GRASP con *Path Relinking* para el problema del *SumCut*

Jesús Sánchez-Oro y Abraham Duarte

Resumen— **En este artículo se propone un algoritmo GRASP combinado con *Path Relinking* para resolver el problema de minimización del *SumCut*. En el problema del *SumCut*, a partir de un grafo de n nodos es necesario etiquetar todos los nodos de forma que cada uno de ellos reciba una etiqueta única del conjunto $\{1, 2, ..., n\}$, con el objetivo de minimizar en cada posición i el número de vértices en una posición $j <= i$ que tienen al menos una arista hacia un vértice en una posición $k > i$. El problema del *SumCut* es muy importante en arqueología (en tareas de serialización) y en el campo de la genética, utilizado por ejemplo en el Proyecto Genoma Humano. Los resultados experimentales muestran que el algoritmo GRASP con *Path Relinking* propuesto supera en términos de desviación media porcentual a los resultados del Estado del Arte utilizando un menor tiempo de ejecución.**

Palabras clave— **metaheurística, GRASP, *Path Relinking*, *SumCut***

I. Introducción

Se define $G = (V, E)$ como un grafo no dirigido, donde V representa el conjunto de vértices y E el conjunto de aristas de G. Dados $n = |V|$ y $m = |E|$, una ordenación lineal φ de los vértices de G es una asignación para cada uno de los vértices del conjunto V de un entero del conjunto $\{1, 2, ..., n\}$, donde cada vértice v tiene una etiqueta única, $\varphi(v) \in \{1, 2, ..., n\}$. Dado un grafo G, y un vértice v situado en la posición $i = \varphi(v)$, se define el conjunto izquierdo $L(i, \varphi, G)$ y el conjunto derecho $R(i, \varphi, G)$ como:

$$L(i, \varphi, G) = \{u \in V : \varphi(u) \leq i\}$$
$$R(i, \varphi, G) = \{u \in V : \varphi(u) > i\}$$

El conjunto $L(i, \varphi, G)$ contiene los vértices a la izquierda de la posición i, incluido el vértice de dicha posición. Por otro lado, el conjunto $R(i, \varphi, G)$ contiene los vértices que quedan a la derecha de la posición i. Teniendo en cuenta la definición de estos dos conjuntos, el *vertex cut* de la posición i de φ se calcula como:

$$\delta(i, \varphi, G) = |\{u \in L(i, \varphi, G) : \exists v \in R(i, \varphi, G) : uv \in E\}|$$

Es decir, el *vertex cut* de la posición i de φ es el número de vértices situados en una posición $j < i$ con al menos un vértice adyacente en una posición $k > i$. El *SumCut* de una ordenación φ, en adelante $SC(\varphi, G)$, se calcula como la suma de todos los δ de cada posición. Formalmente:

Dept. Ciencias de la Computación, Universidad Rey Juan Carlos (España) E-mail: {jesus.sanchezoro, abraham.duarte}@urjc.es

$$SC(\varphi, G) = \sum_{i=1}^{n} \delta(i, \varphi, G)$$

El problema de minimización del *SumCut*, por lo tanto, consiste en obtener el mínimo valor de $SC(\varphi, G)$ para todas las posibles ordenaciones lineales Π_n:

$$SC(G) = \min_{\varphi \in \Pi_n} SC(\varphi, G)$$

En [1] los autores demuestran que el problema del *Profile*, y, por lo tanto, el problema del *SumCut*, son NP-Completos para grafos bipartidos completos. En [2], [3] y [4] se demuestra la misma hipótesis para grafos genéricos. En [5] se demuestra que el problema del *SumCut* es equivalente al problema de minimización *Profile*, con la diferencia de que en dicho problema sería necesario proporcionar la solución inversa a la obtenida. Ambos problemas han sido ampliamente estudiados previamente, por ejemplo en [6], [7] y [8]. Estos problemas son aplicables en genética [9], concretamente en el Proyecto Genoma Humano. El objetivo de dicho proyecto es la secuenciación del ADN de los humanos así como el de otras especies con el objetivo de obtener la información genética que contiene. Para construir un mapa físico de una molécula grande de ADN es necesario extraer copias de ella, para después obtener una huella de cada copia generada. Finalmente, la molécula de ADN se vuelve a ensamblar, siendo necesario determinar cómo se ensamblan las copias entre ellas. Cada copia es una secuencia de nucleotidos obtenidos del conjunto $\{A, C, T, G\}$, por lo que el proceso de ensamblado se basa en permutar una ordenación lineal de un grafo.

