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

"2-node-connected star problem"

Rodrigo Recoba

2016

Diseño topológico de redes
Caso de estudio
“2-node-connected star problem”
Alvaro Recoba
ISSN 0797-6410
Tesis de Maestría en Informática
Reporte Técnico RT 16-08
PEDECIBA
Instituto de Computación – Facultad de Ingeniería
Universidad de la República.
Montevideo, Uruguay, 2016

TESIS DE MAESTRÍA

Diseño Topológico de Redes
Caso de Estudio
"2-Node-Connected Star Problem"

Fecha: 21 de Agosto de 2008
Autor: Ing. Rodrigo Recoba (rrecoba@gmail.com)
Tutor: Dr. Ing. Franco Robledo – UDELAR – INCO (frobledo@fing.edu.uy)

Resumen

En este trabajo se propone un nuevo problema de optimización combinatoria que denominamos Two-Node-Connected Star Problem (2NCSP), el cual, dado un grafo simple con costos de ruteo y costos de asignación, tiene como objetivo encontrar un subgrafo de costo mínimo conformado por una componente 2-nodo conexa mientras que el resto de los nodos están asignadas a la misma mediante un enlace directo.

El 2NCSP es una generalización del conocido Ring Star Problem [1] el cual difiere en que la componente 2-nodo conexa tiene que tener necesariamente topología de anillo.

El objetivo de este trabajo es definir formalmente el problema Two-Node-Connected Star Problem y su resolución mediante una metaheurística GRASP de buen desempeño. Se diseñaron e implementaron búsquedas locales basadas en modelos de programación lineal entera y búsquedas locales que generan movimientos tradicionales para este tipo de problemas.

Los resultados obtenidos muestran una buena performance del algoritmo en relación a instancias de prueba diseñadas y publicadas por otros autores. Dichas instancias de prueba consideran grafos de entre 50 y 200 nodos de la TSPLIB, donde los costos de asignación y conexión se obtienen con un factor de ponderación de la distancia euclidiana entre los nodos. Dicho factor permite determinar que la componente 2-nodo conexa de la solución deba contener la mayoría de los nodos, o solo unos pocos.

Palabras Clave: Diseño Topológico de Redes, Greedy Randomized Adaptive Search Procedure (GRASP), Metaheurísticas, Ring Star Problem (RSP).

Agradecimientos

Agradezco al Dr. Pablo Sartor y al Dr. Pedro Piñeyro por haber aceptado juzgar este trabajo y acceder a participar en el jurado. Agradezco igualmente al Dr. Victor Albornoz por aceptar ser el revisor de este trabajo

Agradezco también al Prof. Ing. Omar Viera por su apoyo incondicional, su comprensión y eterna paciencia.

Especialmente quiero agradecer al Dr. Franco Robledo por su empuje, dedicación, y constante apoyo a mi trabajo, por confiar en mí y por guiarme en este proyecto, sin su colaboración, no hubiera sido posible su finalización.

También agradezco al Msc. Gabriel Baya por haber ofrecido su trabajo, como fuente de evaluación de esta tesis.

Por último, quiero agradecer a mis padres Virginia y Luis, quienes me dieron las herramientas y la convicción de que con esfuerzo y perseverancia, los objetivos se alcanzan, sé que estarán orgullosos en este momento. A mis hijas Sofía y Lucía por darme el empuje de cada día, y a mi esposa Silvana por su apoyo incondicional y su ayuda en la redacción de este trabajo.

Contenido

Resumen

En este trabajo se propone un nuevo problema de optimización combinatoria que denominamos Two-Node-Connected Star Problem (2NCSP), el cual, dado un grafo simple con costos de ruteo y costos de asignación, tiene como objetivo encontrar un subgrafo de costo mínimo conformado por una componente 2-nodo conexa mientras que el resto de los nodos están asignadas a la misma mediante un enlace directo.

El 2NCSP es una generalización del conocido Ring Star Problem [1] el cual difiere en que la componente 2-nodo conexa tiene que tener necesariamente topología de anillo.

El objetivo de este trabajo es definir formalmente el problema Two-Node-Connected Star Problem y su resolución mediante una metaheurística GRASP de buen desempeño. Se diseñaron e implementaron búsquedas locales basadas en modelos de programación lineal entera y búsquedas locales que generan movimientos tradicionales para este tipo de problemas.

Los resultados obtenidos muestran una buena performance del algoritmo en relación a instancias de prueba diseñadas y publicadas por otros autores. Dichas instancias de prueba consideran grafos de entre 50 y 200 nodos de la TSPLIB, donde los costos de asignación y conexión se obtienen con un factor de ponderación de la distancia euclidiana entre los nodos. Dicho factor permite determinar que la componente 2-nodo conexa de la solución deba contener la mayoría de los nodos, o solo unos pocos.

Palabras Clave: Diseño Topológico de Redes, Greedy Randomized Adaptive Search Procedure (GRASP), Metaheurísticas, Ring Star Problem (RSP).

CAPÍTULO 1

1. Introducción

1.1. Motivación

Año tras año, las redes de comunicaciones han ido tomando mayor relevancia en nuestra sociedad, desde la creación del telégrafo y principalmente del teléfono, se fue desarrollando y perfeccionando una infraestructura de comunicaciones que permite ofrecer estos servicios de forma cada vez más eficiente.

Con el paso del tiempo, nuevas aplicaciones surgieron sobre esta infraestructura, como ser la transmisión digital de datos (audio, video e información), y con ello, el uso cada vez más masivo e intensivo de las redes de comunicaciones donde el tráfico llegó a tasas de crecimiento del 100% anual. Este crecimiento motivó a las compañías de telecomunicaciones a intensificar sus esfuerzos en el diseño de redes que permitieran una mayor cobertura al menor costo posible.

Esta forma de planificar y diseñar las redes de comunicaciones provocó que ante sucesos imprevistos, cientos y miles de clientes quedaran sin servicio, como ser el incendio de una central telefónica en Chicago en Mayo de 1988, que dejó sin comunicación a 35.000 abonados y afectó 120.000 líneas troncales de larga distancia, comprometiendo el funcionamiento del aeropuerto de O´Hare (Estados Unidos), y dejando fuera de servicio el número de emergencia 911, como se detalla en el informe Keeping the Phone Lines Open de Zorpette [2].

Estos acontecimientos y el advenimiento de la fibra óptica con su gran capacidad por cable, fue modificando la forma de planificar las redes y por ende la topología de las mismas, haciéndolas cada vez más dispersas, con menor cantidad de conexiones, pero fundamentalmente con la capacidad de tolerar fallas en sus nodos o enlaces.

Desde entonces, la motivación del diseño topológico de redes de telecomunicaciones, es obtener estructuras capaces de satisfacer una cantidad preestablecida de fallas en nodos o enlaces, con la mayor cobertura posible, de forma tal de minimizar el costo total de su construcción.

En este trabajo, presentamos un nuevo problema de optimización combinatoria denominado Two-Node-Connected Star Problem (2NCSP), el cual asegura la supervivencia de la red ante la falla de uno de sus nodos o aristas.

Además de introducir el problema, diseñamos e implementamos una metaheurística para su resolución, la cual es enriquecida con modelos de programación lineal entera para encontrar mejores caminos y componentes 2-nodo conexas de costo mínimo.

1.2. Definición del *"2-Node-Connected Star Problem"*

En esta sección haremos una descripción conceptual del problema 2-Node-Connected Star, al que también denominaremos 2NCSP, y una descripción formal del mismo.

El nombre 2-connected Star se debe al uso de la nomenclatura de Klincewicz [3]. Esta nomenclatura se basa en que, en general, las redes de comunicaciones tienen dos componentes bien definidas: un backbone y una red de acceso. El backbone está conformado por nodos estratégicos para el funcionamiento de la red, mientras que la red de acceso es conformada por nodos menos vitales, como ser, los usuarios finales. Dicha nomenclatura consiste en mencionar primero la topología de backbone y en segunda instancia la topología de la red de acceso.

Por lo tanto, en nuestro problema 2-Connected Star, la componente 2-nodo conexa refiere a los nodos de mayor relevancia, mientras que los nodos de menor importancia se conectan mediante enlaces directos a los anteriores, conformando una topología de estrella.

1.2.1. Descripción Conceptual del Problema 2NCSP

Dada una red conformada por nodos y enlaces entre ellos, el objetivo del problema 2NCSP es encontrar una subred de costo mínimo, en donde los nodos, o bien se encuentran en una única componente con topología 2-nodo conexa (ver definición 1.2.7 con k=2), o se encuentran conectados un nodo de dicha componente, mediante una única arista (denominados nodos colgantes).

El objetivo del problema es minimizar la suma de dos costos: el costo de la componente 2-nodo conexa, y el costo de asignar los nodos que no se encuentran en la componente 2-nodo conexa con el nodo más cercano de ésta.

El modelado del problema se realizará mediante un Grafo, en donde, los vértices representan los nodos de la red y la existencia de una arista entre cualquier par de vértices implica que existe una conexión factible entre ellos.

Los términos "nodo" o "terminal" referirán a cualquier vértice del grafo y serán utilizados indistintamente.

El grafo *G*, tendrá asociadas dos matrices de costo. Una de las matrices determina el costo de conectar cada par de vértices en el caso que ambos formen parte de la estructura 2-nodo conexa (denominado costo de conexión), mientras que la otra matriz, determina el costo cuando uno de los nodos pertenece a la estructura 2-nodo conexa y el otro está conectado al anterior (costo de asignación) pero no forma parte de la estructura 2-nodo conexa (denominado nodo colgante).

Nuestro objetivo entonces es encontrar un subgrafo que cumpla con las restricciones previamente mencionadas minimizando la suma de los costos de conexión y asignación.

1.2.2. Descripción Formal del Problema

A continuación presentaremos algunas definiciones necesarias para la formalización del problema.

Definición 1.2.1: Grado de un nodo
Dado un grafo $G = (V, E)$ donde V es un conjunto de vértices y E un conjunto de aristas, y un vértice $i \in V$, llamamos grado del vértice i (al que denotamos $\delta(i)$) a la cantidad de aristas incidentes a él.

Definición 1.2.2: Camino
Para cada par de nodos $s, t \in V$, un $[s, t]$-camino P es una secuencia de nodos y aristas $[v_0, e_1, v_1, e_2, \ldots\ldots, v_{l-1}, e_l, v_l]$ donde cada arista e_i es incidente a los nodos v_{i-1}, y v_i con $i = [1..l]$ y donde $v_0 = s$ y $v_l = t$ y ningún nodo o arista aparece más de una vez en P.

Definición 1.2.3: Caminos aristas/nodos disjuntos
Una colección $[P_1, P_2, \ldots, P_k]$ de $[s, t]$-caminos P se denomina arista disjunto, si ninguna arista aparece en más de un camino, y es denominado nodo disjunto, si ningún nodo aparece en más de un camino exceptuando s y t.

Definición 1.2.4: k-arista-conectividad
Sea $G = (V, E)$ un grafo no dirigido donde V es un conjunto de vértices y E un conjunto de aristas, se dice que una pareja de nodos $(s, t) \in VxV$ tiene k-arista-conectividad en G, cuando existen al menos k $[s, t]$caminos arista disjuntos.

Definición 1.2.5: k-arista-conexo
Se dice que un grafo $G = (V, E)$ donde V es un conjunto de vértices y E un conjunto de aristas, es k-arista-conexo cuando toda pareja de nodos i, j del mismo está k-arista-conectada.

En forma análoga se realizan las mismas definiciones para nodo conectividad.

Definición 1.2.6: k-nodo-conectividad
Sea $G = (V, E)$ un grafo no dirigido donde V es un conjunto de vértices y E un conjunto de aristas, se dice que una pareja de nodos $(s, t) \in VxV$ tiene k-nodo-conectividad en G, cuando existen al menos k $[s, t]$-caminos nodo disjuntos.

Definición 1.2.7: k-nodo-conexo
Se dice que un grafo $G = (V, E)$ donde V es un conjunto de vértices y E un conjunto de aristas, es k-nodo-conexo cuando toda pareja de nodos i, j del mismo está k-nodo-conectada.

Nótese que si dos caminos son nodo-disjuntos, entonces también son arista disjuntos, pero dos caminos que son arista-disjuntos no necesariamente son nodo disjuntos.

Definición 1.2.8: Topología 2-Nodo Conexa Estrella (2NCS)
Sea $G = (V, E)$ un grafo donde V es el conjunto de vértices y E el conjunto de aristas, se dice que G tiene topología 2NCS si $G = T \cup S$ tal que:

- $T = (U', E')$ *con* $U' \subseteq V$ *y* $E' \subseteq E$ *es un grafo 2-conexo*
- $S = (V'' \cup U'', E'')$ *donde*
  - $V'' = V \,/\, U',\ U'' \subseteq U'$
  - $E''\{(u, v), u \in U'', v \in V''\}, E'' \subset E$
  - $\delta(v) = 1,\ \forall v \in V''$

Con estas definiciones es posible entonces definir nuestro problema de la siguiente forma:

- Sea $G = (V, E)$ un grafo simple no dirigido donde V es el conjunto de vértices y E es el conjunto de aristas.

- Sea $C = \{c_{ij}\}_{i,j \in V}$ la matiz de costos de conexión del grafo, es decir, los costos de las aristas (i, j) si estas pertenecen a T (la componente 2-nodo conexa).

- Sea $D = \{d_{ij}\}_{i,j \in V}$ la matiz de costos de asignación del grafo, es decir, los costos de las aristas (i, j) si la arista pertenece a S (arista colgante).

El problema 2NCS consiste en encontrar un subgrafo con topología 2-nodo conexa minimizando la suma de los costos de asignación más los costos de conexión (i.e. costo de la componente 2-nodo conexa más el costo de las aristas colgantes).

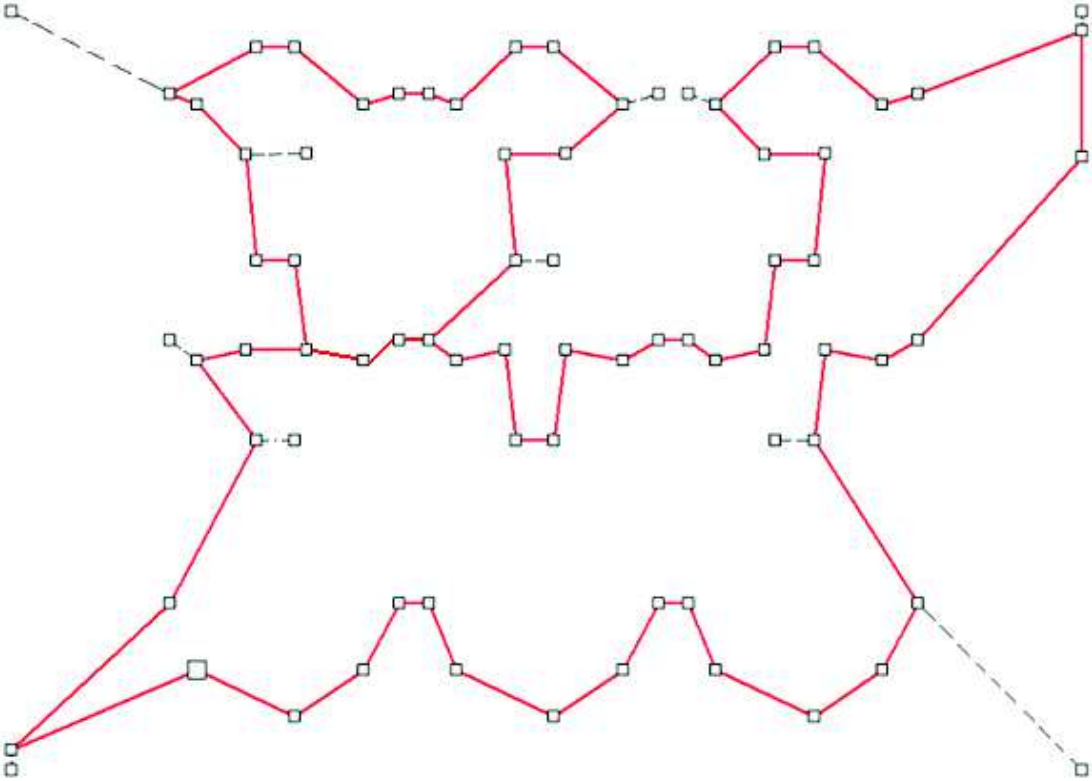


Figura 1 - Solución factible del problema 2NCSP

A manera de ejemplo, la Figura 1 presenta una solución factible del problema 2NCSP calculado sobre la instancia pr76 de la TSPLIB [4]. Las líneas sólidas representan la estructura 2-nodo conexa, mientras que las líneas punteadas representan los enlaces directos entre los nodos no asignados a la estructura 2-nodo conexa, al nodo más cercano del mismo.

1.2.3. Complejidad del Problema 2NCSP

Proposición 1.1
El problema 2NCSP pertenece a la clase de problemas NP-Completo.

Demostración:
Realizaremos la demostración por reducción al Minimun-Weight Two-Connected Spanning Problem (MW2CSP), el cual es NP-Completo como se demuestra en [55].

Sea $G = (V, E)$ un grafo simple y c la función de costos de conexión no negativos y costos de asignación $d_{ij} = \infty$. Es trivial observar que con esas matrices de costos el problema se reduce a resolver el MW2CSP, por lo tanto el 2NCSP también es NP-Completo

1.3. Trabajos Relacionados

Por ser un nuevo problema, no existe en la literatura estudios del *2NCSP*, sin embargo, existen una diversidad de trabajos realizados sobre problemas íntimamente relacionados como el Ring Star Problem, Capacitated m-Ring Star Problem (CmRSP) y Capacitated m Two-Node Survivable Star Problem (CmTNSSP), los cuales destacamos por su similitud al nuestro.

En [5], [6] y [7] se estudian otras variantes del problema las cuales, en lugar de buscar estructuras 2-nodo conexas, procuran obtener estructuras más simples, como caminos o árboles (Tree Star Problem y Path Star Problem respectivamente).

1.3.1. Ring Star Problem

Introducción

El RSP consiste en localizar un ciclo simple a través de un subconjunto de vértices de un grafo con el objetivo de minimizar la suma de dos costos: el costo de conexión del anillo, proporcional al largo del ciclo, y el costo de asignación de los vértices que no se encuentran en el anillo al vértice más cercano del mismo.

Existe una variante del RSP denominada Median Cycle Problem, la cual difiere únicamente en que define una cota superior para el costo total de asignación [8], [9], (11).

Si bien el RSP ha sido estudiado especialmente en el ámbito de las telecomunicaciones, tiene muchas aplicaciones prácticas en problemas como: planificación de tránsito, recolección postal, diseño de rutas de distribución y localización de contenedores de basura entre otros

La Figura 2, representa una solución factible a dicho problema, donde las líneas sólidas representan el anillo, mientras que las líneas punteadas representan los enlaces directos entre los nodos no asignados al anillo, al nodo más cercano del mismo.

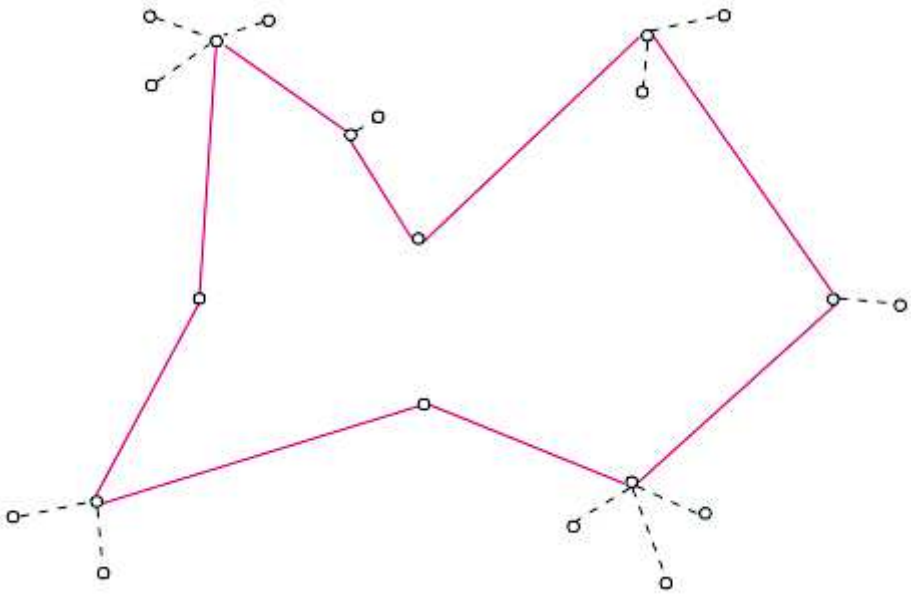


Figura 2 - Solución factible al RSP

El RSP puede ser definido formalmente de la siguiente manera: Sea $G = (V, E \cup A)$ un grafo donde $V = \{v_1, v_2, \dots, v_n\}$ es el conjunto de vértices, $E = \{(v_i, v_j) : v_i, v_j \in V, i < j\}$ es el conjunto de aristas y $A = \{[v_i, v_j] : v_i, v_j \in V\}$ es el conjunto de arcos. El vértice v_1 es denominado raíz o depósito. Cada arista (v_i, v_j) tiene asociado un costo no negativo de conexión c_{ij}, y cada arco $[v_i, v_j]$ tiene asociado un costo no negativo de asignación denominado d_{ij}.

El RSP consiste en encontrar un ciclo por un subconjunto $V' \subseteq V$, $E' \subseteq E$, incluyendo al depósito, tal que minimice la suma de los costos de conexión más los costos de asignación, es decir:

$$\text{Minimizar: } \sum_{v_i \in V \setminus V'} \min_{v_j \in V'} d_{ij} + \sum_{v_i, v_j \in E'} c_{ij}$$

Donde, los costos de conexión son los costos de las aristas del ciclo, mientras que los costos de asignación son los costos de asignar los vértices no incluidos en el ciclo, a los vértices del mismo.

Es posible observar que el Traveling Salesman Problem (TSP), es un caso especial del RSP en donde los costos de asignación son muy elevados respecto a los costos de ruteo. Es conocido que el TSP es un problema NP-Completo, por lo cual el RSP también lo es.

Aplicaciones prácticas para este tipo de modelos, particularmente en telecomunicaciones, contienen restricciones adicionales. El RSP es una versión simplificada de situaciones más realistas donde, por ejemplo, la cantidad de vértices que se pueden asignar a un vértice del anillo es limitada.

Modelo matemático del RSP

El RSP puede ser formulado como un modelo de Programación Entera de la siguiente forma: Para cada arco $[v_i, v_j] \in E$, sea x_{ij} una variable binaria igual a 1 si solo si el arco $[v_i, v_j]$ aparece en el ciclo. Para cada arista $(v_i, v_j) \in A$, sea y_{ij} una variable binaria igual a 1 si y solo si el vértice v_i es asignado al vértice v_j en el ciclo. Si un vértice está en el ciclo, este está asignado a sí mismo, es decir $y_{ii} = 1$. Además, para $S \subset V$, $E(S) = \{[v_i, v_j] \in E : v_i, v_i \in S\}$ y $\delta(S) = \{[v_i, v_j] \in E: v_i \in S, v_j \notin S\}$. Para simplificar si $S = v_i$, su notación será $\delta(i)$ en vez de $\delta(\{v_i\})$. Para $E' \subseteq E$ se define $x(E') = \sum_{[v_i,v_j]\in E'} x_{ij}$

El RSP puede formularse como [1]:

$$Minimizar \sum_{[v_i,v_j]\in E} c_{ij}\, x_{ij} + \sum_{(v_i,v_j)\in A} d_{ij} y_{ij} \qquad (1.3.1)$$

Sujeto a:

$$x\big(\delta(i)\big) = 2y_{ii} \quad \forall v_i \in V \qquad (1.3.2)$$
$$\sum_{v_j \in V} y_{ij} = 1 \quad \forall v_i \in V\backslash\{v_1\} \qquad (1.3.3)$$
$$x\big(\delta(S)\big) \geq 2 \sum_{v_j \in S} y_{ij} \quad \forall S \subset V: v_1 \notin S, v_i \in S \qquad (1.3.4)$$
$$x_{ij} \in \{0,1\} \quad \forall [v_i, v_j] \in E \qquad (1.3.5)$$
$$y_{ij} \geq 0 \quad \forall (v_i, v_j) \in A \qquad (1.3.6)$$
$$y_{11} = 1 \qquad (1.3.7)$$
$$y_{1j} = 0 \quad \forall v_j \in V\backslash\{v_1\} \qquad (1.3.8)$$
$$y_{jj}\ entero\ \forall v_j \in V\backslash\{v_1\} \qquad (1.3.9)$$

En esta formulación, la restricción (1.3.2) condiciona el grado de los vértices del ciclo, la cual asegura que el grado de un vértice v_i es 2, si solo si, pertenece al ciclo. La restricción (1.3.3) implica que un vértice v_i pertenece al ciclo ($y_{ii} = 1$), o bien, v_i está asignado a un vértice v_j que pertenece al ciclo ($v_{ii} = 0$ y $v_{ij} = 1$). La ecuación (1.3.4) exige las restricciones de conectividad, ya que significa que S debe estar conectado con su complemento por al menos dos aristas del ciclo cuando al menos un vértice $v_i \in S$ está asignado a $v_j \in S$, es decir, limitar el número de ciclos exigiendo la conectividad entre vértices con $y_{ii} = 1$ y v_1. La combinación de (1.3.2), (1.3.5), (1.3.7) y (1.3.8) garantiza que la solución va a contener al menos un ciclo incluyendo el depósito. La restricción (1.3.4) limita el número de ciclos a 1 ya que fuerza la conectividad entre vértices con $y_{ii} = 1$ y v_1.

Por último, la combinación de las restricciones (1.3.2), (1.3.3), (1.3.4) y (1.3.6) implican que cada vértice que no pertenezca al ciclo debe ser asignado a un vértice del mismo.

Condiciones de integridad sobre las variables y_{ij} $(i \neq j)$ no son necesarias, ya que en una solución óptima de la formulación sin la restricción (1.3.9), un nodo v_i siempre será asignado al nodo v_j más cercano en el ciclo, i.e. $y_{ij} = 1$. El modelo anterior impone de forma implícita que el ciclo visita al menos tres vértices incluyendo el depósito.

Algoritmos conocidos para el RSP

En esta sección, haremos una revisión de las técnicas utilizadas por diferentes autores para resolver el RSP.

El RSP fue formulado por primera vez en 1999 por M. Labbé et al. en [8] denominándolo MCP1. Años después los autores resuelven el problema mediante un algoritmo Branch & Cut [1]. La performance del algoritmo fue probada en dos clases diferentes de instancias. La primera clase incluía todas las instancias de la TSPLIB [4] con instancias de entre 100 y 150 nodos (kroa100 a krob150), mientras que la segunda clase fue generada aleatoriamente. Los autores concluyen que el algoritmo implementado fue capaz de resolver óptimamente 75 de 76 instancias de la TSPLIB y 195 de 200 de las instancias generadas aleatoriamente [8]. También señalan que las soluciones con pocos vértices en el ciclo brindaban malos resultados, mientras que las soluciones con casi todos los vértices en el ciclo, se comportan de mejor forma.

