Polyline Reduction

David Eberly
Geometric Tools, LLC
http://www.geometrictools.com/

Created: April 15, 2001
Last Modified: February 9, 2008

Contents

1 Introduction

This document describes a method for reducing the number of vertices in a polyline $\{\mathbf{X}_i\}_{i=0}^n$ by removing one vertex at a time based on weights assigned to the vertices. A vertex weight is a measure of variation of the polyline at the specified vertex. A simple measure of weight w_i for vertex $\mathbf{X}_i$ is based on the three consecutive vertices $\mathbf{X}_{i-1}$, $\mathbf{X}_i$, and $\mathbf{X}_{i+1}$,

$$w_i = \frac{\text{Distance}^2(\mathbf{X}_i, \text{Segment}(\mathbf{X}_{i-1}, \mathbf{X}_{i+1}))}{\text{Length}^2(\text{Segment}(\mathbf{X}_{i-1}, \mathbf{X}_{i+1}))} \tag{1}$$

where $\text{Segment}(\mathbf{U}, \mathbf{V})$ denotes the line segment from $\mathbf{U}$ to $\mathbf{V}$. The vertex that is removed is the one corresponding to the minimum weight. Observe that if the minimum weight is zero, then $\mathbf{X}_i$ is already a point on $\text{Segment}(\mathbf{X}_{i-1}, \mathbf{X}_{i+1})$. Removing zero weight points first is ideal for polyline reduction.

Special handling is required at the end points $\mathbf{X}_0$ and $\mathbf{X}_N$. The easiest thing to do is assign $w_0 = w_n = \infty$; that is, the end points are never removed. The polyline is reduced a vertex at a time until only two vertices remain, the end points. However, it is possible that $\mathbf{X}_n = \mathbf{X}_0$ in which case the polyline is closed. Assigning infinite weight to $\mathbf{X}_0$ leads to that point always occurring in a reduction. Instead, the weight formula can be applied to every vertex in a closed polyline with the understanding that the indices are selected modulo n.

Other definitions for vertex weights may be used. For example, a larger neighborhood of $\mathbf{X}_i$ might be used. Or an interpolating polynomial curve could be used to assign the curvature of that curve to each vertex. The choices are many, but the algorithm for determining the order of removal of the vertices can be developed independently of the weight definition.

The algorithm considered here just removes vertices, one at a time. The vertices of the reduced polyline form a subset of the vertices of the original polyline. This is convenient, but not necessary. If $\mathbf{X}_i$ provides the minimum weight of all vertices, it is possible to replace the triple $\langle \mathbf{X}_{i-1}, \mathbf{X}_i, \mathbf{X}_{i+1} \rangle$ by the pair $\langle \mathbf{Y}_{i-1}, \mathbf{Y}_{i+1} \rangle$ where $\mathbf{Y}_{i-1}$ and $\mathbf{Y}_{i+1}$ are quantities derived from the original triple, and possibly from other nearby vertices.

2 A Simple Algorithm

The simplest algorithm for reduction is a recursive one. Given a polyline $P = \{\mathbf{X}_i\}_{i=0}^n$, compute the weights $\{w_i\}_{i=0}^n$. Search the weights for the minimum weight w_k. Remove $\mathbf{X}_k$ from P to obtain the polyline P' that has $n-1$ vertices. Repeat the algorithm on P'. This is an $O(n^2)$ algorithm since the first pass processes n vertices, the second pass processes $n-1$ vertices, and so on, so the total number of processed vertices is $n + (n-1) + \ldots + 3 = n(n+1)/2 - 3$.

3 A Fast Algorithm

A faster algorithm is called for. All n weights are calculated on the first pass. When a vertex $\mathbf{X}_i$ is removed from P, only the weights for $\mathbf{X}_{i-1}$ and $\mathbf{X}_{i+1}$ are affected. The calculation of all weights for the vertices of P' involves many redundant computations. Moreover, if only a couple of weights change, it is not necessary to search the entire sequence of weights for the minimum value. A heap data structure can be used that supports an $O(1)$ lookup. If the heap is implemented as a complete binary tree, the minimum occurs at the root of the tree. When the minimum is removed, an $O(\log n)$ update of the binary tree is required to

convert it back to a heap. The initial construction of the heap requires a comparison sort of the weights, an $O(n \log n)$ operation.

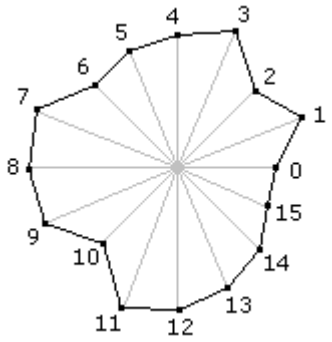
The fast reduction requires an additional operation that is not part of the classic heap data structure. The heap is initially reconstructed in $O(n \log n)$ time. The minimum value is removed and the binary tree is reorganized to form a heap in $O(\log n)$ time. The vertex removal causes a change in two weights in the heap. Once those weights are changed, the binary tree will no longer represent a heap. If we can remove the two old weights from the heap, we could then add the two new weights. Unfortunately, the classic heap data structure does not support removing an element from any location other than the root. As it turns out, if a weight is changed in the heap, the corresponding node in the binary tree can be either propagated towards the root of the tree or towards the leaves of the tree, depending on how the weight compares to the weights of its parent or child nodes. Since the propagation can be performed without changing the tree structure, this update operation is also $O(\log n)$. If the changed weight is smaller than its parent weight, the node is swapped with its parent node, thereby preserving the heap property. If the changed weight is larger than its children's weights, the node is swapper with the the child node of largest weight, thereby preserving the heap property.

Now we encounter the next complication. If a weight at an internal heap node changes, we need to know *where* that node is located to perform the $O(\log n)$ update. If we had to search the binary tree for the changed node, that operation is $O(n)$, a linear search. The only property of a minimum heap is that the weights of the two children of a node are smaller or equal to the weight of the node itself. That is not enough information for a search query to decide which child should be visited during the tree traversal, a necessary piece of information to reduce to search to $O(\log n)$. The solution to this problem is to create a data structure for each vertex in the polyline. Assuming that the binary tree of the heap is stored in a contiguous array, the vertex data structure must store the index to the heap node that represents the vertex. That index is changed whenever a heap element is propagated to its parent or to a child.

