Modified Gradient Techniques for Normalized Solution Vectors

Kelly D. Crawford
ARCO
2300 West Plano Parkway
Plano, TX, 75075 (USA)
kcrawford@acm.org

Michael D. McCormack
Optimization Associates, Inc.
1437 Debon Drive
Plano, TX 75075 (USA)
m.d.mccormack@worldnet.att.net

Donald J. MacAllister
ARCO
2300 West Plano Parkway
Plano, TX 75075 (USA)
dmacall@mail.arco.com

Abstract

We define a technique that is found to be useful when dealing with solution vectors (i.e., chromosomes) that must be normalized before evaluation. Normalized vectors can be either trivially manipulated or trivially interpreted, but not both. This paper studies some modified gradient techniques for both bit and floating point representations that have been successfully used at ARCO on some real world problems where other optimization techniques have failed.

Before Normalization

.5	.8	.2	.3	.4	.3	.9
Group 1			Group 2			
.33	.53	.14	.16	.21	.16	.47

After Normalization

Figure 1: Normalized Vector Example

1 THE PROBLEM

Many problems are formulated as the maximization (or minimization) of an objective function, f, over the set of all possible n-dimensional real solution vectors, x (i.e., $f : x \rightarrow \Re^n$). If f is differentiable (continuous), there are numerous techniques available that can make use of the objective function's derivative, f' (e.g., Hagar, 1988). The derivative is also referred to as the slope, or gradient. Unfortunately, many real world problems do not have the nice properties of being continuous and differentiable.

For our problem, we define a solution vector of length n, $x_i \in [0...1]$, $i = 0...(n-1)$, broken into m subgroups of varying cardinality. Let g_0 be the index of the start of the first group, and so on, with g_{m-1} being the starting index of the last group, and $g_m = n$. Prior to evaluation using $f(x)$, each subgroup must be normalized. Normalizing a subgroup is simply accomplished by summing the members of that group together, and then dividing each member by that sum. The result is that each subgroup now sums to 1.0.

For example, Figure 1 shows a solution vector of size $n = 7$ with $m = 2$ subgroups. The first group contains 3 values and the second contains 4, with $g_0 = 0$, $g_1 = 3$ and $g_2 = 7$.

There are at least two possible ways to handle normalized solution vectors. Using the representation described above allows the vectors to be trivially manipulated using standard crossover and mutation techniques. However, a normalization step is required prior to evaluation, which creates somewhat of a disconnect between the genotype (the actual bit or floating-point representation) and the phenotype (the normalized vector). A change to a single genotype value in group j will actually change all of the phenotype values in group j.

Referring again to Figure 1, consider changing the .2 value in Group 1 to .3. All Group 1 values after normalization will change $(.3125, .5, .1875)$. While the shape of the group remains the same, it is clearly a disruptive operation. Changing multiple values on a single crossover or mutation is even more disruptive.

A second technique would be to actually preserve the normalization within the crossover and mutation operators. In other words, the initial solution vector would be normalized, and each crossover or mutation step would produce a properly normalized solution vector.

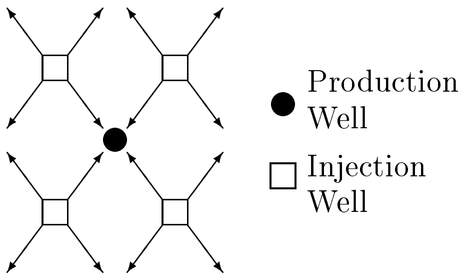

Figure 2: Material Balance Example

In addition to requiring highly specialized operators, this method leaves one uncertain about what sort of building blocks would be preserved during recombination, and just how disruptive a mutation might be.

2 MATERIAL BALANCE

We encountered this sort of representational difficulty at ARCO while working on the Material Balance problem (McCormack, 1999). The details of this problem are beyond the scope of this paper, but we will give a brief description here.

Material Balance is a tool employed by reservoir engineers in developing and efficiently producing the hydrocarbon reserves of an oil field. In this process, engineers determine the fluid saturations and pressure changes that occur in the reservoir as a result of injecting and producing fluids, and compare these results with actual measurements. The oil field is broken up into some number of "patterns", each containing producing and injecting wells. The question is simple: When material is injected into an injection well, where does it go? Figure 2 shows an example of a pattern with 4 injection wells and one producer.

