UNIVERSIDAD DE CHILE
FACULTAD DE FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

DISEÑO E IMPLEMENTACIÓN DE ALGORITMOS APROXIMADOS DE CLUSTERING BALANCEADO EN PSO

TESIS PARA OPTAR AL GRADO DE
MAGÍSTER EN CIENCIAS MENCIÓN COMPUTACIÓN

CHUN HAU LAI

PROFESOR GUÍA:
CLAUDIO GUTIERREZ GALLARDO

MIEMBROS DE LA COMISIÓN:
JÉRÉMY BARBAY
JOSE CORREA HAEUSSLER
MARCELO MENDOZA ROCHA

SANTIAGO DE CHILE
JULIO 2012

Resumen

Este trabajo de tesis está dedicado al diseño e implementación de algoritmos aproximados que permiten explorar las mejores soluciones para el problema de *Clustering* Balanceado, el cual consiste en dividir un conjunto de n puntos en k clusters tal que cada cluster tenga como mínimo $\lfloor \frac{n}{k} \rfloor$ puntos, y éstos deben estar lo más cercano posible al centroide de cada cluster. Estudiamos los algoritmos existentes para este problema y nuestro análisis muestra que éstos podrían fallar en entregar un resultado óptimo por la ausencia de la evaluación de los resultados en cada iteración del algoritmo. Entonces, recurrimos al concepto de *Particles Swarms*, que fue introducido inicialmente para simular el comportamiento social humano y que permite explorar todas las posibles soluciones de manera que se aproximen a la óptima rápidamente. Proponemos cuatro algoritmos basado en *Particle Swarm Optimization* (*PSO*): *PSO-Húngaro*, *PSO-Gale-Shapley*, *PSO-Aborción-Punto-Cercano* y *PSO-Convex-Hull*, que aprovechan la característica de la generación aleatoria de los centroides por el algoritmo *PSO*, para asignar los puntos a estos centroides, logrando una solución más aproximada a la óptima.

Evaluamos estos cuatro algoritmos con conjuntos de datos distribuidos en forma uniforme y no uniforme. Se encontró que para los conjuntos de datos distribuidos no uniformemente, es impredecible determinar cuál de los cuatro algoritmos propuestos llegaría a tener un mejor resultado de acuerdo al conjunto de métricas (intra-cluster-distancia, índice Davies-Doublin e índice Dunn). Por eso, nos concentramos con profundidad en el comportamiento de ellos para los conjuntos de datos distribuidos en forma uniforme.

Durante el proceso de evaluación se descubrió que la formación de los clusters balanceados de los algoritmos *PSO-Absorción-Puntos-Cercanos* y *PSO-Convex-Hull* depende fuertemente del orden con que los centroides comienzan a absorber los puntos más cercanos. En cambio, los algoritmos *PSO-Hungaro* y *PSO-Gale-Shapley* solamente dependen de los centroides generados y no del orden de los clusters a crear. Se pudo concluir que el algoritmo *PSO-Gale-Shapley* presenta el rendimiento menos bueno para la creación de clusters balanceados, mientras que el algoritmo *PSO-Hungaro* presenta el rendimiento más eficiente para lograr el resultado esperado. Éste último está limitado al tamaño de los datos y la forma de distribución. Se descubrió finalmente que, para los conjuntos de datos de tamaños grandes, independiente de la forma de distribución, el algoritmo *PSO-Convex-Hull* supera a los demás, entregando mejor resultado según las métricas usadas.

A Chian Lai e Ming Tzea Chuang

Agradecimientos

Ninguno puede hacer la decisión sin vida. Las personas que nos han entregado la vida son justamente nuestros padres. Mi Padre, un obrero sencillo y esforzado para trabajar; mi Madre, una cocinera talentosa y sociable. Ambos se conocieron en una peluquería y después de seis meses de conocerse, se casaron y dieron lugar a mi. Durante los 30 años de acompañamiento, conocí qué es la esencia natural de los humanos: *el amor*. Nadie nace en este mundo para consumir o destruir, mis padres me animaron a entrar a realizar un postgrado a fin de entregar un aporte a esta sociedad. Y ahora, ya lo estoy haciendo.

Durante el proceso de realizar el postgrado, agredezco al Profesor Claudio Gutierrez por su paciencia y apoyo. Aunque no soy un alumno muy idóneo para que me dirija tesis, igual me brindaron muchos consejos y visiones valiosas para que yo pueda continuar trabajando en mi tesis. En China, siempre se dice lo siguiente: *El profesor es el que transmite la verdad, enseñanza y soluciona las dudas*. Gracias nuevamente al Profesor Claudio Gutierrez que me ha indicado siempre qué pasos a seguir y puntos a verificar en mi proceso de tesis.

A mi esposa, Chia Ching Lee, a quién le debo agradecer mucho. Porque durante estos años de postgrado, ella siempre me ha dado su apoyo y paciencia. Aunque en varios momentos, he tenido que retroceder en mi paso, ella me empuja siempre adelante, y me dice: *si uno siempre piensa en perder antes de ir a la guerra, perderá*. Por esta frase tan sencilla e inspiradora, me animó de nuevo en este c amino de postgrado.

También agradezco a mi gran compañero: Ricardo Barrientos, quien se encuentra ahora en España realizando su doctorado, me ayudó en corregir la ortografía y formato de esta tesis. Él sabe muy bien de que no soy chileno y obviamente mi castellano no va a estar bien, por eso se ofreció en darme este apoyo. Tal como un dicho chino dice: *en las adversidades, uno ve la verdadera amistad*. En realidad, en mi vida no tiene muchos amigos, y el único amigo íntimo que tengo, es justamente este gran compadre: Ricardo. Muchas gracias a tí, he podido entregar una tesis bonita. También debo agradecer al *Departamento de Cienciasde la Computación* de la Universidad de Chile por las becas otorgadas.

Chun Hau Lai, Abril 2012.

Índice general

Resumen I

Agradecimientos III

1. Introducción **1**

1.1. Motivación . . . 1

1.2. Definición del Problema . . . 2

1.2.1. Evaluación de los resultados del *clustering* . . . 2

1.3. Trabajos Relacionados . . . 3

1.3.1. Algoritmos Básicos de Clustering . . . 4

1.3.2. Clustering balanceado basado en algoritmos de *k-means* . . . 5

1.3.3. Falencias del trabajo de *k-means con Ajuste del Borde* . . . 6

1.3.3.1. Falencias del algoritmo . . . 6

1.3.3.2. Falencia de la evaluación del resultado . . . 7

1.4. Objetivos de la tesis . . . 7

2. Preliminares: Particle Swarm Optimization (PSO) **9**

2.1. La evolución de *PSO* . . . 10

2.2. El refinamiento de *PSO* . . . 13

2.2.1. La explosión de Swarm . . . 13

2.2.2. El concepto del peso de inercia . . . 14

2.2.3. El concepto del vecino . . . 14

2.2.4. El valor óptimo de los parámetros . . . 16

2.3. Clustering Basado en *PSO* . . . 16

3. Diseño del algoritmo **18**

3.1. Estrategias para determinar la posición de los centroides . . . 19

3.1.1. Etiquetar los k puntos como centroides con fuerza bruta . . . 20

3.1.2. Etiquetar los k puntos como centroides con *PSO* 20
3.1.3. Generar aleatoriamente k posiciones espaciales como centroides con *PSO* . . 21
3.2. Estrategias de la asignación óptima 21
3.2.1. Clustering Balanceado como un problema de asignación 22
3.2.1.1. El Algoritmo Húngaro 23
3.2.1.2. Modificación del algoritmo de Clustering Basado en *PSO* con la idea del algoritmo Húngaro 24
3.2.2. Clustering Balanceado como un problema de matrimonios estables 26
3.2.2.1. Algoritmo Gale-Shapley 28
3.2.2.2. Algoritmo PSO combinado con el algoritmo Gale-Shapley 28
3.2.3. Algoritmo PSO con absorción de los puntos más cercanos 29
3.2.4. Algoritmo PSO con la envolvente convexa 30

4. El caso de los datos uniformes **38**
4.1. Introducción 38
4.2. Análisis del resultado de la ejecución de los cuatro algoritmos 39
4.2.1. Visualización de los *clusters* generados 39
4.2.1.1. Intra-Cluster-Distancia 39
4.2.1.2. Indice Davies-Bouldin 39
4.2.1.3. Indice Dunn 40
4.2.1.4. Conclusión 40
4.2.2. Influencia de $|D|$ para cada algoritmo 41
4.2.2.1. Intra-Cluster-Distancia 41
4.2.2.2. Indice Davies-Bouldin 41
4.2.2.3. Indice Dunn 42
4.2.2.4. Conclusión 42
4.2.3. Influencia de K para cada algoritmo 42
4.2.3.1. Intra-Cluster-Distancia 42
4.2.3.2. Indice Davies-Bouldin 43
4.2.3.3. Indice Dunn 43
4.2.3.4. Conclusión 43
4.2.4. Tiempo de Ejecución de cada algoritmo 43
4.3. Análisis del resultado de la ejecución de los algoritmos *PSO-Absorcion-Puntos-Cercanos* y *PSO-Convex-Hull* 44
4.3.1. Visualización de los *clusters* generados 44

4.3.2. Influencia de $|D|$ para los algoritmos *PSO-Absorcion-Puntos-Cercanos* y *PSO-Convex-Hull* 45
4.3.3. Influencia de K para los algoritmos *PSO-Absorcion-Puntos-Cercanos* y *PSO-Convex-Hull* 46
4.3.4. Tiempo de Ejecución de los algoritmos *PSO-Absorcion-Puntos-Cercanos* y *PSO-Convex-Hull* 46

5. El caso de los datos no uniformes **62**
5.1. Introducción 62
5.2. Visualización de los *clusters* generados 62
5.2.1. Intra-Cluster-Distancia 62
5.2.2. Indice Davies-Bouldin 63
5.2.3. Indice Dunn 63
5.3. Conclusión 64

6. Conclusiones **76**
6.1. Trabajo Futuro 77

A. Código de implementación de los algoritmos propuestos **78**
A.1. Particle Swarm Optimization 78
A.2. Algoritmo Húngaro 82
A.3. Algoritmo Gale-Shapley 86
A.4. Algoritmo Convex-Hull 87
A.5. Algoritmo PSO-Hungaro 90
A.6. Algoritmo PSO-Gale-Shapley 91
A.7. Algoritmo PSO-Absorción-Puntos-Cercanos 92
A.8. Algoritmo PSO-Convex-Hull 93

B. Bibliografía **96**

Índice de figuras

1.1. El resultado generado por el algoritmo de Han (Algoritmo 1) y el resultado óptimo esperado de clustering balanceado. 6

2.1. Particle Swarm Optimización (figura tomada de [25]). 10
2.2. La nueva posición candidata de la partícula $x_i = (0,0)^T$ (cruz) con $p_i = (2,1)^T$(estrella) y $p_g = (1,3)^T$(cuadrado), para los casos (A)$c_1 = c_2 = 1{,}0$, y (B) $c_1 = c_2 = 2{,}0$ (figura tomada de [25]). 12
2.3. Nuevas posiciones candidatas generadas por la posición 2.2 para la partícula $x_i = (0,0)^T$ (cruz), con $p_i = (2,1)^T$ (estrella), y $c_1 = c_2 = 2{,}0$, cuando (A) R_1 y R_2 son vectores aleatorios n-dimensional, y (B), valores aleatorios unidimensional (figura tomada de [25]). 13
2.4. La divergencia del Swarm durante la búsqueda con (línea sólida) y sin (línea punteada)el peso de inercia donde $N = 20$, $c_1 = c_2 = 2$, $T_{\text{máx}} = 500$, $v_{\text{máx}} = 50$, $w_{up} = 1{,}2$ y $w_{low} = 0{,}1$ (figura tomada de [25]). 15
2.5. La divergencia del Swarm durante la búsqueda con el concepto de los vecinos(lbest, línea solida) y sin los vecinos(gbest, línea punteada) donde $N = 20$, $c1 = c2 = 2$, $T_{\text{máx}} = 500$, $v_{\text{máx}} = 5{,}12$, $w_{up} = 1{,}2$ y $w_{low} = 0{,}1$ y $n = 10$ para miminizar la función de Rastrigin en el rango $[-5{,}12, 5{,}12]^{10}$ con $NB_i = 1$ (figura tomada de [25]). 16

3.1. 9 puntos(círculos) distribuidos en el espacio euclidiano junto con el centroide fijo(cruz) y el centroide generado por PSO(diamante). 19
3.2. La absorción de los puntos cercanos a paritir del centriode más cercano al punto central del mapa(cruz verde) y del centroide más lejano al centro del mapa(cruz amarilla). 30
3.3. Capas de la envolvente convexa. 32
3.4. Las 8 Capas de la envolvente convexa con 31 puntos. 33
3.5. El clustering balanceado para 31 puntos para $k = 5$. 33

4.1. El resultado de la ejecución de los cuatro algoritmos dado $|D| = 500$ y la métrica *Intra-cluster-distancia* con la distribución uniforme de los datos en el plano. 47
4.2. El resultado de la ejecución de los cuatro algoritmos dado $|D| = 500$ y la métrica *Indice Davies-Bouldin* con la distribución uniforme de los datos en el plano. 48
4.3. El resultado de la ejecución de los cuatro algoritmos dado $|D| = 500$ y la métrica *Indice Dunn* con la distribución uniforme de los datos en el plano. 49
4.4. Influencia de $|D|$ para cada algoritmo dada las distintas métricas. 50
4.5. Influencia de K para cada algoritmo dada las distintas métricas. 51
4.6. Influencia de $|D|$ para cada algoritmo. 52
4.7. El resultado de la ejecución de los cuatro algoritmos dado $|D| = 40000$ y la métrica *Intra-cluster-distancia* con la distribución uniforme de los datos en el plano. 53
4.8. El resultado de la ejecución de los cuatro algoritmos dado $|D| = 40000$ y la métrica *Indice Davies-Bouldin* con la distribución uniforme de los datos en el plano. 54
4.9. El resultado de la ejecución de los cuatro algoritmos dado $|D| = 40000$ y la métrica *Indice Dunn* con la distribución uniforme de los datos en el plano. 55
4.10. Influencia de $|D|$ para los algoritmos *PSO-Absorción-Puntos-Cercanos* y *PSO-Convex-Hull* dada las distintas métricas. 56
4.11. Influencia de K para los algoritmos *PSO-Absorción-Puntos-Cercanos* y *PSO-Convex-Hull* dada las distintas métricas. 57
4.12. El tiempo de ejecución de los algoritmos *PSO-Absorción-Puntos-Cercanos* y *PSO-Convex-Hull*. 58

5.1. El resultado de la ejecución de los cuatro algoritmos dado $|D| = 500$, $K = 20$ y la métrica *Intra-cluster-distancia* con el conjunto de datos (D'). 66
5.2. El resultado de la ejecución de los cuatro algoritmos dado $|D| = 500$, $K = 20$ y la métrica *Intra-cluster-distancia* con el conjunto de datos (D''). 67
5.3. El resultado de la ejecución de los cuatro algoritmos dado $|D| = 500$, $K = 20$ y la métrica *Indice Davies-Bouldin* con el conjunto de datos (D'). 68
5.4. El resultado de la ejecución de los cuatro algoritmos dado $|D| = 500$, $K = 20$ y la métrica *Indice Davies-Bouldin* con el conjunto de datos (D''). 69
5.5. El resultado de la ejecución de los cuatro algoritmos dado $|D| = 500$, $K = 20$ y la métrica *Indice Dunn* con el conjunto de datos (D'). 70
5.6. El resultado de la ejecución de los cuatro algoritmos dado $|D| = 500$, $K = 20$ y la métrica *Indice Dunn* con el conjunto de datos (D''). 71
5.7. Velocidad VS. Calidad de Resultado con el conjunto de datos (D') y (D''). 72

Índice de cuadros

3.1. Matriz de Costo de 3×9. 18
3.2. Matriz de Costo de 3×9 con los centroides c_1(1,1), c_2(2,5) y c_3(8,1). 20
3.3. Matriz de Costo de 3×9 con los centroides c_1(1,2), c_2(8,2) y c_3(2,5). 20
3.4. Matriz de Costo de 3×9 con los centroides c_1(3,3), c_2(8,7) y c_3(1,4). 21
3.5. Matriz de Costo de 9×9 con 9 centroides. 23
3.6. Matriz de Costo del problema de asignación. 24
3.7. Matriz de Preferencia Φ. 27
3.8. Matriz de Preferencia Φ'. 27
3.9. Probabilidades de los puntos $d_i \in L_x$ donde $x = 1, 2, 3, 4$ y 5. 35
3.10. La posición ordenada ascendentemente(según probabilidad) en cada capa L_x. 35
3.11. Eliminar los puntos d_4, d_{11}, d_{19}, d_{20}, d_{24} y d_{29} del cluster C_1. 36
3.12. Eliminar los puntos d_1, d_8, d_9, d_{16}, d_{17} y d_{23} del cluster C_2. 36
3.13. Eliminar los puntos d_2, d_{10}, d_{18}, d_2, d_{28} y d_{29} del cluster C_3. 37
3.14. Eliminación de los puntos d_2, d_{10}, d_{18}, d_3, d_{28} y d_{30} del cluster C_4. 37

4.1. La inter-cluster-distancia d generada por los cuatro algoritmos propuestos con diferentes valores de $|D|$ y k. 59
4.2. El índice Davies Doublin d generado por los cuatro algoritmos propuestos con diferentes valores de $|D|$ y k. 60
4.3. El índice Dunn d generado por los cuatro algoritmos propuestos con diferentes valores de $|D|$ y k. 61

5.1. La inter-cluster-distancia d generada por los cuatro algoritmos propuestos con diferentes tamaños de datos y k para los conjunto de dato D' y D''. 73
5.2. El índice Davies-Doublin d generado por los cuatro algoritmos propuestos con diferentes tamaños de datos y k para los conjunto de dato D' y D''. 74
5.3. El índice Dunn d generado por los cuatro algoritmos propuestos con diferentes tamaños de datos y k para los conjunto de dato D' y D''. 75

Indice de algoritmos

1. k-means con Ajuste del Borde 5
2. Particle Swarm Optimization 11
3. Clustering Basado en *PSO* 17
4. Algoritmo Húngaro: ahungaro(A) 25
5. Clustering Basado en *PSO-Hungaro* 26
6. Algoritmo Gale-Shapley: aGaleShapley(A, A') 28
7. Clustering Basado en *PSO-Gale-Shapley* 29
8. Clustering Basado en *PSO-Absorcion-Puntos-Cercanos* 31
9. Clustering Basado en *PSO-CONVEX-HULL* 34

Capítulo 1

Introducción

1.1. Motivación

Una empresa tiene k ejecutivos de venta y N clientes distribuidos en una ciudad, y el gerente de la empresa busca asignar a cada uno de los ejecutivos la misma cantidad de clientes para efectuar una entrevista con la condición de que los clientes a visitar sea lo más cercanos posibles entre sí con el fin de coordinar el calendario de visitas. Un calendario eficiente se obtiene minimizando la distancia entre los clientes a visitar dentro de cada grupo. Este problema se conoce con el nombre de *Clustering* Balanceado. Este problema es NP-Completo [4,22], y existe bastante literatura [4,6,12,16] referente al modelamiento de algoritmos aproximados para obtener el resultado esperado. Kawahara et al. [16] ha propuesto un algoritmo basado en las funciones submodulares; sin embargo, este algoritmo está limitado para la generación de dos clusters, por lo tanto, no es suficiente para los $k > 2$. Han et al. [12] han propuesto un algoritmo que entrega una solución aproximada; sin embargo ésta no es la óptima debido principalmente a que los centroides son siempre fijos, lo que implica que el algoritmo no explora completamente el espacio de posibles soluciones. Si se quiere explorar el total de soluciones posibles, la complejidad del algoritmo se vuelve exponencial. Teniendo todo esto en cuenta, se hace relevante un algoritmo aproximado que permita explorar todas las posibles soluciones de manera que se aproxime a la óptima rápidamente. Se ha mostrado que el modelo *Particle Swarm Optimization* (*PSO*) [15] converge a la solución rápidamente y ha sido aplicado en el área de *Clustering* por Omran et al. [23], en donde se presenta un algoritmo que genera un conjunto de clusters, pero no balanceados en la cantidad de elementos. *PSO* no ha sido aplicado al algoritmo de Han; pero la metodología propuesta de *Ajuste de Borde* [12] es útil para *Clustering* Balanceado con *PSO*. La presente tesis propone cuatro algoritmos basados en el modelo de Omran que busca mejorar la eficacia del algoritmo de Han para el problema de *clustering* balanceado.

1.2. Definición del Problema

Definición 1 (Clustering Balanceado) *Dados* $D = \{d_1, d_2, ..., d_n\}$ *donde cada* $d_i \in R^n$*, un número entero* k*, encontrar* k *clusters* $\{C_1, C_2, ..., C_k\}$ *que forman una partición de* D *de modo que la suma de las distancias euclidianas entre cada punto* d_j *y el centroide de su cluster sea minimizado. Pero con la restricción que cada cluster* C_x *tenga como mínimo* t *puntos. Formalmente:*

$$\begin{gathered} \text{mín} \sum_{1 \leq i \leq k} \sum_{d_j \in C_i} ||c_i - d_j|| \\ |C_i| \geq t, t = \lfloor \tfrac{n}{k} \rfloor \end{gathered} \tag{1.1}$$

1.2.1. Evaluación de los resultados del *clustering*

Es importante evaluar los clusters generados una vez procesados con el algoritmo. Existe varias métricas para ello. A continuación, se presentan algunas métricas para evaluar el rendimiento del *clustering*:

Definición 2 (Intra-distancia) *Dados* k *clusters, la intra-distancia está definida formalmente como:*

$$\sum_{x=1}^{k} \sum_{d_i, d_j \in C_x} ||d_i - d_j|| \tag{1.2}$$

donde d_x es un punto que es asignado al cluster C_x y $||d_i - d_j||$ es la distancia euclidiana entre los puntos d_i y d_j.

Definición 3 (Inter-distancia) *Dado* k *clusters, la inter-distancia está definida como:*

$$\sum_{\substack{i,j=1 \\ i<j}}^{k} \sum_{x \in C_i, y \in C_j} ||x - y||, i, j = 1, 2, \ldots, k, i \neq j \tag{1.3}$$

donde x se refiere a cada uno de los puntos pertenecientes al cluster C_i e y, al cluster C_j.

Ambas métricas mencionadas son criterios básicos para la evaluación de los clusters resultantes. Se puede concluir que mientras menor es la intra-distancia y mayor es la inter-distancia, mejor es el algoritmo de la generación de los clusters. Sin embargo, la evaluación de estas dos métricas puede tener una complejidad de $O(n^2)$ pese a su efectividad evaluativa. A continuación, se presenta otras dos métricas que tienen una complejidad menor para evaluar el rendimiento del *clustering*:

Definición 4 (Intra-cluster-distancia) *Dados* k *clusters, la intra-cluster-distancia está definida como:*

$$\sum_{1 \leq i \leq k} \sum_{x \in C_i} ||c_i - x|| \tag{1.4}$$

donde c_i es el centroide del cluster C_x y $||c_i - x||$ es la distancia euclidiana entre el centroide c_i del cluster C_i y el punto x.

Definición 5 (Inter-cluster-distancia) *Dado k clusters, la inter-cluster-distancia está definida como:*

$$\sum_{c_i \in C_i, c_j \in C_j, i<j} ||c_i - c_j|| \tag{1.5}$$

donde c_x se refiere al centroide del cluster C_x , y $||c_i - c_j||$ la distancia euclidiana entre los centroides c_i y c_j.

A continuación, presentamos otras dos métricas basadas en *intra-cluster-distancia* e *inter-cluster-distancia*.

Definición 6 (Indice Davies-Bouldin) *El Indice Davies-Bouldin está definido:*

$$DB = \frac{1}{k} \sum_{1 \leq i \leq k} \max_{i \neq j} \left(\frac{\sigma_i + \sigma_j}{||c_i - c_j||} \right) \tag{1.6}$$

donde k es la cantidad de clusters, c_x es el centroide del cluster C_x, σ_x es la distancia promedio de todos los puntos en el cluster C_x hacia el centroide c_x, y $||c_i - c_j||$ es la distancia entre los centroides c_i y c_j. Los algoritmos que producen clusters con la menor intra-distancia y la mayor inter-distancia va a tener un bajo Indice Davies-Bouldin. El algoritmo que genera el conjunto de clusters con el menor índice Davies-Bouldin es considerado como el mejor algoritmo basado en este criterio. Se puede observar simplemente que esta métrica puede calcularse en tiempo $O(kn)$.

Definición 7 (Indice Dunn) *El Indice Dunn cumple el propósito de identificar los clusters densos y clusters bien separados. Se define como la razón entre la mínima inter-cluster-distancia y la máxima intra-cluster-distancia. Para cada cluster, el Indice Dunn puede ser calculados mediante la siguiente fórmula:*

$$DN = \frac{1}{\max_{1 \leq h \leq k} d'(h)} \min_{1 \leq i,j \leq k, i \neq j} ||C_i, C_j|| \tag{1.7}$$

donde $||C_i, C_j|| = \min_{d \in C_i, d' \in C_j} ||d - d'||$, el cual representa la distancia entre los clusters i y j, y $d'(h)$ mide la intra-cluster-distancia del cluster h. A diferencia del Indice Davies-Bouldin, los algoritmos que produce clusters con la menor intra-distancia y la mayor inter-distancia va a tener un alto Indice Dunn, y el algoritmo que genera el conjunto de clusters con un alto Indice Davies-Bouldin es considerado como el mejor algoritmo basado en este criterio

1.3. Trabajos Relacionados

En esta sección, se revisan los trabajos relacionados y se definen los algoritmos básicos para el desarrollo de esta tesis.

Una estrategia simple para evaluar el problema del *clustering* balanceado es utilizar el algoritmo *k-means* para particionar a los clientes en K cúmulos, el cual minimiza la suma de las distancias al cuadrado al centro de cada cúmulo. Luego, redistribuir los clientes de cada cúmulo entre los cúmulos cercanos para lograr una asignación equilibrada [12]. Y los centroides finales se convertirán en la solución definitiva.

1.3.1. Algoritmos Básicos de Clustering

El *Clustering de Datos* se basa en dos métodos principales: jerárquico y divisivo [10, 14, 20]. Para cada uno de ellos, se han propuesto una gran variedad de algoritmos para definir cada cluster. En el primero, se muestra un árbol jerárquico donde cada nodo corresponde a un cluster (partición del conjunto de datos) [20]. El algoritmo jerárquico puede ser aglomerativo (bottom-up) o divisivo (top-down). El algoritmo considera inicialmente a cada punto como un cluster y luego se mezclan estos clusters sucesivamente según un criterio determinado hasta formar clusters de mayor tamaño. El algoritmo jerárquico divisivo considera inicialmente a todo el conjunto de datos como un solo cluster y lo divide sucesivamente hasta formar clusters de menor tamaño. El algoritmo jerárquico tiene dos ventajas principales [10]: (i) la cantidad de clusters no necesita especificarse previamente, y (ii) el algoritmo funciona independiente de las condiciones iniciales. Mientras tanto, el algoritmo divisivo descompone el conjunto de datos directamente en clusters disjuntos optimizando un criterio determinado. Típicamente, el criterio global hace referencia a minimizar la intra-distancia de los elementos dentro de cada cluster y maximización a la inter-distancia de diferentes clusters. Una revisión extensiva de varias técnicas de *clustering* puede ser encontrada en [14].

Uno de los más conocidos algoritmos divisivos es *k-means* [21], que ha sido usado comunmente para minimizar la intra-distancia de los puntos dentro de cada cluster C_x, al mismo tiempo que se maximiza la inter-distancia de los elementos que son asignados a diferentes clusters. El problema principal del algoritmo *k-means* es justamente determinar el valor de k para obtener el mejor resultado. Desafortunadamente no existe una solución teórica general para determinar la cantidad óptima de clusters para cualquier conjunto de datos. Un método sencillo es comparar los resultados de varias ejecuciones con distinto valor de k y escoger la mejor opción de acuerdo a un criterio determinado (por ejemplo, *el criterio de Schwarz*[1]). Otro problema de *k-means* es que no puede lograr *clustering* balanceado, ya que no hay restricción de balance sobre los puntos a asignar a los centroides más cercanos. Para solucionar este problema, Bradley et al. [6] modificaron el algoritmo de *k-means* agregándole la restricción de balance: $|C_i| \geq t$ (que la cantidad de puntos en C_i sea mayor o igual que t) y redujo este problema al flujo de costo mínimo.

[1] Este criterio actúa como la función pérdida de un algoritmo genético empleado para seleccionar el modelo óptimo, es decir, dada una familia de modelos, dentro de la cual está incluido el modelo verdadero, la probabilidad de que este criterio seleccione el modelo verdadero tiende a uno cuando el tamaño muestral tiende hacia infinito. De esta forma el criterio Schwarz evita la sobreparametrización y resuelve el problema de compromiso entre ajuste de los datos intramuestrales y la capacidad de generalización extramuestral.

Algoritmo 1 k-means con Ajuste del Borde

1: $D \Leftarrow \{d_1, d_2, ..., d_n\}$
2: $G \Leftarrow kMedias(D, k)\}$//todos los clusters calculados por k-means
3: **for** $C_i, C_j \in G$ **do**
4:   **if** $|C_i| < t$ y $|C_j| > t$ **then**
5:     **for** $d \in |C_j|$ **do**
6:       $\text{diff}(d, C_i) = dist(c_i, d) - dist(c_j, d)$
7:     **end for**
8:     Orderar todos los $\text{diff}(d, C_i)$ ascendentemente
9:     mover los primeros $t - |C_i|$ puntos del C_j a C_i.
10:   **end if**
11: **end for**

Wagstaff et al. [27] propuso la determinación previa de la relación que hay entre cada par de puntos, y en base a las relaciones existentes entre los puntos se forman los clusters correspondientes con *k-means*. Se definieron dos tipos de restricciones: (i) *Must-link* (deben estar relacionados), y (ii) *Cannot-link* (no deben estar relacionados). Luego, se modificó *k-means* para soportar estas restricciones. Se ha demostrado en [9] que con sólo el primer tipo de restricción, el problema se puede resolver en tiempo polinomial; sin embargo, si se considera también el segundo tipo de restricción, el problema llegará a ser NP-Completo. La principal desventaja de este algoritmo es justamente el preprocesamiento de las relaciones, ya que se necesita conocimientos experto previo, y usualmente no se tienen predefinidas las relaciones.

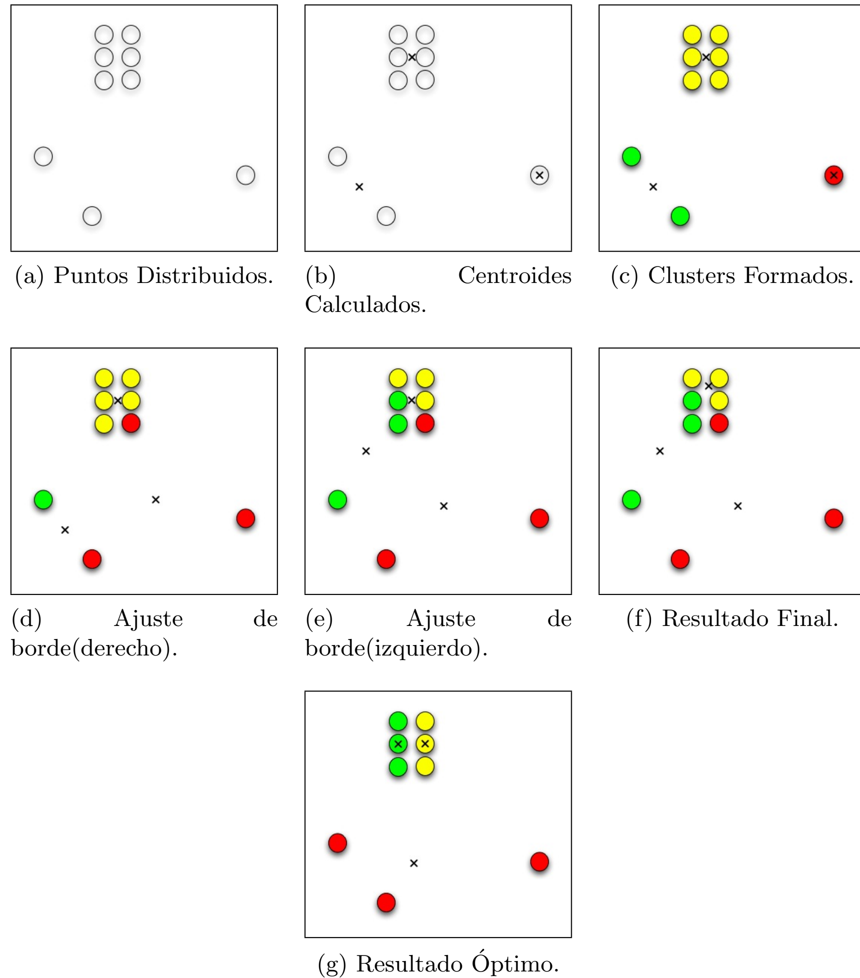
A continuación, se describe un algoritmo propuesto por Han et al. [12] para resolver el problema de *clustering* balanceado.

1.3.2. Clustering balanceado basado en algoritmos de *k-means*

Han et al. [12] propusieron un algoritmo simple para lograr particiones equivalentes de modo que cada partición tenga una cantidad similar de puntos. Básicamente, este algoritmo (Algoritmo 1) calcula los centroides en base a *k-means* [21], luego se le asigna los t puntos más cercanos a cada centroide obtenido para lograr una partición equilibrada. Este último proceso es llamado *Ajuste del Borde*. Este algoritmo presenta dos problemas: (i) al igual que *k-means*, este algoritmo necesita saber el valor de k previamente para iniciar el proceso, (ii) una vez fijados los centroides por *k-means*, la partición se encierra alrededor de ellos y hace imposible explorar otras particiones alternativos con centroides en distintas ubicaciones.

Para dar a entender visualmente el algoritmo, se puede observar que en la Fig. 1.1a están los 9 puntos distribuidos en un plano cartesiano. Se pretende dividir estos 9 puntos en 3 grupos balanceados tal que cada grupo tengan mínimo 3 puntos. En el paso 2 del algoritmo, se determina el centroide (simbolizado con el signo cruz) mediante *k-means* y agrupa los 9 puntos según sus centroides más cercanos (Fig. 1.1b y 1.1c). Luego, se aplica el ajuste del borde empezando por la

parte derecha y después izquierda (Fig. 1.1d y 1.1e). Finalmente, al algoritmo entrega el resultado (Fig. 1.1f). Sin embargo, este algoritmo tiene diferentes falencias que a continuación presentamos.



(a) Puntos Distribuidos. (b) Centroides Calculados. (c) Clusters Formados. (d) Ajuste de borde(derecho). (e) Ajuste de borde(izquierdo). (f) Resultado Final. (g) Resultado Óptimo.

Figura 1.1: El resultado generado por el algoritmo de Han (Algoritmo 1) y el resultado óptimo esperado de clustering balanceado.

1.3.3. Falencias del trabajo de *k-means con Ajuste del Borde*

En esta sección, hablamos sobre las falencias del algoritmo propuesto y del paper de Han et al.

1.3.3.1. Falencias del algoritmo

A pesar de que el Algoritmo 1 pueda entregar un resultado rápido, no siempre entrega el resultado óptimo como puede observarse en la Fig. 1.1g. Esto se debe a que los centroides se fijan al

comienzo del proceso y luego el ajuste de borde depende de la ubicación ya fijada de los centroides. Luego obtiene sólo un mínimo local. El mínimo global, esto es el óptimo, se muestra en la Fig. 1.1g (nótese que los centroides tienen otra ubicación). Ésta es la primera falencia del Algoritmo 1.

La segunda falencia del algoritmo consiste justamente en la falta de claridad de ajuste de borde. En el paso 4 del Algoritmo 1, se puede observar que los clusters con menos puntos tienen derecho a absorber los puntos de otros clusters con más puntos; sin embargo, en el caso de los clusters $C_1, C_2, \ldots, C_k$ tal que uno de ellos tiene los $\frac{n}{2}$ puntos mientras que los clusters restantes contienen cada uno $\frac{n-\frac{n}{2}}{k-1}$, ¿Cuál de los $k-1$ clusters restantes será primero en absorber los puntos del C_k? y ¿Cuál será el segundo? En el paper de Han et al. [12] no se señala claramente cúal es el orden en que los clusters comienzan a absorber los puntos.

1.3.3.2. Falencia de la evaluación del resultado

El trabajo tiene una tercera falencia: la ausencia de la evaluación de los resultados. El autor se preocupa principalmente en "separar los puntos en grupos en forma equitativa visualmente"; pero en su paper carece de una evaluación de resultados de los clusters generados. Esto podría ser importante para la aplicación de la vida real, por ejemplo, los usuarios pueden preferir que los centroides estén lo más lejanos posibles entre sí para que los territorios sean ocupados en forma eficiente. Y este resultado podría ser diferente al separar los puntos en grupos en forma equitativa visualmente.

