Constant Time Steepest Descent Local Search with Lookahead for NK-Landscapes and MAX-kSAT

Darrell Whitley and Wenxiang Chen
Department of Computer Science
Colorado State University
Fort Collins, CO 80523
{whitley,chenwx}@cs.colostate.edu

ABSTRACT

A modified form of steepest descent local search is proposed that displays an average complexity of $O(1)$ time per move for NK-Landscape and MAX-kSAT problems. The algorithm uses a Walsh decomposition to identify improving moves. In addition, it is possible to compute a Hamming distance 2 statistical lookahead: if x is the current solution and y is a neighbor of x, it is possible to compute the average evaluation of the neighbors of y. The average over the Hamming distance 2 neighborhood can be used as a surrogate evaluation function to replace f. The same modified steepest descent can be executed in $O(1)$ time using the Hamming distance 2 neighborhood average as the fitness function. In practice, the modifications needed to prove $O(1)$ complexity can be relaxed with little or no impact on runtime performance. Finally, steepest descent local search over the mean of the Hamming distance 2 neighborhood yields superior results compared to using the standard evaluation function for certain types of NK-Landscape problems.

Categories and Subject Descriptors

I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search

General Terms

Theory, Algorithms

Keywords

Local Search, Elementary Landscapes, NK-Landscapes, MAX-kSAT

1. INTRODUCTION

NK-Landscapes and MAX-kSAT are important examples of the family of k-bounded pseudo-Boolean functions. In this paper we prove that a modified form of steepest descent can be constructed that selects the best improving move from N neighbors in $O(1)$; this means that we never have to evaluate most neighbors and, instead, an $O(1)$ time analysis tells us which neighbors could be improving moves. This is an average complexity result.

GECCO'12, July 7-11, 2012, Philadelphia, Pennsylvania, USA.

Our empirical validation focuses on NK-Landscapes [5], and NKq-Landscapes [2]. However, the theoretical results presented in this paper equally apply to MAX-kSAT and other k-bounded pseudo-Boolean functions. Both NKq-Landscapes and MAX-kSAT display plateaus and result in more equally good improving moves compared to standard NK-Landscapes. The proposed methods of speed-up have the greatest advantage when $N > 2^k$. For example, these methods are highly efficient on problems such as MAX-3SAT, where $N \gg 2^3$ [9].

In an NK-Landscape, N subfunctions are evaluated, where each subfunction depends on exactly $k = K + 1$ bits. If a bit is flipped, a subfunction's evaluation does not change unless the subfunction references the bit that has changed value. In the worst case, calculating which neighbors could yield an improving move requires $O(N)$ time.

The $O(1)$ complexity can be achieved on average because we use a Walsh decomposition of the evaluation function to expose bit interactions in the pseudo-Boolean functions. This information is then used to construct a partial update evaluation function that calculates which neighbors could yield an improving move after a bit flip. On average only a constant number of the neighbors can feasibly change their *relative* evaluation after a bit flip. This also makes it possible to identify a local optimum in $O(1)$ time because, on average, only a constant number of neighbors need to be examined to confirm a local optimum.

The proof assumes that costs are amortized uniformly across all variables, or that the search can avoid flipping the same high cost variables too often. Empirical data shows the restrictions required for the proof are not necessarily in practice. In practice, the high cost variables have a low rate of being flipped. Thus, our implementation uses a standard form of steepest descent without the restrictions needed for the proof, while still achieving $O(1)$ time performance.

For example, when $K = 4$ our empirical results show that as N is increase from $N = 20$ to $N = 500$ the reoccurring execution cost remains virtually constant, increasing from 1.56 seconds for $N = 20$ to only 2.15 seconds for $N = 500$ while the size of total space explored increased from 20,000 to 500,000 points. We achieve this advantage because our analysis tells us where the improving moves will be: we never have to examine most neighbors.

We build on the theory of "Elementary Landscapes" to compute the average of neighborhoods 2 moves ahead in $O(1)$ time. Sutton et al. [8] provide a method to compute the average over all of the $\binom{N}{d}$ solutions that are exactly Hamming distance d away from the current solution. The methods presented here computes much more localized information about averages over neighborhoods that are 2 moves away. Empirical tests suggests that local search over the mean of the neighborhoods 2 moves ahead yields superior results

compared to using the actual evaluation function; this is particularly true for NKq-Landscapes where plateaus and neutral moves are more likely to occur.

2. THE WALSH TRANSFORM

Any discrete function $f : \{0,1\}^n \implies \mathbb{R}$ can be represented in the orthogonal Walsh basis:

$$f(x) = \sum_{i=0}^{2^n-1} w_i \psi_i(x)$$

where w_i is a real-valued constant called a Walsh coefficient and $\psi_i(x) = -1^{i^T x}$ is a Walsh function that generates a sign. The Walsh coefficient is then added or subtracted in the calculation of $f(x)$ depending on the sign generated by the Walsh function ψ.

The Walsh transform can be applied to f as follows:

$$w = (1/N)\,\mathbf{W}f, \quad f = \mathbf{W}^{-1}w$$

where N is the neighborhood size and $\mathbf{W}$ is a $2^N \times 2^N$ matrix

$$\mathbf{W}_{ij} = \psi_i(j) = -1^{i^T j}$$

Normally a Walsh transform requires that the entire search space be enumerated. However, Rana et al. [6] show for MAX-kSAT problems that if a function is composed of subfunctions, where each subfunction is a function over exactly k Boolean variables, then each subfunction f_j contributes at most 2^k nonzero Walsh coefficients to the Walsh transform of $f(x)$. Heckendorn also used Walsh decomposition to study NK-landscapes [3] [4].

