INSTITUTO TECNOLÓGICO DE CIUDAD MADERO

DIVISIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN

TESIS

Métodos exactos para el problema de la bisección de vértices de un grafo conexo no dirigido

Para obtener el grado de:

Maestro en Ciencias Computacionales

Presenta:

I.M. Eduardo Rodríguez Del Angel

Director de tesis:

Dr. Héctor Joaquín Fraire Huacuja

Ciudad Madero, Tamaulipas Noviembre 2014

SEP
SECRETARÍA DE EDUCACIÓN PÚBLICA

TECNOLÓGICO NACIONAL DE MÉXICO
Instituto Tecnológico de Ciudad Madero

"2014, Año de Octavio Paz"

Cd. Madero, Tamps; a 20 de Octubre de 2014.

OFICIO No.: U5.240/14
AREA: DIVISIÓN DE ESTUDIOS
DE POSGRADO E INVESTIGACIÓN
ASUNTO: AUTORIZACIÓN DE IMPRESIÓN DE TESIS

ING. EDUARDO RODRÍGUEZ DEL ÁNGEL
NO. DE CONTROL G12072016
P R E S E N T E

Me es grato comunicarle que después de la revisión realizada por el Jurado designado para su examen de grado de Maestría en Ciencias de la Computación, el cual está integrado por los siguientes catedráticos:

PRESIDENTE :	DR. RODOLFO ABRAHAM PAZOS RANGEL
SECRETARIO :	DRA. GUADALUPE CASTILLA VALDEZ
VOCAL :	DR. HÉCTOR JOAQUÍN FRAIRE HUACUJA
SUPLENTE	DR. ARTURO HERNÁNDEZ RAMÍREZ
DIRECTOR DE TESIS :	DR. HÉCTOR JOAQUÍN FRAIRE HUACUJA

Se acordó autorizar la impresión de su tesis titulada:

"MÉTODOS EXACTOS PARA EL PROBLEMA DE LA BISECCIÓN DE VÉRTICES DE UN GRAFO CONEXO NO DIRIGIDO"

Es muy satisfactorio para la División de Estudios de Posgrado e Investigación compartir con Usted el logro de esta meta.

Espero que continúe con éxito su desarrollo profesional y dedique su experiencia e inteligencia en beneficio de México.

A T E N T A M E N T E
"POR MI PATRIA Y POR MI BIEN"®

M. P. MARÍA YOLANDA CHÁVEZ CINCO
JEFA DE LA DIVISIÓN

S.E.P.
DIVISION DE ESTUDIOS
DE POSGRADO E
INVESTIGACION
I T C M

c.c.p.- Archivo
Minuta

MYCHC 'NLZO ' jar

Ave. 1° de Mayo y Sor Juana I. de la Cruz, Col. Los Mangos, CP. 89440 Cd. Madero, Tam.
Tel. (833) 357 48 20, Fax, Ext. 1002, e-mail: itcm@itcm.edu.mx
www.itcm.edu.mx

ISO 9001 CERTIFIED COMPANY
ISO 14001 CERTIFIED COMPANY

“Debes apreciar los desvíos que tiene la vida,

porque solo recorriéndolos, descubrirás

las cosas que son más importantes

que lo que habías estado buscando”

Declaración de originalidad

Declaro y prometo que este documento de tesis es producto de mi trabajo y que no infringe los derechos de terceros, tales como derechos de publicación, derechos de autor, patentes y similares.

Además, declaro que en las citas textuales que he incluido y en los resúmenes que he realizado de publicaciones ajenas, indico explícitamente los datos de los autores y publicaciones.

Asimismo, en caso de infracción de los derechos de terceros derivada de este documento de tesis, acepto la responsabilidad de la infracción y relevo de ésta a mi director de tesis, así como al Instituto Tecnológico de Ciudad Madero y sus autoridades.

I.M. Eduardo Rodríguez Del Angel

Contenido

Declaración de originalidad 3
Contenido 4
Índice de figuras 7
Índice de algoritmos 8
Índice de tablas 8
Resumen 9
Abstract 10
Capítulo 1 Introducción 11
1.1 Introducción al problema de la bisección de vértices 11
1.2 Objetivos del proyecto 13
1.2.1 Objetivo general 13
1.2.2 Objetivos específicos 13
1.3 Justificación 14
1.4 Alcances y limitaciones 14
Capítulo 2 Marco teórico 15
2.1 Métodos de optimización 15
2.1.1 Problema de optimización 15
2.1.2 Vecindad y óptimos locales 16
2.2 Complejidad computacional 17
2.3 Métodos exactos 18
2.4 Métodos heurísticos 19
2.5 Métodos metaheurísticos 20

2.5.1 Intensificación y diversificación 20
Capítulo 3 Estado del arte 22
3.1 Problemas relacionados con el problema de la bisección de vértices 22
3.2 Trabajos que se relacionan con el problema de la bisección de vértices 24
Capítulo 4 Descripción del problema 25
4.1 Bisección de vértices 26
4.2 Problema de la bisección de vértices 27
4.3 Cálculo de la función objetivo 27
4.4 Solución exhaustiva de la bisección de vértices 29
4.5 Estructura del problema 31
Capítulo 5 Métodos exactos propuestos 32
5.1 Modelos de programación 32
5.1.1 Variables binarias utilizadas en los modelos 32
5.1.2 Modelo de programación cuadrático 33
5.1.3 Modelo de programación lineal entera 34
5.1.4 Modelo de programación lineal entera compactado 35
5.1.5 Comparación entre los modelos 36
5.2 Método B&B basado en permutaciones 39
5.2.1 Ejemplo del proceso para generar permutaciones 42
5.2.2 Ejemplo del B&B utilizando permutaciones 43
5.3 Método enumerativo para combinaciones 44
5.3.1 Método generador de combinaciones 47
5.3.2 Ejemplo del método enumerativo para combinaciones 50
Capítulo 6 Resultados experimentales 51
6.1 Instancias 52

6.2 Resultados 53
6.3 Análisis de los resultados 54
Capítulo 7 Conclusiones y trabajos futuros 55
Anexo A: Óptimos encontrados 56
Referencias 60

Índice de figuras

Figura 1.1) Tarea de divulgación 11
Figura 1.2) Tarea de acumulación 11
Figura 1.3) Tarea de gossip 12
Figura 2.1) Función objetivo con un mínimo local, un mínimo global y una vecindad 17
Figura 4.1) Ejemplos del cálculo de la bisección de vértices 26
Figura 4.2) Ejemplos del cálculo de la función objetivo. 28
Figura 4.3) Ejemplo algoritmo exhaustivo 30
Figura 4.4) Frecuencias del valor objetivo 31
Figura 5.1) Restricciones y variables del modelo cuadrático 37
Figura 5.2) Restricciones y variables del ILP1 38
Figura 5.3) Restricciones y variables del ILP2 38
Figura 5.4) Construcción de las permutaciones 43
Figura 5.5) Grafo de 4 nodos para ejemplificar el B&B 43
Figura 5.6) Grafo de 4 nodos para ejemplificar el método enumerativo para combinaciones 50
Figura 6.1) Óptimos encontrados 53
Figura 6.2) Promedio de soluciones 54

Índice de algoritmos

Algoritmo 1) Solución exhaustiva de la bisección de vértices. 30
Algoritmo 2) B&B de permutaciones 42
Algoritmo 3) Enumerativo por combinaciones 47
Algoritmo 4) Método generador de combinaciones 48

Índice de tablas

Tabla 4.1) Elementos del problema de la bisección de vértices 25
Tabla 4.2) Soluciones repetidas 31
Tabla 5.1) Solución de una instancia por el B&B 44
Tabla 5.2) Ejemplo del generador de permutaciones 49
Tabla 5.3) Solución de una instancia por el enumerativo basado en combinaciones 50
Tabla 6.1) Instancias utilizadas en la experimentación 52
Tabla 6.2) Resultados finales de la experimentación 53
Tabla Anexo A.1) Óptimos de las instancias grid 56
Tabla Anexo A.2) Óptimos de las instancias Harwell - Boeing 56
Tabla Anexo A.3) Óptimos de las instancias tree 56
Tabla Anexo A.4) Óptimos de las instancias small 57

Resumen

En este trabajo de investigación se abordó el problema de bisección de vértices de un grafo conexo no dirigido mediante métodos exactos.

Este problema pertenece a la clase NP-duro y consiste en separar a los nodos de un grafo en dos subconjuntos del mismo tamaño (izquierda y derecha), de tal manera que se minimice la cantidad de vértices en el conjunto izquierdo que tengan al menos una arista que lo conecte al conjunto derecho.

Las contribuciones más importantes del trabajo son un modelo de programación lineal entera (ILP2) y un algoritmo enumerativo basado en combinaciones que aprovecha la estructura del problema para delimitar el espacio de soluciones. Se realizaron pruebas experimentales para estos algoritmos y para otros dos métodos de solución propuestos, en estas pruebas se utilizaron 108 instancias estándar para problemas de ordenamiento lineal.

- El modelo de programación ILP2 obtuvo 99 soluciones óptimas lo cual corresponde a un 91.67%.
- El método enumerativo por combinaciones obtuvo 103 soluciones óptimas lo cual corresponde a un 95.37%.

Ambos programas fueron sujetos a los mismos parámetros de tiempo para cada uno de los experimentos. Los resultados parciales de este trabajo se incorporaron en un artículo titulado "Exact Methods for the Vertex Bisection Problem", el cual fue publicado en el libro "Recent Advances on Hybrid Approaches for Designing Intelligent Systems, Studies in Computational Intelligence".

Palabras clave: Bisección de vértices, métodos exactos, branch and bound, programación lineal entera.

Abstract

In this thesis we approach the vertex bisection problem (VB), for this purpose several exact methods were developed.

This problem belongs to the NP-hard set of problems and it consists in splitting the nodes of a graph in two subsets of the same size (left, right), therefore, the objective is to minimize the number of vertexes on the left set that has a connection to the right set.

The most important contributions of this work are a integer linear programming model (ILP2) and an enumerative algorithm based on combination. The experimental results were carried out with 108 standard instances to assess the performance of these algorithms and other two integer linear programming methods.

- The ILP2 model obtained 99 optimal solutions which is about a 91.67% of the instances.
- The enumerative method that uses combinations obtained 103 optimal solutions which is about 95.37% of the instances.

Both programs were bound to the same time limit for each one of the experiments. The partial experimental results of this work were published on a book chapter entitled "Exact Methods for the Vertex Bisection Problem", which was published in "Recent Advances on Hybrid Approaches for Designing Intelligent systems, Studies in computational Intelligence"

Keywords: Vertex Bisection, exact methods, branch and bound, integer linear programming.

Capítulo 1 Introducción

1.1 Introducción al problema de la bisección de vértices

Una red de comunicaciones se puede representar como un grafo $G = (V, E)$, donde *V* representa los vértices o nodos de la red y *E* representa las aristas o conexiones entre los nodos. Cada vértice representa un dispositivo que almacena y distribuye información. Las tres principales tareas que se realizan en una red de comunicaciones para distribuir la información son: divulgación, acumulación y *gossip* [Böckenhauer 1999].

La tarea de divulgación (figura 1.1) consiste en difundir un mensaje *m1* de un vértice a todos los nodos de la red.

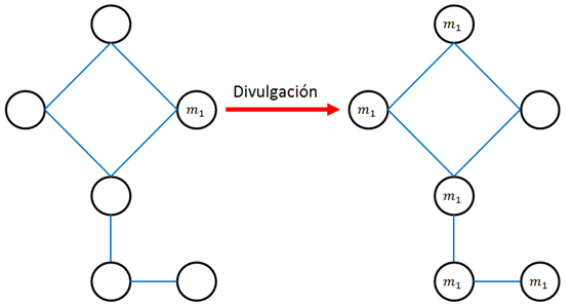


Figura 1.1) Tarea de divulgación

La tarea de acumulación (figura 1.2) en un vértice consiste en almacenar la información recibida de otros nodos *m1* y *m2*.

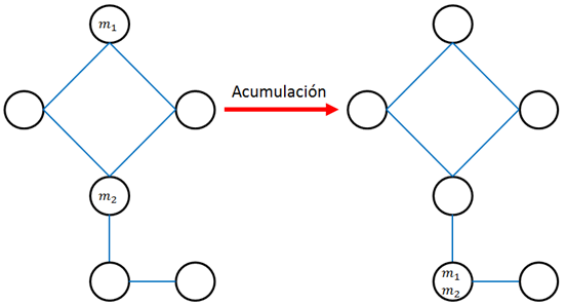


Figura 1.2) Tarea de acumulación

La tarea de *gossip* (figura 1.3) permite alcanzar el propósito de la tarea de divulgación de una manera más eficiente, en lugar de distribuir la información de un vértice a todos los nodos, se divide la red en dos subgrafos con aproximadamente la misma cantidad de nodos y se identifican los nodos conectados entre ambas regiones de la red.

