Búsqueda Dispersa genérica para problemas binarios

Francisco Gortázar[1], Abraham Duarte[2], Manuel Laguna[3], Rafael Martí[4]

Resumen—**El propósito de este artículo es describir un método independiente del contexto que hemos desarrollado para resolver aquellos problemas de optimización combinatoria en los que la solución se puede representar mediante variables binarias. Nuestro heurístico de propósito general está basado en un modelo que considera la función objetivo como una caja negra. Hemos realizado experimentos con tres problemas binarios y hemos comparado nuestro método con métodos especializados y software comercial.**

Palabras clave--**Optimización, Metaheurísticas, evaluación de caja negra**

I. INTRODUCCIÓN

Los métodos de optimización de propósito general (también denominados independientes del contexto) tienen una larga tradición en Investigación Operativa. Este tipo de métodos no utilizan información de la estructura del problema para realizar búsquedas en el espacio de soluciones.

Métodos de optimización de estas características pueden encontrarse en software comercial. Por ejemplo, la versión estándar del *Solver* de *Microsoft Excel* contiene tres procedimientos principales de optimización independientes del contexto: simplex acotado para problemas de programación lineal; GRG2 (método basado en el Gradiente Reducido Generalizado) para problemas no lineales; y ramificación y poda para problemas con variables enteras. Cuando la función objetivo no es suficientemente suave, *Solver* pasa por defecto a utilizar GRG2. El procedimiento de ramificación y poda se utiliza con variables enteras y utiliza el método simplex o GRG2 para resolver los sub-problemas de los nodos dependiendo de si el modelo del problema es lineal o no lineal, respectivamente. *Solver* actúa como un procedimiento de propósito general dado que no requiere que el usuario proporcione ningún contexto. La versión *Premium* de *Solver* incluye algoritmos evolutivos. En la Sección IV compararemos nuestra propuesta con el *Solver* Evolutivo.

Una filosofía similar se utiliza en el software comercial de optimización de propósito general denominado *OptQuest* (de *OptTek Systems*, *Inc*.). Este software funciona considerando la función objetivo como una caja negra. *OptQuest* permite representar soluciones como una mezcla de variables continuas, discretas, enteras, binarias, etc. Claramente, la representación de la solución proporciona a *OptQuest* cierta información sobre el contexto del problema; sin embargo, *OptQuest* no trata de obtener ninguna información sobre las características de la función objetivo. El software elige métodos de resolución en función de las características del modelo de optimización: puro o mixto (en función de si se mezclan o no variables de distinto tipo), con restricciones o sin ellas (en función de si deben cumplirse ciertas restricciones antes y/o después de evaluar la función objetivo o no) y determinista o estocástico (en función de si la evaluación de la función objetivo es determinista o no, por ejemplo, la evaluación puede consistir en una simulación Monte Carlo).

Evolver (http://www.palisade.com/evolver/), de *Palisade Corp*., es un software de optimización basado en algoritmos genéticos que utiliza el enfoque de caja negra. Cuando se utiliza junto con *@Risk*, *Evolver* puede utilizarse para buscar soluciones a problemas cuya función objetivo es el resultado de una simulación Monte Carlo. *Evolver* utiliza *Microsoft Excel* como entorno de modelado, donde las variables se declaran como celdas de ajuste. Los valores asignados a estas celdas de ajuste dependen del método de resolución que el usuario seleccione. El método de resolución se especifica a través de la interfaz de *Evolver* y los valores escogidos para cada tipo de variable se introducen en las celdas de ajuste del modelo para evaluar la función objetivo.

Nuestro interés se centra en abordar un subconjunto de los modelos de optimización de caja negra. En particular, estamos interesados en desarrollar procedimientos de optimización para modelos de caja negra computacionalmente costosos para los cuales las variables de entrada son binarias y que pueden contener restricciones.

Además del software comercial, en la literatura se describen diferentes métodos de optimización para problemas de optimización discreta. Por ejemplo, Rosen y Harmonosky [1] propusieron un método de recocido simulado que utiliza una adaptación de la metodología de superficie de respuesta para optimizar simulaciones que contienen

[1] Departamento de Ciencias de la Computación, Universidad Rey Juan Carlos, Francisco.Gortazar@urjc.es
[2] Departamento de Ciencias de la Computación, Universidad Rey Juan Carlos, Abraham.Duarte@urjc.es
[3] Leeds School of Business, University of Colorado at Boulder, laguna@colorado.edu
[4] Departamento de Estadística e Investigación Operativa, Universitat de València, Rafael.Marti@uv.es

una gran cantidad de variables de decisión discretas. Guikema, Davidson y Changan [2] también abordaron el problema de optimizar simulaciones cuando las variables de entrada son discretas. Los algoritmos genéticos propuestos por Holland [3] eran de hecho métodos independientes del contexto que usaban un *array* de bits como representación genérica de una solución. El procedimiento propuesto no incluía búsqueda local y los operadores genéticos no estaban acoplados al contexto del problema. *Cross Entropy* [4] es un método independiente del contexto más reciente que ha alcanzado un rendimiento respetable para algunos problemas combinatorios difíciles. Sin embargo, como muestran Laguna, Duarte y Martí en [5], en ocasiones C*ross Entropy* puede obtener mejores resultados cuando se hibrida con otros métodos.

