Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Star Routing: Entre Ruteo de Vehículos y Vertex Cover

Tesis presentada para optar al título de
Licenciado en Ciencias de la Computación

Guido Tagliavini Ponce

Director: Diego Delle Donne
Codirector: Javier Marenco
Buenos Aires, 2017

Star Routing: Entre Ruteo de Vehículos y Vertex Cover

En esta tesis consideramos el problema STAR ROUTING (abreviado SR) que toma un grafo simple y no dirigido G y un subconjunto de aristas X, y pide encontrar un camino P de G tal que toda arista de X tiene al menos un extremo en P, de longitud mínima.

Estudiamos la complejidad computacional del problema. Probamos que el problema es **NP-completo** en el caso general, restringido a grafos grillas (asumiendo una representación no estándar de G) y restringido a grafos completos. Probamos que en el caso de los árboles el problema está en **P** y damos un algoritmo de tiempo lineal que lo resuelve en ese caso.

Exhibimos dos algoritmos exactos junto con heurísticas para acelerar el cómputo. La importancia de estos algoritmos es principalmente teórica, pues los resultados experimentales muestran que no son suficientemente rápidos como para resolver instancias de tamaño real, en una cantidad de tiempo razonable.

Exhibimos algoritmos aproximados para el problema en su versión general, y restringido a grillas y a grafos completos. En particular, concluimos que el caso general se puede aproximar por un factor constante del óptimo. Para grafos grilla damos un algoritmo $(9/2)$-aproximado, y para grafos completos damos, para todo $\varepsilon > 0$, un algoritmo $(2+\varepsilon)$-aproximado.

Además de estudiar el problema SR, consideramos un problema asociado denominado STOPS SELECTION (abreviado SS), que toma una instancia (G, X) de SR y un camino P que es solución factible de SR para (G, X), y pide encontrar un mínimo subconjunto S de vértices de P tal que toda arista de X tiene al menos un extremo en S. Probamos que este problema es **NP-completo** en el caso general. Damos un algoritmo exacto, que resulta ser polinomial para grafos bipartitos. Damos además un algoritmo 2-aproximado. Tanto el algoritmo exacto como el algoritmo aproximado son reducciones al problema de vertex cover mínimo.

Hasta donde sabemos, ni el problema SR ni el problema SS han sido estudiados en la literatura previamente.

Palabras claves: ruteo de vehículos, vertex cover, complejidad computacional, algoritmos exactos, algoritmos aproximados.

Star Routing: Between Vehicle Routing and Vertex Cover

In this thesis we consider the STAR ROUTING problem (abbreviated as SR) which takes a simple undirected graph G and a subset of edges X, and asks to find a path P of G such that every edge in X has at least one endpoint in P, with minimum length.

We study the problem's computational complexity. We prove the problem is **NP-complete** in the general case, restricted to grid graphs (assuming a non-standard representation for G) and restricted to complete graphs. We prove that for trees the problem is in **P** and we give a linear-time algorithm that solves it in that case.

We give two exact algorithms along with heuristics to speed up the computation. The importance of these algorithms is mainly theoretical, since experimental results show they are not fast enough to solve real-life instances, in a reasonable amount of time.

We give approximation algorithms for the problem in the general case, and restricted to grid and complete graphs. In particular, we conclude the general case can be approximated within a constant factor of the optimal. For grid graphs we give a $(9/2)$-approximation algorithm, and for complete graphs we give, for all $\varepsilon > 0$, a $(2+\varepsilon)$-approximation algorithm.

Besides studying the SR problem, we consider a related problem called STOPS SELECTION (abbreviated as SS), which takes an instance (G, X) of SR and a path P that is a feasible solution of SR for (G, X), and asks to find a minimum subset S of vertices of P such that every edge in X has at least one endpoint in S. We prove this problem is **NP-complete** in the general case. We give an exact algorithm, that turns out to be polynomial for bipartite graphs. We also give a 2-approximation algorithm. Both the exact algorithm and the approximation algorithm are reductions to the minimum vertex cover problem.

To the best of our knowledge, neither the SR problem nor the SS problem have been studied in the literature.

Keywords: vehicle routing, vertex cover, computational complexity, exact algorithms, approximation algorithms.

Agradecimientos

Índice general

Capítulo 1

Introducción

1.1. Los problemas

Supongamos que estamos a cargo de la distribución de la edición física de un importante diario en una gran ciudad, para clientes suscriptos. Las entregas se hacen al domicilio de cada cliente, utilizando camiones de reparto, como el que se muestra en la Figura 1.1. Estos camiones circulan por la ciudad, recorriendo, uno por uno, los domicilios a los que se debe hacer una entrega.

Durante el procedimiento de reparto usual, un camión se detiene en la puerta de un domicilio y el conductor desciende del mismo, retira tantos diarios como haya que entregar allí, y hace entrega de los mismos. Luego, vuelve a subirse al camión, y conduce hacia la puerta del siguiente domicilio.

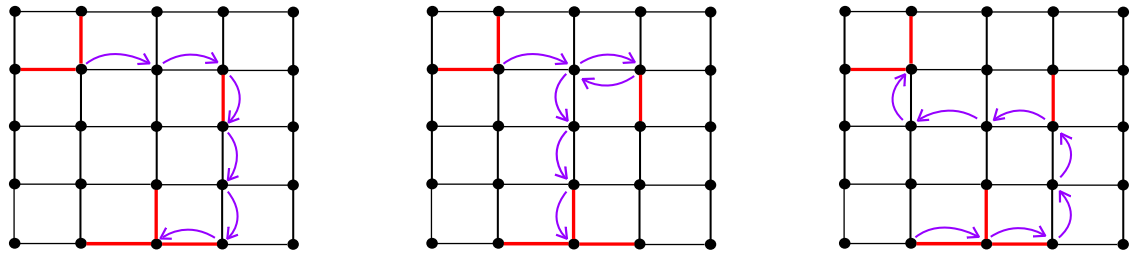
Figura 1.1: Camión de reparto del diario *The Boston Globe*.

Una alternativa a esto es un sistema de reparto en el que un camión no se detiene en la puerta de cada domicilio, sino en las esquinas de las calles de los clientes. Al detenerse en una esquina, el conductor desciende del camión, y realiza las entregas a todos los clientes que están sobre cada una de las cuadras incidentes a esa esquina (que suelen ser 4), a pié. Este sistema reduce el recorrido realizado por el camión, y lo compensa con recorrido a pié. Por lo tanto, es especialmente conveniente cuando hay una alta densidad de tránsito vehicular. También es potencialmente mejor que el sistema tradicional en zonas donde hay una alta concentración de clientes, aún si hay poco tránsito, porque permite realizar varias entregas, haciendo una única descarga de diarios del camión. Notar que este sistema puede ser utilizado en otros contextos que involucren el reparto de productos, como por ejemplo la distribución de encomiendas y paquetes, que realizan empresas como FedEx o UPS.

Como en toda empresa, queremos optimizar costos. En particular, queremos minimizar el recorrido que hace un camión, para realizar todas sus entregas. Como un primer paso en el estudio de este problema, nos vamos a concentrar en el reparto que debe realizar un único camión. El problema que queremos resolver toma un conjunto de domicilios en

los que debemos entregar ejemplares del diario, busca calcular un camino que le permita realizar todas las entregas a un camión (utilizando el sistema de reparto de detenciones en las esquinas), que recorra la mínima cantidad de cuadras posible.

Podemos modelar este problema con un grafo simple y no dirigido G, que representa la ciudad, y un subconjunto X de aristas de G, que representa aquellas cuadras en las que están ubicados los clientes. A cada arista $e \in X$ la llamaremos *cliente*. Si bien podría ocurrir que haya varios clientes en una misma cuadra, en este trabajo sólo nos importa la existencia o no de clientes, y no la cantidad, en una cuadra. El problema STAR ROUTING (abreviado SR) consiste en encontrar un camino de G, tal que cada cliente tiene alguno de sus extremos en el camino, de longitud mínima. Esto es, el camino pasa, en algún momento, por alguno de los cruces adyacentes a la cuadra de cada uno de los domicilios. La Figura 1.2 muestra tres soluciones factibles distintas para una instancia de SR. En ese caso, el grafo de entrada es una grilla. Esta clase de instancias es de especial interés, pues permite modelar adecuadamente la topología de una ciudad.



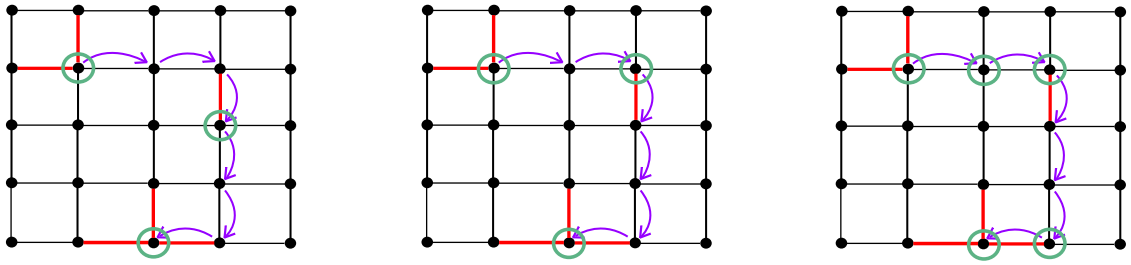
(a) (b) (c)

Figura 1.2: Soluciones factibles de SR. (a) y (b) son óptimas; (c) no es óptima. Las aristas en rojo son clientes, y las flechas violetas indican la solución.

Si una solución factible visita uno de los extremos de un cliente e, el camión de entrega podrá, si así lo desea, detenerse en ese punto y realizar la entrega de productos a ese (y posiblemente otros) cliente. Sin embargo, una solución de SR no establece en qué esquinas debe detenerse el camión para realizar todas las entregas. Lo único que nos garantiza, es que existe alguna forma de elegir las detenciones, que permita realizar todas las entregas. En particular, podríamos detenernos en cada esquina del camino, y entregar a todos los clientes restantes. Esta selección de paradas podría obligarnos a detener más veces de lo necesario, incrementando el tiempo total que insume el reparto, además de aumentar el desgaste del camión, producto de las reiteradas puestas en marcha del mismo.

Esto motiva un problema asociado a SR, que llamamos STOPS SELECTION (abreviado SS). Este problema toma una instancia de SR y una solución factible para la misma, encontrar un subconjunto de vértices del camino, tal que todo cliente tiene un extremo en el conjunto, de cardinal mínimo. La Figura 1.3 muestra tres soluciones factibles de SS para la instancia de SR de la Figura 1.2-(a).

Los problemas SR y SS son el tema de estudio de este trabajo, aunque, como adelanta el título de la tesis, SR es el objeto de estudio principal de la tesis, y es profundamente más complejo que SS.



(a) (b) (c)

Figura 1.3: Soluciones factibles de SS. (a) y (b) son óptimas; (c) no es óptima. Las circunferencias verdes indican la solución.

1.2. Definiciones preliminares y notación

1.2.1. Métricas

Sea X un conjunto. Se dice que una función $d : X \times X \to \mathbb{R}$ es una *métrica* o *distancia*, si para todo $a, b, c \in X$ cumple:

1. $d(a, b) \geq 0$ (no negatividad)
2. $d(a, b) = d(b, a)$ (simetría)
3. $d(a, b) \leq d(a, c) + d(c, b)$ (desigualdad triangular)
4. $d(a, b) = 0$ si y sólo si $a = b$.

Sobre el plano euclídeo $\mathbb{R}^2$ se definen varias métricas frecuentemente utilizadas, dos de las cuales son la *distancia Manhattan* y la *distancia euclídea*. Si $p = (p_1, p_2), q = (q_1, q_2)$, la distancia Manhattan entre p y q es $d_1(p, q) := |p_1 - q_1| + |p_2 - q_2|$, y la distancia euclídea entre p y q es $d_2(p, q) := \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2}$.

1.2.2. Teoría de grafos

Revisamos los conceptos fundamentales de teoría de grafos que usaremos a lo largo de la tesis. Un tratamiento extensivo sobre teoría de grafos básica puede encontrarse en [11].

Grafos, adyacencias y grados

Un *grafo simple y no dirigido* es una tupla $G = (V, E)$, tal que V es un conjunto finito de elementos llamados *vértices*, y $E \subseteq \{\{u, v\} : u, v \in V, u \neq v\}$ es un conjunto de pares no ordenados de vértices, llamados *aristas*. Es *no dirigido* porque las aristas no tienen dirección, y *simple* porque no hay múltiples aristas entre un mismo par de vértices, ni aristas con sus dos extremos en el mismo vértice. Para abreviar, diremos simplemente *grafo*. En ocasiones, llamamos $V(G)$ y $E(G)$ a los conjuntos de vértices y aristas de G, respectivamente. Dada una arista $\{u, v\}$, sus *extremos* son los vértices u y v que la componen.

En un grafo $G = (V, E)$, dos vértices $u, v \in V$ se dicen *adyacentes* si $\{u, v\} \in E$. La *vecindad* de un vértice $u \in V$ es el conjunto $N(u) = \{v \in V : u \text{ y } v \text{ son adyacentes}\}$. El

grado de un vértice $u \in V$ es $d(u) = |N(u)|$. Los grados de los vértices de un grafo cumplen el siguiente resultado, conocido como *Handshake Lemma*.

Lema 1.1. $\sum_{u \in V} d(u) = 2|E|$

Pesos

Llamamos *pesos* de un grafo $G = (V, E)$ a una función $c : E \to \mathbb{Z}_{\geq 0}$. Por simplicidad, si $\{u, v\} \in E$, escribimos $c(u, v)$ ó $c(v, u)$, en lugar de $c(\{u, v\})$. Una función $c : V \times V \to \mathbb{R}$ que sea *simétrica*, puede hacer las veces de pesos de G.

Caminos y circuitos

Un *camino* de un grafo es una secuencia $\langle u_1, \ldots, u_r \rangle$ de vértices del grafo, tal que u_i y u_{i+1} son adyacentes para todo $1 \leq i < r$. Un camino se dice *simple* si no pasa más de una vez por cada vértice.

Un *circuito* de un grafo es un camino que empieza y termina en el mismo vértice. Un circuito se dice *simple* si contiene 3 o más vértices distintos, y no pasa más de una vez por cada uno, salvo por el vértice inicial y final.

Dado un camino o circuito P de un grafo, la *longitud de* P, notada $\text{length}(P)$, es la cantidad de aristas de P. A la cantidad de vértices de P, visto como multiconjunto, la notamos $|P|$. Se tiene, por lo tanto, $\text{length}(P) = |P| - 1$. Si la función c son los pesos del grafo, llamamos *peso de* P, y lo notamos $\text{length}_c(P)$, a la suma de los pesos, dados por c, de las aristas de P.

Un *camino hamiltoniano* (resp. *circuito hamiltoniano*) de un grafo, es un camino (resp. circuito) que pasa exactamente una vez por cada vértice del grafo. Un *camino euleriano* (resp. *circuito euleriano*) de un grafo, es un camino (resp. circuito) que pasa exactamente una vez por cada arista.

Grafos conexos y distancia

Un grafo es *conexo*, si para cada par de vértices del grafo existe un camino entre ellos.

Dado un grafo conexo G, la *distancia* entre dos vértices u y v de G es $\text{dist}_G(u, v) := \text{mín}\{\text{length}(P) : P \text{ es un camino entre } u \text{ y } v \text{ en } G\}$. Cuando no haya ambigüedad sobre el grafo al que nos estamos refiriendo, escribiremos directamente $\text{dist}(u, v)$. Notar que la distancia dist_G es una métrica, en el sentido matemático.

Subgrafos y subgrafos generadores

Dado un grafo $G = (V, E)$, un *subgrafo* de G es un grafo $H = (V', E')$ tal que $V' \subseteq V$ y $E' \subseteq E$. Un *subgrafo generador* de un grafo G es un subgrafo H tal que todo vértice de G también es un vértice de H.

Árboles y árboles generadores

Un *árbol* es un grafo conexo sin circuitos simples. Un *árbol generador* es un subgrafo generador que además es un árbol. Si T es un árbol generador de un grafo G con pesos c, notamos $\text{weight}_c(T)$ a la suma de los pesos, dados por c, de las aristas de T. Un *árbol generador mínimo* es un árbol generador que minimiza weight_c, entre todos los árboles

generadores del grafo. Notamos $\mathrm{MST}(G, c)$ al peso de un árbol generador mínimo del grafo G con pesos dados por c.

Árboles con raíz

Un *árbol con raíz* es un árbol en el que se ha distinguido un vértice, al que llamamos *raíz*. Un árbol con raíz induce un ordenamiento jerárquico de sus vértices, que es una clasificación por nivel tal que los vértices del nivel k son aquellos nodos a distancia k de la raíz.

Subgrafos inducidos por aristas

Dado un grafo $G = (V, E)$ y un subconjunto de aristas $F \subseteq E$, el subgrafo inducido de G por F, notado $G[F]$, es el subgrafo de G que contiene, exactamente, las aristas de F y aquellos vértices que sean extremos de dichas aristas.

Grafos completos

Un grafo es *completo*, si todo par de vértices del grafo son adyacentes.

Dado un conjunto finito A, notamos $K(A)$ al grafo completo que tiene como conjunto de vértices a A.

Grafos bipartitos

Un grafo $G = (V, E)$ es *bipartito*, si existe una partición $\{V_1, V_2\}$ de V, tal que toda arista de G tiene un extremo en V_1 y otro en V_2.

Grafos grilla

Un grafo es *grilla*, si el conjunto de vértices y aristas de G forman una grilla[1]. En la Figura 1.4 se exhibe un grafo grilla de n vértices de alto por m vértices de ancho. Observar que todo grafo grilla es un grafo bipartito. Una bipartición (la única posible) se obtiene tomando los vértices por diagonales, en forma alternada.

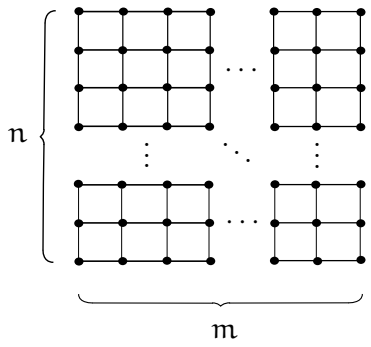


Figura 1.4: Un grafo grilla de n filas y m columnas.

[1]Se puede definir grafo grilla en forma más rigurosa, como un producto cartesiano de grafos. A los fines de esta tesis, alcanza con una definición informal.

Vertex covers y matchings

Dado un grafo G, una arista e de G y un subconjunto W de vértices (o un camino) de G, decimos que W *cubre a* e si e tiene algún extremo en W.

Un *vertex cover* de G es un conjunto S de vértices de G tal que S cubre a cada arista de E. Notamos $\tau(G)$ al mínimo cardinal de un vertex cover de G.

Un *matching* de G es un conjunto M de aristas de G tal que no hay dos aristas de M que tengan un extremo en común. Notamos $\nu(G)$ al máximo cardinal de un matching de G.

Estos dos conceptos están vinculados, como muestra el siguiente teorema.

Teorema 1.1. *En todo grafo* G*, vale que el máximo cardinal de un matching de* G *es menor o igual al mínimo cardinal de un vertex cover de* G*. Esto es,* $\nu(G) \leq \tau(G)$*.*

Demostración. Sea $M = \{e_1, \ldots, e_r\}$ un matching de G, y S un vertex cover de G. Cada arista de M debe ser cubierta por S, es decir que cada e_i tiene al menos un extremo en S. Por ser M un matching, sus aristas son disjuntas en sus extremos, dos a dos. Luego, S contiene al menos $r = |M|$ vértices. Esto prueba que $|M| \leq |S|$, lo cual vale, en particular, si M es máximo y S es mínimo. □

Una consecuencia de este teorema es que dado un grafo G, podemos dar una cota inferior de $\tau(G)$ exhibiendo un matching de G, y podemos dar una cota superior de $\nu(G)$ exhibiendo un vertex cover de G. En el caso de grafos bipartitos $\nu(G)$ y $\tau(G)$ coinciden, como probó Kőnig en 1931.

Teorema 1.2 (Kőnig)**.** *Si* G *es un grafo bipartito, entonces* $\tau(G) = \nu(G)$*.*

1.2.3. Problemas de decisión y optimización

Un *problema de optimización* es aquel en el que cada instancia tiene asociada un conjunto S de *soluciones factibles*, cada una con un valor numérico asociado, y el objetivo es encontrar una solución factible cuyo valor sea *óptimo* (mínimo o máximo, según el problema). Una solución factible cuyo valor es óptimo se denomina *solución óptima*. El valor asociado a cada solución está dado por una función $\text{val} : S \to \mathbb{R}$, llamada *función objetivo*. A modo de ejemplo, consideremos el famoso TRAVELING SALESMAN PROBLEM (abreviado TSP).

TSP (optimización)
INSTANCIA: $G = (V, E)$ un grafo completo y $c : E \to \mathbb{Z}_{\geq 0}$ los pesos de G.
SALIDA: Un circuito hamiltoniano de G, de peso mínimo.

En este caso, el conjunto de soluciones factibles es el de todos los circuitos hamiltonianos de G, y la función objetivo es length_c.

Si I es una instancia de un problema de optimización Π, notamos $\Pi^*(I)$ al valor de la función objetivo de una solución óptima. Cuando el contexto no de lugar a ambigüedad, notaremos $\text{OPT} = \Pi^*(I)$.

Un *problema de decisión* es aquel en el que cada instancia sólo tiene dos respuestas posibles: SI o NO. Retomando el ejemplo del TSP, su versión de decisión es la siguiente.

TSP (decisión)
INSTANCIA: $G = (V, E)$ un grafo completo, $c : E \to \mathbb{Z}_{\geq 0}$ los pesos de G y $k \in \mathbb{Z}$.
SALIDA: ¿Existe un circuito hamiltoniano de G, de peso k o menos?

Un *problema de optimización combinatoria* es un problema de optimización en el que el conjunto de soluciones factibles S es *discreto*. El TSP es un problema de optimización combinatoria, pues los elementos involucrados son discretos. Como muestran las dos versiones exhibidas de TSP, un problema de optimización combinatoria se puede presentar tanto en forma de problema de optimización como de problema de decisión. En general, la forma de traducir la versión de optimización en la versión de decisión, es agregando una "cota de optimización", que en el caso de TSP es el argumento k. A primera vista, la versión de optimización puede parecer más fuerte que la de decisión, pues la primera encuentra una solución óptima, mientras que la segunda sólo indica si el valor de una solución óptima está por debajo de cierta cota. Sin embargo, estas dos versiones muchas veces son equivalentes, en el sentido de que si sabemos resolver una en tiempo polinomial, entonces podemos resolver la otra en tiempo polinomial.

La reducción de decisión a optimización es simple, y la ilustramos a través del TSP. Supongamos que tenemos un algoritmo polinomial $\mathcal{A}$ que resuelve la versión de optimización de TSP. Dada una instancia (G, c, k) de la versión de decisión, primero calculamos una solución óptima $\mathcal{A}(G, c)$. Llamemos k_0 al valor de esta solución óptima. Si $k < k_0$ la respuesta es NO, y si $k \geq k_0$ la respuesta es SI.

Recíprocamente, es posible reducir la versión de optimización de TSP a la de decisión, aunque la transformación es un poco más complicada, y no lo haremos aquí.

En este trabajo utilizaremos la variante de TSP que consiste en encontrar un *camino* hamiltoniano a la cual denominamos PATH TSP (abreviado PTSP).

PATH TSP (optimización)
INSTANCIA: $G = (V, E)$ un grafo completo, $c : E \to \mathbb{Z}_{\geq 0}$ los pesos de G.
SALIDA: Un camino hamiltoniano de G, de peso mínimo.

PATH TSP (decisión)
INSTANCIA: $G = (V, E)$ un grafo completo y $c : E \to \mathbb{Z}_{\geq 0}$ los pesos de G y $k \in \mathbb{Z}_{\geq 0}$.
SALIDA: ¿Existe un camino hamiltoniano de G, de peso menor o igual a k?

Si se requiere que la función c satisfaga la desigualdad triangular, el problema se denomina PTSP *métrico*. Si se requiere que los vértices de G sean puntos del plano, de coordenadas enteras, y c sea la distancia Manhattan entre ellos, el problema se denomina PTSP *rectilíneo*. Análogamente, cuando c es la distancia euclídea, el problema se denomina PTSP *euclídeo*.

1.2.4. Teoría de complejidad computacional

Hacia la década de 1970, había una gran cantidad de problemas computacionales para los que no se conocía ningún algoritmo eficiente que los resuelva. La teoría de complejidad computacional surge en un intento de clasificar problemas según su dificultad, a través de un criterio formal. Esta teoría se concentra principalmente en los problemas decisión, puesto que su estructura es más simple que otros tipos de problemas, como los de optimización. Dado que muchas veces es posible reducir la versión de optimización de un

problema a su versión de decisión, la teoría de complejidad también permite hablar de la dificultad de algunos problemas de optimización.

Daremos una breve introducción, relativamente informal, a esta teoría. En [9] se puede encontrar una maravillosa descripción de la misma. Un problema de decisión Π puede pensarse como una clasificación de instancias de entrada en dos conjuntos: instancias para las que la respuesta es SI (o sencillamente, *instancias de SI*) e instancias para las que la respuesta es NO (*instancias de NO*). Denotamos Y_Π al conjunto de instancias de SI.

Decimos que un algoritmo $\mathcal{A}$ *acepta* una entrada x si $\mathcal{A}(x)$ devuelve SI, y que la *rechaza* si devuelve NO. Notar que un algoritmo $\mathcal{A}$ podría no aceptar una entrada x, pero tampoco rechazarla, pues podría iterar infinitamente sin emitir una respuesta. El conjunto de entradas que acepta un algoritmo $\mathcal{A}$ es $L_\mathcal{A} = \{x : \mathcal{A} \text{ acepta } x\}$.

Dada una instancia x de un problema de decisión, llamamos $\text{size}(x)$ al tamaño de x. Formalmente, $\text{size}(x)$ es la cantidad de bits que necesitamos para representar x. Esto depende de cómo representemos a x, es decir, el *esquema de representación*. Por ejemplo, un número natural n se puede representar en base 2 (ocupando $\sim \lg n$ bits), o en base unaria (ocupando n bits).

Un algoritmo $\mathcal{A}$ que termina para todas las entradas posibles es *polinomial* si existe una función polinomial $T : \mathbb{N} \to \mathbb{N}$ tal que para cada instancia x de tamaño $n = \text{size}(x)$, $\mathcal{A}(x)$ corre en tiempo $O(T(n))$.

Estamos en condiciones de definir la primera clase importante de problemas, de aquellos que se pueden resolver en tiempo polinomial. Definimos, informalmente,

$$\mathbf{P} = \{\Pi : \text{existe un algoritmo polinomial } \mathcal{A} \text{ tal que } L_\mathcal{A} = Y_\Pi\}$$

Por convención, decimos que los problemas de esta clase son exactamente los problemas *tratables*. Todo problema que no está en **P** se dice *intratable*.

Si no sabemos *resolver* un problema en tiempo polinomial, lo siguiente a lo que podemos aspirar es a *verificar* una respuesta del problema, valiéndonos de una "prueba". Por ejemplo, si el problema consiste en determinar si un número es compuesto, y nos dan como certificado la factorización en primos de ese número, es fácil verificar que el número es compuesto, multiplicando los factores provistos y verificando que el resultado sea igual al número de la entrada.

Llamamos *algoritmo verificador* a un algoritmo $\mathcal{A}$ que toma una instancia de un problema junto con un objeto llamado *certificado*. Un tal algoritmo $\mathcal{A}$ *verifica* una instancia x si existe un certificado y tal que $\mathcal{A}(x, y)$ devuelve SI. El conjunto de instancias que verifica $\mathcal{A}$ es $L_\mathcal{A} = \{x : \mathcal{A} \text{ verifica } x\}$.

Un algoritmo verificador $\mathcal{A}$ es *polinomial no determinista* si existe una función polinomial $T : \mathbb{N} \to \mathbb{N}$ tal que para cada instancia x de tamaño $n = \text{size}(x)$, si y es un certificado tal que $\mathcal{A}(x, y)$ devuelve SI, entonces $\mathcal{A}(x, y)$ corre en tiempo $O(T(n))$. Notar que sólo nos importa el tiempo que toman las ejecuciones que efectivamente verifican instancias.

Definimos la segunda clase importante de problemas, de aquellos que se pueden verificar en tiempo polinomial. Se define

$$\mathbf{NP} = \{\Pi : \text{ existe un algoritmo verificador polinomial no determinista } \mathcal{A} \text{ tal que } L_\mathcal{A} = Y_\Pi\}$$

El nombre **NP** proviene del inglés *non-deterministic polynomial*. Este nombre se debe

a que la formulación original de esta clase de problemas (que es equivalente a la que presentamos aquí) utilizaba un modelo de cómputo que involucraba no determinismo.

Es claro que todo problema que está en **P** también está en **NP**, pues un algoritmo verificador podría descartar el certificado y resolver el problema en tiempo polinomial. Luego, $\mathbf{P} \subseteq \mathbf{NP}$. Se desconoce si esta inclusión es estricta: esta incógnita es el famoso problema "$\mathbf{P} = \mathbf{NP}$". Esta pregunta se puede formular, en lenguaje coloquial, como: ¿la dificultad de verificar la solución a un problema es la misma que la de resolver el problema? La intuición dice que no, y hay consenso académico de que debe ser $\mathbf{P} \neq \mathbf{NP}$, aunque nadie ha podido demostrarlo.

Dados dos problemas de decisión Π_1 y Π_2, una *transformación polinomial* de Π_1 en Π_2 es una función computable en tiempo polinomial f que mapea instancias de Π_1 en instancias de Π_2, tal que $x \in Y_{\Pi_1}$ si y sólo si $f(x) \in Y_{\Pi_2}$. En ese caso, se dice que Π_1 se transforma polinomialmente a Π_2 (o que Π_1 se reduce a Π_2) y se nota $\Pi_1 \leq_P \Pi_2$. En este caso, podemos decir que Π_1 es "no es más difícil" que Π_2, pues teniendo un algoritmo $\mathcal{A}$ (resp. polinomial) para Π_2 podemos dar un algoritmo (resp. polinomial) para Π_1, que transforma instancias de Π_1 en instancias de Π_2, vía f, y las resuelve, vía $\mathcal{A}$. Por ende, el concepto de transformación polinomial de problemas captura la noción de relación de dificultad entre problemas.

Dentro de la clase **NP** existe una subclase importante de problemas, la de los "más difíciles" dentro de **NP**. Definimos primero, la clase de los problemas que son "más difíciles" que cualquier problema en **NP**,

$$\textbf{NP-hard} = \{\Pi : \text{para todo } \Pi' \in \mathbf{NP},\ \Pi' \leq_P \Pi\}$$

Los problemas "más difíciles" de **NP** son los de la clase

$$\textbf{NP-completo} = \mathbf{NP} \cap \textbf{NP-hard}$$

En 1971, Cook [3] demuestra que el problema de satisfactibilidad de fórmulas booleanas en forma normal conjuntiva (conocido como SAT) es **NP-completo**. Esta demostración es compleja, pues consiste en dar una transformación polinomial de un problema arbitrario $\Pi' \in \mathbf{NP}$ a $\Pi = \mathsf{SAT}$. Seguidamente, en 1972, Karp [13] da una lista de 21 problemas que son **NP-completo**, y a partir de allí el universo conocido de problemas **NP-completo** no paró de crecer. Por suerte, para demostrar que un problema es **NP-completo**, uno no siempre tiene que enfrentarse a la titánica tarea de dar una reducción desde un problema genérico de **NP**.

Lema 1.2. *La relación $\leq_P$ es transitiva. Esto es, si $\Pi_1 \leq_P \Pi_2$ y $\Pi_2 \leq_P \Pi_3$, entonces $\Pi_1 \leq_P \Pi_3$.*

Este lema tiene como corolario el siguiente resultado, que ofrece un método más sencillo para probar que un problema es **NP-completo**.

Teorema 1.3. $\Pi \in$ **NP-completo** *si y sólo si*

1. $\Pi \in \mathbf{NP}$
2. $\Pi' \leq_P \Pi$ *para algún problema* $\Pi' \in$ **NP-completo**.

La transitividad de $\leq_P$ garantiza que el segundo ítem del teorema, es equivalente a probar que $\Pi' \in$ **NP-hard**. En resumidas cuentas, para probar que un problema Π es **NP-completo**, podemos tomar un problema que ya sepamos que es **NP-completo** y reducirlo a Π.

Complejidad computacional de algunos problemas

El TSP es **NP-completo** [8, p. 211]. En [20] se demuestra que PTSP euclídeo es **NP-completo**, y se menciona que la misma demostración se puede utilizar para probar que PTSP rectilíneo es **NP-completo**. Otros dos problemas que aparecerán frecuentemente en esta tesis son los de matching máximo y vertex cover mínimo. Comencemos dando su descripción formal.

VERTEX COVER
INSTANCIA: $G = (V, E)$ un grafo y $k \in \mathbb{Z}_{\geq 0}$
SALIDA: ¿Existe un vertex cover de G de cardinal menor o igual a k?

MATCHING
INSTANCIA: $G = (V, E)$ un grafo y $k \in \mathbb{Z}_{\geq 0}$
SALIDA: ¿Existe un matching de G de cardinal mayor o igual a k?

El problema VERTEX COVER (abreviado VC) es **NP-completo** [13] y forma parte de la lista de 21 problemas **NP-completo** de Karp. En contraste, MATCHING está en **P**, demostrado por Edmonds [6] en 1965. Recordemos que, como afirma el Teorema 1.1, es $\nu(G) \leq \tau(G)$ para cualquier grafo G, y que el Teorema de Kőnig, indica que en grafos bipartitos, la brecha entre estos dos números se cierra, y vale la igualdad. Por lo tanto, en grafos bipartitos podemos reducir el problema de calcular $\tau(G)$, al de calcular $\nu(G)$. Como MATCHING es polinomial, siempre se puede calcular $\nu(G)$ en tiempo polinomial, y así llegamos al siguiente resultado.

Teorema 1.4. VC *sobre grafos bipartitos está en* **P**.

1.2.5. Algoritmos aproximados

Un algoritmo aproximado $\mathcal{A}$ para un problema de optimización Π es un algoritmo que dada una instancia I de Π produce una solución factible $\mathcal{A}(I)$, que no necesariamente es óptima. Para algunos algoritmos aproximados, es posible demostrar propiedades que indican cuán cerca está el valor de la solución factible obtenida, del valor de una solución óptima. A una de estas propiedades la llamamos *garantía de aproximación*.

Los algoritmos aproximados son de interés cuando no se conocen algoritmos exactos eficientes para un problema, como por ejemplo los de la clase **NP-completo**. Por esta razón, es deseable que un algoritmo aproximado corra en tiempo polinomial. Todos los algoritmos aproximados que consideramos en esta tesis son polinomiales.

Si Π es un problema de minimización, decimos que $\mathcal{A}$ es un algoritmo α-*aproximado* para Π, con $\alpha > 1$, si $\text{val}(\mathcal{A}(I)) \leq \alpha \text{OPT}$ para toda instancia I. Análogamente, si Π es de maximización, decimos que $\mathcal{A}$ es α-*aproximado* para Π, con $\alpha < 1$, si $\text{val}(\mathcal{A}(I)) \geq \alpha \text{OPT}$ para toda instancia I. El valor α se llama *factor de aproximación*, y no necesariamente es un número, sino que puede ser una función (que depende de algunos de los argumentos de la instancia).

Dado un problema de optimización Π, notamos R_Π al mejor factor de aproximación (el más pequeño en el caso de un problema de minimización y el más grande en caso de maximización) de un algoritmo aproximado para el problema Π.

Algoritmos aproximados para algunos problemas

El mejor algoritmo aproximado conocido para VC es 2-aproximado basado en la construcción de un matching maximal, que se puede encontrar en [9, p. 134]. Pese a la simplicidad de este algoritmo, no se conocen otros con mejor factor de aproximación. Se cree que VC no se puede aproximar con un factor más pequeño que 2.

Así como ciertos problemas admiten algoritmos aproximados, existen otros para los que, sorprendentemente, se puede demostrar que no tenemos ninguna esperanza de encontrar uno. Por ejemplo, en relación con el tema de esta tesis, el TSP no admite un algoritmo aproximado con factor constante [8, p. 147], sea cual sea esa constante positiva. La prueba de esto se puede adaptar fácilmente para probar que no existe un algoritmo aproximado con factor constante para PTSP.

El mejor algoritmo aproximado conocido para PTSP métrico es el $(3/2)$-aproximado de Hoogeveen [12], que es una adaptación del algoritmo de Christofides para TSP métrico [1]. Como toda instancia de PTSP rectilíneo es una instancia métrica, el $(3/2)$-aproximado de Hoogeveen también es un algoritmo aproximado para PTSP rectilíneo. No conocemos mejores algoritmos aproximados para PTSP rectilíneo, aunque es muy posible que existan. Para TSP euclídeo se conocen algoritmos aproximados de factor $1+\varepsilon$, para cada $\varepsilon > 0$. Este tipo de familias de algoritmos aproximados se conoce como *PTAS* (por *polynomial-time approximation scheme*). Suponemos que usando ideas similares a las del PTAS para TSP euclídeo, se puede dar un PTAS para la versión análoga de PTSP. Aún más, la existencia de un PTAS para las versiones euclídeas, son una señal de que posiblemente existan PTAS para otras métricas como la distancia Manhattan. De todas formas, notar que el $(3/2)$-aproximado de Hoogeveen genera soluciones factibles que son, en el peor caso, un 33 % peores que las de un potencial $(1+\varepsilon)$-aproximado, con lo cual a nivel práctico el $(3/2)$-aproximado hace un muy buen trabajo aproximando PTSP rectilíneo.

1.3. Descripción formal de SR y SS

Con todo el marco matemático introducido, estamos en condiciones de especificar más rigurosamente SR y SS, para eliminar cualquier ambigüedad posible.

