Caminos nodo-disjuntos, acotados y de menor costo entre dos nodos

Estudiantes

Natalia Chiappara

Guillermo Lacordelle

Tutores

Pablo Romero

Franco Robledo

16 de marzo de 2014, Montevideo, Uruguay.

Informe de Proyecto de Grado presentado al Tribunal Evaluador como requisito de graduación de la carrera Ingeniería en Computación.

Resumen

Este trabajo presenta una heurística capaz de resolver el problema de hallar k caminos nodo-disjuntos entre dos nodos de un grafo de aristas de costos reales no negativos, con la restricción de que todos los caminos deben tener a lo sumo d aristas, minimizando el costo total de la solución. La motivación del diseño de una heurística reside en el hecho de que el problema que se intenta resolver no tiene una solución polinomial conocida.

Existen diversos artículos que estudian problemas relacionados con este trabajo. En particular, la variante en donde no se restringen los largos de los caminos puede ser resuelta por algoritmos exactos y polinomiales, como son los de Suurballe y Bhandari. Algunos artículos tratan problemas similares, pero con hipótesis adicionales, como la de requerir que el grafo sea dirigido y acíclico, o que el grafo sea plano, o el caso donde la cantidad de caminos a hallar es solamente 2. De estos trabajos se extraen ideas que se aplican al nuestro; en particular del algoritmo de Bhandari y del algoritmo DIMCRA, donde este último estudia el problema de hallar 2 caminos arista-disjuntos en un grafo donde los costos de las aristas son multidimensionales y existe un vector de restricciones que ambos caminos deben cumplir.

El diseño de la heurística se basa la metaheurística GRASP (*Greedy Randomized Adaptive Search Procedure*), centrándose en la definición de los algoritmos necesarios para su implementación: la fase de construcción de soluciones factibles y la fase de búsqueda local. La fase de construcción se inspira en el algoritmo de Bhandari, con la diferencia de que el proceso de generación de caminos es aleatorio y también incluye elementos del algoritmo DIMCRA. El algoritmo de búsqueda local utiliza una estrategia de sustitución de subcaminos para explorar el espacio de búsqueda. Al algoritmo resultante lo denominamos CADILAC. También se desarrolla una heurística simple, denominada Greedy, para compararla con el algoritmo propuesto durante el análisis de resultados. La implementación de ambos algoritmos fue realizada en C++, utilizando la biblioteca LEMON para la manipulación de grafos.

El trabajo concluye con la presentación de las pruebas realizadas que comparan a los algoritmos CADILAC y Greedy. Mostramos cómo el algoritmo que proponemos no es comparable con los tiempos de ejecución de la heurística Greedy, sin embargo, logra obtener mejores costos en la mayoría de los casos.

Palabras Clave: grafo, heurística, GRASP, caminos disjuntos

Índice general

Capítulo 1

Introducción

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

Un ejemplo es el aumento de la confiabilidad de una red, que puede lograrse mediante la definición de más de un camino entre un nodo fuente y su destino. Esta técnica es efectiva cuando las interrupciones y atrasos deben ser mínimos al momento de realizar modificaciones sobre la topología que afecten el camino principal, o si la variación de carga hacen que dicho camino deje de ser el óptimo. Si los enlaces son propensos a fallas, el origen puede enviar paquetes duplicados de forma simultánea en caminos paralelos para aumentar la confiabilidad. El nodo origen también puede monitorear el desempeño de los caminos y seleccionar el mejor para el envío de datos. En el caso de que se necesite enviar datos con mayor ancho de banda del disponible en un solo camino, pueden utilizarse caminos disjuntos para lograrlo. Cuando existen múltiples caminos independientes entre una fuente y un destino en una red se logra aumentar la confiabilidad y el rendimiento [1, 2].

Una variante del problema que consiste en hallar caminos disjuntos entre varios pares de nodos es relevante en el contexto del diseño de circuitos VLSI. En este caso, la resolución del problema también es importante para poder brindar tolerancia a fallos y evitar cuellos de botella en los circuitos [3].

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 [4, 5, 6]. Sin embargo, en las aplicaciones que mencionamos anteriormente suelen aparecer restricciones adicionales que vuelven al problema más complejo computacionalmente. Por ejemplo, Guo et al. [7] muestran que el problema de hallar 2 caminos arista-disjuntos entre un par de nodos de un grafo, donde las aristas poseen costos

vectoriales y cada camino de la solución debe obedecer a un vector de restricciones, es un problema NP-completo.

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 [8]. 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 NPO-completo [9], por lo que el objetivo del trabajo es el de diseñar una heurística que lo resuelva, utilizando como marco la metaheurística GRASP.

1.1. Organización del Documento

El presente documento esta organizado en 6 capítulos que se encuentran a su vez subdivididos en secciones. El Capítulo 1 contiene una introducción a este trabajo y presenta la formalización del problema que se desea resolver. En el Capítulo 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 diseño de nuestra solución. En el Capítulo 3 se presenta una introducción a la metaheurística GRASP, utilizada para la resolución del problema, y de la técnica de postoptimización Path-Relinking. También describimos todos los seudocódigos de los algoritmos necesarios para desarrollar una heurística concreta a partir de GRASP y definimos al algoritmo Greedy que resuelve el mismo problema que nuestra heurística, pero utilizando una estrategia golosa. Luego en el Capítulo 4 se presentan los detalles relativos a la implementación realizada, lenguajes de programación y bibliotecas utilizadas, estructura del código fuente e instrucciones para la compilación y ejecución de los programas desarrollados. En el Capítulo 5 se describen las pruebas realizadas y los resultados obtenidos utilizando como punto de comparación al algoritmo Greedy. Por último, en el Capítulo 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.

1.2. Formalización del Problema

Consideramos un grafo $G(V, E)$, donde V es el conjunto de nodos y E es el conjunto de aristas, una función de costo $c : E \to \mathbb{R}^+$ y dos nodos $s, t \in V$

llamados nodos de origen y destino. Dado un camino simple P entre los nodos s y t, definimos como $V(P)$ al conjunto de nodos de P, $E(P)$ al conjunto de aristas de P, $c(P) = \sum_{e \in E(P)} c(e)$, el costo asociado a P y $l(P) = |E(P)|$ el largo de P. Dados dos enteros positivos k y d, el problema consiste en identificar k caminos simples P_i, $1 \leq i \leq k$ entre un par de nodos $s, t \in V$ tales que $l(P_i) \leq d$, $\forall i = 1..k$, $V(Pi) \cap V(Pj) = \{s, t\}$, $\forall i, j = 1..k$, $i \neq j$ y cuyo costo total sea el mínimo posible.

Capítulo 2

Trabajos Previos

Existen varios trabajos relacionados al problema de hallar k caminos nodo o arista disjuntos entre dos nodos de un grafo, como [10, 4, 11, 12, 13, 14, 6, 5, 7], 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 2 caminos arista-disjuntos en [6]. 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. El seudocódigo del proceso puede encontrarse en el Algoritmo 2.1. Luego, en [5] 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 [4] presenta un algoritmo que trata el mismo problema que el algoritmo de Suurballe (ver Algoritmo 2.2). También describe la técnica de separación de vértices para extender un algoritmo que halla caminos arista-disjuntos para que logre encontrar caminos nodo-disjuntos.

En el trabajo de Kobayashi y Sommer [13] 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 múltiples pares de nodos origen

Algoritmo 2.1 Algoritmo de Suurballe para hallar 2 caminos nodo disjuntos entre los nodos s y t de un grafo G con costos no negativos, donde $w(u, v)$ es el costo asociado a la arista (u, v) y $d(s, u)$ es la distancia del camino mas corto entre los nodos s y u.

```
procedure Suurballe
Hallar el árbol de caminos mas cortos T en G
P1 ← Camino más corto de s a t en G
for each (u, v) ∈ E(G)
    w'(u, v) ← w(u, v) − d(s, v) + d(s, u)
end
G' ← Eliminar las aristas sobre P1 que apuntan hacia s
G'' ← Invertir todas las aristas de P1 en G'
P2 ← Camino más corto entre s y t en G''
Solución ← (P1 ∪ P2) \ (P1 ∩ P2)
return Solución
```

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. [14] estudian el problema de hallar k caminos arista o nodo-disjuntos entre 2 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 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 último, 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 [14], que es aplicable a DAG solamente, para generar un grafo intermedio y luego buscar un camino de origen a destino en él.

En el trabajo de Beshir y Kuipers [10] se estudian distintas variantes del problema de hallar 2 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

Algoritmo 2.2 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
    // Se crea un grafo G' ...
    G' ← Grafo()
    // ... que comparte vértices con G ...
    V(G') ← V(G)
    // .. y sus aristas, salvo aquellas que pertenecen
    // también a Solución ...
    E(G') ← E(G) \ Solución ∪ {(u, v) : (v, u) ∈ Solución}
    // ... y el costo de cada arista, a menos que
    // pertenezcan a Solución, en cuyo caso se le asigna
    // el costo opuesto.
    c' : E → ℝ⁺, c'(e) = { c(e)   ∀e ∉ Solución
                         { -c(e)  ∀e ∈ Solución
    // Se halla el camino más corto entre Inicio
    // y Fin en el grafo G'. El algoritmo usado
    // es el de Dijkstra, con la variante de que
    // los nodos pueden ser visitados varias veces.
    SP' ← CaminoMásCorto(Inicio, Fin, G')
    // Se combinan las aristas del camino hallado
    // con la solución parcial, eliminando las
    // aristas comunes a ambos conjuntos.
    Solución ← (Solución ∪ SP') \ (Solución ∩ SP')
    k ← k − 1
end
return Solución
```

total mínimo, pero en donde el costo de cada camino esta acotado inferior y superiormente. Por último, 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. [7] se propone una solución al problema de hallar 2 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:

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

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 (Bhandari): 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 2 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 diseñan 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 búsqueda. 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 último, Xiong et al [12] profundizan aún 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 2 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 búsquedas 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.

Capítulo 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*) [16]. GRASP es una metaheurística que ha demostrado ser efectiva y capaz de producir las mejores soluciones para varios problemas [17].

Esta es una metaheurística iterativa que consiste de dos fases principales que se repiten un número 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 búsqueda local en donde se recorre una estructura de vecindad definida sobre la solución factible en busca de un óptimo local.

El algoritmo de la Figura 3.1 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 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 añaden 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.

Algoritmo 3.1 Seudocódigo de una heurística basada en GRASP. $NúmeroIteraciones$ es la cantidad de iteraciones que realiza el algoritmo, $TamañoLista$ es el tamaño utilizado para la lista de candidatos utilizada en la fase de construcción y $Semilla$ es la semilla inicial utilizada por el generador de números seudoaleatorios.

```
procedure GRASP(NúmeroIteraciones, TamañoLista, Semilla)
f* ← ∞
for k = 1..NúmeroIteraciones do
    S ←
    AlgoritmoGolosoAleatorio(TamañoLista, Semilla)
    S ← BúsquedaLocal(S)
    if f(S) < f* then
        S* ← S
        f* ← f(S)
    end
end
return S*
```

Algoritmo 3.2 Seudocódigo del algoritmo de la fase de construcción GRASP. $TamañoLista$ define el tamaño máximo para la lista restringida de candidatos. $Semilla$ es la semilla del generador de números 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
```

Algoritmo 3.3 Seudocódigo de algoritmo de búsqueda 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
```

3.1.2. Fase de Búsqueda Local

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

La efectividad de un algoritmo de búsqueda local depende de varios aspectos, como la estructura de vecindad, la estrategia de búsqueda dentro de dicha vecindad, la velocidad de la función de evaluación de costo y la solución inicial. La estrategia de búsqueda 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.

3.2. Path-Relinking

La metaheurística GRASP, tal como fue presentada en la sección anterior, es un procedimiento sin memoria. En cada iteración no se utiliza información obtenida en pasos previos. Una modificación posible sobre este aspecto es el uso de la técnica *Path-Relinking* para agregar memoria al procedimiento básico de GRASP, mejorando los tiempos de ejecución y la calidad de las soluciones [16].

Path-Relinking es una estrategia de intensificación que explora caminos que unen soluciones élite en el espacio de soluciones. Estos caminos se generan tomando una o más soluciones y aplicando movimientos dentro del espacio de búsqueda que agreguen atributos de otra solución guía.

El seudocódigo del Algoritmo 3.4 muestra el procedimiento de Path-Relinking aplicado a una solución de origen, S_s, y otra de destino, S_t. El procedimiento

Algoritmo 3.4 Seudocódigo del esquema general de la técnica de *Path-Relinking*.

```
procedure PathRelinking(S_s, S_t)
S̄ ← argmin {f(S_s), f(S_t)}
S ← S_s
while S △ S_t ≠ φ do
    m* ← argmin {f(S ⊕ m) : m ∈ S △ S_t}
    S ← S ⊕ m*
    if f(S) < f(S̄)
        S̄ ← S
    end
end
return S̄
```

comienza estableciendo la solución parcial con el argumento que minimice la función objetivo y luego comienza el recorrido por el espacio de soluciones en S_s, asignándolo a S como posición inicial. Luego, mientras existan diferencias entre la posición actual S y la solución destino S_t, calculado como la diferencia simétrica entre ambas soluciones, se obtiene aquella que minimice el costo de la función objetivo y se aplica a S, actualizando la mejor solución acumulada en caso de que el valor de la función objetivo de S sea mejor.

En el contexto de GRASP, Path-Relinking se aplica a pares de soluciones, siendo una de ellas un óptimo local obtenido al final de una iteración y la otra un miembro de un conjunto de soluciones élite seleccionado mediante algún criterio definido, como podría ser una selección aleatoria. El conjunto de soluciones élite está acotado en su tamaño y comienza siendo vacío. Una vez que este conjunto se llena, es necesario aplicar una política para reemplazar miembros existentes por nuevas soluciones generadas. Una opción es reemplazar el miembro de peor costo, si la solución local generada en el paso actual es mejor que este.

3.3. Heurística Greedy

La heurística Greedy es un algoritmo goloso que utilizamos para comparar el desempeño de la heurística CADILAC. Dicho algoritmo se basa en el algoritmo Remove-Find definido en Guo et al. [7], pero adaptado para hallar k caminos independientes, en lugar de solamente 2.

El Algoritmo 3.5 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 3.5 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
    // Se halla el camino más corto, SP,
    // entre Inicio y Fin en el grafo G.
    // El algoritmo utilizado es el de Bellman-Ford
    // limitando el largo de los caminos a d aristas.
    SP ← BellmanFord(G, Inicio, Fin, d)
    // Se eliminan los nodos del camino SP del
    // grafo G para poder realizar la búsqueda del
    // siguiente camino y se acumula SP en la Solución.
    G ← G \ SP
    Solución ← Solución ∪ SP
    k ← k − 1
end
return Solución
```

algoritmo.

3.4. 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 número 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 búsqueda local. A continuación describimos el diseño 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 búsqueda de CAminos DIsjuntos de Largo ACotado.

3.4.1. Fase de Construcción

La fase de construcción de soluciones factibles se basa en el algoritmo presentado por Bhandari en [4] (ver seudocódigo en el Algoritmo 2.2 del Capítulo 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 [7]. 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 calculo. 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 búsqueda del camino mas corto, la restricciones se multiplican por 2. En otras palabras, al realizar la búsqueda 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 búsqueda, 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 último camino encontrado, menos las aristas superpuestas con la solución parcial, y se realiza la búsqueda de uno nuevo. En nuestro algoritmo, la restricción sobre la cantidad de aristas por camino, d, se relaja al momento de realizar la búsqueda del camino más corto utilizando el parámetro `max-length-multiplier` (ver sección 4.3.3 por una lista completa de parámetros), 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 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 búsqueda 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 búsqueda de un nuevo camino. El algoritmo de Bhandari puede utilizar el valor opuesto debido a que,

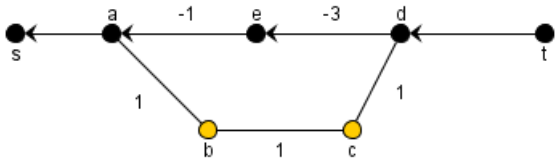


Figura 3.1: Si el algoritmo de Bhandari halló el camino coloreado de negro entre s y t y al invertir las aristas de dicho camino y asignar costo negativo a sus aristas se produjese un ciclo de costo negativo $C = a, b, c, d, e, a$, esto significa que el camino hallado por el algoritmo de Dijkstra en 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.

por más que el grafo resultante contenga aristas de costo negativo, no contendrá ciclos de costo negativo, lo que perjudicaría al algoritmo de búsqueda 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 3.1 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 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 $\vec{c}(e) = (0, 0), \forall e \in \textit{Solución}$, 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

Algoritmo 3.6 Seudocódigo del algoritmo de Dijkstra aleatorizado.

```
procedure DijkstraAleatrizado(G, s, t, d, p)
// El mapa Camino asocia un camino en
// G entre s y t para cada nodo de G.
Camino[v] ← φ, ∀v ∈ G
// La cola de prioridad contiene caminos
// con origen en s ordenados por costo.
Cola ← Insertar(Cola, Camino[s])
while not Vacía(Cola)
    C ← ExtraerCaminoDeCostoMínimo(Cola)
    u ← Destino(C)
    if u = t
        return C
    else
        // Se recorren los vecinos, con
        // probabilidad p.
        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
```

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 3.6 contiene el seudocódigo del algoritmo de Dijkstra aleatorizado.

El algoritmo 3.7 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 metaheurística GRASP, no se genera una lista de candidatos explícita de donde se toman elementos para generar incrementalmente una

Algoritmo 3.7 Seudocódigo del algoritmo goloso aleatorio utilizado para generar soluciones factibles para la heurística.

procedure $AlgoritmoGolosoAleatorio(G, c, k, MaxIntentos)$
$Solución \leftarrow \phi$
$Intentos \leftarrow MaxIntentos$
while $k > 0$ **and** $Intentos > 0$

`// Se crea un grafo` G' `como en Bhandari ...`
$G' \leftarrow Grafo()$
$V(G') \leftarrow V(G)$
$E(G') \leftarrow E(G) \setminus Solución \cup \{(u, v) : (v, u) \in Solución\}$
`// ... con la diferencia de que el costo de`
`// las aristas que pertenecen a` $Solución$ `es` $(0, 0)$.

$$\vec{c}' : E \to \mathbb{R}^+ \times \mathbb{R}^+, \vec{c}'(e) = \begin{cases} \vec{c}(e) & \forall e \notin Solución \\ (0,0) & \forall e \in Solución \end{cases}$$

`// En lugar de hallar el camino más corto entre`
`// Inicio y Fin, se halla un camino aleatorio.`
$RP' \leftarrow CaminoAleatorio(Inicio, Fin, G', p)$
$Solución' \leftarrow (Solución \cup SP') \setminus (Solución \cap SP')$
`// Por último, se verifica que la solución parcial`
`// cumpla con las restricciones del problema.`
if $EsFactible(Solución')$
$Solución \leftarrow Solución'$
$Intentos \leftarrow MaxIntentos$
$k \leftarrow k - 1$
else
$Intentos \leftarrow Intentos - 1$
end
end
if $k > 0$
return $Solución$
else
return $Error('No\ se\ pudo\ hallar\ una\ solución\ factible')$
end

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.

3.4.2. Fase de Búsqueda Local

La fase de búsqueda 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.

3.4.2.1. Estructura de Vecindad

La estructura de vecindad $N : \mathbb{S} \times \mathbb{N}^2 \to 2^{\mathbb{S}}$, 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 según 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. En el ejecutable implementado, dichos parámetros se denominan `max-subpath-length` y `max-subpath-replacement-length`, respectivamente. Ver la sección 4.3.3 por la lista completa de parámetros.

Las figuras 3.2, 3.3, 3.4, 3.5 y 3.6 muestran 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 la estructura de vecindad que transforman una solución en la otra. En nuestro caso, la estructura de vecindad que definimos para el algoritmo de búsqueda local *no* es transitiva, pues existen casos en los que no es posible transformar una solución en otra. La Figura 3.7 muestra un ejemplo de uno de estos casos. En dicha figura puede observarse las soluciones para el problema de hallar 2 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 vació. 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.

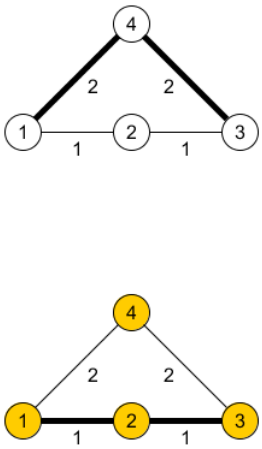


Figura 3.2: Este movimiento cambia el subcamino (1,4,3) por el (1,2,3), lo que en este caso es simplemente la sustitución de un nodo del camino por otro.

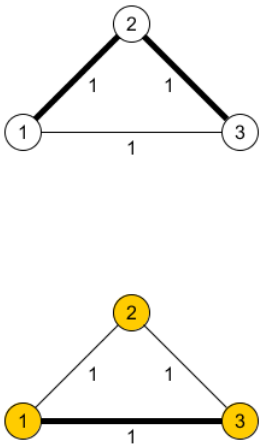


Figura 3.3: En este movimiento se sustituye el subcamino (1,2,3) por el (1,2), efectivamente eliminando un nodo del camino.

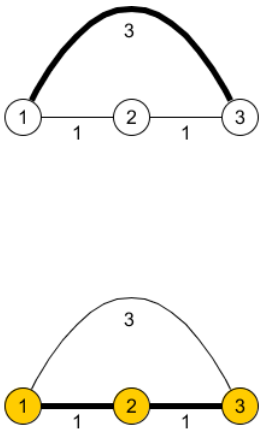


Figura 3.4: Este movimiento sustituye el subcamino (1,3) por el (1,2,3), insertando el nodo 2 en el camino.

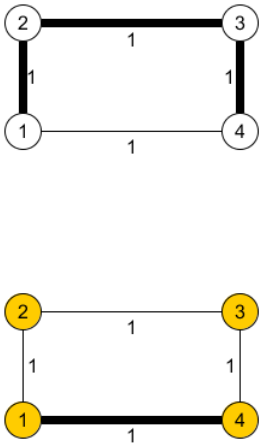


Figura 3.5: En este movimiento se sustituye el subcamino (1,2,3,4) por el (1,4), eliminando un subcamino completo.

3.4.2.2. Algoritmo de Búsqueda

El algoritmo de búsqueda 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 aplicara 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

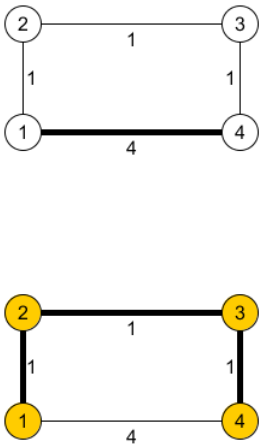


Figura 3.6: En este movimiento se sustituye el subcamino (1,4) por el (1,2,3,4), agregando un subcamino entre dos nodos adyacentes en el camino original.

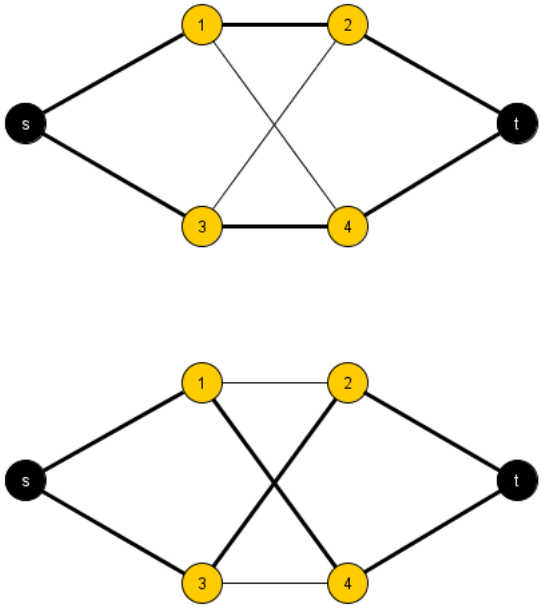


Figura 3.7: Dos soluciones para el problema de hallar 2 caminos nodo-disjuntos en un mismo grafo.

Algoritmo 3.8 Estructura general de un algoritmo de búsqueda local para la estructura de vecindad N.

```
procedure EstructuraBúsquedaLocal(G, S, n, m)
while not CriterioFinalizacion()
    P ← SeleccionarCamino(S)
    P' ← AplicarMovimiento(P, n, m)
    S ← (S \ P) ∪ P'
end
return S
```

proceso se puede repetir hasta que sea necesario. El seudocódigo 3.8 describe este algoritmo general para recorrer la estructura de vecindad.

Nuestro algoritmo de búsqueda 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 búsqueda. El seudocódigo 3.9 describe el algoritmo de búsqueda 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.

3.4.2.3. Algoritmo de Selección del Mejor Movimiento

El algoritmo que selecciona el movimiento a realizar sobre un camino elige el mejor posible según 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 3.4.2.1 sobre la estructura de vecindad). El algoritmo 3.10 muestra el seudocódigo para el proceso descrito.

