DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Heurísticas y Metaheurísticas

Algoritmos y estructuras de datos III

Primer Cuatrimestre de 2021

Integrante	LU	Correo electrónico
Leandro Rodriguez	521/17	leandro21890000@gmail.com
Andres Mauro	39/17	sebaastian_mauro@live.com
Leo Mansini	318/19	leomansini2000@gmail.com
Camilo Semeria	818/97	csemeria@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

Resumen

En el presente trabajo se implementarán algoritmos heurísticos y metaheurísticos para el problema de TSP (*Travelling Salesman Problem*) hallando ciclos hamiltonianos en grafos completos. También se realizaran experimentos en cada algoritmo para conocer su complejidad, cómo se comporta con distintas instancias, y cuales son los parámetros óptimos para cada algoritmo (en el caso de que los haya).

Índice

1. Introducción

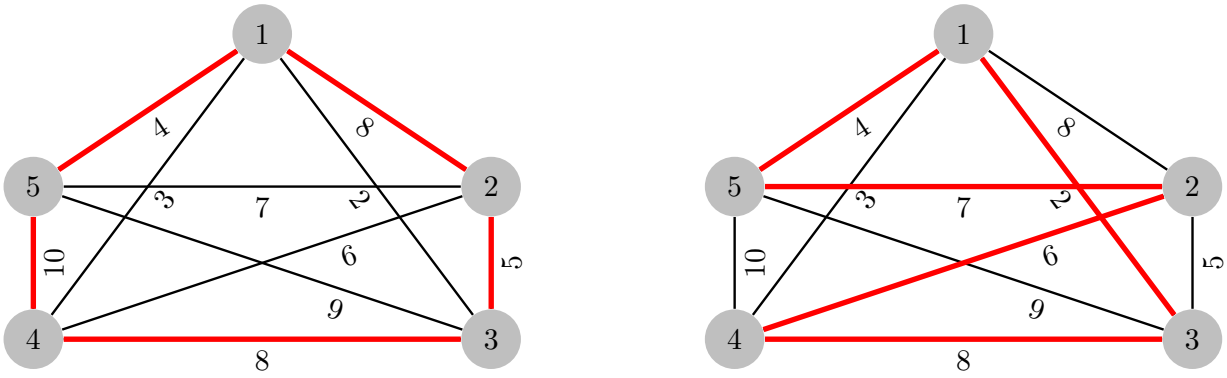
El Problema del Viajante de Comercio[1] (TSP) es uno de los problemas fundamentales de Optimización Combinatoria. En este campo se trata se resolver un problema de la mejor manera, logrando una solución lo más óptima posible de entre el conjunto de soluciones del problema. En el caso de TSP, una forma de plantear el problema es haciendo la siguiente pregunta: ”Dada una lista de ciudades, las distancias entre todas ellas, y un comerciante. ¿Cuál es la manera más eficiente para el comerciante de recorrer todas las ciudades exactamente una vez, empezando y terminando en el mismo lugar?”. La manera en la que se encaró el problema en este trabajo es utilizando un grafo completo, pesado y no dirigido, en donde los nodos son las ciudades y los pesos de las aristas son las distancias entre las ciudades que conecta. De esta manera, la solución del TSP es un ciclo hamiltoniano de costo mínimo en ese grafo (definimos el costo de un ciclo como la suma de los costos de las aristas en ese ciclo).

A continuación se dará un ejemplo de búsqueda de ciclo hamiltoniano en un grafo.

Se tiene el grafo completo de cinco nodos como los de las Figuras 1a y 1b. Si tomamos a las aristas marcadas en rojo como pertenecientes a un ciclo hamiltoniano, y describimos un ciclo como el orden en el que recorre los nodos, tenemos los siguientes ciclos:

- Ciclo 1: 1, 2, 3, 4, 5, 1
- Ciclo 2: 1, 5, 2, 4, 3, 1

Vemos que ambos ciclos son hamiltonianos: los dos recorren todos los nodos exactamente una vez y empiezan y terminan en el nodo 1. Ahora bien, ¿cuál es su peso? Debemos calcular la suma de las aristas que componen cada ciclo, y así encontramos que el ciclo 1 tiene un peso de 35, mientras que el ciclo 2 tiene un peso de 27. Así, una solución como el ciclo 2 es mejor que una como el ciclo 1, si bien las dos son válidas.



(a) Ciclo 1.

(b) Ciclo 2.

Figura 1: Ciclos hamiltonianos para grafo completo de 5 nodos.

A continuación presentamos algunos **problemas de aplicación** frecuentes en la vida real cuya resolución puede ser modelada como una solución particular del problema de TSP:

- Reparto de productos. Mejorar una ruta de entrega para seguir la más corta.
- Transporte. Mejorar el recorrido de caminos buscando la menor longitud.
- Robótica. Resolver problemas de fabricación para minimizar el número de desplazamientos al realizar una serie de perforaciones en un circuito impreso.
- Horarios de transportes laborales y/o escolares. Estandarizar los horarios de los transportes es claramente una de sus aplicaciones, tanto que existen empresas que se especializan en ayudar a las escuelas a programarlos para optimizarlos en base a una solución del TSP.

[1] https://en.wikipedia.org/wiki/Travelling_salesman_problem

- Inspecciones a sitios remotos. Ordenar los lugares que deberá visitar un inspector en el menor tiempo.
- Turismo y agencias de viajes. Aun cuando los agentes de viajes no tienen un conocimiento explícito del Problema del Agente Viajero, las compañías dedicadas a este giro utilizan un software que hace todo el trabajo. Estos paquetes son capaces de resolver instancias pequeñas del TSP.
- Inspecciones a sitios remotos. Ordenar los lugares que deberá visitar un inspector en el menor tiempo.
- Secuencias. Se refiere al orden en el cual n trabajos tienen que ser procesados de tal forma que se minimice el costo total de producción.

El Problema del Viajante de Comercio no tiene algoritmos polinomiales para su resolución (para cualquier grafo), y por consecuente tampoco los hay para la búsqueda de ciclos hamiltonianos mínimos en los grafos que se modelaron para TSP. Por esa razón para obtener soluciones en tiempo polinómico se recurrió a métodos heurísticos. Si bien estas soluciones son ciclos hamiltonianos, no se garantiza que sean mínimos, si bien cada algoritmo aproxima la solución a la de menor costo posible.

Se desarrollaron tres heurísticas diferentes. Dos de ellas son heurísticas constructivas golosas, esto es, algoritmos que van construyendo la solución iterativamente, tomando una decisión en base al contexto local de cada iteración, sin tener en cuenta todo el problema (en este caso, no tienen en cuenta la totalidad del grafo). La tercera heurística esta basada en un árbol generador mínimo, haciendo uso de un recorrido por profundidad.

Además, se utilizaron metaheurísticas, algoritmos que tendrán como entrada una solución al problema, e intentarán mejorar esa solución buscando en el "vecindario" de ella (más sobre esto en la sección 4). Se desarrollaron dos versiones de la metaheurística Búsqueda Tabú, cada una con tipos de elementos distintos en la Lista Tabú.

2. Heurísticas Constructivas Golosas

Las heurısticas constructivas golosas son métodos que permiten construir una solución válida para una instancia de un problema. No garantiza encontrar la mejor solución e incluso podría generar la peor solución válida de todas. Por su facilidad de diseño e implementación son utilizadas frecuentemente.

2.1. Heurística Vecino más cercano

Es un algoritmo goloso que toma un grafo representado como una matriz de adyacencias y partiendo del primer vértice (por convención el vértice 1) calcula un circuito hamiltoniano agregando en cada iteración el vértice adyacente más cercano aún no recorrido, hasta completar el ciclo hamiltoniano.

Algorithm 1 Algoritmo de vecino mas cercano para TSP

```
1:  function heuristica_vmc(Grafo G)
2:      verticeActual ← G[1];                                          ▷ O(1)
3:      camino ← crear_vector_vacio();                                 ▷ O(n)
4:      agregar(camino, verticeActual);                                ▷ O(1)
5:      while camino.size() < G.cantidadNodos() do                     ▷ O(n)
6:          verticeActual ← verticeMasCercanoSinVisitar(verticeActual, camino);   ▷ O(n)
7:          agregar(camino, verticeActual);                            ▷ O(1)
8:      end while
9:      circuito← cerrarCircuito(camino);                              ▷ O(1)
10:     return circuito;                                               ▷ O(1)
11: end function
```

Vemos en el algoritmo 1 que $verticeMasCercanoSinVisitar$ es una función que devuelve el vértice adyacente más cercano al vértice actual que no haya sido agregado en alguna de las iteraciones anteriores.

$cerrarCircuito$ es una función que cierra el circuito de vertices agregando el primero del camino al final del mismo, para finalmente devolver el circuito formado.

2.1.1. Complejidad

Las primeras lineas del algoritmo tendrían complejidad $O(1)$, si es que la variable **camino** se inicializa como una lista y no como un vector de n posiciones, pero aún si **camino** no fuese inicializada como vector, sería irrelevante para la complejidad, pues encontraremos que ella resulta ser $O(n^2)$. Para agregar todos los vértices, es necesario y suficiente que se ejecuten n iteraciones del while, por lo cual su complejidad teniendo en cuenta lo que está adentro de él será $O(n)$ multiplicado por $O(interiordelwhile)$.
Evaluemos lo que hay dentro del while: agregar es una función que coloca en la última posición de **camino** el vértice actual, y por ser en la implementación un vector, se ejecuta en $O(1)$.
La función $verticeMasCercanoSinVisitar$ recorre todos los vértices adyacentes al actual. La peor situación en órdenes de complejidad en el caso general es cuando el vértice actual es adyacente a todos los del grafo, y en particular para el presente TP esto siempre ocurre ya que se utilizan grafos completos, resultando así en $O(n-1)$ al chequearlos todos. Por esto, la complejidad del while es $O(n) * O(n)$, que es igual a $O(n^2)$, siendo esta última la complejidad de la heurística.

