Discrete Multi-Valued Particle Swarm Optimization

Jim Pugh and Alcherio Martinoli
Swarm-Intelligent Systems Group
École Polytechnique Fédérale de Lausanne
1015 Lausanne, Switzerland
Email: {jim.pugh,alcherio.martinoli}@epfl.ch

***Abstract*— We present a new optimization technique based on the Particle Swarm Optimization algorithm to be used for solving problems with unordered discrete-valued solution elements. The algorithm achieves comparable performance to both the previous modification of PSO for binary optimization and to genetic algorithms on a standard set of benchmark problems. Performance is maintained on these problems re-encoded with ternary solution elements. The algorithm is then tested on a simulated discrete-valued unsupervised robotic learning problem and obtains competitive results. Various potential improvements, modifications, and uses of the algorithm are discussed.**

I. Introduction

Discrete optimization is a difficult task common to many different areas in modern research. This type of optimization refers to problems where solution elements can assume one of several discrete values. The most basic form of discrete optimization is binary optimization, where all solution elements can be either 0 or 1, while the more general form is problems that have solution elements which can assume n different unordered values, where n could be any integer greater than 1. While Genetic Algorithms (GA) are inherently able to handle these problems, there has been no adaption of Particle Swarm Optimization able to solve the general case.

Particle swarm optimization (PSO) is a promising new optimization technique developed by James Kennedy and Russell Eberhart [3] [4] which models a set of potential problem solutions as a swarm of particles moving about in a virtual search space. The method was inspired by the movement of flocking birds and their interactions with their neighbors in the group. Every particle in the swarm begins with a randomized position (x_i) and (possibly) randomized velocity (v_i) in the n-dimensional search space, where $x_{i,j}$ represents the location of particle index i in the j-th dimension of the search space. Candidate solutions are optimized by flying the particles through the virtual space, with attraction to positions in the space that yielded the best results. Each particle remembers at which position it achieved its highest performance $(x^*_{i,j})$. Every particle is also a member of some neighborhood of particles, and remembers which particle achieved the best overall position in that neighborhood (given by the index i'). This neighborhood can either be a subset of the particles (local neighborhood), or all the particles (global neighborhood). For local neighborhoods, the standard method is to set neighbors in a pre-defined way (such as using particles with the closest array indices as neighbors modulo the size of the swarm, henceforth known as a "ring topology") regardless of the particles' positions in the search space. The equations executed by PSO at each step of the algorithm are:

$$\begin{aligned} v_{i,j} &= w \cdot v_{i,j} + pw \cdot rand() \cdot (x^*_{i,j} - x_{i,j}) \\ &\quad + nw \cdot rand() \cdot (x^*_{i',j} - x_{i,j}) \\ x_{i,j} &= x_{i,j} + v_{i,j} \end{aligned}$$

where w is the inertia coefficient which slows the velocity over time to prevent explosions of the swarm and ensure ultimate convergence, pw is the weight given to the attraction to the previous best location of the current particle and nw is the weight given to the attraction to the previous best location of the particle neighborhood. $rand()$ is a sampling of a uniformly-distributed random variable in $[0, 1]$.

The original PSO algorithm can only optimize problems in which the elements of the solution are continuous real numbers. A modification of the PSO algorithm for solving problems with binary-valued solution elements was also developed by the creators of PSO [5]. The equations for the modified algorithm are given by:

$$\begin{aligned} v_{i,j} &= v_{i,j} + pw \cdot rand() \cdot (x^*_{i,j} - x_{i,j}) \\ &\quad + nw \cdot rand() \cdot (x^*_{i',j} - x_{i,j}) \\ x_{i,j} &= 1 \text{ if } (rand() < S(v_{i,j})) \\ x_{i,j} &= 0 \text{ otherwise} \end{aligned}$$

where $S(x)$ is the sigmoid function given by

$$S(x) = \frac{1}{1 + e^{-x}}$$

Because it is not possible to continuously "fly" particles through a discrete-valued space, the significance of the velocity variable was changed to instead indicate the probability of the corresponding solution element assuming a value of 0 or 1. The velocity is updated in much the same way as in standard PSO, though no inertia coefficient is used here. For assigning a new particle value, the velocity term is transformed to the range $(0, 1)$, and the particle element is randomly set with probability of picking 1 given by $S(v_{i,j})$. In this variation of the algorithm, the velocity term is limited to $|v_{i,j}| < V_{max}$, where V_{max} is some value typically close to 6.0. This prevents the probability of the particle element assuming either a value of 0 or 1 from being too high. Though the discrete-value modification for PSO (henceforth refered to as DPSO) has

been shown to be able to optimize various discrete-valued problems [5], it is limited to discrete problems with binary-valued solution elements.

There has been some other exploration of PSO techniques for discrete optimization. Yang et al. [14] developed an algorithm based on DPSO which uses a different method to update velocity. Al-kazemi and Mohan [1] used a technique also based on DPSO which had particles alternatively exploit personal best and neighborhood best positions instead of simultaneously. Both methods use the same principles as DPSO and both are limited to binary problems. Pampara et al. [9] solved binary optimization using Angle Modulation with only four parameters in continuous PSO, which allowed for faster optimization of several problems. The technique was again only applied to binary problems.

