Dynamic Hill Climbing

Michael de la Maza

Numinous Noetics Group
Artificial Intelligence Laboratory
Massachusetts Institute of Technology
545 Technology Square
Cambridge, Massachusetts 02139-3563
mdlm@ai.mit.edu

Deniz Yuret

Numinous Noetics Group
Artificial Intelligence Laboratory
Massachusetts Institute of Technology
545 Technology Square
Cambridge, Massachusetts 02139-3563
deniz@ai.mit.edu

1 Introduction

We spend our days minimizing and maximizing. What is the least I can pay for this car? How much will the IRS let me get away with? What is the best way to get to work, the road or the freeway?

The problem of minimizing and maximizing functions has been widely studied in operations research, applied mathematics, and artificial intelligence. Many AI methods have optimization procedures as subcomponents and a large part of their performance hinges critically on the quality of these optimization techniques. For example, the backpropagation algorithm that is used to update the weights of neural networks is a form of gradient descent optimization. Some of the complaints made about standard neural networks – they are slow, they get stuck – can be directly attributed to the backpropagation algorithm.

The problem of minimizing a function can be stated rigorously: Given a function f, find a point x such that there is no y that has the property $f(y) < f(x)$. In this article we will consider the case in which the function f is defined over a subset of Cartesian space, so the point x is a vector of n real numbers, where n is the dimensionality of space. The problem of maximizing a function f can be turned into a minimization problem by simply multiplying the function by -1.

Simple minimization problems can be solved using the techniques of elementary calculus. Consider the polynomial function $f(x) = x^3 + 5x^2 + 3x$. If we want to find the minimum of f over the interval [-4, 4] all we need to do is find the derivative of the function, set it to 0, and then solve for x. The derivative of f is $3x^2 + 10x + 3$. When we solve for x, we get two solutions: $x = -1/3$ and $x = -3$. These two points are local extrema of the function. To check whether they are local maxima or minima, we take the second derivative and check to see if it is positive or negative. The second derivative of f is $6x + 10$. When $x = -1/3$ the second derivative is positive, so $x = -1/3$ is a local minima. When $x = -3$ the second derivative is negative, so $x = -3$ is a local maxima. Finally, to determine if $x = -1/3$ is a global minima over the interval [-4, 4], we need to compare it to the value of the function at the endpoints of the interval. Since $f(-4) > f(-1/3)$ and $f(4) > f(-1/3)$, $x = -1/3$ is indeed the global minima.

Unfortunately, this simple technique cannot be applied to many of the problems that AI researchers are faced with. Indeed, for many problems it may not be possible to find the derivative of f. In those cases the only way to get information about the function is to evaluate it at different values of x.

Thus, more powerful methods must be employed to minimize functions that actually occur in practice. In this article we describe a new optimization algorithm, called dynamic hill climbing (DHC), that draws on ideas from genetic algorithms, hill climbing, and conjugate gradient methods. On DeJong's five function test suite, a well known set of functions that has been widely used to study optimization algorithms, dynamic hill climbing outperforms genetic algorithms and simulated annealing, often by a wide margin. DHC is currently being used in a medical imaging system that helps brain surgeons to plan and execute operations.

2 Dynamic hill climbing

The dynamic hill climbing algorithm is divided into an outer loop and inner loop. The inner loop contains a very efficient technique for locating local optima. The outer loop focuses the attention of the inner loop on different parts of the search space in order to ensure that the entire space has been explored.

The inner loop maintains a list of $2n$ vectors of which n form an orthogonal basis set for the search space and the other n are the negations of these vectors and an additional 2 vectors that are functions of the directions that DHC has moved in recently. In addition, the inner loop stores the current point, $\vec{x}$. The vectors determine where the DHC algorithm is going to "look" next. Starting with the vector that has the largest magnitude, $\vec{v_i}$, DHC evaluates the point $\vec{x} + \vec{v_i}$ and compares it to the value of f at $\vec{x}$. If $f(\vec{x} + \vec{v_i}) < f(\vec{x})$ then $\vec{x} + \vec{v_i}$ becomes the new point ($\vec{x} = \vec{x} + \vec{v_i}$), and the process is repeated beginning once again with the vector that has the greatest magnitude. If $\vec{v_i}$ is chosen twice in a row then it is doubled ($\vec{v_i} = 2\vec{v_i}$). If, on the other hand, $f(\vec{x} + \vec{v_i}) \geq f(\vec{x})$, then the magnitude of $\vec{v_i}$ is halved ($\vec{v_i} = \vec{v_i}/2$) and the vector with the next highest magnitude is tried.

The inner loop continues to update the magnitudes of the vectors and the current position, $\vec{x}$, in this fashion until it finds a point from which it can no longer decrease the function evaluation. This occurs when the magnitude of all of the vectors reaches some minimum threshold that is determined by the

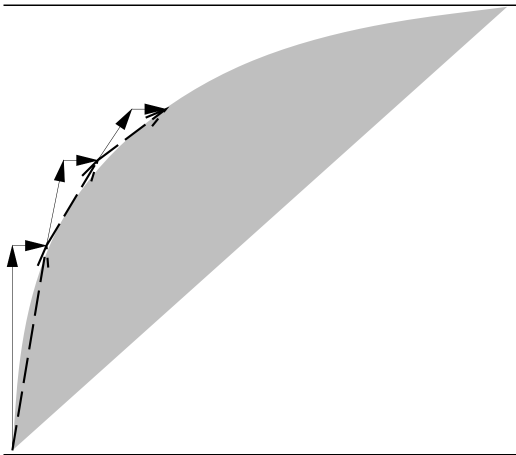

Figure 1: Computing the gradient vectors. The gradient vectors are the bold dashed lines. The positions of the best point are shown by the thin arrows.

user. When this happens the inner loop exits and the outer loop re-initializes the inner loop with a new starting point.

