Cluster-based Particle Swarm Algorithm for Solving the Mastermind Problem

Dan Partynski

***Abstract*—In this paper we present a metaheuristic algorithm that is inspired by Particle Swarm Optimization. The algorithm maintains a set of intercommunicating particle clusters and equips each particle with a specialized local search function. To demonstrate the effectiveness of the algorithm, we analyze its ability to solve Mastermind codes and compare its performance with other algorithms found in the literature. For the Mastermind problem, we have found that our algorithm is comparable to other algorithms for small problem sizes, but has much more efficient scaling behavior.**

***Index Terms*—particle swarm optimization (PSO), cluster, mastermind.**

I. Introduction

PARTICLE Swarm Optimization (PSO), an algorithm by Kennedy and Eberhart described in [1], is a popular method for optimizing continuous nonlinear functions. The algorithm is inspired by the interactions between agents in a flock of birds or a school of fish. Each agent initially begins in a random region of the search space, and more favorable regions are found iteratively as the agents begin to follow the best particle. While the algorithm was designed for continuous problems, various researchers have used the algorithm to solve various discrete problems, such as the Travelling Salesman Problem [2].

Mastermind is a code-breaking game for two players: an encoder and a decoder. The encoder selects a secret permutation of P digits from a list of N digits. There are no limits on the values of P and N, but during a single game these values are fixed. The decoder makes a series of guesses and for each guess gains information about how close the guess is to the secret code. When a guess is presented, the user receives a number of "dark pegs" which correspond to digits that have the correct value in the correct position. The user also receives a number of "light pegs" which correspond to the remaining digits that have the correct value but are in the incorrect position. Note that the dark pegs take precedence over the light pegs. For example, if the secret code is [1 2 2 4] and the guess is [1 4 3 2], then the decoder will receive one dark peg and two light pegs. We shall represent this as (1, 2).

Mastermind can be viewed as a dynamic-constraint optimization problem. The decoder continues to make guesses and receives more information about what the secret code can and cannot be. Every (*guess*, *response*) pair imposes an additional constraint on the decoder. This information can be used to form an optimization function, and as a result various researchers have attempted to solve Mastermind codes using metaheuristics such as genetic algorithms. As the game progresses, it is increasingly difficult to identify a guess that could still be the secret code. A potential guess is considered "consistent" if it could still be the secret code given the current list of constraints, and "inconsistent" otherwise.

The general goal of Mastermind is to break the secret code as quickly as possible. Thus there are two values that a Mastermind algorithm could minimize:

1) **Number of Guesses:** An algorithm's performance may be measured by the average of number of guesses needed to break the secret code. A *guess* is one submission of a code to the encoder.
2) **Number of evaluations:** An algorithm will analyze several potential codes before making a guess. We define an "evaluation" as one call of the fitness function that the algorithm uses to analyze a single code.

Minimizing the number of guesses presents an interesting challenge for large problem sizes, but it is not our main focus. Since we are interested in optimization, our algorithm will attempt to minimize the number of evaluations needed to break the secret code.

In this paper we describe a cluster-based particle swarm algorithm for solving the game of Mastermind. Our algorithm is not a direct application of PSO, rather it uses the underlying idea of communicating particles as inspiration. The algorithm is similar to PSO in that it maintains a group of communicating agents, but it organizes them into clusters. It also introduces a local search function that greedily moves each agent across neighbor states to more promising regions. This combination of local and global search functions allows for more efficient exploration of the search space. Later in this paper we will demonstrate that the algorithm is able to solve high-dimenionsal instances of the Mastermind problem.

Section II discusses the various Mastermind algorithms found in the literature. Section III describes the algorithm we used to solve the problem. Section IV discusses the experiments that we ran to test the algorithm's performance, and Section V concludes this paper.

II. Literature Review

The first Mastermind algorithm to appear in the literature was by Knuth [3] in 1976. This algorithm chooses the next guess that will minimize the maximum number of remaining possibilities. If there is a tie among potential guesses, the algorithm chooses the guess that is still consistent. Knuth tested this algorithm with P = 4 and N = 6, and was able to break the secret code in 4.478 guesses on average (5 in the worst case).

