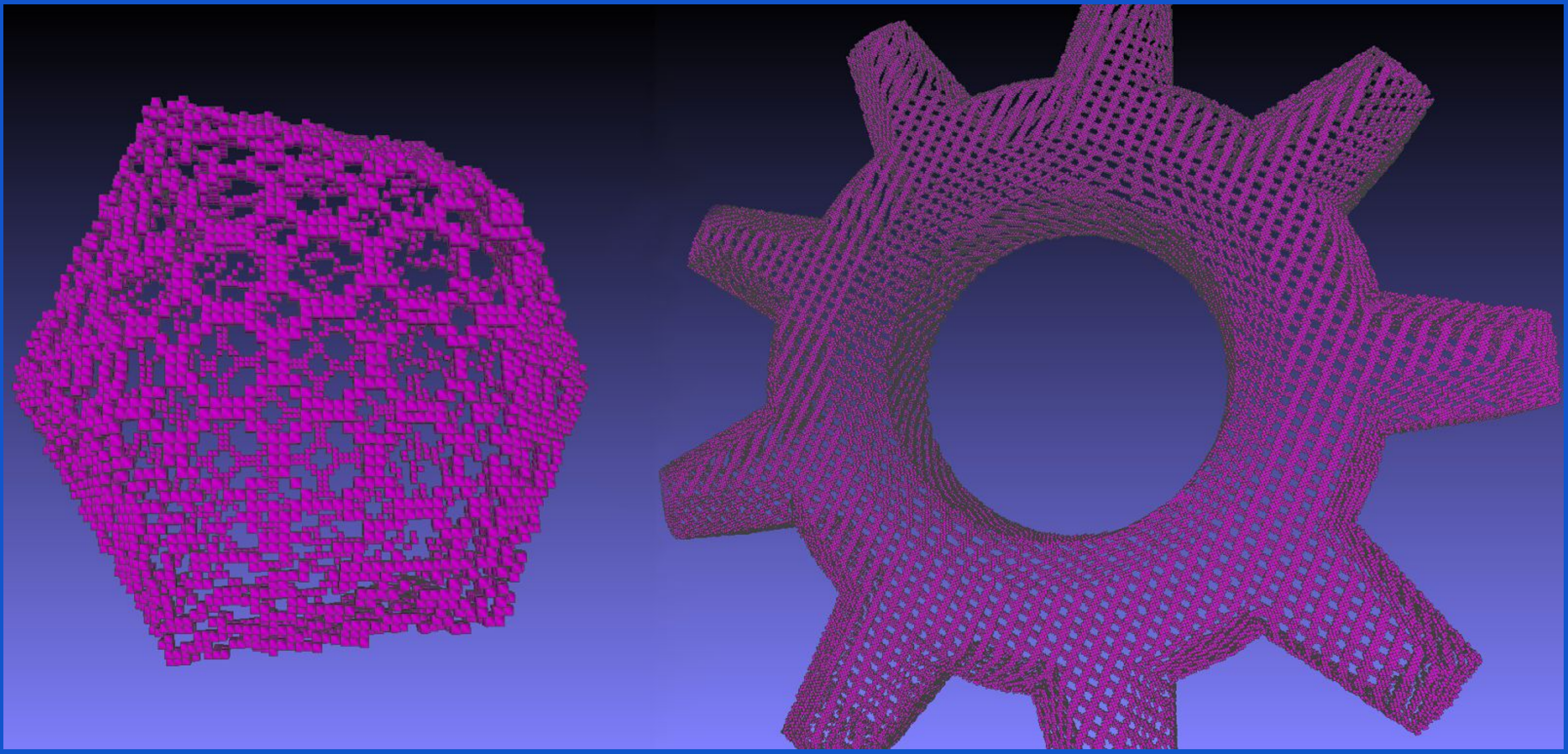
KINSUK DAS

14CS10025

VOXEL PATTERN SYNTHESIS

March 31, 2017



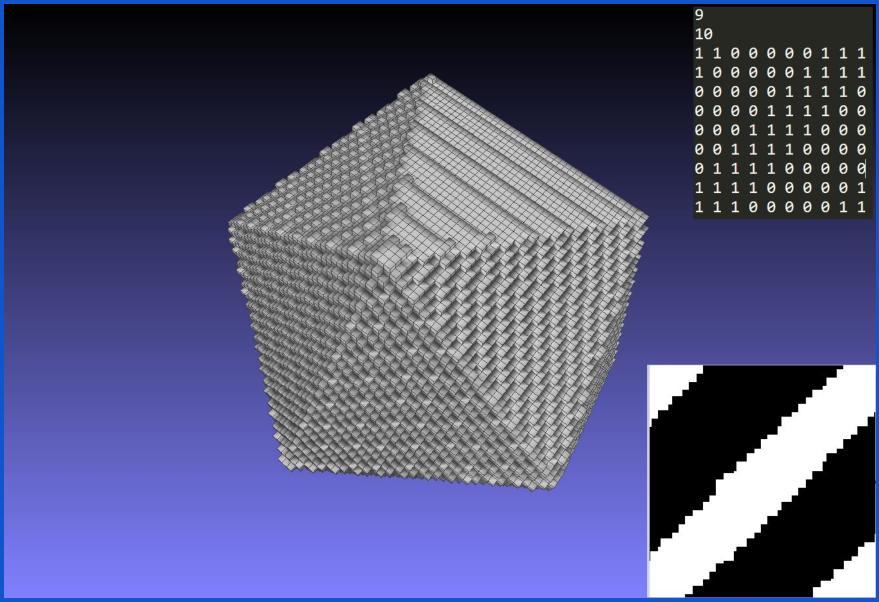
OBJECTIVE

1. **Generating repeated pattern on voxel surface:** The algorithm repeats the pattern on the voxel surface and removes some voxels that are marked as empty on the repeated pattern, and prints the remaining voxels into a new file.
2. **Connectivity of voxels:** The voxels in the output file must be connected in 26 neighborhood

OVERVIEW

Input

This approach starts with a User specified closed voxel surface of an object, say a set of voxel centers “V”, and an exemplar input pattern P. The exemplar pattern is an text file specifying the height (h) and width (w) of the required pattern and the pattern as a matrix of height h, and width w, consisting of 0s and 1s, 0 meaning solid and 1 meaning empty.



```
9
10
1 1 0 0 0 0 0 1 1 1
1 0 0 0 0 0 1 1 1 1
0 0 0 0 0 1 1 1 1 0
0 0 0 0 1 1 1 1 0 0
0 0 0 1 1 1 1 0 0 0
0 0 1 1 1 1 0 0 0 0
0 1 1 1 1 0 0 0 0 0
1 1 1 1 0 0 0 0 0 1
1 1 1 0 0 0 0 0 1 1
```

Voxel Input (middle), Exemplar patter (top right), how pattern look(bottom right)

Pre-Processing

1. As a preprocessing step, we run “find_min_max.cpp” to find the bounds (bounding cube in 3D space) of the given voxels centers. The bounds are stored in “limits.txt” with min first, then max. Accordingly the limits of other cpp files are changed, for proper storage of the voxels with minimum space.
