ALGORITMOS DE APROXIMACIÓN
MINIMUM VERTEX COVER

Mario Garrido

Universidad de Chile
Departamento de Ciencias de la Computación
Santiago, Chile

ABSTRACT

En el presente trabajo se estudia el desempeño de 3 algoritmos de aproximación para el problema de cobertura de vértices mínima y sus costos.

1. INTRODUCCIÓN

De los 3 algoritmos, uno de ellos no tiene un radio de aproximación acotado para ningún valor constante, por lo que constituye una aproximación arbitrariamente mala. Se pondrá a prueba, en base a grafos generados aleatoriamente y algunos grafos correspondientes a *benchmarks*, si las aproximaciones acotadas son una mejor opción en la práctica, y si se justifican los costos adicionales incurridos en cada refinamiento.

2. IMPLEMENTACIÓN

La implementación se desarrolló en el lenguaje *C++*, bajo el standard de *C++11*. La implementación se encuentra contenida en los archivos *experiments.cpp*, *mvc.cpp* y *mvc.h*.

2.1. Grafo

Esta sección describe aspectos relevantes sobre el contenido del archivo mvc.cpp

La clase *graph* simplifica la implementación de los algoritmos de aproximación. Para representar el grafo se utilizan listas de adyacencia. Entre sus métodos más relevantes se encuentran los mencionados a continuación.

add_edge y add_edge_safely: El primero permite agregar un arco, revisando primero si existe, representado como un elemento extra en la lista de adyacencia de ambos vértices involucrados. Pese a que ambas listas, y ambos grados, de los vértices crecen, no se consideran ambos arcos en el total, ya que el grafo es no dirigido. El segundo método es una versión *thread-safe*, la que se utiliza cuando se generan los grafos *random*.

build_heap: Construye un *heap* en que cada vértice está ordenado en base a su grado.

remove_neighbour_from_all: Remueve todos los arcos incidentes a un vértice u. Para esto, se reduce el grado de todos los vértices en la lista de adyacencia de u y se marca u como inactivo en la lista de vértices activos del grafo. También se encarga de mantener actualizado el *heap*, en caso de cambios en el grado de algún vértice. También existe la versión que elimina sin actualizar el *heap*, esto se utiliza en el primer algoritmo, que no necesita encontrar el nodo de mayor grado.

copy_state: Crea una copia del estado actual del grafo. Esta copia comprende la lista de vértices activos, el grado de cada vértice, en base a los vértices activos y el estado del *heap*.

restore_state: Devuelve el grafo al estado almacenado. Esta función se utiliza para restaurar el grafo entre los algoritmos de aproximación, ya que copiar o reconstruir el grafo es muy caro.

2.2. Algoritmos de aproximación

Esta sección describe aspectos relevantes sobre el contenido del archivo mvc.cpp

2-aproximación: En cada paso del *loop* escogemos un vértice u, activo, y un vecino v de u, también activo. Se agregan ambos vértices a la solución y se quitan del grafo los arcos incidentes a u y los arcos incidentes a v. Esto se repite hasta que no quedan arcos.

Heurística del grado mayor: En cada paso del *loop* se toma el vértice de mayor grado, para lo que se utiliza el *heap*, se agrega a la solución y se remueven todos los arcos incidentes a u del grafo. Esto se repite hasta que no quedan arcos.

2-aproximación mejorada: En cada paso del *loop* se escoge el vértice u de mayor grado y su vecino v de mayor grado, se agregan ambos a la solución y se quitan los arcos incidentes a u o a v. Esto se repite hasta que no quedan arcos.

Gracias a la clase grafo, la implementación de los algoritmos es bastante económica.

3. RESULTADOS EXPERIMENTALES

Los experimentos fueron realizados en un computador con procesador AMD Phenom II x4 955, corriendo a $\sim$ 3.2Ghz y con 26624 Megabytes de RAM, bajo el sistema operativo *Ubuntu* 14.04.

La función *generate_random_graph*, de *experiments.cpp*, permite generar grafos *random*, en que cada arco potencial se escoge con probabilidad p. Esto se hace de forma paralela, cuidando que cada *thread* tenga su propio generador, para evitar *dataraces* que puedan modificar la distribución.

De los grafos generados con este procedimiento se aprecia una estrecha relación entre n, el número total de vértices, p y el número de arcos. El número esperado de arcos está dado por $\frac{n(n-1)}{2}p$, de donde notamos que el número esperado de vecinos en la lista de adyacencia de cada vértice es $(n-1)p$. En la figura 1 se aprecia la dependencia lineal del número de arcos generado respecto a p. La figura 2 contrasta las curvas con n^2, lo que parece ser una cota.