En el resto del artículo proponemos un procedimiento independiente del contexto para problemas cuyas soluciones se representan como vectores binarios. Utilizamos la metodología de Búsqueda Dispersa (*Scatter Search*, SS) como algoritmo básico para nuestro procedimiento. Aunque la filosofía de SS es aprovechar el conocimiento específico del dominio, también ha sido utilizada como base para métodos independientes del contexto (por ejemplo, en *OptQuest* y en el procedimiento para problemas de permutación presentado en [6]).

II. MÉTODO DE BÚSQUEDA DISPERSA

SS es una metaheurística que explora el espacio de soluciones mediante la evolución de una serie de soluciones de referencia. Esta metaheurística consiste en cinco métodos y sus respectivas estrategias (Fig. 1). De ellos, la generación de soluciones diversas, el método de mejora y el método de combinación de soluciones son típicamente dependientes del contexto y, por tanto, se diseñan pensando en el problema que se quiere resolver. Los otros dos métodos, la actualización del conjunto de referencia (*Reference Set*, *RefSet*) y el método de generación de subconjuntos, son independientes del contexto y existen implementaciones estándar de los mismos [7].

```
1. Generación de soluciones diversas
2. Mejora
3. Actualización del RefSet
Mientras no se alcance el criterio de parada:
      4. Generación de subconjuntos
      5. Combinación
      6. Mejora
      7. Actualización del RefSet
```

Fig. 1. Esquema genérico de la Búsqueda Dispersa

En este trabajo hemos diseñado un método de optimización basado en la metodología SS independiente del contexto para problemas de optimización cuyas soluciones vienen descritas por variables binarias. Asumimos que una instancia del problema consiste en encontrar un conjunto de valores para $x = (x_1, x_2, \ldots, x_n)$, donde $x_i=\{0,1\}$, que maximice una función objetivo que se desconoce. El usuario debe especificar si el problema es sin restricciones o con restricciones. Para problemas sin restricciones, cualquier vector binario de n elementos es una solución factible. Dentro de los problemas con restricciones distinguimos además dos tipos de problemas: de selección múltiple y de consumo de recursos.

Los problemas de selección múltiple son aquellos en los que se deben seleccionar exclusivamente k elementos de un total de n. Esta restricción obliga a que exactamente k variables tomen el valor 1. En términos matemáticos:

$$\sum_{i=1}^{n} x_i = k$$

Los problemas de consumo de recursos son aquellos que contienen restricciones que limitan la cantidad de recursos que puede utilizar cada solución. En este caso, el consumo de recursos aumenta con el número de variables establecidas a 1. La solución se vuelve no factible cuando se exceden los recursos disponibles. Por ejemplo, en el caso del problema de la mochila este tipo de restricción se puede describir con la siguiente formulación:

$$\sum_{i=1}^{n} a_i x_i \leq b$$

En nuestra implementación, la evaluación de la función objetivo debe devolver un valor lógico que indica si la solución evaluada es factible o no. El método no sabe en qué grado la solución no es factible, caso de serlo, sólo sabe que se puede hacer factible cambiando valores de variables de 1 a 0.

Si el problema presenta restricciones que no pueden ser tratadas como selección múltiple o consumo de recursos, entonces dichas restricciones deben incorporarse a la caja negra penalizando el valor de la función objetivo.

El método desarrollado, basado en SS, comienza con la aplicación del método de generación de soluciones diversas, obteniendo una población P de tamaño *PSize*. Con el objetivo de obtener soluciones con diferentes estructuras aplicamos tres generadores diferentes, creando *PSize*/3 soluciones con cada uno de ellos.

El primero, G1, propuesto por [8] utiliza un enfoque sistemático para construir un conjunto diverso de vectores binarios. Este generador de soluciones se aplica directamente en el caso de problemas sin restricciones, y se aplica ligeramente adaptado en el caso de problemas con restricciones. Para problemas de selección múltiple, la

modificación consiste en terminar prematuramente el método cuando el número de elementos seleccionados alcanza el valor de k. Para problemas de consumo de recursos el proceso termina cuando cambiar el valor de una variable a 1 hace la solución no factible.

Una vez generadas *PSize*/3 soluciones con G1, método centrado en la diversificación que no tiene en cuenta la calidad de las soluciones generadas, se calcula el *score(i)* para cada variable x_i, para estimar su contribución al valor de la función objetivo, de forma que al generar los restantes 2/3 de *PSize* se tenga en cuenta tanto la calidad como la diversidad. El cálculo del *score* se realiza mediante la siguiente fórmula matemática:

$$score(i) = \frac{avg_i^1}{avg_i^1 + avg_i^0},$$

$$avg_i^j = \begin{cases} \dfrac{\sum_{x \in P_i^j} f(x)}{\left|P_i^j\right|}, \left|P_i^j\right| > 0, \\ 0.5 \end{cases} \tag{1}$$

$$P_i^1 = \{x \in P : x_i = 1\}, P_i^0 = \{x \in P : x_i = 0\}$$

El segundo generador, G2, construye una solución paso a paso comenzando con todas las variables establecidas a 0, y cambiando una variable a 1 en cada paso. Las variables se seleccionan de manera aleatoria, y la probabilidad de cambiar la variable seleccionada a 1 viene dada por:

$$Prob(x_i = 1) = 0.1 + score(i)$$

El hecho de añadir 0.1 al valor del *score* para la variable i favorece el cambio de valor de 0 a 1 cuando el *score* está cercano a 0.5. Para problemas sin restricciones G2 realiza cambios mientras la solución que se está construyendo mejore. Para problemas de selección múltiple, G2 se detiene cuando k variables se han puesto a 1. Para problemas de consumo de recursos, G2 se detiene cuando la solución se vuelve no factible.

