Encontrando grafos bipartitos completos mediante Búsqueda de Vecindad Variable

Juan David Quintana
Dept. Computer Sciences
Universidad Rey Juan Carlos
Madrid, España
juandavid.quintana@urjc.es

Jesús Sánchez-Oro
Dept. Computer Sciences
Universidad Rey Juan Carlos
Madrid, España
jesus.sanchezoro@ujrc.es

Abraham Duarte
Dept. Computer Sciences
Universidad Rey Juan Carlos
Madrid, España
abraham.duarte@urjc.es

***Resumen*—El Problema del Máximo Biclique Balanceado, o *Maximum Balanced Biclique Problem* (MBBP), consiste en encontrar el biclique de tamaño máximo inducido por un grafo bipartito no completo. Se tiene la restricción adicional de que tanto el tamaño del grafo de partida como el biclique inducido tienen el mismo número de vértices en cada una de sus capas, por lo tanto se dice que son balanceados. Algunas de sus aplicaciones se dan en el diseño de circuitos en integración a gran escala (VLSI) o el diseño de sistemas nanoelectrónicos, entre otras. Se ha demostrado que este problema de optimización combinatoria es *NP*-Duro. En la literatura se han propuesto diversas heurísticas para solucionarlo, generalmente basadas en la eliminación de vértices, y más recientemente se ha propuesto un algoritmo evolutivo que tiene los mejores resultados en la literatura. En este trabajo se propone el uso de la metodología RVNS para abordar este problema. Esta elección se justifica en la díficultad para diseñar una búsqueda local efectiva para este problema.**

***Index Terms*—biclique, Reduced VNS, bipartito.**

I. Introducción

Sea $G(L, R, E)$ un grafo bipartito balanceado en el que L y R son dos conjuntos (o capas) de vértices con la misma cardinalidad (i.e., $|L| = |R| = n$) y E es el conjunto de aristas. Al ser un grafo bipartito, $L \cap R = \emptyset$, una arista solo puede conectar un vértice $v \in L$ con otro $u \in R$, i.e., $\forall (v, u) \in E\ v \in L \wedge u \in R$. Además, definimos un biclique $B(L'R', E')$ como el grafo inducido por G, en el que $L' \subset L$, $R' \subset R$, de forma que todo vértice $v \in L'$ está conectado a todo vértice $u \in R'$. Dicho de otra forma, B es un grafo bipartito completo, también llamado biclique.

Dado un grafo bipartito balanceado $G(L, R, E)$, este trabajo se centra en resolver el Problema del Máximo Biclique Balanceado, más conocido como *Maximum Balanced Biclique Problem* (MBBP) en inglés, que consiste en identificar un biclique balanceado $B^\star(L', R', E')$ con el mayor número de vértices por capa. En otras palabras, el objetivo del MBBP es maximizar la cardinalidad de los conjuntos L' y R'.

La Figura 1 presenta un ejemplo de grafo bipartito con 8 vértices, 13 aristas, y dos posibles soluciones para el MBBP. La figura 1(b) muestra una solución $B_1(L_1, R_1, E_1)$ con dos vértices en cada capa. En este caso, $L_1 = \{\texttt{A}, \texttt{B}\}$, y $R_1 = \{\texttt{F}, \texttt{G}\}$. Las aristas que pertenecen al biclique inducido se representan con una línea continua, mientras que aquellas con un extremo fuera de la solución se representan con una línea discontinua. Obsérvese que no es posible insertar nuevos vértices en la solución, ya que en dicho caso el grafo bipartito inducido resultante no sería un biclique balanceado. Por ejemplo, no es posible añadir los vértices `E` o `H` porque no son adyacentes a los vértices `B` y `A`, respectivamente. Además, no es posible añadir vértices en la capa L_1 ya que el biclique inducido no sería balanceado (i.e., $|L_1| \neq |R_1|$).

La figura 1(c) muestra una solución $B_2(L_2, R_2, E_2)$ de mejor calidad, ya que tiene 3 vértices en cada capa. En concreto, $L_2 = \{\texttt{B}, \texttt{C}, \texttt{D}\}$, y $R_2 = \{\texttt{F}, \texttt{G}, \texttt{H}\}$. De nuevo, no es posible añadir más vértices sin violar la restricción de tener un biclique balanceado.

