Universidad Oberta de Cataluña

TRABAJO DE FIN DE MÁSTER

Búsqueda de puntos débiles en redes de comunicaciones mediante algoritmos metaheurísticos

---

Autor: Sergio Pérez Peló

Máster Interuniversitario en Seguridad de las TIC

Algoritmos metaheurísticos aplicados a ciberseguridad

Tutor: Richard Rivera Guevara, Jesús Sánchez-Oro Calvo(URJC)

Junio, 2018

Esta obra está sujeta a una licencia de Reconocimiento-NoComercial-SinObraDerivada 3.0 España de Creative Commons

A mis padres, cuyo esfuerzo e inversión en mí culminan junto a esta etapa de mi vida.
A mis amigos, a mis profesores y compañeros de trabajo, fuente de conocimiento e inspiración contínua.
Gracias por guiar mi camino hasta aquí.

Resumen

La mayoría de las infraestructuras críticas hoy en día se pueden representar mediante redes de nodos interconectados entre sí. Cualquier fallo en uno o más nodos de la red puede tener un impacto notable en la conectividad de la red, pudiendo llevar a desconectar por completo la misma [2, 10]. Además, esta separación puede producir fallos en cascada, ya que los nodos que quedan conectados pueden no tener potencia suficiente para mantener la red, sobrecargándose y dejando de funcionar [9].

Debido a esto, uno de los principales objetivos de un atacante es aislar los nodos cuya eliminación hace que la red se descomponga en subredes aisladas de tamaño mínimo. Sin embargo, los atacantes suelen disponer de recursos limitados, por lo que su objetivo es causar el máximo daño utilizando el mínimo número de recursos. Por su parte, el encargado de la seguridad de la red necesita identificar estos puntos débiles para mantener la integridad de la red.

Este Trabajo Fin de Máster (TFM) tiene como objetivo encontrar un separador α de una red que tenga tamaño mínimo. Un separador es un conjunto de nodos cuya eliminación divide a la red en componentes conexas de tamaño menor que $\alpha \cdot$n, donde n es el número de nodos en la red. Encontrar un separador α mínimo es un problema $\mathcal{NP}$-difícil en topologías genéricas de red para valores de $\alpha \geq \frac{2}{3}$[4]. Para algunas topologías específicas como árboles o ciclos, existen algoritmos que encuentran la solución óptima en tiempo polinómico [11], pero requieren conocer la estructura de la red de antemano, lo que no es común en redes reales.

Dada la complejidad del problema, no es posible diseñar algoritmos exactos para encontrar separadores en redes reales. Por ello, en este TFM se abordará el problema de encontrar un separador α mínimo en redes genéricas mediante algoritmos metaheurísticos. Este tipo de algoritmos son capaces de obtener soluciones de alta calidad necesitando tiempos de ejecución reducidos, pero sin garantizar la optimalidad de la solución encontrada.

En concreto, se utilizará la metodología Variable Neighborhood Search (VNS), la cual se basa en cambios sistemáticos de vecindad para escapar de óptimos locales. Dicha metodología se ha aplicado con éxito en numerosos problemas de optimización gracias a su versatilidad.

Tras la realización de varios experimentos con diferentes metodologías, los re-

sultados obtenidos nos indican que la versión BVNS (*Basic Variable Neighborhood Search*) es la que mejores resultados nos ofrece para la obtención de soluciones para el problema presentado. En concreto, a través de la aplicación de esta metodología se consigue obtener los separadores de tamaño mínimo en un tiempo razonable. Durante el desarrollo del trabajo, se detallarán los experimentos realizados, el conjunto de instancias sobre el cual se llevaron a cabo y los resultados obtenidos.

Palabras clave: Variable Neighborhood Search, Metaheuristics, Networks, Alpha separator

Abstract

Most of today's critical infrastructures can be represented by networks of interconnected nodes. Any failure in one or more nodes of the network can have a noticeable impact on network connectivity, leading to the complete disconnection of the same [2, 10]. In addition, this separation can cause cascading failures, since the nodes that are connected may not have enough power to maintain the network, overloading and not working [9].

Because of this, one of the main objectives of an attacker is to isolate the nodes whose elimination causes the network to decompose into isolated subnets of minimum size. However, attackers usually have limited resources, so their goal is to cause maximum damage using the minimum number of resources. For its part, the person in charge of network security needs to identify these weak points in order to maintain the integrity of the network.

This Masters Thesis (TFM) aims to find a minimum α separator for a network. A separator is a set of nodes whose elimination divides the network into connected components smaller than $\alpha \cdot$ n, where n is the number of nodes in the network. Finding a minimum α separator is a $\mathcal{NP}$-hard problem, in generic network topologies for values of $\alpha \geq \frac{2}{3}$ [4]. For some specific topologies such as trees or cycles, there are algorithms that find the optimal solution in polynomial time [11], but they require knowing the structure of the network beforehand, which is not common in real networks.

Given the complexity of the problem, it is not possible to design exact algorithms to find separators in real networks. Therefore, this TFM will address the problem of finding a minimum α spacer in generic networks using metaheuristic algorithms. This type of algorithms are capable of obtaining high quality solutions, requiring reduced execution times, but without guaranteeing the optimality of the solution found.

Specifically, the Variable Neighborhood Search (VNS) methodology will be used, which is based on systematic neighborhood changes to escape local optima. This methodology has been applied successfully in numerous optimization problems thanks to its versatility.

After carrying out several experiments with different methodologies, the results

obtained indicate that the BVNS version (*Basic Variable Neighborhood Search*) is the one that offers the best results for obtaining solutions for the presented problem. In particular, through the application of this methodology it is possible to obtain the minimum size separators in a reasonable time. During the development of the work, the experiments carried out, the set of instances on which they were carried out and the results obtained will be detailed.

Keywords: Variable Neighborhood Search, Metaheuristics, Networks, Alpha separator

Índice

Resumen **v**

Abstract **vii**

I Búsqueda de puntos débiles en redes de comunicaciones mediante algoritmos metaheurísticos 1

1 Introducción **3**
- 1.1 Estado del arte 5
- 1.2 Análisis de la estructura de las instancias 6
  - 1.2.1 Instancias utilizadas por el mejor algoritmo previo. 6
- 1.3 Representación de la solución 7

2 Objetivos **9**

3 Descripción Algorítmica **11**
- 3.1 Algoritmos metaheurísticos 11
- 3.2 Variable Neighborhood Search (VNS) 12
  - 3.2.1 VNS Básico 14
- 3.3 Algoritmos Constructivos Utilizados para generar soluciones 15
  - 3.3.1 Algoritmo Constructivo Aleatorio 15
  - 3.3.2 Algoritmo basado en el criterio voraz Betweenness Centrality . 15
  - 3.3.3 GRASP 17
  - 3.3.4 Búsqueda local que se aplicará al VNS 18

4 Descripción Informática **21**
- 4.1 Metodología de trabajo 21
- 4.2 Esquema general de código 22
- 4.3 Entorno de trabajo 24

5 Resultados obtenidos **25**
5.1 Descripción formal de las instancias 25
5.2 Experimentos preliminares para evitar sobreajustes 26
5.2.1 Comparativa del algoritmo Betweenness con GRASP 26
5.2.2 Comparativa del algoritmo Constructivo Random vs Constructivo Betweenness vs Constructivo Betweenness GRASP ($\alpha = 0.25$) 27
5.2.3 Algoritmo BVNS con Betweenness GRASP ($\alpha = 0.25$) y distintos valores de k 28
5.3 Experimentos finales con un grupo ampliado de instancias 28

