DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Circuitos Hamiltoneanos

Algoritmos y Estructura de Datos III

Segundo Cuatrimestre de 2021

Integrante	LU	Correo electrónico
Lucas Iván Kruger	799/19	Lucaskruger10@gmail.com
Sebastián Cantini Budden	576/19	sebascantini@gmail.com
Bruno Robbio	480/09	brobbio@hotmail.com
Pedro Aisenson	395/20	pedroaisenson@gmail.com

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

http://www.fcen.uba.ar

Índice

1. Introducción

El Problema del Viajante de Comercio (TSP) se puede definir formalmente de la siguiente manera. Sea $G = (V, E)$ un grafo completo donde cada arista $(i, j) \in E$ tiene asociado un costo c_{ij}. Se define el costo de un camino p como la suma de los costos de sus aristas $c_p = \sum_{(i,j)\in p} c_{ij}$.

El problema consiste en encontrar un *circuito hamiltoniano* p de costo mínimo. Decimos que un circuito es hamiltoniano si pasa por cada vértice de G exactamente una vez. Sin pérdida de generalidad, vamos a pedir que el primer vértice del ciclo sea el 1. En los próximos grafos se muestran ejemplos de TSP junto con sus soluciones.

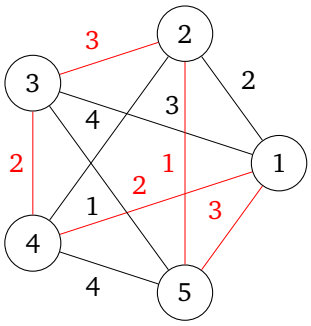

Figura 1: La solución es $1 - 5 - 2 - 3 - 4 - 1$

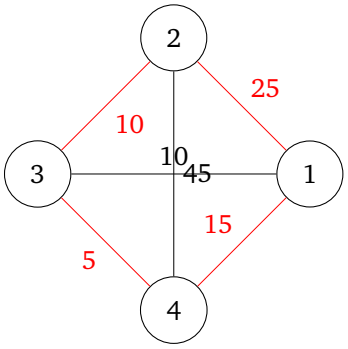

Figura 2: La solución es $1 - 2 - 3 - 4 - 1$

Este problema sirve para modelizar diversas situaciones de la vida real. La función de costo puede representar cualquier métrica que queramos minimizar, por ejemplo distancias o costos de atravesar una arista.

Como en un grafo de n vértices hay $(n - 1)!/2$ posibles recorridos este problema no es óptimo para resolverlo con algoritmos de fuerza bruta. En general no existen algoritmos exactos que puedan resolver este problema en complejidad polinomial así que debemos acudir a heurísticas que nos proveen soluciones decentes en una buena complejidad pero que no son necesariamente óptimas.

Resolver eficientemente el problema del viajante de comercio resulta muy beneficioso en muchas situaciones de la vida real. Entre ellas se encuentran la planificación, la logística y la fabricación de microchips.

Un problema de la vida real donde el TSP resulta útil para modelar se muestra en la figura 3. En éste mapa se buscaba recorrer todos los gimnasios y pokeparadas del juego PokemonGo en la ciudad de Cincinnati, en Estados Unidos. En este caso, se quería recorrer las pokeparadas de la manera más rápida posible, ya que los pokemones que allí figuraban desaparecían en un tiempo limitado. El modelado en éste caso va a tomar como nodos las pokeparadas y los pesos de las aristas son las distintas distancias o tiempos que cuesta ir de un nodo a otro. Otro caso en el que el problema de viajante de comercio es útil es en la fabricación de microchips. Para la manufactura de microchips en fábricas se utilizan robots que automaticen el agujereado de las placas. Para reducir el tiempo que cuesta el agujereado, se debe encontrar la forma óptima de realizar las conexiones.

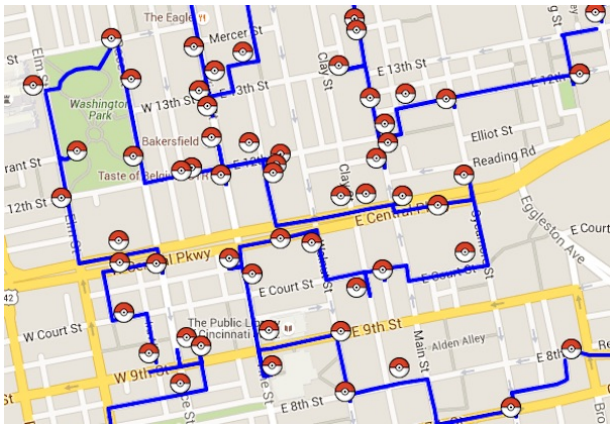

Figura 3: Tour óptimo de pokeparadas en Cincinnati, EEUU.

En éste informe vamos a analizar distintas heurísticas para la resolución de TSP. Concretamente, utilizaremos una heurística constructiva golosa, una heurística basada en árbol generador mínimo, un algoritmo de búsqueda local y una metaheurística de Tabú Search sobre búsqueda local. El objetivo es implementar cada uno de esos algoritmos y realizar experimentos sobre instancias evaluando el rendimiento de cada uno. Para ello, tendremos en cuenta tanto la calidad de la aproximación en cada instancia a la solución real como el tiempo de ejecución.

El informe va a estar organizado de la siguiente forma. En la sección 2 describiremos la implementación de cada una de las heurísticas consideradas. En la sección 3 describimos las instancias consideradas y realizamos los experimentos de cada algoritmo. En la sección 4 daremos las conclusiones obtenidas a partir de los experimentos realizados.

