

Rotamer-Pair Energy Calculations Using a Trie Data Structure

Andrew Leaver-Fay¹, Brian Kuhlman², and Jack Snoeyink¹

¹ Department of Computer Science, University of North Carolina at Chapel Hill

² Department of Biochemistry, University of North Carolina at Chapel Hill

Abstract. Protein design software places amino acid side chains by pre-computing rotamer-pair energies and optimizing rotamer placement. If the software optimizes by rapid stochastic techniques, then the precomputation phase dominates run time. We present a new algorithm for rapid rotamer-pair energy computation that uses a trie data structure. The trie structure avoids redundant energy computations, and lends itself to time-saving pruning techniques based on a simple geometric criteria. With our new algorithm, we compute rotamer-pair energies nearly 4 times faster than the previous approach.

1 Introduction

Researchers have recently achieved notable success in computational protein design. Homme Hellinga’s lab redesigned the active site of Ribose Binding Protein to bind TNT [1, 2]. David Baker’s lab designed a more stable protein-L and created a novel protein fold [3, 4]. The two labs solve a common subproblem: with a fixed protein backbone as scaffold, they search for a side chain placement that packs them snugly without collisions. The snugness-of-fit is captured by an energy function. The problem of minimizing the energy function over all side chain conformations is known as the side chain placement problem.

Side chain conformational flexibility is typically modeled by creating many possible atom placements. Each side chain may be modeled with bond lengths and bond angles fixed to standard or experimentally determined values; selected torsional (or dihedral) angles that are variable give the flexibility. These angles have preferred values that have been observed within the Protein Databank (PDB) [5] and confirmed through quantum mechanical calculations [6]. Designers sample the continuous torsional space near these torsional angles’ preferred values to generate “rotamers:” conformational *isomers* that differ by torsional *rotations*. Scientists have collected preferred side chain conformations into rotamer libraries [7–10].

Designers divide the side chain placement problem into two phases. In phase 1, they precompute all possible rotamer-pair interaction energies for their rotamer library, and in phase 2, they search for the (globally) optimal side chain placement. Significant work has gone into exact algorithms for the side chain placement problem [11–15]. Still, the problem is NP-Complete [16], and many re-

searchers choose fast stochastic optimization techniques [17–21]. The rotamer-pair energy computation of phase 1 can be a significant fraction of the running time for both techniques, and usually dominates the running time of stochastic techniques. Thus, in this paper, we address rotamer-pair energy computation.

The interaction energy between two rotamers, A and B , is the sum of the atom/atom interaction energies over all atoms of A with all atoms of B . When a pair of rotamers on the same residue share torsional angles, they share atoms. Repeated atoms imply repeated atom/atom energy evaluations when computing all rotamer-pair energies

The obvious way to avoid repeating atom/atom energy computations is to store in a table the result of atom/atom energy computations for unique atom pairs. When a unique atom pair is encountered for the first time, calculate the pair’s interaction energy and store it. When a unique atom pair is encountered any subsequent time, simply look up the old result. However, with a moderately large rotamer library of 2K rotamers, which we use as our running example, a single residue can generate $\sim 10\text{K}$ unique atoms. A unique-atom by unique-atom table with $10\text{K} \times 10\text{K}$ entries would occupy 400 MB. This table does not fit in a processor’s cache (~ 512 KB). Although storing energies avoids repeated computation, retrieving the table entries incurs cache misses, eroding any savings in running time.

We use a trie to represent all the rotamers on a single residue. With a pair of these “rotamer tries” we can rapidly compute the rotamer-pair energies, while reducing our memory usage. We have implemented our algorithm within the Rosetta molecular modeling software [17]. Because our memory use is minimal, and because we reuse atom/atom energy computations, our algorithm runs nearly 4 times faster than Rosetta’s existing method.

2 Methods

2.1 Rotamers and Tries

As mentioned in the introduction, rotamer libraries are usually built by sampling certain torsional (or dihedral) angles; these are denoted by χ_1, χ_2, \dots in order from the protein backbone. The most flexible amino acids, lysine and arginine, have four χ dihedrals. Rotamers of the same amino acid on the same residue that share a prefix of χ dihedrals place many of their atoms in the same position. For instance, two leucine rotamers that share a χ_1 dihedral place their $C_\beta, 1H_\beta, 2H_\beta$ and C_γ atoms identically. If we order atoms by distance from the backbone as well, then the shared atoms are also a prefix. Trie data structures are perfect for capturing shared prefixes.

