Efficient Energy Computation for Monte Carlo Simulation of Proteins

Itay Lotan[1] and Fabian Schwarzer[1] and Jean-Claude Latombe[1]

Dept. of Computer Science, Stanford University, Stanford, CA 94305
E-mail: [itayl, schwarzf, latombe]@cs.stanford.edu

Abstract. Monte Carlo simulation (MCS) is a common methodology to compute pathways and thermodynamic properties of proteins. A simulation run is a series of random steps in conformation space, each perturbing some degrees of freedom of the molecule. A step is accepted with a probability that depends on the change in value of an energy function. Typical energy functions sum many terms. The most costly ones to compute are contributed by atom pairs closer than some cutoff distance. This paper introduces a new method that speeds up MCS by efficiently computing the energy at each step. The method exploits the facts that proteins are long kinematic chains and that few degrees of freedom are changed at each step. A novel data structure, called the ChainTree, captures both the kinematics and the shape of a protein at successive levels of detail. It is used to find all atom pairs contributing to the energy. It also makes it possible to identify partial energy sums left unchanged by a perturbation, thus allowing the energy value to be incrementally updated. Computational tests on four proteins of sizes ranging from 68 to 755 amino acids show that MCS with the ChainTree method is significantly faster (as much as 12 times faster for the largest protein) than with the widely used grid method. They also indicate that speed-up increases with larger proteins.

1 Introduction

1.1 Monte Carlo Simulation (MCS)

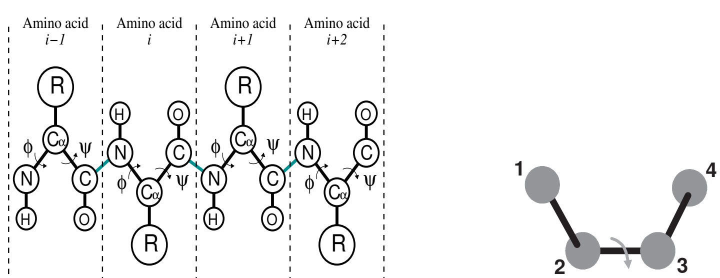
The study of the conformations adopted by proteins is an important topic in structural biology. MCS [1] is one common methodology for this study. In this context, it has been used for two purposes: (1) estimating thermodynamic quantities over a protein's conformation space [2–4] and, in some cases, even kinetic properties [5, 6]; and (2) searching for low-energy conformations of a protein, including its native structure [7–9]. The approach was originally proposed in [10], but many variants and improvements have later been suggested [11].

MCS is a series of randomly generated *trial steps* in the conformation space of the studied molecule. Each such step consists of perturbing some degrees of freedom (DOFs) of the molecule [4–6, 9, 12], in general torsion (dihedral) angles around bonds (see Section 1.2). Classically, a trial step is *accepted* – i.e., the simulation actually moves to the new conformation – with probability $\min\{1, e^{-\Delta E/k_b T}\}$ (the so-called Metropolis criterion [10]), where E is an energy function defined over the conformation space, ΔE is the difference in energy between the new and previous conformations, k_b is the Boltzmann constant, and

T is the temperature of the system. So, a downhill step to a lower-energy conformation is always accepted, while an uphill step is accepted with a probability that goes to zero as the energy barrier grows large. It has been shown that a long MCS with the Metropolis criterion and an appropriate step generator produces a distribution of accepted conformations that converges to the Boltzmann distribution.

The need for general algorithms to speed-up MCS has often been mentioned in the biology literature, most recently in [4]. In this paper, we propose a new algorithm that achieves this goal, independent of a specific energy function, step generator, and acceptance criterion. More precisely, our algorithm reduces the average time needed to decide whether a trial step is accepted, or not, without affecting which steps are attempted, nor the outcome of the acceptance test. It achieves this result by incorporating efficient techniques to incrementally update the value of the energy function during simulation. Although we will describe this algorithm for its application to classic MCS, it could also be used to speed up other kinds of MCS methods, as well as other optimization and sampling techniques. Several such applications will be discussed in Section 8.2.

1.2 Kinematic Structure of a Protein



(a) (b)

Fig. 1. (a) An illustration of a protein fragment with its backbone DOFs. R represents any side-chain (b) A torsional DOF: it is the angle made by the two planes containing the centers of atoms 1, 2, and 3, and 2, 3, and 4, respectively.

A protein is the concatenation of small molecules (the amino acids) forming a long backbone chain with small side chains. Since bond lengths and angles between any two successive bonds are almost constant across all conformations at room temperature [13], it is common practice to assume that the only DOFs of a protein are its torsion angles, also called the internal coordinates. Each amino acid contributes two torsion DOFs to the backbone – the so-called ϕ and ψ angles. See Figure 1 for illustration. Thus, the backbone is commonly modelled as a long chain of links separated by torsion joints (the backbone's DOFs). A link, which designates a rigid part of a kinematic chain, is a group of atoms with no DOFs between them. For example, in the model of Figure 1a, the C and O atoms of amino acid $i-1$ together with the N and H atoms of amino acid i form a link of the protein's backbone, since none of the bonds between them is rotatable. While a backbone may have many DOFs (between 136 and

1510 in the proteins used for the tests reported in this paper), each side-chain has between 0 and 4 torsion DOFs (known as the χ angles). In Figure 1a, these DOFs are hidden inside the ball marked R in each amino acid.

The model of Figure 1a is the most common torsion-DOF representation used in the literature, and is also the one we use in this paper. However, it is possible to apply our algorithm to models that include additional DOFs, such as: ω angles (rotations about the peptide bonds C–N between adjacent amino acids), bond lengths, and bond angles. At the limit, one can make each link a single atom and each joint a rigid-body transform. However, while it is theoretically possible to perform MCS in the Cartesian coordinate space, where each atom has 3 DOFs, it is more efficient to run it in the torsion-DOF space [14]. Hence, the vast majority of MCS are run in this space [4–6, 9, 12].

Due to the chain kinematics of the protein, a small change in one DOF of the backbone may cause large displacements of some atoms. Thus, in an MCS, a high percentage of steps are rejected because they lead to high-energy conformations, in particular conformations with steric clashes (self-collisions). In fact, the rejection rate tends to grow quickly with the number k of DOFs randomly changed in a single step. This fact is well-known in the literature [12, 15] and as a result it is common practice in MCS to change few DOFs (picked at random) at each trial step [4–6, 9, 12, 16, 17].

1.3 Computing the Energy

Various energy functions have been proposed for proteins [16, 18–21]. For all of them, the dominant computation is the evaluation of non-bonded terms, namely energy terms that depend on distances between pairs of non-bonded atoms. These may be physical terms (e.g., van der Waals and electrostatic potentials [20]), heuristic terms (e.g., potentials between atoms that should end up in proximity to each other [18]) and/or statistical potentials derived from a structural database (e.g. [16]).

