Universidad de la República
Facultad de Ingeniería
Proyecto de grado – Ingeniería en Computación

Diseño de Topologías de Red Confiables

Departamento de Investigación Operativa
Instituto de Computación
2006

Sebastián Laborde
Sebastián Ressi
Alvaro Rivoir

Tutores:
Dr. Ing. Franco Robledo
MSc. Ing. Omar Viera

Resumen

En una red las entidades relevantes son nodos y conexiones entre nodos, y en general el principal objetivo buscado es lograr una comunicación segura entre nodos de esta, ya sea para redes telefónicas y de comunicación de datos, de transporte, arquitectura de computadores, redes de energía eléctrica o sistemas de comando y control.
La optimización relativa al costo de una red y la confiabilidad de la misma, relacionada con la supervivencia de esta, son los criterios predominantes en la selección de una solución para la mayor parte de los contextos. A pesar que por años el costo ha sido el factor primario la confiabilidad ha ganado rápidamente en relevancia. Con sistemas de transmisión de fibra óptica de alta capacidad formando los *backbones* de la mayoría de las redes actuales la supervivencia del tráfico por sobre los fallos de red se ha convertido aún en más crítica.
En ese sentido podemos diferenciar, a grandes rasgos, dos de los principales problemas a resolver en el análisis y diseño de topologías de red. Primeramente la obtención de una red óptima en algún sentido, siendo este definido por ejemplo mediante la obtención de la máxima cantidad posible de caminos distintos entre pares de nodos, esto sujeto a determinadas restricciones definidas según el contexto. El segundo problema es la evaluación de la confiabilidad de la red en el sentido de la comunicación entre nodos que la componen. Ambos problemas están fuertemente relacionados, pudiendo tener que comparar en el proceso de búsqueda de redes óptimas la confiabilidad entre soluciones candidatas, o luego de obtener una solución óptima tener que evaluar la confiabilidad de la misma y de esta forma descartarla o no.
El presente trabajo se centra en la resolución del problema enfocado en ambos puntos planteados. Para ello modelamos el problema de diseño de la topología de red sobre la base del GSP-NC (*Generalized Steiner Problem with Node-Connectivity Constraints*).
El presente problema es NP-duro. Nuestro objetivo es atacar de forma aproximada el modelo GSP-NC de tal modo de poder resolver la optimización de la red y luego medir la confiabilidad de la solución obtenida. Para ello optamos por desarrollar la metaherística VNS (*Variable Neighbourhood Search*). VNS es un método potente que combina el uso de búsquedas locales basadas en distintas definiciones de vecindad, el cual ha sido utilizado para obtener soluciones de buena calidad en distintos problemas de optimización combinatoria.
En lo referente al cálculo de confiabilidad de la red desarrollamos RVR (*Recursive Variance Reduction*) como método de simulación, ya que la evaluación exacta de esta medida para redes de tamaño considerable es impracticable.
Las pruebas experimentales fueron realizadas utilizando un conjunto amplio de casos de prueba, de heterogéneas topologías con diferentes características, incluyendo instancias de más de 200 nodos. Los resultados obtenidos indican tiempos de cómputo altamente aceptables acompañados de óptimos locales de buena calidad.

Palabras clave: diseño topológico, confiabilidad, optimización, backbone, simulación, RVR, metaheurística, VNS.

Contenido

1. Introducción

1.1.Contexto Marco

Este trabajo se realiza en el marco del proyecto de grado de la carrera "Ingeniero en Computación" de la Facultad de Ingeniería (Fing), Universidad de la República (UdeLaR). Tiene como cometido ejecutar una actividad creadora en Ingeniería en Computación para el departamento de Investigación Operativa del Inco[1], específicamente para su área de investigación "Diseño topológico de redes resistentes a fallas".
El proyecto presentado se centra dentro de un contexto enmarcado en un proyecto más general de planificación de redes de comunicaciones modernas. Esta tarea compleja y en general costosa tiene como eje principal la optimización, e integra como elemento primordial dentro del bucle del proceso de optimización actividades de análisis y evaluación cuantitativa. Mediante el planteo de este trabajo de grado se desea desarrollar una actividad de investigación que abarque el enfoque del diseño de una red de telecomunicaciones de alto porte. Dada la experiencia que existe en el área, se cree por parte de nuestros tutores de vital importancia disponer de propuestas de solución al problema especificado, el cuál es de altísima utilidad en el diseño de una red de fibra óptica.

1.2.Motivación

La revolución de la información conmocionó el siglo XX y XXI, siendo esta una de las revoluciones más relevantes en el transcurso de la historia. Mayor aún fue el impacto del intercambio de información, posibilitando este contemplar cooperación y convergencia tanto de tecnologías como de personas.
En un comienzo fueron utilizadas las redes[2] telefónicas para satisfacer la necesidad de la comunicación de datos. Hoy día esto no es así y se diseñan las redes de datos considerando desde un comienzo el objetivo del contexto para la cual están siendo construidas, son más aptas desde su construcción para integrar servicios realizando en ellas las adaptaciones necesarias teniendo en cuenta la naturaleza de los servicios, necesidades y tráfico que cursan.
Las redes de datos permiten que las tecnologías surgidas en el transcurso del siglo pasado y lo que va del presente estén siendo integradas en un proceso constante hacia un ente común de comunicación. La interconexión de dichas redes está permitiendo la centralización, concentración y almacenamiento de información dispersa en los distintos continentes, con el resultado natural de agilizar el trabajo, negocios, comercio y diversas situaciones u actividades cotidianas.
La tarea de diseñar dichas redes, considerando la integración de diversos tipos de tráfico y servicios entre otros factores, no es una tarea sencilla en absoluto sino todo lo contrario, la misma resulta compleja siendo tanto el diseño, el dimensionar la red como la optimización[3] de ella tareas de difícil resolución. A la complejidad antes mencionada se suma el lograr que la topología de red[4] escogida supere los parámetros de confiabilidad[5] deseados para el contexto para el cual está siendo construida.
En la construcción de la red son considerados un gran número de sitios, los cuales poseen características disímiles, siendo necesario que la interconexión de estos supere o al menos satisfaga la cota de confiabilidad establecida a un costo mínimo.
El objetivo de cualquier diseño topológico[6] es ajustarse en lo posible a las necesidades tecnológicas del contexto y a la disponibilidad económica del proyecto, teniendo en cuenta un

[1] Instituto de Computación, UdeLaR.

[2] Red: se puede considerar como un conjunto de nodos y conjunto de enlaces que los conectan entre sí.

[3] Optimización: Consiste en tomar una decisión óptima para maximizar o minimizar un criterio determinado.

[4] Topología de red: Disposición física en la que se conectan los nodos de una red.

[5] Confiabilidad: Es la probabilidad que un sistema opere en forma correcta bajo condiciones dadas durante un determinado período de tiempo.

[6] Diseño topológico: Es una etapa en el proceso de planificación de redes la cual consiste en determinar donde situar geográficamente los componentes de una red y como conectarlos entre sí.

sin fin de factores que afectan claramente el mismo, yendo desde el costo propio de los distintos elementos que componen la red a la calidad de servicio esperada.
En el presente trabajo atacaremos el diseño de topologías de red confiables[7], integrando diferentes iteraciones de optimización con evaluaciones cuantitativas que permitan determinar si el parámetro deseado de confiabilidad ha sido alcanzado.

1.3.Introducción al problema

Durante la primera etapa del proyecto se realizó un relevamiento de requerimientos en el cual participaron los autores y los tutores de este trabajo y como resultado de la misma se definió el problema a resolver en los siguientes dos puntos:

- Dada una red de la cuál se conocen los costos de conexión entre pares de nodos, diseñar una red con el menor costo posible que satisfaga las restricciones de conectividad y confiabilidad predeterminadas (datos del problema).
- Evaluar (comparación en términos de costo) la calidad de los resultados obtenidos.

En este contexto es de particular interés disponer de métodos capaces de diseñar topologías de red que satisfagan ciertos requerimientos de conexión entre sus nodos (i.e. Dos caminos nodo disjuntos entre pares de nodos) y paralelamente la confiabilidad de la red (medida en términos probabilísticos) sea superior a cierto umbral preestablecido (dato del problema).
El modelo a resolver implica un modelo mixto de los conceptos confiabilidad estructural y supervivencia topológica[8] de una red, se debe realizar un relevamiento bibliográfico del material relacionado y en una segunda fase con la guía de los tutores proponer posibles soluciones al problema. En una tercera fase implementar la metodología seleccionada y en la cuarta fase realizar pruebas experimentales y medir cualitativamente la calidad de las soluciones obtenidas con la metodología diseñada, si es posible, estimar que "tan buenas" son las soluciones devueltas.

1.4.Objetivos

Objetivo general
El objetivo general del proyecto consiste en desarrollar una heurística[9] que diseñe la topología de una red dada (grafo asociado) de forma de satisfacer ciertos requerimientos de conexión entre pares de nodos (datos del problema) y además esta red resultante satisfaga un nivel de confiabilidad preestablecido (dato del problema).

Objetivos específicos
Como objetivos específicos se destaca introducir a los estudiantes autores de este trabajo en los conceptos de:

- Confiabilidad Estructural (la probabilidad de que dado que ocurren fallas en la red está se mantenga operativa i.e. conexa).
- Supervivencia topológica (cumplimiento de ciertos niveles de conectividad en la red).

Formarse en el área de planificación y confiabilidad en redes. Particularmente en el diseño topológico de redes de alto porte de uso estratégico en aplicaciones de importancia crítica. Adquirir conocimiento en herramientas de planificación de redes y aprender a diseñar e implementar algoritmos eficientes para la resolución de problemas NP-duros[10], como lo es el problema planteado.

[7] Confiabilidad Estructural: Es la probabilidad de que dado ocurren fallas en la red está se mantenga operativa.

[8] Supervivencia topológica: Es el cumplimiento de ciertos niveles de conectividad en la red.

[9] Heurística: Trata de aquellos métodos o algoritmos exploratorios para la resolución de problemas, en los que las soluciones se descubren por la evaluación del progreso logrado en la búsqueda de un resultado final.

[10] NP-duro: Son problemas que no se pueden resolver en tiempo polinomial en función del tamaño de la entrada.

1.5.Resultados Esperados

El resultado esperado a grandes rasgos es disponer de un método tentativo que sirva como base para la toma de decisiones a la hora de la planificación de redes de telecomunicaciones. Típicamente, aplicable al diseño del Backbone[11] de una red Wan[12].Para poder cumplir con lo planteado se debe cumplir los siguientes ítems:

- Comprensión del modelo matemático asociado al problema a resolver.
- Relevamiento bibliográfico del material relacionado.
- Familiarización por parte de los estudiantes autores de este trabajo en los conceptos de:
    - Supervivencia topológica de una red.
    - Confiabilidad estructural de una red.
- Investigar el planteo de alternativas para obtener una solución aproximada al problema planteado.
- Implementación de dicha alternativa.
- Medir la calidad de los resultados obtenidos.

1.6.Método de solución

Previo a plantearse atacar el problema se procedió a realizar una investigación y familiarización con las notaciones, definiciones y temas de estudio involucrados directamente al objetivo del proyecto. Este trabajo se resume y refleja en el *Informe de contexto del problema* del proyecto (Apéndice 1), informe desarrollado como testimonio del proceso de aprendizaje y preparación para poder desarrollar una solución al problema de *"Diseño de topologías de red confiables"*.
Hasta donde sabemos y podemos concluir, la información disponible que abarcara e hiciera foco sobre el tema de estudio a tratar es escasa y prácticamente nula, teniendo si como contexto general disponibilidad de información suficiente sobre los problemas de confiabilidad por un lado y por otro de optimización de una red, de forma independiente.
La primera etapa del proyecto se centra en comprender el problema a resolver y poder plantear una definición formal del mismo, investigando y familiarizándose con ambas problemáticas (dado el contexto general del tema de estudio) de forma independiente una de otra.
De esta forma en lo que refiere al problema de optimización se investigaron y evaluaron diversas metaheurísticas[13] potencialmente utilizables en la resolución del problema objetivo, entre las cuales se encuentran GRASP 101010101010 y su variante modificada para GSP-NC 1010, Algoritmos Genéticos 1010, TABU SEARCH 1010, VNS 10101010 e ILS 1010.
De la misma forma para el problema de confiabilidad se abordaron los métodos de simulación[14] Monte Carlo Crudo 10 y RVR 101010. Debido a que nuestro problema es de clase NP-Duro un algoritmo exacto para su resolución sería impracticable por lo que se decidió utilizar una metaheurístca que sirviera como método de aproximación a la solución optima del problema a resolver, la elección de una metaheurístca nos brinda la posibilidad de contar con un método potente y extensible pudiendo fácilmente generar otros métodos híbridos con heurísticas especificas que se ajusten el problema. Una vez analizadas y comprendidas una variedad de metaheurísticas potencialmente aplicables a la resolución del problema de optimización (primera fase de la construcción de nuestra topología de red confiable), la decisión tomada fue utilizar para ello VNS (Variable Neighbourhood Search) 10101010. VNS 10101010 está basado en un principio claro y relativamente no explorado, el cambio sistemático de la estructura de entornos (vecindades[15]) durante la búsqueda. La precisión en la forma de realizar dichos cambios es crucial. VNS 10101010 ha mostrado eficiencia en muchos experimentos, obteniendo iguales o mejores resultados que la mayoría de las metaheurísticas en muchos problemas de optimización combinatoria, y el hecho que según nuestro conocimiento aún no haya sido aplicado en el problema específico que nos toca resolver hace aun mas atractiva esta elección.

[11] BackBone: Es el esqueleto o troncal principal de una red.

[12] Wan: Wide area network.

[13] Metaheurística: Una metaheurística es un método heurístico para resolver una clase muy general de problemas computacionales combinando procedimientos heurísticos de forma eficiente.

[14] Simulación: Es la experimentación con un modelo de una hipótesis de trabajo.

[15] Vecindad: Una vecindad de un punto x es éste punto con algo de su entorno.

Para la parte de confiabilidad surge el estudio de métodos de reducción de varianza para la simulación de medidas de confiabilidad y dentro de ellos hemos estudiado en particular RVR 101010, método de simulación seleccionado para ser utilizado en este proyecto. La opción tomada se fundamenta en que a pesar de no ser tan sencilla su implementación ha sido probado10 que con RVR 1010 los resultados obtenidos son de mayor precisión que para Monte Carlo Crudo 10, siendo un algoritmo adecuado para el cálculo de confiabilidad en redes muy confiables de cualquier tamaño.

1.7.Conclusiones

En este proyecto se estudia el diseño topológico de redes confiables atacando dos subproblemas debidamente identificados, el problema de optimización de la red y el problema de satisfacer la cota de confiabilidad establecida para la misma. La optimización de la red fue atacada en forma aproximada con una heurística, ya que se trata de un problema NP-Duro 10 y por ende la aplicabilidad de algoritmos exactos es prohibitiva en lo que respecta a tiempo computacional, aún para redes de pequeño y mediano tamaño. Por ello optamos por utilizar la metaheurística VNS (*Variable Neighbourhood Search*) basándonos en las razones que se expondrán en la sección 3.1 *"Selección de la Metaheurística"*. En lo que refiere al cálculo de confiabilidad de la red, dado que la evaluación exacta de la medida para redes de tamaño considerable es impracticable, el mismo fue abordado mediante el método de simulación RVR (*Recursive Variance Reduction*), decisión justificada en la Sección 3.2 *"Selección del método de Calculo de Confiabilidad"*.
No se dispone de casos benchmark[16] para poder medir la calidad de los resultados en base a la comparación de los mismos, no obstante los resultados obtenidos indican tiempos de cómputo altamente aceptables acompañados de óptimos locales de buena calidad medidos en términos de mejora de costos.
Es relevante destacar que la documentación formal que recabó información referente a investigación o experimentación sobre el problema objetivo de esta tesis, hasta donde pudimos relevar es escasa y prácticamente nula, por lo menos a nivel académico.

1.8. Estructura del Informe

Este informe presenta las actividades realizadas, resultados obtenidos y conclusiones extraídas en el marco de este proyecto[17].
El *Capitulo 1* da una introducción a este trabajo, contiene la motivación del proyecto, el contexto marco, la especificación del problema a resolver, los objetivos del proyecto, el método de resolución utilizado, los resultados esperados y las conclusiones. El *Capitulo 2* explica todos los conceptos relevantes en este proyecto para ser comprendido en su completitud. El *Capitulo 3* explica formalmente el problema, se realiza una revisión del contexto del problema, se explican y justifican los métodos de resolución utilizados y se definen formalmente los conceptos del problema a resolver. El Capitulo 4 explica a fondo los algoritmos para la resolución del problema planteado. El Capitulo 5 explica las consideraciones de diseño utilizado en la solución y se justifican las decisiones de implementación que fueron tomadas. El Capitulo 6 explica las pruebas de validación del modelo realizadas. El Capitulo 7 contiene las pruebas experimentales y se exponen y analizan cuantitativamente los resultados obtenidos. En el *Capitulo 8* se establecen las conclusiones de los resultados experimentales obtenidos. El Capitulo 9 propone trabajos futuros y posibles líneas de investigación que extiendan y/o complementen este trabajo. El Capitulo 10 contiene las fuentes de la bibliografía utilizada. Con respecto a los Apéndices en el Apéndice 1 se adjunta el documento "Informe de Contexto del problema". El Apéndice 2 y Apéndice 3 explican la conservación de factibilidad de las búsquedas locales utilizadas y del algoritmo de construcción respectivamente. El Apéndice 4 presenta el algoritmo de construcción de grafos de prueba utilizados para generar datos para las pruebas funcionales. El Apéndice 5 explica la estructura de la información de un grafo y la aplicación de visualización de grafos implementada. El Apéndice 6 trata los mecanismos de

[16] Benchmark: El benchmark es una técnica utilizada para medir el rendimiento de un sistema o parte de un sistema, frecuentemente en comparación con algún parámetro de referencia.

[17] En cursiva capítulos de necesaria lectura para comprender el trabajo.

extensibilidad de la solución desarrollada. En el Apéndice 7 se presentan los resultados finales obtenidos. El Apéndice 8 incluye el cronograma del proyecto y finalmente el Apéndice 9 contiene el índice de figuras del presente documento.

2. Contexto

En este Capitulo se definen todos los conceptos 10 que son relevantes en este proyecto para ser comprendido en su completitud y serán utilizados en los próximos capítulos.

2.1.Conceptos

Red: Una red será considerada en este trabajo como un conjunto de nodos y un conjunto de enlaces que los conectan entre sí (Grafo).

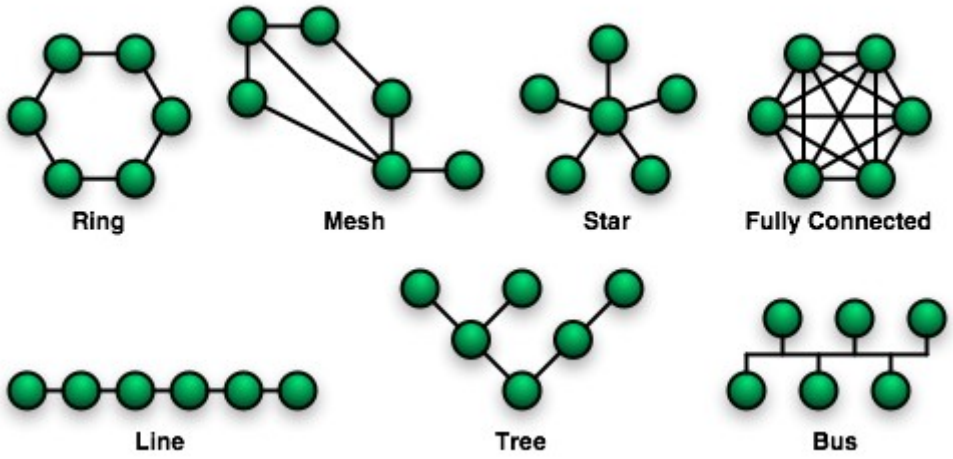
Grafo: Es un conjunto de objetos llamados vértices o nodos unidos por enlaces llamados aristas. Las aristas pueden tener dirección.

Backbone: Es el esqueleto o troncal principal de una red. Una misma red puede tener más de un troncal (i.e. internet).

Confiabilidad (Reliability): es la probabilidad que un sistema opere en forma correcta bajo condiciones dadas durante un determinado período de tiempo.

Confiabilidad estructural de una red: es la probabilidad de que dado que ocurren fallas en la red esta se mantenga operativa (i.e. conexa). La confiabilidad se calcula como la probabilidad de funcionamiento de la red en base a las confiabilidades elementales de cada componente, dadas mediante probabilidades de funcionamiento de los mismos.

Topología de red: es un conjunto formado por los enlaces que conectan pares de nodos de la red. Es determinada únicamente por la configuración de las conexiones entre los nodos. (i.e. Ring, Mesh, Star, Fully Connected, Line, Tree, Bus).



Diferentes topologías de red.

Diseño topológico: Es una etapa en el proceso de planificación de redes la cual consiste en determinar donde situar geográficamente los componentes de una red y como conectarlos entre sí.

Supervivencia (Survivability): Es la habilidad cuantificada de un sistema, subsistema, equipo, proceso o procedimiento de continuar en funcionamiento durante y luego de una alteración.

Supervivencia topológica: es el cumplimiento de ciertos niveles de conectividad en la red. Se refiere a la existencia entre cualquier par de nodos de un número preestablecido de caminos que no tienen nodo o enlace en común.

Heurística: La heurística trata de aquellos métodos o algoritmos exploratorios para la resolución de problemas, en los que las soluciones se descubren por la evaluación del progreso logrado en la búsqueda de un resultado final. Se trata de métodos en los que, aunque la exploración se realiza de manera algorítmica, el progreso se logra por la evaluación puramente empírica del resultado. Se gana eficacia, sobre todo en términos de eficiencia computacional, a costa de la precisión. Las técnicas heurísticas son usadas por ejemplo en problemas en los que la complejidad de la solución algorítmica disponible es función exponencial de algún parámetro; cuando el valor de este crece el problema se vuelve rápidamente inabordable. Una alternativa heurística será practicable, si la complejidad de cómputo depende por ejemplo polinómicamente del mismo parámetro. Las técnicas heurísticas no aseguran soluciones óptimas, sino solamente soluciones válidas aproximadas; y frecuentemente no es posible justificar en términos estrictamente lógicos la validez del resultado.

Metaheurística: Una metaheurística es un método heurístico para resolver una clase muy general de problemas computacionales combinando procedimientos de usuario de caja negra –generalmente heurísticos- de forma eficiente.
En otras palabras, una metaheurística puede ser vista como un algoritmo *framework* genérico, el cual podría ser aplicado a diferentes problemas de optimización con relativamente pocas modificaciones para adaptarlos a uno especifico.
Las metaheurísticas son generalmente aplicadas a problemas para los cuales no hay algoritmos específicos o heurísticas satisfactorias; o cuando no es práctico implementar un método de ese tipo. Las metaheurísticas mas comúnmente usadas tienen como objetivo el resolver problemas de optimización combinatoria.

Optimización: Consiste en tomar una decisión óptima para maximizar (ganancias, velocidad, eficiencia, etc.) o minimizar (costos, tiempo, riesgo, error, etc.) un criterio determinado. Las restricciones significan que no cualquier decisión es posible. En ciencias de la computación es el proceso de mejorar un sistema en cierto sentido para reducir el tiempo de cómputo, el ancho de banda, los requerimientos memoria, etc. No necesariamente significa encontrar la solución óptima a un problema sino una aproximación a ella.

Optimización combinatoria: Es una rama de la optimización en matemáticas aplicadas y ciencias de la computación, relacionada a la investigación operativa, teoría de algoritmos y teoría de la complejidad computacional la cual se apoya en la intersección de varios campos, incluyendo inteligencia artificial, matemáticas e ingeniería de software. Los algoritmos de optimización combinatoria resuelven instancias de problemas que se cree son en general duros, mediante la exploración del usualmente basto espacio de soluciones de dichas instancias. Los algoritmos de optimización combinatoria logran esto mediante la reducción del tamaño efectivo del espacio, y mediante la exploración eficiente del mismo.

Optimo local: El óptimo local de un problema de optimización combinatoria es una solución óptima dentro de un conjunto de soluciones vecinas.

Optimo global: Un óptimo global es una selección de un dominio dado, la cual obtiene los valores más bajos o bien los mas altos (dependiendo del objetivo) cuando una función especifica es aplicada. Es la solución óptima a todo el espacio de soluciones.

Clase NP: es el conjunto de problemas de decisión que pueden ser resueltos en tiempo polinómico por una máquina de Turing no determinista. La importancia de esta clase de problemas de decisión es que contiene muchos problemas de búsqueda y de optimización para los que se desea saber si existe una cierta solución o si existe una mejor solución que las conocidas.

NP-Duro: es el conjunto de los problemas de decisión que contiene los problemas H tales que todo problema L en NP puede ser transformado polinomialmente en H. Esta clase puede ser descrita como conteniendo los problemas de decisión que son al menos tan difíciles como un problema de NP. Esta afirmación se justifica porque si podemos encontrar un algoritmo A que

resuelve uno de los problemas H de NP-duro en tiempo polinómico, entonces es posible construir un algoritmo que trabaje en tiempo polinómico para cualquier problema de NP ejecutando primero la reducción de este problema en H y luego ejecutando el algoritmo A.
Son problemas que no se pueden resolver en tiempo polinomial en función del tamaño de la entrada.

Simulación: es la experimentación con un modelo de una hipótesis de trabajo.

Algoritmo Greedy: Tienen como principal objetivo construir una solución válida que se aproxime a la óptima en el menor tiempo posible. En la mayoría de los casos no encuentran la solución óptima pero si una buena aproximación a ella.

Vecindad: Una vecindad de un punto x es éste punto con algo de su entorno. Tenemos entera libertad para definir el significado de "entorno" y "vecindad" con tal de satisfacer los axiomas siguientes:

- x pertenece a todas sus vecindades.
- Un conjunto que contiene una vecindad de x, es una vecindad de x.
- La intersección de dos vecindades de x es también una vecindad de x.
- En toda vecindad V de x existe otra vecindad U de x tal que V es una vecindad de todos los puntos de U.

2.2.Definiciones básicas

A continuación se presentan una serie de definiciones básicas 10 relativas a la teoría de grafos y otras utilizadas frecuentemente en trabajos relacionados con modelos de redes confiables, que serán utilizadas a lo largo del presente trabajo.

Extremos de Arista: Ambos nodos incidentes a una arista son sus extremos y por lo tanto una arista los conecta. Una arista $\{u,v\}$ puede ser expresada como (u,v) o (v,u).

Nodo Adyacente: Dado un grafo $G=(V,E)$, sea $u \in V$ uno nodo del mismo. Un nodo $v \in V$ es adyacente a u en G si la arista $(u,v) \in E$.

Grado de un Nodo: El grado $d(v)$ de un nodo v es el número $|E(v)|$ de aristas incidentes a v. Un nodo de grado 0 está aislado.

Grafo Inducido: Dado un grafo $G=(V,E)$, si $U \subseteq V$ es un subconjunto de nodos entonces $G(U)$ denota el grafo en U cuyas aristas son aquellas aristas de G cuyos extremos están contenidos en U.

Camino: Un camino es un grafo no vacío $P=(V,E)$ con la siguiente forma:

$$V=\{v_1,v_2,v_3,...,v_k\}, E=\{(v_1,v_2),(v_2,v_3),...,(v_{k-1},v_k)\}$$

Los nodos v_1 y v_k están conectados por P y se los llama sus extremos, el resto de los nodos son nodos internos de P.

Camino nodo disjunto: Sean p y q dos caminos de $G=(V,E)$. Ambos son nodos disjuntos entre sí, si se cumple que $p \cap q = \{v_1,v_2\}$ siendo v_1 y v_2 extremos de p y q. Esta definición es generalizable para N caminos.

Caminos nodo-disjuntos (con los mismos extremos): Dados dos caminos p_1, p_2 incluidos en un grafo $G=(V,E)$ con los mismos extremos $u,v \in V$ decimos que p_1, p_2 son nodo-disjuntos si la intersección de sus conjuntos de nodos internos es vacía.

Ciclo: Dado un camino $P=(v_1 \ldots v_k)$ el grafo C obtenido de concatenar P con $v_k v_1$ es considerado un ciclo.

Caminos Independientes: Decimos que dos caminos p_1, p_2 incluidos en un grafo $G=(V,E)$ son independientes si la intersección de sus conjuntos de nodos es vacía.

Subgrafo: Dado un grafo $G=(V,E)$, $\mathrm{H}=(V',E')$ es un subgrafo de G si $V' \subseteq V$, $E' \subseteq E$ y $\forall (u,v) \in E'$, tenemos $u,v \in V'$.

Grafo Conexo: Un grafo $G=(V,E)$ es conexo si para todo par de nodos $u,v \in V$ existe un camino que una u con v en G.

Árbol: Un grafo $G=(V,E)$ es un árbol si para toda arista $e \in E$ $G'=(V,E \setminus \{e\})$ no es conexo.

Árbol de Expansión: Dado un grafo conexo $G=(V,E)$, un subgrafo $\mathrm{H}=(V,E')$ es un árbol de expansión si H es conexo y para toda arista $e \in E'$, $H'=(V,E' \setminus \{e\})$ es disconexo.

K-Nodo Conectividad: Un grafo $G=(V,E)$ está k-nodo conectado si para $\forall u,v \in V$ existen al menos k caminos nodo-disjuntos en G conectándolos.

Nodos Terminales: Sea T un conjunto de nodos tal que $\forall t \in T$, t esta incluido en el backbone de la red, a estos nodos los llamaremos terminales o fijos, generalmente se corresponden con puntos de acceso de las subredes locales.

Matriz de Requerimientos de Conexión: $R=\left\{r_{ij}\right\}_{i,j \in T}$ es una matriz de enteros de requerimientos de conexión entre pares de nodos $\in E$. Requeriremos r_{ij} caminos nodo-disjuntos entre los nodos terminales i y j, donde usualmente r_{ij} es estrictamente mayor que uno.

Backbone Network Design Problem (BNDP): Es el problema de encontrar un subgrafo H_B de G_B de mínimo costo tal que los nodos de H_B incluyen los nodos del conjunto T de nodos terminales y H_B satisface los requerimientos de conexión especificados en la matriz R.

Key-nodo: Supongamos g_{sol} una solución factible que satisface la matriz de requerimientos de conexión R. Un key-nodo es un nodo no terminal, con grado de por lo menos tres en g_{sol}.

Key-path: Supongamos g_{sol} una solución factible que satisface la matriz de requerimientos de conexión R. Definimos un key-path como un camino en g_{sol} tal que todos sus nodos internos son no terminales con grado dos en g_{sol} y cuyos nodos extremos son nodos terminales o key-nodos.

Key-tree: Supongamos g_{sol} una solución factible que satisface la matriz de requerimientos de conexión R y un key-nodo $v \in g_{sol}$. Definimos un key-tree asociado a v como un árbol de g_{sol} el cuál esta conformado por todos los key-paths qué tienen a v como uno de sus extremos. Topológicamente, podemos verlo como un conjunto de caminos que tienen a v como extremo en común (key-nodo v).

