

Finding Shortest Isothetic Path Inside a 3D Digital Object

Debapriya Kundu and Arindam Biswas^(✉)

Department of Information Technology, Indian Institute of Engineering Science
and Technology, Shibpur, Howrah 711103, West Bengal, India
debapriyakundu1@gmail.com, barindam@gmail.com

Abstract. The problem of finding shortest isothetic path between two points is well studied in the context of two dimensional objects. But it is relatively less explored in higher dimensions. An algorithm to find a shortest isothetic path between two points of a 3D object is presented in this paper. The object intersects with some axis parallel equi-distant slicing planes giving one or more isothetic polygons. We call these polygons as slices. The slice containing the source and destination points are called source and destination slice respectively. A graph is constructed by checking the overlap among the slices on consecutive planes. We call it slice overlap graph. Our algorithm first finds the source and destination slice. Thereafter, it finds the minimum set of slices I_{st} from the slice overlap graph, that need to be traversed to find SIP. Finally BFS is applied to find a SIP through these set of slices. The advantage of this procedure is that it does not search the whole object to find a SIP, rather only a part of the object is considered, therefore making the search faster.

Keywords: Unit grid cube (UGC) · Shortest isothetic path (SIP) · Breadth first search (BFS)

1 Introduction

The problem of finding shortest path is a well studied area in computational geometry. Path planning or motion planning in automated robots is a challenging area of robotics. Computing shortest path between different locations is often used in various map services and navigation systems. Various prior works on finding shortest path in 3D have been done. Problems of finding shortest path can be of different types. Some of the examples are finding shortest path on 3D polyhedral surface, in presence of a sequence of obstacles or polyhedral obstacles, finding shortest rectilinear minimum bending path, minimum distance path etc. The various kinds of shortest path problems in three dimension are discussed below.

An efficient parallel solution to the problem of finding shortest Euclidean path between two points in three dimensional space in the presence of polyhedral obstacles is discussed in [1]. It finds the shortest path touching n lines in a

specified order. A more general problem is of finding collision-free optimal path for a given robot system. A solution to the problem of finding shortest path in three dimension with polyhedral obstacles is discussed in [7]. A new approach to solve this problem is discussed in [5] which solves the problem in $O(n^3 v^k)$ time, where n is the number of vertices of the polyhedra, k is the number of obstacles and v is the largest number of vertices on any one obstacle. Recently an algorithm has been proposed on finding shortest path in three-dimensional cluttered environment [11].

A different kind of problem is construction of multilayer obstacle avoiding rectilinear Steiner minimum tree. A Steiner tree problem is a combinatorial optimization problem. For a given set of geometric points, the Steiner tree problem is about finding a minimum length graph interconnecting all these points where the length of the graph is the sum of the lengths of all edges. The points are called Steiner points. When the edges are restricted to be axis parallel the problem reduces to rectilinear minimum Steiner tree problem. An analysis of the existing rectilinear Steiner minimum tree algorithms is given in [10]. This has application in circuit layout, network design, VLSI physical design, wire routing, telecommunications, etc.

A related problem is finding rectilinear minimum link path problem in three dimensions [3, 13]. It involves finding a continuous path with axis-parallel line segments, such that none of the line segments intersect the interior of any obstacle. [3] solved the problem in $O(n^2 \log n)$ time with worst case running time of $\Omega(n^3)$. A betterment on the worst case running time has been achieved in [13], where the worst case time complexity is $O(n^{5/2} \log n)$.

Our Contribution: Let A be a 3D digital object and s and t be the source and destination points. Our objective is to find a SIP (π_{st}) between s and t such that the SIP lies entirely inside A and consists of moves along grid edges only. There may exist a number of SIPs between s and t , our algorithm will find one of them. Here by point we mean digital points only i.e., points with integer coordinates. The algorithm proposed here is for genus 0 objects. To the best of our knowledge there exists no suitable algorithmic solution on the geometric problem stated above. For a given grid size the proposed algorithm runs in $O(\sum_{i=1}^k y_i)$ time where k is the set of slices in the shortest path from source to destination grid vertex and y_i denotes the number of UGCs in the i^{th} slice of these set of slices. The worst case time complexity of the proposed algorithm is $O(\sum_{i=1}^k y_i) = O(N)$, where N is the total number of UGCs of the given digital object. Though the worst case complexity is high still this algorithm is computationally very fast for best and average cases.