Kooi describes the "Most Parts Strategy" in [4]. This algorithm selects the guess that yields the highest number of possible responses. In the worst case this strategy takes 6

Manuscript received December 07, 2013; revised January 10, 2014.

Dan Partynski is a student of the Department of Mathematics and Computer Science, University of San Diego, San Diego, CA, 92110 USA e-mail: (dpartynski@sandiego.edu).

ISBN: 978-988-19252-5-1
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

guesses to break the secret code, but has a lower average of 4.373. These averages were obtained when P = 4 and N = 6.

Temporel and Kovacs describe an algorithm in [5] which attempts to minimize the number of evaluations required when P = 4 and N = 6. The algorithm requires 4.64 guesses on average which is slightly higher than the previous algorithms, but only evalutes 41.2 codes on average. However, no data is given on larger problem sizes.

These algorithms work well for small instances of Mastermind, but because they require either searching or storing the entire search space, they are computationally infeasible for large values of P and N. Bernier et al. describe the use of genetic algorithms and simulated annealing to solve larger instances of Mastermind [6]. Simulated annealing was better in terms of minimizing the number of evaluations, but the genetic algorithm resulted in a lower number of guesses on average.

Berghman et al. describe an improvement to the genetic algorithm in [7]. They use genetic algorithms to create a set of consistent guesses, and use an exhaustive strategy to select the best guess from the set. This algorithm is comparable to the full-enumeration strategies in terms of average number of guesses, but outperforms them in terms of speed. When P = 8 and N = 12, the algorithm can find the secret code in an average of 20.571 seconds.

Merelo et al. also use this approach in [8]. They use genetic algorithms to create a set of consistent guesses and use the "Most Parts" strategy to select one to use. This algorithm was tested with P = 5 and N = 9 and yielded 5.95 guesses and 38,485 evaluations on average.

III. Algorithm

Our algorithm always selects a guess that is consistent with the previous guesses. The general outline is as follows:

while *Secret Code not found* **do**
  Find a consistent guess g;
  Submit g get response r = (n_{dark}, n_{light});
  if n_{dark} = P **then**
    Secret Code has been found;
  end
end

Fig. 1. Mastermind Algorithm

We now will define a formula for a consistent guess so that we can phrase the Mastermind problem as an optimization problem. We first define a *rule* as a guess submitted by the decoder and the corresponding response from the encoder. More formally, we define a rule as a (*guess*, *response*) pair where *guess* is a vector of size P and *response* is a pair of integers (n_{dark}, n_{light}) representing dark pegs and light pegs.

Now given two guesses g_i and g_j we define a new function:

$$h(g_i, g_j) = (n_{dark}, n_{light}) \tag{1}$$

where n_{dark} is the number of dark pegs and n_{light} is the number of light pegs the decoder would receive if g_i was the secret code and g_j was submitted as a guess. We now define a measure of the similarity of two responses r_i and r_j as follows:

$$d(r_i, r_j) = abs(r_{i\_dark} - r_{j\_dark}) + abs(r_{i\_light} - r_{j\_light}) \tag{2}$$

Informally, the distance between two responses is 0 if they have identical dark and light peg counts. Otherwise the function returns the Manhatten distance between the two response vectors. Now assume the decoder has made n guesses and has received n responses, so we have (g_1, r_1), (g_2, r_2), ..., (g_n, r_n) as our *rule* list. We now define the fitness of some potential guess g as follows:

$$f(g) = \sum_{i=1}^{n} d(h(g, g_i), r_i) \tag{3}$$

We know that the secret code must match each of the given responses when compared to the given guesses. Thus for a guess to be consistent its fitness value must be 0. The higher the fitness value, the farther away a guess is from being consistent. Thus finding a consistent guess can be reduced to minimizing the above function. The following sections describe the algorithm used to achieve this minimization.