6 Conclusiones y trabajo futuro **31**
6.1 Conclusiones 31
6.2 Objetivos cumplidos 31
6.3 Trabajo futuro 32

Bibliografía **34**

Índice alfabético **35**

Índice de figuras

1.1 Red de prueba 4
1.2 Red de prueba eliminados los nodos 2, 3, 5, 8 4
1.3 Red de prueba eliminados los nodos 0 y 1 5
1.4 Instancias del algoritmo previo 7
1.5 Representación de una solución como conjuntos de aristas 7

3.1 Esquema general VNS (Adaptado de [8]) 13

4.1 Diagrama UML del código desarrollado 23

5.1 Grafo generado con el modelo Erdös-Rényi 25

Parte I

Búsqueda de puntos débiles en redes de comunicaciones mediante algoritmos metaheurísticos

Capítulo 1

Introducción

En este apartado del trabajo se definirá el problema a tratar, además de analizar el estado del arte actual respecto al mismo. Además, se expondrán algunos ataques conocidos asociados al problema que resolveremos, se ejemplificará gráficamente y se explicará en qué nos basamos para decir que una solución es mejor que otra.

En los últimos años la importancia de la ciberseguridad tanto para empresas e instituciones como para usuarios de "*a pie*" se ha ido poniendo cada vez más de manifiesto.

Tanto las organizaciones como los usuarios finales de Internet y sus recursos se están dando cuenta, cada vez más, de que es necesario proteger sus activos de forma contundente para evitar que los ciberataques supongan un daño grave.

En los últimos años, los ataques de denegación de servicio (DoS, del inglés *Denial of Service*) y los ataques de denegación de servicio distribuidos (DDoS) han ido ganando relevancia, puesto que la deshabilitación de un servicio para un proveedor de servicios en Internet supone un daño profundo, tanto económico como en cuanto a imagen corporativa. Además, si del servicio en cuestión dependen otros servicios a su vez, el daño producido es aún mayor.

Un caso que ejemplifica este tipo de perjuicio es el sucedido en octubre de 2016, cuando un ataque de denegación de servicio a la empresa Dyn, que proporciona servicios de DNS en Estados unidos, provocó la caída de numerosos servicios como Twitter, Spotify o PayPal.[1]

Este tipo de ataques representan el problema que se pretende resolver en este TFM. Por un lado, en el lado del atacante, interesa conocer cuáles son los nodos que tienen más importancia para el funcionamiento de una red de modo que se sepa dónde invertir los recursos para obtener un beneficio mayor con el ataque realizado; desde el punto de vista de la defensa, interesa conocer cuáles son los nodos más importantes en nuestra red para destinar más recursos a su protección.

Para ejemplificar el problema de manera sencilla, imaginemos una situación en la

[1]https://en.wikipedia.org/wiki/2016_Dyn_cyberattack

que tengamos una red como la de la Figura 1.1, con un valor α de $\frac{2}{3}$:

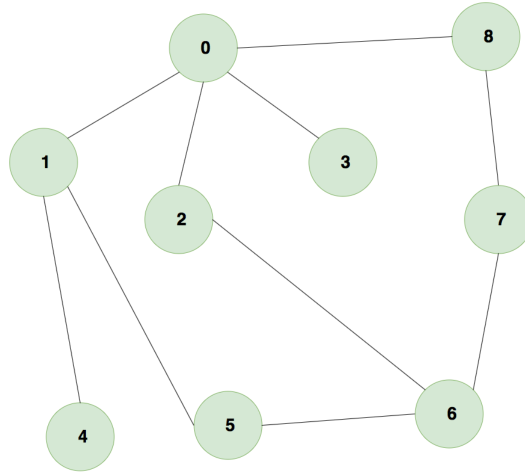


Figura 1.1: Red de prueba

Por ejemplo, una solución para el problema planteado sobre esta red se obtendría eliminando los nodos 2, 3, 5 y 8, la red queda dividida en dos componentes conexas de tamaño menor que $\alpha{\cdot}n$, siendo n el número de nodos de la red (9 en este caso y, por tanto, $\alpha{\cdot}n = 6$), como se puede observar en la Figura 1.2

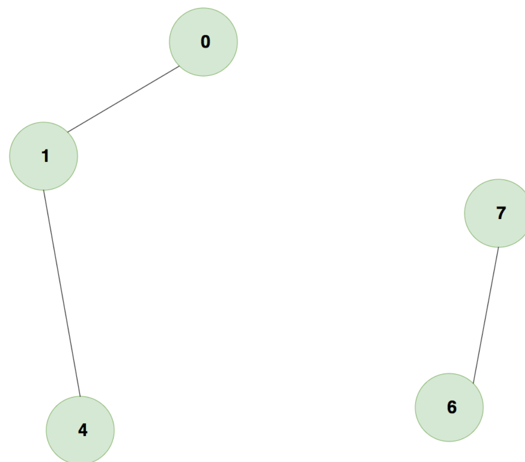


Figura 1.2: Red de prueba eliminados los nodos 2, 3, 5, 8

Sin embargo, podemos llegar a una solución mejor si eliminamos los nodos 0 y 1, tal y como vemos en la figura 1.3. Decimos que una solución es mejor que otra cuando el número de nodos que se han eliminado es menor, ya que son necesarios menos recursos para atacar y defender la red.

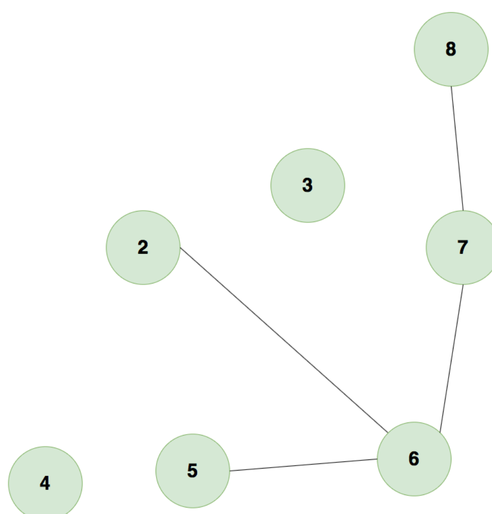


Figura 1.3: Red de prueba eliminados los nodos 0 y 1

Para desarrollar la parte técnica de este TFM, se utilizará el lenguaje de programación Java (con la versión 9 de su JDK). Además, se hará uso de una librería especialmente pensada para desarrollar y resolver problemas de este tipo, desarrollada por el profesor Jesús Sánchez-Oro Calvo, de la Universidad Rey Juan Carlos. Esta librería lleva por nombre GrafoOptiLib, y contiene las interfaces necesarias para representar una instancia, una solución, generar algoritmos constructivos y las funcionalidades básicas para utilizar algoritmos metaheurísticos para resolver problemas de este tipo.

1.1. Estado del arte

Existen numerosos trabajos previos orientados a mejorar los resultados obtenidos en el problema que se trata en este Trabajo Fin de Máster, que se pueden clasificar en tres grupos: aproximaciones deterministas y análisis de la dificultad del problema, problemas relacionados con la búsqueda de puntos débiles, y aproximaciones heurísticas a este problema.

