Algoritmos Genéticos Celulares con operadores de recombinación aplicados a problemas de optimización discretos

Soria, C^1. Pandolfi, D^2. Villagra, S^2
[1] $\{BecariodeInvestigaciondeGrado\}$
[2] $\{DocenteInvestigadorUNPA\}$
$christianlsoria@hotmail.com, \{dpandolfi, svillagra\}$ $@uaco.unpa.edu.ar$
UNPA UACO
Universidad Nacional de la Patagonia Austral - Unidad Académica Caleta Olivia
Departamento de Ciencias Exactas y Naturales
LabTEm- Laboratorio de Tecnologías Emergentes
Caleta Olivia, 2013

RESUMEN

Cuando hablamos de metaheurísiticas hacemos referencia a procedimientos que tratan de aportar soluciones a un problema complejo determinado mediante la creación de algoritmos. El objetivo de estudio es básicamente el desarrollo de nuevos métodos capaces de resolver problemas complejos, con un mayor rendimiento, es decir con un menor esfuerzo computacional. Una de las herramientas más populares de optimización son los Algoritmos Genéticos Celulares (cGAs), estos se enfocan en encontrar soluciones óptimas en un tiempo reducido. En este trabajo proponemos un estudio comparativo de diferentes operadores de recombinación en un cGA aplicado a una serie de problemas académicos de optimización discretos. Realizando un profundo análisis estadístico de los resultados.

Palabras clave: metaheurística, algoritmo genético celular,operadores de recombinación, problemas de optimización discretos.

1. Introducción

La mayoría de los problemas de optimización son, en general, difíciles de resolver en la práctica. Estos problemas están incluidos en la clase de problemas NP-duros [10], ya que no se conocen algoritmos exactos con complejidad polinómica que permitan resolverlos. Debido a su intratabilidad, se han diseñado una gran cantidad de métodos aproximados, los cuales encuentran buenas soluciones en tiempos computacionales razonables. Durante la década de los sesenta se han diseñado diversos métodos aproximados, conocidos como heurísticos, capaces de encontrar soluciones de buena calidad, y que en muchos casos son muy cercanas a la solución óptima o mejor conocida. Gran parte de estos métodos fueron concebidos inspirándose en la resolución de problemas de fácil representación, pero de muy difícil solución.

En los últimos 20 años han aparecido una nueva clase de algoritmos, basados en la combinación de métodos heurísticos básicos en un marco de alto nivel para explorar en forma eficiente y eficaz el espacio de búsqueda. Esos métodos son comúnmente denominados metaheurísticas [12]. El surgimiento de las técnicas metaheurísticas plantea un cambio importante en el desarrollo de técnicas alternativas frente a heurísticos ad hoc, ya que permiten extender de manera estructurada su aplicación a una amplia gama de problemas de optimización representativos del mundo real. Alrededor de los años 60, varias corrientes de investigación comenzaron, en forma independiente, formando lo que ahora se conoce como computación evolutiva, tomando su inspiración en la evolución biológica.

Los algoritmos evolutivos constituyen una técnica general de resolución de problemas de búsqueda y optimización, inspirada en los procesos biológicos que se pueden apreciar en la naturaleza, como la selección natural y la herencia genética. Estos algoritmos permiten abordar problemas complejos que surgen en las ingenierías y los campos científicos: problemas de planificación de tareas, horarios, tráfico aéreo y ferrovario, búsqueda de caminos óptimos, optimización de funciones, etc. Los algoritmos evolutivos se caracterizan como modelos computacionales del proceso evolutivo. En cada iteración, se aplica un cierto conjunto de operadores a los individuos de la población actual para generar los individuos de la población de la próxima generación (iteración). Generalmente, los EAs usan operadores denominados recombinación o cruce para recombinar dos o más individuos para producir nuevos individuos. Los algoritmos genéticos son una variante de algoritmos evolutivos. Consiste en hacer evolucionar una población de enteros binarios sometiéndolos a transformaciones genéticas unitarias y binarias, y a un proceso de selección. Dentro de los EAs estructurados (en los que la población se descentraliza de alguna forma), los algoritmos celulares son una de las herramientas de optimización más populares. En el marco de los celulares encontramos los Algoritmos Genéticos Celulares (cGAs). En este caso la población está estructurada utilizando el concepto de vecindario, de forma que los individuos sólo pueden interactuar con sus vecinos más próximos en la población. Los CAs son sistemas dinámicos formados por un conjunto de celdas que pueden estar en varios estados predefinidos. Estas celdas evolucionan en función de los estados de las celdas vecinas según un conjunto de reglas establecidas. Los CAs han sido ampliamente estudiados en la literatura, por lo que la posibilidad de importar las ideas existentes en el campo de los CAs a los cEAs nos motiva especialmente. Además, otro aspecto importante es que los cEAs están siendo aplicados con mucho éxito en los últimos años en la resolución de problemas complejos académicos [1]

e industriales, como problemas de logística o de ingeniería.

Lo que se propone en este trabajo, es realizar el estudio comparativo de diferentes operadores de recombinación en un cGA aplicado a problemas de optimización discretos, para determinar cuál es el operador adecuado para cada uno de los problemas utilizados.

2. Algoritmos Genéticos Celulares

Las metaheurísticas son estrategias inteligentes para diseñar o mejorar procedimientos heurísticos muy generales con un alto rendimiento. La idea básica, es combinar diferentes métodos heurísticos a un nivel más alto, para conseguir una exploración del espacio de búsqueda de forma eficiente y efectiva. Dentro de clasificación de las metaheurísticas, basadas en la población, se encuentran los Algoritmos Evolutivos (EA).

Una de las ramas más desarrolladas dentro de algoritmos evolutivos son los algoritmos genéticos, denominados originalmente "planes reproductivos genéticos" fueron desarrollados por John H. Holland [13], motivado por resolver problemas de aprendizaje de máquina. Son un tipo de algoritmo evolutivo pensado inicialmente para trabajar con soluciones representadas mediante cadenas binarias, denominadas cromosomas.

La familia de EAs que consideramos como caso de estudio en este trabajo es la de los algoritmos genéticos celulares. Un Algoritmo Genético Celular es un tipo de Algoritmo Genético, basado en una clase de población descentralizada en el que las soluciones tentativas evolucionan en barrios superpuestos. En un cGA, conceptualmente los individuos son situados en una malla toroidal bidimensional (normalmente en bidimensional, aunque el número de dimensiones puede ser extendido fácilmente a tres o más), y se les permite recombinarse con individuos cercanos. Los GAs codifican las variables de decisión de un problema de búsqueda en cadenas de variables de longitud finita de algún alfabeto de cierta cardinalidad. Las cadenas, que son soluciones candidatas al problema de búsqueda, son llamadas $cromosomas$, a cada una de las variables que forman el cromosoma se les llama gen, mientras que los distintos valores que pueden tomar estos genes se llaman $alelos$. En la Figura 1 puede verse un ejemplo de la estructura típica de un individuo de un GA.