Several other forms of discrete optimization have been explored using PSO. Permutation optimization has been attempted in several cases (e.g., [8], [10], [13]). The case study for most of these works has been the Traveling Salesman Problem. This type of optimization, while it does use discrete particle elements, is fundamentally different than the discrete optimization we deal with here. PSO has also been applied to some integer programming problems [11]. Our optimization differs from this because our discrete parameters are unordered, which makes it impossible to simply discretize a continuous search space as was done for integer programming.

Section 2 of this paper introduces our new discrete-valued PSO algorithm, which we designate Multi-Valued Particle Swarm Optimization (MVPSO). Section 3 compares its performance against GA and DPSO on the standard De Jong Test Suite [2]. Section 3 explores the capabilities of the algorithm for solving problems with discrete-valued solution elements encoded in ternary. Section 5 applies the algorithm to an unsupervised robotic learning problem with ternary solution elements. Section 6 discusses possible improvements and applications of the algorithm and concludes the paper.

II. Discrete Multi-Valued Particle Swarm Optimization

A. The Algorithm

In DPSO, the velocity term contains the probabilities of solution elements assuming values, and the particle contains the solution elements. For our algorithm, the particles themselves contain the probabilities of solution elements assuming values. Our particle representation therefore goes from being 2-dimensional to 3-dimensional: $x_{i,j,k}$ is a continuous real-value indicator of the probability of particle i, element j assuming value k (in the case of binary, k would be either 0 or 1). To find a particle's fitness, the solution elements are generated probabilistically at evaluation time, making the evaluation inherently stochastic. Because particle terms are real-valued, this representation allows us to use velocity in the same way as in standard PSO, where $v_{i,j,k}$ represents the velocity of $x_{i,j,k}$.

B. Generating the Solution Elements

At the time of fitness evaluation for particle i, we must transform the real-valued terms $x_{i,j,0}, ..., x_{i,j,n}$ to generate a value from 0 to n for solution element s_j. To do this, we apply the sigmoid function to each term, and use the weighted sum as the probability:

$$x'_{i,j} = \sum_{k=0}^{n} S(x_{i,j,k})$$

$$P(s_j = k) = \frac{S(x_{i,j,k})}{x'_{i,j}}$$

where $x'_{i,j}$ is the normalizing coefficient for particle i, element j. By using this technique, the particle can potentially generate any possible solution, but with varying probabilities depending on its terms.

C. Maintaining Centered Particle Values

The particle swarm maintains the best dynamics when all the particles share a common reference frame for their terms. This is always the case in standard PSO, as each particle term represents a solution element which is given a frame of reference by the fitness function. However, in our situation, the probability of a particular solution value k' is dependent not only on the probability term for that value $x_{i,j,k'}$ but on all other $x_{i,j,k}$ as well due to the probability normalization. We therefore apply an adjustment to particle terms after each modification of the particle values. This adjustment is given by

$$x'_{i,j,k} = x_{i,j,k} - c_{i,j}$$

for all k, where $x'_{i,j,k}$ is the new indicator of the probability of particle i, element j assuming value k, with $c_{i,j}$ chosen such that

$$\sum_{k=0}^{n} S(x'_{i,j,k}) = 1$$

The idea is to shift all the values such that we get a normalizing coefficient of 1. By applying this to all elements and particles, we have a common reference for every particle. This also accomplishes the normalization that would otherwise be required to generate the solution elements.

As there is no straightforward method of finding $c_{i,j}$ analytically, we calculate it by numerical approximation.

D. Dealing with Noise

Because particles in our algorithm contain probabilities of element values instead of actual element values, a particle can assume different element values on different evaluations and therefore return different fitness values. This situation is analogous to a noisy fitness evaluation, and we therefore ought to use a method of dealing with this noise. We apply the technique used in [12] for PSO with noisy optimization, where we reevaluate the previous best particles at each iteration of the algorithm and average all the evaluation results over the particles' lifetimes to get an average fitness. This modification

may also give our algorithm some intrinsic resistance to noisy fitness evaluations.

A flowchart of the algorithm can be seen in Fig. 1.

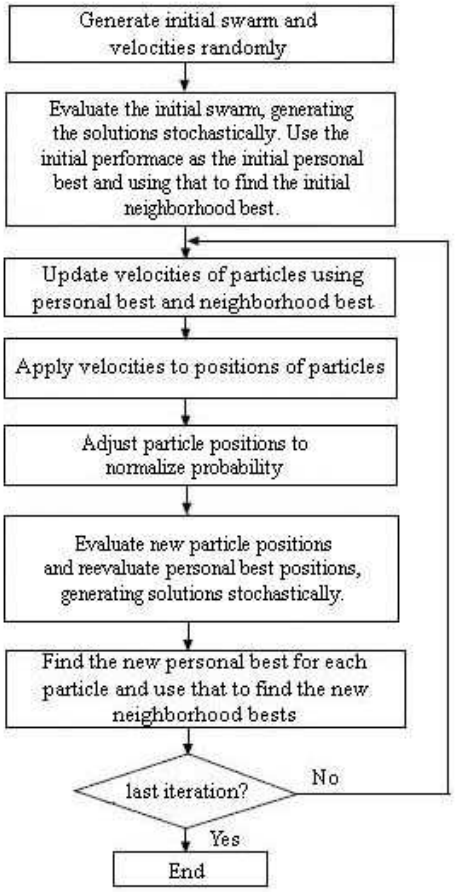

Fig. 1. Evolutionary optimization loop used by MVPSO

E. Limiting the Velocity

Because our technique differs from standard PSO, the impact of the inertia coefficient w was not immediately clear. By evaluation, we found the algorithm performed well for a w which linearly decreases from 1.2 at the start of the algorithm to 0.6 at the end, which is similar to the technique used in previous work with standard PSO. It would be interesting and useful to develop a technique for adapting the inertia coefficient throughout the evolution based on how the optimization of the system is progressing.