Let X denote the search space: the set of binary strings of length N. Each bit corresponds to a single Boolean variable in a k-bounded pseudo-Boolean function. The k-bound functions are constructed from M subfunctions, where each subfunction is a pseudo-Boolean function of at most k bits. In MAX-kSAT each clause acts as a subfunction returning true or false (1 or 0). In NK-landscapes, $M = N$ and each subfunction i is associated with the bit in position i and K other randomly chosen bits. Thus:

$$f(x) = \sum_{j=1}^{M} f_j(x)$$

and f_j evaluates the j^{th} function for string x. We can apply the Walsh transform to each subfunction over k bits:

$$w = \sum_{j=1}^{M} (1/k)\,\mathbf{W}f_j$$

where w is a vector of polynomial coefficients. This generates the Walsh coefficients associated with each subproblem, and then adds them together as needed. The vector w is highly sparse and for fixed k and $M = O(N)$ the number of nonzero Walsh coefficients is at most $M * 2^k = O(N)$.

For $0 \leq i < 2^n$, the i^{th} Walsh function can also be defined as

$$\psi_i(x) = -1^{i^T x} = -1^{bitcount(i \wedge x)}$$

where $bitcount(i \wedge x)$ counts how many 1-bits are in the string produced by the logical operation $i \wedge x$. String i can be used to identify which combination of bits potentially interact and thus contribute to a particular Walsh coefficient. The *order* of the i^{th} Walsh function is $bitcount(i)$ which returns the number of 1 bits in the binary representation of i.

The Walsh function generates a positive sign if $bitcount$ is even, and a negative sign if the $bitcount$ is odd. Each Walsh coefficient w_i contributes a +/- difference around the mean $\bar{f} = w_0$. It is critical to note that if a single bit flip in string x does not impact a bit in string i it has no impact on Walsh coefficient contribution $\psi_i(x)w_i$. If the bit flip in x does impact a bit in string i it flips the sign generated by $\psi_i(x)$ since $bitcount$ changes by exactly 1. Assuming search starts at a random candidate solution, the Walsh functions would have to be computed once to initialize the signs associated with each Walsh coefficient. After local search starts, the signs can be updated on the fly without using the Walsh functions.

We will use the vector w' to store the Walsh coefficients, and this will include the sign relative to the current solution x.

$$w'_i(x) = w_i \psi_i(x)$$

We will need to index bits by position. This will be done from right to left starting at position 1. Thus we can use $\mathbf{p}$ to denote the position of variable $\mathbf{p}$. We can then define the integer value of the string p relative to the bit position $\mathbf{p}$ as follows:

$$p = 2^{\mathbf{P}-1}$$

Thus the string 01000 corresponds to integer $p = 8$ but the "1" bit appears in position $\mathbf{p} = 4$. This notation makes it unnecessary to use an explicit mask to isolate particular variable locations. It also makes it possible to address bit locations using addition.

We will also introduce a simple convention to aid the reader. Consider an NK-landscape or MAX-3SAT problem where $K+1 = k = 3$. Let p, q, r each represent a single bit in string b, where b has length N. Assume that bits are set to 1 at bit values p, q, r (respectively, at positions $\mathbf{p}, \mathbf{q}, \mathbf{r}$) and 0 elsewhere in b, such that $b = p + q + r$. At the first initialization w' is assigned as follows:

$$w'_{p+q+r}(x) = w'_b(x) = w_b \psi_b(x)$$

Such a Walsh coefficient captures the order-3 interactions among variables p, q and r. We will also need the order-2 interactions. If a subfunction references the variables with bit values p, q and r it will also generate the following order-2 Walsh coefficients:

$$w'_{p+q}(x), \quad w'_{p+r}(x), \quad w'_{q+r}(x)$$

and similarly these must be initialized:

$$b = p + q \quad \implies \quad w'_{p+q}(x) = w'_b(x) = w_b \psi_b(x)$$

Finally, $w'_p(x) = w_p \psi_p(x)$. Since we capture the sign as part of the w' vector it is unnecessary to explicitly compute ψ.

Assume the Walsh coefficients and their signs have been computed for some initial solution x; let b index all of the Walsh coefficients in vector $w'(x)$. Let $p \subseteq b$ denote that the bit in position $\mathbf{p}$ has value 1 in string b. We can then compute the sum of all of the components in w' that are affected when local search flips bit p.

$$S_p(x) = \sum_{\forall b,\ p \subseteq b} w'_b(x)$$

In this way, all of the Walsh coefficients whose signs will be changed by flipping bit p are collected into a single number $S_p(x)$. This leads to Lemma 1.

Lemma 1.
Let $y_p \in N(x)$ be the neighbor of string x generated by flipping bit p. Then $f(y_p) = f(x) - 2(S_p(x))$.

Proof: By definition

$$S_p(x) = \sum_{\forall b,\ p \subseteq b} w'_b(x)$$

If $p \subseteq b$ then $\psi_b(y_p) = -1(\psi_b(x))$ and otherwise $\psi_b(y_p) = \psi_b(x)$. Therefore all of the Walsh coefficients that contribute to the sum $S_p(x)$ change sign, and no other Walsh coefficient changes sign when bit y_p is flipped. If a Walsh coefficient changes from negative to positive, the change is $-2(w'_b(x))$, and if the change is from positive to negative, the change is still $-2(w'_b(x))$. □

Corollary: For all bit flips j, $f(y_j) = f(x) - 2(S_j(x))$. Thus, $S_j(x)$ can be used as a proxy for $f(y_j)$ because $f(x)$ is constant as j is varied. Selecting the maximal sum $S_{max}(x)$ yields the best improving (minimizing) move in the neighborhood of $f(x)$.