Dadas estas tres falencias, la presente tesis propone un algoritmo basado en el algoritmo *Clustering Basado en* PSO que busca mejorar la eficacia del algoritmo *k-means con Ajuste del Borde* (Algoritmo 1) para el problema de clustering balanceado.

1.4. Objetivos de la tesis

Objetivo General: Diseño e implementación de cuatro nuevos algoritmos aproximados y competitivos de clustering balanceado usando una heurística basado en *PSO*.

Objetivos Específicos:

1. **Estudio de la factibilidad del empleo de los distintos algoritmos heurísticos**
   Este objetivo corresponde al diseño, la implementación y la complejidad algorítmica de los distintos algoritmos basados en *PSO* factibles para dar solución al problema.

2. **Evaluación**
   Este objetivo consiste en la evaluación de los clusters balanceados generados por los distintos algoritmos en base a las métricas de la sección anterior.

El trabajo está organizado de la siguiente manera. En el capítulo 1 hemos definido el problema de Clustering Balanceado, las métricas estándares de evaluación y las falencias del algoritmo

de Han. En el capítulo 2, describimos la esencia del algoritmo *PSO* basado en el trabajo de Parsopoulos et. al [25]. Junto a éste, describimos un algoritmo de Clustering basado en *PSO* cuya idea será referenciada en la sección siguiente. En el capítulo 3, presentaremos el problema expresado como distintos problemas de Optimización Combinatoria que pertenecen a una clase de problemas aparentemente difíciles desde el punto de vista computacional. Y también presentaremos los distintos algoritmos factibles usados para los problemas de Optimización Combinatoria para dar la solución al problema de *clustering* balanceado junto con un análisis de complejidad algorítmica. En el capítulo 4, realizaremos el testeo de los algoritmos factibles con distintas fuentes de datos distribuidos en forma uniforme para comparar su resultado. En el capítulo 5, evaluamos los resultados producidos por los algoritmos propuestos con datos no distribuidos en forma uniforme. Como apoyo a la revisión, los resultados generados y las evaluaciones en los capítulos 4 y 5 quedan publicados en la siguiente página web: `http://desarrollo.mapcity.com/Tesis`. En el capítulo 6, se describen las conclusiones del presente trabajo.

Capítulo 2

Preliminares: Particle Swarm Optimization (PSO)

El concepto de *Particles Swarms*, aunque fue introducido inicialmente para simular el comportamiento social humano, ha sido muy popular actualmente como una técnica de búsqueda y optimización. De esta forma, con el concepto de Particles Swarms, Kennedy et. al [15] introdujeron un novedoso algoritmo llamado *Particle Swarm Optimization (PSO)* que no necesita la información gradiente para la función que será optimizada, sino las operaciones primitivas matemáticas. Este algoritmo simula el comportamiento colectivo de los pájaros o los peces en base a la siguiente declaración:

"Ningún pájaro sabe dónde está la comida; pero cada uno sabe a qué distancia se encuentra de la comida y también la distancia de sus compañeros a la comida. Entonces los pájaros se mueven hacia el más cercano a la comida durante un tiempo. Luego, cada pájaro vuelve a examinar su distancia a la comida y también la de sus compañeros, y así se mueven hacia el pájaro más cercano a la comida. El examen y el movimiento se repiten en varias iteraciones hasta encontrar la comida."

Cada pájaro X_i representa una *partícula*, que podría ser una posible solución a un problema de optimización, mientras que el conjunto de todos los pájaros S representan la *bandada*. Todos los pájaros definen su posición en base a una función de adaptación ("*fitness function*"). En este caso, la función de adaptación es usada para evaluar la distancia entre el pájaro y la comida. Cada pájaro recuerda todas sus posiciones en los distintos momentos (iteraciones), y así cada pájaro puede evaluar su mejor posición en la historia (mejor posición local) para decidir su siguiente dirección. Como existe siempre un pájaro más cercano a la comida (mejor posicón global) en la bandada, todos los pájaros se mueven hacia éste para encontrar la comida. Luego el movimiento de cada pájaro calcula a partir de su mejor posición histórica y la mejor posición global (Fig. 2.1).

A continuación, presentamos brevemente la evolución de *PSO* en base al trabajo de Parsopoulos et. al [25].

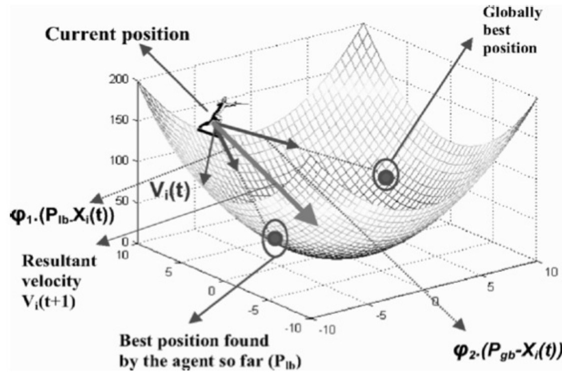


Figura 2.1: Particle Swarm Optimización (figura tomada de [25]).

2.1. La evolución de *PSO*

La primera versión que propusieron Kennedy et. al [15] en 1995 establece que *swarm* se define como el conjunto $S = \{x_1, x_2, \ldots, x_M\}$ de M partículas (soluciones candidatas). Cada partícula se define como: $x_i = (x_{i_1}, x_{i_2}, \ldots, x_{i_m})^T \in A, i = 1, 2, \ldots, M$ donde $A \subset R^m$ (espacio de búsqueda de m-dimensional) y t, la cantidad total de la iteraciones. La función objetivo $f(x)$ se define como la solución candidata generada por A. Cada partícula tiene su valor único de la función, $f_i = f(x_i) \in Y$.

Las partículas se mueven dentro del espacio de búsqueda, A, en forma iterativa. Para lograrlo, es necesario cambiar su posición con alguna forma. El autor introduce el término *Velocidad* para este propósito, que se define como: $v_i = (v_{i_1}, v_{i_2}, \ldots, v_{i_m})^T, i = 1, 2, \ldots, M$. La velocidad también es adaptada en forma iterativa para que las partículas puedan explorar cualquier área en A. De esta manera, la posición y la velocidad de la i-ésima partícula están denotadas como $X_i(t)$ y $V_i(t)$, respectivamente.

La velocidad se actualiza según la información de los pasos previos del algoritmo. Éste se implementa en término de la memoria, donde cada partícula pueda guardar su mejor posición histórica en su movimiento, dentro del espacio de búsqueda. Sea $P = \{p_1, p_2, \ldots, p_M\}$ el conjunto de las posiciones historicas de las partículas y la posición de la i-ésima partícula, y $p_i = (p_{\{i1\}}, p_{\{i2\}}, \ldots, p_{\{im\}})^T \in A, i = 1, 2, \ldots, M$. La mejor posición histórica de ésta se define como: $p_i(t) = \arg\min_t f_i(t)$, lo cual significa obtener la posición histórica de la i-ésima partícula que llegó a tener la mejor solución (o el costo minimizado) entre el tiempo $1 \ldots t$.

Como *PSO* simula la actividad social, las partículas intercambian su experiencia mutuamente. El algoritmo aproxima el minimizador global con la mejor posición de todas las partículas. Sea g el indicador de la mejor posición cuyo valor de la función objetivo es mínimo en P en la iteración t, la mejor posición global se define como: $p_g(t) = \arg\min_t f(p_i(t))$.

Algoritmo 2 Particle Swarm Optimization

1: Inicializar la posición y la velocidad de cada partícula $X_i(t)$ y $V_i(t)$
2: **for** t = 1 hasta máxima iteración **do**
3: **for** i = 1 hasta cantidad de partículas **do**
4: Evaluar la función de adaptación: $= f(X_i(t))$
5: Actualizar $P(t)$ y $g(t)$
6: Adaptar la velocidad de la partícula X_i usando la ecuación 2.1.
7: Actualizar la posición de X_i
8: **end for**
9: **end for**
10: **return** Posición de la mejor partícula X_i a nivel global

Así, la primera versión de *PSO* es definida por las siguientes ecuaciones:

$$\begin{aligned} v_{ij}(t+1) &= v_{ij}(t) + c_1 R_1(p_{ij}(t) - x_{ij}(t)) + c_2 R_2(p_{gj}(t) - x_{ij}(t)) \\ x_{ij}(t+1) &= x_{ij}(t) + v_{ij}(t+1) \end{aligned} \tag{2.1}$$

donde $i = 1, 2, \ldots, M$, $j = 1, 2, \ldots, m$ y R_1 y R_2 son vectores n-dimensionales con valores aleatorios distribuidos uniformemente entre $[0, 1]$; c_1 y c_2 son factores de aprendizaje, conocidos también con el nombre de parámetro cognitivo y parámetro social respectivamente. En la primera versión de *PSO*, c_1 tiene el mismo valor que c_2 en vez de tener dos valores diferentes en la ecuación (2.1).

En cada iteración, despúes de la actualización y la evaluación de las partículas, la mejor posición histórica es actualizada. Y así, la nueva mejor posición de x_i en la iteración $t + 1$ es definida como sigue:

$$p_i(t+1) = \begin{cases} x_i(t+1), & \text{si } f(x_i(t+1)) \leq f(p_i(t)) \\ p_i(t), & \text{en caso contrario.} \end{cases}$$

Despúes, la mejor posición global g se actualiza cuando completa una iteración de *PSO*.

El algoritmo de *PSO* está resumido en el Algoritmo 2. En el paso 1 del algoritmo, las posiciones de las partículas son generadas aleatoriamente de acuerdo a una distribución uniforme en un espacio de búsqueda limitado, es decir, el vector de $X_i(t)$ siempre tiene que estar en un espacio limitado para que la solución siempre esté conforme con las restricciones señaladas del problema a tratar. De acuerdo a la diversidad de los problemas, la forma de la inicialización varía. En el paso 4, la función de adaptación es calculada. En paso 5, 6 y 7, se actualiza su posición local óptima y también su posición global óptima. Finalmente, se retorna la mejor posición de la partícula como resultado.

Para ilustrar *PSO*, considere la Fig. 2.2. Sea $x_i = (0, 0)^T$ (denotado con el símbolo cruz) la posición actual de una partícula, $p_i = (2, 1)^T$ and $p_g = (1, 3)^T$ como la mejor posición histórica y la mejor posición global respectivamente con el símbolo de estrella y cuadrado. Sea también $v_i = 0$. Así, la Fig. 2.2 representa 1000 posibles nuevas posiciones de x_i para $c_1 = c_2 = 1{,}0$ (imagen izquierda) y $c_1 = c_2 = 2{,}0$ (imagen derecha).

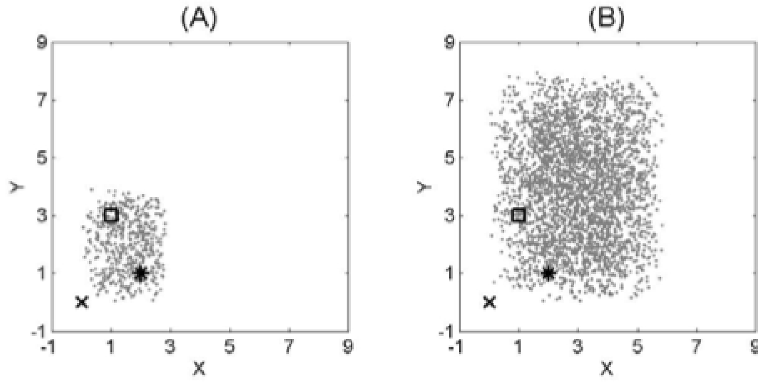


Figura 2.2: La nueva posición candidata de la partícula $x_i = (0,0)^T$ (cruz) con $p_i = (2,1)^T$(estrella) y $p_g = (1,3)^T$(cuadrado), para los casos (A)$c_1 = c_2 = 1{,}0$, y (B) $c_1 = c_2 = 2{,}0$ (figura tomada de [25]).

Aparentemente, la maginitud de búsqueda difiere significativamente en ambos casos. Si uno quiere una mejor exploración global, el valor alto de c_1 y c_2 puede hacer que la partícula se moviese con mayor cobertura en el espacio de búsqueda. En otro caso, si uno quiere una mejor exploración local, el valor de c_1 y c_2 debe bajarse. En caso de $c_1 > c_2$, el movimiento de la partícula tiende a dirigirse hacia su mejor posición local(p_i), mientras que para $c_1 < c_2$, la partícula tiende a dirigirse hacia la mejor posición global(p_g). El valor de estos parámetros variará según la información de los distintos problemas.

Un error típico en la implementación del algoritmo *PSO* es cuando la ecuación 2.1 es considerada de la siguiente forma:

$$\begin{aligned} v_i(t+1) &= v_i(t) + c_1 R_1 (p_i(t) - x_i(t)) + c_2 R_2 (p_g(t) - x_i(t)), \\ x_i(t+1) &= x_i(t) + v_i(t+1) \end{aligned} \tag{2.2}$$

para $i = 1, 2, \ldots, M$. En la ecuación 2.1, R_1 y R_2 deben ser considerados como los vectores n-dimensionales con sus valores distribuidos uniformemente en $[0, 1]$; pero en esta ecuación 2.2, R_1 y R_2 están considerados como valores unidimensionales. Este multiplica su coordenada de la mejor posición histórica de la partícula por el mismo valor de R_1, y a la mejor posición global, por R_2. El efecto de esta conversión de un vector n-dimensional a un valor unidimensional es ilustrado en la Fig. 2.3, donde las 1000 nuevas posiciones de la misma partícula, x_i, fueron generadas con la configuración correcta de la ecuación 2.1 (imagen izquierda) y la incorrecta configuración de la ecuación 2.2 (imagen derecha). Obviamente, esta última restringe las posiciones en una región paralelepípeda entre las dos mejores posiciones, p_i y p_g, por lo cual, al cometer este error, la exploración de las soluciones en el espacio de búsqueda será restringida y habrá mucha probabilidad de dar una solución lejos de la óptima.

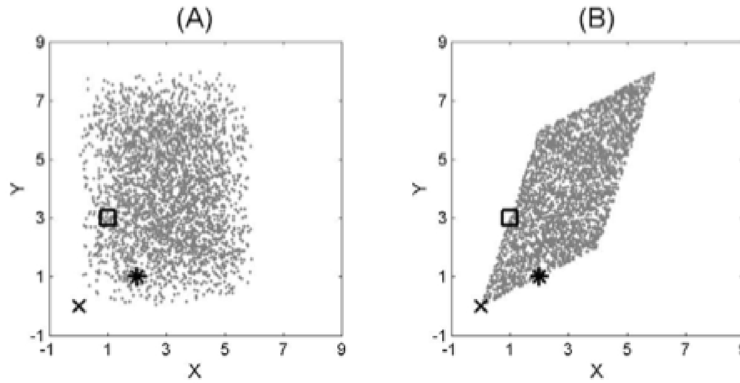


Figura 2.3: Nuevas posiciones candidatas generadas por la posición 2.2 para la partícula $x_i = (0, 0)^T$ (cruz), con $p_i = (2, 1)^T$ (estrella), y $c_1 = c_2 = 2{,}0$, cuando (A) R_1 y R_2 son vectores aleatorios n-dimensional, y (B), valores aleatorios unidimensional (figura tomada de [25]).

2.2. El refinamiento de *PSO*

Pese a que la primera versión de *PSO* puede solucionar satisfactoriamente los problemas de optimización sencillos, al aplicarlo para problemas difíciles con mayor espacio de búsqueda, se presentan varias deficiencias. Por ello, Parsopoulos et. al [25] discuten y reportan algunas mejoras del algoritmo *PSO* que a continuación ponemos.

2.2.1. La explosión de Swarm

El primer problema descubierto por varios investigadores, fue el efecto de la explosión del enjambre. Se trata de un incremento no controlado de la velocidad que se convierte en una divergencia de enjambre. Este error se debe a la falta de un mecanismo que restringe las velocidades en la primera versión de *PSO*.

Para limitar el incremento de la velocidad, se puede definir un parámetro que indica la máxima velocidad permitida $v_{\text{máx}} > 0$ donde $|v_{ij}(t+1)| \leq v_{\text{máx}}$, $i = 1, 2, \ldots, M$, $j = 1, 2, \ldots, m$. En caso de que la velocidad pasara la máxima velocidad, la velocidad es ajustada así:

$$v_{ij}(t+1) = \begin{cases} v_{max}, & \text{si } v_{ij}(t+1) > v_{\text{máx}}, \\ -v_{max}, & \text{si } v_{ij}(t+1) < -v_{\text{máx}} \end{cases}.$$

Cuando la velocidad es restringida, se puede producir eficientemente la solución del problema. Sin embargo, el ajuste de la velocidad no resuelve el problema de la convergencia de la solución, porque las partículas estan condicionadas a moverse cerca de su mejor posición pero no están capacitadas para alcanzar la convergencia. Este problema es resuelto con la introducción de un nuevo parámetro en el modelo original de *PSO* descrito en la siguiente sección.

2.2.2. El concepto del peso de inercia

Aunque el uso de un umbral de velocidad máxima ha mejorado el rendimiento de las primeras variantes de *PSO*, aún no fue suficiente para hacer que el algoritmo sea eficiente en problemas de optimización complejos. A pesar del alivio a la explosión del swarm, el swarm no fue capaz de concentrar sus partículas alrededor de las soluciones más prometedoras en la última fase del procedimiento de optimización con el umbral de velocidad puesta. Para la búsqueda refinada en las regiones prometedoras, es decir, en torno a las mejores posiciones, se requiere una fuerte atracción de las partículas hacia ellas, y pequeños cambios de la posición que prohíben escapar de sus inmediaciones. Para ello, se puede reducir el área del cambio de la posición para que las partículas puedan concentrarse una vez encontradas las mejores posiciones agregando un nuevo parámetro w llamado el peso de inercia en la ecuación 2.1:

$$\begin{aligned} v_{ij}(t+1) &= w(t)v_{ij}(t) + c_1R_1(p_{ij}(t) - x_{ij}(t)) + c_2R_2(p_{gj}(t) - x_{ij}(t)) \\ x_{ij}(t+1) &= x_{ij}(t) + v_{ij}(t+1) \end{aligned} \tag{2.3}$$

donde $i = 1, 2, \ldots, M$, $j = 1, 2, \ldots, m$. Mientras que el valor de w se calcula en la siguiente ecuación:

$$w(t) = w_{up} - (w_{up} - w_{low})\frac{t}{T_{\text{máx}}} \tag{2.4}$$

donde w_{up} se refiere al máximo peso de inercia y w_{low}, el mínimo peso; $t_{\text{máx}}$ es la cantidad máxima de iteraciones en *PSO*. No es difícil obervar que $w(t)$ se va disminuyendo en cada iteración por el término $\frac{t}{T_{max}}$. Por este nuevo parámetro, las partículas tienden a estar rodeadas en las regiones prometedoras sin salirse de ella a medida que se aumente el valor t. En la Fig. 2.4 muestra el efecto de convergencia agregando el peso de inercia con la ecuación 2.3. Se puede observar que a medida que aumente t, el swarm tiende a converger la solución. Para el caso de la ecuación 2.1 sin el peso de inercia, la solución no se converge y las N partículas tienden a continuar explorando su solución por sí solo. Evidentemente, solucionamos eficientemente el problema de la convergencia con el peso de inercia.

2.2.3. El concepto del vecino

La introducción del peso de inercia aumenta la capacidad de convergencia de *PSO*, tal como se describe en la sección anterior. Sin embargo, no fue suficiente para aumentar su eficiencia a los niveles más satisfactorios en problemas complejos y multimodal. Como se muestra en la Fig. 2.4, después de un número de iteraciones, el *Swarm* converge debido a la ausencia completa de la diversidad. Esto implica que una mayor exploración ya no es posible y las partículas sólo pueden realizar búsquedas locales alrededor de su punto de convergencia que, posiblemente, la mayoría se concentra en las cercanías de ésta, y no necesariamente esta convergencia es la mejor solución resultante. Para ello, es bueno que el área en que las partículas se mueven alrededor de la mejor posición global puede

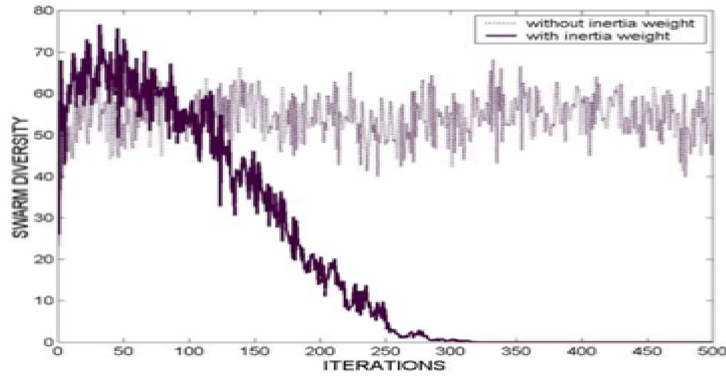


Figura 2.4: La divergencia del Swarm durante la búsqueda con (línea sólida) y sin (línea punteada)el peso de inercia donde $N = 20$, $c_1 = c_2 = 2$, $T_{\text{máx}} = 500$, $v_{\text{máx}} = 50$, $w_{up} = 1{,}2$ y $w_{low} = 0{,}1$ (figura tomada de [25]).

ampliarse para así detectar con mayor probabilidad la mejor solución.

La introducción del nuevo concepto: el vecino. Cada partícula cuenta con un conjunto de las otras partículas que son sus vecinos en cada iteración, y se comunica su mejor posición únicamente a estas partículas en vez de a todo el swarm. Y así, la información sobre la mejor posición global está inicialmente comunicada solo a la mejor posición de los vecinos, y sucesivamente con el resto a través de sus vecinos. Formalmente, los vecinos de la i-ésima partícula x_i están definidos como: $NB_i = \{x_{m_1}, x_{m_2}, \ldots, x_{m_s}\}$ donde $\{m_1, m_2, \ldots, m_s\} \subseteq \{1, 2, \ldots, M\}$ es el conjunto de los índices de sus vecinos. Para determinar quienes son los vecinos de una partícula, existen varias formas. Una simple forma es el calcular la distancia euclidiana entre las partículas, y de acuerdo a la cardenalidad de los vecinos $|NB_i|$, conocido con el nombre de *el volumen de vecino*, se determinan los vecinos. Y así, g_i se define como el índice de la mejor partícula en NB_i: $\underset{x_j \in NB_i}{\arg\min} f(p_j)$. Ahora, la ecuación 2.3 es modificada en la siguiente forma:

$$
\begin{aligned}
&v_{ij}(t+1) = w(t)v_{ij}(t) + c_1R_1(p_{ij}(t) - x_{ij}(t)) + c_2R_2(p_{g_ij}(t) - x_{ij}(t)) \\
&x_{ij}(t+1) = x_{ij}(t) + v_{ij}(t+1)
\end{aligned}
\tag{2.5}
$$

donde $i = 1, 2, \ldots, M$, $j = 1, 2, \ldots, m$. En la Fig. 2.5 se puede observar que la velocidad de convergencia se disminuye con la introducción del concepto de los vecinos, lo cual implica que el área en que se realiza la búsqueda de la solución óptima se amplia, y se puede encontrar con la mejor posición a través de los vecinos , mientras que *PSO* sin el concepto de vecinos converge rápido, está limitada a la mejor posición global de todo el *Swarm*. Esta diferencia aparente se vuelve más intensa a medida que la dimensionalidad del problema incrementa.

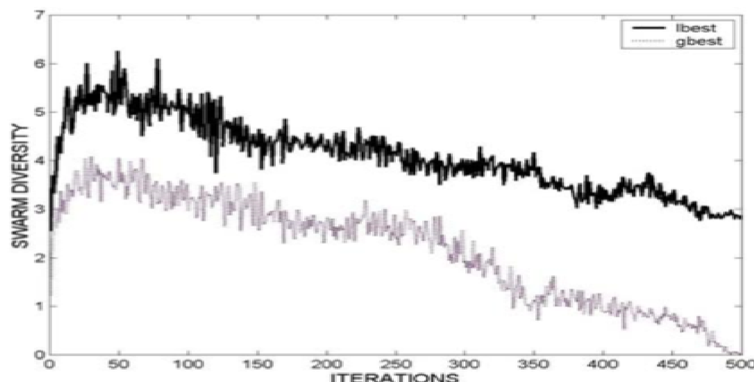


Figura 2.5: La divergencia del Swarm durante la búsqueda con el concepto de los vecinos(lbest, línea solida) y sin los vecinos(gbest, línea punteada) donde $N = 20$, $c1 = c2 = 2$, $T_{\text{máx}} = 500$, $v_{\text{máx}} = 5{,}12$, $w_{up} = 1{,}2$ y $w_{low} = 0{,}1$ y $n = 10$ para miminizar la función de Rastrigin en el rango $[-5{,}12, 5{,}12]^{10}$ con $NB_i = 1$ (figura tomada de [25]).

2.2.4. El valor óptimo de los parámetros

Kennedy [17] hizo el primer estudio teórico sobre las trayectorias de los movimientos de las partículas en el algoritmo *PSO*. Kennedy solamente considera el modelo unidimensional de *PSO* y ha concluido que la suma del valor de c_1 y c_2 en la ecuación 2.1 debe estar en el intervalo (0,4] para converger a la solución. Ozcan et al [24] hicieron el mismo estudio pero sobre el modelo multidimensional de *PSO* y obtuvieron la misma conclusión que Kennedy. Debido a la aparición de los algoritmos variantes de *PSO*, Clerc et. al [8] realizaron análisis teóricos y definieron distintas propiedades, tales como estabilidad, explotación, etc. Trelea [26] concluyó que $w = 0{,}6$ y $c_1 = c_2 = 2{,}83$ ha podido converger en mejores resultados. A continuación, explicaremos la aplicación de *PSO* en el problema del clustering.

2.3. Clustering Basado en *PSO*

Omran et al. [23] propusieron un algoritmo de clustering basado en *PSO* para obtener resultados óptimos. Este algoritmo define una función de adaptación ("*fitness function*") que mide la validez de cada cluster formado y representa a cada partícula como un conjunto de centroides. El Clustering basado en *PSO* está resumido en el algoritmo 3.

La función de adaptación se define en la ecuación 2.6.

$$f(x_i, Z) = w_1 \bar{d}_{max}(Z, x_i) + w_2(Z_{max} - d_{\text{mín}}(x_i)) \tag{2.6}$$

Algoritmo 3 Clustering Basado en *PSO*

1: Inicializar cada partícula $p_i \in P$ con k centroides aleatorios.
2: **for** contador = 1 hasta máxima iteración **do**
3:     **for** $p_i \in P$ **do**
4:         **for** $d \in D$ **do**
5:             Calcular la distancia de d con el centroide de $C_i \in G$.
6:             Asignar d al cluster cuyo centroide está más cerca al d
7:         **end for**
8:         Calcular la función de adaptación.
9:     **end for**
10:     Encontrar la mejor partícula a nivel global y a nivel local.
11:     Calcular el centroide de los *clusters* según la fórmula de actualizar la velocidad y coordenada de PSO.
12: **end for**

$$\bar{d}_{max}(Z, x_i) = \max_{j=1,\ldots,k}\{\sum_{\forall z_p \in C_{ij}} d(z_p, m_{ij})/|C_{ij}|\} \tag{2.7}$$

$$d_{\text{mín}} = \min_{\forall j_1, j_2, j_1 \neq j_2}\{d(m_{ij1}, m_{ij2})\} \tag{2.8}$$

Cada partícula x_i es definida como $x_i = (m_{i1}, m_{i2}, ..., m_{ik})$ donde m_{ij} se refiere al vector del centroide del j-ésimo cluster de la partícula i-ésima. Y así, el grupo representa los *clusters* candidatos con la función $f(x_i, Z)$ donde Z es la matriz que representa la asignación de todos puntos p a los k *clusters*, por ejemplo, cada elemento z_{ijp} indica si el punto d_p pertenece al cluster C_j de la partícula i. Los parámetros w_1 y w_2 son constantes que aluden a la velocidad del aprendizaje. La función $\bar{d}_{max}(Z, x_i)$ es la máxima distancia euclidiana promedio de las partículas a los *clusters* asociados y está definido como la ecuación 2.7. La ecuación 2.8 es la mínima distancia entre cada pareja de *clusters*. A éste ultimo se le multiplica el valor -1, la implicancia de esta operación es justamente minimizar el resultado final de la función, es para que el resultado final sea minimizado. Igualmente, la función de adaptación también podría definirse en base a las distintas métricas presentadas en la sección 1 según las distintas necesidades del problema.

Capítulo 3

Diseño del algoritmo

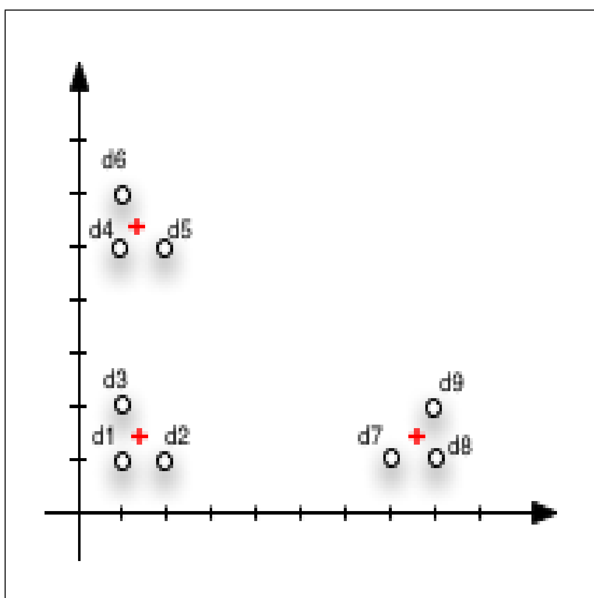
De acuerdo al capítulo anterior, se sabe que cada partícula del algoritmo *PSO* es una solución candidata al problema y a medida que se vaya iterando, todas las soluciones candidatas van a converger en una solución más aproximada a la óptima. Dado que el problema de Clustering Balanceado se basa en la búsqueda la posición óptima de los centroides, entonces se puede aprovechar la característica de la generación aleatoria de los centroides por el algoritmo *PSO* (por ejemplo, el algoritmo 3) y algunos otros algoritmos para asignar los puntos a estos centroides aleatorios generados para proponer una solución candidata. Finalmente, con la convergencia de Algoritmo *PSO*, podemos observar la solución más aproximada a la óptima.

Partimos con un ejemplo para motivar. Vamos a disponer de 9 puntos distribuidos en el plano junto con 3 centroides fijos (Fig. 3.1a). Junto a ello, generamos una matriz de costo de 9×3 formada por la distancia entre cada punto y el centroide (cuadro 3.1). Evidentemente, la asignación de los puntos $\{d_1, d_2, d_3\}$ al centroide c_1, $\{d_3, d_4, d_5\}$ al centroide c_2 y $\{d_6, d_7, d_8\}$ al centroide c_3 resulta ser una mejor asignación porque la suma total de las distancias intra-cluster es igual al 5,91.

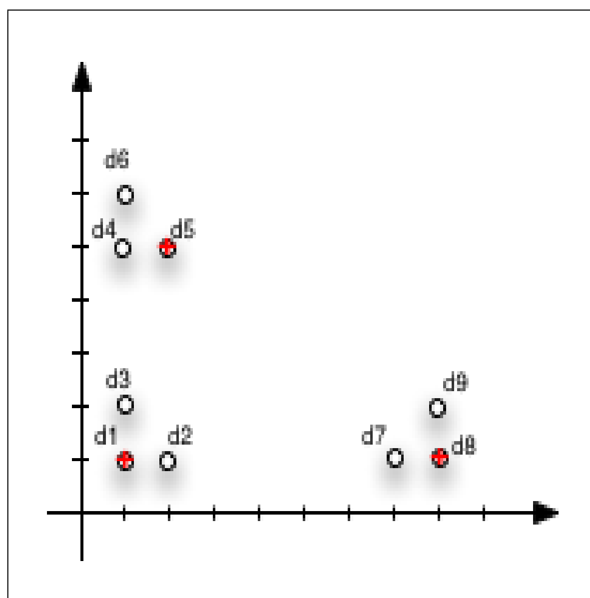
Puntos	$d_1(1,1)$	$d_2(2,1)$	$d_3(1,2)$	$d_4(1,5)$	$d_5(2,5)$	$d_6(1,6)$	$d_7(7,1)$	$d_8(8,1)$	$d_9(8,2)$
c_1(1.3,1.3)	0.47	0.75	0.75	3.68	3.73	4.68	5.68	6.67	6.7
c_2(1.3,5.3)	4.34	4.38	3.34	0.47	0.75	0.75	7.13	7.95	7.45
c_3(7.6,1.3)	6.67	5.67	6.69	7.6	6.75	8.14	0.75	0.47	0.75

Cuadro 3.1: Matriz de Costo de 3×9.

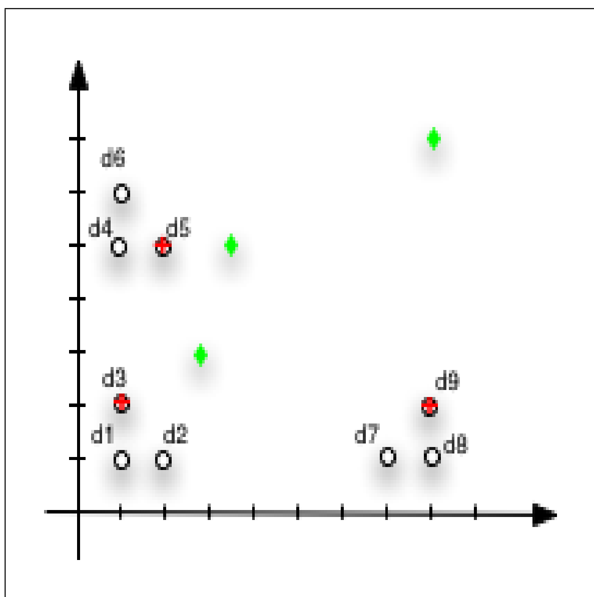
El problema del clustering balanceado, dados los datos del cuadro 3.1, puede dividirse en tres fases: 1) determinar la posición de los centroides, 2) formar la matriz de costo, 3) determinar la mejor asignación de los puntos a los centroides. Las fases 1 y 3 se requieren un diseño de algoritmo eficiente mientras que la fase 2 es simplemente aplicar el cálculo de la distancia euclidiana entre los puntos en tiempo $O(kn)$. A continuación, presentamos las distintas estrategias para la fase 1, y luego los distintos algoritmos existentes para el problema de asignación que es la fase 3.



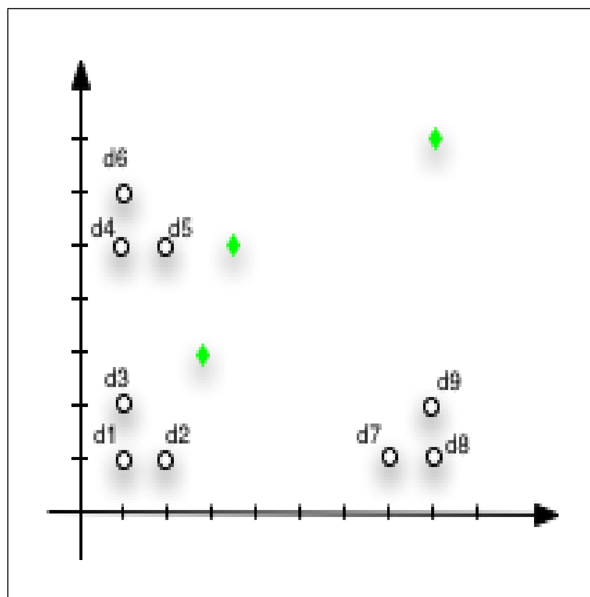
(a) Mejor Resultado.



(b) Fuerza Bruta.



(c) Etiquetar puntos como centroides.



(d) Centroides generados por *PSO*.

Figura 3.1: 9 puntos(círculos) distribuidos en el espacio euclidiano junto con el centroide fijo(cruz) y el centroide generado por *PSO*(diamante).

3.1. Estrategias para determinar la posición de los centroides

En esta sección presentamos tres estrategias principales para determinar los centroides: 1) fuerza bruta, 2) etiquetar a los k puntos como centroides con *PSO* y 3) deteminar la ubicación

espacial de los centroides con *PSO*. Junto a ellas, también analizamos su complejidad algorítmica.