F. Algorithm Parameters

We use a ring topology for the swarm and assign the nearest particle on each side to be a neighbor. pw and nw are both set to 2.0.

III. Test Function Evaluation

A. Setup

We use the very standard De Jong Test Suite ([2]) with binary encoding to evaluate the performance of the algorithm. The functions are given by

$$
\begin{aligned}
f_1(\bar{x}) &= \sum_{i=1}^{n} x_i^2 \\
f_2(\bar{x}) &= 100(x_2^2 - x_1)^2 + (x_1 - 1)^2 \\
f_3(\bar{x}) &= 6n + \sum_{i=1}^{n} \lfloor x_i \rfloor \\
f_4(\bar{x}) &= \mathcal{N}(0,1) + \sum_{i=1}^{n} i x_i^4 \\
\frac{1}{f_5(\bar{x})} &= \frac{1}{500} + \sum_{i=1}^{n} \frac{1}{i + (x_1 - a_{imod5})^6 + (x_2 - a_{i/5})^6}
\end{aligned}
$$

for $a_0 = -32$, $a_1 = -16$, $a_2 = 0$, $a_3 = 16$, $a_4 = 32$. The function parameters are specified in Table I. For comparison, we use standard GA and DPSO (parameters given in Table II). All algorithms use 20 agents. For GA, the parameters were chosen to achieve near-maximal performance on the functions. For DPSO, we use the same parameters used in [5].

TABLE I
Test Function Parameters

Function	n	Bits per Value	Value Range
f_1	3	10	$[-5.12, 5.11]$
f_2	2	10	$[-5.12, 5.11]$
f_3	5	10	$[-5.12, 5.11]$
f_4	30	8	$[-1.28, 1.27]$
f_5	2	17	$[-65.5, 65.5]$

TABLE II
GA and DPSO Parameters

GA		DPSO	
Crossover Probability	0.6	pw, nw	2.0
Mutation Probability	0.2	$Vmax$	6.0

We run GA and DPSO for 400 iterations, except on f_5, where we used 800 iterations. Because our algorithm requires twice as many function evaluations due to the noise-resistant re-evaluations, we only run 200 iterations (400 iterations for f_5) in order to guarantee a fair comparison with an equal amount of computational processing between algorithms.

B. Results and Discussion

The final achieved values and standard deviations for all functions with all algorithms can been seen in Table III.

The progress of the best solutions of f_1 can be seen in Fig. 2. All algorithms manage to obtain low final values, though GA and DPSO have faster convergence and manage to achieve better performances, with DPSO doing slightly better in the end, though no algorithm consistently reaches zero. We

TABLE III
GA, DPSO, AND MVPSO PERFORMANCE (AND STANDARD DEVIATION) ON BINARY PROBLEMS

	GA	**DPSO**	**MVPSO**
f_1	0.00014 (0.00009)	0.00008 (0.00007)	0.00297 (0.00679)
f_2	0.27285 (0.41788)	0.10702 (0.17433)	0.02943 (0.04690)
f_3	0.00000 (0.00000)	0.00000 (0.00000)	1.28255 (0.73066)
f_4	7.13937 (2.19431)	2.52286 (1.08721)	16.50117 (5.64966)
f_5	0.17155 (0.14406)	0.03724 (0.08504)	0.00002 (0.00012)

suspect that the stochastic evaluation of MVPSO often keeps it from completely converging to the minimum value, while the mutation in GA and the deterministic evaluation in DPSO allow them to perform much better.

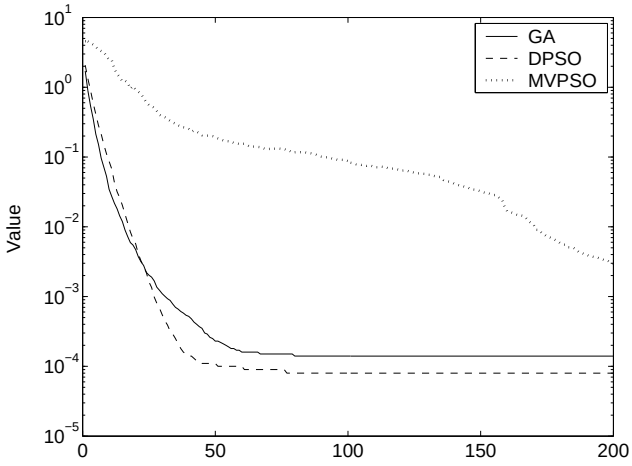

Fig. 2. Average of best solutions on f_1 over 100 runs for GA, DPSO, and MVPSO. Steps represent two iterations of GA and DPSO and one iteration of MVPSO.

The progress of the best solutions of f_2 can be seen in Fig. 3. Here, MVPSO manages to continue optimization through all 200 steps, and surpasses the performances of GA and DPSO partway through. The reason for the worse performances of GA and DPSO was that they would occasionally become stuck in a local minimum with much higher values. MVPSO managed to avoid this at almost every run of the algorithm. Again, DPSO is able to outperform GA.

