Efficient Identification of Improving Moves in a Ball for Pseudo-Boolean Problems

Francisco Chicano
Dept. de Lenguajes y Ciencias de la Computación
Universidad de Málaga, Andalucía Tech, Spain
chicano@lcc.uma.es

Darrell Whitley
Dept. of Computer Science
Colorado State University
Fort Collins CO, USA
whitley@cs.colostate.edu

Andrew M. Sutton
Fakultät für Mathematik und Informatik
Friedrich-Schiller-Universität Jena
07743 Jena, Germany
andrew.sutton@uni-jena.de

ABSTRACT

Hill climbing algorithms are at the core of many approaches to solve optimization problems. Such algorithms usually require the complete enumeration of a neighborhood of the current solution. In the case of problems defined over binary strings of length n, we define the r-ball neighborhood as the set of solutions at Hamming distance r or less from the current solution. For $r \ll n$ this neighborhood contains $\Theta(n^r)$ solutions. In this paper efficient methods are introduced to locate improving moves in the r-ball neighborhood for problems that can be written as a sum of a linear number of subfunctions depending on a bounded number of variables. NK-landscapes and MAX-kSAT are examples of these problems. If the number of subfunctions depending on any given variable is also bounded, then we prove that the method can explore the neighborhood in constant time, despite the fact that the number of solutions in the neighborhood is polynomial in n. We develop a hill climber based on our exploration method and we analyze its efficiency and efficacy using experiments with NKq-landscapes instances.

Categories and Subject Descriptors

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

General Terms

Theory, Algorithms

Keywords

Hill Climbing, Local Search, NK-landscapes, Pseudo-Boolean Optimization

1. INTRODUCTION

Local search algorithms work by starting at an initial solution and then moving from one solution to a "neighboring" solution in the search space. The set of solutions that can be reached in one move from solution x is called the *neighborhood* of x, denoted with $N(x)$. Most modern forms of local search often use some form of Iterated Local Search [4]. Thus, any advance in local search algorithms can have an important impact in the solution of the problem.

A simple implementation of local search requires enumerating the entire neighborhood of the current solution. In *steepest ascent* local search, the algorithm moves to a solution providing the largest improvement of the objective function; in *next ascent* search moves to the first improving solution it finds in the neighborhood. For problems defined over a binary search space, the most common neighborhood is the "bit-flip" Hamming distance 1 neighborhood. Given inputs of binary strings of length n, the size of this neighborhood is n, and exactly n neighbors are examined to locate an improving move.

In the current paper, we introduce a next ascent local search algorithm that is capable of identifying improving moves within r steps for k-bounded Pseudo-Boolean optimization problems. In this case, the neighborhood is a radius r Hamming ball: the set of all solutions at most Hamming distance r from the current solution. The size of this neighborhood is $O(n^r)$. For specific distributions of variables, we prove that all new improving moves can be identified in $\Theta(1)$ time using $\Theta(n)$ space per move.

Whitley and Chen [8] proposed a steepest descent algorithm for NK-landscapes and MAX-kSAT with an average time complexity of $\Theta(1)$ per move for the Hamming distance 1 neighborhood. Their algorithm works n times faster than a naïve implementation of the neighborhood exploration. This result is expressed as average time complexity because the time required in one single move is not bounded by a constant but over m moves the time required for *next ascent* and for approximate *steepest ascent* is bounded by $O(m)$. Their algorithm is based on the Walsh decomposition of the objective function. The secret of the resulting speedup is based on the fact that on average a bit-flip affects only a constant number of terms in the Walsh decomposition of the objective function. For all k-bounded pseudo-Boolean functions, the time complexity of their steepest descent is $O(k^2 2^k)$. For MAX-kSAT the value of k is the number of literals per clause while for NK-landscapes $k = K + 1$.

In a later paper, Chen *et al.* [1] used the concept of second derivative of a pseudo-Boolean function to improve the computation time of the steepest descent for NK-landscapes for the Hamming distance 1 neighborhood. Using a second

GECCO'14, July 12–16, 2014, Vancouver, BC, Canada.

ACM 978-1-4503-2662-9/14/07 ...$15.00.
http://dx.doi.org/10.1145/2576768.2598304

derivative, they reduce the time needed to identify improving moves from $O(k^2 2^k)$ to $O(k^3)$. In addition, the new approach avoids the use of the Walsh transform, making the approach conceptually simpler.

In this paper, we generalize this result to present a local search algorithm that can look r moves ahead and identify all improving moves. This means that moves are being identified in a neighborhood containing all solutions that lie within a *Hamming ball* of radius r around the current solution. We assume that $r = O(1)$. If $r \ll n$, the number of solutions in such a neighborhood is $\Theta(n^r)$. New improving moves located up to r moves away can be identified in constant time. The memory required by our approach is $O(n)$. To achieve $O(1)$ time per move, the number of subfunctions in which any variable appears must be bounded by some constant c. We then prove that the resulting algorithm requires $O((3kc)^r n)$ space to track potential moves.

In order to evaluate our approach we perform an experimental study based on NKq-landscapes. The results reveal not only that the time required by the next ascent is independent of n, but also that increasing r we obtain a significant gain in the quality of the solutions found.

The rest of the paper is organized as follows. In the next section we introduce the pseudo-Boolean optimization problems. Section 3 defines the "Scores" of a solution and provide an algorithm to efficiently update them during a local search algorithm. We propose in Section 4 a next ascent hill climber with the ability to identify improving moves in a ball of radius r in constant time. Section 5 empirically analyzes this hill climber using NKq-landscapes instances and Section 6 outlines some conclusions and future work.

