Constant time Steepest Ascent Local Search with Statistical Lookahead for NK-Landscapes

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 ascent local search is proposed that displays an average complexity of $O(1)$ time per move for NK-Landscape 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 ascent can be executed in $O(1)$ time using the Hamming distance 2 neighborhood average as the fitness function. A modified form of steepest ascent is used to prove $O(1)$ complexity, but in practice these modifications can be relaxed. Finally, steepest ascent local search over the mean of the Hamming distance 2 neighborhood yield 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

Fitness Landscapes, Elementary Landscapes

1. INTRODUCTION

NK-Landscapes, MAX-kSAT and some Ising problems are all important examples of the family of k-bounded pseudo-Boolean functions. In this paper we prove that a restricted form of steepest ascent can be constructed that selects the best 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, 2012, Philadelphia, PA

The result hold for all k-bounded pseudo-Boolean functions. 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 $N >> 2^3$.

The current study focuses on NK-Landscapes [5], as well as NKq-Landscapes [2]. NKq-Landscapes display more plateaus and result in more equally good improving moves compared to standard NK-Landscapes. Thus, NKq-Landscapes have more in common with MAXSAT problems than do other types of NK-Landscapes.

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. 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. For example, if the current solution is an extremum, then it is possible that *every neighbor is an improving move.* In this case it is necessary to examine every neighbor to find an improving move. In order to achieve an average $O(1)$ complexity we prove that when cost is amortized across all of the bit variables, the average cost of updating the evaluation function has $O(1)$ complexity.

This also makes it possible to identify a local optimum in $O(1)$ time because, again, on average only a constant number of neighbors can change their relative evaluation after a bit flip.

Empirical data shows the restrictions required for the proof are not necessarily in practice. Instead of the amortization of cost uniformly across all variables, the search can instead avoid flipping the same high cost variables too often. In practice, the high cost variables have a low rate of being flipped. Thus, our implementation uses a standard form of steepest ascent 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 also build on the theory of "Elementary Landscapes" to compute the average of neighborhoods 2 moves ahead in $O(1)$ time. Sutton et al. (2011) [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 precise information about averages over neighbors that are 2 moves away. Empirical tests show that local search over the mean of the neighborhoods 2 moves ahead yields superior results compared to using the actual evaluation function. This appears to be particularly true for NKq-Landscapes where plateaus and neutral moves are more likely to be encountered.

2. THE WALSH TRANSFORM

Let X denote the search space: the set of binary strings of length N. Each bit corresponds to a single Boolean variable in the NK-Landscapes instance. NK-Landscapes are constructed from N subfunctions, where each subfunction $\mathbf{i}$ is associated with the bit in position $\mathbf{i}$ and K other randomly chosen bits. Each subfunction is a pseudo-Boolean function of $k = K + 1$ bits, and the entire NK-Landscape is a k-bounded pseudo-Boolean function, where:

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

and f_j evaluates the j^{th} function for string x.

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:

$$w = \mathbf{W}f, \quad f = \mathbf{W}^{-1}w$$

where $\mathbf{W}$ is a $2^n \times 2^n$ matrix

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

Normally the Walsh transform requires that the entire search space be enumerated. However, Rana et al. [6] show for the MAX-kSAT problem that if a function is composed of subfunctions, each of which is a function over k Boolean variables, then the order of nonzero Walsh coefficients of the corresponding subfunction f_j is also bounded by 2^k. 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]. Thus, we can apply the Walsh transform to each subfunction in the NK-Landscape:

$$w = \sum_{j=1}^{m} \mathbf{W}f_j$$

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 the number of nonzero Walsh coefficients is at most $N * 2^k = O(N)$.

For $0 \leq i < 2^n$, the i^{th} Walsh function is 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 a single bit flip during local search can have only 2 possible effects. If the 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 stored 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 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-landscapes where $k = K+1 = 3$. 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 in positions 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:

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

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.

We assume the Walsh coefficients and their signs have been computed for some initial solution x, and will use b to index all of the Walsh coefficients in vector $w'(x)$. Let $p \subset b$ denote that bit $\mathbf{p}$ 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.

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

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

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

If $p \subset b$ 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 minimal sum $S_{min}(x)$ yields the steepest ascent move in the neighborhood of $f(x)$.