The chromosome for this pattern would consist of 4 groups of 4 values each (in general, this can vary). For each group, the normalized values (called "allocation factors") represent the percentage of injected material that flowed northwest, northeast, southwest or southeast out of each injection well. A key constraint on this calculation is the conservation of the total mass of injected and produced fluids, hence the name "material balance".

The real world problems we deal with typically have anywhere from 3000 to 7000 floating point allocation factors and hundreds of individual patterns.

3 SOLUTION ATTEMPTS

The initial optimization attempt (as well as all subsequent attempts) used the methodology described earlier of using individual values in the range [0..1], clustered into subgroups that must be normalized before evaluation. The first version used a genetic algorithm with floating point strings. It was successful on small examples (9 patterns), but did not scale linearly in computation time to the full field. A later attempt using a genetic algorithm with bit strings gained a significant speedup, and the addition of gray coding (Wright, 1991) resulted in another significant gain.

Still not satisfied with the genetic algorithm's performance, we used a bit climber (Davis, 1991a) (stochastic gradient technique) and were surprised to see another increase in performance. A bit climber emulates a gradient search by flipping a bit and evaluating the new solution to see if it is an improvement. Since this is a "greedy" search strategy, i.e., we never throw away an improved solution vector (Horowitz, 1978), the bit climber can get stuck in suboptimal peaks or troughs called local optima (Davis, 1991b). Various heuristics were applied to help alleviate this problem.

Two additional modifications to the bit climber resulted in even more speedup. When finished, we had achieved a factor of 200 speedup over the original code, allowing us to run full field studies on an SGI workstation overnight (12 to 15 hours).

Such hill climbing strategies are well known in the literature, and many successful examples of their use can be found (Ishibuchi, 1997; Land, 1997; Rana, 1997). In addition to hill climbing and bit climbing, these techniques are also referred to as "local search" (Yannakakis, 1990), "iterative improvement" (Vaessens, 1992) and "hybrid methods" (when used together with a genetic algorithm (Goldberg, 1989). Many variants exist such as steepest ascent and nearest ascent (Mühlenbein, 1991; Mühlenbein 1992) and random (Davis, 1991a; Forrest, 1993).

In the next sections we will describe our modified gradient techniques for both bit and floating point representations.

3.1 BIT CLIMBER

We began by using a very simple, very standard bit climbing technique shown in Algorithm 1. This technique was successful, suggesting our problem may not be overly multimodal. Not necessarily unimodal, but perhaps having only a small number of peaks.

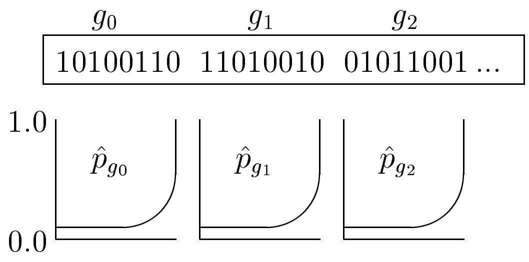

Figure 3: Bit Flip Probabilities

Generate and evaluate a random bit string
Do until stopping criteria satisfied:
    Randomly select a position on the string
    Flip the bit at that position
    Evaluate the new string
    If the fitness is worse, unflip the bit
    Rerandomize the string if no changes
        have occurred lately

Algorithm 1: Standard Bit Climber

3.2 MODIFIED BIT CLIMBER

When considering the fact that we normalized our solution vector before evaluating, it made sense to force the bit climber to only make small changes to the string at any one time. This would minimize disruption in the normalization step.

To accomplish this, we assigned a probability, $\hat{p}_i$ to each bit in the string. In an attempt to avoid some notational confusion, we will refer to an individual bit's probability as $\hat{p}_i$, and the array of probabilities for the entire k'th group as $\hat{p}_{g_k}$.

We used 10 bits to represent each floating-point value (giving us a resolution of approximately .001, which is more than enough accuracy for the material balance problem). For each set of 10 bits, we assigned a group of 10 probabilities that determine how likely we are to flip a particular bit. Low order bits receive a probability of 1.0, while higher order bits receive an exponentially decreasing amount, based on a constant in the range [0...1] supplied by the user to the program (we used 0.25 for most tests). In other words, the second low order bit had probability of 0.25, the third 0.125, and so on. Figure 3 illustrates.

Our modified bit climber is shown in Algorithm 2. This technique forced smaller changes to the normalized vector. While the smaller changes made it climb more slowly, it resulted in a higher number of successful bit flips (i.e., where the fitness improved), and ran faster than the unmodified version.

