Búsqueda de Vecindad Variable secuencial y paralela: una aplicación al problema de la maximización del corte

D. Concha, D. Fernández-López, J. Sánchez-Oro, A.S. Montemayor, J.J. Pantrigo, and A. Duarte[1]

Dept. Ciencias de la Computación, Universidad Rey Juan Carlos (Spain)
{ david.concha, david.fernandez, jesus.sanchezoro, antonio.sanz, juanjose.pantrigo, abraham.duarte }@urjc.es

Resumen En este trabajo se propone un análisis comparativo de la implementación paralela y secuencial de un algoritmo basado en el esquema *Variable Neighborhood Search*. La paralelización se lleva a cabo utilizando NVIDIA CUDA sobre GPU. Para ilustrar el comportamiento de ambas propuestas se utiliza el problema de la maximización del corte (MaxCut Problem - MCP). Dado un grafo G no dirigido, se define un corte como la división del conjunto de vértices de G en dos subconjuntos S y S' de forma que $S' = V \setminus S$. El MCP consiste entonces en encontrar el corte de máximo valor en G de entre todos los posibles. Para resolver dicho problema se propone un esquema basado en la estrategia *Basic VNS*. La paralelización se aplica a la búsqueda local, analizando las diferencias entre la búsqueda local paralela y la secuencial. Los resultados obtenidos muestran como la paralelización de la búsqueda local en GPU consigue reducir en gran medida el tiempo de ejecución, proporcionando soluciones de mayor calidad.

Keywords: maxcut, paralelismo, NVIDIA CUDA, variable neighborhood search, metaheurísticas

1. Introducción

Se define un grafo $G = (V, E)$, con $n = |V|$ y $m = |E|$, como un conjunto de vértices V y el conjunto de aristas que los unen E. Siendo w_{ij} el peso asociado con la arista $(i, j) \in E$, se define un $corte(S, S')$ como la división de V en dos conjuntos S y $S' = V \setminus S$. El valor de dicho corte se calcula mediante la expresión

$$corte(S, S') = \sum_{u \in S, v \in S'} w_{uv}$$

La Figura 1 muestra un ejemplo de grafo G no dirigido (izquierda), así como uno de los posibles cortes para dicho grafo (derecha). El corte de la figura se calcula de la siguiente manera: $corte(S, S') = 1 + 0 + (-1) + (-1) + 0 = -1$.

El problema de la maximización del corte (MaxCut Problem - MCP) consiste en encontrar el corte de máximo valor en G de entre todos los posibles. En [10]

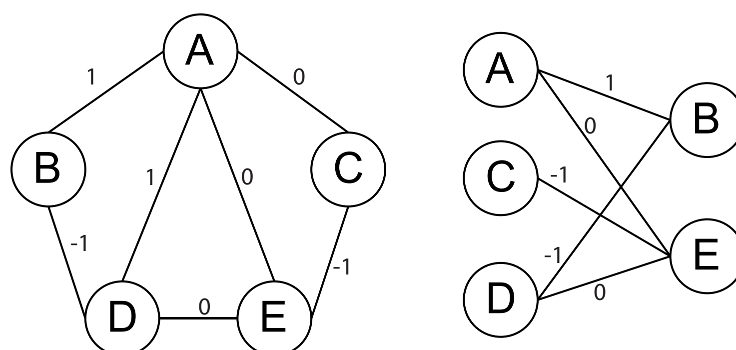

Figura 1. Ejemplo de grafo no dirigido con pesos en las aristas (izquierda) y uno de los posibles cortes (derecha)

se demuestra que el MCP es NP-difícil. El MCP es un problema de optimización con un gran número de aplicaciones prácticas ([4,5]) que ha sido ampliamente trabajado en los últimos años.

