DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

TP 2: TSP

Algoritmos y estructuras de datos III

Integrante	LU	Correo electrónico
Galland, Octavio Adolfo	597/18	octavio.galland@gmail.com
Riera, Leandro	658/18	leandro.riera.m@gmail.com
Miceli, Juan Pablo	424/19	micelijuanpablo@gmail.com
Zolezzi, María Victoria	222/19	zolezzivic@gmail.com

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

Índice

1. Introducción

En este trabajo nos dedicaremos a estudiar el problema del viajante (*TSP* por sus siglas en inglés). Propondremos también distintos algoritmos basados en heurísticas y meta-heurísticas para atacar dicho problema, evaluando de manera experimental tanto la calidad de las soluciones obtenidas como el tiempo insumido por una implementación de dichos algoritmos para arribar a una solución.

1.1. Descripción del problema

El problema TSP surge de plantear la siguiente pregunta:

> Dado un mapa de ciudades y la distancia entre cada par de ciudades ¿Cúal es el recorrido más corto posible que pasa por cada ciudad exactamente una vez y termina volviendo a la ciudad de origen?

Podemos observar que este problema puede ser modelado utilizando teoría de grafos. El problema es análogo a encontrar el camino hamiltoniano de menor costo en un grafo pesado completo. Podemos construir un grafo G de la siguiente manera: cada ciudad es representada con un vértice, y entre cada par de vértices colocamos una arista de peso igual a la distancia entre las ciudades que conecta. Si encontramos un camino hamiltoniano de menor costo C en G, podemos recorrer las ciudades en el orden el que figuran en C y estar seguros de que ese recorrido es el que minimiza la distancia (dado que el costo del camino es la suma de los pesos de las aristas, que es igual a su vez a la distancia recorrida).

Una vez modelado el problema utilizando teoría de grafos notamos que se trata de un problema de optimización combinatoria, ya que el mismo consiste en encontrar el ciclo de menor peso posible en un grafo. Dado que el recorrido tiene que pasar por todos los vértices, y que el grafo es completo, solucionar el problema consiste únicamente en decidir el orden en el cual se visitarán los vértices. Esta ultima observación resulta útil para definir la cantidad de soluciones factibles, dado que hay tantas soluciones factibles como posibles ordenamientos de un conjunto de n vértices. Esto implica que hay $n!$ soluciones factibles a explorar si quisiéramos implementar un algoritmo de búsqueda exhaustiva para solucionar el problema. Claramente implementar un algoritmo de búsqueda exhaustiva tiene un costo computacional prohibitivo inclusive para instancias pequeñas del problema, por lo cual debemos recurrir a otros tipos de algoritmos.

Es importante destacar que TSP es un problema NP-dificil, por lo que no se conocen algoritmos polinomiales para resolverlo. Debido a la falta de algoritmos de complejidad polinomial para resolver el problema, nos vamos a centrar en el tratamiento del mismo mediante el uso de *heurísticas* y *meta-heurísticas*. Las mismas no nos proveen de una solución óptima, sino de una solución "buena". Cuando hablamos de que una solución es buena nos podemos referir a una de dos cosas:

- El algoritmo construye una solución intentando minimizar el costo de la misma, por lo cual asumimos que la calidad de dicha solución es satisfactoria basándonos únicamente en el funcionamiento algoritmo.
- Podemos demostrar que el error de la solución dada por el algoritmo está acotado, en tal caso tenemos una garantía sobre la calidad de la solución y decimos que el algoritmo es aproximado.

El hecho de tener que lidiar con este nivel de incertidumbre nos plantea una situación de tensión entre la calidad de las soluciones obtenidas y el tiempo de ejecución, dado que obtener mejores soluciones implica recorrer más exhaustivamente el espacio de soluciones factibles, lo cual implica un mayor tiempo de ejecución (podemos pensar al algoritmo de fuerza bruta como un extremo del espectro calidad de la solución - eficiencia).

1.2. Aplicaciones de TSP

Además de encontrar su aplicación más natural en problemas tales como el ruteo de vehículos, podemos modelar varios problemas importantes como instancias particulares de TSP. Uno de estos problemas es la construcción de PCBs (*printed circuit boards*), dado que a la hora de construir un PCB se necesita perforar la placa en distintos puntos. Mover el cabezal de la máquina perforadora de un punto a otro insume un determinado tiempo, por lo que elegir eficientemente el orden en el que se realizarán las perforaciones permite ahorrar tiempo y energía al fabricante (si bien este ahorro puede parecer marginal a primera vista, es importante tener en cuenta que dado el volumen de producción de algunas empresas fabricantes de PCBs estos ahorros pueden ser muy significativos). De la misma manera, se puede interpretar el problema de cablear los distintos componentes de una placa en una computadora de forma tal de minimizar la latencia y el ruido producido por la longitud de los cables como una instancia TSP. Otra aplicación, quizás menos obvia, es el *scheduling* de trabajos en distintas máquinas para reducir el tiempo total requerido para despachar todos los trabajos.

