An Improved Output-size Sensitive Parallel Algorithm for Hidden-Surface Removal for Terrains

Neelima Gupta and Sandeep Sen
Department of Computer Science and Engineering
Indian Institute of Technology
New Delhi 110016, India,
{neelima,ssen}@cse.iitd.ernet.in

Abstract

We describe an efficient parallel algorithm for hidden-surface removal for terrain maps. The algorithm runs in $O(\log^4 n)$ steps on the CREW PRAM model with a work bound of $O((n+k)polylog(n))$ where n and k are the input and output sizes respectively. In order to achieve the work bound we use a number of techniques, among which our use of persistent data-structures is somewhat novel in the context of parallel algorithms. To the best of our knowledge this is the most efficient parallel algorithm for hidden-surface removal for an important class of 3-D scenes.

1. Introduction

1.1. The Problem

The hidden-surface elimination problem (see [24] for early history) has been a fundamental problem in computer graphics and can be stated as - given n polyhedral faces in $\mathbb{R}^3$ and a projection plane, we wish to determine which portions of the faces are visible when viewed in a given direction. We are interested in an object-space solution (independent of the display device) for this problem. That is, we are interested in a combinatorial description of the visible scene which can then be rendered on any display device. The image-space solutions compute the visibility information at every pixel which makes them device dependent. It has been shown that the worst case output size for hidden surface elimination can be $\Omega(n^2)$ for n segments, and hence, the worst case optimal algorithms for these problems will have a running time of $\Omega(n^2)$.

There are algorithms whose running times are sensitive to the number of intersections, I, (of the projections of the segments) in the image plane. However, in practice, the size of the displayed image can be far less than the number of intersections in the image plane. By size, we mean the number of vertices and edges of the displayed image as a (planar) graph. This would happen when a large number of these intersections are occluded by the visible surfaces.

We will study a special class of surfaces called polyhedral terrains which occur frequently in practice. A terrain is a three-dimensional polyhedral surface which can be represented as a function of two variables.

Most geographical features can be represented in this manner. A large number of scenes in graphics applications can be modelled efficiently and effectively by polyhedral terrains. The term *(upper) profile* will refer to the piece-wise linear function $Z(y)$, which is the point-wise maximum in $+z$ direction of the projection of edges onto the $z-y$ plane. Other commonly used terms for upper-profile are *upper-envelope* and *silhouette*. Therefore, a profile is a monotone polygon with respect to the $y-$axis. In fact, monotonicity turns out to be a very useful property in making the algorithm somewhat simpler than hidden-surface removal algorithm for general surfaces. However, even for terrains, it is known that the maximum size of the visible image can be $\Omega(n^2)$. Our aim is to design a fast output-sensitive (we will often use output-sensitive instead of output-size sensitive) parallel algorithm for terrains, which computes a description of the output in a device-independent manner.

1.2. Sequential algorithms

Several sequential algorithms exist for the problem. (See [13, 5, 14, 22, 7, 8, 2, 11, 19, 18, 15, 4, 16]).

For the class of polyhedral terrains, Reif and Sen [19] designed the first efficient algorithm whose running time is $O((k+n)\log^2 n)$ where k is the output size.

1.3. Parallel algorithms

Relatively little work has been done in the context of parallel algorithms for hidden-surface removal. Reif and Sen [19] had proposed a parallelization of their algorithm with polylog(n) running time. The more challenging theoretical goal was to keep the work bound close to the output-sensitive sequential algorithm. The resulting algorithm was quite complex and required parallel (dynamic) updates on a

shared nested data structure that were not only hard to implement but also difficult to analyze. Here, we exploit some of their ideas but adopt a different strategy to build the parallel data-structure. The resulting algorithm is relatively simpler and also easier to analyze. The main reason for this is that the underlying data-structure is static although it has to be rebuilt a (small) number of times. Our bounds are also superior in the sense that we are able to match the bounds of [19] in a standard PRAM model (processor allocation was assumed to be free in the model used by [19]).

Goodrich, Ghouse and Bright [12] presented parallel algorithms for hidden-surface elimination. For the general scenes, their algorithm computes all the I pairwise intersections on the projection plane. For the case of iso-oriented rectangles in $\mathbb{R}^3$, their algorithm is output-sensitive and runs in $O(\log^2 n)$ time using $O((n+k)\log n)$ total parallel operations. The crux of their method is a parallel data-structure called *array-of-trees* introduced by Atallah, Goodrich and Kosaraju [10], that has some flavour of persistent data-structures. In this paper, we make more direct use of persistent data-structures in our parallel algorithm.