El tercer generador, G3, puede verse como un método destructivo. Fue sugerido por [8] y adaptado por [9]. El método comienza con todas las variables establecidas a 1 y en cada paso cambia el valor de una variable de 1 a 0. Las variables se seleccionan probabilísticamente en base a su *score*. Se utiliza el complemento del *score* ($1 - Prob(x_i = 1)$) para calcular la probabilidad de cambiar el valor de 1 a 0. El criterio de parada es similar al de G2.

Después de cada construcción el valor de los *scores* se actualiza. Hemos comprobado empíricamente que se obtienen mejores resultados si la contribución al *score* de la última solución construida se suaviza. El *score* suavizado, *sscore*, se calcula como una función del valor del *score* en construcciones previas y el valor actual. Si $score_t(i)$ es el *score* de la variable i después de la t-ésima construcción, y $score_{t-1}(i)$ es el *score* en el instante t-1 (antes de la construcción), ambos calculados con la expresión (1), entonces *sscore* se calcula como:

$$sscore(i) = \alpha\, score_{t-1}(i) + (1-\alpha)\, score_t(i)$$

El parámetro α controla la contribución de la solución actual al *score* suavizado. A partir de entonces utilizamos *sscore* en lugar de *score* para generar los valores de las variables en las siguientes construcciones. Cabe destacar que el suavizado de los *scores* es similar al modo en el que los valores de probabilidad son actualizados en el método *Cross Entropy* [4].

En la metodología SS estándar [7] el método de mejora se aplica a todas las soluciones de P. Sin embargo, en los métodos independientes del contexto, la búsqueda local es extremadamente costosa dado que cada movimiento debe ser evaluado invocando el evaluador de caja negra. Por tanto, en nuestro procedimiento el método de mejora se aplica de manera selectiva.

El *RefSet* inicial debe contener soluciones seleccionadas por calidad y por diversidad. Para ello utilizamos el método habitual de actualización del conjunto de referencia, seleccionando de P las mejores b/2 soluciones (donde $b = |RefSet|$) y las restantes b/2 soluciones por diversidad (aquellas que son más diversas respecto a las que ya están en el *RefSet*). La diversidad se calcula utilizando la distancia de Hamming entre soluciones, genérica para cualquier problema binario:

$$d(x, y) = \sum_{i=1}^{n} |x_i - y_i|$$

El método de mejora, que describiremos en la Subsección II.A, se aplica exclusivamente a las b/2 mejores soluciones del *RefSet*. A partir del conjunto de referencia se generan nuevas soluciones mediante la aplicación de un método de combinación. Para conseguir que la metodología se comporte bien con una gran variedad de problemas binarios, proponemos un conjunto de siete métodos de combinación. El método de combinación aplicado en un momento dado se selecciona de manera probabilística en función de su comportamiento en iteraciones anteriores. Los métodos de combinación se describen en la Subsección II.B.

Nuestra metodología utiliza el método tradicional de generación de subconjuntos, generando todos los pares de soluciones del conjunto de referencia que todavía no han sido combinados. De nuevo, la aplicación del método de mejora se realiza exclusivamente sobre las soluciones más prometedoras. Almacenamos las soluciones resultado de aplicar el método de combinación en un *Pool* temporal de soluciones y aplicamos el método de mejora a las b/2 mejores de este *Pool*. El *RefSet* se actualiza con las b mejores soluciones de la unión de *Pool* y *RefSet*. Si el *RefSet* cambia realizamos una nueva iteración, aplicando

de nuevo el método de generación de subconjuntos. Por el contrario, si el método de combinación no es capaz de producir soluciones que puedan mejorar alguna de las contenidas en el *RefSet*, se reconstruye el conjunto de referencia. En la reconstrucción se mantienen las $b/2$ mejores soluciones y se reemplazan las $b/2$ peores con soluciones diversas de P.

A. Método de mejora

Una iteración global del método de mejora consiste en tres pasos. Primero se construye la lista de candidatos *CL* de elementos (variables) susceptibles de ser cambiados:

$$x_i \mid (x_i=0 \text{ y } score(i) \geq th_1) \text{ o } (x_i=1 \text{ y } score(i) \leq th_2)$$

La *CL* contiene las variables con un valor 0 y un *score* grande (cercano a 1) y las variables con un valor 1 y un *score* pequeño (cercano a 0). Es importante señalar que un *score* grande para una variable indica que históricamente se obtuvieron soluciones de mayor calidad cuando dicha variable tomó el valor 1, y a la inversa. Por tanto, de acuerdo a la información del *score*, deberíamos cambiar el valor de las variables de la *CL* (de 0 a 1 y de 1 a 0).