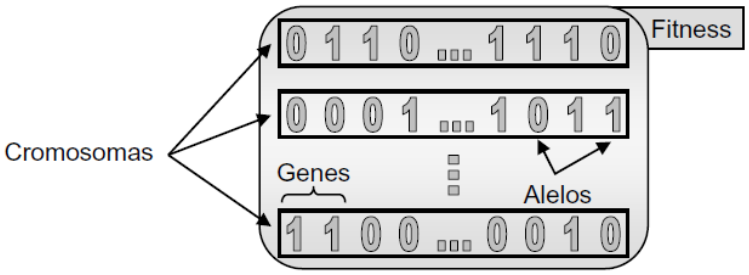

Figura 1: Estructura de un individuo en un GA.

Una vez que tenemos el problema codificado en uno o varios cromosomas y hemos definido

una función de fitness para distinguir las soluciones mejores, de las peores, podemos comenzar a evolucionar la población de soluciones del problema dado, siguiendo los pasos que se dan a continuación:

1. Inicialización. La población inicial de soluciones candidatas se genera usualmente de forma uniformemente aleatoria, aunque se puede incorporar fácilmente conocimiento del problema u otro tipo de información en esta fase.

2. Evaluación. Una vez que hemos inicializado la población, o cuando creamos una nueva solución descendiente, es necesario calcular el valor de adecuación de las soluciones candidatas.

3. Selección. Mediante la selección se trata de dar preferencia en la evolución a aquellas soluciones con un mayor valor de fitness, imponiéndose así un mecanismo que permite la supervivencia de los mejores individuos. La principal idea de la selección consiste en preferir las mejores soluciones frente a las peores, y existen muchos procedimientos de selección propuestos para cumplir con esta idea.

4. Recombinación. La recombinación combina partes de dos o más soluciones padres para crear nuevas soluciones (descendientes), que son posiblemente mejores. Existen muchas formas de conseguir esto, y un rendimiento competente del algoritmo depende de un mecanismo de recombinación bien diseñado. Idealmente, la solución descendiente que se obtiene de la recombinación no será idéntica a ninguno de los padres, sino que contendrá información combinada de los dos.

5. Mutación. Mientras la recombinación opera sobre dos o más cromosomas padre, la mutación modifica una única solución de forma aleatoria. De nuevo, existen multitud de variantes de mutaciones, que usualmente afectan a uno o más rasgos del individuo. En otras palabras, la mutación realiza un camino aleatorio en la vecindad de una solución candidata.

6. Reemplazo. La población descendiente que creamos mediante la selección, recombinación y mutación reemplaza a la población de padres siguiendo algún criterio determinado, como por ejemplo el reemplazo elitista, que preserva la mejor solución a lo largo de la evolución.

En la Figura 2 presentamos el pseudocódigo de un cGA canónico. En él se puede ver cómo, tras la generación y evaluación de la población inicial, los operadores genéticos arriba mencionados (selección de los padres, su recombinación, la mutación del (los) descendiente(s) y el reemplazo del individuo actual por un descendiente) son aplicados a cada uno de los individuos dentro del entorno de sus vecindarios iterativamente hasta que se alcanza la condición de terminación. Los individuos que formarán parte de la población de la siguiente generación (bien los nuevos descendientes generados o bien los individuos de la población actual dependiendo del criterio de reemplazo) van almacenándose en una población auxiliar que tras cada generación reemplaza a la población actual. Por tanto, en este modelo todos los individuos son actualizados simultáneamente en la población.

```
1.  proc Evoluciona(cga)        // Parámetros del algoritmo en 'cga'
2.  GeneraPoblaciónInicial(cga.pobl);
3.  Evaluación(cga.pobl);
4.  para s ← 1 hasta MAX_PASOS hacer
5.     para x ← 1 hasta cga.ANCHO hacer
6.        para y ← 1 hasta cga.ALTO hacer
7.           vecinos ← CalculaVecindario(cga,posición(x,y));
8.           padres ← Selección(vecinos);
9.           descendiente ← Recombinación(cga.Pc,padres);
10.          descendiente ← Mutación(cga.Pm,descendiente);
11.          Evaluación(descendiente);
12.          Remplazo(posición(x,y),pobl_auxiliar,descendiente);
13.       fin para
14.    fin para
15.    cga.pobl ← pobl_auxiliar;
16. fin para
17. fin proc Evoluciona
```

Figura 2: Pseudocódigo de un cGA Canónico

Para la realización de este trabajo utilizamos Algoritmos Genéticos Celulares, aplicando tres tipos de operadores de recombinación de los que hablaremos en la siguiente sección.

2.1. Operadores de Recombinación

El operador de recombinación (*crossover*) es el operador de búsqueda más importante en los algoritmos genéticos. Este es un operador sexuado que intercambia el material genético de un par de padres produciendo descendientes que normalmente difieren de sus padres. La idea central es que segmentos distintos de padres diferentes con alta adaptación deberían combinarse en nuevos individuos que tomen ventaja de esta combinación.

El operador de recombinación opera con probabilidad P_c (esto permite que en algunos casos no haya recombinación y se mantengan los padres). Dados p y q un par de padres, de largo l bits, se escoge aleatoriamente un punto $k \in \{1, \cdots, l-1\}$ y se intercambian los bits a la derecha de esa posición entre ambos individuos, obteniéndose los descendientes s y v, como se indica a continuación:

$$\overrightarrow{p} = (p_1, \ldots, p_{k-1}, p_k, \ldots, p_l) \qquad \overrightarrow{s} = (p_1, \ldots, p_{k-1}, q_k, \ldots, q_l)$$

$$\overrightarrow{q} = (q_1, \ldots, q_{k-1}, q_k, \ldots, q_l) \qquad \overrightarrow{v} = (q_1, \ldots, q_{k-1}, p_k, \ldots, p_l)$$

El operador *crossover* de un punto, sufre de un sesgo posicional ya que un bit cercano al extremo derecho de la tira tiene una alta probabilidad de intercambio, mientras que un bit en el extremo izquierdo tiene una baja probabilidad de intercambio. El operador *crossover* binomial corrige este sesgo, intercambiando bits entre padres sobre una base bit a bit, con probabilidad $p \in [0, 1]$ aleatoria, distinta para cada posición [9].

Los operadores de recombinación son los encargados de mezclar y combinar partes de dos padres para la generación de sus hijos. Hay tres formas clásicas de hacer cruce de vectores: un punto, dos puntos, y *crossover* uniforme.

Estos operadores se centran en combinar información de orden o adyacencia presente en los padres. El principal problema de estos operadores es que no utilizan información sobre el problema que se está intentando resolver. Su importancia para la transición entre generaciones es elevada puesto que las tasa de cruce con las que se suele trabajar rondan el $90\,\%$.

La idea principal del cruce se basa en que, si se toman dos individuos correctamente adaptados al medio y se obtiene una descendencia que comparta genes de ambos, existe la posibilidad de que los genes heredados sean precisamente los causantes de la bondad de los padres. Al compartir las características buenas de dos individuos, la descendencia, o al menos parte de ella, tiene una bondad mayor que cada uno de los padres por separado. Si el cruce no agrupa las mejores características en uno de los hijos es posible que la descendencia tenga peor ajuste que los padres.

