CADILAC: CAaminos DIsjuntos de Largo ACotado

Natalia Chiappara, Guillermo Lacordelle, Franco Robledo, Pablo Romero*
Facultad de Ingeniería, Universidad de la República
Julio Herrera y Reissig 565. PC 11300. Montevideo, Uruguay

ABSTRACT
The problem under study is to find k node-disjoint paths between two terminals of a network. The network has cost in the links, and the goal is to build k node-disjoint paths at the minimum cost, with the additional requirement that the length of each path must not exceed a fixed positive integer d, called the diameter. This combinatorial problem belongs to the $\mathcal{NP}$-Complete class, and as a consequence there is no exact solution available with polynomial time. In the related literature there are particular solutions to the problem when the diameter is not a constraint, such as Suurballe and Bhandari algorithms. Other authors either work with planar, acyclic graphs or study the case where $k = 2$.
In this work we introduce a GRASP heuristic, named CADILAC, for the general problem. The algorithm combines andomization and adaptive diversity to build paths, and it is inspired by Bhandari and DIMCRA algorithms. The latter was considered to build two link-disjoint paths in a graph with multidimensional costs in the links. Randomization is added to Bhandari' s algorithm during the construction search phase. It also includes some ingredients from DIMCRA. The local search phase has a sub-path replacement strategy to explore the seach space.
Finally, we compare the performance of CADILAC versus a Greedy resolution. We remark that Greedy presents lower computational requirements, but CADILAC achieves both feasible and cheaper solutions.

KEYWORDS: Graph Theory, Network Optimization, GRASP

MSC: 58R10

RESUMEN
El problema de estudio consiste en hallar k caminos nodo-disjuntos entre dos terminales de una red. La red tiene costos en sus aristas, y la construcción de tales caminos se debe realizar a costo mínimo, sujeto a la condición de que cada camino debe tener un largo d o inferior (al entero positivo d se le llama diámetro). El problema pertenece a la clase computacional de problemas NP-Completos, y por lo tanto, no tiene una solución polinomial conocida.
En este artículo introducimos el algoritmo CADILAC (CAaminos nodo-DIsjuntos de Largo ACotado). CADILAC sigue la metaheurística GRASP (del inglés Greedy Randomized Adaptive Search Procedure), y agrega aleatorización y diversidad adaptativa a una resolución golosa inspirada en el algoritmo de Bhandari y del algoritmo DIMCRA.
A efectos comparativos, se desarrolla la noción golosa o Greedy de resolución. El trabajo concluye con la presentación de las pruebas realizadas que comparan a los algoritmos CADILAC y Greedy. Pese a que Greedy es superior en lo que concierne al tiempo de cómputo, CADILAC logra soluciones factibles de menor costo.

1. INTRODUCCIÓN

El problema de determinación de caminos disjuntos entre dos nodos terminales es de gran relevancia en varios contextos.

El problema en su variante más básica, en donde las aristas del grafo tienen costos reales no negativos y se busca minimizar el costo total de la solución, no imponiendo mayores restricciones sobre los caminos además de que sean disjuntos, tiene soluciones polinomiales conocidas [2, 13, 12]. Sin embargo, en varias aplicaciones suelen aparecer restricciones adicionales que vuelven al problema computacionalmente más complejo. Por ejemplo, Guo et al. [6] muestran que el problema de hallar dos caminos arista-disjuntos entre un par de nodos de un grafo, donde las aristas poseen costos multidimensionales y cada camino de la solución debe obedecer a un vector de restricciones, es un problema NP-completo.

*promero@fing.edu.uy

El estudio de la complejidad computacional del problema que consiste en hallar la máxima cantidad de caminos disjuntos entre dos nodos con largo acotado se encuentra en [7]. En este artículo se demuestra que dicho problema es NP-completo. Nuestro trabajo se centra en resolver un problema de optimización similar al anterior que consiste en hallar k caminos simples nodo-disjuntos entre un par de nodos de un grafo cuyas aristas poseen costos reales no negativos, cuyo costo total sea el mínimo posible y con la restricción de que todos los caminos de la solución deben tener a lo sumo d aristas. Este problema es NP-completo [3], por lo que el objetivo del trabajo es el de disear una heurística que lo resuelva, utilizando como marco la metaheurística GRASP.

El presente documento se organiza de la siguiente manera. La Sección 1. contiene una introducción a este trabajo y presenta la formalización del problema que se desea resolver. En la Sección 2. se exponen una serie de artículos que, con algunas variantes, resuelven problemas similares al que se estudia en este trabajo. Algunos de los mas relevantes como el algoritmo de Bhandari y el algoritmo DIMCRA fueron de gran utilidad en el diseo de nuestra solución. En la Sección 3. se presenta una introducción a la metaheurística GRASP, utilizada para la resolución del problema. También se describen los algoritmos necesarios para desarrollar una heurística concreta a partir de GRASP, agregando la noción golosa de resolución del problema. En la Sección 4. se presentan los detalles relativos a la implementación realizada, lenguajes de programación y bibliotecas utilizadas. En la Sección 5. se describen las pruebas realizadas y los resultados obtenidos utilizando como punto de comparación al algoritmo Greedy. Por ltimo, en la Sección 6. se elaboran conclusiones a partir de los resultados obtenidos con la heurística propuesta así como pautas para futuros caminos a seguir a partir de nuestro trabajo.

1.1. Formalización del Problema

Dado un grafo G, cuyas aristas tienen costos positivos, y dos nodos distintos s y t pertenecientes a G, se busca la solución de costo mínimo al problema de hallar k caminos simples nodo-disjuntos entre dichos nodos, con la restricción de que cada camino tenga a lo sumo d aristas.

2. TRABAJOS PREVIOS

Existen varios trabajos relacionados al problema de hallar k caminos nodo o arista disjuntos entre dos nodos de un grafo, como [1, 2, 4, 16, 8, 5, 12, 13, 6], que son los que se detallan a continuación.

Dado que el problema, en su formulación más sencilla, tiene una solución conocida, muchos de ellos tratan variantes que imponen restricciones adicionales sobre los caminos, que transforman el problema original en uno de mayor complejidad computacional. En esta sección veremos varios de ellos, aunque la mayoría no se centre en nuestra versión del problema.

Suurballe fue de los primeros en dar una solución al problema en su formulación básica para hallar dos caminos arista-disjuntos en [12]. La idea principal del algoritmo consiste en utilizar el algoritmo de Dijkstra para hallar el camino más corto entre los nodos de origen y destino, modificar los costos de las aristas sobre dicho camino, y luego correr el algoritmo de Dijkstra por segunda vez sobre el grafo resultante. Ambos caminos hallados son combinados para obtener dos caminos independientes que cumplen con las condiciones del problema. Luego, en [13] se presenta una variación del algoritmo anterior que logra un mejor orden de tiempo de ejecución: $O(m \log_{(1+m/n)} n)$ frente al $O(n^2 \log n)$ del primero.

Más adelante, Bhandari [2] presenta un algoritmo que trata el mismo problema que el algoritmo de Suurballe (ver Algoritmo 1).

En el trabajo de Kobayashi y Sommer [8] se estudia el problema de hallar k caminos nodo-disjuntos, con $k \in \{2, 3\}$, entre pares de nodos s_i, t_i para i=1..k de un grafo G plano y no dirigido, minimizando dos funciones distintas: la suma total de los caminos y el largo máximo de todos los caminos. Este problema es más general en el sentido de que considera mltiples pares de nodos origen y destino que son unidos por cada uno de los caminos de la solución. Los elementos propuestos en este artículo se basan fuertemente en la planaridad del grafo, por lo que no resultan fácilmente aplicables a otros contextos como el del problema de estudio.

