DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

TP 2

TSP

12 de septiembre de 2021 — Algoritmos y Estructuras de Datos III

Grupo 04

Integrante	LU	Correo electrónico
Damburiarena, Gabriel	889/19	gabriel.damburiarena@gmail.com
Guastella, Mariano	888/19	marianoguastella@gmail.com
Silva, Ignacio Tomas	410/19	ignaciotomas.silva622@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

1. Introducción

El problema del viajante de comercio (o TSP por sus siglas en inglés) es un problema que se resuelve con optimización combinatoria que consiste en encontrar un circuito hamiltoniano de costo mínimo, es decir, un circuito mínimo que visite todos los vertices exactamente una vez. Se utiliza para modelar situaciones de la vida real para optimizar una relación de efectividad-costo, como el camino que toma un camión que distribuye mercadería o un repartidor de pizza muy sofisticado. Formalmente, dado un grafo completo G = (V, X) donde cada arista $(i, j) \in X$ tiene asociado un costo c_{ij}, el costo de un camino p se define como la suma de los costos de sus aristas y se nota $c_p = \sum_{(i,j)\in p} c_{ij}$.
A continuación daremos unos ejemplos con sus respectivas soluciones:

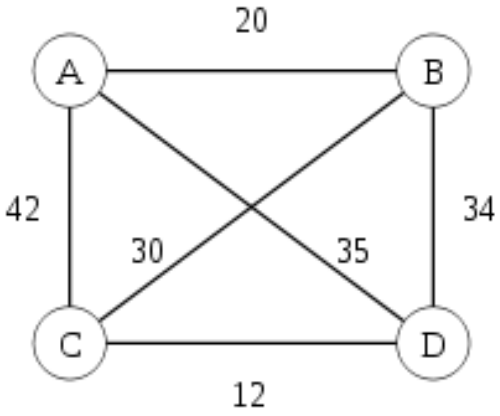

Figura 1: Grafo A

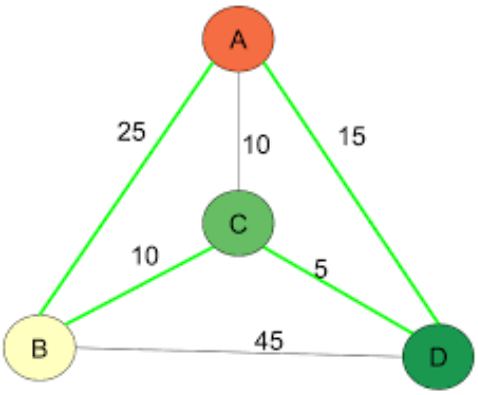

Figura 2: Grafo B

Para el grafo A el ciclo de menor costo es el que va de [A, B, C, D] o alguno equivalente y cuesta 97. Por otro lado, para el grafo B el ciclo de menor costo es el [A, D, C, B] o cualquiera equivalente y cuesta 55.

Aplicaciones en la vida real: Un ejemplo en la vida real que se puede modelar utilizando el TSP es el correo. Las empresas de distribución de bienes tienen que hacerlo de la mejor manera posible para reducir los costos ya que no se le suele cargar al cliente todo el costo del envío y obviamente la empresa quiere tener el mayor rédito posible. Si la empresa es capaz de ahorrarse 2.000 pesos en combustible por kilometro y realiza diariamente 1.000km estariamos hablando de 2 millones de pesos diarios.
Otro ejemplo es para la perforación de las placas de circuito donde se tienen que ir haciendo agujeros de distinto diámetro y para eso hay que cambiar la punta del taladro. Esto es muy costoso en cuanto a tiempo por lo que se busca hacer los agujeros del mismo tamaño lo más rapido posible. Entonces se pueden ver a los agujeros como ciudades y la distancia entre ellas es lo que se mueve el taladro entre los puntos para agujerear. La meta es que el tiempo empleado sea mínimo por cada cabezal del taladro.

El objetivo de este trabajo práctico es utilizar 3 heuristicas y 1 metaheuristicas para resolver el problema y evaluar su eficacia en diversas instancias de problemas ya que este es un problema dificil de resolver para la computación con algoritmos exactos ya que demanda mucho tiempo de cómputo. En primer lugar, utilizamos la heurística constructiva golosa Vecino más cercano. Luego, utilizaremos la metodología del árbol generador mínimo dando lugar a la heurística del mismo nombre. También veremos Búsqueda local que busca una solución cercana a una inicial con el fin de mejorarla. Finalmente, la metaheuristica Tabú que se basa en Búsqueda local pero no se quedan con el primer resultado sino que intenta encontrar una solución mejor mediante distintas técnicas.

El trabajo va a estar ordenado de la siguiente manera: primero en la Sección 2 se define el método Vecino más cercano y se analiza su complejidad. Más tarde, en la Sección 3 se explica el algoritmo de Arbol generador mínimo. Luego, se introducen Búsqueda Local y Tabú la Sección 4 y 5 respectivamente junto con un análisis de complejidad. Finalmente, en la Sección 6 se presentan los experimentos computacionales, y las conclusiones finales se encuentran en la Sección 7.