4 An Illustration

An example is given here for a 16-sided polygon with vertices $\mathbf{X}_k = A_k(\cos(2\pi k/16), \sin(2\pi k/16))$ for $0 \le k < 16$ where the amplitudes were randomly generated as $A_0 = 75.0626$, $A_1 = 103.1793$, $A_2 = 84.6652$, $A_3 = 115.4370$, $A_4 = 104.2505$, $A_5 = 98.9937$, $A_6 = 92.5146$, $A_7 = 119.7981$, $A_8 = 116.1420$, $A_9 = 112.3302$, $A_{10} = 83.7054$, $A_{11} = 117.9472$, $A_{12} = 110.5251$, $A_{13} = 100.6768$, $A_{14} = 90.1997$, and $A_{15} = 75.7492$. Figure 4.1 shows the polygon with labeled vertices.

Figure 4.1 Initial 16-sided polygon.



The min heap is stored as an array of 16 records. Each record is of the form

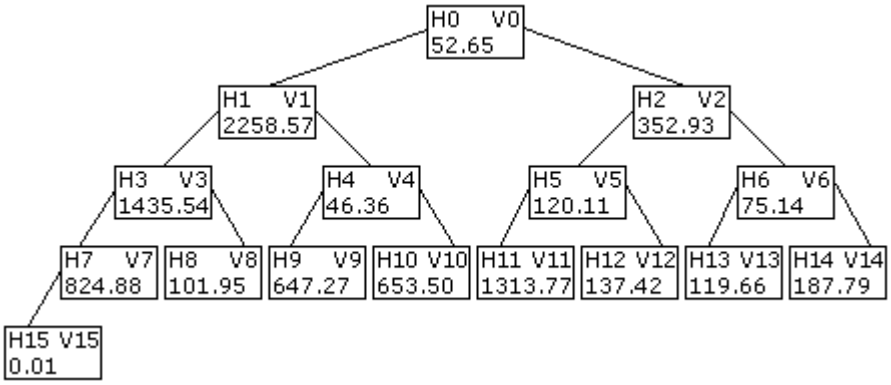
```
HeapRecord
{
    int V;          // vertex index
    int H;          // heap index
    float W;        // vertex weight (depends on neighboring vertices)
    HeapRecord* L;  // points to record of left vertex neighbor
    HeapRecord* R;  // points to record of right vertex neighbor
}
```

The vertex index and doubly linked list structure represent the polyline itself. As vertices are removed, the list is updated to reflect the new topology. The weight is the numeric value on which the heap is sorted. As mentioned earlier, the heap index allows for an $O(1)$ lookup of the heap records whose weights change because of a vertex removal. Without this index, an $O(n)$ search on the vertex indices in the heap would be necessary to locate the heap records to change.

4.1 Initialization of the Heap

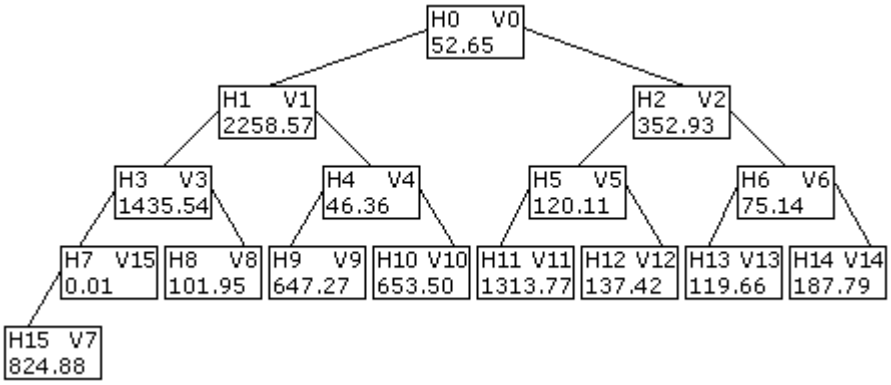
The heap records are initialized with the data from the original vertices. The vertex index and heap index are the same for this initialization. Figure 4.2 shows the heap array after initialization. The heap indices, the vertex indices, and the weights are shown. The weight of vertex $\mathbf{X}_i$ is calculated using equation (1) where the left neighbor is $\mathbf{X}_{(i-1) \bmod 16}$ and the right neighbor is $\mathbf{X}_{(i+1) \bmod 16}$.

Figure 4.2 Initial values in the heap array.



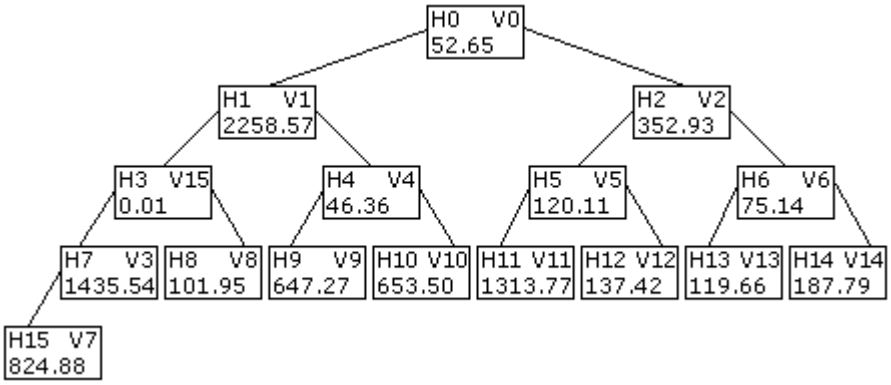
To be a min heap, each node H_i in the binary tree must have a weight that is smaller or equal to the weights of its child nodes H_{2i+1} and H_{2i+2}. The heap array must be sorted so that the min heap property at each record is satisfied. This can be done in a nonrecursive manner by processing the parent nodes from the bottom of the tree towards the root of the tree. The first parent in the heap is located. In this example, H_7 is the first parent to process. Its only child, H_{15}, has a smaller value, so H_7 and H_{15} must be swapped. Figure 4.3 shows the state of the heap array after the swap.

Figure 4.3 The heap array after swapping H_7 and H_{15} in Figure 4.2.



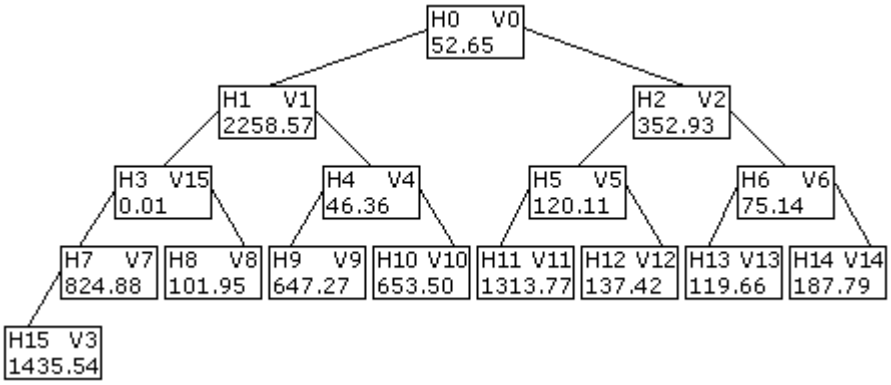
The next parent to process is H_6. The weight at H_6 is smaller than the weights of its two children, so no swapping is necessary. The same is true for the parent nodes H_5 and H_4. Node H_3 has a weight that is larger than both its children's weights. A swap is performed with the child that has smallest weight, in this case H_3 and H_7 are swapped. Figure 4.4 shows the state of the heap array after the swap.

