Análisis, Implementación y Comparativas entre un Algoritmo Genético Tradicional, el Algoritmo Híbrido Genético-Taguchi y el Algoritmo Híbrido Genético-Taguchi Cooperativo-Coevolutivo para Problemas de Optimización Numérica Global

Cristhian Eduardo Fuertes Daza 201123259
Oscar Ivan Tigreros Bolaños 201326645
Director: Doctor Ángel de la Encarnacion Garcia Baños
Cali – Colombia Enero 2015
Universidad del Valle
Facultad de Ingenierías
Escuela de Ingeniería de Sistemas y Computación

Lista de Contenidos

Lista de Tablas

Lista de Figuras

Resumen

En el presente trabajo se plantea el desarrollo de un proyecto en torno a la pregunta : ¿Qué tan buenos son los algoritmos genéticos que utilizan el método Taguchi con respecto a los algoritmos genéticos que se enseñan en la Universidad del Valle para resolver problemas de optimización? Para dar respuesta a esta interrogante se analizan e implementan dos algoritmos genéticos que utilizan el método Taguchi y uno de los que se enseñan en la universidad. Finalmente se prueban los tres algoritmos con un *benchmark* de funciones de optimización y se analizan los resultados para determinar su desempeño computacional.

Palabras clave: algoritmos genéticos, problemas de optimización, método Taguchi, desempeño computacional

1. Introducción

Resolver problemas de optimización es una tarea que en el mundo actual es de vital importancia, especialmente en el campo de la ingeniería, debido a la existencia de sistemas que involucran muchas variables y en muchos casos múltiples objetivos. Naturalmente, estos problemas son computacionalmente difíciles de tratar y una de las herramientas usadas para resolverlos son los denominados algoritmos genéticos, que desde hace varias décadas han sido empleados en muchos sectores de la ingeniería.

Uno de los tantos algoritmos genéticos diseñados para resolver los problemas de optimización numérica global con muchas variables, es aquel que fue desarrollado por Yiu-Wing Leung y Yuping Wang en el 2001. En éste, se incluye el diseño experimental como parte del algoritmo genético para mejorarlo. Más adelante, en el 2004, Jinn-Tsong Tsai, Tung-Kuan Liu, y Jyh-Horng Chou desarrollaron un algoritmo basado en el de Leung y Wang que mejoraría el propuesto por ellos, ya que éste tenía un costo computacional bastante alto, y obtendría mejores resultados. Finalmente, una década después en 2014, S.S. Poorjandaghi. A. Afshar modificaría el algoritmo anterior para tener en cuenta la interacción entre las variables de las funciones.

El algoritmo diseñado por Tsai y colaboradores en el 2004, suscitó a muchos investigadores a crear variantes del mismo, y la más relevante fue la propuesta por Poorjandaghi en 2014. Entonces, para poder evaluar la efectividad de estos dos algoritmos, sería necesario un análisis y una comparación en el desempeño de ambos también frente a un algoritmo genético tradicional que no involucre técnicas de diseño de experimentos para determinar el dominio en cual cada uno de estos algoritmos funcionan mejor (como se afirma en el teorema de No Free Lunch).

2. Planteamiento y Definición del problema

El *Algoritmo Híbrido Genético-Taguchi* (*HTGA* por sus siglas en inglés**)** que fue introducido en el artículo homónimo por Jinn-Tsong Tsai, Tung-Kuan Liu y Jyh-Horng Chou en el año 2004 y que ha sido adaptado/modificado a través de los años subsecuentes, es un algoritmo que usa estrategias evolutivas y por eso cae dentro del campo de la Computación Evolutiva. Entre los algoritmos que han surgido a partir del *HTGA***,** quizás uno de los más relevantes es el *Algoritmo Híbrido Genético-Taguchi Cooperativo y Coevolutivo* (*CCHTGA* por sus siglas en inglés) ya que mejora el desempeño del algoritmo anterior.

De acuerdo a Tsai, Liu y Chou (2004), El *HTGA* sirve para encontrar los valores óptimos (máximos y mínimos globales) en funciones continuas multivariables con una convergencia más rápida que con algoritmos genéticos tradicionales. Además, incluye estrategias del campo de diseño de experimentos, en concreto, el método de Taguchi. Este es un algoritmo robusto y estadísticamente sólido para resolver problemas de optimización numérica.

De acuerdo a Poorjandaghi (2014), el *Algoritmo Híbrido Genético-Taguchi Cooperativo y Coevolutivo* (*CCHTGA* por sus siglas en inglés) es un algoritmo genético robusto que mejora el *HTGA* modificándolo para que funcione para problemas de dimensiones más grandes y que tiene en cuenta las interacciones entre variables usando una estrategia de divide y vencerás.

Estos dos algoritmos han demostrado tener un buen desempeño para la resolución de problemas de optimización, por eso es necesario analizar, implementar y comparar estos algoritmos para aumentar el conocimiento actual que se tiene sobre algoritmos genéticos en el ámbito académico.

Entonces, el propósito de este trabajo será responder a la pregunta: ¿Qué tan buenos son los algoritmos genéticos que utilizan el método Taguchi con respecto a los algoritmos genéticos que se enseñan en la Universidad del Valle para resolver problemas de optimización?

3. Justificación del problema

3.1 Justificación Académica

En la Escuela de Ingeniería de Sistemas y Computación de la Universidad del Valle se ofrece una asignatura electiva profesional llamada *Computación Evolutiva* en la que se enseñan temas relativos a estrategias evolutivas para solucionar problemas de computación, entre estas estrategias están los algoritmos genéticos. También, existe un laboratorio dentro de la misma escuela dedicado al estudio de estos temas conocido como *Laboratorio de Vida Artificial y Computación Evolutiva* (*EVALab*)**.**

La asignatura y el laboratorio han existido desde el año 2001 y durante esos años no se le han hecho actualizaciones importantes en el tema de los algoritmos genéticos, ya que, en el caso de la asignatura, solo se enseña un *Algoritmo Genético Tradiciona*l (*TGA* por sus siglas en inglés), limitándose a explicar los conceptos básicos de los algoritmos genéticos.

Este trabajo pretende generar material adicional que puede ser introducido para complementar y actualizar el actual plan de estudios de la asignatura. El material que será producido consistirá en la implementación de un *TGA*, *HTGA* y *CCHTGA* y los documentos de análisis y de comparativas entre ellos basándose en una batería de pruebas.

4. Objetivos

4.1 General

Analizar, implementar en un lenguaje de programación de alto nivel y con un software de pruebas automáticas el *HTGA***,** el *CCHTGA* y un *TGA* para realizar comparativas entre ellos.

4.2 Específicos

1. Revisar el estado del arte en la literatura científica sobre *HTGA* y algoritmos sucesores para elaborar el fundamento teórico para el desarrollo del proyecto.
2. Detallar el funcionamiento del *HTGA* según es descrito en [Tsai, Liu y Chou, 2004], del *CCHTGA* según es descrito en [PoorJandaghi, 2014] y el *TGA* descrito en el plan de estudios de la asignatura de *Computación Evolutiva*.
3. Implementación de *TGA***,** *HTGA* (incluyendo el algoritmo para *Creación de Arreglos Ortogonales* descrito en [Leung y Wang 2001]) y *CCHTGA*
4. Definir una rúbrica y un conjunto de pruebas para ejecutar, evaluar y comparar el desempeño del *TGA*, *HTGA*, y el *CCHTGA*.

5. Alcance

En este trabajo se realizará el análisis y la implementación de los algoritmos *TGA*, *HTGA* y *CCHTGA*. Se realizará un estudio para determinar rúbricas y un conjunto pruebas a realizar para comparar el desempeño de estos algoritmos basándose en las pruebas publicadas en sus artículos correspondientes y a su vez comparar el desempeño entre ellos, de tal manera que la comparación de los resultados sea independiente de la arquitectura del computador. Finalmente, aunque los algoritmos son diseñados inicialmente para solucionar problemas de optimización numérica global, se pretende encontrar algunos problemas de optimización no numéricos que con unas modificaciones a los algoritmos, puedan ser resueltos por estos, y si es posible, agregar estos al conjunto de pruebas.

No corresponde a este trabajo verificar otros algoritmos basados en el *TGA*, *HTGA* o CC*HTGA*, ni tampoco hacer modificaciones a estos en busca de posibles mejoras.

6. Actividades y Resultados Esperados

Tabla 1. Actividades y resultados esperados.

Objetivo	Actividades	Resultado
1	**⇒ Búsqueda bibliográfica** →Búsqueda en Google académico y bases de datos en Univalle de los artículos y libros citados en el artículo original del HTGA. →Búsqueda en Google académico y bases de datos en Univalle de algoritmos sucesores del HTGA y aplicaciones de este.	Compilación sobre la investigación bibliográfica realizada, incluyendo: algoritmos y aplicaciones más relevantes; y los conceptos asociados a estos, como diseño de experimentos, tablas ortogonales, función SNR, robustez, combinación convexa, etc. (ver Marco Teórico)
2	**⇒ Análisis de los algoritmos.** →Análisis de HTGA. →Análisis de CCHTGA. →Análisis de TGA. → Identificar, si existen, problemas que no sean problemas de optimización numérica, que pueda ser resueltos por los algoritmos.	Pseudocódigo de los tres algoritmos y una descripción detallada de las subrutinas. (ver Análisis)
3	**⇒ Implementación de TGA** →Implementar y probar en la funcionalidad de generación de población inicial. →Implementar y probar la función de selección. →Implementar y probar el operador de cruce. →Implementar y probar el operador de mutación →Implementar y probar la funcionalidad de aptitud y de reinserción a la población.	→ Código fuente de los algoritmos implementados y la definición de las pruebas automáticas (steps). → Archivos de texto plano con el contenido de las matrices ortogonales de Taguchi usadas. (ver Implementación)

	⇒ Implementación de HTGA. →Implementar y probar el algoritmo para generación de arreglos ortogonales de Taguchi. →Implementar y probar la función de selección. →Implementar y probar el operador de cruce. →Implementar y probar el operador de mutación. →Implementar y probar la funcionalidad de ejecución de experimentos de matrices. →Implementar y probar la funcionalidad de aptitud y de reinserción a la población. **⇒ Implementación de CCHTGA** → Implementar en Ruby y probar en Cucumber la función de agrupamiento al azar. → Implementar en Ruby y probar en Cucumber las funcionalidades cooperativas y coevolutivas. → Implementar en Ruby y probar en Cucumber el operador de mutación. → Implementar en Ruby y probar en Cucumber la funcionalidad de ejecución de experimentos de matrices.	
4	**⇒ Definición de Rúbricas y Pruebas** → Identificar el tipo de funciones de pruebas de referencia y rúbricas usadas para problemas de optimización y el análisis estadístico usado en los artículos relacionado al HTGA. **⇒ Ejecución de Pruebas.**	Documento de pruebas con análisis de resultados, tablas, tratamiento estadístico, comparación de desempeño, etc. (ver Pruebas y Resultados)

	→ Implementación de una funcionalidad de salida de información de la ejecución para los tres algoritmos. → Monitoreo y registro del resultado arrojado por la ejecución de los tres algoritmos. **⇒ Evaluar y comparar el desempeño de los algoritmos de acuerdo a los resultados de las pruebas.**	

7. Marco Referencial

7.1 Antecedentes

En la siguiente línea de tiempo se describen algunas de las aplicaciones y variantes del *HTGA* más relevantes en un periodo de trece años. Todos los algoritmos son presentados en artículos académicos.

- **Orthogonal Taguchi Genetic Algorithm with Quantization**
  De acuerdo a Leung y Wang (2001), es un algoritmo genético para optimización numérica global de variables continuas por medio de una cuantización. El propósito de la técnica de cuantización es complementar el *diseño ortogonal* en el operador de cruce, para dar como resultado un algoritmo genético más robusto y estadísticamente sólido.

- **Hybrid Taguchi Genetic Algorithm for Global Numerical Optimization**
  De acuerdo a Tsai, Liu y Chou (2004), es un algoritmo genético que tiene un costo computacional menor al de Leung y Wang y está basada en este mismo. Este incluye el método Taguchi y las matrices ortogonales en los operadores de cruce y mutación, dando como resultado un algoritmo genético robusto, estadísticamente sólido, con una convergencia más rápida y mayor precisión en los resultados.

- **Tuning the Structure and Parameters of a Neural Network by Using Hybrid Taguchi-genetic Algorithm**
  De acuerdo a Tsai y Chou (2006), se usa el HTGA para resolver los problemas de ajuste de la estructura de red y los parámetros de una red neuronal de pre-alimentación. La red neuronal se usa para resolver los problemas de predicción del número de manchas solares, ajuste de memoria asociativa y el problema de XOR.

- **Influence of the Crossover Operator in the Performance of the Hybrid Taguchi GA**
  De acuerdo a Picek, Golub y Jakobovic (2012), se exploró variantes del **HTGA** cambiando el operador de cruce. Y se estudió el desempeño de estas variantes en problemas de optimización.

- **Capacity Optimization of MIMO Wireless Communication Systems Using a Hybrid Genetic-Taguchi Algorithm**
  De acuerdo a Recioui y Bentarzi (2012), se aplicó el **HTGA** para resolver el problema de optimizar la capacidad de sistemas de comunicaciones inalámbricas de múltiples entradas y múltiples salidas con diferentes arreglos de configuraciones geométricas lineales.

- **Hybrid Taguchi-Based Genetic Algorithm for FlowShop Scheduling Problem**
  De acuerdo con Yang, Chou y Chang (2012), se desarrolló una variante del **HTGA** para resolver problemas multiobjetivos de programación de líneas de trabajo. Se utiliza

cruce basado en Taguchi para mejorar el desempeño de la búsqueda y se aplica una técnica de selección dinámica de pesos usando un motor de inferencia difusa para evitar el conflicto de programación.

- **The Hybrid Taguchi-Genetic Algorithm for Mobile Location**
  De acuerdo a Chen, Lin, Lee y Lu (2014), se usa el **HTGA** para resolver un problema en la estimación de la ubicación en comunicación inalámbrica, específicamente el problema de *Sin Línea de Visión* (NLOS) que se presenta cuando se transmite desde estación móvil a una estación base.

- **A Robust Evolutionary Algorithm for Large Scale Optimization**
  De acuerdo a Poorjandaghi (2014), se añade la estrategia de *agrupamiento al azar* para crear conjuntos de variables que son dependientes entre sí. Sobre estos conjuntos de variables se aplica una versión mejorada de **HTGA**, además se permite que los conjuntos interactúen entre sí para encontrar mejores valores de solución.

7.2 Marco Teórico

Los **lenguajes de programación de alto nivel** son más cercanos al lenguaje natural de los humanos que al lenguaje de la computadora. Estos permiten a los programadores la escritura de programas que sean independientes del hardware del computador, lo que agiliza la creación y mantenimiento del software. Para este proyecto se considera como mejor opción para implementar los algoritmos, al lenguaje de programación **Ruby** que es "un lenguaje de programación dinámico y de código abierto enfocado en la simplicidad y productividad. Su elegante sintaxis se siente natural al leerla y fácil al escribirla" [Lenguaje Ruby, 2013]. Adicionalmente, es un lenguaje puramente orientado a objetos, procedural, funcional, interpretado y reflexivo con una sintaxis inspirada en Python.

Un **software de automatización de pruebas** es una herramienta aparte del software implementado por los desarrolladores, usado en las metodologías ágiles para la realización de pruebas del software implementado en el proceso de desarrollo. Para este proyecto se consideró como mejor opción para probar los algoritmos implementados, el software **Cucumber,** el cual **"**es una herramienta de software que los programadores usan para probar el software. Sirve para ejecutar pruebas de aceptación automatizadas escritas al estilo de BDD (Desarrollo Orientado al Comportamiento por sus siglas en inglés)" [M.Wynne A. Hellesøy, 2014].

Los **algoritmos genéticos** utilizan la evolución para abordar problemas que son considerados como "intratables" o que son NP-difíciles. Estos abstraen procesos de la evolución biológica para la resolución de problemas en el mundo de la computación. Estos algoritmos tienen aplicaciones en la ingeniería en procesos de optimización, los cuales son cada vez más importante en el mundo actual. Existen muchas clases de algoritmos genéticos, pero hay procesos que son transversales a todos ellos como lo son:

la *generación de la población inicial*, operaciones de exploración (*cruce* y *mutación* de cromosomas), función de *evaluación* y un proceso de *selección*. Cabe resaltar que un cromosoma en estos algoritmos es simplemente una estructura de datos que codifica un candidato a solución, y un gen es un elemento cualquiera en estas estructuras.

El **teorema de no free lunch** afirma que usando algoritmos de búsqueda para todos los tipos de problemas existentes, todos ellos se comportan igual en promedio. Es decir, "el desempeño de cualquier algoritmo, incluyendo de búsqueda aleatoria, puede ser mejor para cierto tipo de problemas, pero siempre habrá otros tipos donde su desempeño sea peor que el resto" [D. Wolper, 1997].

El **desempeño computacional** de un algoritmo genético es típicamente dado por la cantidad de *evaluaciones* de aptitud de los cromosomas, debido a que es una operación "costosa" y repetitiva dentro del algoritmo, y a que este tipo de medición es independiente de la arquitectura del computador.