3.1.1. Etiquetar los k puntos como centroides con fuerza bruta

Una estrategia simple, es enumerar las distintas posibles configuraciones de los puntos y etiquetar a los k puntos como centroide, luego calcular la matriz de costo con todos los puntos. Por ejemplo, en la Fig. 3.1c, etiquetamos a los puntos d_1, d_5 y d_8 como centroides c_1, c_2, c_3 y formamos el cuadro de matriz de costo (cuadro 3.2):

Puntos	$d_1(1,1)$	$d_2(2,1)$	$d_3(1,2)$	$d_4(1,5)$	$d_5(2,5)$	$d_6(1,6)$	$d_7(7,1)$	$d_8(8,1)$	$d_9(8,2)$
c_1(1,1)	0	1	1	4	4.12	5	6	7	7
c_2(2,5)	4.12	4	3.16	1	0	1.4	6.4	7.21	6.7
c_3(8,1)	7	6	7	8	7.2	8.6	1	0	1

Cuadro 3.2: Matriz de Costo de 3×9 con los centroides c_1(1,1), c_2(2,5) y c_3(8,1).

Se puede observar que esta estrategia también entrega un el resultado óptimo. Sin embargo, ella necesita computar $\binom{n}{k}$ combinaciones y será ineficiente cuando n es grande por la complejidad algorítimica de $O(n!)$. Para ello, presentamos la segunda estrategia en la siguiente sección.

3.1.2. Etiquetar los k puntos como centroides con *PSO*

En el algoritmo Clustering Basado en *PSO* del capítulo anterior, se puede observar que los centroides son asignados por *PSO*, y efectivamente podemos usar la misma metodología para la generación de los k centroides, y luego calculamos el punto más cercano a éstos y finalmente los etiquetamos como los centroides para la formación de la matriz de costo. Por ejemplo, una partícula de *PSO* generó aleatoriamente 3 posiciones: $(3,3)$, $(8,7)$ y $(1,4)$, entonces los tres puntos más cercanos a ellos (Fig. 3.1c) son: d_3, d_9 y d_5 respectivamente. Luego, d_3, d_9 y d_5 son etiquetado comos centroides c_1, c_2, c_3 y ya se puede formar la matriz de costo(cuadro 3.3) para luego determinar la mejor asignación.

Puntos	$d_1(1,1)$	$d_2(2,1)$	$d_3(1,2)$	$d_4(1,5)$	$d_5(2,5)$	$d_6(1,6)$	$d_7(7,1)$	$d_8(8,1)$	$d_9(8,2)$
c_1(1,2)	1	1.4	0	3	3.1	4	6	7	7
c_2(8,2)	7	6	7	7.6	6.7	8	1.4	1	0
c_3(2,5)	4.1	4	3.1	1	0	1.4	6.4	7.2	6.7

Cuadro 3.3: Matriz de Costo de 3×9 con los centroides c_1(1,2), c_2(8,2) y c_3(2,5).

Por la capacidad de convergencia de *PSO*, los mejores centroides serán encontradas después de $T_{\text{máx}}$ iteraciones. Entonces, para determinar la mejor posición de los centroides, esta estregia requiere de $nkMT_{\text{máx}}$ pasos para cumplir, donde $n = |D|$, k es la cantidad de *clusters* y M la cantidad de

partículas. Como k, $T_{\text{máx}}$ y M son constantes, la complejidad algorítmica de esta estrategia es $O(n)$, muy eficiente comparada con la estregia anterior.

3.1.3. Generar aleatoriamente k posiciones espaciales como centroides con *PSO*

Al igual que la estrategia anterior, esta tercera estrategia ocupa la idea del algoritmo *PSO* que genera aleatoriamente las posiciones de los k centroides en el plano. La única diferencia, es justamente la formación de matriz de costo con estos centroides generados sin etiquetar a los puntos actuales como centroides. Por ejemplo, sean las 3 posiciones genedas en la estrategia anterior: $(3,3)$, $(8,7)$ y $(1,4)$ (Fig. 3.1d). Entonces la matriz de costo es la siguiente(cuadro 3.4):

Puntos	$d_1(1,1)$	$d_2(2,1)$	$d_3(1,2)$	$d_4(1,5)$	$d_5(2,5)$	$d_6(1,6)$	$d_7(7,1)$	$d_8(8,1)$	$d_9(8,2)$
c_1(3,3)	2.82	2.23	2.23	2.82	2.23	3.60	4.47	5.38	5.09
c_2(8,7)	7.81	7.21	7.07	5.38	4.47	5.09	6.08	6.32	5.38
c_3(1,4)	3	3.16	2	1	1.41	2	6.7	7.61	7.28

Cuadro 3.4: Matriz de Costo de 3×9 con los centroides c_1(3,3), c_2(8,7) y c_3(1,4).

Esta estrategia requiere de $kMT_{\text{máx}}$ pasos para cumplir, donde k es la cantidad de *clusters* y M la cantidad de partículas, para determinar la mejor posición de los centroides. Como k, $T_{\text{máx}}$ y M son constantes, la complejidad algorítmica de esta estrategia es $O(1)$, mucho más eficiente que la estrategia anterior.

Una vez determinada la posición de los centroides, ya se puede formar la matriz de costo en $O(kn)$ para luego calcular la mejor asignación de los puntos a los centroides correspondientes. En la siguiente sección, presentamos las distintas estrategias para la fase 3 del problema del Clustering Balanceado: la asginación óptima.

3.2. Estrategias de la asignación óptima

En el cuadro 3.1, la matriz de costo de 3 filas y 9 columnas, se puede observar en total 9 celdas con color gris, ellas son la mejor asignación de los puntos a los centroides porque la suma total de las distancias intra-cluster es la mínima. Este tipo de asignación se asemeja al problema de asignación general donde la única diferencia es que el problema de la asignación general hace el *matching* de uno a uno, es decir, un elemento de la columna es asignado a un elemento de la fila, y cada elemento de la fila, a lo más, puede tener asociado a un elemento de la columna. Mientras tanto, el problema de clustering balanceado, requiere un *matching* de uno a $\frac{n}{k}$ elementos. Como existen varios algoritmos que resuelven el problema de asignación en forma eficiente, si podemos reducir el problema del clustering balanceado al problema de la asignación general, entonces podemos ocupar los algoritmos existentes para lograr resolver el problema de clustering balanceado. En esta sección, presentamos

esta adaptación y luego los distintos algoritmos que resuelven el problema de la asignación como la primera estrategia de asignación óptima. Junto con el análisis de la complejidad algorítmica de la primera estrategia, presentamos, en dos secciones, dos estrategias más que son más intuitivas y ocupa una menor complejidad para la resolución del problema: absorción de puntos cercanos y recorrido de los puntos extremos del envolvente convexo a fin de lograr una mejor eficiencia.

3.2.1. Clustering Balanceado como un problema de asignación

Los problemas de asignación [11] asocian igual número de filas con igual número de columnas dada una matriz de costo. El problema de asignación debe su nombre a la aplicación particular de asignar hombres a trabajos (o trabajos a máquinas), con la condición de que cada hombre puede ser asignado a un trabajo y que cada trabajo tendrá asignada una persona. La condición necesaria y suficiente para la solución de esta clase de problema, es que se encuentre balanceado, es decir, que los recursos totales sean iguales a las demandas totales. Formalmente, el problema puede expresarse así:

$$minimizar \sum_{1\leq i\leq m} \sum_{1\leq j\leq m} A_{ij}X_{ij}$$

$$\sum_{1\leq i\leq m} r_{ij}X_{ij} \leq b_i \; i=1,...,m$$

$$\sum_{1\leq i\leq m} X_{ij} = 1 \; j=1,...,m \tag{3.1}$$

$$X_{ij} = \begin{cases} 1 & \text{si el i-ésimo trabajador ejecuta la j-ésima tarea} \\ 0 & \text{en otro caso} \end{cases}$$

donde b_j es la cantidad de recursos para el i-ésimo trabajador, r_{ij} denota los recursos del trabajador i-ésimo necesarios para realizar la j-ésima tarea y A_{ij}, el costo para que el trabajador i-ésimo lleve a cabo la j-ésima tarea. El primer conjunto de restricciones asegura que no se utilizan más recursos de los que están disponibles para cada trabajador; el segundo conjunto de restricciones afianza el hecho que cada uno de los trabajos lo lleva a cabo un solo trabajador.

El problema del clustering balanceado puede expresarse como un problema de asignación de la siguiente forma: *Dado k centroides y n puntos en el plano. Cada punto ha de ser asignado a un centroide, contrayendo algún coste que puede variar dependiendo de la distancia entre el centroide y los puntos asignados a éste. Es necesario asignar los puntos relativamente cercanos a un mismo cluster para que el coste total de la asignación sea minimizado.*

Con la declaración anterior, para $k = n$, cada punto se forma en sí un cluster, por lo que no es necesario diseñar un algoritmo especial para este caso. Para $k = 1$, tampoco es necesario computar ya los n puntos que en sí componen un cluster. Por lo cual, para $k \in \{1, n\}$ el tiempo de computar es constante y no tiene sentido alguno diseñar un algoritmo al respecto. De esta forma, cuando el

valor de k está entre 2 y $\frac{n}{2}$, se hace necesario diseñar un algoritmo para extraer los $\frac{n}{k}$ puntos de los n puntos y luego calcular la distancia de cada punto al centroide de su cluster correspondiente para formar los k *clusters*. Sin embargo, una condición principal del problema de asginación, es justamente tener la misma cantidad de trabajadores y trabajos, es decir, se debe tener una matriz de costos tal que la cantidad de columnas sea igual a la de las filas. Para ello, se hace necesario igualar los valores de k con n de alguna forma. Un método simple es repetir cada centroide de los k *clusters* $\frac{n}{k}$ veces para que la matriz de costo sea $n \times n$. Por ejemplo, para la matriz de costo de el cuadro 3.1 donde $k = 3$ y $n = 9$, podemos transformarla en la siguiente matriz de costo Φ (cuadro 3.5): Φ_{ij} es la distancia euclidiana entre el centroide k_i y el punto d_j. Luego aplicar

Puntos	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9
c_1	0.47	0.75	0.75	3.68	3.73	4.68	5.68	6.67	6.7
c_2	4.34	4.38	3.34	0.47	0.75	0.75	7.13	7.95	7.45
c_3	6.67	5.67	6.69	7.6	6.75	8.14	0.75	0.47	0.75
c_1'	0.47	0.75	0.75	3.68	3.73	4.68	5.68	6.67	6.7
c_2'	4.34	4.38	3.34	0.47	0.75	0.75	7.13	7.95	7.45
c_3'	6.67	5.67	6.69	7.6	6.75	8.14	0.75	0.47	0.75
c_1''	0.47	0.75	0.75	3.68	3.73	4.68	5.68	6.67	6.7
c_2''	4.34	4.38	3.34	0.47	0.75	0.75	7.13	7.95	7.45
c_3''	6.67	5.67	6.69	7.6	6.75	8.14	0.75	0.47	0.75

Cuadro 3.5: Matriz de Costo de 9×9 con 9 centroides.

el algoritmo que resuelve exactamente el problema de asignación y así se consigue una asignación perfecta entre cada elemento de la fila y de la columna en forma equilibrada con la suma mínima del costo. En el cuadro 3.5, se puede observar que si se suman los valores de las celdas con color gris, el costo es mínimo y se puede deducir que la asignación de los puntos $\{d_1, d_2, d_3\}$ al centroide c_1, de $\{d_3, d_4, d_5\}$ al centroide c_2 y de $\{d_6, d_7, d_8\}$ al centroide c_3 resulta ser una mejor asignación porque la suma total de la intra-cluster distancia es igual al 5,91 al igual que la matriz de costo de el cuadro 3.1.

A continuación, se presenta un algoritmo basado en el algoritmo 3 con el algoritmo Húngaro que resuelve exactamente el problema de la asignación.

3.2.1.1. El Algoritmo Húngaro

El Algoritmo Húngaro [19] modela un problema de asignación como una matriz de costes $m \times m$ (cuadro 3.6), donde cada elemento representa el coste(a_{ij}) de asignar el i-ésimo trabajador(p_i) al j-ésimo trabajo(t_j). Por defecto, el algoritmo realiza la minimización de los elementos de la matriz; de ahí que por ser un problema de minimización de costos, es suficiente con comenzar la eliminación de Gauss-Jordan para hacer ceros (al menos un cero por línea y por columna). Sin embargo, en caso de un problema de maximización del beneficio, el coste de la matriz necesita ser modificado para

que la minimización de sus elementos lleve a una maximización de los valores de coste originales. En un problema de costes infinito, el coste inicial de la matriz puede ser remodelado restando a cada elemento de cada línea el valor máximo del elemento de esa línea (o análogamente columna). En un problema de coste infinito, todos los elementos son restados por el valor máximo de la matriz entera.

Trabajador	t_1	t_2	$\dots$	t_m
p_1	a_{11}	a_{12}	$\dots$	a_{1m}
p_2	a_{21}	a_{22}	$\dots$	a_{2m}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
p_m	a_{m1}	a_{m2}	$\dots$	a_{mm}

Cuadro 3.6: Matriz de Costo del problema de asignación.

Las fases para la aplicación del método Húngaro(Algoritmo 4) [19] son:

1. Encontrar primero el elemento más pequeño en cada fila de la matriz de costos $m \times m$; se debe construir una nueva matriz al restar de cada costo el mínimo de cada fila; encontrar para esta nueva matriz, el costo mínimo en cada columna. A continuación se debe construir una nueva matriz (denominada matriz de costos reducidos) al restar de cada costo el mínimo de su columna.

2. Trazar el número mínimo de líneas (horizontales o verticales) que se requieren para cubrir todos los ceros en la matriz de costos reducidos; si se necesitan m líneas para cubrir todos los ceros, se tiene una solución óptima entre los ceros cubiertos de la matriz. Si se requieren menos de m líneas para cubrir todos los ceros, se debe continuar con el paso 3. El número de líneas para cubrir los ceros es igual a la cantidad de asignaciones que hasta ese momento se pueden realizar.

3. Encontrar el menor elemento diferente de cero (llamado q) en la matriz de costos reducidos, que no está cubierto por las líneas dibujadas en el paso 2; a continuación se debe restar q de cada elemento no cubierto de la matriz de costos reducidos y sumar q a cada elemento de la matriz de costos reducidos cubierto por dos líneas (intersecciones). Por último se debe regresar al paso 2.

A continuación, presentamos un algoritmo que combina *PSO* junto al algoritmo Húngaro.

3.2.1.2. Modificación del algoritmo de Clustering Basado en *PSO* con la idea del algoritmo Húngaro

La idea principal de esta modificación es justamente aprovechar la ubicación aleatoria de los k centroides y la capacidad de convergencia de *PSO* para luego computar la mínima intra-cluster

Algoritmo 4 Algoritmo Húngaro: ahungaro(A)

Require: $A \Leftarrow$ matriz de costo de $m \times m$
Ensure: Tabla T con la mejor asignación entre los elementos de la fila y de la fila
1: Reste el menor número en cada fila de cada número en la fila $\in A$. Anotar los resultados en una tabla nueva A' de $m \times m$. //Esto se llama una reducción de la fila.
2: Reste el menor número en cada columna de la nueva tabla A' de cada número en la columna $\in A'$. Anotar los resultados en una nueva otra tabla A''. //Esto se llama una reducción de la columna.
3: Probar si una asignación óptima se puede lograr en A''. Para ello, se determina el número mínimo de líneas necesarias para cubrir todos los ceros.
4: $T \Leftarrow A''$
5: **while** El número de líneas tachadas es menor que m (número de filas) **do**
6: Modificar la tabla T de la siguiente manera:
7: Reste el número más pequeño cubierto desde todos los números descubiertos en la tabla.
8: Añadir el número más pequeõ cubierto a los números en las intersecciones de las líneas tachadas.
9: Trasladar solamente los números tachados a una nueva tabla A''' a excepción de las intersecciones de las líneas cruzadas.
10: $T \Leftarrow A'''$
11: **end while**
12: Hacer las asignaciones. Comience por las filas o columnas con un solo cero. Hacer *matching* los elementos que tienen ceros, con sólo un *matching* para cada fila y cada columna. Tachar la fila y la columna despus del *matching*.
13: **return** T

distancia con el algoritmo de Húngaro (el Algoritmo 5). El paso 4 de éste consiste justamente en la construcción de la matriz de costo A en $O(nk)$. Luego, se aplica el algoritmo Húngaro a la matriz A y se obtiene el resultado $R = \{\{c_1, d_a\}, \{c_2, d_b\}, \ldots, \{c_k, d_c\}\}$ donde c_i corresponde al cluster i y $d_j \in D$. El paso 6 corresponde recalcular la ubicación de los k centroides aleatorios de la partícula con R, es decir, una vez teniendo el *matching* de centroide y punto, ya podemos saber a cuál centroide corresponde cada punto, y el centroide del cluster no necesariamente coincide con la ubicación inicial del centroide c_k, y así con los t puntos pertenecientes a cada centroide c_k, podemos obtener el promedio del eje x e y, y éste reemplaza al centroide c_k para luego calcular el valor exacto de la suma total de los costos(distancia entre cada punto y el centroide correspondiente). Los pasos $4, 5, 6, 7$ se repiten hasta la cantidad máxima de iteraciones.

En el algoritmo 5, la construcción de la matriz de costo es en tiempo $O(kn)$. El algoritmo Húngaro resuelve el problema de asignación en tiempo $O(n^3)$ donde $n = |D|$. Con la máxima iteración q y la cantidad inicial de partículas de P, el algoritmo 5 ocupa en total $O(pq(kn + n^3))$. Evidentemente, este algoritmo no funcionaría eficientemente cuando el valor de n es grande. Para bajar la complejidad, se puede adaptar el problema de Clustering Balanceado en el problema de los matrimonios estables. La siguiente sección presenta otro algoritmo basado en Algoritmo 3 con

Algoritmo 5 Clustering Basado en *PSO-Hungaro*

1: Inicializar cada partícula $p_i \in P$ con k centroides aleatorios.
2: **for** contador = 1 hasta máxima iteración **do**
3: **for** $p_i \in P$ **do**
4: $A \Leftarrow$ la matriz de costo de $n \times n$ con los k centroides aleatorios.
5: $R \Leftarrow ahungaro(A)$.
6: Recomputar la locación de los k centroides.
7: Calcular la función de adaptación.
8: **end for**
9: Encontrar la mejor partícula a nivel global y a nivel local.
10: Calcular el centroide de los *clusters* según la fórmula de actualizar la velocidad y coordenada de PSO.
11: **end for**

la metodología que resuelve el problema de los matrimonios estables.

3.2.2. Clustering Balanceado como un problema de matrimonios estables

En el problema de los matrimonios estables [13] se busca encontrar un *matching* entre agentes de dos conjuntos disjuntos que cumpla con la propiedad de estabilidad. Se dice que un *matching* es estable si no existe ningún par de agentes que hubieran preferido estar juntos, antes que con su pareja actual. Si existiera dicho par, el mismo es denominado par bloqueante (cuando hay dos individuos a y b quienes fueron asignados a A y B respectivamente; pero b prefiere a A y a prefiere a B). Este escenario generalmente provee un modelo para una variedad de problemas con interpretación práctica. En el problema clásico de los matrimonios estables, hay dos conjuntos de *igual tamaño*, uno formado por hombres y el otro por mujeres, donde cada individuo tiene una lista de preferencias estrictamente ordenada con respecto a todos los integrantes del conjunto opuesto. Bajo estas condiciones se considera un matrimonio al conjunto de las relaciones uno a uno de cada mujer con cada hombre y se considera que el matrimonio es estable si no existe ningún par bloqueante.

Con la descrición mencionada anteriormente, el problema del clustering balanceado puede expresarse como un problema de matrimonios estables de la siguiente manera: *Sean n centroides y n puntos. Cada centroide tiene su lista de preferencias de los puntos según su distancia (ordenado de menor a mayor), y cada punto también tiene su propia lista de preferencia de los centroides según su distancia a éstos. Asignar los puntos relativamente cercanos a un mismo cluster de modo que ningún emparejamiento esté inestable.*

Al igual que el problema de la asignación, el problema de los matrimonios estables requiere formar dos matrices de preferencia de $n \times n$, una que describe la lista de preferencia de los k centroides a los n puntos, y otra, la lista de preferencia de los n puntos a los k centroides. Se entiende la lista de preferencia como el orden ascendente de la distancia de los puntos a los centroides o de los

centroides a los puntos. Nuevamente, podemos reusar el método de conversión tratado en la sección anterior: multiplicar los k centroides en $\frac{n}{k}$ veces en ambas matrices. Por ejemplo, para la matriz de costo de el cuadro 3.1 donde $k = 3$ y $n = 9$, podemos transformarla en dos matrices de preferencia Φ y Φ':

Centroides	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9
c_1	1	2	3	4	5	6	7	8	9
c_2	5	6	4	1	2	3	7	9	8
c_3	5	4	6	8	7	9	2	1	3
c_1	1	2	3	4	5	6	7	8	9
c_2	5	6	4	1	2	3	7	9	8
c_3	5	4	6	8	7	9	2	1	3
c_1	1	2	3	4	5	6	7	8	9
c_2	5	6	4	1	2	3	7	9	8
c_3	5	4	6	8	7	9	2	1	3

Cuadro 3.7: Matriz de Preferencia Φ.

Puntos	c_1	c_2	c_3	c_1	c_2	c_3	c_1	c_2	c_3
d_1	1	4	7	2	8	6	3	6	9
d_2	1	4	7	2	8	6	3	6	9
d_3	1	4	7	2	8	6	3	6	9
d_4	4	1	7	5	2	8	6	3	9
d_5	4	1	7	5	2	8	6	3	9
d_6	4	1	7	5	2	8	6	3	9
d_7	7	4	1	8	5	2	9	6	3
d_8	7	4	1	8	5	2	9	6	3
d_9	7	4	1	8	5	2	9	6	3

Cuadro 3.8: Matriz de Preferencia Φ'.

donde Φ_{ij} es el orden de preferencia según la distancia del punto d_j hacia el centroide c_i, y Φ'_{ij} según la distancia del centroide c_i al punto d_j. Luego aplicar el algoritmo existente que resuelve exactamente el problema de matrimonio estable con ambas matrices de preferencia y obtenemos el *matching* perfecto entre cada elemento de la fila y de la columna. En el cuadro 3.7 y el cuadro 3.8 , se puede observar que la asignación de los puntos $\{d_1, d_2, d_3\}$ al centroide c_1, de $\{d_3, d_4, d_5\}$ al centroide c_2 y de $\{d_6, d_7, d_8\}$ al centroide c_3 resulta ser una mejor asignación porque ninguna asignación presenta la inestabilidad, y si suman la distancia que representa cada celda con color gris, el valor total de la intra-cluster distancia es igual al 5,91 al igual que la matriz de costo de el cuadro 3.1. A continuación mostramos el algoritmo de Gale-Shapley [13].

3.2.2.1. Algoritmo Gale-Shapley

La idea principal de este algoritmo es que cada hombre propone su primera elección, luego las mujeres con dos o más propuestas responden rechazando a todos menos al más favorable. Después los rechazados proponen su segunda elección. Los que no fueron rechazados continúan con su propuesta. Y así, se repiten las propuestas hasta que ninguna propuesta sea rechazada(Algoritmo 6). Cabe destacar que el emparejamiento obtenido es óptimo para el conjunto que hace las propuestas, y es el peor para el conjunto que las recibe. En este caso, el emparejamiento obtenido es óptimo-femenino. Si invertimos los roles en el algoritmo, de forma tal que la mujer realice las propuestas, el emparejamiento resultante será óptimo-masculino. Observando el algoritmo 6 puede verse que el mismo posee ciertos elementos no determinísticos en el sentido del orden en el cual cada hombre libre hace sus propuestas. A continuación presentamos el algoritmo de Clustering Balanceado basado en *PSO* con la idea del algoritmo Gale-Shapley.

Algoritmo 6 Algoritmo Gale-Shapley: aGaleShapley(A, A')

Require: $A \Leftarrow$ matriz de preferencia de hombres a mujeres; $A' \Leftarrow$ matriz de preferencia de mujeres a hombres
Ensure: El mejor emparejamiento entre los hombres y las mujeres
1: Cada persona está libre.
2: **while** algún hombre $h \in A$ esté libre **do**
3: $m \Leftarrow$ primera mujer de la lista de preferencias de h que no ha sido propuesta.
4: **if** m está libre **then**
5: Asignar h a m
6: **end if**
7: **if** m prefiere a h antes que a su pareja actual h' **then**
8: Romper la relación m-h' y asociar m con h
9: **else**
10: El *matching* obtenido es estable.
11: **end if**
12: **end while**

3.2.2.2. Algoritmo PSO combinado con el algoritmo Gale-Shapley

La idea principal de este algoritmo es nuevamente aprovechar la ubicación aleatoria de los k centroides y la capacidad de convergencia de *PSO* para luego computar la mínima intra-cluster distancia con el algoritmo de Gale-Shapley (el Algoritmo 7). A diferencia del algoritmo *PSO-Húngaro*, este algoritmo computa dos matrices de preferencia y el contenido de estas matrices son números enteros enumerados de 1 hasta n. El paso $4, 5$ del algoritmo 7 consiste justamente en la construcción de las matrices de preferencia A y A' en $O(nk)$. Luego, se aplica el algoritmo Gale-Shapley y se obtiene el resultado $R = \{\{c_1, d_a\}, \{c_2, d_b\}, \ldots, \{c_k, d_c\}\}$ donde c_i corresponde al cluster i y $d_j \in D$. Al igual que el algoritmo 5, el algoritmo recomputar la ubicación de los

k centroides aleatorios de la partícula con R, para reemplazar al centroide c_k original para luego calcular el valor de la suma total de los costos. Los pasos $4, 5, 6, 7, 8$ se repite hasta la cantidad máxima de iteraciones.

Algoritmo 7 Clustering Basado en *PSO-Gale-Shapley*

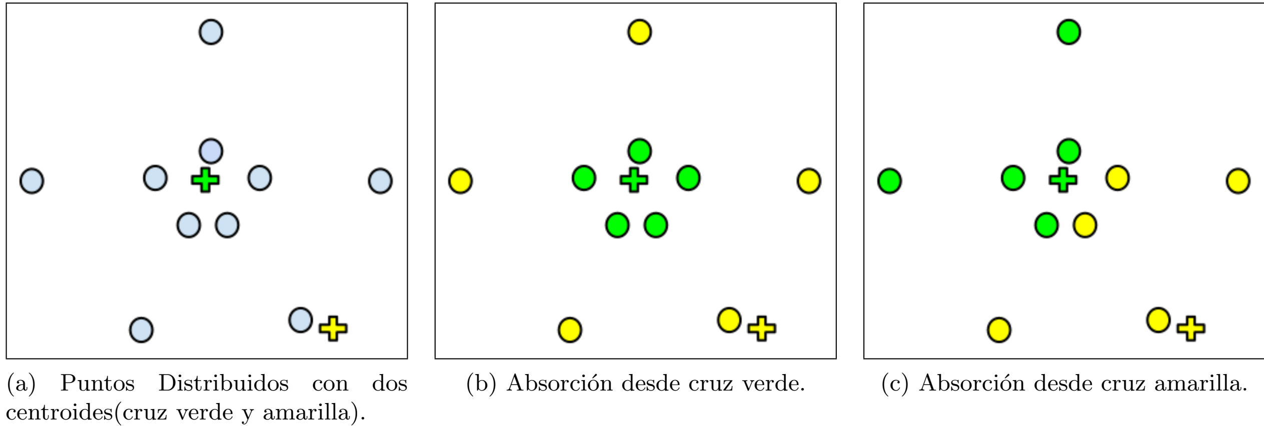
1: Inicializar cada partícula $p_i \in P$ con k centroides aleatorios.
2: **for** contador = 1 hasta máxima iteración **do**
3:   **for** $p_i \in P$ **do**
4:     $A \Leftarrow$ la matriz de preferencias de $n \times n$ de los k centroides aleatorios.
5:     $A' \Leftarrow$ la matriz de preferencias de $n \times n$ de los n puntos.
6:     $R \Leftarrow$ aGaleShapley(A,A')
7:     Recalcular la ubicación de los k centroides.
8:     Calcular la función de adaptación.
9:   **end for**
10:   Encontrar la mejor partícula a nivel global y a nivel local.
11:   Calcular el centroide de los *clusters* según la fórmula de actualizar la velocidad y coordenada de PSO.
12: **end for**

En el Algoritmo 7, la construcción de la matriz de costo es en tiempo $O(kn)$. El algoritmo Gale-Shapley resuelve el problema de asignación en tiempo $O(n^2)$ donde $n = |D|$. Con la máxima iteración q y la cantidad inicial de partículas p, el aAgoritmo 7 ocupa en total $O(pq(kn + n^2))$. Comparado con el Húngaro, el algoritmo Gale-Shapley es relativamente rápido. Sin embargo, igual no funcionaría eficientemente cuando el valor de n es grande. A continuación, presentamos dos estrategias que pretende bajar la complejidad algorítmica, nuevamente basado en Algoritmo 3 con la idea de absorción de puntos más cercanos y la envolvente encerrada respectivamente.

3.2.3. Algoritmo PSO con absorción de los puntos más cercanos

La idea básica de este algoritmo, es aprovechar la característica de la distribución aleatoria de los k centroides con *PSO* y formar la matriz de costo. Luego, los centroides, ordenados descendentemente de acuerdo a su distancia al punto central del mapa, comenzarán a absorber los puntos más cercanos a ellos mismos. La razón principal de este orden descendente, es para evitar una separación inoportuna (Fig. 3.2). Se puede ver que si se comienza la absorción desde el punto más cercano al centro del mapa, va a formar una separación inoportuna (Fig. 3.2b), mientras que por el punto más lejano al mapa central, podemos evitarla (Fig. 3.2c) Finalmente, recalcular la ubicación de los k centroides y calcular el valor de la suma total de los costos al igual que los dos algoritmos (5,7) anteriores. El Algoritmo 8 presenta el Clustering Balanceado basado en *PSO* absorción de los puntos más cercanos. En este algoritmo, hay 2 pasos importantes. El primer paso(Paso 4) es ordenar en forma descendiente los k centroides de acuerdo al punto central del mapa de la distribución de los puntos en D. La implicancia de este paso, es facilitar la absorción

de los puntos de D desde los centroides más extremos para evitar el desequilibrio de la distrubición de los puntos. El segundo paso (Paso 6) es ordenar a todos los puntos de D en forma ascendiente de acuerdo a su distancia al centroide c_j para luego absorber los puntos más cercanos. El paso 11 es igual que los dos algoritmos anteriores.



(a) Puntos Distribuidos con dos centroides(cruz verde y amarilla).

(b) Absorción desde cruz verde.

(c) Absorción desde cruz amarilla.

Figura 3.2: La absorción de los puntos cercanos a paritir del centriode más cercano al punto central del mapa(cruz verde) y del centroide más lejano al centro del mapa(cruz amarilla).

En el Algoritmo 8, el ordenamiento de los k centroides es en tiempo $O(k \log k)$, y el de los n puntos es en $O(n \log n)$. El tiempo de la absorción de los puntos es en $O(\frac{n}{k})$. Con la máxima iteración q y la cantidad inicial de partículas p, el Algoritmo 8 ocupa en total $O(pq(k \log k * ((n \log n) + \frac{n}{k})))$. Comparado con los dos algoritmos anteriores, la complejidad se baja en el parámetro n a $O(n \log n)$. Es mucho más eficiente esta estrategia que la estrategia anterior basada en la adaptación del problema. A continuación, presentamos una estrategia competitiva que se ejecuta también en tiempo $O(n \log n)$.

3.2.4. Algoritmo PSO con la envolvente convexa

La envolvente convexa (en inglés "*convex hull*"), es uno de los más fundamentales constructores geométricos. Una idea intuitiva del significado de la envolvente convexa es el contenido de la figura que formaría una banda elástica que rodeara a una nube de puntos una vez que la soltáramos. El problema de computar la envolvente convexa no sólo está centrado en aplicaciones prácticas, sino también es un vehículo para la solución de un número de cuestiones aparentemente sin relación con él, que surgen en la Geometría Computacional. La computación de la envolvente convexa de una nube finita de puntos, especialmente en el plano, ha sido exhaustivamente estudiada y tiene aplicaciones, como por ejemplo, en el procesado de imágenes y en localización. No es posible construir la definición intuitiva de envolvente convexa citada anteriormente de forma natural, por lo que hay que identificar las nociones apropiadas que nos conduzcan a un algoritmo. Hay varios

Algoritmo 8 Clustering Basado en *PSO-Absorcion-Puntos-Cercanos*

1: Inicializar cada partícula $p_i \in P$ con k centroides aleatorios.
2: **for** contador = 1 hasta máxima iteración **do**
3: **for** $p_i \in P$ **do**
4: Ordenar los k centroides descendentemente según su cercanía al centro del mapa de la distrubución de los puntos
5: **for** $c_j \in p_i$ **do**
6: ordenar los puntos de D ascendentemente de acuerdo a su cercanía al centroide c_j
7: **while** $|c_j| < \frac{n}{k}$ y $D \neq \emptyset$ **do**
8: asignar d_x al centroide c_j
9: $D \Leftarrow D - d_x$
10: **end while**
11: **end for**
12: Recomputar la locación de los k centroides.
13: Calcular la función de adaptación.
14: **end for**
15: Encontrar el mejor partícula a nivel global y a nivel local.
16: Calcule el centroide de los *clusters* según la fórmula de actualizar la velocidad y coordenada de PSO.
17: **end for**

métodos para el cálculo de la envolvente convexa: *Algoritmo Quick Hull* [5], *Scan de Graham* [2], *Envolvimiento de regalo (Gift Wrapping)* [3], *Algoritmo incremental* [7], *Divide y vencerás* [7] y *Algoritmo Kirkpatrick-Seidel* [18] cuyo rendimiento óptimo fue mostrado posteriormente por Afshani et al [1]. Aquí utilizamos la implementación del *Algoritmo Quick Hull* para el desarrollo de esta estrategia.

Dada la característica de aleatoriedad del algoritmo *PSO*, podemos ordenar los puntos en un orden determinado y luego comenzar a agrupar los puntos más cercanos para la formación del cluster. Podemos ordenar los puntos según su posición en las capas de envolvente convexa. En la Fig. 3.3, se muestra en el plano los 31 puntos iniciales. Al computar su envolvente convexa, obtenemos la primera capa que está compuesta por 8 puntos; luego quitamos éstos puntos y nos quedan 23 puntos. A éstos últimos nuevamente computamos su envolvente convexa y obtenemos la segunda capa de envolvente convexa formada por 7 puntos. Y así sucesivamente hasta que todos los puntos sean incluidos al menos en una capa. Finalmente, de afuera hacia adentro(para evitar la separación inoportuna, Fig. 3.2), ordenar aleatoriamente(con *PSO*) el orden de la inserción de los 8 puntos de la primera capa(la capa más externa) en la lista L_1; ordenar aleatoriamente los 7 de la segunda capa y lo insertamos en L_2, y así sucesivamente con las Q capas en las listas $\{L_3, L_4, \ldots, L_8\}$ hasta que todos los puntos del plano sean insertados.

En la segunda etapa, podemos reutilizar la idea de *absorción de los puntos más cercanos.* Una vez ordenados aleatoriamente los puntos de cada capa en la lista $L = \{L_1, L_2, \ldots, L_Q\}$ donde Q es igual a la cantidad de capas de la envolvente convexa, elegimos el primer punto de la lista

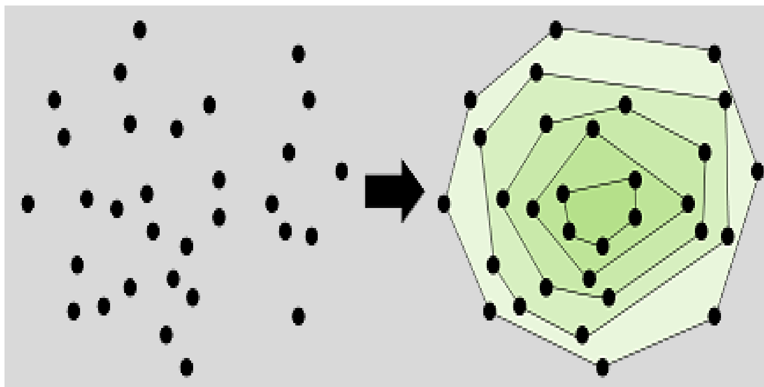


Figura 3.3: Capas de la envolvente convexa.

y agrupamos los $\frac{n}{k}$ puntos cercanos a éste. Luego, elegir el segundo punto de la lista tal que éste último no esté previamente asignado a otro cluster y también agrupamos los $\frac{n}{k}$ puntos cercanos a éste. Y así sucesivamente hasta que todos los puntos hayan sido agrupados. Luego computamos la locación de los k centroides en cada uno de los *clusters* y calculamos la función de adaptación al igual que los tres algoritmos anteriores. Finalmente, nos quedamos con la mejor posición de la partícula como resultado. A continuación, ilustramos la idea de *Algoritmo PSO con la envolvente convexa.*

En la Fig. 3.4 se puede observar que existen 5 capas junto con 31 puntos en total en un plano donde cada capa L_x está formada por los distintos puntos d_i. Con *PSO*, le asignamos a cada d_i una probabilidad P (cuadro 3.9). Luego, ordenamos los puntos d_i de cada capa según las probabilidades y tenemos el cuadro 3.10. Ahora, sea $k = 5$ y cada cluster debe tener mínimo 5 puntos, comenzamos a absorber los puntos más cercanos desde el punto d_4 de L_1, y tenemos el primer cluster $C_1 = \{d_4, d_{11}, d_{19}, d_{24}, d_{20}, d_{29}\}$, luego quitamos los puntos $d_i \in C_1$ en cada capa L_x y generan el nuevo cuadro 3.11. Ahora comenzamos a absorber los puntos más cercanos desde el punto d_1 de L_1, generamos el segundo cluster $C_2 = \{d_1, d_8, d_9, d_{16}, d_{17}, d_{23}\}$ y luego volvemos a quitar los puntos $d_i \in C_2$ y quedamos con el nuevo cuadro 3.12. A continuación, seguimos absorbiendo los 6 puntos más cercanos desde el punto d_7 y generamos el tercer cluster $C_3 = \{d_7, d_{14}, d_{15}, d_{22}, d_{26}, d_6\}$ y damos lugar al cuadro 3.13. Para generar el cuarto cluster, continuamos absorbiendo los 6 puntos más cercanos desde el punto d_2 y tenemos $C_4 = \{d_2, d_{10}, d_{18}, d_3, d_{28}, d_{30}\}$, luego generamos el cuadro 3.14. Finalmente, el cluster $C_5 = \{d_5, d_{12}, d_{13}, d_{21}, d_{25}, d_{27}, d_{31}\}$. Los 5 *clusters* se muestran en la Fig. 3.5.

