DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

# TP 2: Problema del viajante de comercio

Algoritmos y estructuras de datos III


| Integrante | LU | Correo electrónico |
|---|---|---|
| La Ruffa, Franco | 404/19 | francolr97@gmail.com |
| Loria, Ezequiel | 111/16 | c03iif@hotmail.com |
| Asmad, Victor | 760/19 | vasmad@dc.uba.ar |
| Cesaroni, Mateo | 394/20 | mateocesaroni@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
https://exactas.uba.ar

# 1. Introducción

El TSP (Travelling Salesman Problem) o Problema del Viajante de comercio en español, es un problema muy estudiado en el área de las ciencias de la computación Esta incógnita consta de responder la siguiente pregunta: Dada una lista de ciudades y la distancia entre cada par de ellas ¿Cual es la ruta mas corta posible que visita cada ciudad exactamente una vez y finaliza al regresar a la ciudad origen? Esta respuesta trae consigo el denominado Camino Hamiltoniano, nombrado así en honor a W.R. Hamilton quien definió el problema en los años 1800s

A día de hoy, no se conocen algoritmos exactos que puedan devolver resultados en tiempos aceptables, es decir polinomiales, por lo cual se considera al mismo como un problema NP-Hard Para la resolución del problema que se explicará a lo largo de todo este trabajo, se hará un recorrido por diversas técnicas denominadas Heurísticas cuyo objetivo es obtener diversos resultados aproximándonos mas a los óptimos globales, o en peor caso, los óptimos locales. Se define un óptimo local como un valor válido y de costo menor a otros y, el óptimo global como el mejor resultado entre todos los recorridos posibles

Dicho esto, comencemos

# 2. Descripción del problema

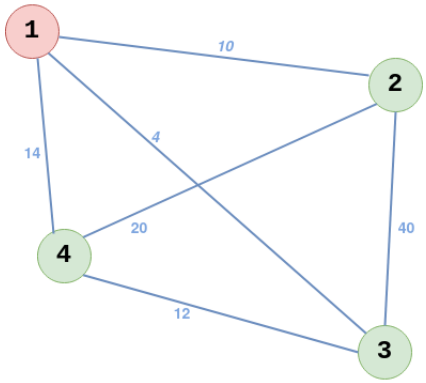
Un comerciante desea vender sus productos pasando por una cierta cantidad de destinos solo *una vez* y tomando el camino *más corto*.

Un viaje puede representarse por un grafo *G=(V,E)* donde cada destino es un vértice y si existe un camino directo entre dos vértices distintos ese camino se denomina arista. El problema del viajante de comercio (de ahora en adelante TSP: *Travelling Salesman Problem*) se resuelve si existe el camino más corto que visita todos los vértices una vez y permite que el viajante regrese a su posición inicial.

Los problemas de TSP pueden dividirse en dos tipos: aquellos problemas donde hay una arista entre todo par de vértices distintos, y aquellos donde no. En este trabajo práctico nos vamos a en focalizar solo en el primer tipo.

## 2.1. Ejemplo en la vida real

Consideremos a un taxista que debe llevar cuatro personas a sus respectivas casas un sábado a la 08:00 a.m. luego de que las mismas salgan de un bar. Como el taxista sabe que hay muchas personas queriendo regresar a sus casas va a tratar de llevar la mayor cantidad que pueda, esto se logra iniciando en la puerta del bar, llevando a sus pasajeros y regresando al mismo lo más rápido posible para repetir el proceso; el taxista necesita, por lo tanto, tomar el camino más corto entre casa y casa para minimizar el tiempo de viaje y volver rápidamente al inicio para tomar más gente. En el grafo de abajo se puede ver una instancia del problema donde el taxista tiene que llevar a tres personas desde el bar *vértice (1)* buscando la mínima distancia entre todas las casas de sus pasajeros y el inicio.




(a) El taxista: El coste óptimo es 46 recorriendo los nodos 1-2-4-3-1

# 3. Descripción de los algoritmos

En esta sección vamos a describir los algoritmos que se utilizaron para la resolución de este trabajo práctico. Los mismos son: Heurística constructiva golosa, Heurística basada en Árbol generador mínimo, Búsqueda local y Búsqueda Tabú

Para este algoritmo, las estructuras sobre las cuales se van a trabajar son grafos pesados completos, es decir, grafos densos. Este grafo no está orientado, esta decisión está tomada porque es indistinto ir desde $a$ hasta $b$ o viceversa puesto que solo voy a pasar por el nodo $a$ o $b$ una única vez por ejecución del algoritmo

Todos estos algoritmos reciben como input un grafo $G = (V, E)$ completo donde $\forall e \in E, e- > ei \forall i = 1, ..., n$

Un detalle de implementación es que el grafo va a ser representado como una lista de adyacencias. Esto es un diccionario donde cada clave es el nodo y, cuyo significado, es una lista de tuplas. Cada tupla contiene dos datos importantes, uno es el vértice destino y el otro es el coste de la arista que une ambos objetivos

## 3.1. Heurística constructiva golosa

El primer algoritmo que se va a describir está basado en la aplicación de la Heurística Golosa Constructiva Esta técnica se basa, aplicando un algoritmo goloso como se menciona en el nombre de la heurística, en empezar a partir de un nodo, llamémoslo $e_1$ y, a partir del mismo, ir recorriendo todo el grafo armando un camino simple.

