Cooperative PSO with Spatially Meaningful Neighbors

Peter Wilson

Supervised by Beatrice M. Ombuki-Berman

Submitted in partial fulfillment
of the requirements for COSC 4F90

Department of Computer Science

Brock University

St. Catharines, Ontario

Abstract

Contents

List of Tables

List of Figures

1 Introduction

2 Background

2.1 Particle Swarm Optimization

2.1.1 Overview

Particle Swarm Optimization (PSO) is a iterative population-based computational intelligence algorithm to find optimal solutions for non-linear continuous problems. Computational Intelligence is a domain of study in which the patterns and behaviours of intelligent agents (such as humans or animals) are used to help solve other types of problems [2]. Particle Swarm Optimization specifically draws from the flight patterns of birds in their search of food as well as the schooling pattern of fish. In both cases, the animals tend to follow each other from a distance which allows each to get a different vantage point. When one of the fish/birds find the best source of food (or the best place to land) they all tend to converge and head toward that point.

2.1.2 Particle

PSO builds off this metaphor by representing the birds or fish as Particles. A Particle has a position, velocity and personal best value. It keeps track of all these values and and updates them with each iteration. the position of the particle maps logically to the location of where the bird is on a map. The x,y,z,... values of the position also correlate to the input values in the problem which we are trying to optimize. The more input values (variables) the problem contains, the higher dimensions are needed to represent the problem. The initial position of the particle should be randomly generated to allow the swarm to effectively test the entire search space. If they all begin with similar solutions, it is unlikely that all potential solutions

are tested which leads to a higher possibility of converging on a local minimum as opposed to the best overall value. Velocity represents the direction and speed of the Particle. It determines how you update the position after each iteration. The velocity, too is updated at the end of each iteration and is influenced by the current velocity, the personal best value (or the best combination of values found previously by the particle which is maintained by the particle) and the global best (the best value found by all the particles in the swarm). The velocity and position are updated each iteration with the following functions.

Position Update Formula

$$x_i = x_i + v_i$$

Updating the position of the particle is as simple as taking the current position and adding the current velocity to it. For multi-dimensional problems where the particles have multiple dimensions, this addition will need to be done for each dimension. In this case, the i represents the index of the position.

Velocity Update Formula

$$v_i = \alpha v_i + \omega C_1(y_i - x_i) + \lambda C_2(\hat{y}_i - x_i)$$

The velocity update function has a little more to it. The new velocity is determined by a combination of it's current velocity, the distance it is from the particles personal best and the distance it is from the global best. In this case y_i represents the personal best for that index and $\hat{y}_i$ represents the global best. The influence that each has is determined by the inertial weight (α), the cognitive weight (ω) and social weight(λ) respectively. These values typically add up to 1 and determine the percentage of influence they have on the new velocity. Additionally, the formula also makes use of a random component. The values C_1

and C_2 are randomly generated values in the range [0,1] that adds more randomness and aids in keeping the swarm from converging too quickly.

2.1.3 Swarm

The Swarm is the heart that makes PSO work. It is the combination of a number of particles working in unison to solve the same problem. The Swarm needs to maintain the individual particles but it also needs to store a Global Best value. The global best is the position of the best solution found by any of the particles within the swarm. Each particle will in turn use that value to help influence the change in velocity for each iteration as seen in the velocity update formula above. This transfer of knowledge amongst the particles helps the swarm converge on the optimal solution. With each iteration, the swarm tries a number of different potential solutions and gains information to help it decide the overall best solution. The code for the swarm is relatively straight forward.

Table 1: PSO Pseudo Code

```
Initialize particles randomly
while current iteration < max iterations do
    for each particle j ∈ swarm do
        //Update the personal and global bests
        if f(j.x⃗) > f(j.y⃗) then
            j.(y⃗) ← j.(x⃗)
        end if
        if f(j.y⃗) > f(ŷ) then
            (ŷ) ← j.(y⃗)
        end if
    end for
    for all particles ∈ swarm do
        Update the Velocity
        Update the Position
    end for
end while
```

For every iteration, all the swarm needs to do it update the values of the personal best for each particle (represented by $\vec{y}$) and update the global best if a new best value is found.

This is accomplished by passing the solution set into the fitness function (see below) and getting a value. If you are working on a minimization problem, the smaller of the two will become the personal best, otherwise you will take the larger value. It then needs to iterate through each particle updating it's velocity and position. It is important to complete the global best update before updating the velocity since the global best has an influence on the new velocity. If the global best changes during an iteration, that new global best needs to be reflected in the velocity update in ALL particles, not just the particles that appear afterwards.