2. Also the normals at each voxels is calculated using “normals.cpp”. The first line of this text file also contain minx, miny, minz, and second line contains maxx, maxy, maxz, and the components of normal at each voxel, given in order of input file is stored in “normals.txt”.

For calculating the normals the following algorithm is followed :

NORMALS (VoxelSet V)

Output : File with min and max coordinates, and normals for voxels in order

Initialize a 3D array “voxelMap” to all 0s .

Adjust the centers of the voxels, so that all negative coordinates are translated to positive (not all zero) coordinates.

Find max and min coordinates and write to normals.txt.

1. BFS(0,0,0) on the 3D array in 26-neighborhood where 0,0,0 is the start position, traversing only 0 tagged voxels.
2. After the BFS, for each voxel on surface, in k-neighborhood (k is hard-coded) :
    a. Initialize a normal vector to zero vector.
    b. If neighboring voxels sharing its opposite faces are marked 0 and 2, then add the unit vector from the voxel marked 0 to voxel marked 2, to “normal” vector. Continue for all pairs of opposite faces in both directions. If all are done continue to step 5e.
    c. Similar procedure as 5b is done, with opposite edges and the proceed to 5e.
    d. Similar procedure as 5b is done, with opposite vertices and the proceed to 5e.
    e. Normalize the vector “normal”, and print it to “normals.txt”.