2. PSEUDO-BOOLEAN OPTIMIZATION

Our method for identifying improving moves in the radius r Hamming ball can be applied to all k-bounded pseudo-Boolean Optimization problems. This makes our method quite general: every compressible pseudo-Boolean Optimization problem can be transformed into a quadratic pseudo-Boolean Optimization problem with $k = 2$.

The family of k-bounded pseudo-Boolean Optimization problems have also been described as an *embedded landscape*. An *embedded landscape* [3] with *bounded epistasis* k is defined as a function $f(x)$ that can be written as the sum of m subfunctions, each one depending at most on k input variables. That is:

$$f(x) = \sum_{i=1}^{m} f^{(i)}(x), \tag{1}$$

where the subfunctions $f^{(i)}$ depend only on k components of x. Embedded Landscapes generalize NK-landscapes and the MAX-kSAT problem. We will consider in this paper that the number of subfunctions is linear in n, that is $m \in O(n)$. For NK-landscapes $m = n$ and is a common assumption in MAX-kSAT that $m \in O(n)$.

3. SCORES IN THE HAMMING BALL

For $v, x \in \mathbb{B}^n$, and a pseudo-Boolean function $f : \mathbb{B}^n \to \mathbb{R}$, we denote the *Score* of x with respect to *move* v as $S_v(x)$, defined as follows:[1]

$$S_v(x) = f(x \oplus v) - f(x), \tag{2}$$

[1]We omit the function f in $S_v(x)$ to simplify the notation.

where $\oplus$ denotes the exclusive OR bitwise operation. The Score $S_v(x)$ is the change in the objective function when we move from solution x to solution $x \oplus v$, that is obtained by flipping in x all the bits that are 1 in v.

All possible Scores for strings v with $|v| \leq r$ must be stored as a vector. The Score vector is updated as local search moves from one solution to another. This makes it possible to know where the improving moves are in a ball of radius r around the current solution. For next ascent, all of the improving moves can be buffered. An approximate form of steepest ascent local search can be implemented using multiple buffers [9].

If we naively use equation (2) to explicitly update this Score vector, we will have to evaluate all $\sum_{i=0}^{r} \binom{n}{i}$ neighbors in the Hamming ball. Instead, if the objective function satisfies some requirements described below, we can design an efficient next ascent hill climber for the radius r neighborhood that only stores a linear number of Score values and requires a constant time to update them. We next explain the theoretical foundations of this next ascent hill climber.

The first requirement for the objective function is that it must be written such that each subfunction depends only on k Boolean variables of x (k-bounded epistasis). In this case, we can write the scoring function $S_v(x)$ as an embedded landscape:

$$S_v(x) = \sum_{l=1}^{m} \left(f^{(l)}(x \oplus v) - f^{(l)}(x) \right) = \sum_{l=1}^{m} S_v^{(l)}(x), \tag{3}$$

where we use $S_v^{(l)}$ to represent the scoring functions of the subfunctions $f^{(l)}$. Let us define $w_l \in \mathbb{B}^n$ as the binary string such that the i-th element of w_l is 1 if and only if $f^{(l)}$ depends on variable x_i. The vector w_l can be considered as a mask that characterizes the variables that affect $f^{(l)}$. Since $f^{(l)}$ has bounded epistasis k, the number of ones in w_l, denoted with $|w_l|$, is at most k. By the definition of w_l, the next equalities immediately follow.

$$f^{(l)}(x \oplus v) = f^{(l)}(x) \quad \text{for all } v \in \mathbb{B}^n \text{ with } v \wedge w_l = 0, \tag{4}$$

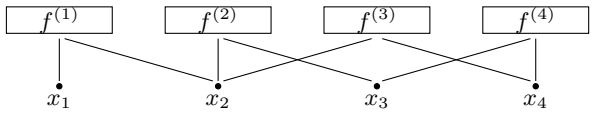
$$S_v^{(l)}(x) = \begin{cases} 0 & \text{if } w_l \wedge v = 0, \\ S_{v \wedge w_l}^{(l)}(x) & \text{otherwise.} \end{cases} \tag{5}$$

Equation (5) claims that if none of the variables that change in the move characterized by v is an argument of $f^{(l)}$ the Score of this subfunction is zero, since the value of this subfunction will not change from $f^{(l)}(x)$ to $f^{(l)}(x \oplus v)$. On the other hand, if $f^{(l)}$ depends on variables that change, we only need to consider for the evaluation of $S_v^{(l)}(x)$ the changed variables that affect $f^{(l)}$. These variables are characterized by the mask vector $v \wedge w_l$. With the help of (5) we can write (3) as:

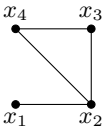
$$S_v(x) = \sum_{\substack{l=1 \\ w_l \wedge v \neq 0}}^{m} S_{v \wedge w_l}^{(l)}(x), \tag{6}$$

3.1 Scores Decomposition

The Score values in a ball of radius r give more information than just the change in the objective function for moves in that ball. Let us illustrate this idea with the moves in the balls of radius $r = 1$ and $r = 2$. Let us assume that x_i and x_j are two variables that do not appear together as arguments of any subfunction $f^{(l)}$. Then, the Score of the move



(a) Function



(b) Variable interaction graph

Figure 1: A function with $k = 2$ bounded epistasis, $n = 4$ variables and $m = 4$ subfunctions (top) and its corresponding variable interaction graph (bottom).