2.1.4 Fitness Function

In order to find the best (or optimal) solution to a problem, we first need a way to evaluate those solutions. The Fitness Function is responsible for taking the list of inputs that make up the solution and determining how well it fits the problem. It will then return a single value that helps the swarm determine if it is a good solution or a bad solution, ultimately helping it converge on the best value. The Fitness Function itself is the representation of the problem that we are looking to optimize and is how we tailor the Particle Swarm Optimization algorithm to solve different problems.

2.1.5 Parameters

There are a number of different parameters that can be altered to affect the success of the algorithm. Increasing or decreasing these values can lead to different results depending on the problem at hand. Experimentation is often required in order to get the best performance out of the algorithm.

Max Iterations: This helps determine how many iterations the swarm will go through in to search for the optimal solution. Increasing this value can potentially lead to better results however is meaningless if the swarm were to have already con-

verged to a local maxima.

Swarm Size: the size of the swarm determines the number of particles it employs to search for the optimal solution. Increasing this gives the swarm more information for each iteration and allows the swarm to have a higher likely hood of finding the overall best solution

Inertial Weight (α): the Inertial weight determines how much of an influence the current velocity has on future velocities. A large Inertial weight will lead to the particle carry on in it's current direction allowing it to better search the search space (though potentially heading off towards worse values). A low Inertial weight will allow it to converge towards the best values more quickly but may have a hard time searching the rest of the potential values

Cognitive Weight (ω): the Cognitive weight notes the effect the particles personal best solution has on the future velocity of the particle. Increasing this will keep the particle from venturing too far from this solution and only explore the neighboring space.

Social Weight (λ): this is called the social weight because it determines the effect other particles have on the current particle. This is the influence the global best (or the best solution currently found by the entire swarm) has on the future velocity of each particle. A low value will have the particles mostly work on their own, whereas a large social weight will have all particles converge toward the best value.

2.2 Co-operative Particle Swarm Optimization

Although Particle Swarm Optimization is a very successful algorithm for solving continuous problems with reasonable dimensions, it often struggles as those dimensions increase. If the problem has a large enough number of inputs, it can be difficult for the swarm to hone in on an optimal solution given the breadth of the search space size, often leading it to get stuck in local optima. This issue is often referred to as the "curse of dimensionality" and affects most other stochastic optimization algorithms [3].

Much of the impetus of this problem spawns from having all the dimensions update at the same time and then calculating the fitness of the function. This can result in a situation where a a new solution is objectively worse for most dimensions but leads to a better fitness value due to the a better value for a single dimension.[4] For instance, if the intention of the algorithm is to minimize the sum of the values (represented as $\sum_{n=0}^{d} x_n$) the position [5,5,5] will give a fitness value of 15, while the position [6,1,7] returns a better value of 14 even though the first and last dimension values are moving away from the optimal solution.[4] This can lead to the particles heading in the wrong direction, away from the optimal solution.

CPSO attempts to solve the "curse of dimensionality" by instead dividing the problem into multiple sub problems. This allows you to solve a subset of the dimensions separately by converting a single high dimensional problem into a number of low dimensional problems. With each iteration, the particles are then combined together to get a total fitness value. It is important to not update all these particles before gathering the fitness value as you will lead to the same issue as previously. There are a number of ways of doing this split which have been implemented into different variant of the algorithm. A problem which arises from this separation is that it removes the dependencies of variables. For instance, if two dimensions are required to work together to improve the fitness, they will be no longer able to do so directly if they are being updated with different swarms.

2.2.1 CPSO-S

The original version of CPSO is called CPSO-S. Developed by Van den bergh and Engelbrecht [3] it tackles the problem by dividing an n-dimensional problem in to n 1-dimensional swarms. It solves each swarm separately and then merges them to get the fitness value. It does this by creating a randomly generated solution and only updating the part that the current swarm is optimizing. It will continue using this randomly generated solution throughout the algorithm to ensure the other swarms do not affect the convergence of the current swarm. Apart from that, the rest is identical to the standard Particle Swarm Optimization algorithm, however on top of iterating through each particle per iteration, you also need to go through each swarm.