The progress of the best solutions of f_3 can be seen in Fig. 4. Both GA and DPSO achieve near-zero values here, with DPSO converging slightly more quickly, while MVPSO, though it does converge, does not approach the optimum. The situation here is similar to that of f_1, and we suspect MVPSO's stochastic evaluation is again causing difficulties in convergence. However, it is interesting to see that the convergence rate of MVPSO actually increases towards the end of the run, when the inertia coefficient is lower. This suggests that low inertia values will encourage convergence

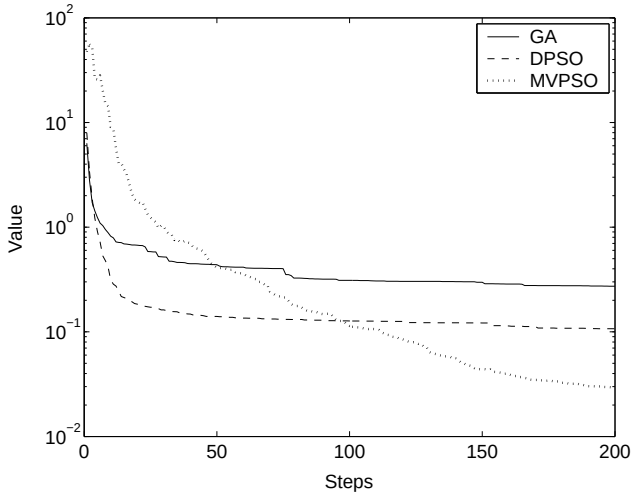

Fig. 3. Average of best solutions on f_2 over 100 runs for GA, DPSO, and MVPSO. Steps represent two iterations of GA and DPSO and one iteration of MVPSO.

in MVPSO, and a proper choice for inertia over time could allow this function to be optimized much more effectively.

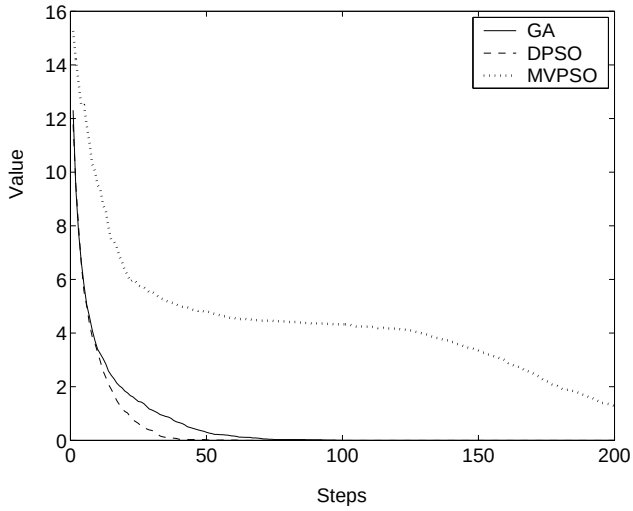

Fig. 4. Average of best solutions on f_3 over 100 runs for GA, DPSO, and MVPSO. Steps represent two iterations of GA and DPSO and one iteration of MVPSO.

The progress of the best solutions of f_4 can be seen in Fig. 5. The situation here is very similar to that of f_3: both GA and DPSO converge to very low values, while MVPSO does not do nearly as well. This function uses many more dimensions than any of the other functions, which could cause MVPSO to have even more convergence difficulties with its stochastic evaluation. The rate of convergence of MVPSO again increases in the latter part of the run, likely as a result of the inertia coefficient, and DPSO again converges further and more quickly than GA.

The progress of the best solutions of f_5 can be seen in Fig. 6. This function has a large number of local optima. Both GA and DPSO quickly converge to solutions, while MVPSO

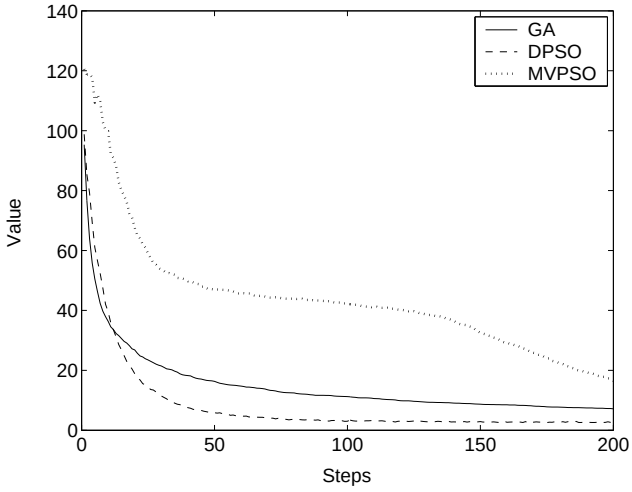

Fig. 5. Average of best solutions on f_4 over 100 runs for GA, DPSO, and MVPSO. Steps represent two iterations of GA and DPSO and one iteration of MVPSO.

converges much more slowly, but is able to eventually surpass the other two algorithms in the latter portion of the run. This suggests that MVPSO may be better able to escape local optima if the algorithm is given enough time. DPSO achieves better performance than GA yet again.