3. STEEPEST ASCENT

After bit p is flipped and the neighbor y_p 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.

$$S_q(y_p) \quad = \quad S_q(x) - 2 \sum_{\forall b,(p \wedge q) \subset b} w'_b(x) \qquad (1)$$

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.

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

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 $O(N)$ Start-Up 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 is clearly has $O(N)$ start-up 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. Thus, when search first begins, there maybe improving moves in the first neighborhood that are independent of one another that can be taken together.

If there is more than one improving move, we will select the "best" improving move, breaking ties arbitrarily. If the selected move is p then let $P = \{p\}$ be the "independent set." We then compute the closure of P; if q is also an improving move, and q does not appear in the same subfunction with any variable in P, then $q \in P$. Note that different ways of breaking ties will result in different independent sets. We will assume that the best move is always placed first in P. If there are ties for the best move, one is selected, and the other equal moves are places in P as long as independence is preserved. We then place the other improving moves in P (in any arbitrary order), again making sure independent is preserved. We can then flip all of the bits in the independent set, updating S as the bits are flipped.

We will define a **supermove** as a move that takes all of the moves in an independent set P in parallel, where $|P| > 1$. A supermove could be necessary to maintain $O(1)$ complexity if there are too many improving moves, such as when search starts at a local extremum. While this does not exactly correspond to steepest ascent local search, it is also not necessarily a bad thing. A supermove allows for rapid improvement in the evaluation function.

The vector S needs to be initialized after the first local search move. The first initialization of vector S will require $O(N)$ time. In addition the first move can be a **supermove** if there are multiple independent improving moves.

3.2 The O(1) average case analysis

To proof 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 practice, we will relax this assumption.

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 ascent with the Tabu mechanism and allowing **supermoves** when needed, the cost of updating the vector S to find the steepest ascent move can be amortized over N steps to yield an average number of dates involving at most $k(k-1)2^{k-2} = O(1)$ Walsh coefficients.

Proof:

At step one, a steepest ascent move p is selected; if there exists more than one independent improving move a **supermove** is taken. This is a one time cost.

After the initial **supermove** the vector S only contains 1) equal moves, 2) disimproving moves, 3) Tabu moves and 4) elements that have been updated: all non-independent improving moves are destroyed by the updates. Thus, the next improving move must be selected from the updated positions in S.

If more than one independent improving move is available among the updates, then a new "independent set" will exist. We will process these by placing them in a buffer B. B can be implemented as a heap, but since the buffer will be of (short) fixed length, the elements in B can be inserted or deleted in $O(1)$ time. However, if there are more than $|B|$ improving moves, a **supermove** must be taken to both exploit the improving moves and clear buffer B. After every move the vector S will again contain only 1) equal moves, 2) disimproving moves, 3) Tabu moves and 4) updated elements.

After N steps every bit has been flipped 1 time. All of the bits taken during a **supermove** count toward the number of bit flips. 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 nonlinear updates to S are given by:

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

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 N subfunctions there would be at most $k(k-1)N$ updates to S plus the N linear updates to $S_p(y_p)$. When this is amortized over all N bits, the average number of elements in S that must be updated is are at most

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

We must also account for each Walsh coefficient that changes sign and contribute 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 subfunctions is at most

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

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

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

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 that is $O(1)$.

3.3 Theory versus Practice

The proof deals with two issues: 1) balancing the number of times that high cost and low cost bit flips occur and 2) making sure there are not too many unexploited improving moves.

A bit variable will in expectation appear in k subfunctions. 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 have been 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 ascent 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 ascent.

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 Coefficients of numbers of interactive bits and numbers of bit-flips. Results are averaged over 100 independent runs for each parameter setting, and the relative standard deviations are displayed in parentheses.

This leaves the question of what to do about the buffer holding the independent improving moves until they can be executed. For purposes of the current experiments, we decided to not limit the size of the buffer. This has the advantage that our results will exactly match standard steepest ascent local search.

But by allowing the size of the buffer holding improving moves to be unrestricted, we could potentially destroy the $O(1)$ time complexity of the algorithm. Nevertheless, the unrestricted buffer does not appear to be a serious problem in practice. In practice the buffer is only "over the limit" occasionally at the beginning of search. As the search approaches a local optimum, the number of improving moves declines.

