CUCS-327-88

Speeding up Dynamic Programming with Application to the Computation of RNA Structure

David Eppstein [1,2]
Zvi Galil [1,2,3]
Raffaele Giancarlo [1,2,4]

February 10, 1988

Abstract:

We describe the solution of a two dimensional recurrence used to compute the secondary structure of RNA. A naive dynamic programming solution to this recurrence takes time $O(n^4)$; this time had previously been improved to $O(n^3)$. Our new algorithm makes use of the convexity of the energy functions for RNA secondary structure to reduce the time to $O(n^2 \log^2 n)$. When the energy function is modeled by logarithms or other simple functions we solve the recurrence in time $O(n^2 \log n \log\log n)$. Our algorithms are simple and practical.

[1] Computer Science Department, Columbia University, New York, USA
[2] Work supported in part by NSF grants DCR-85-11713 and CCR-86-05353
[3] Computer Science Department, Tel-Aviv University, Tel-Aviv, Israel
[4] Supported by Italian Ministry of Education, Project "Teoria degli Algoritmi"

Introduction

A number of researchers have recently examined the following recurrence relation:

$$E[j] = \min_{1 \le i < j} D[i] + w(i,j) \tag{1}$$

We assume that the values of D are easily computed from the corresponding values of E; in particular we might take $D[i] = E[i]$. To begin the recurrence we are given the value of $E[1]$. The values of $E[j]$ for $1 \le j \le n$ can be computed from the above recurrence using a simple dynamic program, in time $O(n^2)$, and if we don't know anything about w we must use that amount of time. However in many important applications of equation 1, the following inequality (or its inverse) holds for all $i \le i' \le j \le j'$:

$$w(i,j) + w(i',j') \le w(i,j') + w(i',j) \tag{2}$$

We will refer to inequality 2 as the *quadrangle inequality*, and call any function w satisfying it *concave*; we call the inverse of inequality 2 the *inverse quadrangle inequality*, and call any function w satisfying it *convex*. If w is convex or concave, recurrence 1 can be solved in linear or close to linear time [4, 5, 10].

In this paper we examine the following two dimensional generalization of the above recurrence:

$$E[i,j] = \min_{\substack{1 \le i' < i \\ 1 \le j' < j}} D[i',j'] + w(i'+j', i+j) \tag{3}$$

In this recurrence, we again assume that $D[i,j]$ can be easily computed from the corresponding value of $E[i,j]$. We are given as initial values $E[1,j]$ and $E[i,1]$ for each i and j from 1 to n, and we require the values for each $E[i,j]$, again with i and j chosen between 1 and n. The recurrence can be solved by a simple dynamic program in time $O(n^4)$; the techniques described in the next section suffice to reduce this time to close to $O(n^3)$. In this paper we present a new algorithm, which when w is convex or concave solves recurrence 3 in time $O(n^2 \log^2 n)$. Our algorithm is simple and practical. For many common choices of w, a more complicated version of the algorithm solves the recurrence in time $O(n^2 \log n \log\log n)$.

The recurrence above has an important application to the computation of RNA secondary structure [7, 9]. After a simple change of variables, one can use it to solve the following recurrence:

$$C[p,q] = \min_{p < p' < q' < q} G[p',q'] + g((p'-p) + (q-q')) \tag{4}$$

In particular, let $f(x) = n + 1 - x$, and take $C[p,q] = E[f(p),q]$ and $D[i,j] = G[f(i),j]$. Then $w(x,y) = g(y-x)$ is convex when g is convex, and with these substitutions recurrence 3 also solves recurrence 4. There is one small complication, which is that to take into account the extra inequality $p' < q'$ in recurrence 4, we let the value of $D[i,j]$ be $+\infty$ for those values i,j such that $f(i) \ge j$.

Recurrence 4 is used to calculate the secondary structure of RNA, assuming the structure contains no multiple loops [7]. Our algorithm computes this structure in time $O(n^2 \log^2 n)$, under

the realistic assumption that the energy function w of a loop is a convex function of the number of exposed bases in that loop. It is possible to calculate RNA secondary structure with multiple loops, but this seems to require time $O(n^3)$ for linear energy functions, or $O(n^4)$ for general functions [9].