To avoid the quadratic cost of computing and summing up the contributions from all pairs, cutoff distances are usually introduced, exploiting the fact that physical and heuristic potentials drop off quickly toward 0 as the distance between atoms increases. We refer to the pairs of atoms that are close enough to contribute to the energy function as the *interacting pairs*. Because van der Waals forces prevent atom centers from getting very close, the number of interacting pairs in a protein is often less than quadratic in practice [22].

Hence, one may try to reduce computation by finding interacting pairs without enumerating all atom pairs. A classical method to do this is the grid algorithm (see Section 2), which indexes the position of each atom in a regular three-dimensional grid. This method takes time linear in the number of atoms, which is asymptotically optimal in the worst case. However, it does not exploit an important property of proteins, namely that they form long kinematic chains. It also does not take advantage of the common practice in MCS to change only a few DOFs at each time-step. Moreover, it does not address the remaining problem of efficiently summing up the contributions of the interacting pairs. These issues are addressed in this paper.

1.4 Contributions

A key consequence of making only a small number of DOF changes in a single MCS step is that at every step large fragments of the protein remain rigid.

Hence, at each step, many partial energy sums are unaffected. The grid method re-computes all interacting pairs at each step and cannot directly identify partial sums that have remained constant. Instead, the method proposed in this paper finds the new interacting pairs and retrieves unaffected partial sums without enumerating all interacting pairs. It uses a novel hierarchical data structure – the *ChainTree* – that captures both the chain kinematics and shape of a protein at successive levels of detail. At each step, the ChainTree can be maintained and queried efficiently to find new interacting pairs. It also enables the identification of unchanged partial energy sums stored in a companion data structure — the *EnergyTree* — thus allowing for efficient energy updates throughout the simulation.

Our test results (see Section 6) show that MCS with the ChainTree method is significantly faster than with the grid method when the number k of DOF changes at each step is sufficiently small. We observed speed-ups by factors up to 12 for the largest of the four proteins. Therefore, not only does a small k sharply increases the step acceptance ratio, it also makes it possible to expedite the evaluation of the acceptance criterion. Simulation methodologies other than classical MCS may also benefit from our algorithm (see Section 8.2).

1.5 Outline of Paper

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 presents the ChainTree data structure. Section 4 introduces the algorithm for finding new interacting atom pairs and Section 5 describes how to efficiently update the energy at each simulation step. Section 6 gives experimental results comparing our algorithm with the grid algorithm in MCS. A downloadable version of our algorithm is described in Section 7. Section 8 discusses applications of our algorithm to other MCS methods as well as to other types of molecular simulation methods and points to possible extensions and future directions of research. The application of the ChainTree to testing a long kinematic chain for self-collision was previously presented in [23].

Throughout this paper, we always use n to denote the number of links of a kinematic chain (e.g., a protein's backbone) and k to denote the number of DOF changes per simulation step. Since the number of atoms in any amino acid is bounded by a constant, the number of atoms in a protein is always $O(n)$. Although k can be as big as $O(n)$, it is much smaller in practice [4–6, 9, 12, 15].

2 Related Work

Because biologists are more interested in simulation results than in the computational methods they use to achieve these results, the literature does not extensively describe algorithms for MCS.

A prevailing algorithm – referred to as the *grid algorithm* in this paper – reduces the complexity of finding all interacting pairs in a molecule to asymptotically linear time by indexing the atoms in a regular grid. This approach exploits the fact that van der Waals potentials prevent atom centers from coming very close to one another. In [22] it is formally shown that in a collection B of n possibly overlapping balls of similar radii, such that no two sphere centers are closer than a small fixed distance, the number of balls that intersect any given ball of B is bounded by a constant. This result yields the grid algorithm, which subdivides the 3D space into cubes whose sides are set to the maximum diameter of the balls in B, computes the cubes intersected by each ball, and stores the results in a hash-table. This data structure is re-computed after each step in $\Theta(n)$ time. Determining which balls intersect any given ball of B then takes $O(1)$ time. Hence, finding all pairs of intersecting balls takes $\Theta(n)$ time. The grid method can be used to find all pairs of atoms within some cutoff distance, by growing each atom by half this distance. The method is asymptotically optimal in the worst case, but updating the data structure always takes linear time. This is too costly for very large proteins, making it impractical to perform MCS in this case.

A variant of this method mostly used for Molecular Dynamics simulation maintains, for each atom, a list of atoms within a distance d somewhat larger than the cutoff distance d_c by updating it every s steps [24]. The idea is that atoms further apart than d will not come closer than d_c in less than s steps. There is a tradeoff between s and $d - d_c$ since the larger this difference, the larger the value of s that can be used. However, choosing d big causes the neighbor lists to become too large to be efficient. A method for updating neighbor lists based on monitoring the displacement of each atom is described in [25].

The ChainTree includes a bounding volume hierarchy (BVH) to represent a protein at successive levels of detail. BVHs have been extensively used to detect collisions and compute distances between rigid objects [26–31]. They have been extended in [26, 31, 32] to handle deformable objects by exploiting the facts that topological proximity in the meshed surface of an object is invariant when the object deforms and implies spatial proximity. However, BVH techniques alone lose efficiency when applied to testing a deformable object for self-collision, because they cannot avoid detecting the trivial interaction of each object component with itself. They also lose efficiency when many components move independently.

Finding interacting pairs in a protein is equivalent to finding self-collision in a deformable chain after having grown all links by the cutoff distance. The ChainTree borrows from previous work on BVHs. It uses a BVH based on the invariance of topological proximity along a chain. But it combines it with a transform hierarchy that makes it possible to efficiently prune the search for interacting pairs, when few DOFs change simultaneously.

3 The ChainTree

In this section we describe the ChainTree, the data structure we use to represent a protein. We begin by stating the key properties of proteins and MCS that motivated this data structure (Subsection 3.1). Then follows a description of the two hierarchies that make up the ChainTree. The *transform* hierarchy that approximates the kinematics of the backbone is introduced in Subsections 3.2 and the *bounding-volume* hierarchy that approximates the geometry of the protein is presented in Subsection 3.3. Next, we discuss the representation of the side-chains (Subsection 3.4). Finally, we describe how the two aforementioned hierarchies are combined to form a single balanced binary tree (Subsection 3.5) and the way it is updated (Subsection 3.6).

In the following we refer to the algorithm that updates the ChainTree as the *updating* algorithm and to the algorithm that finds interacting pairs as the *testing* algorithm.