Las aplicaciones en arqueología se describen en [10], donde es necesario organizar diferentes artefactos (fósiles, herramientas, joyas, etc.) según un orden determinado. A este proceso se le denomina seriación, y consiste en situar en orden cronológico diferentes artefactos de la misma cultura utilizando un método de establecimiento de la fecha relativo. En concreto, la aplicación práctica se basa en la reordenación de una matriz, lo que se convierte en la reordenación de una ordenación lineal de un grafo.

La reducción del *Profile* de una matriz es también un problema de gran importancia en matemáticas, debido a que colabora en la reducción del espacio necesario para almacenar sistemas de ecuaciones. Además, consigue mejorar el rendimiento de diversas operaciones como la factorización de Choleski de sistemas de ecuaciones no lineales [11]. Recientemente, la reducción del *Profile* se ha utilizado en nuevas

áreas de conocimiento como en *Information Retrieval* para navegar por hipertexto [12].

El problema del *Profile* fue propuesto originalmente como un método para reducir el espacio necesario para almacenar matrices dispersas [13], pero se demostró que era equivalente al problema del SumCut [5]. Otra de las aplicaciones prácticas del problema surge en la copia de huellas dactilares [9].

E. Cuthill y J.McKee proponen el algoritmo RCM (*Reverse Cuthill-McKee*) [14] con el objetivo de obtener el *Profile* mínimo de un grafo. Para generar una solución para el *SumCut* tan sólo es necesario invertir la solución proporcionada por el algoritmo RCM.

N.Gibbs et al. solucionan el problema del *SumCut* utilizando un nuevo algoritmo basado en el RCM [15]. El artículo describe tres problemas del algoritmo del RCM y presenta un nuevo algoritmo que soluciona dichos problemas.

Hasta ahora, la mejor heurística propuesta para obtener soluciones en el problema del *SumCut* es la presentada por R.Lewis [17]. En ella se propone la utilización de Recocido Simulado (*Simulated Annealing*) para reducir el *Profile* de una matriz. El algoritmo comienza con una solución previamente calculada y mejora dicha solución con la utilización de la técnica del Recocido Simulado. La solución inicial se calcula utilizando el algoritmo del RCM o el algoritmo de Gibbs-King, y las instancias utilizadas en la experimentación son un subconjunto de la colección de grafos Harwell-Boeing [18].

II. GRASP

La metaheurística GRASP fue desarrollada a finales de la década de los 80 [19] y el acrónimo se estableció en [20]. Un algoritmo GRASP se puede dividir en dos fases. En la primera fase se construye la solución, y durante la segunda, se mejora la solución construida con un procedimiento de búsqueda local. En esta sección se describen dos métodos de construcción así como dos procedimientos de búsqueda local para el problema del *SumCut*. El Algoritmo 1 muestra el pseudo-código de GRASP.

Algoritmo 1 GRASP($G, nIter$)

```
1: mejorSolucion = ∅
2: for i = 1 : nIter do
3:    solucion = construir();
4:    mejorar(solucion);
5:    if SC(mejorSolucion) < SC(solucion) then
6:       mejorSolucion = solucion;
7:    end if
8: end for
```

A. Métodos constructivos

En este artículo se proponen dos métodos constructivos ($C1$ y $C2$) para el problema del *SumCut*. $C1$ implementa una construcción GRASP típica donde se evalúa inicialmente a cada candidato en base a una función voraz para construir la lista de candidatos restringida (RCL). Finalmente se selecciona un elemento de forma aleatoria de la RCL. $C1$ asigna la etiqueta más baja disponible al vértice seleccionado.

El constructivo $C1$ comienza con la creación del conjunto de vértices no etiquetados U, (incialmente, $U = V$), y el conjunto de vértices etiquetados $L = V \setminus U$. El procedimiento etiqueta un vértice en cada iteración, seleccionando el primer vértice de acuerdo a su grado. En concreto, se selecciona el vértice de menor grado (en caso de empate, se elige aleatoriamente). El vértice u_0 se etiqueta con $l_0 = 1$. Tras ello, los conjuntos U y L son actualizados (es decir, $U = U \setminus \{u_0\}$ y $L = L \bigcup \{u_0\}$).