3. STEEPEST DESCENT

Let x be the current solution. After bit p is flipped and the neighbor $y_p \in N(x)$ is accepted, the vector S must be updated.

If bit p is flipped, $S_p(y_p) = -(S_p(x))$ since every Walsh coefficient that contributed to $S_p(x)$ changes sign. Thus, this update occurs in constant time.

If bit p is flipped and there are no Walsh coefficients indexed by a that also contributes to $S_p(y_p)$, then $S_a(y_p) = S_a(x)$ and the summation for that entry in S is unchanged.

After p is flipped, we must update sums that include Walsh coefficients that are jointly indexed by p. Let b represent a bit string such that w'_b is a nonzero Walsh coefficient.

$$S_q(y_p) = S_q(x) - 2 \sum_{\forall b,(p \wedge q) \subseteq b} w'_b(x)$$

This means that if there exists w'_{p+q} then $-2w_{p+q}$ must be added to $S_q(y_p)$. If there exists w'_{p+q+r} then $-2w_{p+q+r}$ must be added to $S_q(y_p)$ and $S_r(y_p)$.

The vector $w'(x)$ must also be updated. The sign is flipped for those coefficients affected by flipping bit p.

$$\begin{array}{lll} \forall b, & if \;\; p \subseteq b, & w'_b(y_p) = -w'_b(x) \\ & otherwise & w'_b(y_p) = w'_b(x) \end{array}$$

Lemma 2

In the worst case, updating the vectors $S(y_p)$ and $w'(y_p)$ requires $O(N)$ time.

Proof:

In the worst case, there exists some variable p that appears in every subfunction so that every other variable is also impacted by flipping p and all of the N sums in S must be updated. Nevertheless, all of the sums in S can be updated in $O(N)$ time since there are at most 2^k Walsh coefficients for each subfunction, and each subfunction contributes to at most k different sums in S. □

3.1 One-time Costs

There are some one-time costs that must be absorbed before search can begin. Obviously, there is the cost of the Walsh analysis: there are N subfunctions that each generates 2^k Walsh coefficients. This clearly has $O(N)$ start-up cost. The S vector must also be initialized, which also has $O(N)$ cost.

The first move can also have $O(N)$ costs. If search is started from a local extremum, then every neighbor is an improving move. Therefore, we need to worry about the case where there are too many (e.g. $O(N)$) improving moves, even if this rarely occurs.

3.2 The O(1) average case analysis

To prove an $O(1)$ average cost, we will assume that a modified Tabu mechanism is used where all N bits must be flipped before any bit is allowed to be flipped again. After all N bits are flipped, the Tabu list is cleared and restarted. This strong assumption will provide an exact average time result. In both theory (see the Corollary to Theorem 1) and in practice, we will relax this assumption. We will assume that improving moves are stored in a buffer B and selected as needed.

Theorem 1

Assume that a Tabu mechanism is used so the same bit is not allowed to flip until all N bits are also flipped. Under steepest descent with the Tabu mechanism the cost of updating the vector S and buffer B can be amortized over N steps to yield an average number of updates involving at most $k(k-1)2^{k-2} = O(1)$ Walsh coefficients. The best new move from the updates and the approximate best old move from the buffer can be selected in $O(1)$ time.

Proof:

At step one, the vector S is created and all of the improving moves are placed in the buffer B. Then the best move p is selected. This is a one time $O(N)$ cost.

After every move the vector S only contains 1) equal or disimproving moves, 2) Tabu moves 3) improving moves stored in buffer B and 4) elements that have been updated. New improving moves can be selected from the buffer if they are not Tabu.

After N steps every bit has been flipped 1 time. For each bit flip there will be one update of the form $S_p(y_p) = -S_p(x)$. Note that this operator involves a simple change of sign. This is the only update that changes a linear Walsh coefficient.

When p flips, other updates to S (involving higher order Walsh coefficients) are given by:

$$S_i(y_p) \;\; = \;\; S_i(x) - 2 \sum_{\forall b,(p \wedge i) \subseteq b} w'_b(x)$$

If there are k variables per subfunction, then when 1 bit flips, there are $k-1$ other sums in S that must be updated. After all k variables are flipped, there is a total of $k(k-1)$ updates to sums in S per subfunction. Over M subfunctions there would be at most $k(k-1)M$ updates to S plus the N linear updates to $S_p(y_p)$. Assume $M = cN$. When the update cost is amortized over all N bits, the average number of elements in S that must be updated is at most

$$(k(k-1)M + N)/N = k(k-1)c + 1.$$

For NK-landscapes $c = 1$; for MAX-kSAT, c is the clause to variable ratio. We must also account for each Walsh coefficient that changes sign and therefore contributes to the updates in S. A Walsh coefficient of order j is affected by j bit flips, each of which results in an update to at most $j-1$ other sums. For each subfunction there are at most $\binom{k}{j}$ order j Walsh coefficients. Thus after all N bits flip, the number of updates per subfunction is at most

$$\sum_{j=2}^{k} j(j-1)\binom{k}{j} = k(k-1)2^{k-2}$$

which is multiplied by M over all subfunctions. When this work is amortized over all N bits, the average number of operations involving Walsh coefficients is at most $ck(k-1)2^{k-2} = O(1)$.

An improving move can only be found in the buffer B or in the set of updates at each time step. The average cost of checking the updated sums in S is $k(k-1)c + 1 = O(1)$ on average after each bit flip. The cost of updating improving moves in the buffer is also

N	K=2		K=4		K=8	
	mean	std.	mean	std.	mean	std.
20	-0.42	0.08	-0.57	0.12	-0.53	0.14
50	-0.41	0.10	-0.56	0.08	-0.62	0.11
100	-0.25	0.07	-0.39	0.08	-0.52	0.06

Table 1: Correlation between the number of Walsh updates and the numbers of bit-flips per variable. The negative correlation shows that the variables that flip the most are those that result in fewer updates. The mean and standard deviation (std) over 100 independent runs are given for each parameter combination.