3. Definición del problema

En el pasado reciente, en el diseño de redes tradicionales de cable de cobre la redundancia y supervivencia[18] no eran consideradas cuestiones de relevancia. Esto se debía básicamente a que la limitada capacidad de los cables de cobre hacía que existiesen diversas rutas entre, por ejemplo, distintas oficinas de telefonía (comúnmente denominadas gateway[19]). En algunos casos esto llevaba a que redes de comunicaciones que no fueron planeadas con la suficiente redundancia no pudiesen sobrevivir a una simple interrupción en uno de sus cables, o a la falla de una de sus oficinas. La llegada de la fibra óptica y su alta capacidad trajo aparejado topologías de red mas dispersas. Como consecuencia de ello el diseño de la red cobra mayor relevancia y se realiza con más detenimiento y cuidado, debiendo estas mantenerse operativas ante un posible corte de un cable o la destrucción de una de sus oficinas. Cuando hablamos de redes de telefonía nos es de interés únicamente su topología, en este caso vemos una red como un conjunto de nodos u oficinas y conexiones de fibra óptica que los interconectan.

En este contexto la supervivencia significa que entre cualquier par de oficinas telefónicas relevantes existe un número de caminos preestablecidos (consistiendo estos de nodos y enlaces) los cuales no tienen nodos en común.

En la práctica primero se crea una topología de red con bajo nivel de costo, para luego optimizar la misma, etapa en donde los costos son considerados (llámese costos de ruteo o de tráfico).

Continuando con el ejemplo de una red de telefonía el problema de supervivencia puede ser descrito informalmente de la siguiente forma:

Nos son dados un número específico de oficinas telefónicas y sus locaciones que tienen que ser conectadas por una red. Las oficinas podrían ser clasificadas según su importancia de la siguiente forma:

- Oficinas especiales o **terminales**, para las cuales un alto grado de supervivencia debe ser asegurado.
- Oficinas ordinarias, las cuales deben estar simplemente conectadas a la red.
- Oficinas opcionales, las cuales pueden o no ser parte de la red.

También son conocidos pares de oficinas entre las cuales puede ser establecido un enlace directo, estando asociado a cada posible enlace el costo del mismo. El problema ahora se resume en seleccionar que enlaces de fibra deben ser establecidos de forma tal que la suma de estos costos es minimizada y ciertas condiciones de supervivencia son aseguradas, tal que:

- La destrucción de cualquier enlace simple no desconecte dos terminales.
- La destrucción de cualquier oficina no desconecte dos oficinas especiales o terminales.

Esto es equivalente a requerir que entre dos terminales deben existir por lo menos dos caminos que no tengan ninguna oficina o nodo en común.

Un mayor refinamiento podría establecer que entre algunos pares de terminales existiesen tres o mas caminos aumentando así su supervivencia frente a la posible destrucción o corte de más de un enlace u oficina.

Este ejemplo fácilmente generalizable para otros contextos de características similares resume las bases de la primera fase del problema objetivo de la presente tesis, siendo este la construcción y optimización de la red en cuestión, asegurando ciertas condiciones de supervivencia. Para la creación de una topología de red con bajo nivel de costo es necesario contar con un algoritmo de construcción que proporcione un grafo de partida sobre el cual posteriormente se realizará la búsqueda de una aproximación al óptimo global del problema. Para resolver el problema de optimización se utilizará una metaheurística cuya selección será tratada en las secciones subsiguientes.

Una metaheurística es un conjunto de conceptos que pueden ser usados para definir métodos heurísticos los cuales pueden ser aplicados a un amplio conjunto de problemas. En otras palabras, una metaheurística puede ser vista como un algoritmo *framework*[20] genérico, el cual

[18] Supervivencia: Es la habilidad cuantificada de un sistema, subsistema, equipo, proceso o procedimiento de continuar en funcionamiento durante y luego de una alteración.

[19] Gateway: En una red de comunicaciones un gateway es un nodo de red equipado para interactuar con otra red que utiliza diferentes protocolos.

[20] Framework: En el desarrollo de software un framework es una estructura de soporte definida en la cual otro proyecto de software puede ser organizado y desarrollado.

podría ser aplicado a diferentes problemas de optimización con relativamente pocas modificaciones para adaptarlos a uno especifico.
Retomando el ejemplo de la red de telefonía asumamos que a cada componente de la red, nodos y enlaces, tenemos asociada su confiabilidad elemental, la cual es dato. En base a dichas confiabilidades es que podemos calcular el índice de confiabilidad de la topología de red resultante de la primera fase del problema.
El objetivo de la segunda fase es justamente que la topología de red construida supere un umbral de confiabilidad de funcionamiento establecido por el usuario o cliente. Para ello es necesario seleccionar un algoritmo que nos permita definir la medida o índice de confiabilidad funcional, selección que es planteada y fundamentada en las secciones subsiguientes.
La solución final del problema resuelve ambas fases descritas mediante la integración de diferentes iteraciones de optimización con evaluaciones cuantitativas que permitan determinar si el parámetro deseado de confiabilidad ha sido alcanzado.

3.1.Selección de la Metaheurística

Una vez analizadas y comprendidas una variedad de metaheurísticas potencialmente aplicables a la resolución del problema de optimización (primera fase de la construcción de nuestra topología de red confiable), la decisión tomada fue utilizar para ello VNS (*Variable Neighbourhood Search*).
La pregunta es "*porque VNS?*". Es claro que la decisión no pasa únicamente por analizar si existe o no éxito en la aplicación de una metaheurística particular a un problema especifico, o si hay controversia en la aplicabilidad de esta a un contexto dado. Basar la decisión únicamente en el criterio mencionado no solo acota enormemente la decisión sino que hace sumamente difícil la misma ya que muchas veces se carece de la información necesaria para realizar una correcta comparación.
El primer paso es considerar globalmente las cualidades deseables de una metaheurística y corroborar que estas se cumplan (ver 11.1 *Informe de contexto del problema*). En ese sentido VNS está basado en un principio claro y relativamente no explorado, el cambio sistemático de la estructura de entornos durante la búsqueda. La precisión en la forma de realizar dichos cambios es crucial. Ha mostrado eficiencia en muchos experimentos, obteniendo iguales o mejores resultados que la mayoría de las metaheurísticas en muchos problemas y de una forma mucho más rápida. La efectividad de esta ha sido probada en la resolución eficaz de problemas de varios bancos de prueba con resultados óptimos o muy cercanos a los óptimos, y con tiempo computacional moderado o al menos razonables.
Basándonos en lo anterior la decisión fue tomada teniendo en cuenta además dos factores que nos inclinaron hacia esta metaheurística. Primeramente se trata de una metaheurística reciente presentada hace poco mas de una década (1995) para la resolución de problemas de optimización, que a pesar de haber tenido un rápido desarrollo la investigación y trabajos basados en ella es escaso en comparación al resto. Esto, a pesar de agregar un riesgo al proyecto, nos permite realizar un trabajo que se diferencie de los ya abordados en proyectos anteriores, ya que hemos podido comprobar la existencia de tesis que han tratado problemas de optimización ya sea con GRASP 10, Algoritmos Genéticos 10, u otras metaheurísticas.
La segunda es la posibilidad de manejar extensiones a la propia metaheurística 10, o el incorporar VNS a otra metaheurística obteniendo así un hibrido 1010, lo cual si bien no forma parte del objetivo central de este proyecto otorga flexibilidad en la búsqueda de una solución y podría ser considerado como un trabajo a futuro de interés.

3.2.Selección del método de cálculo de confiabilidad

Como parte imprescindible del proyecto debemos poder definir una medida o índice de confiabilidad para la red, esto basado en su topología y en la confiabilidad de sus componentes. Se debe calcular la probabilidad de funcionamiento de la red en base a las confiabilidades elementales de cada componente, obteniéndose estas de las probabilidades de funcionamiento de los mismos. Existen algoritmos exactos eficientes para ciertos tipos particulares de redes, pero para redes relativamente grandes dichos algoritmos se hacen impracticables debido a sus excesivos tiempos de ejecución. Una solución posible al problema

planteado es utilizar métodos de simulación, y dentro de estos hemos estudiado particularmente dos, Monte Carlo Crudo 10 y RVR 10.
El método Monte Carlo, ya sea estándar o crudo, es una opción simple en lo que refiere a implementación como forma de simular el comportamiento estocástico del sistema. La mayor desventaja de este método es la excesiva varianza de las estimaciones y como consecuencia de ello el aumento del tamaño del intervalo de confianza de la estimación, trayendo esto aparejado que los resultados obtenidos sean poco precisos. Esta situación se agrava y es notoria particularmente en redes de alta confiabilidad, donde se hace necesario realizar un número excesivo de replicaciones para obtener estados no operativos o de falla. Dada esta problemática surge el estudio de métodos de reducción de varianza para la simulación de medidas de confiabilidad y dentro de ellos hemos estudiado en particular RVR 101010, método de simulación seleccionado para ser utilizado en este proyecto. La opción tomada se fundamenta en que a pesar de no ser tan sencilla su implementación ha sido probado 10 que con RVR 1010 los resultados obtenidos son de mayor precisión que para Monte Carlo Crudo 10, siendo un algoritmo adecuado para el cálculo de confiabilidad en redes muy confiables de cualquier tamaño.

3.3.Formalización del Problema

Como fue planteado previamente en este documento, la primera etapa del proyecto se centra en comprender el problema a resolver y poder plantear una definición formal del mismo. Como resultado de la investigación realizada, la familiarización con la problemática y la colaboración y el aporte del tutor Dr. Ing. Franco Robledo, se obtiene la formalización del problema que se presenta a continuación.
Primeramente definimos la notación necesaria para la formalización del problema:

- $C = \left\{ c_{ij} \right\}_{(i,j) \in E}$ es la matriz que para cualquier par de nodos dados pertenecientes al grafo $G = (V, E)$ retorna el costo (no negativo) de establecer una conexión entre ellos.
- $R = \left\{ r_{ij} \right\}_{i,j \in T}$ es una matriz de enteros de requerimientos de conexión entre pares de nodos $\in E$. Requeriremos r_{ij} caminos nodo-disjuntos entre los nodos terminales i y j, donde usualmente r_{ij} es estrictamente mayor que uno.

Declaradas las notaciones formalizamos el planteo del problema objeto de este trabajo de la siguiente forma:

Dado un grafo simple no dirigido $G = (V, E)$, un sub-conjunto de nodos distinguidos $T \subseteq V$ (denominados terminales), una matriz de costos $C = \left\{ c_{ij} \right\}_{(i,j) \in E}$ asociados a las aristas de G y una matriz de requerimientos de conexión entre pares de nodos terminales $R = \left\{ r_{ij} \right\}_{i,j \in T}$. Supongamos además que cada arista de G tiene asociada cierta probabilidad de operación, i.e. tenemos: $P_E = \left\{ p_e \right\}_{e \in E}$, y cada nodo de $V - T$ tiene asociado también cierta probabilidad de operación, i.e. tenemos $P_{V-T} = \left\{ p_v \right\}_{v \in (V-T)}$.
Dado cierta probabilidad $p_{\min}$ establecida como umbral, se desea encontrar un subgrafo $G_S \subseteq G$ de costo mínimo que satisfaga la matriz de requerimientos de conexión R y además su confiabilidad satisfaga $R_{(V-T)}(G_S) \geq p_{\min}$.

4. Algoritmos Estudiados

Para poder determinar los algoritmos a utilizar se realizó una investigación basada en la bibliografía encontrada en el InCo, Internet y el apoyo de los docentes tutores, a partir de la información recabada seleccionamos los algoritmos teniendo en cuenta los siguientes criterios:

- Simplicidad: debe de ser simple.
- Eficiencia: debe tomar un tiempo razonable.
- Precisión: debe de estar formulado en términos matemáticos precisos.
- Efectividad: debe encontrar las soluciones óptimas para la mayoría de los problemas de prueba en donde se conoce la solución.

En el momento de seleccionar la metaheurística, la decisión se hace sumamente difícil ya que muchas veces se carece de la información necesaria para realizar una correcta comparación. El primer paso es considerar globalmente las cualidades deseables de una metaheurística (Criterios anteriormente definidos) y corroborar que estas se cumplan. VNS ha mostrado eficiencia en muchos experimentos. La efectividad de esta ha sido probada en la resolución eficaz de problemas de varios bancos de prueba con resultados óptimos o muy cercanos a los óptimos, y con tiempo computacional moderado o al menos razonables. Otra de las cualidades de VNS es la simplicidad, además de todo lo mencionado anteriormente nos resulta interesante que se trata de una metaheurística recientemente presentada y que los trabajos basados en ella son escasos en comparación al resto. Otra de las razones de la selección de VNS es la posibilidad de manejar extensiones a la propia metaheurística.

Por otro lado seleccionamos el algoritmo que utilizamos para calcular la confiabilidad de la solución. Dado que para redes grandes no existen algoritmos exactos para calcular la confiabilidad que sean eficientes. Una solución posible al problema planteado es utilizar métodos de simulación, y dentro de estos hemos estudiado particularmente dos, Monte Carlo Crudo 10 y RVR 10.
El método Monte Carlo, es una opción simple de implementar pero tiene la desventaja de la excesiva varianza de las estimaciones, y por ende los resultados obtenidos son poco precisos. Debido a esto se desarrollaron los métodos de reducción de varianza para la simulación de medidas de confiabilidad y dentro de ellos hemos estudiado en particular RVR 101010. La opción tomada se fundamenta en que a pesar de no ser tan sencilla su implementación ha sido probado 10 que con RVR 1010 los resultados obtenidos son de mayor precisión que para Monte Carlo Crudo 10, siendo un algoritmo adecuado para el cálculo de confiabilidad en redes muy confiables de cualquier tamaño.

En las siguientes secciones se describen los algoritmos relevantes utilizados en la implementación de la solución del problema. Se enumera el algoritmo principal, el algoritmo de construcción, las búsquedas locales seleccionadas para ser utilizadas en VNS 10, el algoritmo VNS 10 y el algoritmo RVR 10 utilizado para el cálculo de la confiabilidad.

4.1.Algoritmo Network Design

NetworkDesign es el algoritmo principal que se encarga de ejecutar las distintas fases que resuelven el problema de este proyecto. Las mismas se pueden enumerar de la siguiente forma:

Construction Phase
Survivability Phase
Reliability Phase

Construction Phase toma el grafo inicial y devuelve un grafo solución factible a partir del cual se iniciará *Survivability Phase* para encontrar la solución óptima aproximada mediante una metaheurística al problema planteado. Por último se realiza *Reliability Phase* midiendo la confiabilidad de la solución encontrada. El algoritmo (según Figura 1) recibe como entrada el

grafo G_B la cantidad de iteraciones $cantiter$ en que se va a ejecutar el algoritmo, el entero k para calcular los k caminos mas cortos utilizado en *Construction Phase,* el umbral de confiabilidad y la cantidad de replicaciones utilizados en *Reliability Phase.*

Procedure NetwokDesign ($G_B, cantiter, k, umbral.simiter$ **);**

1. $i \leftarrow 0$; $P \leftarrow \varnothing$; $sol \leftarrow \varnothing$;
2. **while** $(i < cantiter)$ **do**
3. $\overline{g} \leftarrow$ Construction Phase(G_B,P,k) ;
4. $g_{sol} \leftarrow$ Survavility Phase($\overline{g}$,P) ;
5. $conf \leftarrow$ Reliavility Phase(g_{sol},simiter) ;
6. **if** $(conf \geq umbral)$ **then** $sol \leftarrow$ add(g_{sol}) ;
7. **else** discard(g_{sol}) ;
8. **end if;**
9. **end while;**
10. **return** sol ;
11. **end NetworkDesign**;

Figura 1: Pseudo-código de Network Design

En el paso 1 se inicializa el contador en cero, la matriz P vacía y la colección solución vacía. En el ciclo comprendido entre el paso 2 a 9 se computa $cantiter$ veces el algoritmo *Network Design* obteniendo un conjunto de soluciones guardadas en la colección sol. En el paso 3 se computa la fase de construcción obteniendo una solución factible $\overline{g}$, en el paso 4 se computa la fase de Optimización de $\overline{g}$ (calculado en el paso 3) obteniendo la solución g_{sol}, por último se mide la confiabilidad de g_{sol} en la fase de confiabilidad paso 5, si la misma supera el umbral establecido la solución es guardada en la colección sol paso 6 en otro caso se descarta paso 7.

4.2.Algoritmo de Construcción

En esta fase se presenta un algoritmo de construcción que proporcionará el grafo de partida para realizar la búsqueda de una aproximación al óptimo global del problema. Se utilizó un algoritmo de construcción voraz (Greedy) 1010, este algoritmo genera una solución factible basada en caminos, se busca que sea rápida y optimice en lo posible el costo. El algoritmo construye un grafo que satisface la matriz $R = \{r_{ij}\}_{i,j \in S_D^{(I)}}$ de requerimientos de conexión entre los nodos terminales, esto es dado $i, j \in S_D^{(I)}$ existen r_{ij} caminos nodos disjuntos que los conectan en el grafo (de ahora en más $S_D^{(I)}$ es el conjunto de nodos terminales).

El algoritmo (según Figura 2) toma como entrada el grafo G_B la matriz de costos C la matriz de requerimientos R y el parámetro k.

Procedure Greedy (G_B, C, R, k **);**

12. $g_{sol} \leftarrow (S_D^{(I)}, \varnothing)$; $m_{ij} \leftarrow r_{ij} \forall i, j \in S_D^{(I)}$; $P_{ij} \leftarrow \varnothing \forall i, j \in S_D^{(I)}$; $A_{ij} \leftarrow 0 \forall i, j \in S_D^{(I)}$
13. **while** $\exists m_{ij} > 0$ *such that* $A_{ij} < MAX\_ATTEMPT$ **do**
14. *Let* $i, j \in S_D^{(I)}$ *be randomly chosen pair of fixed switch sites such that* $m_{ij} > 0$;
15. $\overline{G} \leftarrow (G_B \setminus P_{ij})$;
16. **Let** $\overline{C}$ **be the matrix given by:** $\overline{c_{uv}} \leftarrow \begin{Bmatrix} 0 \; if (u,v) \in g_{sol} \\ c_{uv} \; if (u,v) \in (\overline{G} \setminus g_{sol}) \end{Bmatrix}$
17. $L_p \leftarrow$ **the shortest** k **paths from** i **to** j **on** $\overline{G}$ **,considering the matrix** $\overline{C}$;
18. **if** $L_p = \varnothing$ **then** $A_{ij} \leftarrow A_{ij} + 1$; $P_{ij} \leftarrow \varnothing$; $m_{ij} \leftarrow r_{ij}$;
19. **else**
20. $p \leftarrow Select\_Random(L_p)$; $g_{sol} \leftarrow g_{sol} \cup \{p\}$;
21. $P_{ij} \leftarrow P_{ij} \cup \{p\}$; $m_{ij} \leftarrow m_{ij} - 1$;
22. $[P, M] \leftarrow General\_Update\_Matrix(g_{sol}, P, M, p, i, j)$;
23. **end if;**
24. **end while;**
25. **return** g_{sol} , P ;

end Greedy;

Figura 2: Pseudo-código de Greedy 10.

En el paso 1 se inicializa la solución g_{sol} con únicamente los nodos terminales $S_D^{(I)}$ y sin aristas que los conecten, la matriz $M = \{m_{ij}\}_{ij \in S_D^{(I)}}$ indica los requerimientos de conexión aun no satisfechos inicializándose con $m_{ij} = r_{ij} \forall i, j \in S_D^{(I)}$, la matriz $P = \{P_{ij}\}_{ij \in S_D^{(I)}}$ que representa el conjunto de caminos nodos disjuntos para cada par de terminales computados por el algoritmo se inicializa $P_{ij} = \varnothing \forall i, j \in S_D^{(I)}$, por ultimo la matriz $A = \{A_{ij}\}_{ij \in S_D^{(I)}}$ que registra la cantidad de intentos en los que se fallo en encontrar r_{ij} caminos nodos disjuntos para el par de terminales i, j se inicializa $A_{ij} = 0 \forall i, j \in S_D^{(I)}$.

El ciclo del paso 2 al 14 se repite hasta que todos los pares de terminales hayan satisfecho sus requerimientos de conexión ó todos los pares de terminales $i, j \in S_D^{(I)}$ que aun no hayan cumplido con el requerimiento de conexión excedan $MAX\_ATTEMPT$ intentos. Cada iteración trabaja de la siguiente manera, línea 3 hace una selección aleatoria y uniforme de un par de terminales $i, j \in S_D^{(I)}$ tal que $m_{ij} > 0$ (exista al menos un camino nodo disjunto no encontrado), línea 4 computa el grafo $\overline{G} \leftarrow (G_B \setminus P_{ij})$ el cual no contiene ninguna arista o nodo de P_{ij} excepto por i, j por lo que todo camino desde i hasta j será nodo disjunto con los caminos ya encontrados en P_{ij}. Línea 5 computa la matriz auxiliar de costos $\overline{C}$ donde cualquier arista $(u,v) \in g_{sol}$ tiene costo cero, esto permite reutilizar aristas ya existentes en g_{sol} (sin considerar su costo) al buscar nuevos caminos nodos disjuntos. Línea 6 se computan los k caminos mas cortos desde i a j en $\overline{G}$ usando $\overline{C}$ estos caminos se guardan en una lista L_p, para resolver este problema de forma eficiente se utilizo el algoritmo de Yen 10 que resuelve el problema de los k caminos mas cortos (KSP). Línea 7 chequea si la lista de caminos es vacía, en ese caso se re-inicializa P_{ij} y m_{ij} dado que i, j se encuentran en componentes desconectadas del grafo al haber seleccionado previamente caminos que no fueran nodos disjuntos con los aun no encontrados, por otro lado si la lista no es vacía en la línea 9 se sortea de forma aleatoria y uniforme un camino p de L_p y es agregado a g_{sol} de esta forma se obtiene un nuevo camino

nodo disjunto con respecto a los caminos en P_{ij} ya encontrados. En la línea 10 se actualiza P_{ij} agregando p y m_{ij} es decrementado en 1. Al agregar este nuevo camino, otros nuevos caminos nodos disjuntos pudieron haberse encontrado entre algunos pares de terminales por este motivo en la línea 11 se ejecuta el método $General\_Update\_Matrix$ que se encarga de encontrar estos caminos (los detalles de su implementación se encuentran en el Apéndice 3 – Algoritmo de Construcción). Una vez finalizado el ciclo la solución factible se ha construido y es devuelta en la línea 14.

La demostración de la conservación de factibilidad de este algoritmo así como la explicación de los métodos auxiliares $General\_Update\_Matrix$ y KSP se encuentran en el Apéndice 3 – Algoritmo de Construcción.

4.3.Algoritmos de Búsqueda Local

En general la solución generada por la fase de construcción no es si quiera un óptimo local. Como consecuencia de esto, la metaheurística VNS 10 utiliza diferentes algoritmos de búsqueda local con el objetivo de ir cambiando de forma sistemática la vecindad al momento de realizar una búsqueda y poder lograr una mejora a la solución original. Proponemos 3 algoritmos de búsqueda local, cada uno de ellos basado en estructuras diferentes de vecindad. Vale aclarar que uno de ellos (*SwapKeyPathLocalSearch* 10) utiliza y eventualmente actualiza el conjunto P que contiene todos los caminos computados entre pares de nodos terminales, mientras que los restantes no. Esta diferencia que puede parecer menor no lo es tanto ya que en si implicó una decisión de implementación de relevancia. El actualizar el conjunto P mencionado dentro de la implementación del algoritmo de búsqueda local condiciona a que dentro de las sucesivas iteraciones de VNS 10 puedan ser utilizadas o bien búsquedas locales que utilicen dicho conjunto o búsquedas que no lo hagan, pero se descarta la posibilidad de combinarlas ya que esto implicaría perder la versión actualizada de P al correr una búsqueda que no lo utilice y por ende imposibilitar el uso posterior de un algoritmo que si lo necesite. Como resultado de lo expuesto, la búsqueda local *SwapKeyPathLocalSearch* 10 es ejecutada de forma independiente a las iteraciones en las que incurren las búsquedas *KeyPathLocalSearch* 10 y *KeyTreeLocalSearch* 10, lo cual será debidamente ilustrado en las secciones que siguen.

Búsqueda Local 1

Antes de describir en detalle el algoritmo de búsqueda local definiremos una estructura acorde para la vecindad y algunas definiciones auxiliares 10.

Definición key-nodo *Supongamos g_{sol} una solución factible que satisface la matriz de requerimientos de conexión R. Un key-nodo es un nodo no terminal, con grado de por lo menos tres en g_{sol}.*

Definición key-path *Supongamos g_{sol} una solución factible que satisface la matriz de requerimientos de conexión R. Definimos un key-path como un camino en g_{sol} tal que todos sus nodos internos son no terminales con grado dos en g_{sol} y cuyos nodos extremos son nodos terminales o key-nodos.*

Definición (Estructura de Vecindad basada en key-paths) *Supongamos g_{sol} una solución factible que satisface la matriz de requerimientos de conexión R. Dado un key-path $p \subset g_{sol}$ definimos una solución vecina de g_{sol} cómo:*

$\hat{g}_{sol} = (g_{sol} \setminus p) \cup \hat{p}$, *donde* $\hat{p}$ *es otro camino conectando los puntos extremos de* p *y manteniendo la factibilidad en la nueva red* $\hat{g}_{sol}$.

La Vecindad de key-paths de g_{sol} *está compuesta por las soluciones vecinas obtenidas al aplicar la operación antes descripta a cada uno de los distintos key-paths en* $K(g_{sol}) = (p_1, ..., p_h)$.

Este algoritmo construye iterativamente soluciones vecinas reemplazando key-paths de la solución por otros key-paths que tengan los mismos nodos extremos, diseñados de forma adecuada tal que la factibilidad sea preservada. Este proceso se repite hasta que los reemplazos de key-paths no lleven a una mejor solución. Cuando esto ocurre estamos frente a un óptimo local para la vecindad utilizada.
Como se puede apreciar en la figura que acompaña esta sección (Figura 3) el algoritmo toma como entradas al grafo G_B de conexiones factibles, la matriz de costos C y la solución factible actual g_{sol}.
En la línea 1 inicializamos la variable *improve* como TRUE, utilizada para indicar mejoras obtenidas a través de la sustitución de key-paths.
El loop 2-13 busca por soluciones vecinas analizando cada key-path en la solución actual g_{sol} y reemplazando estos por otros key-paths de forma tal de poder mejorar el costo sin perder la factibilidad de la solución.
Cada iteración funciona de la siguiente forma. En la línea 3 la variable *improve* es puesta en FALSE.
La línea 4 computa la descomposición en key-paths de g_{sol}, la cual denotaremos $K(g_{sol})$.
El loop interno 5-13 analiza uno a uno los key-paths de $K(g_{sol})$ con el objetivo de encontrar un nuevo key-path de menor costo para reemplazar el correspondiente key-path original.
La línea 6 selecciona aleatoriamente y uniformemente un key-path $p \in K(g_{sol})$ que aún no haya sido analizado. Denotamos por *u* y *v* los nodos extremos del key-path seleccionado.
En la línea 7 computamos la subred inducida por el conjunto de nodos $NODOS(p) \cup (S_D \setminus NODES(g_{sol}))$, la cual es denotada $\hat{\mu}$. Nótese que en $\hat{\mu}$ no hay nodos de $(g_{sol} \setminus p)$ excepto *u* y *v*. Por ende, todos los caminos conectando *u* y *v* en $\hat{\mu}$ reestablecen la factibilidad de $(g_{sol} \setminus p)$. Consecuentemente la línea 8 computa el camino mas corto de *u* a *v* en $\hat{\mu}$, el cual es denotado por $\hat{p}$. La línea 9 compara el costo de $\hat{p}$ con p. Si $\hat{p}$ tiene costo menor que p, en la línea 10 el key-path p es reemplazado por $\hat{p}$ en g_{sol} y en la línea 11 el indicador *improve* es configurado a TRUE para recomenzar la búsqueda local desde la línea 2. De lo contrario, si $\hat{p}$ tiene costo mayor que p, el loop 5-13 continua con otro key-path aún no analizado o finaliza dado que no quedan mas key-paths para analizar.
Una vez que no haya mejoras realizables mediante el reemplazo de key-paths, el loop 2-14 finaliza y la mejor solución vecina encontrada es retornada en la línea 15.

Procedure KeyPathLocalSearch (G_B, C, g_{sol} **);**

26. *improve* $\leftarrow$ TRUE;
27. **while** *improve* **do**
28. *improve* $\leftarrow$ FALSE;
29. $K(g_{sol}) \leftarrow$ **la descomposición en key-paths de** g_{sol} **;**
30. **while not(***improve***) and** $\exists$ **key-paths no analizados**
31. **Sea** $p \in K(g_{sol})$ **un key-path aún no analizado con extremos** *u* **y** *v***;**
32. $\hat{\mu} \leftarrow$ **el subgrafo inducido por** $NODOS(p) \cup (S_D \setminus NODES(g_{sol}))$ **;**
33. $\hat{p}$ **es el camino mas corto de** *u* **a** *v* **en** $\hat{\mu}$ **;**
34. **if** $COST(\hat{p}) < COST(p)$ **then**
35. $g_{sol} \leftarrow (g_{sol} \setminus p) \cup \hat{p}$ **;**
36. *improve* $\leftarrow$ TRUE;
37. **end if;**
38. **end while;**
39. **end while;**
40. **return** g_{sol} **;**

end BusquedaLocal1;

Figura 3: Pseudo-código de KeyPathLocalSearch 10

La demostración de conservación de factibilidad se encuentra en el Apéndice 2 – Búsqueda Local, conservación de factibilidad.

Búsqueda Local 2

Antes de describir la estrategia de búsqueda local basada en reemplazo de key-trees, definimos una estructura de vecindario acorde 10.

Definición key-tree: *Supongamos* g_{sol} *una solución factible que satisface la matriz de requerimientos de conexión* R *y un key-nodo* $v \in g_{sol}$ *. Definimos un key-tree asociado a* v *como a un árbol de* g_{sol} *el cuál esta conformado por todos los key-paths qué tienen a* v *como uno de sus extremos. Topológicamente, podemos verlo como un conjunto de caminos que tienen a* v *como extremo en común (key-nodo* v*).*

Definición (Estructura de Vecindad basada en key-tree) *Supongamos* g_{sol} *una solución factible que satisface la matriz de requerimientos de conexión* R *. Dado un key-nodo* $v \in g_{sol}$ *y su key-tree asociado* $T_v \subset g_{sol}$ *, definimos una solución vecina de* g_{sol} *cómo:*

$\hat{g_{sol}} = (g_{sol} \setminus T_v) \cup T$ *, donde* T *es otro key-tree abarcando los extremos de* T_v *y manteniendo la factibilidad en la nueva red* $\hat{g_{sol}}$ *.*

La Vecindad de key-tree de g_{sol} *está compuesta por las soluciones vecinas obtenidas al aplicar iterativamente la operación antes descripta a cada uno de los distintos key-trees en* g_{sol} *.*

Basándonos en esta estructura de vecindad definimos la siguiente búsqueda local basada en reemplazo de key-trees, a diferencia del reemplazo de key-paths en el que se basa la búsqueda anterior.