Figure 4.4 The heap array after swapping H_3 and H_7 in Figure 4.3.



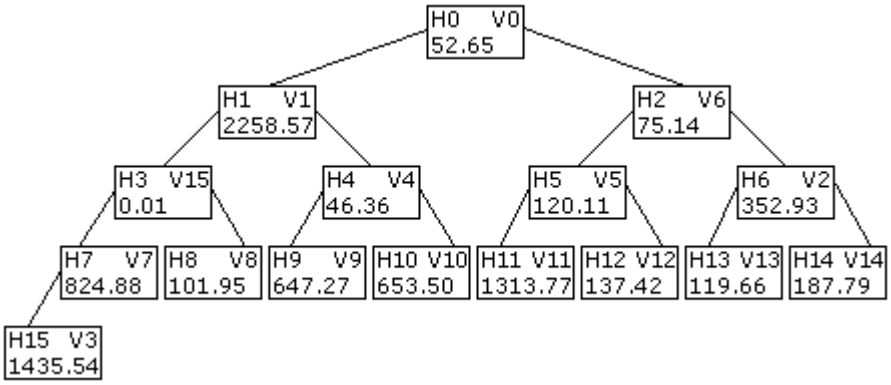
Before the swap, the subtree at the child is already guaranteed itself to be a min heap. After the swap, the worst case is that the weight needs to be propagated down a linear path in the subtree. Any further swaps are always with the child of minimum weight. In the example an additional swap must occur, this time between H_7 and H_{15}. After the swap, the processing at H_3 is finished (for now) and the subtree at H_3 is itself a min heap. Figure 4.5 shows the state of the heap array after the swap of H_7 and H_{15}.

Figure 4.5 The heap array after swapping H_7 and H_{15} in Figure 4.4.



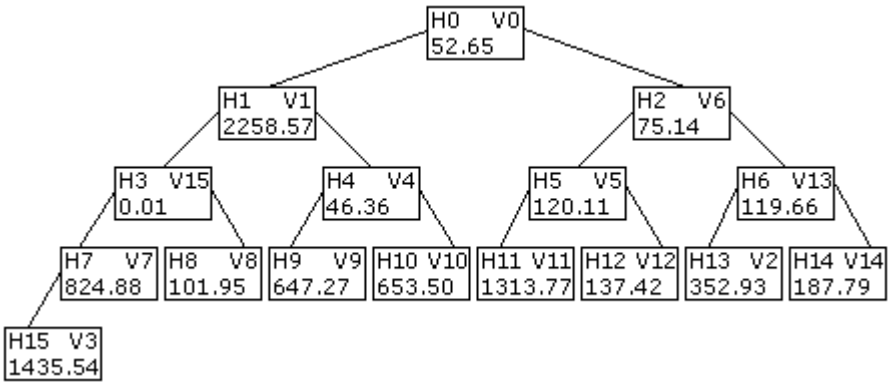
The next parent to process is H_2. The weight at H_2 is larger than the minimum weight occurring at child H_6, so these two nodes must be swapped. Figure 4.6 shows the state of the heap array after the swap.

Figure 4.6 The heap array after swapping H_2 and H_6 in Figure 4.5.



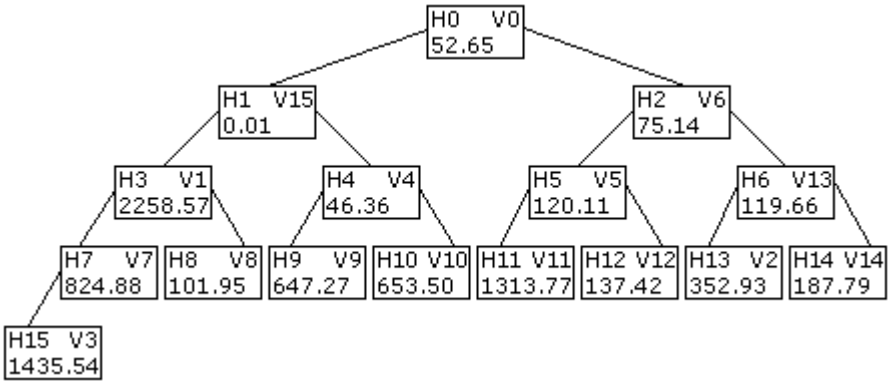
Another swap must occur, now between H_6 and the minimum weight child H_{13}. Figure 4.7 shows the state of the heap array after the swap.

Figure 4.7 The heap array after swapping H_6 and H_{13} in Figure 4.6.



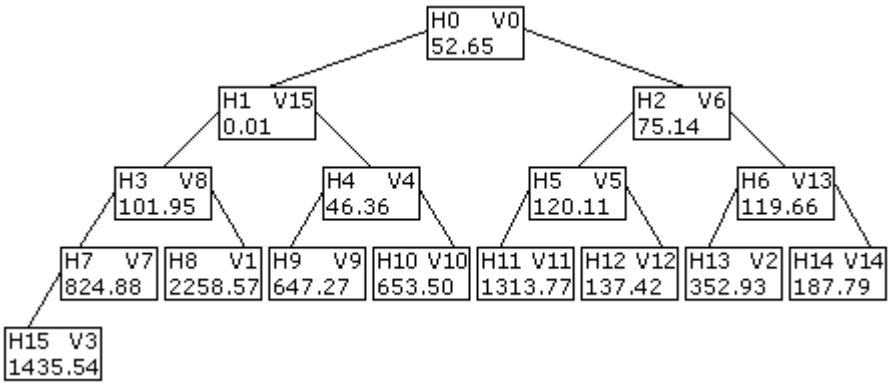
The next parent to process is H_1. The weight at H_1 is larger than the minimum weight occurring at child H_3, so these two nodes must be swapped. Figure 4.8 shows the state of the heap array after the swap.

Figure 4.8 The heap array after swapping H_1 and H_3 in Figure 4.7.