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

Recall that $N(x)$ generates the set of neighbors of solution x. Let $Avg(N(x))$ compute the average evaluation 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. (2009, 2011) [7, 8] derive the same basic result using principles from elementary landscapes.

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

$$\varphi'_{p,j}(x) = \sum_{\forall b, b^T b = j, p \subset b} w'_b(x)$$

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

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

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

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

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

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

Lemma 2.

For any k-bound pseudo-Boolean function:

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

Proof:

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

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

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

□

Theorem 2.

For NK-Landscapes the Tabu-restricted form of steepest ascent using $Avg(N(y_i))$ as a surrogate gradient over all neighbors $y_i \in N(x)$ has $O(1)$ time on average.

Proof:

The vector S and Z are the same, except the computations in Z are weighted by $Order(j)$. 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 $U_i = -2S_i(x) + \frac{4}{N} Z_i(x)$. It follows that maximizing U maximizes $Avg(N(y_i))$ just as minimizing $S(x)$ maximizes $f(y_i)$. We will use the same Tabu restricted modified steepest ascent search 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. On average, a move can be selected and Z updated in $O(1)$ time using exactly the same logic and data structures used to implement steepest ascent over $S(x)$. □

4. EMPIRICAL RESULTS

4.1 Using statistics to guiding search

To empirically demonstrate the utility of using summary statistics of neighborhoods, we replace the standard fitness function for NK-Landscapes with $Avg(N(x))$ in three widely studied combinatorial optimization algorithms: steepest ascent 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 ascent algorithm; we do not use a Tabu mechanism and do not use supermoves. 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 allowed for each algorithm is set to $1000 \times N$, assuming that the $Avg(N(x))$ can be obtained by one evaluation. In this way, one can focus on investigating the impact of utilizing summary statistics.

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 normal evaluation function?" Table 2 displays the results. The means as well as the standard deviations over 30 independent runs are presented. The better mean within the same algorithm category is typeset in **bold** font. In addition, the better one is marked with "⋆", if the difference is statistically significant according to Wilcoxon rank-sum test with 5% significance level.

One can observe that $Avg(N(x))$ actually leads both LSr and 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 manifest the improvements are not statistically significant however. This is due to the characteristics of NK-Landscapes. Each subfunction of NK-Landscapes is a lookup table from uniform distribution over [0,1), thus there are limited number of plateaus. $Avg(N(x))$ is more likely to help search process when there are numerous plateaus. We will discuss empirical validation of this hypothesis in subsequent set of experiments on NKq-Landscapes.

On the other hand, $Avg(N(x))$ always statistically significantly degrades CHC's search ability. This is not surprising though, as CHC employs a radical and highly disruptive neighborhood operator that aims to generate offspring that are maximally different from its parents. 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 (the neighborhood by our definition) is not useful information to the CHC algorithm.

Notice that using $Avg(N(x))$ as the fitness function does not enhance the performance of LSr when $N = 20$. However, LSr almost always finds the global optimum when $N = 20$.

Overall, the CHC algorithm yields the best result most of the time, but it is only slightly better than LSr. However, it should be stressed that CHC is more costly than brute force steepest ascent, and the execution time of CHC is at least 20 times slower than of the fast Walsh steepest ascent method when $N = 100$ and $K \leq 4$. If we want to beat CHC, we can let our fast Walsh steepest ascent algorithm run longer and do more restarts.