2.1.1. Cruce de un Punto

Esta técnica fue propuesta por Holland [13], y fue muy popular durante muchos años. Hoy en día, sin embargo, no suele usarse mucho en la práctica debido a sus inconvenientes. Puede demostrarse, por ejemplo, que hay varios esquemas que no pueden formarse bajo esta técnica de cruza.Formalmente el operador de cruce por un punto puede definirse como:

1. Sea P_1, P_2 dos soluciones padres formadas $P_1[1], \ldots, P_1[n]$ y $P_2[1], \ldots, P_2[n]$ respectivamente
2. Generar un punto de cruce $k, 1 \leq k \leq n$
3. Las soluciones hijo C_1 y C_2 serán:
   a) $C_1 := P_1[1], \ldots, P_1[k], P_2[k+1], \ldots, P_2[n]$
   b) $C_2 := P_2[1], \ldots, P_2[k], P_1[k+1], \ldots, P_1[n]$

El cruce de un punto es la técnica más sencilla de cruce. Una vez seleccionados dos individuos se cortan sus cromosomas por un punto seleccionado aleatoriamente para generar dos segmentos diferenciados encada uno de ellos: la cabeza y la cola. Se intercambian las colas entre los dos individuos para generar los nuevos descendientes. De esta manera ambos descendientes heredan información genética de los padres, tal como puede verse en la siguiente Figura 3.

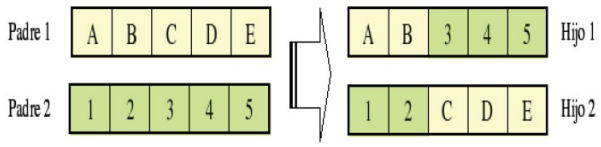

Figura 3: Cruce un punto

La cruza de un punto destruye esquemas en los que la longitud de definición es alta. Esto produce el denominado sesgo posicional: los esquemas que pueden crearse o destruirse por la cruza dependen fuertemente de la localización de los bits en el cromosoma.

El problema fundamental de la cruza de un punto es que presupone que los bloques constructores son esquemas cortos y de bajo orden, y cuando esto no sucede (p.ej., con cadenas largas), suele no proporcionar resultados apropiados. Obviamente, las aplicaciones del mundo real suelen requerir cadenas largas y de ahí que esta cruza no se use comúnmente en dichos problemas. La cruza de un punto trata también preferencialmente algunas posiciones del cromosoma, como por ejemplo los extremos de una cadena.

2.1.2. Cruce de dos Puntos

El operador de cruce de dos puntos es similar al de un punto excepto que genera dos puntos de corte en vez de uno. Formalmente el operador de cruce por dos puntos puede describirse mediante el siguiente algoritmo:

1. Sea P_1, P_2 dos soluciones padres formadas $P_1[1], \ldots, P_1[n]$ y $P_2[1], \ldots, P_2[n]$ respectivamente
2. Generar dos puntos de cruce $k1, k2, 1 \leq k1 \leq n, 1 \leq k2 \leq n$ y $k1 < k2$
3. Las soluciones hijo C_1 y C_2 serán:
   a) $C_1 := P_1[1], \ldots, P_1[k_1], P_2[k_1 + 1], \ldots, P_2[k_2], P_1[k_2 + 1], \ldots, P_1[n]$
   b) $C_2 := P_2[1], \ldots, P_2[k_1], P_1[k_1 + 1], \ldots, P_1[k_2], P_2[k_2 + 1], \ldots, P_2[n]$

Se trata de una generalización del cruce de un punto. En vez de cortar por un único punto los cromosomas de los padres como en el caso anterior se realizan dos cortes. Deberá tenerse en cuenta que ninguno de estos puntos de corte coincida con el extremo de los cromosomas para garantizar que se originen tres segmentos. Para generar la descendencia se escoge el segmento central de uno de los padres y los segmentos laterales de otro padre como lo muestra la siguente Figura 4.

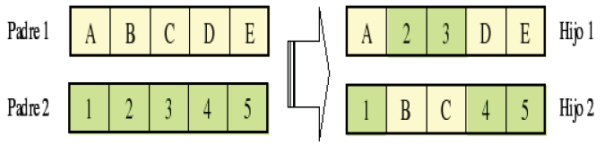

Figura 4: Cruce dos puntos

Generalizando se pueden añadir más puntos de cruce dando lugar a algoritmos de cruce multipunto. Sin embargo existen estudios que desaprueban esta técnica propuesta por De Jong [5]. Aunque se admite que el cruce de dos puntos aporta una sustancial mejora con respecto al cruce de un solo punto, el hecho de añadir un mayor número de puntos de cruce reduce el rendimiento del Algoritmo Genético. El problema principal de añadir nuevos puntos de cruce radica en que es más fácil que los segmentos originados sean corrompibles, es decir, que por separado quizás pierdan las características de bondad que poseían conjuntamente. Sin embargo no todo son desventajas y añadiendo más puntos de cruce se consigue que el espacio de búsqueda del problema sea explorado más a fondo.

El valor $n = 2$ es el que minimiza los efectos disruptivos (o destructivos) de la cruza y de ahí que sea usado con gran frecuencia. No existe consenso en torno al uso de valores para n que sean mayores o iguales a 3. Los estudios empíricos al respecto proporcionan resultados que no resultan concluyentes respecto a las ventajas o desventajas de usar dichos valores. En general, sin embargo, es aceptado que la cruza de dos puntos es mejor que la cruza de un punto.

2.1.3. Cruce Probabilístico o Uniforme

El cruce probabilístico o uniforme propone el intercambio aleatorio de bits entre los individuos padre dependiendo de una probabilidad fija p. Las variantes paramétricas del operador se obtienen considerando p entre 0 y $0{,}5$, de acuerdo a la simetría del problema. El operador tradicional corresponde a utilizar un valor de $p = 0{,}5$. Aunque se puede implementar de muy diversas formas, la técnica implica la generación de una máscara de cruce con valores binarios. Si en una de las posiciones de la máscara hay un 1, el gen situado en esa posicionen uno de los descendientes es copia del primer padre. Si por el contrario hay un 0 el gen se copia del segundo padre. Para producir el segundo descendiente se intercambian los papeles de los padres, o bien se intercambia la interpretación de los unos y los ceros de la máscara de cruce.

Tal como se puede apreciar en la Figura 5, la descendencia contiene una mezcla de genes de cada uno de los padres. El número efectivo de puntos de cruce es fijo pero será por término medio $L/2$, siendo L la longitud del cromosoma (número de los alelos en representaciones binarias o de genes en otro tipo de representaciones).

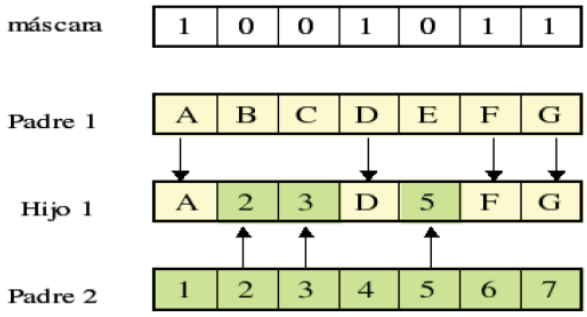

Figura 5: Cruce Uniforme o Probabilistico

Los operadores de recombinación descriptos anteriormente, se utilizan para la realización de experimentos aplicados a los siguientes problemas de Optimización Combinatoria.