2. Implementación

Para representar el grafo utilizamos una matriz de relación donde en la posición $m[i][j]$ esta el costo de la arista que une al nodo i con el nodo j. Notemos que como trabajamos con un grafo completo en el cual todo nodo se conecta con todos los nodos, tenemos $\#\text{vértices}^2$ aristas. Con esto decimos que la complejidad de recorrer la matriz es de $O(m)$ con m siendo la cantidad de aristas.

Utilizamos 4 métodos distintos para poder encontrar el circuito hamiltoneano, y siempre devolveremos la solución empezando desde el primer nodo. Las dos primeras soluciones son heurísticas las cuales no siempre devuelven la solución mas óptima sino una aproximación, y las ultimas dos son soluciones que intentan optimizar una posible solución buscando una nueva solución mas barata.

2.1. Heurística constructiva golosa:

Una Heurística constructiva es un método que construye soluciones factibles para un problema. En este caso construimos las soluciones de forma golosa, osea utilizamos alguna heurística para intentar llegar a una solución óptima o que se acerque a ella. En este caso la heurística que utilizamos es la del vecino mas cercano. En cada iteración nos paramos en un nodo visitado y buscamos su vecino mas cercano no visitado, luego nos paramos en ese y hacemos lo mismo hasta llegar de vuelta al nodo inicial.

En este algoritmo usamos un ciclo while para recorrer todos los nodos hasta llegar al nodo inicial. Dentro del ciclo while, en cada nodo que nos paramos recorremos con el ciclo for todas sus aristas y nos quedamos con la de menor costo. Luego, agregamos al camino el nodo al que lleva la arista, sumamos su costo al costo mínimo y se ejecuta la siguiente iteración del while.

Este algoritmo tiene complejidad de $O(n^2)$ u $O(m)$, donde n es la cantidad de nodos y m la cantidad de aristas, ya que en el while vamos a recorrer todos los nodos y para cada nodo, recorremos los nodos

Algorithm 1 Heurística constructiva golosa*(vector⟨vector⟨int⟩⟩ incidencias)*

```
visitados ← Vector(n, 0)
camino ← Vector(1, 1)
costoMinimo ← 0
j ← 0
visitados[0] ← 1
while True do
    Nodo ← j
    for k = 0...incidencias.size() − 1 do
        if (visitados[k] == 0 and incidencias[nodo][k] < incidencias[nodo][j]) then
            j ← k
        end if
    end for
    if (j == Nodo) then
        BreakWhile
    end if
    camino ← camino.pushback(j + 1)
    costoMinimo ← costoMinimo + incidencias[nodo][j]
end while
costoMinimo ← costoMinimo + incidencias[0][j]
return costoMinimo,camino
```

conectados a éste.

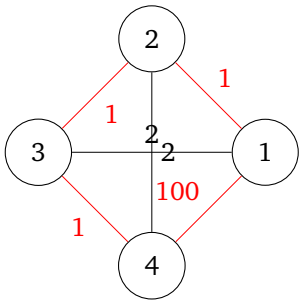

Figura 4: Solución de heurística constructiva golosa

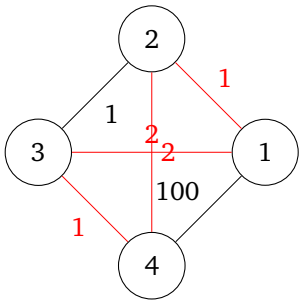

Figura 5: Solución óptima

En la Figura 4 se muestra en rojo el camino recorrido por el algoritmo anterior. Al tomar siempre la arista de menor peso y empezando por el primer vértice, el algoritmo recorre el grafo en orden $1-2-3-4$. Al llegar al nodo 4, está forzado a tomar la arista $\{1, 4\}$, que es la de mayor peso. Por otro lado, a la derecha tenemos el circuito hamiltoniano óptimo, que no coincide con el construido por el método goloso.

2.2. Heurística Basada en Árbol Generador Mínimo

Esta heurística también es constructiva y Golosa pero a diferencia del anterior, utilizamos un algoritmo basado en el de Kruskal. Éste algoritmo va construyendo un árbol generador mínimo que, en cada iteración, recorre todas las aristas y se queda con la de menor peso entre aquellas que cumplan que:

- No forma ciclos con las que ya elegimos
- No hace que ningún vértice ya elegido sea de grado tres

El algoritmo itera $\#\text{nodos}-1$ veces, lo cual produce un árbol generador mínimo del grafo, al que luego le agrega una arista que une el primer vértice con el último. Dado que el grafo es completo, esto

produce un ciclo hamiltoniano, que el algoritmo devuelve. A continuación veremos el pseudocódigo y veremos cómo solucionamos los dos puntos anteriores para elegir una arista.

Algorithm 2 Heurística AGM*(vector<vector<int>> incidencias)*

```
1: i ← 0
2: while i ≤ incidencias.size() − 1 do
3:    MenorArista ← ⟨0, 0⟩
4:    for arista ∈ incidencias do
5:       if NoFormaCircuito(arista, X)andNoFormaVerticesDeGrado3(arista, X) then
6:          if incidencias[arista] ≤ incidencias[MenorArista] then
7:             MenorArista ← arista
8:          end if
9:       end if
10:   end for
11:   X ← X.pushBack(MenorArista)
12:   actualizarEstructura()
13:   i ← i + 1
14: end while
15: return OrdenarAristas(X)
```

A continuación incluímos el pseudocódigo de la función ordenarAristas, que funciona como auxiliar de heurística AGM en el código fuente.

