Notas de la clase 7 – recorridos*

Francisco Soulignac

6 de mayo de 2019

Aclaración: este es un punteo de la clase para la materia AED3. Se distribuye como ayuda memoria de lo visto en clase y, en cierto sentido, es un reemplazo de las diapositivas que se distribuyen en otros cuatrimestres. Sin embargo, no son material de estudio y no suplanta ni las clases ni los libros. Peor aún, puede contener "herrorez" y podría faltar algún tema o discusión importante que haya surgido en clase. Finalmente, estas notas fueron escritas en un corto período de tiempo. En resumen: **estas notas no son para estudiar sino para saber qué hay que estudiar**.

Tiempo total: 0 minutos

1. Grafos dinámicos

- Hasta ahora trabajamos con (di)grafos que, desde el punto de vista computacional, son *estáticos* en el sentido de que no tenemos operaciones que permitan cambiar su estructura.
- Los (di)grafos *dinámicos*, en cambio, proveen al menos una de dichas operaciones.
- En esta sección diseñamos un tipo de grafo dinámico que permite la inserción de vértices y aristas y la remoción de aristas (no de vértices). No discutimos el caso particular de los digrafos, ya que es análogo.
- El costo de insertar un vértice a un grafo G es de $O(n)$ tiempo y de $O(1)$ tiempo amortizado en peor caso.
- El costo de insertar o sacar una arista vw a un grafo G es de $O(d(v) + d(w))$ tiempo y $O(1)$ tiempo amortizado en peor caso.
- La estructura de G está formada por listas de adyacencias. Específicamente:
  - ▷ Los vértices de G son números de 0 a $n - 1$.
  - ▷ Hay un vector `N` de n posiciones, donde `N`$[v]$ contiene el vecindario v con los vértices en un orden arbitrario.
  - ▷ Cada vecindario `N`$[v]$ se implementa con un vector que tiene $d(v)$ números.
- Además de las listas de adyacencias, se mantiene un *índice* `P`, que es un vector de n posiciones. A su vez, `P`$[v]$ es un vector de $d(v)$ posiciones tal que `P`$[v][p]$ denota la posición de v en la lista `N`$[w]$ donde $w \in N(v)$ es el valor en la posición p de `N`$[v]$ (en otras palabras, `P`$[v][p]$ es la posición de v en `N`$[$`N`$[v][p]]$).
- La operación de inserción de un vértice toma exclusivamente al grafo G y retorna un vértice, que en este caso es el número $n + 1$ correspondiente al vértice insertado. (En una versión abstracta, sería simplemente un valor de un tipo abstracto vertex, sin semántica, que funcionaría como handle —o puntero virtual— para acceder al vértice en $O(1)$ tiempo).
- La implementación consiste en agregar una nueva posición al final de `N` y `P`. El costo temporal de esta operación es $O(1)$ amortizado.

*Por falta de tiempo, estas notas son muy preliminares, no incluyen figuras y no fue tan exhaustivamente revisado como las clases previas.

- La operación de inserción de una arista toma el grafo G y dos vértices v y w y no retorna ningún valor. (En una versión abstracta, retornaría un handle para acceder a la arista en $O(1)$ tiempo.)

- La implementación consiste en insertar primero $d(w)$ al final de $\texttt{P}[v]$ y $d(v)$ al final de $\texttt{P}[w]$. Luego, se inserta v al final de $\texttt{N}[w]$ y w al final de $\texttt{N}[w]$. El costo temporal es $O(1)$ amortizado.

- Antes de discutir cómo borrar una arista, veamos una operación auxiliar $\texttt{swap}(v, p, q)$ que, dado un vértice v y dos posiciones p y q, intercambia en $\texttt{N}[v]$ los vértices que ocupan las posiciones p y q sin romper el invariante. (Notar que, en una versión abstracta del grafo, esta es una operación privada ya que exhibe la representación interna del grafo.) Para ello, primero se accede a $\texttt{N}[v]$ para determinar los vértices v_p y v_q en las posiciones p y q, respectivamente. Recordemos que $\texttt{P}[v][p]$ denota la posición x de v en la lista $\texttt{N}[v_p]$, mientras que $\texttt{P}[v_p][x]$ denota la posición de v_p en la lista $\texttt{N}[v]$, i.e., p. Luego, $\texttt{P}[v_p][\texttt{P}[v][p]]$ denota la posición de v_p en la lista $\texttt{N}[v]$, i.e., p. Luego de la operación de swap, este valor debe ser q y, análogamente, $\texttt{P}[v_q][\texttt{P}[v][p]]$ debe valer p. Esta actualización se realiza como segunda operación del algoritmo. Finalmente, se intercambian los valores de las posiciones $\texttt{N}[v][p]$ y $\texttt{N}[v][q]$ y de las posiciones $\texttt{P}[v][p]$ y $\texttt{P}[v][q]$. Claramente, el costo de `swap` es $O(1)$.

- La operación de eliminación de una arista toma el grafo G, un vértice v y una posición p y elimina la arista vw tal que w es el vértice en la posición p de $\texttt{N}[v]$. (En una versión abstracta, en lugar de la posición p se toma un handle a dicha arista.)

- Para la implementación, primero accedemos a $\texttt{N}[v][p]$ para computar w y luego accedemos a $\texttt{P}[v][p]$ para determinar la posición q que v tiene en $\texttt{N}[w]$. A continuación, aplicamos $\texttt{swap}(v, p, d(v) - 1)$ y $\texttt{swap}(w, q, d(w) - 1)$ a fin de lograr que w aparezca al final de $\texttt{N}[v]$ y v aparezca al final de $\texttt{N}[w]$. Posteriormente, eliminamos la última posición tanto de $\texttt{N}[v]$ y $\texttt{P}[v]$ como de $\texttt{N}[w]$ y $\texttt{P}[w]$. Esta operación toma $O(1)$ tiempo amortizado.

- Para fijar estos conceptos se muestra la implementación en C++ sin ningún tipo de abstracción. Queda como ejercicio la implementación de un buen tipo abstracto de datos para grafos que incluya los handles a las aristas.

Grafos con eliminación de aristas

```cpp
using vertex = int;

struct graph {
  vector<vector<vertex>> N;
  vector<vector<size_t>> P;

  graph() : N(1), P(1) {}

  vertex add_vertex() {
    N.push_back(vector<vertex>()); P.push_back(vector<size_t>());
    return static_cast<vertex>(N.size()) - 1;
  }

  //precondicion: v != w son vértices no adyacentes
  void add_edge(vertex v, vertex w) {
    P[v].push_back(N[w].size()); P[w].push_back(N[v].size());
    N[v].push_back(w); N[w].push_back(v);
  }

  void swap(vertex v, size_t p, size_t q) {
    using std::swap;
    auto vp = N[v][p], vq = N[v][q];
```

```
    P[vp][P[v][p]] = q; P[vq][P[v][q]] = p;
    swap(N[v][p], N[v][q]); swap(P[v][p], P[v][q]);
  }

  void remove_edge(vertex v, size_t p) {
    auto w = N[v][p]; auto q = P[v][p];
    swap(v, p, N[v].size()-1); swap(w, q, N[w].size()-1);
    N[v].pop_back(); N[w].pop_back();
    P[v].pop_back(); P[w].pop_back();
  }
};
```

2. Circuitos eulerianos

- Un *recorrido euleriano* en un grafo (dirigido o mixto) general es un camino que pasa exactamente una vez por cada arista.

