PEDECIBA Informática
Instituto de Computación – Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

Tesis de Maestría
en Informática

Diseño topológico de redes. Caso de estudio: The augmentation Steiner two-node survivable network problem

Martín Bentancourt Alves

2011

Bentancourt Alves, Martín
Diseño topológico de redes. Caso de estudio: The augmentation Steiner two-node survivable network problem
ISSN 0797-6410
Tesis de Maestría en Informática
Reporte Técnico RT 11-17
PEDECIBA
Instituto de Computación – Facultad de Ingeniería
Universidad de la República.
Montevideo, Uruguay, diciembre de 2011

TESIS

a ser presentada el día 5 de Diciembre de 2011 en la

Universidad de la República, UdelaR

para obtener el título de

MAGISTER EN INFORMÁTICA (PEDECIBA)

por

Martín BENTANCOURT ALVES

Instituto de Investigación : LPE - IMERL
Componentes universitarios :
UNIVERSIDAD DE LA REPÚBLICA
FACULTAD DE INGENIERÍA

Título de la tesis :

Diseño Topológico de Redes. Caso de Estudio: The Augmentation Steiner two-node Survivable Network Problem

Presentada el 5 de Diciembre de 2011 ante el comité de examinadores

Dr. Franco	ROBLEDO	Director de Tesis
Msc. Ing. Omar	VIERA	Director Académico
Dr. Julio	OROZCO TORRENTERA	Revisor
Dr. Ing. Ariel	SABIGUERO	
Dr. Ing. Pablo	MUSÉ	

Agradecimientos

Agradezco al Dr. Ing. Pablo Musé Docente del IIE, Facultad de Ingeniería, UDELAR, y al Dr. Ing. Ariel Sabiguero Docente de Dpto. de Investigación Operativa, InCo, Facultad de Ingeniería, UDELAR por haber aceptado juzgar este trabajo y acceder a participar en el jurado.

Agradezco al Dr. Julio Orozco Torrentera (IT Service Implementation Expert - Orange Business Services - Applied research - Lannion, France) por aceptar ser el revisor de este trabajo y de participar en el jurado. Le doy las gracias cordialmente por los comentarios sobre la importancia del trabajo realizado en esta tesis.

Agradezco al MSc. Ing. Omar Viera. Investigador Área Informática PEDECIBA / InCo, Facultad de Ingeniería, UDELAR por los comentarios y correcciones que han sido una fuente de mejora del documento.

La realización de mi Maestría, y en particular el desarrollo de esta tesis, resultaron un proceso fluido y disfrutable gracias a mi tutor, Dr. Franco Robledo Amoza, quien supo transmitirme su entusiasmo por los temas abordados, confió plenamente en mis ideas y contribuyó a encauzarlas en el marco de un proyecto concreto.

Por último, quiero expresar un gran agradecimiento a los miembros de mi familia que nunca dejó de animarme. Especialmente quiero agradecer a Sofía, mi novia, que fue mi principal apoyo durante esta tesis. Dedico a ellos la presente tesis.

Índice general

Índice de Contenidos 1

Lista de Figuras 4

I INTRODUCCIÓN 7

1. Motivación y Marco Teórico 11

1.1. Introducción . . . 11

1.2. Objetivo General . . . 12

1.3. Trabajos Relacionados . . . 12

1.4. Estrategia a Seguir . . . 19

1.5. Estructura del Documento . . . 19

II RESOLUCIÓN MEDIANTE MÉTODO EXACTO 21

2. Descripción del Problema 23

2.1. Problema de Steiner Generalizado . . . 23

2.1.1. Definiciones Previas . . . 24

2.1.2. Formulación Matemática de GSP . . . 26

2.2. Descripción Verbal del Problema a Abordar . . . 27

2.3. Formulación Matemática del Problema . . . 28

2.3.1. Modelado del ASTNSNP Mediante un Problema de Programación Lineal Entera . . . 29

2.3.1.1. Definición de Variables . . . 30

2.3.1.2. Modelado del Problema ASTNSNP sobre $G'' = (V'', E'')$ . . . 30

2.3.1.3. Significado de las Restricciones . . . 31

2.3.1.4. Observaciones sobre el modelo . . . 31

2.3.2. ASTNSNP NP-Completo . . . 31

2.3.2.1. Relación del Modelo con GSP . . . 32

3. Implementación del Modelo Exacto **35**
3.1. Como implementar un Modelo de Programación Entera 35
3.2. ¿Por qué usar CPLEX, Concert Technology y C++? 36
3.3. Implementación del Método Exacto para el ASTNSNP 36
3.3.1. Lectura de grafo de entrada 37
3.3.2. Implementación de Splitting Vertex 37
3.3.3. Definir y Resolver el modelo usando CPLEX 38
3.3.3.1. Definición de Variables y Estructuras de Datos 38
3.3.3.2. Funcionalidades de CPLEX Utilizadas 40

4. Casos de Prueba y Resultados Obtenidos con el MIP **43**
4.1. Definición de Casos de Prueba 43
4.2. Resultados 44
4.2.1. Análisis de los Resultados 49

III RESOLUCIÓN POR MEDIO DE VNS **53**

5. La Metaheurística VNS **55**
5.1. ¿Que son las Heurísticas? 55
5.1.1. Variable Neighborhood Search 56
5.1.1.1. Esquema Básico 56
5.1.1.2. VNS Descendente 57
5.1.1.3. VNS Reducida 58
5.1.1.4. VNS Básica 58
5.2. Motivación de VNS 60
5.2.1. Solución Inicial 61
5.2.2. Vecindades 61

6. Resolución del Problema Mediante VNS **65**
6.1. Construcción de una Solución Factible Inicial 65
6.1.1. Algoritmos de camino mínimo 65
6.1.1.1. Algoritmo de Dijkstra 66
6.1.1.2. Algoritmo de Bhandari 67
6.1.2. Solución Factible Inicial 69
6.2. Definición de Vecindades 73
6.2.1. Optimización por reemplazo de key-path 73
6.2.2. Agregar Aristas\ Reducir Redundantes 76
6.2.3. Intercambio de dos Aristas entre cuatro Nodos 78
6.2.4. Optimización por reducción de key-tree 81
6.2.5. Optimización por reducción del key-tree con nodo raíz opcional 85
6.2.6. Optimización por Método Exacto 89
6.2.6.1. Algoritmos Extras 93
6.2.6.2. Formulación del Método Exacto como Búsqueda Local . . . 96

6.3. VNS: Variable Neighborhood Search 97
6.3.1. Implementación de VNS 99
6.3.1.1. Parametrización de VNS 99

7. Resultados Obtenidos Mediante el Uso de VNS **101**
7.1. Definición de casos de Pruebas 101
7.2. Presentación de los Resultados 102
7.2.1. Resultados de VNS comparados con Método Exacto 102
7.2.2. Resultados de VNS 111
7.2.3. Eficacia de las Vecindades 115

IV CONCLUSIONES **117**

8. Conclusiones y Trabajos Futuros **119**
8.1. Conclusiones de la Resolución Mediante Método Exacto 119
8.2. Conclusiones de la Resolución Mediante VNS 119
8.3. Trabajos Futuros 120
8.4. Conclusiones Generales de la Tesis 121

V Anexos **123**

A. Biblioteca Boost **125**
A.1. Boost Graph Library 125
A.1.1. Generalidad en Boost Graph Library 126
A.1.1.1. Algoritmo / Estructura de datos interoperables 126
A.1.1.2. Extensión a través de Visitors 126
A.1.1.3. Propiedades de vertices y aristas multi-parametrizables 126
A.1.2. Algoritmos 126
A.1.3. Estructuras de Datos 127

B. Biblioteca TSPLIB **129**
B.1. Formato de los archivos TSPLIB 129
B.2. Procesamiento de los archivos TSPLIB 131

VI Bibliografía **133**

Bibliografía **143**

Índice de figuras

2.1. Método de Separación de Vértices. 29

4.1. Caso de prueba Burma14 50
4.2. Caso de prueba Ulysses16 con configuración 1. 50
4.3. Caso de prueba Gr17 con configuración 2. 51
4.4. Gráfica con los tiempos insumidos por el Método Exacto en los casos de pruebas 52

5.1. Ejemplo de una instancia de ANTNSNP. 62
5.2. Solución del grafo ejemplo en la Figura 5.1. 63

6.1. Ejecución del paso 1 y 2 del Algoritmo de Bhandari 68
6.2. Grafo obtenido después de hacer la separación de vertices en el Algoritmo de Bhandari 68
6.3. Solución inicial incorrecta por la existencia del punto de articulación. 70
6.4. Grafo inicial con nodo ficticio agregado para solucionar el problema de generar puntos de articulación en la solución final 70
6.5. **Grafo Inicial** - Construcción de la solución inicial - Antes de aplicas Bhandari. 71
6.6. **Grafo Inicial** - Construcción de la solución inicial - 2 primeros caminos agregados. 71
6.7. **Grafo Inicial** - Construcción de la solución inicial - Mas caminos agregados . . 71
6.8. **Grafo Inicial** - Construcción de la solución inicial - 2 nuevos caminos agregados. 72
6.9. **Grafo Inicial** - Construcción de la solución inicial - Solución factible inicial. . 72
6.10. Intercambio de key-path por 2 caminos disjuntos de menor costo. 75
6.11. Vecindad agregar arista y reducir redundates. 77
6.12. Proceso del intercambio de 2 aristas entre 4 nodos. 79
6.13. Intercambio de un key-tree por pares de caminos nodos disjuntos 82
6.14. Proceso de reducción de key-tree por caminos disjuntos alternativos. 83
6.15. Reemplazo de un key-tree por un árbol de menor costo 85
6.16. Pasos iniciales para la reducción de key-tree con raíz opcional 87
6.17. Reemplazo de un key-tree con raíz opcional por un árbol de mejor costo 87
6.18. Proceso de optimización por Método Exacto 92

7.1. Comparativa de los resultados del Método Exacto con los de VNS para Bruma14 107
7.2. Comparativa de los resultados del Método Exacto con los de VNS para Ulysses16 107

7.3. Comparativa de los resultados del Método Exacto con los de VNS para Gr17 . 108
7.4. Comparativa de los resultados del Método Exacto con los de VNS para Gr21 . 108
7.5. Comparativa de los resultados del Método Exacto con los de VNS para Ulysses22 109
7.6. Comparativa de los resultados del Método Exacto con los de VNS para Gr24 . 109
7.7. Comparativa de los resultados del Método Exacto con los de VNS para Fri26 . 110
7.8. Comparativa de los resultados del Método Exacto con los de VNS para Bays29 110
7.9. Grafo original de Berlin52 113
7.10. Grafo solución de Berlin52 113
7.11. Grafo original de Eil51 114
7.12. Grafo original de Eil51 114
7.13. Grafo original de St70 115
7.14. Grafo original de St70 115

Parte I

INTRODUCCIÓN

ABSTRACT

El problema de encontrar una red de costo mínimo que sea 2 nodo conexa juega un papel importante en el diseño de redes confiables, las cuales deben ser tolerantes a fallos en sus componentes. Estas redes deben de tener al menos dos caminos nodos disjuntos para comunicar los nodos más importantes de la red, denominados nodos terminales. En esta tesis se pretende diseñar algoritmos que permitan, a partir de una red ya existente, ampliarla al menor costo posible de forma que siga siendo 2 nodo conexa y cubra el nuevo conjunto de nodos terminales.

En primer lugar, se resuelve el problema mediante la aplicación de un Método Exacto, el cual se define por medio de un modelo de Programación Lineal Entera basado en el Problema de Steiner Generalizado (Generalized Steiner Problem, GSP). Para este método sólo se utilizan grafos (redes) con una cantidad pequeña de nodos (no mayor a 30 nodos) dado que el problema es de clase computacional NP-Hard.

En segundo lugar, se presenta un algoritmo basado en la metodología de optimización conocida como Variable Neighborhood Search (VNS) que utiliza seis estructuras de vecindades diferentes instanciadas al problema planteado. De las cuales, tres están centradas en la optimización de la red a partir de la reducción de los nodos opcionales y las otras tres se basan en optimizar la red a partir de la reducción de aristas con mayor costo. Se destaca que en una de las vecindades se aplica Mixed Integer Programming (MIP) como búsqueda local para optimizar subgrafos del grafo solución. Las pruebas son realizadas sobre diferentes instancias transformadas a partir de problemas de la TSPLib: con una cantidad pequeña de nodos (no mayor a 30) para realizar una comparación de VNS con el Método Exacto diseñado; y con instancias más grandes (mayores a 40 nodos) para analizar la eficacia que tiene VNS en encontrar un óptimo local de buena calidad.

Palabras Claves: redes 2 nodo conexa, modelo de Programación Lineal Entera, Problema de Steiner Generalizado, Variable Neighborhood Search.

Capítulo 1

Motivación y Marco Teórico

1.1. Introducción

El ser humano del Siglo XXI disfruta de los beneficios que le brindan diferentes redes, tales como las redes de luz eléctrica, las redes de agua potable, las redes viales y las diferentes formas de comunicación existentes en la actualidad. Desde mediados del Siglo XX, las telecomunicaciones juegan un papel muy importante en la vida de los seres humanos, debido a la aparición de redes tales como la telegrafía, la radio, el teléfono, la television, las redes de computadoras y las redes de telefonía celular. En algunos casos, como por ejemplo, en las redes de fibra óptica de áreas metropolitanas, es importante que las mismas tengan un alto grado de confiabilidad, es decir, es necesario que la red sea resistente a fallas en algunos de sus componentes. Esta necesidad de confiabilidad, implica que el objetivo de minimizar el costo total de la red y el de lograr un nivel elevado de confiabilidad resultan contrapuestos y deben equilibrase.

La confiabilidad de una red está relacionada con el grado de conexión de la misma, es decir, con la cantidad mínima de caminos alternativos disjuntos que existen entre cualquier par de nodos de la red. Existen dos posibles formas de definir caminos disjuntos. La primera es mediante caminos arista-disjuntos y se refiere a caminos que no tienen aristas en común. La segunda forma es mediante caminos nodos disjuntos y estos son caminos que además de no compartir aristas tampoco comparten nodos entre ellos. Para asegurar que exista comunicación entre cualquier par de nodos de la red, se necesita asegurar un cierto grado de conexión entre dichos nodos.

La gran mayoría de las redes de fibra óptica de área metropolitana son redes 2-conexas, existiendo al menos dos caminos disjuntos comunicando los nodos más importantes de la red, denominados nodos terminales. En esta tesis, se pretende diseñar algoritmos que permitan, a partir de una red de fibra óptica 2-nodo-conexa existente ampliar la misma al menor costo posible, de forma tal que siga siendo 2-nodo-conexa y que cubra el nuevo conjunto de nodos terminales. Los costos manejados son los costos del tendido del enlace de fibra entre pares de nuevos nodos a cubrir o bien entre nodos ya existentes.

1.2. Objetivo General

Este trabajo tiene como objetivo resolver el problema de dos formas diferentes. En primer lugar, se resuelve dicho problema por medio de un Método Exacto, pero dado que el mismo es computacionalmente de clase NP-Hard, este método es aplicable sólo para redes con una cantidad pequeña de nodos. En la resolución mediante Método Exacto, se define un modelo de Programación Lineal Entera, basado en GSP (Generalized Steiner Problem) [97], que es utilizado para el diseño de redes con requerimientos de conectividad heterogéneos.

En segundo lugar, se requiere un algoritmo aproximado, plausible de ser aplicado a redes grandes, que logre soluciones factibles de buena calidad. De esta forma, se resuelve el problema mediante un método heurístico, más precisamente, se diseña un algoritmo basado en la metodología de optimización conocida como Variable Neighborhood Search (VNS) [92]. VNS es una metaheurística que explota la idea de búsqueda local intercambiando vecindades y así poder encontrar un óptimo que sea global o local de buena calidad. En particular, VNS es combinado con Variable Neighborhood Descent (VND) como procedimiento de mejora local.

1.3. Trabajos Relacionados

Usualmente las redes de comunicación se modelan como grafos no dirigidos, en las cuales los nodos representan sitios que emiten, reciben y canalizan información, y las aristas representan enlaces entre tales sitios. De este modo se cuenta con la posibilidad de formalizar y atacar estos problemas por medio de la Teoría de Grafos y el estudio de sus propiedades de conectividad [56]. Inspirados en los problemas mencionados de diseño de redes resistente a fallas, se han modelado y estudiado una serie de problemas de optimización combinatoria sobre grafos [105].

La investigación de la redundancia de conectividad entre nodos de un grafo ha sido fuertemente estudiada [68, 98, 105], en particular para los casos 2-conexos (o sea, donde se requiere que existan dos caminos independientes para comunicar cualquier par de nodos) [68, 83, 98, 115]. Para grafos cuyas aristas tienen costos que cumplen con la desigualdad triangular (lo cual es una restricción que suele ser cercana a los casos reales en el diseño topológico de redes) los problemas son conocidos como *minimum-weight 2-node-connected spanning network* y *minimum-weight 2-edge-connected spanning network* (MW2NCSN y MW2ECSN en adelante) [83].

Estos tipos de problemas normalmente se ubican en la categoría de problemas de clase computacional NP-hard (sabiéndose para algunos que son NP-completos), y su resolución requiere entonces el empleo de técnicas heurísticas de optimización para tamaños de problemas

a partir de cierta envergadura.

El problema de encontrar la topología de red de costo mínimo, que satisface ciertos requerimientos de conexión prefijados, puede ser modelado como un problema de Steiner en redes con requerimientos de conexión entre nodos. Seguidamente, se definen el Problema de Steiner Generalizado y el Problema de la subred de Steiner 2-nodo-confiable.

Dada una red de comunicación con ciertos nodos distinguidos, denominados nodos terminales, se definen:

- El Problema de Steiner Generalizado (*Generalized Steiner Problem* - GSP) refiere a una subred de costo mínimo, que verifique ciertos requerimientos prefijados de conexión entre cualquier par de nodos terminales [97].

- El Problema de la subred de Steiner 2-nodo-confiable (*Steiner 2-node-survivable network problem* - STNSNP) consiste en encontrar una subred de costo mínimo tal que entre todo par de nodos terminales existen al menos 2 caminos nodos disjuntos que los unen [36, 69].

A continuación, como problemas relacionados al abordado en esta tesis, nos enfocaremos en los problemas *Generalized Steiner Problem with Node Connectivity constraints* (expresado por GSP-NC y GSP-EC para la versión *Edge Connectivity*) y *Steiner 2-node-survivable network problem* (expresado por STNSNP) y sus modelos relacionados con la conectividad entre nodos o aristas, como los presentados en [105, 114, 115].

Winter [113–115] ha demostrado que el GSP-NC puede ser resuelto en tiempo lineal si es un grafo serie-paralelo, outerplanar, o es un grafo de Halin. A continuación, se presentará un resumen de los problemas de conectividad relacionados a GSP-NC y STNSNP. Gröstchel, Monma y Stoer [45] consideran un caso particular de GSP-NC trabajando en un contexto un poco diferente. Ellos lo llamaron el problema de *node-connectivity constraints* (NCON). En [105], Stoer introduce un amplio estudio para el NCON y ECON (la versión con *edge-connectivity constraints*), y algunos casos particulares. El NCON (resp ECON) es definido de la siguiente manera. Dado un grafo no dirigido $N = (X, U)$ tal que cada arista $e \in U$ tiene un peso asociado C_e que representa el costo de tendido del enlace. Cada nodo $i \in X$ tiene asociado un entero no negativo r_i, denominado tipo de i (requerimiento de conexión). Sea $H = (W, F)$ un subgrafo de N. Se dice que H satisface los requerimientos de nodo-conectividad si, para cada par $i, j \in X$ nodos distintos, H contiene al menos $r_{i,j} = \min\{r_i, r_j\}$ caminos nodos disjuntos que comunican a i con j. Similarmente para el ECON, se expresa que H satisface la restricción de arista-conectividad si, para cada par $i, j \in X$ nodos distintos, H contiene al menos $r_{i,j} = \min\{r_i, r_j\}$ caminos aristas disjuntos que comunican a i con j. Estas restricciones aseguran que algún camino entre i y j sobrevivirá a un cierto nivel de fallas de nodos (aristas).

Se observa que el modelo GSP-NC generaliza el modelo anterior, ya que en el GSP-NC existen requerimientos de conectividad r_{ij} que son dados independientemente por cada par de

nodos i, j. Sin embargo Gröstchel, Monma y Stoer [46–49] introducen el uso del tipo de nodo, que define los requerimientos de conectividad basado en la premisa de que estos expresan adecuadamente la importancia relativa asignada al mantenimiento de la conectividad entre oficinas. Ellos clasifican los diferentes tipos de problemas según los más grandes tipos de nodos que acontecen y en función de si los tipos de nodos representan requisitos de conectividad de nodo o arista. De esta forma, dado un grafo $N = (X, U)$ y un vector $r = (r_i)_{i \in X}$, asumiendo (sin perder generalidad) que existen al menos dos nodos de tipo k (siendo k el tipo más grande de nodo), ellos definen el problema kNCON (resp. kECON) cuando el objetivo es encontrar la red de costo mínima que satisface los requerimientos de nodo-conectividad (resp. los requerimientos de arista-conectividad). Si el valor más alto de k no es especificado entonces estos problemas se llaman NCON y ECON respectivamente. En particular, si todos los tipos de nodos tienen el mismo valor k, el problema NCON (resp. ECON) es reducido a encontrar la red de costo mínimo que satisface el requerimiento de k-nodo-conectividad (resp. k-arista-conectividad) entre todo par de nodos.

Hay que tener en cuenta que existe varias especializaciones de los problemas de conectividad que pueden ser formulados mediante la variación de sus parámetros, de la siguiente manera:

- Como se mencionó previamente, el GSP-NC y GSP-EC son modelos de conectividad más generales que NCON y ECON, ya que los requerimientos de conectividad son asociados a los pares de nodos de forma independiente y no necesariamente a todos los nodos de X.
- En el NCON y ECON, se tiene que $r_{i,j} = \min\{r_i, r_j\}$ para los tipos de nodos r_i, r_j que a su vez puede ser:
  - general (problemas kECON o kNCON),
  - uniforme (grafos de cubrimiento k-arista-conexo o k-nodo-conexo),
  - en $\{0, 1\}$ (Steiner trees),
- costos generales, euclideanos, uniformes

Existen casos de los problemas de NCON y ECON que son resueltos de forma polinomial. Estos resultan de relajar el problema original con restricciones como ser de costo uniforme, costo 0 o 1, nodos de tipos restringido y grafos especiales subyacentes tales como outerplanar, series-parallel y grafos de Halin. Todos estos casos particulares se exponen brevemente en [105]. Por otro lado, cotas inferiores y heurísticas con radio de performance garantizado en el peor caso fueron encontrados para los problemas kECON con costos restringidos, e.g., costos uniformes o costos que satisfacen la desigualdad triangular. Detalles de estos trabajos se pueden apreciar en [10, 17, 23, 37, 38, 42, 44, 83] y en forma resumida en [105]. En [105], Stoer también sintetiza procedimientos heuristicos para resolver problemas kECON y kNCON. Monma y Shallcross [84] dan heurísticas para los problemas kECON y kNCON. Frederickson y Jájá [38] proponen una heurística para el problema 2NCON con radio de performance garantido en el peor caso de $\frac{3}{2}$ por debajo de las cotas que cumplen la desigualdad

triangular. Ellos en el problema NCON en lugar de usar un vector $r = (r_i)_{i \in X}$ considerar una matriz $R = (r_{ij})_{ij \in X}$; esta variante ya se había planteado en 1969 por Steiglitz, Weiner y Kleitman [104], pero ellos no le dieron un nombre específico. Ellos desarrollaron un heurística simple para este problema que básicamente consiste en una rutina de comienzo randómico y una rutina de optimización donde transformaciones locales son aplicadas a una solución factible. Ko y Monma [73] propusieron heurísticas para el diseño de redes de cubrimiento k-arista o k-nodo conexos. Goemans y Bertsimas [42] proponen una heurística para el problema ECON con radio de performance garantido en el peor caso. Además, Goemans y Williamson [44] propusieron un algoritmo aproximado que puede ser aplicado al GSP-EC permitiendo el uso de multiples aristas paralelas. Khuller y Vishkin [72] proponen un algoritmo para el problema de kECON con radio de performance garantido en el peor caso de 2 con la restricción que las aristas paralelas no son permitidas en la solución y que todos los tipos de nodos son iguales. Balakrishnan, Magnanti y Mirchandani [7] presentaron nuevas formulaciones de Mixed-Integer Programming para el problema GSP-EC. Ellos proporcionan varias heurísticas para estas formulaciones, asegurando cotas para los costos heurísticos, relacionados con los valores óptimos de la Programación Lineal Entera. Por más detalles de estos trabajos (y de los test de performance) se puede consultar las referencias citadas.

Desafortunadamente, existen unos pocos algoritmos exactos para ECON y NCON de costo general. Christofides y Whitlock [24] introducen un algoritmo de cortes de planos junto con branch-and-bound para problemas ECON donde en lugar de un vector $r = (r_i)_{i \in X}$ se tiene una matriz $R = (r_{ij})_{ij \in X}$. Chopra y Gorres [22] dan un algoritmo de cortes de planos combinado con branch-and-bound para resolver problemas 2ECON. Markus Chimani, Maria Kandyba, Ivana Ljubic, y Petra Mutzel [21] presentan una transformación del problema 2NCON en un problema relacionado considerando grafos dirigidos y utilizan estos para establecer 2 formulaciones de Programación Lineal Entera (ILP, por sus siglas en inglés) para resolver el problema 2NCON, basadas en *multi-commodity flow* y en *directed cuts* respectivamente. Se prueban las ventajas de las formulaciones y se compara ambos enfoques tanto teórica como experimentalmente.

En la literatura hay muchos trabajos relacionados con algoritmos de aproximación para el GSP y diferentes casos particulares. En [94] los autores muestran como obtener soluciones óptimas para los problemas 2-arista-conexos abordados en [44]. Trabajos posteriores [39, 43, 112] extienden estos métodos para dar algoritmos de aproximación para el problema de GSP-EC sin duplicar links. Agrawal, Klein and Ravi [3] desarrollaron un algoritmo de aproximación para GSP-EC con garantías de performance de $2\lceil \log_2(r_{\text{máx}} + 1) \rceil$, donde $r_{\text{máx}}$ es el valor más alto exigido en la matriz de conexión R. Kortsarz, Krauthgamer y Lee [74] proporcionaron la primera cota inferior a la aproximación del problema GSP-NC cuando no hay nodos de Steiner.

Un caso especial de GSP-NC importante se produce cuando se buscan subgrafos k-nodo-conexos de costo mínimo que cubren todos los nodos del grafo. En primer lugar, se observará el caso general. En [17, 20, 30, 74, 75, 95, 96] los autores proponen varios algoritmos de aproximación para el problema de encontrar el subgrafo k-nodo-conexos de cubrimiento de costo

mínimo y además dan los respectivos coeficientes de aproximación. Para $k < 7$ un coeficiente de aproximación es $\lceil (k+1)/2 \rceil$, también se conoce para $k = 2$ [70], para $k = 2,3$ [5], para $k = 4,5$ [64] y para $k = 6,7$ [75]. Otra aproximación para $k = 2$ se puede observar en [11, 29]. Además, en [19, 30, 75] los autores proponen algoritmos de aproximación para los siguientes casos especiales: los grafos con topología Euclideana, costos uniformes y costos métricos (i.e cuando el costo cumple con desigualdad triangular).

Por último, se detallaran trabajos relacionados con el STNSNP. En [6] Baïou menciona diferentes problemas relacionados directamente con STNSNP. En particular los problemas conocidos como:

- Steiner 2-edge-connected subgraph problem (STECSP)
- Steiner 2-node-connected subgraph problem (STNCSP)
- Steiner 2-edge-survivable network problem (STESNP)

El STNSNP (resp. STESNP) también se relaciona con el problema kNCON (resp. kECON) en el caso donde $r_i \in \{0, 2\}$, $\forall i \in X$. Dado un grafo $N = (X, U)$, un subgrafo $T \subseteq X$ y una matriz C de costos de conexión asociados a U; el objetivo en el STNSNP (resp. STESNP) es encontrar el subgrafo 2-nodo-conexo de costo mínimo (resp. 2-arista-conexo) que cubre todos los nodos del conjunto T. Si la matriz C es positiva, el conjunto de soluciones óptimas asociado a los problemas STNSNP y STNCSP son iguales. Sucede lo mismo para el conjunto de soluciones óptimas de STENSP y STECSP. Si todos los nodos son obligatorios (no tiene nodos de Steiner) STESNP y STECSP coinciden, y también coinciden el problema STNSNP con STNCSP. Además, es fácil de observar que una solución factible del STNCSP (resp. STECSP) es también solución factible del STNSNP (resp. STESNP). En [27] los autores desarrollaron un algoritmo lineal para resolver el STNCSP en el caso de grafos sin W_4 (un grafo con cuatro nodos) y grafos de Halin. En [14] el autor propuso el desarrollo de métodos paralelos para el caso general (con peor caso complejidad exponencial). Otro trabajo relacionado con el caso particular de STNCSP, e.g. cuando $T = X$ o con costos uniformes, ya se ha mencionado anteriormente.

A continuación, se define formalmente el problema planteado en esta tesis y se presentan trabajos relacionados con la ampliación de grafos nodos-conexos (resp. aristas-conexos) a grafos k-nodos-conexos (resp. k-arista-conexos).

Dado un grafo no dirigido $G = (V, E)$ con una matriz de costos reales positivos $C = \{c_{ij}\}_{(i,j)\in E}$ asociados a las aristas de G. Dado un subgrafo $H \subset G$, donde H se define como $H = (X, A)$, siendo $X \subset V$, $A \subset E$, y dado un conjunto de nodos $T \subset V \setminus X$ que se denominan nodos terminales. Las aristas de H se asume de costo cero. Se define:

- *Augmentation Steiner two-node Survivable Network Problem* - ASTNSNP(V,E,X,A,T,C) como el problema de encontrar un subgrafo $G_{sol} \subset G$ que sea 2-nodo-conexo de costo

mínimo tal que $H \subset G_{sol}$ y cubra todo el conjunto T de nuevos nodos terminales (todo nodo de T está en G_{sol}).

El problema de ampliación de redes nodo-conexas o arista-conexas tiene como objetivo encontrar el conjunto de arista de costo mínimo que amplía un grafo m-nodo-conexo (resp. m-arista-conexa) a un grafo k-nodo-conexo (resp. k-arista-conexa). Algunas de las principales referencias en esta área son [16, 18, 35, 40, 67, 71, 76, 87, 91, 93, 108, 109]. Estos papers proponen diferentes algoritmos de aproximación con sus respectivos coeficientes de aproximación. Algunos de estos trabajos estudian el caso particular cuando los costos son uniformes, que se conoce comúnmente como *minimum-size connectivity problem*.

Para el problema de ampliación de redes arista-conexas, en [110] los autores muestran que la cantidad mínima de aristas que se necesitan para que un grafo sea k-arista-conexo es determinada en $O(kL|V|^4(k|V| + |E|))$ para cualquier grafo $G = (V, E)$ siendo $k \geq 2$ y $L = \text{mín}\{k, |V|\}$. Para el caso de $k = 3$ en [111] los autores resuelven dicho problema en grafos no dirigidos y en [63] Bill Jacksona y Tibor Jordán desarrollan un algoritmo que proporciona una solución óptima en tiempo polinomial para todos los grafos k-nodo-conexos. Recientemente, en [15] los autores establecen que el k-VSCAP (*k-Vertex-Connected Subgraph Augmentation Problem*) es de clase APX-hard (APX es una abreviación de aproximable) e introducen 2 cotas inferiores para los coeficientes de los algoritmos de aproximación desarrollados para resolver estos tipos de problemas. En [41] Anna Galluccio and Guido Proietti prueban que en tiempo polinomial se puede resolver el problemas de encontrar el conjunto de aristas de costo mínimo, de un grafo G, para que un subgrafo H de cubrimiento de G sea ampliado a 2-arista-conexo, con la condición de que H sea un árbol de expansión de G creado por el algoritmo DFS (Depth-First Search). También, ellos proporcionan un algoritmo eficiente para la resolución de este caso especial, con coeficiente de aproximación 2, que se ejecuta en $O(M * \alpha(M, n))$, donde α es la inversa clásica de la función de Ackermann y $M = m * \alpha(m, n)$. En [26] Michele Conforti1, Anna Galluccio, Guido Proietti presentan una formulación para resolver el problema de encontrar un conjunto de aristas de G de costo mínimo, para que un subgrafo H de cubrimiento de G sea ampliado a 2-arista-conexo. Además, analizan las condiciones para que el modelo relajado de la programación lineal de dicha formulación siempre de soluciones enteras. En [106] los autores proponen que, dado un multigrafo $G = (V, E)$ 2-nodo-conexo y dos enteros positivos j, k, el problema de ampliar G con el menor número de aristas para obtener un multigrafo j-arista-conexo y k-nodo-conexo se lo puede resolver en $O(mn^2)$ para cualquier j y para $k = 3$, donde $n = |V|$ y m es el número de pares de vértices.

Dado un grafo dirigido y dos subconjuntos de nodos H_1 y H_2, en [59] los autores definen *bi-level augmentation problem* como el problema de agregar a G un pequeño número de nodos tal que el grafo resultado contenga dos caminos nodos disjuntos entre cualquier par de nodo de H_1 y dos caminos aristas disjuntos entre cualquier par de nodo de H_2. Ellos presentan un algoritmo que resuelve este problema en tiempo lineal y al configurar adecuadamente H_1 y H_2 este algoritmo pertenece a la categoría de algoritmos existente para la resolución del problema de ampliación de grafos k-arista-conexos (resp. k-nodo-conectados).

En [50, 85, 88] los autores proponen varios algoritmos de aproximación para la ampliación de grafos aristas-conexos o nodos-conexos a grafos de costos mínimos k-aristas-conexos o k-nodos conexos, donde cada uno de estos algoritmos tienen sus respectivos coeficientes de aproximación. En particular, el trabajo propuesto por Guy Even a, Guy Kortsarzb y Zeev Nutov [50] dan un coeficiente de aproximación de 1,5 para grafos 2-arista-conexos de costo mínimo y Hiroshi Nagamochi [85] introduce un algoritmo de aproximación con coeficiente $\frac{4}{3}$ para grafos 2-nodos-conexos, además de una cota inferior para los valores óptimos obtenidos.

En [107] Csaba D. Tóth propone que todos grafo conexo planable con n nodos puede ser ampliado a un grafo planable 2-arista-conexo agregando al menos $\frac{2}{3}n+O(1)$ aristas y que todo árbol planable con n nodos puede ser apliado a un grafo planable 2-arista-conexo agregando al menor $\lceil \frac{n}{2} \rceil$ aristas. Además, para estos tipos de grafo, en [100] Ignaz Rutter y Alexander Wolff dan un cota inferior para la solución óptima cuando $k = 2$ y para $k = 3$ identifican todos los casos que tienen soluciones. En [89] proponen un algoritmo para el problema de ampliación de grafos *outerplanar* agregando un mínimo número de nodos tal que el nuevo grafo G' es *outerplanar* y 2-arista-conexo.

En [8] los autores consideran el problema de ampliación de un grafo G que es $(k - 1)$-conexo a un grafo k-conexo cuando k es un número fijo o $k = n - 1, n - 2$. En este problema, n es la cantidad de nodos del grafo G. Ellos proponen que el problema de ampliación de $(n - 3)$-conectividad puede ser reducido al problema de encontrar el máximo *square-free 2-matching in a simple subcubic graph*, cuando se considera el grafo complemento $\bar{G}$ de G. Ellos observan que un grafo es $(n - 2)$-conexo si cada nodo de $\bar{G}$ tiene a lo sumo grado 1. Esto implica que si $k = n - 2$ el problema es equivalente a encontrar un máximo *matching* en el grafo complemento $\bar{G}$. Por lo tanto, se puede verificar fácilmente que un grafo G es $(n - 3)$-conexo si y sólo si el conjunto de aristas de $\bar{G}$ es un *square-free 2-matching*, que significa que cada nodo en $\bar{G}$ tiene a lo sumo grado 2 y $\bar{G}$ no contiene ciclos de largo 4. También observan que si G es $(n - 4)$-conexo entonces el complemento $\bar{G}$ es un grafo *subcubic* (i.e cada nodo tiene a lo sumo grado 3).

Del análisis de la bibliografía existente respecto a los problemas de diseño topológico de redes robustas relacionado al ASTNSNP podemos constatar que existe una vasta literatura inherente a trabajos que presentan algoritmos de aproximación con radio de performance garantido para problemas de aumentación del nivel de nodo o arista conectividad de un grafo. Se mencionaron aquí los que entendemos más relevantes por su similitud y/o cercanía a la formulación del ASTNSNP.

En particular destacamos dos problemas de estrecha relación con el ASTNSNP, a saber: el GSP y el STNSNP. Citamos aquí trabajos donde se resolvieron eficientemente ambos problemas mediante un enfoque de Algoritmos Poliédricos [105] y un enfoque Metaheurístico [13, 14, 36, 98].

1.4. Estrategia a Seguir

En este trabajo de tesis tomamos como hipótesis de partida la existencia de una topología 2-nodo-conexa ya existente sobre la cual se desea conectar nuevos nodos de forma que la red resultante también posea una estructura topológica 2-nodo-conexa. Estaciones opcionales (nodos de Steiner) son también tomados en cuenta en el diseño de planificación. Es así que definimos el ASTNSNP en términos de grafos ponderados con costos de conexión entre sus nodos, modelando los costos de tendido factible de líneas entre nodos (típicamente costos de construcción por dragado). Además, demostraremos la naturaleza del ASTNSNP es de clase computacional NP-Completo.

Los principales aporte de este trabajo de tesis son resolver el problema ASTNSNP mediante un modelo exacto de Programación Lineal Entera para resolver instancias de pequeño porte y mediante un enfoque metaheurístico para grandes instancias del problema (más de 40 nodos). En particular se elige la metodología de optimización VNS (Variable Neighborhood Search) por su gran potencial y eficiencia en la resolución de problemas de optimización combinatoria similares. Destacamos además que como parte del Algoritmo VNS diseñado para el ASTNSNP uno de sus algoritmos de Búsqueda Local es un Método Exacto basado en Programación Lineal Entera para: dado un subgrafo 2-nodo-conexo de la solución factible actual, encontrar el mejor subgrafo de cubrimiento 2-nodo-conexo que sustituya al primero de forma de preservar la factibilidad de la solución. Dicho algoritmo utiliza como base el siguiente resultado estructural:

Lema (Monma et al. 1990) [83]: Sea $G = (V, E)$ un grafo 2-conexo con $G' = (V', E')$ un sugrafo de G inducido por V'. Entonces reemplazando E' en G por cualquier conjunto de aristas E'' definidas sobre V' , donde $G'' = (V', E'')$ es 2-conexo, resulta en un grafo $G^* = (V, (E \setminus E') \cup E'')$ el cual es 2-conexo.

De forma más general, la sustitución de un sugrafo de un grafo 2-nodo-conexo dado por otro sugrafo que es 2-nodo-conexo preserva la 2-nodo-conectividad de la red global [98].

1.5. Estructura del Documento

Este trabajo, se organiza de la siguiente manera. Primero, en el Capítulo 2 se presenta una breve introducción al diseño de redes confiables, se describe el Problema de Steiner Generalizado, las modificaciones a dicho problema para reflejar la realidad planteada en esta tesis y se brinda una definición formal del problema mediante su formulación matemática como un problema de Programación Lineal y Entera. En el Capítulo 3 se brindan las consideraciones más importantes sobre la implementación de Método Exacto. Se describe cómo se implementa el Método Exacto por medio de la herramienta CPLEX [34] y se brinda el pseudocódigo del algoritmo utilizado. En el Capítulo 4 se definen los casos de prueba utilizados y el porcentaje de cada tipo de nodo seleccionado para cada caso de prueba. También, se presentan los resultados obtenidos con el Método Exacto implementado con la herramienta CPLEX. En el Capítulo 5 se realiza una breve introducción sobre heurísticas, se explica en detalle la metaheurística

VNS y la motivación del uso de ésta para la resolución del problema a abordar en esta tesis. En el Capítulo 6 se describe el enfoque de VNS en detalle, primero se explica la construcción de una solución inicial factible para el ASTNSNP. Después se explican las diferentes vecindades realizadas y las distintas técnicas de búsqueda aplicadas en ellas; y por último se detalla el algoritmo VNS aplicado para la resolución del problema. En el Capítulo 7 se definen los casos de prueba y se muestran los resultados obtenidos mediante la aplicación del algoritmo de VNS diseñado. Por último, en el Capítulo 8 se presentan las conclusiones obtenidas como resultado de este trabajo.