$O(1)$ since the elements that are updated in the buffer are a subset of the elements updated in S.

Finally, we must identify the best improving move. We have two sources of improving moves. There are previously identified "old" improving moves in the buffer, and there are the "new" improving moves just added to the buffer. We can identify the best move from a new update in $O(1)$ time on average. We then need to identify the best "old" move already in the buffer. In the worst case, there could be $O(N)$ old improving moves. Let β be the number of old moves in the buffer. We then set a threshold α. If $\beta \leq \alpha$ we scan all of the old moves to find the best old move in $O(1)$ time. If $\beta > \alpha$ we sample α of the old improving moves and select the *approximate* best old move in $O(1)$ time. We then select the best improving move from the best old move and new move. □

Corollary: Assume only those variables that appear more than $\mathbf{T}$ times across all subfunctions become Tabu after 1 flip, where $\mathbf{T}$ is a constant. All other variables can be freely flipped. Each variable that is Tabu remains Tabu for N bit flips independent of all of the other variables. Thus every possible sequence of N flips has an average complexity per move to update S and B that is $O(1)$.

3.3 Theory, Practice and Approximations

There are only two issues that require modifications to steepest descent to ensure $O(1)$ average complexity: 1) limiting the number of times that high cost bit flips occur and 2) making sure there are not too many unexploited old improving moves.

A bit variable will in expectation appear in k subfunctions of a randomly constructed NK-landscape. If the threshold is set at $\mathbf{T} = k$, then in expectation only half of the variables will be Tabu. Alternatively, we can constructed a special class of NK-landscapes in which each variable occurs in exactly k subfunctions; for this class of problems *none* of the variables would be Tabu.

In general, if the threshold is set at $\mathbf{T} = 2k$ before a variable becomes Tabu, then the majority of variables will not be Tabu. Furthermore, if the expected waiting time between flips is greater than or equal to N no Tabu mechanism is needed.

In our experiments we were able to ignore the Tabu restrictions. Empirical data in Table 1 shows there is a negative correlation between how often a bit appears in subfunctions and how often it flips under a traditional steepest descent algorithm. The bits that appear more often in subfunctions (and thus might be Tabu) are the bits that flip the least. Bits that appears in fewer subfunctions are flipped at a higher rate. This suggests that the bits that appear in more subfunctions have a greater impact on the evaluation function, and thus they are less likely to change under steepest descent.

Theorem 1 also introduces an approximation to find the best improving move. We could maintain a heap of the best improving moves, which would allow us to do true steepest descent with an $O(lg\ N)$ worst case time complexity when there are too many improving moves. But this does not appear to be necessary.

The approximation is only used if 1) the best available move is an old move previously stored in the buffer and 2) $\beta > \alpha$ and there are too many old improving moves. Otherwise, the algorithm selects the true steepest descent move. In fact, most of the time the number of improving moves was very small. Therefore we just scanned the entire buffer to obtain the true steepest descent move. This also has the advantage that our empirical results exactly match standard steepest descent local search.

In the case of MAX-kSAT, an even more elegant approximation exists. Again, the "new" improving moves can be checked in $O(1)$ time on averages. In MAX-kSAT, a change in an entry in S can be scaled so that the value of entry S_p directly computes how many additional clauses are satisfied. Assume $B+1$ buckets are created to store (old) improving moves. Number the buckets from 0 to B, where $|B|$ is a constant. If an improving move satisfies $0 \leq i < B$ additional clauses, that move is placed in bucket i. If an improving move satisfies $i \geq B$ additional clauses, it is placed in bucket B. If bucket B is empty then the steepest descent move is selected by finding the first non-empty bucket, scanning from buckets B to O. If bucket B contains more than 1 move, a move is chosen randomly from bucket B to obtain the "old" approximate best move. This method also allows equal moves to be tracked. Also note that the approximate best improving must satisfy B or more clauses [9].

3.4 Computing AVG(N(x)) in O(1) time

Let $Avg(N(x))$ compute the average evaluation of $N(X)$ which generatates the set of the neighbors of x.

$$\begin{aligned} Avg(N(x)) &= 1/N \sum_{\mathbf{i}=1}^{N} f(y_i) \quad \text{where } y_i \in N(x) \\ &= 1/N \sum_{\mathbf{i}=1}^{N} (f(x) - 2(S_i(x)) \\ &= f(x) - 2/N \sum_{\mathbf{i}=1}^{N} (S_i(x)) \qquad (1) \end{aligned}$$

Sutton et al. [7, 8] derive the same basic result using principles from elementary landscapes.

Let $\varphi'_{p,j}$ denote the sum of all Walsh coefficients of order j that reference bit p.

$$\varphi'_{p,j}(x) = \sum_{\forall b, bitcount(b)=j, p \subseteq b} w'_b(x)$$

where $bitcount(b) = j$ is the order of coefficient w_b.

We can now define the update for the vector S as follows, for $y_p \in N(x)$:

$$\begin{aligned} S_i(x) &= \sum_{j=1}^{k} \varphi'_{i,j}(x) \\ S_i(y_p) &= S_i(x) - 2 \sum_{\forall b, (p \wedge i) \subseteq b} w'_b(x) \qquad (2) \end{aligned}$$

We next define a new vector Z that computes a parallel result.

$$\begin{aligned} Z_i(x) &= \sum_{j=1}^{k} j\varphi'_{i,j}(x) \\ Z_i(y_p) &= Z_i(x) - 2 \sum_{\forall b, (p \wedge i) \subseteq b} bitcount(b)(w'_b(x)) \end{aligned}$$

where $bitcount(b)$ is implemented as a lookup table that stores the order of Walsh coefficient w'_b.