2. Golosa - Vecino más cercano

Es la heurísitca más intuitiva para este problema: En cada paso elegimos como siguiente lugar a vistar el que, entre los que todavía no visitados, se encuentre más cerca del vertice actual.
En primer lugar creamos la solución vacia e inicializamos el nodo actual como un número aleatorio entre 0 y el máximo vértice y lo agregamos a la solución. Luego, por cada vértice definimos un máximo mayor a todo peso de toda arista e iteramos por cada arista quedandonos con la mínima. Finalmente, la agregamos atrás y pasamos al siguiente nodo, repetimos hasta terminar de explorar el grafo y devolvemos la solución. El algoritmo es el siguiente:

---

vecinoCercano(max, grafo)

```
1:  solucion = []
2:  actual = randomNumber()
3:  agregarAtras(solucion, actual)
4:  for i = 0 hasta |grafo| - 1 do
5:      min = max + 1
6:      siguiente = 0
7:      for j = 0 hasta |grafo| - 1 do
8:          if j ∉ solucion y grafo[actual][j] < min y actual ≠ j then
9:              min = grafo[actual][j]
10:             siguiente = j
11:         end if
12:     end for
13:     agregarAtras(solucion, siguiente)
14:     actual = siguiente
15: end for
16: return solucion
```

La complejidad va a ser $\mathcal{O}(V^2)$ ya que por cada vértice se mira todo otro vértice. El resto de operaciones son $\mathcal{O}(1)$ con la delicadeza de que agregarAtras tiene que serlo también, por lo que nosotros usamos un vector, que tiene este método y cuesta $\mathcal{O}(1)$ pero se podría también implementar con una lista enlazada con referencia al último nodo.

El peor caso para esta heurística es el siguiente:

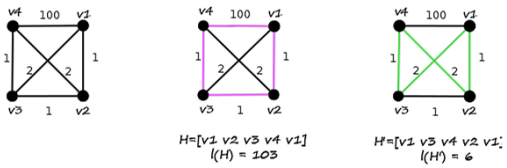

Figura 3: El violeta es la respuesta de la golosa y el verde el óptimo

Es importante notar que este peor caso puede ser tan malo como nosotros querramos ya que le podemos poner cualquier peso a la última arista que, obligadamente, tenemos que poner en nuestro ciclo para cerrarlo.

3. Árbol generador mínimo

La idea de esta heurísitca es usar el AGM. En un esquema general lo primero que vamos a hacer es obtener el AGM del grafo. Luego ejecutamos DFS sobre nuestro AGM y nos guardamos el recorrido. Finalmente armamos el ciclo salteando los nodos ya visitados.
Nuestra implementación obtiene el AGM a través del algoritmo de Prim empezando por un nodo aleatorio y buscando siempre el mínimo camino al próximo nodo. Por cada nodo agregado nos guardamos su padre. Al finalizar nos queda el árbol generador mínimo pero cada nodo lleva a su antecesor, lo cual para posteriormente aplicar DFS no es cómodo, por lo que invertimos el árbol para que nos queden los hijos de cada nodo. Luego, aplicamos DFS modificado de tal forma que se guarda el recorrido que va haciendo, gracias a una pila y nos queda un recorrido DFS del árbol. Lo único que resta por hacer es agregar al final el nodo raíz para hacer el ciclo. El algoritmo es el siguiente:

heuriticaAGM(grafo)

```
1: raiz = randomNumber()
2: AGMP = primAGM(grafo, raiz)
3: AGMH = invertir(AGMP)
4: recorridoDFS = DFS(AGMH, raiz)
5: agregarAtras(recorridoDFS, recorridoDFS[0])
6: return recorridoDFS
```

La complejidad va a ser $\mathcal{O}(V^2)$ ya que la implementación de Prim que usamos es de $\mathcal{O}(V^2)$ porque por cada vértice se miran todos los otros, invertir el AGM cuesta $\mathcal{O}(V)$ ya que se pasa una sola vez por cada uno, DFS cuesta $\mathcal{O}(V)$ porque se pasa una vez por cada uno nuevamente y agregar al final cuesta $\mathcal{O}(1)$. Notar que podria costar $\mathcal{O}(E*log(V))$ si la implementación fuese con una cola de prioridad y el grafo con una lista de adyacencias.

El peor caso para esta heurística es el siguiente:

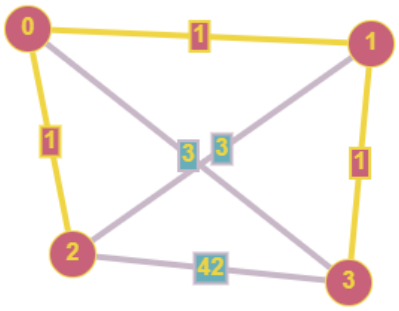

Figura 4: AGM de este grafo enraizado en 0

Notar que si queremos cerrar el ciclo tenemos que quedarnos con la arista más pesada (2, 3), y esta puede ser tan pesada como se nos ocurra. En este caso el costo del ciclo es 45. Una respuesta mejor es [0,2,1,3,0] cuyo costo es 8.

4. Búsqueda Local

Dada una solución de una instancia de nuestro problema, quizás nos convenga buscar soluciones cercanas a nuestra solución. De esto se trata búsqueda local ya que busca encontrar un óptimo local a partir de la solución inicial. Es una técnica iterativa que explora un el conjunto de soluciones, moviendose de una solución a otra con la esperanza de encontrar una mejor, ya sea un óptimo global idealmente o uno local. Lo que vamos a hacer nosostros es aplicar 2-opt que consiste en intercambiar 2 aristas y ver si esto mejora el resultado final.