Se ha demostrado que este problema es $\mathcal{NP}$-Duro en varios trabajos previos [1]–[3]. Algunos resultados teóricos proponen cotas para el tamaño máximo que la solución óptima puede alcanzar [4], y se demostrado difícil de aproximar dentro de un determinado factor [5].

Los grafos biclique han demostrado ser de utilidad en varias aplicaciones prácticas, la mayoría de ellas en el campo de la biología: agrupación de datos de microarrays [6]–[8], optimización de la reconstrucción del arbol filogenético [9], entre otras [10]–[12]. En particular, el MBBP tiene aplicaciones adicionales en diversos campos: diseño eficiente de circuitos en integración a gran escala (VLSI) [13], o el diseño de nuevos sistemas nanoelectrónicos [14]–[16], entre otras.

A pesar de las aplicaciones prácticas del MBBP, no se han propuesto muchos algoritmos eficientes para solucionarlo, principalmente debido a la dificultad del problema. Sin embargo, si del MBBP eliminamos la restricción de que la solución deba ser balanceada, entonces el problema resultante sería resoluble en tiempo polinómico [17], pero la mayoría de las soluciones serían desbalanceadas, haciendo que los resultados no sean adaptables al problema que estamos considerando.

La mayoría de los planteamientos previos siguen un enfoque destructivo en el que la solución inicial contiene todos los vértices y la heurística elimina iterativamente vértices en L' y R' hasta que la solución resultante es factible. Estos algoritmos se diferencian entre sí principalmente por el criterio

Este trabajo ha sido financiado parcialmente por el Ministerio de Economía y Competitividad, ref. TIN2015-65460-C2-2-P.

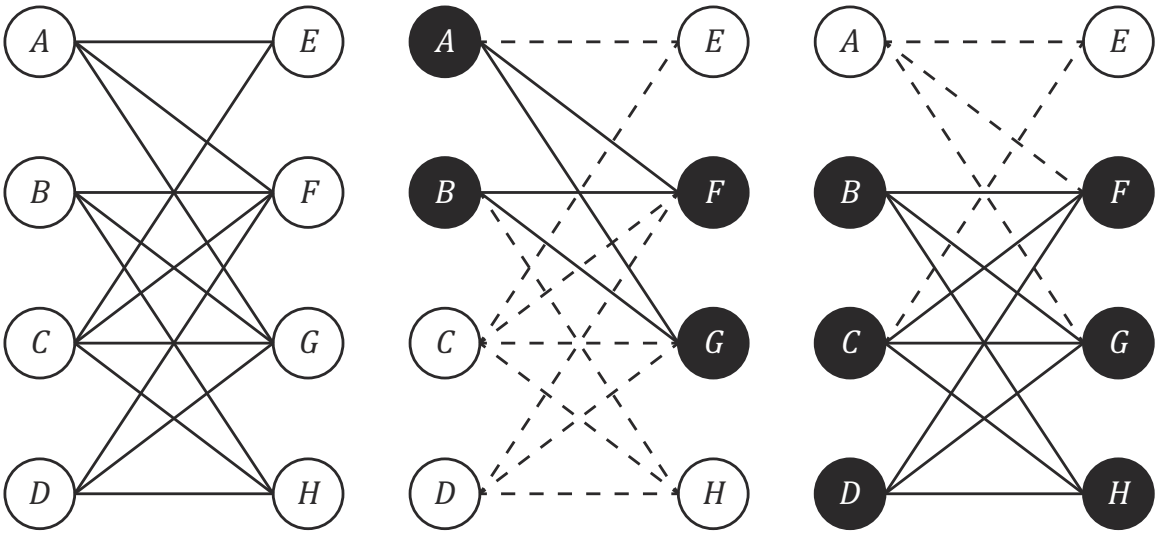

Figura 1: 1(a) Grafo bipartito con 8 vértices y 13 aristas, 1(b) una solución factible con 2 vértices en cada capa (A y B en L', F y G en R') , y 1(c) una solución distinta con 3 vértices en cada capa (B, C y D en L', F, G, y H en R')