The two extra vectors, called gradient vectors, are functions of the last two different directions that the point $\vec{x}$ has moved in. More specifically, if $\vec{x}$ has moved from point $\vec{x'}$ to point $\vec{x'} + \vec{v_i}$ and from there it has moved to point $\vec{x'} + \vec{v_i} + \vec{v_j}$ (where the direction of $\vec{v_i}$ is different from the direction of $\vec{v_j}$) then one of the gradient vectors is the vector from $\vec{x'}$ to $\vec{x'} + \vec{v_i} + \vec{v_j}$ and the other vector is its negation. Figure 1 gives an example of how these two extra vectors are updated as the point $\vec{x}$ changes. The purpose of the gradient vectors is to track the curvature of valleys in the search space.

Before describing the outer loop, stepping through a simple example will illustrate how the inner loop works. Suppose that the goal is to minimize the polynomial $f(\vec{x}) = -x_1^2 - x_2^2$ over the interval $x_1, x_2 \in [-5.12, 5.12]$. If we set the lower threshold to .01 and assume that the starting point is (4,-5) then Table 2 shows how the 6 vectors and the current point change. In this case the outer loop is not needed because the inner loop finds the global optimum of the function on the first pass.

When the inner loop reduces the magnitude of all of the vectors to their minimum size it returns to the outer loop which initializes it with a new starting point. The outer loop maintains a list of the points that have already been evaluated by the inner loop and it chooses the new initialization