A “trie” is a rooted tree. Each node in the trie contains an object. Each root-to-leaf path in the trie represents a string of objects. Tries are often used to represent dictionaries. In a dictionary trie, each node represents a letter. Each root-to-leaf path represents a word. For example, consider a dictionary containing just two words: ‘apple’ and ‘apply.’ The root would be the letter ‘a’.

The shared prefix ‘appl’ would lie along an unbranched path of the trie. The ‘l’ node would have two children, ‘e’ and ‘y’. The path to the leaf node ‘y’ from the root spells out the word ‘apply’.

In a rotamer trie, each node contains an atom. Each root-to-leaf path represents a rotamer. We depict a few rotamer tries in Figure 1. Note that the trie connectivity does not reflect the amino acids’ chemical structure. Note also that the three threonine rotamers depicted differ only in their hydrogen position: their substantial shared prefix lets us save space. The trie for arginine in Figure 1(b&c) shows the trie branching produced by its four χ dihedrals. A trie for a complete rotamer set would be too large to display.

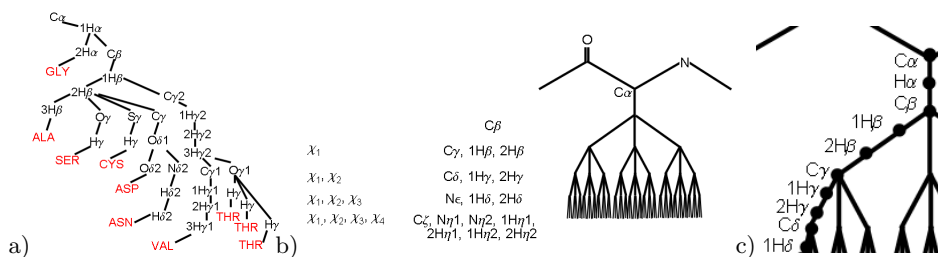


Fig. 1. Two example tries. a) One rotamer from each of the small amino acids and three for threonine. b) A set of arginine rotamers showing the branching pattern for arginine’s four χ dihedrals. c) Angle χ_1 determines the coordinates of $1H_\beta$, $2H_\beta$ and C_γ , so they lie together along a path.

Tries have proven useful in a number of other string problems in computational biology [22–24]. Homme Hellinga previously introduced representing rotamer sets in a trie-like structure to weed out rotamers colliding with the background [25] but does not use tries to compute energies.

It is with a pair of rotamer tries that we compute rotamer-pair energies. We have implemented our algorithm to be compatible with the energy function from Rosetta. We describe the details of Rosetta’s energy function, and the existing rotamer-pair energy subroutine in the next section.

2.2 Rosetta’s Energy Function

Rosetta has four terms that apply on an atom-by-atom basis. Between all heavy atom pairs, Rosetta includes three terms: a van der Waal’s attractive term, a van der Waal’s repulsive term, and a Lazaridis-Karplus implicit solvation [26] term. Each of these terms depend on the atom types of the two heavy atoms, and their distance. For speed, Rosetta uses a maximum distance threshold of 5.5 Å: if two heavy atoms are further than 5.5 Å apart, then their interaction energy is zero.

Between hydrogen/other atom pairs two terms apply: a van der Waal’s repulsive term, and a statistically derived hydrogen bonding term [27]. The hydrogen bonding term is usually described by four atoms acting simultaneously: the donor

hydrogen, the donor heavy atom, the acceptor and the acceptor-base. The term depends on one distance and the cosine of two angles: the hydrogen-acceptor distance, the cosine of the donor heavy atom—hydrogen—acceptor angle and the cosine of the hydrogen—acceptor—acceptor-base angle. We reformulate the hydrogen bond function to depend on only two atoms, the hydrogen and the acceptor, by including orientation vectors with each atom. The orientation vectors allow us to compute the two cosines needed.

Because Rosetta’s terms involving hydrogens are so short-ranged, Rosetta’s developers use a distance threshold between two heavy atoms to determine if their attached hydrogen atoms could be close enough to interact. If two heavy atoms are further than 4.6 Å apart, then all hydrogen/other atom pairs for the attached hydrogens have zero interaction energy.