The importance of output-size sensitivity for parallel algorithms cannot be over-emphasized for the following simple reason. The advantage of using extra processors will be lost otherwise (for small output-size) as compared to an efficient output-sensitive sequential algorithm. The rest of the paper is organized as follows. In section 2, we give a brief overview of our approach. In section 3, we describe some of the basic parallel routines used frequently in the main algorithm. Section 4 forms the crux of the paper. Since the algorithm is somewhat involved, we give a top-down description of the algorithm and the data-structures accompanied by analysis.

2. An overview of our algorithm

Recall from the introduction that terrains in this paper refer to piecewise linear surfaces which meet a vertical line in exactly one point. Assume that the surface is a function $z = f(x, y)$, it is being viewed from $x = \infty$ and the viewing plane is the $z - y$ plane. We are viewing the scene in a direction perpendicular to the projection plane, however the algorithm works for perspective projection as well. A characteristic of these surfaces is that the upper boundary of the projection of the line segments on the $z - y$ plane is monotone with respect to the $y-$axis. We assume that the terrain is available as a graph G whose vertices are 3-tuples (x, y, z) of coordinates such that $z = f(x, y)$ and whose edges correspond to the segments of the polyhedral surface. The terms edges and segments have therefore been used interchangeably. We also assume that only the top part of the surface is visible, i.e., the faces closest to the observer rise from the ground level. A key property that allows one to solve the visibility problem efficiently is that the edges can be ordered from 'front' to 'back' using the following observation. Project G on the $x - y$ plane (call it G_{xy}) and now the ordering of the segments in the scene in the increasing distance from the viewer corresponds to the ordering of the edges of G_{xy} along x. That is, we can define a partial order on the edges as follows : edge $e_i \prec e_j$ if there is a ray in the viewing direction that intersects e_i before e_j. The projection of the edges on the $x - y$ plane preserves this ordering.

In the sequential algorithm, the edges are processed one by one sequentially in order. The algorithm maintains an upper profile of the edges processed so far and tests the visibility of the next edge being processed by determining its intersection with the current profile. Since the edges are processed in the order of increasing distance from the viewer, the profile lies in front of the next edge and therefore occludes the portion of the edge which lies behind it. Thus the portion of the projected edge lying below the profile (which is a simple monotone polygon) is not visible and hence is discarded. The upper profile is updated with the visible portions of the edge . Clearly, the portion of an edge declared visible is visible in the final image (i.e. it cannot be occluded by edges processed later). Some vertices and edges of the profile may be deleted at this point which only means that they are no longer part of the 'upper boundary' of the final image but they are very much visible in the final image and therefore are remembered. Finally, we have all the vertices and edges of the final image which can be used by the rendering procedure to draw it on the screen.

2.1. An overview of the parallel algorithm

In the parallel scenario the above sequential algorithm has two major stumbling blocks. First, the edges are processed sequentially and the upper profiles are computed incrementally. We overcome this problem with the help of a Separator Tree and computing profiles using an approach similar to systolic implementation of parallel prefix computation. Separator tree provides a way to order the edges in the increasing distance from the viewer in parallel and also allows one to process them concurrently. Second, the intersections of an edge with a profile are computed sequentially. We use the divide-and-conquer approach to detect the intersections efficiently in parallel. We order the edges using a separator tree. Let $e_1, e_2 \ldots e_n$ be the ordered set of input edges. Let P_i denote the i^{th} profile, i.e. the upper profile of the edges $e_1, e_2 \ldots e_i$. Our aim is to compute $P_i \, \forall \, i = 1 \ldots n$. We call them *actual* profiles (however we will omit 'actual' most of the times and mention it only if it is not clear from the context).