Dichos nodos se utilizan para acumular (distribuir) la información en la región en que se ubican y los enlaces son utilizados para transmitir información de una región a la otra. La separación de los vértices de la red se realiza de tal manera que el número de vértices conectados entre ambas regiones sea mínimo.

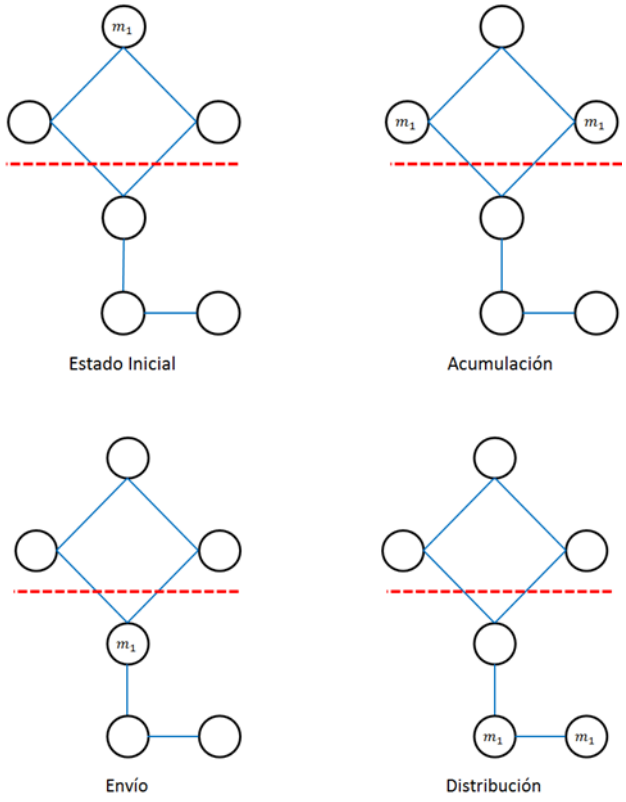


Figura 1.3) Tarea de gossip

Este problema se conoce como el problema de la bisección de vértices de un grafo conexo, no dirigido. En [Brandes 2009] se demuestra que este problema es *NP-duro*.

1.2 Objetivos del proyecto

En esta sección se describirán los objetivos que se propusieron cumplir en el desarrollo del proyecto.

1.2.1 Objetivo general

Desarrollar un método de solución exacta del problema de bisección de vértices basado en un modelo de programación lineal entera y un método de tipo ramificación y acotamiento, *Branch and Bound*, que aproveche la estructura del problema.

1.2.2 Objetivos específicos

- Revisar la literatura relacionada con los métodos de solución del problema bisección de vértices.
- Revisar el marco teórico relacionado con el problema y los métodos de solución exactos.
- Diseñar e implementar un método de solución basado en un modelo de programación lineal entera.
- Diseñar e implementar un método de solución basado en ramificación y acotamiento.
- Evaluar experimentalmente los métodos desarrollados.

1.3 Justificación

El problema de la bisección de vértices, tiene importantes aplicaciones en el área de comunicación, porque permite resolver el problema conocido como *gossip*. Otra aplicación que tiene la bisección de vértices es dividir circuitos de gran escala en dos de menor tamaño. Se ha demostrado que la bisección de vértices pertenece a la categoría de los problemas de complejidad NP-completos, lo cual significa que no existe un algoritmo determinista que lo resuelva en tiempo polinomial.

Actualmente, no existe un modelo de programación lineal entera, ni métodos exactos que resuelvan este problema. Existen en el estado del arte, soluciones aproximadas para algunas versiones del problema, en general centradas en resolver grafos con un número par de nodos. En este trabajo se propone desarrollar métodos exactos de solución del problema de la bisección de vértices.

1.4 Alcances y limitaciones

- En este proyecto se desarrollarán métodos de solución exacta para el problema de la bisección de vértices.
- Para la solución del modelo de programación lineal entera se utilizará el optimizador CPLEX.
- El desempeño de los métodos de solución se evaluará experimentalmente utilizando instancias asociadas a redes de tipo *scale-free* e instancias estándar.
- El proyecto se desarrollará utilizando el lenguaje de programación C#.

Capítulo 2 Marco teórico

2.1 Métodos de optimización

El concepto de optimización se concibe como el proceso de intentar encontrar la mejor solución posible a un problema de optimización, generalmente en un tiempo limitado. Se puede decir que un problema de optimización es simplemente un problema en el que hay varias (en general muchas) posibles soluciones y alguna forma clara de comparación entre ellas, de manera que éste existe si y sólo si se dispone un conjunto de soluciones candidatas diferentes que pueden ser comparadas [Duarte 2007].

La idea principal dentro de la optimización es hacer las cosas mejor, esta mejoría puede presentarse en forma de soluciones más precisas, en una mejora del tiempo de ejecución de un programa para resolver alguna problemática específica, otra forma de mejorar la solución de un problema es la robustez de un algoritmo, es decir, qué tan eficiente es nuestro algoritmo para diferentes tipos de instancias o características del problema.

2.1.1 Problema de optimización

Un problema de optimización P se formula como un trío $P = (f, SS, F)$, donde f es la función a optimizar, F es el conjunto de soluciones factible y SS (del inglés *Search Space*) es el espacio de soluciones. Se pueden clasificar los problemas de optimización tomando en cuenta cómo se codifica la solución del problema al que se enfrenta, lo que da lugar a dos categorías:

- Aquéllos en los que la solución es codifica mediante valores reales.
- Aquéllos cuyas soluciones se codifican con valores enteros, en este caso existe un subconjunto de problemas que es conocido como problemas binarios, es decir las variables que solamente pueden tomar el valor de 1 o de 0.

Entre los problemas que se pueden representar con valores enteros existe un subconjunto conocido como problemas de optimización combinatoria, los cuales consisten en hallar una permutación o alguna estructura del tipo grafo, dentro de un conjunto finito de posibilidades (espacio de soluciones) que arroje la mejor solución al problema de optimización ya sea maximizar o minimizar según se necesite.

2.1.2 Vecindad y óptimos locales

Como lo define [Duarte 2007] para una solución factible x que pertenezca al espacio de soluciones ($x \in SS$), se define la vecindad $N(x)$ como el subconjunto de soluciones factibles que se encuentran *próximas* a la solución propuesta x.

Para los problemas de optimización combinatoria la vecindad, queda definida por todas las soluciones cercanas tales que se puedan generar a partir de la solución x utilizando una única operación llamada *movimiento*.

Cuando se define un subconjunto de soluciones dentro del espacio SS, es posible analizarlo y encontrar ciertos valores como el valor mínimo dentro de este espacio, el cual no necesariamente tiene que ser el mínimo de todo el espacio SS, por lo cual es necesario definir la diferencia entre un mínimo local y un mínimo global.

- Mínimo global: Para el problema de optimización $P = (f, SS, F)$, se dice que una solución factible x, la cual pertenece al espacio de soluciones $x \in F \subseteq SS$, es un mínimo global si:

$$f(x) \leq f(y) \ , \quad \forall\, y \in F \subseteq SS$$

- Mínimo local: Para el problema de optimización $P = (f, SS, F)$ y un subconjunto de soluciones conocido como vecindad $N(x)$, se dice que una solución factible x, la cual pertenece al espacio de soluciones $x \in F \subseteq SS$, es un mínimo local si:

$$f(x) \leq f(y) \ , \quad \forall\, y \in N(x) \subseteq SS$$

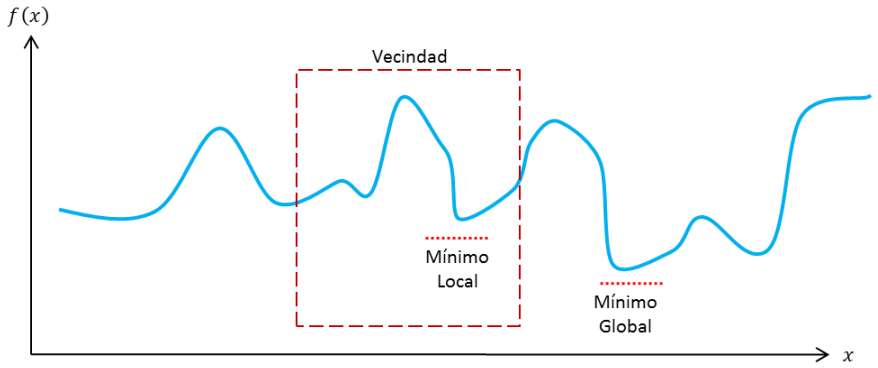


Figura 2.1) Función objetivo con un mínimo local, un mínimo global y una vecindad

2.2 Complejidad computacional

Los problemas de optimización se clasifican dependiendo de la dificultad que tiene generar su solución. Las clases principales de clasificación son problemas P, NP, $NP - Completo$ [Sipser 2006]:

- Clase P: Es el conjunto de todos los problemas de decisión que pueden ser decididos en tiempo polinomial por un algoritmo determinista. A los problemas que pertenecen a esta clase se les denomina tratables.
- Problemas intratables: Son todos los problemas de decisión para los que no existe algún algoritmo determinista de tiempo polinomial que los decide. Son todos los problemas que están en P^C.
- Clase NP: La clase NP es el conjunto de todos los problemas de decisión que son decididos en tiempo polinomial con un algoritmo no determinista.
- Clase NPC: Son los problemas de decisión que pertenecen a la clase NP y a los cuales se transforman polinomialmente todos los problemas de NP.

2.3 Métodos exactos

Los métodos exactos de resolución de problema consisten en la exploración de todo el espacio de soluciones, este tipo de algoritmos garantizan la *solución óptima* para el problema, a diferencia de otros métodos que suelen estancarse en óptimos locales, esto se debe a que explora todas las soluciones y las compara hasta obtener la mejor solución.

Los métodos exactos tienen como característica principal que, dada una instancia del problema a resolver, siempre obtendrán una misma salida. Los problemas combinatorios presentan como particularidad que siempre existe un algoritmo exacto que permite obtener la solución óptima, aunque muchas veces suele ser muy costoso implementarlo.

Los métodos exactos de resolución de problemas se han aplicado con éxito a una cantidad elevada de problemas. Algunos ejemplos de estos métodos son los algoritmos voraces, algoritmos de divide y vencerás, algoritmos de ramificación y poda, entre otros. Los algoritmos exactos son dependientes completamente del problema para el que se diseñan, lo cual significa que no se pueden adaptar con facilidad. Algunos algoritmos exactos se describirán a continuación:

- Retropropagación: este método consiste en generar todas las posibles soluciones en un árbol para explorar todas las soluciones. El árbol empieza a construirse a partir del último elemento que haya visitado y se exploran todos los elementos adyacentes que tenga, en caso de que un movimiento genere una solución no factible, esta solución se anulará y se volverá a posicionar en el primer elemento anterior que tenga elementos que aún no han sido visitados. El recorrido tiene éxito si, procediendo de esta forma, se puede definir por completo una solución. En este caso el algoritmo puede bien detenerse (si lo único que se necesita es una solución del problema) o bien seguir buscando soluciones alternativas (si deseamos examinarlas todas).

- Ramificación y Acotamiento (B&B por sus siglas en inglés es: Branch and Bound): este algoritmo es muy parecido a la retropropagación con la principal diferencia que va evaluando las posibles soluciones y eliminando las ramas que no arrojen buenas soluciones con el fin de seccionar el problema en un problema de menor tamaño y delimitar el espacio de soluciones. Propone establecer subproblemas que son más fáciles de resolver que el problema original, por su tamaño menor o una estructura más susceptible.

2.4 Métodos heurísticos

Para los problemas donde la ejecución de un algoritmo exacto sea muy tardada, se propone resolverlos por métodos aproximados conocidos como métodos heurísticos los cuales permiten encontrar una solución factible con buena calidad en un tiempo sumamente pequeño en comparación del método exacto. El término heurística proviene del vocablo griego *heuriskein* que se traduce como encontrar, descubrir o hallar.

Los métodos heurísticos son procedimientos simples, a menudo basados en el sentido común, que se supone que obtendrán una buena solución (no necesariamente óptima) a problemas difíciles de un modo sencillo y rápido. Una limitación muy grande de los métodos heurísticos es que suelen estancarse en soluciones conocidas como óptimos locales.

Los métodos heurísticos tienen su principal limitación en su incapacidad para escapar de óptimos locales. Esto se debe, fundamentalmente, a que estos algoritmos no utilizan ningún mecanismo que les permita proseguir la búsqueda del óptimo en el caso de quedar atrapados en un óptimo local. Para solventar este problema, se introducen otros algoritmos de búsqueda más inteligentes que evitan, en la medida de lo posible, este problema. Este tipo de algoritmos son procedimientos de alto nivel que guían a métodos heurísticos conocidos, evitando que éstos queden atrapados en óptimos locales.