La idea del algoritmo se basa en ir actualizando el ultimo nodo que estoy visitando, para esto llamemos $e_i coni = 1, ..., n$ al i-ésimo nodo que estoy visitando tal que $e_{i-1} - > e_i$ puesto que se trata de un camino. Para escoger un nuevo nodo, se miran todos los vecinos de $e_i$ y, entre todos aquellos se escoge el nodo cuya arista que une a $e_i$ con $e_{i+1}$ tenga menor peso En todo momento, se tiene que estar revisando cuales vértices ya están incluidos en el camino con la finalidad de no volver de un nodo hacia otro que ya se haya visitado y, de esta forma, estar armando un ciclo intermedio

Una vez haya recorrido todos los nodos y, tenga un camino simple sin ciclos (dato importante puesto que se trata de un camino euclidiano y no pueden haber ciclos intermedios como se menciono al principio), entonces conecto el último nodo que haya recorrido, llamémoslo $e_n$, y uno $e_n$ con $e_1$ sea cual sea el peso de la arista Se conecta el ultimo nodo con el primero puesto que, como se mencionó antes, el objetivo de este problema es armar un camino que inicie y termine en el mismo nodo desde el que partí y que, a su vez, no hayan ciclos intermedios

En la aplicación hecha del algoritmo, existe una constante la cual va conservando la suma de las aristas hasta el momento El algoritmo termina cuando haya encontrado un camino euclidiano, esto es, cuando en el camino estén todos los nodos del grafo original y, una vez suceda esto, se une el ultimo nodo del mismo con el primero. Si esto es posible, devuelve un camino con su coste, caso contrario, se informa que no hay un camino posible

**Algorithm 1** Algoritmo de Búsqueda Golosa Constructiva

```
1: costoFinal ← 0
2: function BG(GrafoG)
3:     S ← []
4:     Vértice n ← algún vértice cualquiera de V(G)
5:     costoAux ← ∞
6:     para cada e ∈ d(v)
7:     if e(v) ∉ S then costoFinal ← costoFinal + c_i
       costoFinal ← costoFinal + (e_n → e_i)
```

El algoritmo mencionado devuelve un camino euclidiano en el cual, se encuentran todos los nodos del grafo Con esto en cuenta, la complejidad del algoritmo se mide en base a los recorridos de los nodos ya que, en cada paso estoy tomando una aristas que une al nodo $e_i$ con el nodo $e_j$ vecino. Es decir, en cada iteración, estoy recorriendo todos los vecinos del nodo $e_i$ y tomando el que tenga arista de menor costo. Esto se realiza en $O(d(e_i))$

A su vez, recorro todos los nodos a lo sumo una vez, esto es $O(V)$ y también todos los vértices puesto que si estoy mirando todos los vecinos de cada nodo, a lo sumo reviso dos veces cada arista. Esto es porque ocurre en varios casos que miro el coste de la arista $e_i -> e_j$ y $e_j -> e_i$ pero en alguno de los dos casos, uno de los nodos ya está agregado al camino, entonces veo dos veces cada arista. Esto resulta en $O(E)$

Luego la complejidad es $O(V) + O(E) + O(d(e_i)) > O(V) + O(E)$ y, como se trata de un grafo completo, es un grafo denso, por lo tanto tengo a lo sumo $O(V^2)$ aristas. Finalmente, la complejidad del algoritmo es $O(V^2)$

No tenemos que olvidarnos tampoco de la complejidad espacial. Como buscamos un camino euclidiano, nos basta con tener una estructura de lista con tamaño $|V|$ puesto que lo que voy a tener son, en orden, todos los vértices que se visitan Si quiero las aristas individuales, me basta revisar en el grafo, la arista de la lista en posición $ij$ y obtengo el peso de la arista. Por lo tanto, como complejidad espacial se obtiene $O(V)$

## 3.2. Heurística basada en AGM

El segundo algoritmo está basado en la técnica de Heurística Basada en Árbol Generador Mínimo. La misma consta en utilizar un árbol generador mínimo como dice el nombre de la heurística y, posteriormente, utilizar otro algoritmo para recorrer el mismo obteniendo en este ultimo caso, el orden de visita a cada vértice del AGM Obtenido el orden de visita, se obtienen las aristas que cumplen que el resultado final sea un camino euclidiano y se cierra el ciclo, obteniendo de esta forma un camino válido

Respecto a la resolución de este algoritmo, se optó por utilizar para obtener el AGM, el algoritmo de Prim ya que, en cada iteración va generando un árbol que va cumpliendo con lo pedido, a demás, se nos hizo mas fácil la implementación del mismo Y, para recorrer árbol, se escogió DFS (Depth First Search) ya que este algoritmo recorre todo el grafo brindando información sobre la estructura del mismo y, obteniendo de esta forma, un orden para recorrer el resultado final

Tanto Prim como DFS son algoritmos golosos. Estos algoritmos van, en cada iteración, escogiendo la arista que tenga menor peso, siendo en parte similar a la Heurística Golosa Constructiva. No obstante, la aplicación de DFS para obtener el orden diferencia al mismo puesto que puede traernos mejores resultados al final, resultados con aristas que no tengan un peso exorbitante. Este tema se tratará posteriormente en la parte de experimentación

Los pasos para la resolución de este algoritmo son los siguientes:

- Se obtiene un AGM a partir del grafo original
- Se aplica DFS sobre el AGM
- Se recorre el orden de los nodos
- Obtenido un camino simple, se cierra el ciclo y se devuelve el camino junto con su costo

**Algorithm 2** Algoritmo de Heurística en Árbol Generador Mínimo

```
1: costoFinal ← 0
2: function AGM(GrafoG)
3:     Grafo S ← []
4:     Grafo G' ← BG(g)
5:     Grafo G" ← DFS(G')
6:     para cada e_i ∈ G'
7:     costoFinal ← costoFinal + e_i
```

La complejidad de este algoritmo se basa principalmente en la complejidad del algoritmo escogido para obtener el árbol generador mínimo Al tratarse de un grafo denso, el costo de ejecución de este algoritmo puede llegar a $O(|V|^2)$ ya que se tienen que recorrer a lo sumo $|V|^2$ aristas El algoritmo DFS se recorre sobre el AGM, entonces la complejidad del mismo es lineal respecto al resultado del algoritmo previo Unir la ultima arista con la primera se realiza en O(1) dependiendo de la estructura utilizada

Si se considera algún que otro cambio en la ejecución del algoritmo, este puede extenderse un poco en tiempo, es decir, ir buscando los vecinos de un nodo que esté visitado dos veces por como esta armado el AGM y hacer este cambio. Teniendo esto en cuenta, podemos saber que el algoritmo no tendrá mas de 2 veces el coste del AGM. Finalmente, la complejidad es $O(|V|^2)$

## 3.3. Búsqueda local

El tercer algoritmo se basa en la técnica de búsqueda local Esta es una heurística enfocada en la optimización de problemas difíciles.