En el segundo paso, se ordenan las variables contenidas en la *CL* de acuerdo a su *score*, poniendo delante aquellas variables con *scores* más cercanos a 0 ó a 1. En el tercer paso, recorremos la lista buscando movimientos de mejora. Realizamos varias iteraciones en este tercer paso alternando movimientos de inversión (cambiar una variable de 0 a 1 ó de 1 a 0), y de intercambio (dos variables distintas intercambian sus valores). En la primera iteración intentamos movimientos de inversión con todas las variables de la *CL* (examinadas en orden), cambiando su valor si dicho cambio mejora el valor de la función objetivo. En la segunda iteración, se realizan intercambios con todas las variables de la *CL* (examinadas en orden), intentando intercambiar el valor de una variable con el de otra. Para ello utilizamos el primer movimiento de beneficio (*first-improvement*), probando variables de la *CL* hasta encontrar la primera cuyo movimiento resulta en una mejora del valor de la función objetivo. Después de este proceso la *CL* se construye de nuevo. Se realizan varias de estas iteraciones alternando inversiones e intercambios. El método se detiene después de *MaxImpIter* iteraciones, o antes si no se puede realizar ninguna mejora tras probar todas las inversiones e intercambios.

B. Método de combinación reactivo

Como se dijo anteriormente, proponemos siete métodos de combinación diferentes y un mecanismo reactivo que selecciona de manera probabilista entre ellos de acuerdo a su habilidad para producir soluciones de calidad en la búsqueda actual. Este método fue aplicado con éxito por [6].

Las soluciones del *RefSet* se ordenan de acuerdo al valor de la función objetivo para cada una de ellas (con la mejor solución en primer lugar). Cuando una solución obtenida por el método de combinación CM_i acaba en la posición j-ésima del *RefSet*, añadimos b-j+1 al valor de *success*(CM_i). Por tanto, métodos de combinación que generan mejores soluciones, acumulan valores de éxito mayores, incrementando la probabilidad de salir seleccionados. Para prevenir efectos no deseados al principio, este mecanismo se activa después de *InitIter* combinaciones, y mientras tanto los métodos se escogen de manera completamente aleatoria. A continuación, se describen los siete métodos de combinación. Estos métodos generan una nueva solución z mediante la combinación de dos soluciones de referencia x e y. El valor del *score*, inicialmente calculado para las soluciones de P, se actualiza durante todo el proceso de búsqueda (es decir, se consideran todas las soluciones examinadas), proporcionando una estimación de la contribución de cada variable (cuando toma el valor 1) al valor de la función objetivo.

El método de combinación CM_1 construye primero una solución parcial z con la unión de x e y, es decir, z_i=1 si x_i=1 o y_i=1, en caso contrario z_i=0. A partir de esta solución parcial realiza una serie de pasos cambiando algunas variables de 1 a 0 de un modo similar a como lo hace el generador G3. En cada paso, el método utiliza los valores del *score* para seleccionar probabilísticamente una variable con valor 1 y cambiarla a 0. Para problemas sin restricciones, este proceso se continúa mientras la solución mejore; para problemas de selección múltiple el método realiza pasos de este tipo hasta que el número de variables a 1 alcanza el valor k. Para problemas de consumo de recursos, se realizan este tipo de pasos hasta que la solución se vuelve factible. Cabe destacar que para este tipo de problemas, la infactibilidad siempre se reduce cambiando variables de 1 a 0. El método de combinación CM_2 es similar a CM_1, con la diferencia de que la selección de la variable a cambiar se realiza de manera equiprobable, sin considerar los valores del *score*.

El método CM_3 es una adaptación de un método propuesto por [7] para el problema de la mochila. El método calcula el peso de cada variable, basándose en el valor de la función objetivo de las dos soluciones de referencia x e y. El peso de la variable i correspondiente a la combinación de x e y se calcula de acuerdo a la siguiente fórmula:

$$weight(i) = \frac{f(x)x_i + f(y)y_i}{f(x) + f(y)},$$

donde $f(x)$ es el valor de la función objetivo para la solución x y x_i es el valor de la i-ésima variable. Si $f(x)+f(y)=0$, entonces $weight(i)=0.05$. La solución z se construye utilizando el peso como probabilidad

de asignar el valor 1 a cada variable. Esto puede implementarse como:

$$z_i = \begin{cases} 1, & r \leq weight(i) \\ 0, & r > weight(i) \end{cases},$$

donde r es un número aleatorio obtenido mediante una distribución uniforme en el intervalo [0,1]. Es importante destacar que este método de combinación asume que la función objetivo está siendo maximizada.