2. Heurística constructiva golosa: el vecino más cercano

La heurística del vecino más cercano construye un circuito hamiltoniano de la siguiente forma:

1. Incluír en la solución el nodo inicial v_1.
2. En la iteración k contamos con la solución parcial $v_1, ..., v_k$, en este punto elegimos como v_{k+1} al nodo más cercano a v_k que aún no fue visitado. Realizamos esta operación hasta haber visitado todos los vértices.
3. Conectamos v_n con v_1.

Algorithm 1 Heurística golosa: vértice más cercano.

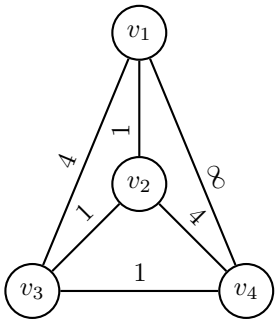
```
1:  function masCercano(G, v_0)
2:      V = [v_0]                                   ▷ Conjunto solución O(1)
3:      cantidadNodos = 0                           ▷ O(1)
4:      pesoDelCamino = 0                           ▷ O(1)
5:      v = v_0
6:      while | V | < n do                          ▷ O(n)
7:          w = argmin{ c_vw, w ∈ E \ V }           ▷ O(n)
8:          V.agregarAtras(w)                       ▷ O(1)
9:          cantidadNodos + +
10:         pesoDelCamino+ = c_vw
11:         v = w                                   ▷ O(1)
12:     endwhile                                    ▷ Complejidad del ciclo: O(n^2)
13:     return V, cantidadNodos, pesoDelCamino      ▷ O(1)
    Complejidad: O(n^2)
```

Como podemos ver en el algoritmo 1, tenemos un ciclo de n iteraciones, donde cada una de ellas consiste en la búsqueda del nodo perteneciente a la vecindad del nodo v cuya distancia a este último sea mínima. Teniendo en cuenta que G es un grafo completo, este conjunto tiene cardinal $n - 1$, con lo cual cada iteración tiene costo lineal en la cantidad de vértices. Luego, $masCercano \in \mathcal{O}(n^2)$.

Este algoritmo nos devuelve siempre un circuito hamiltoniano, sin embargo al ser una heurística, no tenemos la certeza de que el circuito obtenido sea el de menor peso. Por ejemplo, sabiendo que la elección de la arista que cierra el circuito está fijada por la elección de los primeros n-1 ejes, existen familias de grafos para las cuales este algoritmo puede ser tan malo como queramos. Una manera

de construir una de estas de familias es setear el costo de las aristas de un circuito hamiltoniano en x, exceptuando la que une el último nodo visitado con el primero, la cual tendrá costo infinito. El resto de los ejes del grafo pueden tener peso y, con $y > x$. De esta forma, el algoritmo irá agregando los vértices correspondientes a las aristas de costo x, y por último tendrá que cerrar el circuito. Para hacerlo, no le quedará otra opción que incluír el eje de peso infinito, devolviendo un circuito hamiltoniano de este mismo peso.

Veamos un ejemplo de un grafo perteneciente a la familia desarrollada en el párrafo anterior:



Veamos qué resulta de aplicar nuestra heurística este grafo. Antes de empezar, podemos determinar fácilmente que el circuito hamiltoniano de menor costo es $[v_1, v_3, v_4, v_2]$, cuyo costo es 10.

El nodo incial es v_1, por lo que buscaremos entre sus vecinos al más cercano, que en este caso resulta ser v_2. Ahora, debemos continuar el camino desde v_2 repitiendo el procedimiento, lo que hará que agreguemos al camino actual a v_3, y a v_4, cuyo peso es hasta el momento 3. Por último, debemos cerrar el camino, y por lo tanto, debemos conectar a v_4 con v_1, nodos que están conectados mediante una arista de costo infinito. De esta forma, nuestro algoritmo devolverá el circuito $[v_1, v_2, v_3, v_4]$, cuyo costo resulta ser infinito.

3. Heurística constructiva golosa: menor costo

En este caso, la construcción del circuito hamiltoniano se realiza de la siguiente forma:
Sea $H = (V, E_H)$ el grafo que describe el circuito hamiltoniano obtenido por el algoritmo.