Rosetta’s existing rotamer-pair energy function, `get_energies()`, takes two rotamers sets, R and S , and outputs their rotamer-pair energies into a rotamer/rotamer energy table (rot_rot_E). We describe `get_energies()` with the following pseudocode:

```

get_energies(R, S)
  for i = 1 : R.num_rotamers
    for j = 1 : S.num_rotamers
      if cbeta_dis( i, j ) > threshold(amino_acid(i), amino_acid(j) )
        continue;
      energy_sum = 0
      for k = 1 : num_heavy_atoms(i)
        for l = 1 : num_heavy_atoms(j)
          energy_sum += atom_atom_energy( R.atom(i,k), S.atom(j,l) );
          if dis( R.atom(i,k), S.atom(j,l) ) < 4.6
            energy_sum += calc_attached_h_energies( R.atom(i,k), S.atom(j,l) );
        rot_rot_E[ i, j ] = energy_sum;
      return;

```

where `calc_attached_h_energies(k, l)` iterates over the hydrogen/heavy atom pairs and hydrogen/hydrogen atom pairs calling `atom_atom.energy()` for the heavy atoms k and l and their attached hydrogen atoms. Because hydrogens make up roughly half of the atoms in a rotamer, it would be roughly 4 times more expensive to evaluate all atom/atom energies as it would be to evaluate all heavy atom/heavy atom energies, descending into the hydrogens only as needed.

Rosetta does not compute rotamer-pair energies between rotamers if their C_β atoms are so distant that it is impossible for any pair of rotamers of those two amino acid types to interact. Rosetta uses its 5.5 Å heavy atom distance cutoff to calculate these thresholds.

2.3 Trie Node

The trie data structure stores everything we need for evaluating Rosetta’s energy function. It also stores a number of variables needed to prune energy computations, which we describe after the algorithm.

We represent our trie as an array. We store the nodes in their preorder traversal order. In the recursive description of the algorithm that we give below (Sec. 3.1), we refer to child pointers as if they were explicit. However, we store the depth of each node in the tree instead of explicit child pointers. The

preorder/depth representation is sufficient to completely describe the trie structure.

We store the atom type for each atom, its xyz coordinate, and its orientation vector. In a redundant, but time-saving extension of the atom type, we keep several boolean flags: `is_backbone`, `is_heavy_atom`, `is_acceptor`, `is_donor_h`, etc. Each of these flags is stored as a single bit. Each bit value is determined by the atom type and is therefore redundant. However, the logic is somewhat complex to convert between atom type and these boolean values. Instead of evaluating the conversion functions during the trie traversal $O(n^2)$ times, we evaluate the value of these flags outside of the main loop and store them compactly in the trie node.

When we prune, we use 40 bytes per node. The last three variables in the `trie_node` are needed only for pruning. The “no pruning” implementation does not allocate space for these three variables, and so the cost per `trie_node` drops to 32 bytes. We have found it especially important to make sure our trie nodes align with the 32-bit memory boundaries.

```

struct trie_node
  float[3]      xyz;                //12 bytes
  float[3]      o_vector;           //12 bytes
  unsigned char atom_type;          // 4 bytes
  unsigned char depth;
  unsigned char hv_depth;
  unsigned char flags;
  unsigned short flags2;            // 4 bytes
  unsigned short hybridization;
  unsigned short rotamers_in_subtree; // 4 bytes
  unsigned short sibling;
  float         subtree_radius;     // 4 bytes

```

3 Interaction energy between two rotamer tries

We now give the algorithm to calculate the rotamer-pair energies between two tries, R and S . The idea is simple, we perform a preorder traversal of R , and for each atom $r \in R$, we perform a preorder traversal of S . We evaluate `atom_atom_energy(r, s)` for each pair of atoms we encounter ($s \in S$). (We refer to the preorder traversal order when we use the words ‘before’, ‘preceed’ and ‘after’ below.)

To calculate the rotamer-pair energies, we use two recursive functions: `atom_vs_trie()` and `trie_vs_trie()`. For clarity we describe these functions recursively; for speed we implement them iteratively.