Algorithm 3 ordenarAristas*(vector<pair<int, int>> aristas)*

```
1: ans ← ∅
2: caminos ← Vector(∅, aristas.size());
3: for i ∈ range(aristas.size()) do
4:    caminos[aristas[i].first].append(aristas[i].second)
5:    caminos[aristas[i].second].append(aristas[i].first)
6: end for
7: ans.append(0)
8: ans.append(caminos[0][0])
9: for i ∈ range(aristas.size()) do
10:   if (caminos[ans[ans.size() − 1]][0] == ans[ans.size() − 2]) then
11:      ans.append(caminos[ans[ans.size() − 1]][1])
12:   else
         ans.append(caminos[ans[ans.size() − 1]][0])
13:   end if
14: end for
15: for i ∈ range(caminos.size()) do
16:   ans[i] ← ans[i] + 1
17: end for
18: return ans
```

Notemos que en la linea 3 del algoritmo principal, definimos $MenorArista$ como la arista que una el vertice cero con el vertice cero. Aunque esta arista es inexistente, la usamos como placeholder ya que en nuestra matriz, incidencias, las aristas inexistentes tienen INT_MAX el entero mas alto posible, de esta forma, cuando comparemos $MenorArista$ con la primer arista que veamos, esta ultima siempre tendra un costo menor para poder reemplazar.

El algoritmo se vale de tres funciones auxiliares: $NoFormaVerticesDeGrado3$, $NoFormaCircuito$ y $actualizarEstructura()$. Para la implementación de $NoFormaVerticesDeGrado3$ optamos por un arreglo que en la posición i tiene guardado el grado del i-esimo vértice y de esta forma puede verificar si se forman vértices de grado tres y actualizar esa información en $O(1)$. Esto también nos da complejidad es-

pacial de $O(n)$, lo cual no afecta nuestra complejidad espacial de $O(n^2)$ u $O(m)$ (donde m es la cantidad de aristas) que teníamos por la matriz que utilizamos para representar el grafo.

Para la implementación de $NoFormaCircuito$, optamos por un vector de punteros a vectores donde en el elemento i es puntero al vector de relación del nodo i. Por cuestiones de espacio, no incluiremos aquí el pseudocódigo de ésta función auxiliar, pero dejamos una ilustración de un ejemplo para mostrar cómo opera.

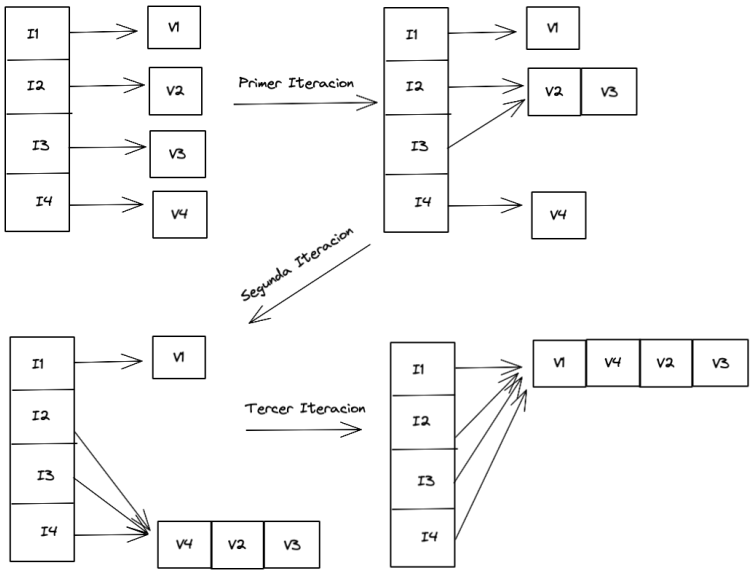

Figura 6: Ejemplo de estructura de relaciones.

Con esta estructura, cada vez que vinculamos dos nodos, unimos los dos vectores apuntados por ellos y luego hacemos que todos los nodos relacionados apunten a la nueva lista. Como vemos luego del primer paso, conseguimos nuestra primer arista que vincularía el segundo con el tercer vértice. Entonces se genera un vector para el cual ambos vértices estén incluidos y luego ambos apunten. De esta forma para saber si dos vértices están vinculados (y, por ende generando un circuito unidos con un vértice) solo debemos corroborar los punteros guardados en el vector padre lo cual se hace en tiempo constante. En caso de ser el mismo, sabremos que están relacionados. Actualizar esta estructura, tiene complejidad lineal ya que hay que hacer una unión de ambos vectores y luego actualizar el puntero de todos los vértices relacionados. Esto se hace en $actualizarEstructura$, que aporta una complejidad de $O(n)$ ya que el vector de punteros tiene longitud n (cantidad de vértices) y la suma de la longitud de todos los vectores apuntados también es igual a n. Esto no nos afecta la complejidad total pues, como vimos con el arreglo de grados, es menor que la complejidad que teníamos originalmente.

La complejidad del algoritmo es $O(n * (for\ interno + ActualizarEstructura))$. A su vez, el ciclo "for" interno del *while* itera m veces con un cuerpo interno con complejidad de $O(1)$, con m siendo la cantidad de aristas. Ya que $O(ActualizarEstructura)$ es $O(n)$, la complejidad de éste algoritmo es $O(n * (m + n))$.