A. Local Function

Our algorithm maintains a collection of potential guesses, called agents, each equipped with a function that allows it to explore neighboring states. If the function to minimize was differentiable, then the agents could use gradient descent or some other fast optimization strategy as their local search function. Mastermind is a discrete problem, so we created a custom search function to move an agent across the search space. Given some guess g = [d_1 d_2 ... d_P], we define three possible ways to move to a neighboring state:

1) Replace some digit d_i with a different value
2) Take two digits d_i and d_j and swap them
3) Take three digits d_i, d_j, and d_k where i <j <k. Here there are two possible neighbor states. Swap so that the order is d_k d_i d_j or d_j d_k d_i. Notice that in each of these new states, every digit is in a new location. If this were not the case, the change could be made by only swapping two elements.

Based on these possible neighbor states, we define three different local functions. Local function (1) iteratively looks at each digit and finds the replacement digit that reduces the fitness value the most. If no replacement digit reduces the fitness, the function does not change the guess vector.

Local function (2) iteratively looks at each pair of digits and swaps them. If the fitness after the swap is less than or equal to the old fitness, the change to the vector is kept. Notice that unlike local function (1), we keep the resultant vector if the new fitness is equal to the old fitness rather than strictly less. This was found to be beneficial through experimentation.

Local function (3) iterates over each triplet of digits in a code and tests the fitness of both unique swaps. If at least one of the swaps results in a fitness that is less than or equal to the old fitness, the change will be kept. If both swaps lead to a reduction in fitness, the function chooses the swap that

ISBN: 978-988-19252-5-1
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

leads to the largest reduction. If each reduction is equal, one of the two swaps is chosen at random.

Note that all three of these functions exhaustively test each possible neighbor state. This was possible to do for the Mastermind problem, but for some problems this may be computationally infeasible. A possible solution in these cases is to only test the fitness of *k* random neighbors.

The agents in the search space need a way to use these functions to move towards more favorable regions of the search space. We decided to simply run these functions sequentially, only moving from function (*i*) to function (*j*) if function (*i*) failed to improve the fitness of *g*. Once function (*i*) improves the fitness of *g*, the algorithm moves back to function (1). If function (3) fails to improve the fitness of *g*, then the algorithm stops as it has converged on a local minimum. Here is a more formal description of the local search algorithm:

```
Given a guess g;
while Not at local minimum do
    Run function (1) to form g';
    if f(g') ≥ f(g) then
        Run function (2) to form g';
        if f(g') ≥ f(g) then
            Run function (3) to form g';
            if f(g') ≥ f(g) then
                Local minimum reached;
            end
        end
    end
end
```

Fig. 2. Local Search Function

B. Metaheuristic

The function described in the previous section often leads agents to a local minimum. Thus we introduce a metaheurtistic that uses the local search function to help locate the global minimum. Intuitively the idea is simple. We define *k* agent clusters $\{c_1, c_2, ..., c_k\}$ where each cluster contains *l* agents. Each cluster is initially given a cluster head, which is set randomly in the search space. Each cluster head then uses a *disperse* function to distribute (*l* - 1) agents around it up to some predefined max distance. Each agent in each cluster uses the local search function until all agents arrive at a local minimum. When this happens, we identify the fittest agent in each cluster and redistribute the remaining agents in that cluster around it. We then locate the fittest individual amongst all the clusters and use a *pull* function to move each agent in its direction. This allows communication between the clusters and moves large amounts of agents to favorable regions of the search space. Our final step is to take the fittest individual in each cluster and set it equal to a random vector, leaving the other agents in the cluster to explore the region that had been discovered. This was introduced primarily as a measure to increase diversity and perhaps allow an agent to move to a favorable region of the search space that no cluster is exploring.

Before giving a formal description of the algorithm, we need to define the *disperse* and *pull* functions that are required by the algorithm. Like the local search function, these functions are dependent on the optimization problem, so we will explain how the functions were defined for the Mastermind problem. The function *disperse*(*agent*) when given some agent will randomly move a new agent to a close region of the search space. The function *pull*($agent_i$, $agent_j$) will move $agent_i$ closer to $agent_j$.