SR (optimización)
INSTANCIA: $G = (V, E)$ un grafo y $X \subseteq E$.
SALIDA: Un camino de G, que cubra X, de longitud mínima.

SR (decisión)
INSTANCIA: $G = (V, E)$ un grafo, $X \subseteq E$ y $k \in \mathbb{Z}_{\geq 0}$.
SALIDA: ¿Existe un camino de G, que cubra X, de longitud menor o igual a k?

Notar que al modelar la ciudad como un grafo, estamos asumiendo que todas las calles son bidireccionales. Además, una solución factible tiene permitido recorrer más de una vez cada calle, y visitar más de una vez cada vértice.

SS (optimización)
INSTANCIA: $G = (V, E)$ un grafo, $X \subseteq E$ y P un camino de G que cubre X.
SALIDA: Un subconjunto de vértices de P, que cubra X, de cardinal mínimo.

SS (decisión)
INSTANCIA: $G = (V, E)$ un grafo, $X \subseteq E$, P un camino de G que cubre X y $k \in \mathbb{Z}_{\geq 0}$.
SALIDA: ¿Existe un subconjunto S de vértices de P, que cubra X, de cardinal menor o igual a k?

1.4. Estado del arte

Una importante familia de problemas de optimización combinatoria son aquellos conocidos como *problemas de ruteo de vehículos* (o VRP, por sus siglas en inglés). Estos problemas surgen comúnmente en la organización de las tareas de distribución de mercadería o personal, planificación de recorridos en robótica móvil o prestación de servicios a un conjunto de clientes mediante una flota de vehículos, entre otros casos. Los *vehículos* realizan sus movimientos a través de una red partiendo de puntos fijos, llamados *depósitos*. Cada tramo entre dos clientes de esta red tiene generalmente asociado un costo y/o tiempo de viaje que puede depender de muchos factores, como por ejemplo del tipo de vehículo o del período durante el cual el tramo es recorrido. Usualmente, se desea cumplir un cierto objetivo minimizando algún criterio dado (tiempos, costos, etc.). Este tipo de problemas tiene una gran relevancia en empresas de tamaño pequeño a grande, tanto en el sector público como privado. Aplicaciones clásicas se presentan en el diseño de las rutas de los vehículos de limpieza de calles o de recolección de basura, en el servicio de correos, en la distribución de mercadería desde depósitos centrales a negocios minoristas, en la planificación de los movimientos de una grúa para la carga y descarga de contenedores en el puerto, etc. Una excelente presentación sobre diferentes variantes de estos problemas puede verse en [23].

El trabajo original de Dantzig, Fulkerson y Johnson de 1954 [5] es el primer registro sobre VRP en la literatura y estudia el TSP, que es un caso particular de VRP. A este trabajo le siguió una gran cantidad de trabajos sobre el TSP. Clarke y Wright [2] incorporan mas de un vehículo al problema, lo cual lleva a la primera formulación propia del VRP, aunque este nombre no se le dio sino hasta el trabajo de Golden, Magnanti y Nguyan [10]. En 1974, Orloff [19] identifica una clase de problemas de ruteo de un sólo vehículo, a la que llama GENERAL ROUTING PROBLEM (abreviado GRP).

GRP
INSTANCIA: $G = (V, E)$ un grafo, $c : E \to \mathbb{Z}_{\geq 0}$ los pesos de G, $W \subseteq V$ y $F \subseteq E$.
SALIDA: Un circuito de G, que recorra cada arista de F y visite cada vértice de W, de peso mínimo.

Este problema es una generalización de otros problemas famosos, que se obtienen especializando algunos de los parámetros. Los dos más conocidos son el CHINESE POSTMAN PROBLEM ($W = \emptyset$, $F = E$) y el RURAL POSTMAN PROBLEM ($W = \emptyset$). Notablemente, el primero de los dos se puede resolver en tiempo polinomial, mientras que el segundo es **NP-completo** [14].

En la década de 1970 emergen muchas versiones distintas de VRP; por ejemplo, ruteo de

flotas [15], sistemas dial-a-bus [24], diseño de redes de transporte [18], ruteo de vehículos públicos [17], administración de la distribución [7] y recolección de residuos [16], entre otros. Solomon [22] agrega en 1983 el concepto de restricciones de "ventanas de tiempo".

En relación al problema de esta tesis, SR es un VRP simple, en el cual tenemos una flota de un único vehículo, con el que debemos realizar una distribución de mercadería, buscando minimizar el costo de la distribución, que está dado por la longitud total del recorrido que realiza el vehículo. Además, no hay un punto de partida prefijado para el vehículo, lo que implica que no se tiene en cuenta la distancia que el mismo deba recorrer desde un hipotético depósito hasta el punto inicial de una solución factible.

El problema SR es una combinación de dos conceptos importantes y frecuentes de la computación: el de VRP, que acabamos de discutir, y el de vertex cover, porque los clientes deben ser cubiertos visitando alguno de sus extremos, del mismo modo que las aristas de un grafo son cubiertas en un vertex cover. Como veremos a lo largo de la tesis, tanto el PTSP como VC son problemas que aparecen una y otra vez, como partes centrales de SR y SS. Hasta donde sabemos, SR no ha sido estudiado en la literatura, así como tampoco otros problemas que relacionen VRP con VC. Es importante destacar que todos los problemas de ruteo suelen basarse en alguna noción de *cubrimiento* de vértices o aristas, como se puede apreciar en la definición del GRP, aunque no conocemos ninguno que utilice la noción de vertex cover.

1.5. Resultados de la tesis

Para cada uno de los dos problemas, SR y SS, estudiamos su dificultad, tanto en términos de la existencia de algoritmos exactos (es decir, si son **NP-completo** o polinomiales), como de la existencia de algoritmos aproximados.

En el Capítulo 2, estudiamos el problema SS. Probamos que es **NP-completo** en el caso general, pero que al restringirlo sobre grafos bipartitos pasa a estar en **P**. Esto se debe a que SS tiene una fuerte relación con VC, que en el caso de grafos bipartitos es un problema polinomial. Mostramos que la relación con VC se extiende a la existencia de algoritmos aproximados, probando que existe un algoritmo α-aproximado para SS si y sólo si existe un algoritmo α-aproximado para VC.

Sobre SR hay mucho más para decir. En el Capítulo 3, comenzamos viendo que es **NP-completo** en el caso general, y luego estudiamos el problema restringiendo la entrada a tres clases de grafos de interés. En primer lugar, lo estudiamos sobre grafos grilla, que es un escenario de interés práctico, pues una ciudad puede modelarse de esta forma. En este caso, logramos demostrar que es **NP-completo**, aunque asumiendo una representación particular del grafo de entrada, que no está entre las usuales. Mientras que una representación tradicional de un grafo (por ejemplo, listas o matriz de adyacencia) contiene toda su topología, la representación que asumimos en este caso se aprovecha de la uniformidad de la estructura de una grilla, para obtener una representación más compacta. Analizamos en profundidad esta hipótesis y sus consecuencias, y argumentamos que es razonable. En segundo lugar, restringimos el problema a grafos completos, y demostramos que se mantiene **NP-completo**. En este caso, observamos que el problema tiene una íntima relación con VC. En tercer lugar, lo restringimos a árboles y, como es esperable, resulta estar en **P**. Más aún, damos un algoritmo lineal en el tamaño del grafo, que resuelve el problema. Éste es un algoritmo de programación dinámica, que procede en forma bottom-up sobre la estructura del árbol.

Habiendo probado que SR es **NP-completo** sobre distintas clases de grafos, en el Capítulo 4 nos dedicamos a buscar algoritmos exactos que resuelvan el problema, aunque sin esperanzas de que sean eficientes. Proponemos dos algoritmos, que resuelven el problema en el caso general, uno que utiliza la técnica de backtracking y corre en tiempo $O(k \cdot k!)$, donde k es la cantidad de clientes, y uno de programación dinámica, que corre en $O(k^4 2^k)$. Lamentablemente, no encontramos un algoritmo más rápido específico para el caso de grafos grilla, que es el de mayor interés.

Estos algoritmos no son eficientes, y sólo pueden manejar instancias del problema relativamente pequeñas, haciendo que su aplicación práctica sea muy limitada. Esto hace imprescindible la implementación de métodos heurísticos para acelerar el cómputo. Una de estas heurísticas es el uso de *funciones de acotación* (en inglés, *bounding functions*), que estiman el costo de completar una solución parcial a una solución factible, y en caso de que este costo sea mayor o igual a la mejor solución encontrada hasta el momento, se poda la rama actual de ejecución. Ideamos diversas funciones de acotación para SR. Implementamos los algoritmos y las funciones de acotación. Sin embargo, los resultados no son tan buenos como quisiéramos, porque aún con estas funciones de acotación, sólo logramos resolver, en un lapso de tiempo razonable, instancias de hasta 25 clientes. Toda la experimentación se realizó sobre grafos grillas cuadrados; la instancia resuelta de mayor cantidad de clientes es una grilla de 5×5 en la que el 60 % de las aristas son clientes.

Como no conocemos algoritmos polinomiales para casi ninguna de las restricciones de SR que consideramos, nos abocamos a buscar algoritmos aproximados, que es el tema del Capítulo 5. Consideramos el problema restringido sobre las tres clases de grafos para las cuales probamos que SR es **NP-completo**. Para cada una de ellas damos uno o más algoritmos aproximados, casi todos ellos basados en reducciones a otros problemas, para los cuales conocemos algoritmos aproximados. Específicamente, probamos lo siguiente:

- Para cada algoritmo α-aproximado para PTSP métrico y cada algoritmo β-aproximado para VC, existe un algoritmo aproximado para SR general, que da soluciones factibles de valor a lo sumo $\alpha(1 + 2\beta)OPT + 2\alpha(\beta - 1)$.
- Para cada algoritmo α-aproximado para PTSP rectilíneo, existe un algoritmo 3α-aproximado para SR sobre grillas.
- Para cada algoritmo α-aproximado para VC, con α constante, y para cada $\varepsilon > 0$, existe un algoritmo $(\alpha + \varepsilon)$-aproximado para SR sobre grafos completos.

En base a estos resultados, y utilizando los mejores algoritmos aproximados que conocemos para VC y PTSP métrico y rectilíneo, se obtienen algoritmos aproximados con las siguientes garantías:

- SR general: 7,5 OPT + 3
- SR sobre grillas: 4,5 OPT
- SR sobre completos: $(2 + \varepsilon)OPT$

Un resultado con un tinte distinto a los anteriores, es un algoritmo aproximado para SR sobre grillas cuya garantía de aproximación está en función de la cantidad $\bar{k}$ de aristas que *no* son clientes, y que es mejor cuanto mayor sea la densidad de clientes, es decir,

cuanto menor sea $\overline{k}$. Concretamente, el algoritmo produce soluciones factibles de valor a lo sumo

$$\left(\frac{3}{2} + C(n, m)\right)(OPT + \overline{k} + 1)$$

donde $C(n, m)$ es un número racional que satisface $C(n, m) \leq \frac{1}{2}$ si $n, m > 1$, y que tiende a 0 a medida que n y m tienden a infinito simultaneamente. Esto implica que si $\overline{k} = O(1)$, el algoritmo tiene una garantía $(\frac{3}{2} + C(n, m))OPT + O(1)$, y que tiende a $\frac{3}{2}OPT + O(1)$.

La Tabla 1.1, sintetiza los principales resultados de esta tesis, en cuanto a la complejidad computacional y a los factores de aproximación de los problemas estudiados.

Problema	**Complejidad**	**Factor de aproximación**
SS general	**NP-completo**	R_{VC}
SS sobre bipartitos	**P**	-
SR general	**NP-completo**	$R_{MPTSP}(1 + 2R_{VC})OPT + 2R_{MPTSP}(R_{VC} - 1)$
SR sobre grillas	**NP-completo** (*)	$3R_{RPTSP}OPT$
SR sobre completos	**NP-completo**	$(R_{VC} + \varepsilon)OPT$
SR sobre árboles	**P**	-

*: asumiendo una representación implícita de la entrada (ver el Capítulo 3).

Tabla 1.1: Principales resultados de la tesis. Llamamos MPTSP a PTSP métrico, y RPTSP a PTSP rectilíneo. La tercera columna indica los mejores factores de aproximación obtenidos para cada uno de los problemas.

Capítulo 2

Complejidad y algoritmos para STOPS SELECTION

En este capítulo estudiamos el problema SS. Comenzaremos viendo que en el caso general es un problema difícil, en lo que respecta a encontrar soluciones exactas. Luego, mostraremos que si nos restringimos a la clase de grafos bipartitos, el problema pasa a ser polinomial, siendo la clave de esto la capacidad de computar un vertex cover mínimo en tiempo polinomial, en grafos bipartitos. Finalmente, damos un algoritmo aproximado para el caso general y, más aún, mostramos que el mínimo factor de aproximación de SS es igual al de VC. Estos dos problemas resultan estar íntimamente relacionados, tanto en términos de algoritmos aproximados como de algoritmos exactos.

2.1. Complejidad del problema

2.1.1. SS general es NP-completo

La estrategia para probar que SS es **NP-completo** es hacer una reducción desde VC sobre grafos conexos. Al pedir que los grafos de entrada sean conexos, la reducción es más simple, pues SS cubre aristas a través de caminos, y sólo puede haber un camino que cubra un conjunto de aristas si todas ellas forman parte de una misma componente conexa.

Previamente debemos probar que VC sobre conexos es **NP-completo**, y para ello hacemos una reducción desde VC sobre grafos arbitrarios, que sabemos que es **NP-completo**.

Proposición 2.1. VC *sobre grafos conexos es* **NP-completo**.

Demostración. El problema está en **NP**, pues usando como certificado a un subconjunto de vértices S del grafo, basta chequear que cada arista sea cubierta por algún vértice de S. Obviamente esto se puede hacer en tiempo polinomial.

Veamos que está en **NP-hard**. Hacemos una reducción desde VC sobre grafos arbitrarios. Sea (G, k) una instancia de VC, y sean $H_1, \ldots, H_r$ las componentes conexas de G. A partir de G construimos el grafo G', tal como indica la Figura 2.1. Concretamente, agregamos dos vértices u y v, una arista $\{u, v\}$ y una arista $\{u, h_i\}$, con h_i un vértice arbitrario de H_i, para cada $1 \leq i \leq r$. Como G' es conexo, $(G', k+1)$ es una instancia de VC sobre conexos. La construcción de $(G', k+1)$ es polinomial, pues a G le agregamos una cantidad polinomial de aristas y vértices.

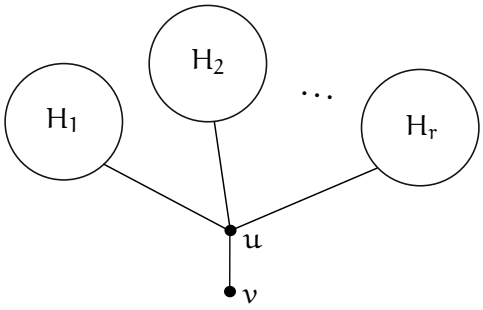


Figura 2.1: Gadget para la reducción de VC a VC sobre conexos.

Debemos ver que (G, k) es una instancia de SI de VC si y sólo si $(G', k+1)$ es una instancia de SI de VC sobre conexos. Si (G, k) es una instancia de SI para VC, entonces G tiene un vertex cover de k o menos vértices. Agregándole el vértice u obtenemos un vertex cover de G'. Entonces G' tiene un vertex cover de $k+1$ o menos vértices, o sea que $(G', k+1)$ es una instancia de SI de VC sobre conexos.

Recíprocamente, si $(G', k+1)$ es una instancia de SI de VC sobre conexos, G' admite un vertex cover S de $k+1$ o menos vértices. Como el conjunto S cubre la arista $\{u, v\}$, necesariamente contiene a u o a v. Como ni u ni v cubren aristas de G, el subconjunto $S \cap V(G)$ es un vertex cover de G, y tiene menos vértices que S, porque no contiene a u ni a v, que no están en $V(G)$. Luego, G tiene un vertex cover de k o menos vértices, y por ende (G, k) es una instancia de SI de VC. □

Lema 2.1. *Dado un grafo conexo* $G = (V, E)$*, podemos construir un camino de* G *que pase por todos sus vértices, que tenga longitud polinomial en el tamaño de* G*.*

Demostración. Comenzamos enumerando los elementos de V en un orden arbitrario. Luego, entre cada par consecutivo de estos vértices, trazamos un camino simple. Esta construcción está bien definida, porque siempre hay un camino entre dos vértices dados, por ser G conexo.

El camino que se obtiene tiene longitud menor o igual a $(|V| - 1)^2$, pues un camino simple entre cada par de vértices consecutivos de la enumeración inicial tiene longitud a lo sumo $|V| - 1$ (si no fuera simple, podría ser arbitrariamente grande). Esta cantidad es polinomial en el tamaño de G. Su construcción se puede hacer fácilmente en tiempo polinomial, pues podemos encontrar un camino mínimo entre dos vértices cualesquiera de G en tiempo polinomial (por ejemplo, haciendo una BFS), y esta operación se realiza $|V| - 1$ veces. □

Lema 2.2. *Sea* $G = (V, E)$ *un grafo conexo, y* P *un camino de* G *que visita todo vértice de* V*. Entonces* $SS^*(G, E, P) = \tau(G)$*.*

Demostración. La hipótesis de que P contiene todo vértice de G es la clave de la demostración. Por un lado, de ella se deduce que P cubre todas las aristas de G, y por ende que (G, E, P) es una instancia válida de SS. Por otro lado, implica que un conjunto de vértices S es vertex cover de G si y sólo si S es un conjunto de paradas válido de (G, E, P). De esto último se deduce la igualdad $SS^*(G, E, P) = \tau(G)$. □

Teorema 2.1. SS *es* **NP-completo**.

Demostración. Veamos que SS está en **NP**. Asumiendo que el certificado es un subconjunto de vértices de P, simplemente hay que chequear que cada una de las aristas de X esté cubierta por algún vértice de P.

Para ver que está en **NP-hard**, hacemos una reducción desde VC sobre grafos conexos. Sea (G, k) una instancia de VC con $G = (V, E)$ conexo.

Sea P un camino de G que cubre todos sus vértices, y de longitud polinomial, que sabemos que existe por el Lema 2.1. Transformamos la instancia (G, k) de VC sobre conexos, en la instancia (G, E, P, k) de SS. Esta transforamción es polinomial, pues el camino P se puede construir en tiempo polinomial. Observar que (G, E, P, k) es una instancia válida de SS, pues como P pasa por todos los vértices de G, debe cubrir todas sus aristas.

Para terminar, debemos ver que (G, k) es una instancia de SI de VC si y sólo si (G, E, P, k) es una instancia de SI de SS, pero esto se sigue directamente del Lema 2.2. □

2.1.2. SS sobre bipartitos está en P

Teorema 2.2. SS *sobre grafos bipartitos está en* **P**.

Para probar el teorema, presentamos el Algoritmo 1, que calcula un conjunto de paradas mínimo. El algoritmo funciona para cualquier clase de grafos, pero en el caso de bipartitos corre en tiempo polinomial, como probaremos.

Algoritmo 1 Algoritmo para SS.

Entrada: $G = (V, E)$ un grafo, $X \subset E$ y P un camino de G que cubre X.
Salida: Una solución óptima de SS para (G, X, P).

1: Sea A el conjunto de aristas e de X tal que exactamente uno de los extremos de e está cubierto por P.
2: Para cada $e \in A$, considerar el único extremo de e contenido en P. Sea S_A el conjunto de estos extremos.
3: Sea $B = X - \{e \in X : S_A \text{ cubre a } e\}$. Calcular un vertex cover mínimo S_B de $G[B]$.
4: **return** $S_A \cup S_B$

Teorema 2.3. *El Algoritmo 1 es correcto. Esto es, calcula una solución óptima de* SS *para* (G, X, P). *Además, si* G *es bipartito, corre en tiempo polinomial.*

Demostración. Cada arista de A se puede cubrir de una única forma, de modo que el conjunto de extremos S_A está contenido en toda solución de SS. El conjunto B contiene los clientes que no son cubiertos por S_A. Luego, cualesquiera dos soluciones factibles de SS sólo pueden diferir en la forma en que cubren B. Un vertex cover mínimo de $G[B]$ es un conjunto mínimo que cubre $G[B]$, y por lo tanto nos da una solución óptima para SS. Notar que todo vertex cover de $G[B]$ está contenido en P, porque, todo cliente de B tiene sus dos extremos en P.

Supongamos que G es bipartito, y veamos que el algoritmo corre en tiempo polinomial. Sólo debemos analizar el paso 3, pues los pasos 1-2 son claramente polinomiales. Como G es bipartito, $G[B]$ también lo es. Encontrar un vertex cover mínimo en un grafo bipartito es un problema polinomial, tal como afirma el Teorema 1.4. □

Corolario 2.1. SS *sobre grafos grilla está en* **P**.

2.2. Algoritmos exactos y aproximados

El Algoritmo 1 es un algoritmo exacto que resuelve SS para grafos arbitrarios. Si se admite cualquier tipo de grafo G en la entrada, sólo se conocen implementaciones exponenciales de la línea 3, pues VC sobre grafos arbitrarios es NP-completo. Este algoritmo puede verse como una reducción de SS a VC, en lo que respecta a encontrar una solución exacta. A partir de esta reducción se podría demostrar que VC es **NP-completo**, asumiendo que SS lo es.

A continuación probamos que VC y SS son igualmente difíciles en lo que respecta a encontrar soluciones aproximadas. Como parte de esto, damos un algoritmo aproximado para SS, que utiliza un algoritmo aproximado (cualquiera) para VC, como caja negra.

Teorema 2.4. *Existe un algoritmo α-aproximado para* VC *si y sólo si existe un algoritmo α-aproximado para* SS.

Demostración. ($\Rightarrow$) Sea $\mathcal{A}$ un algoritmo α-aproximado para VC. El algoritmo aproximado que proponemos para SS es similar al Algoritmo 1, pero en lugar de calcular un vertex cover mínimo en la línea 3, utilizamos $\mathcal{A}$ para obtener una aproximación del mismo. Lo presentamos como el Algoritmo 2.

Algoritmo 2 Algoritmo aproximado para SS.

Entrada: $G = (V, E)$ un grafo, $X \subset E$ y P un camino de G que cubre X.
Salida: Una solución factible de SS para (G, X, P).

1: Sea A el conjunto de aristas e de X tal que exactamente uno de los extremos de e está cubierto por P.
2: Para cada $e \in A$, considerar el único extremo de e contenido en P. Sea S_A el conjunto de estos extremos.
3: Sea $B = X - \{e \in X : S_A \text{ cubre a } e\}$. Calcular $S'_B = \mathcal{A}(G[B])$.
4: **return** $S_A \cup S'_B$

Es fácil ver que el algoritmo es polinomial, y que devuelve una solución factible de SS para (G, X, P). Para ver que el algoritmo es α-aproximado, debemos probar que $|S_A \cup S'_B| \leq \alpha\ \mathrm{SS}^*(G, X, P)$. Notar que de la correctitud del Algoritmo 1 se desprende que $\mathrm{SS}^*(G, X, P) = |S_A| + \tau(G[B])$. Se tiene

$$\begin{aligned}
|S_A \cup S'_B| &\leq |S_A| + |S'_B| \\
&\leq |S_A| + \alpha\ \tau(G[B]) && (\mathcal{A} \text{ es } \alpha\text{-aproximado}) \\
&\leq \alpha|S_A| + \alpha\ \tau(G[B]) && (\alpha \geq 1) \\
&= \alpha(|S_A| + \tau(G[B])) \\
&= \alpha\ \mathrm{SS}^*(G, X, P)
\end{aligned}$$

($\Leftarrow$) Sea $\mathcal{B}$ un algoritmo α-aproximado para SS. El algoritmo que proponemos para VC es el Algoritmo 3. Es fácil ver que el algoritmo es polinomial. Además, como cada conjunto de paradas S_H es un vertex cover de H, el resultado es un vertex cover de todo el grafo. La respuesta es tal que

$$
\begin{aligned}
\left|\bigcup_{H} S_H\right| &= \sum_{H} |S_H| && \text{(los conjuntos son disjuntos)} \\
&\leq \sum_{H} \alpha \; SS^*(H, E(H), P_H) && (\mathcal{B} \text{ es } \alpha\text{-aproximado}) \\
&= \sum_{H} \alpha \; \tau(H) && \text{(Lema 2.2)} \\
&= \alpha \sum_{H} \tau(H) \\
&= \alpha \; \tau(G)
\end{aligned}
$$

Algoritmo 3 Algoritmo aproximado para VC.

Entrada: $G = (V, E)$ un grafo.
Salida: Una solución factible de VC para G.

1: **for each** H componente conexa de G **do**
2: Calcular un camino P_H que pasa por todos los vértices de H, como indica el Lema 2.1.
3: Calcular $S_H = \mathcal{B}(H, E(H), P_H)$.
4: **return** $\bigcup_H S_H$

□

Corolario 2.2. *El mejor factor de aproximación posible para* SS *es igual al mejor factor de aproximación posible para* VC.

Capítulo 3

Complejidad de STAR ROUTING

En este capítulo estudiamos la complejidad de SR, al restringirlo a distintas clases de grafos; específicamente, grafos grilla, grafos completos y árboles. Probamos que es **NP-completo** en el caso general, sobre grillas y sobre completos, haciendo reducciones desde problemas adecuados. En el caso de árboles, demostramos que el problema se vuelve polinomial, exhibiendo un algoritmo de programación dinámica de tiempo lineal, que lo resuelve.

3.1. SR general es NP-completo

Usaremos una reducción desde el problema HAMILTONIAN PATH (abreviado HAM), que consiste en determinar la existencia de un camino hamiltoniano en un grafo. Este problema es **NP-completo** [9, p. 60].

HAM
INSTANCIA: $G = (V, E)$ un grafo.
SALIDA: ¿Existe un camino hamiltoniano en G?

Teorema 3.1. SR *es* **NP-completo**.

Demostración. Veamos primero que está en **NP**. Dada una instancia (G, X, k) de SR, usamos como certificado un camino sobre G, y verificamos que cubra X y tenga longitud a lo sumo k. Esto es claramente polinomial.

Probamos que el problema está en **NP-hard** haciendo una reducción desde HAM. Sea $G = (V, E)$ una instancia de HAM. Escribamos $V = \{v_1, \ldots, v_n\}$. Construimos una instancia de SR del siguiente modo. Por cada vértice v_i, agregamos un vértice u_i, y una arista $\{v_i, u_i\}$. Sea H el grafo resultante. Sea $X = \{\{v_i, u_i\} : 1 \leq i \leq n\}$. Consideramos $(H, X, n-1)$, que es una instancia válida de SR. Es fácil ver que esta construcción es polinomial. Resta ver que G es una instancia de SI de HAM si y sólo si $(H, X, n-1)$ es una instancia de SI de SR.

Si G tiene un camino hamiltoniano, entonces ese mismo camino sobre H es una solución factible de SR para (H, X). Como este camino de G es, más aún, hamiltoniano, tiene $n-1$ aristas, y por ende $(H, X, n-1)$ es una instancia de SI para SR.

Recíprocamente, supongamos que $(H, X, n-1)$ es una instancia de SI para SR. Si $n = 1$, se tiene que $(H, X, 0)$ y G son ambas instancias de SI, y terminamos. Supongamos $n > 1$.

Sea P una solución óptima de SR para (H, X). Como $(H, X, n-1)$ es una instancia de SI, vale $\text{length}(P) \leq n-1$. Afirmamos que P es un camino hamiltoniano de G.

En primer lugar, veamos que P es un camino de G. Esto es, que sólo contiene vértices de V. Supongamos, para llegar a un absurdo, que contiene un vértice u_i. Como $n > 1$, cuando P llega a u_i, debe provenir o ir hacia v_i, de modo que podemos eliminar esta visita, acortando la longitud total, y aún así seguiremos cubriendo todas las aristas de X. Esto contradice la minimalidad de P.

Veamos que P, además de ser un camino de G, es hamiltoniano. Cada vértice v_i es visitado al menos una vez, pues de lo contrario la arista $\{v_i, u_i\}$ de X quedaría sin cubrir (en particular, esto muestra que P tiene longitud $n-1$ o más). Más aún, el camino nunca visita dos veces el mismo vértice v_i, pues P tiene longitud $n-1$ o menos. Luego, P recorre cada vértice de G exactamente una vez. □

3.2. SR sobre grillas es NP-completo

Para demostrarlo, supondremos que las instancias de SR sobre grillas se representan de una forma particular, que condensa la información importante de la entrada. Comenzamos esta sección describiendo tal representación. En la demostración del Teorema 3.2 quedará claro por qué es necesaria. Luego de dicha demostración discutiremos las consecuencias de esta suposición y argumentaremos por qué es razonable.

Definición 3.1. Decimos que una instancia (G, X, k) de SR sobre grillas usa una *representación implícita*, si:

1. La grilla G se representa como un par de enteros (n, m), donde n y m son la cantidad de filas y columnas, respectivamente, de G.

2. Los vértices de G son todos los puntos de coordenadas enteras del plano, contenidos en el rectángulo de extremos $(0, 0)$, $(n-1, 0)$, $(0, m-1)$ y $(n-1, m-1)$. Esto implica que X se representa como un conjunto de pares de puntos de coordenadas enteras.

El nombre *implícito* proviene de que la representación no contiene información explícita sobre la topología del grafo (es decir, no utiliza una matriz, una lista de adyacencias, o alguna otra estructura con información topológica).

Para demostrar que SR sobre grillas es **NP-completo**, vamos a hacer una reducción desde PTSP rectilíneo, que, como ya mencionamos, es **NP-completo**. Antes de presentar la demostración, introducimos la transformación polinomial que usaremos para mapear instancias de PTSP rectilíneo en instancias de SR sobre grillas, y algunos lemas que describen propiedades de esta transformación.

3.2.1. Definición de la transformación

Subdividir una arista $e = \{u, v\}$ consiste en reemplazar e por dos aristas, $e_1 = \{u, w\}$ y $e_2 = \{w, v\}$, agregando un nuevo vértice w. Esta operación se generaliza naturalmente, del siguiente modo. Subdividir d veces una arista $e = \{u, v\}$ consiste en reemplazar e por $d+1$ aristas, $e_1 = \{u, w_1\}, \ldots, e_{d+1} = \{w_d, v\}$, agregando nuevos vértices $w_1, \ldots, w_d$.

Sea (G, c, k) una instancia de PTSP rectilíneo. Escribamos $W = V(G)$. Definimos la transformación $f(G, c, k) := (H, X, (d+1)k)$, donde

- $d = 2(|W| - 1)$ es un entero positivo cuyo valor cobrará sentido más adelante.
- H es el grafo que se obtiene del siguiente modo. Consideremos el mínimo cuadrilátero, con lados paralelos a los ejes de coordenadas, que contiene a todos los puntos W en el plano. Específicamente, es el cuadrilátero limitado por los puntos de más abajo, de más arriba, de más a la izquierda, y de más a la derecha de W. Notar que sus cuatro esquinas son puntos de coordenadas enteras. Este cuadrilátero induce un grafo grilla, cuyos vértices son los puntos de coordenadas enteras contenidos en el cuadrilátero, y tal que tiene una arista exactamente entre cada par de puntos del plano a distancia 1. Sea H_0 esta grilla; le aplicamos las siguientes transformaciones.

  1. Subdividimos d veces cada arista. (Notar que el grafo resultante de las subdivisiones no es una grilla, pues los vértices agregados en las subdivisiones tienen grado 2.)
  2. De la forma natural, agregamos las aristas y vértices necesarios para obtener un grafo grilla.

  El resultado final es el grafo H. Notar que el conjunto W es un subconjunto de vértices de H.
- X es un conjunto de aristas de H que contiene, por cada vértice de W, una arista (cualquiera) incidente a ese vértice.

La Figura 3.1 ilustra una transformación, paso a paso. Es claro que $(H, X, (d+1)k)$ es una instancia válida de SR sobre grillas.

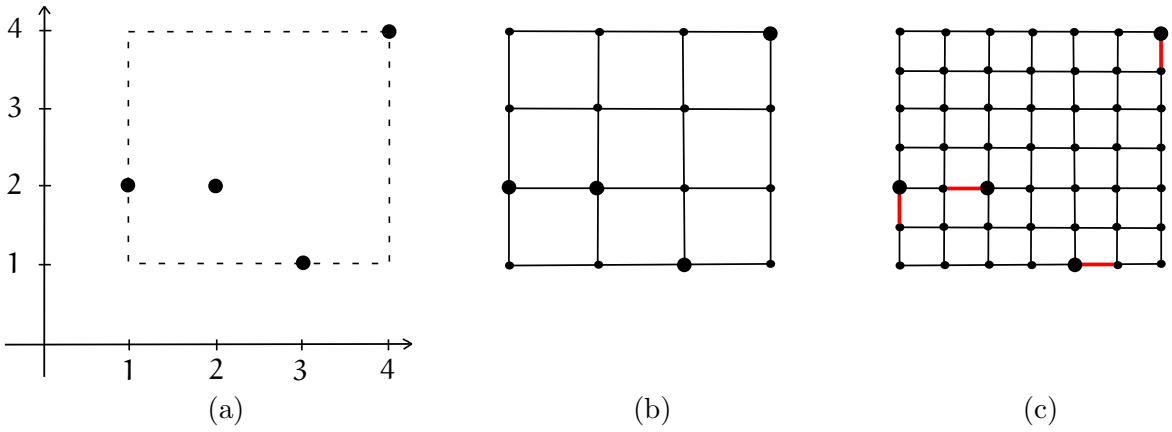


Figura 3.1: (a) Puntos en el plano. (b) El grafo H_0. (c) El grafo H, si se realizaran $d = 1$ subdivisiones por arista (en realidad la transformación usaría $d = 6$, pero esto es más difícil de dibujar). En rojo se indican las aristas de X.

En este punto ya se puede observar la utilidad de la representación implícita. Como el grafo H debe tener tamaño polinomial en el tamaño de (G, c, k), no podemos utilizar una representación para H que almacene toda su topología. Por ejemplo, si G es un grafo de 2 vértices, y c indica que están a distancia 1000000, entonces el tamaño de (G, c, k) es pequeño en relación a los 1000001 o más vértices de H. Como estamos asumiendo que las instancias de SR sobre grillas se representan implícitamente, la transformación f debe

producir $(H, X, (d+1)k)$ con esa representación. Por lo tanto, debemos ver cómo computar la representación implícita de esta instancia.

Supongamos que H_0 tiene n filas y m columnas. Al finalizar la construcción de H, el grafo que obtenemos es una grilla de $N = d(n-1)+n$ filas, pues a las n filas que tenía H_0 se le agregaron $d(n-1)$ a causa de las subdivisiones, y $M = d(m-1)+m$ columnas, por la misma razón. Luego, H se representa como el par (N, M).

Para construir X debemos modificar acordemente las coordenadas de los vértices de G, a lo largo de las transformaciones que le aplicamos a H_0. Esta transformación de las coordenadas debe obedecer a la definición de representación implícita, que indica que los vértices de H son aquellos puntos de coordenadas enteras en el rectángulo determinado por $(0,0)$, $(n-1,0)$, $(0,m-1)$ y $(n-1,m-1)$. Omitimos aquí los detalles técnicos, pero es fácil ver que para cada vértice de G, es posible calcular la variación de sus coordenadas haciendo aritmética básica, en tiempo polinomial.

3.2.2. Propiedades de la transformación

Lema 3.1. *La transformación* f *es polinomial en el tamaño de* (G, c, k).

Demostración. Llamemos $I = (G, c, k)$ a la instancia. Debemos ver que el cómputo de d, H y X son polinomiales en size(I). Asumimos que G, c y k se representan de las formas tradicionales. Si D es el máximo peso entre dos vértices de G, tenemos que

$$\begin{aligned}\text{size}(I) &= \Omega(\text{size}(G) + \text{size}(c) + \text{size}(k)) \\ &= \Omega(|W|^2 + \lg D + \lg k)\end{aligned}$$

pues $\text{size}(G) = \Omega(|W|^2)$ (G es un grafo completo) y $\text{size}(c) = \Omega(\lg D)$ (D es uno de los pesos de c). Sea $S = |W|^2 + \lg D + \lg k$, de modo que $\text{size}(I) = \Omega(S)$. Veamos que el cómputo de la instancia $(H, X, (d+1)k)$ representada implícitamente cuesta $O(\text{poly}(S)) = O(\text{poly}(\text{size}(I)))$.

El costo de calcular $(d+1)k$ es $C_1 = O(1 + \lg d \lg k)$. Como $\lg d = \lg(2(|W|-1)) = O(\lg |W|) = O(\lg S)$, se tiene $C_1 = O(S \lg S) = O(\text{poly}(S))$. El costo de calcular $N = d(n-1)+n$ es $C_2 = O(1+\lg d \lg n)$. Como H_0 tiene n filas y m columnas, el cuadrilátero del que proviene es de tamaño $n \times m$. Entonces $D \geq n$, pues este cuadrilátero tiene al menos un vértice de G en cada borde, y en particular $\lg n = O(\lg D) = O(S)$. Luego, $C_2 = O(S \lg S) = O(\text{poly}(S))$. Análogamente se prueba que el costo de calcular $M = d(m-1)+m$ es $C_3 = O(\text{poly}(S))$.

Finalmente, analicemos el costo C_4 de calcular X. Este conjunto tiene $|W|$ aristas (una por cada vértice de G), y cada una se calcula en tiempo polinomial. Luego, $C_4 = O(|W|\text{poly}(S)) = O(\text{poly}(S))$. □

Lema 3.2. *A partir de un camino hamiltoniano* P *de* G*, se puede obtener un camino* Q *de* H*, que cubre* X*, y tal que* $\text{length}(Q) = (d+1)\text{length}_c(P)$.