- Un recorrido euleriano que empieza y termina en el mismo vértice se llama *circuito euleriano*.

- Si bien la definición es general, en este curso vamos a restringirnos a grafos y digrafos simples, dejando el caso general como ejercicio. El caso mixto es más complicado y requiere de definiciones adicionales.

- Aquellos (di)grafos que tienen algún recorrido euleriano se llaman (di)grafos *semi-eulerianos*, mientras que los que admiten algún circuito euleriano son (di)grafos *eulerianos*.[1]

- Un vértice v de un grafo G es *par* cuando $d(v)$ es par. Para unificar la terminología, decimos que un vértice v de un digrafo G es *par* cuando $d^{\text{out}}(v) = d^{\text{in}}(v)$.

- Un (di)grafo es *par* cuando todos sus vértices son pares.

- Un vértice v de un grafo G es *inicial* y *final* cuando o bien G es par o bien v no es par. Para unificar la terminología, decimos que un vértice v de un digrafo G es *inicial* cuando o bien G es par o bien $d^{\text{out}}(v) = d^{\text{in}}(v) + 1$, mientras que v es *final* cuando o bien G es par o bien $d^{\text{in}}(v) = d^{\text{out}}(v) + 1$.

- Un (di)grafo es *semi-par* cuando tiene a lo sumo dos vértices no pares v, uno de los cuales es inicial.

Observación 1. *Todo grafo par es semi-par. Más aún, si G es semi-par y no par, entonces contiene un vértice inicial v y otro final $w \neq v$ y ningún vértice distinto a v o w es inicial o final.*

- El siguiente procedimiento toma un (di)grafo semi-par G por referencia, junto con un vértice final v y retorna un recorrido euleriano de G, destruyendo G en el proceso.

- Si bien el algoritmo es muy simple, su demostración no lo es tanto; hay algoritmos más complejos con demostraciones más simples.

Recorrido euleriano de un (di)grafo G

```
getEulerianTour(G, v):
  Si d^(in)(v) = 0, entonces retornar v y terminar.
  Remover wv de G para algún w ∈ N^(in)(v).
```

[1] Para nosotros, los grafos eulerianos son semi-eulerianos; esta notación es contraria a lo comúnmente usado, pero la usamos porque simplifica algunas afirmaciones. Tener cuidado con esto al consultar la bibliografía.

```
4   Retornar getEulerianTour(G, w) + getEulerianTour(G, v)
```

Lema 1. *Sea v vértice final de un (di)grafo semi-par G, V la componente conexa de G que contiene a v, C el output de `getEulerianTour(G, v)` y H el grafo resultante de la destrucción de G. Entonces:*

1. *$H = G - E(G[V])$, i.e., H se obtiene de remover todas las aristas entre vértices de V.*
2. *C es un recorrido euleriano de $G[V]$ que termina en v.*
3. *Si $G[V]$ es par, entonces C es un circuito.*
4. *Si $G[V]$ es impar, entonces C empieza en un vértice inicial de G (distinto a v).*

Demostración. La demostración es exactamente la misma para el caso de grafos y digrafos (e incluso debería funciona también para grafos y digrafos generales ajustando bien la terminología). La razón de esto se encuentra en las definiciones del principio que abstraen las pequeñas diferencias. Sin embargo, puede ser compleja de entender por qué funciona en ambos casos, razón por la cual se aconseja leer la demostración dos veces, una pensando que G es un grafo y la otra considerando que G es un digrafo.

La demostración es por inducción en la cantidad de aristas m de $G[V]$.[2] El caso base $m = 0$ es trivial, ya que $G[V]$ es par, $H = G = G - E(G[V])$ y $C = v$ es un circuito euleriano de $G[V]$ que empieza y termina en el vértice inicial y final v. Para el paso inductivo $m > 0$, conviene establecer cierta notación en función de las modificaciones que realiza el algoritmo. Denotemos $v_0 = v_2 = v$, $v_1 = w$, $G_0 = G$ al (di)grafo input, $G_1 = G - wv$ al (di)grafo de la primer invocación recursiva y G_2 al (di)grafo de la segunda invocación recursiva. Más aún, para $i \in \{0, 1, 2\}$, llamemos V_i a la componente conexa de G_i que contiene a v_i y C_i al output de `getEulerianTour(`G_i`, `v_i`)`. Consideremos los siguientes casos.

1. G_0 no es par. En este caso, teniendo en cuenta que G_0 es semi-par, obtenemos que v es el único vértice final no par y, por lo tanto, $G_0[V_0]$ tiene un único vértice inicial $z \neq v$. Si $z = w$, entonces G_1 es un (di)grafo par y, consecuentemente, w es un vértice inicial de G_1. Caso contrario, w es par en G_0 y no par en G_1, con lo cual G_1 es nuevamente semi-par y z y w son sus únicos vértices inicial y final, respectivamente. Esto implica que z y w pertenecen a la misma componente V_1 de G_1. En ambos casos podemos aplicar la hipótesis inductiva sobre G_1 y w porque w es un final del (di)grafo semi-par G_1 que tienen $m-1$ aristas. Luego: 1. $G_2 = G_1 - E(G_1[V_1])$; 2. C_1 es un recorrido euleriano de $G_1[V_1]$ que termina en w y 3. C_1 es un circuito (y $z = w$) o empieza en el vértice inicial $z \neq w$. Luego, tenemos dos posibilidades.

   a) $v \in V_1$. En este caso v no tiene vecinos en G_2 y, por lo tanto, $C_2 = v$. Ahora, dado que tanto v como w pertenecen a V_1, podemos observar que $V_1 = V$ y, por lo tanto $H = G_2 = G_1[V_1] - E(G_1[V_1]) = G_0 - E(G_0[V_0])$. Más aún, esto implica que C_1 es un circuito que termina en w y pasa exactamente una vez por cada arista de $G_0[V_1]$ excepto wv. Luego, $C_1 + C_2$ es un recorrido euleriano de $G_0[V_0]$ que termina en v y cuyo primer vértice z es inicial y distinto a v.

   b) $v \notin V_1$. En este caso, v y w están en componentes distintas de $G_1 = G - wv$. Notemos que las aristas de $G_0[V_0]$ se pueden particionar en tres: vw por un lado, las aristas de $G_1[V_1]$ por otro, y finalmente las aristas de $G_1[V_2] = G_2[V_2]$. Ahora, como la única componente de G_1 que puede no ser par es $G_1[V_1]$, obtenemos que $G_1[V_2] = G_2[V_2]$ es un grafo par. Esto significa que v es final en $G_2[V_2]$ y, por hipótesis inductiva: 1. $H = G_2 - E(G_2[V_2]) = (G_1 - E(G_1[V_1])) - E(G_1[V_2]) = G_0 - E(G[V_0])$ y 2. C_2 es un circuito euleriano de G_2 que empieza y termina en v. Luego, $C_1 + C_2$ es un recorrido euleriano de $G[V_0]$ que empieza en el vértice inicial $z \neq v$ y termina en el vértice final v.