utilizado para elegir los vértices a eliminar. En particular, [14] selecciona los vértices de mayor grado, mientras que [15] elimina los de mayor número de vértices de mínimo grado en la otra capa. Además, algunos algoritmos han intentado combinar ambos criterios [18], [19]. El mejor algoritmo encontrado en la literatura consiste en un algoritmo evolutivo [20] que propone un nuevo operador de mutación así como una nueva busqueda local para mejorar la calidad de las soluciones generadas.

El resto de este artículo está organizado de la siguiente forma: la Sección II describe la propuesta algorítmica para el MBBP; la Sección III presenta los experimentos realizados para evaluar la calidad de la propuesta algortimica; y la Sección IV plantea las conclusiones de esta investigación.

II. Reduced Variable Neighborhood Search

La Búsqueda de Vecindad Variable (VNS por sus siglas en inglés *Variable Neighborhood Search*) [21] es una metaheuristica basada en cambios sistemáticos de vecindad. Al ser una metaheurística, no garantiza la optimalidad de las soluciones obtenidas, pero se centra en obtener soluciones de alta calidad en un tiempo razonable de cómputo. La constante evolución de VNS ha dado lugar a diversas variantes, entre la que podemos destacar *Basic VNS*, *Reduced VNS (RVNS)*, *Variable Neighborhood Descent (VND)* , *General VNS*, *Skewed VNS*, *Variable Neighborhood Decomposition Search*, entre otras.

La mayoría de las variantes se diferencian en la forma de explorar las vecindades establecidas. En particular, *Variable Neighborhood Decomposition Search* realiza una exploración totalmente determinista del espacio de soluciones, mientras que la exploración realizada por RVNS es totalmente estocástica. Algunas variantes combinan cambios de vecindad tanto deterministas como estocásticas (e.g., *Basic VNS*, *General VNS*).

Como se menciona en trabajos previos [19], [20], diseñar una búsqueda local para el MBBP es una tarea difícil principalmente por la complejidad de mantener una solución factible (i.e. un biclique balanceado) tras eliminar o añadir vértices en una solución previa. En otras palabras, el MBBP no es idóneo para proponer métodos de búsqueda local que permitan encontrar un óptimo local en relación a una solución determinada. Por lo tanto, este trabajo propone un algoritmo Reduced VNS, que se basa en la exploración aleatoria de las vecindades establecidas.

RVNS es una variante de VNS útil para instancias grandes en las que la búsqueda local es muy costosa en tiempo, o para aquellos problemas en los que no es fácil diseñar un método de búsqueda local.

Algoritmo 1 $RVNS(B, k_{\text{máx}}, t_{\text{máx}})$

```
1: repeat
2:     k ← 1
3:     while k ≤ k_máx do
4:         B' ← Shake(B, k)
5:         k ← NeighborhoodChange(B, B', k)
6:     end while
7: until CPUTime() ≤ t_máx
8: return B
```

El algoritmo requiere de dos parámetros de entrada: B, una solución factible de partida, que se puede generar aleatoriamente o mediante un procedimiento constructivo más elaborado; y $k_{\text{máx}}$, la vecindad máxima a explorar.

Cada iteración de RVNS empieza desde la vecindad inicial (paso 2). A continuación, el método explora cada una de las vecindades establecidas (pasos 3-6) de la siguiente forma. En

primer lugar, el método genera una solución al azar B' en la vecindad k de la solución actual B con el método *Shake* (paso 4). Luego el método *NeighborhoodChange* (paso 5) se encarga de seleccionar la siguiente vecindad a explorar. En particular, si la nueva solución B' es mejor que la solución de partida B, entonces se actualiza ($B \leftarrow B'$), y reinicia la búsqueda desde la vecindad inicial ($k \leftarrow 1$). En caso contrario, la búsqueda continúa con la siguiente vecindad ($k \leftarrow k+1$). Una iteración de RVNS termina cuando se ha alcanzado la vecindad máxima establecida $k_{\text{máx}}$. Es importante mencionar que la vecindad máxima utilizada en RVNS es generalmente pequeña debido a la naturaleza aleatoria del método *Shake*, ya que un valor mayor para $k_{\text{máx}}$ produciría resultados equivalentes a reiniciar la búsqueda desde una nueva solución inicial. El método RVNS se ejecuta hasta que se alcanza un tiempo límite de cómputo $t_{\text{máx}}$.