Demostración. Podemos replicar el camino P sobre H_0 fácilmente, cambiando cada arista $\{u, v\}$ de P por un camino mínimo entre u y v en la grilla H_0. Así obtenemos un camino P' de H_0 que visita todo W (pues P es hamiltoniano) y de $\text{length}(P') = \text{length}_c(P)$ (pues $\text{dist}_{H_0}(u, v) = c(u, v)$). A P' lo podemos transformar en un camino de H, aplicando las d subdivisiones sobre cada una de sus aristas. Si P'' es este nuevo camino, entonces $\text{length}(P'') = (d+1)\text{length}(P')$. Luego,

$$\text{length}(P'') = (d+1)\text{length}(P') = (d+1)\text{length}_c(P)$$

Como P' visita todo W, P'' también lo hace, y por lo tanto P'' cubre todo X en H. Entonces $Q = P''$ cumple lo buscado. □

Lema 3.3. *A partir de un camino* P *de* H *que cubre* X*, se puede obtener un camino hamiltoniano* Q *de* G*, tal que* $\text{length}_c(Q) < \text{length}(P)/(d+1) + 1$.

Demostración. Escribamos $P = \langle u_1, \ldots, u_r \rangle$. La observación clave es que para cada $w \in W$, existe un vértice u_i de P tal que $\text{dist}_H(w, u_i) \leq 1$. Esto es porque cada vértice de W es extremo de al menos una arista e de X. Como P cubre X, debe pasar por alguno de los extremos de e.

Sea $w_1, \ldots, w_{|W|}$ una enumeración de los vértices de W, y sean $1 \leq j_1, \ldots, j_{|W|} \leq r$ índices tales que $\text{dist}_H(w_i, u_{j_i}) \leq 1$ para cada i. Sin pérdida de generalidad, supongamos que $j_1 \leq \cdots \leq j_{|W|}$ (si no estuvieran ordenados crecientemente, los ordenamos, modificando acordemente la enumeración de W considerada). Consideramos $Q = \langle w_1, \ldots, w_{|W|} \rangle$, que es un camino hamiltoniano de G. Su peso satisface

$$\begin{aligned}
\text{length}_c(Q) &= \sum_{i=1}^{|W|-1} c(w_i, w_{i+1}) \\
&= \sum_{i=1}^{|W|-1} \text{dist}_{H_0}(w_i, w_{i+1}) && \text{(construcción de } H_0\text{)} \\
&= \sum_{i=1}^{|W|-1} \text{dist}_H(w_i, w_{i+1})/(d+1) && \text{(construcción de } H\text{)} \\
&\leq \sum_{i=1}^{|W|-1} (\text{dist}_H(w_i, u_{j_i}) + \text{dist}_H(u_{j_i}, u_{j_{i+1}}) \\
&\qquad + \text{dist}_H(u_{j_{i+1}}, w_{i+1}))/(d+1) && (w_i \to u_{j_i} \rightsquigarrow u_{j_{i+1}} \to w_{i+1} \text{ es un} \\
&&& \text{camino factible de } w_i \text{ a } w_{i+1} \text{ en } H) \\
&\leq \sum_{i=1}^{|W|-1} (1 + \text{dist}_H(u_{j_i}, u_{j_{i+1}}) + 1)/(d+1) \\
&= \left(\sum_{i=1}^{|W|-1} \text{dist}_H(u_{j_i}, u_{j_{i+1}}) + 2(|W|-1) \right) /(d+1) \\
&\leq (\text{length}(P) + 2(|W|-1))/(d+1) && (j_1 \leq \cdots \leq j_{|W|}) \\
&= (\text{length}(P) + d)/(d+1) && (d = 2(|W|-1)) \\
&< \text{length}(P)/(d+1) + 1
\end{aligned}$$

□

3.2.3. La demostración

Teorema 3.2. SR *sobre grafos grilla, usando una representación implícita para las instancias, es* **NP-completo**.

Demostración. La idea para ver que está en **NP** es similar a la del Teorema 3.1, aunque con la nueva representación de la entrada elegida debemos tomar algunos recaudos. La representación implícita nos impide representar el camino como una secuencia de vértices tradicional, en la que todo par de vértices consecutivos son adyacentes, pues esto podría no tener longitud polinomial en el tamaño de la entrada. Dada una instancia (G, X, k) de SR sobre grillas, el certificado sólo debe contener aquellos vértices del camino que sean extremos de aristas de X, en el orden en que aparecen en el camino. Para chequear que cubre X, recorremos la secuencia marcando las aristas de X cubiertas, y al final verificamos que estén todas cubiertas. Para chequear que tenga longitud k o menos, acumulamos la distancia Manhattan entre cada par consecutivo de vértices, y verificamos que esta longitud total sea k o menos.

Veamos que el problema es **NP-hard**. La reducción es desde PTSP rectilíneo, utilizando la transformación f. Debemos ver que (G, c, k) es una instancia de SI de PTSP rectilíneo si y sólo si $f(G, c, k) = (H, X, (d+1)k)$ es una instancia de SI de SR sobre grillas.

Sea (G, c, k) es una instancia de SI de PTSP rectilíneo, entonces existe un camino hamiltoniano P en G de peso a lo sumo k. Por el Lema 3.2, existe un camino Q que cubre todo X, con longitud a lo sumo $(d+1)k$. Luego, $f(G, c, k)$ es una instancia de SI de SR sobre grillas.

Recíprocamente, sea $(H, X, (d+1)k)$ una instancia de SI de SR sobre grillas. Entonces existe un camino P de H que cubre X, de longitud $\text{length}(P) \leq (d+1)k$. Por el Lema 3.3, existe un camino hamiltoniano Q de G tal que

$$\text{length}_c(Q) < \text{length}(P)/(d+1) + 1 \leq k + 1$$

Luego $\text{length}_c(Q) < k+1$, y como $\text{length}_c(Q)$ y k son enteros, es $\text{length}_c(Q) \leq k$. Esto implica que (G, c, k) es una instancia de SI de PTSP rectilíneo. □

3.2.4. Sobre la representación implícita

Es válido cuestionarse si es razonable asumir una representación implícita para las instancias de SR sobre grillas. Para responder esta pregunta, primero debemos comprender las consecuencias de elegir una representación determinada. Formalmente, un problema de decisión es una tupla de dos lenguajes formales de cadenas de bits, uno de instancias de SI y otro de instancias de NO. Esto es, cada instancia es una cadena de bits. Hasta ahora, cada vez que describimos un problema, hablamos de sus instancias como tuplas de objetos matemáticos abstractos (tales como grafos, funciones y números enteros), pero nunca especificamos cuál es la cadena de bits asociada a cada una. Para completar la especificación del problema, deberíamos indicar cuál es la cadena de bits asociada a cada instancia abstracta. Este mapeo de instancias abstractas en cadenas de bits se denomina *esquema de representación* [9, p. 19]. Una *representación* para las instancias es una descripción informal de un esquema de representación, pues indica cómo mapear cada instancia abstracta a otro objeto matemático cuya estructura está "más cerca" de ser su cadena de bits asociada. Al fijar una representación para las instancias de un problema, estamos limitando las representaciones que deseamos admisibles a aquellas que podemos

transformar, en tiempo polinomial, en la representación fijada. (Más formalmente, si nuestra representación describe un esquema e, otra representación que describe un esquema e' es admisible si existe una función computable en tiempo polinomial f, tal que $e = f \circ e'$.) Por lo tanto, al suponer que las instancias de SR sobre grillas se representan implícitamente, estamos asumiendo que las representaciones admisibles son aquellas que se pueden transformar en tiempo polinomial en una del tipo implícita.

Lo segundo que debemos analizar es si tiene sentido que las representaciones admisibles sean aquellas de este tipo. Nuestro argumento a favor de esto es que un algoritmo que intenta resolver SR sobre grillas con una representación implícita, puede obtener toda la información necesaria sobre la topología del grafo eficientemente, sin necesidad de tenerla almacenada en memoria. Entre otras cosas, puede:

- Determinar si dos vértices son adyacentes, en tiempo $O(1)$. Basta comparar las coordenadas de los dos vértices en cuestión.
- Iterar la vecindad de un vértice, con costo $O(1)$ por iteración. Esto es posible debido a que la vecindad de un vértice es el subconjunto de los 4 puntos del plano que lo rodean, que caen dentro de la grilla.

Estas operaciones son las que realizamos usualmente con una representación tradicional de un grafo, por lo que un algoritmo que utiliza la representación implícita probablemente no debería verse limitado a la hora de obtener información topológica del grafo.

En contra de esta representación, tenemos limitaciones a la hora de trabajar con algunas estructuras habituales cuyo tamaño no es polinomial. Por ejemplo, un camino representado como una secuencia de vértices adyacentes de a pares consecutivos podría tener tamaño exponencialmente más grande que el de la instancia, de modo que un algoritmo polinomial que utilice una representación implícita no podrá construir caminos de este tipo.

Si bien el Teorema 3.2 y su demostración asumen una representación implícita para la entrada del problema, conjeturamos que también es posible demostrarlo asumiendo una representación tradicional. Una tal demostración debería utilizar una reducción más compleja, quizás desde otro problema **NP-completo**, que condense información de forma más inteligente. La reducción utilizada en este trabajo es una traducción directa y simple, que se basa en la similitud de los problemas involucrados, reduciendo el esfuerzo necesario para codificar, en forma sintética, la información de una instancia de PTSP rectilíneo como una instancia de SR sobre grillas.

Otra consecuencia de asumir esta representación, es que no nos permite concluir que SR restringido a superclases de los grafos grilla (por ejemplo, la clase de todos los grafos, o la de los grafos bipartitos) es **NP-completo**, pues estas superclases no necesariamente admiten una representación implícita, que está específicamente diseñada para grafos grilla. Luego, una transformación identidad de una instancia de SR sobre grillas, a una instancia de SR sobre una superclase de grafos, implica realizar una transformación de la representación que podría tomar tiempo supra-polinomial.

3.3. SR sobre grafos completos es NP-completo

En este caso, haremos una reducción desde VC. Si G es un grafo completo, tenemos completa libertad para construir el camino que cubra X, pues podemos movernos entre

cualquier par de vértices. Esto hace que el problema se convierta, esencialmente, en calcular un vertex cover de $G[X]$.

Lema 3.4. *Si* G *es un grafo completo,* $SR^*(G, X) = \tau(G[X]) - 1$.

Demostración. ($\leq$) Sea $C = \{u_1, \ldots, u_r\}$ un vertex cover mínimo de $G[X]$. La secuencia $P = \langle u_1, \ldots, u_r \rangle$ es un camino en G, pues G es completo, y además cubre X, con lo cual es una solución factible de SR para (G, X). Luego $SR^*(G, X) \leq \text{length}(P) = |C| - 1 = \tau(G[X]) - 1$.

($\geq$) Sea $P = \langle u_1, \ldots, u_r \rangle$ una solución óptima de SR para (G, X). La observación clave es que todo u_i es extremo de alguna arista de X. Si no fuera así, podríamos sacar cualquiera de los vértices que no lo cumplan, y seguríamos teniendo un camino de G, por ser G completo, que sigue cubriendo X, y que es de menor longitud. Esto contradice la minimalidad de P. Luego, $C = \{u_1, \ldots, u_r\}$ está contenido en $G[X]$. Como además cubre X, es vertex cover de $G[X]$. Por lo tanto, $\tau(G[X]) \leq |C| = \text{length}(P) + 1 = SR^*(G, X) + 1$. □

Teorema 3.3. SR *sobre grafos completos es* **NP-completo**.

Demostración. La prueba de que está en **NP** es idéntica a la del Teorema 3.1. Veamos que está en **NP-hard**. Hacemos una reducción desde VC. Dada una instancia (G, k) de VC, con $G = (V, E)$, la transformamos en la instancia $(K(V), E, k - 1)$ de SR sobre completos. Es fácil ver que la transformación es polinomial.

Debemos ver que (G, k) es una instancia de SI de VC si y sólo si $(K(V), E, k-1)$ es una instancia de SI de SR sobre completos. Para esto, basta probar que $SR^*(K(V), E) = \tau(G)-1$.

Llamemos I al grafo formado por los vértices aislados de G. Notemos que $K(V)[E] \cup I = G$. Por el Lema 3.4, $SR^*(K(V), E) = \tau(K(V)[E]) - 1$. Como un vertex cover mínimo nunca contiene vértices aislados, resulta que $\tau(K(V)[E]) = \tau(K(V)[E] \cup I)$. Juntando todo,

$$SR^*(K(V), E) = \tau(K(V)[E]) - 1 = \tau(K(V)[E] \cup I) - 1 = \tau(G) - 1$$

como queríamos. □

Notar que este resultado nos da una demostración alternativa de que SR general es **NP-completo**.

3.4. SR sobre árboles es P

En esta sección damos un algoritmo que resuelve SR sobre árboles, en tiempo polinomial, que utiliza la técnica de programación dinámica. Comenzamos definiendo algunas variantes de SR, que serán útiles para describir el algoritmo.

s-SR
INSTANCIA: $G = (V, E)$ un grafo, $X \subseteq E$, y $s \in V$.
SALIDA: Un camino de G, que empiece en s y que cubra X, de longitud mínima.

st-SR
INSTANCIA: $G = (V, E)$ un grafo, $X \subseteq E$, y $s, t \in V$.
SALIDA: Un camino de G, que empiece en s, que termine en t y que cubra X, de longitud mínima.

u-SR
INSTANCIA: $G = (V, E)$ un grafo, $X \subseteq E$, y $u \in V$.
SALIDA: Un camino de G, que pase por u y que cubra X, de longitud mínima.

Sea G un árbol, y sea r un vértice de G cualquiera, que fijamos como raíz. Consideremos el ordenamiento jerárquico de G en el que r es la raíz. Para cada vértice t, raíz de un subárbol que notamos T, definimos los siguientes parámetros:

$$\texttt{Start-SR}(t) = \text{s-SR}^*(T, X \cap E(T), t)$$

$$\texttt{Through-SR}(t) = \text{u-SR}^*(T, X \cap E(T), t)$$

$$\texttt{Tour-SR}(t) = \text{st-SR}^*(T, X \cap E(T), t, t)$$

$$\texttt{SR}(t) = \text{SR}^*(T, X \cap E(T))$$

Informalmente, $\texttt{Start-SR}(t)$ es el óptimo en T entre todos los caminos que empiezan en t, $\texttt{Tour-SR}(t)$ el óptimo en T entre todos los circuitos que pasan por t, $\texttt{Through-SR}(t)$ el óptimo en T entre todos los caminos que pasan por t, y $\texttt{SR}(t)$ el óptimo en T, sin ninguna restricción. Se tiene, por lo tanto, $\text{SR}^*(G, X) = \texttt{SR}(r)$.

Diremos que un camino es una solución factible de $\texttt{Start-SR}$ para t si es una solución factible de s-SR para $(T, X \cap E(T), t)$. Diremos que un camino realiza $\texttt{Start-SR}(t)$ si es una solución factible de $\texttt{Start-SR}$ para t y tiene longitud $\texttt{Start-SR}(t)$. Usamos las mismas convenciones con $\texttt{Tour-SR}$, $\texttt{Through-SR}$ y $\texttt{SR}$.

Llamemos $T_1, \ldots, T_k$ a los subárboles de t. Sea t_i la raíz del subárbol T_i, y sea e_i la arista que une t con t_i. La Figura 3.2 ilustra el árbol T.

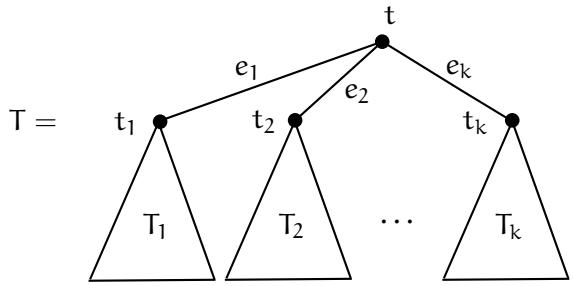


Figura 3.2: El subárbol T y sus componentes.

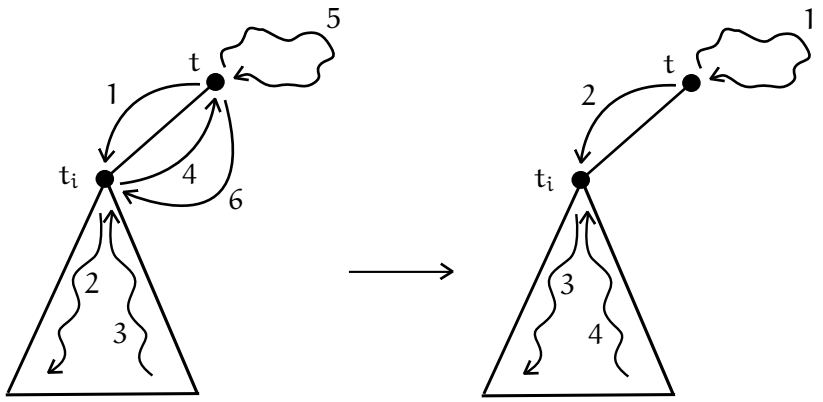
Con el fin de implementar un algoritmo de programación dinámica, queremos expresar recursivamente $\texttt{Start-SR}(t)$, $\texttt{Tour-SR}(t)$, $\texttt{Through-SR}(t)$ y $\texttt{SR}(t)$, en función de los subárboles de t. Comenzamos con un lema que nos ayuda a acotar las posibles formas de los caminos que realizan estos valores. Decimos que un camino de T *entra* en T_i, cuando atraviesa la arista e_i desde t. Decimos que *sale* de T_i, cuando atraviesa la arista e_i desde t_i.

Lema 3.5. *Un camino que realiza* $\texttt{Start-SR}(t)$, $\texttt{Tour-SR}(t)$ *ó* $\texttt{Through-SR}(t)$, *nunca entra más de una vez en un subárbol de* t *y nunca sale más de una vez de un subárbol de* t.

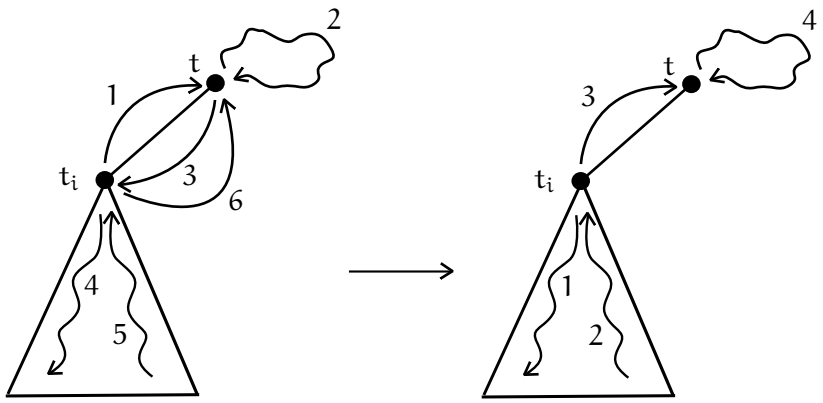
Demostración. Supongamos que un tal camino entra más de una vez en un subárbol de t, y veamos que es posible reconfigurar el camino de modo tal de reducir la longitud

del mismo, sin dejar de cubrir ningún cliente. La transformación consiste en tomar un tramo del camino, que se extienda entre dos entradas consecutivas en un subárbol de t, y sustituirlo por otro recorrido, tal como indica la Figura 3.3-(a). Esta transformación da lugar a un camino válido de G, y preserva el primer y último vértice, porque el tramo original y el sustituto comienzan en el mismo vértice y terminan en el mismo vértice. Además, se puede ver que ambos visitan el mismo conjunto de vértices, con lo cual el camino resultante no deja de visitar ningún vértice ni de cubrir ningún cliente. Luego, si el camino era una solución factible de x-SR para t, con x ∈ {Start, Tour, Through}, entonces el resultado es una solución de x-SR para t. Finalmente, observemos que su longitud es 2 unidades menor respecto del original, lo cual es una contradicción, porque supusimos que el original era óptimo.

La demostración para el caso de múltiples salidas de un subárbol es análoga. La transformación en este caso se puede ver en la Figura 3.3-(b).



(a)



(b)

Figura 3.3: Transformación del tramo de un camino, que se extiende entre dos entradas o salidas consecutivas. Las aristas etiquetadas con números indican el recorrido. (a) Dos entradas consecutivas. (b) Dos salidas consecutivas.

□

El Lema 3.5 permite caracterizar la forma de los caminos que realizan `Start-SR`(t), `Tour-SR`(t) y `Through-SR`(t).

Corolario 3.1. *Un camino que realiza* `Start-SR`(t) *tiene la siguiente forma:*

1. *comienza en* t*;*
2. *entra en, recorre y sale de subárboles* $\mathsf{T}_{i_1}, \ldots, \mathsf{T}_{i_{\ell-1}}$*, todos distintos;*
3. *entra en, recorre y termina en un subárbol* T_{i_ℓ}*, distinto de los anteriores.*

Corolario 3.2. *Un camino que realiza* `Tour-SR`(t) *tiene la siguiente forma:*

1. *comienza en* t*;*
2. *entra en, recorre y sale de subárboles* $\mathsf{T}_{i_1}, \ldots, \mathsf{T}_{i_\ell}$*, todos distintos;*
3. *termina en* t*.*

Corolario 3.3. *Un camino que realiza* `Through-SR`(t)*, o bien es un camino que realiza* `Start-SR`(t)*, o bien es un camino que al invertirlo realiza* `Start-SR`(t)*, o bien tiene la siguiente forma:*

1. *comienza, recorre y sale de un subárbol* T_{i_1}*;*
2. *entra en, recorre y sale de subárboles* $\mathsf{T}_{i_2}, \ldots, \mathsf{T}_{i_{\ell-1}}$*, todos distintos;*
3. *termina de una de las siguientes formas:*
   - *o bien entra, recorre y termina en el subárbol* T_{i_1}*;*
   - *o bien entra, recorre y termina en un subárbol* T_{i_ℓ}*, distinto de todos los anteriores.*

Conociendo la forma de estos caminos, podemos determinar fórmulas recursivas para `Start-SR`(t), `Tour-SR`(t) y `Through-SR`(t), pues cumplen la propiedad de subestructura óptima. Esto es, un camino que realiza uno de estos valores, está compuesto por subcaminos óptimos. Algunas de estas fórmulas recursivas varían según la cantidad de subárboles de t que contengan algún cliente. Intuitivamente, si tenemos sólo un subárbol con clientes, debemos concentrar nuestra atención allí, mientras que si hay dos o más con algún cliente, debemos mantener una visión más global. Haremos una división en casos utilizando este criterio. Llamemos $\mathtt{X}(\mathsf{t})$ a la cantidad de clientes en el subárbol T, es decir, $\mathtt{X}(\mathsf{t}) = |\mathsf{X} \cap \mathsf{E}(\mathsf{T})|$.

Caso 1: ningún subárbol contiene aristas de X

Afirmación 3.1. $\mathit{SR}(\mathsf{t}) = \mathit{Start\text{-}SR}(\mathsf{t}) = \mathit{Tour\text{-}SR}(\mathsf{t}) = \mathit{Through\text{-}SR}(\mathsf{t}) = 0$.

Caso 2: exactamente un subárbol tiene aristas de X

Sea T_p tal subárbol.

Afirmación 3.2. *Start-SR*$(t) = 1 +$ *Start-SR*(t_p).

Justificación. Un camino que realiza Start-SR(t) comienza en t, y luego recorre un camino que realiza Start-SR(t_p). □

Demostración. Sea $P = \langle u_1, \ldots, u_s \rangle$ una solución que realiza Start-SR(t). El camino comienza en $u_1 = t$. El resto del camino $\langle u_2, \ldots, u_s \rangle$ sólo debe cubrir las aristas de X en T_p. Ésto, junto con la minimalidad de P, implican que ese subcamino está contenido en T_p. Luego, $\langle u_2, \ldots, u_s \rangle$ es una solución factible de Start-SR para t_p. Afirmamos que es óptima. Si no lo fuese, podríamos tomar P e intercambiar dicho subcamino por uno óptimo, y obtener una solución de Start-SR para t, de menor longitud. Esto contradice la minimalidad de P. Luego

$$\text{Start-SR}(t) = \text{length}(P) = 1 + \text{length}(\langle u_2, \ldots, u_s \rangle) = 1 + \text{Start-SR}(t_p)$$

□

Afirmación 3.3. *Tour-SR*$(t) = 1 +$ *Tour-SR*$(t_p) + 1$.

Justificación. Un circuito que realiza Tour-SR(t) comienza en t, luego recorre un circuito que realiza Tour-SR(t_p), y finalmente vuelve a t. □

Afirmación 3.4. *Through-SR*$(t) = \text{mín}\{$*Start-SR*$(t),$ *Through-SR*$(t_p) + 2\}$

Justificación. Un camino que realiza Through-SR(t) es de *una* de las siguientes formas:

- Empieza en t, y es un camino que realiza Start-SR(t).
- Termina en t, y es un camino que al invertirlo realiza Start-SR(t).
- No empieza ni termina en t, y es un camino que se obtiene tomando un camino que realiza Through-SR(t_p), e insertándole una visita a t justo después de alguna ocurrencia de t_p (es decir, recorriendo además la arista e_p, en ambos sentidos).

□

Caso 2-1: algún e_i, con $i \neq p$, pertenece a X

Afirmación 3.5. *SR*$(t) =$ *Through-SR*(t).

Justificación. Un camino que realiza SR(t) debe pasar por t, porque e_p y e_i son clientes. □

Caso 2-2: sólo e_p pertenece a X

Afirmación 3.6. *SR*$(t) =$ *Through-SR*(t_p).

Justificación. Hay un camino que realiza SR(t) que está completamente contenido en T_p, porque el único cliente fuera de este subárbol es e_p, que puede cubrirse desde t_p. □

Caso 2-3: ningún e_i pertenece a X

Afirmación 3.7. $SR(t) = SR(t_p)$.

Justificación. Todos los clientes están contenidos en T_p. □

Caso 3: dos o más subárboles tienen aristas de X

Afirmación 3.8. $Tour\text{-}SR(t) = \sum_{\substack{i=1 \\ X(t_i)>0}}^{k} (1 + Tour\text{-}SR(t_i) + 1)$.

Justificación. Un camino que realiza Tour-SR(t) comienza en t, y luego recorre, para cada subárbol T_i con clientes, un camino que realiza Tour-SR(t_i). □

Afirmación 3.9. $Start\text{-}SR(t) = \min_{\substack{1\le p\le k \\ X(t_p)>0}} (Tour\text{-}SR(t - T_p) + 1 + Start\text{-}SR(t_p))$, *donde* $Tour\text{-}SR(t - T_p) := Tour\text{-}SR(t) - (Tour\text{-}SR(t_p) + 2)$.

Justificación. Un camino que realiza Start-SR(t) comienza en t, luego, para cada subárbol T_i con clientes, excepto uno, digamos el T_p, recorre un camino que realiza Tour-SR(t_i), y finalmente recorre un camino que realiza Start-SR(t_p). □

Notar que las recurrencias para Tour-SR(t) y Start-SR(t) de este caso, subsumen a las del caso anterior.

Afirmación 3.10. $Through\text{-}SR(t) = \min\{Start\text{-}SR(t), \min_{\substack{1\le p\le k \\ X(t_p)>0}} (Through\text{-}SR(t_p)+1+Tour\text{-}SR(t - T_p) + 1), \min_{\substack{1\le p,q\le k \\ p\neq q \\ X(t_p)>0 \\ X(t_q)>0}} (Start\text{-}SR(t_p) + 1 + Tour\text{-}SR(t - T_p - T_q) + 1 + Start\text{-}SR(t_q))\}$, *donde* $Tour\text{-}SR(t - T_p - T_q) := Tour\text{-}SR(t) - (Tour\text{-}SR(t_p) + 2) - (Tour\text{-}SR(t_q) + 2)$.

Justificación. Un camino que realiza Through-SR(t) es de *una* de las siguientes formas:

- Empieza en t, y es un camino que realiza Start-SR(t).
- Termina en t, y el camino inverso realiza Start-SR(t).
- No empieza ni termina en t, y es de una de las siguientes formas:
  - Empieza y termina en T_p, un subárbol con clientes, y se obtiene tomando un camino que realiza Through-SR(t_p), e insertándole, justo después de alguna ocurrencia de t_p, un camino que realiza Tour-SR($t - T_p$).
  - Empieza en T_p y termina en T_q ($p \neq q$), ambos subárboles con clientes, y comienza recorriendo, en sentido inverso, un camino que realiza Start-SR(t_p), luego recorre un camino que realiza Tour-SR($t - T_p - T_q$), y finalmente recorre un camino que realiza Start-SR(t_q).

□

Afirmación 3.11. $SR(t) = Through\text{-}SR(t)$.

Justificación. Un camino que realiza $\texttt{SR}(t)$ debe pasar por t, porque hay dos o más subárboles de t que contienen clientes. □

Con todas estas fórmulas recursivas, estamos listos para dar un algoritmo que resuelve el problema.

Teorema 3.4. SR *sobre árboles está en* **P**. *Más aún, el problema se puede resolver en tiempo lineal en el tamaño del grafo de entrada.*

Demostración. Afirmamos que el Algoritmo 4 es correcto (esto es, calcula $SR^*(G, X)$ cuando G es un árbol), y se puede implementar de modo tal que corra en tiempo lineal. La correctitud se sigue de la validez de las recurrencias para `Tour-SR`, `Start-SR`, `Through-SR` y `SR`, y de que respetamos las dependencias entre ellas, i. e., usamos las ecuaciones en un orden que no produce dependencias cíclicas.

Algoritmo 4 Algoritmo para SR sobre árboles.

Entrada: $G = (V, E)$ un árbol y $X \subseteq E$.
Salida: $SR^*(G, X)$.

1: Sea r un vértice de G cualquiera.
2: Ordenar los vértices del árbol G jerárquicamente, tomando a r como raíz, haciendo una DFS desde r.
3: **for each** nivel del árbol, desde el último hasta el primero **do**
4: **for each** vértice t del nivel actual **do**
5: Sea t la raíz de T. Sean $t_1, \ldots, t_k$ las raíces de los subárboles de t. Sea e_i la arista entre t y t_i.
6: Calcular $\texttt{X}(t) = |\{e_1, \ldots, e_k\} \cap X| + \sum_{i=1}^{k} \texttt{X}(t_i)$.
7: Calcular $\texttt{Tour-SR}(t)$, $\texttt{Start-SR}(t)$, $\texttt{Through-SR}(t)$ y $\texttt{SR}(t)$, en ese orden, utilizando las recurrencias dadas, determinando el caso correspondiente según el valor de $\texttt{X}(t_i)$, para cada i.
8: **return** $\texttt{SR}(r)$

Para que corra en tiempo lineal, usamos programación dinámica, memorizando los resultados de `Tour-SR`, `Start-SR`, `Through-SR` y `SR` que vamos calculando. Utilizamos diccionarios con búsquedas e inserciones en $O(1)$. Específicamente, utilizamos arreglos indexados mediante los vértices de G, que los representamos como enteros entre 1 y $|V|$.

El costo del algoritmo está dado por el costo de la línea 2, más el de todas las ejecuciones de las líneas 6-7. La línea 2 cuesta $O(|V| + |E|)$, que es el costo de una DFS. Se puede ver que tales líneas se ejecutan exactamente una vez por cada vértice del árbol, pues los ciclos anidados recorren cada vértice de cada nivel del árbol. Notar que la cantidad k de subárboles que aparece en la línea 5, es

$$k = \begin{cases} d(t) & \text{si } t = r \\ d(t) - 1 & \text{si no} \end{cases}$$

con lo cual $k \leq d(t)$. Cada ejecución de la línea 6 es $O(1 + k)$ (sumamos 1, pues k puede valer 0, pero el costo siempre es positivo). El costo de la línea 7 es el costo de realizar los cómputos indicados por las ecuaciones de recurrencia. Es fácil ver que todas se pueden calcular en $O(1 + k)$, excepto la de la Afirmación 3.10, que contiene la siguiente expresión,

$$\min_{\substack{1\le p,q\le k\\ p\neq q\\ x(t_p)>0\\ x(t_q)>0}} (\texttt{Start-SR}(t_p) + 1 + \texttt{Tour-SR}(t - T_p - T_q) + 1 + \texttt{Start-SR}(t_q))$$

Como p y q recorren, cada una, un rango de $O(k)$ elementos, el cómputo naïve toma tiempo $O(1 + k^2)$. Podemos reescribir la expresión a minimizar del siguiente modo,

$$\begin{aligned}\texttt{Start-SR}(t_p)+1 + \texttt{Tour-SR}(t - T_p - T_q) + 1 + \texttt{Start-SR}(t_q) &= \\ &= \texttt{Start-SR}(t_p) + \texttt{Start-SR}(t_q) + 2 + \texttt{Tour-SR}(t) - \texttt{Tour-SR}(t_p) - \texttt{Tour-SR}(t_q) - 4 \\ &= \texttt{Tour-SR}(t) - \Delta_p - \Delta_q\end{aligned}$$

donde $\Delta_i = \texttt{Tour-SR}(t_i) - \texttt{Start-SR}(t_i) - 1$. Es $\Delta_i \geq 0$, o equivalentemente $\texttt{Start-SR}(t_i) \leq \texttt{Tour-SR}(t_i) - 1$, pues podemos obtener una solución factible de Start-SR para t_i quitándole la última arista a cualquier camino que realice $\texttt{Tour-SR}(t_i)$. Luego, el mínimo se obtiene para un par de índices p y q que maximicen Δ_p y Δ_q. Para calcularlos, simplemente iteramos sobre $1 \leq i \leq k$, y nos quedamos con los índices que realicen los dos valores más grandes de Δ_i, sujeto a $x(t_i) > 0$. Entonces, ésta y todas las demás ecuaciones se pueden calcular en tiempo $O(1 + k)$.

En definitiva, las líneas 6-7 son $O(1 + k)$, con lo cual, como $O(k) = O(d(t))$, todas las ejecuciones de esas líneas tienen un costo $O(\sum_{t\in V}(1 + d(t))) = O(|V| + |E|)$. En total, el costo del algoritmo es $O(|V| + |E|)$, como queríamos probar. □

Si bien el Algoritmo 4 calcula el valor de una solución óptima, demostrando así el teorema, no exhibe una. Sin embargo, es posible realizarle algunas modificaciones y agregados para darle esa capacidad, manteniendo el mismo tiempo de ejecución. La modificación consiste en almacenar junto con cada valor $\texttt{SR}(t)$, $\texttt{Start-SR}(t)$, $\texttt{Through-SR}(t)$ y $\texttt{Tour-SR}(t)$, una *marca* que indique cómo se construye una solución óptima que realice ese valor. Por ejemplo, en el caso de $\texttt{Start-SR}(t)$, tenemos dos recurrencias distintas, la de la Afirmación 3.1 y la Afirmación 3.9. Para la primera simplemente recordamos que la solución óptima es el camino $\langle t \rangle$, y para la segunda recordamos un valor de p para el que se realiza el mínimo. Terminada la ejecución del algoritmo de programación dinámica, todas las marcas necesarias estarán calculadas. Agregamos una función que, utilizando estas marcas, construya una solución óptima. Este procedimiento toma la raíz r de G y construye un camino, siguiendo las marcas y la estructura de una solución óptima, indicada por las recurrencias. Continuando con el ejemplo de $\texttt{Start-SR}(t)$, en el caso de la Afirmación 3.9 tomamos el valor de p dado por la marca guardada, calculamos recursivamente una solución óptima P_1 que realice $\texttt{Tour-SR}(t - T_p)$, una solución óptima P_2 que realice $\texttt{Start-SR}(t_p)$, y la respuesta es $P_1 \circ \langle t \rangle \circ P_2$.

Capítulo 4

Algoritmos exactos para STAR ROUTING

En este capítulo proponemos dos algoritmos exactos para SR general. Comenzamos el capítulo introduciendo el concepto de *función de acotación*, que nos permitirá acortar la ejecución de los algoritmos exactos.

4.1. Funciones de acotación

Los algoritmos exactos para problemas combinatorios suelen utilizar técnicas como backtracking. En este tipo de algoritmos la construcción de una solución suele ser un proceso ordenado que mantiene una solución parcial o incompleta, y en cada paso agrega algún elemento a la misma. Supongamos que estamos en el medio de la construcción de una solución para una instancia (G, X) de SR. Más específicamente, supongamos que ya construímos un camino P que cubre algunos de los clientes X. Nos interesa tener una cota inferior sobre cuánto nos costaría, en longitud, extender P a una solución factible de esta instancia de SR.

Usualmente, una función de acotación se definiría en este caso como una función B definida para cada camino P del grafo de una instancia arbitraria (G, X) de SR, tal que $B(P)$ es una cota inferior para el costo, en longitud, de extender P a una solución factible de (G, X). Vale notar que en nuestro caso, una función de acotación $B(P)$ depende, en general, sólo del último vértice de P y del conjunto de clientes que quedan por cubrir. Teniendo en cuenta esto, utilizamos una definición alternativa de este concepto, para que la función sólo dependa de un vértice y un subconjunto de aristas, y no de todo un camino. Esto tiene la ventaja de hacer más clara la notación, al concentrarse sólo en los elementos relevantes que intervienen en la función de acotación (y esto, a su vez, facilita el trabajo de idear funciones de acotación).

Definición 4.1. Sea (G, X) una instancia de SR. Una *función de acotación* es una función B definida para cada vértice u y cada subconjunto de aristas F de X, tal que $B(u, F)$ es una cota inferior para la longitud de una solución óptima de s-SR para (G, F, u).

A veces una función de acotación sólo depende de F, en cuyo caso omitimos el argumento u.

Notar que $\text{s-SR}^*(G, F, u) \geq \text{SR}^*(G, F)$ para todo F y u, pues SR admite soluciones factibles que empiezan en cualquier vértice, inclusive en u. Luego, si $\text{SR}^*(G, F) \geq B(F)$ para

todo F, entonces B es una función de acotación. Esto indica que podemos buscar funciones de acotación pensando en cotas inferiores para la solución óptima de una instancia (G, F) de SR.

Recíprocamente, supongamos que tenemos una cota $B(u, F)$. Observemos que siempre existe un vértice u_0 tal que $SR^*(G, F) = s\text{-}SR^*(G, F, u_0)$. Luego, $SR^*(G, F) \geq B(u_0, F)$, lo cual nos da una cota inferior para $SR^*(G, F)$. Por lo tanto, a partir de una cota podemos obtener una cota inferior para SR.