Waterman and Smith [9] also claim to have an algorithm for single loop RNA secondary structure, which they believe runs in time $O(n^2)$. However no proof is given, except in the (easy) case that w is linear (both convex and concave).

We describe here only the convex version of our algorithm. The concave version is very similar, but we know of no applications for that case.

Contention Within a Diagonal

In recurrence 3, we call each of the points (i', j') that may possibly contribute to the value of $E[i, j]$ *candidates*. We consider the computation of $E[i, j]$ as a contest between candidates; the *winner* is the point (i', j') with the minimum value of $D[i', j'] + w(i' + j', i + j)$. If we can find a way of eliminating many candidates at once, we can use this to reduce the time of an algorithm for solving recurrence 3.

We say that two points (i, j) and (i', j') in the matrices D or E are on the same *diagonal* when $i + j = i' + j'$. By the *length* of a diagonal we mean the number of points on it; e.g. the longest diagonal in the matrix has length n rather than $n\sqrt{2}$. We say that (k, l) is in the *range* of (i, j) when $k > i$ and $l > j$; that is, when point (i, j) is a candidate for the value of $E[k, l]$.

In this section we describe a way of eliminating candidates within the same diagonal. Using these methods, any given point (i, j) need only compare the values of candidates from different diagonals; there will be only one possible choice for the winning candidate from any given diagonal. Since there are $O(n)$ diagonals, each computation of $E[i, j]$ need involve only finding the minimum of $O(n)$ values; this gives us an algorithm for solving recurrence 3 having total time $O(n^3)$. We will also use the methods of this section in our $O(n^2 \log^2 n)$ time algorithm for convex and concave w.

In what follows we will assume that the region below a diagonal is a right triangle, having as its hypotenuse the diagonal below the given one, and having as its opposite right angled corner the point (n, n). In fact this region need not be triangular, but if we pretend that our matrices D and E are at the bottom right corner of $2n \times 2n$ matrices we can extend the region to a triangle of the given form (figure 1). This extension will not change the time bounds of our algorithms.

We denote rectangles by their upper left and lower right corners; that is, by the rectangle extending from (i, j) to (i', j') we mean the set of points (x, y) such that $i \leq x \leq i'$ and $j \leq y \leq j'$.

Lemma 1. If (i, j) and (i', j') are on the same diagonal, and if $D[i, j] \leq D[i', j']$, then for all (k, l) in the range of both points, $D[i, j] + w(i + j, k + l) \leq D[i', j'] + w(i' + j', k + l)$. In other words, (i', j') need not be considered as a candidate for those points in the range of (i, j).

Proof: Immediate from the assumption that $i + j = i' + j'$. •

Lemma 2. Each point (i, j) of a given diagonal need only be considered as a candidate for some rectangular subset of the range of (i, j), which we call the *domain* of (i, j). The domains for all points on the diagonal are disjoint and together cover the set of all points below the diagonal.

Proof: Consider the point (i, j) such that $D[i, j]$ is the minimum such value for all the points on the diagonal. Then by lemma 1, no other point than (i, j) of the diagonal need be considered

as a candidate for any of the points in the range of (i,j), so we may take the domain of (i,j) to be the range itself. This range is a rectangle, extending from $(i+1, j+1)$ to (n,n).

When we remove the domain for (i,j) from the triangle of points below the diagonal, we are left with two regions, each again a triangle (figure 2). By a similar argument, only the points in the corresponding portions of the diagonal with the minimum values of D need be considered as candidates for a rectangular region within each such triangle, and by induction we see that to each point on the diagonal there corresponds a domain having the properties described in the statement of the lemma. •

We can calculate the domains for each point (i,j) of a diagonal as follows. First we sort the points by the value of $D[i,j]$, lower values first. We will compute the domains for the points in this sorted order.