- `atom_vs_trie($r, s, ancestral_E$)` recursively computes the interaction energy between atom r and all the rotamers in the subtree of S rooted at node s . It stores these energies in a global variable, `AREnergies`, a stack of arrays. `atom_vs_trie()` calls `atom_atom_energy()` and is called by `trie_vs_trie(r, S)`.
- `trie_vs_trie(r, S)` recursively computes the interaction energy between the rotamers in the subtree of R rooted at node r and the rotamers in the trie S . It stores these energies in a global variable, `REnergies`, the table of rotamer/rotamer energies. An invocation of `trie_vs_trie($R.root, S$)` calculates all rotamer-pair energies. `trie_vs_trie()` invokes `atom_vs_trie()`.

3.1 Functions in detail

Global variables We use three global variables in these recursive functions:

- RREnergies. Rotamer/Rotamer Energies. This table has ($R.\text{num_rotamers} \times S.\text{num_rotamers}$) entries, one for each rotamer pair.
- AREnergies. Atom/Rotamer Energies. This is a stack of arrays. Each array contains $S.\text{num_rotamers}$ entries and holds r 's ancestors' interaction energies with rotamers of S . The stack height is limited to the maximum number of ancestors with siblings of any leaf in a rotamer tree.
- ARStackTop. Top of stack pointer for AREnergies.

atom_vs_trie(r , s , ancestral_E) in detail Precondition: r is an atom of R , s is an atom of S . ancestral_E holds the sum of the interaction energies r has with all ancestors of s . AREnergies[ARStackTop] contains the sum of the interaction energies of all of r 's ancestors with the rotamers of S that terminate at or after s , and contains the sum of r 's ancestors' and r 's interaction energies for all rotamers of S that terminate before s .

Postcondition: AREnergies[ARStackTop] contains the sum of interaction energies of r and its ancestors with the rotamers of S that terminate before s or terminate in s 's subtree. AREnergies[ARStackTop] contains the sum of the interaction energies of all other rotamers in S with r 's ancestors only.

Pseudocode:

```
atom_vs_trie(r, s, ancestral_E)
  ancestral_E += atom_atom_energy(r, s);
  if (s.terminal_rotamer_id != -1)
    AREnergies[ARStackTop][s.terminal_rotamer_id] += ancestral_E;
  for (int i = 0; i < s.num_children; i++)
    atom_vs_trie(r, s.child[i], ancestral_E);
  return;
```

trie_vs_trie(r , S) in detail Precondition: r is an atom of R . AREnergies[ARStackTop] contains the sum of the interaction energies of r 's ancestors with the rotamers of S . If r is the root, then ARStackTop must be zero and each entry in AREnergies[0] is zero.

Postcondition: RREnergies contains the interaction energies for all rotamers of S and the rotamers of R that terminate in the subtree of R rooted at r .

Pseudocode:

```
trie_vs_trie(r, S)
  atom_vs_trie(r, S.root, 0);
  if (r.terminal_rotamer_id != -1)
    //copy entire AREnergies row
    RREnergies[r.terminal_rotamer_id] = AREnergies[ARStackTop];
  if (r.num_children > 0)
    ARStackTop++;
    for (int i = 0; i < r.num_children-1, i++)
      //copy stack top for children with siblings
      AREnergies[ARStackTop] = AREnergies[ARStackTop - 1];
      trie_vs_trie(r.child[i], S);
    ARStackTop--;
    //last child doesn't need its own stack copy
    trie_vs_trie(r.child[r.num_children - 1], S);
  return;
```

Because we traverse S repeatedly, it is critical that S fit inside the processor’s cache. The size of each node in the trie is 40 bytes. In our example rotamer set with 10K unique atoms, S would occupy 400KB. Since most cache sizes are 512KB, S fits comfortably. AREnergies’s size would be 4 rows \times 2K rotamers/row \times 4 bytes/float = 32KB. There are only 4 rows in AREnergies since the most flexible amino acids have only 4 χ dihedrals.

3.2 Pruning Computations

We can use the tree structure of the two tries R and S to avoid performing many of the atom/atom energy computations. Suppose we are somewhere in the middle of the trie traversals, examining atoms $r \in R$ and $s \in S$. Beneath r is a subtree containing some number of atoms, beneath s is another subtree containing some number of atoms. If r is a heavy atom, then there are some number of hydrogen atoms bound to r in the subtree beneath r . We have three conceptual entities: atoms, heavy atoms (including their associated hydrogen atoms), and subtrees. We may prune calculations for any combination of entities.

