Teórica 7: Heurísticas y metaheurísticas

1. Problemas de optimización combinatoria

En un ***problema de optimización combinatoria***, de un conjunto de objetos (soluciones) valuados con alguna función, se busca el objeto (solución) con ***mejor*** valor. Puede ser el objeto (solución) de mínimo valor o de máximo valor. El conjunto de soluciones valuadas tiene cardinal finito pero muy grande.

En las clases anteriores ya estudiamos algunos problemas de esta familia, como AGM y camino mínimo. Para estos dos casos presentamos algoritmos polinomiales, pero ésto no es lo común.

Si llamamos ***objetivo*** al valor de una solución y ***restricciones*** a las condiciones que deben ser satisfechas por toda solución para ser ***factible***, lo que nos interesa es determinar una solución factible que minimice (o maximice) el objetivo. A estas soluciones (podrían ser más de una) las llamamos ***soluciones óptimas*** u ***óptimos globales***.

Formalmente,

Definición 1. En un problema de ***optimización combinatoria*** de minimización, dado un conjunto (finito) S de soluciones factibles y una función objetivo a optimizar $f : S \to \mathbb{R}$, el objetivo es encontrar $s \in S$ tal que $f(s) \leq f(s')$ para todo $s' \in S$ (o $f(s) \geq f(s')$ si el problema es de maximización).

Nos interesa estudiarlos porque modelizan muchos problemas de la realidad que tienen importantes aplicaciones prácticas, como por ejemplo:

- problemas de logística, distribución de mercadería y ruteo de vehículos
- planificación de la producción industrial en distintos rubros
- planificación agraria y forestal
- localización de facilidades de distinto tipo (plantas industriales, estaciones de servicio, etc.)
- problemas de diseño y ruteo en redes de comunicaciones
- planificación de horarios de medios de transporte terrestre, ferroviario, aéreo, marítimo
- asignación de tripulaciones
- distribución en contenedores
- gestión de ingresos

Para muchos de los problemas de esta familia, no se conocen (y se piensa que no existen) algoritmos polinomiales. Pero entonces, ¿cómo resolvemos un problema de optimización combinatoria?

Si las instancias que tenemos que resolver no son muy grandes, podemos intentar con un esquema de fuerza bruta o backtracking. Pero veremos que esto muchas veces no es una buena alternativa.

2. Problema del viajante de comercio (TSP)

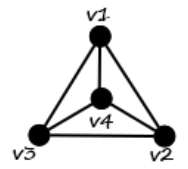
Comencemos primero definiendo ciclo hamiltoniano. Como comentamos hace algunas semanas, la noción de lo que hoy conocemos como ciclo hamiltoniano ya se mencionó en 1771, cuando Vandermonde estudió el ***problema del caballo de ajedrez***, cuyo objetivo era encontrar un camino circular de un caballo de ajedrez que visite todas las casillas del tablero exáctamente una vez.

Muchos años después, en 1855, Kirkman volvió a esta idea. Un tiempito más tarde, en 1857, Hamilton creó el ***juego icosiano***, y aunque fue un fracaso comercial, difundió este problema como unos desafíos ingeniosos y se ganó el reconocimiento imponiendo su nombre a este concepto.

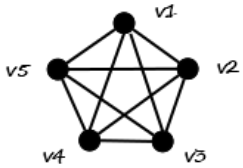
Definición 2.

- Un ciclo en un grafo G es un ***ciclo hamiltoniano*** si pasa por cada vértice de G una y sólo una vez.
- Un grafo se dice ***hamiltoniano*** si tiene un ciclo hamiltoniano.
- Un camino en un grafo G es un ***camino hamiltoniano*** si pasa por cada vértice de G una y sólo una vez.

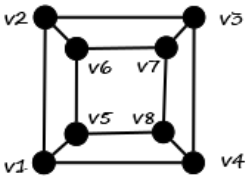
Ejemplo 1. *Para los primeros cuatro grafos se muestra un ciclo hamiltoniano, mientras los dos últimos no son grafos hamiltonianos.*



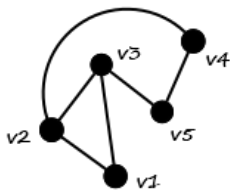
C = v1,v2,v3,v4,v1



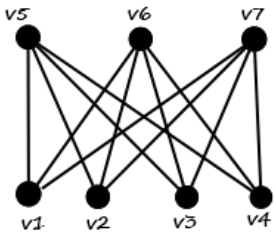
C = v1,v2,v3,v4,v5,v1



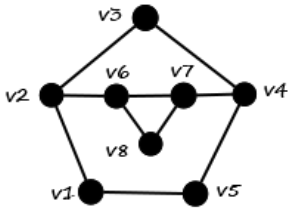
C = v1,v5,v6,v7,v8,v4,v3,v2,v1



C = v2,v1,v3,v5,v4,v2



No es hamiltoniano



No es hamiltoniano

No se conocen buenas caracterizaciones para grafos hamiltonianos. Es decir, no se conocen caracterizaciones que deriven en algoritmos polinomiales para decidir si un grafo es hamiltoniano o no.

Imaginemos que un viajante debe recorrer un conjunto determinado de ciudades. Cuenta con un vehículo para realizar el viaje, debe visitar exactamente una vez cada ciudad y finalmente retornar al origen. El viajante, obviamente, quiere seguir el ***mejor*** recorrido.

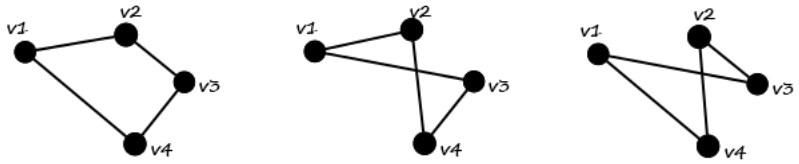
¿Pero cuál es el ***mejor*** recorrido? ¿El más corto (minimiza la distancia recorrida)? ¿El más rápido (minimiza el tiempo total de viaje)?

En términos de grafos, es encontrar un ciclo hamiltoniano de longitud mínima en un grafo completo con longitudes asociadas a sus aristas. Formalmente:

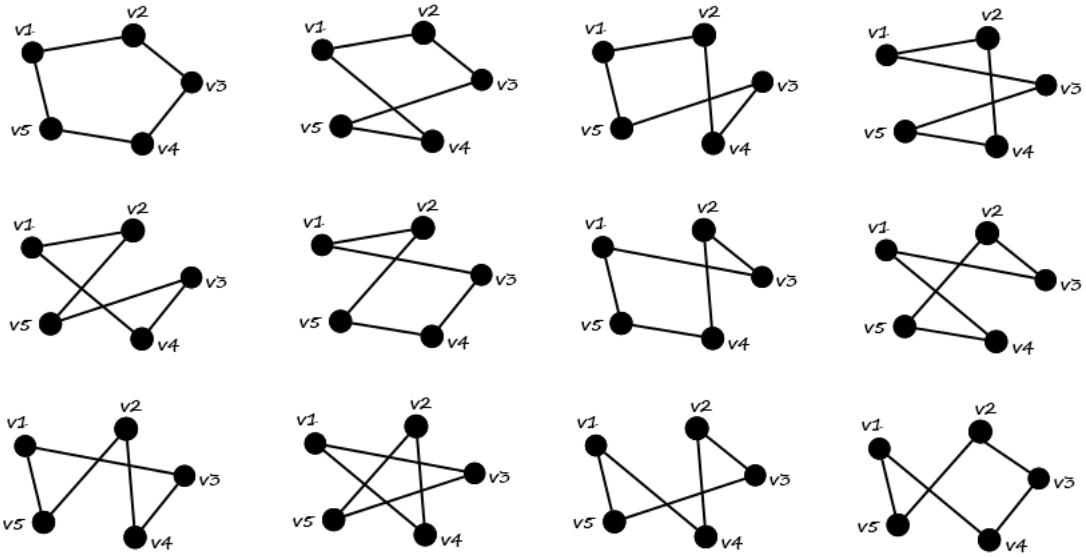
Definición 3. Dado un grafo $G = (V, X)$ con longitudes asignadas a las aristas, $l : X \to \mathbb{R}^{\geq 0}$, queremos determinar un circuito hamiltoniano de longitud mínima. Es decir, encontrar C^0 tal que:

$$l(C^0) = \text{mín}\{l(C) | C \text{ es un circuito hamiltoniano de} G\}.$$

Ejemplo 2. *Si el grafo tiene 4 vértices, tenemos 3 soluciones:*



Si el grafo tiene 5 ciudades, tenemos 12 soluciones:



Para 6 vértices hay 60 posibles recorridos, para 7 hay 360 posibles recorridos, para 8 hay 2520 y para 15... hay 43.589.145.600 posibles recorridos!!!