2.5 Métodos metaheurísticos

El término metaheurística fue acuñado por F. Glover en el año 1986. Etimológicamente, deriva de la composición de dos palabras "meta" y "heurística". El prefijo meta se podría traducir como más allá de, en un nivel superior. Con este término, Glover pretendía definir un procedimiento maestro de alto nivel que guía y modifica otras heurísticas para explorar soluciones más allá de la simple optimalidad local.

Las metaheurísticas son una clase de métodos aproximados que están diseñados para resolver problemas difíciles de optimización combinatoria en los que los heurísticos clásicos no son efectivos. Las metaheurísticas proporcionan un marco general para crear nuevos algoritmos híbridos combinando diferentes conceptos derivados de la inteligencia artificial, la evolución biológica y los mecanismos estadísticos.

La evolución de las metaheurísticas durante los últimos 25 años ha tenido un comportamiento prácticamente exponencial. Desde las primeras reticencias por su supuesta falta de rigor científico hasta la actualidad, se han resuelto problemas que inicialmente parecían inabordables.

Las metaheurísticas incluyen métodos como optimización por colonias de hormigas (ACO), algoritmos evolutivos (EA), donde se incluyen los algoritmos genéticos (GA) y los algoritmos meméticos (MA), procedimientos de búsqueda miope (constructiva, voraz o ávida), aleatorizados y adaptativos (GRASP), búsqueda local iterativa (ILS), recocido simulado (SA), búsqueda dispersa (SS) y búsqueda tabú (TS) [Duarte 2007].

2.5.1 Intensificación y diversificación

La intensificación es la forma de definir el proceso de búsqueda más exhaustivo. Generalmente, la metaheurística no intensifica en cualquier entorno, sino que tiene en cuenta factores como la calidad de las soluciones encontradas en esa vecindad. Es decir, si en esa vecindad no hay ninguna solución de calidad no parece interesante intensificar la búsqueda en dicha región.

La diversificación es la forma de definir el proceso mediante el cual la metaheurística es capaz de visitar diversas vecindades lejanas. Habitualmente, la metaheurística no diversifica constantemente y sin criterio, ya que eso sería una búsqueda aleatoria en el espacio de soluciones, sino que tiene en cuenta factores, como por ejemplo el hecho de que una región no haya sido visitada.

Capítulo 3 Estado del arte

3.1 Problemas relacionados con el problema de la bisección de vértices

Existen algunos problemas relacionados con el problema de bisección de vértices, como son el problema de bisección de vértices pares, el problema de bisección para grafo G_{npr} y el problema de bisección de vértices con costos en las aristas, a continuación se describen cada uno de los siguientes problemas:

- **Problema bisección de vértices pares** [Liu 2009]. En ese trabajo se define el problema de la mínima bisección el cual consiste en partir un grafo en dos conjuntos con una cantidad de elementos iguales de tal forma que se minimice la cantidad de aristas que tiene un extremo entre ambos conjuntos.

  A diferencia del problema que se intenta resolver en este trabajo donde el cruce se realiza en $\lfloor n / 2 \rfloor$, evitando esta ambigüedad. Ese problema se aborda utilizando un algoritmo DNA, que se inspira en la biología, el cual tiene como características principales el uso de almacenamiento de memoria, funciones en paralelo y consume poca energía.

- **Problema bisección para grafos del tipo G_{npr}** [Jerrum 1993]. El problema de la bisección que describe el autor en ese documento es el siguiente: Sea un grafo $G = (V, E) \in G_{npr}$ donde existen dos conjuntos de cardinalidad igual a $n / 2$, conocidos como L (conjunto izquierdo) y R (conjunto derecho del punto de corte) con una probabilidad p de que exista una arista entre dos nodos de un mismo conjunto y con una probabilidad r de que una arista una a dos nodos de diferentes conjuntos, donde n es un número par de nodos.

Una partición o bisección π del grafo consiste en dividir el conjunto de nodos V en los dos conjuntos mencionados L y R para después contar el total de aristas que cruzan entre ambos conjuntos. Ese problema se aborda utilizando la técnica de programación conocida como recocido simulado (*simulated annealing*, en inglés).

- **Problema de bisección de grafos con costos en las aristas** [Karisch 2000]. En ese trabajo el autor describe al problema de la bisección como "Dado un grafo $G = (V, E)$ no dirigido, con costo en las aristas donde V es el conjunto de vértices del grafo y E el conjunto de aristas y el costo de las aristas está dada por una matriz de adyacencia A. El problema de la bisección de grafos consiste en dividir al conjunto V en dos subconjuntos *V1* y *V2*, tal que el costo total de las aristas que unan a los vértices de *V1* con los vértices del otro subconjunto sea mínimo. Estas aristas se dice que son cortadas por la bisección". El autor propone una técnica de solución exacta del tipo *Branch and Bound* que está basada en planos de corte.

  En la experimentación que se propone en ese trabajo se utilizan las siguientes instancias:

  - Instancias generadas aleatoriamente, con la característica de que todas las instancias tienen una cantidad de nodos entre 36 y 120, y tienen una probabilidad de que exista una arista que una un nodo con otro del 50 por ciento.
  - Instancias de la biblioteca Brunetta-Conforti-Rinaldi.
  - Las instancias conocidas como "*compiler design*" que generalmente se utilizan para el problema de cluster mínimo, se utilizan porque las aristas tienen peso.

Como conclusión sobre los trabajos que se han reportado hasta la fecha, se puede observar que para el problema de bisección de vértices con cualquier valor n (sea par o impar), donde el grafo no tenga pesos en sus aristas, no existen modelos de programación lineal entera ni soluciones del tipo ramificación y acotamiento reportados.

3.2 Trabajos que se relacionan con el problema de la bisección de vértices

En [Diaz 2000] y en [Petit 2002] se define de manera formal el problema de la bisección de vértices que se abordará en este trabajo (véase capítulo 4).

En [Brandes 2009] el autor realiza una demostración de que el problema pertenece a la clase NP-completo transformando polinomialmente el problema de la bisección de vértices, descrito en el trabajo de [Diaz 2000] y de [Petit 2002], en el problema 3-SAT (que ya se ha demostrado en la literatura que es un problema NP-completo) además también demuestra que para algunos grafos particulares como los grafos del tipo hipercubos y árboles es posible resolverlos utilizando un algoritmo polinomial. En ese trabajo el autor prueba no sólo el problema de la bisección de vértices sino también la bisección de aristas e indica que ambos problemas son relevantes cuando se trabaja con redes de comunicación donde se involucra el proceso de *gossip* (véase sección 1.1).

En [Hromkovic 1995] se realiza el estudio de una técnica de comunicación e intercambio de información conocida como *gossip* (véase sección 1.1), que consiste en acumular la información en algunos nodos de una red, para luego transmitirlos a los demás nodos, el proceso busca ser lo más eficaz posible y para ello busca la forma de acumular el mensaje en el menor número de nodos y luego transmitirlo por el menor número de conexiones posibles con la finalidad de ahorrar tiempo y recursos.

El autor propone utilizar la técnica de bisección de vértices con el fin de encontrar el menor número de nodos que tengan conexión entre dos secciones de una red para utilizarlos como nodos de recolección y transmitir la información a los nodos de la otra sección y después éstos distribuirán la información al resto de los nodos de esta sección.

Capítulo 4 Descripción del problema

Para calcular la bisección de vértices es necesario representar al grafo de forma lineal, donde cada alteración al orden de los vértices en el arreglo, genera una nueva permutación que deberá ser evaluada. El problema asocia a cada una de las posibles permutaciones un valor de la función objetivo. Una vez que se realiza esta representación se efectuará un corte entre los nodos $\lfloor n / 2 \rfloor$ y el $\lfloor n / 2 \rfloor + 1$. Para definir formalmente el problema se utiliza el marco teórico descrito en [Díaz 2000] que ocupa los elementos descritos en la tabla 4.1.

Tabla 4.1) Elementos del problema de la bisección de vértices

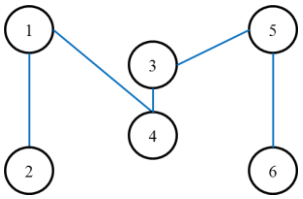
$(\boldsymbol{V}, \boldsymbol{E})$	Grafo conexo, no dirigido, acíclico, cuyas aristas no son ponderadas.
$\boldsymbol{V}$	Conjunto de los vértices del grafo.
$\boldsymbol{E}$	Conjunto de las aristas del grafo.
$\boldsymbol{\varphi}$	Permutación u ordenamiento lineal de los vértices del grafo.
$\boldsymbol{\Phi}$	Conjunto de todas las ordenaciones lineales o permutaciones de los vértices del grafo.
$\boldsymbol{n}$	Número de vértices del grafo $n = \lvert V \rvert$.
$\boldsymbol{L}(\lfloor \boldsymbol{n} / \boldsymbol{2} \rfloor, \boldsymbol{\varphi}, \boldsymbol{G})$	Conjunto de vértices que quedan del lado izquierdo del punto de corte intermedio ($\lfloor n / 2 \rfloor$, $\lfloor n / 2 \rfloor + 1$).
$\boldsymbol{R}(\lfloor \boldsymbol{n} / \boldsymbol{2} \rfloor, \boldsymbol{\varphi}, \boldsymbol{G})$	Conjunto de vértices que quedan del lado derecho del punto de corte intermedio ($\lfloor n / 2 \rfloor$, $\lfloor n / 2 \rfloor + 1$).
$\boldsymbol{\delta}(\lfloor \boldsymbol{n} / \boldsymbol{2} \rfloor, \boldsymbol{\varphi}, \boldsymbol{G})$	Conjunto de vértices en L y que están conectados con un vértice de R por al menos una arista.

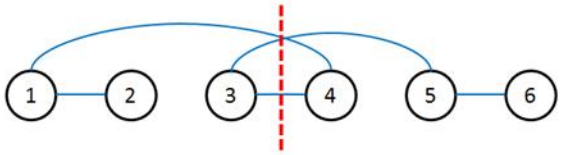
4.1 Bisección de vértices

Dado un grafo conexo no dirigido $G = (V, E)$ y una permutación $\varphi \in \Phi$, la bisección de vértices (ecuación 4.1) del grafo se define como [Petit 2002]:

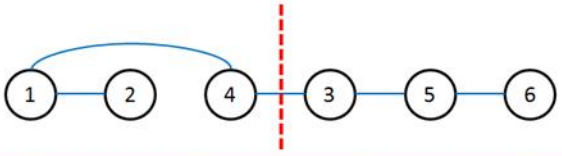
$$VB(\varphi, G) = |\delta(\lfloor n / 2 \rfloor, \varphi, G)| \tag{4.1}$$

A continuación se mostrará un grafo con 6 nodos, así como la bisección de dos de sus posibles acomodos (permutaciones). Sea el siguiente grafo (figura 4.1):





Permutación φ	$L\left(\lfloor \frac{n}{2} \rfloor, \varphi, G\right)$	$R\left(\lfloor \frac{n}{2} \rfloor, \varphi, G\right)$	$\delta\left(\lfloor \frac{n}{2} \rfloor, \varphi, G\right)$	$VB(\varphi, G) = \lvert\delta\rvert$
{1,2,3,4,5,6}	{1,2,3}	{4,5,6}	{1,3}	2



Permutación φ	$L\left(\lfloor \frac{n}{2} \rfloor, \varphi, G\right)$	$R\left(\lfloor \frac{n}{2} \rfloor, \varphi, G\right)$	$\delta\left(\lfloor \frac{n}{2} \rfloor, \varphi, G\right)$	$VB(\varphi, G) = \lvert\delta\rvert$
{1,2,4,3,5,6}	{1,2,4}	{3,5,6}	{4}	1

Figura 4.1) Ejemplos del cálculo de la bisección de vértices

La bisección de vértices $VB(\varphi_2, G)$ para la segunda permutación $\varphi_2 = \{1,2,4,3,5,6\}$ es menor que la bisección $VB(\varphi_1, G)$ utilizando la permutación $\varphi_1 = \{1,2,3,4,5,6\}$.