En el primer grupo se puede destacar el trabajo de Mohamed et al. [11] en el que se presentan algoritmos polinómicos para redes especiales (árboles y ciclos). Además, se propone un algoritmo aproximado cuyo ratio es $\alpha{\cdot}\mathrm{n}{+}1$. Por otra parte, se demuestra que un problema muy similar no puede aproximarse en tiempo polinómico con un factor de $\left(\frac{1}{\alpha}\right)^{1-\epsilon}$ para cualquier constante $\epsilon{>}0$, certificando la dificultad del problema. Siguiendo esta línea, Wachs et al. [7] proponen un algoritmo heurístico para encontrar nodos críticos en *Internet Autonomous Systems*.

Dependiendo del tamaño considerado para las componentes conexas en las que se divide el grafo tras el ataque, el problema puede llegar a ser equivalente a otros problemas relevantes. Por ejemplo, para $\alpha=\frac{1}{n}$, es equivalente a encontrar la cobertura mínima de vértices, mientras que para $\alpha=\frac{2}{n}$ se trata del problema del conjunto de disociación. Por lo tanto, el problema tratado en este TFM es una generalización de estos problemas, siendo ambos $\mathcal{NP}$-difíciles [6]. Existen además numerosos problemas difíciles para la separación de grafos con diferentes formulaciones [3, 12].

El mejor algoritmo previo para resolver el problema tratado es el propuesto por Lee et al. [4], el cual propone un algoritmo basado en *random walks* utilizando un método de Monte Carlo y cadenas de Markov.

1.2. Análisis de la estructura de las instancias

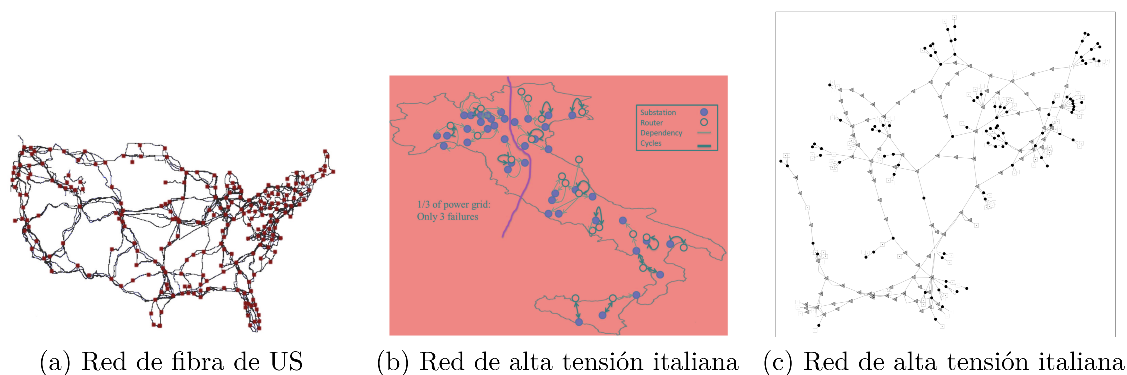
Analizaremos en esta sección las instancias utilizadas por el mejor algoritmo previo.

1.2.1. Instancias utilizadas por el mejor algoritmo previo.

Como se ha comentado en la sección anterior, el mejor algoritmo previo para resolver el problema tratado es el propuesto por Lee et al. [4], el cual propone un algoritmo basado en *random* walks utilizando un método de Monte Carlo y cadenas de Markov.

Las instancias utilizadas por este algoritmo se corresponden con la red de fibra óptica de Estados Unidos para 20 proveedores de Internet (ISP), la red italiana de electricidad y la red italiana de Internet.

Estas son instancias de un tamaño considerable (por ejemplo, en concreto la red de Internet de Estados Unidos es una red de 273 nodos, 2411 aristas y 542 conductos, que se definen como grupos de aristas).



(a) Red de fibra de US (b) Red de alta tensión italiana (c) Red de alta tensión italiana

Figura 1.4: Instancias del algoritmo previo

1.3. Representación de la solución

Para representar las soluciones que se obtengan tras la aplicación del algoritmo, utilizaremos, como ya se mencionó en la introducción del documento, clases que implementen las interfaces de la clase GrafoOptiLib.

Para la representación de una solución, se utilizará una estructura del tipo grafo, que se considera la más apropiada para representar una red interconectada de nodos.

Como sabemos, en computación existen diferentes formas de representar un grafo: listas de adyacencia, matrices de adyacencia, conjuntos, etc. En nuestro caso, dado que es la representación más eficiente que existe para modelar grafos, lo haremos mediante conjuntos de aristas. Es decir, se tendrá una estructura de datos en memoria (concretamente un array) que almacene conjuntos de aristas, de modo que cada posición del array representará un nodo que contendrá un conjunto de los nodos con los que se une (esto es, las aristas salientes de ese nodo). Utilizando este tipo de estructura, tendríamos una complejidad algorítmica en el acceso de $\Theta(1)$, lo cual hace que nuestro algoritmo gane en velocidad y rendimiento.

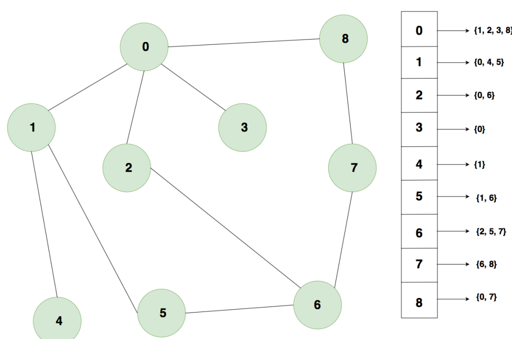


Figura 1.5: Representación de una solución como conjuntos de aristas

Capítulo 2

Objetivos

En este capítulo resumiremos y enumeraremos brevemente los objetivos que se persiguen con este Trabajo de Fin de Máster para que el lector pueda comprender la finalidad del mismo de forma rápida.

1. Resolver el problema $\mathcal{NP}$-difícil de encontrar un separador α de tamaño mínimo en una red, a través del uso de algoritmos metaheurísticos tratando de mejorar los resultados obtenidos por el trabajo previo [4]
2. Comparar los resultados entre diferentes algoritmos constructivos para decidir cuál es el mejor
3. Comparar los resultados entre diferentes criterios para hallar soluciones y decidir cuál encuentra aquellas más cercanas a una solución óptima para el problema planteado.
4. Estudiar las distintas variantes de la metodología VNS
5. Aprender a implementar alguna variante de la metodología VNS
6. Probar diferentes técnicas de generación de soluciones
7. Estudiar el comportamiento de los diferentes algoritmos heurísticos ajustándose a distintos parámetros
8. Estudiar el rendimiento en tiempo de los distintos algoritmos

Capítulo 3

Descripción Algorítmica

En este capítulo se definirán más a fondo los algoritmos metaheurísticos. Posteriormente, se explicará en qué consiste el algoritmo VNS y qué variante del mismo se ha escogido para realizar la búsqueda de soluciones más cercanas a la óptima en el conjunto de soluciones obtenidas por cada algoritmo constructivo. A continuación, se explicarán los algoritmos constructivos que se han desarrollado para obtener dichas soluciones. Además, se expondrán los algoritmos de búsqueda local aplicados para mejorar dichas soluciones. En capítulos posteriores, se mostrarán los resultados obtenidos antes y después de aplicar el algoritmo metaheurístico.