De forma general, si el grafo tiene n vértices tenemos $(n-1)!/2$ posibilidades.

No se conocen algoritmos polinomiales para resolver el problema del viajante de comercio. Tampoco se conocen algoritmos ϵ-aproximados polinomiales para el TSP general (sí se conocen cuando las distancias son euclideanas).

Y si necesitamos resolverlo, ¿qué hacmos?
Supongamos que queremos resolver el problema del viajante de comercio mediante fuerza bruta y tenemos una computadora que realiza mil billones de evaluaciones (calcular el costo de un recorrido) por segundo (1.000.000.000.000.000 x seg).

Para resolver una instancia de 20 vértices, debemos evaluar 19!/2 = 60822550204416000 posibilidades. Entonces tardaríamos 61 seg.

Para una instancia de 30 vértices, hay que evaluar 29!/2 = 4420880996869850977271808000000 posibilidades. Eso resultaría en... 1.401.852 siglos!!!

Aunque es posible desarrollar algoritmos exactos sofisticados que mejoren mucho estos tiempos, ésto muestra la necesidad de diseñar otras alternativas: algoritmos heurísticos.

3. Heurísticas y algoritmos aproximados

Hay problemas, como el TSP o el problema de la mochila entero, para los cuales no se conocen buenos (polinomiales) algoritmos exactos. En muchos escenarios, los algoritmos exactos que se tienen no son aplicables debido al tiempo de cómputo que insumen para resolver las instancias que nos interesan, debido a su tamaño. En estos casos, es posible proponer algoritmos heurísticos. Estos algoritmos, en general, dan buenas soluciones en un tiempo razonable, pero no garantizan que las soluciones que encuentran sean óptimas.

Este tipo de técnicas se puede usar, también, en casos donde el problema de optimización combinatoria no se pueda expresar fácilmente o es difícil de modelar.

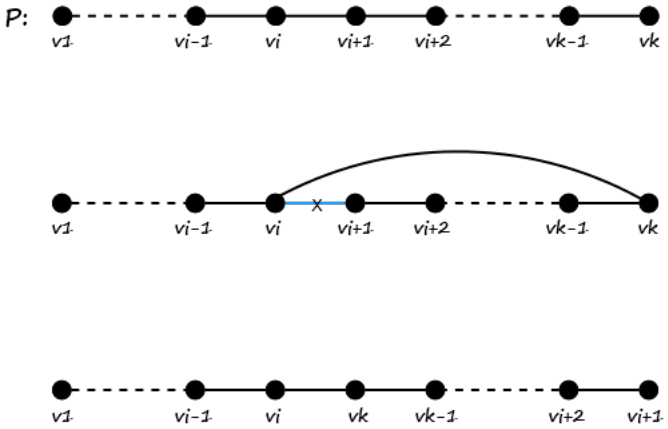
Las heurísticas clásicas, en general, construyen una solución inicial y después pueden aplicar una búsqueda local para mejorar esta primera solución.

No sólo para problemas de optimización combinatoria, también podemos desarrollar heurísticas para problemas de decisión.

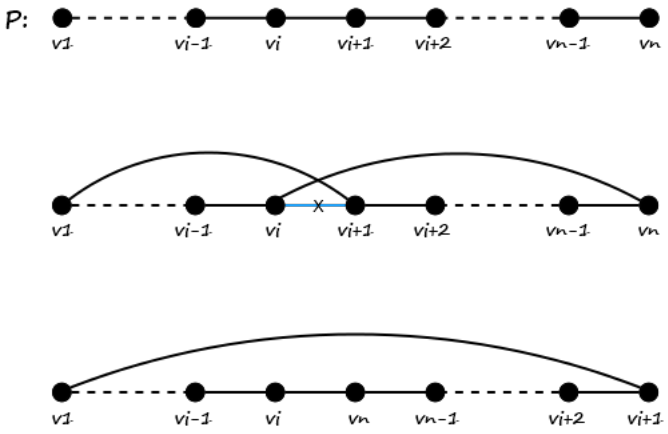
Desde acá al final de esta sección es opcional.
Pósa, en 1974, presentó una heurística para decidir si un grafo es hamiltoniano [5]. Si se retornar una respuesta positiva, seguro que el grafo es hamiltoniano. Pero si la respuesta es negativa, no hay garantía de que el grafo no tenga un ciclo hamiltoniano, sólo indica que el algoritmo no fue capaz de encontrarlo.

Esta heurística avanza construyendo un camino sin repetir vértices mientras sea posible. Cuando se *traba*, es decir que todos lo vértices adyacentes al último vértice agregado ya están en el camino, intenta una ***inversión*** en el camino que permita continuarlo.

Supongamos que el camino construido hasta el momento es $P = v_1, v_2, \ldots, v_k$ y todos los vértices adyacentes a v_k ya están en P. Si existe $v_i \neq v_{k-1}$ adyacente a v_k, podemos transformar P en un camino de igual longitud, P', mediante una inversión, que inserta la arista (v_k, v_i) en P y elimina la arista (v_i, v_{i+1}) de P, $P' = P_{v_1 v_i} + (v_i, v_k) + P_{v_k v_{i+1}}$. Ahora el vértice v_{i+1} es el nuevo último vértice del camino.



Una vez que $|P| = n$ (n es la cantidad de vértices del grafo), si $(v_n, v_1) \in X$ habremos encontrado un circuito hamiltoniano. Si no es así , podemos buscar $2 \leq i \leq n-2$, tal que $(v_i, v_n) \in X$ y $(v_1, v_{i+1}) \in X$. Si encontramos esta situación, tendremos un circuito hamiltoniano, $H = P_{v_1 v_i} + (v_i, v_n) + P_{v_n v_{i+1}} + (v_{i+1}, v_1)$.



Heurística para decidir si un grafo es hamiltoniano (Pósa)

```
heurHamiltoniano(G)
    entrada: G = (V, X) de n vertices
    salida: si retorna VERDADERO, P es un ciclo hamiltoniano

    k ← 1
    P[k] ← s (un vertice cualquiera)
    mientras k ≤ n hacer
        si existe v ∈ V \ P adyacente a P[k] entonces
            k ← k + 1
            P[k] ← v
        sino
            si existe i ∈ {1, ... k − 2} tal que (P[i], P[k]) ∈ X entonces
                invertir(P, i + 1, k)
            sino
                retornar FALSO
            fin si
        fin si
    fin mientras
    si (P[n], P[1]) ∈ X entonces
        retornar VERDADERO y P
    sino
        si existe i ∈ {2, ... k − 2} tal que (P[i], P[n]) ∈ X y (P[i + 1], P[1]) ∈ Xentonces
            invertir(P, i + 1, n)
            retornar VERDADERO y P
        sino
            retornar FRACASO
        fin si
    fin si
```

El procedimiento `invertir(P,a,b)`, invierte los vértices de P entre las posiciones a y b.

Algunas veces, un algoritmo no necesariamente nos brinda una solución óptima pero, sin embargo, es posible acotar el error que comete en el peor caso.

Definición 4. Si A es un algoritmo para un problema de optimización combinatoria (de minimización) con función objetivo a optimizar f y llamamos x^A a la solución dado por A y x^* a una solución óptima, decimos que A es un algoritmo ϵ – aproximado, con $\epsilon \in \mathbb{R}_{>0}$, si para toda instancia se cumple:

$$|f(x^A) - f(x^*)| \leq \epsilon |f(x^*)|$$

Vamos a estudiar diferentes heurísticas y algoritmos aproximados utilizando como problema ejemplo el viajente de comercio (TSP).