4.2 Problema de la bisección de vértices

Dado un grafo conexo no dirigido $G = (V, E)$, el problema de la bisección de vértices consiste en determinar la permutación $\varphi' \in \Phi$ que minimice la bisección de los vértices de G. De manera formal esto se expresa de la siguiente manera:

$$VB(\varphi', G) = \min_{\varphi \in \Phi}\{VB(\varphi, G)\} \tag{4.2}$$

4.3 Cálculo de la función objetivo

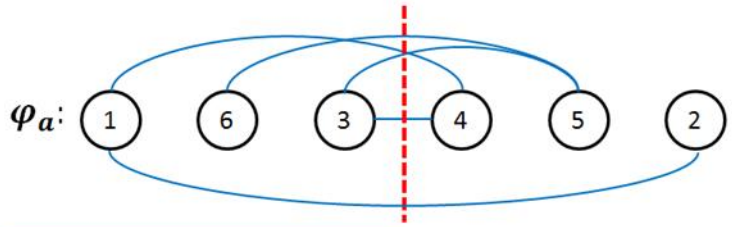
El valor de la función objetivo, se puede calcular utilizando una variable binaria que indica si un vértice i perteneciente al subconjunto $L(\lfloor n \,/\, 2 \rfloor, \varphi, G)$ tiene relación por medio de una arista con algún vértice j que pertenezca al subconjunto $R(\lfloor n \,/\, 2 \rfloor, \varphi, G)$. Se debe verificar si existe el arco (v_i, v_j) como parte del conjunto de arcos *E*. Los valores de i y j representan la posición de los vértices en la permutación.

$$e_i = \begin{cases} 1 & \text{si } \exists\, j \mid (v_i, v_j) \in E \quad \text{para } 1 \leq i \leq \lfloor n \,/\, 2 \rfloor \;, \lfloor n \,/\, 2 \rfloor < j \leq n \\ 0 & \text{en caso contrario} \end{cases} \tag{4.3}$$

Una vez determinado el valor de cada uno de los elementos del conjunto L de la permutación actual, el valor de la bisección de vértices para esa permutación $VB(\varphi, G)$, se obtiene sumando los valores individuales de cada una de las variables binarias.

$$VB(\varphi, G) = \sum_{i=1}^{\lfloor n/2 \rfloor} e_i \, VB(\varphi', G) = \min_{\varphi \in \Phi}\{VB(\varphi, G)\} \tag{4.4}$$

En la figura 4.2 se muestran a modo de ejemplo los cálculos de la función objetivo para 2 permutaciones del mismo grafo.

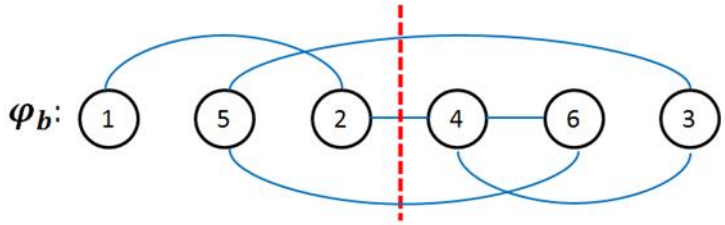


e_1	e_2	e_3
$(1,4) \in E$	$(6,4) \notin E$	$(3,4) \in E$
$(1,5) \notin E$	$(6,5) \in E$	$(3,5) \in E$
$(1,2) \in E$	$(6,2) \notin E$	$(3,2) \notin E$
$e_1 = 1$	$e_2 = 1$	$e_3 = 1$

$$VB(\varphi_a, G) = \sum_{i=1}^{3} e_i$$

$$VB(\varphi_a, G) = e_1 + e_2 + e_3$$

$$VB(\varphi_a, G) = 3$$



e_1	e_2	e_3
$(1,4) \notin E$	$(5,4) \notin E$	$(2,4) \in E$
$(1,6) \notin E$	$(5,6) \in E$	$(2,6) \notin E$
$(1,3) \notin E$	$(5,3) \in E$	$(2,3) \notin E$
$e_1 = 0$	$e_2 = 1$	$e_3 = 1$

$$VB(\varphi_a, G) = \sum_{i=1}^{3} e_i$$

$$VB(\varphi_a, G) = e_1 + e_2 + e_3$$

$$VB(\varphi_a, G) = 2$$

Figura 4.2) Ejemplos del cálculo de la función objetivo.

4.4 Solución exhaustiva de la bisección de vértices

Para que un algoritmo se considere exhaustivo para el problema de la bisección de vértices deberá generar y evaluar todas las posibles permutaciones de *n*-elementos, donde *n* es el número de nodos de la instancia a resolver. A continuación se propone el siguiente algoritmo exhaustivo para la solución de la bisección de vértices:

Algoritmo exhaustivo

Función del algoritmo: Generar y evaluar todas las permutaciones de una instancia del problema de la bisección de vértices.

Entradas:

n: número de vértices que tiene la instancia

E: conjunto de aristas que tiene la instancia

Estructuras:

$i = 1, 2, \ldots, n!$

P: conjunto de permutaciones de *n* elementos

perm: permutación actual que se va a evaluar

costo: valor objetivo asociado a la permutación actual

Funciones:

GeneradorPermutaciones()

Función que se encarga de producir todas las permutaciones posibles de *n* elementos

Evalua(perm)

Función que regresa el valor objetivo asociado a la permutación actual

Salida:

mejorPermEncontrada: devuelve la permutación que tenga la menor *BV*

minCosto: devuelve el valor objetivo de la mejor permutación encontrada

```
1   Lectura_instancia();
2   minCosto = ∞;
3   P = GeneradorPermutaciones();
4   Do {
5       perm = i | i ∈ P;
6       P = P \ i;
7       costo = Evalua(perm);
8       if (costo < minCost)
9       {
10          minCosto = costo;
11          mejorPermEncontrada = perm;
12      }
13  } until (P = ∅)
14  return (mejorPermEncontrada, minCosto);
```

Algoritmo 1) Solución exhaustiva de la bisección de vértices.

A continuación se ejemplifica el funcionamiento del algoritmo exhaustivo propuesto (figura 4.3), utilizando el grafo de la izquierda como instancia a resolver:

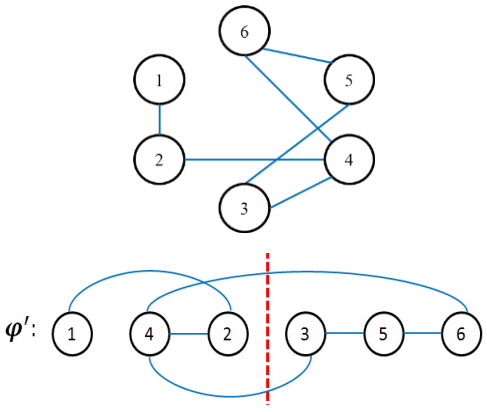


Figura 4.3) Ejemplo algoritmo exhaustivo

Permutación	VB
{1,2,3,4,5,6}	2
{1,2,3,4,6,5}	2
⋮	⋮
{1,3,4,2,5,6}	3
{1,3,4,2,6,5}	3
⋮	⋮
{1,4,2,3,5,6}	1

4.5 Estructura del problema

La tabla 4.2 muestra que la BV, tiene la característica de que muchas soluciones del espacio de búsqueda producen el mismo valor objetivo y que el rango de valores es relativamente pequeño. El enfoque general que se utiliza en este proyecto consiste en diseñar métodos de solución del problema de la BV que aprovechen dicha estructura.

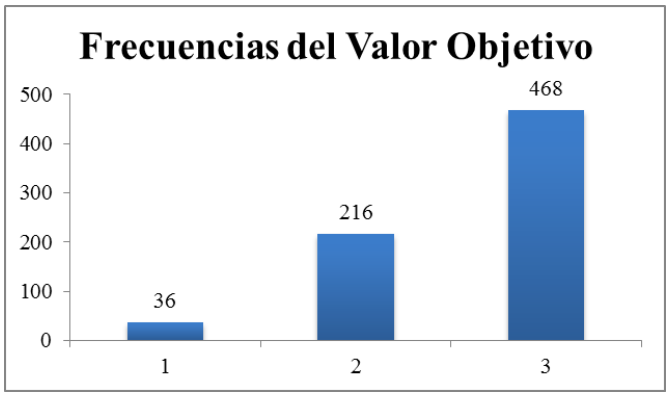


Figura 4.4) Frecuencias del valor objetivo

Para el caso del ejemplo resuelto por el método exhaustivo con 6 vértices, existen $6! = 720$ permutaciones, las cuales están registradas en el gráfico anterior, 36 de las permutaciones coinciden en el óptimo global, aunque algunas permutaciones de éstas son solamente las misma permutaciones pero reordenadas, sin intercambiar elementos entre el conjunto L y el conjunto R. En los siguientes ejemplos se observa que los elementos cambian su orden pero no cambian el subconjunto donde se encuentran los elementos:

Tabla 4.2) Soluciones repetidas

Permutación	Valor Objetivo
{1,2,4,3,5,6}	1
{1,4,2,3,5,6}	1
{2,1,4,3,5,6}	1
{2,4,1,3,5,6}	1
{4,1,2,3,5,6}	1
{4,2,1,3,5,6}	1

Capítulo 5 Métodos exactos propuestos

5.1 Modelos de programación

En la siguiente sección se describirán 3 modelos de programación, el primero de ellos del tipo cuadrático y los otros dos del tipo lineal, primero se introducirán las variables que se ocuparán en los 3 modelos, seguido por los modelos detallados.

5.1.1 Variables binarias utilizadas en los modelos

Para los modelos de programación propuestos para resolver el problema de la bisección de vértices, se utilizan 3 tipos de variables binarias: x_u^p, $y_{u,v}^{p,q}$ y z_p.

Las variables del tipo x_u^p se utilizan para representar el ordenamiento que tienen los elementos en la permutación actual y se definen como:

$$x_u^p = \begin{cases} 1, & p = \varphi(u) \\ 0, & \text{en caso contrario} \end{cases}$$

Las variables del tipo $y_{u,v}^{p,q}$ se utilizan para modelar la conectividad del grafo y se define como $y_{u,v}^{p,q} = x_u^p \wedge x_v^q$, es decir, si existe una arista que una a ese par de nodos en la permutación actual, se puede también definir a estas variables de la siguiente forma:

$$y_{u,v}^{p,q} = \begin{cases} 1, & x_u^p = 1 \wedge x_v^q = 1 \\ 0, & \text{en caso contrario} \end{cases}$$

El último tipo de variables z_p sirve para representar la conexión de los vértices en el lado izquierdo del punto de corte $c = \lfloor n / 2 \rfloor$ con los vértices en el lado derecho del mismo punto de corte, y se define como:

$$z_p = \begin{cases} 1, & y_{u,v}^{p,q} = 1 \wedge p \leq \lfloor n / 2 \rfloor \wedge q > \lfloor n / 2 \rfloor \\ 0, & \text{en caso contrario} \end{cases}$$

5.1.2 Modelo de programación cuadrático

El modelo cuadrático se define utilizando las variables x_u^p, $y_{u,v}^{p,q}$ y z_p

$$\min \sum_{p=1}^{\lfloor n/2 \rfloor} z_p \tag{1}$$

Esta función está sujeta a las siguientes restricciones:

$$\sum_{p \in \{1,\dots,n\}} x_u^p = 1 \qquad \forall u = 1, \dots, n \tag{2}$$

$$\sum_{u \in \{1,\dots,n\}} x_u^p = 1 \qquad \forall p = 1, \dots, n \tag{3}$$

$$y_{u,v}^{p,q} = x_u^p \, x_v^{\mathrm{q}} \qquad \forall (u,v) \in E \vee (v,u) \in E, p, q = 1, \dots, n \tag{4}$$

$$z_p \leq \sum_{q=\lfloor n/2 \rfloor + 1}^{n} \sum_{u=1}^{n} \sum_{v=1}^{n} y_{u,v}^{p,q} \leq (\lfloor n \,/\, 2 \rfloor) z_p \qquad \forall p = 1, \dots, \lfloor n \,/\, 2 \rfloor \tag{5}$$

Las restricciones del tipo (2) y (3) garantizan que a cada posición le corresponde un nodo y a cada nodo le corresponde una posición respectivamente. Estos conjuntos de restricciones aseguran que el ordenamiento lineal del grafo sea factible.

La restricción (4) sirve para especificar el conjunto de aristas, en función de las posiciones relativas de los vértices $u, v \in E$ en el ordenamiento lineal.

La restricción definida en (5) permite determinar el conjunto de vértices en el lado izquierdo del punto de corte $c = \lfloor n \,/\, 2 \rfloor$, que están conectados con los vértices del lado derecho en dicho punto de corte.

El valor obtenido por la restricción (5) es la bisección de vértices del grafo *G*, el cual se intenta minimizar en la función objetivo del modelo (1). Este modelo sirve de base a dos modelos de programación lineal entera que surgen como resultado de linealizar la restricción (4).