Lemma 2.

For any k-bound pseudo-Boolean function, when flipping bit p and moving from solution x to solutions $y_p \in N(x)$:

$$Avg(N(y_p)) = Avg(N(x)) - 2S_p(x) + \frac{4}{N} Z_p(x)$$

Proof:

$$\sum_{\mathbf{i}=1}^{N} \sum_{\forall b,(p\wedge i)\subseteq b} w'_b(x) = \sum_{j=1}^{k} j\ \varphi'_{p,j}(x) = Z_p(x) \qquad (3)$$

because each j^{th} order Walsh coefficient appears j times (e.g., w'_{p+q+r} is counted when $i = p, i = q$, and $i = r$).

$$\begin{aligned}
A &= Avg(N(y_p)) \\
&= f(y_p) - 2/N \sum_{\mathbf{i}=1}^{N} (S_i(y_p)) \qquad \text{by Eqn 1} \\
&= f(y_p) - \frac{2}{N} \sum_{\mathbf{i}=1}^{N} [S_i(x) - 2[\sum_{\forall b,(p\wedge i)\subseteq b} w'_b(x)]] \quad \text{by Eqn 2} \\
&= [f(y_p)] - \frac{2}{N} [\sum_{\mathbf{i}=1}^{N} S_i(x)] + \frac{4}{N} [\sum_{j=1}^{k} j\ \varphi'_{p,j}(x)] \quad \text{by Eqn 3} \\
&= [f(x) - 2S_p(x)] - \frac{2}{N} [\sum_{\mathbf{i}=1}^{N} S_i(x)] + \frac{4}{N} [\sum_{j=1}^{k} j\ \varphi'_{p,j}(x)] \\
&= [f(x) - 2/N \sum_{\mathbf{i}=1}^{N} S_i(x)] - 2S_p(x) + \frac{4}{N} [\sum_{j=1}^{k} j\ \varphi'_{p,j}(x)] \\
&= Avg(N(x)) - 2S_p(x) + \frac{4}{N} [\sum_{j=1}^{k} j\ \varphi'_{p,j}(x)] \quad \text{by Eqn 1} \\
&= Avg(N(x)) - 2S_p(x) + \frac{4}{N} Z_p(x)
\end{aligned}$$

□

Theorem 2.

For all k-bounded pseudo-Boolean functions defined over $O(N)$ subfunctions, the modified of steepest descent using $Avg(N(y_i))$ as a surrogate gradient over all neighbors $y_i \in N(x)$ has $O(1)$ time per move on average.

Proof:

The vector S and Z are the same, except the computations in Z are weighted by $bitcount(b)$ where $p \wedge i \subseteq b$. Vector Z must be updated exactly when S is updated:

$$S_i(y_p) \equiv S_i(x) \iff Z_i(y_p) \equiv Z_i(x)$$

where "≡" denotes equivalence because no update has occurred. (By coincidence, values that have been updated can also be equal.)

Let $U_i(x) = -2S_i(x) + \frac{4}{N} Z_i(x)$. Assuming $bitcount(b) \leq k < N/2$ then $U_i(x)$ is negative; thus, maximizing $U(x)$ minimizes $Avg(N(y_i))$ just as maximizing $S(x)$ minimizes $f(y_i)$. Thus, the same modified steepest descent search methods used to search S can be used to search U. The only additional information that is needed is the order of the Walsh coefficient, which is obtain by table-lookup in $O(1)$ time. □

4. EMPIRICAL RESULTS

4.1 Using statistics to guide search

To demonstrate the utility of using summary statistics of neighborhoods, we replace the fitness function $f(x)$ for NK-Landscapes with $Avg(N(x))$ in three widely studied optimization algorithms: steepest descent local search with random restart (LSr), a simple genetic algorithm (GA), and the CHC algorithm [1].

LSr as implemented here is a strict steepest steepest descent algorithm; we do not use a Tabu mechanism and do not compensate for the number of improving moves in buffer B. When there is no improving move found in neighborhoods, LSr restarts from a random solution. The simple generational GA is implemented with: 1) one-point crossover at the rate of 0.8; 2) one-bit flip mutation at the rate of $1/N$; 3) tournament selection of size 2; 4) and population of size 50. The CHC algorithm in our experimental studies uses the recommended settings from the original literature [1]. The number of evaluations allocated to each algorithm is set to $1000 \times N$, or 1000 moves for Steepest Descent Local Search (LSr). We assume that $Avg(N(x))$ is obtained by one evaluation.

4.1.1 NK-Landscapes

The first question that we would like to investigate is this, "Is there an advantage to using $Avg(N(x))$ to guide search compared to the fitness function $f(x)$?" Table 2 displays the results for NK-landscapes. The mean and standard deviation over 30 independent runs are presented. We only did statistical tests for $N = 100$ and a signicantly better solution is marked with "⋆" if the difference is statistically significant according to Wilcoxon rank-sum test with 5% significance level. The better mean within the same algorithm category is typeset in **bold** font.

Using $Avg(N(x))$ leads both LSr and the GA to a better solution in most cases, compared to using the standard fitness function. Using $Avg(N(x))$ accesses the potential of candidate solution for search algorithms whose neighborhood operator includes a single bit flip, as is the case for LSr and the GA. Statistical tests indicate the improvements are not statistically significant on regular NK-landscapes, but the differences usually are significant on NKq-landscapes. Using $Avg(N(x))$ to guide search is more likely to help search process when there are numerous plateaus.