3.1 Properties of Proteins and MCS

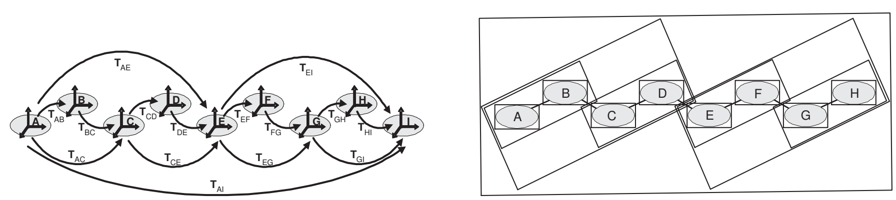
A protein backbone is commonly modelled as a kinematic chain made up of a sequence of n links (atoms or rigid groups of atoms) connected by torsional DOFs. The ChainTree is motivated by three key properties derived from this model:

Local changes have global effects: Changing a single DOF causes all links beyond this DOF, all the way to the end of the chain, to move. Any testing algorithm that requires knowing the absolute position of every link at each step must perform $O(n)$ work at each step even when the number k of DOF changes is $O(1)$.

Small changes may cause large motions: The displacement of a link caused by a DOF change depends not only on the angular variation, but also on the distance between the DOF axis and the link (radius of rotation). So, at each step, any link with a large radius of rotation undergoes a large displacement.

Large sub-chains remain rigid at each step: If we only perturb few DOFs at each step, as is the case during MCS, then large contiguous fragments of the chain remain rigid between steps. So, there cannot be any new interacting pairs inside each of these fragments.

3.2 Transform Hierarchy



(a) (b)

Fig. 2. The two hierarchies. (a) The transform hierarchy: grey ovals depict links; $T_{\alpha\beta}$ denotes the rigid-body transform between the reference frames of links α and β. (b) The bounding volume hierarchy: each BV approximates the geometry of a chain-contiguous sequence of links

We attach a reference frame to each link of the protein's backbone and map each DOF to the rigid-body transform between the frames of the two links it connects. The transform hierarchy is a balanced binary tree of transforms. See Figure 2a, where ovals and labelled arrows depict links and transforms, respectively. At the lowest level of the tree, each transform represents a DOF of the chain. Products of pairs of consecutive transforms give the transform at the next level. For instance, in Figure 2a, T_{AC} is the product of T_{AB} and T_{BC}. Similarly, each transform at every level is the product of two consecutive transforms at the level just below. The root of the tree is the transform between the frames of the first and last links in the chain (T_{AI} in the figure). Each of the $\log n$ levels of the tree can be seen as a chain that has half the links and DOFs of the chain at the level just below it. In total, $O(n)$ transforms are cached in the hierarchy. We say that each intermediate transform $T_{\alpha\beta}$ *shortcuts* all the transforms that are in the subtree rooted at $T_{\alpha\beta}$.

The transform hierarchy is used from the top down by the testing algorithm to propagate transforms defining the relative positions of bounding volumes (from the other hierarchy) that need to be tested for overlap.

3.3 Bounding-Volume Hierarchy

The bounding-volume (BV) hierarchy is similar to those used by prior collision checkers (see Section 2). As spatial proximity in a deformable chain is not invariant, our BVH is based on the proximity of links along the chain. See Figure 2b. Like the transform hierarchy, the BVH is a balanced binary tree. It is constructed bottom up in a "chain-aligned" fashion. At the lowest level, one BV bounds each link. Then, pairs of neighboring BVs at each level are bounded by new BVs to form the next level. The root BV encloses the entire chain. So, at each level, we have a chain with half the number of BVs as the chain at the level below it. This chain of BVs encloses the geometry of the chains of BVs at all lower levels.

The BV type we use is called RSS (for rectangle swept sphere). It was introduced in [29] and is defined as the Minkowski sum of a sphere and a rectangle. The RSS bounding a set of points in 3D is created as follows. The two principal directions spanned by the points are computed and a rectangle is constructed along these directions to enclose the projection of all points onto the plane defined by these directions. The RSS is the Minkowski sum of this rectangle and the sphere whose radius is half the length of the interval spanned by the point set along the dimension perpendicular to the rectangle. To compute the distance between two RSSs, one simply computes the distance between the two underlying rectangles minus the radii of the swept spheres. RSSs offer a good compromise between tightness and efficiency of distance computation. They bound well both globular objects (single atoms, small groups of atoms) and elongated objects (chain fragments). In addition, RSSs are invariant to a rigid-body transform of the geometry they bound.

We construct each intermediate RSS to enclose its two children, thus creating what we term a *not-so-tight* hierarchy (in contrast to a *tight* hierarchy where each BV tightly bounds the links of the sub-chain it encloses). In [23] we show that the size of these BVs does not deteriorate too much as one climbs up the hierarchy. As a result, the shape of the BV stored at each intermediate node depends only on the two BVs held by this node's children.

3.4 Side-Chain Representation

Side-chains, one per amino acid, are short chains with up to 20 atoms, that protrude from the backbone [33]. A side-chain may have some internal torsional DOFs (between 0 and 4). The biology literature proposes different ways to model side-chains ranging from a single sphere approximating the entire side-chain, to a full atomistic model [20, 33]. The choice depends on both the physical accuracy one wishes to achieve and the amount of computation one is willing to pay per simulation step. One may choose to make the side-chains completely rigid, or allow their DOFs to change during the simulation. In both cases we expect the overhead of using a sub-hierarchy for the side-chain atoms to exceed any benefit it may provide. Therefore, we allow each link of the protein backbone to be an aggregate of atoms represented in a single coordinate frame and contained in a BV that is a leaf of the BVH. Each such aggregate includes one or several backbone atoms forming a rigid piece of the backbone and the atoms of the side-chain stemming from it (contained in the circle marked R in Figure 1a).

3.5 Combined Data Structure

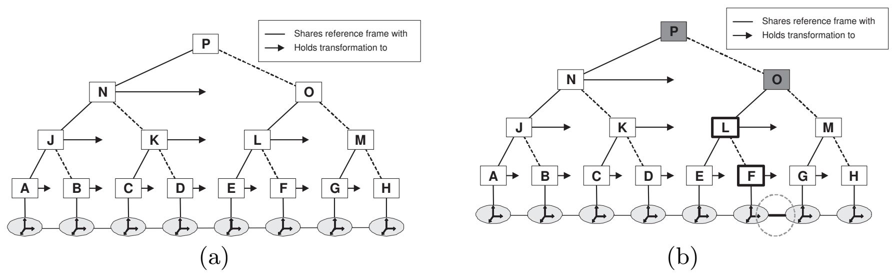


Fig. 3. The ChainTree: (a) a binary tree that combines the transform and BV hierarchies, and (b) after applying a 1-DOF perturbation. The transforms of the nodes with bold contours (F and L) were updated. The BVs of the nodes in grey (O and P) were re-computed.