Nuestra implementación de Búsqueda Local recibe una solución, que en particular la calculamos con AGM, pero bien podría ser de otra heurística. Luego se guarda esa solución como el mejor circuito y busca intercambiando todas las aristas de a 2, por eso 2-opt, alguna que mejore el peso del ciclo. Si un intercambio mejora el ciclo, nos lo guardamos y avisamos que encontramos una mejora para volver a buscar otra variación que mejore aún más. La función swap(), que hace el 2-opt, intercambia 2 vértices recorriendo el vector de los mismos sin cambiar nada hasta encontrar el primero, invierte el sentido hasta el segundo y después deja todo como estaba. El algoritmo de Búsqueda Local es el siguiente:

```
busquedaLocal(grafo, circuito)
 1: mejorCircuito = circuito
 2: mejora = true
 3: while mejora do
 4:     mejora = false
 5:     for i = 0 hasta |mejorCircuito| - 2 do
 6:         for j = i + 1 hasta |mejorCircuito| - 1 do
 7:             nuevoCircuito = Swap(circuito, i, j)
 8:             if sumaCiclo(nuevoCircuito, grafo) < sumaCiclo(mejorCircuito, grafo) then
 9:                 mejorCircuito = nuevoCircuito
10:                 mejora = true
11:             end if
12:         end for
13:     end for
14: end while
15: return mejorCircuito
```

El análisis de la complejidad empieza por $\mathcal{O}(V^2)$, ya que eso cuesta el cálculo de la solución inicial. Luego, hay que sumarle $\mathcal{O}(V^3)$ que es lo que cuesta el ciclo doble, ya que itera por cada vértice, lo intercambia con todo otro vértice y, además recorre todos los vértices en la función swap() para mantener el ciclo. También suma el ciclo para saber si el coste mejora y esto es $\mathcal{O}(V)$ pero $\mathcal{O}(V + V)$ es $\mathcal{O}(V)$. Todas las otras operaciones son de costo constante. La complejidad final es $\mathcal{O}(V^2 + V^3)$ que es igual a $\mathcal{O}(V^3)$.

El peor caso es dificil de encontrar ya que podríamos encontrar un óptimo local a la primera e iterar sin sentido. Pero tampoco cambia demasiado la complejidad con el caso donde iteramos varias veces, ya que terminamos gastando más tiempo buscando una solución cuando se mejora. Además es muy controlable el cuanto queremos iterar, pero eso lo vamos a ver más en Tabú.

5. Tabú

Tabú es una metaheurística que lleva a la herística de búsqueda local a explorar un espacio de soluciones más amplio con el objetivo de no quedarse atascado en un óptimo local. En su funcionamiento es similar a búsqueda local, ya que el procedimiento iterativamente se mueve de una solución a otra. La innovación es el uso de memoria para evitar repetir una secuencia de soluciones ya explorada, ya que podrian generarse ciclos porque Tabú se permite mover a soluciones que empeoran el costo del ciclo con la esperanza de encontrar alguna solución mejor a la inicial.

Nuestra implementación de esta metaheurística también recibe una solución inicial, que bien podría ser calculada con cualquier heurística pero nosotros vamos a usar AGM. Luego, la guarda como la mejor respuesta hasta el momento, inicializa una memoria y el contador de intentos. Mientras no superemos la cantidad de intentos vamos a intentar mejorar la solución que tenemos. Los pasos del ciclo son:

- Primero vamos a obtener la subvecindad y esto requiere una explicación. La subvecindad es un subconjunto de la vecindad que la almacenamos en un vector mientras que a los vecinos almacenamos como los pares de aristas a intercambiar. Hacemos el 2opt y guardamos todos esos vecinos en la vecindad.
- Luego los mezclamos y tomamos un porcentaje de ellos, hacemos esos intercambios al ciclo que tenemos, lo guardamos en la subvecindad y al finalizar lo devolvemos. Volviendo al tronco principal, buscamos el mejor de todos los vecinos sin que este en la memoria y lo guardamos. Si este es un ciclo vacio, significa que no encontramos una variante que no este en memoria y nos quedamos con la que teniamos antes. Si la memoria se llenó, vamos a deshacernos del primer elemento que guardamos en ella para poder guardar este nuevo ciclo.
- Finalmente, si el coste del ciclo nuevo es menor al que teniamos de referencia, nos lo guardamos, reiniciamos el contador y seguimos. En caso contrario solo seguimos. Cuando se llegue al máximo de intentos, devolvemos el mejor ciclo.

El algoritmo es el siguiente:

```
tabuSearch(grafo, memoriaMax, porcentaje, cantIter)
 1: ciclo = heurisitcaAGM(grafo)
 2: mejorCiclo = ciclo
 3: memoria = []
 4: intentos = 0
 5: while intentos < cantIter do
 6:     vecinos = obtenerSubvecindad(ciclo, porcentaje)
 7:     ciclo = obtenerMejor(vecinos, memoria, grafo)
 8:     if ciclo = [] then
 9:         ciclo = mejorCiclo
10:     end if
11:     if tamaño(memoria) = memoriaMax then
12:         sacarPrimero(memoria)
13:     end if
14:     agregarAtras(memoria, ciclo)
15:     if sumaCiclo(ciclo, grafo) < sumaCiclo(mejorCiclo, grafo) then
16:         mejorCiclo = ciclo
17:         intentos = 0
18:     end if
19:     intentos = intentos + 1
20: end while
21: return mejorCiclo
```

```
obtenerSubVecindad(ciclo, porcentaje)
 1: subVecindad = [[]]
 2: vecindad = [(,)]
 3: for i = 0 hasta |ciclo| - 2 do
 4:     for j = 0 hasta |ciclo| - 1 do
 5:         vecino = (i, j)
 6:         agregarAtras(vecindad, vecino)
 7:     end for
 8: end for
 9: mezclar(vecinos)
10: cantidadDeVecinos = |vecindad|*porcentaje/100
11: for i = 0 hasta cantidadDeVecinos do
12:     vecino = vecindad[i]
13:     agregarAtras(subvecindad, swap(ciclo, primero(vecino), segundo(vecino)))
14: end for
15: return subVecindad
```