We compute these profiles in two phases. In phase 1 we compute in parallel, for all the nodes of the separator tree the upper profile of the edges in the leaves of the subtree rooted at the node (the edges in the leaves of the separator tree are sorted in the increasing distance from the viewer). Call the resulting tree as the 'profile computation tree' (PCT). Notice that these profiles are not the actual profiles we are look-

ing for. These are only intermediate profiles which are used to compute the actual profiles in phase 2. In phase 2 we compute the actual profiles using an approach similar to the systolic implementation of parallel prefix computation [9]. Starting from the root of the profile computation tree the computation proceeds towards the leaves level by level. In this phase, at any node, the computation involves 'merging' of two profiles - an (actual) profile inherited from its parent and an (intermediate) profile computed in the previous phase by one of its children.

Merging is done by finding the intersections of the segments of the intermediate profile with the other profile and updating the other profile. However, as we will see later, our visibility structure (i.e. the vertices of the profiles) may be shared among different nodes at the same level of PCT. We can not afford to keep these profiles totally independent of each other because that will jeopardize our main objective of designing an output-sensitive algorithm. Instead of keeping a visibility structure for each profile at a fixed level of PCT we keep just one structure maintaining information about all the intersections computed so far and also provide a search structure to detect the intersections at the next level of PCT. We shall need frequent applications of the slow-down lemma which (in our context) can be formally stated as follows. Let $t_{p,r}$ denote the time to allocate p processors to a number of tasks whose total processor requirement is $O(r)$. That is $t_{p,r}$ is the time to solve the problem of processor allocation of size r with p processors.

Lemma 2.1 *Let A be a parallel algorithm that executes in Π phases and performs a total number of N tasks (each task is not necessarily unit time but is performed by a single processor). Then the algorithm can be executed in time $O(\Pi(t_{p,N}+t)+Nt/p)$ using p processors in a PRAM where t is the time taken for each task.*

Proof: This is a straightforward generalization of Brent's slow-down lemma. □

Lemma 2.2 *Let A be a parallel algorithm that executes in Π phases. Let N_i be the number of tasks in phase i, each executes in $O(t_i)$ time with p_i processors. Let $t = \sum_{i=1}^{\Pi} t_i$ and $N = \max_i\{N_i p_i\}$. Then the algorithm can be executed in $O(\Pi t_{p,N} + t + Nt/p)$ time with p processors.*

3. The parallel algorithm

We describe the algorithm in a top-down manner, treating the important steps in individual subsections accompanied by detailed analysis. Given a 2-D surface as a straight line graph in three dimensions, we project the line-segments on the $x-y$ plane . By the property of terrain maps, no two projected segments will intersect. If the graph is not triangulated, we triangulate the graph using the algorithm of Atallah, Cole and Goodrich [1] for parallel triangulation. Since it is a planar graph, the number of edges and faces is still $O(n)$. Henceforth our discussions will be with respect to the triangulated graph. The main steps of the algorithm are:

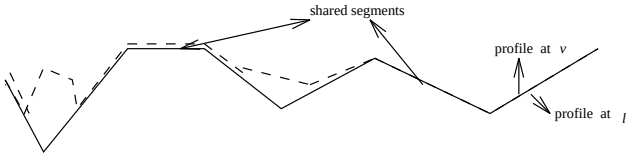

Figure 1. l is the left child of the node v in PCT, profile at l is the profile of the subset of edges of the set whose profile is computed at v.

1. **Order the edges** of the triangulated graph using seperator trees. .
2. **Profile computation**

(a) **Phase 1 : Compute the intermediate profiles** - to compute the profiles we take the projection of the line segments on the $z-y$ plane. For each node v in the separator tree do in parallel: compute the profile of the edges in the leaves of the subtree rooted at v. We shall call these profiles *intermediate* profiles. Note that these are not necessarily part of the final visible scene. As mentioned earlier, we call the resulting tree the Profile Computation Tree . We shall use the term *layer* to imply a level of PCT. Observe that the segments of the profiles may be shared among the layers of PCT (see Figure 1) .

(b) **Phase 2 : Compute the actual profiles** - compute the actual (visible) profiles starting from the root of the PCT, proceeding layer by layer towards the leaves. This step constitutes the crux of our algorithm.