Procedure KeyTreeLocalSearch (G_B, C, g_{sol} **);**

1. *improve* $\leftarrow$ TRUE;
2. **while** *improve* **do**
3. *improve* $\leftarrow$ FALSE;
4. **Dejemos** X **como el conjunto de key-nodes en** g_{sol} **;**
5. $\bar{S} \leftarrow S_D \setminus NODOS(g_{sol})$ **;**
6. **while not(***improve***) and** $\exists$ **key-nodos que no hayan sido analizados do**
7. **Supongamos** $v \in X$ **que no ha sido analizado aún;**
8. $\left[g_{sol}, improve\right] \leftarrow$ General_RecConnect($G_B, C, g_{sol}, v, \bar{S}$);
9. **end while;**
10. **end while;**
11. **return** g_{sol} **;**

end BusquedaLocal2;

Figura 4: Pseudo-código de KeyTreeLocalSearch 10

El algoritmo construye iterativamente soluciones vecinas mediante el reemplazo de key-trees desde la solución actual por otros key-trees que son diseñados de forma acorde para que la factibilidad sea preservada. El proceso se repite hasta que no se induce ninguna mejora mediante el reemplazo de key-trees.
Como se puede apreciar en la figura que acompaña esta sección (Figura 4) el algoritmo toma como entradas al grafo G_B de conexiones factibles, la matriz de costos C y la solución factible actual g_{sol}.
En la línea 1 se inicializa la variable *improve* en FALSE, siendo esta utilizada para indicar mejoras obtenidas por el reemplazo de key-trees.
El loop 2-10 busca por soluciones vecinas analizando cada key-nodo en la solución actual g_{sol} y reemplazando si es posible sus respectivos key-trees por otros key-trees con el objetivo de mejorar su costo sin perder la factibilidad. Cuando alcanzamos una mejor solución factible mediante un reemplazo de key-tree, la búsqueda local reanuda desde esta nueva solución factible.
Cada iteración funciona de la siguiente forma.
En la línea 3 *improve* es puesto en FALSE. La línea 4 computa el conjunto X de key-nodos de g_{sol}.
La línea 5 computa el conjunto $\bar{S}$ de nodos no terminales que no pertenecen a g_{sol}.
El loop interno 6-9 analiza de a uno a cada key-nodo de X con el objetivo de encontrar un key-tree acorde de menor costo para reemplazar el key-tree correspondiente.
La línea 7 selecciona al azar un nodo $v \in X$ (de forma uniforme). En la línea 8 ejecutamos el algoritmo llamado *General_RecConnect* de forma de poder hallar un key-tree substituto para el key-tree asociado con *v*, con menor costo y que preserve la factibilidad de la solución (la descripción del algoritmo y la prueba de la conservación de la factibilidad se encuentran en el Apéndice 2 – Búsqueda Local, conservación de factibilidad.
Si esta búsqueda es satisfactoria, el algoritmo mencionado entrega una solución vecina mejorada y la solución actual g_{sol} es actualizada en la misma línea. Sumado a esto, la variable *improve* es configurada a TRUE, para luego reanudar la búsqueda local desde la línea 2. De otra forma, si *General_RecConnect* no puede hallar un key-tree sustituto, el loop 6-9 considera otro key-nodo que no haya sido analizado hasta el momento o finaliza si es que no quedan key-nodos por analizar. Una vez que no existan mejoras posibles a realizar mediante el reemplazo de key-trees, la solución actual g_{sol} es retornada en la línea 11.

Búsqueda Local 3

Antes de describir la estrategia de búsqueda local basada en intercambio de key-paths, definimos una estructura de vecindario acorde.

Definición (Estructura de Vecindad basada en intercambio de key-paths) *Supongamos* g_{sol} *una solución factible que satisface la matriz de requerimientos de conexión* R*, y* P *es la matriz que contiene los caminos entre los nodos terminales involucrados en la solución* g_{sol}*. Dado un key-path* $p \subset g_{sol}$ *definimos una solución vecina de* g_{sol} *cómo:*

$\overset{\wedge}{g_{sol}} = g_{sol} \setminus p \cup m$ *donde m es un conjunto de nodos y aristas que se agregarán a la solución para seguir conservando la factibilidad de la solución* $\overset{\wedge}{g_{sol}}$*. El conjunto m puede ser vacío si al quitar un keypath de la solución se encuentra que la solución sigue siendo factible.*

La Vecindad de key-paths de g_{sol} *está compuesta por las soluciones vecinas obtenidas al aplicar la operación antes descripta a cada uno de los distintos key-paths en* $K(g_{sol}) = (p_1, ..., p_h)$*.*

Este algoritmo construye iterativamente soluciones vecinas quitando los key-paths y reconstruyendo una solución factible a partir de la información almacenada en la matriz P. Este proceso se repite hasta que los reemplazos de key-paths no lleven a una mejor solución.

Este algoritmo toma como entradas al grafo G_B de conexiones factibles, la matriz de costos C la solución factible actual g_{sol} y la matriz P de caminos entre nodos terminales.

En la línea 1 inicializamos la variable *improve* como TRUE, utilizada para indicar mejoras obtenidas a través de la sustitución de key-paths.

El loop 2-9 busca por soluciones vecinas analizando cada key-path en la solución actual g_{sol} y reemplazando estos por otros nodos y aristas, o simplemente eliminándolos de forma tal de poder mejorar el costo sin perder la factibilidad de la solución.

Cada iteración funciona de la siguiente forma.

En la línea 3 la variable *improve* es puesta en FALSE.

La línea 4 computa la descomposición en key-paths de g_{sol}, la cual denotaremos $K(g_{sol})$.

El loop interno 5-8 analiza uno a uno los key-paths de $K(g_{sol})$ con el objetivo de encontrar un conjunto de vértices y aristas de menor costo para reemplazar el correspondiente key-path original.

La línea 6 selecciona aleatoriamente y uniformemente un key-path $p \in K(g_{sol})$ que aún no haya sido analizado. Denotamos por *u* y *v* los nodos extremos del key-path seleccionado.

En la línea 7 se llama a la rutina *FindSubstituteKeyPath* que eliminará el keypath de la solución y a partir de la información de la matriz P intentará reconstruir una solución factible, Si luego de construir la nueva solución resulta que el costo es menor que el original se retorna true y se substituye la solución por la nueva encontrada, en caso contrario se sigue con el siguiente key-path a analizar

Una vez que no haya mejoras realizables mediante el reemplazo de key-paths, el loop 2-9 finaliza y la mejor solución vecina encontrada es retornada en la línea 10.

Procedure SwapKeyPathLocalSearch (G_B, C, g_{sol}, P **);**

1. *improve* ← TRUE;
2. **while** *improve* **do**
3. *improve* ← FALSE;
4. $K(g_{sol})$ ← **la descomposición en key-paths de** g_{sol} **;**
5. **while not(***improve***) and** $\exists$ **key-paths no analizados**
6. **Sea** $p \in K(g_{sol})$ **un key-path aún no analizado;**
7. *improve* ← FindSubstituteKeyPath(g_{sol}, p, P)
8. **end while;**
9. **end while;**
10. **return** g_{sol} **;**

end BusquedaLocal1;

Figura 5: Pseudo-código de SwapKeyPathLocalSearch 10

Se adjunta demostración de conservación de factibilidad en el Apéndice 2 – Búsqueda Local, conservación de factibilidad.

Finalizada la presentación de los algoritmos de búsqueda local se puede observar que las estructuras de vecindad utilizadas por los mismos difieren entre sí, lo cual favorece la aplicación del algoritmo VNS.

4.4.VNS - Búsqueda de Vecindad Variable

A diferencia de la mayoría de los algoritmos de búsqueda local que usan una única definición de vecindad, VNS 10 se sustenta sobre la idea de ir cambiando en forma sistemática la vecindad al momento de realizar la búsqueda y por ende necesita de un conjunto finito de vecindades distintas predefinidas.
VNS 10 se basa en tres hechos simples:

1. Un mínimo local con respecto a una estructura de vecindad no lo es necesariamente con respecto a otra.
2. Un mínimo global es un mínimo local con respecto a todas las posibles estructuras de vecindades.
3. En muchos problemas el mínimo local con respecto a una o varias estructuras de vecindad están relativamente cerca.

La última observación es empírica e implica que un mínimo local muchas veces brinda información acerca del óptimo global. Puede ser, por ejemplo, que ambas soluciones tengan características comunes. Sin embargo, generalmente no se conoce cuales son esas características. Es procedente, por tanto, realizar un estudio organizado en las proximidades de este óptimo local, hasta que se encuentre uno mejor. Los hechos 1 a 3 sugieren el empleo de varias estructuras de entornos (vecindades) en las búsquedas locales para abordar un problema de optimización. El cambio de estructura de entornos se puede realizar de forma determinística, estocástica, o determinística y estocástica a la vez (ver informe de contexto del problema). En este trabajo se utilizo una solución determinística llamada VNS descendente (VND 10) la cual consiste en reemplazar iterativamente la solución actual por el resultado de la búsqueda local, mientras se produzca mejora. Si se realiza un cambio de estructura de entornos de forma determinística cada vez que se llega a un mínimo local, se obtiene la búsqueda de entorno variable descendente (*Variable Neighbourhood Descent*, VND 10). Los pasos de la VND se muestran a continuación en la Figura 6.

Inicialización:

Seleccionar el conjunto de vencidades $N_k, k = 1 \ . \ . \ k_{\max}$ **que se utilizaran en el descenso.**

Seleccionar la solución inicial x **.**

Iteraciones:

Repetir hasta que no se obtenga mejora la siguiente secuencia:

1. **Hacer** $k \leftarrow 1$
2. **Repetir hasta que** $k = k_{\max}$

   Exploración del entorno: Encontrar la mejor solución $\hat{x}$ **del** $k - esimo$ **entorno de** $x(\hat{x} \in N_k(x))$

   Moverse o no: Si la solución obtenida $\hat{x}$ **es mejor que** x **, hacer** $x \leftarrow \hat{x}$ **y** $k \leftarrow 1$

   En otro caso hacer $k \leftarrow k + 1$

Figura 6: VNS Descendente (VND) 10

La solución final proporcionada por el algoritmo es un mínimo local con respecto a todas las $k \max$ vecindades, y por tanto la probabilidad de alcanzar un mínimo global es mayor que usando una sola vecindad 10.

A continuación explicamos el algoritmo utilizado para implementar VNS 10 en nuestra solución.

La misma se trata de una implementación de VNS descendente (VND 10) con algunas variantes que mencionaremos (Figura 7). El algoritmo recibe como entrada el grafo G solución inicial y la matriz P de caminos nodos disjuntos de G, ambos salida del algoritmo de construcción explicado previamente y una colección de búsquedas locales cls. Inicialmente se calcula el costo de G y se le aplica la búsqueda local $SwapKeyPathLocalSearch$ la cual utiliza la matriz de caminos P también salida del algoritmo de construcción (líneas 1 y 2). Cabe destacar como se explico previamente que $SwapKeyPathLocalSearch$ es incompatible con las otras búsqueda locales, debido a esto se corre solo al inicio del algoritmo una única vez y no se considera parte del conjunto de vecindades mencionado anteriormente; no obstante si forma parte de nuestra implementación de VNS y es fundamental su incorporación para lograr mejoras importantes en las soluciones iniciales generadas por el algoritmo de construcción mejorando el tiempo de computo del algoritmo. En la línea 3 se obtiene el nuevo costo de G, se inicializa $notimprove \leftarrow 0$ (línea 4).

Procedure VNS(G, cls, P **);**

1. $\cos t \leftarrow getCost(G); k_{\max} \leftarrow 0$;
2. $G \leftarrow SwapKeyPathLocalSearch(G, P)$;
3. $\cos t \leftarrow getCost(G)$;
4. $notimprove \leftarrow 0$;
5. **while** $(notimprove < k_{\max})$ **do**
6. $\bar{G} \leftarrow cls[k_{\max}] -> Search(G, k)$;
7. $\cos t \leftarrow getCost(G)$;
8. $new\cos t \leftarrow getCost(\bar{G})$;
9. **if** $(new\cos t < \cos t)$ **then**
10. $\cos t \leftarrow newcost$;
11. $notimprove \leftarrow 0$;
12. $G \leftarrow \bar{G}$;
13. **else** $notimprove \leftarrow notimprove + 1$;
14. **end_if**
15. $k \leftarrow (k+1) \bmod k_{\max}$;
16. **end_while**
17. **return** G ;

end VNS

Figura 7: VNS Descendente (VND) 10

En el ciclo 5-14, hasta que no se encuentren mejoras por parte del conjunto de vecindades, se realiza la búsqueda local k en busca de una mejor solución (línea 6) si se logra una mejora, se actualiza el costo (línea 7) y se setea $notimprove \leftarrow 0$ (línea 8) y se actualiza la nueva solución (línea 9), en caso de no haber mejora $notimprove$ es incrementado en uno (línea 13), por último haya o no habido mejora se pasa a la siguiente vecindad de forma circular (línea 15), aquí tenemos una variante del algoritmo general visto en la Figura 4 ya que al encontrarse una mejora se continua explorando nuevas soluciones con la siguiente vecindad en lugar de volver al comienzo explorando en vecindades ya utilizadas, debido a que las vecindades utilizadas en nuestra solución son igualmente relevantes consideramos adecuado eliminar la prioridad de las vecindades que queda establecida en el algoritmo general. Por último en la línea 17 se obtiene la solución calculada. Es importante destacar que este algoritmo es configurable para utilizar N búsquedas locales pasadas por parámetro en la colección cls .

4.5.RVR - Reducción Recursiva de Varianza

RVR 1010 reduce el problema original a un problema en una red más pequeña construida a partir de la original. El proceso, como su nombre lo menciona, es recursivo, deteniéndose cuando la red se encuentra siempre en estado operativo o no operativo, independientemente del estado de sus componentes. La construcción del método se realiza para obtener una estimación de la medida Q_k (anti-confiabilidad para un conjunto ***K*** de terminales), pero se puede generalizar para todas las medidas restantes. Definiciones, propiedades y seudo-códigos presentes en esta sección fueron extraídos de 10, por mayor información referirse a la citada bibliografía.

DEFINICIONES

Red *K*-conexa: una red ***G***=(***V***,***E***) con un conjunto de terminales asociado ***K***$\subseteq$***V*** es *K*-conexa cuando existe por lo menos un camino entre todo par de nodos del conjunto ***K*** (red en estado operativo).

Q(***G***): anti-confiabilidad de ***G*** (probabilidad de que la red ***G*** no sea *K*-conexa).

Un conjunto $D \subset V \cup E$ es un *K*-corte extendido de G si la subred $G'=(V - D, E - D)$ no es *K*-conexa.
Para una arista $e \in E$ en $G=(V, E)$ con terminales en K, $G - e$ es la red cuyo conjunto de nodos es V y cuyo conjunto de aristas se obtiene a partir de E eliminando e. El conjunto de terminales de $G - e$ es igual a K.
Para un nodo $v \in V$ en $G=(V, E)$ con terminales en K, $G - v$ es la red cuyo conjunto de nodos es $V - \{v\}$ y cuyo conjunto de aristas se obtiene a partir de E eliminando todas las aristas incidentes a v. El conjunto de terminales de $G - v$ es igual a $K - \{v\}$.
Sea $G=(V, E)$ con terminales en K y d un componente (nodo o arista) de la misma. Se denota $G \mid d$ a la red derivada de G seteando la probabilidad de funcionamiento de d a 1 (d será un componente perfecto).
Sea $G=(V, E)$ con terminales en K y d un componente (nodo o arista) de la misma. Se denota $G * d$ a la red derivada de G seteando la probabilidad de funcionamiento de d a 1. Si luego se halla una arista $e=(v_1, v_2)$ (si d es una arista será $e=d$; si d es un nodo será $v_1=d$ ó $v_2=d$) tal que $r_e=r_{v1}=r_{v2}=1$ se debe realizar la contracción de la misma, esto es, eliminando e, fundiendo sus extremidades v_1 y v_2 en un nuevo nodo w y seteando el nuevo conjunto de terminales a $K - \{v_1, v_2\} \cup \{w\}$ si v_1 o v_2 pertenece a K, o simplemente K si ninguno es terminal.

El objetivo ahora, es construir un estimador con igual esperanza que el utilizado en *Monte Carlo Crudo* (ver Informe de contexto del problema, Apéndice 1) y con menor varianza.
Para ello se consideran las siguientes propiedades:

PROPIEDAD UNO
Sea $G=(V, E)$ con terminales en $K = \{v_1, v_2, ..., v_{|K|}\}$, entonces se cumple:

$$R(G) = \left(\prod_{v \in K} r_v \right) R(G \mid v_1 \mid v_2 \mid ... \mid v_{|k|})$$

y

$$Q(G) = \left(1 - \prod_{v \in K} r_v \right) + \left(\prod_{v \in K} r_v \right) Q(G \mid v_1 \mid v_2 \mid ... \mid v_{|k|})$$

Significa que la medida de confiabilidad puede obtenerse seteando los nodos terminales como perfectos, multiplicando el resultado de la confiabilidad de esa red resultante por los productos de las confiabilidades de los nodos terminales. Esta propiedad permite reducir el caso de fallas en aristas y nodos al modelo de nodos perfectos.

PROPIEDAD 2
Sea $G=(V, E)$ con terminales en K, y d un componente de G, entonces se cumple $R(G \mid d) = R(G * d)$.

Significa que si al setearse un componente como perfecto en la red original se puede realizar contracción, la confiabilidad de la red original es igual a la confiabilidad de la red "contraída". Utilizando esta propiedad, el algoritmo **RVR** (que se presenta más adelante) reduce el problema original a un problema en una red cada vez de menor tamaño.

Si definimos la variable aleatoria $Y(G)$ como $Y(G) = 1 - \phi(X_G)$ (cuya esperanza es el estimador de la confiabilidad para *Monte Carlo Crudo*, donde X_G es un vector de los estados aleatorios de los componentes), el objetivo de la técnica RVR es construir una variable aleatoria $Z(G)$ con la misma esperanza de $Y(G)$ y menor varianza. Dicha variable aleatoria se construye a partir de un *K*-corte extendido D, y se expresa en términos de $|D|$ variables aleatorias $Y(G_i)$ correspondientes a estados de la red original.

PROPIEDAD 3
Para una red ***G*** tal que $r_v = 1$ $v \in K$ (red donde los terminales no fallan) sean

$D = \{d_1, d_2, ..., d_{|D|}\}$ un *K*-corte extendido en ***G***
A_D el evento *"todos los componentes en **D** fallan"*

$Q_D = \Pr\{A_D\} = \prod_{i=1}^{|D|}\left(1 - r_{d_i}\right)$:la probabilidad del evento A_D

B_i el evento *"todos los componentes en $D_i = \{d_1, d_2, ..., d_{i-1}\}$ fallan y d_i funciona"*
$G_i = (G - d_1 - ... - d_{i-1}) * d_i$
V variable aleatoria discreta independiente de las $Y(G_i)$ con

$$\Pr\{V = v\} = \Pr\{B_v\} / (1 - Q_D) = r_{d_v} \prod_{i=1}^{v-1}\left(1 - r_{d_i}\right) \ / (1 - Q_D) \text{ para } 1 \le v \le |D|$$

entonces la variable aleatoria

$$Z(G) = Q_D + (1 - Q_D)\sum_{i=1}^{|D|} 1_{v=i}\, Y(G_i)$$

verifica

$E\{Z(G)\} = Q(G)$ (a)

$\text{Var}\{Z(G)\} = (Q(G) - Q_D)R(G) \le Q(G)R(G) = \text{Var}\{Y(G)\}$ (b)

La Propiedad 3 muestra que la variable aleatoria Z tiene la misma esperanza, y menor varianza que la original Y. A grandes rasgos, la justificación de la igualdad (a) se obtiene observando que el suceso AD y los sucesos Bi constituyen una partición del suceso "la red se encuentra en estado no operativo" cuya probabilidad Q se desea calcular, y utilizando el teorema de probabilidades totales; la reducción de varianza se puede ver observando que en la estimación de la medida Q, siempre se tienen en cuenta estados de falla (suceso AD).

En base a lo anterior, se construye en forma recursiva la siguiente variable aleatoria ***F***:

$$F(G) = \begin{cases} 1 & \text{si G no es K - conexo} \\ 0 & \text{si K está formado por un solo nodo (G siempre K - conexo)} \\ Q_D + (1 - Q_D)\sum_{i=1}^{|D|} 1_{v=i} F(G_i) & \text{de otra forma} \end{cases}$$

y el siguiente algoritmo denominado **RVR** para obtener una muestra de ***F*** para una red ***G*** :

Algoritmo **RVR**

1 - Chequear fin de recursión:
Si ***G*** es siempre *K*-conexo (|***K***|=1) retornar 0
Si ***G*** no es *K*-conexo (evaluando ϕ) retornar 1

2 - Encontrar un *K*-corte extendido ***D*** de ***G***: $\boldsymbol{D} = \{\boldsymbol{d_1}, \boldsymbol{d_2}, ..., \boldsymbol{d_{|D|}}\}$

3 - Calcular $\boldsymbol{Q_D}$ (probabilidad de que todos los componentes en ***D*** fallen)

4 - Sortear una muestra ***v*** de la variable aleatoria ***V***

5 - Construir $\boldsymbol{G_V} = (\boldsymbol{G} - \boldsymbol{d_1} - ... - \boldsymbol{d_{v-1}}) * \boldsymbol{d_v}$ (eliminaciones y contracción)

6 - Paso recursivo: retornar $\boldsymbol{Q_D} + (1 - \boldsymbol{Q_D})\boldsymbol{RVR}(\boldsymbol{G_V})$

Figura 8: Algoritmo RVR

El algoritmo RVR 1010 se implementó en una clase que lleva su nombre, la cual hereda las características de la clase *Reliability*, la misma define las medidas de interés R_v, R_{st}, R_K (ver sección 6.3 – *Pruebas de Validación Algoritmo RVR*). Fueron implementadas también las funcionalidades necesarias en la clase *Graph* de forma de poder aplicar el algoritmo RVR a la estructura previamente diseñada.
A continuación se describe brevemente detalles de implementación del mismo.

```
function Conf(G, sem, N):{[0..1], real}
        s = 0;                  /* acumula los resultados de cada replicación */
        ss = 0;                 /* acumula los cuadrados de cada replicación */
        setearSemilla(sem);
          for I =1 to N do
                G' = G;
                /* RVR calcula la anti-confiabilidad */
                x = 1 - Rvr(G');
                s = s + x;
                ss = ss+ x * x
        end for;
          esp = s / N;
          var = (1/(N*(N-1)))*(ss - s*s/N);
          return (esp, var);
end conf;
```

Figura 9: Pseudo-código de cálculo de confiabilidad de una red utilizando RVR

La clase RVR cuenta con un algoritmo para el cálculo de la confiabilidad utilizando el algoritmo motivo de su nombre (Figura 9). El mismo recibe el grafo, un identificador de semilla para el generador de número seudo-aleatorios (utilizamos (**unsigned**)time(0)) y la cantidad de replicaciones a realizar.

```
function Rvr(G) : [0..1]
        if (terminales = 1) then
                return 0;
        else
                if not fi(G) then
                        return 1;
                else
                        D := GetKExtendedCut(G);
                        Qd := AllFailedProb(D);
                        indice := GetRandomItem(D);
                        c := D[indice];
                        remove(G,D, indice – 1);
                        add(G, c);
                        return Qd + (1 - Qd)* Rvr(G);
end Rvr;
```

Figura 10: Pseudo-código implementación RVR

La Figura 10 muestra la implementación de RVR, en donde la variable *terminales* lleva la cuenta de la cantidad de terminales en la red. Dicha variable es definida según la medida de interés y configurada con una de las operaciones heredadas de la clase *Reliability*. La misma se decrementa ya sea por eliminación de nodos terminales o por la ocurrencia de una contracción. La función fi es un algoritmo de evaluación de estructura que indica si el grafo está en estado operativo. En nuestro caso un estado operativo está ligado a la existencia de por lo menos un camino entre todo par de nodos terminales, verificándose esto mediante una recorrida en profundidad partiendo desde uno de ellos. La falla de un nodo terminal indica un estado no operativo por lo que dicho chequeo forma parte del algoritmo de verificación de estado.

```
function fi(G') : {0,1}
         foreach v ∈  K do
                if (v no operativo) then
                        return 0;
         end for;
         alcanzados := 0;
         v := terminalArbitrario();
         DFS (v);
         if (alcanzados = |K |) then
                return 1;
         else
                return 0;
end fi;



procedure DFS (v)
        marcar v como visitado;
        if (v es terminal) then
                alcanzados++;
        foreach u∈ adyacentes(v)
                 if (u no visitado) then
                        DFS(u);
        end for;
end DFS;
```

Figura 11: Pseudo-código de función de estructura

Retomando, describiremos brevemente las funciones auxiliares utilizadas en la Figura 11.
GetKExtendedCut toma de forma arbitraria un nodo terminal de la red y considera el conjunto de todos los nodos adyacentes y aristas incidentes al mismo con probabilidad de funcionamiento estrictamente menos que 1 agregando estos al resultado.

AllFailedProb calcula el producto de los complementos de las probabilidades de funcionamiento de cada uno de los componentes del corte extendido.

GetRandomItem obtiene, en base al sorteo de una variable aleatoria uniforme en el intervalo [0,1], una muestra de la variable aleatoria v devolviendo el índice del componente sorteado del K corte extendido.

Remove elimina del grafo G todos los elementos del corte hasta la posición índice – 1 (siendo índice el índice del componente sorteado). Para ello se deshabilita el mismo en el grafo y si se trata de un nodo terminal se decrementa la variable terminales.

Add configura la probabilidad de funcionamiento del componente a 1 y corrobora la posibilidad de contraer el grafo luego de realizado dicho cambio.
La contracción es realizada en caso de encontrar una arista y sus nodos extremos con probabilidad de funcionamiento 1. En el caso que uno de los nodos extremos sea terminal se considera w como tal (fusión de ambos nodos y arista). En caso que ambos extremos sean terminales se selecciona uno arbitrariamente como nodo w y se decrementa la variable terminales.
Un caso especial se da cuando ya existe una arista entre el nodo *x* y un adyacente a *y* (sea y_i tal nodo, r_1 su probabilidad de funcionamiento y r_2 la probabilidad de funcionamiento de la arista (y, y_i)). En este caso, al realizar la contracción se debe efectuar una reducción en paralelo de las dos aristas (sustituyéndose las dos originales por una sola) donde la probabilidad de funcionamiento de la nueva arista será $r_1 + r_2 - r_1 * r_2$.

De esta forma podemos ver como en cada paso recursivo el problema original se reduce a uno más pequeño debido a que el algoritmo reduce la cantidad de componentes del grafo original ya sea eliminando componentes o mediante contracciones.
Por lo tanto la cantidad de invocaciones recursivas se puede acotar por |**V**|+|**E**| y observando que la operación más costosa en cada paso es la evaluación de la función de estructura ***fi*** (de orden O(|**V**|)), el orden del algoritmo **RVR** tendrá orden de complejidad O(|**V**| * (|**V**|+|**E**|)).
Para realizar varias replicaciones de la simulación, el algoritmo anterior es invocado varias veces mediante una iteración. La esperanza de **F**(**G**) se estima como:

$$E(F) = \frac{1}{N}\sum_{i=1}^{N} F_i$$

y la varianza se estima mediante la siguiente expresión

$$Var(F) = \frac{1}{N(N-1)}\sum_{i=1}^{N}\left(F_i - E(F)\right)^2$$

que por comodidad para su cálculo se puede llevar a

$$Var(F) = \frac{1}{N(N-1)}\left(\sum_{i=1}^{N} F_i^2 - \frac{1}{N}\left(\sum_{i=1}^{N} F_i\right)^2\right)$$

Modelos matemáticos extraídos de 10.

5. Proceso de implementación del modelo

El desarrollo del modelo a implementar se baso en tres etapas bien definidas siendo las mismas el relevamiento de requerimientos, análisis y diseño y finalmente implementación.
Con respecto al relevamiento de requerimientos cabe señalar que los mismos fueron relevados en base a las características del problema objetivo de este proyecto y mas específicamente al modelo matemático base del mismo, siendo primeramente definida una lista mínima de ellos en lenguaje natural la cual fue incrementada a medida que las exigencias del problema lo ameritaban.

5.1.Especificación de Requerimientos

La especificación de requerimientos se realizo en base a la propuesta inicial de proyecto brindada y a las sucesivas entrevistas realizadas con los tutores. De las entrevistas con el Dr. Ing. Franco Robledo usuario responsable del trabajo planteado, se llego a la definición del siguiente problema a resolver.

Dado un grafo simple no dirigido $G = (V, E)$, un sub-conjunto de nodos distinguidos $T \subseteq V$ (denominados terminales), una matriz de costos $C = \left\{ c_{ij} \right\}_{(i,j) \in E}$ asociados a las aristas de G y una matriz de requerimientos de conexión entre pares de nodos terminales $R = \left\{ r_{ij} \right\}_{i,j \in T}$. Supongamos además que cada arista de G tiene asociada cierta probabilidad de operación, i.e. tenemos: $P_E = \left\{ p_e \right\}_{e \in E}$, y cada nodo de $V - T$ tiene asociado también cierta probabilidad de operación, i.e. tenemos $P_{V-T} = \left\{ p_v \right\}_{v \in (V-T)}$.
Dado cierta probabilidad $p_{\min}$ establecida como umbral, se desea encontrar un subgrafo $G_S \subseteq G$ de costo mínimo que satisfaga la matriz de requerimientos de conexión R y además su confiabilidad satisfaga $R_{(V-T)}(G_S) \geq p_{\min}$.

Para resolver este problema se debe desarrollar una heurística o metaheurística que obtenga una solución aproximada al problema de optimización, y se debe desarrollar un método para medir la confiabilidad de dicha solución.
Además se considera relevante que la biblioteca creada sea extensible, por lo que se tendrá en cuenta dicho requerimiento de forma que puedan ser agregados nuevos algoritmos sin necesidad de modificar modulo alguno. Para ello se debe posibilitar la extensibilidad en lo que respecta a los algoritmos de búsqueda local, algoritmos de construcción de solución factible, algoritmos de optimización, y algoritmos de confiabilidad. No hay requerimientos no funcionales en este trabajo no obstante se recomienda utilizar un lenguaje de programación performante en cuanto al tiempo de computo debido al alto orden de complejidad de los algoritmos necesarios para resolver el problema planteado.

5.2.Diseño

Se opto por adoptar una metodología orientada a objetos utilizando para la especificación tanto del análisis como del diseño notación **UML 10**.

Arquitectura

Debido a las características del software a desarrollar para los requerimientos relevados, la arquitectura utilizada consta de una única capa llamada capa “Lógica”. No se cuenta con una capa de presentación ni capa de acceso a datos, la misma capa lógica accede a los datos utilizando archivos XML en los cuales se representan los grafos y archivos de texto plano donde se guardan los resultados obtenidos. En la Figura 12 se muestran los distintos componentes que conforman la arquitectura.

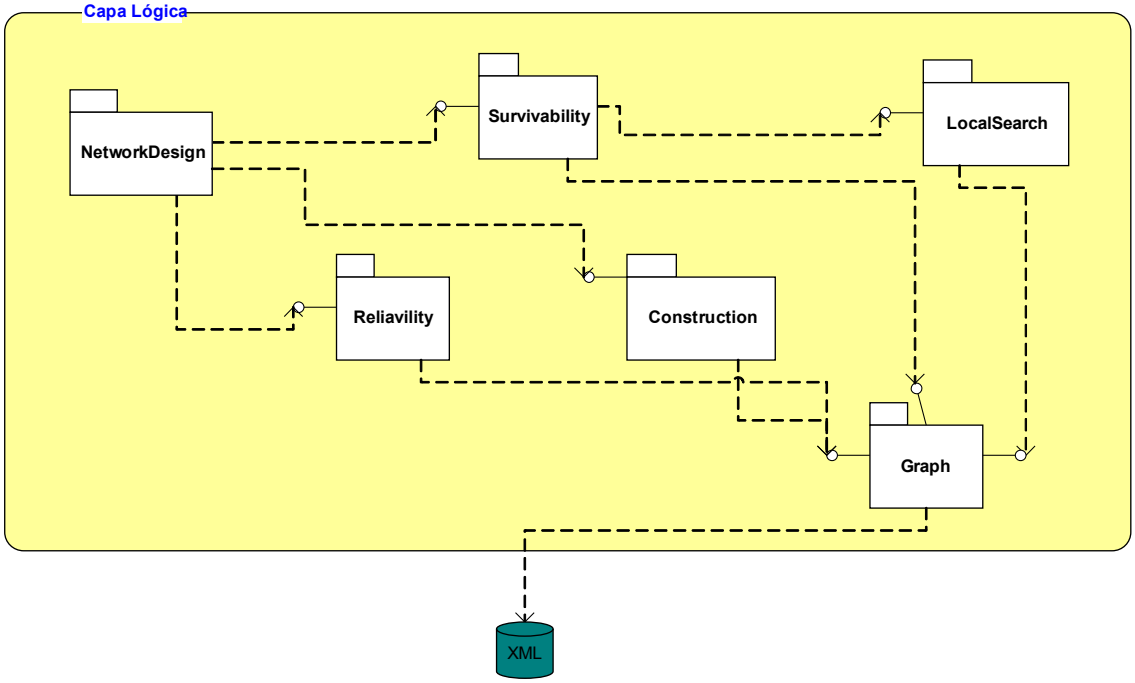


Figura 12: Diagrama de componentes

Descripción de Componentes

Network Design
Se encarga del algoritmo principal del proyecto. El mismo toma la red original y a partir de ella construye una solución factible, luego se optimiza la solución factible y por ultimo mide la confiabilidad de la solución optimizada.