Pese a que el costo amortizado de agregar elementos a la lista de adyacencia es $\mathcal{O}(1)$, la selección de elementos *random*, en particular de vecinos *random*, se encuentra ligada al número esperado de vecinos para cada vértice, por lo que, en el peor caso (la solución contiene todos los vértices posibles), tendremos, al menos, $\mathcal{O}(n^2)$ en tiempo, para la 2-aproximación. La operación para quitar los arcos correspondientes es $\mathcal{O}(m)$, con m la cantidad de vecinos del vértice. La figura 3 muestra tiempos de ejecución del algoritmo de 2-aproximación, y los contrasta con las curvas para n^2 y $n^2 \log n$, donde parece ser esta última la cota adecuada.

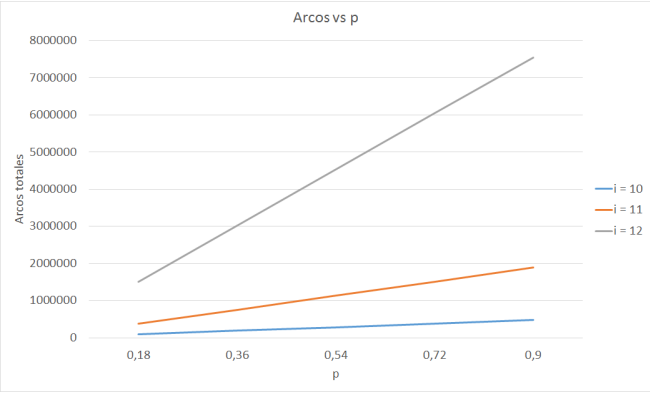

Fig. 1. Promedio de arcos vs p.

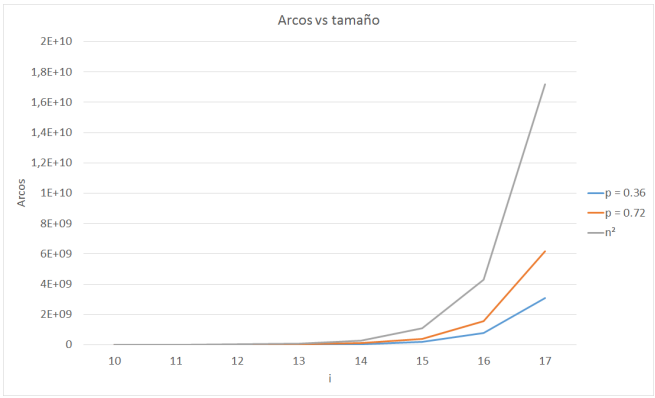

Fig. 2. Promedio de arcos vs tamaño.

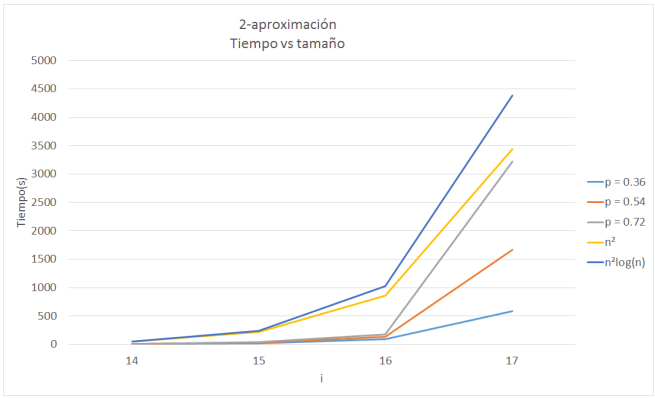

Fig. 3. Tiempo de 2-aproximación vs i.

En la figura 4, notamos que se repite la cota para el caso de la Heurística, lo cual no es extraño si observamos, por ejemplo, la figura 6, donde consta que los tiempos de los 3 algoritmos son similares. Para intentar entender esto, notamos que obtener el vértice de mayor grado es $\mathcal{O}(1)$, gracias al *heap*, y agregar a la lista de adyacencia tiene costo amortizado $\mathcal{O}(1)$, como se mencionó previamente, pero al remover los arcos incidentes al vértice de mayor grado es necesario restaurar el *heap* desde todos los vértices que representan a vecinos del máximo, esto es $\mathcal{O}(m \log n)$. En el peor caso, la solución contiene todos los vértices y $m \sim n$, por lo que es posible considerar que la Heurística sea $\mathcal{O}(n^2 \log n)$.