2.3. Búsqueda Local

Los algoritmos de búsqueda local o mejoramiento, buscan encontrar un óptimo local a partir de una solución inicial, con la esperanza de que ésta solución sea también un óptimo global o de valor cercano al óptimo. Para éste método implementamos una búsqueda local para vecindades 2-opt.

Este método parte de tener un ciclo cualquiera con su costo y modificar el ciclo hasta encontrar uno de costo mínimo. Este algoritmo cambia aristas de a pares para formar nuevos ciclos y así poder tener un ciclo similar (vecino) para comparar. El algoritmo tiene como variables de entrada a la matriz de incidencias del grafo y un entero l, el cual se usa para limitar la cantidad de iteraciones del ciclo del

algoritmo.

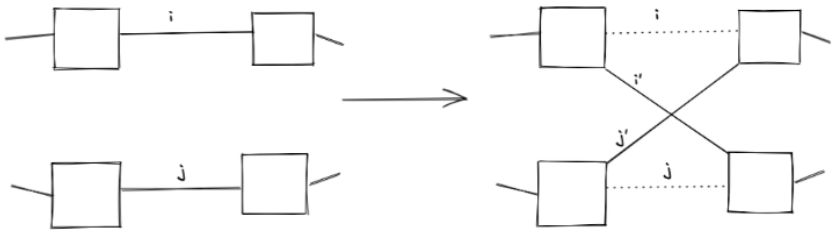

Figura 7: Ejemplo de como el algoritmo cambia el ciclo para encontrar un vecino.

Algorithm 4 Búsqueda Local*(vector<vector<int>> incidencias, int l)*

```
1: ciclo ← agm(incidencias)
2: costo_viejo ← ciclo.costo
3: it ← 0
4: while it < l do
5:    for arista i, arista j ∈ incidencias do
6:       costo_nuevo ← costo_viejo − i − j + i' + j'
7:       if costo_nuevo < costo_viejo then
8:          ciclo.costo = costo_nuevo
9:          swap(ciclo, i, j)
10:      end if
11:   end for
12:   it ← it + 1
13: end while
14: return ciclo
```

Algorithm 5 swap*(vector<int> &ciclo, int i, int j)*

```
1: for k ∈ range((j − i)/2) do
2:    temp ← ciclo[i + 1 + k]
3:    ciclo[i + 1 + k] ← ciclo[j − k]
4:    ciclo[j − k] ← temp
5: end for
```

Este algoritmo agarra un par de aristas por iteración del for, luego si no hay intersección entre los vértices que vinculan, calculamos el costo del nuevo ciclo, en el cual le restamos el costo de las aristas seleccionadas y le sumamos el costo de las nuevas aristas. Finalmente, si el nuevo costo es menor al costo anterior nos quedamos con el nuevo ciclo utilizando la función $swap$.

La función $swap$ recibe el ciclo y dos aristas, editando el ciclo cambiando estas dos aristas y todos los nodos intermedios con el fin de corregir el orden ya que nuestro ciclo esta representado por un vector de nodos, no de aristas. La complejidad de esta función auxiliar es de $O(n)$ ya que invierte parte del vector que forma al ciclo.

La complejidad del algoritmo es de $O(m * n + l * m^2 * n)$ u $O(l * m^2 * n)$ donde n son la cantidad de nodos, m la cantidad de aristas y l la cantidad de iteraciones que hace esta búsqueda. El $m * n$ viene del algoritmo de agm que nos da nuestro ciclo original. Esto es por que el algoritmo $swap$ tiene complejidad de $O(n)$ y la tenemos que correr por cada par de las m aristas. Todo eso una cantidad de l veces.

2.4. Tabú Search

Tabú Search, similar a búsqueda local, explora la vecindad a partir de un ciclo inicial para intentar de encontrar un nuevo mínimo. Sin embargo, este intenta solucionar el problema que tiene búsqueda local. Este nuevo algoritmo toma vecinos generados al azar, con el fin de explorar otras regiones del espacio de soluciones y así intentar de encontrar el mínimo absoluto en vez de un mínimo local. Por esto este algoritmo correrá hasta que no encuentre mejora en vez de correr hasta llegar a un mínimo local. Este algoritmo utiliza una lista tabu (la cual le da el nombre al algoritmo). Esta lista, la cual llamaremos memoria en el algoritmo, contendrá los mejores ciclos ya visitados para prohibir la repetición de ciclos a visitar. Esta lista no sera de longitud infinita sino que tendrá una longitud fija predeterminada por parámetros de entrada y utilizara un criterio de first in first out (FIFO) para determinar con cual ciclo visitado se queda luego de haberse llenado.