The ChainTree combines both the transform and the BV hierarchies into a single binary tree as the one depicted in Figure 3a. The leaves of the tree (labelled A through H in the figure) correspond to the links of the protein's backbone with their attached side-chains (using any of the representations discussed above). Each leaf holds both the BV of the corresponding link and the transform (symbolized by a horizontal arrow in the figure) to the reference frame of the next link. Each internal node (nodes J through P) has the frame of the leftmost link in its sub-tree associated with it. It holds both the BV of the BVs of its two children and the transform to the frame of the next node at the same level, if any. The ChainTree contains both pointers from children to parents, which are used by the updating algorithm to propagate updates from the bottom up, as described below, and pointers from parents to children, which are used by the testing algorithm (Section 4).

3.6 Updating the ChainTree

When a change is applied to a single arbitrary DOF in the backbone, the updating algorithm re-computes all transforms that shortcut this DOF and all BVs that enclose the two links connected by this DOF. It does this in bottom-up fashion, by tracing the path from the leaf node immediately to the left of the changed DOF up to the root. A single node is affected at each level. If this node holds a transform, this transform is updated. If it holds a BV that contains the changed DOF, then the BV is re-computed. For example, see Figure 3a. Since the shape of an RSS is invariant to a rigid-body transform of the objects it bounds, all other BVs remain unchanged.

If a DOF is changed in a side-chain, the BV stored at the corresponding leaf node of the ChainTree and the BVs of all the ancestors of this node are re-computed, but *all transforms in the hierarchy remain unchanged.* By updating from the bottom up, each affected transform is re-computed in $O(1)$ time. By using not-so-tight BVs – thus, trading tightness for speed – re-computing each BV is also done in $O(1)$ time. Since the ChainTree has $O(\log n)$ levels, and at each level at most one transform and one BV are updated, the total cost of the update process is $O(\log n)$.

When multiple DOFs are changed simultaneously (in the backbone and the side-chains), the ChainTree is updated one level at a time, starting with the lowest level. Hence, all affected transforms and BVs at each level are updated at most once before proceeding to the next level above it. The total updating time is then $O(k \log (n/k))$. When k grows this bound never exceeds $O(n)$.

The updating algorithm marks every node whose BV and/or transform is re-computed. This mark will be used later by the testing algorithm. See Figure 3b.

4 Finding Interacting Pairs

BVHs have been widely used to detect collision or compute separation distance between pairs of rigid objects, each described by its own hierarchy [27–31, 34]. If each object is divided into small fragments (e.g. links of a chain) the hierarchies are easily applied to finding all pairs of fragments that are closer than some threshold. A simple variant of this algorithm can detect pairs of fragments of the same object that are closer than a threshold by testing the BVH of the object against itself. This variant skips the test of a BV against itself and proceeds directly to testing the BV's children. However, it takes $\Omega(n)$ time, since all leaves will inevitably be visited as each leaf is at distance 0 from itself. The ChainTree allows us to avoid this lower bound by exploiting the third property stated in Section 3.1 — large sub-chains remain rigid between steps.

Since each leaf BV may contain a number of atoms, when two leaf BVs are within the cutoff distance, the interacting atom pairs are found by examining all pairs of atoms, one from each leaf.

When only a small number k of DOFs are changed simultaneously, long sub-chains remain rigid at each step. These sub-chains cannot contain new interacting pairs. So, when we test the BVH contained in the ChainTree against itself, we prune the branches of the search that would look for interacting pairs within rigid sub-chains.

There are two distinct situations where pruning occurs:

1. If the algorithm is about to test a BV against itself and this BV was not updated after the last DOF changes, then the test is pruned.
2. If the algorithm is about to test two different BVs, and neither BV was updated after the last DOF changes, and no backbone DOF between those two BVs was changed, then the test is pruned.

The last condition in this second situation – that no backbone DOF between the two BVs was changed – is slightly more delicate to recognize efficiently. We say that two nodes at the same level in the ChainTree are *separated* if there exists another node between them at the same level that holds a transform that was modified after the last DOF changes. This node will be dubbed *separator*. Hence, if two nodes are separated, a DOF between them has changed. We remark that:

- If two nodes at any level are separated, then any pair consisting of a child of one and a child of the other is also separated.
- If two nodes at any level are *not* separated, then a child of one and a child of the other are separated if and only if they are separated by another child of either parent.

Hence, by pushing separation information downward, the testing algorithm can know in constant time whether a DOF has changed between any two BVs it is about to test. The algorithm also propagates transforms from the transform tree downward to compute the relative position of any two separated BVs in constant time before performing the overlap test.

To illustrate how the testing algorithm works, consider the ChainTree of Figure 3b obtained after a change of the DOF between F and G. F and L are the only separators. The algorithm first tests the BV stored in the root P against itself. Since this BV has changed, the algorithm examines all pairs of its children, (N, N), (N, O) and (O, O). The BV held in N was not changed, so (N, N) is discarded (i.e., the search along this path is pruned). (N, O) is not discarded since the BV of O has changed, leading the algorithm to consider the four pairs of children (J, L), (J, M), (K, L), and (K, M). Both (J, L) and (K, L) satisfy the conditions in the second situation described above; thus, they are discarded. (J, M) is not discarded because J and M are separated by L. The same is true for (K, M), and so on.

In [23] we proved that the worst-case complexity of the testing algorithm to detect self-collision is $\Theta(n^{\frac{4}{3}})$. This bound holds unchanged when using RSSs to compute all interacting pairs. Note, however, that it is an asymptotic bound, which relies on the fact that the number of atoms that may interact with any given atom is bounded by a constant as the size n of the protein *grows arbitrarily large*. For many proteins, this constant may correspond to a significant fraction of n.

The worst-case bound on finding all interacting pairs using the ChainTree hides the practical speed-up allowed by search pruning. We evaluate this speed-up, in a set of benchmarks in Section 6.

5 Energy Maintenance

A typical energy function is of the form $E = E_1 + E_2$, where E_1 sums terms depending on a single parameter commonly referred to as *bonded* terms (e.g., torsion-angle and bond-stretching potentials) and E_2 sums terms commonly known as *non-bonded* terms, which account for interactions between pairs of atoms or atom groups closer than a cutoff distance [5, 6, 12, 16, 17, 19]. Updating E_1 after a conformational change is straightforward. This is done by computing the sum of the differences in energy in the terms affected by the change and adding it to the previous value of E_1. After a k-DOF change, there are only $O(k)$ affected single-parameter terms. So, in what follows we focus on the maintenance of E_2.