El objetivo de esta heurística es, dada alguna solución valida, buscar otras soluciones con mejores resultados aplicando cambios locales hasta que se encuentre un resultado óptimo o pase un limite de tiempo.

Para la resolución de este problema, consideramos esta solución válida estará basada en un camino hamiltoniano $S$ y el costo de recorrerlo $c$, entonces la forma de obtener otros valores consta de buscar los vecinos de $S$. A este conjunto, se lo denomina la vecindad.

Para ello, definimos la vecindad como $Vecindad(S) = (S'|S' = swap(S, i, j)parai, jen(1, ..., n))$. Esto se obtiene de aplicar la técnica 2-opt que consiste en, dados 4 vértices en la solución original, intercambiar las aristas que las unen. Esto lo podemos hacer puesto que G es un grafo completo, si no, no sería válido. Entonces, sean $a, b, c, d \in E$, $a- > b$ y $c- > d$ vértices unidos por sus aristas en el camino $S$, el resultado obtenido después de intercambiar aristas es $a- > c$ y $d- > b$. Este cambio nos permite obtener otro camino válido que se denomina vecino de $S$ ya que surge a partir de usarlo para generar otro. Entonces, definimos la vecindad como: $Vecindad(S) = (S'|S' = swap(S, i, j)parai, jen(1, ..., n))$

Dijimos previamente que para realizar una búsqueda local, se necesita un camino válido ya calculado. Para esto se decidió tomar el camino generado por la heurística heurística puesto que, luego de varias pruebas que se explicarán mas adelante en la sección de experimentación, nos devuelve un camino válido con un coste mucho menor que la heurística heurística. Entonces, teniendo ya un camino con un coste aceptable, buscamos la vecindad del mismo para buscar una mejor solución.

El algoritmo finaliza cuando no puede encontrar una solución mejor a la ultima ya encontrada Luego, el algoritmo se define como:

**Algorithm 3** Algoritmo de Búsqueda Local

```
1: costoFinal ← 0
2: function BÚSQUEDA LOCAL(GrafoG)
3:     S ← BG(G)
4:     hayMejora ← true
5:     solOptima ← S
6:     while hayMejora do
7:         Vec ← Vecindad(S)
8:         if costo(Vec) < costo(S) then S ← Vec
9:         if costo(Vec) > costo(S) then hayMejora ← false
```

Debido a que el algoritmo se detiene cuando no encuentra una mejora al ver los vecinos, definimos una variable cantDeMejoras que va a representar la cantidad de iteraciones en donde encontró una mejor solución (linea 6) para poder explicar correctamente la complejidad.
Obtener el vecindario tiene costo $O(n^2)$ ya que para cada vértice, hago permutaciones con los demás vértices El obtener el costo del camino implica recorrer una lista de tamaño N ($O(N)$) revisando las aristas de vec[i],vec[i+1] también en $O(N)$ ya que las ve mediante lista de adyacencia en donde ubica al vértice i en $O(1)$ y recorre sus vecinos en $O(N)$. Finalmente la complejidad es $O(N^2)$
Con todo esto en cuenta, hacemos se hace el calculo de la complejidad: $O(AGM) + (O(N^2) + O(N^2)) * O(cantDeMejoras) = O(AGM) + 2O(N^2) * O(cantDeMejoras) \equiv O(cantDeMejoras * O(n^2)$ Finalmente, resumimos que la complejidad de obtener una solución válida resulta en $O(cantDeMejoras * O(n^2)$

## 3.4. Heurística de Búsqueda Tabú

El ultimo algoritmo que vamos a utilizar está basado en la meta heurística de la Búsqueda tabú. Este método surge como una aplicación de la búsqueda local (ya presentada en el algoritmo anterior) para obtener mejores resultados

La razón de este uso es, que mientras la búsqueda local puede quedarse en óptimos locales buscando en cierta vecindad de la solución y, siempre avanza buscando una mejor solución, la búsqueda tabú se puede permitir el tomar resultados peores con la finalidad de, en una próxima iteración buscando otra solución, obtengamos un resultado mucho mejor que sin haber tomado esta decisión

En algunos casos, el realizar esta acción de permitirse ir por peores resultados para luego encontrar uno mejor, permite que se pueda encontrar un óptimo global frente a un óptimo local como si sucedería con la búsqueda local

Para poder obtener mejores soluciones, nos basamos en la búsqueda local, especialmente vamos a enfocarnos en la técnica 2-opt para la cual, daremos ciertos parámetros que nos permitan extender los resultados y obtener mejores valores, estos se denominan, los componentes los cuales nos dan control sobre el mismo

Como componentes nos vamos a enfocar en los siguientes items:

- Criterios de parada - Cantidad de iteraciones: Este parámetro limita la cantidad de operaciones o, iteraciones, que se pueden realizar para obtener mejores resultados. Por iteración nos referimos a la cantidad de vecindades que podemos obtener dado el ultimo resultado obtenido, de esta forma limitamos cuanto podemos buscar dentro del vecindario de la solución ya que, puede ocurrir que la cantidad de vecinos sea incluso infinita si tenemos infinitos vértices, si no limitamos este parámetro, podemos quedarnos iterando eternamente

