Isosurface Generation by Using Exterma Graphs

Takayuki ITOH Koji KOYAMADA
Tokyo Research Laboratory, IBM Japan

Abstract

A high-performance algorithm for generating isosurfaces is presented. In this algorithm, extrema points in a scalar field are first extracted. A graph is then generated in which the extrema points are taken as nodes. Each arc of the graph has a list of IDs of the cells that are intersected by the arc. A boundary cell list ordered according to cells' values is also generated. The graph and the list generated in this pre-process are used as a guide in searching for seed cells. Isosurfaces are generated from seed cells that are found in arcs of the graph. In this process isosurfaces appear to propagate themselves. The algorithm visits only cells that are intersected by an isosurface and cells whose IDs are included in cell lists. It is especially efficient when many isosurfaces are interactively generated in a huge volume. Some benchmark tests described in this paper show the efficiency of the algorithm.

1 Introduction

Three-dimensional numerical simulations have become popular due to the increased computing power available in engineering and scientific applications. Three-dimensional measurement has also become widely used, as a result of new technologies in the medical field. Volume visualization has consequently become more important, because it allows users to understand the results of these simulations and measurements. So that the results can be understood easily through interactive operations, various methods for representing spatial numerical data are implemented in visualization tools. Isosurfaces are expected to be one of the most effective media for representing scalar fields.

An isosurface is defined as a set of points that satisfy the following equation:

$$S(x, y, z) - C = 0,$$

where $S(x, y, z)$ is a spatial function such as temperature or pressure, and C is a constant value.

An isosurface is usually approximated as a set of triangle facets [1, 2]. It is displayed as a set of edges of triangles, or as a set of drawn triangles.

In conventional straightforward methods, the cost of generating an isosurface is estimated at $O(n)$ when all cells are visited in order to search for intersections with the surface. These methods may be wastely for huge volumes, since the number of intersected cells is regarded as $O(n^{2/3})$. Here, n means the number of cells. The high cost may even prevent understanding of the distribution of the scalar field, because of the long time taken to generate a surface from a huge volume of data. Interactive operation (e.g. through a probing interface [3]) may also be difficult because of the high cost. In some tools, geometric data on the generated surfaces are held in storage for prompt repetition of the continuous display. However, these tools require a large amount of storage for huge volumes of data, and therefore some tools may refuse to handle such data.

Various efficient algorithms that eliminate non-intersected cells have been proposed. These algorithms are particularly effective for huge volumes. Some efficient algorithms generate cell lists ordered by cell's values as a pre-process [4, 5]. These algorithms have been also applied to isosurfaces. However, the number of cells to be visited is regarded as $O(n)$ in their algorithms. Other such algorithm has been proposed by Speray and Kennon [6]. Their algorithm eliminate all non-intersected cells outside the process. However, their algorithm has been applied only to slice surfaces. Moreover, some intersected cells may not be visited in case of non-convex volumes.

2 Related work

2.1 Triangulation algorithm

Suppose that the scalar values $S(x, y, z)$ are assigned at each grid-point, and that the scalar field inside a cell is linear. Intersections between an isosurface and the edges of a cell are found by extracting edges for which the signs of the difference between the given scalar value and $S(x, y, z)$ at their two ends are different. Several triangles are formed inside the cell by connecting these intersections. An isosurface is approximated as a set of triangles by the above process for all cells [1, 2] (see Figure 1).

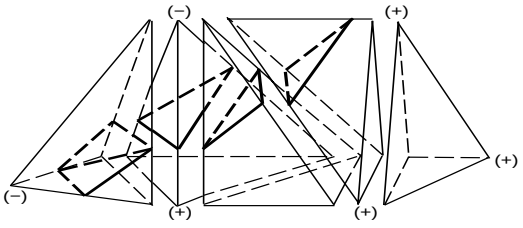

Figure 1: Triangulation

A set of triangular facets is held in the structure

shown in Figure 2.

```
typedef struct SURFACE {
  int numT; /* Number of Triangles */
  TRIANGLE tgl[]; /* Triangle Data */
  int numV; /* Number of Vertices */
  VERTEX vtx[]; /* Vertex Data */
} SURFACE;
typedef struct TRIANGLE {
  int vID[3]; /* VertexIDs */
} TRIANGLE;
typedef struct VERTEX {
  float pos[3]; /* PositionData(x,y,z) */
  float nml[3]; /* NormalVectorData(x,y,z) */
} VERTEX;
```

Figure 2: Structure of a surface

In the method proposed by Doi and Koide [2], a unique vertex identifier process is also proposed. This process avoids duplication of vertices, and thus reduces the required storage capacity and the cost of calculating vertex data.