2.2. Heurística Arista más corta

Esta técnica se basa en hacer uso del concepto detrás del algoritmo de Kruskal al revisar si se forma o no un ciclo antes de agregar aristas, pero trasladado al problema de encontrar un circuito hamiltoniano. La heurística selecciona en cada iteración la arista de menor peso tal que ningún vértice tenga grado mayor a 2 y no forme ciclos (antes de agregar todos los vértices), por lo cual su comportamiento también es goloso al igual que la anterior. En **AristasOrdenadas** se guardan las aristas del grafo ordenadas de menor a mayor según su peso. **AristasAgregadas** es el vector de aristas, en el cual se agregan las aristas que correspondan para crear el circuito hamiltoniano. Se analiza si agregar o no cada arista del grafo, respetando su orden en **AristasOrdenadas**, hasta haber agregado $n-1$ aristas para luego agregar la ultima arista (que es la que cierra el circuito) con la función cerrarCircuitoDeAristas.
Se transforma el circuito de aristas en circuito de nodos (ya que por convención se devuelve el circuito de nodos no de aristas) y se guarda en **circuitoNodos**. Finalmente, se devuelve **circuitoNodos**.

Algorithm 2 Algoritmo de arista mas corta para TSP

```
1: function heuristica_amc(Grafo G)
2:     AristasOrdenadas ← G.aristasOrdenadas();                          ▷ O(m^2)
3:     cantAristasAgregadas ← 0;                                         ▷ O(1)
4:     AristasAgregadas ← crear_vector_vacio();                          ▷ O(1)
5:     i ← 0;                                                            ▷ O(1)
6:     while cantAristasAgregadas < G.cantidadNodos() − 1 and i<AristasOrdenadas.size() do ▷ O(m)
7:         if puedoAgregar?(AristasOrdenadas[i],AristasAgregadas) then   ▷ O(m)
8:             agregar(aristasAgregadas, AristasOrdenadas[i]);           ▷ O(1)
9:             cantAristasAgregadas ++;                                  ▷ O(1)
10:        end if
11:        i++;                                                          ▷ O(1)
12:    end while
13:    circuitoAristas ← cerrarCircuitoDeAristas(aristasAgregadas);      ▷ O(m^2)
14:    circuitoNodos ← CrearCircuitoNodos(circuitoAristas)               ▷ O(n^2)
15:     return circuitoNodos;                                            ▷ O(1)
16: end function
```

2.2.1. Complejidad

El ordenamiento de las aristas tiene un costo de $O(m^2)$ ya que se ordenan con el algoritmo selection sort. En el ciclo while, se itera m veces a lo sumo, y la operación mas costosa es $puedoAgregar?$ la

cual se encarga de verificar que ningun vertice de la arista considerada tenga grado mayor que 2 (en **aristasAgregadas**) ni forme un ciclo, esto toma $O(m)$. Finalmente, cerrar el circuito de aristas y crear el de nodos se hace en $O(m^2) + O(n^2)$=$O(m^2)$.
Por todo lo mencionado anteriormente la complejidad final de la heurística es$O(m^2)$. Lo que es igual a $O(n^4)$ ya que sabemos que los grafos con completos y por ende $m = n(n1)$.

2.3. Caso patológico

Para estos algoritmos (HVMC y HAMC) comparten un caso patológico, siempre que la última arista (con la que se cierra el ciclo) tenga un valor superior al peso total del circuito óptimo.
De esta forma se obtiene una solución siempre mayor a la ideal, y como siempre se parte del primer vértice se pueden construir circuitos tan lejos del óptimo como uno quiera. Existen familias de grafos que se pueden construir para que el circuito resultante se acerque al peor caso posible, estas son aquellas que fuerzan al algoritmo a que cierre el circuito con un una arista determinada esto explota la vulnerabilidad del algoritmo goloso, que una vez tomada una decisión, no vuelve a considerar otro camino posible.
Construir estas instancias para mapas grandes no es trivial y es poco frecuente en el mundo real por lo cual no se experimentará directamente sobre ellas. La figura 2 muestra con un ejemplo como son este tipo de instancias.

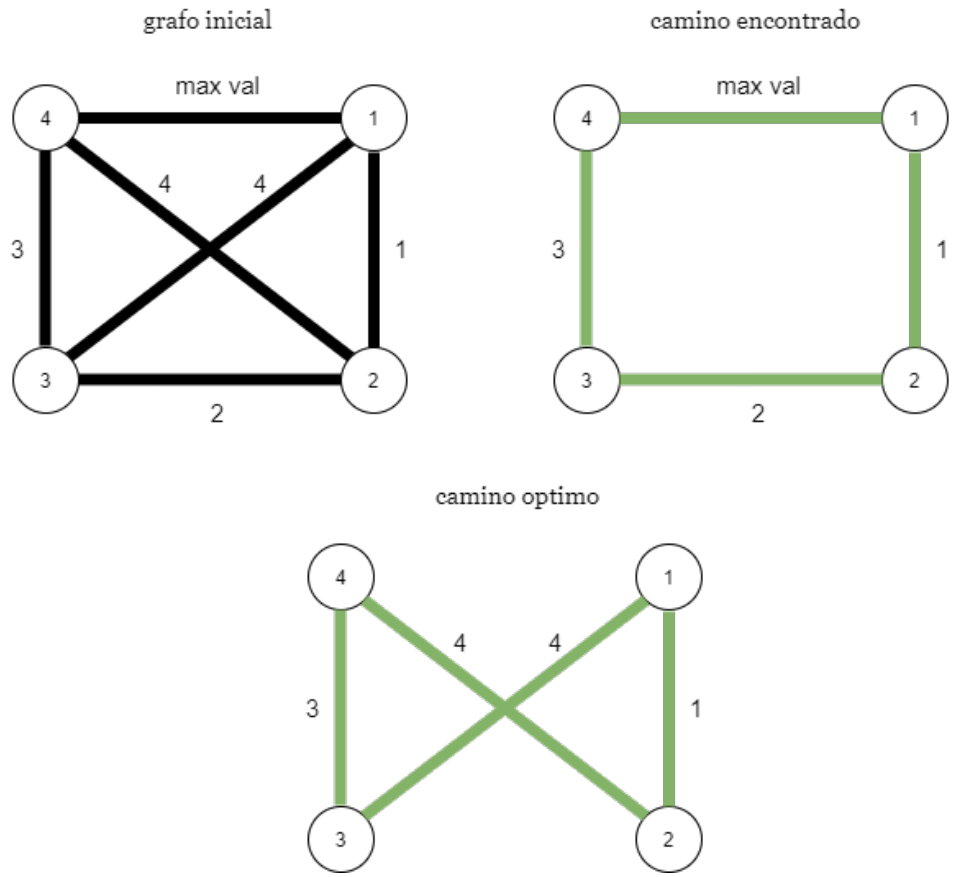


Figura 2: Ejemplo de grafos patológicos para HAMC y HVMC. El max valor puede ser tan grande como uno quiera.

3. Heurística basada en Árbol Generador Mínimo

La heurística de árbol generador mínimo (AGM) se basa en construir un AGM y luego recorrerlo utilizando DFS tomando nota del orden en que los nodos se visitan. El camino solución será aquel definido por esa lista el cual se cierra como circuito incorporando el eje que va desde el último nodo al primero.
Cómo método para la construcción del AGM utilizamos el algoritmo de PRIM, el cual se basa en partir de la arista de menor peso y agregarla como primera arista del AGM para luego buscar en cada iteración la arista de menor peso que tenga uno de sus extremos en un nodo ya presente en el árbol en construcción y otro no. El algoritmo termina cuando todos los nodos son parte del árbol. Este algoritmo tiene la particularidad de que el circuito encontrado tiene a lo sumo dos veces el peso del óptimo cuando el grafo es euclídeo, debido que se sabemos que los ejes que conforman un AGM constituyen la forma más eficiente (en peso = distancia) de recorrer un grafo y que por desigualdad triangular el eje que une dos nodos puede ser a lo sumo tan pesado como la suma de los pesos de los ejes que conectan los mismos nodos y son parte del AGM.

Algorithm 3 Algoritmo de heurística AGM

1: **function** $heuristicaAGM(Grafo\ G)$
2: Generar un AGM A de G ▷ $O(n^4)$
3: Recorrer A usando DFS agregando a una lista L los nodos de A en el orden en que son visitados ▷ $O(n)$
4: Cerrar el camino descripto en L con la arista que va del último al primer nodo para obtener un circuito C ▷ $O(1)$
5: **return** C;
6: **end function**

Algorithm 4 Algoritmo de PRIM

1: **function** $PRIM(Grafo\ G)$
2: Generar una lista L de aristas de G ordenadas por peso ▷ $O(n^4)$
3: creo un grafo G' con los vértices de G y sin ninguna arista ▷ $O(n^2)$
4: Crear una lista de vértices ya agregados V ▷ $O(1)$
5: Tomar la arista de menor peso A_0 y agregar los vértices a los que es incidente a V ▷ $O(1)$
6: Agregar A_0 a G' ▷ $O(1)$
7: **while** ($\#V <$ vértices(G)) **do** ▷ $O(n)$
8: Recorro L buscando la arista A_k de menos peso con un extremo en V y otro fuera de V ▷ $O(n^3)$
9: Agregar el vértice de A_k que no estaba en V a V ▷ $O(1)$
10: Agregar A_k a G' ▷ $O(1)$
11: **end while**
12: **return** circuitoNodos;
13: **end function**

3.1. Complejidad

Generamos la lista ordenada en $O(m^2) = O(n^4)$ usando selection sort. Luego, generamos el AGM usando PRIM que también tiene $O(n^4)$ porque el ciclo de la línea (7) se ejecuta n veces y la línea (8) implica, en el peor caso, recorrer todos los ejes ($O(n^2)$) y verificar para cada uno que sus vértices estén o no en V ($O(n)$). El resto del algoritmo PRIM es $O(1)$ salvo la línea (2) debido a que crear el grafo G' implica inicializar su matriz de adyacencia. Finalmente usamos DFS para recorrer el AGM y construir la lista de vértices en el orden en que son visitados. El pseudocódigo de DFS no se adjunta por no tener variantes ni consideraciones a realizar sobre el conocido y como es sabido tiene O(m) con lo que siendo en éste caso $m = n - 1$ (por ser un árbol) equivale a O(n).
Por lo tanto la heurística tiene orden total $O(n^4)$ + $O(n^4)$ + $O(n^1)$ + $O(n)$ = $O(n^4)$