2. G_0 es par. En este caso tanto v como w son pares en G_0 y, por lo tanto, G_1 es un di(grafo) semi-par que no es par y tiene a v como inicial y a w como final. Esto es posible únicamente cuando v

[2] Notar que m no es estrictamente la cantidad de aristas de G en este caso, pero lo usamos porque tiene el mismo significado.

y w pertenecen a la misma componente V_1 de G_1. Luego, por hipótesis inductiva, obtenemos que 1. $H = G_2 = G_1 - E(G_1[V_1]) = G_0 - E(G_0[V_0])$ y 2. C_1 es un recorrido euleriano de $G_1[V_1]$ que empieza en v y termina en w. Pero entonces, como v no tiene vecinos en G_2, obtenemos que $C_2 = v$ y, como en el caso 1 arriba, $C_1 + C_2$ es un circuito euleriano de G que empieza y termina en v.

□

- Digamos que G es *virtualmente conexo* si todas las aristas de G pertenecen a la misma componente conexa de G.

- Equivalentemente, G es virtualmente conexo si $V = V(G)$ o $G - V$ es conexo, donde V es el conjunto de vértices aislados de G.

Teorema 1 (Euler [5] y Hierholzer [6])**.** *Un (di)grafo general es (semi-)euleriano si y sólo si es virtualmente conexo y (semi-)par.*

Demostración. Supongamos primero que G admite un recorrido euleriano $C = v_0, \ldots, v_m$. Como C pasa exactamente una vez por cada arista, obtenemos que G es virtualmente conexo y que la cantidad de aristas incidentes en v es la cantidad de aristas de C incidentes en v. Luego, si definimos $x_i = 1$ cuando $v = v_i$ y $x_i = 0$ en caso contrario, para $0 \leq i \leq m$, obtenemos que $d(v) = 2\sum_1^{m-1} x_i + x_0 + x_m$, mientras que si G es un digrafo, entonces $d^{\text{out}}(v) = \sum_0^{m-1} x_i$ y $d^{\text{in}}(v) = \sum_{i=1}^{m} x_i$. Esto es así porque para cada $x_i = 1$, el vértice $v = v_i$ tiene dos aristas $v_{i-1}v$ y vv_{i+1} salvo que $i = 0$ o $i = m$. En consecuencia, G es semi-par porque todos los vértices salvo x_0 y x_m son pares. Más aún, si C es un circuito, entonces G es par porque $v_0 = v_m$.

Si bien vale para (di)grafos generales, lo demostramos sólo para (di)grafos simples y dejamos el caso general como ejercicio. La vuelta es trivial si G no tiene aristas, ya que cualquier vértice es un ciclo que pasa por todas las aristas de G. Caso contrario, G tiene al menos un vértice final v con $d(v) > 0$. Luego, como todas las aristas de G pertenecen a la componente V que contiene a v, el Lema 1 implica que `getEulerianTour(`G`, `v`)` retorna un recorrido euleriano de G. Más aún, este recorrido es un circuito cuando G es par. □

- Reconocimiento de (di)grafos (semi-)eulerianos

  Input: un (di)grafo G conexo.

  Output: si G es semi-euleriano, entonces un recorrido euleriano; caso contrario, un vértice que no sea par ni inicio ni fin, o tres vertices que no sean pares. Notar que esto alcanza para determinar si G es euleriano, ya que en caso afirmativo el output es necesariamente un circuito, mientras que si G es semi-euleriano pero no euleriano, el primer vértice no par del recorrido sirve como certificado negativo.

- No vamos a discutir cómo determinar si G es semi-euleriano y, en tal caso, encontrar un vértice inicial v.

- Sí discutimos la complejidad de encontrar un recorrido euleriano cuando G es semi-euleriano.

- Notemos que en cada invocación recursiva de `getEulerianTour(`G`, `v`)` que no sea un caso base se elimina una arista de G.

- En consecuencia, se realizan $O(m)$ llamadas recursivas para computar `getEulerianTour(`G`, `v`)`.

- La eliminación de cada arista cuesta $O(1)$ tiempo amortizado cuando el (di)grafo G se implementa usando listas de adyacencias con índices como en la sección anterior.

- Finalmente, la concatenación de los resultados de ambas invocaciones es innecesaria si se utiliza una secuencia para ir guardando los valores en cada invocación, a un costo temporal de $O(1)$ tiempo.

- En conclusión, la invocación `getEulerianTour(`G`, `v`)` cuesta $O(n+m)$ tiempo cuando G es un grafo con la siguiente implementación en C++.

Recorrido euleriano de un (di)grafo G en C++

```
void getEulerianTour(graph& G, vertex v, vector<int>* res):
  if(G.N[v].empty()) {res->push_back(v); return;}
  vertex w = G.N[v][0];
  G.remove_edge(v, 0);
  getEulerianTour(G, w, res);
  getEulerianTour(G, v, res);
```

Teorema 2. *El problema de reconocimiento de grafos eulerianos se puede resolver en $O(n+m)$ tiempo.*

3. Problema del cartero chino o de inspección de rutas

- Digamos que un circuito C de G es una *inspección* cuando pasa por todas las aristas de G al menos una vez.
- Para una función de costos $c \geq 0$, decimos que C es una *inspección mínima* de (G, c) si $c_+(C)$ es el mínimo de entre todas las inspecciones de G.
- Problema del cartero chino o de inspección de rutas.

  Input: un (di)grafo (fuertemente) conexo G y una función de costos $c \geq 0$.

  Output: una inspección mínima de (G, c).
- Notemos que si G es euleriano, entonces cualquier circuito euleriano es una inspección mínima.
- Por simplicidad, en esta sección consideramos sólo el caso no dirigido; el caso dirigido es similar y queda como ejercicio.
- Vamos a reformular el problema de inspección en un problema equivalente que consiste en encontrar un bosque generador mínimo de G particular.
- Decimos que un bosque F es un *odd-completion* de G cuando F es un bosque generador de G cuyos vértices pares se corresponden con los vértices pares de G.
- El siguiente par de teoremas demuestra que alcanza con encontrar un odd-completion de peso mínimo.