3. Problemas de Optimización Combinatoria

En esta sección describimos los distintos problemas de optimización combinatoria que hemos utilizado para el análisis. En la definición de algunos de ellos se utiliza la distancia de Hamming. La distancia de Hamming entre una cadena de bits a y otra b se mide con la ecuación:

$$d_{ab} = \sum_{i=1}^{l} a_i \otimes b_i \tag{1}$$

Por tanto, cadenas de bits idénticas tienen una distancia de Hamming de $d_{ab} = 0{,}0$, mientras que cadenas de bits completamente diferentes tendrán una distancia de $d_{ab} = l$, siendo l la longitud de las dos cadenas.

3.1. Problema COUNTSAT

El problema COUNTSAT [8] es una instancia de MAXSAT. En COUNTSAT el valor de la solución se corresponde con el número de cláusulas (de entre todas las posibles cláusulas de Horn de tres variables) satisfechas por una entrada formada por n variables booleanas. Es fácil comprobar que el óptimo se consigue cuando todas las variables tienen el valor 1. En este estudio consideramos una instancia de $n = 20$ variables, y el valor de la solución óptima al problema es 6860

$$fCOUNTSAT(s) = s{+}n(n{-}1)(n{-}2){-}2(n{-}2)(\frac{s}{2}){+}6(\frac{s}{3}) = s{+}6840{-}18s(s{-}1){+}s(s{-}1)(s{-}2) \tag{2}$$

La función COUNTSAT esta extraída de MAXSAT con la intención de ser muy difícil de resolver con GAs [8]. Las entradas aleatorias generadas uniformemente tendrán $n = 2$ unos en término medio. Por tanto, los cambios locales decrementando el número de unos nos conducirán hacia entradas mejores, mientras que si se incrementa el número de unos se decrementará el valor de adecuación. En consecuencia, cabe esperar que un GA encuentre rápidamente la cadena formada por todos sus componentes a cero y tenga dificultades para encontrar la cadena compuesta por unos en todas sus posiciones.

3.2. Problema del Diseño de Códigos Correctores de Errores - ECC

El problema ECC se presentó en [15]. Consideremos una tupla (n, M, d) donde n es la longitud (número de bits) de cada palabra de código, M es el número de palabras de código, y d es la mínima distancia de Hamming entre dos palabras de código cualesquiera. El objetivo será obtener un código con el mayor valor posible de d (reflejando una mayor tolerancia al ruido y a los errores), y con valores de n y M fijados previamente. La función de adecuación que maximizaremos se corresponde con:

$$f_{ECC}(C) = \frac{1}{\sum_{i=1}^{M} \sum_{\substack{j=1 \\ i \neq j}}^{M} \frac{1}{d_{ij}^2}} \tag{3}$$

donde d_{ij} representa la distancia de Hamming entre las palabras i y j del código (compuesto de M palabras de longitud n). En este estudio consideramos una instancia donde C está formado por $M = 24$ palabras de longitud $n = 12$ bits. Lo que forma un espacio de búsqueda de tamaño

$\binom{4096}{24}$, que es 10^{87} aproximadamente. La solución óptima para la instancia con $M = 24$ y $n = 12$ tendrá el valor de adecuación $0{,}0674$ [4]o ha. En algunos de nuestros estudios, hemos implicado el espacio de búsqueda del problema reduciendo a la mitad $(M/2)$ el número de palabras que forman el código, y la otra mitad se compone del complemento de las palabras encontradas por el algoritmo.

3.3. Problema de la Modulación en Frecuencia de Sonidos - FMS

El problema FMS [17] consiste en determinar los 6 parámetros reales $\vec{x} = (a_1, w_1, a_2, w_2; a_3, w_3)$ del modelo de sonido en frecuencia, donde $= 2\pi/100$. Este problema puede ser definido como un problema de optimización discreta o continua. En el caso que discreto, los parámetros están codificados en el rango $[-6{,}4, +6{,}35]$ con cadenas de 32 bits

$$y(t) = a_1 sin(w_1 t\theta + a_2 sin(w_2 t\theta + a_3 sin(w_3 t\theta))) \tag{4}$$

$$y_0(t) = 1.0 sin(5.0 t\theta - 1.5 sin(4.8 t\theta + 2.0 sin(4.9 t\theta))) \tag{5}$$

El objetivo de este problema consiste en minimizar la suma del error cuadrático medio entre los datos de muestreo (Ecuación 4) y los reales (Ecuación 5), como se detalla en la Ecuación 6. Este problema es una función multimodal altamente compleja. Debido a la extrema dificultad para resolver este problema con alta precisión sin utilizar operadores específicos de optimización continua, el algoritmo se detiene cuando el error cae por debajo de 10^{-2}. La función de adecuación que tendremos que maximizar se corresponde con la inversa de la Ecuación 6 y obtenemos su máximo valor cuando $E_{FMS} = 0.0$

$$E_{FMS}(\vec{x}) = \sum_{t=0}^{100} (y(t) - y_0(t))^2 \tag{6}$$

3.4. Problema del Máximo Corte de un Grafo - MAXCUT

El problema MAXCUT consiste en dividir un grafo ponderado $G = (V, E)$ en dos subgrafos disjuntos $G_0 = (V_0, E_0) y G_1 = (V_1, E_1)$de manera que la suma de los pesos de los ejes que posean un extremo en V_0 y el otro en V_1 sea máxima. Para codificar una partición de los vértices utilizaremos una cadena binaria $(x_1, x_2, \ldots, x_n)$ donde cada dígito se corresponde con un vértice. De manera que si un dígito tiene valor 1, su vértice correspondiente estaría en el conjunto V_1, y si tuviera valor 0 el vértice respectivo pertenecería a V_0. La función que maximizaremos es [14]:

$$f_{MAXCUT}(\vec{x}) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} w_{ij} \cdot [x_i (1 - x_j) + x_j (1 - x_i)] \tag{7}$$

Téngase en cuenta que w_{ij} contribuye a la suma sólo en el caso de que los nodos i y j se encuentren en distintas particiones. Aunque es posible generar instancias de grafos diferentes de

forma aleatoria, hemos utilizado tres instancias distintas del problema. Dos de ellas se corresponden con grafos generados aleatoriamente: uno disperso "cut20. 01" y otro denso "cut20. 09" , ambos formados por 20 vértices. La otra instancia es un grafo escalable de 100 vértices. Las soluciones óptimas para estas instancias son 10,119812 para el caso de "cut20. 01", 56,740064 para "cut20. 09" y 1077 para "cut100"

3.5. Problema Masivamente Multimodal - MMDP

El problema MMDP ha sido específicamente diseñado para ser difícil de afrontar para los EAs. Está compuesto de k sub-problemas engañosos (s_i) de 6 bits cada uno. El valor de cada uno de estos sub-problemas $(fitness_{s_i})$ depende del número de unos que tiene la cadena binaria que lo compone (ver la Figura 6). Es fácil comprender por qué se considera a esta función engañosa, puesto que estas sub-funciones tienen dos máximos globales y un atractor engañoso en el punto medio (gráfica de la Figura 6).