Para el algoritmo de búsqueda 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

Algoritmo 3.9 Algoritmo de búsqueda local utilizado para optimizar el resultado devuelto por la fases de construcción de la heurística CADILAC.

```
procedure BúsquedaLocal(G, k, S, n, m)
// La lista de posiciones se utiliza
// para almacenar índices identificadores
// de los caminos de la solución S.
// Almacena valores de 1 a k sin repetirse.
Posiciones ← [1..k]
repeat
    // Esta variable indica si durante la
    // iteración actual se logró realizar
    // alguna mejora con alguno de los
    // movimientos.
    HuboMejoras ← false
    // Se permutan las posiciones de forma
    // aleatoria para poder aplicar los
    // movimientos a todos los caminos.
    Posiciones ← PermutarAleatoriamente(Posiciones)
    // Se iteran los caminos según el orden
    // definido por la permutación.
    for each i in Posiciones
        P ← ObtenerCamino(S, i)
        // Se aplica el mejor movimiento posible
        // para el camino P, según las restricciones
        // impuestas por los parámetros n y m.
        // El camino resultante preserva la
        // factibilidad de la solución.
        P' ← AplicarMejorMovimiento(P, n, m)
        if Costo(P') < Costo(P)

            // Si se halló un mejor camino como
            // resultado de aplicar un movimiento,
            // se agrega a la solución.
            S ← (S \ P) ∪ P'
            HuboMejoras ← true
        end
    end
until not HuboMejoras
return S
```

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 búsqueda de caminos que esté más adaptado a la realidad de nuestro problema, por lo que decidimos diseñar una solución alternativa. De esta forma llegamos al algoritmo de búsqueda 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 pequeños para los tamaños máximos, la velocidad de ejecución de la heurística CADILAC es menor que con las opciones originales, y los resultados obtenidos por el algoritmo de búsqueda local no son significativamente peores. El seudocódigo puede verse en el Algoritmo 3.11.

Algoritmo 3.10 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
        // Se calcula el costo y largo máximo
        // que puede tener el reemplazo del subcamino
        // que existe entre u y v.
        CostMáx ← Costo + MejorIncremento
        LargoDisponible ← d − (Largo(P) − Largo)
        LargoMáx ← min(LargoDisponible, m)
        // Se calcula el camino más corto entre
        // S y T, cuyo costo sea menor a CostoMáx
        // y su largo sea menor o igual a LargoMáx
        R ←
        CaminoMásCorto(S, T, CostoMáx, LargoMáx)
        if Existe(R)
            MejorReemplazo ← R
            MejorIncremento ← Costo(R) − Costo
        end
        v ← SiguienteArco(v, P)
    until Largo = n and Existe(v)

end
if Existe(MejorReemplazo)

    return Reemplazar(P, MejorReemplazo)

end
```

Algoritmo 3.11 Algoritmo de búsqueda 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 es el camino que sale de origen.
// TP es el camino que sale desde el destino.
// Se toman el destino u del camino de origen
// y el origen v del camino de destino.
u ← Destino(OP)
v ← Origen(TP)
// Se calculan los costos y largos parciales
// combinando los de ambos caminos.
Costo' ← Costo(OP) + Costo(TP)
Largo' ← Largo(OP) + Largo(TP)
// Si ambos caminos se encontraron y su costo
// es mejor que el del camino mas corto actual
// (inicialmente Costo(SP) = CostoMax), entonces
// se establece su concatenación como el mejor
// camino actual.
if u = v and Costo' < Costo(SP)
    SP ← OP + TP
else
    // Si su largo combinado no supera el largo
    // máximo, ambos nodos están conectados y
    // el costo combinado mas el costo de la arista
    // que los conecta es mejor que el costo del
    // mejor camino actual, entonces se establece
    // como mejor camino actual la concatenación
    // de los 3 elementos.
    if Largo' < LargoMax
    and ∃(u, v) ∈ E(G)
    and Costo' + Costo(u, v) < Costo(SP)
        SP ← OP + (u, v) + TP
    end
    // Si se puede seguir agregando aristas,
    // continuar recursivamente.
    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
```

Capítulo 4

Implementación del Algoritmo

4.1. Herramientas de Desarrollo

Para realizar este trabajo construimos tres programas. El primero implementa el algoritmo propuesto. El segundo implementa la heurística Greedy. El tercer algoritmo es el algoritmo de Suurballe, que es utilizado para poder hallar en la sección de resultados la solución al problema cuando se elimina la restricciones sobre el largo de los caminos.

4.1.1. 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 soluciones a problemas generales de implementación como la manipulación de entrada y salida, definición de estructuras de datos, etc... de forma rápida. Estas dos características nos permitieron, una vez que elegimos el resto de las herramientas a utilizar, concentrarnos en la implementación eficiente de nuestro algoritmo sin preocuparnos por estos detalles.

La variante que utilizamos del lenguaje C++ es la que se conoce como C++11, la última versión aprobada por ISO en Agosto de 2011. Esta versión incluye extensiones como expresiones lambda, inferencia de tipos básica, metaprogramación con *templates*, etc... que simplifican el desarrollo de *templates* haciendo que la sintaxis sea menos engorrosa, lo que nos resultó útil en la implementación del algoritmo.

4.1.2. 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*) [18], 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*) [19].

TCLAP es una pequeña 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 disponibles, las que más nos interesaron fueron LEMON y BGL. Nos decidimos por LEMON porque parecía tener una API más intuitiva, aparte de poseer una implementación del algoritmo de Suurballe que, a priori, podría haber resultado útil. Efectivamente, LEMON resultó ser, una vez que aprendimos sus idiomas particulares, una biblioteca que facilita mucho la manipulación de grafos y posee una batería de algoritmos muy completa. La implementación del algoritmo de Suurballe incluida en ella nos permitió realizar pruebas sobre los datos que utilizamos en la sección de resultados.

4.2. Estructura

Los archivos que contienen las rutinas principales de los programas se encuentran en el directorio *main* mientras que los archivos que implementan las clases y subrutinas que resuelven el problema se hallan en directorio *src* del archivo comprimido que contiene el trabajo completo. A continuación se detalla la lista de archivos que compone a cada programa.

4.2.1. Heurística CADILAC

heuristica-grasp.cpp Contiene la rutina principal del programa donde se invocan al resto de las subrutinas.

GraspHeuristicAlgorithm.h Implementa el esquema de la heurística CADILAC.

BhandariDijkstraAleatorizado.cpp Contiene la implementación del algoritmo de Bhandari modificado que a su vez utiliza el algoritmo de Dijkstra

Aleatorizado. Se corresponde con el algoritmo de generación de soluciones factibles.

BhandariDijkstraAleatorizado.h Cabezal del archivo BhandariDijkstraAleatorizado.cpp.

RandomizedDjikstra.h Implementación del algoritmo de Dijkstra aleatorizado.

SolutionOptimizer.h Implementación del algoritmo de búsqueda local de la heurística CADILAC.

Movements.h Implementación del algoritmo de búsqueda del mejor movimiento posible para un camino de una solución factible.

BfsShortestPath.h Implementación del algoritmo de búsqueda del camino más corto entre 2 nodos de un grafo mediante la ejecución de un algoritmo DFS directo e inverso de forma simultánea.

BoundedPathCost.h Estructura de datos que representa el costo compuesto de una arista de un grafo.

Helpers.h Implementación de rutinas auxiliares para el programa.

4.2.2. Heurística Greedy

heuristica-simple.cpp Rutina principal del programa.

NaiveHeuristicAlgorithm.cpp Implementación el algoritmo Greedy.

NaiveHeuristicAlgrorithm.h Cabezal del archivo NaiveHeuristicAlgorithm.cpp.

LengthLimitedBellmanFord.h Adaptador para la implementación provista por LEMON del algoritmo de Bellman-Ford. Este adaptador ejecuta el algoritmo de Bellman-Ford limitando el largo máximo de la solución hallada.

Helpers.h Implementación de rutinas auxiliares para el programa.

4.2.3. Algoritmo de Suurballe

suurballe.cpp Rutina principal del programa.

NodeDisjointSuurballe.h Adaptador para el algoritmo de Suurballe provisto por la biblioteca LEMON. La versión del algoritmo de Suurballe provista por defecto considera el caso del problema en el que se desean que los caminos sean arista-disjuntos. Este adaptador modifica el algoritmo para que considere el caso donde los caminos deben ser nodo-disjuntos.

Helpers.h Implementación de rutinas auxiliares para el programa.

4.3. Compilación y Ejecución

4.3.1. Requisitos Previos

Antes de compilar el código fuente es necesario descargar la versión 1.2.1 de la biblioteca TCLAP [18] y la versión 1.3 de la biblioteca LEMON [19]. La biblioteca TCLAP esta compuesta exclusivamente por archivos cabezales (*.h), por lo que no requiere compilación, en cambio LEMON si debe compilarse antes de ser utilizado. LEMON utiliza el sistema CMake para configurar y compilar su código por lo que recomendamos al instalación de dicho sistema antes de proceder.

4.3.2. Compilación del Código Fuente

El código fuente del programa se encuentra organizado dentro de un proyecto del IDE Code::Blocks, por lo que se recomienda su utilización para la compilación de los ejecutables. Los proyectos que contienen los programas implementados se llaman heuristica-grasp.cbp, heuristica-simple.cbp y suurballe.cbp que implementan los algoritmos CADILAC, Greedy y de Suurballe, respectivamente. Para que la compilación sea exitosa, es necesario configurar los directorios donde se encuentran los cabezales de TCLAP y LEMON, y el directorio donde se encuentra la biblioteca compilada de LEMON.

En caso de no disponer del IDE, los fuentes se pueden compilar de la forma tradicional, considerando que se debe definir la macro NDEBUG al momento de hacerlo. Por ejemplo:

```
gcc -std=c++11 -O3 -DNDEBUG -Isrc -I<path_to_lemon_include_dir>
-I<path_to_tclap_include_dir> -L<path_to_lemon_lib_dir> -llemon -o
heuristica-grasp main/heuristica-grasp.cpp src/*.cpp
```

4.3.3. Ejecución de los Programas

Los programas se ejecutan en la línea de comandos. Todos sus parámetros son especificados en el comando de invocación, no hay archivos de configuración. La salida generada por los programas es enviada a la salida estándar y consiste de 3 secciones que muestran los parámetros especificados, un resumen del resultado obtenido y la solución detallada. La Figura 4.1 muestra un ejemplo de salida para el programa que ejecuta el algoritmo de Suurballe.

```
Parametros: -f ..\dimacs\ch9-1.1\inputs\Random4-n\
   Random4-n.10.0.gr -s 1 -t 1000 -k 4
0,8565,0
{
    #: 0,
    solution: {
        {
            cost: 2175,
            length: 6,
            edges: [(1,193),(193,61),(61,939),(939,11)
                ,(11,291),(291,1000)]
        },
        {
            cost: 1837,
            length: 6,
            edges: [(1,997),(997,138),(138,459)
                ,(459,507),(507,416),(416,1000)]
        },
        {
            cost: 2247,
            length: 7,
            edges: [(1,664),(664,145),(145,123)
                ,(123,481),(481,21),(21,554)
                ,(554,1000)]
        },
        {
            cost: 2306,
            length: 10,
            edges: [(1,787),(787,27),(27,759)
                ,(759,192),(192,423),(423,1006)
                ,(1006,331),(331,24),(24,72),(72
                ,1000)]
        },
    },
    cost: 8565
},
```

Figura 4.1: Salida de ejemplo de los programas. Contiene 3 secciones. La primera, de una línea contiene los parámetros especificados en la línea de comandos. La segunda muestra en formato CSV el número de ejecución, el costo total de la solución encontrada y el tiempo de ejecución del algoritmo (no incluye tiempo de lectura del archivo del grafo ni la impresión de la salida. La tercer sección muestra en detalle la solución encontrada en formato JSON.

Los parámetros de cada uno de los ejecutables pueden hallarse con el parámetro -h en la línea de comandos. A continuación los listamos en su totalidad y describimos el propósito de cada uno:

-s, --source Nodo de origen de los caminos a hallar.

-t, --target Nodo de destino de los caminos a hallar.

-f, --file Archivo de entrada que contiene el grafo donde se ejecutará el algoritmo. Debe estar en formato DIMACS y *no* debe poseer aristas repetidas. En caso de que contenga aristas repetidas se puede utilizar el programa dimacs-dedup incluido con el código fuente de los programas. Para compilar dimacs-dedup se deben seguir las mismas instrucciones que fueron indicadas para los ejecutables principales.

-l, --max-length Largo máximo permitido para los caminos de la solución.

-k, --path-number Cantidad de caminos que debe contener la solución.

-n, --replication-number Cantidad de veces que se ejecuta el algoritmo.

-p, --probability Probabilidad de visitar un vecino al momento de buscar un camino en el algoritmo de Dijkstra aleatorizado.

--grasp-n Cantidad de repeticiones realizadas por el algoritmo GRASP.

--max-subpath-length Largo máximo del subcamino que se puede optimizar en la búsqueda local.

--max-subpath-replacement-length Largo máximo del camino que se usa como reemplazo en la búsqueda local.

--dijkstra-max-n Número máximo de repeticiones del algoritmo de Dijkstra dentro de una ejecución del algoritmo de construcción.

--max-length-multiplier Multiplicador del largo máximo para los caminos hallados por el algoritmo de Dijkstra aleatorizado.

Capítulo 5

Resultados

5.1. Pruebas

Las pruebas consisten en la ejecución de los tres algoritmos: la heurística CADILAC, la heurística Greedy y el algoritmo de Suurballe 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 [11]. Antes de poder utilizar los archivos de los grafos, los filtramos con el programa dimacs-dedup, provisto con el código fuente de los programas, para eliminar aristas duplicadas, puesto que no están soportadas por los algoritmos (ver parámetro --file en la subsección 4.3.3 sobre la ejecución de los programas).

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 5.1, 5.2 y 5.3 muestran los resultados obtenidos en las pruebas realizadas según se describió en la sección anterior sobre Juego de Datos. La última 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 señalados con un '+'. Los casos inversos están señalados con un '-'. Los tiempos de ejecución están reportados en segundos.

En los cuadros correspondientes a las pruebas de los dos grafos aleatorios, puede observarse que la heurística CADILAC supera a la Greedy en el costo de las soluciones halladas. En los casos donde ambas heurísticas hallan soluciones y la heurística CADILAC logra un mejor costo, dicha mejora es en promedio de 3.42 % en Random4-n.10.0, 1.83 % en Random4-n.16.0 y 2.75 % en USA-road-d.NY. Los tiempos de ejecución sin embargo son sustancialmente peores, como era de esperar. El tiempo de ejecución de la heurística CADILAC es en promedio 1919, 480 y 38 veces mayor que el de la heurística Greedy en los grafos Random4-n.10.0, Random4-n.16.0 y USA-road-d.NY, respectivamente.

En las tablas de datos correspondiente al grafo USA-road-d.NY, puede notarse un desmejoramiento en el rendimiento de la heurística CADILAC. Allí puede observarse que, en varios casos, el resultado obtenido con la heurística CADILAC, realizando 800 iteraciones, no logró mejorar el resultado obtenido por la heurística Greedy; incluso existen casos donde la heurística Greedy halla soluciones y la heurística CADILAC no lo logra.

Entrada				Suurballe			Heurística Greedy			Heurística CADILAC			Mejora
s	t	k	d	costo	mayor largo	tiempo	costo	mayor largo	tiempo	costo	mayor largo	tiempo	
63	427	5	11	11107	11	0.00606	11609	10	0.00276	11107	11	1.76269	4.32 %
63	427	5	9	-	-	-	12150	9	0.00245	11544	9	1.77847	4.99 %
63	427	5	7	-	-	-	11663	7	0.00108	11663	7	1.49335	0.00 %
63	427	5	6	-	-	-	-	-	-	13595	6	0.83802	+
63	427	2	11	3754	9	0.00222	3754	9	0.00146	3754	9	0.58929	0.00 %
63	427	2	9	-	-	-	3754	9	0.00039	3754	9	0.60367	0.00 %
63	427	2	7	-	-	-	4158	6	0.00055	4158	6	0.57471	0.00 %
63	427	2	6	-	-	-	4158	6	0.00026	4158	6	0.44590	0.00 %
21	412	5	12	11679	12	0.00453	11748	12	0.00363	11679	12	1.92767	0.59 %
21	412	5	10	-	-	-	11828	9	0.00107	11727	10	3.87256	0.85 %
21	412	5	8	-	-	-	12245	8	0.00272	12079	8	3.70956	1.36 %
21	412	5	7	-	-	-	12146	7	0.00068	12146	7	1.88642	0.00 %
21	412	2	12	3854	8	0.00083	3854	8	0.00103	3854	8	1.50858	0.00 %
21	412	2	7	-	-	-	4114	6	0.00033	4114	6	1.60520	0.00 %
21	412	2	5	-	-	-	4528	5	0.00015	4528	5	0.51553	0.00 %
646	268	5	12	13442	12	0.00441	13485	11	0.00118	13442	12	3.05050	0.32 %
646	268	5	11	-	-	-	13485	11	0.00106	13453	8	2.77576	0.24 %
646	268	5	7	-	-	-	14020	7	0.00063	14020	7	0.91301	0.00 %
646	268	2	12	3187	5	0.00042	3187	5	0.00157	3187	5	0.63204	0.00 %
646	268	2	5	-	-	-	3187	5	0.00039	3187	5	0.14404	0.00 %
575	242	5	12	11511	10	0.00375	12128	8	0.00366	11511	10	3.99109	5.09 %
575	242	5	9	-	-	-	12128	8	0.00092	11783	7	2.91597	2.84 %
575	242	5	7	-	-	-	11783	7	0.00224	11783	7	1.23807	0.00 %
575	242	2	12	4110	8	0.00046	4110	8	0.00148	4110	8	2.18925	0.00 %
575	242	2	7	-	-	-	4209	6	0.00033	4209	6	1.17831	0.00 %
575	242	2	5	-	-	-	4484	4	0.00054	4484	4	0.27507	0.00 %
528	98	5	12	14096	9	0.00575	14804	9	0.00099	14096	9	4.74253	4.78 %
528	98	5	8	-	-	-	15010	8	0.00082	14484	8	1.83381	3.50 %
528	98	5	7	-	-	-	-	-	-	16944	7	1.29918	+
528	98	2	12	4506	9	0.00064	4554	9	0.00143	4506	9	2.56661	1.05 %
528	98	2	8	-	-	-	4666	8	0.00052	4638	8	1.36790	0.60 %
528	98	2	7	-	-	-	5534	7	0.00095	5330	7	1.73124	3.69 %
528	98	2	6	-	-	-	6023	6	0.00021	6023	6	1.12245	0.00 %
66	224	5	12	10943	12	0.00155	10943	12	0.00359	10943	12	4.36861	0.00 %
66	224	5	11	-	-	-	11070	9	0.00346	11070	9	4.20029	0.00 %
66	224	5	8	-	-	-	11390	7	0.00276	11390	7	2.40210	0.00 %
66	224	2	12	3487	6	0.00153	3487	6	0.00143	3487	6	1.93185	0.00 %
929	848	5	12	10118	12	0.00119	10118	12	0.00154	10118	12	4.00224	0.00 %
929	848	5	11	-	-	-	10492	9	0.00100	10492	9	4.18849	0.00 %
929	848	5	8	-	-	-	10549	7	0.00304	10549	7	2.44468	0.00 %
929	848	2	12	2877	5	0.00123	2877	5	0.00137	2877	5	1.39434	0.00 %
336	369	5	12	15658	13	0.00197	16222	12	0.00346	15663	10	5.66477	3.45 %
336	369	5	9	-	-	-	16313	9	0.00295	15751	9	4.33316	3.45 %
336	369	5	8	-	-	-	17521	8	0.00120	16454	8	2.82868	6.09 %
336	369	2	12	5095	9	0.00235	5464	12	0.00153	5095	9	2.79523	6.75 %
336	369	2	8	-	-	-	5533	8	0.00114	5203	6	2.42782	5.96 %
258	812	5	12	10587	9	0.00395	11225	9	0.00343	10587	9	3.75434	5.68 %
258	812	5	8	-	-	-	12200	8	0.00092	11201	8	2.30213	8.19 %
258	812	5	7	-	-	-	12421	7	0.00072	12248	7	1.75508	1.39 %
258	812	5	6	-	-	-	-	-	-	14367	6	0.69371	+
258	812	2	12	3018	4	0.00088	3018	4	0.00047	3018	4	0.65782	0.00 %

Cuadro 5.1: Resultados numéricos para el grafo Random4-n.10.0

Entrada				Suurballe			Heurística Greedy			Heurística CADILAC			Mejora
s	t	k	d	costo	mayor largo	tiempo	costo	mayor largo	tiempo	costo	mayor largo	tiempo	
38760	316	5	13	1027311	13	0.23026	1027311	13	0.26885	1027311	13	77.47830	0.00 %
38760	316	5	11	-	-	-	1033594	11	0.23908	1033594	11	69.77460	0.00 %
38760	316	5	10	-	-	-	1061631	10	0.15771	1061631	10	94.91310	0.00 %
38760	316	5	9	-	-	-	-	-	-	1103528	9	74.92600	+
38760	316	2	13	330609	13	0.03330	330609	13	0.11425	330609	13	11.19300	0.00 %
38760	316	2	12	-	-	-	336892	10	0.09596	336892	10	17.22730	0.00 %
38760	316	2	9	-	-	-	345821	9	0.04838	345821	9	11.76780	0.00 %
38760	316	2	8	-	-	-	399281	7	0.02741	399281	7	12.66000	0.00 %
16599	19775	5	18	941330	17	0.12944	941330	17	0.31852	941330	17	48.59300	0.00 %
16599	19775	5	16	-	-	-	949322	14	0.30281	949322	14	51.76650	0.00 %
16599	19775	5	13	-	-	-	949813	12	0.26557	949813	12	60.77510	0.00 %
16599	19775	5	11	-	-	-	1014540	11	0.18445	1011610	11	65.56950	0.29 %
16599	19775	5	10	-	-	-	1110161	10	0.13818	1104406	10	52.68120	0.52 %
16599	19775	2	12	321404	12	0.04430	321404	12	0.09311	321404	12	14.31460	0.00 %
16599	19775	2	11	-	-	-	325234	11	0.09766	325234	11	18.07540	0.00 %
16599	19775	2	10	-	-	-	347136	10	0.06959	345232	9	21.10580	0.55 %
16599	19775	2	8	-	-	-	360164	8	0.02072	360164	8	15.78120	0.00 %
31647	33248	5	16	943980	16	0.29208	959842	13	0.36526	943980	16	116.42000	1.65 %
31647	33248	5	15	-	-	-	959842	13	0.33348	953010	11	121.11900	0.71 %
31647	33248	5	10	-	-	-	993667	10	0.20656	993667	10	112.90100	0.00 %
31647	33248	5	9	-	-	-	1289524	9	0.12841	1271838	9	124.33000	1.37 %
31647	33248	2	10	289318	10	0.04705	289318	10	0.08286	289318	10	15.34120	0.00 %
31647	33248	2	9	-	-	-	350443	9	0.06174	350443	9	29.89920	0.00 %
31647	33248	2	8	-	-	-	406284	8	0.03206	406284	8	28.07350	0.00 %
5608	1780	5	12	942854	12	0.28591	955101	12	0.24495	942854	12	89.71730	1.28 %
5608	1780	5	11	-	-	-	983231	10	0.24047	978849	10	80.50630	0.45 %
5608	1780	5	9	-	-	-	1082457	9	0.13477	1045872	8	86.91670	3.38 %
5608	1780	2	12	340202	12	0.07626	340202	12	0.10576	340202	12	25.35470	0.00 %
5608	1780	2	11	-	-	-	352709	9	0.10445	352709	9	42.24360	0.00 %
5608	1780	2	8	-	-	-	354919	8	0.03488	354919	8	17.52790	0.00 %
5608	1780	2	7	-	-	-	374574	6	0.01419	374574	6	9.32795	0.00 %
17465	63785	5	14	1176835	14	0.37397	1176835	14	0.29872	1176835	14	133.40300	0.00 %
17465	63785	5	13	-	-	-	1177575	13	0.26703	1177575	13	146.33500	0.00 %
17465	63785	5	12	-	-	-	1293743	11	0.22854	1232227	12	113.34800	4.75 %
17465	63785	5	11	-	-	-	1293743	11	0.18679	1293743	11	138.80500	0.00 %
17465	63785	5	10	-	-	-	-	-	-	1372362	10	103.58800	+
17465	63785	2	13	397073	13	0.09970	397073	13	0.10551	397073	13	45.02800	0.00 %
17465	63785	2	12	-	-	-	433628	11	0.10782	429424	12	66.82590	0.97 %
17465	63785	2	11	-	-	-	433628	11	0.09131	433628	11	58.47560	0.00 %
17465	63785	2	10	-	-	-	435065	10	0.06513	435065	10	64.92490	0.00 %
17465	63785	2	9	-	-	-	523033	9	0.03915	483347	9	45.85560	7.59 %
14006	4838	5	17	1180542	17	0.34548	1194852	16	0.30725	1180542	17	119.20400	1.20 %
14006	4838	5	16	-	-	-	1194852	16	0.29664	1180847	14	116.33200	1.17 %
14006	4838	5	13	-	-	-	1216725	12	0.29501	1196793	12	146.19700	1.64 %
14006	4838	5	11	-	-	-	1274916	11	0.20590	1274916	11	85.86410	0.00 %
14006	4838	5	10	-	-	-	1444970	10	0.15127	-	-	-	-
14006	4838	2	16	419363	16	0.11342	419363	16	0.13554	419363	16	38.87930	0.00 %
14006	4838	2	15	-	-	-	419583	12	0.12721	419583	12	47.81650	0.00 %
14006	4838	2	11	-	-	-	443709	11	0.09505	443709	11	47.10430	0.00 %
14006	4838	2	10	-	-	-	479448	10	0.06160	479448	10	53.53840	0.00 %
14006	4838	2	9	-	-	-	537819	9	0.04306	537819	9	64.62740	0.00 %