3.1. Algoritmos metaheurísticos

La aplicación de heurísticas y metaheurísticas para resolver problemas complejos es una técnica que nos permite encontrar soluciones cercanas a la óptima (incluso, en ocasiones, la óptima) a dichos problemas que llamamos complejos. Estos problemas suelen ser aquellos que no se pueden resolver mediante un algoritmo exacto, bien por restricciones de tiempo (supondrían un coste demasiado alto respecto a tiempo resolverlos de forma exacta) o porque su complejidad intrínseca es demasiado elevada para encontrar un algoritmo exacto que los resuelva devolviéndonos su solución óptima.

Definimos *heurística* como un proceso que nos permite encontrar una solución (óptima o no) a un problema complejo de una forma rápida.

El problema de las heurísticas es que, a menudo, quedan atrapadas en lo que se conoce como un óptimo local. Un óptimo local es una solución óptima dentro de un conjunto reducido de soluciones para el problema, que no tiene por qué ser un óptimo global.

Un óptimo global es aquella solución del conjunto completo de soluciones que se define como la mejor solución para el problema. Puede haber varios óptimos locales, mientras que sólo puede existir un único óptimo global para un problema (aunque

puede haber varias soluciones distintas que compartan este valor óptimo).

Dado que las heurísticas pueden quedar, como decimos, atrapadas en óptimos locales (caer en una región del conjunto de soluciones de la que no pueda escapar, encontrando continuamente la misma solución óptima), se emplean lo que conocemos como *metaheurísticas*. Una metaheurística es un algoritmo que, podríamos decir, dota de inteligencia a una heurística para ayudarla a escapar de un óptmo local, y encontrar otras posibles soluciones dentro del conjunto de soluciones factibles que mejoren a ese óptimo local.

Será de este tipo de algoritmos del que nos valdremos para resolver el problema que aquí se plantea: la búsqueda de puntos débiles en redes de comunicaciones.

3.2. Variable Neighborhood Search (VNS)

Esta metodología fue propuesta en origen por P. Hansen y N. Mladenovìc[8]. Se trata de una metaheurística, o un marco para construir heurísticas, destinada a resolver problemas combinatorios y de optimización global, que intenta evitar el problema que comentábamos en la introducción del capítulo: que un algoritmo quede atrapado en óptimos locales. Su idea básica consiste en un cambio sistemático de vecindad combinado con una búsqueda local. Desde su inicio, VNS ha experimentado muchos desarrollos y se ha aplicado en numerosos campos. En VNS, si definimos x como una solución del problema, $\forall x \in \mathcal{SS}$ se define un conjunto de vecindades $N_k(x)$, donde $1 \leq \mathrm{k} \leq k_{max}$. Cada una de estas vecindades $N_k(x)$ se puede construir mediante una o varias métricas que dependerán de cada problema concreto.

La metaheurística VNS se basa en los siguientes tres puntos [1]:

1. Un óptimo local con respecto a una vecindad $N_i(x)$ no tiene por qué serlo con respecto a otra vecindad $N_j(x)$.

2. Un óptimo global es un óptimo local con respecto a todas las posibles vecindades.

3. Para muchos problemas, los óptimos locales con respecto a una o varias vecindades, están relativamente próximos.

La última observación es empírica pero implica que, en algunas ocasiones, el óptimo local proporciona información sobre el óptimo global.

Para un problema de optimización concreto, se puede decir que una solución es localmente óptima exclusivamente con respecto a una estructura de vecindad. Esto no tiene por qué ser cierto con respecto a otras estructuras de vecindad. Por lo tanto, la estructura de vecindad determina el perfil del espacio de búsqueda. Estos conceptos se pueden resumir en la siguiente expresión [5]:

Un operador, una superficie de soluciones (One operator, one landscape).

La idea que reside detrás de esta afirmación es que la estructura de vecindad determina las propiedades topológicas del espacio de búsqueda.

Figura 3.1: Esquema general VNS (Adaptado de [8])

Existen varias implementaciones de VNS, que se clasifican teniendo en cuenta fundamentalmente cómo se combinan las tres ideas introducidas con anterioridad. Típicamente, podemos distinguir:

- Combinación mixta: BVNS (*Basic VNS*): En esta implementación también se selecciona de forma aleatoria un punto perteneciente a una vecindad, pero en ésta se mejora utilizando una búsqueda local.

- Combinación estocástica: RVNS (*Reduced VNS*): En esta implementación se selecciona aleatoriamente un punto perteneciente a una vecindad dada, sin realizar ninguna optimización local. RVNS resulta muy útil cuando la busqueda local es costosa.

- Combinación determinista: VND (*Variable Neighbourhood Descent*): En esta implementación, cuando la búsqueda se estanca en un óptimo local, se cambia la estructura de la vecindad para poder escapar de éste (en lugar de perturbar la solución obtenida). La solución final obtenida es un óptimo local con respecto a todas las vecindades consideradas por la metaheurística.

- GVNS: Es el esquema general de VNS, consistente en alterar la solución de la que se parte y realizar el cambio de vecindad. Consiste en una versión de BVNS utilizando VND como búsqueda local.

3.2.1. VNS Básico

Para escapar de óptimos locales en la búsqueda de una solución óptima para nuestro problema optaremos por la versión básica de VNS, o *Basic VNS*. El método VNS básico (BVNS) combina los cambios determinísticos y estocásticos de vecindad. La idea es seleccionar un punto aleatorio en cierta vecindad, y luego aplicarle a ese punto una búsqueda local. Si se encuentra una mejor solución, se reemplaza la solución actual y se empieza de nuevo con el primer esquema de vecindad (k_{step}). Si no, se busca una mejora en el siguiente esquema de vecindad($k + k_{step}$).

En nuestra implementación, se realizan dos fases a destacar que son previas a la búsqueda local: la fase de *shake* y la fase de *reconstruct*.

Durante la fase de *shake* es cuando se altera la solución obtenida a través del algoritmo constructivo que se le pase al BVNS, de modo que podamos escapar del óptimo local. En esta fase, lo que se hace es eliminar nodos de la instancia que no se han eliminado previamente.

Una vez se ha realizado el movimiento de *shake*, se pasa a la fase de reconstruir la solución (*reconstruct*). En este momento, lo que se hace es volver a añadir nodos a la solución de entre los nodos que se habían eliminado antes de la fase de *shake*, de modo que se intenta lograr una solución mejor que la previa, devolviendo al grafo original nodos que se habían eliminado durante la construcción de la solución y eliminando otros distintos.

Cuando se ha reconstruido la solución, que siempre será factible, se aplica una búsqueda local sobre la solución reconstruída. El algoritmo aplicado para realizar la búsqueda local se explica más adelante en este capítulo (sección 3.3.4).

El pseudocódigo correspondiente al algoritmo BVNS implementado es el siguiente:

Algorithm 1 Basic VNS (S, K_{max})

1: **while** $k \leq k_{max}$ **do**
2: $S' \leftarrow shake(sol)$
3: $S'' \leftarrow resconstruct(sol)$
4: $S''' \leftarrow improveSol(sol)$
5: **if** $removedNodesFrom(S''') < removedNodesFrom(sol)$ **then**
6: $sol = S'''$
7: $k = k_{step}$
8: **else**
9: $k = k + k_{step}$
10: **end if**
11: **end while**

3.3. Algoritmos Constructivos Utilizados para generar soluciones

En esta sección se exponen los algoritmos constructivos utilizados para generar soluciones a las que se aplicarán posteriormente los algoritmos metaheurísticos expuestos, tratando así de mejorar la solución de la que se parte.

