Reducing the Computational Load of Energy Evaluations for Protein Folding

Eunice E. Santos

Department of Computer Science

Virginia Polytechnic Institute & State University

santos@cs.vt.edu

Phone: +1 (540) 231 5368

Eugene Santos, Jr.

Department of Computer Science and Engineering

University of Connecticut, Storrs

eugene@cse.uconn.edu

Abstract

Predicting the native conformation using computational protein models requires a large number of energy evaluations even with simplified models such as hydrophobic-hydrophilic (HP) models. Clearly, energy evaluations constitute a significant portion of computational time. We hypothesize that given the structured nature of algorithms that search for candidate conformations such as stochastic methods, energy evaluation computations can be cached and reused, thus saving computational time and effort. In this paper, we present a caching approach and apply it to 2D triangular HP lattice model. We provide theoretical analysis and prediction of the expected savings from caching as applied this model. We conduct experiments using a sophisticated evolutionary algorithm that contains elements of local search, memetic algorithms, diversity replacement, etc. in order to verify our hypothesis and demonstrate a significant level

of savings in computational effort and time that caching can provide.

Keywords: protein folding, triangular lattice, HP energy model, caching, reuse, evolutionary algorithms

1 Introduction

In predicting the native conformation of proteins using various computational protein models, even the simplified models suffer from computational intractability in the worst case. For example, optimizing the simple 2-D square lattice hydrophobic-hydrophilic (HP) [6] has been shown to be NP-Complete [2]. In the past several years, numerous algorithms and techniques have been proposed and explored for quickly determining native conformations based on models such as the HP models. Methods such as genetic and memetic algorithms [8], tabu search [9], and ant colony optimization [11] use approximation and randomized search in an effort to find good solutions in a reasonable amount of time. The fundamental nature of such approaches relies on heuristics and/or randomization to quickly search large numbers of candidate solutions in order to achieve better solutions over time. Hence, the more computational resources (time) that are provided to these methods, the more likely a good solution can be found. Clearly, these methods rely on large numbers of evaluations of the candidate solutions generated. As such, a significant component of the computational effort rests in the evaluations (often called fitness evaluations).

Instead of exploring new algorithms for determining good protein conformations, we take existing algorithms and ask the question: Where can we save on computational effort in order to increase the total number of candidates considered while fixing total time? Our hypothesis is that there is significant redundancy among the large numbers of fitness evaluations of which re-use of computations can drastically reduce computational effort. Intuitively, when we examine the various methods that generate candidate solutions, new solutions are derived from earlier solutions already explored, thus new solutions share attributes with those earlier solutions.

Recently, Santos and Santos [10] proposed a method for caching and re- using partial fitness evaluation results. They applied their approach to 2D and 3D square lattice HP models and formally predicted a 50% savings in fitness evaluations using their caching approach. Empirical tests using a simple genetic algorithm (single point crossover and multipoint mutation) for optimization on 2D and 3D square HP lattice validated the prediction.

In this paper, we further explore the effectiveness of partial fitness evaluation caching. In particular, we consider triangular lattices (also called Honeybee lattices) [3] which are much harder protein conformation optimization problems even in the 2D case. The degrees of freedom (6 directions for 2D) impose more computational requirements given the larger conformational space. Because of the extra difficulty, simple genetic algorithms (GA) and evolutionary approaches have a difficult time solving 2D triangular lattice HP problems. As researchers have empirically demonstrated [8], more sophisticated optimization techniques such as hybrid local/global search, multi- meme GAs, scatter/gather searching, etc. must be employed in a variety of ways to improve diversity of the searching space, faster convergence, and better identification of promising conformations. As it turns out, each of the methods mentioned here require further additional fitness function computations. Thus, the need for caching to reduce the overall computational effort becomes even more critical. We focus on providing predictive performance analysis on the caching approach and validate our predictions for solving 2D triangular lattice HP problems. In order to demonstrate the validity and utility of our analysis and caching approach, we implement an evolutionary algorithm for effectively solving 2D triangular HP lattice that incorporates aspects of memetic algorithms and local search, two-point crossover, random replacement for diversity, and single point mutation with replacement of bad conformations with good conformations, i.e., bad conformations (non-self avoiding walks) resulting from mutation are replaced by a randomly generated good conformation which further promotes diversity. As we can easily see, we have unified elements from various effective optimization algorithms for determining a good conformation. When combined with benchmark

HP problems, this will allows us to study a fairly realistic testbed to demonstrate the feasibility of partial fitness evaluation caching.