2 Definitions and Preliminaries

2.1 Digital Grid

A *digital grid* \mathbb{G} consists of three orthogonal sets of equi-spaced grid lines, \mathbb{G}_{yz} , \mathbb{G}_{zx} , and \mathbb{G}_{xy} , where $\mathbb{G}_{yz} = \{l_x(j \pm ag, k \pm bg) \mid a \in \mathbb{Z}, b \in \mathbb{Z}\}$. Similarly, \mathbb{G}_{zx}

and \mathbb{G}_{xy} can be represented in terms of l_y and l_z for a grid size $g \in \mathbb{Z}^+$. Here, $l_x(j, k) = \{(x, j, k) : x \in \mathbb{R}\}$, $l_y(i, k) = \{(i, y, k) : y \in \mathbb{R}\}$, and $l_z(i, j) = \{(i, j, z) : z \in \mathbb{R}\}$ denote the *grid lines* (Fig. 1) along x -, y -, and z -axes respectively, where i , j , and k are integer multiples of g . The three orthogonal lines $l_x(j, k)$, $l_y(i, k)$, and $l_z(i, j)$ intersect at the point $(i, j, k) \in \mathbb{Z}^3$, which is called a *grid point*; a shift of $(\pm 0.5g, \pm 0.5g, \pm 0.5g)$ with respect to a grid point designates a *grid vertex*, and a pair of adjacent grid vertices defines a *grid edge* [6]. Therefore, the grid point set is \mathbb{Z}^3 . A grid square is defined by the four grid edges that form a square. A grid cube is defined by six grid squares that form a cube. It is also called a 3-cell, grid square is a 2-cell, a grid edge is a 1-cell, and a grid vertex is a 0-cell. A *unit grid cube* (UGC) is a (closed) cube of length g whose vertices are *grid vertices*, edges constituted by *grid edges*, and faces constituted by *grid faces*.

A *unit grid cube* (UGC) is grid cube of length g whose vertices are *grid vertices*, edges constituted by *grid edges* and faces constituted by *grid faces*. Each face of a UGC lies on a *face plane* (hence referred as a UGC-face), which is parallel to one of three coordinates planes. A UGC contains $g \times g \times g$ number of voxels. So a UGC is same as a voxel when $g = 1$ [8]. A grid for $g = 2$ with its elements are shown in Fig. 1.

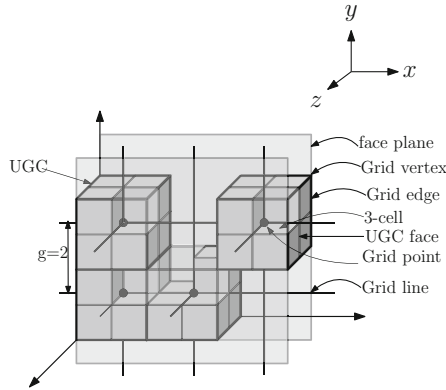


Fig. 1. 3D grid for $g = 2$.

2.2 Adjacency

Two cells c_1 and c_2 are called α adjacent if $c_1 \neq c_2$ and the intersection contains an α -cell ($\alpha \in 0, 1, 2$). Two 3D grid points $p_1 = (x_1, y_1, z_1)$ and $p_2 = (x_2, y_2, z_2)$ are called 6-adjacent iff $0 < d_e(p_1, p_2) \leq 1$, 18 adjacent iff $0 < d_e(p_1, p_2) \leq \sqrt{2}$, and 26 adjacent iff $0 < d_e(p_1, p_2) \leq \sqrt{3}$.

Let c_1 and c_2 be 3-cells and let p_i be the center of c_i ($i = 1, 2$). Then c_1 and c_2 are 0-adjacent iff p_1 and p_2 are 26-adjacent iff c_1 and c_2 are not identical but share a grid vertex; c_1, c_2 are 1-adjacent iff p_1 and p_2 are 18 adjacent iff c_1 and c_2 are not identical but share a grid edge; and c_1, c_2 are 2-adjacent iff p_1 and p_2 are 6 adjacent iff c_1 and c_2 are not identical but share a grid square [8] (Fig. 2).

2.3 Digital Object

Let A be a 3D digital object, (referred as an object), which is defined as a finite subset of \mathbb{Z}^3 , with all its constituent points (i.e., voxels) having integer coordinates and connected in 26-neighborhood [6]. Here, a digital object is a 26-connected component.

2.4 Isothetic Path

An isothetic path from a grid vertex $p \in A$ to a grid vertex $q \in A$ is defined as the sequence of 6-adjacent grid vertices which are all distinct and lie on or inside the digital object A (Fig. 3). An isothetic path of minimum length is called a shortest isothetic path (SIP) [2].

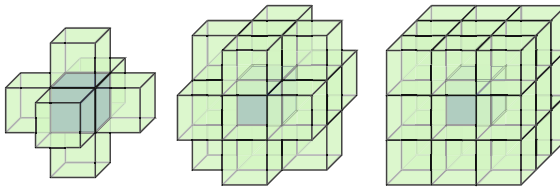


Fig. 2. Left: 6-adjacency; Middle: 18-adjacency; Right: 26-adjacency.