N	algo	Eval	K=2		K=4		K=8	
			mean	std.	mean	std.	mean	std.
20	LSr	f(x)	2.767e-01	0.000e+00	2.124e-01	0.000e+00	**2.189e-01**	2.992e-03
		Avg(N(x))	2.767e-01	0.000e+00	2.124e-01	0.000e+00	2.191e-01	4.668e-03
	GA	f(x)	2.963e-01	1.231e-02	2.604e-01	2.023e-02	2.856e-01	2.811e-02
		Avg(N(x))	**2.937e-01**	9.238e-03	**2.556e-01**	2.142e-02	⋆**2.739e-01**	2.613e-02
	CHC	f(x)	⋆**2.809e-01**	6.117e-03	⋆**2.406e-01**	1.059e-02	⋆**2.385e-01**	2.007e-02
		Avg(N(x))	2.998e-01	8.153e-03	2.548e-01	1.127e-02	2.579e-01	1.399e-02
50	LSr	f(x)	2.652e-01	3.619e-03	2.381e-01	7.149e-03	2.385e-01	8.145e-03
		Avg(N(x))	**2.642e-01**	3.436e-03	**2.371e-01**	6.873e-03	**2.379e-01**	7.296e-03
	GA	f(x)	2.880e-01	1.210e-02	2.818e-01	2.230e-02	2.831e-01	2.704e-02
		Avg(N(x))	**2.877e-01**	1.424e-02	**2.777e-01**	2.481e-02	**2.808e-01**	1.776e-02
	CHC	f(x)	⋆**2.629e-01**	3.571e-03	⋆**2.366e-01**	7.024e-03	⋆**3.025e-01**	1.838e-02
		Avg(N(x))	3.621e-01	1.479e-02	3.469e-01	1.597e-02	3.270e-01	1.301e-02
100	LSr	f(x)	2.772e-01	3.776e-03	**2.508e-01**	5.884e-03	2.550e-01	8.086e-03
		Avg(N(x))	**2.765e-01**	3.795e-03	2.519e-01	7.247e-03	**2.523e-01**	8.277e-03
	GA	f(x)	**2.946e-01**	9.620e-03	2.802e-01	1.787e-02	2.900e-01	1.769e-02
		Avg(N(x))	2.951e-01	8.566e-03	**2.781e-01**	1.181e-02	**2.836e-01**	1.618e-02
	CHC	f(x)	⋆**2.728e-01**	4.060e-03	⋆**2.422e-01**	6.838e-03	⋆**3.019e-01**	5.218e-02
		Avg(N(x))	4.054e-01	1.014e-02	3.977e-01	1.375e-02	3.882e-01	1.079e-02

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 guiding 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)	fit	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.

4.1.2 NKq-Landscapes with q=2

NKq-Landscapes [2] are almost identical with 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 likely to be more plateaus, if q is small. In order to distinguish NK-Landscapes and NKq-Landscapes, we choose $q = 2$ for 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 functions results in fewer plateaus. Even when 2 neighbors have the same evaluation, the averages of their neighborhoods could be different.

Table 4 displays results using the experimental methodology described in Subsubsection 4.1.1. 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 the fitness function is far clearer for NKq-Landscapes, as can be seen in Table 2. $Avg(N(x))$ helps the search much more when there are a large number of plateaus.

For CHC, using the original evaluation function is significantly better on all cases. This appears to be related to inherent property of the CHC algorithm. The disruptive crossover operator used by CHC generates offspring that are not neighbors of the current solution. Thus, the traditional notion of "neighborhood" does not appear to be relevant to the CHC algorithm.

Both Table 2 and Table 4 discloses the helpfulness of incorporating summary statistics of neighborhoods into single-bit-flip based search algorithms, especially for local search using steepest ascent. However, brute force method requires N^2 evaluations for taking one move in steepest ascent, which renders it impractical in most situations. However, the neighborhood averages can be computed very efficiently using the Walsh decomposition, assuming that $N >> 2^k$.

4.2 Accelerating Steepest Ascent

Lemma 2 in Section 3 shows Walsh analysis for steepest ascent requires $O(N)$ time in worst case. And when a modified Tabu scheme is introduced to constrain the pattern of bit flips, $O(1)$ complexity can be achieved on average.

When implementing our fast Walsh steepest ascent 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. We do absorb a few $O(N)$ costs associated with the buffer of improving moves.

The overall algorithm is sketched out as follow:

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

2. Suppose the pth neighbor is considered as the next best move and flips the pth bit of solution x; all elements in S vector whose bits interacting with the pth bit will potentially change, and should be rechecked. Improving moves are updated in buffer B.