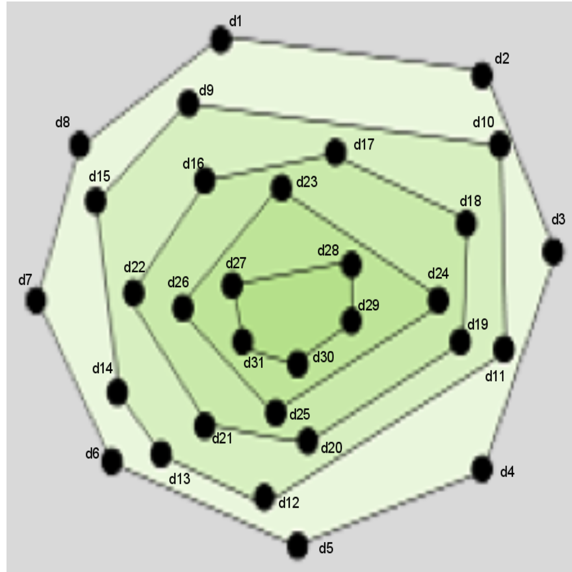


Figura 3.4: Las 8 Capas de la envolvente convexa con 31 puntos.

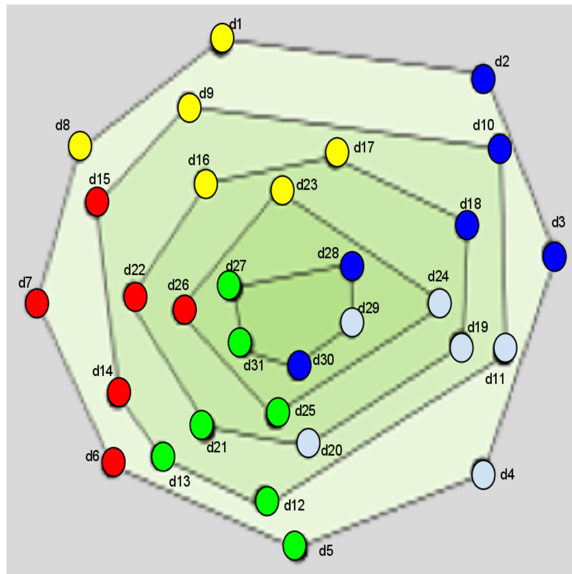


Figura 3.5: El clustering balanceado para 31 puntos para $k = 5$.

El algoritmo de Clustering Balanceado basado en *PSO* con la idea de la envolvente convexa se muestra en el Algoritmo 9, donde el ordenamiento de los puntos con el algoritmo de la envolvente convexa es en tiempo $O(n \log n)$, y el de asginar los t más cercanos a un punto es en $O(n \log n)$. Como se tiene máximo k grupos, la máxima iteración q y la cantidad inicial de partículas p, el algoritmo 8 ocupa en total $O(pq(n \log n + k(n \log n)))$. Comparado con los dos algoritmos anteriores, la complejidad se baja a $O(n \log n)$. El cual es competitivo con el algoritmo 8.

Algoritmo 9 Clustering Basado en *PSO-CONVEX-HULL*

1: $L \Leftarrow \emptyset$
2: $cortes \Leftarrow \emptyset$
3: **while** $D \neq \emptyset$ **do**
4: $L' \Leftarrow convexHull(D)$
5: $D \Leftarrow D - L'$
6: $cortes \Leftarrow cortes \cup |L'|$
7: $L \Leftarrow L \cup L'$
8: **end while**
9: Inicializar cada partícula $p_i \in P$ tal que la dimensión de p_i es igual a n.
10: **for** contador $=$ 1 hasta máxima iteración **do**
11: **for** $p_i \in P$ **do**
12: $corteInicial \Leftarrow 0$
13: $C \Leftarrow \emptyset$
14: **for** $corte \in cortes$ **do**
15: ordenar $d_{corteInicial,corte}$ según la posición de entre *corteInicial* y *corte* en p_i
16: $corteInicial \Leftarrow corteInicial + corte$
17: **end for**
18: **while** $L \neq \emptyset$ **do**
19: $c \Leftarrow \emptyset$
20: $d_{aux} \Leftarrow L.elementoEn(0);$
21: ordenar los puntos de D ascendentemente de acuerdo a su cercanía al d_{aux}
22: **while** $|c| < \frac{n}{k}$ y $L \neq \emptyset$ **do**
23: asignar d_x al cluster c
24: $L \Leftarrow L - d_x$
25: **end while**
26: $C \Leftarrow C \cup c$
27: **end while**
28: Recomputar la locación del centroide de los k *clusters*.
29: Calcular la función de adaptación.
30: **end for**
31: Encontrar el mejor partícula a nivel global y a nivel local.
32: Calcule el centroide de los *clusters* según la fórmula de actualizar la velocidad y coordenada de PSO.
33: **end for**

Capa		Probabilidades							
L_1	$d_i \in L_1$	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8
	$P(d_i)$	0.3	0.4	0.7	0.2	0.8	0.6	0.3	0.6
L_2	$d_i \in L_2$	d_9	d_{10}	d_{11}	d_{12}	d_{13}	d_{14}	d_{15}	
	$P(d_i)$	0.5	0.3	0.8	0.3	0.4	0.4	0.8	
L_3	$d_i \in L_3$	d_{16}	d_{17}	d_{18}	d_{19}	d_{20}	d_{21}	d_{22}	
	$P(d_i)$	0.7	0.2	0.5	0.7	0.3	0.3	0.7	
L_4	$d_i \in L_4$	d_{23}	d_{24}	d_{25}	d_{26}				
	$P(d_i)$	0.8	0.4	0.3	0.2				
L_5	$d_i \in L_5$	d_{27}	d_{28}	d_{29}	d_{30}	d_{31}			
	$P(d_i)$	0.9	0.7	0.1	0.2	0.6			

Cuadro 3.9: Probabilidades de los puntos $d_i \in L_x$ donde $x = 1, 2, 3, 4$ y 5.

Capa		Probabilidades							
L_1	$d_i \in L_1$	d_4	d_1	d_7	d_2	d_6	d_8	d_3	d_5
	$P(d_i)$	0.2	0.3	0.3	0.4	0.6	0.6	0.7	0.8
L_2	$d_i \in L_2$	d_{10}	d_{12}	d_{13}	d_{14}	d_9	d_{11}	d_{15}	
	$P(d_i)$	0.3	0.3	0.4	0.4	0.5	0.8	0.8	
L_3	$d_i \in L_3$	d_{17}	d_{20}	d_{21}	d_{18}	d_{16}	d_{19}	d_{22}	
	$P(d_i)$	0.2	0.3	0.3	0.5	0.7	0.7	0.7	
L_4	$d_i \in L_4$	d_{26}	d_{25}	d_{24}	d_{23}				
	$P(d_i)$	0.2	0.3	0.4	0.8				
L_5	$d_i \in L_5$	d_{29}	d_{30}	d_{31}	d_{28}	d_{27}			
	$P(d_i)$	0.1	0.2	0.6	0.7	0.9			

Cuadro 3.10: La posición ordenada ascendentemente(según probabilidad) en cada capa L_x.

Capa	Probabilidades								
L_1	$d_i \in L_1$		d_1	d_7	d_2	d_6	d_8	d_3	d_5
	$P(d_i)$		0.3	0.3	0.4	0.6	0.6	0.7	0.8
L_2	$d_i \in L_2$	d_{10}	d_{12}	d_{13}	d_{14}	d_9		d_{15}	
	$P(d_i)$	0.3	0.3	0.4	0.4	0.5		0.8	
L_3	$d_i \in L_3$	d_{17}		d_{21}	d_{18}	d_{16}		d_{22}	
	$P(d_i)$	0.2		0.3	0.5	0.7		0.7	
L_4	$d_i \in L_4$	d_{26}	d_{25}		d_{23}				
	$P(d_i)$	0.2	0.3		0.8				
L_5	$d_i \in L_5$		d_{30}	d_{31}	d_{28}	d_{27}			
	$P(d_i)$		0.2	0.6	0.7	0.9			

Cuadro 3.11: Eliminar los puntos d_4, d_{11}, d_{19}, d_{20}, d_{24} y d_{29} del cluster C_1.

Capa	Probabilidades								
L_1	$d_i \in L_1$			d_7	d_2	d_6		d_3	d_5
	$P(d_i)$			0.3	0.4	0.6		0.7	0.8
L_2	$d_i \in L_2$	d_{10}	d_{12}	d_{13}	d_{14}			d_{15}	
	$P(d_i)$	0.3	0.3	0.4	0.4			0.8	
L_3	$d_i \in L_3$			d_{21}	d_{18}			d_{22}	
	$P(d_i)$			0.3		0.7		0.7	
L_4	$d_i \in L_4$	d_{26}	d_{25}						
	$P(d_i)$	0.2	0.3						
L_5	$d_i \in L_5$		d_{30}	d_{31}	d_{28}	d_{27}			
	$P(d_i)$		0.2	0.6	0.7	0.9			

Cuadro 3.12: Eliminar los puntos d_1, d_8, d_9, d_{16}, d_{17} y d_{23} del cluster C_2.

Capa	Probabilidades							
L_1	$d_i \in L_1$				d_2		d_3	d_5
	$P(d_i)$				0.4		0.7	0.8
L_2	$d_i \in L_2$	d_{10}	d_{12}	d_{13}				
	$P(d_i)$	0.3	0.3	0.4				
L_3	$d_i \in L_3$			d_{21}	d_{18}			
	$P(d_i)$			0.3		0.7		
L_4	$d_i \in L_4$		d_{25}					
	$P(d_i)$		0.3					
L_5	$d_i \in L_5$		d_{30}	d_{31}	d_{28}	d_{27}		
	$P(d_i)$		0.2	0.6	0.7	0.9		

Cuadro 3.13: Eliminar los puntos d_2, d_{10}, d_{18}, d_2, d_{28} y d_{29} del cluster C_3.

Capa	Probabilidades				
L_1	$d_i \in L_1$				d_5
	$P(d_i)$				0.8
L_2	$d_i \in L_2$	d_{12}	d_{13}		
	$P(d_i)$	0.3	0.4		
L_3	$d_i \in L_3$		d_{21}		
	$P(d_i)$		0.3		
L_4	$d_i \in L_4$	d_{25}			
	$P(d_i)$	0.3			
L_5	$d_i \in L_5$		d_{31}	d_{27}	
	$P(d_i)$		0.6	0.9	

Cuadro 3.14: Eliminación de los puntos d_2, d_{10}, d_{18}, d_3, d_{28} y d_{30} del cluster C_4.

Capítulo 4

Evaluación de los algoritmos: El caso de los datos uniformes

4.1. Introducción

En el capítulo anterior se presentaron las metodologías propuestas para solucionar el problema del *Clustering Balanceado*. Corresponde ahora mostrar la forma de cómo se comporta sobre un conjunto de datos concreto. Para ello, se generó un conjunto de datos distribuidos uniformemente en el espacio euclidiano de diferentes tamaños: 100, 300, 500, 1000, 2000, 5000, 10000, 20000 y 40000. En la primera etapa del proceso de evaluación, se ejecutará los cuatro algoritmos propuestos (Algoritmos 5, 7, 8 y 9) con los datos de tamaño 100, 300 y 500 con las tres métricas propuestas en la sección 1.2.1 para verificar la aproximación de los resultados: *Intra-cluster-distancia*, *Indice DaviesBouldin* e *Indice Dunn*. Junto con estas métricas, también evaluaremos el tiempo de ejecución de estos algoritmos. Los valores de K aplicados en las pruebas son $2, 5, 20$ y 25 respectivamente. Por la alta complejidad algorítmica de los algoritmos 5 y 7, junto con el costoso procesamiento con la gran cantidad de datos, se procederá los algoritmos 8 y 9 con los datos de tamaño $1000, 2000, 5000, 10000, 20000$ y 40000. Una vez procesados todos los datos, se presentarán los costos del clustering balanceado generado por cada algoritmo propuesto dada las distintas configuraciones y se analizarán los siguientes puntos:

1. *Influencia de $|D|$ para cada algoritmo.* Evaluar el comportamiento de las métricas y el rendimiento de cada uno de los cuatro algoritmos a medida que se va aumentando el tamaño del conjunto de datos D dado un K fijo.

2. *Influencia de K para cada algoritmo.* Evaluar el rendimiento de cada uno de los cuatro algoritmos a medida que se va aumentando el valor de K dada una cantidad fija de los datos y una métrica.

3. *Tiempo de ejecución de cada algoritmo.* Medir el tiempo de la ejecución del algoritmo.

Los algoritmos fueron implementados sobre Java 1.5, utilizando un PC con procesador AMD Athlon(tm) - 1.33 Ghz con 256 MB de memoria RAM DDR. A continuación, presentamos los resultados generados con las distintas configuraciones para los algoritmos y luego su análisis respectivo.

4.2. Análisis del resultado de la ejecución de los cuatro algoritmos

4.2.1. Visualización de los *clusters* generados

Aunque con las métricas resultantes uno puede comparar el rendimiento de los algoritmos, ello es insuficiente para explicar el comportamiento de los algoritmos. Para ello, se recurre a la visualización de los *clusters* generados por los distintos algoritmos. En esta sección, presentamos en forma visual los resultados del clustering balanceado producidos por los cuatro algoritmos con las distintas métricas sobre los datos distribuidos en forma uniforme dado $|D| = 500$ y $K = 20$.

4.2.1.1. Intra-Cluster-Distancia

En la Fig. 4.1 se presentan las imágenes resultado: la Fig. 4.1a muestra el resultado del algoritmo *PSO-Hungaro*, la Fig. 4.1b el resultado del algoritmo *PSO-Converx-Hull*, la Fig. 4.1c el resultado del algoritmo *PSO-Absorcion-Puntos-Cercanos* y la Fig. 4.1d el resultado del algoritmo *PSO-Gale-Shapley*. Se puede observar que el algoritmo *PSO-Hungaro* logra tener el mejor costo mínimo $\approx$ 45909,73. Seguido de éste, el algoritmo *PSO-Converx-Hull* con un costo de $\approx$ 45996,94, el algoritmo *PSO-Absorcion-Puntos-Cercanos* con un costo de $\approx$ 47254,5 y finalmente, el algoritmo *PSO-Gale-Shapley* con un costo $\approx$ 70584,32 (ver cuadro 4.1).

4.2.1.2. Indice Davies-Bouldin

En la Fig. 4.2 se presentan las imágenes resultado: la Fig. 4.2a muestra el resultado del algoritmo *PSO-Hungaro*, la Fig. 4.2b el resultado del algoritmo *PSO-Converx-Hull*, la Fig. 4.2c el resultado del algoritmo *PSO-Absorcion-Puntos-Cercanos* y la Fig. 4.2d el resultado del algoritmo *PSO-Gale-Shapley*. Se puede observar que el algoritmo *PSO-Hungaro* logra tener el mejor costo mínimo: $\approx$ 8,10. Seguido de éste, el algoritmo *PSO-Converx-Hull* con un costo de $\approx$ 8,41, el algoritmo *PSO-Absorcion-Puntos-Cercanos* con un costo de $\approx$ 8,73 y finalmente, el algoritmo *PSO-Gale-Shapley* con un costo $\approx$ 15,05. Al igual que la métrica *Intra-Cluster-Distancia*, mientras que el algoritmo *PSO-Gale-Shapley* continua siendo la optimización menos eficiente (ver cuadro 4.2).

4.2.1.3. Indice Dunn

En la Fig. 4.3 se presentan las imágenes resultado: la Fig. 4.3a muestra el resultado del algoritmo *PSO-Hungaro*, la Fig. 4.3b el resultado del algoritmo *PSO-Converx-Hull*, la Fig. 4.3c el resultado del algoritmo *PSO-Absorcion-Puntos-Cercanos* y la Fig. 4.3d el resultado del algoritmo *PSO-Gale-Shapley*. Se puede observar que el algoritmo *PSO-Hungaro* logra tener el mejor Indice Dunn: $\approx 519{,}71$. Seguido de éste, el algoritmo *PSO-Converx-Hull* con Indice Dunn de $\approx 408{,}99$, el algoritmo *PSO-Absorcion-Puntos-Cercanos* con Indice Dunn de $\approx 414{,}73$ y finalmente, el algoritmo *PSO-Gale-Shapley* con un costo $\approx 321{,}91$. Al igual que la métrica *Intra-Cluster-Distancia*, mientras que el algoritmo *PSO-Gale-Shapley* continua siendo la optimización menos eficiente (ver cuadro 4.3).

4.2.1.4. Conclusión

Evidentemente, los *clusters* generados por el Algoritmo *PSO-Hungaro* son más compactos (los elementos de cada cluster están más cercanos relativamente comparado con los *clusters* generados por otros algoritmos) porque el algoritmo *Hungaro* (Algoritmo 4) es un algoritmo exacto para el problema de asignación. Por eso el resultado de la asignación no está limitado a una optimización local dados los K centroides generados por *PSO*. En cambio, los *clusters* generados por el Algorimto *PSO-Gale-Shapley* no son muy perfectos (en la Fig. 4.1b, se puede observar que el cluster 17 es casi una línea) y produce un costo menos óptimo, porque el algoritmo *Gale-Shapley* no considera el costo final para el emparejamiento sino la preferencia de cada punto a un centroide c_i, y esto de no considerar el costo final para el emparejamiento podría conducir a una optimización local.

Los *clusters* generados por los algoritmos *PSO-Convexl-Hull* y *PSO-Absorcion-Puntos-Cercanos*, visualmente están muy cercanos en cuanto a su distribución de los puntos en cada cluster. Ambos algoritmos son *Centroide-Orden-Dependiente*, es decir, la formación de los *clusters* depende principalmente de la ubicación del centroide y el orden con que cada centroide comienza a absorber los puntos más cercanos. Y así ambos algoritmos tiene el siguiente problema: la optimización local. Por ejemplo, en la Fig. 4.1d, se puede observar que el cluster 03 tiene una forma de *Luna Creciente* porque los centroides del cluster 01 y 02 han absorbido los puntos más cercanos, y el centroide de 03, también absorbe los puntos cercanos a éste mismo y hace que la forma del cluster sea "no perfecta"(no redonda). En la Fig. 4.1c, se puede observar el problema de la optimización local para el cluster 05.

Como característica del algoritmo *PSO-Convexl-Hull*, la absorción de los puntos es a partir de los puntos externos que forman el polígono que rodea al resto de los puntos internos, por eso los puntos extremos del mapa van ser incluidos en primera instancia de los *clusters* y la probabilidad de que los puntos internos del polígono forman un cluster compacto aumenta[1]. En cambio, como

[1] Lo compacto del cluster evalúa que tan buenos son los *clusters* de la salida que se redistribuye en el plano, en comparación con el conjunto de entrada general, en términos de la homogeneidad de los datos reflejados por la métrica utilizada por el sistema de agrupamiento.

característica del algoritmo *PSO-Absorcion-Punto-Cercanos*, la absorción de los puntos es a partir de los centroides entregados por *PSO*, y esto podría causar un problema: los puntos extremos podrían formarse en un grupo con una probabilidad más alta que el del algoritmo *PSO-Convexl-Hull* (por ejemplo, el cluster 05 del la Fig. 4.1c).

4.2.2. Influencia de $|D|$ para cada algoritmo

El aspecto central en el que se evalúa el efecto del algoritmo es la métrica que éste presenta como solución al problema. Las métricas durante el proceso de la exploración de solución que sirvieron como punto de referencia para evaluar la eficiencia de los algoritmos son las siguientes: *Intra-Cluster-Distancia*, *Indice Davies-Bouldin* e *Indice Dunn*. El cómo éstas evolucionan a medida que se va aumentando el tamaño del conjunto de datos, es justamente el análisis que se hará en esta sección: el comportamiento de las métricas y el rendimiento de cada uno de los cuatro algoritmos a medida que se va aumentando el tamaño del conjunto de datos D dado un K fijo.

4.2.2.1. Intra-Cluster-Distancia

En la Fig. 4.4a, para $K = 20$, se puede observar que el algoritmo *PSO-Hungaro* logra la mejor optimización; en cambio, el algoritmo *Gale-Shapley* siempre queda con la peor optimización. Cuando el valor de $|D|$ es chico, el resultado de los cuatro algoritmos generados son casi cercanos independiente del tamaño de D ($|D|$). A medida que se va aumentando el valor de $|D|$, la diferencia entre el algoritmo *Gale-Shapley* y el algoritmo *PSO-Hungaro* se va aumentando. En cambio, el comportamiento de los algoritmos *PSO-Convex-Hull* y *PSO-Absorcion-Punto-Cercanos* son muy similares con el aumento del valor de $|D|$.

4.2.2.2. Indice Davies-Bouldin

En la Fig. 4.4b, para todos los valores de K, se puede observar que el algoritmo *PSO-Hungaro* logra la mejor optimización; en cambio, el algoritmo *Gale-Shapley* siempre queda con la optimización menos eficiente. También se puede observar que la diferencia del *Indice Davies-Bouldin* del algoritmo *PSO-Hungaro* no se varía mucho a medida que se aumente el tamaño de los datos para los diferentes K. Esto es debido a dos razones: 1) los datos se van distribuyendo en forma uniforme en cada cluster, 2) la ecuación 1.6 considera la máxima proporción entre la intra-cluster-distancia y la inter-cluster-distancia. A medida que se aumente $|D|$, la inter-cluster-distancia entre cada pareja podrían no variar mucho, y la diferencia entre el promedio de la intra-cluster-distancia tampoco variarán mucho. Y así la proporción entre ambas métricas tampoco variará mucho.

4.2.2.3. Indice Dunn

Debido a que *Indice Dunn* mide que tan compacto es el cluster y si están bien separados los *clusters*, mientras mayor es el Indice Dunn, mejor es clustering. En la Fig. 4.4c, para todos los valores de K, se puede observar que el algoritmo *PSO-Hungaro* logra la mejor optimización; en cambio, el algoritmo *Gale-Shapley* siempre queda con la optimización menos eficiente. También se puede observar que la diferencia del *Indice Dunn* del algoritmo *PSO-Hungaro* no se varía mucho a medida que se aumente el tamaño de los datos para los diferentes $|D|$. En cambio, para los algoritmos *PSO-Convex-Hull* y *PSO-Absorcion-Puntos-Cercanos*, a medida que aumente $|D|$, la posiblidad de tener los *clusters* compactos se va bajando, por eso cuyo Indice Dunn también se va bajando.

4.2.2.4. Conclusión

Según los gráficos de comparación presentados anteriormente, se puede observar que el algoritmo *PSO-Hungaro* sigue liderando la mejor opitmización entre los algoritmos, debido a que éste considera en forma exacta el costo final de la mejor asignación de los puntos a cada cluster. Mientras que el algoritmo *PSO-Gale-Shapley* continua generando el resultado menos optimizado porque éste no considera el costo final a medida que asigne los puntos al mejor centroide de los *clusters*. Los algoritmos *PSO-Convex-Hull* y *PSO-Absorcion-Puntos-Cercanos* aunque presentan similar comportamiento a medida que se va aumentando $|D|$, debido a que son *Centroide-Orden-Dependiente*. Por ello, cuando $|D|$ es pequeño, la probabilidad de tener los *clusters* compactos aumenta; en caso contrario, va a disminuir. Y cuando la probabilidad de tener los *clusters* compactos es alto, el costo producido van a ser casi igual al Algoritmo *PSO-Hungaro*; en caso contrario, se va aumentando la diferencia de su propio costo con el del algoritmo *PSO-Hungaro*.

4.2.3. Influencia de K para cada algoritmo

Al igual que la sección anterior, en esta sección se evalua el comportamiento de las métricas usadas para encontrar la solución a medida que se aumente la cantidad de *clusters* K y luego se compara el rendimiento de los cuatro algoritmos propuestos.

4.2.3.1. Intra-Cluster-Distancia

En la Fig. 4.5a, para $|D| = 500$, se puede observar que el algoritmo *PSO-Hungaro* sigue logrando la mejor optimización; en cambio, el algoritmo *Gale-Shapley* continua con la optimización menos eficiente. A medida que se va aumentando el valor de $|D|$, la diferencia entre el algoritmo *Gale-Shapley* y el algoritmo *PSO-Hungaro* se va aumentando. En cambio, el comportamiento de los algoritmos *PSO-Convex-Hull* y *PSO-Absorcion-Punto-Cercanos* son muy similares con el aumento del valor de K.

4.2.3.2. Indice Davies-Bouldin

En la Fig. 4.5b, para todos los valores de K, se puede observar que el algoritmo *PSO-Hungaro* sigue logrando la mejor optimización; en cambio, el algoritmo *Gale-Shapley* continua con la optimización menos eficiente. A pesar de que los algoritmos *PSO-Convex-Hull* y *PSO-Absorcion-Puntos-Cercanos* no logran tener la mejor asignación, siguen teniendo el mismo comportamiento al igual que el algoritmo *PSO-Hungaro* a medida que se aumente el valor de K.

4.2.3.3. Indice Dunn

Debido a que *Indice Dunn* mide que tan compacto es el cluster y si están bien separados los *clusters*, mientras mayor es el Indice Dunn, mejor es clustering. En la Fig. 4.5c, para todos los valores de K, se puede observar que el algoritmo *PSO-Hungaro* logra la mejor optimización; en cambio, el algoritmo *Gale-Shapley* siempre queda con la optimización menos eficiente. Se puede observar evidentemente que, a medida que se aumente el valor de K, los cuatro algoritmos mejoran su compactitud de cluster. Esto es debido a que mientras mayor es K, menor es la suma total de intra-cluster-distancia, por ende, aumenta la compactitud de cada cluster.

4.2.3.4. Conclusión

Según los gráficos de comparación presentados anteriormente y la inter-cluster-distancia del clustering balanceado generado por los cuatro algoritmos propuestos (cuadro 4.1) para $|D| \leq 500$, se puede observar que el algoritmo *PSO-Hungaro* sigue liderando la mejor opitmización entre los algoritmos en la mayoría de los casos en mis experimentos, mientras que el algoritmo *PSO-Gale-Shapley* continua generando el resultado menos eficiente. Los algoritmos *PSO-Convex-Hull* y *PSO-Absorcion-Puntos-Cercanos* aunque presentan similar comportamiento a medida que se va aumentando K, debido a que son *Centroide-Orden-Dependiente*, siguen siendo inferior al Algoritmo *PSO-Hungaro*.

4.2.4. Tiempo de Ejecución de cada algoritmo

Considendo constante el tiempo de ejecución de las tres métricas presentadas, en la Fig. 4.6, se puede observar fácilmente que los algoritmos *PSO-Absorcion-Puntos-Cercanos* y *PSO-Convex-Hull* logra ser ejecutado en menos tiempo para generar los *clusters* en comparación con el tiempo de ejecución de los algoritmos *PSO-Hungaro* y *PSO-Gale-Shapley*. Y la diferencia del tiempo de ejecución de éstos dos algoritmos, con el aumento del valor K, tiende a presentar poca diferencia.

Se puede notar un extraño fenómeno con el aumento de $|D|$: en la Fig. 4.6a, el tiempo de la ejecución del algoritmo *PSO-Hungaro* es mayor que cualquier de los tres algoritmos, debido a su complejidad $O(n^3)$ donde $n = |D|$; sin embargo, cuando $|D|$ aumenta, por ejemplo en la Fig. 4.6b

y la Fig. 4.6c, el tiempo del algoritmo *PSO-Gale-Shapley* se supera del tiempo del algoritmo *PSO-Hungaro*. Como la complejidad algorítmica del algoritmo *PSO-Gale-Shapley* es $O(n^2)$, entonces debería ocupar menos tiempo que el algoritmo *PSO-Hungaro*. ¿Qué está pasando? Todo ello es debido a la posición de los centroides generados inicialmente. Para el algoritmo *PSO-Hungaro*, si la matriz de costo está formada como mejor caso, entonces la complejidad algoritmica es baja; en cambio, si está como el peor caso, se aumenta. En este caso, puede que la matriz de costo formado para el algoritmo *PSO-Hungaro* está más eficiente que la del algoritmo *PSO-Gale-Shapley*

4.3. Análisis del resultado de la ejecución de los algoritmos *PSO-Absorcion-Puntos-Cercanos* y *PSO-Convex-Hull*

Por la alta complejidad temporal de los algoritmos *PSO-Hungaro* y *PSO-Gale-Shapley*, se hace imposible procesar datos con tamaño grande (por ejemplo, cuando $|D| = 300$ se demora casi 2 días completos). Para ellos, los algoritmos *PSO-Absorcion-Puntos-Cercanos* y *PSO-Convex-Hull* podrán ser las alternativas para el procesamiento de datos. A continuación, se presenta los resultados de la ejecución de ambos algoritmos.

4.3.1. Visualización de los *clusters* generados

En esta sección, presentamos en forma visual los resultados del clustering balanceado producidos por los dos algoritmos con las distintas métricas con los datos distribuidos uniforme dado $|D| = 40000$ y $K = 20$.

En la Fig. 4.7 se presentan las imágenes resultado: la Fig. 4.7a muestra el resultado del algoritmo *PSO-Absorcion-Puntos-Cercanos* y la Fig. 4.7b el resultado del algoritmo *PSO-Convex-Hull*. Se puede observar que el algoritmo *PSO-Convex-Hull* logra tener el mejor costo mínimo $\approx$ 3789224. Seguido de éste, el algoritmo *PSO-Absorcion-Puntos-Cercanos* con un costo de $\approx$ 3856572 (ver cuadro 4.1).

En la Fig. 4.8 se presentan las imágenes resultado: la Fig. 4.8a muestra el resultado del algoritmo *PSO-Absorcion-Puntos-Cercanos* y la Fig. 4.8b el resultado del algoritmo *PSO-Convex-Hull*. Se puede observar que el algoritmo *PSO-Convex-Hull* logra tener el mejor costo mínimo $\approx$ 8,6. Seguido de éste, el algoritmo *PSO-Absorcion-Puntos-Cercanos* con un costo de $\approx$ 9,13 (ver cuadro 4.2).

En la Fig. 4.9 se presentan las imágenes resultado: la Fig. 4.9a muestra el resultado del algoritmo *PSO-Absorcion-Puntos-Cercanos* y la Fig. 4.9b el resultado del algoritmo *PSO-Convex-Hull*. Se puede observar que el algoritmo *PSO-Convex-Hull* logra tener el mejor Indice Dunn $\approx$ 415,74. Seguido de éste, el algoritmo *PSO-Absorcion-Puntos-Cercanos* con Indice Dunn de $\approx$ 383,41 (cuadro 4.3).

Evidentemente, los *clusters* generados por el algoritmo *PSO-Convex-Hull* son más compactos

que el Algorimto *PSO-Absorcion-Puntos-Cercanos*. Como característica del algoritmo *PSO-Convex-Hull*, la absorción de los puntos es a partir de los puntos externos que forman el polígono que rodea al resto de los puntos internos, por eso los puntos extremos del mapa van ser incluidos en primera instancia de los *clusters* y la probabilidad de que los puntos internos del polígono se hace un cluster compacto aumenta. En cambio, como característica del algoritmo *PSO-Absorcion-Punto-Cercanos*, la absorción de los puntos es a partir de los centroides entregados por *PSO*. Estos centroides podrían estar casi juntos en una zona, y esto hace que los *clusters* generados se presentan en forma de ondas. Por ejemplo, en la Fig. 4.9a, los 10 centroides generados por *PSO* podrían estar juntos en el rincón derecho inferior del mapa, y el algoritmo va absorbiendo los puntos más cercanos, que no están asginados a un cluster, a cada uno de ellos, hasta que finalmente se queda con la forma de ondas. Como ambos algoritmos son *Centroide-Orden-Dependiente*, los dos algoritmos intentan hacer que los *clusters* formados sean *compactos* (redondos). Sin embargo, esta característica en sí se hace una restricción para la generación de los *clusters* y ello podría quedarse en una optimización local, y a veces, los *clusters* no necesariamente van a tener forma redonda sino otras formas, por ejemplo, cuadrado.

4.3.2. Influencia de $|D|$ para los algoritmos *PSO-Absorcion-Puntos-Cercanos* y *PSO-Convex-Hull*

En la Fig. 4.10a, para $K = 20$, se puede observar que el algoritmo *PSO-Convex-Hull* logra la mejor optimización; en cambio, el algoritmo *PSO-Absorción-Puntos-Cercanos* continua siendo una optimización menos eficiente. Cuando el valor de $|D|$ es chico, el resultado de los cuatro algoritmos generados son casi cercanos independiente del tamaño de D ($|D|$). A medida que se va aumentando el valor de $|D|$, la diferencia entre el algoritmo *PSO-Absorcion-Puntos-Cercanos* y el algoritmo *PSO-Convex-Hull* va aumentando.

En la Fig. 4.10b, para todos los valores de K, se puede observar que el algoritmo *PSO-Convex-Hull* logra la mejor optimización; en cambio, el algoritmo *PSO-Absorción-Puntos-Cercanos* continua con la optimización menos eficiente.

Debido a que el *Indice Dunn* mide qué tan compacto es el cluster y si están bien separados los *clusters*, mientras mayor es el Indice Dunn, mejor es el clustering. Debido a la aleatoriedad del algoritmo *PSO*, el resultado de la ejecución del algorimo no necesariamente es producido con los mismos centroides finales. Por ende, los *clusters* compactos formados varían con los centroides generados. En la Fig. 4.4c, para todos los valores de $|D|$, es difcícil definir cuál algoritmo será mejor para aplicarse con esta métrica, justamente por la aleatoriedad de los centroides generados. Por eso, se puede concluir que el crecimiento del valor de $|D|$ no determinará cuál algoritmo sería óptimo en caso de contarse con la métrica de Indice Dunn.

Según los gráficos de comparación presentados anteriormente, se puede observar que el algoritmo *PSO-Convex-Hull* logra mejor optimización. La estrategia que usa el algoritmo *PSO-*

Convex-Hull: *formar capas convexa - absorber los puntos extremos primero - los puntos internos al final*, hace que la distribución de los centroides generados por *PSO* tengan una mayor cobertura en el mapa; mientras que la cobertura de los centroides entregados por *PSO-Absorción-Puntos-Cercanos* podría no tener una cobertura mayor porque el área en que se mueven los centroides converge a medida que se va disminuyendo la cantidad de iteraciones. Por eso, se puede concluir que el tamaño de D no afecta nada en la generación de los *clusters* balanceados sino en los centroides entregados.