Construction
Se encarga del algoritmo de construcción. Construye una solución factible a partir de una red.

Survivability
Se encarga del algoritmo de optimización. Busca una solución óptima por intermedio de una metaheurística.

Local Search
Se encarga de resolver el/los algoritmo/s de búsqueda local que se utilizan para realizar la optimización de la red.

Reliability
Se encarga del algoritmo de confiabilidad. Mide la confiabilidad de una red.

Graph
Se encarga de resolver toda la lógica necesaria para manipular una red y para realizar las operaciones que necesitan los algoritmos de Construcción, Optimización y Confiabilidad.

Diagrama de Clases

En la Figura 13 y Figura 14 se detallan los diagramas de clases simplificados (no se incluyen todos los métodos y atributos), con las clases y sus respectivos métodos y atributos más relevantes utilizadas en el diseño de la solución.

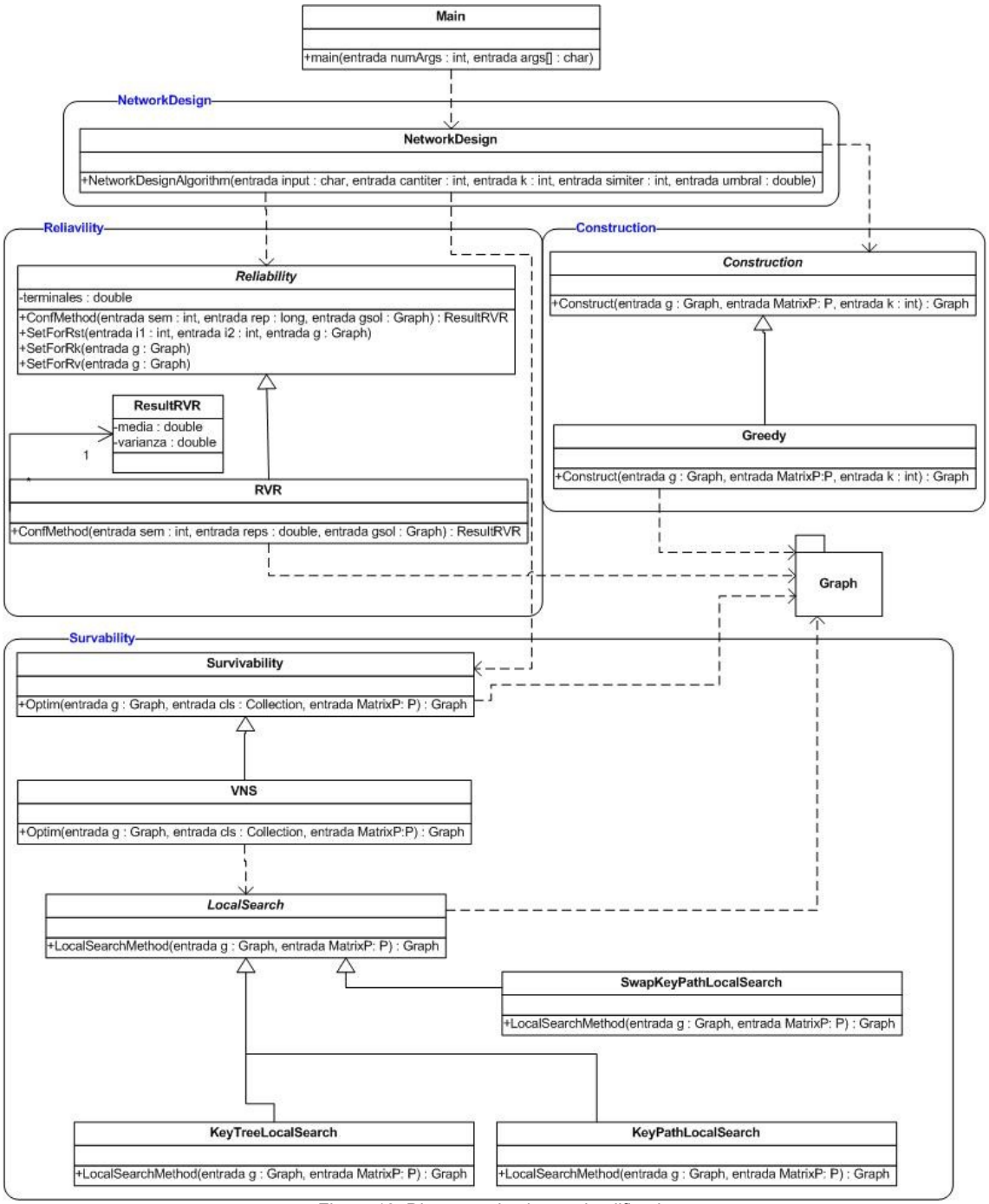


Figura 13: Diagrama de clases simplificado

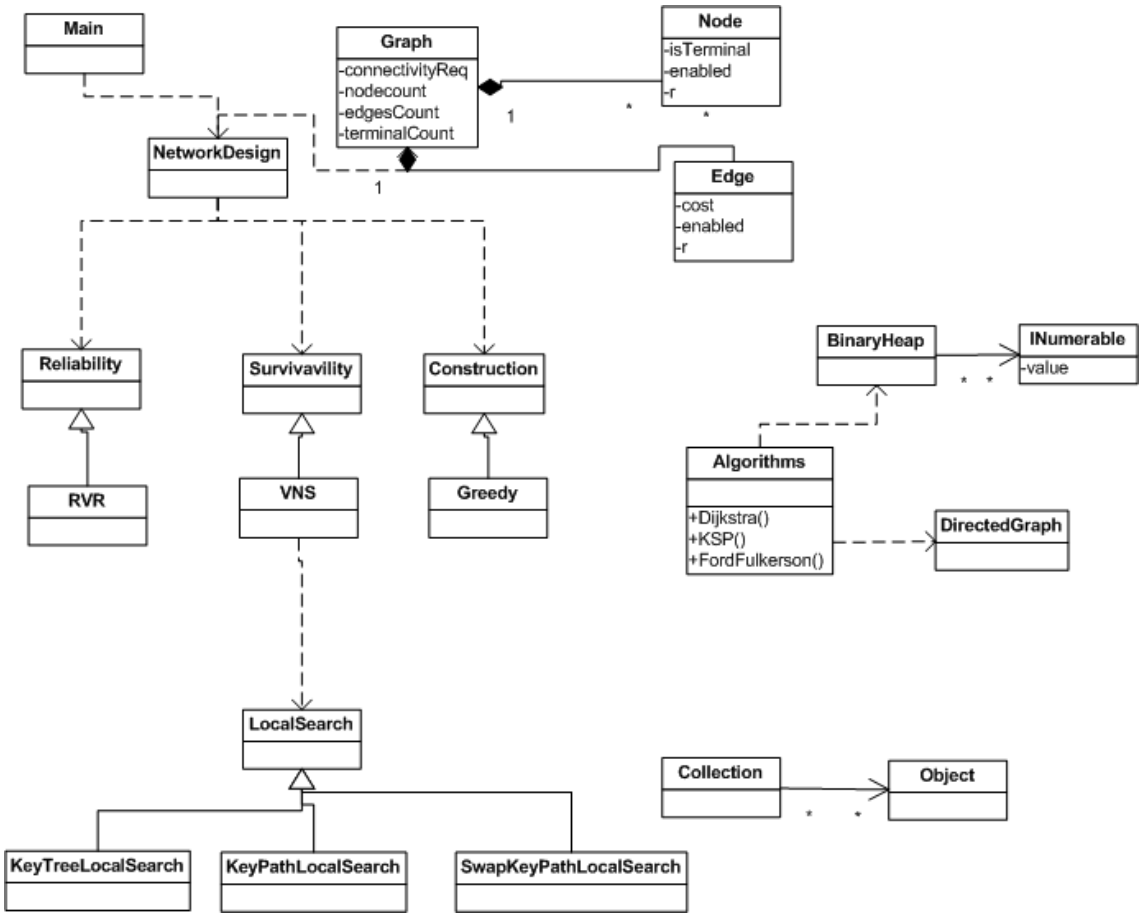


Figura 14: Diagrama de clases simplificado

Graph : Representa un grafo de Steiner, está asociado a las aristas, y a los vértices, y contiene información sobre los requerimientos de conexión entre nodos terminales. En esta clase se resuelve toda la lógica necesaria para manipular una red, como por ejemplo agregar aristas, desactivar nodos y aristas, calcular el costo de un grafo, y hasta averiguar si el grafo activo es una solución factible.

Adyacence: Representa la matriz de adyacencia del grafo.

Edge: Representa una arista del Grafo.

Node: Representa un nodo del Grafo.

Algorithms: Contiene algoritmos de uso general al proyecto, como por ejemplo Dijkstra 10, KSP 10, FordFulkerson 10, etc.

DirectedGraph: Grafo Dirigido utilizado para al algoritmo Ford Fulkerson 10.

Inumerable: Representa una interfaz para objetos que se pueden ordenar. Esta interfaz contiene la operación GetValue()

BinaryHeap: Montículo binario que permite tener una colección de objetos y obtener de la misma el mayor elemento, o el menor, en orden 1. Los elementos del montículo heredan de la interfaz INumerable, que representa un objeto numerable.

Collection: representa una colección de objetos heredados de la clase *Object* los objetos del tipo object deberán implementar la operación Equals para que se puedan ejecutar las operaciones de Remove, Delete, etc.

Object: Clase abstracta que representa los objetos que se almacenan en las colecciones.

LocalSearch: Interfaz común para todas las búsquedas locales, en ella se encuentra la operación de búsqueda local, y la misma es implementada por *KeyTreeLocalsearch*, *KeyPathLocalsearch*, *SwapKeyPathLocalSearch*. Esta interfaz deberá ser implementada en caso que se desee agregar una nueva búsqueda local.

KeyPathLocalSearch: Implementa la búsqueda local basada en sustitución de key-paths.

KeyTreeLocalSearch: Implementa la búsqueda local basada en sustitución de key-trees.

SwapKeyPathLocalSearch: Implementa la búsqueda local basada en intercambio de key-paths.

Construction: Interfaz para implementar el algoritmo de construcción de una solución factible, toma un grafo y devuelve una solución que sea factible aunque no deberá ser óptima con respecto a ningún esquema de vecindad.

Greedy: implementa el algoritmo de construcción de una solución factible, toma un grafo y devuelve una solución que sea factible aunque no deberá ser óptima con respecto a ningún esquema de vecindad. Este algoritmo construye una solución de forma voraz.

Survivability: Interfaz para construir el algoritmo de optimización de la solución factible encontrada por el algoritmo de construcción. En el entorno de este proyecto el algoritmo de optimización es *VNS* 10, este se podría cambiar para implementar otro como por ejemplo GRASP 10, etc.

VNS: Implementa el algoritmo de optimización.

Reliability: Interfaz para el algoritmo de confiabilidad, recibe un grafo y devuelve el valor de confiabilidad del mismo, en este proyecto utilizamos *RVR* 10 como algoritmo para encontrar la confiablidad de una red, pero se podría utilizar otro como por ejemplo MonteCarlo 10, etc.

RVR: Implementa el algoritmo de confiabilidad RVR 10, recibe un grafo y devuelve el valor de confiabilidad del mismo.

NetworkDesign: Esta clase contiene el algoritmo principal de este proyecto, en el mismo se utiliza el algoritmo *Greedy* 1010, *VNS* 10, y *RVR* 10 como los principales componentes.

Main: Algoritmo que comienza la ejecución del programa, contiene la lectura de los parámetros y la invocación al algoritmo principal.

La solución fue diseñada de forma de ser fácilmente extensible y customizable pensando en poder utilizarse como un framework para el desarrollo de metaheurísticas, de esta forma es posible utilizar diferentes algoritmos de confiabilidad, diferentes metaheurísticas y búsquedas locales. Para lograr esta cualidad en el software desarrollado se utilizo el patrón de diseño Strategy 10. Los mecanismos de extensión de la solución se explican en el Apéndice 6 – Mecanismos de Extensibilidad.

5.3.Especificación de las operaciones para las clases relevantes

Se asume que los nodos son identificados por números enteros en el rango $1 \ . \ . \ |V|$ y las aristas por pares de números enteros identificando los nodos extremos de las mismas.

Clase Graph
La clase *Graph* implementa el grafo estocástico.

```
//Constructores
Graph();
Graph(int);
//Destructor
virtual ~Graph(void);
//Realiza una copia del grafo
Graph * Copy();
//Obtiene la cantidad de nodos del grafo
int GetNodesCount();
//Obtiene la cantidad de aristas del grafo
int GetEdgesCount();
//Muestra el grafo en la consola
void Show();
//Activa un nodo
void EnableNode(node: in int);
//Desactiva un nodo
void DisableNode(node: in int);
//Desactiva el nodo y las aristas adyacentes de ser indicado
void DisableNode(node: in int, disableAdyacentEdges: in bool);
//Activa una arista
void EnableEdge(ext1: in int, ext2: in int);
//Desactiva una arista
void DisableEdge(ext1: in int, ext2: in int);
//Agrega una arista al grafo
void AddEdge(ext1: in int, ext2: in int);
//Agrega una arista al grafo, costo incluido
void AddEdge(ext1: in int, ext2: in int, cost: in double);
//Obtiene la probabilidad de falla de un nodo
double GetNodeProbability(node: in int);
//Obtiene la probabilidad de falla de una arista
double GetEdgeProbability(ext1: in int, ext2: in int);
//Obtiene el costo de una arista
double GetEdgeCost(ext1: in int, ext2: in int);
//Configura la probabilidad de falla de un nodo
void SetNodeProbability(node: in int, prob: in double);
//Configura la probabilidad de falla de una arista
void SetEdgeProbability(e1: in int,e2: in int, prob: in double);
//Seta el costo de una arista
void SetEdgeCost(ext1: in int, ext2: in int, cost: in double);
//Marca a un nodo como Terminal
void MarkTerminal(node: in int);
//Marca un nodo como No Terminal
void MarkNoTerminal(node: in int);
//Pregunta si un nodo es Terminal
bool IsTerminal(node: in int);
//Obtiene la colección de nodos adyacentes al que se le pasa
como parámetro.
//No se tiene en cuenta los nodos inactivos.
```

```
//El resultado es una colección de Objetos de tipo "Integer"
Collection * GetAdyacents(node: in int);
//Obtiene la colección de nodos terminales
//El resultado es una colección de Objetos de tipo "Integer"
Collection * GetTerminals();
//Obtiene la cantidad de nodos terminales
int GetTerminalsCount();
//Obtiene un nodo terminal
int GetTerminal();
//Pregunta si existe una arista entre un par de nodos
especificados como parámetros
bool ExistEdge(node1: in int, node2: in int);
//Pregunta si un nodo se encuentra activado
bool IsNodeEnabled(node: in int);
//Pregunta si una arista se encuentra activada
bool IsEdgeEnabled(ext1: in int, ext2: in int);
//Obtiene el grado de un nodo
int GetNodeDegree(node: in int);
//Muestra en la consola un resumen de los nodos y aristas
activos
void ShowEnableds();
//Activa todos los nodos y aristas del grafo
void EnableAll();
//Desactiva todos los nodos y aristas del grafo
void DisableAll();
//Activa todos los nodos Terminales
void EnableAllTerminals();
//Desactiva todos los nodos Terminales
void DisableAllTerminals();
//Obtiene los nodos activos
Collection * GetEnabledNodes();
//Obtiene las aristas activas
//Retorna una colección de "EdgeTypes"
Collection * GetEnabledEdges();
//Obtiene el grafo resultado de aplicar la operación \ entre un
grafo y un conjunto de nodos
//PRE: los elementos de la colección son de tipo "Integer"
void Rest(nodes: in Collection *);
//Agrega un keyTree al grafo
void Union(T: in KeyTree *);
//Agrega un camino al grafo
void Union(path: in Path *);
//Devuelve true si el nodo es un keynode, false en caso
contrario
bool IsKeyNode(node: in int);
//Obtiene los key nodes
//Retorna una colección de "Integers"
Collection * GetKeyNodes();
Collection * GetKeyNodes(bool);
//Obtiene el requerimiento de conexión entre un par de
terminales
int GetConnReq(terminal1: in int, terminal2: in int);
//Configura el requerimiento de conexión entre un par de nodos
void SetConnReq(node1: in int,node2: in int, req: in int);
//Obtiene el costo de un grafo
double GetCost();
//Carga un grafo
static Graph * LoadGraph(file: in string);
//Salva un grafo
static void SaveGraph(file: in string,graph: in Graph *);
```

```
//Contrae si es posible todo el grafo
void Contract();
//Contrae por una arista
int Contract(int, int);
//Da un corte extendido
ExtendedCut GetKExtendedCut();
//Retorna la probabilidad de que fallen todos
double AllFailedProb(ExtendedCut &d);
//Retorna el valor del atributo terminales del grafo
int GetTCount();
```

Clase NetworkDesign
La clase *NetworkDesign* implementa el algoritmo principal de la solución.

```
// Algoritmo principal
static void NetworkDesignAlgorithm(string input,int cantiter,int
k,double umbral, int simiter);
```

Clase Construction
Interfaz del Algoritmo de Construcción. Define el método que realiza la construcción de la solución factible.

```
// Define la construcción de la solución factible de G
virtual Graph * Construct(Graph * g, MatrixP *& p,int k)=0;
```

Clase Greedy
La clase Greedy implementa el algoritmo de construcción.

```
// Método que implementa la interfaz Construction
Graph * Construct(Graph * g, MatrixP *& p,int k);
// Algoritmo de Construccion
static Graph * GreedyConstruction(Graph *, MatrixP *&, int k);
// Deshabilita el camino p en G
static void DisablePath(Graph * g, Path * p);
// Deshabilita los caminos de pij en G
static void DisablePathsPij(Graph * g, MatrixP * p, int i, int
j);
// Sortea un par de nodos i,j
static Pair * Sortea(Collection *);
// Sorte un camino de la coleccion c
static Path * SorteaPath(Collection * c);
// Actualiza la matriz de costos
static void ActualizarCostos(Graph * , Graph * );
// Calcula nuevos caminos a partir del camino p
static void GeneralUpdateMatrix(Graph *G, MatrixP *P, MatrixM
*M, Path *p, int i, int j);
```

Clase Survivability
Interfaz del Algoritmo de Optimización. Define el método que realiza la búsqueda de la solución óptima.

```
// Define la búsqueda de la solución factible de G
virtual Graph * Optim(Graph * g, Collection * cls, MatrixP *
p)=0;
```

Clase VNS
La clase *VNS* implementa la metaheurística de la solución.

```
// Método que implementa la búsqueda de la solución factible
virtual Graph * Optim(Graph * g, Collection * cls, MatrixP * p);
// Metodo que implementa la metaheuristica
static Graph * VNSAlgorithm(Graph * g);
// Metodo que invoca la busqueda local identificada por k
static Graph* Search(Graph * g, int k);
```

Clase LocalSearch
Interfaz del Algoritmo de Búsqueda Local. Define el método que realiza la búsqueda local.

```
// Define la búsqueda de local
virtual Graph * LocalSearchMethod(Graph * g, MatrixP * p)=0;
```

Clase KeyPathLocalSearch
La clase *KeyPathLocalSearch* implementa la búsqueda local basada en key-paths.

```
// Realiza la búsqueda de local
virtual Graph * LocalSearchMethod(Graph * g, MatrixP * p);
// Busqueda local por key-path
Graph* KeyPathLS(Graph *gsol);
// Retorna la descomposicion de un grafo en key-paths
Collection* KeyPathDecomp(Graph *g);
// Chequea si restan por analizar key-paths de la coleccion
bool NotAnalizedKP(Collection *K);
// Sortea un indice de key-path que no haya sido analizado
int GetRandomIndex(Collection *analized);
```

Clase KeyTreeLocalSearch
La clase *KeyTreeLocalSearch* implementa la búsqueda local basada en sustitución de key-tree.

```
// Realiza la búsqueda de local
virtual Graph * LocalSearchMethod(Graph * g, MatrixP * p);
// Busqueda local basada en la estructra de vecindario key-tree
static Graph * KeyTreeLS(Graph * g);
// Calcula el grafo inducido por SGorro
static Graph * SubGraphInducedBySGorro(Graph * g, Collection *
Z, Collection * SGorro);
// Obtiene los nodos que no pertenecen a la solución
static Collection * GetNonSolutionNodes(Graph * g);
// Encuentra un key-tree substituto
static bool FindSubstituteKeyTree(Graph * g, int v, Collection *
STecho);
```

Clase SwapKeyPathLocalSearch
La clase Swap*KeyPathLocalSearch* implementa la búsqueda local basada en intercambio de key-paths.

```
// Realiza la búsqueda de local
virtual Graph * LocalSearchMethod(Graph * g, MatrixP * p);
// Busqueda local basada en intercambio de key-paths
static Graph * SwapEdgeLS(Graph * &g, MatrixP * &p);
// Devuelve true si existe un key-path sustituto
static bool   FindSubstituteKeyPath(Graph * &g, Path * path,
MatrixP * &p);
```

Clase Algorithms
La clase *Algorithms* implementa los algoritmos generales utilizados en las búsquedas locales y en el algoritmo de construcción.

```
//encuentra el camino menos costoso entre dos nodos de un grafo.
static Path * Dijkstra(Graph * g,int node1, int node2);
//Retorna true si el grafo es conectado
static bool IsConnected(Graph * g);
// implementa el algoritmo Ford Fulkerson
static double FFAlgorithm(DirectedGraph *,int,int);
// calcula los k caminos mas cortos entre s y t
static Collection * KSP(Graph * g, int s, int t, int k);
```

Clase Reliability
La clase *Reliability* representa las medidas de confiabilidad utilizadas en la solución y es la interfaz que define el método de confiabilidad utilizado.

```
// Define el método de confiabilidad
virtual ResultRVR ConfMethod(int sem, long reps, Graph *gsol)=0;
//Configura para calcular Rst, probabilidad de existencia de por
lo menos
//un camino entre los nodos s y t
void SetForRst(int, int, Graph * g);
//Configura para calcular Rk, probabilidad de existencia de por
lo menos
//un camino entre todo par de nodos del subconjunto K (nodos
terminales)
void SetForRk(Graph * g);
//Configura para calcular Rv, probabilidad de existencia de por
lo menos
//un camino entre todo par de nodos del grafo
void SetForRv(Graph * g);
```

Clase RVR
La clase *RVR* implementa el algoritmo de confiabilidad de la solución.

```
// Realiza el método de confiabilidad
virtual ResultRVR ConfMethod(int sem, long reps, Graph *gsol);
//Calcula la confiabilidad de una red mediante RVR.
//Parte de una semilla y un numero de iteraciones pasados por
parametro.
//Retorna el tiempo de referencia.
ResultRVR Conf(int, long, Graph*);
// Devuelve el resultado del ultimo calculo realizado.
ResultRVR GetResult();
//Quita n elementos del corte extendido del grafo
void Remove(Graph *, ExtendedCut &, int);
//Agrega con probabilidad uno el componente parametro al grafo
void Add(Graph *, Component);
// Ejecuta una replicacion para obtener un valor de la medida
// de la anti-confiabilidad (1 - confiabilidad).
double Rvr(Graph *);
//Sortea un elemento del corte
int GetRandomItem(ExtendedCut &);
//Chequea si la estructura esta operativa.
//Para estar operativa deben estar operativos todos los nodos
terminales y además existir al menos un camino entre todo par de
nodos terminales.
```

5.4.Diagrama de implementación

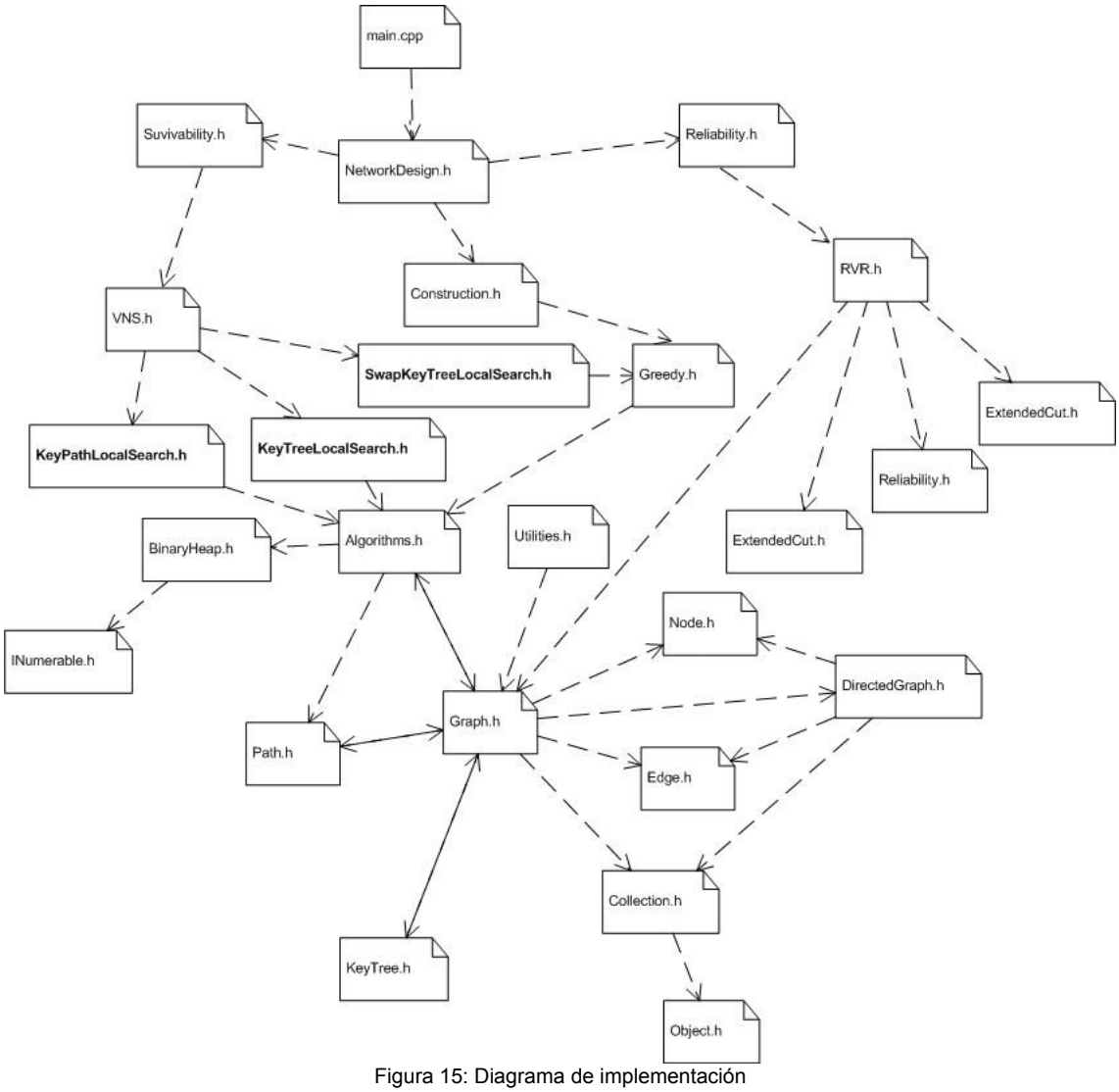


Figura 15: Diagrama de implementación

En este diagrama se describen las clases que han sido desarrolladas y las dependencias entre las mismas.

5.5.Implementación

En lo que respecta a implementación se optó por utilizar el lenguaje C++ teniendo como razón fundamental para ello que el mismo es un lenguaje orientado a objetos y que este facilita la realización de esfuerzos para lograr eficiencia en los algoritmos al permitir trabajar a bajo nivel con las estructuras de datos. Como principales desventajas se debe desarrollar todas las estructuras básicas y sus operaciones (i.e. listas, colecciones, iteradores) ya que el lenguaje no provee de APIs para el manejo de estas estructuras, el manejo de la memoria se debe realizar a bajo nivel lo que implica un mayor cuidado para no afectar la performance y cometer errores de acceso de memoria inválidos. Otra desventaja es que no se cuenta con facilidades para implementar interfaces graficas en caso de necesitarlas.

Los requerimientos básicos necesarios en la solución a este problema constan de la necesidad de representación del modelo matemático grafo estocástico, esto es, con probabilidades de operación asociadas tanto a sus nodos como a sus aristas (datos del problema).
Las aristas del grafo pueden ser recorridas en ambos sentidos por lo que el grafo es no dirigido.
El grafo carece de auto o multi-aristas, por ende estamos ante un grafo simple.
Debemos poder representar nodos terminales en el mismo, pudiendo identificar si un nodo perteneciente a un grafo es terminal mediante una operación debidamente implementada.
Es necesaria la disponibilidad de operaciones para agregar y quitar tanto aristas como nodos.

Estructuras de Datos

En casi todos los casos se utilizaron arreglos dinámicos (se determina su tamaño en tiempo de ejecución) con capacidad de redimensionamiento, para poder representar las estructuras de datos mas importantes como grafos (para estos se utilizo la representación de matriz), colecciones y matrices, debido a su simplicidad y eficiencia para acceder a los elementos conociendo su índice en el arreglo. En la implementación de algunos algoritmos se utilizaron otras estructuras como colas de prioridad (heaps). Las colecciones se implementaron de forma genérica para lograr una mayor reutilización.

Se utilizaron clases utilitarias para generar grafos de forma automática y aleatoria (casos de grafos con un alto número de nodos y aristas) y realizar chequeos de factibilidad, también se utilizaron clases para realizar testings funcionales y de performance de los algoritmos implementados.

Entre los algoritmos auxiliares mas importantes implementados se destacan Dijkstra 10,Ford Fulkerson 10, KSP (K Shortest Loopless Paths) 10 y DFS (Depth Forward Search).

A continuación se explican las estructuras utilizadas para el Grafo.

Nodos
El grafo mantiene un arreglo dinámico de punteros a nodos de largo n, siendo n igual a |V|. Cada nodo del arreglo dispone de la siguiente información:

- **enabled** (bool): Indica si el nodo está habilitado.
- **r** (double): Probabilidad de operación.
- **terminal** (bool): Indica si el nodo es Terminal.

La estructura es dinámica dado que el tamaño de esta es determinado cuando el grafo es cargado desde el archivo *xml* correspondiente.