Fleischer et al. [5] estudian el problema de hallar k caminos arista o nodo-disjuntos entre dos nodos de un grafo dirigido acíclico (DAG, por sus siglas en inglés) considerando objetivos distintos: MinMax k-DP, *Balanced* k-DP, MinSum-MinMax k-DP y MinSum-MinMin k-DP. El problema MinMax k-DP busca

Algorithm 1 Seudocódigo del algoritmo de Ramesh Bhandari para resolver el problema de hallar k caminos independientes entre un par de nodos de un grafo.

```
procedure Bhandari(G : Grafo, c : E → ℝ⁺, k)
Solución ← φ
while k > 0

    G' ← Grafo()
    V(G') ← V(G)
    E(G') ← E(G) \ Solución ∪ {(u, v) : (v, u) ∈ Solución}
    c' : E → ℝ⁺, c'(e) = { c(e)    ∀e ∉ Solución
                          { −c(e)   ∀e ∈ Solución
    // Camino más corto entre Inicio y Fin.
    // Se usa el algoritmo de Dijkstra, con la
    // variante de que los nodos pueden ser // visitados varias veces.
    SP' ← CaminoMásCorto(Inicio, Fin, G')
    Solución ← (Solución ∪ SP') \ (Solución ∩ SP')
    k ← k − 1

end
return Solución
```

una solución que minimice el costo del camino más costoso. El problema *Balanced* k-DP tiene como objetivo minimizar la diferencia de costos entre el camino de mayor costo y el de menor costo. El problema MinSum-MinMax k-DP busca minimizar el costo total de la solución y, adicionalmente dentro del conjunto de soluciones de costo mínimo, tiene como objetivo secundario minimizar el costo de su camino más costoso. Por ltimo, el problema MinSum-MinMin k-DP es análogo al anterior, pero su objetivo secundario busca minimizar el costo del camino menos costoso. Este algoritmo se basa en el de Perl-Shiloach [5], que es aplicable solamente a grafos acíclicos dirigidos, para generar un grafo intermedio y luego buscar un camino de origen a destino en él.

En el trabajo de Beshir y Kuipers [1] se estudian distintas variantes del problema de hallar dos caminos arista-disjuntos entre un par de nodos, de forma similar al trabajo anterior. Las variantes son Min-Sum Min-Min, Min-Sum Min-Max, *Bounded* Min-Sum y *Widest* Min-Sum. Las variantes Min-Sum Min-Min y Min-Sum Min-Max son análogas a los problemas MinSum-MinMax k-DP y MinSum-MinMin k-DP del trabajo anterior, pero para grafos generales y con $k = 2$. La variante *Bounded* Min-Sum busca hallar una solución que tenga costo total mínimo, pero en donde el costo de cada camino esta acotado inferior y superiormente. Por ltimo, el problema *Widest* Min-Sum define que cada arista tiene un ancho de banda asociado, y la solución busca minimizar el costo total de ambos caminos y que, adicionalmente, para la arista de la solución que tenga el menor ancho de banda, se cumpla que dicho ancho de banda sea el máximo posible. El algoritmo propuesto para tratar estos problemas es básicamente una combinación del algoritmo de Bhandari y el algoritmo para hallar los primeros N caminos más cortos entre un par de nodos.

En el trabajo de Guo et al. [6] se propone una solución al problema de hallar dos caminos arista-disjuntos entre un par de nodos cuyo costo total sea mínimo. La variante sobre el problema que ataca el algoritmo de Bhandari está en que la función de costo no es lineal y en que cada camino debe obedecer a un vector de restricciones. En esta variante del problema, cada arista e tiene asociada un vector de costos w con M componentes w_m. El costo total de un camino P está definido como:

$$c(P) = \max{}_{m=1..M} \left(\frac{w_m(P)}{C_m} \right)$$

donde

$$w_m(P) = \sum_{e \in P} w_m(e)$$

y C es el vector de restricciones. La no linealidad de la función de costo produce que la siguiente propiedad

no se cumpla, como sí ocurre en el caso de costo lineal: Dado el camino más corto P_1 entre dos nodos s y t; si tomamos dos nodos u, v pertenecientes al camino, el fragmento del camino P_1 que une a u con v, es el camino más corto entre u y v. La propiedad anterior produce que en el algoritmo de Bhandari no se formen ciclos negativos, que podrían provocar un bucle infinito en la ejecución. También permite que el algoritmo de Dijkstra halle eficientemente el camino más corto entre dos nodos. Debido a que en el caso no lineal no se cumple esta propiedad, no se puede utilizar el algoritmo de Bhandari directamente para resolver el problema. Por esto es que los autores disean un nuevo algoritmo, DIMCRA, basado en el algoritmo de Bhandari. DIMCRA se diferencia de Bhandari en que es un algoritmo aproximado. A diferencia de Bhandari, al invertir las aristas del camino más corto, los costos no se cambian por su valor opuesto, sino que se establecen en 0. En cada paso, el camino más corto no se halla usando el algoritmo de Dijkstra, sino que en su lugar se utiliza un algoritmo denominado SAMCRA, que se presenta en [15], y se considera un vector de restricciones $C' = 2C$. Finalmente, si luego de combinar ambos caminos alguno de los resultantes no cumple con el vector de restricciones C, las aristas del segundo camino hallado por SAMCRA que no se superponen con el camino más corto se eliminan del grafo, para luego volver a ejecutar la bsqueda. El mayor atractivo de este algoritmo para nuestro problema es que puede ser utilizado directamente para resolver el caso donde $k = 2$, realizando una modificación sobre la función de costo.

Por ltimo, Xiong et al [16] profundizan an más sobre el algoritmo DIMCRA. Se propone una versión exacta del algoritmo, manteniendo el esquema básico. Los resultados de este artículo no son fácilmente generalizables debido a que se centra en el caso de dos caminos nodo disjuntos para ajustar el criterio de dominancia entre caminos utilizado por SAMCRA y el vector de restricciones C' utilizado al momento de realizar bsquedas de caminos cuyo valor resulta ser $C' < 2C$, o sea menor al que utiliza DIMCRA. Sin embargo, la estructura del algoritmo puede ser utilizada de la misma manera para generar una versión exacta para nuestro problema, aunque podría no resultar eficiente.

3. DISEÑO DEL ALGORITMO

3.1. GRASP

El algoritmo que proponemos en este trabajo está basado en la metaheurística GRASP (*Greedy Randomized Adaptive Search Procedure*) [10]. GRASP es una metaheurística que ha demostrado ser efectiva y capaz de producir las mejores soluciones para varios problemas [11].

Esta es una metaheurística iterativa que consiste de dos fases principales que se repiten un nmero predefinido de veces. Estas fases son la de construcción, que tiene como objetivo generar una solución factible utilizando un algoritmo goloso aleatorizado, y la de bsqueda local en donde se recorre una estructura de vecindad definida sobre la solución factible en busca de un óptimo local.

El Algoritmo 2 describe la estructura general de un algoritmo basado en la metaheurística GRASP.

3.1.1. Fase de Construcción

La fase de construcción construye una solución factible en forma golosa y aleatorizada. El proceso comienza (ver Algoritmo 3) con una solución vacía y agrega en cada paso elementos seleccionados de una lista de candidatos llamada RCL (*Restricted Candidate List*). Esta lista contiene *TamañoLista* elementos que se pueden agregar en cada paso, definidos por una función que mide el beneficio de agregarlos utilizando la información disponible a su alcance en ese momento. En general, se aaden a la lista los elementos que producen el menor incremento de costo en la solución parcial, aunque otras estrategias pueden ser utilizadas. Para seleccionar el elemento de la RCL a agregar a la solución, se sortea uno de forma aleatoria. Estos pasos se repiten hasta que se conforma una solución factible para el problema.

3.1.2. Fase de Bsqueda Local

Dado que las soluciones generadas en la fase de construcción no son en general óptimas, es necesario aplicar una bsqueda local para mejorarla. La fase de bsqueda local mejora la solución factible S generada en la fase anterior realizando una bsqueda dentro de una estructura de vecindad $N(S)$ definida para esta.