Cuadro 5.2: Resultados numéricos para el grafo Random4-n.16.0

Entrada				Suurballe			Heurística Greedy			Heurística CADILAC			Mejora
s	t	k	d	costo	mayor largo	tiempo	costo	mayor largo	tiempo	costo	mayor largo	tiempo	
131377	187202	3	500	850983	328	0.107883	851169	328	6.87614	850983	328	128.953	0.02 %
131377	187202	3	300	-	-	-	852049	288	2.96023	851863	288	137.983	0.02 %
131377	187202	3	280	-	-	-	852171	279	2.90346	852053	276	119.438	0.01 %
131377	187202	3	270	-	-	-	852486	270	2.52559	857921	269	112.696	-0.64 %
131377	187202	3	260	-	-	-	856669	260	2.36755	-	-	-	-
131377	187202	2	500	502912	267	0.0663311	503098	268	4.43448	502912	267	63.692	0.04 %
131377	187202	2	250	-	-	-	504694	250	1.55615	504605	249	77.2971	0.02 %
131377	187202	2	240	-	-	-	506891	240	1.43853	504850	202	73.63	0.40 %
177014	166800	3	700	556164	170	0.0325777	578423	167	11.7055	556414	172	52.9875	3.81 %
177014	166800	3	170	-	-	-	578423	167	0.836909	556164	170	44.4514	3.85 %
177014	166800	3	150	-	-	-	608664	150	0.708009	-	-	-	-
177014	166800	2	700	363619	170	0.0242099	365691	167	7.76897	363644	168	28.3297	0.56 %
177014	166800	2	170	-	-	-	365691	167	0.579829	363619	170	28.8091	0.57 %
177014	166800	2	150	-	-	-	370374	150	0.538741	370374	150	17.1552	0.00 %
204401	218166	3	700	1455789	566	0.246956	-	-	-	1465782	562	356.745	+
204401	218166	3	500	-	-	-	-	-	-	1474279	499	266.321	+
204401	218166	2	700	866113	533	0.12388	998086	467	6.07711	866900	522	210.248	13.14 %
204401	218166	2	500	-	-	-	998086	467	5.07668	871412	491	214.422	12.69 %
204401	218166	2	400	-	-	-	1007572	400	3.95778	923135	399	171.515	8.38 %
14624	9495	3	700	317548	105	0.0237651	340128	102	8.89344	317548	105	30.4993	6.64 %
14624	9495	3	100	-	-	-	340804	100	0.473076	322336	99	23.843	5.42 %
14624	9495	2	200	202516	97	0.0173091	209003	102	0.848514	202516	97	15.6269	3.10 %
14624	9495	2	90	-	-	-	-	-	-	211255	88	11.8201	+
186986	73410	3	700	1652094	531	0.316427	1692743	530	9.00988	1654429	526	353.493	2.26 %
186986	73410	3	500	-	-	-	1694659	500	7.34241	1690604	499	285.41	0.24 %
186986	73410	3	400	-	-	-	1775205	400	6.62669	-	-	-	-
186986	73410	2	700	1046995	530	0.181153	1046995	530	5.54406	1046995	530	203.135	0.00 %
186986	73410	2	500	-	-	-	1048911	500	5.19312	1050219	494	236.216	-0.12 %
186986	73410	2	400	-	-	-	1071151	400	4.57502	1074232	400	218.123	-0.29 %
186986	73410	2	350	-	-	-	1120328	350	3.95453	-	-	-	-
126313	72919	3	700	613444	210	0.0520591	618638	204	8.40751	613743	210	83.3901	0.79 %
126313	72919	3	200	-	-	-	619073	200	1.25659	617734	196	64.7395	0.22 %
126313	72919	3	190	-	-	-	622860	190	1.0967	623619	190	52.9849	-0.12 %
126313	72919	3	180	-	-	-	643567	180	1.15538	-	-	-	-
126313	72919	2	500	388882	206	0.0429569	396634	204	4.80258	388882	206	58.8643	1.95 %
126313	72919	2	200	-	-	-	397069	200	0.945506	391777	195	56.1769	1.33 %
126313	72919	2	190	-	-	-	399990	190	0.839708	401861	190	51.3259	-0.47 %
126313	72919	2	180	-	-	-	406105	180	0.766545	406780	180	43.6704	-0.17 %
126313	72919	2	170	-	-	-	413740	170	0.689639	-	-	-	-
123278	69036	3	700	785180	211	0.0711206	-	-	-	785180	211	88.0732	+
123278	69036	3	200	-	-	-	-	-	-	792884	198	52.4504	+
123278	69036	3	190	-	-	-	-	-	-	798502	189	41.9097	+
123278	69036	2	200	509665	211	0.0415987	513519	200	0.752445	515918	197	48.4	-0.47 %
123278	69036	2	190	-	-	-	518979	190	0.709457	520189	190	37.6299	-0.23 %
123278	69036	2	180	-	-	-	530849	180	0.706261	-	-	-	-
40616	113310	3	700	398415	152	0.0352233	404377	143	9.41533	398793	151	47.2207	1.38 %
40616	113310	3	150	-	-	-	404377	143	0.770563	399206	148	43.6251	1.28 %
40616	113310	3	140	-	-	-	405871	140	0.749444	402880	140	34.9936	0.74 %
40616	113310	3	130	-	-	-	411028	130	0.641138	419217	129	23.2345	-1.99 %
40616	113310	3	120	-	-	-	433634	120	0.530938	-	-	-	-

Cuadro 5.3: Resultados numéricos para el grafo USA-road-d.NY

Capítulo 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 aún 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 número de iteraciones, la probabilidad de visita a nodos vecinos en el algoritmo de búsqueda de caminos, el multiplicador del largo máximo de los caminos, también para la búsqueda de caminos, y los largos máximos utilizados en la sustitución de subcaminos en el algoritmo de búsqueda 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. También podría estudiarse, para los casos en el que la heurística Greedy no halla la solución óptima, la relación entre los costos de las soluciones de ambas heurísticas y analizar cuán alejados están unos de otros.

Al momento de implementar el algoritmo propuesto para tratar el problema presentado en este informe de forma práctica, se debería incluir a los otros dos algoritmos con los cuales fue comparado: el de Suurballe y 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 retornada por 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 último, 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 adicionar la técnica de Path-Relinking discutida en la sección 3.2, logrando las ventajas allí mencionadas. Una forma sencilla de implementar esta técnica podría ser mediante el intercambio de caminos no superpuestos entre las soluciones élite y la solución local hallada en cada iteración. La implementación de ésta técnica quedó por fuera del trabajo dado que, al momento de diseñar el algoritmo concreto a utilizar, concluimos que extendería la duración del proyecto más allá de lo previsto.

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 2, 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.

Bibliografía

[1] R. G. Ogier, V. Rutenburg, and N. Shacham, "Distributed algorithms for computing shortest pairs of disjoint paths," *Information Theory, IEEE Transactions on*, vol. 39, no. 2, pp. 443–455, 1993.

[2] A. Srinivas and E. Modiano, "Finding minimum energy disjoint paths in wireless ad-hoc networks," *Wireless Networks*, vol. 11, no. 4, pp. 401–417, 2005.

[3] A. Aggarwal, J. Kleinberg, and D. P. Williamson, "Node-disjoint paths on the mesh and a new trade-off in vlsi layout," *SIAM Journal on Computing*, vol. 29, no. 4, pp. 1321–1333, 2000.

[4] R. Bhandari, *Survivable networks: algorithms for diverse routing.* Springer, 1999.

[5] J. W. Suurballe and R. E. Tarjan, "A quick method for finding shortest pairs of disjoint paths," *Networks*, vol. 14, no. 2, pp. 325–336, 1984.

[6] J. Suurballe, "Disjoint paths in a network," *Networks*, vol. 4, no. 2, pp. 125–145, 1974.

[7] Y. Guo, F. Kuipers, and P. Van Mieghem, "Link-disjoint paths for reliable QoS routing," *International Journal of Communication Systems*, vol. 16, no. 9, pp. 779–798, 2003.

[8] A. Itai, Y. Perl, and Y. Shiloach, "The complexity of finding maximum disjoint paths with length constraints," *Networks*, vol. 12, no. 3, pp. 277–286, 1982.

[9] A. Bley, *On the complexity of vertex-disjoint length-restricted path problems.* Citeseer, 1998.

[10] A. Beshir and F. Kuipers, "Variants of the min-sum link-disjoint paths problem," in *16th Annual Symposium on Communications and Vehicular Technology (SCVT)*, 2009.

[11] "9th DIMACS implementation challenge - shortest paths," http://www.dis.uniroma1.it/challenge9/index.shtml.

[12] K. Xiong, Z.-d. Qiu, Y. Guo, and H. Zhang, "Multi-constrained shortest disjoint paths for reliable QoS routing." *ETRI journal*, vol. 31, no. 5, pp. 534–544, 2009.

[13] Y. Kobayashi and C. Sommer, "On shortest disjoint paths in planar graphs," *Discrete Optimization*, vol. 7, no. 4, pp. 234–245, 2010.

[14] R. Fleischer, Q. Ge, J. Li, and H. Zhu, "Efficient algorithms for k-disjoint paths problems on DAGs," in *Algorithmic Aspects in Information and Management*. Springer, 2007, pp. 134–143.

[15] P. Van Mieghem, H. De Neve, and F. Kuipers, "Hop-by-hop quality of service routing," *Computer Networks*, vol. 37, no. 3, pp. 407–423, 2001.

[16] M. G. Resende and P. Pardalos, *Handbook of Optimization in Telecommunications*. Springer, 2008, pp. 103–128.

[17] C. Ribeiro and P. Hansen, *Essays and surveys in metaheuristics*. Kluwer Academic Publishers, 2002.

[18] "TCLAP : Templatized C++ command line parser library," http://tclap.sourceforge.net/.

[19] "LEMON : Library for efficient modeling and optimization in networks," http://lemon.cs.elte.hu/trac/lemon.

[20] H. De Neve and P. Van Mieghem, "TAMCRA: a tunable accuracy multiple constraints routing algorithm," *Computer Communications*, vol. 23, no. 7, pp. 667–679, 2000.

Caminos nodo-disjuntos, acotados y de menor costo entre dos nodos

Natalia Chiappara Guillermo Lacordelle

Tutores

Pablo Romero Franco Robledo

Universidad de la República

Proyecto de Grado - Ingeniería en Computación, Abril 2014

Organización de la Presentación

Organización de la Presentación

Organización de la Presentación

Definición del problema

El problema de estudio consiste en:

Dado un grafo G cuyas aristas tienen costos positivos y dos nodos distintos s y t pertenecientes a G, hallar k caminos nodo-disjuntos entre ambos con la restricción de que cada camino tenga a lo sumo d aristas y con el mínimo costo total.

Complejidad computacional

- El problema es NPO-Completo (A. Bley, On the complexity of vertex-disjoint length-restricted path problems)
- Esto motiva la implementación de una heurística para su resolución

Organización de la Presentación

Motivación

Proporcionar más de un camino entre nodos de origen y destino permite:

- mantener el funcionamiento de la red en caso de fallas
- balancear la carga del tráfico entre los nodos

Motivación

Proporcionar más de un camino entre nodos de origen y destino permite:

- mantener el funcionamiento de la red en caso de fallas
- balancear la carga del tráfico entre los nodos

Motivación

Proporcionar más de un camino entre nodos de origen y destino permite:

- mantener el funcionamiento de la red en caso de fallas
- balancear la carga del tráfico entre los nodos

Motivación

Proporcionar más de un camino entre nodos de origen y destino permite:

- mantener el funcionamiento de la red en caso de fallas
- balancear la carga del tráfico entre los nodos

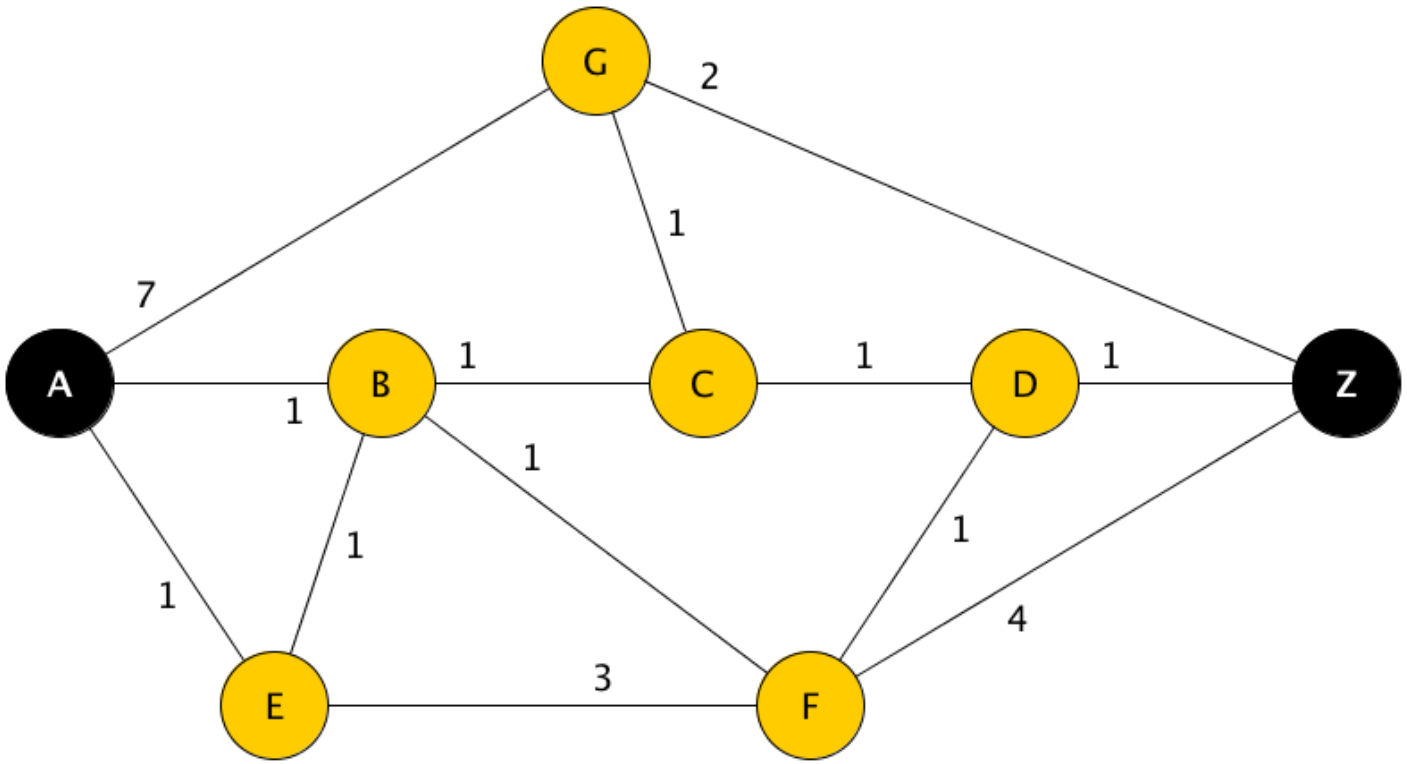
Motivación

Proporcionar más de un camino entre nodos de origen y destino permite:

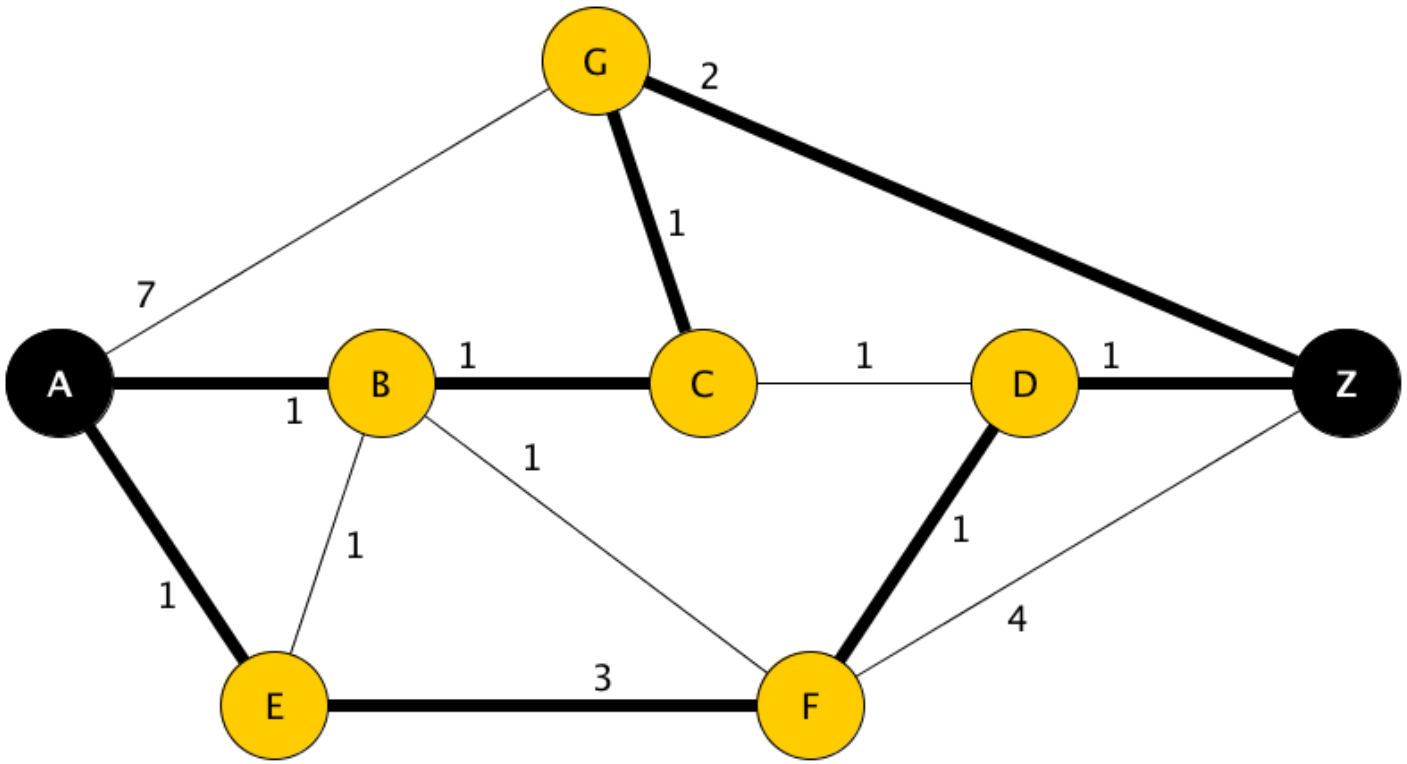
- mantener el funcionamiento de la red en caso de fallas
- balancear la carga del tráfico entre los nodos