3.2. Casos patológicos

Existen dos casos que son particularmente inciertos cuando se usa esta heurística. En primer lugar, al igual que en vecino más cercano y arista más corta, no hay validación alguna sobre el peso de la arista que completa el circuito (por fuera de lo que explicitado sobre grafos euclídeos), por lo que alcanza con que esta tenga un valor arbitrariamente alto para que la heurística de una solución tan mala como se desee. La visualización más sencilla de éste caso es cuando el AGM es un camino, de esa forma el problema se reduce a lo ya explicado para vecino más cercano y arista más corta con la salvedad de que en euclídeos esa última arista puede a lo sumo ser tan pesada como la suma de todo el camino obtenido (por desigualdad triangular).
Si bien de mucho menor interés, el segundo caso digno de mención es cuando el AGM forma un grafo estrella, es decir, uno donde la distancia entre dos nodos cualesquiera es exactamente 2 excepto por uno que oficia de centro de la estrella y está a distancia 1 de todos los demás. En ese caso, DFS recorre los vértices de forma absolutamente aleatoria (aunque, por supuesto, determinística dentro de una implementación dada) generando que con la única salvedad de tener garantía de que la arista más corta del grafo es parte del circuito obtenido pero no pudiendo garantizar nada respecto a ninguna de las otras, dado que no son parte del AGM. Cómo esto sigue estando acotado en el caso de los grafos euclídeos por 2D* (dos veces el resultado óptimo) y el peor caso no ser peor que el ya propuesto, dejamos constancia de su existencia como mera curiosidad sin considerar necesario ulterior análisis del caso.

4. Metaheurística Búsqueda Tabú

Los algoritmos metahuerísticos implementados en este trabajo toman un grafo y un ciclo hamiltoniano, y devuelven otro ciclo hamiltoniano que será de igual o menor peso.

En primer lugar, los algoritmos hallan “vecinos” del ciclo dado utilizando el método 2-opt, es decir, reemplazando un par de aristas aleatorias por otras, de modo que se forma otro ciclo hamiltoniano. El vecino tendrá posiblemente peso diferente ya que las aristas no son iguales. De esa forma se crean a lo sumo |V| vecinos, de los cuales se elige el de menor peso, el mejorVecino. Aunque ese mejorVecino tenga mayor peso que el ciclo inicial, se reescribe el ciclo por el mejorVecino, y se repite el proceso, es decir se le buscan vecinos al mejorVecino. Mientras tanto se guarda el menor camino de todos los encontrados para devolverlo al final del algoritmo.

Es posible que este proceso cicle entre los mismos caminos, sin desplazarse a otras familias de soluciones (ciclos hamiltonianos), por lo que se encontraría el mejor camino en un subconjunto más acotado que el deseado. Para evitar esto, se guarda en memoria una Lista Tabú, en la que se guardarán los vecinos que reemplacen al camino en cada iteración. Esto evita repetir las mismas soluciones. Sin embargo, es deseado que la Lista Tabú recuerde a lo sumo |T| elementos, de forma que si una solución se agregó a la Lista Tabú |T| soluciones atrás, se vuelve a admitir. Por lo tanto, la Lista Tabú tiene una capacidad limitada, |T|.

Una manera más eficiente en memoria de implementar la Lista Tabú es, en vez de memorizar y prohibir pasadas soluciones, memorizar ”movimientos” que conviertan una solución en otra. Si esos movimientos tienen reverso, al momento de aplicar un movimiento en una solución, reemplazándola por un vecino de esta, se puede memorizar el reverso de ese movimiento, previniendo que se vuelva a la solución anterior. En el caso de ciclos hamiltonianos y vecinos 2-opt, se considerará un movimiento el hecho de reemplazar dos aristas por otras, y se agregarán a la Lista Tabú las dos aristas removidas, prohibiendo las soluciones que impliquen volver a agregarlas.

Al prohibir movimientos en vez de soluciones es posible que se estén omitiendo soluciones nunca visitadas y que, aunque sean Tabú, también puedan ser óptimas o cercanas a una solución óptima. Para evitar esto, se define una función que decide si se admite una solución, aunque sea tabú. Esta función es llamada función de aspiración. La función elegida fue permitir una solución tabú en el caso de que sea la mejor solución calculada hasta el momento.

El algoritmo sigue buscando soluciones hasta una cantidad máxima de iteraciones (maxIter), siendo cada iteración un reemplazo de un ciclo por uno de sus vecinos.

Finalmente, se devuelve el mejor ciclo de los encontrados, el de peso mínimo.

Se implementaron ambas versiones de la Lista Tabú, en la sección 4.2 se describe el algoritmo con Lista Tabú basada en últimas soluciones exploradas, y en la sección 4.3 la metaheurística con una Lista Tabú basada en aristas.

4.1. Implementación algorítmica de las metaheurísticas

Ambas metaheurísticas comparten las siguientes entradas:

- Grafo (completo, con más de 3 nodos para que haya más de una solución, se pueden acceder a pesos de aristas en $O(1)$)
- Camino (hamiltoniano, se representa como vector de enteros, siendo cada elemento el índice de nodo del grafo)
- longTabu (longitud de la Lista Tabú, la cantidad máxima de elementos a recordar)
- porcentajeVecindario (porcentaje del vecindario de una solución a explorar)
- maxIter (cantidad máxima de iteraciones)

Y devuelven el peso del camino con menor peso encontrado. Con la única salvedad de que los elementos de la lista tabú serán caminos si se trata de MH1, mientras que si se trata de MH2 los elementos serán aristas.

Se cuenta con una variable camino que será reemplazada por un vecino (*maxIter*) veces, mientras que se guardará en mejorPeso el peso del camino con menor peso de entre todos los que se exploran.

---

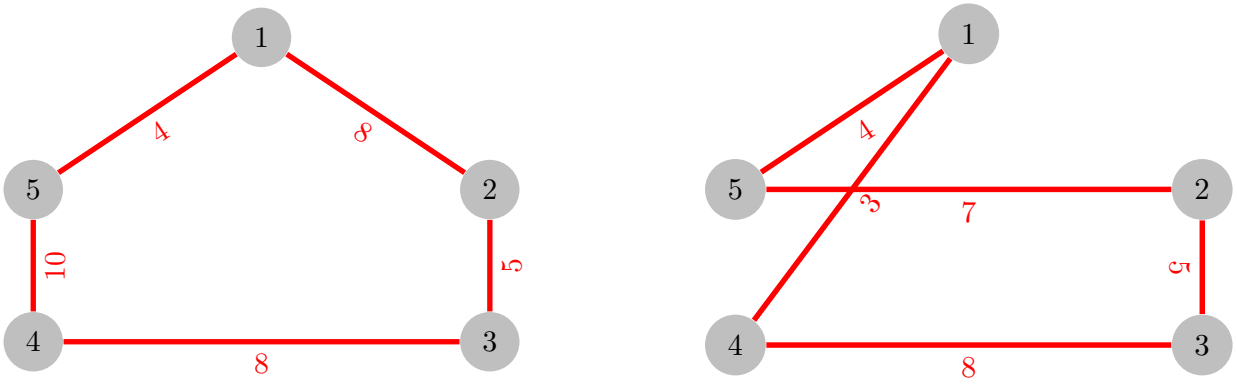
while maxIter $>$ 0 **do** ▷ O(maxIter * n^2 * porcentajeVecindario * T)
  vecino $\leftarrow$ hallarMejorVecino(camino) ▷ O(n^2 * porcentajeVecindario * T)
  if peso(vecino) $<$ mejorPeso **then**
    mejorPeso $\leftarrow$ peso(vecino) ▷ O(1)
  end if
  camino $\leftarrow$ vecino ▷ O(n)
  maxIter$--$
end while
return mejorPeso

La manera en la que se explora el vecindario de una solución es la siguiente: se miran a lo sumo *cantVecinos* vecinos, con cantVecinos = Cantidad de aristas * *porcentajeVecindario* /100.

Los vecinos 2-opt se construyen generando dos números aleatorios entre 1 y la longitud del camino (o la cantidad de nodos del grafo). Estos números generados representan índices en el camino, que es un vector de enteros, cada uno representando un nodo del grafo.

Llamando *n1* y *n2* a los números generados aleatoriamente, para generar un vecino 2-opt se remueven las aristas entre los nodos camino[*n1*-1] y camino[*n1*] y los nodos camino[*n2*] y camino[*n2*+*1*], y se agregan las aristas entre los nodos camino[*n1*-1] y camino[*n2*] y los nodos camino[*n1*] y camino[*n2*+1], formando otro camino hamiltoniano. Por supuesto, si *n1* o *n2* están en el borde del camino, al ser este un ciclo, los nodos consecutivos a estos pueden ser tomados del otro extremo del camino. Por ejemplo, si *n2* = (Cantidad de nodos), se elige "*n2*+1" = 0.

Para ilustrar este proceso se creó la Figura 3. Suponiendo que el camino original es el Vecino 1, y el número sobre el nodo es el orden que tiene en dicho camino, si se escogen *n1* y *n2* como 2 y 4 respectivamente, el camino resultante es el del Vecino 2. Las aristas removidas son las (1, 2) y (4, 5), y las agregadas son (1, 4) y (2, 5).



(a) Vecino 1.

(b) Vecino 2.

Figura 3: Vecinos 2-opt.

4.2. Metaheurística con memoria basada en soluciones exploradas

Por cada vecino obtenido mediante 2-opt, añadimos tantos candidatos como nos sea posible, para conformar un vecindario, y luego buscamos dentro del mismo al mejor candidato que tenga el menor peso posible:

---

while cantVecinos $>$ 0 **do** ▷ O(n*cantVecinos*T)

  vecino $\leftarrow$ crearVecino-2-Opt(camino) ▷ O(n)

  if vecino $\notin$ ListaTabu **then** ▷ O(1)

    cantVecinos$--$ ▷ O(1)

    vecindario.agregar(vecino) ▷ O(1)

  end if

end while

mejorCandidato $\leftarrow$ obtenerCandidatoDeMenorPeso(vecindario) ▷ O(n)

AgregarATabu(ListaTabu, mejorCandidato) ▷ O(1)

return mejorCandidato ▷ O(1)

De esta manera, para cada iteración de la metaheurística, vamos a determinar el mejor vecino dentro del vecindario, actualizar la lista tabú y quedarnos con la mejor solución explorada hasta el momento. De ser la solución explorada de menor peso que la mejor solución global encontrada hasta el momento, actualizamos esta última quedándonos con la mejor. La lista tabú se encarga de no seleccionar como candidato a algún vecino que ya haya sido explorado, con lo que expandimos las posibles exploraciones a distintos vecindarios posibles.