2 The 2D Triangular HP Lattice Protein Folding Problem

Currently, a primary concern in biochemistry is the problem of protein native structure prediction. It is commonly assumed that the sequence of amino acids in the protein molecule corresponds to the equilibrium minimum free energy state (the thermodynamic hypothesis) which might help to solve a large number of pharmaceutical and biotechnological problems. Therefore, several models have been presented for the protein folding problem. One of these is the well-known 2D-HP model [6]. The algorithms we presented here are all based on 2D-HP model, that is:

(1) all the type of amino acids are represented by a set A={H,P},

(2) protein instances are represented by a binary sequence,

(3) an energy formula specifying how the conformational energy is computed by

$$E = \sum(e(a, b)),$$

if a=b=H, then e(a,b)=-1, otherwise e(a,b)=0, and

(4) the conformation structure is presented as a self-avoiding walk on a 2D-lattice.

The standard assumption has been that the lattice is square in structure. Under this assumption, it has been proven that protein folding on the two- dimensional HP model is NP-complete [2, 5]. Several methods have been presented to try to solve this problem, such as the chain growth algorithm[4], fast protein folding approximating algorithms [7], and genetic algorithm(s) [12].

However, it has been noted that square or 90-degree angles have serious issues and drawbacks [1], including the particularly serious parity constraint, i.e. any pair of amino acids which are an odd-distance apart from each other can never lie on adjacent square lattice points.

Due to this, triangular lattice structures is a focus of attention [1]. In this paper, we will focus on 2D triangular lattices. These lattices are structures in which any point in the interior of the lattices has 6 neighbors and are typically denoted by their direction (NW, NE, SE, SW, EE - straight east, WW - straight west) from the point in consideration. As such, for any amino sequences of H's and P's, there are a variety of confirmations represented by a string of directions mapped to the appropriate amino acid. The goal is to determine the string of directions that minimizes the energy formula given above.

The computational issues are the same regardless of which lattice structure is utilized. The overhead in time due to repetitive fitness/energy computation provides a significant bottleneck to obtaining a solution in a timely fashion. As stated previously, if one can cache partial results, it may be possible to reduce overall computation time. However, in order for this to be applicable to 2D Triangular HP lattice protein folding, this domain must be shown to be decomposable to the point where partial results are easily substituted in the process of computing overall energy formulae.

To show that this is in fact the case, we first discuss Divide and Conquer approaches, which clearly are subproblem decomposable and discuss how our energy formula can be shown to be a divide and conquer method.

We begin with a brief discussion on divide and conquer design and definition.

3 Divide and Conquer and Application to Fitness Evaluations in Genetic Algorithms

The triangular HP lattice problem has as part of its main components, fitness evaluation using the neighbor energy formula. This evaluation can be viewed as a divide and conquer method in its fitness value computation. Methods of this type are defined by their subproblem/instance decomposition. In cases where same or similar subinstances appear with high regularity, it may be

possible to store or *cache* partial fitness calculations in order to reduce overall computation time.

In [10], the approach of caching partial results was introduced. Divide and conquer is a classic algorithm design paradigm. Below is the skeletal structure of a divide and conquer algorithm:

ALGORITHM 3.1. DC (I, n, O)

/I = current problem instance, n = problem size of I, O = output (solution) */*

if $n < c$ *then*

solve directly

else

Divide I into smaller instances $I_1, I_2, \cdots I_k$

with problem sizes $n_1, n_2, \cdots n_k$, resp.

For $j = 1$ to k do

Call $DC(I_j, n_j, O_j)$

Combine $O_1, O_2, \cdots O_k$ to compute O.

Denote the running time of DC for problem size n by $R_{DC}(n)$. Denote the divide time of DC for problem size n by $D_{DC}(n)$. Denote the combine time of DC for problem size n by $C_{DC}(n)$.

Therefore, if $n < c$ then $R_{DC}(n)$=time to solve directly for size n. Else,

$$R_{DC}(n) = D_{DC}(n) + C_{DC}(n) + \sum_{j=1}^{n} R_{DC}(n_j)$$

Fitness computations that can be decomposed into partial results from subinstances can inherently be solved using divide and conquer methods. Thus, caching partial results from the gene fitness computations can reduce future fitness computation time. In particular, in GAs, much of a gene is preserved through the various operations.

We denote $A_T(k)$ to be the time to access the cache table to determine whether a particular substring of size k resides in the cache, and if so, to access its partial fitness value. We denote

$S_T(k)$ to be the time to store into the cache table a substring of size k. The notation T refers to the cache table.

Let us assume that the fitness function evaluation can be represented by a divide and conquer strategy. By taking into account caching, we modify the divide and conquer scheme for the fitness function evaluation.

ALGORITHM 3.2. F (I, n, O)

if $n < c$ *then*

solve directly

else

if I *is cached then*

output O *directly*

else

Divide I *into smaller instances* $I_1, I_2, \cdots I_k$ *with problem sizes* $n_1, n_2, \cdots n_k$*, resp.*

For $j = 1$ *to* k *do*

if I_j *is not fully cached then*

Call $F(I_j, n_j, O_j)$

else

$O_j \leftarrow access(T, I_j)$

Combine $O_1, O_2, \cdots O_k$ *to compute* O

Analyzing the running time of F, we see that if a partial result is stored in a cache, we can truncate the amount of recursion occurring in a divide and conquer method, thereby reducing overall fitness computational time.