evals	vector	xbst	xcur	vec0	vec1	vec2	vec3	vec4	vec5
1	0	(4.00,-5.00)	(9.12,-5.00)	(5.12,0.00)	(0.00,5.12)	(0.00,0.00)	(0.00,0.00)	(0.00,-5.12)	(-5.12,0.00)
2	1	(4.00,-5.00)	(4.00,0.12)	(2.56,0.00)	(0.00,5.12)	(0.00,0.00)	(0.00,0.00)	(0.00,-5.12)	(-5.12,0.00)
3	1	(4.00,0.12)	(4.00,5.24)	(2.56,0.00)	(0.00,5.12)	(0.00,0.00)	(0.00,0.00)	(0.00,-2.56)	(-5.12,0.00)
4	5	(4.00,0.12)	(-1.12,0.12)	(2.56,0.00)	(0.00,2.56)	(0.00,0.00)	(0.00,0.00)	(0.00,-2.56)	(-5.12,0.00)
5	2	(-1.12,0.12)	(-6.24,5.24)	(2.56,0.00)	(0.00,2.56)	(-5.12,5.12)	(5.12,-5.12)	(0.00,-2.56)	(-5.12,0.00)
6	3	(-1.12,0.12)	(4.00,-5.00)	(2.56,0.00)	(0.00,2.56)	(-2.56,2.56)	(5.12,-5.12)	(0.00,-2.56)	(-5.12,0.00)
7	5	(-1.12,0.12)	(-6.24,0.12)	(2.56,0.00)	(0.00,2.56)	(-2.56,2.56)	(2.56,-2.56)	(0.00,-2.56)	(-5.12,0.00)
8	2	(-1.12,0.12)	(-3.68,2.68)	(2.56,0.00)	(0.00,2.56)	(-2.56,2.56)	(2.56,-2.56)	(0.00,-2.56)	(-2.56,0.00)
9	3	(-1.12,0.12)	(1.44,-2.44)	(2.56,0.00)	(0.00,2.56)	(-1.28,1.28)	(2.56,-2.56)	(0.00,-2.56)	(-2.56,0.00)
10	0	(-1.12,0.12)	(1.44,0.12)	(2.56,0.00)	(0.00,2.56)	(-1.28,1.28)	(1.28,-1.28)	(0.00,-2.56)	(-2.56,0.00)
11	1	(-1.12,0.12)	(-1.12,2.68)	(1.28,0.00)	(0.00,2.56)	(-1.28,1.28)	(1.28,-1.28)	(0.00,-2.56)	(-2.56,0.00)
12	4	(-1.12,0.12)	(-1.12,-2.44)	(1.28,0.00)	(0.00,1.28)	(-1.28,1.28)	(1.28,-1.28)	(0.00,-2.56)	(-2.56,0.00)
13	5	(-1.12,0.12)	(-3.68,0.12)	(1.28,0.00)	(0.00,1.28)	(-1.28,1.28)	(1.28,-1.28)	(0.00,-1.28)	(-2.56,0.00)
14	2	(-1.12,0.12)	(-2.40,1.40)	(1.28,0.00)	(0.00,1.28)	(-1.28,1.28)	(1.28,-1.28)	(0.00,-1.28)	(-1.28,0.00)
15	3	(-1.12,0.12)	(0.16,-1.16)	(1.28,0.00)	(0.00,1.28)	(-0.64,0.64)	(1.28,-1.28)	(0.00,-1.28)	(-1.28,0.00)
16	0	(-1.12,0.12)	(0.16,0.12)	(1.28,0.00)	(0.00,1.28)	(-0.64,0.64)	(0.64,-0.64)	(0.00,-1.28)	(-1.28,0.00)
17	0	(0.16,0.12)	(1.44,0.12)	(1.28,0.00)	(0.00,1.28)	(-0.64,0.64)	(0.64,-0.64)	(0.00,-1.28)	(-0.64,0.00)
18	1	(0.16,0.12)	(0.16,1.40)	(0.64,0.00)	(0.00,1.28)	(-0.64,0.64)	(0.64,-0.64)	(0.00,-1.28)	(-0.64,0.00)
19	4	(0.16,0.12)	(0.16,-1.16)	(0.64,0.00)	(0.00,0.64)	(-0.64,0.64)	(0.64,-0.64)	(0.00,-1.28)	(-0.64,0.00)
20	2	(0.16,0.12)	(-0.48,0.76)	(0.64,0.00)	(0.00,0.64)	(-0.64,0.64)	(0.64,-0.64)	(0.00,-0.64)	(-0.64,0.00)
21	3	(0.16,0.12)	(0.80,-0.52)	(0.64,0.00)	(0.00,0.64)	(-0.32,0.32)	(0.64,-0.64)	(0.00,-0.64)	(-0.64,0.00)
22	0	(0.16,0.12)	(0.80,0.12)	(0.64,0.00)	(0.00,0.64)	(-0.32,0.32)	(0.32,-0.32)	(0.00,-0.64)	(-0.64,0.00)
23	1	(0.16,0.12)	(0.16,0.76)	(0.32,0.00)	(0.00,0.64)	(-0.32,0.32)	(0.32,-0.32)	(0.00,-0.64)	(-0.64,0.00)
24	4	(0.16,0.12)	(0.16,-0.52)	(0.32,0.00)	(0.00,0.32)	(-0.32,0.32)	(0.32,-0.32)	(0.00,-0.64)	(-0.64,0.00)
25	5	(0.16,0.12)	(-0.48,0.12)	(0.32,0.00)	(0.00,0.32)	(-0.32,0.32)	(0.32,-0.32)	(0.00,-0.32)	(-0.64,0.00)
26	2	(0.16,0.12)	(-0.16,0.44)	(0.32,0.00)	(0.00,0.32)	(-0.32,0.32)	(0.32,-0.32)	(0.00,-0.32)	(-0.32,0.00)
27	3	(0.16,0.12)	(0.48,-0.20)	(0.32,0.00)	(0.00,0.32)	(-0.16,0.16)	(0.32,-0.32)	(0.00,-0.32)	(-0.32,0.00)
28	0	(0.16,0.12)	(0.48,0.12)	(0.32,0.00)	(0.00,0.32)	(-0.16,0.16)	(0.16,-0.16)	(0.00,-0.32)	(-0.32,0.00)
29	1	(0.16,0.12)	(0.16,0.44)	(0.16,0.00)	(0.00,0.32)	(-0.16,0.16)	(0.16,-0.16)	(0.00,-0.32)	(-0.32,0.00)
30	4	(0.16,0.12)	(0.16,-0.20)	(0.16,0.00)	(0.00,0.16)	(-0.16,0.16)	(0.16,-0.16)	(0.00,-0.32)	(-0.32,0.00)
31	5	(0.16,0.12)	(-0.16,0.12)	(0.16,0.00)	(0.00,0.16)	(-0.16,0.16)	(0.16,-0.16)	(0.00,-0.16)	(-0.32,0.00)
32	2	(0.16,0.12)	(-0.00,0.28)	(0.16,0.00)	(0.00,0.16)	(-0.16,0.16)	(0.16,-0.16)	(0.00,-0.16)	(-0.16,0.00)