- Subconjunto de vecindad - 2opt + Random : Este parámetro consiste en, dada una solución $S$ y $Vecindad(S)$, de todos los posibles valores en la vecindad, tomamos un porcentaje random de los resultados, es decir, los caminos con aristas cambiadas
- Memoria : Para este item, se decidió utilizar una lista como estructura de memoización
  - Recordar Ciclos : En cada iteración se recuerda el camino entero obtenido sobre el cual, posteriormente, se buscarán mejores resultados
  - Recordar Swaps : Dado un camino S' resultado de aplicar la operación swap, se recordarán los vértices que intercambiaron de aristas para formar una nueva vecindad

  En el algoritmo utilizado se aplicarán ambos tipos de memoria, por ende tenemos un parámetro que nos indicará que tipo de memoria utilizar y, dependiendo del mismo, podemos obtener incluso distintos resultados El porque de usar la memoria consiste en, que si un ciclo o un swap ya ha sido visitado, no se vuelva a visitar en alguna próxima iteración donde haya algo similar Por ejemplo: Es lo mismo el camino que contenga a la arista $a \to b$ como $b \to a$ pues el grafo no es dirigido La misma idea ocurre con memorizar los ciclos
- Aspiración Este ultimo componente consiste en recordar en todo momento la mejor solución vista hasta el momento. Esta será considerado nuestro resultado final hasta que encontremos uno mejor, en tal caso, hacemos el cambio

Como solución inicial elegimos el camino obtenido por el algoritmo alg Teniendo esto en cuenta, definimos el pseudocódigo del algoritmo como:

**Algorithm 4** Algoritmo de Búsqueda Tabú

```
1: costoFinal ← 0
2: function BÚSQUEDA TABÚ(G : Grafo)
3:     S ← AGM(G)
4:     sol ← S
5:     iteración ← 0
6:     while mientras iteración < iteracionesMax do
7:         subVec ← obtenerSubVecindad(S, porcentaje)
8:         mejorEnSubVec ← obtengo el mejor en la subvecindad de S
9:         if coste(sol) ¡coste(S) then
10:            S ← sol
```

**Algorithm 5** Algoritmo de obtención de SubVecindad

```
1: function OBTENERSUBVECINDAD(G : Grafo, porcentaje : ℝ)
2:     vecinos ← []
3:     i, j ← 0
4:     while i ≤ n do
5:         while j ≤ n do
6:             vecinos ← Ag(vecinos, swap(G, i, j))
7:             j ← j + 1
8:         i ← i + 1
9:     SubVecindad ← []
10:    while |vecindario| < vecinos * porcentaje do
11:        Ag(vecindario, dameUno(vecinos)
12:    devolver SubVecindad
```

**Algorithm 6** Algoritmo de mejorSubvecino

```
1: function MEJORSUBVECINO(GrafoG, subVecindadsv, memoriamemo)
2:     mejor ← INFTY
3:     mejorOrden ← 0
4:     swap ← []
5:     mejorSwap ← []
6:     if tipoMemo== Ciclo then
7:         i ← 0
8:         while i < len(sv) do
9:             if sv[i] ∉ memo then sol ← costo(sv[i])
10:                if sol<mejor then
11:                    mejor ← sol
12:                    mejorOrden ← sv[i]
13:        if len(memo)<tamanioMemo then
14:            Ag(memo, sv[i])
15:        if len(memo)==tamanioMemo then
16:            memo.popfront()
17:            Ag(memo, sv[i])
18:    if tipoMemo==swap then
19:        i ← 0
20:        while i < len(sv) do
21:            swap ← swapEn(sv[i])
22:            if swap ∉ memo then sol← costo(sv[i])
23:                if sol¡mejor then
24:                    mejor ← sol
25:                    mejorOrden ← sv[i]
26:                    mejorSwap ← swap
27:        if len(memo)<tamanioMemo then
28:            Ag(memo, swap)
29:        if len(memo)==tamanioMemo then
30:            memo.popfront()
31:            Ag(memo, swap)
32:    mejorSv ← (mejor, mejorOrden)
33:    devolver mejorSv
```

El cálculo de complejidad de este algoritmo es distinto en relación a la búsqueda local pese a estar basado en el mismo y sus métodos. Para esto, analizaremos el algoritmo respecto a los valores de las componentes, esto es, Criterios de parada, Conjunto de subvecindad, memoria y Aspiración

Empecemos por Aspiración. Esta parte indica que se almacena siempre la mejor solución obtenida hasta el momento. Esto es un camino de $|V|$ vértices en memoria. O(n)

El algoritmo de obtención de la sub vecindad (linea consiste en, para todos los vértices obtener todas las permutaciones de este mismo e ir almacenándolos en una lista. El obtener todos estos vecinos tiene costo $O(n^2)$ ya que consiste en recorrer dos veces todos los vértices para permutar las aristas Posteriormente, entra en juego el parámetro del conjunto de sub vecindad. Como este valor es un número que nos dice que porcentaje de todos los vecinos de la solución se toma, el peor caso consiste que el valor sea de 100 ya que ese valor representa el 100 de los vecinos, en tal caso, se llena una lista a devolver con todos los vecinos. Esto se hace puesto que, si el valor es inferior a 100, vamos tomando de manera aleatoria los vecinos y los colocamos en esta lista. Para verificar si el vecino no fue agregado ya, usamos otra estructura de memoización de tamaño $|vecinos|$ donde tenemos O si no fue agregado y 1 si lo fue, esta estructura tiene tamaño $O(|vecinos|)$ como la que va a devolverse como respuesta. Como para seleccionar que valores tomamos usamos una

generación aleatoria, puede no ser uniforme la selección de que valore se toman y, para casos como un porcentaje de 50, puede pasar muchas veces por un mismo valor antes de cambiar a otro Entonces la complejidad para este algoritmo consta de $O(max(n^2, filtro)$.