Durante el resto del capítulo, frecuentemente diremos *cota* en lugar de *función de acotación*. A continuación presentamos varias cotas, las cuales hemos clasificado en distintas clases, utilizando como criterio la similitud entre ellas.

4.1.1. Cotas con cubrimientos de vértices

Las siguientes cotas involucran el concepto de vertex cover mínimo.

Teorema 4.1. $B(F) = \tau(G[F]) - 1$ *es una cota.*

Demostración. El conjunto de vértices de cualquier solución factible P de SR para G y F, contiene un cubrimiento de vértices de $G[F]$, con lo cual $|P| \geq \tau(G[F])$. Luego, si además P es óptimo, tenemos $SR^*(G, F) = \text{length}(P) = |P| - 1 \geq \tau(G[F]) - 1$. □

Corolario 4.1. $SR^*(G, X) \geq \tau(G[X]) - 1$

Puesto que utilizaremos a las cotas como parte de un algoritmo exacto, es de nuestro interés que su cómputo sea eficiente. En el caso general, no sabemos computar la cota del Teorema 4.1 en tiempo polinomial, pues VC es **NP-completo** [13], aunque sí podemos hacerlo si G es un grafo grilla, pues en este caso G es bipartito, al igual que $G[F]$, por lo que podemos calcular $\tau(G[F])$ eficientemente, como indica el Teorema 1.4. De todos modos, no todo está perdido en el caso general. Como cualquier cota inferior sobre $\tau(G[F])$ nos da una nueva cota, si encontramos una buena cota inferior computable en tiempo polinomial, habremos obtenido una función de acotación polinomial. El siguiente lema exhibe dos posibles cotas inferiores, una de las cuales vale para grafos arbitrarios.

Lema 4.1. *Sea* H *un grafo con* m *aristas.*

1. *Si* H *es un subgrafo de un grafo grilla, entonces* $\tau(H) \geq \lceil m/4 \rceil$.
2. *Sea* $d_1 \geq \cdots \geq d_n$ *la secuencia de grados de* H. *Entonces* $\tau(H) \geq \min\{\ell : d_1 + \cdots + d_\ell \geq m\}$.

Demostración.

1. Como H es un subgrafo de un grafo grilla, cada vértice puede cubrir, como mucho, 4 aristas.
2. Notar que la cota $\tau(H) \geq \lceil m/4 \rceil$ sobrestima cuánto es capaz de cubrir cada vértice de H. Podemos hacer una estimación más precisa si tenemos en cuenta la secuencia de grados de H. Sea $d'_1 \geq \cdots \geq d'_{\tau(H)}$ la secuencia de grados de un vertex cover mínimo cualquiera de H. La secuencia (d'_i) es una subsecuencia de (d_i), y son ambas monótonas no crecientes, con lo cual es $d_i \geq d'_i$ para cada $1 \leq i \leq \tau(H)$. Por otro lado, como (d'_i) proviene de un vertex cover de H, vale $\sum_{i=1}^{\tau(H)} d'_i \geq m$. Entonces $\tau(H) \geq \min\{\ell : \sum_{i=1}^{\ell} d'_i \geq m\}$. Usando que $d_i \geq d'_i$, es $\min\{\ell : \sum_{i=1}^{\ell} d'_i \geq m\} \geq \min\{\ell : \sum_{i=1}^{\ell} d_i \geq m\}$. Por transitividad, llegamos a la desigualdad buscada.

□

Corolario 4.2.

1. *Si* G *es un grafo grilla, entonces* $B(F) = \lceil |F|/4 \rceil - 1$ *es una cota.*
2. *Sea* $d_1 \geq \cdots \geq d_n$ *la secuencia de grados de* $G[F]$*. Entonces* $B(F) = \min\{\ell : d_1 + \cdots + d_\ell \geq |F|\} - 1$ *es una cota.*

4.1.2. Cotas con distancias

Las siguientes cotas involucran el cómputo de distancias entre vértices y aristas, y entre pares de aristas. Primero, debemos introducir estas nociones de distancias.

Definición 4.2 (Distancia entre aristas)**.** Dadas dos aristas e y f de un grafo, definimos $\mathrm{dist}(e, f)$ como la distancia más pequeña entre un extremo de e y uno de f. Esto es,

$$\mathrm{dist}(e, f) = \min_{\substack{u \text{ extremo de } e \\ v \text{ extremo de } f}} \mathrm{dist}(u, v)$$

Definición 4.3. Sean e una arista y v un vértice de un grafo. Se define

$$\mathrm{dist}(e, v) = \mathrm{dist}(v, e) = \min_{u \text{ extremo de } e} \mathrm{dist}(u, v)$$

Teorema 4.2. $B(u, F) = \max_{\{e,f\} \subseteq F} (\min(\mathrm{dist}(u, e), \mathrm{dist}(u, f)) + \mathrm{dist}(e, f))$ *es una cota.*

Demostración. Tomemos dos aristas arbitrarias $e, f \in F$. Cualquier camino desde u que las cubra visitará a una de las dos en primer lugar (en caso de que ambas aristas sean cubiertas por primera vez por el mismo vértice, tomamos cualquiera de e o f). Desde u hasta ese punto, el camino tendrá longitud, al menos, $\min(\mathrm{dist}(u, e), \mathrm{dist}(u, f))$. Luego de eso, el camino eventualmente llegará a la segunda arista. La distancia recorrida entre ellos será al menos $\mathrm{dist}(e, f)$. En total, el camino tendrá longitud al menos $\min(\mathrm{dist}(u, e), \mathrm{dist}(u, f)) + \mathrm{dist}(e, f)$. Como e y f son cualesquiera, podemos tomar el máximo valor de esta longitud, sobre todos los e y f (es decir, nos quedamos con la mejor cota). □

Notemos que esta cota calcula cuál de los recorridos $u \rightsquigarrow e \rightsquigarrow f$ o $u \rightsquigarrow f \rightsquigarrow e$ es más corto, para cada par de aristas $e, f \in F$. Ésto se generaliza en forma natural a más de dos aristas de F.

Teorema 4.3.

$$\begin{aligned} B(u, F) = \max_{\{e,f,g\} \subseteq F} \min\{&\mathrm{dist}(u, e) + \mathrm{dist}(e, f) + \mathrm{dist}(f, g), \\ &\mathrm{dist}(u, e) + \mathrm{dist}(e, g) + \mathrm{dist}(g, f), \\ &\mathrm{dist}(u, f) + \mathrm{dist}(f, e) + \mathrm{dist}(e, g), \\ &\mathrm{dist}(u, f) + \mathrm{dist}(f, g) + \mathrm{dist}(g, e), \\ &\mathrm{dist}(u, g) + \mathrm{dist}(g, e) + \mathrm{dist}(e, f), \\ &\mathrm{dist}(u, g) + \mathrm{dist}(g, f) + \mathrm{dist}(f, e)\} \end{aligned}$$

es una cota.

Las siguiente cota es una especie de generalización de las dos anteriores, pero que no involucra un vértice u.

Teorema 4.4. $B(F) = PTSP^*(K(F), dist)$ *es una cota.*

Intuitivamente, esta cota es válida porque un camino que cubre un conjunto de clientes F puede pensarse como un camino en el grafo completo $K(F)$. La demostración se sigue directamente del Teorema 5.1. No la hacemos aquí porque utiliza un concepto que aún no hemos introducido (el de conjunto de permutaciones de clientes asociadas a un camino, que se presenta en el Capítulo 4).

Como sucedió antes, no sabemos computar esta cota en tiempo polinomial. En este caso, utilizamos la cota inferior $PTSP^*(K(F), dist) \geq MST(K(F), dist)$, que sí sabemos computar eficientemente.

Corolario 4.3. $B(F) = MST(K(F), dist)$ *es una cota.*

Demostración. Una solución factible de PTSP es un camino hamiltoniano, y un camino hamiltoniano es un árbol generador. Luego, toda solución factible tiene peso mayor o igual al peso de un árbol generador mínimo. □

4.1.3. Cotas combinadas

Las siguientes cotas combinan el concepto de vertex cover mínimo con el de distancias.

Definición 4.4. Sean H_1 y H_2 subgrafos de un grafo. Se define

$$dist(H_1, H_2) = \min_{\substack{u_1 \in V(H_1) \\ u_2 \in V(H_2)}} dist(u_1, u_2)$$

Teorema 4.5. *Sean* $H_1, \ldots, H_r$ *las componentes conexas de* $G[F]$. *Sea* $H = K(\{H_1, \ldots, H_r\})$. *Entonces* $B(F) = \tau(G[F]) - 1 + PTSP^*(H, dist) - (r-1)$ *es una cota.*

Demostración. Sea P una solución óptima de SR para (G, F). El camino P cubre cada arista de cada H_i. Más aún, cada vértice de P sólo puede cubrir aristas de sólo una componente H_i. Luego, P usa al menos

$$\sum_{i=1}^{r} \tau(H_i) = \tau(G[F]) \tag{4.1}$$

vértices para cubrir las aristas de las componentes conexas de $G[F]$.

Además de estos vértices que cubren las componentes, P recorre vértices intermedios, que no pertenecen a ningún H_i. A continuación, acotamos esta cantidad. Sean $i_1, \ldots, i_s$ tales que H_{i_j} es el j-ésimo subgrafo que visita P en su recorrido. Esto significa que P primero visita H_{i_1}, recorre algunos de sus vértices y eventualmente se va, para dirigirse hacia H_{i_2}. Así sucesivamente. Notar que una componente conexa podría ser visitada varias veces, así que podría aparecer varias veces en la secuencia.

Entre H_{i_j} y $H_{i_{j+1}}$ hay al menos $dist(H_{i_j}, H_{i_{j+1}}) - 1$ vértices que no pertenecen a ninguna componente. Por ende, en total hay al menos $\sum_{j=1}^{s-1}(dist(H_{i_j}, H_{i_{j+1}}) - 1)$ vértices de P que no pertenecen a ninguna componente de $G[F]$. Como $\langle H_{i_1}, \ldots, H_{i_s} \rangle$ es un camino de H que recorre todos sus vértices, y H es completo, podemos extraerle un camino hamiltoniano de H. Teniendo en cuenta este camino hamiltoniano, separamos los términos de la suma $\sum_{j=1}^{s-1}(dist(H_{i_j}, H_{i_{j+1}}) - 1)$ en dos, según la arista $\{H_{i_j}, H_{i_{j+1}}\}$ forme parte del camino hamiltoniano, o no. Hay $r-1$ términos correspondientes a aristas que están en el camino

hamiltoniano, y suman al menos $\mathsf{PTSP}^*(H, \mathrm{dist}) - (r-1)$, . Los demás términos suman una cantidad no negativa. Entonces

$$\sum_{j=1}^{s-1} (\mathrm{dist}(H_{i_j}, H_{i_{j+1}}) - 1) \geq \mathsf{PTSP}^*(H, \mathrm{dist}) - (r-1) \tag{4.2}$$

Como los vértices contados en (4.1) y (4.2) son disjuntos, P tiene al menos tantos vértices como la suma de esas cantidades. Luego

$$\mathsf{SR}^*(G, F) = \mathrm{length}(P) = |P| - 1 \geq \tau(G[F]) - 1 + \mathsf{PTSP}^*(H, \mathrm{dist}) - (r-1)$$

como queríamos probar.

□

Notar que como $\mathsf{PTSP}^*(H, \mathrm{dist}) \geq r-1$, esta cota es una generalización de la cota del Teorema 4.1. Como ya hicimos antes, el siguiente corolario acota PTSP por MST.

Corolario 4.4. $B(F) = \tau(G[F]) - 1 + \mathrm{MST}(H, \mathrm{dist}) - (r-1)$ *es una cota.*

En el caso en que G es grilla, podremos computar esta nueva cota en tiempo polinomial. En el caso general, la cota sigue siendo exponencial, y debemos utilizar una cota inferior polinomial para $\tau(G[F])$.

4.1.4. Otras cotas

La última cota que presentamos no encaja en ninguna de las categorías anteriores.

Teorema 4.6. *Si* G *es un grafo grilla, entonces* $B(F) = \lfloor |F|/3 \rfloor - 1$ *es una cota.*

Demostración. Llamemos $M(k)$ al mínimo valor de una solución óptima de SR para una instancia con k clientes, es decir,

$$M(k) = \mathrm{mín}\{\mathsf{SR}^*(G, F) : (G, F) \text{ es una instancia de SR sobre grillas con } |F| = k\}$$

Se tiene $\mathsf{SR}^*(G, F) \geq M(|F|)$, con lo cual toda cota inferior de $M(|F|)$ nos da una cota para SR. El resto de la demostración consiste en probar que $M(k) \geq \lfloor k/3 \rfloor - 1$.

Si $k < 3$, el resultado es trivial. Sea $k \geq 3$. Construiremos un conjunto de aristas F_k, pertenecientes a un grafo grilla (no importa cuál), tal que $M(k) = \mathsf{SR}^*(-, F_k)$, al mismo tiempo que probamos que $\mathsf{SR}^*(-, F_k) \geq \lfloor k/3 \rfloor - 1$.

Para ganar intuición sobre la forma de F_k, consideramos el conjunto de clientes de la Figura 4.1. Veamos que el camino indicado con flechas es una solución óptima de SR para esos clientes. El primer vértice cubre 4 clientes, y los restantes cubren 3 nuevos clientes cada uno. No hay otra forma de disponer esa misma cantidad de clientes, de modo tal que exista un camino más corto cubriéndolos a todos, puesto que cada vértice del camino maximiza la cantidad de *nuevos* clientes que son cubiertos.

La demostración es una generalización del ejemplo anterior. Primero, elegimos cierto conjunto F_k de k clientes, contenidos en un grafo grilla, y consideramos un camino P que cubra F_k. Esto es, P es una solución factible de SR para $(-, F_k)$. Luego, se prueba que *cualquier* solución factible de *cualquier* instancia de SR sobre grillas con k clientes, tiene longitud mayor o igual a $\mathrm{length}(P)$. Esto implica que $\mathsf{SR}^*(-, F_k) = \mathrm{length}(P) \leq \mathsf{SR}^*(G, F)$

para toda instancia (G, F) de SR sobre grillas con $|F| = k$, de lo cual se deduce que $M(k) = \text{length}(P)$.

Antes de continuar con la demostración, introducimos un concepto que formaliza la noción de cantidad de nuevos clientes que se van cubriendo a lo largo de un camino. Dado un grafo G, un subconjunto de aristas F de G, y un camino $P = \langle u_1, \ldots, u_r \rangle$ de G, la *secuencia de cobertura de* P *para* F es la secuencia de números $a_1, \ldots, a_r$, tal que si recorremos a P de principio a fin, a_i es el número de aristas de F que son cubiertos por u_i, y que no son cubiertos por ninguno de $u_1, \ldots, u_{i-1}$.

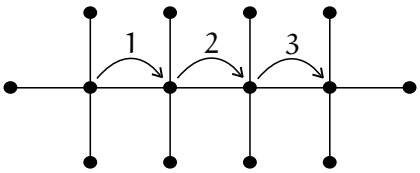


Figura 4.1: F_k con $k = 13$.

Consideramos tres casos, dependiendo del resto módulo 3 de k. Esta división en casos se debe a que no siempre es posible disponer los clientes exactamente como en la Figura 4.1.

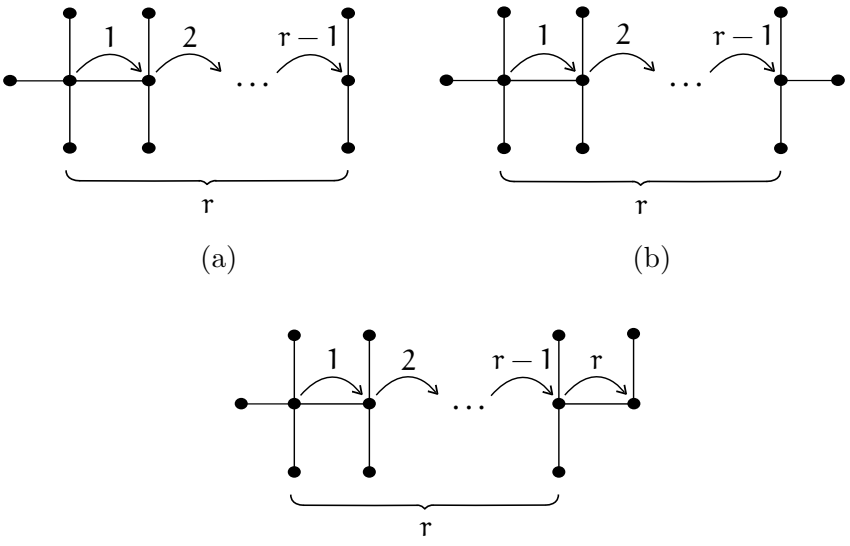


Figura 4.2: F_k con (a) $k = 3r$, (b) $k = 3r + 1$, (c) $k = 3r + 2$.

Si $k = 3r + 1$ para cierto r, elegimos F_k como indica la Figura 4.2-(b). Sea P el camino indicado con flechas. La secuencia de cobertura de P para F_k es $4, \underbrace{3, \ldots, 3}_{r-1 \text{ veces}}$. Llamemos $a_1, \ldots, a_r$ a esta secuencia. Sea (G, F) una instancia arbitraria de SR sobre grillas, y sea Q una solución factible de SR para (G, F). Sea $b_1, \ldots, b_s$, con $s = |Q|$, su secuencia de cobertura para F. Queremos ver que $\text{length}(P) \leq \text{length}(Q)$, o equivalentemente $r \leq s$.

Notar que $b_i \leq 4$ para todo i, puesto que G es un grafo grilla. Además $b_i \leq 3$ si $i > 1$, dado que un vértice puede cubrir sólo 4 clientes si y sólo si es el primer vértice en un camino y está rodeado por 4 clientes. Todo otro vértice distinto del primero no

puede cubrir 4 nuevos clientes, puesto que el vértice previo en el camino tuvo que haber cubierto alguno de los cuatro clientes incidentes. Entonces, tenemos que $b_i \leq a_i$ para todo $1 \leq i \leq \min\{r, s\}$. Como además las sumas $\sum_{i=1}^{r} a_i$ y $\sum_{i=1}^{s} b_i$ son iguales (a k), debe ser $r \leq s$, como queríamos.

En definitiva, si $k = 3r + 1$, es $M(k) = \text{length}(P) = r - 1 = \lfloor k/3 \rfloor - 1$. Resta probar la desigualdad para $k = 3r$ y $k = 3r+2$. Para $k = 3r+2$ elegimos F_k como indica la Figura 4.2-(c). El camino indicado P tiene secuencia de cobertura $4, \underbrace{3, \ldots, 3}_{r-1 \text{ veces}}, 1$, y se puede probar, igual que en el caso anterior, que es igual o más corto que cualquier solución factible de cualquier instancia sobre grillas con k clientes. En este caso, tenemos $\text{length}(P) = r$, de modo que $M(k) = \text{length}(P) = r = \lfloor k/3 \rfloor$ que obviamente cumple la desigualdad.

Finalmente, para $k = 3r$ consideramos el conjunto de clientes indicado en la Figura 4.2-(a). El camino considerado tiene longitud $r - 1$, obteniéndose también la desigualdad buscada. □

Corolario 4.5. *Si* G *es un grafo grilla,* $SR^*(G, X) \geq \lfloor |X|/3| \rfloor - 1$.

4.2. Algoritmos exactos

En esta sección presentamos dos algoritmos exactos para SR: uno basado en la técnica de *backtracking* y otro en la de *programación dinámica*. A lo largo de la sección consideraremos una instancia arbitraria (G, X) de SR y llamamos $k = |X|$ a la cantidad de clientes de la misma.

4.2.1. Algoritmo de backtracking

El algoritmo de backtracking que proponemos considera todas las posibles permutaciones de las aristas de X. La idea detrás de esto es que toda solución factible de SR cubre los clientes en cierto orden. El algoritmo produce todos esos ordenamientos, ya que alguno de ellos corresponde al orden en que alguna solución óptima visita las aristas. Surgen algunos interrogantes, que iremos respondiendo. ¿Dada una solución factible, cuál es el orden en que se cubren los clientes? ¿Es único este orden?

Definición 4.5. Sea F un subconjunto de aristas de G. Sea $P = \langle u_1, \ldots, u_r \rangle$ un camino de G que cubre F. Definimos el *conjunto de permutaciones de* F *asociadas a* P, y lo notamos $\Pi(F, P)$, del siguiente modo. Una permutación $\langle e_1, \ldots, e_\ell \rangle$ de F pertenece a $\Pi(F, P)$ si existe una función monótona no decreciente $f : \{1, \ldots, \ell\} \to \{1, \ldots, r\}$ tal que $u_{f(i)}$ cubre a e_i, para cada $1 \leq i \leq \ell$.

Intuitivamente, $\pi \in \Pi(F, P)$ si P puede cubrir F en el orden dado por π.

Dada una permutación $\pi = \langle e_1, \ldots e_k \rangle$ de X, nos interesa saber cuál es la mínima longitud de un camino P tal que $\pi \in \Pi(X, P)$. La llamaremos *longitud mínima de* π, y la notamos $L(\pi)$. Si no existe tal camino P, definimos $L(\pi) = \infty$. Antes de ver cómo computar $L(\pi)$, motivemos la utilidad de este número.

Proposición 4.1. $\Pi(F, P) \neq \emptyset$

Demostración. Escribamos $P = \langle u_1, \ldots, u_r \rangle$. Iterando, de principio a fin, sobre cada vértice u_i de P, vamos enumerando las aristas de F que son cubiertos por u_i y que no habían sido

cubiertas aún. Si u_i cubre más de una nueva arista, las enumeramos en cualquier orden. Sea $\pi = \langle e_1, \ldots, e_\ell \rangle$ el resultado de esta enumeración. Observemos que toda arista de F está en π, pues P cubre F. Luego, π es una permutación de los aristas de F.

Para ver que $\pi \in \Pi(F, P)$, debemos encontrar una función f tal como indica la definición de $\Pi(F, P)$. Definimos, para cada $1 \leq i \leq \ell$, el número $f(i)$ como el índice del primer vértice de P que cubre e_i. Veamos que esta f cumple las condiciones necesarias. Obviamente, $u_{f(i)}$ cubre e_i. Además, como π es resultado de ir enumerando las aristas exactamente cuando son cubiertos por primera vez por P, es $f(i) \leq f(i+1)$. □

Observación 4.1. Si P es una solución factible de SR para (G, X) y $\pi \in \Pi(X, P)$, entonces $L(\pi) \leq \text{length}(P)$.

Diremos que una solución factible P de SR para (G, X) realiza $L(\pi)$, si $\pi \in \Pi(X, P)$ y $\text{length}(P) = L(\pi)$.

Lema 4.2. *Entre todas las permutaciones de X, sea $\pi_{\min}$ una tal que $L(\pi_{\min})$ es mínimo. Sea P una solución factible de SR para (G, X), que realiza $L(\pi_{\min})$. Entonces P es una solución óptima.*

Demostración. Sea Q una solución óptima de SR para (G, X). Por la Proposición 4.1, es $\Pi(X, Q) \neq \emptyset$, así que podemos tomar un elemento $\pi_{\text{opt}} \in \Pi(X, Q)$, cualquiera. Se tiene $L(\pi_{\min}) \leq L(\pi_{\text{opt}})$. Luego

$$\text{length}(P) = L(\pi_{\min}) \leq L(\pi_{\text{opt}}) \leq \text{length}(Q)$$

Como P es una solución factible, es $\text{length}(Q) \leq \text{length}(P)$. Luego, debe ser $\text{length}(P) = \text{length}(Q)$, o sea que P es una solución óptima. □

Corolario 4.6. $SR^*(G, X) = \text{mín}\{L(\pi) : \pi \text{ permutación de } X\}$

Estos resultados sugieren que podemos encontrar el valor de una solución óptima de SR para (G, X), buscando el mínimo $L(\pi)$. Veamos cómo calcular $L(\pi)$, dada una permutación π. Escribamos $\pi = \langle e_1, \ldots, e_k \rangle$. La Figura 4.3 ayuda a hacernos una idea gráfica del problema que intentamos resolver. Los trazos en líneas punteadas representan un camino entre los correspondientes vértices. Intuitivamente, un camino P es tal que $\pi \in \Pi(X, P)$ si y sólo si recorre las líneas punteadas, pasando por cada e_i. Buscamos la mínima longitud de un camino con estas características.

Durante el resto de este capítulo, asumimos que los dos extremos de toda arista e están enumerados de cierta forma (es decir, que e es un par ordenado de vértices), y los llamamos $e[1]$ y $e[2]$.

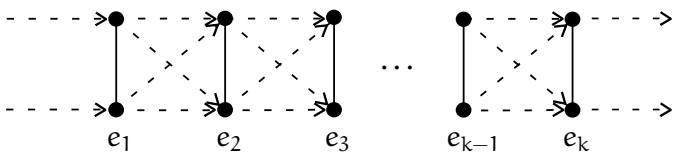


Figura 4.3: Calcular $L(\pi)$ puede pensarse como encontrar la mínima longitud de un camino que recorre las líneas punteadas, pasando por cada e_i. (Notar que los extremos de las aristas e_i no son necesariamente disjuntos, con lo cual algunos de los trazos punteados podrían tener longitud 0.)

Fijada una permutación $\pi = \langle e_1, \ldots, e_k \rangle$ de X, llamamos $\pi_i = \langle e_1, \ldots, e_i \rangle$ y $X_i = \{e_1, \ldots, e_i\}$. Definimos

$$L_1(i) = \text{mínima longitud de un camino } P \text{ tal que } \pi_i \in \Pi(X_i, P) \text{ y que termina en } e_i[1]$$

Análogamente se define $L_2(i)$, que requiere que el camino P termine en $e_i[2]$. Notemos que $L(\pi) = \min\{L_1(k), L_2(k)\}$ es el número buscado. Es fácil ver que $L_1(1) = 0$, pues el camino $\langle e_1[1] \rangle$ realiza el mínimo. Análogamente, $L_2(1) = 0$.

Diremos que un camino P de G que cubre X_i realiza $L_j(i)$, si $\pi_i \in \Pi(X_i, P)$, P termina en $e_i[j]$, y $\text{length}(P) = L_j(i)$.

Teorema 4.7.

$$L_1(i) = \min\{L_1(i-1) + \text{dist}(e_{i-1}[1], e_i[1]), L_2(i-1) + \text{dist}(e_{i-1}[2], e_i[1])\}$$
$$L_2(i) = \min\{L_1(i-1) + \text{dist}(e_{i-1}[1], e_i[2]), L_2(i-1) + \text{dist}(e_{i-1}[2], e_i[2])\}$$

Demostración. Probaremos la recurrencia para L_1; la de L_2 es análoga.

Sea $P = \langle u_1, \ldots, u_r \rangle$ un camino que realiza $L_1(i)$. Como $\pi_i \in \Pi(X_i, P)$, existe una función monótona no decreciente f tal que $u_{f(j)}$ cubre e_j, para cada $1 \leq j \leq i$. Observemos que $r = f(i)$, pues como el subcamino $\langle u_1, \ldots, u_{f(i)} \rangle$ realiza $L_1(i)$, no puede ser más corto que P.

Partimos a P en dos subcaminos, $P_1 = \langle u_1, \ldots, u_{f(i-1)} \rangle$ y $P_2 = \langle u_{f(i-1)}, \ldots, u_{f(i)} \rangle$. Como $P = P_1 \circ P_2$, tenemos que $\text{length}(P) = \text{length}(P_1) + \text{length}(P_2)$. Observemos que es $u_{f(i-1)} = e_{i-1}[1]$ o $u_{f(i-1)} = e_{i-1}[2]$, pues $u_{f(i-1)}$ cubre e_{i-1}.

Supongamos que $u_{f(i-1)} = e_{i-1}[1]$. Probaremos que $\text{length}(P) = L_1(i-1) + \text{dist}(e_{i-1}[1], e_i[1])$.

Afirmación 4.1. $\text{length}(P_1) = L_1(i-1)$

Demostración. Supongamos, para llegar a un absurdo, que $\text{length}(P_1) \neq L_1(i-1)$. Debe ser $\text{length}(P_1) > L_1(i-1)$, pues P_1 es un camino que cumple los requerimientos de la definición de $L_1(i-1)$. Pero entonces podemos tomar un camino P_1' que sí realice $L_1(i-1)$, y concatenarle un camino mínimo P_2' entre $e_{i-1}[1]$ y $u_{f(i)}$. Escribamos $P_1' = \langle v_1, \ldots, v_s = e_{i-1}[1] \rangle$ y $P_2' = \langle e_{i-1}[1] = v_s, \ldots, v_t = u_{f(i)} \rangle$. Llamemos $P' = P_1' \circ P_2'$ a este nuevo camino.

Veamos que P' cumple los requerimientos de la definición de $L_1(i)$. Por un lado, como P_1' realiza $L_1(i-1)$, existe una función monótona no decreciente g tal que $v_{g(j)}$ cubre a e_j, para cada $1 \leq j \leq i-1$. Extendemos g, definiendo $g(i) = t$, de modo tal que $v_{g(i)} = v_t = u_{f(i)}$ cubre a e_i. La existencia de esta función implica que $\pi_i \in \Pi(X_i, P')$. Por otro lado, P' termina en $e_i[1]$, pues $v_t = u_{f(i)} u_r$, que es el último vértice de P, camino que termina en $e_i[1]$, puesto que realiza $L_1(i)$.

Más aún, P' es más corto que P, pues

$$\begin{aligned}
\text{length}(P') &= \text{length}(P_1') + \text{length}(P_2') \\
&= L_1(i-1) + \text{dist}(e_{i-1}[1], u_{f(i)}) \\
&< \text{length}(P_1) + \text{dist}(e_{i-1}[1], u_{f(i)}) && (L_1(i-1) < \text{length}(P_1)) \\
&= \text{length}(P_1) + \text{dist}(u_{f(i-1)}, u_{f(i)}) && (e_{i-1}[1] = u_{f(i-1)}) \\
&\leq \text{length}(P_1) + \text{length}(P_2) \\
&= \text{length}(P)
\end{aligned}$$

Esto implica que $L_1(i) \leq \text{length}(P') < \text{length}(P)$, lo cual contradice la suposición de que $L_1(i) = \text{length}(P)$. El absurdo provino de suponer que $\text{length}(P_1) \neq L_1(i-1)$. □

Afirmación 4.2. $\text{length}(P_2) = \text{dist}(e_{i-1}[1], e_i[1])$

Demostración. El subcamino $P_2 = \langle u_{f(i-1)}, \ldots, u_{f(i)} \rangle$ debe ser un camino mínimo entre $u_{f(i-1)}$ y $u_{f(i)}$, pues de lo contrario podemos tomar un camino entre estos dos vértices que sí sea mínimo, concatenarlo a P_1, y obtener un camino que realice $L_1(i)$, y que sea más corto que P. Luego, es $\text{length}(P_2) = \text{dist}(u_{f(i-1)}, u_{f(i)})$.

Veamos que $u_{f(i-1)} = e_{i-1}[1]$ y $u_{f(i)} = e_i[1]$. Lo primero es una hipótesis. Lo segundo se desprende de que P termina en $e_i[1]$, con lo cual $u_{f(i)} = e_i[1]$. □

Estas dos afirmaciones implican directamente que $\text{length}(P) = L_1(i-1) + \text{dist}(e_{i-1}[1], e_i[1])$. El caso $u_{f(i-1)} = e_{i-1}[2]$ es análogo, y se prueba que $\text{length}(P) = L_2(i-1) + \text{dist}(e_{i-1}[2], e_i[1])$. Sea cual sea el caso, es $\text{length}(P) \in \{L_1(i-1) + \text{dist}(e_{i-1}[1], e_i[1]), L_2(i-1) + \text{dist}(e_{i-1}[2], e_i[1])\}$. Como P tiene longitud mínima, y ambos valores representan longitudes de caminos que cumplen con los requerimientos de la definición de $L_1(i)$, $L_1(i) = \text{length}(P)$ debe ser el mínimo de los dos. □

A partir de este marco teórico, describimos el algoritmo de backtracking, que es el Algoritmo 5. En él se utilizan variables globales para almacenar la entrada del mismo y algunos valores relacionados con la mejor solución encontrada hasta el momento. Específicamente, estas variables son:

- (G, X) la instancia de SR a resolver.
- `opt_permut` la permutación con la longitud mínima más pequeña encontrada hasta el momento. Se inicializa en NIL.
- `opt_length` la longitud mínima de `opt_permut`. Se inicializa en ∞.

Una vez inicializadas estas variables, el problema se resuelve llamando a la función SR-Backtracking-Solver. Esta función realiza múltiples llamadas a SR-Backtracking, la función de backtracking propiamente dicha, y luego ejecuta Build-Path-Backtracking, que devuelve una solución óptima de SR para (G, X), a partir de lo computado por SR-Backtracking y registrado en las variables globales.

Algoritmo 5

Salida: Una solución óptima de SR para (G, X).

```
1: SR-Backtracking-Solver()
2:     Sea F una copia de X.
3:     for each e ∈ X do
4:         F ← F − {e}
5:         SR-Backtracking(⟨e⟩, F, 0, 0)
6:         F ← F ∪ {e}
7:     return Build-Path-Backtracking()
```

El conjunto de llamadas a SR-Backtracking produce cada una de las permutaciones de X. La función recibe una permutación parcialmente construida π, el conjunto F de

Algoritmo 6 Algoritmo de backtracking para SR.

Entrada: La permutación π hasta el momento, el conjunto F de clientes que resta cubrir, y ℓ_1 y ℓ_2 los valores de L_1 y L_2, respectivamente, sobre los extremos de la última arista agregado a π.

```
1:  SR-Backtracking(π, F, ℓ1, ℓ2)
2:      if |F| = 0 then
3:          if mín{ℓ1, ℓ2} < opt_length then
4:              opt_length ← mín{ℓ1, ℓ2}
5:              Escribir en opt_permut una copia de π.
6:          return
7:      Sea e la última arista de π.
8:      for each f ∈ F do
9:          ℓ'1 ← mín{ℓ1 + dist(e[1], f[1]), ℓ2 + dist(e[2], f[1])}
10:         ℓ'2 ← mín{ℓ1 + dist(e[1], f[2]), ℓ2 + dist(e[2], f[2])}
11:         π ← π ∘ ⟨f⟩
12:         F ← F − {f}
13:         SR-Backtracking(π, F, ℓ'1, ℓ'2)
14:         Quitar el último elemento de π.
15:         F ← F ∪ {f}
```

clientes que resta cubrir, y ℓ_1 y ℓ_2 los valores de L_1 y L_2 para la última arista agregada a π, respectivamente. Comienza verificando, en la línea 2, si ya terminó de cubrir todos los clientes. De ser así, compara la longitud mínima de la permutación π producida, que es mín$\{\ell_1, \ell_2\}$, contra la longitud mínima más pequeña calculada hasta el momento. Actualiza las variables globales en caso de haber encontrado una mejor solución.

Si no estamos en el caso base, el algoritmo toma las aristas de F, y prueba poniendo a cada una de ellas como la siguiente arista de la permutación π, realizando una llamada recursiva por cada elección. En las líneas 9-10, utiliza las ecuaciones del Teorema 4.7 para calcular los valores de L_1 y L_2 sobre los extremos de la arista agregada a la permutación.

La función Build-Path-Backtracking parece compleja a primera vista, pero su lógica es simple. La idea es, a partir de la permutación opt_permut calculada por SR-Backtracking, construir un camino que realice su longitud mínima. Para esto, repite el cómputo de L_1 y L_2, utilizando las ecuaciones del Teorema 4.7, pero llevando registro de los vértices que nos llevan a la longitud mínima. Para esto se definen dos arreglos, $\texttt{pred}_1$ y $\texttt{pred}_2$, de modo tal que en $\texttt{pred}_j[i]$ almacenamos el extremo de la $(i-1)$-ésima arista de opt_permut que realiza $L_j(i)$. En otras palabras, el extremo de la $(i-1)$-ésima arista para el cual se obtiene el mínimo de la recurrencia de L_j del Teorema 4.7. Esto es lo que hacen las líneas 5-15 del algoritmo. Luego del ciclo 5-15, las líneas 16-19 eligen un extremo del último vértice de la permutación, que realiza la longitud mínima. A partir de allí, las líneas 20-24 construyen un camino, de atrás hacia adelante, utilizando la información recabada en $\texttt{pred}_1$ y $\texttt{pred}_2$.

Observemos que las líneas 9-10 de SR-Backtracking utilizan la distancia entre vértices de $G[X]$. Por esto, se asume que se puede obtener dist(u, v), para cada $u, v \in G[X]$, en tiempo $O(1)$. Estas distancias se pueden precomputar, haciendo una BFS sobre G desde cada vértice de $G[X]$, almacenando las distancias resultantes en una matriz. Para ciertos grafos de entrada G, estas distancias se puede obtener en $O(1)$, aún sin haberlas pre-

Algoritmo 7 Construcción de una solución óptima.

Salida: Una solución óptima de SR para (G, X).

```
1:  BUILD-PATH-BACKTRACKING()
2:      Sean pred_1 y pred_2 arreglos de k posiciones.
3:      pred_1[1], pred_2[1] ← NIL
4:      ℓ_1, ℓ_2 ← 0
5:      for i ← 2, ..., k do
6:          e ← opt_permut[i − 1]
7:          f ← opt_permut[i]
8:          for j ← 1, 2 do
9:              ℓ'_j ← mín{ℓ_1 + dist(e[1], f[j]), ℓ_2 + dist(e[2], f[j])}
10:             if ℓ_1 + dist(e[1], f[j]) ≤ ℓ_2 + dist(e[2], f[j]) then
11:                 pred_j[i] ← 1
12:             else
13:                 pred_j[i] ← 2
14:         ℓ_1 ← ℓ'_1
15:         ℓ_2 ← ℓ'_2
16:     if ℓ_1 ≤ ℓ_2 then
17:         j ← 1
18:     else
19:         j ← 2
20:     P ← ⟨⟩
21:     for i ← k, ..., 1 do
22:         e ← opt_permut[i]
23:         P ← ⟨e[j]⟩ ∘ P
24:         j ← pred_j[i]
25:     return P
```

computado. Este es el caso de los grafos grilla representados como puntos de coordenadas enteras del plano, en los cuales la distancia entre dos vértices $u = (x_1, y_1)$ y $v = (x_2, y_2)$ es $\text{dist}(u, v) = |x_1 - x_2| + |y_1 - y_2|$.

