Voronoi Cells on XY-Plane

In this case, a square grid structure is created, which is equivalent to 104 Voronoi cells at each layer (Figure 4.18). Note that, not all of the grid cells intersect the object at each layer. The results are given in Table 4.14. Without 2.5D encryption, the sum of *corners.txt* and *paths.txt* becomes 1136 KB, which enables 75.3% reduction compared to the G-code file with a size of 4602 KB. With 2.5D encryption, the reduction becomes 77.4%. This result is surprising because the shape can be divided into two 2.5D shapes. Thus, a reduction ratio around 99% would be expected, similar to the cube. The problem is due to the intersection algorithm. The edges of the circle are obtained at different coordinates throughout the shape for most of the time. Therefore, since the corners defining the circle are not at the same coordinates, the algorithm does not regard this shape as a 2.5D object, and consequently 2.5D encryption cannot be applied. Proposed algorithm takes 27.3% less time compared to the Modified Hierholzer's algorithm.

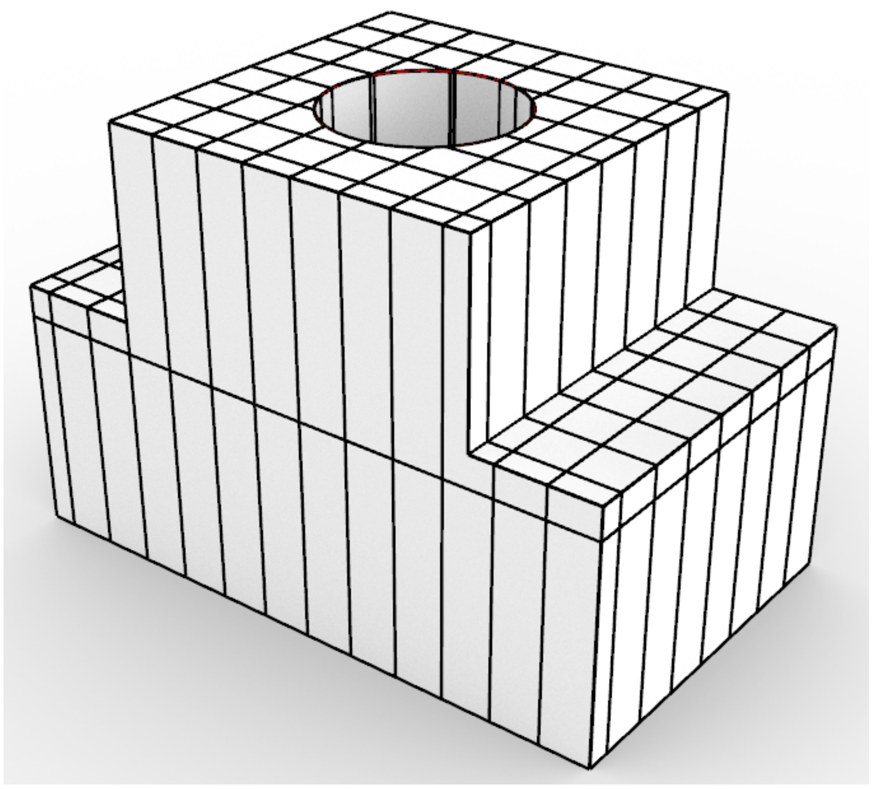


Figure 4.18: Custom shape with 104 Voronoi cells at each layer.

Table 4.14: The results for the case of custom shape with 104 Voronoi cells.

Algorithm	**Time [s]**	**Size [KB]**			**Fast Travels**	**U Turns**
		G-code	**corners**	**paths**		
Naive	68.431	5450			27009	4
Improved Naive	108.465	5068			18915	2661
Hierholzer's	73.430	4602			350	324
Modified Hierholzer's	77.450	4602			350	0
Yaman et. al.	148.361	4602			350	18342
Proposed (w/o 2.5D Enc.)	56.299		595	544	350	0
Proposed (w/ 2.5D Enc.)	56.299		542	498	350	0
Ultimaker Cura	3.210	4974			13839	0

4.5.3.4 Custom Shape with 416 Square Voronoi Cells on XY-Plane

In this case, a square grid structure is created, which is equivalent to 416 Voronoi cells at each layer (Figure 4.19). Note that, not all of the grid cells intersect the object at each layer. The results are given in Table 4.15. Without 2.5D encryption, the sum of *corners.txt* and *paths.txt* becomes 3395 KB, which enables 75.8% reduction compared to the G-code with 14033 KB. With 2.5D encryption, the reduction becomes 76.4%. Proposed algorithm takes 50.9% less time compared to the Modified Hierholzer's algorithm.

4.5.4 Stanford Bunny

In this case, a well known mesh structure, Stanford Bunny is utilized. There are 836 layers when the layer thickness is set to 0.1 mm. The boundary of the Bunny changes continuously over the layers. Therefore, it is not suitable for 2.5D encryption.

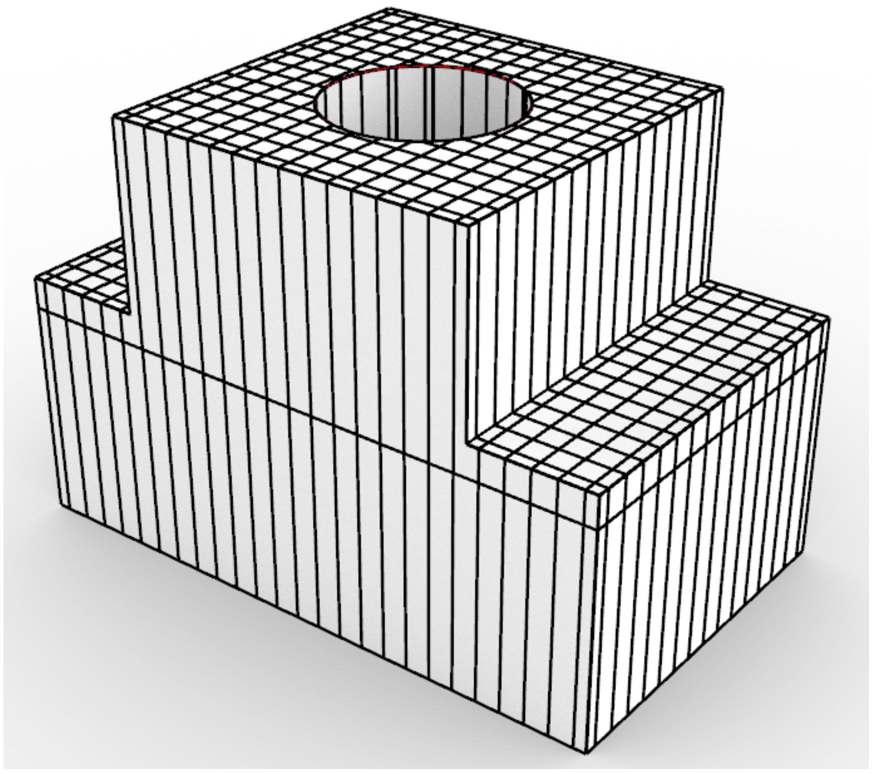


Figure 4.19: Custom shape with 416 Voronoi cells at each layer.

Table 4.15: The results for the case of custom shape with 416 Voronoi cells.

Algorithm	**Time [s]**	**Size [KB]**			**Fast Travels**	**U Turns**
		G-code	**corners**	**paths**		
Naive	176.017	17719			98799	11
Improved Naive	556.065	16812			79443	8592
Hierholzer's	391.304	14033			350	1418
Modified Hierholzer's	396.337	14033			350	0
Yaman et. al.	1266.429	14033			350	67484
Proposed (w/o 2.5D Enc.)	194.781		1499	1896	350	0
Proposed (w/ 2.5D Enc.)	194.781		1462	1851	350	0
Ultimaker Cura	3.680	6568			13009	0

4.5.4.1 Stanford Bunny with Randomly Placed 100 3D Voronoi Cells

In this case, 100 randomly placed 3D Voronoi cells are generated inside the bounding box of the Stanford Bunny, then their intersection is taken (Figure 4.20). The results are given in Table 4.16. Without 2.5D encryption, the sum of *corners.txt* and *paths.txt* becomes 11021 KB, which enables 66.7% reduction compared to the G-code file with a size of 33027 KB. For this case, no 2.5D encryption can be applied. Proposed algorithm takes 25.4% less time compared to the Modified Hierholzer's algorithm.

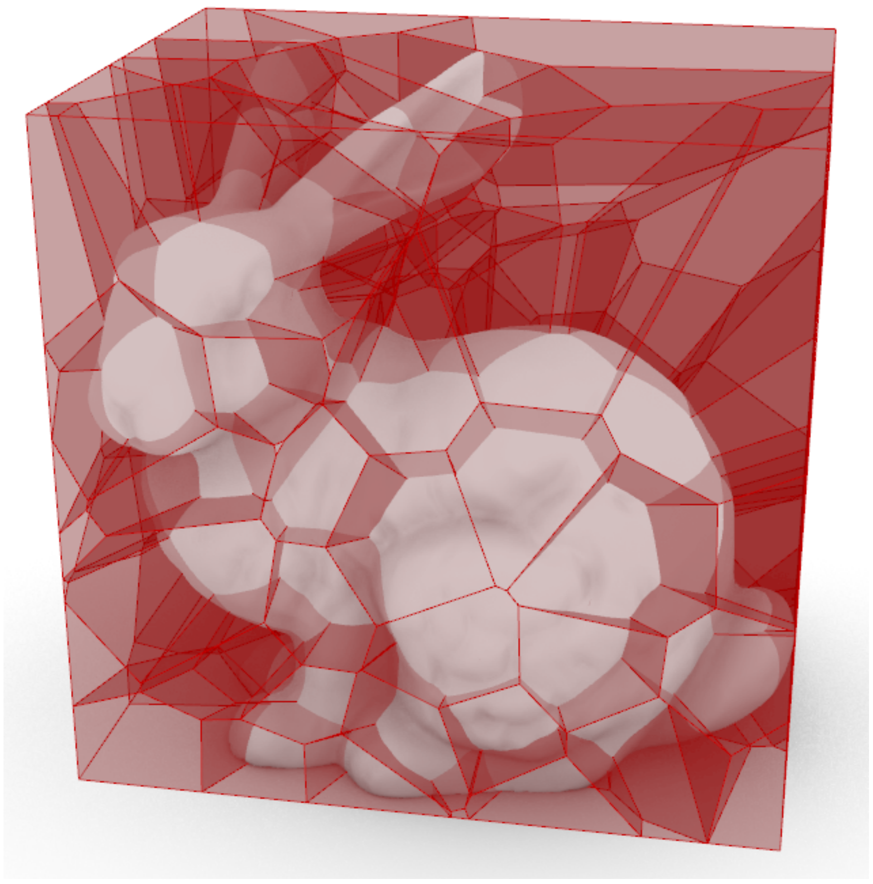


Figure 4.20: Stanford Bunny with 100 randomly placed 3D Voronoi cells.