La primera solución propuesta para el MCP se encuentra en [18] y más tarde se presentan algunas variantes ([17,11]). En [8] se obtienen cotas superiores y un método heurístico con un rendimiento garantizado de 0.878. En [13] se propone una paralelización de dicho método. En cuanto a trabajos recientes, en [2] se propone un método basado en una relajación que genera soluciones mejores que los métodos anteriores en un menor intervalo de tiempo. En [6] se proponen seis algoritmos diferentes basados en metodologías *Variable Neighborhood Search* (VNS), *Greedy Randomized Adapted Search Procedures* (GRASP) y *Path Relinking* (PR). En dicho trabajo se muestra como la metodología VNS junto con PR obtiene soluciones de alta calidad pero necesitando largos períodos de tiempo. Finalmente, en [14] se propone un algoritmo basado en *Scatter Search* combinado con *Path Relinking* que genera soluciones de alta calidad en un menor período de tiempo.

La paralelización de algoritmos está siendo muy estudiada en los últimos años. En [16] aparece una de las primeras paralelizaciones sobre GPU (Graphics Processing Units) de algoritmos heurísticos, aplicado sobre el mismo problema que se trata en este trabajo. En dicho trabajo se propone una nueva representación del problema, modelando la solución basándose en las aristas en lugar de en los vértices. Dicho modelo era necesario en ese momento debido a las limitaciones técnicas que existían a la hora de trabajar con el hardware gráfico. Con la evolución de los paradigmas paralelos de programación hoy día es más sencillo aprovechar el potencial del hardware gráfico, pero cabe destacar la importancia de hacer una buena adaptación del modelo del problema para que sea eficiente en la plataforma donde se van a realizar las pruebas, ya que, de otra forma, es posible que el modelo paralelo llegue a ser más lento que el secuencial. Además, en [7,3] se proponen varias estrategias de paralelización de la metodología Varia-

ble Neighborhood Search aplicadas al problema de la p-mediana. Los sistemas paralelos se pueden clasificar mediante la taxonomía de arquitecturas paralelas de Flynn, la cual se basa en dos criterios: el número de flujos de instrucciones (FI) y el de flujos de datos (FD), definiendo cuatro modelos diferentes: SISD (1-FI, 1-FD), SIMD (1-FI, varios FD), MISD (varios FI, 1-FD) y MIMD (varios FI, varios FD). En el contexto de las arquitecturas gráficas (GPU), SIMD es el modelo más común, siendo el elegido para este trabajo. En la actualidad, las tecnologías más utilizadas para la paralelización de algoritmos sobre GPU son OpenCL y NVIDIA CUDA. OpenCL es un estándar multiplataforma de programación de procesadores modernos (tanto CPU como GPU) que permite tanto paralelismo de datos como de tareas. Por otro lado, NVIDIA CUDA es una plataforma y un modelo de programación orientado al paralelismo de datos sobre sobre GPU. En este trabajo se elige NVIDIA CUDA como tecnología de paralelización debido a que presenta una mayor estabilidad y rendimiento.

NVIDIA CUDA es un API (*Application Program Interface*) para la programación de propósito general de procesadores gráficos NVIDIA. Si bien el paradigma de programación es muy similar al tradicional, es necesario conocer ciertos aspectos internos del hardware gráfico para poder aprovechar el rendimiento que puede llegar a obtener.

La arquitectura unificada de las GPU de NVIDIA está formada por varios multiprocesadores de multitud núcleos cada uno, llegando en total a ofrecer cientos ó miles de núcleos CUDA (CUDA-cores), denominándose arquitectura masivamente paralela. Para aprovechar el paralelismo masivo de NVIDIA CUDA es necesario lanzar una gran cantidad de hilos de ejecución, que se agrupan en bloques. Un bloque es ejecutado por un multiprocesador y los distintos núcleos de dicho multiprocesador se encargan de ejecutar los hilos de ese bloque. Una tarea es llevada a cabo por varios bloques que se agrupan en un *grid*. Se denomina kernel a aquella función que es ejecutada en la GPU por un *grid*. Un kernel tiene el código que ejecuta un hilo, y como el modelo de ejecución de una tarjeta gráfica es SIMD, todos los hilos ejecutan el mismo código.

NVIDIA CUDA aplica una serie de instrucciones a un conjunto muy amplio de datos. Por lo tanto, para sacar el mayor rendimiento posible es necesario conocer la jerarquía de la memoria en NVIDIA CUDA para poder transmitir los datos suficientemente rápido.