Se asume que los argumentos de todas las funciones involucradas se pasan por referencia. Esto evita desperdiciar tiempo y espacio para la copia de esos valores, que es innecesaria.

Por último, notar que la respuesta de Build-Path-Backtracking (y, por lo tanto, también la de SR-Backtracking-Solver) no es un camino propiamente dicho, sino una secuencia de vértices no necesariamente adyacentes en G. Una solución óptima legítima se obtiene, a partir de esta respuesta, tomando cada par de vértices consecutivos y trazando un camino mínimo entre ellos. La ventaja de dar una respuesta con "huecos", es que mantiene la complejidad del algoritmo independiente del tamaño del grafo, pues la respuesta siempre tiene k elementos.

Teorema 4.8.

1. *El Algoritmo 5 es correcto.*
2. *Corre en tiempo* $O(k \cdot k!)$.
3. *Usa* $O(k)$ *memoria adicional a la entrada* (G, X).

Demostración.

1. El conjunto de llamadas a SR-Backtracking calcula todas las permutaciones de X. Por el Corolario 4.6, la longitud mínima más pequeña que se encuentre será la longitud de una solución óptima de SR para (G, X). La función SR-Backtracking calcula la longitud mínima de cada permutación producida, en base a las ecuaciones del Teorema 4.7, y se queda con una que tenga longitud mínima lo más chica posible.

   La función Build-Path-Backtracking construye un camino que realiza la longitud mínima de la permutación óptima `opt_permut`.

   La función SR-Backtracking-Solver ejecuta todas las llamadas a SR-Backtracking necesarias para producir todas las permutaciones de X, y luego devuelve la solución óptima que construye Build-Path-Backtracking.

2. El costo del algoritmo está dado por el costo de las llamadas a SR-Backtracking que realiza SR-Backtracking-Solver en el ciclo 3-6, más el costo de la llamada a Build-Path-Backtracking. Los calculamos por separado.

   Consideremos el árbol de recursión de las llamadas a SR-Backtracking, que se presenta en la Figura 4.4. La raíz del árbol representa la ejecución de SR-Backtracking-Solver, que realiza k llamadas a SR-Backtracking. Para evitar confusiones, usaremos el término *nodo* para referirnos a los vértices del árbol de ejecución. Cada nodo, excepto la raíz, representa una instancia de SR-Backtracking, y tiene un costo asociado. Luego, el costo de todas las llamadas a SR-Backtracking está dado por la suma de los costos asociados a los nodos, exceptuando la raíz. Los nodos internos, salvo la raíz, representan casos recursivos de SR-Backtracking, mientras que las hojas representan casos base.

   Las hojas tienen un costo $O(k)$, pues ejecutan, en particular, la línea 5, que copia la permutación π generada, con un costo $O(k)$. Como hay $k!$ permutaciones de X, y

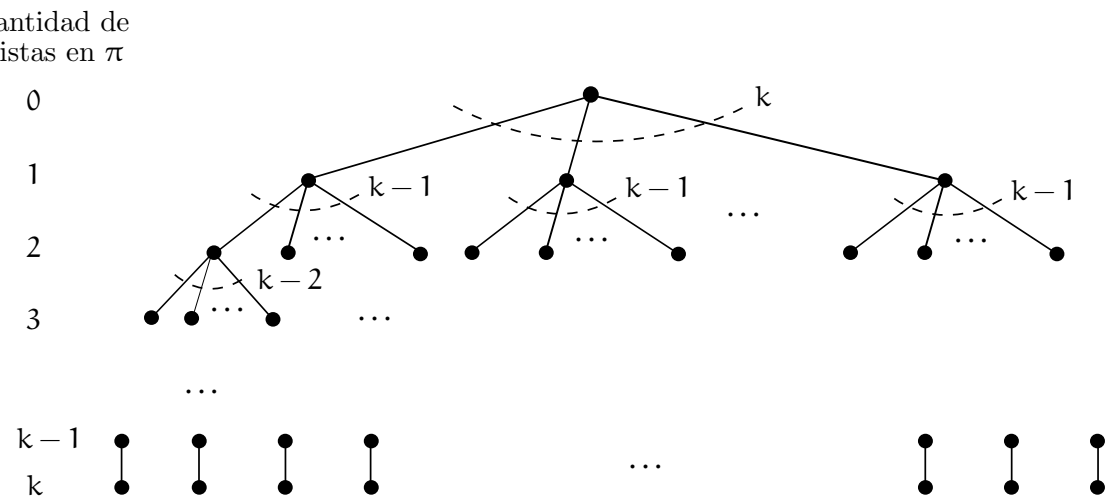


Figura 4.4: Árbol de recursión de las ejecuciones de SR-Backtracking. La raíz representa la llamada a SR-Backtracking-Solver.

hay exactamente una hoja por cada una de ellas, son $k!$ hojas en total. Luego, las hojas aportan un costo $O(k \cdot k!)$.

Los nodos internos, salvo la raíz, tienen un costo $O(1)$, pues esas instancias ejecutan las líneas 2 y 7-15, que son todas $O(1)$. Se puede ver que la cantidad de nodos internos, sacando la raíz, es

$$N(k) = k + k(k-1) + \cdots + k(k-1)\ldots 2$$

pues esta es la suma de la cantidad de nodos de todos los niveles, excepto la raíz y las hojas. La siguiente afirmación provee una estimación asintótica de $N(k)$.

Afirmación 4.3. $k! \leq N(k) < 2k!$

Demostración. Es evidente que $k! \leq N(k)$, pues el último término de $N(k)$ es $k!$. Veamos la otra desigualdad. Reescribimos $N(k)$ como sigue,

$$\begin{aligned} N(k) &= k + k(k-1) + \cdots + k(k-1)\ldots 2 \\ &= \frac{k!}{(k-1)!} + \frac{k!}{(k-2)!} + \cdots + \frac{k!}{1!} \\ &= k!\left(\frac{1}{(k-1)!} + \frac{1}{(k-2)!} + \cdots + \frac{1}{1!}\right) \end{aligned}$$

Como $1/n! \leq 1/2^{n-1}$ para todo $n \in \mathbb{N}$, es

$$N(k) \leq k!\left(\frac{1}{2^{k-2}} + \frac{1}{2^{k-3}} + \cdots + \frac{1}{2^0}\right)$$

Usando que $\frac{1}{2^{k-2}} + \frac{1}{2^{k-3}} + \cdots + \frac{1}{2^0} < 2$, llegamos a la desigualdad deseada.

□

En definitiva, hay $\Theta(k!)$ nodos internos, sacando a la raíz. Como el costo de todo nodo interno es $O(1)$, el costo total de los mismos es $O(k!)$. Luego, la suma de los costos de todos los nodos, salvo la raíz, es $O(k \cdot k!)$.

La función BUILD-PATH-BACKTRACKING es claramente $O(k)$, costo que es absorbido por la complejidad de SR-BACKTRACKING. Sumando todo, llegamos a que el costo de SR-BACKTRACKING-SOLVER es $O(k \cdot k!)$.

3. Las variables globales, sacando la entrada del problema, usan memoria $O(k)$. El resto de la memoria utilizada corresponde a la memoria local utilizada por cada función.

   SR-BACKTRACKING-SOLVER usa $O(k)$ memoria local, pues copia el conjunto X en la línea 2. Cada instancia de SR-BACKTRACKING utiliza $O(1)$ memoria local. Notemos que hay $O(k)$ instancias de esa función ejecutándose en un momento dado, pues esa es la profundidad del árbol de recursión de las llamadas a SR-BACKTRACKING. Entonces SR-BACKTRACKING usa $O(k)$ memoria local, entre todas sus instancias en ejecución. Por último, BUILD-PATH-BACKTRACKING utiliza $O(k)$ memoria local para almacenar los arreglos $\texttt{pred}_1$ y $\texttt{pred}_2$, y el camino construido P. En total, se utiliza $O(k)$ memoria local en un momento cualquiera de la ejecución.

   Sumando la memoria local y la global, es $O(k)$ memoria adicional.

□

Una característica notable de este algoritmo es que su complejidad sólo depende de k, la cantidad de clientes, y no del tamaño del grafo de entrada. Esto permite que el algoritmo pueda completar la ejecución en un tiempo razonable, inclusive para instancias de grafos grandes. Como SR nace como un problema de ruteo en ciudades, resulta importante no depender del tamaño de las mismas, que pueden ser muy grandes.

Se puede ver que este mismo algoritmo sirve también para una versión de SR en el que el grafo G tenga pesos. El algoritmo no utiliza el hecho de que los pesos sean unitarios y, más aún, tampoco utiliza información sobre la topología de G. Toda la información que necesita es la distancia entre cada par de vértices de $G[X]$.

4.2.2. Algoritmo de programación dinámica

Consideremos un punto arbitrario en la ejecución de SR-BACKTRACKING, en el que hay una permutación π construida parcialmente, y queda un subconjunto de clientes F por cubrir. Sea e la última arista agregada a π. La idea clave es que la mejor forma extender π a una permutación óptima, sólo depende de la arista actual e y los clientes restantes F, por lo que cualquier otra permutación parcial π' cuya última arista sea e y tal que deje el conjunto de clientes F sin cubrir, puede ser extendida a una permutación óptima de la misma forma que π. Luego, sólo es necesario determinar la mejor extensión a partir de e, que cubra F, una única vez. Para llevar a la práctica esta idea, necesitamos formular el problema en función de un vértice inicial y un subconjunto de aristas a cubrir.

Consideremos

$$J(u, F) = \text{ mínima longitud de un camino que empieza en } u \text{ y cubre } F$$

Diremos que un camino P de G realiza $J(u, F)$, si es un camino que empieza en u y cubre F, y $\text{length}(P) = J(u, F)$.

Observación 4.2. $SR^*(G, X) = \min_{e \in X} \min\{J(e[1], X - \{e\}), J(e[2], X - \{e\})\}$

Teorema 4.9.

$$J(u, F) = \min_{e \in F} \min\{\text{dist}(u, e[1]) + J(e[1], F - \{e\}), \text{dist}(u, e[2]) + J(e[2], F - \{e\})\}$$

Demostración. Sea $P = \langle u_1, \ldots, u_r \rangle$ un camino que realiza $J(u, F)$. Consideremos el mínimo índice i, tal que u_i cubre una arista de F. Sea e un cliente de F cualquiera cubierto por u_i. Entonces $u_i = e[1]$ o $u_i = e[2]$.

Supongamos que $u_i = e[1]$, y probemos que $\text{length}(P) = \text{dist}(u, e[1]) + J(e[1], F - \{e\})$. Partimos a P en dos subcaminos, $P_1 = \langle u_1, \ldots, u_i \rangle$ y $P_2 = \langle u_i, \ldots, u_r \rangle$.

Afirmación 4.4. $\text{length}(P_1) = \text{dist}(u, e[1])$

Demostración. Los vértices $u_1, \ldots, u_{i-1}$ no cubren aristas de F, con lo cual podemos reemplazar, en P, el subcamino P_1 por cualquier otro camino entre $u = u_1$ y u_i, y seguiremos teniendo un camino que empieza en u, y que cubre F. En particular, si P_1 no fuera un camino mínimo entre u_1 y u_{i-1}, lo podríamos reemplazar por uno que sí lo fuera, obteniéndose un camino más corto, y por ende contradiciendo la minimalidad de P. Luego, debe ser $\text{length}(P_1) = \text{dist}(u_1, u_i) = \text{dist}(u, e[1])$. □

Afirmación 4.5. $\text{length}(P_2) = J(e[1], F - \{e\})$

Demostración. Como $u_1, \ldots u_{i-1}$ no cubren aristas de F, son $u_i, \ldots, u_r$ los que cubren todos las aristas de F. Esto es, P_2 cubre F. Esto prueba que $\text{length}(P_2) \geq J(e[1], F - \{e\})$. No puede ser $\text{length}(P_2) > J(e[1], F - \{e\})$, pues esto implicaría que podemos reemplazar a P_2, en P, por un camino más corto que empiece en $e[1]$ y cubra $F - \{e\}$, y obtendríamos un camino que cumple los requerimientos de $J(u, F)$, pero más corto que P, lo cual es absurdo. □

Como $\text{length}(P) = \text{length}(P_1) + \text{length}(P_2)$, estas afirmaciones implican que $\text{length}(P) = \text{dist}(u, e[1]) + J(e[1], F - \{e\})$, como queríamos probar. El caso $u_i = e[2]$ es análogo, y se prueba que $\text{length}(P) = \text{dist}(u, e[2]) + J(e[2], F - \{e\})$.

En cualquier caso, es $\text{length}(P) \in \{\text{dist}(u, e[1]) + J(e[1], F - \{e\}), \text{dist}(u, e[2]) + J(e[2], F - \{e\})\}$. Para terminar, notemos que como un tal camino óptimo P no es conocido a priori (y, por lo tanto, tampoco e), el valor de $J(u, F)$ se obtiene tomando mínimos, tal como en la ecuación del enunciado. □

El algoritmo de programación dinámica para SR que proponemos es el Algoritmo 8. Este programa utiliza variables globales para almacenar la entrada del algoritmo, los valores de J que se van calculando, y otra información relativa a la mejor solución encontrada hasta el momento. Específicamente:

- (G, X) la instancia de SR a resolver.
- `opt` un diccionario con todos los valores calculados de J. Se inicializa como un diccionario vacío.
- `suc` un diccionario utilizado para reconstruir una solución óptima. Lo explicaremos más adelante. Se inicializa como un diccionario vacío.

- $\texttt{opt\_length}$ la mínima longitud de una solución factible de SR para (G, X) encontrada hasta el momento. Se inicializa en ∞.
- $\texttt{opt\_start}$ una tupla (e, j), con $e \in \mathsf{X}$ y $j \in \{1, 2\}$, tal que $\texttt{opt\_length}$ se realiza para un camino que comienza en $e[j]$. Se inicializa en NIL.

Luego de inicializar las variables, el problema se resuelve llamando a la función SR-DP-Solver. Ésta realiza todas las llamadas necesarias a SR-DP para calcular todos los valores necesarios de J, al mismo tiempo que actualiza la información sobre la mejor solución encontrada hasta el momento. Luego ejecuta Build-Path-DP, que arma una solución óptima, a partir de todo lo calculado.

Algoritmo 8

Salida: Una solución óptima de SR para (G, X).

```
1:  SR-DP-Solver()
2:      Sea F una copia de X.
3:      for each e ∈ X do
4:          F ← F − {e}
5:          ℓ1 ← SR-DP(e[1], F, 0)
6:          ℓ2 ← SR-DP(e[2], F, 0)
7:          F ← F ∪ {e}
8:          if mín{ℓ1, ℓ2} < opt_length then
9:              opt_length ← mín{ℓ1, ℓ2}
10:             if ℓ1 ≤ ℓ2 then
11:                 opt_start ← (e, 1)
12:             else
13:                 opt_start ← (e, 2)
14:     return Build-Path-DP()
```

La función SR-DP es simplemente un cómputo top-down de J, basándose en la ecuación del Teorema 4.9. La llamada SR-DP(u, F) calcula el valor $J(u, F)$, y lo almacena en una entrada del diccionario $\texttt{opt}$ para futuras consultas. Por esto, decimos que SR-DP computa el *estado* (u, F). Las líneas 3-4 devuelven la respuesta, en caso que este valor ya haya sido computado previamente. Las líneas 5-7 almacenan y devuelven la respuesta en el caso base. Las líneas 8-18 calculan y almacenan la respuesta en el caso recursivo. Las líneas 15-18 actualizan el diccionario $\texttt{suc}$, en caso de haber encontrado una mejor solución que la conocida, para el estado (u, F). La entrada de $\texttt{suc}$ asociada a (u, F) indica cuál es la arista y el extremo de la misma para los cuales se obtiene el mínimo valor conocido de la ecuación de $J(u, F)$ del Teorema 4.9.

La función Build-Path-DP recorre una secuencia de estados (u, F) que llevan a una solución óptima, a medida que registra el camino recorrido. Para pasar de un estado a otro utiliza el diccionario $\texttt{suc}$, que contiene esa información.

Al igual que en el algoritmo de backtracking, la respuesta que da Build-Path-DP tiene "huecos". Para calcular la complejidad, volveremos a asumir que todas las distancias entre los vértices de $\mathsf{G}[\mathsf{X}]$ se obtienen en $O(1)$.

Teorema 4.10.

1. *El Algoritmo 8 es correcto.*

Algoritmo 9 Algoritmo de programación dinámica para SR.

Entrada: El vértice actual u y el conjunto F de clientes que resta cubrir.

```
1:  SR-DP(u, F)
2:      st ← (u, F)
3:      if opt[st] está definido then
4:          return opt[st]
5:      if |F| = 0 then
6:          opt[st] ← 0
7:          return 0
8:      for each e ∈ F do
9:          F ← F − {e}
10:         ℓ1 ← dist(u, e[1]) + SR-DP(e[1], F)
11:         ℓ2 ← dist(u, e[2]) + SR-DP(e[2], F)
12:         F ← F ∪ {e}
13:         if opt[st] no está definido or mín{ℓ1, ℓ2} < opt[st] then
14:             opt[st] ← mín{ℓ1, ℓ2}
15:             if ℓ1 ≤ ℓ2 then
16:                 suc[st] ← (e, 1)
17:             else
18:                 suc[st] ← (e, 2)
19:     return opt[st]
```

Algoritmo 10 Construcción de una solución óptima.

Salida: Una solución óptima de SR para (G, X).

```
1:  Build-Path-DP()
2:      (e, j) ← opt_start
3:      Sea F una copia de X.
4:      P ← ⟨⟩
5:      for i ← 1, . . . , k do
6:          P ← P ∘ ⟨e[j]⟩
7:          F ← F − {e}
8:          (e, j) ← suc[(e[j], F)]
9:      return P
```

2. *Sea* $Q(n)$ *una cota superior del costo de una consulta en los diccionarios* opt *y* suc*, cuando tienen* n *entradas. Sea* $I(n)$ *una cota superior del costo de inserción, cuando tienen* n *entradas. Supongamos que* $Q(n)$ *e* $I(n)$ *son funciones monótonas no decrecientes. Entonces el costo del Algoritmo 8 es* $O(k^2 2^k(Q(N) + I(N)))$*, con* $N = k2^{k+1}$.

3. *Sea* $S(n)$ *una cota superior de la cantidad de memoria utilizada por* opt *y por* suc*, cuando tienen* n *entradas. Supongamos que* $S(n)$ *es una función monótona no decreciente, y que* $S(n) = \Omega(n)$ *(es decir, ocupan, asintóticamente, al menos una unidad de memoria por cada entrada). Entonces el Algoritmo 8 usa* $O(S(N))$ *memoria adicional a la entrada* (G, X)*, con* $N = k2^{k+1}$.

Demostración.

1. Una llamada SR-DP(u, F) calcula $J(u, F)$, utilizando las ecuaciones del Teorema 4.9. Durante su ejecución, completa el diccionario opt con todos los valores de J que calcula, y el diccionario suc con información de los estados sucesores en una solución óptima.

   La función Build-Path-DP construye una solución óptima de SR, utilizando todo lo calculado por SR-DP y SR-DP-Solver.

   La función SR-DP-Solver ejecuta las llamadas a SR-DP para computar todos los valores de J necesarios, y al mismo tiempo actualiza la información sobre la mejor solución encontrada. Por la Observación 4.2, la longitud mínima encontrada corresponde a la de una solución óptima. Por último, la función devuelve la solución óptima que construye Build-Path-DP.

2. El tiempo de ejecución de SR-DP-Solver es la suma del costo de las llamadas a SR-DP, más el costo de Build-Path-DP. Calculamos cada uno por separado.

   Comenzamos notando que $N = k2^{k+1}$ es la máxima cantidad de entradas que pueden tener los diccionarios opt o suc, puesto que en el peor caso tienen una entrada por cada estado (u, F). Como u siempre es el extremo de alguna arista de X, y F es un subconjunto de X, son $(2k)2^k = k2^{k+1}$ estados posibles. Como las funciones Q e I son monótonas no decrecientes, podemos acotar el costo de cualquier consulta por $O(Q(N))$ y el de cuaquier inserción por $O(I(N))$.

   Para calcular el costo de todas las llamadas a SR-DP, consideramos el árbol de recursión de todas esas llamadas, que tiene como raíz a la ejecución de SR-DP-Solver. Como antes, usamos la palabra *nodo* para referirnos a los vértices del árbol de ejecución. Cada nodo, salvo la raíz, representa la ejecución de una llamada a SR-DP, y tiene un costo asociado. El costo buscado es la suma de los costos asociados a los nodos, salvo la raíz. Los nodos internos, salvo la raíz, representan casos recursivos de SR-DP, mientras que las hojas representan o bien instancias en las que el valor buscado de J ya estaba calculado, o bien casos base.

   La ejecución de un nodo hoja está compuesta por las líneas 3-4 (la respuesta ya estaba calculada) o bien las líneas 5-7 (es un caso base). Luego, el costo de cualquiera de estos nodos es $O(Q(N) + I(N))$, puesto que en las líneas 3-4 se consulta a opt, y en la línea 6 se inserta un valor en opt.

   En la ejecución de un nodo interno se ejecutan las líneas 8-18. Entonces, el costo de cualquiera de ellos es $O(k(Q(N)+I(N)))$, pues el ciclo 8-18 se ejecuta $O(k)$ veces, las

líneas 9-12 son $O(1)$, la línea 13 ejecuta una consulta a opt, la línea 14 una inserción en opt, y las líneas 16 y 18 inserciones en suc.

Luego, si α es la cantidad de nodos internos, y β la cantidad de hojas, la suma de los costos asociados a los nodos, sacando a la raíz, es $(\alpha - 1) \times O(k(Q(N) + I(N))) + \beta \times O(Q(N) + I(N))$. Como cada valor de J se computa a lo sumo una vez, gracias a la memoización de los resultados de esos cómputos, hay $\alpha = O(N)$ nodos internos. Como cada nodo interno tiene dos hijos (pues el caso recursivo realiza dos llamadas recursivas), es $\beta \leq 2\alpha$. Luego, el costo de todas las llamadas a SR-DP es $O(\alpha k(Q(N) + I(N)) + \alpha(Q(N) + I(N))) = O(kN(Q(N) + I(N))) = O(k^2 2^k(Q(N) + I(N)))$.

El costo de Build-Path-DP es $O(kQ(N))$, pues el ciclo 5-8 realiza k iteraciones, y en cada una la operación costosa es la consulta a suc de la línea 8. Luego, el costo de Build-Path-DP es absorbido por el de las llamadas a SR-DP, y en definitiva SR-DP-Solver toma tiempo $O(k^2 2^k(Q(N) + I(N)))$.

3. La variables globales opt_length y opt_start usan espacio $O(1)$. Los diccionarios opt y suc usan espacio $O(S(N))$ entre los dos, pues a lo sumo tienen N entradas y S es monótona no decreciente. El resto de la memoria utilizada corresponde a la memoria local empleada por cada función.

   SR-DP-Solver usa $O(k)$ memoria local, pues realiza una copia de X en la línea 2. Cada instancia de SR-DP usa $O(k)$ memoria local para la variable st con el estado (u, F) a calcular. Hay $O(k)$ instancias de SR-DP en ejecución en un momento dado, pues esa es la profundidad del árbol de recursión de las llamadas a SR-DP. Entonces, SR-DP usa $O(k^2)$ memoria local en todo momento de la ejecución. Finalmente, Build-Path-DP usa $O(k)$ memoria para copiar X en la variable F en la línea 3, y $O(k)$ memoria para construir el camino P.

   Luego, la ejecución consume $O(k)$ memoria local, y $O(S(N) + k)$ memoria en total. Como $k = O(N) = O(S(N))$, la expresión se simplifica a $O(S(N))$.

□

Calculemos los costos para una representación de los diccionarios como árboles binarios de búsqueda balanceados. Un árbol de este tipo, con n nodos, tiene altura $O(\lg n)$. Estos árboles requieren que las claves (u, F) sean comparables, de modo que debemos proveer una función de comparación. Es sencillo implementar una comparación para este tipo de elementos, que tome tiempo $O(k)$. Al lector interesado lo remitimos al código fuente.

Con esta representación, una consulta debe recorrer el árbol, en el peor caso, desde la raíz hasta una hoja, haciendo una comparación en cada nivel. Luego, ponemos $Q(n) = c_1 \cdot \lg n \cdot k$, para cierta constante positiva c_1. Una inserción también debe descender desde la raíz hasta una hoja, realizando una comparación en cada nivel, y luego insertar un nuevo nodo. La creación de un nuevo nodo toma $O(k)$, pues ese es el costo de copiar una clave (u, F) (en el caso de los diccionarios opt y suc, los valores asociados a las claves ocupan $O(1)$ memoria). Luego, tomamos $I(n) = c_2 \cdot \lg n \cdot k$, para cierta constante positiva c_2. Finalmente, como cada nodo del árbol ocupa $O(k)$ memoria, elegimos $S(n) = c_3 \cdot n \cdot k$, para cierta constante positiva c_3. Por lo tanto, para esta estructura de datos, el teorema indica que el algoritmo corre en tiempo $O(k^4 2^k)$, y usa espacio $O(k^2 2^k)$. Observar que con esta representación, el algoritmo de programación dinámica es más rápido que el backtracking,

que corre en $O(k \cdot k!)$. Esto se logra a costa del uso de una cantidad exponencialmente mayor de memoria que el backtracking, el cual sólo usa $O(k)$ espacio.

4.2.3. Implementación de funciones de acotación

Recordemos que el objetivo de las funciones de acotación es, dada una solución factible de SR construída parcialmente, acotar el mínimo costo en el que debemos incurrir para completarla. Una función de acotación de SR, es una función de un vértice u, el vértice en el que estamos parados, y un subconjunto de clientes F, aquellos que resta visitar.

Supongamos que un algoritmo para SR construyó un camino que termina en un vértice u, y le falta cubrir a los clientes F. Supongamos que el camino construído tiene longitud ℓ. Sea `opt_length` la longitud de la mejor solución encontrada hasta el momento. Entonces, dada una función de acotación B, si

$$\ell + B(u, F) \geq \texttt{opt\_length}$$

no nos interesa continuar construyendo este camino, pues sabemos que no mejorará a la mejor solución encontrada.

El Algoritmo 11 realiza este chequeo para una familia de funciones de acotación. Si la respuesta de PRUNE es afirmativa, la construcción del camino debe abortarse, o sea que se *poda* una rama de la ejecución.

Algoritmo 11 Aplicación de las funciones de acotación.

Entrada: El vértice actual u, el conjunto F de clientes que resta cubrir, y la longitud ℓ del camino recorrido hasta u.

Salida: **true** si alguna función de acotación indica que el camino recorrido no podrá mejorar `opt_length`, y **false** en caso contrario.

```
1: PRUNE(u, F, ℓ)
2:     for each función de acotación B do
3:         if ℓ + B(u, F) ≥ opt_length then
4:             return true
5:     return false
```

¿Cómo adaptamos esta verificación a los algoritmos exactos desarrollados? Si bien el Algoritmo 6 no construye un camino, sino una permutación de aristas, mantiene las longitudes ℓ_1 y ℓ_2 que son las longitudes de dos caminos que realizan la longitud mínima de la permutación parcial π, terminando, cada uno, en un extremo distinto del última arista de π. En otras palabras, ℓ_1 y ℓ_2 son las longitudes de dos caminos parciales, que están implícitos en la permutación parcial. A ambos caminos les falta cubrir los clientes del conjunto F. El Algoritmo 12 muestra como incorporar la función PRUNE. Notar que para detener la construcción, el chequeo de PRUNE debe ser afirmativo para los dos caminos parciales. Si uno es afirmativo pero el otro es negativo, debemos continuar extendiendo la permutación π con la esperanza de poder extender el segundo a una solución factible mejor, independientemente de que el primero no tenga ningún futuro. De todos modos, como la mayoría de las funciones de acotación sólo dependen de F y no de u, y dado que ℓ_1 y ℓ_2 son casi iguales (es fácil ver que sólo pueden diferir en 1), es altamente probable que las dos llamadas a PRUNE arrojen el mismo resultado.

Resulta más claro cómo implementar las llamadas a PRUNE en el Algoritmo 9, pues éste construye un único camino a la vez, aunque también en forma implícita. Cada transición

Algoritmo 12 Algoritmo 6 con pruning.

```
1: SR-Backtracking(π, F, ℓ1, ℓ2)
2:     if |F| = 0 and mín{ℓ1, ℓ2} < opt_length then
3:         ...
4:     Sea e la última arista de π.
5:     if Prune(e[1], F, ℓ1) and Prune(e[2], F, ℓ2) then
6:         return
7:     for each f ∈ F do
8:         ...
```

de un estado a otro del algoritmo de programación dinámica representa una elección del próximo vértice en el camino que se construye. La adaptación se presenta en el Algoritmo 13. Notar que se suma un nuevo argumento, cuya finalidad es mantener la longitud del camino parcial construido.

Algoritmo 13 Algoritmo 9 con pruning.

```
1: SR-DP(u, F, ℓ)
2:     ...
3:     if |F| = 0 then
4:         ...
5:     if Prune(u, F, ℓ) then
6:         return ∞
7:     for each e ∈ F do
8:         F ← F − {e}
9:         ℓ1 ← dist(u, e[1]) + SR-DP(e[1], F, ℓ + dist(u, e[1]))
10:        ℓ2 ← dist(u, e[2]) + SR-DP(e[2], F, ℓ + dist(u, e[2]))
11:        F ← F ∪ {e}
12:        ...
13:    ...
14:    return opt[st]
```

4.3. Implementación y resultados experimentales

Implementamos los dos algoritmos exactos, junto con la mayoría de las funciones de acotación que encontramos. Lo hicimos en lenguaje C++ y el código se puede encontrar en https://github.com/guidotag/MSc-Thesis/tree/master/src. Los experimentos se corrieron sobre una CPU Intel® Core™ i5-3470 de 4 cores a 3.20GHz cada uno, 8GB de RAM, en un sistema operativo Ubuntu 14.04 LTS.

Todos los experimentos se realizaron sobre grafos grilla, que es el caso de mayor interés práctico. Restringirnos a esta clase permite implementar eficientemente el cómputo de vertex covers mínimos, que son utilizados por dos de las funciones de acotación.

4.3.1. Funciones de acotación

Las funciones de acotación implementadas son las que se muestran en la Tabla 4.1.

Nombre	Función	Referencia
`clients_count`	$\lfloor \lvert F \rvert / 3 \rfloor - 1$	Teorema 4.6
`dist_x2`	$\max_{\{e,f\} \subseteq F} (\min(\text{dist}(u, e), \text{dist}(u, f)) + \text{dist}(e, f))$	Teorema 4.2
`dist_x3`	$\max_{\{e,f,g\} \subseteq F} \min\{\dots\}$	Teorema 4.3
`mst`	$\text{MST}(K(F), \text{dist})$	Corolario 4.3
`vc`	$\tau(G[F]) - 1$	Teorema 4.1
`vc_mst`	$\tau(G[F]) - 1 + \text{MST}(H, \text{dist}) - (r - 1)$	Corolario 4.4

Tabla 4.1: Funciones de acotación implementadas.

Nuestras implementaciones de las cotas son competitivas, en el sentido de que utilizan algoritmos que son óptimos o están cerca de serlo, a excepción de `vc` y `vc_mst`. Estas dos cotas calculan $\tau(G[X])$ reduciendo el problema a matching máximo sobre grafos bipartitos, que a su vez se resuelve vía el algoritmo de flujo máximo de Edmonds y Karp [4, p. 664], aunque son conocidos otros algoritmos de flujo máximo más rápidos para calcular un flujo máximo como el algoritmo Push-relabel [4, p. 669]. De todos modos, en estos casos un mejor algoritmo no redundará en una mejora sustancial en el tiempo de cómputo, pues podemos suponer que la cantidad $k = |X|$ de clientes es relativamente pequeña, con lo cual $G[X]$ es pequeño. Esta hipótesis se basa en que, como veremos, los algoritmos exactos desarrollados sólo son capaces de resolver instancias con pocos clientes. En la sección que sigue, profundizamos sobre las limitaciones prácticas de nuestros algoritmos exactos.

Dos son las características de las cotas que nos interesa medir: el costo temporal y la *efectividad*. Recordemos que una función de acotación es una cota inferior del costo mínimo para extender una solución parcial, a una factible. Cuanto más grande sea el valor de la misma, más cerca del costo mínimo está, y en ese caso decimos que es más efectiva. Pero la efectividad no es lo único que hace a una función de acotación, pues una tal función puede ser muy efectiva pero muy lenta de computar, haciendo que el tiempo que se gana reduciendo la cantidad de pasos del algoritmo exacto, se pierda computando esta función. Por esta razón es que también nos interesa medir el tiempo de ejecución de estas funciones.

Para realizar la experimentación computacional, consideramos una serie de instancias de SR y sobre cada una calculamos cada función de acotación. Cada instancia representa un momento dado de la ejecución de cualquiera de los algoritmos exactos, en el que hay un subconjunto de clientes que falta cubrir. Específicamente, cada instancia es una grilla de $n \times n$, junto con un subconjunto de clientes F tomado al azar. Para generar F, fijamos un parámetro $0 \leq p \leq 100$, que indica el porcentaje de aristas la grilla que son clientes. Experimentamos considerando $n \in \{5, 10, 20, 30, 40, 50, \dots, 100\}$, y para cada uno de estos valores consideramos $p \in \{10, 20, 30, \dots, 100\}$. Sobre cada instancia, calculamos cada función de acotación. Cuando una función de acotación depende de un vértice u (en nuestro caso, sólo `dist_x2` y `dist_x3`), elegimos a u como el centro de la grilla. No experimentamos con valores de n más grandes, porque, como mostraremos en la Subsección 4.3.2, nuestros algoritmos exactos, incluso cuando son implementados con funciones de acotación, no logran resolver en un tiempo razonable instancias de $k > 25$ clientes.

Esta experimentación tiene dos objetivos. En primer lugar, determinar si hay funciones de acotación que sean más efectivas, en la mayoría de los casos, que otras. Así podremos descartar funciones de acotación que terminan siendo poco útiles, ante la presencia de

otras más efectivas y no mucho más costosas, o inclusive menos costosas. En segundo lugar, buscamos estimar el desempeño que cada función tendrá en cada momento de la ejecución de un algoritmo exacto. La ejecución de una rama de cualquiera de los algoritmos exactos se puede pensar como la variación decreciente del porcentaje de clientes que resta cubrir. Dicho de otro modo, fijado un n, cada valor de p representa una etapa distinta de la ejecución de un algoritmo exacto. Por lo tanto, este análisis nos permite determinar cuáles son las funciones de acotación que se espera que produzcan resultados más efectivos, en cada momento de la ejecución, y así podremos evitar computar todas las funciones, todo el tiempo. Esto es, podemos hacer que PRUNE dependa del porcentaje de clientes restantes.

Hay otras preguntas interesantes que la experimentación responde. A priori, no queda claro cuáles cotas son mejores que otras. Hay cotas, como `clients_count`, que son muy fáciles de computar, y otras como `vc_mst`, que son mucho más sofisticadas, pero no es claro que `vc_mst` sea mejor que `clients_count`. Las únicas cotas para las que conocemos su relación de efectividad, son `vc` y `vc_mst`, y `dist_x2` y `dist_x3`. Por un lado, `vc_mst` siempre es mayor o igual a `vc`, pues sólo difieren en el factor $\mathrm{MST}(H, \mathrm{dist}) - (r-1) \geq 0$ de `vc_mst`. Sin embargo, no queda claro si la diferencia es suficientemente grande como para que el cómputo de $\mathrm{MST}(H, \mathrm{dist}) - (r-1)$ se justifique. Por otro lado, `dist_x3` siempre es mayor o igual a `dist_x2`, pues `dist_x3` es un refinamiento de `dist_x2`. La pregunta que nos hacemos, nuevamente, es si `dist_x3` es suficientemente más efectiva que `dist_x2`, para compensar su mayor tiempo de ejecución.

La Figura 4.5, la Figura 4.6, la Figura 4.7 y la Figura 4.8 presentan los tiempos de ejecución del cálculo de cada cota, para $n = 5, 10, 30$ y 50, respectivamente. Para $n = 5, 10$ y 30, cada figura incluye dos gráficos, uno de los cuales sólo contiene la medición de `dist_x3`. Esta cota sólo fue medida para $n \leq 30$ pues su tiempo de cómputo es excesivamente grande, inclusive para esos valores n, y este costo no se compensa con una buena efectividad. Este tiempo de ejecución se grafica separadamente, porque es varios órdenes de magnitud más grande que los demás. Análogamente, la Figura 4.9, la Figura 4.10, la Figura 4.11 y la la Figura 4.12 presentan los valores de las cotas registrados, también para $n = 5, 10, 30$ y 50. No incluimos los resultados para los demás valores de n, porque las conclusiones no cambian.

