Computing Four-Body Protein Energy Potentials with Incremental 3D Delaunay Triangulation in CGAL

David O'Brien Jack Snoeyink
Department of Computer Science
University of North Carolina at Chapel Hill
{obrien,snoeyink}@cs.unc.edu

Abstract

We use the CGAL library to improve the speed and efficiency of the lattice chain growth algorithm, an ab initio *method for protein structure prediction and for constructing protein decoys. This Monte Carlo algorithm repeatedly computes the energy of partial protein chains as they are grown one amino acid at a time. Researchers had tried to apply an energy function based on a 3D Delaunay triangulation but could not, due to the cost of triangulating and evaluating each tetrahedron. Using CGAL's incremental 3D Delaunay triangulation routines, we dramatically reduce this cost, allowing the* four-body potential *to be calculated quickly enough for lattice chain growing.*

1 Introduction

One of the greatest outstanding problems in structural biochemistry is the *ab initio* protein folding problem. Simply stated, the goal is to predict the 3D folded state of a protein given only it's amino acid sequence. In this work, we use CGAL to improve the implementation of one well-know *ab initio* method known as the lattice chain growth algorithm. This algorithm produces thousands of decoy structures from which we either pick or derive one or more solutions. The decoys are built with a Monte Carlo procedure that adds successive amino acids into a candidate folded structure.

Fundamental to the chain growth algorithm is the calculation of an energy function. Recent implementations have tried to use the four-body statistical potential function, which is based on the idea of contact energies. When two or more amino acid side chains are in close proximity in the folded state, they contribute some contact energy to the chain. The total energy potential of the folded protein chain is the sum of all the contact energies of nearby clusters of amino acid side chains. This energy function is based on the frequency of observed occurrences of specific clusters of four amino acid side chain types. The 3D Delaunay triangulation of the side chain locations determines the clusters for any chain.

Gan et al. [6, 7] sought to use the four-body potential to evaluate energies for the chain growth algorithm on a lattice, but were unable to do so because of the cost of computing the 3D Delaunay triangulation. Instead they had to use a simpler two-body potential by Miyazawa and Jernigan [10] to build decoys. They could use the four-body potential only for *a postiori* analysis of the decoys. This paper describes how to reduce the cost of the 3D Delaunay triangulation using incremental scheme supported by the CGAL library. These improvements allow the use of four-body potentials in chain growing and in other applications.

The next section gives an overview of protein geometry, the four-body potential, and the details of the chain growth algorithm. Section 3 describes our improved implementation of the algorithm using the CGAL library. Section 4 presents our results.

2 Lattice Chain Growth Algorithm

A protein is a complex 3D structure that is built from a sequence of amino acids attached to a backbone polypeptide chain. The backbone is built of a three-atom sequence, N-C^{α}-C, repeated once for each amino acid. A hydrogen atom is bonded to the first nitrogen atom and to the C^{α} carbon atom while an oxygen atom is double bonded to the second carbon atom. Amino acid side chains are attached to each C^{α} atom on the backbone. See figure 1.

The backbone of a protein can be modeled with all of the atoms described above, or with a simplified model. The lattice chain growth algorithm described in this work uses the C^{α} chain model, which consists of just the C^{α} atom positions. Although this simplification does not capture the side chain positions, it still captures the majority of a protein's folded shape and structure. See figure 2.

Krishnamoorthy et al. and Carter et al. have used *a four-body statistical potential* to distinguish between native and non-native protein structures [2, 9]. The

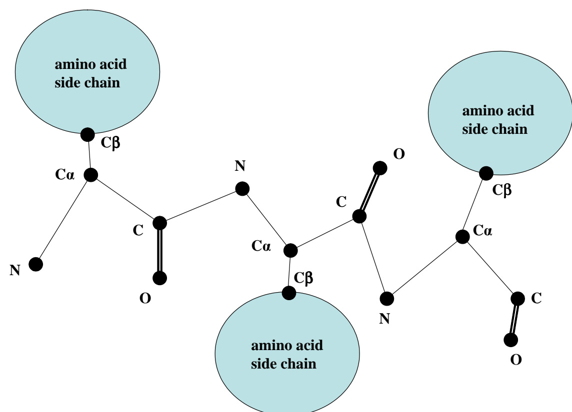

Figure 1. Simple three amino acid protein showing all non-hydrogen atoms in the backbone and large circles for the side chains.

potential scores proteins by using the 3D Delaunay triangulation to group sets of four neighboring amino acid side chains into clusters [11, 9]. Each cluster is classified by its four amino acid types (8855 combinations), and by the adjacency of the amino acids along the backbone (5 cases). In preprocessing, a table is created that stores the log-likelihood of the occurrence of each type of cluster in a training set of proteins. A new protein is then scored by summing the log-likelihood of each of its Delaunay tetrahedra.