4.5.4.2 Stanford Bunny with Randomly Placed 400 3D Voronoi Cells

In this case, 400 randomly placed 3D Voronoi cells are generated inside the bounding box of the Stanford Bunny, then their intersection is taken (Figure 4.21). The results are given in Table 4.17. Without 2.5D encryption, the sum of *corners.txt* and *paths.txt*

Table 4.16: The results for the case of Stanford Bunny with 100 Voronoi cells.

Algorithm	**Time [s]**	**Size [KB]**			**Fast Travels**	**U Turns**
		G-code	**corners**	**paths**		
Naive	202.507	33987			16334	17
Improved Naive	990.055	33933			15314	864
Hierholzer's	571.896	33027			1372	265
Modified Hierholzer's	577.707	33027			1372	0
Yaman et. al.	1235.800	33027			1372	12766
Proposed (w/o 2.5D Enc.)	724.346		7157	3864	1372	0
Proposed (w/ 2.5D Enc.)	724.346		7157	3864	1372	0
Ultimaker Cura	NA					

becomes 12323 KB, which enables 67.8% reduction compared to the G-code with 38275 KB. For this case, no 2.5D encryption can be applied. Proposed algorithm takes 33.3% less time compared to the Modified Hierholzer's algorithm.

Table 4.17: The results for the case of Stanford Bunny with 400 Voronoi cells.

Algorithm	**Time [s]**	**Size [KB]**			**Fast Travels**	**U Turns**
		G-code	**corners**	**paths**		
Naive	588.145	40025			41253	53
Improved Naive	1570.140	39931			39517	1227
Hierholzer's	1060.214	38275			1372	1184
Modified Hierholzer's	1062.393	38275			1372	0
Yaman et. al.	1235.800	38275			1372	29275
Proposed (w/o 2.5D Enc.)	1415.850		7869	4454	1372	0
Proposed (w/ 2.5D Enc.)	1415.850		7869	4454	1372	0
Ultimaker Cura	NA					

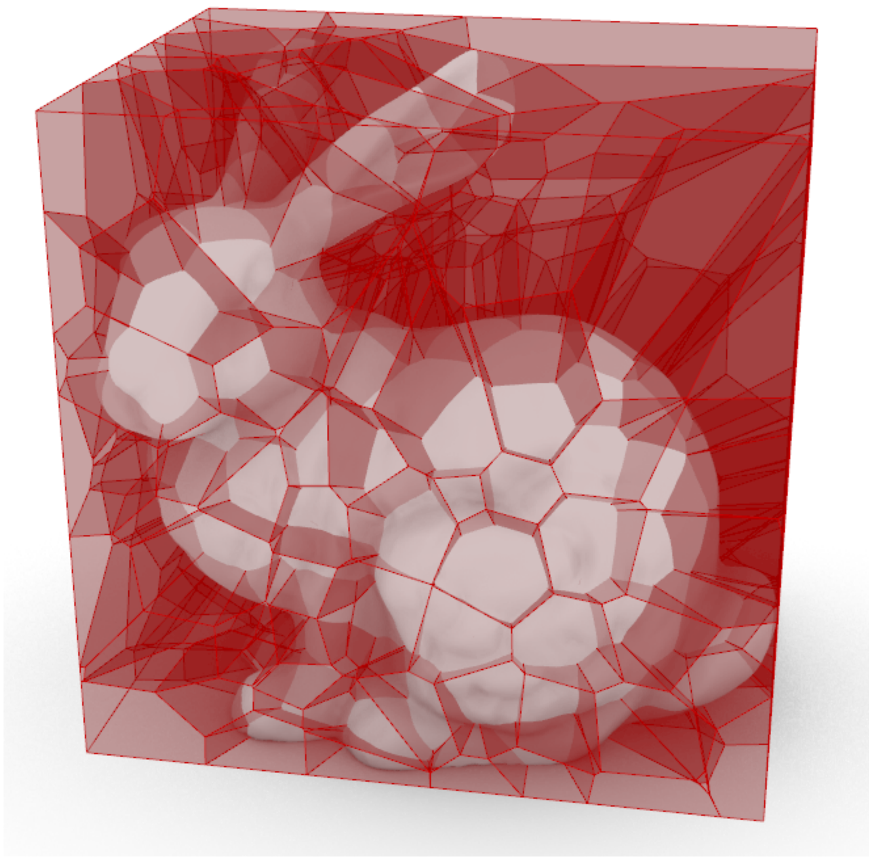


Figure 4.21: Stanford Bunny with 400 randomly placed 3D Voronoi cells.

4.5.4.3 Stanford Bunny with 80 Square Voronoi Cells on XY-Plane

In this case, a square grid structure is created, which is equivalent to 80 Voronoi cells at each layer (Figure 4.22). For a better understanding, section view of the object is also given (Figure 4.23). Note that, not all of the grid cells intersect the object at each layer. The results are given in Table 4.18. Without 2.5D encryption, the sum of *corners.txt* and *paths.txt* becomes 11111 KB, which enables 67.1% reduction compared to the G-code file with a size of 33785 KB. For this case, no 2.5D encryption can be applied. Proposed algorithm takes 37.0% more time compared to the Modified Hierholzer's algorithm.

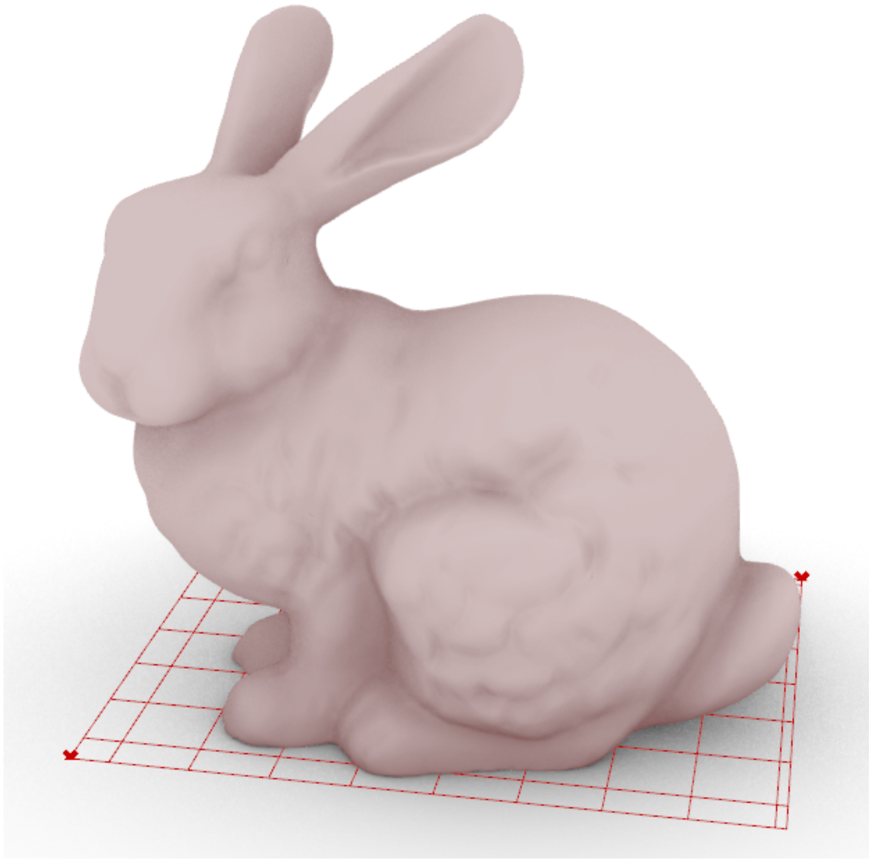


Figure 4.22: Stanford Bunny with 80 Voronoi cells at each layer.

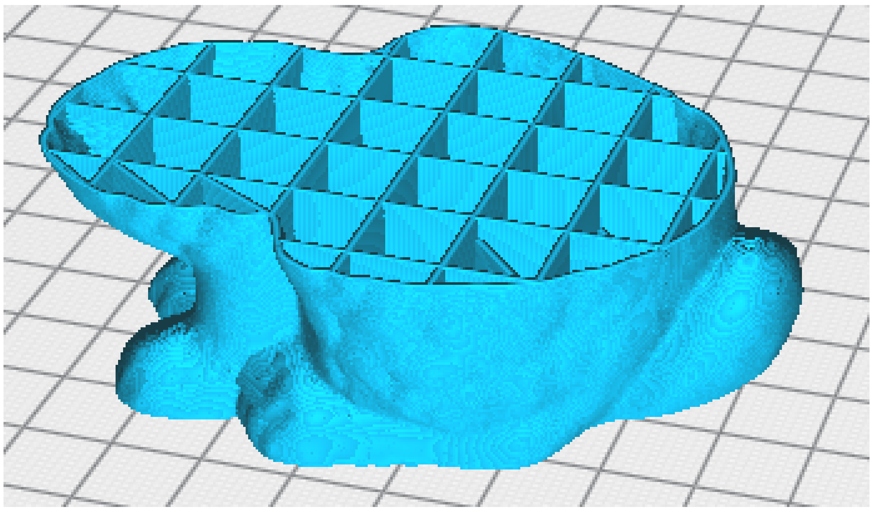


Figure 4.23: Section view of the Stanford Bunny with 80 Voronoi cells at each layer.

Table 4.18: The results for the case of Stanford Bunny with 80 Voronoi cells.

Algorithm	**Time [s]**	**Size [KB]**			**Fast Travels**	**U Turns**
		G-code	**corners**	**paths**		
Naive	86.745	34684			19331	22
Improved Naive	901.550	34592			17608	2754
Hierholzer's	471.555	33785			1372	487
Modified Hierholzer's	489.035	33785			1372	0
Yaman et. al.	1243.321	33785			1372	28500
Proposed (w/o 2.5D Enc.)	670.110		7174	3937	1372	0
Proposed (w/ 2.5D Enc.)	670.110		7174	3937	1372	0
Ultimaker Cura	9.200	27879			47404	0

4.5.4.4 Stanford Bunny with 414 Square Voronoi Cells on XY-Plane

In this case, a square grid structure is created, which is equivalent to 411 Voronoi cells at each layer (Figure 4.24). For a better understanding, section view of the object is also given (Figure 4.25). Note that, not all of the grid cells intersect the object at each layer. The results are given in Table 4.19. Without 2.5D encryption, the sum of *corners.txt* and *paths.txt* becomes 13784 KB, which enables 69.7% reduction compared to the G-code file with a size of 45468 KB. For this case, no 2.5D encryption can be applied. Proposed algorithm takes 19.4% more time compared to the Modified Hierholzer's algorithm.