Una vez se ha asignado la primera etiqueta l_0 al vértice u_0, $C1$ construye la lista de candidatos (CL) con los vértices adyacentes a u_0 que no han sido previamente etiquetados, esto es, $v \notin L$. Tras esto, se evalúan todos los vértices en la CL según una función voraz g. Para este problema se propone la siguiente función voraz:

$$g(v) = |N_L(v)| - |N_U(v)|$$

donde $|N_L(v)|$ indica el número de vértices adyacentes a v que ya han sido etiquetados y $|N_U(v)|$ es el número de vértices adyacentes a v que no han sido etiquetados aún. El método constructivo almacena en g_{min} y g_{max}, respectivamente, el mínimo y el máximo valor obtenido en la función voraz. El siguiente paso consiste en construir la RCL con todos los vértices no etiquetados de la CL que tienen un valor para la función voraz mayor o igual que un valor de umbral determinado, th. Más formalmente:

$$RCL = \{v \in CL : g(v) > th\}$$

donde

$$th = g_{min} + \alpha_1 \cdot (g_{max} - g_{min})$$

y

$g_{min} = \arg \min_{u \in CL} \{g(u)\}$ y $g_{max} = \arg \max_{u \in CL} \{g(u)\}$

El procedimiento $C1$ selecciona un elemento de la RCL de forma aleatoria y le asigna la siguiente etiqueta. Tras ello, se actualiza la CL con los vértices adyacentes a u. El método termina cuando todos los vértices han sido etiquetados.

El siguiente método constructivo $C2$, se basa en otra estrategia donde la selección aleatoria y la voraz se intercambian. Esta estrategia fue introducida en [21] y usada recientemente en [22] y [23]. En concreto, en $C2$ primero se eligen aleatoriamente un número prefijado de candidatos de la CL y posteriormente se evalúan siguiendo el mismo criterio voraz que $C1$. Formalmente, la RCL se construye seleccionando de forma aleatoria $\alpha_2 \cdot |CL|$ elementos de la CL. Tras ello, se selecciona el vértice $u \in RCL$ con mayor valor de g para pasar a formar parte de la solución que se está construyendo. $C1$ y $C2$ utilizan dos parámetros, α_1 y α_2.

B. Procedimientos de búsqueda local

Una solución para el problema del *SumCut* se representa como una permutación. Este tipo de representación de soluciones tiene asociado dos tipos de movimientos: intercambios e inserciones. El primero de ellos consiste en intercambiar las etiquetas de dos vértices diferentes. Es decir, dados dos vértices u y v ($u, v \in V$), el operador $intercambio(u, v)$ asigna la etiqueta $\varphi(u)$ al vértice v y la etiqueta $\varphi(v)$ al vértice u, obteniendo una nueva ordenación φ'.

Por otra parte, el movimiento de inserción consiste en introducir un vértice en una posición diferente a la actual. Es decir, dado un vértice v y una posición i, $insercion(v, i)$ asigna la etiqueta i al vértice v (tras el movimiento, $\varphi(v) = i$), y todos los nodos que se encuentren entre u y v cambiarán sus etiquetas de la siguiente manera:

$$\varphi(v_j) = \begin{cases} \varphi(v_{j+1}) & \text{for } i < j \leq \varphi(v) \\ \varphi(v_{j-1}) & \text{for } i > j \geq \varphi(v) \end{cases}$$

El procedimiento de búsqueda basado en intercambios, *LS_intercambio*, tiene dos parámetros de entrada, la solución construida φ y el grafo G. El Algoritmo 2 muestra el pseudo-código del procedimiento. El algoritmo lleva a cabo movimientos mientras se produzca mejora (líneas 2 a 13). Los vértices se recorren al azar. En concreto, el procedimiento de búsqueda local selecciona de forma aleatoria un vértice v (línea 4) situado en la posición $\varphi(v)$. Entonces, el vértice u (línea 5) es aquel que se encuentra en la posición $\varphi(v) + 1$. Los vértices restantes se exploran en orden lexicográfico, teniendo en cuenta que tras explorar el último vértice (posición $\varphi(u) = n$), se comienza a explorar de nuevo por la primera posición.

El procedimiento intenta llevar a cabo el correspondiente intercambio (línea 6). En caso de que dicho movimiento reduzca el valor de la función objetivo (línea 7), la solución original se actualiza (línea 9). En otro caso, el método lleva a cabo movimientos hasta que no quede ninguno de mejora.