Parte II

RESOLUCIÓN MEDIANTE MÉTODO EXACTO

Capítulo 2

Descripción del Problema

Este Capítulo se estructura de la siguiente manera: en la Sección 2.1, se presenta, a modo de introducción, el problema GSP. En la Sección 2.2, se realiza la descripción formal del problema a abordar en esta tesis y en la Sección 2.3, se realiza la formulación matemática del problema.

2.1. Problema de Steiner Generalizado

En esta Sección, se introduce el Problema de Steiner Generalizado (GSP por su sigla en inglés) y la formulación matemática asociada al mismo, la cual es presentada en [99] y [86]. Presentar GSP tiene como objetivos, en primer lugar, introducir al lector en la aplicación del mismo, es decir, el problema de encontrar redes de costo mínimo que cumplen con ciertos requerimientos de conectividad entre sus nodos, y en segundo lugar, comparar el mismo con el problema que se plantea en la sección 2.3 y mostrar que este último puede verse como un caso particular del GSP.

En el diseño de redes con requerimientos de conectividad simple (un camino entre cualquier par de nodos), el principal objetivo es encontrar la estructura de red que minimice los costos de realizar dicha red. Por otra parte, cuando se incorporan requerimientos de conectividad extra, que garantizan la existencia de caminos alternativos entre pares de nodos ante la eventual caída de algún componente de la red, el problema crece en complejidad ya que se debe buscar la estructura de red con mínimo costo que asegure dichos requerimientos de conectividad.

Ante esta realidad planteada, mediante el uso de grafos para representar la estructura de red, el GSP propone encontrar una subred de costo mínimo, que cumpla con los requisitos de conectividad para cualquier par de nodos terminales. En el contexto del problema GSP los nodos terminales son aquellos que poseen requerimientos de conectividad y aquellos que no poseen ninguno son denominados nodos opcionales o nodos de Steiner.

A continuación, y haciendo uso de algunas definiciones previas, presentaremos formalmente el problema GSP.

2.1.1. Definiciones Previas

En este apartado estableceremos algunas definiciones previas y convenciones de las cuales haremos uso en el resto del trabajo.

En el contexto de problemas en redes con requisitos de supervivencia, éstos suelen expresarse básicamente en dos formas distintas:

- requisitos respecto a la cantidad de aristas (links) que debe ser posible suprimir de la red sin que queden desconexos ciertos pares de nodos terminales; se traducen en requisitos de caminos arista-disjuntos entre pares de nodos terminales.
- requisitos respecto a la cantidad de nodos (terminales y/o opcionales) que debe ser posible suprimir (obviamente junto con las aristas que les inciden), sin que queden desconexos ciertos pares de nodos terminales; se traducen en requisitos de caminos nodo-disjuntos entre pares de nodos terminales.

Definición 2.1.1 *Se dice que una pareja de nodos i, j tiene **k-arista-conectividad** o está **k-arista-conectada** en un grafo dado, cuando existen al menos k caminos arista disjuntos (o sea, que no comparten ninguna arista tomados 2 a 2) que conectan i con j.*

Esta definición es equivalente a afirmar que todo corte del grafo para i, j contiene al menos k aristas.

Definición 2.1.2 *Se dice que un grafo es **k-arista-conexo** cuando toda pareja de nodos i, j del mismo está k-arista-conectada.*

En forma análoga se definen los conceptos para nodo-conectividad.

Definición 2.1.3 *Se dice que una pareja de nodos i, j tiene **k-nodo-conectividad** o está **k-nodo-conectada** en un grafo dado, cuando existen al menos k caminos nodo disjuntos (o sea, que no comparten ningún nodo salvo i, j) que conectan i con j.*

Definición 2.1.4 *Se dice que un grafo es **k-nodo-conexo** cuando toda pareja de nodos i, j del mismo están k-nodo-conectada.*

Nótese que si dos caminos con los mismos extremos i, j son nodo-disjuntos, entonces son también arista-disjuntos; pero la afirmación contraria es falsa, es decir, dos caminos arista-disjuntos pueden en general ser o no nodo-disjuntos.

Para presentar el problema de GSP, se consideran los siguientes elementos:

- Un grafo no dirigido $G = (V, E)$, siendo V el conjunto de nodos (representan los puestos de red) y E el conjunto de aristas (representan los potenciales enlaces de comunicación). Se utiliza la notación de arista (i, j) para representar los enlaces de comunicación sin dirección.

- Una matriz $C = \{c_{ij}\}_{(i,j)\in E}$ de costos no negativos asociados a las aristas del grafo G.
- Un conjunto de nodos terminales T, $T \subseteq V$, donde la cardinalidad de T es $n_T =\mid T \mid$ tal que $2 \leq n_T \leq n$ donde $n =\mid V \mid$ la cardinalidad del conjunto V.
- Una matriz $R = \{r_{ij}\}_{i,j\in T}$ de dimensión $n_T * n_T$, cuyos elementos son enteros positivos que indican los requerimientos de conectividad entre todo par de nodos terminales (cantidad de caminos disjuntos requeridos entre pares de nodos terminales).

Definición 2.1.5 ***Problema GSP***. *Dados un grafo conexo $G = (V, E)$, una matriz C de costos positivos asociados a las aristas del grafo, un subconjunto de nodos $T \subseteq V$ tal que $|T| > 2$, y una matriz de requerimientos de conexión entre nodos terminales $R = \{r_{ij}\}_{i,j\in T}$ donde los r_{ij} son enteros no negativos; el objetivo es encontrar un subgrafo $G_T = (V_T; E_T)$ de G tal que $T \subseteq V_T$, cada par de nodos terminales distintos i, j tenga r_{ij}-conectividad (o sea estén unidos en G_T mediante al menos r_{ij} caminos disjuntos) y tal que el costo total de G_T sea el mínimo posible. Por disjuntos puede entenderse "nodo-disjuntos" o "arista-disjuntos" obteniéndose las versiones "nodo-conectividad" o "arista-conectividad" respectivamente del problema.*

Sobre los nodos no pertenecientes al conjunto de nodos terminales no se plantean requisitos de conectividad. Estos nodos, conocidos como nodos opcionales o nodos de Steiner, pueden formar parte de la solución si su inclusion mejora los costos de la solución.

La versión del problema en la que R especifica requerimientos de arista-conectividad se denomina GSP-EC (GSP Edge-Connected) o también GSP-ED (GSP Edge-Disjoint). La versión en la que R especifica requerimientos de nodo-conectividad se denomina GSP-NC o GSP-ND (GSP Node-Connected o Node-Disjoinct). Al definir una instancia del problema, es necesario especificar el grafo ponderado G, el subconjunto T de nodos terminales, la matriz de costos C y la matriz de requerimientos R; así como también debe especificarse claramente si se trata de la versión EC o la NC.

A continuación se define los conceptos de solución factible y solución optimal.

Definición 2.1.6 *Una **solución factible** G_{fac} es un subgrafo de G que contiene al menos r_{ij} caminos disjuntos $\forall i, j \in T$, $i \neq j$. A su vez, una **solución globalmente optimal** G_T es una solución factible cuyo costo es igual al menor costo de los elementos del conjunto de soluciones factibles.*

Dados G, T, C, R, cualquier solución factible para la versión NC será también solución factible del problema versión EC, puesto que si existen r_{ij} caminos nodo-disjuntos entre i, j, dichos caminos son también arista-disjuntos. Ahora bien, si además es una solución optimal del NC, no necesariamente lo será para el EC puesto que pueden existir soluciones de menor costo en éste último problema que hagan uso de la posibilidad de compartir nodos intermedios entre caminos disjuntos para un mismo par de terminales.

En el sentido opuesto, cualquier solución factible para la versión EC puede ser o no una solución factible del problema NC; ello dependerá de que el subgrafo contenga un conjunto

de r_{ij} caminos nodo-disjuntos para cada par de terminales i, j. Por último, si una solución optimal para EC resulta factible para NC, entonces es también optimal para NC; puesto que de no serlo, existiría otra solución factible para el NC de menor costo, que como ya hemos visto sería también factible en EC, lo cual es absurdo.

2.1.2. Formulación Matemática de GSP

La solución al problema GSP (versión Arista-Conectividad), se obtiene resolviendo el siguiente problema de Programación Lineal Entera (PLE). Para la formulación del PLE, se definen las variables $x_{i,j}$ (binaria) e $y^{uv}_{i,j}$. La variable $x_{i,j}$ representa la existencia en la solución final del arco que une los nodos i y j, es decir, si la variable toma el valor 1 el arco esta en la solución final y si toma el valor 0 ese arco no es parte de la solución. Para toda arista $(i, j) \in E$, y $u, v \in T, u \neq v$ la variable $y^{uv}_{i,j}$ denota la utilidad de la arista (i, j) en la dirección de i hacia j en un camino que une el nodo terminal u con el nodo terminal v; si la variable toma el valor 1 dicha arista es utilizada en el camino de u hacia v y si toma el valor 0 no es utilizada en el camino.

$$\text{mín} \sum_{(i,j)\in E} c_{ij} * x_{ij} \tag{2.1}$$

s.a.:

$$\sum_{(u,j)\in E} y^{uv}_{(u,j)} \geq r_{u,v} \; \forall u, v \in T, u \neq v. \tag{2.2}$$

$$\sum_{(i,v)\in E} y^{uv}_{(i,v)} \geq r_{u,v} \; \forall u, v \in T, u \neq v. \tag{2.3}$$

$$\sum_{(i,p)\in E} y^{uv}_{(i,p)} - \sum_{(p,i)\in E} y^{uv}_{(p,i)} \geq 0 \; \forall u, v \in T, \forall p \in V \backslash u, v. \tag{2.4}$$

$$y^{uv}_{(i,j)} + y^{uv}_{(j,i)} \leq x_{ij} \; \forall u, v \in T, u \neq v, \forall (i, j) \in E. \tag{2.5}$$

$$x_{(i,j)} \in \{0, 1\} \; \forall (i, j) \in E. \tag{2.6}$$

$$y^{uv}_{(i,j)} \geq 0 \; \forall i, j : (i, j) \in E, \; \forall u, v \in T, u \neq v. \tag{2.7}$$

La restricción 2.2 implica que para todo nodo u que pertenece al conjunto T salen al menos 2 caminos hacia cualquier nodo v que pertenece al conjunto T siendo $u \neq v$. La restricción 2.3 implica que para todo nodo v que pertenece al conjunto T llegan al menos 2 caminos de cualquier nodo u que pertenece al conjunto T siendo $u \neq v$. La restricción 2.4 define el balance de aristas entrantes y salientes en cada uno de los nodos intermedios de los caminos que existen entre los nodos u, v, mientras que la restricción 2.5 determina que los caminos sean aristas disjunto, es decir, si la arista es usada en un camino entre u, v no es usada en otro camino entre

los mismos nodos. Además representa la existencia de la arista (i, j) en la solución.

Llamaremos $U = \{u_{ij}\}$ a la solución óptima del problema GSP planteado, donde U queda determinado por el conjunto de aristas que cumplen $(i, j) \in E, x_{ij} = 1$ y determina el subgrafo solución G_T.

Observación: Una versión análoga de GSP para el problema nodo-conexo, puede obtenerse mediante el uso de la técnica de separación de vertices (Splitting Vertex) [90].

2.2. Descripción Verbal del Problema a Abordar

El problema que se desea resolver, es la ampliación de una red de fibra óptica ya existente, la cual tiene una estructura anillada, esto determina que es una red 2-nodo-conexa. Dicha ampliación implica la conexión de nuevos nodos de carácter obligatorios y la posibilidad de conectar otros nodos, que son opcionales. Es decir, todo el conjunto de nodos obligatorios estarán sin excepción en la nueva red ampliada y aquellos nodos opcionales cuya inclusión en la red mejore los costos de la ampliación, también formarán parte de la nueva red. La inclusión de un nodo en la nueva red implica la realización del cableado de fibra óptica, así como la conexión a la red existente, por esto, cada nueva conexión tendrá un costo asociado.

El objetivo, es encontrar una estructura óptima para la nueva red, de forma tal de incorporar todos los nuevos nodos obligatorios a la red ya existente a un costo mínimo y mantener la característica de 2-nodo-conectividad en la nueva red obtenida. En este trabajo, a los nodos obligatorios se les denomina nodos terminales.

En la siguiente sección, se utilizan grafos para representar la estructura de la red, donde en cada grafo los vértices son los nodos de la red y la existencia de una arista entre cualquier par de vértices implica que existe una conexión de fibra óptica entre ellos o bien la factibilidad de un tendido de línea entre ellos. Los costos de conexión entre cualquier par de nodos de la red, es reflejado en el grafo mediante el costo asociado a dicha arista.

Se empleará la siguiente notación:

- G : Grafo compuesto por todos los nodos de la red que forman el conjunto de nodos V y las conexiones entre ellos que representa el conjunto de aristas E;
- V : Conjunto de nodos del grafo G, representan los sitios de conmutación representados por los que ya están conectados, los nuevos nodos terminales y opcionales;
- E : Aristas del grafo G; representan posibles links entre sitios que es posible construir / operar;
- $C : E \rightarrow \mathbb{R}^+$: Costos de las aristas (reales no negativos), representan el costo de construir / operar un link; nos referiremos a ella como “matriz de costos”;

- H : Se identifica como la red ya existente (ya está construida e instalada) que es subgrafo de G;
- X : Conjunto de nodos del subgrafo H;
- A : Aristas del grafo H; los costos de las aristas del subgrafo H es cero puesto que representan tendidos de línea ya realizados;
- $T \subseteq V, T \cap X = \emptyset$: Nuevos nodos terminales que se desea integrar a la red.
- $S \subseteq V, S \cap X = \emptyset, S \cap T = \emptyset$: nuevos nodos opcionales o de Steiner que son plausibles de ser utilizados en la integración de T a la red existente.

2.3. Formulación Matemática del Problema

Dado un grafo no dirigido $G = (V, E)$ con una matriz de costos reales positivos $C = \{c_{ij}\}_{(i,j)\in E}$ asociados a las aristas de G. Dado un subgrafo $H \subset G$, donde H se define como $H = (X, A)$, siendo $X \subset V$, $A \subset E$, y dado un conjunto de nodos $T \subset V \setminus X$ que se denominan nodos terminales. Las aristas de H se asume de costo cero. Se define:

Definición 2.3.1 ***(Augmentation Steiner two-node Survivable Network Problem - ASTNSNP)*** *Definiremos el problema de ampliación de una red 2-nodo-conexa ASTNSNP(V,E,X,A,T,C) como el problema de encontrar un subgrafo $G_{sol} \subset G$ que sea 2-nodo-conexo de costo mínimo tal que $H \subset G_{sol}$ y cubra todo el conjunto T de nuevos nodos terminales (todo nodo de T está en G_{sol}).*

Este problema pertenece a la clase computacional NP-Hard, esto se demostrará en la Sección 2.3.2 por medio de la reducción del STNSNP (Steiner two-node Survivable Network Problem) a él.

Enfoque de Resolución: Para modelar el problema como un problema de Programación Lineal Entera se transforma el grafo $G = (V, E)$ en un grafo dirigido $G' = (V', E')$, donde para cada arista $(i, j) \in E$ existe una arista dirigida $(i \to j) \in E'$ y otra arista dirigida $(j \to i) \in E'$. De esta manera se obtiene el subgrafo $H' \subset G'$ y la matriz de costos $C' = \{c'_{ij}\}_{(i\to j)\in E'}$. Desde ahora se utiliza la notación de arista (i, j) para especificar la dirección de i a j.

Un grafo 2-nodo-conexo tiene la propiedad que dado cualquier par de vértices existen al menos 2 caminos nodo-disjuntos que los unen. Para imponer que el grafo solución cumpla con esta propiedad, se realiza una transformación del grafo dirigido $G' = (V', E')$ mediante la aplicación de la técnica de separación de vértices (Splitting Vertex en inglés)[90] y se obtiene un nuevo grafo $G'' = (V'', E'')$.

La técnica de separación de vértices, implica que dado un vértice v en el grafo original G' este es sustituido por 2 vértices v_1, v_2 en el grafo resultante G'', de forma tal que cada arista que llegaba a v llegará al nuevo vértice v_1 y cada arista que sale de v saldrá del vértice v_2. A su vez, v_1 y v_2 son unidos a través de una arista adicional que tiene la dirección $(v_1 \rightarrow v_2)$ y con costo cero. En la Figura 2.1 se representa graficamente esta técnica.

Una vez obtenido el grafo $G'' = (V'', E'')$ y modelando el problema como un problema de Programación Lineal Entera (ver Sección 2.3.1), se obtiene un subgrafo solución que es 2-arista-conexo. Esta solución tiene la particularidad, que dado cualquier vértice de T y cualquier vértice de X existen al menos 2 caminos que los unen sin repetir aristas. Por lo tanto, las aristas que fueron agregadas por la técnica de separación de vértices no se repiten en dichos caminos, logrando así, que tampoco se repitan los vértices determinados por estas aristas. Realizando la operación inversa al splitting, es decir, uniendo los vértices que fueron separados, se obtiene caminos que son nodo-disjuntos, logrando así un grafo solución del ASTNSNP que es 2-nodo-conexo.

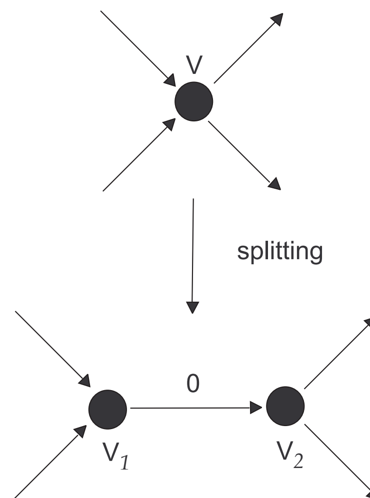


Figura 2.1: Método de Separación de Vértices.

2.3.1. Modelado del ASTNSNP Mediante un Problema de Programación Lineal Entera

Para representar el problema planteado, se modela el mismo por medio de un problema de Programación Lineal Entera. En primer lugar, se definen las variables necesarias para modelar el problema, en segundo lugar se presenta el modelo, luego se realiza una breve descripción del significado de cada una de las restricciones y por último se hacen algunas observaciones.

2.3.1.1. Definición de Variables

Para poder representar la realidad planteada, se definen dos variables $y_{i,j}^{uv}$ y $x_{i,j}$. La variable $y_{i,j}^{uv}$ denota la utilidad de la arista (i,j) en la dirección de i hacia j entre cualquier par de nodos u, v, donde el nodo u pertenece al conjunto de los nuevos nodos terminales T y el nodo v pertenece al conjunto de nodos X. Recordar que los nodos del conjunto X son parte de la red existente que se desea ampliar. La variable binaria $x_{i,j}$ representa la existencia en la solución final del arco que une los nodos i y j. Si la variable toma el valor 1 el arco esta en la solución final y si toma el valor 0 ese arco no es parte de la solución.

Variables de Decisión sobre $G'' = (V'', E'')$:

$$y_{(i,j)}^{uv} = \begin{cases} 1 & \text{si el arco } (i,j) \in E'' \text{ esta en el camino de } u \in T'' \text{ a } v \in X'' \\ 0 & \text{sino} \end{cases} \tag{2.8}$$

$$x_{(i,j)} = \begin{cases} 1 & \text{si el arco } (i,j) \in E'' \backslash A'' \text{ es utilizado} \\ 0 & \text{sino} \end{cases} \tag{2.9}$$

2.3.1.2. Modelado del Problema ASTNSNP sobre $G'' = (V'', E'')$

$$\text{mín} \sum_{(i,j) \in E''} c_{ij} * x_{ij} \tag{2.10}$$

s.a.:

$$\sum_{(u,j) \in E'' \backslash A''} y_{(u,j)}^{uv} \geq 2 \, \forall u \in T'', \forall v \in X''. \tag{2.11}$$

$$\sum_{(i,v) \in E''} y_{(i,v)}^{uv} \geq 2 \, \forall u \in T'', \forall v \in X''. \tag{2.12}$$

$$\sum_{(i,p) \in E''} y_{(i,p)}^{uv} - \sum_{(p,i) \in E''} y_{(p,i)}^{uv} \geq 0 \, \forall u \in T'', \forall p \in V'' \backslash \{u, v\}, \forall v \in X''. \tag{2.13}$$

$$y_{(i,j)}^{uv} + y_{(j,i)}^{uv} \leq x_{(i,j)} \, \forall u \in T'', \forall v \in X'', \forall (i,j) \in E''. \tag{2.14}$$

$$x_{i,j} \in \{0, 1\} \, \forall (i,j) \in E''. \tag{2.15}$$

$$y_{(i,j)}^{uv} \geq 0 \, \forall i, j : (i,j) \in E'' \, \forall u \in T'', \forall v \in X''. \tag{2.16}$$

2.3.1.3. Significado de las Restricciones

A continuación, se describe el significado de cada una de las restricciones:

- **Restricción 1 (Ecuación 2.11):** Impone que para todo nodo que se encuentra en el conjunto de terminales T'', debe haber al menos 2 caminos hacia cualquier nodo del conjunto X'', o sea 2 caminos hacia cualquier nodo de la red ya existente.

- **Restricción 2 (Ecuación 2.12):** Esto implica, que para todo nodo que se encuentra en el conjunto X'', debe haber al menos 2 caminos hacia cualquier nodo del conjunto T''. Es decir, que para cualquier nodo de la red ya existente, hay 2 caminos hacia cualquiera de los nuevos nodos terminales.

- **Restricción 3 (Ecuación 2.13):** Dados los nodos $u \in T''$ y $v \in X''$, esta restricción representa el balance de aristas entrantes y salientes en cada uno de los nodos intermedios de los caminos que existen entre los nodos u, v. Este balance de aristas en los nodos intermedios de los caminos obliga a que exista la cantidad de caminos exigidos mediante las restricciones 1 y 2.

- **Restricción 4 (Ecuación 2.14):** Determina que los caminos sean aristas disjuntos y la existencia del arco (i, j) en la solución. Esta restricción obliga a que si una arista es usada para un camino entre el par de vértices u, v no sea usada para otro camino entre dicho par de vértices. Al existir dicha arista en cualquiera de los dos sentidos ($i \to j$ o $j \to i$), implica que existe el arco (i, j) en el grafo solución final.

2.3.1.4. Observaciones sobre el modelo

- La red H es anillada, por tanto ya es 2-nodo-conexa y está construida, por este motivo se asume que el costo de sus aristas es 0, ya que el objetivo del problema es minimizar los costos del cableado de fibra óptica que se desea realizar para incorporar los nuevos nodos terminales pertenecientes a T a la red ya existente.

- Se asume que el grafo G es 2-nodo-conexo. Esto implica, que el grafo original G brinda la posibilidad de conectar la red existente H con los nuevos nodos terminales T y mantener la 2-nodo conectividad.

A continuación se demuestra que el problema ASTNSNP es de clase computacional NP-Completo.

2.3.2. ASTNSNP NP-Completo

Teorema 2.3.2 *(**ASTNSNP NP-Completitud**) El ASTNSNP es un problema de clase NP-Completo.*

Prueba. Demostraremos que el ASTNSNP es de clase NP-Completo reduciendo el *Steiner Two-Nodo Survivable Network Problem* (STNSNP) a él.

Sea STNSNP(V, E, T, C) una instancia del *Steiner Two-Nodo Survivable Network Problem*. Para esta instancia construiremos una instancia de ASTNSNP de la siguiente manera:

- Elegimos $z \in T$ fijo y tomamos $X = \{z\}$ y $A = \emptyset$; $H = (\{z\}, \emptyset)$
- $\hat{T} = T \backslash \{z\}$

Tenemos entonces el problema ASTNSNP$(V, E, \{z\}, \emptyset, \hat{T}, C)$. Claramente este proceso de transformación de una instancia STNSNP a una instancia ASTNSNP es polinomial. Veremos ahora que cualquier solución factible minimal del STNSNP induce una solución factible minimal del correspondiente ASTNSNP (teniendo el mismo costo) y recíprocamente.

$(\Rightarrow)$ Sea $\bar{H}$ una solución factible minimal del STNSNP. $\bar{H}$ es en si mismo una solución factible minimal para el ASTSNSP puesto que:

- $H \subset \bar{H}\,(z \in \bar{H})$
- $\bar{H}$ cubre T y en particular $\hat{T}$
- $\bar{H} \setminus e$ **no** es 2-nodo-conexo $\forall e \in \bar{H}$

$(\Leftarrow)$ Sea $\hat{H}$ una solución factible minimal del ASTNSNP$(V, E, \{z\}, \emptyset, \hat{T}, C)$. Tenemos que:

- $\hat{H}$ es 2-nodo-conexo (por definición de ASTNSNP)
- $\hat{H}$ cubre $\{z\}$ y a $\hat{T}$, por lo tanto a T.
- $\hat{H}$ contiene solamente aristas de E
- $\hat{H} \backslash e$ **no** es 2-nodo-conexo, $\forall e \in \hat{H}$

Luego $\hat{H}$ es también solución factible minimal para el STNSNP(V, E, T, C). Finalmente, dado que el STNSNP es NP-Completo, entonces el ASTNSNP es NP-Completo.

QED

2.3.2.1. Relación del Modelo con GSP

Como se puede observar, el modelo planteado en esta tesis es muy similar al modelo planteado en GSP. En particular, el modelo planteado en este trabajo se lo puede considerar como un caso particular de GSP. Si se toma en cuenta, en primer lugar, que los requerimientos de conectividad $r_{i,j}$ del problema GSP tomam el valor 2 para que el grafo cumpla con la restricción de 2 arista conectividad. Y en segundo lugar, considerando que al conjunto de nodos de T del problema GSP, se lo puede dividir en los subconjuntos X y T del modelo planteado para este

trabajo. Se puede apreciar entonces que el modelo planteado para GSP se lo puede transformar en el modelo presentado en esta tesis.

A continuación se demuestra que a partir de una instancia del modelo STNSNP se puede resolver el modelo ASTNSNP.

Proposición 2.3.3 *Dada una instancia de ASTNSNP, existe una instancia del problema STNSNP que lo resuelve.*

Prueba. Sea ASTNSNP(V, E, X, A, T, C) con $H = (X, A)$. Construiremos la siguiente instancia del STNSNP:

- $\hat{T} = T \cup X$
- $\hat{C} = \{\hat{c}_{i,j}\}_{(i,j) \in E}$ /

$$\hat{c}_{i,j} = \begin{cases} c_{i,j} & \text{si } (i,j) \in E \backslash A. \\ 0 & \text{sino} \end{cases} \tag{2.17}$$

- $\hat{V} = V$
- $\hat{E} = E$

Tenemos entonces el STNSNP$(\hat{V}, \hat{E}, \hat{T}, \hat{C})$. Sea $\hat{H}$ solución óptima global de este problema. Sea $\bar{H} = \hat{H} \cup A$. Se cumple que:

$i)$ $\bar{H}$ es solución factible de STNSNP

$ii)$ $cost(\bar{H}) = cost(\hat{H})$ (puesto que $cost(A) = 0$)

De (i) y (ii) $\bar{H}$ también es óptimo global del STNSNP. Veremos ahora que $\bar{H}$ es óptimo global del ASTNSNP. Se cumple que:

- Claramente $\bar{H}$ es factible para el ASTNSNP pues cubre $T \cup X$, $A \subset \bar{H}$ y $\bar{H} \subseteq G$.
- Supongamos que $\bar{H}$ no es óptimo global. Existe entonces $\tilde{H}$ solución factible del ASTNSNP tal que: $cost(\tilde{H}) < cost(\bar{H})$. Pero $\tilde{H}$ es factible para el STNSNP$(\hat{V}, \hat{E}, \hat{T}, \hat{C})$ lo cual contradice la optimalidad de $\hat{H}$ pues se tendría: $cost(\tilde{H}) < cost(\bar{H}) = cost(\hat{H})$.

Luego $\bar{H}$ es óptimo global del ASTNSNP(V, E, X, A, T, C).

Dado que el STNSNP es NP-completo, entonces el ASTNSNP es al menos NP-Hard.

QED

En el siguiente Capítulo se detalla la implementación del Método Exacto planeado para resolver el ASTNSNP. Se explican las herramientas utilizadas para realizar dicha implementación y se brinda el pseudocódigo del algoritmo diseñado.

Capítulo 3

Implementación del Modelo Exacto

3.1. Como implementar un Modelo de Programación Entera

Una vez expresado el problema ASTNSNP como un problema de programación lineal entera, se debe decidir como se realiza la implementación del código que permita resolver dicho problema y/o cuales son las herramientas o bibliotecas que se utilizan para lograrlo.

Para la resolución del problema por Método Exacto, se aplican los métodos para resolución de problemas de programación entera mixta, conocidos como problemas MIP por su nombre en inglés Mixer Integer Programming [66]. La resolución de problemas MIP se realiza aplicando los algoritmos basados en Branch-And-Bound, conocidos en español como algoritmo de Ramificación y Acotación[1] y algoritmos derivados de este, a los que se les incorpora planos de corte, como ser el algoritmo Branch and Cut, llamado en español Ramificación y Corte [79] [80]. Por lo tanto, se analizan posibles paquetes de software construidos específicamente para la resolución de problemas de programación entera y programación lineal. En este caso se decide investigar ILOG CPLEX y ABACUS.

ILOG CPLEX o simplemente CPLEX [2] como se le conoce comúnmente, es un paquete de software para resolver problemas de optimización [34]. IBM[3] lo define como un motor de programación matemática de alto rendimiento. CPLEX toma su nombre de la combinación de método simplex y el lenguaje C, ya que fue diseñado originalmente para resolver el método simplex y escrito en lenguaje C. Hoy en día, contiene interfaces para los lenguajes C++, C# y Java. CPLEX resuelve problemas de programación lineal básica (PL), problemas de programación cuadrática (PQ)[4], problemas de programación entera mixta (MIP)[5], entre otros. Mas información sobre CPLEX puede leerse en [61].

[1] En la literatura en español se puede encontrar otras traducciones para el algoritmo de Branch and Bound como por ejemplo Ramificación y Cota, Ramificación y Acotamiento.

[2] CPLEX Fue originalmente desarrollado por Robert E. Bixby y vendido como CPLEX Optimization Inc a la empresa ILOG en 1997 que a su vez en 2009 es adquirida por IBM.

[3] http://www-142.ibm.com/software/products/es/es/ilogcple/

[4] La función objetivo contiene términos cuadráticos.

[5] Estos son problemas PL o PQ con la restricción que alguna de las variables toman valores enteros.

ABACUS[6] es un framework orientado a objetos, escrito en lenguaje C++, diseñado para la implementación de programación lineal basada en los algoritmos Branch and Cut, Branch and Price, Branch and Bound y sus combinaciones [102] [103]. Utilizando ABACUS la definición y resolución de un problema se realiza mediante la implementación de algunas clases derivadas de clases abstractas, que son las interfaces que brinda ABACUS.

3.2. ¿Por qué usar CPLEX, Concert Technology y C++?

Durante la investigación sobre el modelo planteado y luego de consultar bibliografía (por ejemplo [21] [77] [4] [78]) enfocada a resolver problemas similares al planteado en esta tesis, se observó que CPLEX es la herramienta más utilizada en la resolución de este tipo de problemas. En base a esa apreciación se decidió investigar con mayor profundidad CPLEX y si este podría cubrir las necesidades para la implementación del modelo de programación entera planteado. Además, CPLEX ofrece la posibilidad de hacer uso de sus bondades mediante interfaces definidas para lenguajes como C, C++ y Java entre otros [28] [61].

Analizadas las bondades que brindan las API de CPLEX para C (denominada Callable Library) y C++ (denominada Concert Technology) las que se detallan en [25], se decidió usar lenguaje C++ y la API Concert Technology de CPLEX version 10.0.1. Esta decisión se fundamenta en la facilidad de uso que brinda la interfaz para C++ y en el hecho de que tiene una menor curva de aprendizaje, esto permite una mejor eficiencia en cuanto a tiempos de implementación.

3.3. Implementación del Método Exacto para el ASTNSNP

La implementación del método exacto puede resumirse en cuatro etapas: en primer lugar la lectura del grafo de entrada, segundo la transformación de grafo mediante Splitting Vertex, tercero la implementación del modelo definido para el problema usando CPLEX y por último la realización de la función inversa al Splitting sobre el grafo solución obtenido como resultado de dicho problema.

La lectura de grafo de entrada desde archivos de la biblioteca TSPLib [1] es desarrollada en la Sección 3.3.1, la transformación mediante Splitting Vertex se explica en la Sección 3.3.2 y en la Sección 3.3.3 se realiza una breve descripción de la implementación del problema mediante el uso de la API de CPLEX para C++.

[6] ABACUS fue originalmente desarrollado por Stefan Thienel como tesis de doctorado defendida en diciembre de 1995. En enero de 1996 es publicada la primera versión y su primera versión comercial fue la version 2.3. Al momento de este trabajo la última versión lanzada es la 3.0.

3.3.1. Lectura de grafo de entrada

Para la obtención de grafos que permitan realizar pruebas, se utilizó la biblioteca TSPLib [1] y en particular los grafos definidos para el problema de Traveling Salesman Problem (TSP)[7]. TSPLib reúne una cantidad considerable de grafos definidos y utilizados para la resolución de problemas TSP y otros relacionados, como por ejemplo la búsqueda de ciclos hamiltonianos mínimos. En el Anexo B de esta tesis, se presenta un análisis sobre TSPLib, se describen los archivos con extension "tsp" que contienen grafos para el problema mencionado y se detalla como se realiza la lectura de los mismos. En la referencia [1], se puede encontrar información detallada de esta biblioteca.

En la Sección 4.1, se describe cómo a partir de los datos obtenidos de la biblioteca TSPLib, se transforman los mismos para obtener grafos que nos permiten definir los datos de entrada para el problema planteado.

3.3.2. Implementación de Splitting Vertex

Como se describe en la Sección 2.3, para lograr que el grafo solución del problema de programación entera además de ser 2-arista-conexo sea también 2-nodo-conexo, se realiza previamente una transformación del grafo mediante la técnica de separación de vértices conocida como Splitting Vertex. Para realizar esta transformación, se toma el grafo dirigido G obtenido como entrada y se transforma dicho grafo en otro grafo G' dirigido, donde cada vértice v del grafo G es separado en 2 vértices v_1 (vértice de entrada) y v_2 (vértice de salida) en el grafo G'. Además, se realiza una transformación del conjunto de aristas del grafo, es decir, para toda arista que llega a v desde cualquier nodo u en el grafo G, se define una arista en el grafo G', que va desde el vértice de salida u_2 al vértice de entrada v_1 y para toda arista que sale de v a un nodo u en el grafo G, se define una arista en G' que va desde el vértice de salida v_2 al vértice de entrada u_1 y además en el grafo G', se crea la arista que va de v_1 a v_2 para todo vértice v del grafo G.

Para realizar el Splitting Vertex, se transforma el vector de vértices V del grafo G en el vector de vértices V' del grafo G', la matriz de existencia de aristas $M_{i,j}$ del grafo G en la matriz $M'_{i,j}$ del grafo G' y la matriz de costos asociados a las aristas $C_{i,j}$ del grafo G en la matriz $C'_{i,j}$ del grafo G'.

Primero, se realiza la transformación del vector de vértices V en el vector V', pasando de tener n elementos a tener $2n$, donde se cumple, que para todo vértice v del vector V que ocupa la posición i los vértices v_1 y v_2 del vector V' ocupan las posiciones $2i$ y $2i + 1$. En el caso de la matriz de existencia de aristas, la transformación genera, a partir de la matriz $M_{i,j}$ de tamaño $n \times n$, la matriz $M'_{i,j}$ con tamaño $2n \times 2n$ donde los elementos de la nueva matriz se obtienen mediante la siguiente formula.

Por cada arista $x = (i, j)$ en la matriz $M_{i,j}$ donde i, j son vértices del grafo original, se generan las nuevas aristas:

[7] Traveling Salesman Problem - En español se traduce en Problema del Viajante de Comercio. TSP es uno de los problemas más famosos en el campo de la optimización combinatoria computacional.

$$x_{2i,r} = (2i, r) = \begin{cases} 1 & \text{si } r = 2i + 1 \\ 0 & \text{en otro caso} \end{cases} \tag{3.1}$$

$$x_{2i+1,r} = (2i + 1, r) = \begin{cases} 1 & \text{si } r = 2j\, y\, si\, (i, j) = 1 en\, G \\ 0 & \text{en otro caso} \end{cases} \tag{3.2}$$

La matriz de costos $C'_{i,j}$ del grafo G', con la separación de vértices ya realizada, se genera a partir de la matriz de costos $C_{i,j}$ del grafo original G. Al igual que para la matriz de existencia de aristas, la matriz de costos original es de tamaño $n * n$, por lo tanto, la matriz $C'_{i,j}$ tiene tamaño $2n * 2n$ y la generación de elementos de la nueva matriz se realiza mediante una transformación similar a la vista para la matriz de existencia de aristas.

Cada elemento $c_{i,j}$ en la matriz $C_{i,j}$ representa el costo asociado a la arista (i, j), donde i, j son vértices del grafo original G. La nueva matriz se genera siguiendo la transformación siguiente:

$$c_{r,p} = \begin{cases} c_{i,j} & \text{si } r = 2i + 1 \text{ y } p = 2j \\ 0 & \text{en otro caso} \end{cases} \tag{3.3}$$

Concluida la transformación del grafo mediante Splitting Vertex, se obtiene el grafo dirigido que es el parámetro de entrada para el problema de programación entera que se resuelve mediante CPLEX.

3.3.3. Definir y Resolver el modelo usando CPLEX

La API Concert Technology de CPLEX brinda funcionalidades que hacen posible implementar, es decir, definir y solucionar, el modelo de programación entera definido en la Sección 2.3.1.2. Las funcionalidades mencionadas son brindadas principalmente mediante el uso de objetos y métodos que permiten representar problemas matemáticos. Entre los objetos, se encuentran los que permiten definir variables (tanto de datos como de decisión) y expresiones (que representan las restricciones y la función objetivo). Se destacan los métodos que permiten aplicar una optimización al modelo, métodos para consultar valores de variables, para consultar resultados obtenidos luego de la optimización y aquellos que permiten modificar el problema y el comportamiento del algoritmo de optimización.

3.3.3.1. Definición de Variables y Estructuras de Datos

En la resolución del Problema Matemático planteado en la Sección 2.3.1.2, se debe representar las variables de decisión que son $y^{uv}_{i,j}$ y $x_{i,j}$. Como ya se explicó en la Sección 2.3.1.1, la variable $y^{uv}_{i,j}$ denota la utilidad de la arista (i, j) en la dirección de i hacia j entre cualquier par de nodos u, v, donde el nodo u pertenece al conjunto de nuevos terminales (obligatorios) y el nodo v pertenece al conjunto de nodos que ya están conectados en la red que se desea

ampliar. La variable $x_{i,j}$ representa la existencia del arco que une los nodos i y j en la solución final, es decir, la imposición de la restricciones del modelo determina, si la variable toma el valor 1 el arco esta en la solución final y si toma el valor 0 ese arco no es parte de la solución.

Para expresar la variable $y_{i,j}^{uv}$, se necesita que por cada par de nodos u, v exista una matriz donde reflejar las posibles aristas (i, j) que conforman los caminos que los unen. Por lo tanto, para representar todos los caminos conformados por cualquier par de nodos u, v, se necesita una matriz de dos dimensiones, donde cada elemento es a su vez es una matriz de elementos i, j. En resumen, $y_{i,j}^{uv}$ se representa mediante una matriz de cuatro dimensiones.

Para expresar la variable $x_{i,j}$, se necesita simplemente una matriz de dos dimensiones, donde reflejar la existencia o no de la arista (i, j) en la solución final.

Utilizando la API Concert Technology de CPLEX para la definición de las variables mencionadas, primero se realiza la declaración de tipos de variables que reflejan una matriz variable de dos dimensiones y otra de cuatro dimensiones. El código asociado a esto es el siguiente:

1 Definiciones de Tipos para Variables de decisión usando CPLEX Concert Technology

```
typedef IloArray < IloIntVarArray > IntVarMatrix
typedef IloArray < IntVarMatrix > IntVarMatrix3dim
typedef IloArray < IntVarMatrix3dim > IntVarMatrix4dim
```

Luego se define la variable y del tipo $IntVarMatrix4dim$, que representa $y_{i,j}^{uv}$ y la variable x del tipo $IntVarMatrix$, que representa $x_{i,j}$.