El método de combinación CM_4 primero calcula la solución parcial z formada por la intersección de x e y, es decir, z_i=1, si x_i=1 y y_i=1, en caso contrario, z_i=0. Partiendo de esta solución, se realizan una serie de pasos en los cuales se elige una variable con valor 0. La probabilidad de seleccionar una variable es proporcional al peso de la misma, *weight*(*i*). La diferencia entre CM_3 y CM_4 es que CM_3 comienza con todas las variables a 0, mientras que CM_4 comienza con una solución parcial z que representa la intersección de las soluciones de referencia x e y. Además, el valor de r en CM_3 es un número aleatorio entre 0 y 1, mientras que en CM_4 r=0.5, es decir, CM_4 establece el valor de la variable a 1 ó 0 dependiendo de si su peso supera el valor 0.5 o no, respectivamente. El criterio de parada del método CM_4 es el mismo que el de CM_1, dependiendo del tipo de problema.

El método de combinación CM_5 es similar a CM_4. Se parte de una solución parcial construida como la intersección de las soluciones de referencia, y se escogen variables cuyo valor es 0 para cambiarlas a 1. La diferencia con CM_4 es que la selección se realiza de manera completamente aleatoria, ignorando el peso de las variables.

El método CM_6 comienza asignando a todas las variables el valor 0. A continuación aplica el método constructivo G2 con la restricción de que las variables z_i que pueden ser seleccionadas son aquellas que tienen un valor 1 en x o en y (aquellas para las que $x_i + y_i \geq 1$).

El método de combinación CM_7 está basado en la metodología de Reencadenamiento de Trayectorias (PR) [10]. PR genera nuevas soluciones explorando las trayectorias que unen soluciones de alta calidad. El proceso comienza por una de las soluciones, denominada solución de partida, y genera un camino a través del espacio de vecindad que lleva hasta la otra solución, denominada solución guía. Este proceso se lleva a cabo seleccionando movimientos que van introduciendo atributos de la solución guía en una solución intermedia inicialmente originada a partir de la solución de partida. El método CM_7 explora el camino entre dos soluciones x e y del *RefSet*.

Sean x e y dos vectores binarios, interpretados como dos soluciones de nuestro problema. El procedimiento de reencadenamiento de trayectorias CM_7 propuesto comienza primero con la solución x, y de manera gradual va transformándola en la solución de guía y, cambiando el valor de las variables de x por el correspondiente valor en y (cuando son diferentes). Las variables se examinan en orden lexicográfico y, en n pasos, se llega a la solución y. El procedimiento utiliza la estrategia de primer movimiento de beneficio devolviendo la primera solución encontrada que mejora a x o a y. Si no se encuentra ninguna solución que mejore, se devuelve como solución combinada la solución de la mitad del camino recorrido. Además de explorar el camino de x a y, el procedimiento también explora el camino de y a x, y escoge la mejor solución encontrada entre los dos caminos.

III. PROBLEMAS DE OPTIMIZACIÓN UTILIZADOS

Hemos utilizado tres problemas de optimización combinatoria para probar el comportamiento de nuestro procedimiento: el problema del corte máximo sobre grafos, el problema de la máxima diversidad y el problema de la mochila. Las soluciones a estos tres tipos de problemas se representan de manera natural como vectores binarios.

Escogimos estos problemas porque son bien conocidos, de naturaleza diferente y hay disponibles soluciones óptimas (o de gran calidad) para diferentes instancias de cada uno de los problemas. A continuación, describimos brevemente cada uno de los problemas.

El problema del corte máximo (*MaxCut*) consiste en encontrar una partición del conjunto de vértices de un grafo ponderado no dirigido en dos subconjuntos de forma que se maximice la suma de los pesos de los arcos cuyos extremos se encuentran en subconjuntos distintos. Una solución para este tipo de problemas se puede representar como un vector binario con una cardinalidad igual al número de vértices del grafo, donde el valor 0 ó 1 indica que dicho vértice pertenece a un subconjunto u otro. Este problema encaja dentro de los que denominamos problemas sin restricciones. Desde el enfoque introducido por Sahni y Gonzales [11] diferentes heurísticas y metaheurísticas se han propuesto para este problema. Recientemente, Festa y otros [12] han desarrollado seis algoritmos diferentes basados en búsqueda de vecindad variable, GRASP y PR. Compararemos nuestro método con el algoritmo de búsqueda dispersa propuesto en [13] que obtuvo los mejores resultados.

El problema de la máxima diversidad (*MDP*) consiste en seleccionar un subconjunto de k elementos de un conjunto de n elementos de tal forma que se maximice la suma de las distancias entre los elementos seleccionados. Una solución a este problema puede representarse como un vector binario x si consideramos que cada variable x_i representa un elemento, y su valor, 1 ó 0, indica si

dicho elemento está seleccionado o no, respectivamente. Este problema encaja dentro de lo que hemos denominado problemas de selección múltiple. Se han propuesto diferentes métodos para este problema. Recientemente, Silva y otros [14] han presentado dos enfoques GRASP. Duarte y Martí [15] introdujeron una búsqueda tabú y Gallego y otros [16] han propuesto un método basado en la búsqueda dispersa. Utilizaremos este último método en nuestra comparativa, dado que es el que obtiene los mejores resultados.