Algoritmo 2 $LS\_intercambio(\varphi, G)$

```
1:  improve = true
2:  while improve do
3:    improve = false
4:    for all v ∈ V do
5:      for all u ∈ V do
6:        φ' = intercambio
7:        if SC(φ', G) < SC(φ, G) then
8:          improve = true
9:          φ = φ'
10:       end if
11:     end for
12:   end for
13: end while
```

La búsqueda local basada en inserciones es similar. En concreto, se sustituye la línea 6 por $\varphi' = insert(u, \varphi(v))$.

El *SumCut* y el *Profile* son problemas donde un único movimiento puede modificar el valor de la función objetivo, debido a que como el valor de la función objetivo se obtiene como la suma del corte en todos los vértices, cada uno de los vértices contribuye al valor de la función objetivo.

III. *Path Relinking*

Path Relinking (PR), fue propuesto en 1977 [24] y actualizado en 1998 [25]. Este algoritmo genera nuevas soluciones explorando trayectorias que conectan soluciones de alta calidad. Para ello, a partir de una de estas soluciones, definida como solución inicial, se genera un camino en el espacio de vecindad que guía a la solución inicial hacia las otras soluciones, denominadas soluciones guía. Para recorrer dicho camino se llevan a cabo movimientos que involucran atributos pertenecientes a la solución guía, e incorporándolos inmediatamente a una solución intermedia generada a partir de la solución inicial. En esta sección se proponen dos adaptaciones diferentes de PR para el problema del *SumCut*.

Se definen $\varphi_X = (4, 2, 3, 5, 1)$ y $\varphi_Y = (3, 4, 5, 1, 2)$ como dos soluciones para el problema del *SumCut*. Dadas dichas soluciones, el algoritmo PR actúa sobre el conjunto D de vértices que no tienen la misma etiqueta en ambas soluciones, es decir, no están situados en la misma posición de la ordenación. Considerando φ_X y φ_Y, el conjunto $D = \{1, 2, 3, 4, 5\}$ ya que todos los elementos tienen etiquetas diferentes. D es por lo tanto el conjunto candidato de vértices a ser examinado. Para crear un camino de X a Y, se localiza en X el vértice u con etiqueta $\varphi_X(u) = \varphi_Y(v)$ y se lleva a cabo el movimiento $intercambio(u, v)$, donde cada intercambio genera una nueva solución intermedia. Tras ello, se elimina el vértice v del conjunto D y se elige el siguiente vértice de D. PR lleva a cabo una selección voraz en cada paso, es decir, evalúa todos los posibles movimientos $intercambio(u, v)$ para todo $v \in V$y lleva a cabo el mejor movimiento en términos de valor de la función objetivo. La Figura 1 muestra un ejemplo de camino generado.

En este artículo se proponen dos algoritmos *Path Relinking* diferentes, *Static Path Relinking* (SPR) y *Dynamic Path Relinking* (DPR).

A. Static Path Relinking (SPR)

El primer paso en el algoritmo SPR es la creación del conjunto de referencia ($RefSet$). Para ello se ejecuta el procedimiento GRASP durante $maxIter$ iteraciones generando un conjunto p de soluciones diferentes ($|p| \leq maxIter$). Del conjunto p se seleccionan b (con $|RefSet| = b$) soluciones donde $b/2$ se seleccionan de acuerdo a su calidad (valor de la función objetivo) y $b/2$ se seleccionan de acuerdo a su diversidad (valor de la función de distancia). La distancia entre dos soluciones se calcula como la su-