Notar que tanto la memoria como el porcentaje, la cantidad de iteraciones, la heurística de la solución inicial y hasta el tipo de memoria que usamos se pueden modificar. Nosotros vamos a utilizar AGM porque consideramos que funciona mejor en la mayoría de los casos, según el criterio que formamos viendo documentación, como en la conclusión 18. Hace falta remarcar que no siempre es la mejor y se debe elegir cuidadosamente según el caso. Por ejemplo en la comparación 8 utilizamos otra instancia que deja mejor parado a VMC. El resto de parámetros los vamos a ir variando en las experimentaciones y nos quedaremos con las mejores combinaciones. Para el cambio del tipo de memoria hicimos otra implementación que se guarda en memoria el inverso al cambio de aristas del ciclo nuevo, con tal de no intentar volver a hacer ese cambio y terminar en algo que ya visitamos. Esto es un poco más restrictivo pero nos permite ampliar nuestra búsqueda.

El análisis de la complejidad empieza por $\mathcal{O}(V^2)$, ya que eso cuesta el cálculo de la solución inicial (AGM). Metiendonos en las funciones, obtener la subvecindad cuesta $\mathcal{O}(V * V^2 * P)$ $(Porcentaje : (0 < P \leq 1))$ crear los pares de vecinos ya que iteramos por todos los vértices dos veces de manera anidada. Todas las otras operaciones son constantes. El mezclar que usamos es lineal. Por ultimo, iteramos por la cantidad de vecinos y agregamos ese swap lo cual cuesta $\mathcal{O}(V * V^2 * P)$ ya que la cantidad de vecinos es $\mathcal{O}(V^2)$ si el porcentaje $P = 1$ y swap cuesta $\mathcal{O}(V)$. Volviendo a la función principal, obtener el mejor es $\mathcal{O}(V^3 * P * T)$ siendo T el tamaño de la memoria ya que, por

cada vecino sumamos sus aristas, revisamos si esta en la memoria ese ciclo y si es mejor que el que teníamos guardado, nos lo quedamos. Luego, lo que se ejecuta adentro del ciclo 'while' es $\mathcal{O}(V^3 * P * T)$ y esto se repite hasta un máximo de iteraciones (I). Si P es suficientemente chico puede que $V^3 * P < V^2$ por lo tanto, la complejidad del algoritmo dependerá del máximo entre éstos dos valores. Finalmente, la complejidad final sería $\mathcal{O}(max(I * V^3 * P * T, I * V^2 * P))$.

El peor caso sería tomar el 100 % de la vecindad, o sea $P = 100\,\%$, lo cual hace que la complejidad de obtener la subvecindad sea $\mathcal{O}(V^3)$. Además, tener una memoria muy grande agranda el T, por lo que eso también suma. Finalmente, la cantidad de iteraciones mayor a V complicaría más aún las cosas. Si estos parámetros son del orden de V la complejidad quedaría $\mathcal{O}(V^5)$, por lo que hay que tener mucho cuidado a la hora de seleccionar los valores.

6. Experimentación

6.1. Instancias

Para experimentar usamos las siguientes instanciasde tsplibrary obtenidas en:
http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/index.html

- **ulysses16:** Odyssey of Ulysses (Groetschel/Padberg) Grafo completo euclideano de 16 vértices.
- **eil51:** 51-city problem (Christofides/Eilon) Grafo completo euclideano de 51 vértices.
- **pr76:** 76-city problem (Padberg/Rinaldi) Grafo completo euclideano de 76 vértices.
- **gr202:** Europe-Subproblem of 666-city TSP (Groetschel) Grafo completo euclideano de 202 vértices.
- **ulysses22:** Odyssey of Ulysses (Groetschel and Padberg) Grafo completo euclideano de 22 vértices.

Lo que vamos a hacer es elegir una instancia y correrla entre 40 - 50 veces ya que el factor de aleatoriedad va a cambiar los resultados. El tiempo se va a medir en milisegundos.

6.2. Experimento 1: Golosa - Vecino más cercano

Para el algoritmo goloso, esperamos una performance normal en cuanto a tiempo de ejecución ya que el algoritmo toma una decision en tiempo constante por cada vertice que visita, sin importar con qué vertice comienza. Además, al tener un componente aleatorio esperamos que los gaps sean erráticos (dentro de limites razonables). Corremos la instancia del dataset ulysses16 40 veces y observamos los resultados.

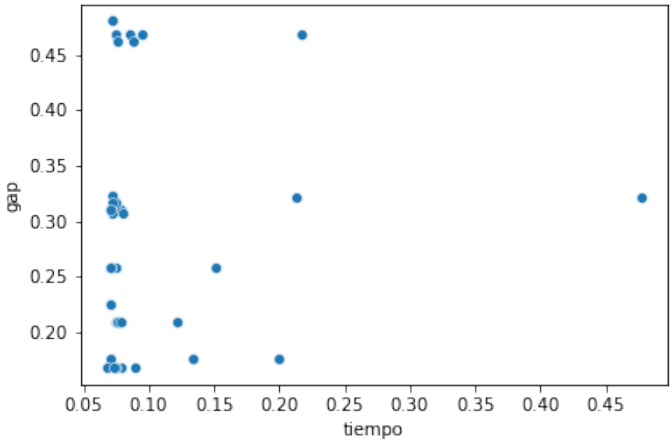

Figura 5: Scatter plot de 40 ejecuciones de Vecino más cercano. En el eje 'x' se mide el tiempo en milisegundos y en el eje 'y' se mide el gap, que es la proporción sobresaliente del resultado encontrado en comparación con el óptimo conocido.

