Algoritmo de Optimización por Enjambre de Partículas para el Problema del Viajante de Comercio

Labayen, Franco – labayen_francko@hotmail.com
Wals Ochoa, Lucas German – walslucas@gmail.com

Índice

1. Introducción

En el presente trabajo estudiaremos e implementaremos una de las tantas metaheuristicas basadas en población que existen, la Optimización por Enjambre de Partículas (PSO).

Primero empezaremos introduciendo las familias de algoritmos a la cual pertenece, lo que nos permitirá ir obteniendo una idea general de su funcionamiento y nos presentara algunos conceptos claves que intervienen en el algoritmo. Una vez finalizado, nos meteremos de lleno con PSO. Veremos cómo se compone la estructura del algoritmo y cómo funciona el mismo.

Haremos una breve presentación del problema a tratar, denominado Problema del Viajante de Comercio, explicando en que consiste y las variantes que existen del mismo. Luego, presentaremos la adaptación del algoritmo PSO, necesaria para hacerlo compatible al problema, donde mostraremos todos los cambios realizados a la versión continua del algoritmo para transformarlo en una versión discreta.

Se hará una breve explicación sobre el diseño del algoritmo, realizado en el lenguaje C, y luego se pasará a mostrar los experimentos realizados y los resultados obtenidos del mismo.

Finalmente, hablaremos de las conclusiones obtenidas a lo largo de la realización de este proyecto.

2. Metaheuristicas Basadas en Población

Las metaheurísticas basadas en población [1] comienzan con una población inicial de soluciones, e iterativamente van generando una nueva población que reemplazara a la población actual. La mayoría de los algoritmos de esta categoría están inspirados en la naturaleza.

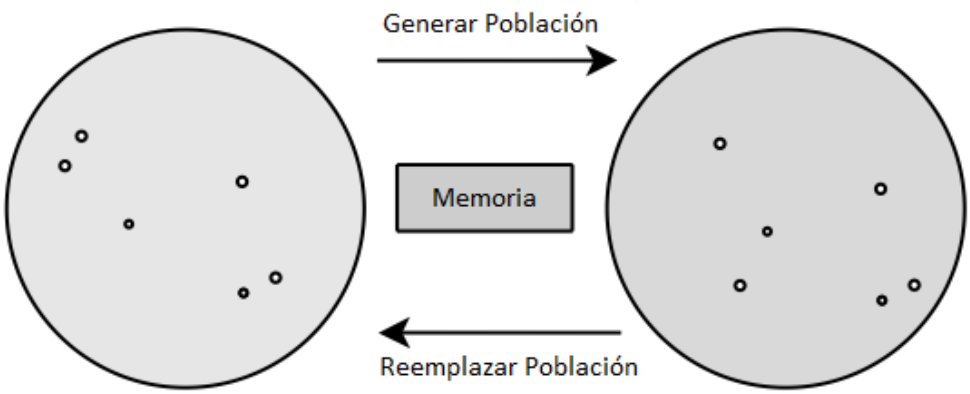

Figura 1. Principios de las Metaheuristicas Basadas en Población

A grandes rasgos, estas metaheuristicas consisten de dos fases, las cuales se pueden apreciar en la Figura 1, que se irán alternando en cada iteración hasta satisfacer un cierto criterio de

parada. También se puede poseer, o no; una "memoria compartida", la cual representa el conjunto de información extraída y memorizada durante la búsqueda. Lo que contenga esta memoria dependerá de la metaheuristica utilizada.

Primero tenemos una fase de "generación" en la cual se genera una nueva población de soluciones. Como se puede observar en la Figura 2, tenemos dos estrategias de generación posible:

- Basadas en Evolución (Evolution based): En esta categoría, las soluciones que componen la población son seleccionadas y reproducidas usando operadores de variación, como por ejemplo, la mutación.
- Basadas en Pizarrón (Blackboard based): Aquí, las soluciones participan para construir la memoria compartida, la cual será la principal entrada para la generación de la nueva población de soluciones.

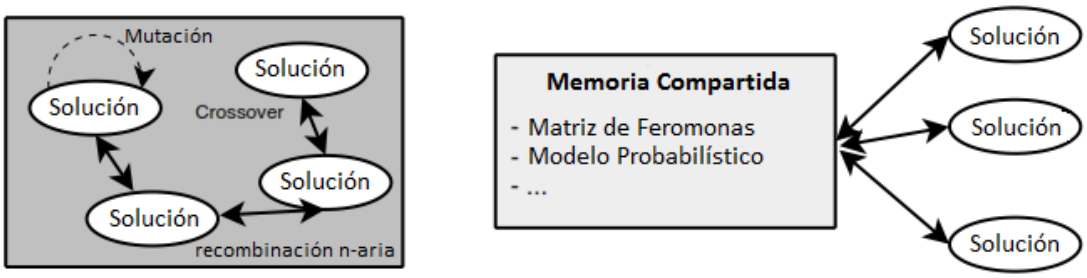

Figura 2. Estrategias Evolutivas (izquierda) contra Basadas en Pizarrón (derecha)

La segunda y última fase de estos algoritmos consiste en la "selección" de nuevas soluciones a partir de la unión de la población actual con la generada. La estrategia más común consiste en seleccionar la población generada como la nueva población con la cual trabajar. Otras estrategias pueden basarse en el "elitismo", es decir, elegir las mejores soluciones de la unión de ambas poblaciones.