4.3.3. Influencia de K para los algoritmos *PSO-Absorcion-Puntos-Cercanos* y *PSO-Convex-Hull*

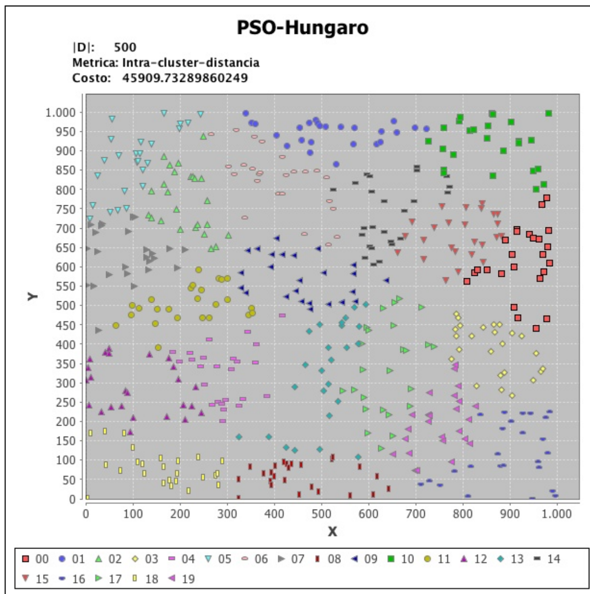
En las figuras 4.5a y 4.5b, para todos los valores de K, se puede observar que el algoritmo *PSO-Convex-Hull* sigue logrando la mejor optimización; en cambio, el algoritmo *PSO-Absorcion-Puntos-Cercanos* continua con la optimización menos eficiente. A medida que se va aumentando el valor de K, la diferencia entre las métricas el algoritmo *PSO-Absorcion-Puntos-Cercanos* y el algoritmo *PSO-Convex-Hull* va aumentando.

Nuevamente, debido a que el *Indice Dunn* depende de los centroides generados para la formación de los clustes balanceados, a pesar de que el algoritmo *PSO-Convex-Hull* tiene la ventaja de tener mayor cobertura para el movimiento de los centroides, a veces, el algoritmo *PSO-Absorción-Puntos-Cercanos* podría generar los centroides bien ubicados para la generación de los *clusters* balanceados compactos. Por eso, en la Fig. 4.4c, para todos los valores de K, es difícil definir cuál algoritmo será mejor para aplicarse con esta métrica. Se puede concluir que el crecimiento del valor de K no determinará cuál algoritmo sería óptimo en caso de contarse con la métrica de Indice Dunn.

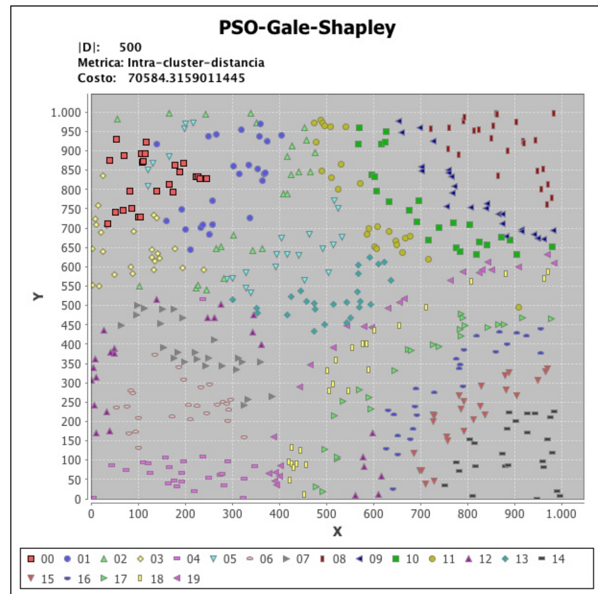
En el cuadro 4.1, se presentan la medida de inter-cluster-distancia del clustering generado por los cuatro algoritmos propuestos. En mis experimentos, se puede observar fácilmente que para $K < 20$, el algoritmo *PSO-Absorcion-Puntos-Cercanos* es el que obtiene ventaja sobre el algoritmo *PSO-Convex-Hull*, mientras que para $K \geq 20$, el algoritmo *PSO-Convex-Hull* logra el resultado más bueno.

4.3.4. Tiempo de Ejecución de los algoritmos *PSO-Absorcion-Puntos-Cercanos* y *PSO-Convex-Hull*

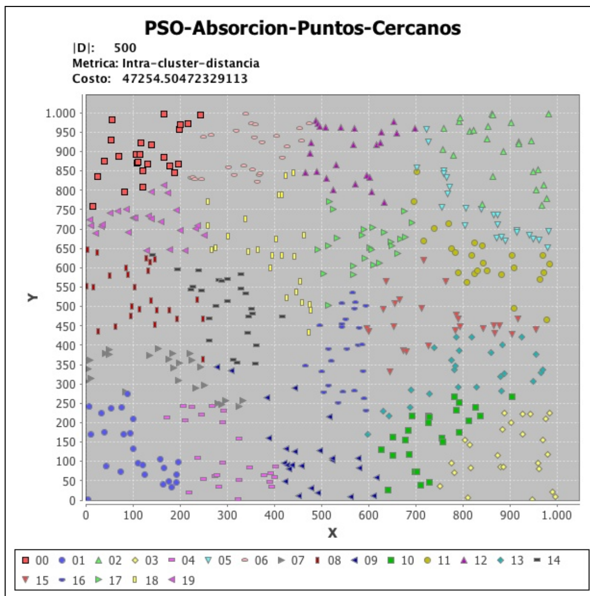
Considerando constante el tiempo de ejecución de las tres métricas presentadas. En las figuras del 4.12, se puede observar fácilmente que los algoritmos *PSO-Absorcion-Puntos-Cercanos* y *PSO-Convex-Hull* tienen el mismo comportamiento a medida que se va aumentando el valor de K. Evidentemente, el algoritmo *PSO-Convex-Hull* ocupa mayor tiempo que el algoritmo *PSO-Absorción-Puntos-Cercanos* por el tiempo constante para la formación de las capas convexas.



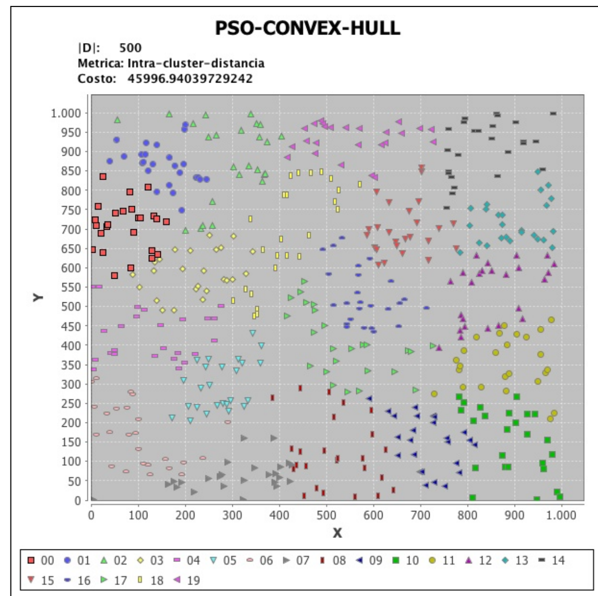
(a) PSO-Hungaro, $K = 20$.



(b) PSO-Gale-Shapley, $K = 20$.

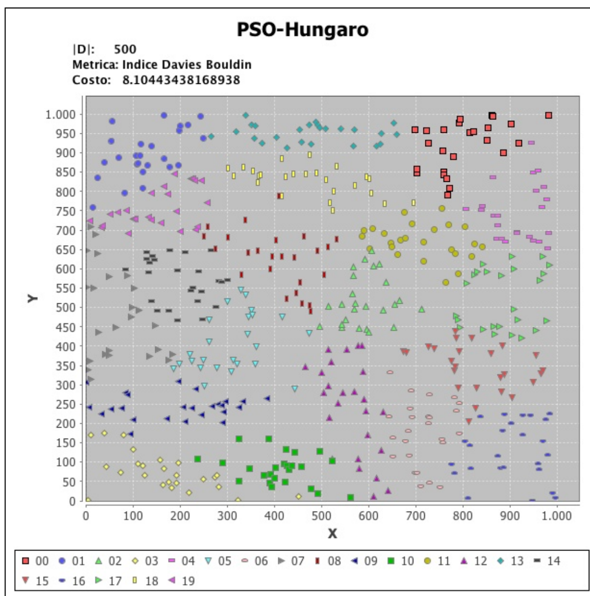


(c) PSO-Absorcion-Puntos-Cercanos, $K = 20$.

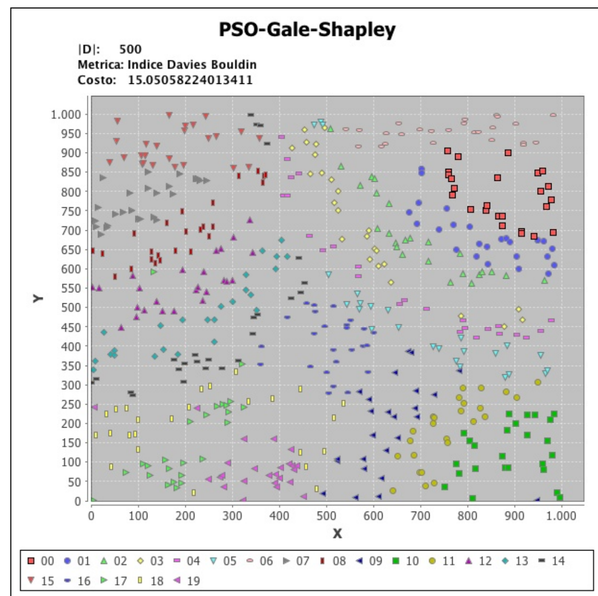


(d) PSO-Convex-Hull, $K = 20$.

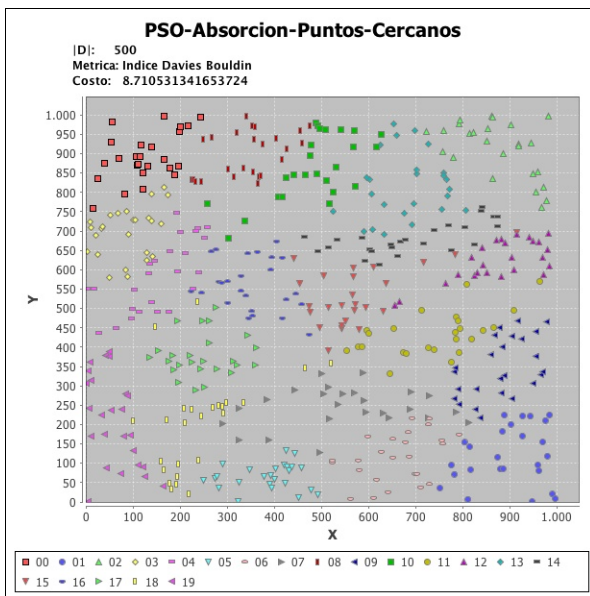
Figura 4.1: El resultado de la ejecución de los cuatro algoritmos dado $|D| = 500$ y la métrica *Intra-cluster-distancia* con la distribución uniforme de los datos en el plano.



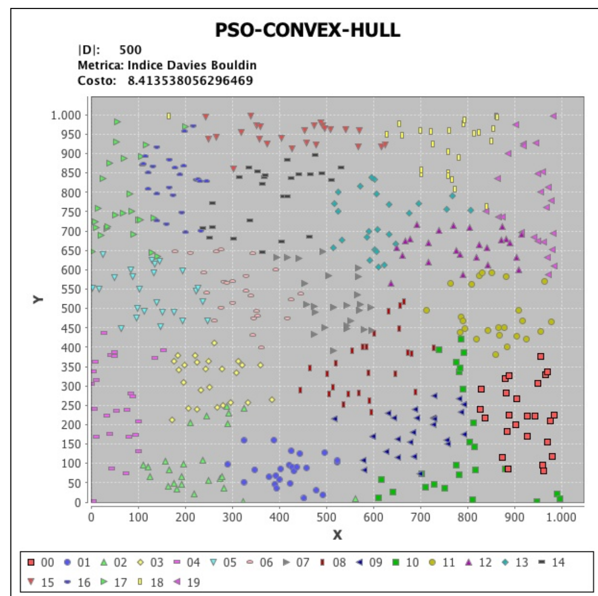
(a) PSO-Hungaro, $K = 20$.



(b) PSO-Gale-Shapley, $K = 20$.

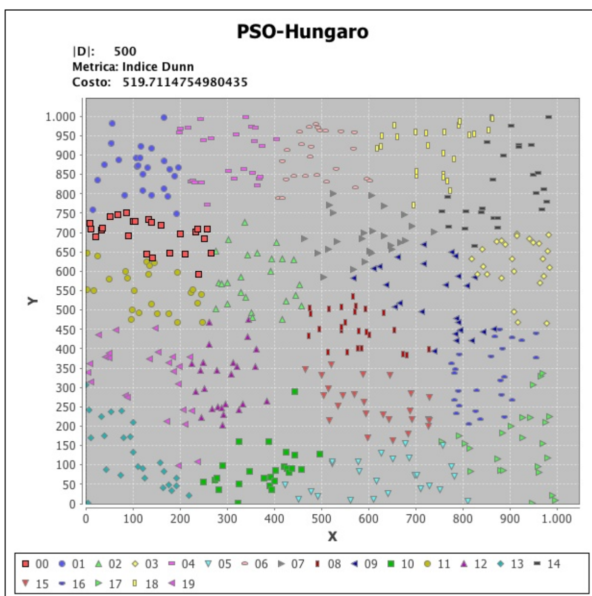


(c) PSO-Absorcion-Puntos-Cercanos, $K = 20$.

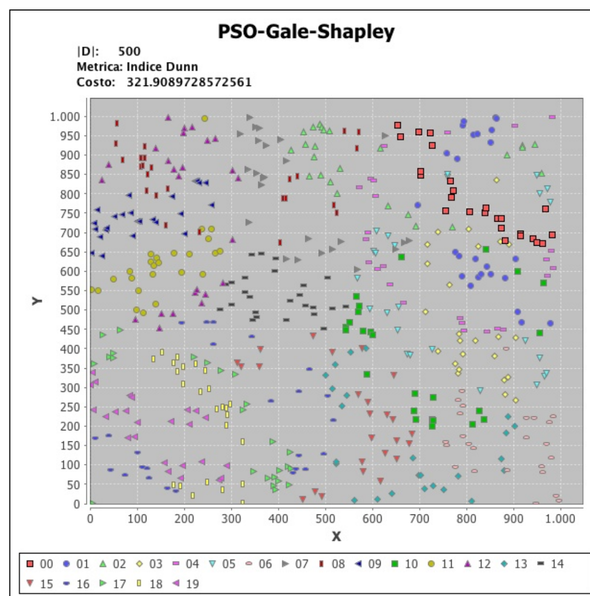


(d) PSO-Convex-Hull, $K = 20$.

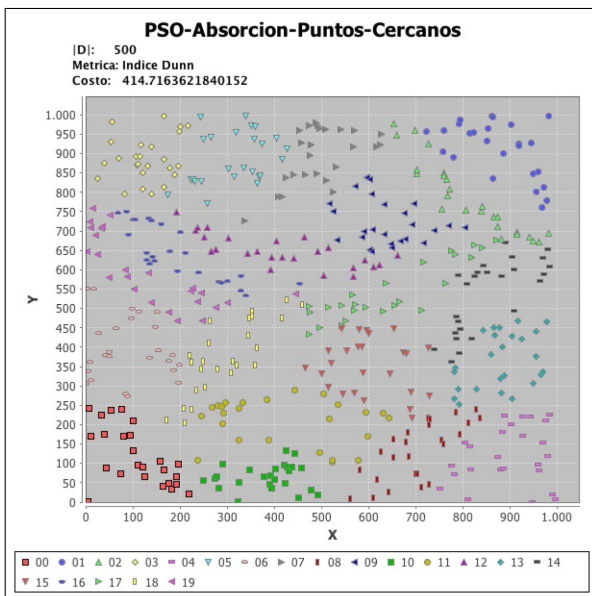
Figura 4.2: El resultado de la ejecución de los cuatro algoritmos dado $|D| = 500$ y la métrica *Indice Davies-Bouldin* con la distribución uniforme de los datos en el plano.



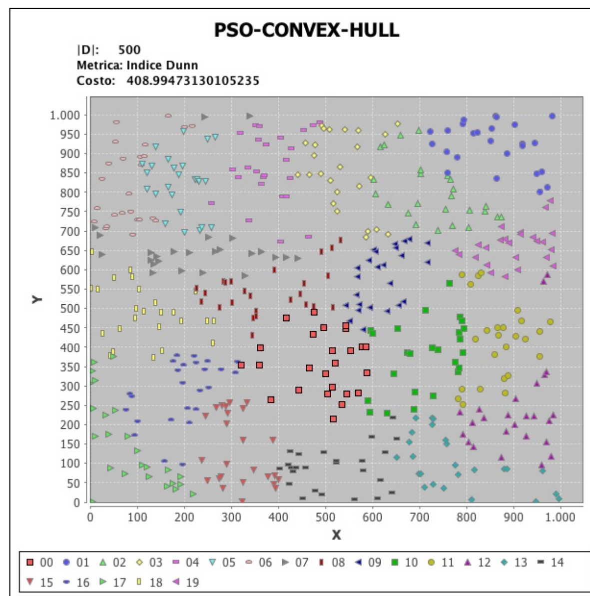
(a) PSO-Hungaro, $K = 20$.



(b) PSO-Gale-Shapley, $K = 20$.

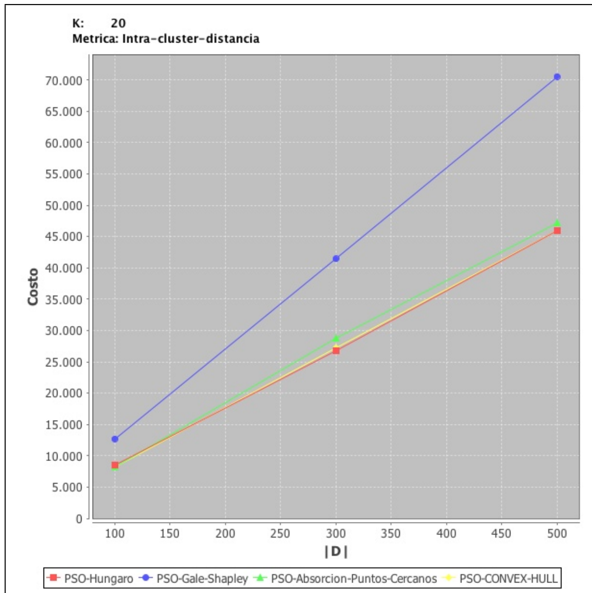


(c) PSO-Absorcion-Puntos-Cercanos, $K = 20$.

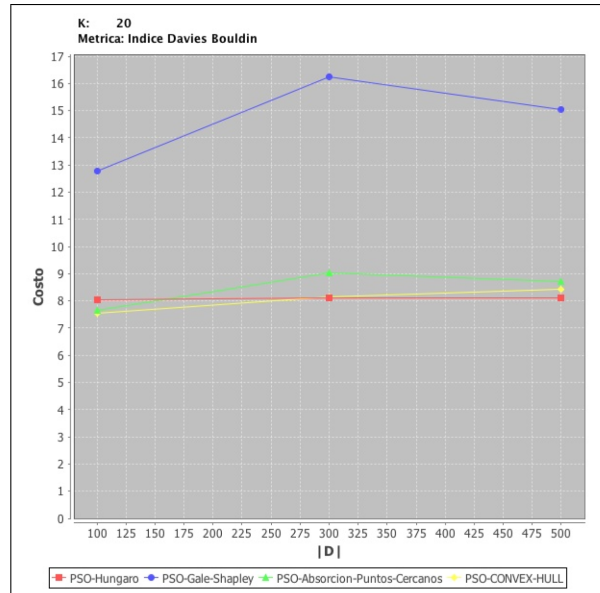


(d) PSO-Convex-Hull, $K = 20$.

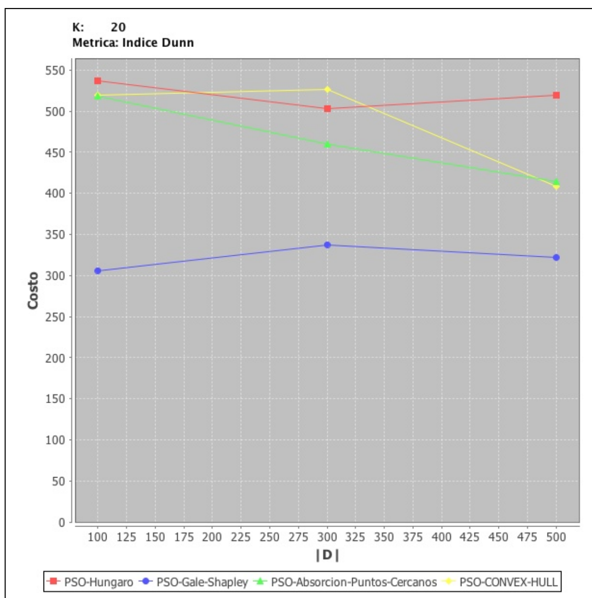
Figura 4.3: El resultado de la ejecución de los cuatro algoritmos dado $|D| = 500$ y la métrica *Indice Dunn* con la distribución uniforme de los datos en el plano.



(a) La métrica *Intra-Cluster-Distancia*.

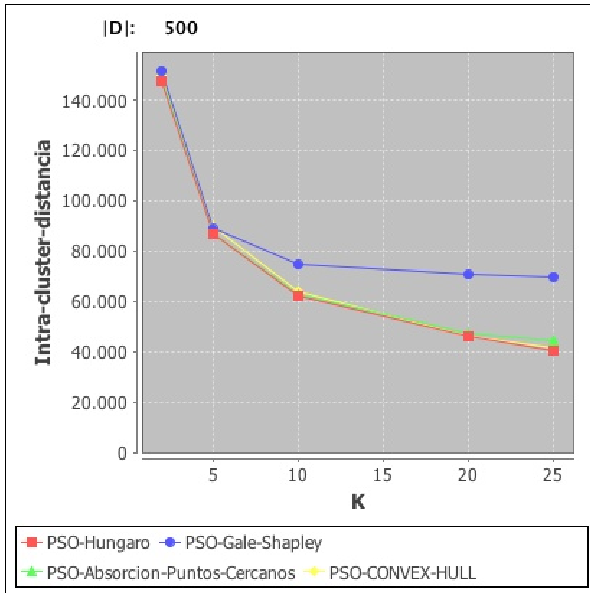


(b) La métrica *Indice Davies-Bouldin*.

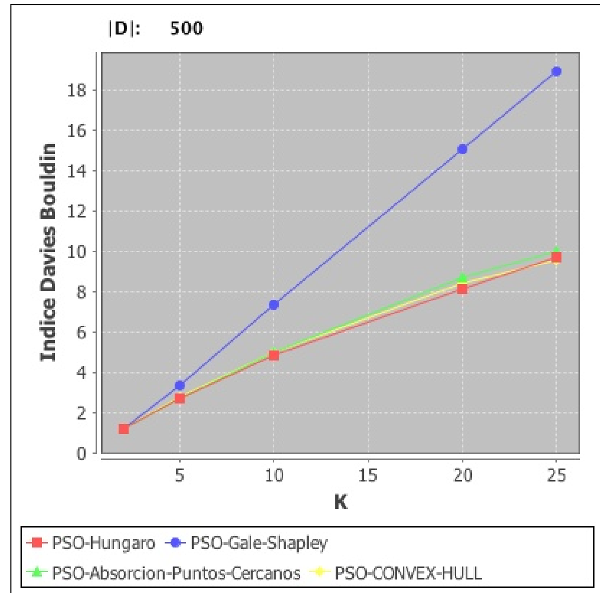


(c) La métrica *Indice Dunn*.

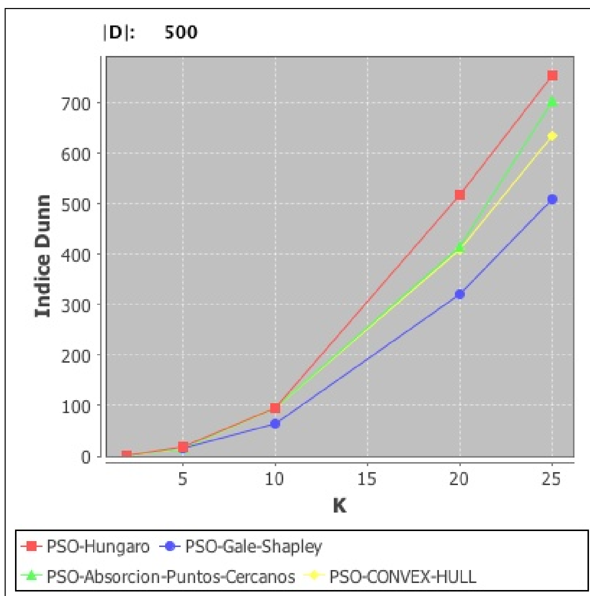
Figura 4.4: Influencia de $|D|$ para cada algoritmo dada las distintas métricas.



(a) La métrica *Intra-Cluster-Distancia*.

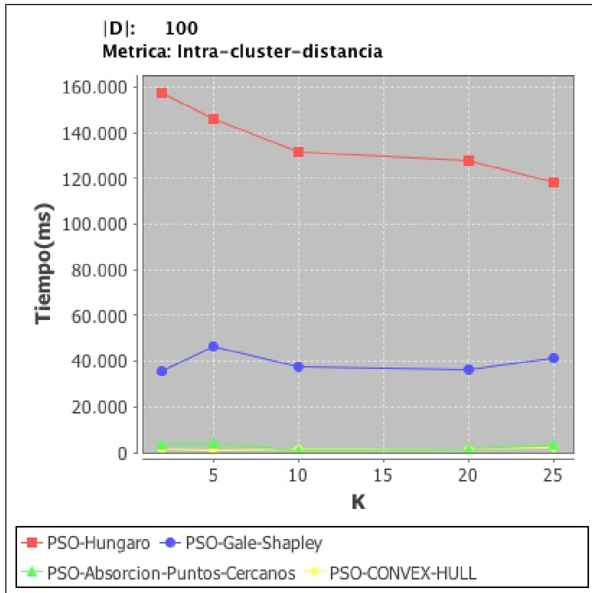


(b) La métrica *Indice Davies-Bouldin*.

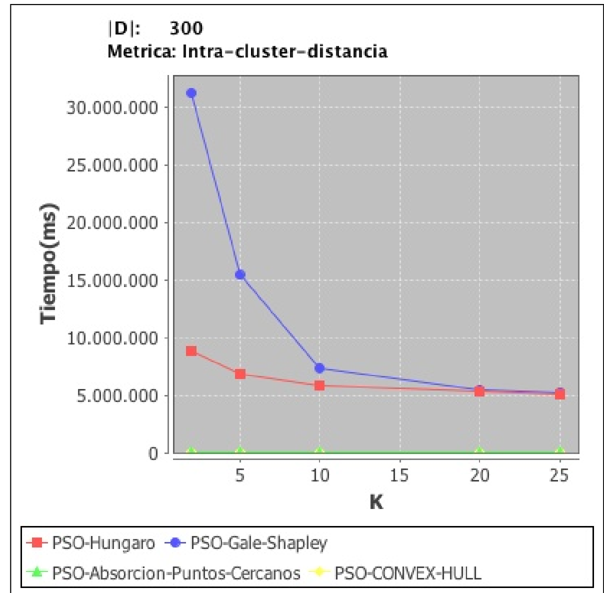


(c) La métrica *Indice Dunn*.

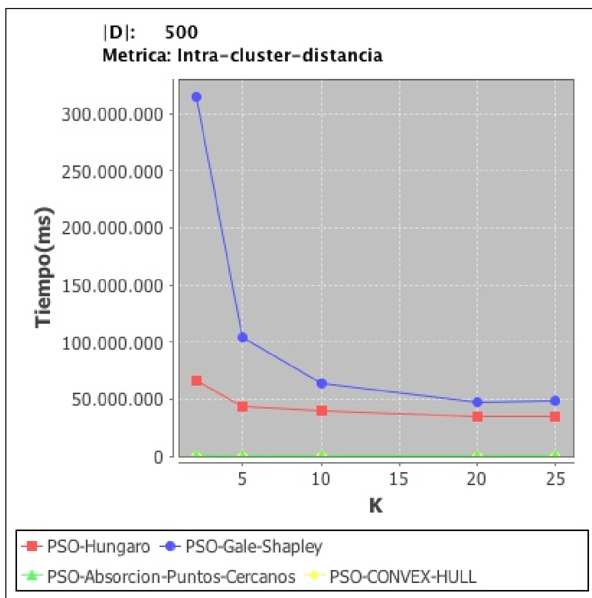
Figura 4.5: Influencia de K para cada algoritmo dada las distintas métricas.



(a) $|D| = 100$.

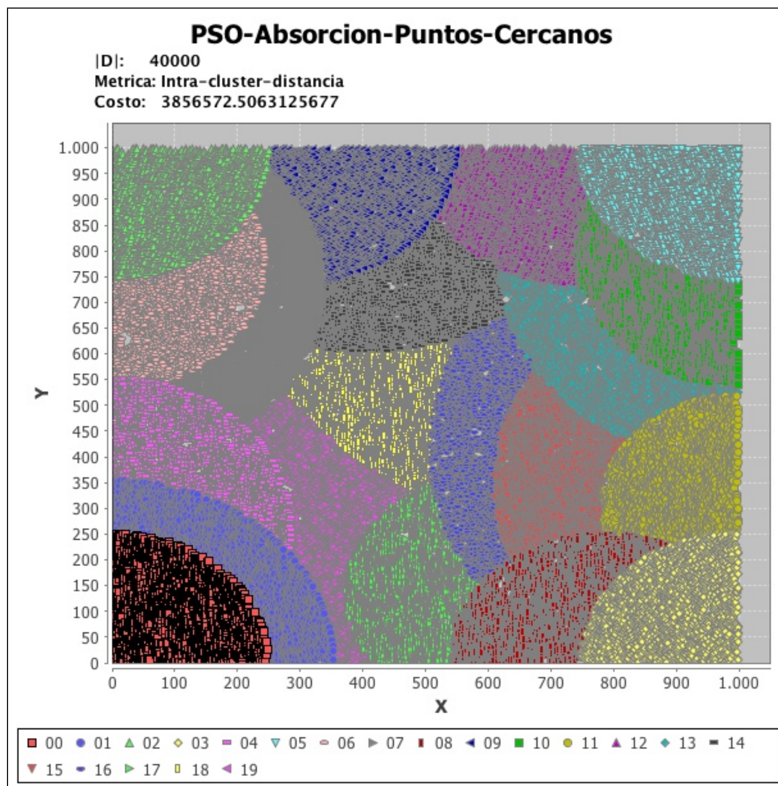


(b) $|D| = 300$.

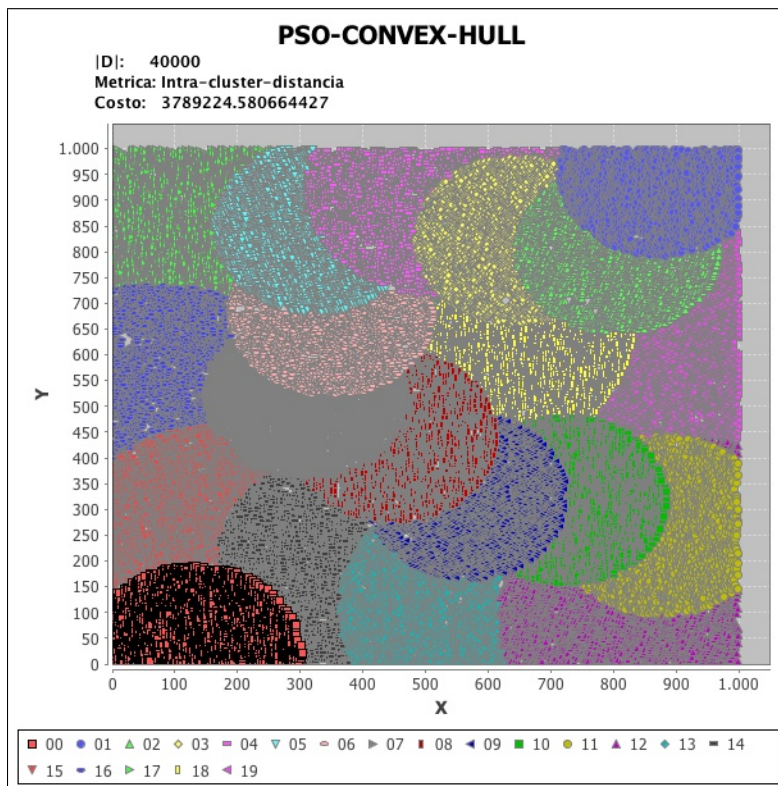


(c) $|D| = 500$.

Figura 4.6: Influencia de $|D|$ para cada algoritmo.

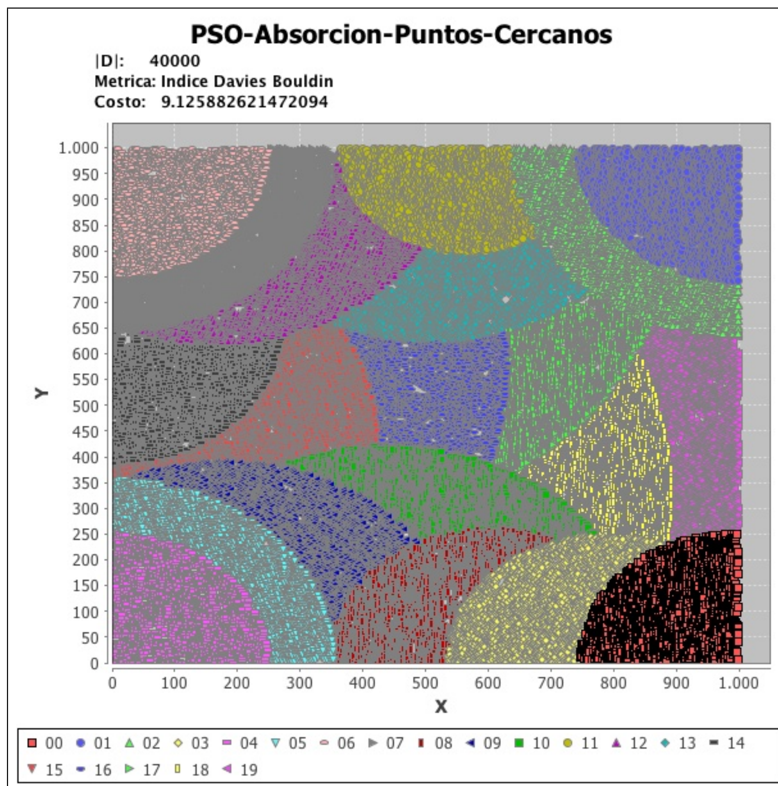


(a) PSO-Absorcion-Puntos-Cercanos, $K = 20$.

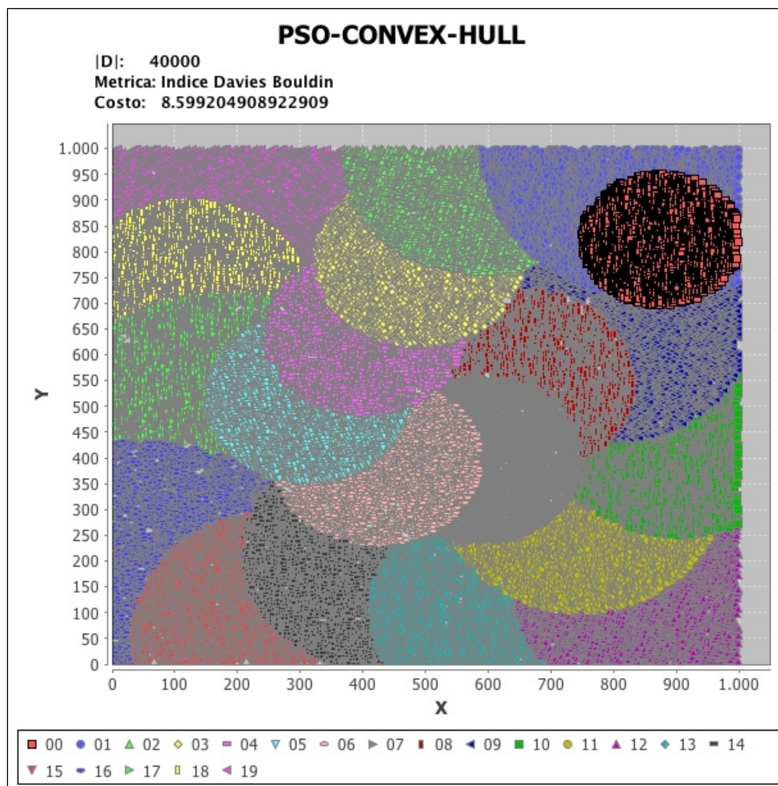


(b) PSO-Convex-Hull, $K = 20$.

Figura 4.7: El resultado de la ejecución de los cuatro algoritmos dado $|D| = 40000$ y la métrica *Intra-cluster-distancia* con la distribución uniforme de los datos en el plano.