Generally, cells that are intersected by an isosurface are only a part of a volume. These straightforward triagulation algorithms involve wasteful processes for visiting non-intersected cells.

2.2 High-performance triangulation algorithms

Recently, some efficient algorithms that eliminate non-intersected cells have been reported.

Giles and Haimes have reported a sorting algorithm [4], in which two ordered cell lists are formed in a preprocess by sorting the cells' maximum values and minimum values. The maximum value of the difference between the maximum and minimum values in a cell is also calculated. When a scalar value is specified, an active list that purges all non-intersected cells is created by referring to the two ordered lists. Only cells in an active cell list are then visited in order to generate a surface.

Gallagher has reported a filtering algorithm [5]. In his algorithm, cells whose IDs are consecutive and whose minimum values are also close are grouped. All groups are inserted into span lists that are classified according to the minimum values. Only cells in the specified span lists are visited in order to generate an isosurface.

Generally, number of intersected cells is regarded as $O(n^{2/3})$. Therefore, algorithms that visits $O(n2/3)$ cells are required for high-performance generation. In Giles's method, number of visited cells for creating a cell list is regarded as $O(n)$. In Gallagher's method, number of cells in each span lists are regarded as $O(n)$. Their algorithms may be costly for huge volumes.

Speray and Kennon have reported a propagation algorithm for generating slice surfaces [6]. Their algorithm does not require any heavy pre-processes. On the other hand, it uses cell adjacency to propagate itself through the faces of cells. It requires a data structure in which the IDs of adjacent cells can be specified from each cell. For an unstructured volume, a data structure that has the IDs of adjacent cells for each cell [7] is required.

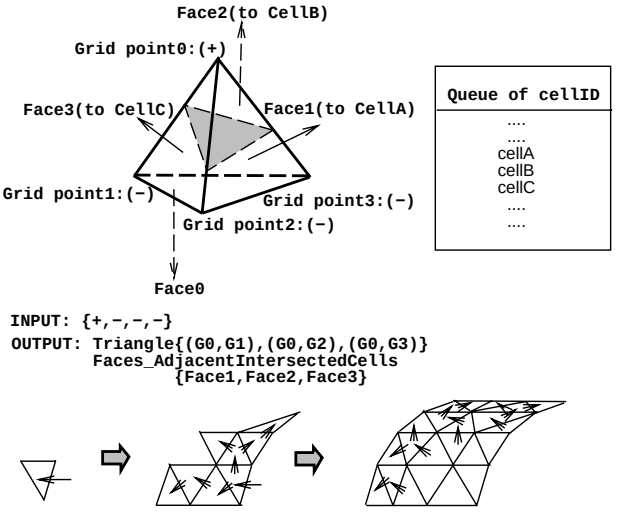

Figure 3: Propagation

An edge of a triangle exists on the faces of a cell that have both lower-valued grid-points and higher-valued grid-points. The adjacent cell that shares the face is obviously intersected by the slice surface. In Speray and Kennon's algorithm, the IDs of adjacent intersected cells are put into a FIFO queue. IDs are then dequeued and the cells are then visited in that order. A set of triangles is efficiently generated by repeating this process until the FIFO queue is empty. This algorithm visits only intersected cells (see Figure 3).

The propagation algorithm is shown in Figure 4.

```
void PropagateSurface(int seedcellID, float C){
  Enqueue the seedcellID;
  while( the queue is not empty ){
    dequeue a cellID;
    Make triangles in the dequeued cell;
    Enqueue the cellIDs of
        unmarked adjacent intersected cells;
    Mark the enqueued cells;
  }
  for(each vertex) Calculate data;
}
```

Figure 4: Propagation algorithm

The algorithm requires manual operations to specify seed cells, and is therefore not useful for visualization tools that continuously display surfaces without any manual specification.