2 Definiciones de Variables de decisión usando CPLEX Concert Technology

```
IntVarMatrix x(env, nNodos)
IntVarMatrix4dim y(env, nNodos)
```

Además de la representación de las variables de decisión para definir el problema, es necesario contar con la matriz de costos asociados a las aristas, la matriz de existencia de aristas y un vector donde se almacena la información del conjunto de vértices al que pertenece cada nodo (nodo de red existente, nodo obligatorio o nodo opcional), que llamaremos vector de tipos de nodo. Las matrices y el vector definidos conforman las variables de datos del problema original.

Luego, al aplicar el Splitting Vertex, por cada nodo, se genera un par de nodos, donde uno es identificado como nodo de entrada y el otro como nodo de salida. Con el fin de diferenciar los nodos de salida de los nodos de entrada, se crea una nueva variable de datos, representada mediante un vector, donde se refleja el sentido de cada nodo. A continuación, se muestran la definición de las variables de datos mencionadas.

3 Definiciones de Variables de datos usando CPLEX Concert Technology

$IntMatrix\ cost(env)$
$IntMatrix\ aristas(env)$
$IloIntArray\ tipoNodos(env)$
$IloIntArray\ sentidoNodosSplit(env)$

Una vez definidas las variables de decisión y las variables de datos, se está en condiciones de representar mediante expresiones la función objetivo y las restricciones del modelo.

3.3.3.2. Funcionalidades de CPLEX Utilizadas

A continuación, se brinda un resumen de los objetos y métodos más importantes utilizados en la resolución de un problema de programación entera como el planteado en este trabajo:

- **Definición de ambiente (IloEnv):** Para utilizar el resto de los objetos y métodos que provee Concert Technology, antes de crear cualquier modelo se debe crear un objeto de la clase IloEnv, este objeto es responsable de administrar la entrada/salida, la asignación de memoria y otros servicios necesarios para el funcionamiento de CPLEX.
- **Definición de modelo (IloModel):** En un objeto de la clase IloModel, se adjuntan los datos que son necesarios para describir el problema.
- **Definición de variables de datos (IloInt, IloFloat, IloBool, IloIntArray, IloNumArray, IloBoolArray):** Mediante la definición de objetos de estas clases, se definen las variables que contienen los datos asociados al problema.
- **Definición de variables de decisión (IloNumVar, IloIntVar, IloBoolVar, IloNumVarArray, IloIntVarArray, IloBoolVarArray):** Mediante la definición de objetos de estas clases, se definen las variables de decisión que tomarán un valor una vez resuelto el problema.
- **Definición de Expresiones (IloExpr):** Mediante la definición de objetos de la clase IloExpr, se reflejan las restricciones y la función objetivo asociadas al problema.
- **Adjuntar expresiones al modelo (modelo.add()):** Mediante el método $add()$ del objeto IloModel se adjuntan al modelo la función objetivo y las restricciones correspondientes. En el caso de la función objetivo, se utiliza además uno de lo métodos $IloMinimize$ o $IloMaximize$.
- **Configurar algoritmo de resolución (IloCplex):** Para resolver el problema se hace uso de un objeto de la clase IloCplex, esta clase hereda de la clase IloAlgorithm y es responsable de ejecutar el algoritmo de resolución. El algoritmo a utilizar es definido mediante este objeto, así como los parámetros, comportamientos y configuraciones que se desee imponer a dicho algoritmo.

- **Resolución del problema (IloCplex.extract(), IloCplex.solve()):** Mediante el método $extract()$ de un objeto de la clase IloCplex, se extrae el problema desde el objeto de la clase IloModel y se resuelve luego mediante la invocación del método $solve()$.

- **Lectura de los resultados (IloCplex.getStatus(), IloCplex.getObjectValue(), IloCplex.getValue()):** La lectura de la solución del problema, se realiza a través de las funciones miembro de la clase IloCplex. Una vez resuelto el problema, mediante la función $getStatus()$, se puede saber el estado del algoritmo. Con el uso de la función $getObjectValue()$ se recupera el valor obtenido para la función objetivo y mediante la función $getValue(variable)$ se recuperan los valores obtenidos para las variables de decisión.

Con estos objetos y métodos presentados, se está en condiciones de reflejar las variables y expresiones que conforman el problema así como lograr la resolución del mismo. A continuación se muestra el seudocódigo de la definición y resolución del problema.

Como se puede observar en el Seudocódigo 4, para la resolución del problema mediante CPLEX, se define un objeto *env* de la clase *IloEnv* que define el ambiente, un objeto *model* de la clase *IloModel* para definir el modelo. Se definen las variables y luego se adjuntan al modelo la función objetivo *obj* y las restricciones mediante expresiones reflejadas en objetos de la clase *IloExpr*, que representan operaciones algebraicas aplicadas sobre las variables de datos y de decisión. Una vez cargados todos los datos al modelo, se define un objeto *cplex* de la clase *IloCplex*, donde se extraen los datos del problema con el método *extract()* y a continuación se ejecuta el método *solve()* que resuelve el problema. Después, se recupera desde el objeto *cplex* el valor mínimo de la función objetivo obtenido como resultado de la resolución del problema, así como, los valores de las variables de decisión, que son en definitiva los que determinan la estructura de la red óptima, que es la solución al problema planteado. Con los resultados obtenidos, se realiza la técnica inversa al splitting de vértices para lograr el grafo solución del problema aplicado al grafo original.

Seudocódigo 4 Resolución del problema mediante método exacto usando CPLEX Concert Technology

```
IloEnv env;
try{
    IloModel model(env);
    Leer Archivos TSPLib
    SplittingVertex()
    Definir Variables
    Cargar Datos a Variables
    IloExpr funcionObjetivo = ....;
    model.add(IloMinimize(env, funcionObjetivo));
    IloExpr expre1 =  ....;
    ....
    IloExpr expre5 =  ....;
    model.add(expre1);
    ....
    model.add(expre5);
    IloCplex cplex(env);
    cplex.extract(model);
    cplex.solve();
    InversoSplittingVertex()
    Obtener resultados
}
catch (IloException &e) \{
    desplegar error
}
env.end();                    \\free all memory
```

Capítulo 4

Casos de Prueba y Resultados Obtenidos con el MIP

En este Capítulo, se presentan los resultados obtenidos con el método exacto implementado en CPLEX. En la Sección 4.1, se definen los casos de prueba utilizados y el porcentaje de cada tipo de nodo utilizado para cada caso de prueba. En la Sección 4.2 se muestran los resultados obtenidos para los casos de prueba y un análisis de los mismos.

4.1. Definición de Casos de Prueba

Con el objetivo de generar casos de prueba tanto para el modelo de Programación Lineal Entera como para la heurística propuesta, es necesario contar con un juego de datos. De ese juego de datos, se deben obtener grafos que permitan representar tanto la red de fibra óptica, ya existente, así como los nuevos nodos terminales y los nodos opcionales que se desean considerar para la ampliación de dicha red. Además de ello, es necesario contar con un peso asociado a las aristas que unen dichos nodos. En el problema planteado, este peso representa el costo de realizar el cableado de fibra óptica entre pares de nodos. Para el caso de los nodos que ya están cableados, es decir que ya forman parte de la red existente, ese costo es considerado con valor 0.

Para la obtención de grafos, se hace uso de la biblioteca TSPLib [57] que provee una amplia cantidad de ejemplos de grafos almacenados en archivos independientes con un determinado formato, que hacen posible su procesamiento. La estructura que poseen dichos archivos y el proceso de lectura de los mismos se presenta en el anexo B de esta tesis. Particularmente, se utilizaron para las pruebas los archivos definidos para el problema Traveler Salesman Problem (TSP).

Una vez recabados los datos desde los archivos de la TSPLib [57], se generan los tres conjuntos de nodos (nodos pertenecientes a red existente, nuevos nodos terminales y opcionales) y se crea la matriz de costos asociada a las aristas. La creación de la matriz de costos implica realizar cálculos para la obtención de dichos costos, por ejemplo, en caso de tomar un grafo donde el costo de las aristas está dado por la distancia euclideana, dicha distancia se debe cal-

cular a partir de las coordenadas de los nodos.

La asignación de nodos a los diferentes conjuntos se realiza al azar, es decir, se recorre la lista de nodos de las instancias TSP elegidas y por cada nodo se realiza una elección aleatoria de cual será el conjunto al que pertenecerá dicho nodo en la instancia ASTNSNP que se construye. En este procedimiento de asignación se toma en cuenta la cantidad de nodos de cada conjunto y dicha cantidad es definida mediante porcentajes del total de nodos del grafo, estos porcentajes son leídos desde un archivo de configuración. Los diferentes porcentajes de nodos definidos y un determinado grafo asociado con dichos porcentajes determinan un caso de prueba. Esto permite realizar un análisis de cómo se comporta el algoritmo ante distintas cantidades de nodos de la red fija (conjunto X), de los nuevos nodos terminales (conjunto T) y de los nodos opcionales (conjunto Op).

Por cada instancia TSP elegida, los porcentajes que se tomaron son las siguientes:

- **Configuración 1**: Conjunto X = 20 %, Conjunto T = 60 %, Conjunto Op = 20 %
- **Configuración 2**: Conjunto X = 20 %, Conjunto T = 40 %, Conjunto Op = 40 %
- **Configuración 3**: Conjunto X = 20 %, Conjunto T = 20 %, Conjunto Op = 60 %
- **Configuración 4**: Conjunto X = 50 %, Conjunto T = 25 %, Conjunto Op = 25 %
- **Configuración 5**: Conjunto X = 50 %, Conjunto T = 40 %, Conjunto Op = 10 %
- **Configuración 6**: Conjunto X = 50 %, Conjunto T = 10 %, Conjunto Op = 40 %
- **Configuración 7**: Conjunto X = 70 %, Conjunto T = 5 %, Conjunto Op = 25 %
- **Configuración 8**: Conjunto X = 70 %, Conjunto T = 15 %, Conjunto Op = 15 %
- **Configuración 9**: Conjunto X = 70 %, Conjunto T = 25 %, Conjunto Op = 5 %

Una vez concluida la asignación de nodos a los diferentes conjuntos, se modifica la matriz de aristas para asignar costos con valor cero a las aristas que conecten dos nodos cualesquiera del conjunto X y así reflejar que el costo de realizar el cableado en la red existente es nulo porque el mismo ya ha sido realizado.

4.2. Resultados

En esta Sección, se muestran los resultados obtenidos para los casos de pruebas realizados. Dichas pruebas fueron realizadas en un Intel Core i5 de 2.67 GHz, de 8 Gbytes de RAM y sobre Windows 7 de 64 bits.

En primer lugar, en la Tabla 4.1, se presentan las instancias de grafos seleccionados de la librería TSPLib y sus características.

Instancia TSP	Cantidad de nodos	Cantidad de Aristas
gr17	17	136
gr21	21	210
gr24	24	226
ulysses16	16	120
ulysses22	22	231
fri26	26	325
bays29	29	406
burma14	14	91

Tabla 4.1: Instancias de TSPLib utilizadas para pruebas con el Método Exacto

Es importante destacar que la cantidad de aristas indicada en la Tabla 4.1 es el número de aristas para el grafo no dirigido completo sin contabilizar las aristas que van desde un nodo a si mismo.

En las Tablas 4.2 a 4.10 se muestran los resultados obtenidos aplicando el Método Exacto para las nueve configuraciones de casos de prueba definidos. Los datos desplegados en dichas tablas son: nombre de la instancia TSP del grafo original, cantidad de nodos y aristas totales del grafo inicial, cantidad de nodos para cada conjunto de nodos, cantidad de aristas en el grafo $H = (X, A)$, cantidad de nodos opcionales en el grafo solución, costo de la solución y tiempo consumido por CPLEX para resolver el problema.

Para mejorar la visualización de las tablas se definen algunos términos que son presentados a continuación:

- N: Cantidad total de nodos del grafo.
- A: Cantidad total de aristas del grafo.
- N_X: Cantidad de nodos del conjunto X (nodos de la red existente).
- N_T: Cantidad de nodos del conjunto T (nuevos nodos obligatorios: terminales).
- N_{Op}: Cantidad de nodos del conjunto Op (nuevos nodos opcionales: nodos de Steiner).
- A_X: Cantidad aristas en la red existente (aristas entre nodos del conjunto X).
- $Tiempo$: Tiempo en segundos insumido por el solver de CPLEX para resolver el problema.
- VAL_{sol}: Costo del grafo solución obtenido (grafo 2-nodo-conexo).

- OP_{sol}: Cantidad de nodos opcionales en el grafo solución obtenido.

- A_{sol}: Cantidad de aristas en el grafo solución obtenido.

También se identifican algunos resultados especiales en aquellos casos de prueba donde no se logra obtener un resultado. Más precisamente se define los siguientes resultados:

- **No Aplica:** Se registra este resultando en los grafos que, dada una determinada configuración, el conjunto de nodos terminales es vacío, por lo cual no tiene sentido ejecutar el caso de prueba para ese grafo. Por ejemplo, en la configuración 7, en los grafos gr17 y ulysses16 el conjunto de nuevos nodos terminales es vacía, por tanto no se realiza el caso de prueba.

- **Out Memory:** Este resultado indica que, dada una configuración de casos de prueba aplicada a un grafo, CPLEX no puede resolver el modelo debido a que la memoria es insuficiente.

Instancia	N	A	N_X	N_T	N_{Op}	$Tiempo(s)$	VAL_{sol}	OP_{sol}	A_{sol}
gr17	17	136	3	10	4	3,60	1.515,00	1	15
gr21	21	210	4	12	5	59,25	2.477,00	0	19
gr24	24	276	4	14	6	355,73	1.038,00	1	22
ulysses16	16	120	3	9	4	2,18	56,21	0	13
ulysses22	22	231	4	13	5	25,54	58,92	0	20
fri26	26	325	5	15	6	62,34	724,00	0	26
bays29	29	406	5	17	7	103,11	1.371,00	0	28
burma14	14	91	2	8	4	0,41	22,41	0	10

Tabla 4.2: Resultados obtenidos para MIP con la Configuración 1

Instancia	N	A	N_X	N_T	N_{Op}	$Tiempo(s)$	VAL_{sol}	OP_{sol}	A_{sol}
gr17	17	136	3	6	8	2,18	921,00	2	12
gr21	21	210	4	8	9	7,69	2.029,00	1	16
gr24	24	276	4	9	11	11,40	744,00	0	16
ulysses16	16	120	3	6	7	1,12	23,77	0	10
ulysses22	22	231	4	8	10	7,39	53,59	0	15
fri26	26	325	5	10	11	30,03	516,00	0	21
bays29	29	406	5	11	13	45,74	1.179,00	0	22
burma14	14	91	2	4	7	0,27	18,14	0	7

Tabla 4.3: Resultados obtenidos para MIP con la Configuración 2

Instancia	N	A	N_X	N_T	N_{Op}	$Tiempo(s)$	VAL_{sol}	OP_{sol}	A_{sol}
gr17	17	136	3	3	11	0,42	858,00	1	8
gr21	21	210	4	4	13	2,00	1.655,00	2	14
gr24	24	276	4	4	16	2,54	488,00	1	13
ulysses16	16	120	3	3	10	0,45	18,36	0	7
ulysses22	22	231	4	4	14	1,73	47,69	0	12
fri26	26	325	5	5	16	9,00	345,00	0	16
bays29	29	406	5	5	19	11,11	825,00	0	16
burma14	14	91	2	2	10	0,09	13,70	0	4

Tabla 4.4: Resultados obtenidos para MIP con la Configuración 3

Instancia	N	A	N_X	N_T	N_{Op}	$Tiempo(s)$	VAL_{sol}	OP_{sol}	A_{sol}
gr17	17	136	8	4	5	3,01	452,00	0	34
gr21	21	210	10	5	6	14,45	1.425,00	1	53
gr24	24	276	12	6	6	48,70	505,00	0	74
ulysses16	16	120	8	4	4	2,78	42,60	0	35
ulysses22	22	231	11	5	6	18,19	48,52	0	62
fri26	26	325	13	6	7	111,51	345,00	0	85
bays29	29	406	14	7	8	116,56	850,00	0	102
burma14	14	91	7	3	4	0,80	10,96	0	25

Tabla 4.5: Resultados obtenidos para MIP con la Configuración 4

Instancia	N	A	N_X	N_T	N_{Op}	$Tiempo(s)$	VAL_{sol}	OP_{sol}	A_{sol}
gr17	17	136	8	6	3	6,29	1.108,00	0	36
gr21	21	210	10	8	3	34,63	2.003,00	0	55
gr24	24	276	12	9	3	92,23	685,00	0	77
ulysses16	16	120	8	6	2	4,77	46,32	0	37
ulysses22	22	231	11	8	3	58,08	19,34	0	65
fri26	26	325	13	10	3	209,26	406,00	0	89
bays29	29	406	14	11	4	Out Memory	-	-	-
burma14	14	91	7	5	2	1,59	11,58	0	28

Tabla 4.6: Resultados obtenidos para MIP con la Configuración 5

Instancia	N	A	N_X	N_T	N_{Op}	$Tiempo(s)$	VAL_{sol}	OP_{sol}	A_{sol}
gr17	17	136	8	1	8	0,27	161,00	0	30
gr21	21	210	10	2	9	2,26	1.010,00	0	49
gr24	24	276	12	2	10	5,26	269,00	0	70
ulysses16	16	120	8	1	7	0,22	4,76	0	30
ulysses22	22	231	11	2	9	2,76	8,13	0	58
fri26	26	325	13	2	11	5,94	207,00	0	81
bays29	29	406	14	2	13	9,78	454,00	0	94
burma14	14	91	7	1	6	0,14	4,78	0	23

Tabla 4.7: Resultados obtenidos para MIP con la Configuración 6

Instancia	N	A	N_X	N_T	N_{Op}	$Tiempo(s)$	VAL_{sol}	OP_{sol}	A_{sol}
gr17	17	136	11	0	6	No Aplica	-	-	-
gr21	21	210	14	1	6	1,01	159,00	0	91
gr24	24	276	16	1	7	1,76	124,00	0	122
ulysses16	16	120	11	0	5	No Aplica	-	-	-
ulysses22	22	231	15	1	6	1,33	3,65	0	107
fri26	26	325	18	1	7	2,59	164,00	0	155
bays29	29	406	20	1	8	4,49	110,00	0	192
burma14	14	91	9	0	5	No Aplica	-	-	-

Tabla 4.8: Resultados obtenidos para MIP con la Configuración 7

Instancia	N	A	N_X	N_T	N_{Op}	$Tiempo(s)$	VAL_{sol}	OP_{sol}	A_{sol}
gr17	17	136	11	2	4	1,09	277,00	0	58
gr21	21	210	14	3	4	7,16	1.067,00	1	97
gr24	24	276	16	3	5	10,87	326,00	0	125
ulysses16	16	120	11	2	3	0,94	8,13	0	58
ulysses22	22	231	15	3	4	9,66	41,02	0	111
fri26	26	325	18	3	5	18,30	243,00	0	158
bays29	29	406	20	4	5	65,83	523,00	0	197
burma14	14	91	9	2	3	0,51	8,32	0	40

Tabla 4.9: Resultados obtenidos para MIP con la Configuración 8

Instancia	N	A	N_X	N_T	N_{Op}	$Tiempo(s)$	VAL_{sol}	OP_{sol}	A_{sol}
gr17	17	136	11	4	2	3,42	648,00	0	61
gr21	21	210	14	5	2	21,60	1.138,00	0	98
gr24	24	276	16	6	2	57,21	560,00	0	129
ulysses16	16	120	11	4	4	2,71	15,33	0	61
ulysses22	22	231	15	5	5	25,33	44,07	0	114
fri26	26	325	18	6	6	93,63	305,00	0	160
bays29	29	406	20	7	7	Out Memory	-	-	-
burma14	14	91	9	3	2	0,97	8,56	0	41

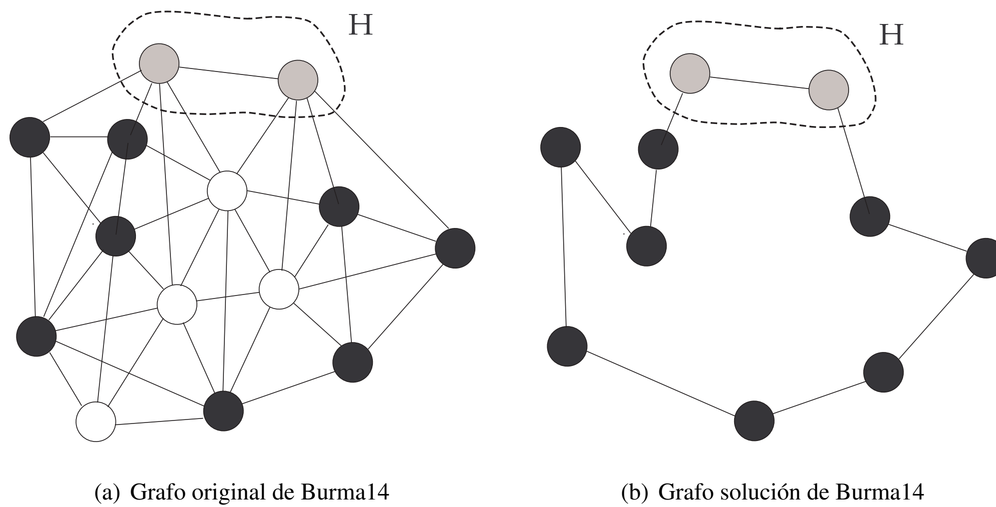
Tabla 4.10: Resultados obtenidos para MIP con la Configuración 9

4.2.1. Análisis de los Resultados

Antes de presentar el análisis de los resultados obtenidos, se debe aclarar que las pruebas se realizan sobre grafos que cuentan con menos de 30 nodos, ya que si tienen una mayor cantidad, el requisito de memoria que precisa CPLEX supera la capacidad del equipo donde se realizaron las pruebas. En la bibliografía consultada sobre el Problema de Steiner Generalizado, se pueden encontrar casos prácticos donde se ha resuelto dicho problema para grafos de 50 a 60 nodos, en nuestro caso, y debido a la necesidad de realizar Splitting para resolver el problema de nodo-conectividad, los grafos donde se aplica el modelo tienen el doble de nodos que el grafo original (entre 34 y 58 nodos). Inclusive se puede observar que en dos configuraciones de casos de prueba, el modelo para el grafo bays29 no pudo obtener un resultado debido al motivo mencionado anteriormente.

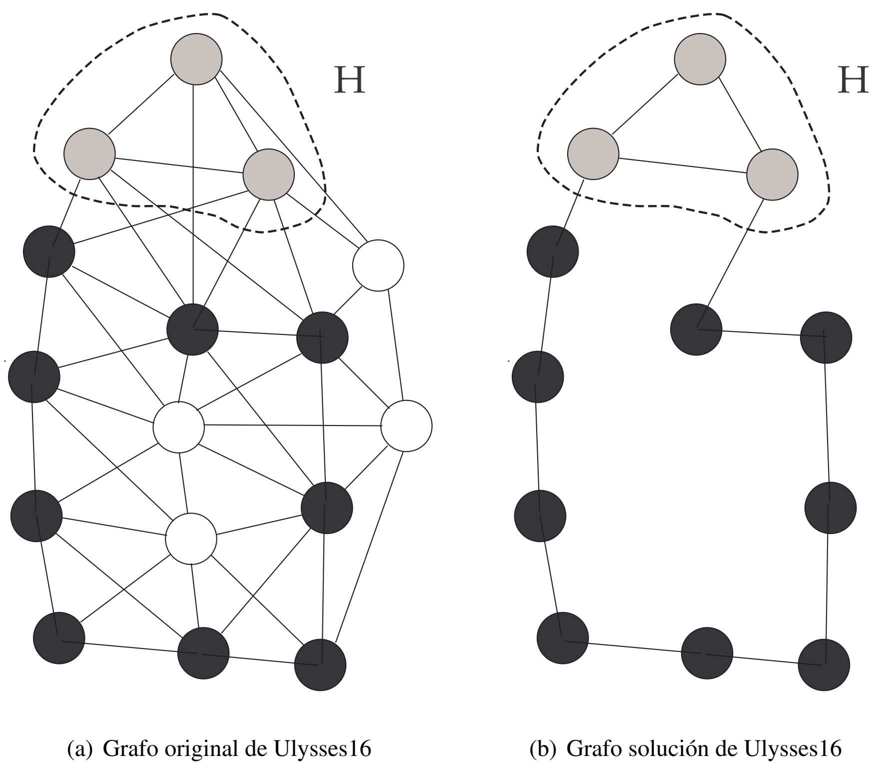
Analizando los datos obtenidos, se observa que la utilización de nodos opcionales (o nodos de Steiner) es muy baja y la misma se realiza principalmente en las tres primeras configuraciones, que es precisamente donde existe mayor cantidad de ellos. En relación a esta observación se destaca que las primeras configuraciones brindan resultados más interesantes, ya que el conjunto de nodos de X (la red existente), a diferencia de los nuevos nodos, por tener aristas con costos nulos incide en menor medida en la solución. Desde el punto de vista matemático esta observación posee una explicación y ésta consiste en que el espacio de soluciones factibles es mayor cuanto menor es el conjunto X.

En las Figuras 4.1, 4.2 y 4.3 se puede observar unos ejemplos de los grafos solución que se obtienen después de aplicar el Método Exacto. En la Figura 4.4 se puede apreciar una gráfica con los tiempos insumidos por el solver de CPLEX en resolver el problema para las nueve configuraciones.



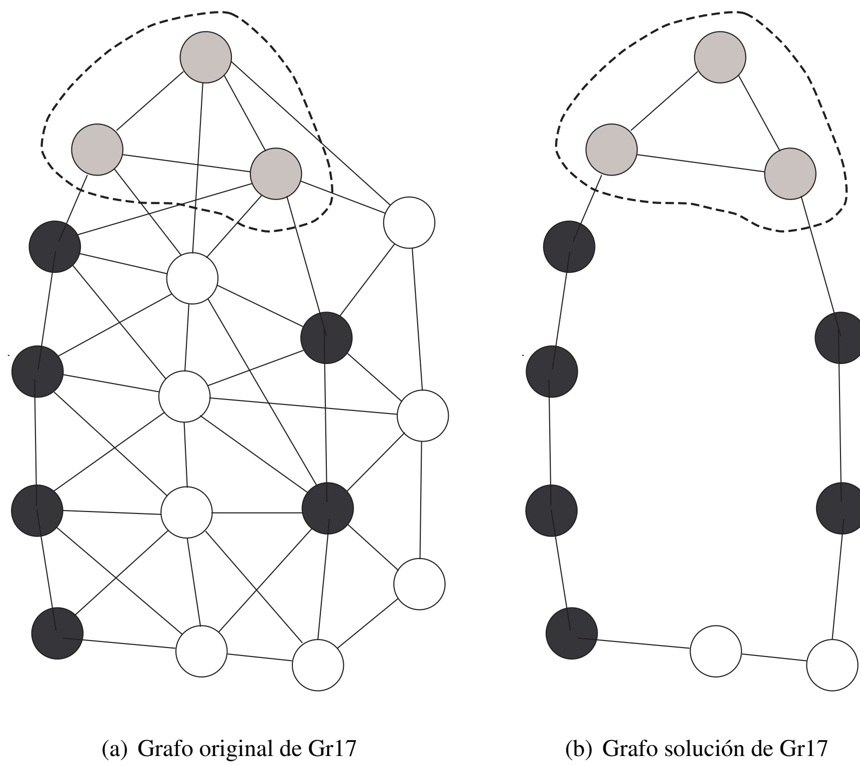
(a) Grafo original de Burma14 (b) Grafo solución de Burma14

Figura 4.1: Caso de prueba Burma14



(a) Grafo original de Ulysses16 (b) Grafo solución de Ulysses16

Figura 4.2: Caso de prueba Ulysses16 con configuración 1.



(a) Grafo original de Gr17 (b) Grafo solución de Gr17

Figura 4.3: Caso de prueba Gr17 con configuración 2.

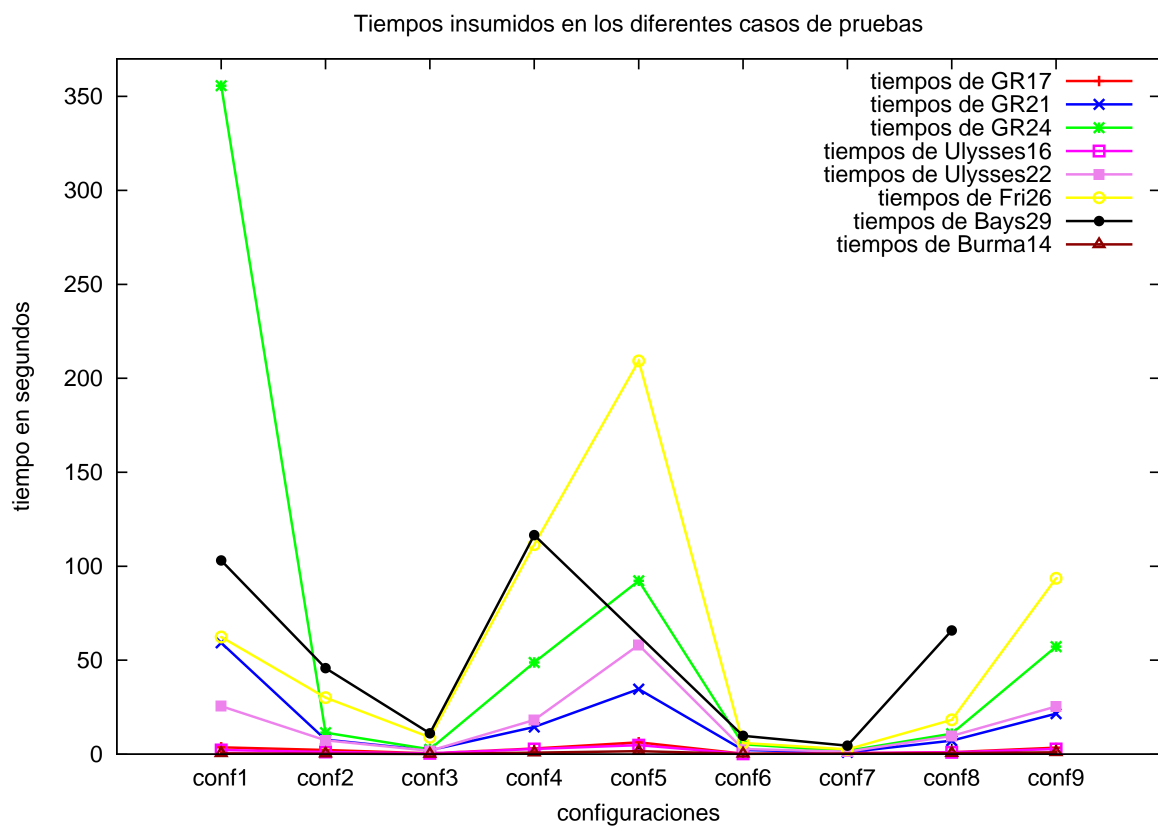


Figura 4.4: Gráfica con los tiempos insumidos por el Método Exacto en los casos de pruebas

Parte III

RESOLUCIÓN POR MEDIO DE VNS

Capítulo 5

La Metaheurística VNS

En este Capítulo, se dará una breve introducción al significado de heurísticas y la metaheurística VNS. En la Sección 5.2, se explica la motivación por la que se usó VNS y como se la aplico para la resolución del problema planteado.

5.1. ¿Que son las Heurísticas?

Con el fin de resolver problemas complejos con eficiencia, en ocasiones es necesario comprometer algunos requisitos de optimalidad y construir una estructura de control que no garantiza encontrar la mejor respuesta, pero que casi siempre encuentra una solución que es cercana a la óptima. De esta forma, surge la idea de heurística. La palabra heurística, tiene su origen de la palabra griega heuriken, que significa "descubrir", que es también origen de eureka, derivado de la famosa exclamación de Arquímedes, heurika ("lo encontré").

Las heurísticas son técnicas que buscan soluciones de buena calidad, es decir, permiten obtener un valor cercano al óptimo a un costo computacional razonable, aunque sin garantizar la optimalidad de las mismas. Debido a la existencia de problemas importantes, con un gran interés práctico que son difíciles de resolver, problemas que son de una complejidad polinomial y problemas de optimización que son NP-Hard, comienzan a surgir algoritmos capaces de ofrecer posibles soluciones, que aunque no consiguen el resultado óptimo, se acercan a este en un tiempo de cálculo razonable. Estos algoritmos, están basados en el conocimiento heurístico y por lo tanto reciben el nombre de algoritmos heurísticos.

Una metaheurística es un método heurístico para resolver un tipo de problema computacional general, usando los parámetros dados por el usuario sobre procedimientos genéricos y abstractos de una manera que se espera sea eficiente [62]. En definitiva, las metaheurísticas son estrategias inteligentes para diseñar o mejorar procedimientos heurísticos muy generales con un alto rendimiento.

La metaheurística, se aplica generalmente a problemas clasificados como NP-Hard o NP-Completos. Sin embargo, la metaheurística también podría aplicarse a problemas de optimi-

zación combinatoria en los cuales se sabe que existe una solución de tiempo polinómico, pero que no es alcanzable en la práctica.

Existen muchos ejemplos de metaheurísticas utilizadas en la actualidad, las cuales se pueden clasificar en dos grandes grupos:

Métodos de Trayectoria:

- Búsquedas locales y variantes (Local Search)
- Recocido Simulado (Simulated Annealing)
- Búsqueda Tabú (Tabú Search)
- GRASP (Greedy Randomize Adaptive Search Procedure)

Métodos basado en poblaciones:

- Computación Evolutiva (Evolutionary Computation)
- Colonias de Hormigas (Ant Colony Optimization)

A continuación, se explica en detalle la metaheurísticas VNS utilizada en la resolución del problema a abordar en esta tesis.

5.1.1. Variable Neighborhood Search

Variable Neighborhood Search (conocida por sus siglas en inglés VNS) es una metaheurística reciente para resolver problemas de optimización, cuya idea básica es el cambio sistemático de vecindad dentro de una búsqueda por vecindades [52, 53, 81, 82, 92]. VNS está basada en un principio simple, cambiar la estructura de entornos cuando la búsqueda local se estanca en un óptimo local. Se han realizado muchas extensiones, principalmente para obtener la solución de problemas de gran tamaño, pero en la mayoría de ellas, se ha hecho un esfuerzo por mantener la simplicidad del esquema básico.

5.1.1.1. Esquema Básico

Un problema de optimización consiste en encontrar, dentro de un conjunto X de soluciones factibles, la que optimiza una función $f(x)$. Si el problema es de minimización se formula de la siguiente manera:

$$\text{mín}\,\{f(x)/x \in X\} \tag{5.1}$$

La variable x representa una solución alternativa, f es la función objetivo y X es el espacio de soluciones factibles del problema. Una solución óptima x^* (o mínimo global) del problema

es una solución factible donde se alcanza el mínimo del problema 5.1.

Una estructura de vecindad en el espacio de soluciones X es una función $N : X \to 2^X$ que asocia a cada solución $x \in X$ un subconjunto de soluciones $N(x)$ denominado vecindad de x, cuyas soluciones se dicen *"vecinas"* de x. Las metaheurísticas de búsqueda local aplican una transformación o movimiento a la solución de búsqueda y por tanto utilizan, explícita o implícitamente, una estructura de vecindad. Denotemos por $N_k, k = 1, \ldots, k_{max}$, a un conjunto finito de estructuras de vecindad en el espacio X. Las vecindades N_k pueden ser inducidas por una o más métricas introducidas en el espacio de soluciones X. La mayoría de las heurísticas de búsqueda local usan una única estructura de entornos.

Una solución $x^* \in X$ es un mínimo global del problema 5.1 si no existe una solución $x \in X$ tal que $f(x) < f(x^*)$. Decimos que $x' \in X$ es un mínimo local con respecto a N_k, si no existe una solución $x \in N_k(x') \subseteq X$ tal que $f(x) < f(x')$. Una búsqueda local descendente cambia la solución actual por otra solución mejor de su vecindad, por tanto tiene el peligro de quedarse atascada en un mínimo local. Las metaheurísticas basadas en procedimientos de búsqueda local aplican distintas formas de continuar la búsqueda después de encontrar el primer óptimo local.

VNS está basada en tres principios simples:

1. Un mínimo local en una estructura de vecindad no lo es necesariamente en otra.

2. Un mínimo global es mínimo local con todas las posibles estructuras de vecindades.

3. Los mínimos locales para la misma o distinta estructura de vecindad están relativamente cerca.

Esta última observación, implica que los óptimos locales proporcionan información acerca del óptimo global, por tanto, es conveniente realizar estudios en las proximidades de este óptimo local hasta que se encuentre uno mejor.

Los principios 1 a 3 sugieren el empleo de varias estructuras de vecindades en las búsquedas locales para abordar un problema de optimización. El cambio de estructura de vecindad se puede realizar de forma determinística, estocástica, o determinística y estocástica a la vez.

A continuación explicaremos los diferentes tipos de procedimientos de VNS que se han utilizado para la resolución de los problemas de minimización.

5.1.1.2. VNS Descendente

Una búsqueda local determina iterativamente una mejor solución en la vecindad de la solución actual. La búsqueda de $greedy$ descendente consiste en reemplazar la solución actual por la mejor solución de su vecindad, mientras produzca una mejora. Si se realiza un cambio

de estructura de vecindad de forma determinística cada vez que se llega a un mínimo local, se obtiene el método de Variable Neighborhood Descent (denotado por VND). En el pseudocódigo 5 se muestra el VND planteado.

Seudocódigo 5 VND

1: Seleccionar el conjunto de vecindades N_k
2: Encontrar una solución inicial x
3: **Repetir**
4: $k \leftarrow 1$
5: **Repetir**
6: Encontrar la mejor solución x' de la k-ésima vecindad de x ($x' \in N_k(x)$)
7: **Si** $x' < x$ **entonces**
8: $x \leftarrow x'$
9: $k \leftarrow 1$
10: **Si no**
11: $k \leftarrow k + 1$
12: **Fin Si**
13: **Hasta** $k == k_{max}$
14: **Hasta** no se encuentra mejoras en la solución
15: **Retornar** x

En el paso 6, se puede apreciar como se obtiene una solución x' de forma determinística, la cual se repite cuando se reinicia el ciclo. A este paso se lo puede denominar de agitación. La solución final proporcionada por el procedimiento es un mínimo local con respecto a todas las k_{max} estructuras de vecindad y por lo tanto, la probabilidad de alcanzar un mínimo global, es mayor que usando una sola estructura. Como ya se mencionó, la mayoría de las heurísticas de búsqueda local usan en sus descensos solamente una vecindad y algunas veces dos ($k_{max} = 2$).

5.1.1.3. VNS Reducida

El método *Reduced Variable Neighborhood Search* (denotado por RVNS) se obtiene seleccionando soluciones aleatorias de $N_k(x)$ sin aplicar a continuación un descenso. En el pseudocódigo 6 se muestra el RVNS planteado.

RVNS es útil para instancias muy grandes de problemas en las que la búsqueda local es muy costosa. Como condición de parada se usa generalmente un máximo número de iteraciones entre dos mejoras.

5.1.1.4. VNS Básica

El método de Basic Variable Neighborhood Search (denotado por BVNS) combina cambios determinísticos y aleatorios de estructura de entornos. En el pseudocódigo 7 se muestra el

BVNS planteado.

Seudocódigo 6 RVNS

1: Seleccionar el conjunto de vecindades N_k
2: Encontrar una solución inicial x
3: Seleccionar una condición de parada
4: **Repetir**
5: $k \leftarrow 1$
6: **Repetir**
7: Generar al azar una solución x' de la k-ésima vecindad de x ($x' \in N_k(x)$)
8: **Si** $x' < x$ **entonces**
9: $x \leftarrow x'$
10: $k \leftarrow 1$
11: **Si no**
12: $k \leftarrow k + 1$
13: **Fin Si**
14: **Hasta** $k == k_{max}$
15: **Hasta** cumpla la condición de parada
16: **Retornar** x

Seudocódigo 7 BVNS

1: Seleccionar el conjunto de vecindades N_k
2: Encontrar una solución inicial x
3: Seleccionar una condición de parada
4: **Repetir**
5: $k \leftarrow 1$
6: **Repetir**
7: Generar al azar una solución x' de la k-ésima vecindad de x ($x' \in N_k(x)$)
8: Aplicar algún método de búsqueda local con x' como solución inicial, siendo x'' el mínimo local obtenido
9: **Si** $x'' < x$ **entonces**
10: $x \leftarrow x''$
11: $k \leftarrow 1$
12: **Si no**
13: $k \leftarrow k + 1$
14: **Fin Si**
15: **Hasta** $k == k_{max}$
16: **Hasta** cumpla la condición de parada
17: **Retornar** x

La condición de parada puede ser, por ejemplo, el tiempo máximo de CPU permitido, el máximo número de iteraciones, o el máximo número de iteraciones entre dos mejoras. Se puede apreciar, que la solución x' se genera al azar en el paso 7 para evitar el ciclo, lo que puede

ocurrir si se utiliza cualquier regla determinística.