Once each function is fully specified then a closed form for $R_F(n)$ can be derived.

The original (non-caching) run-time is obviously:

$$R_F^{orig}(n) = D_F(n) + C_F(n) + \sum_{j=1}^{k} R_F^{orig}(n_j)$$

If $R_F(n) < R_F^{orig}(n)$ then caching will produce results more efficiently than non-caching.

Precise comparison/results can be done only after the various functions in the equations are fully specified. However, it is quite clear that in general, when the access and storage time are comparable or less than the divide and combine times, caching should be more efficient than non-caching.

4 Triangular HP Lattice: A Cached Divide and Conquer Approach

In order to design a divide and conquer approach, we must first discuss how a fitness value is computed from the storage string of directions.

Once a conformation is laid out on the lattice, then all pairs of non-adjacent amino acids (H's and P's) are considered such that each H-H pair decreases the energy value by 1. This being the case, we can also compute the energy value as each amino acid is considered linearly. By doing so, each non-neighbor pair is computed exactly once in order to obtain the correct final value. Either end of the directional string can be used as the header. Thus energy pairs can be computed going either *left to right* or *right to left* in the directional string.

Regardless of the lattice structure used, the fitness value computation can be presented as a divide and conquer method due to the linearity of computation. In other words, the problem decomposes into one subinstance with a decrease in size of 1. The divide time is negligible and the combined time relies on merger of the subinstance result with the main result. Clearly, caching the partial results can severely truncate the amount of recursion that has to be performed in fitness

value computation and can thus be a major computational time saver.

While conceptually, this is the case, this hinges on the fact that it is important to determine how best to denote lattice points in such a way that neighbor lattice points can be quickly identified. We will do this by mapping a 2-D triangular lattice into a 2-D square lattice. We perform our construction on a lattice with no boundary conditions (i.e. a wrap-around lattice) however it is easy to see how it can be adapted to lattices with boundary conditions.

First, we note that the size of a 2-D triangular lattice denoted by $n \times n$ will refer to the fact that no walk can traverse more than n as a max number of steps either *north-south* or *east-west*. For example, a walk consisting of just two neighbor points with one point NW in direction from the other would be 1 step north and 1 step west, while a neighbor point WW would denote 0 steps north and 1 steps west.

Our mapping works as follows:

Given a triangular $n \times n$ lattice $\mathcal{L}_{\mathcal{T}}$, we construct a square lattice $\mathcal{L}_{\mathcal{S}}$ of size $2n \times n$ such that the set of lattice points in $\mathcal{L}_{\mathcal{T}}$ are mapped to $\mathcal{L}_{\mathcal{S}}$ such that given a point p in $\mathcal{L}_{\mathcal{T}}$, if it is mapped to point (x, y) in $\mathcal{L}_{\mathcal{S}}$ then the neighbors of p are mapped to:

- *North-West(NW) Neighbor*: $(x-1, y+1)$
- *North-East(NE) Neighbor*: $(x+1, y+1)$
- *Straight-West(WW) Neighbor*: $(x-2, y)$
- *Straight-East(EE) Neighbor*: $(x+2, y)$
- *South-West(SW) Neighbor*: $(x-1, y-1)$
- *South-East(SE) Neighbor*: $(x+1, y-1)$

Clearly, it will take $O(n^2)$ time to construct and initialize the lattice. However, we need not re-initialize the whole lattice every time we begin a fitness evaluation. In fact, instead of a re-occurring $O(n^2)$ initialization time, we incur a $O(n^2)$ once and then for every time we lay out lattice points, we erase only those points laid on the grid (i.e. n points) allowing us to incur a repetitive $O(n)$ time

rather than the quadratic function. Now, we can compute the amount of time a divide and conquer approach to fitness value computation would take. However, in order to determine the effect of caching in running time, we must first discuss an appropriate caching policy, which is discussed in the next subsection.

4.1 A Caching Policy

We now describe a caching policy that can be appropriately used for the Triangular 2D-HP problem. This is a modification of the one for square lattices in [10]. However, since for this paper, implementation utilizes 2-point crossover and the amount of potential neighbors increases to 6, new insights were gained.

Our approach is to use a tree structure to maintain our necessary gene caching. Given that the length of our genes is n, our tree will be of height n where level i in the tree will correspond to the ith index of the gene. We call this tree the *left-cache* since the root of the tree corresponds to the leftmost entry in each gene. Each node in the tree has either n children ordered left-to-right from 1 to n or is a leaf. Also, each node has a key corresponding to the partial value computed for the substring formed from indices 1 to h of the gene where h is the level of the node starting at 1. The *right-cache* is similarly constructed.

The primary properties of the left/right-cache are:

- Size of cache is linear with respect to number of genes stored.
- No collisions ever occur in the cache.
- Worst-case access and storage are $O(n)$ for genes as well as any prefix or suffix of these genes.