3. Only elements in buffer B are examined to find the next

N	algo	Eval	K=2		K=4		K=8	
			mean	std.	mean	std.	mean	std.
20	LSr	f(x)	5.000e-02	0.000e+00	5.000e-02	0.000e+00	8.333e-03	2.267e-02
		Avg(N(x))	5.000e-02	0.000e+00	5.000e-02	0.000e+00	⋆**0.000e+00**	0.000e+00
	GA	f(x)	7.833e-02	3.337e-02	**8.667e-02**	3.145e-02	**1.150e-01**	5.025e-02
		Avg(N(x))	⋆**6.000e-02**	2.000e-02	8.833e-02	3.078e-02	1.267e-01	4.422e-02
	CHC	f(x)	⋆**5.167e-02**	8.975e-03	⋆**5.167e-02**	8.975e-03	⋆**7.333e-02**	3.091e-02
		Avg(N(x))	9.333e-02	2.134e-02	9.333e-02	1.700e-02	9.833e-02	1.572e-02
50	LSr	f(x)	1.460e-01	1.052e-02	5.800e-02	1.400e-02	7.667e-02	2.006e-02
		Avg(N(x))	⋆**1.293e-01**	9.978e-03	⋆**2.533e-02**	8.844e-03	⋆**5.867e-02**	1.454e-02
	GA	f(x)	1.727e-01	2.159e-02	1.067e-01	3.664e-02	1.427e-01	3.958e-02
		Avg(N(x))	**1.687e-01**	2.045e-02	⋆**9.000e-02**	3.000e-02	**1.273e-01**	2.988e-02
	CHC	f(x)	⋆**1.420e-01**	1.301e-02	⋆**2.867e-02**	9.911e-03	⋆**1.653e-01**	3.304e-02
		Avg(N(x))	2.673e-01	2.394e-02	2.227e-01	2.351e-02	2.127e-01	2.159e-02
100	LSr	f(x)	1.543e-01	1.174e-02	9.800e-02	1.424e-02	1.037e-01	1.016e-02
		Avg(N(x))	⋆**1.280e-01**	7.483e-03	⋆**5.633e-02**	7.951e-03	⋆**7.733e-02**	9.286e-03
	GA	f(x)	1.877e-01	2.692e-02	1.373e-01	2.516e-02	1.463e-01	2.429e-02
		Avg(N(x))	⋆**1.563e-01**	1.354e-02	⋆**9.400e-02**	1.960e-02	⋆**1.233e-01**	2.150e-02
	CHC	f(x)	⋆**1.373e-01**	1.031e-02	⋆**5.300e-02**	1.242e-02	⋆**1.750e-01**	6.811e-02
		Avg(N(x))	3.587e-01	1.477e-02	3.167e-01	1.832e-02	2.920e-01	1.833e-02

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

steepest ascent move.

4. If termination condition is not met, go to Step 2 until buffer B is empty.

5. Execute a random restart, and then go to Step 1.

In this way, brute force steepest ascent and the fast Walsh steepest ascent behaves exactly the same during the whole course of search, thus the same results could be guaranteed.

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

Table 5 shows the fast Walsh steepest ascent dramatically accelerates LSr, especially when K is small and N is large. A nearly *10000x* speedup is achieved for $N = 100$ and $K = 2$ when the surrogate fitness is used. The results are consistent with the theory in Section 3 in several aspects:

The updates to both S and Z vector take O(1) time on average. LSr requires N fitness evaluation for taking one move, and each move requires a round of updates to $S(x)$, $Z(x)$ and $w'(x)$. According to the foregoing theory, one round of updates takes $O(1)$ time, which is denoted as c. Remember that the maximum number of evaluations is set to $1000 \times N$, thus there are 1000 moves for a steepest ascent algorithm to take and the total time spend on updates is $1000c$.

Table 6 shows the initialization of data structures. This is main part of algorithm that will grow at the order $O(N)$. Since the initialization only happens once, it does not play a key role in the overall computation, when N is small. The growth of overall CPU time for LSr using Walsh analysis should be very slow. Results associated with Walsh analysis in Table 5 display runtimes that are nearly constant for the reoccurring update costs. Operations to the unrestricted buffer should be the only factor that would occasionally cause the update work to expand as N increases, but this does not appear to be a significant factor. As N increases from $N = 20$ to $N = 500$, for $K = 4$ the update cost increase from 1.56 second