However, it has a great advantage in efficiency, since it eliminates all non-intersected cells outside the pro-

cess. It is very useful with an intuitive interface that helps to specify the position or scalar value [3].

Speray and Kennon did not use their algorithm for generating isosurfaces. We guess that this is because isosurfaces have a higher possibility of separation. Their algorithm requires many manual specifications if the surface is separated into several parts.

3 Algorithm

The method proposed in this section is an expansion of Speray and Kennon's propagation algorithm. Their efficient algorithm for generating slice surfaces is used to generate isosurfaces without any manual specification of seed cells.

Our algorithm is shown in Figure 5.

```
void main(){

  /* Begin Pre-process */
  ExtractExtrema();
  GenerateGraph();
  GenerateBoundList();
  /* End Pre-process */

  /* Begin Main-process */
  while( 1 ){
    Specify the scalar value C;
    GenerateSurface(C);
  }
  /* End Main-process */

}
```

Figure 5: Outline of our algorithm

Here, the function ExtractExtrema(), GenerateGraph(), GenerateBoundList(), and MainProcess() are shown in Figures 7, 12, 14, and 16.

As a pre-process, extrema points in a scalar field are extracted, and then connected by a graph whose arcs have a cell list that contains the IDs of cells intersected by the graph. At the same time, a boundary cell list ordered according to scalar value is generated. Seed cells can be found in the cell list of arcs or boundary cell list, without any manual specification. Isosurfaces propagate themselves, starting from the seed cells.

Our algorithm is based on the following rule:
Rule: Any given closed isosurface is intersected by at least one arc of a graph in which all extrema points are connected by arcs. *(unless there is no interior point in each area, or the scalar field is flat in each area).*
Any given open isosurface is intersected by at least one boundary cell.

3.1 Extracting extrema points

In this paper, extrema points are defined as grid-points whose scalar values are maximum or minimum in all cells that share them.

The extrema points are held in the structure shown in Figure 6.

```
typedef struct EXTRM-PNT {
  int gID;
    /* grid-pointID of the extrema point */
  int flag;
    /* flag for checking its connectivity */
} EXTRM-PNT;
typedef struct EXTRM-LST {
  int numE;
    /* Number of extrema points */
  EXTRM-PNT extrm[];
    /* Extrema point data */
} EXTRM-LST;
```

Figure 6: Structure of an extrema point

To extract extrema points, the scalar values of all grid-points for each cell are compared. All grid-points except the maximum-valued grid-points are marked as "Not maximum." Similarly, all grid-points except the minimum-valued are marked as "Not minimum." Intermediate valued grid-points are marked as both. After values in all cells have been compared, only grid-points that have either a "Not maximum" or "Not minimum" mark are extracted as extrema points.

The algorithm for extracting extrema points is shown in Figure 7.

```
void ExtractExtrema(){
  for(each cell){
    mark ''Not maximum'' valued grid-points;
    mark ''Not minimum'' valued grid-points;
    }
  }
  for(each grid-point){
    Insert the ID into the extrema list
      that is not marked
      either ''Not maximum'' or ''Not minimum'';
  }
}
```

Figure 7: Algorithm for extracting extrema points

3.2 Generating an extrema graph

In this paper, extrema graphs are defined as a group of arcs that connects two extrema points.

The graph is held in the structure shown in Figure 8.

To generate an arc, an extrema point is first chosen as a "start" point. The other extrema point is then chosen as a "goal" point. Here, we assume that the group of arcs that minimize the total cost of a graph is generated when closer extrema points are connected by a graph. (Here the cost of a graph means the number of cells inserted into the list.) In accordance with this

```
typedef struct ARC {
  int eID[2];
    /* IDs of two extrema points */
  float value[2];
    /* The maximum value and minimum value */
  int numC;
    /* Number of cells included in the cell list */
  int cID[];
    /* Intersected cell list */
} ARC;

typedef struct GRAPH {
  int numN;
    /* Number of nodes (=extrema points) */
  int node[];
    /* Grid-pointID of nodes (=extrema points) */
  ARC arc[];
    /* arc data */
} GRAPH;
```

Figure 8: Structure of a graph

assumption, several closer extrema points are enqueued as a candidate of the "goal" point.