La **robustez** de un algoritmo evolutivo puede definirse como "un desempeño similar bajo condiciones iniciales distintas" [Poorjandaghi, 2014]. Es decir, mientras menos variación haya en el desempeño del mismo algoritmo ejecutado con distintas condiciones iniciales, más robusto será.

La **selección por ruleta** es un método utilizado en algoritmo genéticos para seleccionar los cromosomas que serán cruzados/mutados (los padres). El proceso consiste en asignar a cada cromosoma una probabilidad de ser seleccionado, esta probabilidad es proporcional a la aptitud del cromosoma, una ruleta imaginaria, de tal manera que el mejor cromosoma tiene la tajada más grande de la ruleta y el peor tiene el más pequeño. Luego se genera un número pseudoaleatorio, se gira la ruleta, todas las veces que sea necesario para seleccionar a los padres.

El **muestreo estocástico universal** es otro método de *selección* utilizado en algoritmos genéticos. Está basado en la selección por ruleta, solo que este garantiza que las frecuencias de selección observadas de cada cromosoma estén en línea con las frecuencias esperadas. Así que si tenemos un cromosoma que ocupa el 4,5% de la ruleta y seleccionamos 100 cromosomas, esperamos que, en promedio, ese cromosoma sea seleccionado entre cuatro y cinco veces. El muestreo estocástico universal garantiza esto. El cromosoma será seleccionado cuatro veces o cinco veces, no tres veces, no cero veces y no 100 veces. La selección por ruleta estándar no garantiza esto [Dyer. D, 2010].

La **combinación convexa** es una técnica usada en algoritmos genéticos derivada de la *teoría de conjuntos convexos*, explicado con detalle en [Gen y Cheng , 2000] para hacer cruces de cromosomas, pero básicamente consiste en ver el cruce de dos cromosomas como la combinación lineal de dos vectores:

$x_1^* = \alpha x_1 + \lambda x_2$ **(1)** $x_2^* = \alpha x_2 + \lambda x_1$ **(2)**

Donde α y λ son constantes, y dependiendo de cómo se restrinja su dominio, se obtienen tres clases de cruces: *linear crossover* si la única restricción es que λ y α sean reales; *affine crossover* sin son reales y $\alpha + \lambda = 1$; y *convex crossover* si además de ser reales y que su suma sea uno se añade que $\alpha, \lambda > 0$.

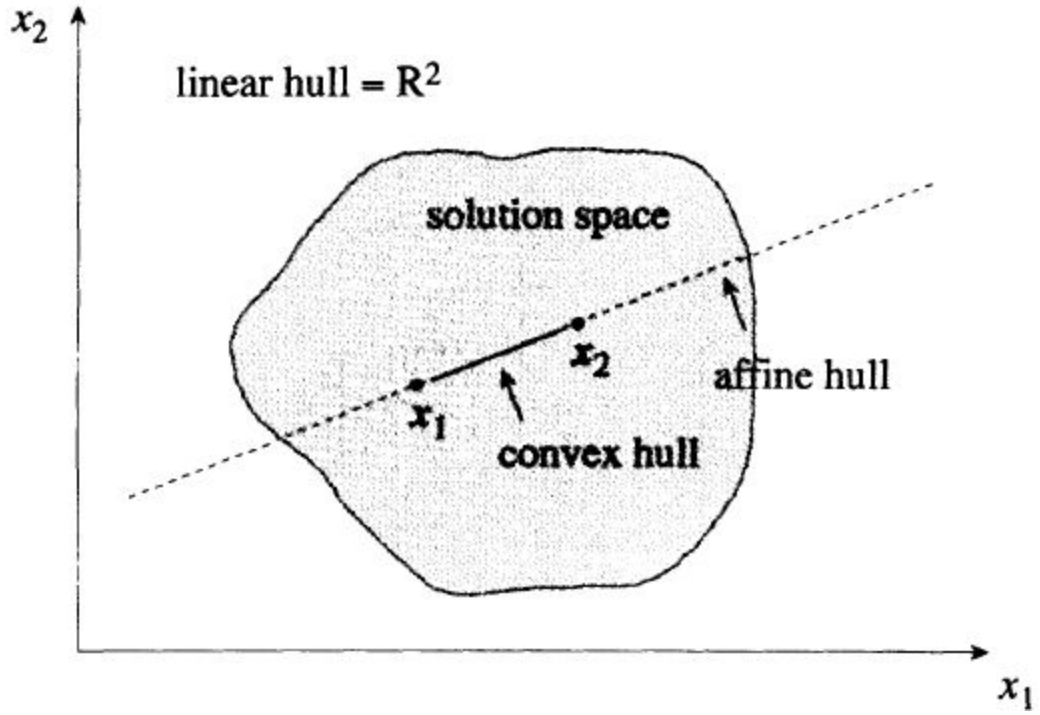

Figura 1: *Gen, M. (2000) Convex combination.[Figura] recuperado de: Genetic algorithms and engineering optimization*

En la figura anterior se denota como *convex hull* (línea en negrita) el espacio compuesto por todas las combinaciones convexas entre dos cromosomas para un espacio en R^2.

El **agrupamiento al azar** es una técnica usada en algoritmos evolutivos explicada en [Omidvar, 2010] para agrupar variables de funciones que interactúan entre sí. Básicamente consiste en que, como a priori no se conoce la mejor manera de agrupar las variables de un problema a resolver, algo recomendable para hacer es agruparlas al azar en subsistemas, luego se usa un algoritmo evolutivo para solucionar cada subsistema generado por separado. Esta estrategia se basa en el trabajo realizado en [Yang, 2008] que demostró, con un algoritmo evolutivo, que la probabilidad de capturar dos variables que interactúan entre sí en grupos seleccionados al azar es más alta a medida que la tasa de agrupamiento al azar aumenta durante el paso de las generaciones.

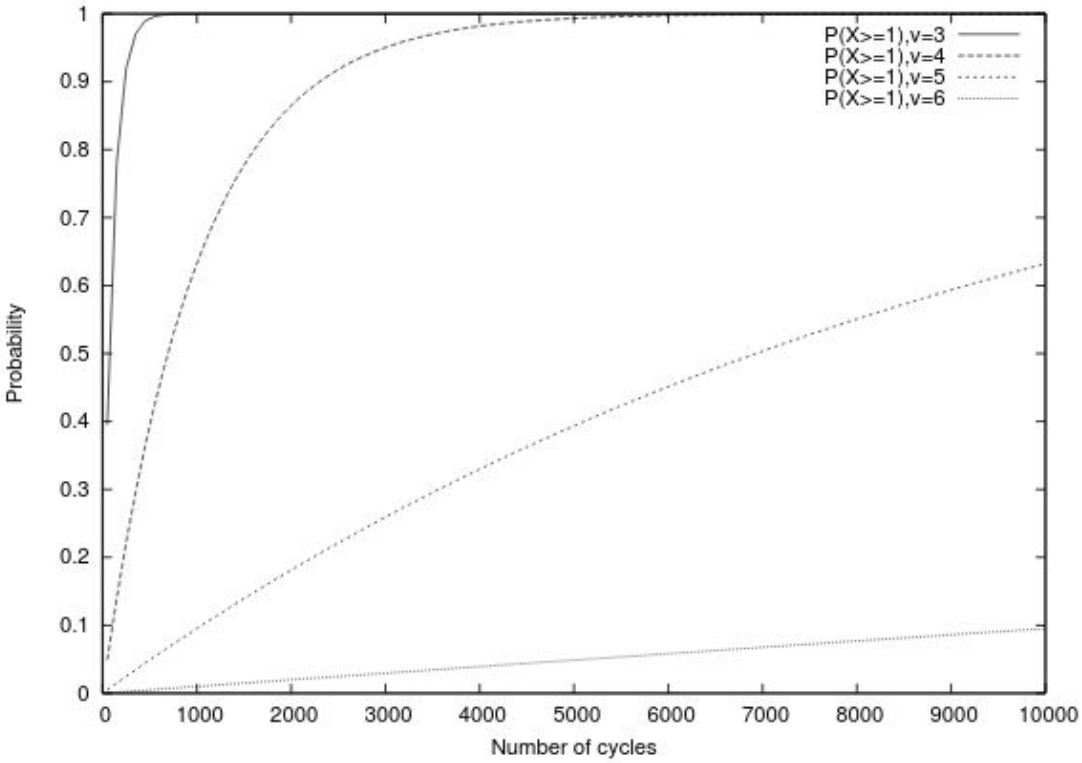

Figura 2. *Omidvar, M. (2010) Interacting variables in subsystem using random grouping.[Figura] Tomado de: Cooperative coevolution for large scale optimization through more frequent random grouping.*

La anterior figura ilustra los hallazgos de Omidvar sobre el agrupamiento al azar, en ella se aprecia que mientras más generaciones o ciclos tiene el algoritmo evolutivo, la probabilidad de agrupar v variables que interactúan entre sí en un mismo subsistema aumenta.

El **método de Taguchi** es un enfoque robusto de diseño, que utiliza muchas ideas del diseño experimental estadístico para la evaluación e implementación de mejoras de productos, procesos y equipo. Tiene como principio fundamental mejorar la calidad de un producto por medio de la minimización de los efectos de las causas de variación sin eliminar las causas. [Tsai, J., Liu, T., & Chou, J. (2004)].

Dos elementos del método son utilizados en este trabajo, el primero son los *arreglos ortogonales de Taguchi* (o *matrices ortogonales de Taguchi*), que también se usan en modelos de experimentación. Estos arreglos, "son matrices fraccionadas factoriales, las cuales aseguran una comparación balanceada de los niveles de cualquier factor o interacción de factores y, debido a su propiedad de ortogonalidad (sus columnas son linealmente independientes) permite la separabilidad de los factores, es decir, que si removemos algunas columnas de la matriz, la matriz resultante sigue siendo ortogonal." [Tsai, J., Liu, T., & Chou, J. (2004)].

Tabla 2. Arreglo Ortogonal de Taguchi $L_8(2^7)$

Experimentos	Factores						
	1	2	3	4	5	6	7
1	0	0	0	0	0	0	0
2	0	0	0	1	1	1	1
3	0	1	1	0	0	1	1
4	0	1	1	1	1	0	0
5	1	0	1	0	1	0	1
6	1	0	1	1	0	1	0
7	1	1	0	0	1	1	0
8	1	1	0	1	0	0	1

La tabla anterior muestra el *arreglo ortogonal* más sencillo utilizada en este trabajo, el arreglo se denomina $L_8(2^7)$ siguiendo la notación *Latin Square:*
$L_n(Q^{n-1})$
Donde:
$n = 2^k$ número de experimentos (filas de la matriz)
k número entero positivo mayor o igual a uno
Q número de niveles
$n - 1$ cantidad de factores (columnas de la matriz)

[1] En el resto de este documento se usarán las palabras factor, variable y gen de manera indistinta.

En este trabajo sólo se utilizan *arreglos ortogonales de Taguchi* con dos niveles, debido a que en este caso los experimentos tratan de encontrar la mejor manera de cruzar *dos* cromosomas. En concreto solo se usan los arreglos $L_8(2^7)$, $L_{16}(2^{15})$, $L_{32}(2^{31})$, $L_{64}(2^{63})$, $L_{128}(2^{127})$, $L_{256}(2^{255})$, $L_{512}(2^{511})$ y $L_{1024}(2^{1023})$ para problemas de hasta 7, 15, 31, 63, 127, 255, 511 y 1023 variables respectivamente.

El segundo elemento del *método Taguchi* utilizado es la *SNR* (Tasa Señal Ruido por sus siglas en inglés), la cual se define como "la tasa del potencial de una señal (la información significativa) y el potencial del ruido de fondo (señal no deseada)" [N. Dewangan, 2013]. La *SNR* sirve para identificar qué tanto contribuye el valor de una variable al valor de la función, ya que esta "transforma varias repeticiones en un solo valor, que refleja la cantidad de variación presente y la respuesta media"[Tsai, J., Liu, T., & Chou, J. (2004)].

La fórmula general de la *SNR,* encontrada en [Su, 2013] es:

$$\eta = 10 \times \log_{10}(\frac{signal}{noise}) \quad \textbf{(3)}$$

Para este trabajo se tiene una señal fija, por lo tanto la tasa se convierte en MSD (Desviación Media Cuadrada por sus siglas en inglés). Entonces, la ecuación resultante es:

$$\eta = -10 \times \log_{10}(MSD) \quad \textbf{(4)}$$

Estos valores *SNR* tienen unidades en decibeles, sencillamente porque es recomendado por Taguchi, cuyo historial de trabajo está en las telecomunicaciones. La *SNR* se considera favorable mientras más alto sea su valor (menor pérdida de calidad), es decir, la influencia de los valores de los factores estudiados en el valor de la función es significativo; y es desfavorable en caso contrario (mayor pérdida de calidad).

8. Análisis de los Algoritmos

8.1 HTGA

Presentado en el artículo por Tsai, Liu, and Chou en 2004 y basado en el algoritmo *Orthogonal Taguchi Genetic Algorithm with Quantization* del 2001 introducido por Leung y Wang, es un algoritmo que pretende solucionar problemas con variables continuas, combinando los *algoritmos genéticos,* que tiene una gran capacidad de exploración, con el *método estadístico de Taguchi*. Los dos elementos del método se introducen en el operador de cruce, para buscar la mejor descendencia, esto se logra seleccionando los mejores genes de cada cromosoma para realizar la operación. Con esto se logra que el *HTGA* sea a su vez más robusto y estadísticamente sólido que un *algoritmo genético tradicional*.

En este trabajo la generación de los *arreglos ortogonales de Taguchi* se logró mediante la utilización de un algoritmo propuesto en [Leung y Yang, 2001] denominado como *OA Permut* (Orthogonal Array Permutations), que permite generar arreglos ortogonales.

Algoritmo 1: Pseudocódigo de OA Permut

```
1:  procedure OA_PERMUT(J, Q, N, M)
2:      for k ← 0 to J - 1 do
3:          j ← (Q^(k-1) − 1)/(Q − 1)
4:          for i ← 0 to Q^j − 1 do
5:              a_{i,j} ← mod(⌊(i − 1)/Q^(j−k)⌋, Q)
6:      for k ← 1 to J - 1 do
7:          j ← (Q^(k-1) − 1)/(Q − 1)
8:          for s ← 0 to j do
9:              for t ← 0 to Q - 1 do
10:                 a_{j+(s+1)(Q−1)×(t+1)} ← mod(a_s × (t + a_j), Q)
```

Donde:

$a_{i,j}$ es la matriz siendo generada

Q son los niveles de la matriz

N es el número de columnas

J es un número seleccionado tal que $N = \frac{Q^J-1}{Q-1}$

M es el número de filas

Las bucles de las líneas dos a cinco, generan lo que en el artículo denominan como *columnas básicas*. Y los bucles de las líneas seis a diez, generan las *columnas no básicas,* también denominadas así en el artículo.

Ahora, la *SNR* es usada en el *HTGA* con las siguientes fórmulas:

$$\eta = -10\ log(\frac{1}{N}\sum_{i=1}^{N} x_i^2) \qquad \textbf{(5)}$$

$$\eta = -10\ log(\frac{1}{N}\sum_{i=1}^{N} \frac{1}{x_i^2}) \qquad \textbf{(6)}$$

Donde:
N es la dimensión del problema (cantidad de variables)
x_i la i-ésima variable.

La Ecuación 5 es usada para casos de minimización y la Ecuación 6 para casos de maximización.

En una operación de cruce, existen muchas maneras de combinar los genes de dos cromosomas, unas mejores que otras, pero como la combinatoria de posibilidades es muy grande, resulta impráctico probar todas las alternativas de hacerlo para encontrar la mejor. Es aquí donde se incorporan los *arreglos ortogonales* y la *SNR* en el *HTGA*. Aplicando el método de Taguchi se toman los genes de cada cromosoma como columnas del *arreglo ortogonal*, escogiendo siempre el arreglo con el número de columnas más cercano que sea mayor o igual a la cantidad de genes (como las columnas son linealmente independientes, no importa si se tienen menos genes que columnas), y se ejecutan los experimentos o filas del arreglo. Por cada experimento se genera un cromosoma posible al cual se le calcula su *SNR*. Finalmente, se calcula el mejor nivel para cada gen y se genera un cromosoma óptimo.