The lattice chain growth algorithm grows proteins by successively adding amino acid side chains until the entire sequence is complete. It was originally proposed by Levitt [8]. Gan and Tropsha [6, 7] implemented it using statistical potentials as the energy functions. This implementation constrains the C^α atom locations to the nodes of the 311 lattice, which requires that successive atoms be placed three steps apart in one of the three spatial dimensions and only one step apart in the other two. The basic algorithm is independent of the lattice type and can even be implemented in free space without a lattice. However, the lattice greatly simplifies collision detection and placement of candidate positions. The outline of the algorithm is given below.

1. Place the first two C^α atoms into the lattice.
2. Find all possible lattice locations for the next C^α atom. Discard any grid locations that are less than a prescribed distance from all other C^α atoms.
3. For each location, compute the energy of the partial chain.
4. Convert energy values into transition probabilities.
5. Choose a location based on these probabilities and add the next C^α atom at this location.
6. Loop back to step 2 until all C^α atoms are added.

To build a protein of 100 amino acids typically requires between 1000 and 1500 energy evaluations

Figure 2. Side by side comparison of an all non-hydrogen atom backbone representation on the left and the C^α chain model on the right. Structures are rendered with depth cueing. Thinner and lighter segments are deeper in the background. Two secondary structures, an α helix and a β sheet, are discernible in both models. The protein is 1PNH, a scorpion toxin analog with 31 amino acid residues.

depending on the type of lattice. An equal number of 3D Delaunay triangulations must be calculated if we use the four-body potential as our energy function step 4. This can easily dominate the running time of the algorithm. Indeed, Gan et al. [7] were unable to build with this potential due to their triangulation method. In this work, we use the CGAL 3D Delaunay triangulation data structure to implement an incremental triangulation that overcomes this limitation.

Incremental 3D Delaunay triangulation is difficult because the insertion of a new vertex can radically alter the geometry of the triangulation. Insertion will always create new tetrahedra, and may also force several other tetrahedra to be removed. In the worst case, every tetrahedra may be altered. For this reason, the typical approach to computing the four-body potential has been to rebuild the entire triangulation on each insertion or deletion and then rescore all tetrahedron to avoid either overlooking or double counting tetrahedra. This is wasteful, as typically only a handful of tetrahedra are either created or destroyed with the insertion of a new vertex. A better approach is to determine which tetrahedra would conflict with the new vertex and then subtract their contribution from the current total score. One can then insert the new vertex, create new tetrahedra and add their contribution to the total score.

3 Methods

The CGAL library function *T.find_conflicts(v)* returns a list of tetrahedra that would have to be removed if vertex v were inserted into a 3D Delaunay triangulation, T. We can score these tetrahedra and remove their contributions from the current score of the partial chain without actually removing them. *T.find_conflicts(v)*

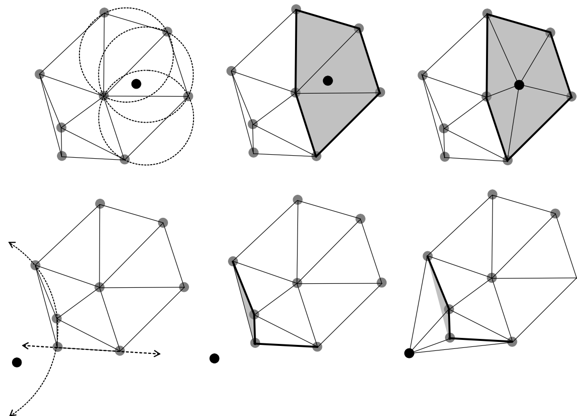

Figure 3. Two 2D examples of *T.find_conflicts(v)*. A new vertex v, black dot, is placed inside the convex hull of the triangulation (top row) and outside the convex hull (bottom row). The left figures show the empty sphere properties violated by v. The middle figures show the result of the function call *T.find_conflicts(v)*, with conflicting triangles in grey and boundary facets in bold. The right figures show the new cells.