we obtain when we flip both variables is the sum of the Score values of the moves of each variable in isolation. To see this, let us denote with $\underline{i,j}$ the binary string with position i and j set to 1 and the rest set to 0. Similarly, we denote with $\underline{i}$ the binary string having 1 only in position i. According to (6) we can write:

$$S_{\underline{i,j}}(x) = \sum_{\substack{l=1 \\ w_l \wedge \underline{i,j} \neq 0}}^{m} S^{(l)}_{\underline{i,j} \wedge w_l}(x),$$

since x_i and x_j do not appear together in any subfunction, there is no w_l with 1s in both i and j positions, and we can write:

$$\begin{aligned} S_{\underline{i,j}}(x) &= \sum_{\substack{l=1 \\ w_l \wedge \underline{i} \neq 0}}^{m} S^{(l)}_{\underline{i} \wedge w_l}(x) + \sum_{\substack{l=1 \\ w_l \wedge \underline{j} \neq 0}}^{m} S^{(l)}_{\underline{j} \wedge w_l}(x) \\ &= S_{\underline{i}}(x) + S_{\underline{j}}(x), \end{aligned}$$

which is the claimed result. This means that if $\underline{i}$ and $\underline{j}$ are not improving moves, then $\underline{i,j}$ cannot be an improving move. Thus, the Scores in the 1-ball are also implicitly providing information of higher order moves in which the involved variables do not interact. In our example, we only need to track and update all the 2-ball Scores $\underline{i,j}$ for which i and j appear together in at least one subfunction $f^{(l)}$.

This reasoning can be generalized to higher order moves. In order to do this let us introduce the *variable interaction graph* $G = (V, E)$, where V is the set of Boolean variables and E contains all the pairs of variables (x_i, x_j) that interact each other, that is, $(x_i, x_j) \in E$ if there exists a subfunction $f^{(l)}$ that depends on x_i and x_j. In Figure 1 we show a function with k-bounded epistasis and its corresponding variable interaction graph.

Let us consider the Score of the move with characteristic vector v. Abusing notation, we use v to represent also the set of variables that will be flipped in the move. Let us denote with $G[v]$ the subgraph of G induced by v, that is, the subgraph containing only the vertices in v and the edges of E between vertices in v. If $G[v]$ is connected then we cannot compute the Score of the move as the sum of Scores of lower order moves. If the graph is not connected, then we can decompose the Score as a sum of at least two Scores of lower order moves. These necessary lower order moves are determined by the connected components of $G[v]$. For example, in the function of Figure 1 the scoring function $S_{\underline{1,3,4}}$ can be written as the sum of the scoring functions $S_{\underline{1}}$ and $S_{\underline{3,4}}$. If we want to know if there is at least one improving move in a ball of radius r, we have to consider the Scores of all the moves v with $|v| \leq r$ for which $G[v]$ is a connected graph. We will denote this set of moves with M^r to simplify the notation in the following. In the next subsection we will address the question of how many of these Scores exist and what is the cost in time of updating them after a move.

3.2 Scores Update

Instead of computing the Scores from scratch using (6) after every move, it is more efficient in time to store the Scores $S_v(x)$ of the current solution x in memory and update only those that are affected by the move. We will consider in the next paragraphs the cost in time of this update and we will prove that it is independent of n if the number of subfunctions in which any variable appears is bounded by some constant c. Under the same assumptions, we will prove that the number of Scores we need to store in memory is linear in n.

When we flip a set of bits t, only the value of some subfunctions $f^{(l)}$ will be affected: exactly those subfunctions for which $w_l \wedge t \neq 0$. If subfunction $f^{(l)}$ is affected, all the $S^{(l)}_v(x)$ values with $v \wedge w_l \neq 0$ could also change and, as a consequence, we have to update the S_v stored Scores for $v \wedge w_l \neq 0$. For each of these Scores, the update related to subfunction $f^{(l)}$ can consist in computing the value $S^{(l)}_v(x \oplus t) = f^{(l)}(x \oplus t \oplus v) - f^{(l)}(x \oplus t)$, computing the value $S^{(l)}_v(x) = f^{(l)}(x \oplus v) - f^{(l)}(x)$ and updating the stored value of S_v by subtracting $S^{(l)}_v(x)$ and adding $S^{(l)}_v(x \oplus t)$. This procedure is shown in Algorithm 1, where the term S_v represents the Score of move v stored in memory.

Algorithm 1 Efficient algorithm for Scores update

Input: S, x, t
1: **for** $l = 1$ to m such that $w_l \wedge t \neq 0$ **do**
2: **for** $v \in M^r$ such that $w_l \wedge v \neq 0$ **do**
3: $S_v \leftarrow S_v + f^{(l)}(x \oplus t \oplus v) - f^{(l)}(x \oplus t) - f^{(l)}(x \oplus v) + f^{(l)}(x)$
4: **end for**
5: **end for**

Let us prove that Algorithm 1 is correct. First, we can observe that in the inner loop we can make variable v to take values for which $w_l \wedge v = 0$, since in this case $f^{(l)}(x \oplus t \oplus v) = f^{(l)}(x \oplus t)$ and $f^{(l)}(x \oplus v) = f^{(l)}(x)$, what means that the right hand side in line 3 is reduced to S_v (no update). For the same reason, we can make variable l in the first loop to take all values from 1 to m, even those for which $w_l \wedge t = 0$. The reason why we use the conditions $w_l \wedge t \neq 0$ and $w_l \wedge v \neq 0$ in Algorithm 1 is efficiency: each condition reduces the computational complexity of the algorithm by a factor of n (this will be clear later). But efficiency is irrelevant in the proof of correctness, so we will assume that the conditions are not there. Under this assumption lines 1 and 2 can be swapped in the algorithm. The result is shown in Algorithm 2.