Para obtener una descripción más detallada de esta metaheurística se puede consultar en las referencias [51–55, 81, 82, 92]. En las mismas, se podrá encontrar información sobre las diferentes formas de extender VNS para que cuente con algunas características adicionales, tales como: *VNS con Descomposición*, *VNS Sesgado*, *VNS Paralela* e híbridos con otras metaheurísticas (como ser búsqueda Tabú y GRASP).

5.2. Motivación de VNS

VNS ha mostrado ser una metaheurística muy eficaz en encontrar un óptimo, muy cercano al global, en problemas similares al planteado en este proyecto. Por dicha razón se utiliza VNS para poder encontrar un óptimo en la ampliación de una red que es 2-nodo-conexa de forma tal que la nueva red hallada sea 2-nodo-conexa y que cubra el nuevo conjunto de nodos obligatorios (terminales). Se analizaran diferentes mecanismos para las fases de optimización por vecindades del algoritmo de VNS, que a partir de una solución factible inicial, por medio de movimientos hacia soluciones vecinas (cambiar la estructura de entornos), alcancen soluciones factibles de menor costo y que no puedan ser mejoradas por posteriores movimientos. Por eso VNS tiene la ventaja de poder mejorar la solución factible obtenida mediante el agregado con facilidad de nuevas vecindades en el algoritmo planteado.

En este trabajo la utilización de VNS permitió realizar una búsqueda local en una vecindad utilizando una variante del Método Exacto planteado en la Sección 2.3.1, logrando de esta forma una combinación de una heurística con la aplicación de un Método Exacto. Esta variante del Método Exacto es aplicada a subgrafos de la solución factible obtenida hasta ese momento, con el requisito de que cada uno de los subrago sea 2-nodo-conexo. De esta manera se puede obtener el óptimo local de la estructura de vecindad definida por cada subgrafo, lo cual mejoraría el resultado final de la solución.

Junto con VNS, se utilizó VND como procedimiento de mejoras en las búsquedas locales. Esto permite que cada vecindad se pueda explorar de forma alternativa, de modo que si se logra una mejora en alguna de estas vecindades se comienza con la secuencia desde el principio. En virtud de que no se eligió ningún criterio especifico para ordenar las vecindades, se seleccionó al azar cuál es la primera vecindad en ejecutar, luego se sigue la secuencia hasta que se haya completado el total de vecindades. Una ventaja de utilizar VND es que el agregar o quitar vecindades permite ver como se mejora o empeora el potencial del algoritmo VNS, con el fin de encontrar un mejor solución. En el Capítulo de resultados se muestra un cuadro que indica cuales son las vecindades más eficientes en encontrar un mejor solución.

A continuación explicaremos brevemente cuales son los algoritmos utilizados para la construcción de la solución factible inicial y las diferentes vecindades utilizadas en la implementación del VND.

5.2.1. Solución Inicial

Para poder obtener una solución factible inicial se debe tener como restricción que para todos los nodos terminales $t \in T$ tiene que haber en G, por lo menos, dos caminos nodos disjuntos hacia cualquier elemento $x \in X$. Por lo tanto se analizaron dos algoritmos posibles que permitan cumplir con esta restricción: Dijkstra [33] y Bhandari [9]. En la Sección 6.1.1, se puede encontrar una descripción más detallada de cada uno de estos algoritmos. En virtud de que Bhandari utiliza el algoritmo de Dijkstra en una versión modificada devuelve los dos caminos nodos-disjuntos de menor costo entre dos nodos, y teniendo en cuenta que esta solución es mejor que aplicar dos veces el algoritmo de Dijkstra, se decidió utilizar el algoritmo de Bhandari como base para obtener una solución factible inicial. A partir de esta solución, que es la utilizada en el VNS, se aplica VND para encontrar una solución de costo mínimo que mejore la solución factible inicial sin perder la factibilidad de la misma.

En la Sección 6.1, se explica de forma detallada la implementación de la solución inicial para la resolución del problema.

5.2.2. Vecindades

Para la implementación de VND se definieron seis diferentes tipos de vecindades, de las cuales tres trabajan con los nodos opcionales, tratando de minimizar el costo del grafo solución actual, reduciendo e intercambiando este tipo de nodos. En el caso de que la solución inicial obtenida no tenga nodos opcionales, estas vecindades no se aplican ya que la búsqueda local que realizan es en el entorno de estos tipos de nodos.

Antes de definir las vecindades se harán algunas definiciones auxiliares que serán usadas en la descripción de las vecindades propuestas.

Definición 5.2.1 *Se define **ciclo simple** a un camino en el que no se repite ningún nodo a excepción del primero que aparece dos veces, como principio y fin del camino.*

Definición 5.2.2 *Dada una solución factible a una instancia del ASTNSNP, un **key-node** es todo nodo opcional al cual inciden aristas de dicha solución y cuyo grado es igual o mayor a 3 en la misma.*

Definición 5.2.3 *Dada una solución factible a una instancia del ASTNSNP, un **key-path** es cualquier camino simple de la misma, cuyos extremos son ambos terminales, o ambos key-nodes, o uno terminal y el otro key-node; o uno perteneciente a la red fija y el otro terminal o key-node; y cuyos nodos intermedios son nodos opcionales con grado 2.*

Notación 5.2.4 *Sea G_{sol} una solución factible de ASTNSNP. Si cada arista de G_{sol} pertenece a un camino con extremos un nodo de la red fija y otro nodo nuevo terminal, entonces es posible descomponer G_{sol} en key-paths, es decir, en un conjunto de key-paths en el cual cada arista de G_{sol} pertenece sólo a un key-path. Se denota $K(G_{sol}) = (p_1 .... p_n)$ como la descomposición de G_{sol} en key-paths.*

Definición 5.2.5 *Dada una instancia del ASTNSNP, una solución factible G_{sol}, y un nodo $v \in G_{sol}$ que es terminal o key-node o perteneciente a la red fija, definimos como **key-tree** asociado a v como el árbol de G_{sol} que se forma mediante la unión de todos los key-paths que parten desde v.*

Las definiciones previas son inspiradas en las definiciones de key-node, key-path y de key-tree usada en [98]. Se presentará una instancia de ejemplo basada en el grafo de la Figura 5.1. En el grafo se puede apreciar los nodos de la red fija $H1, H2, H3, H4$ de color gris, los nodos terminales $T1, T2, T3$ de color negro y los nodos opcionales v, u de color blanco. La Figura 5.2 muestra una solución factible del problema. Como se puede observar en la solución existe un key-node, un key-path y un key-tree.

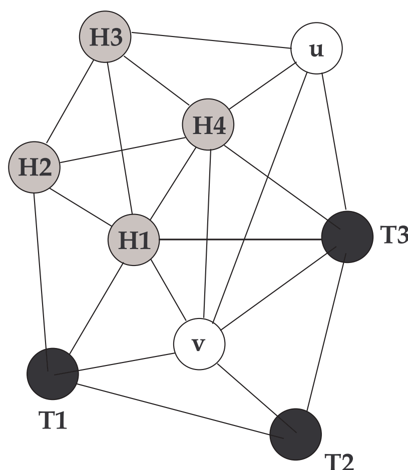


Figura 5.1: Ejemplo de una instancia de ANTNSNP.

Las vecindades definidas en este trabajo son las siguientes:

- **Optimización por reemplazo de key-path**: Se reemplaza un key-path con extremos u y v por 2 caminos nodos disjuntos con los mismos extremos pero cuyo costo en conjunto sea menor al costo del key-path.

- **Agregar Aristas$\backslash$ Reducir Redundantes**: Se plantea la modificación del conjunto de aristas del grafo solución $G_{sol} = (V_{sol}, E_{sol})$, mediante el agregado de una nueva arista y posteriormente la eliminación de las que quedan redundantes.

- **Intercambio de 2 Aristas entre 4 Nodos**: Dado el grafo solución $G_{sol} = (V_{sol}, E_{sol})$, dos nodos adyacentes $x, y \in V_{sol}$ y dos aristas $(w, x), (y, z) \in E_{sol}$ adyacente a los nodos x, y se itercambian dichas aristas por dos aristas $(w, y), (z, x) \notin E_{sol}$ para obtener una solución mejor a la actual.

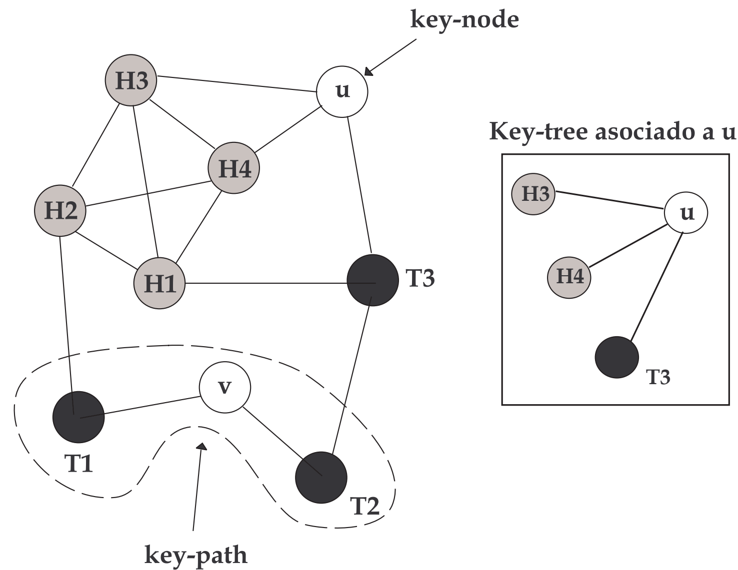


Figura 5.2: Solución del grafo ejemplo en la Figura 5.1.

- **Optimización por reducción de key-tree**: Se reemplaza un key-tree, con nodo raíz v y un conjunto de nodos hojas S, por pares de caminos nodos disjuntos cuyos origen son el nodo v y destino cada nodo hoja perteneciente a S.

- **Optimización por reducción del key-tree con nodo raíz opcional**: Se reemplaza un key-tree, con nodo raíz un key-node y un conjunto de nodos hojas S, por un árbol con nodo raíz opcional que no pertenezca a la solución y cuyas hojas sea el conjunto S.

- **Optimización por Método Exacto**: Se seleccionan subgrafos Q_i de tamaños razonables (menor a 16 nodos) del grafo G_{sol} para aplicar una variante del Método Exacto definido en la Sección 2.3.1.1. Al aplicar este método a cada uno de los subgrafos Q_i se obtiene como resultado subgrafos Q_{sol_i} 2-nodo-conexo de costo mínimo del grafo Q_i. Luego, los subgrafos Q_i son reemplazados por el subgrafo solución Q_{sol_i} en el grafo G_{sol}.

En el siguiente Capítulo, se explicará en profundidad la composición del Algoritmo VNS, explicando la fase de construcción de la solución inicial que se elaboró y los diferentes tipos de vecindades que se plantearon para la resolución del problema. También se detallan los pseudocódigos de los Algoritmos de VNS y VND realizados.

Capítulo 6

Resolución del Problema Mediante VNS

En este Capítulo, se describe el enfoque de VNS en detalle, primero se explica la construcción de una solución inicial factible para el ASTNSNP. Posteriormente en la Sección 6.2, se explica las distintas vecindades realizadas y las diferentes técnicas de búsqueda aplicadas en ellas y por último en la Sección 6.3 se detalla el algoritmo VNS aplicado para la resolución del problema.

6.1. Construcción de una Solución Factible Inicial

En la búsqueda de una solución factible inicial del problema, se utiliza un algoritmo para obtener los dos caminos nodos disjuntos que sumen menor costo para todos los nodos del conjunto de terminales T hacia algún nodo del conjunto X. Iterativamente, los caminos que se van obteniendo se agregan al grafo solución y de esta forma se genera una solución factible inicial una vez se hayan integrado a la solución todos los nodos de T. En la Sección 6.1.2 se puede apreciar mas en detalle el algoritmo planteado. A continuación se explica los algoritmos de caminos mínimos investigados.

6.1.1. Algoritmos de camino mínimo

Los algoritmos de caminos mínimos tratan de hallar la ruta más corta o de mínimo costo entre dos puntos. Este costo puede ser la distancia entre los puntos origen y destino o bien cualquier otro tipo de métrica, como el tiempo o el costo económico asociado a cada ruta. Dentro del conjunto de estos métodos, se ha considerado la aplicación del algoritmo de Dijkstra [33] y la modificación realizada sobre el mismo por Bhandari [9].

6.1.1.1. Algoritmo de Dijkstra

Es un algoritmo empleado para la determinación del camino más corto dentro de un grafo desde un vértice origen hacia un vértice destino. El grafo es dirigido y con costos asociados a cada arista. La idea en este algoritmo, consiste en ir explorando todos los caminos más cortos que parten del vértice origen y que llevan a todos los demás vértices, cuando se obtiene el camino más corto desde el vértice origen al resto de vértices que componen el grafo, el algoritmo se detiene.

Básicamente se parte de un grafo $G = (V, E)$ entre los cuales hay un origen a y un destino z. A cada arista de dicho grafo que une dos vértices, u y v, hay asociado un costo $c(u, v)$. Si no existe la arista entonces el costo $c(u, v) = \infty$.

El objetivo es mantener actualizado el conjunto S de vértices visitados, para los cuales se conoce el camino más corto y ampliar este conjunto hasta que S sea el conjunto total de vértices, de manera que al final se obtiene $S = V$. Para ello se etiqueta cada vértice z con $d(z)$, que es el costo del camino más corto ya encontrado.

A continuación se describen los pasos del algoritmo de Dijkstra.

Inicialización

- Sea $S = \{a\}$,$d(a) = 0$.
- Para todos los vecinos de a, $c(i) = c(a, i)$ con $i \in \Gamma_a$ (conjunto de vecinos de a). En caso que i no sea vecino de a, $c(i) = \infty$
- $S = V \setminus \{a\}$
- $Predecesor(i) = a, \forall i \in S$

Iteración

1. Elegir el vértice $j \in S$ (conjunto de vértices visitados) tal que su costo sea mínimo.
2. $S = S \setminus \{j\}$
3. Si $j = z$ se termina la búsqueda. En caso contrario continúa.
4. Para todos los vértices i vecinos de j, si $d(i) < d(j) + c(i, j)$ entonces $d(i) = d(j) + c(i, j)$ y $Predecesor(i) = j$
5. Vuelve al paso 1

El algoritmo de Dijkstra es una especialización de la búsqueda de costo uniforme y como tal, no funciona en grafos con aristas de costo negativo, ya que al elegir siempre el nodo con distancia menor, pueden quedar excluidos de la búsqueda nodos que en próximas iteraciones bajarían el costo general del camino al pasar por una arista con costo negativo.

6.1.1.2. Algoritmo de Bhandari

El Algoritmo de Dijkstra [33] es un método eficiente para encontrar el camino de costo mínimo entre dos vértice en un grafo. Sin embargo, a la hora de encontrar dos caminos vértice disjuntos dentro de un grafo, cuyo costo en conjunto sea mínimo, conduce a soluciones no optimales. Para resolver este problema existe otro algoritmo llamado Algoritmo de Bhandari [9], que utiliza dos etapas del camino más corto de Dijkstra y a continuación, combina los resultados para encontrar los dos caminos vértices disjuntos con menor costo total combinados.

Los pasos que realiza el algoritmo de Bhandari para encontrar los dos mejores caminos vértices disjuntos entre un par de vértice del grafo son los siguientes:

1. Se ejecuta el algoritmo de Dijkstra modificado para un par dado de vértices A, Z. Como se muestra en la Figura 6.1(a) se asume que A es el vértice origen y Z es el vértice destino.

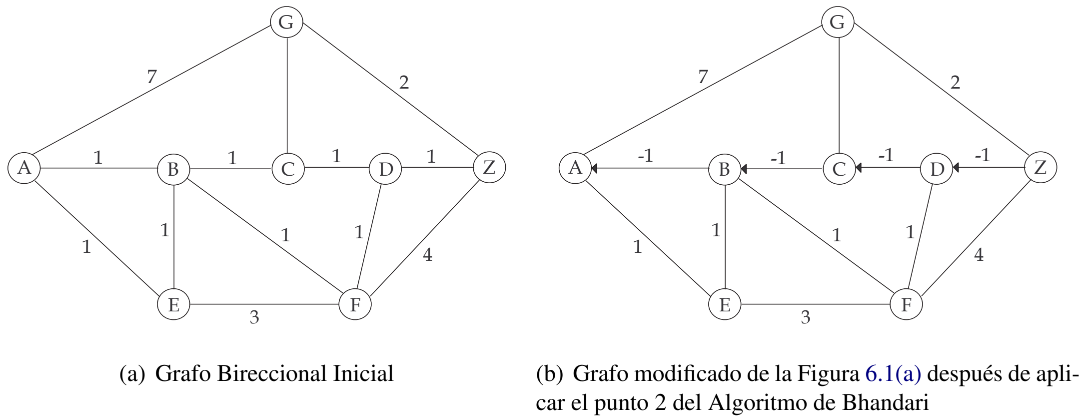
2. Reemplazar cada arista del camino obtenido en el paso anterior por un arista dirigida con sentido inverso, es decir, con dirección hacia el nodo origen A y con costo negativo, como se observa en la Figura 6.1(b).

3. En este paso se aplica la técnica de separación de vertices explicada en la Sección 2.3. Dividir cada vértice del camino v en dos subvértices v_1, v_2 unidos por una arista dirigida de costo cero. La dirección de esta arista es con sentido al vértice origen A. Reemplazar cada arista externa al camino conectadas a los vértices v del camino por dos aristas dirigidas en direcciones opuestas con los mismo costos, y estas aristas se las conecta a los dos nuevos subvértices, como se muestra en la Figura 6.2.

4. Ejecuta de nuevo el algoritmo de Dijkstra modificado en el nuevo grafo modificado.

5. En el camino obtenido en el paso anterior, se quitan las aristas con costo cero y se unen los dos subvertices para formar el vértice original. También se reemplazan las aristas dirigidas por las aristas originales de costos positivos.

6. En este paso se quitan las aristas repetidas de las dos caminos encontrados y de esta forma se obtienen los dos mejores caminos vértice disjuntos de menor costo.



(a) Grafo Bireccional Inicial

(b) Grafo modificado de la Figura 6.1(a) después de aplicar el punto 2 del Algoritmo de Bhandari

Figura 6.1: Ejecución del paso 1 y 2 del Algoritmo de Bhandari

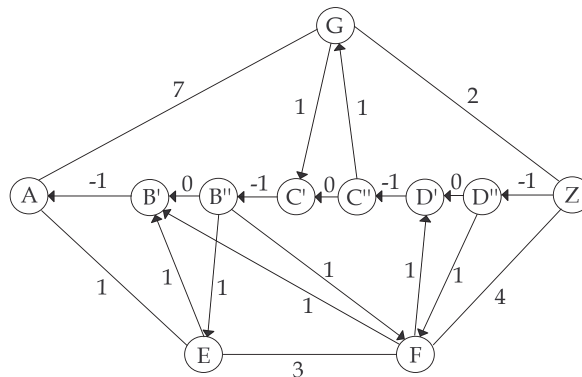


Figura 6.2: Grafo obtenido después de hacer la separación de vertices en el Algoritmo de Bhandari

En la Figura 6.1(a) de ejemplo, los caminos hallados con el Algoritmo de Dijkstra modificado son: $ABCDZ$ y $AEFDCGZ$. En el paso 6 del Algoritmo de Bhandari se quita la arista en común a los dos caminos (D, C), de esa forma hallando los dos caminos nodos disjuntos $ABCGZ$ y $AEFDZ$.

Algoritmo Dijkstra modificado

El algoritmo que se muestra a continuación es una variante del algoritmo original de Dijkstra. Es diferente en el paso 2 en el cual analiza todos los vecinos del vértice seleccionado en el paso 1. Se define a $d(i)$ como la distancia del vértice i al vértice inicial A y a $P(i)$ como los predecesores de i.

Inicialización

- $d(A) = 0, d(i) = c(A, i)$ si $i \in \Gamma_A$ y $d(i) = \infty$ en caso contrario ($\Gamma_i \equiv$ conjunto de los vecinos del vértice i y $c(i, j) =$ es el costo de la arista (i, j))

- $P(i) = A \, \forall i \in \Gamma_A$

- $S = \Gamma_A$

Iteración

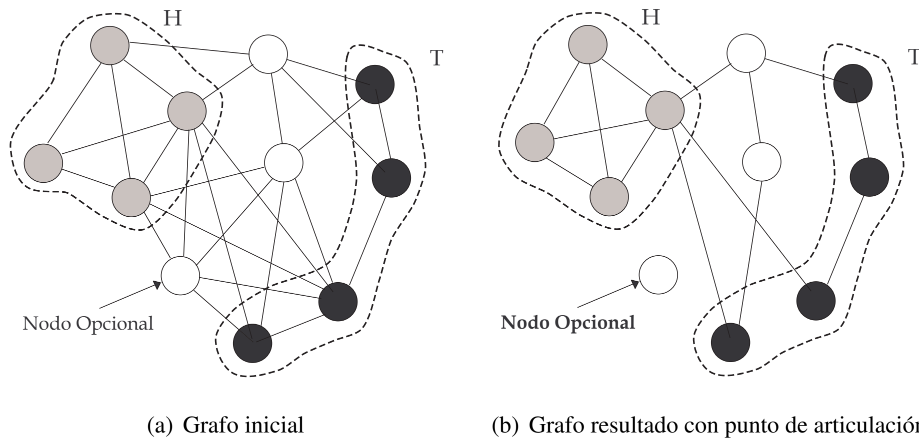
1.
   - Buscar $j \in S$ tal que $d(j) = \min d(i), i \in S$
   - $S = S \backslash \{j\}$
   - Si $j = Z$ (Z el vértice terminal) FIN
   - En caso contrario, voy al paso 2

2.
   - $\forall i \in \Gamma_i$, si $d(j) + c(j, i) < d(i)$ entonces $d(i) = d(i) + c(j, i)$, $P(i) = j$ y $S = S \cup \{i\}$
   - ir al paso 1

Después de la inicialización, el algoritmo alterna entre el paso 1 y 2. En cada iteración, el vértice con el menor costo es seleccionado del conjunto S. El algoritmo hace la búsqueda haciendo un movimiento a la vez y termina cuando el vértice seleccionado del conjunto S es igual a Z.

6.1.2. Solución Factible Inicial

La idea del algoritmo de construcción de la solución factible inicial para el ASTNSNP, que se puede observar más adelante en el Algoritmo 6.1.8, consiste en agregar un nodo ficticio especial z y aristas con costos 0 desde z hacia todos los nodos de X y después aplicar iterativamente el algoritmo de Bhandari [9], para obtener los 2 caminos nodos disjuntos, con origen cada nodo $t \in T$ y con destino el nodo ficticio z. La función de este nodo es evitar que se produzca un punto de articulación en los nodos del conjunto X cuando se obtienen los dos caminos disjuntos por medio de Bhandari, de esta forma cada camino pasa por nodos de X diferentes. En la Figura 6.3 se puede apreciar el problema de la existencia del punto de articulación y en la Figura 6.4 se observa la solución propuesta por medio de la incorporación del nodo ficticio z al grafo original G. Dado que el nodo z simplemente tiene como objetivo que no se genere puntos de articulación al computar dos caminos nodos disjuntos desde todo nodo de T hacia H, se conecta z con los nodos $x \in X$ por intermedio de aristas con costo cero.



(a) Grafo inicial (b) Grafo resultado con punto de articulación

Figura 6.3: Solución inicial incorrecta por la existencia del punto de articulación.

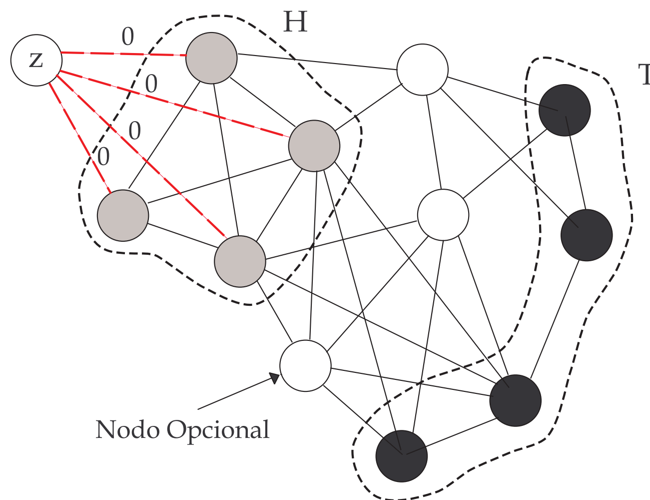
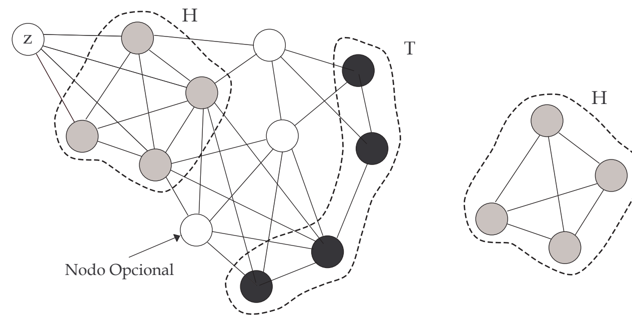


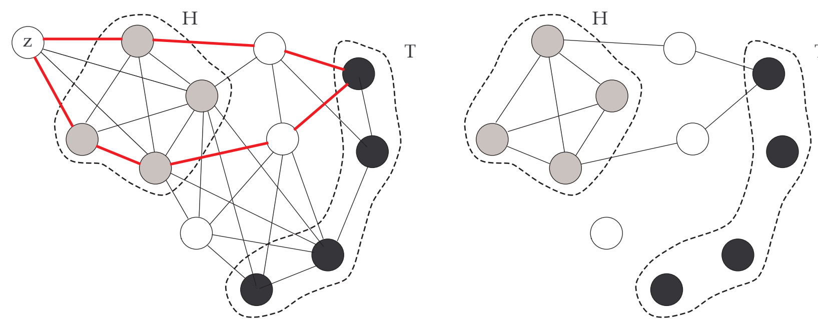
Figura 6.4: Grafo inicial con nodo ficticio agregado para solucionar el problema de generar puntos de articulación en la solución final

Al principio el grafo solución inicial G_{sol} esta compuesto por los nodos del conjunto X y las aristas que conectan dichos nodos, o sea el grafo que representa la red existente $H = (X, A)$ como se muestra en la Figura 6.5(b). Iterativamente, de los caminos nodos disjuntos que se van obteniendo por medio del Algoritmo de Bhandari, con origen cada nodo $t \in T$ y destino el nodo ficticio z, se agregan los nodos y aristas que conforman dichos caminos al grafo solución inicial, como se muestra en las Figuras 6.6 a 6.9. De esta manera se puede construir una solución factible inicial para el ASTNSNP; en la Figura 6.9(b) se observa el grafo factible inicial construido.



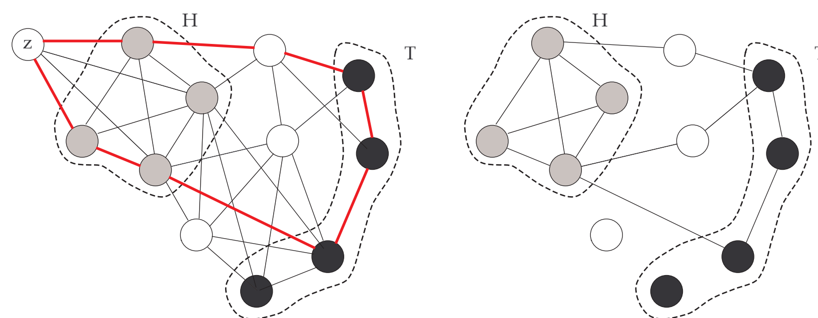
(a) Grafo inicial con nodo ficticio agregado (b) Grafo solución inicial

Figura 6.5: **Grafo Inicial** - Construcción de la solución inicial - Antes de aplicas Bhandari.



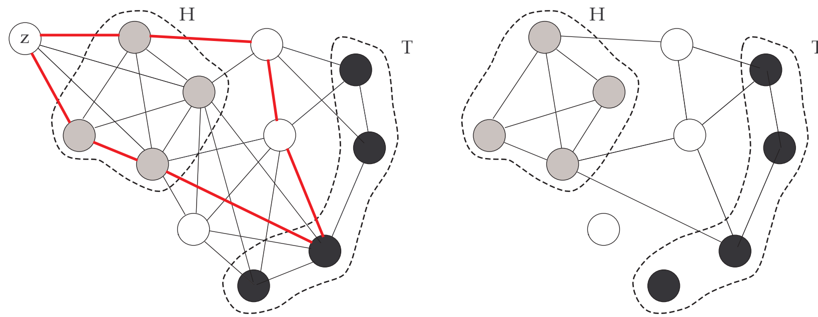
(a) Grafo inicial con 2 primeros caminos hallados (b) Grafo solución inicial con los 2 primeros caminos agregados

Figura 6.6: **Grafo Inicial** - Construcción de la solución inicial - 2 primeros caminos agregados.



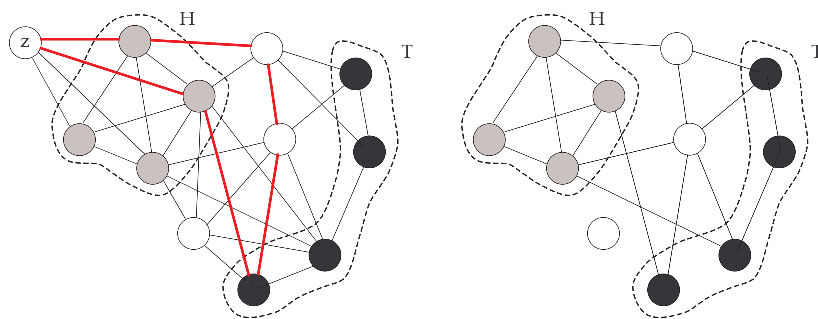
(a) Grafo inicial con 2 nuevos caminos hallados (b) Grafo solución inicial con los 2 nuevo caminos agregados

Figura 6.7: **Grafo Inicial** - Construcción de la solución inicial - Mas caminos agregados



(a) Grafo inicial con 2 nuevo caminos hallados (b) Grafo solución inicial con los 2 nuevos caminos agregados

Figura 6.8: **Grafo Inicial** - Construcción de la solución inicial - 2 nuevos caminos agregados.



(a) Grafo inicial últimos 2 nuevo caminos hallados (b) Solución inicial construida

Figura 6.9: **Grafo Inicial** - Construcción de la solución inicial - Solución factible inicial.

En el Algoritmo 6.1.8 se presenta el pseudocódigo para la construcción de la solución factible inicial para el ASTNSNP. El mismo toma como entrada un grafo G (grafo original de la librería TSP) y devuelve como resultado una solución factible inicial G_{sol} que contiene la red ya existente H y todos los nodos de T. En la línea 1 se inicializa G_{sol} con la red ya existente H, en las líneas 2 a 3 se inicializan las variables $camino1$ y $camino2$ con el conjunto vacío. En estas variables se van a guardar los dos caminos nodos disjuntos obtenidos con el Algoritmo de Bhandari. En la línea 4 se agrega el nodo ficticio z al grafo y aristas de costo 0 desde z hacia todos los nodos $x \in X$ en G_{sol}. En las líneas 5 a 11 se itera en los nodos de T que no fueron agregados a G_{sol}. En la línea 7 se aplica el Algoritmo de Bhandari para encontrar los dos caminos nodos disjuntos de menor costo entre el nodo ficticio z y el nodo $\hat{t} \in T$. Los caminos obtenidos se guardan en las variables $camino1$ y $camino2$ respectivamente. Luego las aristas y nodos de estos caminos son agregados al grafo solución G_{sol}. El algoritmo finaliza cuando todos los nodos de T son agregados al grafo solución G_{sol}.

Algoritmo 6.1.8 Solución Inicial

Entrada: G // Grafo original

Salida: G_{sol} // Solución factible inicial para el ASTNSNP

1. $G_{sol} = (X, A)$
2. $camino1 = \emptyset$
3. $camino2 = \emptyset$
4. $G \leftarrow Genera_Grafo_Bhandari(G, z)$ // Agrego el nodo z adyacente a todos los nodos de X
5. **Mientras** $\exists t \in T/t \notin G_{sol}$ **Hacer**
6. $\hat{t} \leftarrow$elijo aleatoriamente $t \in T/t \notin G_{sol}$
7. $[camino1, camino2] \leftarrow Alg_Bhandari(G, \hat{t}, z)$
8. $G_{sol} = G_{sol} \cup camino1$
9. $G_{sol} = G_{sol} \cup camino2$
10. $camino1 = \emptyset$ // Inicializo el conjunto con el vacío
11. $camino2 = \emptyset$
12. **Fin Mientras**

6.2. Definición de Vecindades

En esta Sección se presentan en detalle las seis vecindades aplicadas en el Algoritmo de VNS para el ASTNSNP.

6.2.1. Optimización por reemplazo de key-path

La idea de esta primer vecindad consiste en el reemplazo de un key-path con extremos u y v por 2 caminos nodos disjuntos con los mismos extremos pero cuyo costo en conjunto sea menor al costo del key-path.

Definición 6.2.1 *Dada una instancia del ASTNSNP y una solución factible G_{sol} definimos como vecindad a toda solución factible G'_{sol} que se pueda obtener mediante el reemplazo de cualquiera de sus key-path, con origen y destino u, v respectivamente, por 2 caminos nodos disjuntos cuyo costo en conjunto sea menor al costo del key-path y con origen y destino u, v respectivamente.*

Este algoritmo, que se puede observar en el Algoritmo 6.2.1, consiste en que dado una solución $G_{sol} = (V_{sol}, E_{sol})$ y el conjunto de key-paths $K(G_{sol}) = (p_1...p_n)$ se puede obtener una mejor solución mediante el reemplazo del key-path $p_i \in K$ con extremos u, v por 2 caminos nodos disjuntos p', p'' con los mismos extremos u, v pero con los nodos y aristas intermedios pertenecientes al subgrafo inducido de G originado por $E(p_i) \cup (E(G) \backslash E(G_{sol})) \cup E'$, siendo E' las aristas de G_{sol} pero con costo cero. El reemplazo ocurre cuando el $cost(p') + cost(p'') < cost(p_i)$. Para llevar a cabo el reemplazo, primero se debe generar el subgrafo G_{aux} inducido por los nodos del grafo original G y las aristas $E'' = E(p_i) \cup (E(G) \backslash E(G_{sol})) \cup E'$, siendo

E' las aristas de G_{sol}, sin las aristas del key-path p, pero con costo cero. Luego, se obtienen los 2 caminos nodos disjuntos con menor costo p', p'' y con extremos u, v mediante la aplicación del Algoritmo de Bhandari. Por último, si la suma de los costos de p' y p'' es menor al costo del key-path p_i entonces se reemplaza el key-path por los 2 caminos. La vecindad planteada se puede observar en la Figura 6.10.

Observación 1: Para no dejar de lado la posibilidad de emplear las aristas de la solución actual G_{sol} se define el conjunto de arista E' con costos 0. El costo en dichas aristas se lo considera cero para que los caminos hallados reutilicen en lo posible las aristas ya usadas en la solución actual G_{sol}.

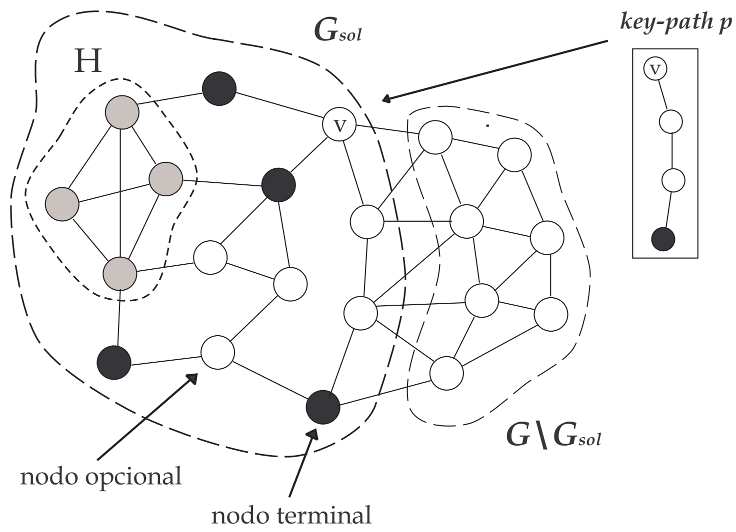
Observación 2: Cabe destacar que para obtener los 2 caminos nodos disjuntos entre u, v se utiliza el Algoritmo de Bhandari planteado en la Sección 6.1.1.2

Algoritmo 6.2.1 Optimización por reemplazo de key-path

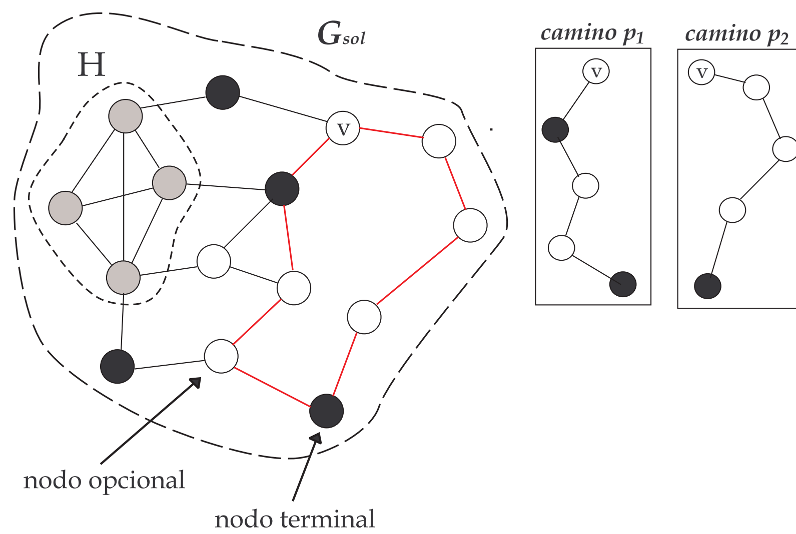
Entrada: G_{sol} // Solución Actual
Salida: G_{sol} // Solución con costo menor o igual al de G_{sol} actual

1. $G_a = G_{sol}$
2. $p' = \emptyset$
3. $p'' = \emptyset$
4. Sea $K(G_a) = Descomposicion_KeyPaths(G_a)$ // el conjunto de key-paths de G_a
5. **Mientras** $|K| > 0$ **Hacer**
6. sorteo p perteneciente a K
7. $G_{aux} \leftarrow$ el subgrafo inducido por $E(p) \cup (E(G) \setminus E(G_a)) \cup E'$ en G original, siendo E' las aristas de G_{sol} pero con costo cero.
8. sean u, v los extremos del key-path p
9. $[p', p''] \leftarrow$ AlgBhandari(G_{aux}, u, v)
10. **Si** $cost(p') + cost(p'') < cost(p)$ **entonces**
11. $G_{sol} = (G_a \setminus \{p\}) \cup \{p'\} \cup \{p''\}$
12. $G_a = G_{sol}$
13. $K(G_a) = Descomposicion_KeyPaths(G_a)$
14. **Si no**
15. $G_a = G_{sol}$
16. $K = K \setminus \{p\}$
17. **Fin Si**
18. $p' = \emptyset$
19. $p'' = \emptyset$
20. **Fin Mientras**

En el Algoritmo 6.2.1 se presenta el pseudocódigo de la vecindad "Optimización por Remplazo de key-path". El mismo toma como entrada el grafo solución actual G_{sol} y devuelve un grafo solución con costo menor o igual al grafo solución actual G_{sol}. En las líneas 2 y 3 se inicializan los caminos p', p'' con el vacío. En la línea 4 se obtiene la descomposición en



(a) Grafo G_{aux} generado e identificación del key-path p



(b) Intercambio del key-path p por los caminos p_1 y p_2

Figura 6.10: Intercambio de key-path por 2 caminos disjuntos de menor costo.

key-paths $K(G_a) = (p_1...p_n)$. En las líneas 5 a 20 se itera sobre los key-paths pertenecientes a $K(G_a)$. En la línea 6 se sortea aleatoriamente un key-path p del conjunto $K(G_a)$ que todavía no haya sido analizado. En la línea 7 se genera el subgrafo G_{aux} inducido por los nodos del grafo original y las aristas $E'' = E(p_i) \cup (E(G)\ E(G_{sol})) \cup E'$, siendo E' las aristas de G_{sol}, sin las aristas de key-path p, pero con costo cero. En la línea 9 se obtienen los 2 caminos nodos disjuntos p' y p''. Para la obtención de los 2 caminos nodos disjuntos de menor costo se utiliza el algoritmo de Bhandari con k = 2, siendo k la cantidad de caminos. En la línea 10 se compara la suma de los costos de los caminos p' y p'' con el costo del key-path p. En caso que la suma de los costos de los dos caminos sea menor se actualiza el grafo solución G_{sol} por el nuevo grafo $(G_a \backslash \{p\}) \cup \{p'\} \cup \{p''\}$, como se muestra en la línea 11, y luego se recomputa nuevamente la descomposición en key-paths, ya que este cambio en el grafo solución puede generar nuevos key-paths. En caso contrario se elimina el key-path p del conjunto K. El algoritmo finaliza cuando se analizan todos los key-paths de la solución actual y no es posible mejorar el costo de ninguno de ellos.