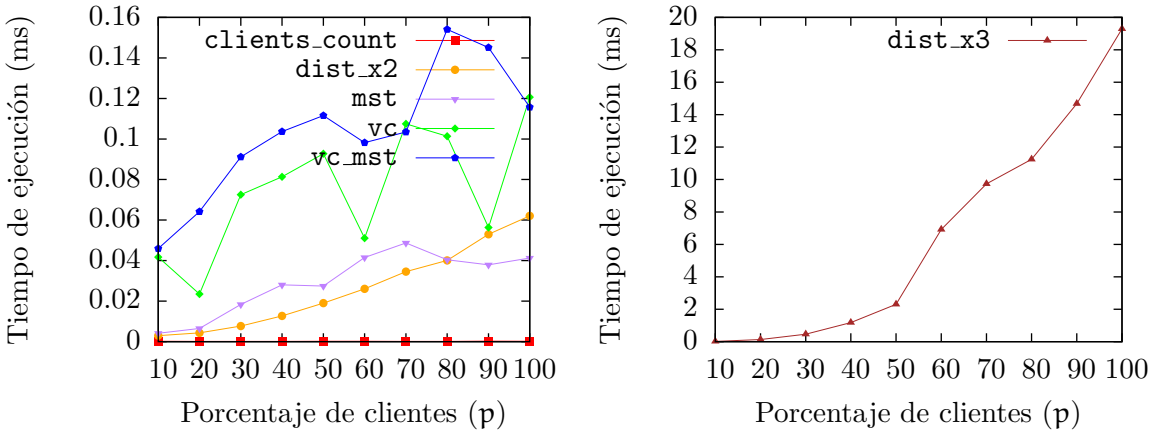


Figura 4.5: Tiempos de ejecución de las cotas para $n = 5$.

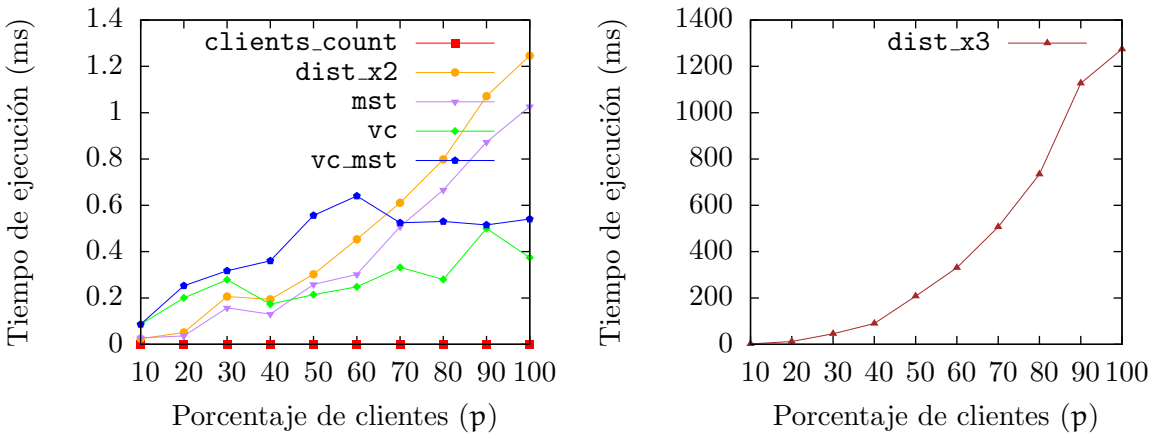


Figura 4.6: Tiempos de ejecución de las cotas para $n = 10$.

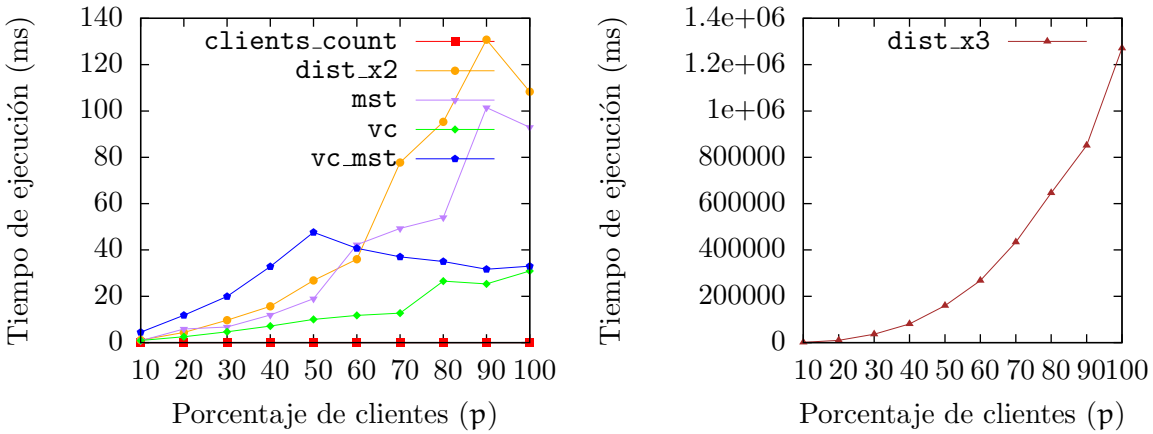


Figura 4.7: Tiempos de ejecución de las cotas para $n = 30$.

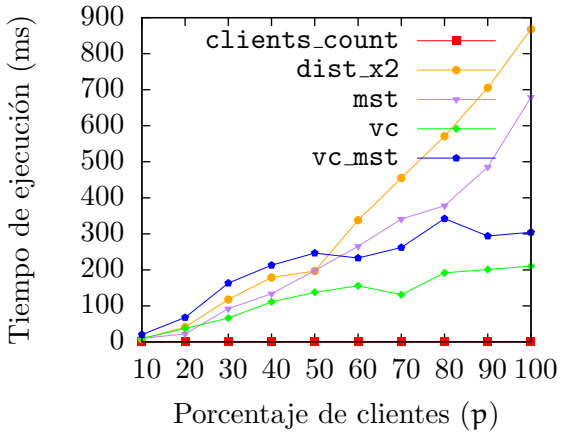


Figura 4.8: Tiempos de ejecución de las cotas para $n = 50$.

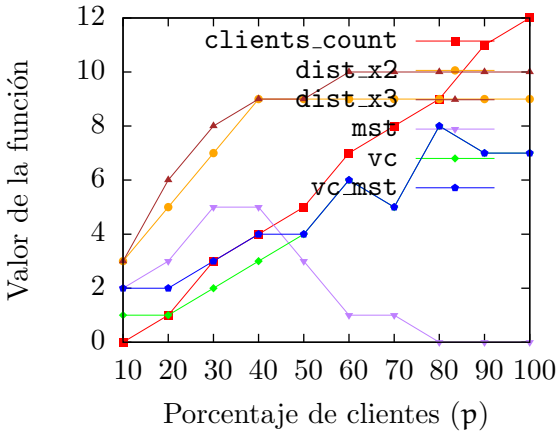


Figura 4.9: Valores de las cotas para $n = 5$.

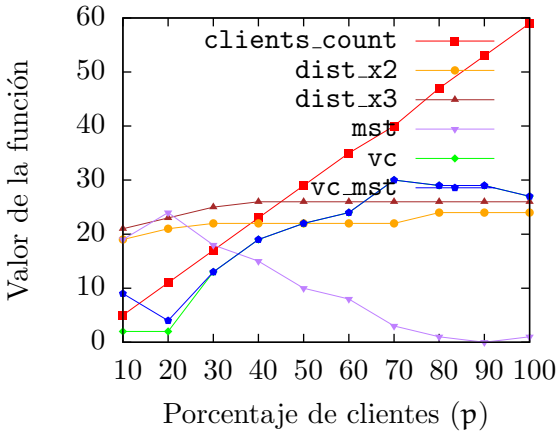


Figura 4.10: Valores de las cotas para $n = 10$.

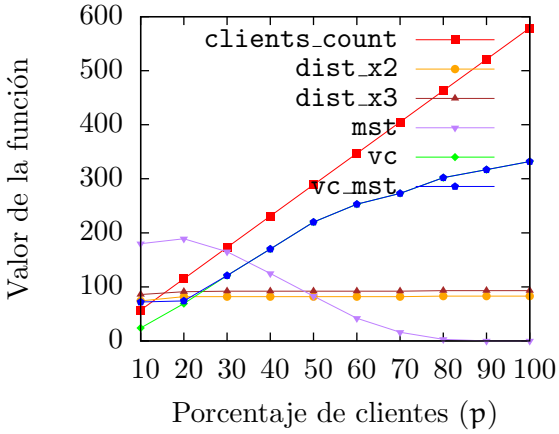


Figura 4.11: Valores de las cotas para $n = 30$.

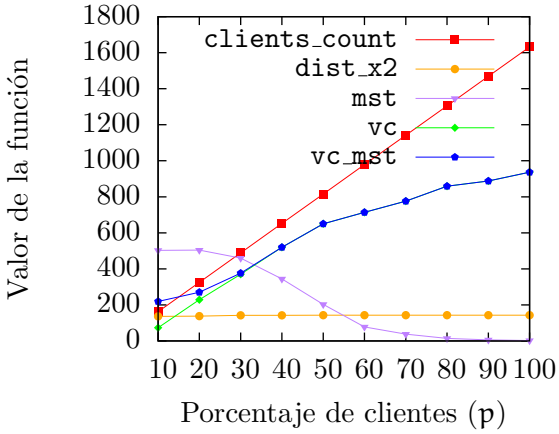


Figura 4.12: Valores de las cotas para $n = 50$.

La primera conclusión que se puede sacar es que las mediciones (tanto de tiempo de ejecución como de valor de las cotas) convergen a medida que n crece, lo cual se observa en la tendencia de los resultados para $n = 10, 30$ y 50. Lo segundo que se observa es que los resultados para $n = 5$ no siguen el patrón general, lo cual es razonable, debido a que las instancias de ese tamaño son demasiado chicas. En lo que sigue, sólo analizamos los resultados para $n = 10, 30$ y 50.

En todos los casos, la cota más costosa es `dist_x3`, que insume tres o más órdenes de magnitud más de tiempo que el resto de las cotas. Dejamos de lado esta cota para analizar y comparar el resto. Se puede ver todas las cotas mantienen, aproximado, el mismo orden relativo a medida que p crece, excepto por `vc_mst`, que comienza siendo la más costosa pero es superada por `dist_x2` y `mst`. La Tabla 4.2 muestra el ranking aproximado de cotas, por tiempos de ejecución, según indica la tendencia general. Para construir este ranking particionamos la escala de porcentajes en tres intervalos, $[10, 30)$, $[30, 50)$ y $[50, 100]$, elegidos en base a las variaciones de costo y efectividad que observamos en los gráficos. En cada intervalo, el ranking de cada cota está dado por una estimación del área debajo de su curva, de la Figura 4.7.

#	$p \in [10, 30)$	$p \in [30, 50)$	$p \in [50, 100]$
1	`dist_x3`	`dist_x3`	`dist_x3`
2	`vc_mst`	`vc_mst`	`dist_x2` (+1)
3	`dist_x2`	`dist_x2`	`mst` (+1)
4	`mst`	`mst`	`vc_mst` (-2)
5	`vc`	`vc`	`vc`
6	`clients_count`	`clients_count`	`clients_count`

Tabla 4.2: Ranking aproximado de cotas, por tiempo de ejecución.

Elaboramos el mismo ranking aproximado para la efectividad de las cotas, que se muestra en la Tabla 4.3. En este caso, confeccionamos el ranking en base a la Figura 4.11. Hay varias observaciones interesantes a hacer sobre la efectividad registrada de las cotas. En primer lugar, es notable que la cota `clients_count`, la más simple de calcular de todas, sea la más efectiva, por una diferencia importante, para $p \in [30, 100]$. En segundo lugar, se puede ver que `vc` y `vc_mst` convergen rápidamente al (aproximadamente) mismo valor, a medida que p crece. Esto se debe, presumiblemente, a que a mayor densidad de clientes, las componentes conexas de $G[X]$ están una más cerca de la otra, en G, lo que se traduce en que $\mathrm{MST}(H, \mathrm{dist})$ tiende a $r - 1$, y por ende el término $\mathrm{MST}(H, \mathrm{dist}) - (r - 1)$ de `vc_mst` tiende a 0. En tercer lugar, se puede ver que la diferencia entre `dist_x2` y `dist_x3` es muy pequeña, con lo cual se puede concluir que, en general, no tiene mucho sentido usar `dist_x3`, que tiene un tiempo de ejecución órdenes de magnitud más grande. Para terminar, hacemos una mención al comportamiento de `mst`, que es la única cota que decrece a medida que p crece. Esto se explica del mismo modo que la convergencia de `vc` y `vc_mst` al mismo valor: a medida que la densidad de clientes crece, la distancia entre los clientes decrece, y por ende $\mathrm{MST}(K(X), \mathrm{dist})$ tiende a 0.

Enumeramos a continuación algunas conclusiones surgidas de la experimentación:

- En general, `dist_x2` es un buen sustituto de `dist_x3`, y `vc` lo es de `vc_mst`.
- Como `clients_count` toma unos pocos milisegundos, siempre conviene ejecutarla,

#	$p \in [10, 30)$	$p \in [30, 50)$	$p \in [50, 100]$
1	`mst`	`clients_count` (+1)	`clients_count`
2	`clients_count`	`vc_mst` (+2)	`vc_mst`
3	`dist_x3`	`vc` (+3)	`vc`
4	`vc_mst`	`mst` (-3)	`dist_x3` (+1)
5	`dist_x2`	`dist_x3` (-2)	`dist_x2` (+1)
6	`vc`	`dist_x2` (-1)	`mst` (-2)

Tabla 4.3: Ranking aproximado de cotas, por valor.

y en primer lugar.

- Para $p \geq 30$, `clients_count` es la cota más efectiva, seguida de `vc`. Como además, estas dos cotas son las que menos tiempo insumen, son indiscutiblemente la mejor opción.
- Para $p < 30$, `mst` es la cota más efectiva, seguida de `clients_count`, y sus costos confirman que, son una excelente elección.
- En grillas muy pequeñas (del orden de 5×5), las conclusiones anteriores no son válidas. En estos casos, la mejor opción pareciera ser `dist_x2`, tanto por su efectividad como por su costo.

En base a estas conclusiones, confeccionamos una función PRUNE, que se presenta en el Algoritmo 14.

Algoritmo 14 Aplicación de las funciones de acotación, sobre grillas.

Entrada: El vértice actual u, el conjunto F de clientes que resta cubrir, y la longitud ℓ del camino recorrido hasta u.

Salida: **true** si alguna función de acotación indica que el camino recorrido no podrá mejorar `opt_length`, y **false** en caso contrario.

```
1:  PRUNE(u, F, ℓ)
2:      Sean n y m las dimensiones de la grilla.
3:      M ← n(m − 1) + m(n − 1)                    ▷ la cantidad de aristas de la grilla
4:      p ← |F|/M
5:      if M < 80 then                  ▷ la grilla es, aproximadamente, de 7 × 7 o más chica
6:          if ℓ + dist_x2(u, F) ≥ opt_length then
7:              return true
8:      else
9:          if 0,3 ≤ p and (ℓ+clients_count(F) ≥ opt_length or ℓ+vc(F) ≥ opt_length)
    then
10:             return true
11:         else if p < 0,3 and (ℓ + clients_count(F) ≥ opt_length or ℓ + mst(F) ≥
    opt_length) then
12:             return true
13:     return false
```

4.3.2. Algoritmos exactos

En lo que sigue, llamamos *algoritmo BT* al algoritmo de backtracking y *algoritmo PD* al algoritmo de programación dinámica. Las implementaciones de los estos algoritmos no ofrecen mayores dificultades técnicas. Debemos mencionar que en la del algoritmo PD utilizamos `std::map` para implementar los diccionarios. Esta clase de la biblioteca estándar de C++ es la representación de un diccionario como un árbol binario de búsqueda balanceado [21]. En ambos algoritmos, implementamos el subconjunto de clientes F con un `std::set`, que usa la misma estructura interna que `std::map`. Si bien en el análisis previo asumimos que F realiza inserciones, borrado y consulta en $O(1)$, una representación de `std::map` respeta, a nivel práctico, estos costos temporales, pues la cantidad de clientes que manejamos es pequeña.

Como los dos algoritmos exactos corren en tiempo supra-polinomial en la cantidad k de clientes, es esperable que sólo sean capaces de resolver instancias de pocos clientes. Para tener una idea de cuál es la eficiencia esperable, en tiempo y espacio, de nuestros algoritmos, calculamos algunos valores puntuales de las estimaciones asintóticas de las complejidades temporal y espacial, y los presentamos en la Tabla 4.4. Lógicamente, estos números no deben interpretarse en forma absoluta, pues los cálculos asintóticos esconden constantes y términos que aportan al costo total. Para analizar la tabla, tengamos en mente que un procesador *commodity* moderno ejecuta alrededor de 10^9 (mil millones) de instrucciones por segundo, y que los experimentos se ejecutaron sobre una computadora con alrededor de 8GB de RAM. En función de estos datos, podemos plantear algunas tesis previas a la experimentación:

- El algoritmo BT no podrá ejecutar instancias de más de 15 clientes, en una cantidad de tiempo razonable (del orden de unas pocas horas), a menos que las funciones de acotación reduzcan el cómputo necesario.

- El algoritmo PD no podrá ejecutar instancias de más de 20 clientes, en una computadora con una cantidad de memoria razonable (del orden de unos pocos GB), a menos que las funciones de acotación reduzcan el cómputo necesario.

k	Tiempo BT $k \cdot k!$	Tiempo PD $k^4 2^k$	Espacio PD $k^2 2^k$	Instancias (n, p)
5	$0{,}6 \cdot 10^3$	$20 \cdot 10^3$	0,8k	$(5, 10)$
10	$\sim 36 \cdot 10^6$	$\sim 10 \cdot 10^6$	$\sim$ 100k	$(5, 20)$, $(5, 30)$
15	$\sim 19 \cdot 10^{12}$	$\sim 1{,}6 \cdot 10^9$	$\sim$ 7M	$(5, 40)$
20	$\sim 48 \cdot 10^{18}$	$\sim 167 \cdot 10^9$	$\sim$ 419M	$(5, 50)$, $(10, 10)$
25	$\sim 387 \cdot 10^{24}$	$\sim 13 \cdot 10^{12}$	$\sim$ 20G	$(5, 60)$
30	$\sim 8 \cdot 10^{33}$	$\sim 869 \cdot 10^{12}$	$\sim$ 966G	$(5, 70)$
35	-	-	-	$(10, 20)$
40	-	-	-	$(15, 10)$

Tabla 4.4: Estimaciones de requerimientos de tiempo y espacio de los algoritmos de backtracking (BT) y de programación dinámica (PD). La columna de instancias enumera los parámetros de instancias con aproximadamente la cantidad de clientes de cada línea.

Corrimos las implementaciones de los dos algoritmos exactos con las funciones de acotación, y medimos el tiempo de ejecución en cada caso. Cada instancia se ejecutó durante 1 hora, y cumplido ese plazo se detuvo la ejecución. Se comprobó que aún utilizando funciones de acotación para acelerar el cómputo, los algoritmos exactos sólo pueden resolver instancias de 25 o menos clientes en la ventana de tiempo provista. Presentamos los resultados en la Tabla 4.5 y Tabla 4.6. Estas tablas presentan los tiempos de ejecución de los dos algoritmos exactos, con y sin funciones de acotación, para las instancias que completaron su ejecución en menos de 1 hora. A las ejecuciones que sufrieron detenciones forzosas por excederse del límite de 1 hora, las simbolizamos *TLE* (por sus siglas en inglés, *time-limit exceeded*).

n	p	Tiempo BT sin cotas (seg)	Tiempo BT con cotas (seg)
5	10	$\sim 10^{-5}$	$\sim 10^{-5}$
5	20	$\sim 0{,}01$	$\sim 10^{-4}$
5	30	~ 200	$\sim 0{,}01$
5	40	TLE	$\sim 0{,}1$
5	50	TLE	~ 9
5	60	TLE	~ 2700
10	10	TLE	$\sim 3{,}5$

Tabla 4.5: Tiempos de ejecución para el algoritmo BT.

n	p	Tiempo PD sin cotas (seg)	Tiempo PD con cotas (seg)
5	10	$\sim 10^{-4}$	$\sim 10^{-4}$
5	20	$\sim 0{,}01$	$\sim 0{,}01$
5	30	~ 1	$\sim 0{,}1$
5	40	~ 55	~ 20
10	10	TLE	~ 175

Tabla 4.6: Tiempos de ejecución para el algoritmo PD.

Se puede ver que las cotas hacen un buen trabajo disminuyendo el tiempo de ejecución. Por ejemplo, la instancia $(n, p) = (5, 30)$ toma aproximadamente 200 segundos para ser ejecutada por el algoritmo BT sin cotas, y apenas 0,01 segundo para ese mismo algoritmo pero con cotas. Se pueden observar análogas, aunque menores, reducciones en los tiempos del algoritmo PD. En ambos casos, se puede observar que al utilizar las cotas, los algoritmos logran ejecutar, en el tiempo provisto, instancias que no terminan de correr de otro modo.

Lamentablemente, el buen trabajo de las cotas no es suficiente para que estos algoritmos logren ejecutar, en un tiempo razonable, instancias de tamaño real, del orden de las decenas y cientos de clientes.

Como PRUNE utiliza únicamente la cota `dist_x2` para instancias de $n = 5$, es necesaria una experimentación con una ventana de tiempo más grande, de varias horas, para medir con precisión el trabajo de las demás cotas, cuyos buenos resultados sólo pudimos apreciar en las dos instancias de $n = 10$ que lograron terminar su ejecución.

Capítulo 5

Algoritmos aproximados para STAR ROUTING

Recordemos que un algoritmo aproximado es un algoritmo que produce soluciones factibles para un problema, pero que no necesariamente son óptimas. En este capítulo presentamos una variedad de algoritmos aproximados para SR sobre distintas clases de grafos: general, sobre grillas y sobre grafos completos. El estudio de la existencia de algoritmos aproximados para un problema es interesante tanto desde el punto de vista teórico como práctico. Por un lado, es interesante desde lo teórico, porque la mera existencia o no de algoritmos aproximados para un problema habla sobre su grado de dificultad. Por ejemplo, es bien sabido que TSP no admite algoritmos aproximados con factor de aproximación constante [8, p. 147], mientras que para TSP métrico hay un algoritmo $(3/2)$-aproximado [8, p. 132], por lo que podemos decir que la versión métrica es "más fácil", aún cuando no se conocen algoritmos eficientes para ninguno de los dos problemas. Desde el punto de vista práctico, un algoritmo aproximado es un buen método para generar una solución factible inicial en el contexto de un algoritmo exacto. Si el factor de aproximación es pequeño, tenemos la garantía de que dicha solución factible no está demasiado lejos de la solución óptima. Combinando esto con buenas funciones de acotación podemos acelerar radicalmente la exploración del espacio de soluciones.

Durante este capítulo, (G, X) es una instancia de SR y escribimos $Z = SR^*(G, X)$ y $k = |X|$.

5.1. Relación matemática entre SR y PTSP

Empezamos explorando ciertas relaciones entre SR y PTSP. Este análisis será útil posteriormente, pues construiremos aproximaciones para SR general y sobre grillas, utilizando algoritmos aproximados para PTSP métrico y rectilíneo.

Definición 5.1 (Distancia máxima)**.** Dadas dos aristas e y f de un grafo, definimos $\overline{\text{dist}}(e, f)$ como la distancia más grande entre un extremo de e y uno de f. Esto es,

$$\overline{\text{dist}}(e, f) = \max_{\substack{u \text{ extremo de } e \\ v \text{ extremo de } f}} \text{dist}(u, v)$$

Notar que $\overline{\text{dist}}$ no es una distancia en el sentido matemático, porque $\overline{\text{dist}}(e, e) = 1 \neq 0$. De todos modos, la denominamos de ese modo, por comodidad. Durante este capítulo

haremos referencia a una variedad de resultados acerca de dist sobre aristas y $\overline{\text{dist}}$, todos presentados en el Apéndice A.

Lema 5.1.

$$\text{PTSP}^*(K(X), \overline{\text{dist}}) \leq \text{PTSP}^*(K(X), \text{dist}) + 2(k-1)$$

Demostración. Sea $P = \langle e_1, \ldots, e_k \rangle$ una solución óptima de PTSP para $(K(X), \text{dist})$. Este camino es una solución factible de PTSP para la instancia $(K(X), \overline{\text{dist}})$, con lo cual $\text{PTSP}^*(K(X), \overline{\text{dist}}) \leq \text{length}_{\overline{\text{dist}}}(P)$. Luego, basta ver que $\text{length}_{\overline{\text{dist}}}(P) \leq \text{PTSP}^*(K(X), \text{dist}) + 2(k-1)$. Se tiene

$$\begin{aligned}
\text{length}_{\overline{\text{dist}}}(P) &= \sum_{i=1}^{k-1} \overline{\text{dist}}(e_i, e_{i+1}) \\
&\leq \sum_{i=1}^{k-1} (\text{dist}(e_i, e_{i+1}) + 2) && \text{(Lema A.1)} \\
&= \sum_{i=1}^{k-1} \text{dist}(e_i, e_{i+1}) + 2(k-1) \\
&= \text{length}_{\text{dist}}(P) + 2(k-1) \\
&= \text{PTSP}^*(K(X), \text{dist}) + 2(k-1)
\end{aligned}$$

□

Lema 5.2. *A partir de una solución factible* P *de* PTSP *para* $(K(X), \text{dist})$*, podemos construir, en tiempo polinomial en el tamaño de* (G, X)*, una solución factible* Q *de* SR *para* (G, X)*, tal que* $\text{length}(Q) \leq \text{length}_{\overline{\text{dist}}}(P)$.

Demostración. Escribamos $P = \langle e_1, \ldots, e_k \rangle$. Para cada $1 \leq i \leq k$, sea u_i cualquiera de los extremos de e_i. Sea Q_i un camino mínimo entre u_i y u_{i+1} en G. Definimos $Q = Q_1 \circ \cdots \circ Q_{k-1}$. Se puede ver que esta es una solución factible de SR para (G, X). Además

$$\begin{aligned}
\text{length}(Q) &= \sum_{i=1}^{k-1} \text{length}(Q_i) \\
&= \sum_{i=1}^{k-1} \text{dist}(u_i, u_{i+1}) \\
&\leq \sum_{i=1}^{k-1} \overline{\text{dist}}(e_i, e_{i+1}) && (u_i \text{ es extremo de } e_i \\
& && \quad \text{y } u_{i+1} \text{ de } e_{i+1}) \\
&= \text{length}_{\overline{\text{dist}}}(P)
\end{aligned}$$

□

El siguiente teorema es el primer paso importante, en la comprensión de la relación entre SR y PTSP.

Teorema 5.1. *Las siguientes afirmaciones son verdaderas:*

1. $\mathsf{PTSP}^*(\mathsf{K}(\mathsf{X}), \mathrm{dist}) \leq \mathsf{Z}$
2. $\mathsf{Z} \leq \mathsf{PTSP}^*(\mathsf{K}(\mathsf{X}), \overline{\mathrm{dist}})$
3. $\mathsf{Z} - 2(\mathsf{k}-1) \leq \mathsf{PTSP}^*(\mathsf{K}(\mathsf{X}), \mathrm{dist})$
4. $\mathsf{PTSP}^*(\mathsf{K}(\mathsf{X}), \overline{\mathrm{dist}}) \leq \mathsf{Z} + 2(\mathsf{k}-1)$

Demostración.

1. Sea $\mathsf{P} = \langle u_1, \ldots, u_r \rangle$ una solución óptima de SR para (G, X). A partir de P, definimos una solución factible de PTSP para $(\mathsf{K}(\mathsf{X}), \mathrm{dist})$ del siguiente modo. Consideremos un elemento cualquiera de $\Pi(\mathsf{X}, \mathsf{P}) \neq \emptyset$ (ver la Definición 4.5), digamos $\mathsf{Q} = \langle e_1, \ldots, e_k \rangle$. Como Q es una permutación de X, es una solución factible de PTSP para $(\mathsf{K}(\mathsf{X}), \mathrm{dist})$. Debemos ver que $\mathrm{length}_{\mathrm{dist}}(\mathsf{Q}) \leq \mathsf{Z} = \mathrm{length}(\mathsf{P}) = r - 1$.

   Como $\mathsf{Q} \in \Pi(\mathsf{X}, \mathsf{P})$, existe una función monótona no decreciente $f : \{1, \ldots, k\} \to \{1, \ldots, r\}$, tal que $u_{f(i)}$ cubre a e_i. Veamos que $\mathrm{length}_{\mathrm{dist}}(\langle e_1, \ldots, e_i \rangle) \leq \mathrm{length}(\langle u_1, \ldots, u_{f(i)} \rangle)$, en forma inductiva. El caso base es obvio, pues $\mathrm{length}_{\mathrm{dist}}(\langle e_1 \rangle) = 0$. Sea $1 \leq i < k$, asumamos que la afirmación vale para i, y veamos que se mantiene para $i+1$.

$$
\begin{aligned}
\mathrm{length}_{\mathrm{dist}}(\langle e_1, \ldots, e_{i+1} \rangle) &= \mathrm{length}_{\mathrm{dist}}(\langle e_1, \ldots, e_i \rangle) + \mathrm{dist}(e_i, e_{i+1}) \\
&\leq \mathrm{length}(\langle u_1, \ldots, u_{f(i)} \rangle) + \mathrm{dist}(e_i, e_{i+1}) && \text{(hipótesis inductiva)} \\
&\leq (f(i) - 1) + \mathrm{dist}(e_i, e_{i+1}) \\
&\leq (f(i) - 1) + \mathrm{dist}(u_{f(i)}, u_{f(i+1)}) && (u_{f(i)} \text{ es extremo de } e_i \\
& && \text{y } u_{f(i+1)} \text{ de } e_{i+1}) \\
&\leq (f(i) - 1) + \mathrm{length}(\langle u_{f(i)}, u_{f(i)+1}, \ldots, u_{f(i+1)} \rangle) && (f(i) \leq f(i+1)) \\
&= (f(i) - 1) + (f(i+1) - f(i)) \\
&= f(i+1) - 1
\end{aligned}
$$

   Esto completa la inducción. Poniendo $i = k$, y dado que $f(k) \leq r$ (más aún, es $f(k) = r$ por la optimalidad de P), concluimos que $\mathrm{length}_{\mathrm{dist}}(\mathsf{Q}) \leq \mathrm{length}(\langle u_1, \ldots, u_{f(k)} \rangle) = f(k) - 1 \leq r - 1$, como queríamos probar.

2. Sea $\mathsf{P} = \langle e_1, \ldots, e_k \rangle$ una solución óptima de PTSP para $(\mathsf{K}(\mathsf{X}), \overline{\mathrm{dist}})$. Sea Q una solución factible de SR para (G, X), como indica el Lema 5.2. Como Q es solución factible, vale $\mathsf{Z} \leq \mathrm{length}(\mathsf{Q})$. Además, $\mathrm{length}(\mathsf{Q}) \leq \mathrm{length}_{\overline{\mathrm{dist}}}(\mathsf{P}) = \mathsf{PTSP}^*(\mathsf{K}(\mathsf{X}), \overline{\mathrm{dist}})$.

3. Es consecuencia del Lema 5.1 y el ítem 2 de este teorema.

4. Es consecuencia del Lema 5.1 y el ítem 2 de este teorema.

□

Corolario 5.1.

$$\mathsf{Z} - 2(\mathsf{k}-1) \leq \mathsf{PTSP}^*(\mathsf{K}(\mathsf{X}), \mathrm{dist}) \leq \mathsf{Z} \leq \mathsf{PTSP}^*(\mathsf{K}(\mathsf{X}), \overline{\mathrm{dist}}) \leq \mathsf{Z} + 2(\mathsf{k}-1)$$

Nos preguntamos qué sucede si en lugar de considerar PTSP sobre $K(X)$, lo hacemos sobre grafos completos definidos a partir de otros elementos de G. Estos elementos deben ser tales que al recorrerlos, el conjunto de clientes X sea cubierto por el camino. Una alternativa natural, es considerar un conjunto de vértices que cubra X, esto es, un vertex cover de $G[X]$.

Lema 5.3. *Sea S un vertex cover de $G[X]$. A partir de una solución factible P de PTSP para $(K(S), \mathrm{dist})$, podemos construir, en tiempo polinomial en el tamaño de (G, X), una solución factible Q de SR para (G, X), tal que $\mathrm{length}(Q) = \mathrm{length}_{\mathrm{dist}}(P)$.*

Demostración. La demostración es similar a la del Lema 5.2. Escribamos $P = \langle u_1, \ldots, u_t \rangle$. Sea Q_i un camino mínimo entre u_i y u_{i+1} en G. Definimos $Q = Q_1 \circ \cdots \circ Q_{t-1}$. Este camino es una solución factible de SR para (G, X), pues pasa por todos los vértices de S, y por lo tanto cubre X. Además

$$\begin{aligned} \mathrm{length}(Q) &= \sum_{i=1}^{t-1} \mathrm{length}(Q_i) \\ &= \sum_{i=1}^{t-1} \mathrm{dist}(u_i, u_{i+1}) \\ &= \mathrm{length}_{\mathrm{dist}}(P) \end{aligned}$$

□

Teorema 5.2. *Sea S un vertex cover de $G[X]$, y escribamos $t = |S|$.*

1. $Z \leq \mathrm{PTSP}^*(K(S), \mathrm{dist})$
2. $\mathrm{PTSP}^*(K(S), \mathrm{dist}) \leq Z + 2(t-1)$

Demostración.

1. Sea $P = \langle u_1, \ldots, u_t \rangle$ una solución óptima de PTSP para $(K(S), \mathrm{dist})$. Sea Q una solución factible de SR para (G, X), como indica el Lema 5.3. Como Q es solución factible, vale $Z \leq \mathrm{length}(Q)$. Además, $\mathrm{length}(Q) = \mathrm{length}_{\mathrm{dist}}(P) = \mathrm{PTSP}^*(K(S), \mathrm{dist})$.

2. Sea $P = \langle u_1, \ldots, u_r \rangle$ una solución óptima de SR para (G, X). La observación clave es que para cada $s \in S$, existe un vértice u_i de P tal que $\mathrm{dist}(s, u_i) \leq 1$, porque s es extremo de al menos una arista e de X, y como el camino P cubre e, debe pasar por alguno de los extremos de e.

   Sea $s_1, \ldots, s_t$ una enumeración de los vértices de S, y sean $1 \leq j_1, \ldots, j_t \leq r$ índices tales que $\mathrm{dist}(s_i, u_{j_i}) \leq 1$ para cada i. Sin pérdida de generalidad, supongamos que $j_1 \leq \cdots \leq j_t$. Si no estuvieran ordenados crecientemente, los ordenamos, modificando acordemente la enumeración de S considerada. Consideramos $Q = \langle s_1, \ldots, s_t \rangle$. Este camino es una solución factible de PTSP para $(K(S), \mathrm{dist})$, por lo que $\mathrm{PTSP}^*(K(S), \mathrm{dist}) \leq \mathrm{length}_{\mathrm{dist}}(Q)$. Además

$$\begin{aligned}
\text{length}_{\text{dist}}(Q) &= \sum_{i=1}^{t-1} \text{dist}(s_i, s_{i+1}) \\
&\leq \sum_{i=1}^{t-1} (\text{dist}(s_i, u_{j_i}) + \text{dist}(u_{j_i}, u_{j_{i+1}}) + \text{dist}(u_{j_{i+1}}, s_{i+1})) \\
&\leq \sum_{i=1}^{t-1} (1 + \text{dist}(u_{j_i}, u_{j_{i+1}}) + 1) \\
&= \sum_{i=1}^{t-1} \text{dist}(u_{j_i}, u_{j_{i+1}}) + 2(t-1) \\
&\leq \text{length}(P) + 2(t-1) \\
&= \text{SR}^*(G, X) + 2(t-1)
\end{aligned}$$

□

Corolario 5.2. *Si S es un vertex cover mínimo de* G[X]*, entonces*

$$Z \leq \text{PTSP}^*(K(S), \text{dist}) \leq Z + 2(\tau(G[X]) - 1)$$

5.2. Algoritmo aproximado para SR general

En esta sección presentamos un primer algoritmo aproximado, para SR general. Este algoritmo utiliza un algoritmo aproximado para PTSP métrico y otro para VC, como cajas negras. Para cada par de algoritmos aproximados elegidos, obtendremos un algoritmo distinto para SR. Por esta razón, la construcción que hacemos describe, en realidad, una familia de algoritmos aproximados para SR.

El funcionamiento de este algoritmo aproximado para SR se explica gráficamente a través del diagrama de la Figura 5.1. La idea es transformar una instancia de SR en una instancia de PTSP métrico, pasando por VC. Luego resolvemos la instancia de PTSP métrico, y finalmente volvemos hacia atrás, transformando esta solución factible de PTSP en una solución factible de SR.

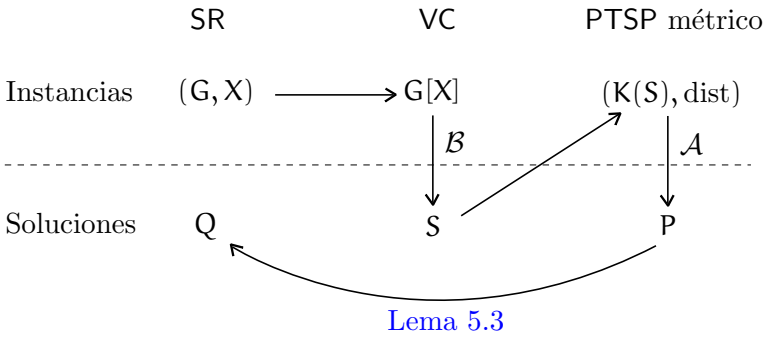


Figura 5.1: Diagrama del algoritmo aproximado para SR; $\mathcal{A}$ es un algoritmo aproximado para PTSP métrico, y $\mathcal{B}$ es un algoritmo aproximado para VC.

Teorema 5.3. *Sea $\mathcal{A}$ un algoritmo α-aproximado para* PTSP *métrico. Sea $\mathcal{B}$ un algoritmo β-aproximado para* VC*. Existe un algoritmo aproximado $\mathcal{C}$ para* SR*, que usa a $\mathcal{A}$ y $\mathcal{B}$ como cajas negras, tal que*

$$\mathcal{C}(G, X) \leq \alpha(1 + 2\beta)Z + 2\alpha(\beta - 1)$$

Demostración. Sea $\mathcal{C}$ el Algoritmo 15. El algoritmo $\mathcal{C}$ es polinomial en el tamaño de la entrada, porque cada paso es polinomial. Una observación importante es que el paso 2 del algoritmo está bien definido, en el sentido de que la entrada $(K(S), \text{dist})$ de $\mathcal{A}$, es una instancia de PTSP métrico, tal como requiere $\mathcal{A}$. Esto es porque dist es una métrica, de modo que satisface la desigualdad triangular. Por otro lado, notar que el resultado Q del algoritmo es una solución factible de SR para (G, X), como indica el Lema 5.3.