Este ejemplo, tomado del artículo original del *HTGA***,** detalla el proceso:

Dado dos cromosomas binarios

Tabla 3. Cromosomas binarios

cromosoma cero	1	1	1	1	0	0	0
cromosoma uno	0	0	0	0	1	1	1

Y la función para minimizar

$$\sum_{i=1}^{N} x_i^2 \qquad \textbf{(7)}$$

Se procede a generar un cruce entre ambos usando el *método Taguchi*, teniendo en cuenta que el problema a resolver es de minimización. Primero, se selecciona $L_8(2^7)$ como el arreglo ortogonal más conveniente para el caso. Segundo, se ejecutan los experimentos generando los ocho cromosomas posibles, colocando el gen del cromosoma cero (nivel uno) si se encuentra un cero en la matriz de experimentos y eligiendo el gen del cromosoma uno (nivel dos) en caso contrario, después se calcula la *SNR* (usando la Ecuación 5) para cada experimento. Los resultados se muestran en la siguiente tabla:

Tabla 4. Cruces experimentales entre cromosomas cero y uno

Experimentos	Factores							SNR
	1	2	3	4	5	6	7	
1	1	1	1	1	0	0	0	1/16
2	1	1	1	0	1	1	1	1/36
3	1	0	0	1	0	1	1	1/16
4	1	0	0	0	1	0	0	1/4
5	0	1	0	1	1	0	1	1/16
6	0	1	0	0	0	1	0	1/4
7	0	0	1	1	1	1	0	1/16
8	0	0	1	0	0	0	1	1/4

Tercero, se calculan los efectos de los factores por cada nivel usando la fórmula:

$$E_{fk} = \textit{suma de } SNR_i \textit{ del factor } f \textit{ en el nivel } k, \quad k \in \{1, 2\}\ i \in \{1,2,3,..,\textit{número experimentos}\} \qquad \textbf{(8)}$$

Entonces para los factores uno y dos, el cálculo queda así:

$$E_{11} = \frac{1}{4^2} + \frac{1}{6^2} + \frac{1}{4^2} + \frac{1}{2^2} = 0.403 \qquad E_{12} = \frac{1}{4^2} + \frac{1}{2^2} + \frac{1}{4^2} + \frac{1}{2^2} = 0.625$$

$$E_{21} = \frac{1}{4^2} + \frac{1}{6^2} + \frac{1}{4^2} + \frac{1}{2^2} = 0.403 \qquad E_{22} = \frac{1}{4^2} + \frac{1}{2^2} + \frac{1}{4^2} + \frac{1}{2^2} = 0.625$$

Como $E_{12} > E_{11}$ entonces, el nivel óptimo del gen uno es el dos. Igualmente el nivel dos es óptimo para el gen dos ya que $E_{22} > E_{21}$.

Repitiendo este proceso para cada uno de los genes, al final se tiene que los niveles y cromosoma óptimos son:

Tabla 5. Resultados del cruce con Taguchi

Nivel óptimo	2	2	2	2	1	1	1
Cromosoma óptimo	0	0	0	0	0	0	0

Como en este caso la aptitud del cromosoma es igual a la evaluación de la función objetivo, se evidencia que el cromosoma generado realmente es un cruce óptimo entre los cromosomas cero y uno.

En el *HTGA* se agrega el anterior operador de cruce al *algoritmo genético*, pero además se utiliza otro operador de cruce, el operador *de cruce de un punto de corte* integrado con *combinación convexa*. En él se selecciona los genes x_k y y_k de ambos cromosomas, donde k se elige al azar entre cero y el tamaño de los cromosomas. Luego se generan dos nuevos genes x^*_k y y^*_k utilizando las siguientes fórmulas:

$$x^*_k = x_k + \beta(y_k - x_k) \quad \textbf{(9)}$$

$$y^*_k = l_k + \beta(u_k - l_k) \quad \textbf{(10)}$$

Donde:

β es un elemento escogido al azar del conjunto $\{0.0, 0.1, 0.2, 0.3, ..., 1.0\}$

u_k y l_k son las cotas superiores e inferiores de la k-ésima variable respectivamente.

Después, se intercambian las partes derechas de ambos cromosomas generando dos cromosomas así:

$$x^* = [x_0, x_1, ..., x^*_k, y_{k+1}, y_{k+2}, ..., y_n]$$

$$y^* = [y_0, y_1, ..., y^*_k, x_{k+1}, x_{k+2}, ..., x_n]$$

Algoritmo 2: Pseudocódigo para el operador de cruce de un punto de corte del *HTGA*

```
1:  procedure CROSSOVER_OP(chromos,p_c,M, N,u,l)
2:      for m ← 0 to M-1 do
3:          r ← Random{ U(0,1) }
4:          if r ≥ p_c then
5:              i ← IntegerRandom{ U(0, M-1)}
6:              j ← IntegerRandom{ U(0, M-1)}
7:              x ← chromos_i
8:              y ← chromos_j and ≠ x
9:              k ← Random{ U(0, N)}
10:             swap_right_parts(k, x, y)
11:             β ← Random{ U(0, 1)}
12:             x_k ← x_k + β(y_k − x_k)
13:             y_k ← l_k + β(u_k − l_k)
14:             add_to_list(chromos, [x, y])
```

Donde:
chromos es la lista de cromosomas
p_c es la tasa de cruce
M es el tamaño de la población
N tamaño del cromosoma
u, l son las cotas superiores e inferiores de los genes respectivamente.

El tercer y último operador de reproducción usado en el *HTGA* es el de la mutación, que también se basa en la *teoría de conjuntos convexos*. Por cada cromosoma seleccionado para mutación, se eligen al azar dos genes x_i y x_k para mutar y generar dos nuevos genes x^*_i y x^*_k (el valor β es aleatorio igual al usado en el operador de cruce de un punto de corte.) siguiendo las fórmulas:

$x^*_i = (1-\beta)x_i + \beta x_k$ **(11)**

$x^*_k = \beta x_i + (1-\beta)x_k$ **(12)**

El propósito de mutar los genes de esta manera, es el de acercar los valores de estos en el espacio de búsqueda.

Algoritmo 3: Pseudocódigo para operador de mutación del HTGA

```
1:  procedure MUTATION_OP(chromos,p_m,N)
2:      S ← size(chromos)
3:      for m ← 0 to S-1 do
4:          r ← Random{ U(0,1) }
5:          if r ≥ p_m then
6:              t ← IntegerRandom{ U(0, S-1)}
7:              x ← chromos_t
8:              i ← IntegerRandom{ U(0, N-1) }
9:              k ← IntegerRandom{ U(0, N-1)}and ≠ i
10:             β ← Random{ U(0, 1)}
11:             x_i ← (1-β )x_i + βx_k
12:             x_k ← βx_i+ (1-β )x_k
13:             add_to_list(chromos, x)
```

Donde:
chromos es la lista de cromosomas
p_m es la tasa de mutación
N es el tamaño del cromosoma

Como última subrutina relevante del *HTGA* se tiene a la *generación de la población inicial*, en ella, por cada gen de cada cromosoma de la población, se utiliza la fórmula:

$x_i = l_i + \beta(u_i + l_i)$ **(13)**

Donde:
x_i es el i-ésimo gen
l_i es la cota inferior del i-ésimo gen
u_i es la cota superior del i-ésimo gen
β es un número escogido al azar entre cero y uno

Algoritmo 4: Pseudocódigo para la generación de la población inicial del HTGA

```
1: procedure GENERATE_INIT_POP(u, l, M, N)
2:     chromos ← [1...M]
3:     for j ← 0 to M-1 do
4:         x ← [1...N]
5:         for i ← 0 to N - 1 do
6:             β ← Random{ U(0, 1) }
7:             x_i ← l_i + β(u_i − l_i)
8:         chromos_j ← x
9:     return chromos
```

Donde:
u, l son cotas superiores e inferiores de los genes respectivamente.
M es el tamaño de la población.
N es el tamaño del cromosoma.

Finalmente, se resume el *HTGA* en el Algoritmo 5:

Algoritmo 5: Pseudocódigo del funcionamiento general del HTGA

```
1: procedure HTGA(u, l, M,N, p_c, p_m)
2:     generation ← 0
3:     chromos ← generate_init_pop(u, l, M, N)
4:     loop
5:         roulette_selection_op(chromos)
6:         crossover_op(chromos, pc, M, N, u, l)
7:         taguchi_method_for_cross(chromos)
8:         mutation_op(chromos,pm,N)
9:         sort_chromos_by_fitness_vals(chromos)
10:        select_better_M_chromos(chromos, M)
11:        if stoping_criterion_has_been_met? then
12:            terminate()
13:        generation ← generation + 1
```

Donde:
N es la cantidad de genes.
M tamaño de la población.
u y l son arreglos de tamaño N con las cotas superiores e inferiores de los genes respectivamente.
p_c y p_m son las tasas de cruce y mutación respectivamente.

La línea dos inicializa el contador de generaciones; la línea tres corresponde a la generación de población inicial; en la línea cuatro inicializa el bucle de ejecución del algoritmo; línea cinco realiza la selección por ruleta; línea seis realiza el cruce de un punto de corte unas $p_c \times M$ veces; en la línea siete realiza el cruce con el *método Taguchi* unas $0.5 \times p_c \times M$ veces: en la línea ocho se realiza mutación unas $p_m \times M$ veces; en la línea nueve se ordena los cromosomas por su aptitud; en la línea diez se seleccionan los mejores M cromosomas para la siguiente generación; en las líneas once a doce se verifica si el criterio de parada se ha cumplido, y de ser así, detiene la ejecución ; en la línea trece se aumenta el contador de ejecuciones y se ejecuta todo a partir de la línea cinco.

La complejidad computacional teórica del *HTGA* viene siendo dada por la complejidad de las subrutinas:

- Generación de población inicial: $O(MN)$
- Ruleta: $O(M)$
- Cruce de un punto de corte: $O(MN)$
- Cruce de Taguchi: $O(MN^2)$
- Mutación: $O(MN)$
- Ordenamiento de la población: $Mlog(M)$

y sea g la cantidad de generaciones del algoritmo, se obtiene:
$O\left(gMN^2\right)$
[2] Para simplificación se asume que el costo de evaluar la aptitud es $O(N)$, lo cual en realidad muchas veces no se cumple, ya que las funciones objetivos pueden ser más costosas de evaluar.

8.2 CCHTGA

Presentado en el artículo *A Robust Evolutionary Algorithm for Large Scale Optimization*[Poorjandhagi, 2014], es un algoritmo que combina la estrategia de divide y conquista utilizando la técnica *agrupamiento al azar* para generar un algoritmo robusto que funcione bien con problemas de grandes dimensiones. Este algoritmo se deriva del *HTGA* y, según su autor, funciona mejor para resolver problemas de optimización de funciones donde existen dependencias entre variables.

El *CCHTGA* contiene una versión mejorada del *HTGA* que se usa como subrutina denominado en este trabajo como *IHTGA* (*Algoritmo Genético Híbrido de Taguchi*

Mejorado por sus siglas en inglés). Son tres los cambios realizados en esta mejora. Primero, se cambia la fórmula de la *SNR* para mejorar el desempeño del algoritmo en problemas de optimización.

La nueva fórmula del *SNR* es:

$$\eta_i = (z_i - f(\hat{y}))^{-2} \qquad \textbf{(14)}$$

Donde:

z_i es la aptitud del cromosoma en el i-ésimo experimento

$f(\hat{y})$ es la aptitud del mejor cromosoma hasta el momento

La Ecuación 14 es derivada de la fórmula de la *BSD* (Mejor Desviación Cuadrada por sus siglas en inglés) introducida en [Poorjandaghi, 2014].

$$BSD = \sigma^2 + \bar{z}^2 \qquad \textbf{(15)}$$

Poorjandaghi menciona que la Ecuación 15 tiene una estrecha relación con la Ecuación 5 que funciona bien para funciones con mínimo cero, pero no tan bien en otros casos, cosa que la Ecuación 15 si hace.

El segundo cambio es una nueva regla de mutación para mejorar la capacidad de búsqueda.

Algoritmo 6: Pseudocódigo del operador de mutación en CCHTGA

```
1:  procedure MUTATION_OP(chromos, N p_m, u, l)
2:      S ← size(chromos)
3:      ŷ ← get_best(chromos)
4:      for m ← 0 to S - 1 do
5:          b ← Random{ U(0, 1) }
6:          if b ≤ p_m then
7:              x ← chromos_m
8:              y ← best_experiences(x)
9:              for i ← 0 to N-1 do
10:                 r ← Random{ U(0, 1) }
11:                 p ← Random{ U(0, 1) }
12:                 if p < 0.5 then
13:                     x_i ← l_i + r(u_i − l_i)
14:                 else
15:                     x_i ← x_i + (2r -1) × |y_i − ŷ_i|
16:             add_to_list(chromos, x)
```

Donde:

chromos es la lista de cromosomas

p_m es la tasa de mutación

N es el tamaño del cromosoma
u, l son las cotas superiores e inferiores de los genes respectivamente

La nueva regla de mutación se encuentra en el bucle de las líneas nueve a quince, que consiste en mutar aproximadamente la mitad de los genes de manera exploratoria (por amplitud) en la línea trece, y la otra mitad son mutados por explotación (por profundidad) en la línea quince.

Como con la nueva regla de mutación existe la posibilidad de que los genes del cromosoma excedan sus cotas, se utiliza la siguiente función correctiva:

Algoritmo 7: Pseudocódigo para la función correctiva de genes del CCHTGA

```
1: procedure CORRECT_GENE(x_i, l_i, u_i)
2:     if x_i < l_i then
3:         x_i ← 2l_i − x_i
4:     else
5:         if u_i < x_i then
6:             x_i ← 2u_i − x_i
7:     return x_i
```

Esta función sencillamente regresa el gen "corrupto" a su dominio proporcionalmente al doble del tamaño de la violación.

Y el tercer cambio realizado al *HTGA* fue el de reemplazar la *selección por ruleta* al inicio del algoritmo por *muestreo estocástico universal* para seleccionar los cromosomas al final de cada iteración.

La parte más importante del *CCHTGA* es como dividir y resolver el sistema general en subsistemas, teniendo en cuenta que estos pueden tener dependencia entre sus variables. Para esto hace uso de la *cooperación coevolutiva* o *CC*, que consiste de los siguientes tres pasos:

Paso 1: Sea N la dimensión del problema, se selecciona al azar un entero s de una lista de divisores de N, con este divisor se calcula la cantidad de subsistemas K en el que se dividirá el problema, siguiendo la fórmula:

$$K = \frac{N}{s} \qquad \textbf{(16)}$$

Paso 2: Utilizar *agrupamiento al azar* para determinar qué variables entrarán en qué subsistema y dividir los genes de los cromosomas.

Paso 3: Resolver con un algoritmo evolutivo cada subsistema y reconstruir los cromosomas divididos.

Ahora, se resume con pseudocódigo el *CCHTGA,* según como es descrito por Poorjandhagi en el Algoritmo 8:

Algoritmo 8: Pseudocódigo del funcionamiento general del CCHTGA

```
1: procedure CCHTGA(p_m, p_c, u, l, M, N)
2:     chromos ← generate_init_pop(u, l, M, N)
3:     G ← {}
4:     (K, s) ← divide_variables(N)
5:     for t ← 0 to MAX_ITER-1 do
6:         Δf(ŷ) ← f(ŷ_t) − f(ŷ_{t−1})
7:         if Δf(ŷ) > 0 or Δf(ŷ) < f(ŷ_{t−1}) × 0.1 then
8:             G ← random_grouping(chromos, K, s)
9:         for j ← 0 to K-1 do
10:            for i ← 0 to M-1 do
11:                if f(best(j,G_j.x_i)) < f(best(j,G_j.y_i)) then
12:                    G_j.y_i ← G_j.x_i
13:                if f(best(j,G_j.y_i)) < f(best(j,G_j.ŷ_t)) then
14:                    G_j.ŷ_t ← G_j.y_i
15:                if f(best(j,G_j.ŷ_t)) < f(ŷ_t) then
16:                    jth_part_of(ŷ_t,j) ← G_j.ŷ_t
17:                correct_gene(ŷ_t, l_t, u_t)
18:        for j ← 0 to K-1 do
19:            apply_HTGA_to(G_j, p_m, p_c, u, l, M, N)
```

Donde:
p_m es la tasa de mutación.
p_c es la tasa de cruce.
u, l son las cotas superiores e inferiores de los genes respectivamente.
M es el tamaño de la población de cromosomas.
N es el tamaño del cromosoma.

f es la función de aptitud (función objetivo).
K es la cantidad de grupos o subsistemas.
s es la cantidad de variables en los subsistemas.
x_i es el i-ésimo cromosoma.
y_i es la mejor experiencia del cromosoma x_i.
$\hat{y}_t, \hat{y}_{t-1}$ son los mejores cromosomas de la generación actual y la anterior respectivamente.
$G_j.x_i$ es el i-ésimo sub cromosoma en el j-ésimo subsistema.
$G_j y_i$ es el mejor mejor experiencia del cromosoma x_i en el j-ésimo subsistema.
$G_j.\hat{y}_t$ es el mejor cromosoma actual en el j-ésimo subsistema.
best(j, z) es una función que reemplaza los genes de z del j-ésimo subsistema en $\hat{y}_t$.