5.1.3 Modelo de programación lineal entera

El modelo de programación lineal entera (ILP1) se define utilizando las variables x_u^p, $y_{u,v}^{p,q}$ y z_p y la técnica de linealización tradicional.

$$\min \sum_{p=1}^{\lfloor n/2 \rfloor} z_p \tag{6}$$

Esta función está sujeta a las siguientes restricciones:

$$\sum_{p \in \{1,\dots,n\}} x_u^p = 1 \qquad \forall u = 1, \dots, n \tag{7}$$

$$\sum_{u \in \{1,\dots,n\}} x_u^p = 1 \qquad \forall p = 1, \dots, n \tag{8}$$

$$y_{u,v}^{p,q} \leq x_u^p \qquad \forall (u,v) \in E \vee (v,u) \in E, p, q = 1, \dots, n \tag{9}$$

$$y_{u,v}^{p,q} \leq x_v^q \qquad \forall (u,v) \in E \vee (v,u) \in E, p, q = 1, \dots, n \tag{10}$$

$$x_u^p + x_v^q \leq y_{u,v}^{p,q} + 1 \qquad \forall (u,v) \in E \vee (v,u) \in E, p, q = 1, \dots, n \tag{11}$$

$$z_p \leq \sum_{q=\lfloor n/2 \rfloor + 1}^{n} \sum_{u=1}^{n} \sum_{v=1}^{n} y_{u,v}^{p,q} \leq (\lfloor n \,/\, 2 \rfloor) z_p \qquad \forall p = 1, \dots, \lfloor n \,/\, 2 \rfloor \tag{12}$$

Las restricciones del tipo (7) y (8) garantizan que a cada posición le corresponde un nodo y a cada nodo le corresponde una posición respectivamente. Estos conjuntos de restricciones aseguran que el ordenamiento lineal del grafo sea factible. Las restricciones del tipo (9), (10) y (11) son el resultado de aplicar la linealización tradicional al conjunto (4) de restricciones del modelo cuadrático, y sirven para especificar el conjunto de aristas en función de las posiciones relativas de los vértices $u, v \in E$ en el ordenamiento lineal.

La restricción definida en (12) permite determinar el conjunto de vértices en el lado izquierdo del punto de corte $c = \lfloor n \,/\, 2 \rfloor$, que están conectados con los vértices del lado derecho en dicho punto de corte. El valor obtenido por la restricción (12) es la bisección de vértices del grafo *G*, el cual se intenta minimizar en la función objetivo del modelo (6).

5.1.4 Modelo de programación lineal entera compactado

El modelo de programación lineal entera compactado (ILP2) se define utilizando las variables x_u^p, $y_{u,v}^{p,q}$ y z_p y la técnica de linealización propuesta por Leo Liberti.

$$\min \sum_{p=1}^{\lfloor n/2 \rfloor} z_p \tag{13}$$

Esta función está sujeta a las siguientes restricciones:

$$\sum_{p \in \{1,\dots,n\}} x_u^p = 1 \qquad \forall u = 1, \dots, n \tag{14}$$

$$\sum_{u \in \{1,\dots,n\}} x_u^p = 1 \qquad \forall p = 1, \dots, n \tag{15}$$

$$\sum_{p=1}^{n} y_{u,v}^{p,q} = x_v^q \qquad \forall (u,v) \in E \vee (v,u) \in E, q = 1, \dots, n \tag{16}$$

$$y_{u,v}^{p,q} = y_{v,u}^{q,p} \qquad \forall (u,v) \in E \vee (v,u) \in E, p, q = 1, \dots, n \tag{17}$$

$$z_p \leq \sum_{q=\lfloor n/2 \rfloor + 1}^{n} \sum_{u=1}^{n} \sum_{v=1}^{n} y_{u,v}^{p,q} \leq (\lfloor n \,/\, 2 \rfloor) z_p \qquad \forall p = 1, \dots, \lfloor n \,/\, 2 \rfloor \tag{18}$$

Las restricciones del tipo (14) y (15) garantizan que a cada posición le corresponde un nodo y a cada nodo le corresponde una posición respectivamente. Estos conjuntos de restricciones aseguran que el ordenamiento lineal del grafo sea factible.

Las restricciones del tipo (16) y (17) son el resultado de aplicar la linealización propuesta por Leo Liberti al conjunto (4) de restricciones del modelo cuadrático y sirven para especificar el conjunto de aristas en función de las posiciones relativas de los vértices $u, v \in E$ en el ordenamiento lineal.

La restricción definida en (18) permite determinar el conjunto de vértices en el lado izquierdo del punto de corte $c = \lfloor n \,/\, 2 \rfloor$, que están conectados con los vértices del lado derecho en dicho punto de corte. El valor obtenido por la restricción (18) es la bisección de vértices del grafo *G*, el cual se intenta minimizar en la función objetivo del modelo (13).

5.1.5 Comparación entre los modelos

Debido a que los 3 modelos propuestos son técnicas de solución exacta donde no se incluyen elementos aleatorizados, es posible realizar un análisis de complejidad de los 3 modelos, observando el número de restricciones y de variables que genera cada modelo.

Modelo	Restricciones	Variables
Cuadrático	$\|E\|n^2 + 2n + 2\lfloor n/2 \rfloor$	${n^4}/{2} + (3\|E\| + 2)n^2 + n$
ILP1	$3\|E\|n^2 + 2n + 2\lfloor n/2 \rfloor$	${n^4}/{2} + (5\|E\| + 2)n^2 + n$
ILP2	$\|E\|n^2 + (\|E\| + 2)n + 2\lfloor n/2 \rfloor$	${n^4}/{2} + (3\|E\| + 2)n^2 + (\|E\| + 1)n$

Donde $|E|$ es la cardinalidad del conjunto de aristas es decir la cantidad de aristas en el grafo, n es la cantidad de nodos que tiene el grafo. Debido a que todos los modelos presentan un elemento de segundo orden en las restricciones ($o(n^2)$) y un elemento de cuarto orden en las variables ($o(n^4)$), cuán complejo es un modelo respecto de otro será definido por una constante de proporcionalidad. En caso de que los elementos del mismo orden tengan la misma constante, se buscará en el orden inmediato menor.

- El coeficiente que acompaña a la n^2 en el modelo cuadrático es menor, tanto en las restricciones como en las variables, con respecto al ILP1.
- El coeficiente que acompaña a la n en el modelo cuadrático es menor, tanto en las restricciones como en las variables, con respecto al ILPL2.

Debido a lo anterior se observó que el modelo que genera menos restricciones y variables es el modelo cuadrático, seguido por el modelo de programación lineal 2, denominado modelo compactado, y por último el modelo de programación lineal 1. Se hizo una comparación para ver cuántas restricciones adicionales generaban los modelos de programación lineal con respecto al modelo cuadrático propuesto.

ILP1 vs Cuadrático	$\{3\|E\|\,n^2 + 2n + 2\lfloor n/2 \rfloor\} - \{\|E\|\,n^2 + 2n + 2\lfloor n/2 \rfloor\} = 2\|E\|n^2$
ILP2 vs Cuadrático	$\{\|E\|\,n^2 + (\|E\| + 2)n + 2\lfloor n/2 \rfloor\} - \{\|E\|\,n^2 + 2n + 2\lfloor n/2 \rfloor\} = \|E\|n$

De la misma forma se hizo una comparación pero de las variables que generaban de más los modelos de programación lineal en comparación del modelo cuadrático.

ILP1 vs Cuadrático	$\{n^4/2 + (5\|E\| + 2)n^2 + n\} - \{n^4/2 + (3\|E\| + 2)\,n^2 + n\} = 2\|E\|\,n^2$
ILP2 vs Cuadrático	$\{n^4/2 + (3\|E\| + 2)\,n^2 + (\|E\| + 1)n\} - \{n^4/2 + (3\|E\| + 2)\,n^2 + n\} = \|E\|n$

Debido a lo anterior se puede concluir que en el mismo tiempo de ejecución los resultados que se obtienen utilizando el ILP2 serán mejores que los obtenidos por el ILP1, ya que en el mismo tiempo, recorrerá una sección más grande del conjunto de soluciones debido a que las variables y las restricciones que tiene que revisar son menos. De igual forma el modelo de programación cuadrática genera aún menos restricciones y variables con lo cual se puede anticipar que tiene mejores resultados.

En la figura 5.1 se muestra el número de restricciones y de variables que genera el modelo cuadrático para 4 instancias diferentes. La primera de 18 nodos (*n*) y 38 aristas (*e*), la segunda de 18 *n* y 40 *e*, la tercera en 20 *n* y 38 *e* y la última de 20 *n* y 40 *e*.

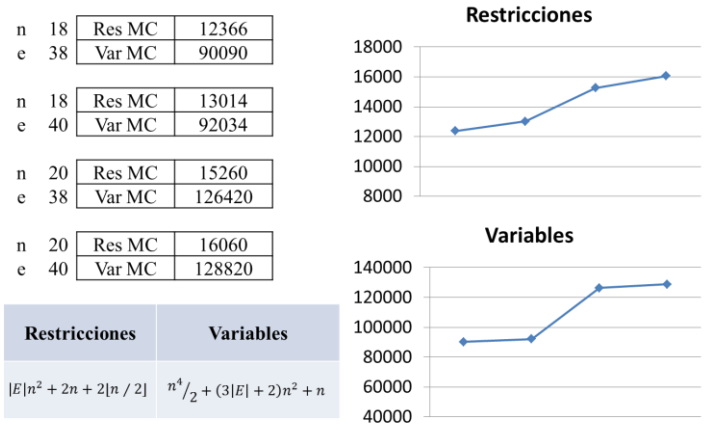
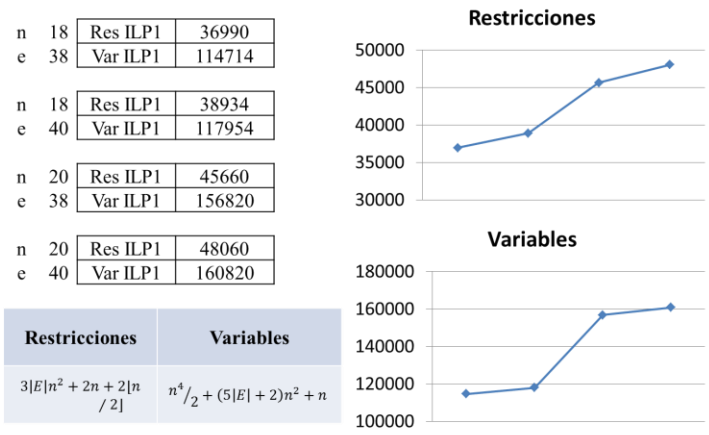


Figura 5.1) Restricciones y variables del modelo cuadrático

En la figura 5.2 se muestra el número de restricciones y de variables que genera ILP1 para las mismas 4 instancias descritas anteriormente.

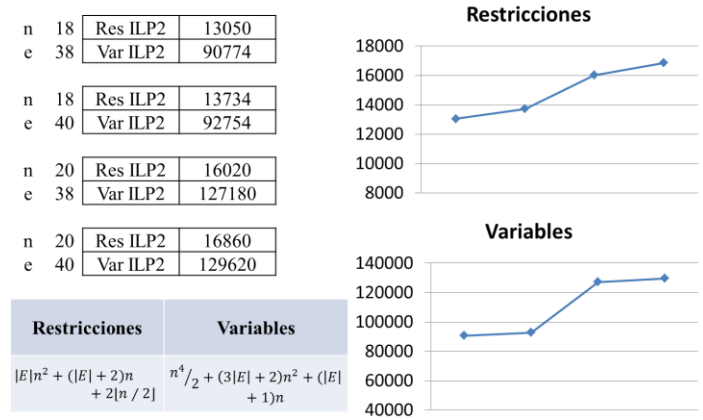


n	18	Res ILP1	36990
e	38	Var ILP1	114714
n	18	Res ILP1	38934
e	40	Var ILP1	117954
n	20	Res ILP1	45660
e	38	Var ILP1	156820
n	20	Res ILP1	48060
e	40	Var ILP1	160820

Restricciones	Variables
$3\lvert E\rvert n^2 + 2n + 2\lfloor n/2 \rfloor$	$n^4/_2 + (5\lvert E\rvert + 2)n^2 + n$

Figura 5.2) Restricciones y variables del ILP1

En la figura 5.3 se muestra el número de restricciones y de variables que genera ILP2 para las mismas 4 instancias descritas anteriormente.



n	18	Res ILP2	13050
e	38	Var ILP2	90774
n	18	Res ILP2	13734
e	40	Var ILP2	92754
n	20	Res ILP2	16020
e	38	Var ILP2	127180
n	20	Res ILP2	16860
e	40	Var ILP2	129620

Restricciones	Variables
$\lvert E\rvert n^2 + (\lvert E\rvert + 2)n + 2\lfloor n/2 \rfloor$	$n^4/_2 + (3\lvert E\rvert + 2)n^2 + (\lvert E\rvert + 1)n$

Figura 5.3) Restricciones y variables del ILP2

5.2 Método B&B basado en permutaciones

Este método de solución se desarrolló aprovechando que al generar el subconjunto *L*, es decir los nodos que están del lado izquierdo del punto de corte (véase sección 4), el valor objetivo de la permutación es independiente de cómo se acomoden los nodos restantes (subconjunto *R*). Por lo cual sólo es necesario generar las permutaciones del subconjunto *L*. Con lo que se logra reducir el espacio de soluciones (*n*!) que se deben explorar y evaluar para garantizar la solución óptima en:

$$Permutaciones\_L = n!/\lfloor n \,/\, 2 \rfloor!$$

Una vez leída la instancia, como es un problema de minimizar se inicializa al *mejor valor* con el máximo valor posible, se propone que este valor inicialmente sea igual a $\lfloor n \,/\, 2 \rfloor$, lo cual significaría que todos los nodos de la izquierda están conectados con al menos un nodo de la derecha. Mientras existan permutaciones sin explorar y el tiempo de ejecución no haya acabado se realizará un ciclo donde se generarán las permutaciones de *L*.