For this caching policy: $A_T(n) = 5n$ and $S_T(n) = 9n$. In our analysis, we assume that accessing an item from a memory cell (whether in the cache, directional string, etc.) requires 1 time unit. Creating or storing an item to a memory cell is double that time.

4.2 Theoretical Analysis

To recap the design discussion in this section, we can design a divide and conquer approach to energy computations for Triangular 2D HP protein folding. In this scheme, there was one subinstance decomposition of one size smaller, a negligible divide time, and a simple combine procedure which merges the subinstance result with the neighbor H-H pairs of the current directional string item. Moreover, the protein conformation on the triangular lattice can be mapped and layout out on a square grid for easier neighbor determination. Furthermore, we laid out a caching scheme for partial results that saves linearly a left and right substring (prefix or suffix) of each gene. Using such a caching scheme, we can develop a divide and conquer algorithm that relies on the potential of truncated recursion by cache lookup.

Performing a theoretical analysis, the fitness computation time for one conformation for the Triangular 2D Lattice Protein Folding problem (TPF) for non-caching has an expected run time of

$$R_{TPF}^{orig}(n) = 78n.$$

If caching is used, the fitness evaluation time will be determined by how large a part of the amino acid string to be computed is found to be cached. If a substring of length z is cached, then the expected run time is

$$R_{TPF}(n, z) = 18n + 69(n - z) + 5z.$$

Comparing caching to non-caching, if z is quite small then the overhead needed to write to and access the cache will force caching to take more time. Therefore, caching is worthwhile only when z is sufficiently large.

Clearly, while fitness computation is an important part of GA run-time, it is just one of many

components. Below is the pseudocode for one iteration of our evolutionary algorithm, i.e., the construction of the new generation of genes from the previous generation, consists of the following sequence:

ALGORITHM 4.1. *Iteration(G_{old}, G_{new}, k, n, c_{rate}, m_{rate}, d_{rate})*

/* G_{old} = array of individuals in the current generation

G_{new} = output of new array of individuals

k = length of array G_{old}

n = problem size (length of gene)

c_{rate} = rate of 2-point crossover

m_{rate}= rate of 1-point mutation

d_{rate}= rate of diversity replacement */

/* Selection */

Construct G_{new} from G_{old} with roulette wheel selection

/* Crossover */

for i = 1 to k do

  if (Rnd() $< c_{rate}$ then

    Mark ith individual as crossable

For all consecutive pairs (I_1, I_2) of crossable individuals in G_{new} do

  Do 2-point crossover on I_1 and I_2 and replace in G_{new}

  Compute fitness for I_1 and I_2

  Mark I_1 and I_2 as modified

/* Local Search */

For each modified I in G_{new} do

  for i = 1 to n do

    for j = EE, WW, NE, NW, SE, SW} do

Construct I_j from I by replace ith element in I with j

Compute fitness for I_j

Replace I with I_j that has best fitness

/* Mutation */

for i = 1 to k do

if (Rnd() $< m_{rate}$ then do

Do 1-point mutation on I_i in G_{new} and replace

Compute fitness of I_i

if I_i is invalid walk then

replace with new randomly generate individual

compute fitness of I_i

/* Diversity Replacement */

Randomly replace up to d_{rate} individuals in G_{new} with new randomly generated individuals

Compute fitness for new individual

Analyzing each component of the algorithm, we see that for one generation of our GA, the expected running time is:

- $\frac{3k^2}{2} + 4k$ is the time for selection
- $k(2 + c_{rate}(8 + 82n))$ is the time for crossover using non-caching fitness evaluation, and $k(2 + c_{rate}(4n + 8 + R_{TPF}(n, \frac{2n}{3})) = k(4 + c_{rate}(8 + 44.33n))$ is the expected time for crossover using caching for fitness evaluation
- $c_{rate}k(36n + 468n^2 + 4)$ is the time for local search using non- caching fitness evaluation, and $c_{rate}k(6n(6 + R_{TPF}(n, \frac{3n}{4})) + 4) = c_{rate}k(36n + 234n^2 + 4)$ is the expected time for local search using caching in fitness evaluation
- $k(2 + m_{rate}(12 + 78n))$ is the time for mutation using non-caching fitness evaluation, and

$k(2 + m_{rate}(12 + R_{TPF}(n, \frac{3n}{4})) = k(2 + m_{rate}(12 + 39n))$ is the expected time for mutation using caching fitness evaluation

- $k(4 + d_{rate}(78n))$ is the time for diversity replacement using non- caching, and this is also the worst case possibility if using caching.

