

### 3.3. Local Search

*mejorar una solución (factible!)  
p dada*

La búsqueda local es un método heurístico para encontrar una solución factible a un problema. Se basa en obtener una solución inicial,  $s$ , y luego mejorar esa solución iterativamente, tomando la mejor solución de un conjunto de vecinos de  $s$ . El conjunto de los vecinos se determina mediante algún criterio y se usa una función objetivo para comparar las soluciones en la vecindad de  $s$  y discernir cuál es la mejor.

La búsqueda local se puede expresar de la siguiente manera:

Sea  $s \in S$  una solución inicial  $< ?$   
Mientras exista  $s' \in N(s)$  con  $f(s') > f(s)$ :  
 $s \leftarrow s'$

*Esta es la versión  
"quedarse con cualquier  
(el primer) vecino  
mejor".*

Siendo  $N(s)$  la vecindad de  $s$  y  $f$  la función objetivo.

*(la otra versión se queda con el  
mejor vecino) +*

#### 3.3.1. Solución inicial

Para obtener una solución inicial utilizamos Dijkstra. Experimentamos con la funciones objetivo Greedy A y Greedy C descritas en el apartado anterior (ver sección de heurística golosa). No experimentamos con Greedy B, ya que de encontrar una solución usando la sumatoria de los pesos  $w_2$  como función objetivo, podía pasar que no encontremos una solución o que encontremos la solución y en ese caso, no tiene sentido usar búsqueda local, ya que no hay mejor solución posible.

Resultó que el comportamiento experimentado de Greedy A y Greedy C no mostró diferencias notables. Por esa razón decimos usar únicamente Greedy A como solución inicial.

*¿Dónde se ve? ¿Qué resultados obtuvieron?*

*en este caso n! hay que usarla! porque es la solución!*

#### 3.3.2. Definición de la vecindad

En cada iteración definimos la vecindad de  $s$ ,  $N(s)$ , de la siguiente manera: dado el grafo inicial  $G$ , y una solución formada por un camino  $c \in G$ , definimos sus soluciones vecinas como aquellas resultantes de tomar un subcamino  $d_{n_1, n_2} \in c$  entre un par de nodos  $n_1, n_2$  cualesquiera, y reemplazarlo por otro camino  $d_{n_1, n_2}^* \in G$ , de tal forma que el camino  $c^* = c - d_{n_1, n_2} + d_{n_1, n_2}^*$  resultante cumpla:

- $w_1(c^*) < K$  *¿< 5?*
- $w_2(c^*) < w_2(c)$

*S\* = conjunto S = c confuso...*

De la forma descripta, dada una solución  $S$  formada por un camino  $c$  definimos su vecindad como el conjunto  $S^*$  de todos los caminos  $c^*$  posibles.

Para obtener el camino  $d_{n_1, n_2}^*$  utilizamos Dijkstra con la sumatoria de los pesos  $w_2$  como función objetivo (Greedy B). Lo que buscamos es mejorar el subcamino  $d_{n_1, n_2} \in c$

*¿todo a la vez?*

obteniendo otro camino que tenga menor  $\omega_2$ , y dado que usamos Dijkstra para encontrar el camino con  $\omega_2$  mínimo entre los nodos, entonces estamos obteniendo un camino que va a tener igual o menor  $\omega_2$ . En caso de que sea posible hacer el reemplazo, como disminuimos el  $\omega_2$  de una parte del camino y dejamos el resto del camino igual, estamos logrando disminuir el  $\omega_2$  del camino completo.

- ¿qué para si  $d_{n_1, n_2}^*$  que minimiza  $\omega_2$  tiene  $\omega_1 \geq K$ , o  $\text{hace}(C^*) \geq K$ ?  
 - \* esto no es reemplazar por cualquier  $d_{n_1, n_2}^*$  posible, sino solo por uno en particular.

Dada la vecindad  $S^*$ , se elige al vecino usando Steepest descent, con  $\omega_2$  como función objetivo.

??