Luego tenemos el método para obtener el mejor de los vecinos. Para este método tenemos que tener en cuenta algo importante. Se reciben como parámetro el tipo de la memoria y el tamaño de la memoria. Dependiendo del tipo de memoria la complejidad puede cambiar. Esto se debe a que, en la estructura de memoización se almacenan 2 tipos de datos, o ciclo o swap y, en cada iteración se busca si el valor se ha visitado antes o no. Dicha comparación se realiza a motivo de no repetir procesos innecesarios ya que recorrer una arista $(a, b)$ es lo mismo que recorrer $(b, a)$ por ejemplo. El costo de hacer una comparación del swap es O(1) puesto que se comparan tuplas de 2 valores En el caso de los ciclos, puede ocurrir que para dos caminos distintos, un swap haga que los caminos queden iguales. En este caso, el costo de comparar se eleva a $|V|$ ya que, en la estructura, se guardan caminos enteros Sumado a esto, en cada comparación se debe recorrer la estructura de memoización ya que es una lista que se va extendiendo mientras la memoria no se haya completado Entonces, buscar si un valor está o no en la estructura tiene costo $O(tamanioMemoria * N)$.

Luego, hacemos esto por cada elemento en los vecinos. Por ende la complejidad incrementa a $O(cantVecinos * tamanioMemoria * N)$. La cantidad de vecinos es a lo sumo $O(n^2)$ porque se permuta a lo sumo 2 aristas de cada camino, entonces el tiempo es acotado. Con esto, la complejidad es $O(n^3 * tamanioMemoria)$.

Finalmente, teniendo estos dos algoritmos, la complejidad es de $O(max(n^2, filtro)) + O(N^3 * tamanioMemoria)$. Esta complejidad se realiza cantIteraciones, que es el componente del criterio de parada. Con esto en cuenta, la complejidad final del algoritmo asciende a: $O((max(n^2, filtrar) + tamanioMemo * N^3) * CantIt)$

# 4. Experimentación

## 4.1. experimentos con datasets dados

Para estos experimentos vamos a utilizar las siguientes instancias:

- **ulysses16**
- **berlin52**
- **st70**
- **kroE100**

Estas instancias fueron elegidas para las experimentaciones de Tabú ya que representan una variedad suficiente de tamaños de entrada y se consideró que ver otras instancias con N muy grande sería demasiado costoso con respecto al tiempo y sus resultados no serían lo suficientemente distintos a las instancias propuestas como para que sea de interés analizarlos.

Para empezar, vamos a comparar los resultados de AGM y BG en las instancias presentadas anteriormente con la finalidad de decidir cual de los algoritmos se va a utilizar como solución inicial para búsqueda local y tabú.

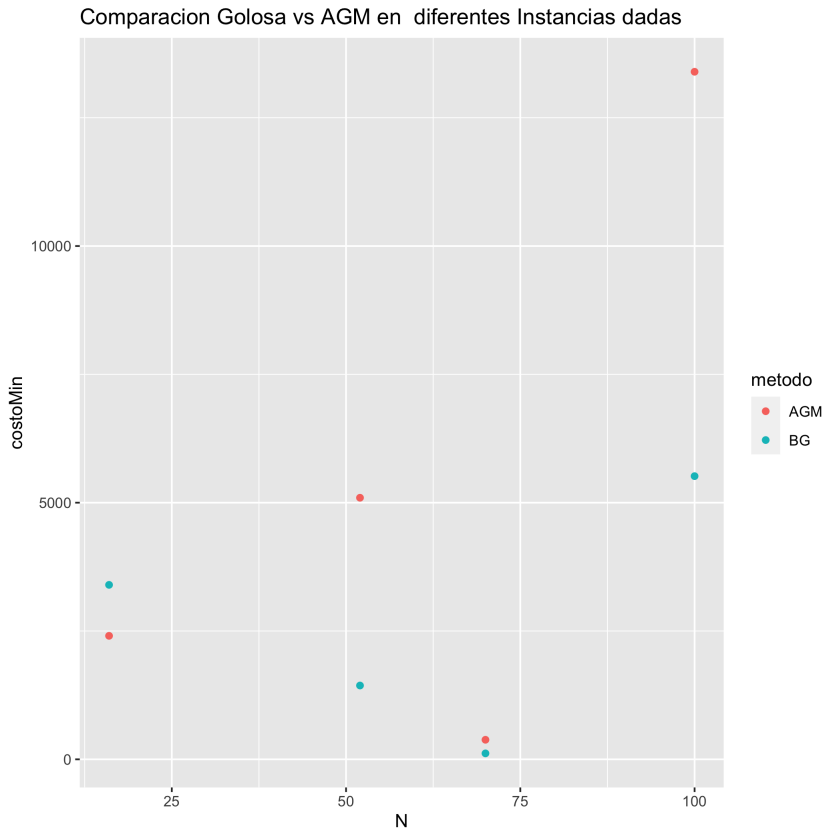



Figura 2: comparación Agm vs BG

Como podemos observar en 2, en la mayoría de los casos probados la heurística golosa que implementamos en el algoritmo 1 devuelve un costo más cercano al óptimo, por lo tanto vamos procedemos a utilizarla.