The theoretical run-time for one generation using non-caching is:

$$T(k, n) = \frac{3}{2}k^2 + 14k + c_{rate}k(12 + 118n + 468n^2) + m_{rate}k(12 + 78n) + d_{rate}78kn.$$

The expected running time with caching is:

$$T_c(k, n) = \frac{3}{2}k^2 + 14k + c_{rate}k(12 + 80.33n + 234n^2) + m_{rate}k(12 + 39n) + d_{rate}74kn.$$

Comparing the ratio of $T_c(k, n)/T(k, n)$, we see that the dominating factor is $c_{rate}234kn^2/c_{rate}468n^2 = 50\%$. Hence we expect a 50% savings in computational time.

5 Experimental Results

Given our theoretical predictions, we now compare them against actual runs. We implemented our evolutionary algorithm according to Algorithm 4.1 with two versions: (1) without caching of fitness computations, and (2) with left and right caching. Our goal is to compute the computational effort saved through caching.

We implemented both versions in C++ on a Pentium IV 2GHz machine with 1G RAM running Linux. The testbed proteins we employ are drawn from [8] for triangular 2D-HP and from [10] which have yet to be computed in triangular 2D-HP. Table 5.1 contains the HPs used in our experiments. We note again that our goal in this experiment is to determine savings from cache re-use. Hence, we limited our number of generations applied to our evolutionary algorithms to 100. Clearly, as can be seen from Table 5.1, 100 generations is sufficient for short proteins but not for

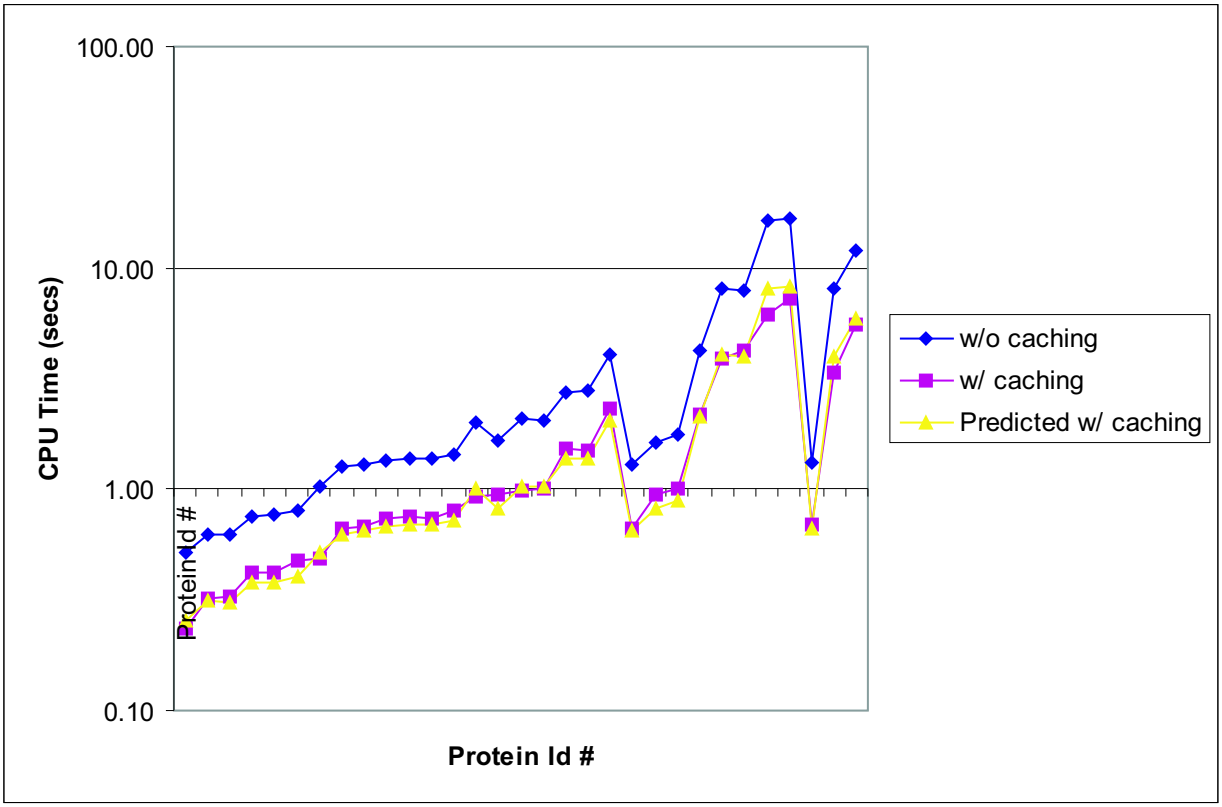


FIG. 5.1. Average CPU Time (in seconds) of w/o caching vs w/caching over 10 runs per protein at 100 generations. Predicted CPU Time also plotted.

longer proteins. However, this will allow us to demonstrate the utility of caching by allowing us to extend the number of generations that can be computed using the same amount of time needed without caching. Our theoretical prediction results in a 50% savings. As such, for our caching runs, we allow the algorithm to continue for another 100 generations.