4.2.1. Complejidad

La creación de un vecino utilizando 2-opt, cuesta complejidad O(n), siendo n la cantidad de nodos que tiene el camino original al cual se le está creando el vecindario. Luego, vamos a crear tantos vecinos como sea necesario según el porcentajeRandomización ingresado como parámetro, es decir que la complejidad nos queda: O(n * porcentajeVecindario * cantAristas) = O(n^3 * porcentajeVecindario). Luego, para cada vecino posible además tenemos que recorrer toda la memoria tabú, para saber si el vecino ya fue explorado o no, con lo cual sumada la complejidad de esta operación nos queda: O(n^3 * porcentajeVecindario * longTabu). Por ultimo, la cantidad de iteraciones que exploramos diferentes vecindarios es (maxIter) veces, con lo cual el algoritmo tiene una complejidad en peor caso de O(maxIter * longTabu * porcentajeVecindario * n^3).

4.3. Metaheurística con memoria basada en aristas

Cada vecino, al ser obtenido mediante 2-opt, tiene dos aristas removidas y dos aristas agregadas en relación al camino original, por lo que al tener un vecino se lo analiza de la siguiente manera:

```
while cantVecinos > 0 do                                              ▷ O(cantVecinos * T)
    vecino ← crearVecino()                                            ▷ O(1)
    if aristasAgregadas ∉ ListaTabu OR peso(vecino) < mejorPeso then  ▷ O(T)
        cantVecinos−−
        if peso(vecino) < peso(mejorVecino) then
            mejorVecino ← vecino                                      ▷ O(1)
        end if
    end if
end while
AgregarATabu(ListaTabu, aristasRemovidas)                             ▷ O(1)
return mejorVecino
```

La Lista Tabú revisa que ninguna de las aristas agregadas en el vecino sean tabú, y si lo son, la función de aspiración admite el vecino si es el de menor peso encontrado desde que comenzó el algoritmo.

Cuando se terminan de explorar la cantidad de vecinos calculada antes, se actualiza la Lista Tabú y se devuelve el vecino no tabú de menor peso de la parte del vecindario explorado.

Para prevenir que la función no termine en el caso en el que una Lista Tabú pueda memorizar tantas aristas que sea capaz de prohibir todo un vecindario, si se crean 2*cantVecinos y todos son tabú, se detiene la busqueda de vecinos y se devuelve el vecino creado de menor peso, aunque sea tabú.

4.3.1. Complejidad

Para la construcción de un vecino se toman dos valores aleatorios, se guardan las aristas removidas y agregadas, y se calcula el peso del vecino en O(1) ya que se puede hallar haciendo peso(camino) + peso(AristasAgregadas) - peso(AristasRemovidas), siendo el peso del camino ya conocido y el peso de cualquier arista obtenida en O(1).

Luego, se hallan a lo sumo (2*cantVecinos) vecinos, esto es, (2**porcentajeVecindario***cantAristas) vecinos, y se revisa si cada vecino es Tabú, recorriendo la memoria linealmente. Esto tarda O(*longTabu*), y como el grafo es completo, la búsqueda en un vecindario tiene complejidad O(*longTabu***porcentajeVecindario***n^2), siendo n la cantidad de nodos.

Finalmente, la variable *camino* se reescribe (*maxIter*) veces, es decir, se recorre un vecindario (*maxIter*) veces. Además de obtener el mejorVecino la función reeplaza por copia la variable *camino* (O(n)), por lo que el algoritmo tiene una complejidad en peor caso de O(*maxIter * longTabu * porcentajeVecindario * n^2*).

5. Experimentación

En este trabajo se corrieron experimentos para cada uno de los cinco algoritmos implementados, comparando tiempos de ejecución y calidad de las soluciones obtenidas (comparando estas con una solución óptima), en función de la entrada, que en el caso de las heurísticas es el grafo, y en el caso de las metaheurísticas es tanto el grafo como los parámetros relacionados a la búsqueda tabú.

Además, se realizó un experimento comparando todos los algoritmos, viendo cual de ellos es más eficiente teniendo en cuenta tanto tiempo de ejecución como optimalidad de la solución.

Toda la experimentación se hizo en una computadora con CPU Intel Core I5 con 8GB de memoria RAM. Cada ejecución del algoritmo se realizó 5 veces, y el valor tomado en el análisis es la mediana de esos 5 valores, para evitar outliers.

5.1. Instancias

Para las siguientes experimentaciones utilizamos distintos conjuntos de instancias específicas para los objetivos de cada experimento. Durante el análisis de complejidad temporal utilizamos grafos completos pesados de tamaño variable de hasta 95 vértices. Dado que solo nos interesa el tiempo de cómputo y no el resultado, podemos usar aristas de pesos semi-aleatorios sin ninguna consecuencia sobre el rendimiento. El peso de cada arista (i, j) de este conjunto esta dado por la fórmula $((i + 1)(j + 1))mod100)$. Para evaluar la calidad de las heurísticas a fines prácticos, seleccionamos el conjunto de instancias conformado por mapas de ciudades reales, donde cada vértice representa una parada o ubicación. Para ello seleccionamos un subconjunto de las disponibles en TSPLIB[2] que son euclidianas, fáciles de parsear e incluyen su solución óptima al problema de TSP expresada en el peso total de los vértices del circuito hamiltoniano mínimo. Estas mismas se utilizaron a su vez para evaluar las metaheurísticas y sus mejores parámetros para este caso de estudio. A continuación las instancias que empleamos son:

- 'eli101': problema de 101 vértices. Solución óptima: no se conoce.
- 'berlin52': problema de 52 ubicaciones (52 vertices) en Berlín. Solución óptima: 7542.
- 'a280': problema de 280 ubicaciones en Berlín. Solución óptima: 2579.
- 'eli51': problema de Christofides/Eilon de 51 ubicaciones en una ciudad. Solución óptima: 426.
- 'bier127': problema de 127 ubicaciones. Solución óptima: 118282.
- 'kroA100': problema de Krolak/Felts/Nelson de 100 ubicaciones en una ciudad. Solución óptima: 21282.
- 'kroC100': problema de 100 vértices. Solución óptima: 20749.
- 'pr107': problema de 107 vértices. Solución óptima: 44303.
- 'ch130': problema de 130 vértices. Solución óptima: 6110.

5.2. Experimento 1: Complejidad de los algoritmos heurísticos

5.2.1. Complejidad del algoritmo heurístico vecino más cercano

Para poner a prueba nuestra hipótesis de que la complejidad asumida por la heurística del vecino más cercano es efectivamente $O(n^2)$, tomamos un dataset de grafos con aristas definidas según lo mencionado en la sección 5.1. No hicimos uso de los casos patológicos del punto 2.3, pues están relacionados a la calidad de la solución, no a la complejidad, sin embargo es interesante destacar que para el caso de grafos completos como cada vértice tiene $n1$ aristas y estas siempre se deben recorrer para asegurarnos de encontrar el mínimo, se puede establece una cota inferior $\Omega(n^2)$.

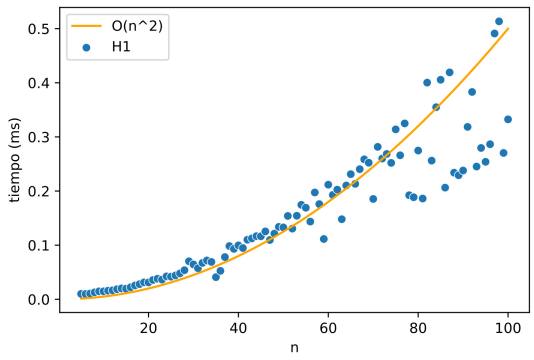
Corrimos instancias del dataset con tamaños del 5 al 100, un total de 5 veces cada instancia y tomamos la media, de forma que tengan mejor precisión las mediciones.

En la Figura 4a contrastamos el tiempo transcurrido de correr las instancias contra una parábola de la forma $O(n^2)$, y se presentaron unos pocos outliers mayores a lo que esperábamos, una hipótesis sobre estos es que se deban a interrupciones del sistema operativo u otros procesas en la maquina donde se realizaron los experimentos, sin embargo corroboramos que la complejidad es la correcta en la segunda gráfica, donde se visualiza la correlación de Pearson entre lo esperado y lo obtenido con un valor de aproximadamente 0,9103. En efecto, podemos ver que que se confirma la hipótesis sobre la complejidad, es decir que los resultados se asemejan a la curva esperada y en la Figura 4b se aprecia una correlacion directa entre las variables mencionadas

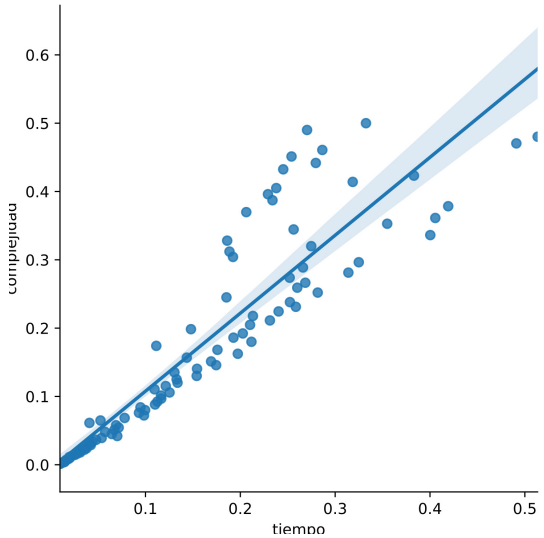
5.2.2. Complejidad del algoritmo heurístico arista más corta

En la Sección 2.2.1 explicamos el uso de la heurística de la arista más corta (HAMC) en la cual concluimos que su complejidad era de $O(n^4)$, mediante el mismo conjunto de instancias que usamos en la sección anterior llamado completos-random, también procedimos a correr las instancias de la misma

[2] http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/index.html



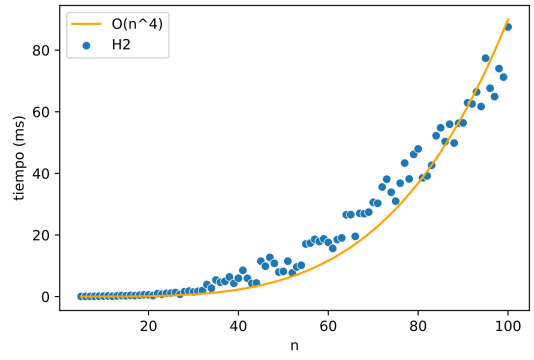
(a) Tiempo de ejecución H1 intancias random.