En el año 2003, J. Moreno et al. [10] proponen la metaheurística Variable Neighborhood Tabu Search (VNTS) para resolver el RSP. El algoritmo diseñado consiste en la construcción de una solución inicial con al menos tres vértices. Luego se aplica una búsqueda local mediante los movimientos estándar “Agregar”, “Quitar” e “Intercambiar” nodos hasta que no se encuentren mejoras en la solución. Por último, se aplica un procedimiento de exploración denominado “Shake” o “Perturbación”, que consiste en aplicar un número aleatorio de movimientos procurando obtener una solución fuera del óptimo local, con este nueva solución se retoman las búsquedas locales. Tanto en la búsqueda local como en el procedimiento de exploración se hace uso una lista tabú, que indica los movimientos que no pueden realizarse. Los autores ejecutaron pruebas de performance utilizando un conjunto de instancias de la TSPLIB y compararon los resultados con los obtenidos por el algoritmo Branch & Cut de M. Labbé señalado anteriormente, concluyendo que el VNTS brinda buenos resultados cuando el ciclo de la solución es pequeño, sin embargo, cuando la cantidad de vértices del ciclo es de más del 50% del total de vértices de la instancia de prueba, las soluciones ofrecen peores resultados respecto al algoritmo de comparación.

En el año 2004, M. Labbé et al. [1] diseñan un algoritmo Branch & Cut utilizado para resolver instancias de prueba de hasta 300 vértices. Los autores consideran que presentaron el primer algoritmo exacto que resuelve el RSP, evaluando los resultados en

tres diferentes clases de instancias de prueba. Entre ellas destacamos la Clase I, basada en instancias de la TSPLIB 2.1 que involucran entre 50 y 200 nodos, ya que es la que se tomó como referencia para evaluar la performance de nuestra metaheurística.

También en el año 2004, J. Reanud et al. (10) proponen resolver el problema mediante dos heurísticas diferentes. La primera denominada Multistart Greedy Add Heurisitc (MGA) conformada por una fase de construcción y una fase de mejora de la solución. En la fase de construcción se conforma el ciclo con vértices seleccionados aleatoriamente, al cual se le agrega iterativamente el vértice que genere el menor incremento de costo, hasta que no se encuentren más reducciones. Para la fase de mejora se propone aplicar, sobre la solución inicial, los procedimientos: "Agregar", "Quitar", "Intercambiar" nodos, Mejora (TSP) y Perturbación. El segundo algoritmo propuesto, es un algoritmo evolutivo que denominan Random Keys Evolutionary Algorithm (RKEA), para el cual utilizan una codificación denominada Random Key por la simplicidad que esta brinda a la hora de aplicar el operador de cruzamiento entre dos soluciones, y, porque permite fácilmente recombinar ciclos de largos diferentes. Debido a que el algoritmo MGA produce familias de buenas soluciones, éstas fueron utilizadas como la primera generación del algoritmo evolutivo con el objetivo de mejorar aún más estas soluciones. Los autores probaron la performance de ambas heurísticas sobre un subconjunto de las instancias usadas por M. Labbé de la TSPLIB [4], y las compararon con los resultados del algoritmo VNTS. Las comparaciones indican que el algoritmo obtiene soluciones más próximas al óptimo y en menor tiempo que el VNTS, e indican que si bien los tiempos del RKEA son elevados, su aplicación aumenta notoriamente la exactitud de la solución obtenida por el algoritmo MGA.

S. Dias et al. presentaron, en el año 2006, una nueva metaheurística para resolver el RSP [11]. Los autores proponen un algoritmo GRASP el cual, en su fase de construcción, crea una solución inicial a partir de un solo vértice, y se agregan nuevos vértices de forma que se genere el menor incremento posible a la solución. La función de evaluación considera el incremento de agregar el vértice en el anillo más el costo promedio de asignación del resto de los vértices al nuevo vértice del anillo. La aleatoriedad de la fase de construcción está dada porque no siempre se elige el mejor candidato, sino que se selecciona aleatoriamente de una lista de candidatos denominada Restricted Candidate List (RCL). Una vez finalizada la construcción de la solución inicial, comienza la fase de búsquedas locales con los **siguientes** movimientos: "Agregar", "Quitar" y "k-opt" [12]. Por último se utiliza un algoritmo General Variable Neighborhood Search (GVNS) con la finalidad de explorar el espacio de soluciones generando una cantidad aleatoria de movimientos aleatorios, para obtener una nueva solución inicial. Los autores probaron la performance de esta heurística sobre un subconjunto de instancias (C2) de la TSPLIB y sobre un conjunto de instancias (C1) obtenidas de [1]. Los resultados muestran que su algoritmo mejoró significativamente los resultados obtenidos por el algoritmo VNTS encontrando mejores soluciones para 43 de 210 instancias de C1 y 13 de 33 en C2.

En el año 2008 A. Liefooghe et al. propusieron el RSP como un problema de optimización bi-objetivo [13]. Para resolver el problema, se propusieron tres metaheurísticas, una búsqueda local basada en población y dos algoritmos evolutivos. Dichos algoritmos son variaciones de IBMOLS [14], IBEA [15] y NSGA-II [16] respectivamente. Los autores probaron la performance de los diversos algoritmos en un conjunto de instancias de la TSPLIB, que involucraban entre 50 y 300 nodos, concluyendo que los la nueva versión iterativa del IBMOLS retornó, en general, resultados significativamente mejores que los obtenidos mediante los algoritmos evolutivos. Sin embargo, para instancias con gran cantidad de nodos (pr264 y pr299), los algoritmos evolutivos presentaron mejores resultados.

En 2010 S. Kedad-Sidhoum et al. proponen un nuevo algoritmo Branch & Cut para resolver el RSP [17] a partir de una nueva formulación matemática del problema basada en *st-chains* (caminos simples no dirigidos entre un par específico de nodos *s* y *t*). Para probar la performance del algoritmo, se utilizó y comparó contra el mismo conjunto de instancias de la TSPLIB que las utilizadas por Labbé et al. en [1], concluyendo que el nuevo algoritmo Branch & Cut es 15 veces más rápido, siendo capaz de resolver la instancia kroA200 en 6 minutos.

En 2013, H. Calvete et al. [18] proponen un algoritmo evolutivo para resolver el RSP formulándolo como un problema de programación de dos niveles. Los autores utilizaron instancias de la TSPLIB involucrando entre 50 y 200 nodos para sus pruebas, concluyendo que el nuevo algoritmo presenta la mejor solución conocida en un 92.74% de los casos de prueba.

1.3.2. Capacitated m-Ring Star Problem

Introducción

El Capacitated m-Ring Star Problem (CmRSP) fue introducido por Baldacci et al. en el año 2007 [19]. El problema consiste en encontrar un conjunto de m anillos nodos disjuntos entre sí, a excepción de un nodo central (denominado depósito) el cual debe formar parte de cada uno de los anillos. Aquellos vértices que no formen parte de ningún anillo deben ser conectados directamente a nodos pertenecientes a algún anillo (denominados colgantes). El número total de nodos visitados o conectados a un anillo está acotado por un límite superior Q, conocido como la capacidad del anillo. El objetivo del CmRSP es minimizar los costos de conexión de los anillos más el costo de asignación de los nodos que no pertenecen a los anillos a algún nodo de ellos.

El CmRSP tiene muchas aplicaciones prácticas, y ha sido especialmente estudiado en el ámbito de las redes de telecomunicaciones, en donde problemas del estilo, asisten el proceso de diseño de redes de fibra óptica a un costo mínimo [20], [21] y [22].

El CmRSP puede ser definido formalmente de la siguiente forma: Sea un grafo $\mathrm{G} = (\mathrm{V}, \mathrm{E} \cup \mathrm{A})$, donde $\mathrm{V} = \{0, \mathrm{n}+1\} \cup \mathrm{U} \cup \mathrm{W}$ es el conjunto de nodos, $\mathrm{E} = \{\{\mathrm{i}, \mathrm{j}\}: \mathrm{i}, \mathrm{j} \in \mathrm{V}, \mathrm{i} \neq \mathrm{j}\}$ es el conjunto de aristas y A es el conjunto de arcos. El conjunto de nodos U contiene los nodos terminales, mientras que el conjunto W contiene los nodos de transición o nodos de Steiner. Los nodos 0 y $n+1$ representan el depósito, con lo cual diremos que un anillo es un camino del nodo 0 al n+1. Para cada nodo terminal $i \in U$, definimos $C_i \subseteq U \cup \mathrm{W}$ como el subconjunto de nodos al cual el nodo terminal i puede ser conectado. El conjunto A representa las posibles conexiones entre nodos terminales, es decir $A = \{(i,j): i \in U, j \in C_i\}$. Cada arista $e = \{i,j\} \in E$ tiene asociado un costo no negativo c_{ij}, denominado costo de conexión, y cada arco $a = (i,j)$ tiene asociado un costo no negativo de asignación d_{ij}. Dado un subconjunto $E' \subset E$, $V(E')$ denota el conjunto de nodos incidentes a al menos una arista en E'.

Un anillo R es un par (E', A') donde $E' \subset E$ es un camino simple entre 0 y $n+1$, y $A' \subseteq A$ son las conexiones entre los nodos de $U \backslash V(E')$ y los nodos de $V(E')$. Se dice que un nodo está asignado a un anillo si es visitado por el camino simple, o bien, está conectado a algún nodo del mismo. Un anillo es factible si el número de nodos terminales asignados a él no excede su capacidad Q. Se asume que $mQ \geq |U|$.

El costo del anillo es la suma del costo de conexión de las aristas en E' más la suma de los costos de asignación de los arcos en A'. El objetivo del CmRSP es diseñar m anillos de forma tal que cada nodo terminal esté asignado exactamente a uno de ellos y tal que la suma de los costos de los anillos sea minimizada.

La Figura 3 [23], representa una solución factible del CmRSP, donde $n = 25$, $|U| = 26$, $|W| = 4$ y $Q = 15$. En esta Figura, los puntos de transición están representados mediante

triángulos, mientras que los nodos terminales son representados con círculos. Por último, el depósito central se representa mediante dos rectángulos negros. Las líneas sólidas representan las aristas de los ciclos, mientras que las líneas punteadas representan las conexiones directas o arcos.

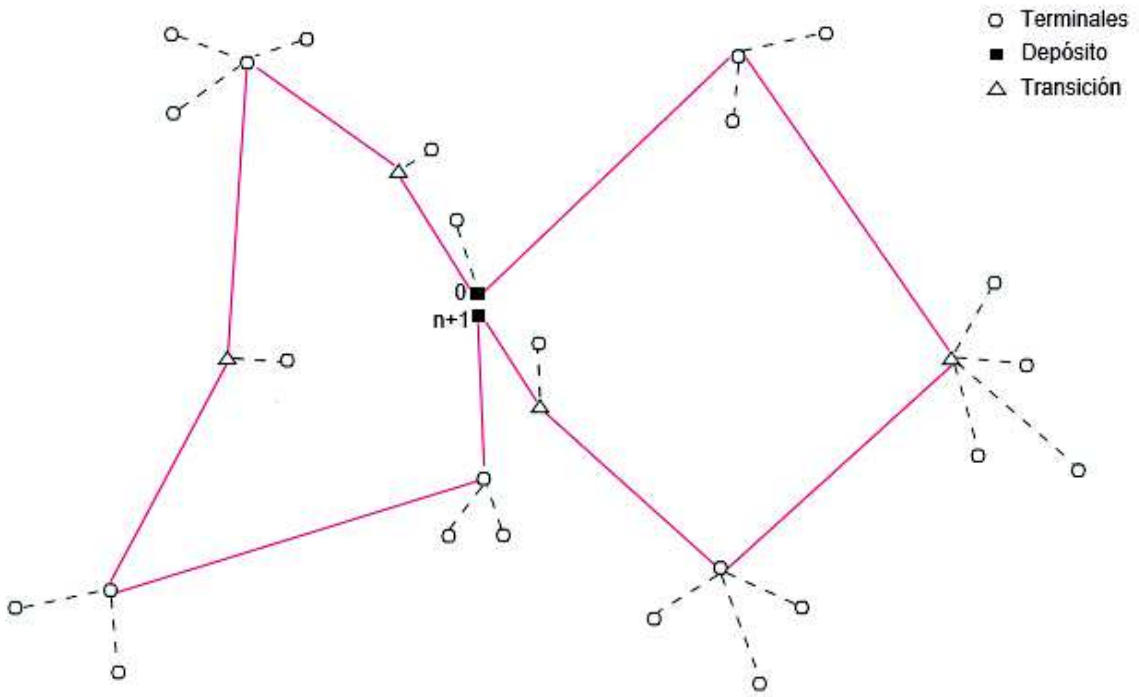


Figura 3 - Solución factible CmRSP

Modelo matemático

En esta sección presentamos uno de los dos modelos matemáticos para el CmRSP propuestos por Baldacci et al. en [19], el cual se inspira en problemas estándar de ruteo y al cual los autores denominan “two-index formulation”.

Para cada arista $e \in E$, se define una variable binaria x_e igual a 1 si solo si e pertenece a algún anillo en la solución. De ahora en más, si e conecta dos nodos i y j, haremos referencia a la misma como e o $\{i,j\}$ indistintamente. Para cada arco $(i,j) \in A$, se define la variable binaria z_{ij} de forma tal que su valor sea 1 si solo si el cliente i está asignado al nodo j. Si un cliente es visitado por algún anillo, entonces se asume que el cliente está asignado a sí mismo, es decir $z_{ii} = 1$. Por último, para cada $j \in W$, se define una variable binaria w_{ij} igual a 1 si solo si el nodo de Steiner j pertenece a un anillo.

El CmRSP puede formularse como [19]:

$$minimizar \sum_{e \in E} c_e x_e + \sum_{(i,j) \in A} d_{ij} z_{ij} \quad (1.3.10)$$

Sujeto a:

$$\sum_{e \in \delta(0)} x_e = m \quad (1.3.11)$$

$$\sum_{e \in \delta(n+1)} x_e = m \quad (1.3.12)$$

$$\sum_{e \in \delta(i)} x_e = 2z_{ii} \quad \forall i \in U \quad (1.3.13)$$

$$\sum_{e \in \delta(i)} x_e = 2w_i \quad \forall i \in W \quad (1.3.14)$$

$$\sum_{j \in V'} z_{ij} = 1 \quad \forall i \in U \quad (1.3.15)$$

$$\sum_{e \in \delta(S)} x_e \geq \frac{2}{Q} \sum_{i \in U} \sum_{j \in S} z_{ij} \quad \forall S \subseteq V' : S \neq \phi \quad (1.3.16)$$

$$x_e \in \{0,1\} \quad \forall e \in E \quad (1.3.17)$$

$$z_{ij} \in \{0,1\} \quad \forall (i,j) \in A \quad (1.3.18)$$

$$w_i \in \{0,1\} \quad \forall i \in W \quad (1.3.19)$$

Las restricciones (1.3.11) y (1.3.12) indican que m anillos deben partir del nodo 0 y terminar en el nodo $n+1$. Las restricciones (1.3.13) y (1.3.14) son restricciones de conectividad que indican que el grado de cada nodo $i \in V'$ debe ser 2 si éste pertenece a un anillo. La restricción (1.3.15) implica que un cliente $i \in U$, o bien se encuentra en un anillo ($z_{ii} = 1$), o bien está asignado a un nodo de algún anillo. Por último, restricción (1.3.16) implica que dado un subconjunto S de nodos, al menos $\sum_{i \in U} \sum_{j \in S} z_{ij} / Q$ anillos son necesarios para visitar todos los clientes asignados a los nodos en S.

Algoritmos conocidos para el CmRSP

En el año 2007, Baldacci et al [19] presentaron dos formulaciones matemáticas de Programación Lineal Entera para resolver el problema mediante algoritmos exactos Branch & Cut. La performance de los algoritmos fue probada en primera instancia con casos de prueba, para luego ser probado con algunas salvedades, en el caso real que motivó este problema: la red de fibra óptica de la ciudad Reggio Emilia en Italia. Los casos de prueba fueron tomados de las instancias del TSPLIB, adaptadas levemente para simular los nodos de Steiner. Los autores concluyen que el algoritmo Branch & Cut propuesto fue capaz de resolver óptimamente las instancias del caso real, y que el algoritmo es capaz de encontrar buenas soluciones para instancias de prueba de hasta 100 nodos.

En el año 2008 E. Hoshino et al. [24] propusieron un algoritmo para resolver de forma exacta el CmRSP aplicando la técnica conocida como Column Generation Algorithm, basado en un algoritmo Branch & Price. El algoritmo propuesto probó ser competitivo con el algoritmo Branch & Cut presentado en [19]. Ninguno de los algoritmos demostró ser dominante, existiendo instancias que eran más adecuadas para uno y otro indistintamente. Por último sugieren continuar el estudio mediante el desarrollo de un algoritmo Branch & Cut & Price que procure combinar ambas técnicas.

En [25] resuelven el problema también de forma exacta, mediante el algoritmo Branch & Cut & Price sugerido previamente. Esta nueva versión del algoritmo mejora los resultados obtenidos por Baldacci en [23], para muchos de los casos de prueba.

Luego, A. Mauttone et al. [26] presentaron un algoritmo que combina GRASP y Tabú Search para resolver el CmRSP. El algoritmo comienza construyendo iterativamente m anillos, en donde, en cada paso, se selecciona un nodo de forma aleatoria entre los k más lejanos a la solución parcial y es agregado a un ciclo, o bien, es asignado a un nodo de un ciclo, dependiendo de qué opción implique menor incremento de costo de la solución. Para la fase de búsqueda local se utilizan los movimientos clásicos: Agregar, Eliminar e Intercambio de nodos. Los autores evaluaron la performance del algoritmo utilizando un conjunto de 90 instancias propuestas en [23] donde observaron que en todos los casos se encontraron soluciones óptimas o muy próximas al óptimo. Por último proponen la posibilidad de mejorar la calidad de los resultados mediante el uso de procedimientos de memoria adaptativa como una forma natural de compartir información entre diferentes iteraciones del GRASP.

Recientemente, Naji-Azimi et al [20], proponen una heurística en la cual, luego de una fase de construcción, se aplican búsquedas locales mediante diferentes procedimientos aplicados iterativamente. Un aspecto interesante de esta metaheurística es el uso de un algoritmo de clustering para la creación de los m anillos iniciales. En la fase de mejora de la solución, el algoritmo procura encontrar nuevas soluciones mediante el uso de los procedimientos: Intercambio, Eliminación de nodo de Steiner y Perturbación. La performance de la heurística fue evaluada utilizando las instancias del TSPLIB utilizadas

por Baldacci et al. [23]. Los autores concluyen que su algoritmo es capaz de obtener soluciones óptimas en la mayoría de los casos, inclusive mejorando algunas de las mejores soluciones conocidas para el problema hasta el momento.

1.3.3. Capacitated m Two-Node Survivable Star Problem

El CmTNSSP fue introducido por G.Baya en (44), el cual es una generalización del CmRSP, en donde las m componentes 2-nodo-conexa, no tiene por qué tener topología de anillo.

Sea un grafo simple no dirigido $G = (V, E)$, donde d es un nodo distinguido de V al que denomina "depósito". El conjunto de vértices restantes $V \setminus \{d\}$ se particiona en dos conjuntos disjuntos, uno de ellos llamado conjunto de nodos terminales T, y el otro conjunto de nodos auxiliares o de Steiner.

Los nodos terminales son aquellos que deberán estar obligatoriamente presentes en la solución al problema mientras que los nodos de Steiner participarán de la misma solamente si su inclusión mejora los costos de dicha solución.

Una solución factible al CmTNSSP estará compuesta por un cierto número m de subgrafos 2-nodo conexos, que solamente tendrán en común el nodo d, de modo que si quitamos este nodo el grafo resultante quedaría dividido en m componentes conexas.

Cada componente conecta el depósito d con una cierta cantidad de nodos terminales que no podrá ser mayor a un parámetro de capacidad Q dado.

Los nodos terminales presentes en cada una de estas m componentes conexas, o bien pertenecen a una estructura 2-nodo conexa, o bien, están unidos a dicha estructura mediante una arista.

Los nodos de Steiner, si se incluyen, solamente podrán pertenecer a las estructuras 2-nodo conexas mencionadas en el párrafo anterior. Dichos nodos no podrán estar unidos a estas estructuras mediante una arista.

El grafo inicial G tiene asociadas dos matrices de costos. Una de ellas determina el costo de conectar cada par de vértices si ambos forman parte de la estructura 2-nodo conexa con redundancia (costos de conexión) y la otra determina el costo de conectar un par de vértices si uno de ellos está unido a la estructura mediante una arista (costos de asignación).

El problema consiste en obtener un subgrafo del grafo inicial, que sea de costo mínimo y construido bajo las premisas anteriores.

Hasta donde sabemos G. Baya fue el único en estudiar el problema hasta el momento, y por lo tanto, tomamos su trabajo como referencia.

1.4. Estructura del documento

El trabajo está estructurado de la siguiente manera. En el Capítulo 1 se presentó y definió formalmente el Problema 2-Node-Connected Star. Debido a que no se encontró literatura al respecto, se describió y detalló el estado del arte de dos problemas que consideramos íntimamente relacionados: el RSP y el CmRSP.

En el Capítulo 2 se realiza una breve introducción a las metaheurísticas, especialmente a la metaheurística GRASP por ser la seleccionada para resolver el problema.

En los capítulos 3 y 4 se presenta la resolución del problema mediante el uso de una metaheurística GRASP VNS descendente. En el Capítulo 3 se explica en detalla la Fase de Construcción, mientras que en el Capítulo 4 se detalla en profundidad la Fase de Búsquedas Locales.

En el capítulo 5 se explica el estudio experimental consistente en el armado de casos de prueba, ejecución de los algoritmos y evaluación de los resultados en comparación con los resultados presentados por Labbé et al. en [1].

CAPÍTULO 2

2. La metaheurística GRASP

En este Capítulo se realizará una breve introducción a las metaheurísticas, detallándose con mayor profundidad la metaheurística GRASP por ser la estrategia seleccionada para la resolución del problema abordado en esta tesis.

2.1. Introducción a las metaheurísticas

Usualmente nos encontramos con problemas complejos para los cuales encontrar soluciones óptimas, en un tiempo computacional razonable, resulta imposible. En estos casos, se suelen buscar técnicas alternativas de resolución que permitan obtener buenas soluciones de forma eficiente. Dichas técnicas son conocidas cómo heurísticas del latín heuriken, que significa “descubrir”.

Más formalmente podemos decir que una heurística es una técnica para resolver un problema determinado, que brinda soluciones cercanas al óptimo, a un tiempo computacional razonable, sin garantizar la optimalidad del mismo.

En 1986 F. Glover [27] introduce el término metaheurística para definir la búsqueda “Tabú” también introducida por el autor en el mismo artículo. El término deriva del griego meta, que significa “más allá de, o en nivel superior a”, y heuriken definido previamente.

Una metaheurística es una estrategia general utilizada para resolver una gran variedad de problemas de optimización. Dichas estrategias son guías para el diseño de heurísticas que resuelven problemas específicos de optimización [28].

Las metaheurísticas se aplican generalmente a problemas NP-Hard o NP-Completos, sin embargo, también podrían aplicarse a problemas de optimización combinatoria en los cuales se sabe que existe una solución de tiempo polinómico, pero que no es alcanzable en la práctica.

Dado que las metaheurísticas son estrategias para diseñar procedimientos heurísticos, éstas se clasifican en primer lugar, en función del tipo de procedimiento al cual refiere [29] . A continuación se detallan los tipos más conocidos.

Metaheurísticas de relajación

Refieren a procedimientos de resolución de problemas que utilizan relajaciones del problema original, es decir, modificaciones del modelo que hacen que el problema sea más sencillo de resolver.

Cuando se resuelve un problema real, uno de los principales desafíos es encontrar un modelo que permita aplicar una técnica de resolución. Si con este modelo el problema resulta difícil de resolver, se acude a modelos modificados en los que sea más eficiente y sencillo encontrar buenas soluciones.

Muchas heurísticas de relajación modifican elementos del problema para proponer la solución de estas modificaciones como solución del problema original. Las buenas relajaciones son las que simplifican el problema y hacen más eficientes los procedimientos de solución, pero cuya resolución proporciona muy buenas soluciones del problema original.

Entre las metaheurísticas de relajación se encuentran los métodos de relajación Lagrangiana y de restricciones subordinadas.

Metaheurísticas constructivas

Las heurísticas constructivas aportan soluciones del problema por medio de un procedimiento que incorpora iterativamente elementos a una estructura, inicialmente vacía, que representa la solución.

Las metaheurísticas constructivas establecen estrategias para seleccionar las componentes con las que se construye una buena solución del problema.

Entre las metaheurísticas constructivas se encuentra la estrategia “greedy” o “voraz”, que implica la elección que da mejores resultados inmediatos, sin evaluar otras posibilidades. En este contexto, la metaheurística GRASP en su primera fase, utiliza una estrategia de construcción greedy para la creación de soluciones.

Metaheurísticas de búsqueda

Las metaheurísticas de búsqueda establecen estrategias para recorrer el espacio de soluciones del problema transformando, de forma iterativa, las soluciones iniciales. En primera instancia se diseña un procedimiento que mejore la solución del problema, el cual, se aplica iterativamente mientras fuera posible obtener mejores soluciones.

Estos procedimientos también son conocidos cómo búsquedas locales, ya que la mejora se obtiene en base al análisis de soluciones similares, denominadas soluciones vecinas. Estrictamente hablando, una búsqueda local es la que basa su estrategia en el estudio de soluciones del vecindario o entorno de la solución.

El principal inconveniente de estas búsquedas es que se quedan atrapadas en óptimos locales. Por ello, se diseñaron las siguientes pautas para escapar de los óptimos locales: volver a iniciar la búsqueda desde otra solución de partida, modificar la estructura de entornos que se está aplicando y permitir movimientos o transformaciones de la solución de búsqueda aunque no conduzcan a una mejora de la solución.

Algunas metaheurísticas de búsqueda conocidas son GRASP (en su segunda fase), Tabú Search y Simulated Annealing.

Metaheurísticas evolutivas

Las metaheurísticas evolutivas establecen estrategias de optimización y búsqueda de soluciones basadas en los postulados de la evolución biológica. El proceso inicia con una población de individuos (que representan soluciones al problema) los cuales se mezclan, mutan, y compiten entre sí, de tal manera que los más aptos son capaces de prevalecer a lo largo del tiempo, evolucionando hacia mejores soluciones en cada nueva generación.

Entre las metaheurísticas evolutivas se encuentran los algoritmos genéticos, meméticos y culturales.

Metaheurísticas de aprendizaje

Las metaheurísticas de aprendizaje establecen estrategias que permiten mejorar soluciones en función del aprendizaje obtenido por las decisiones tomadas al construir soluciones anteriores. Este tipo de metaheurística hace uso de una función de evaluación de soluciones que permite definir qué tan buena es una solución, y en base a esto se define si las decisiones tomadas para construir dicha solución fueron acertadas o no.

