Cooperative PSO with Spatially Meaningful Neighbours

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

1 Introduction **5**

2 Background **5**

2.1 Particle Swarm Optimization 5

2.1.1 Overview 5

2.1.2 Particle 5

2.1.3 Position Update Formula 6

2.1.4 Velocity Update Formula 6

2.1.5 Swarm 7

2.1.6 Fitness Function 8

2.1.7 Parameters 8

2.2 Co-operative Particle Swarm Optimization 10

2.2.1 CPSO-S 11

2.2.2 Variant: CPSO-S_k 12

2.2.3 Variant: CPSO-R_k 13

2.3 Neighborhood Topologies 13

2.3.1 Common Spatially Random Topologies 14

2.3.2 Spatially Meaningful Neighbors 15

2.4 Dynamic Connections 16

2.5 Delaunay Triangulation 19

2.5.1 Triangulation 19

2.5.2 Delaunay Triangulation 19

3 Literature Review/Previous work **20**

3.1 CPSO 20

3.2 Delaunay Triangulation 22

4 CPSO using Delaunay Triangulation **23**

4.1 Experimental Setup 23

4.2 CPSO configuration 24

4.3 Benchmark functions 25

5 Experiment Results and Discussion **26**

5.1 Experiment 1: Fixed Iteration Results 26

5.1.1 Results 26

5.1.2 Discussion 26

5.2 Experiment 2: Robustness Test 26

5.2.1 Results 27

5.2.2 Discussion 27

6 Conclusion **27**

A Preliminary Trials **27**

Bibliography **28**

List of Tables

1 PSO Pseudo Code 7

2 CPSO-S Pseudo Code 11

3 CPSO-S_k Pseudo Code 12

4 Choosing The Best Neighbour 18

5 Functions, Stop Criteria and Domains 25

List of Figures

1 Visual Representation of the Neighborhoods[1] 14

2 Von Neumann[2] 16

3 How to know if the particles are working together[1] 17

4 Circumcircles drawn amongst the Triangles[3] 20

1 Introduction

2 Background

2.1 Particle Swarm Optimization

2.1.1 Overview

Particle Swarm Optimization (PSO) is an iterative population-based computational intelligence algorithm which finds an optimal solution 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 [4]. 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 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/network best (the best value found by all the particles in the swarm). The velocity and position are updated each iteration with the following functions.

2.1.3 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.

2.1.4 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.

2.1.5 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.

2.1.6 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.

2.1.7 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 converged 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 [5].

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.[6] 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.[6] 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 [5] 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.

2.3 Neighborhood Topologies

The Neighborhood Topology of a PSO dictates how the Swarms' individual particles are allowed to communicate with each other. This communication is the real strength of the PSO as it allows it numbers to work together in solving the problem. This particle communication is achieved through the use of 'Global Best' values. For each particle, when it is updating it's velocity for the next iteration, it will consult the global best in its network to aid in determining it's next move. In the standard PSO (Section **??**), all particles are able to communicate with each other. This means that all particles in the swarm are 'led' by the

overall best particle in the entire swarm. This behaviour can sometimes lead to a naive swarm that quickly converges on an 'optimal' solution without full exploring the entire search space of the problem. To combat this, a local PSO [2] was proposed, which utilized different, more constrained, neighborhood topologies in hopes of creating a better PSO .