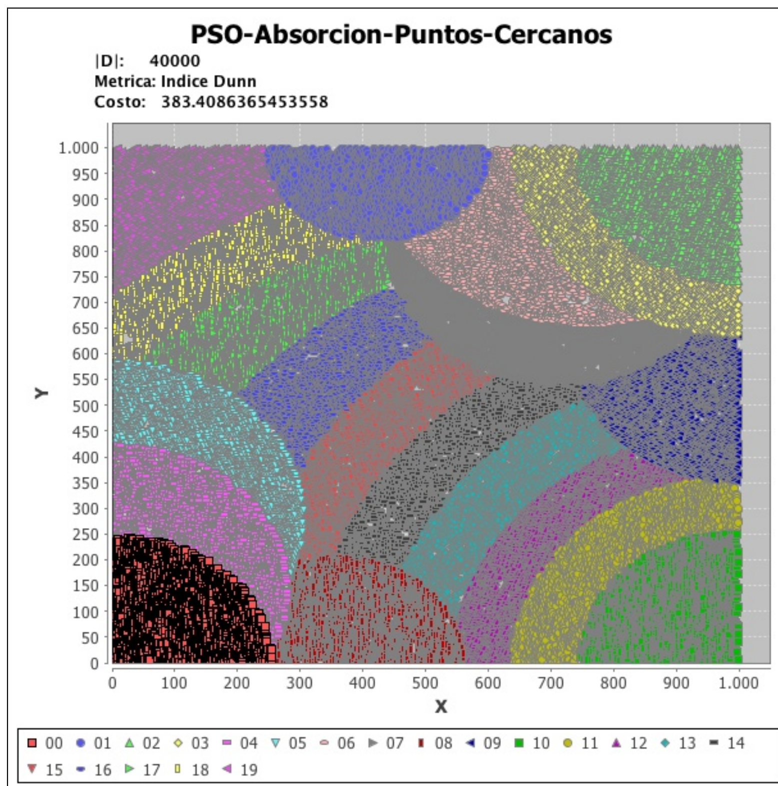


(a) PSO-Absorcion-Puntos-Cercanos, $K = 20$.

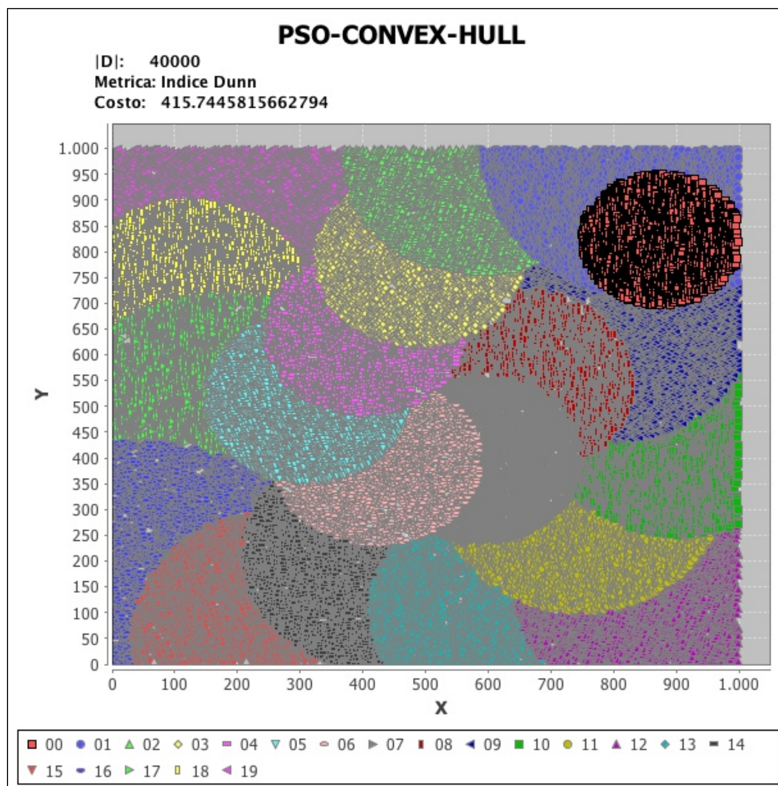


(b) PSO-Convex-Hull, $K = 20$.

Figura 4.8: El resultado de la ejecución de los cuatro algoritmos dado $|D| = 40000$ y la métrica *Indice Davies-Bouldin* con la distribución uniforme de los datos en el plano.

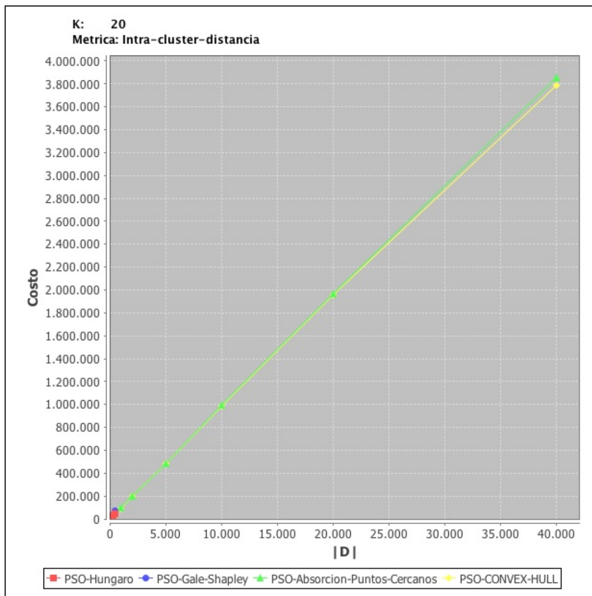


(a) PSO-Absorcion-Puntos-Cercanos, $K = 20$.

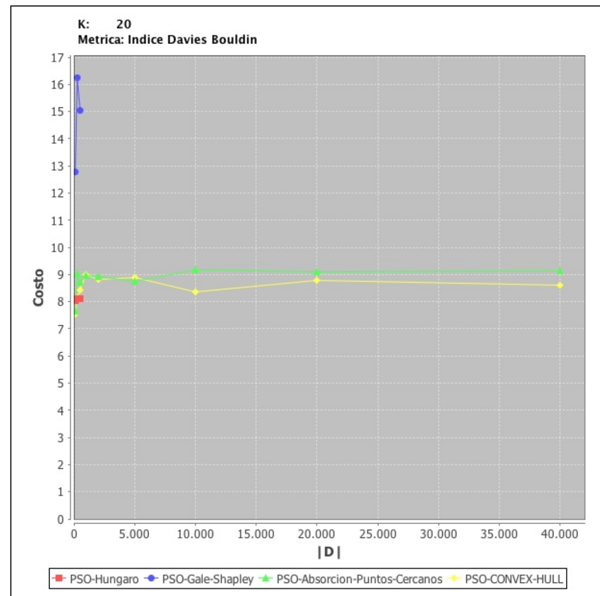


(b) PSO-Convex-Hull, $K = 20$.

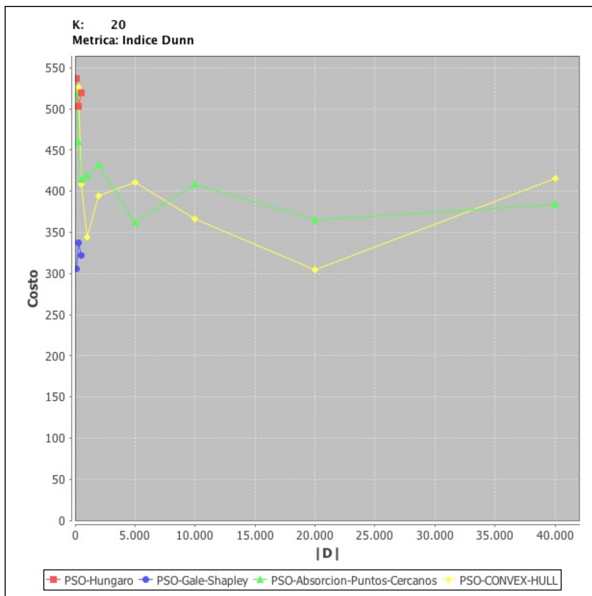
Figura 4.9: El resultado de la ejecución de los cuatro algoritmos dado $|D| = 40000$ y la métrica *Indice Dunn* con la distribución uniforme de los datos en el plano.



(a) La métrica *Intra-Cluster-Distancia*.

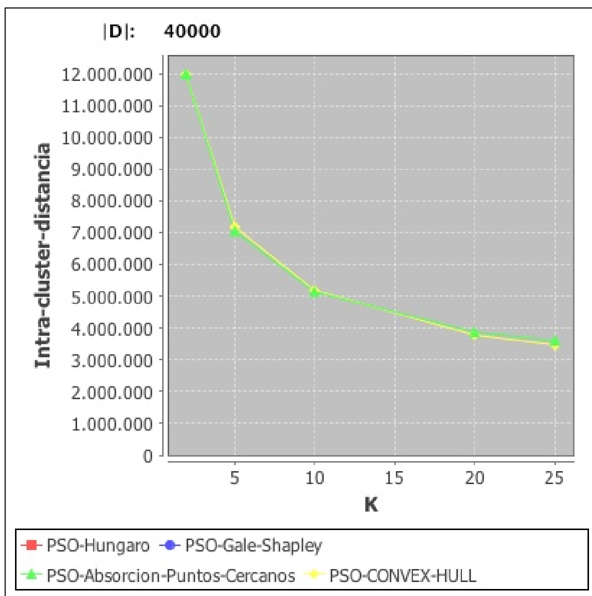


(b) La métrica *Indice Davies-Bouldin*.

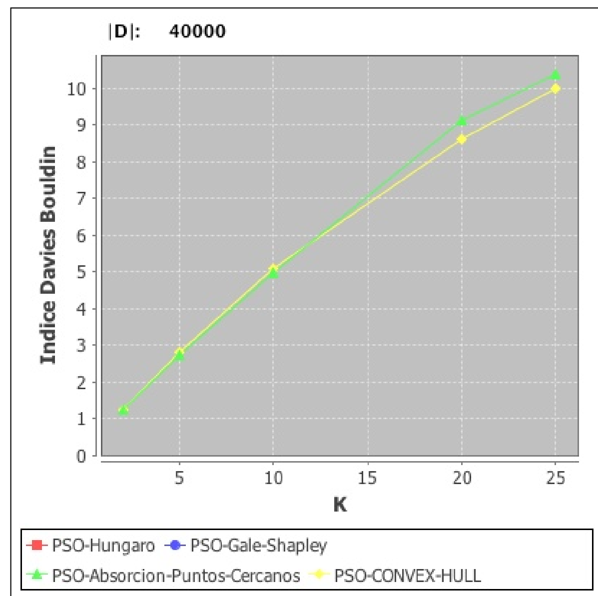


(c) La métrica *Indice Dunn*.

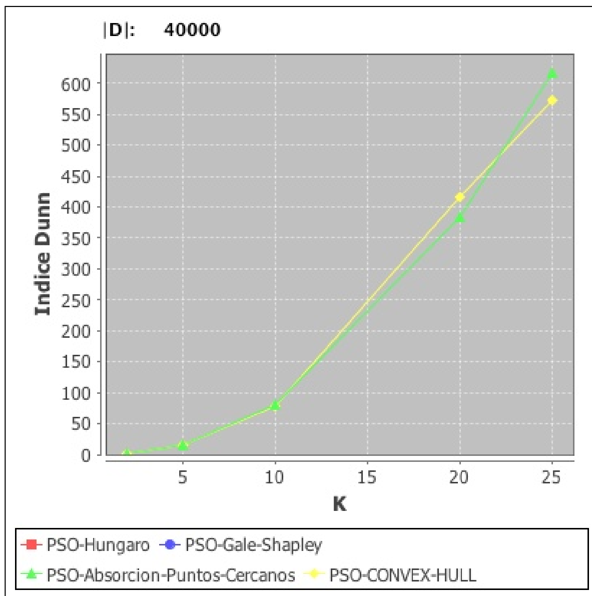
Figura 4.10: Influencia de $|D|$ para los algoritmos *PSO-Absorción-Puntos-Cercanos* y *PSO-Convex-Hull* dada las distintas métricas.



(a) La métrica *Intra-Cluster-Distancia*.

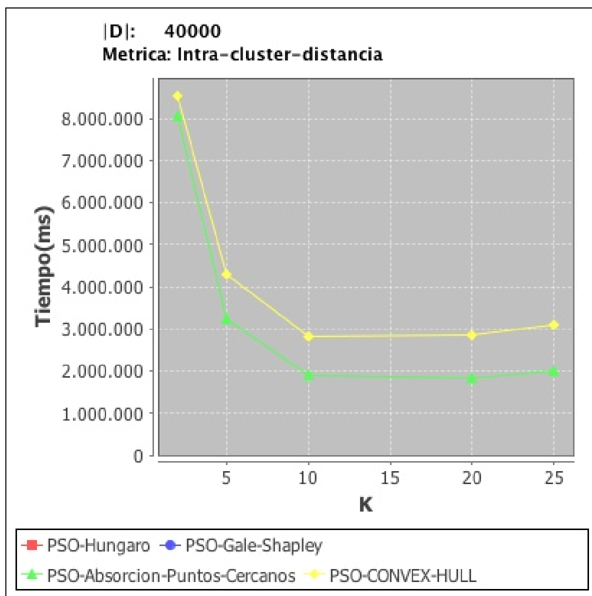


(b) La métrica *Indice Davies-Bouldin*.

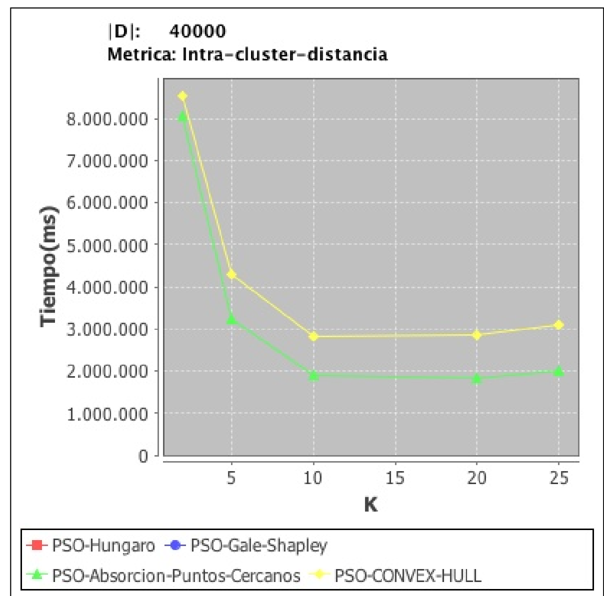


(c) La métrica *Indice Dunn*.

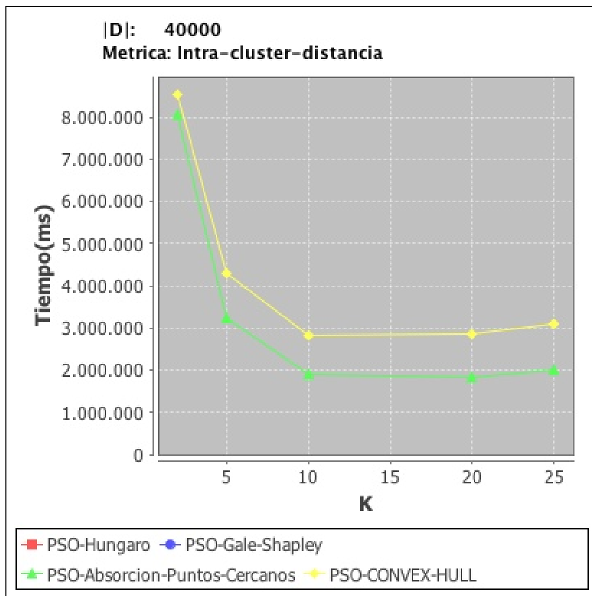
Figura 4.11: Influencia de K para los algoritmos *PSO-Absorción-Puntos-Cercanos* y *PSO-Convex-Hull* dada las distintas métricas.



(a) $|D| = 100$.



(b) $|D| = 300$.



(c) $|D| = 500$.

Figura 4.12: El tiempo de ejecución de los algoritmos *PSO-Absorción-Puntos-Cercanos* y *PSO-Convex-Hull*.

Algoritmo	$\|D\|$	k	d	k	d	k	d	k	d	k	d
PSO-Hungaro	100	2	27511	5	16536	10	11116	20	8590	25	7507
PSO-Gale-Shapley	100	2	28443	5	18258	10	14707	20	12681	25	12216
PSO-Absorcion-Puntos-Cercanos	100	2	27806	5	16776	10	12419	20	8188	25	7387
PSO-CONVEX-HULL	100	2	27824	5	16954	10	12384	20	8151	25	7035
PSO-Hungaro	300	2	87294	5	51294	10	36715	20	26768	25	24213
PSO-Gale-Shapley	300	2	89197	5	52096	10	45929	20	41437	25	39674
PSO-Absorcion-Puntos-Cercanos	300	2	88030	5	51378	10	37672	20	28809	25	26311
PSO-CONVEX-HULL	300	2	88805	5	52534	10	39298	20	27297	25	23948
PSO-Hungaro	500	2	147934	5	86719	10	62449	20	45909	25	40697
PSO-Gale-Shapley	500	2	151820	5	88964	10	74663	20	70584	25	69842
PSO-Absorcion-Puntos-Cercanos	500	2	149506	5	86846	10	62732	20	47254	25	44554
PSO-CONVEX-HULL	500	2	149672	5	89506	10	64160	20	45996	25	41742
PSO-Absorcion-Puntos-Cercanos	1000	2	289943	5	172503	10	124018	20	100328	25	89282
PSO-CONVEX-HULL	1000	2	290623	5	175593	10	127807	20	91021	25	83717
PSO-Absorcion-Puntos-Cercanos	2000	2	598822	5	353485	10	253664	20	199659	25	177515
PSO-CONVEX-HULL	2000	2	599387	5	365210	10	265164	20	194334	25	173848
PSO-Absorcion-Puntos-Cercanos	5000	2	1510250	5	886136	10	634158	20	481687	25	452252
PSO-CONVEX-HULL	5000	2	1513307	5	912911	10	655236	20	481959	25	431289
PSO-Absorcion-Puntos-Cercanos	10000	2	3011860	5	1766425	10	1273678	20	990674	25	900782
PSO-CONVEX-HULL	10000	2	3017112	5	1832567	10	1323899	20	981353	25	883412
PSO-Absorcion-Puntos-Cercanos	20000	2	6016397	5	3522712	10	2577809	20	1965648	25	1840354
PSO-CONVEX-HULL	20000	2	6030715	5	3634459	10	2634496	20	1954027	25	1743216
PSO-Absorcion-Puntos-Cercanos	40000	2	11990040	5	7033914	10	5161467	20	3856572	25	3609158
PSO-CONVEX-HULL	40000	2	12007876	5	7196957	10	5179822	20	3789224	25	3455615

Cuadro 4.1: La inter-cluster-distancia d generada por los cuatro algoritmos propuestos con diferentes valores de $|D|$ y k.

Algoritmo	$\lvert D\rvert$	k	d	k	d	k	d	k	d	k	d
PSO-Hungaro	100	2	1,1369	5	2,6976	10	4,9803	20	8,0377	25	8,9904
PSO-Gale-Shapley	100	2	1,1457	5	3,2156	10	7,1257	20	12,7792	25	16,0193
PSO-Absorcion-Puntos-Cercanos	100	2	1,1759	5	2,917	10	4,8565	20	7,6533	25	9,9418
PSO-CONVEX-HULL	100	2	1,1969	5	2,8417	10	5,547	20	7,5395	25	8,4198
PSO-Hungaro	300	2	1,1323	5	2,6631	10	4,8085	20	8,1286	25	9,8329
PSO-Gale-Shapley	300	2	1,1804	5	2,8311	10	6,6959	20	16,2611	25	18,4876
PSO-Absorcion-Puntos-Cercanos	300	2	1,1567	5	2,6729	10	4,8856	20	9,0412	25	10,1028
PSO-CONVEX-HULL	300	2	1,1922	5	2,7469	10	5,1363	20	8,1369	25	9,4765
PSO-Hungaro	500	2	1,1638	5	2,6627	10	4,8313	20	8,1044	25	9,6531
PSO-Gale-Shapley	500	2	1,1893	5	3,3335	10	7,3306	20	15,0506	25	18,915
PSO-Absorcion-Puntos-Cercanos	500	2	1,1971	5	2,6725	10	4,9659	20	8,7105	25	9,9586
PSO-CONVEX-HULL	500	2	1,2015	5	2,7345	10	4,9342	20	8,4135	25	9,5574
PSO-Absorcion-Puntos-Cercanos	1000	2	1,1776	5	2,7137	10	5,238	20	8,9689	25	10,2424
PSO-CONVEX-HULL	1000	2	1,1859	5	2,8068	10	5,2097	20	8,9977	25	10,0911
PSO-Absorcion-Puntos-Cercanos	2000	2	1,2114	5	2,725	10	4,8903	20	8,9386	25	11,0449
PSO-CONVEX-HULL	2000	2	1,2138	5	2,872	10	5,0377	20	8,8146	25	10,185
PSO-Absorcion-Puntos-Cercanos	5000	2	1,231	5	2,7404	10	4,8794	20	8,7574	25	10,4968
PSO-CONVEX-HULL	5000	2	1,2347	5	2,8397	10	5,0857	20	8,8772	25	10,2068
PSO-Absorcion-Puntos-Cercanos	10000	2	1,2237	5	2,7178	10	4,89	20	9,1755	25	11,4055
PSO-CONVEX-HULL	10000	2	1,226	5	2,8642	10	5,2142	20	8,3454	25	10,0258
PSO-Absorcion-Puntos-Cercanos	20000	2	1,2304	5	2,7249	10	4,8538	20	9,0852	25	10,7691
PSO-CONVEX-HULL	20000	2	1,2334	5	2,8641	10	4,9026	20	8,7934	25	10,0576
PSO-Absorcion-Puntos-Cercanos	40000	2	1,2297	5	2,7383	10	4,9759	20	9,1259	25	10,3867
PSO-CONVEX-HULL	40000	2	1,2316	5	2,8125	10	5,086	20	8,5992	25	10,0013

Cuadro 4.2: El índice Davies Doublin d generado por los cuatro algoritmos propuestos con diferentes valores de $|D|$ y k.

Algoritmo	$\|D\|$	k	d	k	d	k	d	k	d	k	d
PSO-Hungaro	100	2	0,9831	5	18,9379	10	102,4791	20	537,5829	25	843,9296
PSO-Gale-Shapley	100	2	0,7888	5	15,0629	10	76,9214	20	306,1154	25	630,4065
PSO-Absorcion-Puntos-Cercanos	100	2	0,9027	5	17,2725	10	103,0303	20	518,5239	25	966,8067
PSO-CONVEX-HULL	100	2	0,882	5	16,8649	10	93,8381	20	519,773	25	823,0163
PSO-Hungaro	300	2	0,9935	5	17,7482	10	107,4434	20	503,4672	25	897,6327
PSO-Gale-Shapley	300	2	0,7344	5	14,6951	10	62,5761	20	337,612	25	543,1229
PSO-Absorcion-Puntos-Cercanos	300	2	0,8508	5	17,6172	10	91,2352	20	460,3033	25	812,701
PSO-CONVEX-HULL	300	2	0,8523	5	16,7063	10	82,3947	20	526,5578	25	776,2924
PSO-Hungaro	500	2	0,9321	5	17,6375	10	96,862	20	519,7115	25	756,2115
PSO-Gale-Shapley	500	2	0,5984	5	14,8935	10	64,3349	20	321,909	25	509,5046
PSO-Absorcion-Puntos-Cercanos	500	2	0,8909	5	16,8856	10	96,4509	20	414,7164	25	704,9964
PSO-CONVEX-HULL	500	2	0,8683	5	15,7327	10	95,9915	20	408,9947	25	634,5645
PSO-Absorcion-Puntos-Cercanos	1000	2	0,8435	5	16,4998	10	93,1437	20	419,4886	25	688,0965
PSO-CONVEX-HULL	1000	2	0,824	5	15,1966	10	83,2197	20	343,7763	25	609,1378
PSO-Absorcion-Puntos-Cercanos	2000	2	0,8299	5	15,6227	10	83,1238	20	432,3303	25	618,7941
PSO-CONVEX-HULL	2000	2	0,8173	5	15,0065	10	83,9472	20	394,579	25	642,5458
PSO-Absorcion-Puntos-Cercanos	5000	2	0,8198	5	15,9221	10	84,6743	20	361,3056	25	704,2156
PSO-CONVEX-HULL	5000	2	0,7989	5	14,4883	10	77,9345	20	410,2509	25	598,6523
PSO-Absorcion-Puntos-Cercanos	10000	2	0,8172	5	15,4499	10	81,7012	20	408,0883	25	584,1919
PSO-CONVEX-HULL	10000	2	0,8019	5	14,7981	10	73,4909	20	366,3471	25	563,9642
PSO-Absorcion-Puntos-Cercanos	20000	2	0,8203	5	15,3555	10	80,7399	20	364,7807	25	579,5771
PSO-CONVEX-HULL	20000	2	0,8	5	14,263	10	82,9655	20	304,9746	25	618,1355
PSO-Absorcion-Puntos-Cercanos	40000	2	0,801	5	15,0686	10	80,7413	20	383,4086	25	617,6549
PSO-CONVEX-HULL	40000	2	0,7916	5	14,1795	10	77,0055	20	415,7446	25	572,9247

Cuadro 4.3: El índice Dunn d generado por los cuatro algoritmos propuestos con diferentes valores de $|D|$ y k.

Capítulo 5

Evaluación de los algoritmos: El caso de los datos no uniformes

5.1. Introducción

En el capítulo anterior se ha expuesto las metodologías propuestas para solucionar el problema del *Clustering Balanceado*. Corresponde ahora mostrar la forma de cómo se comporta sobre un conjunto de datos no uniformes. Para los ejemplos presentados en este capítulo, se generaron dos conjuntos de datos D' y D'' que corresponden datos distribuidos no uniformemente en el espacio euclidiano. Cada conjunto de datos está compuesto de diferente tamaño: 100, 300, 500, 1000, 2000, 5000, 10000, 20000 y 40000. En la primera etapa, presentamos en forma visual los resultados del clustering balanceado producidos por los cuatro algoritmos con las distintas métricas sobre los datos distribuidos en forma uniforme dado $|D| = 500$ y $K = 20$ para cada uno de los conjuntos. Presentamos también los costos del clustering balanceado generado por estos cuatro algoritmos propuestos para las distintas configuraciones. Finalmente, en la sección 5.3 se presentan las conclusiones del presente capítulo.

5.2. Visualización de los *clusters* generados

En esta sección, presentamos en forma visual los resultados del clustering balanceado producidos por los cuatro algoritmos con las distintas métricas sobre los dos conjuntos de datos distribuidos en forma no uniforme dado $|D| = 500$ y $K = 20$.

5.2.1. Intra-Cluster-Distancia

En la Fig. 5.1 se presentan las imágenes resultado del clustering para D': la Fig. 5.1a muestra el resultado del algoritmo *PSO-Hungaro*, la Fig. 5.1b el resultado del algoritmo *PSO-Gale-Shapley*,

la Fig. 5.1c el resultado del algoritmo *PSO-Absorcion-Puntos-Cercanos* y la Fig. 5.1d el resultado del algoritmo *PSO-Converx-Hull*. Se puede observar que el algoritmo *PSO-Converx-Hull* logra tener el mejor costo mínimo $\approx$ 1,22. Seguido de éste, el algoritmo *PSO-Absorcion-Puntos-Cercanos* con un costo de $\approx$ 1,39, el algoritmo *PSO-Gale-Shapley* con un costo $\approx$ 1,44 y finalmente, el algoritmo *PSO-Hungaro* con un costo $\approx$ 1,68 (ver cuadro 5.1).

En la Fig. 5.2 se presentan las imágenes resultado del clustering para D'': la Fig. 5.2a muestra el resultado del algoritmo *PSO-Hungaro*, la Fig. 5.2b muestra el resultado del algoritmo *PSO-Gale-Shapley*, la Fig. 5.2c muestra el resultado del algoritmo *PSO-Absorcion-Puntos-Cercanos* y la Fig. 5.2d muestra el resultado del algoritmo *PSO-Converx-Hull*. Se puede observar que el algoritmo *PSO-Converx-Hull* logra tener el mejor costo mínimo $\approx$ 1,95. Seguido de éste, el algoritmo *PSO-Hungaro* con un costo $\approx$ 2,18, el algoritmo *PSO-Absorcion-Puntos-Cercanos* con un costo de $\approx$ 2,18, y finalmente, el algoritmo *PSO-Gale-Shapley* con un costo $\approx$ 2,27 (ver cuadro 5.1).

5.2.2. Indice Davies-Bouldin

En la Fig. 5.3 se presentan las siguientes imágenes como resultado del clustering para D': la Fig. 5.3a muestra el resultado del algoritmo *PSO-Hungaro*, la Fig. 5.3b muestra el resultado del algoritmo *PSO-Gale-Shapley*, la Fig. 5.3c muestra el resultado del algoritmo *PSO-Absorcion-Puntos-Cercanos* y la Fig. 5.3d muestra el resultado del algoritmo *PSO-Converx-Hull*. Se puede observar que el algoritmo *PSO-Converx-Hull* logra nuevamente tener el mejor costo mínimo $\approx$ 7,65. Seguido de éste, el algoritmo *PSO-Absorcion-Puntos-Cercanos* con un costo de $\approx$ 9,66, el algoritmo *PSO-Gale-Shapley* con un costo $\approx$ 9,94 y finalmente, el algoritmo *PSO-Hungaro* con un costo $\approx$ 13,03 (ver cuadro 5.2).

En la Fig. 5.4 se presentan las siguientes imágenes como resultado del clustering para D'': la Fig. 5.4a muestra el resultado del algoritmo *PSO-Hungaro*, la Fig. 5.4b el resultado del algoritmo *PSO-Gale-Shapley*, la Fig. 5.4c el resultado del algoritmo *PSO-Absorcion-Puntos-Cercanos* y la Fig. 5.4d el resultado del algoritmo *PSO-Converx-Hull*. Se puede observar que el algoritmo *PSO-Converx-Hull* logra tener el mejor costo mínimo $\approx$ 8,31. Seguido de éste, el algoritmo *PSO-Absorcion-Puntos-Cercanos* con un costo de $\approx$ 10,91, el algoritmo *PSO-Gale-Shapley* con un costo $\approx$ 11,29 y finalmente, el algoritmo *PSO-Hungaro* con un costo $\approx$ 11,5 (ver cuadro 5.2).

5.2.3. Indice Dunn

En la Fig. 5.5 se presentan las siguientes imágenes como resultado del clustering para D': la Fig. 5.5a muestra el resultado del algoritmo *PSO-Hungaro*, la Fig. 5.5b el resultado del algoritmo *PSO-Gale-Shapley*, la Fig. 5.5c el resultado del algoritmo *PSO-Absorcion-Puntos-Cercanos* y la Fig. 5.5d el resultado del algoritmo *PSO-Converx-Hull*. Se puede observar que el algoritmo *PSO-Absorcion-Puntos-Cercanos* logra tener el mejor Indice Dunn $\approx$ 372,07. Seguido de éste, el algoritmo *PSO-Hungaro* con Indice de $\approx$ 338,24, el algoritmo *PSO-Convex-Hull* con un costo $\approx$ 335,42 y

finalmente, el algoritmo *PSO-Gale-Shapley* con un costo $\approx 332{,}38$ (ver cuadro 5.3).

En la Fig. 5.6 se presentan las siguientes imágenes como resultado del clustering para D'': la Fig. 5.6a muestra el resultado del algoritmo *PSO-Hungaro*, la Fig. 5.6b el resultado del algoritmo *PSO-Gale-Shapley*, la Fig. 5.6c el resultado del algoritmo *PSO-Absorcion-Puntos-Cercanos* y la Fig. 5.6d el resultado del algoritmo *PSO-Converx-Hull*. Se puede observar que el algoritmo *PSO-Converx-Hull* logra tener el mejor Indice Dunn $\approx$ 237,63. Seguido de éste, el algoritmo *PSO-Absorcion-Puntos-Cercanos* con un costo de $\approx$ 225,02, el algoritmo *PSO-Hungaro* con un costo $\approx$ 209,89 y finalmente, el algoritmo *PSO-Gale-Shapley* con un costo $\approx$ 204,82 (ver cuadro 5.3).

5.3. Conclusión

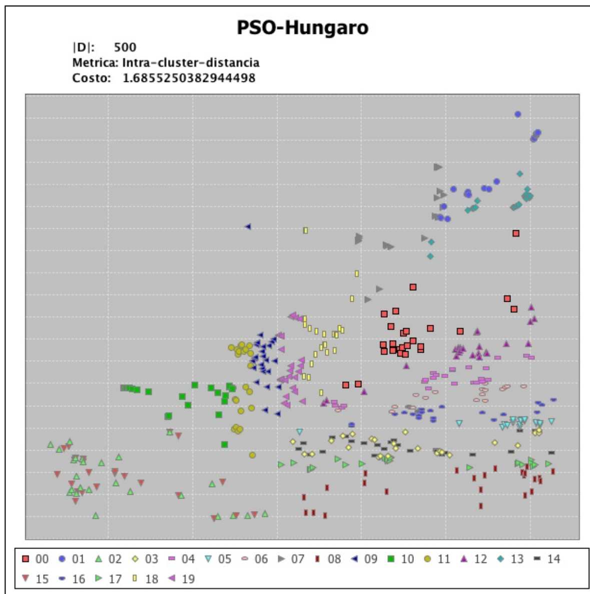
De acuerdo a los resultados presentados anteriormente, podemos concluir que dado un conjunto de datos distribuido de forma no uniforme, los algoritmos *PSO-Convex-Hull* y *PSO-Absorcion-Puntos-Cercanos* generan el mejor clustering balanceado, mientras que el clustering balanceado generado por el algoritmo *PSO-Hungaro* es menos óptimo que los dos anteriores. En la sección 4.2.1.4 se observa que el algoritmo *PSO-Hungaro* es el que presenta el mejor rendimiento para los datos distribuidos en forma uniforme, pero ¿por qué con los datos no uniformes presenta el peor rendimiento? Esto es debido a que los centroides aleatorios generados por *PSO-Hungaro* podrían estar en todas las coordenadas del rectángulo mínimo englobante[1], y una vez localizados los centroides, el algoritmo *PSO-Hungaro* comienza a balancear los puntos distruibuidos más cercanos a cada uno de los centroides. Este último paso podría estar sesgado a una optimización local a medida que las iteraciones van avanzando haciendo imposible continuar explorando las mejores soluciones (Fig. 5.7). Por eso, la solución final del algoritmo *PSO-Hungaro* depende fuertemente de los centroides iniciales generados, y si ellos caen inicialmente dentro de la cierre envolvente, aumenta la posibilidad de tener una solución mejor.

Por otra parte, aunque en la sección 4.2.1.4, el algoritmo *PSO-Convex-Hull* no se comporta mejor que el algoritmo *PSO-Hungaro* para los datos distribuidos uniformemente, sí se comporta mucho mejor que éste para los datos distribuidos no uniformemente. Esto es debido a que los centroides generados iniciales por *PSO-Convex-Hull* están ya dentro del cierre envolvente que abarca a los puntos y por ello, se llegan a formar con mayor facilidad *clusters* compactos, y así la convergencia de la solución final tiende a generar también *clusters* balanceados más compactos.

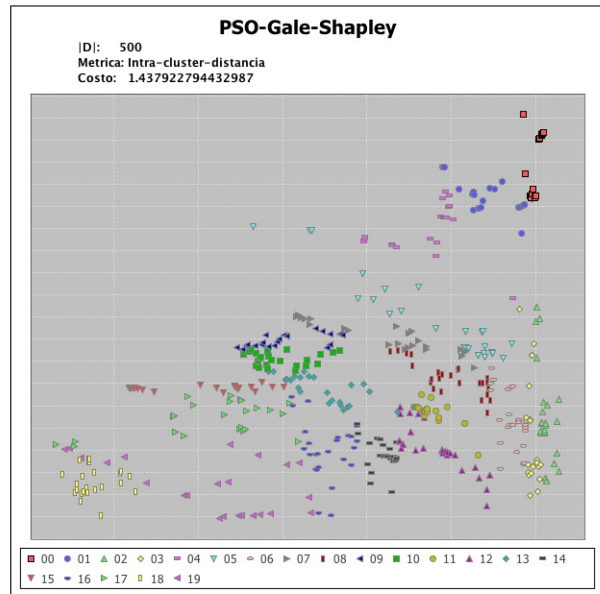
En caso del algoritmo *PSO-Absorcion-Puntos-Cercanos*, al igual que el algoritmo *PSO-Hungaro*, los centroides generados también podrían estar en todas las coordenadas del rectángulo mínimo englobante, ¿Por qué con los datos no uniformes toma ventaja sobre el algoritmo *PSO-Hungaro* en algunos casos (por ejemplo, las figuras en 5.1) y en algunos casos no (las figuras en 5.2)? Nuevamente se debe a las ubicaciones iniciales de los centroides generados aleatoriamente

[1]El rectángulo mínimo que encierra una región en la cual existen 1 o más elementos.