La efectividad de un algoritmo de bsqueda local depende de varios aspectos, como la estructura de vecindad, la estrategia de bsqueda dentro de dicha vecindad, la velocidad de la función de evaluación de

Algorithm 2 Seudocódigo de heurística en GRASP. $NúmeroIteraciones$ es la cantidad de iteraciones que realiza el algoritmo, $TamañoLista$ es el tamao utilizado para la lista de candidatos utilizada en la fase de construcción y $Semilla$ es la semilla inicial utilizada por el generador de nmeros seudoaleatorios.

```
procedure GRASP(NroIteraciones, TamañoLista, Semilla)
f* ← ∞
for k = 1..NroIteraciones do

    S ← AlgoritmoGolosoAleatorio(TamañoLista, Semilla)
    S ← BúsquedaLocal(S)
    if f(S) < f*then

        S* ← S
        f* ← f(S)

    end

end
return S*
```

Algorithm 3 Seudocódigo del algoritmo de la fase de construcción GRASP. $TamañoLista$ define el tamao máximo para la lista restringida de candidatos. $Semilla$ es la semilla del generador de nmeros seudoaleatorios. El conjunto E contiene los elementos que pueden ser agregados incrementalmente a la solución parcial S.

```
procedure AlgoritmoGolosoAleatorio(TamañoLista, Semilla)
S ← φ
Evaluar costo incremental de cada e ∈ E
while not EsFactible(S) do

    RCL ← GenerarRCL(TamañoLista)
    s ← SeleccionarElementoAleatorio(RCL)
    S ← S ∪ {s}
    Actualizar costo incremental de cada e ∈ E \ S

end
return S
```

costo y la solución inicial. La estrategia de bsqueda dentro de la vecindad puede ser *best-improving* o *first-improving*. En el primer caso, todos los elementos del vecindario $N(S)$ de la solución S son visitados y se elige el mejor de todos, en caso de que alguno supere a S. En el segundo, la solución S es reemplazada por el primer vecino que mejora su costo. Véase el Algoritmo 4 por un ejemplo de algoritmo de bsqueda local *best-improving*.

Algorithm 4 Seudocódigo de algoritmo de bsqueda local *best-improving*.

```
procedure BúsquedaLocal(S)
while not EsOptimoLocal(S) do

    Buscar S' ∈ N(S) tal que f(S') < f(S)
    if Existe(S')

        S ← S'

    end

end
return S
```

3.2. Heurística Greedy

La heurística Greedy es un algoritmo goloso que utilizamos para comparar el desempeo de la heurística CADILAC. Dicho algoritmo se basa en el algoritmo Remove-Find definido en Guo et al. [6], pero adaptado para hallar k caminos independientes, en lugar de solamente dos.

El Algoritmo 5 muestra el seudocódigo de esta heurística. El procedimiento consiste en el agregado iterativo de caminos menos costosos, y respectiva remoción de los nodos y aristas utilizados por éste, salvando los nodos extremos. Esto se repite k veces hasta hallar una solución factible, que es el resultado del algoritmo.

Algorithm 5 Seudocódigo del algoritmo Greedy para hallar k caminos independientes en un grafo G.

```
procedure Greedy(G : Grafo, c : E → ℝ⁺, k, d)
Solución ← φ
while k > 0

    // Busqueda del camino más corto entre
    // Inicio y Fin usando BellmanFord,
    // restringiendo los caminos a d aristas.
    SP ← BellmanFord(G, Inicio, Fin, d)
    G ← G \ SP
    Solución ← Solución ∪ SP
    k ← k − 1

end
return Solución
```

3.3. Heurística CADILAC

Como mencionamos inicialmente, el objetivo del trabajo es desarrollar una heurística basada en la metaheurística GRASP que resuelva el problema planteado. La estructura general del algoritmo desarrollado sigue el esquema de esta metaheurística, utilizando un nmero máximo de iteraciones para detener la ejecución y acumulando la mejor solución en cada iteración. Aun así, existen variantes sobre el esquema general de GRASP que se encuentran en los algoritmos de construcción de soluciones factibles y de bsqueda local.

A continuación describimos el diseo de estos dos algoritmos y sus puntos de contacto con la estrategia general de la metaheurística GRASP. Al algoritmo resultante lo hemos denominado heurística CADILAC, sigla proveniente de algoritmo de bsqueda de CAminos DIsjuntos de Largo ACotado.

3.3.1. Fase de Construcción

La fase de construcción de soluciones factibles se basa en el algoritmo presentado por Bhandari en [2] (ver seudocódigo en el Algoritmo 1 de la sección 2.).

Nuestro algoritmo toma la estructura básica del de Bhandari y modifica algunos de sus elementos para poder generar soluciones factibles aleatorias que cumplan con las restricciones del problema. Algunas de estas modificaciones están inspiradas en el algoritmo DIMCRA presentado en [6]. Como se mencionó en la sección de trabajos previos, DIMCRA adapta el algoritmo de Bhandari para el caso en que los costos de las aristas tienen más de una dimensión y el cálculo del largo de un camino no es lineal, como es nuestro caso.

El primer elemento es la utilización de una función de costo no lineal en el cálculo. En nuestro problema, el costo original y la cantidad de aristas de un camino se maneja como un costo de dos componentes. Por lo tanto, definimos una nueva función de costo $\vec{c}: E \to \mathbb{R}^2$ donde $\vec{c}(e) = (c(e), 1)$, $\forall e \in E$. El costo total de un camino P queda definido entonces como:

$$\vec{c}(P) = \begin{cases} \sum_{e \in P} \vec{c}(e) & si \ \sum_{e \in P} \vec{c_1}(e) \leqslant d \\ \infty & en \, caso \, contrario \end{cases}$$

El segundo elemento es la modificación de la restricción sobre el largo al hallar el camino más corto, que en el caso de nuestro algoritmo, se controla por parámetro. En el algoritmo DIMCRA, en la fase de la bsqueda del camino mas corto, la restricciones se multiplican por 2. En otras palabras, al realizar la bsqueda de un camino, modificaría la función $\vec{c}$ reemplazando d por $2d$. La razón de utilizar esta restricción, y no la original, consiste en que de no hacerlo se reduciría el espacio de bsqueda, aumentando consecuentemente el riesgo de excluir soluciones factibles. En la etapa posterior, en donde se combina el camino encontrado con la solución parcial, si alguno de los caminos resultantes viola la restricción original, se descarta el ltimo camino encontrado, menos las aristas superpuestas con la solución parcial, y se realiza la bsqueda de uno nuevo. En nuestro algoritmo, la restricción sobre la cantidad de aristas por camino, d, se relaja, al momento de realizar la bsqueda del camino más corto, utilizando un parámetro de entrada que multiplica a la restricción original. La razón de no dar un valor fijo a este parámetro reside en que no disponemos al momento de elementos teóricos suficientes como para justificar ninguno. Lo que si es claro es que dicho valor debe ser mayor a 1, puesto que un valor inferior restringiría el espacio de soluciones factibles y que cuanto mas grande sea el valor, más lento será el algoritmo, pues causará que el algoritmo de bsqueda del camino más corto genere mayor cantidad de caminos que derivarán en soluciones no factibles.