At each simulation step we must find the interacting pairs of atoms and change E_2 accordingly. When k is small, many interacting pairs are unaffected by a k-DOF change. The number of affected interacting pairs, though still $O(n)$ in the worst case, is usually much smaller than the total number of interacting pairs at the new conformation. Therefore, an algorithm like the grid algorithm that

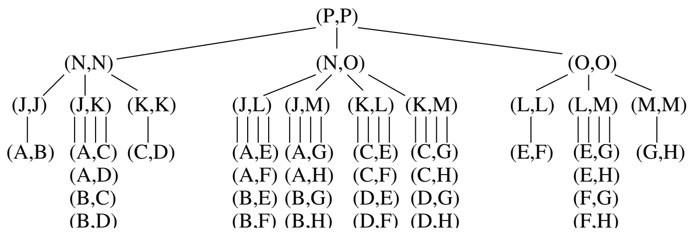


Fig. 4. EnergyTree for the ChainTree of Figure 3a. For simplification, leaves of the form (α, α) are not shown.

computes all interacting pairs at each step is not optimal in practice. Moreover, after having computed the new set of interacting pairs, we still have to update E_2, either by re-computing it from scratch, or by scanning the old and new sets of interacting pairs to determine which terms should be subtracted from the old value of E_2 and which terms should be added to get the new value. In either case, we perform again a computation at least proportional to the total number of interacting pairs. Instead, our method detects partial energy sums unaffected by the DOF change (these sums correspond to interacting pairs where both atoms belong to the same rigid sub-chains). The energy terms contributed by the new pairs are then added to the unaffected partial sums to obtain the new value of E_2. In practice, the total cost of this computation is roughly proportional to the number of *changing* interacting pairs.

Recall that when the testing algorithm examines a pair of sub-chains (including the case of two copies of the same sub-chain), it first tests whether these sub-chains have not been affected by the DOF change and are contained in the same rigid sub-chain. If this is the case, the two sub-chains cannot contribute new interacting pairs, and the algorithm prunes this search path. But, for this same reason, the partial sum of energy terms contributed by the interacting pairs from these sub-chains is also unchanged. So, we would like to be able to retrieve it. To this end, we introduce another data structure, the *EnergyTree*, in which we cache the partial sums corresponding to all pairs of sub-chains that the testing algorithm may possibly examine. Figure 4 shows the EnergyTree for the ChainTree of Figure 3a.

Let α and β be any two nodes (not necessarily distinct) from the same level of the ChainTree. If they are not leaf nodes, let α_l and α_r (resp., β_l and β_r) be the left and right children of α (β). Let $E(\alpha, \beta)$ denote the partial energy sum contributed by all interacting pairs in which one atom belongs to the sub-chain corresponding to α and the other atom belongs to the sub-chain corresponding to β. If $\alpha \neq \beta$, we have:

$$E(\alpha, \beta) = E(\alpha_l, \beta_l) + E(\alpha_r, \beta_r) + E(\alpha_l, \beta_r) + E(\alpha_r, \beta_l). \qquad (1)$$

Similarly, the partial energy sum $E(\alpha, \alpha)$ contributed by the interacting pairs inside the sub-chain corresponding to α can be decomposed as follows:

$$E(\alpha, \alpha) = E(\alpha_l, \alpha_l) + E(\alpha_r, \alpha_r) + E(\alpha_l, \alpha_r). \qquad (2)$$

These two recursive equations yield the EnergyTree.

The EnergyTree has as many levels as the ChainTree. Its nodes at any level are all the pairs (α, β), where α and β are nodes from the same level of the ChainTree. If $\alpha \neq \beta$ and they are not leaves of the ChainTree, then the node (α, β) of the EnergyTree has four children $(\alpha_l, \beta_l), (\alpha_r, \beta_r), (\alpha_l, \beta_r)$, and (α_r, β_l). A node (α, α) has three children $(\alpha_l, \alpha_l), (\alpha_r, \alpha_r)$, and (α_l, α_r). The leaves of the EnergyTree are all pairs of leaves of the ChainTree (hence, correspond to pairs of links of the protein chain). For simplification, Figure 4 does not show the leaves of the form (α, α). Each node (α, β) of the EnergyTree holds the partial energy sum $E(\alpha, \beta)$ after the last accepted simulation step. The root holds the total sum.

At each step, the testing algorithm is called to find new interacting pairs. During this process, whenever the algorithm prunes a search path, it marks the corresponding node of the EnergyTree to indicate that the energy sum stored at this node is unaffected. The energy sums stored in the EnergyTree are updated next. This is done by performing a recursive traversal of the tree. The recursion along each path ends when it reaches a marked node or when it reaches an unmarked leaf. In the second case, the sum held by the leaf is re-computed by adding all the energy terms corresponding to the interacting pairs previously found by the testing algorithm. When the recursion unwinds, the intermediate sums are updated using Equations (1) and (2). In practice, the testing algorithm and the updating of the EnergyTree are run concurrently, rather than sequentially.

The size of the EnergyTree grows quadratically with the number n of links. For most proteins this is not a critical issue. For example, in our experiments, the memory used by the EnergyTree ranges from 0.4 MB for 1CTF ($n = 137$) to 50 MB for 1JB0 ($n = 1511$). If needed, however, memory could be saved by representing only those nodes of the EnergyTree which correspond to pairs of RSSs closer than the cutoff distance.

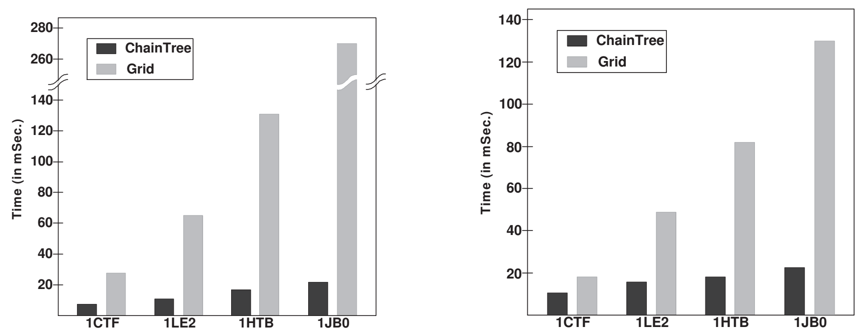
6 Experimental Results for MCS

6.1 Experimental Setup

We implemented ChainTree as described in Section 3. Since each step of an MCS may be rejected, we keep two copies of the ChainTree and the EnergyTree. RSS and distance computation routines were borrowed from the PQP library [27, 29]. We also implemented the grid method (henceforth called Grid) to find interacting pairs by setting the side length of the grid cubes to the cutoff distance. As we mentioned in Section 5, Grid finds all interacting pairs at each step, not just the new ones, and does not cache partial energy sums. So, it computes the new energy value by summing the terms contributed by all the interacting pairs.