Organización de la Presentación

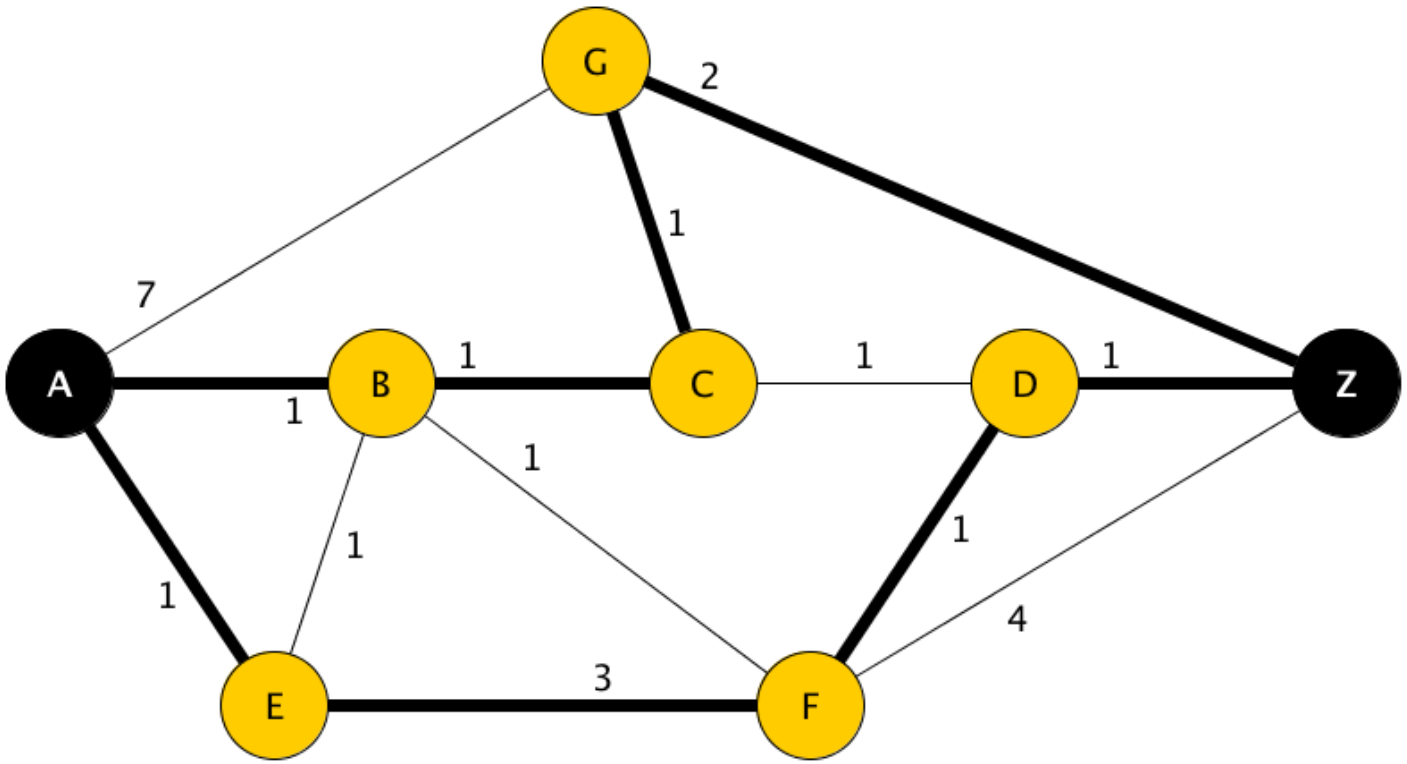
Ejemplo: $k = 2$, caminos no acotados



Ejemplo: $k = 2$, caminos no acotados

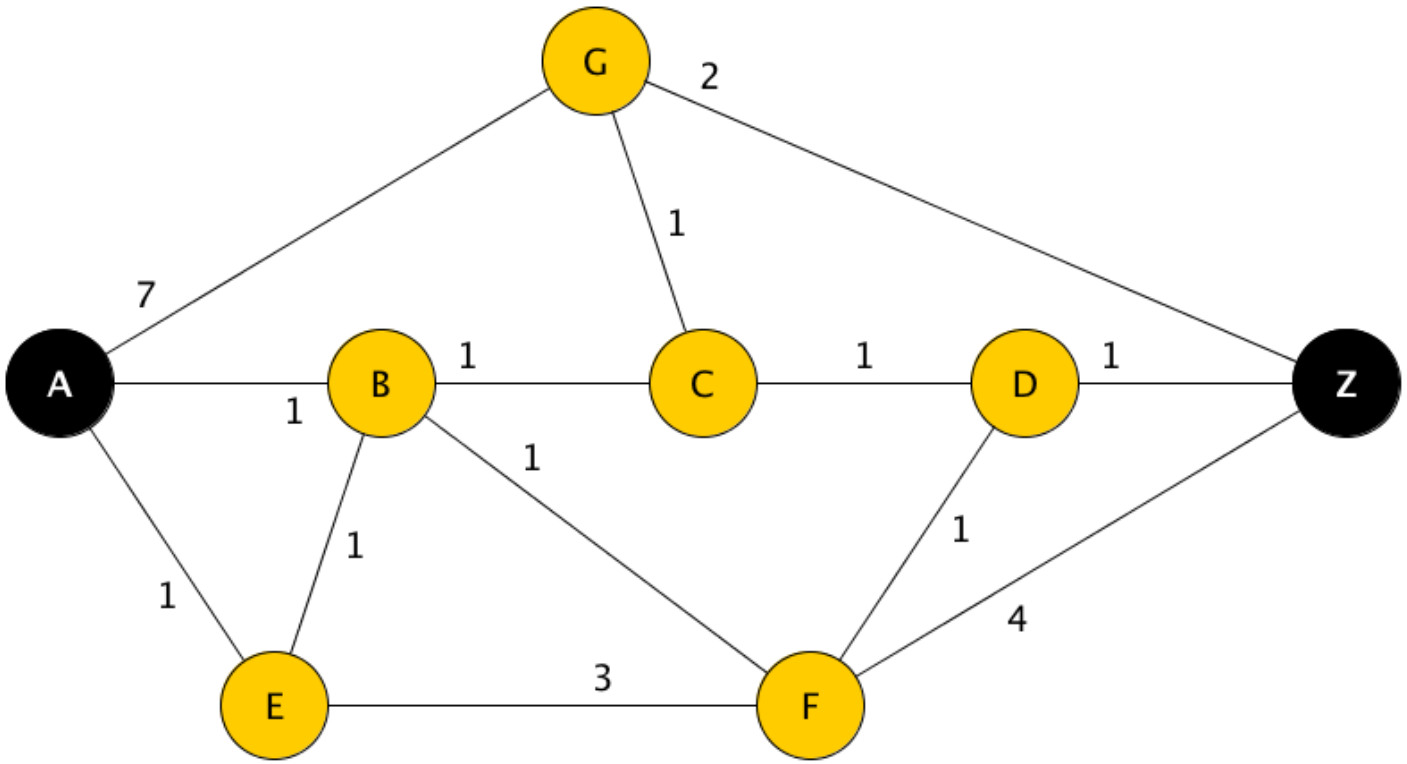


Ejemplo: $k = 2$, caminos no acotados

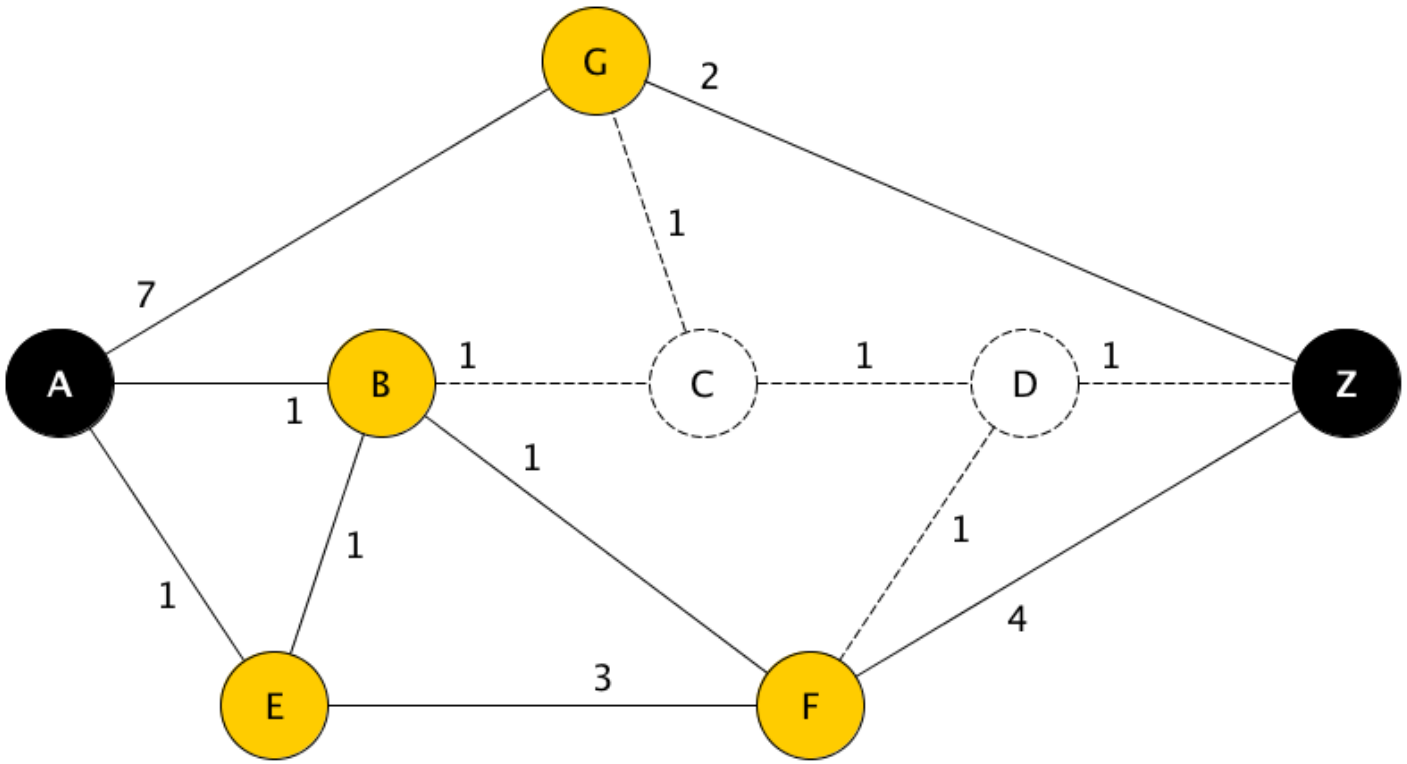


Costo = 11

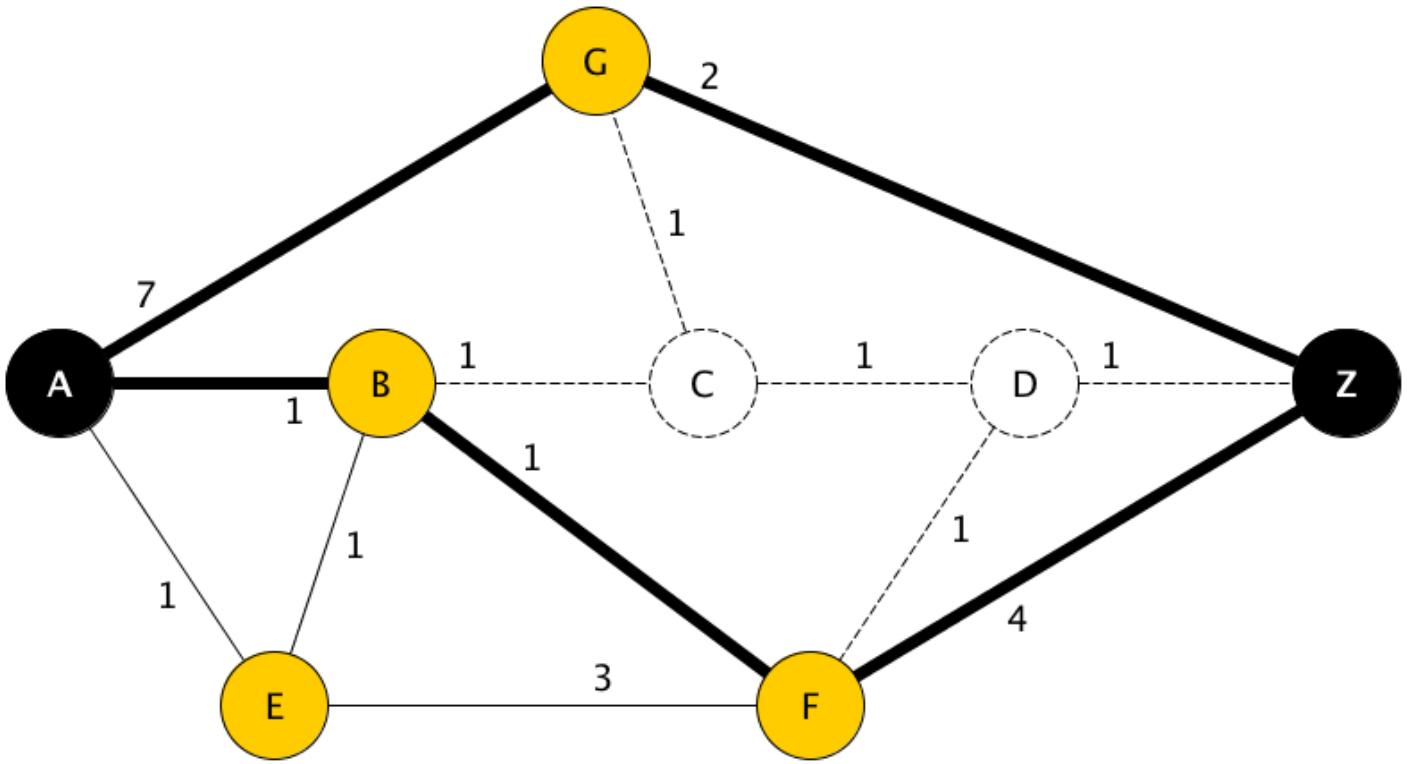
Ejemplo: $k = 2,\ d = 3$



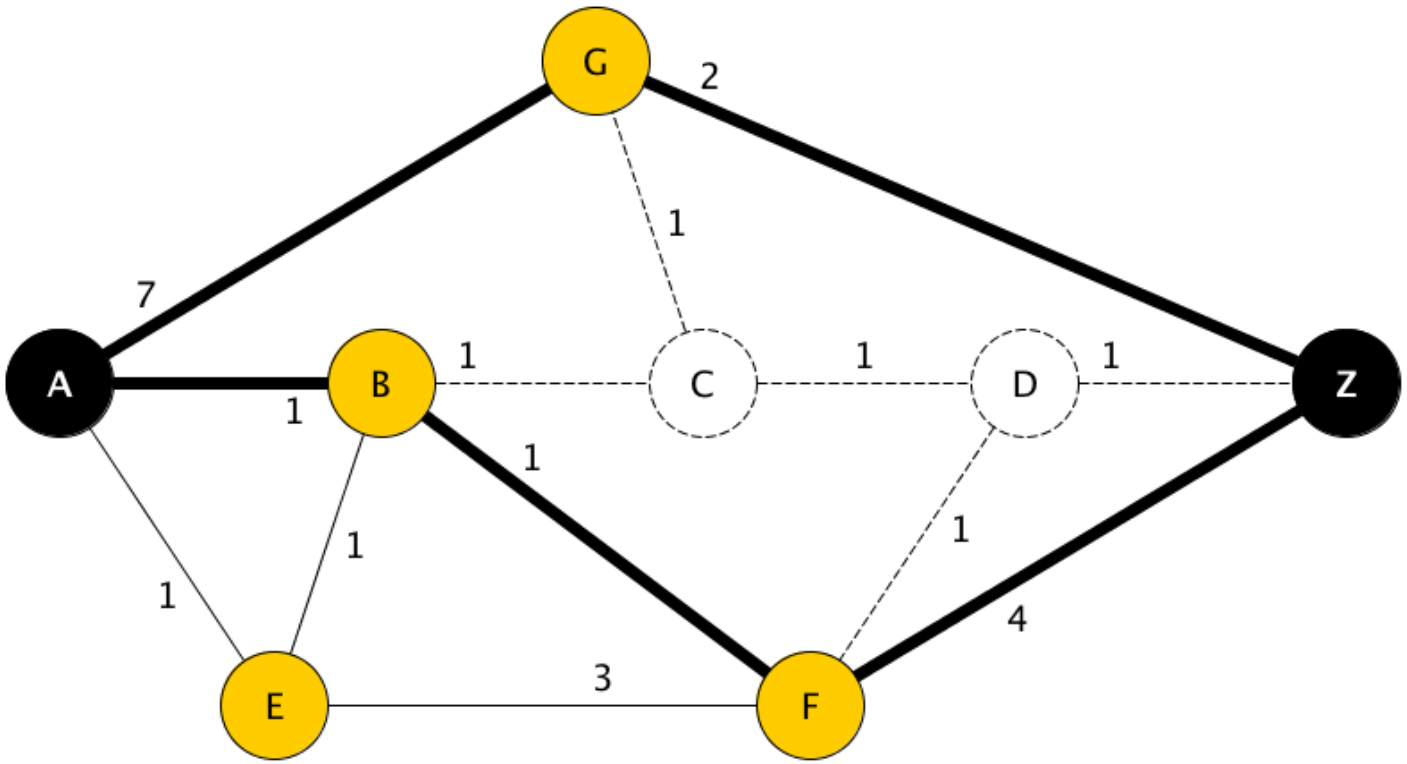
Ejemplo: $k = 2,\ d = 3$



Ejemplo: $k = 2,\ d = 3$



Ejemplo: $k = 2,\ d = 3$



Costo = 15

Organización de la Presentación

Organización de la Presentación

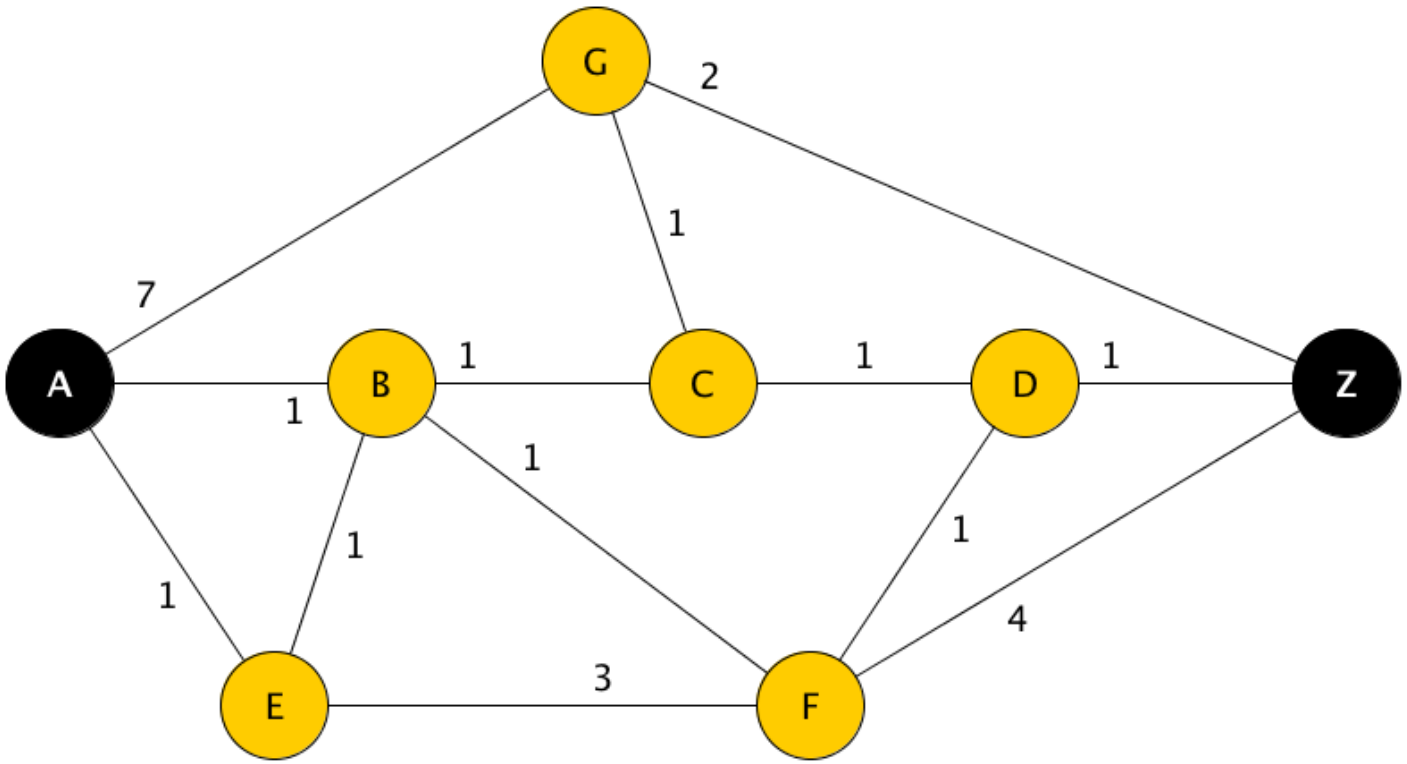
Greedy

- Heurística golosa utilizada como punto de comparación
- Construye una solución mediante el agregado sucesivo de caminos menos costosos

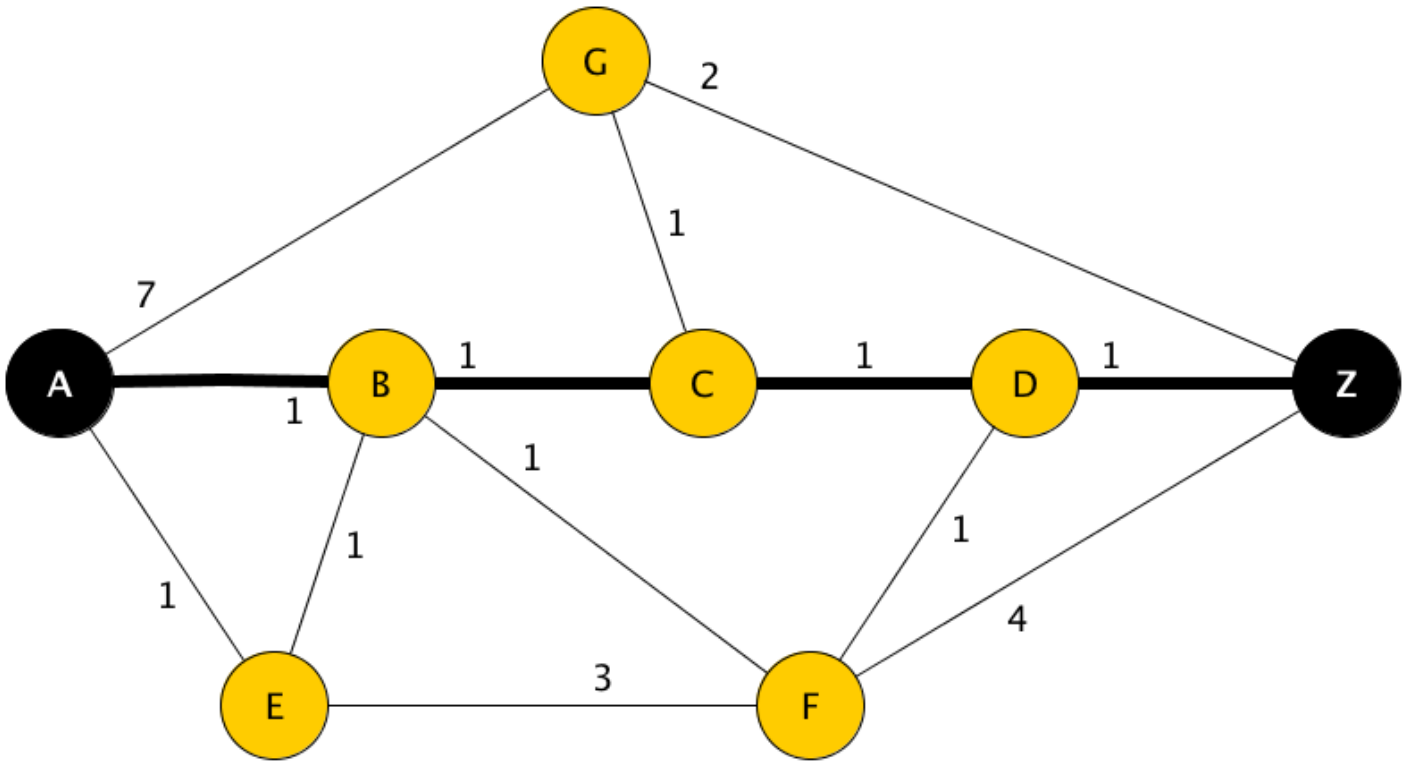
Algoritmo Greedy

```
procedure Greedy(G : Grafo, c : E → ℝ⁺, k, d)
Solución ← ϕ
while k > 0
        SP ← BellmanFord(G, Inicio, Fin, d)
        Solución ← Solución ∪ SP
        G ← G \ V(SP)
        k ← k − 1
end
return Solución
```