El tercer elemento que se toma del algoritmo DIMCRA es asignar costo $(0, 0)$, en lugar del valor opuesto como hace Bhandari, a las aristas que pertenecen a la solución parcial al momento de realizar la bsqueda de un nuevo camino. El algoritmo de Bhandari puede utilizar el valor opuesto debido a que, por más que el grafo resultante contenga aristas de costo negativo, no contendrá ciclos de costo negativo, lo que perjudicaría al algoritmo de bsqueda de caminos. Esto sucede porque en cada paso, el algoritmo de Bhandari construye una solución que es minimal respecto a su costo total en el conjunto de soluciones factibles del problema considerando su misma cantidad de caminos. En otras palabras, en cada paso en el camino de hallar los k caminos, el algoritmo de Bhandari halla la solución para $1 \leqslant n < k$. Por lo tanto, si al momento de invertir las aristas del grafo y asignarles su costo opuesto se genera un ciclo negativo, esto significa que dicha solución no era minimal, puesto que remplazando el fragmento de la solución que pertenece a dicho ciclo, por el fragmento que no pertenece a la solución, se estaría hallando una solución parcial de menor costo, lo que contradice la propiedad mencionada anteriormente. La Figura 1 muestra un ejemplo de esto. Si el algoritmo de Bhandari halló el camino coloreado en negro entre s y t y al invertir las aristas de dicho camino y asignar costo negativo a sus aristas se produjese un ciclo de costo negativo $C = a, b, c, d, e, a$, como en la figura, esto significa que el camino hallado por el algoritmo de Dijkstra en la fase anterior no halló el camino más corto, puesto que si sustituimos el fragmento $P_1 = a, e, d$ por el fragmento $P_2 = a, b, c, d$, el resultado sería una solución de menor costo, lo cual es un absurdo. En nuestro caso, el algoritmo goloso aleatorio no genera soluciones óptimas sino aleatorias, como su nombre indica, utilizando un algoritmo para hallar caminos que no retorna el camino más corto. Por esto es que en nuestro caso si es posible que se generen ciclos de costo

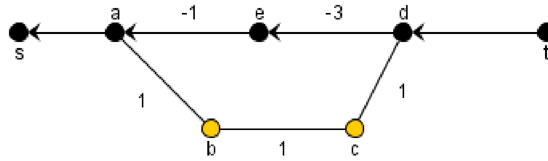

Figure 1: Ejemplo de una situación que no es posible en el algoritmo de Bhandari.

negativo en el grafo al invertir las aristas y esto afectaría el funcionamiento del algoritmo. Por esto mismo es que se le asigna un costo $\vec{c}(e) = (0, 0), \forall e \in \mathit{Solución}$, al momento de invertir las aristas.

Aparte de los elementos incorporados de DIMCRA, otra modificación que realizamos es la aleatorización del algoritmo de Dijkstra utilizado para hallar el camino más corto en cada paso del algoritmo, como se mencionó en el párrafo anterior. La aleatorización es sencilla y consiste en sortear con probabilidad p la posibilidad de visitar un vecino al estar procesando un nodo. Esta simple modificación permite obtener resultados aleatorios que no se alejan demasiado de la solución óptima, si el parámetro p se encuentra en un valor próximo a 1. El algoritmo 6 contiene el seudocódigo del algoritmo de Dijkstra aleatorizado.

El algoritmo 7 describe nuestro algoritmo propuesto para la generación de soluciones factibles para la fase de construcción de nuestra heurística.

Puede observarse que a diferencia del algoritmo de construcción de soluciones factibles de la meta-heurística GRASP, no se genera una lista de candidatos explícita de donde se toman elementos para generar incrementalmente una solución. En nuestro caso, dicha lista esta implícita en la estructura del algoritmo, puesto que en cada paso, los elementos a seleccionar están limitados a los vecinos del nodo visitado.

3.3.2. Fase de Bsqueda Local

La fase de bsqueda local toma la solución factible generada en la fase de construcción por el algoritmo goloso aleatorio y aplica un algoritmo que recorre una estructura de vecindad en busca de una mejor solución.

Estructura de Vecindad

La estructura de vecindad $N : \mathbb{S} \times \mathbb{N}^2 \rightarrow 2^{\mathbb{S}}$, donde $\mathbb{S}$ es el conjunto de todas las soluciones factibles, transforma una solución $S \in \mathbb{S}$, dados dos parámetros n y m, en un conjunto de soluciones factibles, que se denomina vecindad de S segn N y se denota $N(S, n, m)$. El conjunto $N(S, n, m)$ se obtiene a partir de S mediante la aplicación de todos los movimientos del conjunto $M(S, n, m)$ que definimos para nuestro algoritmo sobre este. El conjunto $M(S, n, m)$ está definido por todos los movimientos que reemplazan un subcamino de un camino de la solución S por otro subcamino, manteniendo la factibilidad de la solución resultante y con la condición de que el largo del camino a sustituir en S es menor o igual a n y que el del camino sustituto es menor o igual a m.

La figura 2 muestra ejemplos de movimientos aplicados sobre un camino de una solución, asumiendo que dichos movimientos siempre mantienen la factibilidad de la solución final.

Una estructura de vecindad es transitiva si dadas dos soluciones existe una secuencia de movimientos dentro de esta que transforman una solución en la otra. En nuestro caso, la estructura de vecindad que definimos para el algoritmo de bsqueda local *no* es transitiva, pues existen casos en los que no es posible transformar una solución en otra. La Figura 3 muestra un ejemplo de uno de estos casos. En dicha figura puede observarse las soluciones para el problema de hallar dos caminos nodo-disjuntos en un mismo grafo. Puede notarse fácilmente que cualquier subcamino que intente reemplazarse en alguno de los dos caminos siempre va a resultar en caminos superpuestos, resultando en una solución no factible, por lo que el vecindario de cada solución es vacío. Por lo tanto, no es posible transformar una solución en otra realizando movimientos dentro de la estructura de vecindad, lo que demuestra que no es transitiva.

Algorithm 6 Seudocódigo del algoritmo de Dijkstra aleatorizado.

```
procedure DijkstraAleatrizado(G, s, t, d, p)
// Camino asocia un camino entre s y t a cada nodo.
Camino[v] ← φ, ∀v ∈ G
// La cola de prioridad ordenada por costo.
Cola ← Insertar(Cola, Camino[s])
while not Vacía(Cola)

    C ← ExtraerCaminoDeCostoMínimo(Cola)
    u ← Destino(C)
    if u = t

        return C

    else

        for each v ∈ Vecinos(G, u)

            if Random(0, 1) < p
                P ← C + (u, v)
                if Largo(P) ⩽ d
                and Costo(Camino[v]) > Costo(P)
                    Eliminar(Cola, Camino[v])
                    Camino[v] ← P
                    Insertar(Cola, P)
                end
            end

        end

    end

end
```

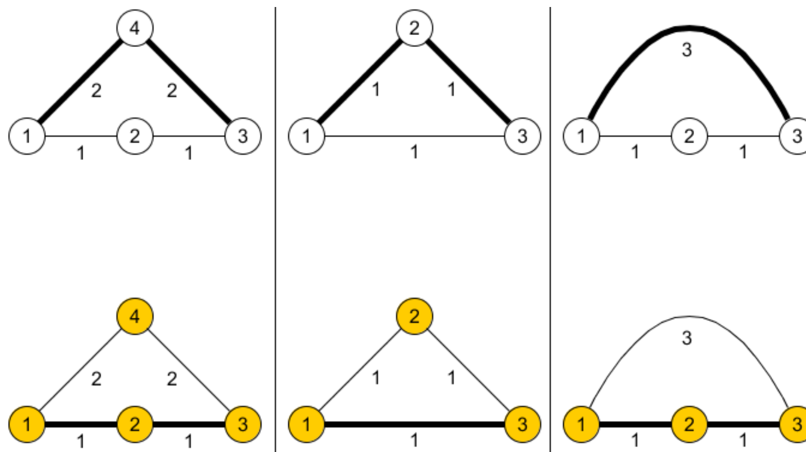

Figure 2: Ejemplos de movimientos. El primero corresponde a una sustitución del camino $(1, 4, 3)$ por el $(1, 2, 3)$, el segundo a la eliminación del nodo 2 y el tercero a la inserción del nodo 2.