One of the closer extrema points is selected as a "goal" point, and the vector of the arc between the start point and the goal point is then calculated. Starting from a cell that includes the start point, the arc is traversed and the adjacent cell that is intersected by the arc is then visited. This process is repeated until it arrives at the cell that includes the goal point. Here, the maximum and minimum values are updated by the scalar value of each grid-point during visits to cells. The IDs of the visited cells are inserted into the cell list of the traversed arc. If the line goes outside the volume, the traverse is terminated and a similar traverse is started after selection of the next-closest extrema point as a "goal" point (see Figure 9).

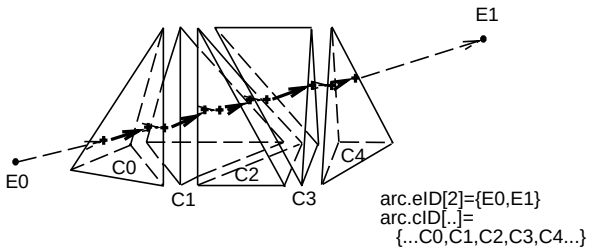

Figure 9: Traversing along an arc

If no extrema points are connected with the start point, the closest extrema point among those whose flags' values are not equal is chosen as a goal point. Starting from a cell that includes the start point, the adjacent cells that share the face whose sum is minimum are visited in order. For each visited cell, the distance to the goal point from each grid-point is calculated. The sum of the distance values is then calculated for each face of the visited cell. This process is repeated until it arrives at a cell that includes the goal extrema point. The IDs of the visited cells are inserted into the cell list of the arc (see Figure 10). This distance-based process necessarily connects the chosen extrema points, but it is not always efficient, because of the cost of calculating the distance for each grid-point. Therefore it should be the second process after traversal of straight arcs.

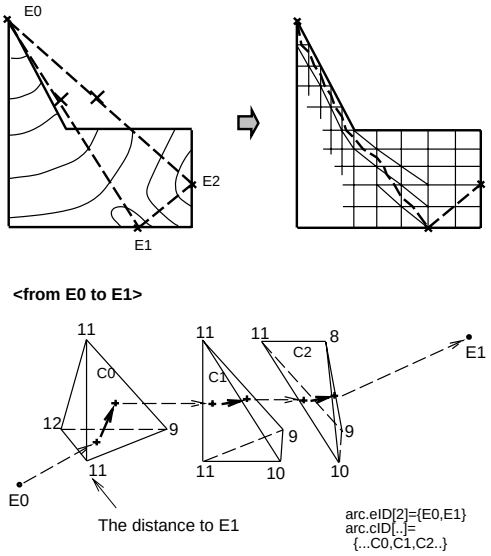

Figure 10: Traversing by calculating distance

In our implementation, each extrema point has a flag to be checked for connectivity. At first, the flag of each extrema point is substituted with its grid-pointID. When an extrema point is connected with another extrema point, the values of the two flags are compared. The flag that has the larger value is substituted with the smaller value. In addition, flags of other extrema points that have the larger value are substituted with the smaller value. When the start point is chosen, an extrema point whose flag has an equal value to the flag of the start point is not chosen as a goal point (see Figure 11).