At any point in the computation, as in the proof of lemma 2, the remaining points below the diagonal will fall into a collection of triangles (the small triangles along the diagonal in figure 3). No two triangles overlap, so there is a natural ordering of the triangles according to the position of their hypotenuses along the diagonal. We will use a balanced search tree to represent the ordered list of triangles. Initially this search tree will contain only the triangle of all points below the diagonal.

To find the domain for point (i,j), we first use the search tree to find the triangle having (i,j) on its hypotenuse. Then, because of the sorted ordering of the points, the ranges of all points (i',j') such that $D[i',j'] < D[i,j]$ have already been removed from the triangles in the search tree, and all points (i',j') not yet processed have $D[i',j'] \geq D[i,j]$. Further, the intersection of the range of (i,j) with the remaining triangles in the search tree is exactly the rectangle extending from $(i+1, j+1)$ to the corner of the triangle containing (i,j). Therefore we can take the domain of (i,j) to be this rectangle. We remove the triangle from the search tree and add the two new triangles left by the removal of the domain.

Lemma 3. The domains for each of the points on a diagonal having m total points can be found in time $O(m \log m)$.

Proof: Each point adds at most one triangle to the search tree of triangles, so there are at most $m+1$ triangles ever in the search tree. Each domain is found in a constant number of search tree operations, taking time $O(\log m)$ each, and a constant number of other computations. Therefore the total time taken to compute the domains is $O(m \log m)$. •

Theorem 1. Recurrence 3 can be solved for general w in time $O(n^3)$.

Proof: We maintain a matrix of the best candidates seen so far for all points (i,j). We process a diagonal at a time; all candidates for points on the diagonal will have been processed already so to compute $E[i,j]$ we need only look up the winning candidate in our matrix. Then we compute the domains for the diagonal; for each point (i,j) on the diagonal, and for each point (i',j') in the domain of (i,j), we compare (i,j) with the previous best candidate recorded for (i',j'), and update the matrix of candidates if (i,j) is better.

There are $O(n)$ diagonals; for each diagonal we take time $O(n)$ to compute the values of $E[i,j]$, time $O(n \log n)$ to compute the domains, and time $O(n^2)$ to update the candidate matrix. Therefore the total time is $O(n^3)$. •

We should note that recurrence 3 can also be solved in time $O(n^3)$ by a simple three dimensional dynamic program, where the three dimensions are i, j, and the diagonal $i'+j'$ of a candidate [9]. In effect this alternate method computes the domains of points (i', j') a diagonal at a time, by noting that the domain containing (i, j) is either that containing $(i-1, j)$ or $(i, j-1)$. We will need to use our more complicated domain computation algorithm in the next section, because this simple method requires time $O(n^3)$ to compute the domains of all diagonals.

Convex and Concave Weight Functions

We describe now an improved algorithm for the important case that the weight function w is either convex or concave. In fact we describe only the convex algorithm, and note without proof how to change it to solve the concave case. In the previous section we described a method for quickly resolving the competition between candidates from a single diagonal; here we will add to this an algorithm for resolving competition between candidates from different diagonals.

We will need for our algorithm a data structure that maintains a partition of the sequence of numbers from 1 through n into intervals. We perform the following operations in the data structure:

(1) Find which interval contains a given number.

(2) Find which interval follows another given interval in the sequence.

(3) Join two adjacent intervals into one larger interval.

(4) Split an interval at a given point into two smaller intervals.

Such a data structure may be implemented at a cost of $O(\log n)$ per operation using balanced search trees [2, 6, 8]. A different algorithm, due to Peter van Emde Boas [3], implements these operations at a cost of $O(\log \log n)$ per operation. In general the simple search tree version of the data structure will be sufficient.

We keep a separate such partition for each row and column of the matrix of the original problem. Each interval in each partition will have a pointer to its *owner*, one of the points (i, j) for which we have already calculated $E[i, j]$. Any point (i, j) may own either some set of row intervals in the row partitions, or some set of column intervals in the column partitions; but no point may own both row and column intervals.