Conociendo ya ambas fases y que es lo que se hace en cada una de ellas, nos resta conocer dos conceptos fundamentales para las metaheuristicas basadas en población, que son la determinación de la población inicial y la definición del criterio de parada.

2.1. Población Inicial

Determinar cuál será la población inicial es un paso crucial en la eficacia y la eficiencia del algoritmo. El criterio principal que se debe de tener en cuenta es la diversificación de la población, ya que si esta no se encuentra bien dispersa puede llevar a la metaheuristica a una rápida convergencia.

Las estrategias para la generación de la población se pueden clasificar en cuatro categorías:

- Generación Aleatoria: Como el nombre indica, la población inicial es generada aleatoriamente, y eso se puede hacer utilizando números seudo-aleatorios o una secuencia de números cuasi-aleatorios.

- ❖ Diversificación Secuencial: Las soluciones se generan en secuencia, de forma tal de que se optimice la diversidad, distribuyendo la población uniformemente en el espacio de búsqueda.
- ❖ Diversificación Paralela: La población se genera de forma independiente y en paralelo. Una forma de hacerlo es dividir el espacio de búsqueda en varios bloques, e ir generando soluciones en cada uno de ellos. (hay dos formas más que no se si nombrarlas)
- ❖ Inicialización Heurística: Se trata de usar alguna heurística para generar la población.

A continuación se muestra en la Tabla 1 un análisis de cada estrategia de acuerdo a la diversidad, el costo computacional y la calidad de las soluciones generadas:

Tabla 1. Análisis de las Diferentes Estrategias de Inicialización

Estrategia	Diversidad	Costo Computacional	Calidad de las Soluciones Iniciales
Seudo-Aleatoria	++	+++	+
Cuasi-Aleatoria	+++	+++	+
Diversificación Secuencial	++++	++	+
Diversificación Paralela	++++	+++	+
Heurística	+	+	+++
La evaluación de cada apartado mejora cuanto mayor cantidad de signos “más” (+) posea.			

2.2. Criterio de Parada

Se pueden usar muchos criterios basados en la evolución de la población. Algunos de ellos son similares a aquellos diseñados para las metaheuristicas basadas en una solución:

- ❖ Método Estático: El fin de la búsqueda puede conocerse a priori. Por ejemplo, uno puede usar un número fijo de iteraciones, o un límite en el consumo de recursos.
- ❖ Método Adaptativo: El fin de la búsqueda no puede ser conocido a priori. Por ejemplo, podemos usar un número fijo de iteraciones sin mejora, es decir, iterar hasta que sea alcanzada una solución satisfactoria.

Otros criterios son específicos de las metaheuristicas basadas en población. Estos generalmente están basados en alguna estadística sobre la población actual o la evolución de la población, principalmente, la diversidad de la población. Este criterio se encarga del estancamiento que puede sufrir la población, deteniendo el algoritmo cuando la medida de diversidad de la población cae por debajo de un cierto umbral, bajo el cual continuar con la ejecución del algoritmo es inútil.

3. Inteligencia de Enjambre

Esta categoría abarca todos aquellos algoritmos inspirados en el comportamiento colectivo de especies como las hormigas, abejas, avispas, termitas, pescados y aves.

Las características principales de estos algoritmos son la utilización de agentes simples llamados "partículas", las cuales cooperan a través de un medio de comunicación indirecto, conocido como "enjambre", y se van moviendo por el espacio de búsqueda.
Los algoritmos más exitosos de esta categoría son el de Colonia de Hormigas y el de Optimización por Enjambre de Partículas, el cual es empleado en este trabajo.

4. Optimización por Enjambre de Partículas (PSO)

Abreviado PSO[1], fue desarrollado por el psicólogo-sociólogo James Kennedy y por el ingeniero electrónico Russel Eberhart en el año 1995. Originalmente fue diseñado para resolver problemas de optimización continuos, siendo capaz de minimizar cualquier función objetivo del tipo $f: R^n \rightarrow R$, a la cual de ahora en más denominaremos *f*.