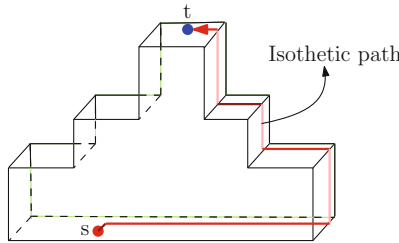


Fig. 3. An isothetic path with source and destination are marked as red and blue respectively. (Color figure online)

3 Slicing

The object A is considered to be a set of voxels. A UGC is said to be object occupied if it contains an object voxel. For slicing the object all the object occupied UGCs need to be found. The object intersects with some grid planes along xy -, yz - and zx - axis. The intersection of the object with a grid plane gives one or more isothetic polygons. We call these polygons as slices. Let us consider $\{H_1, H_2, H_3, \dots\}$ be the set of slicing planes separated by g parallel to any one of yz -, zx -, or xy -plane. Each slice is uniquely identified using a *faceid*. For slicing

an object we used the algorithm proposed in [6]. The entire process of slicing is discussed below in brief.

To start tracing a slice along a slicing plane a start vertex, v_s , is identified. A vertex qualifies as a start vertex if it is unvisited. Each vertex has eight neighboring UGCs. Depending on the object occupancy of these UGCs a starting direction is determined. Tracing a slice starts from v_s and concludes when it comes back to v_s . During traversal all the grid vertices lying in the path of traversal are marked visited. Along a given slicing plane the traversal at some point can be in four possible directions. The direction of traversal depends on the occupancy of the four neighboring UGCs along that plane and the incoming direction of traversal at that point. By applying a set of combinatorial formulas the outgoing direction is found such that the object always lies left. A slice contains the top faces of the adjacent UGCs along that plane. The slicing procedure traces each slice exactly once. An object and its slices are shown in Fig. 4. The set of slices is stored in an adjacency list (L). Each list in L contains the slices along one grid plane. Therefore, the number of lists in L is equal to the number of grid planes the object intersects with the direction of slicing i.e., either xy- or yz- or zx-. Let us consider this count as l_s . The result of slicing bunny is presented in Fig. 5. Here, w.l.o.g., we are slicing the object along zx-plane.

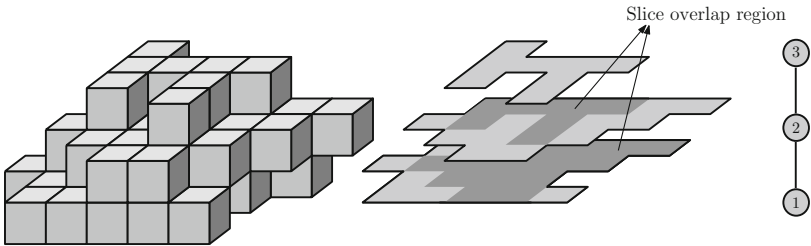


Fig. 4. Left: 3D object A ; Middle: Its slices along zx - axis for $g = 1$. The slices are shown in light grey and the overlapping regions are shown in dark grey; Right: corresponding slice overlap graph.

3.1 Slice Containing a Given Point

To find the slice containing a given point p , the slicing plane containing p is identified first using its coordinates thereafter searching the slices on that plane. Given a point i and a slice S_i , to check whether $i \in S_i$, a LeftOn test [12] is done for each edge of the slice. This technique checks for each edge whether the query point p lies on it or on the left (right) of it, when the slice is traversed in an anti-clockwise (clockwise) manner. The detailed steps of this procedure is discussed in IDENTIFYSLICE (Fig. 6) which identifies the slice S_p containing a given point p . Using this procedure the source and the destination slices corresponding to the source and destination points are obtained respectively. Figure 10 (d, e) shows a given source and destination points with their corresponding UGCs and slices.

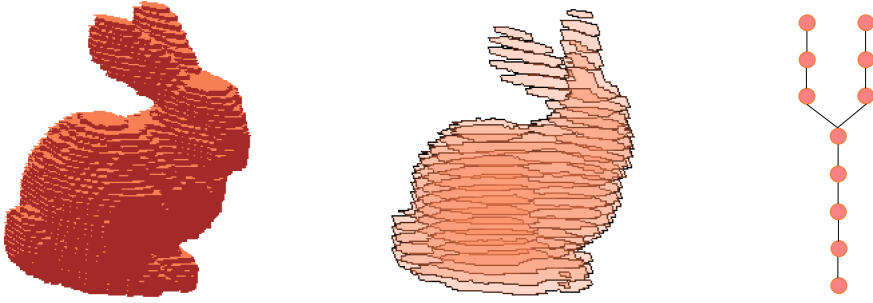


Fig. 5. Left: Bunny represented as a set of UGCs for $g = 1$; Middle: Its slices along xz -axis for $g = 1$; Right: Slice overlap graph, (here for clarity of view only a few nodes are shown).