also returns a list of triangles, or tetrahedra facets, that define the boundary region of these tetrahedra. These facets tell us what new tetrahedra would need to be added. The three points on each facets would form a new tetrahedron with the vertex v. Therefore, we can calculate the contribution of these new tetrahedra without actually inserting vertex v. See figure 3. We also maintain a hash table that stores the contribution to the potential of previously scored tetrahedra and that uses a CGAL *cell_handle* as an index.

The two new routines *insertNewVertex(tess T, vert v)* and *scoreNewCandidate(tess T, vert v)* are the basis of the incremental 3D Delaunay triangulation algorithm. The algorithm maintains the triangulation of the partial chain, T, the current score, $currentPartialScore$, and the hash table of previously scored tetrahedra. The steps of *insertNewVertex(tess T, vert v)* are as follows.

1. Call *T.find_conflicts(v)* from the CGAL library to get a list of conflicting tetrahedra, $confCells$, and a list of facets defining the boundary of $confCells$.
2. For each conflicting tetrahedron, look up its score in the hash table and subtract it from *currectPartialScore*.
3. Call *T.insert_in_hole(v, confCells)* from the CGAL library to insert v into the triangulation T. This is more efficient than *T.insert(v)* which would internally duplicate the work of *T.find_conflicts(v)* in step 1.
4. For each tetrahedron adjacent to v in the triangulation, compute its score and add it to $curentPartialScore$. Insert each new tetrahedron score into the hash table.

The function *scoreVertexCandidate(tess T, vert v)* returns what would be the total score of the chain if v were inserted into T. It computes the score without actually inserting v into T or scoring each individual tetrahedron in the triangulation. The steps of *scoreVertexCandidate(tess T, vert v)* are as follows.

1. Set *candidateScore* equal to *currentPartialScore*.
2. Call *T.find_conflicts(v)* from the CGAL library to get a list of tetrahedra with conflicts and a list of facets defining the boundary of the regions in conflict.
3. For each conflicting cell, look up its contribution in the hash table and subtract it from *candidateScore*.
4. For each boundary facet, combine its three vertices with v to define a new tetrahedron. Compute this tetrahedron's score and add it to *candidateScore*.
5. Return *candidateScore*.

An alternative to the hash table is to store the scores in the Delaunay cells by adding a new data member. However, we have also implemented a variation of the algorithm that calculates the potential of inserting the next two C^{α} atoms. This requires that the first of these C^{α} atoms actually be inserted to score the candidate configuration. Because this may destroy some cells, we must use an external hash table. The insertion will also invalidate some of our *cell_handle* keys, so we use a different key based on index values stored in the vertices. We choose to be consistent and use a hash table in all cases, but we use the more efficient *cell_handle* key when possible.

Using routines *insertNewVertex(tess T, vert v)* and *scoreVertexCandidate(tess T, vert v)*, we can now modify the lattice chain growth algorithm to use the four-body potential as its energy function.

1. Place the first two C^{α} atoms into the lattice and insert them into the triangulation T.
2. Find all possible lattice locations for the next C^{α} atom. Discard any grid locations that are less than a prescribed distance from all other C^{α} atoms.
3. For each location v, compute the energy of the partial chain with *scoreVertexCandidate(T, v)*.
4. Convert energy values into a transition probabilities.
5. Choose a location based on these probabilities and add the next C^{α} atom at this location. Also add it to triangulation T with *insertNewVertex(T, v)*
6. Loop back to step 2 until all C^{α} atoms are added.