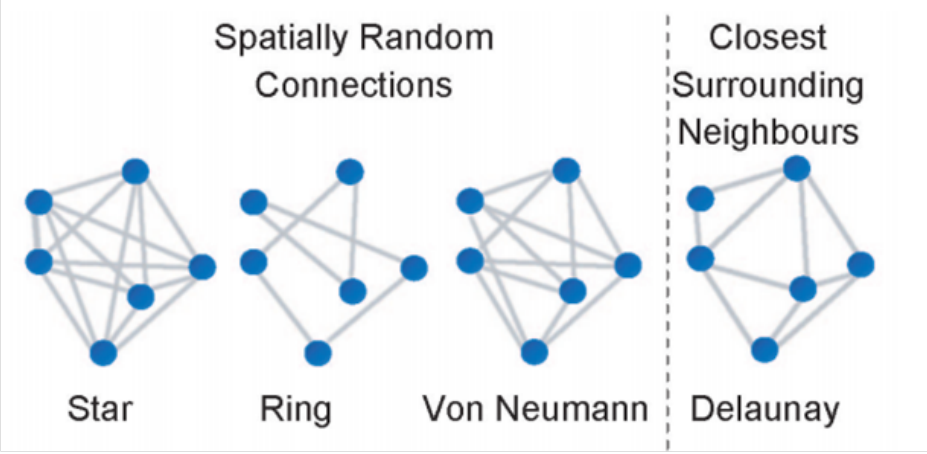

Figure 1: Visual Representation of the Neighborhoods[1]

2.3.1 Common Spatially Random Topologies

There are a number of Neighborhood Topologies that are currently being used and tested in the PSO community. Many of these topologies have their own pros and cons and can lead to different or better results depending on the problem in question. These topologies are known as 'Spatially Random' since the location of the particles at any point in the calculation has no bearing on what particles it is connected to.

Star

The Star Topology is the most common since it is used in the standard PSO. As mentioned in the intro, Star Topology connects every particle with each other. That means the 'neighborhood best' for every particle is the same as the 'global best' position in the swarm. The Star Topology uses this information to quickly converge onto the optimal best without a thorough exploration of the search space.

Ring

The Ring Topology changes the network topology by reducing the connections per particle to just two. All the particles are ultimately still connected to each other, but through multiple intermediary particles. This topology slows the transfer of information among the particles, decreasing the impact the global best has on the swarm itself[2]. This helps to slow down the convergence of the swarm, allowing a more thorough investigation of the search space. However, as a result, it requires a larger number of iterations to find an optimal solution. The connections are often determined randomly during the initialization of the particles.

Von Neumann

The Von Neumann Topology expands on the Ring Topology by increasing the number of connections each particles has while still maintaining the slow information transfer rate within the network. This helps to speed up convergence while allowing for exploration. The particles are connected in a cubic-lattuce arrangement where particles are connected to particles on the left, right, above and below. This isn't, however, determined by actual location of the particle, but often based on the index value that particle is in the Swarm array, where above and below are determined by some sort of offset based on the swarm size (See Figure **??**). This means it does not take advantage of positioning information and the connections are generated only in the initialization phase.

2.3.2 Spatially Meaningful Neighbors

Where the updates to the Spatially Random Neighborhood Topologies allowed for more time in searching the search space, the actual searching ultimately ends up being mostly random

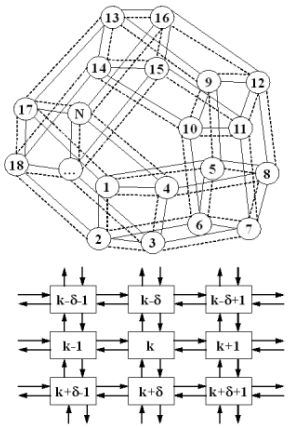

Figure 2: Von Neumann[2]

movements. This is mostly due to the fact that these connections are both randomly selected and only determined once during the life cycle of the swarm. If the connections were dynamically chosen based on information within the swarm, that information could be used to better direct the collaboration between particles. In the paper "Particle Swarm Using Spatially Meaningful Neighbors" [1] Lane, Engelbrecht and Gain proposed using the distance between particles to determine which particle is connected to which. It was theorized that particles that are close together have a high probability that they are searching the same subspace (a pocket of space that has it's own local max/min). The proposed a new Neighborhood Topology called 'Spatially Meaningful Neighbors' that uses the Delaunay Triangulation algorithm to connect all particles with the neighbors that are closest to them. Those connections then represent which particles they can talk to and draw information from. Decisions are then made to avoid or proceed in searching a space based on how the particles are moving (See Section **??**). This led to a more guided search and a higher success rate in their tests. [1]

2.4 Dynamic Connections

Just having the list of the two closest neighbours for each particle isn't in itself inherently helpful for Particle Swarm Optimization. Of this list, we now want to determine which of

these particles are are working together so we can have the particles cooperate to find a local optima. Setting up connections between these particles will allow them to communicate and share information, allowing them to more quickly converge on that optima. A particle is considered to be working with another if:

- Particle P_1 is following behind Particle P_2. i.e. both are heading in the same direction. This this case, we'd want to establish a directed connection so the particle in front can send information to the trailing particle
- Both Particle P_1 and Particle P_2 are heading towards each other but not passing each other. Here we would want to set up an undirected connection so that both particles can share information.