We will maintain as an invariant that, if the owner of the row interval containing point (i, j) is (i_r, j_r), then (i, j) is in the range of (i_r, j_r), and $D[i_r, j_r] + w(i_r + j_r, i + j)$ is the minimum such value among all points (i', j') owning rows. Similarly, the owner (i_c, j_c) of the column interval containing (i, j) is the best point among all points owning columns. When we compute $E[i, j]$ it will be the case for each point (i', j') such that (i, j) is in the range of (i', j'), that either (i', j') owns some intervals or else (i', j') can not be the winning candidate for (i, j). Therefore we may calculate $E[i, j]$ as the minimum between $D[i_r, j_r] + w(i_r + j_r, i + j)$ and $D[i_c, j_c] + w(i_c + j_c, i + j)$, which requires only two queries to the interval data structures, followed by a constant number of arithmetic operations.

It remains to show how to add a point (i, j) to the interval data structures, after $E[i, j]$ has been computed, so that the invariants above are maintained. We will add points a diagonal at a time. First we use the previously described algorithm to compute the domains for each point

on the diagonal. Each domain is a rectangle; we cut it into strips, which will be intervals either of rows or columns. We choose whether to cut the domain into row intervals or column intervals according to which direction results in the fewer strips (figure 4). Then we combine each strip with the corresponding row or column partition so that (i, j) ends up owning the subinterval of the strip containing exactly those points for which (i, j) is better than the points previously owning intervals in the partition. First let us compute a bound on the number of strips formed when we cut the domains.

Lemma 4. The total number of strips from a single diagonal is $O(n \log n)$.

Proof: Assume without loss of generality, as in the previous section, that the region to be cut into domains and then strips is a triangle, rather than having its corners cut off. The length of the diagonal of the triangle is at most $2n$.

Let $T(m)$ be the largest number of strips obtainable from a triangle having m elements on the diagonal. As in the proof of lemma 3, the point on the diagonal having the least value has a rectangular domain extending to the corner of the triangle, leaving two smaller triangular regions to be divided up among the remaining diagonal points. Let us say the sides of this outer rectangular domain are $i + 1$ and $j + 1$; then i and j are the diagonal lengths of the smaller triangles, and $i + j = m - 1$. The number of strips formed by this outer domain is the smaller of $i + 1$ and $j + 1$; without loss of generality we will assume this to be $i + 1$. Then

$$T(m) = \min_{\substack{i+j=m-1 \\ i \le j}} T(i) + T(j) + i + 1. \tag{5}$$

Now assume inductively that $T(k) \le ck \log k$ for $k < m$ and some constant $c \ge 1$. Let i and j be the indices giving the minimum in equation 5; note that $i < m/2$. Plugging these values into equation 5, we see that

$$\begin{aligned} T(m) &= c(i \log i + j \log j) + i + 1 \\ &\le c(i(\log m - 1) + j \log m) + i + 1 \\ &= c(m-1) \log m + (1 - c)i + 1 \\ &\le cm \log m. \end{aligned} \tag{6}$$

By induction, $T(m) = O(m \log m)$ for all m. But the number of strips for any diagonal is certainly no more than $T(2n) = O(n \log n)$. •

Now if we can add a single strip to our row and column partition data structures in logarithmic time, the resulting algorithm will take the time bounds claimed in the introduction. In fact our algorithm may take more than logarithmic time to add a strip, so we cannot bound the time so easily. The result of adding a strip will be the creation of at most two intervals in the partition, so the total number of intervals ever created is proportional to the number of strips. When we add a strip to the partition, we may also remove some intervals from the partition; we will charge the time for this removal to the previous creation of these intervals. Therefore when we add a strip to the data structure we will be charged $O(\log n)$ time for the creation of intervals, and another $O(\log n)$ for those intervals' later removal, for a total time of $O(\log n)$ per strip.

Before we describe how to perform the insertion of a strip, we need the following lemma, which is where we use the assumption that w is convex.

