Cache Diversity in Genetic Algorithm Design

Eunice E. Santos  
Department of Electrical Engineering  
and Computer Science  
Lehigh University  
Bethlehem, PA 18015  
santos@eeecs.lehigh.edu

Eugene Santos, Jr.  
Department of Computer Science  
and Engineering  
University of Connecticut  
Storrs, CT 06269  
eugene@cse.uconn.edu

Abstract
Fitness function computations are a bottleneck in genetic algorithms (GAs). Caching of partial results from these computations can reduce this bottleneck. We provide a rigorous analysis of the run-times of GAs with and without caching. By representing fitness functions as classic Divide and Conquer algorithms, we provide a formal model to predict the efficiency of caching GAs vs. non-caching GAs. Finally, we explore the domain of protein folding with GAs and demonstrate that caching can significantly reduce expected run-times.

Introduction
In genetic algorithms, the computation of the fitness function provides the largest computational load for the algorithms. Each population generation is composed of individuals who are formed from previous generations via cloning, crossover, or mutation.

Therefore, it is quite clear that the fitness functions of these individuals are based in part on the fitness calculations of their ancestors. That being the case, storing either full fitness values or, potentially more rewarding, storing partial results of fitness computations from previous generations, can be beneficial. In other words, when would caching results be beneficial? However, storing, accessing and determining the existence of partial fitness computations are not a straightforward task. When should we store (cache)? What partial computation should we store? When is it worth accessing the cache to determine whether a partial result exists? Or, at a more general level, which fitness functions should utilize caches and how do we ensure a diversity of cached results? While there has been some work on exploring the ideas of caching partial results (Langdon 1998), these have only concentrated on empirical analyses. To the best of our knowledge, we are the first paper to provide concrete theoretical analyses on caching and cache diversity of fitness function computation. In fact, for fitness functions which can be represented as classic Divide and Conquer algorithms, efficiency of the genetic algorithm under any number of conditions can be significantly improved.

This paper is divided into the following sections. First, we provide a brief overview of genetic algorithms and the general idea of caching. Next, we briefly present an overview on the divide and conquer paradigm central to our analyses. With this background, we present our theoretical analysis and provide a formal model of the effectiveness of caching and apply it to protein folding.

Overview GA
The class of algorithms based on simple Genetic Algorithms (GA) (Michalewicz 1992) is a randomized approach to combinatorial optimization. Optimization is achieved when genetic algorithms take a small sample from the space of possible solutions (called the population) and use it to generate other (possibly better) solutions. The method of generating new solutions is modeled after natural genetic evolution.

Each population is subjected to three basic operations (selection, crossover and mutation) during the course of one generation; the results of the operations determine the composition of the population for the next generation. The three operations are probabilistic in nature; this allows the GA to explore more of the search space than a deterministic algorithm.

The two issues that must be addressed when mapping a problem domain into a problem that is solvable by GAs are:
- How to represent a solution to the problem as a gene containing a set of chromosomes that can be genetically manipulated
- How to evaluate the fitness of a solution

The genetic operations manipulate each gene by changing the values of the chromosomes.¹

¹The authors have been actively pursuing GAs for a number of domains including bayesian reasoning (Zhong & Santos 1999; Santos & Shimony 1998), protein folding (Santos, Lu, & Santos 2000), and scheduling.
The selection operation is the standard "roulette wheel" selection approach, based on the ranking of the individuals within the population instead of the absolute performance value. With the wide range of performance values typical to this problem domain, a strictly performance-based selection disproportionately favors the highest probability solution; this causes premature convergence of the population onto a local optimum. The crossover operation performs a two-point crossover: two selected genes are broken in two randomly selected places and the middle sections are exchanged to form the new members of the population. Mutation randomly selects a chromosome to modify and then randomly chooses a new value for that chromosome.