We first give our defintion of the *disperse* function. Given some agent *a* = [d_1 d_2 ... d_P], we would like to produce *a*' by dispersing slightly from *a*. To do this, we will simply move *a* across some small number of neighbor states randomly. The number of neighbor states to move across is a random integer between two bounds: $dist_{min}$ and $dist_{max}$. A more formal description of the *disperse* function is as follows:

```
Given agent a;
dist := random integer ∈ [dist_min, dist_max];
for i ← 1 to dist do
    Choose random value r ∈ [0, 1);
    if r <.5 then
        Randomly swap two digits in a;
    else
        Change a random digit in a;
    end
end
```

Fig. 3. Disperse Function

This produces a new agent that is slightly farther away from the central agent. Note that for simplicity, our *disperse* function does not move to neighbors that require a triple swap. As an illustration, suppose we have the guess vector [1 2 3 4 5] and we wish to use the disperse function to create a new vector. If the random distance is 2, then the function may replace the second digit with the value 5 and swap the fourth and fifth digits. The resulting guess vector is [1 5 3 5 4], which is 2 neighbor states away from the original.

We now define our *pull* function. For this, we use an extra parameter called $dist_{pull}$.

```
Given agent_i and agent_j;
for i ← 1 to dist_pull do
    k := radom integer ∈ {1, 2, 3,...,P};
    k^th digit of agent_i := k^th digit of agent_j;
end
```

Fig. 4. Pull Function

This function takes digits from the best agent and places them into the corresponding digit of the agent being moved. The value *k* can be selected with or without replacement. In our implementation, we select *k* with replacement because we found through experimentation that this strategy produces slightly better results. As an example, suppose we wish to use the pull function to move the vector [1 2 3 4 5] in the direction of the vector [6 7 7 3 4]. Depending on the random values, the function may replace the third digit of the first vector with the third digit of the second vector, resulting in [1 2 7 4 5].

Now that we have defined these two functions, we give a more detailed description of the whole algorithm for finding a consistent guess:

ISBN: 978-988-19252-5-1
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

```
for i ← 1 to k do
    a_head := random agent;
    add a_head to c_i;
    for j ← 1 to (l - 1) do
        a_new := disperse(a_head);
        add a_new to c_i;
    end
end
while Sufficiently fit agent not found do
    for i ← 1 to k do
        for j ← 1 to l do
            Run local search function on a_ij;
            if f(a_ij) = 0 then
                return a_ij;
            end
            if f(a_ij) is lowest fitness found then
                a_best := a_ij;
            end
        end
    end
    for i ← 1 to k do
        a_best_i := best agent ∈ c_i;
        for j ← 1 to l do
            a_ij := disperse(a_best_i);
        end
        a_best_i := random agent;
    end
    for i ← 1 to k do
        for j ← 1 to l do
            pull(a_ij, a_best);
        end
    end
end
```

Fig. 5. Particle Cluster Search Algorithm

We will now summarize the parameters used by this algorithm:

1) *k*: The number of clusters
2) *l*: The number of agents in a cluster
3) $dist_{min}$: The minimum number of neighbor states to move across in the *disperse* function
4) $dist_{max}$: The maximum number of neighbor states to move across in the *disperse* function
5) $dist_{pull}$: The maximum number of neighbor states that $agent_i$ will move across toward $agent_j$ in the *pull* function

These parameters have a strong impact on the performance of the algorithm. Higher values of *k* and *l* are often better in terms of avoiding local minima, but lead to higher amounts of computation. The distance parameters are highly dependent on the size of the problem. As we describe in our experiment section, we do not perform extensive analysis to determine good values for these parameters, but rather use our judgement depending on the size of the Mastermind problem being solved.