In the above algorithm, after the BFS traversal , voxels marked as 0 are the voxels interior to the surface, 1 are the surface voxels, and 2 are the exterior voxels in 3D array. For each voxel the net normalized vector, with the above algorithm (from step 5) gives a measure of the normal at that surface voxel.

```
PRE PROCESS( VoxelSet V)

Output : L , the maximum and minimum value of coordinates
         N , the normals at each voxel and minimum bounding box of the given voxels set

1. L <- Limits(V)
2. Change the parameters in NORMALS and SURFACE GRAPH
3. N <- NORMALS(V)
```

RUNNING TIME

The computation of normals takes the longest time. If the bounds are of order O(h), k is the k-neighborhood considered, and the number of voxels of order O(n), then normal computation time complexity is O(h^3 + k*n), but since O(n) = O(h^3) as the bounding box will have all the voxels so it boil down to O(h^3).

Algorithm

PATTERN SYNTHESIZER(VoxelSet V, Pattern P, NormalVectors N)

Output : V’, i.e. connected set of surface voxels centers with pattern repeated, connected in 26-neighborhood of solid voxels

P’ <- APPLY PATTERN(V, P, N) , where P’ is the set voxels V tagged as solid(1) or empty (2), according to pattern repeated. (can have repeated voxels)

P <- REMOVE DUP(P’) , where

G <- SURFACE GRAPH(P) , where G is set of voxels with its coordinates; a set of indices pointing to the neighbors of the voxel, in 26-neighborhood, marked as solid or empty; and a component number. The component number is follows the following rules:

a. If voxel i and voxel j are connected in 26-neighborhood and, either both are solid or both are empty, i and j will have the same component number.
b. If voxel i and voxel j are both solid, and are not connected by a path of voxels tagged as solid, then voxels i and voxel j will have two different component numbers.
c. If voxel i is empty then component number assigned is two, else it greater than 2.

V’ <- CONNECTED SAMPLES(G) , where V’ is the final required output of voxel centers.

APPLY PATTERN(VoxelSet V, Pattern P, NormalVectors N)

Output : P’, i.e. set of surface voxels centers with pattern repeated

```
    BOX <- set of 6 planes, which act as a bounding box of the voxels.
1.  For each voxel in V
        a. Ray cast the normal from the current voxel to intersect all the planes, with
           parameter (t) . If t < 0, then ignore the intersection.
        b. Consider the point of intersection with least positive parameter t, let the
           plane with  which it intersects be Pi.
        c. Round of the coordinates of point of intersection to integer coordinates,
           let they be (x,y,z),  x,y,z $\in \mathbb{Z}$ . Since they are on the faces of the bounding
           box, so one of x,y,z will be zero.
        d. Now if P[i%h][j%w] (h=height, w=width) is 1 then tag the voxel with “1”, else
           tag with “2”. Here i, and j are the non-zero coordinates among x,y,z. As to
           which coordinate is i, and which coordinate is j, is hard-coded so as to
           obtain a satisfactory result. A different choice of i, j, among the non-zero
           coordinates x,y,z. Will lead a different result.
```

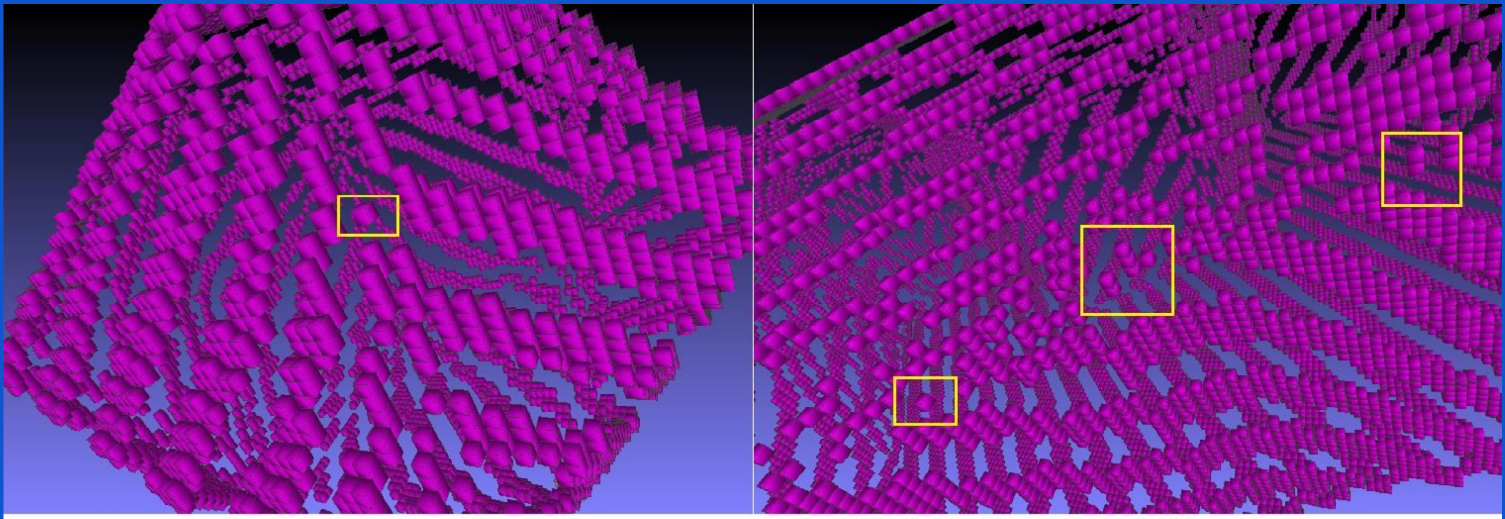
1. Here if V is the position vector of a voxel, N is the Normal vector at voxel and considering a plane x=maxX, then the following equation holds :
2. (maxX, reqY, reqZ) = V + N*t , where t ranges from 0 to inf. And in the code we check the value of P[reqY%width][reqZ%height], if zero then taaged 2, else tagged 1. From the equation maxX = V.x + N.x*t, we get value of t, and then use this value to get reqY and reqZ.