On the other hand, $Avg(N(x))$ always statistically significantly degrades CHC's search ability. This is not surprising. The Hamming distance 1 neighborhood is not part of the neighborhood space explored by CHC. As a consequence, the average evaluation of all solutions which are one bit different from current solution is not useful information to the CHC algorithm.

We can discount the results when $N = 20$ because LSr almost always finds the global optimum.

On regular NK-landscapes, the CHC algorithm yields the best result most of the time. However, CHC is more costly than brute force steepest descent, and far slower than fast LSr using the Walsh decomposition. In expectation the fast LSr algorithm looks at $k(k-1)$ neighbors per move. For $K = 4$ and $N = 500$, this means that the fast LSr algorithm is evaluating approximately 20 partial updates for every 500 full evaluations used by CHC. To be fair, more evaluations could be allocated to the fast LSr algorithm.

4.1.2 NKq-Landscapes with q=2

NKq-Landscapes [2] are almost identical to NK-Landscapes except that subfunctions are lookup tables from uniform distribution of integers $[0, q)$, where q is the discretization level. The smaller q is, the less variance appears in the values of subfunctions. Hence there are more plateaus when q is small. In order to generate very

N	Algorithm	Eval	K=2		K=4		K=8	
			mean	std.	mean	std.	mean	std.
20	LSr	f(x)	0.277	0.0	0.212	0.0	0.219	0.003
		Avg(N(x))	0.277	0.0	0.212	0.0	0.219	0.005
	GA	f(x)	0.296	0.0123	0.260	0.0202	0.286	0.028
		Avg(N(x))	0.294	0.0092	0.256	0.0214	0.274	0.026
	CHC	f(x)	0.28	0.006	0.241	0.011	0.239	0.020
		Avg(N(x))	0.30	0.008	0.255	0.011	0.258	0.014
50	LSr	f(x)	0.265	0.004	0.238	0.007	0.239	0.008
		Avg(N(x))	0.264	0.003	0.237	0.007	0.238	0.007
	GA	f(x)	0.288	0.012	0.28	0.022	0.283	0.027
		Avg(N(x))	0.288	0.014	0.28	0.025	0.281	0.018
	CHC	f(x)	0.263	0.0036	0.237	0.007	0.30	0.018
		Avg(N(x))	0.362	0.0148	0.347	0.016	0.33	0.013
100	LSr	f(x)	0.277	0.0038	**0.251**	0.006	0.255	0.008
		Avg(N(x))	0.277	0.0038	0.252	0.007	**0.252**	0.008
	GA	f(x)	0.295	0.0096	0.280	0.0018	0.290	0.0178
		Avg(N(x))	0.295	0.0086	**0.278**	0.0012	**0.284**	0.016
	CHC	f(x)	⋆**0.273**	0.004	⋆**0.242**	0.007	⋆**0.302**	0.052
		Avg(N(x))	0.405	0.010	0.398	0.014	0.388	0.011

Table 2: Best solutions found on unrestricted NK-landscapes as minimization problems. The main entries of table are the means over 30 runs for specific settings, and standard deviations are colored in gray. The "Eval" column indicates which evaluation function is used to guide search, where $f(x)$ is the original evaluation function and Avg(N(x)) is the surrogate fitness function.

Landscapes	Eval	K=2	K=4	K=8
NK	f(x)	25	459	3676
	Avg(N(x))	15	306	1944
NK-q (q=2)	f(x)	2465	3802	13018
	Avg(N(x))	65	253	2253

Table 3: The number of local optimum, when $N = 20$. Whenever there is no improving move in neighborhood of a candidate solution, we count it as local optimum, thus plateaus are included as well.

different NK-Landscapes and NKq-Landscapes, we choose $q = 2$ for our empirical studies.

Table 3 shows that number of local optimum increases remarkably when the landscape is switched from NK to NKq (q=2). However, using $Avg(N(x))$ as the surrogate fitness function results in fewer plateaus. Even when 2 neighbors have the same evaluation, the averages of their neighborhoods could be different.

Table 4 displays results. The same search methods were used for both NKq-Landscapes and NK-landscapes. Both LSr and GA with $Avg(N(x))$ as the fitness function perform better with statistical significance, or is at least comparable on *all* cases compared to using the standard evaluation function. The advantage of using $Avg(N(x))$ as a surrogate fitness function is clearer for NKq-Landscapes. Using $Avg(N(x))$ as a surrogate fitness function helps to guide the search and reduce the number of local optima when there are a large number of plateaus.

For CHC, using the original evaluation function is significantly better on all cases. Again, the traditional notion of "neighborhood" does not appear to be relevant to the CHC algorithm.

Both Table 2 and Table 4 illustrates the helpfulness of incorporating summary statistics of neighborhoods into single-bit-flip based search algorithms. The LSr algorithm now yields the best results in most cases, while still doing dramatically less work. Of course, it only makes sense to use $Avg(N(x))$ as a surrogate evaluation function with the fast LSr algorithm. Only a local search algorithm that flips one bit at at time can exploit the fast partial Walsh updates to the vector Z. Neighborhood averages can be computed very efficiently using the Walsh decomposition for the Hamming distance 2 averages; it can be faster than brute force even when $2^k > N$.

4.2 Accelerating Steepest Descent

When implementing our fast Walsh steepest descent algorithm we ignored the Tabu mechanism and we allowed the buffer of improving moves to be unrestricted. While this could degrade the desired $O(1)$ performance, empirically there was little degradation of performance. Empirically the Tabu mechanism was not required, and typically the number of improving moves is small.