Algorithm 7 Seudocódigo del algoritmo goloso aleatorio utilizado para generar soluciones factibles para la heurística.

```
procedure AlgoritmoGolosoAleatorio(G, c, k, MaxIntentos)
Solución ← φ
Intentos ← MaxIntentos
while k > 0 and Intentos > 0

    G′ ← Grafo()
    V(G′) ← V(G)
    E(G′) ← E(G) \ Solución ∪ {(u, v) : (v, u) ∈ Solución}
    c⃗′ : E → ℝ⁺ × ℝ⁺, c⃗′(e) = { c⃗(e)    ∀e ∉ Solución
                                  { (0, 0)  ∀e ∈ Solución
    RP′ ← CaminoAleatorio(Inicio, Fin, G′, p)
    Solución′ ← (Solución ∪ SP′) \ (Solución ∩ SP′)
    if EsFactible(Solución′)

        Solución ← Solución′
        Intentos ← MaxIntentos
        k ← k − 1

    else

        Intentos ← Intentos − 1

    end

end
if k > 0

    return Solución

else

    return Error(′No fue posible hallar una solución′)

end
```

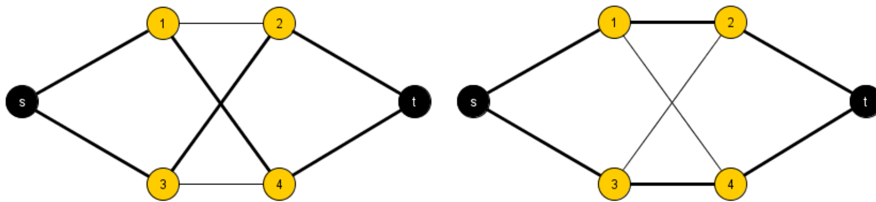

Figure 3: Dos soluciones para el problema de hallar dos caminos nodo-disjuntos en un mismo grafo.

Algoritmo de Bsqueda

El algoritmo de bsqueda define una estrategia mediante la cual se explora el espacio de soluciones definida por la estructura de vecindad N. Como se mencionó en la subsección anterior, el vecindario de una solución S, $N(S, n, m)$, esta definido a partir de un conjunto de movimientos $M(S, n, m)$. Cada uno de los movimientos definidos por M es un cambio que se realiza sobre uno de los caminos de S, por lo que al momento de recorrer el vecindario de S es necesario definir sobre cual camino $P \subseteq S$ se aplicará un movimiento. Una vez que se define el camino P, se puede elegir cual movimiento dentro de los posibles se va a aplicar. De esta forma se obtiene una nueva solución S'. Este proceso se puede repetir hasta que sea necesario. El seudocódigo 8 describe este algoritmo general para recorrer la estructura de vecindad.

Algorithm 8 Estructura general de un algoritmo de bsqueda local para la estructura de vecindad N.

procedure $EstructuraBúsquedaLocal(G, S, n, m)$
while not $CriterioFinalizacion()$

$P \leftarrow SeleccionarCamino(S)$
$P' \leftarrow AplicarMovimiento(P, n, m)$
$S \leftarrow (S \setminus P) \cup P'$

end
return S

Nuestro algoritmo de bsqueda local implementa la estructura anterior definiendo una forma de elegir un camino P de la solución S, una forma de elegir el movimiento que se aplica sobre P y el criterio de finalización de la bsqueda. El seudocódigo 9 describe el algoritmo de bsqueda local utilizado en nuestra heurística. En cada iteración, se recorren todos los caminos de la solución S de forma aleatoria, sin repetirse ninguno. Para cada camino P, se selecciona el mejor movimiento posible. El proceso se repite hasta que no hay más mejoras posibles, o en otras palabras, hasta que se alcanza un óptimo local dentro de la estructura de vecindad.

Este algoritmo no cae dentro de las categorías de *best-improving* o *first-improving* estrictamente, puesto que cada movimiento implica la selección aleatoria de uno de los caminos P, y luego en ese camino sí se aplica una estrategia *best-improving*. Sin embargo, considerando la solución completa, dicho movimiento seleccionado no es necesariamente el mejor.

Algoritmo de Selección del Mejor Movimiento

El algoritmo que selecciona el movimiento a realizar sobre un camino elige el mejor posible segn la estructura de vecindad N. Para lograr esto, recorre todos los nodos del camino y, para cada nodo, toma los distintos subcaminos que comienzan en él, cuyo largo no sea mayor a n. Luego, para cada subcamino, busca el camino más corto posible de largo menor o igual a m (ver sección 3.3.2. sobre la estructura de vecindad). El algoritmo 10 muestra el seudocódigo para el proceso descrito.

Para el algoritmo de bsqueda del camino más corto, en un principio, utilizamos el algoritmo de Bellman-Ford puesto que permite que se acote el largo del camino hallado. Sin embargo, al momento de realizar las pruebas preliminares del algoritmo, este resulto ser lo suficientemente lento como para elevar demasiado el tiempo de ejecución de la heurística. Luego probamos con el algoritmo de Dijkstra, mejorando un poco los tiempos, aunque no lo suficiente. Pensamos que quizás podríamos implementar un algoritmo de bsqueda de caminos que esté más adaptado a la realidad de nuestro problema, por lo que decidimos disear una solución alternativa. De esta forma llegamos al algoritmo de bsqueda de caminos mediante el uso del algoritmo DFS (*Depth First Search*). Este algoritmo avanza a partir del origen y del destino en direcciones opuestas de forma recursiva, agregando aristas en cada paso usando el esquema general de un algoritmo DFS. El algoritmo encuentra un camino cuando un camino parcial del origen se encuentra con un algoritmo parcial desde el destino. Este algoritmo no es mejor que el de Dijkstra o el de Bellman-Ford en el caso general. Pero en nuestro caso, este algoritmo permite fácilmente limitar el largo máximo permitido para el camino hallado, lo que a su vez limita cuantos niveles de recursión realiza el algoritmo. Tomando valores suficientemente

Algorithm 9 Algoritmo de bsqueda local utilizado para optimizar el resultado devuelto por la fases de construcción de la heurística CADILAC.

```
procedure BúsquedaLocal(G, k, S, n, m)
// Identificadores de los caminos de la solución S.
Posiciones ← [1..k]
repeat

    HuboMejoras ← false
    Posiciones ← PermutarAleatoriamente(Posiciones)
    for each i in Posiciones

        P ← ObtenerCamino(S, i)
        P' ← AplicarMejorMovimiento(P, n, m)
        if Costo(P') < Costo(P)

            S ← (S \ P) ∪ P'
            HuboMejoras ← true

        end

    end

until not HuboMejoras
return S
```

pequeos para los tamaos máximos, la velocidad de ejecución de la heurística CADILAC es menor que con las opciones originales, y los resultados obtenidos por el algoritmo de bsqueda local no son significativamente peores. El seudocódigo puede verse en el Algoritmo 11.

4. IMPLEMENTACIÓN DEL ALGORITMO

Para realizar este trabajo construimos 2 programas. El primero implementa el algoritmo propuesto y el segundo implementa la heurística Greedy.

4.1. Lenguaje

El algoritmo fue desarrollado en el lenguaje C++ por ser un lenguaje compilado con una gran cantidad de bibliotecas disponibles. El hecho de que C++ pueda ser compilado a un ejecutable nativo permite tener tiempos de ejecución más contenidos que si se utilizan lenguajes interpretados. Tener una gran disponibilidad de bibliotecas permite encontrar rápidamente soluciones a problemas generales de implementación como la manipulación de entrada y salida, definición de estructuras de datos, etc.

4.2. Bibliotecas

Para el desarrollo de las aplicaciones nos apoyamos en dos bibliotecas. Una para la lectura y reconocimiento de parámetros de línea de comando llamada TCLAP (*Templatized C++ Command Line Parser*) [14], y otra para las estructuras de datos para la representación de grafos y algoritmos asociados a estas estructuras llamada LEMON (*Library for Efficient Modeling and Optimization in Networks*) [9].