4 Finding the Minimum Set of Slices (Πst) to be Traversed to Move from Source to Destination Slice

4.1 UGC Corresponding to a Point

In this paper, a grid based approach of finding a SIP is presented. Therefore, the UGCs corresponding to the source and destination points needs to be found. For any point i , we need to identify the UGC corresponding to it and for that the voxel corresponding to that point needs to be selected first. We denote the voxel and UGC corresponding to i as v_i and u_i respectively. There are eight neighboring voxels of i . Not necessarily all of them will be object voxel. To choose one voxel from this set of eight voxels as v_i , an object voxel is selected based on a defined order of preference. A voxel with lower preference value will get higher priority. The eight neighboring voxels of a given point with their preference values are shown in Fig. 7(Left). Once v_i is found the UGC containing v_i is selected as u_i . We denote the source and destination points as s and t , hence the source and destination UGCs as u_s and u_t respectively.

4.2 Slice Overlap Graph

Two slices on consecutive planes are said to have overlap if their projection on a plane overlap with each other. A graph is constructed considering each slice of A as a node and an edge between two nodes if there is overlap between the corresponding slices. We call this graph as slice overlap graph. As each slice has an unique *faceid*, so the node corresponding to that slice is uniquely identified using that *faceid*. As the object A is a digital object hence its slices will be orthogonal polygons. To check overlap between two orthogonal polygon (say S_i and S_j) we adopt a simple method, which is discussed below.

S_i and S_j are traced starting from any of their boundary grid vertices. Each grid vertex occurring in the path of traversal of each slice are given label equal to

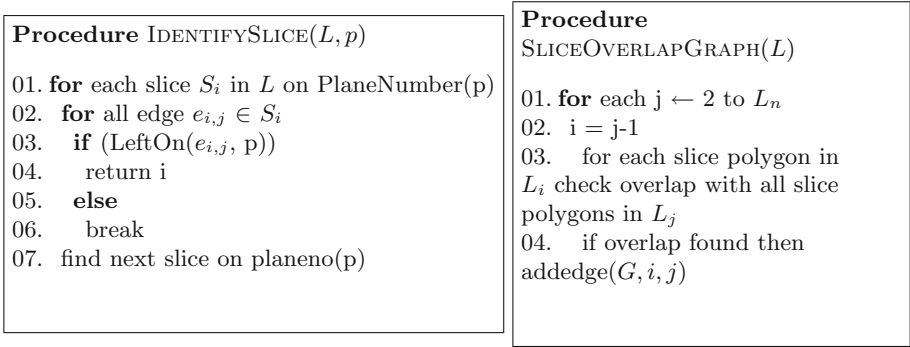


Fig. 6. Left: Procedure IDENTIFYSLICE; Right: Procedure SLICEOVERLAPGRAPH.

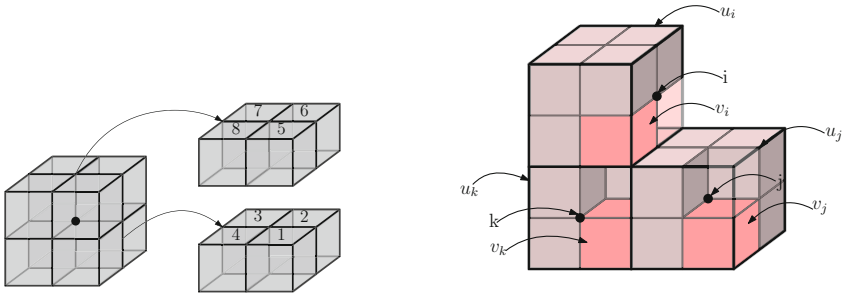


Fig. 7. Left: Order of preference of eight neighboring voxels of a point (the point is shown as a black dot); Right: voxels and UGCs corresponding to some given points. Here, the points are shown in black dots, the voxel corresponding to a point is marked red and the UGC containing that voxel is marked light red. (Color figure online)

the *faceid* of its corresponding slice. Therefore by this process each grid vertex on the boundary of a slice gets a label. Here by slice we mean only object slices, not any hole polygon. To check overlap we start from the slices on plane Π_2 . Let us denote the slices on i^{th} plane as $S_{i,1}, S_{i,2}, S_{i,3}, \dots$. Hence, j^{th} slice on i^{th} plane is denoted by $S_{i,j}$. The grid vertices on the boundary of $S_{i,j}$ are denoted by $v_{i,j,1}, v_{i,j,2}, \dots$. Therefore, the k^{th} grid vertex of polygon j on slice i will be denoted by $v_{i,j,k}$. We traverse each grid vertex on the boundary of a slice and for each grid vertex $v_{i,j,k}$ with label l we check label l_1 of $v_{i,j-1,k}$ (w.l.o.g., for slices along zx -plane), if $l_1 \neq 0$, then an edge is added to the slice overlap graph between nodes with *faceid* l and l_1 , if this edge does not already exist in the slice overlap graph. This method is able to identify the overlap between projections of two slices if the projections intersect with each other. If the projections of slices do not intersect each other and one is contained within another then the following method is adopted. For two slices (say, $S_{i,j}$ and $S_{i,k}$) between which no edge has been reported by the previous method we find whether $S_{i,j}$ is contained within $S_{i,k}$ or vice-versa. We take any one point from the vertex set of $S_{i,j}$ and