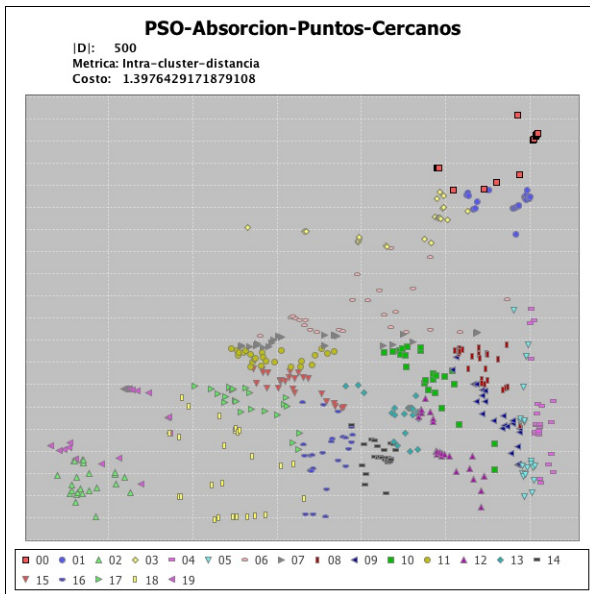
por *PSO* que se explicó anteriormente. Esto mismo ocurre con el caso del algoritmo *PSO-Gale-Shapley*. Según el capítulo 4, el algoritmo *PSO-Gale-Shapley* es el de menos calidad en cuanto a los resultados producidos, pero para los casos de datos distribuidos no uniforme éste puede llegar a entregar mejores resultados que el algoritmo *PSO-Hungaro* (por ejemplo, la Fig. 5.1). Por esta razón, podemos concluir que las ubicaciones iniciales de los centroides generados aleatoriamente tiene un grado importante de influencia sobre los resultados generados en los cuatro algoritmos para el caso de datos no uniformemente distribuidos.



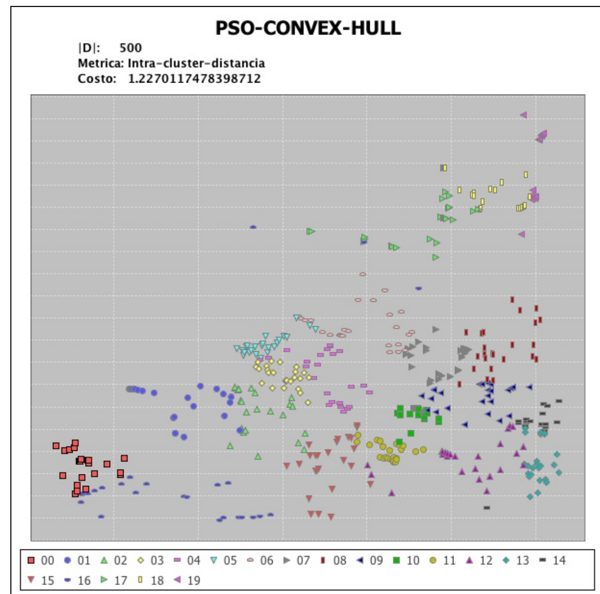
(a) PSO-Hungaro.



(b) PSO-Gale-Shapley,

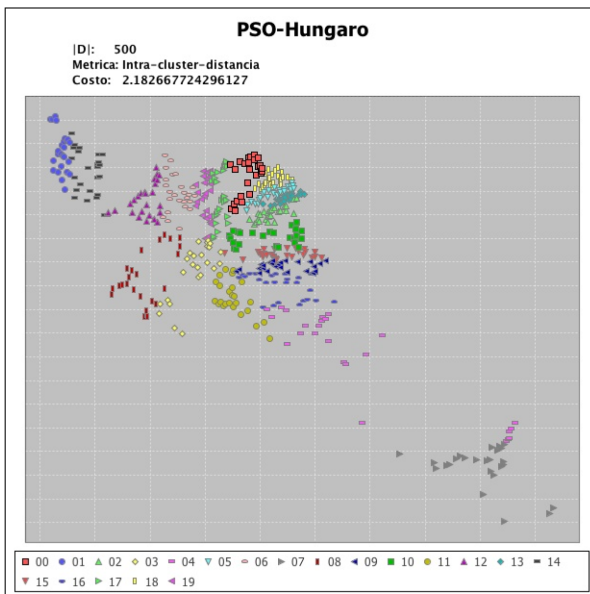


(c) PSO-Absorcion-Puntos-Cercanos.

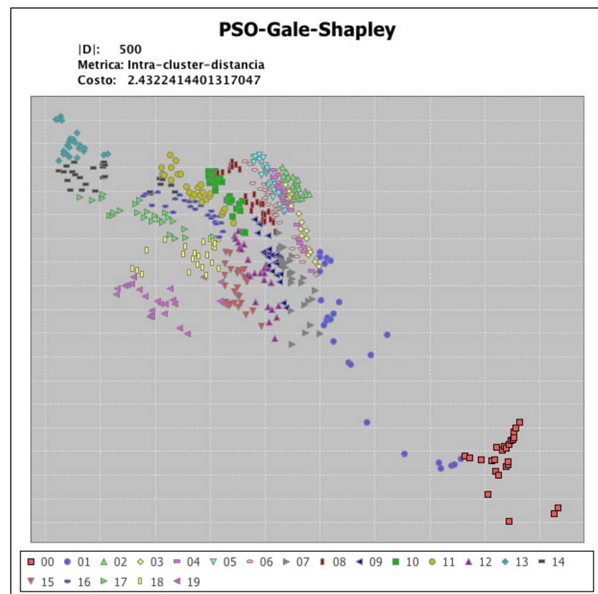


(d) PSO-Convex-Hull.

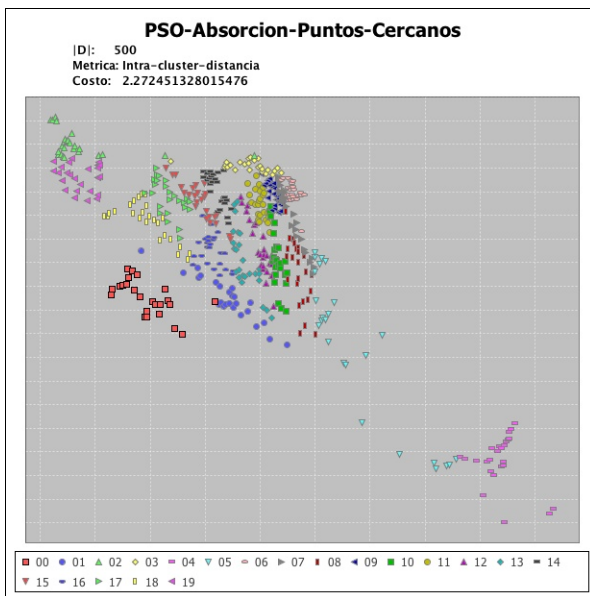
Figura 5.1: El resultado de la ejecución de los cuatro algoritmos dado $|D| = 500$, $K = 20$ y la métrica *Intra-cluster-distancia* con el conjunto de datos (D').



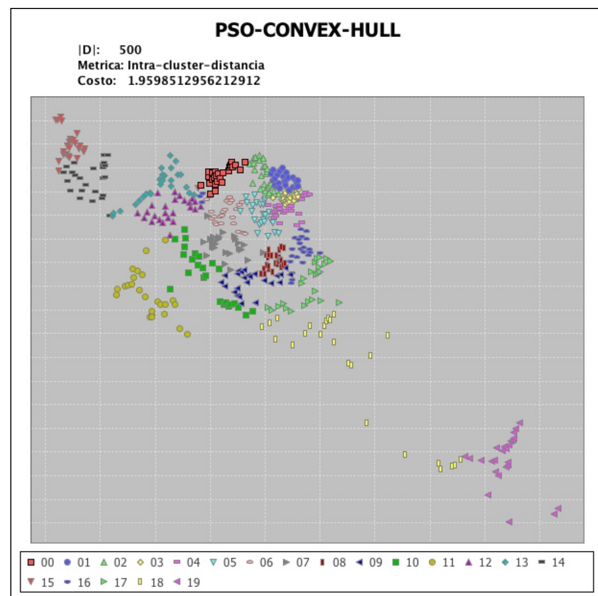
(a) PSO-Hungaro.



(b) PSO-Gale-Shapley.

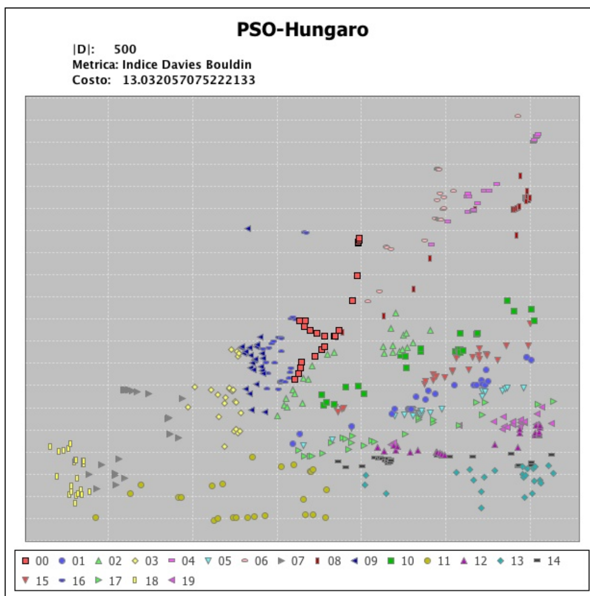


(c) PSO-Absorcion-Puntos-Cercanos.

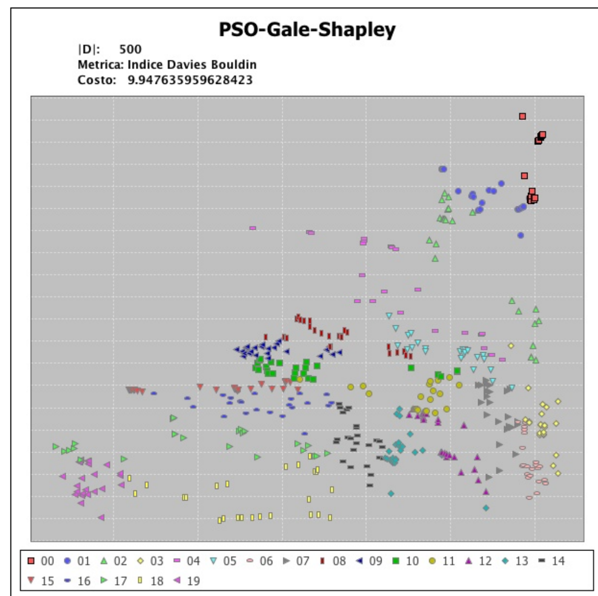


(d) PSO-Convex-Hull.

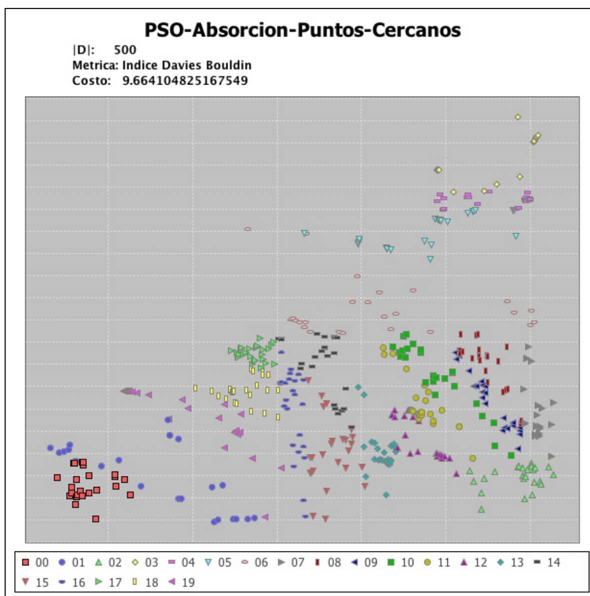
Figura 5.2: El resultado de la ejecución de los cuatro algoritmos dado $|D| = 500$, $K = 20$ y la métrica *Intra-cluster-distancia* con el conjunto de datos (D'').



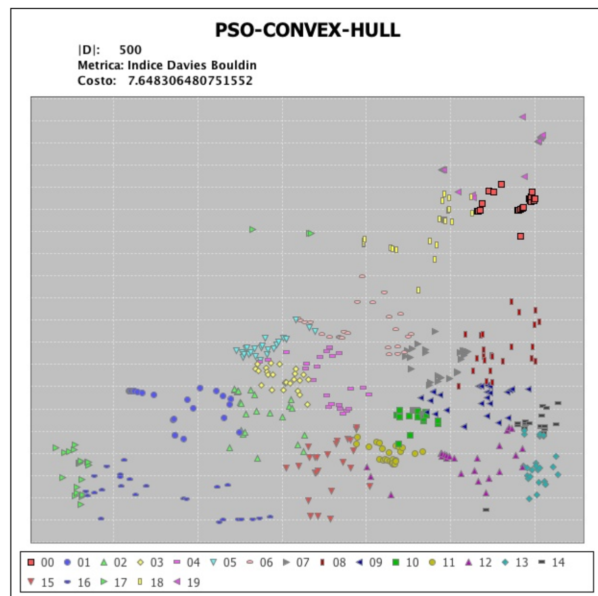
(a) PSO-Hungaro.



(b) PSO-Gale-Shapley.

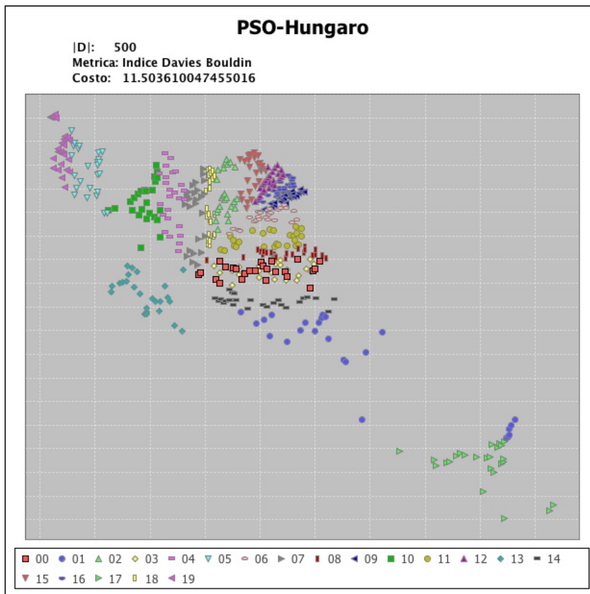


(c) PSO-Absorcion-Puntos-Cercanos.

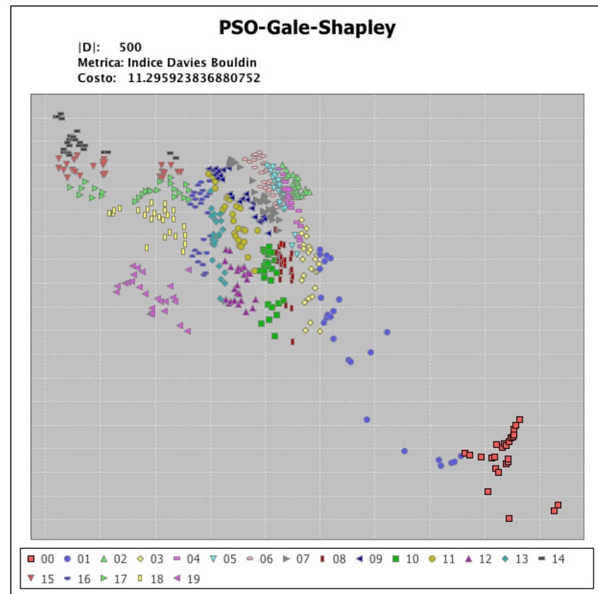


(d) PSO-Convex-Hull.

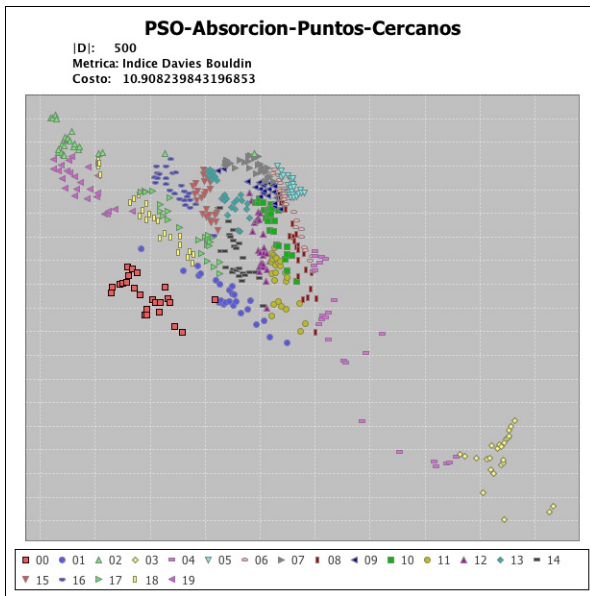
Figura 5.3: El resultado de la ejecución de los cuatro algoritmos dado $|D| = 500$, $K = 20$ y la métrica *Indice Davies-Bouldin* con el conjunto de datos (D').



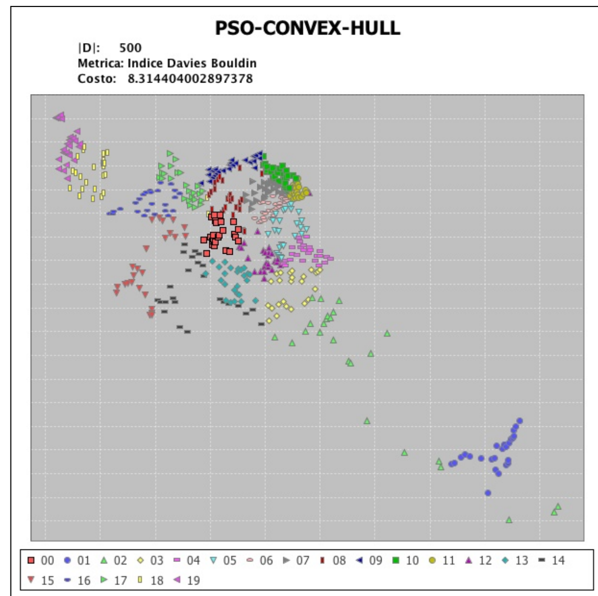
(a) PSO-Hungaro.



(b) PSO-Gale-Shapley.

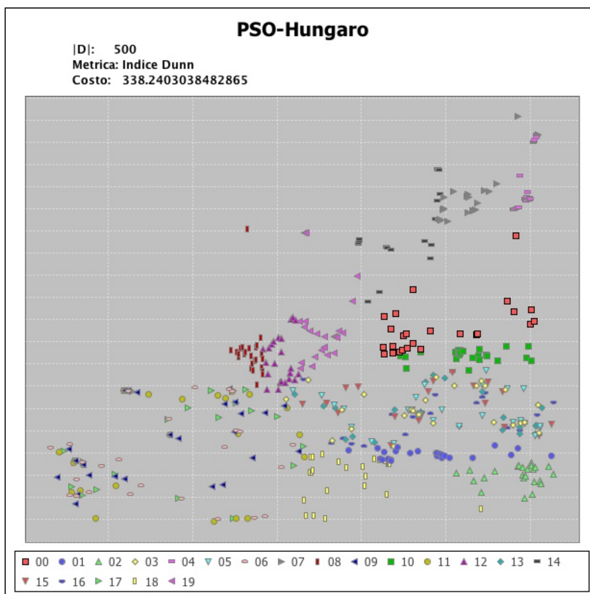


(c) PSO-Absorcion-Puntos-Cercanos.

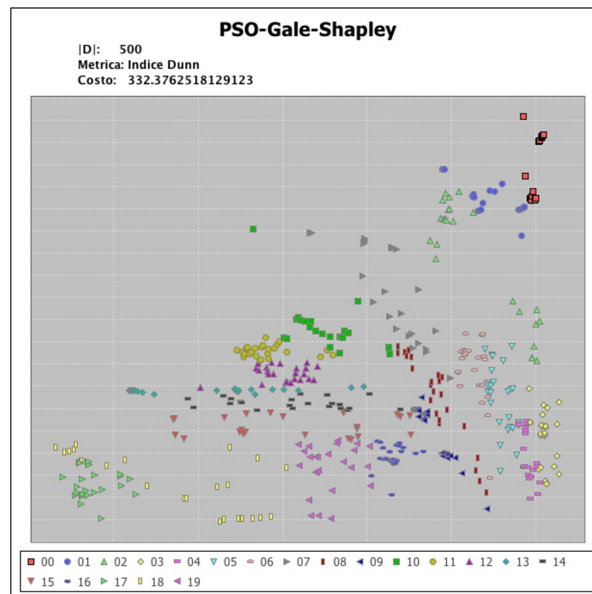


(d) PSO-Convex-Hull.

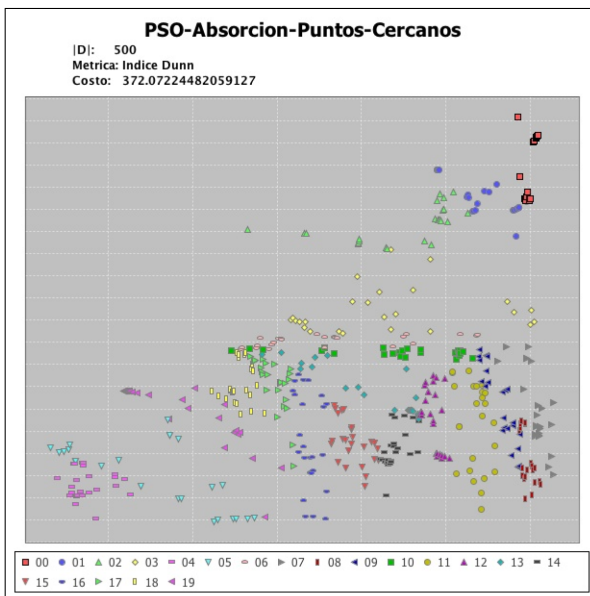
Figura 5.4: El resultado de la ejecución de los cuatro algoritmos dado $|D| = 500$, $K = 20$ y la métrica *Indice Davies-Bouldin* con el conjunto de datos (D'').



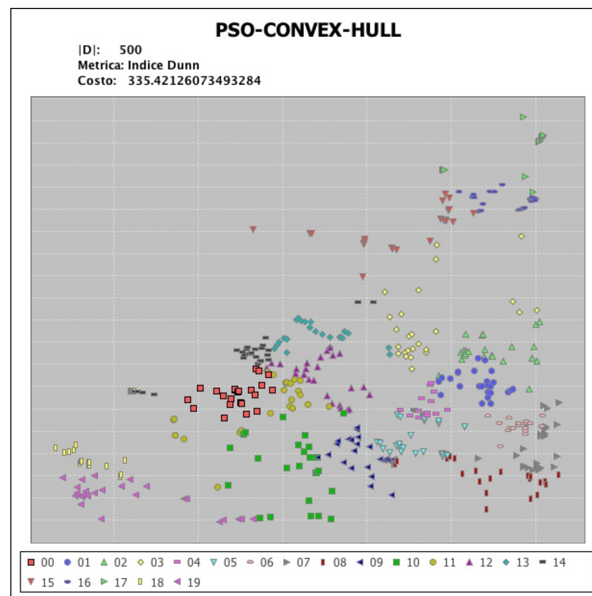
(a) PSO-Hungaro.



(b) PSO-Gale-Shapley.

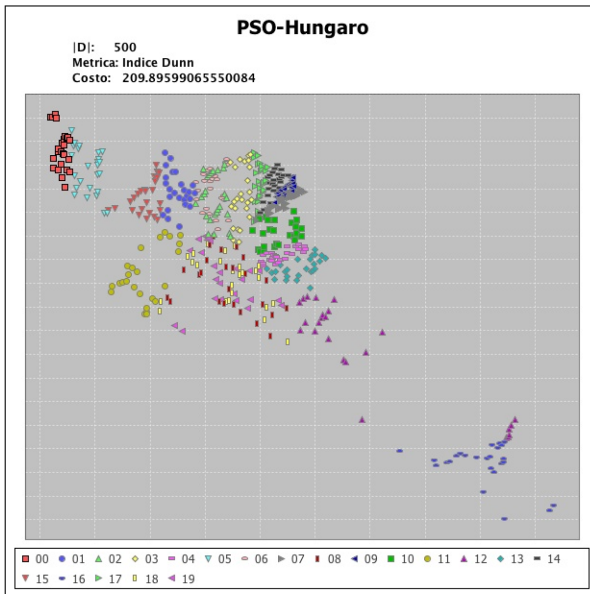


(c) PSO-Absorcion-Puntos-Cercanos.

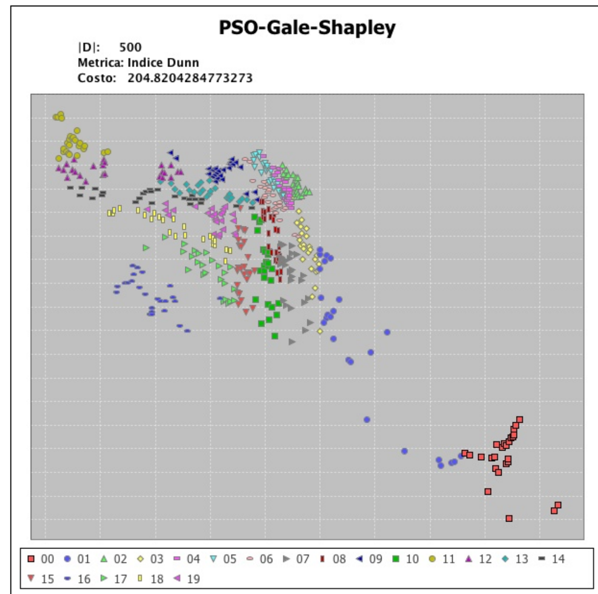


(d) PSO-Convex-Hull.

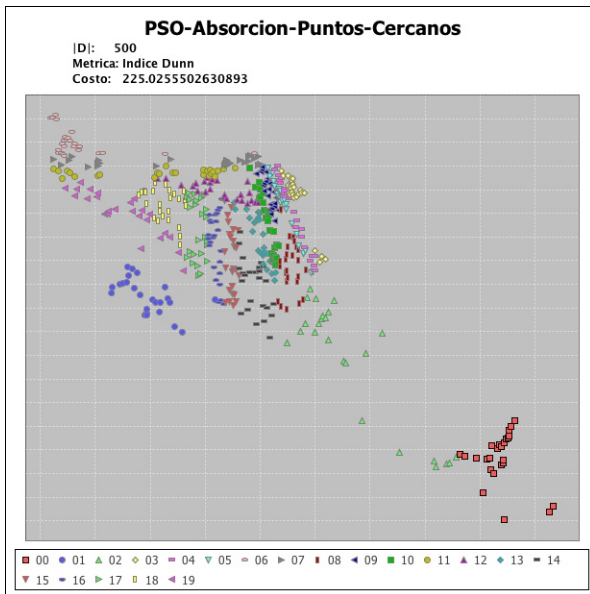
Figura 5.5: El resultado de la ejecución de los cuatro algoritmos dado $|D| = 500$, $K = 20$ y la métrica *Indice Dunn* con el conjunto de datos (D').



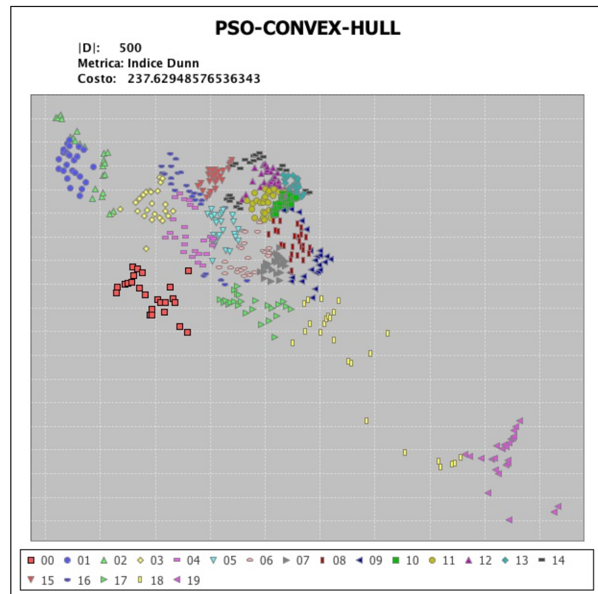
(a) PSO-Hungaro.



(b) PSO-Gale-Shapley.

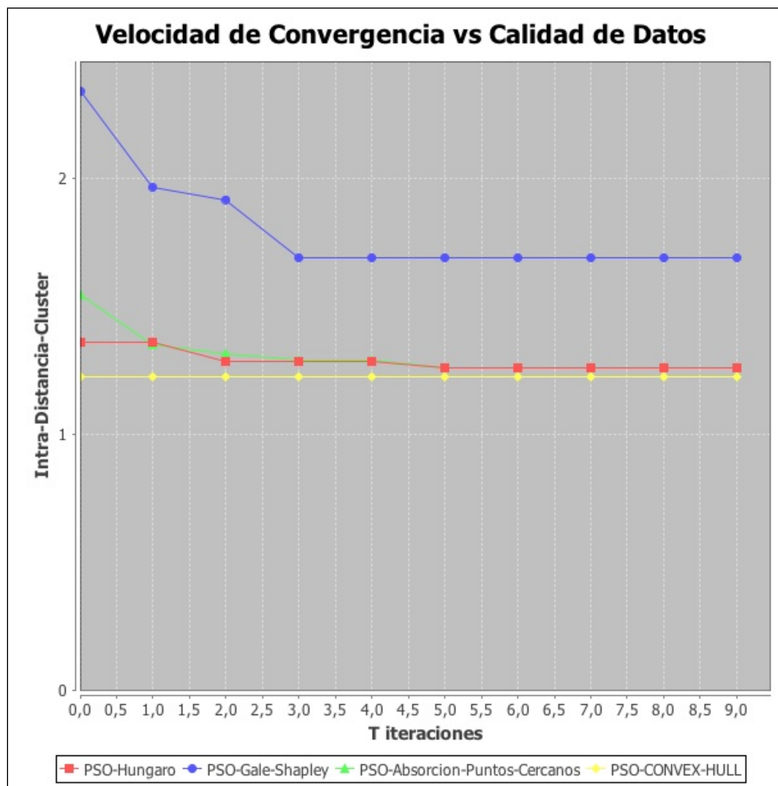


(c) PSO-Absorcion-Puntos-Cercanos.

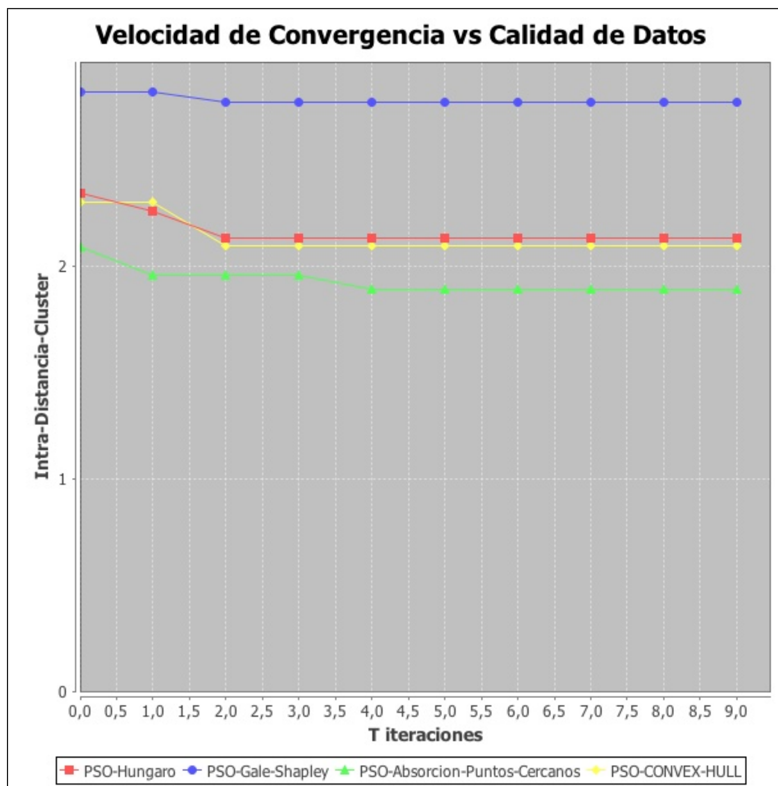


(d) PSO-Convex-Hull.

Figura 5.6: El resultado de la ejecución de los cuatro algoritmos dado $|D| = 500$, $K = 20$ y la métrica *Indice Dunn* con el conjunto de datos (D'').



(a) Datos (D').



(b) Datos (D'').

Figura 5.7: Velocidad VS. Calidad de Resultado con el conjunto de datos (D') y (D'').

Algoritmo	Datos	$\|D\|$	k	d	k	d	k	d	k	d	k	d
PSO-Hungaro	D'	100	2	0,3823	5	0,2717	10	0,1433	20	0,1064	25	0,1004
PSO-Gale-Shapley	D'	100	2	0,3815	5	0,2497	10	0,1428	20	0,0904	25	0,0811
PSO-Absorcion-Puntos-Cercanos	D'	100	2	0,3815	5	0,2497	10	0,1489	20	0,1004	25	0,0802
PSO-CONVEX-HULL	D'	100	2	0,3829	5	0,2472	10	0,1345	20	0,0882	25	0,0757
PSO-Hungaro	D'	300	2	1,9897	5	1,1761	10	0,8836	20	0,6844	25	0,7065
PSO-Gale-Shapley	D'	300	2	1,9897	5	1,1535	10	0,8238	20	0,6266	25	0,5742
PSO-Absorcion-Puntos-Cercanos	D'	300	2	1,9897	5	1,144	10	0,8584	20	0,6243	25	0,5326
PSO-CONVEX-HULL	D'	300	2	1,9938	5	1,1524	10	0,8335	20	0,5104	25	0,4497
PSO-Hungaro	D'	500	2	4,451	5	2,5702	10	2,0169	20	1,6855	25	1,6802
PSO-Gale-Shapley	D'	500	2	4,4424	5	2,4891	10	1,8313	20	1,4379	25	1,3126
PSO-Absorcion-Puntos-Cercanos	D'	500	2	4,4424	5	2,4576	10	1,7924	20	1,3976	25	1,2914
PSO-CONVEX-HULL	D'	500	2	4,4534	5	2,4823	10	1,8024	20	1,227	25	1,0665
PSO-Hungaro	D''	100	2	1,3029	5	0,8943	10	0,6973	20	0,6161	25	0,6121
PSO-Gale-Shapley	D''	100	2	1,3454	5	0,8668	10	0,6698	20	0,5865	25	0,5303
PSO-Absorcion-Puntos-Cercanos	D''	100	2	1,3029	5	0,8713	10	0,6637	20	0,561	25	0,5141
PSO-CONVEX-HULL	D''	100	2	1,3104	5	0,8891	10	0,6873	20	0,5674	25	0,4987
PSO-Hungaro	D''	300	2	3,6022	5	2,8512	10	2,2087	20	1,4532	25	1,5899
PSO-Gale-Shapley	D''	300	2	3,6022	5	2,8364	10	2,2445	20	1,2489	25	1,3632
PSO-Absorcion-Puntos-Cercanos	D''	300	2	3,6022	5	2,8364	10	2,2262	20	1,2539	25	1,3695
PSO-CONVEX-HULL	D''	300	2	3,5552	5	2,7419	10	2,1776	20	1,2811	25	1,2823
PSO-Hungaro	D''	500	2	7,1994	5	4,8588	10	3,0509	20	2,1827	25	2,33
PSO-Gale-Shapley	D''	500	2	7,2967	5	5,2406	10	3,4835	20	2,4322	25	2,2581
PSO-Absorcion-Puntos-Cercanos	D''	500	2	7,2967	5	5,1512	10	3,4273	20	2,2725	25	2,304
PSO-CONVEX-HULL	D''	500	2	7,2043	5	4,8289	10	3,0884	20	1,9599	25	1,8605

Cuadro 5.1: La inter-cluster-distancia d generada por los cuatro algoritmos propuestos con diferentes tamaños de datos y k para los conjunto de dato D' y D''.