II-A. Procedimiento constructivo

La solución inicial para RVNS puede ser generada al azar o con un procedimiento constructivo más elaborado. Este trabajo propone un procedimiento constructivo basado en *Greedy Randomized Adaptive Search Procedure* (GRASP) [22]. Esta metodología considera una función voraz que evalúa la importancia de insertar cada vértice en la solución a construir. En el Algoritmo 2 se presenta el pseudocódigo del procedimiento constructivo propuesto.

Algoritmo 2 *Construct*$(G = (L, R, E), \alpha)$

1: **function** UpdateLayer(CL_1, CL_2, α)
2: $g_{\text{mín}} \leftarrow \text{mín}_{v \in CL_1}\, g(v)$
3: $g_{\text{máx}} \leftarrow \text{máx}_{v \in CL_1}\, g(v)$
4: $\mu \leftarrow g_{\text{mín}} + \alpha \cdot (g_{\text{máx}} - g_{\text{mín}})$
5: $RCL \leftarrow \{v \in CL_1 \ ; \ g(v) \leq \mu\}$
6: $v \leftarrow Random(RCL)$
7: $CL_1 \leftarrow CL_1 \setminus \{v\}$
8: $CL_2 \leftarrow CL_2 \setminus \{u \in CL_2 \ : \ (v,u) \notin E\}$
9: **return** v
10: **end function**
11: $v \leftarrow Random(L)$
12: $L' \leftarrow \{v\}$
13: $CL_L \leftarrow L \setminus \{v\}$
14: $CL_R \leftarrow \{u \in R \ : \ (v,u) \in E\}$
15: **while** $CL_L \neq \emptyset$ **and** $CL_R \neq \emptyset$ **do**
16: $v_r \leftarrow$ UpdateLayer(CL_R, CL_L, α)
17: $R' \leftarrow R \cup \{v_r\}$
18: $v_l \leftarrow$ UpdateLayer(CL_L, CL_R, α)
19: $L' \leftarrow L \cup \{v_l\}$
20: **end while**
21: **return** $B = (L', R', \{(v,u)\ v \in L' \wedge u \in R'\})$

El método empieza por seleccionar un vértice al azar de la capa L (paso 11), insertarlo la capa correspondiente L' de la solución (paso 12). Entonces, se crean dos listas de candidatos (CL), una por cada capa del grafo (paso 13-14). Obsérvese que ambas listas de candidatos solo contienen aquellos vértices de cada capa que pueden ser seleccionados manteniendo la solución factible (i.e., la solución es un grafo bipartito completo). Por lo tanto, en este paso inicial, CL_L contiene todos los vértices de L exceptuando al vértice seleccionado v. Por otra parte, CL_R contiene todos los vértices adyacentes a v en R, ya que de otra forma la solución no sería un grafo bipartito completo. A continuación, el método añade un vértice en la capa R' y luego en L' iterativamente, mientras que queden candidatos en ambas capas (paso 16-19).

La selección del vértice siguiente se describe en la función *UpdateLayer* que requiere de tres parámetros: la lista de candidatos desde la que se debe elegir el vértice, CL_1, la lista de candidatos de la otra capa, CL_2, y el parámetro α que determina el grado de voracidad / aleatoriedad de la selección. Una función voraz g que evalúa la calidad de un vértice candidato v debe ser definida. En este trabajo, proponemos como función el número de vértices adyacentes en la lista de candidatos de la capa opuesta. Formalmente,