En la línea dos se inicializa la población inicial; en las línea tres a cuatro se inicializa la lista de subsistemas y divide las N variables en K subsistemas; en la línea cinco se inicializa el bucle principal del algoritmo, cada iteración en este es una generación; en las líneas seis a siete se verifica que la mejor aptitud de la generación actual sea mejor que la anterior (línea siete, primera condición del OR) o si es significativamente mejor (línea siete, segunda condición del OR), es decir, si la diferencia entre la mejor aptitud actual y la mejor aptitud anterior es mayor al 10% de la mejor aptitud anterior; en línea ocho se realiza *agrupamiento al azar* y los grupos resultantes se almacenan en G; el bucle de la línea nueve itera sobre todos los grupos en G y el bucle de la línea diez itera sobre los cromosomas; la validación de las líneas once a doce es para verificar si los genes en $G_j x_i$ mejoran la aptitud de y_i (con este cromosoma es que se logra la *cooperación*) más que los genes en $G_j y_i$, de ser así, se actualiza el valor de $G_j y_i$ con el de $G_j x_i$ (con estos chequeos es que se logra la *coevolución*); las validaciones en las líneas trece a catorce y quince a dieciséis son iguales a las de las líneas once a doce, solo que se cambia el grupo de cromosomas; en la línea diecisiete se corrigen los genes que hayan violado sus cotas. Finalmente, las líneas dieciocho a diecinueve ejecutan la versión mejorada del *HTGA* por cada subsistema en G.

La complejidad computacional teórica del CCHTGA viene siendo dada por la complejidad de las subrutinas:

- Generación de población inicial: $O(MN)$
- División de variables: $O(N)$
- Agrupamiento al azar: $O(KN)$
- Bucles (líneas 9-17): $O(KMN)$
- Bucle de *IHTGA* (líneas 18-19): $O\left(KMN^2\right)$

Y como K es una función $f(N)$, se obtiene:
$O\left(gMN^3\right)$

8.3 TGA

Es un algoritmo genético básico usado en la asignatura de *Computación Evolutiva* [García, 2008] para explicar los conceptos fundamentales de estos algoritmos. Fue inspirado en el algoritmo genético ideado por John Holland en los setentas. Este es de los algoritmos genéticos más sencillos que hay, contando con las siguientes subrutinas: *población inicial*, *reproducción*, *selección*, y *reinserción*. En este se utilizan como operadores de reproducción el cruce y la mutación, y en la selección se puede usar torneo o ruleta.

La *generación de la población inicial* consiste en generar valores aleatorios entre las cotas superiores e inferiores de cada gen por cada cromosoma de la población.

En la *selección* se utiliza *torneo* con dos iteraciones, en las que se seleccionan dos cromosomas de forma aleatoria de la población y se comparan sus aptitudes, eligiendo al mejor de ellos.

Los elegidos en la selección se pasan al *mating pool*, que es un subconjunto de cromosomas que generará descendencia. Para ello, se tendrán dos cromosomas para aplicar *cruce* de punto de corte, generando dos cromosomas nuevos. Después, a los cromosomas seleccionados originalmente (los padres en el cruce) se les aplica *mutación discreta,* que consiste en elegir un gen de un cromosoma aleatoriamente y asignarle un nuevo valor, escogido aleatoriamente de entre la lista posible de genes (alelos). Al final de este proceso, se habrán generado cuatro cromosomas nuevos, dos producto del cruce y dos producto de la mutación.

En la *reinserción* se eliminan aleatoriamente cuatro cromosomas de la población y se insertan los cuatro cromosomas generados en la reproducción para mantener un tamaño de población fijo.

Algoritmo 9: Pseudocódigo del funcionamiento general del TGA

```
1:  procedure TGA(N, M, u, l)
2:      chromos ← generate_init_pop(N, M, u, l)
3:      generation ← 0
4:      loop
5:          mating_pool ← selection(chromos)
6:          chromo_1, chromo_2 ← crossover(mating_pool)
7:          chromo_3, chromo_4 ← mutation(mating_pool, u, l)
8:          offspring ← [chromo_1, chromo_2, chromo_3, chromo_4]
9:          insertion_operation(offspring, chromos)
10:         if stoping_criterion_has_been_met? then
11:             terminate()
12:         generation ← generation + 1
```

Donde:
N tamaño del cromosoma
M tamaño de la población de cromosomas
u, l son las cotas superiores e inferiores de los genes respectivamente

La complejidad computacional teórica del *TGA* viene siendo dada por la complejidad de las subrutinas:

- Generación de población inicial: $O(NM)$.
- Operación de selección: $O(M)$
- Cruce: $O(N)$
- Mutación: $O(N)$
- Inserción: $O(1)$

El total sería:
$O(gNM)$

8.4 TGAKnapsack y HTGAKnapsack

Son adaptaciones del *TGA* y *HTGA* para resolver el problema de la mochila 0-1 [Goddard, 2004] con múltiples restricciones. Estas adaptaciones se basaron en el trabajo realizado por Nils Haldenwang en [Haldenwang, 2011] y consisten de cuatro cambios.

- La *generación de la población inicial* consiste en generar valores binarios aleatorios.
- Se utiliza un operador de *mutación* que genera el *complemento a uno* del valor que tiene el gen, es decir, si el gen tiene el valor cero, se coloca un uno y visceversa.
- Se utiliza *cruce de un punto de corte* sin *combinación convexa*.
- Se reparan los genes de los cromosomas, que durante el cruce o la mutación son generados y son inválidos (no cumplen con las restricciones de la mochila), la

corrección se hace eligiendo al azar los primeros k genes del cromosoma ($k \leq N$) que no rompan las restricciones y asignar cero al resto de los genes.

9. Implementación

9.1 Arquitectura

La estructura de ficheros del proyecto es la siguiente:

```
├──── base_ga
│     └──── base_ga.rb
├──── cchtga
│     ├──── cchtga.rb
│     ├──── features
│     └──── ihtga.rb
├──── execute.rb
├──── helpers
│     ├──── chromosome.rb
│     ├──── roulette.rb
│     ├──── subsystem.rb
│     ├──── sus.rb
│     └──── test_functions.rb
├──── htga
│     ├──── features
│     ├──── htga_knapsack
│     │     └──── htga_knapsack.rb
│     └──── htga.rb
├──── oa_permut
│     ├──── features
│     └──── oa_permut.rb
├──── taguchi_orthogonal_arrays
│     ├──── L1024
│     ├──── L128
│     ├──── L16
│     ├──── L256
│     ├──── L32
│     ├──── L512
│     ├──── L64
│     └──── L8
└──── tga
      ├──── features
      ├──── tga_knapsack
      │     └──── tga_knapsack.rb
      └──── tga.rb
```

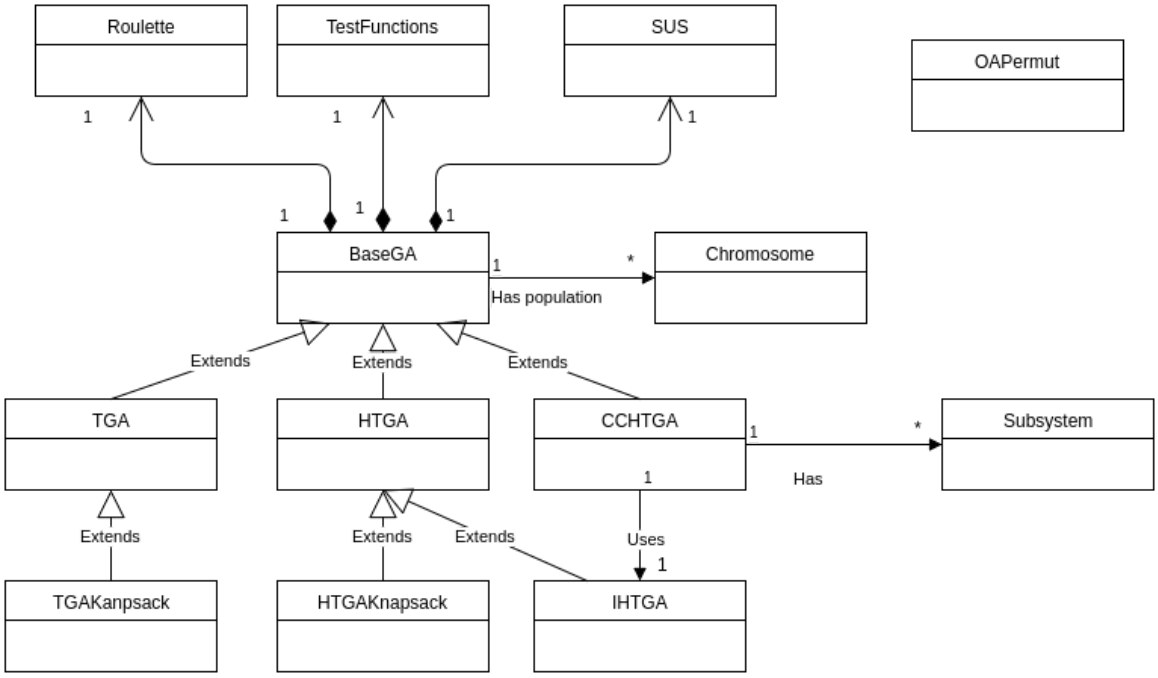

Figura 3: *Diagrama de clases*

Cada uno de los cuatro algoritmos implementados (*OA Permut*, *HTGA*, *CCHTGA* y *TGA*) tiene su propia carpeta con ficheros fuente y carpeta *features* con las pruebas escritas en *Cucumber*. Las entidades y subrutinas que son transversales a los algoritmos genéticos se encuentran en la carpeta *helpers*. Hay una carpeta para los *arreglos ortogonales de Taguchi* generados por *OA Permut*. También está la carpeta *base_ga* que contiene un archivo con una clase del mismo nombre, de la cual heredan los tres algoritmos genéticos. Por último, está el archivo ejecutable *execute.rb* que contiene la lógica para ejecutar los algoritmos genéticos.

oa_permut.rb

Contiene la implementación como módulo en Ruby del algoritmo denominado como *OA Permut* (*Orthogonal Arrays Permutations*). La implementación presentada en este trabajo se basó en una implementación previa en Matlab realizada en [Jeppu, 2014].

Este se ejecuta por consola con el comando

```
$ ruby oa_permut.rb q n j
```

Argumentos:
q número de niveles del arreglo ortogonal

n número de columnas del arreglo ortogonal
j número entero seleccionado tal que $n = \frac{q^j - 1}{q - 1}$

Salida:
Un archivo de texto plano sin extensión cuyo nombre es "L" concatenado con el número de filas del arreglo (L8, L16, L32, L64, L128, L256, L512, L1024) que se guarda en el directorio *taguchi_ortghogonal_arrays.*

base_ga.rb

Contiene la implementación de la clase en Ruby llamada *BaseGA* que sirve de clase padre para las clases de los tres algoritmos genéticos *TGA*, *HTGA* y *CCHTGA*. Las subrutinas contenidas en esta clase son: los métodos de *selección de ruleta* y *muestreo estocástico universal* , y la *evaluación de cromosomas*

tga.rb

Contiene la implementación de la clase en Ruby llamada TGA que hereda de la clase BaseGA. Esta clase implementa todas las subrutinas del *TGA*: *generación de la población inicial, selección por torneo, cruce uniforme, mutación discreta y reinserción* a la población.

Argumentos:
pop_size: Entero que indica la cantidad de cromosomas .
upper_bounds: Un arreglo de números reales que serán las cotas superiores de los genes.
lower_bounds: Un arreglo de números reales que serán las cotas inferiores de los genes.
num_genes: Entero que indica la cantidad de genes en un cromosoma.
selected_func: Entero que indica la función de prueba a resolver.
max_generation: Entero que indica el número de máximo de generaciones permitidas.
continuous: Booleano que indica si el problema a solucionar es continuo o no.
is_high_fit: Booleano que indica si el problema a resolver es de minimización o maximización.

Salida:
best_fit: La mejor aptitud encontrada por el algoritmo.
gen_of_best_fit: La generación en la que se encontró la mejor aptitud.
func_evals_of_best_fit: Cantidad de evaluaciones realizadas para encontrar la mejor aptitud.
relative_error: el error relativo.
optimal_func_val: el óptimo real de la función resuelta.

tga_knapsack.rb

Contiene la implementación de la clase en Ruby llamada *TGAKnapsack* que hereda de la clase TGA. Esta clase adapta las subrutinas de su clase padre para resolver el problema de la mochila 0-1.

Argumentos:
pop_size: Entero que indica la cantidad de cromosomas.
num_genes: Entero que indica la cantidad de genes en un cromosoma.
optimun: El valor óptimo del problema.
max_generation: Entero que indica el número de máximo de generaciones permitidas.
number of constraints: La cantidad de restricciones de la mochila.
values: Arreglo de los beneficios.
max_weights: Arreglo con las constante al lado derecho de las inecuaciones de las restricciones.
weights: Arreglo bi-dimensional en el que cada cada fila contiene las constantes de las restricciones de la mochila.

La salida es la misma que la del TGA.

htga.rb

Contiene la implementación de la clase en Ruby llamada *HTGA* que hereda de la clase BaseGA. Esta clase implementa todas las subrutinas descritas en [Tsai, Liu, Chou 2004] que son: *generación de la población inicial*, *selección por ruleta*, *cruce de un punto de corte*, cruce con *método Taguchi* y *mutación.*

Argumentos:
beta_values: Una cadena de caracteres indicando la manera en que se generarán los valores aleatorios usados en la generación de la población inicial (ver Algoritmo 4). Los posibles valores son ‘discrete’ o ‘uniform distribution’.
upper_bounds: Un arreglo de números reales que serán las cotas superiores de los genes.
lower_bounds: Un arreglo de números reales que serán las cotas inferiores de los genes.
pop_size: Entero que indica la cantidad de cromosomas .
cross_rate: Número real entre cero y uno que indica la tasa de cruce de los cromosomas.
mut_rate: Número real entre cero y uno que indica la tasa de mutación de los cromosomas.
num_genes: Entero que indica la cantidad de genes en un cromosoma.
continuous: Booleano que indica si el problema a solucionar es continuo o no.
selected_func: Entero que indica la función de prueba a resolver.
*is_negative_fi*t: Booleano que indica si en el problema a resolver se generan aptitudes negativas.

is_high_fit: Booleano que indica si el problema a resolver es de minimización o maximización.
max_generation: Entero que indica el número de máximo de generaciones permitidas.

Las salidas del *HTGA* son las mismas que las del *TGA*

htga_knapsack.rb

Contiene la implementación de la clase en Ruby llamada *HTGAKnapsack* que hereda de la clase *HTGA*. Esta clase adapta las subrutinas de su clase padre para resolver el problema de la mochila 0-1.

Argumentos:
pop_size: Entero que indica la cantidad de cromosomas.
num_genes: Entero que indica la cantidad de genes en un cromosoma.
optimun: El valor óptimo del problema.
max_generation: Entero que indica el número de máximo de generaciones permitidas.
number of constraints: La cantidad de restricciones de la mochila.
values: Arreglo de los beneficios.
max_weights: Arreglo con las constante al lado derecho de las inecuaciones de las restricciones.
weights: Arreglo bi-dimensional en el que cada cada fila contiene las constantes de las restricciones de la mochila.
cross_rate: Número real entre cero y uno que indica la tasa de cruce de los cromosomas.
mut_rate: Número real entre cero y uno que indica la tasa de mutación de los cromosomas.

La salida es la misma que la del *HTGA*.

ihtga.rb

Contiene la implementación de la clase en Ruby llamada *IHTGA* que hereda de la clase *HTGA*. Esta clase contiene la lógica de la subrutina descrita en [Poorjandaghi, 2014]. A excepción del paso 1 (ver Discusión), el *IHTGA* es una implementación fiel del algoritmo como es descrito por Poorjandaghi. Las subrutinas implementadas en *IHTGA* incluyen las mismas encontradas en el *HTGA* y las versiones mejoradas de *SNR*, el nuevo operador de mutación y el uso de la selección por *muestreo estocástico universal*.

Argumentos:
sub_chromosomes: Un arreglo con los cromosomas del subsistema.
lower_bounds: Un arreglo con las cotas inferiores de los genes del subsistema.
upper_bounds: Un arreglo con las cotas superiores de los genes del subsistema.

beta_values: Una cadena de caracteres indicando la manera en que se generarán los valores aleatorios usados en la generación de la población inicial (ver Algoritmo 4). Los posibles valores son 'discrete' o 'uniform distribution'.
pop_size: Entero que indica la cantidad de cromosomas .
cross_rate: Número real entre cero y uno que indica la tasa de cruce de los cromosomas.
mut_rate: Número real entre cero y uno que indica la tasa de mutación de los cromosomas.
continuous: Booleano que indica si el problema a solucionar es continuo o no.
selected_func: La función a optimizar.
*is_negative_fi*t: Booleano que indica si en el problema a resolver se generan aptitudes negativas.
is_high_fit: Booleano que indica si el problema a resolver es de minimización o maximización.
subsystem: El subsistema a resolver (objeto *Subsystem*).
mutation_prob: Número real entre cero y uno usado en la regla de mutación definida para el *CCHTGA* (ver Algoritmo 6).
taguchi_array: Arreglo bidimensional donde se almacena el *arreglo de Taguchi* seleccionado para realizar el cruce.

El algoritmo no produce salida alguna, ya que se dedica a cambiar el estado de las variables que le pasan como entrada.

cchtga.rb

Contiene la implementación de la clase en Ruby llamada CC*HTGA* que hereda de la clase BaseGA. Las subrutinas *implementadas en CCHTGA incluyen: generación de población inicial*, *división de las variables*, *agrupamiento al azar* de las variables en subsistemas, bucles de *cooperación-coevolución* y llamado al *IHTGA* para resolver problema en subsistemas.