Table 2: CPSO-S Pseudo Code

```
create and initialize n one-dimensional PSO Swarms
Randomly initialize solution vector
while current iteration < max iterations do
    for each swarm i ∈ swarms do
        for each particle j ∈ i do
            //Update the personal and global bests
            if f(j.x⃗) > f(j.y⃗) then
                j.y⃗ ← j.x⃗
            end if
            if f(j.y⃗) > f(ŷ) then
                ŷ ← j.y⃗
            end if
            Update the Velocity
            Update the Position
        end for
    end for
end while
```

Since 1 dimensional problems are trivial to solve through Particle Swarm Optimization, separating the problem into 1-dimensional problems, it significantly simplifies the end goal. However, since each swarm operates almost in a vacuum, it has no idea of the values obtained by the other problems. This means if one value is dependent or affected by another, there

is no way of that being taken into account. This leads to the swarms getting stuck in a position that is locally optimal for each dimension but is not the global best solution to the problem.

2.2.2 Variant: CPSO-S$_k$

The first variant proposed by Van den bergh and Engelbrecht is CPSO-S$_k$. Instead of dividing the problem dimensions into n 1-dimensional swarms, it now evenly divides the problem into k swarms (or as close to even as possible). The basis of this variant is to attempt to combine dimensions that are dependent on each other in hopes to overcome getting stuck in local optima. The rest of the algorithm is identical to CPSO-S. It optimizes those multi-dimensional swarms separately and have them converge on a local optima, ultimately combining the individual solutions as it's final optimization value.

Table 3: CPSO-S$_k$ Pseudo Code

```
K_1 ← nmodK
K_2 ← K − (nmodK)
Initialize K_1⌈n/K⌉-dimensional PSOs
Initialize K_2⌊n/K⌋-dimensional PSOs
Randomly initialize solution vector
while current iteration < max iterations do
    for each swarm i ∈ swarms do
        for each particle j ∈ i do
            //Update the personal and global bests
            if f(j.x⃗) > f(j.y⃗) then
                j.(y⃗) ← j.(x⃗)
            end if
            if f(j.y⃗) > f(ŷ) then
                (ŷ) ← j.(y⃗)
            end if
            Update the Velocity
            Update the Position
        end for
    end for
end while
```

However, though this leads to a better chance of matching dependent dimensions, it still tends to get stuck in local optima. This is because the dimension split that exists in the CPSO solutions don't exist in the actual problem. By dividing the problem into sub problems, we are fundamentally altering the initial problem. It is also unlikely that dividing the algorithm into even sections will capture all the connected values and since there is no communication between swarms, it is impossible for the algorithm to break out of those local optima.

2.2.3 Variant: CPSO-R_k

One potential solution to the local optima problem would be to randomly divide the problem into differently sized sub problems. This leads to an even larger likelihood of combining dependant variables though that is entirely left up to chance, however, if it does successfully make those connections it can very quickly converge on the optimal solution. CPSO-R_k (also known as Random CPSO) tackles this methodology by dividing up the problem into k randomly sized PSOs. Once the swarms are created, it functions very similar to CPSO-S_k. It solves all the swarms separately, placing them into a temporary solution vector to get the fitness. Once the max number of iterations is reached, it will take the global best of each swarm and combine them into a single solution.

2.2.4 Variant: CPSO-H_k

CPSO-H_k (otherwise known as Hybrid CPSO) also attempts to solve the problem of local optima by also running a standard PSO algorithm in parallel to CPSO-S_k and have the two algorithms swap information. This aids in solving the problem of the dimension split by also updating those values in the PSO that is solving the original problem. The two algorithms work together using a knowledge-transfer strategy to share information. This requires taking the best overall value from one algorithm and putting it into the other. To do so, however, you need to then 'override' the value of an existing particle. Since we want to ensure this

knowledge swap leads to a better solution, the particle it overrides needs to follow a couple constraints. First, you cannot override the global best particle. Second, some problem sets disallow for 'Illegal Candidates', this means the new particle does not follow the constraints of the problem. In this case, the particle swap will only be accepted if the new particle is a legal candidate.

Table 4: CPSO-H$_k$ Pseudo Code

Initialize an n-dimensional PSO swarm
Generate a random integer k such that k mod n = 0,0¡k¡n
Initialize k [n/k]-dimensional PSOs
Randomly initialize solution vector
while current iteration < max iterations **do**
  for each swarm $i \in swarms$ **do**
    for each particle $j \in i$ **do**
      //Update the personal and global bests
      if $f(j.\vec{x}) > f(j.\vec{y})$ **then**
        $j.\vec{(y)} \leftarrow j.\vec{(x)}$
      end if
      if $f(j.\vec{y}) > f(\hat{y})$ **then**
        $\hat{(y)} \leftarrow j.\vec{(y)}$
      end if
      Update the Velocity
      Update the Position
    end for
  end for
  Transfer knowledge from CPSO-H$_k$ to PSO
  for each particle $j \in$ n-dimensional PSO **do**
    //Update the personal and global bests
    if $f(j.\vec{x}) > f(j.\vec{y})$ **then**
      $j.\vec{(y)} \leftarrow j.\vec{(x)}$
    end if
    if $f(j.\vec{y}) > f(\hat{y})$ **then**
      $\hat{(y)} \leftarrow j.\vec{(y)}$
    end if
  end for
  for all *particles* $\in$ n-dimensional PSO **do**
    Update the Velocity
    Update the Position
  end for
end while