4. Heurísticas constructivas

Las ***heurísticas constructivas***, como su nombre lo indica, son métodos que construyen una solución factible de una instancia de un problema. Una posibilidad podría ser generar una solución aleatoria, pero no es esperable que esto de una solución de calidad aceptable. Por su facilidad de diseño e implementación, es muy frecuente utilizar

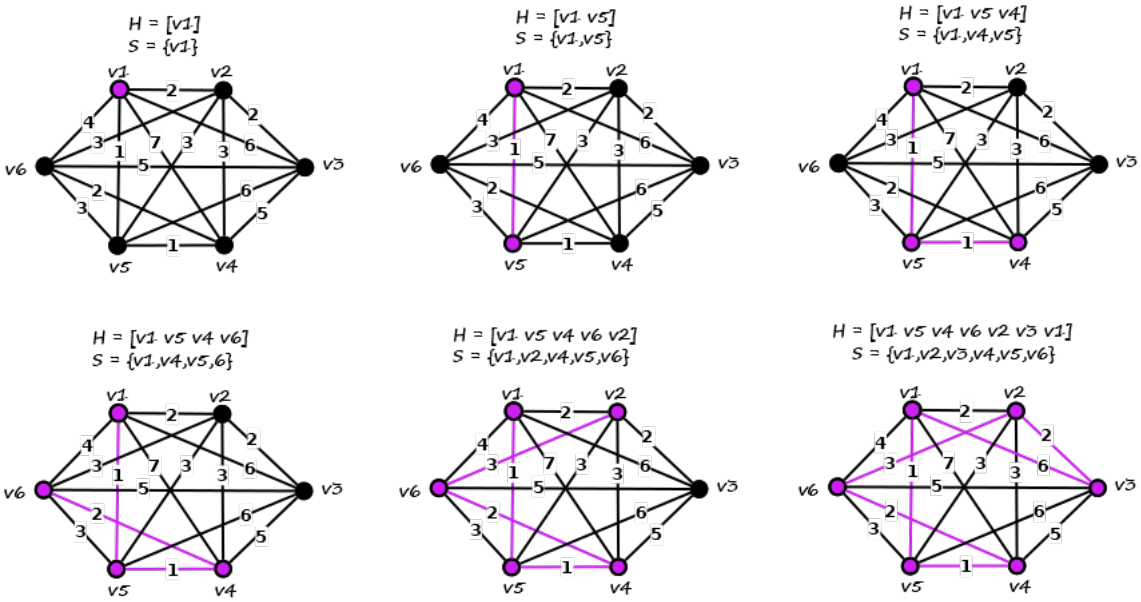
procedimientos golosos como heurísticas constructivas, que si bien no aseguran que nos brinden una solución óptima (y en general no lo hacen), retornan soluciones de calidad aceptables en compensación por el esfuerzo computacional que requieren.

Veamos algunos ejemplos para el TSP.

Heurística del vecino más cercano

Es la heurística más intuitiva para este problema y lo que haríamos naturalmente: En cada paso elegimos como siguiente lugar a vistar el que, entre los que todavía no visitados, se encuentre más cerca de donde nos encontramos. Claramente este es un procedimiento goloso.

Ejemplo 3. *Marcamos en violeta los vértices ya visitados y las aristas utilizadas para hacerlo.*

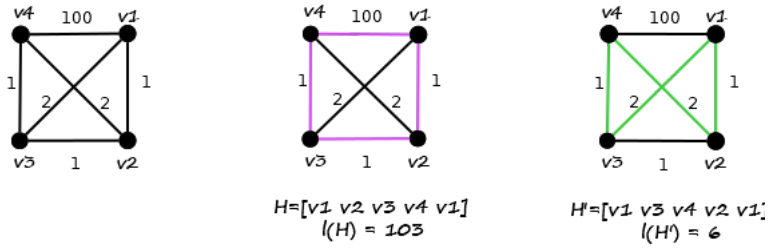


vecinoMasCercano(G)
entrada: $G = (V, X)$ de n vertices y $l : X \to \mathbb{R}$
salida: H un circuito hamiltoniano

$v \leftarrow$ un nodo cualquiera
$H \leftarrow [v]$
mientras $|H| \leq n$ **hacer**
  $w \leftarrow \arg\min\{l(v, w), w \in V \setminus H\}$
  $H \leftarrow H + w$
  $v \leftarrow w$
fin mientras
retornar H

Es fácil entontrar ejemplos donde este procedimiento funciona tan mal como uno quiera.

Ejemplo 4. *En este grafo, si se comienza por v_1, la heurística del vecino más cercano retorna H (el ciclo violeta) con $l(H) = 103$, mientras las solución óptima es H' (el ciclo verde) con $l(H') = 6$.*

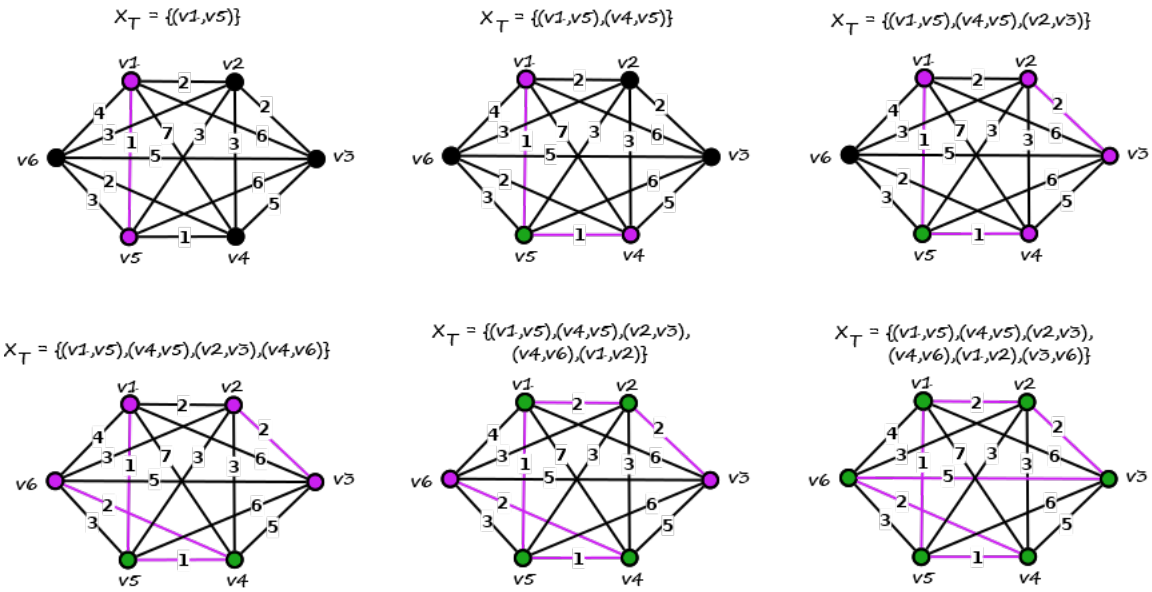


Heurística de la arista más corta

Podemos modificar un poco el algoritmo de Kruskal para que la solución sea un circuito hamiltoniano.

Lo que tenemos que tener cuidado ahora es no formar ciclos (antes de tiempo) y además que no haya vértices de grado 3 o mayor en el grafo generador que construimos. En cada iteración agregaremos la arista de menor peso entre todas las que cumplan estas dos caraterísticas. Si agregamos $n - 1$ aristas (en un grafo de n vértices), habremos formado un camino hamiltoniano. Para obtener un ciclo, debemos agregar la arista entre los dos vértices extremos de este camino. Este procedimiento también es goloso.