TCLAP es una pequea biblioteca que se centra en resolver el problema de la lectura y reconocimiento de parámetros de una aplicación de línea de comandos. Tiene una sintaxis sencilla de utilizar y una curva de aprendizaje prácticamente inexistente, lo que nos atrajo para utilizarlo en nuestra aplicación.

De las bibliotecas específicas para el desarrollo de algoritmos que manipulan grafos, las que más nos interesaron fueron LEMON y BGL. Nos inclinamos por LEMON puse posee una API intuitiva, que facilita la manipulación de grafos, y una batería de algoritmos muy completa.

Algorithm 10 Algoritmo que realiza el mejor movimiento posible dentro de la estructura de vecindad N sobre el camino P.

```
procedure AplicarMejorMovimiento(P, c, d, n, m)
for each u ∈ P

    s ← NodoOrigen(u)
    Costo ← 0
    Largo ← 0
    v ← u
    // Para cada arista u del camino P
    // se recorren n aristas, incluyendo a u
    repeat

        t ← NodoDestino(v)
        Costo ← Costo + c(v)
        Largo ← Largo + 1
        CMáx ← Costo + MejorIncr
        LargoDisponible ← d − (Largo(P) − Largo)
        LMáx ← min(LargoDisponible, m)
        R ← CaminoMásCorto(s, t, CMáx, LMáx)
        if Existe(R)

            MejorReemplazo ← R
            MejorIncr ← Costo(R) − Costo

        end
        v ← SiguienteArco(v, P)

    until Largo = n and Existe(v)

end
if Existe(MejorReemplazo)

    return Reemplazar(P, MejorReemplazo)

end
```

Algorithm 11 Algoritmo de bsqueda del camino más corto mediante el uso de un algoritmo DFS que inicia desde el nodo s y el nodo t.

```
procedure CaminoMásCorto(OP, TP, CostoMax, LargoMax)
// OP camino que sale de origen.
// TP camino que sale desde el destino.
u ← Destino(OP)
v ← Origen(TP)
Costo' ← Costo(OP) + Costo(TP)
Largo' ← Largo(OP) + Largo(TP)
if u = v and Costo' < Costo(SP)

    SP ← OP + TP

else

    if Largo' < LargoMax
    and ∃(u, v) ∈ E(G)
    and Costo' + Costo(u, v) < Costo(SP)

        SP ← OP + (u, v) + TP

    end
    if Largo' + 2 ⩽ LargoMax

        for each (u, o) ∈ E(G) and (i, v) ∈ E(G)

            CaminoMásCorto(OP + (u, o), (i, v) + TP)

        end

    end

end
```

5. RESULTADOS

5.1. Pruebas

Las pruebas consisten en la ejecución de la heurística CADILAC y la heurística Greedy sobre el mismo conjunto de datos para poder realizar comparaciones sobre los resultados obtenidos a partir de ellos. Para realizar estas pruebas se utilizaron grafos aleatorios y representaciones de rutas de Estados Unidos extraídas del noveno concurso de DIMACS [4]. Antes de poder utilizar los archivos de los grafos, los filtramos con un programa para eliminar aristas paralelas, puesto que no están soportadas por las implementaciones de los algoritmos.

5.2. Juego de Datos

Para realizar las pruebas se utilizaron 3 archivos de DIMACS. El primero se llama *Random4-n.10.0* y consiste de un grafo aleatorio de 1024 nodos y 4096 aristas. El segundo también es un grafo aleatorio y se llama *Random4-n.16.0*; contiene 65536 nodos y 262144 aristas. El tercero se llama *USA-road-d.NY* y consiste en un grafo no dirigido que representa un mapa de Nueva York que contiene 264346 nodos y 733846 aristas dirigidas.

El mecanismo de las pruebas fue el siguiente: Para cada uno de los grafos se sorteo un conjunto de varios pares de nodos origen y destino de forma aleatoria y se ejecutó el algoritmo de Suurballe para hallar el camino más largo de su solución. Con un largo mayor a este se ejecutan las heurísticas Greedy y CADILAC. Luego, se repiten las pruebas con valores menores para la cota sobre la cantidad de aristas, hasta que alguna de las heurísticas no halla solución. Los casos en donde ninguna de las heurísticas logran hallar soluciones no está reportado en los resultados.

5.3. Ambiente de Ejecución

Las pruebas fueron realizadas sobre una computadora portátil con un procesador Intel® Core i5-3337U, 3.7 GiB de memoria RAM y sistema operativo Ubuntu 12.04 LTS 64bit.

5.4. Resultados Numéricos

Los Cuadros 1, 2 y 3 muestran, a modo de ejemplo, algunos de los resultados obtenidos en las pruebas realizadas segn se describió en la sección anterior sobre Juego de Datos. La ltima columna de la tabla indica el porcentaje de mejora que logra la heurística CADILAC sobre Greedy. Los casos en los cuales la heurística CADILAC encontró una solución y la heurística Greedy no lo logró están sealados con un '+'. Los casos inversos están sealados con un '-'. Los tiempos de ejecución están reportados en segundos.

El objetivo principal de las pruebas realizadas es averiguar si CADILAC es capaz de hallar mejores soluciones que Greedy, si también es capaz de hallar soluciones en los casos en que Greedy no lo logra, o si ocurre el caso inverso, en donde Greedy halla soluciones en casos donde CADLIAC no logra hallar nada. También registramos los tiempos de ejecución para ver cuánto difieren.

A partir de los datos antes mencionados se obtiene el gráfico de la figura 4, donde puede observarse la proporción de casos en donde CADILAC superó a Greedy y viceversa.

En la prueba con el primer grafo vemos que CADILAC supera a Greedy en casi la mitad de los casos pues logra hallar mejores soluciones o soluciones que Greedy no es capaz de encontrar. En los casos restantes, ambos algoritmos empatan. Si observamos los tiempos de ejecución vemos que, en promedio, a CADILAC le tomó 65 segundos en completar su ejecucion, contra 0.156 segundos que le tomó a Greedy. Esto era de esperar dado el volumen y complejidad de operaciones que se realizan en la heruística CADILAC.

En la prueba con el segundo grafo, CADILAC sigue sacandole ventaja a Greedy, aunque de manera menos contundente que en el caso anterior. Incluso comienza a asomarse un pequeo problema, pues en un caso Greedy halló una solución en donde CADILAC no fué capaz de lograrlo.

Finalmente, en la ltima prueba se observa un comportamiento distnto al de los casos anteriores. Por un lado, CADILAC supera a Greedy en mayor proporción de casos que en las pruebas anteriores. Por otro lado, aparecen casos en donde Greedy supera a CADILAC. Esto en particular no es tan alarmante, puesto que las diferencias de costos en esos casos es relativamente baja si se compara con la diferencia de costos en los casos