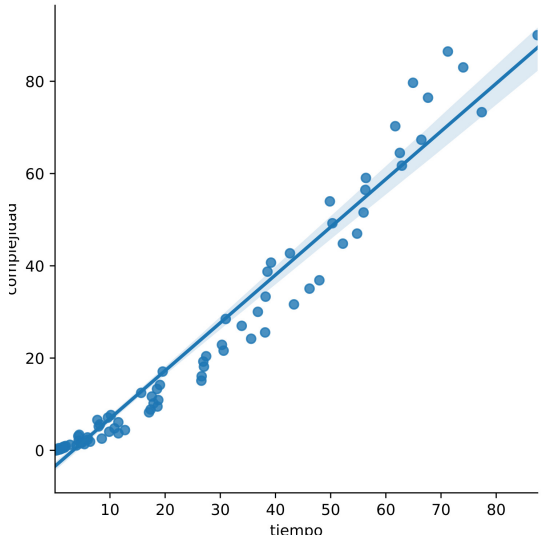
(b) Correlación entre el tiempo de ejecución y complejidad esperada.

Figura 4: Análisis de complejidad de Heurística vecino más cercano.

forma, es decir 5 veces por instancia con grafos completos de tamaño 5 a 100. En la Figura 5a podemos ver como los tiempos de ejecución de la heurística se ajusta a medida que crece n con la complejidad esperada. Igual que en el caso anterior, la cantidad de outliers no es significativa y la correlación de los datos esperados y obtenidos se puede terminar de verificar con la Figura 5b donde se obtuvo un índice de Pearson de aproximadamente 0.9797. Vale la pena destacar que tiene una complejidad mayor a las otras heurísticas, siendo que el $O(n^4)$ no solo se obtiene por la complejidad del ordenamiento inicial sino que proviene también de la complejidad con la que se encuentran las aristas a agregar, por ende no es de las heurísticas más eficientes en tiempo esto mismo se muestra más adelante en otra experimentación.



(a) Tiempo de ejecución H2 intancias random.

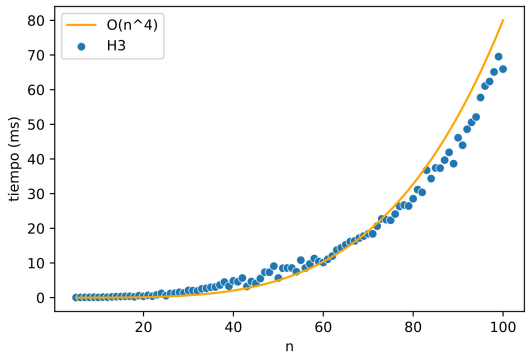


(b) Correlación entre el tiempo de ejecución y complejidad esperada.

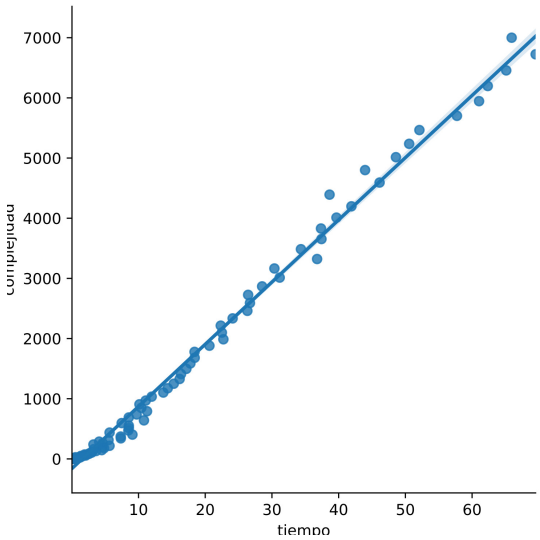
Figura 5: Análisis de complejidad de Heurística arista mas corta.

5.2.3. Complejidad del algoritmo heurístico árbol generador mínimo

Hasta ahora vimos los resultados de dos heurísticas basadas en conceptos mas sencillos de implementar, en este sección mostraremos los resultados de la heurística del árbol generador mínimo (o HAGM) la cual como fue detallado en la sección 3.1 esta basada en el algoritmo de Prim, y además se mostró que la complejidad del algoritmo es $O(n^4)$. Se utiliza el mismo conjunto de instancias que en los experimentos anteriores, con la misma cantidad de ejecuciones por instancia e igual n mínimo y máximo.



(a) Tiempo de ejecución H3 intancias random.



(b) Correlación entre el tiempo de ejecución y complejidad esperada.

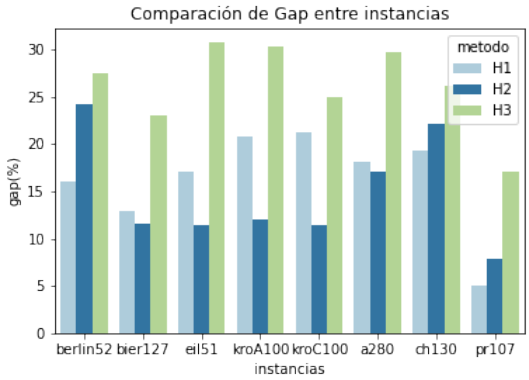
Figura 6: Análisis de complejidad de la Heurística árbol generador mínimo.

En la Figura 6a podemos ver como los tiempos de ejecución de la heurística se ajusta a medida que crece n con la complejidad esperada. Igual que en el caso anterior, la cantidad de outliers no es significativa y la correlación de los datos esperados y obtenidos se puede terminar de verificar con la Figura 6b donde se obtuvo un índice de Pearson de aproximadamente 0.9966.

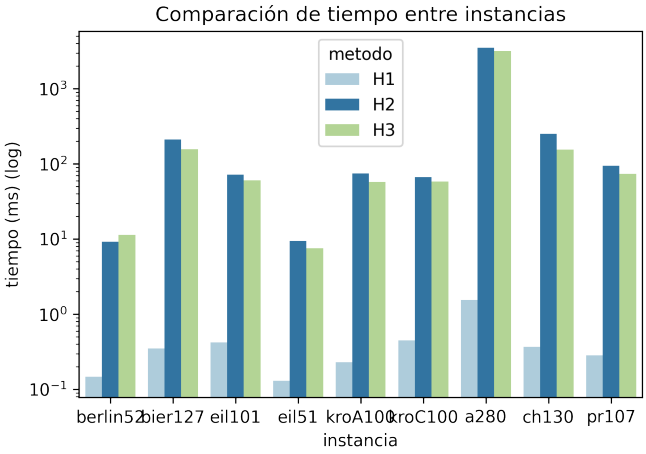
5.3. Experimento 2: comparación de los algoritmos heurísticos

Con el objetivo de comparar tiempos de ejecución y calidad del resultado (cercanía con el óptimo) vamos a evaluar un conjunto de 8 instancias para las cuales conocemos sus soluciones óptimas reales. Estas instancias son: 'berlin52', 'a280', 'bier127', 'eil51', 'kroA100', 'kroC100', 'pr107' y 'ch130' ya mencionadas en la sección 5.1. La hipótesis es que la heurística de AGM no debe dar una mejor en general, pero si garantizar una solución 1-aproximada para grafos euclidianos como vimos en la sección 3.1. Además la heurística de HAMC debería encontrar una mejor solución en general, pero con un costo de tiempo peor. En la Figura 7b se muestran los tiempos de ejecución en milisegundos (expresados de forma logarítmica), y a la izquierda en la Figura 7a se muestra el gap relativo entre la solución obtenida y la óptima. El método 'H1' se corresponde con la heurística de vecino más cercano, 'H2' con la heurística de arista más corta, y 'H3' con la heurística de árbol generador mínimo.

Podemos observar que la heurística de arista más corta y la heurística del árbol generador mínimo demoran mucho más tiempo que las de vecino más cercano. Esto refleja lo visto en la sección 5 donde ya sabíamos que la complejidad de la heurística es menor a las demás heurísticas, por lo que ahora puesto en contra de ellas se puede ver la diferencia importante que tiene con respecto al tiempo. La calidad de las soluciones obtenidas por 'HAGM' no fue la esperada, siendo superada por las heurísticas golosas incluso en instancias donde los vértices corresponden a coordenadas euclidianas donde suponíamos que la heurística AGM funcionaría mejor. La heurística más precisa, bajo este conjunto de instancias, es la de vecino más cercano ('H1'), que a su vez consume un tiempo de cómputo muy inferior a las otras dos. La heurística del vecino más cercano ('H1') fue la más rápida en el mismo conjunto de instancias y



(a) gap entre la solución óptima de las instancias TSPLIB y la solución encontrada por las heurísticas.



(b) Tiempo de ejecución H1, H2 y H3 intancias TSPLIB.

Figura 7: Análisis de comparación de las Heurísticas.

sus soluciones fueron más aceptables obteniendo gap relativo entre $\%20$ y $\%5$. Este método es el más balanceado entre tiempo y calidad. Sin embargo, esta heurística por ser golosa, no revisa las decisiones ya tomadas sobre el camino y en el peor caso puede encontrar circuitos mucho más largos que el óptimo para ciertas familias de instancias e inclusive existen instancias donde el algoritmo puede dar el peor camino posible. La heurística de árbol generador mínimo da soluciones aceptables con un gap relativo entre $\%20$ y $\%30$ y tiempos de cómputo muy altos, no es una mejora por sobre las otras dos en calidad de solución.