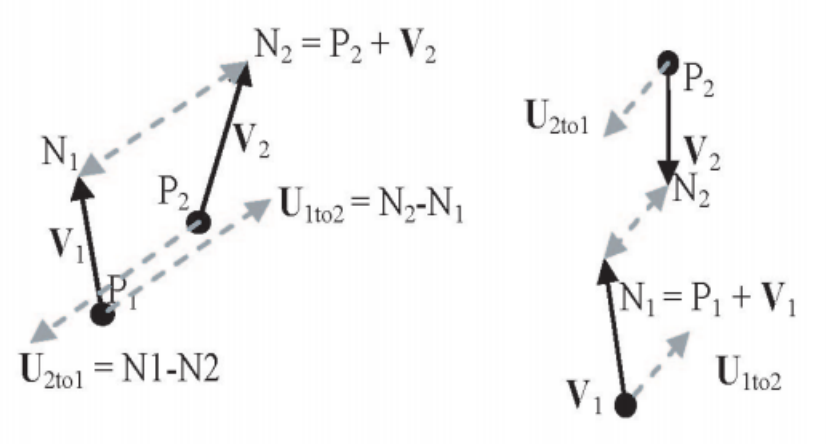

Figure 3: How to know if the particles are working together[1]

You can test both cases with a simple equation:

$$V_1 \cdot U_{1to2} > 0$$

In the function above, the V_1 represents the current velocity of P_1 and U_{1to2} is the vector representing the distance from P_1 to P_2 after the two Particles next position update. Using the dot product on both these vectors allows us to determine the angle between them. If

the dot product is greater than 0, we know the angle between the two vectors is less than 90 degrees. This outcome can only occur if it satisfies one of the two.

A caveat to this would be that we still wish to encourage local exploration. Delaunay Triangulation does not guarantee that the particles are close together. If two neighbouring Particles are working together, but are still reasonably far away, having them communicate will inhibit local exploration. Therefore we need to set a threshold in which we want the particles to search within before they start communicating. 4 details the process of finding the best neighbour to communicate with.

Table 4: Choosing The Best Neighbour

input: N_i Particle i's closest neighbours
procedure CHOOSEBESTNEIGHBOUR(N_i)
    hasConnectedNeighbours = false
    $P_f \leftarrow min(N_k)$
    for $k \leftarrow 1toneighboursets_ize$ **do**
        if $working_together(P_i, P_k)$ and $dist(X_i - P_c) < dist(X_i - P_k)$ and $f(P_k) < f(X_i)$ **then**
            $P_c \leftarrow P_k$
            hasConnectedNeighbours $\leftarrow$ true
        end if
    end for
    $localExploitationRation \leftarrow swarm.diameter/200$
    if $hasConnectedNeighbours$ and $distance(X_i - P - c)/distance(X_i - P - f) > localExploitationRatio$ and $V_i < 2 * distance(X_i - P_x)$ **then**
        $P_n \leftarrow P_c$
    else
        $P_n \leftarrow P_f$
    end if
return P_n
end procedure

2.5 Delaunay Triangulation

2.5.1 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.

2.5.2 Delaunay Triangulation

Delaunay Triangulation (DT) is considered to be one of the most efficient triangulation methods in computational geometry and often leads to optimal or close to optimal triangulations. This triangulation methodology creates 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.