Entre las metaheurísticas de aprendizaje se encuentran las redes neuronales y colonias de hormigas.

2.2. Elección de la metaheurística GRASP

Para la resolución del 2NCSP decidimos utilizar la metaheurística GRASP, debido a los buenos resultados que este tipo de estrategia obtiene en diversos problemas de optimización combinatoria con aplicaciones al diseño topológico de redes [30], [31], [32].

Además de la eficiencia del algoritmo, GRASP es una metaheurística maleable que permite realizar modificaciones de forma sencilla. Generalmente, dichas adaptaciones son requeridas luego del análisis empírico de su comportamiento.

2.3. La metaheurística GRASP

La metaheurística Greedy Random Adaptative Search Procedure (conocida por su sigla en inglés GRASP), fue introducida en 1995 por T.A. Feo y M.G.C. Resende [33] para resolver problemas de optimización combinatoria. Resumidamente GRASP procura obtener soluciones de buena calidad que son procesadas posteriormente para conseguir otras aún mejores. A continuación se detalla el significado de cada uno de sus componentes.

Greedy

Un algoritmo Greedy o Voraz indica que para alcanzar un objetivo en un procedimiento iterativo, en cada iteración se debe tomar la decisión que consiga el mejor resultado inmediato. El pseudocódigo de un procedimiento Greedy se presenta en el Algoritmo 1.

Algoritmo 1 – Secuencia de ejecución del Procedimiento Greedy

```
1: procedure greedy
2: input pool
3: sol ← ∅
4: Evaluar los costos incrementales c_i para todos los elementos del pool
5: repeat
6:     elemento ← Seleccionar el elemento del pool con menor costo incremental
7:     sol ← sol + elemento
8:     pool ← pool – elemento
9:   Reevaluar los costos incrementales c_i del pool
10: until pool = ∅
```

Randomized

Un algoritmo es aleatorio si su comportamiento depende de los valores producidos por un generador de números aleatorios. En el caso de GRASP, el componente aleatorio es esencial para explorar todo el espacio de soluciones posibles, evitando quedar atrapado en un óptimo local.

Los algoritmos Greedy Randomized se basan en el mismo principio que los algoritmos Greedy, con la diferencia que hacen uso de la aleatoriedad para obtener diferentes soluciones en ejecuciones distintas del procedimiento. El Algoritmo 2 presenta un pseudocódigo de un procedimiento Greedy Randomized.

Algoritmo 2 – Secuencia de ejecución para el Procedimiento Greedy Randomized

```
1: procedure greedyRandomized
2: input pool
3: sol ← ∅
4: Evaluar los costos incrementales c_i para todos los elementos del pool
5: repeat
6:     lista ← Construir una lista de elementos con menor costo incremental
7:     elemento ← Seleccionar el elemento de la lista de forma aleatoria
8:     sol ← sol + elemento
9:     pool ← pool – elemento
10:    Reevaluar los costos incrementales c_i del pool
11: until pool = ∅
```

Adaptative

Un algoritmo es adaptable (Adaptative) si es capaz es incorporar cambios dinámicos en el transcurso de la construcción de la solución.

GRASP

De acuerdo a las clasificaciones realizadas en la sección 2.1 podemos decir que GRASP es una metaheurística de Construcción y Búsqueda.

GRASP es un procedimiento iterativo en donde cada iteración consiste de dos fases, una primera fase de construcción de una solución factible y una segunda fase de mejora de la misma. El Algoritmo 3 presenta el pseudocódigo del procedimiento GRASP, el cual recibe dos parámetros: la cantidad de iteraciones a realizar *MaxIter* y la instancia a optimizar *Instancia*.

Algoritmo 3 – Secuencia de ejecución para el algoritmo GRASP

```
1: procedure grasp
2: input instancia, maxIter
3: mejorSolucion ← ∅
4: for (i = 1 to maxIter)
5:     solucionFactible ← construirSolucionGreedyRand(instancia)
6:     solucionLS ← busquedaLocal(solucionFactible)
7:     If costo(solucionLS) < costo(mejorSolucion)
8:         mejorSolucion ← solucionLS
9:     end if
10: end for
11: return mejorSolucion
```

Fase de Construcción.

La fase de construcción se inicia con una solución vacía y se van agregando elementos tomados aleatoriamente de una lista de candidatos posibles (denominada lista restringida de candidatos o LRC).

La LRC es la principal característica de GRASP y lo que lo diferencia de un algoritmo puramente Greedy. Si bien los candidatos de la LRC son aquellos que aseguran los mejores resultados, no necesariamente se elige el mejor, sino que se selecciona uno de la lista al azar, de esta forma, el procedimiento asegura que ante cada ejecución se construyan soluciones iniciales diferentes.

Existen diferentes métodos de seleccionar los elementos de la LRC, el más común de ellos es basado en cardinalidad. Dicho método consiste en seleccionar los k candidatos que generan soluciones de menor costo incremental entre todos los posibles.

Otro criterio de seleccionar los elementos de la lista es basado en valores. Este criterio selecciona los mejores elementos dentro de un rango. Sea c^{min} y c^{max} el menor y mayor costo incremental para el conjunto de elementos considerados, la lista de candidatos se construye de la siguiente forma:

$$RCL = \{e_i : c_i \leq c^{min} + \alpha(c^{max} - c^{min})\} \; con \; \alpha \in [0, 1]$$

De esta forma, cuando $\alpha = 0$ estamos ante un procedimiento Greedy, dado que el elemento a incluir será el que genere la solución incremental de menor costo.

El Algoritmo 4 detalla el pseudocódigo del procedimiento de construcción.

Algoritmo 4 – Secuencia de ejecución de la fase de construcción

```
1: procedure construirSolucionAleatoriaGreedy
2: input instancia
3: solucion ← ⌀
4: Evaluar costos incrementales c_i de todos los candidatos
5: Repeat
6:     RCL ← construirRCL(instancia)
7:     elemento ← seleccionAleatoria(RCL)
8:     solucion ← solucion + elemento
9:     Reevaluar costos incrementales c_i de todos los candidatos
10: Until solucionCompleta(solucion)
11: return solucion
```

Fase de Búsqueda Local.

Una vez finalizada la fase de construcción, se procede a una segunda fase de búsqueda local que pretende mejorar la solución factible creada. El algoritmo 5 presenta el seudocódigo de la fase de búsquedas locales en un esquema descendente (VND [34]).

Algoritmo 5 – Secuencia de ejecución de la fase de búsqueda local

```
1: procedure busquedaLocal
2: input instancia
3: repeat
4:     instancia _1 = busquedaLocal_1(instancia)
5:     if costo(instancia_1) < costo(instancia)
6:         instancia ← instancia _1
7:         Volver a busquedaLocal_1
8:     end if
9:     instancia _2 = busquedaLocal_2(instancia)
10:    if costo(instancia _2) < costo(instancia)
11:        instancia ← instancia _2
12:        Volver a busquedaLocal_1
13:    end if
14:    …….
15:    instancia _n = busquedaLocal_n(instancia)
16:    if costo(instancia _n) < costo(instancia)
17:        instancia ← instancia _n
18:        Volver a busquedaLocal_1
19:    end if
20: until noHayMejoras()
21: return instancia
```

Este esquema será detallado en profundidad en el Capítulo 4, el cual consiste, de forma simplificada, en aplicar una serie de búsquedas locales que permitan mejorar la solución. Cada vez que una búsqueda local logra el objetivo de mejorar la solución, el procedimiento vuelve a iniciar desde la primera búsqueda local de la secuencia, hasta que no se obtengan mejoras.

CAPÍTULO 3

3. GRASP para el 2NCSP – Fase de Construcción

En este Capítulo, se explicará en profundidad el procedimiento diseñado para la construcción de soluciones factibles iniciales al problema 2NCSP. Previamente detallaremos algunos algoritmos necesarios tanto para la fase de construcción como para la fase de búsquedas locales.

3.1. Algoritmos generales

Trataremos en esta sección algunos algoritmos utilizados tanto en la fase de construcción como en la fase de búsqueda local. Dichos algoritmos nos permiten identificar el camino de menor costo entre dos nodos, identificar si dos caminos son nodo disjuntos, establecer si un grafo tiene más de una componente conexa, etc.

3.1.1. Algoritmo de Dijkstra

Dijkstra fue diseñado en 1959 por Edsger Dikjstra para hallar el camino más corto desde un vértice origen al resto de los vértices de un grafo cuyas aristas tienen un costo no negativo. Usualmente se utiliza también para encontrar el camino de menor costo entre dos vértices específicos.

El algoritmo recibe como parámetros el grafo $G = (V, E)$ y un vértice origen a. Cada arista del grafo entre dos vértices i, j tiene asociado un costo no negativo $c(i, j)$. Si no existe la arista se define el costo $c(i, j) = \infty$.

Para lograr su objetivo, el algoritmo de Dijkstra mantiene un conjunto S de vértices visitados, para los cuales ya se conoce el camino de menor costo. El conjunto S se va ampliando en cada iteración hasta que todos los vértices del grafo son visitados.

Para determinar si un vértice ha sido visitado se lo etiqueta con la distancia del camino mínimo $d(z)$, cuyo valor inicial es ∞. También es posible almacenar en un vector el vértice anterior en el camino más corto encontrado, lo que permite reconstruir el camino mínimo fácilmente.

El Algoritmo 6 presenta el pseudocódigo del algoritmo de Dijkstra clásico, mientras que el Algoritmo 7 es una versión modificada de Dijkstra que permite calcular caminos mínimos, evitando pasar por los nodos recibidos en el parámetro *listaTabu*.

Algoritmo 6 – Secuencia de ejecución del algoritmo de Dijkstra

```
 1: procedure dijkstra
 2: input G, nodoOrigen
 3: begin
 4:     distancias[n] ←∞          // Vector con las distancias al nodo origen
 5:     visitados[n] ← false      // Vector con los nodos visitados
 6:     for each w = adyacentes(nodoOrigen)
 7:         distancias[w] ← costo(nodoOrigen, w)
 8:     end if
 9:     distancias [nodoOrigen] ← 0
10:    visitados[nodoOrigen] ←true
11:    while quedanNodosPorVisitar(visitados)
12:         vertice ← verticeMenorDistanciaNoVisitado(distancias, visitados)
13:         visitados[vertice] ← true
14:         for each w ∈ adyacentesNoVisitados(G, vertice)
15:             if distancias[w] > distancias[vertice]+costo(vertice , w)
16:                 distancias[w] ← distancias[vertice]+costo(vertice, w)
17:             end if
18:         end for
19:     end while
20: end begin
```

Algoritmo 7 – Secuencia de ejecución del algoritmo de Dijkstra Modificado

```
 1: procedure dijkstra
 2: input G, nodoOrigen, listaTabu
 3: begin
 4:     distancias[n] ←∞          // Vector con las distancias al nodo origen
 5:     visitados[n] ← false      // Vector con los nodos visitados
 6:     for each w = adyacentes(nodoOrigen)
 7:         distancias[w] ← costo(nodoOrigen, w)
 8:     end if
 9:     distancias [nodoOrigen] ← 0
10:    visitados[nodoOrigen] ←true
11:    while quedanNodosPorVisitar(visitados)
12:         vertice ← verticeMenorDistanciaNoVisitado(distancias, visitados)
13:         visitados[vertice] ← true
14:         if vertice ∉ listaTabu
15:             for each w ∈ adyacentesNoVisitados(G, vertice)
16:                 if distancias[w] > distancias[vertice]+costo(vertice , w)
17:                     distancias[w] ← distancias[vertice]+costo(vertice, w)
18:                 end if
19:             end for
20:         end if
21:     end while
20: end begin
```

3.1.2. Algoritmo Depth-First Search

El algoritmo DFS es de gran utilidad para un sin fin de problemas, como ser, detectar si un grafo es conexo, detectar ciclos en el grafo, descubrir la cantidad de componentes conexas, encontrar puntos de articulación, etc.

DFS explora sistemáticamente los vértices de un grafo a partir de un nodo dado, avanzando en profundidad tanto como le sea posible. Cuando ya no quedan más nodos que visitar en esa rama, regresa al nodo anterior y que repite el mismo proceso con cada uno de los hermanos del último nodo procesado.

Existen dos formas de hacer un recorrido en profundidad: usando una pila, o de manera recursiva. Los algoritmos 7 y 8 presentan el pseudocódigo del algoritmo DFS.

Algoritmo 7 – Secuencia de ejecución del algoritmo de DFS usando una pila

```
 1: procedure dfs
 2: input G, nodoOrigen
 3: begin
 4:     pila ← ∅
 5:     visitados[n] ← false
 6:     pila ← push(nodoOrigen)
 7:     Visitados[nodoOrigen] ← true
 8:    while quedanNodos(pila)
 9:         vertice ← pop(pila)
10:         for each w ∈ adyacentesNoVisitados(G, vertice)
11:             visitados [w] ← true
12:             pila ← push(w)
13:         end for
14:     end while
15: end begin
```

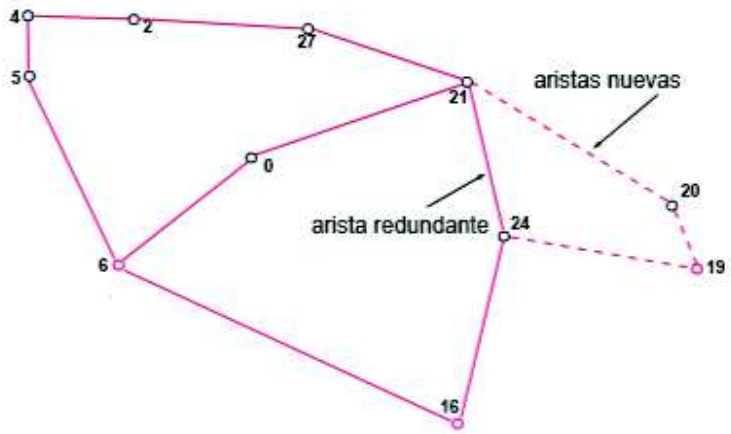
Algoritmo 8 – Secuencia de ejecución del algoritmo de DFS usando recursividad

```
1: procedure dfs
2: input G, nodoOrigen, visitados
3: begin
4:     visitados[nodoOrigen] ← true
5:     for each w ∈ adyacentesNoVisitados(G, nodoOrigen)
6:         DFS(G, w, visitados)
7:     end for
8: end begin
```

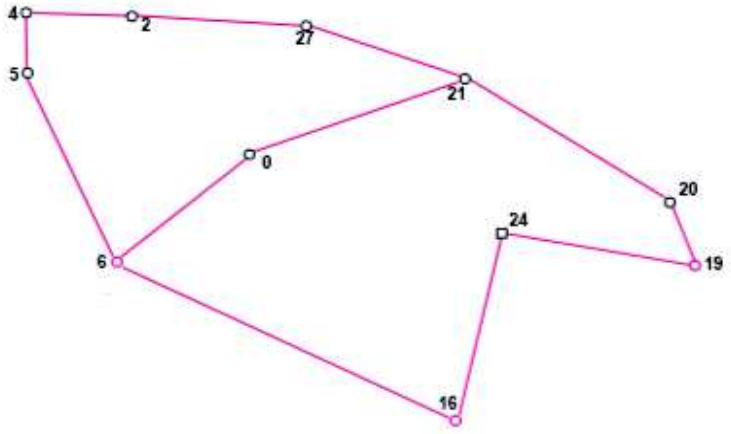
3.1.3. Algoritmo para eliminar aristas redundantes

Debido a la forma de construir la componente de 2-nodo conexa de nuestra solución para el 2NCSP, es altamente probable que la misma contenga aristas cuya eliminación no viole el requisito de 2-nodo-conectividad e incremente el costo de la solución.

En la Figura 4 se puede apreciar que al agregar el nodo 19 a la componente 2-nodo conexa, se generan los caminos (19, 24) y (19, 20, 21) (representados con líneas punteadas) haciendo que la arista (21,24) sea redundante y agregue costo a la solución.



(a) Grafo inicial



(b) Grafo sin aristas redundantes

Figura 4: Eliminación de aristas redundantes

Algoritmo 9 – Secuencia del algoritmo de eliminación de aristas redundantes

```
1:  procedure eliminarAristasRedundates
2:  input G
3:  begin
4:      for each w ∈ nodos(G)
5:          if δ(w) ≥ 3
6:              for each v ∈ nodos(G)
7:                  if δ(v) ≥ 3
8:                      eliminarArista(G, v, w)
9:                  end if
10:             end for
11:         end if
12:     end for
13:     return G
13: end begin
```

3.2. Construcción de soluciones factibles para el 2NCSP

En esta sección se describe el algoritmo para la construcción de soluciones factibles iniciales de buena calidad para el GRASP diseñado.

Una característica del algoritmo es el uso del parámetro $minNodos \in \mathbb{Z}^+$ que indica la cantidad mínima de nodos que debe tener la componente 2-nodo conexa de la solución inicial.

Algoritmo de Construcción para el 2NCSP

- **Paso 1.** El primer paso consiste en la selección de tres nodos para la creación de una componente 2-nodo conexa inicial (que en principio será un ciclo). Para la selección de dichos nodos se utilizan dos estrategias: seleccionar los tres nodos de forma aleatoria, o, seleccionar los tres nodos tal que su área sea la mayor posible.

  Para presentar ilustrativamente el funcionamiento del algoritmo se utilizará el grafo de la Figura 5, en el cual se presentan diferenciados los nodos seleccionados en el Paso 1 (nodos 6, 16 y 21).

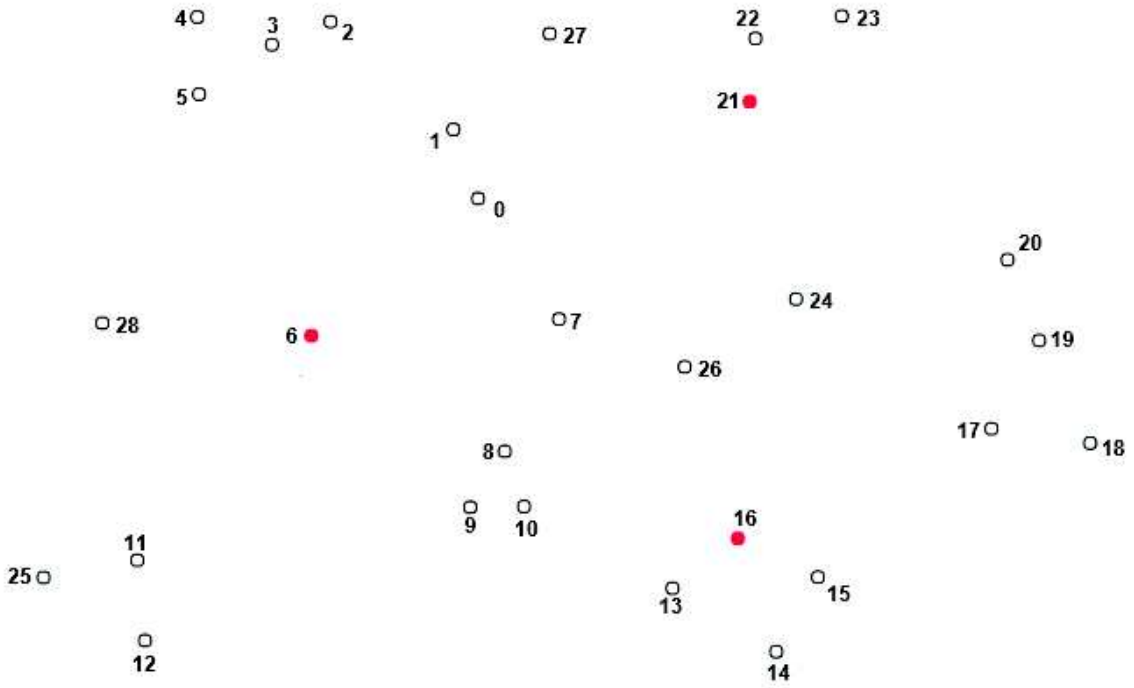


Figura 5: Resultado ejecución Paso 1

- **Paso 2.** Sean i, j, k los nodos seleccionados en el paso 1, se procede a conectar i con j, i con k y j con k, mediante caminos nodos disjuntos de costo mínimo, formando así la estructura 2-nodo conexa inicial. Para calcular el camino de costo mínimo se utiliza el algoritmo de Dijkstra explicado en la sección 3.1.1, con una leve modificación, el nuevo algoritmo utiliza una lista de nodos que no pueden ser considerados en los caminos a computar. De esta forma es posible calcular un camino mínimo, sin utilizar los nodos de los caminos ya calculados.

  La Figura 6 presenta el resultado del procedimiento luego de finalizado el Paso 2. Se puede apreciar los caminos mínimos encontrados, (6,0,21), (6,16) y (16,24,21), los cuales son nodo disjuntos y forman una estructura 2-nodo conexa.

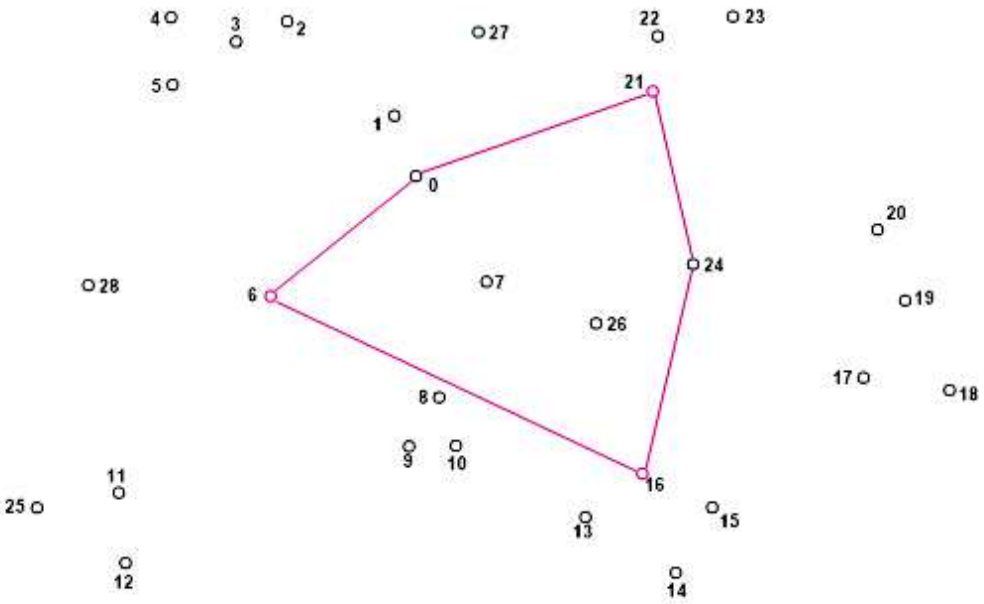


Figura 6: Resultado ejecución Paso 2

Definición 4.2.1: *H*-path
Dado un grafo H decimos que el camino P es un H-path de H si P es no trivial y se conecta a H exactamente en sus extremos. Cuando H-path es de una sola arista, dicha arista nunca es arista de H.

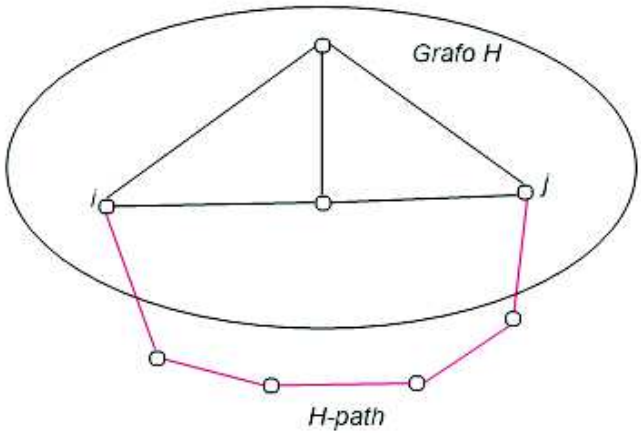


Figura 7: Ejemplo H-path

En la Figura 7 se observa un grafo H y el H-path que se conecta al Grafo H en los extremos i y j.

- **Paso 3.** Se selecciona aleatoriamente un nodo que no haya sido seleccionado previamente y se lo agrega a la estructura 2-nodo conexa formando un *H*-path. Este paso se repite mientras que la estructura 2-nodo conexa contenga a lo más la cantidad de nodos indicada por el parámetro $\max\_Nodos$.

  Para generar el H-path, se calcula el camino de costo mínimo desde el nodo seleccionado a todos los nodos de la estructura 2-nodo conexa construida hasta el momento y se lo agrega a la solución. Luego se calcula un segundo camino de costo mínimo desde el nodo seleccionado a un nodo de la estructura 2-nodo conexa, evitando cualquier nodo del camino calculado anteriormente incluyendo el que forma parte de la estructura 2-nodo conexa, y se lo agrega también a la solución.

  Para ejemplificar este tercer paso, supongamos que el nodo seleccionado aleatoriamente es el nodo 19. El algoritmo calcula todos los caminos mínimos entre los pares de nodos (19,0), (19,6), (19,16), (19,21) y (19,24), seleccionando el de menor costo, en este caso el camino (19,24) conformado por una única arista como se presenta en la Figura 8.

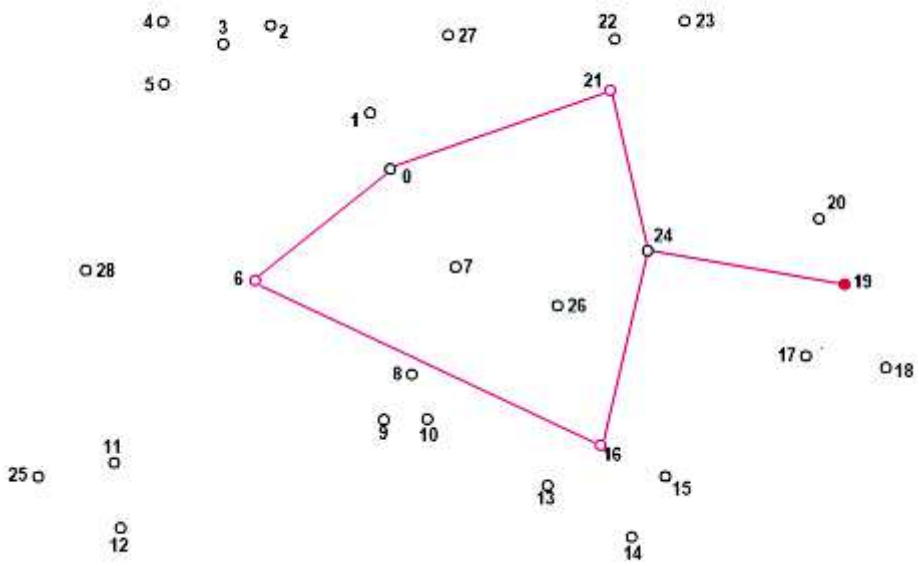


Figura 8: Construcción H-Path 1

Luego se utiliza el algoritmo de Dijkstra modificado para calcular los caminos mínimos entre los pares de nodos (19,0), (19,6), (19,16) y (19,21) (obsérvese que el

nodo 24 no es considerado por ser el seleccionado previamente). En esta nueva ejecución del algoritmo se hace uso de la lista de nodos prohibidos, integrada por los nodos que conforman el camino calculado anteriormente. Como resultado se obtiene el camino (19,20,21) como se presenta en Figura 9.