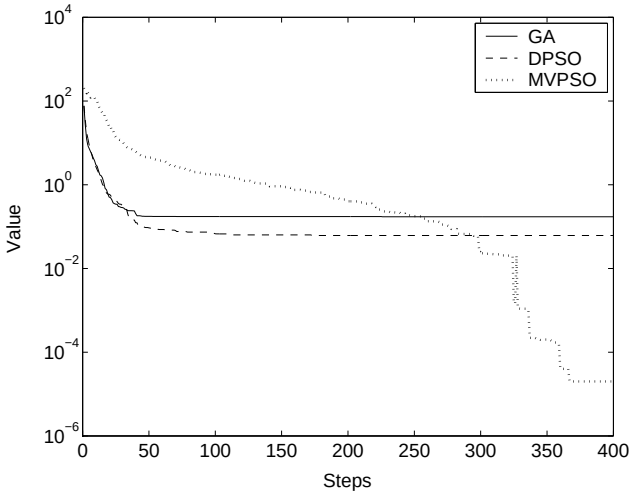

Fig. 6. Average of best solutions on f_5 over 100 runs for GA, DPSO, and MVPSO. Steps represent two iterations of GA and DPSO and one iteration of MVPSO.

All algorithms were able to converge for all functions. DPSO outperformed GA on every function, while MVPSO was able to eventually achieve better performance on f_2 and f_5. MVPSO seems to have some difficulties with with total convergence on several functions, but it seems better able to escape local minima than the other two algorithms. It appeared as though decreasing the inertia coefficient might have improved the performance of MVPSO on f_3 and f_4. Using an intelligently adaptive inertia coefficient in MVPSO might allow it to achieve much superior performance.

IV. Discrete Values Beyond Binary: Ternary

Up until now, we have only used MVPSO for optimization problems with binary-valued solution elements. We wish to compare the performance on problems with discrete-valued solution elements with more than two possibilities. The next logical encoding is ternary, where we convert the numerical representation of the function terms to base-3.

A. Setup

We compare the performances of GA and MVPSO on all functions from the De Jong Test Suite. However, we now encode the numbers using a ternary scheme instead of binary. This requires us to recalculate the number of bits using different function bounds. The new function parameters can be found in Table IV. Correspondingly, function f_3 is changed to

$$f_3(\bar{x}) = 4n + \sum_{i=1}^{n} \lfloor x_i \rfloor$$

to reflect the new minimum value of the terms. The number of iterations remain the same.

TABLE IV
Test Function Parameters: ternary

Function	n	Bits per Value	Value Range
f_1	3	6	$[-3.64, 3.64]$
f_2	2	6	$[-3.64, 3.64]$
f_3	5	6	$[-3.64, 3.64]$
f_4	30	5	$[-1.21, 1.21]$
f_5	2	11	$[-88.6, 88.6]$

B. Results and Discussion

The final achieved values and standard deviations for all functions with all algorithms can been seen in Table V.

TABLE V
GA and MVPSO Performance (and Standard Deviation) on Ternary Problems

	GA	MVPSO
f_1	0.00000 (0.00000)	0.01082 (0.01287)
f_2	0.05922 (0.10503)	0.01719 (0.02318)
f_3	0.00000 (0.00000)	0.46443 (0.57156)
f_4	0.44819 (0.67467)	24.30198 (7.79275)
f_5	5.06314 (4.87844)	0.45841 (0.83823)

The progress of the best solutions of f_1, f_3, and f_5 in ternary can be seen in Fig. 7, 9, and 8, respectively. We see similar results here to what was observed with binary encoding; both functions converge, with MVPSO ultimately surpassing GA in

f_2 and f_5. In f_1, GA manages to converge to zero in every run, something it was unable to do in binary encoding; this is likely the result of a local minima created by the binary encoding scheme, which is no longer present in ternary. In f_3 and f_4, we still see the convergence rate of MVPSO increase in the latter stages of the evolution, though to a lesser degree than with binary. This could indicate that the effect of the inertia coefficient may be damped as the number of possible discrete values increases. In f_5, both functions perform much more poorly than in the binary case; this may be due to the larger range of values used here, or as a result of new local optima that may have been introduced by this re-encoding scheme. Overall, we observe similar trends between GA and MVPSO here to what was observed in the binary case, with GA maintaining/improving its performance slightly more than MVPSO.

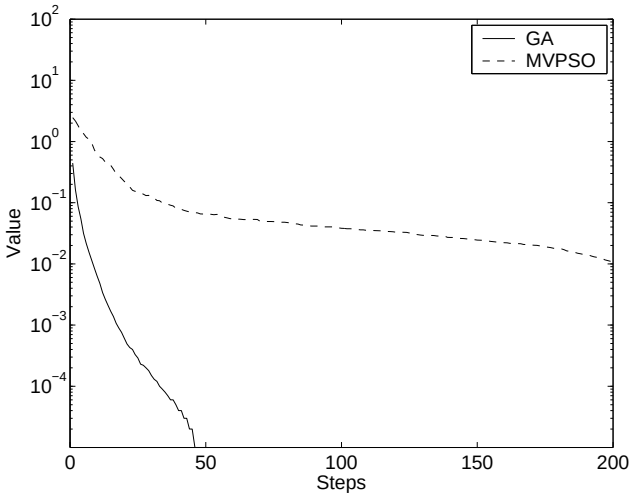

Fig. 7. Average of best solutions on f_1 in ternary over 100 runs for GA and MVPSO. Steps represent two iterations of GA and one iteration of MVPSO. The GA algorithm obtains a value of zero when it reaches the bottom of the graph.