Algoritmo 15 Algoritmo aproximado del Teorema 5.3.

Entrada: Una instancia (G, X) de SR.
Salida: Una solución factible de SR para (G, X).
1: Calcular $S = \mathcal{B}(G[X])$.
2: Calcular $P = \mathcal{A}(K(S), \text{dist})$.
3: A partir de P, construir Q como indica el Lema 5.3.
4: **return** Q

Veamos que $Q = \mathcal{C}(G, X)$ satisface la garantía de aproximación buscada. Tenemos que

$$\begin{aligned}
\text{length}(Q) &= \text{length}_{\text{dist}}(P) && \text{(Lema 5.3)} \\
&\leq \alpha\ \text{PTSP}^*(K(S), \text{dist}) && (\mathcal{A} \text{ es } \alpha\text{-aproximado}) \\
&\leq \alpha(Z + 2(|S| - 1)) && \text{(Teorema 5.2)} \\
&\leq \alpha(Z + 2(\beta\tau(G[X]) - 1)) && (\mathcal{B} \text{ es } \beta\text{-aproximado}) \\
&= \alpha(Z + 2\beta(\tau(G[X]) - 1) + 2\beta - 2) \\
&\leq \alpha(Z + 2\beta Z + 2\beta - 2) && \text{(Corolario 4.1)} \\
&= \alpha(1 + 2\beta)Z + 2\alpha(\beta - 1)
\end{aligned}$$

como queríamos.

□

5.3. Algoritmos aproximados para SR sobre grillas

El algoritmo aproximado que proponemos es similar al Algoritmo 15, excepto que ahora podemos calcular un vertex cover mínimo en tiempo polinomial, porque el grafo es bipartito, por vías del Teorema 1.4. Además, aprovechamos que la instancia $(K(S), \text{dist})$ de PTSP no sólo es métrica, sino, más aún, rectilínea.

Teorema 5.4. *Sea $\mathcal{A}$ un algoritmo α-aproximado para* PTSP *rectilíneo. Existe un algoritmo 3α-aproximado $\mathcal{B}$ para* SR *sobre grillas, que usa a $\mathcal{A}$ como caja negra.*

Demostración. Sea $\mathcal{B}$ el Algoritmo 16. El algoritmo $\mathcal{B}$ es polinomial en el tamaño de la entrada, porque cada paso es polinomial. En particular, el paso 1 es polinomial, porque

al ser G bipartito, $G[X]$ también lo es, y le podemos computar un vertex cover mínimo en tiempo polinomial. Observar que el paso 2 del algoritmo está bien definido, porque la entrada $(K(S), \text{dist})$ de $\mathcal{A}$, es una instancia de PTSP rectilíneo, tal como requiere $\mathcal{A}$. Esto es porque el subconjunto de vértices S proviene de la grilla G.

Algoritmo 16 Algoritmo aproximado del Teorema 5.4.

Entrada: Una instancia (G, X) de SR sobre grillas.
Salida: Una solución factible de SR para (G, X).
1: Calcular un vertex cover mínimo S de $G[X]$.
2: Calcular $P = \mathcal{A}(K(S), \text{dist})$.
3: A partir de P, construir Q como indica el Lema 5.3.
4: **return** Q

El resultado $Q = \mathcal{B}(G, X)$ del algoritmo satisface

$$\begin{aligned}
\text{length}(Q) &= \text{length}_{\text{dist}}(P) && \text{(Lema 5.3)}\\
&\leq \alpha\ \text{PTSP}^*(K(S), \text{dist}) && (\mathcal{A} \text{ es } \alpha\text{-aproximado})\\
&\leq \alpha\ (Z + 2(|S| - 1)) && \text{(Teorema 5.2)}\\
&= \alpha\ (Z + 2(\tau(G[X]) - 1)) && (S \text{ es vertex cover mínimo de } G[X])\\
&\leq \alpha\ (Z + 2Z) && \text{(Corolario 4.1)}\\
&= 3\alpha Z
\end{aligned}$$

□

5.3.1. Algoritmos para el caso de alta densidad de clientes

Un escenario que se puede presentar en el mundo real, es el de una ciudad que tiene barrios específicos con una alta densidad de entregas. Si nos concentramos en el barrio, olvidándonos del resto de la ciudad, podemos modelar esta situación como una instancia (G, X) de SR, con $G = (V, E)$ una grilla, en la que casi toda arista de G está en X. En este caso, es de esperarse que una solución óptima de la instancia (G, E), que es una solución factible para (G, X), sea una buena aproximación de $\text{SR}^*(G, X)$. En esta sección probamos que esto es efectivamente así, y damos aproximaciones para la instancia (G, E), que derivan en algoritmos aproximados para (G, X). Los valores $\text{SR}^*(G, E)$ y $\text{SR}^*(G, X)$ se relacionan a través de $\tau(G)$, de modo que estudiaremos la relación entre $\text{SR}^*(G, X)$ y $\tau(G)$, y luego la relación entre $\tau(G)$ y las aproximaciones de SR para (G, E).

Lema 5.4. *Sea e una arista de un grafo G. Entonces $\tau(G - e) \geq \tau(G) - 1$.*

Demostración. Consideremos un vertex cover mínimo S de $G - e$. Este conjunto cubre todas las aristas de G, excepto quizás e, con lo cual agregándole cualquiera de los dos extremos de e obtenemos un vertex cover de G. Luego, G tiene un vertex cover de cardinal $|S|+1$, con lo cual $\tau(G) \leq |S|+1 = \tau(G-e)+1$, o equivalentemente $\tau(G-e) \geq \tau(G)-1$. □

Lema 5.5. *Supongamos que (G, X) es una instancia de SR general. Sea $\overline{k} = |E| - k$ la cantidad de aristas de G que no son clientes. Entonces $\tau(G) \leq Z + \overline{k} + 1$.*

Demostración. Llamemos $\overline{X} = E - X$ al conjunto de aristas que no son clientes. Como ya sabemos, por el Corolario 4.1 se tiene $Z \geq \tau(G[X]) - 1$. El resto de la demostración consiste en ver que $\tau(G[X]) \geq \tau(G) - \overline{k}$.

Aplicando sucesivas veces el Lema 5.4, se deduce que $\tau(G - \overline{X}) \geq \tau(G) - \overline{k}$. Además, es $\tau(G[X]) = \tau(G - \overline{X})$, porque $G[X]$ y $G - \overline{X}$ sólo difieren en un conjunto de vértices aislados, que no forman parte de ningún vertex cover mínimo. □

Pasamos a estudiar la relación entre $\tau(G)$ y $SR^*(G, E)$, en el caso en que G es un grafo grilla. Comenzamos calculando $\tau(G)$.

Teorema 5.5. *Sea G un grafo grilla de n filas y m columnas. Entonces $\tau(G) = \lfloor nm/2 \rfloor$.*

Demostración. ($\leq$) Buscaremos un vertex cover de G, de cardinal menor o igual a $\lfloor nm/2 \rfloor$. Como G es bipartito, podemos considerar una bipartición $\{V_1, V_2\}$ de los vértices. Notar que tanto V_1 como V_2 son vertex cover de G. De estos dos conjuntos, tomemos el más pequeño. Supongamos, sin pérdida de generalidad, que es V_1. Entonces $|V_1| \leq \lfloor |V(G)|/2 \rfloor = \lfloor nm/2 \rfloor$. Luego, V_1 es un vertex cover como el que queríamos.

($\geq$) Por el Teorema 1.1, basta encontrar un matching de G de tamaño $\lfloor nm/2 \rfloor$. Separamos en casos, según la paridad de n y m.

Si m es par, consideramos el matching M de la Figura 5.2-(a). Este matching se construye tomando aristas alternadamente de cada columna de la grilla. Se puede ver que $|M| = n(m/2)$, pues por cada una de las n filas, hay $m/2$ aristas en el matching.

Si m es impar, consideramos el matching M de la Figura 5.2-(b) si n es par, o el de la Figura 5.2-(c) si n es impar. Estos matchings se obtienen de la misma forma que el del caso m par, pero ahora, además, la paridad permite tomar aristas alternadamente de la última columna. Lo que resta de este argumento vale tanto para n par como impar. Se puede ver que $|M| = n\lfloor m/2 \rfloor + \lfloor n/2 \rfloor$, pues por cada una de las n filas, hay $\lfloor m/2 \rfloor$ aristas horizontales del matching, que se suman a las $\lfloor n/2 \rfloor$ aristas verticales de la última columna. Como m es impar, es $|M| = n(m-1)/2 + \lfloor n/2 \rfloor = nm/2 - n/2 + \lfloor n/2 \rfloor$. Veamos que esta expresión es igual a $\lfloor nm/2 \rfloor$. Si n es par, $|M| = nm/2 - n/2 + n/2 = nm/2$. Si n es impar, $|M| = nm/2 - n/2 + (n-1)/2 = nm/2 - 1/2 = (nm-1)/2$. Como n y m son impares, nm es impar, y por ende $(nm-1)/2 = \lfloor nm/2 \rfloor$.

En cualquier caso, encontramos un matching M de cardinal $\lfloor nm/2 \rfloor$.

□

Llamamos recorrido de tipo *zig-zag por filas* a un camino sobre un grafo grilla que tiene la forma indicada en la Figura 5.3

Teorema 5.6. *Sea $G = (V, E)$ un grafo grilla de n filas y m columnas. Sea P un camino que realiza un recorrido tipo zig-zag por filas. Entonces P es una solución factible de SR para (G, E), y*

$$\text{length}(P) = \begin{cases} 2\tau(G) & \text{si } n \text{ y } m \text{ son impares} \\ 2\tau(G) - 1 & \text{si no} \end{cases}$$

Demostración. Sean n y m la cantidad de filas y columnas, respectivamente, de G. Es claro que P es una solución factible de SR para (G, E), porque toda arista de G es incidente a algún vértice de P.

Este camino recorre cada arista de cada fila, que son $n(m-1)$ en total, más $n-1$ aristas verticales, para pasar entre filas. Por lo tanto,

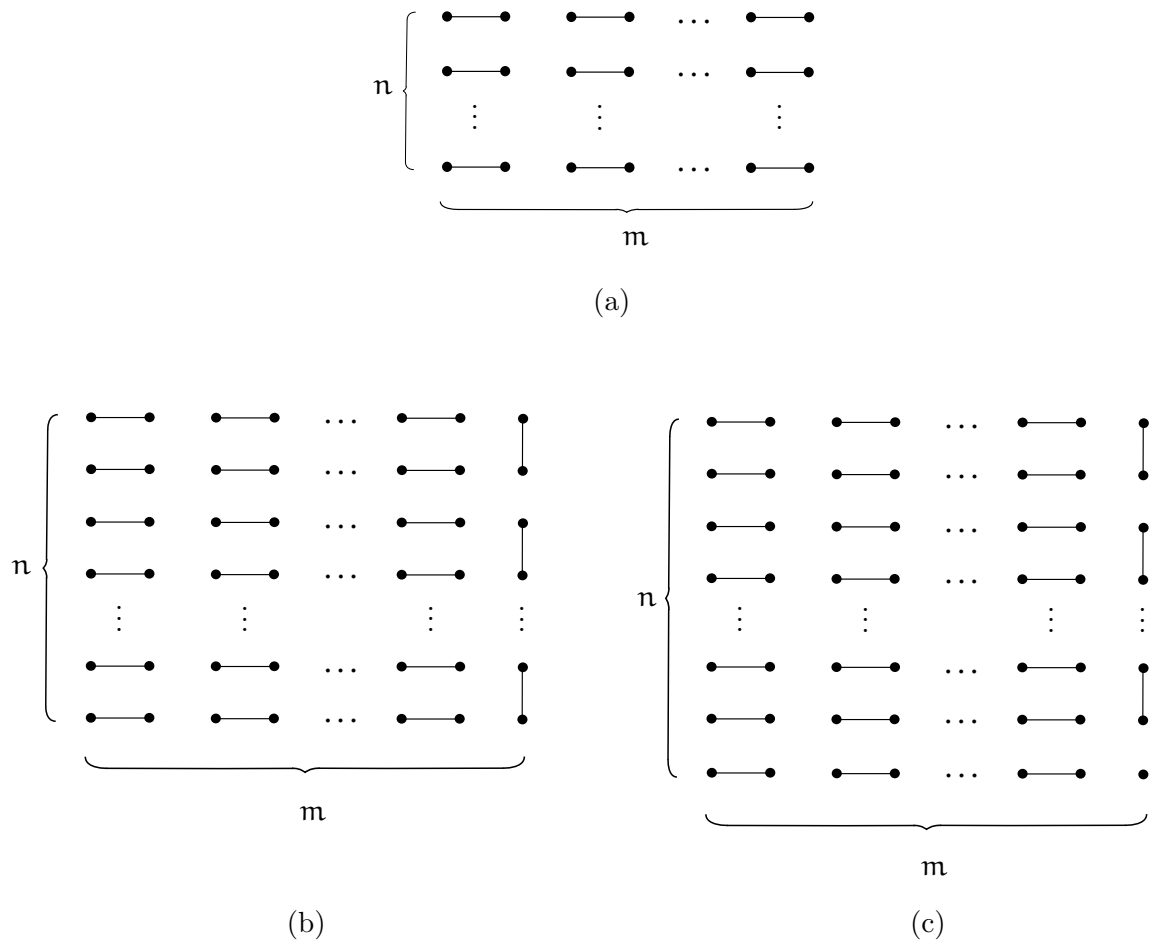


Figura 5.2: Matchings: (a) para m par; (b) para m impar y n par; (c) para m impar y n impar.

$$\begin{aligned}\text{length}(P) &= n(m-1)+n-1\\ &= nm-n+n-1\\ &= nm-1\end{aligned}$$

Si n y m son impares, nm es impar, y $nm-1 = 2\frac{nm-1}{2} = 2\lfloor nm/2\rfloor = 2\tau(G)$. Si no, nm es par, y $nm-1 = 2\frac{nm}{2}-1 = 2\lfloor nm/2\rfloor - 1 = 2\tau(G)-1$.

□

Teorema 5.7. *Existe un algoritmo aproximado $\mathcal{A}$ para* SR *sobre grillas, tal que*

$$\text{length}(\mathcal{A}(G,X)) \leq 2(Z+\overline{k}+1)$$

Demostración. El algoritmo $\mathcal{A}$ es un procedimiento que construye un camino en zig-zag por filas. Notar que este algoritmo no depende de X, pues siempre produce el mismo recorrido en zig-zag por filas. El resultado del mismo es un camino P, que es solución factible de SR para (G,E), y por ende también para (G,X), y tal que $\text{length}(P) \leq 2\tau(G)$. Usando el Lema 5.5 llegamos a que $\text{length}(P) \leq 2(Z+\overline{k}+1)$, que es lo que queríamos. □

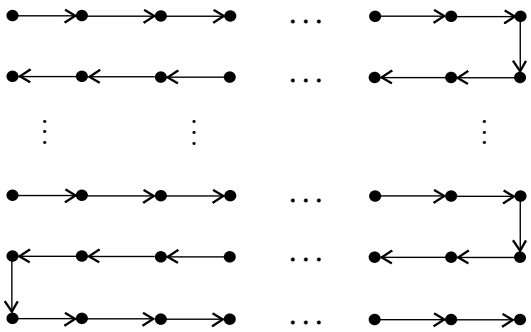


Figura 5.3: Recorrido en zig-zag por filas.

El algoritmo del Teorema 5.7 es útil en el caso en que $\overline{k} = O(1)$, es decir, cuando casi todas las aristas son clientes. En ese caso, el recorrido de la grilla en zig-zag por filas tiene longitud a lo sumo $2Z + O(1)$.

La garantía de aproximación de este teorema se basa en la relación entre la longitud de un camino en zig-zag por filas y $\tau(G)$, probada en el Teorema 5.6. De encontrar una solución factible de SR para (G, E) que sea más corta, podremos dar un algoritmo aproximado para SR sobre grillas que supere al del Teorema 5.7.

A continuación describimos otro recorrido de una grilla, más corto que zig-zag por filas, al que llamamos recorrido de tipo *onda*. Comenzamos en la esquina superior izquierda, y realizamos un recorrido con forma de onda cuadrada, como indica la Figura 5.4. A cada una de estas partes constituyentes del camino las llamamos *período*. Realizamos un período tras otro, mientras sea posible, alternando entre dos filas consecutivas de la grilla. A cada par de filas consecutivas en las que ocurre esta repetición de períodos, lo llamamos *franja*. Al llegar al borde derecho de la grilla (i. e., el final de una franja), será necesario cortar anticipadamente el período actual. Los distintos cortes anticipados que se pueden presentar aparecen en la Figura 5.5.

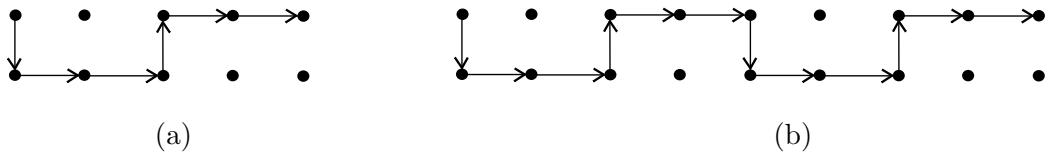


Figura 5.4: (a) Un período. (b) Secuencia de períodos.

Al llegar al borde y no poder continuar el período actual, recorremos dos aristas hacia abajo. A continuación realizamos el mismo recorrido de la franja previa, pero en sentido inverso, y con períodos en forma espejada horizontalmente. De esta forma, recorremos otras dos nuevas filas. Al llegar al borde izquierdo, cortamos anticipadamente un período, y volvemos a bajar. En ocasiones, recorremos dos veces la misma arista, sobre los bordes de la grilla. La Figura 5.6 ilustra esta situación.

Este proceso se repite hasta que se hayan recorrido todas las filas, o hasta que nos quede una única fila sin recorrer. Si recorrimos todas las filas, terminamos el recorrido. Si queda una única fila, la recorremos con un simple camino recto, hasta llegar al borde opuesto. Para ciertas dimensiones específicas de la grilla, este camino recto de la última fila sólo necesita ir hasta la anteúltima columna. Por simplicidad, siempre recorremos la

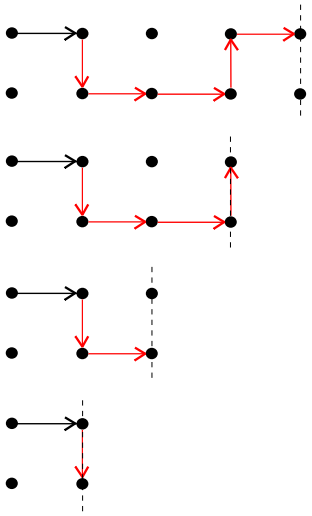


Figura 5.5: Corte anticipado de un período. En rojo se indica el período que se corta. Las líneas punteadas indican el borde derecho de la grilla.

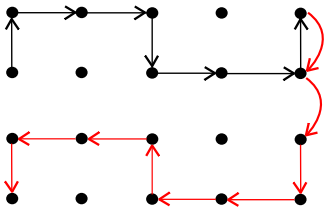


Figura 5.6: El recorrido llega al final de la franja, baja a la siguiente, y realiza el camino inverso.

fila completa.

Notar que cuando la grilla tiene una sola columna, el camino tal como fue descripto repite aristas innecesariamente; en lugar de esto, podría recorrer un simple camino vertical en línea recta, tal como haría un camino en zig-zag por filas. En el caso de una sola fila, el procedimiento se comporta eficientemente, recorriendo un camino recto. Durante el análisis que sigue, excluiremos estos dos casos bordes, que se pueden resolver en forma particular, y asumiremos que hay más de una fila y más de una columna.

Teorema 5.8. *Sea* $G = (V, E)$ *un grafo grilla de* n *filas y* m *columnas, con* $n, m > 1$. *Si un camino* P *de* G *realiza un recorrido tipo onda, entonces* P *es una solución factible de* SR *para* (G, E), *y* $\text{length}(P) = \left(\frac{3}{2} + C(n, m)\right) \tau(G)$, *donde*

$$C(n,m) = \begin{cases} \frac{1}{m} - \frac{4}{nm} & \text{si } n \text{ es par y } m \text{ es par} \\ \frac{3}{2m} - \frac{4}{nm} & \text{si } n \text{ es par y } m \text{ es impar} \\ \frac{1}{m} + \frac{1}{2n} - \frac{3}{nm} & \text{si } n \text{ es impar y } m \text{ es par} \\ \left(1 + \frac{1}{nm-1}\right)\left(\frac{3}{2m} + \frac{1}{2n} - \frac{2}{nm}\right) & \text{si } n \text{ es impar y } m \text{ es impar} \end{cases}$$

Demostración. Comenzamos viendo que es una solución factible de SR para (G, E). Las aristas dentro de cada franja son cubiertas por los períodos de dicha franja. Las aristas

verticales que se ubican entre dos franjas consecutivas, son cubiertas por los recorridos sobre la franja superior e inferior. Para convencerse de esto, mirar fijamente la Figura 5.7.

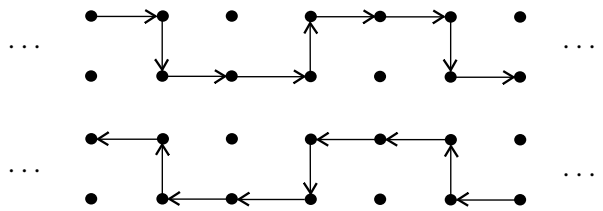


Figura 5.7: Todas las aristas entre dos franjas consecutivas (que no están dibujadas) quedan cubiertas.

Para calcular la longitud de P contamos la cantidad de aristas que recorre el camino, dividiéndolas en dos clases: aquellas que forman parte de un período (quizás inconcluso), y aquellas utilizadas para pasar de una franja a otra. Además, dependiendo de la paridad de n, puede que la última fila sea recorrida con un camino recto, por lo que además de las aristas de este camino, debemos contar las dos aristas necesarias para pasar de la última franja a la última fila.

Cada franja se compone de $m-1$ aristas horizontales y $\lceil m/2 \rceil$ aristas verticales, y hay $\lfloor n/2 \rfloor$ franjas. Adicionalmente, si n es impar, la última fila es recorrida por un camino horizontal. Por lo tanto,

$$\text{length}(P) = \underbrace{\lfloor n/2 \rfloor \, (m - 1 + \lceil m/2 \rceil)}_{\text{aristas en franjas}} + \underbrace{(\lfloor n/2 \rfloor - 1)2}_{\text{aristas entre franjas}} + \underbrace{\delta(n) \, (2 + (m-1))}_{\text{aristas de la última fila}}$$

donde $\delta(n)$ vale 1 si n es impar, y 0 en caso contrario.

Si n es par,

$$\begin{aligned} \text{length}(P) &= (n/2)(m - 1 + \lceil m/2 \rceil) + (n/2 - 1)2 \\ &= (n/2)(m + \lceil m/2 \rceil) - n/2 + n - 2 \\ &= (n/2)(m + \lceil m/2 \rceil) + n/2 - 2 \end{aligned}$$

Dentro de este caso, tenemos otros dos, según la paridad de m. Si m es par,

$$\begin{aligned} \text{length}(P) &= (n/2)(m + m/2) + n/2 - 2 \\ &= (n/2)(3m/2) + n/2 - 2 \\ &= (3/2)(nm/2) + n/2 - 2 \\ &= \frac{3}{2}\tau(G) + \frac{1}{2}n - 2 \end{aligned}$$

En caso contrario, si m es impar,

$$\begin{aligned} \text{length}(P) &= (n/2)(m + (m+1)/2) + n/2 - 2 \\ &= (n/2)(3m/2) + n/4 + n/2 - 2 \\ &= (3/2)(nm/2) + 3n/4 - 2 \\ &= \frac{3}{2}\tau(G) + \frac{3}{4}n - 2 \end{aligned}$$

Pasamos al caso n impar,

$$\begin{aligned}
\text{length}(P) &= ((n-1)/2)(m-1+\lceil m/2\rceil) + ((n-1)/2-1)2 + m + 1 \\
&= ((n-1)/2)(m+\lceil m/2\rceil) - ((n-1)/2) + (n-1) - 2 + m + 1 \\
&= (n/2)(m+\lceil m/2\rceil) - (1/2)(m+\lceil m/2\rceil) + (n-1)/2 + m - 1 \\
&= (n/2)(m+\lceil m/2\rceil) + (1/2)(m-\lceil m/2\rceil) + n/2 - 1/2 - 1 \\
&= (n/2)(m+\lceil m/2\rceil) + \lfloor m/2\rfloor/2 + n/2 - 3/2
\end{aligned}$$

Si m es par,

$$\begin{aligned}
\text{length}(P) &= (n/2)(m+m/2) + (m/2)/2 + n/2 - 3/2 \\
&= (3/2)(nm/2) + m/4 + n/2 - 3/2 \\
&= \frac{3}{2}\tau(G) + \frac{1}{2}n + \frac{1}{4}m - \frac{3}{2}
\end{aligned}$$

Finalmente, si m es impar,

$$\begin{aligned}
\text{length}(P) &= (n/2)(m+(m+1)/2) + ((m-1)/2)/2 + n/2 - 3/2 \\
&= (3/2)(nm/2) + n/4 + m/4 - 1/4 + n/2 - 3/2 \\
&= (3/2)((nm-1)/2 + 1/2) + (3/4)n + (1/4)m - 7/4 \\
&= (3/2)\tau(G) + 3/4 + (3/4)n + (1/4)m - 7/4 \\
&= \frac{3}{2}\tau(G) + \frac{3}{4}n + \frac{1}{4}m - 1
\end{aligned}$$

La expresión $\text{length}(P) = (\frac{3}{2} + C(n,m))\tau(G)$ se obtiene dividiendo cada una de estas expresiones por $\tau(G)$. La fórmula enunciada de $C(n,m)$ se obtiene haciendo las cuentas correspondientes, en función de la paridad de n y m. □

El siguiente lema exhibe algunas propiedades de $C(n,m)$, que nos permitirán concluir que el recorrido de tipo onda es mejor que el recorrido en zig-zag, para todo $n, m > 1$.

Lema 5.6. *La función* $C(n,m)$ *satisface:*

1. $C(n,m) \leq \frac{1}{2}$ *para todo* $n, m > 1$.
2. *Si* $(a_k)_{k\in\mathbb{N}}$ *y* $(b_k)_{k\in\mathbb{N}}$ *son dos sucesiones de números naturales mayores a* 1*, tales que* $\lim_{k\to\infty} a_k = \lim_{k\to\infty} b_k = \infty$*, entonces* $\lim_{k\to\infty} C(a_k, b_k) = 0$.

Demostración.

1. Probaremos que cada uno de los casos de la fórmula de $C(n,m)$ está acotado por $1/2$.

   Supongamos n par y m par. Entonces $n \geq 2$, $m \geq 2$, y vale

   $$C(n,m) = \frac{1}{m} - \frac{4}{nm} < \frac{1}{m} \leq \frac{1}{2}$$

   Supongamos n par y m impar. Entonces $n \geq 2$, $m \geq 3$, y vale

$$C(n,m) = \frac{3}{2m} - \frac{4}{nm} < \frac{3}{2m} \leq \frac{3}{2 \cdot 3} = \frac{1}{2}$$

Supongamos n impar y m par. Entonces $n \geq 3$ y $m \geq 2$. Separamos en dos casos, según sea $m = 2$ o $m \geq 4$. Por un lado,

$$C(n,2) = \frac{1}{2} + \frac{1}{2n} - \frac{3}{2n} = \frac{1}{2} - \frac{1}{n} < \frac{1}{2}$$

Por otro lado, si $m \geq 4$, vale

$$C(n,m) = \frac{1}{m} + \frac{1}{2n} - \frac{3}{nm} < \frac{1}{m} + \frac{1}{2n} \leq \frac{1}{4} + \frac{1}{2 \cdot 3} = \frac{5}{12} < \frac{1}{2}$$

Supongamos n impar y m impar. Entonces $n \geq 3$ y $m \geq 3$. Separamos en dos casos, según sea $m = 3$ o $m \geq 5$. Por un lado,

$$C(n,3) = \left(1 + \frac{1}{3n-1}\right)\left(\frac{3}{2 \cdot 3} + \frac{1}{2n} - \frac{2}{3n}\right) = \frac{3n}{3n-1}\left(\frac{1}{2} - \frac{1}{6n}\right) = \frac{3n}{3n-1} \cdot \frac{3n-1}{6n} = \frac{1}{2}$$

Por otro lado, si $m \geq 5$, vale

$$\begin{aligned} C(n,m) &= \left(1 + \frac{1}{nm-1}\right)\left(\frac{3}{2m} + \frac{1}{2n} - \frac{2}{nm}\right) \\ &\leq \left(1 + \frac{1}{nm-1}\right)\left(\frac{3}{2m} + \frac{1}{2n}\right) \\ &\leq \left(1 + \frac{1}{3 \cdot 5 - 1}\right)\left(\frac{3}{2 \cdot 5} + \frac{1}{2 \cdot 3}\right) \\ &= \frac{15}{14} \cdot \frac{14}{30} \\ &= \frac{1}{2} \end{aligned}$$

2. Llamemos $C_1(n,m)$, $C_2(n,m)$, $C_3(n,m)$, y $C_4(n,m)$ a las expresiones de los cuatro casos de la fórmula de $C(n,m)$. Es fácil ver que $\lim_{k\to\infty} C_i(a_k, b_k) = 0$, usando álgebra de límites. Por lo tanto, cada sucesión $C_i(a_k, b_k)$ tiende a 0, lo que implica que para cada $\varepsilon > 0$ existe un $k_i \in \mathbb{N}$ tal que $|C_i(a_k, b_k)| \leq \varepsilon$ para todo $k \geq k_i$. Veamos que $\lim_{k\to\infty} C(a_k, b_k) = 0$. Sea $\varepsilon > 0$ y tomemos k_1, k_2, k_3 y k_4 como dijimos. Sea $k_0 = \max\{k_1, k_2, k_3, k_4\}$. Sea $k \geq k_0$, y supongamos que es $C(a_k, b_k) = C_i(a_k, b_k)$ (es decir, caemos en el caso i de la definición de C). Entonces, es $|C(a_k, b_k)| = |C_i(a_k, b_k)| \leq \varepsilon$, usando, en la última desigualdad, que $k \geq k_0 \geq k_i$.

□

El ítem 1 del Lema 5.6 nos permite concluir que un recorrido de tipo onda no es peor que un recorrido zig-zag por filas, mientras que la parte 2 muestra que si hacemos tender a n y m a infinito simultaneamente, la longitud del camino tiende a $\frac{3}{2}\tau(G)$, lo cual es un 25 % mejor que un recorrido zig-zag por filas.

Teorema 5.9. *Sea* $G = (V, E)$ *un grafo grilla de* n *filas y* m *columnas. Supongamos* $n, m > 1$. *Existe un algoritmo aproximado* $\mathcal{A}$ *para* SR *sobre grillas, tal que*

$$\text{length}(\mathcal{A}(G, X)) \leq \left(\frac{3}{2} + C(n, m)\right)(Z + \overline{k} + 1)$$

Demostración. El algoritmo $\mathcal{A}$ es un procedimiento que produce un recorrido de tipo onda. La garantía de aproximación se sigue directamente del Teorema 5.8 y el Lema 5.5. □

5.4. Algoritmo aproximado para SR sobre completos

Recurrimos nuevamente a la estrategia de utilizar otro algoritmo aproximado como caja negra. En este caso, nos basaremos en un algoritmo aproximado para VC. Recordemos que $\nu(G)$ denota el máximo cardinal de un matching de un grafo G.

Lema 5.7. *Sea* G *un grafo completo.*

1. *Si* M *es un matching de* $G[X]$, *entonces* $|M| - 1 \leq Z$.
2. $Z \leq 2\nu(G[X]) - 1$

Demostración.

1. Probaremos que cualquier solución factible de SR para (G, X) tiene longitud $|M| - 1$ o más. Escribamos $M = \{e_1, \ldots, e_r\}$. Sea P una solución factible de SR para (G, X). El camino P visita al menos un extremo de cada e_i, porque $M \subseteq X$ y P cubre X. Estos extremos son todos distintos, por ser M un matching. Luego, P visita al menos r vértices distintos, y por ende $\text{length}(P) \geq r - 1 = |M| - 1$.

2. Llamemos $r = \nu(G[X])$ y sea $M = \{e_1, \ldots, e_r\}$ un matching máximo de $G[X]$. Escribamos $e_i = \{u_i, v_i\}$. Consideremos la secuencia $P = \langle u_1, v_1, u_2, v_2, \ldots, u_r, v_r\rangle$, que se obtiene concatenando todas las aristas de M (el orden es indiferente). Como G es completo, P es un camino de G y tiene longitud igual a $2r - 1$, pues r de sus aristas son los $e_i = \{u_i, v_i\}$, y además usa $r - 1$ aristas $\{v_i, u_{i+1}\}$ para conectar pares de aristas e_i y e_{i+1}. Notar que $v_i \neq u_{i+1}$, porque estos pares de vértices son extremos de dos aristas distintas de un matching.

   Por otro lado, como M es máximo, toda arista de $G[X]$ comparte un extremo con alguna arista de M. Esto implica que P cubre todas las aristas de X.

   En definitiva, P es solución factible de SR para (G, X), y por ende $Z \leq \text{length}(P) = 2r - 1 = 2\nu(G[X]) - 1$.

□

Teorema 5.10. *Sea* $\mathcal{A}$ *un algoritmo* α*-aproximado para* VC, *con* α *una constante. Para todo* $\varepsilon > 0$, *existe un algoritmo* $(\alpha + \varepsilon)$*-aproximado* $\mathcal{B}$ *para* SR *sobre completos, que usa a* $\mathcal{A}$ *como caja negra.*

Demostración. Sea $\mathcal{B}$ el Algoritmo 17. El algoritmo usa la constante $\sigma := \lceil (\alpha - 1)/\varepsilon + 1 \rceil \in \mathbb{Z}_{>0}$, que a primera vista resulta extraña pero toma sentido más adelante en la demostración.

El algoritmo se basa en la siguiente propiedad. Si $G[X]$ *no* tiene un matching de cardinal σ o mayor, entonces es $\nu(G[X]) \leq \sigma - 1$, y por el ítem 2 del Lema 5.7 se tiene $Z \leq 2(\sigma - 1) - 1 = 2\sigma - 3$. Luego, en este caso, Z es igual o más chico que una constante. Si no, $G[X]$ tiene un matching de cardinal σ o mayor, y por el ítem 1 del Lema 5.7 vale $Z \geq \sigma - 1$. Por ende, en este caso, Z es igual o más grande que la constante $\sigma - 1$.

Con esto en mente, pasamos a explicar la idea clave del algoritmo. Éste separa las entradas en dos casos, a través de la guarda de la línea 1, que verifica que no exista un matching grande, para así concluir que Z tampoco es grande. En este caso, se ejecutan las líneas 2-5, que calculan una solución exacta, lo cual se puede hacer en tiempo polinomial, porque Z es suficientemente chico. Si la guarda es falsa, se ejecutan las líneas 6-8, que computan una solución aproximada, y al ser Z suficientemente grande podemos garantizar que la aproximación es buena.

Algoritmo 17 Algoritmo aproximado del Teorema 5.10.

Entrada: Una instancia (G, X) de SR sobre completos.
Salida: Una solución factible de SR para (G, X).

```
1: if G[X] no tiene un matching de cardinal σ o mayor then
2:     for i ← 1, ..., 2σ − 3 do
3:         for each P camino (de G) de longitud i, formado sólo por vértices de G[X] do
4:             if P cubre X then
5:                 return P
6: Calcular S = A(G[X]).
7: Sea P una permutación arbitraria de S.
8: return P
```

Veamos que el algoritmo es polinomial. Comenzamos por las líneas 1-5. Podemos verificar si $G[X]$ tiene un matching de cardinal σ o mayor computando un matching máximo de $G[X]$, lo cual es posible hacer en tiempo polinomial [6], y luego verificando si su cardinal es mayor o igual a σ.

El ciclo 3-5 tiene $O((2k)^{i+1})$ iteraciones, pues como $G[X]$ tiene a lo sumo $2k$ vértices, hay a lo sumo $\underbrace{(2k)\dots(2k)}_{i+1 \text{ veces}}$ caminos de longitud i, formado sólo por vértices de $G[X]$. Cada iteración de este ciclo corre en tiempo $O(i + k)$, pues la verificación de la línea 4 implica recorrer los i vértices de P, marcando las aristas de X que son cubiertos. Como $i = O(\sigma) = O(1)$, es $i + k = O(k)$. Luego, cada ejecución del ciclo 3-5 toma tiempo $O(k(2k)^{i+1})$, que es $O(\text{poly}(k))$ puesto que $i = O(1)$. Este ciclo es ejecutado $2\sigma - 3 = O(\sigma)$ veces por el ciclo 2-5, con lo cual este ciclo externo corre en $O(\sigma\ \text{poly}(k)) = O(\text{poly}(k))$.

En definitiva, las líneas 1-5 corren en tiempo $O(\text{poly}(k))$. Las líneas 6-8 son claramente polinomiales. Luego, el algoritmo es polinomial.

Veamos que es correcto. Si la guarda de la línea 1 es verdadera, $G[X]$ no tiene un matching de cardinal σ o mayor, y por lo tanto es $\nu(G[X]) \leq \sigma - 1$. Luego, por la parte 2 del Lema 5.7, $Z \leq 2(\sigma - 1) - 1 = 2\sigma - 3$. Esto indica que para encontrar una solución óptima, nos alcanza con probar todos los caminos de G, de longitud menor o igual a

$2\sigma - 3$. Observando que una solución óptima sólo usa vértices que son extremos de aristas de X, podemos limitarnos a probar caminos formados sólo por vértices de $G[X]$. Esto es exactamente lo que hace el ciclo 2-5.