Inicialmente los datos están en memoria principal (CPU). La primera memoria que muestra NVIDIA CUDA es la que denomina *memoria global*. Esta memoria es similar a la memoria principal de la CPU pero en la tarjeta gráfica. Todos los hilos del *grid* pueden leer y escribir a esta memoria pero su acceso es el más lento de todos (a pesar de que esta memoria tiene mayor ancho de banda que la principal). A continuación se puede encontrar la memoria constante, que mejora el rendimiento de la memoria global gracias a un sistema de *cacheo* a cambio de ser una memoria relativamente pequeña de solo lectura. Dentro de cada bloque se encuentra la memoria compartida, la cual es una memoria de lectura y escritura muy rápida, pero que solo existe dentro de cada bloque, por lo tanto solo permite compartir información entre los hilos de un mismo bloque,

no siendo posible la comunicación entre bloques. Por ultimo, los registros aparecen en cualquier procesador, pero que en el caso de las GPU es un factor a tener en cuenta, ya que su número es limitado y hacer un uso intensivo de ellos puede traducirse en movimientos de datos a través de la memoria global, lo que repercute en el rendimiento.

2. Búsqueda de Vecindad Variable

El objetivo de este trabajo es llevar a cabo una comparativa entre un algoritmo secuencial tradicional y su implementación paralela sobre GPU para ilustrar el comportamiento de la paralelización de algoritmos sobre GPU aplicado sobre el problema del MCP. Para ello se propone un algoritmo basado en la metodología *Variable Neighborhood Search* (VNS), propuesta en [15]. VNS es una metaheurística para resolver problemas de optimización basada en un cambio sistemático de estructuras de vecindad. Existen diversas variantes de VNS, entre las que destacan: *Variable Neighborhood Descent* (VND), *Reduced VNS* (RVNS), *Basic VNS* (BVNS), *Skewed VNS* (SVNS), *General VNS* (GVNS) y *Variable Neighborhood Decomposition Search* (VNDS), entre otras (ver [9] para más información). En este trabajo se propone una variante basada en BVNS que combina cambios de vecindad estocásticos y deterministas, descrita en el Algoritmo 1. En este problema, se define la vecindad k de una solución φ, $N_k(\varphi)$ como el conjunto de soluciones en los cuáles hay exactamente k nodos situados en diferentes posiciones del corte respecto a la solución de la que se parte (φ).

Algorithm 1 BVNS(k_{max}, t_{max})

1: $\varphi \rightarrow Construir()$
2: **repeat**
3: $\quad k \rightarrow 1$
4: $\quad$ **repeat**
5: $\quad\quad \varphi' \rightarrow Shake(\varphi, k)$
6: $\quad\quad \varphi'' \rightarrow BusquedaLocal(\varphi'')$
7: $\quad\quad CambioVecindad(\varphi, \varphi'', k)$
8: $\quad$ **until** $k = k_{max}$
9: $\quad t \rightarrow Time()$
10: **until** $t = t_{max}$
11: **return** φ

El algoritmo comienza a partir de una solución inicial (paso 1). En este trabajo se considera una solución inicial aleatoria. El algoritmo se ejecuta durante un tiempo de ejecución predefinido t_{max} (pasos 2-10). El procedimiento de búsqueda comienza en la primera vecindad (paso 3). El algoritmo entonces lleva a cabo una serie de cambios estocásticos en las estructuras de vecindad hasta alcanzar la mayor vecindad permitida (pasos 4-8). Dentro del bucle principal se pueden distinguir los tres métodos más importantes: $Mezcla$, $BusquedaLocal$ y

CambioVecindad. En el método *Mezcla* se genera una solución en la vecindad k actual. A la solución generada se le aplica el procedimiento de búsqueda para encontrar un óptimo local. Por último, el cambio de vecindad comprueba si la nueva solución es mejor que la mejor solución encontrada hasta el momento. En caso afirmativo, se actualiza la mejor solución y se vuelve a la primera vecindad. En caso contrario, se pasa a la siguiente vecindad. El algoritmo finaliza cuando se ha alcanzado el tiempo máximo de ejecución permitido t_{max}.