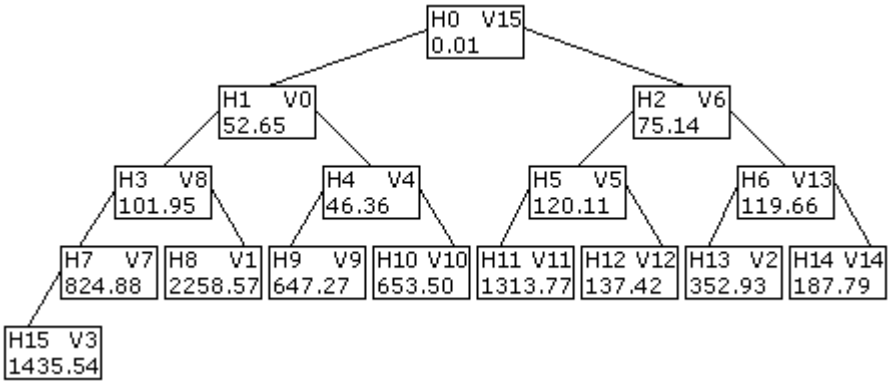
Another swap must occur, now between H_3 and the minimum weight child H_8. Figure 4.9 shows the state of the heap array after the swap.

Figure 4.9 The heap array after swapping H_3 and H_8 in Figure 4.8.



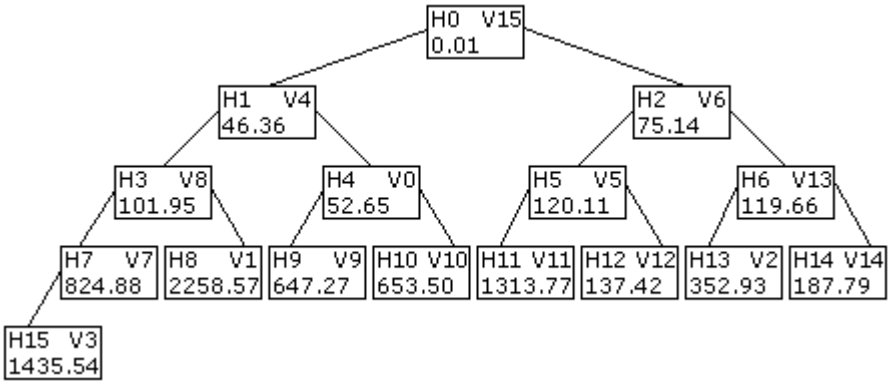
The last parent to process is H_0. The weight at H_0 is larger than the minimum weight occurring at child H_1, so these two nodes must be swapped. Figure 4.10 shows the state of the heap array after the swap.

Figure 4.10 The heap array after swapping H_0 and H_1 in Figure 4.9.



Another swap must occur, now between H_1 and the minimum weight child H_4, but no other swaps are necessary in that subtree. Figure 4.11 shows the state of the heap array after the swap.

Figure 4.11 The heap array after swapping H_1 and H_4 in Figure 4.10.

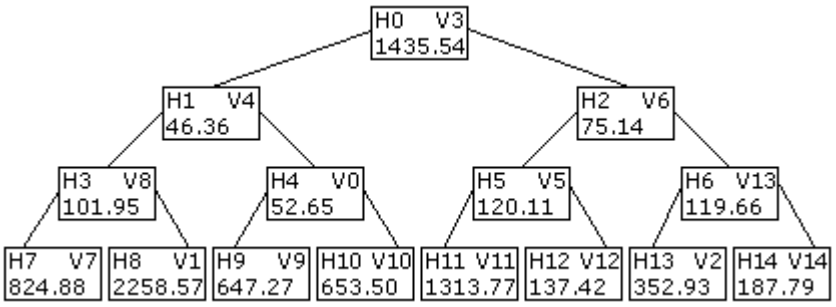


Now the heap array does represent a min heap since the children weights at each node are smaller or equal to the parent weights.

4.2 Remove and Update Operations

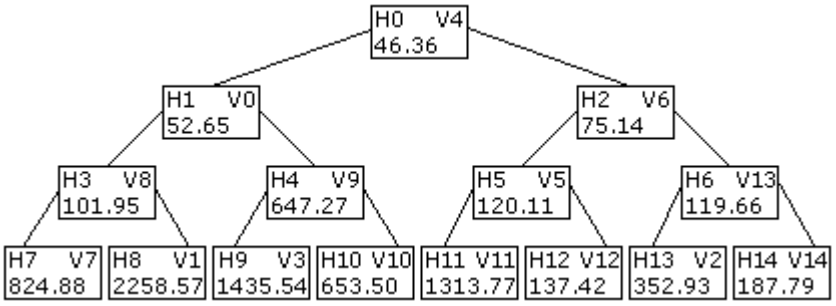
The vertex with minimum weight is the first to be removed from the polyline. The root of the heap corresponds to this vertex, so the root is removed from the heap. The vertex to be removed is V_{15}. To maintain a complete binary tree, the last item in the heap array is placed at the root location. Figure 4.12 shows the state of the heap array after moving the last record to the root position.

Figure 4.12 The heap array after removing the contents of H_0 and moving the contents of H_{15} to H_0.



The array does not satisfy the min heap property since the root weight is larger than the minimum child weight. The root node H_0 must be swapped with H_1, the child of minimum weight. The swapping is repeated as long as the minimum weight child has smaller weight than the node under consideration. In this example, H_1 and H_4 are swapped, then H_4 and H_9 are swapped. Figure 4.13 shows the state of the heap after the three swaps.

Figure 4.13 The heap after swapping H_0 with H_1, H_1 with H_4, and H_4 with H_9.



This is the typical operation for removing the minimum element from the heap. However, in the polyline application, there is more work to be done. The weights of vertices V_{14} and V_0 depended on V_{15}. The right neighbor of V_{14} was V_{15}, but is now V_0. The left neighbor of V_0 was V_{15}, but is now V_{14}. The weights of V_{14} and V_0 must be recalculated because of the change of neighbors. The old weight for V_{14} is 187.79 and the new weight is 164.52. The old weight for V_0 is 52.65 and the new weight is 52.77. Neither change leads to an invalid heap, so no update of the heap array is necessary. Figure 4.14 shows the state of the heap after the two weight changes.

Figure 4.14 The heap after changing the weights on V_0 and V_{14}. The new weights are shown in gray.