Los argumentos y salidas son iguales a los del *HTGA*, exceptuando una salida adicional correspondiente al número de subsistemas utilizados.

test_functions.rb

Contiene la implementación de 15 funciones de prueba y los valores óptimos de las mismas, implementadas dentro de un módulo en Ruby llamado *TestFunctions*.

roulette.rb

Contiene la implementación como módulo de Ruby de la subrutina de selección de la ruleta llamado *Roulette*.

sus.rb

Contiene la implementación como módulo de Ruby llamado *SUS* de la subrutina del *muestreo estocástico universal*.

chromosome.rb

Contiene la implementación como clase de Ruby de la estructura de datos llamado *Chromosome,* que codifica las soluciones generadas en los algoritmos genéticos.

subsystem.rb

Contiene la implementación como clase de Ruby de la estructura de datos llamada *Subsystem,* que es utilizada en el *CCHTGA* para agrupar variables.

execute.rb

Contiene la implementación de una clase en Ruby llamada *TestRunner* y otra llamada *TestRunnerKnapsack* que leen las entradas de cada uno de los algoritmos genéticos a partir de archivos de texto plano con extensión .ini y retorna la salida de cada ejecución en un archivo con extensión .csv. Este archivo se ejecuta desde consola de la siguiente manera:

```
$ ruby execute.rb <algoritmo> <ruta al archivo de pruebas>
```

Ejemplo:

```
$ ruby execute.rb cchtga test_cases/cchtga/f2_test.ini
```

Una muestra de archivo de entrada para *TGA*, *HTGA* o *CCHTGA*:

```
function: 2
minimization: yes
continuous: yes
negative fitnesses: no
population size: 30
number of genes: 100
mutation rate: 0.7
crossover rate: 0.8
beta values: discrete
max number of generations: 10000
upper bounds: 100
lower bounds: -100
```

```
number of runs: 2
```

Una muestra de archivo de entrada para *TGAKnapsack* o *HTGAKnapsack*:

```
population size: 200
number of genes: 24
mutation rate: 0.02
crossover rate: 0.1
optimum: 1130
max number of generations: 1000
number of constraints: 1
values:
[150,200,60,60,30,70,15,40,75,20,50,1,35,160,45,40,10,30,10,70,80,12
,10,150]
max weights: [500]
weights:
[[9,153,15,27,23,11,24,73,43,7,4,90,13,50,68,39,52,32,48,42,22,18,30,20
0]]
number of runs: 1
```

Una muestra de archivo de salida:

```
best fitness,generation of best fitness,function evaluations of best
fitness,optimal value,relative error
7.362757364126615e-06,6985,383728,0,7.362757364237638e-06
2.663377660638844e-07,3582,196640,0,2.663377660638844e-07
```

Finalmente, se provee el repositorio en Github con el código fuente y la documentación del software en HTML*,* esta se encuentran en la carpeta *doc* y los archivos pueden ser consultados con el navegador.

Enlace a repositorio: https://github.com/Evalab-Univalle/TGA-HTGA-CCHTGA.git

10. Pruebas y Resultados

10.1 Funciones de Prueba

Las funciones de prueba para medir el desempeño de los algoritmos genéticos implementados en este trabajo fueron sacadas de dos fuentes primarias: [Tsai, Liu y Chou 2004] y [Jamil and Yang, 2013]. Fueron 15 las funciones seleccionadas teniendo en cuenta las propiedades de:

Separabilidad: una función es separable si sus variables son independientes. Si N es la cantidad de variables de una función y todas estas son independientes, es posible separar el problema en N partes, y todas pueden ser optimizadas de manera separada.

Modalidad: también conocido como picos, se refiere a la cantidad de máximos/mínimos locales o globales que tiene una función. Un función unimodal tiene un único valor óptimo y una multimodal tiene mínimo dos.

Todas las funciones pertenecen a *benchmarks* de optimización. Estas son continuas en el dominio proporcionado, el óptimo es denotado como $f(x^*)$ y se proveen, para algunas funciones, los gráficos para dos dimensiones ($N = 2$).

1. Función de Schwefel

   $f_1 = 418.9829 \cdot N - \sum_{i=1}^{N} x_i sin(\sqrt{|x_i|})$

   Descripción:
   - Multimodal
   - Separable
   - $x \in [-500, 500]^N$
   - $f(x^*) = 0$

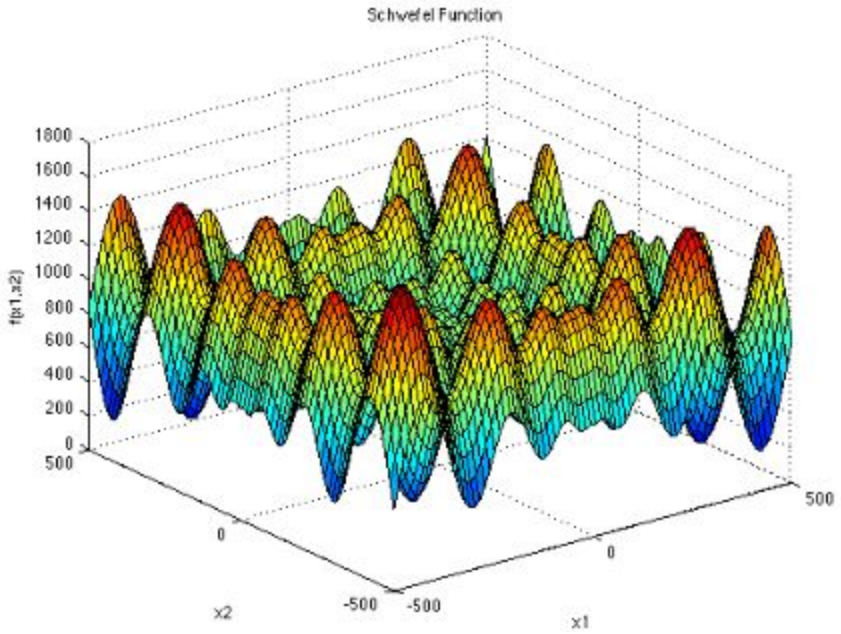

Figura 4: Surjanovic, S (2015) Schwefel function [Gráfico]. Recuperado de http://www.sfu.ca/~ssurjano/schwef.html

2. Función de Rastrigin

$$f_2 = 10 \cdot N + \sum_{i=1}^{N} (x_i^2 - 10 \cdot cos(2\pi x_i))$$

Descripción:

- Multimodal
- No separable
- $x \in [-5.12,\ 5.12]^N$
- $f(x^*) = 0$

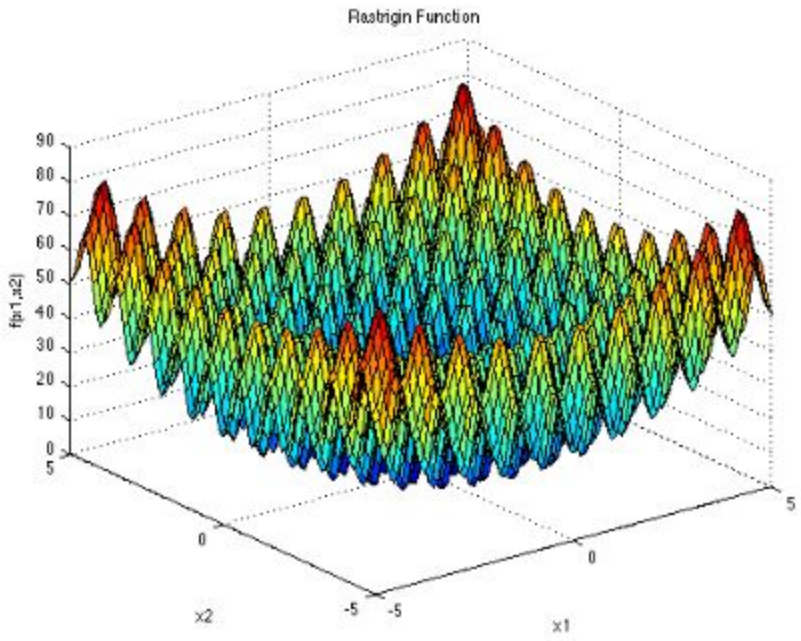

Figura 5: Surjanovic, S (2015) Rastrigin function [Gráfico]. Recuperado de http://www.sfu.ca/~ssurjano/rastr.html

3. Función de Ackley

$$f_3 = -20 \cdot exp\left(-0.2 \cdot \sqrt{\frac{1}{N}\sum_{i=1}^{N} x_i^2}\right) - exp\left(\frac{1}{N}\sum_{i=1}^{N} cos(2\pi x_i)\right) + 20 + exp(1)$$

Descripción:

- Multimodal
- No separable
- $x \in [-32,\ 32]^N$
- $f(x^*) = 0$

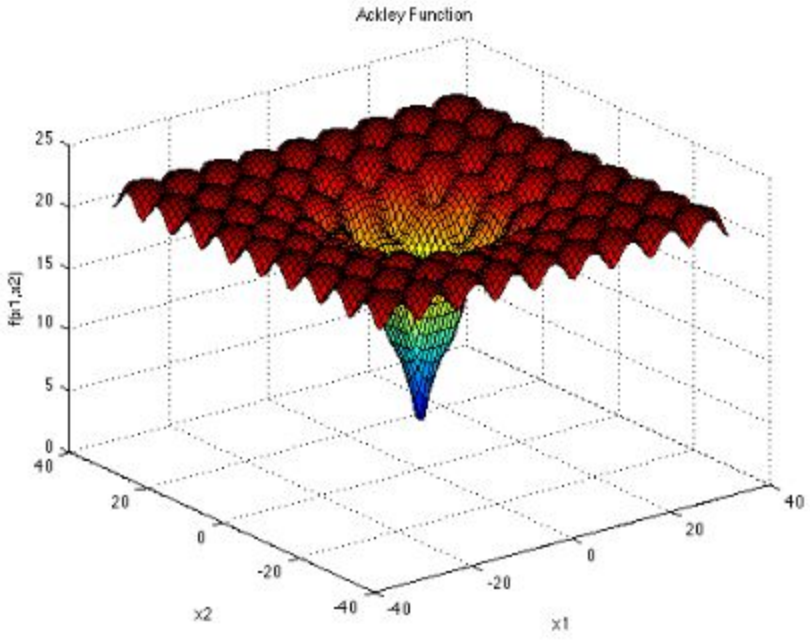

Figura 6: Surjanovic, S (2015) Ackley function [Gráfico]. Recuperado de http://www.sfu.ca/~ssurjano/ackley.html

4. Función de Griewank

$$f_4 = 1 + \frac{1}{4000}\sum_{i=1}^{N} x_i^2 - \prod_{i=1}^{N} cos\left(\frac{x_i}{\sqrt{i}}\right)$$

Descripción:

- Multimodal
- No separable
- $x \in [-600,\ 600]^N$
- $f(x^*) = 0$

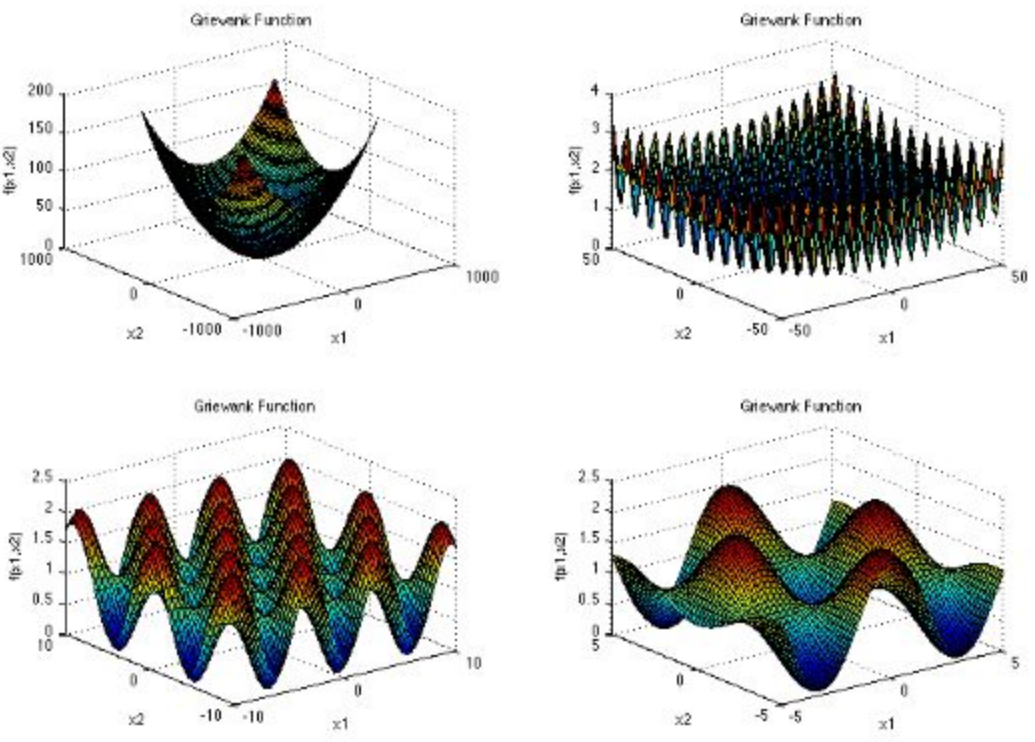

Figura 7: Surjanovic, S (2015) Griewank function [Gráfico]. Recuperado de http://www.sfu.ca/~ssurjano/griewank.html

5. Función de Levy #1

$$f_5 = \frac{\pi}{N}\left\{10\cdot sin^2(\pi y_1) + \sum_{i=1}^{N-1}(y_i-1)\left[1+10\cdot sin^2(\pi y_{i+1})\right] + (y_N-1)^2\right\} + \sum_{i=1}^{N} u(x_i,10,100,4),$$

$$y_i = 1+\tfrac{1}{4}(x_i+1),$$

$$u(x,a,k,m) = \begin{cases} k(x_i-a)^m & if \quad x_i > a \\ 0 & if \quad -a \le x_i \le a \\ k(-x_i-a)^m & if \quad x_i < -a \end{cases}$$

Descripción:

- Multimodal
- No separable
- $x \in [-50,\ 50]^N$
- $f(x^*) = 0$

6. Función de Levy # 2

$$f_6 = \frac{1}{10}\left\{ sin^2(3\pi x_1) + \sum_{i=1}^{N-1}(x_i - 1)^2\left[1 + sin^2(3\pi x_{i+1})\right] + (x_N - 1)^2\left[1 + sin^2(2\pi x_N)\right]\right\}$$
$$+ \sum_{i=1}^{n} u(x_i, 5, 100, 4)$$

Descripción:

- Multimodal
- No separable
- $x \in [-50, 50]^N$

7. Función de Michalewicz

$$f_7 = -\sum_{i=1}^{N} sin\,(x_i)\, sin^{20}\left(\frac{i \cdot x_i 2}{\pi}\right)$$

Descripción:

- Multimodal
- Separable
- $x \in [0,\ \pi]^N$
- $f(x^*) = -28.9263,\ N = 30 \quad f(x^*) = -99.2784,\ N = 100$

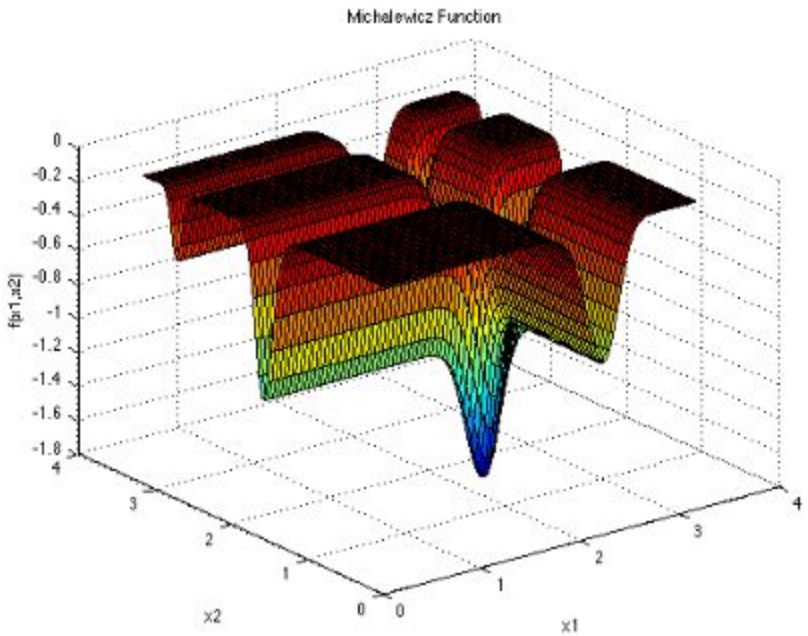

Figura 8: Surjanovic, S (2015) Michelewicz function [Gráfico]. Recuperado de http://www.sfu.ca/~ssurjano/michal.html

8. Función de Brown

$$f_8 = \sum_{i=1}^{N-1} (x_i^2)^{(x_{i+1}^2 + 1)} + (x_{i+1}^2)^{(x_i^2 + 1)}$$

Descripción:

- Unimodal
- No separable

- $x \in [-1,\ 4]^N$
- $f(x^*) = 0$

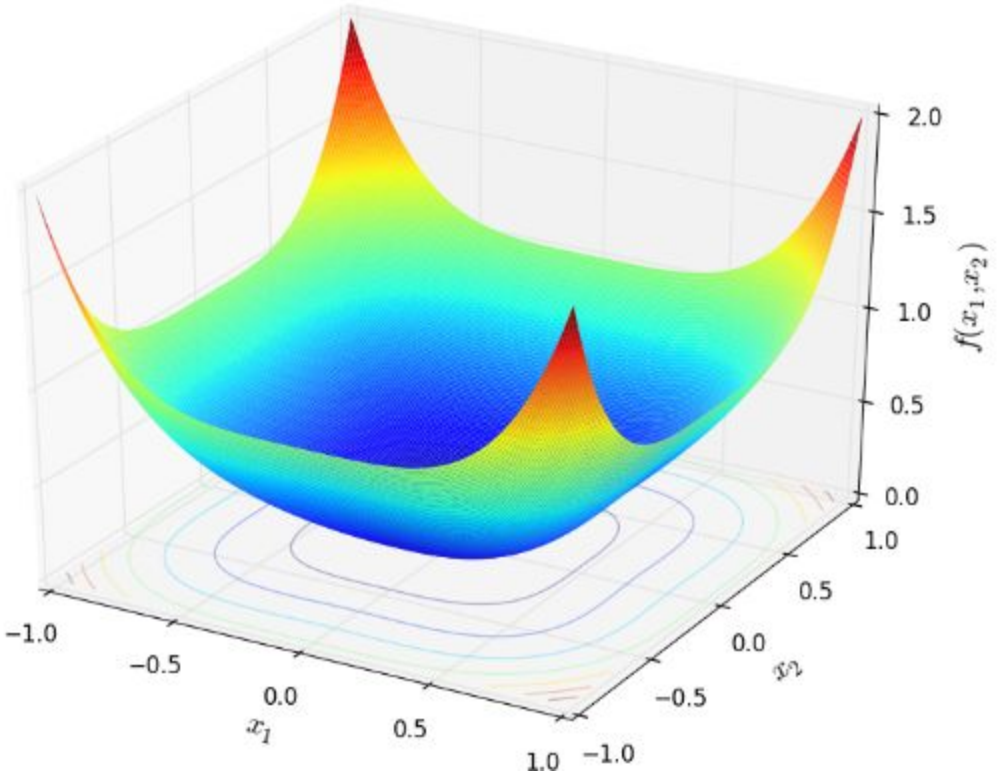

Figura 9: Anon (2013) Brown function [Gráfico]. Recuperado de: http://infinity77.net/global_optimization/test_functions_nd_B.html

9. Función Styblinki-Tang

$$f_9 = \frac{1}{N}\sum_{i=1}^{N}(x_i^4 - 16x_i^2 + 5x_i)$$

Descripción:

- Multimodal
- No separable
- $x \in [-5, 5]^N$
- $f(x^*)$ = -78.33236

10. Función de Rosenbrock

$$f_{10} = \sum_{j=1}^{N-1}\left[100\left(x_j^2 - x_{j+1}\right)^2 + \left(x_j - 1\right)^2\right]$$

Descripción:

- Unimodal
- No separable
- $x \in [-5,\ 10]^N$

- $f(x^*) = 0$

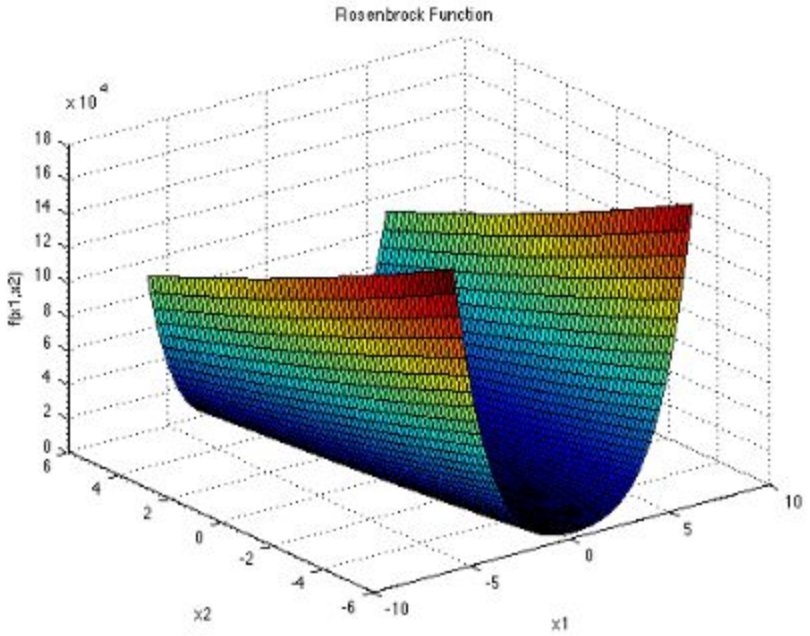

Figura 10: Surjanovic, S (2015) Rosenbrock function [Gráfico]. Recuperado de http://www.sfu.ca/~ssurjano/rosen.html

11. Función Sphere

$$f_{11} = \sum_{i=1}^{N} x_i^2$$

Descripción:

- Unimodal
- Separable
- $x \in [-100,\ 100]^N$
- $f(x^*) = 0$

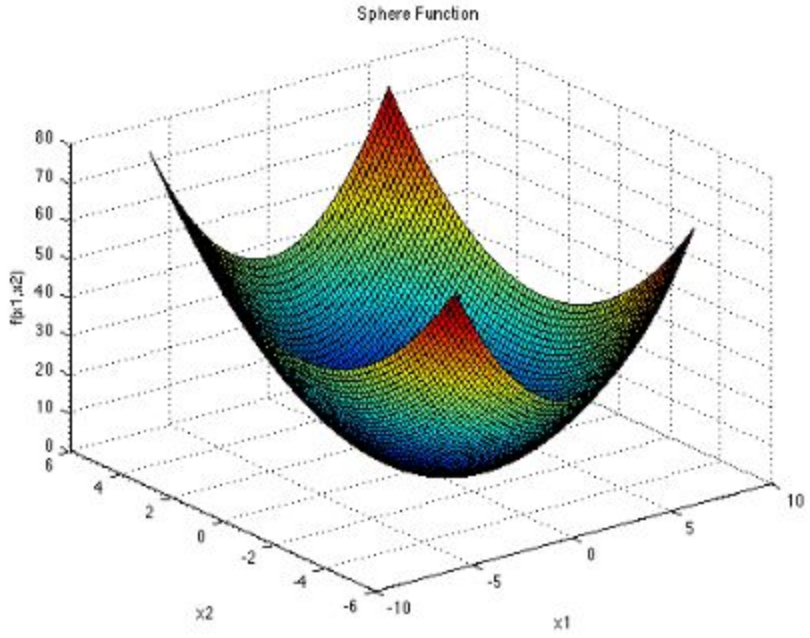

Figura 11: Surjanovic, S (2015) Sphere function [Gráfico]. Recuperado de http://www.sfu.ca/~ssurjano/spheref.html

12. Función Cuártica con Ruido.

$$f_{12} = \sum_{i=1}^{N} x_i^4 + random[0,\ 1)$$

Descripción:

- Unimodal
- Separable
- $x \in [-1.28,\ 1.28]^N$
- $f(x^*) = 0$

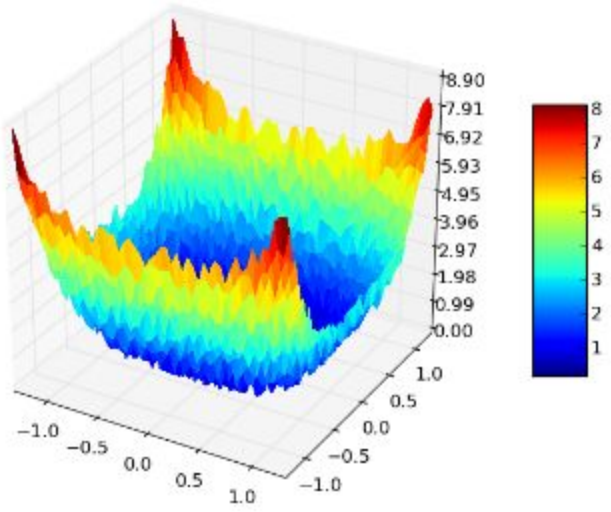

Figura 12: Anon (2013). Quartic function with noise [Grafico]. Recuperado de http://www.cs.unm.edu/~neal.holts/dga/benchmarkFunction/quartic.html

13. Función de Schwefel 2.22

$$f_{13} = \sum_{i=1}^{N} |x_i| + \prod_{i=1}^{N} |x_i|$$

Descripción:

- Unimodal
- No separable
- $x \in [-10,\ 10]^N$
- $f(x^*) = 0$

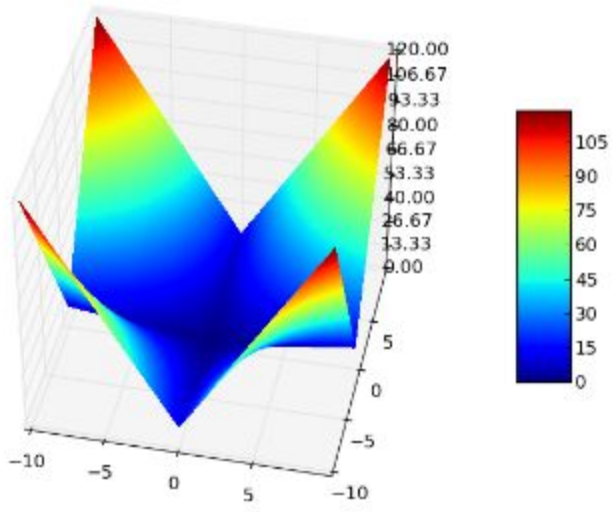

Figura 13: Anon (2013). Schwefel 2.22 function [Grafico]. Recuperado de http://www.cs.unm.edu/~neal.holts/dga/benchmarkFunction/schwefel222.html

14. Función de Schwefel 1.2

$$f_{14} = \sum_{i=1}^{N} \left(\sum_{j=1}^{i} x_j \right)^2$$

Descripción:

- No separable
- Unimodal
- $x \in [-100,\ 100]^N$
- $f(x^*) = 0$

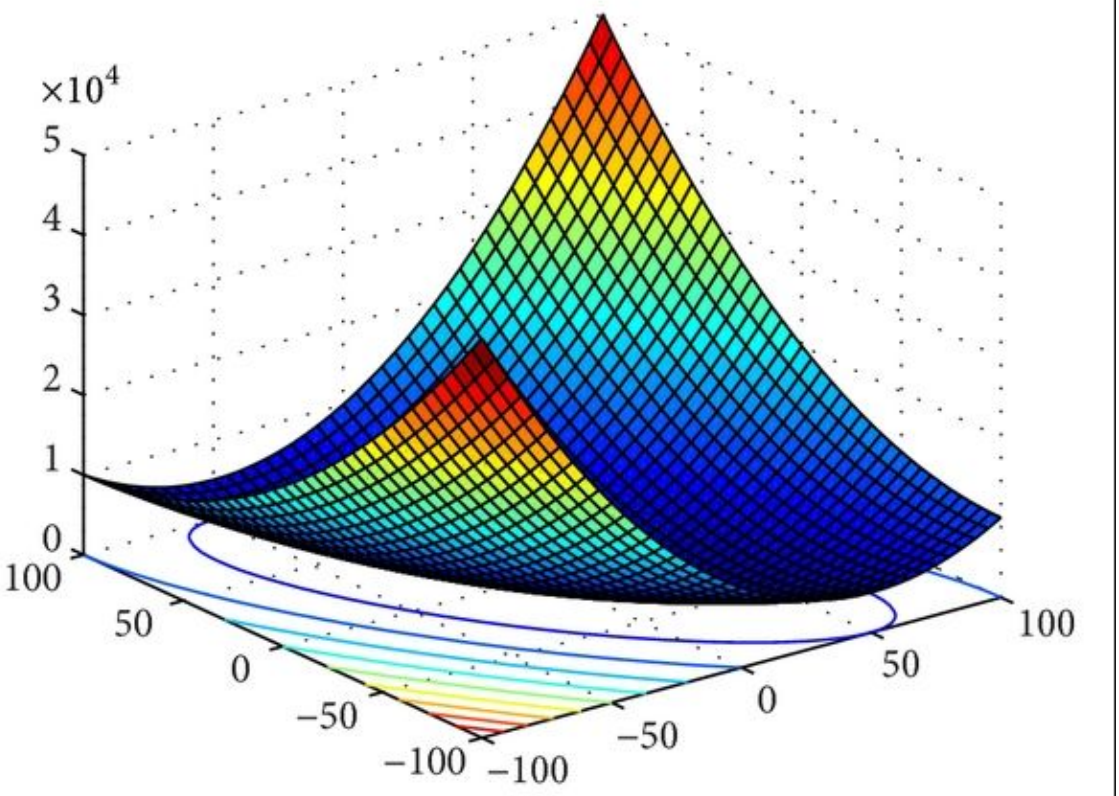

Figura 14: Anon (2016) Schwefel problem 1.2 [Gráfico]. Recuperado de https://www.hindawi.com/journals/jam/2013/653749/fig2/

15. Función de Schwefel 2.21

$f_{15} = Max\{|x_i|,\ i = 1, 2, ..., N\}$

Descripción:

- Unimodal
- Separable
- $x \in [-100,\ 100]^N$
- $f(x^*) = 0$

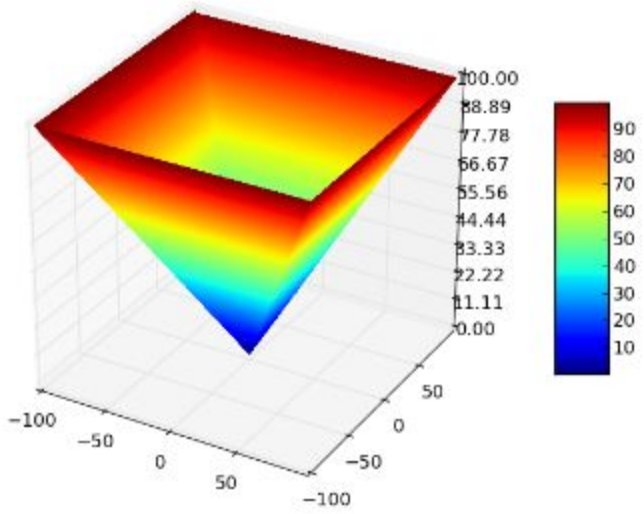

Figura 15: Anon (2013). Schwefel problem 2.21 [Gráfico]. Recuperado de http://www.cs.unm.edu/~neal.holts/dga/benchmarkFunction/schwefel221.html

Tabla 6. Resumen de las funciones de prueba

Función	Modalidad	Separabilidad	Óptimo
f_1	Multimodal	Separable	0
f_2	Multimodal	No separable	0
f_3	Multimodal	No separable	0
f_4	Multimodal	No separable	0
f_5	Multimodal	No separable	0
f_6	Multimodal	No separable	0
f_7	Multimodal	No separable	-28.9263, N=30 -99.2784, N=100
f_8	Unimodal	No separable	0
f_9	Multimodal	No separable	-78.33236

f_{10}	Unimodal	No separable	0
f_{11}	Unimodal	Separable	0
f_{12}	Unimodal	Separable	0
f_{13}	Unimodal	No separable	0
f_{14}	Unimodal	No separable	0
f_{15}	Unimodal	Separable	0

10.2 Resultados Computacionales y Comparativas

La ejecución de las pruebas se realizó en ambientes de 30, 100 y 1000 variables, y con los argumentos de entrada que más favorecen a cada algoritmo. Esto es para *TGA* una población de 200; para *HTGA* es población 200, tasa de cruce del 10% y tasa de mutación del 2%; para *CCHTGA* es población de 30, tasa de cruce del 80% y tasa de mutación del 70%. También, se cuenta con dos condiciones de parada para todos los algoritmos, una es un máximo de generaciones de 10000 y la otra es si el óptimo conocido ha sido alcanzado. Por último, cada función de prueba cuenta con 30 ejecuciones independientes y se registran el valor óptimo encontrado, el número de evaluaciones, y en el caso de *CCHTGA*, la cantidad de subsistemas.

Como el desempeño del *TGA* es tan pobre en todas las funciones de prueba, este se ejecutó solo hasta 100 variables. Por tanto, el enfoque principal de las comparativas es entre el *HTGA* y *CCHTGA*. A continuación, se exhiben los resultados de las pruebas, los mejores resultados están sombreados.