To determine reasonable parameters for our evolutionary algorithm, we conducted some pre-trial runs. For our experiment, we used a crossover rate of 0.7, a mutation rate of 0.05, and a diversity replace rate of 10%. For our evolutionary algorithms with caching, we gathered results after 100 generations and then again after 200 generations. Each protein was run 10 times with each evolutionary algorithm.

We computed the average CPU time (in seconds) over 10 runs per protein on each evolutionary

#	HP Sequence	Length	Optimal Soln	100 Generations Best Soln
1	HHPHPHPHPHPH	12	-11	-11
2	HHPPHPHPHPHPHP	14	-11	-11
3	HHPPHPPHPHPHPH	14	-11	-11
4	HHPHPPHPPHPPHPPH	16	-11	-11
5	HHPPHPPHPHPHPPHP	16	-11	-11
6	HHPPHPPHPPHPPHPPH	17	-11	-11
7	HHPHPHPHPHPHPHPHH	17	-17	-17
8	HHPPHPPHPHPHPPHPHPHH	20	-17	-16
9	HHPHPHPHPHPPHPPHPPHH	20	-17	-15
10	HHPPHPPHPHPPHPHPPHPHH	21	-17	-15
11	HHPHPPHPPHPHPHPPHPPHH	21	-17	-15
12	HHPPHPHPHPPHPHPPHPPHH	21	-17	-16
13	HHPPHPPHPHPHPPHPPHPPHH	22	-17	-16
14	HHHPHPHPHPHPHPHPHPHPHHH	23	-25	-24
15	HHPPHPPHPPHPPHPPHPPHPPHH	24	-17	-15
16	HHHPHPHPPHPHPHPHPHPHPHHH	24	-25	-21
17	HHHPHPHPHPPHPHPHPHPHPHHH	24	-25	-24
18	HHHPPHPPHPPHPPHPHPPHPHPPHPPHHH	30	-25	-18
19	HHHPPHPPHPPHPHPPHPHPPHPPHPPHHH	30	-25	-20
20	HHHPPHPPHPPHPHPHPPHPPHPPHPPPPPHPHPHHH	37	-29	-21
21	HPHPPHHPHPPHPHHPPHPH	20	n/a	-15
22	HHPPHPPHPPHPPHPPHPPHPPHH	24	n/a	-14
23	PPHPPHHPPPPHHPPPPHHPPPPHH	25	n/a	-16
24	PPPHHPPHHPPPPPHHHHHHHPPHHPPPPHHPPHPP	36	n/a	-23
25	PPHPPHHPPHHPPPPPHHHHHHHHHHPPPPPPHHPPHHP PHPPHHHHH	48	n/a	-37
26	HHPHPHPHPHHHHPHPPPHPPPHPPPPHPPPHPPPHPHH HHPHPHPHPHH	50	n/a	-32
27	PPHHHPHHHHHHHHPPPHHHHHHHHHHPHPPPHHHHHH HHHHHPPPPPHHHHHHPHHPHP	60	n/a	-62
28	HHHHHHHHHHHHPHPHPPHHPPHHPPHPPHHPPHHPPH PPHHPPHHPPHPHPHHHHHHHHHHHH	64	n/a	-54
29	HHHPPHPHPHPPHPHPHPPH	20	n/a	-17
30	PHHHPHHHPPPHPHHPHHPPHPHHHHPHPPHHHHHPHP HHPPHHP	45	n/a	-36
31	HPHHHPHHHPPHHPHPHHPHHHPHPHPHHPPHHHPPHP HPPPPHPPHPPHHPPHPPH	57	n/a	-38

TABLE 5.1. HP Testbed Proteins. Protein ID 1-20 are from [8]. Protein ID21-31 are from [10].