La estructura de la solución utilizada es un vector de n elementos donde cada elemento representa a un nodo del grafo, que tendrá valor 0 si pertenece a un lado del corte y 1 si pertenece al otro lado. De esta forma se propone como movimiento el cambio de lado de corte de un nodo. Es decir, el movimiento $Cambio(v, \varphi, G)$ vendrá definido como:

$$Cambio(v, \varphi, G) = \begin{cases} \varphi(v) \to 1 \; si \; \varphi(v) = 0 \\ \varphi(v) \to 0 \; si \; \varphi(v) = 1 \end{cases}$$

El procedimiento de $Mezcla(\varphi, k)$ propuesto en este trabajo selecciona k vértices de manera aleatoria y realiza con ellos el movimiento *Cambio* anteriormente definido, alterando la solución de forma aleatoria para incrementar la diversidad del algoritmo.

Algorithm 2 $LS(\varphi, G))$

```
1: mejora → TRUE
2: while mejora do
3:    mejora → FALSE
4:    φ_best → φ
5:    for v ∈ V do
6:       φ' → Cambio(v, φ, G)
7:       if MCP(φ') > MCP(φ_best) then
8:          φ_best → φ'
9:          mejora = TRUE
10:      end if
11:   end for
12:   t → Time()
13: end while
14: return φ_best
```

La parte intensificadora del procedimiento se basa en las búsquedas locales. En este trabajo se presenta una búsqueda local con dos implementaciones, una de forma secuencial en CPU y la otra de forma paralela en GPU. Se trata de una búsqueda local exhaustiva en la que en cada iteración para cada vértice v de una solución φ sobre un grafo G se lleva a cabo el movimiento $Cambio(v, \varphi, G)$ y se comprueba si la solución resultante, φ' es mejor que la mejor solución encontrada hasta el momento (φ_{best}). En caso afirmativo se actualiza la mejor solución y se continúa con la siguiente iteración. La búsqueda local termina cuando no es

posible llevar a cabo el movimiento sobre ningún vértice del grafo. El Algoritmo 2 muestra el pseudocódigo de la búsqueda local propuesta.

La Figura 2 ilustra cómo se llevan a cabo los movimientos en la búsqueda local. Se define una solución al problema como un vector de dimensión igual al número de nodos, donde el conjunto al que pertenece el nodo i viene dado por el valor de la posición i del vector. Las aristas con línea continua en la Figura 2 son aquellas que afectan a la función objetivo, ya que unen dos nodos situados en diferentes conjuntos. Por otra parte, las aristas con línea discontinua son las que no contribuyen, por unir nodos del mismo conjunto. La zona sombreada separa los dos conjuntos de manera gráfica. El nodo A estará situado en la posición 1, el nodo B en la posición 2 y así sucesivamente hasta el nodo E en la posición 5. La Figura 2.(a) tiene como vector solución $\varphi_a = [1, 1, 0, 0, 0]$, mientras que el vector solución de la Figura 2.(b) sería $\varphi_b = [1, 1, 0, 1, 0]$.

Dada una solución φ sobre la que se aplicará un movimiento de cambio, se puede calcular el valor de la nueva solución únicamente atendiendo a las aristas del nodo que se mueve. El valor de la función objetivo para el caso de la Figura 2.(a) es la suma de los costes de las aristas 2, 3, 4 y 5; por otra parte, el valor para la Figura 2.(b) es la suma de los costes de las aristas 2, 4, 5, 6 y 7.