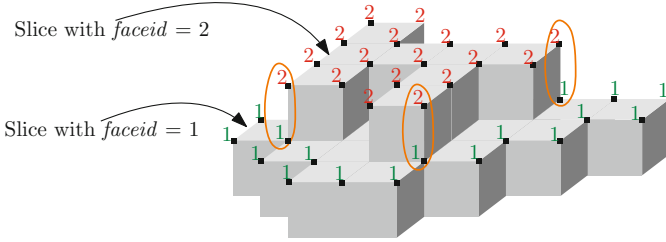


Fig. 8. The labelling of grid vertices on each slice

the method discussed in Sect. 3.1 is applied to find whether it is contained within $S_{i,k}$. If it returns false then the similar method is applied to check whether $S_{i,k}$ is contained within $S_{i,j}$. If there exists an overlap, an edge is added between the nodes corresponding to $S_{i,j}$ and $S_{i,k}$ to the slice overlap graph else no edge is added. Hence by visiting only the boundary grid points of all the slices the complete slice overlap graph is constructed. Figure 8 shows an object with two slices and the labels of the grid vertices on the boundary of each slice.

The procedure SLICEOVERLAPGRAPH is given in Fig. 6. Figures 4, 5 and 10(b, c) shows the slice overlap graph of some objects. The overlapping regions are shown in dark shade.

For genus 0 objects slice overlap graph will be a tree, but for objects of type genus 1 or more it will contain one or more cycles in it. The algorithm proposed in this paper is for genus 0 objects only.

Instead of constructing a slice overlap graph taking all the slices it can be made specific to the source and destination slices. This will require less amount of storage space as well as time to find Π_{st} . This graph construction starts from the source slice and only those slices that overlap with it are further explored and this process continues till the destination slice is visited. This process follows the BFS technique. So this graph will not necessarily contain all the slices of the object, which makes it time and space efficient. But the problem associated with this approach is, as it is specific to a given source and destination slice, the source and destination slices can not be changed dynamically.

4.3 Storage of Slice Overlap Graph

The slice overlap graph is stored in an adjacency list (L') where the number of lists is equal to the number of nodes in the graph. Each list gives information about a particular node i.e., the nodes adjacent to it. Each node of a list contains 2 elements a pointer to the first node of the slice corresponding to it and a pointer to the next node. If a node's *faceid* is 1 it is stored in the 1st list, similarly a node with *faceid* 2 is stored in the 2nd list. Figure 9 shows the memory representation of the slice overlap graph of the object represented in Fig. 4.

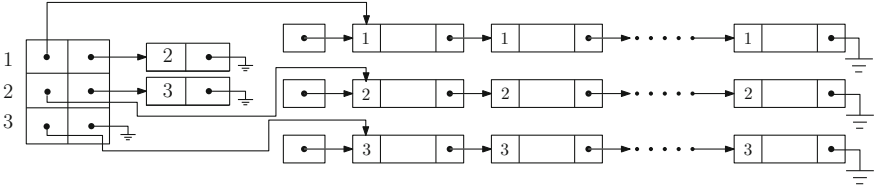


Fig. 9. Representation of slice overlap graph in memory

4.4 Finding the Minimum Set of Slices (Π_{st}) to be Traversed to Find SIP

To find path between two nodes in slice overlap graph BFS is applied. It starts from the source node and continues search until the destination node is visited. The source node is labeled 0 and marked visited. For a node with label i , its adjacent unvisited nodes are labeled $i+1$ and marked visited. Therefore, the label at a node is its shortest distance from the source node. The shortest path between the source and destination node is found by retracing. This is done in Step 5–6 of the algorithm FINDSIP. The minimum set of nodes or slices found this way, are stored in a linked list (L''). Each node of this list contains the value of the *faceid* of the corresponding slice and a pointer to the next node of Π_{st} .

5 Algorithm FINDSIP

To find a SIP, BFS is applied on the set of UGCs, that are bounded by the slices in the list L'' . BFS starts from u_s which is labeled as zero and it continues till u_t is visited. The label of a given UGC gives the count of the minimum number of UGCs that need to be traversed to reach it from the source UGC. Here, BFS is done in 26-neighborhood as the object is connected in 26-neighborhood. Finally SIP is found by retracing the visited UGCs. The complete algorithm to find SIP is given below. A step by step process of finding SIP is shown in Fig. 10. This algorithm follows grid based approach of solving the problem. The BFS done in Step 7 can be considered as an extension of Lee's [9] wavefront technique to three dimension.