4.6 Results

Among all algorithms, the naive algorithm has the highest number of *fast travels* as it covers all Voronoi cells one by one. The algorithm of Yaman et al. [1] has the highest number of *u turns*, because it goes to the neighbor of a Voronoi cell after traversing an edge of it. The improved naive algorithm has slightly better performance than the naive algorithm since it has fewer *fast travels*.

Proposed algorithm, Hierholzer's and Modified Hierholzer's algorithms, and algo-

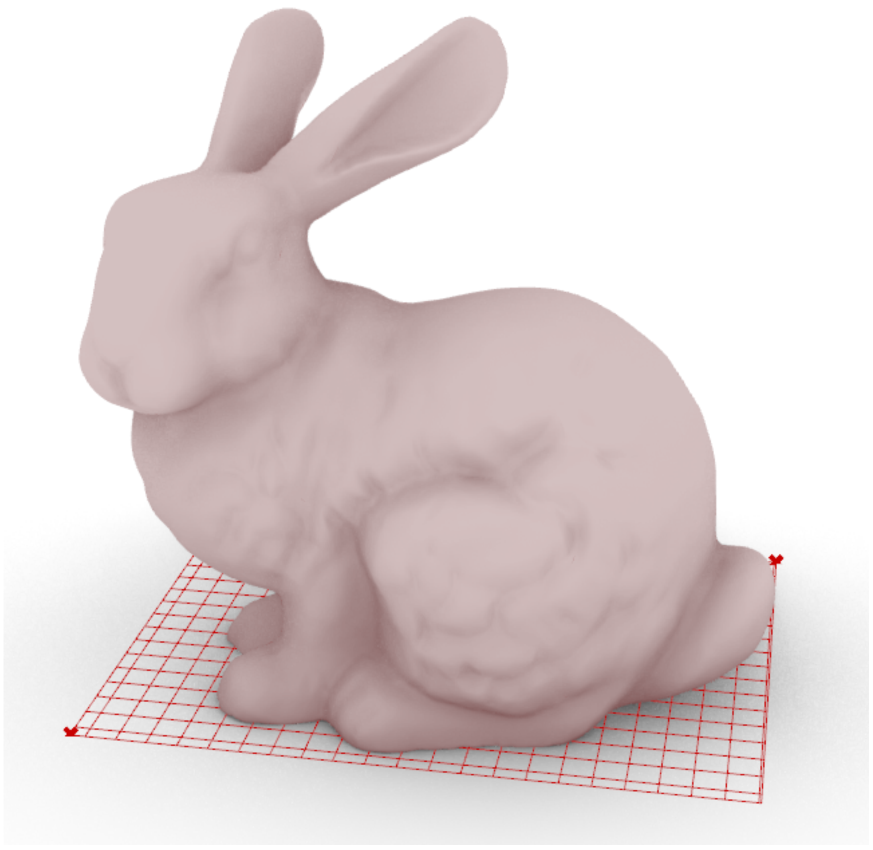


Figure 4.24: Stanford Bunny with 414 Voronoi cells at each layer.

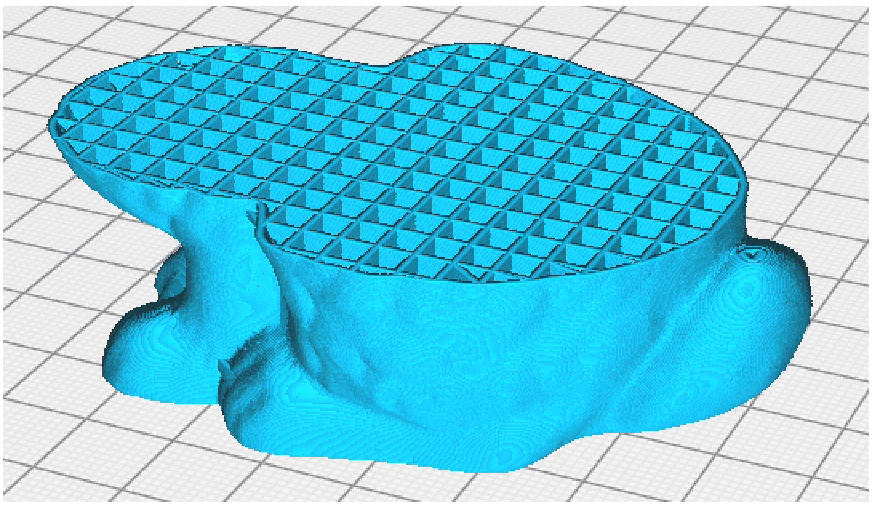


Figure 4.25: Section view of the Stanford Bunny with 414 Voronoi cells at each layer.

rithm of Yaman et al. [1] make no *fast travels*, and the Hierholzer's algoritm can make *u turns* occasionally. Modified Hierholzer's algorithm and the proposed algo-

Table 4.19: The results for the case of Stanford Bunny with 414 Voronoi cells.

Algorithm	**Time [s]**	**Size [KB]**			**Fast Travels**	**U Turns**
		G-code	**corners**	**paths**		
Naive	305.427	48954			91728	48
Improved Naive	1733.129	48078			74502	14528
Hierholzer's	1045.800	45468			1372	1076
Modified Hierholzer's	1128.900	45468			1372	0
Yaman et. al.	2590.571	45468			1372	116926
Proposed (w/o 2.5D Enc.)	1347.482		8485	5299	1372	0
Proposed (w/ 2.5D Enc.)	1347.482		8485	5299	1372	0
Ultimaker Cura	9.350	30196			49262	0

rithm tackles this problem and make no *u turns*. Whenever possible, the algorithm of Ultimaker Cura is also utilized. As observed, it makes no *u turns*, but it does have many *fast travels*.

The proposed algorithm updates the path at subsequent steps, and the Modified Hierholzer's algorithm, which is also developed by us, computes the path at each layer. If the structure has grids, i.e., if the path at subsequent layers are similar, the algorithm usually has a time advantage compared to the Modified Hierholzer's algorithm. Only in STL files, like Stanford Bunny, the algorithm takes a longer time. Nonetheless, as the density of the grid is increased, the performance of the algorithm becomes better. Encryption and path update approaches are very advantageous when the part is a 2.5D object. Reduction rate of 99.7% in size compared to the G-code file, and reduction rate of 58.6% in time compared to the Modified Hierholzer's algorithm is achieved. Overall, it is believed that the proposed algorithm must be used when one wants no *fast travels* and *u turns*, and if the object is mostly 2.5D. For 3D objects, the Modified Hierholzer's algorithm can be considered. Nonetheless, as seen in the test cases, presented encryption method can be utilized in any scenario, as it results in around 70% reduction at least.

Although Ultimaker Cura's algorithm does not eliminate *fast travels*, it works very fast. Note that, objects with randomly placed Voronoi cells cannot be printed by that algorithm.

4.7 Adaptations for the 3D Printed Parts

In this study, solid objects are filled with 3D Voronoi cells, which might be ordered or randomly distributed. Then, the resulting model is sliced, and a 2D Voronoi graph is obtained for each layer. Finally, different algorithms are utilized to traverse the resulting graph. Although this approach is sufficient for comparing the algorithms, some adaptations are needed while printing the parts in order to make it more intact and durable. Those enhancements are discussed in this section.

4.7.1 Path Optimization

4.7.1.1 On a Layer

Since developed graphs are Eulerian, the path ends at the same corner it starts if the graph is connected. However, sometimes several distinct graphs may be obtained as in the case of the ears of the Stanford Bunny. In such a case, after a graph is fully traversed, the closest corner among the remaining graphs is computed, and starting from the closest corner, that graph is traversed. This process continues until all of the graphs are traversed. By this method, it is aimed to minimize the fast travels on the same layer.

4.7.1.2 Between the Layers

After a layer is completely covered, the printer head moves in vertical direction for the layer thickness, which is typically $0.1\ mm$. Then, the closest corner of the current layer's graph is determined and the current path is started from that corner. For grid-type structures, corners from the previous and current layers' Voronoi cells lie at the same positions, excluding boundary cells. On the other hand, for randomly placed Voronoi cells, positions of the corners change continuously. By computing the closest corner and starting the traversal from that position, the fast travels between the layers are minimized. If this optimization is skipped, traces caused by fast travels become highly visible. For instance, instead of starting traversal from the closest corner, a random corner might be chosen, which would result in many fast travel lines that don't fit with Voronoi structure (Figure 4.26).

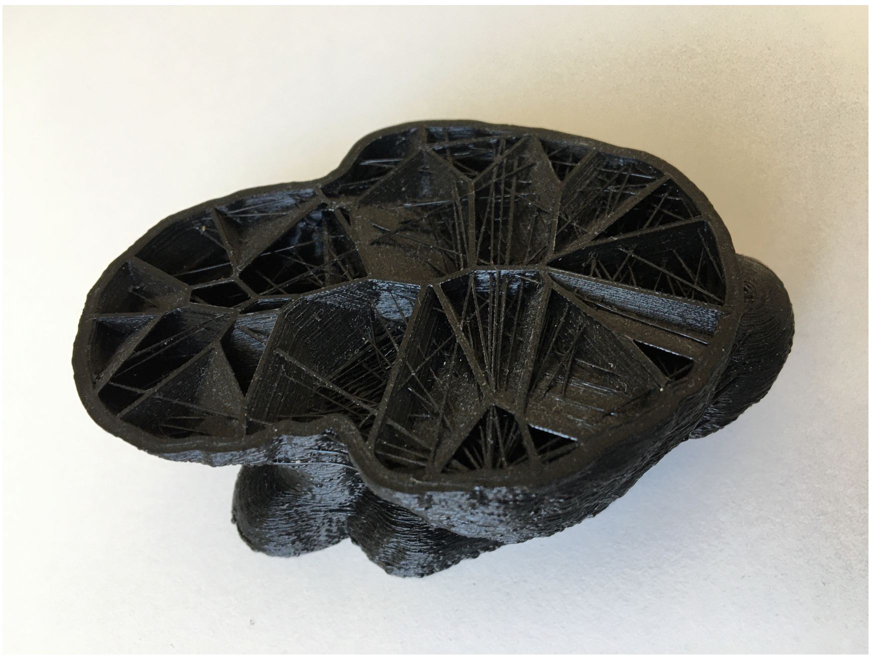


Figure 4.26: If the path is not optimized, many fast travels occur.