Step 1 can be implemented in $O(\log n)$ time using a linear number of processors in an EREW PRAM using a procedure due to Tamassia and Vitter [25]. Their result can be summarized as follows:

Fact 1 *Let S be a planar triangulated subdivision with n vertices. Then the separator tree consisting of monotone chains that decompose S can be constructed by an EREW PRAM in $O(\log n)$ time using n processors.*

Lemma 3.1 *The profile of a set of m segments can be constructed in $O(\log^2 m)$ time using $O(m\alpha(m)/\log m)$ processors in a CREW PRAM .*

Proof: This is done by dividing the set of segments in two equal halves, computing the profiles recursively for each half and merging the profiles. Details are ommitted here. □

Thus Step 2a can be done in $O(\log^2 n)$ time using $O(n\alpha(n))$ processors in a CREW PRAM.

3.1. Computing the actual profiles

Given the profiles and the data structure for intersection detection at a given layer, say L of PCT, several actual profiles are computed at the next layer in parallel. Suppose at a

node u of PCT, we compute the actual profile P_j by merging P_i ($i < j$, P_i inherited from the parent of u) with an intermediate profile π_{ij} precomputed (by the left child of u) in phase 1. For each segment s of π_{ij} we compute the intersection of s with P_i. Some of the vertices of P_i may be deleted as they lie below s and hence do not contribute to P_j. Some new intersections may be detected and added to P_j. The intersection of a segment with a profile is computed as follows.

Lemma 3.2 *Given the data structure for intersection detection of a profile P of size m and a line segment s, we can find all the k_s intersections of s with P in time $O(max\{(T_I + t_{p,k_s}) \log m, k_s T_I/p\})$ with p processors on CREW PRAM, where T_I is the sequential time to detect the first intersection of a segment with P.*

Proof: Split the segment s around the middle diagonal d (among the diagonals that the segment spans). Find the intersection closest to d in both the subsegments and recurse. The result follows by applying Lemma 2.1. Further details are ommitted here. □

To detect the first intersection between a segment and a profile (if an intersection exists), we use the data structure and recursive search procedure of Chazelle and Guibas ([3]). The sequential algorithm of Reif and Sen revolved around making this data structure dynamic. See Figure 2. We will

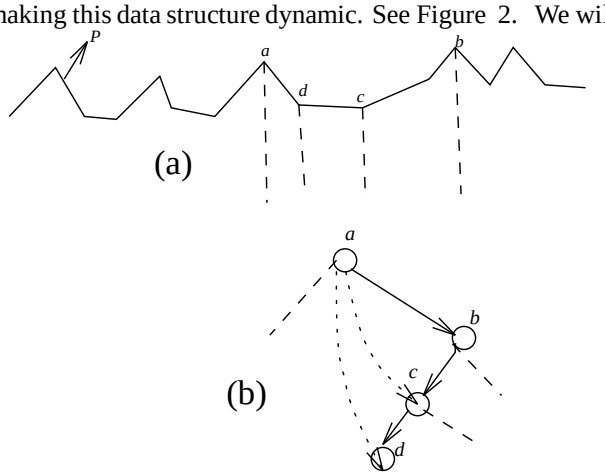

Figure 2. (a) profile P. (b) CG data structure for P.

refer to this structure of Chazelle and Guibas by CG data structure in future. By an *edge* of CG we would imply either a tree edge or a shooting pointer (shown as dotted arcs in the figure) unless explicitly stated otherwise.

Lemma 3.3 *For a profile with m vertices, we can construct the CG data structure in $O(\max\{\log m, t_{p,m} + m \log m/p\})$ time using p processors on the CREW PRAM.*

Proof: Follows from a straight-forward divide-and-conquer strategy and then applying Lemma 2.1.

The original data structure of CG was somewhat more complex based on dual transforms. Here instead, to detect whether a segment intersects a profile between two diagonals a and b we augment each edge ab of the CG data structure with the lower convex chain of the vertices of the profile between a and b. Call the resulting structure as *augmented* CG and refer to it as ACG hereafter. The above procedure is along the lines of Preparata and Vitter [18]. A crucial factor here is sharing of common visible segments between the profiles being computed at different nodes of the same layer of PCT. To handle this problem, instead of keeping an ACG structure for every profile, we keep a single ACG structure for all the profiles. That is, we construct the CG on all the intersections found upto a certain layer, say, L of PCT. and augment each edge of it with the lower convex chains of all those profiles (all profiles computed so far) which participate in detecting the intersections at the next layer so that the proper chain is searched for intersection. (see Figure 3). Here, we use a shared data-structure along the lines of a *persistent binary tree* structure [6] to store the convex chains of all the profiles taking care of their shared visible portions.