La figura parece acompañar lo esperado, el gap con el óptimo oscila entre 0 y 0.45. Con el tiempo de ejecución hay algo de 'ruido' que suponemos que se debe al procesador en si y no al algoritmo en cuestión, ya que se puede ver que la

mayoría de los resultados estan agrupados. Teniendo en cuenta que el algoritmo no implementa mejoras a la solución, esta resulta ser bastante aceptable en varios casos. En algunos casos da resultados no tan deseados, probablemente debido a particularidades del dataset.

6.3. Experimento 2: Heurística de Árbol generador mínimo

Para el experimento con el AGM repetimos la instancia ulysses16, esta vez por la naturaleza del algoritmo esperamos tiempos de ejecución en el mismo orden y alguna mejora en las diferencias con el circuito óptimo con respecto a Vecino más cercano. Esto se debe a que en los estudios que estuvimos revisando, AGM al ser más refinado, suele correrse de manera más óptima en promedio y no caer en tan malas decisiones como VCM.

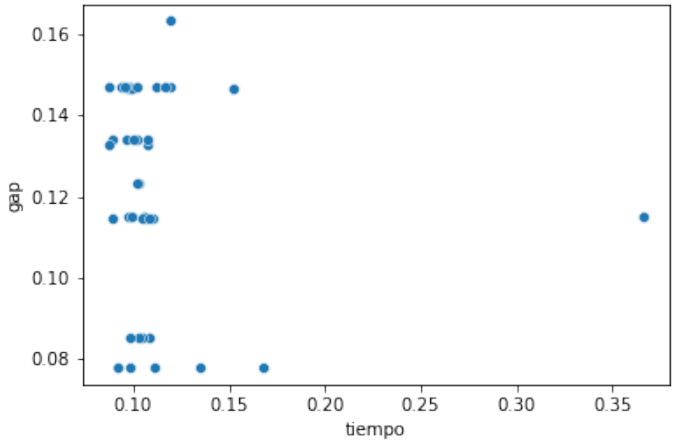

Figura 6: Scatter plot de 40 ejecuciones de Arbol generador mínimo. En el eje 'x' se mide el tiempo en milisegundos y en el eje 'y' se mide el gap, que es la proporción sobresaliente del resultado encontrado en comparación con el óptimo conocido.

Los resultados tambien apoyan las hipotesis teóricas. Hay una mejora notable en los gaps de la solución y, quitando el outlier, un comportamiento similar en el tiempo de ejecucion. Esto entendemos que sucede justamente porque el árbol generador mínimo construye una mejor solución en general que VMC. El aumento en tiempo de ejecución es mínimo.

6.4. Experimento 3: Búsqueda Local

En el caso de búsqueda local esperamos un salto considerable en la calidad de la solución y también un aumento importante en el tiempo de ejecución, tratandose además de una instancia relativamente chica, el algoritmo deberia ser capaz de encontrar el óptimo alguna vez. Debería ser siempre mejor que AGM ya que actúa optimizando una solución de este. Usamos la instancia ulysses16.

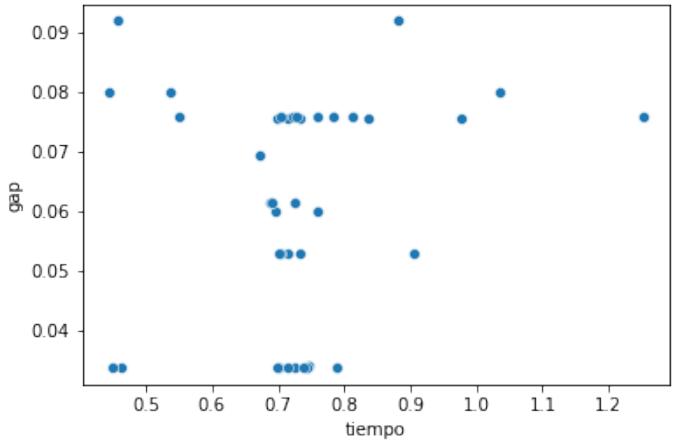

Figura 7: Scatter plot de Búsqueda local con 40 iteraciones. En el eje 'x' se mide el tiempo en milisegundos y en el eje 'y' se mide el gap, que es la proporción sobresaliente del resultado encontrado en comparación con el óptimo conocido.

Como se ve en el gráfico, hay un aumento importante en tiempo de ejecución y pero también en calidad con respecto a los dos algoritmos anteriores. Hay también algunos casos donde logra encontrar el óptimo absoluto, pero los consideramos casos fortuitos. El techo del gap arranca en el piso del gap de AGM, pero el tiempo de procesamiento se cuadruplica en promedio en esta instancia.

6.5. Vecino más cercano vs AGM vs Búsqueda Local

Ahora vamos a comparar los algoritmos entre sí con un dataset más grande para ver si se repiten los resultados vistos en los casos aislados previamente expuestos. Después de probar varios datasets distintos, en los cuales se mantenía la relación antes vista, encontramos un resultado interesante que vamos a mostrar a continuación, el dataset eil51. ¿Qué sucede si AGM nos da en promedio peores soluciones que el Vecino más cercano?, vamos a verlo.