For each of the following planes following are the details of the transformations of the plane :

1) x=0 : P[y][width-1-z%width]
2) x=maxX : P[y][z]
3) y=0 : P[x][z]
4) y=maxY : P[x][width-1-z%width]
5) z=0 : P[y][z]
6) z=maxZ : P[y][width-1-z]

The time complexity of the above algorithm is O(n), where n is the number of voxels in V. A better result might be obtained if more number of planes are taken into consideration, but for simplicity only 6 planes, with normals parallel to one of the coordinate axes, is taken.

The Voxel Set obtained from the above algorithm is rarely connected, so a 3D printing of this object set will result it to crumble. Below are the results, if only Apply Pattern is executed.



ICOSAHEDRAL - NOT CONNECTED COGWHEEL - NOT CONNECTED

Hence our following steps converts a number of empty voxels to solid voxels , which is close to the actual minimum, by performing Dijkstra's algorithm.

P <- REMOVE DUPLICATES(P’) just ensures that only instance of a voxel exists, in P at a particular coordinate, by removing duplicates in P’.

This step is necessary because, during the formation of graphs in SURFACE GRAPH, the edges between two voxels are specified by the index to the other neighboring voxel. If duplicates remain then the duplicated voxel won’t receive a valid index.

SURFACE GRAPH(TaggedVoxelSet P)

Output : G, a adjacency list graph where each vertex is a voxel with, a component number, number of neighbors (ni) regardless of solid or empty, followed by ni indices (where each index represents a unique voxel).

```
        M <- MAP(empty).

        VOXELARR <- 3D array of voxels, which should contain the component number at
that voxel coordinate.

        Count = 0, CompNum = 3.

    1. Foreach voxel in P
        a. Convert the coordinate of P as a key in M, and at that key insert value of
           Count in M. Increment Count after insertion.
    2. Foreach voxel v in P
        a. If it is tagged as solid, do a CustBFS(v), before moving to 4b, else move to
           4b.
        b. In 26-neighborhood of the current voxel, store the indices of voxels in P,
           stored in M, which are in its neighborhood, say the set is Neighbors.
        c. In G, add current vertex, with component number assigned by CustBFS,
           and list of indices in Neighbors, along with number of neighbors.
```

CustBFS(Voxel v)

Output : Update the VOXELARR, with the current CompNum, for all the solid voxels connected to solid voxel v. Two solid voxels are connected if there exists a path of solid voxels in 26-neighbohood between them.

```
1.  Q <- Empty Queue
2.  Update the VOXELARR[v.x][v.y][v.z] <- CompNum
3.  Q.Enqueue(v)
4.  While Q is not empty
        a.  R <- Q.Dequeue()
        b.  Check the 26-neighborhood of R, if it is a solid voxel, say S, then update
            VOXELARR[S.x][S.y][S.z] <- CompNum.
            And do Q.Enqueue(v)
5.  Increment CompNum
```

The time complexity of the above algorithm is basically the time complexity for performing a standard BFS, i.e. O(n + 26*n) where n is the number of voxels, and 26*n is the number of edges, hence basically the time complexity is O(n).