Teorema 3. *Sea G un grafo conexo y $c \geq 0$ una función de costos. Si C una inspección mínima de (G, c) que contiene la menor cantidad de aristas, entonces sus aristas se pueden particionar en $E(G)$ y $E(F)$ donde F es un odd-completion de G. Notar que $c_+(C) = c_+(G) + c_+(F)$.*

Demostración. Por definición, C es un recorrido euleriano del multigrafo $H = (V(G), E(C))$ (i.e., formado por las aristas de C). Por Teorema 1, esto implica que H es un multigrafo par. Si alguna arista vw de H estuviera repetida al menos tres veces, entonces podemos sacar dos repeticiones de vw obteniendo un multigrafo par con menos aristas cuyo circuitos eulerianos C' son inspecciones de G. Pero esto contradice la minimalidad de C porque el peso de C' es $c_+(C') = c_+(C) - 2c(vw) \leq c_+(C)$. Análogamente, si H tuviera un ciclo R formado exclusivamente por aristas repetidas, entonces el multigrafo que se obtiene de sacar de H una copia de cada arista en R sería par y sus circuitos eulerianos inspeccionarían G con un costo menor o igual a C, lo cual también es imposible. En consecuencia, el multigrafo F que se obtiene de H sacando una copia de cada arista de G es un bosque simple. Más aún, como $d_H(v) = d_G(v) + d_F(v)$ es par para todo $v \in V(G)$, entonces v es par en F si y sólo si v es par en G. □

Teorema 4. *Sea G un grafo conexo y $c \geq 0$ una función de costos. Si F es un odd-completion de G, entonces el múltigrafo H que se obtiene de repetir las aristas de F en G es euleriano. Luego, cualquier circuito euleriano C de H es una inspección de G de costo $c_+(C) = c_+(G) + c_+(F)$.*

Demostración. En rigor, esto es una observación, ya que H es conexo por ser G conexo y $d_H(v) = d_G(v) + d_F(v)$ es par para todo $v \in V(G)$. Luego, H es euleriano por Teorema 1. □

- En función de los teoremas anteriores, para resolver el problema del cartero chino, alcanza con encontrar el odd-completion F de peso mínimo de G. Luego, tomando cualquier circuito euleriano de $G + F$ se obtiene la inspección mínima.

- Vamos a modelar el problema de encontrar F con otro problema equivalente.

- Digamos que una familia de caminos $\mathcal{P}$ de un grafo G es un *odd-join* cuando:

  - ▷ todo camino de $\mathcal{P}$ une dos vértices no pares de G, y
  - ▷ todo vértice no par de G es extremo de un único camino de $\mathcal{P}$

- Definimos el peso de un odd-join $\mathcal{P}$ como $c_+(\mathcal{P}) = \sum_{P \in \mathcal{P}} c_+(P)$.

- Los odd-joins y los odd-completions son dos caras de la misma moneda, de acuerdo al siguiente lema.

Lema 2. *Si F es un odd-completion de un grafo conexo G, entonces $E(F)$ se puede particionar en un odd-join de G. Recíprocamente, existe un odd-join $\mathcal{P}$ de peso mínimo tal que $(V(G), \bigcup \mathcal{P})$ es un odd-completion de G.*

Demostración. Supongamos primero que F es un odd-completion de G y veamos por inducción en m que $E(F)$ se puede particionar en un odd-join. El caso base $m = 0$ es trivial. Para el paso inductivo, consideremos un camino P_1 entre dos hojas v y w de F. Notemos que $G - E(P_1)$ es un grafo cuyos vertices pares son los mismos que en G, salvo por v y w. Luego, $F - E(P_1)$ es un odd-completion de $G - E(P_1)$. Por hipótesis inductiva, $F - P_1$ se puede particionar en un odd-join $P_2, \ldots, P_k$ de $G - E(P_1)$. Ciertamente, P_1 y P_i son disjuntos en aristas para todo $1 < i$. Más aún, cada P_i une dos vértices no pares de $G - E(P_1)$ y por lo tanto dos vértices no pares de G, mientras que P_1 une v y w que son impares en G. Finalmente, todo vértice no par de $G - E(P_1)$ es el extremo de un camino de $\mathcal{P} = \{P_1, \ldots, P_k\}$ (porque no es extremo de P_1), mientras que v y w son extremos sólo de P_1. En consecuencia, $\mathcal{P}$ es un odd-join de G.

Para la vuelta, tomemos el odd-join $\mathcal{P}$ de G de peso mínimo cuya unión tenga la menor cantidad posible de aristas y definamos $F = (V(G), \bigcup \mathcal{P})$. Notar que F es potencialmente un multigrafo, porque los caminos de $\mathcal{P}$ no son necesariamente disjuntos en aristas. Si F tiene un ciclo (que puede ser por dos aristas repetidas) es porque existen dos caminos $P = v_1, \ldots, v_k$ y $Q = w_1, \ldots, v_h$ en $\mathcal{P}$ tales que $v_i = w_j$ y $v_x = w_y$ para $i < j$ y $x < y$. Supongamos que i y x son mínimos mientras que j e y son máximos. Entonces, $P' = v_1, \ldots, v_i, w_{j-1}, \ldots, w_1$ y $Q' = v_k, \ldots, v_x, w_{y+1}, \ldots, w_h$ son dos caminos de G que unen a los cuatro vertices no pares v_1, v_k, w_1, w_h que no están unidos en $\mathcal{P} \setminus \{P, Q\}$. En consecuencia, $\mathcal{P}' = \mathcal{P} \setminus \{P, Q\} \cup \{P', Q'\}$ es un odd-join de peso menor o igual a $\mathcal{P}$ que tiene menos aristas que $\mathcal{P}$. Pero esto es imposible. □

- En función del ultimo teorema, en lugar de un odd-completion, podemos concentrarnos en buscar un odd-join $\mathcal{P}$ de peso mínimo.

- El odd-join $\mathcal{P}$ induce en cierta forma un odd-completion F que podemos usar para determinar la inspección como antes.

- Sin embargo, también podemos notar que para cualquier odd-join $\mathcal{P}$ (tenga o no la menor cantidad de aristas y sea o no mínimo), el multigrafo $H = G + \bigcup \mathcal{P}$ es euleriano y, por el teorema anterior, cualquiera de sus circuitos eulerianos es una inspección mínima.

- Para encontrar el odd-join de peso mínimo, definamos un grafo completo K que tiene un vértice v por cada vértice no par de G.

- El peso de la arista vw de K es el costo de un camino mínimo de v a w.

- Por definición, K tiene $2k$ vértices para algún k.

- Luego, se pueden particionar los vértices de K utilizando k aristas $e_1, \ldots, e_k$ disjuntas. A dichos conjuntos de aristas se los llaman *matching máximos*.