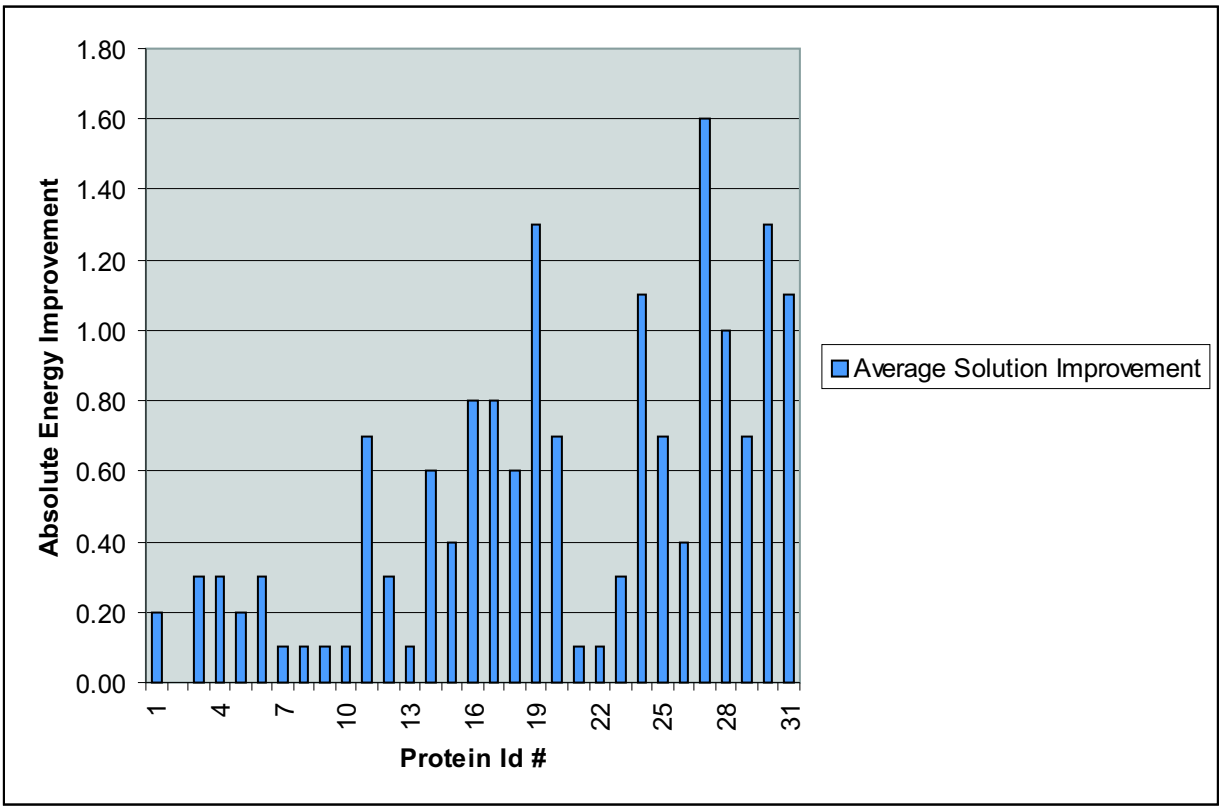
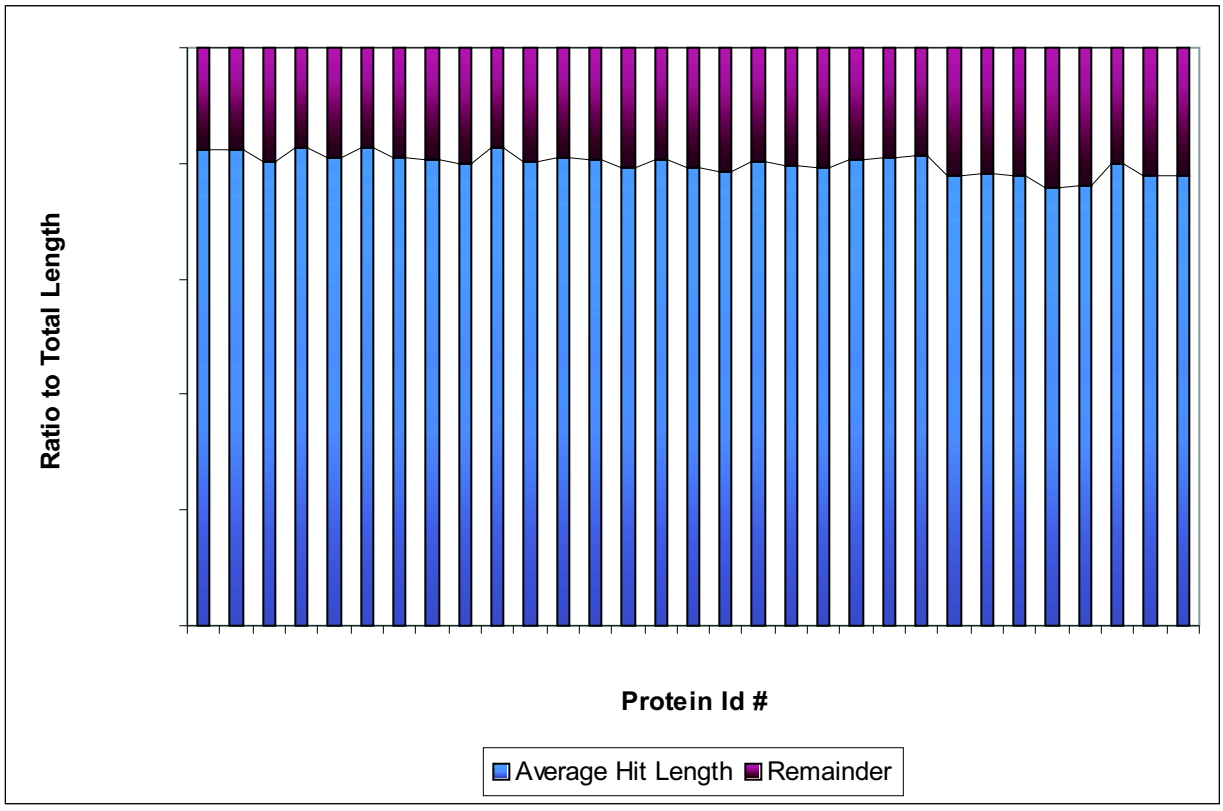


FIG. 5.2. Average absolute energy value improvement.

algorithm. Figure 5.1 shows the comparison of times with the predicted time in a logarithmic plot. Our predicted runtimes are very close to our actual runtimes. For predicted runtime, we simply factored our predicted savings with the actual runtime w/o caching.