Algorithm FINDSIP(A, s, t)

01. $L = \text{SLICE}(A)$
02. $S_s = \text{IDENTIFYSLICE}(L, s)$
03. $S_t = \text{IDENTIFYSLICE}(L, t)$
04. $L' = \text{SLICEOVERLAPGRAPH}(L)$
05. $\text{BFS}(L')$
06. $L'' = \text{RETRACE}(S_s, S_t)$
07. $\text{BFS}(u_s, u_t, L'')$
08. $\pi_{st} = \text{RETRACE}(u_t, u_s)$

This is to be noted that this algorithm FINDSIP never backtracks during the traversal in Step 7. This is due to the underlying logic in the BFS traversal method. BFS starts from u_s and extends in breadth till u_t is visited. Each visited UGC gets a label during traversal which gives its shortest distance from the source. BFS starts from u_s so its label is 0. As no UGC will be visited after u_t so no UGC will ever get a label greater than that of u_t . Hence there will be no possibility of backtracking during the traversal.

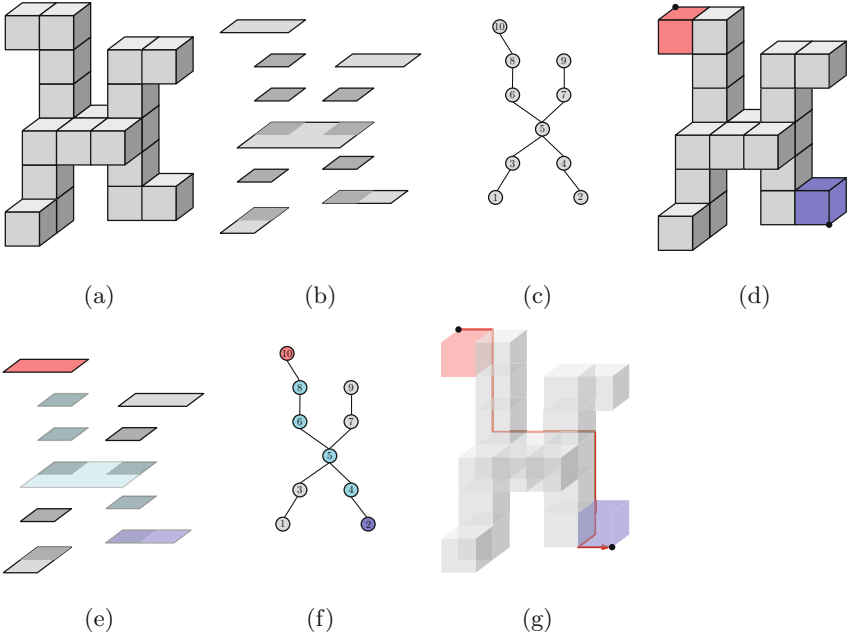


Fig. 10. Demonstration of various steps in finding SIP. (a) Object represented as a set of UGCs; (b) Object slices along zx -plane. Each of the slices are shown in light grey colour and overlapping regions are shown in dark grey. (c) Corresponding slice overlap graph; (d) Source and destination points with their corresponding UGCs; (e) Source and destination slices are marked red and blue respectively and the set of slices in the path from source to destination are marked light blue; (f) The minimum set of nodes in the path from source to destination node of the slice overlap graph are shown in light blue; (h) Final SIP is shown in red. (Color figure online)

6 Time Complexity

Slicing takes $O(n/g)$ time in best case and $O(ng)$ time in worst case, where n is the total number of voxels constituting the object surface in 26-neighborhood. Therefore, we can consider the complexity of slicing in general as $O(n)$, if $n \gg g$.

The procedure IDENTIFYSLICE, in the worst case, traverses edges of all the slices on the plane containing a given point. Therefore, the procedure needs

$O(s_{max}e_{max})$ time, s_{max} is the maximum possible count of slices on a slicing plane and e_{max} is the maximum possible count of edges of a slice.

In procedure SLICEOVERLAPGRAPH the boundary grid vertices of all the slices are visited twice, first time for labeling and second time for checking overlap among the slices. Hence, in this procedure all the grid vertices on the whole object surface are visited twice. The number of grid vertices on an object surface is always in the order of the total number of UGCs constituting that object surface. For an object with n voxels the number of UGCs in it, in best case is $O(n/g^3)$ and in worst case $O(n/g)$. Therefore, the complexity of the procedure SLICEOVERLAPGRAPH becomes $O(n/g^3)$ in best case and $O(n/g)$ in worst case.

The procedure BFS in line 5 of the algorithm FINDSIP starts from the source node of the slice overlap graph and for each node its adjacent nodes are also visited, thereby requiring $O(mk)$ time, where k is total number of slices of the object and m is the maximum possible number of nodes adjacent to a node in the slice overlap graph.

BFS in line 7 visits each UGC bounded by the slices $\in \Pi_{st}$ and for each UGC it visits its 26-adjacent UGCs which are bounded by any of the slices $\in \Pi_{st}$. Hence it requires $O(\sum_{i=1}^k y_i)$ time, where y_i denotes the number of UGCs in the i^{th} slice of L'' .