CONNECT SAMPLES(Adjacency list Graph G)

Output : V’, i.e. connected set of surface voxels centers with pattern repeated, connected in 26-neighborhood of solid voxels

```
    G’ <- new empty graph, array of vertices where each index stores a vertex in G, component number, and neighbors.

    MST<- new empty tree, array of tree nodes, where each index stores a vertex in G, comp number, its parent and children in the n-ary tree.

    dist[N] <- initialized to all infinity, N is the number of voxels in G

    V’ <- empty set

    Q <- empty Priority Queue, each element has vertex v, and distance of v from source, such than it is a min-Heap on the distances.

1. Read and store the graph G, foreach vertex v in G
    a. If component number of v is 3 then dist[v.index] <- 0, and Q.Enqueue(v)
2. MST <- SHORTEST PATH(G’, Q), updates all the tree nodes with respective parent and children.
3. MST <- CONVERT EMPTY TO SOLID(MST), basically converts those empty voxels which fall on path in between two solid voxels in MST.
4. Foreach node in MST
    a. If node comp number is >=3, add the voxel coordinate to V’
```

SHORTEST PATH(VERTEX ARRAY G’ , Priority Queue Q)

Output : MST, an array of Tree nodes, with parent and child, which is a shortest path tree in G’ from the vertices which have comp number =3.

```
1. While Q is not empty
    a. R <- Extract Minimum distance vertex from Q. [Q.Dequeue()]
    b. For neighbors of R, say S in G’
        i. If dist[S] > dist[R] + 1
            1. dist[S] = dist[R]+1
            2. Q.Enqueue(dist[S], S)
            3. MST[u].SetChild(v)
            4. MST[v.parent].deleteChild(v)
            5. MST[v].setParent(u)
```

Note that here the shortest path tree is created irrespective of empty or solid voxels

CONVERT EMPTY TO SOLID(Tree Node Array MST)

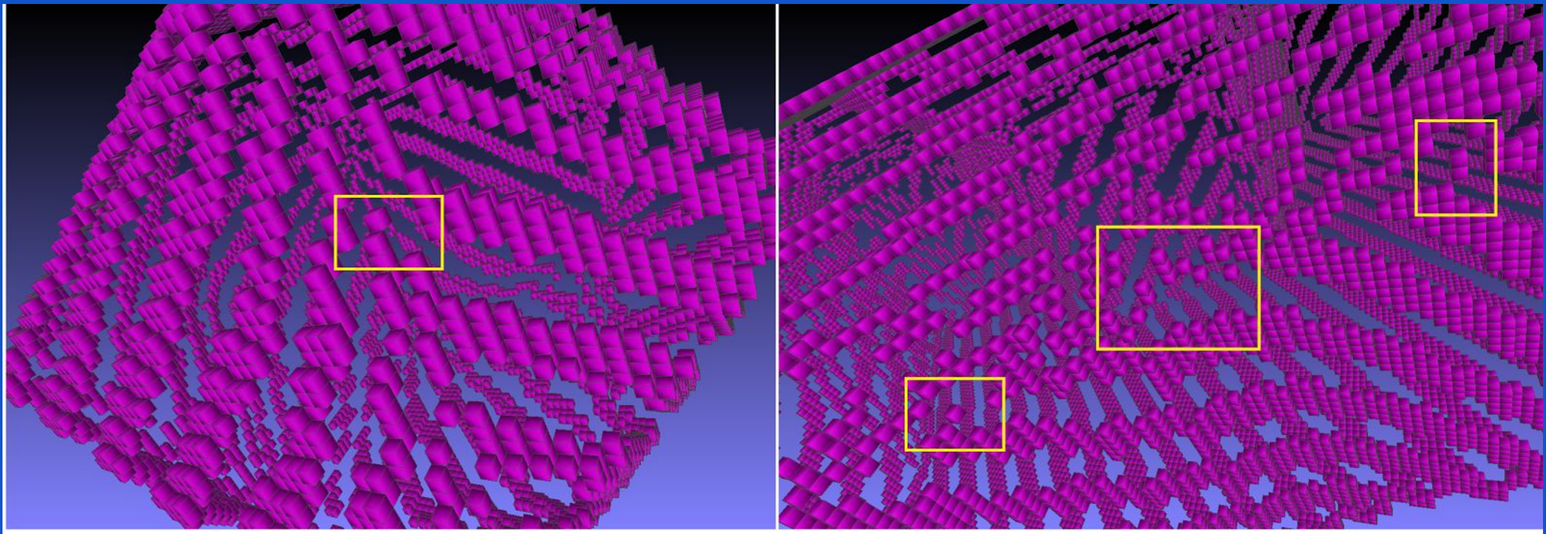
Output : MST, updated with new component numbers such that minimal number of empty voxels are converted to solid, and path between no two solid voxels contains an empty voxel.

```
1. leastNode[totCompNum], where leastNode[i] should contain the vertex index ,
   with component number i, and at least distance from source. This can be done
   by traversing the voxels in G and checking array "dist".
2. For each element in leastNode , say j
   a. While MST[j].parent != -1
       i.  If MST[j].compNum < 3 then MST[j].compNum = totCompNum;
       ii. j = MST[j].parent;
```