We decide how to prune based on r and s ’s distance. In the following sections we describe the additional data structures we maintain. Briefly, here is a sketch of our pruning options.

1. atom/atom: If the distance between r and s exceeds a threshold, we can assign their interaction energy to zero without doing a more detailed calculation. After this prune, we still must continue calculating interactions between the atoms in r ’s and s ’s subtrees. Rosetta already includes this prune within its `atom_atom_energy()` function. Because we use this function as well, we get this prune for free.
2. atom/subtree: If the distance between r and s is so great that r must be too far to interact with any atom in s ’s subtree, then we can make an atom/subtree prune.
3. subtree/subtree: If the distance between r and s is so great that all atoms in r ’s subtree must be too far to interact with any atom in s ’s subtree, then we can make a subtree/subtree prune. Rosetta makes a similar prune using C_β atoms (see Section 2.2 above).
4. heavy atom/heavy atom: If the distance between heavy atoms r and s exceeds 4.6 Å, we can skip calculating interactions among their bound hydrogens. Rosetta already employs this prune, so we must too, if we hope to improve upon the running time. After this prune, we still must continue calculating interactions between the atoms in r ’s and s ’s subtrees.
5. heavy atom/subtree: Much like the atom/subtree pruning, we may see that a heavy atom r and all of its attached hydrogens are too far to interact with all the atoms in s ’s subtree and then perform this skip.

Heavy Atom/Heavy Atom Pruning As we described above, if a pair of heavy atoms are further apart than $\lambda = 4.6$ Å, then all the interactions between the hydrogen/heavy atom pairs and the hydrogen/hydrogen pairs is zero. We

prune computations based on this cutoff by 1) restricting the atom ordering within the trie and 2) including another global variable in our algorithm, skipH.

We order the atoms in our trie so that the closest ancestral heavy atom for a hydrogen is the heavy atom to which it is chemically bound. For instance, the ordering of atoms for alanine would be: C O N H CA HA CB 1HB 2HB 3HB. (We include backbone atoms as part of the trie. This leaves room for later incorporating backbone flexibility as part of a protein redesign task.) We store each atom’s “heavy atom depth” (hv_depth). The heavy atom depth for a heavy atom is its position in a list of an amino acid’s heavy atoms. Alanine’s CA heavy atom depth is 4. The heavy atom depth for a hydrogen atom is the depth of its parent heavy atom. Alanine’s HA heavy atom depth is also 4.

Our new global variable, skipH, is a stack of booleans, represented as a table. It has MAX_HEAVY rows (the largest number of heavy atoms for a single amino acid, which in tryptophan is 14). Each row has $S.num_heavyatoms$ entries. This table is a stack in that its contents describes properties for ancestor atoms of our currently focused r atom. In essence, the top-of-stack pointer is stored within each atom of R by its heavy atom depth.

Now we’ll describe how we use skipH. If a heavy atom r at heavy atom depth d is at least λ away from heavy atom s of S , then we set skipH[d][s] to ‘true.’ Later, if we want to know if the parent heavy atom for a hydrogen atom of R at heavy atom depth, d , and the heavy atom s of S are greater than λ apart, then skipH[d][s] tells us. To capture this formally, we revise the atom_vs_trie($r, s, ancestral_E$) pre- and postconditions.

Additional Precondition: For all heavy atom ancestors r' of r , skipH[$r'.hv_depth$][s'] holds ‘true’ iff s' is further than λ from r' for all heavy atoms $s' \in S$. Additionally if r is a heavy atom, then for all s'' that precede s , skipH[$r.hv_depth$][s''] holds ‘true’ iff s'' is further than λ from r .

Additional Postcondition: For all heavy atom ancestors r' of r , (including r if r is a heavy atom), skipH[$r'.hv_depth$][s'] holds ‘true’ iff s' is further than λ from r' for all heavy atoms $s' \in S$.

skipH scales in size with the number of heavy atoms in S , unlike some of our other global variables that scale with the number of rotamers in S . It is a rather large data structure. In our example rotamer trie with 10K atoms, roughly 5K would be heavy atoms. In this case, skipH would occupy 70KB.