Entrada				Greedy		CADILAC		Mejora
s	t	k	d	costo	tiempo	costo	tiempo	
63	427	5	11	11609	0.00276	11107	1.76269	4.32%
			9	12150	0.00245	11544	1.77847	4.99%
			7	11663	0.00108	11663	1.49335	0.00%
			6	-	-	13595	0.83802	+
63	427	2	11	3754	0.00146	3754	0.58929	0.00%
			9	3754	0.00039	3754	0.60367	0.00%
			7	4158	0.00055	4158	0.57471	0.00%
			6	4158	0.00026	4158	0.44590	0.00%
21	412	5	12	11748	0.00363	11679	1.92767	0.59%
			10	11828	0.00107	11727	3.87256	0.85%
			8	12245	0.00272	12079	3.70956	1.36%
			7	12146	0.00068	12146	1.88642	0.00%
21	412	2	12	3854	0.00103	3854	1.50858	0.00%
			7	4114	0.00033	4114	1.60520	0.00%
			5	4528	0.00015	4528	0.51553	0.00%
646	268	5	12	13485	0.00118	13442	3.05050	0.32%
			11	13485	0.00106	13453	2.77576	0.24%
			7	14020	0.00063	14020	0.91301	0.00%
646	268	2	12	3187	0.00157	3187	0.63204	0.00%
			5	3187	0.00039	3187	0.14404	0.00%
575	242	5	12	12128	0.00366	11511	3.99109	5.09%
			9	12128	0.00092	11783	2.91597	2.84%
			7	11783	0.00224	11783	1.23807	0.00%
575	242	2	12	4110	0.00148	4110	2.18925	0.00%
			7	4209	0.00033	4209	1.17831	0.00%
			5	4484	0.00054	4484	0.27507	0.00%
528	98	5	12	14804	0.00099	14096	4.74253	4.78%
			8	15010	0.00082	14484	1.83381	3.50%
			7	-	-	16944	1.29918	+
528	98	2	12	4554	0.00143	4506	2.56661	1.05%
			8	4666	0.00052	4638	1.36790	0.60%
			7	5534	0.00095	5330	1.73124	3.69%
			6	6023	0.00021	6023	1.12245	0.00%
66	224	5	12	10943	0.00359	10943	4.36861	0.00%
			11	11070	0.00346	11070	4.20029	0.00%
			8	11390	0.00276	11390	2.40210	0.00%
66	224	2	12	3487	0.00143	3487	1.93185	0.00%
929	848	5	12	10118	0.00154	10118	4.00224	0.00%
			11	10492	0.00100	10492	4.18849	0.00%
			8	10549	0.00304	10549	2.44468	0.00%
929	848	2	12	2877	0.00137	2877	1.39434	0.00%

Table 1: Resultados numéricos para el grafo Random4-n.10.0

Entrada				Greedy		CADILAC		Mejora
s	t	k	d	costo	tiempo	costo	tiempo	
38760	316	5	13	1027311	0.26885	1027311	77.47830	0.00%
			11	1033594	0.23908	1033594	69.77460	0.00%
			10	1061631	0.15771	1061631	94.91310	0.00%
			9	-	-	1103528	74.92600	+
38760	316	2	13	330609	0.11425	330609	11.19300	0.00%
			12	336892	0.09596	336892	17.22730	0.00%
			9	345821	0.04838	345821	11.76780	0.00%
			8	399281	0.02741	399281	12.66000	0.00%
16599	19775	5	18	941330	0.31852	941330	48.59300	0.00%
			16	949322	0.30281	949322	51.76650	0.00%
			13	949813	0.26557	949813	60.77510	0.00%
			11	1014540	0.18445	1011610	65.56950	0.29%
			10	1110161	0.13818	1104406	52.68120	0.52%
16599	19775	2	12	321404	0.09311	321404	14.31460	0.00%
			11	325234	0.09766	325234	18.07540	0.00%
			10	347136	0.06959	345232	21.10580	0.55%
			8	360164	0.02072	360164	15.78120	0.00%
31647	33248	5	16	959842	0.36526	943980	116.42000	1.65%
			15	959842	0.33348	953010	121.11900	0.71%
			10	993667	0.20656	993667	112.90100	0.00%
			9	1289524	0.12841	1271838	124.33000	1.37%
31647	33248	2	10	289318	0.08286	289318	15.34120	0.00%
			9	350443	0.06174	350443	29.89920	0.00%
			8	406284	0.03206	406284	28.07350	0.00%
5608	1780	5	12	955101	0.24495	942854	89.71730	1.28%
			11	983231	0.24047	978849	80.50630	0.45%
			9	1082457	0.13477	1045872	86.91670	3.38%
5608	1780	2	12	340202	0.10576	340202	25.35470	0.00%
			11	352709	0.10445	352709	42.24360	0.00%
			8	354919	0.03488	354919	17.52790	0.00%
			7	374574	0.01419	374574	9.32795	0.00%
17465	63785	5	14	1176835	0.29872	1176835	133.40300	0.00%
			13	1177575	0.26703	1177575	146.33500	0.00%
			12	1293743	0.22854	1232227	113.34800	4.75%
			11	1293743	0.18679	1293743	138.80500	0.00%
			10	-	-	1372362	103.58800	+
17465	63785	2	13	397073	0.10551	397073	45.02800	0.00%
			12	433628	0.10782	429424	66.82590	0.97%
			11	433628	0.09131	433628	58.47560	0.00%
			10	435065	0.06513	435065	64.92490	0.00%
			9	523033	0.03915	483347	45.85560	7.59%

Table 2: Resultados numéricos para el grafo Random4-n.16.0

Entrada				Greedy		CADILAC		Mejora
s	t	k	d	costo	tiempo	costo	tiempo	
131377	187202	3	500	851169	6.87614	850983	128.953	0.02%
			300	852049	2.96023	851863	137.983	0.02%
			280	852171	2.90346	852053	119.438	0.01%
			270	852486	2.52559	857921	112.696	-0.64%
			260	856669	2.36755	-	-	-
131377	187202	2	500	503098	4.43448	502912	63.692	0.04%
			250	504694	1.55615	504605	77.2971	0.02%
			240	506891	1.43853	504850	73.63	0.40%
177014	166800	3	700	578423	11.7055	556414	52.9875	3.81%
			170	578423	0.836909	556164	44.4514	3.85%
			150	608664	0.708009	-	-	-
177014	166800	2	700	365691	7.76897	363644	28.3297	0.56%
			170	365691	0.579829	363619	28.8091	0.57%
			150	370374	0.538741	370374	17.1552	0.00%
204401	218166	3	700	-	-	1465782	356.745	+
			500	-	-	1474279	266.321	+
204401	218166	2	700	998086	6.07711	866900	210.248	13.14%
			500	998086	5.07668	871412	214.422	12.69%
			400	1007572	3.95778	923135	171.515	8.38%
14624	9495	3	700	340128	8.89344	317548	30.4993	6.64%
			100	340804	0.473076	322336	23.843	5.42%
14624	9495	2	200	209003	0.848514	202516	15.6269	3.10%
			90	-	-	211255	11.8201	+
186986	73410	3	700	1692743	9.00988	1654429	353.493	2.26%
			500	1694659	7.34241	1690604	285.41	0.24%
			400	1775205	6.62669	-	-	-
186986	73410	2	700	1046995	5.54406	1046995	203.135	0.00%
			500	1048911	5.19312	1050219	236.216	-0.12%
			400	1071151	4.57502	1074232	218.123	-0.29%
			350	1120328	3.95453	-	-	-
126313	72919	3	700	618638	8.40751	613743	83.3901	0.79%
			200	619073	1.25659	617734	64.7395	0.22%
			190	622860	1.0967	623619	52.9849	-0.12%
			180	643567	1.15538	-	-	-
126313	72919	2	500	396634	4.80258	388882	58.8643	1.95%
			200	397069	0.945506	391777	56.1769	1.33%
			190	399990	0.839708	401861	51.3259	-0.47%
			180	406105	0.766545	406780	43.6704	-0.17%
			170	413740	0.689639	-	-	-