Ahora vamos a analizar que tipos de parámetros son mejores para la resolución de Tabú:

Primero vamos a comparar el porcentaje de subvecindad:

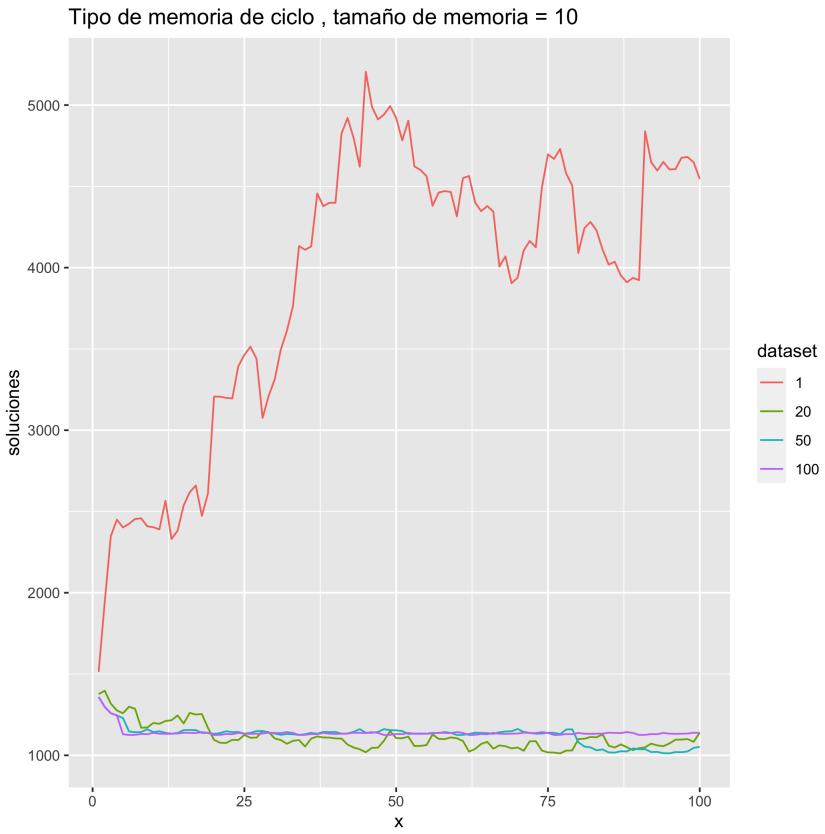



Figura 3: comparación subvecindad

Como podemos apreciar, con subvecindad 1 el algoritmo se mueve a distintas soluciones de manera errática entre iteraciones, devolviendo soluciones muy variadas, mientras que con mayor subvecindad las soluciones que va encontrando son más cercanas entre sí y por lo tanto es mas fácil que vaya mejorando, mientras que en el extremo de subvecindad 100 el algoritmo se comporta similar a búsqueda local, quedándose solo en soluciones muy similares, sin poder salir de su vecindario.

Siguiente comparamos tamaño de memoria:

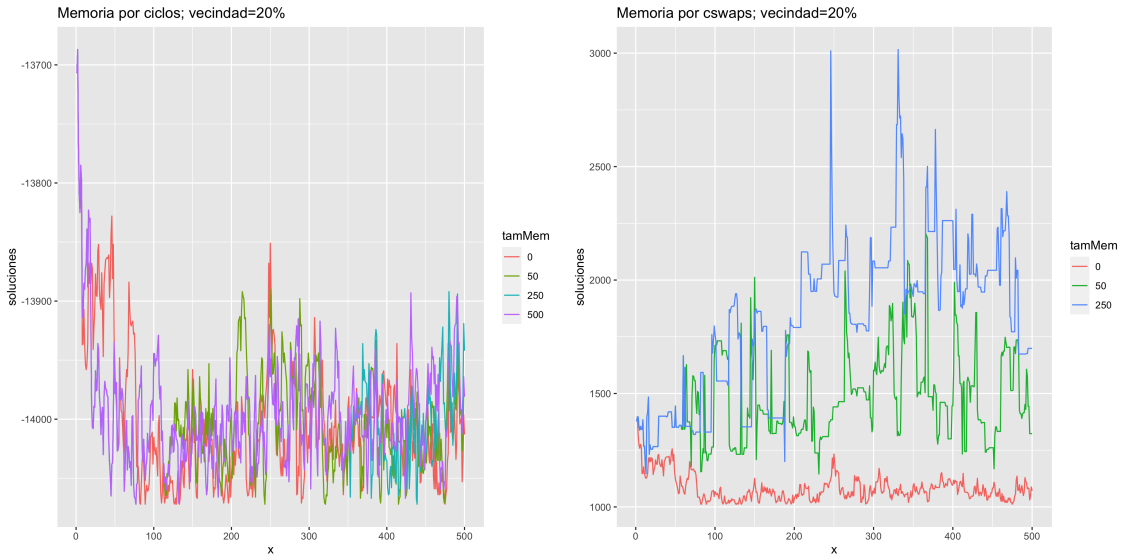



Figura 4: comparación tamaños por tipo de memoria

En la figura 4 podemos observar que, cuando se trata de memoria por ciclos, el tamaño no afecta mucho a que tipo de soluciones se mueve, siendo todos los tamaños similares en cuanto a como se mueven entre soluciones por iteración.

Por otro lado, vemos que si la memoria almacena los *swaps*, mientras mas grande es la memoria, las iteraciones se empiezan a mover a soluciones peores ya que este tipo de memoria bloquea las soluciones posibles al bloquear movimientos en vez de ciclos.