Subtree/Subtree Pruning In a subtree/subtree prune, we avoid calculating atom/atom energies for all pairs of atoms in the subtrees of r and s . We prune based on a sphere overlap test. The “interaction sphere” of heavy atom r is the sphere centered at r that has a radius of one half of the threshold distance for heavy atom/heavy atom interaction; in our case, one half of 5.5 Å. For two heavy atoms to interact, their interaction spheres must overlap. The “subtree-interaction sphere” of heavy atom r is centered at r , and is large enough that, for any atom s to interact with an atom in r ’s subtree, s ’s interaction sphere must overlap with r ’s subtree-interaction sphere. Equivalently, the radius of the

subtree-interaction sphere is the greatest distance between r and all heavy atoms in r 's subtree + $(5.5 \text{ \AA}/2)$.

When two subtree-interaction spheres do not overlap, we may make a subtree/subtree prune. The non-overlapping condition is met when the squared distance between r and s exceeds the square of the sum of r and s 's subtree-interaction-sphere radii. This comparison is very fast and we can afford to make it at each heavy atom pair we encounter.

When we decide to skip computations involving the subtrees rooted at r and s we immediately add `ancestral_E` to `AREnergies` for all rotamers of S that terminate in the subtree rooted at s . We then skip to the first node of S , that is not in s 's subtree. To make this jump, we must know the sibling of s 's closest ancestor (which may be s itself if s has a sibling).

We also skip over s 's subtree for all of r 's descendants in later calls to `atom_vs_trie()`. We maintain another global array, `trim_depth`, to record which subtrees should be skipped. This array has one entry for each heavy atom of S . The values stored in each entry are small (< 14) so we can get away with using a single byte per entry. In our example rotamer trie with 5K heavy atoms, `trim_depth` occupies only 5KB.

We describe the functionality of this variable with another pre- and postcondition pair for `atom_vs_trie($r, s, \text{ancestral}_E$)`.

Additional Precondition: If r is a heavy atom (hydrogen), then `trim_depth[s]` is less than (less than or equal to) `r.hv_depth` if 1) the subtree-interaction sphere of the heavy atom ancestor of r at depth `trim_depth[s]` (call this ancestor, r') does not overlap with s 's subtree-interaction sphere, and 2) s is the only atom amongst it and its ancestors whose subtree-interaction sphere does not overlap with r' 's subtree-interaction sphere. The value of `trim_depth[s]` is undefined for those atoms of S for which condition 1, but not condition 2, holds. If s 's interaction sphere overlaps with the subtree-interaction spheres for all ancestors of r , then `trim_depth[s]` is greater than or equal to `r.hv_depth` when r is a heavy atom, or strictly greater than `r.hv_depth` when r is a hydrogen atom.

Additional Postcondition: If r is a heavy atom (hydrogen) and `trim_depth[s]` was less than (less than or equal to) `r.hv_depth`, then `trim_depth[s]` remains the same, and the values in `trim_depth` for atoms in the subtree rooted at s are undefined. If r and s are heavy atoms, and r and s 's subtree-interaction spheres do not overlap, then `trim_depth[s]` is `r.hv_depth`. Otherwise, `trim_depth[s]` is `MAX_HEAVY + 1`.

We also make subtree/subtree prunes when we encounter colliding atoms. Collisions reflect physically impossible situations, and an exact representation of a collision's energy is unnecessary. We prune when we find an atom/atom energy that exceeds 20 kcal/mol.

Heavy Atom/Subtree Pruning If r 's interaction sphere and s 's subtree-interaction sphere do not overlap we may skip past s 's subtree. In order to repeat this subtree-skip for the hydrogen atoms attached to r , we maintain another global variable, `skipSubtree`. `skipSubtree`, like `skipH`, is a stack of boolean arrays

represented as a table. Each array has $S.\text{num_heavyatoms}$ entries. There are MAX_HEAVY rows. We do not provide additional pre- and postconditions as they are so similar to those describing skipH. skipSubtree would occupy 70KB in our example 10K atom rotamer trie.

4 Results

We wrote our algorithm in C++ and verified that it generates the same energies as Rosetta’s existing rotamer-pair energy function, `get_energies()`. We compared the running time of our algorithm using six pruning options against `get_energies()` in 57 complete protein redesign tasks (Fig. 2). All six variants included heavy atom/heavy atom pruning. We measured running times on Intel Xeon 2.8 GHz processors each with 2.5 GB RAM. Our algorithm runs 3.87 times faster than `get_energies()`.