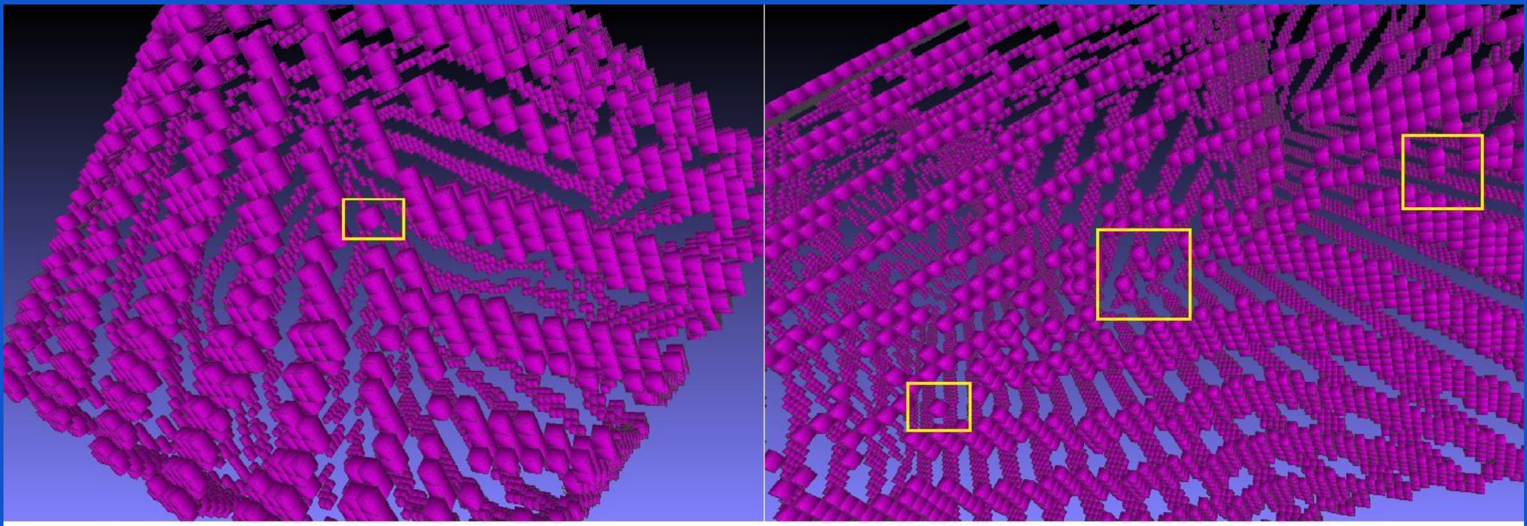
The time complexity of the above algorithm is O(nlog(n)) where n is the number of vertices in graph.

Shortest path simply follows Dijkstra's algorithm, however the queue initially supplied to Shortest path algorithm contain not only the source voxel but also other solid voxels which are connected to source by a path of solid voxels, with distances initialized to zero.

The results obtained after performing the algorithm are as follows.



ICOSAHEDRAL CONNECTED COGWHEEL CONNECTED



ICOSAHEDRAL - NOT CONNECTED COGWHEEL - NOT CONNECTED

Conclusion

1. All the algorithms mentioned above conclude the synthesis of pattern on a voxel surface.
2. Further improvements to this algorithm can be done by increasing the number of planes from which the repeated pattern is applied initially on the voxel surface, ideally a sphere with repeated pattern on it, better ways to compute the normals at each voxel on surface
3. Improving the connectivity feature, by having a user specified force profile, considering each edge as a beam, and thus connecting the voxel pattern to withstand more force, hence allowing it to be printed with heavier material.
4. In addition to the above improvements, a thickness parameter can be introduced, where each voxel is extended to the interior with specified thickness, so that the final structure can become more stronger.