Generate and evaluate a random bit string
Do until stopping criteria satisfied:
    Randomly select a position on the string
    Randomly generate a number from 0..1
    If number < probability for this bit:
        Flip the bit at that position
        Evaluate the new string
        If the fitness is worse, unflip the bit.
    Rerandomize the string if no changes
        have occurred lately

Algorithm 2: Modified Bit Climber

3.3 A BIT OF MEMORY?

Our final modification to the bit climber involves keeping track of which bits have been flipped, and lessening the likelihood of flipping them again (Jones, 1995). After successfully flipping a bit (resulting in an improved fitness), we multiply the probability of flipping that bit by some user-supplied constant (say, 0.5). So if we successfully flip the lowest order bit, which has a probability of 1.0, we would change its probability to 0.5, giving us less chance to flip the bit again. If we flip it again, we will change it to 0.25, and so on. As a result, we decrease the mutation amount at each step (Mahfoud, 1995). The new algorithm is shown in Algorithm 3.

Generate and evaluate a random bit string
Do until stopping criteria satisfied:
    Randomly select a position on the string
    Randomly generate a number from 0..1
    If number < probability for this bit:
        Flip the bit at that position
        Evaluate the new string
        If fitness is worse, unflip the bit
        Decrease probability for this bit
    If no recent improvements, rerandomize,
        reset probabilities and start over

Algorithm 3: Adding Memory

As stated a number of times, these techniques worked well, but they are not without problems. The worst problem is that they tend to get stuck searching for a better solution after a while, because all of the low order bits have been flipped, but the probabilities are too low to be able to flip any of the high order ones very often. With the addition of memory, the problem is compounded. As the probabilities decrease, it becomes more difficult to find a bit that will allow itself to be flipped.

Decimal	Binary	Gray Code
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

Table 1: Table 1: Decimal and Binary codes

Numerous heuristics have been applied to alleviate this problem. For example, we keep a count of how many times you have been disallowed from flipping bits, together with an occasional resetting of the probabilities when you reach a preset threshold (losing one's memory, so to speak). After doing this several times, we eventually set all probabilities to 1.0 and continue without memory (losing one's memory and preferences). After some time, we simply give up, rescramble the bits, reset the probabilities and start anew.

However, a detailed analysis of these bit climbers leads us to other, more attractive alternatives, as shown in the next section.

4 MODIFIED BIT CLIMBER ANALYSIS

A mathematical analysis of the modified bit climber allows us to define a floating-point version of the algorithm that runs much faster. For reference, Table 1 shows decimal and binary values with corresponding gray codes.

Consider what happens to the decimal value when you flip a binary bit using the standard binary to decimal encoding. We define a function, $\Delta(i)$, which gives us the decimal change when bit i is flipped. Table 2 shows what happens. Likewise, Table 3 shows what happens to the decimal values when you flip a gray coded bit.

Note that on average, for the binary encoding, flipping bit 0 results in an average absolute change of 4. Bit 1 averages 2, and bit 2 averages 1. These correspond directly with powers of 2 as shown in Equation 1. Since we are dealing with groups, we can more generally define the function as shown in Equation 2. Note that the same function also applies for Gray coded bits, when considering an absolute average (the average of the absolute values, which tells us the average net change in the decimal value).

Binary	$\Delta(2)$	$\Delta(1)$	$\Delta(0)$
000	+4	+2	+1
001	+4	+2	-1
010	+4	-2	+1
011	+4	-2	-1
100	-4	+2	+1
101	-4	+2	-1
110	-4	-2	+1
111	-4	-2	-1
Avg Chg	$2^2=4$	$2^1=2$	$2^0=1$

Table 2: Table 2: Binary Bit Flip Effects

Gray	$\Delta(2)$	$\Delta(1)$	$\Delta(0)$
000	+7	+3	+1
001	+5	+1	-1
010	+3	-1	+1
011	+1	-3	-1
100	-1	+3	+1
101	-3	+1	-1
110	-5	-1	+1
111	-7	-3	-1
Avg Chg	$2^2=4$	$2^1=2$	$2^0=1$

Table 3: Table 3: Gray Code Bit Flip Effects

$$\Delta(i) = 2^{l-1-i} \quad (1)$$

$$\Delta_k(i) = 2^{g_{k+1}-1-i} \quad (2)$$

What we see from this analysis is that the absolute average decimal change for a particular bit position is the same for binary and gray coded bits. This means that flipping the lower order gray coded bits will, on average, still produce relatively smaller decimal changes, and thus smaller floating point changes in our normalized solution vector the same as with regular binary encoding.