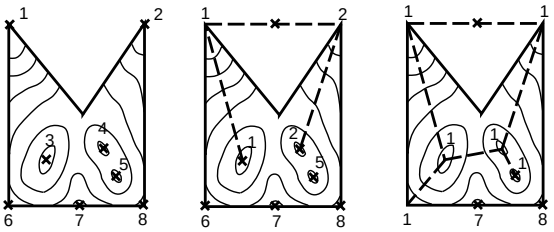

Figure 11: Checking connectivity of extrema points

The algorithm for generating an extrema graph is shown in Figure 12. Here, "adjcell" means an adjacent cell.

```
void GenerateGraph(){
  for(each extrema i) extrm[i].flag = extrm[i].gID;
  for(each extrema n) {
    for(each other extrema i) {
      if(extrm[n].flag != extrm[i].flag) {
        if(extrm[i] is NOT too far) Enqueue extrm[i];
      }
    }
    while(the arc has not been connected) {
      Dequeue an extrema extrm[m];
      if(MakeGraph1(extrm[n], extrm[m]) == SUCCESS) {
        break;
      }
      if(the extrema-points-queue is empty){
        MakeGraph2(extrm[n], extrm[m]); break;
      }
    }
    for(each extrema i)
      if(extrm[i].flag == extrm[m].flag)
        extrm[i].flag = extrm[n].flag;
  }
}

int MakeGraph1(extrmA, extrmB){
  VisitCell = A cell that includes extrmA;
  while(1){
    arc.cID[(arc.numC++)] = VisitCell;
    Update arc.value[0] and arc.value[1];
    if(VisitCell includes extrmB) return(SUCCESS);
    VisitCell = the adjcell that intersects the arc;
    if(VisitCell = NULL) return(FAILURE);
  }
}

void MakeGraph2(extrmA, extrmB){
  VisitCell = A cell that includes extrmA;
  while(1){
    arc.cID[(arc.numC++)] = VisitCell;
    Update arc.value[0] and arc.value[1];
    if(VisitCell includes extrmB) return;
    VisitCell = the adjcell shares the nearest face;
  }
}
```

Figure 12: Algorithm for generating graphs

3.3 Generating boundary cell lists

In this paper, boundary cells are defined as a group of cells that have faces not connected to an adjacent cell. First, maximum and minimum values are defined for each boundary cell. Two ordered lists are then generated by using minimum and maximium values of the cells based on a quick-sort algorithm. These lists have pointers to each boundary cell data.

Boundary cells are held in the structure shown in Figure 13.

The algorithm for generating boundary cell lists is shown in Figure 14.

```
typedef struct BCELL {
  float min; /* minimum value */
  float max; /* maximum value */
} BCELL;
typedef struct BLIST {
  int numB; /* No. of boundary cells */
  BCELL bcell[];
  BCELL *bcell-min[]; /* Ordered by minimum values */
  BCELL *bcell-max[]; /* Ordered by maximum values */
} BLIST;
```

Figure 13: Structure of boundary cell lists

```
void GenerateBoundList(){
  for(each boundary cell cID[i]) {
    define cID[i].max and cID[i].min;
  }
  Generate a minimum value-based boundary cell list;
  Generate a maximum value-based boundary cell list;
}
```

Figure 14: Algorithm for generating boundary cell lists

3.4 Generating isosurfaces

Our algorithm generates an isosurface by traversing an adjacency cell list, starting from seed cells. The main point of our algorithm is to extract seed cells automatically and efficiently. An isosurface is generated when a scalar value is specified. Seed cells are searched for by traversing a boundary cell list and all arcs of the extrema graphs, in order to propagate an isosurface.

First, boundary cells are visited in order of the minimum-value-based cell list until the minimum value becomes higher than the given value. If the maximum value of the visited cell is higher, the cell is regarded as a seed cell. A maximum-value-based cell list can also be usede for visiting cells in order.

Next, seed cells are searched for by traversing arcs of the extrema graphs. The specified value and the maximum and minimum values of each arc are compared. If the specified value lies between the maximum and minimum values of an arc, all the cells in the list are visited in order.

When an extracted seed cell is unmarked, its ID is put into the FIFO queue and an isosurface is then generated (see Figure 15).

This process is shown in Figure 16. Here, the function PropagateSurface() is shown in Figure 4.

Here, the number of intersected cells is regarded as $O(n^{2/3})$. The number of cells in arcs of extrema graphs is regarded as $O(n^{1/3})$. The number of cells in boundary cell lists is regarded as $O(n^{2/3})$. Therefore, the cost of generating an isosurface in our algorithm is estimated as $O(n^{2/3})$. Our algorithm is thus especially efficient for huge volumes.