C. Domain Heuristics

Domain-specific heuristics can greatly improve the convergence speed of the algorithm. Based on the reponses of the decoder, we are able to eliminate large poritions of the search space. We do this by maintaining a 2-dimensional array called *valid*[][], which has P rows and N columns. We say that *valid*[i][j] is *true* iff it's possible for a consistent guess to have the value *j* for d_i. If it is impossible, then *valid*[i][j] is *false*. Initially, all entries in the array are set to *true*. The *disperse* function and all local search functions avoid setting some digit d_i to some value *j* if *valid*[i][j] is *false*. Thus the more entires that are set to *false*, the more inconsistent guesses will be avoided by the algorithm.

The question then is how to determine if an array entry can be set to *false*. Our algorithm identifies three general categories of a response *r* = (n_{dark}, n_{light}) that allow us to alter this array:

1) Suppose n_{dark} = n_{light} = 0. This is the most informative case, since no digit in the guess *g* can appear in a consistent guess. Thus $\forall$ i $\in$ {1, 2, 3,...,P} and $\forall$ d_j $\in$ *g*, *valid*[i][d_j] = *false*
2) Suppose n_{dark} = 0 but n_{light} >0. In this case, a consistent guess cannot have any digit d_j in slot j because this would produce a dark peg. Thus $\forall$ d_j $\in$ *g* *valid*[j][d_j] = *false*
3) Suppose n_{dark} + n_{light} = P. In this case, the decoder has identified all of the digits that appear in the secret code, but not in the right order. Thus the secret code is a permutation of the digits in *g*, so all digits not in *g* can be eliminated. More formally, $\forall$ i $\in$ {1, 2, 3,...,P} and $\forall$ j $\in$ {1, 2, 3,...,N}, *valid*[i][j] = *false* iff j $\notin$ *g*

After condition (3) is met, we also flag a special variable called *endgame*. The *endgame* occurs after all of the digits have been identified and the algorithm simply needs to find a permutation of them. When this happens, the algorithm first runs local functions (2) and (3) on the most recent guess played in the hopes that only a few swaps are needed to produce a consistent guess. If this fails, then the algorithm proceeds as normal. We have found experimentally that this enhancement often does lead to the quick discovery of a consistent guess.

IV. Experiments and Results

To test the performance of the proposed algorithm, we ran several trials for several values of P and N. We used an Inspiron N4010 computer with a 2.43 GHz Intel i3 processor and 4 GB of RAM. The computer was equipped with the Windows 7 operating system. The programming language we chose was C++ for purposes of speed. Each agent is represented as a 64-bit integer, with each digit given 4 bits of information. Thus each digit can have a maximum of 16 posssible values.

The algorithm has many parameters, and each trial uses 5 clusters and 8 particles in a cluster. All other parameters varied depending on the problem size, and we used our best judgment to select appropriate values. We compare our results to the results of Berghman et al. in [7] and Merelo et al. in [8] by comparing the average number of guesses, average number of evaluations, and average time needed to break the secret code. We then demonstrate our algorithm's scaling behavior on problem sizes not yet found in the literature.

ISBN: 978-988-19252-5-1
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

A. P = 4 and N = 6

For this configuration, we used [0 0 1 1] as our first guess. We ran 10 trials for each possible secret code and averaged the results. As expected, our algorithm has a worse performance in terms of the average number of guesses. However, it is efficient in terms of the number of evaluations required to find the secret code.

TABLE I
COMPARISON OF ALGORITHMS FOR P = 4 AND N = 6

Algorithm	Guesses	Evaluations	Time
Berghman et al.	4.39	N/A	.614 sec
Merelo et al.	4.414	3899	N/A
New Algorithm	4.64	253	0.003 sec

B. P = 5 and N = 8

For this configuration we use [0 1 2 3 4] as our first guess and run 10000 trials, each with a randomaly generated secret code.

TABLE II
COMPARISON OF ALGORITHMS FOR P = 5 AND N = 8

Algorithm	Guesses	Evaluations	Time
Berghman et al.	5.618	N/A	N/A
Merelo et al.	5.619	19758	N/A
New Algorithm	5.95	1464	0.009 sec

C. P = 8 and N = 12

This is the largest configuration of Mastermind that we were able to find in the literature. We chose [0 0 1 1 2 2 3 3] as our first guess and ran 3000 trials with randomly generated secret codes. We compare our results to Berghman et al. We see that our algorithm exibits a higher guess average, but the time needed to break the secret code is significantly reduced.

