A DISCRETE BINARY VERSION OF THE PARTICLE SWARM ALGORITHM

James Kennedy[1] and Russell C. Eberhart[2]

[1]Bureau of Labor Statistics
Washington, DC 20212
kennedy_jim@bls.gov

[2]Purdue School of Engineering and Technology
Indianapolis, IN 46202-5160
eberhart@engr.iupui.edu

ABSTRACT

The particle swarm algorithm adjusts the trajectories of a population of "particles" through a problem space on the basis of information about each particle's previous best performance and the best previous performance of its neighbors. Previous versions of the particle swarm have operated in continuous space, where trajectories are defined as changes in position on some number of dimensions. The present paper reports a reworking of the algorithm to operate on discrete binary variables. In the binary version, trajectories are changes in the probability that a coordinate will take on a zero or one value. Examples, applications, and issues are discussed.

1. INTRODUCTION

The particle swarm algorithm (Eberhart and Kennedy, 1995; Eberhart, Dobbins, and Simpson, 1996; Kennedy and Eberhart, 1995; Kennedy, 1997) has been introduced as an optimization technique for use in real-number spaces. A potential solution to a problem is represented as a particle having coordinates x_{id} and rate of change v_{id} in a D-dimensional space. Each particle i maintains a record of the position of its previous best performance in a vector called p_{id}. An iteration comprises evaluation of each particle, then stochastic adjustment of v_{id} in the direction of particle i's best previous position and the best previous position of any particle in the neighborhood. Neighborhoods can be defined in innumerable ways: most implementations evaluate a particle i in a neighborhood consisting of itself, particle i-1, and particle i+1, with arrays wrapped so i=1 is beside i=N. The variable g is assigned the value of the index of the particle with the best performance so far in the neighborhood. Thus in the original version particles move by the following formula:

$$v_{id} = v_{id} + \varphi(p_{id} - x_{id}) + \varphi(p_{gd} - x_{id})$$

where φ is a random positive number generated for each id, whose upper limit is a parameter of the system, and:

$$x_{id} = x_{id} + v_{id}$$

The particle swarm algorithm has been found to be robust in solving problems featuring nonlinearity and nondifferentiability, multiple optima, and high dimensionality through adaptation which is derived from social-psychological theory (Kennedy, 1997).

2. DISCRETE SPACE

Many optimization problems are set in a space featuring discrete, qualitative distinctions between variables and between levels of variables. Typical examples include problems which require the ordering or arranging of discrete elements, as in scheduling and routing problems. Besides these pure combinatorial problems, researchers frequently cast floating-point problems in binary terms, and solve them in a discrete number space. As any problem, discrete or continuous, can be expressed in a binary notation, it is seen that an optimizer which operates on two-valued functions might be advantageous.

The particle swarm works by adjusting trajectories through manipulation of each coordinate of a particle. At least some of the success of the algorithm in real numbered functions appears to derive from the fact that it "overflies" known local optima, exploring beyond as well as between them. The immediate question then is, what are the meanings of concepts such as trajectory, velocity, between, and beyond, in a discrete space.

In a binary space, a particle may be seen to move to nearer and farther corners of the hypercube by flipping various numbers of bits; thus, velocity of the particle overall may be described by the number of bits changed per iteration, or the Hamming distance between the particle at time t and at t+1. A particle

0-7803-4053-1/97/$10.00 © 1997 IEEE

with zero bits flipped does not move, while it moves the "farthest" by reversing all of its binary coordinates. This does not answer the question, however, of what corresponds to v_{id} in a binary function, that is, what is the velocity or rate of change of a single bit or coordinate.

The solution to this dilemma is to define trajectories, velocities, etc., in terms of *changes of probabilities* that a bit will be in one state or the other. Thus, a particle moves in a state space restricted to zero and one on each dimension, where each v_{id} represents the probability of bit x_{id} taking the value 1. In other words, if $v_{id} = 0.20$, then there is a twenty percent chance that x_{id} will be a one, and an eighty percent chance it will be a zero. If the previous best positions have had a zero in that bit, then $(p_{id} - x_{id})$ can be reasonably calculated as -1, 0, or +1, and used to weight the change in probability v_{id} at the next step.

In sum, the particle swarm formula:

$$v_{id} = v_{id} + \varphi(p_{id} - x_{id}) + \varphi(p_{gd} - x_{id})$$

remains unchanged, except that now p_{id} and x_{id} are integers in {0, 1} and v_{id}, since it is a probability, must be constrained to the interval [0.0, 1.0]. A logistic transformation $S(v_{id})$ can be used to accomplish this last modification. The resulting change in position then is defined by the following rule:

if (rand() < $S(v_{id})$) then $x_{id} = 1$;
else $x_{id} = 0$

where the function $S(v)$ is a sigmoid limiting transformation and rand() is a quasirandom number selected from a uniform distribution in [0.0, 1.0].