33	3	(0.16,0.12)	(0.32,-0.04)	(0.16,0.00)	(0.00,0.16)	(-0.08,0.08)	(0.16,-0.16)	(0.00,-0.16)	(-0.16,0.00)
34	0	(0.16,0.12)	(0.32,0.12)	(0.16,0.00)	(0.00,0.16)	(-0.08,0.08)	(0.08,-0.08)	(0.00,-0.16)	(-0.16,0.00)
35	1	(0.16,0.12)	(0.16,0.28)	(0.08,0.00)	(0.00,0.16)	(-0.08,0.08)	(0.08,-0.08)	(0.00,-0.16)	(-0.16,0.00)
36	4	(0.16,0.12)	(0.16,-0.04)	(0.08,0.00)	(0.00,0.08)	(-0.08,0.08)	(0.08,-0.08)	(0.00,-0.16)	(-0.16,0.00)
37	2	(0.16,-0.04)	(1.44,-0.20)	(0.08,0.00)	(0.00,0.08)	(1.28,-0.16)	(-1.28,0.16)	(0.00,-0.16)	(-0.16,0.00)
38	3	(0.16,-0.04)	(-1.12,0.12)	(0.08,0.00)	(0.00,0.08)	(0.64,-0.08)	(-1.28,0.16)	(0.00,-0.16)	(-0.16,0.00)
39	2	(0.16,-0.04)	(0.80,-0.12)	(0.08,0.00)	(0.00,0.08)	(0.64,-0.08)	(-0.64,0.08)	(0.00,-0.16)	(-0.16,0.00)
40	3	(0.16,-0.04)	(-0.48,0.04)	(0.08,0.00)	(0.00,0.08)	(0.32,-0.04)	(-0.64,0.08)	(0.00,-0.16)	(-0.16,0.00)
41	2	(0.16,-0.04)	(0.48,-0.08)	(0.08,0.00)	(0.00,0.08)	(0.32,-0.04)	(-0.32,0.04)	(0.00,-0.16)	(-0.16,0.00)
42	3	(0.16,-0.04)	(-0.16,0.00)	(0.08,0.00)	(0.00,0.08)	(0.16,-0.02)	(-0.32,0.04)	(0.00,-0.16)	(-0.16,0.00)
43	3	(-0.16,0.00)	(-0.48,0.04)	(0.08,0.00)	(0.00,0.08)	(0.16,-0.02)	(-0.32,0.04)	(0.00,-0.16)	(-0.16,0.00)
44	2	(-0.16,0.00)	(-0.00,-0.02)	(0.08,0.00)	(0.00,0.08)	(0.16,-0.02)	(-0.16,0.02)	(0.00,-0.16)	(-0.16,0.00)
45	2	(-0.00,-0.02)	(0.16,-0.04)	(0.08,0.00)	(0.00,0.08)	(0.16,-0.02)	(-0.08,0.01)	(0.00,-0.16)	(-0.16,0.00)
46	4	(-0.00,-0.02)	(-0.00,-0.18)	(0.08,0.00)	(0.00,0.08)	(0.08,-0.01)	(-0.08,0.01)	(0.00,-0.16)	(-0.16,0.00)
47	5	(-0.00,-0.02)	(-0.16,-0.02)	(0.08,0.00)	(0.00,0.08)	(0.08,-0.01)	(-0.08,0.01)	(0.00,-0.08)	(-0.16,0.00)
48	2	(-0.00,-0.02)	(0.08,-0.03)	(0.08,0.00)	(0.00,0.08)	(0.08,-0.01)	(-0.08,0.01)	(0.00,-0.08)	(-0.08,0.00)
49	3	(-0.00,-0.02)	(-0.08,-0.01)	(0.08,0.00)	(0.00,0.08)	(0.04,-0.01)	(-0.08,0.01)	(0.00,-0.08)	(-0.08,0.00)
50	0	(-0.00,-0.02)	(0.08,-0.02)	(0.08,0.00)	(0.00,0.08)	(0.04,-0.01)	(-0.04,0.01)	(0.00,-0.08)	(-0.08,0.00)
51	1	(-0.00,-0.02)	(-0.00,0.06)	(0.04,0.00)	(0.00,0.08)	(0.04,-0.01)	(-0.04,0.01)	(0.00,-0.08)	(-0.08,0.00)
52	4	(-0.00,-0.02)	(-0.00,-0.10)	(0.04,0.00)	(0.00,0.04)	(0.04,-0.01)	(-0.04,0.01)	(0.00,-0.08)	(-0.08,0.00)
53	5	(-0.00,-0.02)	(-0.08,-0.02)	(0.04,0.00)	(0.00,0.04)	(0.04,-0.01)	(-0.04,0.01)	(0.00,-0.04)	(-0.08,0.00)
54	2	(-0.00,-0.02)	(0.04,-0.02)	(0.04,0.00)	(0.00,0.04)	(0.04,-0.01)	(-0.04,0.01)	(0.00,-0.04)	(-0.04,0.00)
55	3	(-0.00,-0.02)	(-0.04,-0.01)	(0.04,0.00)	(0.00,0.04)	(0.02,-0.00)	(-0.04,0.01)	(0.00,-0.04)	(-0.04,0.00)
56	0	(-0.00,-0.02)	(0.04,-0.02)	(0.04,0.00)	(0.00,0.04)	(0.02,-0.00)	(-0.02,0.00)	(0.00,-0.04)	(-0.04,0.00)
57	1	(-0.00,-0.02)	(-0.00,0.02)	(0.02,0.00)	(0.00,0.04)	(0.02,-0.00)	(-0.02,0.00)	(0.00,-0.04)	(-0.04,0.00)
58	4	(-0.00,-0.02)	(-0.00,-0.06)	(0.02,0.00)	(0.00,0.02)	(0.02,-0.00)	(-0.02,0.00)	(0.00,-0.04)	(-0.04,0.00)
59	5	(-0.00,-0.02)	(-0.04,-0.02)	(0.02,0.00)	(0.00,0.02)	(0.02,-0.00)	(-0.02,0.00)	(0.00,-0.02)	(-0.04,0.00)
60	2	(-0.00,-0.02)	(0.02,-0.02)	(0.02,0.00)	(0.00,0.02)	(0.02,-0.00)	(-0.02,0.00)	(0.00,-0.02)	(-0.02,0.00)
61	3	(-0.00,-0.02)	(-0.02,-0.02)	(0.02,0.00)	(0.00,0.02)	(0.01,-0.00)	(-0.02,0.00)	(0.00,-0.02)	(-0.02,0.00)
62	0	(-0.00,-0.02)	(0.02,-0.02)	(0.02,0.00)	(0.00,0.02)	(0.01,-0.00)	(-0.01,0.00)	(0.00,-0.02)	(-0.02,0.00)
63	1	(-0.00,-0.02)	(-0.00,0.00)	(0.01,0.00)	(0.00,0.02)	(0.01,-0.00)	(-0.01,0.00)	(0.00,-0.02)	(-0.02,0.00)
64	2	(-0.00,0.00)	(-0.16,0.04)	(0.01,0.00)	(0.00,0.02)	(-0.16,0.04)	(0.16,-0.04)	(0.00,-0.01)	(-0.02,0.00)