$$g(v, CL) \leftarrow \{u \in CL \ : \ (v,u) \in E\}$$

El primer paso para seleccionar el vértice siguiente consiste en obtener los valores mínimo ($g_{\text{mín}}$) y máximo ($g_{\text{máx}}$) para la función voraz (paso 2-3). A continuación, se construye (paso 5) la lista de candidatos restringida (*RCL*) con aquellos vértices cuyo valor de la función objetivo sea mayor o igual que el umbral hallado previamente μ (paso 4). Los valores para el parámetro α están en el rango 0–1, dónde $\alpha = 0$ implica que el método es totalmente aleatorio, y $\alpha = 1$ transforma el algoritmo en completamente voraz. El siguiente vértice se selecciona al azar de *RCL* (paso 6), y luego se actualiza cada lista de candidatos. En particular, CL_1 se actualiza mediante quitar el vértice seleccionado de sí mismo, mientras CL_2 se actualiza quitando aquellos vértices que no son adyacentes a v, debido a que no pueden ser elegidos en iteraciones futuras sin que la solución se vuelva infactible.

II-B. Shake

El método *Shake* es una etapa dentro de la metodología VNS diseñada para escapar de óptimos locales encontrados durante la fase de búsqueda. Consiste en perturbar aleatoriamente una solución con el objetivo de explorar regiones más amplias en el espacio de búsqueda. Esta fase de la metodología VNS se centra en diversificar la búsqueda.

Dada una vecindad k, el método *Shake* propuesto en este trabajo elimina k elementos al azar de cada capa. La solución resultante es factible pero el valor de función objetivo siempre es menor, ya que se reduce el número de vértices seleccionados.

Teniendo en cuenta las restricciones del problema, si un vértice v se añade a la solución, entonces todos los vértices en la capa opuesta que no sean adyacentes a v no pueden ser añadidos en iteraciones futuras, ya que en dicho caso la solución obtenida no sería un biclique. Sin embargo, quitar algunos vértices con el método *Shake* puede permitir eventualmente que nuevos vértices sean añadidos a la solución

(i.e., aquellos que no eran adyacentes a ninguno de los vértices eliminados).

Por lo tanto, proponemos una etapa de reconstrucción que se ejecuta después de cada *Shake*. En concreto, la fase de reconstrucción intenta añadir nuevos vértices a la solución, aquellos que no eran candidatos factibles antes de ejecutar el método *Shake*.

La etapa de reconstrucción siempre mejora o, al menos, mantiene la calidad de la solución obtenida tras la llamada al método *Shake*. Obsérvese que la solución reconstruida solo supera a la inicial si y solo si la etapa de reconstrucción es capaz de insertar más de k vértices en cada capa.

La naturaleza aleatoria de este procedimiento hace que sea difícil obtener mejoras en la calidad de la solución. Para mitigar este efecto se proponen cuatro variantes del método *Shake*. Estas se diferencian en cómo se realizan las fases de destrucción y reconstrucción de la solución, cada fase puede ser respectivamente aleatoria (R) o voraz (G), con lo que obtenemos cuatro variantes: RR, RG, GR, GG. La Tabla I muestra el comportamiento voraz o aleatorio de cada una de las variantes propuestas. Por ejemplo, la variante de *Shake* GR realiza una destrucción voraz con una reconstrucción aleatoria.

Variante	Destrucción	Reconstrucción
RR	Aleatoria	Aleatoria
RG	Aleatoria	Voraz
GR	Voraz	Aleatoria
GG	Voraz	Voraz

Tabla I: Enumeración de las cuatro variantes de *Shake* propuestas.

En la fase de reconstrucción se utiliza el mismo método constructivo que genera la solución inicial, pero modificado para ser totalmente aleatorio ($\alpha = 0$) o totalmente voraz ($\alpha = 1$). En la fase de destrucción se plantea un algoritmo, con una estructura similar a la del método constructivo, que elimina vértices de una solución factible. Para que sea una destrucción voraz es necesario definir una nueva función heurística g' que puntúe los vértices a eliminar. El valor de esta función heurística para cada vértice $v \in S$ se calculará por el número de vertices en la otra capa con los que no está conectados.