### 3.3.4. Pseudocódigo

El algoritmo está implementado en la función main:

**Algoritmo 3** *main*(int tipo\_solucionInicial, Graph g, Nodo n1, Nodo n2)

```

1: crearMatrizCaminosMinimos(g) Ahá! → FALTA explicar
2: Solution solucion = obtenerSolucionInicial(tipo_solucionInicial, g, n1, n2)
3: si tipo_solucionInicial ≠ Greedy_A &&  $\omega_1(\text{solucion}) > K$  entonces solucion =
   obtenerSolucionInicial(Greedy_A, g, n1, n2)
4: fin si
5: si  $\omega_1(\text{solucion}) \leq K$  entonces
6:   mientras True hacer
7:     Solution nuevaSolucion = dameMejorVecino(solucion)
8:     si nuevaSolucion == NULL entonces
9:       break
10:    fin si
11:    solucion = nuevaSolucion
12:  fin mientras
13: fin si
    
```

Si el código está en inglés, dejálo acá en inglés:  
 Sean consistentes.

**Algoritmo 4** *obtenerSolucionInicial*(int tipo, Graph g, Nodo n1, Nodo n2)

```

1: si tipo == Greedy_C entonces
2:   return resolverConDijkstra(g, n1, n2, ObjectiveFunctionC)
3: fin si
4: return resolverConDijkstra(g, n1, n2, ObjectiveFunctionA)
    
```

B?

Claramente en lugar de todo esto hay que correr las 3 y quedarse con la mejor (correr B, y si no funciona correr A, y si funciona correr C) da factible

Este camino puede tener un ciclo !!

**Algoritmo 5** *dameMejorVecino*(Solution solucionOriginal)

```

1: Solution mejorSolucion = solucionOriginal
2: vector<int> nodos = nodos(solucionOriginal)
3: para i=0; i<nodos.size(); i++ hacer
4:     para j=i+1; j<nodos.size(); j++ hacer
5:         Solution subSolucion = crearSubSolucionEntre(solucionOriginal, nodos[i],
            nodos[j])
6:         si subSolucion == NULL entonces
7:             continue
8:         fin si
9:         Solution solucion_ij = dameCaminoResueltoEntre(nodos[i], nodos[j])
10:        Solution nuevaSolucion_w2 = w2(solucionOriginal) - w2(subSolucion) +
            w2(solucion_ij)
11:        Solution nuevaSolucion_w1 = w1(solucionOriginal) - w1(subSolucion) +
            w1(solucion_ij)
12:        si nuevaSolucion_w2 < w2(mejorSolucion) && nuevaSolucion_w1 <= K entonces
13:            mejorSolucion = crearSolucionReemplazandoCamino(solucionOriginal,
                solucion_ij)
14:        fin si
15:    fin para
16: fin para
17: return mejorSolucion
    
```

Notar que  $A < B \Leftrightarrow w_2(soluc_{ij}) - w_2(subsol) < B - w_2(sol_{orig})$   
 y  $C \leq K \Leftrightarrow w_1( // ) - w_1( // ) \leq K - w_1(sol_{orig})$

y no hace falta en part. ni  $B = w_2(sol_{orig})$  queda "0"

**Algoritmo 6** *crearSolucionReemplazandoCamino*(Solution orig, Solution sub)

```

1: Solution res
2: res = obtenerCaminoHasta(nodos(sub)[0])
3: res += sub
4: int subSize = size(nodos(sub))
5: res += obtenerCaminoDesde(nodos(sub)[subSize-1])
6: return res
    
```

### 3.3.5. Complejidad

Implementamos el algoritmo en la función *main*. La complejidad del algoritmo resulta la suma de obtener la solución inicial y el ciclo que se usa para mejorarla buscando un mejor vecino en cada iteración. Además, en *main*, inicialmente, se llama a una función *crearMatrizCaminoMinimos* que abordaremos más adelante.

→ No lo ves en el fuente. !

Para obtener la solución inicial, usamos *obtenerSolucionInicial*. Esta función devuelve un camino mínimo entre 2 nodos llamando a *resolverConDijkstra* con alguna función objetivo (Greedy\_A o Greedy\_C). La función *resolverConDijkstra* primero ejecuta Dijkstra para encontrar todos los caminos mínimos entre *n1* y los demás nodos y después hace un *traceback* desde *n2* hasta *n1* para dar con el camino mínimo entre ellos. Como se comprobó en el apartado anterior, la complejidad de Dijkstra es  $O(m \log(n))$

MAL JUSTIFICADO: En cada reemplazo el camino cambia.