The overall algorithm can be outlined as follow:

1. The first move requires that the vector S be constructed, and all N elements in the S vector are searched to find improving moves. These improving moves are placed in the buffer B so that a strict steepest descent local search can be done. This is a one time cost for each restart.

2. Find the p^{th} neighbor in the buffer which is the next steepest descent move. The p^{th} bit is flipped. Elements in the S vector (and Z vector) as well as improving moves in buffer B whose bits interacting with the p^{th} bit will potentially change, and must be updated. The w vector is also updated.

3. If buffer B is empty, execute a random restart and then goto Step 1.

4. If termination condition is not met, go to Step 2.

In this way, brute force steepest descent and the fast Walsh steepest descent behaves exactly the same; this allowed us to cross validate results. Let v denoted the average cost of updating S, Z and w'. The maximum number of evaluations is set to $1000 \times N$ under brute force steepest descent, and there are 1000 moves. For the fast method using the Walsh decomposition, the total time spend on updates is $1000v$ for 1000 moves.

N	Algorithm	Eval	K=2		K=4		K=8	
			mean	std.	mean	std.	mean	std.
20	LSr	f(x)	0.05	0.0	0.05	0.0	0.0083	0.023
		Avg(N(x))	0.05	0.0	0.05	0.0	0.0	0.0
	GA	f(x)	0.078	0.033	0.086	0.031	0.11	0.05
		Avg(N(x))	0.06	0.02	0.088	0.03	0.126	0.044
	CHC	f(x)	0.052	0.009	0.052	0.009	0.073	0.031
		Avg(N(x))	0.093	0.021	0.093	0.017	0.098	0.016
50	LSr	f(x)	0.146	0.01	0.058	0.014	0.076	0.02
		Avg(N(x))	0.129	0.01	0.025	0.009	0.059	0.015
	GA	f(x)	0.0172	0.022	0.11	0.0366	0.14	0.039
		Avg(N(x))	0.168	0.02	0.09	0.03	0.0127	0.029
	CHC	f(x)	0.142	0.013	0.028	0.0099	0.165	0.033
		Avg(N(x))	0.267	0.0239	0.2227	0.0235	0.212	0.02
100	LSr	f(x)	0.1543	0.0117	0.098	0.0142	0.1037	0.01
		Avg(N(x))	⋆**0.128**	0.0075	⋆**0.056**	0.0079	⋆**0.077**	0.009
	GA	f(x)	0.188	0.0269	0.137	0.025	0.146	0.024
		Avg(N(x))	⋆**0.156**	0.0135	⋆**0.094**	0.0196	⋆**0.123**	0.0215
	CHC	f(x)	⋆**0.137**	0.01	⋆**0.053**	0.0124	⋆**0.175**	0.0682
		Avg(N(x))	.359	0.0148	0.317	0.018	0.292	0.018

Table 4: Best solutions found on unrestricted NKq-landscapes with $q = 2$ as minimization problems.

N	Method	Eval	K=2	K=4	K=8
20	brute force	f(x)	4.33	6.15	8.59
	fast Walsh		0.27	4.62	367
	brute force	Avg	42.0	56.8	85.7
	fast Walsh		0.513	8.93	681
50	brute force	f(x)	18.2	31.1	49.1
	fast Walsh		0.325	5.10	448
	brute force	Avg	591	792	1250
	fast Walsh		0.62	10.1	869
100	brute force	f(x)	53.3	93.7	145
	fast Walsh		0.363	5.49	466
	brute force	Avg	4510	6200	10200
	fast Walsh		0.677	10.9	925
200	fast Walsh	f(x)	0.450	5.92	482
		Avg	0.790	11.5	962
500	fast Walsh	f(x)	0.791	7.55	512
		Avg	1.210	13.6	1020

Table 5: Runtime comparison in terms of CPU time (seconds) for a single run of LSr under specific setting of NK-Landscapes. "Method" means the method of computation, either by brute force enumeration of all of the neighbors, or by using Walsh analysis to find which neighbors could be improving moves. Runtime of brute force approach for cases of $N > 100$ is not provided here. We assume that Walsh coefficients are available beforehand.

N	Eval	Stage	K=2	K=4	K=8
20	f(x)	initial	0.151	3.06	272
		update	0.123	1.56	94.4
	Avg(N(x))	initial	0.279	5.78	485
		update	0.233	3.16	195
50	f(x)	initial	0.186	3.35	332
		update	0.140	1.75	116
	Avg(N(x))	initial	0.357	6.54	634
		update	0.263	3.54	234
100	f(x)	initial	0.205	3.61	343
		update	0.159	1.88	123
	Avg(N(x))	initial	0.387	7.06	671
		update	0.290	3.83	254
200	f(x)	initial	0.259	3.94	354
		update	0.191	1.99	128
	Avg(N(x))	initial	0.458	7.42	690
		update	0.333	4.11	271
500	f(x)	initial	0.507	5.40	381
		update	0.284	2.15	131
	Avg(N(x))	initial	0.748	9.00	726
		update	0.462	4.57	298

Table 6: CPU time (seconds) consumed by initial one time cost and recurring update costs for the fast Walsh LSr.

Table 5 compares the runtime between LSr with the normal evaluation function $f(x)$ and the one using $Avg(N(x))$; runtime results are given for both brute force steepest descent and the fast Walsh steepest descent. All algorithms are implemented in Python. The runtime are measured under Cray Linux Environment with an AMD Opteron Processor 285 at 2.6GHz.

Table 5 shows the fast Walsh steepest descent dramatically accelerates LSr when K is small and N is large. A nearly 10,000 fold speedup is achieved for $N = 100$ and $K = 2$ when the surrogate fitness is used. The brute force method is faster when $K = 8$ and $2^{K+1} > N$. The results are consistent with the theory in Section 3.