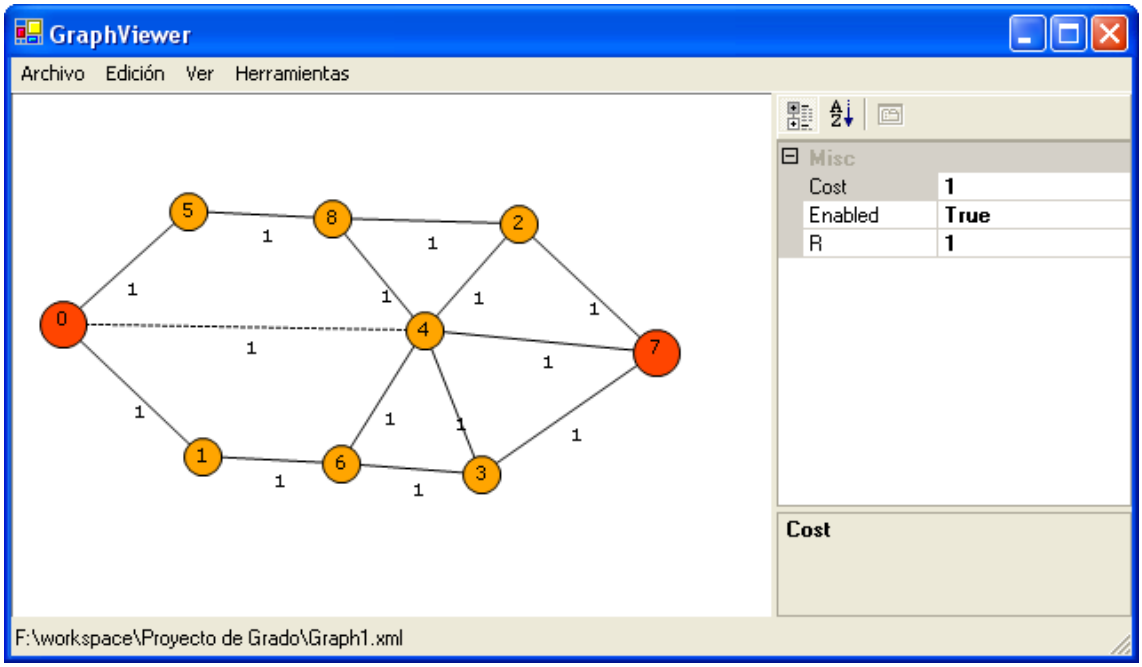
Aristas
La matriz de adyacencia es representada como una matriz cuadrada (*n* x *n*) de punteros a aristas. Dicha estructura está implementada en la clase **Adyacence**, la misma proporciona la funcionalidad para acceder a la arista correspondiente si son facilitados los identificadores de los nodos extremos. En caso de existir tal arista la información disponible es la siguiente:

- **cost** (double): Costo de la arista.
- **enabled** (bool): Indica si la arista está habilitada.
- **r** (double): Probabilidad de operación.

La elección de una matriz por sobre una lista para representar la estructura descripta se baso en la necesidad de eficiencia para las distintas operaciones que obtienen información u operan sobre las aristas a partir de los identificadores de sus nodos extremos. La misma es dinámica al igual que la estructura utilizada para los nodos. Para poder guardar el resultado de los algoritmos se utilizo un archivo con formato xml, se opto xml por ser un estándar para el formato de la información, en el "Apéndice 5 – Estructura de la información de un grafo, y visor del grafo" se explica el formato utilizado.

Interfaces Gráficas

Si bien el contar con interfaces graficas no fue un requerimiento en este proyecto, se creyó en la necesidad de contar con una herramienta que permitiera crear grafos de forma rápida y sencilla y al mismo tiempo visualizarlos para facilitar el proceso de pruebas de validación de los algoritmos. Por este motivo se desarrollo una herramienta gráfica (Graph Viewer) que permite crear grafos y grabarlos en archivos XML que luego pueden ser cargados por la aplicación y viceversa. Esta herramienta resulto útil a la hora de crear grafos de prueba y luego visualizar gráficamente los grafos solución obtenidos. Los detalles del funcionamiento de esta herramienta como la estructura del los archivos XML se explican en el "Apéndice 5 – Estructura de la información de un grafo, y visor del grafo".



Administración de Memoria

En este trabajo los algoritmos que se manejan son de complejidad computacional alta, debiéndose ejecutar los mismos en ciclos para lograr un alto numero de replicaciones, esto con una mala administración de memoria resultaría en una caída de perforrmance considerablemente importante. El lenguaje de programación utilizado no pertenece a la clase de lenguajes de cuarta generación, por lo que no posee una administración de memoria automática. Por estos motivos se debió realizar una administración manual de memoria adecuada para no dejar memoria "colgada".

Utilización del Software

La solución de software implementada se desarrollo sobre la plataforma Windows XP Professional. Al no ser un software destinado a usuarios finales, se prescindió de una interfaz gráfica y se decidió correr la aplicación desde una consola del sistema. No obstante se utilizo una herramienta grafica para diseñar los grafos, la cual se detalla en el Apéndice 5 – Estructura de la información de un grafo, y visor del grafo.
A continuación se detalla la forma de correr la aplicación junto con los parámetros necesarios.

```
NetworkDesignAlgorithm "GraphPath",cantiter,K,umbral,simiter
```

En Dónde:

- `NetworkDesignAlgorithm`: Es el ejecutable que implementa el algoritmo.
- `GraphPath`: Es la ruta a la red para la cual se quiere aplicar el algoritmo. Dentro de esta red se define además la matriz de requerimientos de conexión y la matriz de probabilidades de nodos y aristas.
- `cantiter`: Es la cantidad de replicaciones que se correrá el algoritmo.
- `K`: es la cantidad de caminos más cortos a ser calculados en el algoritmo de construcción.
- `umbral`: Es el nivel de confiabilidad que se utilizara para aceptar o rechazar las soluciones obtenidas en las distintas iteraciones.
- `simiter`: Es la cantidad de replicaciones que se utilizaran en la simulación que realiza el algoritmo RVR 101010.

Herramientas

Para el desarrollo de la aplicación se utilizó la plataforma Windows XP Professional, como IDE Eclipse SDK versión 3.1.1 en conjunto con la herramienta CDT versión 3.0 (*C/C++ Development Tools*), plug-in disponible para el IDE antes mencionado.
Como compilador fue escogido MSYS (versión 1.0.10), el cual forma parte del paquete de herramientas de desarrollo de libre acceso y distribución MinGW (versión 3.1). La elección se baso en la facilidad de configuración del mismo con el IDE Eclipse.
Para el desarrollo de la aplicación gráfica se utilizo el IDE VS.net y el lenguaje C#, esta elección se baso en la ventajas proporcionadas por este lenguaje para programar la interfaz gráfica junto a la disponibilidad de una API para el manejo de grafos implementada en este lenguaje.

6. Testeo

El objetivo de este capítulo es realizar las pruebas de validación de los diferentes algoritmos implementados en el proyecto, El capítulo se encuentra dividido en las siguientes secciones: “Pruebas de validación algoritmo Greedy” en esta sección se valida que la implementación realizada cumpla con el modelo descrito en “Algoritmo de Construcción”, en la sección 6.2 se valida cada una de las búsquedas locales contra sus respectivos modelos descritos en la sección “Algoritmos de Búsqueda Local”, y por último la sección “Pruebas de validación algoritmo RVR”, valida la implementación de RVR contra el modelo “RVR - Reducción Recursiva de Varianza”. Cabe destacar que en este capítulo no se presentan resultados de los algoritmos, los mismos se presentan en el capitulo “Resultados”, en este solamente se valida que la implementación esté realizada de acuerdo a los modelos.
Para probar los algoritmos se generaron casos de pruebas, para la validación de los algoritmos se utilizó un algoritmo que genera grafos ver “Apéndice 4 – Algoritmo de construcción de grafos de prueba”, de esta forma se logran casos de prueba de forma aleatoria y que no dependan del algoritmo a probar. Para obtener los resultados del algoritmo general se utilizaron casos de prueba conocidos obtenidos de la tsplib 10, para que le resulten más familiares al lector.

6.1.Pruebas de validación algoritmo Greedy

Para validar el algoritmo de construcción *Greedy* 1010 utilizado en la solución del problema y teniendo en cuenta que el mismo es no determinista se realizaron pruebas sobre grafos pequeños, una de las cuales a modo ilustrativo es presentada en esta sección. Los tests para los algoritmos mencionados así como para otros implementados se encuentran en la clase Test.h.
A continuación pueden observarse una serie de imágenes que ilustran el funcionamiento del algoritmo *Greedy* 1010. Para validar el mismo se realizaron numerosas pruebas sobre grafos pequeños, de forma de poder asegurar que el algoritmo operase de forma correcta. El grafo inicial (ver Figura 16) para este caso tiene un costo de 100.

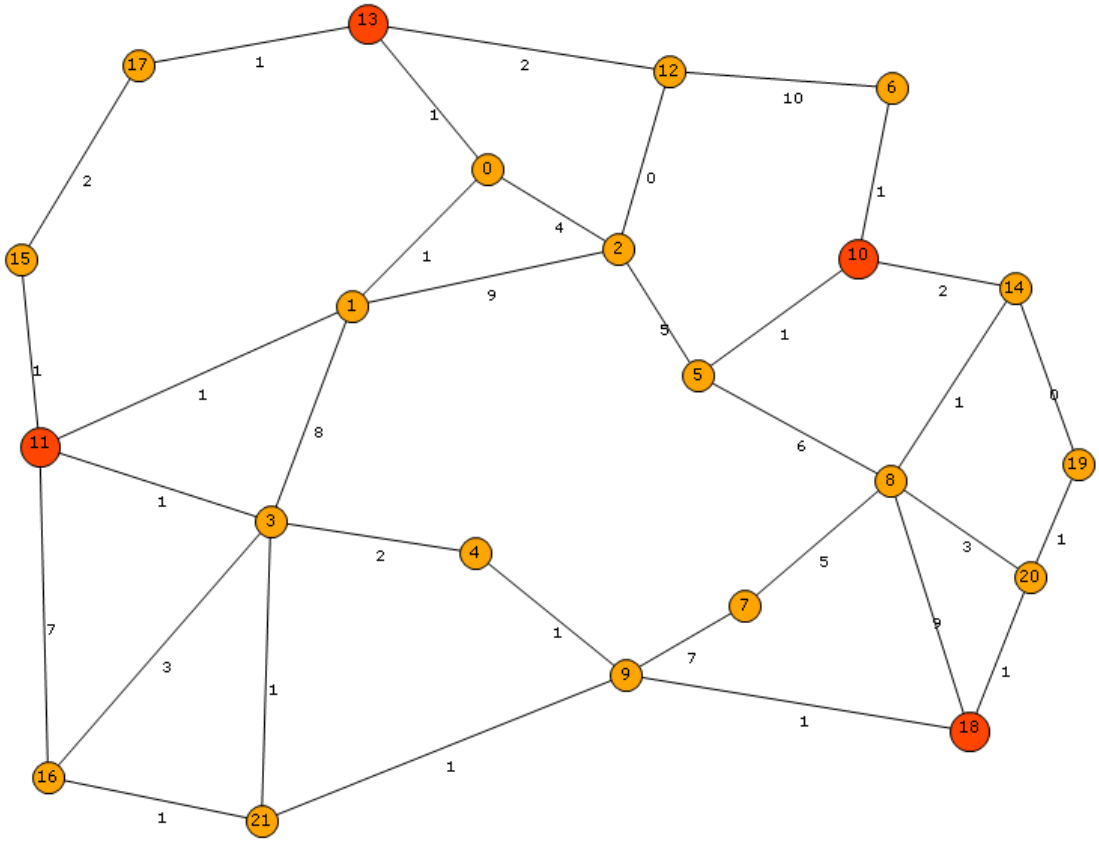


Figura 16: Grafo inicial Greedy test

Inicialmente el grafo solución se compone únicamente de los nodos terminales Figura 17, completándose de caminos nodos disjuntos entre pares de terminales por cada iteración. En la Figura 18 se puede ver como se construye el grafo solución agregando nuevos caminos nodo disjuntos.

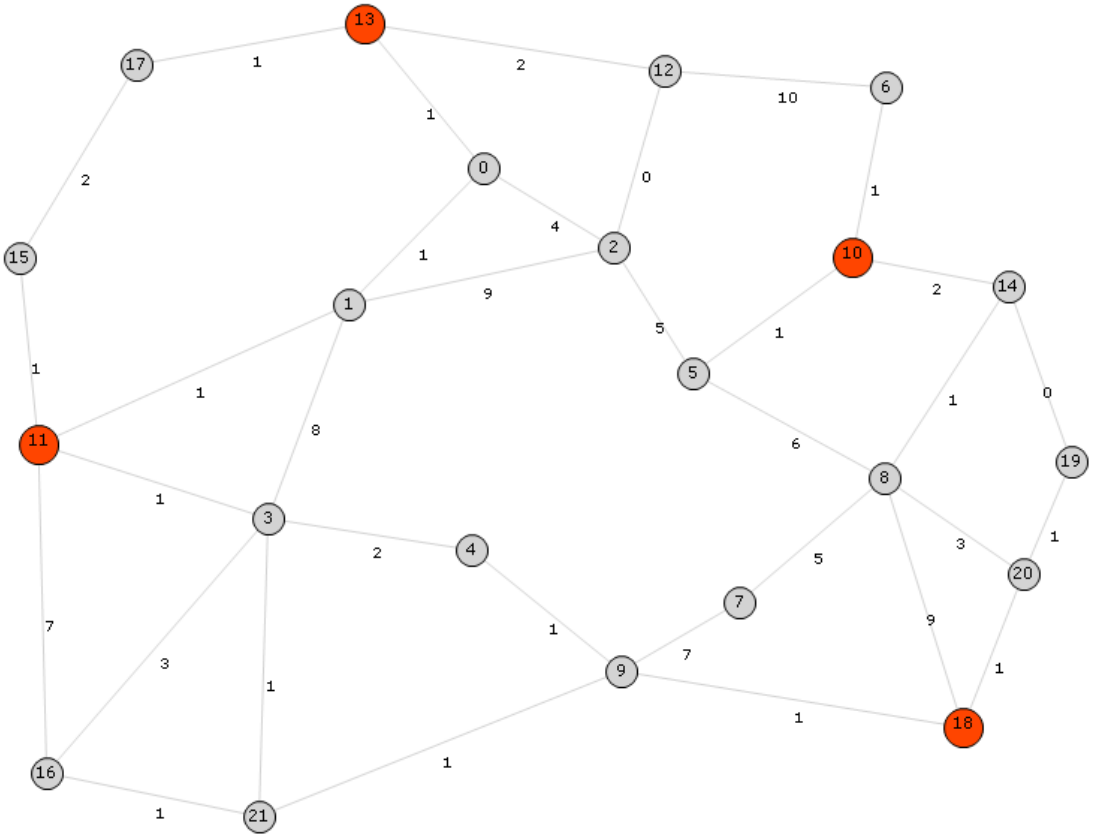


Figura 17: Greedy test

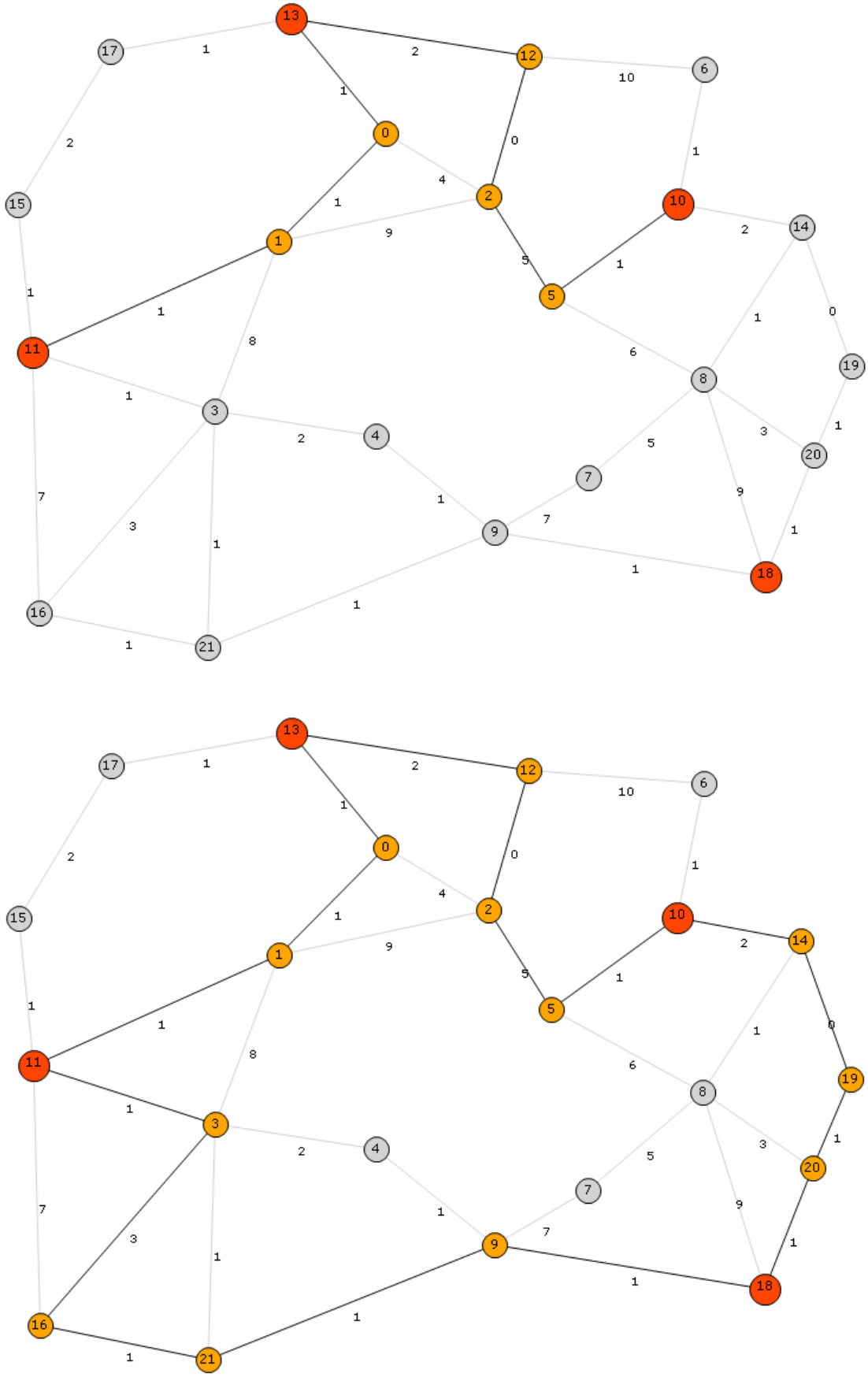


Figura 18: Greedy test

Al finalizar de ejecutarse el algoritmo de construcción *Greedy* 1010 se obtiene una solución factible de bajo costo (30), en donde se encontraron al menos dos caminos nodos disjuntos para cada par de terminales como se muestra en la Figura 19.

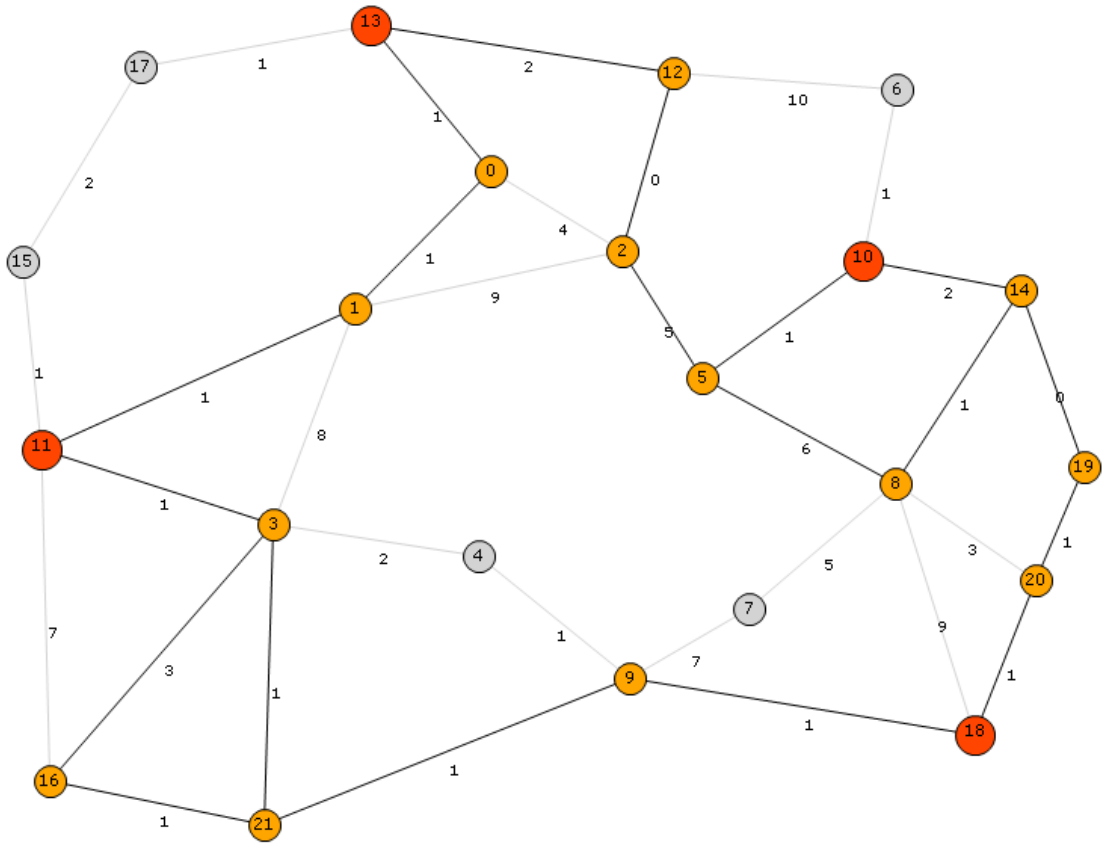


Figura 19: Greedy test

6.2.Pruebas de validación algoritmos de Búsqueda Local

Al igual que para el algoritmo anterior para validar los diferentes algoritmos de búsqueda local utilizados en la solución al problema y teniendo en cuenta que estos son no determinísticos se realizaron pruebas sobre grafos pequeños, algunas de las cuales a modo ilustrativo son presentadas en esta sección. Los tests para los algoritmos mencionados se encuentran en la clase *Test.h*.

KeyTreeLocalSearch

A continuación pueden observarse una serie de imágenes que ilustran el funcionamiento del algoritmo KeyTreeLocalSearch. Para validar el mismo se realizaron numerosas pruebas sobre grafos pequeños, de forma de poder asegurar que el algoritmo operase de forma correcta. El grafo inicial (ver Figura 20) para este caso tiene un costo de 54.

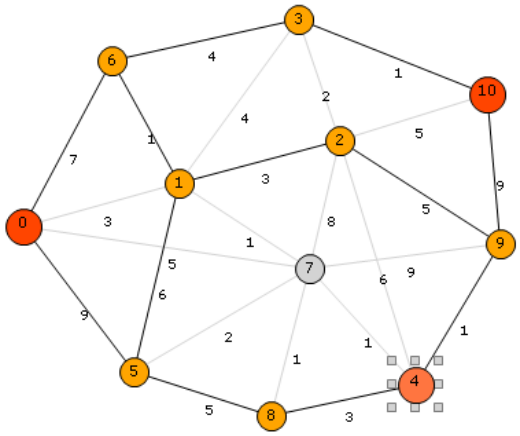


Figura 20: Grafo inicial KeyPathLocalSearch test

El primer nodo seleccionado es el “1”, como puede ser observado en la Figura 21. El *keytree* ha sido resaltado y es el que se muestra en la imagen mencionada. Posteriormente este es sustituido por el *keytree* indicado en la Figura 22.

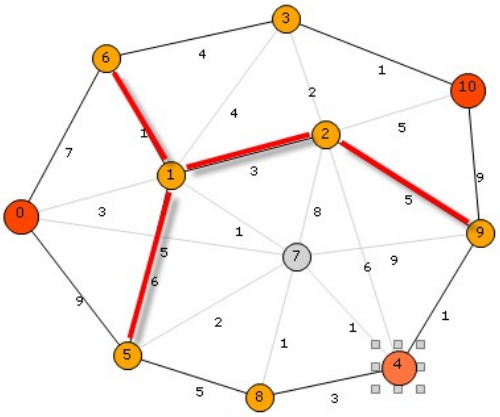


Figura 21: KeyPathLocalSearch test, keytree

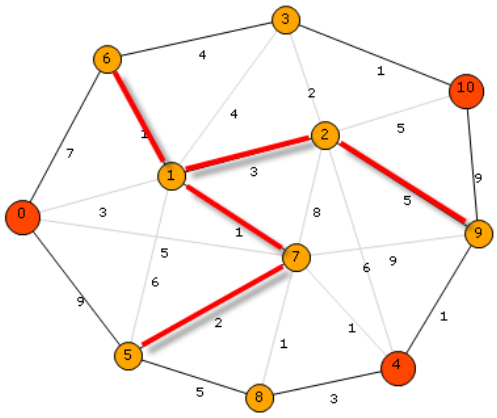


Figura 22: KeyTree Sustituto

Una vez realizada esta sustitución el costo del grafo es de 51. El segundo nodo seleccionado por el algoritmo es el “5”. Para este caso el *keytree* es el siguiente:

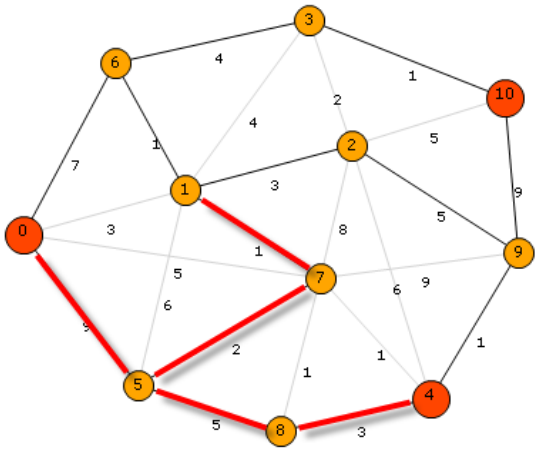


Figura 23: KeyTree con raíz "5"

En este caso se produce la contracción del nodo “5”, ya que todas las ramas del *keytree* salen directamente hacia el nodo “7”, por ende se elimina el nodo “5”. El resultado es el ilustrado en la figura siguiente.

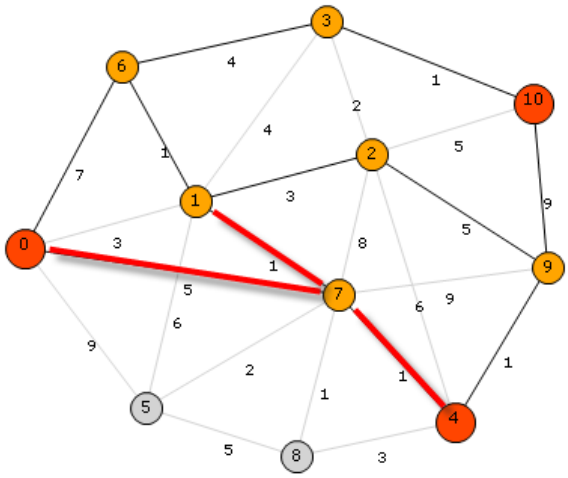


Figura 24: Resultado de eliminación del nodo "5"

En los pasos sucesivos no se producen nuevas mejoras al grafo, siendo el costo final del mismo de 38.

KeyPathLocalSearch

Sobre el mismo grafo inicial aplicamos la búsqueda local KeyPathLocalSearch.

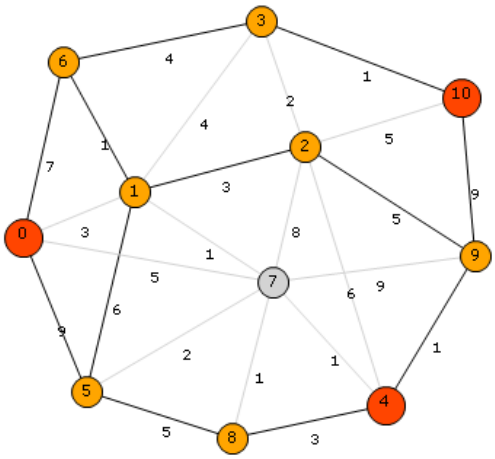


Figura 25: Grafo inicial KeyPathLocalSearch test

El costo del grafo inicial es de 54, pudiendo ser sustituido únicamente un *keypath* como se observa en la Figura 26. El costo final del grafo es de 49.

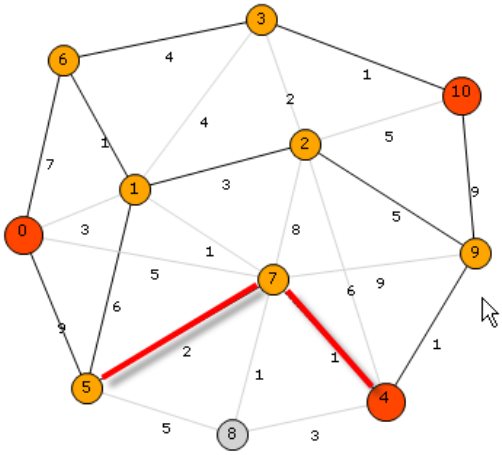


Figura 26: Grafo obtenido de sustituir el keypath

SwapKeyPathLocalSearch

Para realizar las pruebas de este algoritmo tenemos que tener en cuenta que en esta búsqueda local es necesario considerar la matriz de requerimientos de conexión ya que es la única búsqueda que la utiliza, dado que el resto no elimina redundancias sino que se limita a conservar los requerimientos de conexión de la solución inicial. En esta búsqueda se podrán eliminar key-paths si se encuentra que los mismos son innecesarios para conservar la factibilidad.

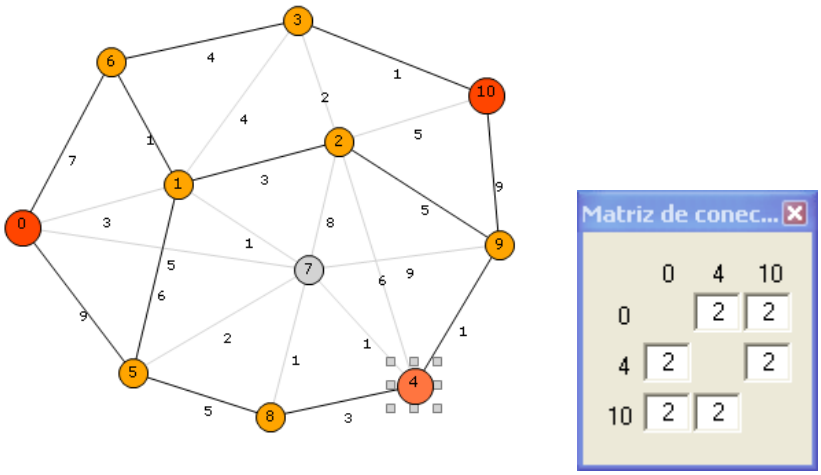


Figura 27: Grafo inicial SwapKeyPathLocalSearch test

Como podemos observar en la Figura 27 y al igual que en las pruebas anteriores el costo del grafo inicial para este test es de 54 siendo el requerimiento de conexión entre los nodos terminales 2 para todos los casos.