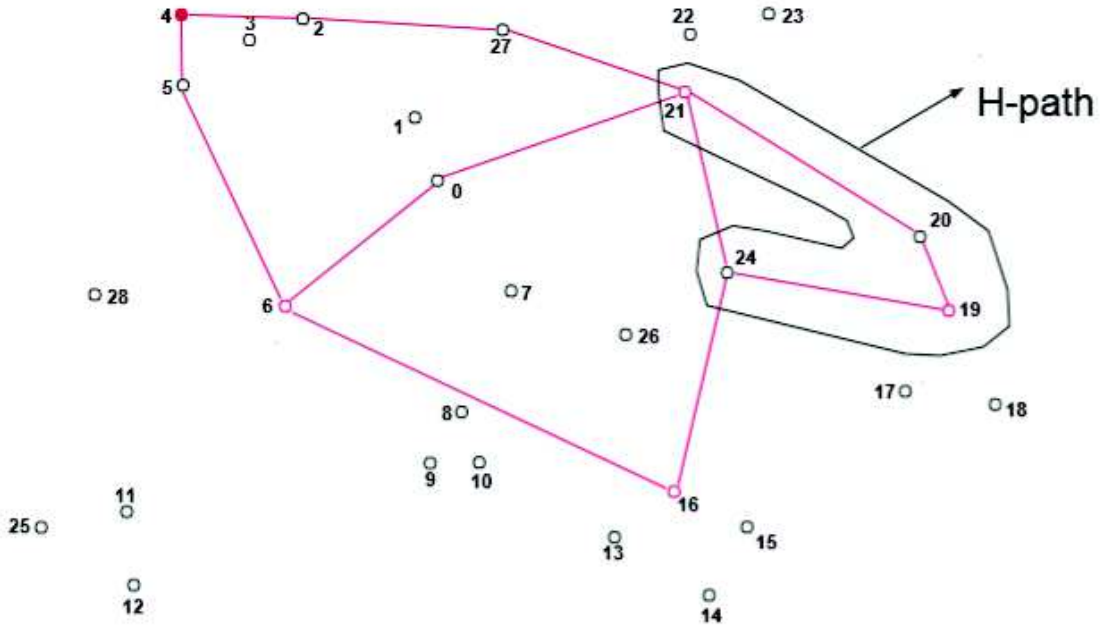


Figura 9: Construcción H-Path 2

De esta forma se construyó un H-Path que se conecta a la estructura 2-nodo conexa en los extremos 24 y 21.

- **Paso 4.** Una vez que la estructura 2-nodo conexa contiene como mínimo la cantidad de nodos especificada por parámetro, se asigna el resto de los nodos no seleccionados, al nodo más cercano en la componente 2-nodo-conexa. En caso de que alguno de los nodos no pueda ser asignado por no existir una arista entre éste y cualquier nodo de la de la componente 2-nodo-conexa, la solución es no factible, volviendo a ejecutarse el Paso 1, en busca de una nueva solución.

  En la Figura 10 se representa con líneas punteadas la asignación de los nodos que no forman parte de la solución 2-nodo conexa, al nodo más cercando de la misma.

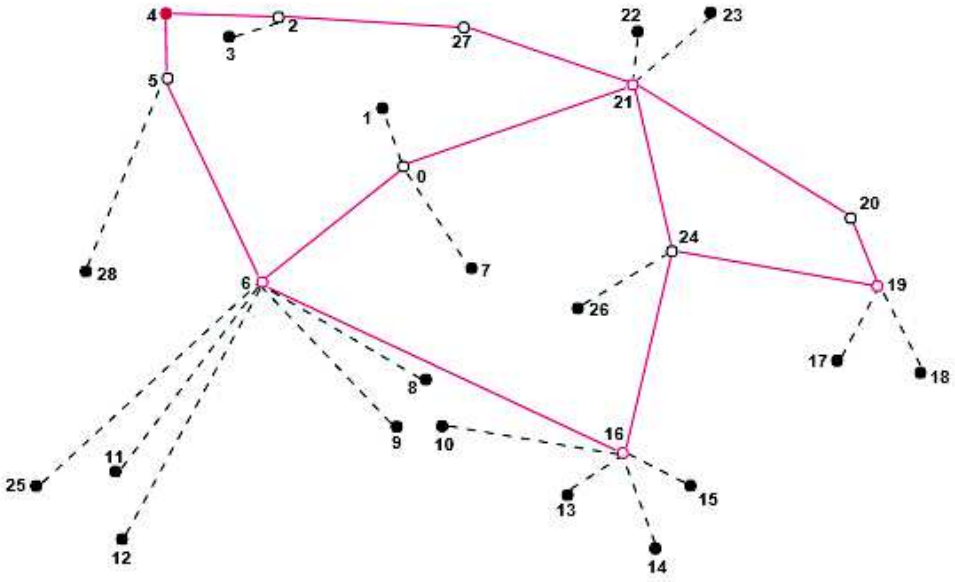


Figura 10: Asignación de nodos colgantes

- **Paso 5.** Por último, se analiza la estructura 2-nodo conexa en búsqueda de aristas redundantes, es decir, aquellas aristas que al eliminarlas no se viola el requisito de 2-conectividad, y cuya eliminación reduce el costo de la solución.

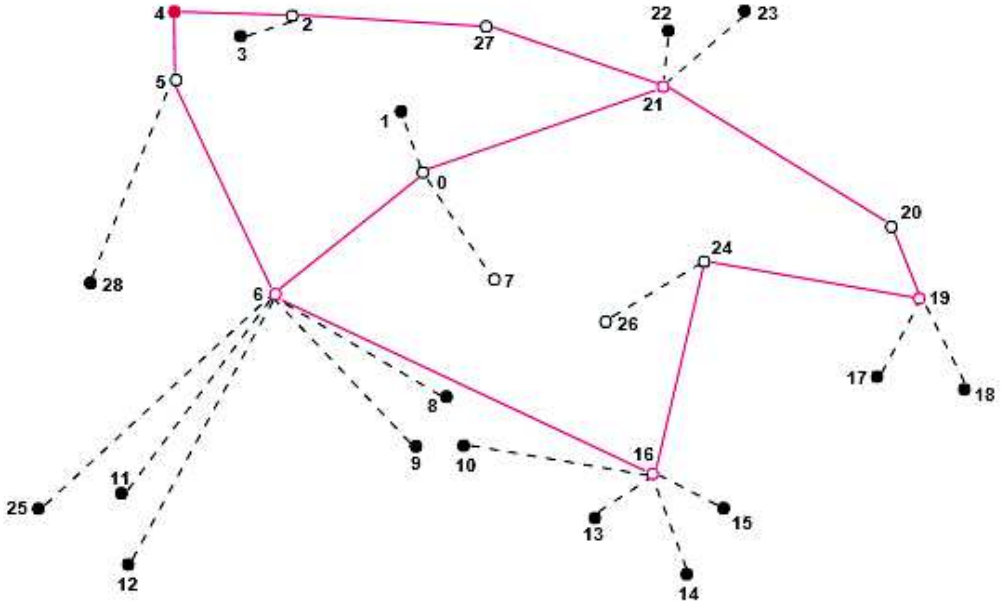


Figura 11: Eliminación de aristas redundantes

En la Figura 11 se presenta la solución factible inicial luego de haber eliminado la arista redundante (21, 24).

El Algoritmo 10 presenta el pseudocódigo detallado para la generación de soluciones factibles iniciales, correspondiente con la fase de construcción del algoritmo GRASP diseñado.

Algoritmo 10 – Secuencia de ejecución del algoritmo de generación de soluciones iniciales.

```
1:  procedure Construir_Solucion_Aleatoria_Greedy
2:  input G, maxNodos
3:  begin
4:     GSol ← ∅
5:     compConexa ← ∅
6:     sinAsignar ← obtenerNodos(G)
7:    compConexa ← obtenerCicloIncial(G)
8:    GSol ← compConexa
9:    sinAsignar ← quitarNodos(sinAsignar, compConexa)

10:    while obtenerCantidadNodos(compConexa) < maxNodos
11:       construirRCL(sinAsignar)
12:       nodoSel ← seleccionAleatoria(RCL)
13:       path_1 ← Dijkstra(nodoSel, G, compConexa)
14:       path_2 ← Dijkstra(nodoSel, G, compConexa, obtenerNodos (path_1))
15:      GSol ← agregarCamino(GSol, path_1)
16:      GSol ← agregarCamino(GSol, path_2)
17:      compConexa ← agregarCamino(compConexa, path_1)
18:      compConexa ← agregarCamino(compConexa, path_2)
19:      sinAsignar ← quitarNodos(sinAsignar, path_1)
20:      sinAsignar ← quitarNodos(sinAsignar, path_2)
21:    end while

22:    while obtenerCantidadNodos(sinAsignar) > 0
23:      nodoSel ← seleccionarNodo (sinAsignar)
24:      nodoAux ← obtenerAdyacenteMasCercano(nodoSel, compConexa)
25:      sinAsignar ← quitarNodos(sinAsignar, nodoSel)
26:      GSol ← agregarArista(GSol, nodoSel, nodoAux)
27:    end while

28:    GSol ← eliminarAristasRedundantes (GSol)
29:    return GSol
30:  end begin
```

El procedimiento recibe como parámetros el grafo original $G = (V, E)$ y la cantidad mínima de nodos que deberá tener la componente 2-nodo conexa de la solución inicial. Cada arista $e \in E$ tiene asociada dos costos no negativos: el costo de conexión c_e y el costo de asignación d_e.

El algoritmo comienza con una solución vacía G_{sol} (línea 4), la cual se irá completando hasta obtener una solución factible. Se utiliza también un grafo auxiliar *compConexa*, que contiene la componente 2-nodo conexa de nuestra solución y una lista de nodos que aún no han sido asignados a la solución, la cual se inicializa con todos los nodos del grafo original.

En la línea 7 se ejecuta el procedimiento que genera un grafo con topología de anillo el cual será la base de la componente 2-nodo conexa de la solución (el Algoritmo 10.1 detalla el pseudocódigo del procedimiento).

En las líneas 10 a 21, se incorporan nodos seleccionados de forma aleatoria (línea 12) a la componente 2-nodo conexa mediante H-paths. Para esto se utiliza el algoritmo de Dijkstra y Dijkstra modificado para obtener dos caminos nodos disjuntos desde la componente al nodo seleccionado (líneas 13 y 14). Luego se procede a agregar los caminos encontrados a la solución y a actualizar la estructura auxiliar *compConexa.* Por último se actualiza la lista de nodos no asignados hasta el momento.

Una vez creada la componente 2-nodo conexa con la cantidad de nodos deseada, se asignan el resto de los nodos que no forman parte de dicha componente, al nodo más cercano de la misma (líneas 22 a 27). La asignación es mediante un enlace directo, por lo tanto, de no existir una arista que una alguno de estos nodos a al menos uno de la componente 2-nodo conexa, la solución no es factible y se descarta. Las condiciones de factibilidad no fueron detalladas en el pseudocódigo para simplificarlo.

Por último se eliminan las aristas redundantes que puede haber generado la construcción de la solución, haciendo uso del algoritmo detallado en la sección 3.1.3.

A continuación describimos brevemente las funciones auxiliares utilizadas:

- ***obtenerNodos(param*):** retorna una lista con todos los nodos del grafo o camino recibido como parámetro.

- ***quitarNodos(listaNodos, param):*** elimina de la *listaNodos* los nodos del grafo o camino recibido en el parámetro *param*.

- ***agregarCamino(G, path):*** agrega el camino al grafo *G*.

- ***obtenerCantidadNodos(listaNodos):*** retorna la cantidad de nodos en la lista *listaNodos*.

- ***seleccionAleatoria(listaNodos):*** retorna un nodo seleccionado aleatoriamente de la lista *listaNodos.*

- ***seleccionarNodo(listaNodos):*** retorna el primer nodo de la lista de nodos *listaNodos.*

- ***obtenerNodoAdyacenteMasCercano(nodo, G):*** retorna el nodo adyacente al *nodo* recibido como parámetro, que pertenezca al grafo G y cuyo costo de asignación sea el menor posible.

Algoritmo 10.1 – Secuencia de ejecución del algoritmo construcción del ciclo inicial

```
1:  procedure obtenerCicloIncial
2:  input G
3:  begin
4:    ciclo ← ∅

5:    If selectRandomBoolean = 1
6:        [i,j,k] ← seleccionarNodosAleatoriamente(3, G)
7:    else
8:        [i,j,k] ← seleccionarNodosMayorArea(G)
9:    end if

10:   path_ij ← Dijkstra(Grafo, i, j)
11:   listaNodos ← obtenerNodos(path _ij)
12:   path _jk ← Dijkstra(Grafo, j, k, listaNodos)
13:   listaNodos ← agregarNodos(listaNodos, obtenerNodos(path _jk))
14:   path _ik ← Dijkstra(Grafo, i, k, listaNodos)
15:   ciclo ← agregarCamino(ciclo, path _ij)
16:   ciclo ← agregarCamino(ciclo, path _jk)
17:   ciclo ← agregarCamino(ciclo, path _ik)

18:   return ciclo
19: end begin
```

CAPÍTULO 4

4. GRASP para el 2NCSP – Fase de Búsquedas Locales

Una vez construida la solución factible inicial para el 2NCSP, el algoritmo GRASP procede a mejorar la solución mediante un procedimiento denominado "*búsqueda local*". En este trabajo se utiliza una búsqueda local variable, implementando vecindades clásicas, otras basadas en algoritmos evolutivos y modelos exactos de programación lineal entera, las cuales son descriptas en detalle en este capítulo.

4.1. Búsqueda local variable

La búsqueda local variable (por su sigla en inglés VNS) fue introducida por P. Hansen y N. Mladenovic en 1997, y se basa en un cambio sistemático de las estructuras de vecindades con el fin mejorar la solución [35].

Para comprender mejor la definición anterior es necesario definir el concepto "*estructura de vecindades*" o "*vecindario*".

Dado un problema de optimización que consiste en encontrar dentro de un conjunto X de soluciones factibles, aquella solución que optimiza una función *f(x)*. Un vecindario es una función $V : X \rightarrow 2^X$ que asocia cada solución $x \in X$ a un entorno de soluciones $V(x) \subset X$ que se dicen "*vecinas*" de x. El entorno de una solución $x \in X$ está constituido por todas aquellas soluciones que se pueden obtener desde x mediante la aplicación de la función V.

En la Figura 12 se presenta dos gráficas correspondientes a posibles espacios de búsqueda obtenidos a partir de diferentes vecindarios. En el Figura 12.a el procedimiento podría avanzar de la solución x_0 al óptimo local x_1, donde quedaría atrapado. Si a esta nueva solución, se le aplica otra estructura de vecindad, el procedimiento podría avanzar hacia la solución x_2, como se puede apreciar en la Figura 12.b, obteniendo así una mejor solución.

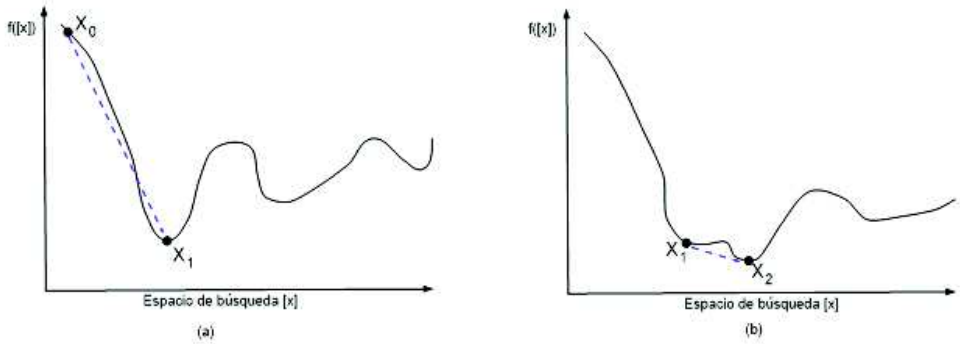


Figura 12: Espacios de búsqueda para dos estructuras de vecindad

En general las búsquedas locales aplican un movimiento o transformación a la solución, por lo tanto, utilizan una estructura de vecindad. La principal diferencia con una búsqueda local variable es que en esta se utiliza un conjunto de vecindarios, es decir, para cada solución $x \in X$ se define un conjunto finito de vecindarios $V_k(x)$, donde $1 \leq k \leq k_{max}$.

VNS se basa en tres hechos [36]:

1. Un mínimo local para un vecindario no lo es necesariamente en otro.

2. Un mínimo global es mínimo local en todos los posibles vecindarios

3. Para muchos problemas, los mínimos locales con la misma o distinta estructura de vecindades están relativamente cerca.

Esta última observación, implica que los óptimos locales proporcionan información acerca del óptimo global, por tanto, es conveniente realizar un estudio organizado en las proximidades de este óptimo local.

Los principios 1 a 3 sugieren el empleo de varias estructuras de vecindades en la fase de búsqueda local para abordar un problema de optimización. El cambio de estructura de vecindad se puede realizar de forma sistemática o al azar.

A continuación se describirá el procedimiento VNS de tipo descendente por ser el utilizado en nuestro trabajo.

4.1.1. Búsqueda local descendente

Una búsqueda local descendente consiste en encontrar iterativamente una mejor solución mediante alguna transformación o movimiento.

Dado un conjunto de vecindarios N_i $(i = 1 \,..\, i_{max})$, N_1 el primer vecindario a usar y x la solución inicial, el VNS descendente procede de la siguiente manera: si no es posible una mejora de la solución x en la estructura de vecindario actual $N_i(x)$, la vecindad se cambia de N_i a N_{i+1}; si se encuentra una mejora de la solución actual x, se vuelve a utilizar la primer estructura de vecindad $N_1(x)$. El Algoritmo 11 presenta el pseudocódigo de una búsqueda local descendente.

Algoritmo 11 – Secuencia de ejecución del algoritmo VNS Descente

```
1:  procedure VNSDescente
2:  input solucionInicial, Ni con i = 1 .. imax
3:  begin
4:    x ← solucionInicial
5:    i ← 1
6:    while i < imax
7:        Encontrar mejor vecino de x en el vecindario I, x' = Ni(x)
8:        if f(x') < f(x)
9:              x ← x'
10:             i ← 1
11:       else
12:             i ← i + 1
13:       end if
14:   end while
15:   return x'
16: end begin
```

El diagrama de flujo que presentamos en la Figura 13 corresponde al algoritmo GRASP diseñado para resolver el problema 2NCSP, en donde se pueden apreciar las seis vecindades implementadas, las cuales serán detalladas en profundidad en la siguiente sección. Una particularidad de nuestro algoritmo es el diseño de un procedimiento de perturbación de la solución, el cual tiene como finalidad escapar de óptimos locales, cuando el procedimiento alcanza el mejor óptimo local para todas las vecindades.

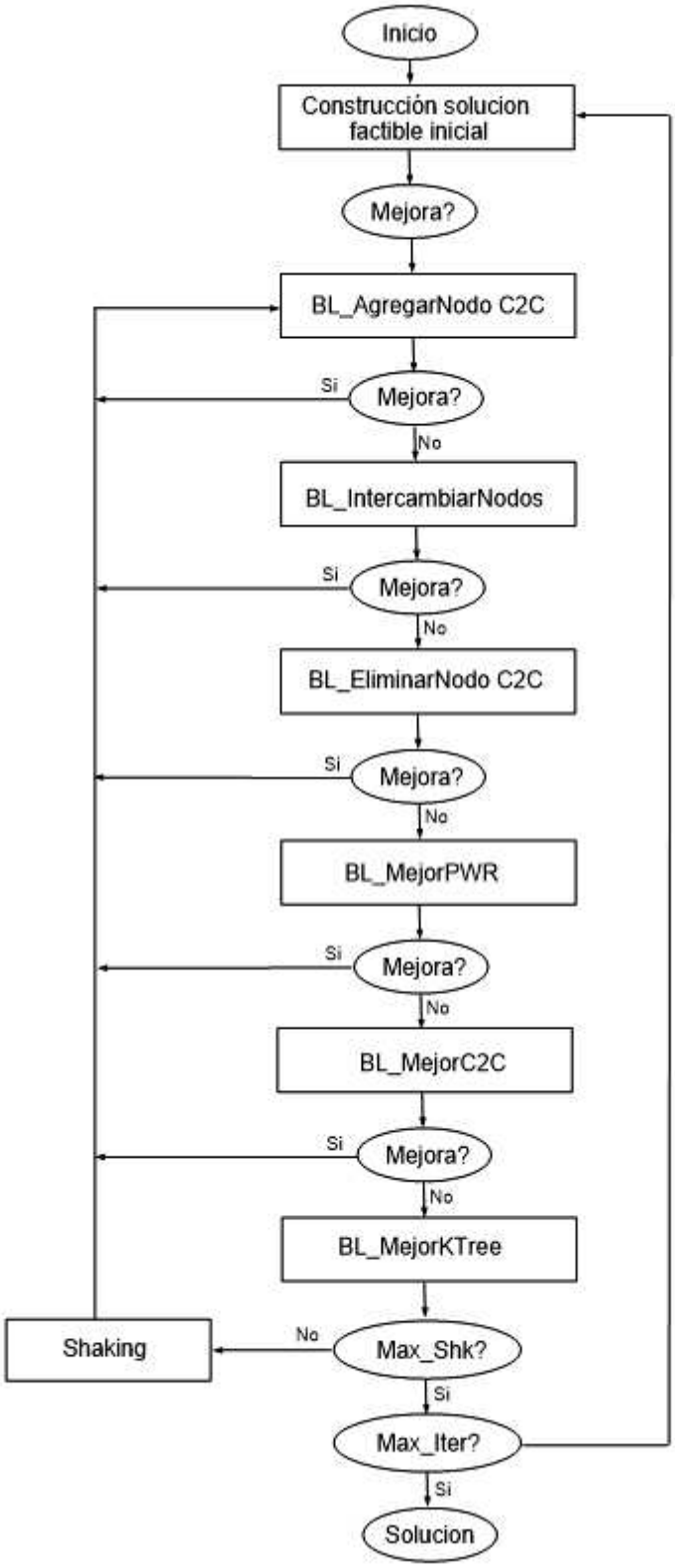


Figura 13: Flujo GRASP-VND de la metaheurística 2NCSP

4.2. Búsqueda Local: Agregar Nodo a componente 2-nodo conexa

El objetivo de esta búsqueda local es agregar cada nodo colgante en la mejor posición dentro de la componente 2-nodo conexa de la solución, procurando mejorar el costo de la misma sin perder factibilidad.

Definición 5.2.1: *Dada una instancia del 2NCSP y una solución factible G_{sol} definimos como vecindad a toda solución G'_{sol} que se pueda obtener mediante el agregado de un nodo colgante de G_{sol} a la componente 2-nodo conexa de la misma.*

Al agregar un nodo a la componente 2-nodo conexa es posible asignarle nodos colgantes que antes estaban asignados a otros nodos con un mayor costo. Por este motivo, para saber si el movimiento realizado mejora la solución, no es suficiente con evaluar el costo de conectar el nodo seleccionado a la componente 2-nodo conexa, sino que además, es necesario volver a asignar los nodos que ya eran colgantes.

Para agregar un nodo colgante a la componente 2-nodo conexa se implementaron dos procedimientos diferentes los cuales serán explicados a continuación.

En las Figuras 14.a y 14.b se presenta el primero de los procedimientos de inserción, el cual consiste en conectar un nodo colgante i a los nodos más cercanos en la componente 2-nodo conexa, los cuales denominaremos j y k. La conexión entre los pares de nodos (i, j) e (i, k) se realiza calculando el camino de costo mínimo entre cada par de nodos.

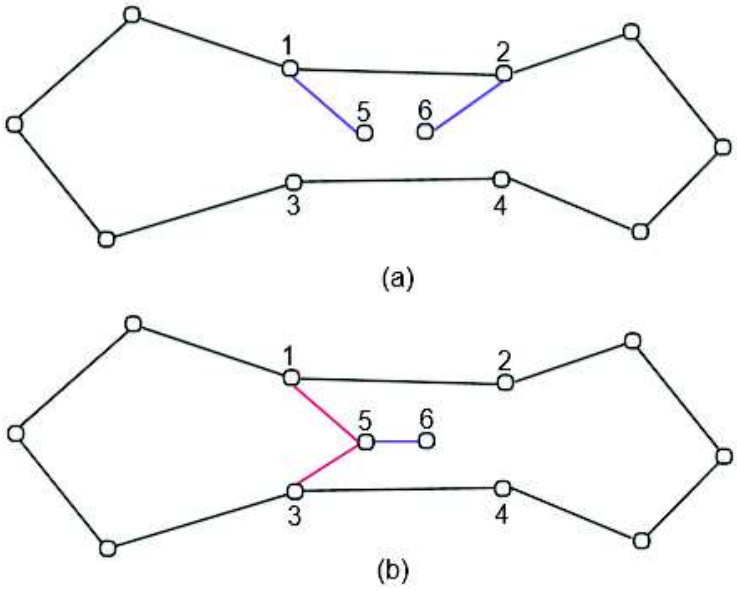


Figura 14: Búsqueda local agregar nodo I

En nuestro ejemplo seleccionamos el nodo colgante $i = 5$, siendo $j = 1, k = 3$ los nodos más cercanos de la componente 2-nodo conexa, para facilitar el ejemplo, los nodos más cercanos se conectan a la componente mediante aristas, pero perfectamente podrían ser caminos de costo mínimo.

Esta forma de agregar un nodo puede generar aristas redundantes en la componente 2-nodo conexa. Imaginemos que en nuestro ejemplo existiese una arista entre los nodos 1 y 3, y se hubiese agregado el nodo 5 tal cual y como lo hicimos, es claro observar que la arista (1,3) se puede eliminar manteniendo el requerimiento de conectividad, disminuyendo entonces el costo de la componente. Por este motivo, una vez asignado un nuevo nodo, se procede a eliminar posibles aristas redundantes haciendo uso del algoritmo descrito en la sección 4.1.3.

En caso de que no existan 2 caminos entre el nodo seleccionado y la componente 2-nodo conexa, el mismo es desechado en esta búsqueda local y se continúa con el siguiente nodo colgante sin analizar.

El segundo procedimiento se ejemplifica en las Figuras 15.a y 15.b, el cual consiste en conectar un nodo colgante i entre dos nodos contiguos j y k de la componente 2-nodo conexa, removiendo luego la arista que conecta j con k. Luego se procede a asignar el resto de los nodos colgantes a algún nodo de la nueva componente 2-nodo conexa. En este caso no es necesario verificar la existencia de aristas redundantes ya que es trivial comprobar que el procedimiento no genera este tipo de aristas.