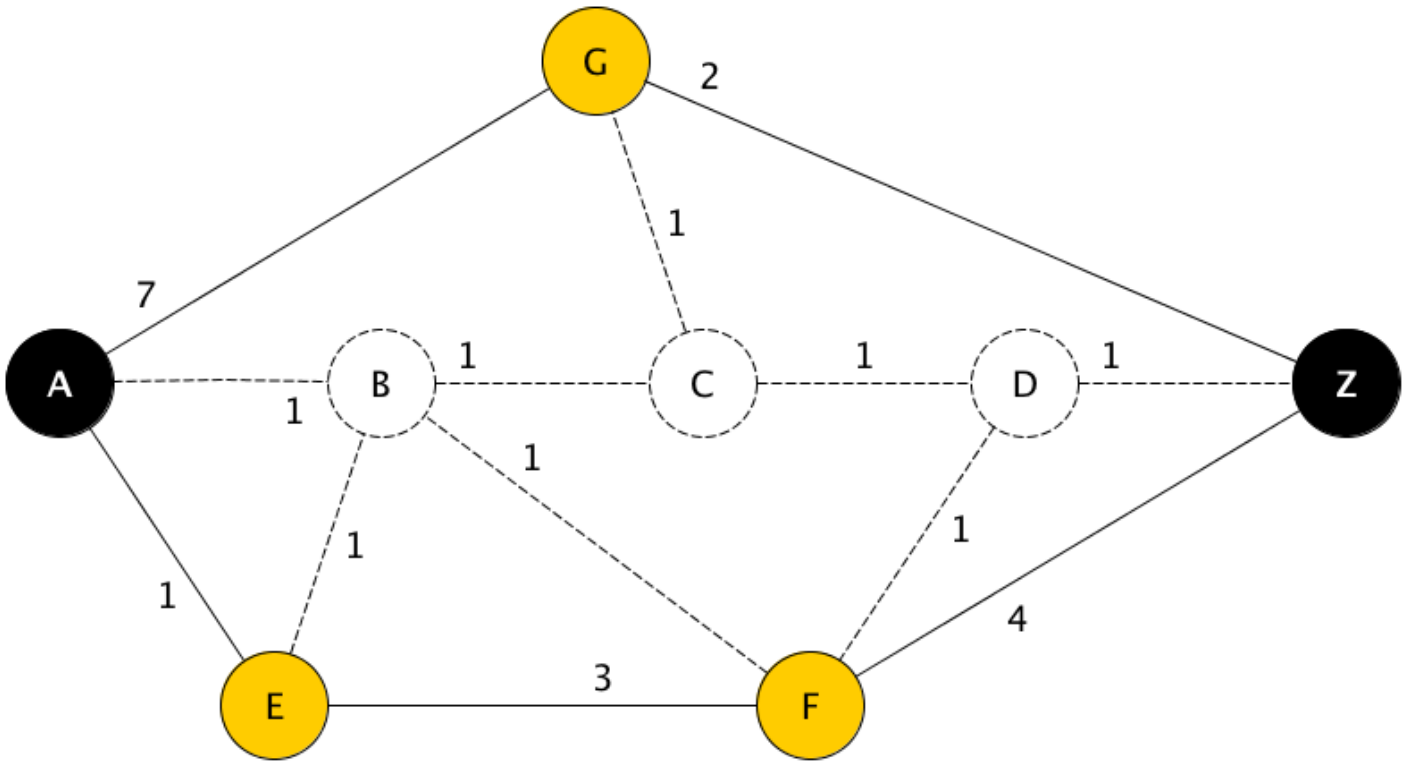
Ejemplo: $k = 2,\ d = 4$



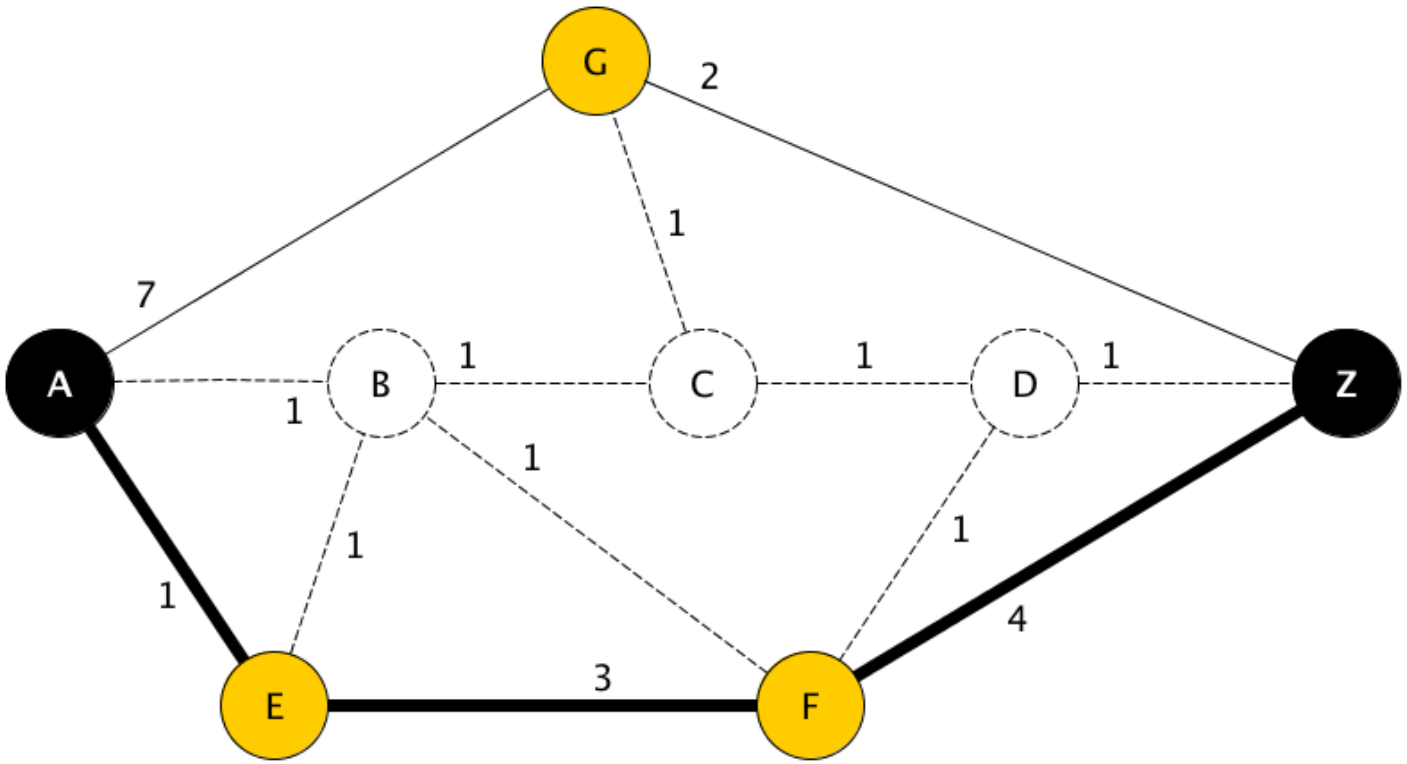
Ejemplo: $k = 2,\ d = 4$



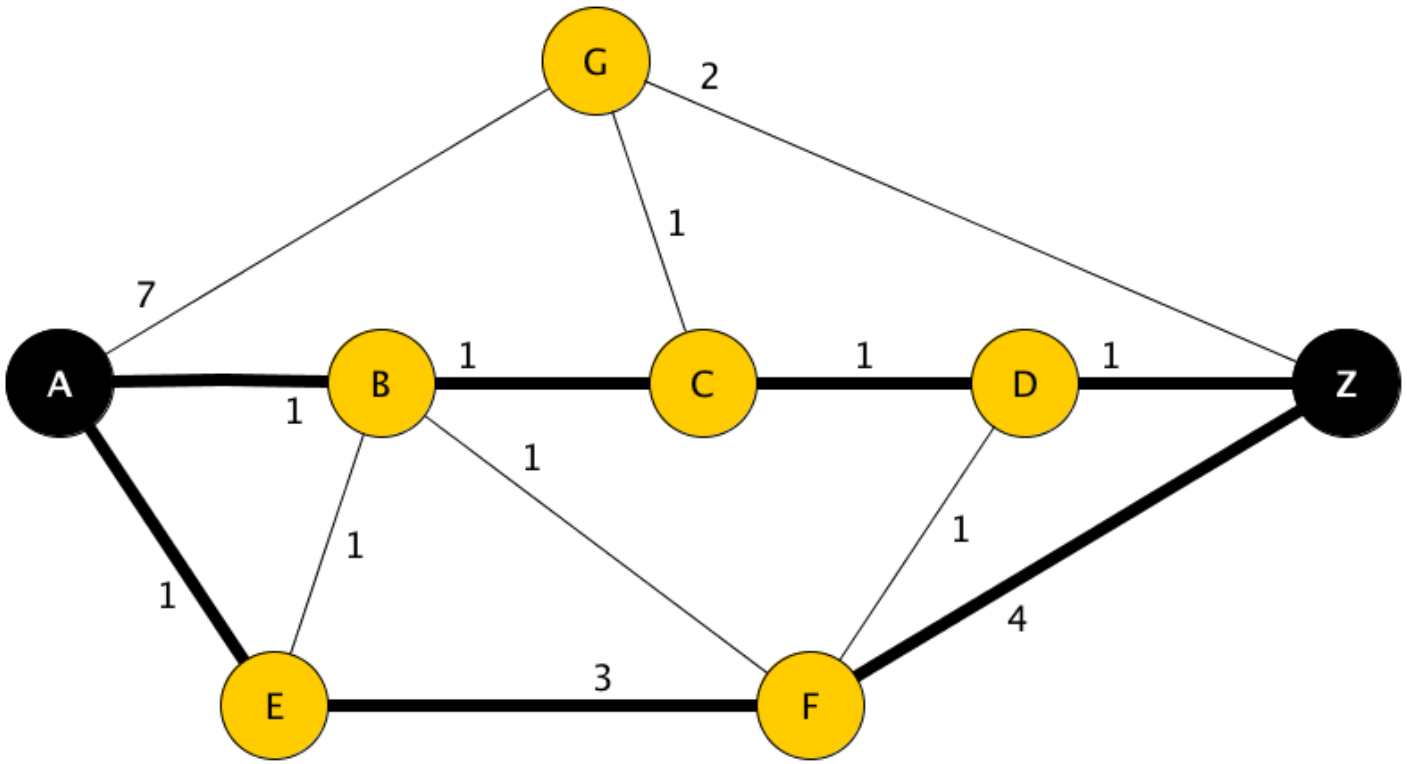
Ejemplo: $k = 2,\ d = 4$



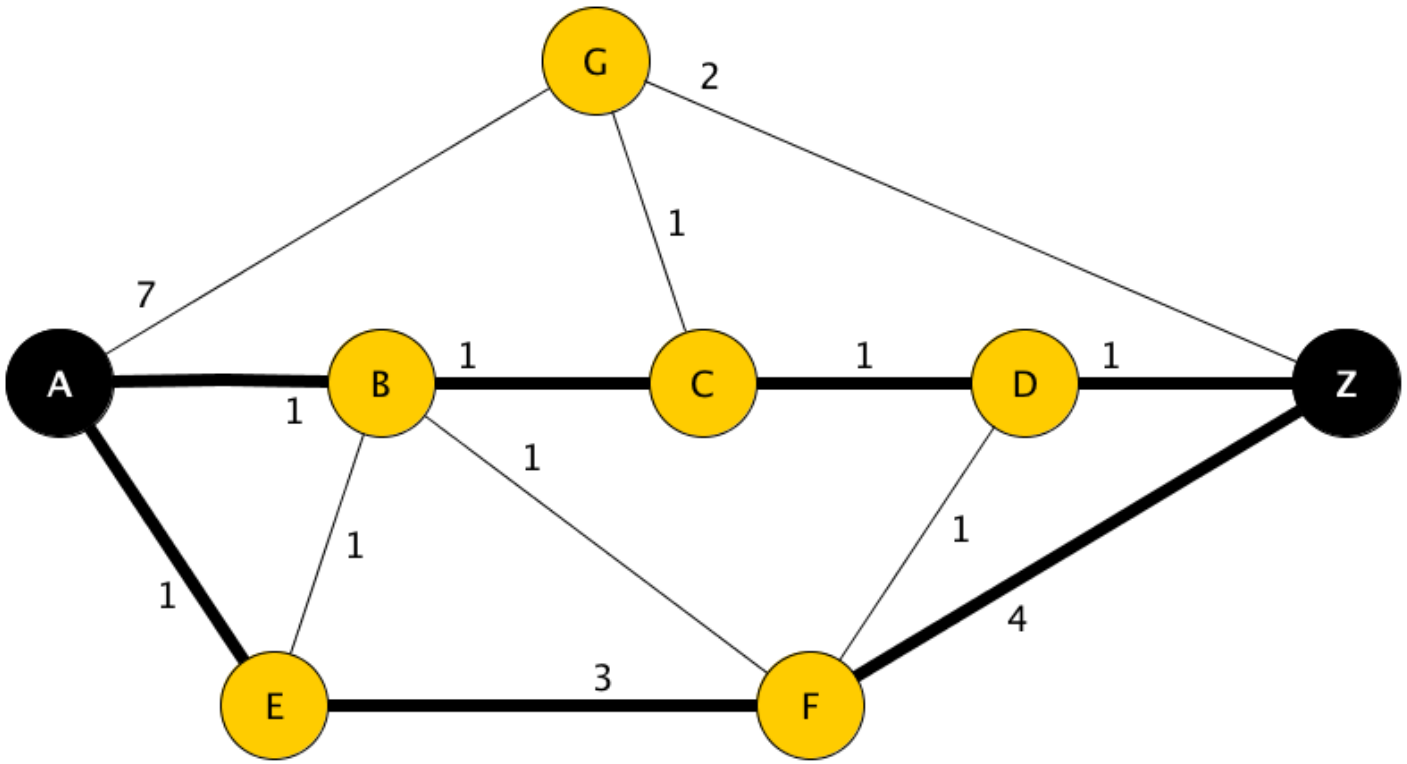
Ejemplo: $k = 2,\ d = 4$



Ejemplo: $k = 2,\ d = 4$



Ejemplo: $k = 2,\ d = 4$



Costo = 12

Organización de la Presentación

GRASP

- GRASP *Greedy Randomized Adaptive Search Procedure*
- Es una metaheurística de optimización combinatoria
- Ha logrado los mejores resultados en varios problemas sobre grafos.

GRASP

- GRASP *Greedy Randomized Adaptive Search Procedure*
- Es una metaheurística de optimización combinatoria
- Ha logrado los mejores resultados en varios problemas sobre grafos.

GRASP

```
procedure GRASP(Iteraciones, TamañoLista)
f* ← ∞
for n = 1..Iteraciones do
    S ← ConstructorGolosoAleatorio(TamañoLista)
    S ← BúsquedaLocal(S)
    if f(S) < f*
        S* ← S
        f* ← f(S)
    end
end
return S*
```

GRASP

```
procedure GRASP(Iteraciones, TamañoLista)
f* ← ∞
for n = 1..Iteraciones do
    S ← ConstructorGolosoAleatorio(TamañoLista)
    S ← BúsquedaLocal(S)
    if f(S) < f*
        S* ← S
        f* ← f(S)
    end
end
return S*
```

Fase de construcción

- Construye una solución factible para el problema
- Utiliza un algoritmo goloso y aleatorio para lograrlo
- Comienza con una solución vacía y en cada paso agrega un elemento
- Cada elemento es seleccionado, aleatoriamente, de una lista de candidatos llamada *Restricted Candidate List*

Fase de construcción

```
procedure ConstructorGolosoAleatorio(TamañoLista)
S ← ϕ
while not EsFactible(S) do
        RCL ← GenerarRCL(TamañoLista)
        s ← SeleccionarElementoAleatoriamente(RCL)
        S ← S ∪ {s}
end
return S
```

GRASP

```
procedure GRASP(Iteraciones, TamañoLista)
f* ← ∞
for n = 1..Iteraciones do
    S ← ConstructorGolosoAleatorio(TamañoLista)
    S ← BúsquedaLocal(S)
    if f(S) < f*
        S* ← S
        f* ← f(S)
    end
end
return S*
```

Fase de búsqueda local

- Busca mejorar la solución generada en la fase de construcción
- Se define una estructura de vecindad para dicha solución
- Se define una estrategia de búsqueda dentro de la vecindad

Fase de búsqueda local

```
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
```

Organización de la Presentación

CADILAC

- CADILAC algoritmo de búsqueda de CAminos DIsjuntos de Largo ACotado
- Es una heurística basada en GRASP

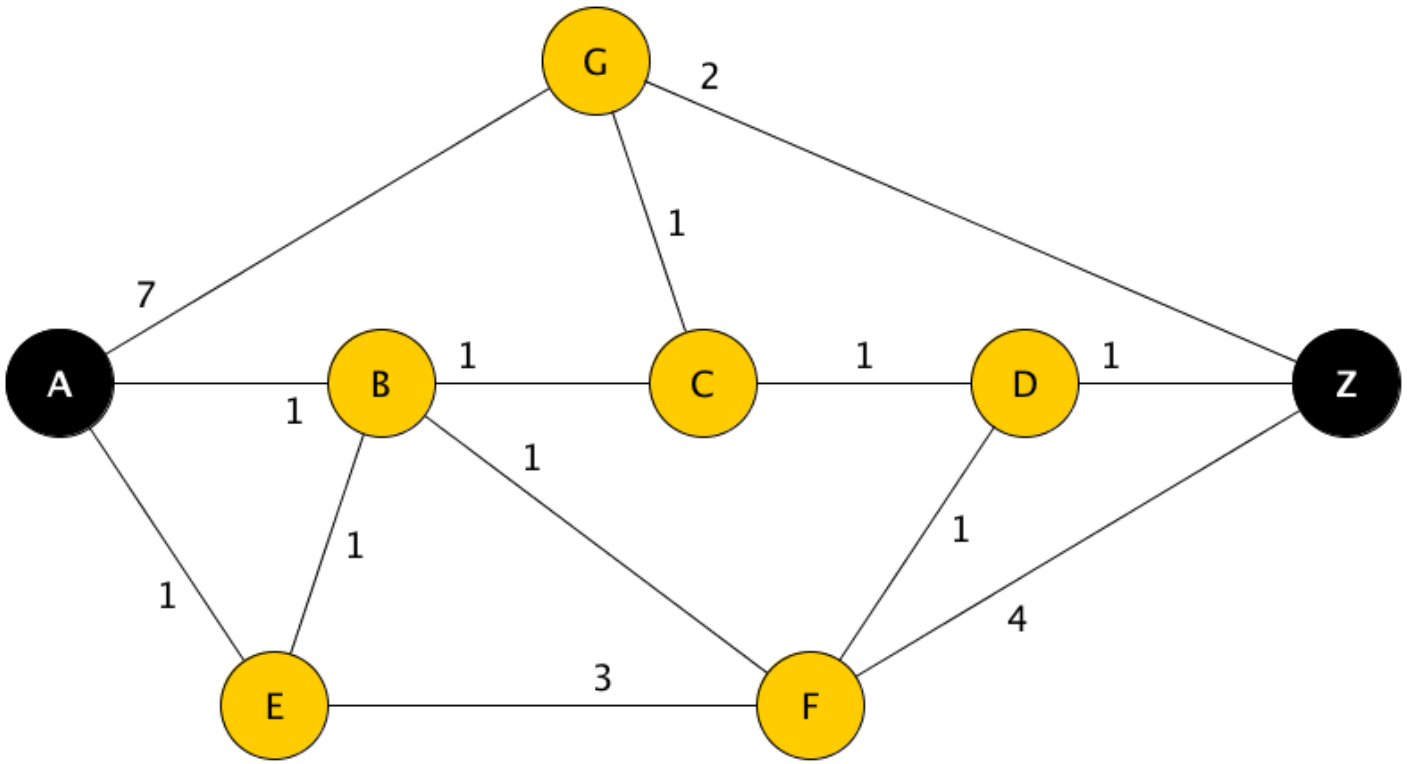
CADILAC

- CADILAC algoritmo de búsqueda de CAminos DIsjuntos de Largo ACotado
- Es una heurística basada en GRASP

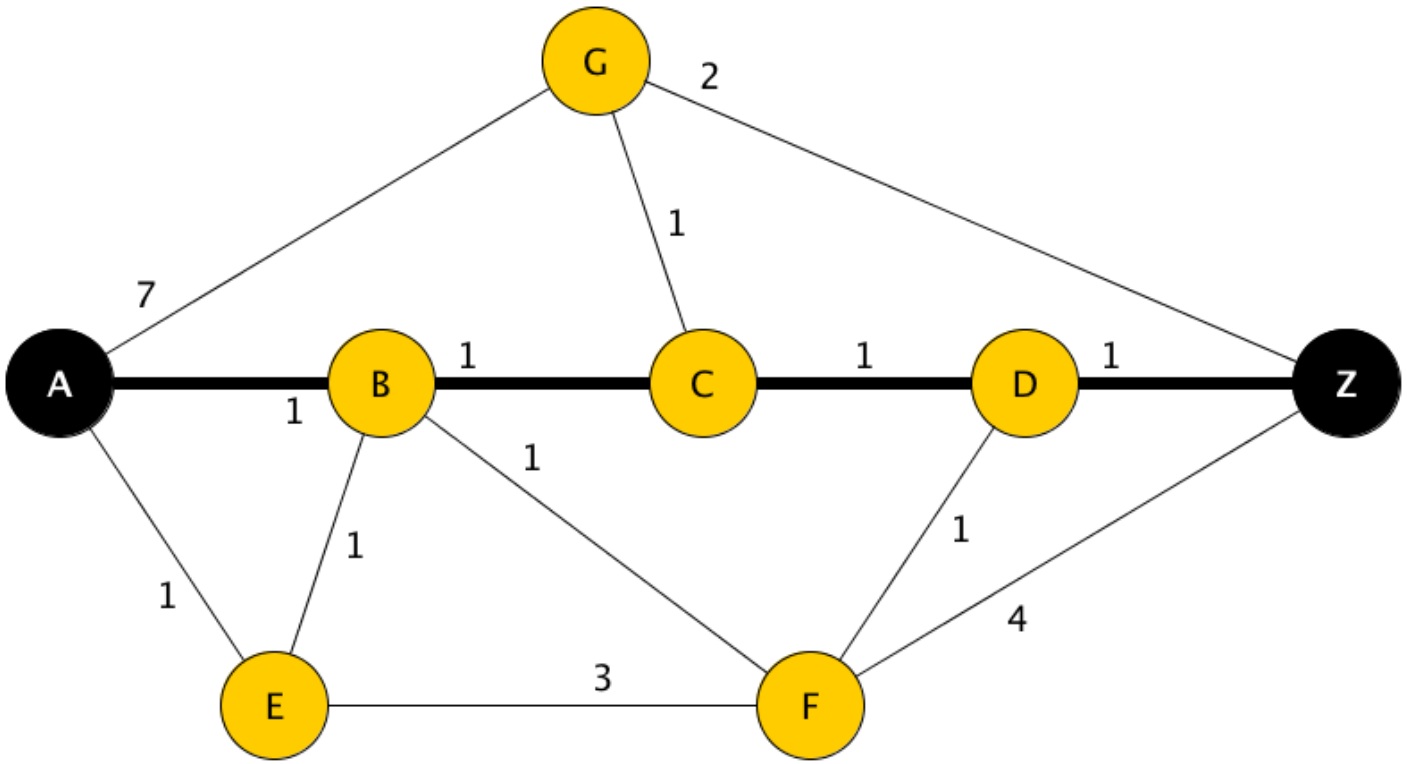
Fase de construcción

- Basado en el algoritmo de Bhandari (R. Bhandari, Survivable networks: algorithms for diverse routing. Springer, 1999)
- Toma ideas del algoritmo DIMCRA (Y. Guo, F. Kuipers, and P. Van Mieghem, "Link-disjoint paths for reliable QoS routing," International Journal of Communication Systems, 2003)