Ejemplo 5. *En el ejemplo mostramos las iteraciones de esta heurística, marcando en violeta las aristas seleccionadas y en verde los vértices ya saturados, es decir, los que tienen grado igual a 2 en (V, X_T).*



```
aristaMasCorta(G)
    entrada: G = (V, X) de n vertices y l : X → ℝ
    salida: H = un circuito hamiltoniano de G

    X_T ← ∅
    i ← 1
    mientras i ≤ n − 1 hacer
        e ← arg mín{l(e), e = (u, v) y d_{X_T}(u) ≤ 1 y d_{X_T}(v) ≤ 1 y
                no forma circuito con las aristas de X_T}
        X_T ← X_T ∪ {e}
        i ← i + 1
    fin mientras
    X_T ← X_T ∪ {(u, v)} con d_{X_T}(u) = 1 y d_{X_T}(v) = 1
    retornar H = (V, X_T)
```

Heurísticas de inserción

Las heurísticas de inserción comienzan con un ciclo cualquiera y en cada iteración incorporan un vértice al ciclo hasta llegar a incorporar a todos los vértices. En general la selección del próximo vértice a incorporar y donde insertarlo son decisiones golosas.

```
insercion(G)
    entrada: G = (V, X) de n vertices y l : X → ℝ
    salida: H un circuito hamiltoniano

    H ← [u, v, w]        (3 nodos cualquiera)
    mientras |H| ≤ nhacer
        ELEGIR un nodo de v ∈ V \ H
        INSERTAR v en H
    fin mientras
    retornar H
```

Las diferentes implementaciones de ***ELEGIR*** e ***INSERTAR*** dan lugar a distintas variantes de la heurística de inserción.

Algunas reglas para ***ELEGIR*** el nuevo vértice v para agregar al circuito pueden ser:

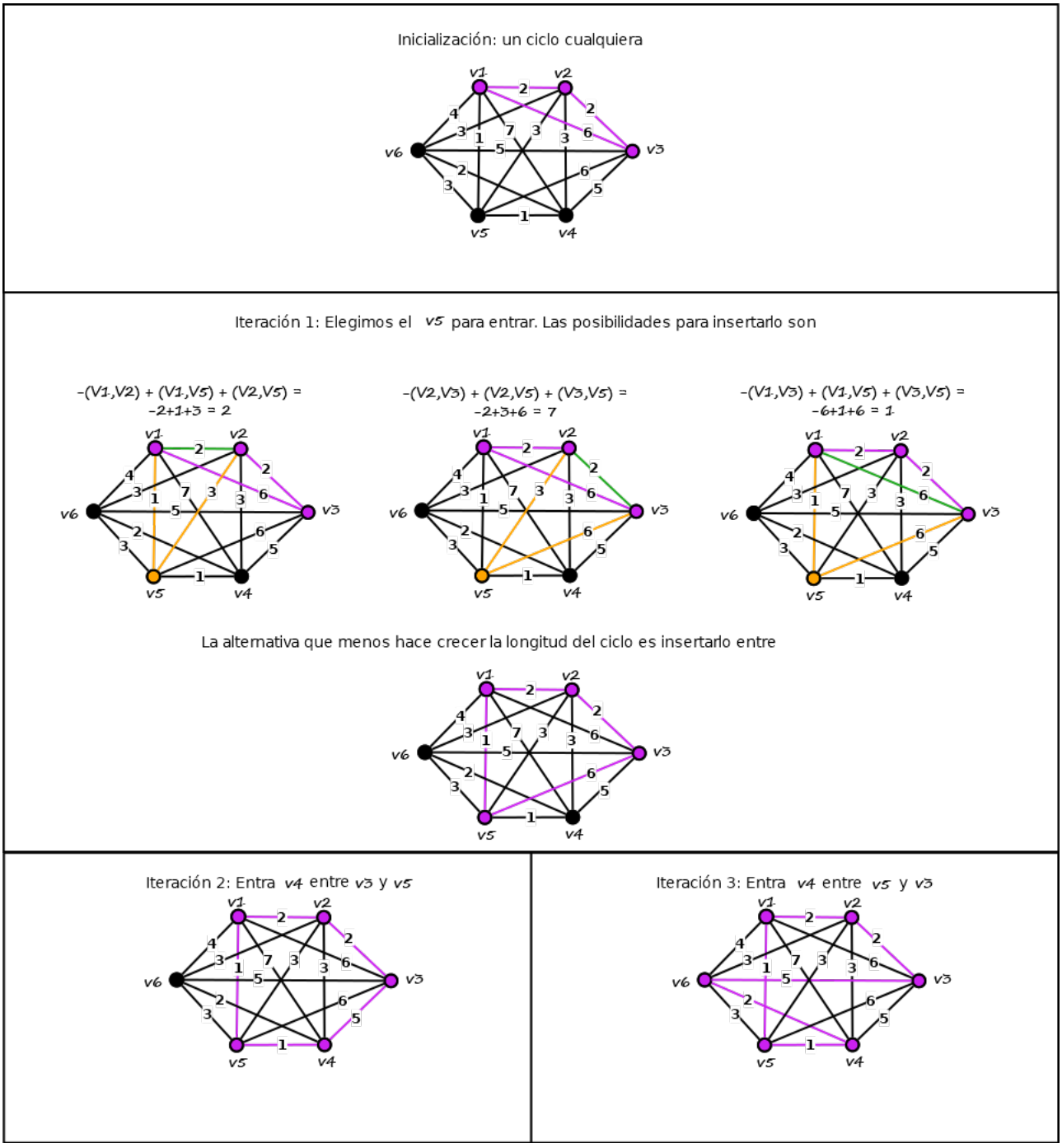
- el vértice más cercano a un vértice que ya está en el circuito.
- el vértice más lejano a un vértice que ya está en el circuito.
- el vértice ***más barato***, o sea el que hace crecer menos la longitud del circuito en ese paso.
- un vértice eligido al azar (este criterio no es goloso).

Para ***INSERTAR*** el vértice v elegido una posibilidad es elegir dos vértices consecutivos en el circuito H, v_i, v_{i+1} tal que:

$$l((v_i, v)) + l((v, v_{i+1})) - L((v_i, v_{i+1}))$$

sea mínimo, e insertar v entre v_i y v_{i+1}.

Ejemplo 6. *En este ejemplo utilizamos como criterio el vértice más cercano a un vértice que ya está en el circuito para* ***ELEGIR****:*



En el caso de grafos euclideanos (por ejemplo grafos en el plano $\mathbb{R}^2$), se puede implementar un algoritmo de inserción:

- Usando la cápsula convexa de los nodos como circuito inicial.
- Insertando en cada paso un nodo v tal que el ángulo formado por las aristas (w, v) y (v, z), con w y z consecutivos en el circuito ya construido, sea máximo.

Heurística del árbol generador *(esta subsección es opcional)*

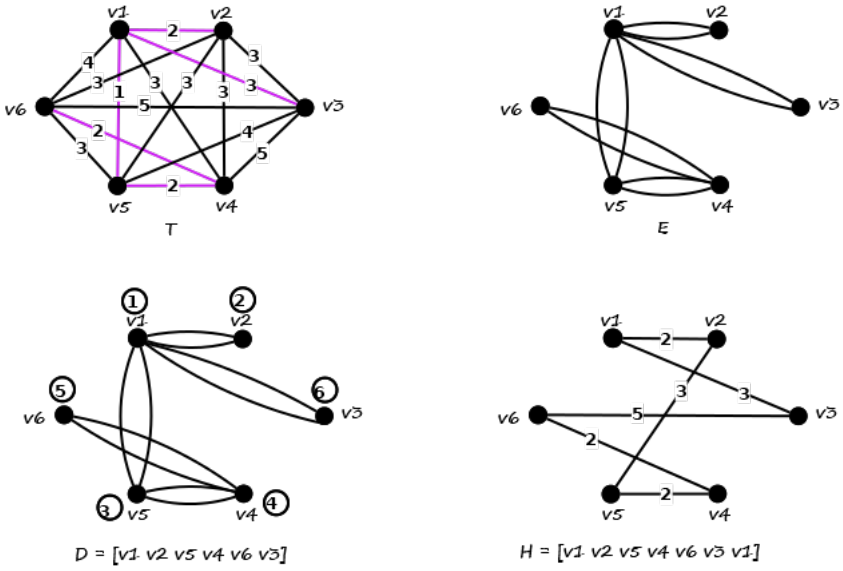
Si bien no se conocen algoritmos polinomiales aproximados para el TSP en general (volveremos a ésto en el final de la materia), si se conocen para casos particulares. Veremos que esta heurística es 1-aproximada en el caso de grafos euclideanos.