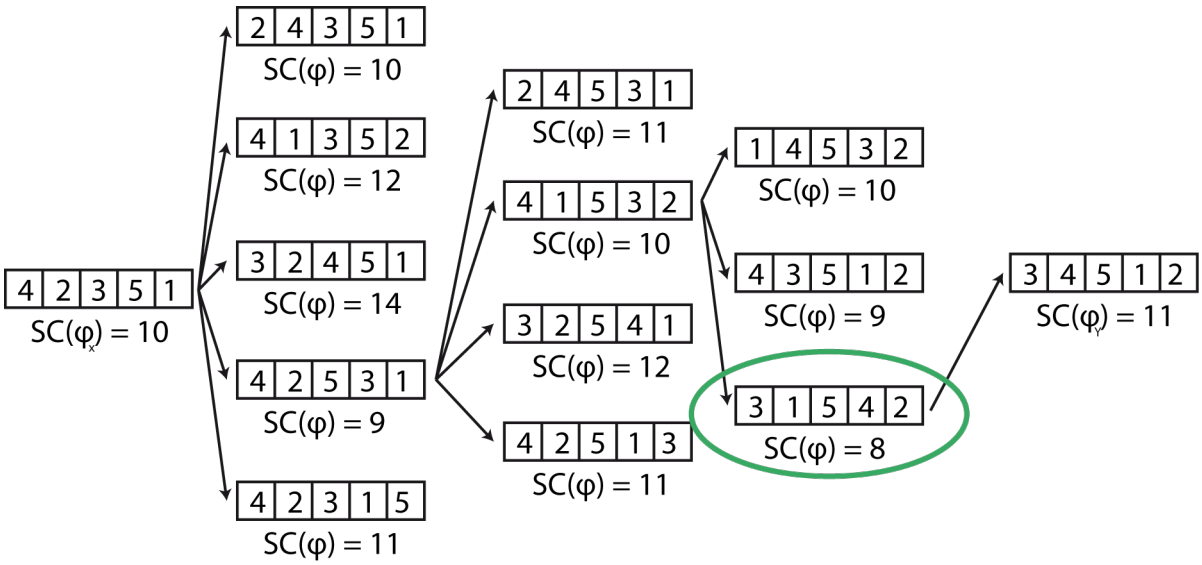

Fig. 1. Camino generado desde la solución φ_X a φ_Y.

ma de las diferencias (en valor absoluto) entre cada par de etiquetas en la ordenación correspondiente. En términos matemáticos:

$$d(\varphi_1, \varphi_2) = \sum_{i \in |\varphi|} |\varphi_1(i) - \varphi_2(i)|$$

Una vez el $RefSet$ se ha generado, el procedimiento SPR genera un camino para cada par de soluciones del $RefSet$. El algoritmo termina cuando todos los pares de soluciones se han combinado. El Algoritmo 3 muestra el pseudo-código del SPR.

Algoritmo 3 $SPR(G, p, b)$

```
1:  refSet = ∅
2:  for i = 1 : p do
3:     sol = construir();
4:     mejorar(sol);
5:     añadirSiMejor(sol, refSet, b/2)
6:     añnadirSiDiversa(sol, refSet, b/2)
7:  end for
8:  for all solInicial ∈ refSet do
9:     for all solGuia ∈ refSet do
10:       if solInicial ≠ solGuia then
11:          mejorCombinacion = combinar(solInicial, solGuia)
12:          actualizarMejor(mejorCombinacion);
13:       end if
14:    end for
15: end for
```

B. Dynamic Path Relinking (DPR)

El algoritmo DPR en primer lugar genera el $RefSet$ ejecutando el procedimiento GRASP y seleccionando del mismo las primeras b soluciones diferentes. Una vez el $RefSet$ ha sido generado, se crea una nueva solución con el procedimiento GRASP, que será utilizada como solución inicial. Tras ello, se selecciona una solución de forma aleatoria del $RefSet$, que será utilizada como solución guía. A partir de estas soluciones, el algoritmo DPR genera un camino desde la solución inicial a la solución guía, almacenando la mejor solución encontrada en el camino. Finalmente, el algoritmo DPR comprueba si la nueva solución cumple una serie de características que le permitan entrar a formar parte del $RefSet$. Para probar si una solución debe entrar en el $RefSet$ es necesario comprobar tres afirmaciones:

1. Si la solución s es mejor que la mejor solución del $RefSet$ (en valor de la función objetivo), s se admite en el $RefSet$.
2. Si la solución s es peor que la peor solución del $RefSet$ (en valor de la función objetivo), s no se admite en el $RefSet$.
3. En otro caso, si la solución s es lo suficientemente diferente (en valor de la función distancia), s se admite en el $RefSet$.