4.1. Estructura

El algoritmo consiste de un proceso iterativo que va trabajando sobre un enjambre de partículas. Generalmente, la partícula está compuesta por los siguientes elementos:

- Un vector $x \in R^n$, que almacena la posición actual en el espacio de búsqueda, y representa una posible solución al problema.
- Un vector $pBest \in R^n$, que almacena la posición de la mejor solución por la que se ha desplazado la partícula.
- Un vector $v \in R^n$, que representa la rapidez y dirección en la cual se trasladará la partícula.
- Un valor $fitness\_x \in R$, que representa el valor devuelto por *f* evaluada en la posición actual.
- Un valor $fitness\_pBest \in R$, que representa el valor devuelto por f evaluada en la mejor solución encontrada por la partícula.

Por su parte, el enjambre también debe mantener los siguientes valores:

- Un vector $gBest \in R^n$, que almacena la posición de la mejor solución por la que se ha desplazado todo el conjunto de partículas.
- Un valor $fitness\_gBest \in R$, que representa el valor devuelto por f evaluada en la mejor solución encontrada por el conjunto de partículas.

4.2. Funcionamiento

En una primera fase, el enjambre se inicializa generando las posiciones y las velocidades iniciales de cada partícula. Las mismas se pueden inicializar de forma aleatoria, utilizando alguna heurística que distribuya uniformemente las partículas o, mediante una combinación

[1] PSO, del inglés, Particle Swam Optimization

de ambas. Luego, se calcula el valor fitness de cada una y se actualizan las variables $fitness\_x$ y $fitness\_pBest$. Al mismo tiempo se deben renovar los valores de $gBest$ y $fitness\_gBest$, con el mejor de todos los resultados encontrados hasta el momento.

Una vez finalizada la inicialización del enjambre, las partículas deben desplazarse dentro del espacio de búsqueda. Para esto, se genera un ciclo donde por cada iteración todas actualizan su velocidad y se "mueven" simplemente sumando su posición, obtenida en un tiempo anterior, a la velocidad recién calculada. Una vez deducida la nueva situación de todas las partículas, se calcula el valor de $fitness_x$ para cada una de ellas. Si el nuevo resultado encontrado es el mejor para una partícula se actualizan los valores de $pBest$ y $fitness\_pBest$ correspondientes. Además, si es el mejor del enjambre, también se renovarán las variables $gBest$ y $fitness\_gBest$.

La forma de modificar las velocidades y posiciones viene dada por un modelo matemático que representa el corazón del algoritmo y que explicaremos en la siguiente sección.

4.3. Ecuaciones

Operador actualización de velocidad:

$$\boldsymbol{v_i^t} = \omega \cdot v_i^{t-1} + \varphi_1 \cdot rand_1 \cdot \left(pBest_i - x_i^{t-1}\right) + \varphi_2 \cdot rand_2 \cdot \left(gBest - x_i^{t-1}\right)$$

Operador Movimiento:

$$\boldsymbol{x_i^t} = x_i^{t-1} + \boldsymbol{v_i^t}$$

$\boldsymbol{t}$**:** Representa el tiempo de la iteración.
$\boldsymbol{v_i}$: Velocidad de la partícula i.
$\boldsymbol{\omega}$: Factor de inercia.
$\boldsymbol{\varphi_1}\ y\ \boldsymbol{\varphi_2}$: Son ratios de aprendizaje (pesos) que controlan los componentes cognitivo y social respectivamente.
$\boldsymbol{rand_1}\ y\ \boldsymbol{rand_2}$: Números aleatorios entre 0 y 1.
$\boldsymbol{x_i}$: Posición de la partícula i.
$\boldsymbol{pBest_i}$: Mejor posición (solución) encontrada por la partícula i hasta el momento.
$\boldsymbol{gBest}$: Representa el mejor valor encontrado por el enjambre.

El vector velocidad de cada partícula es modificado en cada iteración utilizando la velocidad anterior, un componente cognitivo y un componente social. La unidad cognitiva está modelada por el factor $\varphi_1 \cdot rand_1 \cdot \left(pBest_i - x_i^{t-1}\right)$ y representa la distancia entre la posición actual y la mejor conocida por esa partícula, es decir, la decisión que tomará la partícula influenciada por su propia experiencia a lo largo de su vida. El componente social está modelado por $\varphi_2 \cdot rand_2 \cdot \left(gBest - x_i^{t-1}\right)$ y representa la distancia entre la posición actual y la mejor posición del enjambre, es decir, la decisión que tomará la partícula según la influencia que el resto del conjunto ejerce sobre ella.

Por otro lado, el factor de inercia, nos permite controlar el hecho de que una velocidad muy grande puede producir grandes saltos en los movimientos de las partículas, fomentando una exploración global; o por el contrario si la velocidad es muy pequeña se podrían producir saltos pequeños que estimulen una búsqueda local. Existen estudios que evalúan la posibilidad de que este factor sea dinámico, aunque no los trataremos en este proyecto.

4.4. Clasificación

Desde el punto de vista del componente social [2], se pueden distinguir dos importantes variaciones de este algoritmo: PSO local y PSO global.