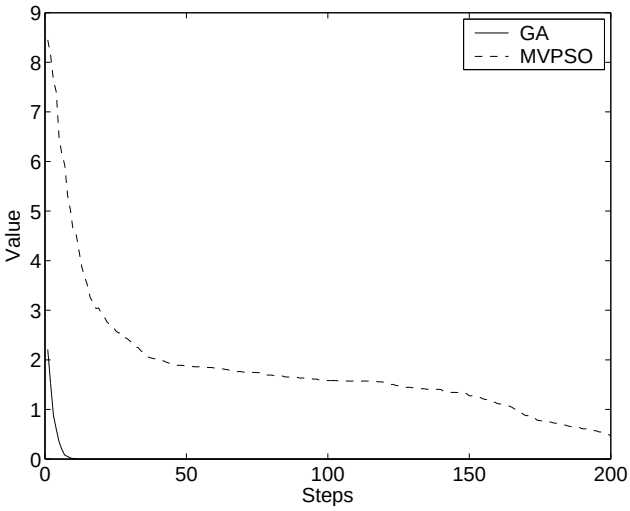

Fig. 8. Average of best solutions on f_3 in ternary over 100 runs for GA and MVPSO. Steps represent two iterations of GA and one iteration of MVPSO.

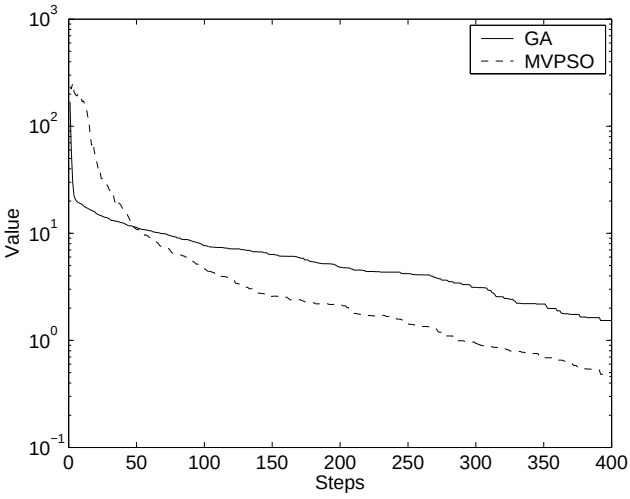

Fig. 9. Average of best solutions on f_5 in ternary over 100 runs for GA and MVPSO. Steps represent two iterations of GA and one iteration of MVPSO.

V. Discrete-Valued Unsupervised Robotic Learning

We further test Multi-Valued Particle Swarm Optimization by using it for simulated unsupervised robotic learning. The scenario is very similar to that explored in [12], where a single small-scale mobile robot must learn obstacle avoidance behavior using an Artificial Neural Network (ANN).

A. Setup

We use Webots, an embodied simulator, for our robotic simulations [6], using the Khepera robot model [7]. The Khepera has been modified in simulation to use twelve evenly spaced proximity sensors instead of the standard eight; this should give the robot better sensor coverage and not introduce a bias which might result from the uneven sensor spacing on the real Khepera (see Fig. 10). The robot operates in a 2.0 m x 2.0 m square arena.

The robotic controller is a single-layer discrete-time artificial neural network of two neurons, one for each wheel speed. However, we modify the ANN functionality from that which was used in [12]. We convert the proximity sensors from linear distance sensors to binary detection sensors (1 upon activation and 0 otherwise); these sensors are triggered when detecting an obstacle closer than some fixed distance (2.56 cm in our case with 10% noise) and are a common type of proximity detector in robotics. Each sensor can generate three possible effects on each motor neuron: positive, negative, or none. These correspond effectively to neural weight values of $+1$, -1, or 0. Additionally, each motor has a bias which is also of the same ternary form. The equation for neural output

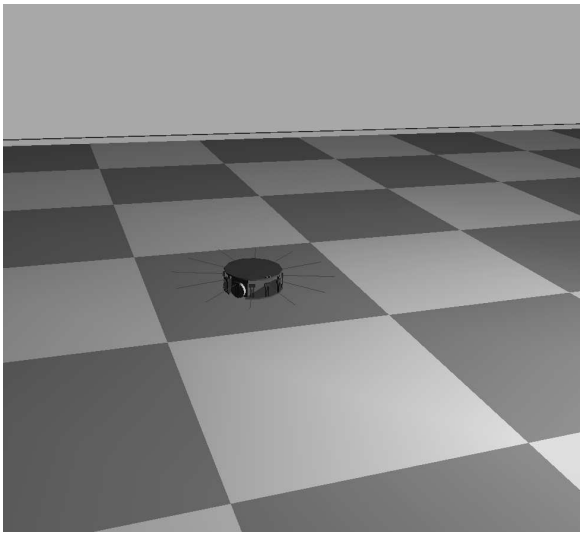

Fig. 10. Robot arena with Khepera robot. Rays represent proximity sensors.

then becomes:

$$\begin{aligned} s_n &= \theta_n + \sum_{x_i} w_{i,n} x_i \\ r(s_n) &= \begin{cases} 1, & \text{if } s_n > 0 \\ -1, & \text{if } s_n < 0 \\ 0, & \text{if } s_n = 0 \end{cases} \end{aligned}$$

where $r(s_n)$ is the output of neuron n, θ_n is the bias of that neuron, x_i is the activation value of sensor i, and $w_{i,n}$ is the corresponding ternary weight. The neuron therefore has a positive output for a positive total weight, a negative output for a negative total weight, and no output for an equal number of positive and negative inputs. A positive output corresponds to forward motor movement at the maximum speed, negative corresponds to backward movement at the maximum speed, and no output corresponds to no motion.