The result of this genetic manipulation is that the population tends to converge towards a local optimum in the solution landscape; the convergence is exhibited by the population containing a large number of the same solution. If the mutation operation is disabled, this convergence typically occurs quite rapidly; unfortunately it is not possible to determine if this local optimum is actually the global optimum. The mutation operator helps the GA find other (better) local optimum by forcing some members of the population to lie outside of the current local optimum.

The crossover operator moves the population in small steps "uphill" (towards the closest local optimum); the steeper the slope, the faster the population as a whole converges. If the landscape is level, the crossover and selection operators have no direction in which to move the population, and convergence does not occur.

**Divide and Conquer**

One of the classic paradigms in algorithm design and analysis is divide and conquer. The concept is elegant in its simplicity. In essence, a problem is solved by designing an algorithm that is based on dividing the problem into smaller instances of the problem and then combining the results of the instances in order to obtain the solution for the original problem. Below is the skeletal structure of a divide and conquer algorithm:

```plaintext
ALGORITHM 0.1. 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 I1, I2,...Ik
 with problem sizes n1, n2,...nk, resp.
 For j = 1 to k do
 Call DC(Ij, nj, Oj)
Combine O1, O2, ...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)
```

**Cache Diversity and Storage**

Our goal is to cache partial results from the gene fitness computations in order to reduce future fitness computation time. In particular, we observe that in GAs, much of a gene is preserved through the various operations. Cloning, of course, is the ideal case where no additional computations are required.

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.

**Assumptions and Results**

We assume that the fitness function evaluation can be represented by a divide and conquer strategy. Therefore, obvious partial fitness computations to store include smaller problem instance results.

For the simple GA, we assume mutation and crossover occur at only one point. By taking into account caching, we modify the divide and conquer scheme for the fitness function evaluation. The modification is presented below:

```plaintext
ALGORITHM 0.2. F (I, n, O)
if n < c then
 solve directly
else
 if I is a clone then
 output O directly
 else
 Divide I into smaller instances I1, I2,...Ik
 with problem sizes n1, n2,...nk, resp.
 if I is a mutation then
 x = point of mutation
 For j = 1 to k do
 if Ij contains x then
 Call F(Ij, nj, Oj)
 else
 Oj = access(T, Ij)
```
else

y and y + 1 = crossover points (i.e. crossover occurs between y and y + 1)

For k = 1 to m do

if Ij contains only points from

1. y or (y + 1)..n then

Call F(Ij, n_j, O_j)

else

O_j = access(T, I_j)

Combine O_1, O_2, ... O_k to compute O

Note that this algorithm ensures cache hits at all times. Furthermore, each cache store operation is performed only once for each fitness computation. Analyzing the running time of F, we see that:

- if n < c then the time required is the time to solve the instance directly. If n ≥ c then the following cases below are utilized.
- if I is a clone,

RF(n) = c_t

where c_t is a constant representing the time to determine the type of operation.
- if I is a mutation,

RF(n) = DF(n) + CF(n) + c_t + \sum_{j=1}^{k} (c_a + G_{I_j}(n_j))

where x is the mutation point, c_a is a constant representing the time needed to determine whether I_j contains x, and

G_{I_j}(n_j) = \begin{cases} A_T(n_j) & \text{if I_j does not contain x} \\
R_F(n_j) & \text{otherwise} \end{cases}

- if I is a crossover,

RF(n) = DF(n) + CF(n) + c_t + \sum_{j=1}^{k} (c_b + H_{I_j}(n_j))

where y is the crossover point, c_b is a constant representing the time needed to determine whether I_j values only from 1 to y or only from y + 1 to n and

H_{I_j}(n_j) = \begin{cases} A_T(n_j) & \text{if I_j contains values from 1 to y or from y + 1 to n} \\
R_F(n_j) & \text{otherwise} \end{cases}

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

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

RF^{rig}(n) = DF(n) + CF(n) + \sum_{j=1}^{k} R_F^{rig}(n_j)

If RF(n) < RF^{rig}(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.