The outer loop of Algorithm 2 is just iterating over all the possible moves $v \in M^r$ to update all the Scores. The inner

Algorithm 2 Simple algorithm for Scores update

Input: S, x, t

1: **for** $v \in M^r$ **do**
2: **for** $l = 1$ to m **do**
3: $S_v \leftarrow S_v + f^{(l)}(x \oplus t \oplus v) - f^{(l)}(x \oplus t) - f^{(l)}(x \oplus v) + f^{(l)}(x)$
4: **end for**
5: **end for**

loop computes the new value for S_v using the expression:

$$\begin{aligned} S_v(x) + \sum_{l=1}^{m} \Big(& f^{(l)}(x \oplus t \oplus v) - f^{(l)}(x \oplus t) \\ & - f^{(l)}(x \oplus v) + f^{(l)}(x) \Big) \\ &= S_v(x) + \sum_{l=1}^{m} \Big(S_v^{(l)}(x \oplus t) - S_v^{(l)}(x) \Big) \\ &= S_v(x) + \sum_{l=1}^{m} S_v^{(l)}(x \oplus t) - \sum_{l=1}^{m} S_v^{(l)}(x) \\ &= S_v(x \oplus t), \end{aligned}$$

where we used (3). This proves that Algorithm 1 (and Algorithm 2) are correct, since its goal is to compute $S_v(x \oplus t)$ starting from $S_v(x)$.

In the update of line 3 of Algorithm 1 only subfunction $f^{(l)}$ is involved. The time required to evaluate $f^{(l)}$ must be some function of k, since $f^{(l)}$ depends only on k bits of the input string. Let us call this function $b(k)$. In the case of NK-landscapes and MAX-kSAT, $b(k) \in O(k)$ (we should remember that in NK-landscapes $k = K + 1$).

Let us now count how many times the body of the outer loop in Algorithm 1 is executed. We do this by counting how many subfunctions $f^{(l)}$ have the property $w_l \wedge t \neq 0$.

We first introduce an additional assumption: *each Boolean variable x_i appears in at most c subfunctions*. Under this assumption, the number of subfunctions containing at least one of the bits in t (condition $w_l \wedge t \neq 0$) is at most $c|t|$. Since $|t|$ is bounded by the constant r, the inner loop is run at most cr times, which is a constant. It is common to have $m \in O(n)$, so the difference between including the condition $w_l \wedge t \neq 0$ or not in the first loop is a factor of n, as we mentioned earlier.

Once the outer loop has fixed a value for l, we need to compute how many moves $v \in M^r$ fulfill $w_l \wedge v \neq 0$ to end with a final expression for the time required by Algorithm 1. There is a way to enumerate all these moves v that also give us a bound for its cardinality. Since $G[v]$ must be a connected graph, we can always find a spanning tree of the graph. If we build all the possible trees of subgraphs of G having at most r variables and with root in each variable that appears in $f^{(l)}$, we are sure that we considered all the possible $v \in M^r$ such that $w_l \wedge v \neq 0$. In effect, let us suppose that there is a $v \in M^r$ such that $w_l \wedge v \neq 0$. Then, there is a variable x_i in w_l that is included in v and we can build a spanning tree of $G[v]$ with root x_i.

How many sets v we will find this way? We can easily give an upper bound. Given a variable x_i, the number of variables that interact with x_i (adjacent nodes in the variable interaction graph) is bounded by ck. Given a tree of r nodes with x_i in the root, we have to assign variables to the rest of the nodes in such a way that two connected nodes have interacting variables. The ways in which we can do this is bounded by $O((ck)^{r-1})$. We have to repeat the same operation for all the possible rooted trees and each of the k variables in subfunction $f^{(l)}$. Thus, a bound for the $v \in M^r$ such that $w_l \wedge v \neq 0$ is $O(k(ck)^{r-1}T_r)$, where T_r is the number of rooted trees with r vertices.

The number of rooted trees of r vertices can be computed using the following recurrence [2]:

$$T_{r+1} = \frac{1}{r} \sum_{l=1}^{r} \left(\sum_{d|l} d\, T_d \right) T_{r-l+1}, \tag{7}$$

$$T_1 = 1. \tag{8}$$

This series has the following asymptotic behaviour [5]:

$$\lim_{r \to \infty} \frac{T_r}{T_{r-1}} \approx 2.955765, \tag{9}$$

Using this property, a bound for the number of moves $v \in M^r$ such that $w_l \wedge v \neq 0$ is $O(c^{r-1}(3k)^r)$. This process must be repeated for all the subfunctions affected by t, which we proved to be bounded by $c|t|$. Taking into account that the time to evaluate each Score is $O(b(k))$ we finally have the total time $O(b(k)(3kc)^r|t|)$ to update all the Scores S_v for $v \in M^r$ when we flip $|t|$ bits. This time is independent of n, provided that neither k, r nor c depends on n. This result was already proved by Szeider for the MAX-kSAT and SAT problems [6]. We generalize Szeider's result and design an algorithm based on it. To the best of our knowledge there is no previous local search algorithm based on this result.

The number of Scores S_v we have to store is the cardinality of M^r. We can find a bound for this cardinality using the following argument. If we flip all the n variables in turn we will update all the Scores S_v with $v \in M^r$ at least once. According to the previous paragraph the time required to update all the affected Scores after 1 bit flip is $O(b(k)(3kc)^r)$. Dividing by the cost of evaluating a subfunction $b(k)$ and multiplying by the n bits that we flip, we have that the number of moves in M^r is $O((3kc)^r n)$. This means $O(n)$ space to store the Scores. This is the reason why in line 2 of Algorithm 1 removing the condition $w_l \wedge v \neq 0$ means executing line 3 around $O(n)$ more times.

Two other complexity issues should be noted. There is an up-front one-time initialization cost. The Score vector must be initialized and this also has cost $O((3kc)^r n)$. We also need to distinguish between "old" improving moves and "new" improving moves. After any set of bit flips unto radius r there are only a constant number of changes to the Score vector. Therefore there are at most a constant number of "new" improving moves that are created as a result of the most recent move. However, there can be a significant number of "old" improving moves that have already been identified. Just as there are at most a constant number of "new" improving moves after any update of the Score vector, there are also at most a constant number of "old" improving moves that are destroyed by any update of the Score vector.