El problema de la mochila (*Knapsack*) es un problema bien conocido entre los problemas combinatorios NP-difíciles. El problema consiste en escoger, de un conjunto de ítems, un subconjunto que maximice el valor de la función objetivo sujeto a una restricción en la capacidad. En [17] se proporciona una descripción más detallada del problema de la mochila, y otros problemas relacionados. Pisinger [18] presentó un método especializado para este problema. Utilizaremos este método en los experimentos, comparando los resultados con las soluciones óptimas.

IV. RESULTADOS EXPERIMENTALES

Esta sección describe los experimentos que hemos realizado para comprobar la eficiencia de nuestro procedimiento de búsqueda dispersa independiente del contexto. Hemos implementado los métodos en Java SE 6 y todos los experimentos se ejecutaron en un Pentium 4 a 3 GHz con 2 GB de RAM. Hemos empleado tres conjuntos de instancias en la experimentación.

El conjunto *Max-Cut* contiene 94 instancias obtenidas de dos fuentes. El conjunto Hartmann consiste en 10 instancias con 125 nodos y 375 aristas, todas con peso 1 ó -1. El segundo conjunto consiste en 84 instancias de 50, 100, 200 y 300 nodos generadas con *rudy* (un generador de grafos desarrollado por Giovanni Rinaldi). Las aristas de estos grafos tienen pesos -1, 0 ó 1.

El conjunto *MDP* contiene 92 instancias extraídas de dos fuentes. La primera, conocida como Glover, contiene 40 instancias en las que los valores son distancias euclídeas entre un conjunto de puntos generados aleatoriamente con coordenadas en el rango [0, 100]. La segunda, conocida como Silva, contiene 52 instancias donde los valores son números enteros generados aleatoriamente entre 500 y 100. El valor de n varía entre 50 y 300, y el valor de k entre $0.1n$ y $0.3n$.

El conjunto *Knapsack* contiene 96 instancias de cuatro tipos: fuertemente correladas, subconjunto suma, débilmente correladas y no correladas. Hemos utilizado 24 instancias de cada grupo con n = 100, 300, 1000 y 3000 (6 instancias por cada valor de n).

En cada experimento calculamos para cada instancia el mejor valor (*BestValue*) obtenido de todas las ejecuciones de todos los métodos. A partir de dicho valor, para cada método calculamos el porcentaje de desviación relativa entre la mejor solución obtenida por el método y el *BestValue* para esa instancia. En las tablas indicamos la media de este porcentaje de desviación relativa (columna *Dev.*) entre todas las instancias de cada experimento. También indicamos para cada método el número de instancias en las cuales el valor de la mejor solución obtenida por dicho método fue el *BestValue* para esa instancia (columna *#Best*).

En nuestra experimentación preliminar consideramos un conjunto de 34 instancias representativas formado por 10 instancias *Max-cut*, 8 *MDP* y 16 *Knapsack*. En el primer experimento preliminar estudiamos el parámetro α que utilizamos para actualizar el *score* en los procedimientos de construcción. Para ello se utilizó una búsqueda dispersa básica sin método de mejora y con CM_1 como único mecanismo para combinar soluciones. Con esta implementación pretendemos aislar el efecto de los cambios en el parámetro α. Para esta experimentación establecimos $|P|$ = 300 y un tiempo de CPU límite de un segundo. Dados los resultados de la TABLA I, fijamos para el resto de los experimentos el valor de α en 0.8, porque proporciona la menor desviación.

TABLA I
ANÁLISIS DEL EFECTO DEL VALOR DE ALFA

α	DEV. (%)	#BEST
0.0	8.60	1
0.1	8.66	1
0.2	8.62	0
0.3	9.03	0
0.4	7.80	0
0.5	8.30	1
0.6	7.83	0
0.7	7.80	0
0.8	7.59	1
0.9	7.68	1

En el segundo experimento preliminar comprobamos la efectividad de la *CL* en el método de mejora. Consideramos una versión en la cual los umbrales de los parámetros th_1 y th_2 que definen la *CL* son 0.6 y 0.4 respectivamente, y otra versión en la cual los parámetros son 0 y 1 (lo que equivale a explorar todos los cambios en todas las variables). En esta ocasión activamos el método de mejora, y utilizamos el mismo tiempo límite.

TABLA II
EFECTO DE LA *CL* EN EL MÉTODO DE MEJORA

(th_1, th_2)	DEV. (%)	#BEST
(0, 1)	3.55	3
(0.6, 0.4)	8.14	0

Los resultados de la TABLA II indican que la mejor elección es explorar la vecindad completa y evitar introducir limitaciones en la búsqueda con la lista de candidatos descrita en la Subsección II.A.

En el tercer experimento preliminar ajustamos el parámetro *MaxImpIter* del método de mejora. En este caso se limitó el tiempo de experimentación a 5 segundos. La Fig. 2 muestra el porcentaje de desviación (en media) en el eje de ordenadas y el valor del parámetro *MaxImpIter* en el eje de abcisas.

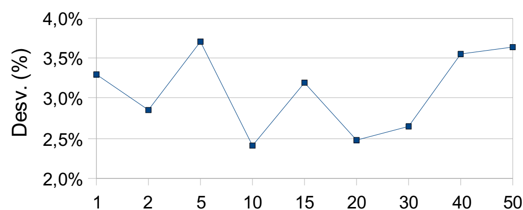

Fig. 2. *MaxImpIter* frente a la desviación

Los mejores resultados se obtuvieron con un valor de *MaxImpIter* de 10. Este es el valor que utilizamos en el resto de experimentos. Con este experimento concluye el ajuste de los mejores parámetros de la búsqueda.