Triangulating points on a regular lattice will lead to many degeneracies. We tried many of the CGAL

Chain	Gan Impl		CGAL Impl		Speedup
Length	MJ	DT	DT	Incr	DT/Incr
30	44.0	27,005.4	4,715.4	151.3	31.2
63	56.8	60,043.9	18,310.2	417.0	43.9
92	65.4	N/A	35,612.9	673.2	52.9
155	102.5	N/A	89,982.6	1,261.7	71.3

Table 1. Running times in seconds to build 1000 decoys. Timings are in seconds for various length chains for the Gan 2-body potential build, the Gan four-body potential build, a CGAL version that calculates a complete triangulation at each step, and the CGAL incremental 3D Delaunay triangulation. The incremental version performs more than 30 times faster in all cases. Timings were done on an SGI Onyx2 R12000 processor at 300 MHz.

kernels that use filtering or exact number types, but were unsatisfied with their speed. We settled on the $Simple_cartesian\langle double\rangle$ kernel and used our own simple perturbation scheme that is very stable. Because we are using a Monte Carlo algorithm that occasionally takes a less than optimal move, we just accept any inaccurate tetrahedra the $Simple_cartesian\langle double\rangle$ kernel may produce. This allows us to meet our primary goals of speed and stability.

4 Results

To measure the improvement in performance due to incremental 3D Delaunay triangulation, we compared four versions of the lattice chain growth algorithm, two from the original program by Gan et al. [7] and two based on our implementation of the algorithm. From the Gan implementation, we tested building with the two-body MJ potential and with the four-body potential. From our CGAL implementation, we tested building with a completely new triangulation for each potential evaluation and with incremental triangulation. We also calculated the speed-up, which is the running time of our version using a complete retriangulation divided by the running time of the incremental version. We built chains of length 30, 63, 92 and 155 with each run building 1000 decoys. The complete timings are given in table 1.

Although the MJ potential version is the fastest, researchers still wish to build with a potential that can more accurately discriminate between folded and unfolded protein chains. Our version using a complete retriangulation is already faster than the Gan implementation, and the incremental version is more than two orders of magnitude faster. Speed-up best measures the advantage of incremental triangulation. The incremental version is more than 30 times faster for short chains and more than 70 times faster for the longest chains. The speed-up increases because the running time of the incremental version grows more slowly than the complete retriangulation version. The cost of a complete triangulation is directly related to the number of amino acids in the chain, while the cost of an incremental insertion remains relatively constant.

5 Acknowledgements

We thank our colleagues in the biogeometry group and in the School of Pharmacy for their input. Special acknowledgement goes to Bala Krishnamoorthy and Alex Tropsha for their work on four-body potentials. Hin Hark Gan provided a great service by giving us his original source code, which greatly aided our implementation of the algorithm. We gratefully acknowledge support from NFS grants 9988742 and 0076984.

References

[1] Branden C., Tooze J. (1999), *Introduction to Protein Structure*, 2nd ed., Garland Publishing.

[2] Carter C. W., LeFebvre C., Cammer S., Tropsha A., Edgell H. M. (2001) Four-body potentials reveal protein-specific correlations to stability changes caused by hydrophobic core mutations. *J Mol Bio*, 311: 625–638.

[3] de Berg M., van Kreveld M., Overmars M., Schwarzkopf O. (2000) Computational Geometry: Algorithms and Applications. Berlin, Germany: Springer-Verlag.

[4] Devillers O. (2002) On deletion in Delaunay triangulation. *Internat J Comput Geom Appl*, 12: 193–205.

[5] Fabri A., Giezeman G., Kettner L., Schirra S., Schonherr S. (2000) On the design of CGAL, a computational geometry algorithms library. *Softw Pract Exper*, 30: 1167–1202.

[6] Gan H. H., Tropsha A., Schlick T. (2000) Generating folded protein structures with a lattice chain growth algorithm. *J Che Phys*, 13: 5511–5524.

[7] Gan H. H., Tropsha A., Schlick T. (2001) Lattice Protein Folding With Two and Four-Body Statistical Potentials. *Proteins*, 43: 161–174.

[8] Hinds D., Levitt M., (1992) A lattice model for protein structure prediction at low resolution. *Proc Natl Acad Sci USA*, 89: 2536–2540.

[9] Krishnamoorthy B., Tropsha A. (2003) Development of a four-body statistical pseudo-potential to discriminate native from non-native protein conformations. *Bioinformatics*, 19: 1540-1548.

[10] Miyazawa S., Jernigan R. (1996) Residue-residue potentials with a favorable contact pair term and an unfavorable high packing density term, for simulation and threading. *J Mol Bio*, 256: 623–644.

[11] Singh R., Tropsha A., Vaisman I. (1996) Delaunay tessellation of proteins. *J Comput Biol*, 3: 213–222.