5.4. Experimento 3: Complejidad temporal de metaheurística basada en estructura(aristas).

Como fue explicado en la sección 5.1, para el análisis temporal se corrió el algoritmo en el dataset de grafos completos random con 5 a 100 nodos. Para analizar únicamente el impacto del tamaño del grafo, se dejaron constantes los parámetros *longTabu*, *porcentajeVecidario*, y *maxIter*. El resultado fue la Figura 8a, en donde se ve que el tiempo de ejecución del algoritmo tiene relación con la función n^2, siendo n la cantidad de nodo. La correlación es clara en la Figura 8b en donde se gráfico el tiempo de ejecución en el eje horizontal, y el tiempo esperado, una función del tipo cn^2 con $c \in \mathbb{R}$, en el eje vertical. El índice de correlación de Pearson entre los tiempos de ejecución, y la función cuadrática es de aproximadamente $0{,}9984$.

Habiendo comprobado que el tiempo de ejecución depende de la cantidad de nodos, y por lo tanto la de aristas, queda verificar si los otros parámetros del algoritmo influyen en la complejidad temporal, como se estima en la sección 4.3

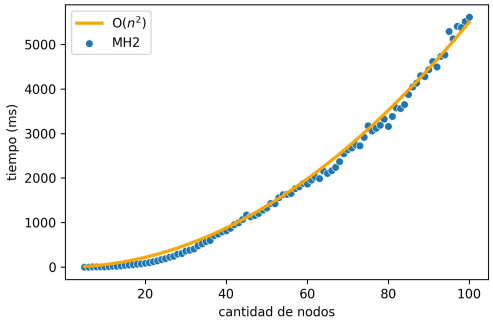
Se corrió el algoritmo en el grafo ”berlin52”, del dataset de TSPLIB, dejando primero el parámetro *porcentajeVecindario* constante en 25, y variando *longTabu* entre los valores 5, 10, 25, y 50. *maxIter* se varió de 1 a 100. Luego se hizo el inverso, se dejó constante *longTabu* en 26 (pensando en la mitad de cantidad de nodos), y se varió *porcentajeVecindario* entre los valores 5, 10, 20, y 50, con los mismos valores de *maxIter* ya mencionados.

En la Figura 9a se ve que para mayores tamaños de la Lista Tabú hay un ligero incremento en los tiempos de ejecución. Esta diferencia no es tan marcada como se hubiera esperado, y una razón posible es que el recorrido de la Lista Tabú se detiene en el momento que se detecta que la arista en cuestión está en la Lista. La búsqueda en la Lista es $O(T)$ solo si la arista no es tabú, lo cual es el peor caso, no todos los casos evaluados en la experimentación.

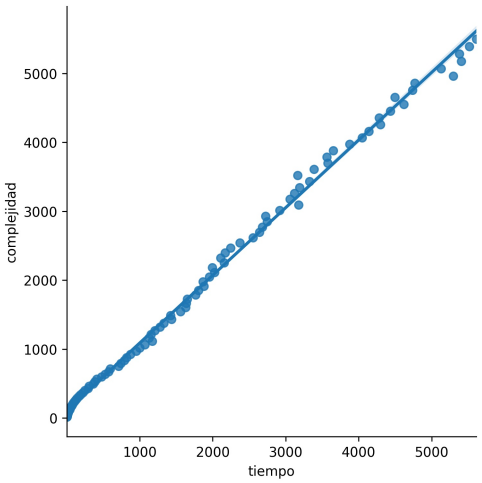
Por otro lado, al mirar una parte mayor del vecindario de una solución el algoritmo tarda más, con claras diferencias para cada valor del parámetro, en la Figura 9b se puede observar esto.

En ambas Figuras es claro que ante mayor cantidad de iteraciones, mayor es el tiempo de ejecución,

y que esta relación es lineal.

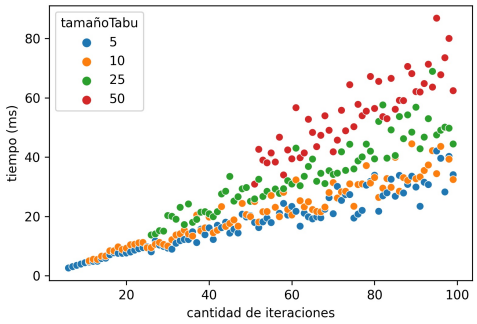


(a) Tiempo de ejecución metaheurística basada en estructuras.

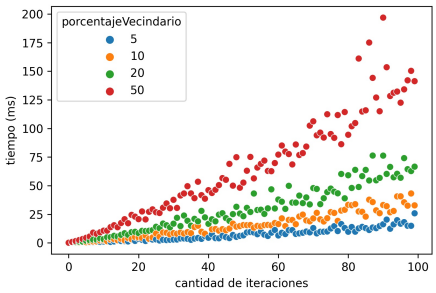


(b) Correlación entre el tiempo de ejecución y complejidad esperada ($O(n^2)$).

Figura 8: Análisis de complejidad temporal de la metaheurística basada en estructuras.



(a) Tiempo de ejecución variando longitud de la Lista Tabú.



(b) Tiempo de ejecución variando porcentaje del vecindario explorado.

Figura 9: Comparación de tiempos de ejecución para distintos parámetros de la metaheurística basada en estructura.

5.5. Experimento 4: Optimalidad de la solución de metaheurística basada en estructura.

La pregunta que motivó este experimento fue: ¿hay valores para los parámetros del algoritmo que provoquen soluciones de mayor calidad?

Para esto se corrió el algoritmo en grafos en los que se supiera la solución óptima y se calculó qué tanto mayor era el peso devuelto por el algoritmo, en comparación al óptimo.

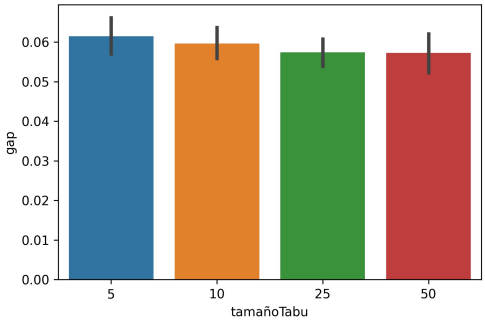
Para cada uno de los valores de *longTabu* y de *porcentajeVecindario* en el experimento 3, se evaluó el ”gap”, calculado como (solucion - óptimo) / óptimo.

El algoritmo se corrió en tres grafos del dataset de TSPLIB. En el caso del grafo ”berlin52”, es importante notar que el camino hamiltoniano con el que inicia el algoritmo es el resultado de la ejecución del algoritmo heurístico vecino más cercano. En el experimento 2 se mostró que para este grafo, el algoritmo logra una solución con un gap algo mayor que 0.15. Es esperable que toda solución de la metaheurística mejore a partir de ese valor.

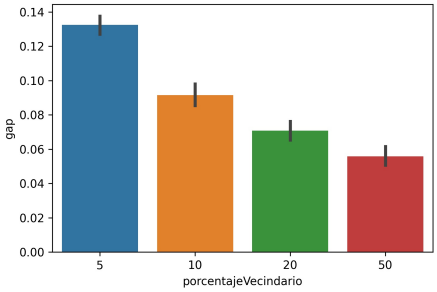
El resultado fue que aumentar el porcentaje del vecindario explorado supuso un claro aumento en la calidad de la solución. En la Figura 10b se ve que para un porcentaje de 5, la solución es un 13 % mayor que el óptimo, pero si se explora el 50 % del vecindario, la solución alcanza un promedio de solo 6 % mayor que el óptimo. Estos valores son el promedio de las ejecuciones del algoritmo, con *longTabu* en 26 y haciendo de, 1 a 100 iteraciones.

Por otro lado, una Lista Tabú más grande no supuso ningún cambio notorio en la cantidad de la solución, y si lo hubo, no fue detectable para la magnitud de las variaciones que se le hizo a *longTabu*, ya que en la Figura 10a las barras de error dicen que bien podría no haber ningún cambio en la calidad de la solución con la variación de *longTabu* hecha.

En la Figura 11 se ve una tabla con los valores promedio de gap, para distintos tamaños de la Lista Tabú, y los tres grafos usados en este experimento. Salvo en el caso de ”pr107” con tamaño tabú de 50, hay una baja en el gap a medida que se incrementa el tamaño tabú, si bien es muy pequeña.



(a) Calidad de la solución variando longitud de la Lista Tabú.



(b) Calidad de la solución variando porcentaje del vecindario explorado.

Figura 10: Evaluación de la calidad de la solución devuelta por la metaheurística basada en estructuras, para distintos parámetros. Las barras son el promedio del gap de todas las soluciones obtenidas, y las barras de error son el intervalo de confianza nivel 0.95

Tamaño Tabú \ Instancia	berlin52	ch130	pr107
5	0.0615	0.0756	0.00947
10	0.05962	0.07085	0.00865
25	0.0574	0.068349	0.008548
50	0.05724	0.06452	0.00904

Figura 11: Valores gap para los tres grafos y los distintos tamaños de la Lista Tabú

Para poder ver como mejora la solución a medida que aumentan las iteraciones se creó la Figura 12, que muestra el gap de la solución obtenida para los grafos ”berlin52”, ”ch130” y ”pr107” (todos de TSPLIB). Además se varía entre 4 valores diferentes de *porcentajeVecindario*.

Se puede ver que la solución llega a la mejor posible aproximadamente a las 40 iteraciones, con una mejora notoria en las primeras 25. Aunque se hagan más iteraciones, la solución rara vez alcanza un mejor valor. Sin embargo, es evidente que un *porcentajeVecindario* más elevado le permiten al algoritmo alcanzar una solución mejor que la que podría haber alcanzado de otra manera, aunque se tuvieran más iteraciones para intentarlo. Por eso es que un *porcentajeVecindario* de 50 es lo más beneficioso para la optimalidad de la solución.

El caso de *longTabu* no es tan notorio, para los cuatro valores evaluados la solución disminuye su gap muy ligeramente, como se expuso en la Figura 11, sin embargo para alcanzar la mejor solución, tomar una Lista Tabú de tamaño 50 es lo mejor entre lo que se experimentó.