Tests were run on a 400 MHz UltraSPARC-II CPU of a Sun Ultra Enterprise 5500 machine with 4.0 GB of RAM.

We performed MCS with the ChainTree and Grid on the four proteins 1CTF, 1LE2, 1HTB and 1JB0 of length 68, 144, 374 and 755 amino-acids respectively, which represent the different sizes of known proteins. The total number of atoms in the MCS was between 487 (1CTF) and 5878 (1JB0). The side-chains were included in the models, as rigid groups of atoms (no internal DOF) and no sub-hierarchies was used to represent each link with its side-chain (see Subsection 3.4). So, if two leaf RSSs are within the cutoff distance, ChainTree finds the interacting pairs from the two corresponding links by examining all pairs



(a) (b)

Fig. 5. Comparing the average time per MCS step of ChainTree and Grid (a) when $k = 1$ and (b) when $k = 5$.

of atoms. The energy function we used for these tests includes a van der Waals (vdW) potential with a cutoff distance of 6Å, an electrostatic potential with a cutoff of 10Å, and a native-contact attractive quadratic-well potential with a cutoff of 12Å. Hence, the cutoff distance for both ChainTree and Grid was set to 12Å.

Each simulation run consisted of 300,000 trial steps. The number k of DOFs changed at each step was constant throughout a run. We performed runs with $k = 1, 5$ and 10. Each change was generated by picking k backbone DOFs at random and changing each DOF independently with a magnitude picked uniformly at random between 0° and 12°. Each run started with a random, partially extended conformation of the protein. Since the vdW term for a pair of atoms grows as $O(d^{-12})$ where d is the distance between the atom centers, it quickly approaches infinity as d becomes small (steric clash). When a vdW term was detected to cross a very large threshold, the energy computation was halted (in both ChainTree and Grid), and the step was rejected.

ChainTree and Grid compute the same energy values for the same protein conformations. Hence, to better compare their performance, we ran the same MCS with both of them on each protein, by starting at the same initial conformation and using the same seed of the random-number generator.

6.2 Results

The results for all the experiments are found in Table 1. Illustrations of the average time results for $k = 1$ and $k = 5$ are presented in Figures 5a and 5b respectively. As expected, ChainTree gave its best results for $k = 1$, requiring on average one quarter of the time of Grid per step for the smallest protein (1CTF) and one twelfth of the time for the largest protein (1JB0). The average number of interacting pairs for which energy terms were evaluated at each step was almost 5 times smaller with ChainTree than with Grid for 1CTF and 30 times smaller for 1JB0.

We see similar results when $k = 5$. In this case, ChainTree was only twice as fast as Grid for 1CTF and 6 times faster for 1JB0. The average number of

	$k = 1$		$k = 5$		$k = 10$	
	CT	Grid	CT	Grid	CT	Grid
1CTF	7.82	27.7	8.34	18.22	12.57	15.07
1LE2	11.16	65.05	14.31	48.84	14.29	27.12
1HTB	16.72	130.9	18.2	81.86	21.75	60.33
1JB0	21.71	271.4	22.18	130.5	29.88	133.8

(a)

	$k = 1$		$k = 5$		$k = 10$	
	CT	Grid	CT	Grid	CT	Grid
1CTF	5,100	25,100	7,400	16,900	8,000	13,500
1LE2	5,100	48,500	6,000	36,600	7,700	23,400
1HTB	5,400	100,000	7,000	56,800	8,200	43,100
1JB0	5,900	200,000	7,000	95,600	10,300	102,000

(b)

Table 1. MCS results: (a) average time per simulation step (in milliseconds) and (b) number of interacting pairs for which energy terms were evaluated per step, when $k =$ 1, 5 and 10. (CT stands for ChainTree.)

	$k = 1$		$k = 5$	
	CT	Grid	CT	Grid
1CTF	12.8	37.2	29.6	37.7
1LE2	20.9	86.5	24.6	65.4
1HTB	26.6	185	51.8	173
1JB0	40.0	401	89.1	348

(a)

	$k = 1$		$k = 5$	
	ChainTree	Grid	ChainTree	Grid
1CTF	8,600	33,300	21,000	34,700
1LE2	9,900	61,900	11,400	47,500
1HTB	9,900	134,000	21,500	129,000
1JB0	12,000	280,000	30,300	248,000

(b)

	unfolded		folded	
	CT	CT+	CT	CT+
1CTF	8.34	2.6	15.74	6.2
1LE2	14.31	6.4	32.37	9.06
1HTB	18.2	9.23	68.92	11.35
1JB0	22.18	6.33	81.15	15.51

(c)

Table 2. (a) Average time (in milliseconds) per step when running an MCS without a threshold on the vdW terms. (b) Average number of interacting pairs evaluated per step for the same simulation. (c) Average running times (in milliseconds) of ChainTree and ChainTree+ per step when the simulations start at unfolded conformations and when they start at the folded conformation of the proteins. (CT stands for ChainTree.)

interacting pairs for which energy terms were evaluated was twice smaller with ChainTree for 1CTF and 14 times smaller for 1JB0. When $k = 10$, the relative effectiveness of ChainTree declined further, being only 1.2 times faster than Grid for 1CTF and 4 times faster for 1JB0. The average number of interacting pairs for which energy terms were evaluated using ChainTree was 60% of the number evaluated using Grid for 1CTF and 10 times smaller for 1JB0.

The larger k, the less effective our algorithm compared with Grid. When k is small, there are few new interacting pairs at each step, and ChainTree is very effective in exploiting this fact. For both ChainTree and Grid the average time per step decreases when k increases. This stems from the fact that a larger k is more likely to yield over-threshold vdW terms and so to terminate energy computation sooner.

In order to examine the full effect of reusing partial energy sums, we re-ran the simulations for the four proteins without the vdW threshold for $k = 1$ and 5. The results are presented in Tables 2a and 2b. Removing the vdW threshold does not significantly alter the behavior of the algorithms. The average time per step is of course larger, since no energy computation is cut short by a threshold crossing. The relative speed-up of ChainTree over Grid is only slightly smaller without the threshold.