III. Resultados experimentales

En esta sección se presentan dos grupos de experimentos: los experimentos preliminares, realizados para ajustar los parámetros del algoritmo propuesto; y el experimento final, diseñado para evaluar la calidad de nuestra propuesta algorítmica y compararla con el algoritmo previo de la literatura. Todos los algoritmos fueron implementados en Java y se ejecutaron en un sistema con una CPU Intel i7 (7660U) y 8 GB de RAM, en la máquina virtual de Java 8.

Se ha utilizado el mismo conjunto de datos presentado en [20], facilitado por dichos autores, que consiste en 30 grafos bipartitos balanceados con tamaños $n = \{250, 500\}$, donde n representa el número de vértices en cada capa; y con diferentes probabilidades $p = \{85\,\%, 90\,\%, 95\,\%\}$ de que una arista exista en el grafo. Se trata de grafos densamente poblados en los que falta un porcentaje bajo de aristas para que lleguen a ser completos. Este conjunto de datos se utiliza para comparar el rendimiento de nuestro algoritmo con el estado del arte. A parte de las instancias utilizadas para presentar sus resultados, los autores de [20] también facilitan otras instancias generadas con los mismos parámetros, no utilizadas en su artículo, y que en este trabajo se utilizan para ajustar los parámetros del algoritmo.

En estos experimentos presentamos los siguientes datos: el tamaño promedio del biclique balanceado más grande obtenido para el conjunto de datos, Tamaño; el tiempo promedio de ejecución por instancia, Tiempo (s); el porcentaje de desviación promedio respecto de la mejor solución obtenido por instancia, %Desv.; y el número de veces que se encuentra la mejor solución para cada instancia, #Mejores.

III-A. Experimentos preliminares

Los experimentos presentados a continuación fueron diseñados para elegir la mejor variante para el algoritmo propuesto. Se utilizó un grupo reducido de 6 instancias representativas, una por cada combinación posible de parámetros, seleccionadas a partir de las excluidas del experimento final para no sobreajustar el algoritmo. Estas instancias forman parte de las facilitadas por [20] y han sido generadas al azar con los mismos parámetros n (tamaño) y p (probabilidad) que aquellas utilizadas en la comparación de resultados. En el primer experimento se comprueba cuál es el mejor valor de α para el procedimiento constructivo, en el segundo se comparan cuatro variantes del procedimiento *Shake*, y finalmente se examina cuál es el valor más conveniente para la vecindad máxima a explorar. Todos los experimentos se han ejecutado iterativamente hasta alcanzar el limite de tiempo establecido, un número de segundos igual al tamaño n de la instancia.

α	Tamaño	Tiempo (s)	%Desv.	#Mejores
0.25	46.33	375.00	11.57	0
0.50	48.33	375.00	7.96	0
0.75	52.00	375.00	0.96	2
rnd	**52.50**	**375.00**	**0.00**	**6**

Tabla II: Comparación del algoritmo constructivo con distintos valores del parámetro α.

En el primer experimento ejecutamos iterativamente únicamente el procedimiento constructivo hasta que se alcanza el límite de tiempo para cada instancia y se guarda la mejor solución alcanzada en dicho tiempo. Se consideran los siguientes valores posibles para el parámetro alfa $\alpha = \{0.25, 0.50, 0.75, random\}$. En particular, el valor "random" significa que en cada iteración se elige un alfa distinto de forma aleatoria. Los resultados en la Tabla II muestran que los mejores resultados se obtienen cuando se elige aleatoriamente el parámetro alfa en cada iteración. En este caso, se obtiene un biclique balanceado de tamaño 52.50 en promedio y, además,

alcanza la mejor solución hallada en este experimento en todas las 6 instancias. Por consiguiente, tanto en los experimentos sucesivos como en el final, se utilizará este valor de alfa para el constructivo inicial que forma parte de nuestra propuesta de *RVNS*.

Shake	Tamaño	Tiempo (s)	%Desv.	#Mejores
RR	48.83	375.00	10.21	0
RG	**54.33**	375.00	0.00	6
GR	49.50	375.00	8.95	0
GG	53.17	375.00	2.37	1

Tabla III: Comparación del algoritmo *RVNS* con las cuatro variantes del método *Shake*. Se ha fijado la vecindad máxima en $K_{max} = 50\,\%$.