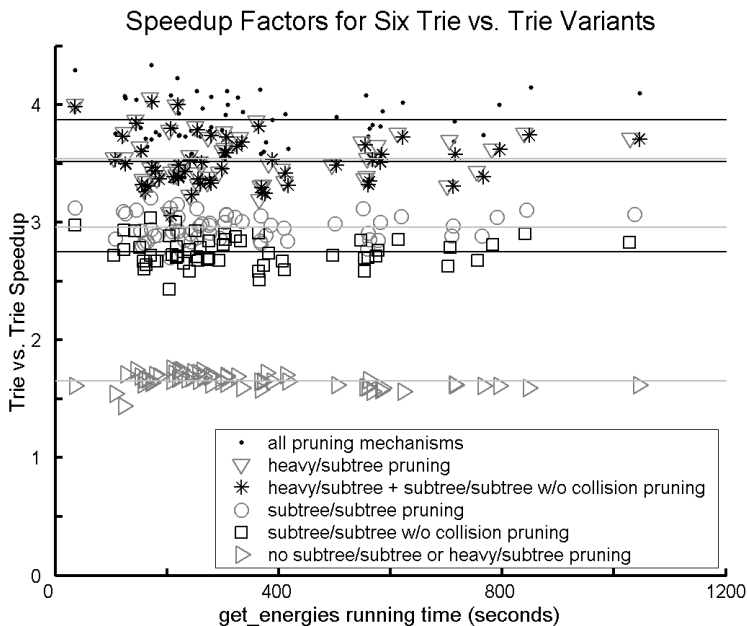


Fig. 2. Comparing `trie_vs_trie()` and `get_energies()` for 57 entire-protein-redesign energy computations. Mean speedup factors for the six pruning combinations were 1.65, 2.75, 2.96, 3.52, 3.54, and 3.87.

5 Discussion

We have sped up the bottleneck stage of Rosetta’s protein design module. There are a few direct consequences of our trie structure we would like to highlight.

Hydroxyl Hydrogens Fine grained sampling of dihedral space for terminal hydroxyl groups now comes at a reduced cost. The shared atomic prefix for two rotamers that differ in their hydroxyl hydrogen placement spans all atoms but the last two: the hydroxyl oxygen and hydrogen. Because the orientation vectors on hydroxyl oxygens point at the hydrogen, each oxygen is distinct. For tyrosine the shared atomic prefix includes 13 side chain atoms.

Uniform Rotamer Libraries Common dihedral angles can improve trie performance by 23%. Currently Rosetta selects its rotamers using Dunbrack’s backbone dependent rotamer library. In this library, very few χ_1 dihedrals agree. The overlap that buys us our performance boost comes from the additional rotamer samples Rosetta takes at χ_2 that surround ($\pm\sigma$) Dunbrack’s rotamers. We measured a 10% decrease in the number of unique atoms in the rotamer trie when we construct our rotamers using a new rotamer library built from rounding Dunbrack’s rotamers to the nearest 10° .

Flexible Backbone Design Imagine sampling a few backbone conformations for a pair of residues [28] and attaching hundreds of rotamers to each sample. This setup for flexible backbone design promotes the backbone from the role of static background into the role of structural variable. With flexible backbone design, side chain/backbone energies must be included in the rotamer-pair energy calculations. We incorporated backbone atoms into our tries so that when we begin using flexible backbone design, we can make effective reuse of side chain/backbone computations.

References

1. Looger, L.L., Dwyer, M.A., Smith, J.J., Hellinga, H.W.: Computational design of receptor and sensor proteins with novel functions. *Nature* **423** (2003) 185–190
2. Dwyer, M., Looger, L., Hellinga, H.: Computational design of a biologically active enzyme. *Science* **304** (2004) 1967–1971
3. Kuhlman, B., O’Niell, J.W., Kim, D.E., Zhang, K.Y., Baker, D.: Accurate computer-based design of a new backbone conformation in the second turn of protein L. *J. Mol. Bio.* **315** (2002) 471–477
4. Kuhlman, B., Dantas, G., Ireton, G., Varani, G., Stoddard, B., Baker, D.: Design of a novel globular protein fold with atomic-level accuracy. *Science* **302** (2003) 1364–1368
5. Berman, H.M., Westbrook, J., Feng, Z., Gilliland, G., Bhat, T., Weissig, H.N.S.I., Bourne, P.: The protein data bank. *Nucleic Acids Research* **28** (2000) 235–242
6. Butterfoss, G., Hermans, J.: Boltzmann-type distribution of side-chain conformation in proteins. *Protein Science* **12** (2003) 2719–2731
7. Ponder, J.W., Richards, F.: Tertiary templates for proteins. Use of packing criteria in the enumeration of allowed sequences for different structural classes. *J. Mol. Bio.* **193** (1987) 775–791
8. R. L. Dunbrack, J., Karplus, M.: Backbone dependant rotamer library for proteins: application to side chain prediction. *J. Mol. Bio.* **230** (1993) 543–574