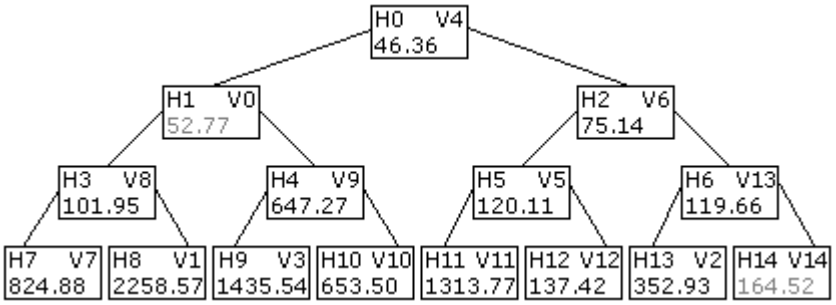
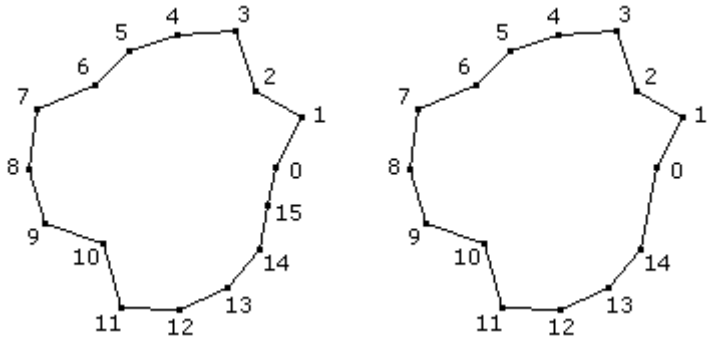


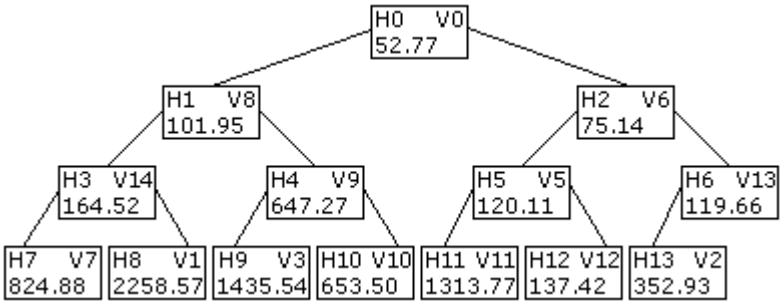
Figure 4.15 shows the polygon of Figure 4.1 and the polygon with V_{15} removed.

Figure 4.15 Left: The polygon of Figure 4.1. Right: The polygon with V_{15} removed.



The next vertex to be removed is V_4. The contents of the last heap node H_{14} are moved to the root, resulting in an invalid heap. Two swaps must occur, H_0 with H_1 and H_1 with H_3. Figure 4.16 shows the state of the heap after these changes.

Figure 4.16 The heap after moving H_{14} to H_0, then swapping H_0 with H_1 and H_1 with H_3.



The adjacent vertices whose weights must be updated are V_3 and V_5. For V_3, the old weight is 1435.54 and the new weight is 1492.74. This does not invalidate the heap at node H_9. For V_5, the old weight is 120.11 and the new weight is 157.11. This change invalidates the heap at node H_5. Nodes H_5 and H_{12} must be swapped to restore the heap. Figure 4.17 shows the state of the heap after the two weight changes and the swap.

Figure 4.17 The heap after changing the weights on V_3 and V_5 and swapping H_5 and H_{12}. The new weights are shown in gray.

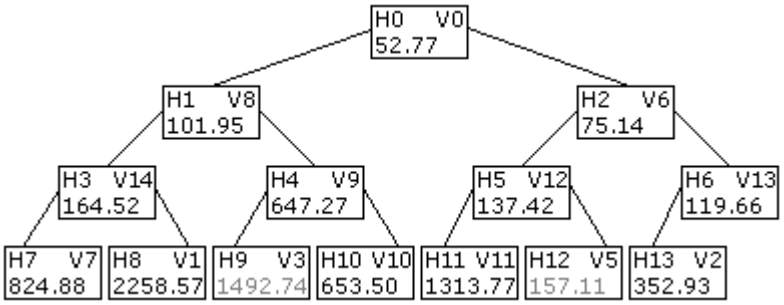
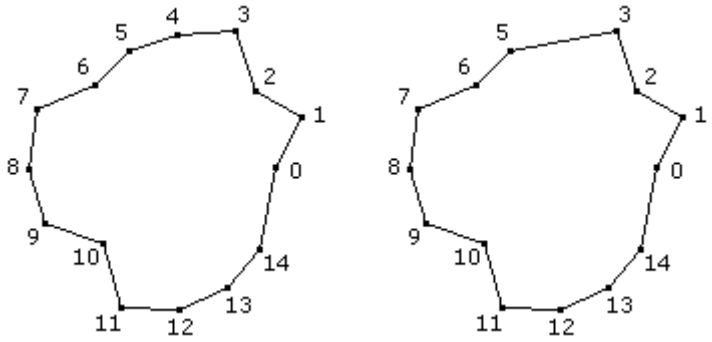


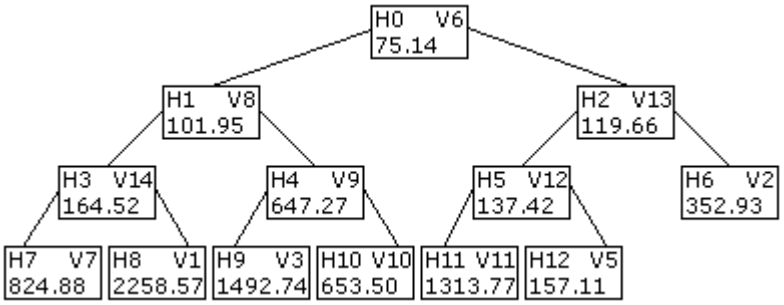
Figure 4.18 shows the right polygon of Figure 4.15 and the polygon with V_4 removed.

Figure 4.18 Left: The right polygon of Figure 4.15. Right: The polygon with V_4 removed.



The next vertex to be removed is V_0. The contents of the last heap node H_{13} are moved to the root, resulting in an invalid heap. Two swaps must occur, H_0 with H_2 and H_2 with H_6. Figure 4.19 shows the state of the heap after these changes.

Figure 4.19 The heap after moving H_{13} to H_0, then swapping H_0 with H_2 and H_2 with H_6.



The adjacent vertices whose weights must be updated are V_1 and V_{14}. The left neighbor is processed first in the implementation. For V_{14}, the old weight is 164.52 and the new weight is 65.80. The heap is invalid since the parent node H_1 has a weight that is larger than the weight at H_3. Two swaps must occur, H_3 with H_1 and H_1 with H_0. For V_1, the old weight is 2258.57 and the new weight is 791.10, but the heap is still valid. Figure 4.20 shows the state of the heap after the weight change and the swaps.