Dada una solución φ_1 construida con el procedimiento GRASP y una solución $\varphi_2 \in RefSet$, dichas soluciones serán suficientemente diferentes si $d(\varphi_1, \varphi_2) > d_{th}$, donde el umbral de distancia se calcula como $d_{th} = \delta \cdot d_{max}$ y d_{max} es la máxima distancia entre dos soluciones, que se puede evaluar como:

$$d_{max} = \sum_{1 \leq i \leq n} |i - (n - i + 1)|$$

Finalmente, si la solución es admitida en el $RefSet$, sustituirá a la solución más parecida del subconjunto de soluciones peores que ella. Es necesario señalar que el parecido entre dos soluciones se evalúa con la función distancia descrita anteriormente. El Algoritmo 4 muestra el pseudocódigo del algoritmo DPR.

Algoritmo 4 $DPR(G, p, b)$

```
1:  refSet = ∅
2:  while tamaño(refSet) < b do
3:     sol = construir();
4:     mejorar(sol);
5:     añadirSiDiferente(sol, refSet)
6:  end while
7:  for all i = 1 : p do
8:     sol = construir();
9:     mejorar(sol);
10:    refSetSol = aleatoria(refSet)
11:    mejorCombinacion =
       combinar(sol, refSetSol)
12:    entrarSiAdmitida(mejorCombinacion, refSet)
13: end for
```

IV. Resultados experimentales

Esta sección describe los resultados experimentales llevados a cabo para probar la eficiencia de la heurística GRASP propuesta. Los métodos han sido implementados en Java y los experimentos se han llevado a cabo sobre un PC Intel Core 2 Duo E4800 a 3.00 Ghz con 3 Gb de memoria RAM.

Se ha utilizado un conjunto de 22 problemas de test en los experimentos. El conjunto de instancias se deriva de la colección de matrices dispersas Harwell-Boeing [18]. Esta colección está formada por un conjunto de matrices generadas a partir de problemas sobre sistemas lineales, mínimos cuadrados, y cálculo de auto-valores de una amplia variedad de disciplinas científicas e ingenierías. Los problemas abarcan desde matrices pequeñas, utilizadas como contra-ejemplos de hipótesis en investigación de matrices dispersas, hasta matrices de gran tamaño con aplicaciones directas. Los grafos se obtienen de dichas matrices de la siguiente manera. Siendo M_{ij} el elemento de la i-ésima fila y j-ésima columna de la matriz dispersa M de tamaño $n \times n$. El grafo correspondiente tiene n vértices. La arista (i, j) existirá en el grafo si y sólo si $M_{ij} \neq 0$.

Los experimentos se dividen en dos partes. En la primera parte, se evalúa el rendimiento de las estrategias propuestas. Para ello se utiliza un subconjunto de 10 ejemplos representativos (de diferente tamaño y densidad). Una vez se ha identificado la mejor combinación de estrategias se comparan con los resultados del Estado del Arte.

En el primer experimento comparamos los dos métodos constructivos propuestos para seleccionar el mejor de ellos. Ambos procedimientos construyen 100 soluciones diferentes, recuperando la mejor de todas las construcciones. Los parámetros α_1 y α_2 se han establecido de forma aleatoria en cada iteración, para incrementar la diversidad de las soluciones generadas. La Tabla I muestra la comparación entre los dos métodos constructivos propuestos ($C1$ y $C2$), el algoritmo RCM, descrito en [14], que es uno de los algoritmos más utilizados para el problema del *SumCut*, y el algoritmo de Gibbs-King (GK), descrito en [15] y [16]. Para cada método se proporciona la desviación porcentual media respecto a la mejor solución conocida (Desv.), el número de veces que cada método obtiene la mejor solución conocida (#Mejores) y el tiempo de ejecución (Tiempo) en segundos. Los valores del RCM y el GK son aquellos publicados directamente por los autores.

TABLA I
Comparativa entre los métodos constructivos

	Desv	**#Mejores**	**Tiempo**
C1	25.64 %	1	0,33
C2	13.32 %	2	0,36
RCM	19.73 %	2	*
C1	15.08 %	1	*

La Tabla I muestra claramente que el mejor constructivo es $C2$ (13.32 %), seguido de GK (15.08 %). Es importante tener en cuenta que el tiempo de ejecución del RCM y el GK no están publicados en los respectivos artículos.

En el segundo experimento se lleva a cabo un estudio sobre cómo los procedimientos de búsqueda local reaccionan con los métodos constructivos. En concreto, se compara el constructivo $C1$ junto a la búsqueda local basada en intercambios ($C1$+LS1), $C1$ con la búsqueda local basada en inserciones ($C1$+LS2) y de la misma forma con $C2$, es decir, $C2$+LS1 y $C2$+LS2. Para evitar tiempos largos de ejecución, todos los algoritmos se paran tras un máximo de 1000 segundos. La Tabla II muestra el rendimiento de los cuatro métodos considerando las mismas estadísticas anteriores.