By combining both PSO and CPSO-S_k, it combines the benefits of both algorithms. It takes advantage of CPSO's ability to more quickly converge on the optimum solution, while utilizing PSO's ability to escape from local optima.

2.3 Delaunay Triangulation

2.3.1 Polygon Triangulation

Triangulation in Computational Geometry is the decomposition of a polygon or set of points into a set of pairwise non-intersecting triangles in which all points in the set or edge in the polygon are an edge in at least one of newly the formed triangles. In n-dimensions, the triangles are instead represented as simplices (the n-dimensional equivalent). Triangulation is mainly used in computer graphics to simplify complex shapes into easily drawn triangles, though the information gathered from this process can be utilized in a number of different problems. An optimal triangulation is one that minimizes the total length of all the edges, otherwise known has having the minimum weight. What this means is, an optimal triangulation will, on average, connect all the points to their nearest neighbours. Definitively finding the optimal triangulation of a set of points is considered to be an NP-Hard problem, however, there are a number of triangulation algorithms that do a very good job of coming close to optimal.

2.3.2 Delaunay Algorithm

Delaunay Triangulation is considered to be one of the most efficient triangulation algorithms in computational geometry and often leads to optimal or close to optimal triangulations. The algorithms does this by creating a set of triangles such that the circumcircle of each triangle contains no other points. What this means is that if one were to draw a circle around every triangle where each point of the triangle lies on the circumference of that circle, no points will fall inside another triangles circle. This guarantees the the triangles are evenly spaces out and often results in a lower number of 'thin' triangles.

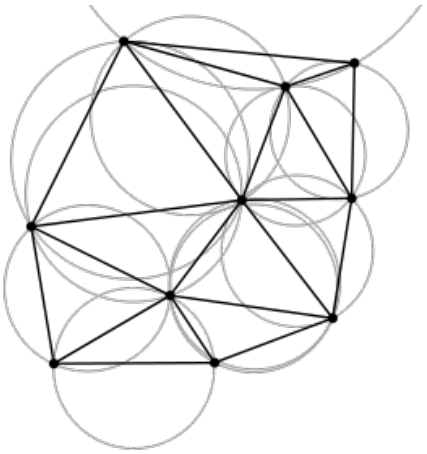

Figure 1: [1]

In 2-dimensions, Delaunay Triangulation has a worst-case time complexity of just O($nlogn$) also making it one of the quickest triangulation algorithms. As dimensions increase, however, the triangulation becomes more difficult to compute making it a less viable solution to this problem. 4-dimensions take O(n^3) time and d-dimensions takes O($n^{\lceil d/2 \rceil + 1}$) time.

2.4 Dynamic Connections

¡include the pseudocode for n-dimensions¿

3 Literature Review

¡discuss the cpso papers and the delaunay triangulation papers¿

$$addgeneralGPsymbolicregressionhere(nofinance)$$

4 Experiments

4.1 Experiment 1: CPSO-S

4.1.1 Setup

4.1.2 Results

4.1.3 Discussion

4.2 Experiment 2: CPSO-Sk

4.2.1 Setup

4.2.2 Results

4.2.3 Discussion

4.3 Experiment 3: CPSO-Hk

4.3.1 Setup

4.3.2 Results

4.3.3 Discussion

4.4 Experiment 4: CPSO-Rk

4.4.1 Setup

4.4.2 Results

4.4.3 Discussion

5 Conclusion

A Preliminary Trials

References

[1] Gjacquenot. Delaunay circumcircles vectorial. [Online; accessed May 3 2016].

[2] Alan Mackworth David Poole and Randy Goebel. Computational intelligence: A logical approach. pages 1–21, 19986.

[3] F. Van den Bergh and A.P. Engelbrecht. A cooperative approach to particle swarm optimization. *IEEE Transactions on Evolutionary Computation*, page 225–239, June 2004.

[4] Justin Maltese. Vector-evaluated particle swarm optimization using co-operative swarms.