El siguiente experimento está diseñado para elegir la mejor variante del método *Shake*. Se ha fijado la vecindad máxima del algoritmo RVNS en $K_{max} = 50\,\%$ y se comparan las cuatro variantes propuestas para el método *Shake*. En este experimento se observa en la Tabla III que la variante RG, es decir, destrucción aleatoria con reconstrucción voraz, alcanza los mejores resultados. Esta obtiene un biclique balanceado con un tamaño de 54.33 vértices en promedio y, además, también obtiene las 6 mejores soluciones de este experimento.

K_{max}	Tamaño	Tiempo (s)	%Desv.	#Mejores
10 %	53.17	375.00	2.37	1
20 %	53.33	375.00	1.85	2
30 %	53.83	375.00	0.98	3
40 %	53.83	375.00	0.87	3
50 %	**54.33**	375.00	0.00	6

Tabla IV: Comparación del algoritmo *RVNS* para diferentes valores de la vecindad máxima K_{max}.

En el último experimento preliminar queremos evaluar la influencia que tiene el tamaño máximo de la vecindad k_{max} sobre el algoritmo propuesto. Se utiliza la mejor configuración hallada en los experimentos anteriores, es decir, una construcción incial con un alfa aleatorio en cada iteración y el método *Shake* que realiza primero una destrucción aleatoria y luego una reconstrucción voraz (RG); y se ejecuta el algoritmo con diferentes tamaños para la vecindad máxima $K_{max} = \{10\,\%, 20\,\%, 30\,\%, 40\,\%, 50\,\%\}$. En este experimento se puede ver que, en general, expandir el tamaño de la vecindad permite que el algoritmo alcance mejores soluciones, disminuyendo en consecuencia la desviación. Se toma como mejor vecindad $K_{max} = 50\,\%$ con la que se han obtenido los mejores resultados de este experimento.

III-B. Experimento final

En el experimento final se presenta una comparación con el algoritmo memético [20] encontrado en la literatura previa. La comparación se realiza utilizando el mismo conjunto de 30 instancias que se han utilizado en dicho trabajo. El algoritmo *RVNS* se ejecuta por iteraciones hasta que se llega a un tiempo límite, en este caso es un tiempo en segundos igual al triple del tamaño (n) de las instancias. En la Tabla V se puede ver que nuestros resultados son muy cercanos al algoritmo previo respecto a la calidad de las soluciones obtenidas, está a menos de un vértice de distancia en promedio, pero tiene un tiempo de ejecución que se aproxima a la mitad del tiempo requerido por el algoritmo memético.

	Tamaño	Tiempo (s)	%Desv.	#Mejores
EA/SM	55.10	2075.11	0.04	29
RVNS	54.33	1125.00	1.71	10

Tabla V: Comparación del RVNS con el algoritmo evolutivo EA/SM encontrado en la literatura.

IV. Conclusiones

En este trabajo se ha propuesto el uso de la metodología RVNS para buscar soluciones eficientes para el MBBP. Se ha constatado que en este problema es difícil plantear una búsqueda local efectiva y debido a ello se ha elegido una variante de VNS que explora aleatoriamente una vecindad predefinida. Se ha conseguido plantear un algoritmo competitivo en calidad respecto al actual estado del arte que utiliza aproximadamente la mitad del tiempo que este, por lo que todavía es un enfoque prometedor que permite explorar en un futuro otras técnicas complementarias para mejorar estos resultados.

Referencias

[1] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York: Freeman, 1979.

[2] D. S. Johnson, "The np-completeness column: An ongoing guide." *J. Algorithms*, vol. 13, no. 3, pp. 502–524, 1992.

[3] N. Alon, R. A. Duke, H. Lefmann, V. Rödl, and R. Yuster, "The algorithmic aspects of the regularity lemma," *J. Algorithms*, vol. 16, no. 1, pp. 80–109, 1994.