The issue with Delaunay Trinagulation is one of dimensionality. In 2-dimensions, Delaunay Triangulation can be done with a worst-case time complexity of just O($nlogn$) which is one of the quickest methods for pseudo-optimal triangulation. However, it becomes in-

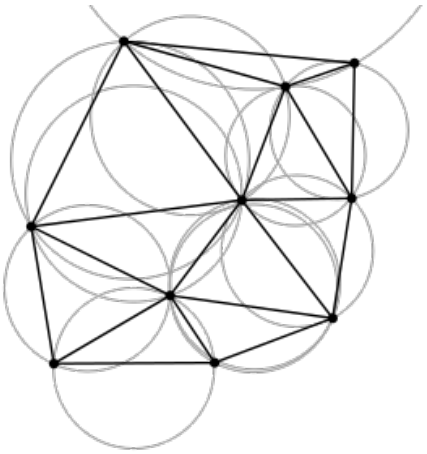

Figure 4: Circumcircles drawn amongst the Triangles[3]

creasingly difficult to compute as dimensions increase making it a less viable solution to all versions of problem. The majority of algorithms that compute Delaunay Triangulation are typically limited to just 2 and 3 dimensions. One way of computing DT in higher dimensions would be to increase the dimension by 1 (giving the value of the new dimension as $|p|^2$ and then computing the convex hull on those sets of points. However, as the dimensions increase even that becomes increasingly complex to compute. In 4-dimensions the best case algorithm will have a worst-case $O(n^3)$ runtime and in d-dimensions, it takes $O(n^{\lceil d/2 \rceil + 1})$ time.

3 Literature Review/Previous work

3.1 CPSO

In van den Bergh and Engelbrecht's 2004 paper "A Cooperative Approach to Particle Swarm Optimization", they introduced a new concept called Cooperative Particle Swarm Optimization (CPSO). They noted that, like other stochastic optimization algorithms, PSO suffered from the "curse of dimensionality". This references the increasing difficulty of finding an optimal solution as dimension increases as the potential search space increases exponentially. Taking note of Potter's dimensional decomposition to help solve this problem in Genetic Algorithms (another stochastic optimization algorithm), they attempted to do the same for

Particle Swarm Optimization and in the process, solving the issue of pseudominima, in which each swarm gets stuck in a local optima which isn't globally optimal.

Their approach was to create two new PSO algorithm variants. The first was CPSO-S_k, which took the idea of dividing the problem into smaller dimensions, but instead of n 1-dimensional sub problems, it is divided into k problems of equal size. This is in direct response to the issue of pseudominima, which spawns from the issue of each swarm optimizing completely separately. The division is now done in hopes that the dependent variables are grouped together in the same smaller sub swarms. The second proposed variant was CPSO-H_k (Hybrid CPSO). This took advantage of PSOs ability to overcome local optima and CPSO's ability for quick convergence by having both CPSO-S_k and a standard PSO run simultaneously. The two algorithms would share information and help each other find a global optima

Their tests revealed that both variants of CPSO vastly outperformed the standard Particle Swarm Optimization algorithm in higher dimension problems. Each algorithm was tested against a number of standard optimization formulas with 30 dimensions each and not only did they see an increase in terms of the quality of solutions returned, but also in the robustness of the solutions (meaning they were able to reach an optimum solution prior to hitting the max loops). CPSO algorithms were able to converge onto solutions significantly faster than the standard PSO while still leading to global optimums. The problem of pseudominima still persisted in the CPSO-S_k, though at a lesser degree, however the other proposed variant CPSO-H_k performed much better in this case. While the Hybrid variant required more memory space and processing power to compute, it overall was able to overcome local minima better than its counter part and overall led to the best solutions.

Overall, they found that CPSO is an essential tool for solving optimization problems in higher dimensions and leads to objectively better results than it's non-cooperative counterparts

3.2 Delaunay Triangulation

Particle Swarm Optimization is a powerful algorithm for optimizing continuous non-linear function by having Particles swarm around the search space to help converge on the global optimum value. One way PSO can be improved upon is by introducing communication techniques to allow the swarms to more directly work together and 'team up' to better evaluate a local search space. A number of Particle knowledge-transfer topologies have been introduced in the past, however, in Lane, Engelbrecht and Gain's paper titled "Particle Swarm Optimization with Spatially Meaningful Neighbours" They propose a different topology utilizing Delaunay Triangulation's ability to easily locate each particles closest neighbour. In this paper, they propose the idea that trading information amongst close particles allow for better use of the local search space.