Table 6 shows the initialization of data structures. This part of algorithm has time complexity $O(N)$. While the initialization only happens once, nevertheless, the $O(N)$ initial costs is always greater than the actual search (update) costs in every experiment. The growth of overall CPU time for LSr using Walsh analysis should be very slow. The update results in Table 5 display runtimes that are nearly constant for the reoccurring update costs. For example, as N increases from $N = 20$ to $N = 500$, for $K = 4$ the update cost increase from 1.56 seconds to 2.15 seconds for the standard evaluation function. Even the one time costs increase from only 3.06 to 5.4 seconds.

N	K=2	K=4	K=8
20	0.0047	0.082	18.4
50	0.0118	0.171	46.0
100	0.0258	0.409	92.5
200	0.0605	0.666	187
500	0.2161	1.88	468

Table 7: CPU time (seconds) for Computing Walsh Coefficients

If the steepest descent local search were allowed to run longer, the advantages of the fast Walsh update when k is small would be magnified.

Table 7 yields the computational cost for computing Walsh coefficients. We observe that the time required for the Walsh analysis when k is small is only a small portion (mostly less than 10% for $K = 2$ and $K = 4$) of the overall runtime listed in Table 5.

5. CONCLUSIONS AND DISCUSSION

This paper introduces a fast method of implementing Steepest Descent Local Search for k-bounded pseudo-Boolean functions. A specific implementation has been developed for NK-Landscapes and NKq-Landscapes, but the results equally hold for domains such as MAX-kSAT. This paper also introduced the use of the average of the neighborhood at Hamming distance 2 as a new surrogate fitness function. Using this surrogate fitness function generally improves the search behavior of Steepest Descent Local Search, particularly on NKq-Landscapes with plateaus.

We have proven that it is possible to implement a modified form of Steepest Descent Local Search where, on average, we only need to examine a constant number of neighbors in order to find the steepest descent improving move. The modifications include a Tabu mechanism, and an approximation to the best improving move under special conditions.

It should be pointed out that next descent local search could also be implemented in constant time as well. To obtain a proof of $O(1)$ average complexity for next descent, the Tabu mechanism would still be required. However, moves could just randomly be selected out of the improving move buffer in $O(1)$ time with no special processing.

Empirically we find that the Tabu mechanism is not needed for steepest descent. Also, approximations to the steepest descent move are only necessary if there are too many “old" improving moves waiting to be selected. In practice, this was not a problem, and true steepest descent was implemented.

There are many opportunities to continue this line of research. Instead of implementing local search with restarts, iterated local search could be used where fixed length random walks could be used to escape local optima; updates for a fixed length random walk would have an $O(1)$ complexity and avoid the $O(N)$ reinitialization costs of a random restart.

Another direct extension to the results presented in this paper makes is possible to identify the best improving move over the entire Hamming distance 2 neighborhood in $O(N)$ time, and to identify the neighborhood with the best average at Hamming distance 3. This can be done by expliciting searching one move ahead (which takes $O(N)$ time) and then applying the methods outlined in this paper (which takes $O(1)$ for each neighbor). This is a worst case result. Explicit amortization is not required since the Hamming distance 1 neighbors is fully explored and the cost is automatically amortized. The cost of picking the best move can also be absorbed in the $O(N)$ cost. Thus, this $O(N)$ result does not require any modifications to the Steepest Descent Local Search algorithm [9].

6. ACKNOWLEDGMENTS

This research was sponsored by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number FA9550-11-1-0088. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. This research utilized the CSU ISTeC Cray HPC System supported by NSF Grant CNS-0923386.

7. REFERENCES

[1] Larry J. Eshelman. The chc adaptive search algorithm: How to have safe search when engaging in nontraditional genetic recombination. In *FOGA'90*, pages 265–283, 1990.

[2] N Geard, J Wiles, J Hallinan, B Tonkes, and B Skellett. A comparison of neutral landscapes - nk, nkp and nkq. In D B Fogel, M A El-Sharkawi, X Yao, G Greenwood, H Iba, P Marrow, and M Shackleton, editors, *Congress on Evolutionary Computation (CEC2002)*, pages 205–210. IEEE Press, 2002.

[3] Robert Heckendorn and Darrell Whitley. A walsh analysis of NK-landscapes. In *Proceedings of the Seventh International Conference on Genetic Algorithms*, 1997.

[4] Robert B. Heckendorn. Embedded landscapes. *Evolutionary Computation*, 10(4):345–369, 2002.

[5] Stuart Kauffman and Simon Levin. Towards a general theory of adaptive walks on rugged landscapes. *Journal of Theoretical Biology*, 128:11–45, 1987.

[6] Soraya Rana, Robert B. Heckendorn, and L. Darrell Whitley. A tractable Walsh analysis of SAT and its implications for genetic algorithms. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI'98)*, pages 392–397, 1998.

[7] Andrew M. Sutton, Adele E. Howe, and L. Darrell Whitley. A theoretical analysis of the k-satisfiability search space. In *Proceedings of the Second International Workshop on Engineering Stochastic Local Search Algorithms. Designing, Implementing and Analyzing Effective Heuristics*, SLS '09, pages 46–60, Berlin, Heidelberg, 2009. Springer-Verlag.

[8] Andrew M. Sutton, L. Darrell Whitley, and Adele E. Howe. Computing the moments of -bounded pseudo-boolean functions over hamming spheres of arbitrary radius in polynomial time. *Theoretical Computer Science*, (0):–, 2011.

[9] Darrell Whitley. Defying Gravity: constant time steepest ascent for MAX-kSAT. *Technical Report, Colorado State University, December 2011.*