Consider the probabilities in the modified bit climber. We generate higher probabilities for the lower order bits, and lower probabilities for the higher order bits. Recall that we represent the array of probabilities for each group g_k as $\hat{p}_{g_k}$. In order to compute the statistics of the actual effect of a bit flip (i.e., the mean and standard deviation of the absolute change), we must normalize these values for each individual group, which will give us an actual probability distribution.

Note that the original bit climber is equivalent to the modified bit climber with all $\hat{p}_i = 1.0$. The probability distribution for the group is the normalization of each group's probabilities, as shown in Equation 3.

$$p_{g_k} = normalize(\hat{p}_{g_k}) \quad (3)$$

For our 4-bit case with the original bit climber, we have $(0.25, 0.25, 0.25, 0.25)$. In general, given any p_{g_k} for a single group, the average decimal change is described in Equation 4.

$$\sum_{i=g_k}^{g_{k+1}-1} p_i \Delta_k(i) = M_{g_k} \quad (4)$$

We can now compute the average integer change for a bit flip on any group. But since we are converting these into floating-point values, we need to find out what our floating-point resolution is. In other words, find the smallest floating-point number we can represent with our bit string. In a 4-bit example, it is simply $\frac{1}{2^4} = \frac{1}{16}$. In general, Equation 5 defines our epsilon floating point resolution for using subgroups consisting of l bits.

$$\epsilon = \frac{1}{2^l} \quad (5)$$

While this assumes we are storing values in the range [0...1] with identical precision for each floating-point value, it can trivially be adapted to stings of varied range and precision. Now we only need to multiply this epsilon to our integer mean to get a floating point mean as shown in Equation 6.

$$\epsilon M_{g_k} = \mu_{g_k} \quad (6)$$

Lastly, we can compute a standard deviation as shown in Equation 7.

$$S_{g_k} = \sqrt{\sum_{i=g_k}^{g_{k+1}-1} [p_i(\Delta_k(i) - M_{g_k})^2]} \quad (7)$$

And Equation 8 completes the analysis by converting our standard deviation to its floating-point counterpart.

$$\epsilon S_{g_k} = \sigma_{g_k} \quad (8)$$

5 FLOATING POINT GRADIENT TECHNIQUE

Now that we can easily calculate, from the provided probabilities, a floating-point mean and standard deviation for each subgroup, we can construct a floating-point version of the modified bit climber. Unlike the bit climbers, the floating point version never gets stuck not being able to make a change to the string, as it only has to select a position on the string, and then generate a mutation value normally distributed around the mean and standard deviation ($\delta = N(\mu, \sigma)$). As a result, it runs around 10 times faster than the bit version. Algorithm 4 outlines the process.

Randomly generate a floating point string
Evaluate the string
Untilstopping criteria satisfied:
    Select a single string position, i
    Generate a mutation value $N(\mu, \sigma)$
    Add mutation to the value at position i
    Evaluate the new string
    If fitness is worse, undo the mutation

Algorithm 4: Floating Point Technique

Adding memory to this technique is a bit more cumbersome, but worth the effort. You must convert the mutation value into its integer counterpart (as if you were using encoded bit strings) and locate the nearest power of two. The base 2 log of this value is the probability that needs to be reduced. Once it has been reduced, then the mean and standard deviation must then be recalculated. The overall effect is that the means and standard deviations continue to get smaller and smaller. This has the nice effect of larger mutations at the beginning, and smaller ones at the end. The result is a self-tuning system. The technique is the same as in Algorithm 4, except that you update the mean and standard deviation after a successful mutation.

6 CONCLUSIONS

It is difficult to make honest claims about one technique being superior to another. The fact that the original gray-coded bit version of the genetic algorithm outperformed the floating-point version is a simple matter of differences of implementation techniques (e.g., crossover and mutation operators, generational vs. steady state model, bit vs. floating point). There were far too many variables to make such a call.

Tests on numerous optimization problems at ARCO have shown that these modified gradient techniques often outperform both the genetic algorithm and evolutionary programming. This suggests that either the technique is robust, or that our problems are not as complicated as we once thought, or perhaps both. We have also encountered situations where the bit climber does not perform well, and then the genetic algorithm and evolutionary programming techniques are necessary (e.g., Stoisits, 1999).

The optimization tests performed at ARCO strongly suggest that minimizing changes during recombination, together with avoiding undoing recent changes, have significant advantages when dealing with normalized solution spaces. The gradient techniques described above are the fastest and most reliable methods we have found to date for solving such problems.