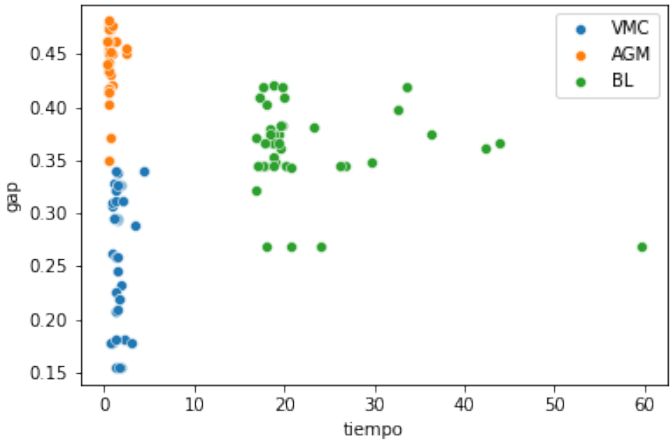

Figura 8: Comparativa entre las 3 heurísticas(VMC = vecino más cercano, AGM = árbol generador mínimo y BL = búsqueda local). En el eje 'x' se mide el tiempo en milisegundos y en el eje 'y' se mide el gap, que es la proporción sobresaliente del resultado encontrado en comparación con el óptimo conocido.

En el experimento sucede que el algoritmo del Vecino más cercano da soluciones de mejor calidad que las del AGM ya que, la instancia en cuestión, es mala para AGM. Y, como Búsqueda Local opera mejorando las soluciones del AGM, estas terminan siendo mejores que las del AGM pero peores que las del vecino más cercano. Este experimento abre la posibilidad de evaluar, previo a usar Búsqueda Local, que heurística se adapta mejor al dataset para luego mejorar sus soluciones con Búsqueda Local.

6.6. Experimento 4: Búsqueda Tabú

Vamos a mirar a continuación que ocurre cuando variamos el porcentaje de la vecindad en ambos tipos de memoria. Estamos expectantes por como afecta el tipo de memoria en Búsqueda Tabú, ya que lo esperado es que la memoria de ciclos sea más lenta y la de aristas más rápida. Esto es debido a que almacena menos memoria que la de ciclos. Pero es importante saber que al explorar más a lo 'ancho' con la memoria de aristas podria encontrar mejores soluciones pero tardar más. Utilizamos la instancia pr76. Importante denotar que utilizaremos AGM para Búsqueda Tabú (BT). Empezamos estudiando la variación del porcentaje de la subvecindad.

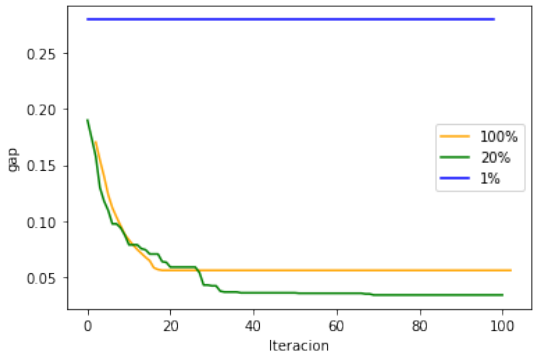

Figura 9: Variación del porcentaje en memoria de ciclos. El gap esta en el eje 'y' y las iteraciones en el eje 'x'

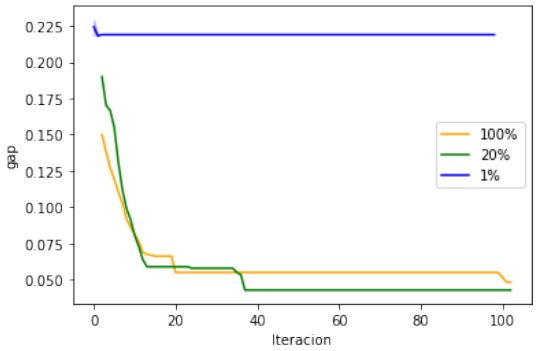

Figura 10: Variación del porcentaje en memoria de aristas. El gap esta en el eje 'y' y las iteraciones en el eje 'x'

Interesantemente, la calidad de la solución aumenta de manera más escalonada en la memoria de ciclos que en la de aristas, en la cual es mas vertiginosa, ya que desciende mucho más rápido. Esto entendemos que sucede ya que la memoria de ciclos va encontrando mejoras pequeñas de a poco mientras que la memoria de aristas mejora de golpe con un intercambio de aristas cada varias exploraciones. También vamos a concluir que un 20 % es el porcentaje óptimo de los que evaluamos ya que al usar el 100 % de la memoria no mejora la calidad y, obviamente, con un 1 % no se logra mucho.

A continuación vamos a probar variando el tamaño de la memoria. Esperamos la memoria de ciclos se vea más afectada mientras menos memoria que la de aristas, ya que esta última requiere menos memoria para su ejecución. Utilizaremos la instancia pr76.