TABLA II
Comparativa entre las búsquedas locales

	Desv	**#Mejores**	**Tiempo**
C1+LS1	13.21 %	2	228,52
C1+LS2	13.42 %	1	291,87
C2+LS1	8.05 %	1	205,97
C2+LS2	8.46 %	2	264,33

La Tabla 2 muestra que el mejor algoritmo es C2+LS1 en términos de valor de la función objetivo y tiempo de ejecución. Sin embargo, C2+LS2 obtiene un mayor número de mejores soluciones.

En el siguiente experimento se comparan los mejores procedimientos GRASP, GRASP1(C2+LS1) y GRASP2(C2+LS2) utilizando las 22 instancias. Como se ha visto anteriormente, la Tabla II muestra que el mejor algoritmo en cuanto a valor de función objetivo y tiempo de ejecución es GRASP1, pero se ha optado por incluir GRASP2 debido a que encuentra un mayor número de mejores soluciones que GRASP1.

En la Tabla III se puede observar que aunque GRASP2 obtiene mejores soluciones un mayor

TABLA III
COMPARATIVA ENTRE LOS PROCEDIMIENTOS GRASP

	Desv	#Mejores	Tiempo
GRASP1	8.02 %	2	283,78
GRASP2	8.63 %	4	357,36

número de veces, GRASP1 obtiene una menor desviación porcentual y un menor tiempo de ejecución.

El objetivo principal del siguiente experimento es llevar a cabo una comparación entre el *Static Path Relinking* (SPR) y el *Dynamic Path Relinking* (DPR) con el mejor método encontrado en el Estado del Arte. En concreto los dos métodos se comparan con el *Recocido Simulado* (SA) presentado en [17]. Para este experimento, se han ejecutado ambos algoritmos para mejorar los resultados del algoritmo GRASP1. En la Figura IV se puede observar la comparación de los algoritmos SPR y DPR con el algoritmo previo SA.

TABLA IV
COMPARATIVA ENTRE LOS ALGORITMOS SPR Y DPR Y EL RECOCIDO SIMULADO (SA)

	Desv	#Mejores	Tiempo
SPR	1.48 %	11	63,22
DPR	1.06 %	14	45,57
SA	1.68 %	16	179,62

V. CONCLUSIONES

En este artículo se presentan dos métodos constructivos diferentes para el problema del SumCut, así como dos búsquedas locales, obteniendo por lo tanto 4 posibles procedimientos GRASP y dos estrategias de optimización basadas en la metodología *Path Relinking*. Los resultados experimentales muestran que los dos mejor métodos presentados superan al algoritmo previo del Recocido Simulado en términos de desviación porcentual media y tiempo de ejecución. Por otro lado, el Recocido Simulado encuentra un mayor número de veces la mejor solución conocida pero mucho de este mérito parece estar debido al método constructivo GK.

Se propone como trabajo futuro el cálculo eficiente de la función objetivo, de manera que sea posible evaluar una solución en un tiempo menor. Esta mejora llevará al procedimiento a realizar más movimientos en el mismo tiempo, lo que parece que finalizará en una mejor solución. Además, se está intentando desarrollar nuevos métodos de búsqueda local con el objetivo de mover únicamente los nodos críticos basándose en las características de la solución.

AGRADECIMIENTOS

Este trabajo ha sido parcialmente financiado por la Comunidad de Madrid a través del proyecto con referencia S2009/TIC-1542 y el Ministerio de Ciencia y Tecnología con código TIN2009-07516.

REFERENCIAS

[1] J. Yuan, Y. Lin, Y. Liu, and S. Wang, "Np-completeness of the profile problem and the fill-in problem on cobipartite graphs," *Journal of Mathematical Study*, 1998.

[2] J. Díaz, A. Gibbons, M. Paterson, and J. Torán, "The minsumcut problem," in *Algorithms and Data Structures*, Frank Dehne, Jörg-Rüdiger Sack, and Nicola Santoro, Eds., vol. 519 of *Lecture Notes in Computer Science*, pp. 65–79. Springer Berlin / Heidelberg, 1991, 10.1007/BFb0028251.