Uno de los elementos clave de nuestra implementación de la búsqueda dispersa es la naturaleza adaptativa de los métodos de combinación. Con el objetivo de conocer si determinados métodos de combinación son mejores que otros para los problemas considerados, registramos el número de veces que se seleccionaba cada uno de los métodos. La TABLA III muestra, en porcentaje, la frecuencia relativa de selección por cada tipo de problema. Para este experimento, utilizamos nuestro procedimiento de búsqueda dispersa completo limitando el tiempo a 5 segundos.

TABLA III

FRECUENCIA DE SELECCIÓN DE LOS MÉTODOS DE COMBINACIÓN

	CM_1	CM_2	CM_3	CM_4	CM_5	CM_6	CM_7
Max-cut	29.93	10.96	5.25	3.94	0.30	27.68	21.94
MDP	23.79	18.61	3.37	0.12	21.18	25.97	6.96
Knapsack	1.67	29.91	11.05	0.62	13.71	9.47	33.57

La TABLA III revela la importancia de introducir varios mecanismos para combinar soluciones de referencia en la metodología de búsqueda dispersa. Si consideramos, por ejemplo, CM_5, este método se utiliza en muy raras ocasiones en el caso de *Max-Cut*, pero es el tercero más usado en el caso de *MDP*. Podemos encontrar otros casos similares, con las posibles excepciones de CM_2 y CM_4. CM_2 se comporta bien en los tres tipos de problemas, mientras que CM_4 se selecciona con poca frecuencia independientemente del problema.

En nuestro experimento principal comparamos nuestro algoritmo de búsqueda dispersa (*BinarySS*) utilizando los parámetros obtenidos como resultado de los experimentos preliminares, con tres conocidos *solvers* comerciales: *Evolutionary Premium Solver*, *OptQuest* y *Evolver*. También consideramos métodos especializados descritos en la Sección III. En el caso de los problemas *Max-Cut* y *MDP* comparamos los diferentes métodos con las mejores soluciones conocidas. En el caso de las instancias del problema de la mochila comparamos con las soluciones óptimas. El tiempo de finalización se fijó en 30 segundos. Sin embargo, algunos métodos terminaron antes debido a su lógica interna.

TABLA IV

RESULTADOS OBTENIDOS SOBRE LOS TRES PROBLEMAS CONSIDERADOS

PROBLEMA	MÉTODO	DEV.	#BEST
MAX-CUT	OPTQUEST	35.49	0
	EVOLVER	17.11	3
	SOLVER	8.10	5
	BINARYSS	4.08	17
	SS (MARTI ET AL 2008)	0.00	90
MDP	OPTQUEST	19.18	0
	EVOLVER	9.03	0
	SOLVER	2.43	1
	BINARYSS	0.17	36
	SS (GALLEGO ET AL 2008)	0.06	87
KNAPSACK	OPTQUEST	9.34	1
	EVOLVER	5.34	11
	SOLVER	50.95	10
	BINARYSS	0.80	34
	EXPKNAP (PISINGER 1995)	0.01	77

El comportamiento de *BinarySS* es muy consistente a lo largo de los tres tipos de problemas (TABLA IV). El método muestra desviaciones bajas respecto a los mejores valores conocidos (o a los valores óptimos.) La gran desviación que presenta *Premium Solver* en el caso de las instancias *Knapsak* se debe a que para las instancias más grandes (con 1000 y 3000 variables) no es capaz de terminar en el tiempo asignado, devolviendo un 100% de desviación. Excluyendo estas instancias la desviación en el resto de casos para este problema es de 0.86%.

En el último experimento volvimos a utilizar el conjunto reducido de instancias que empleamos en los experimentos preliminares. Ejecutamos los procedimientos independientes del contexto durante 10 segundos y registramos cada segundo el valor de la desviación para cada método.

Como se puede observar en la Fig. 3 nuestro método es capaz de encontrar soluciones de gran

calidad desde las fases más tempranas de la búsqueda. No hay ningún momento durante la búsqueda en el cual nuestro método tenga una desviación peor que alguno de los demás. La media de las desviaciones se calcula a partir del valor de la mejor solución encontrada hasta ese momento. En el contexto de simulación-optimización, es importante encontrar soluciones de buena calidad desde las fases más tempranas de la búsqueda, debido a que el horizonte de búsqueda (definido como el número de llamadas al evaluador de la función objetivo) puede estar especialmente limitado.

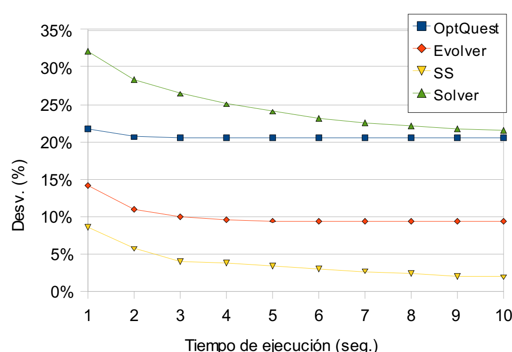

Fig. 3. Evolución del procedimiento de búsqueda.