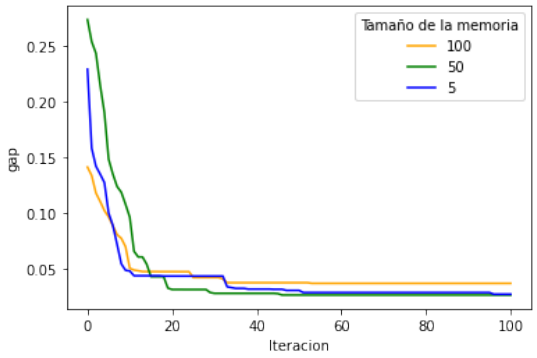

Figura 11: Variación del máximo memoria de ciclos. El gap esta en el eje 'y' y las iteraciones en el eje 'x'

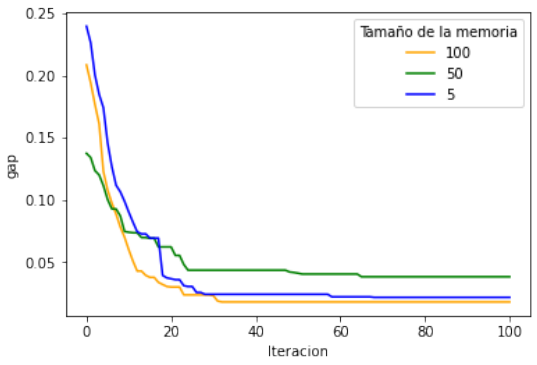

Figura 12: Variación del máximo en memoria de aristas. El gap esta en el eje 'y' y las iteraciones en el eje 'x'

Los resultados muestran que la cantidad de memoria no afecta al resultado final en una gran medida en ambos algoritmos. Las diferencias en gap que se observan se deben a la aleatoriedad del algoritmo mas que nada. De todas formas, concluimos que tener una memoria chica es lo más adecuado, ya que una cantidad de memoria más chica reduciría la complejidad del algoritmo y mejoraría su tiempo.

A continuación vamos a evaluar el gap variando la cantidad máxima de iteraciones y la cantidad de memoria máxima. Viendo resultados anteriores esperamos que el aumento en cantidad de iteraciones mejore la solucion encontrada independientemente de la cantidad de memoria empleada. Usaremos la instancia ulysses16.

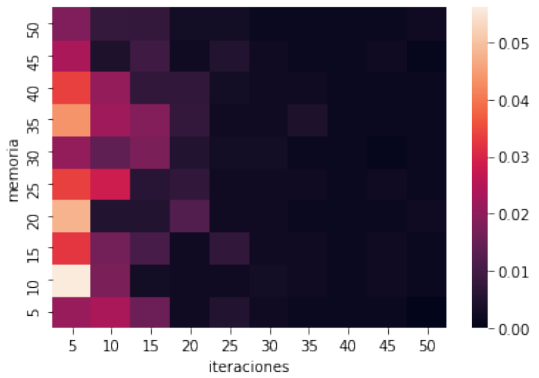

Figura 13: Heatmap con el gap (en el eje 'y') según memoria de ciclos con 50 iteraciones (en el eje 'x')

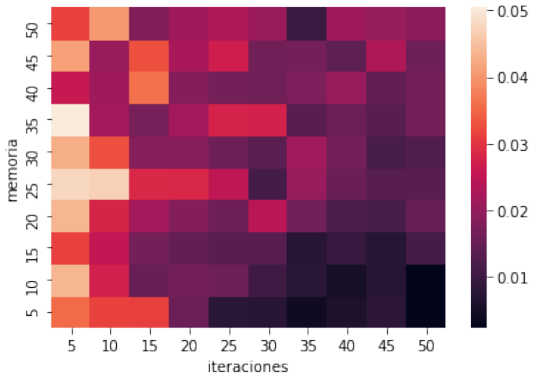

Figura 14: Heatmap con el gap (en el eje 'y') según memoria de aristas con 50 iteraciones (en el eje 'x')

Lo que esperabamos se cumplió en parte. Para la memoria de ciclos, a mayor límite de iteraciones, mejor resultado, sin importar el tamaño de la memoria. Entendemos que mientras más se itera, mejor es la solución. Luego, para la memoria de aristas observamos por primera vez un comportamiento distinto a la memoria de ciclos. Es cierto que a más iteraciones, mejor es la solución, pero la mejora no es tan drástica como en la primera imagen. Ademas, es claro que si se usa memoria más chica el algoritmo encuentra mejores soluciones. Esto se debe a que al algoritmo le es conveniente recordar 'swaps' cercanos a la solución que encontró hasta el momento y, si recordara 'swaps' que hizo hace muchas iteraciones, terminaría bloqueando vecinos que tal vez lo llevarian a una solución óptima. Entendemos que el factor de aleatoriedad es lo que genera el 'ruido' que se observa en ambos gráficos, quizás un poco es generado por el procesador, pero en general se observa que el gap es de los mejores que encontramos hasta el momento si los comparamos con los algoritmos anteriores.

Para cerrar esta etapa, vamos a evaluar una instancia nueva con los parámetros que entendemos como óptimos, los cuales son: memoria $T = 5$, porcentaje de vecindad $P = 20\,\%$ y cantidad de iteraciones máxima $I = 60$. Luego de esta experimentación nos decantaremos por una memoria o por la otra. Esperamos un comportamiento similar, con diferencias sutiles probablemente. Usaremos la instancia gr202.

Finalmente, nos vamos a decantar por usar la memoria de ciclos, dado que ambos algoritmos tienen resultados similares. Como se puede ver en el segundo gráfico, en tiempo es mejor y si bien en gap son parecidas, la relación costo beneficio nos dice que la memoria de ciclos es más redituable.
Entendemos con estos gráficos que Búsqueda Tabú, a un costo relativamente alto temporalmente hablando, logra reducir el gap a valores que son prácticamente óptimos.