Lemma 5. If w is convex, and if all intervals in the partition of a row (or column) currently belong to points on an earlier diagonal or the same diagonal as that of point (i,j), then if (i,j) is better than the the previous owners for any points in that row (or column) contained within the domain of (i,j), it is better in a single interval starting at the lowest numbered point of the row (or column) in the domain for (i,j).

Proof: We prove the lemma for rows; the proof for columns is the same.

We know from lemma 2 that (i,j) is the best of all points on the diagonal within its own domain. An alternate way of stating the lemma is that, if (i',j') comes from an earlier diagonal and is better than (i,j), then it continues to be better for later points in the row. If this holds, we know that (i,j) is worse than the previous owners of all remaining points in the row, for if it is worse than (i',j') it is certainly worse than any points which have already been found to be better than (i',j').

Let (i'',j'') be the first point in the row for which (i,j) is worse than (i',j'); that is,

$$D[i,j] + w(i+j, i''+j'') \geq D[i',j'] + w(i'+j', i''+j''). \tag{7}$$

Then if $k > 1$, point $(i'', j''+k)$ follows (i'',j'') in this row, and $i''+j'' < i''+j''+k$. By assumption $i'+j' < i+j$. Then by convexity of w, the inverse quadrangle inequality

$$w(i+j, i''+j''+k) + w(i'+j', i''+j'') \geq w(i+j, i''+j'') + w(i'+j', i''+j''+k) \tag{8}$$

holds. Equation 8 can be rewritten as

$$w(i+j, i''+j''+k) - w(i+j, i''+j'') \geq w(i'+j', i''+j''+k) - w(i'+j', i''+j''), \tag{9}$$

and adding equations 7 and 9 gives

$$D[i,j] + w(i+j, i''+j''+k) \geq D[i',j'] + w(i'+j', i''+j''+k), \tag{10}$$

which states exactly that (i,j) continues to be worse than (i',j'). The proof for columns is identical to that for rows. •

We note without proof that, when w is concave, the points of the row for which (i,j) is best again form a single interval, this time ending at the last point in the domain.

We are now ready to describe how to insert the strip for (i,j) into the row (or column) interval partition, once we have calculated the domain as described in the previous section. We first look at the first point (i',j') of the row contained in the domain, and find the interval containing that point. If the previous owner is better at that point, then (i,j) is never better in this row, and we are done inserting its strip. Otherwise, we split the interval containing (i',j') into two intervals, so that the second of the two starts at (i',j'). But for now we leave both intervals having the same original owner. It may be that when we have finished, more than one interval in the row has the same owner; this is not a problem.

We have found an interval at the start of which (i,j) is better than the other points owning intervals in this row. This interval, which we call the *candidate interval*, is currently owned by

some other point; this owner may be better than (i,j) at the other end of the candidate interval. We also need to remember the interval we have already assigned to owner (i,j); for now this is the empty interval.

We repeat the following steps: First we find the next interval following the candidate interval. If this interval starts within the domain of (i,j), and if (i,j) is better than the owner of this new interval at the first point of the interval, then (i,j) must be better than the owner of the candidate interval for all of the candidate interval by lemma 5, and because it is better at the start of the new interval. In this case we merge the candidate interval with the interval owned by (i,j), and set the owner of the merged interval to be again (i,j). We remember the following interval as the new candidate interval, and continue the loop. Otherwise the interval in which (i,j) is best is contained within the candidate interval, so we halt the loop.

We now know that the interval in which (i,j) is best stops somewhere in the candidate interval. The interval needs to be divided into two parts; the first part will be owned by (i,j), and the second part left to its original owner. We find the point at which to split the interval by binary search, at each step comparing the value of $D+w$ for (i,j) with that of the previous owner of the candidate interval. The points searched start at the start of the candidate interval and end either at the end of the interval or the end of the strip, whichever comes first.

At each step of the loop other than the last, an interval is removed from the partition, and we can charge the costs of that step to the interval being removed as mentioned earlier. The costs not charged are a constant number of interval operations, including the creation of a constant number of new intervals, for a cost of $O(\log n)$, and the binary search to find the split point, for another $O(\log n)$. For many functions w we can compute the split point directly, without performing a binary search; if w is such a function we can use the more complicated data structure of van Emde Boas [3] to achieve a time of $O(\log\log n)$ per strip insertion.