Nº de Unos	Valor de la Subfunción
0	1.000000
1	0.000000
2	0.360384
3	0.640576
4	0.360384
5	0.000000
6	1.000000

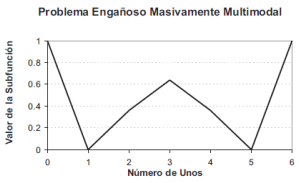

Figura 6: Función bipolar engañosa básica (s_i) para MMDP.

En MMDP, cada sub-problema si contribuye al valor de *fitness* total en función del número de unos que tiene (Figura 6). El óptimo global del problema tiene el valor k, y se alcanza cuando todos los sub-problemas están compuestos de cero o seis unos. El número de óptimos locales es muy grande (22^k), mientras que sólo hay 2^k soluciones globales. Por tanto, el grado de multimodalidad viene dado por el parámetro k. Nosotros utilizaremos una instancia considerablemente grande de $k = 40$ sub-problemas. La función que trataremos de maximizar para resolver el problema se muestra en la Ecuación 8, y su valor máximo es 40.

$$f_{MMDP}\left(\vec{s}\right) = \sum_{i=1}^{k} fitness_{s_i} \tag{8}$$

3.6. Problema de las Tareas de Mínima Espera - MTTP

El problema MTTP [3] es un problema de planificación de tareas en el que cada tarea i del conjunto de tareas $T = \{1, 2, \ldots, n\}$ tiene una longitud l_i -el tiempo que tarda en ejecutarse- un límite de tiempo (o deadline) d_i -antes del cual debe ser planificada- y un peso w_i. El peso es una penalización que debe ser añadida a la función objetivo en el caso de que la tarea permanezca sin estar planificada. Las longitudes, pesos, y límites de tiempo de las tareas son todos números enteros positivos. Planificar la tarea de un sub-conjunto S de T consiste en encontrar el tiempo

de comienzo de cada tarea en S, de forma que como mucho se realiza una tarea cada vez y que cada tarea termina de realizarse antes de su límite de tiempo.

Definimos una función de planificación g sobre $S \subseteq T : S \mapsto \mathbb{Z}^+ \bigcup \{0\} \mid \forall i.j \in S$ con estas dos propiedades:

1. Una tarea no puede ser planificada antes de que haya terminado alguna otra ejecutada previamente: $g(i) < g(j) \Rightarrow g(i) + l_i \leq g(j)$

2. Toda tarea tiene que ser analizada antes de su límite de tiempo: $g(i) + l_i \leq di$.

El objetivo del problema es minimizar la suma de los pesos de las tareas no planificadas (Ecuación 9). Por tanto, la planificación óptima minimiza la función:

$$W = \sum_{i \in T-S} w_i \tag{9}$$

La planificación S está representada mediante un vector $\vec{x} = (x_1, x_2, \ldots, x_n)$ que contiene todas las tareas ordenadas según su límite de tiempo para ser realizada. Cada $x_i \in \{0, 1\}$, donde si $x_i = 1$ entonces la tarea i está contenida en la planificación S, mientras que si $x_i = 0$ la tarea i no está incluida en S. La función que optimizaremos, descrita en [14], es la inversa de la Ecuación 9: $f_{MTTP} = 1 = W$. Hemos utilizado para nuestros estudios tres diferentes instancias [14]: “mttp20”, “mttp100” y “mttp200”, de tamaños 20, 100 y 200, y valores óptimos de $0{,}02439$, $0{,}005$ y $0{,}0025$, respectivamente.

3.7. Problema OneMax

El problema OneMax [16] (o recuento de bits) es un problema muy simple que consiste en maximizar el número de unos que contiene una cadena de bits. Formalmente, este problema se puede describir como encontrar una cadena $\vec{x} = \{x_1, x_2, \ldots, x_n\}$ con $x_i \in \{0, 1\}$, que maximice la siguiente ecuación:

$$f_{OneMax}(\vec{x}) = \sum_{i=1}^{n} x_i \tag{10}$$

La función f_{OneMax} tiene $(n+1)$ posibles valores diferentes, que están distribuidos multimodalmente. Para las funciones aditivamente descomponibles (ADF_S) la distribución multimodal ocurre con bastante frecuencia [11]. Para definir una instancia de este problema, sólo necesitamos definir la longitud de la cadena de bits (n). Para un n dado, la solución óptima al problema es una cadena de n unos, es decir, se les fija el valor uno a todos los bits de la cadena.

3.8. Problema P-PEAK

El problema P-PEAKS [6] es un generador de problemas multimodales. Un generador de problemas es una tarea fácilmente parametrizable, con un grado de dificultad que se puede afinar, de manera que nos permita generar instancias con una complejidad tan grande como

queramos. Adicionalmente, el uso de un generador de problemas elimina la posibilidad de afinar a mano los algoritmos para un problema particular, permitiendo de esta forma una mayor justicia al comparar los algoritmos. Utilizando un generador de problemas, evaluamos nuestros algoritmos sobre un elevado número de instancias de problema aleatorias, ya que cada vez que se ejecuta el algoritmo resolvemos una instancia distinta. Por tanto, se incrementa el poder de predicción de los resultados para la clase de problemas como tal, no para instancias concretas.

La idea de P-PEAKS consiste en generar P cadenas aleatorias de N bits que representan la localización de P cimas en el espacio de búsqueda. El valor de adecuación de una cadena se corresponde con la distancia de Hamming de esa cadena con respecto a la cima más próxima, dividida por N (como se muestra en la Ecuación 11). Usando un mayor (o menor) número de cimas obtenemos problemas más (o menos) epistáticos. En nuestro caso hemos utilizado una instancia de $P = 100$ cimas de longitud $N = 100$ bits cada una, representando un nivel de epistasis medio/alto [2]. El máximo valor de fitness que podremos conseguir es 1.0

$$f_{P-PEAK}(\vec{x}) = \frac{1}{N} \max_{i=1}^{p} \{N - Hamming(\vec{x}, Peak_i)\} \tag{11}$$

4. Diseño de experimentos y resultados

Para la realización de los experimentos propuestos en este trabajo, utilizamos una parametrización para problemas discretos propuesta en [7]. El tamaño de la población se fija en 400 individuos (20x20). El mecanismo de selección de padres es la selección por torneo. El operador de mutación utilizado es el binario con una probabilidad de mutación establecida en 1. Para los operadores de recombinación que utilizamos (Spx, Dpx y Px) la probalibilidad utilizada es 1 y el número máximo de evaluaciones es de 1000000. A continuación en la Tabla 1 se describen los valores utilizados.

Tamaño de la Población	400 individuos (20,20)
Vecindario	L5
Selección de los Padres	torneo binario + torneo binario
Bit Mutuación	$Bit - Flip, pm = 1/L$ ($10/L$ para FMS)
Longitud de Individuos	L
Reemplazo	Reemp si Mejor

Tabla 1: Parametrización utilizada en los problemas propuestos.

Utilizamos los siguiente operadores de cruce: cruce de un punto (Spx), de dos puntos (Dpx) y probabilístico o uniforme (Px).

A continución analizamos los resultados obtenidos por cGA con distintos opradores de cruce, con el fin de determinar con que operador de cruce se obtienen mejores resultados según

cada caso y si existen diferencias significativas entre los resultados obtenidos. En la Tabla 2 se muestra el porcentaje de éxito obtenido de 150 corridas independientes para los doce problemas y los 3 operadores de recombinación aplicados en cGA utilizando la parametrización descrita anteriormente, los mejores resultados se muestran en **negrita**. La primer columna corresponde al problema.