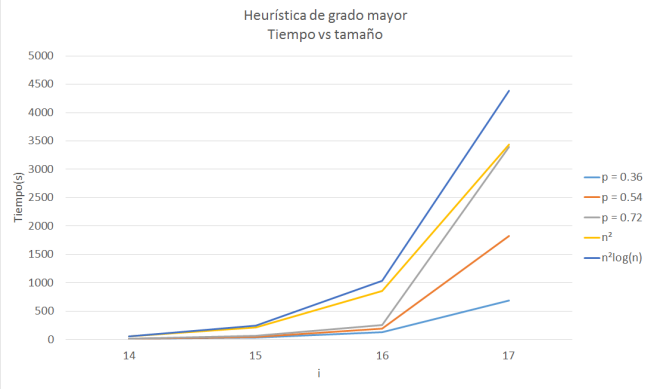

Fig. 4. Tiempo de Heurística de grado mayor vs i.

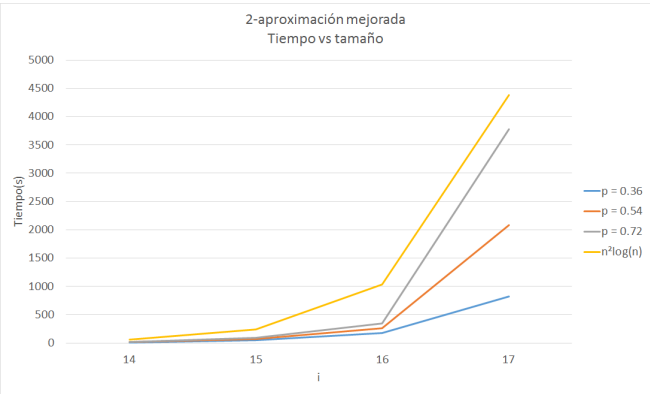

Fig. 5. Tiempo de 2-aproximación mejorada vs i.

La diferencia, en términos de costos, entre la 2-aproximación mejorada y la Heurística es la selección del vecino de mayor grado, que en la implementación actual es $\mathcal{O}(m)$. Sumar esto, sin embargo, no debería afectar el orden del algoritmo, lo que se ve reflejado en la figura 5.

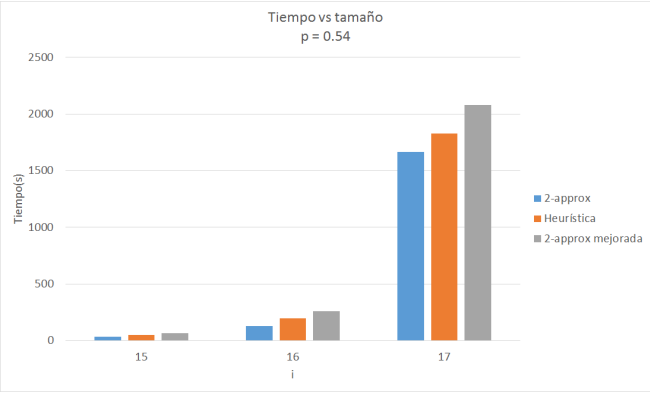

Fig. 6. Tiempo vs i, los 3 algoritmos.

Respecto a las soluciones que producen los algoritmos frente a los grafos *random*, estas son muy cercanas a la totalidad de los vértices. El valor más bajo de p inspeccionado (0.18) parece producir grafos demasiado densos como para que estos algoritmos puedan escoger soluciones considerablemente mejores que la trivial. En la figura 7 notamos que las mejores soluciones parecen venir de la Heurística, la cual pese a no ser una aproximación acotada, en ejemplos no tan patológicos como el de la demostración parece comportarse mucho mejor.

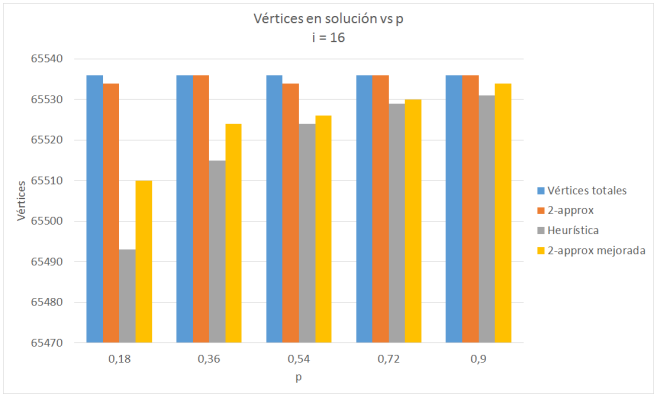

Fig. 7. Número de vértices en la solución vs p, i = 16.

Se experimentó, además, con los grafos planares *planar_embedding1000000.pg, planar_embedding5000000.pg, planar_embedding10000000.pg*, de 1000000, 5000000 y 10000000 de vértices, respectivamente. La densidad de estos grafos es mucho menor a las densidades alcanzadas por los grafos *random*, por lo que los 3 algoritmos son capaces de generar soluciones que ahorran una cantidad significativa de vértices (figura 8). Vale la pena notar que si un grafo es muy poco denso, entonces la probabilidad de necesitar más vértices aumenta, al igual que si existen muchos vértices, por lo que hay un *sweep spot* en el medio.