Figure 4.20 The heap after changing the weight on V_{14} and swapping H_3 with H_1 and H_1 with H_0, then changing the weight on V_1. The new weights are shown in gray.

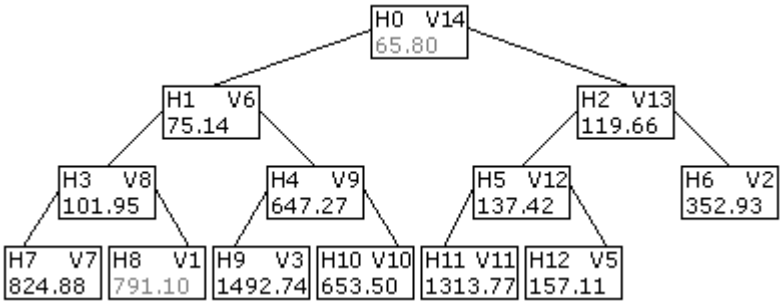
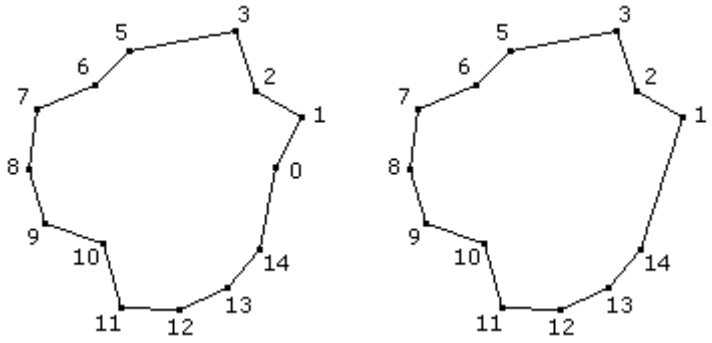


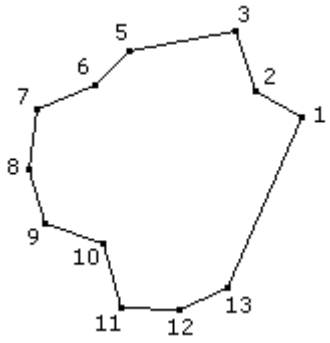
Figure 4.21 shows the right polygon of Figure 4.18 and the polygon with V_0 removed.

Figure 4.21 Left: The right polygon of Figure 4.18. Right: The polygon with V_0 removed.

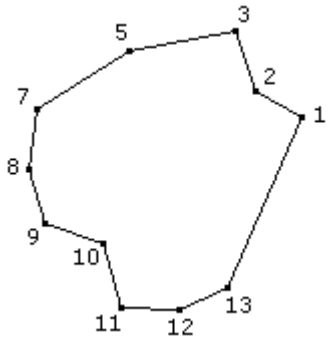


The process is similar for the remaining vertices, removed in the order V_{14}, V_6, V_5, V_8, V_{12}, V_2, V_{13}, V_{10}, V_9, and V_1. Vertices V_7, V_3, and V_{11} are the remaining ones. Figure 4.22 shows the corresponding reduced polygons. Collapses occur from left to right, top to bottom.

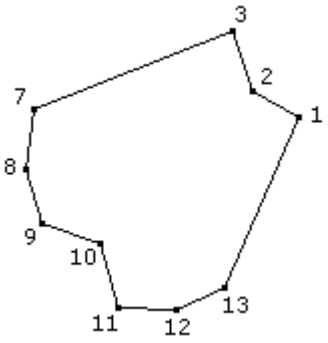
Figure 4.22 The remaining vertex collapses. The order is top to bottom in rows, and left to right in each row.



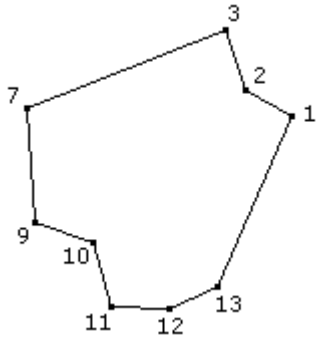
collapse V_{14}



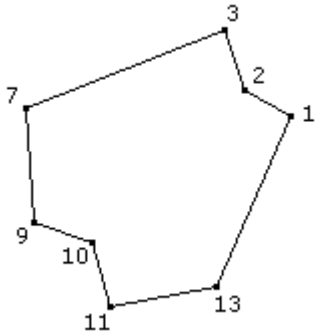
collapse V_6



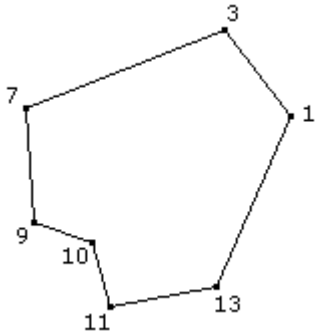
collapse V_5



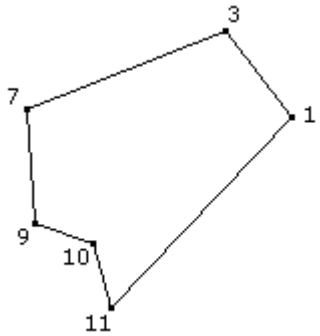
collapse V_8



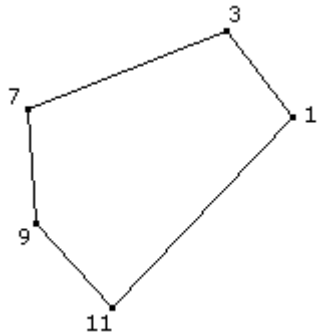
collapse V_{12}



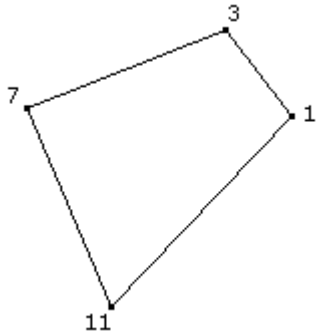
collapse V_2



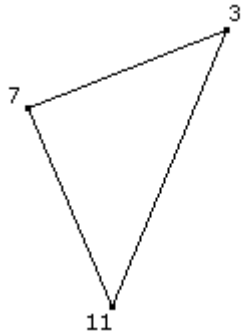
collapse V_{13}



collapse V_{10}



collapse V_9



collapse V_1

5 Dynamic Change in Level of Detail

The vertex collapses can be computed according to the algorithm presented earlier. An application might not want to only decrease the level of detail by vertex collapses, but increase it on demand. To support this, the edge connectivity must be stored with the polyline. The connectivity data structure will change based on the given addition or removal of a vertex.