3.3.1. Algoritmo Constructivo Aleatorio

Esta variante del algoritmo escoge, dada una instancia, un nodo al azar de la misma. Si eliminando ese nodo se consigue una solución factible, el algoritmo para de ejecutarse y devuelve la solución. De lo contrario, elimina otro nodo, hasta que se llegue a una solución factible. El pseudocódigo de dicho algoritmo se corresponde con:

Algorithm 2 Constructivo Aleatorio

```
1: while not esSolucionFactible(S) do
2:     v ← ElegirNodoAleatorio()
3:     S′ ← S − {v}
4:     solucionFactible = esSolucionFactible(S)
5: end while
```

La función esSolucionFactible(S) comprueba la restricción de que la red se divide en componentes o subredes de tamaño menor que $\alpha \cdot n$

3.3.2. Algoritmo basado en el criterio voraz Betweenness Centrality

Esta variante del algoritmo utilizará un criterio voraz para la búsqueda de soluciones mediante el cual escogerá el nodo que posea mayor grado de intermediación (*betweenness centrality*) y lo eliminará del grafo. Esta metodología es muy novedosa, puesto que no se ha utilizado previamente en el campo de las heurísticas.

Este criterio considera que un nodo es importante dentro de una red si actúa como flujo de información del grafo. Es decir, si la mayoría de comunicaciones entre dos nodos cualesquiera pasan por él, esto es, encuentra el nodo que participa en la mayor parte de los caminos más cortos:

$$c_B(v) = \sum_{s,t \in V} \frac{\sigma(s,t|v)}{\sigma(s,t)}$$

donde V es el conjunto de nodos, $\sigma(s,t)$ son los caminos más cortos (en número) con origen en s y destino en t, y $\sigma(s,t|v)$ es el número de esos caminos que pasan por v. Si s = t, $\sigma(s,t) = 1$, y si $v \in s,t$, $\sigma(s,t|v) = 0$.

Para calcular el grado de intermediación de un nodo (v), se deben calcular todos los caminos más cortos desde un punto de la instancia hasta otro (s y t) en los que aparece dicho nodo v, siendo éste comparado con el número total de caminos más cortos entre s y t. Para hallar estos caminos más cortos, se hará uso de una aplicación de código abierto[1]. Esta aplicación encuentra los k-caminos más cortos entre dos puntos dados de un grafo. En concreto, lo hace utilizando el algoritmo de Yen[13] (también conocido como KSP o K Shortest Path). Este algoritmo permite hallar los K caminos de menor coste entre dos vértices determinados de un grafo, evitando que éstos contengan bucles.

Se asume que para hallar el primer camino de menor coste se use un algoritmo como Dijkstra o BFS (búsqueda en anchura). En las iteraciones sucesivas, se procede a realizar algunas modificaciones del camino previo para generar rutas alternativas y ordenarlas por orden de coste. El algoritmo devolverá un array A_k con los K caminos explorados y en orden creciente, es decir, ordenado en base al resultado de calcular el coste mínimo de cada trayecto analizado. Los pasos que realiza el algoritmo son:

1. Hallar el camino de menor coste para determinar $\mathcal{A}\infty$, es decir, la primera iteración del algoritmo con el camino de coste mínimo, a través de un algoritmo de búsqueda del camino más corto.

2. Modificar el camino previo y eliminar el enlace de menor coste para generar una ruta diferente entre el nodo fuente S y el nodo destino D. Almacenar el nuevo trayecto generado en un array temporal B.

3. Repetir el paso 2 para cada enlace del camino más corto hallado anteriormente.

4. Ordenar todos los caminos almacenados en orden ascendente y guardarlos en el array A. Repetir para K-1 veces.

Este algoritmo se ejecutará para encontrar los 100 caminos más cortos entre cada uno de los nodos de las instancias y se almacenarán en un fichero.

Tras realizar este preproceso, el algoritmo realizará el cálculo del grado de intermediación (o *betweenness centrality*) de cada uno de los nodos basándose en esta información y eliminará del grafo el nodo de mayor centralidad. Se irán eliminando nodos de este modo hasta que se alcance una solución factible. El pseudocódigo de este algoritmo se corresponde con:

[1] https://github.com/yan-qi/k-shortest-paths-java-version.git

Algorithm 3 Constructivo Betweenness

1: **while not** esSolucionFactible(S) **do**
2: $v^\star \leftarrow \arg\max_{v \in V \setminus S} \textit{betweenness}(v)$
3: $S' \leftarrow S - \{v^\star\}$
4: solucionFactible = esSolucionFactible(S)
5: **end while**

3.3.3. GRASP

Greedy Randomized Adaptive Search Procedure (GRASP), que en castellano se podría traducir como *procedimientos de búsqueda voraz, aleatorizados y adaptativos*, es un procedimiento multi-arranque en el que cada arranque se corresponde con una iteración. Los procedimientos multi-arranque son un conjunto de procedimientos que intentan evitar quedar atrapados en óptimos locales re-arrancando el procedimiento, lo que permite la diversificación de la estrategia de búsqueda. Cada iteración en GRASP tiene dos fases bien diferenciadas: la de construcción, que se encarga de obtener una solución factible de alta calidad, y la de mejora, que se basa en la optimización (local) de la solución obtenida en la primera fase[1].

Para llevar a cabo la fase de optimización, se elige aleatoriamente un candidato de un conjunto de candidatos que se consideran buenos. Este conjunto recibe el nombre de lista de candidatos reastringidos (Restricted Candidate List o RCL).

Numéricamente, la RCL se construye utilizando los valores máximo y mínimo del coste asignado a los elementos seleccionables en una iteración dada. Este coste se le asigna a los candidatos en función de un criterio voraz (en nuestro caso, el criterio Betweenness).

Si suponemos que g_{max} y g_{min} son respectivamente los valores más alto y más bajo coste, la RCL estará formada por todos aquellos elementos cuyo coste supere (dado que en nuestro algoritmo queremos maximizar) el umbral dado por la siguiente expresión:

$$\mu = g_{max} - \alpha * (g_{max} - g_{min})$$

donde el parámetro $\alpha : 0 \leq \alpha \leq 1$ determina el tamaño de la RCL. Si $\alpha = 1$ en la RCL sólo estará el mejor candidato (función totalmente voraz o voraz pura). Por el contrario, si $\alpha = 0$, en la lista se encontrarán todos los candidatos (función totalmente aleatoria o aleatoria pura).

La RCL se formará según el siguiente pseudocódigo:

Algorithm 4 Generación de la RCL

1: $v \leftarrow Random(V)$
2: $S \leftarrow \{v\}$
3: $CL \leftarrow V \setminus \{v\}$
4: **while not** esSolucionFactible(S) **do**
5: $\quad g_{min} \leftarrow min_{v \in CL} g(v)$
6: $\quad g_{max} \leftarrow max_{v \in CL} g(v)$
7: $\quad \mu \leftarrow g_{min} - \alpha * (g_{max} - g_{min})$
8: $\quad RCL \leftarrow \{v \in CL : g(v) \geq \mu\}$
9: $\quad v \leftarrow Random(RCL)$
10: $\quad S \leftarrow S \cup \{v\}$
11: $\quad CL \leftarrow CL \setminus \{v\}$
12: **end while**

En implementaciones estándares de GRASP, el parámetro α se determina de forma aleatoria. En nuestro caso, realizaremos experimentos preliminares con distintos valores de α y seleccionaremos aquel valor de α que proporcione mejores soluciones. Este algoritmo será el que utilicemos para construir soluciones iniciales en nuestro algoritmo de VNS.