The total number of weights to be optimized is 26 (twelve sensor weights for each of the two motors, plus two bias weights). Slip noise of 10% is applied to the wheel speed. The time step for neural updates is 128 ms. We use the same fitness function as in [12]:

$$\begin{aligned} &F = V \cdot (1 - \sqrt{\Delta v}) \cdot (1 - i) \\ &0 \leq V \leq 1 \\ &0 \leq \Delta v \leq 1 \\ &0 \leq i \leq 1 \end{aligned}$$

where V is the average absolute wheel speed of both wheels, Δv is the average of the difference between the wheel speeds, and i is the average activation value of the most active proximity sensor over the evaluation period (for this calculation, we use the linear distance sensor values from [12] to have comparable fitness values). These parameters reward robots that move quickly, turn as little as possible, and spend little time near obstacles, respectively. The evaluation period of the fitness tests for these experiments is 240 steps, or approximately 30 seconds. Between each fitness test, a random speed is briefly applied to each of the robot's motors to ensure the randomness of the next evaluation.

Because unsupervised robotic learning is a noisy optimization task, we also run our experiment using the noise-resistant GA algorithm from [12], where at each iteration of the algorithm, the parent set of the population is re-evaluated and the worst performance is taken as the new fitness. This technique has been shown to improve performance on problems with noisy fitness and on unsupervised robotic learning.

We use the same algorithmic parameters here as for the previous experiments for both GA and PSO. The population size is set to 60 agents. The optimization is run for 200 iterations of GA and 100 iterations of MVPSO and noise-resistant GA to guarantee a fair comparison, since both require twice as many evaluations as standard GA.

B. Results

The results for the best evolved controllers can be seen in Fig. 11. Best controllers were selected by choosing the candidate solution with the best aggregate performance at the end of the evolution. This controller was then evaluated 30 times, and the final performance for that run taken as the average of these performances.

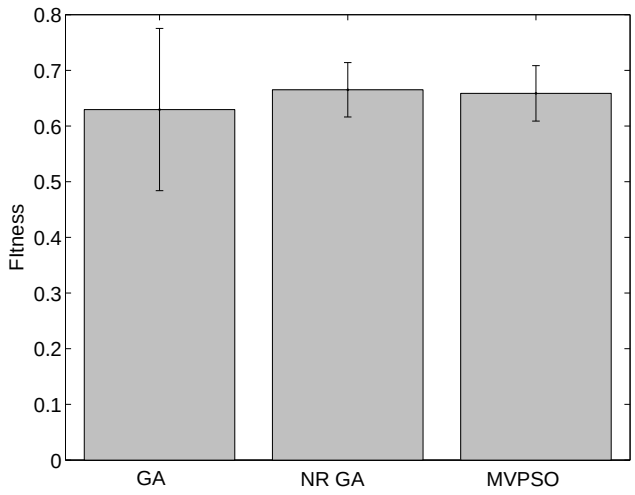

Fig. 11. Average performance of best evolved controllers over 50 runs for standard GA, noise-resistant GA, and MVPSO. Error bars represent standard deviation between runs.

MVPSO and noise-resistant GA both achieved very good performances here, while standard GA did slightly worse and had a much higher standard deviation. This shows that MVPSO can accomplish discrete-valued optimization beyond binary in a variety of scenarios. The intrinsic noise-resistance of MVPSO may have helped it overcome the noise inherent to the unsupervised robotic learning. The average perceived performance of the candidate solutions throughout the evolution for all algorithms can be seen in Fig. 12; this metric shows how well the algorithm believes its candidate solutions perform, and we can see that standard GA has a false perception of the

quality of its solutions due to the lack of reevaluation, while noise-resistant GA and MVPSO are much more accurate. As in the other cases, both GA algorithms initially converge much more quickly than MVPSO, but MVPSO continues to improve after progress stops for the GA algorithms.

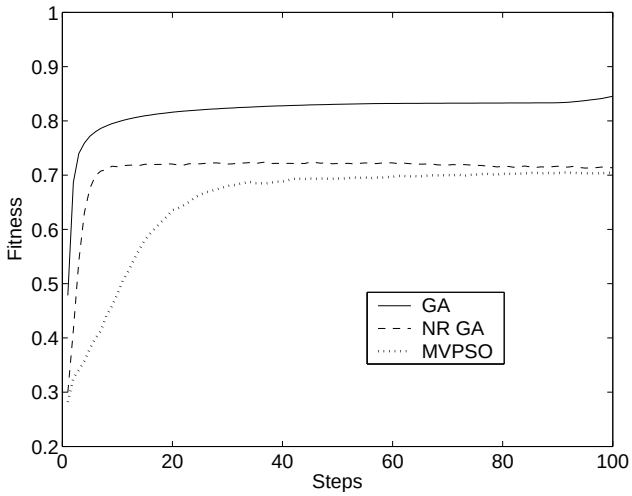

Fig. 12. Average perceived performance of population throughout evolution averaged over 50 runs for standard GA, noise-resistant GA, and MVPSO. Standard GA falsely perceives a very high performance for its controllers while noise-resistant GA and MVPSO have a much more accurate perception.