- PSO local: Consiste en dividir el enjambre en subgrupos mutuamente excluyentes de partículas que compartan ciertas características. Generalmente se usa el principio de distancia entre puntos para lograr la separación. Este concepto, a diferencia de lo desarrollado hasta el momento, guía las partículas, no solo por un único mejor global, sino por varios correspondientes a cada uno de los grupos o vecindarios a los que pertenecen las partículas. Un miembro puede pasar a formar parte de otro grupo si los cambios que sufre durante la iteración lo llevan a cumplir las condiciones de otro conjunto. Este punto de vista no será expandido en este proyecto.
- PSO global: Consiste en concentrar la búsqueda en la experiencia propia de cada partícula y en un único mejor global. Este concepto es el que hasta el momento hemos desarrollado, y en el que basaremos el resto del informe.

A continuación mostraremos un pseudocódigo, tomando como punto de partida esta idea de implementación.

4.5. Pseudocódigo del Algoritmo PSO

Teniendo en consideración los apartados anteriores, estamos en condiciones de presentar el pseudocódigo correspondiente a la implementación de un PSO Global.

Algoritmo 1. PSO de tipo Global.

Para cada partícula i = 1, …, hasta el total de partículas:

- Inicializar la posición $x_i \in R^n$ aleatoriamente.
- Inicializar la velocidad $v_i \in R^n$ aleatoriamente.
- Inicializar la posición $pBest_i \in R^n$ con el valor de x_i: $pBest_i \leftarrow x_i$
- Inicializar $fitness\_x_i \in R$ de la siguiente forma: $fitness\_x_i \leftarrow f(x_i)$
- Inicializar $fitness\_pBest_i \in R$ de la siguiente forma: $fitness\_pBest_i \leftarrow fitness\_x_i$
- Si ($fitness\_pBest_i < fitness\_gBest$) entonces:
  - Actualizar el mejor resultado global: $fitness\_gBest \leftarrow fitness\_pBest_i$
  - Actualizar la mejor posición global: $gBest \leftarrow pBest_i$

Mientras no se alcance el límite máximo de iteraciones, o no se cumpla el criterio de parada:

- Para cada partícula i = 1, …, hasta el total de partículas:
  - Actualizar la velocidad v_i: $v_i \leftarrow \omega \cdot v_i + \varphi_1 \cdot rand_1 \cdot (pBest_i - x_i) + \varphi_2 \cdot rand_2 \cdot (gBest - x_i)$
  - Trasladar la posición x_i: $x_i \leftarrow x_i + v_i$

- Para cada partícula i = 1, …, hasta el total de partículas:
  - Reevaluar $fitness\_x_i$: $fitness\_x_i \leftarrow f(x_i)$
  - Si ($fitness\_x_i < fitness\_pBest_i$) entonces:
    - Actualizar el mejor resultado local: $fitness\_pBest_i \leftarrow fitness\_x_i$
    - Actualizar la mejor posición local: $pBest_i$: $pBest_i \leftarrow x_i$
    - Si ($fitness\_pBest_i < fitness\_gBest$) entonces:
      - Actualizar el mejor resultado global: $fitness\_gBest \leftarrow fitness\_pBest_i$
      - Actualizar la mejor posición global: $gBest \leftarrow pBest_i$

Devolver $gBest$ como la mejor solución.

5. Presentación del Problema

El problema que se nos dio a resolver es el Problema del Viajante o TSP[2], que aborda lo siguiente:

Un viajante quiere visitar N ciudades una y sólo una vez cada una, empezando por una cualquiera de ellas y regresando al mismo lugar del que partió.
Supongamos que conoce la distancia entre cualquier par de ciudades. ¿De qué forma debe hacer el recorrido si pretende minimizar la distancia total?

El anterior párrafo nos indica que este problema se basa en encontrar una combinación de ciudades de manera tal que, la suma de los costos de ir de una ciudad a otra, sin pasar dos veces por la misma, sea mínimo. Ese costo viene dado por una matriz cuadrada, a la cual llamaremos *D*, en la cual el elemento d_{ij} representa el costo de ir de la ciudad *i* a la ciudad *j*.

Existen algunos casos particulares de este problema, que son:

- Problema del Viajante Simétrico o STSP[3]: Es aquel en el que la matriz es simétrica, es decir, $d_{ij} = d_{ji} \; \forall \; i, j \in D$
- Problema del Viajante Asimétrico o ASTP[4]: Aquel en el que la matriz es anti-simétrica, es decir, $d_{ij} \neq d_{ji} \; \forall \; i, j \in D$

La forma más directa de encontrar la solución es aplicar la fuerza bruta, es decir, evaluar todas las posibles combinaciones de recorridos y quedarse con aquella cuyo costo es el menor. El problema reside en el número de posibles combinaciones que viene dado por el factorial de la

[2] TSP, del inglés, Travelling Salesman Problem
[3] STSP, del inglés, Symmetric TSP
[4] ATSP, del inglés, Asymmetric TSP

cantidad de ciudades, y esto hace que la solución por fuerza bruta sea impracticable para una cantidad moderada de ciudades.