Problema	Spx	Dpx	Px
COUNTSAT	0 %	0 %	0 %
ECC	**100 %**	**100 %**	**100 %**
FMS	0 %	0 %	0 %
MMDP	0 %	0 %	0 %
P-PEAK	**100 %**	**100 %**	**100 %**
OneMax	**100 %**	**100 %**	**100 %**
“cut20.01”	**100 %**	**100 %**	**100 %**
“cut20.09”	**100 %**	**100 %**	**100 %**
“cut100”	0 %	3 %	6 %
“mttp20”	**100 %**	**100 %**	**100 %**
“mttp100”	**38 %**	**100 %**	**100 %**
“mttp200”	0 %	0 %	15 %

Tabla 2: Porcentaje de éxito de cGA con diferentes operadores de recombinación (Spx, Dpx y Px)

A continuación analizamos solamente los resultados de 30 corridas exitosas para cada una de los algoritmos utilizados.

En la Tabla 3 se muestra para cada uno de los operadores de recombinación, el número de evaluaciones y el tiempo (en milisegundos). Podemos observar que para cuatro de los siete problemas (OneMax, “cut20.09 ”, “mttp20”, “mttp100”) el menor número de evaluaciones es obtenido por cGA con recombinación probabilística o uniforme (Px). En cuanto al tiempo, los resultados que obtuvimos son más variados. En tres de los siete problemas (OneMax, “cut20.09 ”, “mttp100”) los menores tiempos de ejecución se obtuvieron con recombinación probabilística o uniforme (Px), en otros tres (P-PEAK, “cut20.01 ”, “mttp20”) el menor tiempo de ejecución se obtuvo con cruce de un punto (Spx).

Para realizar un análisis más profundo de los resultados obtenidos aplicaremos a continuación diferentes test estadísticos. Primero necesitamos determinar qué tipo de test aplicar, estos pueden ser paramétricos y no paramétricos, para determinar esto debemos analizar la independencia de los datos, la normalidad y homocedasticidad. Para realizar estos test usaremos la herramienta SPSS. El test de Kolmogorov-Smirnov nos permite conocer la normalidad y el test de Levene la homocedasticidad.

De los test obtenemos el respectivo p-valor asociado, que representa la disimilitud de los resultados con respecto a la forma normal. Por lo tanto, un p-valor bajo señala un distribución no normal. En este estudio, vamos a considerar un nivel de significancia $\alpha = 0,005$, por lo que un p-valor mayor que α indica que la condición de normalidad se cumple.

Problema	Spx		Dpx		Px	
	NºEv.	Tiempo	NºEv.	Tiempo	NºEv.	Tiempo
ECC	**38200**	414	40800	**408**	130600	2050
P-PEAK	**39600**	**1991**	194200	10619	177200	10381
OneMax	694600	2274	128600	2282	**22000**	**452**
“cut20.01”	**2400**	**10**	3200	20	2800	15
“cut20.09”	4000	20	4800	20	**3800**	**16**
“mttp20”	2800	**10**	3000	**10**	**2600**	15
“mttp100”	269600	1131	104000	382	**25400**	**210**

Tabla 3: Resultados obtenidos por cGA con diferentes operadores de recombinación

En la Tabla 4 se muestra la prueba de Kolmogorov-Smirnov aplicado a cGA con diferentes operadores de recombinación, donde el símbolo “*” indica que los resultados obtenidos por el algoritmo no cumple con la condición de normalidad. Cuando esto sucede, debemos aplicar test no paramétricos.

Problema	Spx		Dpx		Px	
	NºEv.	Tiempo	NºEv.	Tiempo	NºEv.	Tiempo
ECC	$(*)0{,}000$	$(*)0{,}000$	$(*)0{,}000$	$(*)0{,}000$	$(*)0{,}000$	$0{,}200$
P-PEAK	$(*)0{,}071$	$0{,}200$	$(*)0{,}004$	$(*)0{,}002$	$(*)0{,}001$	$(*)0{,}001$
OneMax	$0{,}156$	$0{,}185$	$0{,}197$	$0{,}061$	$(*)0{,}001$	$(*)0{,}000$
“cut20.01”	$0{,}182$	$(*)0{,}000$	$(*)0{,}016$	$(*)0{,}000$	$0{,}094$	$(*)0{,}000$
“cut20.09”	$(*)0{,}003$	$(*)0{,}002$	$(*)0{,}007$	$(*)0{,}002$	$(*)0{,}022$	$(*)0{,}000$
“mttp20”	$(*)0{,}018$	$(*)0{,}000$	$(*)0{,}002$	$(*)0{,}000$	$(*)0{,}009$	$(*)0{,}000$
“mttp100”	$(*)0{,}001$	$(*)0{,}001$	$(*)0{,}000$	$(*)0{,}002$	$0{,}116$	$(*)0{,}004$

Tabla 4: Prueba de Kolmogorov-Smirnov aplicado a variables de “Nºde Evaluaciones” y “Tiempo” de un cGA con diferentes operadores de recombinación

Para determinar si el conjunto de datos a analizar posee homocedasticidad, se realiza el test de Levene. A continuación mostramos los resultados (Tabla 5), donde el símbolo “*” indica que el resultado obtenido para el conjunto de datos no cumple con la condición de homocedasticidad.

Problema	Estadistico de Levene	
	NºEv.	Tiempo
ECC	0,168	(∗)0,000
P-PEAK	(∗)0,000	(∗)0,000
OneMax	(∗)0,000	(∗)0,000
“cut20.01”	(∗)0,013	0,903
“cut20.09”	(∗)0,019	0,825
“mttp20”	0,444	0,516
“mttp100”	(∗)0,000	(∗)0,000

Tabla 5: Test de Levene aplicado a variables de “Nºde Evaluaciones” y “Tiempo” de un cGA con diferentes operadores de recombinación

Con los resultados obtenidos de los estadísticos de Levene y Kolmogorov-Smirnov, determinamos que para todos los casos analizados, se deben realizar test no paramétricos (Kruskal-Wallis) a las variables de número de evaluaciones y tiempo, para determinar la existencia de diferencias estadísticamente significativas (Tabla 6). Se utiliza el signo (+) para especificar que existen diferencias estadísticamente significativas entre los resultados obtenidos en los problemas y (-) en el caso contrario.

Problema	Spx		Dpx		Px		Kruskall-Wallis	
	NºEv.	Tiempo	NºEv.	Tiempo	NºEv.	Tiempo	NºEv.	Tiempo
ECC	**38200**	414	40800	**408**	130600	2050	(+)	(+)
P-PEAK	**39600**	**1991**	194200	10619	177200	10381	(+)	(+)
OneMax	694600	2274	128600	2282	**22000**	**452**	(+)	(+)
“cut20.01”	**2400**	**10**	3200	20	2800	15	(+)	(-)
“cut20.09”	4000	20	4800	20	**3800**	**16**	(-)	(-)
“mttp20”	2800	**10**	3000	**10**	**2600**	15	(-)	(-)
“mttp100”	269600	1131	104000	382	**25400**	**210**	(+)	(+)

Tabla 6: Test de Kruskal-Wallis aplicado a variables de “Nºde Evaluaciones” y “Tiempo” de un cGA con diferentes operadores de recombinación

Se puede observar en cuanto al número de evaluaciones que en 5 de los 7 problemas analizados existen diferencias estadísticamente significativas entre los resultados obtenidos por los algoritmos. Teniendo en cuenta el tiempo en 4 de todos los otros casos evaluados existe diferencia estadística significativa. Se aplica entonces el test de Tukey, que identifica el operador de recombinación con el cual se encontraron los mejores resultados y nos permite determinar si existen diferencias significativas entre este y el resto.