En la figura 9 se muestran los resultados para los grafos *C1000.9.clq, C2000.5.col, C2000.9.clq, C4000.5.col*, los cuales corresponden a grafos generados de forma *random* con 1000, 2000, 2000 y 4000 vértices y $p = 0{,}9$, $p = 0{,}5$, $p = 0{,}9$ y $p = 0{,}5$, respectivamente. No difieren, mayormente, de los grafos generados por *generate_random_graph* y dada la alta densidad que presentan los algoritmos otorgan soluciones muy cercanas a la trivial.

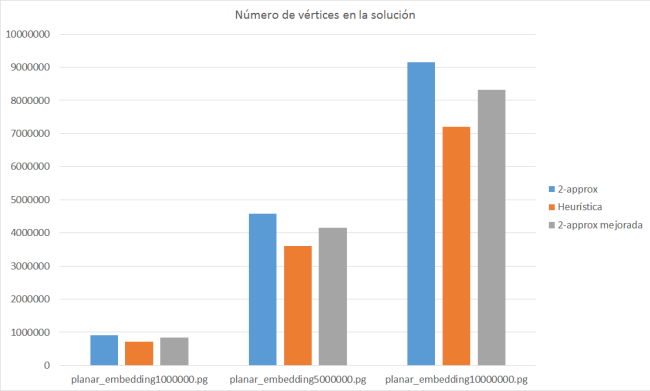

Fig. 8. Número de vértices en la solución.

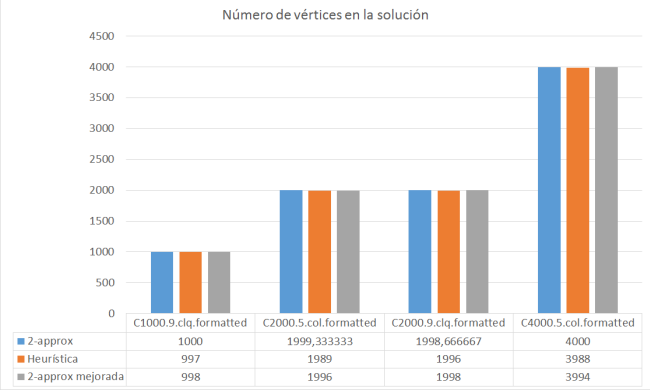

Fig. 9. Número de vértices en la solución.

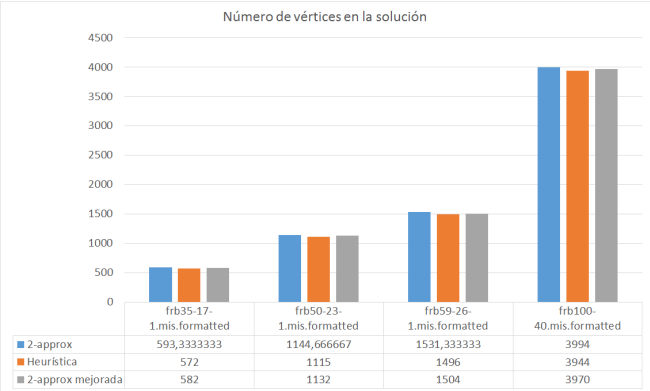

Fig. 10. Número de vértices en la solución.

4. CONCLUSIONES

Los algoritmos estudiados parecen tener utilidad limitada en ambientes teóricos y en casos patológicos, pero en el caso de grafos con alto número de vértices y baja densidad en las conexiones (como es el caso en varias aplicaciones prácticas) pueden resultar ser útiles para aproximar el problema de la cobertura de vértices mínima, considerando que la complejidad en tiempo de los algoritmos es mucho más manejable que su complejidad en espacio, pese a la representación compacta de la lista de adyacencia. El costo de mantener el *heap* es considerable pero el costo de almacenarlo es despreciable frente al costo de la lista de adyacencia. Considerando que ambos algoritmos que utilizan el grado de los vértices producen mejores resultados que la estrategia aleatoria, parece ser un costo razonable.

5. REFERENCES

[1] HOSLIB: Benchmarks with Hidden Optimum Solutions for Graph Problems http://www.nlsde.buaa.edu.cn/ kexu/benchmarks/graph-benchmarks.htm

[2] Experimental datasets Graphs, trees and parentheses https://users.dcc.uchile.cl/ jfuentess/datasets/graphs.php

[3] DIMACS Graphs: Benchmark Instances and Best Upper Bounds http://www.info.univ-angers.fr/pub/porumbel/graphs/

[4] Clique Benchmark Instances https://turing.cs.hbg.psu.edu/txn131/clique.html