El TSP está entre los problemas denominados *NP-completos*, es decir, los problemas que no se pueden resolver en tiempo polinomial en función del tamaño de entrada (en este caso, la entrada seria N, la cantidad de ciudades que el viajante debe recorrer)

Para la experimentación, se tomaran cuatro instancias[5] correspondientes a la versión asimétrica que serán:

- Br17.atsp (17 ciudades)
- Ftv33.atsp (34 ciudades)
- Ft53.atsp (53 ciudades)
- Ft70.atsp (70 ciudades)

6. Adaptando el Algoritmo al Problema: PSO con Permutación de Enteros (PSOP)

Como hemos visto, la configuración básica del algoritmo PSO no tiene un diseño que nos permita resolver problemas de dominios discretos. Sin embargo, se han realizado investigaciones que presentan diferentes alternativas que lo habilitan a desempañarse en estos ambientes. Específicamente, y a efectos de resolver el problema introducido en la sección anterior, desarrollaremos una de las posibles adaptaciones, la cual llamaremos PSO para permutación de enteros (PSOP), basándonos en uno de los artículos de Maurice Clerc [3].

En forma muy general, la tarea consiste en restringir el dominio por el cual las partículas pueden desplazarse, de tal manera que cumpla determinadas condiciones y se transforme en discreto. Es fácil deducir que este concepto requiere redefinir la estructura de la partícula, así como también las operaciones de actualización que se aplican sobre la misma. Es importante que retomemos lo analizado (Ver apartado 4) y estudiemos los cambios pertinentes.

6.1. Cambios Estructurales

Las posiciones deben interpretarse como un conjunto de números enteros, mientras que las velocidades se pueden ver como una o más permutaciones que se aplican a una posición.

- Posiciones: La posiciones de una partícula estarán formadas por una lista de N posibles valores enteros sin que existan repeticiones u omisiones. Desde el punto de vista del problema del viajante, cada posición representará un posible tour y el valor N

[5] Estas instancias pueden ser descargadas desde el sitio: http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/atsp/

corresponderá a la cantidad de ciudades involucradas en la instancia del problema. Un ejemplo sencillo seria:

$$x = [5,4,3,1,6,2]$$

- Velocidades: Las velocidades estarán representadas por una lista de listas, donde cada sub-lista representará un par de enteros $[i \rightarrow j]$. Cada uno de estos pares simbolizará un intercambio o permutación a realizar sobre los elementos de una posición. Explicaremos su aplicación en el Apartado 6.2.2 junto con el nuevo operador de movimiento. Un ejemplo es el siguiente:

$$v = [[9 \rightarrow 1], [4 \rightarrow 6], [1 \rightarrow 5]]$$

6.2. Cambios en las Ecuaciones

Es imprescindible reconfigurar las fórmulas del algoritmo PSO para que permitan implementar un ambiente discreto y que mantengan las condiciones del mismo.

6.2.1 Adaptando el Operador Actualización de Velocidad

La actualización de la velocidad mantiene los mismos principios y el formato anterior, aunque es necesario describir nuevos operadores de suma, diferencia y multiplicación adaptados a la permutación de enteros.

$$\boldsymbol{v_i^t} = \omega \otimes v_i^{t-1} \circ \varphi_1 \otimes \left(pBest_i \; \theta \; x_i^{t-1}\right) \circ \varphi_2 \otimes \left(gBest \; \theta \; x_i^{t-1}\right)$$

- Operador Resta de Posiciones (θ): Al restar dos posiciones el resultado es una lista velocidad, es decir, una lista de pares. Un par i está formado por el i-ésimo valor del segundo operando y el i-ésimo valor de primer operando de la resta. Por ejemplo:

$$[1,2,3,4,5,6] \; \theta \; [5,2,4,6,3,1]$$
$$=$$
$$[[5 \rightarrow 1], [2 \rightarrow 2], [4 \rightarrow 3], [6 \rightarrow 4], [3 \rightarrow 5], [1 \rightarrow 6]]$$

- Operador Suma de Velocidades (∘): La suma de dos velocidades consiste en ir "solapando" los pares de la primera con la segunda, de manera que se obtenga una nueva lista de listas. Para que se produzca una nueva pareja, deben coincidir los valores final e inicial de las sub-listas involucradas, es decir, los pares $[i \rightarrow j]$ y $[j \rightarrow k]$ se solapan formando el elemento $[i \rightarrow k]$. Si un par del primer operador no combina con ninguno del segundo, el resultado será el primer par. Se debe tener en cuenta que los compuestos de la segunda lista no pueden ser utilizados más de una vez, y que la longitud de la lista resultante será siempre igual a la del primer miembro de la suma. Un ejemplo sencillo es el siguiente:

$$[[5 \to 1], [2 \to 2]]$$
$$\circ$$
$$[[1 \to 1], [5 \to 1], [2 \to 4]]$$
$$=$$
$$[[5 \to 1], [2 \to 4]]$$