[4] M. Dawande, P. Keskinocak, J. M. Swaminathan, and S. Tayur, "On bipartite and multipartite clique problems," *Journal of Algorithms*, vol. 41, no. 2, pp. 388 – 403, 2001.

[5] U. Feige and S. Kogan, "Hardness of approximation of the balanced complete bipartite subgraph problem," Tech. Rep., 2004.

[6] Y. Cheng and G. M. Church, "Biclustering of expression data," in *Proc. of the 8th ISMB*. AAAI Press, 2000, pp. 93–103.

[7] A. Tanay, R. Sharan, and R. Shamir, "Discovering statistically significant biclusters in gene expression data." in *ISMB*, 2002, pp. 136–144.

[8] H. Wang, W. W. 0010, J. Yang, and P. S. Yu, "Clustering by pattern similarity in large data sets." in *SIGMOD Conference*, M. J. Franklin, B. Moon, and A. Ailamaki, Eds. ACM, 2002, pp. 394–405.

[9] M. J. Sanderson, A. C. Driskell, R. H. Ree, O. Eulenstein, and S. Langley, "Obtaining maximal concatenated phylogenetic data sets from large sequence databases," *Molecular Biology and Evolution*, vol. 20, no. 7, pp. 1036–1042, 2003.

[10] E. J. Chesler and M. A. Langston, "Combinatorial genetic regulatory network analysis tools for high throughput transcriptomic data." in *Systems Biology and Regulatory Genomics*, ser. Lecture Notes in Computer Science, E. Eskin, T. Ideker, B. J. Raphael, and C. T. Workman, Eds., vol. 4023. Springer, 2005, pp. 150–165.

[11] E. J. Baker, J. J. Jay, V. M. Philip, Y. Zhang, Z. Li, R. Kirova, M. A. Langston, and E. J. Chesler, "Ontological discovery environment: A system for integrating gene–phenotype associations," *Genomics*, vol. 94, no. 6, pp. 377 – 387, 2009.

[12] R. A. Mushlin, A. Kershenbaum, S. T. Gallagher, and T. R. Rebbeck, "A graph-theoretical approach for pattern discovery in epidemiological research." *IBM Systems Journal*, vol. 46, no. 1, pp. 135–150, 2007.

[13] S. S. Ravi and E. L. Lloyd, "The complexity of near-optimal programmable logic array folding." *SIAM J. Comput.*, vol. 17, no. 4, pp. 696–710, 1988.
[14] M. B. Tahoori, "Application-independent defect tolerance of reconfigurable nanoarchitectures," *JETC*, vol. 2, no. 3, pp. 197–218, 2006.
[15] A. A. Al-Yamani, S. Ramsundar, and D. K. Pradhan, "A defect tolerance scheme for nanotechnology circuits." *IEEE Trans. on Circuits and Systems*, vol. 54-I, no. 11, pp. 2402–2409, 2007.
[16] M. B. Tahoori, "Low-overhead defect tolerance in crossbar nanoarchitectures." *JETC*, vol. 5, no. 2, 2009.
[17] M. Yannakakis, "Node-deletion problems on bipartite graphs." *SIAM J. Comput.*, vol. 10, no. 2, pp. 310–327, 1981.
[18] B. Yuan and B. Li, "A low time complexity defect-tolerance algorithm for nanoelectronic crossbar," in *International Conference on Information Science and Technology*, 2011, pp. 143–148.
[19] ——, "A fast extraction algorithm for defect-free subcrossbar in nanoelectronic crossbar," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 10, no. 3, p. 25, 2014.
[20] B. Yuan, B. Li, H. Chen, and X. Yao, "A new evolutionary algorithm with structure mutation for the maximum balanced biclique problem," *IEEE Trans. Cybernetics*, vol. 45, no. 5, pp. 1040–1053, 2015.
[21] P. Hansen and N. Mladenović, *Variable Neighborhood Search*. Boston, MA: Springer US, 2014, pp. 313–337.
[22] T. A. Feo and M. G. Resende, "Greedy randomized adaptive search procedures," *Journal of global optimization*, vol. 6, no. 2, pp. 109–133, 1995.