The continuous-valued particle swarm algorithm also limited v_{id} by a value V_{max} which was a parameter of the system. In the discrete version V_{max} is retained, that is, $|v_{id}| < V_{max}$, but as can be seen, this simply limits the ultimate probability that bit x_{id} will take on a zero or one value. For instance, if V_{max}=6.0, then probabilities will be limited to $S(v_{id})$, between 0.9975 and 0.0025. The result of this is that new vectors will still be tried, even after each bit has attained its best position. Specifying a higher V_{max}, e.g., 10.0, makes new vectors less likely. Thus part of the function of V_{max} in the discrete particle swarm is to set a limit to further exploration after the population has converged; in a sense, it could be said to control the ultimate mutation rate or temperature of the bit vector. Note also that, while high V_{max} in the continuous-valued version increases the range explored by a particle, the opposite occurs in the binary version; smaller V_{max} allows a higher mutation rate.

3. A CHANGE IN THE MEANING OF CHANGE OF CHANGE

The revision of the particle swarm algorithm from a continuous to a discrete operation may be more fundamental than the simple coding changes would imply. The floating-point algorithm operated through a stochastic change in the rate of change of position. The original

$$x_{id} = x_{id} + v_{id}$$

seems very different from

if (rand() < $S(v_{id})$) then $x_{id} = 1$;
else $x_{id} = 0$.

The difference of course lies in the interpretation of the concept of the "change of change" that is central to the continuous-valued version. In the original, trajectories were adjusted by altering the velocity or rate of change of position, as distinguished from previous algorithms where the actual position of the particle is adjusted (Goldberg, 1989). In the present discrete version, however, it appears that with v_{id} functioning as a probability threshold, changes in v_{id} might represent a change in first-order position itself. As $S(v_{id})$ approaches zero, for instance, the "position" of the particle fixes more probably on the value 0, with less chance of change.

Trajectory in the current model is probabilistic, and velocity on a single dimension is the probability that a bit will change; thus even if v_{id} should remain fixed, the "position" of the particle on that dimension remains dynamic as it flips polarity with probability following from the value of v_{id}.

The probability that a bit will be a one = $S(v_{id})$, and the probability that it will be a zero = 1 - $S(v_{id})$. Further, if it is a zero already, then the probability that it will change = $S(v_{id})$, and if it is a one the probability it will change = 1 - $S(v_{id})$. Thus we can see that the probability of the bit changing is given by:

$$p(\Delta) = S(v_{id})\,(1 - S(v_{id})),$$

which is equivalent to saying

$$p(\Delta) = S(v_{id}) - S(v_{id})^2$$

which is the absolute (nondirectional) rate of change for that bit given a value of v_{id}. Thus change in v_{id} is still a change in the rate of change, with a new interpretation.

4. REPRESENTATIONS AS PROBABILITIES

In the previous version of the particle swarm algorithm, as in other evolutionary computation

paradigms, individuals is a population represent potential problem solutions. Individuals are evaluated, changed in some way or replaced, and tested again; adaptation occurs as the fitnesses of individual problem solutions increase.

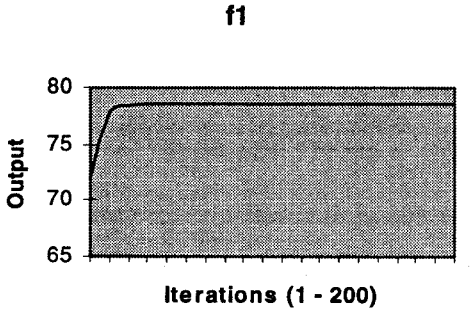
In the present method, population members are not viewed as potential solutions: they are probabilities. The value of v_{id} for each dimension determines the probability that a bit x_{id} will take on one or the other value, but x_{id} itself does not have a value until it is evaluated: the actual position of the particle in D dimensional space is ephemeral. An individual particle i with a particular vector v_{id} might have a different position x_{id} at every iteration. In this way the particle swarm differs from any optimization paradigm currently known.

5. IMPLEMENTATION IN A CLASSIC TESTBED

De Jong's (1975) suite of five test functions was designed to test a carefully constructed set of dimensions of performance of an optimizer. Code for these five functions was provided through the Internet by William Spears[1], and altered for integration with the present paradigm. In some cases it appears that the optimum can only be approximated within several decimal places, due to the imprecision of the binary encoding.

The functions were run twenty times each, with a population of 20 and V_{max} set at 6.0. Graphs show the mean best performance, measured on every tenth iteration.

As seen in Figure 1, the particle swarm converged quickly on the optimum for f1. The target output value was 78.60, though in any trial the closest the particle swarm was able to come was 78.599998, which it found on ten of the twenty trials. It is presumed that the difference between the found optimum and the target is due to insufficient precision in the binary encoding, rather than a failure of the algorithm to hit the target.



On the second function, f2, the particle swarm was able to attain a best value of 3905.929932, compared to a given target of 3905.93; again, the difference is thought to derive from the precision of the encoding, rather than the algorithm.