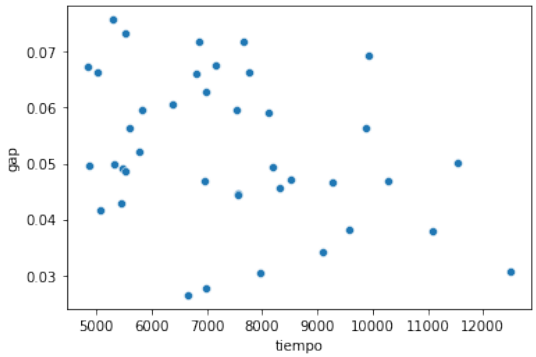

Figura 15: Scatter plot con parámetros ideales para memoria de ciclos con 40 iteraciones. En el eje 'x' se mide el tiempo en milisegundos y en el eje 'y' se mide el gap, que es la proporción sobresaliente del resultado encontrado en comparación con el óptimo conocido.

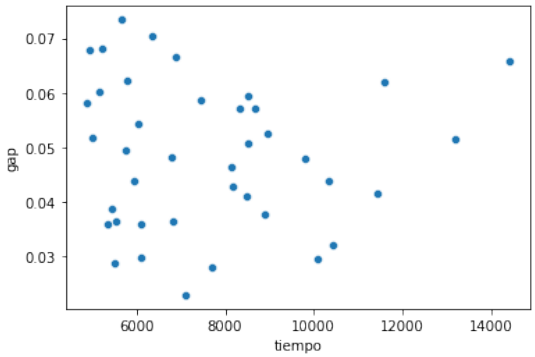

Figura 16: Scatter plot con parámetros ideales para memoria de aristas con 40 iteraciones. En el eje 'x' se mide el tiempo en milisegundos y en el eje 'y' se mide el gap, que es la proporción sobresaliente del resultado encontrado en comparación con el óptimo conocido.

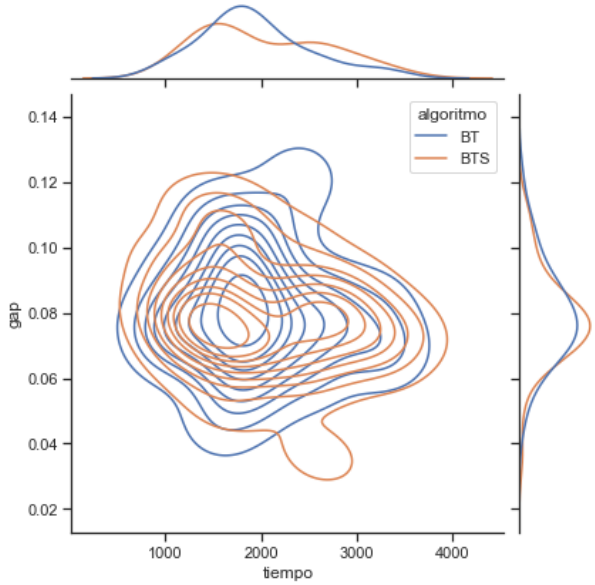

Figura 17: Kde plot de tiempo vs. gap para Búsqueda Tabú con mermoria de ciclos (Notación BT) y Búsqueda Tabú con memoria de aristas (Notacion BTS). En el lado superior se observa la distribución de tiempo y sobre el lado derecho la distribución de gap.

7. Conclusiones

Vamos a ver un último gráfico comparando todos las heurísticas utilizando una nueva instancia: ulysses22. Los parámetros que entendemos como óptimos para Tabú son: memoria $T = 5$, porcentaje de vecindad $P = 20\,\%$, cantidad de iteraciones máxima $I = 60$, heurística AGM y memoria de ciclos.

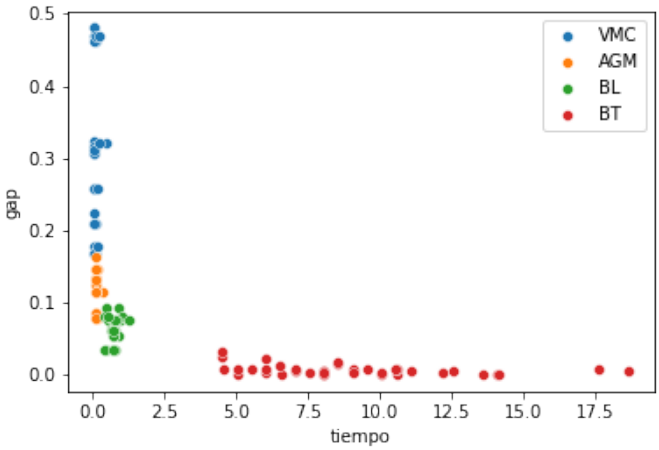

Figura 18: Gap contra tiempo de cada heurística. VMC es la heurística de vecino más cercano, AGM la de árbol generador mínimo, BL la de búsqueda local y BT es la de tabú search con los párametros que consideramos óptimos

Entonces tenemos que, entre las heurísticas golasas, AGM suele tener mejor rendimiento pero vimos casos donde no (dataset eil51. Ver figura 8), habría que experimentar siempre ya que es muy variable y debido a su bajo costo temporal, es razonable realizarlo. Luego, si se dispusiese de poco tiempo, pero queremos una mejor solución, optariamos por Búsqueda Local, ya que su tiempo de ejecución es bajo y los resultados son buenos. Por último, si realmente necesitamos exprimir al máximo para sacar la mejor solución posible, dentro de lo que nos permite esta metaheurística, usariamos Búsqueda Tabú (con memoria de ciclos, memoria $T = 5$, porcentaje de vecindad $P = 20$ y cantidad de iteraciones máxima $I = 60$.). A pesar de consumir más tiempo, entendemos que si se dispone del mismo, vale la pena consumirlo para encontrar un óptimo global o algo muy cercano a ello.