3.3.4. Búsqueda local que se aplicará al VNS

La búsqueda local que se utilzará en nuestro algoritmo consiste en realizar intercambios de nodos 2 *por* 1, esto es, por cada nodo eliminado en una solución se probará a eliminar dos y volver a añadir el nodo eliminado previamente al grafo, buscando mejorar la solución.

El pseudocódigo de esta búsqueda local es el siguiente:

Algorithm 5 Local Search(S)

```
1: improved ← true
2: S_b ← S
3: while improved do
4:     improved ← FALSE
5:     for v ∈ V \ S do
6:         S ← S ∪ {v}
7:         for u ∈ S \ {v} do
8:             S ← S \ {u}
9:             for w ∈ S \ {v} do
10:                S ← S \ {w}
11:                if esSolucionFactible(S) then
12:                    improved = true
13:                    S_b ← S
14:                end if
15:                S ← S ∪ {w}
16:            end for
17:            S ← S ∪ {u}
18:        end for
19:        S ← S \ {v}
20:    end for
21:    S ← S_b
22: end while
```

Capítulo 4

Descripción Informática

En este capítulo se realizará la descripción informática del TFM. Esto es, se explicará la metodología de trabajo utilizada para llegar a la versión final, así como un esquema general del código desarrollado y el entorno de trabajo utilizados.

4.1. Metodología de trabajo

Para la realización del trabajo se ha seguido una metodología de desarrollo ágil, en concreto SCRUM. Esta metodología se basa en realizar iteraciones de trabajo en períodos de tiempo no superiores a dos semanas (aunque dependiendo de la situación este límite de tiempo puede variar, pero nunca será superior a un mes), conocidas como Sprints.

El objetivo de la metodología es obtener un producto de calidad que podría ser liberado como una versión del producto final al acabar cada iteración, de modo que cada Sprint finaliza habiendo aportado valor al producto. En cada iteración se van añadiendo características al producto hasta llegar a una versión de release. Para cada Sprint se establecen una serie de tareas que deben realizarse en el tiempo que dure ese Sprint.

Para la organización de las tareas, se utiliza un tablero SCRUM, que consiste en una pizarra dividida en columnas que incluyen las tareas pendientes, las tareas en proceso y las tareas finalizadas. Opcionalmente, se puede tener una cuarta columna en la que se incluyan todas las tareas a realizar para considerar el trabajo finalizado. A cada tarjeta colocada en el tablero (conocida como historia de usuario) se le asigna un valor numérico en función de su importancia para obtener una versión funcional en el Sprint en cuestión (este valor se conoce como puntos historia). En función de los puntos historia asignados a cada tarea, se priorizan y se añaden al tablero. Para la realización de este trabajo, se utilizó la aplicación web Trello[1] para mantener el

[1]https://trello.com/

tablero Scrum.

En el caso concreto de este TFM, se planificó a 13 semanas (comenzando el 05/03/2018 y estableciendo como fecha final el 04/06/2018, aunque la última semana sirvió para revisar el trabajo final), con Sprints de 2 semanas de duración. El objetivo era que al final de cada Sprint se obtuviese una versión totalmente funcional del TFM, incluida la redacción de la memoria correspondiente a las tareas realizadas en dicho Sprint.

4.2. Esquema general de código

En esta sección se presenta un esquema general del código desarrollado durante el TFM. Se presentará un esquema UML de las clases principales.

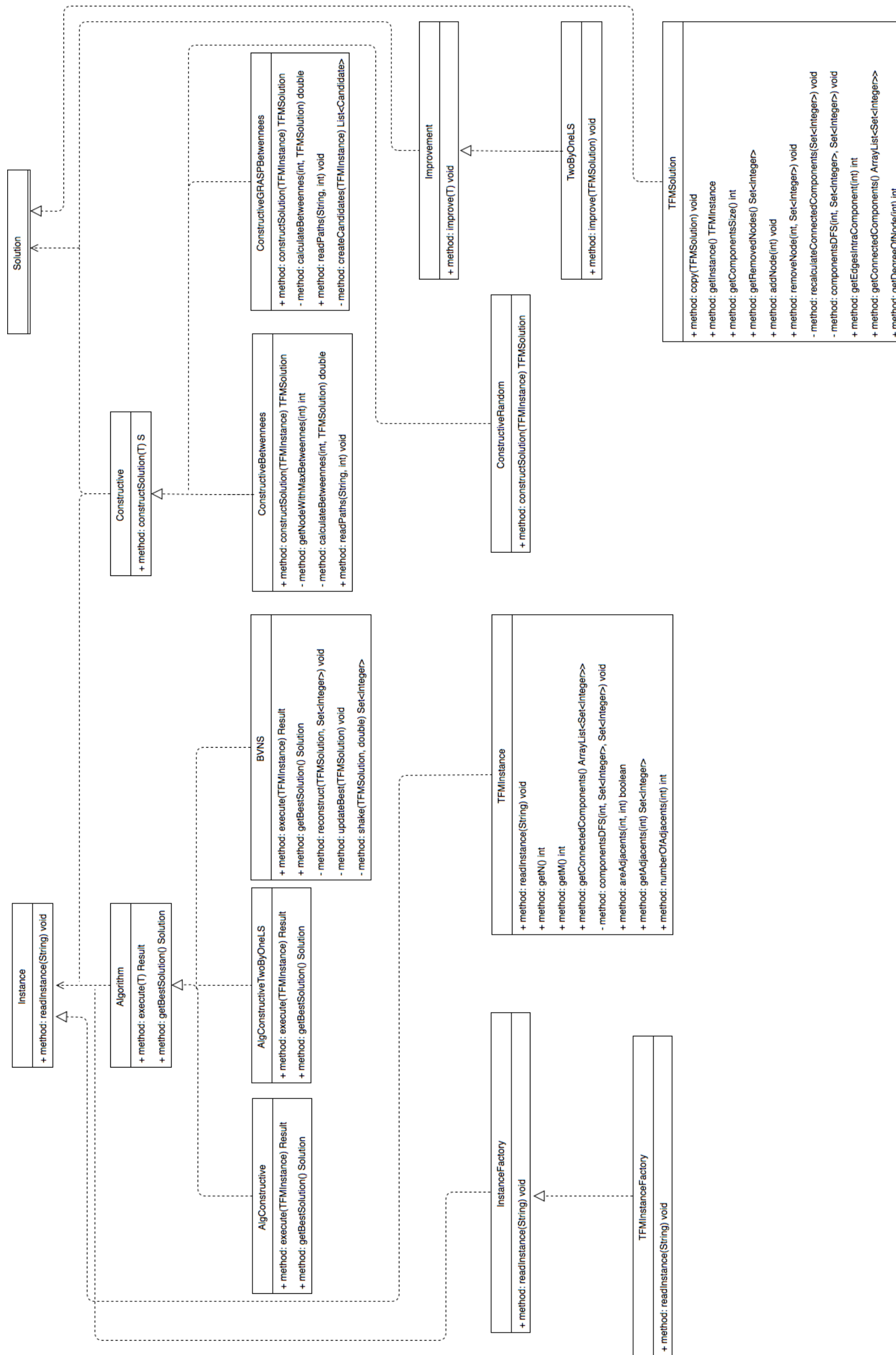


Figura 4.1: Diagrama UML del código desarrollado

4.3. Entorno de trabajo

El código desarrollado para este proyecto ha sido realizado en un entorno de trabajo con las siguientes características:

- Entorno de desarrollo (IDE): IntelliJ IDEA 2017.3.5
- Java JDK 9.0.4
- Macbook Pro Intel Core i7 2,5 GHz, 8 GB RAM 2133 MHz LPDDR3
- Máquina de pruebas Ubuntu Server Intel(R) Core(TM)2 Duo CPU E7300 2.66GHz, 4GB RAM

Capítulo 5

Resultados obtenidos

5.1. Descripción formal de las instancias

Para la realización de este TFM se utilizarán instancias que simulan redes reales a través de grafos. Estas instancias, como ya se ha mencionado, son instancias de gran tamaño. Para simular redes lo más parecidas a redes reales, se utilizarán grafos del tipo Erdös-Rényi. En teoría de grafos, el modelo Erdös–Rényi es uno de los métodos empleados en la generación de grafos aleatorios. En este modelo se tiene que un nuevo nodo se enlaza con igual probabilidad con el resto de la red, es decir, posee una independencia estadística con el resto de nodos. Hoy en día se emplea como una base teórica en la generación de otras redes.