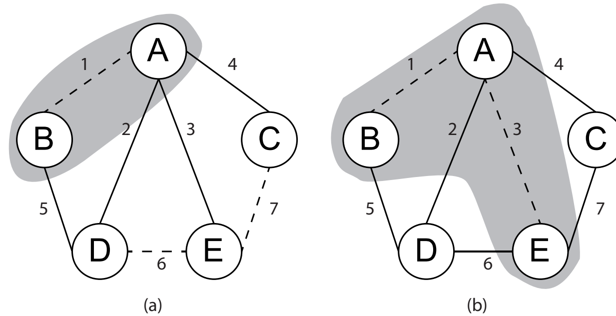

Figura 2. Ejemplo de movimiento del nodo D en la búsqueda local

Si partimos de φ_a, para calcular el valor de la solución φ_b sólo es necesario tener en cuenta las aristas del nodo a mover, en este caso el nodo D (aristas 3, 6 y 7). Si observamos la solución φ_a, las aristas cuyos nodos están en distinto conjunto, se restarán, como es el caso de la arista 3 entre los nodos A y B. Por el contrario las aristas de nodos que están en el mismo conjunto se sumarán, como son las aristas 6 y 7 entre el nodo D y los nodos C y E, respectivamente. Esto es así ya que al mover un nodo, las aristas que antes sumaban por estar en distinto conjunto, dejaran de hacerlo, y las que no sumaban pasaran a hacerlo.

Dada una pareja de nodos i y j, la operación en GPU que calcula el aporte que realiza a la función objetivo la arista entre i y j es la siguiente:

$$w_{(i,j)} = \begin{cases} M[i,j] \text{ si } S[i] = S[j] \\ -M[i,j] \text{ si } S[i] <> S[j] \end{cases}$$

siendo $w_{(i,j)}$ el peso que aporta la arista (i,j) a la solución (con valor 0 si $(i,j) \notin E$), S el vector solución antes del movimiento y M la matriz de adyacencias.

Posteriormente se suman todas las aportaciones de un nodo, lo que refleja cómo afecta al valor de solución realizar un movimiento sobre ese nodo. Se elije el nodo que más aporte, y si el aporte es positivo, se realiza dicho movimiento. El algoritmo se repite hasta no encontrar ningún movimiento que mejore.

3. Resultados experimentales

Esta sección está destinada a comparar el rendimiento del algoritmo BVNS utilizando las dos búsquedas locales propuestas. El algoritmo ha sido implementado en C++ y NVIDIA CUDA y todos los experimentos se han llevado a cabo sobre un Intel Core 2 Duo E8400 CPU (3 GHz) con 4 GB de RAM. La GPU utilizada para la implementación paralela es el modelo NVIDIA GTX-680. El conjunto de instancias utilizado se compone de 88 instancias divididas en tres conjuntos diferentes:

1. **Conjunto 1**: Se compone de instancias generadas por [12] que utiliza un generador de grafos para crear 54 instancias cuyo número de nodos varía entre 800 y 3000. Se trata de grafos cuyos pesos pueden valer -1,0,1 utilizados en [6,2].
2. **Conjunto 2**: Contiene 30 instancias descritas en [6]. Las 10 primeras son de pequeño tamaño con un promedio de 128 vértices y 300 aristas, las 10 siguientes de medio tamaño (promedio de 1000 vértices y densidad 0.60 %) y las 10 últimas son instancias grandes, con un promedio de 2744 vértices y densidad igual a 0.22 %. Los pesos de las aristas varían entre -1,0 y 1.
3. **Conjunto 3**: Se compone de cuatro instancias obtenidas del 7th DIMACS Implementation Challenge. El número de vértices está entre 1536 y 10125. En [2] se utilizan estas cuatro instancias mientras que en [6] se utiliza solo una de ellas.

Las instancias están disponibles en `http://www.optsicom.es/maxcut/`. En el experimento se lleva a cabo una comparativa del algoritmo BVNS utilizando la búsqueda local secuencial en CPU (VNS_CPU) frente al BVNS utilizando la búsqueda local paralela en GPU (VNS_GPU). Ambos algoritmos utilizan un valor de $k_{max} = 0.20 \cdot n$, es decir, el valor máximo que tomará k será el 20 % del número de nodos de cada instancia. La Tabla 1 muestra los resultados obtenidos en la ejecución de los dos algoritmos propuestos sobre cada conjunto de instancias descritas anteriormente. De cada algoritmo se reporta el promedio de la función objetivo (F.O.), el tiempo de ejecución medido en segundos (Tiempo), el promedio del porcentaje de la desviación estándar respecto al mejor valor del experimento (Desv.) y el número de mejores soluciones encontradas (#Mejores).

En los resultados obtenidos para el Conjunto 1 se puede apreciar como, aunque VNS_GPU es notablemente más rápido que VNS_CPU, este último proporciona soluciones ligeramente mejores, con un 0.1 % menos de desviación respecto al mejor valor obtenido en el experimento y dos mejores soluciones encontradas.