4.7.2 Solid Base Creation

A geometry can be filled with any specified number of Voronoi cells. The greater the number of the cells, the denser the structure becomes. Therefore, to create a lighter part, one should choose a lower number of cells. Nonetheless, to make the part stable, it should be completely covered for the first 4-5 layers, i.e., a very dense Voronoi structure must be implemented. Then, the desired density can be applied.

4.7.3 Shell Generation

In order to make the boundary of the printed part stronger, some shells are generated, for which 1 or 2 layers are found to be sufficient, which are $0.5\ mm$ apart from each other.

4.8 3D Printed Parts

Several parts are 3D printed for experimenting with the algorithms and tuning the parameters. At first, the path optimization is observed to be necessary, which is explained in Section 4.7.1. Fast travel lines are seen to be excessive (Figure 4.26). Therefore, path optimization is applied, and the regarding problem is eliminated.

Another problem encountered is the traces of the fast travels on a layer. While printing the ears of the Stanford Bunny, printer head is guided from one ear to the other, during which a fast travel is inevitable. Nonetheless, checking the finished part, it can be seen that fast travel lines look like solid lines (Figure 4.27). Although it is not very difficult to remove those portions (Figure 4.28), the aim is to avoid them as much as possible. The reason behind the lines is the feedrate of the printer head for fast travels, i.e., *G0* motion. By increasing the feedrate from $4257.3\ mm/s$ to $10000\ mm/s$, most of those undesired lines are avoided.

Using transparent type of filament, the Stanford Bunny is once more fabricated. This time, the effect of the number of the shells is tested. The same model is first printed with 2 shells (Figure 4.29), and then with a single shell (Figure 4.30). As one can observe, Voronoi cells are much more visible when only one shell is generated. For a better comparison, a figure with both models is also provided (Figure 4.31).

Another 3D printed part, for which two filaments are sequentially used, is the cube that is filled by 36 Voronoi cells (Figure 4.32). Of those cells, 9 are desired to be filled completely with black filament. On the other hand, only the boundaries of the remaining 27 cells are filled. Note that, there exists a slight shift around the middle of the cube. Since it takes a long time to print the part, printer is turned off at some point, and then turned on again the following day, which caused aforementioned slight offset.

4.9 Surface Roughness Tests

For a better comparison between the algorithms, surface roughness tests are conducted. In the first test, a $100x100\ mm$ area is divided into 5383 triangular Voronoi

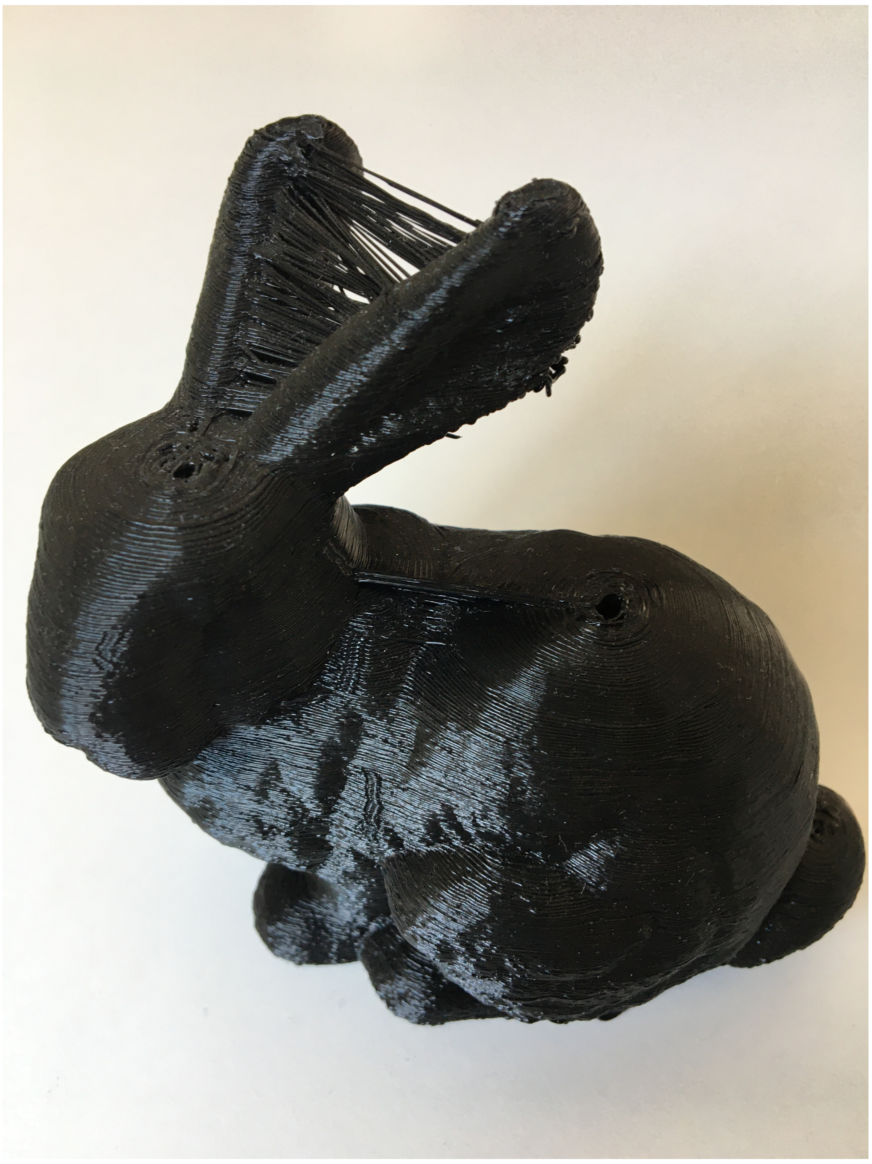


Figure 4.27: When feedrate of the printer head is not high enough in *G0* motion, fast travels do result in solid lines.

cells, so that the area will be fully covered. Then, different algorithms are used for coverage. Designed model and the resulting parts can be seen in Figure 4.33. Roughness profiles for those geometries are measured using the profilometer MarSurf PS 10, which has a maximum measuring range from -0.2 to $0.15\ mm$, in four directions, perpendicular to each edge. Sample profiles obtained among the drawn lines are given in Figure 4.34. Average *Ra* and *Peak-to-Valley* values are given in Table 4.20.

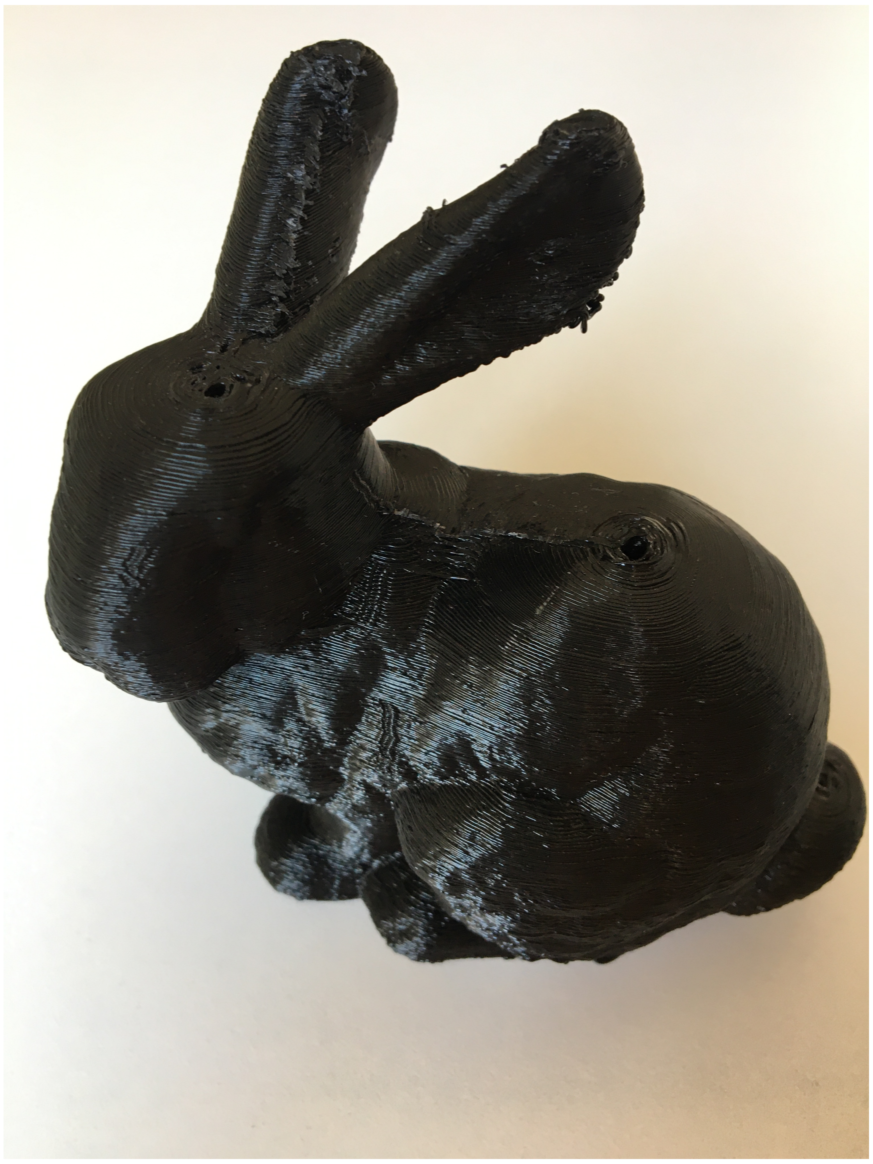


Figure 4.28: Bunny after the excessive material caused by insufficient fast travel feedrate is removed.

In another test, a box that has a $30x30$ mm base and 20 mm height is filled with 36 Voronoi cells. Then, each algorithm is used to cover it. The resulting geometries can be seen in Figure 4.35. Among the 4 surfaces of the box that are in the height direction, roughness profiles are measured, twice at each surface (Figure 4.36). Sample profiles are given in Figure 4.37. Average *Ra* and *Peak-to-Valley* values are given in Table 4.21.