Additional research is required to explore the dynamics of these normalized search spaces. Further planned ARCO research includes a test function generator and additional enhancements to the floating-point gradient technique with memory. Of particular interest is the self-tuning nature of the algorithm, and how well it performs on future ARCO optimization problems.

References

L. Davis (1991). "Bit-climbing, Representational Bias and Test Suite Design", In L. Booker and R. Belew, editors, Proceedings of the Fourth International Conference on Genetic Algorithms, Morgan Kaufmann Publishers, San Francisco, CA.

L. Davis (1991). "Handbook of Genetic Algorithms", Van Nostrand Reinhold, New York.

S. Forrest, M. Mitchell (1993). "Relative Building-Block Fitness and the Building-Block Hypothesis", In L. Darrell Whitley, editor, Foundations of Genetic Algorithms, 2, Morgan Kaufmann Publishers, San Mateo, CA.

D. E. Goldberg (1989). "Genetic Algorithms in Search, Optimization & Machine Learning", Addison-Wesley, Reading, MA.

W. Hager (1988). "Applied Numerical Linear Algebra", Prentice Hall, Englewood Cliffs, NJ.

E. Horowitz, S. Sahni (1978). "Fundamentals of Computer Algorithms", Computer Science Press, Inc., Rockville, MD.

H. Ishibuchi, T. Murata, S. Tomioka (1997). "Effectiveness of Genetic Local Search Algorithms", In Thomas Bck, editor, Proceedings of the Seventh International Conference on Genetic Algorithms, Morgan Kaufmann Publishers, San Francisco, CA.

T. Jones (1995). "Crossover, Macromutation, and Population-Based Search", In Larry J. Eshelman, editor. Proceedings of the Sixth International Conference on Genetic Algorithms, Morgan Kaufmann Publishers, San Francisco, CA.

M. Land, J. Sidorowich, R. Belew (1997). "Using Genetic Algorithms with Local Search for Thin Film Metrology", In Thomas Bck, editor, Proceedings of the Seventh International Conference on Genetic Algorithms, Morgan Kaufmann Publishers, San Francisco, CA.

M. D. McCormack, D. J. MacAllister, K. D. Crawford, R. F. Stoisits (1999). "Maximizing Production from Hydrocarbon Reservoirs Using Genetic Algorithms", to appear, The Leading Edge, SEG, Tulsa, OK.

S. Mahfoud (1995). "A Comparison of Parallel and Sequential Niching Methods", In Larry J. Eshelman, editor. Proceedings of the Sixth International Conference on Genetic Algorithms, Morgan Kaufmann Publishers, San Francisco, CA.

H. Mühlenbein (1991). "Evolution in Time and Space - the Parallel Genetic Algorithm", In G. Rawlins, editor, Foundations of Genetic Algorithms, Morgan Kaufmann Publishers, San Francisco, CA.

H. Mühlenbein (1992). "How Genetic Algorithms Really Work; I. Mutation and Hillclimbing", In R. Mnner and B. Manderick, editors, Parallel Problem Solving from Nature, 2, Elsevier Science Publishers, The Netherlands.

S. Rana, D. Whitley (1997). "Bit Representations with a Twist", In Thomas Bck, editor, Proceedings of the Seventh International Conference on Genetic Algorithms, Morgan Kaufmann Publishers, San Francisco, CA.

R. F. Stoisits, K. D. Crawford, D. J. MacAllister, M. D. McCormack, A. S. Lawal, D. O. Ogbe (1999). "Production Optimization at the Kuparuk River Field Utilizing Neural Networks and Genetic Algorithms", In 1999 Mid-Continent Operations Symposium (SPE Paper 52177), Society of Petroleum Engineers, Richardson, TX.

R. J. M. Vaessens, E. H. L. Aarts, J. K. Lenstra (1992). "A Local Search Template", In R. Mnner and B. Manderick, editors, Parallel Problem Solving from Nature, 2, Elsevier Science Publishers, The Netherlands.

A. Wright (1991). "Genetic Algorithms for Real Parameter Optimization In G. Rawlins, editor, Foundations of Genetic Algorithms, Morgan Kaufmann Publishers, San Francisco, CA.

M. Yannakakis (1990). "The Analysis of Local Search Problems and Their Heuristics", Proceedings of the 7th Annual Symposium on Theoretical Aspects of Computer Science (STACS 90), Lecture Notes in Computer Science 415, Springer, Berlin.