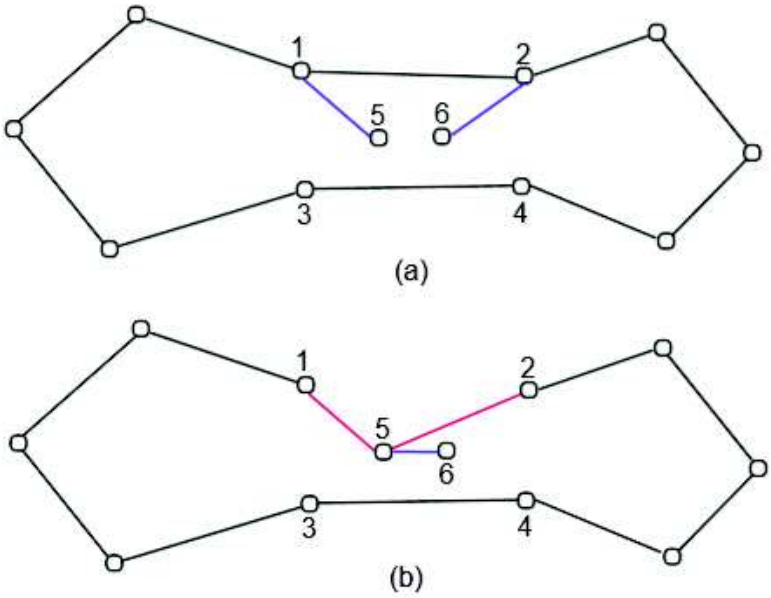


Figura 15: Búsqueda local agregar nodo II

En nuestro ejemplo seleccionamos el nodo colgante $i = 5$, siendo $j = 1$ y $k = 2$ los nodos de la componente 2-nodo conexa entre los cuales se conectará. El algoritmo procede a crear las aristas (5,1) y (5,2) y eliminar la arista (1,2). Luego se asignan los restantes nodos colgantes a los nodos de la nueva componente dos conexa.

El ciclo para cada nodo colgante termina luego de haber contemplado todas las posibles posiciones donde insertar dicho nodo y haber seleccionado el que genera menor costo total.

Una vez finalizado el proceso de inserción (cualquiera sea el utilizado) se reasignan los nodos colgantes restantes, notar en el ejemplo que al reasignarse los nodos, el nodo 6 es asignado al nodo 5, generando posiblemente un mejor costo para la solución.

El procedimiento descrito es aplicado iterativamente para todos los nodos colgantes no examinados, hasta que finalmente se selecciona el movimiento que produce menor costo de la solución.

Consideraciones generales.

En la Figura 14.b se puede observar que el primer procedimiento de inserción diseñado es capaz de generar soluciones 2-nodo conexas que no tengan topología de anillo, mientras que el segundo procedimiento, por su forma de inserción, mantiene la topología de anillo de las soluciones, como se puede apreciar en la Figura 15.b.

Es importante destacar que el algoritmo no genera soluciones no factibles. Si bien es posible que un movimiento no pueda realizarse debido a que no existan caminos o aristas que conecten el nodo a insertar a la componente, una vez que el nodo es agregado, la nueva solución preserva la factibilidad.

Para entender esto, es simple visualizar que en el primer procedimiento, siempre se agrega un nodo en forma 2-nodo conexa, por lo tanto, la solución lo seguirá siendo. En el segundo, si bien se elimina una arista entre un par de nodos de la componente 2-nodo conexa, los mismos se vuelven a conectar utilizando el nuevo nodo, por lo tanto se mantiene la conectividad entre ambos, y por ende, de la solución.

El Algoritmo 12 presenta el primer procedimiento de inserción descrito, el cual recibe la solución al 2NCSP $G_{inicial}$ como parámetro de entrada, y retorna la mejor solución encontrada,G_{best}, la cual es inicializada en la línea 4 con la solución inicial. En la línea 5 se obtiene la componente 2-nodo conexa de $G_{inicial}$, $compConexa$.

Algoritmo 12 – BL_AgregarNodo_C2C

```
1:  procedure BL_AgregarNodo_C2C
2:  input G_inicial
3:  begin
4:    G_best ← G_inicial
5:    compConexa ← obtenerComponenteConexa(G_inicial)
6     for each n ∈ nodosColgantes(G_inicial)
7:        path_1 ← Dijkstra(n, G_inicial, compConexa)
8:        path_2 ← Dijkstra(n, G_inicial, compConexa, obtenerNodos (path_1))
9:        compConexa ← agregarCamino(compConexa, path _1)
10:       compConexa ← agregarCamino(compConexa, path _2)
11:       G_sol ← reasignarColgantes(compConexa, G_inicial)
12:       if costo(G_sol) < costo(G_best)
13:             G_best ← G_sol
14:             compConexa ← obtenerComponenteConexa(G_inicial)
15:       else
16:             compConexa ← obtenerComponenteConexa(G_inicial)
17:       end if
18:   end for
19:   return G_best
20: end begin
```

En las líneas 6 a 18 el algoritmo itera para cada nodo colgante n de $G_{inicial}$ realizando el siguiente procedimiento. En las líneas 7 y 8 se calculan los 2 caminos nodos disjuntos de menor costo entre el nodo colgante n y la componente $compConexa$. En las líneas 9 y 10 se agregan los caminos encontrados a dicha componente. En la línea 11 se reasignan los nodos colgantes restantes de $G_{inicial}$ a algún nodo de la nueva componente 2-nodo-conexa, generando de esta manera la nueva solución al 2NCSP denominada G_{sol}.

En las líneas 12 a 17, se compara el costo de G_{sol} con el costo de la mejor solución G_{best}, obtenida hasta el momento. En caso de ser mejor, se actualiza G_{best} y se procede a insertar el siguiente nodo colgante en la componente 2-nodo conexa de $G_{inicial}$.

Una vez que finalizan todas las iteraciones del bucle 6-18, el algoritmo retorna la solución G_{best}, por ser la mejor solución encontrada por el algortimo

Algoritmo 13 – BL_AgregarNodo_C2C

```
 1: procedure BL_AgregarNodo_C2C
 2: input G_inicial
 3: begin
 4:   G_best ← ∅, best_costo ←∞
 5:   compConexa ← obtenerComponenteConexa(G_inicial)
 6   for each i ∈ nodosColgantes(G_inicial)
 7:      for each j ∈ nodos (compConexa)
 8:         for each k ∈ adyacentes(j) en compConexa
 9:            costoActual ← c_jk
10:            costoNuevo ← c_ij + c_ik - c_jk
11:            if costoNuevo < bestCosto
12:               best_i ← i
13:               best_j ← j
14:               best_k ← k
15:            end if
16:         end for
17:      end for
18:   end for
19:   compConexa ← addPath(arista(best_i, best_j))
20:   compConexa ← addPath(arista(best_i, best_k))
21:   compConexa ← removePath(arista(best_j, best_k))
22:   G_best ← reasignarColgantes(compConexa, G_inicial)
23:   return G_best
24: end begin
```

El Algoritmo 13 presenta el segundo procedimiento de inserción descrito. En la línea 4, se inicializa la mejor solución obtenida hasta el momento G_{best}. En la línea 5 se obtiene la componente 2-nodo conexa de $G_{inicial}$, denominándola $compConexa$.

En las líneas 6 a 18 se itera por cada nodo colgante i de $G_{inicial}$, y por cada par de nodos adyacentes j, k en la componente 2-nodo conexa, realizado el siguiente procedimiento. En las líneas 9 y 10 se realiza el cálculo del costo incremental de insertar el nodo i entre los nodos j y k de las $compConexa$. En la línea 11 se evalúa si el nuevo costo es menor al mejor costo obtenido hasta el momento, de serlo, se actualiza el mejor nodo colgante $best\_i$, y los nodos entre los cuales debe agregarse, $best\_j$ y $best\_k$.

Una vez que finalizan todas las iteraciones, en las líneas 19 y 20 se agrega el nodo $best\_i$, entre los nodos $best\_j$ y $best\_k$ de la componente 2-nodo conexa. En la línea 21 se elimina la arista $(best\_j, best\_k)$, finalizando así el movimiento de inserción. Por último, en la línea 22, se vuelven a reasignar los restantes nodos colgantes retornando la nueva solución G_{best}.

4.3. Búsqueda Local: Intercambiar posición nodo en componente 2-nodo conexa

La idea de esta búsqueda local es extraer todos los nodos de grado dos de la componente 2-nodo conexa y reubicarlos en otra posición que mejore costo de la solución sin perder factibilidad.

Definición 5.3.1: *Dada una instancia del 2NCSP y una solución factible G_{sol} definimos como vecindad a toda solución G'_{sol} que se pueda obtener modificando la ubicación de un nodo cualquiera de la componente 2-nodo conexa dentro de la misma.*

El procedimiento de extracción consiste en eliminar un nodo de grado dos, al que llamaremos *i*, de la componente 2-nodo conexa (junto con sus dos aristas) y conectar entre si los dos nodos adyacentes al mismo. En caso de no existir una arista que conecte ambos nodos, el mismo se descarta y se procede a intercambiar otro nodo. En la Figura 16.a se presenta la componente 2-nodo conexa antes de extraer el nodo *i* = 5 y en la Figura 16.b el estado de la solución luego de extraer dicho nodo.

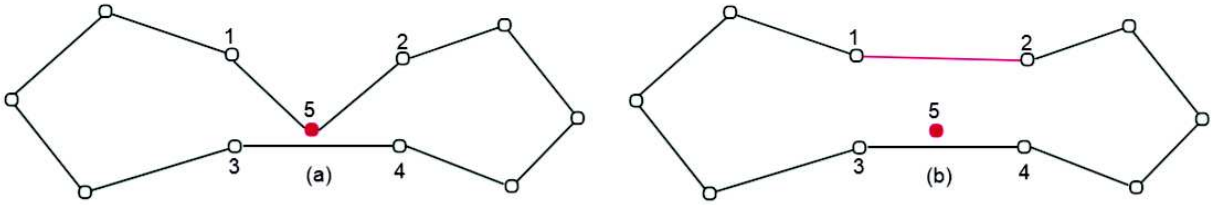


Figura 16: Búsqueda local intercambio de nodos – extracción

Para agregar el nodo eliminado en la componente 2-nodo conexa se utiliza la misma técnica que en segundo procedimiento de inserción de la búsqueda local anterior. En la Figura 17.b se presenta la solución al agregar el nodo eliminado anteriormente entres los nodos 3 y 4, mientras que en la Figura 17.a se puede observar la componente 2-nodo conexa final, luego de eliminar la arista (3,4).

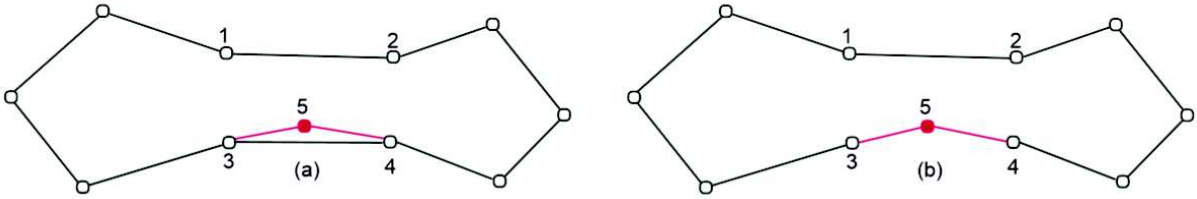


Figura 17: Búsqueda local intercambio de nodos - inserción

Luego que se realizó el procedimiento de inserción en todas las posibles posiciones de la componente 2-nodo conexa, se elige aquella ubicación que genere menor costo, y en caso de ser la mejor solución encontrada hasta el momento, la misma se actualiza y el procedimiento continúa seleccionando un nuevo nodo hasta que todos los nodos hayan sido procesados. Una vez examinados todos los nodos el procedimiento aplica el movimiento que haya generado la mejor solución.

Consideraciones generales.

Esta búsqueda local no realiza movimientos que violen la restricción de conectividad, ya que en caso de que el nodo eliminado no pueda asignarse en ningún otro lugar de la componente 2-nodo conexa, siempre se podrá asignar en el mismo lugar en el que estaba inicialmente.

El Algoritmo 14 presenta el pseudocódigo de la búsqueda local de intercambio de nodos, el cual recibe como parámetro de entrada la solución inicial $G_{inicial}$ al 2NCSP la cual será modificada en búsqueda de soluciones de mejor costo.

En la línea 4 se inicializa la variable $compConexa$ con la componente 2-nodo conexa de $G_{inicial}$, mientras que en la línea 5 se inicializa de igual manera la variable auxiliar $compConexaBest$, que mantendrá la mejor componente 2-nodo conexa encontrada durante la búsqueda local.

Algoritmo 14 – BL_IntercambiarNodos

```
 1: procedure BL_IntercambiarNodos
 2: input Ginicial
 3: begin
 4:   compConexa ← obtenerComponenteConexa(Ginicial)
 5:   compConexaBest ← compConexa
 6:   for each i ∈ nodos(compConexa)
 7:       if grado(i) = 2
 8:           compConexaAux ← eliminarNodo(i, compConexa)
 9:           compConexaAux ← agregarNodoEnMejorPosicion(i, compConexaAux)
10:           if costo(compConexaAux) < costo(compConexaBest)
11:                   compConexaBest ← compConexaAux
12:           end if
13:       end if
14:   end for
15:   Gbest ← reemplazarComponenteConexa(Ginicial, compConexaBest)
19:   return Gbest
20: end begin
```

El bucle de las líneas 6 a 14 se repite para cada nodo i de $compConexa$, procediendo de la siguiente manera. En la línea 7 se verifica que el nodo i tenga grado 2, en caso contrario se procesa el siguiente nodo de $compConexa$. Si el nodo es de grado 2, en la línea 8 se lo elimina de la componente 2-nodo conexa, almacenando el resultado en la variable $compConexaAux$. En la línea 8 se agrega dicho nodo en $compConexaAux$, pero esta vez, en la posición que minimice el costo total de la misma. Si el costo de la nueva componente es mejor que el mejor costo obtenido hasta el momento, se actualiza $compConexaBest$, con la nueva componente ($compConexaAux$).

Al finalizar el bucle, se reemplaza la componente 2-nodo conexa de $G_{inicial}$, por $compConexaBest$, la componente 2-nodo conexa de menor costo, encontrada por esta búsqueda local

4.4. Búsqueda Local: Eliminar nodo de componente 2-nodo conexa

El objetivo de esta búsqueda local es extraer nodos de grado dos de la componente 2-nodo conexa y colgarlos de algún otro nodo de la misma, siempre que esto produzca un mejor costo de la solución, sin perder su factibilidad.

Definición 5.4.1: *Dada una instancia del 2NCSP y una solución factible G_{sol} definimos como vecindad a toda solución G'_{sol} que se pueda obtener mediante la extracción de cualquier nodo de la componente 2-nodo conexa y asignarlo como colgante al nodo más cercano de la misma.*

El procedimiento para extraer nodos es igual al utilizado en la búsqueda local de intercambio de nodos de la sección 4.3, el cual consiste en quitar las dos aristas incidentes al nodo seleccionado (de grado 2) y conectar sus dos nodos adyacentes mediante una arista en común. En caso de no existir dicha arista, no es factible extraer el nodo.

Luego de eliminado el nodo, se transforma en colgante, asignándolo al nodo más cercano de la componente 2-nodo conexa y se evalúa el nuevo costo de la solución. En caso de que la nueva solución tenga mejor costo que la mejor solución encontrada hasta el momento, la misma se actualiza y se vuelve a ejecutar el procedimiento con un nuevo nodo de grado 2.

El procedimiento anterior se repite para todos los nodos de grado dos de la componente 2-nodo conexa, hasta que finalmente se selecciona el movimiento que produjo la mejor solución. El Algoritmo 15 presenta el pseudocódigo de esta búsqueda local que recibe cómo parámetro la solución $G_{inicial}$ al 2CNSP.

Algoritmo 15 – BL_EliminarNodo_C2C

```
 1: procedure BL_EliminarNodo_C2C
 2: input G_inicial
 3: begin
 4:   G_best ← G_inicial
 5:   compConexa ← obtenerComponenteConexa(G_inicial)
 6   for each i ∈ nodos(compConexa)
 7:     if δ(i) = 2
 8:       compConexa ← agregarArista(adyacentes(i)[0], adyacentes(i)[1])
 9:       compConexa ← eliminarArista(i, adyacentes(i)[0])
10:       compConexa ← eliminarArista(i, adyacentes(i)[1])
11:       G_sol ← reasignarColgantes(compConexa, G_inicial)
12:       if costo(G_sol) < costo(G_best)
13:           G_best ← G_sol
14          end if
15:       compConexa ← obtenerComponenteConexa(G_inicial)
16:    end if
17:  end for
18:  return G_best
19: end begin
```

En la línea 4 se inicializa la variable G_{best} con $G_{inicial}$, la cual se utilizará para mantener la mejor solución que vaya encontrando el algoritmo. En la línea 5 se inicializa la variable $compConexa$ con la componente 2-nodo conexa de $G_{inicial}$.

El bucle de las líneas 6 a 17 itera por cada nodo i de $compConxa$ realizando las siguientes acciones. En la línea 7 se verifica que el nodo i tenga grado 2, en caso contrario, se prosigue con el siguiente nodo de $compConexa$. Si el nodo es de grado 2, en la línea 8 se agregar la arista que une los dos nodos adyacentes a i, mientras que en las líneas 9 y 10, se desconecta el nodo i, eliminando las dos aristas que lo conectaban a sus adyacentes. En la línea 11 se reasignan los nodos colgantes, ahora con el nodo i como un colgante más, obteniendo una nueva solución G_{sol}. Si el cosot de la nueva solución es menor al mejor costo obtenido hasta el momento, se reemplaza G_{best} con la solución G_{sol}. Por último, en la línea 12 se vuelve a la $compConexa$ original, para comenzar el bucle con un nuevo nodo.

Al finalizar el bucle, en la línea 18, se retorna G_{best}, que contiene la solución al 2NCSP de menor costo encontrada.

4.5. Búsqueda Local: Mejor camino con nodos colgantes

El objetivo de esta búsqueda local es reemplazar un camino con nodos colgantes por el camino con nodos colgantes de costo mínimo, mediante la implementación de un modelo de programación lineal entera.

Definición 5.5.1: Camino con nodos colgantes.
Dado un grafo $G(V, E)$ es un camino con nodos colgantes con extremos $u, v \in V$, si se cumplen las siguientes condiciones:

- *G es conexo y sin ciclos*
- *$p(u, v) \subseteq G$ es el camino que conecta a u con v en G*
- *Todos los nodos que no pertenecen a p están conectados a algún nodo de p a través de una arista simple.*

La Figura 19 presenta un ejemplo de un camino con nodos colgantes con extremos $u = 6$ y $v = 21$.

Definición 5.5.2: *Dada una instancia del 2NCSP y una solución factible G_{sol} definimos como vecindad a toda solución G'_{sol} que se pueda obtener modificando cualquier camino con nodos colgantes por otro camino con nodos colgantes que incluya los mismos nodos.*

El procedimiento de esta búsqueda local consiste en encontrar caminos de nodos colgantes y aplicar a cada uno de ellos un modelo de programación lineal entera para encontrar la mejor solución posible con esta topología. Para comprender mejor esta búsqueda local, haremos uso del ejemplo de la Figura 19.

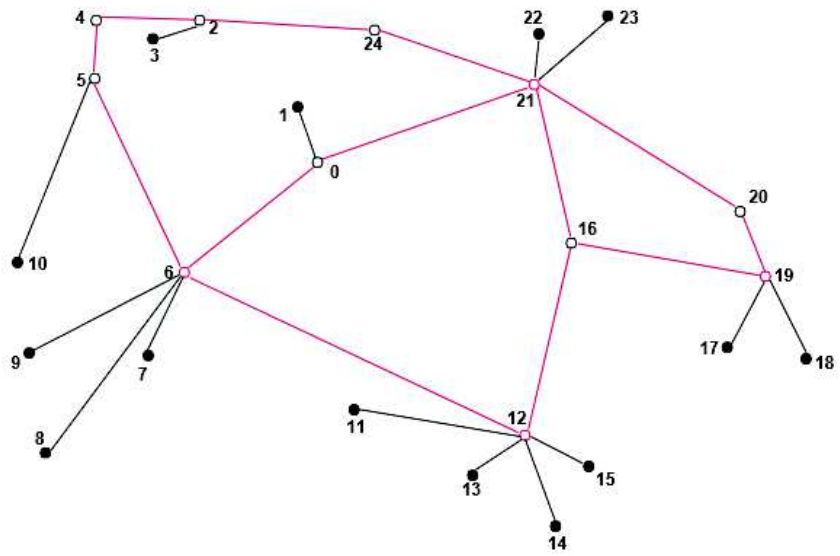


Figura 18: Búsqueda local camino con nodos colgantes - Solución factible inicial

Supongamos que el camino seleccionado es el conformado por los nodos (6, 5, 4, 2, 24, 21), lo primero que se verifica es que la cantidad de nodos del camino, más los nodos colgantes del mismo no supere un parámetro dado MAX_NODOS_CAMINO.

El motivo de esta restricción es que al tratarse de un método exacto, con un elevado costo computacional, si la cantidad de nodos es elevada, los tiempos de ejecución hacen impracticable esta técnica (en nuestro caso, usamos MAX_NODOS_CAMINO = 10). En la Figura 19 se presenta el camino con nodos colgantes resultante.

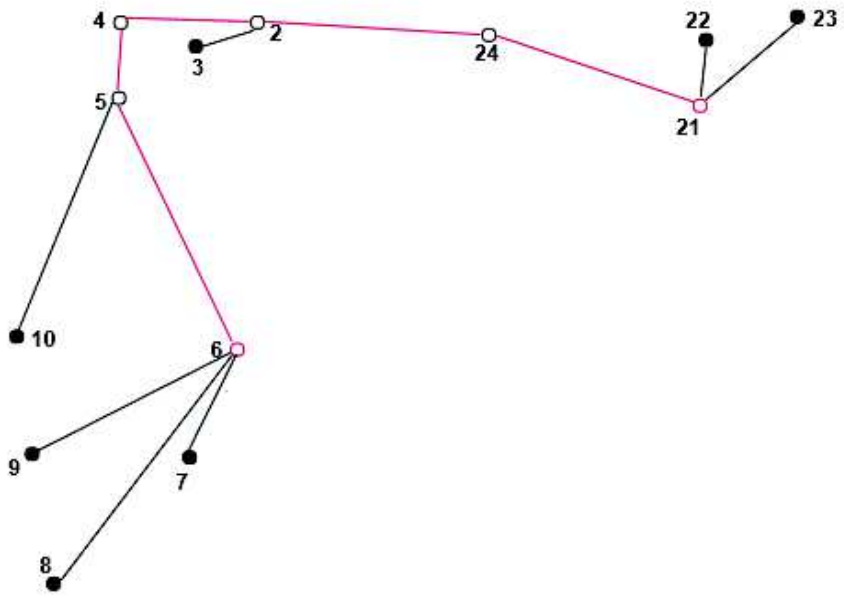


Figura 19: Búsqueda local camino con nodos colgantes

Una vez seleccionado el camino, se considera el grafo inducido que contiene los vértices del camino y sus nodos colgantes. El grafo inducido se considera sobre el grafo original, por lo tanto, contiene todas las aristas de dicho grafo, cuyos extremos pertenecen al conjunto de nodos descrito. La Figura 20 presenta el grafo inducido para nuestro ejemplo.

El grafo inducido es utilizado como entrada del algoritmo exacto que calcula el mejor camino con nodos colgantes entre los nodos extremos del camino con colgantes, nodos 6 y 21. Dichos nodos deberán mantenerse como inicio y fin del nuevo camino con colgantes. En la Figura 21 se presenta el mejor camino encontrado.

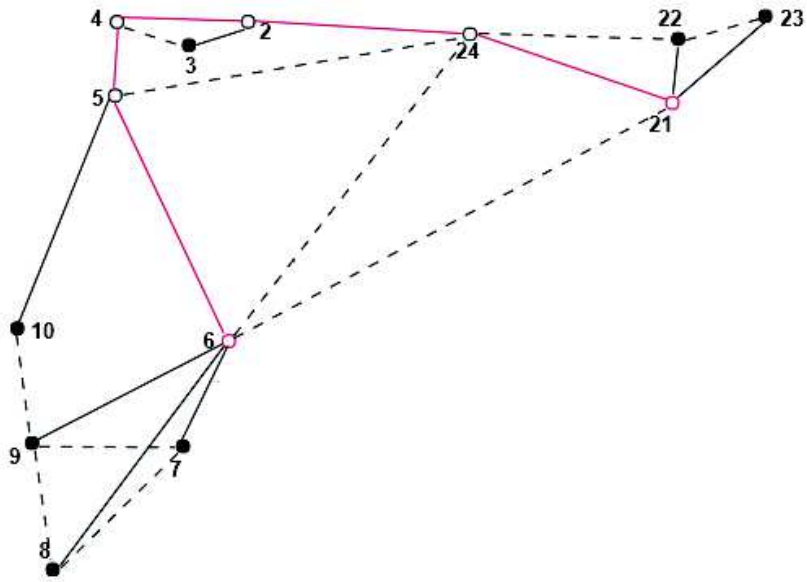


Figura 20: Búsqueda local camino nodos colgantes - grafo inducido

Una vez obtenido el mejor camino con colgantes, sustituimos el camino original por el camino con colgantes óptimo, siempre que el costo haya mejorado, obteniendo así una nueva solución factible de menor costo. En la Figura 22 se puede apreciar la nueva solución factible al 2NCSP.