- Operador Producto Coeficiente Velocidad ($\otimes$): Mediante el producto coeficiente velocidad se realiza la multiplicación de un coeficiente real φ y una lista velocidad. Se pueden dar los siguientes casos:
  - Si φ se encuentra entre [0, 1] , se obtiene φ' = rand(0, 1) y se compara:
    - $\varphi' < \varphi \Rightarrow [i \to j] = [i \to i]$
    - $\varphi' \geq \varphi \Rightarrow [i \to j] = [i \to j]$

  - Si $\varphi > 1$, tal que $\varphi = k + \varphi'$, con k entero y $\varphi' < 1$, se realiza la suma de la velocidad k veces, y al resultado obtenido se le suma el producto del coeficiente φ' por velocidad:

$$\underbrace{v \circ v \circ \ldots \circ v}_{k\ veces} \circ \varphi' \otimes v$$

Por ejemplo, si multiplicamos:

$$0.5 \otimes [[1 \to 3], [2 \to 5], [4 \to 4], [3 \to 2]]$$

Para la secuencia de valores aleatorios (φ') 0.2, 0.8, 0.5 y 0.3 se obtiene como resultado:

$$[[1 \to 1], [2 \to 5], [4 \to 4], [3 \to 3]]$$

6.2.2 Adaptando el Operador Movimiento

Se mantiene la idea de que el movimiento se realiza sumando o aplicando una velocidad a una posición, sin embargo es necesario redefinir la suma para que sea compatible con la nueva estructura.

$$\boldsymbol{x_i^t = x_i^{t-1} \oplus v_i^t}$$

- Operador Suma de Posición con Velocidad ($\oplus$): Mediante este operador, se realiza el proceso de permutar los valores de la posición de una partícula para generar una nueva posición. El proceso consiste en tomar en orden los pares $[i \to j]$ de una lista velocidad, e ir aplicando una a una las permutaciones de los elementos i y j en el vector posición. A continuación planteamos un ejemplo:

$$x = [2,4,3,6,1,5]$$
$$\oplus$$
$$v = [[3 \to 1], [6 \to 6], [1 \to 5]]$$

Primero, al aplicar $[3 \rightarrow 1]$ obtenemos $x = [2,4,1,6,3,5]$
Luego, al aplicar $[6 \rightarrow 6]$ el vector no sufre cambios.
Finalmente, al aplicar $[1 \rightarrow 5]$ obtenemos $x = [2,4,5,6,3,1]$

6.2.3 Optimización de las ecuaciones

El objetivo es tomar el operador de velocidad para permutación de enteros y perfeccionarlo para lograr un mejor desempeño en la resolución del problema del viajante.

De acuerdo al artículo de Clerc [3], hemos simplificando la ecuación de velocidad (Ver Apartado 4.3) considerando $\varphi_1 = \varphi_2$. Definimos una posición intermedia y, a partir de esta, una nueva fórmula abreviada. Finalmente, presentamos el sistema matemático que utilizaremos para resolver las instancias de los problemas ATSP:

Posición intermedia:
$\mathrm{pInt}_i = pBest_i \oplus \frac{1}{2} \otimes (\mathrm{gBest}\ \theta\ pBest_i)$
Operador Actualización de Velocidad:
$\boldsymbol{v_i^t} = c1 \otimes v_i^{t-1} \circ c2 \otimes (pInt_i\ \theta\ x_i^{t-1})$ c1 y c2 escalares
Operador Movimiento:
$\boldsymbol{x_i^t} = x_i^{t-1} \oplus \boldsymbol{v_i^t}$

6.3. Pseudocódigo del algoritmo PSOP

Una vez desarrolladas todas las modificaciones necesarias, presentamos el pseudocódigo correspondiente al nuevo algoritmo.

Algoritmo 2. PSO adaptado al Problema del Viajante

Para cada partícula i = 1, …, hasta el total de partículas:

- Inicializar la posición x_i con un tour generado aleatoriamente.
- Inicializar la velocidad de la partícula $v_i$mediante la resta de dos posiciones aleatorias.
- Inicializar la posición $pBest_i$ con el valor de x_i: $pBest_i \leftarrow x_i$
- Inicializar $fitness\_x_i$ de la siguiente forma: $fitness\_x_i \leftarrow f(x_i)$
- Inicializar $fitness\_pBest_i$ de la siguiente forma: $fitness\_pBest_i \leftarrow fitness\_x_i$
- Si ($fitness\_pBest_i < fitness\_gBest$) entonces:
  - Actualizar el mejor resultado global: $fitness\_gBest \leftarrow fitness\_pBest_i$