Checking the results, it is seen that the best performances are those obtained by Hi-

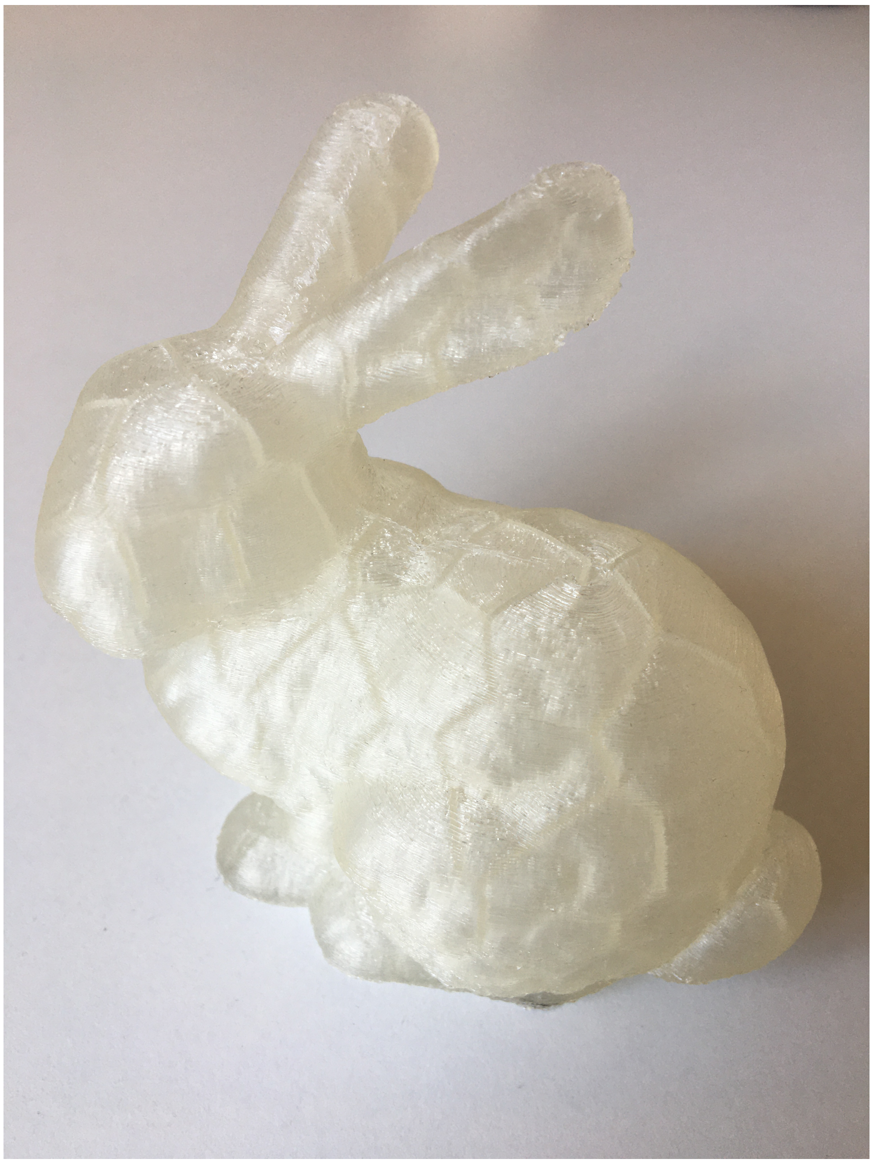


Figure 4.29: Bunny with 100 Voronois and 2 layers of shell, 3D printed with transparent filament.

erholzer's and Modified Hierholzer's algorithms. They are better than the algorithms Naive, Improved Naive and Yaman et al. [1] in the second case. In the first case, Hierholzer's, Modified Hierholzer's and Yaman et al. [1] show similar performances, which are better than those of the Naive and Improved Naive algorithms. Furthermore, only continuous coverage algorithms are able to print the part correctly, thanks to the elimination of fast travels (Figure 4.35).

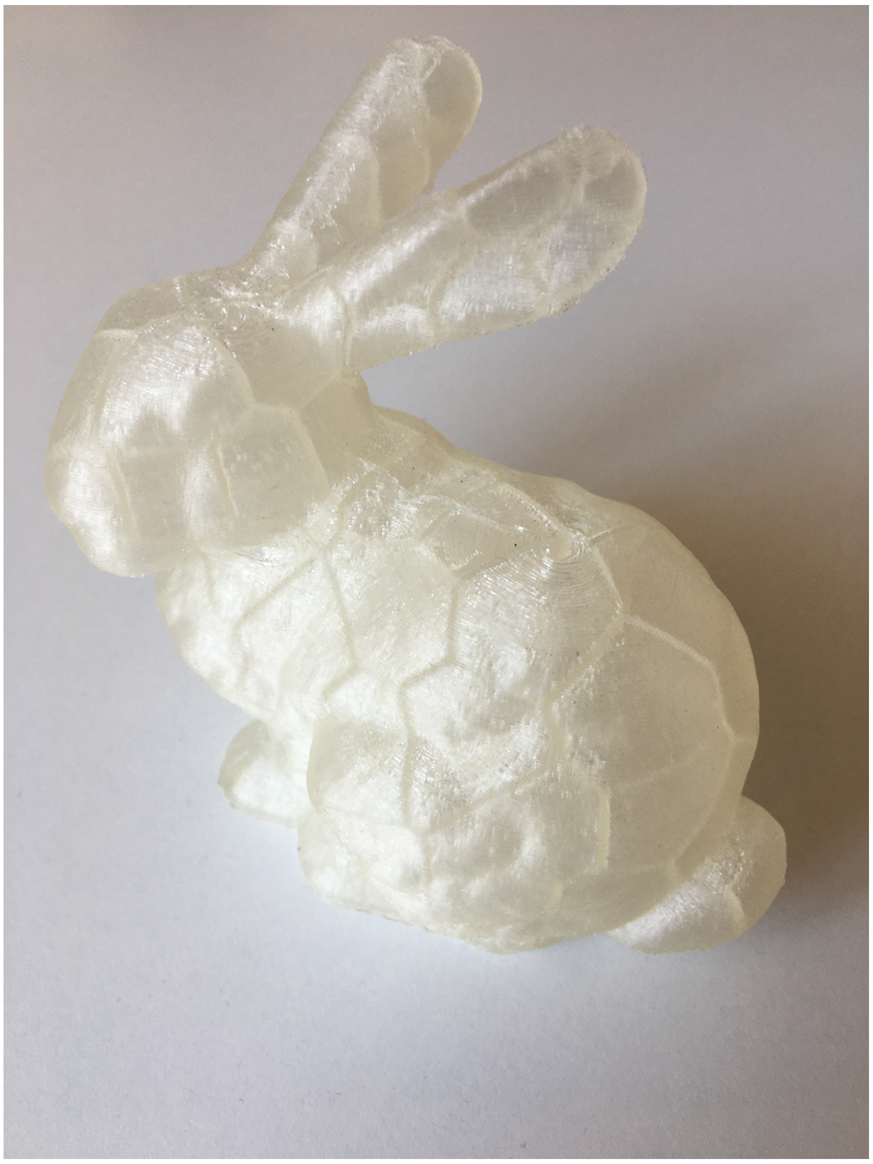


Figure 4.30: Bunny with 100 Voronois and 1 layer of shell, 3D printed with transparent filament.

4.10 Intersection Problems due to Grasshopper

For the implementation of the algorithms, Rhino3D and Grasshopper3D are used. Utilizing the GhPython component of Grasshopper3D, algorithms are implemented and G-code files are generated. It works fine for solid shapes, for which the native components and functions are mostly utilized. However, for the STL files and additional capabilities, such as 3D printing with dual extruders, new functionalities are

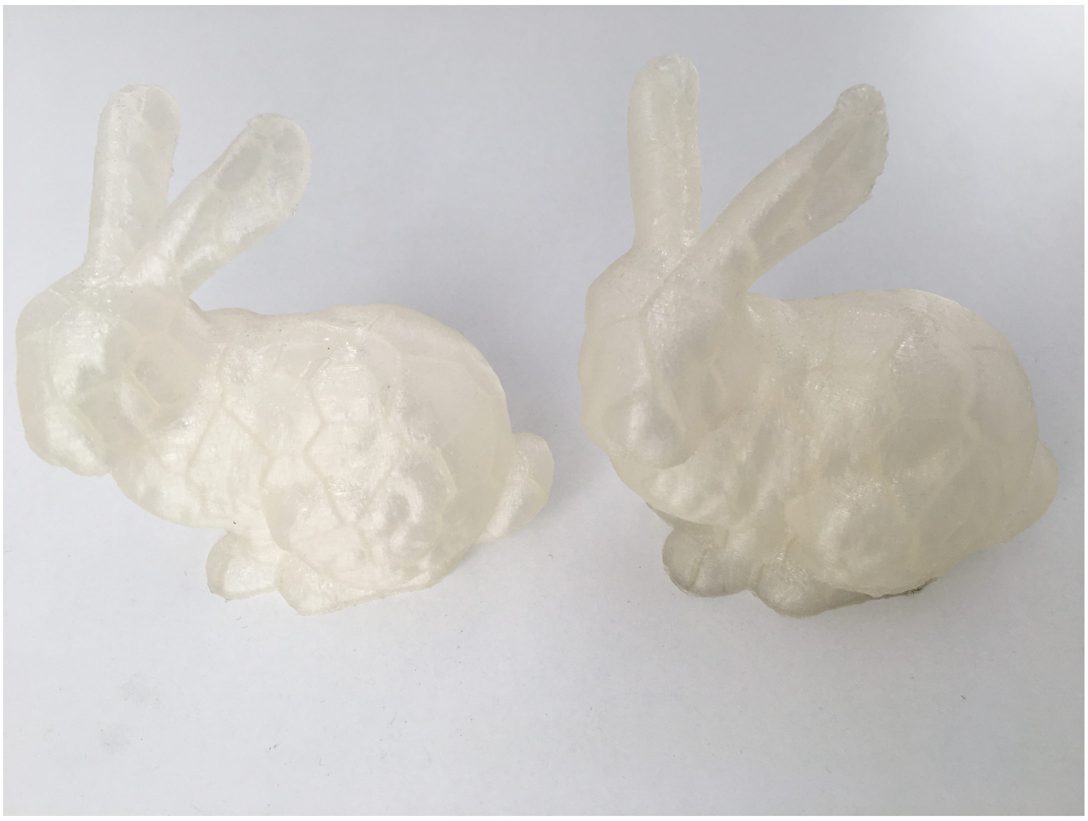


Figure 4.31: Bunny models with 1 layer (left) and 2 layers (right) of shells.