(Pero la idea puede funcionar, hay que tratar de charlar!)

## TP2: TÉCNICAS ALGORITMICAS AVANZADAS

¿Por qué no puede reemplazarse más de una vez? Pueden tener ciclos  $\Rightarrow$  un reemplazo puede ocurrir en varias partes. Etc.

y el traceback es  $O(n)$ , por lo que en total la complejidad de obtener SolucionInicial es  $O(m \log(n))$ . Cabe notar que la función objetivo usada en Dijkstra no influye en la complejidad, ya que solo comparar los valores de  $w_1$  y  $w_2$ , por lo que tiene complejidad  $O(1)$ .

(De hecho tal vez pueda probarse que no hay más de  $n$  reemplazos.)

Para mejorar la solución usamos un ciclo y en cada iteración obtenemos el mejor vecino del camino actual usando dameMejorVecino. Ejecutamos el ciclo mientras que hayamos encontrado una mejora al camino actual en la última iteración. Dado que un vecino consiste en reemplazar la porción del camino actual que une 2 nodos  $n_1$  y  $n_2$  por el camino mínimo entre ellos, y que cada vez que se hace el reemplazo se disminuye  $w_2$  del camino actual, se pueden hacer a lo sumo tantos reemplazos como caminos mínimos entre todo par de nodos del grafo existan. La cantidad de caminos mínimos entre todo par de nodos de un grafo es  $n * (n - 1) / 2$ , ya que cada nodo tiene un camino mínimo hacia todos los demás y no a sí mismo. Esto es, si hay  $n$  nodos, el nodo  $n_1$ , tiene un camino mínimo hacia los nodos  $n_2, \dots, n_n$ , el nodo  $n_2$  tiene un camino hacia  $n_3, \dots, n_n$  ya que no se vuelve a contar el camino entre  $n_1$  y  $n_2$  y así hasta  $n_{n-1}$  que tiene un camino hasta  $n_n$ . Luego, la cantidad máxima de iteraciones es  $n * (n - 1) / 2$ .

Para analizar la complejidad de cada iteración hay que analizar la complejidad de dameMejorVecino. Lo primero que hace la función es obtener el arreglo de nodos de la solución pasada por parámetro, que llamamos solucionOriginal. Como el camino de solucionOriginal está representado como una lista de ejes, lo que hace es iterar por todos los ejes y tomar el nodo1 del eje y al final adicionar el nodo2 del último eje. Por ende, tomando  $t$  como la cantidad de nodos en el camino, en cada iteración, por cada eje del camino, se toma un nodo y esto se hace  $t-1$  veces y luego se añade el nodo final. Como  $t$  puede ser a lo sumo  $n$ , entonces la complejidad de esto resulta  $O(n^2)$ . Luego de ejecuta un doble for de  $t * (t - 1) / 2$  iteraciones, que en cada iteración intenta mejorar el mejor camino encontrado hasta el momento, que llamamos mejorSolucion. Inicialmente mejorSolucion es igual a solucionOriginal. Para encontrar una mejor solución en cada iteración hacemos lo siguiente:

¿Dónde?

FALTA justificar (Puede tener ciclos!)

¿De dónde sale el  $n^2$ ?

¿Realmente? ¿hace falta?

- Creamos el subcamino de solucionOriginal entre los nodos  $n_1$  y  $n_2$ . ¿Qué costo tiene?
- Obtenemos un camino mínimo entre los nodos  $n_1$  y  $n_2$ . ¿" " " ?
- Obtenemos un nuevo camino reemplazando el el subcamino entre los nodos  $n_1$  y  $n_2$  por un el camino mínimo entre ellos. ¿" " " ?

Para crear el subcamino de solucionOriginal entre  $n_1$  y  $n_2$  usamos crearSubSolucionEntre. Esta función recibe un par de nodos y un camino y devuelve el subcamino que une a los nodos. Para esto tiene que recorrer a lo sumo todos los nodos del camino, o sea  $t$ , que puede ser a lo sumo  $n$ . idem.