Delaunay Triangulation is a quick and effective way of finding neighbours in low dimensions that typically represent the closest neighbours to each particle. The algorithm returns triangles which make use of all the particles and also has approximately the minimum-weight (or shortest overall edge lengths). Lane, Engelbrecht and Gain use this information along with a number of other heuristics to determine which particle should communicate with which for best result. They noted that having particles that are 'working together' share information allowed them to collaborate to better explore the smaller local area. Particles were defined as working together if the were headed either in the same direction or towards each other. They also noted that communication with the optimal neighbor was also effective as long as the neighbor wasn't too far way.

After testing their theory, they were able to conclude that Delaunay Triangulation for information sharing between particles led to much better results in low dimensions. Its success rates were almost uni-formally higher than the other knowledge transfer approaches, meaning it was more likely to find the global optimum solution. The problem arose with larger dimension problems. As the dimension rose, the Delaunay Triangulation increasingly became the algorithms bottleneck. In 2 or 3 dimensions, Delaunay Triangulation is a quick

way of determining the particle's nearest neighbours. In larger dimensions, however, it becomes significantly harder to compute and is therefore less helpful for use in Particle Swarm Optimization

Lane, Engelbrecht and Gain go on to conclude that the use of spatially significant neighbours to knowledge transfer is comparatively successful to the other common knowledge transfer topologies. The use of Delaunay Triangulation, however limits the approach to just 2 and 3 dimensions.

4 CPSO using Delaunay Triangulation

Building off of the previous work, it is clear the major downside to the use of Delaunay Triangulation spawns from its issues in higher dimensions. In the past, such issues were solved by utilizing CPSO's ability to break the problem into a number of smaller sub problems, as seen with PSO's "curse of dimensionality". Knowing that it has been known to lead to better results, it would be worth trying Delaunay Triangulation in conjunction with CPSO algorithms to see if they can both benefit from each others skills to better solve problems in n-dimensions. It was stated that the tipping point for this benefit is for problems greater than 3 dimensions, as such, all CPSO iterations have been initialized such that no one swarm is handling more than 3 dimensions.

4.1 Experimental Setup

To test this theory, a number of known CPSO algorithms were implemented with the added functionality of the communication via spatially significant neighbours using the Delaunay Triangulation approach. The following algorithms were tested for this experiment:

- **PSO**: For comparison's sake, all CPSO variants will be compared to the standard PSO algorithm to ensure either of the changes lead to a meaningful benefit. All the values will be identical to their CPSO counterparts.

- **CPSO-S**: This variant restricts each swarm to just one dimension each particle. Since Delaunay Triangulation does not exist in 1-dimension, it was not used (nor required) for finding the closest neighbours. The rest of the algorithm, however, was still applicable.

- **CPSO-S$_k$**: Due to Delaunay Triangulation being limited to just 2 and 3 dimensions, a k needed to be selected in which the swarm is divided into 2 or 3 dimension chunks.

- **CPSO-R$_k$**: The Algorithm was altered to ensure all the random swarms are selected to only be between 1 and 3 dimensions to ensure the spatially significant neighbours algorithm works.