Table 4.20: Average Ra and Peak-to-Valley values of the square objects printed by different algorithms.

Algorithm	**Ra [μm]**	**Peak-to-Valley [μm]**
Naive	50.921	263.325
Improved Naive	56.460	283.803
Yaman et. al.	44.061	264.544
Hierholzer's	46.932	229.919
Modified Hierholzer's	45.520	263.021

developed. Intersection problems sometimes occur in those cases.

While 3D printing with dual extruders, some of the Voronoi cells are set to be filled with a grid structure. In order to do this, those Voronoi cells, which are randomly

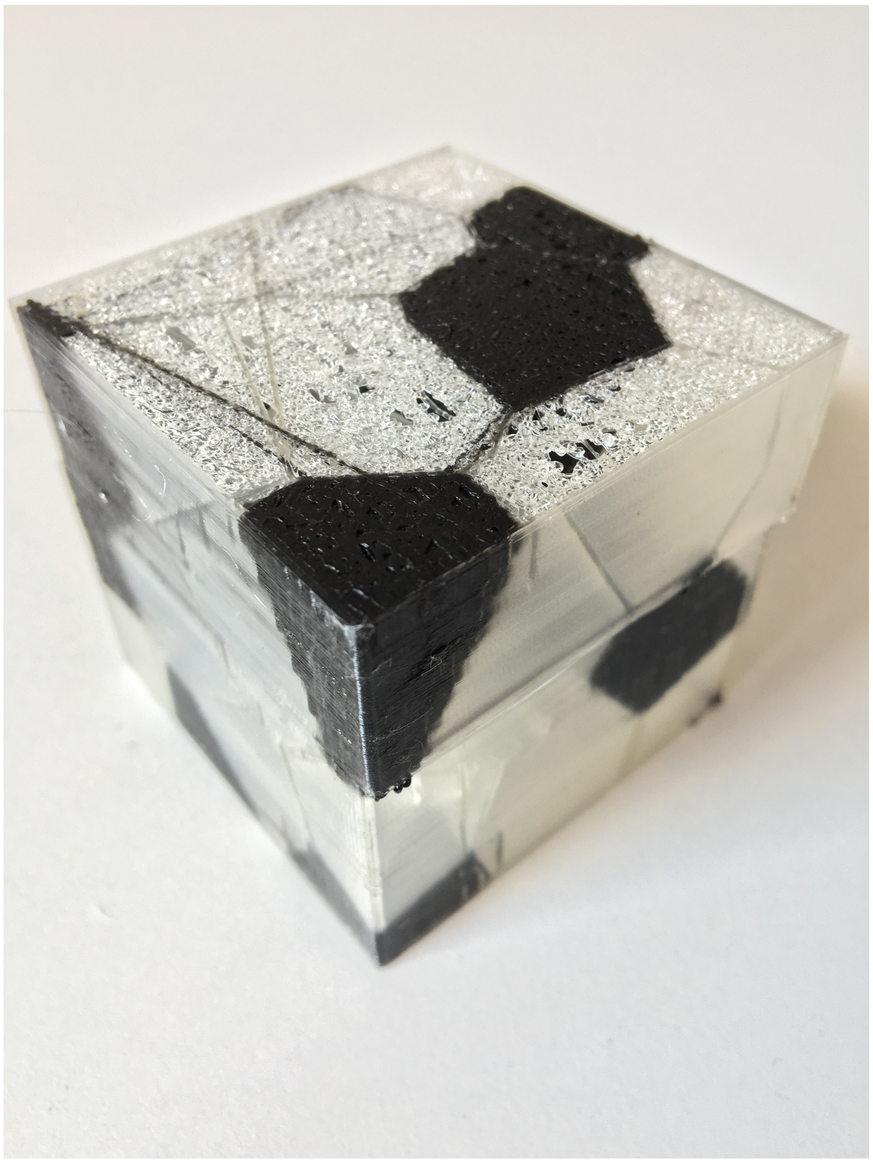
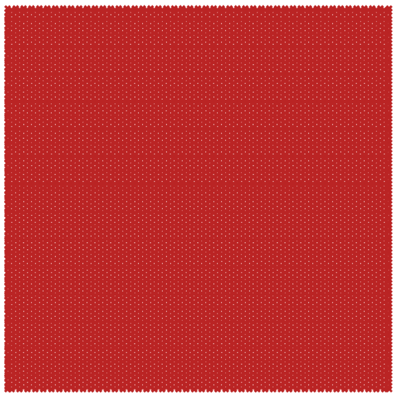
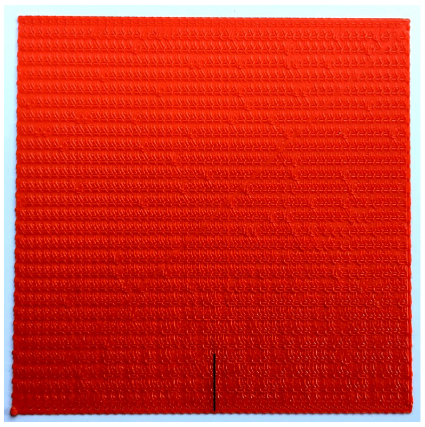


Figure 4.32: The cube with 36 Voronoi cells, 3D printed with two filaments.

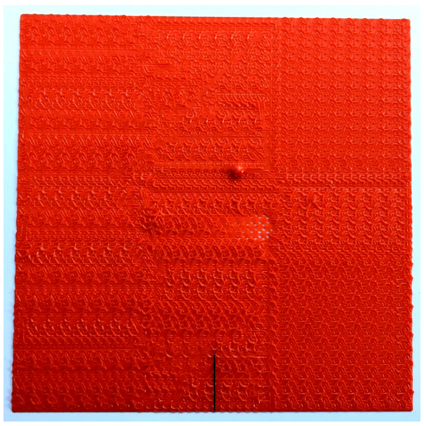
distributed, are intersected with grid-type Voronoi cells. As the problem occurs, parts of those grid cells might become missing (Figure 4.38). The reason behind the errors might be the improper implementations of the functions. Another possibility is that the intersection algorithms in Grasshopper3D might be optimized for BRep geometries. It is not possible to take intersection of two Brep sets in Grasshopper3D without scripting, no native module exists. Therefore, a solution in GhPython module is needed to be implemented, and some errors couldn't be avoided.



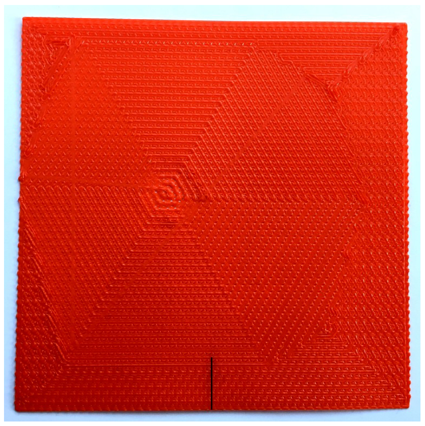
(a) CAD model taken from Rhino



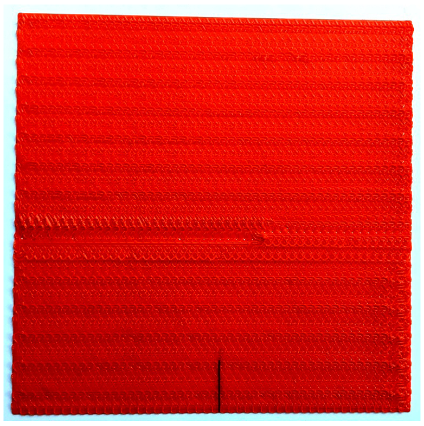
(b) Naive Algorithm



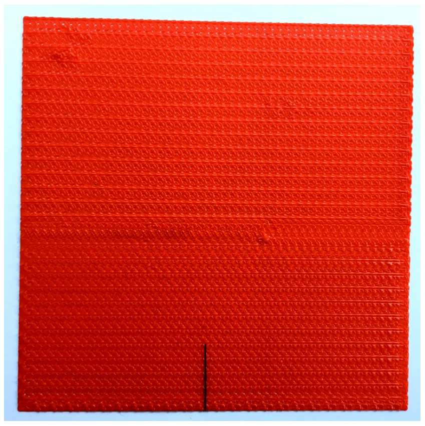
(c) Improved Naive Algorithm



(d) Algorithm of Yaman et al.



(e) Hierholzer's Algorithm



(f) Modified Hierholzer's Algorithm

Figure 4.33: Model file and printed parts for a square with 5383 Voronoi cells. Surface roughness values are obtained along the black lines (Figure 4.34).

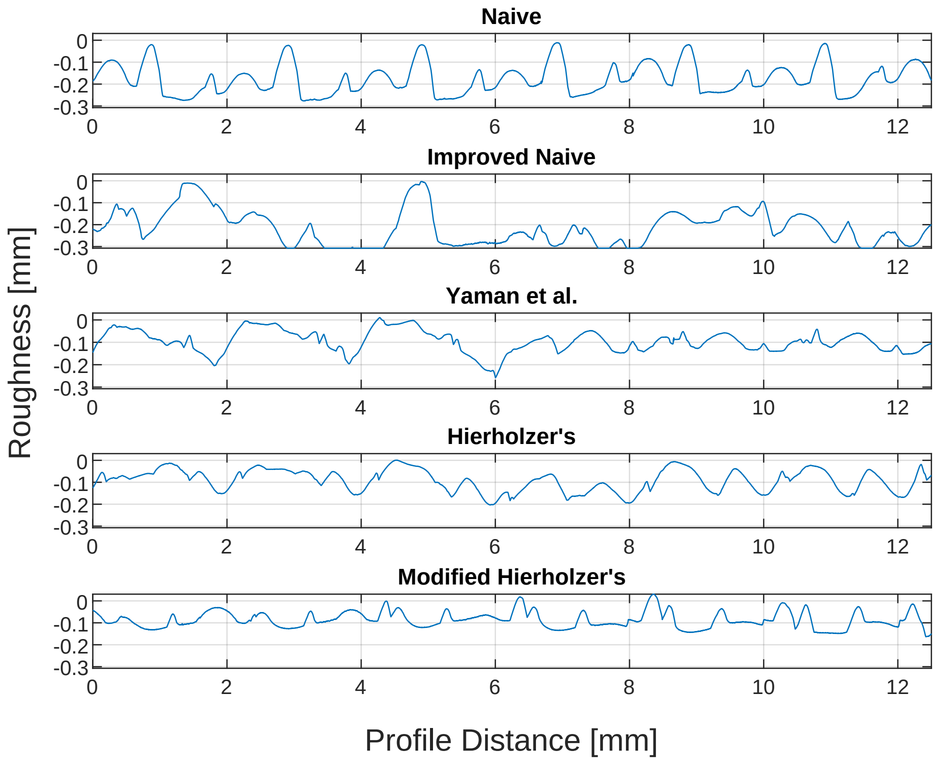


Figure 4.34: Surface Roughness Profiles for the Square Object.