Conjunto 1				
Algoritmo	F.O.	Tiempo (seg.)	Desv. (%)	#Mejores
VNS_CPU	5071.3	249.1	0.5	28
VNS_GPU	5076.9	71.9	0.4	26
Conjunto 2				
Algoritmo	F.O.	Tiempo (seg.)	Desv. (%)	#Mejores
VNS_CPU	1053.5	301.6	1.3	10
VNS_GPU	1049.7	56.6	1.3	13
Conjunto 3				
Algoritmo	F.O.	Tiempo (seg.)	Desv. (%)	#Mejores
VNS_CPU	78426958.8	548.6	1.1	2
VNS_GPU	78426964.3	286.2	0.0	4

Tabla 1. Resultados obtenidos por los dos algoritmos propuestos sobre cada conjunto de instancias

Sin embargo, analizando los resultados del experimento sobre los Conjuntos 2 y 3, se puede observar como VNS_GPU proporciona mejores resultados en un menor tiempo de ejecución.

Una de las medidas más utilizadas para comparar algoritmos secuenciales frente a algoritmos paralelos es el *speedup* [1]. Dicha medida establece una relación entre el tiempo de ejecución paralelo y el secuencial para indicar cuánto de rápido es el algoritmo paralelo frente al secuencial. Sin embargo, esta medida solo es significativa si el algoritmo paralelo es una traducción directa de la versión secuencial del algoritmo [3]. En este trabajo se da dicha situación, por lo que resulta de interés llevar a cabo un análisis del *speedup* obtenido sobre cada conjunto de instancias y sobre el total de las mismas. El cálculo del *speedup* se lleva a cabo de la siguiente manera: $speedup = T_s/T_p$ siendo T_s el tiempo utilizado por el algoritmo secuencial y T_p el tiempo requerido por el algoritmo paralelo.

Instancias	Tiempo VNS_CPU	Tiempo VNS_GPU	Speedup
Conjunto 1	249.1	71.9	3.5
Conjunto 2	301.6	56.6	5.3
Conjunto 3	548.6	286.2	1.9
Total	280.6	76.4	3.7

Tabla 2. Comparativa del *speedup* obtenido en cada conjunto de instancias frente al *speedup* global.

Como se puede apreciar en la Tabla 2 el *speedup* obtenido sobre el conjunto total de instancias es de 3.67, siendo 5.33 el máximo y 1.92 el mínimo. Esto significa que, en términos generales, el algoritmo paralelo es entre 2 y 5 veces más rápido que el secuencial. Ya que el tiempo de ejecución necesario para VNS_GPU es mucho menor que el de VNS_CPU, se decide llevar a cabo un últi-

mo experimento para comprobar si el algoritmo VNS_GPU es capaz de generar mejores soluciones si aumentamos su tiempo de ejecución. Para ello, se establece el parámetro $k_{max} = 0.50$, en lugar del 0.20 original. La Tabla 3 muestra una comparativa de VNS_CPU frente a VNS_GPU sobre el conjunto total de instancias, estableciendo el parámetro k_{max} de VNS_GPU en 0.5 y manteniendo el de VNS_CPU en 0.2.

Algoritmo	F.O.	Tiempo (seg.)	Desv. (%)	#Mejores
VNS_CPU	3568332.9	280.6	0.7	53
VNS_GPU	3594570.8	207.6	0.5	40

Tabla 3. Comparativa del algoritmo secuencial VNS_CPU con $k = 0.2$ frente al paralelo VNS_GPU con $k = 0.5$

Como se puede observar en la Tabla 3, con esta modificación de k_{max} el algoritmo VNS_GPU sigue necesitando menos tiempo de ejecución, siendo 1.35 veces más rápido que el algoritmo secuencial. Además, la desviación estándar de la versión paralela ha disminuido hasta casi la mitad de la versión secuencial, obteniendo también un mayor número de mejores soluciones.