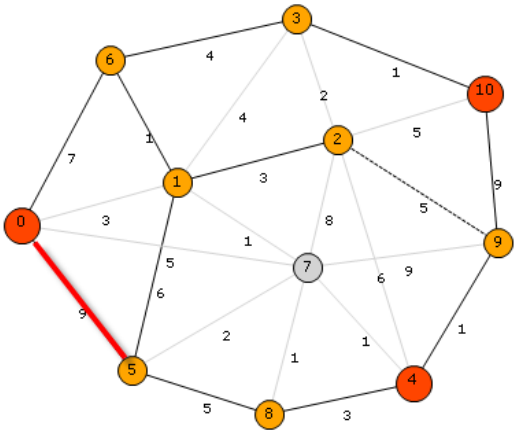


Figura 28: KeyPath a sustituir

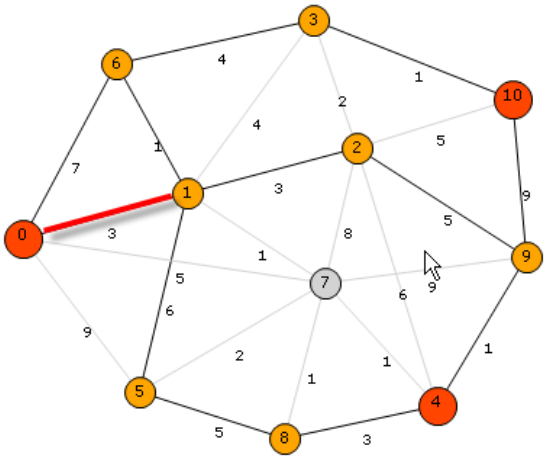


Figura 29: Arista sustituida

En la Figura 28 puede ser apreciado el key-path [0,5], el cual el algoritmo sustituye por la arista [0,1]. El costo del grafo luego de la mencionada sustitución es de 48.

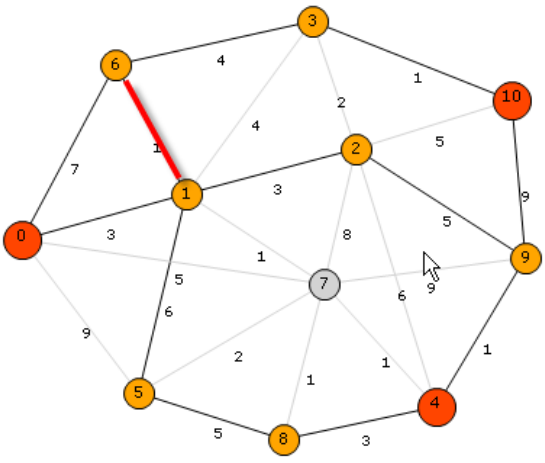


Figura 30: KeyPath redundante

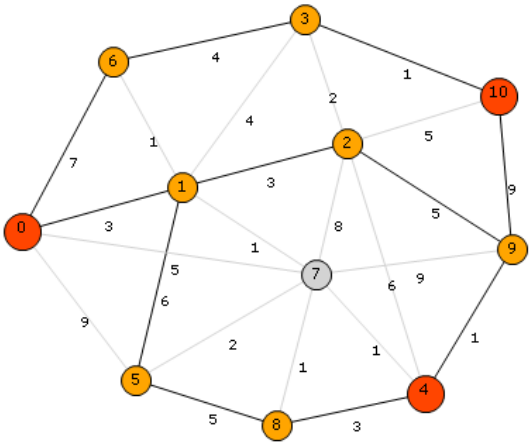


Figura 31: Grafo resultante

La Figura 30 muestra el key-path [6,1], el cual es eliminado resultando en el grafo que se aprecia en la Figura 31, cuyo costo es de 47.

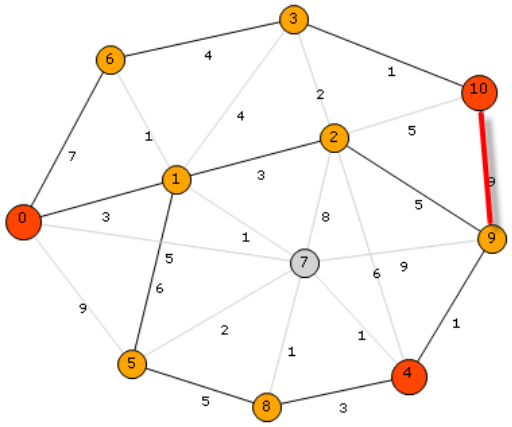


Figura 32: KeyPath a sustituir

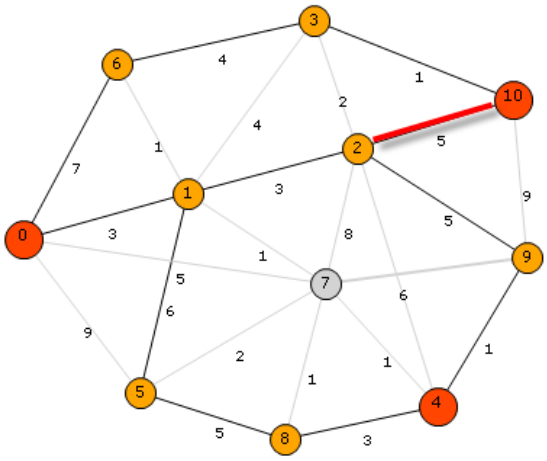


Figura 33: Arista sustituta

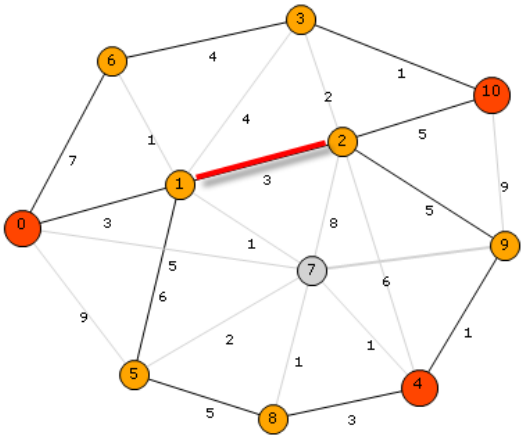


Figura 34: KeyPath redundante

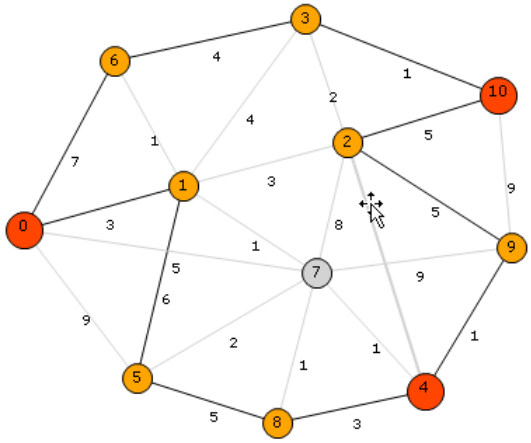


Figura 35: Grafo resultante

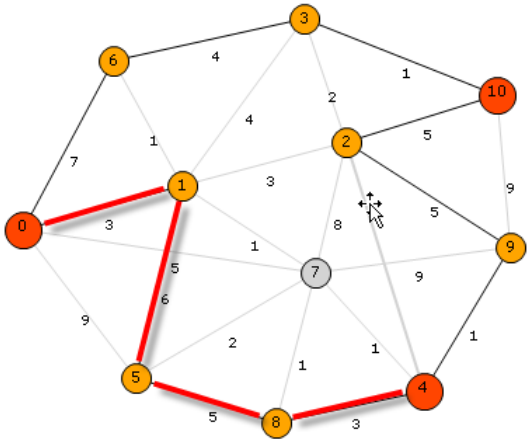


Figura 36: KeyPath a sustituir

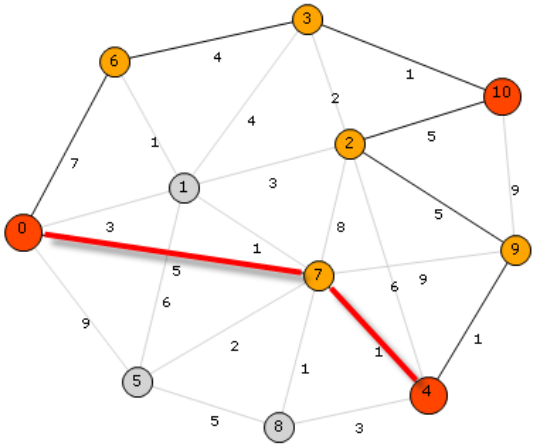


Figura 37: Arista sustituta

Posteriormente el algoritmo elimina el key-path [9,10] y agrega la arista [2,10] (ver Figura 32 y Figura 33) siendo el costo parcial del grafo resultante de 43.
El key-path [1,2] (Figura 34) es eliminado decrementando el costo del grafo a 40. Finalmente el key-path [0,1,5,8,4] es sustituido por el camino [0,7,4] como se ilustra en las Figura 36 y Figura 37. El costo final del grafo es de 29.
Como podemos observar el costo obtenido en esta búsqueda local es mucho menor que con el resto de las búsquedas, ya que se eliminaron las redundancias en la matriz de requerimientos preservándose la factibilidad.

6.3.Pruebas de validación algoritmo RVR

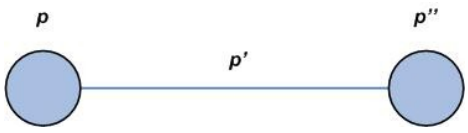
El objetivo de esta sección es presentar los casos de prueba utilizados para validar la implementación del algoritmo RVR. Se presentan las topologías, valores de confiabilidad de los distintos componentes y los resultados obtenidos.
Como precondición a esta sección se tiene que el software desarrollado esta libre de errores en tiempo de ejecución, no descartándose la existencia de errores de origen lógico los cuales deberán ser detectados al aplicar las pruebas descriptas a continuación.
La estrategia a seguir es utilizar redes pequeñas, calcular el índice de confiabilidad de forma manual y comparar los resultados con los proporcionados al ejecutar el algoritmo implementado. Las pruebas de validación para RVR se basan en la sección del mismo nombre presente en 10.

Medidas de Interés

- $R_v : G \rightarrow [0,1]$ (confiabilidad global): Probabilidad de existencia de por lo menos un camino entre todo par de nodos del grafo.
- $R_{st} : G \times V \times V \rightarrow [0,1]$ (confiabilidad fuente-terminal): Probabilidad de existencia de por lo menos un camino entre los nodos **s** y **t**.
- $R_K : G \times K \subseteq V \rightarrow [0,1]$ (confiabilidad entre terminales): Probabilidad de existencia de por lo menos un camino entre todo par de nodos del subconjunto ***K*** de nodos terminales.

Redes Consideradas

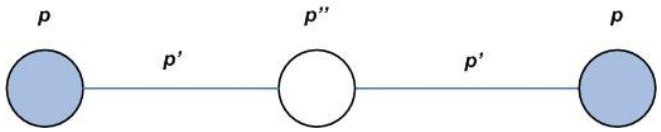
Caso 1
Grafo compuesto por dos nodos terminales y una única arista que los une. En este caso interesa la medida R_{st} que se calcula como ***pp'p''***.



Grafo completo de 2 nodos

Valores	Analítico	RVR	
		R	Var
p=p'=p''=0.9	0.729	0.729	3.13631e-017

Caso 2
Grafo compuesto por tres nodos, de los cuales dos son terminales, y dos aristas. La medida R_{st} para el mismo se calcula como $p^2p'^2p''$.

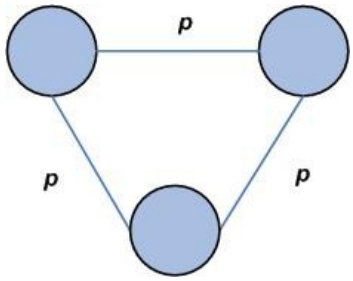


Grafo de 3 nodos

Valores	Analítico	RVR	
		R	Var
p=p'=p''=0.9	0.59049	0.593354	3.3341e-006

Para todas las pruebas siguientes se considerarán terminales perfectos (probabilidad de funcionamiento igual a 1).

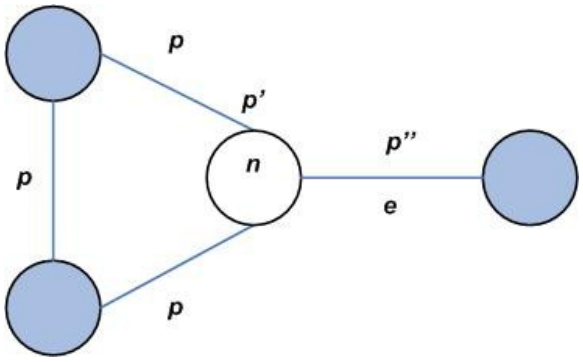
Caso 3: K_3
Grafo completo de 3 nodos, donde todos son terminales (en este caso interesa la medida R_v). Se considera una misma probabilidad *p* de funcionamiento para todas las aristas. Un grafo completo con probabilidades altas de funcionamiento tanto en aristas como en nodos es una red altamente confiable. La confiabilidad para este grafo se puede calcular en forma analítica mediante la expresión $p^3 + 3(1 - p)p^2$, donde el primer término de la suma corresponde a la probabilidad del suceso *"todas las aristas funcionan"* y el segundo a la probabilidad del suceso *"todas las aristas funcionan menos una"* (para cada arista).



Grafo completo de 3 nodos

Valores	Analítico	RVR	
		R	Var
p=0.9	0.972	0.972473	6.21454e-008

Caso 4: Red con puente y punto de articulación
Se considera la siguiente red (donde interesa la medida R_k) que presenta un puente (arista *e*) y un punto de articulación (nodo *n*).

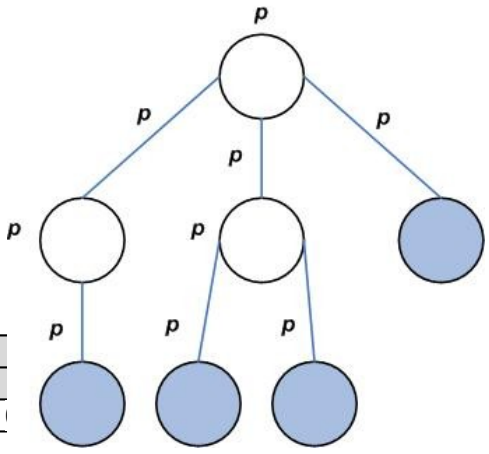


Red con puente y punto de articulación

La confiabilidad de esta red depende fuertemente de las probabilidades de funcionamiento del nodo *n* y de la arista *e*, como se ve reflejado en la siguiente expresión obtenida para su cálculo en forma analítica: $R_k = p''p'[p^3 + 3(1 - p)p^2]$

Valores	Analítico	RVR	
		R	Var
p=0.9 p'=0.5 p''=0.9	0.4374	0.438066	3.18052e-007
p=0.9 p'=0.9 p''=0.5	0.4374	0.437479	4.32176e-007

Caso 5: Árbol
Se considera un árbol donde los nodos terminales son las hojas (en este caso interesa la medida R_k), con probabilidad de funcionamiento *p* para todas las aristas y nodos que no son terminales. Un árbol tendrá valores bajos de confiabilidad pues la falla de cualquier nodo o arista desconecta cualquier par de nodos de la red. La confiabilidad de una red con topología de árbol se calcula como el producto de las probabilidades de funcionamiento de cada uno de sus componentes.

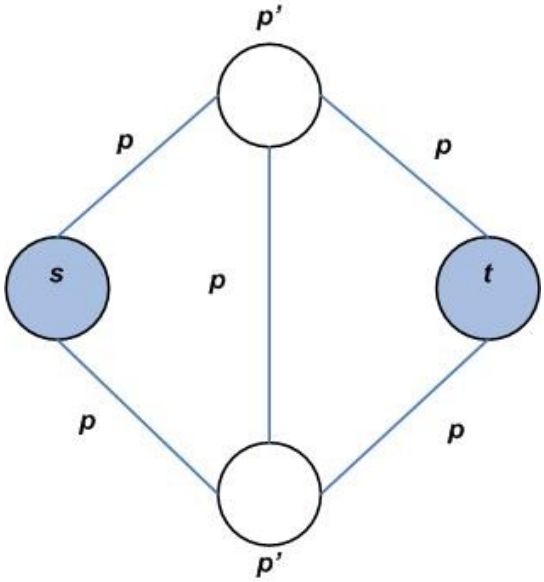


Valores	Analítico
p=0.9	0.387420489

Caso 6: Grafo puente

En este caso la medida de interés es R_{st}. La confiabilidad de esta red depende fuertemente del valor de *p*', que es la probabilidad de funcionamiento de los nodos intermedios en los caminos de ***s*** a ***t***. La expresión analítica obtenida para su cálculo es la siguiente:

$$R_{st} = pp'p + (1-p)pp'p + (1-p')ppp'p + (1-p)p'ppp'p + pp'(1-p)(1-p)p'pp + (1-p)p'ppp'(1-p)p$$



Grafo puente

Valores	Analítico	RVR	
		R	Var
p=0.9 p'=0.5	0.64962	0.654653	6.7791e-006
p=0.9 p'=0.9	0.9383688	0.937484	1.89235e-006

Sobre los resultados

Se realizaron diferentes pruebas con valores particulares tanto para las probabilidades de funcionamiento de nodos como de aristas. Se detectaron errores en las pruebas de validación de RVR, los cuales fueron corregidos mediante modificaciones de la lógica del algoritmo implementado.

En lo que respecta al algoritmo en si, el mismo trabaja con número reales en doble precisión pudiendo tener hasta 15 dígitos significativos.

El número de replicaciones llevadas a cabo para las pruebas del algoritmo fue de 10^4.

7. Resultados

Esta sección introduce los resultados experimentales obtenidos al aplicar el algoritmo *NetworkDesign*, algoritmo que engloba la fase de construcción mediante el algoritmo *Greedy*, el uso de la metaheurística *VNS* (la cual incluye las búsquedas locales *KeyTreeLocalSearch*, *KeyPathLocalSearch* y *SwapKeyPathLocalSearch*) y la aplicación de *RVR* para el cálculo de la confiabilidad del grafo resultante. La medida de interés para RVR es en este caso *Rk* (confiabilidad entre terminales), siendo esta la probabilidad de existencia de por lo menos un camino entre todo par de nodos del subconjunto ***K*** de nodos terminales.
Los resultados experimentales fueron obtenidos en un Pentium IV 3.0 GHz (Processor 3E GHz 1 MB L2-Cache 800 MHz FSB) y 1 Gbytes de RAM (DDR400) corriendo bajo Windows XP Professional Edition.
Para todos los casos el parámetro *k* para el algoritmo *Greedy* toma el valor 5 y el umbral de confiabilidad establecido es 60%. El valor del parámetro *k* se establece en base a pruebas preliminares realizadas con grafos aleatorios generados mediante el algoritmo de construcción de grafos de prueba (ver 11.4 *Algoritmo de construcción de grafos de prueba*) en las cuales se experimentó con diferentes valores para el mismo. El valor finalmente seleccionado mostró resultados aceptables en dichas pruebas.
En lo que respecta al umbral de confiabilidad el mismo fue establecido entendiendo que estando el presente trabajo orientado al diseño de redes de alta confiabilidad carece de sentido conservar en un set de soluciones posibles una red cuya confiabilidad global no supere un mínimo de 60%.
Los valores de confiabilidades elementales para nodos y aristas serán en todos los casos cercanos a uno fundamentándose dicha elección en que el problema objetivo de la presente tesis es aplicable a redes altamente confiables.

7.1.Descripción del Test Set

Por lo relevado en la bibliografía, una posibilidad para conformar los conjuntos de prueba o test-sets para el testing computacional de la algoritmia generada es generar grafos aleatorios. Ese enfoque fue utilizado para las pruebas realizadas sobre los distintos algoritmos que conforman la solución desarrollada y para pruebas preliminares del algoritmo principal *NetworkDesign*. Para las pruebas formales sobre este último se decidió utilizar casos para el Travelling Salesman Problem (TSP) extraídos de la librería TSPLIB 10. Esta decisión se fundamenta en que hasta donde pudimos investigar y es de nuestro conocimiento no existen casos benchmark para el problema en estudio, por lo que se opto por adaptar casos de una librería conocida y de fácil acceso.
Se seleccionaron instancias de esta bien conocida librería, modificando estas para que los grafos resultantes fuesen completos, siendo los costos de las distintas aristas el resultado del cálculo de la distancia euclidiana correspondiente.
Las instancias TSP seleccionadas fueron: att48, berlin52, brazil58, ch150, d198, eil51, gr137, gr202, kroA100, kroA150, kroB100, kroB150, kroB200, lin105, pr152, rat195, st70, tsp225, u159, rd100. El número de nodos de la instancia esta definido en el nombre de la misma (Ej: kroA100 contiene 100 nodos).
La primera columna de la Tabla 1 indica el nombre de la instancia, el resto de las columnas de izquierda a derecha indican:

- **% T:** Indica el porcentaje de nodos terminales sobre los nodos totales del grafo. Para el test-set conformado utilizamos los valores 20%, 35% y 50% generando así 3 instancias diferentes por cada problema TSP seleccionado.

- **% Conf:** Indica los valores de confiabilidad utilizados para nodos y aristas en ese orden.

- **% ReqConex:** Esta columna presenta tres valores por tupla, siendo cada valor el porcentaje de requeriremos r_{ij} entre los nodos terminales i y j para los cuales queremos 2, 3 y 4 caminos nodo-disjuntos respectivamente.

- **Iter ND:** Número de iteraciones a realizar por el algoritmo *NetworkDesign* por instancia según %T.

- **Iter RVR:** Número de iteraciones a realizar por el algoritmo *RVR*.

- **Instancias:** Número de instancias generadas.

Vale aclarar que el cálculo de confiabilidad y por ende la verificación de satisfacción del umbral establecido será llevado a cabo únicamente en un subconjunto de las instancias de prueba establecidas, siendo expuestos los resultados correspondientes en la sección que sigue. Se indicará como "No aplica" a aquellos casos en donde la verificación de satisfacción del umbral de confiabilidad establecido no sea realizada. Por otro lado la construcción *Greedy* y la aplicación de *VNS* se llevará a cabo sobre el test-set completo. Motiva dicha restricción los tiempos computacionales elevados (horas por iteración) en los que se incurre al aplicar *Greedy-VNS-RVR* sobre redes de tamaño considerable (mas de 100 nodos).
Las instancias de test cuyos nombres incluyen (E) son variantes de las primeras con diferentes requerimientos de conexión.
El número de iteraciones a realizar por el algoritmo *NetworkDesign* (**Iter ND**) se establece en 100 para aquellas instancias en que los tiempos computacionales son relativamente cortos (minutos) y en 20 para aquellas en que estos tiempos se incrementan considerablemente.
En lo relativo al número de iteraciones a realizar por el algoritmo *RVR* (**Iter RVR**) establecemos el mismo en $_{10}4$, teniendo como base a resultados obtenidos en pruebas preliminares.

Problema	% T	% Conf	%ReqConex	Iter ND	Iter RVR	Instancias
att48	20-35-50	99-95	100-0-0	100-100-100	10000	3
berlin52	20-35-50	99-95	100-0-0	100-100-100	10000	3
brazil58	20-35-50	99-95	100-0-0	100-100-100	10000	3
ch150	20-35-50	99-95	100-0-0	20-20-20	No aplica	3
d198	20-35-50	99-95	100-0-0	20-20-20	No aplica	3
eil51	20-35-50	99-95	100-0-0	100-100-100	10000	3
gr137	20-35-50	99-95	100-0-0	100-20-20	No aplica	3
gr202	20-35-50	99-95	100-0-0	20-20-20	No aplica	3
kroA100	20-35-50	99-95	100-0-0	100-100-100	No aplica	3
kroA150	20-35-50	99-95	100-0-0	100-20-20	No aplica	3
kroB100	20-35-50	99-95	100-0-0	100-100-100	No aplica	3
kroB150	20-35-50	99-95	100-0-0	100-20-20	No aplica	3
kroB200	20-35-50	99-95	100-0-0	20-20-20	No aplica	3
lin105	20-35-50	99-95	100-0-0	100-100-100	No aplica	3
pr152	20-35-50	99-95	100-0-0	20-20-20	No aplica	3
rat195	20-35-50	99-95	100-0-0	20-20-20	No aplica	3
st70	20-35-50	99-95	100-0-0	100-100-100	10000	3
tsp225	20-35-50	99-95	100-0-0	20-20-20	No aplica	3
u159	20-35-50	99-95	100-0-0	20-20-20	No aplica	3
rd100	20-35-50	99-95	100-0-0	100	No aplica	3
berlin52(E)	20	99-90	65-25-10	100	10000	1
eil51(E)	20	99-90	65-25-10	100	10000	1
att48(E)	35	99-90	65-25-10	100	10000	1
st70(E)	35	99-90	65-25-10	100	10000	1
brazil58(E)	50	99-90	65-25-10	100	10000	1
eil51(E)	50	99-90	65-25-10	100	10000	1
kroB100(E)	20	99-90	65-25-10	100	No aplica	1
lin105(E)	20	99-90	65-25-10	100	No aplica	1
kroA100(E)	35	99-90	65-25-10	20	No aplica	1
rd100(E)	35	99-90	65-25-10	20	No aplica	1

Tabla 1: Test Set

7.2.Resultados Numéricos

Observemos ahora los resultados numéricos obtenidos. La Tabla 2 indica los resultados para cada instancia TSP generada. La primera columna indica el nombre de la instancia, la segunda el porcentaje de nodos terminales de la misma, el resto de las columnas de izquierda a derecha indican:

- **%MG:** Porcentaje de mejora del algoritmo *Greedy* por sobre el costo inicial de la instancia.

- **%MVNS:** Porcentaje de mejora del algoritmo *VNS* por sobre el costo resultante luego de la aplicación del algoritmo de construcción *Greedy*.

- **TP-I:** Tiempo promedio por iteración de *NetworkDesign*.

- **%R-Conf:** Confiabilidad promedio obtenida.

- **VAR:** Varianza promedio obtenida.

Problema	%T	%MG	%MVNS	TP-I	%R-Conf	VAR
att48	20	99,27	34,61	11,466	96,7	7,608E-07
att48	35	98,60	36,83	29,769	94,3	3,448E-06
att48	50	98,22	37,10	65,904	92,7	5,322E-06
berlin52	20	98,98	30,55	30,605	93,7	3,294E-06
berlin52	35	99,06	33,93	33,433	93,8	3,190E-06
berlin52	50	98,02	33,48	106,945	90,7	6,487E-06
brazil58	20	98,92	31,96	62,377	88,5	6,722E-06
brazil58	35	99,25	39,45	68,891	86,0	8,347E-06
brazil58	50	98,75	35,26	103,553	91,0	7,093E-06
ch150	20	99,76	37,51	222,552	No aplica	No aplica
ch150	35	99,72	36,65	546,652	No aplica	No aplica
ch150	50	99,69	34,42	1.203,054	No aplica	No aplica
d198	20	99,90	32,22	320,142	No aplica	No aplica
d198	35	99,86	34,12	2.086,376	No aplica	No aplica
d198	50	99,81	33,39	5.548,639	No aplica	No aplica
eil51	20	99,34	38,79	14,870	96,0	1,183E-06
eil51	35	98,54	36,11	39,017	94,2	3,736E-06
eil51	50	98,56	37,32	44,798	93,7	4,284E-06
gr137	20	99,79	36,31	137,496	No aplica	No aplica
gr137	35	99,71	34,18	404,061	No aplica	No aplica
gr137	50	99,68	34,61	976,369	No aplica	No aplica
gr202	20	99,89	32,43	528,162	No aplica	No aplica
gr202	35	99,75	34,56	3.511,698	No aplica	No aplica
gr202	50	99,74	33,36	9.505,629	No aplica	No aplica
kroA100	20	99,61	36,77	44,225	No aplica	No aplica
kroA100	35	99,53	38,23	101,498	No aplica	No aplica
kroA100	50	99,45	35,89	280,833	No aplica	No aplica
kroA150	20	99,83	36,70	102,712	No aplica	No aplica
kroA150	35	99,75	36,30	412,970	No aplica	No aplica
kroA150	50	99,70	32,32	2.035,062	No aplica	No aplica
kroB100	20	99,68	38,71	17,301	No aplica	No aplica
kroB100	35	99,59	36,32	53,740	No aplica	No aplica
kroB100	50	99,49	34,98	191,722	No aplica	No aplica
kroB150	20	99,84	37,49	112,099	No aplica	No aplica
kroB150	35	99,77	36,05	665,676	No aplica	No aplica
kroB150	50	99,73	34,53	1.327,528	No aplica	No aplica
kroB200	20	99,89	36,14	279,156	No aplica	No aplica

kroB200	35	99,84	35,06	2.234,738	No aplica	No aplica
kroB200	50	99,80	33,82	7.448,424	No aplica	No aplica
lin105	20	99,74	35,89	9,439	No aplica	No aplica
lin105	35	99,61	37,04	86,855	No aplica	No aplica
lin105	50	99,50	36,40	245,246	No aplica	No aplica
pr152	20	99,79	37,14	281,166	No aplica	No aplica
pr152	35	99,77	36,86	808,477	No aplica	No aplica
pr152	50	99,74	36,88	1.673,465	No aplica	No aplica
rat195	20	99,88	37,31	280,948	No aplica	No aplica
rat195	35	99,82	34,70	1.925,985	No aplica	No aplica
rat195	50	99,80	34,99	4.599,873	No aplica	No aplica
st70	20	99,44	39,84	39,852	91,9	4,072E-06
st70	35	99,30	39,56	63,650	90,6	5,743E-06
st70	50	99,16	36,37	128,195	91,3	7,027E-06
tsp225	20	99,88	34,98	1.658,773	No aplica	No aplica
tsp225	35	99,85	34,65	4.684,367	No aplica	No aplica
tsp225	50	99,82	33,26	12.088,726	No aplica	No aplica
u159	20	99,81	35,84	333,263	No aplica	No aplica
u159	35	99,76	36,14	864,992	No aplica	No aplica
u159	50	99,75	35,61	1.278,130	No aplica	No aplica
rd100	20	99,68	37,15	22,421	No aplica	No aplica
rd100	35	99,50	34,54	126,822	No aplica	No aplica
rd100	50	99,42	36,13	245,827	No aplica	No aplica
berlin52(E)	20	98,45	25,25	34,209	99,3	4,848E-07
eil51(E)	20	98,47	28,45	29,623	99,6	2,707E-07
att48(E)	35	97,45	31,74	62,967	99,4	4,930E-07
st70(E)	35	98,52	31,87	135,508	99,3	6,549E-07
brazil58(E)	50	97,48	31,84	172,636	99,4	4,825E-07
eil51(E)	50	97,26	32,67	74,473	99,1	7,942E-07
kroB100(E)	20	99,37	30,25	39,255	No aplica	No aplica
lin105(E)	20	99,33	31,95	64,409	No aplica	No aplica
kroA100(E)	35	98,99	35,88	225,505	No aplica	No aplica
rd100(E)	35	99,15	35,30	130,008	No aplica	No aplica

Tabla 2: Resultados numéricos

El porcentaje de mejora del algoritmo *VNS* por sobre el costo resultante luego de la aplicación del algoritmo de construcción *Greedy* (**%MVNS**) se encuentra, dependiendo de la instancia que esté siendo evaluada y sus características de evaluación dispuestas en el test-set, entre el 25,25% y el 39,84%

Se puede observar que para las instancias en las que la confiabilidad promedio (**%R-Conf**) fue evaluada, la misma supera ampliamente el umbral de 60% establecido. Para aquellas instancias en que la confiabilidad nodo-arista se estableció en 99%-95% respectivamente el guarismo evaluado se encuentra acotado por lo valores 86,0%-96,7% mientras que para aquellas en que los valores de confiabilidad nodo-arista se estableció en 99%-90% los valores de confiabilidad promedio se encuentran entre 99,1% y 99,6%.

Para todos los casos evaluados la varianza promedio obtenida (**VAR**) es pequeña, asociada esta a valores de confiabilidad que a pesar de ser en algunos casos muy cercanos a uno (100% de confiabilidad) mantienen el grado de precisión esperado, lo cual será analizado en la sección siguiente.