We end this section providing a theorem with the main result of the paper. The proof of this theorem has been given in the previous paragraphs.

THEOREM 1. *Let f be a function defined over $\mathbb{B}^n$ that can be written in the form (1) where each subfunction $f^{(l)}$*

depends only on at most k Boolean variables and each variable appears in at most c subfunctions. Let the graph G be the variable interaction graph of f and M^r the set of moves $v \in \mathbb{B}^r$ up to order r such that G[v] is a connected graph. Then:

- *The cardinality of M^r is $O((3kc)^r n)$.*
- *We only need to check the Scores $S_v(x)$ with $v \in M^r$ of the current solution x to determine the presence of an improving move in a ball of radius r around x. If the Scores are stored in memory this check requires constant time.*
- *The Scores $S_v(x)$ for $v \in M^r$ can be updated when we move from one solution x to $x \oplus t$ using Algorithm 1 in time $O(b(k)(3kc)^r|t|)$, where $b(k)$ is a bound of the time required to evaluate any subfunction $f^{(l)}$. This time is independent of n if k, r and c are independent of n.*

4. THE HAMMING-BALL HILL CLIMBER

The efficient Scores update of Algorithm 1 can be used in combination with any trajectory-based method like, Iterated Local Search or Tabu Search. In this section we describe a next ascent hill climber based on it. The proposed hill climber is shown in Algorithm 3. We assume maximization and for this reason we say that it is an "ascent" hill climber. However, the algorithm can consider minimization just changing the $>$ operators in lines 5 and 10 by $<$ operators. In the algorithm, variable *best* stores the best found solution at any given time. The algorithm starts by assigning the special value $\perp$, which means "no solution" to *best*. Next, it enters a loop that is repeated until the stopping condition is met. Each run of the loop body is an ascent starting from a random solution in the search space. In the loop, once a random solution has been selected and stored in x, the algorithm computes the Scores $S_v(x)$ for all $v \in M^r$ using the expression (6). The inner loop starting in line 5 implements the next ascent. While an improving move exists ($S_v > 0$ for some $v \in M^r$), the algorithm selects one of the improving moves t (line 6), updates the Scores using Algorithm 1 (line 7) and changes the current solution by the new one (line 8).

Algorithm 3 Hamming-ball next ascent.

```
 1: best ← ⊥
 2: while stop condition not met do
 3:    x ← randomSolution();
 4:    S ← computeScores(x);
 5:    while S_v > 0 for some v ∈ M^r do
 6:       t ← selectImprovingMove(S);
 7:       updateScores(S,x,t);
 8:       x ← x ⊕ t;
 9:    end while
10:    if best = ⊥ or f(x) > f(best) then
11:       best ← x;
12:    end if
13: end while
```

Regarding the selection of the improving move, our approach in the experiments was to select always the one with the lowest Hamming distance to the current solution, that is, the lowest value of $|t|$. The main reason for selecting the nearest improving move is that, as stated by Theorem 1, the time required for updating the Scores is proportional to $|t|$. Thus, the nearest improving move is the fastest move in our algorithm. This selection can be done in constant time, since the classification of the moves as improving or non-improving can be done when the Scores are calculated and the Scores can be added to different lists depending on the distance to the current solution.

From the point of view of computation time, we expect the statements between lines 6 and 8 to run in a time that is independent of n. However, this time depends on k, c and r, where r is the only algorithm-dependent parameter (the others depend on the problem instance). According to Theorem 1, the time is exponential in r. Therefore, we have to be especially careful when we set the radius of the ball. Lines 3 and 4 are expected to run in linear time with respect to n, since the number of Scores (and variables) is linear.

Let us now consider the quality of the solutions obtained by the algorithm when we change r. Let us pick two radii for the ball: $r_1 < r_2$. Let us imagine that two ascents start from the same solution, one in each algorithm. Both instances of the algorithm will visit the same solutions until the algorithm with radius r_1 gets stuck in a local optimum or a plateau. In this case, however, the algorithm with r_2, which is considering a larger neighborhood, could find an improving move and could continue. Thus, we expect longer ascents and better quality solutions at the end of the ascents for larger values of r.

The fitness value of the best solution is a non-decreasing function of the time. As we increase r we expect better solutions at the end of the ascents, but the time spent in each ascent is also longer. It could happen that an instance of the algorithm with a lower value for r can reach better solutions earlier just because it is faster and finds an appropriate path joining lower distance moves. This fact makes impossible to set an *a priori* value for r that is valid for all the problems and instances. In the experimental section we will discuss this issue again on the results.

5. EXPERIMENTAL RESULTS

In this section we present experimental results obtained with the Hamming-ball next ascent. For the experiments we used NKq-landscapes. They are randomly generated functions that are specially interesting because it is possible to change their ruggedness and neutrality changing K and q, respectively. The objective function is:

$$f(x) = \sum_{l=1}^{n} f^{(l)}(x), \tag{10}$$

where each subfunction $f^{(l)}$ depends on variable x_l and other K variables and we have $k = K + 1$. These variables can be randomly selected from the remaining $n-1$, which yields the random-model NKq-landscapes. If the variables for subfunction $f^{(l)}$ are $\{x_l, x_{l+1}, \ldots, x_{l+K}\}$ with sum modulo n (and indices in the range 1 to n), then we have the adjacent-model NKq-landscapes. The latter has the advantage that a global optimum can be found in polynomial time using an algorithm proposed by Wright *et al.* [10], but also the number of subfunctions depending on a given variable is constant: $c = k = K + 1$. In the random-model there is no constant bound for the number of subfunctions in which the variables

appear: a variable could appear in up to n subfunctions. The 2^{K+1} values that each subfunction $f^{(l)}$ can take in both, the adjacent and random models, are randomly generated from the set of integers $\{0, 1, \ldots, q-1\}$.