The algorithm as a whole proceeds as follows. For each diagonal $i+j=k$, k from 2 to $2n$, we perform the following steps:

(1) Look up the owners of each point (i,j) of the diagonal in the row and column partition data structures, and compute $E[i,j]$ and $D[i,j]$.

(2) Compute the domains of the diagonal points.

(3) For each point (i,j), cut the domain into strips, either by rows or by columns depending on which gives fewer strips, and combine the strips with the appropriate partitions.

Theorem 2. The above algorithm computes the values of $E[i,j]$ in total time $O(n^2\log^2 n)$ for general w, or $O(n\log n\log\log n)$ for simple w.

Proof: The time taken for each diagonal is as follows. Step 1 takes at most $O(\log n)$ time for each point, for a time of $O(n\log n)$. Step 2 takes time $(n\log n)$ as described in the previous section. Step 3 takes time $O(\log n)$ time per strip for convex or concave w, and $O(\log\log n)$ per strip for simple w. There are $O(n\log n)$ strips, so step 3 takes time $O(n\log^2 n)$ or $O(n\log n\log\log n)$. There are $O(n)$ diagonals, so the total time for the algorithm is $O(n^2\log^2 n)$ for convex or concave w, and $O(n^2\log n\log\log n)$ for simple w. •

References

[1] Alok Aggarwal, Maria Margaret Klawe, Shlomo Moran, Peter Shor, and Robert Wilber, Geometric Applications of a Matrix-Searching Algorithm, Algorithmica 2 (1987), pp. 209–233.

[2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, 1974.

[3] Peter van Emde Boas, Preserving Order in a Forest in Less Than Logarithmic Time, Proc. 16th Symp. Found. Comput. Sci., 1975.

[4] Zvi Galil and Raffaele Giancarlo, Speeding Up Dynamic Programming with Applications to Molecular Biology, Theor. Comput. Sci., to appear.

[5] Maria Margaret Klawe, Speeding Up Dynamic Programming, manuscript.

[6] Donald Ervin Knuth, The Art of Computer Programming, Volume 3: Sorting and Searching, Addison-Wesley, 1973.

[7] David Sankoff, Joseph B. Kruskal, Sylvie Mainville, and Robert J. Cedergren, Fast Algorithms to Determine RNA Secondary Structures Containing Multiple Loops, in D. Sankoff and J.B. Kruskal, editors, Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison, Addison-Wesley, 1983, pp. 93–120.

[8] Robert Endre Tarjan, Data Structures and Network Algorithms, SIAM, 1985.

[9] Michael S. Waterman and Temple F. Smith, Rapid Dynamic Programming Algorithms for RNA Secondary Structure, in Advances in Applied Mathematics 7 (1986), pp. 455–464.

[10] Robert Wilber, The Convex Least Weight Subsequence Problem Revisited, J. Algorithms, to appear.

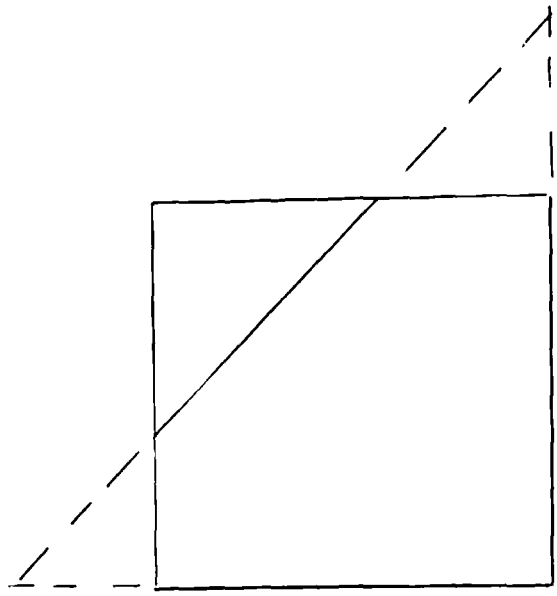

Figure 1. Making the region below a diagonal triangular.