VI. Conclusion and Outlook

We have presented a new PSO-based algorithm capable of optimizing problems with discrete multi-valued solution elements. The algorithm's performance is competitive with those of GA and DPSO. The algorithm continues to function well for discrete-valued solution elements in ternary form, and successfully accomplishes discrete-valued unsupervised robotic learning.

There are several aspects of MVPSO which could be interesting to explore or examine in more detail:

Calculating the Solution Element Probabilities - the technique we use for calculating probabilities here is very heuristic (i.e. use of sigmoid for squashing, linear probability adjustment). There may be better performing methods of accomplishing this.

Improved Noise-resistance - the noise-resistance technique used to compensate for the stochastic evaluation of particles is very simple; it could be expanded to better choose when more fitness evaluations are needed and more intelligently combine multiple fitness evaluations. This would not only improve the general functionality, but also improve the algorithm's resistance to external noise when dealing with stochastic fitness functions.

Adjusting the Inertia Coefficient - we saw evidence that using different inertia coefficients could allow MVPSO to optimize more quickly and effectively. This effect should be explored in detail and, if possible, an adaptive inertia coefficient scheme should be developed which gives the best performance in all situations.

Possibilities/Limitations for Discrete-Valued Optimization Beyond Binary - there were indications that the performance of the algorithm decreased as the number of discrete values increased. This effect should be further explored in different scenarios.

Combining Standard PSO and MVPSO - there is no reason why a PSO particle could not be a hybrid of continuous values and discrete values by using both the standard PSO algorithm and MVPSO. This union could result in a very powerful algorithm, and should be explored.

Strengths and Weaknesses of PSO vs GA for Discrete Optimization - something which is still being analyzed for continuous optimization is which scenarios are more favorable for standard PSO and which are more favorable for GA. This will also need to be addressed for the discrete cases.

VII. Acknowledgements

Jim Pugh and Alcherio Martinoli are currently sponsored by a Swiss NSF grant (contract Nr. PP002-68647).

References

[1] Al-kazemi, B. & Mohan, C. K. "Multi-phase Discrete Particle Swarm Optimization" Fourth International Workshop on Frontiers in Evolutionary Algorithms, 2000.

[2] De Jong, K. A. "An analysis of the behavior of a class of genetic adaptive systems" Ph.D. dissertation, U. Michigan, Ann Arbor, 1975.

[3] Eberhart, R. & Kennedy, J. "A new optimizer using particle swarm theory" Micro Machine and Human Science, 1995. MHS '95., Proceedings of the Sixth International Symposium on, Vol., Iss., 4-6 Oct 1995, pages:39-43

[4] Kennedy, J. & Eberhart, R. "Particle swarm optimization" Neural Networks, 1995. Proceedings., IEEE International Conference on, Vol.4, Iss., Nov/Dec 1995, pages:1942-1948 vol.4

[5] Kennedy, J. & Eberhart, R. "A Discrete Binary Version of the Particle Swarm Algorithm" IEEE Conference on Systems, Man, and Cybernetics, Orlando, FA, 1997, pages:4104-4109

[6] Michel, O. "Webots: Professional Mobile Robot Simulation" Int. J. of Advanced Robotic Systems, 2004, pages:39-42, vol.1

[7] Mondada, F., Franzi, E. & Ienne, P. "Mobile robot miniaturisation: A tool for investigation in control algorithms" Proc. of the Third Int. Symp. on Experimental Robotics, Kyoto, Japan, October, 1993, pages:501-513

[8] Onwubolu, G. C. & Clerc, M. "Optimal operating path for automated drilling operations by a new heuristic approach using particle swarm optimisation", International Journal of Production Research, Vol. 42, No. 3, 2004, pp.473-491.

[9] Pampara, G., Franken, N., & Engelbrecht, A. P. "Combining Particle Swarm Optimisation with angle modulation to solve binary problems", IEEE Congress on Evolutionary Computing, Vol. 1, 2005, pp. 89-96.

[10] Pang, W., Wang, K., Zhou, C., & Dong, L. Fuzzy Discrete Particle Swarm Optimization for Solving Traveling Salesman Problem, Proceedings of the 4th International Conference on Computer and Information Technology (CIT04), IEEE Computer Society, 2004.

[11] Parsopoulos, K. E. & Vrahatis, M. N. "Recent approaches to global optimization problems through particle swarm optimization", Natural Computing 1, 2002, No. 2-3, pp. 235-306.

[12] Pugh, J., Zhang, Y. & Martinoli, A. "Particle swarm optimization for unsupervised robotic learning" Swarm Intelligence Symposium, Pasadena, CA, June 2005, pp. 92-99.

[13] Secrest, B. R. "Traveling Salesman Problem for Surveillance Mission using Particle Swarm Optimization", Thesis, School of Engineering and Management of the Air Force Institue of Technology, Air University , 2001.

[14] Yang, S., Wang, M. & Jiao, L. "A Quantum Particle Swarm Optimization", Congress on Evolutionary Computing, June 2004, Vol. 1, pp. 320-324.