We proved in Section 3 that the Scores can be updated in a time that is independent of the size of the problem n. However, in a real implementation of Algorithm 3 there are two initialization procedures that require more than constant time. First, it is necessary to compute the set of moves M^r and allocate memory for the corresponding Scores. This is done only once for each problem instance and we call it *instance-dependent initialization.* Second, at the beginning of each ascent, a random solution has to be selected from the search space and the Scores for the solution have to be computed. We call this procedure *solution-dependent initialization.*

5.1 Illustrating Average Runtime

In a first experiment, we will estimate the average cost of each move empirically. We will do this by flipping *every variable the same number of times*; each individual flip results in a unique set of updates to the Score vector. This means we ignore improving moves and focus exclusively on the cost of updating the Score vector for every possible bit flip.

For a test problem we will use the adjacent-model NKq-landscape. We used instances of the problem with $n = 1,000$ to $n = 12,000$, $K = 1$ to $K = 4$ and $q = 2^{K+1}$. We generated 30 instances for each combination of values and averaged the results over the 30 instances. For r we used values $r = 1$ to 4. In each run we start from a random solution and flip each bit of the solution in turn until we reach $120,000$ flips. For instances with $n = 12,000$ this means flipping each bit exactly ten times. In order to have an unbiased distribution of bit flips only divisors of $12,000$ are allowed values for n. This way we ensure that each bit is flipped the same number of times in each instance.

In Figure 2 we can see the time required by the $120,000$ bit flips for $K = 3$ and all the values of r as a function of the size n. Except for some random fluctuations, the time is constant. Figure 3(a) shows the cardinality of M^r as a function of n. We can clearly see the predicted linear evolution. In Figures 3(b) and 3(c) we can observe the times required by the instance-dependent and solution-dependent initialization procedures. They increase more or less linearly with n. They can impose a time-limit to add to the memory-limit imposed by $|M^r|$ in the scalability of our proposal. The same behaviour can be observed for the other values of K (not shown). However, as we increase K, the times and $|M^r|$ also increase.

Since the time for performing the 120,000 moves is the same for all n, we have computed the mean time for all the values of n while K and r are fixed. Figure 4 shows how this time depends on K for different values of r. According to the results of Section 3 this time is exponential in r with K appearing in the base of the expression.

We repeated the previous experiment with the random-model NKq-landscape. In Figure 5(a) we can see the time required by the $120,000$ bit flips for $K = 3$, $q = 16$ and $r = 1$ to 3 as a function of the size n. Now we can observe a slight increase of the time with n. This is a consequence of the increasing number of subfunctions that depend on the same variable. Figure 5(c) shows the average (in the 30 instances) of the largest number of subfunctions a single

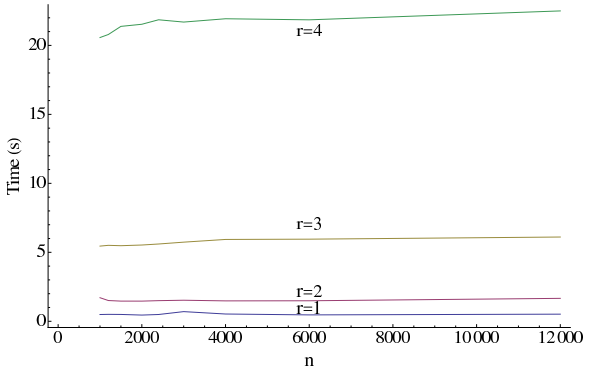
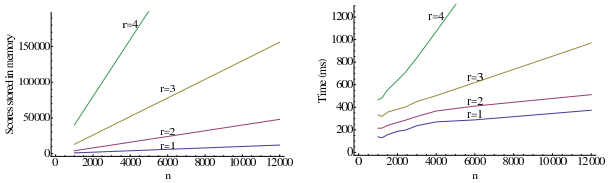
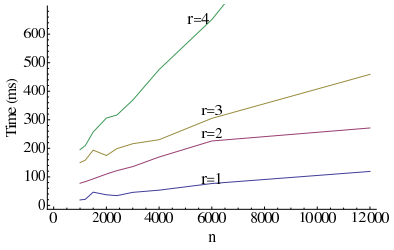


Figure 2: Time for Score updates in the adjacent-model NKq-landscapes.



(a) Scores stored in memory (b) Instance-dependent init.



(c) Solution-dependent init.

Figure 3: Times and Scores for adjacent-model NKq-landscapes.

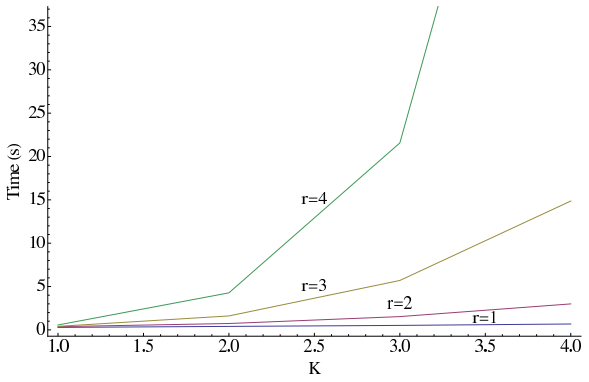
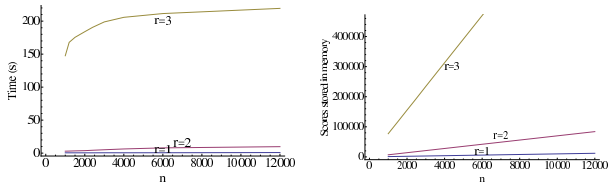