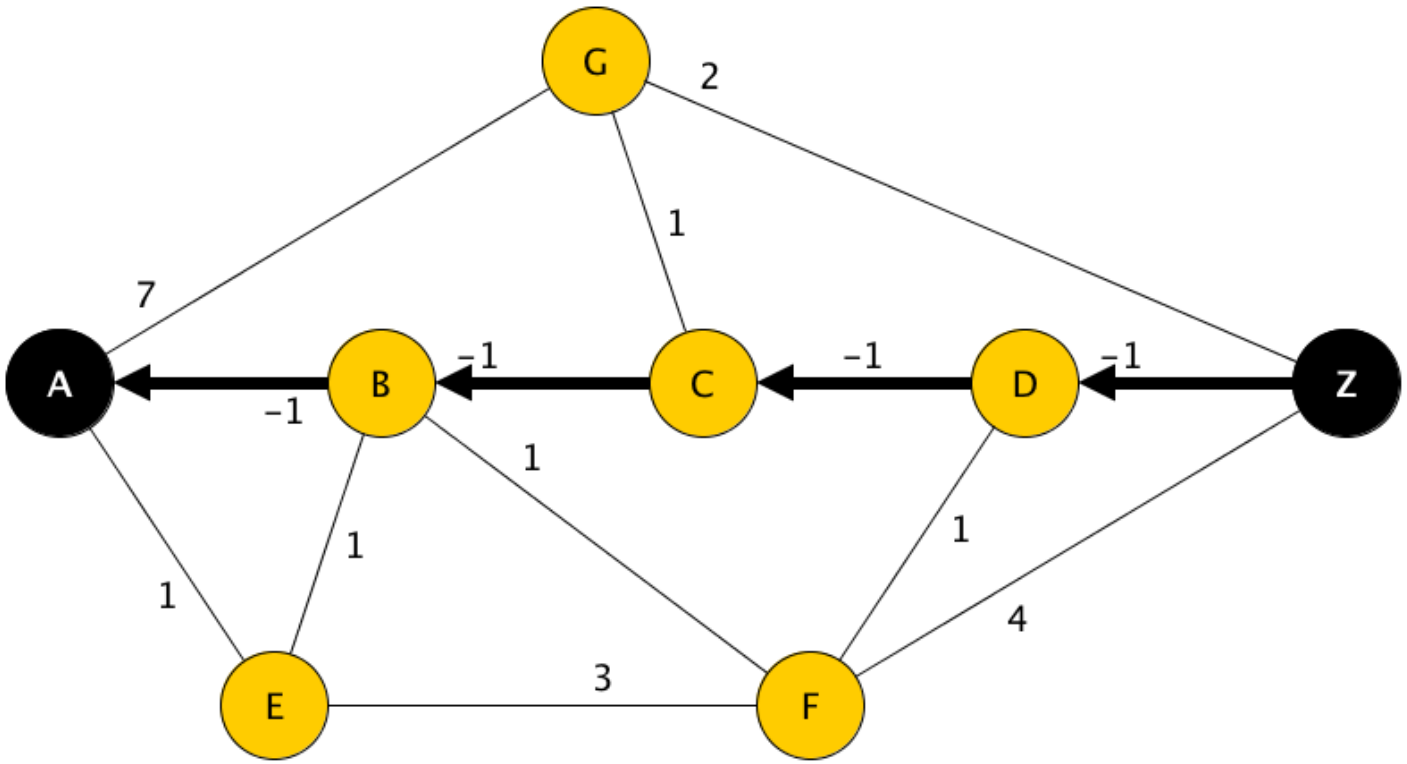
Algoritmo de Bhandari



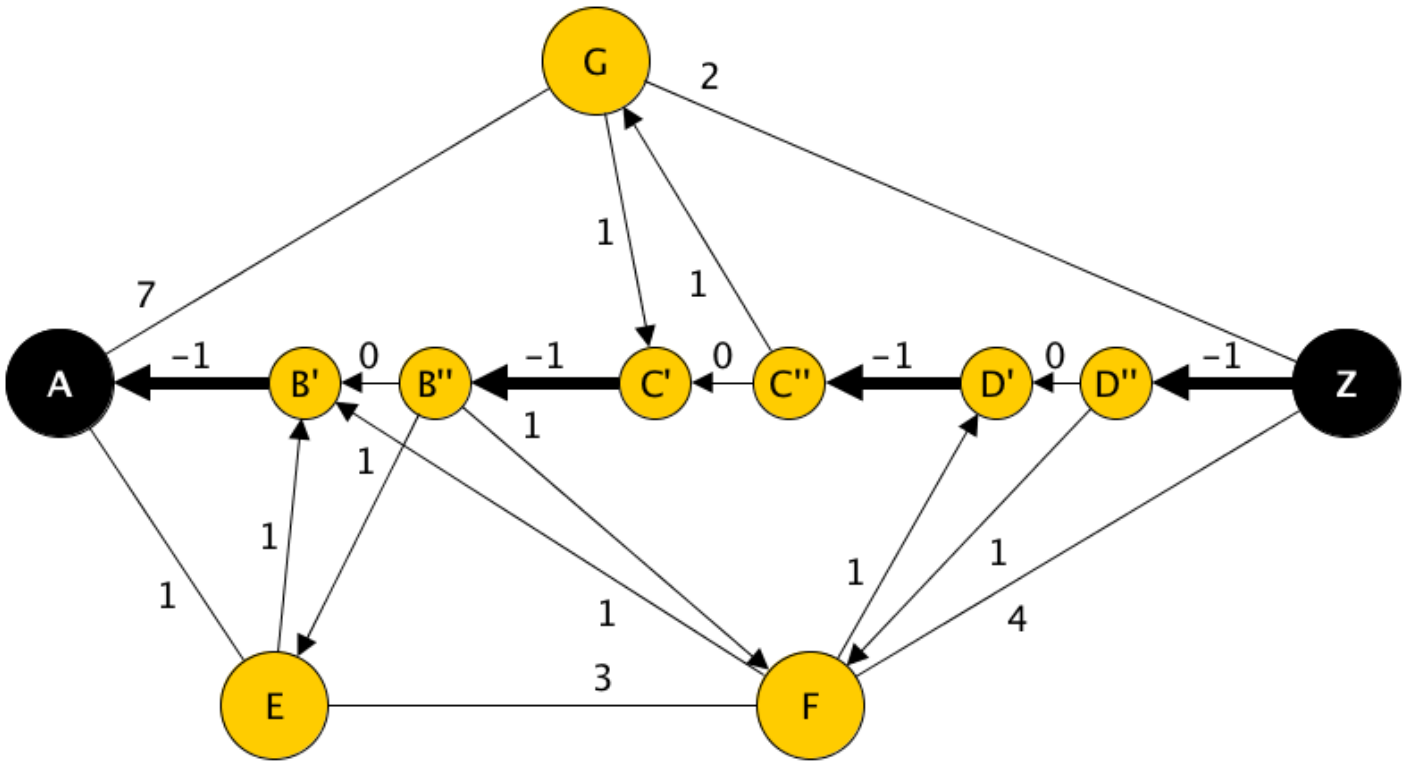
Algoritmo de Bhandari



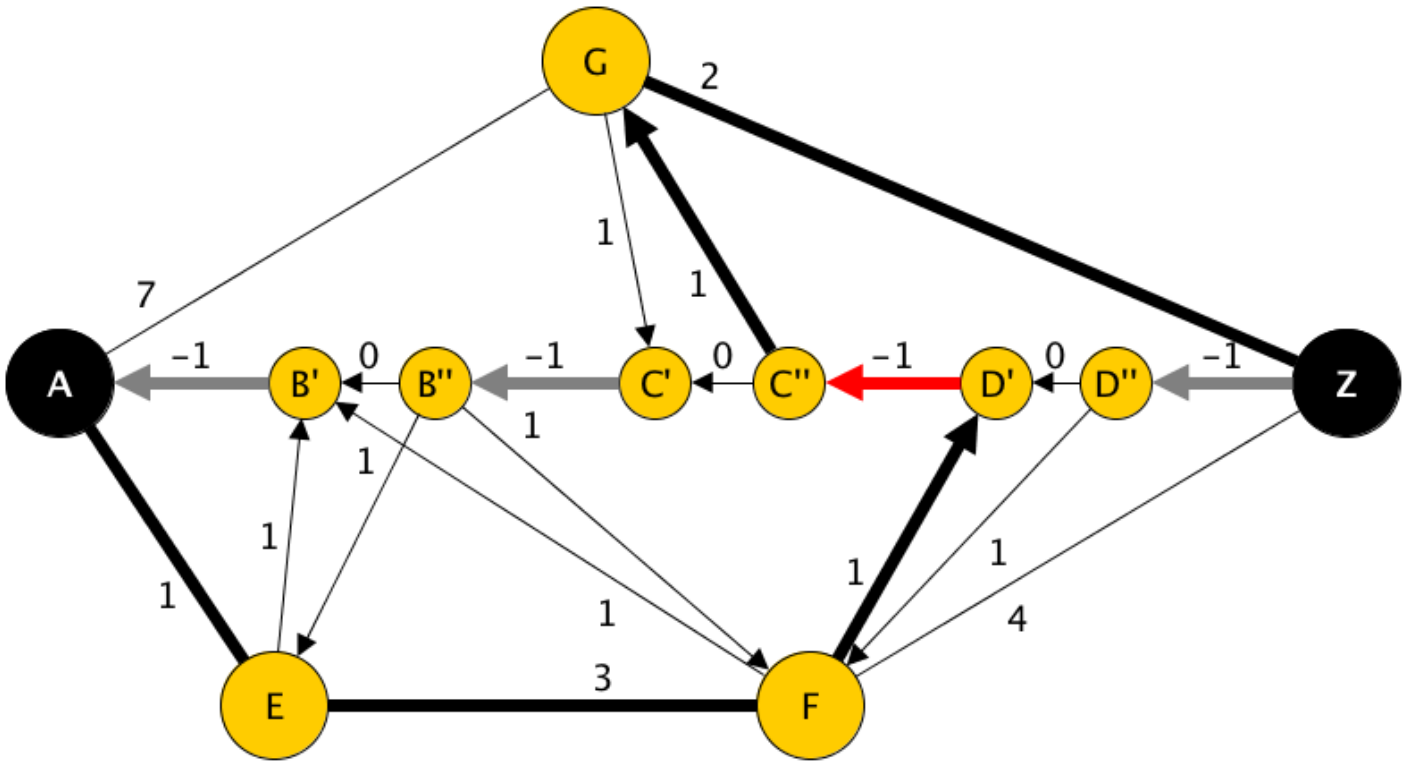
Algoritmo de Bhandari



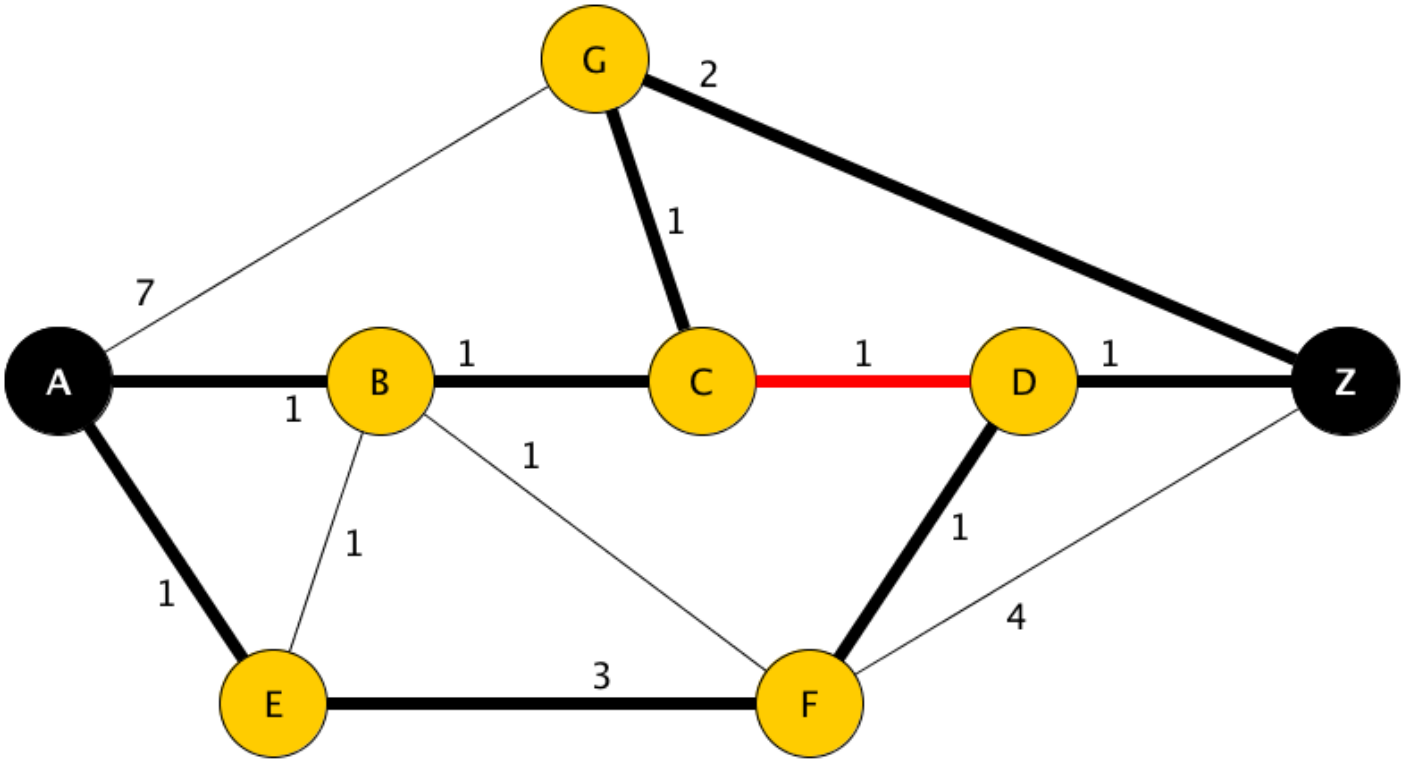
Algoritmo de Bhandari



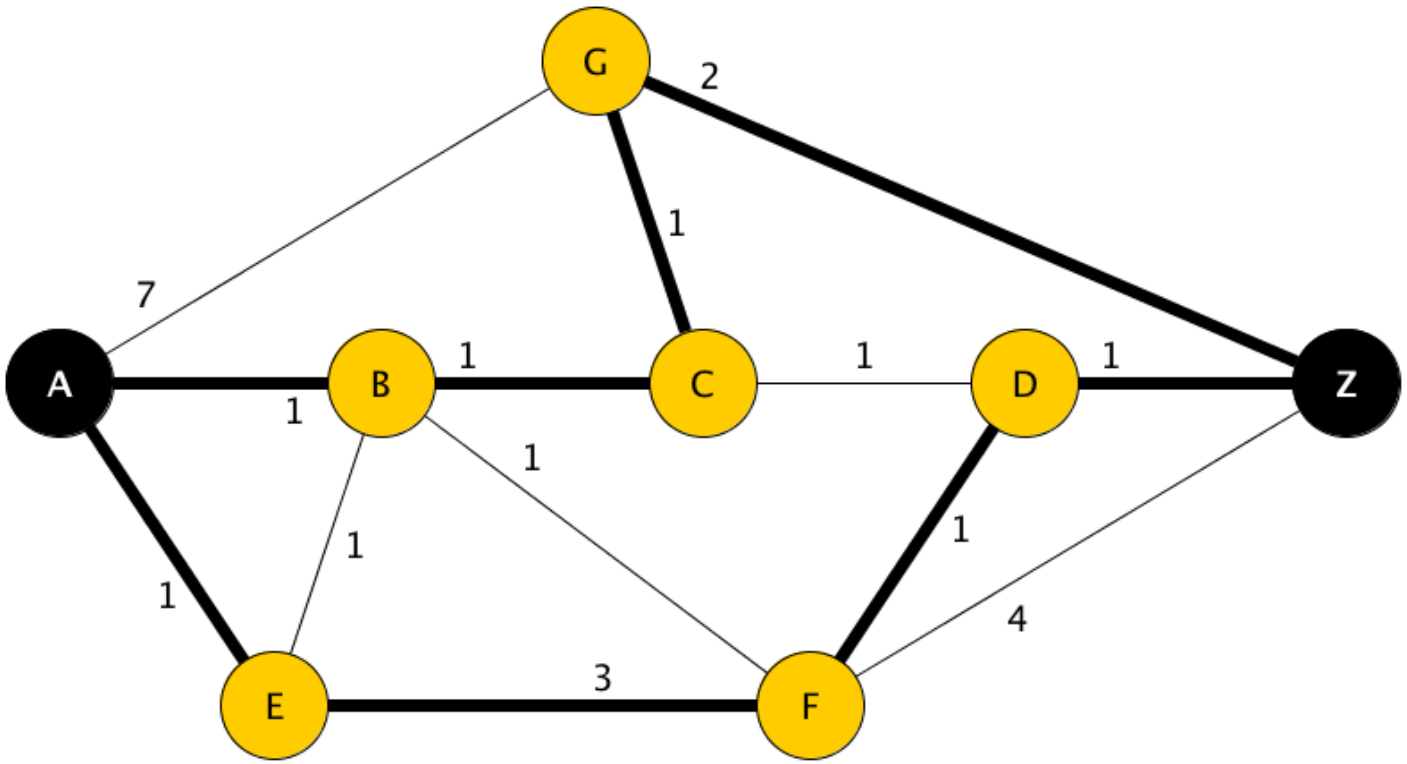
Algoritmo de Bhandari



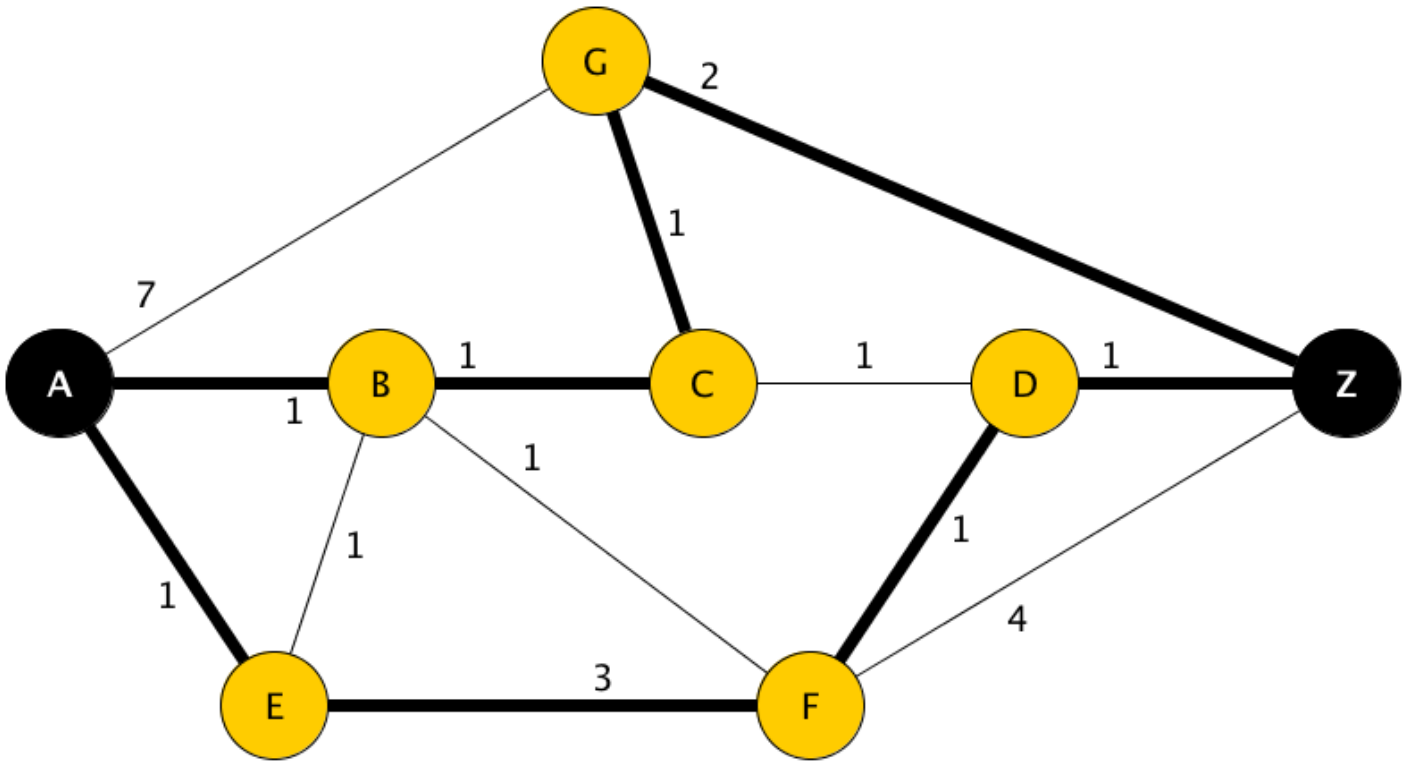
Algoritmo de Bhandari



Algoritmo de Bhandari



Algoritmo de Bhandari



Costo = 11

Fase de construcción

Diferencias con el algoritmo de Bhandari:

- Los costos son vectoriales de dos dimensiones
- Se asigna costo $\vec{0}$ en lugar de -1 al invertir la solución
- La búsqueda de caminos es aleatoria
- Los caminos encontrados están acotados

Particularidad adicional:

- Al buscar un camino se relajan las restricciones de largo

Fase de construcción

Diferencias con el algoritmo de Bhandari:

- Los costos son vectoriales de dos dimensiones
- Se asigna costo $\vec{0}$ en lugar de -1 al invertir la solución
- La búsqueda de caminos es aleatoria
- Los caminos encontrados están acotados

Particularidad adicional:

- Al buscar un camino se relajan las restricciones de largo

Fase de construcción

Diferencias con el algoritmo de Bhandari:

- Los costos son vectoriales de dos dimensiones
- Se asigna costo $\vec{0}$ en lugar de -1 al invertir la solución
- La búsqueda de caminos es aleatoria
- Los caminos encontrados están acotados

Particularidad adicional:

- Al buscar un camino se relajan las restricciones de largo

Fase de construcción

Diferencias con el algoritmo de Bhandari:

- Los costos son vectoriales de dos dimensiones
- Se asigna costo $\vec{0}$ en lugar de -1 al invertir la solución
- La búsqueda de caminos es aleatoria
- Los caminos encontrados están acotados

Particularidad adicional:

- Al buscar un camino se relajan las restricciones de largo

Fase de construcción

Diferencias con el algoritmo de Bhandari:

- Los costos son vectoriales de dos dimensiones
- Se asigna costo $\vec{0}$ en lugar de -1 al invertir la solución
- La búsqueda de caminos es aleatoria
- Los caminos encontrados están acotados

Particularidad adicional:

- Al buscar un camino se relajan las restricciones de largo

Fase de construcción

Diferencias con el algoritmo de Bhandari:

- Los costos son vectoriales de dos dimensiones
- Se asigna costo $\vec{0}$ en lugar de -1 al invertir la solución
- La búsqueda de caminos es aleatoria
- Los caminos encontrados están acotados

Particularidad adicional:

- Al buscar un camino se relajan las restricciones de largo

Fase de construcción

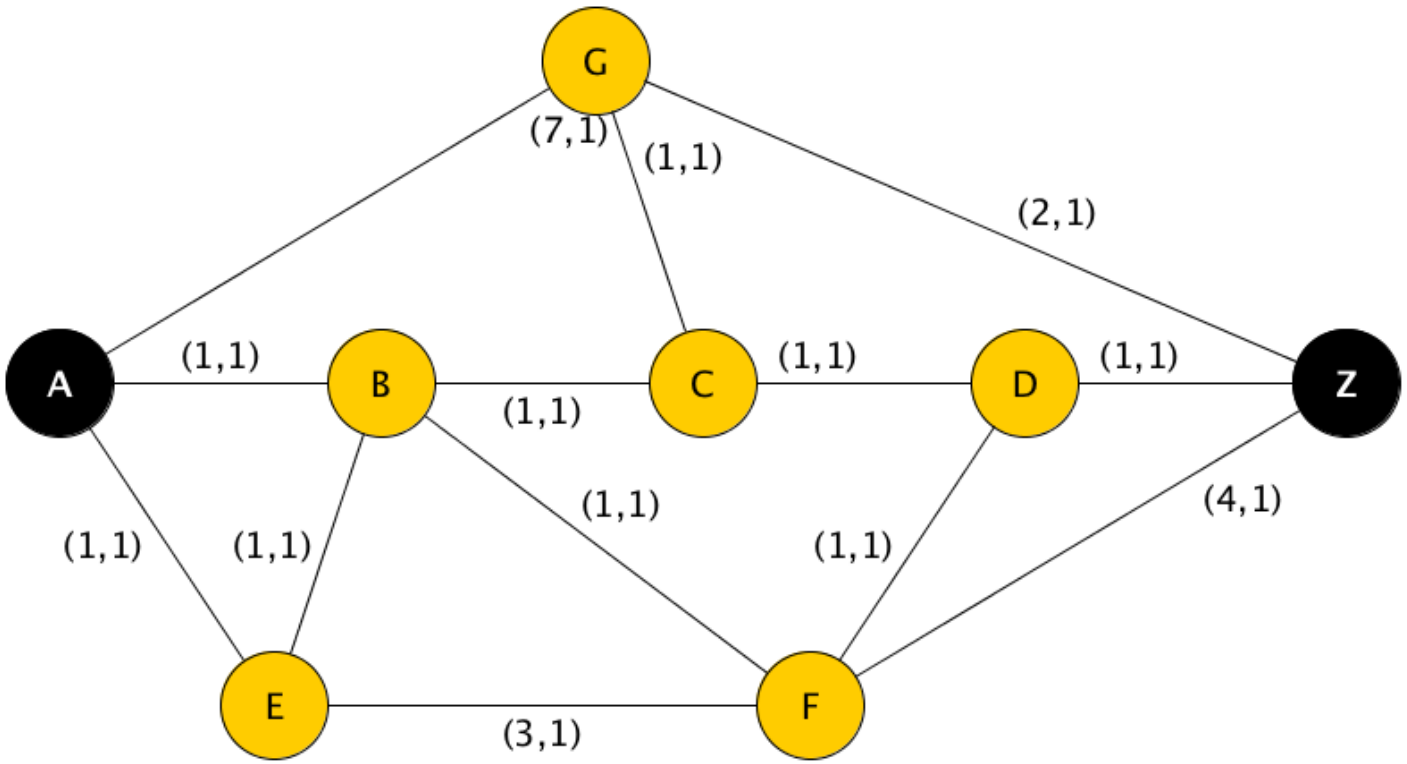
Diferencias con el algoritmo de Bhandari:

- Los costos son vectoriales de dos dimensiones
- Se asigna costo $\vec{0}$ en lugar de -1 al invertir la solución
- La búsqueda de caminos es aleatoria
- Los caminos encontrados están acotados

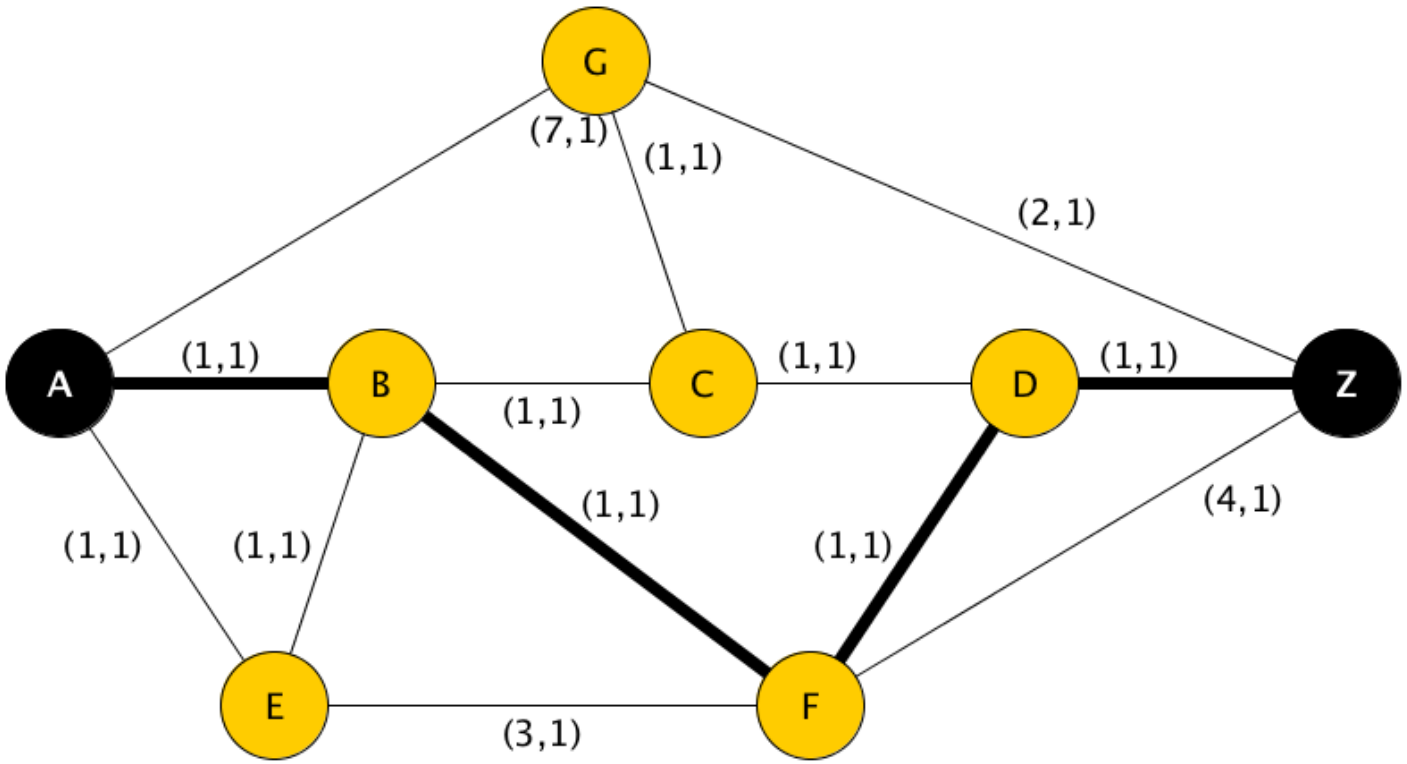
Particularidad adicional:

- Al buscar un camino se relajan las restricciones de largo

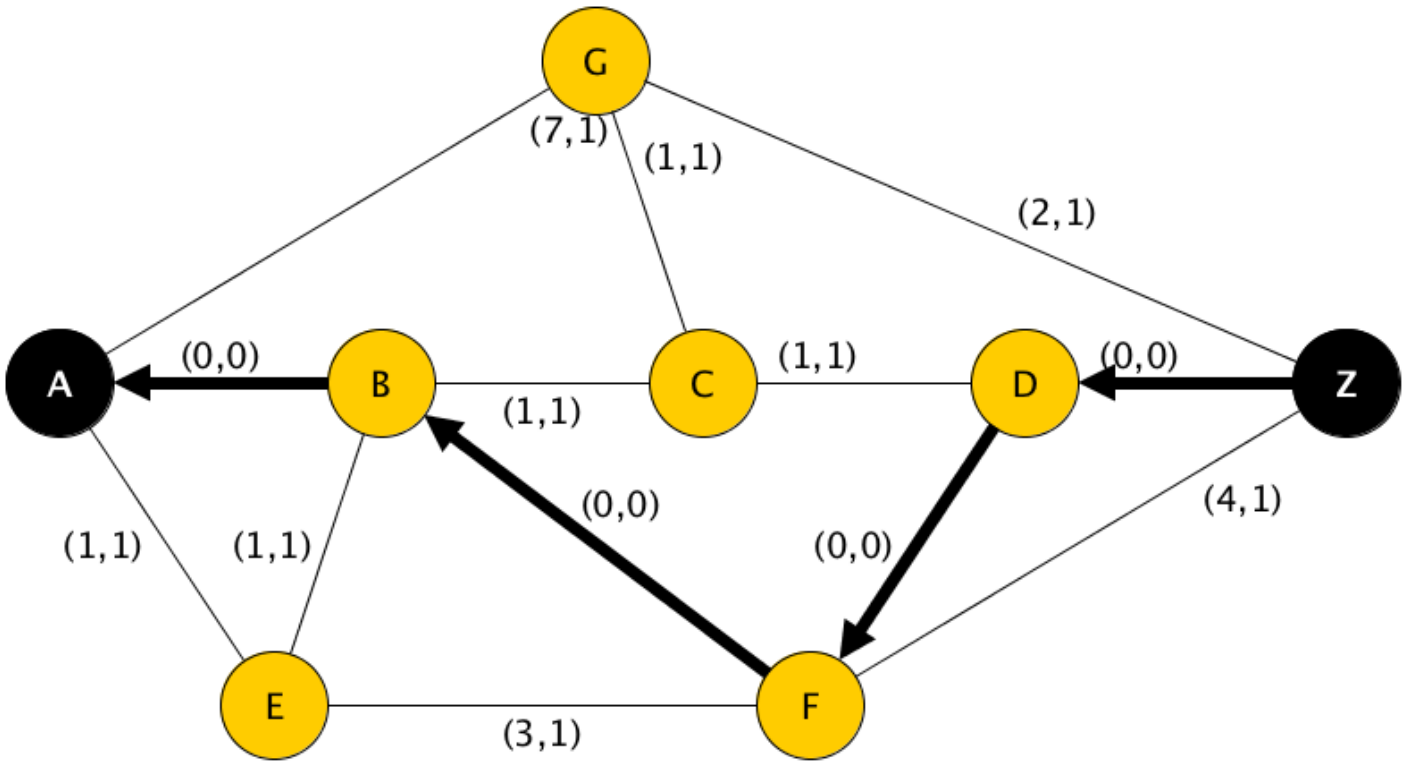
Ejemplo



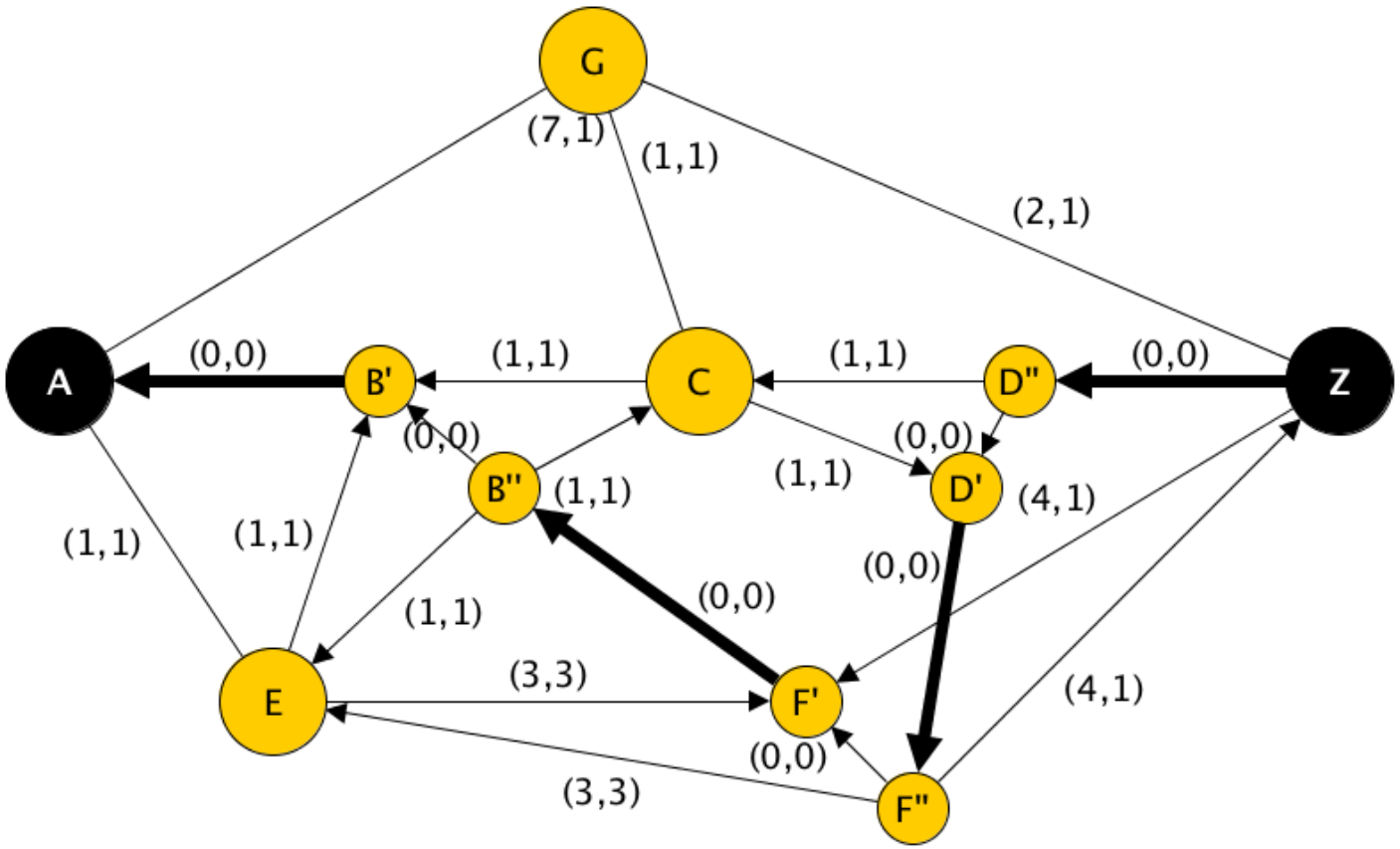
Ejemplo



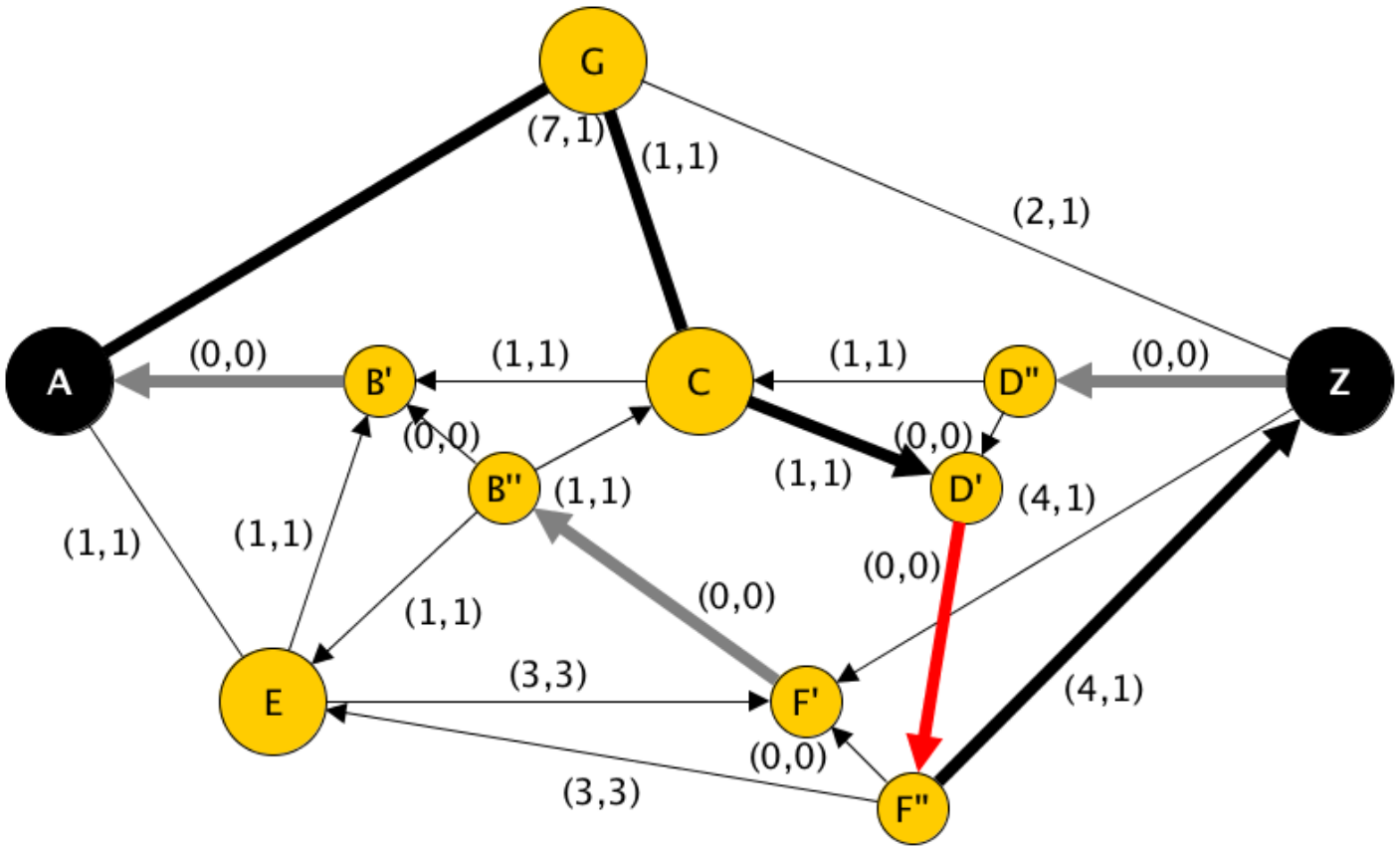
Ejemplo



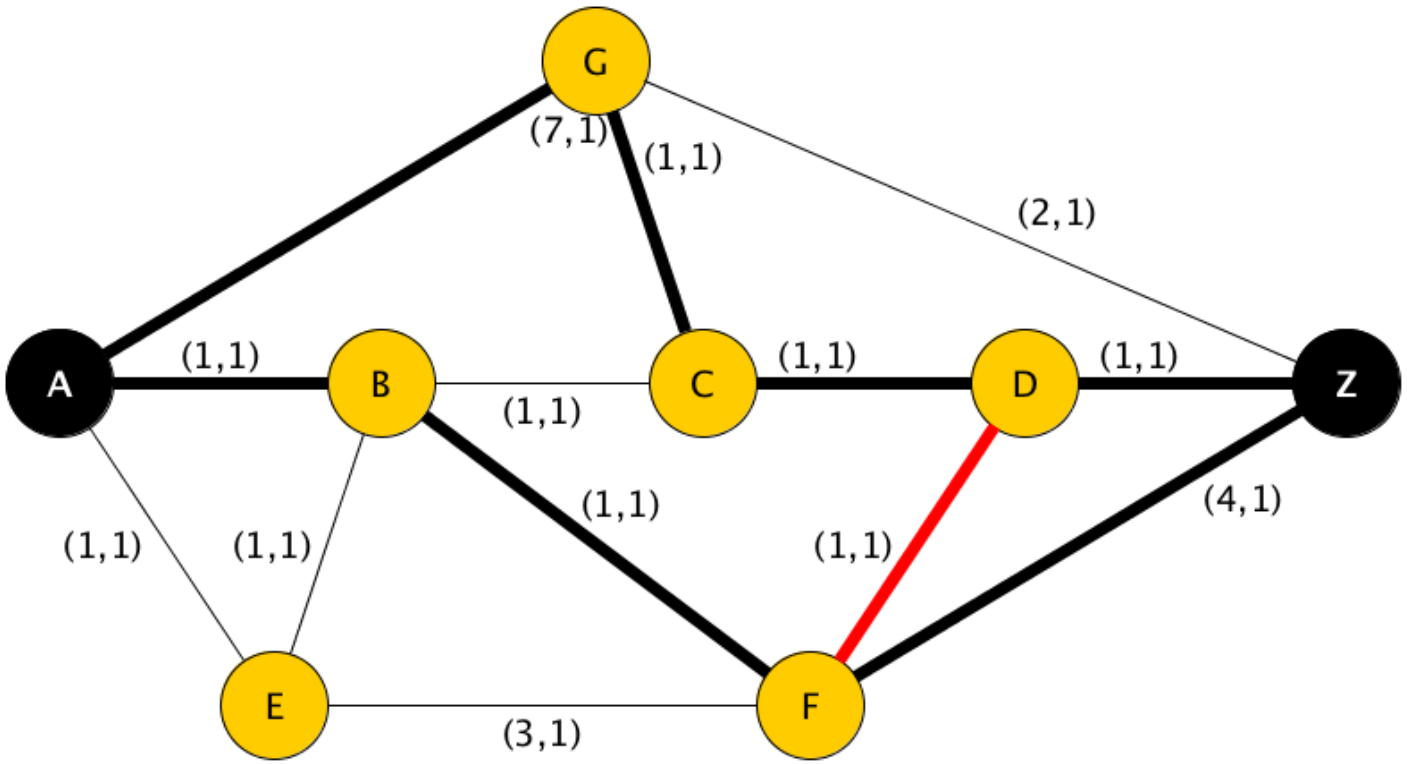
Ejemplo


G
(7,1)
(1,1)
(2,1)
A
(0,0)
B'
(1,1)
C
(1,1)
D"
(0,0)
Z
(0,0)
B"
(1,1)
(1,1)
D'
(4,1)
(1,1)
(1,1)
(0,0)
(0,0)
(1,1)
(3,3)
E
F'
(4,1)
(0,0)
(3,3)
F"

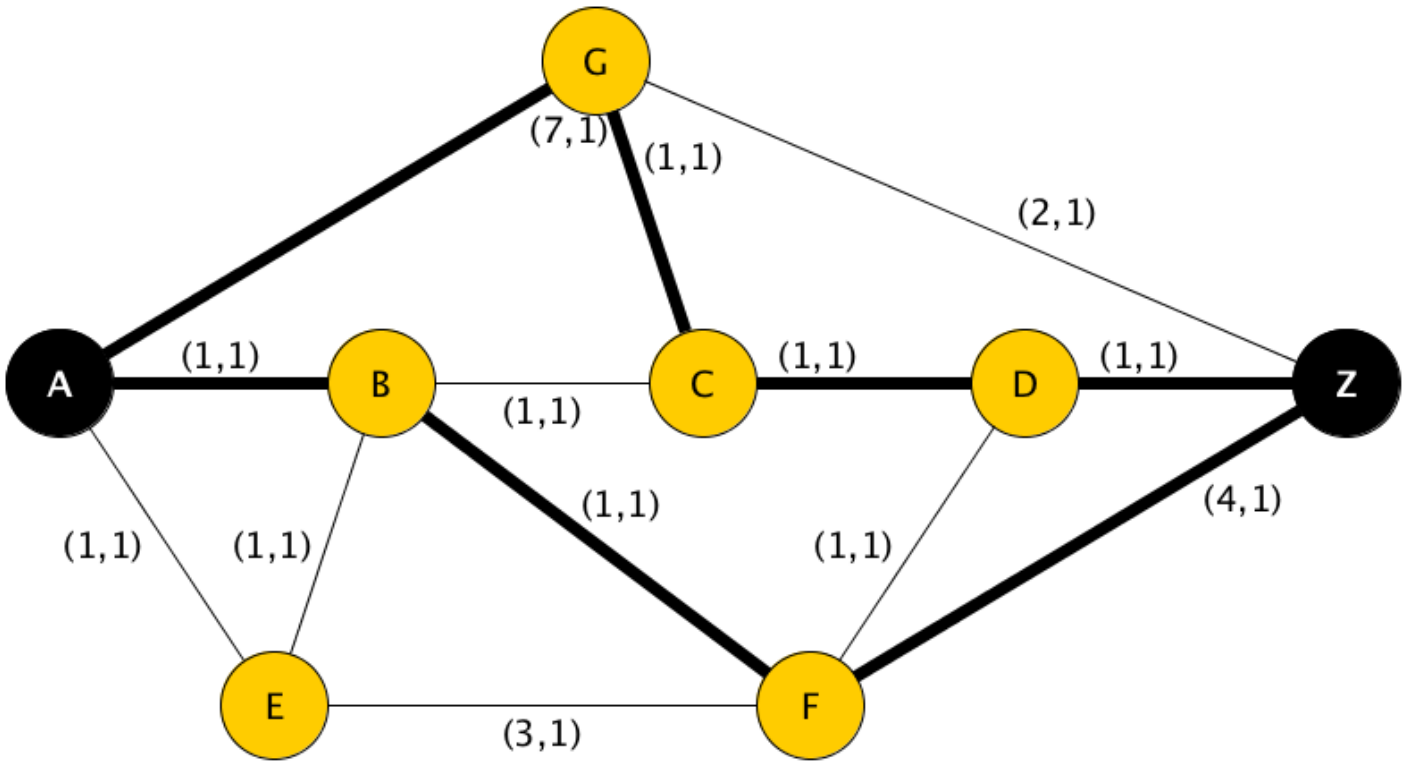
Ejemplo


G
(7,1)
(1,1)
(2,1)
A
(0,0)
B'
(1,1)
C
(1,1)
D"
(0,0)
Z
(0,0)
B"
(1,1)
(1,1)
D'
(4,1)
(1,1)
(1,1)
(0,0)
(0,0)
(1,1)
(3,3)
E
F'
(4,1)
(0,0)
(3,3)
F"

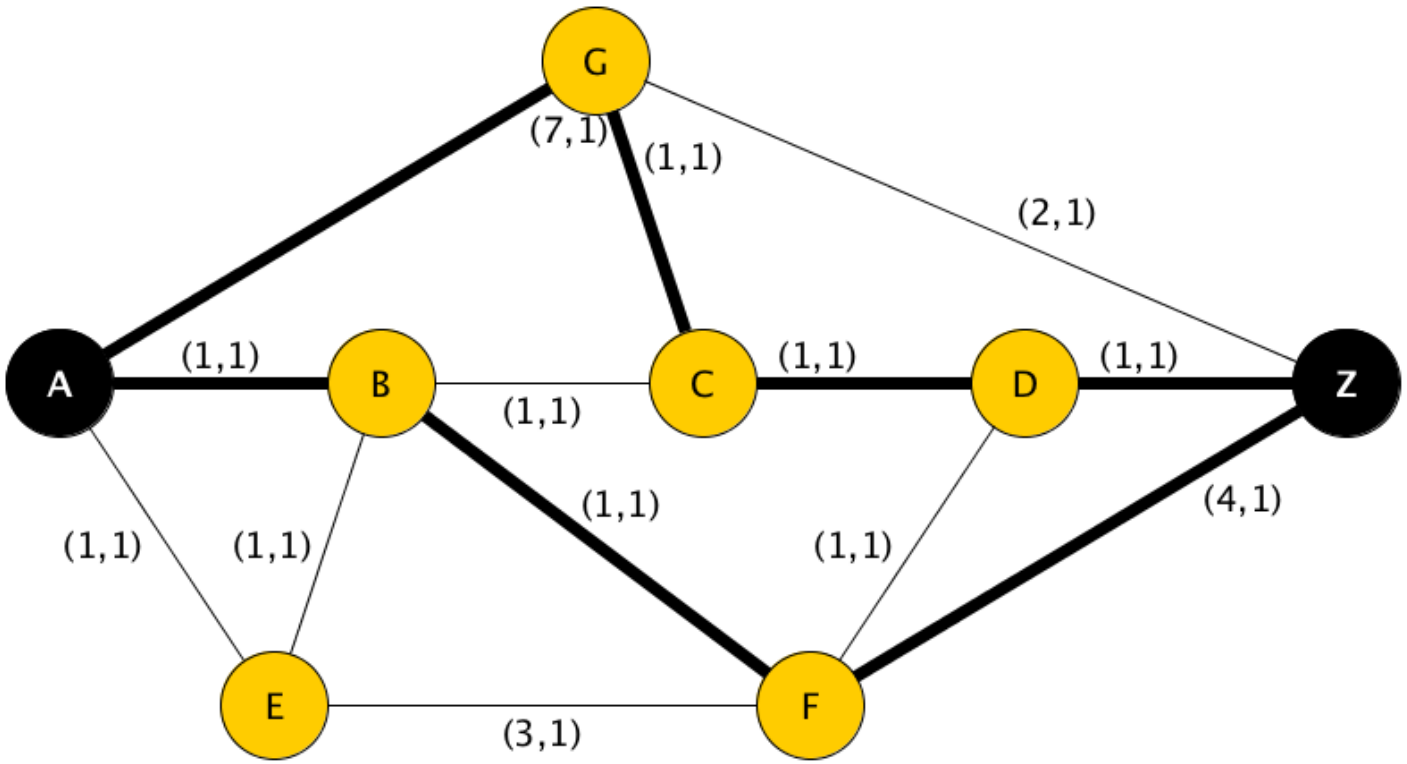
Ejemplo



Ejemplo



Ejemplo



Costo = 16

Fase de búsqueda local

- Busca optimizar la solución encontrada en la fase anterior
- Se recorre una estructura de vecindad en busca de mejores soluciones.