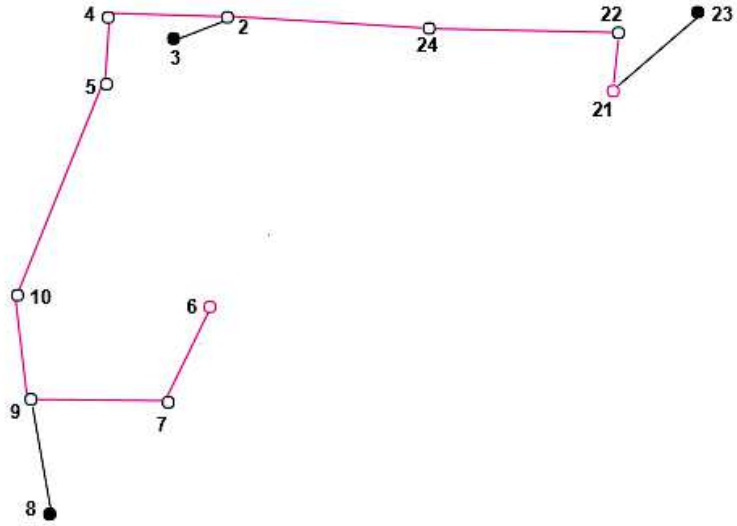


Figura 21: Búsqueda local camino nodos colgantes - camino óptimo

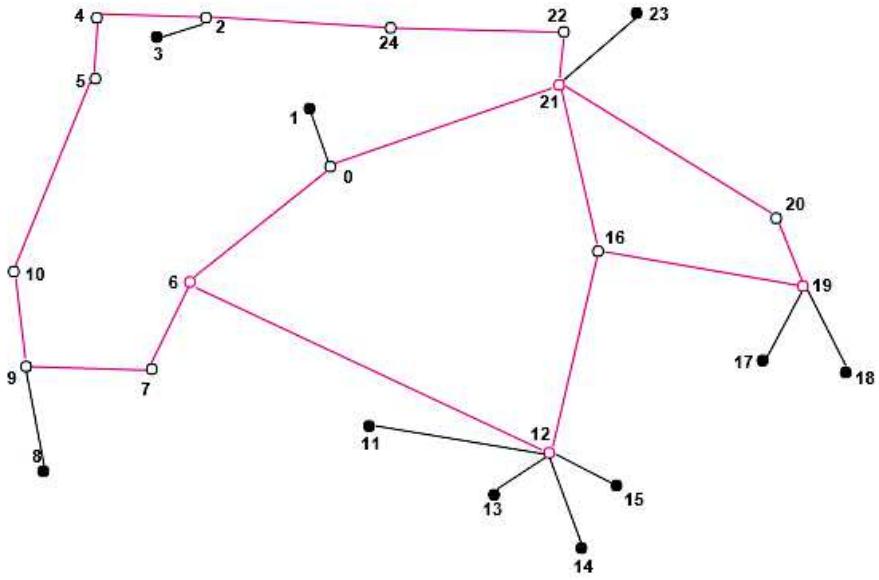


Figura 22: Búsqueda local camino nodos colgantes - nueva solución factible

El Algoritmo 16 presenta el pseudocódigo de la búsqueda local mejor camino con nodos colgantes, el cual recibe como parámetros la solución G_{sol} al 2NCSP, y el grafo original $G_{original}$ del problema. En la línea 4, se inicializa la variable G_{best} con la solución G_{sol}, la cual es utilizada para mantener la mejor solución obtenida por la búsqueda local.

El bucle de las líneas 6 a 17 itera por cada camino C con colgantes de G_{sol}, realizando el siguiente procedimiento. En la línea 7 se verifica que el camino con colgantes C no contenga más de MAX_NODOS_CAMINO nodos, en caso contrario, el camino se descarta y se continúa con el siguiente. En las líneas 8 y 9 se obtienen los nodos extremos del camino, denominados $nodo\_a$ y $nodo\_b$. En la línea 10 se genera el grafo inducido por los nodos del camino con colgantes en el grafo original, mientras que en la línea 11 se hace uso del método exacto que retorna C_{best}, el mejor camino con nodos colgantes con extremos en $nodo\_a, nodo\_b$.

En caso de que el costo de C_{best} sea menor que el costo del camino con colgantes original, se finaliza la búsqueda local retornando la solución luego de reemplazar el camino C por C_{best} (línea 13).

Algoritmo 16 – BL_MejorCaminoConNodosColgante

```
1:  procedure BL_ MejorCaminoConNodosColgantes
2:  input Gsol, Goriginal
3:  begin
4:     Gbest ← Gsol
5:
6:     for each C ∈ caminosConColgantes(Gsol)
7:        if cantidadNodos(camino) < MAX_NODOS_CAMINO
8:           nodo_a ← obtenerNodos(C)[0]
9:           nodo_b ← obtenerNodos(C)[c.length-1]
10:          I ← obtenerGrafoInducido(C, Gsol, Goriginal)
11:          Cbest ← obtenerMejorCaminoConColgantes(I, nodo_a, nodo_b)
12:          if costo(Cbest) < costo(C)
13:                Gbest ← Gbest – C + Cbest
14:                return Gbest
15:           end if
16:       end if
17:    end for
18:    return Gbest
19:  end begin
```

4.5.1. Método exacto para computar camino con colgantes óptimo

Como se mencionó previamente, este algoritmo se resuelve de forma exacta basado en un modelo de Programación Lineal Entera, el cual, dado un grafo con dos nodos distinguidos a y b, retorna el mejor camino $p(a, b)$ con nodos colgantes. A continuación se presenta el modelo implementado.

- Sea $G(V, E)$ un grafo donde V es el conjunto de vértices y E es el conjunto de aristas.
- Sean u, v dos nodos distinguidos del conjunto de vértices V.
- Sea $V' = V \setminus (\{a\} \cup \{b\})$ el conjunto de nodos que no incluye los extremos.
- Sea $C = \{c_{ij}\}_{i,j \in V}$ la matriz de costos de conexión del grafo, es decir, los costos de una arista (i, j) cuando la misma pertenece al camino $p(a, b)$.
- Sea $D = \{d_{ij}\}_{i,j \in V}$ la matriz de costos de asignación del grafo, es decir, los costos de una arista (i, j) cuando la arista tiene uno de los extremos en algún nodo del camino principal $p(a, b)$ y el otro extremo es un nodo colgante.

Se definen las siguientes variables para el modelo.

$$X_i = \begin{cases} 1 \; si \; el \; nodo \; i \; \in V \; forma \; parte \; del \; camino \\ 0 \; en \; caso \; contrario \end{cases}$$

$$Y_i = \begin{cases} 1 \; si \; el \; nodo \; i \; \in V' es \; un \; nodo \; colgante \\ 0 \; en \; caso \; contrario \end{cases}$$

$$Yc_{i,j} = \begin{cases} 1 \; si \; el \; nodo \; \; i \in V' \; cuelga \; del \; nodo \; j \in V \\ 0 \; en \; caso \; contrario \end{cases}$$

$$x_{i,j} = \begin{cases} 1 \; si \; la \; arista \; (i,j) \; es \; usada \; en \; la \; solución \\ 0 \; en \; caso \; contrario \end{cases}$$

$$w_{ij} = \begin{cases} 1 \; si \; la \; arista \; (i,j) \; colgante \; pertenece \; a \; la \; solución \\ 0 \; en \; caso \; contrario \end{cases}$$

$$y_{(i,j)}^{(u,v)} = \begin{cases} 1 \; si \; la \; arista \; (i,j) \; es \; usada \; en \; el \; camino \; que \; va \; de \; u \; a \; v \\ 0 \; en \; caso \; contrario \end{cases}$$

$$Z = variable \; auxiliar \; usada \; para \; condicionar \; restricciones$$

Considerando las variables anteriores, el modelo de programación lineal entera se define de la siguiente forma:

$$\min\Big(\sum_{i,j \in V} c_{ij}\big(x_{ij} - w_{ij}\big) + \sum_{i,j \in V} d_{ij} w_{ij} \Big) \tag{4.5.1}$$

Sujeto a:

$$X_i + Y_i = 1 \;\; \forall i \in V' \tag{4.5.2}$$

$$X_i = 1 \ \ \forall i \in (\{u\} \cup \{v\}) \tag{4.5.3}$$

Las restricciones 4.5.2 y 4.5.3 las denominamos restricciones de pertenencia, la restricción 2 nos asegura que un nodo o pertenece al camino principal o está colgado del camino mediante una arista, mientras que la restricción 3 indica que los extremos a y b del camino deben pertenecer al camino principal.

$$\sum_{j \in ady(u)} y_{(u,j)}^{(u,v)} = 1 \qquad \forall u, v \ \in V; \ u \neq v \tag{4.5.4}$$

$$\sum_{i \in ady(v)} y_{(i,v)}^{(u,v)} = 1 \qquad \forall u, v \in V; \ u \neq v \tag{4.5.5}$$

$$\sum_{i \in ady(p)} y_{(i,p)}^{(u,v)} - \sum_{i \in ady(p)} y_{(p,i)}^{(u,v)} = 0 \ \ \forall u, v \in V; \forall p \in V \backslash u, v; \ u \neq v; \ u \neq p \tag{4.5.6}$$

$$y_{(i,j)}^{(u,v)} + y_{(j,i)}^{(u,v)} \ \leq x_{i,j} \qquad \forall u, v \in V; \ u \neq v; \ \forall (i,j) \in E \tag{4.5.7}$$

Las restricciones 4.5.4 a 4.5.7 son restricciones de conexidad. Las ecuaciones 4.5.4 y 4.5.5 indican que en un camino entre u y v solo hay una arista saliente u y una única arista que llega a v. La ecuación 4.5.6 es una restricción de balance, mientras que la restricción 4.5.7 implica que el camino con colgantes sea arista-disjunto.

$$Yc_{ij} \ \leq X_j \ \ \forall i \in V'; \forall j \in ady(i) \tag{4.5.8}$$

$$Y_i \leq Z_i \qquad \forall i \in V' \tag{4.5.9}$$

$$\sum_{j \in ady(i)} x_{i,j} \leq K(1 - Z_i) + 1 \ \ \forall i \in V'; M \leq \max(\delta i) \, con \, i = 1..|V| \tag{4.5.10}$$

$$Yc_{i,j} = w_{i,j} \qquad \forall i \in V; j \in ady(i) \tag{4.5.11}$$

$$w_{i,j} \leq x_{i,j} \qquad \forall i \in V'; \ j \in ady(i) \tag{4.5.12}$$

$$\sum_{j \in ady(i)} w_{i,j} \leq Y_i \qquad \forall i \in V \tag{4.5.13}$$

$$Y_i = \sum_{j \in ady(i)} Yc_{i,j} \qquad \forall i \in V \tag{4.5.14}$$

Las ecuaciones 4.5.8 a 4.5.14 establecen restricciones para los nodos colgantes. La restricción 4.5.8 indica que si los nodos i y j están conectados siendo i un nodo colgante, entonces j debe pertenecer al camino principal. Las restricciones 4.5.9 y 4.5.10 implican que el grado de cualquier nodo colgante debe ser 1, pero si el nodo pertenece al camino principal, no se establece una restricción en cuanto a su grado.

La restricción número 4.5.11 implica que si el nodo i cuelga del nodo j, entonces la arista (i, j) es colgante y pertenece a la solución, mientras que la restricción 4.5.12 establece que si una arista es colgante, entonces pertenece a la solución. Por último la restricción 4.5.13 asegura no haya más de una arista que sea incidente que al nodo i si éste es colgante, mientras que la restricción 4.5.14 asegura que si un nodo i cuelga del camino principal, entonces cuelga de un solo nodo.

$$\sum_{i \in ady(j)} \left(Yc_{i,j}\right) + X_j - \sum_{i \in ady(j)} x_{j,i} = 0 \qquad \forall j \in (\{a\} \cup \{b\}) \tag{4.5.15}$$

La restricción 4.5.15 implica que el grado de los nodos extremos a y b debe ser 1, mientras que la ecuación 4.5.16 asegura que los nodos del camino principal deben ser de grado dos, en ambos casos sin considerar para calcular el grado de los nodos, las aristas colgantes a los mismos.

$$\sum_{i \in ady(j)} \left(Yc_{i,j} + Yc_{j,i}\right) + 2X_j - \sum_{i \in ady(j)} x_{j,i} = 0 \qquad \forall j \in V \tag{4.5.16}$$

$$x_{i,j} = x_{j,i} \;\; \forall i, j \in V; i \neq j \tag{4.5.17}$$

La última ecuación, se denomina ecuación de simetría e indica que si la arista (i, j) pertenece a la solución, entonces la arista (j, i) también pertenece.

4.6. Búsqueda Local: Mejor componente 2-nodo-conexa

El objetivo de esta búsqueda local es aplicar a cada ciclo Q de un grafo solución factible al 2NCSP, un procedimiento que a partir del grafo inducido por Q en el grafo original, determine un subgrafo Q_{sol} de mejor costo, con topología 2-nodo-conexa, sin que necesariamente tenga topología de anillo.

Definición 5.6.1
Dada una instancia del 2NCSP y una solución factible G_{sol} definimos como vecindad a toda solución factible G'_{sol} que se pueda obtener reemplazando un ciclo Q, del grafo G_{sol} por un subgrafo Q_{sol} 2-nodo-conexo de menor costo que cubra los nodos del ciclo Q.

Esta búsqueda local implementa 3 algoritmos que permiten mejorar el costo de los ciclos de la solución, los cuales se basan en:

- un modelo de programación lineal entera
- algoritmos genéticos
- movimientos 2-opt

En la Figura 23 se presenta un ejemplo de sustitución de un ciclo por una componente 2-nodo conexa de mejor costo.

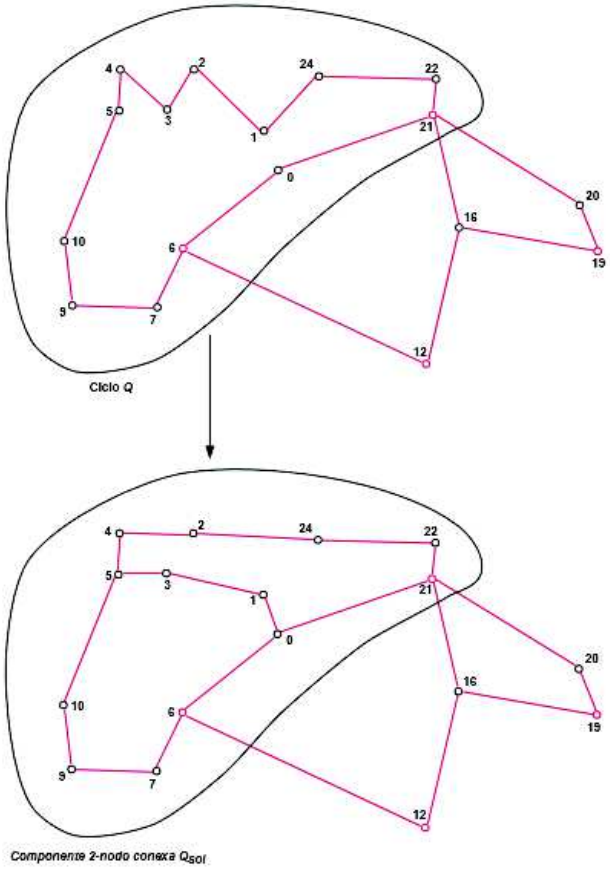


Figura 23: Sustitución del ciclo Q por la componente 2-nodo conexa de mejor costo

El método basado en un modelo de programación lineal entera no sólo ofrece soluciones óptimas, sino que además, las soluciones son 2-nodo conexas. Como es sabido, la solución óptima 2-nodo conexa que cubre un determinado conjunto de nodos no necesariamente tiene topología de anillo, por lo tanto, este método es capaz de encontrar soluciones que las otras técnicas implementadas no son capaces de generar.

El Algoritmo 17 presenta el pseudocódigo de la búsqueda local mejor componente 2-nodo-conexa, el cual recibe como parámetros la solución G_{sol} al 2NCSP, y el grafo original $G_{original}$ del problema. En la línea 4, se inicializa la variable G_{best} con la solución G_{sol}, la cual es utilizada para mantener la mejor solución obtenida por la búsqueda local.

El bucle de las líneas 6 a 18 itera por cada ciclo C de la componente 2-nodo conexa de G_{sol}, realizando el siguiente procedimiento. En la línea 7 se verifica que el ciclo C no contenga más de MAX_NODOS_CICLO nodos, en caso contrario, el ciclo se descarta y se continúa con el siguiente. En la línea 8 se obtiene el grafo inducido por C en el grafo original, el cual luego es utilizado por el procedimiento de Split en la línea 9. El grafo resultante S es utilizado en la línea 10 donde se hace uso del método exacto que retorna C_{best}, la componente 2-nodo conexa de costo mínimo.

En caso de que el costo de C_{best} sea menor que el costo del ciclo original C, se finaliza la búsqueda local retornando la solución luego de reemplazar el ciclo C por C_{best} (línea 15).

Algoritmo 17 – BL_MejorComponente2Conexa

```
 1: procedure BL_ MejorComponente2Conexa
 2: input Gsol, Goriginal
 3: begin
 4:    Gbest ← Gsol
 5:
 6:    for each C ∈ ciclos(Gsol)
 7:       if cantidadNodos(ciclo) < MAX_NODOS_CICLO
 8:          I ← obtenerGrafoInducido(C, Goriginal)
 9:          S ← split(I)
10:          Cbest ← obtenerMejorComponente2Conexa(S)
11:       else
12:             Cbest ← 2-opt (C, Goriginal)
13:       end if
14:       if costo(Cbest) < costo(C)
15:             Gbest ← Gbest – C + Cbest
16:             return Gbest
17:       end if
18:   end for
19:   return Gbest
20: end begin
```

4.6.1. Método exacto para computar componente 2-nodo conexa óptima

Se formula un modelo de programación lineal entera, el cual, dado un grafo, retorna otro con topología 2-arista-conexo que cubre todos los nodos del grafo original.

Para que el grafo resultante sea 2-nodo-conexo es necesario transformar el grafo original $G''(V'', E'')$ en un grafo dirigido $G'(V', E')$, y aplicar la técnica de separación de vértices o Splitting [37] (que se detallará más adelante), para obtener un nuevo grafo $G(V, E)$, el cual es usado como entrada del modelo.

Por último debe aplicarse al resultante del modelo de programación lineal entera las operaciones inversas de spliting, es decir, unir los nodos y quitar la dirección de las aristas. De esta forma, obtenemos un grafo con topología 2-nodo conexa de costo mínimo.

A continuación se presenta el modelo implementado.

- Sea $G(V, E)$ un grafo donde V es el conjunto de vértices y E es el conjunto de aristas.
- Sea $C = \{c_{ij}\}_{i,j \in V}$ la matriz de costos de conexión del grafo, es decir, los costos de una arista (i, j) cuando la misma pertenece a la estructura 2-nodo-conexa

Se definen las siguientes variables para el modelo.

$$x_{i,j} = \begin{cases} 1 \; si \; la \; arista \; (i,j) \; es \; usada \; en \; la \; solución \\ 0 \; en \; caso \; contrario \end{cases}$$

$$y_{(i,j)}^{(u,v)} = \begin{cases} 1 \; si \; la \; arista \; (i,j) \; es \; usada \; en \; el \; camino \; que \; va \; de \; u \; a \; v \\ 0 \; en \; caso \; contrario \end{cases}$$

Considerando las variables anteriores, el modelo de programación lineal entera se define de la siguiente forma:

$$\min\left(\sum_{i,j \in V} c_{ij} \, x_{ij} \right) \tag{4.6.1}$$

Sujeto a:

$$\sum_{j \in ady(u)} y_{(u,j)}^{(u,v)} \geq 2 \qquad \forall u, v \in V;\ u \neq v \tag{4.6.2}$$

$$\sum_{i \in ady(v)} y_{(i,v)}^{(u,v)} \geq 2 \qquad \forall u, v \in V;\ u \neq v \tag{4.6.3}$$

Las ecuación 4.6.2 implica que para todo nodo de V, salen al menos dos caminos hacia cualquier otro nodo de V. De la misma manera, la ecuación 4.6.3 indica que para cada nodo de V llegan al menos dos caminos de cualquier otro nodo de V.

$$\sum_{i \in ady(p)} y_{(i,p)}^{(u,v)} - \sum_{i \in ady(p)} y_{(p,i)}^{(u,v)} \geq 0 \quad \forall u, v \in V;\ u \neq v;\ \forall p \in V \backslash u, v \tag{4.6.4}$$

$$y_{(i,j)}^{(u,v)} + y_{(j,i)}^{(u,v)} \leq x_{i,j} + x_{j,i} \qquad \forall u, v \in V;\ u \neq v;\ \forall (i,j) \in E \tag{4.6.5}$$

$$x_{i,j} + x_{j,i} \leq 1 \qquad \forall (i,j) \in E \tag{4.6.6}$$

$$y_{(i,j)}^{(u,v)} \geq 0 \qquad \forall u, v \in V;\ u \neq v;\ \forall (i,j) \in E \tag{4.6.7}$$

$$x_{i,j} \leq 1 \qquad \forall (i,j) \in E \tag{4.6.8}$$

La restricción 4.6.4 obliga al equilibrio entre las aristas de salida y de entrada en los nodos intermedios de los caminos. Mientras que la restricción número 4.6.5 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. La restricción 4.6.6 indica que si la arista (i, j) pertenece a la solución, entonces la arista (j, i) no puede pertenecer a la misma.

Separación de vértices o Splitting.

La técnica de separación de vértices de un grafo $G'(V', E')$ consiste en sustituir cada vértice $v \in V'$ por dos v_1, v_2 en el grafo resultante $G(V, E)$, de forma tal que cada arista que llegaba a v llegará al nuevo vértice v_1 y cada arista que salía de v, saldrá del vértice

v_2. Por último v_1 y v_2 se conectan a través de una nueva arista (v_1, v_2) con costo cero. En la Figura 24 se representa esta técnica.

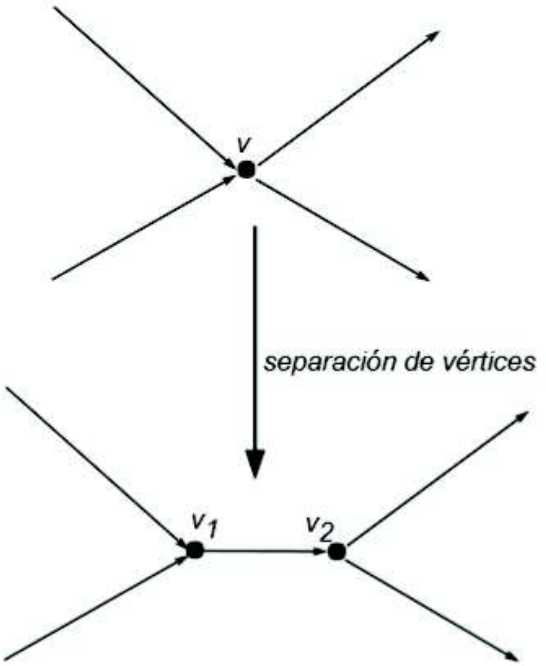


Figura 24: Técnica de separación de vértices.

La técnica de separación de vértices obliga a que una vez que se llega a un vértice v_1, la forma de "salir" es utilizando la única arista disponible (v_1, v_2). Considerando que el modelo de programación lineal entera presentado nos asegura que dado un par de nodos u, v existen dos caminos arista-disjuntos que los conectan, y que, una vez que se usa un nodo en un camino, solo existe una arista de salida, concluimos que ese nodo sólo será visitado por uno de los dos caminos, por lo tanto, la solución es también 2-nodo-conexa.

4.6.2. Algoritmo genético para computar mejores ciclos

Este algoritmo tiene la finalidad de encontrar un ciclo con el menor costo posible, a partir de un ciclo obtenido en la solución del 2NCSP. Básicamente resuelve el Problema del Agente Viajero (TSP) introducido en 1930 por Karl Menger, el cual ha sido ampliamente estudiado y entre sus técnicas de resolución se utilizan los algoritmos genéticos [38] [39] [40] [41].

Los algoritmos genéticos se basan en el principio Darwiniano de evolución de una población de individuos, en donde, la selección natural y la supervivencia de los más fuertes mejoran las siguientes generaciones.

El algoritmo genético diseñado comienza con una población de cromosomas (o individuos), donde cada uno de ellos representa un ciclo. Dichos cromosomas son sometidos a operadores de cruzamiento, mutación y selección que permiten generar nuevos individuos que formarán la siguiente generación. La selección apropiada de los operadores es fundamental para que las nuevas generaciones sean cada vez mejores (según una función de adaptación o fitness).

Es fundamental seleccionar una codificación de los cromosomas que represente adecuadamente nuestra solución al problema y facilite aplicar las operaciones definidas. En nuestro caso utilizamos la codificación más usada para el TSP que consiste en que cada cromosoma tenga tantos genes como el largo del ciclo, y en cada gen se ubique el nodo a visitar en el ciclo en el orden establecido. En la Figura 26 se presenta gráficamente un cromosoma que representa un ciclo formado por los siguientes nodos (1→ 3 → 2 → 5 → 4 → 1).

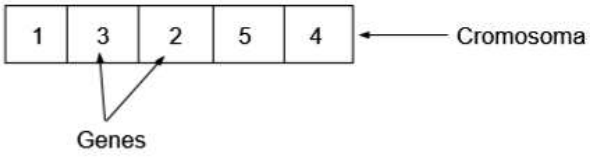


Figura 25: Cromosoma que representa el ciclo (1, 3, 2, 5, 4, 1)

La estructura de el algoritmo genético diseñado es la siguiente:

- **Inicialización**: Se genera una población inicial constituida por el ciclo original y n ciclos generados aleatoriamente. De esta forma aseguramos diversidad en la estructura de las soluciones de la población inicial, para evitar la convergencia prematura.

- **Evaluación**: A cada uno de los cromosomas de esta población se aplicará la función de aptitud o fitness que indicará qué tan "buena" es la solución. Los algoritmos genéticos son usados para problemas de maximización, por lo cual, para problemas de maximización la función de aptitud suele ser la misma que la función objetivo. Sin embargo, el TSP es un problema de minimización, por lo tanto la función objetivo se suele definir como:

$$\mathrm{F(x)} = \frac{1}{\mathrm{f(x)}}$$

en donde f(x) es el costo del ciclo representado por el cromosoma x.

- **Selección:** Una vez que se ha determinado la aptitud de cada individuo se seleccionan aquellos que serán cruzados. Existen diferentes técnicas de selección, en nuestro caso utilizamos la selección de ruleta, donde la probabilidad de seleccionar un individuo es proporcional a su fitness.

  Sea N el número de individuos existentes en la población y f_i el fitness del i-ésimo individuo, la probabilidad asociada a su selección está dada por:

$$p_i = \frac{f_i}{\sum_{j=1..N} f_j}$$

  Además se aplica el concepto de elitismo, donde los mejores n individuos siempre son seleccionados.

- **Cruzamiento:** El cruzamiento es el principal operador genético operando sobre dos cromosomas a la vez para generar dos descendientes. En la Figura 26 se presenta un ejemplo de cruzamiento que funciona de la siguiente manera:

  1. Selecciona aleatoriamente dos puntos de cruzamientos: Inicio y Fin
  2. En el primer hijo copia los cromosomas del primer padre para los genes entre Inicio y Fin
  3. En el segundo hijo copia los cromosomas del primer padre para los genes por fuera del rango Inicio y Fin
  4. Completa el primer hijo con los genes del segundo padre, en orden, que no heredó del primero
  5. Completa el segundo hijo con los genes del segundo padre, en orden, que no heredó del primero.

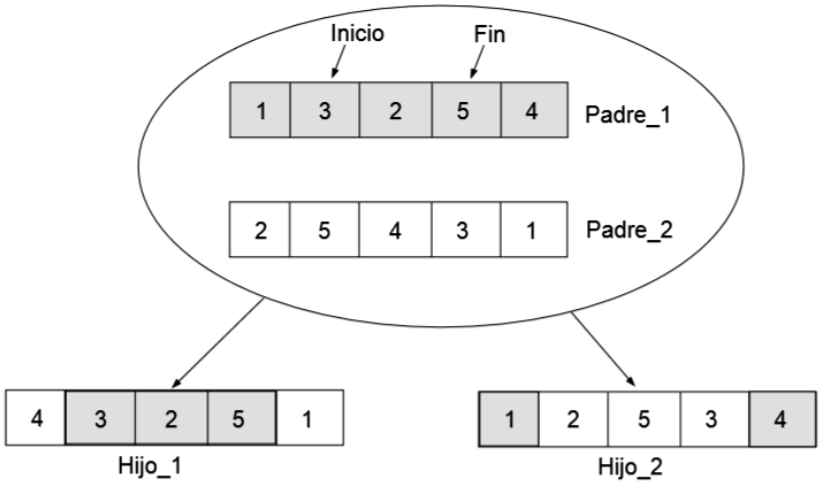


Figura 26: Operador de cruzamiento

- **Mutación:** Este operador tiene como objetivo explorar el espacio de soluciones, para esto, simplemente selecciona aleatoriamente dos genes del cromosoma y los intercambia. La probabilidad de que un cromosoma sufra una mutación debe ser baja para evitar la caminata aleatoria o "Random walk". En la Figura 27 se presenta gráficamente el resultado de aplicar el operador de mutación.