Tabla 7. Promedio y desviación estándar de la aptitud en dimension 30

Función	**TGA**	**HTGA**	**CCHTGA**
f_1	5.47E+03(1.50E+02)	2,38E+02(5.00E+02)	3.32E-04 (8.90E-04)
f_2	2.29E+01 (3.88E+00)	0.00E+00 (0.00E+00)	0.00E+00(0.00E+00)
f_3	4.65E+00 (6.22E-01)	2.66E-16 (0.00E+00)	2.66E-16 (0.00E+00)
f_4	6.49E+00 (2.47E+00)	0.00E+00 (0.00E+00)	0.00E+00 (0.00E+00)
f_5	5.81E+00 (1.93E+00)	7.55E-03 (5.18E-02)	1.09E-01 (4.46E-01)

f_6	4.36E+01 (2.11E+01)	2.07E-01 (9.07E-01)	1.55E-02 (7.67E-02)
f_7	-1.59E+01 (3.29E-01)	-3.25E+01 (4.87E-01)	-1.01E+01 (2.81E+00)
f_8	1.06E+01(5.36E+00)	0.00E+00 (0.00E+00)	9.76E-04 (8.91E-03)
f_9	-4.61E+01 (2.84E-01)	-4.13E+01 (1.51E+00)	-4.70E+01 (0.00E+00)
f_{10}	8.54E+02 (4.20E+02)	4.27E+01 (3.52E+01)	3.51E-05(3.07E-04)
f_{11}	5.98E+02 (2.26E+02)	0.00E+00 (0.00E+00)	0.00E+00 (0.00E+00)
f_{12}	7.13E-02 (3.31E-02)	1.79E-06 (2.52E-06)	1.04E-01(4.66E-02)
f_{13}	4.86E+00 (1.44E+00)	0.00E+00 (0.00E+00)	0.00E+00 (0.00E+00)
f_{14}	8.61E+03 (4.55E+03)	0.00E+00 (0.00E+00)	7.61E+03 (8.50E+03)
f_{15}	1.60E+01 (2.50E+00)	0.00E+00 (0.00E+00)	4.56E+00 (9.21E+00)

Tabla 8. Mejor aptitud en dimension 30

Función	**TGA**	**HTGA**	**CCHTGA (#grupos)**
f_1	5.47E+03	1.34E-02	3.82E-04 (todos)
f_2	2.56E+01	0.00E+00	0.00E+00 (todos)
f_3	6.59E+00	4.44E-16	4.44E-16 (todos)
f_4	6.94E+00	0.00E+00	0.00E+00(todos)
f_5	5.12E+00	1.57E-32	1.57E-32 (5,6,15,10)
f_6	3.31E+01	0.00E+00	1.35E-32(3,5,6,15,)
f_7	-2.71E+01	-9.63E+01	-2.27E+01(15)
f_8	9.87E+00	0.00E+00	0.00E+00(todos)
f_9	-7.74E+01	-7.17E+01	-7.83E+01(todos)
f_{10}	5.89E+02	2.86E+01	0.00E+00 (2, 6,10, 15)

f_{11}	5.31E+02	0.00E+00	0.00E+00 (todos)
f_{12}	4.68E-02	1.52E-07	9.89E-02(todos)
f_{13}	5.33E+00	0.00E+00	0.00E+00 (todos)
f_{14}	7.35E+03	0.00E+00	1.90E+03(10)
f_{15}	2.20E+01	0.00E+00	0.00E+00(5,6,10,15)

Tabla 9. Promedio y desviación estándar del número de evaluaciones en dimension 30

Función	**TGA**	**HTGA**	**CCHTGA**
f_1	2.41E+04(0.00E+00)	1.79E+05(1.01E+05)	2.59E+06 (1.21E+07)
f_2	2.41E+04(0.00E+00)	7.58E+02(1.71E+02)	8.13E+04 (5.82E+04)
f_3	2.41E+04 (0.00E+00)	5.12E+02 (1.02E+02)	3.38E+04 (1.06E+04)
f_4	2.41E+04 (0.00E+00)	8.26E+02 (4.68E+02)	5.51E+05 (1.88E+06)
f_5	2.41E+04(0.00E+00)	2.71E+05 (5.51E+04)	7.26E+06(1.01E+07)
f_6	2.41E+04(0.00E+00)	2.34E+05(1.44E+05)	5.49E+06(4.35E+06)
f_7	2.41E+04(0.00E+00)	1.84E+05(7.76E+03)	7.50E+06(1.10E+07)
f_8	2.41E+04 (0.00E+00)	1.58E+03 (7.85E+02)	4.18E+06 (1.02E+07)
f_9	2.41E+04(0.00E+00)	2.08E+04(1.18E+03)	1.56E+06(1.74E+06)
f_{10}	2.41E+04 (0.00E+00)	1.18E+05(1.83E+05)	5.32E+06 (7.27E+06)
f_{11}	2.41E+04 (0.00E+00)	7.52E+02 (1.65E+02)	1.23E+05 (9.91E+04)
f_{12}	2.41E+04 (0.00E+00)	1.59E+05 (1.25E+05)	8.10E+06 (1.15E+07)
f_{13}	2.41E+04(0.00E+00)	6.80E+02(1.38E+02)	5.97E+05(1.69E+06)
f_{14}	2.41E+04(0.00E+00)	1.38E+03(7.40E+02)	8.20E+06(1.58E+07)
f_{15}	2.41E+04 (0.00E+00)	7.09E+02 (2.09E+02)	1.29E+06 (3.04E+06)

Tabla 10. Promedio y desviación estándar de la aptitud en dimension 100

Función	TGA	HTGA	CCHTGA
f_1	9.38E+03(5.86E+02)	1.11E+03(2.40E+03)	9.09E+02(1.80E+03)
f_2	3.51E+02(1.83E+01)	0.00E+00(0.00E+00)	2.15E+01(5.17E+01)
f_3	1.32E+01(4.76E-01)	2.66E-16(0.00E+00)	1.47E+00 (2.64E+00)
f_4	1.63E+02(1.93E+01)	1.82E-02(8.10E-02)	3.13E+01(5.38E+01)
f_5	1.46E+02(3.64E+01)	6.23E-02(3.36E-01)	3.77E+01(4.04E+01)
f_6	3.92E+03(7.40E+02)	3.10E+00(4.67E+00)	5.17E+02(1.40E+03)
f_7	-7.17E+01(1.94E+00)	-5.73E+01 (7.86E-01)	-2.36E+01(1.11E+01)
f_8	2.17E+12 (1.46E+13)	0.00E+00(0.00E+00)	1.73E+02(4.26E+02)
f_9	-7.03E+01(8.65E-01)	-5.01E+01(0.00E+00)	-4.46E+01(3.51E+00)
f_{10}	7.37E+04(1.54E+04)	5.90E+01(3.16E-01)	7.70E+04(2.05E+05)
f_{11}	1.80E+04(1.97E+03)	0.00E+00(0.00E+00)	2.50E+03(5.93E+03)
f_{12}	1.08E+00(1.93E-01)	8.23E-05(3.29E-04)	1.10E+00(6.49E-01)
f_{13}	7.37E+01(5.13E+00)	0.00E+00(0.00E+00)	2.10E+01(3.32E+01)
f_{14}	1.85E+05(2.51E+04)	9.54E-02(3.15E-01)	1.71E+05(6.28E+04)
f_{15}	3.94E+01(3.43E+00)	1.12E+01(5.07E+00)	2.95E+01(2.03E+01)

Tabla 11. Mejor aptitud en dimension 100

Función	TGA	HTGA	CCHTGA(#grupos)
f_1	7.94E+03	1.29E-03	1.27E-03 (4,5)
f_2	3.14E+02	0.00E+00	0.00E+00 (2,4,5)
f_3	1.24E+01	4.44E-16	4.44E-16(2,4,5,10)

f_4	1.32E+02	0.00E+00	0.00E+00(2,4,5)
f_5	9.73E+01	4.83E-10	4.83E-06 (2)
f_6	2.53E+03	2.70E-05	1.35E-32(2,4)
f_7	-7.56E+01	-9.69E+01	-6.77E+01(50)
f_8	8.58E+03	0.00E+00	3.40E+01 (10)
f_9	-7.27E+01	-7.83E+01	-7.83E+01(2,5)
f_{10}	4.96E+04	9.80E+01	0.00E+00 (2)
f_{11}	1.40E+04	0.00E+00	0.00E+00(2,4,5)
f_{12}	7.22E-01	7.11E-07	6.31E-01(2)
f_{13}	6.23E+01	0.00E+00	0.00E+00(2,4,5)
f_{14}	1.36E+05	2.20E-04	1.93E+05 (20)
f_{15}	5.74E+01	0.00E+00	0.00E+00(10,5)

Tabla 12. Promedio y desviación estándar del número de evaluaciones en dimension 100

Función	**TGA**	**HTGA**	**CCHTGA**
f_1	4.02E+04 (0.00E+00)	2.41E+05(2.30E+05)	2.14E+07(3.08E+07)
f_2	4.02E+04 (0.00E+00)	3.10E+03(3.43E+03)	1.21E+07(2.84E+07)
f_3	4.02E+04 (0.00E+00)	2.50E+03(2.95E+03)	2.14E+07(4.30E+07)
f_4	4.02E+04 (0.00E+00)	2.70E+04(1.25E+05)	1.63E+07(3.60E+07)
f_5	4.02E+04 (0.00E+00)	3.08E+05(1.38E+05)	1.79E+07(2.46E+07)
f_6	4.02E+04 (0.00E+00)	1.78E+05(2.75E+05)	1.77E+07(2.43E+07)
f_7	4.02E+04 (0.00E+00)	3.26E+05(6.17E+03)	1.55E+07(2.23E+07)
f_8	4.02E+04(1.46E+13)	4.30E+03(4.52E+03)	1.69E+07(2.94E+07)

f_9	4.02E+04 (0.00E+00)	3.52E+05(5.01E+02)	2.43E+07(3.43E+07)
f_{10}	4.02E+04 (0.00E+00)	3.30E+05(3.65E+02)	1.93E+07(3.54E+07)
f_{11}	4.02E+04 (0.00E+00)	2.90E+03(3.23E+03)	1.18E+07(3.06E+07)
f_{12}	4.02E+04 (0.00E+00)	2.42E+05(1.20E+05)	1.67E+07(2.66E+07)
f_{13}	4.02E+04 (0.00E+00)	3.26E+03(3.46E+03)	1.40E+07(2.18E+07)
f_{14}	4.02E+04 (0.00E+00)	3.30E+05(7.53E+02)	1.92E+07(3.13E+07)
f_{15}	4.02E+04 (0.00E+00)	1.64E+03(2.99E+03)	9.03E+06(3.41E+07)

Tabla 13. Promedio y desviación estándar de aptitud en dimensión 1000

Función	**HTGA**	**CCHTGA**
f_1	3.20E+04(6.81E+02)	2.19E+04(3.31E+04)
f_2	0.00E+00(0.00E+00)	5.63E+02(9.74E+02)
f_3	2.66E-16(0.00E+00)	4.24E+00(5.62E+00)
f_4	8.76E-02(2.44E-01)	7.22E+02(1.11E+03)
f_5	2.41E-01(6.15E-01)	6.65E+07(8.43E+07)
f_6	5.79E+01(4.31E+00)	1.25E+08(1.47E+08)
f_7	-7.71E+01(3.24E+00)	-7.62E+01(5.42E+00)
f_8	0.00E+00(0.00E+00)	1.51E+08(7.30E+15)
f_9	-4.68E+01(2.43E-02)	-4.39E+01(4.71E+00)
f_{10}	6.08E+02(1.82E+01)	2.63E+06(3.09E+06)
f_{11}	0.00E+00(0.00E+00)	1.03E+05(1.41E+05)
f_{12}	0.00E+00(0.00E+00)	2.58E+01(1.61E+01)
f_{13}	0.00E+00(0.00E+00)	3.20E+02 (4.60E+02)

f_{14}	3.97E+04(3.51E+04)	
f_{15}	6.00E+01(0.00E+00)	5.27E+01(1.02E+01)

Tabla 14. Mejores aptitudes en dimensión 10000

Función	**HTGA**	**CCHTGA(#grupos)**
f_1	5.22E+04	1.92E+01(2)
f_2	0.00E+00	0.00E+00(2)
f_3	4.44E-16	4.44E-16(5,2)
f_4	0.00E+00	0.00E+00(2)
f_5	7.88E-09	3.36E-03(2)
f_6	8.21E+01	3.37E-02(2)
f_7	-1.37E+02	-1.40E+02(250)
f_8	0.00E+00	3.42E+03 (200)
f_9	-7.80E+01	-7.83E+01(2)
f_{10}	1.00E+03	0.00E+00(2)
f_{11}	0.00E+00	0.00E+00(2)
f_{12}	0.00E+00	5.19E+00(4)
f_{13}	0.00E+00	0.00E+00 (2)
f_{14}	2.38E+04	
f_{15}	1.00E+02	7.60E+01(100)

Tabla 15. Promedio y desviación estándar del número de evaluaciones en dimensión 1000

Función	HTGA	CCHTGA
f_1	3.30E+05(2.89E+02)	1.63E+08(3.88E+08)
f_2	6.19E+04(3.70E+04)	1.87E+08(4.17E+08)
f_3	5.85E+04(3.59E+04)	9.76E+07(1.32E+08)
f_4	1.02E+05(1.26E+05)	3.16E+07(1.27E+08)
f_5	2.32E+05(2.54E+05)	1.03E+08(1.55E+08)
f_6	2.34E+05(2.12E+05)	9.32E+07(1.39E+08)
f_7	1.53E+02(2.53E+00)	1.40E+04(1.35E+04)
f_8	7.28E+04(4.86E+04)	1.01E+08(1.51E+08)
f_9	3.30E+05(3.32E+02)	1.35E+08(2.05E+08)
f_{10}	3.30E+05(1.97E+02)	1.06E+08(1.05E+08)
f_{11}	6.50E+04(3.51E+04)	1.51E+08(3.73E+08)
f_{12}	6.46E+04(3.36E+04)	1.61E+08(3.29E+08)
f_{13}	6.99E+04(3.39E+04)	1.22E+08(1.95E+08)
f_{14}	3.30E+05(4.14E+02)	
f_{15}	1.53E+02(2.38E+00)	8.45E+06(6.68E+07)

Con respecto al *desempeño computacional* , las tablas 9, 12 y 15 muestran que el *HTGA* es menos costoso que el *CCHTGA,* esto se debe a que el segundo tiene tasas de cruce y mutación más altas, y que este además de evaluar cromosomas, también evalúa sub cromosomas . Pero el *TGA* es el que menos evaluaciones realiza, debido a que es el más sencillo de los tres algoritmos. Y, se evidencia que el *TGA* tiene un *desempeño computacional* casi determinista, ya que casi siempre encuentra su mejor aproximación con la misma cantidad de evaluaciones.

Con respecto a la *calidad de solución* y la *robustez* las tablas 7, 10, 13 muestran que el *HTGA* ofrece la mejor calidad de solución a una baja desviación estándar, también que el *CCHTGA* ofrece buenos resultados en dimensión 30 sin importar que grupos use, pero no tan buenos en dimensiones 100 y 1000 (el efecto de los agrupamientos no lo favorece), aunque empata con el *HTGA* en muchas instancias. Y, que el en *TGA* ofrece la peor calidad de solución con la mayor desviación, exceptuando los casos de f_7 y f_9 (las únicas con óptimos negativos) en dimensión 100, en las que se acercó bastante al valor .

Los resultados de las tablas 8, 11 y 14 registran los mejores valores encontrados por los algoritmos, y en el caso del *CCHTGA*, se registran los grupos que generaron dicha solución. En este caso el CCHTGA es el que arroja los mejores valores, y en el resto casi siempre empata con el *HTGA*. También, se evidencia que existen maneras de agrupamiento en el *CCHTGA* que producen resultados óptimos o cercanos al óptimo. Por ejemplo, en promedio para f_{10} en dimensiones 100 y 1000 los mejores resultados los ofreció el *HTGA*, pero este último nunca se acercó al óptimo, cosa que el *CCHTGA* usando dos grupos si pudo resolver. Además de este caso, la evidencia parece mostrar que en dimensiones 100 y 1000, el CCHTGA con dos grupos (de 50 y 100 genes para dimension 100 y 1000 respectivamente), parece dar los mejores resultados, aunque las razones exactas son desconocidas, mientras que en dimensión 30 los efectos del agrupamiento no afectan mucho al resultado.

Un caso especial es el de f_7 o función de *Michalewicz*, ya que el óptimo de esta cambia con su dimensión, y según [ECAI, 2012] los valores óptimos siempre son aproximados usando algoritmos estocásticos. En la búsqueda bibliográfica, para este trabajo sólo se encontraron los valores óptimos para dimensiones 30 y 100 (ver Tabla 6), por tanto se ofrece el menor valor calculado por el CC*HTGA,*

$$f(x^*) = -139.696201,$$

como valor óptimo para esta función en dimension 1000.

Otro caso especial, es el de f_{14} o función de Schwefel 1.2, que resulta problemático de resolver para el *CCHTGA*, especialmente en dimensión 1000. Ya que, durante más de 9 dias, solo logró terminar 4 de las 30 ejecuciones dispuestas, a pesar de que cada una de las ejecuciones se corría en paralelo en núcleos de procesamiento distintos. Por esta razón no se disponen de datos sobre esta función en dimensión 1000.

Un comportamiento interesante se presenta en las funciones f_{10} y f_{15}, y es que estas durante la ejecución no presenta el comportamiento de convergencia gradual de las demás, es decir, generalmente a medida que pasan las generaciones se nota una mejora continua de las aptitudes de los cromosomas en las funciones de prueba, pero en el caso

de estas, les toma varias generaciones para empezar a converger, pueden permanecer estancadas en un valor varias generaciones y de repente empezar aproximarse al óptimo.