[3] P.A. Golovach, "The total vertex separation number of a graph," *Diskr. Mat.*, vol. 7, pp. 631–636, 1997.

[4] Yixun Lin and Jinjiang Yuan, "Profile minimization problem for matrices and graphs," *Acta Mathematicae Applicatae Sinica (English Series)*, vol. 10, pp. 107–112, 1994, 10.1007/BF02006264.

[5] R. Ravi, Ajit Agrawal, and Philip Klein, "Ordering problems approximated: single-processor scheduling and interval graph completion," in *Automata, Languages and Programming*, Javier Albert, Burkhard Monien, and Mario Artalejo, Eds., vol. 510 of *Lecture Notes in Computer Science*, pp. 751–762. Springer Berlin / Heidelberg, 1991, 10.1007/3-540-54233-7_180.

[6] Josep Díaz, Mathew D. Penrose, Jordi Petit, and María Serna, "Convergence theorems for some layout measures on random lattice and random geometric graphs," *Comb. Probab. Comput.*, vol. 9, pp. 489–511, November 2000.

[7] J. Díaz, J. Petit, Serna.M., and L. Trevisan, "Approximating layout problems on random sparse graphs," Tech. Rep., Universidad PolitÃ©cnica de Valencia, 2001.

[8] Josep Díaz, Jordi Petit, and Maria Serna, "A survey of graph layout problems," *ACM Comput. Surv.*, vol. 34, pp. 313–356, September 2002.

[9] Richard M. Karp, "Mapping the genome: some combinatorial problems arising in molecular biology," in *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, New York, NY, USA, 1993, STOC '93, pp. 278–285, ACM.

[10] R. Kendall, "Incidence matrices, interval graphs and seriation in archaeology," *STOC'93 Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, 1993.

[11] Y. Saad, *Iterative Methods for Sparse Linear Systems*, PWS Publishing Company, 1996.

[12] Rodrigo A. Botafogo, "Cluster analysis for hypertext systems," in *Proceedings of the 16th annual international ACM SIGIR conference on Research and development in information retrieval*, New York, NY, USA, 1993, SIGIR '93, pp. 116–125, ACM.

[13] R. Tewarson, *Sparse Matrices*, Academic Press, 1973.

[14] E. Cuthill and J. McKee, "Reducing the bandwidth of sparse symmetric matrices," in *Proceedings of the 1969 24th national conference*, New York, NY, USA, 1969, ACM '69, pp. 157–172, ACM.

[15] N E Gibbs, W G Poole, and P K Stockmeyer, "An algorithm for reducing the bandwidth and profile of a sparse matrix," *SIAM Journal on Numerical Analysis*, vol. 13, no. 2, pp. 236–250, 1976.

[16] John G. Lewis, "Algorithm 582: The gibbs-poole-stockmeyer and gibbs-king algorithms for reordering sparse matrices," *ACM Trans. Math. Softw.*, vol. 8, pp. 190–194, June 1982.

[17] Robert R Lewis, "Simulated annealing for profile and fill reduction of sparse matrices," *International Journal for Numerical Methods in*, 1994.

[18] "Matrix market (online),`http://math.nist.gov/MatrixMarket/collections/hb.html`," .

[19] Thomas A Feo and Mauricio G C Resende, "A probabilistic heuristic for a computationally difficult set covering problem," *Operations Research Letters*, vol. 8, no. 2, pp. 67–71, 1989.

[20] T A Feo, M G C Resende, and S H Smith, "A greedy randomized adaptive search procedure for maximum independent set," *Operations Research*, vol. 42, no. 5, pp. 860–878, 1994.

[21] M G C Resende and Renato F Werneck, "A hybrid heuristic for the p-median problem," *Journal of Heuristics*, vol. 10, no. 1, pp. 59–88, 2004.
[22] A. Duarte, E.G. Pardo, and J.J. Pantrigo, "Scatter search for the cutwidth problem," *Annals of Operations Research*, 2010.
[23] M.G.C. Resende, R. Martí, Gallego M., and A. Duarte, "Grasp and path relinking for the max-min diversity problem," *Computers and Operations Research*, 2010.
[24] F Glover, "Heuristics for integer programming using surrogate constraints," *Decision Sciences*, vol. 8, no. 1, pp. 156–166, 1977.
[25] Fred Glover, "A template for scatter search and path relinking," *Computer*, , no. February 1998, pp. 13–54, 1998.