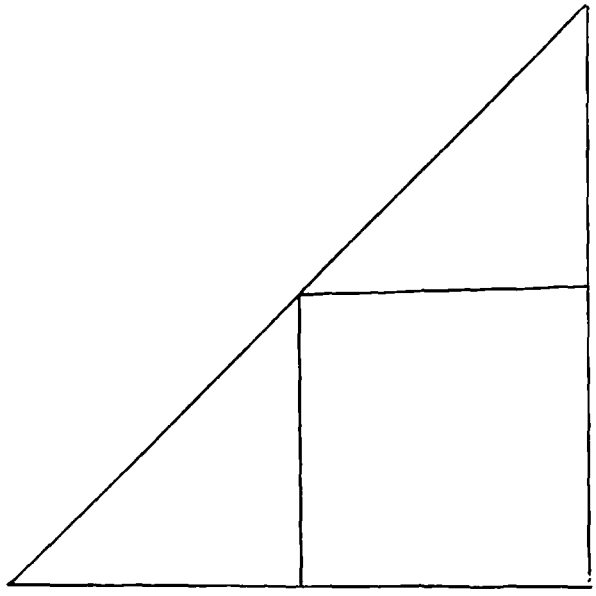

Figure 2. Removing a rectangle leaves two triangles.

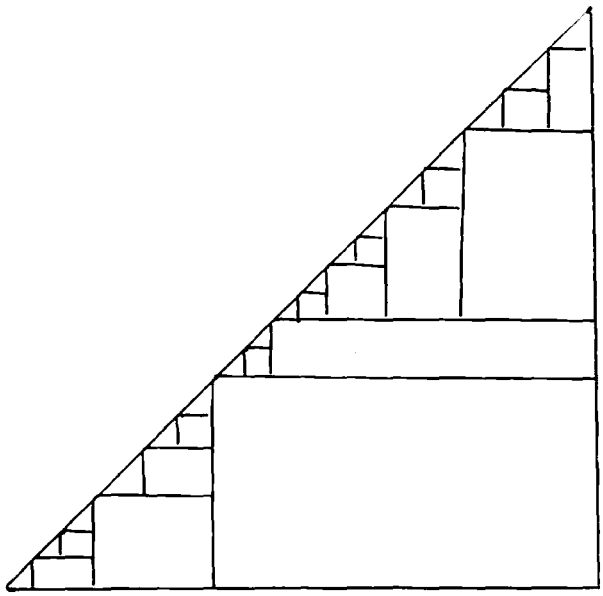

Figure 3. Domains of the points on a single diagonal.

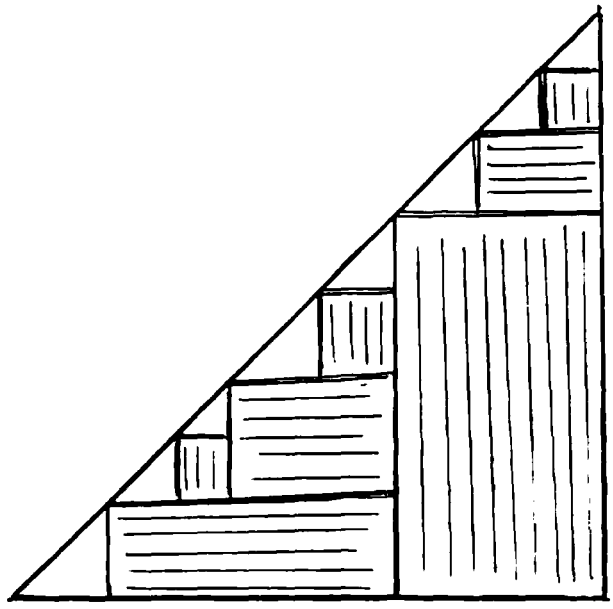

Figure 4. Cutting domains into strips.