- Cada una de estas aristas e_i se corresponde con un camino P_i de G entre dos vértices no pares. Por lo tanto, $\mathcal{P} = P_1, \ldots, P_k$ es un odd-join de G.

- En otras palabras, alcanza con encontrar el matching máximo de peso mínimo de G para determinar la inspección mínima. Este problema es polinomial y se puede resolver en $O(k^3 \log k)$ para el grafo K.

- El algoritmo completo se resume a continuación.

Algoritmo para el problema del cartero chino

```
CPP(G, c):
  Sean v_1,...,v_2k los vértices no pares de G.
  Sea K el grafo completo con vértices v_1,...,v_2k
  Sea c_K la función de costos donde c_K(vw) es el costo del camino mínimo de v a w en G
  Determinar un matching máximo de costo mínimo e_1,...,e_k de K
  Crear el multigrafo H agregando a G las aristas del camino P_i representado por e_i para
  ↪ cada 1 ≤ i ≤ k
  Retornar un circuito euleriano de H
```

- El costo del algoritmo cuando G tiene k vértices no pares, aplicando el método de Johnson para encontrar los caminos mínimos, es de $O(nm + k(n + k^2) \log n)$.

- El algoritmo para digrafos es similar, aunque en lugar de tener que encontrar un matching en un grafo completo, se busca en uno bipartito.

- El problema de inspección de rutas se puede generalizar a grafos mixtos. En este caso, no se conocen algoritmos polinomiales para resolver el problema y se cree que dichos algoritmos no existen.

4. Circuitos hamiltonianos

- Un ciclo (resp. camino) es *hamiltoniano* cuando pasa exactamente un vez por cada vértice.

- Un grafo es *hamiltoniano* cuando tiene algún ciclo hamiltoniano.

- Problema de ciclo (resp. camino) hamiltoniano.

  Input: un (di)grafo G.

  Output: verdadero si y sólo si G tiene un ciclo (resp. camino) hamiltoniano. En caso afirmativo, se podría exhibir un ciclo hamiltoniano de G.

- No se conocen algoritmos polinomiales para el problema de ciclo (resp. camino) hamiltoniano.

- Sí se conocen algoritmos polinomiales para algunos grafos particulares, como veremos más adelante.

- Tampoco se conocen buenos certificados negativos, i.e., estructuras verificables en tiempo polinomial tales que, dado cualquier grafo G no hamiltoniano, sirvan para demostrar que G no es hamiltoniano.[3]

[3] Comparar con el certificado positivo que demuestra que G sí es hamiltoniano: el ciclo. El mismo es verificable en tiempo polinomial.

4.1. Certificado negativo

- Vamos a empezar mostrando una condición necesaria para que G sea hamiltoniano.

- En términos algorítmicos, esta condición se puede utilizar para mostrar que un grafo no es hamiltoniano y se puede verificar en tiempo polinomial. Desafortunadamente, hay grafos no hamiltonianos que sí satisfacen la condición y, por lo tanto, no sirve como certificado general. (Que, como dijimos antes, no se conoce.)

- Por ejemplo, podemos ver que G es hamiltoniano sólo si G es conexo, con lo cual se puede dar una partición de $V(G)$ en dos componentes conexas para demostrar que un grafo (no conexo) G no es hamiltoniano.

- De la misma forma, $G - v$ no puede tener dos componentes conexas; caso contrario, x, v, y, x no pueden aparecer en este orden en un ciclo para x e y en componentes distintas de $G - v$, ya que no hay camino de y a x. Luego, podemos certificar que G no es hamiltoniano mostrando algún punto de corte v, que se puede verificar correcto en tiempo polinomial.

- Generalizando, $G - W$ no puede tener más de $|W|$ componentes conexas (ver abajo). Luego, podemos certificar que G no es hamiltoniano mostrando este conjunto de corte, que se puede verificar en tiempo polinomial.

Teorema 5. *Si G es un grafo hamiltoniano y $W \subseteq V(G)$, entonces $G \setminus W$ tiene a lo sumo $|W|$ componentes conexas.*

Demostración. Si C un circuito hamiltoniano de G, entonces $C \setminus W$ consiste de a lo sumo $|W|$ caminos que pasan por todos los vértices de $G \setminus W$. Luego, $G \setminus W$ no tiene más de $|W|$ componentes. □

- La vuelta no vale: existen grafos no hamiltonianos tales que para todo $W \subseteq V(G)$ ocurre que $G \setminus W$ tiene menos de $|W|$ componentes.

  ▷ Como ejemplo, consideremos al grafo G que tienen una clique v_1, v_2, v_3, v_4 y tres vértices w_1, w_2, w_3 tal que w_i es adyacente a w_1 y w_{i+1}. Ciertamente, este grafo no tiene un circuito hamiltoniano, ya que el mismo debería contener las dos aristas de w_i para cada i, pero v_1 no puede tener tres vecinos en el ciclo. Ahora, para desconectar a w_i de la clique $v_1, \ldots, v_4$ sí o sí debemos eliminar a v_1 y a v_{i+1}. Esto implica que para tener k componentes tenemos que remover al menos k vértices.

- Dado un conjunto W, podemos verificar que $G \setminus W$ tiene más de $|W|$ componentes en tiempo $O(n + m)$ (ejercicio!).

4.2. Certificado positivo

- Vamos ahora a ver un algoritmo general que permite "simplificar" la busqueda de un ciclo hamiltoniano.

- Dado un grafo G, se define su *clausura transitiva* (de suma de grados) como el grafo H que se obtiene de agregar iterativamente aristas de la forma vw tal que $d(v) + d(w) \geq n$ para $vw \notin E(G)$ hasta que no queden vértices no adyacentes de esta forma.

- Es fácil ver que cualquier ciclo hamiltoniano C de G es un ciclo hamiltoniano de H; la recíproca es obviamente falsa.

- En consecuencia, pareciera que encontrar un ciclo hamiltoniano en H puede ser llegar a ser más simple.

- Aunque no vale la recíproca, si vale una versión más débil: si H es hamiltoniano, entonces G también lo es.

- Para demostrar esto vamos a diseñar un algoritmo que transforme el ciclo hamiltoniano de H en uno de G.

- La demostración se basa en los *movimientos* 2-*opt* que también vamos a usar más adelante.

- Sea $C = v_1, \ldots, v_n$ una permutación cíclica de un conjunto V (i.e., v_1 sigue a v_n). Definimos 2-opt(C, i, j) como la permutación cíclica $C' = v_i, v_j, v_{j-1}, \ldots, v_{i+1}, v_{j+1}, v_{j+2} \ldots, v_i$.