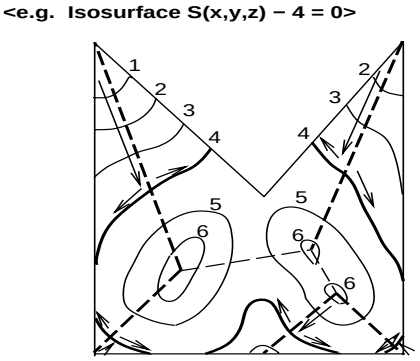

Figure 15: Searching for seed cells

```
void GenerateSurface(float C){
  for(each cell bcell[i] in the boundary cell list) {
    if( bcell[i].min < C && bcell[i].max > C &&
        bcell[i] is unmarked ) {
      PropagateSurface( bcell[i],C );
  }
  for(each arc){
    if(arc.value[1] < C < arc.value[0]) {
      for(each unmarked cell in the arc){
        if(cell cID[i] is intersected)
          PropagateSurface( cID[i],C );
      }
    }
  }
}
```

Figure 16: Algorithm for generating isosurfaces

4 Benchmark tests

Our method has been implemented and tested on an IBM PowerStation RS/6000 (Model 560). An example of an extrema graph is shown in Figure 17, and 18, where white "+" marks show extrema points. An example of isosurfaces are shown in Figure 19, and 20.

Here we analyze our algorithm and the straightforward algorithm [2] by comparing their performance. In benchmark tests, a series of 20 isosurfaces were generated for each volume. The results for example unstructured volumes are as follows.

The size of the volumes are shown in Table 1.

Table 1: Size of volume

Dataset No.	1	2	3	4
N_{tc}	20736	61680	346644	557868
N_{bf}	2048	5432	16908	97473
N_{gp}	4002	11624	62107	17876

where

- N_{tc} is the number of tetrahedral cells (including boundary cells).
- N_{bc} is the number of boundary tetrahedral cells.
- N_{gp} is the number of grid points.

The cost of pre-processing is shown in Tables 2.

Table 2: Performance in the pre-process

Dataset No.	1	2	3	4
T_{pe}	0.541	1.382	6.290	9.719
T_{pc}	0.062	0.166	0.600	4.151
T_{pb}	0.081	0.275	1.138	1.359
N_e	21	46	135	540
N_g	468	1048	3250	10210

where

- T_{pe} is the total time for extracting extrema points.
- T_{pc} is the total time for connecting extrema by a graph.
- T_{pb} is the total time for creating ordered boundary cell lists.
- N_e is the number of extrema points.
- N_g is the number of cells in the list of arcs.

The above results show that the simple process for extracting extrema points is much more costly than other processes, since its cost is estimated as $O(n)$.

The cost T_{pc} of dataset No.4 is much higher that those of other datasets. We suppose that this result is due to the cost for selecting close extrema points, since its cost is estimated as $O(n^2)$. If the scalar value is noisy and there are enoumous extrema points, the cost T_{pc} may be much higher than that of dataset No.4.

The cost of the main process is shown in Tables 3,

Table 3: Performance in the main process

Dataset No.	1	2	3	4
T_{mg} (sec.)	0.008	0.088	0.248	0.197
T_{mb} (sec.)	0.019	0.032	0.091	0.144
T_{mt} (sec.)	1.471	1.748	4.495	8.331
T_{mv} (sec.)	0.567	0.693	1.690	3.114
T_1 (sec.)	2.065	2.561	6.524	11.786
T_2 (sec.)	5.068	11.416	63.113	103.072
R_{12}	2.454	4.458	9.674	8.745
N_{cg}	3063	4774	15659	14196
N_{ci}	51357	62377	148836	277457
N_{cc}	414720	1233600	6932880	11157360
N_v	34921	33874	103742	185622

where

- T_{mg} is the total time spent searching for seed cells in arcs.
- T_{mb} is the total time spent creating an active boundary cell list.

- T_{mt} is the total time spent forming triangles.
- T_{mv} is the total time spent calculating vertex data.
- T_1 is the total time of our method.
- T_2 is the total time of the conventional method.
- R_{12} is the ratio of T_2 to T_1.
- N_{cg} is the number of cells visited in searching for seed cells.
- N_{ci} is the number of cells intersecting isosurfaces.
- N_{cc} is the number of cells visited in the conventional method.
- N_v is the number of vertices of surfaces.