TABLE III
COMPARISON OF ALGORITHMS FOR P = 8 AND N = 12

Algorithm	Guesses	Evaluations	Time
Berghman et al.	8.366	N/A	20.571 sec
New Algorithm	8.92	26149	0.18 sec

D. Higher Dimensions

To demonstrate that our algorithm can scale to higher dimensions, we present data on higher values of P and N. Due to our representation of Mastermind Codes as 64-bit integers, we limited P and N to a maximum of 16.

As we can see, the algorithm scales very well to higher dimensions of Mastermind. Observe that when P = N = 16, there are 16^{16} possible secret codes, which is approximately 10^{19}. This means that, on average, the algorithm evalutes only $\frac{1}{10^{13}}$ of the search space.

TABLE IV
PERFORMANCE FOR HIGH DIMENSIONS

(P, N)	Guesses	Evaluations	Time
(10, 12)	10.385	69057	.655 sec
(10, 16)	11.53	131925	1.32 sec
(11, 16)	12.43	195126	2.2 sec
(12, 16)	13.19	355021	4.41 sec
(13, 16)	14.19	402877	5.81 sec
(14, 16)	15.11	604736	9.95 sec
(15, 16)	16.03	1.003×10^6	18.54 sec
(16, 16)	17.26	1.557×10^6	33.70 sec

V. CONCLUSION

In this paper we presented a new algorithm inspired by PSO. The algorithm is similar to PSO in that it uses a swarm of agents to explore a search space, but it introduces particle clusters and a specialized local search function. We have shown that the algorithm works well for solving Mastermind codes compared to other algorithms found in the literature.

We plan to further analyze the parameters of the algorithm to determine good rules of thumb for parameter selection. We would also like to test the algorithm's performance on various other discrete optimization problems that are typically solved with genetic algorithms or other metaheuristics. This would require modifying the *disperse*, *pull*, and local search functions in order to suit the given optimization problem. We further plan to enhance the algorithm through parallelism by assigning each particle cluster its own processing core.

REFERENCES

[1] J. Kennedy, "Particle swarm optimization," in *Encyclopedia of Machine Learning*. Springer, 2010, pp. 760–766.
[2] X. H. Shi, Y. C. Liang, H. P. Lee, C. Lu, and Q. Wang, "Particle swarm optimization-based algorithms for tsp and generalized tsp," *Information Processing Letters*, vol. 103, no. 5, pp. 169–176, 2007.
[3] D. E. Knuth, "The computer as master mind," *Journal of Recreational Mathematics*, vol. 9, no. 9, pp. 1–6, 1976.
[4] B. P. Kooi, "Yet another mastermind strategy." *ICGA Journal*, vol. 28, no. 1, pp. 13–20, 2005.
[5] A. Temporel and T. Kovacs, "A heuristic hill climbing algorithm for mastermind," in *Proceedings of the 2003 UK Workshop on Computational Intelligence (UKCI-03)*, 2003, pp. 189–196.
[6] J. L. Bernier, C. I. Herráiz, J. Merelo, S. Olmeda, and A. Prieto, "Solving mastermind using gas and simulated annealing: a case of dynamic constraint optimization," in *Parallel Problem Solving from NaturePPSN IV*. Springer, 1996, pp. 553–563.
[7] L. Berghman, D. Goossens, and R. Leus, "Efficient solutions for mastermind using genetic algorithms," *Computers & operations research*, vol. 36, no. 6, pp. 1880–1885, 2009.
[8] J.-J. Merelo, A. M. Mora, and C. Cotta, "Optimizing worst-case scenario in evolutionary solutions to the mastermind puzzle," in *Evolutionary Computation (CEC), 2011 IEEE Congress on*. IEEE, 2011, pp. 2669–2676.

ISBN: 978-988-19252-5-1
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)