V. CONCLUSIONES

Hemos descrito el desarrollo de un algoritmo basado en Búsqueda Dispersa para problemas de optimización en los cuales la solución viene dada por un vector binario. Hemos presentado la noción de los métodos independientes del contexto, y cómo los *solvers* tratan de utilizar alguna información, aunque limitada, para mejorar la calidad de las soluciones obtenidas. En el procedimiento que hemos propuesto, distinguimos dos tipos de restricciones respecto de las soluciones. Esto nos permite obtener una cierta ventaja cuando se construyen y se combinan soluciones, puesto que la factibilidad puede alcanzarse más fácilmente que cuando se representan las restricciones como penalizaciones (lineales o no lineales) de la función objetivo.

Los resultados de nuestros experimentos, realizados sobre 282 instancias de los tres tipos de problemas, son muy positivos respecto a la efectividad del método que hemos desarrollado. Durante el desarrollo del método hemos podido observar las ventajas de utilizar diferentes métodos de combinación y las desventajas de reducir el tamaño del espacio de búsqueda en el método de mejora. La estrategia basada en el primer movimiento de beneficio, cuando se aplica a toda la vecindad, resulta en un uso efectivo del tiempo de búsqueda. Esperamos que nuestra experiencia pueda ser de ayuda para mejorar los *solvers* comerciales basados en procesos evolutivos.

AGRADECIMIENTOS

Este trabajo de investigación ha estado parcialmente financiado por el Ministerio de Educación y Ciencia (TIN2006-02696 y SEJ2005-08923/ECON) y el Gobierno de Castilla y León (BU008A06).

REFERENCIAS

[1] Scott L. Rosen y Catherine M. Harmonosky, An Improved Simulated Annealing Simulation Optimization Method for Discrete Parameter Stochastic Systems, Computer and Operations Research, vol. 32, pp. 343—358, 2005.
[2] Seth D. Guikema, Rachel A. Davidson y Zehra Çagnan, Efficient Simulation-Based Discrete Optimization, En Proceedings of the 2004 Winter Simulation Conference.
[3] John H. Holland, Adaptation in Natural and Artificial Systems, MIT Press, 1992.
[4] Reuven Y. Rubinstein y Dirk P. Kroese, The Cross-Entropy Method: A Unified Approach to Combinatorial Optimization, Monte-Carlo Simulation, and Machine Learning, Springer-Verlag, New York, 2004.
[5] Manuel Laguna, Abraham Duarte y Rafael Martí, Hybridizing the Cross Entropy Method: An Application to the Max-Cut Problem, Computers and Operations Research, 2007.
[6] Vicente Campos, Manuel Laguna y Rafael Martí, Context-Independent Scatter and Tabu Search for Permutation Problems, INFORMS Journal of Computing, vol. 17, no. 1, pp. 111—122, 2005.
[7] Manuel Laguna y Rafael Martí, Scatter Search: Methodology and Implementations in C, Springer, 2003.
[8] Fred Glover, A Template for Scatter Search and Path Relinking, Lecture Notes in Computer Science, vol. 1363, pp. 13—54, 1998.
[9] Abraham Duarte y Rafael Martí, Tabu Search for the Maximum Diversity Problem, European Journal of Operational Research, vol. 178, pp. 71—84, 2007.
[10] Fred Glover y Manuel Laguna, Tabu Search, Kluwer Academia Publishers, Boston, 1997.
[11] Sartaj Sahni y Teofilo Gonzales, P-Complete Approximation Problem, Journal of the Association of Computing Machinery, vol. 46, pp. 48—61, 1976.
[12] Paola Festa, Panos M. Pardalos, Mauricio G. C. Resende y Celso C. Ribeiro, Randomized Heuristics for the Max-Cut Problem, Optimization Methods and Software, vol. 7, pp. 1033—1058, 2002.
[13] Rafael Martí, Abraham Duarte y Manuel Laguna, Advanced Scatter Search for the Max-Cut Problem, INFORMS Journal of Computing, 2008.
[14] Geiza C. Silva, Luiz S. Ochi y Simona L. Martins, Experimental Comparison of Greedy Randomized Adaptive Search Procedures for the Maximum Diversity Problem, Lecture Notes in Computer Science, vol. 3059, pp. 498—512, 2004.
[15] Abraham Duarte y Rafael Martí, Tabu Search for the Maximum Diversity Problem, European Journal of Operational Research, vol. 178, pp. 71—84, 2007.
[16] Micael Gallego, Abraham Duarte, Manuel Laguna y Rafael Martí, Heuristics Algorithm for the Maximum Diversity Problem, Computational Optimization and Applications, 2008.
[17] Silvano Martello y Paolo Toth, Knapsack Problems: Algorithms and Computer Implementations, John Wiley & Sons, New York, 1990.
[18] David Pisinger, An Expanding-Core Algorithm for the Exact 0-1 Knapsack Problem, European Journal of Operational Research, vol. 87, pp. 175—187, 1995.