6.2.2. Agregar Aristas\ Reducir Redundantes

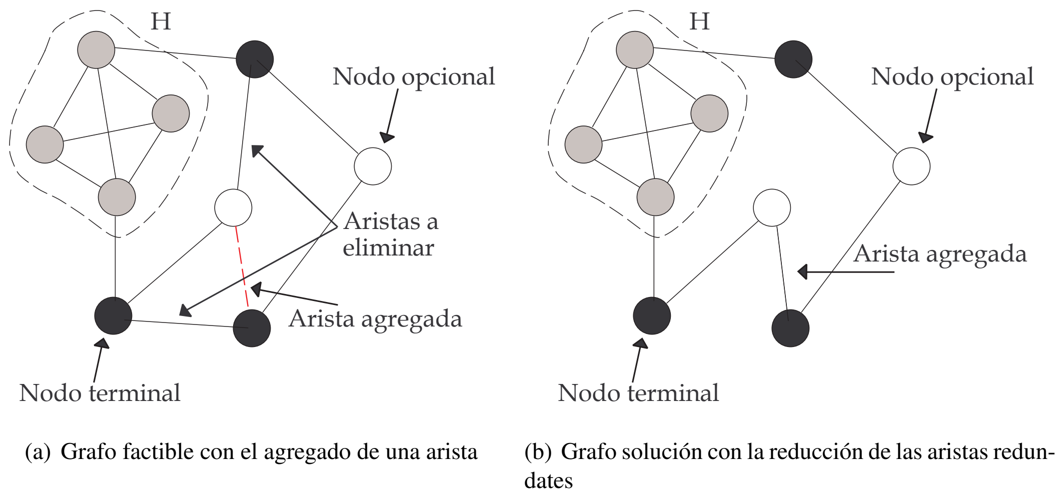
En esta vecindad se plantea la modificación del conjunto de aristas del grafo solución $G_{sol} = (V_{sol}, E_{sol})$, mediante el agregado de una nueva arista y posteriormente la eliminación de las aristas que quedan redundantes.

Definición 6.2.2 *Dada una instancia del ASTNSNP y una solución factible G_{sol} definimos como vecindad a toda solución factible G'_{sol} que se pueda obtener mediante el agregado de una nueva arista no pertenecientes a G_{sol} y posteriormente la eliminación de las aristas que quedan redundantes en G_{sol}.*

La idea de esta vecindad es, dado $G_{sol} = (V_{sol}, E_{sol})$, agregar aristas $e \notin E_{sol}$ entre dos nodos $x, y \in V_{sol}$. El agregado de dichas aristas puede determinar la existencia de aristas redundantes, es decir, aristas que al eliminarlas se mantiene la factibilidad de la solución y además se reduce el costo del grafo solución G_{sol}. Para lograr esto, se seleccionan las aristas $e = (i, j) \notin E_{sol}$, siendo $i, j \notin X$, y se agregan dichas aristas a G_{sol}, obteniendo así el grafo G_a. Luego se aplica el Algoritmo 6.2.3 de eliminación de aristas redundantes al grafo G_a. Si el costo del grafo resultante de aplicar dicho algoritmo mejora el costo del grafo solución entonces se define como nuevo grafo solución a G_a. En la Figura 6.11 se muestra un ejemplo de como funciona la vecindad planteada.

Se destaca que, luego de la inclusión de una nueva arista e, si no se eliminan aristas redundantes entonces la arista e queda como redundante, en cuyo caso la inclusión de esta no mejoraría la solución G_{sol} por lo cual se descarta su inclusión.

En el Algoritmo 6.2.2 se presenta el pseudocódigo de la vecindad “Agregar Aristas\ Reducir Redundantes”. El mismo toma como entrada el grafo solución actual G_{sol} y devuelve un



(a) Grafo factible con el agregado de una arista

(b) Grafo solución con la reducción de las aristas redundates

Figura 6.11: Vecindad agregar arista y reducir redundates.

Algoritmo 6.2.2 Agregar Arista\ Reducir Redundantes

Entrada: G_{sol} // Solución Actual

Salida: G_{sol} // Solución con costo menor igual al de G_{sol} actual

1. $G_a = G_{sol}$
2. **Para** $i = 1 \ldots N$
3.   **Para** $j = i + 1 \ldots N$
4.   **Si** $((i, j) \notin E_{sol})$ and $(i, j \notin X)$ **entonces**
5. $G_a = G_a \cup \{(i, j)\}$
6. $Eliminar_Aristas_Redundantes(G_a, i, j)$
7.   **Si** costo de G_a < costo de G_{sol} **entonces**
8. $G_{sol} = G_a$
9.   **Si no**
10. $G_a = G_{sol}$
11.   **Fin Si**
12.   **Fin Si**
13.   **Fin Para**
14. **Fin Para**

grafo solución con costo menor o igual al grafo solución actual G_{sol}. En la línea 1 se inicializa G_a con el grafo solución G_{sol}. En las líneas 2 y 3 se recorren todos los pares de nodos $i, j \in V_{sol}$ del grafo G_{sol}. Las líneas 4 a 12, se ejecutan si la arista (i, j) no pertenece a E_{sol} y además si los nodos $i, j \notin X$. Si se cumplen estas condiciones, se agrega la arista (i, j) al grafo G_a y por medio de la función $Eliminar_Aristas_Redundantes$ se eliminan las aristas redundantes que se generan. Esta función obtiene los nodos adyacentes a i y si se cumple que el grado de dichos nodos es mayor a dos, entonces se elimina la arista que une el nodo i con el nodo adyacente. Se realiza el mismo procedimiento para el nodo j. En esta función, después de haber eliminado las aristas redundantes, se verifica si el grafo solución queda 2 nodo conexo. Esta verificación es necesaria ya que los cambios establecidos en el grafo G_a pueden romper la 2 nodo conectividad. En caso que no sea 2 nodo conexo la función devuelve el grafo sin las modificaciones establecidas. El pseudocódigo de la función $Eliminar_Aristas_Redundantes$ se muestra en el algoritmo 6.2.3. En las líneas 7 a 11, se actualiza el grafo solución si el nuevo grafo G_a tiene menor costo que el G_{sol} actual. Finaliza el algoritmo cuando se probaron todas las aristas $(i, j) \notin E_{sol}$ y al reducir las aristas redundantes por cada arista (i, j) agregada no mejora el costo del grafo solución actual.

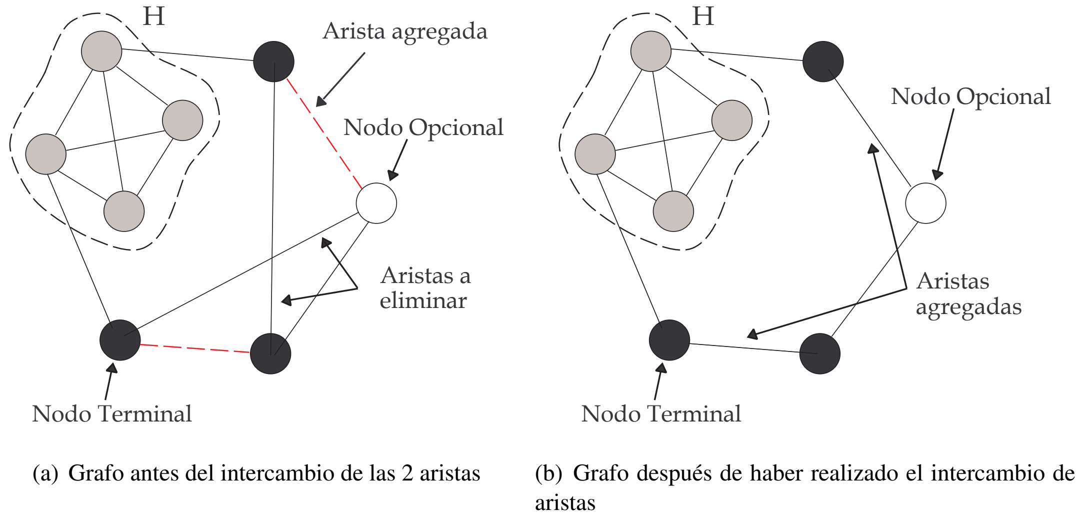
Algoritmo 6.2.3 Eliminar Aristas Redundantes

Entrada: G_a, i, j
Salida: G_a //Grafo con la eliminación de aristas realizadas

1. $S \leftarrow Adyacentes(G_a, i)$ //Se obtiene los nodos adyacentes a i
2. **Mientras** $\mid S \mid > 0$ **Hacer**
3.   sorteo s
4.   **Si** $s \neq j$ and $grado(s) > 2$ **entonces**
5. $Borrar_Arista(s, i, G_a)$ // Borra la arista (s, i) en G_a
6.   **Fin Si**
7. $S = S \backslash \{s\}$
8. **Fin Mientras**
9. $S \leftarrow Adyacentes(G_a, j)$ //Se obtiene los nodos adyacentes a j
10. **Mientras** $\mid S \mid > 0$ **Hacer**
11.   sorteo s
12.   **Si** $s \neq i$ and $grado(s) > 2$ **entonces**
13. $Borrar_Arista(s, j, G_a)$ // Borra la arista (s, j) en G_a
14.   **Fin Si**
15. $S = S \backslash \{s\}$
16. **Fin Mientras**

6.2.3. Intercambio de dos Aristas entre cuatro Nodos

Definición 6.2.3 *Dada una instancia del ASTNSNP y una solución factible $G_{sol} = (V_{sol}, E_{sol})$ definimos como vecindad a toda solución factible G'_{sol} que se pueda obtener por medio del intercambio de las aristas (w, x) y (y, z) pertenecientes a E_{sol}, siendo $w, x, y, z \in V_{sol}$, por 2*



(a) Grafo antes del intercambio de las 2 aristas

(b) Grafo después de haber realizado el intercambio de aristas

Figura 6.12: Proceso del intercambio de 2 aristas entre 4 nodos.

aristas $(w, y), (z, x) \notin E_{sol}$ *tal que la arista* (x, y) *exista en* E_{sol}.

En esta vecindad se intercambia sólo una arista incidente al nodo x con otra arista incidente al nodo y por cada pareja de nodos x e y adyacentes. Esto implica que dado un grafo $G_{sol} = (V_{sol}, E_{sol})$ y cuatro nodos $w, x, y, z \in V_{sol}$ conectados mediante aristas $(w, x), (x, y), (y, z) \in E_{sol}$, se puede transformar dicho grafo G_{sol} en un grafo G'_{sol} a través del cambio de aristas entre los nodos de forma tal que los mismos queden conectados por medio de las aristas $(w, y), (x, z) \notin E_{sol}$ y la arista (x, y). Por tanto G'_{sol} queda definido por $G'_{sol} = (V_{sol}, E'_{sol})$ siendo $E'_{sol} = \{(w, y)\} \cup \{(x, z)\} \cup (E_{sol} \backslash (\{(w, x)\} \cup \{(y, z)\}))$. Un ejemplo de los intercambios de aristas realizados se puede observar en la Figura 6.12. En el Algoritmo 6.2.4 se muestra la implementación de la vecindad planteada.

Observación: Para el caso particular de nuestro problema se debe considerar que w, x, y, z no pueden pertenecer todos al conjunto X ya que el costo de sus aristas es cero y son fijos (los enlaces ya fueron construidos), por lo tanto no tiene sentido realizar el intercambio de aristas.

En el Algoritmo 6.2.4 se presenta el pseudocódigo de la vecindad "Intercambio de dos Aristas entre cuatro Nodos". El mismo toma como entrada el grafo solución actual G_{sol} y devuelve un grafo solución con costo menor o igual al grafo solución actual G_{sol}. En las líneas 2 a 3 se recorren todos los nodos del grafo G_a para obtener todas las aristas de dicho grafo. La condición de que las aristas $(x, y) \in E_a$, que se puede ver en la línea 4, es para que se cumpla que el intercambio de una arista incidente se produzcan entre nodos adyacentes entre si. En el caso que estos nodos no fueran adyacentes no se podría hacer el mencionado intercambio de una arista porque se perdería la factibilidad de la solución generada. En las líneas 5 y 6 se obtienen las aristas adyacentes de x e y respectivamente. En las líneas 7 y 8 se iteran sobre las aristas adyacentes para que posteriormente en la línea 9 por medio de la función

Algoritmo 6.2.4 Intercambio de 2 Aristas entre 4 Nodos

Entrada: G_{sol} // Solución Actual

Salida: G_{sol} // Solución con costo menor igual al de G_{sol} actual

1. $G_a = G_{sol}$
2. **Para todo** $x \in V_a$
3.   **Para todo** $y \in V_a$
4.   **Si** $(x, y) \in E_a$ **entonces**
5. $A_x \leftarrow adyacentes(x, G_a)$
6. $A_y \leftarrow adyacentes(y, G_a)$
7.   **Para todo** $w \in A_x \ and \ w \neq y$
8.   **Para todo** $z \in A_y \ and \ z \neq x$
9. $G_a \leftarrow Intercambiar_Adyacentes(x, y, w, z, G_a)$
10.   **Si** costo de G_a < costo de G_{sol} **entonces**
11. $G_{sol} = G_a$
12.   **Si no**
13. $G_a = G_{sol}$
14.   **Fin Si**
15.   **Fin Para**
16.   **Fin Para**
17.   **Fin Si**
18.   **Fin Para**
19. **Fin Para**

$Intercambiar_Adyacentes(x, y, w, z, G_a)$ se intercambia la arista de la iteración. Esta función, primero obtiene las aristas (w, x) y (z, y). Luego, se eliminan dichas aristas en el grafo G_a y por último se generar las nuevas aristas (z, x) y (w, y) en el grafo G_a. El pseudocódigo de la función $Intercambiar_Adyacentes(x, y, w, z, G_a)$ se muestra en el Algoritmo 6.2.5. Las líneas 10 a 14 se utilizan para actualizar el grafo solución si el nuevo grafo generado tiene menor costo que el G_{sol} actual. El algoritmo finaliza cuando se probaron todas las aristas $(x, y) \in E_{sol}$ y al intercambiar las aristas adyacentes a dichos nodos no se mejora el costo del grafo solución actual.

En la función $Intercambiar_Adyacentes$, después de haber realizado el intercambio de aristas entre los nodos, se verifica si el grafo solución queda 2 nodo conexo. Esta verificación es necesaria ya que los cambios establecidos en el grafo G_a pueden romper la 2 nodo conectividad. En caso que no sea 2 nodo conexo la función devuelve el grafo sin las modificaciones establecidas.

Algoritmo 6.2.5 Intercambio de Aristas Adyacentes

Entrada: x, y, w, z, G_a
Salida: G_a //Grafo con el intercambio de aristas realizado
1. $Borrar_Adyacentes(w, x, G_a)$ //Se eliminan las aristas (w, x) y (z, y)
2. $Borrar_Adyacentes(z, y, G_a)$
3. $G_a \leftarrow Agregar_Adyacentes(G_a, x, z)$ //Se agregan las nuevas aristas (z, x) y (w, y)
4. $G_a \leftarrow Agregar_Adyacentes(G_a, y, w)$

6.2.4. Optimización por reducción de key-tree

Esta vecindad consiste en reemplazar un key-tree, con nodo raíz v y conjunto de nodos hojas S, por pares de caminos nodos disjuntos cuyos origen son el nodo v y destino cada nodo hoja perteneciente a S.

Definición 6.2.4 *Dada una instancia del ASTNSNP y una solución factible G_{sol} definimos como vecindad a toda solución factible G'_{sol} que se pueda obtener mediante el reemplazo de cualquier de sus key-trees, con nodo raíz v y conjunto de nodos hojas S, por pares de caminos nodos disjuntos cuyos origen son el nodo v y destino cada nodo hoja perteneciente a S. Cada key-path que "cuelga" del nodo v es reemplazado por uno de estos pares de caminos nodos disjuntos.*

La idea de esta vecindad es, dado una solución factible $G_{sol} = (V_{sol}, E_{sol})$, y su descomposición en key-trees $KT(G_{sol}) = (t_1...t_n)$ se puede obtener una mejor solución por medio del reemplazo de un key-tree $t_j \in KT$, con raíz v y conjunto de nodos hojas S_j, por 2 caminos nodos disjuntos $p_{v,i}, p'_{v,i} \forall i \in S$ con nodo origen v y nodo destino cada elemento $i \in S_j$. En la Figura 6.13 se puede observar los pares de caminos nodos disjuntos por cada elemento del conjunto S. El reemplazo ocurre cuando $\sum_{i \in S_j} cost(p_{v,i}) + cost(p'_{v,i}) < cost(t_j)$. Para llevar

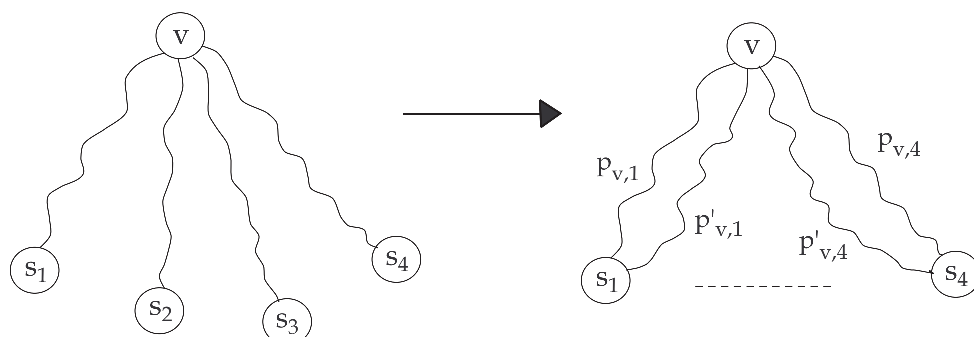


Figura 6.13: Intercambio de un key-tree por pares de caminos nodos disjuntos

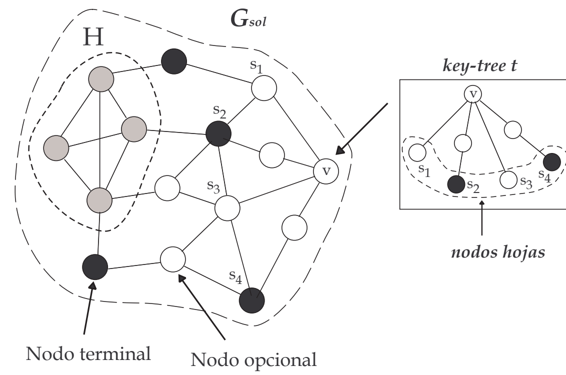
a cabo dicho reemplazo, primero se debe de generar el subgrafo G_{aux} inducido por los nodos del grafo original G y las aristas $E'' = E(t_j) \cup (E(G) \backslash E(G_{sol})) \cup E'$, siendo E' las aristas de G_{sol}, sin las aristas de key-tree t_j, pero con costo cero. Luego se obtiene los 2 caminos nodos disjuntos de menor costo $p_{v,i}, p'_{v,i}$ con origen en el nodo raíz v y con destino cada elemento i perteneciente al conjuntos de nodos hojas S_j. Cabe señalar que a un nodo se le dice hoja cuando el grado de dicho nodo en el key-tree analizado es igual a uno. Por último, si la sumatoria de la suma de todos los caminos $p_{v,i}$ y $p'_{v,i}$ es menor al costo del key-tree t_j entonces se reemplaza el key-tree t_j por los caminos encontrados. El pseudocódigo del algoritmo se puede observar en el Algoritmo 6.2.6 y en la Figura 6.14 se puede apreciar un ejemplo de la vecindad planteada.

Observación 1: Al igual que en la vecindad de reemplazo de key-path, para no dejar de lado la posibilidad de emplear las aristas de la solución actual G_{sol} se define el conjunto de aristas E' con costos 0. El costo en dichas aristas se lo considera cero para que los caminos hallados reutilicen en lo posible las aristas ya usadas en la solución actual G_{sol}.

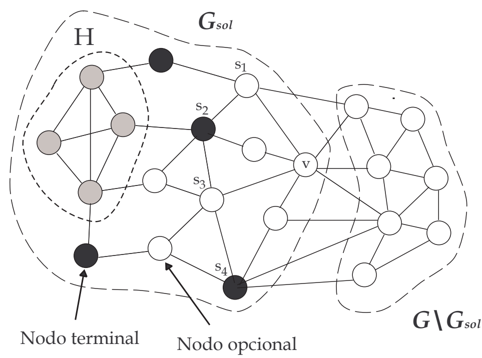
Observación 2: Al igual que en la vecindad de reemplazo de key-path, para obtener los 2 caminos nodos disjuntos entre la raíz del key-tree y cada nodo hoja del mismo se utiliza el Algoritmo de Bhandari planteado en la Sección 6.1.1.2.

Cabe destacar que en el ejemplo planteado en la Figura 6.14 sólo se necesitó 2 caminos nodos disjuntos p'_{v,s_1}, p'_{v,s_4}. Esto sucede, debido a que existe un camino entre los elementos del conjunto S, el camino formado por $(s_1, s_2), (s_2, s_3), (s_3, s_4)$, y cuando se aplicó el Algoritmo de Bhandari para hallar los pares de caminos nodos disjuntos con nodo origen v y destino cada elemento del conjunto S, se obtuvo el mismo par de caminos nodos disjuntos p'_{v,s_1}, p'_{v,s_4}. Por dicha razón el grafo solución resultante es el mostrado en la Figura 6.14(d).

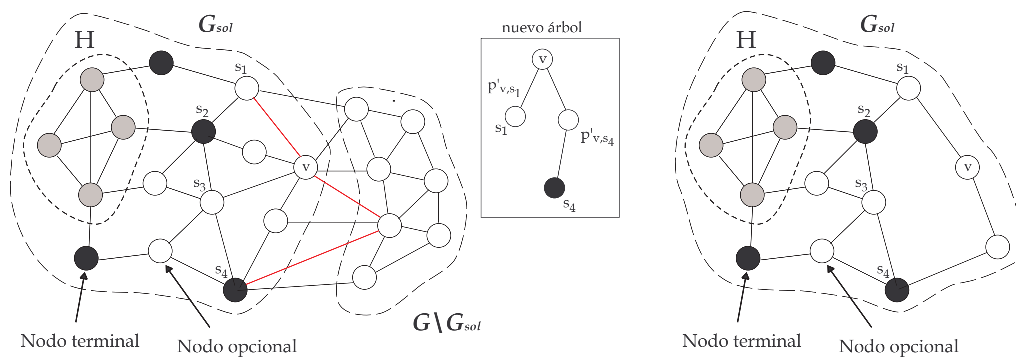
En el Algoritmo 6.2.6 se presenta el pseudocódigo de la vecindad "Optimización por reducción de key-tree". El mismo toma como entrada el grafo solución actual G_{sol} y devuelve un grafo solución con costo menor o igual al grafo solución actual G_{sol}. En la línea 2 se inicializa la variable $caminos$ con el vacío, la cual guardará los caminos obtenidos por el Algoritmo de



(a) Grafo solución con la identificación de un key-tree



(b) Grafo G_{aux} generado para realizar la búsqueda de los caminos p y p'



(c) Obtención de los dos caminos disjuntos p y p'

(d) Grafo solución final después de cambiar el key-tree por p y p'

Figura 6.14: Proceso de reducción de key-tree por caminos disjuntos alternativos.

Bhandari. En la línea 3 se obtiene el conjunto de key-trees $KT(G_a) = (t_1...tn)$. En las líneas 4 a 19 se itera sobre los key-trees pertenecientes a $KT(G_a)$. En la línea 5 se elige aleatoriamente un key-tree t del conjunto $KT(G_a)$. En la línea 6 se genera el subgrafo G_{aux} inducido por los nodos del grafo original y las aristas $E'' = E(t_i) \cup (E(G) \backslash E(G_{sol})) \cup E'$, siendo E' las aristas de G_{sol}, sin las aristas del key-tree t, pero con costo cero. En las líneas 7 y 8 se obtiene el nodo raíz v y el conjunto de nodos hoja del key-tree respectivamente. En la línea 9 se obtiene el conjunto de caminos nodos disjuntos de menor costo. Para la obtención de los caminos nodos disjuntos se utiliza el algoritmo de Bhandari con k = 2, siendo k la cantidad de caminos para cada par de nodos raíz, hoja. En la línea 10 se compara la sumatoria de los costos de los pares de caminos con el costo del key-tree t. En caso que la sumatoria de los costos de los pares de caminos sea menor se actualiza el grafo solución G_{sol} por el nuevo grafo $(G_a \backslash \{t\}) \cup \{caminos\}$, como se muestra en la línea 11, y se recomputa nuevamente el conjunto de key-trees K, como se muestra en la línea 13, ya que este cambio en el grafo solución puede generar nuevos key-trees. En caso contrario se elimina el key-tree t del conjunto KT. El algoritmo finaliza cuando se analizan todos los key-trees de la solución actual y no es posible mejorar el costo de ninguno de ellos.

Algoritmo 6.2.6 Optimización por reemplazo de key-trees

Entrada: G_{sol} // Solución Actual
Salida: G_{sol} // Solución con costo menor igual al de G_{sol} actual

1. $G_a = G_{sol}$
2. $caminos = \emptyset$
3. Sea $KT(G_a) = Descomposicion_KeyTrees(G_a)$ el conjunto de key-trees de G_a
4. **Mientras** $\mid KT \mid > 0$ **Hacer**
5. sorteo t perteneciente a KT
6. $G_{aux} \leftarrow$ el subgrafo inducido por $E(t_i) \cup (E(G) \backslash E(G_a)) \cup E'$ en G original, siendo E' las aristas de G_{sol} pero con costo cero.
7. sea v el nodo raíz del key-tree t
8. sea S_t el conjunto de nodos hojas del key-tree t
9. $caminos \leftarrow Alg_Bhandari(G_{aux}, v, S_t)$
10. **Si** $\sum cost(caminos) < cost(p)$ **entonces**
11. $G_{sol} = (G_a \backslash \{t\}) \cup \{caminos\}$
12. $G_a = G_{sol}$
13. $KT(G_a) = Descomposicion_KeyTrees(G_a)$
14. **Si no**
15. $G_a = G_{sol}$
16. $KT = KT - \{t\}$
17. **Fin Si**
18. $caminos = \emptyset$
19. **Fin Mientras**

6.2.5. Optimización por reducción del key-tree con nodo raíz opcional

Esta vecindad consiste en reemplazar un key-tree, con nodo raíz un key-node y un conjunto de nodos hojas S, por un árbol con nodo raíz opcional que no pertenezca a la solución y cuyas hojas sea el conjunto S.

Definición 6.2.5 *Dada una instancia del ASTNSNP y una solución factible G_{sol} definimos como vecindad a toda solución factible G'_{sol} que se pueda obtener mediante el reemplazo de cualquiera de sus key-trees, con nodo raíz un key-node y un conjunto de nodos hojas S (S son los extremos de los key-paths que penden de la raíz del key-tree), por un árbol con nodo raíz un nodo opcional que no pertenezca a G_{sol} y cuyas hojas sea los nodos del conjunto S.*

Esta vecindad, la cual se puede observar en el Algoritmo 6.2.7, consiste en que dada una solución factible $G_{sol} = (V_{sol}, E_{sol})$ y su descomposición en key-trees $KT(G_{sol}) = (t_1...t_n)$, con raíz un key-node v y un conjunto de nodos hojas S por cada elemento de KT, se puede obtener una mejor solución por medio del reemplazo de un key-tree $t_j \in KT$ por un árbol t'_j de menor costo, con raíz un nodo opcional $o' \notin V_{sol}$ y con todos los nodos hojas de t'_j igual a los nodos hojas de t_j. Si se denomina S'_j al conjunto de nodos hoja de t'_j entonces tiene que suceder que $S'_j \equiv S_j$.

El análisis de cada nodo opcional $o' \notin V_{sol}$ como nodo raíz del nuevo árbol sustituto surge porque la potencialidad de mejorar la solución actual que tiene el reemplazo de key-path, realizada en la vecindad "Optimización por reemplazo de key-path", se ve limitada por casos donde es necesario trabajar con estructuras de vecindad más grandes. Por ejemplo, en la Figura 6.15 se puede observar un ejemplo donde en la parte (a) muestra una solución factible de costo 25, en la cual no es posible reemplazar ningún key-path por otro de menor costo. Pero el key-tree con nodo raíz v puede ser reemplazado por un nuevo árbol con raíz en o' como se muestra en la parte (b), con un costo total de 15.

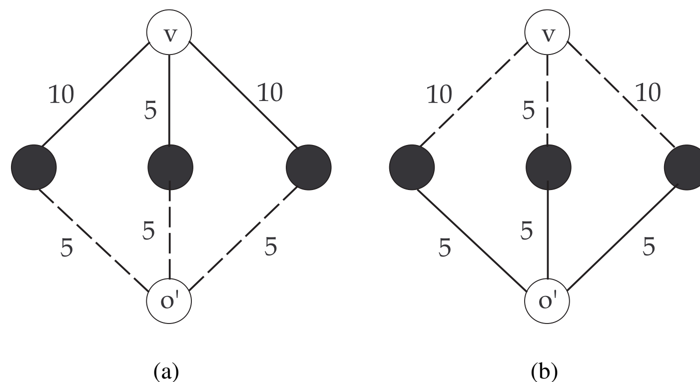


Figura 6.15: Reemplazo de un key-tree por un árbol de menor costo

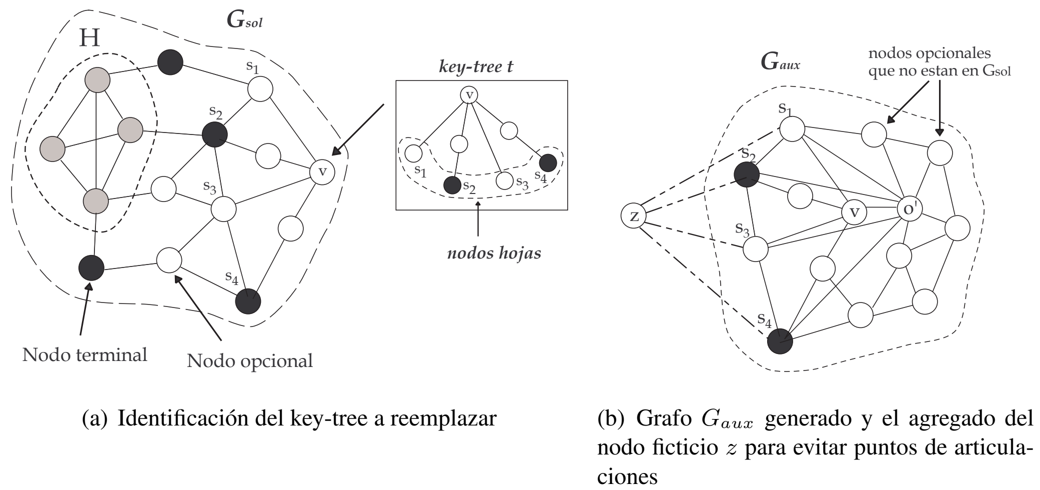
Para lograr una mejor solución, primero se debe de generar el subgrafo $G_{aux} = (G \backslash G_{sol}) \cup \{S_j\}$, siendo G el grafo original. Luego, se obtiene un árbol de menor costo que t_j con raíz el nodo opcional $o' \notin V_{sol}$ y cuyas hojas son los nodos del conjunto S_j. Para obtener este árbol se utiliza el algoritmo llamado $Obtener_Menor_Arbol$ (a continuación se da un descripción detallada de este algoritmo). Por último, se compara si el costo del nuevo árbol t'_j es menor que el costo del key-tree t_j y en caso que sea menor se reemplaza el key-tree t_j por el árbol t'_j.

El algoritmo $Obtener_Menor_Arbol$ (se muestra en el pseudocódigo 6.2.8) es similar al algoritmo de la solución factible inicial del VNS. Para obtener un nuevo árbol sustituto de t de menor costo con raíz un nodo opcional $o' \notin V_{sol}$ y cuyas hojas son los nodos del conjunto S primero, se agrega un nodo ficticio z y aristas de costos 0 desde z hacia todos los nodos $u \in S$ en el subgrafo G_{aux}. Luego, se aplica iterativamente el algoritmo de Bhandari para obtener los k caminos nodos disjuntos, siendo k la cantidad de nodos del conjunto S, con origen cada nodo opcional $o' \notin V_{sol}$ y con destino el nodo ficticio z. El nodo z tiene como objetivo que no se genere puntos de articulación al computar los k caminos nodos disjuntos desde el nodo opcional $o' \notin V_{sol}$ hacia todo los nodos de S. El árbol t' queda generado por los caminos nodos disjuntos obtenidos por el Algoritmo de Bhandari.

En las Figuras 6.16 y 6.17 se puede observar un ejemplo de la vecindad planteada.

- El la Figura 6.16(a) se muestra el grafo solución G_{sol} y el key-tree t_v que va a ser reemplazado por otro árbol de menor costo construido por el algoritmo $Obtener_Menor_Arbol$.
- El la Figura 6.16(b) se observa como queda generado el subgrafo G_{aux} y como se agrega el nodo ficticio z al subgrafo G_{aux}.
- El la Figura 6.17(a) se muestra los caminos nodos disjuntos obtenidos por el Algoritmo de Bhandari con origen el nodo opcional o y destino el nodo ficticio, generando de esta forma el nuevo árbol t'_o.
- El la Figura 6.17(b) se observa el grafo resultado de reemplazar el key-tree t_v por el árbol t'_o

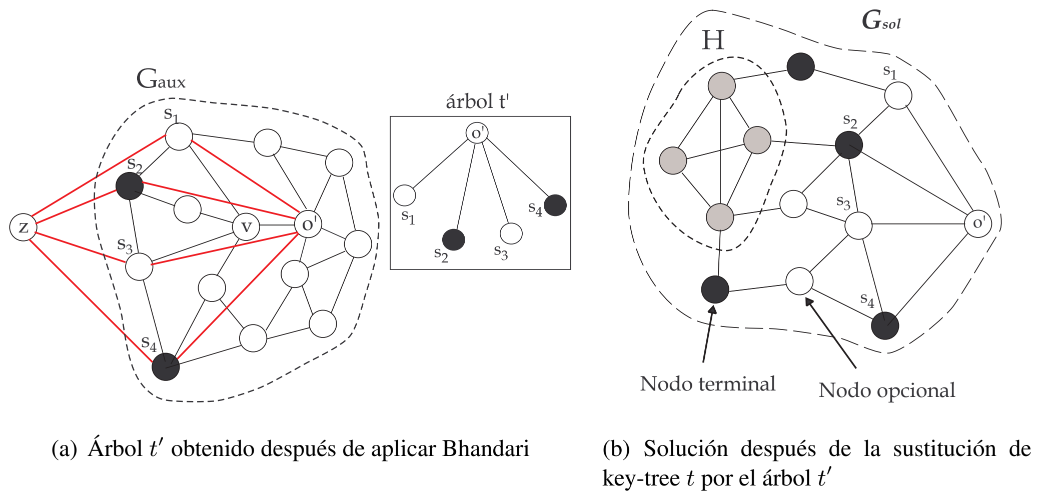
En el Algoritmo 6.2.7 se presenta el pseudocódigo de la vecindad "Optimización por reducción del key-tree con nodo raíz opcional". El mismo toma como entrada el grafo solución actual G_{sol} y devuelve un grafo solución con costo menor o igual al grafo solución actual G_{sol}. En la línea 1 se inicializa G_a con la solución factible actual de entrada G_{sol}. En la línea 2 se obtiene el conjunto de key-trees $KT(G_a) = (t_1...t_n)$ con nodo raíz un key-node. En las líneas 3 a 16 se itera sobre los key-trees pertenecientes a $KT(G_a)$. En la línea 4 se sortea un key-tree t del conjunto $KT(G_a)$. En la línea 5 se obtiene el conjunto de nodos hoja S del key-tree t. En la línea 6 se generar el subgrafo $G_{aux} = (G \backslash G_a) \cup \{S\}$, siendo G el grafo original. En la línea 7 se ejecuta el Algoritmo $Obtener_Menor_Arbol$ para obtener un árbol de costo mínimo con raíz un nodo opcional $o' \notin V_{sol}$ y cuyas hojas son los nodos del conjunto S. Si la búsqueda del árbol t' es exitosa entonces se actualiza el grafo solución G_{sol} por el nuevo grafo $(G_a \backslash \{t\}) \cup \{t'\}$, como se muestra en la línea 9, y luego se recomputa nuevamente el conjunto de key-trees KT, ya que este cambio en el grafo solución puede generar nuevos key-trees. En



(a) Identificación del key-tree a reemplazar

(b) Grafo G_{aux} generado y el agregado del nodo ficticio z para evitar puntos de articulaciones

Figura 6.16: Pasos iniciales para la reducción de key-tree con raíz opcional



(a) Árbol t' obtenido después de aplicar Bhandari

(b) Solución después de la sustitución de key-tree t por el árbol t'

Figura 6.17: Reemplazo de un key-tree con raíz opcional por un árbol de mejor costo

Algoritmo 6.2.7 Optimización por reducción del key-tree con nodo raíz un key-node

Entrada: G_{sol} // Solución Actual
Salida: G_{sol} // Solución con costo menor igual al de G_{sol} actual

1. $G_a = G_{sol}$
2. Sea $KT(G_a) = Descomposicion_KeyTrees_KeyNode(G_a)$ el conjunto de key-trees de G_a con nodo raíz un key-node.
3. **Mientras** $| KT |> 0$ **Hacer**
4. se sortea t perteneciente a KT
5. sea S el conjunto de nodos hojas del key-tree t
6. $G_{aux} = (G \backslash G_a) \cup \{S\}$, siendo G el grafo original.
7. $t' \leftarrow Obtener_Menor_Arbol(G_{aux}, t, S)$
8. **Si** $t' \neq t$ **entonces**
9. $G_{sol} = (G_a \backslash \{t\}) \cup \{t'\}$
10. $G_a = G_{sol}$
11. $KT(G_a) = Descomposicion_KeyTrees_KeyNode(G_a)$
12. **Si no**
13. $G_a = G_{sol}$
14. $KT = KT \backslash \{t\}$
15. **Fin Si**
16. **Fin Mientras**

caso contrario se elimina el key-tree t del conjunto KT. El algoritmo finaliza cuando se analizan todos los key-trees de la solución actual y no es posible mejorar el costo de ninguno de ellos.

En el Algoritmo 6.2.8 toma como entrada el subgrafo auxiliar G_{aux}, el key-tree t a sustituir, el conjunto de nodos hojas S y devuelve el árbol t' de costo mínimo con raíz un nodo opcional $o' \notin V_{sol}$ y cuyas hojas son los nodos del conjunto S. En la línea 1 se inicializa la variable $caminos$ con el vacío, la cual guardará los caminos obtenidos por el Algoritmo de Bhandari. En la línea 2 se inicializa la variable k con la cantidad de nodos del conjunto S y en la línea 3 se inicializa el árbol t' con el key-tree t. En la línea 4 se agrega el nodo ficticio z y aristas de costo 0 desde z hacia todos los nodos del conjunto S en G_{aux}. En la línea 5 se obtiene el conjunto de nodos opcionales O, no pertenecientes al conjunto S, del grafo G_{aux}. En las líneas 6 a 13 se itera en los nodos de O. En la línea 7 se sortea $o' \in O$. En la línea 8 se aplica el Algoritmo de Bhandari para encontrar los k caminos nodos disjuntos con origen el nodo o' y destino el nodo ficticio z. En la línea 9 se compara la sumatoria de los costos de los caminos obtenidos con el costo de t'. En caso que la sumatoria de los costos de los caminos sea menor, el nuevo árbol t' queda generado por los caminos obtenidos como se muestra en la línea 10. En la línea 11 se elimina el nodo o' del conjunto O. El algoritmo finaliza cuando se analizaron todos los nodos opcionales de conjunto O y no es posible encontrar un árbol de menor costo que el actual.