Si la guarda de la línea 1 es falsa, las líneas 6-8 se encargan de devolver una solución factible aproximada. Notar que como G es completo, cualquier permutación de un vertex cover de $G[X]$ es una solución factible de SR para (G, X).

Finalmente, veamos que la solución P que devuelve el algoritmo, satisface la garantía de aproximación afirmada. Si la ejecución sigue la rama de las líneas 2-5, P es una solución óptima, y no hay nada que probar. Supongamos que el resultado devuelto es el de las líneas 6-8. En ese caso, $G[X]$ tiene un matching de tamaño σ. Por la parte 1 del Lema 5.7, es $Z \geq \sigma - 1$. Es $\sigma \geq (\alpha - 1)/\varepsilon + 1$, con lo cual $\varepsilon(\sigma - 1) \geq \alpha - 1$. Todo esto implica que $\varepsilon Z \geq \alpha - 1$. Luego

$$\begin{aligned}
\text{length}(P) &= |S| - 1 && (P \text{ es una permutación de } S)\\
&\leq \alpha\ \tau(G[X]) - 1 && (\mathcal{A} \text{ es } \alpha\text{-aproximado})\\
&= \alpha(Z + 1) - 1 && (\text{Lema 3.4})\\
&= \alpha\ Z + \alpha - 1\\
&\leq \alpha\ Z + \varepsilon Z && (\varepsilon Z \geq \alpha - 1)\\
&= (\alpha + \varepsilon)Z
\end{aligned}$$

□

Es interesante observar que para construir Algoritmo 17 necesitamos conocer el valor de α, a diferencia de otros algoritmos aproximados estudiados que ignoraban el factor de aproximación de los algoritmos aproximados que usaban en forma de cajas negras. En general, esto no es un inconveniente, pues cuando utilizamos un algoritmo aproximado conocemos algún factor de aproximación concreto del mismo. Sin embargo, si usamos un factor de aproximación constante α de la caja negra $\mathcal{A}$ que no es ajustado (es decir, que se puede mejorar), entonces la garantía de aproximación de $\mathcal{B}$ que probamos queda ligada a ese valor de α, y no al mínimo factor que realiza $\mathcal{A}$. Esto es, la garantía de aproximación demostrada está limitada por la eficacia de $\mathcal{A}$ *conocida*. El siguiente teorema muestra que el Algoritmo 17 tiene la propiedad notable de adaptarse al mínimo factor de aproximación de $\mathcal{A}$, aún si el factor que conoce, y que utiliza para construir σ, no es el mejor.

Teorema 5.11. *Sea $\mathcal{A}$ un algoritmo α-aproximado para VC, con α una constante. Supongamos que no conocemos α, pero que sí conocemos una constante $\beta > \alpha$, tal que $\mathcal{A}$ es β-aproximado. Para todo $\varepsilon > 0$, existe un algoritmo $(\alpha + \varepsilon)$-aproximado $\mathcal{B}$ para SR sobre completos, que usa a $\mathcal{A}$ como caja negra, y que no usa α.*

Demostración. Utilizamos el Algoritmo 17, poniendo $\sigma := \lfloor(\beta - 1)/\varepsilon + 1\rfloor$. Lo único que hay que ver es que la garantía de aproximación es $\alpha + \varepsilon$. Para esto repetimos la cuenta del Teorema 5.10, pero ahora usamos que $\varepsilon Z \geq \beta - 1 > \alpha - 1$. □

Como el mejor algoritmo aproximado conocido para VC es un 2-aproximado, y ese factor es ajustado, no necesitamos apelar al Teorema 5.11 si usamos ese algoritmo.

5.5. Algoritmos aproximados menos efectivos

En esta sección damos dos algoritmos aproximados para SR sobre grillas, que se basan en aproximaciones de la instancia $(K(X), \overline{dist})$ de PTSP. Pese a que ninguno de estos algoritmos mejora a los anteriores, los exhibimos igualmente, porque, por un lado, son una prueba de que la alternativa de aproximar la instancia $(K(X), \overline{dist})$ de PTSP no conduce a mejores resultados, y por otro lado, porque motiva algunas ideas interesantes.

Recordemos el Corolario 4.5, que afirma que $Z \geq \lfloor k/3 \rfloor - 1$ si G es una grilla. Durante esta sección usaremos una consecuencia de este resultado. Notemos que la antedicha desigualdad implica que $Z > (k/3) - 2$, o equivalentemente $k < 3Z + 6$. Como k y Z son enteros, es $k \leq 3Z + 5$.

5.5.1. Aproximando la instancia $(K(X), \overline{dist})$ de PTSP

Teorema 5.12. *Sea $\mathcal{A}$ un algoritmo α-aproximado para PTSP métrico. Para todo $\varepsilon > 0$ existe un algoritmo $(7\alpha + \varepsilon)$-aproximado $\mathcal{B}$ para SR sobre grillas, para instancias de $k \geq (24/\varepsilon)\alpha + 6$ clientes, que usa a $\mathcal{A}$ como caja negra.*

Demostración. Sea $\mathcal{B}$ el Algoritmo 18. El algoritmo $\mathcal{B}$ es claramente polinomial en el tamaño de la entrada. Es importante notar que en el paso 1, la entrada $(K(X), \overline{dist})$ de $\mathcal{A}$ es, efectivamente, una instancia de PTSP métrico, como indica el Teorema A.1. Por otro lado, el resultado Q del algoritmo es una solución factible de SR para (G, X), como dice el Lema 5.2.

Algoritmo 18 Algoritmo aproximado del Teorema 5.12.

Entrada: Una instancia (G, X) de SR sobre grillas.
Salida: Una solución factible de SR para (G, X).
1: Calcular $P = \mathcal{A}(K(X), \overline{dist})$.
2: A partir de P, construir Q como indica el Lema 5.2.
3: **return** Q

Calculemos el valor de la respuesta $Q = \mathcal{B}(G, X)$,

$$\begin{aligned}
\text{length}(Q) &\leq \text{length}_{\overline{dist}}(P) && \text{(Lema 5.2)} \\
&\leq \alpha \text{PTSP}^*(K(X), \overline{dist}) && (\mathcal{A} \text{ es } \alpha\text{-aproximado}) \\
&\leq \alpha(Z + 2(k-1)) && \text{(Teorema 5.1)} \\
&\leq \alpha(Z + 2((3Z+5) - 1)) \\
&= \alpha(7Z + 8)
\end{aligned}$$

La hipótesis $k \geq (24\alpha)/\varepsilon + 6$ está fabricada de modo tal que implique $\lfloor k/3 \rfloor - 1 \geq 8\alpha/\varepsilon$, y así poder usar el Corolario 4.5 para concluir que $\varepsilon Z \geq 8\alpha$. Luego,

$$\text{length}(Q) \leq 7\alpha Z + \varepsilon Z = (7\alpha + \varepsilon)Z$$

que es lo que queríamos demostrar. □

5.5.2. Revisión y aplicación de la clásica aproximación de PTSP

Podemos explotar un poco más a las propiedades de la instancia $(K(X), \overline{dist})$. La observación clave es que si G es grilla, entonces $\overline{dist}$ sobre X no sólo satisface la desigualdad triangular, sino que lo hace en forma estricta. Esto es lo que afirma el Teorema A.2. ¿Cómo se aprovecha que la desigualdad triangular para $\overline{dist}$ sea estricta? Comencemos recordando la clásica 2-aproximación de PTSP métrico [9, p. 131] (que, en realidad, suele presentarse como una aproximación de TSP). El Algoritmo 19 es esta aproximación. La heurística para TSP es idéntica, salvo que se quita el paso 5.

Algoritmo 19 Heurística de duplicado de aristas para PTSP métrico.

Entrada: Una instancia (G, c) de PTSP métrico.
Salida: Un camino hamiltoniano de G.

1: Computar un árbol generador mínimo T de G con pesos c.
2: Cambiar cada arista de T por dos aristas dirigidas en sentidos opuestos. Sea T' el árbol resultante.
3: Computar un circuito euleriano C de T'.
4: Acortar el circuito, eliminando vértices repetidos. Sea C' el circuito hamiltoniano de G resultante.
5: Quitarle cualquier arista a C' de modo tal que se transforme en un camino hamiltoniano de G.
6: **return** C'

El análisis clásico de este algoritmo comienza observando que $\text{weight}_c(T) \leq \text{PTSP}^*(G, c)$, pues toda solución factible de PTSP para (G, c) es un camino hamiltoniano de G, y por lo tanto también es un árbol generador de G. Entonces

$$\text{length}_c(C) = \text{weight}_c(T') = 2\ \text{weight}_c(T) \leq 2\ \text{PTSP}^*(G, c)$$

La otra parte del análisis consiste en mostrar que al acortar C, en la línea 4 del algoritmo, nos queda un camino de peso menor o igual. Asumiendo que c satisface la desigualdad triangular, cada vez que tenemos una secuencia de tres vértices $\langle u, v, w\rangle$ en el circuito, podemos quitar el vértice v del medio, y garantizar que la longitud no empeora, pues $c(u, w) \leq c(u, v) + c(v, w)$.

¿Qué sucede si sabemos que, más aún, $c(u, w) \leq c(u, v) + c(v, w) - 1$? En este caso, podemos asegurar que el cambio reduce el peso del circuito en, al menos, una unidad. Por lo tanto, con esta versión más fuerte de desigualdad triangular, podemos garantizar que $\text{length}_c(C') \leq \text{length}_c(C) - r$, donde r es la cantidad de veces que removemos un vértice de C. A continuación calculamos r.

Sea $n = |V(G)|$. Como T es árbol generador de G, tiene n vértices y $n - 1$ aristas. Luego, T' tiene $2(n-1) = 2n-2$ aristas. Como C es un circuito hamiltoniano de T', tiene la misma cantidad de aristas que T'. Escribamos, entonces, $C = \langle u_1, \ldots, u_{2n-2}, u_1\rangle$. En el camino $\langle u_1, \ldots, u_{2n-2}\rangle$, cada uno de los n vértices de G aparece una vez, y los demás vértices del camino son repetidos. Entonces, se remueven $r = (2n-2)-n = n-2$ vértices de C.

Teorema 5.13. *Sea* (G, c) *una instancia de* PTSP*, tal que* c *cumple* $c(u, w) \leq c(u, v) + c(v, w) - 1$ *para todo* $u, v, w \in V(G)$*. En estas condiciones, el Algoritmo 19 produce un camino hamiltoniano* P *de peso* $\text{length}_c(P) \leq 2\ \text{PTSP}^*(G, c) - (|V(G)| - 2)$.

Para la instancia $(K(X), \overline{\text{dist}})$, que satisface las hipótesis del teorema, se puede refinar el análisis un poco más, notando que $\overline{\text{dist}}(e, f) \geq 2$ para cualesquiera $e, f \in X$, $e \neq f$, con lo cual el paso 5 del Algoritmo 19 reduce la longitud del camino en, al menos, dos unidades más.

Teorema 5.14. *El Algoritmo 19 sobre la instancia* $(K(X), \overline{\text{dist}})$ *produce un camino hamiltoniano* P *de peso* $\text{length}_{\overline{\text{dist}}}(P) \leq 2\ \text{PTSP}^*(K(X), \overline{\text{dist}}) - k$.

Este análisis muestra que este algoritmo es mejor que una 2-aproximación. Si ponemos este algoritmo aproximado como $\mathcal{A}$ en el Algoritmo 18, obtenemos el siguiente resultado.

Teorema 5.15. *Para cada* $\varepsilon > 0$ *existe un algoritmo* $(11+\varepsilon)$*-aproximado* $\mathcal{B}$ *para* SR *sobre grillas, para instancias de* $k \geq 33/\varepsilon + 6$ *clientes, que usa al Algoritmo 19 como caja negra.*

Demostración. El algoritmo $\mathcal{B}$ es idéntico al Algoritmo 18, pero utilizando al Algoritmo 19 como $\mathcal{A}$. El análisis es análogo, excepto que ahora tenemos que $\text{length}_{\overline{\text{dist}}}(P) \leq 2\ \text{PTSP}^*(K(X), \overline{\text{dist}}) - k$. Teniendo en cuenta esto, el algoritmo nos devuelve una solución factible $Q = \mathcal{B}(G, X)$ que satisface $\text{length}(Q) \leq 2(Z + 2(k-1)) - k = 2Z + 3k - 4$. Como G es grilla, vale $k \leq 3Z + 5$, con lo cual $\text{length}(Q) \leq 2Z + 3(3Z + 5) - 4 = 11Z + 11$.

Si $k \geq 33/\varepsilon + 6$ entonces $\lfloor k/3 \rfloor - 1 \geq 11/\varepsilon$, y usando el Corolario 4.5, llegamos a que $\varepsilon Z \geq 11$. Luego,

$$\text{length}(Q) \leq 11Z + \varepsilon Z = (11 + \varepsilon)Z$$

como queríamos. □

Lamentablemente este algoritmo aproximado no mejora al mejor algoritmo que se puede obtener vía el Teorema 5.12. La mejor aproximación conocida para PTSP métrico es la de Hoogeveen [12], que es de factor 3/2. Eligiendo a esta aproximación como $\mathcal{A}$ en el Teorema 5.12, obtenemos un algoritmo $(10{,}5 + \varepsilon)$-aproximado para SR sobre grillas. Esto es mejor que el $(11 + \varepsilon)$-aproximado que ofrece el Teorema 5.15.

Pese a que el nuevo análisis no termina siendo útil para obtener un mejor algoritmo aproximado para SR sobre grillas, en otro contexto podría llegar a ser útil tener presente que ante una hipótesis más fuerte que la desigualdad triangular, podemos obtener mejores garantías de aproximación de este simple algoritmo aproximado.

Capítulo 6

Conclusiones

En esta tesis estudiamos el problema STAR ROUTING, que puede ser sintetizado como una combinación entre el problema de ruteo de un vehículo y el problema de computar un vertex cover en un grafo. En menor medida estudiamos el problema afín STOPS SELECTION.

En primer lugar, logramos caracterizar SS, al probar que es equivalente al problema de vertex cover, tanto desde el punto de vista de algoritmos exactos como aproximados. Dimos una transformación polinomial de VC a SS, y mostramos un algoritmo exacto para SS basado en un algoritmo para VC. Este algoritmo es supra-polinomial si la entrada admite cualquier tipo de grafo, pero polinomial para el caso de grafos bipartitos. Por otro lado, probamos que el mínimo factor de aproximación de SS es igual al mínimo factor de aproximación de VC. Esto nos da un algoritmo 2-aproximado para SS.

Con respecto a SR, logramos demostrar resultados de complejidad sobre varias clases de grafos, concluyendo que es **NP-completo** en el caso general, sobre grafos grilla y sobre grafos completos, y que es polinomial sobre árboles. Debemos tener en cuenta que en el caso de grafos grilla se probó que el problema es difícil asumiendo una representación no tradicional de la entrada, aunque la misma es razonable. Para el caso de grafos completos, realizamos una reducción desde VC. Para el caso de árboles, dimos un algoritmo de programación dinámica que corre en tiempo lineal.

Los algoritmos exactos para SR que propusimos y analizamos, son interesantes desde el punto de vista teórico, pero no son suficientemente rápidos como para satisfacer necesidades prácticas. El mejor algoritmo encontrado tiene complejidad $O(k^4 2^k)$, aunque utiliza $O(k^2 2^k)$ memoria, haciéndolo impráctico. Utilizando la técnica de backtracking, llegamos a un algoritmo $O(k \cdot k!)$ que, aunque usa $O(k)$ memoria, sigue siendo exponencial. La experimentación mostró que, luego de implementar podas mediante funciones de acotación los tiempos de ejecución se reducen notablemente, pero aún así estos algoritmos no son capaces de resolver instancias de más de 25 clientes, en menos de una hora. Realizando simples proyecciones podemos concluir que instancias grandes demorarían una cantidad de tiempo en el orden de las horas o días.

En contraste, los algoritmos aproximados para SR propuestos son satisfactorios, en el sentido de que garantizan soluciones factibles cuyo valor es a lo sumo un factor constante más grande que el óptimo. De ellos podemos concluir, en primer lugar, que SR es aproximable a no más de un factor constante y, en segundo lugar, que todas las restricciones consideradas de SR se pueden aproximar con un factor de aproximación pequeño. Específicamente, pudimos obtener un $(9/2)$-aproximado sobre grafos grilla, y un $(2+\varepsilon)$-aproximado

sobre grafos completos.

Trabajo futuro

Al no existir trabajos previos relacionados a SR en la literatura, realizamos en esta tesis un estudio inicial del problema con respecto a los enfoques clásicos en los trabajos de optimización combinatoria. Es decir, estudiamos su complejidad y desarrollamos algoritmos (exactos y aproximados) para el problema. Más allá de esta tesis, son numerosas las incógnitas que nos planteamos a lo largo del recorrido. A continuación listamos algunas de esas preguntas abiertas, y posibles vías de trabajo a futuro:

- Demostrar, si es cierto, que SR sobre grillas es **NP-completo**, asumiendo una representación tradicional de la entrada.
- Los algoritmos exactos que propusimos admiten cualquier tipo de grafo de entrada. ¿Existirá un algoritmo exacto específico para SR sobre grillas, que tenga un mejor tiempo de ejecución?
- Buscar e implementar mejores funciones de acotación.
- Implementar algunos de los algoritmos aproximados propuestos y utilizarlos para producir soluciones iniciales en los algoritmos exactos.
- Buscar algoritmos aproximados para SR que no utilicen otros algoritmos aproximados como cajas negras.
- Buscar algoritmos aproximados aleatorizados. En la demostración del ítem 2 del Teorema 5.2, partimos de un vertex cover S y proponemos cierta permutación de S como solución factible de PTSP, a través de una solución óptima P de SR. Esta permutación tiene, como solución de PTSP, valor menor o igual a $\text{length}(P) + 2(|S| - 1)$, aunque esta cota es para el peor caso. ¿Hay alguna forma de elegir S o P de modo tal de mejorar esta cota (bajo ciertas hipótesis probabilísticas sobre X)?
- Implementar metaheurísticas para SR y SS, y comparar su eficacia contra los algoritmos aproximados.
- Calcular, si es posible, y en forma exacta, $\text{SR}^*(G, E)$ con G un grafo grilla. Conjeturamos que un recorrido de tipo onda o similar realiza la longitud mínima.
- Explorar generalizaciones o variantes del problema. Por ejemplo, buscar un circuito de costo mínimo, agregar un punto de partida del vehículo de entregas, admitir más de un vehículo de entregas, o admitir pesos en el grafo.
- Desde un punto de vista de negocios, evaluar cuán conveniente puede llegar a ser un sistema de entregas como el que modela SR. Si es factible, proponerlo a una gran empresa y hacerse millonario.

Apéndice A

Propiedades de $\overline{\text{dist}}$ y dist sobre aristas

En este apéndice demostramos algunas propiedades de las funciones dist sobre aristas y $\overline{\text{dist}}$, introducidas en la Definición 4.2 y la Definición 5.1, respectivamente. El estudio de estas funciones se ve motivado por los algoritmos aproximados para SR sobre grillas basados en la aproximación de la instancia $(K(X), \overline{\text{dist}})$ de PTSP. Para poder aproximar $(K(X), \overline{\text{dist}})$, dicha instancia debe ser métrica, i. e., $\overline{\text{dist}}$ debe cumplir la desigualdad triangular. Esta propiedad es la que se utiliza en el Algoritmo 18 (para el Teorema 5.12). Luego de eso, en el Teorema 5.15 se propone un algoritmo similar, excepto que utiliza un procedimiento que aproxima $(K(X), \overline{\text{dist}})$ basándose en el hecho de que, en el caso de grafos grilla, $\overline{\text{dist}}$ satisface la desigualdad triangular en forma estricta. Adicionalmente, el Lema 5.1 utiliza una relación entre $\overline{\text{dist}}$ y dist sobre aristas, que también demostramos aquí.

A.1. Propiedades generales

Lema A.1. *Sean* e *y* f *aristas de un grafo. Entonces* $\overline{\text{dist}}(e, f) \leq \text{dist}(e, f) + 2$.

Demostración. Sean e_1 y e_2 los extremos de e, y f_1 y f_2 los de f. Supongamos, sin pérdida de generalidad, que $\text{dist}(e, f) = \text{dist}(e_1, f_1)$. Sean $i, j \in \{1, 2\}$ cualesquiera. Un camino factible de e_i a f_j consiste en ir de e_i a e_1, luego de e_1 a f_1, y finalmente de f_1 a f_j. Entonces

$$\begin{aligned} \text{dist}(e_i, f_j) &\leq \text{dist}(e_i, e_1) + \text{dist}(e_1, f_1) + \text{dist}(f_1, f_j) \\ &\leq \text{dist}(e_1, f_1) + 2 \\ &= \text{dist}(e, f) + 2 \end{aligned}$$

Como $\overline{\text{dist}}(e, f)$ se realiza para algún $\text{dist}(e_i, f_j)$, lo anterior implica que $\overline{\text{dist}}(e, f) \leq \text{dist}(e, f) + 2$. □

Teorema A.1. $\overline{\text{dist}}$ *cumple la desigualdad triangular.*

Demostración. Dadas tres aristas e, f y g de un grafo, queremos ver que $\overline{\text{dist}}(e, g) \leq \overline{\text{dist}}(e, f) + \overline{\text{dist}}(f, g)$. Sean e_1 y g_1 extremos de e y g, respectivamente, tales que $\overline{\text{dist}}(e, g) =$

$\text{dist}(e_1, g_1)$. Análogamente, sean e_2, f_1, f_2, g_2 tales que $\overline{\text{dist}}(e, f) = \text{dist}(e_2, f_1)$ y $\overline{\text{dist}}(f, g) = \text{dist}(f_2, g_2)$.

Se puede ver que $\overline{\text{dist}}(e, f) = \text{dist}(e_2, f_1) \geq \text{dist}(e_1, f_1)$, porque e_1 también es extremo de e. Análogamente, $\overline{\text{dist}}(f, g) = \text{dist}(f_2, g_2) \geq \text{dist}(f_1, g_1)$ porque f_1 también es extremo de f y g_1 lo es de g. Luego

$$\begin{aligned}
\overline{\text{dist}}(e, f) + \overline{\text{dist}}(f, g) &\geq \text{dist}(e_1, f_1) + \text{dist}(f_1, g_1) \\
&\geq \text{dist}(e_1, g_1) && \text{(dist sobre vértices es una métrica)} \\
&= \overline{\text{dist}}(e, g)
\end{aligned}$$

que es lo que queríamos probar. □

Como curiosidad, vale la pena mencionar que, el Teorema A.1 no vale si cambiamos $\overline{\text{dist}}$ por dist. En particular, no vale cuando el grafo es grilla.

Observación A.1. dist sobre aristas no necesariamente cumple la desigualdad triangular.

Demostración. Consideremos el grafo grilla de la Figura A.1. Se tiene $\text{dist}(e, g) = 2L + 1$, mientras que $\text{dist}(e, f) + \text{dist}(f, g) = L + L = 2L$.

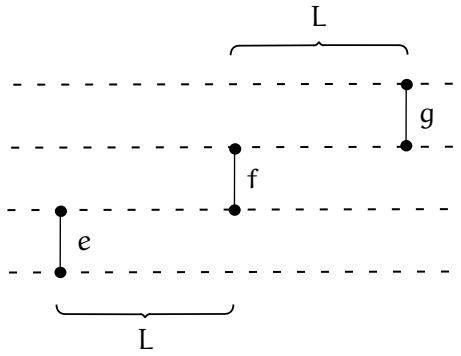


Figura A.1: Grafo grilla con tres aristas distinguidas e, f y g.

□

De todas formas, dist sobre aristas está cerca de cumplir la desigualdad triangular, como indica el siguiente resultado.

Lema A.2. *Sean e, f y g aristas de un grafo. Entonces* $\text{dist}(e, g) \leq \text{dist}(e, f) + \text{dist}(f, g) + 1$.

Demostración. Sean e_1 y f_1 extremos de e y f respectivamente, tales que $\text{dist}(e, f) = \text{dist}(e_1, f_1)$. Análogamente, sean f_2 y g_1 tal que $\text{dist}(f, g) = \text{dist}(f_2, g_1)$. Un camino factible de e a g consiste en ir de e_1 a f_1, luego de f_1 a f_2, y finalmente de f_2 a g_1. Entonces

$$\text{dist}(e, g) \leq \text{dist}(e_1, f_1) + 1 + \text{dist}(f_2, g_1) = \text{dist}(e, f) + \text{dist}(f, g) + 1$$

□

La desigualdad del Lema A.2 es ajustada, como muestra el ejemplo de la Observación A.1.

A.2. Propiedades sobre grafos grilla

Lema A.3. *Sean* e *y* f *aristas de un grafo grilla. Entonces* $\overline{\text{dist}}(e, f) \geq \text{dist}(e, f) + 1$.

Demostración. Basta ver que hay algún camino mínimo, entre un extremo de e y otro de f, de longitud mayor o igual a $\text{dist}(e, f) + 1$.

Sean e_1 y f_1 extremos de e y f, respectivamente, tales que $\text{dist}(e, f) = \text{dist}(e_1, f_1)$. Sea e_2 el extremo opuesto de e_1. La clave es que al ser el grafo una grilla, vale que $\text{dist}(e_2, f_1) \neq \text{dist}(e_1, f_1)$, pues e_1 y e_2 son vértices adyacentes. Luego, debe ser $\text{dist}(e_2, f_1) > \text{dist}(e_1, f_1) = \text{dist}(e, f)$, o sea que $\text{dist}(e_2, f_1) \geq \text{dist}(e, f) + 1$. Como $\overline{\text{dist}}(e, f) \geq \text{dist}(e_2, f_1)$, el resultado se sigue. □

Corolario A.1. *Sean* e *y* f *aristas de un grafo grilla. Entonces* $\overline{\text{dist}}(e, f) \in \{\text{dist}(e, f) + 1, \text{dist}(e, f) + 2\}$.

Definición A.1. Sea G un grafo grilla. Sean e y f dos aristas de G. Decimos que e y f están alineados en G, si al considerar una orientación de G en la que e es una arista horizontal, resulta que f es una arista horizontal que tiene sus extremos en las mismas columnas que los extremos de e. La Figura A.2 muestra tres ejemplos de esta definición.

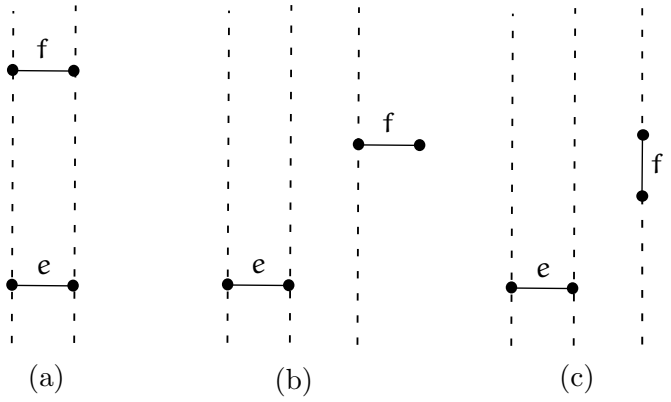


Figura A.2: (a) e y f están alineados. (b), (c) e y f no están alineados.

La definición es buena, en el sentido de que no depende de cuál de las dos orientaciones de G que dejan a e horizontal elijamos. Notar que tampoco depende del orden en que consideremos e y f.

Lema A.4. *Sean* e *y* f *aristas de un grafo grilla. Entonces* $\overline{\text{dist}}(e, f) = \text{dist}(e, f) + 1$ *si y sólo si* e *y* f *están alineados.*

Demostración. ($\Leftarrow$) Es obvio.

($\Rightarrow$) Demostramos el contrarrecíproco. Supongamos que e y f no están alineados. Consideremos una orientación de la grilla que deje horizontal a e. Separamos en dos casos, según f sea horizontal o vertical, en esta orientación. La Figura A.3 ilustra los dos casos. Se puede ver que siempre es $\overline{\text{dist}}(e, f) = \text{dist}(e, f) + 2$.

□

Lema A.5. *Sean* e, f *y* g *aristas de un grafo grilla. Si* e *y* f *están alineados o si* f *y* g *están alineados, entonces* $\text{dist}(e, g) \leq \text{dist}(e, f) + \text{dist}(f, g)$.

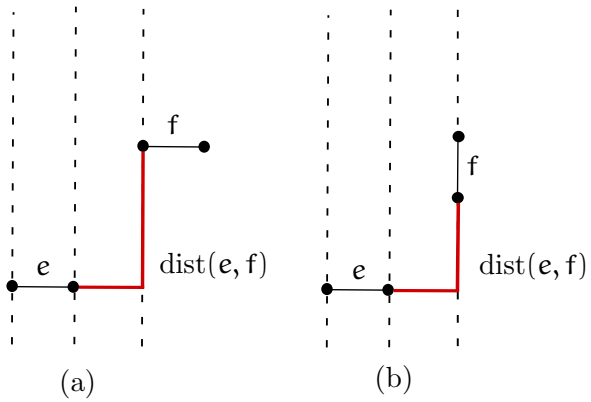


Figura A.3: (a) f es horizontal. (b) f es vertical.

Demostración. Supongamos que e y f están alineados. Sean f_1 y g_1 extremos de f y g, respectivamente, tal que $\text{dist}(f, g) = \text{dist}(f_1, g_1)$. Como e y f están alineados, consideramos e_1, el extremo de e en la misma columna que f_1. Entonces $\text{dist}(e, f) = \text{dist}(e_1, f_1)$. Esta situación se ilustra en la Figura A.4. Un camino factible de e a g consiste en ir de e_1 a f_1, y luego de f_1 a g_1. Entonces $\text{dist}(e, g) \leq \text{dist}(e_1, f_1) + \text{dist}(f_1, g_1) = \text{dist}(e, f) + \text{dist}(f, g)$. Notar que es indistinto si f y g están o no alineados.

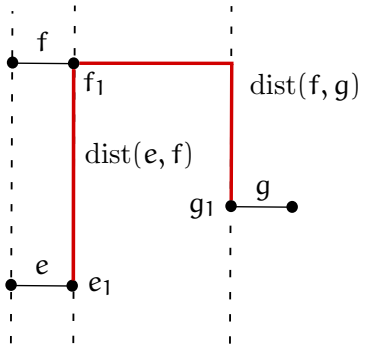


Figura A.4: e y f están alineados.

Pasamos al otro caso. Si f y g están alineados, usamos el caso anterior y la simetría de dist,

$$\begin{aligned}\text{dist}(e, g) &= \text{dist}(g, e)\\ &\leq \text{dist}(g, f) + \text{dist}(f, e)\\ &= \text{dist}(e, f) + \text{dist}(f, g)\end{aligned}$$

□

Teorema A.2. $\overline{\text{dist}}$ *sobre grafos grilla cumple la desigualdad triangular en forma estricta. Esto es, para cualesquiera tres aristas* e, f *y* g *de un grafo grilla, vale*

$$\overline{\text{dist}}(e, g) \leq \overline{\text{dist}}(e, f) + \overline{\text{dist}}(f, g) - 1$$

Demostración. Separamos en casos, según las aristas estén o no alineados entre sí.

1. e y g están alineados

   Entonces, por el Lema A.4, es $\overline{\text{dist}}(e, g) = \text{dist}(e, g) + 1$. Además, f está alineado con e y g, o no está alineado con ninguno.

   a) f está alineado con e y g

   Por el Lema A.4, es $\overline{\text{dist}}(e, f) = \text{dist}(e, f) + 1$ y $\overline{\text{dist}}(f, g) = \text{dist}(f, g) + 1$. Por el Lema A.5, vale $\text{dist}(e, g) \leq \text{dist}(e, f) + \text{dist}(g, f)$. Luego

   $$\begin{aligned}\overline{\text{dist}}(e, g) &= \text{dist}(e, g) + 1 \\ &\leq (\text{dist}(e, f) + \text{dist}(f, g)) + 1 \\ &= (\text{dist}(e, f) + 1) + (\text{dist}(f, g) + 1) - 1 \\ &= \overline{\text{dist}}(e, f) + \overline{\text{dist}}(f, g) - 1\end{aligned}$$

   b) f no está alineado con ninguno de e o g

   Por el Lema A.4, es $\overline{\text{dist}}(e, f) = \text{dist}(e, f) + 2$ y $\overline{\text{dist}}(f, g) = \text{dist}(f, g) + 2$. Por el Lema A.2, es $\text{dist}(e, g) \leq \text{dist}(e, f) + \text{dist}(f, g) + 1$. Se tiene

   $$\begin{aligned}\overline{\text{dist}}(e, g) &= \text{dist}(e, g) + 1 \\ &\leq (\text{dist}(e, f) + \text{dist}(f, g) + 1) + 1 \\ &= (\text{dist}(e, f) + 2) + (\text{dist}(f, g) + 2) - 2 \\ &= \overline{\text{dist}}(e, f) + \overline{\text{dist}}(f, g) - 2\end{aligned}$$

2. e y g no están alineados

   Entonces $\overline{\text{dist}}(e, g) = \text{dist}(e, g) + 2$. En este caso, f está alineado con, a lo sumo, uno de e o g.

   a) f está alineado con exactamente uno de e o g

   Supongamos que está alineado con e. El otro caso es igual. Se tiene $\overline{\text{dist}}(e, f) = \text{dist}(e, f) + 1$ y $\overline{\text{dist}}(f, g) = \text{dist}(f, g) + 2$. Además, por el Lema A.5, es $\text{dist}(e, g) \leq \text{dist}(e, f) + \text{dist}(f, g)$. Luego

   $$\begin{aligned}\overline{\text{dist}}(e, g) &= \text{dist}(e, g) + 2 \\ &\leq (\text{dist}(e, f) + \text{dist}(f, g)) + 2 \\ &= (\text{dist}(e, f) + 1) + (\text{dist}(f, g) + 2) - 1 \\ &= \overline{\text{dist}}(e, f) + \overline{\text{dist}}(f, g) - 1\end{aligned}$$

   b) f no está alineado con ninguno de e o g

   Entonces, es $\overline{\text{dist}}(e, f) = \text{dist}(e, f) + 2$ y $\overline{\text{dist}}(f, g) = \text{dist}(f, g) + 2$, y

$$\begin{aligned}\overline{\text{dist}}(e, g) &= \text{dist}(e, g) + 2 \\ &\leq (\text{dist}(e, f) + \text{dist}(f, g) + 1) + 2 \\ &= (\text{dist}(e, f) + 2) + (\text{dist}(f, g) + 2) - 1 \\ &= \overline{\text{dist}}(e, f) + \overline{\text{dist}}(f, g) - 1\end{aligned}$$

□

Bibliografía

[1] Nicos Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical Report 388, Graduate School of Industrial Administration, Carnegie Mellon University, 1976.

[2] G. Clarke and J. W. Wright. Scheduling of vehicles from a central depot to a number of delivery points. *Operations Research*, 12(4):568–581, 1964.

[3] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[4] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[5] G. Dantzig, R. Fulkerson, and S. Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the Operations Research Society of America*, 2(4):393–410, 1954.

[6] Jack Edmonds. *Paths, Trees, and Flowers*, pages 361–379. Birkhäuser Boston, Boston, MA, 1987.

[7] S. Eilon, C. D. T. Watson-Gandy, N. Christofides, and R. de Neufville. Distribution management-mathematical modelling and practical analysis. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-4(6):589–589, Nov 1974.

[8] M. R. Garey, R. L. Graham, and D. S. Johnson. Some np-complete geometric problems. In *Proceedings of the Eighth Annual ACM Symposium on Theory of Computing*, STOC '76, pages 10–22, New York, NY, USA, 1976. ACM.

[9] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[10] B. L. Golden, T. L. Magnanti, and H. Q. Nguyen. Implementing vehicle routing algorithms. *Networks*, 7(2):113–148, 1977.

[11] Jonathan L. Gross and Jay Yellen. *Graph Theory and Its Applications, Second Edition (Discrete Mathematics and Its Applications)*. Chapman & Hall/CRC, 2005.

[12] J. A. Hoogeveen. Analysis of christofides'heuristic: Some paths are more difficult than cycles. *Oper. Res. Lett.*, 10(5):291–295, July 1991.

[13] Richard M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA, 1972.

[14] J. K. Lenstra and A. H. G. Rinnooy Kan. On general routing problems. *Networks*, 6(3):273–280, 1976.

[15] Amos Levin. Scheduling and fleet routing models for transportation systems. *Transportation Science*, 5(3):232–255, 1971.

[16] J. C. Liebman. Mathematical models for solid waste collection and disposal. *37 th national meeting of the Operations Research Society of America*, (2), 1970.

[17] D. H. Marks and R. Stricker. Routing for public service vehicles. *ASCE Journal of the Urban Planning and Development Division*, 97(2):195–178, 1974.

[18] A. D. O'Connor and C. A. De Wald. A sequential deletion algorithm for the design of optimal transportation networks. *37 th national meeting of the Operations Research Society of America*, 18(1).

[19] C. S. Orloff. A fundamental problem in vehicle routing. *Networks*, 4(1):35–64, 1974.

[20] Christos H. Papadimitriou. The euclidean travelling salesman problem is np-complete. *Theoretical Computer Science*, 4(3):237 – 244, 1977.

[21] P.J. Plauger, Meng Lee, David Musser, and Alexander A. Stepanov. *C++ Standard Template Library*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000.

[22] M. Solomon. Vehicle routing and scheduling with time window constraints: Models and algorithms. Technical Report 83–42, College of Business Admin., Northeastern University, 1983.

[23] Paolo Toth and Daniele Vigo, editors. *The Vehicle Routing Problem*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2001.

[24] N. Wilson and J. Sussman. Implementation of computer algorithms for the dial-a-bus system. *39th national meeting of the Operations Research Society of America*, 19(1).