4.2 CPSO configuration

The CPSO and PSO parameters were selected to match those tested in both the Delaunay Triangulation and the CPSO paper in attempt to replicate the results achieved in those experiments. As per the other experiments, the number of iterations were capped at 10,000. To keep the playing field even, the particles allotted to each algorithm will be distributed evenly amongst each swarm. This means that each iteration for the swarm will contain the exact same number of fitness calls, allowing us to directly compare their results. Each experiment was performed 50 times; the reported results are the average of the global/network bests after all of the 50 runs. The experiments were also repeated for each type of swarm using 10, 15, and 20 particles for 6 dimension problems and 100,150,200 for 60 dimension problem also with and without the use of spatially significant neighbours. All variants of CPSO used the following input values:

- **Social Weight** (λ): 1.49

- **Cognitive Weight** (ω): 1.49

- **Inertial Weight** (α): starts at 1 and linearly decreases with each iteration, ultimately hitting 0 on the last iteration

- **Max Velocity** (V_{max}): clamped to the domain of the function
- **Max Position** (P_{max}): also clamped to the domain of the function
- **Particle Count**: particles are evenly distributed among the swarms in the algorithm. for example, if the particle count is set to 100 and the CPSO has 10 different swarms, each swarm will only have 10 particles to solve its particular subproblem.

4.3 Benchmark functions

Again, in an attempt to replicate the results of the previous experiments, for comparisons sake, the same benchmark functions were selected here as were in the two previous papers. The functions which were selected are as follows:

Function	Domain	Criterion
Rastrigin	[-5.12;5.12]	0.01
Rosenbrock	[-30;30]	100
Griewanck	[-600;600]	0.05
Ackley	[-32;32]	0.01

Table 5: Functions, Stop Criteria and Domains

The "Success Rate" evaluation indicates the percentage of runs that reached the end criterion and the "Iterations to Criterion" value returns the average number of iterations it took to reach that stopping point.

The functions themselves are as follows:

Rastrigin:

$$\sum_{i=1}^{n} x_i^2 + 10 - 10cos(2\pi x_i)$$

Rosenbrock:

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

Griewanck:

$$\frac{1}{4000}\sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} cos(\frac{x_i}{\sqrt{i}})x_i + 1$$

Ackley:

$$20 + e - 20e^{-0.2\sqrt{\frac{\sum_{i=1}^{n} x_i^2}{n}}} - e^{\frac{\sqrt{\sum_{i=1}^{n} cos(2\pi x_i)}}{n}}$$

5 Experiment Results and Discussion

5.1 Experiment 1: Fixed Iteration Results

The first experiment compares how close the final results got to the optimal best after a fixed number of iterations. The tables presented list the following criteria. The first column is the algorithm tested, the second is the number of particles (s) used in that run of the application, the third is the Mean error and 95% confidence interval after the 10,000 iterations without Spatially Meaningful Neighbors while the fifth is with. It is to be noted that all of the tested functions have a minimum fitness value of 0 so the preferable result would be a confidence interval very close to or around that value. Line graphs are also provided that visualize the global best of each algorithm over the coarse of the fixed iteration run-time. These values are the average of all 50 tested runs.

5.1.1 Results

5.1.2 Discussion

5.2 Experiment 2: Robustness Test

The mark of a good PSO algorithm is one that can hit or get very close to the optimal result on a consistent basis. As such, the robustness of the algorithm needed to be tested as well. In this case, "Robustness" refers to the number of times and the speed at which the algorithm hit the exit threshold over the course of 50 runs. The tables presented list the algorithm being tested first, followed by the number of particles used in that calculation. Following that is the percentage of successful runs with the average iterations without Spatially Meaningful Neighbors followed by the algorithm with. The best runs are those that we successful the

most times while taking the least number of iterations to achieve it. Note: if the run was not successful, it's iteration count does not factor into the average iterations.

5.2.1 Results

5.2.2 Discussion

6 Conclusion

A Preliminary Trials

References

[1] Andries Engelbrecht James Lane and James Gain. Particle swarm optimization with spatially meaningful neighbours. *Swarm Intelligence Symposium*, September 2008.

[2] Mu˜noz Zavala Angel Eduardo. A comparison study of pso neighborhoods. *EVOLVE - A Bridge between Probability, Set Oriented Numerics, and Evolutionary Computation II*, 175:251–265.

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

[4] Alan Mackworth David Poole and Randy Goebel. *Computational Intelligence: A Logical Approach*. Oxford University Press, 19986.

[5] 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.

[6] Justin Maltese. Vector-evaluated particle swarm optimization using co-operative swarms. *3F90 Thesis*, 2014.