Therefore the total time complexity is given by $TC = O(n) + 2 \times O(s_{max}e_{max}) + O(n/g) + O(mk) + O(\sum_{i=1}^k y_i)$. As n is the number of voxels on the object surface hence it will have value less than $\sum_{i=1}^k y_i$ for most of the objects. Similarly $O(n/g)$ will also have value less than $\sum_{i=1}^k y_i$. $s_{max}e_{max}$ can never exceed the value of $\sum_{i=1}^k y_i$ and the number of slices of an object will always be much less than the number of UGCs on a set of slices, hence $mk < \sum_{i=1}^k y_i$. Therefore, $TC = O(\sum_{i=1}^k y_i)$ and in worst case $O(\sum_{i=1}^k y_i) = O(N)$. Though the worst case time complexity is high still the algorithm is computationally very fast in best and average cases due to the process of doing BFS only on the UGCs of the selected set of slices. An improvement in terms of complexity can be done using Hadlock's [4] method of traversal (it has to be extended to 3D). This can be considered as a future scope of work.

7 Results and Conclusion

The proposed algorithm has been implemented in C in Linux Fedora Release 7, Kernel 2.6.21.1.3194.fc7, Dual Intel Xeon Processor 2.8 GHz, 800 MHz FSB. It has been tested on several 3D objects for slicing along zx-plane. Some test results are presented in Fig. 11. Some test results found by our algorithm and the results found by doing only BFS to find the SIP between two points has been presented in Fig. 12. For better understanding the UGCs visited by both the methods has been marked blue. The count of voxels visited by each of the methods as well as the percentage of voxels traversed with respect to the total number of object voxel and the respective average CPU times (in milliseconds) for each of the methods has been given under each figure of second and third

columns. For the type of cases shown in first four rows of Fig. 12 a significant reduction in the visited voxel count by the proposed algorithm can be observed. However for the type of cases shown in the last row of Fig. 12 worst case occurs, so the visited voxel count is not much reduced by the proposed algorithm.

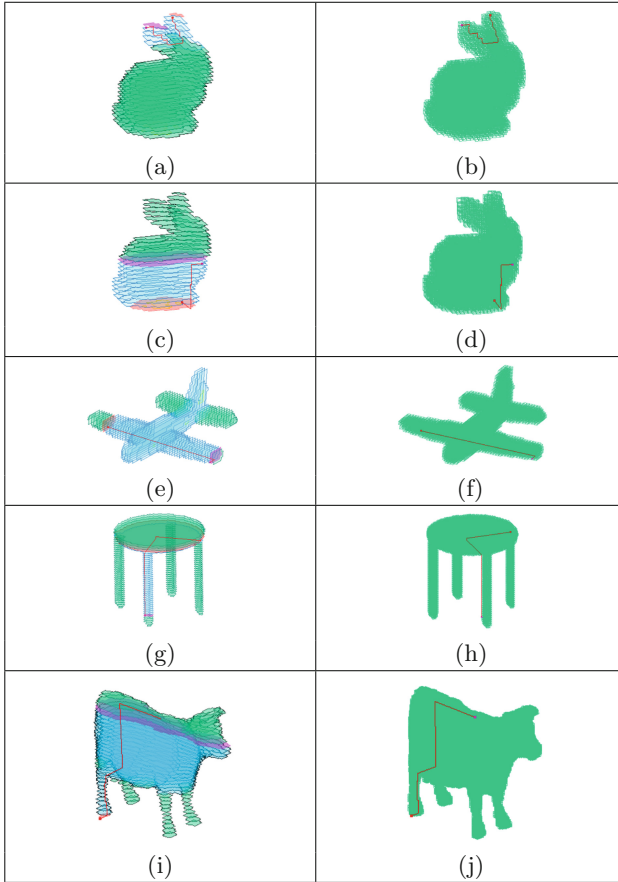


Fig. 11. SIP found by our algorithm on different objects. Left: Object represented as a set of slices. Right: Object represented as a set of UGCs. (a, b, c, d) SIP found in bunny for $g = 2$; (e, f) SIP found in aeroplane for $g = 2$; (g, h) SIP found in table for $g = 2$; (i, j) SIP found in cow for $g = 2$; Here the source and destination points and the corresponding slices are marked in red and pink respectively and the set of slices $\in \Pi_{st}$ are marked blue and the final SIP is shown in red. (Color figure online)

Hence it can be noticed that the algorithm proposed in this paper can give much reduction in the count of visited voxels for best and average cases. This algorithm finds SIP between any pair of points of a 3D digital object of type genus 0. The proposed method can be improved for application on objects of type

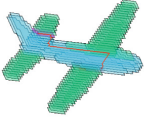
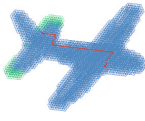
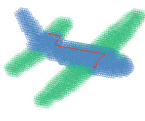
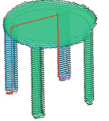
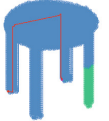
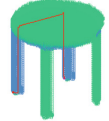
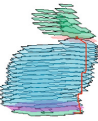