Lo primero que hace este algoritmo es buscar un AGM del grafo T. Luego duplica cada arista de T y recorre los vértices según DFS. Por último, arma un circuito hamiltoniano recorriendo los vértices en este orden y agregando la arista desde el último vertice al primero.

```
heurAG(G)
  entrada: G = (V, X) de n vertices y l : X → ℝ
  salida: H circuito hamiltoniano

  T ← AGM(G)
  E ← duplicar las aristas de T
  D ← recorrer E usando DFS
  H ← armar un circuito hamiltoniano siguiendo el orden dado por D
  retornar H
```

Ejemplo 7.



En este ejemplo, v_1 y v_2 son vértices consecutivos de H y esta arista también pertenece a E. En cambio, para v_2 y v_5, que también son consecutivos en H, la arista que los une, la (v_2, v_5) no pertenece a E. El camino en E entre v_2 y v_5 es $[v_2 v_1 v_5]$. Para armar H se quitan de E las aristas (v_2, v_1) y (v_1, v_5) y se agrega la arista (v_2, v_5).

Algo similar pasa para v_6 y v_3. Se quitan de E las aristas (v_6, v_4), (v_4, v_5), (v_5, v_1) y (v_1, v_3) y se agrega la arista (v_6, v_3).

En principio, el intercambio de aristas que se hace en el ejemplo para pasar de E a H, podría ser muy malo. Sin embargo, para cierta clase de grafos es posible controlar eso.

Definición 5. Un grafo pesado completo $G = (V, X)$ es euclideano si la función de peso, $l : X \to \mathbb{R}_{\geq 0}$, cumple la desigualdad triangular:

$$l((i,k)) \leq l((i,j)) + l((j,k)) \text{ para todo } i, j, k \in V.$$

Teorema 1. *Si las distancias del grafo G es euclideano, la heurística del árbol generador es un algoritmo 1-aproximado. Esto es, su perfomance en el peor caso está dada por*

$$l(H^H)/l(H^*) \leq 2,$$

donde H^H es el ciclo hamiltoniano retornado por la heurísitca aplicada a G y H^* es un circuito hamiltoniano óptimo de G.

Demostración: Sea T un AGM de G.

Cualquier ciclo hamiltoniano menos una arista es un camino hamiltoniano, que es, a su vez, un árbol generador. Como el peso de toda arista es mayor o igual a 0, $l(H^*) \geq l(H^*$ menos una arista$) \leq l(T)$.

Por otro lado, el peso total de E es $l(E) = 2l(T)$. Cada arista de H^H es una arista de E o une directamente dos vértices terminales de un camino de E que no está en H^H. Como se cumple la desigualdad triangular, $l(H^H) \leq l(E)$.

Entonces

$$l(H^H) \leq l(E) = 2l(T) \leq 2l(H^*) \Longrightarrow \frac{H^H(G)}{H^*(G)} \leq 2.$$

■

En realidad no es necesaria pasar por E, se pueden enumerar los vértices utilizando DFS en T. Pero enunciar así el algoritmo facilita la demostración.

Este teorema muestra que esta heurística es un algoritmo 1-aproximado para el problema del TSP para grafos euclideanos. Hay variantes más sofisticadas de este algoritmo, como por ejemplo el algoritmo de Christofides que es 1/2-aproximado.

4.1. Heurísticas de mejoramiento - Algoritmos de búsqueda local

¿Cómo podemos mejorar la solución obtenida por alguna heurística constructiva como las anteriores?

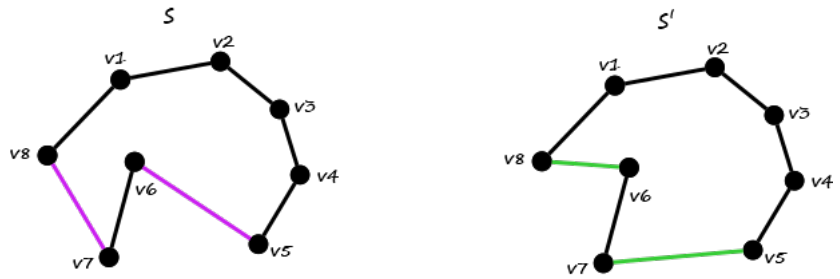
Dada una solución de una instancia de un problema particular, en varias situaciones es útil definir un conjunto de soluciones ***cercanas***, en algún sentido, a esa solución.

Definición 6. Dado un problema específico y una instancia de este problema con conjunto de soluciones factibles S, una ***vecindad*** es una mapeo $N : S \to \mathcal{P}(S)$, donde $\mathcal{P}(S)$ es el conjunto potencia o de partes de S. Para $s \in S$, al conjunto $N(s)$ lo llamamos vecindad de s.

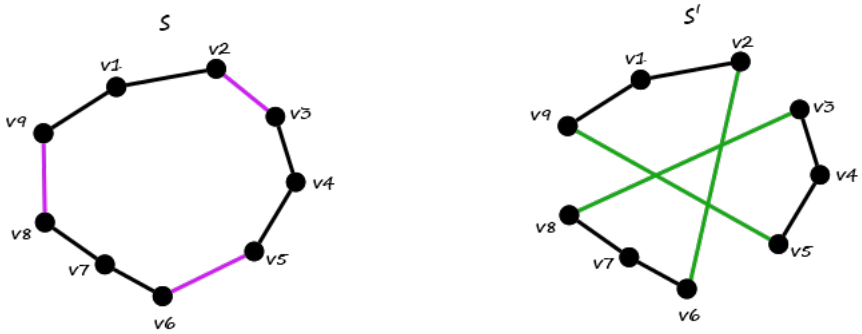
Ejemplo 8. *En el TSP, una vecindad muy utilizada se llama 2-opt, definida por*

$$N_2(s) = \{s' : s' \in S \text{ y } s' \text{ puede ser obtenido desde } s \text{ reemplazando dos aristas del ciclo por otras dos } \}.$$

Es decir, si $s = [v_1 v_2 ... v_n]$, obtenemos s' intercambiando dos aristas de s, (v_i, v_{i+1}) y (v_j, v_{j+1}) por (v_i, v_j) y (v_{i+1}, v_{j+1}) para algún par i, j.



También se utiliza 3-opt, donde, si $s = [v_1 v_2 ... v_n]$*, obtenemos* s' *intercambiando tres aristas de* s*,* (v_i, v_{i+1})*,* (v_j, v_{j+1}) *y* (v_k, v_{k+1}) *por* (v_i, v_{j+1})*,* (v_k, v_{i+1}) *y* (v_j, v_{k+1}) *para alguna terna* i, j, k*.*



Esta definición de vecindad puede ser generalizada a N_k*,* k*-opt, donde se intercambian* k *aristas del ciclo definido por* s*.*

Ejemplo 9. *En el problema de AGM, se puede definir una vecindad para un árbol generador* T *de un grafo* $G = (V, X)$ *como:*

$N(T) = \{T' : T'$ *es una AG y* T' *se obtiene desde* T *agregando una arista de* $X \setminus T$ *y borrando una del ciclo que se forma* $\}$*.*

Muchas veces, si bien encontrar una solución óptima global es un problema computacionalmente difícil, encontrar una solución que sea localmente óptima (que no hay una solución mejor en su vecindad) si es posible.

Definición 7. Dado un problema de minimización con función objetivo f, una instancia con soluciones factibles S y una definición de vecindad $N : S \to \mathcal{P}(S)$, $s \in S$ es un ***óptimo local*** si $f(s) \leq f(s')$ para todo $s' \in N(s)$.

Los algoritmos de búsqueda local o mejoramiento, justamente encuentran un óptimo local a partir de una solución inicial, con la esperanza que esta solución sea también un óptimo global o de valor cercano al óptimo. Es una técnica iterativa que explora un conjunto de soluciones factibles, moviéndose de una solución s a otra solución en la vecindad $N(s)$. Estos movimientos se realizan con el objetivo de alcanzar una solución *buena* (óptima o cercana a la óptima) según la función f que se quiere optimizar (minimizar o maximizar).

El siguiente algoritmo de Lin y Kernighan está basado en la vecindad 2-opt.