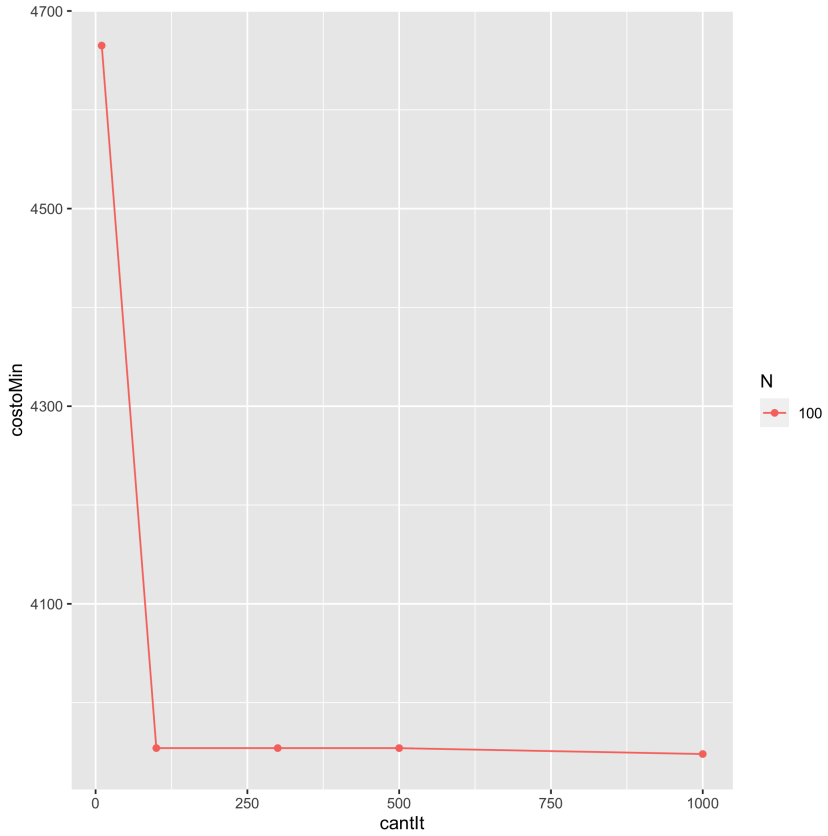



Figura 5: comparación iteraciones

Para la cantidad de iteraciones sabemos que mientras mas hagamos, más posibilidades de encontrar una solución mejor, por lo tanto hacer más iteraciones es mejor con respecto a sacar un mejor óptimo, pero afecta de gran manera al tiempo de ejecución final.

Con los resultados obtenidos en nuestro análisis, concluimos que mientras mayor cantidad de iteraciones la solución va a ser mejor, también vemos que el tipo de memoria por ciclos es mas consistente para encontrar nuevas y mejores soluciones sin importar mucho su tamaño, y una subvecindad medianamente baja es mejor para poder recorrer mayor cantidad de vecinos distintos, sin llegar a comportarse de manera errática viendo soluciones muy alejadas de la óptima.

## 4.2. experimentos con datasets propios

Para estos experimentos vamos a utilizar los siguientes datasets:

- **random**:instancias con N que va de 10 a 50 donde las aristas tienen pesos similares.
- **randomVariacionAlta**:instancias con N que va de 10 a 50 donde las aristas tienen una variación grande de peso entre sí.

Con las consideraciones anteriores sobre los parámetros a usar en tabú, vamos a comparar los cuatro algoritmos pedidos, en cuanto a tiempo y calidad de resultado final. Para tabu vamos a utilizar cantIteraciones 1000, memoria de 300, subvecindad de 20 en los dos tipos de memoria

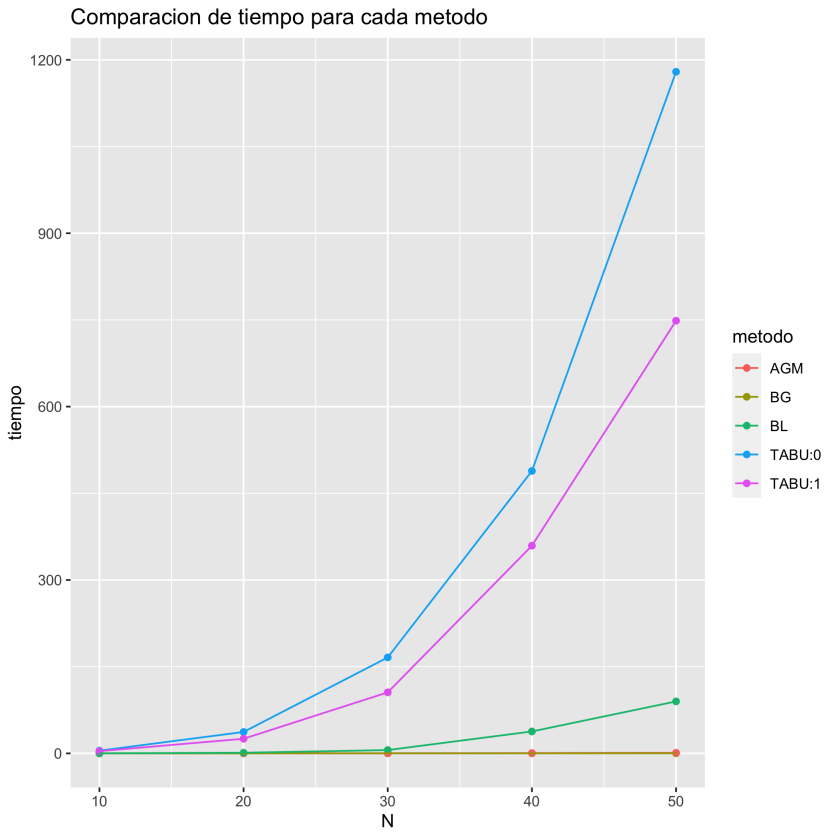



Figura 6: comparación tiempos por algoritmo

Primero aclaramos que TABU:0 representa tabú por memoria de ciclos mientras que TABU:1 representa memoria por swaps. Con eso en cuenta, vemos que en la figura 6 podemos diferenciar claramente que cada algoritmo tiene tiempos distintos menos BG y AGM que tienen tiempos similares. A su vez podemos ver que tabú cuando es memoria por swaps es mas rápida que por ciclos como explicamos en la parte de complejidad del algoritmo 4.