En lo que respecta a los tiempos promedios por iteración (**TP-I**) podemos decir que son aceptables, no llegando a ser dichos tiempos de ejecución prohibitivos en lo que respecta a tiempo computacional en ninguna de las instancias del test-set evaluadas.

8. Conclusiones

En este proyecto se estudia el diseño topológico de redes confiables atacando dos subproblemas debidamente identificados, el problema de optimización de la red y el problema de satisfacer la cota de confiabilidad establecida para la misma. Para ello modelamos el problema de diseño de la topología de red sobre la base del GSP-NC (*Generalized Steiner Problem with Node-Connectivity Constraints*).

La optimización de la red fue atacada heurísticamente, ya que se trata de un problema NP-Duro 10 y por ende la aplicabilidad de algoritmos exactos es prohibitiva en lo que respecta a tiempo computacional, aún para redes de pequeño y mediano tamaño. Por ello optamos por utilizar la metaheurística VNS (*Variable Neighbourhood Search*) basándonos en las razones debidamente expuestas en la Sección 3.1 *"Selección de la Metaheurística"*.

En lo que refiere al cálculo de confiabilidad de la red, dado que la evaluación exacta de la medida para redes de tamaño considerable es impracticable, el mismo fue abordado mediante el método de simulación RVR (*Recursive Variance Reduction*), decisión justificada en la sección 3.2 *"Selección del método de calculo de confiabilidad"*.

Dichos algoritmos, el modelo para la representación de grafos estocásticos y los restantes algoritmos que forman parte de la solución descripta a lo largo del presente documento, fueron implementados en C++ bajo plataforma Microsoft Windows XP Professional Edition. Como ya ha sido expuesto han sido generados casos de prueba para la validación y testeo de los diferentes algoritmos.

A continuación se presenta un resumen de los resultados experimentales obtenidos para los problemas planteados previamente.

Primeramente vale recordar que para las pruebas formales realizadas se utilizaron adaptaciones de instancias obtenidas de la librería TSP 10 debido a que hasta donde es de nuestro conocimiento no existen casos benchmark que permitan realizar medición de rendimiento alguna del problema en estudio.

Como se menciono en la sección anterior el porcentaje de mejora del algoritmo *VNS* por sobre el costo resultante luego de la aplicación del algoritmo de construcción *Greedy* se sitúa entre el 25,25% y el 39,84%, dependiendo de la instancia evaluada y sus respectivas características. Considerando que la implementación del algoritmo *Greedy* se basa en el algoritmo de construcción utilizado por el Dr. Ing. Franco Robledo en trabajos previos 10 y sabiendo en base a la documentación resultante de dichos trabajos y a charlas mantenidas con el citado tutor que el mencionado algoritmo es un buen algoritmo aleatorio de construcción de soluciones iniciales (entiéndase que construye grafos iniciales factibles de muy bajo costo) podemos concluir que la mejora obtenida por *VNS* es aceptable para todas las instancias. Es decir, el mejorar el costo de la red resultante de la aplicación del algoritmo de construcción en al menos un 25,25% parece ser un guarismo para nada despreciable.

La confiabilidad promedio de las instancias evaluadas supero en todos los casos el umbral establecido situándose los valores resultantes entre 86,0% y 99,6%, dependiendo del valor de confiabilidad elemental asignado a los nodos y aristas de la instancia evaluada y a los requerimientos de conexión que fuesen necesarios satisfacer. Para todos los casos evaluados la varianza promedio obtenida es pequeña, asociada esta a valores de confiabilidad que a pesar de ser en algunos casos muy cercanos a uno (100% de confiabilidad) mantienen el grado de precisión esperado. Primeramente es destacable el hecho de que los resultados obtenidos son en todos los casos los esperados para una red de alta confiabilidad, a su vez y a pesar de que las redes evaluadas son casi perfectas es destacable el hecho de lograr precisión en los resultados ya que en trabajos anteriores 10 ha quedado demostrado que otros algoritmos (como por ejemplo *Montecarlo Crudo*) retornan confiabilidad uno y varianza cero en evaluación de instancias de redes muy confiables. Es igualmente importante aclarar que los guarismos de confiabilidad fueron medidos con 6 cifras significativas, las cuales fueron redondeadas en la presente sección y en la sección 7.2 *Resultados Numéricos* a efectos de facilitar la lectura de dichos valores.

En lo que respecta a los tiempos promedios por iteración se puede concluir que son aceptables ya que los mismos no son prohibitivos en lo que respecta a tiempo computacional en ninguna de las instancias del test-set evaluadas. De todas formas, dicho lo anterior, es importante considerar que la evaluación de las instancias con cantidad de nodos mayor a 100 no incluyo la ejecución del calculo de confiabilidad mediante *RVR*.

Finalmente amerita recalcar que la documentación formal que recabe información referente a investigación o experimentación sobre el problema objetivo de esta tesis, hasta donde pudimos relevar es escasa y prácticamente nula, por lo menos a nivel académico. Por lo que hemos podido constatar en el Instituto de Computación de la Facultad de Ingeniería de la Universidad de la República no han sido desarrolladas otras tesis de grado que aborden el mismo problema lo cual haría del presente trabajo la primera tesis en enfocarse sobre la problemática mencionada.
Tendiendo en cuenta entonces que se trata de la primera experiencia en el trato del problema abordado, estando los resultados obtenidos dentro de parámetros aceptables y considerando la importancia de los problemas de supervivencia y confiabilidad se entiende que es posible y de seguro rentable y provechoso el continuar con el estudio de estos problemas.

9. Trabajos Futuros

Los trabajos futuros de interés están básicamente referidos a la implementación de los algoritmos utilizados y a la experimentación sobre los mismos. A continuación se enumeran los que a nuestro criterio serían trabajos de relevancia:

- Dado que el cambio sistemático de estructura de vecindad es una herramienta simple y muy potente la posibilidad de incorporar VNS a otra metaheurística y obtener de esta forma un hibrido 1010 parece ser una opción interesante a desarrollar para potenciar de esta forma la metaheurística solución al problema de optimización de la red. La búsqueda tabú o TS 1010 generalmente usa una estructura de vecindad con respecto a la que ejecuta movimientos de ascenso y descenso explotando diferentes tipos de memoria. En principio habrían dos formas de combinar y hacer híbridos de VNS y TS, usando algún tipo de memoria para orientar la búsqueda dentro de VNS o usar VNS dentro de TS. La metaheurística GRASP 101010101010 puede también ser utilizada junto con VNS y resultar en un hibrido. Evidentemente la posibilidad de hibridación no se limita únicamente a GRASP y TS.
- VNS se sustenta sobre la idea de ir cambiando en forma sistemática la vecindad al momento de realizar la búsqueda y por ende necesita de un conjunto finito de vecindades distintas predefinidas. En el presente proyecto dicho conjunto está compuesto por tres búsquedas locales, *KeyPathLocalSearch*, *KeyTreeLocalSearch* y *SwapKeyPathLocalSearch* conjunto que dado el calendario que rige un proyecto de grado no pudo ser ampliado. A pesar de ello el enfoque orientado a objetos utilizado para el análisis y diseño facilita desarrollar e incluir en el sistema nuevas búsquedas locales para ser utilizadas por VNS, extendiendo de esta forma el conjunto finito de vecindades del cual se vale la metaheurística.
- Existen diversas propuestas para realizar extensiones a VNS y dotarlo de algunas características adicionales. Entre ellas y para la resolución de problemas grandes se encuentran la búsqueda de entorno variable con descomposición (VNDS), la búsqueda de entorno variable sesgada (SVNS) y la búsqueda de entorno variable paralela (PVNS). Ahondar en estas variantes y estudiar la aplicabilidad al problema de forma de potenciar el algoritmo de optimización implementado parece ser un trabajo a futuro relevante.
- Dado que se opto por el uso de una metodología orientada a objetos (ver diagrama UML) existe la posibilidad de sustituir tanto el algoritmo de construcción como el utilizado para el cálculo de la confiabilidad (*Greedy* y *RVR* respectivamente). Experimentar sobre variantes de dichos algoritmos podría conducir a mejoras en la solución global al problema objetivo de la presente tesis.

10.Bibliografía

[1] Santiago Arraga, Miguel Aroztegui. “Algoritmos Genéticos Paralelos para el Problema General de Steiner en Grafos”. Proyecto Final – Ingeniería en Computación. Julio 2002.

[2] Héctor Cancela, Franco Robledo, Gerardo Rubino. “Network design with node connectivity constraints”. Universidad de la República. IRISA. Applications, Technologies, Architectures and Protocols for Computer Communication. Proceedings of 2003 IFIP/ACM Latin America conference Towards a Latin American agenda for network research. La Paz, Bolivia, páginas 13-20, ISBN 1-58113-789-3.

[3] Griselda I. Friss de Kerebi, Mario Maneyero, Franco Robledo, Ariel Sabiguero. “Modelo de Confiabilidad en Redes”. Taller V. 1996.

[4] Antonio Mauttone. “Método RVR en la simulación de medidas de confiabilidad en redes”. Proyecto de Taller V. Universidad de la República. Febrero 2000.

[5] Eduardo Morales. “Búsqueda Optimización y Aprendizaje”. Capítulos 4, 5, 6 y 9. Tec. de Monterrey (ITESM). 2004. http://ccc.inaoep.mx/~emorales/Cursos/Busqueda04/principal.html. Ultima fecha de ingreso 10/06/2006.

[6] Mauricio G.C. Resende, José Luis González Velarde. “GRASP: Greedy Randomized Adaptative Search Procedures”. Algorithms and Optimization Research Deparment AT\&T Labs-Research. Tecnológico de Monterrey. 2003. Inteligencia Artificial, Revista Iberoamericana de Inteligencia Artificial. No. 19 (2003), pp. 61-76 ISSN: 1137-3601.

[7] Franco Robledo Amoza. “Diseño Topológico de Redes”. Caso de Estudio: “The Generalized Steiner Problem” and “The Steiner 2-Edge-Connected Subgraph Problem”. Tesis de Maestría. Universidad de la República. Febrero 2000.

[8] T. Cormen, C. Leiserson, R. Rivest. “*Introduction to Algorithms”*. MIT Press, 1990. Chapter 16, Greedy Algorithms. ISBN 0262032937.

[9] Garey, M. and D. Johnson, “Computers and Intractability; A Guide to the Theory of NP-Completeness”, 1979. ISBN 0716710455.

[10] Hector Cancela. “Adapting RVR simulation techniques for general network reliability models”. Investigación Operativa – InCo – Pedeciba Informática. Facultad de Ingeniería. Universidad de la República. Montevideo, Uruguay. Reporte Técnico INCO 99.05. Mayo 1999.

[11] Franco Robledo. “GRASP heuristics for Wide Area Network design”. Tesis de Doctorado. Universidad de Rennes – Francia. Febrero 2005.

[12] H. Cancela and M. El Khadiri. “Series Parallel reductions in RVR reliavility evaluation”. Technical Report INCO 96-01, PEDECIBA .

[13] Craig Larman. “UML y Patrones. Introducción al análisis y diseño orientado a objetos”. Primera Edición. Prentice Hall, Inc. ISBN 970-17-0261-1.

[14] Ernesto de Queiros Vieira Martins, Marta Margarida Braz Pascoal. “A New Implementation of Yen’s Ranking Loopless Paths Algorithm”. Universidade de Coimbra – Portugal. Octubre 2000.

[15] Marcelo Mejía, Pablo E. Olmos Aguirre. “Network Topology Optimization using Tabu Search”. Instituto Tecnológico Autónomo de México, México, D.F., México.

[16] Pierre Hansen, Nenad Mladenovic, José Andrés Moreno Pérez. “Variable Neighbourhood Search”. Inteligencia Artificial, Revista Iberoamericana de Inteligencia Artificial. No. 19 (2003), pp. 77-92. ISSN: 1137-3601.

[17] Leonardo M. Gomes, Viviane B. Diniz, Carlos A. Martinhon. “An Hybrid GRASP+VNS Metaheuristic for the Prize-Collecting Traveling Salesman Problem”. Departamento de Ciencias de la Computación / Instituto de Computación Universidad Federal Fluminense (UFF) – Niterói, RJ, Brasil.

[18] Pierre Hansen – GERAD and HEC Montreal, Nenad Mladenovic – GERAD and Mathematical Institute, SANU, Belgrade. “A Tutorial on Variable Neighborhood Search”. Julio 2003.

[19] Helena R. Lourenco – Universitat Pompeu Fabra, Barcelona, España. Olivier C. Martin – Université Paris-Sud, Orsay, Francia. Thomas Stutzle – Darmstadt University of Technology, Darmstandt, Alemania. “Iterated Local Search”. Febrero 2001.

[20] E. Dijkstra. “A note on two problems in connection with graphs”. Numerical Mathematics, 1:395-412, 1959.

[21] E. L. Lawler. “A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem”. Management Science, 18:401-405, 1972.

[22] E. Q. V. Martins, M. M. B. Pascoal, and J. L. E. Santos. “Deviation algorithms for ranking shortest paths”. The International Journal of Foundations of Computer Science, 10:(3):247-263, 1999.

[23] L. R. Ford, Jr. And D. R. Fulkerson, “Flows in Networks”, Princeton. N. J.: Princeton UniversityPress 1962.

[24]TSPLIB, library of sample instances for the travelling salesman problem and related problems. http://elib.zib.de/pub/Packages/mp-testdata/tsp/tsplib/tsplib.html. Ultima fecha de ingreso 10/06/2006.

[25] Wikipedia la enciclopedia libre. www.wikipedia.org. Ultima fecha de ingreso 10/06/2006.

11.Apéndices

11.1.Apéndice 1 – Informe de Contexto del problema

El informe mencionado se adjunta como documento anexo al final del presente informe.

11.2.Apéndice 2 – Búsqueda Local, conservación de factibilidad

A continuación se demuestra la conservación de las factibilidades 10 para la Búsqueda Local 1, Búsqueda Local 2 y Búsqueda Local 3, presentadas en la sección 3. Sumado a esto se describe el procedimiento auxiliar General_RecConnect para la Búsqueda Local 2 (KeyTreeLocalSearch) 10.

Proposición Dada una solución factible g_{sol} que satisfaga la matriz de requerimientos de conexión R, el algoritmo de búsqueda local KeyPathLocalSearch preserva la factibilidad, retornando una solución factible vecina que satisfaga R.
Prueba:
Supongamos que KeyPathLocalSearch no preserva la factibilidad. Necesariamente, en alguna iteración, deberíamos tener:

- La solución actual g_{sol} es factible
- El camino $\hat{p}$ computado en las líneas 7-8 satisface: $COST(\hat{p}) < COST(p)$, donde $p \in K(g_{sol})$ es el key-path actual.
- La red $\hat{g} = (g_{sol} \setminus p) \cup \hat{p}$ no es factible, es decir, $\exists i, j \in K(g_{sol})$ tal que en $\hat{g}$ no hay r_{ij} caminos nodo-disjuntos conectándolos.

Por construcción esto implicaría que $NODOS\_INTERNOS(\hat{p}) \cap NODOS(g_{sol} \setminus p) \neq 0$, lo cual es una contradicción dado que $NODOS(\hat{p}) \subseteq (NODOS(p) \cup (S_D \setminus NODOS(g_{sol}))$. Por ende la red $\hat{g}$ es factible y satisface la matriz de requerimientos de conexión R, completando entonces la prueba.

Descripción General_RecConnect
Este algoritmo es un procedimiento auxiliar usado por el algoritmo KeyTreeLocalSearch. Dada la solución actual g_{sol} y un key-nodo $v \in g_{sol}$, General_RecConnect intenta construir un mejor key-tree T cubriendo los extremos de T_v, donde T_v es el key-tree asociado a *v*. Para preservar la factibilidad, el key-tree sustituto T es construido usando solo nodos de T_v y nodos no terminales no pertenecientes a g_{sol}. Sumado a eso las aristas entre los extremos de T_v no son consideradas.

Procedure General_RecConnect ($G_B, C, g_{sol}, v, \bar{S}$ **)**

1. *cost* ← Cost_Key_Tree(v, g_{sol} **);**
2. Y ← Nodes_Key_Tree(v, g_{sol} **);**
3. Z ← Ends_Key_Tree(v, g_{sol} **);**
4. $\hat{S} \leftarrow (Y \setminus Z) \cup \bar{S}$;
5. $U = \left\{ (i,j) \in G_B \mid i \in Z, j \in \hat{S} \right\}$;
6. $\hat{\mu} \leftarrow$ **el subgrafo inducido por** $\hat{S}$ **en** G_B ; $\hat{\mu} \leftarrow \hat{\mu} \cup U$;
7. $T \leftarrow \{ v \}$;
8. **while** $\exists u \in Z$ **tal que** $u \notin T$ **do**
9. $u \leftarrow$ Select_Random(*X*) **donde** $X = \{ u \in Z \mid u \notin T \}$;
10. $\mu \leftarrow \hat{\mu} \setminus (Z \setminus \{ u \})$;
11. $p \leftarrow$ **el camino mas corto de** *u* **a** T **considerando** μ **;**
12. $T \leftarrow T \cup p;$
13. **end while;**
14. **Iterativamente remover todo** $s \in \hat{S}$ **de** T **con grado 1;**
15. **If (** $COSTO(T) < COSTO$ **) then**
16. $g_{sol} \leftarrow (g_{sol} \setminus (Y \setminus Z)) \cup T$ **;**
17. *improve* ← TRUE;
18. **else** *improve* ← FALSE;
19. **return** g_{sol} , *improve***;**

end General_RecConnect;

Figura 38: Pseudo-código de GeneralRecConnect

Como se puede apreciar en la figura el algoritmo toma como entradas al grafo G_B de conexiones factibles, la matriz de costos C y la solución factible actual g_{sol}, el key-nodo actual *v*, y el conjunto de nodos no terminales $\bar{S}$ no pertenecientes a g_{sol}. Supongamos T_v sea el key-tree asociado a *v.* La línea 1 calcula el costo de T_v. La línea 2 computa el conjunto Y de nodos pertenecientes a T_v. La línea 3 computa el conjunto $Z \subset Y$ de extremos de T_v. La línea 4 computa el conjunto $\hat{S} = (Y \setminus Z) \cup \bar{S}$, el cual incluye todos los nodos no terminales que no pertenecen a g_{sol} y todos los nodos de T_v con excepción de los extremos. En la línea 5 calculamos el conjunto U conteniendo todas las conexiones de G_B que tienen un extremo en Z y otro en $\hat{S}$. Claramente en U no hay conexiones entre nodos de Z. Supongamos $\hat{\mu}$ la subred inducida por $\hat{S}$ en g_{sol}. La línea 6 añade a $\hat{\mu}$ el conjunto U. Notemos que cualquier árbol de expansión computado en $\hat{\mu}$ es un potencial substituto para T_v en g_{sol} dado que al reemplazar T_v por este se mantiene la factibilidad. En la línea 7 inicializamos el árbol substituto T con el nodo terminal *v*. El loop 8-13 construye iterativamente un nuevo key-tree añadiendo de a uno nodos de Z a T. La línea 9 selecciona aleatoriamente y uniformemente

un nodo $u \in Z$ que aún no haya sido añadido a T. En la línea 10 consideramos la red auxiliar $\mu = \hat{\mu} \setminus (Z \setminus u)$ para calcular un camino de u a T. Los nodos de $(Z \setminus u)$ no son considerados al conectar u a T, dado que estos deben ser extremos en T. La línea 11 calcula el camino mas corto de u a T en μ. Supongamos p sea dicho camino, en la línea 11 añadimos p a T. Una vez que todos los nodos de Z han sido añadidos a T el loop 8-13 termina y los nodos de Steiner colgantes son quitados de T en la línea 14. Notar que estos no son necesarios para garantizar la factibilidad. Además es fácil ver que un key-tree substituto puede ser siempre construido desde $T_v \subseteq \hat{\mu}$. En la línea 15 comparamos los costos de T y T_v. Si T es un mejor key-tree, la solución actual g_{sol} es actualizada en la línea 16, reemplazando T_v por T. El indicador *improve* es configurado a TRUE en la línea 17 (usado en KeyTreeLocalSearch). De lo contrario, si T tiene mayor costo que T_v, *improve* es puesta en FALSE en la línea 18. El indicador *improve* y la solución g_{sol} son retornados en la línea 19.

Proposición Dada una solución factible g_{sol} que satisfaga la matriz de requerimientos de conexión R, el conjunto de nodos no terminales $\bar{S}$ no pertenecientes a g_{sol} y el key-nodo actual *v*. El algoritmo General_RecConnect construye una solución vecina reemplazando el key-tree asociado a *v* por otro árbol que preserve la factibilidad.

Prueba:

Supongamos T_v sea el key-tree asociado con *v*. Las líneas 1-5 computan el costo de T_v, el conjunto Y de nodos en T_v, el conjunto $Z \subset T_v$ de extremos, el conjunto $\hat{S} = (Y \setminus Z) \cup \bar{S}$, y el conjunto $U = \left\{ (i,j) \in G_B \mid i \in Z, j \in \hat{S} \right\}$. La línea 6 calcula la red $\hat{\mu} = U \cup G_B(\hat{S})$. La línea 7 inicializa T (el key-tree substituto) con el nodo *v*. Es fácil de ver que por construcción, una vez finalizado el loop 8-13 y la línea 14, la red T tiene topología de árbol y además:

1. $Z \subset T$
2. Los extremos de T son exactamente los nodos de Z
3. $NODOS(T) \cap NODOS(g_{sol}) = Z \cup J \; con \; J \subseteq (Y \setminus Z)$
4. existe un nodo $\hat{s} \in \hat{S}$ raíz del árbol T (no necesariamente $\hat{s} = v$).

Si la condición en la línea 15 es verdadera, el algoritmo computa la red $\hat{g} = (g_{sol} \setminus T_v) \cup T$ en la línea 16. Los puntos 1-4 inducen la factibilidad de la red $\hat{g}$ dado que al reemplazar el key-tree T_v con T la perdida de requerimientos de nodo-conectividad en $(g_{sol} \setminus T_v)$ son reestablecidos al añadir T. Por ende, la red retornada en la línea 19 es factible y satisface la matriz R.

Proposición Si SwapKeyTreeLocalSearch recibe como entrada una solución factible g_{sol} que satisfaga la matriz de requerimientos de conexión R, el algoritmo de búsqueda local SwapKeyTreeLocalSearch preserva la factibilidad.

Prueba:

Supongamos que SwapKeyTreeLocalSearch no preserva la factibilidad. Necesariamente, en alguna iteración, deberíamos tener:

i. La solución actual g_{sol} es factible

ii. El camino $\hat{p}$ computado en las líneas 6-8 satisface: $COST(\bar{p} \setminus (g_{sol} \setminus p)) < COST(p)$, donde $p \in K(g_{sol})$ es el key-path actual, por lo tanto la linea 10 es computada.

iii. La red $\hat{g} = (g_{sol} \setminus p) \cup \hat{p}$ no es factible, es decir, $\exists i, j \in S_D^{(i)}$ tal que en $\hat{g}$ no hay r_{ij} caminos nodo-disjuntos conectándolos.

Sea $\hat{p}_{ij} \in P_{ij}$ tal que $p \subseteq \hat{p}_{ij}$ Definimos el camino $p_{aux} = (\hat{p}_{ij} \setminus p) \cup \bar{p}$. Los nodos de $X_p(P)$ son excluidos de $\hat{H}$ por lo tanto en $\hat{H}$ no hay nodos de $(P_{ij} \setminus \hat{p}_{ij})$, entonces tenemos que $NODOS\_INTERNOS(p_{aux}) \bigcap NODOS\_INTERNOS(p_{ij}) \neq \emptyset, \forall p_{ij} \in (P_{ij} \setminus \hat{p}_{ij})$, lo cual contradice (iii). Por lo tanto la red $\hat{G}$ computada en la linea 10 is factible satisfaciendo la matriz de conecciones R. Para completar la prueba notar que:

- Todos los caminos $\hat{p} \in P$ que contienen p son actualizados en la linea 10 remplazando p por $\bar{p}$, De acuerdo con esto hay r_{ij} caminos nodo-disjuntos en P_{ij} conectando i con j, $\forall i, j \in S_D^{(i)}$
- Las lineas 13-15 recomputan la descomposición en key-paths de g_{sol} (La cuál fue actualizada en la linea 10 por $\hat{G}$) si se introduce un nuevo key-nodo

En este sentido, la factibilidad de la topología actual es garantizada en cada iteración de la búsqueda local

Proposición Si KeyTreeLocalSearch recibe como entrada una solución factible g_{sol} que satisface la matriz de requerimientos de conexión R, la factibilidad es preservada durante todas las iteraciones del algoritmo.

Prueba:

Por contradicción, para ciertas iteraciones tenemos que g_{sol} es factible satisfaciendo por completo a la matriz R, X es su conjunto de key-nodos y existe un key-nodo $u \in X$ tal que General_RecConnect retorna una solución no factible. Esto contradice la proposición anterior. Por ende, el algoritmo preserva la factibilidad en todo momento.

11.3.Apéndice 3 –Algoritmo de Construcción

A continuación se demuestra la conservación de factibilidad 10 para el Algoritmo de Construcción Greedy y se explican los métodos $General\_Update\_Matrix$ y KSP.

Procedure General_Update_Matrix(G_{sol}, P, M, p, i, j **);**

1. **for each** $k \in S_D^{(I)}, k \neq i, j$ **such that** $k \in p$ **do**
2. **if** $m_{ik} > 0$ **then**
3. **if** $(NODES(P_{ik}) \cap NODES(p_{(i,k)}) = \{i, k\})$ **then**
4. $P_{ik} \leftarrow P_{ik} \cup \{p_{(i,k)}\}$;
5. $m_{ik} \leftarrow m_{ik} - 1$; $m_{ki} \leftarrow m_{ki} - 1$
6. **end_if**
7. **end_if**
8. **if** $m_{kj} > 0$ **then**
9. **if** $(NODES(P_{kj}) \cap NODES(p_{(k,j)}) = \{k, j\})$ **then**
10. $P_{kj} \leftarrow P_{kj} \cup \{p_{(k,j)}\}$;
11. $m_{kj} \leftarrow m_{kj} - 1$; $m_{jk} \leftarrow m_{jk} - 1$
12. **end_if**
13. **end_if**
14. **end_foreach**
15. **return** P, M ;

end General_Update_Matrix;

Figura 39: Pseudo-código de GeneralUpdateMatrix 10

El algoritmo (mostrado en la Figura 39) recibe como entrada la solución generada hasta el momento (en la construcción) G_{sol}, la matriz de caminos P, la matriz de requerimientos de conexión M, los terminales i, j y el camino p encontrado entre ellos. El ciclo de 1-14 analiza cada nodo Terminal $k \in S_D^{(I)}, k \neq i, j$ tal que $k \in p$ chequeando si en existe un sub-camino que conecta k con i (resp. j) el cual es nodo disjunto con los caminos ya existentes en P_{ik} (resp. P_{kj}). Si este es el caso se actualiza el conjunto P_{ik} (resp P_{kj}) agregando $p_{(i,k)}$ (resp. $p_{(k,j)}$), y m_{ik} y m_{ki} (resp. m_{kj} y m_{jk}) son decrementados en uno. Como optimización a este algoritmo también se chequea y realiza la inclusión de nuevos caminos entre los terminales intermedios del camino p.

Proposición Una vez terminado el algoritmo $General\_Update\_Matrix$ los siguientes puntos son satisfechos:

i. $P_{ij} = \varnothing \Leftrightarrow m_{ij} = r_{ij}$.
ii. Si $m_{ij} = k$ (con $k \in 0..r_{ij}$) entonces existe al menos $r_{ij} - k$ caminos nodos disjuntos desde i a j en G_{sol}.
iii. La relación $|P_{ij}| = r_{ij} - m_{ij}$ es satisfecha en cada iteración del algoritmo de construcción.

Prueba: Primeramente, asumimos que cuando se invoca el método $General\_Update\_Matrix$ P y M satisfacen los puntos i-iii. Tomemos $i, j \in S_D^{(I)}$ como el par de terminales y p el camino que los conecta computado por el algoritmo de construcción. El ciclo analiza $\forall k \in S_D^{(I)}, k \in p, k \neq i, j$ los siguientes casos:

Caso 1: Si $m_{ik} > 0$ sabemos que existe $r_{ij} - m_{ij}$ caminos nodos disjuntos entre i , j en G_{sol} . Además, si $m_{ij} = r_{ij}$ tenemos $P_{ij} = \varnothing$. Si la condición $NODES(P_{ik}) \cap NODES(p_{(i,k)}) = \{i,k\}$ es verdadera, el sub-camino $p_{(i,k)}$ es agregado a P_{ik} ya que es un camino nodo disjunto con respecto a los caminos en P_{ik} . Los valores m_{ik} y m_{ki} son decrementados en uno preservando los puntos i-iii.

Caso 2: Idem al caso anterior.

Proposición: Si $A_{ij} < MAX\_ATTEMPT, \forall i, j \in S_D^{(I)}$ entonces el grafo solución retornado por el algoritmo de construcción es factible (satisface la matriz R de requerimientos de conexión).

Prueba: En la prueba asumimos que existe una sub-red $G_{sol} \subseteq G_B$ que satisface la matriz R , como se vio anteriormente en la línea 1 el algoritmo inicializa:

- G_{sol} con un conjunto de nodos terminales $S_D^{(I)}$ y un conjunto vacío de aristas.
- La matriz auxiliar M con $m_{ij} = r_{ij}, \forall i, j \in S_D^{(I)}$.
- La matriz P vacía.

Suponemos que para cierta iteración la condición en la línea 2 es verdadera. En la línea 3 elegimos aleatoriamente un par $i, j \in S_D^{(I)}$ tal que $m_{ij} > 0$ .En la línea 4 computamos la red auxiliar $\overline{G} = (G_B \setminus P_{ij})$ .Notemos que, si existe un camino desde i a j en $\overline{G}$, este camino es nodo disjunto con respecto a los caminos en P_{ij} . Además, la línea 5 computa la matriz de costos auxiliar $\overline{C}$ asociada con $\overline{G}$, asignando costo cero a las aristas que están en G_{sol} . Sabiendo que el método $General\_Update\_Matrix$ preserva la condición $|P_{ij}| = r_{ij} - m_{ij}$, líneas 5-12 buscan un nuevo camino desde i a j en G_{sol} considerando $\overline{C}$. Analicemos ahora los siguientes casos:

Caso 1: $\neg \exists p \subset \overline{G}$ que conecte i con j . En este caso, la línea 7 re-inicializa P_{ij} y m_{ij} ya que P_{ij} contiene un camino que es intersección de dos o mas caminos nodos disjuntos entre i , j . La construcción se retoma en la línea 2.

Caso 2: $\exists p \subset \overline{G}$ .En este caso en la línea 9 seleccionamos un camino p de la lista de caminos L_p (línea 6). Como $p \not\subset G_{sol}$ la solución actual G_{sol} es actualizada en la línea 9. El indicador m_{ij} es decrementado en uno línea 10.

Basado en el proceso de construcción descrito arriba, es fácil ver que una vez finalizado el ciclo 2-13, si $m_{ij} = 0, \forall i, j \in S_D^{(I)}$ entonces la solución obtenida G_{sol} satisface la matriz R .

Finalmente explicaremos el algoritmo de Yen 10 utilizado para implementar el problema de los K caminos más cortos, sin ciclos entre un par de nodos s y t. El algoritmo solución pertenece a la clase de Algoritmos de Desviación 10 en el cual se construye un seudo árbol de caminos sin ciclos.