2-opt(G)
entrada: $G = (V, X)$ `de` n `vertices y` $l : X \to \mathbb{R}$
salida: H `un circuito hamiltoniano`

$H \leftarrow$ `solucionInicial`(G)
mientras `sea posible` **hacer**
`elegir` (u_i, u_{i+1}) `y` $(u_k, u_{k+1}) \in H$ `tal que`
$c_{u_i u_{i+1}} + c_{u_k u_{k+1}} > c_{u_i u_k} + c_{u_{i+1} u_{k+1}}$
$H \leftarrow H \setminus \{(u_i, u_{i+1}), (u_k, u_{k+1})\} \cup \{(u_i, u_k), (u_{i+1}, u_{k+1})\}$
fin mientras
retornar `H`

Esta idea se extiende en las heurísticas k-opt donde se hacen intercambios de k aristas. Es decir, en vez de sacar dos aristas, sacamos k aristas de H y vemos cual es la mejor forma de reconstruir el circuito. En la práctica se usa sólo 2-opt o 3-opt, porque el estudio empírico ha demostrado que el esfuerzo computacional de alternativas más complejas no da buen resultado.

Cuando el problema no es simétrico, ésto es que el costo de un arco dependerá de la dirección en que lo atravesemos ($l(v, u)$ puede ser distinto al costo $l(u, v)$), el intercambio 2-opt invierte la dirección de parte del circuito y esto puede modificar mucho el costo de esa parte (además de tener que calcularlo). En cambio, 3-opt mantiene la dirección de todas las partes del circuito.

En vez de elegir para sacar de H un par de aristas cualquiera que nos lleve a obtener un circuito de menor longitud, podríamos elegir, entre todos los pares posibles, el par que nos hace obtener el menor circuito.

Un refinamiento de la búsqueda local, puede ser elegir $s' \in N(s)$ tal que $f(s') \leq f(\bar{s}) \forall \bar{s} \in N(s)$. Ésto lleva más trabajo computacional en cada ciclo del **mientras**, pero la mejora en cada iteración sería mayor (o al menos igual).

Es decir, una opción es moverse a la primera solución vecina encontrada que mejore la función objetivo, mientras que la otra opción es moverse a la mejor solución de la vecindad. En la primera opción es esperable que se tarden más iteraciones en llegar a un óptimo local y en la segunda que cada iteración insuma más tiempo.

Pero según la definición de vecindad, puede ser que sea impracticable computacionalmente explorar toda $N(s)$, porque consume mucho tiempo. En esos casos, también puede implementarse una estrategia de compromiso, como por ejemplo explorar sólo una porción de la vecindad y quedarse con la mejor solución encontrada en esa porción. En lugar de considerar $N(s)$, se restringe la búsqueda a un subconjunto $V^* \subset N(s)$ con $|V^*| \ll |N(s)|$.

Éstas son decisiones a tomar al momento de implementar un algoritmo de búsqueda local que estará basada en la definición de vecindad y un análisis empírico del comportamiento de cada estrategia.

El esquema básico de un algoritmo de búsqueda local es el siguiente:

```
busquedaLocal(...)
    entrada: ...
    salida: s solucion factible

    s ← solucionInicial(G)
    mientras ∃s' ∈ N(s) con f(s') < f(s) hacer
        elegir s' ∈ N(s) tal que f(s') < f(s)
        s ← s'
    fin mientras
    retornar s
```

Ejemplo 10. *Queremos resolver el siguiente problema:*

- *Tenemos que asignar n tareas a una sola máquina.*
- *Cada trabajo j tiene un tiempo de procesamiento p_j y una fecha prometida de entrega d_j.*
- *El objetivo es minimizar*

$$T = \sum_{j=1}^{n} \text{máx}\{(C_j - d_j), 0\}$$

donde C_j es el momento en que se completa el trabajo j.

Cómo elegir las soluciones iniciales:

- *Mediante alguna heurística constructiva golosa.*
- *Se podría tomar cualquier permutación random de las tareas.*

Determinación de los vecinos de una solución dada: *Podemos definir como soluciones vecinas las que se obtengan de la solución actual cambiando la posición de un trabajo con otro.*

En un problema con 4 trabajos, los vecinos de $s = (1, 2, 3, 4)$ serán:

$$N(s) = \{(2, 1, 3, 4), (3, 2, 1, 4), (4, 2, 3, 1), (1, 3, 2, 4), (1, 4, 3, 2), (1, 2, 4, 3)\}$$

5. Metaheurísticas

A comienzos de los 80, aunque algunas ideas sean anteriores, se comenzaron a desarrollar lo que se conoce como metaheurísticas o sistemas inteligentes. Estos métodos proveen metodologías generales que guían la exploración del espacio de búsqueda y que conducen a la construcción de heurísticas específicas para cada problema. No son técnicas para un problema específico. El objetivo es lograr una búsqueda más amplia en el espacio de soluciones que la obtenida mediante un algoritmo de búsqueda local tradicional para no quedar estancados en óptimos locales. Las técnicas metaheurísticas van desde algoritmos simples de búsqueda local a complejos procesos de aprendizaje, en muchos casos son algoritmos no determinísticos y algunas utilizan memoria de la búsqueda para guiar los pasos futuros. Para la implementación de un algoritmo basado en alguna técnica metaheurística, lo principal es

poder determinar una buena representación de las soluciones y de los movimientos que llevan de una solución a otra.

Muchas de esas técnicas están inspiradas en el comportamiento de sistemas de la naturaleza o procesos físicos, como los algoritmos genético o evolutivos, colonia de hormigas, de optimización de enjambre de partículas, de optimización de ondas de agua, algoritmos basados en el principio de selección clonal, de optimización de reacciones químicas, de búsqueda de armonía, de recocido simulado, de optimización basada en la enseñanza y el aprendizaje, entre muchos otros. En otro grupo se encuentran, por ejemplo, búsqueda tabú, GRASP y búsqueda local variable (VNS).

Las metaheurísticas se destacan por su flexibilidad y adaptabilidad a modificaciones de los datos o del problema una vez que ya se obtuvo un resultado y por ser fáciles de implementar y transportables. Usan información específica del problema y aprendizaje adquirido para lograr mejores soluciones y tienen mecanismos que reducen el estancamiento en soluciones de baja calidad. Se basan en tener una gran capacidad de cálculo.

6. Búsqueda tabú

Este cuatrimestre vamos implementar en el laboratorio la metaherurística ***búsqueda tabú***.

La ***búsqueda tabú***, BT, fue introducida por Fred Glover en 1986 [1, 4, 2, 3]. Es una metaheurística que guía una heurística de búsqueda local para explorar el espacio de soluciones, con el principal objetivo de evitar quedar atascados en un óptimo local. La base es similar a la búsqueda local, ya que el procedimiento iterativamente se mueve de una solución a otra hasta que se cumple algún criterio de terminación.

Cada solución $s' \in N(s)$ es alcanzada desde s realizando una operación llamada **movimiento**.

BT restringe la definición del problema de optimización local a un subconjunto $V^* \subset N(s)$ con $|V^*| \ll |N(s)|$ en una forma estratégica, haciendo uso sistemático de ***memoria*** para explotar el conocimiento ya adquirido de la función $f(s)$ y la vecindad $N(s)$. Considera V^* generados estratégicamente en lugar de hacerlo de forma aleatoria, con el objetivo de que el mínimo de f sobre V^* sea lo más cercano posible al mínimo sobre $N(s)$.

Con heurísticas basadas en la técnica de descenso standard, como búsqueda local, puede caerse en un mínimo local de f. Para salir de un mínimo local, el procedimiento tiene que permitir moverse de s a s' aún si $f(s') > f(s)$. Ésto puede generar ciclos, haciendo que el procedimiento repita indefinidamente una secuencia de soluciones. TS evita eso utilizando una estructura de memoria que prohibe o penaliza ciertos movimientos o soluciones que podrían hacerlo retornar a una solución ya visitada.