Se concluye entonces que explorar un porcentaje mayor del vecindario se traduce en una clara mejora en la calidad de la solución, y eso sucede también con la longitud de la Lista Tabú .

Además, para llegar a la mayor solución posible es necesaria una cantidad de iteraciones mínimas, pero si se supera ese valor, la solución se mantiene constante en calidad, por lo que no ayuda a la optimalidad poner la mayor cantidad de iteraciones posible.

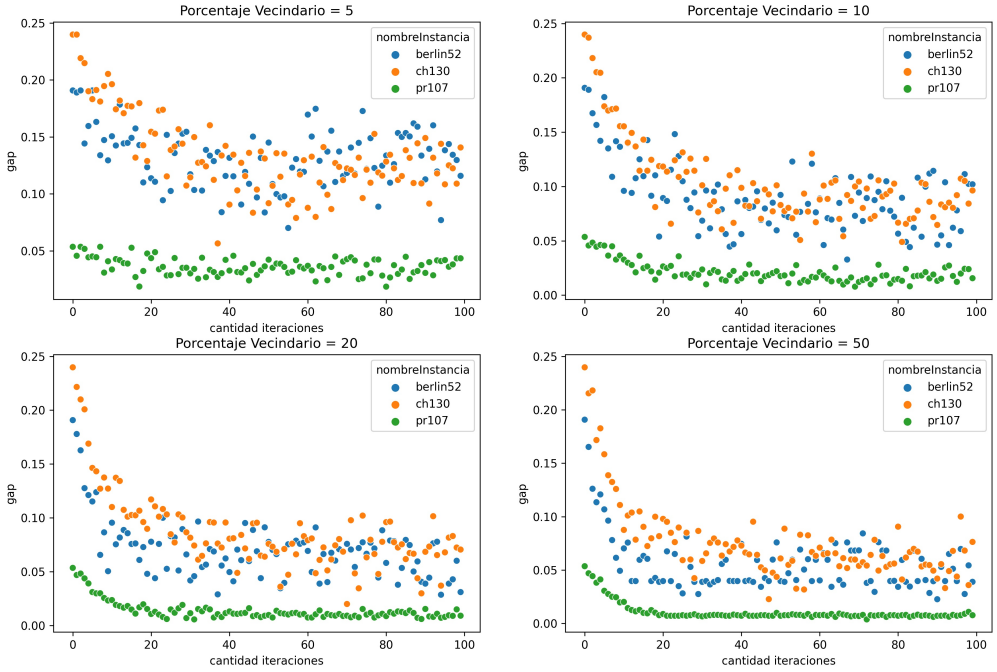


Figura 12: Para iteraciones de 1 a 100, gap de la solución obtenida, para distintos parámetros.

5.6. Experimento 5: Complejidad temporal de metaheurística basada en soluciones exploradas.

Como fue explicado en la sección 5.1, para el análisis temporal se corrió el algoritmo en el dataset de grafos completos random con 5 a 100 nodos. Para analizar únicamente el impacto del tamaño del grafo, se dejaron costantes los parámetros *longTabu*, *porcentajeVecidario*, y *maxIter*. El resultado fue la Figura 13a, en donde se ve que el tiempo de ejecución del algoritmo tiene relación con la función n^3, siendo n la cantidad de nodos. La correlación es clara en la Figura 13b en donde se graficó el tiempo de ejecución en el eje horizontal, y el tiempo esperado, una función del tipo $c * n^3$ con $c \in \mathbb{R}$, en el eje vertical. El índice de correlación de Pearson entre los tiempos de ejecución, y la función cúbica es de aproximadamente $0{,}9951$.

Habiendo comprobado que el tiempo de ejecución depende de la cantidad de nodos, y por lo tanto la de aristas, queda verificar si los otros parámetros del algoritmo influyen en la complejidad temporal, como se estima en la sección 4.2

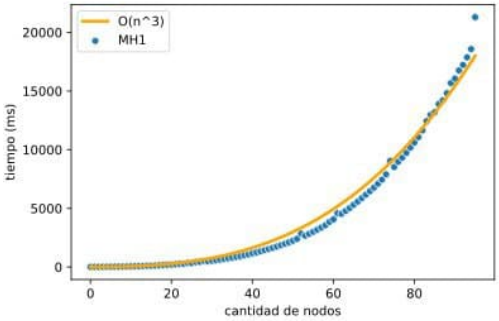
Se corrió el algoritmo en el grafo ”berlin52”, del dataset de TSPLIB, dejando primero el parámetro *porcentajeVecindario* constante en 25, y variando *longTabu* entre los valores 5, 10, 25, y 50. *maxIter* se varió de 1 a 100. Luego se hizo el inverso, se dejó constante *longTabu* en 26 (pensando en la mitad de cantidad de nodos), y se varió *porcentajeVecindario* entre los valores 5, 10, 20, y 50, con los mismos valores de *maxIter* ya mencionados.

En la Figura 14a se ve que para mayores tamaños de la Lista Tabú hay un ligero incremento en los tiempos de ejecución. Esta diferencia es muy pequeña, a pesar de que siempre se recorre la totalidad de la lista tabú para comparar si las soluciones candidatas ya fueron exploradas. Una posible justificación de este comportamiento es que la diferencia en tamaño es que el recorrido de la Lista Tabú se detiene

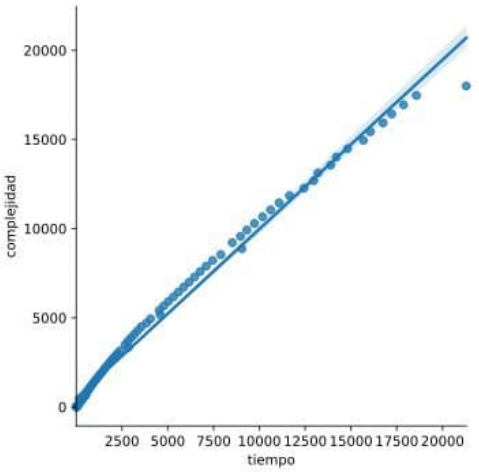
en el momento que se detecta que la solución candidata esta en la Lista. La búsqueda en la Lista es $O(T)$ solo si la solución candidata no es tabú, lo cual es el peor caso, no todos los casos evaluados en la experimentación.

Por otro lado, al mirar una parte mayor del vecindario de una solución el algoritmo tarda más, con claras diferencias para cada valor del parámetro, en la Figura 14b se puede observar esto.

En ambas Figuras es claro que ante mayor cantidad de iteraciones, mayor es el tiempo de ejecución, y que esta relación es lineal.

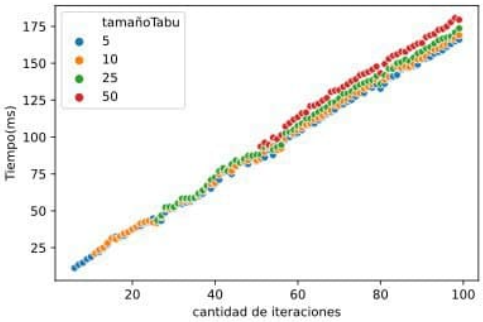


(a) Tiempo de ejecución metaheurística 1.

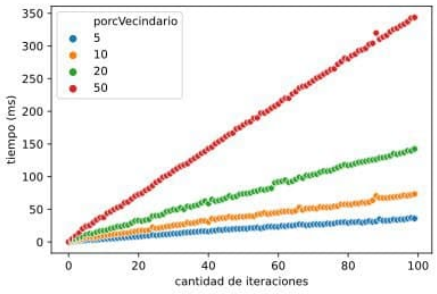


(b) Correlación entre el tiempo de ejecución y complejidad esperada ($O(n^3)$).

Figura 13: Análisis de complejidad temporal de la metaheurística basada en soluciones exploradas.



(a) Tiempo de ejecución variando longitud de la Lista Tabú.



(b) Tiempo de ejecución variando porcentaje del vecindario explorado.

Figura 14: Comparación de tiempos de ejecución para distintos parámetros de la metaheurística basada en soluciones exploradas.

5.7. Experimento 6: Optimalidad de la solución de metaheurística basada en soluciones exploradas.

Para cada uno de los valores de *longTabu* y de *porcentajeVecindario* en el experimento 5, se evaluó el ”gap”, calculado como (solución - óptimo) / óptimo.

El algoritmo se corrió en el grafo ”berlin52”, del dataset de TSPLIB, y es importante notar que el camino hamiltoniano con el que inicia el algoritmo es el resultado de la ejecución del algoritmo heurístico vecino más cercano. En el experimento 2 se mostró que para este grafo, el algoritmo logra una solución

con un gap algo mayor que 0.15. Es esperable que toda solución de la metaheurística mejore a partir de ese valor.

El resultado fue que aumentar el porcentaje del vecindario explorado supuso un claro aumento en la calidad de la solución. En la Figura 16b se ve que para un porcentaje de 5, la solución es un 11 % mayor que el óptimo, pero si se explora el 50 % del vecindario, la solución alcanza un promedio de solo 5 % mayor que el óptimo. Estos valores son el promedio de las ejecuciones del algoritmo, con *longTabu* en 26 y haciendo de 1 a 100 iteraciones.

Por otro lado, una Lista Tabú más grande supuso un cambio pequeño en la calidad de la solución, si bien no fue extremadamente notorio, a medida que fue aumentando *longTabu*, se puede apreciar en la Figura 16a que la calidad de la solución mejora un 0.005 %.

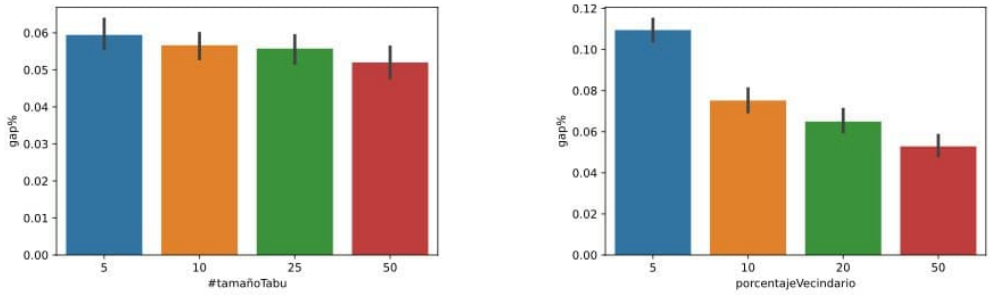
Para poder ver como mejor la solución a medida que aumentan las iteraciones se creo la figura 17a, que muestra el gap de la solución obtenida para los grafos ”berlin52”, ”ch130” y ”pr107” (todos de TSPLIB). Además se varía entre 4 valores diferentes de *porcentajeVecindario*.