11. Discusión

Al momento de implementar los algoritmos estudiados en este trabajo, se presentaron algunos problemas por diversas razones, ya sea por un error de redacción o tipográfico de los autores que nadie notó, algún vacío dejado sobre alguna sección del artículo o alguna contradicción presentada. A continuación se documentan las dificultades presentadas a la hora de implementar los algoritmos del *HTGA* y especialmente el del *CCHTGA*.

Con el *HTGA*, solo se presentó un inconveniente: los autores afirman que el método de selección utilizado es el de la ruleta, pero no entran en más detalles. Existen muchas variantes dentro del método de la ruleta, especialmente si se trabajan con aptitudes negativas como es en este caso. A la hora de implementar esta subrutina se optó por usar una variante de la ruleta que linealiza las aptitudes de los cromosomas para normalizar todos los valores negativos a positivos y evita los efectos de los denominados *super individuos* (cromosomas con una aptitud relativamente mucho mejor que el resto de la población)[Cole, 2009]. Para implementar este mecanismo, se toma el clásico algoritmo de la ruleta, pero antes de calcular la probabilidad del cromosoma se aplica la siguiente regla para modificar la aptitud:

$$base = fit_{max} + fit\_factor \times (fit_{max} - fit_{min}) \quad \textbf{(17)}$$

$$fit_j = base - fit_j \quad \textbf{(18)}$$

Donde:

fit_j es la aptitud del j-ésimo cromosoma

fit_{max} es la máxima aptitud de la población

fit_{min} es la mínima aptitud de la población

fit_factor es un número entre cero y uno que es asignado arbitrariamente, en este caso se dejó como uno.

Una observación que vale la pena hacer sobre el *HTGA* tiene que ver con cómo es descrito en el artículo original, en este, el paso cuatro, correspondiente a la selección del *arreglo ortogonal*, entra dentro del bucle general del algoritmo, es decir, que de acuerdo a la descripción, el arreglo se debe seleccionar en cada generación, lo cual es innecesario y por tanto subóptimo. Ya que el arreglo se selecciona en base a la dimensión del problema que está siendo resuelto, que no cambia durante la ejecución. Este paso solo se debería hacer una vez, ya sea antes o después de la generación de la población inicial.

Ahora, en el caso del *CCHTGA* se encontraron muchos inconvenientes, puntualmente:

- Parece haber un error tipográfico en la definición de la nueva regla de mutación (ver Algoritmo 6), la regla originalmente era:

  $$x_i = b_i + (r \times 2 - 1) \times |y_i - \hat{y}_i| \quad \textbf{(19)}$$

  Pero en ningún momento se define qué es b_i. Como a esta regla se le describe como "de explotación cerca al valor de óptimo" se llega a la conjetura de que la

intención del autor era escribir x_i en lugar de b_i. Como este no respondió ninguno de los correos enviados pidiendo una aclaración, no se tuvo más opción que seguir esta conjetura.

- En la práctica la regla de corrección de genes no funcionó como se describe, ya que existían casos en los que al aplicar la regla sobre un gen dañado, éste no se arreglaba. Para lograr que funcionara era necesario aplicar la función recursivamente hasta que el gen regresará a los límites correspondientes. Además, el artículo menciona que la corrección solo se debe aplicar al mejor cromosoma actual (ver Algoritmo 8, línea diecisiete) lo cual en la práctica no corrige todos los cromosomas dañados, porque todos estos tienen alguna probabilidad de ser mutados y que sus genes se salgan de los límites. Lo que funcionó en este trabajo fue aplicar la corrección de los genes a todos los cromosomas que son mutados, no solo al mejor de la generación.
- Parece haber un error de identación en el pseudocódigo del *CCHTGA* (ver Algoritmo 8), pues la validación para actualizar el mejor cromosoma general con los genes del mejor cromosoma de algún subsistema (líneas quince a dieciséis) debería hacerse sólo al final del bucle que itera sobre los subsistemas (bucle exterior) no al final del bucle que itera sobre los cromosomas (bucle interior). Tiene más sentido que sea así ya que el iterador i no se necesita para esta validación.
- Hay una contradicción aparente sobre la *generación de la población inicial* del *CCHTGA*. Al principio del artículo, se dice que esta subrutina es manejada dentro de cada subsistema como primer paso del *HTGA* mejorado (*IHTGA*), pero luego en el pseudocódigo del *CCHTGA* esta subrutina se mueve al inicio del algoritmo (ver Algoritmo 8, línea dos) antes de que algún subsistema sea creado. En la práctica, colocar la generación de población inicial en el *IHTGA* cada vez que intentaba solucionar un subsistema sólo produjo una pérdida de la presión selectiva, lo que causa que el algoritmo no mejore con el paso de las generaciones. Esto tiene sentido, ya que si una población inicial es generada en cada generación al inicio de la subrutina, se pierde el historial de las generaciones pasadas. Por tanto, se optó por dejar que la población inicial sea generada solo una vez al inicio del algoritmo.
- Se usa el término “mejor experiencia del cromosoma” pero jamás se explica de manera explícita a que se refiere. Se deduce entonces que a lo que se refiere con esto es a los valores de los genes que para el cromosoma han producido la mejor aptitud hasta el momento.
- No se menciona explícitamente donde se debe actualizar el mejor cromosoma actual ni cuándo. Tampoco se menciona que se debe hacer durante las dos primeras iteraciones del algoritmo. De nuevo, como el autor no respondió ninguno de los correos, por lo que se decidió por actualizar el mejor cromosoma actual justo al final del bucle más externo (ver Algoritmo 8, línea 5) y que durante las dos primeras generaciones el algoritmo no hiciera nada más que inicializar variables y actualizarlas.

12. Conclusiones

1. La adaptación de los algoritmos para que funcionen con problemas discretos consiste en dos cosas: la primera es que se necesita modificar solamente las subrutinas que cambian los valores de los genes y que producen valores continuos (cruce, mutación y generación de población inicial) para que produzcan valores discretos. La segunda es desarrollar un método de corrección en caso de que las modificaciones a las subrutinas mencionadas puedan producir cromosomas inválidos.
2. El proceso de hacer pruebas a software con características estocásticas es desafiante, ya que el código de éste a menudo contiene *bugs* que son intermitentes y por tanto resultan muy difíciles de reproducir en el ambiente de pruebas, especialmente empleando aquellas que chequean el comportamiento de varios componentes juntos, como es el caso de las pruebas funcionales, de integración o de aceptación (las realizadas para este trabajo). Un enfoque más prometedor parece ser el de usar en el ambiente de pruebas una semilla constante para el *generador de números pseudoaleatorios*, combinado con pruebas unitarias y el uso de *mocks* para aislar y controlar mejor el comportamiento aleatorio en el sistema.
3. Aunque las características del lenguaje Ruby proveen una manera elegante, estructurada y sencilla de programar orientado objetos, además de que el código generado es sencillo de leer, este tiene serios problemas de desempeño que dificulta la ejecución de los algoritmos con entradas de grandes dimensiones. También, se sufre por la falta de una verdadera concurrencia o paralelismo en el lenguaje que permita aprovechar el paralelismo inherente de los algoritmos (por ejemplo, la subrutina del *IHTGA*) y que obliga a que las ejecuciones tengan que ser secuenciales.

   Existen varias implementaciones del intérprete de Ruby, y la más problemática en cuanto a desempeño es el CRuby o MRI (Intérprete Ruby de Martz por sus siglas en inglés), y que es la implementación por defecto de Ruby en lenguaje C, y la usada en este trabajo. Pero, existen algunas implementaciones que tienen características como mejor manejo de memoria y soporte de un verdadero paralelismo que pudieran mejorar el desempeño de las ejecuciones, como es el caso de JRuby (Intérprete de Ruby que es compatible con la *máquina virtual de Java*) y Rubinius (implementación de Ruby sobre una *máquina virtual de bajo nive*l).

   Una manera de lidiar con la lentitud del programa, es utilizar el software de HTC (Computación de Alto Rendimiento por sus siglas en inglés) HTCondor, que si bien no hace que el programa se ejecute en paralelo, si permite realizar cada ejecución de manera independiente y paralela en distintos núcleos de procesamiento.

4. Se constata que la integración del método Taguchi en el operador de cruce de los algoritmos genéticos en efecto produce, generalmente, resultados con baja desviación y por tanto hace que el algoritmo genético resultante sea más robusto que los algoritmos genéticos sencillos como los que se enseñan en la Universidad del Valle.
5. La implementación del *HTGA* arrojó resultados similares a los reportados en el artículo original, con excepción de f_{10}, que jamás pudo generar un resultado aproximado al óptimo. Esto probablemente se deba al proceso de linealización utilizado en la *selección por ruleta* (Ecuaciones 17 y 18). Por otro lado, la implementación del *CCHTGA* no arrojó aptitudes promedio tan buenas como las reportadas en el artículo original ni demostró ser particularmente mejor que el *HTGA* para resolver problemas no separables, probablemente se deba a las dificultades en el análisis del algoritmo y que afectaron la implementación del mismo (ver Discusión). Y, se encontró que el algoritmo más robusto, con mejor desempeño computacional y con mejor calidad de solución es el *HTGA* para todas las dimensiones.
6. El desempeño de un algoritmo genético en un problema particular y en una dimensión particular, es un pobre indicador de cómo se comportará el mismo en otras dimensiones o en otros tipos de problema. Basta ver el caso de f_7 y f_9, que en dimensión 100 fueron mejor resueltos por el *TGA* (ver Tabla 10), a pesar de que en el resto de las dimensiones el *HTGA* y el *CCHTGA* encontraron mejores óptimos. Y el caso de f_{15}, que se es resuelto fácilmente por *HTGA* en dimensión 30, pero en dimensión 100 y 1000 no es capaz. Por tanto, tal y como se deduce del teorema del *No Free Lunch,* el desempeño de los algoritmos de búsqueda, depende de los problemas que están resolviendo.

13. Trabajos Futuros

1. Debido a la imposibilidad de paralelizar la ejecución de los algoritmos con el intérprete CRuby, un trabajo futuro sería cambiar este por JRuby para permitir este tipo de ejecuciones.
2. Durante el desarrollo del proyecto, se logró adaptar los algoritmos para que funcionaran con otro tipo de problemas distintos a los puramente continuos y numéricos. Como trabajo futuro se pretende buscar cómo adaptar los algoritmos a otros problemas de este tipo, como fuese el problema del viajero o el de la multiplicación de matrices.
3. Como los resultados de la implementación del CCHTGA en este proyecto, difieren tanto de los resultados en el artículo original, se propone continuar con implementaciones alternativas o nuevas estrategias que puedan mejorar el desempeño de este algoritmo.

14. Referencias

Tsai, J., Liu, T., & Chou, J. (2004). Hybrid Taguchi-genetic algorithm for global numerical optimization. *Evolutionary Computation, IEEE Transactions on*, *8*(4), 365-377.

Omidvar, Mohammad Nabi et al. "Cooperative co-evolution for large scale optimization through more frequent random grouping." *IEEE Congress on Evolutionary Computation* 18 Jul. 2010: 1-8.

Yang, Zhenyu, Ke Tang, and Xin Yao. "Large scale evolutionary optimization using cooperative coevolution." *Information Sciences* 178.15 (2008): 2985-2999.

Gen, Mitsuo, and Runwei Cheng. *Genetic algorithms and engineering optimization*. John Wiley & Sons, 2000.

Leung, Yiu-Wing, and Yuping Wang. "An orthogonal genetic algorithm with quantization for global numerical optimization." *Evolutionary Computation, IEEE Transactions on* 5.1 (2001): 41-53.

Stjepan Picek, Marin Golub, Domagoj Jakobovic. (2012). Influence of the Crossover Operator in the Performance of the Hybrid Taguchi GA. 2015, de IEEE Transactions on Evolutionary Computation Sitio web: https://bib.irb.hr/datoteka/588565.bare_conf.pdf

Recioui, Abdelmadjid, and Hamid Bentarzi. "Capacity optimization of MIMO wireless communication systems using a hybrid genetic-Taguchi algorithm." *Wireless personal communications* 71.2 (2013): 1003-1019.

Chen, Chien-Sheng et al. "The Hybrid Taguchi-Genetic Algorithm for Mobile Location." *International Journal of Distributed Sensor Networks* 2014 (2014).

Yang, Ching-I, Jyh-Horng Chou, and Ching-Kao Chang. "Hybrid Taguchi-based genetic algorithm for flowshop scheduling problem." *International Journal of Innovative Computing, Information and Control* 9.3 (2013): 1045-1063.

Tsai, Jinn-Tsong, Jyh-Horng Chou, and Tung-Kuan Liu. "Tuning the structure and parameters of a neural network by using hybrid Taguchi-genetic algorithm." *Neural Networks, IEEE Transactions on* 17.1 (2006): 69-80.

Poorjandaghi, Seyyed Saeed, and Ahmad Afshar. "A Robust Evolutionary Algorithm for Large Scale Optimization." *World Congress* 24 Aug. 2014: 7037-7042.

Picek, S., Golub, M., & Jakobovic, D. (2012, June). Influence of the crossover operator in the performance of the hybrid Taguchi GA. In *Evolutionary Computation (CEC), 2012 IEEE Congress on* (pp. 1-8). IEEE.

Department Of Psychology -- University of Nebraska-Lincoln. Friedman's Two-way Analysis of Variance by Ranks. Available at http://psych.unl.edu/psycrs/handcomp/hcfried.PDF

J. Derrac, S. Garcia, D. Molina, and F. Herrera, "A practical tutorial on the use of nonparametric statistical tests as a methodology for comparing evolutionary and swarm intelligence algorithms," Swarm and Evolutionary Computation, vol. 1, pp. 3–18, 2011.

N. Dewangan (n.d) a detailed Study of 4G in Wireless Communication: Looking insight in issues in Ofdm. Retrieved from: <https://books.google.com/books?id=84RdAgAAQBAJ&pg=PP1&lpg=PP1&dq=A+Detailed+Study+of+4G+in+Wireless+Communication:+Looking+Insight+in+Issues&source=bl&ots=SUWD1Jv7Nu&sig=FBgSi0eq2XHkde4mhVc-YQ6hIKY>

"Lenguaje de Programación Ruby." 2013. 24 Dic. 2015 <https://www.ruby-lang.org/es/>

"The Pragmatic Bookshelf | The Cucumber Book." 2014. 7 Jan. 2016 <https://pragprog.com/book/hwcuc/the-cucumber-book>

Surjanovic, S., & Bingham, D. (2015, January). Virtual Library of Simulation Experiments:. Retrieved March 01, 2016, <http://www.sfu.ca/~ssurjano/optimization.html>

Global Optimization Benchmarks and AMPGO. (2013, February 6). Retrieved March 01, 2016, from <http://infinity77.net/global_optimization/index.html>

Benchmarks. (2013, January 17). Retrieved March 01, 2016, from <http://www.cs.unm.edu/~neal.holts/dga/benchmarkFunction/index.html>

Benchmark Functions for Large Scale Optimization. (2009, February 3). Retrieved November 20, 2015, from <http://www.al-roomi.org/multimedia/CEC_Database/CEC2008/CEC2008_TechnicalReport.pdf>

Momin Jamil and Xin-She Yang, A literature survey of benchmark functions for global optimization problems, Int. Journal of Mathematical Modelling and Numerical Optimisation, Vol. 4, No. 2, pp. 150–194 (2013).

Su, C. (2013). Quality engineering: Off-line methods and applications. London, New York: CRC Press.

Yogananda Jeppu Yogananda Jeppu (view profile) 12 files 51 downloads 4.75, J. (2014, March/April). Orthogonal Array - File Exchange - MATLAB Central. Retrieved October 16, 2015, from http://www.mathworks.com/matlabcentral/fileexchange/47218-orthogonal-array?requestedDomain=www.mathworks.com

Garcia, A. (2008, February 4). Computación Evolutiva. Retrieved May 3, 2015, from http://eisc.univalle.edu.co/~angarcia/ce/ce_clases/

Goodard, S. (2004, October 2). Dynamic Programming. Retrieved October 17, 2016, from http://cse.unl.edu/~goddard/Courses/CSCE310J/Lectures/Lecture8-DynamicProgramming.pdf

Haldenwang, N. (2011, May 15). Genetic Algorithm vs 0-1 Knapsack. Retrieved June 2, 2016, from http://www.nils-haldenwang.de/

Cole, E. A. (2009). The mathematical and numerical modelling of heterostructure semiconductor devices: From theory to programming. London: Springer.

Raedt, L. D. (2012). ECAI 2012: 20th European conference on artificial intelligence, 27-31 August 2012, Montpellier, France including Prestigious applications of artificial intelligence (PAIS-2012) systems demonstrations track. Amsterdam: IOS Press.

Dyer, D. (2010). Fitness Proportional Selection. Retrieved November 3, 2016, from http://watchmaker.uncommons.org/manual/ch03s02.html

No Free Lunch Theorems For Optimization - Evolutionary ... (1997, April 1). Retrieved November 3, 2016, from https://ti.arc.nasa.gov/m/profile/dhw/papers/78.pdf