Procedure KSP(G,s,t,k **);**

18. $ksp \leftarrow \varnothing$; $X \leftarrow \varnothing$; $kaux = 0$;
19. $p \leftarrow Dijkstra(G,s,t)$;
20. **if** $(p \neq \varnothing)$ **then**
21. $X \leftarrow X \cup \{ p \}$; $d(p) \leftarrow s$;
22. **while** $((X \neq \varnothing) and (kaux < k))$ **do**
23. $kaux \leftarrow kaux + 1$;
24. $p_k \leftarrow GetMinCostPath(X); v_i^k \leftarrow d(p_k)$;
25. $X \leftarrow X - \{ p_k \}$; $ksp \leftarrow ksp \cup \{ p_k \}$;
26. **if** $(kaux < k)$ **then**
27. **while** $(v_i^k \neq t)$ **do**
28. $\overline{G} \leftarrow G - Nodes(subp_k(s, v_{i-1}^k))$;
29. $\overline{G} \leftarrow \overline{G} - Arc(v_i^k, v_{i+1}^k)$;
30. $\overline{G} \leftarrow \overline{G} - Arcs(starting\_in\_v_i^k\_removed\_when\_p_k\_was\_computed)$;
31. $\hat{p} \leftarrow Dijkstra(\overline{G}, v_i^k, t)$;
32. **if** $(\hat{p} \neq \varnothing)$ **then**
33. $\hat{p} \leftarrow subp_k(s, v_{i-1}^k) \Diamond \hat{p}; X \leftarrow X \cup \hat{p}$;
34. **end_if**
35. $v_i^k \leftarrow v_{i+1}^k$;
36. **end_while**
37. **end_if**
38. **end_while**
39. **end_if**
40. **return** ksp ;

end KSP;

Figura 40: Pseudo-código de K Shortest Path 10

El algoritmo de la figura (Figura 40) recibe como entrada el grafo G un par de nodos s,t y un entero $k \geq 1$ que representa la cantidad de caminos más cortos, sin ciclos entre s,t . Inicialmente se inicializa la colección de caminos solución $ksp \leftarrow \varnothing$, $X \leftarrow \varnothing$ colección que representa los caminos candidatos computados durante el algoritmo. En el paso 2 se calcula el camino más corto entre s,t utilizando el algoritmo de Dijkstra 10. En caso de existir el camino p se agrega a X y $d(p) \leftarrow s$, siendo $d(p)$ su nodo de desviación (línea 4). En el ciclo 5-21 se computan los k caminos más cortos. En la iteración k primero se obtiene el camino más corto p_k ya calculado en X (línea 7) y su nodo de desviación $v_i^k \leftarrow d(p_k)$, p_k se agrega en ksp y se elimina de X (línea 8). Si aún no se encontraron los k caminos más cortos en el ciclo 10-19 nuevos caminos desviaciones de p_k son computados, para evitar recalcular solo los nodos desde v_i^k hasta t son analizados. Para cada uno de estos nodos, tomemos v_i^k , el camino más corto de la forma $\hat{p} \leftarrow subp_k(s, v_{i-1}^k) \Diamond \hat{p}$ (línea 16) donde $\hat{p}$ es el camino mas corto en G desde v_i^k hasta t , será computado (línea 14) utilizando nuevamente el algoritmo de Dijkstra 10, si el camino existe se agrega a X (línea 16). Previo al cálculo de $\hat{p}$ de forma evitar ciclos y no obtener un camino ya computado, se eliminan de G los nodos de p_k anteriores a v_i^k ,

$\overline{G} \leftarrow G - Nodes(subp_k(s, v_{i-1}^k))$ (línea 11) y también los arcos $\overline{G} \leftarrow \overline{G} - Arc(v_i^k, v_{i+1}^k)$ y $\overline{G} \leftarrow \overline{G} - Arcs(starting\_in\_v_i^k\_removed\_when\_p_k\_was\_computed)$ (líneas 12-13). De esta forma se obtienen los k caminos más cortos sin ciclos entre s,t. Para cada nodo analizado se debe resolver el algoritmo de Dijkstra de complejidad $O(m + n\log n)$ 10 suponiendo en el peor caso para los $p_1 \ . \ . \ . \ p_k$ se deben analizar n nodos la complejidad del KSP es $O(kn(m + n\log n))$ 10.

11.4.Apéndice 4 – Algoritmo de construcción de grafos de prueba

Para poder probar los algoritmos de construcción, optimización, confiabilidad, etc. construimos un algoritmo para crear grafos de diferentes tamaños y densidad sin tener que construirlos manualmente, este algoritmo toma como parámetros la cantidad de nodos, el grado mínimo que tendrán los nodos, y la cantidad de terminales, como resultado del algoritmo obtendremos un grafo. A continuación se muestra un seudo-código del algoritmo mencionado.

```
Procedure GenerateGraph( nodeCount, nodeDegree, Ter min alCount );
1.  g ← newGraph(nodeCount);
2.  for(i < nodesCount, i++)
3.  {
4.      g ← SetX, Y //Posición en la pantalla
5.      g ← SetNode Pr obability(i, 0.99)
6.  }
7.  ter min als = 0
8.  while(ter min als < ter min alsCount)
9.  {
10.     node ← Rand()
11.     if (!node.IsTer min al())
12.     {
13.             g ← MarkTer min al(node)
14.             g ← SetNode Pr obability(node, 1)
15.             ter min als++
16.     }
17. }
18. for(node1 < nodeCount, node1++)
19. {
20.     j = g → GetnodeDegree(node1)
21.     while( j < nodeDegree)
22.     {
23.             node2 ← Rand()
24.             if (node1 <> node2 && !g → ExistEdge(node1, node2))
25.             {
26.                     g ← AddEdge(node1, node2, Rand()%10 + 1, 0.9)
27.                     j++
28.             }
29.     }
30. }
31. ter min als = g → GetTer min als()
32. for(i < ter min als.Count, i++)
33. {
34.     node1 = ter min als.GetItem(i)
35.     for( j < ter min als.Count, j++)
36.     {
37.             node2 = ter min als.GetItem( j)
38.             g ← SetConn Re q(node1, node2, 2)
39.     }
40. }
41. return g ;
end GenerateGraph;
```

Figura 41: Algoritmo de construcción de pruebas

El algoritmo se divide en cuatro partes, en la primera construye los nodos, luego marca los terminales, después construye las aristas, y por último agrega los requerimientos de conexión.
A continuación se explica línea por línea el funcionamiento de este algoritmo.
En la línea 11.4 se construye un grafo vacío con la cantidad de nodos recibida por parámetro, desde la línea 11.4 hasta la 11.4 se setean aleatoriamente las posiciones del nodo en pantalla con respecto a los ejes X, Y, y también se setea la probabilidad del nodo con el valor 0.99.
Desde la línea 11.4 hasta la 11.4 se marcan los nodos terminales, los mismos se seleccionan aleatoriamente y se termina cuando se han marcado tantos nodos como el valor que se halla pasado como parámetro, hay que tener en cuenta que el número de terminales debe ser menor que el número de nodos.
Desde la línea 11.4 hasta la 11.4 se agregan las aristas. Por cada nodo se obtiene el grado del mismo, y se agregan aristas hasta completar el requerimiento de grado de los nodos ("*nodeDegree*") pasado por parámetro. El procedimiento para agregar aristas es el siguiente, se obtiene un nodo aleatoriamente y se verifica que no sea el mismo en cuestión, y que no exista ya una arista entre estos dos nodos, si se cumplen estos requerimientos se agrega la arista al grafo y se incrementa el grado del nodo en cuestión, al agregar la arista el costo es un numero aleatorio entre 1 y 10, y la probabilidad es 0.9.
Desde la línea 11.4 hasta la 11.4 se agregan los requerimientos de conexión, el procedimiento es el siguiente: por cada nodo terminal recorro todos los otros terminales y agrego el requerimiento de conexión con el valor 2, lo que significa que entre cada par de nodos terminales debe haber 2 caminos nodo disjuntos.
En la línea 11.4 se retorna el grafo generado.

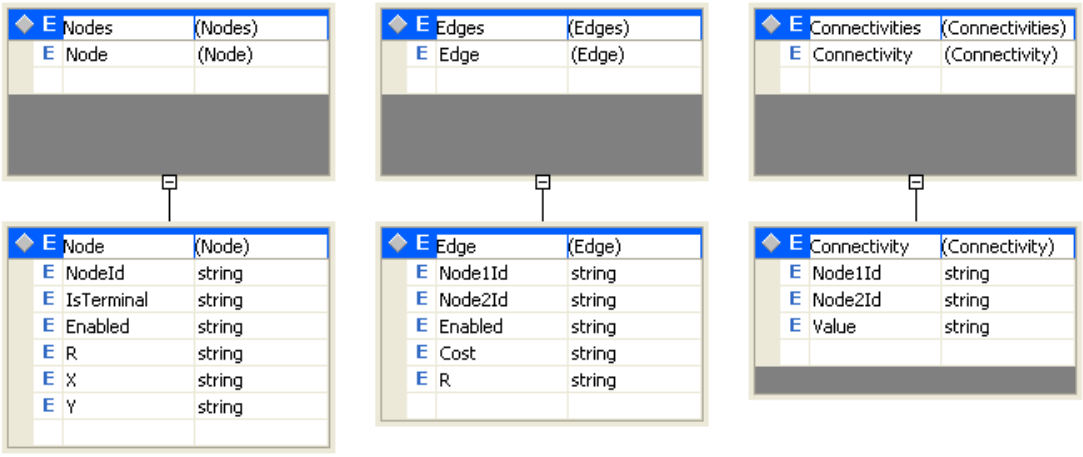
11.5.Apéndice 5 – Estructura de la información de un grafo, y visor del grafo

Para poder guardar el resultado de los algoritmos escribimos un archivo con formato xml, Elegimos xml por ser un estándar para el formato de la información.
El formato es el siguiente:
Un tag raiz llamado “Graph” que contiene la información de todo el gafo, dentro del tag “Graph”, se encuentran 3 tags “Nodes”, “Edges”, y ”Connectivities”.

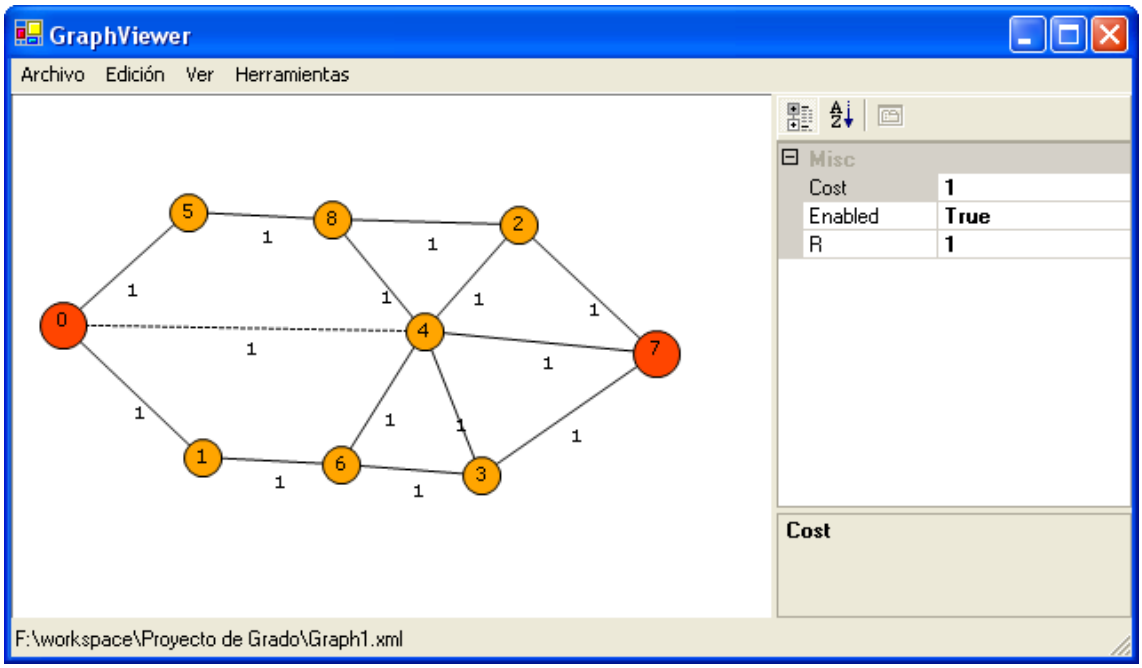
- El Tag ”Nodes” tiene un secuencia de tags “Node” que representan cada uno de los nodos.
- El tag “Edges” contiene una secuencia de tags “Edge” que representa cada una de las aristas, cada arista contiene la información sobre que nodos conecta.
- El tag “Conectivities” contiene una secuencia de tags “Connectivity” que describen las conectividades entre pares de nodos terminales

```
<?xml version="1.0" standalone="yes"?>
<Graph xmlns="http://tempuri.org/Graph.xsd">
  <Nodes>
    <Node>
      <NodeId>0</NodeId>
      <IsTerminal>True</IsTerminal>
      <Enabled>True</Enabled>
      <R>1</R>
      <X>16</X>
      <Y>117</Y>
    </Node>
    <Node>
      <NodeId>1</NodeId>
      <IsTerminal>False</IsTerminal>
      <Enabled>True</Enabled>
      <R>1</R>
      <X>97</X>
      <Y>194</Y>
    </Node>
  </Nodes>
  <Edges>
    <Edge>
      <Node1Id>0</Node1Id>
      <Node2Id>5</Node2Id>
      <Enabled>True</Enabled>
      <Cost>1</Cost>
      <R>1</R>
    </Edge>
    <Edge>
      <Node1Id>5</Node1Id>
      <Node2Id>8</Node2Id>
      <Enabled>True</Enabled>
      <Cost>1</Cost>
      <R>1</R>
    </Edge>
  </Edges>
  <Connectivities>
    <Connectivity>
      <Node1Id>0</Node1Id>
      <Node2Id>7</Node2Id>
      <Value>2</Value>
    </Connectivity>
  </Connectivities>
</Graph>
```

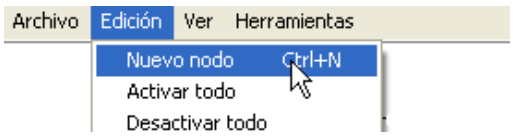
La siguiente imagen muestra gráficamente la estructura del xml utilizado para guardar la información.



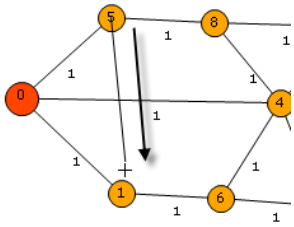
Para poder visualizar el resultado de una ejecución, o sea el archivo xml generado por el algoritmo NetworkDesign, utilizamos la siguiente aplicación.



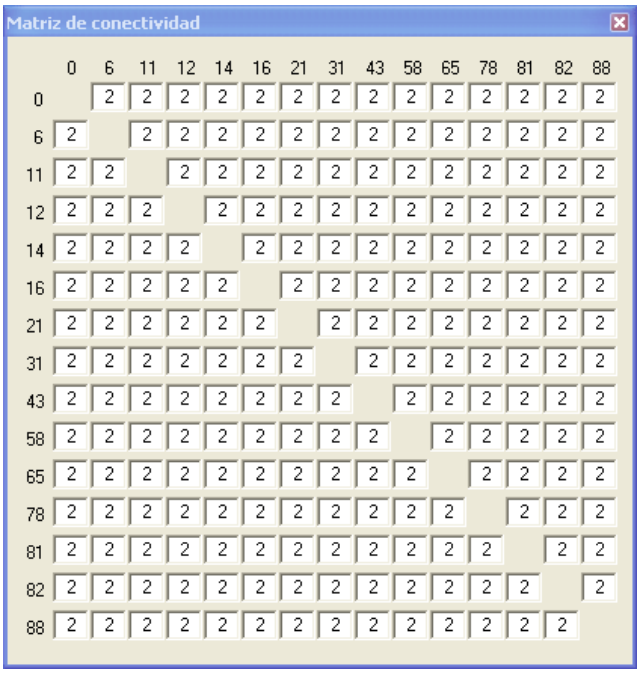
Esta aplicación también nos permite crear un nuevo grafo, y modificarlo, lo que agiliza la creación de los casos de prueba, la estandarización de los mismos, y facilita la visualización.
Para modificar las propiedades de un nodo ó una arista solo se necesita hacer doble clic sobre la misma y editarlo en la grilla que se encuentra sobre la derecha.
Para agregar un nodo hay que dirigirse al menú “Edición y oprimir “Nuevo nodo”



Si se desea agregar una arista, hay que pararse sobre el centro del nodo origen hacer clic y arrastrar el Mouse hasta el nodo destino



Dentro de las opciones se puede acceder a la matriz de conectividad y editar sus valores.
Los números en las etiquetas representan los números de nodos terminales y los valores en las cajas de texto los requerimientos de conectividad.



Se incluyeron algunas funcionalidades auxiliares como calcular el algoritmo de Dijkstra 10 entre pares de nodos, y calcular el costo del grafo.

11.6.Apéndice 6 – Mecanismos de Extensibilidad

En este apéndice se explican los distintos mecanismos para poder extender y configurar la solución brindada en el presente proyecto. Como se explico anteriormente la solución al problema planteado consta de 3 fases: Construcción, Optimización y Confiabilidad, en cada una de las mismas se utilizo un diseño pensado para poder extender y configurar cada una de las distintas fases, pudiendo además combinar las fases extendidas y las implementadas en este proyecto.

Construction

Implementa el algoritmo de construcción de la solución factible. Se debe implementar la interfaz *Construction,* como se muestra en la Figura 42 a continuación. La misma recibe el grafo G y un entero k y devuelve el grafo solución factible y la matriz P.

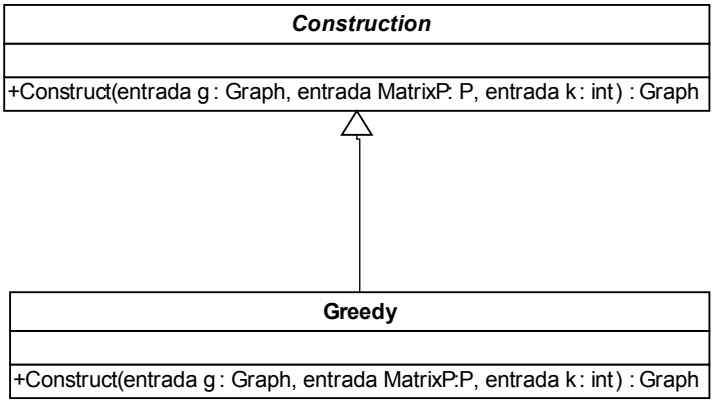


Figura 42: Interfaz de Construcción

Es importante destacar que no existe obligación de utilizar tanto el entero k como la matriz P en la implementación del algoritmo de construcción que se desea utilizar. En este proyecto utilizamos el algoritmo de construcción Greedy 1010 que implementa dicha interfaz y utiliza el parámetro entero k para calcular los caminos mas cortos entre un par dado de terminales y devuelve la matriz P con los caminos nodos disjuntos para cada par de terminales i, j de G. De esta forma se pueden utilizar distintos algoritmos de construcción.

Survivability

Implementa el algoritmo de optimización. Se debe implementar la interfaz *Survivability* como se muestra en la Figura 43 a continuación. La misma recibe un grafo G, una colección y una matriz P y devuelve el grafo solución óptima, luego de ejecutar la meta heurística.

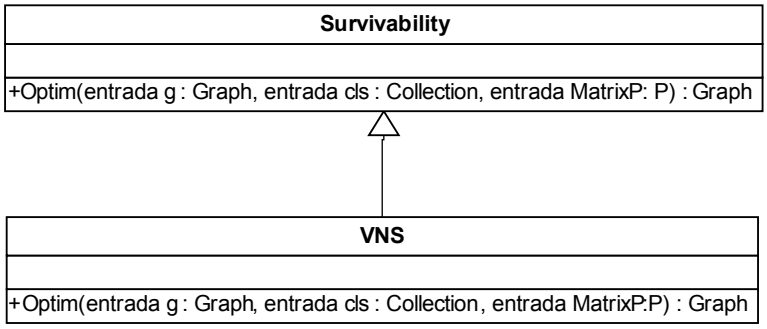


Figura 43: Interfaz de Survivability

Es importante destacar que no existe obligación de utilizar tanto la colección cls como la matriz P en la implementación del algoritmo de optimización que se desea utilizar. En este proyecto utilizamos el algoritmo de optimización VNS 10 que implementa dicha interfaz y utiliza la colección para invocar las distintas búsquedas locales que se deseen utilizar de esta forma este algoritmo se puede configurar para ser utilizado con N búsquedas locales pasadas como parámetro en la colección cls. Este algoritmo además utiliza una búsqueda local que utiliza la matriz P con los caminos nodos disjuntos para cada par de terminales i, j de G. De esta forma se pueden utilizar distintos algoritmos de optimización.

Local Search

Implementa el algoritmo de búsqueda local para una determinada vecindad utilizado como parte del algoritmo de optimización. Se debe implementar la interfaz *LocalSearch* como se muestra en la Figura 44 a continuación. La misma recibe un grafo G y una matriz P y devuelve el grafo solución óptima de la búsqueda en la vecindad en cuestión.

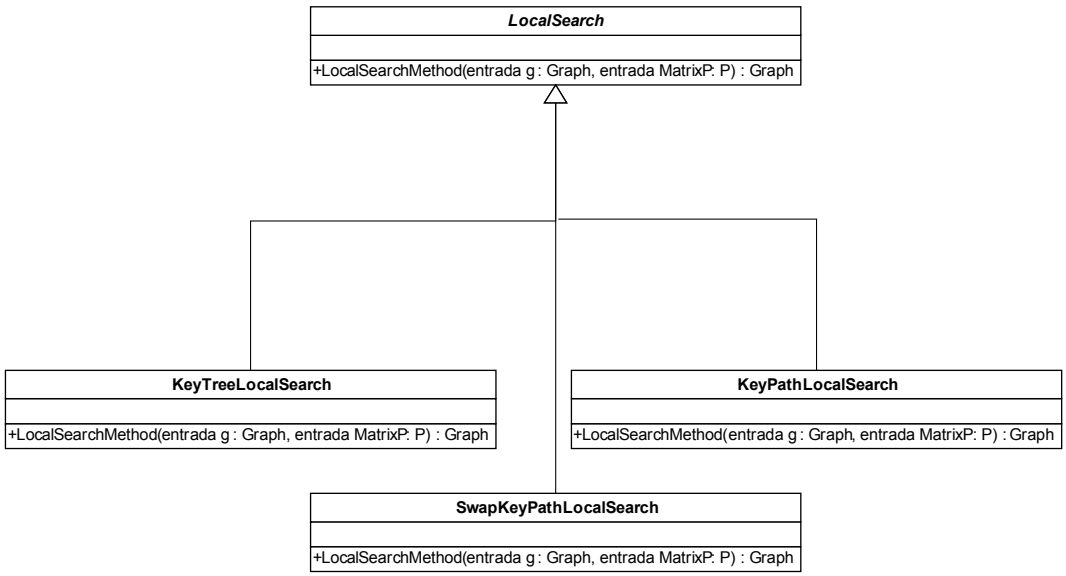


Figura 44: Interfaz LocalSearch

Es importante destacar que no existe obligación de utilizar la matriz P en la implementación del algoritmo de optimización que se desea utilizar. . En este proyecto utilizamos 3 algoritmos de distintos de búsquedas locales y solo uno la utiliza la matriz P con los caminos nodos disjuntos para cada par de terminales i, j de G. De esta forma se pueden implementar N búsquedas locales pudiendo personalizar el algoritmo de optimización para que utilice las N búsquedas locales encargándose el mismo de invocar el método *LocalSearch* cuando sea necesario o utilizarlas en otros algoritmos de optimización, extendidos como se explico anteriormente.

Reliability

Implementa el algoritmo de optimización. Se debe implementar la interfaz *Reliability* como se muestra en la Figura 45 a continuación. La misma recibe un grafo $gsol$, una semilla sem y un número de replicaciones y devuelve la confiabilidad de $gsol$.

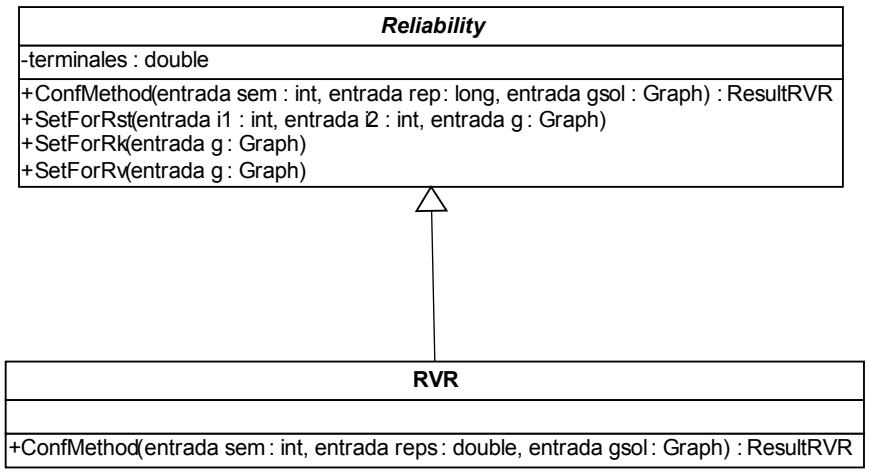


Figura 45: Interfaz Reliability

El único método que se debe implementar obligatoriamente es *ConfMethod*, los otros métodos se pueden utilizar para configurar el tipo de confiabilidad deseado. En este proyecto se implemento el algoritmo de confiabilidad *RVR* 10 en el cual se utilizo el tipo de confiabilidad *Rk*. De esta forma se pueden generar nuevos algoritmos de confiabilidad o personalizar el algoritmo *RVR* 10 pudiéndose utiliza con 3 diferentes medidas de confiabilidad *Rv*, *Rk* y *Rst*.

11.7.Apéndice 7 – Resultados NetworkDesign

A continuación se exponen los resultados obtenidos para aquellas instancias de test en las cuales el algoritmo NetworkDesign fue ejecutado por completo, incluyendo *Greedy, VNS* y *RVR*. Aquellas instancias cuyo porcentaje de terminales está acompañado de (E) corresponden a casos en que la distribución de los requerimientos de conexión es de 65-20-10, siendo el resto de los casos de 100-0-0 (Ver Descripción del Test Set).

att48		
%Terminales	**Resultado**	
	Costo	**Confiabilidad**
20	16331.877154	0.977784
	16683.012554	0.978495
	20668.151653	0.986922
35	26402.111083	0.919162
	27327.616856	0.928013
	30413.012875	0.929602
	33415.033828	0.932021
	34056.520336	0.947725
50	36611.185347	0.909502
	37358.616055	0.927918
	39280.913788	0.933306
35(E)	57424.616281	0.990815
	58154.446514	0.993155
	59008.666856	0.994816
	64559.340812	0.997323

berlin52		
%Terminales	**Resultado**	
	Costo	**Confiabilidad**
20	4534.109370	0.844772
	4637.742346	0.846686
	4651.621341	0.883492
	4688.654791	0.923383
	4931.917157	0.929829
	5130.319348	0.956321
35	4101.517930	0.898807
	4222.507126	0.940985
	4640.251985	0.955959
50	9173.724124	0.868482
	9350.510042	0.870897
	9373.263188	0.904911
	10276.815209	0.937714
20(E)	8211.164402	0.990701
	8247.865606	0.993036
	8363.624552	0.993768
	8433.589024	0.994261
	9489.748702	0.994737

eil51		
%Terminales	Resultado	
	Costo	Confiabilidad
20	140.353333	0.967727
	148.626282	0.967927
	158.474903	0.977934
	173.160827	0.979158
	175.266770	0.983413
35	331.672797	0.921166
	335.521649	0.925231
	373.646632	0.936566
	394.999228	0.937526
50	314.634140	0.898270
	319.095552	0.926383
	324.343604	0.929830
	328.421702	0.930735
	338.145487	0.948727
	367.429922	0.961513
	405.819661	0.965968
20(E)	402.330808	0.996572
	418.998754	0.996912
	433.057622	0.997152
	446.012376	0.997728
50(E)	673.239615	0.984967
	708.543594	0.990023
	719.472647	0.992551
	749.564653	0.993031
	782.910184	0.994058

st70		
%Terminales	Resultado	
	Costo	Confiabilidad
20	327.077759	0.864583
	350.329025	0.908533
	449.073809	0.931741
	473.454474	0.940622
35	393.961047	0.786015
	444.154968	0.855917
	447.123386	0.910170
	489.036874	0.933101
	552.475300	0.934766
50	581.728586	0.878438
	584.135695	0.888196
	632.494988	0.913915
	641.001240	0.921869
	716.774231	0.930525
	784.818795	0.952836
35(E)	1163.557353	0.991151
	1175.975216	0.994590
	1254.705262	0.995233

brazil58		
%Terminales	Resultado	
	Costo	Confiabilidad
20	21861.000000	0.780283
	22072.000000	0.876799
	23793.000000	0.876843
	24534.000000	0.891684
	25106.000000	0.917399
35	12722.000000	0.781356
	12992.000000	0.831314
	14415.000000	0.846178
	15311.000000	0.880375
	17077.000000	0.908160
50	25452.000000	0.885663
	25647.000000	0.888103
	26157.000000	0.919816
	28908.000000	0.924111
	29874.000000	0.940076
50(E)	53827.000000	0.992196
	55910.000000	0.992351
	56531.000000	0.993587
	56923.000000	0.995540

11.8.Apéndice 8 – Cronograma

A continuación se detalla el cronograma del proyecto y la metodología de trabajo seguida para llevarlo a cabo.

Cronograma

1. Interiorización con el problema a resolver.
2. Background en: Confiabilidad Estructural y Sobrevivencia Topológica (con la ayuda de los tutores).
3. Relevamiento del material bibliográfico relacionado con el problema a resolver.
4. Evaluar metodologías de solución (con la guía de los tutores).
5. Diseño e implementación de algoritmos para la solución aproximada del problema.
6. Diseño de casos de prueba y testeo de el/los algoritmos implementados.
7. Evaluación de los resultados obtenidos.
8. Escritura del informe, ajustes de diferentes puntos y defensa.

Nombre de tarea	Duración	Comienzo	Fin
Interiorización del problema	44 días	lun 02/05/05	jue 30/06/05
Background	38 días	mar 10/05/05	jue 30/06/05
Relevamiento del material bil	88 días	lun 02/05/05	mié 31/08/05
Evalución de metodologías	45 días	jue 01/09/05	mié 02/11/05
Diseño e implementación de	41 días	jue 03/11/05	jue 29/12/05
Diseño de casos de prueba	61 días	vie 30/12/05	vie 24/03/06
Evaluación de resultados	32 días	lun 27/03/06	mar 09/05/06
Escritura del informe	111 días	lun 12/12/05	lun 15/05/06

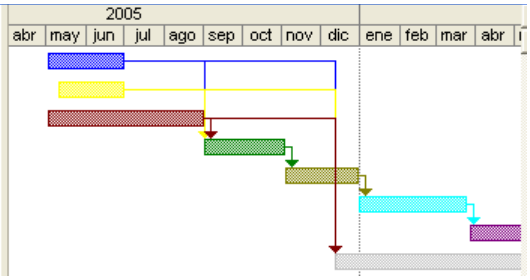


Figura 46: Diagrama de Gantt aproximado

Metodología de trabajo

- Familiarización con diferentes problemas de diseño topológico de redes altamente confiables.
- Interacción con los tutores en aspectos puntuales inherentes a la comprensión de conceptos nuevos requeridos para la realización del proyecto.
- Se debe lograr un Background básico en base a la lectura del material bibliográfico.
- Contemplar técnicas de implementación performantes a nivel de algoritmos y estructuras de datos.

11.9.Apéndice 9 – Índice de Figuras