Observación 2. *Si $C = v_1, \ldots, v_n$ es un ciclo hamiltoniano de un grafo G y v_iv_j y $v_{i+1}v_{j+1}$ son aristas de G para algún $i > 1$, entonces 2-opt(C, i, j) es un ciclo hamiltoniano de G.*

Teorema 6 (Bondy y Chvátal [3])**.** *Un grafo G es hamiltoniano si y sólo si su clausura transitiva H es hamiltoniano.*

Demostración. La ida es trivial, ya que cualquier ciclo hamiltoniano de G es un ciclo hamiltoniano de H porque G es un subgrafo generador de H.

Para la vuelta, supongamos que $H = G_k$ donde $G_0 = G$ y G_{i+1} se consigue a partir de G_i agregando una arista x_iy_i entre dos vértices no adyacentes con $d_i(x_i) + d_i(y_i) \geq n$, donde d_i denota el grado de un vértice en G_i. Dado un ciclo hamiltoniano C_{i+1} de G_{i+1}, vamos a mostrar cómo conseguir un ciclo hamiltoniano C_i de G_i. Luego, empezando con un ciclo hamiltoniano de H podemos obtener un ciclo hamiltoniano de G en k pasos. Claramente, podemos tomar $C_i = C_{i+1}$ si C_{i+1} no pasa por x_iy_i. Supongamos, pues, que $C_{i+1} = v_1, \ldots, v_n$ donde $v_1 = x_i$ y $v_2 = y_i$. Sea I_j el conjunto de índices tales que v_j es adyacente en G_i a todos los vértices de I_j para $j = 1, 2$. Como $|I_1| + |I_2| \geq n$ e $I_1 \cup I_2 \subseteq \{3, \ldots, n\}$, entonces existe j tal que $j \in I_1$ y $j + 1 \in J_2$ (¿por qué? ejercicio). Luego, por Observación 2, $C_i = 2\text{-opt}(C_{i+1}, 1, j)$ es un ciclo hamiltoniano de G_{i+1} que no pasa por x_iy_i y, por lo tanto, también es un ciclo hamiltoniano de G_i. □

- El teorema de Bondy y Chvátal permite diseñar un algoritmo para el problema de ciclo hamiltoniano que requiere que exista un algoritmo para resolver el problema de ciclo hamiltoniano sobre la clausura transitiva.

- Por ejemplo, si la clausura transitiva es un grafo completo, entonces hay algoritmos lineales para calcular un ciclo hamiltoniano.

Algoritmo para computar ciclo hamiltoniano a partir de clausura

```
1  getHamiltonianCycle(G):
2    //Computar la clausura transitiva de G y mantener el orden
3    Poner S = ∅
4    Mientras G tenga dos vértices v y w no adyacentes con d_G(v) + d_G(w) ≥ n:
5      Poner G = G + vw y S = S + vw.
6    Determinar un ciclo hamiltoniano C de G.  Si C no existe, retornar error.
7    Mientras S no esté vacío:
8      Sacar de S y G la última arista vw
9      Si vw es la arista C[i]C[i + 1] de C, entonces:
10       Buscar el mínimo j tal que C[i]C[i + j] y C[i + 1]C[i + j + 1] sean aristas de G.
11       Poner C = 2-opt(C, i, i + j)
```

- Para la implementación del algoritmo, almacenamos el grafo G en una matriz de adyacencia, d en un vector, y S y C en vectores.

- Para el primer loop, primero creamos un conjunto auxiliar X con todos los pares (v, w) de vértices no adyacentes tales que $d(v) + d(w)$. Este paso toma tiempo $O(n^2)$.

- Luego, el primer loop procesa y elimina cada par (v, w) de X en algún orden.

- La inserción de vw a S y G requiere $O(1)$ tiempo.

- Una vez insertado vw, recorremos cada vértice z de G a fin de agregar a X los pares (v, z) con $vz \notin E(G)$ y $d(v) + d(z) = n$ y los pares (w, z) con $wz \notin E(G)$ y $d(w) + d(z) = n$.

- En resumen, como cada par (v, w) se procesa en $O(n)$ tiempo y hay a lo sumo $O(n^2)$ pares, el primer loop requiere tiempo $O(n^3)$.

- Para la búsqueda del circuito hamiltoniano en la clausura transitiva aplicamos un algoritmo que toma tiempo $O(\alpha(n))$.

- En el segundo loop se procesan $O(n^2)$ aristas a lo sumo.

- La remoción de vw de S y G requiere $O(1)$ tiempo.

- Luego, revisar si vw es una arista de C y cuál es el mínimo j que permite un 2-opt se realiza en $O(n)$ tiempo.

- Finalmente, el 2-opt de C no consume más de $O(n)$ tiempo, ya que consiste en computar el reverso de una porción de C.

- En resumen, el segundo loop también requiere $O(n^3)$ tiempo.

- El costo del algoritmo es $O(n^3 + \alpha(n))$ donde $\alpha(n)$ es el costo del algoritmo que encuentra un ciclo hamiltoniano en la clausura transitiva de G.

- Este algoritmo se puede mejorar considerablemente cuando podemos encontrar el ciclo hamiltoniano agregando un subconjunto de las aristas de la clausura transitiva de G.

- Por ejemplo, si pudieramos encontrar un ciclo hamiltoniano luego de aplicar n iteraciones del primer loop, el algoritmo requeriría $O(n^2)$ tiempo.

- Digamos que un grafo es *(suficientemente) denso* cuando $d(v) + d(w) \geq n$ para todo par de vértice no adyacentes v y w.

Observación 3. *Si G es denso, entonces $m = \Omega(n^2)$*

Demostración. Ejercicio. □

- Los siguientes corolarios del teorema de Bondy y Chvátal se conocen con los nombres de Teorema de Ore y Teorema de Dirac.

Corolario 1 (Ore [7])**.** *Si G es un grafo denso con $n \geq 3$ vertices, entonces G es hamiltoniano.*

Demostración. Porque la clausura transitiva de G contiene es completo y, por lo tanto, hamiltoniano. □

Corolario 2 (Dirac [4])**.** *Si G es un grafo con $n \geq 3$ tal que $2d(v) \geq n$ para todo $v \in V(G)$, entonces G es hamiltoniano.*

Demostración. Porque $d(v) + d(w) \geq n$ para todo $v, w \in V(G)$. □

- Veamos como mejorar el algoritmo de la clausura para el caso particular en que G es un grafo denso.

- Como la clausura transitiva H de G es un grafo completo, sabemos que $C = v_1, \ldots, v_n$ es un circuito hamiltoniano de H, cualquiera sea el orden $v_1, \ldots, v_n$.

- Por otra parte, sabemos que si $v_i v_{i+1} \notin E(G)$, entonces $d(v_i) + d(v_{i+1}) \geq n$, con lo cual podemos restringir el primer loop al agregado de las aristas $v_i v_{i+1}$ de C que no están en G, para obtener grafo supergrafo G' de G que es subgrafo de H y que contiene el ciclo hamiltoniano C. Este agregado se puede hacer en cualquier orden, e.g., incremental.