1. Ordenamos las aristas por costo de mayor a menor.
2. En cada iteración tomamos la arista (u, v) de menor costo que todavía no fue seleccionada que cumpla que $d_H(u) < 2$, $d_H(v) < 2$ y que (u, v) no forme un ciclo al ser incorporada al conjunto de aristas ya elegidas.
3. Agregamos la única arista (u, v) tal que $d_H(u) = 1$ y $d_H(v) = 1$, es decir, aquella que cierra el circuito.

Algorithm 3

```
1: function menorCosto(G, v_0)
2:     ordenar(E)                                                    ▷ O(n^2 log n^2)
3:     E_H = ∅                                                       ▷ Conjunto solución O(1)
4:     V_H = ∅                                                       ▷ Conjunto solución O(1)
5:     cantidadNodos = 0                                             ▷ O(1)
6:     pesoDelCamino = 0                                             ▷ O(1)
7:     while | E_H | < n − 1 do
8:         (u, v) = argmin{ c_uv, (u, v) ∈ E \ E_H ∧ d_H(u) < 2 ∧ d_H(v) < 2 ∧
9:                          (u, v) no forma ciclos en E_H ∪ (u, v)}
10:        if u ∉ V_H then                                           ▷ O(n)
11:            V_H ∪ {u}                                             ▷ O(1)
12:            cantidadNodos + +                                     ▷ O(1)
13:        if v ∉ V_H then                                           ▷ O(n)
14:            V_H ∪ {v}                                             ▷ O(1)
15:            cantidadNodos + +                                     ▷ O(1)
16:        E_H ∪ {(u, v)}                                            ▷ O(1)
17:        pesoDelCamino+ = c_uv                                     ▷ O(1)
18:    endwhile                                  ▷ Complejidad del ciclo: O(n^2)
19:    E_H ∪ {(u, v)} tq (u, v) ∈ E \ E_H ∧ d_H(u) = 1 ∧ d_H(v) = 1}  ▷ O(m)
20:    circuito = reconstruirCircuito(E_H, v_0)
21:    return circuito, cantidadNodos, pesoDelCamino                 ▷ O(1)
```

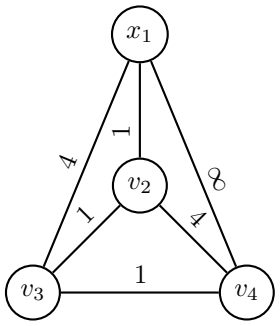
Complejidad: $\mathcal{O}(n^2 logn)$

```
1: function reconstruirCircuito(E, v_0)
2:     res = {v_0}
3:     nodoActual = v_0
4:     for 0 ≤ i <| E | −1 do                          ▷ O(| E |) = O(n)
5:         nuevaArista = (nodoActual, v)con(nodoActual, v) ∈ E       ▷ O(n)
6:         nodoActual = v                                            ▷ O(1)
7:         res.agregarAtras(nodoActual)                              ▷ O(1)
8:         E = E \ {nuevaArista}                                     ▷ O(n)
9:     endfor                                    ▷ Complejidad del ciclo: O(n^2)
10:    return res                                                    ▷ O(1)
```

Complejidad: $\mathcal{O}(n^2)$

Comenzaremos analizando con mayor detalle la complejidad del algoritmo 2. Como primer paso, ordenamos las aristas del grafo G por costo, cuya complejidad es $\mathcal{O}(mlogm)$=$\mathcal{O}(n^2 logn^2)$, ya que G es completo. Luego, tenemos un ciclo, en el que, a lo sumo, recorremos cada arista una vez, es decir que pertenece a $\mathcal{O}(m) = \mathcal{O}(n^2)$. Por último, buscamos el eje que cierra el circuito hamiltoniano a devolver, y para esto, como mucho recorremos el conjunto de aristas de G. Sumando estas complejidades, obtenemos que la heurística del menor costo pertenece a $\mathcal{O}(n^2 logn)$.