Figure 2: How the inner loop works. The first column shows the number of function evaluations, the second column shows the number of the current vector, and the last six columns show the values of the six vectors. The middle two of these vectors are the gradient vectors and the other four are the vectors that lie along the axes of the space. The current point is computed by adding the current vector to the best point.

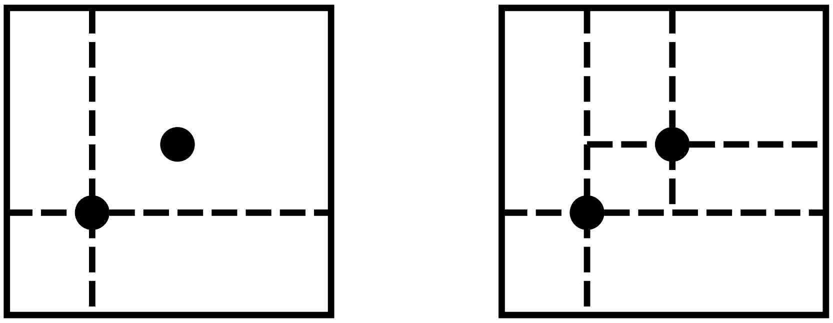

Figure 3: Hyper-planes.

point from a region that is far away from these points. This has the effect of forcing the inner loop to uniformly explore the search space. More specifically, whenever the inner loop visits a new point in space, the hyper-rectangle that the point is in is subdivided. The initialization point is chosen from the hyper-rectangle with the largest area. Figure 3 shows how this procedure works on a simple example. One disadvantage of this procedure is that it only works for functions with low dimensionality because the number of hyper-rectangles is exponential in the number of dimensions. Many other procedures could be substituted for this outer loop. For example, choosing a set of points randomly, evaluating each one, and re-initializing the inner loop with the best of these points is another possible outer loop procedure. To generate the data shown in Figure 5 this latter procedure was used.

The Lisp code in the appendix gives an implementation of the inner loop.

3 DeJong's test functions

DeJong's test suite, initially created to explore the characteristics of genetic algorithms, consists of five functions:

- Sphere. A three dimensional unimodal function.

$$f_1(\vec{x}) = \sum_{i=1}^{n} x_i^2$$

- Rosenbrock's saddle. A devilish two dimensional function that has a curving valley which fools many optimization algorithms into halting prematurely and returning a point that is not the global optima.

$$f_2(\vec{x}) = \sum_{i=1}^{n-1} (100 \cdot (x_{i+1} - x_i^2)^2 + (x_i - 1)^2)$$

- Step. A five dimensional function that has long flat surfaces surrounded by discontinuities. Simple hill climbing algorithms often get stuck on these flat surfaces.

$$f_3(\vec{x}) = 6 \cdot n + \sum_{i=1}^{n} \lfloor x_i \rfloor$$

- Quartic. A thirty dimensional function in which evaluation is modified by Gaussian noise. Thus, two evaluations of the same point typically return different values.

$$f_4(\vec{x}) = \sum_{i=1}^{n} i x_i^4 + \text{gauss}(0, 1)$$

- Shekel's foxholes. A two dimensional function that has many local minima.

$$\frac{1}{f_5(\vec{x})} = \frac{1}{K} + \sum_{j=1}^{25} \frac{1}{c_j + \sum_{i=1}^{2} (x_i - a_{ij})^6}$$

$$(a_{ij}) = \begin{pmatrix} -32 & -16 & 0 & 16 & 32 & -32 & \cdots & 0 & 16 & 32 \\ -32 & -32 & -32 & -32 & -32 & -16 & \cdots & 32 & 32 & 32 \end{pmatrix}$$

$$K = 500 \qquad c_j = j$$

Diagrams of two dimensional versions of all five of these test functions are given in Figure 4. These functions were used to compare the optimization capabilities of DHC to those of genetic algorithms (as implemented by Bäck's GENEsYs package) and to simulated annealing (as implemented by Ingber and Rosen's Adaptive Simulated Annealing (ASA) program). For all the functions the genetic algorithm (GA) used a population of 50 individuals and proportional selection. All of the parameters in Ingber and Rosen's adaptive simulated annealing algorithms were set to their defaults with the exception of NO_COST_TEMP_TEST which was set to true. This parameter prevents ASA from halting when the temperature drops below a certain level. For an introduction to genetic algorithms see Sergio Margarita's "The Towers of Hanoi: A New Approach" (AI Expert, Mar. 1993) and for an introduction to simulated annealing see the article by Lissa W. Light and Peter Anderson ("Designing Better Keyboards", AI Expert, Sept. 1993).

Dynamic hill climbing has four user settable parameters: the objective function, the initial starting point, and the maximum and minimum step sizes. We typically set the maximum step size to half of the search range. We have found that for some functions the minimum step size greatly affects DHC's performance. If the parameter is set too low, then the inner loop will spend an excessive amount of time evaluating points that are very close to the best point. If the parameter is set too high, then the gradient vectors will not be able to adjust to the curvature of valleys and the inner loop will get stuck.