Lemma 3.4 *The CG structure constructed on the k intersections computed upto a fixed layer of PCT can be augmented with all the convex chains in $O(\max\{\log^2 k, t_{p,k} \log k + k \log^2 k/p\})$ time using p processors on a CREW PRAM.*

Proof: The constructions and the proof are very long and therefore ommitted. □

Lemma 3.5 *At a fixed layer of PCT, the ACG structure can be constructed in $O(\max\{\log^2 k, t_{p,k} \log k + k \log^2 k/p\})$ time using p processors on a CREW PRAM, where k is the number of intersections computed upto that layer.*

Proof: Follows from Lemmas 3.3 and 3.4. □

Lemma 3.6 *All k_s intersections of a segment s with a profile can be detected in $O(\max\{\log^2 k + t_{p,k_s}) \log n, k_s \log^2 k/p\})$ time with p processors, where k is total number of intersections computed upto a fixed layer of PCT.*

Proof: To search whether a ray intersects a profile P and detect the first intersection in case it does, proceed level by level of ACG starting from the root, according to the recursive search procedure mentioned earlier. At each level, it involves searching whether a ray intersects the convex chain corresponding to P between two diagonals (not necessarily of P). This requires $O(\log k)$ phases each requiring $O(\log k)$ time. Thus the result follows from Lemma 3.2. □

Hence by Lemma 2.1 all the intersections of all the segments of all the intermediate profiles at the next layer of PCT can be computed in $O(\max\{\log^3 k, t_{p,k+n\alpha(n)} \log k + (k + n\alpha(n)) \log^2 k/p\})$ time, where $n\alpha(n)$ is the total number of segments whose intersections are to be computed and k is the maximum number of intersections. Finally the processor allocation problem of size r can be done in $O(r \log r/p)$

time using p processors on CREW PRAM. Hence all the intersections at the next layer of PCT can be computed in $O(max\{\log^3 n, (k+n\alpha(n))\log^2 n/p\})$ time using p processors, or all intersections can be detected in $O(max\{\log^4 n, (k+n\alpha(n))\log^3 n/p\})$ time over all layers of PCT using p processors. we summarize our final result as follows.

Theorem 3.1 *The hidden-surface elimination problem for terrains can be solved in* $O(\max\{\log^4 n, (k+n\alpha(n))\log^3 n/p\})$ *time using* p *CREW processors where* n *and* k *are the input and the output sizes respectively.*

Remark For $p = n\alpha(n)/\log n$, the work bound is $O((k+n\alpha(n))\log^3 n)$ which is within $O(\log n)$ factor of the sequential bound of Reif and Sen [20].

References

[1] M. J. Atallah, R. Cole, and M. T. Goodrich. Cascading divide-and-conquer : A technique for designing parallel algorithms. *Proceedings of 28th IEEE Symposium on Foundations Of Computer Science*, pages 151 – 160, 1987.

[2] M. Bern. Hidden surface removal for rectangles. *Proc. of the 4th ACM Symp. on Computational Geometry*, pages 183–192, 1988.

[3] B. Chazelle and L. Guibas. Visibility and intersection problems in plane geometry. *Proc. ACM Symposium on Computational Geometry*, pages 135 – 146, 1985.

[4] M. deBerg, D. Halpern, M. Overmars, J. Snoeyink, and M. Krevald. Efficient ray-shooting and hidden-surface removal. *Proc. 7th ACM Symposium on Computational Geometry*, pages 21 – 30, 1991.

[5] F. Devai. Quadratic bounds for hidden-line elimination. *Proc. 2nd Annual Symp. on Computational Geometry*, pages 269 – 275, 1986.

[6] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarzan. Make the data-structures persistent. *Journal of Computer and System Sciences*, 38:86 – 124, 1989.

[7] M. T. Goodrich. A polygonal approach to hidden-line elimination. *GVGIP : Graphical Models and Image Processing*, 54(1):1 – 12, 1992.

[8] R. Guting and T. Ottmann. New algorithms for special cases of the hidden-line elimination problem. *Proc. of the STACS*, pages 161–171, 1985.

[9] R. Ladner and M. Fischer. Parallel prefix computation. *Journal of ACM*, 27(4):831 – 838, 1980.