Algoritmo 6.2.8 Obtener Menor Árbol

Entrada: G_{aux}, t, S

Salida: t' // Árbol de menor costo sustituto de t

1. $caminos = \emptyset$
2. $k =\mid S \mid$ // Cantidad de elementos del conjunto S
3. $t' = t$ // Inicializo t' con el key-tree t
4. $G_{aux} \leftarrow Genera_Grafo_Auxiliar(G_{aux}, z, S)$ // Agrego el nodo ficticio z adyacente a todos los nodos de S
5. $O \leftarrow Obtener_Opcionales(G_{aux}, S)$ // Obtiene todos los nodos opcionales de G_{aux} que no están en S
6. **Mientras** $\mid O \mid > 0$ **Hacer**
7. sorteo o' perteneciente a O
8. $caminos \leftarrow Alg_Bhandari(G_{aux}, o', z, k)$
9. **Si** $\sum cost(caminos) < cost(t')$ **entonces**
10. $t' = caminos$ // t' queda generado por los caminos obtenidos
11. **Fin Si**
12. $O = O - \{o\}$
13. **Fin Mientras**

6.2.6. Optimización por Método Exacto

Esta vecindad, tiene como finalidad seleccionar subgrafos Q, de tamaños razonables (menor a 16 nodos), del grafo G_{sol} para aplicar una variante del Método Exacto definida en la Sección 2.3.1.1. Al aplicar este método a cada uno de los subgrafos Q se obtiene como resultado subgrafos Q_{sol} 2-nodo-conexos de costo mínimo que cubren los nodos obligatorios del grafo Q, por tanto, el grafo solución G_{sol} queda determinado reemplazando el subgrafo Q por el nuevo subgrafo Q_{sol} de menor costo. Se entiende como nodos obligatorios a los nodos de Q que no pueden ser eliminados por el Método Exacto al obtener el subgrafo Q_{sol}, ya que si uno de estos nodos no forma parte del subgrafo Q_{sol}, el nuevo grafo solución G_{sol} inducido por Q_{sol} perdería la 2-nodo-conectividad.

Definición 6.2.6 *Dada una instancia del ASTNSNP y una solución factible G_{sol} definimos como vecindad a toda solución factible G'_{sol} que se pueda obtener reemplazando un subgrafo Q, de tamaño razonable, del grafo G_{sol} por un subgrafo Q_{sol} 2-nodo-conexos de costo mínimo que cubre los nodos obligatorios del grafo Q. Para obtener el subgrafo Q_{sol} se aplica al subgrafo Q una variante del Método Exacto definida en la Sección 2.3.1.1.*

La idea de esta vecindad es, dado una solución factible $G_{sol} = (V_{sol}, E_{sol})$ se obtiene el subconjunto O de nodos opcionales pertenecientes a V_{sol}. Se seleccionan estos nodos en particular porque al aplicar el Método Exacto generalmente se eliminan las aristas redundantes incidentes a ellos. A partir de cada uno de estos nodos opcionales, se buscan ciclos simples en el grafo G_{sol} y de estos ciclos simples se generan los subgrafos Q para después aplicar a cada uno de ellos una variante del Método Exacto definida en la Sección 2.3.1.1. Estos ciclos tienen como mínimo N_{min} nodos y como máximo N_{max} nodos, siendo N_{min} y N_{max} la cota inferior

y superior de la cantidad de nodos que puede tener un subgrafo Q.

Para obtener el conjunto de ciclos simples $F = \{F_0, \dots F_m\}$ por cada nodo opcional, se utiliza el Algoritmo $Obtener_Ciclos$ (en la Sección 6.2.6.1 se da una descripción detallada de este algoritmo). Como la cantidad de ciclos simples encontrados para cada nodo opcional puede ser muy grande y aplicar el Método Exacto para todos estos ciclos puede llevar mucho tiempo, entonces, por cada nodo opcional se selecciona sólo un ciclo al azar del total de ciclos hallados. A cada ciclo simple F_i seleccionado al azar se le agrega, si existen, aristas internas al ciclo, es decir, aquellas aristas $(u, v) \in E_{sol}$ tal que los nodos u, v son parte del ciclo F_i pero las aristas (u, v) no pertenece al ciclo F_i. De esta forma los subgrafos $Q_i \subseteq G_{sol}$ quedan determinados por la unión entre el ciclo simple F_i y las aristas internas al ciclo, es decir, $Q_i = F_i \cup \{aristas\, internas\, al\, ciclo\}$.

En el caso que la cantidad de nodos del subgrafo Q_i sea menor a N_{max}, la idea es seguir agregando nodos y aristas que forman parte del grafo G_{sol} pero que no se encuentra en Q_i hasta que la cantidad de nodos sea igual a N_{max}. Al igualar la cantidad de nodos del subgrafo Q_i con N_{max} se logra que Q_i sea el mayor subgrafo que se pueda obtener del grafo G_{sol} y así poder reducir la mayor cantidad de nodos y aristas redundante en el subgrafo Q_i por medio de la variante del Método Exacto que se plantea en la sección 6.2.6.2. Para lograr dicha igualdad de nodos se buscan componentes conexas, las cuales se denotan por Y_i, en $G_{sol} \setminus Q_i$ tal que $\mid Nodos(Y_i) \mid + \mid Nodos(Q_i) \mid \leq N_{max}$. De esta manera, el nuevo grafo Q_i queda determinado por $Q_i \cup Y_i$. Para buscar y agregar las componentes conexas al subgrafo Q_i se utiliza el Algoritmo $Agregar_Componentes_Conexas$ (en la Sección 6.2.6.1 se da una descripción detallada de este algoritmo). Luego que Q_i es determinado se aplica una variante del Método Exacto, definida en la Sección 2.3.1.1, a cada subgrafo Q_i con el fin de resolver el subproblema de encontrar subgrafos solución Q_{sol_i} 2-nodo-conexos de costo mínimo que cubre los nodos obligatorios del grafo Q_i. El grafo solución final queda determinado por el reemplazo del subgrafo Q_i por el nuevo subgrafo Q_{sol_i} en G_{sol}. En el Algoritmo 6.2.9 se presenta el pseudocódigo de esta vecindad.

Observación: Para aplicar el Método Exacto al subgrafo Q_i se considera que los nodos que forman parte del ciclo simple F_i sean nodos obligatorios, salvo los nodos opcionales de grado 2. La razón de denotar a los nodos del ciclo como nodos obligatorios es porque al reemplazar el subgrafo Q_i por el subgrafo Q_{sol_i} en G_{sol} no se pierda la 2-nodo-conectividad de la solución. Los nodos opcionales de grado 2 que forman parte del ciclo simple F_i no se los considera como nodos obligatorios porque estos nodos pueden ser eliminados al obtener el subgrafo Q_{sol_i}, ya que, al reemplazar el subgrafo Q_i por dicho grafo en G_{sol}, la solución obtenida no pierde la 2-nodo-conectividad.

En la Figura 6.18 se puede observar un ejemplo de la vecindad planteada.

- La Figura 6.18(a) muestra el ciclo F_1 hallado con el Algoritmo $Obtener_Ciclos$ para el nodo opcional o. También se señalan las aristas internas al ciclo F_1 que son agregadas.
- En la Figura 6.18(b) se observa el agregado de las componentes conexa al subgrafo Q_1.

En este caso dichas componentes son 2 nodos independientes.

- La Figura 6.18(c), muestra el reemplazo del subgrafo Q_1 por el nuevo subgrafo solución Q_{sol_1} en G_{sol}.
- En la Figura 6.18(d) se observa el grafo final obtenido.

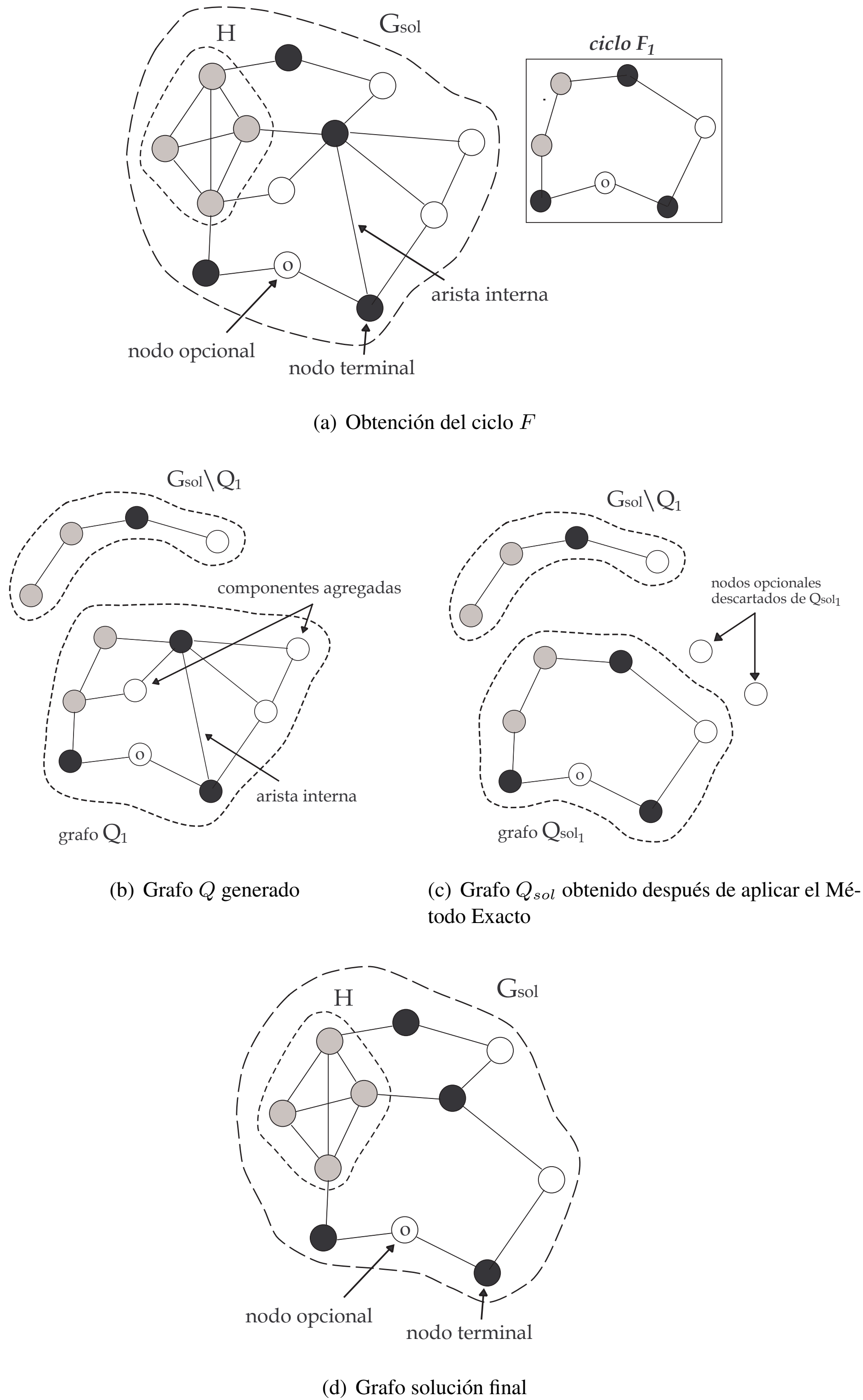
Algoritmo 6.2.9 Optimización por Método Exacto

Entrada: G_{sol} // Solución Actual

Salida: G_{sol} // Solución con costo menor o igual al de G_{sol} actual

1. $G_a = G_{sol}$
2. Sea $O = \{o \in V_{sol}\}$
3. **Mientras** $\mid O \mid > 0$ **Hacer**
4. sorteo o perteneciente a O
5. $F \leftarrow Obtener_Ciclos(G_a, o, N_{min}, N_{max})$ //se buscan todos los ciclos con origen en o en el grafo G_a tal que tiene al menos N_{min} y no mas de N_{max} nodos.
6. se selecciona al azar un ciclo F_i del conjunto $F = \{F_0, \ldots F_m\}$
7. $Q_i = F_i \cup \{aristas\,internas\,al\,ciclo\}$
8. **Si** $|Nodos(Q_i)| < N_{max}$ **entonces**
9. $Q_i \leftarrow Agregar_Componentes_Conexas(G_a, Q_i)$
10. **Fin Si**
11. $Q_{sol_i} \leftarrow Metodo_Exacto(Q_i)$
12. $G_a = (G_a \backslash Q_i) \cup Q_{sol_i}$
13. **Si** costo de G_a < costo de G_{sol} **entonces**
14. $G_{sol} = G_a$
15. $O = O \backslash \{$ nodos opcionales eliminados en $Q_{sol_i}\}$
16. **Si no**
17. $G_a = G_{sol}$
18. $O = O \backslash \{o\}$
19. **Fin Si**
20. **Fin Mientras**

En el Algoritmo 6.2.9 se presenta el pseudocódigo de la vecindad "Optimización por Método Exacto". El mismo toma como entrada el grafo solución actual G_{sol} y devuelve un grafo solución con costo menor o igual al grafo solución actual G_{sol}. En la línea 2 se obtiene el conjunto de nodos opcionales O tal que dichos nodos pertenecen a G_{sol}. En la líneas 3 a 20 se itera sobre los nodos opcionales del conjunto O. En la línea 4 se sortea un nodo opcional o del conjunto O. En la línea 5 se buscan todos los ciclos con origen en o en el grafo G_a tal que tenga al menos N_{min} y no más de N_{max} nodos. Para encontrar esto ciclos se utiliza el Algoritmo $Obtener_Ciclos$, que se describe con detalle en la Sección 6.2.6.1. Con esta función se obtiene el conjunto de ciclos F y en la línea 6 se selecciona un ciclo al azar F_i del conjunto $F = \{F_0, \ldots F_m\}$. En la línea 7 se obtiene el grafo Q_i a partir de la unión del ciclo F_i con las aristas internas al ciclo. Estas aristas internas son aristas $(u, v) \in E_{sol}$ tal que los nodos u, v son parte del ciclo F_i pero las aristas (u, v) no pertenecen al ciclo F_i. En la línea 8 se compara



(a) Obtención del ciclo F

(b) Grafo Q generado

(c) Grafo Q_{sol} obtenido después de aplicar el Método Exacto

(d) Grafo solución final

Figura 6.18: Proceso de optimización por Método Exacto

la cantidad de nodos de Q_i con N_{max}. En caso que sea menor se buscan componentes conexas Y_i en $G_a \setminus Q_i$ tal que $\mid Nodos(Y_i) \mid + \mid Nodos(Q_i) \mid \leq N_{max}$. Para lograr esto, se utiliza el Algoritmo $Agregar_Componentes_Conexas$, que se describe con detalle en la Sección 6.2.6.1. En la línea 11, se aplica el Método Exacto en Q_i (definido en la Sección 6.2.6.2) para poder encontrar el subgrafo solución Q_{sol_i} 2-nodo-conexo de costo mínimo que cubre los nodos obligatorios de Q_i. Luego, en la línea 12, se reemplaza el subgrafo Q_i por el subgrafo Q_{sol_i} en G_{sol}. En las líneas 13 a 19, se actualiza el grafo solución si el nuevo grafo G_a tiene menor costo que el G_{sol} actual. Si se actualiza G_{sol} se elimina del conjunto O los nodos opcionales que fueron eliminados del Q_{sol_i} y que todavía se encuentra en el conjunto O, de lo contrario sólo se elimina el nodo opcional o seleccionado. El algoritmo finaliza cuando se analizan todos los nodos opcionales de la solución actual y no es posible mejorar el costo de dicha solución.

6.2.6.1. Algoritmos Extras

Algoritmo Obtener Ciclos

El Algoritmo $Obtener_Ciclos$ es una función auxiliar del Algoritmo "Optimización por Método Exacto". Dado el nodo opcional o y G_{sol}, el algoritmo obtiene todos los ciclos simples asociado a dicho nodo o en el grafo G_{sol}. Para lograr esto, se realiza un algoritmo muy similar al algoritmo de Depth First Search (DFS por sus siglas en inglés). Su funcionamiento consiste en ir explorando cada arista del grafo de manera que, primero se visitan los nodos adyacentes a los visitados más recientemente, formando así un camino desde el nodo inicial hasta el que es visitado. Cuando en la exploración se visita al nodo origen o y además se cumple que el largo del camino es mayor a N_{min} y menor o igual a N_{max}, entonces dicho camino es considerado como un ciclo válido y se regresa (Backtracking), de modo de repetir el mismo proceso con cada uno de los adyacente al nodo ya procesado. De esta manera se obtiene todos los ciclos asociados al nodo opcional o. En el Algoritmo 6.2.10 se puede apreciar el pseudocódigo de $Obtener_Ciclos$.

El Algoritmo 6.2.10 toma como entrada al grafo solución G_{sol}, el nodo opcional o, la cantidad mínima de nodos N_{min}, la cantidad maxima de nodos N_{max} y devuelve el conjunto de ciclo F asociado al nodo opcional o. En la línea 1 se inicializa el conjunto de ciclos F con el vacío. En la línea 2 se inicializa el camino p con el nodo opcional o, en dicho camino se guardan los nodos que se van explorando en el grafo. En las líneas 3 a 5 se marcan todos los nodos del grafo como "no visitados". En la línea 6 se marca como visitado al nodo opcional o y en la línea 7 se obtienen todos los nodos adyacentes a o. En las líneas 8 a 13 se itera sobre los nodos adyacentes. En la línea 9 se sortea un nodo v del conjunto A y en la línea 10 se construyen los ciclos aplicando el Algoritmo recursivo $Construir_Ciclo$. A medida que se encuentra un ciclo y este cumple con que el largo del ciclo es mayor a N_{min} y menor o igual a N_{max}, dicho ciclo es agregado al conjunto de ciclos F. El pseudocódigo de la función recursiva $Construir_Ciclo$ se muestra en el Algoritmo 6.2.11. En la línea 11 se elimina del conjunto A el nodo v seleccionado.

El Algoritmo 6.2.11 recursivo toma como entrada al grafo solución G_{sol}, el nodo opcional

Algoritmo 6.2.10 Obtener Ciclos

Entrada: G_{sol}, o, N_{min}, N_{max}

Salida: F // Conjunto de todos los ciclos encontrados

1. $F = \emptyset$
2. $p = \{o\}$
3. **Para todo** $v \in V_{sol}$
4. $visitado[v] = false$
5. **Fin Para**
6. $visitado[o] = true$
7. $A \leftarrow Adyacentes(G_{sol}, o)$ // Se obtienen los nodos adyacentes a o
8. **Mientras** $|A| > 0$ **Hacer**
9. sorteo $v \in A$
10. $F \leftarrow Construir_Ciclo(G_{sol}, o, v, N_{min}, N_{max}, visitado, p, F)$
11. $A = A \backslash \{v\}$
12. **Fin Mientras**

Algoritmo 6.2.11 Construir Ciclo

Entrada: G_{sol}, o, v, N_{min}, N_{max}, p, F

Salida: F // Conjunto de los ciclos encontrados

1. $visitado[v] = true$
2. $p = p \cup \{v\}$
3. $A \leftarrow Adyacentes(G_{sol}, v)$ // Se obtienen los nodos adyacentes a v
4. **Mientras** $|A| > 0$ and $|p| \leq N_{max}$**Hacer**
5. sorteo $a \in A$
6. **Si** $\neg visitado[a]$ **entonces**
7. $F \leftarrow Construir_Ciclo(G_{sol}, o, a, N_{min}, N_{max}, visitado, p, F)$
8. **Si no Si** $(a == o)$ and $(|p| > N_{min})$ **entonces**
9. $F = F \cup \{p\}$
10. **Fin Si**
11. $A = A \backslash \{a\}$
12. **Fin Mientras**
13. $visitado[v] = false$
14. $p = p \backslash \{v\}$

o, el nodo v no visitado, la cantidad mínima de nodos N_{min}, la cantidad maxima de nodos N_{max}, el camino p que va explorando el algoritmo en el grafo, el conjunto de ciclos F y devuelve el conjunto de ciclos F con los nuevos ciclos encontrados. En la línea 1 se marca el nodo v como visitado y en la línea 2 se agrega el nodo v al camino p. En la línea 3 se obtienen el conjunto A de nodos adyacentes al nodo v. En las líneas 4 a 13 se itera en los nodos adyacentes a v mientras la cantidad de nodos del caminos sea menor o igual a N_{max}. Si la cantidad de nodos supera la cota superior de nodos para el ciclo no tiene sentido seguir explorando el grafo. En la línea 5 se sortea un nodo del conjunto A. En la línea 6 se compara si el nodo a es visitado. En el caso que el nodo a no sea visitado se invoca de forma recursiva el Algoritmo $Construir_Ciclo$ con el nodo a pasado como parámetro, como se muestra en la línea 7. En el caso contrario, se compara si el nodo a es igual al nodo opcional o y si la cantidad de nodos del camino es mayor a N_{min}. Si se cumple estas 2 condiciones entonces el camino p forma un ciclo y por tanto se agrega dicho ciclo al conjunto F. En la línea 11 se elimina el nodo sorteado a del conjunto A. En la línea 13 se desmarca el nodo v como visitado para que el mismo sea utilizado en otros ciclos y por lo tanto dicho nodo es eliminado del camino p.

Algoritmo Agregar Componentes Conexas

El Algoritmo $Agregar_Componentes_Conexas$ es una función auxiliar del Algoritmo "Optimización por Método Exacto". Dado el grafo $G_{aux} = G_{sol} \setminus Q_i$ se obtiene las componentes conexas Y_i que cumplan con la condición de que $\mid Nodos(Y_i) \mid + \mid Nodos(Q_i) \mid \leq N_{max}$. Para lograr esto, se utiliza el Algoritmo de Breadth First Search (BFS por su sigla en inglés) por cada nodo del grafo G_{aux}. BFS permite explorar las aristas del grafo G_{aux} para "descubrir" todos los nodos alcanzables desde cada nodo del grafo G_{aux}. En el Algoritmo 6.2.12 se puede apreciar el pseudocódigo de $Agregar_Componentes_Conexas$.

Algoritmo 6.2.12 Agregar Componentes Conexas

Entrada: G_{sol}, Q_i
Salida: Q_i // Componentes conexas

1. $G_{aux} = G_{sol} \setminus Q_i$
2. $S = \leftarrow Nodos(G_{aux})$
3. **Mientras** $|S| > 0$ **Hacer**
4. sorteo $v \in S$
5. $Y \leftarrow BFS(G_{aux}, v)$
6. **Si** $\mid Nodos(Y) \mid + \mid Nodos(Q_i) \mid \leq N_{max}$ **entonces**
7. $Q_i = Q_i \cup Y$
8. $S = S \setminus \{nodos(Y)\}$
9. **Si no**
10. $S = S \setminus \{v\}$
11. **Fin Si**
12. **Fin Mientras**

El Algoritmo 6.2.12 toma como entrada al grafo G_{sol}, el subgrafo Q_i y devuelve el grafo generado por la unión del subgrafo Q_i con las componentes conexas halladas. En la línea

1 se genera el grafo G_{aux} para buscar las componentes conexas. En la línea 2 se obtiene el conjunto S de nodos del grafo G_{aux}. En las líneas 3 a 12 se itera en los nodos del conjunto S. En la línea 4 se sortea un nodo $v \in S$ y en la línea 5 se invoca el algoritmo de BFS para encontrar todos los nodos que son alcanzables desde el nodo v. En la línea 6 se compara si $\mid Nodos(Y) \mid + \mid Nodos(Q_i) \mid \leq N_{max}$. Si se cumple con la condición entonces el nuevo subgrafo Q_i queda determinado por $Q_i \cup Y$ y se eliminan del conjunto S los nodos de la componente conexa Y. De lo contrario sólo se elimina el nodo sorteado v del conjunto S.

6.2.6.2. Formulación del Método Exacto como Búsqueda Local

Para modelar el subproblema de encontrar el subgrafo de cubrimiento de costo mínimo 2 nodo conexo, se transforma el grafo $Q = (V_Q, E_Q)$ en un grafo dirigido $Q' = (V'_Q, E'_Q)$, donde para cada arista $(i, j) \in E_Q$ existe una arista dirigida $i \to j \in E'$ y otra arista dirigida $j \to i \in E'$. También se tiene la matriz de costos $C'_Q = \{c_{ij}\}_{(i,j) \in E'}$. Se utiliza la notación de arista (i, j) para especificar la dirección de i a j.

Se debe tener en cuenta, que los nodos $x \in X$ que pertenecen a la red existente en el problema original, en el grafo $Q' = (V'_Q, E'_Q)$ se los considera en forma indistinta como nodos terminales. Esto se define así, ya que a diferencia del problema original, donde tienen que haber dos caminos nodos disjuntos entre los nodos terminales y los nodos de la red fija, en esta vecindad, el modelo impone la existencia de dos caminos nodo disjuntos entre pares de nodos de cualquiera de estos tipos, debido a que no se puede asegurar la 2-nodo-conectividad entre los nodos de X que se encuentran en el subgrafo Q'.

Cabe destacar que en la definición de esta vecindad se habla de nodos obligatorios para aquellos nodos que no pueden ser eliminados al obtener el subgrafo Q'_{sol}, por lo tanto, en la formulación del Método Exacto los nodos obligatorios también son considerados como nodos terminales.

Al igual que en la formulación del método exacto realizada en la sección 2.3, para que el grafo solución sea 2-nodo-conexo, se utilizó la técnica de separación de vértices (splitting vertex) en $Q' = (V'_Q, E'_Q)$ obteniéndose el grafo $Q'' = (V''_Q, E''_Q)$. En este modelo, el conjunto de nodos terminales esta representado por $T''_Q \subset V''_Q$.

Se considera el siguiente problema de Programación Lineal Entera:

$$y^{uv}_{(i,j)} = \begin{cases} 1 & \text{si la arista } (i,j) \in E''_Q \text{ esta en el camino de } u \in T''_Q \text{ y } v \in T''_Q,\ u \neq v \\ 0 & \text{sino} \end{cases} \tag{6.1}$$

$$x_{(i,j)} = \begin{cases} 1 & \text{si el arco } (i,j) \in E''_Q \text{ es utilizado} \\ 0 & \text{sino} \end{cases} \tag{6.2}$$

$$\text{mín} \sum_{(i,j)\in E''_Q} c_{ij} * x_{ij} \tag{6.3}$$

s.a.:

$$\sum_{(u,j)\in E''_Q} y^{uv}_{(u,j)} \geq 2 \; \forall u,v \in T'',\, u \neq v. \tag{6.4}$$

$$\sum_{(i,v)\in E''_Q} y^{uv}_{(i,v)} \geq 2 \; \forall u,v \in T''_Q,\, u \neq v. \tag{6.5}$$

$$\sum_{(i,p)\in E''_Q} y^{uv}_{(i,p)} - \sum_{(p,i)\in E''_Q} y^{uv}_{(p,i)} \geq 0 \; \forall u,v \in T''_Q, u \neq v, \forall p \in T''_Q \backslash \{u,v\}. \tag{6.6}$$

$$y^{uv}_{(i,j)} + y^{uv}_{(j,i)} \leq x_{(i,j)} \; \forall u,v \in T''_Q, u \neq v, \forall (i,j) \in E''_Q. \tag{6.7}$$

$$x_{(i,j)} \in \{0,1\} \; \forall (i,j) \in E''_Q. \tag{6.8}$$

$$y^{uv}_{(i,j)} \geq 0 \; \forall u,v \in T''_Q, u \neq v, \forall (i,j) \in E''_Q. \tag{6.9}$$

La restricción 6.4, expresa que para todo nodo de T''_Q salen al menos dos caminos hacia cualquier otro nodo de T''_Q. La restricción 6.5, garantiza que para todo nodo de T''_Q le llegan al menos dos caminos de cualquier otro nodo de T''_Q. La restricción 6.6, fuerza el equilibrio entre las aristas de salida y entrada en los nodos intermedios de los caminos. La restricción 6.7, impone que los caminos sean de aristas disjuntas y también que, si existe una arista dirigida entonces existe un arco no dirigido en el grafo solución.

6.3. VNS: Variable Neighborhood Search

En este trabajo se utiliza el esquema tradicional de VNS, en el cual se realizan cambios de vecindad de forma determinística cada vez que se llega a un mínimo local en la vecindad actual, es decir, se utiliza un VND para poder lograr mejoras en los óptimos locales. En el algoritmo 6.3.1 se puede observar el VNS y en el algoritmo 6.3.2 se puede ver el VND.

En la línea 1 del algoritmo 6.3.1 se realiza la construcción de una solución factible inicial G_{sol} mediante el algoritmo definido en la sección 6.1.2. En las líneas 2 a 17 se itera hasta que se cumpla un cierto criterio de corte, que en nuestro caso es la cantidad máxima de iteraciones que se realiza el VNS. En las líneas 4 a 16 se itera tantas veces como lo indica el parámetro K_{max}, siendo este la cantidad maxima que se aplica VND como búsqueda locas sin lograr mejoras en la solución actual. En caso que se logre mejoras en la solución actual se reinicia dicho iterador. La línea 7 se realiza después que se haya aplicado por primera vez VND sobre la solución factible inicial. Para mejorar la diversidad en las búsquedas locales, en las diferentes estructuras de vecindad, se agrega una función $variarSolInicial$ que realiza cambios en la solución obtenida hasta ese momento sin perder la factibilidad. La idea de la función $variarSolInicial$ es

Algoritmo 6.3.1 VNS

Entrada: G //Grafo original
Salida: G_{sol}
1. obtener solución inicial factible G_{sol}
2. **Mientras** algún criterio de corte conocido
3. $k = 0$
4. **Mientras** $k < K_{max}$
5. $G_a = G_{sol}$
6. **Si** $k > 0$ **entonces**
7. $variarSolInicial(G_a, k)$;
8. **Fin Si**
9. $VND(G_a)$
10. **Si** costo de G_a < costo de G_{sol} **entonces**
11. $G_{sol} = G_a$
12. $k = 1$
13. **Si no**
14. $k = k + 1$
15. **Fin Si**
16. **Fin Mientras**
17. **Fin Mientras**

agregar aristas, de forma aleatoria, las cuales no se encuentran en la solución. A este algoritmo se lo conoce comúnmente con el nombre de "Agitación" del grafo solución obtenido hasta ese momento. De esta manera, se logra un óptimo local cercano al óptimo global que esta menos ligado al grafo inicial del cual se parte. La línea 9 invoca el algoritmo de VND como búsqueda local y en la línea 10 se verifica si se ha logrado una mejora en el mejor costo hallado hasta ese momento; de ser así, en la línea 11 se registra la nueva solución en G_{sol} y se reinicia a 1 el iterador k. En caso que no se mejore el costo hallado hasta ese momento se incrementa en 1 el iterador k. Finalmente, cuando se alcanza la cantidad maxima de iteraciones de VNS, se devuelve la mejor solución encontrada.

El algoritmo de VND toma como entrada la solución encontrada hasta ese momento y devuelva la mejor solución obtenida después de aplicar las diferentes vecindades definidas en la sección 6.2. En la línea 3 se sortea al azar cual de las vecindades definidas es la que primero se ejecuta en la secuencia. En las líneas 4 a 13 se itera tantas veces como cantidad de vecindades exista, en nuestro caso es 6 veces ya que se definieron 6 vecindades diferentes. En la línea 7 se aplica una vecindad diferente por cada valor que toma i determinado en la línea anterior. Luego, en la línea 8 se compara si la solución obtenida por la vecindad que se aplico en la línea anterior mejora el mejor costo hallado hasta ese momento; de ser así, en la línea 8 se registra la nueva solución en G_{sol} y se reinicia a 1 el iterador l. En caso que no se mejore el costo hallado hasta ese momento se incrementa en 1 el iterador l. El algoritmo finaliza cuando todas las vecindades definidas son ejecutadas.

Algoritmo 6.3.2 VND

Entrada: G_{sol} //Solución obtenida hasta ese momento

Salida: G_{sol}

1. $l = 1$
2. $G_a = G_{sol}$
3. $a = random(cantVecindades)$ //Se selecciona al azar la primera vecindad a ejecutar
4. **Mientras** $l =< cantVecindades$
5. $i = (l + a)\ mod\ cantVecindades$
6. Por cada valor que toma i se aplica una vecindad diferente
7. **Si** costo de G_a < costo de G_{sol} **entonces**
8. $G_{sol} = G_a$
9. $l = 1$
10. **Si no**
11. $l = l + 1$
12. **Fin Si**
13. **Fin Mientras**

6.3.1. Implementación de VNS

Los algoritmos planteados para la resolución mediante VNS, fueron implementados en C++ utilizando las biblioteca de Boost version 1.4.2 [12], en particular se utilizó la biblioteca de grafos Boost Graph Library [101] [65] (denotado por BGL). En el anexo A.1 se puede obtener más detalles de esta biblioteca. Se resolvió la utilización de BGL, ya que proporciona las estructuras de datos necesarias para la implementación de grafos así como algunos métodos y algoritmos utilizados en la resolución del VNS.

Al igual que para la resolución del método exacto, la implementación de MIP utilizada en la vecindad "Optimización por Método Exacto", se realizó mediante la herramienta CPLEX version 10.0.1 [60].

A continuación se explican los diferentes parámetros definidos y utilizados en el algoritmo de VNS.

6.3.1.1. Parametrización de VNS

Se decidió que el algoritmo VNS sea configurable con el propósito de lograr diferentes resultados y así poder analizar el comportamiento del mismo de manera sencilla, sin tener que modificar el programa realizado. Para lograr este objetivo se define un conjunto de parámetros, leídos desde un archivo de configuración en el programa principal. Estos parámetros representan los diferentes valores de los porcentajes y constantes que son utilizadas en el algoritmo.

Se definieron los siguientes parámetros:

- Porcentaje de agregación: Este porcentaje indica la cantidad de aristas que se agregan a la solución actual en la función $variarSolInicial$. Con este parámetro se puede controlar la cantidad de aristas en que varia el grafo solución actual para después aplicar el algoritmo de VND.
- criterio de corte P_{max}: Es la cantidad máxima de iteraciones que se realiza el VNS.
- criterio de corte K_{max}: Esta contante indica la cantidad máxima de iteraciones, que se aplica VND, sin lograr mejoras en la solución actual.
- Cota inferior en el MIP: Es utilizada en la generación del ciclo F_i en la vecindad que realiza MIP como cota inferior de la cantidad de nodos (N_{min}) de dicho ciclo.
- Cota superior en el MIP: Es utilizada en la generación del ciclo F_i en la vecindad que realiza MIP como cota superior de la cantidad de nodos (N_{max}) de dicho ciclo. Es de destacar, que la cantidad de nodos en el ciclo F_i está dada por la elección de un número randómico obtenido entre los valores de N_{min} y N_{max}.
- Se define una constante binaria para cada vecindad que indica si la vecindad es utilizada o no en el algoritmo de VND. Si la constante vale 1 se aplica la vecindad y en caso contrario vale 0. Estas constantes son utilizadas en los casos de prueba donde se pretende comparar la eficacia de cada una de las vecindades definidas.

Capítulo 7

Resultados Obtenidos Mediante el Uso de VNS

En este Capítulo se presentan los resultados obtenidos con el algoritmo VNS. En la Sección 7.1, se definen los casos de prueba utilizados, así como la cantidad de nodos y el porcentaje de cada tipo de nodo utilizado en los distintos casos de prueba. En la Sección 7.2 se muestran los resultados obtenidos con los casos de prueba seleccionados.

7.1. Definición de casos de Pruebas

Al igual que para el método exacto, los grafos seleccionados para los casos de prueba del algoritmo VNS fueron obtenidos de la librería TSPlib [57]. Para realizar las pruebas se identificaron dos grupos de grafos. El primer grupo está compuesto por los grafos utilizados en la resolución del método exacto, los cuales se muestran en la tabla 4.1. Para este grupo de grafos, se realizaron pruebas con VNS con las mismas configuraciones que fueron hechas en la definición de los casos de prueba del Método Exacto (ver sección 4.1). La ejecución de VNS con este grupo de grafos, tiene como objetivo comparar los resultados que se obtienen aplicando este algoritmo con los resultados que se obtienen mediante la resolución exacta del problema. El segundo grupo de grafos, presentado en la tabla 7.1, está compuesto por grafos con mayor cantidad de nodos y aristas, a los cuales no es posible aplicar el Método Exacto. En este grupo, se decidió aplicar sólo las configuraciones 1, 2 y 3 para realizar los casos de prueba en virtud de los excesivos tiempos insumidos por VNS en la ejecución de los mismos.

Con el objetivo de facilitar la lectura de las tablas de resultados, se numeraron cada una de las vecindades de la siguiente manera:

1. Agregar Aristas\ Reducir Redundantes.

2. Intercambio de 2 Aristas entre 4 Nodos.

3. Optimización por Método Exacto.

4. Optimización por reducción de key-tree.

5. Optimización por reducción del key-tree con nodo raíz opcional.
6. Optimización por reemplazo de key-path.

Instancia	Cantidad de nodos	Cantidad de Aristas
Swiss42	42	861
Gr48	48	1128
Eil51	51	1275
Berlin52	52	1326
St70	70	2415
Eil76	76	2850
Pr76	76	2850
KroA100	100	4950
KroB100	100	4950
Rd100	100	4950
KroB100	100	4950
Pr124	124	7626

Tabla 7.1: Instancias de TSPLib utilizadas para pruebas de VNS

7.2. Presentación de los Resultados

En esta Sección se presentan los resultados obtenidos para el algoritmo de VNS. Las pruebas fueron realizadas en un Intel Core i5 de 2.67 GHz, de 8 Gbytes de RAM y sobre Windows 7 de 64 bits. Con el fin de calcular los valores promedios de los costos obtenidos por VNS como resultados del problema, se realizaron diez ejecuciones independientes de dicho algoritmo para los grafos grandes y diez ejecuciones independientes para los grafos chicos, estos últimos utilizados para la comparación con el Método Exacto.

Los resultados obtenidos con los casos de prueba pueden analizarse desde tres perspectivas diferentes. En primer lugar, se comparan los resultados de VNS con los resultados obtenidos por el Método Exacto. En segundo lugar, se obtienen los resultados del VNS para instancias de grafos grandes, es decir, con una cantidad de nodos mayor a 40. Por último, se presentan los resultados de VNS variando la cantidad de vecindades, aplicadas en el VND, en cada ejecución.

7.2.1. Resultados de VNS comparados con Método Exacto

Para realizar la comparación entre los resultados obtenidos aplicando VNS y los obtenidos aplicando el Método Exacto, se tomó el conjunto de grafos que se utilizó para probar este último. En las tablas 7.2 a 7.10, se muestran los resultados obtenidos. En estas tablas, se registraron los costos de la solución óptima $C(Gsol^*)$ obtenidos por el Método Exacto y su

respectivo tiempo. En el caso de VNS se registra el promedio del costo $C(Gsol)$ y el promedio del tiempo que demora en obtener la solución óptima, así como también el mejor costo obtenido $C(Gsol_{best})$.

Para facilitar la visualización de las tablas a continuación se definen algunos términos utilizados:

- N_X: Cantidad de nodos del conjunto X (nodos de la red existente).
- N_T: Cantidad de nodos del conjunto T (nuevos nodos obligatorios: terminales).
- N_{Op}: Cantidad de nodos del conjunto Op (nuevos nodos opcionales: nodos de Steiner).
- T_{MIP}: Tiempo en segundos insumido por el solver de CPLEX para resolver el problema.
- T_{VNS}: Tiempo en segundos insumido por VNS para resolver el problema.
- $C(Gsol^*)$: Costo del grafo solución obtenido mediante MIP.
- $\bar{C}(Gsol)$: Costo promedio del grafo solución obtenido mediante VNS.
- $C(Gsol_{best})$: Mejor costo del grafo solución obtenido mediante VNS.
- Gap: Porcentaje de sobrecosto (conocido como "gap" en inglés) de la solución obtenida por medio de VNS respecto a la dada por el Método Exacto.