- Actualizar la mejor posición global: $gBest \leftarrow pBest_i$

Mientras no se alcance el límite máximo de iteraciones, o no se cumpla el criterio de parada:

- Para cada partícula i = 1, …, hasta el total de partículas:
  - Actualizar la velocidad: $\boldsymbol{v_i^t} = c1 \otimes v_i^{t-1} \circ \text{c2} \otimes \left(pInt_i\ \theta\ x_i^{t-1}\right)$
  - Actualizar la posición: $\boldsymbol{x_i^t} = x_i^{t-1} \oplus \boldsymbol{v_i^t}$

- Para cada partícula i = 1, …, hasta el total de partículas:
  - Reevaluar $fitness\_x_i$: $fitness\_x_i \leftarrow f(x_i)$
  - Si ($fitness\_x_i < fitness\_pBest_i$) entonces:
    - Actualizar el mejor resultado local: $fitness\_pBest_i \leftarrow fitness\_x_i$
    - Actualizar la mejor posición local: $pBest_i$: $pBest_i \leftarrow x_i$
    - Si ($fitness\_pBest_i < fitness\_gBest$) entonces:
      - Actualizar el mejor resultado global: $fitness_{gBest} \leftarrow fitness\_pBest_i$
      - Actualizar la mejor posición global: $gBest \leftarrow pBest_i$

Devolver $gBest$ como la mejor solución.

7. Diseño del Algoritmo

A fines de obtener mejores resultados, principalmente en cuanto al tiempo, decidimos implementar el algoritmo en lenguaje C. Nombraremos y explicaremos la composición, es decir, las carpetas y archivos involucrados en el desarrollo; así como también algunas consideraciones a tener en cuenta.

El diseño final consta de los siguientes elementos:

- Una carpeta “Instancias” que contiene las cuatro instancias del problema ATSP nombradas en el Apartado 5. Estos archivos sufrieron una pequeña alteración, a motivo de poder ser “cargados” más fácilmente en el programa. El cambio consistió en la “estandarización” de la cantidad de caracteres en las filas anteriores a la matriz de costos.
- Una carpeta “Resultados” donde se almacenan las soluciones junto con los parámetros que dieron lugar a esas respuestas, y otros datos de interés. Un archivo, por ejemplo, “resultado34” guardará la información relacionada con la instancia del problema que implique el recorrido de 34 ciudades. El almacenamiento se realiza utilizando el formato CSV.
- Un archivo “algoritmoPSOP” donde se implementa el pseudocódigo nombrado y estudiado en el proyecto. (Ver Algoritmo 2)
- Un archivo “operadoresPSOP”, donde se implementan cada uno de los operadores propios del algoritmo PSOP (Ver apartados 6.2 y 6.3) y, a su vez, necesarios para poder realizar la actualización de las velocidades y el movimiento de las partículas.
- Un archivo “operadoresArray”, en el cual se implementan funciones básicas para el trabajo simplificado de vectores, tales como copiar un vector, generar un vector aleatorio, etc.

- Un archivo “instanciasPSOP”, donde se implementan las tareas que leen y “cargan”, como una matriz de costos, las instancias de los problemas ubicados en la carpeta “Instancias”.
- Un archivo “resultadosPSOP”, en el cual se implementan las tareas para guardar los resultados de una ejecución. Como dijimos anteriormente, se almacenan dentro de la carpeta “Resultados”.
- Un ejecutable “psop”, que corresponde a la compilación del archivo “algoritmoPSOP”. El binario está compilado para arquitecturas de 64 bits.

También se incluye un script bash llamado “automatizar”, que nos facilita la labor de ejecutar el algoritmo varias veces, con los mismos parámetros, para luego evaluar los resultados. El script está adaptado para aprovechar todos los procesadores que posea la máquina que lo ejecute, pudiendo realizar múltiples ejecuciones simultáneamente pero dejando un margen para poder realizar otras tareas en caso de que fuese necesario.

8. Experimentación y Resultados Obtenidos

La experimentación fue realizada en una computadora con las siguientes características:

- Procesador Intel® Core™ i7-3630QM de 2.4 GHz.
- Sistema Operativo de 64 bits, OpenSuse versión 12.3.

Se realizaron pruebas con cuatro configuraciones diferentes. Las mismas fueron elegidas para obtener los mejores resultados en una instancia y ver como se comportaba con el resto.

En este informe mostraremos, por cuestiones de espacio, únicamente las estadísticas obtenidas de analizar las treinta ejecuciones realizadas por instancia para cada configuración. De todas formas, todos los archivos CSV y las planillas de cálculo utilizadas serán adjuntados a este informe

8.1. Primer Configuración

Esta configuración fue pensada para optimizar lo más posible la instancia “br17.atsp”. Los parámetros utilizados fueron los siguientes:

- Cantidad de Partículas: 200.
- Cantidad Máxima de Iteraciones: 10000.
- $c1$: 0,4.
- $c2$: 0,1.