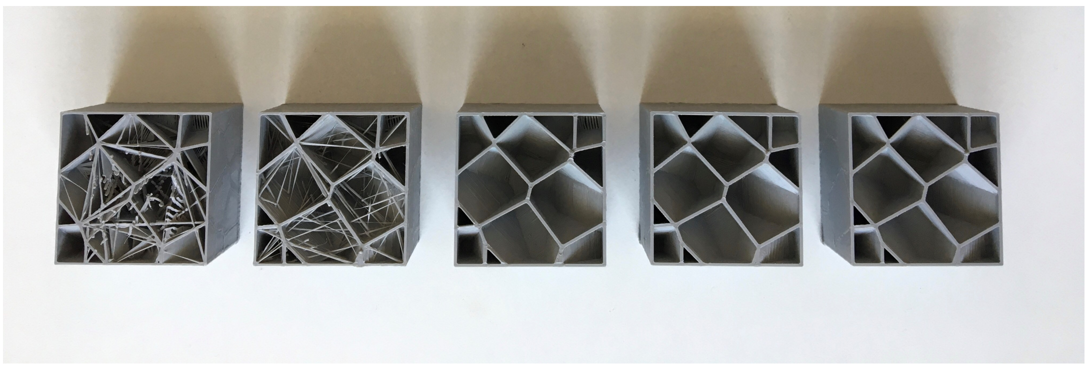


Figure 4.35: 3D printed boxes for all algorithms: From left to right; Naive, Improved Naive, Yaman et al. [1], Hierholzer's, and Modified Hierholzer's. Due to the fast travels, geometric structure can only be preserved using continuous planning algorithms.

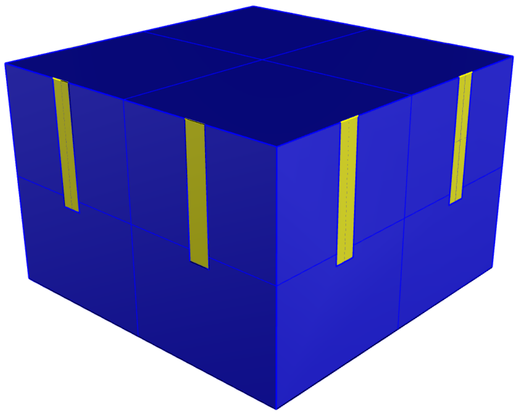


Figure 4.36: Roughness profiles of the box are measured among the surfaces in the height direction, twice at each surface, in the area denoted by yellow.

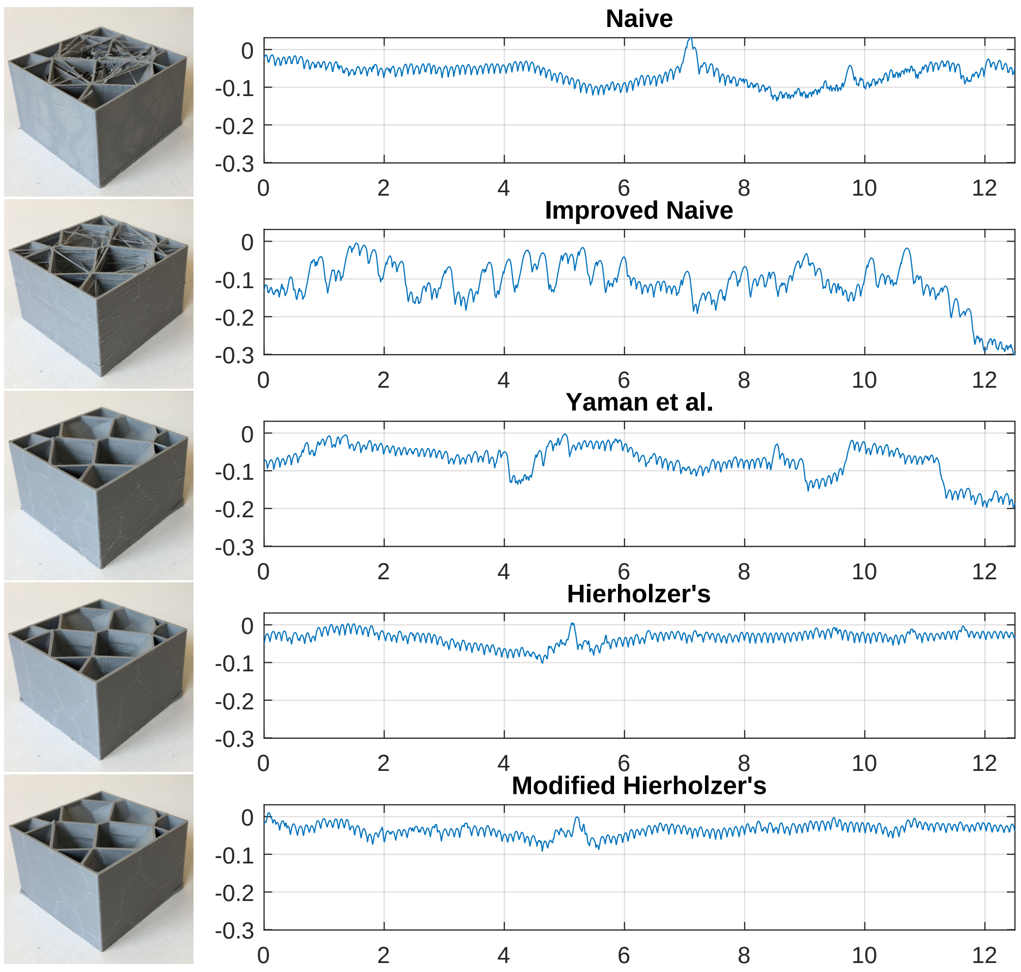


Figure 4.37: 3D printed boxes, and a sample surface roughness profile measured on an edge in the height direction. Both axes are in mm.

Table 4.21: Average Ra and Peak-to-Valley values of the boxes printed by different algorithms.

Algorithm	**Ra [μm]**	**Peak-to-Valley [μm]**
Naive	24.575	189.561
Improved Naive	40.270	278.668
Yaman et. al.	29.904	191.150
Hierholzer's	15.378	106.102
Modified Hierholzer's	14.562	103.356

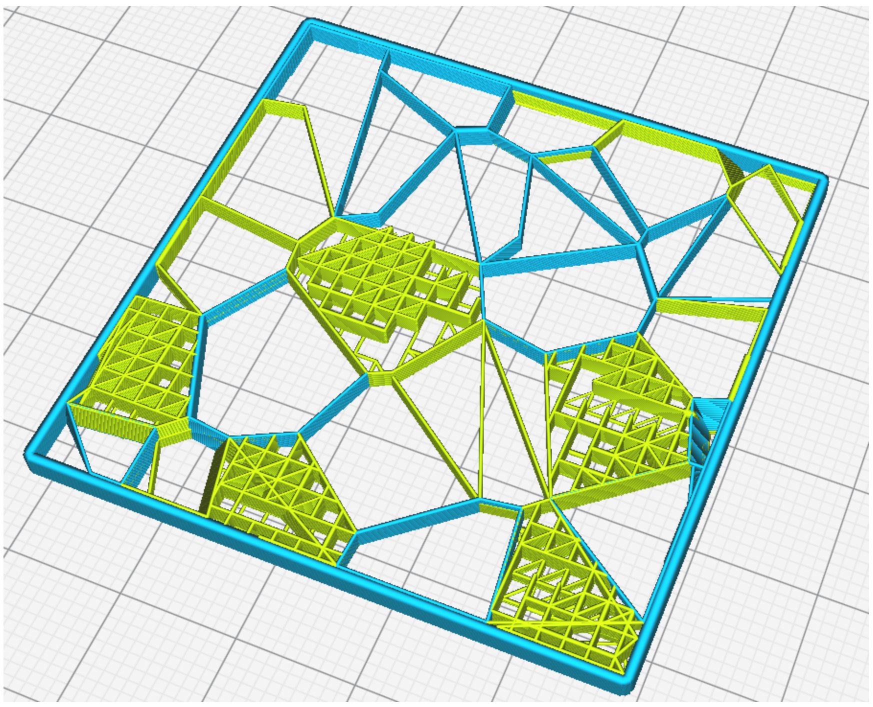


Figure 4.38: Some grid-type Voronoi cells are missing, which is an intersection problem.

CHAPTER 5

DISCUSSIONS & CONCLUSION

In this thesis, a path planning algorithm taking advantage of Voronoi diagrams is studied. Coverage algorithms are and will be used extensively in various applications, like cleaning, farming and monitoring. By designing the Voronoi cell structure, regions with different densities and grid types can be obtained. Depending on the application, a suitable cell structure can be applied and utilized for various applications.

Some applications may require visiting the recently covered areas, whereas some may require the opposite. If *u turns* are required to be eliminated, Modified Hierholzer's algorithm will be a good option. If updating the path is preferred, like in 3D printing, proposed algorithm would be a better option. However, if *u turns* are encouraged, the algorithm by Yaman et al. must be used. All of those algorithms successfully eliminate *fast travels* as they enable continuous traversal by taking advantage of the Eulerian graphs. Additionally, an encryption method for manufacturing applications is presented to replace G-code files resulting in considerable decrease in file size.

Considering AM, various tests made for the study show that the proposed algorithm is advantageous for parts with grid structures, and especially preferable if the part is 2.5D. Additionally, the algorithm also offers great flexibility. Shapes with different densities, and with random or ordered cell structures may be generated and fabricated. Since AM is getting more popularity in household usage and industry, presented methods can be quite useful for such applications.

Voronoi diagrams are utilized in many applications, both for 2D and 3D. Although the presented methods are using them, they are applicable to any connected graphs. As robotics and automation advance, so does the need for versatile coverage and path planning algorithms, and the given algorithms are believed to answer some of those needs.