Algorithm 6 Tabú Search*(vector<vector<int>> incidencias, int l, int k, int v)*

```
ciclo ← Heuristica_AGM(incidencias)
mejor_ciclo ← ciclo
memoria ← vector(size = |incidencias|*k / 100)
iterator ← 0
memory_index ← 0
while iterator < l do
    vecinos ← ObtenerSubVecindad(incidencias, ciclo, v)
    ciclo ← ObtenerMejor(vecinos)
    Guardar(ciclo)
    if ciclo.costo < mejor_ciclo.costo then
        mejor_ciclo ← ciclo
    end if
    iterator ← iterator + 1
end while
return mejor_ciclo
```

Algorithm 7 ObtenerSubVecindad*(vector⟨vector⟨int⟩⟩ incidencias, pair⟨vector⟨int⟩, int⟩ ciclo, int v)*

```
ciclos_vecinos ← vector⟨pair⟨vector⟨int⟩, int⟩⟩(∅)
costo_viejo ← ciclo.second
for arista i, arista j ∈ incidencias do
    ciclo.second ← costo_viejo − i − j + i' + j'
    swap(ciclo, i, j)
    ciclos_vecinos.append(ciclo)
    swap(ciclo, i, j)
end for
cantidad ← ciclos_vecinos*v / 100
shuffle(ciclos_vecinos)
ans ← (ciclos_vecinos, cantidad)
return ans
```

Este algoritmo recibe tres parámetros adicionales: l, k, v, donde l es la cantidad de ciclos que debe hacer el algoritmo antes finalizar, k define el tamaño de la memora (funciona como porcentaje de la cantidad de nodos) y v es el porcentaje de vecinos que tomaremos para explorar de manera al azar. Notemos que la memoria y el memory index están como variables globales.

Este algoritmo utiliza la heuristica de arbol generado minimo base como ciclo inicial. A partir de este se generan vecinos con $obtenerSubVecindad$, este algoritmo genera todos los vecinos de la misma forma que búsqueda local y agrega todos los vecinos a $ciclos_vecinos$ la cual después mezclamos con shuffle y nos quedamos con un $v\,\%$. Este algoritmo tiene una complejidad temporal de $O(m^2 * n)$ con m siendo la cantidad de aristas y n la cantidad de vértices que irán intercambiándose en cada vecino. Esta

Algorithm 8 ObtenerMejor*(vector<vector<int>> incidencias, vector⟨pair⟨vector⟨int⟩, int⟩⟩ vecindario)*

```
mejor_ciclo ← ⟨vector⟨int⟩(∅), INT_MAX⟩
mejor_ciclo_total ← ⟨vector⟨int⟩(∅), INT_MAX⟩
i ← 0
while i < |vecindario| do
    if vecindario[i].second < mejor_ciclo.second and vecindario[i] ∉ memoria then
        mejor_ciclo ← vecindario[i]
    end if
    if vecindario[i].second < mejor_ciclo_total.second then
        mejor_ciclo_total ← vecindario[i]
    end if
    i ← i + 1
end while
if mejor_ciclo.second = INT_MAX then
    return mejor_ciclo_total
end if
return mejor_ciclo
```

Algorithm 9 Guardar*(pair<vector<int>, int> ciclo)*

```
memoria[memory_index] ← ciclo
memory_index ← (memory_index + 1) %|memory|
return
```

complejidad proviene de que se generan un vecino para cada par de las m aristas y generar el vecino en si tiene complejidad de $O(n)$. El shuffle tiene complejidad de $O(|vector|)$ la cual es m^2. Sabemos que como v es esta acotado por cien, este no afectara la complejidad.

Luego en $obtenerMejor$ generamos dos pares (vector con un entero) donde el vector es vació y el entero es el mas alto posible. De esta forma podemos guardar en $mejor\_ciclo$ el mejor ciclo que no este en memoria y en $mejor\_ciclo\_total$ el mejor ciclo de los que entraron como parámetro. De esta forma al final de la búsqueda del mejor, si el $mejor\_ciclo$ tiene el mas alto posible, sabemos que todos los ciclos que nos llegaron estaban en memoria (ya que no se sobrescribió) y podemos devolver el $mejor\_ciclo\_total$. De esta forma nos quedamos con el ciclo de menor costo de todos los vecinos que no este guardado en memoria (en caso de que todos estén en memoria, devolvemos el de menor que se recibió de la vecindad), lo guardamos en memoria ($O(n)$) y si este es mas barato que el ciclo que teníamos, lo reemplazamos y repetimos. Esta función solo analiza los m ciclos que nos devolvió la función anterior, y luego compara todos con memoria quedándose con el mas barato. Esta función, por ende, tendrá una complejidad de $O(m * n^2 * k)$ ya que tenemos m vecinos de longitud n en una memoria de tamaño $\frac{n*k}{100}$

La complejidad del cuerpo del while queda $O(m^2 * n + m * n^2 * k + n)$ u $O(m^2 * n^2 * k)$. El while siempre iterara l veces, por ende, tenemos una complejidad final de $O(n * m + l * m^2 * n^2 * k)$ u $O(l * m^2 * n^2 * k)$ con el primer $n * m$ originando de la complejidad de la heuristica basada en AGM.

3. Experimentación

En ésta sección presentamos los resultados de las experimentaciones usando las distintas heurísticas y metaheurísticas descriptas en las secciones previas. Dividimos la sección en dos subsecciones: en la primer parte realizamos experimentos generales de rendimiento en grafos aleatorios, mientras que en la segunda subsección experimentamos de calibración de parámetros usando las instancias obtenidas en http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/index.html.

3.1. Hipótesis

Las hipótesis iniciales que vamos a tener en cuenta para los experimentos son los siguientes:

Hipotesis de calibración :

1. Esperamos que el gap se disminuye cuantas más iteraciones haga el algoritmo.
2. Esperamos que, para tabú search, el gap mínimo se realice con un porcentaje de vecinos al azar muy por encima del $\%1$ y muy por debajo del $\%100$. Esto es porque muy pocos vecinos al azar dificultan la detección de mínimos globales, mientras que demasiados vecinos al azar vuelve demasiado caótica la exploración.
3. Esperamos que a medida que aumente la longitud de la tabla tabú, el gap disminuya. Esto es porque cuanto mayor la tabla, más precisa será la exploración del método.