Analysis Example

We now take our analysis and apply it to the domain of protein folding. 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 (Lau & Dill 1989). The algorithms we presented here are all based on 2D-HP model, that is:

- all the type of amino acids are represented by a set A={H,P},
- protein instances are represented by a binary sequence,
- 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
- the conformation structure is presented as a self-avoiding walk on a 2D-lattice.

It has been proven that protein folding on the two-dimensional HP model is NP-complete (Crescenzi et al. 1998). Several methods have been presented to try to solve this problem, such as the chain growth algorithm (Bornberg-Bauer 1997), fast protein folding approximating algorithms (Hart & Istrail 1995), and genetic algorithm(s) (Unger & Moult 1993).

A Caching Policy

We now describe a caching policy that can be appropriately used for the 2D-HP problem. Given the importance of partial results for the divide and conquer fitness computation, a traditional hash-table approach is not appropriate for our simple GA. For example, in one point crossover, if the crossover occurs at index i, there is no need to recompute the partial fitness of either the left or right portions of the new gene since these computations have already been made for the originating parent genes. Hence, it also becomes important to store the partially computed values. Furthermore, since crossover can occur at any point, we
would wish to retrieve substrings of the full gene as well.

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 \( i \)th 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. A left-cache example is shown in Figure 0.1.

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: \( A_T(n) = 4n \) and \( S_T(n) = 6n \).

**Analysis**

We can formulate the fitness computation for the 2D-HP model as a divide and conquer task on a grid which can be achieved in linear time with careful design. The gene can be laid out on this grid in a divide and conquer fashion such that the partial fitness computations are achieved by computing a left substring (prefix) of each gene and combined with the remaining right substring (suffix). In other words, the divide and conquer algorithm relies on only one subinstance of size \( n_1 \geq n/2 \). For the protein-folding problem \( PF \) for non-caching, the expected run time is

\[
R_{PF}^{\text{orig}}(n) = 104n.
\]

Analyzing the caching algorithm, we see that:

- if \( I \) is a clone, \( R_{PF}(n) = 1 \)
- if \( I \) is a mutation, \( R_{PF}(n) = 4 + 60n \)
- if \( I \) is a crossover, \( R_{PF}(n) = 4 + 60n \)

The average time for caching is at most \( 4 + 60n \). Dividing the two results, we get

\[
\frac{4 + 60n}{104n} \approx \frac{60}{104} = 58\%.
\]

improvement. As we can see, even for such a simple fitness function, we can get significant improvement, more than doubling the number of computations over the same amount of time.

**Conclusion**

We have provided a rigorous analysis of the benefits of caching in genetic algorithms to reduce the time necessary for fitness function computations. A cache hit with at worst linear overhead eliminates the cost of a fitness computation clearly resulting in significant savings when the fitness computation time is a high-degree polynomial. We demonstrated that even if the fitness functions are linear in nature with regards to their computations, caching can still have a significant impact. In particular, we studied the 2D-HP Lattice model for protein folding where caching can potentially reduce the time for an individual fitness calculation by nearly half. This directly translates to double in the number of generations that can now be explored in the same allotted amount of time for GAs without caching. We believe that as long as a fitness function can be reformulated in terms of divide and conquer, caching will always improve efficiency. Future work we intend to pursue would be to consider general dynamic programming decompositions of fitness functions as well as classes of caching policies.

**Acknowledgments**

Eunice E. Santos was supported by an NSF CAREER Grant.

Eugene Santos, Jr. was supported by AFOSR Grant F49620-99-1-0244.

**References**


FIG. 0.1. Caching Example. Genes are of length 4. Indices 1 through 4 have range values \( \{0, 1, \ldots, 7\} \), \( \{a, b, c, d\} \), \( \{+,-,*\} \), and \( \{t, f\} \) respectively. Assuming we have cached the following 4 genes: \((6, d, -, f)\), \((1, a, *, t)\), \((6, d, *, t)\), and \((1, a, +, t)\). Each cell consists of a partial fitness value and a pointer. An x indicates NULL or no value.