Este algoritmo generará solamente el conjunto izquierdo. Después de que se genere una nueva solución actual se tendrá que evaluar parcialmente, nodo por nodo, cuántos elementos del subconjunto izquierdo tienen conexión con el subconjunto derecho. Si durante la evaluación parcial la solución empeora con respecto al mejor valor actual, se detendrá la evaluación y se desechará esta solución.

Si la solución actual mejora el óptimo, será necesario actualizar la solución, grabándola en un arreglo temporal y el mejor valor (de aquella permutación que tenga la menor bisección de vértices) también se actualizará con el valor actual, con el fin de que en próximas permutaciones se compare contra este nuevo valor. Después se regresará al nodo anterior más próximo y se explorará otra permutación, hasta agotar los nodos posibles a explorar. Cuando el ciclo se detenga, se generará una solución final con la mejor solución encontrada que se está almacenando en el arreglo temporal al finalizar el ciclo y los $n - \lfloor n \,/\, 2 \rfloor$ elementos restantes se rellenarán con elementos no tomados por la mejor permutación en cualquier orden, recordando que no importa el orden de los elementos restantes.

Algoritmo Branch and Bound (B&B) por permutaciones

Función del algoritmo: Genera y evalúa las permutaciones del subconjunto *L*.

Entradas:

n: número de vértices que tiene la instancia

E: conjunto de aristas que tiene la instancia

Estructuras:

mejorVal: variable que se actualiza con la menor bisección de vértices encontrada durante el proceso del B&B.

existenPermutaciones: variable booleana que indica si aún existen permutaciones por explorar (*true*) o si ya se exploraron todas las permutaciones (*false*).

tiempoDeEjecucion: variable booleana que indica si el tiempo de ejecución del algoritmo se ha agotado (*true*).

perm: arreglo donde se va construyendo la permutación de los elementos de *L*

soluciónActual: variable donde se almacena el valor objetivo de la permutación actual.

Funciones:

Generación_estructuras(): función que genera la pila de elementos *S*, así como el resto de las estructuras que ocupa el algoritmo.

toma_de_S(): esta función toma un elemento de *S* para utilizarlo en la permutación actual.

insertar_en_S(perm[i]): toma el último elemento de la permutación y lo agrega a la pila *S*.

Evalua(perm): función que evalúa la solución actual.

Actualizar_solución(): esta función graba la permutación actual en caso de que haya mejorado la solución.

valorActual: variable donde se guarda el valor de la bisección de vértices de la permutación que se está explorando.

Regresa_al_nodo_anterior_para_explorar(): función que verifica si es posible regresar al nodo y seguir construyendo el resto de las permutaciones, en caso de que no sea posible detiene la ejecución del algoritmo.

Generar_solución_final(): función que graba el subconjunto *L* y genera el resto de la permutación rellenando el subconjunto *R*, con los nodos que no hayan sido utilizados.

Imprimir(soluciónFinal, mejorVal): función que devuelve el resultado encontrado por el algoritmo.

Salida:

soluciónFinal: devuelve la mejor permutación encontrada.

mejorVal: función objetivo asociada a la mejor permutación.

```
1   Generación_estructuras();
2   mejorVal = ⌊n / 2⌋;
3   while (existenPermutaciones == True and tiempoDeEjecucion == True)
4    {
5       permuta (i) {
6           for (i = 0; i < nodos – i + 1; i++) {
7               perm [i] = toma_de_S();
8               if ( i == ⌊n / 2⌋) {
9                   evaluación_parcial(perm);
10              }
11              if ( i < ⌊n / 2⌋)  {
12                  perm([i + 1];
13              }
14              insertar_en_S(perm[i]);
15          }
16      }
```

```
17      soluciónActual = Evalua(perm);
18      if ( soluciónActual  <  mejorVal)  {
19          Actualizar_solución();
20          mejorVal  = valorActual;
21      }
22      Regresa_al_nodo_anterior_para_explorar();
23      if ( nodosPorExplorar == False)  {
24          existenPermutaciones  = False;
25      }
26  }
27  Generar_solución_final();
28  Imprimir(soluciónFinal, mejorVal);
```

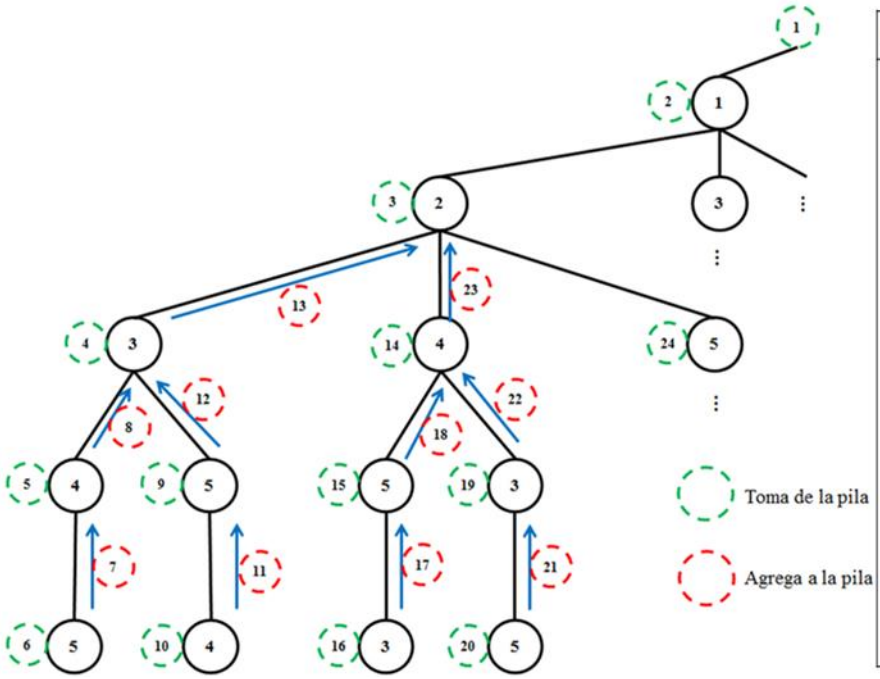
Algoritmo 2) B&B de permutaciones

5.2.1 Ejemplo del proceso para generar permutaciones

Para generar las permutaciones se utiliza una pila (*S*), en la cual al iniciar se le agregan todos los nodos del grafo. En cada nivel se toma un elemento de la pila para generar el siguiente nivel del árbol de permutaciones, en la figura 5.4 se representa este proceso con círculos verdes.

Una vez que se agotan los elementos de la pila, se vuelven a agregar los nodos a la pila y se escala el árbol de permutaciones, hasta que se encuentre un nodo donde se pueda volver a expandir el árbol, en la figura 5.4 se representa este proceso con círculos rojos.

En la figura 5.4 se crea una sección del árbol de permutaciones de 5 nodos, utilizando el algoritmo antes descrito, la figura incluye una tabla, donde se describen paso a paso cuáles elementos constituyen la pila en ese instante.



Iteración	Elementos en la pila S
1	1, 2, 3, 4, 5
2	2, 3, 4, 5
3	3, 4, 5
4	4, 5
5	5
6	0
7	5
8	5, 4
9	4
10	0
11	4
12	4, 5
13	4, 5, 3
14	5, 3
15	3
16	0
17	3
18	3, 5
19	5
20	0
21	5
22	5, 3
23	5, 3, 4
24	3, 4
⋮	⋮

Figura 5.4) Construcción de las permutaciones

5.2.2 Ejemplo del B&B utilizando permutaciones

Sea un grafo con 4 nodos y 3 aristas como se muestra a continuación figura 5.5:

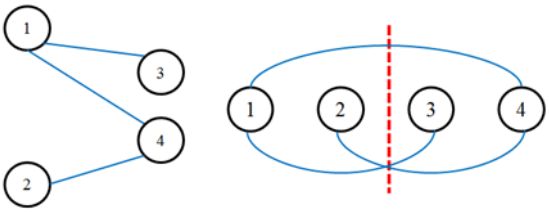


Figura 5.5) Grafo de 4 nodos para ejemplificar el B&B

Este grafo se resolverá utilizando el método B&B para permutaciones el cual tendrá que evaluar todas las permutaciones posibles de los nodos del subconjunto *L*. En la tabla 5.1, se observan los resultados obtenidos para cada una de las permutaciones generadas por este algoritmo y se marca en color oscuro el resultado que se devolvió como óptimo.

Tabla 5.1) Solución de una instancia por el B&B

Nodo 1	Nodo 2	Nodo 3	Nodo 4	Permutación *L*	Valor Objetivo
Conjunto *L*		Conjunto *R*			
1	2	3	4	1, 2, 3, 4	2
	3	2	4	1, 3, 2, 4	1
	4	2	3	1, 4, 2, 3	2
2	1	3	4	2, 1, 3, 4	2
	3	1	4	2, 3, 1, 4	2
	4	1	3	2, 4, 1, 3	1
3	1	2	4	3, 1, 2, 4	1
	2	1	4	3, 2, 1, 4	2
	4	1	2	3, 4, 1, 2	2
4	1	2	3	4, 1, 2, 3	2
	2	1	3	4, 2, 1, 3	1
	3	1	2	4, 3, 1, 2	2

5.3 Método enumerativo para combinaciones

El método enumerativo por combinaciones es un método exacto de solución para el problema de bisección de vértices. Este método está diseñado para aprovechar el hecho de que al tomar dos permutaciones diferentes, en las cuales no se hayan hecho movimientos que involucren cambios en el ordenamiento de los vértices entre los conjuntos *L* y *R*, el valor objetivo de ambos no cambia.

Entonces, se puede plantear el problema de la bisección de vértices como un problema de combinaciones, no de permutaciones, es decir que el espacio de soluciones se reduce de $n!$ hasta la combinación de $\lfloor n \,/\, 2 \rfloor$ en n.

$$nC_{\lfloor n/2 \rfloor} = \frac{n!}{(n - \lfloor n \,/\, 2 \rfloor)!\ \ \lfloor n \,/\, 2 \rfloor!}$$

Lo anterior reduce las soluciones que se tienen que evaluar para garantizar la solución óptima, por lo que se obtienen mejores resultados como se verá en la sección de experimentación. El método funciona de la siguiente manera:

Una vez leída la instancia, como es un problema de minimizar se inicializa la variable *mejorVal* con el máximo posible valor. Mientras existan combinaciones sin explorar y el tiempo de ejecución no haya acabado se realizará un ciclo donde se generarán combinaciones de $\lfloor n \,/\, 2 \rfloor$ elementos.

Estas combinaciones crearán sólo el conjunto izquierdo. Después de generar una nueva combinación se tendrá que evaluar cuántos de los nodos del conjunto izquierdo, tienen conexión con alguno de los nodos que no se encuentren en este arreglo (los cuales formarán el lado derecho).

Si la solución actual mejora el óptimo actual, será necesario actualizar la solución grabándola en un arreglo temporal y la variable *mejorVal* también se actualizará con el valor actual, con el fin de que en próximas combinaciones se compare contra estos nuevos valores. Este algoritmo se detiene cuando no queden combinaciones por explorar o cuando el tiempo de ejecución finalice.

Cuando el ciclo se detenga se generará una solución final con la mejor solución encontrada que será almacenada en el arreglo temporal al finalizar el ciclo y los $n - \lfloor n \,/\, 2 \rfloor$ elementos restantes se rellenarán con elementos no tomados por la mejor combinación en cualquier orden, recordando que no importa el orden de los elementos restantes.

Algoritmo enumerativo por combinaciones

Función del algoritmo: Genera y evalúa las combinaciones del subconjunto *L*.

Entradas:

n: número de vértices que tiene la instancia

E: conjunto de aristas que tiene la instancia

Estructuras:

mejorVal: variable que se actualiza con la menor bisección de vértices encontrada durante el proceso del algoritmo.

existenCombinaciones: variable booleana que indica si aún existen combinaciones por explorar (*true*) o si ya se exploraron todas las

combinaciones (*false*).

tiempoDeEjecucion: variable booleana que indica si el tiempo de ejecución del algoritmo se ha agotado (*true*).

comb: arreglo donde se va construyendo la combinación de los elementos de *L*

valorActual: variable donde se almacena el valor objetivo de la combinación actual.