F2 is the hardest of the functions for the particle swarm. The system converged on the best known optimum four times in this set of twenty. The hardness of the function might be explained by the existence of very good local optima in regions which are distant from the best known optimum. For instance, the local optimum:

010111111101111000000111

returns a value of 3905.929688, while the vector:

110111101001110111011111

returns 3905.929443, and

111000011001011001000001

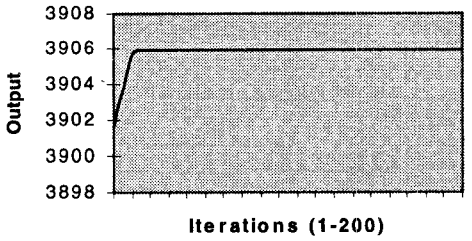
returns 3905.924561.

The best known optimum, returning 3905.929932, is found at:

110111101110110111101001,

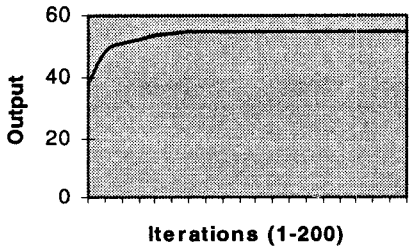
which is a Hamming distance of 12 (in a 24-element bit string) from the first local optimum given above, 7 from the second (rather close), and 15 from the third. Thus, there is very little chance that a search would move from one of the local optima to the global optimum. This kind of problem is probably better handled by a genetic algorithm with one- or two-point crossover, where long blocks of the parent vector are swapped in their entirety into the child.

f2



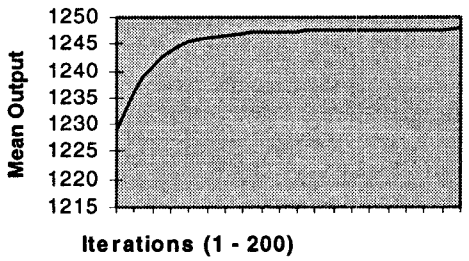
The third function, f3, is an integer function with a target value of 55, which was attained on every trial, as shown in figure 3.

f3



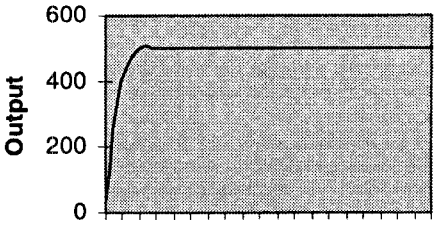
De Jong's f4 function introduces gaussian noise to the function, and was measured as an average over the entire population, rather than a population best.

f4



Finally, on f5 the algorithm was able to attain a best value of 499.056335, compared to a target value of 500.0, on twenty out of twenty attempts. As seen in Table 5, the system converged rapidly on this value.

f5



6. DISCUSSION

The De Jong testbed is considered to be a reputable and thorough suite of functions for testing the robustness of an optimization algorithm. The results reported in the preceding section demonstrate that the binary particle swarm implementation is capable of solving these various problems very rapidly. The new version of the algorithm extends the capabilities of the continuous-valued one; the present version is able to optimize any function, continuous or discrete.

The five functions were implemented in a single program, where the only code changed from one function to another was the evaluation function. All other aspects of the program, including parameter values, ran identically on the various functions. Thus it appears that the binary particle swarm is extremely flexible and robust.

The most difficult functions were the first two, and apparently for the same reason. Unlike evolutionary algorithms, the particle swarm has "memory" of past successes, and tends to converge upon regions of the search space that have afforded success previously. There is no mechanism for catastrophic leaps from one region to another. Evolutionary algorithms featuring crossover, however, combine aspects of the parents in a way that allows leaps. In cases such as neural networks, where many unique global optima exist, this feature of genetic algorithms works against them: two networks may be better than any combination of their parts. The particle swarm performs especially well on those kinds of problems. On the other hand, where a leap from one region to a distant other region is necessary, crossover is probably the preferred operator. Note that the particle swarm performed quite well on these first two functions, consistently return very good

evaluations, but it had trouble jumping out of good local optima.

7. REFERENCES

De Jong, K. A. (1975). *An analysis of the behavior of a class of genetic adaptive systems.* Doctoral dissertation, University of Michigan.

Eberhart, R. C., and Kennedy, J. (1995). A new optimizer using particle swarm theory. *Proc. Sixth Intl. Symposium on Micro Machine and Human Science* (Nagoya, Japan), IEEE Service Center, Piscataway, NJ, 39-43.

Eberhart, R. C., Dobbins, R. C, and Simpson, P. (1996). *Computational Intelligence PC Tools.* Boston: Academic Press.

Kennedy, J., and Eberhart, R. C. (1995). Particle swarm optimization. *Proc. IEEE Intl. Conf. on Neural Networks* (Perth, Australia), IEEE Service Center, Piscataway, NJ, IV: 1942-1948.

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

Kennedy, J. (1997). The particle swarm: Social adaptation of knowledge. *Proc. IEEE Intl. Conf. on Neural Networks* (Indianapolis, Indiana), IEEE Service Center, Piscataway, NJ, 303-308.

---

[1]C function code kindly provided by William Spears at http://www.aic.nrl.navy.mil/~spears/functs.dejong.html