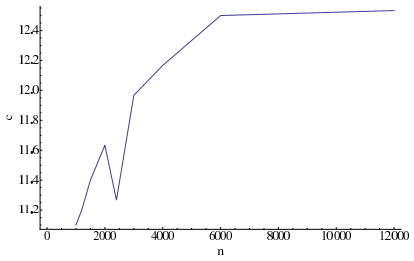
Figure 4: Time required for the 120,000 moves in the adjacent-model NKq-landscape as a function of K. Each curve corresponds to a different value of r.

variable affects. In the adjacent-model this number was constant, exactly $c = K + 1$. In the random-model this number is increasing with n and is not bounded by a constant. Regarding the cardinality of M^r, it still increases in a linear

way with n as illustrated in Figure 5(b), although in theory the increase could be more than linear.



(a) Time for Score updates (b) Scores stored in memory



(c) Maximum subfunctions affected, c

Figure 5: Time, Scores and maximum number of subfunctions affected by a single variable for random-model NKq-landscapes.

5.2 On the Quality of the Solutions

In Section 4 we argued that an ascent of the Hamming-ball next ascent with radius $r_1 < r_2$ cannot obtain a better solution than an ascent starting from the same solution but using radius r_2. As a consequence, if the stopping condition of Algorithm 3 is to reach a fixed number of ascents, then we expect the quality of the solutions using r_2 to be no worse than the quality of the solutions when r_1 is the radius. However, larger radius also means longer execution time, not only for the Scores update but also for the different initialization procedures. If the stopping condition is to reach a fixed amount of time we cannot say which radius is the best option. The experiments in this section illustrate what happens in this case of NKq-landscapes.

Let us start analyzing the adjacent-model. We fix $n = 10,000$ and generated 30 instances for each combination of K used. We run the algorithm for 120 seconds in all the cases. Figure 6 shows the average fitness over the 30 instances of the best solution found by the algorithms against the elapsed time from the start of the search. The radius used are $r = 1$ to 6. In the figure $K = 2$ and $q = 8$. We can observe that using larger values for r the algorithm is able to find better solutions at any time. In the figure, it is clear that the algorithm with $r = 1$ cannot reach in 120 seconds the quality of the solutions found by the one with $r = 2$ at the beginning of its search. The same applies in the comparison with other radii. We can conclude that, in this example and for $r = 1$ to 6, increasing the value of r we obtain progressively better algorithms, even when the stopping condition is a time limit.

However, this improvement has also limitations. The gain in the quality of the solutions when we increase r from 1 to 2 is larger than the gain when we increase r from 2 to 3. Since the fitness value of the global optimum acts as a bound for the quality of the solutions, there must be an r value for which there is no gain. At this point, increasing r we can only penalize the efficacy of the algorithm because the run-

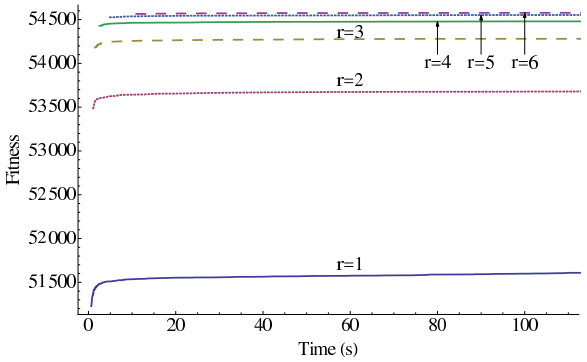


Figure 6: Best solution fitness over time for the Hamming-ball next ascent in adjacent-model NKq-landscapes.

time will still increase. To check this fact, we show in Figure 7 the normalized distance to the global optimum of the Hamming-ball next ascent for different radii (note that we are maximizing the objective function, but minimizing the distance to the global optimum). The points are averages over 30 random instances with $n = 10,000$, $K = 1$, $q = 4$ and $r = 1$ to 10. The algorithms were run for 120 seconds. The normalized distance to the optimum, n_d, is:

$$n_d(x) = \frac{f^* - f(x)}{f^*}, \tag{11}$$

where f^* is the fitness value of the global optimum, computed using the algorithm by Wright *et al.* [10].

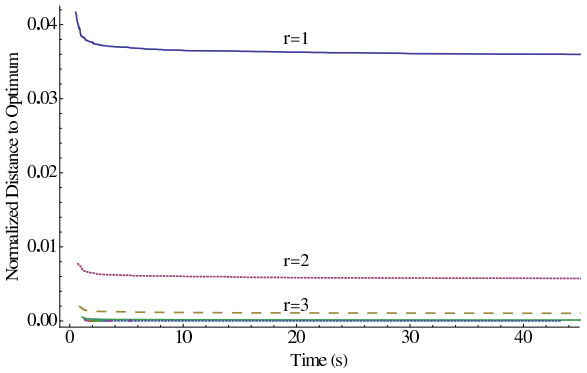


Figure 7: Normalized distance to the global optimum for the Hamming-ball next ascent.

With $r = 1$ the algorithm is approximately within 3.6% of the global optimum after 20 seconds. As we increase r, the algorithms get closer to the global optimum and the gains shrink. For $r = 3$ the results are withing 0.2% of the global optimum. For $r > 3$ the differences are so small that cannot be observed in the plot. For $r \geq 6$ an optimal solution is found in all the instances. For $r = 6$ the solution is found in 3.7 seconds. For $r = 7$ it is found in 2.1 seconds, and this is the fastest algorithm. For $r = 8, 9$ and 10 the algorithm requires 2.8 s, 3.4 s, and 5 s to find the optimal solution. For $r = 10$ the algorithm finds an optimal solution in the first ascent for all the instances.