[10] M. Atallah, M. Goodrich, R. Kosaraju. Parallel algorithms for evaluating sequences of set-manipulation operations. *LNCS 319: Aegean Workshop on Computing*, pages 1–10, 1988.

[11] M. Atallah, M. Goodrich and M. Overmaars. An input-size/output-size trade-off inthe time-complexity of rectilinear hidden-surface removal. *Proc. of ICALP*, 1990.

[12] M. Ghouse, M. Goodrich and J. Bright. Generalized sweep methods for parallel computatioonal geometry. *Proc. of the 2nd ACM Symp. on Parallel Algorithms and Architectures*, pages 280–289, 1990.

[13] M. McKenna. Worst-case optimal hidden-surface removal. *ACM Transactions on Graphics*, pages 19 – 28, 1987.

[14] O. Nurmi. A fast line-sweep algorithm for hidden-line elimination. *BIT*, 25:466 – 472, 1985.

[15] M. Overmars, M. Kartz, and M. Sharir. Efficient hidden surface removal for objects with small union size. *Computational Geometry : Theory and Application*, 2:223 – 234, 1992.

[16] M. Overmars and M. Sharir. Output-sensitive hidden-surface removal. *Proc. 30th IEEE Symp. on Foundations of Computer Science*, pages 598 – 603, 1989.

[17] M. H. Overmars and J. Leeuwen. Maintenance of configuration in the plane. *Journal of Comput. and System Sciences*, 23:166 – 204, 1981.

[18] F. Preparata and J. Vitter. A simplified technique for hidden-line elimination in terrains. *Proc. of STACS*, 1992.

[19] J. Reif and S. Sen. An efficient output-sensitive hidden-surface removal algorithm and its parallelization. *Proc. of 4th annual symposium on computational geometry*, pages 193 – 200, 1988.

[20] J. Reif and S. Sen. An efficient output-sensitive hidden-surface removal algorithm for polyhedral terrains. *Mathematical Computing Modelling*, 21(5):89 – 104, 1995.

[21] J. H. Reif. *Synthesis of Parallel Algorithms*. Morgan Kaufmann, San Mateo, California, 1993.

[22] A. Schmitt. Time and space bounds for hidden-line and hidden-surface elimination algorithms. *EUROGRAPHICS*, pages 43 – 56, 1981.

[23] Y. Shiloach and U. Vishkin. Finding the maximum, merging and sorting in a parallel computation model. *Journal of algorithms*, 2(1):88 – 102, 1981.

[24] R. F. Sproull, I. E. Sutherland, and R. A. Schumacker. A characterization of ten hidden-surface algorithms. *Computing Surveys*, 6(1):1 – 25, 1974.

[25] R. Tamassia and J. S. Vitter. Optimal parallel algorithms for transitive closure and point location in planar structures. *Proceedings of ACM Symposium on Parallel Algorithm and Architectures*, pages 399 – 408, 1989.

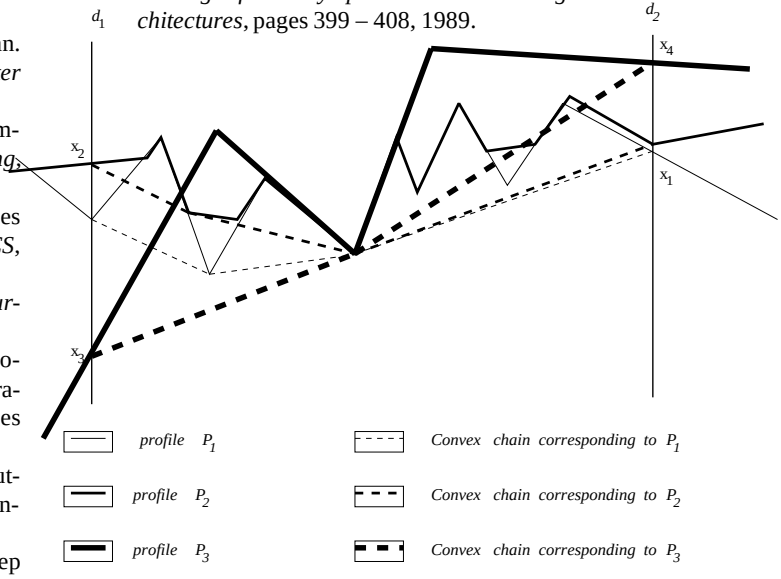

Figure 3.