REFERENCES

[1] U. Yaman, N. Butt, E. Sacks, and C. Hoffmann, “Slice coherence in a query-based architecture for 3d heterogeneous printing,” *Computer-Aided Design*, vol. 75-76, pp. 27 – 38, 2016.

[2] K. Erciyes, *Distributed Graph Algorithms for Computer Networks*. Springer Publishing Company, Incorporated, 2013.

[3] M. d. Berg, O. Cheong, M. v. Kreveld, and M. Overmars, *Computational Geometry: Algorithms and Applications*. Santa Clara, CA, USA: Springer-Verlag TELOS, 3rd ed. ed., 2008.

[4] F. Yasutomi, M. Yamada, and K. Tsukamoto, “Cleaning robot control,” in *Proceedings. 1988 IEEE International Conference on Robotics and Automation*, pp. 1839–1841 vol.3, 1988.

[5] P. N. Atkar, A. Greenfield, D. C. Conner, H. Choset, and A. A. Rizzi, “Uniform coverage of automotive surface patches,” *The International Journal of Robotics Research*, vol. 24, no. 11, pp. 883–898, 2005.

[6] S. Hert, S. Tiwari, and V. J. Lumelsky, “A terrain-covering algorithm for an auv,” *Autonomous Robots*, vol. 3, pp. 91–119, 1996.

[7] E. Acar, H. Choset, Y. Zhang, and M. Schervish, “Path planning for robotic demining: Robust sensor-based coverage of unstructured environments and probabilistic methods,” *I. J. Robotic Res.*, vol. 22, pp. 441–466, 07 2003.

[8] M. Bosse, N. Nourani-Vatani, and J. Roberts, “Coverage algorithms for an under-actuated car-like vehicle in an uncertain environment,” in *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pp. 698–703, 2007.

[9] M. Ollis and A. Stentz, “Vision-based perception for an automated harvester,” in *Proceedings of the 1997 IEEE/RSJ International Conference on Intelligent*

Robot and Systems. Innovative Robotics for Real-World Applications. IROS '97, vol. 3, pp. 1838–1844 vol.3, 1997.

[10] M. Farsi, K. Ratcliff, J. P. Johnson, C. R. Allen, K. Z. Karam, and R. Pawson, "Robot control system for window cleaning," in *Proceedings of 1994 American Control Conference - ACC '94*, vol. 1, pp. 994–995 vol.1, 1994.

[11] B. Englot and F. Hover, "Sampling-based coverage path planning for inspection of complex structures," *ICAPS 2012 - Proceedings of the 22nd International Conference on Automated Planning and Scheduling*, 06 2014.

[12] E. Galceran and M. Carreras, "A survey on coverage path planning for robotics," *Robotics and Autonomous Systems*, vol. 61, no. 12, pp. 1258 – 1276, 2013.

[13] H. Choset, "Coverage for robotics - a survey of recent results," *Ann. Math. Artif. Intell.*, vol. 31, pp. 113–126, 10 2001.

[14] J.-C. Latombe, *Robot Motion Planning*. USA: Kluwer Academic Publishers, 1991.

[15] H. Choset, K. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. Kavraki, and S. Thrun, *Principles of Robot Motion: Theory, Algorithms, and Implementations*. May 2005.

[16] E. U. Acar, H. Choset, A. A. Rizzi, P. N. Atkar, and D. Hull, "Morse decompositions for coverage tasks," *The International Journal of Robotics Research*, vol. 21, pp. 331 – 344, 2002.

[17] A. Elfes, "Sonar-based real-world mapping and navigation," *IEEE Journal on Robotics and Automation*, vol. 3, no. 3, pp. 249–265, 1987.

[18] S. Thrun, "Learning metric-topological maps for indoor mobile robot navigation," *Artificial Intelligence*, vol. 99, no. 1, pp. 21 – 71, 1998.

[19] I. Rekleitis, A. New, E. Rankin, and H. Choset, "Efficient boustrophedon multi-robot coverage: an algorithmic approach," *Ann. Math. Artif. Intell.*, vol. 52, pp. 109–142, 04 2008.

[20] T. Cabreira, L. Brisolara, and P. R. Ferreira Jr., "Survey on coverage path planning with unmanned aerial vehicles," *Drones*, vol. 3, p. 4, Jan 2019.

[21] C. Di Franco and G. Buttazzo, “Coverage path planning for uavs photogrammetry with energy and resolution constraints,” *Journal of Intelligent Robotic Systems*, vol. 83, 02 2016.

[22] T. M. Cabreira, C. D. Franco, P. R. Ferreira, and G. C. Buttazzo, “Energy-aware spiral coverage path planning for uav photogrammetric applications,” *IEEE Robotics and Automation Letters*, vol. 3, no. 4, pp. 3662–3668, 2018.

[23] J. Acevedo, B. Arrue, I. Maza, and A. Ollero, “Cooperative large area surveillance with a team of aerial mobile robots for long endurance missions,” *Journal of Intelligent Robotic Systems*, vol. 70, 04 2013.

[24] J. J. Acevedo, B. C. Arrue, I. Maza, and A. Ollero, “Distributed approach for coverage and patrolling missions with a team of heterogeneous aerial robots under communication constraints,” *International Journal of Advanced Robotic Systems*, vol. 10, no. 1, p. 28, 2013.

[25] S. Koenig and Y. Liu, “Terrain coverage with ant robots: A simulation study,” pp. 600–607, 01 2001.

[26] J. Sauter, R. Matthews, V. Parunak, and S. Brueckner, “Performance of digital pheromones for swarming vehicle control,” pp. 903–910, 01 2005.

[27] D. Zhao and W. Guo, “Shape and performance controlled advanced design for additive manufacturing: A review of slicing and path planning,” *Journal of Manufacturing Science and Engineering*, vol. 142, pp. 1–87, 01 2020.

[28] I. Gibson, D. Rosen, and B. Stucker, *Additive manufacturing technologies: 3D printing, rapid prototyping, and direct digital manufacturing, second edition*. 01 2015.

[29] Y. Jin, Y. He, G. Fu, A. Zhang, and J. Du, “A non-retraction path planning approach for extrusion-based additive manufacturing,” *Robotics and Computer-Integrated Manufacturing*, vol. 48, pp. 132 – 144, 2017.

[30] H. Zhao, F. Gu, Q.-X. Huang, J. Garcia, Y. Chen, C. Tu, B. Benes, H. Zhang, D. Cohen-Or, and B. Chen, “Connected fermat spirals for layered fabrication,” *ACM Trans. Graph.*, vol. 35, no. 4, 2016.

[31] S. Shaikh, N. Kumar, P. K. Jain, and P. Tandon, “Hilbert curve based toolpath for fdm process,” in *CAD/CAM, Robotics and Factories of the Future* (D. K. Mandal and C. S. Syan, eds.), (New Delhi), pp. 751–759, Springer India, 2016.

[32] P. Gupta, B. Krishnamoorthy, and G. Dreifus, “Continuous toolpath planning in a graphical framework for sparse infill additive manufacturing,” *Computer-Aided Design*, vol. 127, p. 102880, 2020.

[33] Z. Lin, F. Jianzhong, Y. Sun, Q. Gao, G. xu, and Z. Wang, “Non-retraction toolpath generation for irregular compound freeform surfaces with the lkh tsp solver,” *The International Journal of Advanced Manufacturing Technology*, vol. 92, pp. 1–15, 09 2017.

[34] J. Feng, F. Jianzhong, Z. Lin, C. Shang, and X. Niu, “Layered infill area generation from triply periodic minimal surfaces for additive manufacturing,” *Computer-Aided Design*, vol. 107, 10 2018.

[35] D. Ding, Z. Pan, D. Cuiuri, and H. Li, “A practical path planning methodology for wire and arc additive manufacturing of thin-walled structures,” *Robotics and Computer-Integrated Manufacturing*, vol. 34, pp. 8 – 19, 2015.

[36] J.-h. Kao and F. Prinz, “Optimal motion planning for deposition in layered manufacturing,” 02 2001.

[37] J. Hergel, K. Hinz, S. Lefebvre, and B. Thomaszewski, “Extrusion-based ceramics printing with strictly-continuous deposition,” *ACM Trans. Graph.*, vol. 38, Nov. 2019.

[38] S. Kapil, P. Joshi, H. Vithasth, D. Rana, P. Kulkarni, R. Bhagchandani, and K. K.P., “Optimal space filling for additive manufacturing,” *Rapid Prototyping Journal*, vol. 22, pp. 660–675, 06 2016.

[39] J. Martínez, S. Hornus, H. Song, and S. Lefebvre, “Polyhedral voronoi diagrams for additive manufacturing,” *ACM Trans. Graph.*, vol. 37, July 2018.

[40] J. Martínez, J. Dumas, and S. Lefebvre, “Procedural voronoi foams for additive manufacturing,” *ACM Trans. Graph.*, vol. 35, July 2016.

[41] P. Tran, T. D. Ngo, A. Ghazlan, and D. Hui, "Bimaterial 3d printing and numerical analysis of bio-inspired composite structures under in-plane and transverse loadings," *Composites Part B: Engineering*, vol. 108, pp. 210 – 223, 2017.

[42] X. Zhai and F. Chen, "Path planning of a type of porous structures for additive manufacturing," *Computer-Aided Design*, vol. 115, pp. 218 – 230, 2019.

[43] M.-S. Pham, C. Liu, I. Todd, and J. Lertthanasarn, "Damage-tolerant architected materials inspired by crystal microstructure," *Nature*, vol. 565, 01 2019.

[44] M. Candeloro, A. M. Lekkas, and A. J. Sørensen, "A voronoi-diagram-based dynamic path-planning system for underactuated marine vessels," *Control Engineering Practice*, vol. 61, pp. 41 – 54, 2017.

[45] A. Shkolnik and R. Tedrake, "Path planning in 1000+ dimensions using a task-space voronoi bias," in *2009 IEEE International Conference on Robotics and Automation*, pp. 2061–2067, 2009.

[46] S. M. Lavalle, "Rapidly-exploring random trees: A new tool for path planning," tech. rep., 1998.

[47] O. Arslan and D. E. Koditschek, "Voronoi-based coverage control of heterogeneous disk-shaped robots," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 4259–4266, 2016.

[48] L. C. A. Pimenta, V. Kumar, R. C. Mesquita, and G. A. S. Pereira, "Sensing and coverage for a network of heterogeneous robots," in *2008 47th IEEE Conference on Decision and Control*, pp. 3947–3952, 2008.

[49] D. Jungnickel, *Graphs, Networks and Algorithms*. Springer Publishing Company, Incorporated, 3rd ed., 2007.