Se puede ver que la variación del *porcentajeVecindario*, a medida que aumentan la cantidad de iteraciones, se llega cada vez a una solución mejor que la que podría haber alcanzado de otra manera, aunque se tuvieran más iteraciones para intentarlo.

Se concluye entonces que explorar un porcentaje mayor del vecindario se traduce en una clara mejora en la calidad de la solución.

Tamaño Tabú \ Instancia	berlin52	ch130	pr107
5	0.0594	0.0757	0.0091
10	0.05664	0.07170	0.00820
25	0.05574	0.06540	0.00736
50	0.05200	0.07255	0.00719

Figura 15: Valores gap para los tres grafos y los distintos tamaños de la Lista Tabú



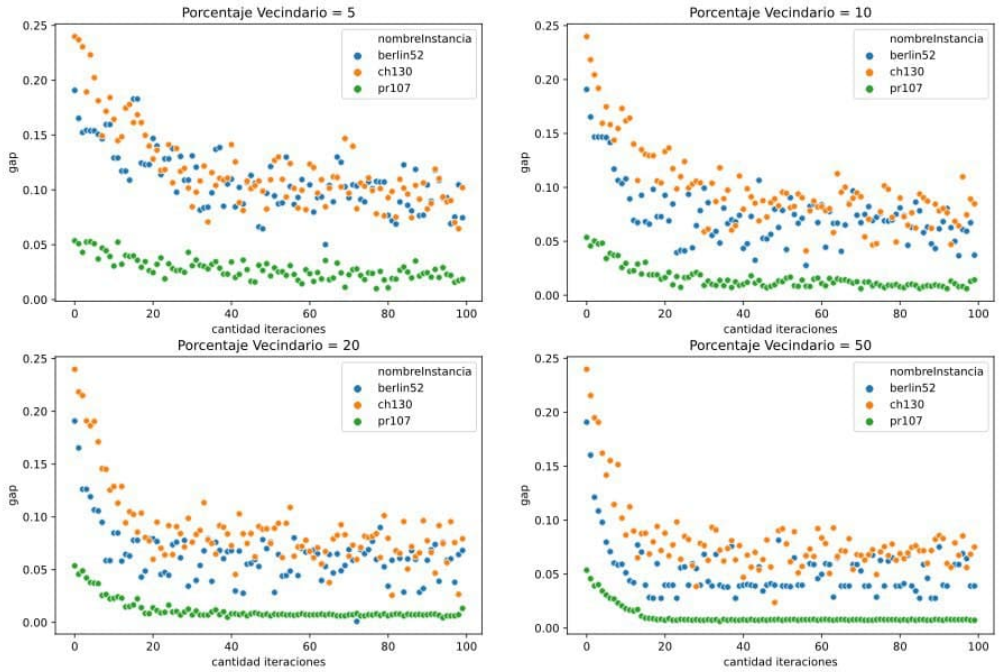
(a) Calidad de la solución variando longitud de la Lista Tabú.

(b) Calidad de la solución variando porcentaje del vecindario explorado.

Figura 16: Evaluación de la calidad de la solución devuelta por la metaheurística basada en soluciones exploradas, para distintos parámetros. Las barras son el promedio del gap de todas las soluciones obtenidas, y las barras de error son el intervalo de confianza nivel 0.95

5.8. Experimento 7: Performance de los métodos implementados comparando calidad de soluciones obtenidas y tiempos de ejecución en función del tamaño de entrada.

Mediante la experimentación realizada en las secciones 5.5 y 5.7, se utilizaron los datasets de la cátedra berlin52, ch130 y pr107 para hallar los mejores parámetros posibles de las metaheurísticas, de tal manera que se obtengan las soluciones mas cercanas a las óptimas, es decir las que minimicen el gap %.



(a) Calidad de la solución en función de iteraciones, para distintos valores del porcentaje del vecindario explorado.

Figura 17: Para iteraciones de 1 a 100, gap de la solución obtenida, para distintos parámetros.

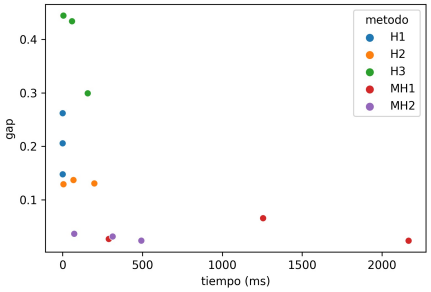
En este experimento, queremos utilizar nuevas instancias representativas de la vida real, en donde vamos a comparar las implementaciones heurísticas vs metaheurísticas y determinar cual de todas nos convendría usar, basándonos en los mejores valores de configuración obtenidos durante la experimentación con los datasets presentados anteriormente.

La hipótesis es que claramente vamos a obtener una mejora significativa en la calidad obtenida cuando utilicemos las metaheurísticas en comparación a las heurísticas, pero lo interesante es analizar el trade-off entre conseguir una solución razonablemente "buena" en un tiempo razonablemente chico, y conseguir una muy buena solución cercana a la óptima aunque perdiendo tiempo de ejecución.

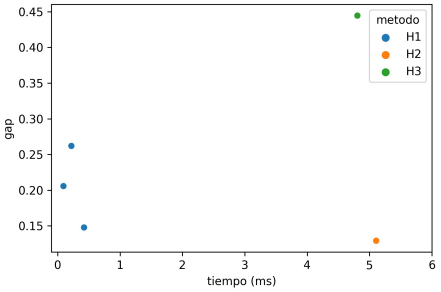
Lo esperado a partir del estudio mencionado, es que se hipotetiza que para obtener una solución razonablemente buena en un tiempo de ejecución corto, lo mejor es usar la heurística de vecino mas cercano, ya que se trata de la que tiene menor complejidad teórica(O(n^2)), y presentaba soluciones de calidad razonable. Luego, se espera que las metaheurísticas encuentren soluciones mucho mas cercanas al óptimo pero a cambio de una pérdida en el tiempo de cómputo, es decir que tardarán mas en encontrar esta mejor solución. Cabe mencionar, que de entre las 2 metaheurísticas se espera que la mejor sea MH2, ya que tiene menor complejidad teórica(O(n^2)) que MH1(O(n^3)) y durante la experimentación mientras se ajustaban los parámetros, ambas eran similarmente efectivas.

En la figura 18a se puede apreciar que efectivamente, como se sospechaba, para las 3 instancias se consigue una solución cercana a la óptima utilizando la heurística de vecino mas cercano en un tiempo corto. Se puede observar, que si bien para uno de los datasets podría parecer que H2 le gana en tiempo a H1, esto ocurre debido a la escala, ya que realmente el tiempo insumido por H1 se encuentra a una distancia de al menos 4ms de cualquier tiempo insumido por H2, como puede apreciarse haciendo zoom en la figura 18b. También se observa en el gráfico, que la heurística 3 dio como resultado que tarda mas que H1, y que la calidad de su solución no es lo suficientemente buena. Para el caso de las metaheurísticas se aprecia que como se esperaba, la calidad aumenta notablemente con respecto a las heurísticas, y además el tiempo insumido por MH2 es menor que MH1. De esta manera concluimos, que si queremos una solución sin que nos importe mucho la calidad, pero la queremos rápido conviene usar H1, mientras

que si nos importa mucho la calidad de la solución y no nos importa el consumo de tiempo requerido, conviene utilizar MH2.



(a) Calidad de las soluciones en función del tiempo, para los datasets bier127, kroA100 y eil51 en los mejores parámetros encontradas de las heurísticas y metaheurísticas.



(b) Zoom de las soluciones en función del tiempo, mostrando la diferencia entre el peor tiempo insumido por H1 vs el mejor tiempo insumido por H2.

Figura 18: Tiempo vs gap variando heurísticas y metaheurísticas para datasets bier127, kroA100 y eil51.

6. Conclusiones

En la experimentación de las heurísticas más simples, la de vecino más cercano (HVMC) obtuvo la mejor relación tiempo-calidad de la solución, además de ser un algoritmo de fácil desarrollo y baja complejidad. Sin embargo en el caso genérico no existen garantías de calidad para este algoritmo por lo cual se recomiendan otras heurísticas en casos donde importe más garantizar una mejor solución que el tiempo de cómputo.
En el caso de la arista más cercana (HAMC) ocurre una peor relación tiempo-calidad, presentando una mejor solución en la mayoría de los experimentos pero con un costo de tiempo mucho mayor, por lo cual en casos donde el tiempo es muy limitado y la calidad de la solución no es tan relevante se recomienda la primera heurística.
Por último está la HAGM que no tuvo el desempeño más rápido ni la mejor solución, como era de esperarse, pero en particular permite definir una cota del gap (cuando los grafos de distancias son euclidianos) al ser 1-aproximado. Con lo cual seria el algoritmo mas recomendable cuando es garantizar cotas para el gap sobre grafos euclídeos.
Se iteró de forma experimental sobre los diferentes parámetros de las meta-heurísticas con memoria tanto por soluciones como por aristas y se encontraron valores que logran minimizar el gap de la solución de manera aceptable y suficiente para la expectativa del estudio en las instancias esperadas. Para las pruebas finales, una vez elegidos los parámetros de las meta-heurísticas, se obtuvo buenos resultados para otras instancias de prueba de similar tamaño. La selección de los parámetros que optimizan las meta-heurísticas para instancias de la vida real como son las de TSPLIB parecieran ser posibles de fijar en forma general. El presente trabajo deja propuestos parámetros para las instancias estudiadas de entre 50 y 100 vértices. Finalmente aunque ambas meta-heurísticas proporcionan buenos resultados, en casos donde la cantidad de memoria disponible es una limitante se recomienda el uso de meta-heurística de memoria por aristas ya que su consumo en espacio es menor que el utilizando en memoria por soluciones completas.