Figura 27: Operador de mutación

- **Reemplazo**: Una vez aplicados los operadores genéticos se seleccionan los mejores individuos para conformar la población de la generación siguiente.

El procedimiento anterior se repite durante una cantidad preestablecida de generaciones, retornando al final, el mejor individuo de la última generación.

4.6.3. Movimiento 2-opt para computar mejores ciclos

La algoritmo 2-opt fue propuesto por Croes en 1958 como un algoritmo simple para resolver el TSP, basado en la idea de modificar el orden en que se visitan los nodos para evitar cruzamientos. La Figura 28 presenta gráficamente esta idea.

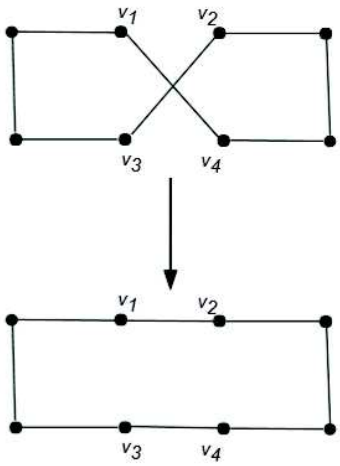


Figura 28: Swap de nodos v_1 con v_3

Algoritmo 18 – Algoritmo 2-opt

```
 1: procedure 2-opt
 2: input Ciclo
 3: begin
 4:   nuevoCiclo ← Ciclo
 5:
 6:    for (i=0; i<cantidadNodos (Ciclo); i++)
 7:       for (k=i+1; k<cantidadNodos (Ciclo); k++)
 8:          nuevoCiclo ← swapNodos(Ciclo, i, k)
 9:          if costo(nuevoCiclo) < costo(Ciclo)
10:               Ciclo ← nuevoCiclo
11:               return 2-opt(Ciclo)
12:          end if
13:       end for
14:   end for
15:   return Ciclo
16: end begin
```

El Algoritmo 18 presenta el pseudocódigo del procedimiento 2-opt, el cual recibe como parámetro el ciclo a optimizar. En la línea 4, se inicializa la variable $nuevoCiclo$ con el ciclo recibido por parámetro, el cual es utilizado para mantener las modificaciones a la solución.

El bucle de las líneas 6 a 14 itera por cada par de nodos i, k del ciclo, realizando el siguiente procedimiento. En la línea 8 intercambia la posición de los nodos i, k. En la línea 9 se compara el costo del nuevo ciclo con el costo del ciclo recibido por parámetro. Si el nuevo costo es menor, en la línea 10 se modifica la solución, mientras que en la línea 11 se vuelve a ejecutar el procedimiento 2-opt con el nuevo ciclo, hasta que no se encuentre un intercambio de nodos que produzca un mejor costo.

El procedimiento encargado de intercambiar la posición de un par de nodos i, k (línea 8) procede de la siguiente manera:

1. Cada nodo del ciclo original entre las posiciones $[0..i-1]$ lo agrega al nuevo ciclo en el mismo orden.
2. Cada nodo del ciclo original entre las posiciones $[i..k]$ lo agrega al nuevo ciclo en orden inverso
3. Cada nodo del ciclo original entre las posiciones $[k+1..cantidad\_nodos]$ lo agrega al nuevo ciclo respetando el orden.

4.7. Perturbación de la solución

Los algoritmos de "perturbación" o "shaking" tienen la finalidad de escapar de óptimos locales realizando modificaciones aleatorias a la mejor solución obtenida hasta el momento. En nuestro caso, un óptimo local es alcanzado cuando todas las búsquedas locales son realizadas sin encontrar mejora alguna, en cuyo caso, se procede a perturbar la mejor solución y se comienza nuevamente la fase de búsquedas locales.

El procedimiento de perturbación implementado toma al azar un número aleatorio de nodos de la componente 2-nodo conexa y los elimina, agregándolos posteriormente como nodos colgantes. La cantidad de nodos a eliminar debe asegurar que la componente 2-nodo conexa quede con al menos tres nodos. En caso de que no sea posible reasignar los nodos como colgantes, la solución no es factible, y se vuelve a iniciar el proceso de perturbación.

En la Figura 13, página 47, se puede ver que el procedimiento búsquedas locales y perturbación es ejecutado MAX_SHK veces, luego nuestra metaheurística GRASP vuelve a crear una solución inicial y a aplicar nuevamente las búsquedas locales tantas veces como se indique en el parámetro MAX_ITER. Luego que todas las iteraciones fueron realizadas, el procedimiento retorna la mejor solución obtenida entre todas las iteraciones.

Algoritmo 19 – Algoritmo de perturbación

```
 1: procedure shaking
 2: input Gsol
 3: begin
 4:   compConexa ← obtenerComponenteConexa(Gsol)
 5:   cantidadNodosEliminar ← random(0, cantidadNodos(compConexa)-3)
 6:   for (i=0; i< cantidadNodosEliminar; i++)
 7:       nodo ← seleccionarNodoAleatoriamente(compConexa)
 8:       compConexa ← eliminarNodo(compConexa, nodo)
 9:   end for
10:      Gsol ← reasignarColgantes(compConexa, Ginicial)
11:   return Gsol
12: end begin
```

El Algoritmo 19 presenta el pseudocódigo del algoritmo de perturbación, el cual recibe como parámetro de entrada la solución G_{sol} al 2NCSP.

En la línea 4 se inicializa la variable $compConexa$ con la componente 2-nodo conexa de G_{sol}, la cual será perturbada en el resto del procedimiento. En la línea 5 se elige aleatoriamente $cantNodosEliminar$, la cantidad de nodos a eliminar de $compConexa$.

El bucle de las líneas 6 a 9 se ejecuta $cantNodosEliminar$ veces, realizando el siguiente procedimiento. En la línea 7 selecciona el nodo a eliminar, y lo elimina de $compConexa$ en la línea 8.

Una vez que todos los nodos fueron quitado de la componente 2-nodo conexa original, se asignan como colgantes todos los nodos que no forman parte de $compConexa$ a algún nodo de la misma.

En la línea 11 se retorna la solución perturbada $\boldsymbol{G_{sol}}$.

CAPÍTULO 5

5. Estudio experimental

5.1. Introducción

Hasta donde sabemos, no existe en la literatura estudios sobre el 2NCSP, por lo que no contamos con fuentes de datos contra las cuales comparar los resultados de nuestro algoritmo.

Decidimos usar de referencia trabajos relacionados al RSP por tratarse de una relajación del problema 2NCSP, más específicamente utilizamos el estudio realizado por M. Labbé et al. en [1].

Además se utilizó el algoritmo desarrollado por G. Baya en [42], empleado para resolver el Capacitated m-Two Node Survivable Star Problem (CmTNSSP), el cual parametrizado correctamente, resuelve un problema topológicamente cercano al 2NCSP.

Las pruebas fueron realizadas en una máquina virtual con procesador Intel i7 de dos núcleos, 4 GB de memoria RAM y sistema operativo Windows XP.

Tanto las estructuras de datos, como los algoritmos de manipulación de grafos y el algoritmo evolutivo, fueron implementados íntegramente en este trabajo, sin hacer uso de ninguna librería o biblioteca que proporcione dichas capacidades.

Las búsquedas locales exactas fueron validadas utilizando el lenguaje de modelado AMPL y el solver CPLEX 12.5. Una vez validadas se utilizó la librería “IBM Concert Techonology” para la programación del modelo de programación lineal entera en C++ y para su resolución con el solver CPLEX.

5.2. Casos de prueba

En [1] M. Labbé et al. definen tres clases de casos de prueba para evaluar la performance de su algoritmo para resolver el RSP. La Clase I está basada en instancias de la TSPLIB 2.1 de 50 hasta 200 vértices, y será la que utilizaremos para evaluar nuestra metaheurística. A continuación describimos la forma en que el autor genera los casos de prueba.

Sea l_{ij} la distancia euclidiana entre los vértices v_i y v_j, se define el costo de conexión $c_{ij} = \lceil \alpha l_{ij} \rceil$ y el costo de asignación $d_{ij} = \lceil (10 - \alpha) l_{ij} \rceil$ con $\alpha \in \{3, 5, 7, 9\}$. De esta forma el autor pretende obtener soluciones óptimas en donde la solución 2-nodo conexa utilice aproximadamente el 100%, 75%, 50% y 25% de los vértices en cada instancia.

5.2.1. Parametrización de la metaheurística GRASP

Los parámetros utilizados para inicializar la metaheurística GRASP son los siguientes:

- *CANT_NODOS_INICIAL*: 10%. Este parámetro especifica la cantidad de nodos que debe contener la componente 2-nodo conexa de la solución factible inicial.

- *MAX_NODOS_CAMINO*: 10. Este parámetro indica la cantidad máxima de nodos que puede tener un camino para aplicarle la búsqueda local exacta "camino con nodos colgantes" presentada en la sección 4.5.

- *MAX_NODOS_CICLO*: 10. Este parámetro indica la cantidad máxima de nodos que puede tener un ciclo para que se pueda aplicar la búsqueda local exacta "mejor componente 2-nodo conexa" presentada en la sección 4.6.

- *MAX_SHK*: 30. Este parámetro indica la cantidad de veces que aplicará el procedimiento de perturbación presentado en la sección 4.7.

- *MAX_ITER*: 30. Este parámetro indica la cantidad de veces que se ejecutará el procedimiento GRASP.

5.3. Resultados

La Tabla 1 presenta los resultados de las soluciones para las instancias de Labbé et al. [1] de Clase I, comparándolas con los resultados obtenidos con nuestro algoritmo. Las notaciones correspondientes a cada una de las columnas son las siguientes:

- Instancia: Identificación de la instancia en la TSPLIB 2.1
- $\propto$: Factor de ponderación de los costos de conexión y asignación
- opt_{Labbe}: Valor objetivo encontrado en el trabajo de Labbé
- opt_{best}: Valor objetivo encontrado
- Gap: Diferencia porcentual de ambas soluciones calculada como:

$$gap = \frac{opt_{best} - opt_{Labbé}}{opt_{Best}} * 100$$

- $t(s)$: Tiempo del proceso en segundos.

$Instancia$	α	opt_{Labbe}	opt_{best}	$gap\%$	$t(hh:mm)$
eil51	3	1.278	1.286	0,626	0:46
eil51	5	1.995	2.010	0,752	0:25
eil51	7	2.113	2.123	0,473	0:13
eil51	**9**	**1.244**	**1.224**	**-1,608**	**0:07**
berlin52	3	22.626	22.633	0,031	0:31
berlin52	5	36.115	36.125	0,028	0:20
berlin52	**7**	**37.376**	**37.029**	**-0,928**	**0:10**
berlin52	**9**	**20.361**	**19.887**	**-2,328**	**0:05**
st70	3	2.025	2.031	0,296	1:12
st70	5	3.110	3.124	0,450	0:52
st70	7	3.402	3.412	0,294	0:45
st70	**9**	**2.610**	**2.513**	**-3,716**	**0:32**
eil76	3	1.614	1.638	1,487	1:16
eil76	5	2.460	2.496	1,463	0:54
eil76	7	2.504	2.544	1,597	0:47
eil76	**9**	**1.710**	**1.690**	**-1,170**	**0:37**
pr76	3	324.477	324.477	0,000	1:13
pr76	5	500.395	500.401	0,001	0:56
pr76	7	555.858	555.863	0,001	0:42
pr76	**9**	**424.359**	**378.396**	**-10,831**	**0:31**
rat99	3	3.633	3.663	0,826	>2:00
rat99	5	5.885	5.896	0,187	1:42
rat99	**7**	**6.436**	**6.301**	**-2,051**	1:30
rat99	**9**	**5.150**	**4.655**	**-9,612**	1:25
kroa100	3	63.846	63.856	0,016	>2:00
kroa100	5	100.785	100.787	0,002	1:42
kroa100	7	115.388	115.389	0,001	1:30
kroa100	9	94.265	94.360	0,101	1:25
krob100	3	66.423	66.529	0,254	>2:00
krob100	5	104.550	104.566	0,015	1:45
krob100	7	118.111	118.219	0,091	1:37
krob100	9	93.938	94.025	0,093	1:21

Tabla 1: Tabla comparativa para instancias Labbé Clase I

$Instancia$	α	opt_{Labbe}	opt_{best}	$gap\%$	$t(hh:mm)$
kroc100	3	62.247	62.556	0,496	>2:00
kroc100	5	99.065	99.069	0,004	1:42
kroc100	**7**	**113.533**	**112.764**	**-0,677**	**1:30**
kroc100	**9**	**92.894**	**91.451**	**-1,553**	**1:25**
krod100	3	63.882	64.126	0,382	>2:00
krod100	5	101.645	101.652	0,007	1:39
krod100	**7**	**116.849**	**116.715**	**-0,115**	**1:28**
krod100	**9**	**92.102**	**90.615**	**-1,615**	**1:25**
kroe100	3	66.204	66.319	0,174	>2:00
kroe100	5	104.915	105.122	0,197	1:50
kroe100	7	116.471	116.466	**-0,004**	1:36
kroe100	9	96.116	96.139	0,024	1:27
rd100	3	23.730	23.832	0,430	>2:00
rd100	5	37.975	37.983	0,021	1:46
rd100	**7**	**40.915**	**40.912**	**-0,007**	**1:40**
rd100	**9**	**31.776**	**31.317**	**-1,444**	**1:19**
eil101	3	1.887	1.927	2,120	**>2:00**
eil101	5	2.905	2.988	2,857	1:41
eil101	7	2.926	2.958	1,094	1:28
eil101	9	1.955	1.980	1,279	1:27
lin105	3	43.137	43.218	0,188	>2:00
lin105	**5**	**69.365**	**69.354**	**-0,016**	**>2:00**
lin105	7	83.597	84.190	0,709	>2:00
lin105	**9**	**96.920**	**63.277**	**-9,501**	**>2:00**
pr107	3	132.909	132.974	0,049	>2:00
pr107	**5**	**210.465**	**210.153**	**-0,148**	**>2:00**
pr107	**7**	**259.571**	**258.191**	**-0,532**	**>2:00**
pr107	**9**	**264.918**	**258.389**	**-2,465**	**>2:00**
pr124	3	177.090	177.092	0,001	>2:00
pr124	5	286.115	286.117	0,001	>2:00
pr124	7	385.853	385.851	**-0,001**	>2:00
pr124	**9**	**340.153**	**323.764**	**-4,818**	**>2:00**

Tabla 1: Tabla comparativa para instancias Labbé Clase I

$Instancia$	α	opt_{Labbe}	opt_{best}	$gap\%$	$t(hh:mm)$
bier127	3	354.846	359.342	1,267	>2:00
bier127	5	539.955	541.274	0,244	>2:00
bier127	7	567.110	568.260	0,203	>2:00
bier127	**9**	**347.845**	**343.893**	**-1,136**	**>2:00**
ch130	3	18.330	18.483	0,835	>2:00
ch130	**5**	**28.790**	**28679**	**-0,386**	**>2:00**
ch130	**7**	**32.707**	**32.499**	**-0,636**	**>2:00**
ch130	**9**	**23.639**	**23.657**	0,076	**>2:00**
pr136	3	290.316	292.440	0,732	>2:00
pr136	5	468.520	469.193	0,144	>2:00
pr136	**7**	**491.981**	**491.296**	**-0,139**	**>2:00**
pr136	**9**	**387.327**	**378.136**	**-2,373**	**>2:00**
pr144	3	175.611	176.231	0,353	>2:00
pr144	5	290.945	291.796	0,292	>2:00
pr144	**7**	**383.041**	**382.827**	**-0,056**	>2:00
pr144	**9**	**366.833**	**357.958**	**-2,419**	>2:00
ch150	3	19.584	20.015	2,201	>2:00
ch150	5	31.170	31.556	1,238	>2:00
ch150	7	34.930	34.950	0,057	>2:00
ch150	**9**	**26.371**	**25.777**	**-2,252**	**>2:00**
kroa150	3	79.572	80.180	0,764	>2:00
kroa150	5	125.435	126.897	1,166	>2:00
kroa150	7	140.961	141.168	0,147	>2:00
kroa150	**9**	**113.080**	**111.882**	**-1,059**	**>2:00**
krob150	3	78.390	79.124	0,936	>2:00
krob150	5	122.857	122.900	0,035	>2:00
krob150	7	135.382	135.412	0,022	>2:00
krob150	**9**	**108.885**	**107.925**	**-0,882**	**>2:00**
pr152	3	221.046	221.463	0,189	>2:00
pr152	**5**	**376.155**	**364.406**	**-3,123**	**>2:00**
pr152	**7**	**475.052**	**463.221**	**-2,490**	**>2:00**
pr152	**9**	**475.440**	**461.264**	**-2,982**	**>2:00**

Tabla 1: Tabla comparativa para instancias Labbé Clase I

$Instancia$	α	opt_{Labbe}	opt_{best}	$gap\%$	$t(hh:mm)$
rat195	3	6.969	7.245	3,960	>2:00
rat195	5	11.320	11.666	3,057	>2:00
rat195	7	12.319	12.430	0,901	>2:00
rat195	**9**	**9.395**	**8.585**	**-8,622**	**>2:00**
kroa200	**3**	**93.699**	**90.999**	**-2,882**	**>2:00**
kroa200	5	138.885	140.278	1,003	>2:00
kroa200	7	158.277	159.485	0,795	>2:00
kroa200	**9**	**124.678**	**122.809**	**-1,499**	**>2:00**
krob200	3	88.311	89.664	1,532	>2:00
krob200	5	138.905	140.252	0,970	>2:00
krob200	7	156.638	158.099	0,933	>2:00
krob200	**9**	**127.800**	**124.086**	**-2,906**	**>2:00**

Tabla 1: Tabla comparativa para instancias Labbé Clase I

Las Figuras 29 a 32 presentan la solución obtenida por nuestro algoritmo, para la instancia pr107, con $\propto = 3$, $\propto = 5$, $\propto = 7$ y $\propto = 9$ respectivamente. En este grafo en particular se puede apreciar claramente el comportamiento de las soluciones en función de $\propto$, cuanto mayor su valor, menor es el costo de asignación por lo cual, es más conveniente colgar nodos, que hacerlos parte de la componente 2-nodo-conexa.

Figura 29: Solución al problema pr107 con $\propto = 3$

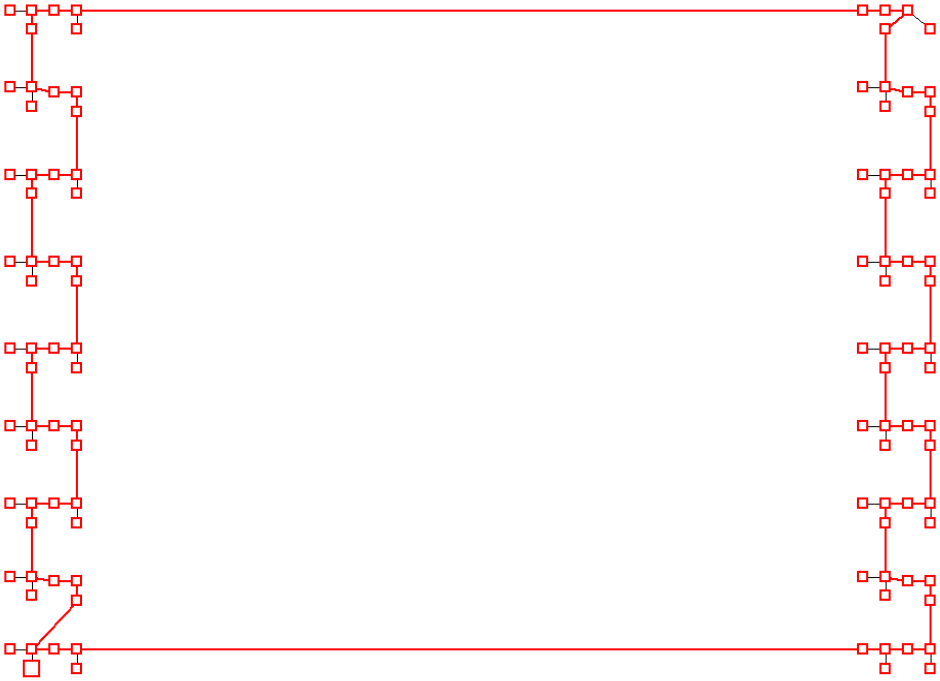


Figura 30: Solución al problema pr107 con ∝= 5

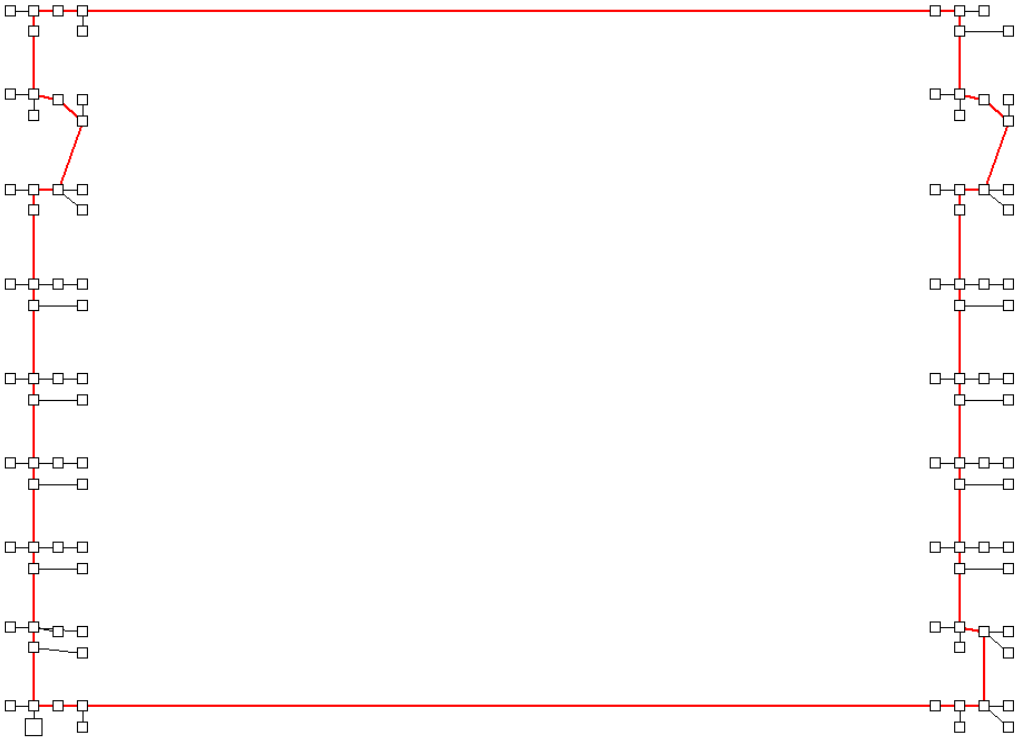


Figura 31: Solución al problema pr107 con ∝= 7

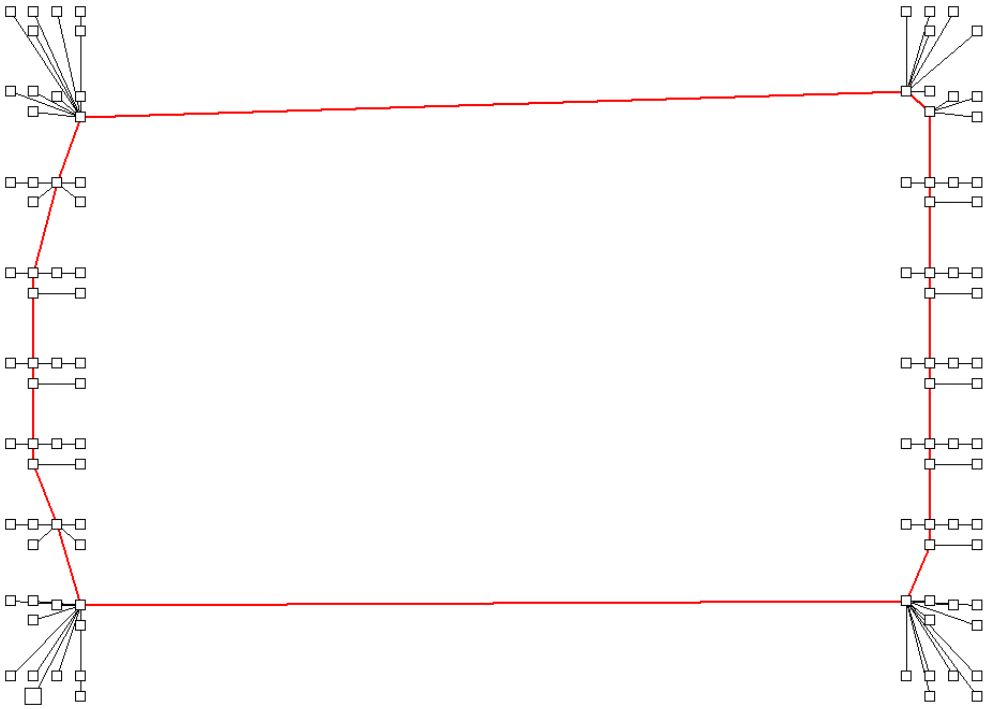


Figura 32: Solución al problema pr107 con ∝= 9

En la Figura 33 se presenta la solución al problema kroa200, por ser uno de los problemas con más nodos y cuyo resultado mejoró la performance de Labbé para ∝= 3.