Hipotesis generales :

1. Búsqueda local y tabú search son métodos temporalmente más caros que heurística de árbol generador mínimo y golosa.
2. Esperamos que el algoritmo con mejor rendimiento sea el de tabú search, seguido por búsqueda local. Esperamos que estos dos tengan, a su vez, un mejor rendimiento que la heurística golosa y la del árbol generador mínimo.
3. Esperamos que al aumentar las iteraciones en búsqueda local y tabu search, obtengamos resultados mejores.

3.2. Experimentos de Calibracion

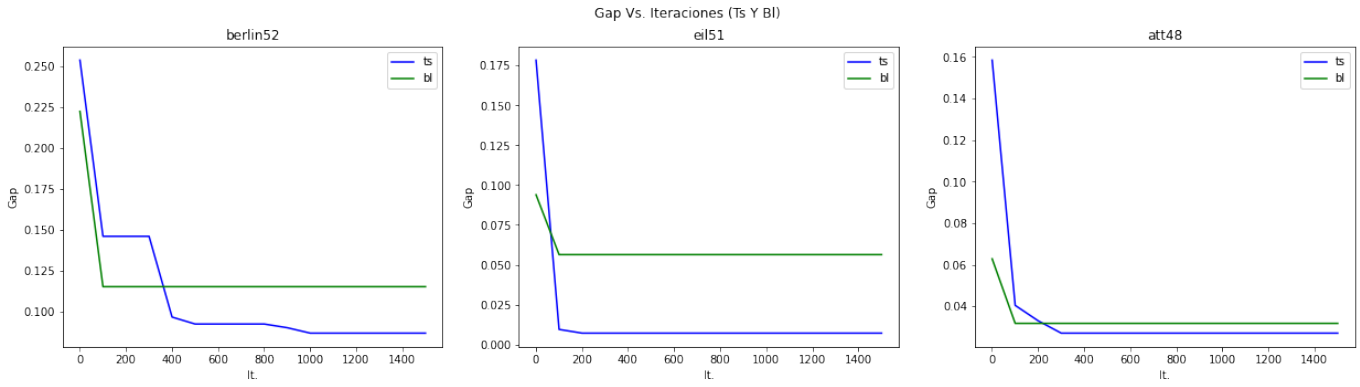

Figura 8: Calibracion de iteracion en tabú search y búsqueda local (vecindad: 40 % - tam tabla : 100 %)

En éstos experimentos estudiamos los métodos de tabú search y búsqueda local, enfocándonos en cada unos de los parámetros de entrada. Para ello, empezamos realizando experimentación de calibración con respecto a la cantidad de iteraciones. En la figura 8 se muestran tres gráficas con experimentaciones sobre tres instancias conocidas ('berlin52', 'eil51' y 'att48') en las cuales se puede visualizar cómo disminuye el gap de los resultados de búsqueda local y tabú search a medida que aumentan las iteraciones. Es interesante notar cómo el método de tabú search va alcanzando mínimos locales a medida que avanzan las iteraciones. Un ejemplo de esto último se ve muy claramente en el primer gráfico de la Figura 8, lo cual comprueba la primer hipótesis de calibración. Más aún, no pudimos observar una mejora substancial del rendimiento de tabú search a partir de las 1000 iteraciones.

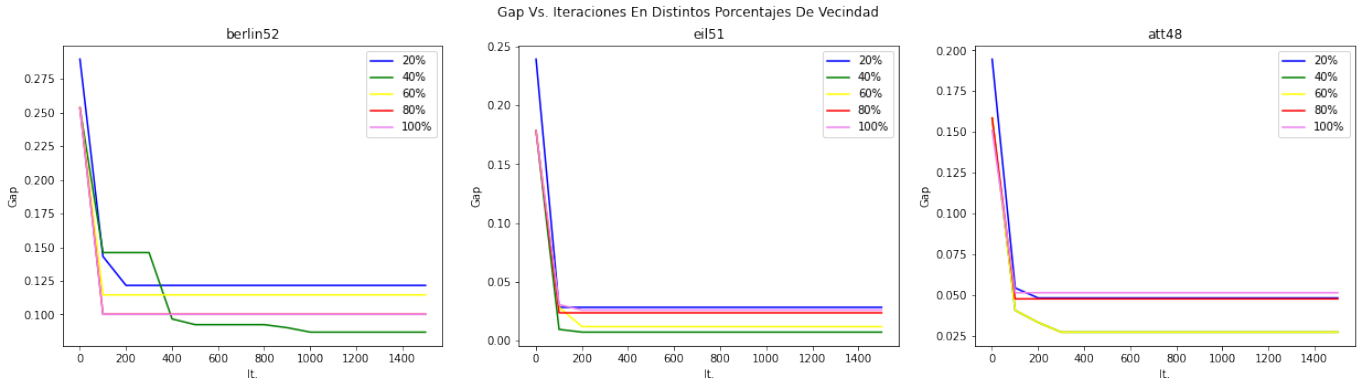

Figura 9: Calibracion del porcentaje de vecindad en ts (iteraciones:1000 - tam tabla : 100 %)

En la Figura 9 se visualizan en las mismas instancias la manera en la que disminuye el gap usando el método de tabú search si se emplean distintos porcentajes de vecinos al azar de un ciclo específico. Tal y como planteábamos en nuestra hipótesis, los menores gaps se dieron en aquellas corridas con porcentajes alejados tanto del 1 % como del 100 %. Consideramos que el mejor porcentaje es 40 % ya que vemos que este valor obtiene el gap mas bajo en las dos primeras imágenes y uno bastante bajo en la tercera.