N	Method	Eval	K=2	K=4	K=8
20	brute force	f(x)	4.33e+00	6.15e+00	8.59e+00
	fast Walsh		2.74e-01	4.62e+00	3.67e+02
	brute force	Avg	4.20e+01	5.68e+01	8.57e+01
	fast Walsh		5.13e-01	8.93e+00	6.81e+02
50	brute force	f(x)	1.82e+01	3.11e+01	4.91e+01
	fast Walsh		3.25e-01	5.10e+00	4.48e+02
	brute force	Avg	5.91e+02	7.92e+02	1.25e+03
	fast Walsh		6.20e-01	1.01e+01	8.69e+02
100	brute force	f(x)	5.33e+01	9.37e+01	1.45e+02
	fast Walsh		3.63e-01	5.49e+00	4.66e+02
	brute force	Avg	4.51e+03	6.20e+03	1.02e+04
	fast Walsh		6.77e-01	1.09e+01	9.25e+02
200	fast Walsh	f(x)	4.50e-01	5.92e+00	4.82e+02
		Avg	7.90e-01	1.15e+01	9.62e+02
500	fast Walsh	f(x)	7.91e-01	7.55e+00	5.12e+02
		Avg	1.21e+00	1.36e+01	1.02e+03

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 assumes that Walsh coefficients are available beforehand.

N	fitness	stage	K=2	K=4	K=8
20	$f(x)$	initial	1.51e-01	3.06e+00	2.72e+02
		update	1.23e-01	1.56e+00	9.44e+01
	$Avg(N(x))$	initial	2.79e-01	5.78e+00	4.85e+02
		update	2.33e-01	3.16e+00	1.95e+02
50	f(x)	initial	1.86e-01	3.35e+00	3.32e+02
		update	1.40e-01	1.75e+00	1.16e+02
	Avg(N(x))	initial	3.57e-01	6.54e+00	6.34e+02
		update	2.63e-01	3.54e+00	2.34e+02
100	f(x)	initial	2.05e-01	3.61e+00	3.43e+02
		update	1.59e-01	1.88e+00	1.23e+02
	Avg(N(x))	initial	3.87e-01	7.06e+00	6.71e+02
		update	2.90e-01	3.83e+00	2.54e+02
200	f(x)	initial	2.59e-01	3.94e+00	3.54e+02
		update	1.91e-01	1.99e+00	1.28e+02
	Avg(N(x))	initial	4.58e-01	7.42e+00	6.90e+02
		update	3.33e-01	4.11e+00	2.71e+02
500	f(x)	initial	5.07e-01	5.40e+00	3.81e+02
		update	2.84e-01	2.15e+00	1.31e+02
	Avg(N(x))	initial	7.48e-01	9.00e+00	7.26e+02
		update	4.62e-01	4.57e+00	2.98e+02

Table 6: CPU time (seconds) consumed by initial one time cost and recurring update costs.

N	K=2	K=4	K=8
20	4.75e-03	8.17e-02	1.84e+01
50	1.18e-02	1.71e-01	4.60e+01
100	2.58e-02	4.09e-01	9.25e+01
200	6.05e-02	6.66e-01	1.87e+02
500	2.16e-01	1.88e+00	4.68e+02

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

to 2.15 second for the standard evaluation function. Even the one time costs increase from only 3.06 to 5.4 seconds.

The table also shows however, that the costs are very different when $2^k > N$. When $k = 8$ the Walsh analysis is more costly than brute force search, but this is not surprising.

One might also wonder about the cost needed to computing the Walsh coefficients, as they are necessary for Walsh analysis. Table 7 yields the computational cost for computing Walsh coefficients in reality. We observe that time requires for computing Walsh analysis is only small portion (mostly less than 10%) of the overall runtime listed in Table 5, thus this should not be our major concern.

5. CONCLUSIONS

This paper introduces a fast method of implementing Steepest Ascent 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 Ascent Local Search, particularly on NKq-Landscapes with plateaus.

We have proven that it is possible to implement a restricted form of Steepest Ascent Local Search where, on average, we only need to examine a constant number of neighbors in order to find the steepest ascent improving move. This restrictions include a Tabu mechanism, and the use of **supermoves** to avoid the accumulation of improving moves.

We have also shown empirically that the Tabu mechanism is not needed. **Supermoves** are only necessary is there are too many (i.e., $O(N)$) improving moves waiting to be selected. In practice this is usually not a problem. Thus, these restrictions were not implemented.

There are many opportunities to continue to improve this line of research. While we do not exploit **supermoves** in the current study, there seems to be no good reason not to exploit **supermoves** as often as possible. To achieve true $O(1)$ complexity we could use a form of soft restarts instead of random restarts. Fixed length random walks could be used to escape local optima; updates for a fixed length random walk would preserve the $O(1)$ complexity and avoid the $O(N)$ random restart costs.

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.