Para obtener el camino mínimo entre los nodos  $n_1$  y  $n_2$  usamos una optimización que es que en la función main, al comienzo, ejecutamos crearMatrizCaminosMinimos que crea una matriz de caminos mínimos entre todo par de nodos del grafo. Generamos esta matriz ejecutando Dijkstra para todos los nodos usando como función objetivo  $w_2$ . Usamos esta matriz para obtener en  $O(1)$  el camino mínimo entre los nodos  $n_1$  y  $n_2$ .

no lo veo

¿el camino o la distancia?

\* Por ej. reempl. 7..30, luego 12..40, luego 7..30 otra vez.

Como crearMatrizCaminosMinimos ejecuta un Dijkstra por cada nodo, su complejidad es  $O(n * (m \log n))$ .

*Se puede hacer más rápido si el grafo es denso.*

Para obtener el nuevo camino reemplazando el subcamino entre  $n_1$  y  $n_2$  por el camino mínimo entre ellos, usamos crearSolucionReemplazandoCamino. Lo que hacemos es generar una nueva solución que es la concatenación de 3 caminos: el camino mínimo entre  $n_1$  y  $n_2$  y los dos pedazos del camino original sin el subcamino que unía  $n_1$  y  $n_2$ . Para crear el camino, hay que recorrer el solucionOriginal y añadir todos los nodos hasta  $n_1$  y luego añadir todos los nodos del camino mínimo y al final añadir todos los nodos desde  $n_2$  hasta el final de solucionOriginal. Por ende es como iterar sobre el nuevo camino que a lo sumo puede tener  $n$  nodos y por ende la complejidad resulta  $O(n)$ .

*FALTA demostrar (! puede tener ciclos!)*

Entonces, resulta ser que encontrar el mejor vecino, consiste en ejecutar el doble for de algo que tiene complejidad  $O(n + 1 + n)$ , o sea  $O(n)$ , entonces en total es  $O(n^2 * n) = O(n^3)$ .

*Se puede hacer en  $O(1)$  !*

Como habíamos explicado que dameMejorVecino se ejecuta en un ciclo de hasta  $n * (n-1)/2$  iteraciones, o sea  $O(n^2)$ , y entonces resulta que la complejidad de encontrar el mejor vecino es  $O(n^2 * n^3) = O(n^5)$ .

Finalmente la complejidad total resulta ser la suma de las complejidades de crearMatrizCaminosMinimos, 2 veces obtenerSolucionInicial y  $n^2$  veces dameMejorVecino. Es decir,  $O(n * (m \log n) + 2 * (m \log n) + n^5) = O(n^5)$ .

*¿cuál es el costo por iteración?*

### 3.3.6. Familias malas

Si elegimos como solución inicial a la heurística Greedy B en la sección previa, se ha expuesto que puede fallar en el intento de dar una solución factible, aún existiendo una. El Greedy A siempre encuentra una solución factible de existir ésta.

*¿Por qué?*

Veamos que tomando nuestra heurística de búsqueda local como solución inicial, puede quedar arbitrariamente lejos de la solución óptima.

*¿Quién toma qué? No entiendo !!*

Presentamos la siguiente familia de grafos:

*FALTA justificar*

### 3.4. GRASP

La metaheurística Grasp es una combinación <sup>de</sup> entre una heurística golosa aleatorizada y un procedimiento de búsqueda local. Sea  $S$  el conjunto de soluciones iniciales, el algoritmo que se usa es el siguiente:

Mientras no se alcance el criterio de terminación:

- Obtener  $s \in S$  mediante una heurística golosa aleatorizada.
- Mejorar  $s$  mediante búsqueda local.
- Recordar la mejor solución obtenida hasta el momento.

#### 3.4.1. Solución inicial

Usamos Dijkstra como nuestra heurística golosa, pero modificado para agregarle aleatoriedad. El factor aleatorio consiste en que en cada iteración de Dijkstra, en vez de tomar el nodo no visitado que minimiza la función objetivo, tomamos uno de entre los  $\beta$  menores. ¿al azar?

Dado la aleatoriedad de la solución inicial, es posible que ésta no sea factible. En caso de obtener una solución de ese tipo, no ejecutamos búsqueda local.

Para Dijkstra usamos la tres función objetivo Greedy, A definida en el apartado Greedy.