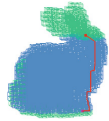
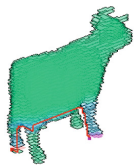
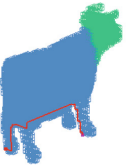
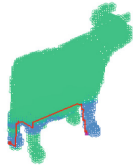

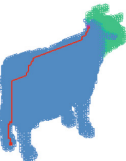

		
Object: Aeroplane ($g = 2$). Total number of voxels = 31720		
		
Object: Table ($g = 2$). Total number of voxels = 111304		
		
Object: Bunny ($g = 2$). Total number of voxels = 33392		
		
Object: Cow ($g = 4$). Total number of voxels = 782592		
		
Object: Cow ($g = 4$). Total number of voxels = 782592		

Fig. 12. Left: SIP found by algorithm FINDSIP on different objects where the objects are represented as set of slices. The source and destination points and the corresponding slices are marked in red and pink respectively and the set of slices $\in \Pi_{st}$ are marked blue and the final SIP is shown in red. Middle: SIP found by doing BFS starting from the source UGC to the destination UGC. Right: SIP found by algorithm FINDSIP. (Color figure online)

genus 1 or more. Some improvement in terms of complexity can be considered as future scope of work. An algorithm for finding multiple SIPs on a 3D object for a given set of control points remains an open problem.

Acknowledgement. This research is funded by All India Council for Technical Education, Government of India.

References

1. Bajaj, C.: An efficient parallel solution for Euclidean shortest path in three dimensions. In: IEEE International Conference on Robotics and Automation, Proceedings, vol. 3, pp. 1897–1900. IEEE (1986)
2. Dutt, M., Biswas, A., Bhowmick, P., Bhattacharya, B.B.: On finding shortest isothetic path inside a digital object. In: Barneva, R.P., Brimkov, V.E., Aggarwal, J.K. (eds.) IWCI 2012. LNCS, vol. 7655, pp. 1–15. Springer, Heidelberg (2012). doi:[10.1007/978-3-642-34732-0_1](https://doi.org/10.1007/978-3-642-34732-0_1)
3. Fitch, R., Butler, Z., Rus, D.: 3D rectilinear motion planning with minimum bend paths. In: 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems, Proceedings, vol. 3, pp. 1491–1498. IEEE (2001)
4. Hadlock, F.: A shortest path algorithm for grid graphs. *Networks* **7**(4), 323–334 (1977)
5. Jiang, K., Seneviratne, L.D., Earles, S.: Finding the 3D shortest path with visibility graph and minimum potential energy. In: Proceedings of the 1993 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 1993, vol. 1, pp. 679–684. IEEE (1993)
6. Karmakar, N., Biswas, A., Bhowmick, P.: Fast slicing of orthogonal covers using DCEL. In: Barneva, R.P., Brimkov, V.E., Aggarwal, J.K. (eds.) IWCI 2012. LNCS, vol. 7655, pp. 16–30. Springer, Heidelberg (2012). doi:[10.1007/978-3-642-34732-0_2](https://doi.org/10.1007/978-3-642-34732-0_2)
7. Khouri, J., Stelson, K.A.: An efficient algorithm for shortest path in three dimensions with polyhedral obstacles. In: American Control Conference, pp. 161–165. IEEE (1987)
8. Klette, R., Rosenfeld, A.: *Digital Geometry: Geometric Methods for Digital Picture Analysis*. Elsevier, Amsterdam (2004)
9. Lee, D.T.: Rectilinear paths among rectilinear obstacles. In: Ibaraki, T., Inagaki, Y., Iwama, K., Nishizeki, T., Yamashita, M. (eds.) ISAAC 1992. LNCS, vol. 650, pp. 5–20. Springer, Heidelberg (1992). doi:[10.1007/3-540-56279-6_53](https://doi.org/10.1007/3-540-56279-6_53)
10. Lin, C.W., Huang, S.L., Hsu, K.C., Lee, M.X., Chang, Y.W.: Multilayer obstacle-avoiding rectilinear steiner tree construction based on spanning graphs. *IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst.* **27**(11), 2007–2016 (2008)
11. Lu, J., Diaz-Mercado, Y., Egerstedt, M., Zhou, H., Chow, S.N.: Shortest paths through 3-dimensional cluttered environments. In: 2014 IEEE International Conference on Robotics and Automation (ICRA), pp. 6579–6585. IEEE (2014)
12. O’Rourke, J.: *Computational Geometry in C*. Cambridge University Press, Cambridge (1998)
13. Wagner, D.P., Drysdale, R.S., Stein, C.: An $O(n^5/2\log n)$ algorithm for the rectilinear minimum link-distance problem in three dimensions. *Comput. Geom.* **42**(5), 376–387 (2009)