Instancia	N_X	N_T	N_{Op}	$C(Gsol^*)$	$T_{MIP}(s)$	$\bar{C}(Gsol)$	$T_{VNS}(s)$	$C(Gsol_{best})$	Gap
burma14	2	8	4	22,41	0,41	24,80	0,80	22,41	**0,0 %**
ulysses16	3	9	4	56,21	2,18	61,32	2,00	57,49	**2,2 %**
gr17	3	10	4	1.515,00	3,60	1.773,00	2,80	1.515,00	**0,0 %**
gr21	4	12	5	2.477,00	59,25	2.858,00	6,80	2.477,00	**0,0 %**
ulysses22	4	13	5	58,92	25,54	63,89	6,60	58,92	**0,0 %**
gr24	4	14	6	1.038,00	355,73	1.134,00	7,00	1.038,00	**0,0 %**
fri26	5	15	6	724,00	62,34	726,20	12,00	724,00	**0,0 %**
bays29	5	17	7	1.371,00	103,11	1.878,00	14,50	1.371,00	**0,0 %**

Tabla 7.2: Resultados MIP vs VNS con la Configuración 1

Instancia	N_X	N_T	N_{Op}	$C(Gsol^*)$	$T_{MIP}(s)$	$\bar{C}(Gsol)$	$T_{VNS}(s)$	$C(Gsol_{best})$	Gap
burma14	2	4	7	18,14	0,27	23,41	0,10	18,14	**0,0 %**
ulysses16	3	6	7	23,77	1,12	53,06	0,60	23,77	**0,0 %**
gr17	3	6	8	921,00	2,18	1.247,50	1,50	984,50	**6,4 %**
gr21	4	8	9	2.029,00	7,69	2.228,20	31,60	2.029,00	**0,0 %**
ulysses22	4	8	10	53,59	7,39	62,45	2,20	53,59	**0,0 %**
gr24	4	9	11	744,00	11,40	1.124,00	3,30	744,00	**0,0 %**
fri26	5	10	11	516,00	30,03	627,00	4,30	516,00	**0,0 %**
bays29	5	11	13	1.179,00	45,74	1.918,00	6,60	1.179,00	**0,0 %**

Tabla 7.3: Resultados MIP vs VNS con la Configuración 2

Instancia	N_X	N_T	N_{Op}	$C(Gsol^*)$	$T_{MIP}(s)$	$\bar{C}(Gsol)$	$T_{VNS}(s)$	$C(Gsol_{best})$	Gap
burma14	2	2	10	13,70	0,09	21,55	0,01	13,70	**0,0 %**
ulysses16	3	3	10	18,36	0,45	40,70	0,06	18,36	**0,0 %**
gr17	3	3	11	858,00	0,42	1.037,30	2,70	858,00	**0,0 %**
gr21	4	4	13	1.655,00	2,00	1.759,10	8,45	1.655,00	**0,0 %**
ulysses22	4	4	14	47,70	1,73	53,97	1,02	52,68	**9,5 %**
gr24	4	4	16	488,00	2,54	595,00	3,73	488,00	**0,0 %**
fri26	5	5	16	345,00	9,00	453,00	1,10	345,00	**0,0 %**
bays29	5	5	19	825,00	11,11	1.384,00	1,30	825,00	**0,0 %**

Tabla 7.4: Resultados MIP vs VNS con la Configuración 3

Instancia	N_X	N_T	N_{Op}	$C(Gsol^*)$	$T_{MIP}(s)$	$\bar{C}(Gsol)$	$T_{VNS}(s)$	$C(Gsol_{best})$	Gap
burma14	7	3	4	10,96	0,80	11,21	0,60	10,96	**0,0 %**
ulysses16	8	4	4	42,60	2,78	43,86	2,20	42,60	**0,0 %**
gr17	8	4	5	452,00	3,01	527,00	1,40	452,00	**0,0 %**
gr21	10	5	6	1.425,00	14,45	1.510,00	4,70	1.425,00	**0,0 %**
ulysses22	11	5	6	48,52	18,19	50,67	9,45	48,52	**0,0 %**
gr24	12	6	6	505,00	48,70	550,00	12,50	505,00	**0,0 %**
fri26	13	6	7	345,00	111,51	407,80	16,95	362,00	**4,7 %**
bays29	14	7	8	850,00	116,56	991,70	26,70	850,00	**0,0 %**

Tabla 7.5: Resultados MIP vs VNS con la Configuración 4

Instancia	N_X	N_T	N_{Op}	$C(Gsol^{*})$	$T_{MIP}(s)$	$\bar{C}(Gsol)$	$T_{VNS}(s)$	$C(Gsol_{best})$	Gap
burma14	7	5	2	11,58	1,59	15,09	2.35	11,58	**0,0 %**
ulysses16	8	6	2	46,32	4,77	49,20	4,60	46,32	**0,0 %**
gr17	8	6	3	1.108,00	6,29	1.223,60	3,50	1.108,00	**0,0 %**
gr21	10	8	3	2.003,00	34,63	2.117,00	11,60	2.003,00	**0,0 %**
ulysses22	11	8	3	19,34	58,08	24,04	23,50	19,51	**0,0 %**
gr24	12	9	3	685,00	92,23	756,00	30,60	750,00	**0,9 %**
fri26	13	10	3	406,00	209,26	440,00	47,50	406,00	**0,0 %**
bays29	14	11	4	-	Out Memory	1.215,80	74,50	1.072,00	**0,0 %**

Tabla 7.6: Resultados MIP vs VNS con la Configuración 5

Instancia	N_X	N_T	N_{Op}	$C(Gsol^{*})$	$T_{MIP}(s)$	$\bar{C}(Gsol)$	$T_{VNS}(s)$	$C(Gsol_{best})$	Gap
burma14	7	1	6	4,78	0,14	4,78	0,25	4,78	**0,0 %**
ulysses16	8	1	7	4,76	0,22	4,76	0,23	4,76	**0,0 %**
gr17	8	1	8	161,00	0,27	161,00	0,50	161,00	**0,0 %**
gr21	10	2	9	1.010,00	2,26	1.010,00	1,00	1.010,00	**0,0 %**
ulysses22	11	2	9	8,12	2,76	8,36	2,20	8,12	**0,0 %**
gr24	12	2	10	269,00	5,26	269,00	2,20	269,00	**0,0 %**
fri26	13	2	11	207,00	5,94	228,00	6,95	207,00	**0,0 %**
bays29	14	2	13	454,00	9,78	458,00	4,10	454,00	**0,0 %**

Tabla 7.7: Resultados MIP vs VNS con la Configuración 6

Instancia	N_X	N_T	N_{Op}	$C(Gsol^{*})$	$T_{MIP}(s)$	$\bar{C}(Gsol)$	$T_{VNS}(s)$	$C(Gsol_{best})$	Gap
burma14	9	0	5	No Aplica	-	No Aplica	-	-	**0,0 %**
ulysses16	11	0	5	No Aplica	-	No Aplica	-	-	**0,0 %**
gr17	11	0	6	No Aplica	-	No Aplica	-	-	**0,0 %**
gr21	14	1	6	159,00	1,01	159,00	1,40	159,00	**0,0 %**
ulysses22	15	1	6	3,65	1,33	3,65	1,90	3,65	**0,0 %**
gr24	16	1	7	124,00	1,76	124,00	2,40	124,00	**0,0 %**
fri26	18	1	7	164,00	2,59	164,00	3,70	164,00	**0,0 %**
bays29	20	1	8	110,00	4,49	110,00	5,10	110,00	**0,0 %**

Tabla 7.8: Resultados MIP vs VNS con la Configuración 7

Instancia	N_X	N_T	N_{Op}	$C(Gsol^{*})$	$T_{MIP}(s)$	$\bar{C}(Gsol)$	$T_{VNS}(s)$	$C(Gsol_{best})$	Gap
burma14	9	2	3	8,32	0,51	8,32	0,60	8,32	**0,0 %**
ulysses16	11	2	3	8,12	0,94	10,89	1,80	8,12	**0,0 %**
gr17	11	2	4	277,00	1,09	277,00	1,40	277,00	**0,0 %**
gr21	14	3	4	1.067,00	7,16	1.113,00	5,90	1.067,00	**0,0 %**
ulysses22	15	3	4	41,02	9,66	41,02	9,60	41,02	**0,0 %**
gr24	16	3	5	326,00	10,87	374,90	12,30	326,00	**0,0 %**
fri26	18	3	5	243,00	18,30	262,20	17,40	243,00	**0,0 %**
bays29	20	4	5	523,00	65,83	552,20	37,90	523,00	**0,0 %**

Tabla 7.9: Resultados MIP vs VNS con la Configuración 8

Instancia	N_X	N_T	N_{Op}	$C(Gsol^{*})$	$T_{MIP}(s)$	$\bar{C}(Gsol)$	$T_{VNS}(s)$	$C(Gsol_{best})$	Gap
burma14	9	3	2	8,56	0,97	8,80	1,6	8,56	**0,0 %**
ulysses16	11	4	1	15,33	2,71	20,54	5,60	15,33	**0,0 %**
gr17	11	4	2	648,00	3,42	708,00	4,20	648,00	**0,0 %**
gr21	14	5	2	1.138,00	21,60	1.216,10	13,00	1.138,00	**0,0 %**
ulysses22	15	5	2	44,07	25,33	49,80	19,40	44,07	**0,0 %**
gr24	16	6	2	560,00	57,21	684,20	41,00	560,00	**0,0 %**
fri26	18	6	2	305,00	93,63	318,00	56,20	305,00	**0,0 %**
bays29	20	7	2	-	Out Memory	957,40	109,40	862,00	**0,0 %**

Tabla 7.10: Resultados MIP vs VNS con la Configuración 9

En los resultados obtenidos por VNS para todas las configuraciones, se puede observar que el mejor resultado ($C(Gsol_{best})$) iguala, en la gran mayoría de los casos, al resultado obtenido por el Método Exacto, es decir, al valor óptimo obtenido por el Método Exacto. Además, todas las tablas muestran el porcentaje de sobrecosto (conocido como “gap” en inglés) de la solución obtenida con el VNS respecto a la dada por el Método Exacto. Como se puede apreciar para la configuraciones 6, 7, 8 y 9 el gaps es del $0\,\%$ en todos los casos de pruebas realizados y en la configuración 5, sólo para el caso de prueba “Gr24”, se obtuvo un muy pequeño gaps (sólo $0{,}09\,\%$) con respecto al valor óptimo. En las configuraciones 1, 2, 3, 4, sólo 4 casos de prueba no alcanzaron el óptimo, en promedio tuvieron un gaps del $5,7\,\%$ con respecto al valor óptimo. Otro dato importante de destacar, es que los tiempos insumidos por VNS son relativamente menor que los tiempos insumidos por el Método Exacto.

En las Figuras 7.1 a 7.8 se pueden observar gráficas con la comparación de los valores obtenidos con el Método Exacto y los valores promedios ($\bar{C}(Gsol)$) obtenidos con el VNS. Como se pueden apreciar, no todas las ejecuciones independientes realizadas del Algoritmo VNS, de las 10 que se hicieron, obtuvieron valores iguales a los obtenidos por el Método Exacto.

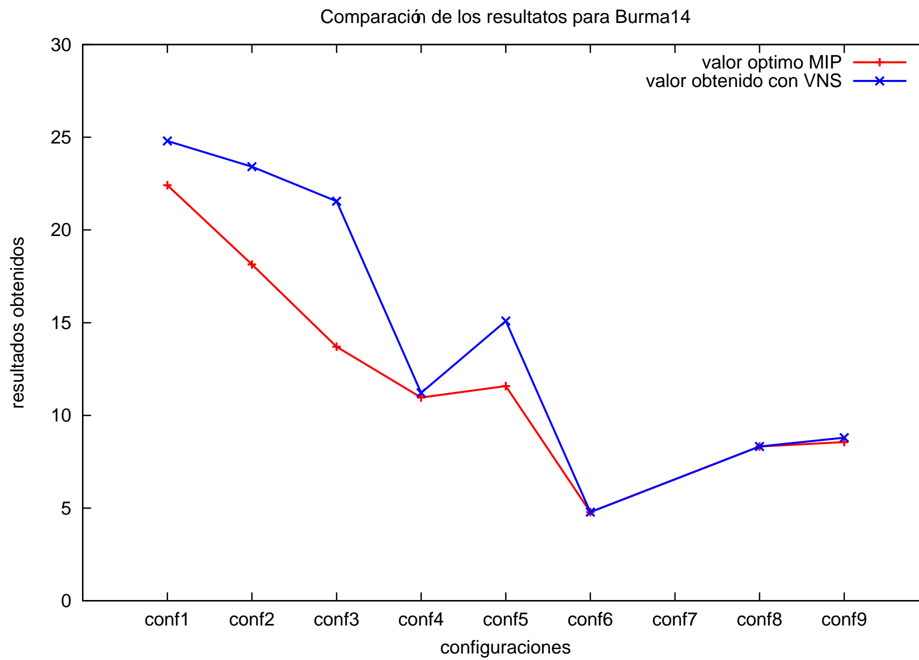


Figura 7.1: Comparativa de los resultados del Método Exacto con los de VNS para Bruma14

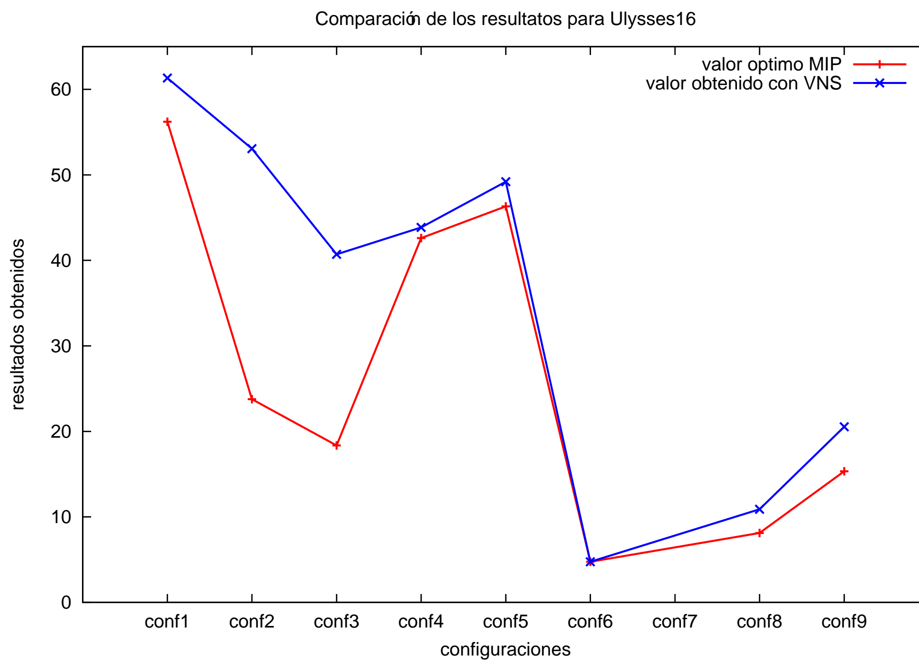


Figura 7.2: Comparativa de los resultados del Método Exacto con los de VNS para Ulysses16

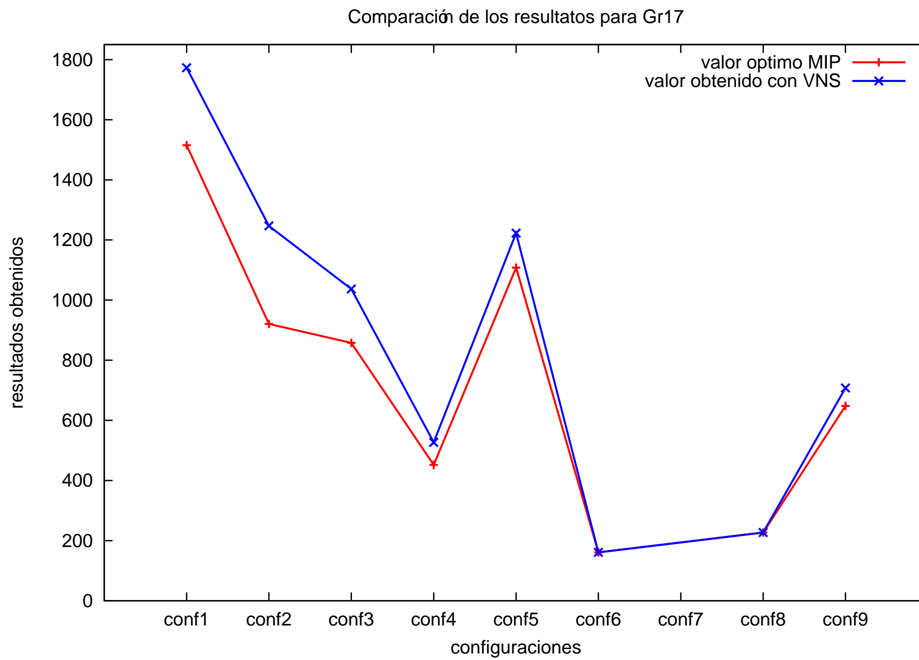


Figura 7.3: Comparativa de los resultados del Método Exacto con los de VNS para Gr17

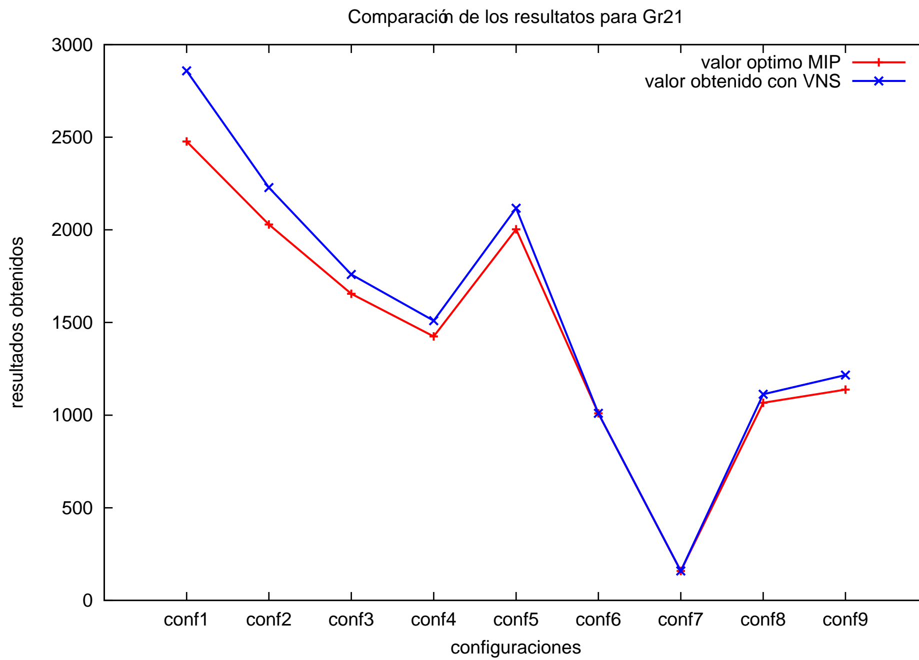


Figura 7.4: Comparativa de los resultados del Método Exacto con los de VNS para Gr21

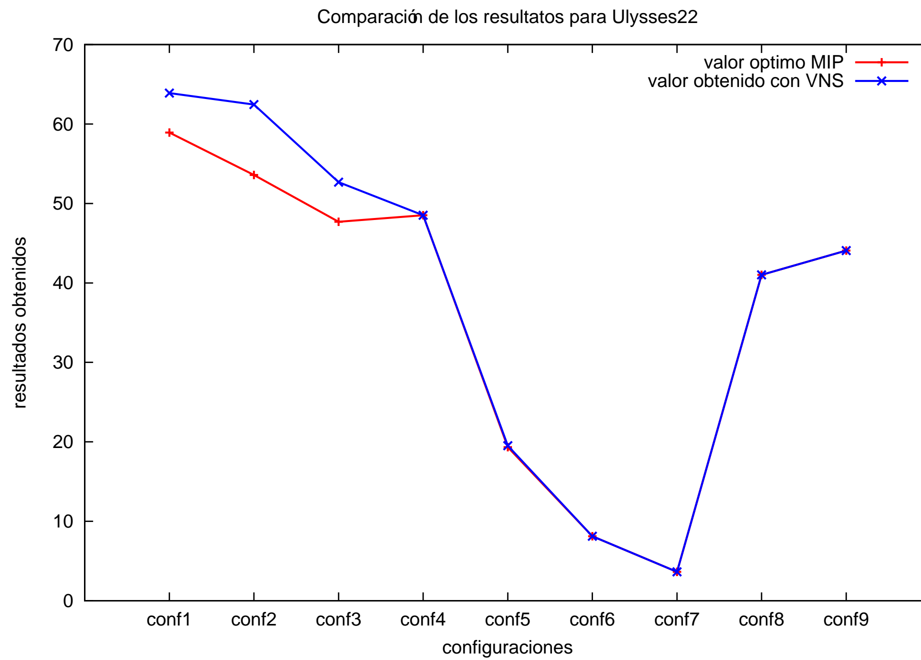


Figura 7.5: Comparativa de los resultados del Método Exacto con los de VNS para Ulysses22

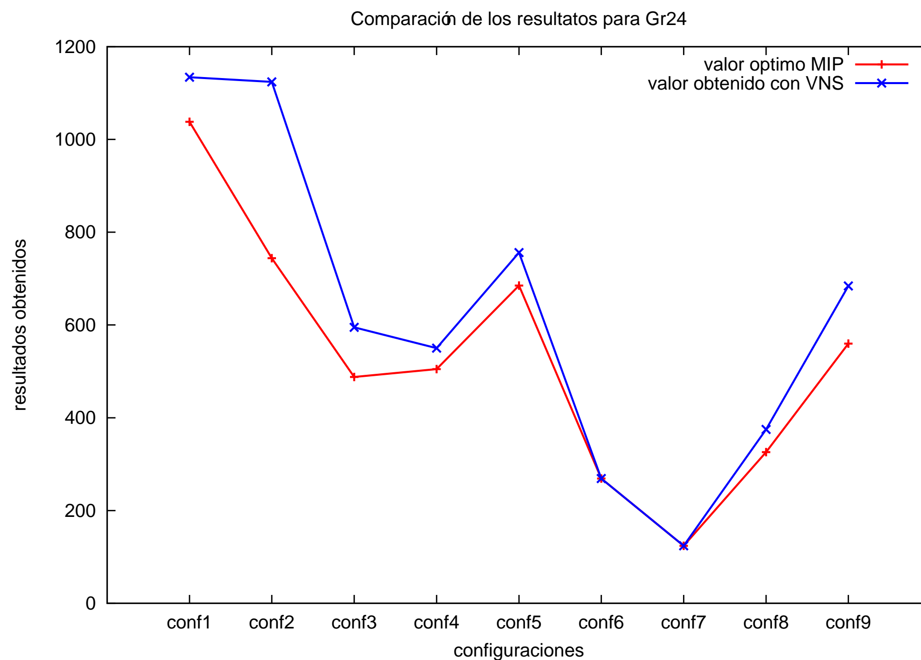


Figura 7.6: Comparativa de los resultados del Método Exacto con los de VNS para Gr24

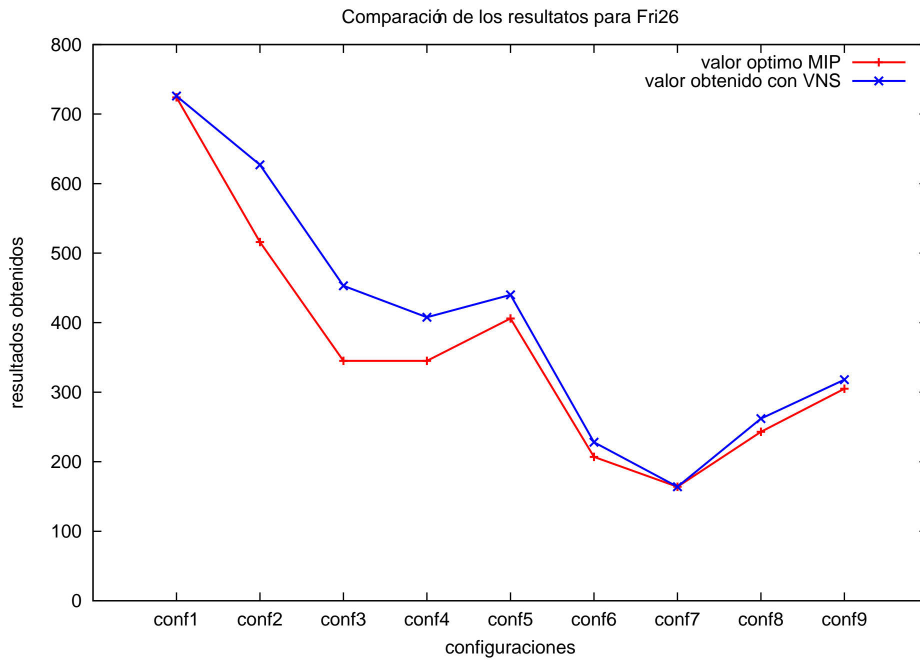


Figura 7.7: Comparativa de los resultados del Método Exacto con los de VNS para Fri26

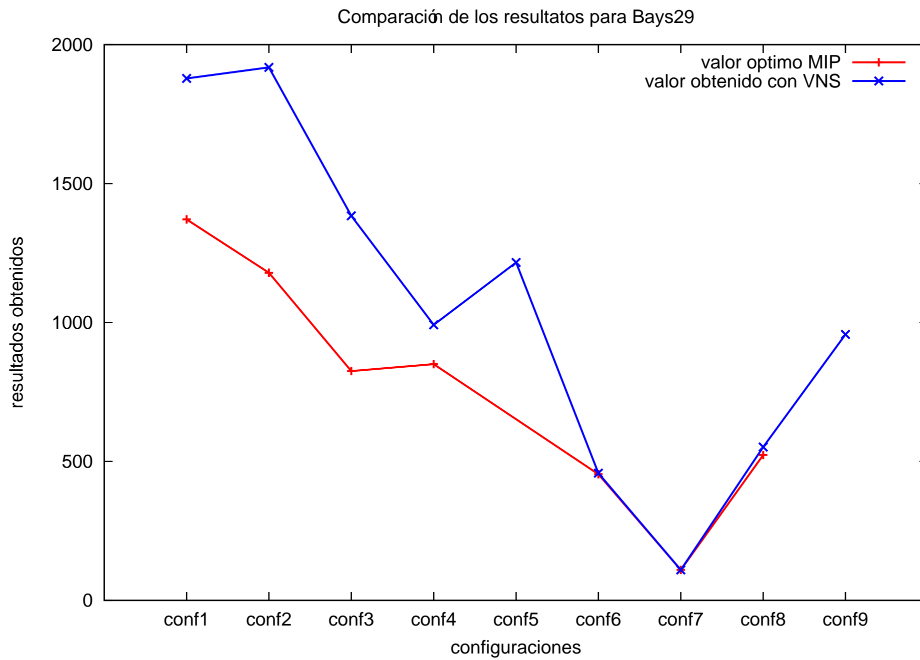


Figura 7.8: Comparativa de los resultados del Método Exacto con los de VNS para Bays29

7.2.2. Resultados de VNS

En esta sección, se presentan los resultados de la heurística VNS con los grafos mostrados en la tabla 7.1. Se registraron los datos obtenidos en la fase de construcción de la solución inicial, en particular, el costo de la solución inicial $C(G_{ini})$ y el tiempo (T_{Ini}), para poder realizar una comparación con los valores obtenidos en la búsqueda local en el algoritmo de VNS. En las tablas 7.11 a 7.13, se observan los resultados alcanzados. Para los casos de prueba sólo se utilizaron las configuraciones 1, 2 y 3, debido a los excesivos tiempos insumidos por VNS en grafos grandes.

En base a los resultados obtenidos, se observa que en la medida que disminuye la cantidad de nodos en el conjunto de terminales, los costos y los tiempos de ejecución de VNS también disminuyen. Es esperable que esto suceda, ya que los nodos opcionales no son tenidos en cuenta en la solución final, salvo que mejoren el costo de dicha solución. Por tanto, la cantidad de nodos procesados es menor al descartar los nodos opcionales no utilizados, logrando de esta forma reducir el tiempo de ejecución del algoritmo.

A pesar de que la calidad de la solución inicial no es buena en comparación con las soluciones finales obtenidas por VNS, ya que estas últimas en promedio mejoran el resultado final por encima de un 60 % para las dos primeras configuraciones y de un 48 % para la última configuración, es importante destacar que la fase de construcción sólo requiere unos pocos segundos para la mayoría de los casos, mientras que con VNS se requiere un tiempo mucho mas grande.

En las Figuras 7.9 a 7.14 se pueden observar algunos ejemplos de los resultados obtenidos.

Instancia	N_X	N_T	N_{Op}	$C(G_{ini})$	$T_{Ini}(s)$	$\bar{C}(Gsol)$	$T_{VNS}(s)$	$C(Gsol_{best})$	**Mejora**
Swiss42	8	25	9	3499	0,024	1188,9	214	1075	**69 %**
Gr48	9	28	11	11406	0,035	4845,1	219,5	3933	**66 %**
Eil51	10	30	11	1017,38	0,042	389,90	686,3	346,21	**66 %**
Berlin52	10	31	11	16.136,7	0,050	7.131,14	574,1	6.703,2	**58 %**
St70	14	42	14	1.858,15	0,110	694,58	5.385,2	618.08	**67 %**
Eil76	15	45	16	1.397,74	0,142	515,55	6.799,2	457,89	**67 %**
Pr76	15	45	16	361.107	0,155	109.619,85	5.938,8	102.673	**72 %**
Rd100	20	60	20	23.333,55	0,325	8.642,60	18.393,8	7947,1	**66 %**
KroA100	20	60	20	63.582	0,390	23.025,47	20.706,7	21.611,9	**66 %**
KroB100	20	60	20	59.521,5	0,385	23.367,95	19.767,6	21.424,75	**64 %**
Pr124	24	74	26	395.449,5	2,534	73.368,25	85.864,4	60.228	**85 %**

Tabla 7.11: Resultados de VNS con la Configuración 1

Instancia	N_X	N_T	N_{Op}	$C(G_{ini})$	$T_{Ini}(s)$	$\bar{C}(Gsol)$	$T_{VNS}(s)$	$C(Gsol_{best})$	**Mejora**
Swiss42	8	16	18	1873	0,017	851,8	100,3	811	**57 %**
Gr48	9	19	20	6866	0,028	3091,8	167,5	2853	**58 %**
Eil51	10	20	21	690,43	0,032	286,52	238,9	265,45	**62 %**
Berlin52	10	20	22	10.559,1	0,035	5756,77	190,6	5587,5	**47 %**
St70	14	28	28	1.213,29	0,075	533,48	4.019,2	509,32	**58 %**
Eil76	15	30	31	870,33	0,106	357,36	4.462,3	337,89	**61 %**
Pr76	15	30	31	213.790	0,120	81.137,5	7.599,2	75.305,5	**65 %**
Rd100	20	40	40	13.625,35	0,220	5.964,20	14.121,8	5.217,65	**62 %**
KroA100	20	40	40	39.097,95	0,230	17.057,7	17.135,1	15.765,75	**60 %**
KroB100	20	40	40	34.912,65	0,235	17.058,11	25.255,3	16.090,6	**54 %**
Pr124	24	49	51	218.277,5	2,105	55.989,4	46.517	50.235,5	**77 %**

Tabla 7.12: Resultados de VNS con la Configuración 2

Instancia	N_X	N_T	N_{Op}	$C(G_{ini})$	$T_{Ini}(s)$	$\bar{C}(Gsol)$	$T_{VNS}(s)$	$C(Gsol_{best})$	**Mejora**
Swiss42	8	8	26	754	0,010	369,1	76,7	348	**54 %**
Gr48	9	9	39	3.588	0,012	3.367,3	107,3	2.336	**35 %**
Eil51	10	10	31	326,98	0,017	189,08	78	186,87	**43 %**
Berlin52	10	10	32	5.335,85	0,017	3.197,10	135,5	3.151,71	**41 %**
St70	14	14	42	653,35	0,044	352,87	1.708,2	340,59	**48 %**
Eil76	15	15	46	436,41	0,056	242,29	2.907,7	219,72	**50 %**
Pr76	15	15	46	82.683,5	0,055	37.013,54	3.229,4	34.299.25	**59 %**
Rd100	20	20	60	7.489,2	0,115	4.188,15	11.083,8	3.943,93	**47 %**
KroA100	20	20	60	25.772,7	0,135	10.281,34	6.343,2	10.094,9	**61 %**
KroB100	20	20	60	18.178,8	0,125	11.561,85	8.792,6	11.139,05	**39 %**
Pr124	24	24	76	57.448	0,825	28.807,34	26.373,1	25.449,75	**56 %**

Tabla 7.13: Resultados de VNS con la Configuración 3

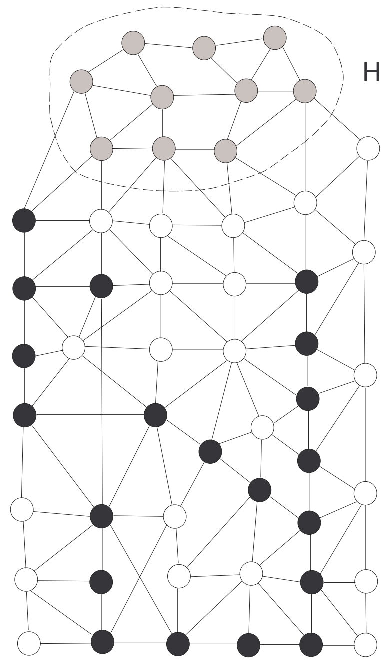


Figura 7.9: Grafo original de Berlin52

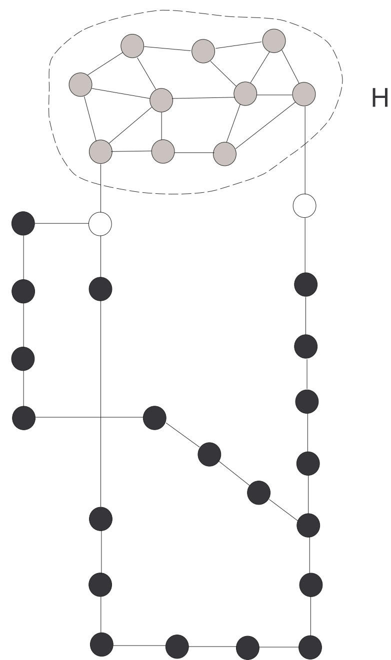


Figura 7.10: Grafo solución de Berlin52

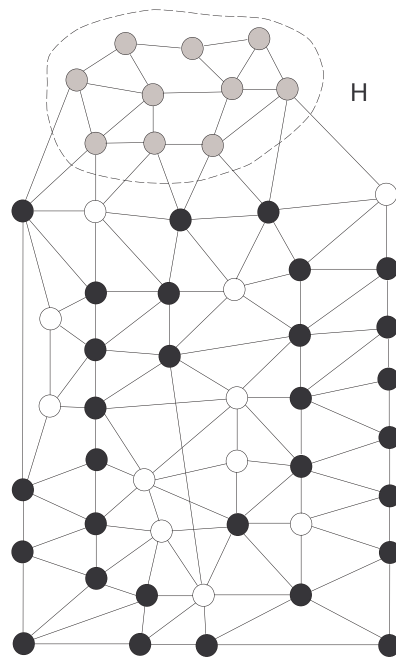


Figura 7.11: Grafo original de Eil51

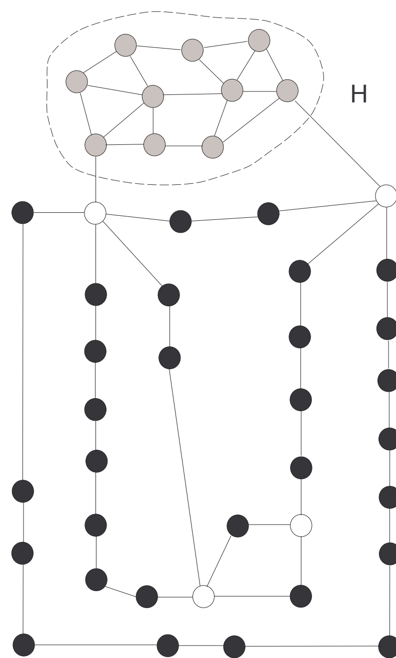


Figura 7.12: Grafo original de Eil51

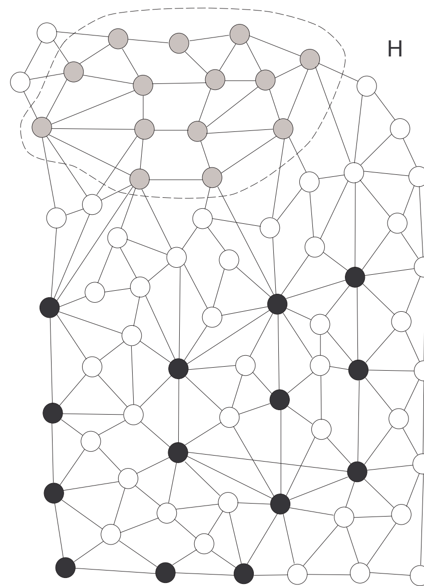


Figura 7.13: Grafo original de St70

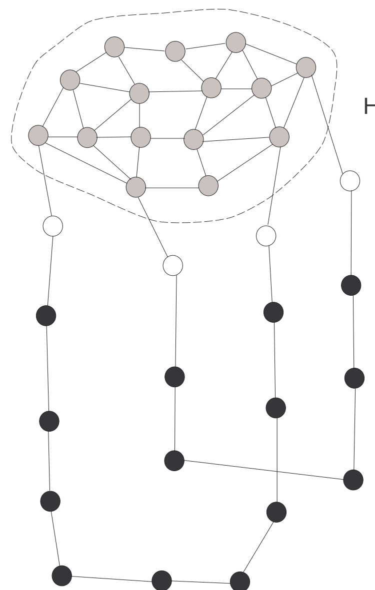


Figura 7.14: Grafo original de St70

7.2.3. Eficacia de las Vecindades

Con el objetivo de comparar la eficacia de cada una de las vecindades definidas, se ejecuta el VNS sin una vecindad, es decir, en el algoritmo de VND no se considera la vecindad que

Vecindades	N_X	N_T	N_{Op}	$C(Gsol_i)$	$T_{VNS}(s)$	$C(Gsol_{best})$
Vecindad 1	15	45	16	300.354,16	7.929,00	267.892,50
Vecindad 2	15	45	16	148.842,00	1.901,14	132.641,50
Vecindad 3	15	45	16	115.773,85	2.565,57	104.885,50
Vecindad 4	15	45	16	115.750,79	6.991,29	109.424,50
Vecindad 5	15	45	16	112.455,79	7.792,71	107.612,50
Vecindad 6	15	45	16	110.942,93	7.531,93	105.265,50
Todas	15	45	16	109.619,85	5.938,80	102.673,00

Tabla 7.14: Resultados de la Eficacia de VNS con la Configuración 1 para el caso de prueba Pr76

se quiere analizar. Esto se repite para cada una de las vecindades y los resultados obtenidos se los compara con el resultado del caso de prueba Pr76 de la Tabla 7.11. El costo obtenido con el algoritmo VNS sin una vecindad, se lo representa con $C(Gsol_i)$, siendo i la vecindad que no fue utilizada. Se utilizó la configuración 1 de nodos para comparar la eficacia de cada vecindad. En la Tabla 7.14 se observan los resultados obtenidos.

Si se aprecian los resultados de la tabla 7.14 puede observarse que las vecindades "Agregar Aristas\ Reducir Redundantes" y "Intercambio de 2 Aristas entre 4 Nodos" son importantes para la obtención de una solución de buena calidad, ya que si alguna de estas vecindades no son tomadas en cuanta en el algoritmo de VNS entonces los resultados obtenidos se alejan de la mejor solución obtenida con todas las vecindades. Cabe destacar que la vecindad "Agregar Aristas\ Reducir Redundantes" es la que tiene mayor impacto en obtener una mejor solución y a la ves es la que insume más tiempo en el algoritmo.

Otro punto que se destaca es que la combinación de todas las vecindades en el mismo algoritmo logra obtener una mejor solución. Esto indica que a mayor combinación de vecindades en la resolución del problema, mayores posibilidades existen de lograr una solución que sea más próxima al óptimo global.

Parte IV

CONCLUSIONES

Capítulo 8

Conclusiones y Trabajos Futuros

En este Capítulo se exponen las conclusiones sobre las implementaciones del método exacto y del VNS para la resolución del ASTNSNP, se plantean posibles trabajos futuros para modificar las soluciones planteadas y finalmente se reflejan las conclusiones generales sobre el proyecto.