Ésto se puede poner en práctica haciendo que la vecindad de s, $N(s)$, dependa también de la iteración k actual, memorizando las soluciones y vecindades consideradas en iteraciones anteriores a la k. Entonces, el conjunto explorado en la iteración k será $V^* \subseteq N(s,k)$. La elección de buenas $N(s,k)$ es determinante para el éxito del procedimiento. Una forma de matener esta memoria es mediante una ***lista tabú***, T, que almacene las $|T|$ últimas soluciones visitadas y definiendo $N(s,k) = N(s) \setminus T$. Esta estrategia evita ciclos de longitud menor o igual a $|T|$. En la práctrica, guardar $|T|$ soluciones puede requerir mucho espacio y puede ser costoso saber si una solución de $N(s)$ está en T.

Una opción más práctica y efectiva de conservar memoria puede ser definir la vecindad de s en términos de movimientos que transformen a s en nuevas soluciones. Para cada solución s se consideran un conjunto de movimientos $M(s)$ que pueden ser aplicados a s para obtener una nueva solución. Si $m \in M(s)$, al aplicar m a s se obtiene una nueva solución $s' = s \oplus m$, es decir $N(s) = \{s' : s' = s \oplus m \ \forall m \in M(s)\}$. Si estos movimientos tienen "reverso", T puede consistir de los movimientos reversos asociados con los movimientos ya hechos. Por ejemplo, si en el TSP, dado un ciclo hamiltoniano, s, definimos su vecindad según 2-opt, el movimiento consiste en reemplazar dos aristas

del ciclo. Se puede considerar como movimiento cada una de las aristas que se sacan o el par.

También se puede clasificar como tabú ciertos ***atributos***, prohibiendo soluciones que los tengan. Para el TSP, se puede considerar un atributo prohibido, por ejemplo, el uso de una determinada arista en el ciclo hamiltoniano.

En todos los casos, hay que definir el tamaño de la lista tabú y el tiempo de permanencia de una solución, movimiento o atributo en ella. Algunas posibilidades son:

- valor fijo,
- valor aleatorio entre un t_{min} y t_{max} dados a priori,
- valor variable de acuerdo al tamaño de la lista y las variaciones del valor de la función objetivo.

Esta decisión estará basada en experimentación.

En general, listas tabú que almacenan atributos o movimientos son más efectivas, pero pueden generar un nuevo problema. Cuando un atributo o movimiento es clasificado como tabú, puede hacer que más de una solución se vuelva tabú (solución prohibida). Algunas de estas soluciones pueden ser de buena calidad y aún no haber sido visitadas. Para permitir que estas soluciones sean visitadas, se define una ***función de aspiración***. Si la función de aspiración aplicada a una solución actualmente clasificada como tabú está sobre cierto umbral, la solución es considerada como permitida. Una función de aspiración muy usada es si s' es una solución tabú, se permite si $f(s') < f(s^*)$, donde s^* es la mejor solución encontrada hasta el momento (se permiten soluciones que son mejores que la mejor encontrada hasta ese momento). Otra posibilidad es que cuando todos los movimientos o vecinos posibles son tabú, se elige alguno de ellos.

Todas estas alternativas representan memoria a ***corto plazo***.

Esquema general:

```
busquedaTabu(...)
    entrada: ...
    salida: s solucion factible

    s ← solucionInicial
    s* ← s
    inicializar la lista tabu T
    inicializar la funcion de aspiracion A
    mientras no se verifique criterio de parada hacer
        definir V*
        elegir s' ∈ V* tal que f(s') < f(s̄)  ∀s̄ ∈ V*
        actualizar la funcion de aspiracion A
        actualizar la lista tabu T
        si f(s') < f(s*) hacer
            s* ← s'
        fin si
        s ← s'
    fin mientras
    retornar s*
```

Para implementar este esquema se debe:

- Determinar el conjunto de soluciones factibles S.
- Determinar la función objetivo f.
- Dar un procedimiento para generar los elementos de $N(s)$.
- Decidir el tamaño del conjunto $V^* \subseteq N(s)$ que será considerado en cada iteración.
- Definir la lista Tabú T y su tamaño.
- Definir la función de aspiración y el umbral de aceptación.
- Definir criterios de parada.

Los ***criterios de parada*** más simples son:

- Se encontró una solución óptima (si es posible saberlo).
- $\{s' \in N(s) : s' \text{ no es tabú o } A(s') \leq A(s^*)\} = \emptyset$.
- Se alcanzó el número máximo de iteraciones permitidas.
- El número de iteraciones realizadas sin modificar s^* es mayor que un número máximo determinado.
- Cualquier combinación de los anteriores.

En algunas aplicaciones, la incorporación de la memoria ***a largo plazo*** puede mejorar significativamente la eficiencia del procedimiento. Esta memoria considera:

- Frecuencia: guardar la frecuencias de ocurrencias de atributos en las soluciones visitadas para penalizar o premiar (según convenga) movimientos que usan atributos muy usados en el pasado.
- Intensificación: En algunos casos puede ser útil intensificar la búsqueda en alguna región de S porque es considerada buena bajo algún criterio, como por ejemplo aparición de atributos asociados a buenas soluciones encontradas. Para ésto se puede dar prioridad a soluciones que tengan atributos en común con la mejor solución actual, introduciendo un término adicional a la función f que penalice soluciones lejanas a la mejor solución actual. Esto se puede mantener durante pocas iteraciones para después permitir moverse a otra región de S.
- Diversificación: El objetivo es explorar nuevas regiones de S no exploradas. Como antes, mediante la adición de un término adicional en f se prioriza atributos que no han sido usados frecuentemente (soluciones lejanas a la actual). Los pesos de ambos términos son modificados durante el proceso de búsqueda para alternar fases de intensificación con fases de diversificación.
- Recorrer el camino de soluciones entre dos soluciones prometedoras.

7. Evaluación de técnicas heurísticas y metaheurísticas

La eficiencia de los algoritmos exactos se mide principalmente por el tiempo de cómputo (o memoria en algunas circunstancias) necesario para resolver una instancia y hay un marco teórico que ayuda al análisis. En cambio, para los métodos heurísitcos se agrega un factor que define la eficiencia del algoritmo y es necesario evaluar: la calidad de las soluciones encontradas. Es esperable que estas dos dimensiones, calidad de soluciones y tiempo de cómputo consumido, sean contrapuestas y que los métodos sacrifiquen el nivel de una de ellas a favor de la otra. En general, lo métodos más rápidos tienen peor calidad de soluciones. Dependiendo del contexto de aplicación, será el grado de compromiso entre estos dos factores que será más adecuado.

En algunas oportunidades es posible realizar un estudio teórico del peor caso o hacer un análisis probabilístico de la eficiencia del algoritmo desarrollado. Sin embargo, en la mayoría de los casos, los desarrollos heurísticos sólo pueden evaluarse empíricamente, mediante la experimentación computacional, ejecutando el procedimiento a una colección de instancias específicas y comparando la calidad de la solución obtenida y el tiempo computacional consumido. La construcción de un método heurístico implica una etapa importante de experimentación computacional, hasta conseguir valores de los parámetros necesarios para cada técnica que lleven a construir un algoritmo que de buenas soluciones.

Un experimento es un conjunto de pruebas que se ejecutan bajo condiciones controladas para, en este caso, evaluar el desempeño de un nuevo desarrollo. En la experimentación computacional de un algoritmo, un experimento consiste en resolver una serie de instancias del problema utilizando una implementación específica. Para ésto se deben implementar los algoritmos, elegir un entorno informático, elegir las instancias de prueba, seleccionar medidas de rendimiento, establecer las opciones a testear e informar los resultados. Cada uno de estos factores puede tener gran impacto en los resultados obtenidos, por lo que es necesario documentar cada uno de ellos. El objetivo de la experimentación puede ser para comparar el rendimiento de diferentes algoritmos para el mismo problema o para evaluar y definir parámetros de un algoritmo de forma aislada, y ésto puede influir en su diseño.