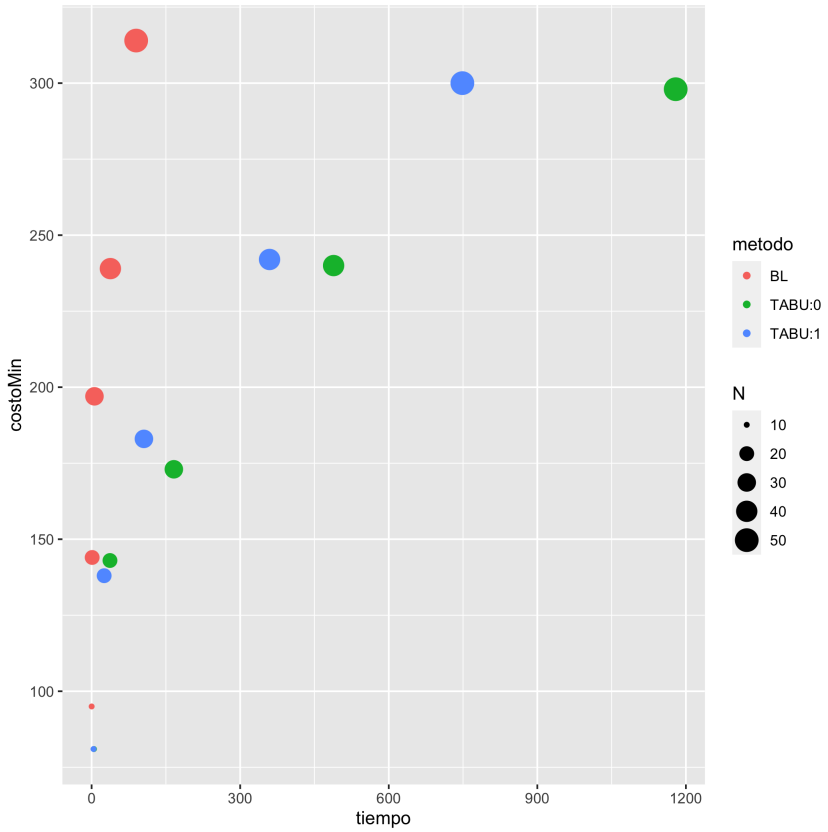



Figura 7: comparación calidad solución entre tabu y local

En la figura 7 se puede observar la comparación entre tiempo y solución óptima hallada. Podemos notar que tabú es considerablemente más lento que búsqueda local, pero devuelve mejores soluciones en la mayoría de casos. Por otro lado, apreciamos que la búsqueda local devuelve un resultado bastante cercano a la que brinda tabú en un tiempo mucho menor.
Por lo tanto podemos concluir que en casos donde N sea muy grande, si se necesita obtener una solución rápida, realizar búsqueda local va a brindar una respuesta aceptable en poco tiempo, mientras que si se dispone de mucho tiempo, no hay razón para no utilizar tabú y obtener un mejor resultado.

# 5. Conclusión

Para el desarrollo de este trabajo, como bien se explicó y presentó a lo largo de este trabajo, se utilizaron cuatro algoritmos, algunos distintos entre si y otros basados en el anterior, inclusive alguno mejorado otro.

Para empezar, los algoritmos mas sencillos como la búsqueda golosa o la heurística basada en AGM son aquellos que, en tiempo de ejecución, son rápidos para brindar una respuesta no obstante, la misma puede no ser la óptima incluso con un ratio de error bastante grande, pero para obtener un resultado y poder operar, son bastante eficaces.

Entre estos dos algoritmos, su comparación de resultados, podemos decir que la Búsqueda Golosa nos brindó resultados mucho mas cercanos a los óptimos que se han encontrado al momento para varios ejemplos tales como Berlin52, ST70, Ulysses16, Kroe100.

Por otro lado, tenemos el algoritmo de Búsqueda Local que dada una de las dos soluciones previas de goloso y AGM se enfoca en encontrar una mejor a partir de ciertos cambios en el camino. Este algoritmo en relación de tiempo puede ser un poco peor que los anteriores, pero para

encontrar un mejor resultado, es mucho mejor a la hora de dar soluciones mucho mas cercanas al óptimo. En relación al tiempo de ejecución con los dos algoritmos previos, este es el que más tiempo demora en ejecutarse para dar una solución.

Para finalizar, tenemos al algoritmo basado en la búsqueda tabú que, en términos generales, fue aquel que mejores resultados nos ha brindado dándonos soluciones más cercana a los óptimos encontrados hasta el momento en diversos ejemplos. No obstante, el uso de las componentes que se explicaron en su sección, ocasiona que el algoritmo pueda extender su tiempo de ejecución en una muy pero muy amplia diferencia y también pueda ocupar mas recursos de la computadora tal como la memoria ya que, para extender mejores soluciones, almacena caminos enteros y, para grafos con muchos nodos, esta diferencia puede crecer bastante. Si lo comparamos con los tres algoritmos previos, este es el que más recursos y tiempo consume no obstante, es el que mejor resultados brinda en todos los casos

Entre todos estos algoritmos, si se necesita una respuesta rápida y efectiva, podemos usar la Búsqueda Local para tener una mejor respuesta, sin embargo, si necesitamos una solución mas cercana al óptimo, la búsqueda local resulta darnos mejores resultados

Teniendo estas conclusiones previas sobre cada algoritmo, una futura actividad a tener presente para una posible mejora, es probar en la heurística del AGM otro algoritmo de generación de Árbol tal así como Kruskal. Proponemos este cambio ya que, contrario al algoritmo de Prim que va generando un árbol a lo largo del grafo, Kruskal genera bosques y, la elección de como arma esos bosques, puede afectar al resultado final dando otros resultados y podría ser un buen punto de partida comparar los resultados obtenidos.

También otro objetivo puede ser, probar con otras estructuras de representación de grafos tales como Matriz de Adyacencia o Lista de Adyacencias de forma que para comparar tiempo de ejecución ya que alguna estructura puede obtener datos como aristas o valores en tiempo O(1) y esto puede reducir las ejecuciones considerablemente.