- Luego, como S tiene $O(n)$ aristas después del primer loop, el segundo loop cuesta $O(n^2)$ tiempo.

- Obviamente, esta descripción muestra cómo mejorar un algoritmo ya existente; sin embargo, no es estrictamente necesario computar el primer loop ya que las aristas a agregar están implícitamente codificadas en C.
- A continuación se muestra el código simplificado, que es una adaptación del algoritmo de Palmer [8].
- Como antes, G se implementa sobre una matriz de adyacencia.
- Notar que el algoritmo es lineal porque $m = \Omega(n^2)$.

Adaptación del algoritmo de Palmer [8]

```
getHamiltonianCycle(G):
  //Precondición: G es denso
  Poner C como cualquier permutación de V(G).
  Mientras C contenga una arista C[i]C[i+1] que no pertenece a G:
    Buscar el mínimo j tal que C[i]C[i+j] y C[i+1]C[i+j+1] sean aristas de G.
    Poner C = 2-opt(C, i, i+j)
```

5. Problema del viajante de comercio

- Problema (a)simétrico de viajante de comercio ((A)TSP).

  Input: un (di)grafo completo G y una función de costos c.

  Output: un ciclo hamiltoniano C de G de costo $c_+(C)$ mínimo.

- El problema es simétrico cuando G es un grafo y asimétrico cuando G es un digrafo; si no aclaramos, es simétrico.
- Notar que no hay perdida de generalidad en suponer que el (di)grafo es completo: si queremos resolver TSP para un grafo G no completo, entonces podemos usar el grafo H que se obtiene de G agregando cada arista faltante vw con peso $c(vw) > nM = n\sum_{xy \in E(G)} |c(xy)|$. Notar que G es hamiltoniano si y sólo si G tiene un ciclo hamiltoniano de peso a lo sumo M y esto ocurre si y sólo si el ciclo hamiltoniano de peso mínimo de H pesa a lo sumo M.
- Esta misma idea se puede usar en otros problemas vistos previamente, como camino mínimo o árbol generador mínimo. La razón para no hacerlo es porque el grafo H es denso, lo cual puede empeorar la complejidad temporal con respecto al uso de G.
- Las razones para sí hacerlo en este caso son dos:

  1. Primero, porque no se conocen algoritmos polinomiales para TSP; caso contrario, podríamos usarlos para resolver el problema de ciclo hamiltoniano sobre G usando el algoritmo anterior.
  2. Segundo, y más importante, porque de esta forma sabemos que cualquier permutación de los vértices de H es un ciclo hamiltoniano. De esta forma, podemos obtener ciclos fácilmente, lo que va a ser muy útil a la hora de diseñar buenos algoritmos para el problema. Visto desde el punto de vista de G esto, en cierta forma, es generalizar el problema aceptando tener algunos ciclos "infactibles" en G. Notar que esta es la misma idea que usamos previamente para computar ciclos hamiltonianos de grafos densos: primero generamos un grafo completo y luego sacamos las aristas "malas".

- Si bien faltan algunas clases para ver el tema de NP-completitud, conviene hacer una breve introducción aca: se define con Pal conjunto de todos los problemas que se pueden resolver en tiempo polinomial (e.g., camino minimo, AGM, etc.)

- Esta clase Pestá incluida en una clase llamada NPy la pregunta abierta más importante en Ciencias de la Computación es determinar si $\mathsf{P} = \mathsf{NP}$.

- Una de las respuestas condicionales es la siguiente: $\mathsf{P} = \mathsf{NP}$ si y sólo si existe un algoritmo polinomial para TSP.

- Esto significa que mientras nadie demuestre que $\mathsf{P} = \mathsf{NP}$ no se va a conocer un algoritmo polinomial para TSP y, recíprocamente, si se demostrara que $\mathsf{P} \neq \mathsf{NP}$, entonces dicho algoritmo no puede existir.

- Esto es desafortunado, dado que TSP tiene muchas aplicaciones prácticas que se **deben** resolver en la vida real.

- En general, el problema de la vida real consiste en encontrar un ciclo hamiltoniano del *menor peso posible*.

- Desde un punto de vista teórico, la única solución admisible es un ciclo hamiltoniano C^* de peso $c_+(C^*)$ mínimo.

- Desde el punto práctico, se requiere encontrar sí o sí un ciclo hamiltoniano C, donde el peso de C representa alguna noción de costo.

- Por ejemplo, si tenemos que sí o sí distribuir productos a un conjunto de clientes, entonces sí o sí necesitamos un camino que pase por todos los clientes. En general, el costo de cada arista representa el costo de ir de un cliente a otro; si bien queremos minimizar este costo, igual necesitamos distribuir los productos de alguna forma. Queremos que esa forma sea la mejor que podamos conseguir.

- En resumen, dado que se tiene que encontrar un ciclo hamiltoniano sí o sí y no es posible esperar eternamente por una solución de costo mínimo, el objetivo es encontrar un ciclo hamiltoniano C que sea lo suficientemente "bueno".

- En este punto es donde vuelve a cobrar sentido que el grafo G sea completo, ya que podemos garantizar que vamos a encontrar eficientemente al menos un ciclo hamiltoniano de G.

- Pero, no queremos cualquier ciclo, sino uno "bueno". Qué interpretamos por "bueno" depende de muchos factores, pero básicamente podemos pensar que un ciclo hamiltoniano C es bueno cuando su valor es cercano a $c_+(C^*)$.

- Una forma de definir qué es una "bueno" es de acuerdo a qué distancia está $c_+(C)$ de $c_+(C^*)$, i.e., podemos pedir que C sea a lo sumo $1\,\%$ más caro que C^*.

- En cualquier caso, en computación no nos interesan las soluciones a un grafo particular, sino los algoritmos para generar soluciones.

- En este caso, queremos un algoritmo "bueno", i.e., un algoritmo polinomial que, dado cualquier grafo completo G y cualquier función de costos c, retorne un ciclo hamiltoniano "bueno". Más aún, queremos que el algoritmo sea rápido en la práctica y no simplemente polinomial.

- Por ejemplo, sería fantástico contar con un algoritmo lineal que, para cualquier grafo G y cualquier función de costos c, retorne un ciclo hamiltoniano C tal que $c_+(C) \leq (1+\epsilon)c_+(C^*)$, donde C^* es el ciclo hamiltoniano de peso mínimo de G. Esto garantizaría que siempre estamos a un factor ϵ de la solución óptima.

- Desafortunadamente, para TSP no se conoce dicho algoritmo. Peor aun...