These results show that R_{12} is better for larger volumes than for smaller volumes, indicating that our method is especially efficient for large volumes.

The total cost of an extrema graph is not always minimum, and optimization of the graph remains as future work. However, we do not expect such optimization to contribute significantly to reducing the cost, because the cost of searching for seed cells in the arcs T_{mg} is so tiny.

Next, we will discuss the cost of each routine. In our method, the total time is estimated as

$$T_1 = t_1(N_{bc} + N_g) + (t_2 + t_3)N_{cp} + t_5 N_v,$$

In the straightforward method,

$$T_2 = t_2 N_{cc} + t_4 N_{cp} + t_5 N_v.$$

where

$$t_5 = Tv/N_v.$$

The costs are shown in Table 4.

Table 4: The costs of each variables

t_1 (sec.)	3.648×10^{-6}
t_2 (sec.)	8.233×10^{-6}
t_3 (sec.)	2.161×10^{-5}
t_4 (sec.)	2.920×10^{-5}
t_5 (sec.)	1.744×10^{-5}

where

- t_1 is the cost for searching for seed cells.
- t_2 is the cost of checking the sign of the difference between the given scalar value and the value of each grid-point.
- t_3 is the cost of making triangles and enqueueing the IDs of unmarked adjacent intersected cells, in our method.
- t_4 is the cost of making triangles in the straightforward method.
- t_5 is the cost of calculating the position and normal vector for each vertex of triangles.

To our surprise, t_3 is lower than t_4, even though it includes the cost of enqueueing and marking cells in our method. We suppose that this result is due to the efficiency of the vertex identifier process. We implemented the process by referring to the vertex data in order of newness. We suppose that the vertex identifier process in the propagation algorithm is more efficient, since the newest triangle has the tendency to share a newer vertex. The sorting algorithm [4] and the filtering algorithm [5] do not use adjacency such as in the straightforward method, and therefore these vertex identifier processes are not seemed so efficient as those in the propagation method.

5 Conclusion

In this paper we have proposed a high-performance algorithm for generating isosurfaces, by using extrema graphs and ordered boundary cell lists as a guide to search for seed cells. Our algorithm especially efficient for huge volumes, since the number of visited cells is regarded as $O(n^{2/3})$.

Acknowledgements

We would like to thank K. Shimizu, manager of Advanced Graphics at the Tokyo Research Laboratory, IBM Japan, for his encouragement in this work. We would also like to thank M. Makino, assistant professor of Chuo university, for helpful discussion about scalar fields and extrema points.

References

[1] W. E. Lorensen and H. E. Cline: "Marching Cubes: A High Resolution 3D Surface Construction Algorithm," Computer Graphics, Vol. 21, No. 4, pp. 163-169, 1987.

[2] A. Doi and A. Koide: "An Efficient Method of Triangulating Equi-valued Surfaces by Using Tetrahedral Cells," IEICE Transactions, Vol. E74, No. 1, pp. 214-224, 1991.

[3] K. Koyamada and T. Itoh: "A Measurement System for 3-D Numerical Simulation Results," IPSJ Technical Report, 93-HPC-48, 1993.

[4] M. Giles and R. Haimes: "Advanced Interactive Visualization for CFD," Computer Systems in Engineering, Vol. 1, No. 1, pp. 51-62, 1990.

[5] R. S. Gallagher, "Span Filtering: An Optimization Scheme for Volume Visualization of Large Finite Element Models," IEEE Visualization '91, pp. 68-74, 1991.

[6] D. Speray and S. Kennon, "Volume Probe: Interactive Data Exploration on Arbitrary Grids," Computer Graphics, Vol. 24, No. 5, pp. 5-12, 1990.

[7] K. Koyamada, "Visualization of Simulated Airflow in a Clean Room," IEEE Visualization '92, pp. 156-163.

Figure 17: An extrema graph

Figure 18: An extrema graph (dataset No. 1)

Figure 19: Generating isosurfaces

Figure 20: Generating isosurfaces (dataset No. 1)