An array of edge indices is used to represent the connectivity. The initial connectivity for an open polyline of n vertices is an array of $2n-2$ indices grouped in pairs as $\langle 0,1\rangle, \langle 1,2\rangle, \ldots, \langle n-2, n-1\rangle$. A closed polyline has one additional pair, $\langle n-1, 0\rangle$. If vertex V_i is removed, the pair of edges $\langle i-1, i\rangle$ and $\langle i, i+1\rangle$ must be replaced by a single edge $\langle i-1, i+1\rangle$. The change in level of detail amounts to inserting, removing, and modifying the elements of an array. An array is not well suited for such operations.

Instead, the initial array of edge indices should be sorted so that the last edge in the array is the first one removed by a collapse operation. If the indices of the collapsed vertices are sorted as $c_0, \ldots c_{n-1}$ where the last vertex in the array is the first one removed by a collapse operation, then the initial edge array should be

$$\langle c_0, c_0+1\rangle, \langle c_1, c_1+1\rangle, \ldots \langle c_{n-1}, c_{n-1}+1\rangle = \langle e_0, \ldots e_{2n-1}\rangle$$

where the index sum $i+1$ is computed modulo n to handle both open and closed polylines. To remove the vertex with index c_{n-1}, the last edge $\langle e_{2n-2}, e_{2n-1}\rangle$ is simply ignored. In an implementation, an index to the last edge in the array is maintained. When the level of detail decreases, that index is decremented. When the level increase, the index is incremented. The removal of the edge indicates that the vertex with index c_{n-1} is no longer in the polyline. That same index occurs earlier in the edge array and must be replaced by the second index of the edge. In the current example, $e_{2n-2} = c_{n-1}$ and $e_{2n-1} = c_{n-1}+1$. A search is made in the edge array for the index $e_{m_{n-1}}$ that is also equal to c_{n-1}, then $e_{m_{n-1}} \leftarrow e_{2n-1}$. The mapping m_{n-1} should be stored in order to increase the level of detail by restoring the original value of $e_{m_{n-1}}$ to c_{n-1}.

The algorithm is iterative. To remove the vertex with index c_k, observe that $e_{2k} = c_k$ and $e_{2k+1} = c_k + 1$. The edge quantity is decreased by one. A search is made in $\langle e_0, \ldots, e_{2k-1}$ for the index e_{m_k} that is equal to c_k, then replace $e_{m_k} \leftarrow e_{2k+1}$. Adding the vertex with index c_k back into the polyline is accomplished by replacing $e_{m_k} \leftarrow c_k$. The iteration stops when $k = 1$ for open polylines so that the final line segment is not collapsed to a single point. The iteration stops when $k = 5$ for closed polylines so that the smallest level of detail is a triangle that is never collapsed to a line segment.

Example. Consider the example shown previously that consisted of a 16-sided polygon. The vertex indices ordered from last removed to first removed are $3, 11, 7, 1, 9, 10, 13, 2, 12, 8, 5, 6, 14, 0, 4, 15$. The initial edge array is

$$\langle 3,4\rangle\langle 11,12\rangle\langle 7,8\rangle\langle 1,2\rangle\langle 9,10\rangle\langle 10,11\rangle\langle 13,14\rangle\langle 2,3\rangle\langle 12,13\rangle\langle 8,9\rangle\langle 5,6\rangle\langle 6,7\rangle\langle 14,15\rangle\langle 0,1\rangle\langle 4,5\rangle\langle 15,0\rangle$$

and the edge quantity is $Q_e = 16$. The vertex quantity is $Q_v = 16$. The removal of V_{15} is accomplished by decrementing $Q_v = 15$ and $Q_e = 15$. The last edge $\langle 15, 0\rangle$ is ignored (iterations over the edges use Q_e as the upper bound for the loop index). A search is made in the first 15 edges for index 15 and is found at $e[25]$ (in the edge $\langle 14, 15\rangle$). That index is replaced by $e[25] = 0$ where 0 is the second index of the removed edge $\langle 15, 0\rangle$. The mapping index is $m_{15} = 25$. The following table lists the vertex collapses, the mapping indices, and the edge array (only through the valid number of edges):

vertex	map	edges
15	25	$\langle 3,4\rangle\langle 11,12\rangle\langle 7,8\rangle\langle 1,2\rangle\langle 9,10\rangle\langle 10,11\rangle\langle 13,14\rangle\langle 2,3\rangle\langle 12,13\rangle\langle 8,9\rangle\langle 5,6\rangle\langle 6,7\rangle\langle 14,0\rangle\langle 0,1\rangle\langle 4,5\rangle$
4	1	$\langle 3,5\rangle\langle 11,12\rangle\langle 7,8\rangle\langle 1,2\rangle\langle 9,10\rangle\langle 10,11\rangle\langle 13,14\rangle\langle 2,3\rangle\langle 12,13\rangle\langle 8,9\rangle\langle 5,6\rangle\langle 6,7\rangle\langle 14,0\rangle\langle 0,1\rangle$
0	25	$\langle 3,5\rangle\langle 11,12\rangle\langle 7,8\rangle\langle 1,2\rangle\langle 9,10\rangle\langle 10,11\rangle\langle 13,14\rangle\langle 2,3\rangle\langle 12,13\rangle\langle 8,9\rangle\langle 5,6\rangle\langle 6,7\rangle\langle 14,1\rangle$
14	13	$\langle 3,5\rangle\langle 11,12\rangle\langle 7,8\rangle\langle 1,2\rangle\langle 9,10\rangle\langle 10,11\rangle\langle 13,1\rangle\langle 2,3\rangle\langle 12,13\rangle\langle 8,9\rangle\langle 5,6\rangle\langle 6,7\rangle$
6	21	$\langle 3,5\rangle\langle 11,12\rangle\langle 7,8\rangle\langle 1,2\rangle\langle 9,10\rangle\langle 10,11\rangle\langle 13,1\rangle\langle 2,3\rangle\langle 12,13\rangle\langle 8,9\rangle\langle 5,7\rangle$
5	1	$\langle 3,7\rangle\langle 11,12\rangle\langle 7,8\rangle\langle 1,2\rangle\langle 9,10\rangle\langle 10,11\rangle\langle 13,1\rangle\langle 2,3\rangle\langle 12,13\rangle\langle 8,9\rangle$
8	5	$\langle 3,7\rangle\langle 11,12\rangle\langle 7,9\rangle\langle 1,2\rangle\langle 9,10\rangle\langle 10,11\rangle\langle 13,1\rangle\langle 2,3\rangle\langle 12,13\rangle$
12	3	$\langle 3,7\rangle\langle 11,13\rangle\langle 7,9\rangle\langle 1,2\rangle\langle 9,10\rangle\langle 10,11\rangle\langle 13,1\rangle\langle 2,3\rangle$
2	7	$\langle 3,7\rangle\langle 11,13\rangle\langle 7,9\rangle\langle 1,3\rangle\langle 9,10\rangle\langle 10,11\rangle\langle 13,1\rangle$
13	3	$\langle 3,7\rangle\langle 11,1\rangle\langle 7,9\rangle\langle 1,3\rangle\langle 9,10\rangle\langle 10,11\rangle$
10	9	$\langle 3,7\rangle\langle 11,1\rangle\langle 7,9\rangle\langle 1,3\rangle\langle 9,11\rangle$
9	5	$\langle 3,7\rangle\langle 11,1\rangle\langle 7,11\rangle\langle 1,3\rangle$
1	3	$\langle 3,7\rangle\langle 11,3\rangle\langle 7,11\rangle$