Como mencionamos, es imprescindible un ***conjunto de instancias de prueba*** y es crítica su selección. Es necesaria una cantidad suficiente de instancias de prueba realistas, que tengan el tamaño y la variedad para abarcar todas las características de interés. Las características relevantes dependerán del problema y contexto de utilización del algoritmo (tamaño, densidad, estructura, distribución de parámetros, por ejemplo). En general, cuantas más instancias se evalúen, más informativo será el análisis y es muy sustancial reportar casos que prueban los límites del algoritmo o lo hacen fallar. Para la experimentación, se pueden utilizar instancias provenientes de diferentes fuentes:

Instancias reales: Si se trata de un desarrollo proveniente de una problemática real, las mejores instancias de prueba son probablemente las tomadas de una aplicación real. Lamentablemente, rara vez es posible obtener un conjunto significante de datos reales. Muchas veces las organizaciones son reacias a hacerlos públicos y otras veces no cuentan con esta información de forma sistemática y digital. Además, la recopilación de un conjunto de datos real es un esfuerzo tedioso que requiere mucho tiempo y no siempre se dispone de éste.

Variantes random de instancias reales: En esta alternativa se intenta replicar la estructura de las instancias reales. Para producir nuevas instancias, se conserva la estructura de la aplicación real, pero los parámetros se cambian aleatoriamente.

Instancias test: Algunos problemas son nuevos en la literatura mientras que otros están bastante tratados, con muchos métodos propuestos para resolverlos. A medida que se avanza en el estudio de un problema, los conjuntos de instancias utilizadas en las diferentes experimentaciones pueden fusionarse en colecciones de referencia clásicas utilizadas por todos los investigadores que trabajan en el mismo problema. Por ejemplo, TSPLIB (http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsplib.html), creada por Bixby y Reinelt en 1990, es una bibloteca de instancias testigo para el TSP y algunas de sus variantes. Una gran ventaja de esta alternativa, es que muchas veces se cuenta con las soluciones óptimas de las instancias, o al menos, cotas de

estos valores. También es muy importante estas instancias porque permiten la comparación con desarrollos de otros autores.

Instancias generadas al azar: Estas instancias se crean de manera totalmente artificial y aleatoria, pero sus características pueden imitar instancias reales o ser diseñadas siguiendo características críticas que impacten en la performance del algoritmo. Ciertos parámetros son generados al azar para conseguir una cantidad suficientes de instancias. Es necesario asegurar que las instancias creadas sean lo suficientemente diversas. Para algunos problemas, fue posible diseñar generadores que producen instancias con valor óptimo conocido. Es muy importante que esté bien documentado el generador.

Evaluar la calidad de las soluciones encontradas, por lo general, no es una tarea fácil, ya que es muy difícil establecer qué tan cerca está una solución heurística del valor óptimo. Usualmente, los problemas que se abordan mediante técnicas heurísticas son problemas ***NP-Difícil*** (no se conocen algoritmos polinomiales para resolverlos), por lo que los algoritmos exactos sólo son aplicables a instancias pequeñas. Entonces, para instancias interesantes para los métodos heurísticos frecuentemente no se conocen los valores óptimos. Entre otras, algunas posibilidades para realizar esta evaluación son:

Soluciones óptimas en instancias pequeñas: Se aplica un algoritmo exacto y el método heurístico a instancias pequeñas y se comparan los resultados obtenidos. Por un lado, el tiempo necesario para ejecutar el algoritmo exacto es grande. Por otro lado, muy frecuentemente los resultados obtenidos para instancias pequeñas no son representativos de lo que sucede en instancias grandes. Hay heurísticas asintóticamente óptimas para algunos problemas (cuanto mayor sea la instancia, mayores serán las posibilidades de que la heurística produzca una buena solución) o heurísticas golosas en donde una mala elección sea decisiva en instancias pequeñas pero no tan críticas en instancias grandes. Mientras otras heurísticas, como una búsqueda tabú o un algoritmo GRASP, en un tiempo breve explorarán una gran parte del espacio de búsqueda si éste es pequeño, obteniendo resultados engañosamente buenos en instancias chicas. En definitiva, hay que ser cuidadosos cuando se aplica esta metodología.

Cotas de las soluciones óptimas: Otra posibilidad es encontrar un límite inferior que esté lo más cerca posible de la solución óptima (para un problema de minimización, límite superior para problemas de maximización) y comparar este valor con el encontrado por la heurística. De esta manera obtenemos una cota superior de lo lejos que está el valor heurístico del óptimo. Este límite inferior puede encontrarse mediante relajaciones del problema. La principal contra es que no se sabe qué tan buena es la cota.

Solución óptima conocida: Utilizar instancias con solución óptima conocida, ya sea por provenir de bibliotecas o generadas. Ésto sería ideal, pero no siempre se dispone de estas instancias, y, algunas veces, son instancias sesgadas.

Mejor solución conocida: Si las instancias provienen de una biblioteca, puede ser que no se conozca la solución óptima (o que no haya certeza), pero sí buenas soluciones, muchas veces encontradas por desarrollos de otros investigadores. También se pueden ejecutar algoritmos por mucho tiempo, y de esta manera conseguir soluciones que se espera que sean óptimas o al menos estén muy cerca. Si en el trabajo se desarrollan diferentes métodos, también es posible tomar como referencia la mejor solución encontrada por cualquiera de ellos. Cuando es un problema ya tratado en otros trabajos, es muy importante comparar los resultados y tiempos de ejecución con los desarrollos de los otros trabajos.

Llegó el momento de reportar y analizar los datos recopilados y, ahora, un nuevo problema que se presenta es cómo hacer esto. El análisis debe abordar los objetivos y preguntas planteados que dieron origen a la experimentación y permitir la interpretación, sacar conclusiones evaluar sus implicaciones y hacer recomendaciones. Frecuentemente se toma como métrica el porcentaje de error cometido (valor relativo obtenido al dividir la diferencia entre el

valor óptimo, o cota, y el heurístico por el óptimo, o cota). Los resultados deben resumirse utilizando medidas de tendencia y variabilidad. Muchas veces se agrupan instancias de igual naturaleza y con características similares y se calculan promedios, desviación estándar y el mínimo y máximo de estos valores. La presentación puede ser mediante gráficos y tablas, ya sea mostrando datos estadísticos o particulares.
Llegó el momento de reportar y analizar los datos recopilados y, ahora, un nuevo problema que se presenta es cómo hacer esto. El análisis debe abordar los objetivos y preguntas planteados que dieron origen a la experimentación y permitir la interpretación, sacar conclusiones, evaluar sus implicaciones y hacer recomendaciones. Frecuentemente se toma como métrica el porcentaje de error cometido (valor relativo obtenido al dividir la diferencia entre el valor óptimo, o cota, y el heurístico por el óptimo, o cota). Los resultados deben resumirse utilizando medidas de tendencia y variabilidad. Muchas veces se agrupan instancias de igual naturaleza y con características similares y se calculan promedios, desviación estándar y el mínimo y máximo de estos valores. La presentación puede ser mediante gráficos y tablas, ya sea mostrando datos estadísticos o particulares.

En cuanto a la evaluación del tiempo consumido por un algoritmo heurístico, nos puede interesar comparar, por ejemplo, el tiempo necesario para encontrar la mejor solución obtenida, el tiempo total del algoritmo, el tiempo consumido por cada fase del proceso. Un análisis que suele ser de utilidad es la evolución de la mejor solución encontrada vs tiempo consumido.
En cuanto a la evaluación del tiempo consumido por un algoritmo heurístico, nos puede interesar comparar, por ejemplo, el tiempo necesario para encontrar la mejor solución obtenida, el tiempo total del algoritmo, el tiempo consumido por cada fase del proceso. Un análisis que suele ser de utilidad es la evolución de la mejor solución encontrada vs tiempo consumido.

8. Bibliografía recomendada

- Sección 6.4 de J. Gross and J. Yellen, *Graph theory and its applications*, CRC Press, 1999.

Referencias

[1] F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers and Operations Research*, 13(5):533 – 549, 1986. Applications of Integer Programming.

[2] F. Glover. Tabu search – part i. *ORSA Journal on Computing*, 1(3):190–206, 1989.

[3] F. Glover. Tabu search – part ii. *ORSA Journal on Computing*, 2(1):4–32, 1990.

[4] F. Glover. Tabu search: A tutorial. *Interfaces*, 20:74–94, 08 1990.

[5] L. Pósa. Hamiltonian circuits in random graphs. *Discrete Mathematics*, 14(4):359 – 364, 1976.