4. Conclusiones

En este artículo se presenta una comparativa entre un algoritmo secuencial y uno paralelo basados en la metodología *Variable Neighborhood Search*. Se ha utilizado el problema de la maximización del corte en grafos para ilustrar el comportamiento de ambos algoritmos. En concreto se propone un método *shake* así como dos búsquedas locales (una secuencial y otra paralela). La comparativa se ha llevado a cabo mediante la ejecución de ambos algoritmos sobre un total de 88 instancias ampliamente utilizadas en el estado del arte, divididas en tres subconjuntos diferentes. Los resultados experimentales muestran como el algoritmo paralelo obtiene soluciones ligeramente mejores que las obtenidas por el algoritmo secuencial. Además, la comparativa del tiempo de ejecución ilustra como el algoritmo paralelo necesita hasta 5 veces menos tiempo que el algoritmo secuencial para obtener la solución.

Agradecimientos

Este trabajo ha sido parcialmente financiado por la Comunidad de Madrid a través del proyecto con referencia S2009/TIC-1542 y por el Ministerio de Educación y Ciencia a través de los proyectos con referencias TIN2012-35632 y TIN2011-28151.

Referencias

1. Barr R.S., Hickman B.L. Reporting Computational Experiments with Parallel Algorithms: Issues, Measures, and Experts Opinions. ORSA Journal on Computing, 5(1):2–18 (1993)
2. Burer S., Monteiro R.D.C. Rank-two relaxation heuristics for max-cut and other binary quadratic programs. SIAM J. Optim, 12:503–521 (2001)
3. Crainic, T.G., Gendreau M., Hansen P., Mladenović N. Cooperative Variable Neighborhood Search for the p-Median Journal of Heuristics, 10:293–314 (2010)
4. Chang K.C., Du D.-Z. Efficient algorithms for layer asignment problems. IEEE Trans. Comput.-Aided Design, 6:67–78 (1987)
5. Chen R., Kajitani Y., Chan S. A graph-theoretic via minimization algorithm for two layer printed circuit boards. IEEE Trans. Circuits System, 30:284–299 (1983)
6. Festa P., Pardalos P.M., Resende M.G.C., Ribeiro C.C. Randomized heuristics for the max-cut problem. Optim. Methods Software, 7:1033–1058 (2002)
7. García-López F., Melián-Batista B., Moreno-Pérez J.A., Moreno-Vega M. The Parallel Variable Neighborhood Search for the p-Median Problem. Journal of Heuristics, 8:375–388 (2002)
8. Goemans M.X., Williamson D.P. Improved approximation algorithms for the max-cut and satisfability problems using semidefinite programming J. ACM, 42:1115–1145 (1995)
9. Hansen P., Mladenović N., Moreno J.A. Variable Neighborhood Search: methods and applications. Annals of Operations Research, 175(1):367–407 (2010)
10. Karp R.M. Reducibility among combinatorial problems. R.Miller, J.Tatcher, eds. Complexity of Computer Computations. Plenum Press, New York. 85–103 (1972)
11. Haglin D.J., Venkatesen M. Approximation and intractability results for the maximum cut problem and its variants. IEE Trans. Comput., 40:110–113 (1991)
12. Helmberg C., Rendl F. A spectral bundle method for semidefinite programming. SIAM J. Optim. 10:673–696
13. Homer S., Peinado M. Design and performance of parallel and distributed approximation algorithms for max cut. J. Parallel Distributed Comput., 46:48–61
14. Martí R., Duarte A., Laguna M. Advanced Scatter Search for the Max-Cut Problem INFORMS Journal on Computing, 21(1):26–38
15. Mladenović N., Hansen P. Variable Neighborhood Search Computers & Operations Research, 24(11):1097–1100
16. Montemayor, A.S., Duarte., A., Pantrigo, J.J., Cabido R.: High-Performance VNS for the Max-Cut Problem using Commodity Graphics Hardware 18th Mini EURO Conference on VNS, Tenerife, Spain (2005)
17. Poljak S., Tuza Z. A polynomial algorithm for constructing a large bipartite subgraph, with an application to a satisfiability problem. Canadian J.Math, 34:519–524 (1982)
18. Sahni S., Gonzales T. P-complete approximation problem. J. ACM 46:48–61