Funciones:

Generación_estructuras(): función que genera las estructuras que ocupa el algoritmo, como el arreglo donde se almacenarán las combinaciones de $\lfloor n / 2 \rfloor$ elementos.

Generador_de_combinaciones(): en esta función se genera la combinación que se va a evaluar actualmente y se verifica que existan combinaciones por explorar, en caso de que se acaben las combinaciones detiene el algoritmo.

Evalua(comb): función que evalúa la solución actual.

Actualizar_solución(comb): esta función graba la combinación actual en caso de que haya mejorado la solución.

Generar_solución_final(): función que graba el subconjunto *L* y genera el resto de la combinación rellenando el subconjunto *R*, con los nodos que no hayan sido utilizados.

Imprimir(soluciónFinal, mejorVal): función que devuelve el resultado encontrado por el algoritmo.

Salida:

soluciónFinal: devuelve la mejor combinación encontrada.

mejorVal: función objetivo asociada a la mejor combinación.

```
1   Generación_estructuras();
2   mejorVal = ⌊n / 2⌋;
3   while (existenCombinaciones and tiempoDeEjecucion == True)  {
4       Generador_de_combinaciones();
```

```
5       valorActual = Evalua(comb);
6       if ( valorActual  <  mejorVal)  {
7           Actualizar_solución(comb);
8           mejorVal  =  valorActual;
9       }
10  }
11  Generar_solución_final(comb);
12  Imprimir(soluciónFinal, mejorVal);
```

Algoritmo 3) Enumerativo por combinaciones

5.3.1 Método generador de combinaciones

El método generador de combinaciones utiliza los conjuntos iniciales $s = \{v_1, v_2, v_3, \dots, v_{\lfloor n/2 \rfloor}\}$ y su complemento $s^c = \{v_{\lfloor n/2 \rfloor+1}, v_{\lfloor n/2 \rfloor+2}, v_{\lfloor n/2 \rfloor+3}, \dots, v_n\}$ los cuales contienen a todos los nodos de la instancia en orden del nodo 1 hasta $\lfloor n / 2 \rfloor$ y desde $\lfloor n / 2 \rfloor + 1$ hasta *n* respectivamente. También utiliza dos conjuntos de cardinalidad variable: *x* que tomará elementos de *s* y el conjunto *y* que tomará elementos de s^c.

Método generador de combinaciones

Función del método: Genera las permutaciones del subconjunto *L*.

Entradas:

n: número de vértices que tiene la instancia

Conjuntos utilizados:

s: conjunto que se carga con los primeros $\lfloor n / 2 \rfloor$ nodos de la instancia.

s^c: conjunto complemento de *s*, se carga con los nodos restantes n - $\lfloor n / 2 \rfloor$.

x: conjunto de cardinalidad variable (*i*) que tomará nodos de *s*.

y: conjunto de cardinalidad variable (*i*) que tomará nodos de s^c.

A: conjunto donde se almacenarán temporalmente los elementos restantes de *s*, después de que hayan sido tomado los elementos de *x*.

Salida:

B: conjunto de elementos que regresa el método que consiste en los nodos que restaron en *A* con los nodos que se tomaron del conjunto s^c (*y*).

1 $|s| = \lfloor n \,/\, 2 \rfloor$
2 $|s^c| = n - \lfloor n \,/\, 2 \rfloor$
// Se define el cantidad de elementos que tiene cada uno de los conjuntos, *s* con $\lfloor n \,/\, 2 \rfloor$ elementos y s^c con los elementos restantes.
3 **for** $i = 0$ to $\lfloor n \,/\, 2 \rfloor$ {
4 $\forall\, x \subseteq s$ tal que $|x| = i$
5 $A = s - x$
6 $\forall\, y \subseteq s^c$ tal que $|y| = i$
7 $B = A \cup y$
8 }

Algoritmo 4) Método generador de combinaciones

En cada iteración del ciclo la cardinalidad de los conjunto *x* y *y*, aumenta respecto a una variable *i*, después se carga *x* con todos los posibles subconjuntos que contienen *i* elementos tomados de *s* y cada uno de esos posibles subconjuntos serán remplazados con la misma cantidad de elementos de todos los posibles subconjuntos de s^c.

Se empieza tomando y remplazando 0 elementos (i = 0) de s^c, con lo que se garantiza que la permutación $\varphi_1 = \{v_1, v_2, v_3, \dots, v_{\lfloor n/2 \rfloor}\}$ también sea explorada. Después se incrementa la cardinalidad (*i*) de *x* y de *y* de tal forma que se toman todos los subconjuntos de 1 elemento (i = 1) y se sustituyen por todos los subconjuntos de 1 elemento. Este proceso se repite para dos elementos (i = 2) y así consecutivamente.

En el subconjunto *A*, se almacenan los elementos de *s* después de que se le extrajeron los elementos de *x*. El subconjunto B se almacena la unión de los elementos de *A* con los elementos de *y*, que son los nuevos elementos tomados de s^c.

Este método genera todas las posibles combinaciones de $\lfloor n \,/\, 2 \rfloor$ en n, con la finalidad de encontrar el valor óptimo en un espacio de soluciones reducido. En el ejemplo mostrado en la tabla 5.2 se crearon las combinaciones de 6 nodos en 3 posiciones.

$$6C_3 = \frac{6!}{(6-3)!\ \ 3!} = \frac{720}{36} = 20$$

Tabla 5.2) Ejemplo del generador de permutaciones

		s			s^c		
		1	**2**	**3**	**4**	**5**	**6**
Combinaciones a Evaluar	***i*=0**	**1**	**2**	**3**	**4**	**5**	**6**
	***i*=1**	**1**	**2**	**4**	**3**	**5**	**6**
		1	**2**	**5**	**4**	**3**	**6**
		1	**2**	**6**	**4**	**5**	**3**
		1	**4**	**3**	**2**	**5**	**6**
		1	**5**	**3**	**4**	**2**	**6**
		1	**6**	**3**	**4**	**5**	**2**
		4	**2**	**3**	**1**	**5**	**6**
		5	**2**	**3**	**4**	**1**	**6**
		6	**2**	**3**	**4**	**5**	**1**
	***i*=2**	**1**	**4**	**5**	**2**	**3**	**6**
		1	**4**	**6**	**2**	**5**	**3**
		1	**5**	**6**	**4**	**2**	**3**
		4	**2**	**5**	**1**	**3**	**6**
		4	**2**	**6**	**1**	**5**	**3**
		5	**2**	**6**	**4**	**1**	**3**
		4	**5**	**3**	**1**	**2**	**6**
		4	**6**	**3**	**1**	**5**	**2**
		5	**6**	**3**	**4**	**1**	**2**
	***i*=3**	**4**	**5**	**6**	**1**	**2**	**3**

5.3.2 Ejemplo del método enumerativo para combinaciones

Sea un grafo con 4 nodos y 3 aristas como se muestra a continuación en la figura 5.6:

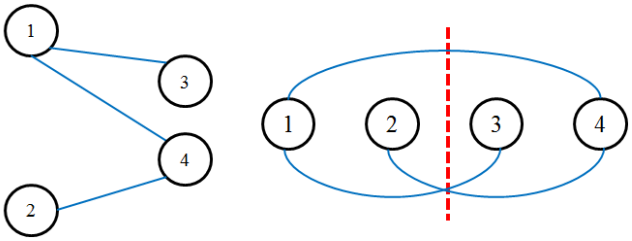


Figura 5.6) Grafo de 4 nodos para ejemplificar el método enumerativo para combinaciones

Este grafo se resolverá utilizando el método enumerativo para combinaciones, el cual tendrá que evaluar todas las combinaciones posibles de los nodos del subconjunto *L*. En la tabla 5.3, se observan los resultados obtenidos para cada una de las combinaciones generadas por este algoritmo y se resalta en oscuro el resultado que se devolvió como óptimo.

Tabla 5.3) Solución de una instancia por el enumerativo basado en combinaciones

Nodo 1	**Nodo 2**	**Nodo 3**	**Nodo 4**	**Combinación *L***	**Valor Objetivo**
Conjunto *L*		**Conjunto *R***			
1	**2**	**3**	**4**	**1, 2, 3, 4**	**2**
	3	**2**	**4**	**1, 3, 2, 4**	**1**
	4	**2**	**3**	**1, 4, 2, 3**	**2**
2	**3**	**1**	**4**	**2, 3, 1, 4**	**2**
	4	**1**	**3**	**2, 4, 1, 3**	**1**
3	**4**	**1**	**2**	**3, 4, 1, 2**	**2**

Capítulo 6 Resultados experimentales

Toda la experimentación se realizó en una computadora portátil con las siguientes características:

- Sistema operativo: Windows 7, service pack 1.
- Procesador Intel inside core i7.
- Frecuencia del procesador: 2.90 GHz
- Memoria RAM: 8 GB.
- Sistema operativo de 64 bits.

Todos los algoritmos se programaron utilizando C#. Para plantear y resolver los modelos de programación lineal entera se utilizó el optimizador CPLEX versión 10.9.

Para las instancias del conjunto Grid, Small y Tree, el tiempo de experimentación se delimitó a 5 minutos por instancia, mientras que en las instancias del conjunto Harwell – Boeing (H-B) se delimitó el tiempo a 1 hora por instancia.

6.1 Instancias

Para la experimentación reportada, se utilizaron las siguientes instancias (tabla 6.1) tomadas del conjunto para problemas de ordenamiento lineal [Duarte 12, Pantrigo 10]:

Tabla 6.1) Instancias utilizadas en la experimentación

Small	p17_16_24_n	p38_18_19_n	p59_20_23_n	p80_22_30_n
	p18_16_21_n	p39_18_19_n	p60_20_22_n	p81_23_46_n
	p19_16_19_n	p40_18_32_n	p61_21_22_n	p82_23_24_n
	p20_16_18_n	p41_19_20_n	p62_21_30_n	p83_23_24_n
	p21_17_20_n	p42_19_24_n	p63_21_42_n	p84_23_26_n
	p22_17_19_n	p43_19_22_n	p64_21_22_n	p85_23_26_n
	p23_17_23_n	p44_19_25_n	p65_21_24_n	p86_23_24_n
	p24_17_29_n	p45_19_25_n	p66_21_28_n	p87_23_30_n
	p25_17_20_n	p46_19_20_n	p67_21_22_n	p88_23_26_n
	p26_17_19_n	p47_19_21_n	p68_21_27_n	p89_23_27_n
	p27_17_19_n	p48_19_21_n	p69_21_23_n	p90_23_35_n
	p28_17_18_n	p49_19_22_n	p70_21_25_n	p91_24_33_n
	p29_17_18_n	p50_19_25_n	p71_22_29_n	p92_24_26_n
	p30_17_19_n	p51_20_28_n	p72_22_49_n	p93_24_27_n
	p31_18_21_n	p52_20_27_n	p73_22_29_n	p94_24_31_n
	p32_18_20_n	p53_20_22_n	p74_22_30_n	p95_24_27_n
	p33_18_21_n	p54_20_28_n	p75_22_25_n	p96_24_27_n
	p34_18_21_n	p55_20_24_n	p76_22_30_n	p97_24_26_n
	p35_18_19_n	p56_20_23_n	p77_22_37_n	p98_24_29_n
	p36_18_20_n	p57_20_24_n	p78_22_31_n	p99_24_27_n
	p37_18_20_n	p58_20_21_n	p79_22_29_n	p100_24_34_n
Tree	TREE_22_3_rot1	TREE_22_3_rot5	TREE_22_3_rot4	TREE_22_3_rot3
	TREE_22_3_rot2	TREE_22_3_rot1	TREE_22_3_rot5	TREE_22_3_rot4
	TREE_22_3_rot3	TREE_22_3_rot2	TREE_22_3_rot1	TREE_22_3_rot5
	TREE_22_3_rot4	TREE_22_3_rot3	TREE_22_3_rot2	
Grid	Grid3x3.rnd	Grid4x4.rnd	grid_5.rnd	grid_6.rnd
	grid_7.rnd			
H-B	bcspwr01.rnd	bcsstk01.rnd	bcspwr02.rnd	can___24.rnd

6.2 Resultados

En caso de que el algoritmo resolviera la instancia en menor tiempo del que se designó como restricción, se garantiza que el resultado obtenido es óptimo debido a que los cuatro algoritmos tienen un enfoque exhaustivo, lo cual significa que recorre todo el espacio de soluciones y encontró la solución óptima. Puede que el resultado obtenido en algunas instancias donde se detuvo por el tiempo de ejecución sea el óptimo, pero no es posible garantizarlo debido a que no se exploró todo el espacio de soluciones.