6.3 Two-pass ChainTree

In the previous MCS the percentage of steps that were rejected before energy computation completed, due to an above-threshold vdW term for 1CTF, for example, rose from 60% when $k = 1$ to 98% when $k = 10$. This observation not only motivates choosing a small k. It also suggests the following two-pass approach. In the first pass, ChainTree uses a very small cutoff distance chosen such that atom pairs closer than this cutoff yield above-threshold vdW terms.

In this pass, the algorithm stops as soon as it finds an interacting pair, and then the step is rejected. In the second pass the cutoff distance is set to the largest cutoff over all energy terms and ChainTree computes the new energy value. We refer to the implementation of this two-pass approach as ChainTree+.

We compared ChainTree and ChainTree+ by running an MCS of 300,000 trial steps with $k = 5$ and measuring the average time per step. The results for the four proteins are given in Table 2c. We ran two different simulations for each protein. One that started at a partially extended conformation and another that started at the folded state of the protein. Hence, the conformations reached in the first case were less compact than in the second case. Consequently, the rejection rate due to self-collision was higher in the second case. While ChainTree+ is faster in both cases, speed-up factors are greater (as much as 5) when starting from the folded state.

7 MCS Software

We have extended our implementation of the ChainTree algorithm to include a physical, full-atomic energy function. We chose to implement the force-field *EEF1* [19]. This force field is based on the CHARMM19 potential energy function [35] with an added implicit solvent term. We chose *EEF1* because it has been shown to discriminate well between folded and misfolded structures [36]. It is well suited for ChainTree because its implicit solvent term is pairwise and thus our algorithm can compute it efficiently. We have packaged our software into a program that runs fast MCS. It can be downloaded from `http://robotics.stanford.edu/~itayl/mcs`.

This software loads an initial structure that is described in terms of its amino acid sequence and the corresponding backbone angles of each residue. It then performs a classical MCS. The user can control some parameters of the simulation (e.g. the number of angles to change, the length of the simulation, the temperature ...) by specifying them on the command line.

8 Conclusion

8.1 Summary of Contribution

This paper presents a novel algorithm based on the ChainTree and EnergyTree data structures to reduce the average step time of MCS of proteins, independent of the energy function, step generator, and acceptance criterion used by the simulator. Tests show that, when the number of simultaneous DOF changes at each step is small (as is usually the case in MCS), the new method is significantly faster than previous general methods — including the worst-case optimal grid method — especially for large proteins. This increased efficiency stems from the treatment of proteins as long kinematic chains and the hierarchical representation of their kinematics and shape. This representation — the ChainTree — allows us to exploit the fact that long sub-chains stay rigid at each step, by systematically re-using unaffected partial energy sums cached in a companion data structure — the EnergyTree. Our tests also demonstrate the advantage of using the ChainTree to detect steric clash before computing the energy function.

8.2 Other Applications

Although we have presented the application of our algorithm to classical Metropolis MCS, it can also be used to speed up other MCS methods as well as other optimization and simulation methodologies.

For example, MCS methods that use a different acceptance criterion can benefit from the same kind of speed-up as reported in Section 6, since the speed-up only derives from the faster maintenance of the energy function when relatively few DOFs are changed simultaneously, and is independent of the actual acceptance criterion. Such methods include Entropic Sampling MC [3], Parallel Hyperbolic MC [4], and Parallel-hat Tempering MC [9]. MCS methods that use Parallel Tempering [2] (also known as Replica Exchange) such as [4, 9], which require running a number of replicas in parallel, could also benefit by using a separate ChainTree and EnergyTree for each replica.

Some MCS methods use more sophisticated move sets (trial step generators). Again, our algorithm can be applied when the move sets do not change many DOFs simultaneously, which is in particular the case of the moves sets proposed in [7, 8] (biasing the random torsion changes), and in [37] and [38] (moves based on fragment replacement). More computationally intensive step generators use the internal forces (the gradient of the energy function) to bias the choice of the next conformation (e.g., Force-Biased MC [39], Smart MC [40] and MC plus minimization [12]). For such step generators, the advantage of using our algorithm is questionable, since they may change all DOFs at each step.

Some optimization approaches could also benefit from our algorithm. For instance, a popular one uses genetic algorithms with crossover and mutation operators [41–43]. The crossover operator generates a new conformation by combining two halves, each extracted from a previously created conformation. Most mutation operators also reuse long fragments from one or several previous conformations. For both types of operators, our algorithm would allow partial sums of energy terms computed in each fragment to be re-used, hence saving considerable amounts of computation.

8.3 Current and Future Work

We are currently using our algorithm to run MCS of proteins on the order of 100 residues and larger using a full-atomic model and a physical energy function (*EEF1* [19]). To the best of our knowledge this has not been attempted so far. We also intend to perform MCS of systems of several small proteins, in order to study protein misfolding, which is known to cause diseases such as Alzheimer. Each protein in the system will have its own ChainTree, which will be used to detect interaction both within each molecule and between molecules.

One possible extension of our work would be to use the ChainTree to help select "better" simulation steps. Indeed, the rejection rate in MCS becomes so high for compact conformations that simulation comes to a quasi standstill. This is a known weakness of MCS, which makes it less useful around the native conformation. This happens because almost any DOF change causes a steric clash. To select DOF changes less likely to create such clashes, one could precompute the radius of rotation of every link relative to each DOF. These radii and the distances between interacting atom pairs at each conformation would allow computing the range of change for each DOF such that no steric clash will occur.

Another natural extension, which exploits the hierarchical nature of the ChainTree, is to vary the resolution of the molecular representation as MCS progresses. This could be accomplished by changing on the fly the level of the ChainTree that is considered the leaf level (the bottom level). For full atomic resolution, searches in the ChainTree would continue until reaching the absolute bottom level. If a coarser resolution can be tolerated, the search could be stopped at a higher level in the hierarchy, where each node represents one or some amino acids. A different energy function could then be used for each level of resolution. This scheme could entail large savings in CPU time in regions of the conformation space where the protein structure is not very compact, while not compromising precision in other regions when it is needed.

Acknowledgements: This work was partially funded by NSF ITR grant `CCR-0086013` and a Stanford BioX Initiative grant.