A continuación aplicamos el test de Tukey en los resultados de los algoritmos que tienen diferencias estadísticamente significativas. Por medio de este test podemos determinar entre que algoritmos existen las diferencias. Se seleccionaron algunos casos significativos para mostrar gráficamente.

Las Figuras 7 y 8 muestran el test de Tukey aplicado a las variables “Nºde Evaluaciones” y “Tiempo” respectivamente de un cGA con los tres operadores de cruce aplicado al problema de P-PEAK. Observando el resultado para la variable número de evaluaciones vemos que existen diferencias estadísticamente significativas con respecto a Spx y Px, pero no existen diferencias significativas entre Spx y Dpx. Para la variable "Tiempo.ª diferencia de lo anterior, Spx tiene diferencias estadísticamente significativas con respecto a los otros dos operadores. Esto significa que para el problema P-PEAK el mejor rendimiento se obtiene utilizando un operador de cruce un punto (Spx) respecto a cGA con los otros dos operadores de recombinación (Dpx y Px).

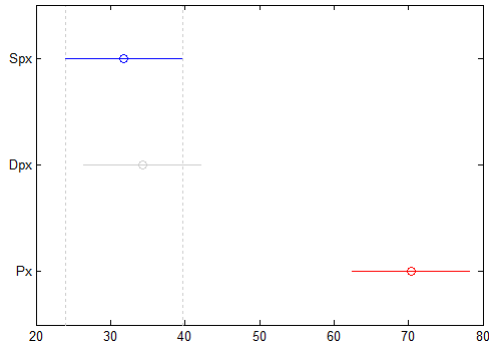

Figura 7: Test de Tukey aplicado a la variable “Nºde Evaluaciones” de un cGA con diferentes operadores de recombinación para resolver el problema P-PEAK

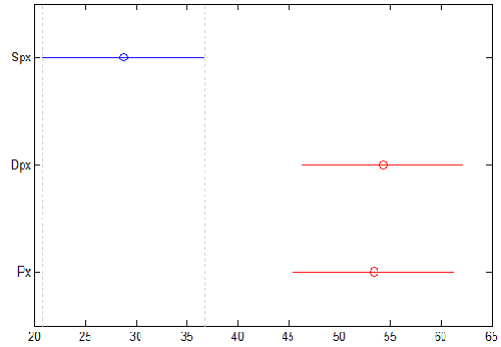

Figura 8: Test de Tukey aplicado a la variable de “Tiempo” de un cGA con diferentes operadores de recombinación para resolver el problema P-PEAK

En la Figura 9 tanto como en la Figura 10 podemos observar que para la instancia “mttp100” con cGA el operador probabilístico (Px) es el que obtiene los mejores resultados, reflejando en ambos casos diferencia significativa con respecto a operadores de recombinación Dpx y Spx siendo este último el de peor rendimiento para esta instancia en las dos variables.

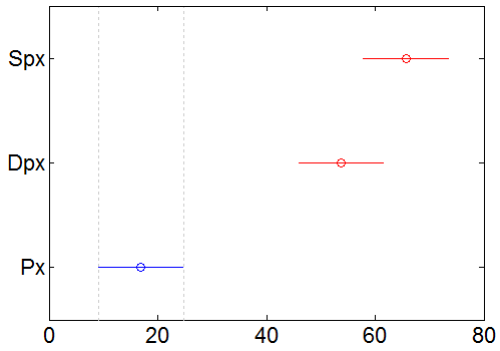

Figura 9: Test de Tukey aplicado a la variable “Nºde Evaluaciones” de un cGA con diferentes operadores de recombinación para resolver el problema “mttp100”

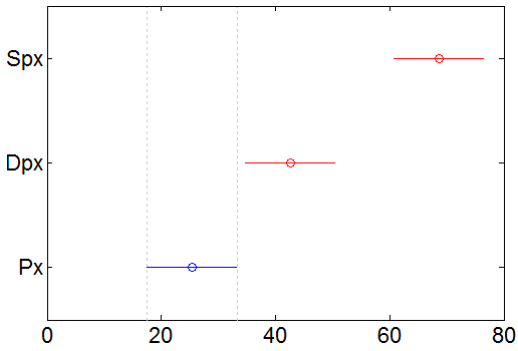

Figura 10: Test de Tukey aplicado a la variable de “Tiempo” de un cGA con diferentes operadores de recombinación para resolver el problema “mttp100”

En la Figura 11 se muestra el resultado del test de Tukey para el Nºde evaluaciones, donde se

puede observar que la mejor performance con cGA se obtiene utilizando operador de recombinación Spx, marcando una diferencia significativa con el operador Dpx. Como para la variable "Tiempo"para este problema las diferencias no fueron estadísticamente significativas mostramos en la Figura 12 el box-plot donde la distribución de los datos refleja como se distribuyen los resultados de cada algoritmo alrededor de la mediana. Vemos que cGA con Spx obtiene los menores valores en mediana y en todos los casos menores a los resultados obtenidos con cGA con los dos otros operadores.

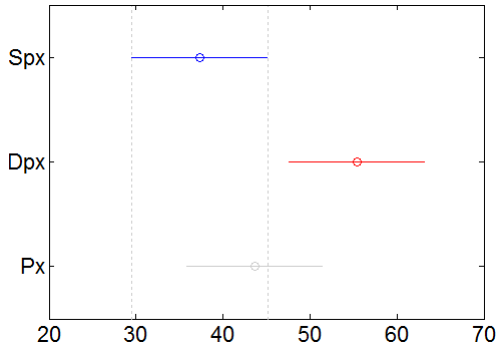

Figura 11: Test de Tukey aplicado a la variable de "Nºde Evaluaciones" de un cGA con diferentes operadores de recombinación para resolver el problema "cut20.01".

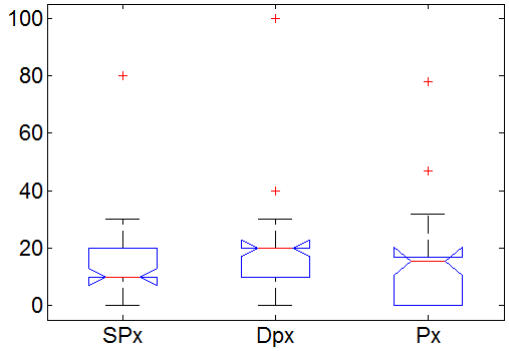

Figura 12: Boxplot de los resultados de la variable de "Tiempo" de un cGA con diferentes operadores de recombinación para resolver el problema "cut20.01".