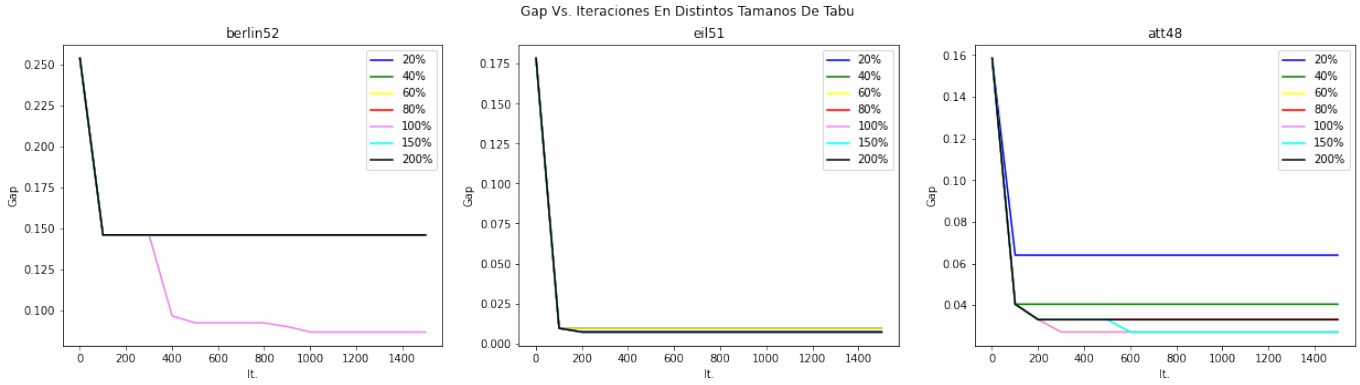

Figura 10: Calibracion del porcentaje de vecindad en ts (iteraciones:1000 - vecindad : 40 %)

Notar que la primer gráfica de Figura 10 muestra un gap considerablemente mejor que el resto de los vecinos. Creemos que esto se debe a una anomalía por el componente azaroso que tiene el método, dado que corrimos el mismo algoritmo con una semilla distinta sobre la misma instancia y dicha anomalía no estaba presente. Aún así, consideramos el parámetro $\%100$ como el mejor al compararlo con los otros gráficos.

3.3. Experimentos con instancias random

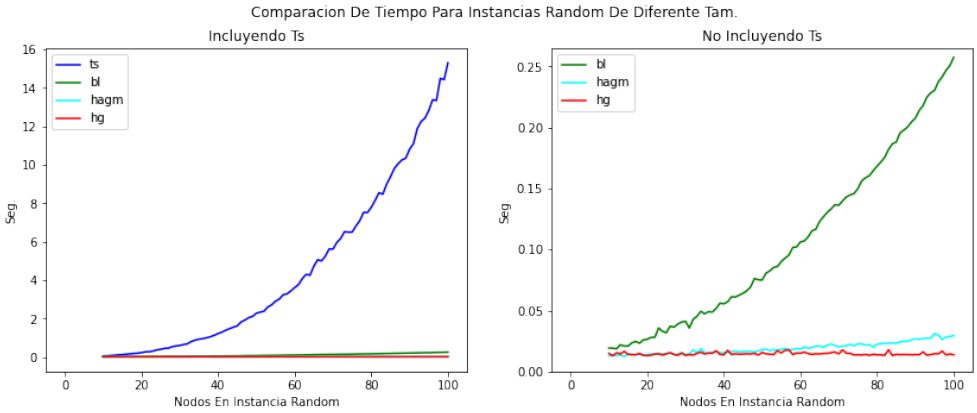

Figura 11: Experimento en instancias random con de 10 a 100 nodos (iteraciones:1000 - vecindad : 40 % - tam tabla : 100 %)

En ésta sección se analiza la eficiencia de cada uno de los métodos en la práctica y su correlación con la cota teórica calculada en la Sección 2. Para ésto, se ejecutan instancias de grafos al azar donde la cantidad de nodos está entre 10 y 100 con cada uno de los métodos y se grafican sus resultados en la Figura 11. Para mayor claridad, figuran dos gráficos: uno que incluye el método de tabú search y otro que no. Esto es porque, como ya fue demostrado antes, la complejidad de tabú search es notablemente peor que el resto, y es necesario dos gráficos para que se pueda apreciar la complejidad práctica de cada uno. En la Figura 12 figuran dos gráficos donde se analiza el costo de cada resultado en función del tiempo de ejecución utilizado por cada uno de los métodos. Como se puede ver, aunque tabú search sea mas costoso temporalmente, tiene mejor rendimiento que los otros tres algoritmos.