Given the final triangle after all collapses, to restore vertex V_9 we need to increment Q_v to 4, increment Q_e to 4, and set $e[5] = 9$ where 5 is the mapping index associated with V_9.

6 Reordering Vertices

In an application that wants to rigidly transform the polyline, it might be useful to have the vertices at any level of detail stored as a packed array. This supports any optimized code for batch transforming a contiguous block of vertices. The collapse indices $(c_0, c_1, \ldots, c_{n-1})$ represent a permutation of $(0, 1, \ldots, n-1)$. The vertices themselves can be reordered using this permutation. Subsequently, the edge indices themselves must be converted properly. The reindexing requires the inverse permutation, $(d_0, d_1, \ldots, d_{n-1})$ where $d_{c_i} = i$. The mapping index does not change since the edge reindexing does not change the order of items in the edge array. If U_i are the reordered vertices, then $U_i = V_{c_i}$. If an edge is $E = \langle e_i, e_j\rangle$, then the reindexed edge is $F = \langle d_{e_i}, d_{e_j}\rangle$.

Example. The inverse permutation for

$$\mathbf{c} = (3, 11, 7, 1, 9, 10, 13, 2, 12, 8, 5, 6, 14, 0, 4, 15)$$

is

$$\mathbf{d} = (13, 3, 7, 0, 14, 10, 11, 2, 9, 4, 5, 1, 8, 6, 12, 15).$$

The initial edge array is

$$\langle 0,14\rangle\langle 1,8\rangle\langle 2,9\rangle\langle 3,7\rangle\langle 4,5\rangle\langle 5,1\rangle\langle 6,12\rangle\langle 7,0\rangle\langle 8,6\rangle\langle 9,4\rangle\langle 10,11\rangle\langle 11,2\rangle\langle 12,15\rangle\langle 13,3\rangle\langle 14,10\rangle\langle 15,13\rangle.$$

The vertex collapse table from the last example is reindexed to

vertex	map	edges
15	25	$\langle 0,14\rangle\langle 1,8\rangle\langle 2,9\rangle\langle 3,7\rangle\langle 4,5\rangle\langle 5,1\rangle\langle 6,12\rangle\langle 7,0\rangle\langle 8,6\rangle\langle 9,4\rangle\langle 10,11\rangle\langle 11,2\rangle\langle 12,13\rangle\langle 13,3\rangle\langle 14,10\rangle$
14	1	$\langle 0,10\rangle\langle 1,8\rangle\langle 2,9\rangle\langle 3,7\rangle\langle 4,5\rangle\langle 5,1\rangle\langle 6,12\rangle\langle 7,0\rangle\langle 8,6\rangle\langle 9,4\rangle\langle 10,11\rangle\langle 11,2\rangle\langle 12,13\rangle\langle 13,3\rangle$
13	25	$\langle 0,10\rangle\langle 1,8\rangle\langle 2,9\rangle\langle 3,7\rangle\langle 4,5\rangle\langle 5,1\rangle\langle 6,12\rangle\langle 7,0\rangle\langle 8,6\rangle\langle 9,4\rangle\langle 10,11\rangle\langle 11,2\rangle\langle 12,3\rangle$
12	13	$\langle 0,10\rangle\langle 1,8\rangle\langle 2,9\rangle\langle 3,7\rangle\langle 4,5\rangle\langle 5,1\rangle\langle 6,3\rangle\langle 7,0\rangle\langle 8,6\rangle\langle 9,4\rangle\langle 10,11\rangle\langle 11,2\rangle$
11	21	$\langle 0,10\rangle\langle 1,8\rangle\langle 2,9\rangle\langle 3,7\rangle\langle 4,5\rangle\langle 5,1\rangle\langle 6,3\rangle\langle 7,0\rangle\langle 8,6\rangle\langle 9,4\rangle\langle 10,2\rangle$
10	1	$\langle 0,2\rangle\langle 1,8\rangle\langle 2,9\rangle\langle 3,7\rangle\langle 4,5\rangle\langle 5,1\rangle\langle 6,3\rangle\langle 7,0\rangle\langle 8,6\rangle\langle 9,4\rangle$
9	5	$\langle 0,2\rangle\langle 1,8\rangle\langle 2,4\rangle\langle 3,7\rangle\langle 4,5\rangle\langle 5,1\rangle\langle 6,3\rangle\langle 7,0\rangle\langle 8,6\rangle$
8	3	$\langle 0,2\rangle\langle 1,6\rangle\langle 2,4\rangle\langle 3,7\rangle\langle 4,5\rangle\langle 5,1\rangle\langle 6,3\rangle\langle 7,0\rangle$
7	7	$\langle 0,2\rangle\langle 1,6\rangle\langle 2,4\rangle\langle 3,0\rangle\langle 4,5\rangle\langle 5,1\rangle\langle 6,3\rangle$
6	3	$\langle 0,2\rangle\langle 1,3\rangle\langle 2,4\rangle\langle 3,0\rangle\langle 4,5\rangle\langle 5,1\rangle$
5	9	$\langle 0,2\rangle\langle 1,3\rangle\langle 2,4\rangle\langle 3,0\rangle\langle 4,1\rangle$
4	5	$\langle 0,2\rangle\langle 1,3\rangle\langle 2,1\rangle\langle 3,0\rangle$
3	3	$\langle 0,2\rangle\langle 1,0\rangle\langle 2,1\rangle$