El siguiente análisis corresponde al resultado con cGA para la instancia "cut20.09" de las variables número de evaluaciones y tiempo en ejecución de los tres tipos de operadores con los que venimos trabajando. Si bien se registró la mejor performance utilizando operador de recombinación Px con cGA , como dijimos anteriormente para esta instancia ("cut20.09") no existe diferencia estadísticamente significativa entre los tres operadores de recombinación para ninguna de las dos variables en estudio. Las Figuras 13 y 14 corresponden a el box-plot de los resultados del número de evaluaciones y el tiempo en ejecución respectivamente, donde podemos ver como se distribuyen los valores a través de la mediana. Los resultados obtenidos por cGA con los tres tipos de operadores de cruce son similares tanto en el caso de número de evaluaciones como en el análisis del tiempo en ejecución, todos muestran ser robustos (los valores están compactos y cercanos a la mediana) y bastante parecidos. No obstante para este problema Px es el que obtiene los valores más pequeños.

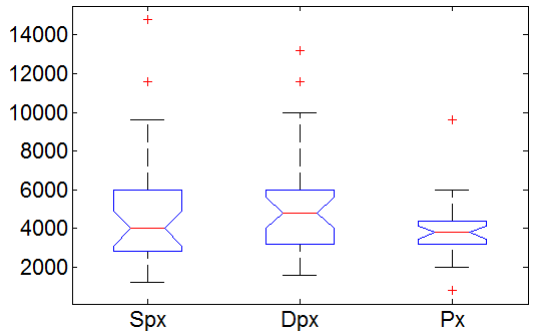

Figura 13: Boxplot de los resultados de la variable "Nºde Evaluaciones" de un cGA con diferentes operadores de recombinación para resolver el problema "cut20.09".

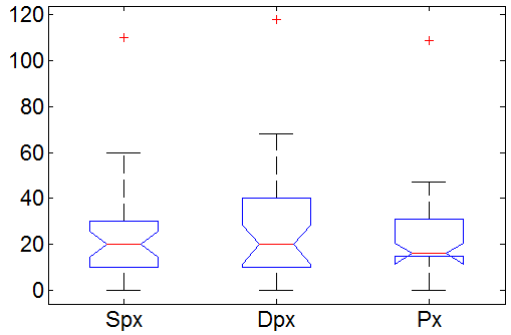

Figura 14: Boxplot de los resultados de la variable de "Tiempo" de un cGA con diferentes operadores de recombinación para resolver el problema "cut20.09".

5. Conclusiones

En el presente trabajo se ha realizado el estudio del comportamiento de un cGA con tres operadores de recombinación diferentes (cruce de un punto, cruce dos puntos y probabilístico). Para ello se han seleccionado una serie de problemas de diversas características (COUNTSAT, ECC, FMS, "cut20.01", "cut20.09", "cut100", MMDP, "mttp20", "mttp100", "mttp200", OneMax y P-PEAK). De los resultados obtenidos se realizó un estudio estadístico detallado sobre las variables de performance, número de evaluaciones y tiempo. Podemos concluir que utilizando la parametrización definida y aplicada a los problemas estudiados, los mejores resultados han sido obtenidos por cGA con el operador de cruce de un punto (Spx) y el operador de cruce (Px). Esto se ve reflejado en los resultados estadísticos pues cGA con Spx y cGA con Px han contendido diferencias estadísticamente significativas con respecto a cGA con Dpx en la mayoría de los problemas para las dos variables de performance analizadas. Se analizaron solamente las 30 corridas exitosas de cGA con los tres operadores de recombinación (ECC, P-PEAK, OneMax, "cut20.01", "cut20.09", "mttp20" y "mttp100").

Como trabajo futuro se estudiaran otras componentes del cGA, en particular operadores de mutación, selección y reemplazo a fin de determinar la mejor configuración de los mismos. Además se pretende realizar alguna propuesta híbrida en alguno de los operadores para mejorar los resultados obtenidos.

6. Agradecimientos

Se agradece la cooperación del equipo de proyecto del LabTEm y la Universidad Nacional de la Patagonia Austral, de los cuales se recibe apoyo continuo.

Referencias

[1] Enrique Alba and Bernabé Dorronsoro. The exploration/exploitation tradeoff in dynamic cellular genetic algorithms. *Evolutionary Computation, IEEE Transactions on*, 9(2):126–142, 2005.

[2] Enrique Alba and José Ma Troya. Cellular evolutionary algorithms: Evaluating the influence of ratio. In *Parallel Problem Solving from Nature PPSN VI*, pages 29–38. Springer, 2000.

[3] Charles Babbage Research Centre and Douglas Robert Stinson. *An introduction to the Design and Analysis of Algorithms*. Charles Babbage Research Centre, 1985.

[4] Hao Chen, Nicholas S Flann, and Daniel W Watson. Parallel genetic simulated annealing: a massively parallel simd algorithm. *Parallel and Distributed Systems, IEEE Transactions on*, 9(2):126–136, 1998.

[5] Kenneth De Jong. Adaptive system design: a genetic approach. *Systems, Man and Cybernetics, IEEE Transactions on*, 10(9):566–574, 1980.

[6] Kenneth A De Jong, Mitchell A Potter, and William M Spears. Using problem generators to explore the effects of epistasis. In *ICGA*, pages 338–345. Citeseer, 1997.

[7] Bernabé Dorronsoro Díaz. *Diseño e implementación de algoritmos genéticos celulares para problemas complejos*. PhD thesis, Universidad de Málaga, 2007.

[8] Stefan Droste, Thomas Jansen, and Ingo Wegener. *A natural and simple function which is hard for all evolutionary algorithms*, volume 4. IEEE, 2000.

[9] M. Garey and D. Johnson. Computers and intractability: A guide to the theory of np-completeness. w.h. freeman. In *New York*, 1979.

[10] Michael R Gary and David S Johnson. Computers and intractability: A guide to the theory of np-completeness, 1979.

[11] David E Goldberg, Kalyanmoy Deb, Hillol Kargupta, and Georges Harik. Rapid, accurate optimization of difficult problems using messy genetic algorithms. In *Proceedings of the Fifth International Conference on Genetic Algorithms,(Urbana, USA)*, pages 59–64. Proceedings of the Fifth International Conference on Genetic Algorithms,(Urbana, USA), 1993.

[12] DE Goldberg. Genetic algorithms in search, optimization, and machine learning, addison-wesley, reading, ma, 1989.

[13] John H Holland. *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press, 1975.

[14] Sami Khuri, Thomas Bäck, and Jörg Heitkötter. An evolutionary approach to combinatorial optimization problems. In *ACM Conference on Computer Science*, pages 66–73. Citeseer, 1994.

[15] F Florence Jessie MacWilliams and NJ Neil James Alexander Sloane. *The Theory of Error-correcting Codes: Part 2*, volume 16. Elsevier, 1977.

[16] J David Schaffer and Larry J Eshelman. On crossover as an evolutionarily viable strategy. In *ICGA*, volume 91, pages 61–68, 1991.

[17] Shigeyoshi Tsutsui, Ashish Ghosh, David Corne, and Yoshiji Fujimoto. A real coded genetic algorithm with an explorer and an exploiter populations. In *ICGA*, pages 238–245, 1997.