Algoritmo	Datos	$\|D\|$	k	d	k	d	k	d	k	d	k	d
PSO-Hungaro	D'	100	2	0,7245	5	2,6299	10	4,2263	20	6,57	25	9,0982
PSO-Gale-Shapley	D'	100	2	0,7158	5	2,4723	10	3,7454	20	6,2949	25	7,8637
PSO-Absorcion-Puntos-Cercanos	D'	100	2	0,7158	5	2,4723	10	3,7865	20	6,815	25	7,5717
PSO-CONVEX-HULL	D'	100	2	0,7159	5	2,4284	10	3,9427	20	6,0329	25	6,6462
PSO-Hungaro	D'	300	2	1,1618	5	2,8426	10	5,7136	20	11,7099	25	14,7155
PSO-Gale-Shapley	D'	300	2	1,1478	5	2,7105	10	5,1637	20	9,6193	25	10,6314
PSO-Absorcion-Puntos-Cercanos	D'	300	2	1,1478	5	2,7813	10	5,1661	20	9,3072	25	10,532
PSO-CONVEX-HULL	D'	300	2	1,1524	5	2,6083	10	4,8602	20	6,9956	25	8,6239
PSO-Hungaro	D'	500	2	1,1056	5	2,661	10	6,3569	20	13,0321	25	15,6044
PSO-Gale-Shapley	D'	500	2	1,1028	5	2,5884	10	5,3534	20	9,9476	25	11,7962
PSO-Absorcion-Puntos-Cercanos	D'	500	2	1,1028	5	2,6041	10	5,1363	20	9,6641	25	11,5771
PSO-CONVEX-HULL	D'	500	2	1,1067	5	2,6046	10	4,9424	20	7,6483	25	9,1831
PSO-Hungaro	D''	100	2	0,982	5	3,0732	10	5,9479	20	11,2038	25	12,84
PSO-Gale-Shapley	D''	100	2	1,0148	5	2,9519	10	5,4561	20	10,9956	25	11,6716
PSO-Absorcion-Puntos-Cercanos	D''	100	2	0,9827	5	2,8958	10	5,2217	20	8,799	25	10,2781
PSO-CONVEX-HULL	D''	100	2	1,0102	5	3,1098	10	5,3492	20	8,8363	25	9,4008
PSO-Hungaro	D''	300	2	0,7554	5	2,9919	10	6,2004	20	12,5141	25	16,4782
PSO-Gale-Shapley	D''	300	2	0,7537	5	2,936	10	5,6976	20	9,4462	25	12,5984
PSO-Absorcion-Puntos-Cercanos	D''	300	2	0,7537	5	2,9482	10	5,6524	20	9,1841	25	10,9594
PSO-CONVEX-HULL	D''	300	2	0,7537	5	3,1026	10	5,3479	20	8,46	25	10,0495
PSO-Hungaro	D''	500	2	1,0455	5	3,0439	10	4,6926	20	11,5036	25	14,3238
PSO-Gale-Shapley	D''	500	2	1,0474	5	3,3764	10	5,7364	20	11,2959	25	13,8637
PSO-Absorcion-Puntos-Cercanos	D''	500	2	1,0456	5	3,309	10	5,9083	20	10,9082	25	13,9941
PSO-CONVEX-HULL	D''	500	2	1,048	5	2,9582	10	5,0227	20	8,3144	25	9,4344

Cuadro 5.2: El índice Davies-Doublin d generado por los cuatro algoritmos propuestos con diferentes tamaños de datos y k para los conjunto de dato D' y D''.

Algoritmo	Datos	$\|D\|$	k	d	k	d	k	d	k	d	k	d
PSO-Hungaro	D'	100	2	1,4229	5	18,6574	10	102,8953	20	408,4512	25	1067,6102
PSO-Gale-Shapley	D'	100	2	1,4229	5	16,2601	10	105,5044	20	524,2822	25	844,0857
PSO-Absorcion-Puntos-Cercanos	D'	100	2	1,4229	5	16,5082	10	104,4051	20	523,4213	25	749,8495
PSO-CONVEX-HULL	D'	100	2	1,4229	5	15,3086	10	104,2681	20	631,0637	25	950,1954
PSO-Hungaro	D'	300	2	0,9582	5	14,0885	10	66,6667	20	295,4448	25	485,3874
PSO-Gale-Shapley	D'	300	2	0,9645	5	11,5231	10	64,7383	20	313,1013	25	485,7107
PSO-Absorcion-Puntos-Cercanos	D'	300	2	0,9645	5	14,6372	10	71,6915	20	349,8597	25	654,2857
PSO-CONVEX-HULL	D'	300	2	0,9436	5	15,5004	10	66,5528	20	357,562	25	725,5699
PSO-Hungaro	D'	500	2	0,8697	5	12,855	10	75,1038	20	338,2403	25	526,7766
PSO-Gale-Shapley	D'	500	2	0,9443	5	13,6973	10	73,5091	20	332,3763	25	548,5552
PSO-Absorcion-Puntos-Cercanos	D'	500	2	0,9443	5	13,7405	10	77,5402	20	372,0722	25	594,9788
PSO-CONVEX-HULL	D'	500	2	0,9414	5	13,9907	10	71,7303	20	335,4213	25	558,4322
PSO-Hungaro	D''	100	2	0,2369	5	2,388	10	11,7663	20	49,2678	25	77,6352
PSO-Gale-Shapley	D''	100	2	0,2253	5	2,2607	10	11,7536	20	48,5359	25	77,3136
PSO-Absorcion-Puntos-Cercanos	D''	100	2	0,2368	5	2,4077	10	11,8407	20	49,7077	25	78,3579
PSO-CONVEX-HULL	D''	100	2	0,2315	5	2,4174	10	11,6543	20	49,5792	25	78,5271
PSO-Hungaro	D''	300	2	0,3807	5	3,4117	10	15,2133	20	63,0432	25	98,5855
PSO-Gale-Shapley	D''	300	2	0,3807	5	3,4915	10	15,7202	20	66,0042	25	103,0288
PSO-Absorcion-Puntos-Cercanos	D''	300	2	0,3807	5	3,4935	10	15,6574	20	66,1814	25	103,8241
PSO-CONVEX-HULL	D''	300	2	0,3807	5	3,4807	10	15,7085	20	64,9618	25	97,0888
PSO-Hungaro	D''	500	2	0,6862	5	7,7342	10	38,6943	20	209,896	25	362,3626
PSO-Gale-Shapley	D''	500	2	0,6133	5	6,6034	10	35,6621	20	204,8204	25	287,1227
PSO-Absorcion-Puntos-Cercanos	D''	500	2	0,6889	5	7,2708	10	37,6501	20	225,0256	25	372,5977
PSO-CONVEX-HULL	D''	500	2	0,6889	5	7,8683	10	39,2686	20	237,6295	25	382,3247

Cuadro 5.3: El índice Dunn d generado por los cuatro algoritmos propuestos con diferentes tamaños de datos y k para los conjunto de dato D' y D''.

Capítulo 6

Conclusiones

Del análisis de los capítulos anteriores, se pueden obtener las siguientes conclusiones:

1. La creación de *clusters* balanceados de los algoritmos *PSO-Absorción-Puntos-Cercanos* y *PSO-Convex-Hull* depende fuertemente del orden con que los centroides comienzan a absorber los puntos más cercanos. En cambio, los algoritmos *PSO-Hungaro* y *PSO-Gale-Shapley* solamente dependen de los centroides generados y no del orden de los *clusters* a crear.

2. Cuando se prioriza el resultado óptimo sin tener consideración del tiempo de ejecución, el algoritmo *PSO-Hungaro* es el que presenta el mejor rendimiento para lograr el resultado esperado. Sin embargo, en el caso de considerar el tiempo de ejecución, el algoritmo *PSO-Convex-Hull* es el que obtiene la mayor ventaja sobre los demás para los puntos distribuidos uniformemente para $k \geq 20$ y $|D| \geq 1000$, mientras que *PSO-Absorcion-Puntos-Cercanos* es el que obtiene la ventaja cuando $k \leq 20$ y $|D| \geq 1000$.

3. Cuando se prioriza lo compacto de los *clusters* generados sin considerar el tiempo de ejecución, el algoritmo *PSO-Hungaro* es el de mejor rendimiento para lograr el resultado esperado. En caso de considerar el tiempo de ejecución, tanto el algoritmo *PSO-Absorción-Puntos-Cercanos* como *PSO-Convex-Hull* podrán ser considerados como algoritmos alternativos. Esto es porque el primero tiene la ventaja de tener los centroides generados desde diferentes posiciones en el mapa, lo cual hace que los resultados puedan acercarse al óptimo; pero se necesita ejecutar varias veces el algoritmo para generar los *clusters* balanceados compactos; el segundo tiene la mayor cobertura de centroide; sin embargo, si los *clusters* balanceados se forman desde los puntos internos a los extremos, entonces la generación de los *clusters* balanceados no siempre estará compactos.

4. El algoritmo *PSO-Gale-Shapley* es el algoritmo más deficiente para la generación de *clusters* balanceados. Esto es debido a que para que éstos sean creados, se reemplaza la distancia del punto al centroide en el orden de preferencia, y así cuando todos los puntos ya han sido

asignados, es cuando se calcula el costo final. En cambio, el algoritmo *PSO-Hungaro* calcula la suma total de los costos de cada cluster a medida que éstos se crean, logra obtener el mejor resultado..

5. Generalmente, cuando la cantidad de los datos es grande ($D \geq 300$), el tiempo de ejecución de los cuatro algoritmos aumenta. Sin embargo, dada una cantidad fija de datos, el tiempo de ejecución disminuye a medida que aumenta la cantidad de *clusters*. En los experimentos que desarrollamos, cuando la cantidad de datos llega a superar 300 y la cantidad de *clusters* supera los 20, el tiempo de ejecución de los cuatro algoritmos se estabiliza (Fig. 4.12).

6. Para los conjuntos de datos distribuidos de manera no uniforme, no se puede determinar cual de los 4 algoritmos llegaría a tener un resultado de calidad por la ubicación de los centroides iniciales generados aleatoriamente. Esto merece un estudio más detallado para intentar obtener condiciones para ciertos algoritmos que se usan para las distintas configuraciones.

6.1. Trabajo Futuro

- En cuanto al diseño de los algoritmos, se propone como trabajo futuro:
  - Diseñar un algoritmo exacto que produce *clusters* balanceados en función de las distintas métricas *Intra-Cluster-Distancia*, *Indice Davies-Bouldin* e *Indice Dunn.*
  - Diseñar algorimos aproximados que no dependan de los centroides para la creación de *clusters*, sino de la inter-distancia entre los puntos.
  - Investigar cómo se determina el grado de la uniformidad de los datos para usarlo como un criterio de comparación entre los algoritmos de clustering balanceado.
- En cuanto a la adaptación de los algoritmos:
  - Ajustar el algoritmo de *PSO* para generar los centroides siempre dentro de la *mínima envolvente de cierre* para los cuatro algoritmos propuestos.
  - Adaptar el algoritmo *PSO-Hungaro* de forma natural a la ejecución en entornos multiprocesador, especialmente en sistemas de memoria compartida.
  - Agregar el mecanismo de aprendizaje para el algoritmo *PSO-Absorción-Puntos-Cercanos* de manera que cada centroide podría absorber inteligentemente los puntos más cercanos para la creación de los *clusters* balanceados.

Apéndice A

Código de implementación de los algoritmos propuestos

Los algoritmos propuestos del presente trabajo de tesis se desarrollaron bajo la plataforma estándar de JAVA 5. Los algoritmos usados son *Particle Swarm Optimization* (Algoritmo 2), *Algoritmo Húngaro* (Algoritmo 4), *algoritmo Gale-Shapley*,*algoritmo Convex-Hull*, *Algoritmo PSO-Húngaro* (Algoritmo 5), *Algoritmo PSO-Gale-Shapley* (Algoritmo 7), *Algoritmo PSO-Absorción-Puntos-Cercanos* (Algoritmo 8) y *Algoritmo PSO-Convex-Hull* (Algoritmo 9). El código de implementación de los algoritmos señalados quedan publicados en la siguiente página web: `http://desarrollo.mapcity.com/Tesis`. A continuación, se presenta el código de implementación de cada uno de estos algoritmos.

A.1. Particle Swarm Optimization

El código de implementación de *PSO* es bajado de la siguiente página web: `http://jswarm-pso.sourceforge.net/`. Una clase principal que se modificará para la implementación de los algoritmos propuestos es *Particle*, la cual es una superclase que consta de los métodos generales. Aquí, presentamos algunos de los métodos principales:

```java
/**
 * Initialize a particles's position and velocity vectors
 * @param maxPosition : Vector stating maximum position for each dimension
 * @param minPosition : Vector stating minimum position for each dimension
 * @param maxVelocity : Vector stating maximum velocity for each dimension
 * @param minVelocity : Vector stating minimum velocity for each dimension
 */
public void init(double maxPosition[], double minPosition[],
          double maxVelocity[], double minVelocity[]) {
        for (int i = 0; i < position.length; i++) {
                if (Double.isNaN(maxPosition[i]))
                        throw new RuntimeException("maxPosition[" + i + "] is NaN!");
                if (Double.isInfinite(maxPosition[i]))
                        throw new RuntimeException("maxPosition[" + i + "] is Infinite!");

                if (Double.isNaN(minPosition[i]))
                        throw new RuntimeException("minPosition[" + i + "] is NaN!");
                if (Double.isInfinite(minPosition[i]))
                        throw new RuntimeException("minPosition[" + i + "] is Infinite!");

                if (Double.isNaN(maxVelocity[i]))
                        throw new RuntimeException("maxVelocity[" + i + "] is NaN!");
                if (Double.isInfinite(maxVelocity[i]))
                        throw new RuntimeException("maxVelocity[" + i + "] is Infinite!");

                if (Double.isNaN(minVelocity[i]))
                        throw new RuntimeException("minVelocity[" + i + "] is NaN!");
                if (Double.isInfinite(minVelocity[i]))
```

```
                        throw new RuntimeException("minVelocity[" + i + "] is Infinite!");

                // Initialize using uniform distribution
                position[i] = (maxPosition[i] - minPosition[i]) *
                               Math.random() + minPosition[i];
                velocity[i] = (maxVelocity[i] - minVelocity[i]) *
                              Math.random() + minVelocity[i];
                bestPosition[i] = Double.NaN;
            }
        }
```

El método *init* se trata de generar con un valor aleatorio para las coordenadas de la partícula para inicializar la primera iteración. Una vez inicalizado, ya no se volverá a ocupar. Este método podrían cambiarse de acuerdo a la estrategias que se van a ocupar para generar los valores con mayor precisión.

Ahora presentamos ahora los métodos principales la clase *Swarm*:

```
public void evolve() {
        // Initialize (if not already done)
        if (particles == null) init();
        evaluate(); // Evaluate particles
        update(); // Update positions and velocities
        variablesUpdate.update(this);
}
```

En la clase *Swarm*, el método *evolve()* se trata de hacer una iteración. En cada iteración, se hace una evaluación de cada partícula incluida calculando su función de adaptación (*evaluate()*). Luego, se invocar al método *update()*, que actualiza la posición de las partículas con la mejor local y la mejor histórica. A continuación, presentamos cómo puede construir el propio modelo:

1. Crear una propia función de adaptación. Se debe heredar de la clase *cl.mapcity.routing.pso.FitnessFunction* y el mtodo de evaluacin personalizada debe ser creado.

   ```
   import cl.mapcity.routing.pso.FitnessFunction;
   public class MyFitnessFunction extends FitnessFunction {
           public double evaluate(double position[]) {
                   return position[0] + position[1];
           }
   }
   ```

2. Crear una clase partícula que hereda de la clase *cl.mapcity.routing.pso.Particle*:

   ```
   import cl.mapcity.routing.pso.Particle;
   public class MyParticle extends Particle {
           // Create a 2-dimentional particle
           public MyParticle() { super(2); }
   }
   ```

3. Crear el *Swarm* e iterarlo.

   ```
   // Create a swarm (using 'MyParticle' as sample particle
   ```

```
// and 'MyFitnessFunction' as finess function)
Swarm swarm = new Swarm(Swarm.DEFAULT_NUMBER_OF_PARTICLES
                , new MyParticle()
                , new MyFitnessFunction());
// Set position (and velocity) constraints.
// i.e.: where to look for solutions
swarm.setMaxPosition(1);
swarm.setMinPosition(0);
// Optimize a few times
for( int i = 0; i < 20; i++ ) swarm.evolve();
// Print en results
System.out.println(swarm.toStringStats())
```

Para los algoritmos propuestos basados en *PSO*, bastan crear una clase que hereda de *cl.mapcity.routing.pso.Particle* donde guarda las coordenadas y una clase que hereda de cl.mapcity.routing.pso.FitnessFunction donde se ejecuta el algoritmo. Más tarde, presentaremos las creadas al respecto.

A.2. Algoritmo Húngaro

El código de implementación de *algoritmo Húngaro* es bajado de la siguiente página web: `http://konstantinosnedas.com/dev/soft/munkres.htm`. Aquí, presentamos el código del método que será reutilizado para el algoritmo *PSO-Húngaro*: *hgAlgorithm*.

```
public static int[][] hgAlgorithm (double[][] array, String sumType)
	{		//Create the cost matrix
		double[][] cost = copyOf(array);
			//Then array is weight array. Must change to cost.
		if (sumType.equalsIgnoreCase("max"))
		{
			//Generate cost by subtracting.
			double maxWeight = findLargest(cost);
			for (int i=0; i<cost.length; i++)
			{
				for (int j=0; j<cost[i].length; j++)
				{
					cost [i][j] = (maxWeight - cost [i][j]);
				}
			}
		}
		//Find largest cost matrix element (needed for step 6).
		double maxCost = findLargest(cost);
		//The mask array.
		int[][] mask = new int[cost.length][cost[0].length];
		//The row covering vector.
		int[] rowCover = new int[cost.length];
		//The column covering vector.
		int[] colCover = new int[cost[0].length];
		//Position of last zero from Step 4.
		int[] zero_RC = new int[2];
		int step = 1;
		boolean done = false;
```

```
while (done == false)   //main execution loop
{
        switch (step)
    {
                case 1:
                        step = hg_step1(step, cost);
            break;
        case 2:
            step = hg_step2(step, cost, mask, rowCover, colCover);
                        break;
        case 3:
            step = hg_step3(step, mask, colCover);
                        break;
        case 4:
            step = hg_step4(step, cost, mask, rowCover, colCover, zero_RC);
                        break;
        case 5:
                        step = hg_step5(step, mask, rowCover, colCover, zero_RC);
                        break;
        case 6:
                step = hg_step6(step, cost, rowCover, colCover, maxCost);
                        break;
            case 7:
            done=true;
            break;
    }
}//end while
```

```
        int[][] assignment = new int[array.length][2];  //Create the returned array.
        for (int i=0; i<mask.length; i++)
        {
            for (int j=0; j<mask[i].length; j++)
            {
                if (mask[i][j] == 1)
                {
                    assignment[i][0] = i;
                    assignment[i][1] = j;
                }
            }
        }
        return assignment;
    }
```

A.3. Algoritmo Gale-Shapley

El código de implementación de *algoritmo Gale-Shapley* es bajado de la siguiente página web: `http://rosettacode.org/wiki/Stable_marriage_problem#Java`. Aquí, presentamos el código del método que será reutilizado para el algoritmo *PSO-Gale-Shapley*: *match*.

```
private static Map<String, String> match(
        List<String> guys,
        Map<String, List<String>> guyPrefers,
        Map<String, List<String>> girlPrefers){
            Map<String, String> engagedTo = new TreeMap<String, String>();
            List<String> freeGuys = new LinkedList<String>();
            freeGuys.addAll(guys);
            while(!freeGuys.isEmpty()){
                        //get a load of THIS guy
                String thisGuy = freeGuys.remove(0);
                List<String> thisGuyPrefers = guyPrefers.get(thisGuy);
                for(String girl:thisGuyPrefers){
                    if(engagedTo.get(girl) == null){//girl is free
                        engagedTo.put(girl, thisGuy); //awww
                        break;
                    }else{
                        String otherGuy = engagedTo.get(girl);
                        List<String> thisGirlPrefers = girlPrefers.get(girl);
                        if(thisGirlPrefers.indexOf(thisGuy) <
                            thisGirlPrefers.indexOf(otherGuy)){
                            //this girl prefers this guy to the guy
                            //she's engaged to
                            engagedTo.put(girl, thisGuy);
                            freeGuys.add(otherGuy);
                            break;
                        }//else no change...keep looking for this guy
                    }
                }
            }
            return engagedTo;
        }
```

A.4. Algoritmo Convex-Hull

El código de implementación de *algoritmo Convex-Hull* es bajado de la siguiente página web: `http://geoinformatica-msc.googlecode.com/svn-history/r51/trunk/ConvexHull/src/business/ConvexHull.java`. Aquí, presentamos el código del método que será reutilizado para el algoritmo *PSO-Convex-Hull*: *match*.

```
public void calcularPoligonoConvexo(List<T> puntos){
        this.calcularMenorOrdenanda(puntos);
        this.calcularArgumento(puntos);
        this.ordenarQuicksort(puntos, 0, puntos.size()-1);
        this.definirPuntosCH(puntos);
    }

    private void calcularMenorOrdenanda(List<T> vector){
            minimo.setY(vector.get(0).getY());
            minimo.setX(vector.get(0).getX());
            for (int i = 1; i<vector.size();i++) {
                    if (vector.get(i).getY()<minimo.getY()){
                            minimo.setY(vector.get(i).getY());
                            minimo.setX(vector.get(i).getX());
                    }
            }
    }

    private void calcularArgumento(List<T> puntos){
            double xmin = minimo.getX();
            double ymin = minimo.getY();
            double xpt = 0;
            double ypt = 0;
            double arg = 0;

            for (int i = 0; i<puntos.size();i++)
            {
                xpt =  puntos.get(i).getX();
                ypt =  puntos.get(i).getY();

                if ( (xpt==0) && (ypt > 0) ){
                        arg = 90;
```

```
            }
            else if ( (xpt==0) && (ypt<0)){
                    arg = 270;
            }
            else{
                    if (ypt==ymin && xpt==xmin)
                            arg = 0;
                    else
                            arg = Math.atan((ypt-ymin)/
                            (xpt-xmin))*180/Math.PI;
                    if (xpt>0 && ypt < 0) {
                            arg = arg + 360;
                    }
                    else if (xpt<0){
                            arg = arg + 180;
                    }
            }
                argMap.put(puntos.get(i).getObjName(), arg);
        }
}

private void definirPuntosCH(List<T> vector){
    puntosEliminados = new ArrayList<T>();
    double xa,xb,xc,ya,yb,yc;
    int i1,i2,i3;
    double pcruz;

    for (int i=0; i < vector.size(); i++){
                    if(vector.size()<=3) return;
            i1 = i;
            if (i+1 < vector.size())
                    i2 = i+1;
            else
                    i2 = i+1-vector.size();
            if (i+2 < vector.size())
                    i3 = i+2;
            else
                    i3 = i+2-vector.size();
```

```
            xa = vector.get(i1).getX();
            xb = vector.get(i2).getX();
            xc = vector.get(i3).getX();
            ya = vector.get(i1).getY();
            yb = vector.get(i2).getY();
            yc = vector.get(i3).getY();
            pcruz = (yc-yb)*(xa-xb)-(ya-yb)*(xc-xb);
            if (pcruz >= 0){
                    puntosEliminados.add(vector.get(i2));
                    vector.remove(i2);
                    i--;
            }
        }
}

private void ordenarQuicksort(
            List<T> vector,
            int primero, int ultimo){
    int i=primero, j=ultimo;
    Double pivote =
            argMap.get(vector.get((primero + ultimo) / 2).getObjName());
    T auxiliar;
    do{
            while(argMap.get(vector.get(i).getObjName())<pivote) i++;
            while(argMap.get(vector.get(j).getObjName())>pivote) j--;
            if (i<=j){
                    auxiliar=vector.get(j);
                    vector.set(j, vector.get(i));
                    vector.set(i, auxiliar);
                    i++;
                    j--;
            }
    } while (i<=j);
    if(primero<j) ordenarQuicksort(vector,primero, j);
    if(ultimo>i) ordenarQuicksort(vector,i, ultimo);
}
```

A.5. Algoritmo PSO-Hungaro

Tal como se señala en la sección A.1, para implementar los algoritmos basados en *PSO*, se ha de crear una clase que hereda de cl.mapcity.routing.pso.FitnessFunction donde se ejecuta el algoritmo. Aquí presentamos el código de la implementación del algoritmo *PSO-Hungaro*.

```
public double evaluate(double position[]) {
                Cluster []clusters = new Cluster[position.length/2];
                int kc=0;
                for(int i=0; i<position.length;i+=2){
                        clusters[kc] = new Cluster(String.valueOf(i+1));
                        clusters[kc].mDataPoints = new ArrayList<DataPoint>();
                        Centroid centroid =
                        puntos.get(0).getFactory().
                        createNewVertexCentroide(position[i], position[i+1]);
                        clusters[kc++].setCentroid(centroid);
                }
                long time=System.currentTimeMillis();
                double[][] array =
                new double[this.puntos.size()][this.puntos.size()];
                for(int i=0; i<clusters.length;i++)
                        for(int j=0; j<this.puntos.size();j++){
                                array[i][j] = this.puntos.get(j).
                                calcDistance(clusters[i].getCentroid());
                        }
                for(int i=clusters.length-1;i<this.puntos.size();i++){
                        for(int j=0; j<this.puntos.size();j++){
                                array[i][j] =array[i%clusters.length][j];
                        }
                }
                time=System.currentTimeMillis();
                int[][] assignment = HungarianAlgorithm.hgAlgorithm(array,"min");
                double sum = 0;
                for (int i=0; i<assignment.length; i++)
                        sum = sum + array[assignment[i][0]][assignment[i][1]];

                for(int i=0; i<assignment.length;i++){
                        clusters[i%clusters.length].
                        addDataPoint(this.puntos.get(assignment[i][1]));
```

```
        }
        List<Cluster> cx = new ArrayList<Cluster>();
        for(Cluster c: clusters)
                cx.add(c);
        return super.getMetricValue(cx, factory);
}
```

A.6. Algoritmo PSO-Gale-Shapley

En esta sección, presentamos el código de la implementación del algoritmo *PSO-Gale-Shapley*.

```
public double evaluate(double position[]) {
        Map<String, Cluster> clustersMap = new HashMap<String, Cluster>();
        List<String> guys = new ArrayList<String>();
        List<String> girls = new ArrayList<String>();
        Map<String, List<String>> guyPrefers =
                new HashMap<String, List<String>>();
        Map<String, List<String>> girlPrefers =
                new HashMap<String, List<String>>();
        Cluster [] clusters = new Cluster[position.length/2];
        int kc=0;
        for(int i=0; i<position.length;i+=2){
                clusters[kc] = new Cluster(String.valueOf(i+1));
                clusters[kc].mDataPoints = new ArrayList<DataPoint>();
                Centroid centroid =
                puntos.get(0).getFactory().
                createNewVertexCentroide(position[i], position[i+1]);
                clusters[kc].setCentroid(centroid);
                Collections.sort(this.puntos,
                new SortByDistance(centroid, factory));
                List<String> nombres1 = new ArrayList<String>();
                for(DataPoint p: this.puntos){
                        nombres1.add(p.getObjName());
                }
                for(int k=0; k<this.puntos.size()/clusters.length;k++){
                        guys.add(clusters[kc].getName()+"-"+k);
                        guyPrefers.put(
                        clusters[kc].getName()+"-"+
```

```
                    k,new ArrayList<String>(nombres1));
                    clustersMap.put(
                    clusters[kc].getName()+"-"+k, clusters[kc]);
            }
            kc++;
        }
        for(DataPoint p:this.puntos){
            Arrays.sort(clusters,
            new SortByClusterCentroid(
            new Centroid(p.getX(), p.getY()), factory));
            List<String> nombres1 = new ArrayList<String>();
            for(Cluster d: clusters){
                for(int k=0; k<this.puntos.size()/clusters.length;k++){
                    nombres1.add(d.getName()+"-"+k);
                }
            }
            girls.add(p.getObjName());
            girlPrefers.put(p.getObjName(),nombres1);
        }
        long tiempo = System.currentTimeMillis();
        Map<String, String> matches = match(guys, guyPrefers, girlPrefers);
        for(Map.Entry<String, String> couple:matches.entrySet()){
            clustersMap.get(couple.getValue()).
        addDataPoint(this.puntosMap.get(couple.getKey()));
        }
            List<Cluster> cx = new ArrayList<Cluster>();
            for(Cluster c: clusters)
                cx.add(c);
            return super.getMetricValue(cx, factory);

}
```

A.7. Algoritmo PSO-Absorción-Puntos-Cercanos

En esta sección, presentamos el código de la implementación del algoritmo *PSO-Absorción-Puntos-Cercanos*.

```
public double evaluate(double position[]) {
```

```
            Cluster [] clusters = new Cluster[position.length/2];
            int kc=0;
            for(int i=0; i<position.length;i+=2){
                    clusters[kc] = new Cluster(String.valueOf(i+1));
                    clusters[kc].mDataPoints = new ArrayList<DataPoint>();
                    Centroid centroid = puntos.get(0).getFactory().
                    createNewVertexCentroide(position[i], position[i+1]);
                    clusters[kc++].setCentroid(centroid);
            }
            List<Cluster> clustersAux =getCluster(clusters);
            return super.getMetricValue(clustersAux, factory);
    }
```

A.8. Algoritmo PSO-Convex-Hull

En esta sección, presentamos el código de la implementación del algoritmo *PSO-Convex-Hull*.

```
public ClusteringBalancedFitnessConvexHull(List<T> puntos,
         int max,
         VertexGeograficFactory factory){
        super(false); // Minimize this function
        this.puntos = puntos;
        this.factory = factory;
        this.lx = max;
        this.cortes = new ArrayList<Integer>();
        List<T> finalList = new ArrayList<T >();
        List<T> testear = new ArrayList<T>();
        for(T x: this.puntos)
                testear.add(x);
        int co =0;
        while(!testear.isEmpty()){
                if(testear.size()<6){
                        co += testear.size();
                        this.cortes.add(co);
                        finalList.addAll(testear);
                        break;
                }else{
                        ConvexHull<T> x = new ConvexHull<T>();
```

```
                    x.calcularPoligonoConvexo(testear);
                    co += testear.size();
                    this.cortes.add(co);
                    finalList.addAll(testear);
                    testear = x.getPuntosEliminados();
                }
        }
        this.puntos = finalList;
}

public double evaluate(double position[]) {
        List<DataPoint> newList = new ArrayList<DataPoint>();
        int init=0;
        for(Integer corte: this.cortes){
                Map<Double, List<Integer>> t =
                new HashMap<Double, List<Integer>>();
                double positionAux[] = new double[corte-init];
                int posix = 0;
                for(int i=init; i<corte;i++){
                        if(!t.containsKey(position[i]))
                        t.put(position[i], new ArrayList<Integer>());
                        t.get(position[i]).add(i);
                        positionAux[posix++] = position[i];
                }
                Arrays.sort(positionAux);
                for(Map.Entry<Double, List<Integer>> entry: t.entrySet())
                        for(Integer ix:entry.getValue())
                                newList.add(this.puntos.get(ix));
                init = corte;
        }
        List<DataPoint> newList2 = new ArrayList<DataPoint>(newList);
        List<Cluster> clustersFinales = new ArrayList<Cluster>();
        Cluster auxCluster=null;
        int i=0;
        while(!newList2.isEmpty()){
                if(auxCluster==null){
                        auxCluster = new Cluster(String.valueOf(i+1));
                        auxCluster.mDataPoints = new ArrayList<DataPoint>();
```

```
                        Centroid centroid =
                                puntos.get(0).getFactory().
                                createNewVertexCentroide(
                                        newList2.get(0).getX(),
                                        newList2.get(0).getY());
                                auxCluster.setCentroid(centroid);
                                clustersFinales.add(auxCluster);
                        }
                Collections.sort(newList2,
                new SortByDistance(auxCluster.getCentroid(), factory));
                while(auxCluster.getNumDataPoints()<lx && !newList2.isEmpty()){
                        DataPoint p=newList2.remove(0);
                        auxCluster.addDataPoint(p);
                        newList.remove(p);
                }
                newList2 = newList;
                auxCluster=null;
                }
        }
        return super.getMetricValue(clustersFinales, factory);
}
```

Bibliografía

[1] Peyman Afshani, Jérémy Barbay, and Timothy M. Chan. Instance-optimal geometric algorithms. In *Proceedings of the 2009 50th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '09, pages 129–138, Washington, DC, USA, 2009. IEEE Computer Society.

[2] Selim G. Akl and Godfried T. Toussaint. A fast convex hull algorithm. *Information Processing Letters*, 7(5):219 – 222, 1978.

[3] Maher M. Atwah, Johnnie W. Baker, Selim Akl, and Johnnie W. Baker Selim Akl. An associative implementation of classical convex hull algorithms. In *Proceedings of Eighth IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 435–438, 1996.

[4] Arindam Banerjee and Joydeep Ghosh. On scaling up balanced clustering algorithms. In *In Proceedings of the SIAM International Conference on Data Mining*, pages 333–349, 2002.

[5] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Trans. Math. Softw.*, 22(4):469–483, December 1996.

[6] P. S. Bradley, K. P. Bennett, and A. Demiriz. Constrained k-means clustering. Technical report, MSR-TR-2000-65, Microsoft Research, 2000.

[7] David Bremner. Incremental convex hull algorithms are not output sensitive. In *Proceedings of the 7th International Symposium on Algorithms and Computation*, ISAAC '96, pages 26–35, London, UK, UK, 1996. Springer-Verlag.

[8] Michael Clerc and James Kennedy. The particle swarm - explosion, stability, and convergence in a multidimensional complex space. *Evolutionary Computation, IEEE Transactions on*, 6(1):58 –73, feb 2002.

[9] Ian Davidson and S. S. Ravi. Agglomerative hierarchical clustering with constraints: Theoretical and empirical results. In *Lecture notes in computer science*, pages 59–70. Springer, 2005.

[10] Hichem Frigui and Raghu Krishnapuram. A robust competitive clustering algorithm with applications in computer vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21:450–465, 1998.

[11] Deon Garrett, Joseph Vannucci, Rodrigo Silva, Dipankar Dasgupta, and James Simien. Genetic algorithms for the sailor assignment problem. In *Proceedings of the 2005 conference on Genetic and evolutionary computation*, GECCO '05, pages 1921–1928, New York, NY, USA, 2005. ACM.

[12] Ruhan He, Weibin Xu, Jiaxia Sun, and Bingqiao Zu. Balanced k-means algorithm for partitioning areas in large-scale vehicle routing problem. In *Intelligent Information Technology Application, 2009. IITA 2009. Third International Symposium on*, volume 3, pages 87 –90, nov 2009.

[13] Kazuo Iwama, Shuichi Miyazaki, and Naoya Yamauchi. A 1.875: approximation algorithm for the stable marriage problem. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '07, pages 288–297, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[14] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Comput. Surv.*, 31(3):264–323, September 1999.

[15] Kennedy James and Eberhart R. Particle swarm optimization. In *Proceedings of IEEE International conference on Neural Networks*, pages 1942–1948, 1995.

[16] Yoshinobu Kawahara, Kiyohito Nagano, and Yoshio Okamoto. Balanced clustering via discrete dc programming. Technical Report TR09-0001, Tokyo Institute of Technology, 2009.

[17] James Kennedy. The behavior of particles. In *Proceedings of the 7th International Conference on Evolutionary Programming VII*, EP '98, pages 581–589, London, UK, 1998. Springer-Verlag.

[18] David G Kirkpatrick and Raimund Seidel. The ultimate planar convex hull algorithm. *SIAM J. Comput.*, 15(1):287–299, February 1986.

[19] H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1-2):83–97, 1955.

[20] Yee Leung, Jiang-She Zhang, and Zong-Ben Xu. Clustering by scale-space filtering. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22:1396–1410, 2000.

[21] James MacQueen. Some methods of classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297, 1967.

[22] Mahajan Meena, Nimbhorkar Prajakta, and Varadarajan Kasturi. The planar k-means problem is np-hard. In *Proceedings of the 3rd International Workshop on Algorithms and Computation*, WALCOM '09, pages 274–285, Berlin, Heidelberg, 2009. Springer-Verlag.

[23] Mahamed Omran, Ayed Salman, and Andries Engelbrecht. Dynamic clustering using particle swarm optimization with application in image segmentation. *Pattern Analysis and Applications*, 8:332–344, 2006.

[24] Ender Ozcan and Chilukuri K. Mohan. Particle swarm optimization: surfing the waves. In *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on*, volume 3, page 1944 Vol. 3, 1999.

[25] Konstantinos Parsopoulos and Michael Vrahatis. *Particle Swarm Optimization and intelligence: advances and applications*. ICI GLOBAL, 2010.

[26] Ioan Cristian Trelea. The particle swarm optimization algorithm: convergence analysis and parameter selection. *Information Processing Letters*, 85(6):317 – 325, 2003.

[27] Kiri Wagstaff, Claire Cardie, Seth Rogers, and Stefan Schroedl. Constrained k-means clustering with background knowledge. In *Proceedings of 18th International Conference on Machine Learning (ICML-01)*, pages 577–584. Morgan Kaufmann, 2001.