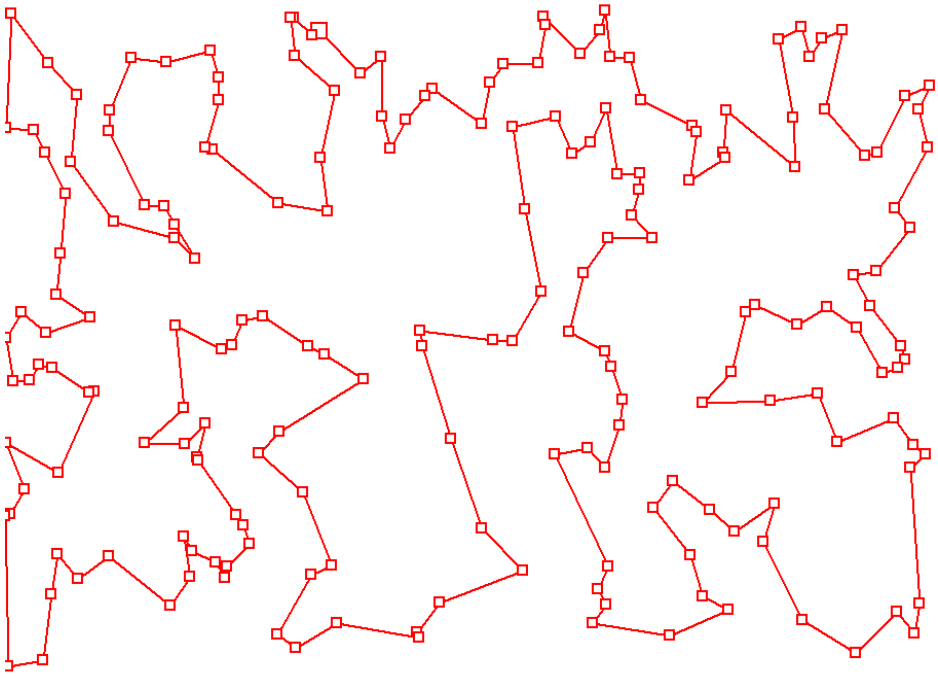


Figura 33: Solución al problema kroA200 con ∝ = 3

En la Figura 34 se presenta la solución obtenida para la instancia lin105 con $\propto = 9$, por ser una de las instancias que presentó mejores resultados comparativos con el algoritmo de M. Labbé et al.

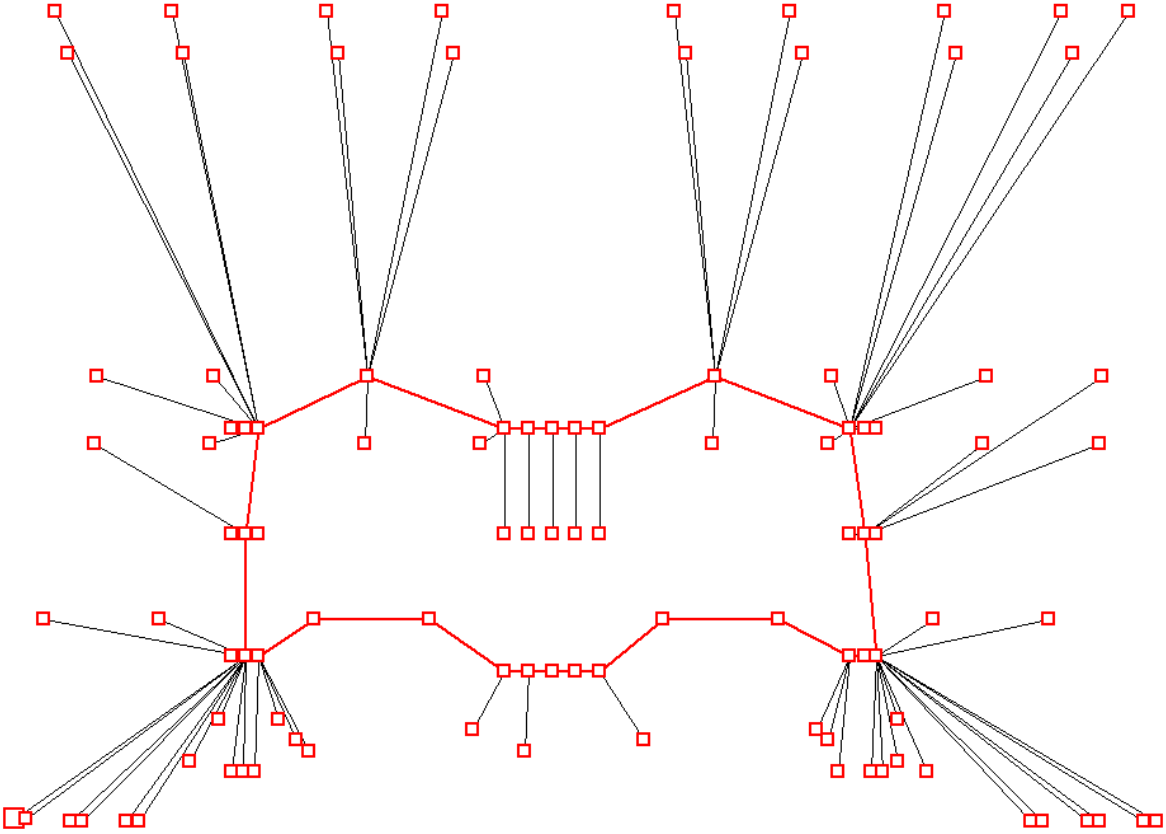


Figura 34: Solución al problema lin105 con ∝ = 9

En la Tabla 2 se describen los resultados para las instancias eil51 a eil76 de la TSPLIB en comparación con los resultados obtenidos mediante el algoritmo de G. Baya parametrizado para resolver el CmTNSSP [42] (Capacitated m Two-Node-Survivable Star Problem) de la siguiente manera:

- Cantidad de componentes 2-nodo conexa: $m = 1$
- Capacidad de cada componente 2-nodo conexa: $Q =$ cantidad de nodos del grafo
- El depósito se selecciona como el nodo más cercano al centro de masa.

$Instancia$	α	opt_{Baya}	opt_{best}	$gap\%$	$t(hh:mm)$
eil51	**3**	**1.290**	**1.286**	**-0,31**	**0:46**
eil51	**5**	**2.020**	**2.010**	**-0,50**	**0:25**
eil51	**7**	**2.252**	**2.123**	**-5,73**	**0:13**
eil51	**9**	**2.003**	**1.224**	**-38.89**	**0:07**
berlin52	3	22.632	22.633	0,00	0:31
berlin52	**5**	**36.444**	**36.125**	**-0,88**	**0:20**
berlin52	**7**	**40.684**	**37.029**	**-8,98**	**0:10**
berlin52	**9**	**33.456**	**19.887**	**-40,56**	**0:05**
st70	**3**	**2.066**	**2.031**	**-1,69**	**1:12**
st70	**5**	**3.158**	**3.124**	**-1,08**	**0:52**
st70	**7**	**3.565**	**3.412**	**-4,29**	**0:45**
st70	**9**	**3.517**	**2.513**	**-28,55**	**0:32**
eil76	**3**	**1.690**	**1.638**	**-3,08**	**1:16**
eil76	**5**	**2.576**	**2.496**	**-3,11**	**0:54**
eil76	**7**	**2.774**	**2.544**	**-8,29**	**0:47**
eil76	**9**	**2.585**	**1.690**	**-34,62**	**0:37**

Tabla 2: Tabla comparativa con algoritmo CmTNSSP

La Figura 35 presenta la solución al 2NCSP obtenida por nuestra metaheurística para la instancia eil76 con $\propto = 7$.

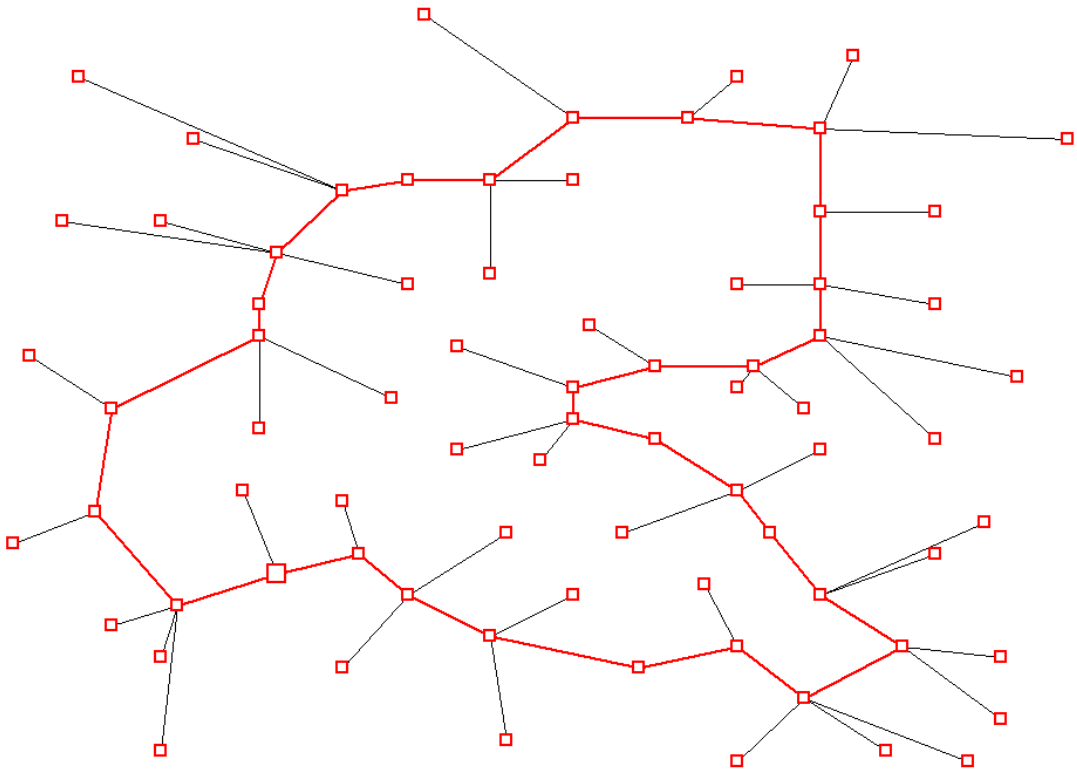


Figura 35: Solución al problema lin105 con ∝ = 7

5.4. Topologías de las componentes 2-nodo-conexa

Todas las soluciones obtenidas para las instancias de prueba detalladas anteriormente han dado como resultados componentes 2-nodo conexas con topología de anillo. Lo cual motivó la creación de una instancia de prueba que permita verificar que la metaheurística implementada es capaz de generar soluciones con topología 2-nodo conexas no cíclicas.

Se diseñó entonces el grafo $G = (V, E)$ completo con $V = \{0, 1, 2, 3, 4\}$, donde los costos de conexión y asignación de sus aristas es un número entero positivo elevado, exceptuando los siguientes costos conexión:

$$c_{0,1} = c_{0,3} = c_{0,4} = 1$$
$$c_{1,0} = c_{1,2} = 1$$
$$c_{2,1} = c_{2,4} = c_{2,3} = 1$$
$$c_{3,0} = c_{3,2} = 1$$
$$c_{4,0} = c_{4,2} = 1$$

El grafo fue diseñado de forma tal que su óptimo global consista en una componente 2-nodo conexa no cíclica, pudiéndose comprobar que nuestra metaheurística GRASP fue capaz de encontrar el óptimo global, como se puede ver en la Figura 36.

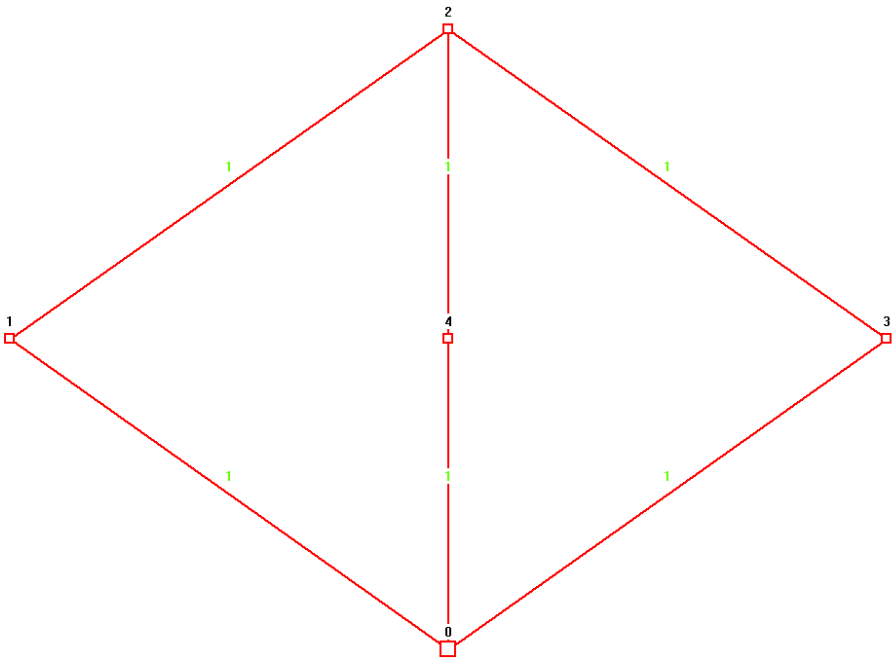


Figura 36: Solución con componente 2-nodo conexa no cíclica

CAPÍTULO 6

6. Conclusiones

En este trabajo se estudió el problema 2NCSP del cual no se han encontrado referencias en la literatura reciente, el cual definimos naturalmente y en lenguaje de grafos ponderados con costos de conexión y asignación entre sus nodos.

Para resolver el problema se diseñó una metaheurística GRASP/VNS descendente y una serie de búsquedas locales que permitieron encontrar soluciones de buena calidad. Dentro de las búsquedas locales destacamos aquellas que hacen uso de un método exacto, por ser las más complejas y las que más colaboraron en la mejora de las soluciones.

Los casos de prueba realizados hacen uso de un factor $\propto$ de ponderación de los costos de conexión y asignación, el cual determina que la componente 2-nodo conexa tenga más o menos nodos. Se observa que para $\propto = 3$, las soluciones tienden a incorporar todos los nodos en la componente 2-nodo-conexa, mientras que para $\propto = 9$, la componente 2-nodo conexa tiende a tener la menor cantidad de nodos posible, asignando el resto a alguno de los nodos de la misma.

A pesar del comportamiento variado que debe tener el algoritmo en función de los costos de asignación y conexión, se observa que en todos los casos se desenvuelve correctamente, fundamentalmente para valores de $\propto$ elevados. En la Tabla 3 se presenta los gaps promedios con la solución de Labbé et al. [1].

α	$gap_{promedio}$
3	0.64
5	0.39
7	-0.01
9	-2.78

Tabla 3: gaps promedios comparados con M. Labbé

En todos los casos el algoritmo encuentra soluciones con topología Ring Star, no se ha logrado conseguir para ninguno de estos casos, una componente 2-nodo conexa que no sea un anillo.

Al tener esta topología, para $\propto = 3$, los resultados obtenidos están muy cerca del óptimo, inclusive en algunos casos, se obtiene el mismo, por ejemplo en las instancias: berlin52,

pr76, kroa100 y pr124 e inclusive, en un caso si bien no se obtiene el óptimo, se consigue una solución mejor que la obtenida por Labbé (kroa200).

El algoritmo se destaca para $\propto = 9$, en donde se encuentran mejores resultados en la mayoría de las instancias probadas.

En comparación con los resultados obtenidos por G. Baya, nuestra metaheurística consigue mejores resultados en el 94% de los casos de prueba. Para $\propto=9$, la diferencia notoria se debe a que para poder hacer uso de la metaheurística de G. Baya, se tiene que seleccionar un nodo como punto de partida de la componente 2-nodo conexa. Una mala selección de dicho nodo puede ser determinante en el resultado de la metaheurística, por tratarse de soluciones con componente 2-nodo conexa muy costosas. Pero para valores de $\propto = 3$ o 5, la comparación tiene más sentido, siendo los gaps promedios los presentados en la Tabla 4.

α	$gap_{promedio}$
3	-1.27
5	-1.40

Tabla 4: gaps promedios comparados con G. Bayá

Si bien en esta caso se hicieron pruebas con instancias de hasta 76 nodos, es posible ver que nuestro algoritmo se comporta bastante mejor al aumentar la cantidad de nodo de la instancia de prueba.

Más allá de las conclusiones obtenidas a partir de la comparación de resultados con trabajos de características similares, es importante resumir algunas conclusiones obtenidas a partir de las pruebas realizadas.

En primer lugar se concluye que la forma de construir la solución inicial, si bien impacta en el resultado de la metaheurística, no es determinante en su performance. En consecuencia, lo más importante es que sea factible y se construya con poco costo computacional.

También es importante destacar que la secuencia de aplicación de búsquedas locales impacta notoriamente en las soluciones finales. En nuestro caso, en primera instancia se ejecutaba cada búsqueda local mientras esta retornaba mejoras, y luego se seguía con la próxima. Luego se modificó por el esquema descendente, comprobando que las nuevas soluciones mejoraban ampliamente las anteriores.

Se concluye además que las búsquedas locales basadas en métodos exactos aportan muy buenos resultados, lamentablemente no es posible aplicarlas para problemas con más de

15 nodos porque los tiempos se enlentecen excesivamente. La variante utilizando algoritmos genéticos e intercambios 2-opt, fueron diseñados analizando la bondad del método exacto y buscando una alternativa para ciclos con mayor cantidad de nodos.

Si bien el algoritmo genético ofrece buenas soluciones, no son tanto mejores a las obtenidas por el algoritmo simple 2-opt, siendo el segundo mucho más rápido, por lo cual se optó por este último para la realización de las pruebas.

Finalmente es importante destacar el rol significativo del algoritmo de perturbación, dado que en la gran mayoría de las instancias de prueba, el hecho de perturbar la solución y comenzar nuevamente con las búsquedas locales generó soluciones de mejor costo, comprobando así que a pesar de la diversidad de búsquedas aplicadas, siempre terminan convergiendo en un óptimo local.

6.1. Trabajos futuros

Uno de los principales motivos por los cuales se eligió GRASP es la simplicidad con la cual se pueden agregar nuevas búsquedas locales. Creemos que diseñar nuevas búsquedas locales es un camino a recorrer para mejorar las soluciones ofrecidas hasta el momento, además se podría profundizar en la búsqueda de algoritmos alternativos a los métodos exactos que permitan trabajar con grafos de mayor porte sin penalizar los tiempos de ejecución.

Resulta de particular interés estudiar y aplicar a nuestro algoritmo GRASP la técnica de post optimización “Path Re-Linking”, ya que varios autores han probado su eficacia.

Por último, creemos interesante estudiar la posibilidad de paralelizar algunas búsquedas locales, lo cual mejoraría ampliamente los tiempos de ejecución.

Si bien, tanto en la fase de construcción como de búsquedas locales se realizan todas las validaciones posibles para asegurar que las soluciones sean factibles, sería de gran utilidad la implementación de un algoritmo que permita validar la topología de las soluciones.

Bibliografía

[1] M. Labbé, G. Laporte, I, Rodríguez Martín y J. J. Salazar González. The Ring Star Problem: Polyhedral Analysis and Exact Algothim. *Networks*, 43(3), 177–189, 2004.

[2] G. Zorpette. Keeping the Phone Lines Open. Spectrum, IEEE, 26(6):32–36, 1989. ISSN 0018-9235.

[3] Klincewicz. Nomenclatura. 1998.

[4] G. Reinelt. TSPLIB - A traveling salesman problem library. Universität Heidelberg, Institut für Informatik. URL: comopt.ifi.uni-heidelberg.de/software/TSPLIB95.

[5] M. B. Richey. Optimal location of a path or tree on a network with cycles. Networks. 20(4): 391-407, 1990.

[6] S. Khuller, A. Zhu. The General Steiner Tree-Star Problem. University of Maryland, 2002.

[7] J. Xu, S. chiu, G. Glover. Using Tabu search to solve de Steiner tree-star problem in telecommunication. Telecomm. 6: 117-125, 1996.

[9] M. Labbé, G. Laporte, I. Rodríguez Martín y J. J. Salazar González. The median cycle problem. *Université de Montreál*, 1999.

[10] J. Renaud, F. Boctor, Fayez, G. Laporte. Efficient Heuristics for Median Cycle Problems. *Journal of the Operational Research Society*. 55: 179–186, 2004.

[11] J. A. Moreno Pérez, J. M. Moreno-Vega, I.Rodríguez Martín. Variable Neighborhood Tabu Search and its Application to the Median Cycle Problem. *Universidad de Laguna, Dpto de Estadística, I.O. y Computación*. 2003.

[13] S. Dias. An Efficient Heuristic for the Ring Star Problem. S*pringer-Verlag Heidelberg. Lecture in Computer Scince*. 4007: 24-35, 2006

[14] S. Lin, B. W. Kernighan. An Effective Heuristic Algorithm for the Traveling-Salesman Problem. *Opererations Research.* 21: 498-516, 1973.

[15] A. Liefooghe, L. Jourdan, M. Basseur, E. Talbi, E. K. Burke. Metaheuristics for the Bi-objective Ring Star Problem. *Lecture Notes in Computer Science*. 4972: 206-217, 2008.

[16] M. Basseur, E.K. Burke. Indicator-based multi-objective local search. *Evolutionary Computation*, *In CEC'2007*, 3100 - 3107, 2007.

[17] E. Zitzler, S. Künzli. Indicator-based selection in multiobjective search. *Lecture Notes in Computer Science*. 3242: 832-842, 2004.

[18] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan. A fast and elitist multiobjective genetic algorithm. *IEEE Transactions on evolutionary computation*. 6(2): 182 - 197, 2002.

[19] S. Kedad-Sidhoum, V. H. Nguyen. An exact algorithm for solving the ring star problem. *Optimization: Journal of Mathematical Programming and Operations Reseach*. 59(1): 125-140, 2010.

[20] H. I. Calvete, C. Galé, J. A. Irzano. An efficient evolutinary algorithm for the Ring Star Problem. *European Journal of Operational Research*. 231: 22-33, 2013.

[21] R. Baldacci, M. Dell'Amico, J. J. Salazar González. The capacitated m-ring star problem. *Operations Research*. 55(6): 1147–1162, 2007.

[22] Z. Naji-Azimi, P. Toth, M. Salari. A heuristic procedure for the Capacitated m-Ring-Star problem. *European Journal of Operational Research*. 207: 1227-1234, 2013.

[23] Z. Naji-Azimi, P. Toth, M. Salari. An integer linear programming based heuristic for the Capacitated m-Ring Star Problem. *European Journal of Operational Research*. 217(1): 17–25, 2012.

[24] Z. Zhang, H. Qin, A. Lim. A memetic algorithm for the Capacitated m-Ring Star Problem. *Appl. Intell.*. 40(2): 305–321, 2014.

[25] R. Baldacci, M. Dell´Amico, J. Salazar. The Capacitated m-Ring-Star problem. *Operations Research*. 55: 1147-1162, 2007.

[26] E. A. Hoshino, C. C. de Souza. Column Generation Algorithms for the Capacitated m-Ring-Star Problem. *Lecture Notes In Computer Science*, págs. 5092: 631 - 641, 2008.

[27] E. A. Hoshino. A Branch-and-Cut-and-Price Approach for the Capacitated m-Ring Star Problem. *Electronic Notes in Discrete Mathematics*. 35: 103-108, 2009.

[28] A. Mautone, S. Nesmachnow, A. Olivera, F. Robledo. Solving a Ring Star Problem Generalization. *Computational Intelligence for Modelling Control & Automation*. 981 - 986, 2008.

[29] F. Glover. Future Paths for integer programming and links to artificial inteligence. *Computers and Operations Research.* 5: 533-549, 1986.

[30] E. Talbi. Metaheuristics: From Design to Implementation. *Wiley Series on Parallel and Distributed Computing*. 2009.

[31] B. Melian, J. A. Moreno Pérez, J. M. Moreno Vega. Metaheuristics: A global view. *Revista Iberoamericana de Inteligencia Artificial.* 19: 7-28, 2003.

[32] C. S. Ferreira, L. S. Ochi, V. Parada, E. Uchoa. A GRASP-based approach to the generalized minimum spanning tree problem. *Expert Systems with Applications*. 39: 3526–3536, 2012.

[33] G. Kontoravdis, J. F. Bard. A GRASP for the Vehicle Routing Problem with Time Windows. *Journal on Computing*. 7(1), 1995.

[34] E. F. Gouvêa Goldbarg, M. C. Goldbarg, J. P.F. Farias. Grasp with Path-Relinking for the Tsp. *Operations Research, Computer Science Interfaces Series*. 39: 137-152, 2007.

[35] T. A. Feo, M. G. C. Resende. Greedy Randomized Adaptative Search Procedures. *Journal of Global Optimization.* 6: 109 - 133, 1995.

[36] E. Talbi. Hybrid Metaheuristics. *Studies in Computational Intelligence*. 8(5): 514-564, 2012.

[37] N. Mladenovic, P. Hansen. Variable neighborhood search. *Computers & Operations Research.* 24(11):1097 – 1100, 1997.

[38] P. Hansen, N. Mladenovic, J. A. Moreno Pérez. Busqueda de Entorno Variable. R*evista Iberoamericana de Inteligencia Artificial*. 19: 77 - 92, 2003.

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

[40] K. Bryant. Genetic Algorithms and the Traveling Salesman Problem. *Harvey Mudd College.* 2000.

[41] Z. H. Ahmed. Genetic Algorithm for the Traveling Salesman Problem using Sequential Constructive Crossover Operator*. International Journal of Biometrics & Bioinformatics*. 3(6): 96-105.

[42] Chiung Moon, Jongsoo Kim, Gyunghyun Choi, Yoonho Seo. An efficient genetic algorithm for the traveling salesman problem with precedence constraints. *European Journal of Operational Research.* 140: 606–617, 2002.

[43] J. Potvin. Genetic algorithms for the traveling salesman problem. *Annals of Operations Research.* 63: 339-370, 1996.

[44] G. Baya. Msc. Thesis, Capacitated m Two-Node-Survivable Star Problem. *INCO*. 2010.

[45] H. Calvete, C. Galé, J. A. Iranzo. An efficient evolutionary algorithm for the ring star problem. *European Journal of Operational Research*., Vol. 231 (1): 22-33, 2013.

[46] M. Stoer. Design of Survivable Networks. *Springer-Verlag*, 1992. ISBN 3-540-56721-0.

[47] S. Dias, G. F. de Sousa Filho, E. M. Macambir, L. dos Anjos, F. Cabral, M.H. a C. Fampa. An Efficient Heuristic for the Ring Star Problem. *Lecture Notes in Computer Science*. 4007: 24-35, 2006.

[50] Z. Naji-Azimi, M.Salari, P. Toth. A Variable Neighborhood Search and its Application to a Ring Star Problem Generalization. *Electronic Notes in Discrete Mathematics*. 36: 343-350, 2010.

[51] M. Labbé, G. Laporte, I. R. Rodríguez Martín, J. J. Salazar. Locating Median Cycles in Networks. *European Journas of Operational Research*. 160: 457-470, 2005.

[52] A. Liefooghe, L. Jourdan, E. Talbi. Metaheuristics and Their Hybridization to Solve the Bi-objective Ring Star Problem: a Comparative Study. *Computers & Operations Research.* 37: 1033-1044. 2010:

[53] J. M. Pérez, M. Moreno Vega y I. R. Martín. Variable neighbourhood tabu search and its application to the median cycle problem. *European Journal of Operational Research*. 151:365–378, 2003.

[54] J. M. Pérez, M. Moreno-Vega, M. Rodriguez. Variable Neighborhood Tabu Search and its Applicatoin to the Median Cycle Problem. *European Journal of Operational Research*. 151: 365-378, 2003.

[55] C.L Monma, B.S. Munson and W.R. Pulleyblank, “Minimum-Weight Two-Connected Spanning Networks”, Mathematical Programming, 46:153-171, 1990.