For these experiments the smallest step size was set to the precision used by the genetic algorithm for all the functions except Rosenbrock function. For this function, the smallest step size was decreased until the inner loop was able to find points close to the global optima without calling on the outer loop. We recommend the following protocol for setting the smallest step size. Begin with the step size equal to the desired precision. If it fails, decrease it by a factor of two until DHC's performance is satisfactory. We are in the process of building a shell around DHC that will automatically set the step size.

The results of this comparison are given in Figure 5 which shows the evaluation of the best point averaged over ten runs as a function of the number of evaluations. Of the three optimization techniques, the genetic algorithm fared the worst, consistently underperforming both dynamic hill climbing and adaptive simulated annealing. The optima that the genetic algorithm found were worse and it took more function evaluations to find them. Dynamic hill climbing outperforms adaptive simulated annealing on four of the five functions. DHC, unlike both the genetic algorithm and adaptive simulated annealing, consistently finds the global optimum.

(A) (B)

(C) (D)

(E)

Figure 4: Diagrams of DeJong's test functions in two dimensions. In order to accentuate the important features of the functions, they have been multiplied by -1 and modified by order preserving transformations. (A) Sphere. (B) Rosenbrock's saddle. (C) Step. (D) Quartic. (E) Shekel's foxholes.

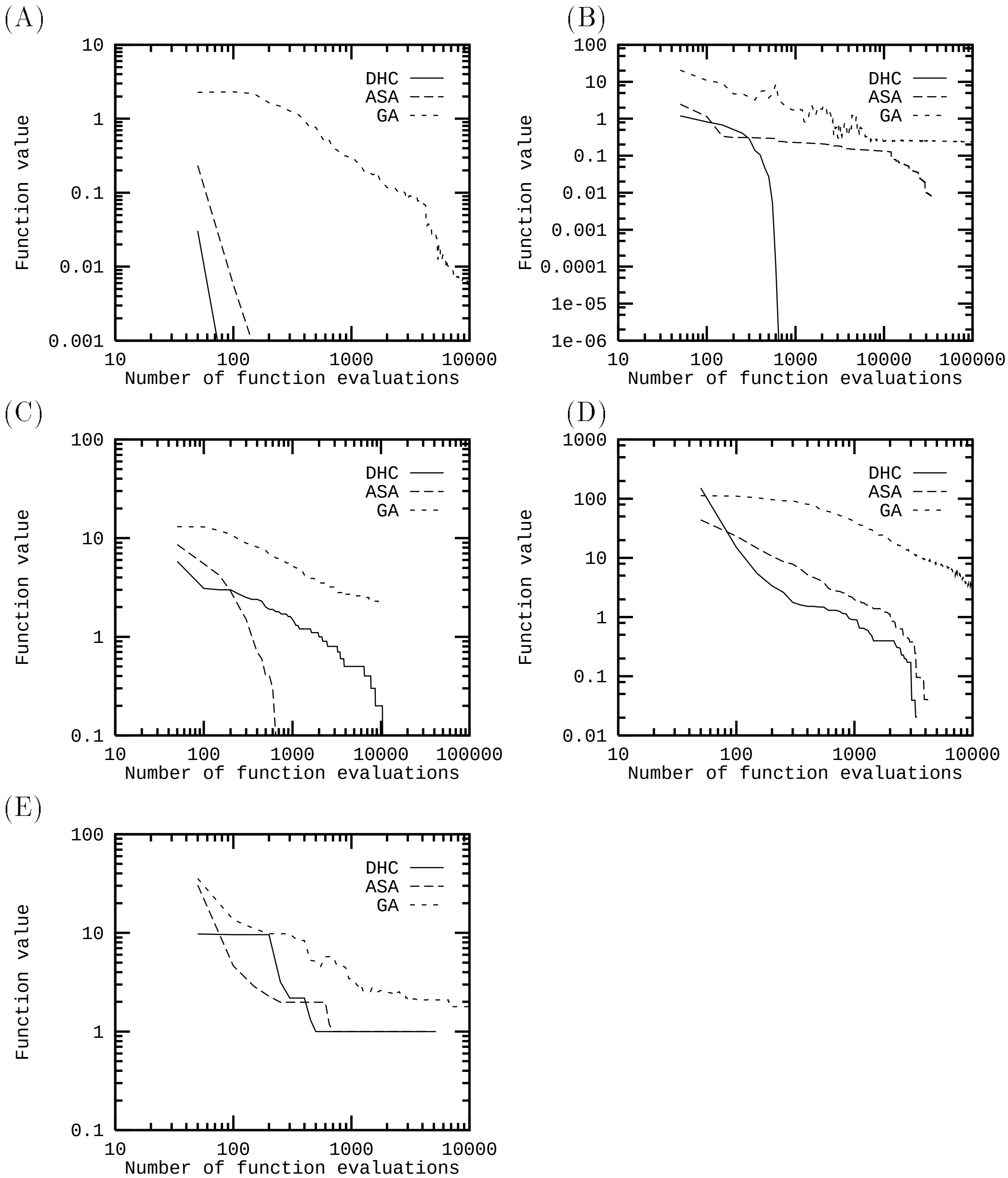

Figure 5: Comparison of genetic algorithm and simulated annealing to DHC. Each graph shows the average evaluation of the best point over ten runs as a function of the number of evaluations. (A) Sphere. (B) Rosenbrock's saddle. (C) Step. (D) Quartic. (E) Shekel's foxholes.