Let us now consider a random NKq-Landscape, one that does not use adjacent interacting variables. In Figure 8 we

plot the quality of the best solution as a function of time. The instance used has $n = 10,000$, $K = 2$ and $q = 8$. The plot shows the average over 30 randomly generated instances. We observe the same behaviour as in the adjacent-model. The only remarkable difference is that the instance dependent initialization is longer and this is why the curves are displaced in time as r increases.

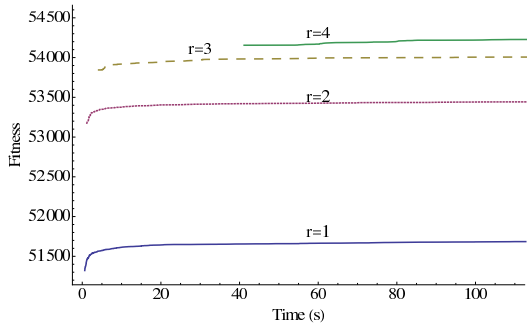


Figure 8: Best solution fitness over time for the Hamming-ball next ascent in random-model NKq-landscapes.

5.3 NKq-landscapes and MAX-kSAT

It has been empirically observed a common pattern in some industrial instances of MAX-3SAT. The number of clauses in which the same three variables appear is four with a high frequency. An hypothesis to explain this pattern is that these groups of clauses appear as a consequence of the Tseitin transformation [7], which is used to transform a general Boolean expression into CNF. The result is that these MAX-3SAT instances are similar in structure to NKq-landscapes instances with $K = 2$ and $q = 5$. Thus, we could expect the Hamming-ball next ascent to share the same behaviour in MAX-3SAT instances (and MAX-kSAT instances) as the NKq-landscape instances we have studied. In order to confirm this hypothesis we show in Figure 9 the fitness of the best solution against time for 30 randomly generated MAX-3SAT instances with groups of four clauses with the same variables. The number of variables is $n = 10,000$ and clauses are $m = 40,000$.

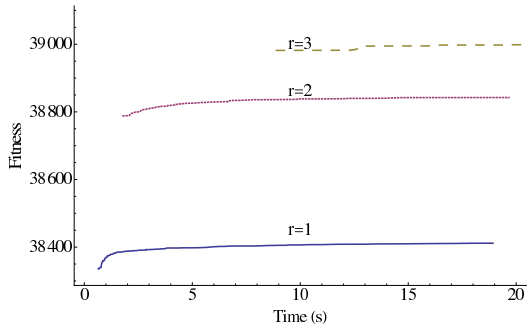


Figure 9: Best solution fitness over time for the Hamming-ball next ascent in MAX-3SAT.

6. CONCLUSIONS AND FUTURE WORK

In this paper we have provided an algorithm to efficiently identify improving moves in a Hamming ball of radius r around a solution of a k-bounded pseudo-Boolean optimization problem that can be written as a sum of subfunctions.

The empirical results on NKq-landscape instances show that increasing r improves the quality of the solutions found by next ascent local search. Further work is needed to study the benefits and limitations of using the Hamming Ball r-radius neighborhood for problems such as MAX-kSAT. The efficient algorithm for updating the Scores vector can also be combined with different strategies to escape from plateaus and local optima. In particular, a random walk flipping a small fraction of the decision variables could considerably reduce the computation time of the random restart.

7. ACKNOWLEDGMENTS

This research was sponsored by the Fulbright program, the Spanish Ministry of Education ("José Castillejo" mobility program), the Universidad de Málaga, Andalucía Tech, the Spanish Ministry of Science and Innovation and FEDER under contract TIN2011-28194, VSB-Technical University of Ostrava under contract OTRI 8.06/5.47.4142 and 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. The authors would also like to thank the organizers and participants of the seminar on Theory of Evolutionary Algorithms (13271) at Schloß Dagstuhl - Leibniz-Zentrum für Informatik.

8. REFERENCES

[1] W. Chen, D. Whitley, D. Hains, and A. Howe. Second order partial derivatives for NK-landscapes. In *Proceeding of GECCO*, pages 503–510. ACM, 2013.

[2] S. R. Finch. *Mathematical Constants*, chapter Otter's Tree Enumeration Constants, pages 295–316. Cambridge University Press, 2003.

[3] R. Heckendorn, S. Rana, and D. Whitley. Test function generators as embedded landscapes. In *FOGA*, pages 183–198. Morgan Kaufmann, 1999.

[4] H. H. Hoos and T. Stützle. *Stochastic Local Search: Foundations and Applications*. Morgan Kaufman, 2004.

[5] R. Otter. The number of trees. *Annals of Mathematics*, 49(3):583–599, 1948.

[6] S. Szeider. The parameterized complexity of k-flip local search for SAT and MAX SAT. *Discrete Optimization*, 8(1):139–145, 2011.

[7] G. S. Tseitin. On the complexity of derivations in the propositional calculus. *Studies in Mathematics and Mathematical Logic*, Part II:115–125, 1968.

[8] D. Whitley and W. Chen. Constant time steepest descent local search with lookahead for NK-landscapes and MAX-kSAT. In *Proceeding of GECCO*, pages 1357–1364. ACM, 2012.

[9] D. Whitley, A. Howe, and D. Hains. Greedy or not? best improving versus first improving stochastic local search for MAXSAT. In *Proc.of AAAI-2013*, 2013.

[10] A. Wright, R. Thompson, and J. Zhang. The computational complexity of NK fitness functions. *IEEE Trans. Evol. Comp.*, 4(4):373–379, 2000.