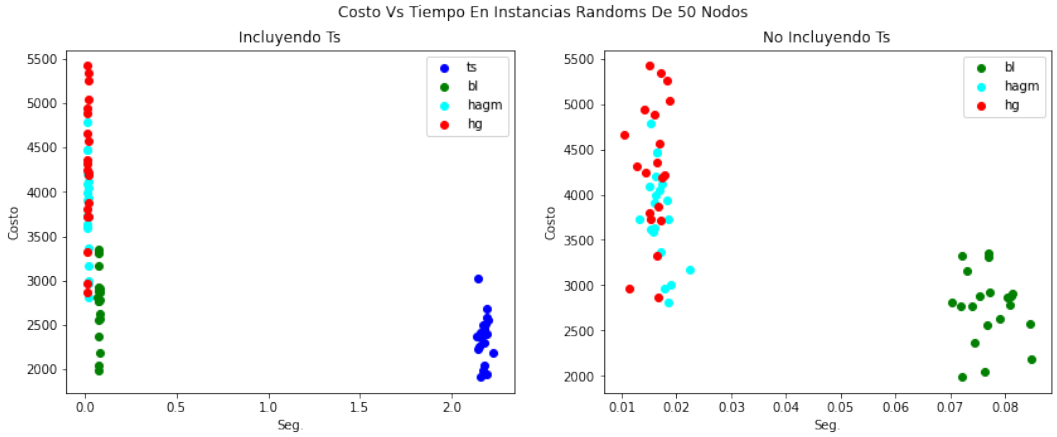

Figura 12: Experimento en instancias random con 50 nodos (iteraciones:1000 - vecindad : 40 % - tam tabla : 100 %)

En la Figura 12 podemos la relación entre el costo obtenido y el tiempo tomado para ejecutar en 20 instancias al azar de 50 nodos. Se opto por el uso de dos gráficos para apreciar mejor las escalas. El algoritmo tabú search obtuvo el mayor tiempo pero también costos menores a todos los demás en la mayoría de los casos. Esto tiene sentido como explicamos antes por la complejidad y por el hecho de poder llegar a mas mínimos locales que los que obtiene búsqueda local.

El algoritmo de búsqueda local mostró ser mucho mas rápido, y con costos no tan chicos como tabú search pero muy similares. Esto también se debe a que tiene menor complejidad y a que mejora un circuito obtenido usando heurística de árbol generador mínimo.

Por ultimo, la heurística de árbol generador mínimo y heurística golosa muestran tener mucho menor tiempo debido que son heurísticas golosas. No obtienen tan bajo costo como búsqueda local y tabú search. La heurística de árbol generador mínimo tiene menor costo que heurística golosa en la mayoría de los casos, debido a que esta basado en kruskal.

3.4. Experimento con instancias particulares

Las instancias utilizadas para la experimentación son bier127,dantzig42, gr96, lin105, ulysses22. De las cuales conocemos su costo real, por lo que podemos calcular el gap.

Podemos observar en la figura 13 que tabú search obtuvo el menor gap en cuatro de los cinco casos. Por su parte, búsqueda local obtuvo un gap muy similar al de tabú search y uno menor a éste en un caso. La heurística de árbol generador mínimo obtuvo el tercer menor valor en tres de los cinco casos mientras que heurística golosa fue tercero en dos casos. Esto apoya nuestra segunda hipótesis las hipótesis generales del principio de la sección.

Tabú search obtiene el menor gap la mayoría de las veces debido a que puede salir de mínimos locales que cae búsqueda local. Búsqueda local es el segundo menor porque mejora un circuito de árbol generador mínimo y llega a mínimos locales.

La heurística de árbol generador mínimo, al basarse en Kruskal, obtiene mejores resultados que la heurística golosa.

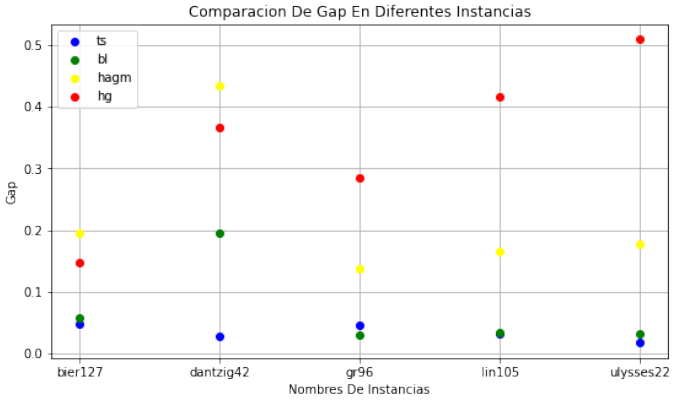

Figura 13: Experimento con instancias particulares (iteraciones:1000 - vecindad : 40 % - tam tabla : 100 %)

4. Conclusión

En este trabajo se presentan 4 heurísticas para resolver el problema de viajante de comercio: dos heurísticas y dos metaheurísticas. Para cada una de ellas interesa medir principalmente su tiempo de ejecución y su efectividad de resolver el problema. La heurística golosa es muy efectiva en términos de velocidad, pero en muchas instancias demuestra que su resultado no es tan cercano al óptimo como las otras tres heurísticas. Una mejora de éste algoritmo es la heurística de árbol generador mínimo, que logra una gran rapidez de corrida y un mejor rendimiento que el de heurística golosa. Sin embargo, éste último demuestra ser no tan eficiente como los métodos metaheurísticos. El método de tabú search demuestra ser un métodos más costoso computacionalmente, aunque su rendimiento es mucho mejor en cuanto a resultado. Una mejora para éste método es el de búsqueda local, aunque en la experimentación se ve que la optimalidad es mucho menor. Finalmente, en líneas generales, las hipótesis descriptas al principio de la Sección 3 fueron comprobadas afirmativamente.