Figure 6: Alignment of MRI with laser data.

4 Medical imaging

DHC has been incorporated as part of a medical imaging system that helps brain surgeons plan and execute surgeries. As part of this system DHC performs a six degree coordinate change transformation that matches information from a CT (computer tomography) scan or MRI (magnetic resonance image) to data gathered from a laser scanning device. The CT scan or MRI is taken prior to the brain surgery and is used to plan the operation. When the patient is on the operating table, a laser scanning device is used to determine the patient's head position. DHC aligns these two models to within 2mm root mean squared distance, on average. Figure 6 shows an alignment of an MRI scan (orange skull) with the laser scan data (blue dots).

DHC replaced a multi-start version of Powell's method, an optimization technique that uses conjugate directions and one-dimensional line optimizations to find global optima. On one test problem DHC was able to find a better optima than Powell's method using only 15% as many function evaluations.

5 Limitations

We have yet to find a continuous optimization problem on which DHC underperforms genetic algorithms (challenges welcome!) and it is also usually better than adaptive simulated annealing. However, it is not clear how to generalize the DHC procedure to solve optimization problems, such as the traveling salesman problem, in which the search space is not a subset of Cartesian space.

References

[1] T. Bäck and F. Hoffmeister. A user's guide to *GENEsYs 1.0*. Software package documentation, 1992.

[2] K. De Jong. *Analysis of the Behavior of a Class of Genetic Adaptive Systems*. PhD thesis, University of Michigan, Ann Arbor, 1975.

[3] L. Ingber and B. Rosen. Genetic algorithms and very fast simulated re-annealing: A comparison. 1991.

[4] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes in C: The Art of Scientific Computation*. Cambridge University Press, 1992.

[5] D. Yuret and M. de la Maza. Dynamic hill climbing: Overcoming the limitations of optimization techniques. In *Second Turkish Symposium on Artificial Intelligence and Artificial Neural Networks*, pages 208–212, 1993.