9. R. L. Dunbrack, J.: Rotamer libraries in the 21st century. *Curr. Opin. Struct. Biol.* **12** (2002) 431–440
10. Lovell, S.C., Word, J.M., Richardson, J.S., Richardson, D.C.: The penultimate rotamer library. *Proteins: Structure Function and Genetics* **40** (2000) 389–408
11. Desmet, J., Maeyer, M.D., Hazes, B., Lasters, I.: The dead-end elimination theorem and its use in protein side-chain positioning. *Nature* **356** (1992) 539–541
12. Goldstein, R.F.: Efficient rotamer elimination applied to protein side-chains and related spin glasses. *Biophysical Journal* **66** (1994) 1335–1340
13. Looger, L.L., Hellinga, H.W.: Generalized dead-end elimination algorithms make large-scale protein side-chain structure prediction tractable: implications for protein design and structural genomics. *J Mol Biol* **307** (2001) 429–45
14. Gordon, D., Mayo, S.: Branch-and-terminate: a combinatorial optimization algorithm for protein design. *Structure Fold Des* **7** (1999) 1089–98
15. Leaver-Fay, A., Kuhlman, B., Snoeyink, J.: An adaptive dynamic programming algorithm for the side chain placement problem. In: *Pacific Symposium on Bio-computing, 2005, The Big Island, HI* (2005) 17–28
16. Pierce, N., Winfree, E.: Protein design is NP-hard. *Protein Engineering* **15** (2002) 779–82
17. Bradley, P., Chivian, D., Meiler, J., Misura, K., Rohl, C., Schief, W., Wedemeyer, W., Schueler-Furman, O., Murphy, P., and C. Strauss, J.S., Baker, D.: Rosetta predictions in CASP5: Successes, failures, and prospects for complete automation. *Proteins: Structure Function and Genetics* **53** (2003) 457–68
18. Dahiyat, B.I., Mayo, S.L.: De novo protein design: fully automated sequence selection. *Science* **278** (1997) 82–87
19. Holm, L., Sander, C.: Fast and simple monte carlo algorithm for side chain optimization in proteins: application to model building by homology. *Proteins* **14** (1992) 213–23
20. Saven, J.G., Wolynes, P.G.: Statistical mechanics of the combinatorial synthesis and analysis of folding macromolecules. *J. Phys. Chem. B* **101** (1997) 8375–8389
21. Desjarlais, J.R., Handel, T.M.: De novo design of the hydrophobic cores of proteins. *Protein Science* **4** (1995) 2006–2018
22. Weiner, P.: Linear pattern matching algorithms. In: *Proc. 14th IEEE Annual Symp. on Switching and Automata Theory.* (1973) 1–11
23. McCreight, E.M.: A space-economical suffix tree construction algorithm. *Jrnl. of Algorithms* **23** (1976) 262–272
24. Ukkonen, E.: On-line construction of suffix trees. *Algorithmica* **14** (1995) 249–260
25. Hellinga, H., Richards, F.: Construction of new ligand binding sites in proteins of known structure. I: Computer-aided modeling of sites with pre-defined geometry. *J. Mol. Bio.* **222** (1991) 763–85
26. Lazaridis, T., Karplus, M.: Effective energy function for proteins in solution. *Proteins: Structure Function and Genetics* **35** (1999) 133–152
27. Kortemme, T., Morozov, A.V., Baker, D.: An orientation-dependent hydrogen bonding potential improves prediction of specificity and structure for proteins and protein-protein complexes. *J. Mol. Bio.* **326** (2003) 1239–1259
28. Noonan, K., O’Brien, D., Snoeyink, J.: Probik: Protein backbone motion by inverse kinematics. In: *WAFR’04, Utrecht/Zeist, The Netherlands* (2004)