Tabla 6.2) Resultados finales de la experimentación

Método	Instancias	Tiempo de solución acumulado	Promedio soluciones	Cantidad óptimos	Porcentaje óptimos
ILP1	108	38,803.40	3.4191	20	18.52%
ILP2	108	16,161.33	2.3939	99	91.67%
B&B Permutaciones	108	49,434.18	2.5869	3	2.78%
Enumerativo Combinaciones	108	11,283.23	2.2964	103	95.37%

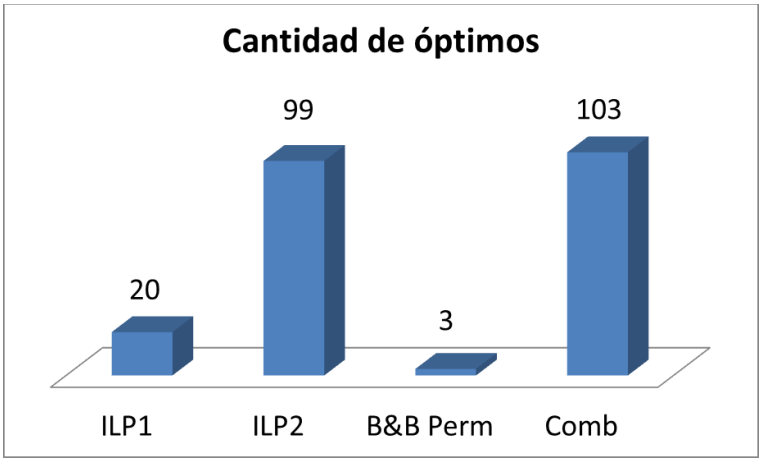


Figura 6.1) Óptimos encontrados

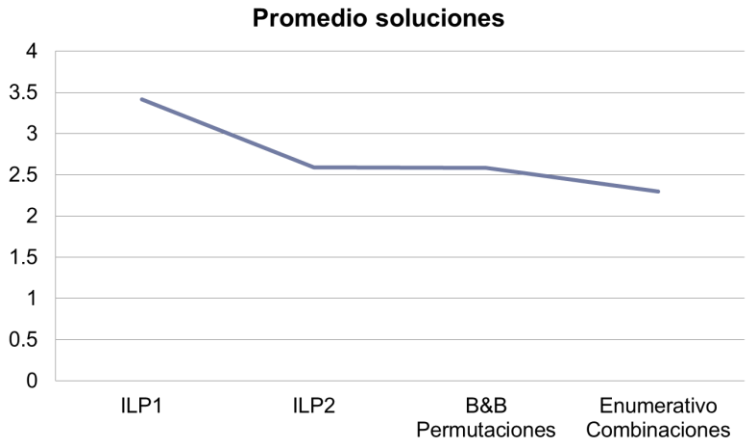


Figura 6.2) Promedio de soluciones

6.3 Análisis de los resultados

Las diferencias observadas en los resultados para los dos modelos de programación lineal entera se deben a que el segundo modelo (ILP2) genera menos restricciones en su linealización, lo cual le permite en el mismo tiempo explorar una sección más amplia del espacio de soluciones con respecto al primer modelo.

El método B&B para permutaciones, obtiene los promedios de soluciones menores a los obtenidos por los modelos matemáticos, algunos resultados obtenidos coincidieron con los óptimos de algunas instancias encontradas por los otros métodos, aunque por cuestiones del tiempo de ejecución no se puede garantizar que estas soluciones sean óptimas.

Los mejores resultados obtenidos fueron por parte del método enumerativo para combinaciones, lo cual se debe a que el espacio de soluciones que revisa es menor en comparación de los otros métodos. Este algoritmo es eficaz para el problema de un grafo conexo no dirigido, pero este enfoque no es óptimo para otros casos, por ejemplo con grafos dirigidos.

El construir algoritmos eficientes que resuelvan el problema de la bisección de vértices, sirve para poder distribuir mensajes de manera adecuada en una red, por medio del proceso de *gossip* antes descrito. A medida que se pueda encontrar el mejor ordenamiento de los nodos de un grafo, es posible mejorar la comunicación en la red.

Capítulo 7 Conclusiones y trabajos futuros

Durante el desarrollo del proyecto se alcanzaron satisfactoriamente todos los objetivos propuestos en el trabajo, se implementaron 4 algoritmos exactos para el problema de la bisección de vértices de un grafo conexo no dirigido, que resuelven instancias con una cantidad de nodos pares e impares.

Las dos principales contribuciones de este trabajo son las siguientes:

a) Un modelo de programación lineal (ILP2), que utiliza una técnica de linealización que permite reducir el número de restricciones y variables del modelo de programación lineal ILP1. Para una instancia determinada el modelo generado con ILP2 es más pequeño que el generado por ILP1, logrando una ventaja significativa cuando se resuelven con CPLEX. (Ver la sección 5.1.4)

b) Un algoritmo enumerativo basado en combinaciones. Este algoritmo construye combinaciones de $\lfloor n / 2 \rfloor$ elementos para el subconjunto izquierdo y evalúa estas combinaciones de tal forma que se reduce considerablemente el espacio de soluciones que se requiere explorar para encontrar la solución óptima. Este algoritmo se construyó observando que las permutaciones que involucran cambios de elementos entre los mismos conjuntos no generan diferentes valores objetivos, por lo cual se decidió explorar sólo el espacio de combinaciones. (Ver la sección 5.3)

Se propone como trabajo futuro, realizar algoritmos del tipo heurísticas o metaheurísticas que aprovechen la estructura del problema para resolverlo el problema con un mayor grado de precisión y que pueda resolver instancias más grandes que las resueltas por los métodos exactos aquí propuestos.

Anexo A: Óptimos encontrados

En la siguiente sección se muestran los resultados óptimos que devolvieron los 4 métodos de solución propuestos, agrupados por tipos de instancias.

Tabla Anexo A.1) Óptimos de las instancias grid

Conjunto Grid	ILP1	ILP2	B&B	Enumerativo
Grid3x3.mtx.rnd	3	3	3	3
Grid4x4.mtx.rnd		4		4
grid_5.mtx.rnd				5
grid_6.mtx.rnd				
grid_7.mtx.rnd				

Tabla Anexo A.2) Óptimos de las instancias Harwell - Boeing

Harwell - Boeing	ILP1	ILP2	B&B	Enumerativo
bcspwr01.mtx.rnd		3		
bcspwr02.mtx.rnd		2		
bcsstk01.mtx.rnd				
can___24.mtx.rnd		4		4

Tabla Anexo A.3) Óptimos de las instancias tree

Tree	ILP1	ILP2	B&B	Enumerativo
TREE_22_3_rot1.mtx		2		2
TREE_22_3_rot2.mtx		2		2
TREE_22_3_rot3.mtx		2		2
TREE_22_3_rot4.mtx		2		2
TREE_22_3_rot5.mtx		2		2
TREE_22_3_rot1.mtx		2		2
TREE_22_3_rot2.mtx		2		2

TREE_22_3_rot3.mtx		2		2
TREE_22_3_rot4.mtx		2		2
TREE_22_3_rot5.mtx		2		2
TREE_22_3_rot1.mtx		2		2
TREE_22_3_rot2.mtx		2		2
TREE_22_3_rot3.mtx		2		2
TREE_22_3_rot4.mtx		2		2
TREE_22_3_rot5.mtx		2		2

Tabla Anexo A.4) Óptimos de las instancias small

Small	ILP1	ILP2	B&B	Enumerativo
p17_16_24_n.txt		3		3
p18_16_21_n.txt	2	2	2	2
p19_16_19_n.txt	2	2	2	2
p20_16_18_n.txt	2	2		2
p21_17_20_n.txt	2	2		2
p22_17_19_n.txt	2	2		2
p23_17_23_n.txt	2	2		2
p24_17_29_n.txt		3		3
p25_17_20_n.txt	2	2		2
p26_17_19_n.txt	2	2		2
p27_17_19_n.txt	2	2		2
p28_17_18_n.txt		2		2
p29_17_18_n.txt	1	1		1
p30_17_19_n.txt	2	2		2
p31_18_21_n.txt		2		2
p32_18_20_n.txt	2	2		2
p33_18_21_n.txt		3		3
p34_18_21_n.txt		2		2
p35_18_19_n.txt		2		2
p36_18_20_n.txt	2	2		2
p37_18_20_n.txt	2	2		2
p38_18_19_n.txt		2		2
p39_18_19_n.txt	2	2		2
p40_18_32_n.txt		4		4

p41_19_20_n.txt	1	1		1
p42_19_24_n.txt		3		3
p43_19_22_n.txt		2		2
p44_19_25_n.txt		3		3
p45_19_25_n.txt		2		2
p46_19_20_n.txt		2		2
p47_19_21_n.txt		2		2
p48_19_21_n.txt		2		2
p49_19_22_n.txt	2	2		2
p50_19_25_n.txt		2		2
p51_20_28_n.txt		4		4
p52_20_27_n.txt		2		2
p53_20_22_n.txt		2		2
p54_20_28_n.txt		3		3
p55_20_24_n.txt		2		2
p56_20_23_n.txt		3		3
p57_20_24_n.txt		2		2
p58_20_21_n.txt		2		2
p59_20_23_n.txt		2		2
p60_20_22_n.txt		2		2
p61_21_22_n.txt		2		2
p62_21_30_n.txt		3		3
p63_21_42_n.txt				5
p64_21_22_n.txt		2		2
p65_21_24_n.txt		2		2
p66_21_28_n.txt		3		3
p67_21_22_n.txt		2		2
p68_21_27_n.txt		3		3
p69_21_23_n.txt		2		2
p70_21_25_n.txt		3		3
p71_22_29_n.txt		3		3
p72_22_49_n.txt				5
p73_22_29_n.txt		2		2
p74_22_30_n.txt		3		3
p75_22_25_n.txt		2		2
p76_22_30_n.txt		2		2

p77_22_37_n.txt				4
p78_22_31_n.txt		3		3
p79_22_29_n.txt		3		3
p80_22_30_n.txt		3		3
p81_23_46_n.txt				6
p82_23_24_n.txt		2		2
p83_23_24_n.txt		1		1
p84_23_26_n.txt		2		2
p85_23_26_n.txt				1
p86_23_24_n.txt		2		2
p87_23_30_n.txt		3		3
p88_23_26_n.txt		2		2
p89_23_27_n.txt		3		3
p90_23_35_n.txt		3		3
p91_24_33_n.txt		3		3
p92_24_26_n.txt		2		2
p93_24_27_n.txt		2		2
p94_24_31_n.txt		3		3
p95_24_27_n.txt		2		2
p96_24_27_n.txt		2		2
p97_24_26_n.txt		2		2
p98_24_29_n.txt		2		2
p99_24_27_n.txt		2		2
p100_24_34_n.txt		3		3

Referencias

1. [Díaz 2000] DÍAZ JOSEP, PETIT JORDI, SERNA MARÍA. A Survey on Graph Layout Problems. Octubre 17 del 2000.
2. [Petit 2002] PETIT JORDI. Addenda to the Survey of Layout Problems. 2002.
3. [Sipser 2006] MICHAEL SIPSER. "Introduction to the Theory of Computation". Editorial Thomson, segunda edición 2006.
4. [Böckenhauer 1999] BÖCKENHAUER HANS-JOACHIM. "Two open problems in communication in edge-disjoint paths modes", Acta Mathematica et Informatica Universitatis Ostraviensis, Vol. 7 (1999), No. 1.
5. [Brandes 2009] BRANDES ULRIK, FLEISCHER DANIEL. "Vertex Bisection is Hard, too", Journal of Graph Algorithms and Applications. 2009
6. [Duarte 2007] DUARTE ABRAHAM, PANTRIGO JUAN JOSÉ, GALLEGO MICAEL. Metaheurísticas. 2007
7. [Liu 2009] XIANGCHANG LIU, Solving the minimum bisection problem using a biologically inspired computational model. 2009
8. [Jerrum 1993] MARK JERRUM, Simulated Anneling for Graph Bisection. 1993
9. [Karisch 2000] STEFAN E KARISCH, Solving graph bisection problem with semidefinite programming. 2000
10. [Hromkovic 1995] JURAJ HROMKOVIC, RALF KLASING, Gossiping in Vertex-disjoin path mode in d-dimensional grids and planar graphs. 1995
11. [Castilla 2013] CASTILLA VALDEZ GUADALUPE, Metaheurísticas aplicadas a la solución del problema de ordenamiento lineal. 2013
12. [Terán 2013] TERÁN VILLANUEVA DAVID, Metaheurísticas aplicadas a la solución del problema de ordenamiento lineal con costos acumulados. 2013