En concreto, nuestro conjunto de instancias está compuesto por grafos de entre 100 y 200 nodos y hasta 2000 aristas aproximadamente.

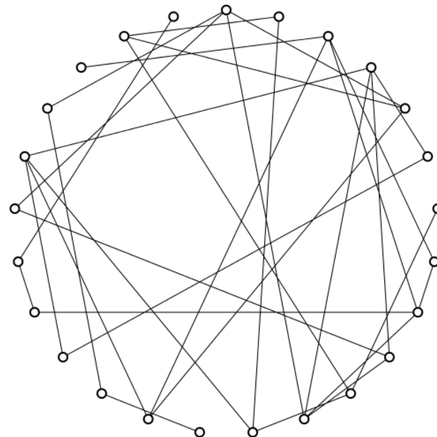


Figura 5.1: Grafo generado con el modelo Erdös-Rényi

5.2. Experimentos preliminares para evitar sobreajustes

Los experimentos que se han realizado para evitar sobreajustes han consistido en ejecutar los siguientes algoritmos sobre un subconjunto de 20 instancias:

- Ejecución del algoritmo constructivo aleatorio con 100 iteraciones y guardando la mejor solución
- Ejecución del algoritmo constructivo con el criterio voraz Betweenness con una iteración
- Ejecución del algoritmo constructivo con el criterio voraz Betweenness con diez iteraciones y aplicando la metaheurística GRASP, con valores de α de 0.25, 0.5, 0.75 y RND (un valor que hace que el algoritmo sea totalmente aleatorio).
- Ejecución del algoritmo BVNS con la búsqueda local de intercambio 2 por 1 con 100 iteraciones, aplicando el algoritmo constructivo de Betweenness con GRASP con con valores de α de 0.25, 0.5, 0.75 y RND para la construcción de soluciones iniciales.
- Ejecución del algoritmo constructivo aleatorio con búsqueda local de intercambio 2 por 1 con 100 iteraciones.
- Ejecución del algoritmo constructivo con el criterio voraz Betweenness con búsqueda local de intercambio 2 por 1 con una iteración

Los resultados obtenidos de estos experimentos preliminares se muestran en las siguientes subsecciones del trabajo. El formato en el que se presentan es tabular, donde cada una de las columnas de la tabla representa, respectivamente: el algoritmo al que corresponden las métricas, la media de nodos eliminados para cada una de las instancias de prueba y ejecuciones del algoritmo, la media de tiempo que ha tardado en ejecutarse el algoritmo para el conjunto de instancias (en segundos), el porcentaje de desviación de la mejor solución para cada uno de los algoritmos y el número de ocasiones en el que se alcanza la mejor solución para cada instancia.

5.2.1. Comparativa del algoritmo Betweenness con GRASP

Los resultados obtenidos tras la ejecución de los experimentos preliminares con 20 instancias (tabla 5.1) nos muestran la siguiente información acerca del algoritmo Constructivo Betweenness aplicando la metaheurística GRASP para la construcción de soluciones iniciales con distintos valores de α:

Algorithm	Removed	Time (s)	Dev(%)	#Best Removed
GraspBetweenness (0.25)	59,8	592,43065	0,59 %	16
GraspBetweenness (0.5)	60,45	590,301	2,08 %	12
GraspBetweenness (0.75)	62,25	590,79775	4,26 %	9
GraspBetweenness (-1.00)	59,85	612,4999	0,94 %	13

Tabla 5.1: Tabla comparativa algoritmo Betweenness con GRASP para distintos valores de α

Se puede observar que, el algoritmo que mejor funciona es el algoritmo que utiliza el criterio Betweenness como método constructivo de soluciones iniciales aplicando la metaheurística GRASP con un parámetro α de 0.25 (lo cual indica un algoritmo cercano a uno puramente voraz). Este algoritmo nos da un total de 16 veces la mejor solución con respecto a los experimentos realizados para el resto de algoritmo con distintos valores de α. El siguiente mejor es el algoritmo que utiliza un α de -1.00 (GRASP totalmente aleatorio). Le siguen los algoritmos con valores α 0.5 y 0.75 respectivamente.

5.2.2. Comparativa del algoritmo Constructivo Random vs Constructivo Betweenness vs Constructivo Betweenness GRASP ($\alpha = 0.25$)

En la tabla 5.2 podemos ver la comparativa entre los siguientes algoritmos Constructivos: el aleatorio, el que sigue el criterio voraz Betweenness y el que utiliza Betweenness aplicando GRASP con un valor $\alpha = 0.25$, es decir, el mejor GRASP visto en la sección anterior.

Algorithm	Removed	Time (s)	Dev(%)	#Best Removed
Random	63,25	0,3038	5,51 %	10
Betweenness	60,65	29,55455	3,01 %	6
GraspBetweenness (0.25)	59,8	592,43065	1,17 %	13

Tabla 5.2: Tabla comparativa algoritmos Random, Betweenness y Betweenness con GRASP ($\alpha = 0.25$)

Como podemos observar, el algoritmo Constructivo Betweenness aplicando GRASP con un valor $\alpha = 0.25$ vuelve a proporcionar las mejores soluciones para un total de 13 instancias, seguido por el algoritmo aleatorio para un total de 10 instancias y el algoritmo que utiliza el criterio Betweenness para un total de 6 instancias.

5.2.3. Algoritmo BVNS con Betweenness GRASP ($\alpha = 0.25$) y distintos valores de k

En esta sección mostraremos los resultados obtenidos tras lanzar los experimentos preliminares aplicando el algoritmo BVNS utilizando el constructivo que utiliza el criterio voraz Betweenness aplicando GRASP con un valor $\alpha = 0.25$ para distintos valores máximos de k (tabla 5.3). Recordemos que en este algoritmo la búsqueda local que se utiliza es el intercambio de dos eliminados de la red por un nodo presente en el grafo, de modo que si obtenemos soluciones factibles con el intercambio, habremos reducido el número de nodos eliminados de la red y, por tanto, habremos encontrado una mejor solución.