#### 3.4.2. Criterio de terminación

Usamos 3 criterios de terminación distintos al mismo tiempo. De alcanzarse alguno de los criterios, se termina la ejecución del algoritmo.

Los criterios que usamos son:

- Cantidad máxima de iteraciones.
- Cantidad máxima de iteraciones sin haber encontrado mejoras.
- Cantidad máxima de iteraciones sin haber encontrado una solución inicial factible.

Parametrizamos estos valores usando  $n$ . Elegimos usar  $n$  para la cantidad máxima de iteraciones sin haber encontrado mejoras y sin haber encontrado una solución factible y  $n \cdot \log(n)$  para la cantidad máxima de iteraciones. Esta elección para las constantes fue a partir de experimentar con distintos valores basados en  $n$ , que nos pareció un buen parámetro para definirlos, ya que generalmente mientras más nodos tenga el grafo,

<sup>3</sup>: Luego de experimentar con distintos valores, encontramos que  $\beta=10$  era un valor que presentaba suficiente aleatoriedad.

con "restart": la BL es arrancada desde múltiples inicios (cf. Tabu search)  
me parece que esto no tiene sentido / buena def. inicial...

Notem que esto no es exactamente "goloso randomized" porque no va construyendo una solución, sino varias al mismo tiempo (que interactúan entre sí...).

¿Por qué usan " $\beta$ " y no " $\alpha$ " (de GRASP)? 

¡OJO! No resetean estos contadores. O sea, no reinician iteraciones consecutivas.

Ok. y bien por decirlo acá.  
Pero ¿qué quiere decir "eficiente" (aleatoriedad)?  
¿cómo lo determinaron?  
¿qué relación hay entre eso y la calidad del resultado obtenido por GRASP? (que es lo que más importa...)

### 3.4.5. Complejidad

El algoritmo se ejecuta en un ciclo hasta cumplir con alguno de los criterios de terminación. Dado que el criterio de mayor valor es la cantidad de iteraciones totales (*iteracionesMax*), tomamos ese valor como cota para calcular la complejidad. Cada iteración del ciclo es casi idéntica a la ejecución de búsqueda local. La única diferencia es cómo se obtiene la solución inicial. Para encontrar la solución inicial se usa *resolverConDijkstraAleatorio*. Esta función, en vez de tomar el nodo no visitado con  $w_2$  mínimo, toma uno entre los *beta* menores. Pero resulta que la complejidad no se altera con este cambio, ya que Dijkstra itera sobre todos los nodos de cualquier manera y lo único que cambia es el orden en que se toma el nodo no visitado.

¿Cómo está implementado?

Vale hacer una aclaración: ~~que es que~~ como se usa una cola con prioridad para los nodos no visitados en Dijkstra, para sacar uno entre los *beta* menores, hay que ~~remover~~ <sup>quitar</sup> los primeros *beta* nodos de la cola y luego volver a agregar todos menos uno que es con el que nos quedamos. Como remover y agregar de la cola con prioridad toma  $O(\log(n))$ , entonces para remover y agregar los *beta* nodos se toma  $O((\beta + \beta - 1) * \log(n))$ , pero como *beta* es constante entonces la complejidad resulta  $O(\log(n))$ .

¿es obligatorio? este "n" es el tamaño de la cola, o/s

Por lo tanto la complejidad de *resolverConDijkstraAleatorio* no resulta diferente a *resolverConDijkstra*, usada en búsqueda local y por ende la complejidad de cada iteración resulta igual a la complejidad de una ejecución de búsqueda local, o sea tiene complejidad  $O(n^5)$ .

no resulta.

FALTAN justificar los  $O(n^5)$  (en BL)

Luego la complejidad total es  $O(n^5 * \text{iteracionesMax}) = O(n^5 * n * \log(n)) = O(n^6 * \log(n))$ .

Es bueno/útil expresar la complejidad como  $O(\text{cant. iter} \cdot \text{costo por iter})$

### 3.4.6. Experimentación

A continuación presentamos los resultados de la experimentación del tiempo de ejecución de la metaheurística GRASP.

⊗ Es tan "constante" como  $n$ . Es mejor/más útil/menos engorroso describir la complejidad (de ese par) como  $O(\beta \log n)$ .  $\Rightarrow$  ¿Cómo queda "Dijkstra aleatorizado"?