Table 3: Resultados numéricos para el grafo USA-road-d.NY

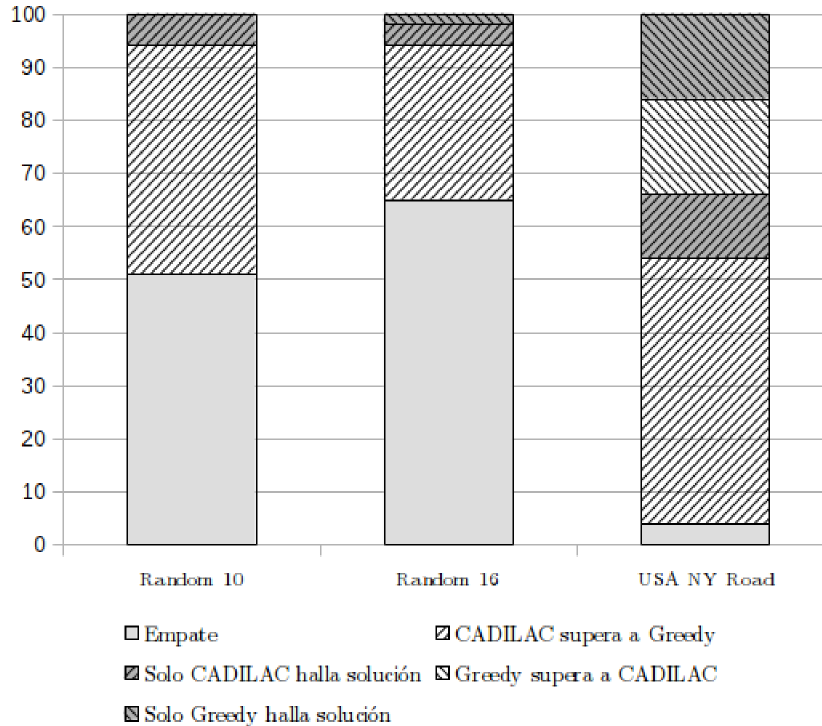

Figure 4: Comparación entre las heurísticas CADILAC y Greedy

que CADILAC aventaja a Greedy. Esto puede mejorarse ajustando parámetros del algoritmo, en particular, el nmero de repeticiones. Lo que si es más relevante es el aumento considerable de casos donde Greedy halla soluciones y CADILAC no lo hace. Esto muestra que probablemente existe cierto sesgo a la hora de generar soluciones factibles aleatorias, lo que le impide encontrar las soluciones que Greedy si fue capaz de hallar.

6. CONCLUSIONES Y TRABAJO FUTURO

Los resultados muestran como la heurística CADILAC soluciona el problema de forma correcta, superando el rendimiento de la heurística Greedy en términos del costo de las soluciones halladas en buena parte de los casos probados. Sin embargo, los tiempos de ejecución del algoritmo propuesto son significativamente mayores, por lo que su aplicabilidad práctica está limitada a los casos en donde tiempos de ejecución del orden de las decenas de segundos, o an mas, son aceptables. En caso de que se requiera un algoritmo veloz, la heurística Greedy es una alternativa viable si el hecho de obtener soluciones con costos ligeramente mayores no es relevante en el problema tratado. Existen, de todas maneras, parámetros que se pueden ajustar para adecuar el rendimiento de la heurística CADILAC como son el nmero de iteraciones, la probabilidad de visita a nodos vecinos en el algoritmo de bsqueda de caminos, el multiplicador del largo máximo de los caminos, también para la bsqueda de caminos, y los largos máximos utilizados en la sustitución de subcaminos en el algoritmo de bsqueda local. Es posible jugar con ellos de forma de obtener el balance deseado entre la precisión del algoritmo y su tiempo de ejecución. Trabajos futuros podrían involucrar la optimización de los parámetros anteriores para distintos tipos de grafos.

Una de las cosas que consideramos importantes para futuros trabajos es la mejora del algoritmo de construcción. Esto se debe al fenómeno que se observó en las pruebas realizadas sobre el grafo USA-road-d.NY. Un algoritmo basado en la aleatorización de Greedy, en lugar de utilizar el algoritmo de Bhandari como base, como hicimos en este trabajo, puede ser un buen candidato.

Al momento de implementar el algoritmo propuesto para tratar el problema presentado en este informe de forma práctica, se debería incluir tanto el algoritmo de Suurballe como la heurística Greedy. Esto es recomendable debido a que la solución hallada por el algoritmo de Suurballe puede llegar a cumplir con las

restricciones de largo. En caso de que la solución elaborada por el algoritmo de Suurballe no cumpla con las restricciones del problema, la heurística Greedy puede presentar una solución tentativa que luego podrá ser superada por la heurística CADILAC. Como el tiempo de ejecución de los algoritmos adicionales es bajo en comparación con el de la heurística CADILAC, su combinación no resultaría mucho mas lenta y sería capaz de brindar más soluciones, puesto que el algoritmo Greedy puede llegar a encontrar algo que el CADILAC no logre hallar, y potencialmente en menor tiempo en el caso de que Suurballe halle una solución que sea factible. Por ltimo, como el algoritmo de Suurballe resuelve una versión relajada del problema, el hecho de que no retorne ninguna solución puede ser utilizado para detener el algoritmo de forma temprana, puesto que tampoco existirá solución al problema original, evitando la ejecución innecesaria de las heurísticas, disminuyendo así el tiempo de ejecución promedio de la heurística combinada.

Una mejora adicional para el algoritmo consiste en utilizar la técnica de *Path-Relinking* [10]. Una forma sencilla de implementar esta técnica se logra mediante el intercambio de caminos no superpuestos entre las soluciones élite y la solución local hallada en cada iteración.

Es difícil realizar comparaciones con otros algoritmos puesto que no encontramos trabajos que presenten alguno que ataque exactamente el mismo problema que estudiamos aquí. Sin embargo, se podría llegar a adaptar el algoritmo DIMCRA de manera que resuelva este problema, para luego comparar ambas soluciones. Dicha adaptación consistiría básicamente en extender el algoritmo para que halle k caminos, en lugar de solamente dos, y en modificar la función de costo utilizada por la misma que se encuentra definida en nuestro trabajo. De todas formas, para que la heurística propuesta en este trabajo resulte competitiva, necesita de las mejoras mencionadas anteriormente.

RECEIVED: JULY, 2015.
REVISED: DECEMBER, 2015.

REFERENCIAS

[1] BESHIR, A. AND KUIPERS, F. [2009]: Variants of the min-sum link-disjoint paths problem In **16th Annual Symposium on Communications and Vehicular Technology (SCVT)**.

[2] BHANDARI, R. [1999]: **Survivable networks: algorithms for diverse routing** Springer.

[3] BLEY, A. [1998]: **On the complexity of vertex-disjoint length-restricted path problems** Citeseer.

[4] DIMACS [2015]: 9th dimacs implementation challenge - shortest paths **http://www.dis.uniroma1.it/challenge9/index.shtml**.

[5] FLEISCHER, R., GE, Q., L., J., AND ZHU, H. [2007]: **Efficient algorithms for k-disjoint paths problems on DAGs**, page 134143 Springer.

[6] GUO, Y., KUIPERS, F., AND MIEGHEM, P. V. [2003]: Link-disjoint paths for reliable qos routing **International Journal of Communication Systems**, 16, 9,:779798.

[7] ITAI, A., PERL, Y., AND SHILOACH, Y. [1982]: The complexity of finding maximum disjoint paths with length constraints **Networks**, 12, 3,:277 286.

[8] KOBAYASHI, Y. AND SOMMER, C. [2010]: On shortest disjoint paths in planar graphs **Discrete Optimization**, 7, 4,:234245.

[9] LEMON [2015]: Library for efficient modeling and optimization in networks **http://lemon.cs.elte.hu/trac/lemon**.

[10] RESENDE, M. G. P. P. [2008]: **Handbook of Optimization in Telecommunications** Springer.

[11] RIBEIRO, C. AND HANSEN, P. [2002]: **Essays and surveys in metaheuristics** Kluwer Academic Publishers.

[12] SUURBALLE, J. [1974]: Disjoint paths in a network **Networks**, 4, 2,:125145.

[13] SUURBALLE, J. W. AND TARJAN, R. E. [1984]: A quick method for finding shortest pairs of disjoint paths **Networks**, 14, 2,:325336.

[14] TCLAP [2015]: Templatized c++ command line parser library **http://tclap.sourceforge.net/**.

[15] VAN MIEGHEM, P., NEVE, H. D., AND KUIPERS, F. [2001]: Hop-by-hop quality of service routing **Computer Networks**, 37, 3,:407423.

[16] XIONG, K., D. QIU, Z., GUO, Y., AND ZHANG, H. [2009]: Multi-constrained shortest disjoint paths for reliable qos routing **ETRI journal**, 31, 5,:534544.