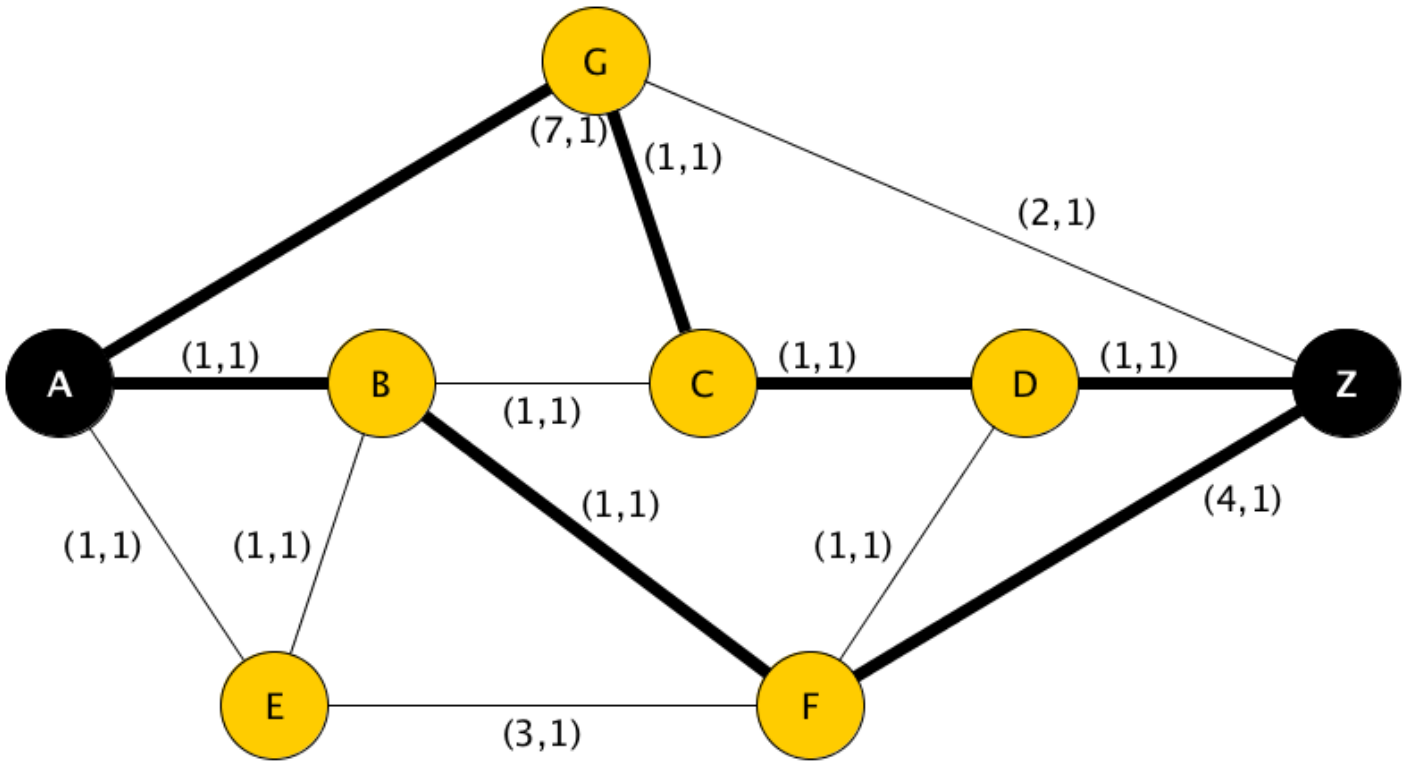
Estructura de vecindad

- $N : \mathbb{S} \times \mathbb{N}^3 \rightarrow 2^{\mathbb{S}}$
- $N(S, n, m, d) = \{(S \setminus P) \cup P' : P \in S \wedge P' \in sub_{n,m,d}(P)\}$
- $sub_{n,m,d}(P)$ es el conjunto de caminos de largo menor o igual a d que se obtiene de sustituir un subcamino sp_n de P, de largo menor o igual a n, por otro, sp_m, de largo menor o igual a m.
- donde $d \geqslant m \geqslant n$

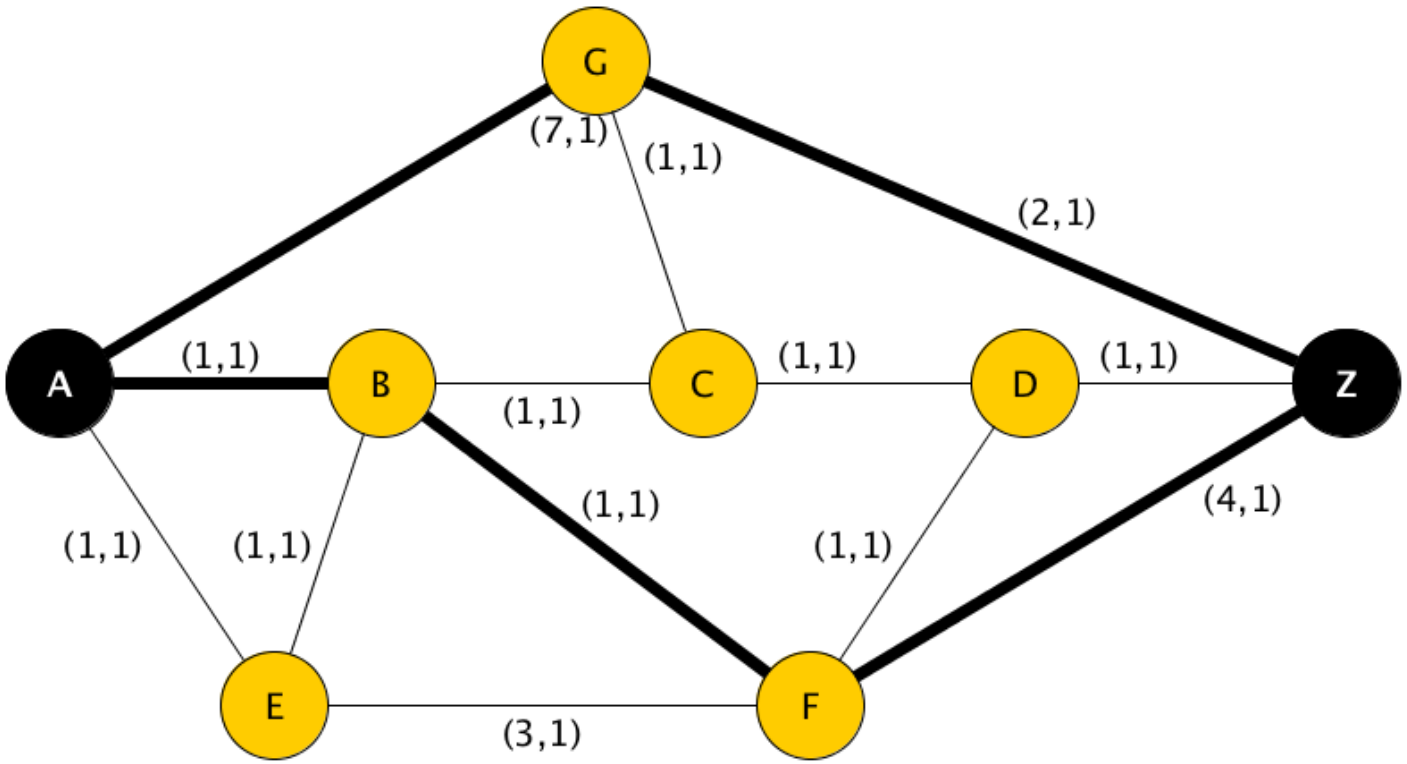
Estrategia de búsqueda

- El algoritmo de búsqueda local aplica mejoras hasta que alcanza un óptimo local
- Falta definir cómo se selecciona un vecino:
  - al comienzo de la búsqueda local se define una permutacion *perm* de los números de 1 al k
  - en el paso $n.k + i$ se selecciona el camino $P = S_{perm(i)}$
  - sobre P se aplica la mejor sustitución posible
  - luego de recorrer todos los caminos, se define una nueva permutacion

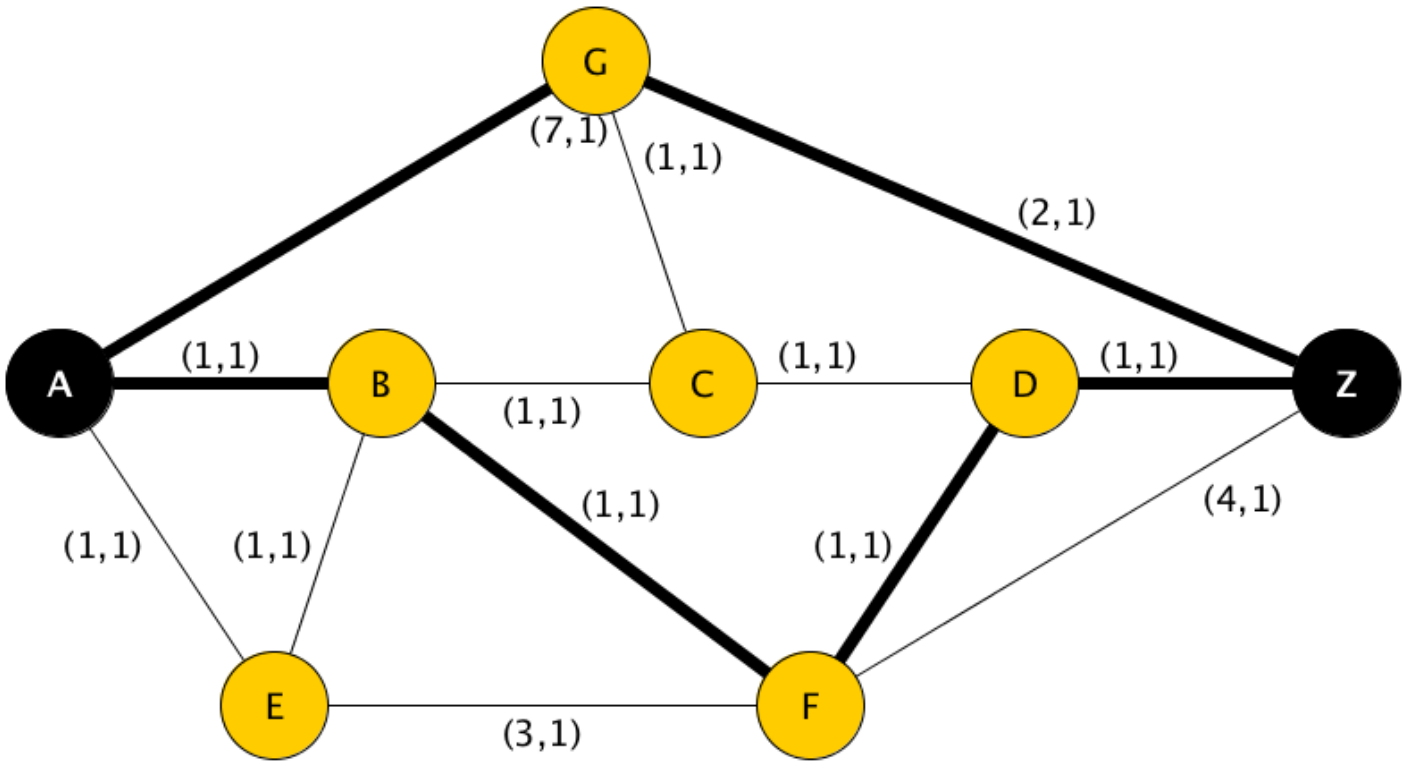
Ejemplo



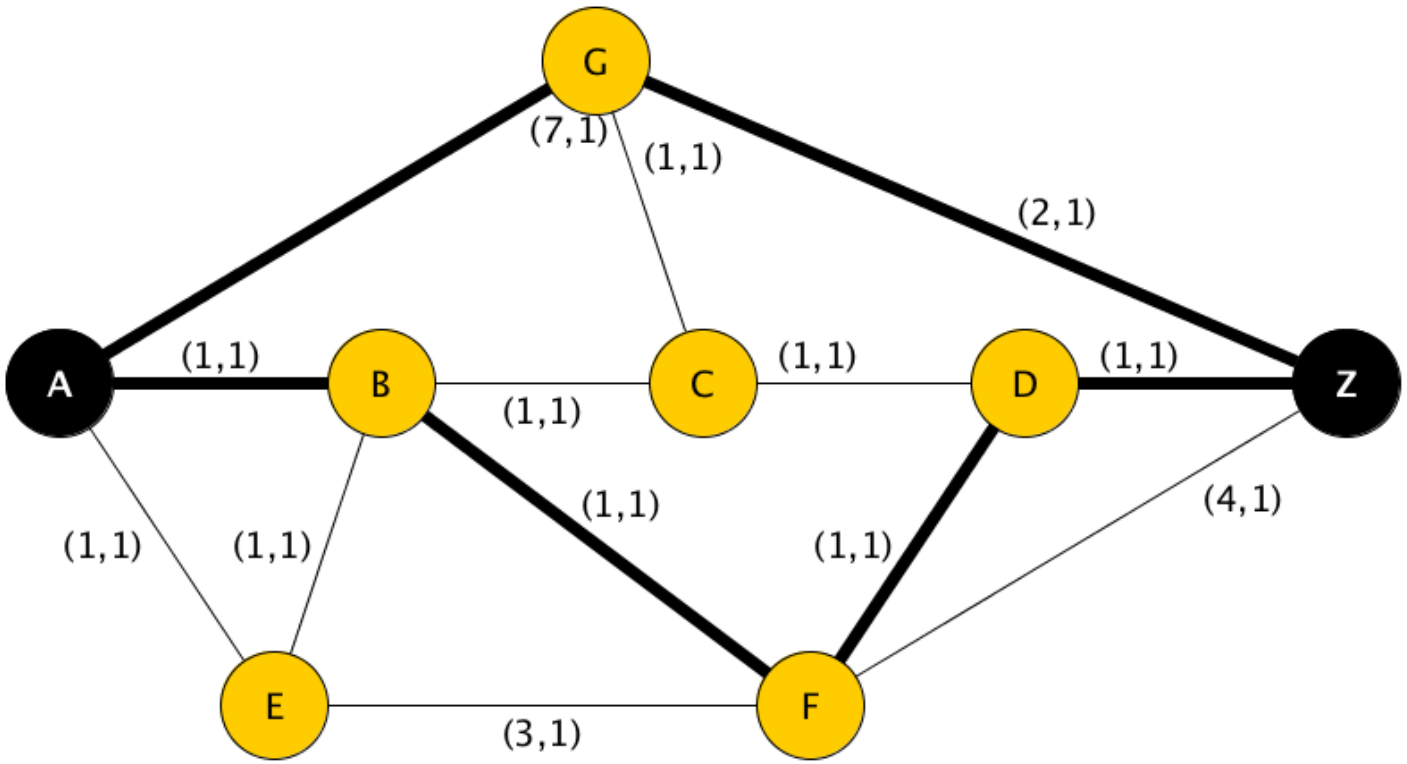
Ejemplo



Ejemplo



Ejemplo



Costo = 13

Organización de la Presentación

Datos

Se utilizaron 3 grafos del 9º concurso de DIMACS:

Random4-n.10.0 grafo aleatorio de 1024 nodos y 4096 aristas

Random4-n.16.0 grafo aleatorio de 65536 nodos y 262144 aristas

USA-road-d.NY mapa de Nueva York de 264346 nodos y 733846 aristas

Datos

Se utilizaron 3 grafos del 9º concurso de DIMACS:

Random4-n.10.0 grafo aleatorio de 1024 nodos y 4096 aristas

Random4-n.16.0 grafo aleatorio de 65536 nodos y 262144 aristas

USA-road-d.NY mapa de Nueva York de 264346 nodos y 733846 aristas

Preguntas

¿Qué preguntas intentamos responder al realizar las pruebas?

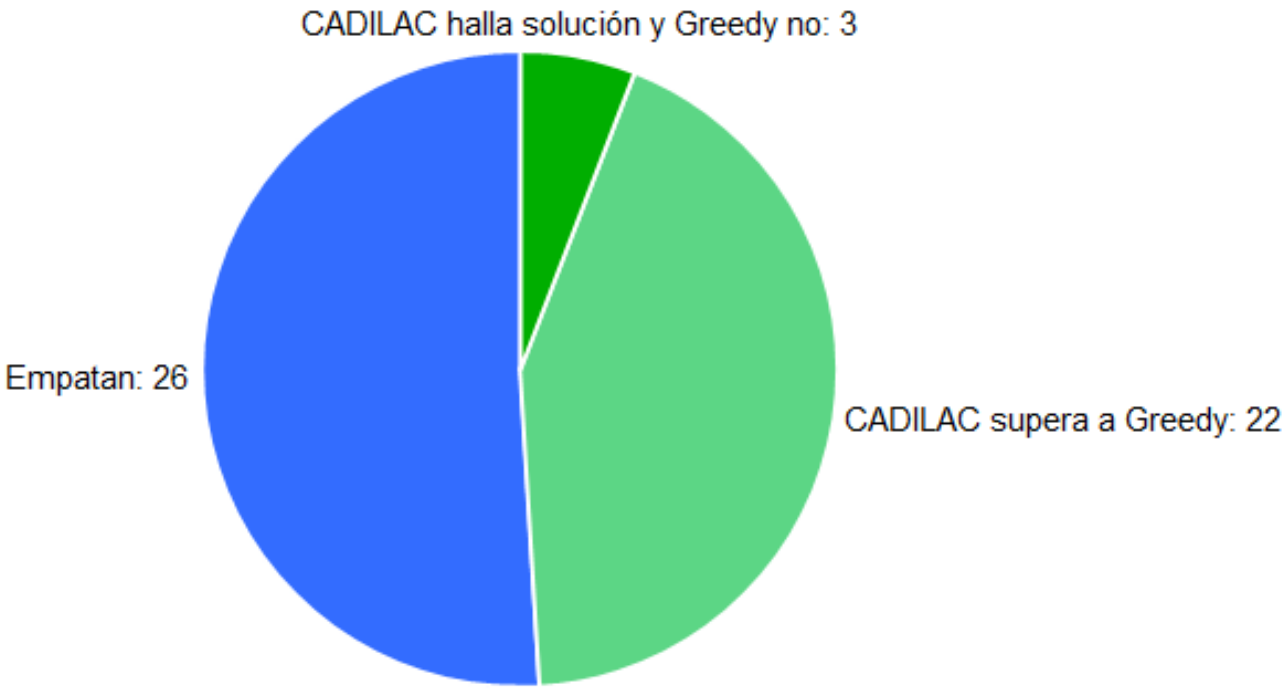
- ¿Es CADILAC capaz de hallar mejores soluciones que Greedy?
- ¿Puede CADILAC hallar soluciones cuando Greedy no?
- ¿Existen casos en donde Greedy supera a CADILAC?
- ¿Cuánto más lento es CADILAC con respecto a Greedy?

Preguntas

¿Qué preguntas intentamos responder al realizar las pruebas?

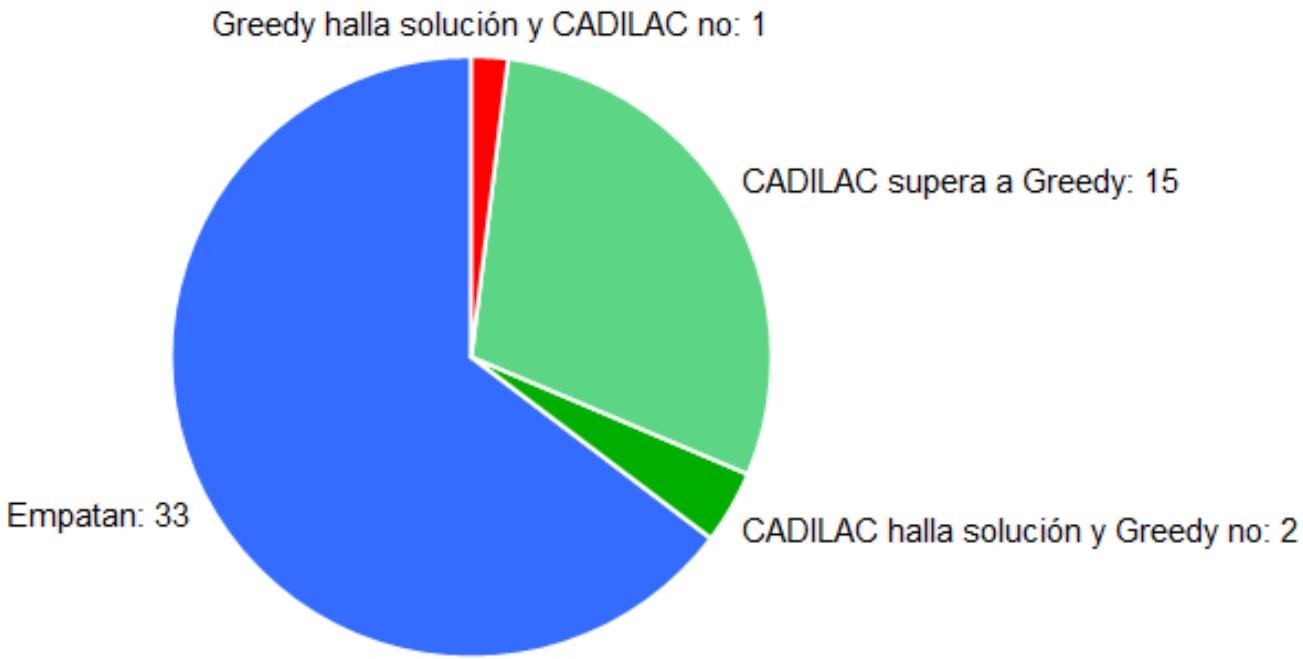
- ¿Es CADILAC capaz de hallar mejores soluciones que Greedy?
- ¿Puede CADILAC hallar soluciones cuando Greedy no?
- ¿Existen casos en donde Greedy supera a CADILAC?
- ¿Cuánto más lento es CADILAC con respecto a Greedy?

Resultados Random4-n.10.0



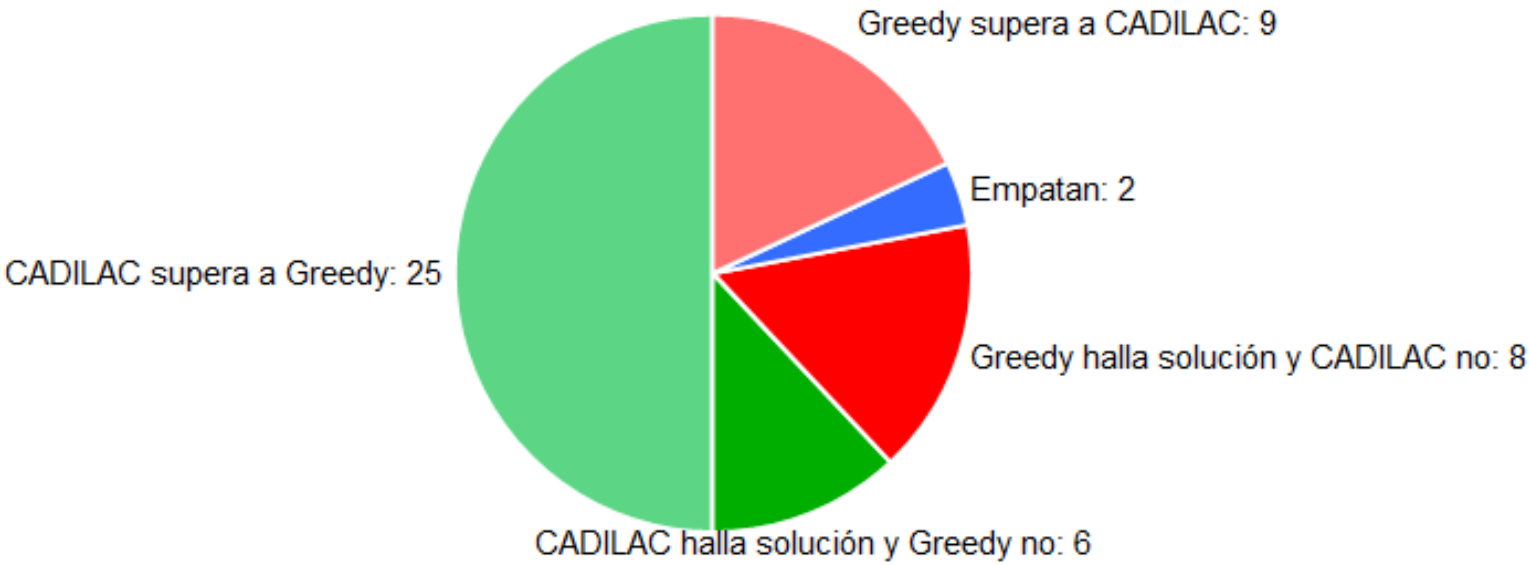
- Ejecuciones totales: 51
- Tiempos de ejecución: CADILAC: 2,14 s | Greedy: 0,0015 s

Resultados Random4-n.16.0



- Ejecuciones totales: 51
- Tiempos de ejecución: CADILAC: 65 s | Greedy: 0,156 s

Resultados USA-road-d.NY



- Ejecuciones totales: 50
- Tiempos de ejecución: CADILAC: 102 s | Greedy: 3,39 s

Ejemplo de resultado

	Suurballe			Heurística Greedy			Heurística CADILAC		
d	costo	mayor largo	tiempo	costo	mayor largo	tiempo	costo	mayor largo	tiempo
11	11107	11	0.00606	11609	10	0.00276	11107	11	1.76269
9	-	-	-	12150	9	0.00245	11544	9	1.77847
7	-	-	-	11663	7	0.00108	11663	7	1.49335
6	-	-	-	-	-	-	13595	6	0.83802

- Grafo = Random4-n.10.0
- $s = 63$
- $t = 427$
- $k = 5$

Conclusiones

- CADILAC puede hallar soluciones donde Greedy no es capaz
- También puede hallar mejores soluciones que Greedy
- No es suficientemente robusto, requiere mejoras en la construcción
- Es más lento que Greedy

Conclusiones

- CADILAC puede hallar soluciones donde Greedy no es capaz
- También puede hallar mejores soluciones que Greedy
- No es suficientemente robusto, requiere mejoras en la construcción
- Es más lento que Greedy

Conclusiones

- CADILAC puede hallar soluciones donde Greedy no es capaz
- También puede hallar mejores soluciones que Greedy
- No es suficientemente robusto, requiere mejoras en la construcción
- Es más lento que Greedy

Conclusiones

- CADILAC puede hallar soluciones donde Greedy no es capaz
- También puede hallar mejores soluciones que Greedy
- No es suficientemente robusto, requiere mejoras en la construcción
- Es más lento que Greedy

Trabajo futuro

- Cambiar el algoritmo de construcción para hacerlo más robusto (i.e. basado en Greedy en lugar de Bhandari)
- Mejorar la búsqueda local con una metaheurística como *Simulated Annealing*.
- Añadir *Path-Relinking* para mejorar el rendimiento de la heurística
- Complementar con el algoritmo de Suurballe o Bhandari

Trabajo futuro

- Cambiar el algoritmo de construcción para hacerlo más robusto (i.e. basado en Greedy en lugar de Bhandari)
- Mejorar la búsqueda local con una metaheurística como *Simulated Annealing*.
- Añadir *Path-Relinking* para mejorar el rendimiento de la heurística
- Complementar con el algoritmo de Suurballe o Bhandari

Trabajo futuro

- Cambiar el algoritmo de construcción para hacerlo más robusto (i.e. basado en Greedy en lugar de Bhandari)
- Mejorar la búsqueda local con una metaheurística como *Simulated Annealing*.
- Añadir *Path-Relinking* para mejorar el rendimiento de la heurística
- Complementar con el algoritmo de Suurballe o Bhandari

Trabajo futuro

- Cambiar el algoritmo de construcción para hacerlo más robusto (i.e. basado en Greedy en lugar de Bhandari)
- Mejorar la búsqueda local con una metaheurística como *Simulated Annealing*.
- Añadir *Path-Relinking* para mejorar el rendimiento de la heurística
- Complementar con el algoritmo de Suurballe o Bhandari

Fin