References

1. Binder, K., Heerman, D.: Monte Carlo Simulation in Statistical Physics. 2nd edn. Springer Verlag, Berlin (1992)
2. Hansmann, U.: Parallel tempering algorithm for conformational studies of biological molecules. Chemical Physics Letters **281** (1997) 140–150
3. Lee, J.: New Monte Carlo algorithm: entropic sampling. Physical Review Letters **71** (1993) 211–214
4. Zhang, Y., Kihara, D., Skolnick, J.: Local energy landscape flattening: Parallel hyperbolic Monte Carlo sampling of protein folding. Proteins **48** (2002) 192–201
5. Shimada, J., Kussell, E., Shakhnovich, E.: The folding thermodynamics and kinetics of crambin using an all-atom Monte Carlo simulation. J. Mol. Bio. **308** (2001) 79–95
6. Shimada, J., Shakhnovich, E.: The ensemble folding kinetics of protein G from an all-atom Monte Carlo simulation. Proc. Natl. Acad. Sci. **99** (2002) 11175–80
7. Abagyan, R., Totrov, M.: Biased probability Monte Carlo conformational seraches and electrostatic calculations for peptides and proteins. J. Mol. Bio. **235** (1994) 983–1002
8. Abagyan, R., Totrov, M.: Ab initio folding of peptides by the optimal-bias Monte Carlo minimization procedure. J. of Computational Physics **151** (1999) 402–421
9. Zhang, Y., Skolnick, J.: Parallel-hat tempering: A Monte Carlo search scheme for the identification of low-energy structures. J. Chem. Phys. **115** (2001) 5027–32
10. Metropolis, N., Rosenbluth, A., Rosenbluth, M., Teller, A., Teller, E.: Equation of state calculations by fast computing machines. J. Chem Phys **21** (1953) 1087–1092
11. Hansmann, H., Okamoto, Y.: New Monte Carlo algorithms for protein folding. Current Opinion in Structural Biology **9** (1999) 177–183
12. Li, Z., Scheraga, H.: Monte Carlo-minimization approach to the multiple-minima problem in protein folding. Proc. National Academy of Science. **84** (1987) 6611–15
13. Grosberg, A., Khokhlov, A.: Statistical physics of macromolecules. AIP Press, New York (1994)
14. Northrup, S., McCammon, J.: Simulation methods for protein-structure fluctuations. Biopolymers **19** (1980) 1001–1016
15. Abagyan, R., Argos, P.: Optimal protocol and trajectory visualization for conformational searches of peptides and proteins. J. Mol. Bio. **225** (1992) 519–532
16. Kikuchi, T.: Inter-Ca atomic potentials derived from the statistics of average interresidue distances in proteins: Application to bovine pancreatic trypsin inhibitor. J. of Comp. Chem. **17** (1996) 226–237
17. Kussell, E., Shimada, J., Shakhnovich, E.: A structure-based method for derivation of all-atom potentials for protein folding. Proc. Natl. Acad. Sci. **99** (2002) 5343–8

18. Gō, N., Abe, H.: Noninteracting local-structure model of folding and unfloding transition in globular proteins. Biopolymers **20** (1981) 991–1011
19. Lazaridis, T., Karplus, M.: Effective energy function for proteins in solution. Proteins **35** (1999) 133–152
20. Leach, A.: Molecular Modelling: Principles and Applications. Longman, Essex, England (1996)
21. Sun, S., Thomas, P., Dill, K.: A simple protein folding algorithm using a binary code and secondary structure constraints. Protein Engineering **8** (1995) 769–778
22. Halperin, D., Overmars, M.H.: Spheres, molecules and hidden surface removal. Comp. Geom.: Theory and App. **11** (1998) 83–102
23. Lotan, I., Schwarzer, F., Halperin, D., Latombe, J.C.: Efficient maintenance and self-collision testing for kinematic chains. In: Symp. Comp. Geo. (2002) 43–52
24. Thompson, S.: Use of neighbor lists in molecular dynamics. Information Quaterly, CCP5 **8** (1983) 20–28
25. Mezei, M.: A near-neighbor algorithm for metropolis Monte Carlo simulation. Molecular Simulations **1** (1988) 169–171
26. Brown, J., Sorkin, S., Latombe, J.C., Montgomery, K., Stephanides, M.: Algorithmic tools for real time microsurgery simulation. Med. Im. Ana. **6** (2002) 289–300
27. Gottschalk, S., Lin, M.C., Manocha, D.: OBBTree: A hierarchical structure for rapid interference detection. Comp. Graphics **30** (1996) 171–180
28. Klosowski, J.T., Mitchell, J.S.B., Sowizral, H., Zikan, K.: Efficient collision detection using bounding volume hierarchies of k-DOPs. IEEE Tr. on Visualization and Comp. Graphics **4** (1998) 21–36
29. Larsen, E., Gottschalk, S., Lin, M.C., Manocha, D.: Fast distance queries with rectangular swept sphere volumes. In: IEEE Conf. on Rob. and Auto. (2000)
30. Quinlan, S.: Efficient distance computation between non-convex objects. In: IEEE Intern. Conf. on Rob. and Auto. (1994) 3324–29
31. van den Bergen, G.: Efficient collision detection of complex deformable models using AABB trees. J. of Graphics Tools **2** (1997) 1–13
32. Guibas, L.J., Nguyen, A., Russel, D., Zhang, L.: Deforming necklaces. In: Symp. Comp. Geo. (2002) 33–42
33. Creighton, T.E.: Proteins : Structures and Molecular Properties. 2nd edn. W. H. Freeman and Company, New York (1993)
34. Hubbard, P.M.: Approximating polyhedra with spheres for time-critical collision detection. ACM Tr. on Graphics **15** (1996) 179–210
35. Brooks, B., Bruccoleri, R., Olafson, B., States, D., Swaminathan, S., Karplus, M.: CHARMM: a program for macromolecular energy minimizationand dynamics calculations. J. of Computational Chemistry **4** (1983) 187–217
36. Lazaridis, T., Karplus, M.: Discrimination of the native from misfolded protein models with an energy funbction including implicit solvation. J. Mol. Bio. **288** (1998) 477–487
37. Elofsson, A., LeGrand, S., Eisenberg, D.: Local moves, an efficient method for protein folding simulations. Proteins **23** (1995) 73–82
38. Simons, K., Kooperberg, C., Huang, E., Baker, D.: Assembly of protein tertiary structure from fragments with similar local sequences using simulated annealing and bayesian scoring functions. J. Mol. Bio. **268** (1997) 209–225
39. Pangali, C., Rao, M., Berne, B.J.: On a novel Monte Carlo scheme for simulating water and aqueous solutions. Chemical Physics Letters **55** (1978) 413–417
40. Kidera, A.: Smart Monte Carlo simulation of a globular protein. Int. J. of Quantum Chemistry **75** (1999) 207–214
41. Pedersen, J., Moult, J.: Protein folding simulations with genetic algorithms and a detailed molecular description. J. Mol. Bio. **269** (1997) 240–259
42. Sun, S.: Reduced representation model of protein structure prediction: statistical potential and genetic algorithms. Protein Science **2** (1993) 762–785
43. Unger, R., Moult, J.: Genetic algorithm for protein folding simulations. J. Mol. Bio. **231** (1993) 75–81