8.1. Conclusiones de la Resolución Mediante Método Exacto

Por medio de la definición del modelo matemático presentado en la sección 2.3 y su implementación en CPLEX se logró obtener soluciones óptimas globales, en grafos con menos de 30 nodos, para el problema ASTNSNP planteado en la sección 2.2 de este documento. Dada la complejidad NP-Hard del problema planteado, el modelo desarrollado utilizando CPLEX excede la capacidad de memoria del equipo con grafos de aproximadamente 30 nodos, que luego de realizar el Splitting Vertex se transforman en grafos de 60 nodos. Es importante destacar, que en la gran mayoría de las configuraciones planteadas para los casos de pruebas, los cortos períodos de tiempo (unos pocos segundos) que le insume al solver de CPLEX resolver el método exacto planteado.

8.2. Conclusiones de la Resolución Mediante VNS

Se destaca como principal conclusión que, el algoritmo de VNS implementado, para el problema planteado en la Sección 2.2, encuentra soluciones de buena calidad en los casos de pruebas realizados, dado que los mejores resultados obtenidos logran mejoras, en general, por encima del 60 % del costo de la solución inicial, en los casos de pruebas com más de 40 nodos. Para los casos de prueba chicos, menores a 30 nodos, los resultados obtenidos igualaron al valor óptimo obtenido por el Método Exacto, salvo en 5 resultados donde en promedio el gap

estuvo por debajo del $5\,\%$.

La búsqueda local mediante la incorporación de una arista y la eliminación de aristas redundantes es, ampliamente, la vecindad que aporta una mayor mejora a la solución actual y por tanto su inclusión en el VND es muy importante para la obtención de buenos resultados. A pesar de ello esta vecindad insume un período de tiempo muy extenso en la búsqueda local, esto se debe a que la eliminación de una arista implica la realización previa de una búsqueda BFS, como método para asegurar que dicha eliminación no produce la pérdida de la 2-nodo conectividad. Es importante destacar que la ejecución de todas las vecindades mejora el resultado global, llegando a la conclusión de que a mayor combinación de vecindades en la resolución del problema más próxima estará la solución del óptimo global.

La obtención de la solución inicial mediante el algoritmo de Bhandari tiene algunas características que son importantes de destacar. En primer lugar, la solución inicial obtenida mediante este algoritmo es de buena calidad desde el punto de vista del costo de la misma si la comparamos con el costo del grafo inicial del cual parte. En segundo lugar, la obtención de la solución inicial es muy rápida comparada con los tiempos de procesamiento que insume la ejecución de VNS.

La implementación de una selección randómica de la vecindad inicial para la secuencia de búsquedas locales, aplicadas en el VND, es un buen mecanismo para no caer en óptimos locales de los cuales luego la solución actual no puede escapar.

8.3. Trabajos Futuros

En la construcción de la solución inicial del Algoritmo de VNS se puede obtener grafos sin nodos opcionales o con escasa cantidad de éstos. La falta de nodos opcionales afecta el rendimiento del VNS, ya que este último, no puede realizar búsquedas locales en aquellas vecindades donde se intenta reducir el costo del grafo en función de la existencia de este tipo de nodos. Por tal motivo, un interesante aporte a la solución planteada es la incorporación de nodos opcionales, que no sean parte de la solución actual, en el algoritmo de "Agitación" aplicado sobre el grafo solución obtenido hasta ese momento. Una forma de agregar esto nodos podría ser mediante las sucesivas agregaciones de H-paths que contenga nodos opcionales al grafo solución. Por más detalles de H-path se puede consultar la referencia [32].

Aplicar el algoritmo de Bhandari es la mejor solución para incorporar un nuevo nodo terminal a la solución en construcción desde el punto de vista del costo, ya que obtiene los dos mejores caminos nodos disjuntos con menor costo total combinados. Con esta optimización en los costos en general se reduce la cantidad de nodos opcionales que puede tener la solución inicial, lo cual restringe la capacidad de realizar búsquedas locales en el entorno de dichos nodos. La obtención de una solución inicial diferente mediante la utilización de otro algoritmo, como por ejemplo aplicar dos veces Dijkstra para la obtención de dos caminos nodos disjuntos de un terminal a la solución en construcción o aplicar el algoritmo de Yen [31] que obtiene los k

caminos disjuntos más cortos entre dos nodos de un grafo. Yen establece un ranking para todos los caminos disjuntos encontrados entre dos nodos, de los cuales, los 2 primeros caminos son los que se seleccionaría. Otro algoritmo interesante de utilizar es el Algoritmo de k-caminos más cortos de Hershberger [58]. La utilización de los algoritmos descritos puede implicar una multiplicidad más amplia de la cantidad de nodos opcionales en la construcción de la solución factible inicial.

Después de haber aplicado el Algoritmo de VNS para encontrar un grafo solución de costo mínimo es importante preguntarse si dicho grafo tiene la propiedad de arista-minimal. O sea, al eliminar cualquiera de las arista del grafo, que no son parte de la red ya existente, éste deja de ser una solución. Comprobar esta propiedad sería un buen aporte a la solución planteada en el algoritmo de VNS para saber cuales de los resultados obtenidos con este algoritmo cumple con la propiedad de arista-minimal y así asegurar que el resultado sea de buena calidad.

Una de las principales bondades del algoritmo de VNS es que se lo puede modificar fácilmente agregando y quitando vecindades en la implementación de VND. La definición e incorporación de nuevas vecindades puede ser un muy buen aporte para este trabajo.

8.4. Conclusiones Generales de la Tesis

En este trabajo de tesis de maestría se estudió la forma de ampliar una red 2-nodo-conexa ya existente sobre la cual se desea incorporar nuevos nodos de carácter obligatorios de forma que la nueva red resultante mantenga la 2-nodo-conectividad. Es así que se definió el ASTNSNP en términos de grafos ponderados con costos de conexión entre sus nodos.

Para lograr este objetivo en una primera etapa se utilizó un Método Exacto, el cual se define por medio de un modelo de Programación Lineal Entera basado en el Problema de Steiner Generalizado. Los resultados obtenidos muestran que dicho modelo resuelve de forma satisfactoria el problema planteado en instancias del ASTNSNP de menor de 30 nodos. Luego, dichos resultados son utilizados como medida para evaluar la calidad de la solución obtenida mediante un método heurístico.

Dada la NP-Completitud del ASTNSNP, en una segunda etapa se utilizó la metaurística VNS para encontrar soluciones optimales de buena calidad para el ASTNSNP. Además, se utilizó VNS por la posibilidad de poder definir una vecindad que utilizara una variante del método exacto, planteado en la primera etapa, como método de búsqueda local aplicado a un subconjunto de nodos del grafo. A través de los resultados obtenidos se puede observar que si se agregan o quitan vecindades ello incide en el aumento o disminución del potencial del algoritmo de VNS. En particular se observó que un conjunto más amplio de vecindades logra una solución más cercana al óptimo global, validando de esta manera los principios en que está basado VNS.

Parte V

Anexos

Apéndice A

Biblioteca Boost

Boost [12] es un conjunto de bibliotecas de libre distribución, que mejora las capacidades del lenguaje de programación C++. Su licencia permite que sea utilizada en cualquier tipo de proyecto, ya sean comerciales o no.

Su diseño e implementación, permiten que sea utilizada en un amplio espectro de aplicaciones y plataformas. Abarca desde librerías de propósito general hasta abstracciones del sistema operativo. Con el objetivo de alcanzar el mayor rendimiento y flexibilidad, se hace un amplio uso de templates. Esta librería representa un intenso trabajo e investigación en programación genérica y metaprogramación en C++.

Actualmente, Boost está formada por más de 80 librerías individuales, incluidas las de álgebra lineal, la generación de números pseudoaleatorios, multihilos, procesamiento de imágenes, expresiones regulares, pruebas unitarias, entre otros. La mayoría de las librerías de Boost están basadas según su encabezado (funciones que utilizan templates) por lo que no tienen que ser construidas antes de su uso.

A.1. Boost Graph Library

Boost Graph Library(BGL)[101] [65] es una librería con interfaz genérica que permite acceder a la estructura de grafos, ocultando los detalles de su implementación. Es una interfaz "abierta", en el sentido de que cualquier biblioteca de grafos que implementa esta interfaz se la considera interoperable con los algoritmos genéricos de BGL y con otros algoritmos que también utilizan esta interfaz. BGL proporciona algunas clases genéricas, que se ajustan a las necesidades de estas interfaces pero no son las únicas clases de grafos que se pueden utilizar, sino que hay otras que son mejores para determinados casos. La interfaz de BGL y sus componentes de grafos son de carácter genérico, al igual que Standard Template Library(STL)[2].

A continuación, se muestra el papel que desempeña BGL en la programación genérica.

A.1.1. Generalidad en Boost Graph Library

Existen tres razones por las que se dice que BGL es genérica y se detallan a continuación:

A.1.1.1. Algoritmo / Estructura de datos interoperables

Los algoritmos de BGL, son escritos en un interfaz abstracta que oculta la estructura de datos de los grafos. Esta interfaz genérica, permite que funciones como $breadth_first_search()$ trabajen con una amplia gama de grafos con diferentes estructura de datos, desde grafos implementados con punteros a nodos a grafos implementados con arrays. Si los programadores quieren reusar algunos de estos algoritmos, deben convertir o copiar la información del grafo en las estructuras de grafos ya definidas en la librería.

A.1.1.2. Extensión a través de Visitors

Los algoritmos de BGL se pueden ampliar, BGL introduce la noción de “visitors” que son funciones que pueden tener varios métodos. En los algoritmos de grafos existen eventos claves donde es útil que el usuario pueda agregar una operación propia. Los "visitors"disponen de diferentes métodos que se invocan en estos eventos.

A.1.1.3. Propiedades de vertices y aristas multi-parametrizables

La generalidad de BGL es análoga a la parametrización de los tipos de elementos en STL, el cual asocia valores llamados "propiedades", con los vértices y aristas de los grafos. Las clases de BGL tienen parámetros para las "propiedades"de los vertices y aristas, a las cuales se le asignan una etiqueta que las identifica. Esta etiqueta, se utiliza para distinguir entre las varias propiedades que pueda tener una arista o vértice. Por ejemplo, un valor de una propiedad que está asociada a un vértice, se puede obtener a través de un mapa de la propiedad. BGL define un mapa de propiedades diferentes para cada propiedad existente en el grafo.

A.1.2. Algoritmos

Los algoritmos se pueden diferenciar en un conjunto básico (implementados como algoritmos genéricos) y en un conjunto más amplio de algoritmos de grafos. Los algoritmos principales son:

- Breadth First Search.
- Depth First Search.
- Uniform Cost Search.

BGL además incluye los siguientes algoritmos de grafos:

- Dijkstra’s Shortest Paths.
- Bellman-Ford Shortest Paths.

- Johnson's All-Pairs Shortest Paths.
- Kruskal's Minimum Spanning Tree.
- Prim's Minimum Spanning Tree.
- Connected Components.
- Strongly Connected Components.
- Dynamic Connected Components (using Disjoint Sets).
- Topological Sort.
- Transpose.
- Reverse Cuthill Mckee Ordering.
- Smallest Last Vertex Ordering.
- Sequential Vertex Coloring Vertex.

A.1.3. Estructuras de Datos

BGL ofrece las siguientes estructuras de datos:

- adjacency_list
- adjacency_matrix
- edge_list

La clase adjacency_list es la estructura de datos más utilizada. Es altamente parametrizable, esto significa que los grafos pueden ser dirigidos o no, permitir o no paralelismo de aristas, obtener de forma eficiente las aristas de entrada o salida, rápida inserción o borrado de vertices, entre otras.

La clase adjacency_matriz es una matriz de $V \times V$, donde V es el número de vértices. Los elementos de la matriz, representan las aristas del grafo y esta estructura es utilizada cuando los grafo son muy densos.

La clase edge_list es un adaptador que toma cualquier tipo de iterador de aristas e implementa un grafo de listas de aristas.

Apéndice B

Biblioteca TSPLIB

La biblioteca TSPLIB provee un conjunto de instancias de grafos para resolver el Problema del Viajante de Comercio y otros problemas relacionados, en inglés el mismo se conoce como Traveler Salesman Problem y sus iniciales son TSP, de aquí la biblioteca toma su nombre.

Cada una de las instancias de grafos que provienen de la TSP, viene en un archivo independiente y su extensión nos define el tipo del problema al que se aplica. Para este trabajo, se toman instancias de grafos para el Problema del Viajante de Comercio (archivos con extensión tsp).

B.1. Formato de los archivos TSPLIB

Los archivos de la TSPLIB, tienen dos partes bien definidas, en la primera parte se especifica información general del grafo y los formatos en que están expresados los datos sobre nodos y/o aristas. En la segunda parte se detalla la información de nodos y/o aristas, dependiendo de cual sea el formato definido para expresar la geometría del grafo.

La sección de especificación de formatos, se puede detallar brevemente analizando cada uno de los posibles cabezales (etiquetas), que se detallan a continuación:

- NAME: Nombre que identifica el problema
- TYPE: Especifica el tipo de problema al que refiere el archivo (TSP, ASTP, SOP, HCP, CVRP, TOUR)
- COMMENT: Comentarios adicionales
- DIMENSION: Para los problemas TSP y ATSP, indica la cantidad de nodos del grafo.
- CAPACITY: Para CRVP, indica la capacidad del vehículo.

- EDGE_WEIGHT_TYPE: Especifica como vienen dadas las distancias o pesos de las aristas (EXPLICIT, EUC_2D, EUC_3D, GEO, etc.)

- EDGE_WEIGHT_FORMAT: Describe el formato en que están descritos los pesos de las aristas (FUNCTION, FULL_MATRIX, UPPER_ROW,...). Este cabezal, aplica para el caso que los pesos de las aristas vengan expresados en forma explícita (EDGE_WEIGHT_TYPE = EXPLICIT).

- EDGE_DATA_FORMAT: Describe el formato en el que vienen expresadas las aristas del grafo, se aplica para el caso que este no sea completo. Los valores pueden ser EDGE_LIST O ADJ_LIST.

- NODE_COORD_TYPE: En esta sección, se especifican las coordenadas de cada nodo, las que pueden ser usadas para dibujar el grafo o para calcular las distancias entre ellos. (TWOD_COORDS, THREED_COORDS, NO_COORDS)

- DISPLAY_DATA_TYPE: Describe como se despliegan gráficamente los nodos (COORD_DISPLAY, TWOD_DISPLAY, NO_DISPLAY)

- EOF (opcional): Indica el fin del archivo.

La segunda parte del archivo, esta compuesta por la información sobre nodos o aristas, ésta depende de como está conformada, de acuerdo con las opciones que vengan en los cabezales que ya se describieron. La información sobre nodos o aristas inicia con una etiqueta que identifica su comienzo y puede tener (o no) una etiqueta de fin. En caso de tener información sobre nodos, la etiqueta inicial es NODE_COORD_SECTION mientras que para el caso de aristas es EDGE_DATA_SECTION. Si se tiene información sobre los pesos de las aristas, la etiqueta inicial es EDGE_WEIGHT_SECTION.

Si existen datos sobre nodos, estos vienen a continuación de la etiqueta NODE_COORD_SECTION, donde son listadas las coordenadas de los nodos (un nodo por linea). El formato de estas coordenadas dependen del valor asociado al cabezal NODE_COORD_TYPE. Si NODE_COORD_TYPE tiene asociado TWOD_COORDS, cada línea tiene el formato <INTEGER><REAL><REAL> que indican índice del nodo, coordenada X y coordenada Y. Si NODE_COORD_TYPE tiene asociado THREED_COORDS, cada línea tiene el formato <INTEGER><REAL><REAL><REAL>, que indican índice del nodo, coordenada X, coordenada Y y coordenada Z.

Si existen datos sobre aristas, estos vienen a continuación de la etiqueta EDGE_DATA_SECTION en uno de los dos formatos posibles. Dicho formato queda determinado por el valor asignado al cabezal EDGE_DATA_FORMAT, si este último tiene asignado el valor EDGE_LIST, las aristas vienen dadas en una secuencia de líneas <INTEGER><INTEGER>, que indican los nodos extremos de dichas aristas. Si EDGE_DATA_FORMAT tiene asignado el valor ADJ_LIST, la información consiste en una lista de nodos adyacentes a un nodo dado. En este caso, cada fila es de la forma <INTEGER> <INTEGER> ..... <INTEGER> -1, donde el primer entero indica el índice del nodo X y los siguientes son sus nodos adyacentes; para indicar el fin de la lista la

secuencia termina con -1.

Si vienen dados los datos sobre los pesos de las aristas, estos vienen luego de la etiqueta EDGE_WEIGHT_SECTION. Dichos pesos son representados de acuerdo al formato especificado en EDGE_WEIGHT_FORMAT. La información (si esta presente) viene en una matriz con un tamaño conocido definida por la cantidad de nodos.

Para las distancias es importante destacar, que según el formato de representación indicado en EDGE_WEIGHT_TYPE, la información sobre el peso de las aristas es definida mediante el uso de algunas funciones de distancias conocidas, como por ejemplo la distancia euclidiana para 2 y 3 dimensiones. Las funciones utilizadas, son explicadas en [1].

B.2. Procesamiento de los archivos TSPLIB

En este proyecto, se plantea leer archivo en dos etapas, primero en la especificación del archivo y luego en base a ellas, realizar la lectura de la información sobre aristas y/o nodos. La primera etapa, recorre los cabezales (o etiquetas) y almacena su valor asociado, debido a que estos determinan la posterior lectura de datos sobre aristas y/o nodos. Para este proyecto, el seudocódigo del proceso de lectura de etiquetas es el siguiente:

Algoritmo B.2.1 Leer Cabezales

```
file = abrir archivo TSPLIB
Repetir
  linea = getLine(file);
  etiqueta = getEtiqueta(linea);
  Si (etiqueta == DIMENSION) entonces
    almacenar cantidad de nodos
  Si no Si (etiqueta == EDGE_WEIGHT_TYPE) entonces
    almacenar tipo de medida de la representación del peso de arista
  Si no Si (etiqueta == EDGE_WEIGHT_FORMAT) entonces
    almacenar formato de representación de las aristas
  Si no Si (etiqueta == NODE_COORD_TYPE) entonces
    almacenar el formato de representación de las coordenadas de los nodos
  Si no
    en las otras etiquetas no realiza ninguna acción
  Fin Si
Hasta ((etiqueta == DISPLAY_DATA_TYPE) OR (etiqueta == EOF))
```

En la segunda etapa, se lee la segunda parte del archivo que contiene los datos de los nodos y/o de las aristas, tomando en cuenta las especificaciones de formato leídos en los cabezales. Se diseña un algoritmo de lectura adecuado a cada formato de representación de la información

de nodos y/o aristas. También, se deben implementar las funciones para calcular el costo de las aristas, en el caso que estos no vengan de manera explícita.

Parte VI

Bibliografía

Bibliografía

[1] Tsplib - gerard reinelt's library of tsp instances. http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/.

[2] A. A. Stepanov and M. Lee. The standard template library. Technical Report X3J16/94-0095, WG21/N0482, ISO Programming Language C++ Project, May 1994.

[3] A. Agrawal, P. Klein, and R. Ravi. When trees collide: an approximation algorithm for the generalized Steiner problem on networks. *SIAM Journal on Computing*, 24(3):440–456, 1995.

[4] Ernst Althaus, Tobias Polzin, and Siavash Vahdati Daneshmand. Improving linear programming approaches for the steiner tree problem. In *WEA'03: Proceedings of the 2nd international conference on Experimental and efficient algorithms*, pages 1–14, Berlin, Heidelberg, 2003. Springer-Verlag.

[5] V. Auletta, Y. Dinitz, Z. Nutov, and D. Parente. A 2-approximation algorithm for finding an optimum 3-vertex-connected spanning subgraph. *Journal of Algorithms*, 32(1):21–30, 1999.

[6] Mourad Baïou. *Le problème du suos-graphe Steiner 2-arête connexe: approache polyédrale*. PhD thesis, Université de Rennes I, 1996.

[7] A. Balakrishnan, T. Magnanti, and P. Mirchandi. Connectivity-splitting models for survivable network design. *Networks*, 43(1):10–27, 2004.

[8] K. Berczi and Y. Kobayashi. An algorithm for (n-3)-connectivity augmentation problem: Jump system approach. Technical Report METR 2009-12, Department of Mathematical Engineering, University of Tokyo, April 2009.

[9] R. Bhandari. Optimal physical diversity algorithms and survivable networks. In *Computers and Communications, 1997. Proceedings., Second IEEE Symposium on*, pages 433 – 441, July 1997.

[10] D. Bienstock, E.F. Brickell, and C.L. Monma. On the structure of minimum weight k-connected spanning networks. *SIAM Journal on Discrete Mathematics*, 3(3):320–329, 1990.

[11] H.J. Böckenhauser, D. Bongartz, J. Hromkovič, R. Klasing, G. Proietti, S. Seibert, and W. Unger. On the hardness of constructing minimal 2-connected spanning subgraphs in complete graphs with sharpened triangle inequality. In *Proceedings of the 22nd Conference Kanpur on Foundations of Software Technology and Theoretical Computer Science*, pages 59–70. Springer-Verlag, 2002.

[12] Boost. Boost c++ libraries. http://www.boost.org/.

[13] H. Cancela, F. Robledo, and G. Rubino. A GRASP algorithm for designing a Wide Area Network backbone. In *Proceedings of the International Network Optimization Conference (INOC'03)*, pages 138–143, Evry/Paris, France, October 2003.

[14] H. Cancela, F. Robledo, and O. Viera. A parallel algorithm for the Steiner 2-edge-survivable network problem. *Journal of ICHIO (Chilean Institute of Operations Research)*, 7(1):15–27, 2005.

[15] Yuexuan Wang Wei Wang Nassim Sohaee Changcun Ma, Donghyun Kim and Weili Wu. Hardness of k-vertex-connected subgraph augmentation problem. *Journal of Combinatorial Optimization*, 20(3):249 – 258, 2010.

[16] Zhi-Zhong Chen. Approximating unweighted connectivity problems in parallel. *Information and Computation*, 171:125–126, 2001.

[17] J. Cheriyan, T. Jordan, and Z. Nutov. On rooted node-connectivity problems. *Algorithmica*, 30(3):353–375, 2001.

[18] J. Cheriyan, A. Sebő, and Z. Szigeti. Improving on the 1.5-approximation of a smallest 2-edge connected spanning subgraph. *SIAM Journal on Discrete Mathematics*, 14(2):170–180, 2001.

[19] J. Cheriyan and R. Thurimella. Approximating minimum-size k-connected spanning subgraphs via matching. *SIAM Journal on Computing*, 30:528–560, 2000.

[20] J. Cheriyan, S. Vempala, and A. Vetta. Approximation algorithms for minimum-cost k-vertex-connected subgraphs. In *Proceedings of the 34th Annual ACM Symposium on the Theory of Computing*, pages 306–312, 2002.

[21] Markus Chimani, Maria Kandyba, Ivana Ljubić, and Petra Mutzel. Strong formulations for 2-node-connected steiner network problems. Technical Report TR07-1-008, Technical University of Dortmund, November 2007.

[22] S. Chopra. Polyhedra of the equivalent subgraph problem and some edge connectivity problems. *SIAM Journal on Discrete Mathematics*, 5(3):321–337, 1992.

[23] W. Chou and H. Frank. Survivable communication networks and the terminal capacity matrix. *IEEE Transactions on Circuit Theory*, 17:192–197, 1970.

[24] N. Christofides and C.A. Whitlock. An algorithm for the design of optimal invulnerable networks. Technical Report IC-OR-81-6, Imperial College, London, 1981.

[25] Lloyd W. Clarke. Embedding cplex using ilog concert technology. http://www.lkn.ei.tum.de/arbeiten/faq/man/ConcertTechnology.pdf.

[26] Michele Conforti, Anna Galluccio, and Guido Proietti. Edge-connectivity augmentation and network matrices. In Juraj Hromkovic, Manfred Nagl, and Bernhard Westfechtel, editors, *WG*, volume 3353 of *Lecture Notes in Computer Science*, pages 355–364. Springer, 2004.

[27] R. Coullard, A. Rais, R.L. Rardin, and D.K. Wagner. Linear-time algorithm for the 2-connected Steiner subgraph problem on special classes of graphs. Technical Report No. 91-25, School of Industrial Engineering, Purdue University, 1991.

[28] ILOG CPLEX. Ilog cplex 11.0 - getting started, Setiembre 2007.

[29] B. Csaba, M. Karpinski, and P. Krysta. Approximability of dense and sparse instances of minimum 2-connectivity, TSP and path problems. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 74–83, 2002.

[30] A. Czumaj and A. Lingas. On approximability of the minimum-cost k-connected spanning subgraph problem. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 281–290, 1999.

[31] Ernesto de Queirós Vieira Martins and Marta M. B. Pascoal. A new implementation of yen's ranking loopless paths algorithm. *In Proceedings of 4OR*, pages 121–133, 2003.

[32] R. Diestel. *Graph theory*. Springer-Verlag, 2 edition, 2000.

[33] Dijkstra, E. W. *A note on two problems in connexion with graphs*, volume 1, pages 269–271. Springer, Berlin / Heidelberg, 1959.

[34] Eliseo Melgarejo. Taller de Ilog Cplex. http://www2.udec.cl/ eliseomelgarejo/.

[35] C.G. Fernandes. A better approximation ratio for the minimum k-edge-connected spanning subgraph problem. *Journal of Algorithms*, 28(1):105–124, 1998.

[36] Gerardo Rubino Franco Robledo Amoza, Hector Cancela. Solving the steiner two-node-survivable network problem. *International Journal of Logistics Systems and Management*, 6(2):218 – 234, January 2010.

[37] H. Frank and W. Chou. Connectivity considerations in the design of survivable networks. *IEEE Transactions on Circuit Theory*, 17:486–490, 1970.

[38] G.N. Frederickson and J. Jàjà. On the relationship between biconnectivity augmentation and traveling salesman problem. *Theoretical Computer Science*, 19:189–201, 1982.

[39] H.N. Gabow, M.X. Goemans, and D.P. Williamson. An efficient approximation algorithm for the survivable network design problem. In *Proceedings of the 3rd MPS Conference on Integer Programming and Combinatorial Optimization*, pages 57–74, 1993.

[40] A. Galluccio and G. Proietti. Polynomial time algorithms for edge-connectivity augmentation problems. *Algorithmica*, 36(4):361–374, 2004.

[41] Anna Galluccio and Guido Proietti. Polynomial time algorithms for edge-connectivity augmentation of hamiltonian paths. In *Proceedings of the 12th International Symposium on Algorithms and Computation*, ISAAC '01, pages 345–354, London, UK, 2001. Springer-Verlag.

[42] M.X. Goemans and D.J. Bertsimas. Survivable networks, linear programming relaxations and the parsimonious property. *Mathematical Programming*, 60:143–166, 1993.

[43] M.X. Goemans, A.V. Goldberg, S. Plotkin, É. Tardos, and D.P. Williamson. Improved approximation algorithms for network design problems. In *Proceedings of the 5th ACM-SIAM Symposium on Discrete Algorithms*, pages 223–232, 1994.

[44] M.X. Goemans and D.P. Williamson. A general approximation technique for constrained forest problems. *SIAM Journal on Computing*, 24(2):296–317, 1992.

[45] M. Grötschel and C.L. Monma. Integer polyhedra associated with certain network design problems with connectivity constraints. *SIAM Journal on Discrete Mathematics*, 3:502–523, 1990.

[46] M. Grötschel, C.L. Monma, and M. Stoer. Polyhedral Approaches to Network Survivability. In F. Roberts, F. Hwang, and C.L. Monma, editors, *Reliability of Computer and Communication Networks, Proc. Workshop 1989, New Brunswick, NJ/USA*, volume 5 of *Series in Discrete Mathematics and Theoretical Computer Science*, pages 121–141. American Mathematical Society, 1991.

[47] M. Grötschel, C.L. Monma, and M. Stoer. Computational results with a cutting plane algorithm for designing communication networks with low-connectivity constraints. *Operations Research*, 40(2):309–330, 1992.

[48] M. Grötschel, C.L. Monma, and M. Stoer. Facets for polyhedra arising in the design of communication networks with low-connectivity constraints. *SIAM Journal on Optimization*, 2(3):474–504, 1992.

[49] M. Grötschel, C.L. Monma, and M. Stoer. Polyhedral and computational investigations for designing communications networks with high survivability requirements. *Operations Research*, 43(6):1012–1024, 1995.

[50] Guy Kortsarzb Guy Even and Zeev Nutov. A 1.5-approximation algorithm for augmenting edge-connectivity of a graph from 1 to 2. *Journal of Combinatorial Optimization*, 111:296 – 300, 2011.

[51] Pierre Hansen and Nenad Mladenovic. Variable neighborhood search: Principles and applications. *European Journal of Operational Research*, 130(3):449–467, 2001.

[52] Pierre Hansen and Nenad Mladenovic. Variable neighborhood search methods. In *Encyclopedia of Optimization*, pages 3975–3989. 2009.

[53] Pierre Hansen, Nenad Mladenovic, and José A. Moreno-Pérez. Búsqueda de entorno variable. *Inteligencia Artificial, Revista Iberoamericana de Inteligencia Artificial*, 7(19):77–92, 2003.

[54] Pierre Hansen, Nenad Mladenovic, and José A. Moreno-Pérez. Variable neighbourhood search: methods and applications. *Annals OR*, 175(1):367–407, 2010.

[55] Pierre Hansen, Nenad Mladenovic, and Dionisio Pérez-Brito. Variable neighborhood decomposition search. *J. Heuristics*, 7(4):335–350, 2001.

[56] F. Harary. The maximum connectivity of a graph. *Proc. Nat. Acad. Sciences USA*, 48(1):1142–1146, 1962.

[57] Ruprecht-Karls-Universitat Heidelberg. Tsplib network optimization problems library. http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95.

[58] John Hershberger, Matthew Maxel, and Subhash Suri. Finding the k shortest simple paths: A new algorithm and its implementation. *ACM Transactions on Algorithms (TALG)*, 3(45), November 2007.

[59] Tsan-Sheng Hsu and Ming-Yang Kao. A unifying augmentation algorithm for two-edge connectivity and biconnectivity. *Journal of Combinatorial Optimization*, 2(3):237 – 256, 1998.

[60] IBM España. ïbm ilog cplex". "http://www-142.ibm.com/software/products/es/es/ilogcple/", Setiembre 2010.

[61] ILOG Inc. ILOG CPLEX 11.0 User's Manual. http://www.lingnan.net/lab/uploadfile/200864184419679.pdf, Setiembre 2007.

[62] J. Brito Santana, C. Campos Rodríguez, F.C. García López, M. Gacía Torres, B.M Batizta, J.A. Moreno Perez, J.M. Moreno Vega . Metaherísticas: una revisión actualizada. Technical Report 02, Departamento de Estadística, Investigación Operativa y Computación, Junio 2004.

[63] Bill Jacksona and Tibor Jordánb. Independence free graphs and vertex connectivity augmentation. *Journal of Combinatorial Theory*, 94:31 – 77, May 2005.

[64] K. Jain. A 3-approximation algorithm for finding optimum 4,5-vertex-connected spanning subgraphs. *Journal of Algorithms*, 32(1):31–40, 1999.

[65] Jeremy G. Siek, Lie-Quan Lee, Andrew Lumsdaine. *Boost Graph Library, The: User Guide and Reference Manual*. Addison-Wesley Professional. Part of the C++ In-Depth Series series, 2002.

[66] Ellis L. Johnson, George L. Nemhauser, Martin, and W. P. Savelsbergh. Progress in linear programming-based algorithms for integer programming: An exposition. *INFORMS Journal on Computing*, 12:2–23, 2000.

[67] T. Jordan. On the optimal vertex-connectivity augmentation. *Journal of Combinatorial Theory*, Series B 63:8–20, 1995.

[68] H. Kerivin and A. Mahjoub. Design of Survivable Networks: A survey. *Networks*, 46(1):1 – 21, 2005.

[69] Hervé Kerivin and A. Ridha Mahjoub. Design of survivable networks: A survey. *Journal of Networks*, 46(1):1 – 21, August 2005.

[70] S. Khuller and B. Raghavachari. Improved approximation algorithms for uniform connectivity problems. *Journal of Algorithms*, 21(2):434–450, 1996.

[71] S. Khuller and R. Thurimella. Approximation algorithms for graph augmentation. *Journal of Algorithms*, 4(2):214–225, 1991.

[72] S. Khuller and U. Vishkin. Biconnectivity approximations and graph carvings. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on the Theory of Computing*, pages 759–770, 1992.

[73] C.W. Ko and C.L. Monma. Heuristics methods for designing highly survivable communication networks. Technical report, Bellcore, 1989.

[74] G. Kortsarz, R. Krauthgamer, and J. R. Lee. Hardness of approximation for vertex-connectivity network-design problems. *SIAM Journal on Computing*, 33(3):704–720, 2004.

[75] G. Kortsarz and Z. Nutov. Approximating node connectivity problems via set covers. *Algorithmica*, 37(2):75–92, 2003.

[76] P. Krysta and V.S.A. Kumar. Approximation algorithms for minimum size 2-connectivity problems. In *Proceedings of the 18th Annual Symposium Theoretical Aspects of Computer Science (STACS'01, Berlin), Lecture Notes in Computer Science*, volume 2010, pages 431–442. Springer, 2001.

[77] Ivana Ljubić, René Weiskircher, Ulrich Pferschy, Gunnar W. Klau, Petra Mutzel, and Matteo Fischetti. An algorithmic framework for the exact solution of the prize-collecting steiner tree problem. *Mathematical Programming*, 105:427–449, 2006. 10.1007/s10107-005-0660-x.

[78] Abilio Lucena and Mauricio G.C. Resende. Strong lower bounds for the prize collecting steiner problem in graphs. *Brazilian symposium on graphs, algorithms and combinatorics*, 141, May 2004.

[79] John E. Mitchell. Integer programming: Branch-and-cut algorithms. *Encyclopedia of Optimization*, 2:519–525, Agosto 2001.

[80] John E. Mitchell. Branch-and-cut algorithms for combinatorial optimization problems. In *Handbook of Applied Optimization*, pages 65–77. Oxford University Press, January 2002.

[81] Nenad Mladenovic. A variable neighborhood algorithm - a new metaheuristic for combinatorial optimization. In *Abstracts of Papers Presented at Optimization Days*, page 112, 1995.

[82] Nenad Mladenovic and Pierre Hansen. Variable neighborhood search. *Computers & OR*, 24(11):1097–1100, 1997.

[83] C.L. Monma, B.S. Munson, and W.R. Pulleyblank. Minimum-weight two connected spanning networks. *Mathematical Programming*, 46:153–171, 1990.

[84] C.L. Monma and D.F. Shallcross. Methods for designing communication networks with certain two-connected survivability constraints. *Operations Research*, 37:531–541, 1989.

[85] Hiroshi Nagamochi. A 4/3 -approximation for the minimum 2-local-vertex-connectivity augmentation in a connected graph. *J. Algorithms*, 56:77–95, August 2005.

[86] Sergio Nesmachnow. Diseño de redes de comunicaciones confiables - el problema de steiner generalizado. http://www.fing.edu.uy/ sergion/gp/documentos/propios/DRCC.pdf, Agosto 2002.

[87] Z. Nutov and M. Penn. Faster approximation algorithms for weighted triconnectivity augmentation problems. *Operations Research Letters*, 21:219–223, 1997.

[88] Zeev Nutov. Approximating node-connectivity augmentation problems. In *Proceedings of the 12th International Workshop and 13th International Workshop on Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, APPROX '09 / RANDOM '09, pages 286–297, Berlin, Heidelberg, 2009. Springer-Verlag.

[89] Alfredo García Olaverri, Ferran Hurtado, Marc Noy, and Javier Tejel. Augmenting the connectivity of outerplanar graphs. *Algorithmica*, 56(2):160–179, 2010.

[90] Doowon Paik, Sudhakar M. Reddy, and Sartaj Sahni. Vertex splitting in dags and applications to partial scan designs and lossy circuits. *Int. J. Found. Comput. Sci.*, 9(4):377–398, 1998.

[91] Michael Penn and Haya Shasha-Krupnik. Improved approximation algorithms for weighted 2- and 3- vertex connectivity augmentation problems. *Journal of Algorithms*, 22:187–196, 1997.

[92] Pierre Hansen and Nenad Mladenovic. *Variable Neighborhood Search*, volume 57 of *International Series in Operations Research, Management Science*, chapter 6 en Handbook of Metaheuristics, pages 145–184. Ins: F. Glover, G.A. Kochenberber. Kluwer Academics, Springer, New York, 2003.

[93] J.S. Provan and R.C. Burk. Two-connected augmentation problems in planar graphs. *Journal of Algorithms*, 32:87–107, 1999.

[94] R. Ravi and P.N. Klein. When cycles collapse: a general approximation technique for constrained 2-connectivity problems. In *Proceedings of the 3rd Symposium on Integer Programming and Combinatorial Optimization*, pages 39–55, 1993.

[95] R. Ravi and D.P. Williamson. An approximation for minimum-cost vertex-connectivity problems. *Algorithmica*, 18(1):21–43, 1997.

[96] R. Ravi and D.P. Williamson. Erratum: An approximation for minimum-cost vertex-connectivity problems. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1000–1001, 2002.

[97] F. Robledo and E. Canale. Designing backbone networks using the generalized steiner problem. In *Design of Reliable Communication Networks, 2009. DRCN 2009. 7th International Workshop on*, volume 1, pages 327 – 334, October 2009.

[98] Franco Robledo. *GRASP heuristics for Wide Area Network design*. PhD thesis, IRISA, Université de Rennes I, Rennes, France, february 2005.

[99] Franco Robledo Amoza. Diseño topológico de redes; Casos de estudio: The Generalized Steiner Problem and the Steiner 2-Edge-Connected Subgraph Problem. Master Thesis, Universidad de la República-Facultad de Ingeniería, J. Herrera y Reissig 565, Montevideo, Uruguay, 2000.

[100] Ignaz Rutter and Alexander Wolff. Augmenting the connectivity of planar and geometric graphs. *Electronic Notes in Discrete Mathematics*, 31:53–56, 2008.

[101] Jeremy Siek and a University of Notre Dame team. The boost graph library. http://www.boost.org/doc/libs/1_44_0/libs/graph/doc/index.html.

[102] Stefan Thienel. Abacus - a branch and cut system - user's guide and reference manual - version 3.0. www.informatik.uni-koeln.de/abacus/userguide/manual.htm, August "2007".

[103] Stefan Thienel, Micheal Jünker. *Introduction to ABACUS - A Branch And CUt System*, volume 22. Ïnstitut Für Informatik - Universität zu Köln".

[104] K. Steiglitz, P. Weiner, and D.J. Kleitman. The design of minimum-cost survivable networks. *IEEE Transactions on Circuit Theory*, 16:455–460, 1969.

[105] M. Stoer. *Design of survivable networks*, volume 1531 of *Lecture Notes in Mathematics*. Springer-Verlag, 1992.

[106] Hiroshi Nagamochi Toshimasa Ishii and Toshihide Ibaraki. Optimal augmentation of a 2-vertex-connected multigraph to a k-edge-connected and 3-vertex-connected multigraph. *Journal of Combinatorial Optimization*, 4(1):35 – 77, 2000.

[107] Csaba D. Tóth. Connectivity augmentation in plane straight line graphs. *Electronic Notes in Discrete Mathematics*, 31:49–52, 2008.

[108] S. Vempala and A. Vetta. Factor 4/3 approximations for minimum 2-connected subgraphs. In *Proceedings of the Third International Workshop on Approximation Algorithms for Combinatorial Optimization Problems (Berlin, Germany)*, pages 262–273, 2000.

[109] T. Watanabe and A. Nakamura. A minimum 3-connectivity augmentation of a graph. *Joournal of Computer and System Sciences*, 46(1):91–128, 1993.

[110] Toshimasa Watanabe and Akira Nakamura. Edge-connectivity augmentation problems. *Journal of Computer and System Sciences*, 35:96 – 144, February 1987.

[111] Toshimasa Watanabe and Akira Nakamura. A minimum 3-connectivity augmentation of a graph. *Journal of Computer and System Sciences*, 46:91 – 128, 1990.

[112] D.P. Williamsom, M.X. Goemans, M. Mihail, and V.V. Vazirani. A primal-dual approximation algorithm for the generalized Steiner network problem. *Combinatorica*, 15:435–454, 1995.

[113] P. Winter. Generalized Steiner problem in outerplanar graphs. *BIT Numerical Mathematics*, 25(3):485–496, 1985.

[114] P. Winter. Generalized Steiner problem in series-parallel networks. *Journal of Algorithms*, 7:549–566, 1986.

[115] P. Winter. Steiner problem in networks: A survey. *Networks*, 17(2):129–167, 1987.