	PI[6]	PTS[6]	PE[6]	PTT[6]	PC[6]	MS[6]
Br17.atsp	569,47	0,63	881733,33	9,93	39,03	39
Ftv33.atsp	4831,73	15,11	12257108,33	30,58	1798,03	1574
Ft53.atsp	7132,77	46,77	17314308,33	64,93	11209,40	10219
Ft70.atsp	7935,83	85,08	14271325	106,54	48830,47	46952

8.2. Segunda Configuración

Esta configuración fue pensada para optimizar lo más posible la instancia "ftv33.atsp". Los parámetros utilizados fueron los siguientes:

- Cantidad de Partículas: 250.
- Cantidad Máxima de Iteraciones: 75000.
- $c1$: 0,1.
- $c2$: 0,4.

	PI[6]	PTS[6]	PE[6]	PTT[6]	PC[6]	MS[6]
Br17.atsp	3526,93	4,53	113893,33	98,88	39	39
Ftv33.atsp	49028,43	199,38	966346,67	298,99	1665,43	1491
Ft53.atsp	69257,23	593,33	1426553,33	642,72	10898,37	9425
Ft70.atsp	57085,30	781,36	1587166,67	1020,04	54075,47	47468

8.3. Tercer Configuración

Esta configuración fue pensada para optimizar lo más posible la instancia "ft53.atsp". Los parámetros utilizados fueron los siguientes:

- Cantidad de Partículas: 100.
- Cantidad Máxima de Iteraciones: 150000.
- $c1$: 0,2.
- $c2$: 1,8.

	PI[6]	PTS[6]	PE[6]	PTT[6]	PC[6]	MS[6]
Br17.atsp	3442,00	2,24	344200	94,91	39,17	39
Ftv33.atsp	80401,53	153,58	8040153,33	281,71	1733,73	1521
Ft53.atsp	118616,13	470,82	11861613,33	594,16	10857,33	9934
Ft70.atsp	132425,47	863,76	13242546,67	976,93	50525,20	46968

8.4. Cuarta Configuración

Esta configuración fue pensada para optimizar lo más posible la instancia "ft70.atsp". Los parámetros utilizados fueron los siguientes:

- Cantidad de Partículas: 300.

[6] PI: Promedio de Iteraciones en encontrar la mejor solución, PTS: Promedio de Tiempo en encontrar la mejor Solución, PE: Promedio de Evaluaciones hasta encontrar la mejor solución, PTT: Promedio de Tiempo Total de ejecución, PC: Promedio de Costo, MS: Mejor Solución encontrada.

- Cantidad Máxima de Iteraciones: 50000.
- $c1$: 0,25.
- $c2$: 0,25.

	PI[6]	PTS[6]	PE[6]	PTT[6]	PC[6]	MS[6]
Br17.atsp	578,17	0,90	173450	78,10	39	39
Ftv33.atsp	17767,60	85,15	5330280	231,85	1692,67	1464
Ft53.atsp	35092,17	351,80	10527650	497,08	10490,00	9274
Ft70.atsp	42237,03	700,53	12671110	826,27	47210,47	45108

9. Conclusión

Primero de todo, queremos aclarar que no se ha implementado el algoritmo totalmente de acuerdo a lo dicho por Clerc [3], concretamente la parte de la suma de velocidades, porque nos empeoraba los resultados, haciendo que las aproximaciones no fuesen buenas.

Como conclusiones del presente trabajo podemos destacar la importancia que tiene elegir una buena configuración para optimizar la instancia con la cual se está trabajando, resaltando el hecho de que la última configuración, la utilizada para obtener los mejores resultados para la instancia de 70 ciudades, también obtuvo muy buenos resultados para a aquellas instancias para las cuales no estaba pensado, en algunos casos llegando a superar los valores devuelto por la configuración orientada a ellas.

Con las experimentaciones, pudimos observar que el hecho de aumentar la cantidad máxima de iteraciones no es una muy buena opción para obtener mejores resultados, y que lo más recomendable seria aumentar la cantidad de partículas utilizadas antes que las iteraciones. También queremos mencionar el hecho de que, en la mayoría de los casos, valores altos de $c1$ y $c2$ producen aproximaciones muy malas.

10. Referencias

[1] El-Ghazali Talbi, *Metaheuristics: From Design to Optimization*. Wiley, 2009.

[2] José Manuel García Nieto, *Algoritmos Basados en Cúmulos de Partículas para la Resolución de Problemas Complejos*, 2006.

[3] Maurice Clerc, *Discrete Particle Swarm Optimization Illustrated by the Traveling Salesman Problem*, http://clerc.maurice.free.fr/pso/pso_tsp/Discrete_PSO_TSP.htm , 2000