Appendix

```lisp
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; DHC.LISP              ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Inner Loop            ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; The following is an implementation of the dynamic hill climbing algorithm
;;; in LISP.  The DHC procedure expects four arguments:
;;;
;;; >> func: Function from float vectors to float values to be minimized.
;;; >> xini: A float vector specifying the initial starting point.
;;; >> vmin: A float vector specifying the minimum step size in each dimension.
;;;          This determines the precision of the search.
;;;          Using a finer grained search usually helps but is more expensive.
;;; >> vmax: A float vector specifying the maximum step size in each dimension.
;;;          This determines how fast DHC can move in the search space.
;;;          When DHC gets stuck in a maximum it will try expanding its steps
;;;          to find a way out.  VMAX also determines when to stop trying.
;;;
;;; To achieve scale independence, DHC represents its steps in units of vmin.
;;; The step array has the form: (u0 u1 u2 ... ud -ud ... -u1 -u0); where
;;;   d is the number of dimensions,
;;;   ui (0 < i <= d) is the step vector in the i'th dimension
;;;   u0 is the incrementally adjusted gradient vector
;;;   -ui is the step in the opposite direction to ui
;;; A unit step corresponds to an actual step of vmin in the search space.
;;;
;;; When DHC gets stuck at an optimum, it returns the best point it has found.
;;; This might be a local optimum.  Thus a global search strategy should be
;;; used in conjunction with DHC for more reliable results.  DHC should be used
;;; for efficient local optimization.
;;;
;;; Order of function evaluations made by DHC in the sphere function is
;;;   O(DIMS*LOG(DIST)); where
;;;   DIMS is the dimensionality of space
;;;   DIST is the distance of initial point to global optimum in vmin units
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                                                ;;;
(defun dhc                                      ;;;
    (&key                                       ;;; >> PARAMETERS:
     (func #'sphere)                            ;;; target function
     (xini #(4.0 -5.0))                         ;;; initial starting point
     (vmin #(0.01 0.01))                        ;;; vector of minimum step size
     (vmax #(5.12 5.12))                        ;;; vector of maximum step size
     &aux                                       ;;; >> LOCAL VARIABLES:
     (dims (length xini))                       ;;; dimensionality of space
     (imax (+ (* 2 dims) 1))                    ;;; index of last step vector
     (fini (funcall func xini))                 ;;; value at initial point
     (smax (concatenate '(vector single-float)  ;;; the array of max step size
             (map 'list #'/ vmax vmin)          ;;; is ratio of vmax to vmin
             (list -1.0))))                     ;;; plus a -1 for extra vector
  (do                                           ;;; >> STATE VARIABLES:
   ((stuck nil)                                 ;;; flag set when stuck
    (xbst (copy-seq xini)) (fbst fini)          ;;; best point and its value
    (xcur (copy-seq xini)) (fcur fini)          ;;; current point and its value
    (step (init-step smax))                     ;;; array of int step vectors
    (xstep (init-step smax))                    ;;; step before last motion
    (gradv (v0 dims)) (gradi -1)                ;;; gradient vec and its index
    (last (v0 dims))                            ;;; last step taken
    (i -1))                                     ;;; index into step
   ((progn                                      ;;; >> TERMINATION TEST:
      (if stuck                                 ;;; if stuck flag set
          (setf i (argmin step smax))           ;;;   choose the smallest step
          (setf i (argmax step)))               ;;;   else choose the largest
      (= i -1))                                 ;;; if no more steps
    xbst)                                       ;;;   return the best point
                                                ;;; >> MAIN LOOP:
    (map-into xcur                              ;;; set xcur to the
              #'(lambda (x y z) (+ x (* y z)))  ;;;   best point plus the
              xbst (aref step i) vmin)          ;;;   step vector normalized
    (setf fcur (funcall func xcur))             ;;; set fcur to value at xcur
    (cond                                       ;;;
      ((< fcur fbst)                            ;;; >> IF NEW POINT BETTER
       (if stuck (setf stuck nil))              ;;; clear the stuck flag
       (replace xbst xcur) (setf fbst fcur)     ;;; update best point & val
       (dotimes (j (1+ imax))                   ;;; since we have moved
         (replace (aref xstep j) (aref step j)));;;   record step in xstep
       (map-into (aref step (- imax i))         ;;; set opp step to -curr/2
                 #'-half (aref step i))         ;;;   not to retry same point
       (when (equalp (aref step i) last)        ;;; if last step repeated
         (map-into (aref step i)                ;;;   set the current step
                   #'double (aref step i))      ;;;   to double the size
         (replace (aref xstep i) (aref step i)));;;   update xstep as well
       (replace last (aref step i))             ;;; record the last step
       (cond                                    ;;; Update the grad vector:
         ((= gradi -1)                          ;;;   if gradv is empty
          (replace gradv (aref step i))         ;;;     set gradv to curr step
          (setf gradi (min i (- imax i))))      ;;;     record the index
         ((= gradi (min i (- imax i)))          ;;;   else if curr vec is ||
          (map-into gradv                       ;;;     add the curr vec
                    #'+ gradv (aref step i)))   ;;;       to the gradv
         (t                                     ;;;   else update the extra vec
          (map-into (aref step dims)            ;;;     set extra step to
                    #'+ (aref step i) gradv)    ;;;       gradv + curr vec
          (setf (aref smax dims)                ;;;     set smax to the max of
            (max (aref smax (min i (- imax i))) ;;;       curr vec smax
                 (aref smax gradi)))            ;;;       and gradv smax
          (map-into (aref step (1+ dims))       ;;;     set the opposite step
                    #'- (aref step dims))       ;;;       to minus extra step
          (setf gradi -1))))                    ;;;     erase the gradv
      (stuck                                    ;;; >> ELSE IF ALREADY STUCK
       (map-into (aref step i)                  ;;; double the current step
                 #'double (aref step i)))       ;;;
      ((>= (size (aref step i)) 1.0)            ;;; >> ELSE IF STEP >= VMIN
       (map-into (aref step i)                  ;;; half the current step
                 #'half (aref step i)))         ;;;
      (t                                        ;;; >> ELSE IF STEP TOO SMALL
       (setf stuck t)                           ;;; set the stuck flag
       (dotimes (j (1+ imax))                   ;;; reset the step vectors to
         (map-into (aref step j)                ;;;   double the last recorded
                   #'double (aref xstep j)))))));;;   step vectors
                                                ;;;
                                                ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; Following is a list of definitions of the procedures called by DHC.
;;; >> SQ        : returns the square of a number
;;; >> V0        : takes # of dims and returns a zero vector
;;; >> SIZE      : calculates the size of a vector
;;; >> DOUBLE    : multiplies a number by two
;;; >> HALF      : divides a number by two
;;; >> -HALF     : minus the half of a number
;;; >> INIT-STEP : initializes step to unit vectors
;;; >> ARGMIN    : returns the minimum sized step within bounds in a step array
;;; >> ARGMAX    : returns the maximum sized step in a step array
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defun sq (x) (* x x))

(defun v0 (d) (make-array d :element-type 'single-float :initial-element 0.0))

(defun size (vec) (sqrt (apply #'+ (map 'list #'sq vec))))

(defun double (x) (* x 2))

(defun half (x) (/ x 2))

(defun -half (x) (- (half x)))

(defun init-step (smax)
  (let* ((dims (1- (length smax)))
         (imax (1+ (* 2 dims)))
         (step (make-array (* 2 (1+ dims)))))
    (dotimes (i (length step) step)
      (setf (aref step i) (v0 dims))
      (cond ((< i dims) (setf (aref (aref step i) i) (aref smax i)))
            ((<= i (1+ dims)))
            (t (setf (aref (aref step i) (- imax i))
                     (- (aref smax (- imax i))))))))))

(defun argmin (step smax)
  (let ((nstep (length step))
        (imin -1) (min MOST-POSITIVE-SINGLE-FLOAT))
    (dotimes (i nstep imin)
      (let ((si (size (aref step i))))
        (if (and (<= si (aref smax (min i (- nstep i 1)))) (< si min))
            (setf min si imin i))))))

(defun argmax (step)
  (let ((nstep (length step))
        (imax -1) (max 0.0))
    (dotimes (i nstep imax)
      (let ((si (size (aref step i))))
        (if (> si max)
          (setf max si imax i))))))
```