Algorithm	Removed	Time (s)	Dev(%)	#Best Removed
BVNS $k = 0{,}05$	50,6	416,72	3,82 %	4
BVNS $k = 0{,}1$	49,95	512,88	2,51 %	8
BVNS $k = 0{,}25$	49,2	768,90	0,67 %	17
BVNS $k = 0{,}5$	49,05	783,66	0,04 %	18

Tabla 5.3: Resultados del algoritmo BVNS para distintos valores de k

Como se puede observar, el algoritmo que mejor resultado nos proporciona es el que utilza un valor máximo de $k = 0.5$. Sin embargo, la mejora que supone respecto al algoritmo que utiliza un valor máximo de k de 0.25 es poco significativa. Solo en 2 ocasiones se mejoran los resultados, y el tiempo que se emplea para alcanzar soluciones es bastante mayor. Por este motivo, seleccionaremos el algoritmo cuyo valor máximo de k es 0.25 para realizar los experimentos finales sobre el conjunto ampliado de instancias.

5.3. Experimentos finales con un grupo ampliado de instancias

En esta sección se mostrarán los resultados finales obtenidos utilizando los algoritmos que han dado mejores resultados en la fase de experimentos previos sobre un conjunto de 50 instancias. En concreto, compararemos entre sí los algoritmos Constructivo Betweenness aplicando GRASP con un valor $\alpha = 0.25$, Constructivo Betweenness aplicando GRASP con un valor $\alpha = 0.25$ aplicando búsqueda local con intercambio 2 por 1, y BVNS utilizando Constructivo Betweenness aplicando GRASP con un valor $\alpha = 0.25$ como algoritmo constructivo para las soluciones iniciales. Al igual que sucedía con los experimentos preliminares, mostraremos una tabla (tabla 5.4) resumen de las mismas características descritas anteriormente.

	Removed Nodes	Time	Desviation (%)	#Best Removed
GRASP (0.25)	61,82	577,54	20,34 %	4
GRASP (0,25) + LS	51,86	987,2	3,30 %	18
BVNS $k = 0.25$	50,82	850,72	0,33 %	44

Tabla 5.4: Tabla de resultados finales

Como podemos observar, el algoritmo BVNS supera con diferencia a los otros dos algoritmos. Además el tiempo empleado en la obtención de soluciones es menor que en el algoritmo que sigue el criterio betweenness con búsqueda local, y no mucho mayor que el mismo algoritmo sin búsqueda local. La diferencia de calidad respecto a las soluciones obtenidas hace que merezca la pena aplicar la metaheurística BVNS para obtener soluciones al presente problema.

Capítulo 6

Conclusiones y trabajo futuro

En este último capítulo se recogen las conclusiones obtenidas tras el desarrollo del trabajo, así como los objetivos cumplidos y el posible trabajo futuro que podría realizarse desde el punto actual.

6.1. Conclusiones

La realización de este trabajo demuestra que en problemas del mundo real es complicado (incluso en ocasiones imposible) llegar a una solución exacta en un tiempo razonable. Las heurísticas y metaheurísticas ayudan a obtener soluciones a estos problemas que, a pesar de no poder garantizar ser óptimas, se acercan a esta optimalidad. En concreto, este problema del separador mínimo es complejo en redes de gran tamaño como las que se presentan hoy en día en nuestro mundo. Es por ello que este trabajo me ha aportado una nueva visión y nuevas herramientas para solucionar problemas que. a priori, pueden parecer imposibles de resolver.

6.2. Objetivos cumplidos

Tras el desarrollo del TFM se ha logrado cumplir con los siguientes objetivos planteados:

- Resolver el problema $\mathcal{NP}$-difícil de encontrar un separador α de tamaño mínimo en una red, a través del uso de algoritmos metaheurísticos: la consecución de este objetivo se logró mediante la aplicación de las metaheurísticas GRASP y VNS, esta última en su versión BVNS.
- Comparar los resultados entre diferentes algoritmos constructivos para decidir cuál es el mejor: este objetivo se alcanzó implementando diferentes ideas iniciales y ejecutando experimentos con un conjunto reducido de instancias para evaluar

los resultados obtenidos y conocer así cuál era la mejor versión para realizar las construcciones iniciales sobre las que se aplicarían los algoritmos heurísticos y metaheurísticos.

- Comparar los resultados entre diferentes criterios para hallar soluciones y decidir cuál encuentra aquellas más cercanas a una solución óptima para el problema planteado: del mismo modo que con el objetivo anterior, esta comparación se realizó a través de experimentos sobre un conjunto reducido de instancias.
- Estudiar las distintas variantes de la metodología VNS
- Aprender a implementar alguna variante de la metodología VNS
- Probar diferentes técnicas de generación de soluciones
- Estudiar el comportamiento de los diferentes algoritmos heurísticos ajustándose a distintos parámetros: este objetivo se consiguió a través de los mismos experimentos realizados previamente para la consecución de los objetivos primero, segundo y tercero.
- Estudiar el rendimiento en tiempo de los distintos algoritmos: al igual que sucede con los objetivos previos, fue la realización de los experimentos preliminares y finales como se evaluó el rendimiento en tiempo de los algoritmos implementados. La clase *Timer* de Java es la que nos proporciona información acerca de este hecho.

Debido a la imposibilidad de contactar con los autores del artículo previo para obtener las instancias sobre las que ellos trabajaron y poder comparar los resultados de su algoritmo frente al nuestro, no se ha realizado dicha comparación.

6.3. Trabajo futuro

Las líneas de trabajo que podrían seguirse para la continuación del presente TFM consisten en:

- Reimplementar el algoritmo previo para poder comparar resultados con el mismo.
- Utilizar nuevos algoritmos de búsqueda local que nos permitan guiar al algoritmo metaheurístico hacia mejores soluciones.
- Buscar nuevos criterios para construir soluciones

Bibliografía

[1] Micael Gallego Carrillo Abraham Duarte Muñoz, Juan Jose Pantrigo Fernández. *Metaheurísticas*. S.L. - DYKINSON, 2007.

[2] R. Farmer N. Hatziargyriou I. Kamwa P. Kundur N. Martins J. Paserba P. Pourbeik J. Sanchez-Gasca et al. G. Andersson, P. Donalek. Causes of the 2003 major grid blackouts in north america and europe, and recommended means to improve system dynamic performance. *IEEE transactions on Power Systems*, 20(4):1922–1928, 2005.

[3] V. Kumar G. Karypis. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[4] H.-W. Lee J. Lee, J. Kwak and N. B. Shroff. Finding minimum node separators: A markov chain monte carlo method. *DRCN 2017*, 2017.

[5] T. Jones. One operator, one landscape. *Technical report, Santa Fe Institure, California, USA*, 1995.

[6] D. S. Johnson M. R. Garey. Computers and intractability: a guide to the theory of np-completeness. *W. H. Freeman New York*, 29, 2002.

[7] R. Thurimella M. Wachs, C. Grothoff. Partitioning the internet. In *International Conference on Risks and Security of Internet and Systems (CRiSIS)*, 2012.

[8] N. Mladenović and P. Hansen. Variable neighborhood search. *Comput. Oper. Res.*, 24(11):1097–1100, 1997.

[9] M. Marchiori P. Crucitti, V. Latora. Model for cascading failures in complex networks. *APS Physical*, 69(4), 2004. Review E.

[10] G. Paul H. E. Stanley S. Havlin S. V. Buldyrev, R. Parshani. Catastrophic cascade of failures in interdependent networks. *Nature*, 464(7291):1025–1028, 2010.

[11] M. A. M. Sidi. *K-separator problem.* PhD thesis, Evry, Institut national des télécommunications, 2014.

[12] C. Jones T. N. Bui. Finding good approximate vertex and edge partitions is np-hard. *Information Processing Letters*, 42(3):153–159, 1992.

[13] J.Y. Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17(11):712–716, 1971.