Teorema 7 ([9])**.** *Sea $p(n)$ un polinomio cualquiera. Si $\mathsf{P} \neq \mathsf{NP}$, entonces no existe ningún algoritmo de tiempo polinomial que, para todo grafo G y toda función de costos c, retorne un ciclo hamiltoniano C de costo $c_+(C) \leq 2^{p(n)}c_+(C^*)$, donde C^* es un ciclo hamiltoniano de G con mínimo $c_+(C^*)$.*

- Este resultado es más bien teórico y, bien interpretado, dice que no busquemos un algoritmo que ofrezca garantías teóricas.

- En su lugar, busquemos algoritmos que funcionen bien en la "práctica", donde la "práctica" se refiere a ciertos inputs comunes.

- Hay al menos dos formas de estudiar qué la práctica:

  1. Conseguir instancias reales del problema y generar un conjunto de instancias representativas para comparar los algoritmos. A este conjunto de instancias se lo llama *benchmark* y pueden haber distintos benchmarks con distintas particularidades para cada problema.
  2. Restringiendo el problema a inputs que satisfagan ciertas condiciones particulares que aparecen en distintos casos prácticos.

- Un ejemplo sobre el segundo ítem surge cuando queremos resolver ciertos problemas de ruteo. Supongamos, por ejemplo, que tenemos un grafo H que representa una ciudad sobre la que se encuentran un conjunto de clientes V representado por vértices de H. Las aristas del grafo H tienen un peso no negativo que representa el costo de transitar dicha arista. Ahora, supongamos que queremos visitar a exactamente una vez a cada cliente de V. Ciertamente, la mejor forma de visitar a w inmediatamente después de visitar a v es recorriendo el camino mínimo de v a w. Luego, podemos definir un grafo completo G donde cada vértice es un cliente y cada arista vw tiene un costo $c(vw)$ que se corresponde al costo del camino mínimo de v a w en H. Cualquier ciclo hamiltoniano de G representa un recorrido posible de los clientes. De todos ellos queremos encontrar el de mínimo costo, i.e., queremos resolver TSP sobre G.

- En el ejemplo anterior H no tiene ciclos de peso negativo. En consecuencia, $c(vw) + c(wz) \geq c(vz)$ para todo par de clientes de G. A esta condición se la conoce como *desigualdad triangular*.

- En el problema general los costos no tienen por qué satisfacer la desigualdad triangular.

- Problema métrico del viajante de comercio (metric (S)TSP)

  Input: un (di)grafo completo G y una función de costos c que satisface la desigualdad triangular.

  Output: un ciclo hamiltoniano C de costo $c_+(C)$ mínimo.

- El Teorema 7 no es cierto si nos restringimos a inputs métricos. De hecho, la clase que viene veremos un método eficiente que garantiza que $c_+(C) \leq 2c_+(C^*)$.

- La pregunta que sigue es qué tan chico podemos hacer la constante de aproximación.

- La respuesta es que esta constante tiene algún límite, aunque no sepamos cuál es.

Teorema 8 ([2])**.** *Si* $\mathsf{P} \neq \mathsf{NP}$*, entonces existe una constante* $\epsilon > 0$ *tal que ningún algoritmo polinomial que, para cualquier (di)grafo completo* G *y una función de costos* c*, retorne un ciclo hamiltoniano* C *de costo* $c_+(C) \leq (1+\epsilon)c_+(C^*)$*, donde* C^* *es la solución óptima para* (G, c)*.*

- Nuevamente, este resultado es esencialmente teórico y nos dice que las garantías teóricas están acotadas.

- El lugar de buscar garantías cada vez mejores, conviene concentrarse en métodos que funcionen mejor en la práctica.

- Adicionalmente, nos cuenta algo que ya sabemos: el problema métrico es considerablemente más simple que el problema general. Ergo, deberíamos aprovechar la desigualdad triangular para mejorar nuestros algoritmos.

- Hay un caso importante del problema métrico que está definido cuando los vértices del grafo son puntos en el plano y la distancia es la distancia euclideana.

- Problema del viajante de comercio euclideano (Euclidean TSP)

  Input: un conjunto $P = \{p_1, \ldots, p_n\}$ de puntos en $\mathbb{R}^2$.

Output: una permutación $q_1, \ldots, q_n, q_{n+1} = q_1$ de P tal que $\sum_{i=1}^{n} d(q_i, q_{i+1})$ es mínima, donde d es la distancia euclideana.

- En este caso, existen algoritmos polinomiales que aproximan a la solución óptima tanto como queramos.

Teorema 9 ([1])**.** *Existe un algoritmo que, para cualquier $\epsilon > 0$ y conjunto de puntos P en $\mathbb{R}^2$, retorna una permutación con costo $c < (1 + \epsilon)c^*$ en tiempo $O(n^{1/\epsilon})$, donde c^* es la permutación de costo mínimo.*

- Nuevamente, este algoritmo es teórico, ya que el tiempo de ejecución para computar una solución que tenga *garantía* de estar a 1 % del valor óptimo es de $O(n^{100})$.

- En la práctica existen algoritmos más eficientes que suelen dar este tipo de garantías para inputs "comunes".

Referencias

[1] Sanjeev Arora. Polynomial time approximation schemes for Euclidean TSP and other geometric problems. In *37th Annual Symposium on Foundations of Computer Science (Burlington, VT, 1996)*, pages 2–11. IEEE Comput. Soc. Press, Los Alamitos, CA, 1996.

[2] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. Proof verification and the hardness of approximation problems. *J. ACM*, 45(3):501–555, 1998.

[3] J. A. Bondy and V. Chvátal. A method in graph theory. *Discrete Math.*, 15(2):111–135, 1976.

[4] G. A. Dirac. Some theorems on abstract graphs. *Proc. London Math. Soc. (3)*, 2:69–81, 1952.

[5] Leonhard Euler. Solutio problematis ad geometriam situs pertinentis. *Comment. Acad. Sci. U. Petrop*, 8:128–140, 1736. En Latín. Traducción en Ingles en Biggs, N. L., Lloyd, E. K., y Wilson, R. J., *Graph theory. 1736–1936*, **Oxford University Press**, 1986.

[6] Carl Hierholzer and Chr Wiener. Ueber die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren. *Math. Ann.*, 6(1):30–32, 1873. En Alemán. Traducción en Ingles en Biggs, N. L., Lloyd, E. K., y Wilson, R. J., *Graph theory. 1736–1936*, **Oxford University Press**, 1986.

[7] Oystein Ore. Note on Hamilton circuits. *Amer. Math. Monthly*, 67:55, 1960.

[8] E. M. Palmer. The hidden algorithm of Ore's theorem on Hamiltonian cycles. *Comput. Math. Appl.*, 34(11):113–119, 1997. Graph theory in computer science, chemistry, and other fields (Las Cruces, NM, 1991).

[9] Sartaj Sahni and Teofilo Gonzalez. P-complete approximation problems. *J. Assoc. Comput. Mach.*, 23(3):555–565, 1976.