Next, we allowed our algorithm with caching to 200 generations. In 9 out of the 31 proteins, the algorithm was able to find a better solution out of its 10 runs. Furthermore, the average solution found were better when given the additional time (see Figure 5.2). We note that this is especially significant since our energy values are simply negative integers.

Finally, we examine that average hit rate in our caching technique in Figure 5.3. In general, we find that we are achieving on average an 80% hit rate which translates to reusing nearly 80% of fitness computations. One primary factor that resulted in such a high rate is the local search that has been employed.



}

FIG. 5.3. Ratio of cache hit to total length.

6 Conclusion

Determining the native conformation using computational protein models requires a large number of energy evaluations especially with stochastic search algorithms that rely on diversity of the search in order to find a good solution. Such evaluations clearly consume a significant amount of available computational resources. In this paper, we have examined a caching approach that exploits divide and conquer to re-use past energy evaluations for partially recomputing the quality of new candidate solutions. We have provided theoretical analysis and prediction on the savings that can be gained through our caching approach. We then compared our theoretical analysis against a real-world testbed of 2D triangular lattice HP proteins using a sophisticated evolutionary algorithm that consists of local search, memetic elements, and diversity factors. Our comparisons demonstrated the promising savings through our caching approach and matches our predicted analysis. Furthermore, we applied the savings gained from caching towards more search which resulted in better solutions.

In conclusion, caching is a promising approach for better utilizing computational time and resource. Given that the search for candidate solutions are relatively structured by nature, caching should have a tremendous impact in a variety of domains. One future direction would be to examine more general decomposition approaches towards caching in order to handle even more specific energy models.

References

[1] R. Agarwala, S. Batzoglou, V. Dancik, S. Decatur, M. Farach, S. Hannenhali, S. Muthukrishnan, and S. Skiena. Local rules for protein folding on a triangular lattice and generalized hydrophobicity in the HP model. *Journal of Computational Biology*, 4(2):275-296, 1997.

[2] B. Berger, T. Leighton. Protein folding in the hydrophobic-hydrophilic (HP) model is NP-

complete. *Journal of Computational Biology*, 5(1), 27-40, 1998.

[3] B.P. Blackburne and J.D. Hirst. Evolution of Functional model Proteins. *Journal of Chemical Physics*, 115(4), 1934-1942, 2001.

[4] E. Bornberg-Bauer. Simple folding model for hp lattice proteins. In *Proceedings of Bioinformatics German Conference on Bioinformatics GCB '96*, 125-36, 1997.

[5] P. Crescenzi, D. Goldman, C. Piccolboni, M. Yannakanis. On the complexity of protein folding. *Journal of Computational Biology* 5(3):423-465, 1998.

[6] K.A. Dill, S. Bomberg, K. Yue, K.M. Fiebig, D.P. Yee, P.D. Thomas, H.S. Chan. Principles of protein folding: A perspective from simple exact models. *Protein Sci 4*, 561-602. 1995.

[7] Hart, W. E., and Istrail, S. Fast protein folding in the hydrophobic-bydrophilic model within three-eighths of optimal. In *Proceedings of Twenty-seventh Annual ACM Symposium on Theory of Computing(STOC95)*, 157-68, 1995.

[8] N. Krasnogor, B. Blackburnem, J.D. Hirst, E.K. Burke. Multimeme Algorithms for Protein Structure Prediction in *Proceedings of Parallel Problem Solving From Nature*, 2002, Lecture Notes in Computer Science.

[9] M. Milostan, P. Lukasiak, K. Dill, J. Blazewicz. A tabu search strategy for finding low energy structures of proteins in HP-model. *RECOMB Poster Proceedings*, 2003.

[10] E.E. Santos, E Santos Jr. Effective and Efficient Caching in Genetic Algorithms. *International Journal on Artificial Intelligence Tools* 10(1-2): 273-301, 2001.

[11] A. Shmygelska, R. Aguirre-Hernndez, H.H. Hoos. An Ant Colony Optimization Algorithm for the 2D HP Protein Folding Problem. *Proceedings of the Third International Workshop, ANTS 2002, Proceedings*, Springer's Lecture Notes in Computer Science (LNCS) series, Vol. 2463.

[12] R. Unger, and J. Moult. Genetic algorithms for protein folding simulations. *Journal of Molecule Biology* 231:75-81, 1993.