A la hora de buscar alguna familia que "rompa" nuestro algoritmo, vemos que no es difícil darnos cuenta que tomando la familia descrita en la sección 2 podemos conseguir que el algoritmo nos de resultados infinitamente distantes a la solución óptima. De esta forma, logramos hallar una serie de instancias para las que esta heurística no puede retornar nunca la mejor solución, en particular puede devolvernos soluciones tan malas como queramos.Veamos cómo es que esto sucede con un ejemplo:



Lo primero que hace el algoritmo es ordenar las aristas del grafo, que en nuestro caso nos retornará $[(v_1, v_2), (v_2, v_3), (v_3, v_4), (v_1, v_3), (v_2, v_4), (v_1, v_4)]$. Luego, iremos agregando las aristas en orden, siempre y cuando no hagan que algún vértice se recorra más de una vez o que se forme un ciclo dentro de nuestro camino hamiltoniano. Este procedimiento hará que incluyamos en nuestro camino a las aristas $(v_1, v_2)(v_2, v_3)(v_3, v_4)$, y al momento de querer cerrar el camino, el algoritmo tendrá que incluír sí o sí a la arista (v_1, v_4), cuyo costo es infinito. Luego, habremos obtenido un circuito hamiltoniano de costo infinito.

4. Heurística del árbol generador

Podemos observar que una cota inferior para el peso del camino hamiltoniano de menor costo es el costo del árbol generador mínimo. Esto es así porque el AGM es el árbol de menor costo que está incluido en el grafo, y cualquier recorrido hamiltoniano tiene un árbol como subgrafo (podemos pensar que el recorrido es un grafo conexo y con un circuito simple, por lo que removiendo una arista obtenemos un grafo conexo y sin ciclos, es decir, un árbol). Esto nos resulta importante porque conocemos algoritmos eficientes para poder calcular el AGM de cualquier grafo conexo (como por ejemplo el algoritmo de Prim, cuya complejidad es de orden cuadrático en la cantidad de nodos).

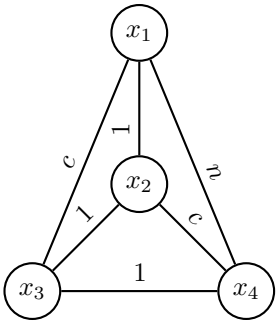
Una vez establecida esta cota inferior podemos buscar un camino hamiltoniano que se acerque lo mas posible a esta cota, y por extensión, a la solución óptima. Para intentar aproximarnos a esta cota aplicamos el algoritmo *DFS* sobre la raíz del AGM T guardando los vértices en el orden en el que se van visitando. Al aplicar este proceso obtenemos un recorrido que empieza y termina en la raíz de T, y que pasa por cada arista del AGM exactamente dos veces. Podemos notar que este recorrido tiene el doble de peso que T, dado que contiene los mismos vértices pero recorre dos veces cada arista. Además, se puede demostrar que $l(C) \leq 2P_T$ para todo C circuito hamiltoniano y P_T el peso de un arbol generador mínimo cuando G es un grafo euclidiano (es decir, que sus aristas cumplen la desigualdad triangular).

Ahora solo nos falta generar un camino hamiltoniano a partir de estos datos. Esto lo podemos lograr recorriendo los vértices del grafo en el mismo orden en el que aparecen en el recorrido del DFS, pero omitiendo los vértices repetidos. Esto nos permite encontrar un ciclo en G, pero no resulta inmediatamente obvio por qué debería ser una buena aproximación para el circuito hamiltoniano óptimo. Resulta importante observar que si las aristas del grafo original cumplen la desigualdad triangular, entonces al recorrer los vértices del DFS omitiendo los repetidos estamos omitiendo vértices intermedios. Es decir, al omitir un vértice repetido en la lista generada por el DFS estamos omitiendo puntos intermedios en el recorrido entre dos vértices, lo cual implica que el ciclo resultante va a tener un peso menor o igual al del DFS. Todo esto nos dice que si generamos el circuito de esta manera sobre un grafo cuyas aristas cumplan la desigualdad triangular, entonces nuestra solución va a ser 1-aproximada, dado que va a estar contenida en el intervalo $[P_T, 2P_T]$, en el cual ya sabemos que también está la solución óptima.

También podemos observar que cuanto más "lineal"sea el árbol, es decir, cuantas menos ramas tenga, más se va a parecer al recorrido hamiltoniano generado por este algoritmo. Esto nos lleva a pensar que cuantas más ramas tenga el AGM, más aristas que no se encontraban en el AGM van

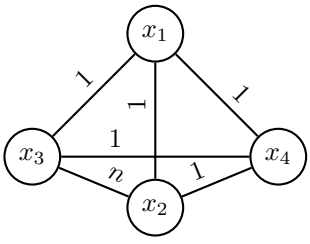
a ser agregadas al recorrido pudiendo influenciar negativamente en el costo del recorrido sin que el algoritmo contemple otras opciones. Esto podría generar que, para cierta familia de instancias, este algoritmo se comporte peor que otras heurísticas.

Si el grafo no respetara la desigualdad triangular, entonces esta heurística deja de ser 1-aproximada y se puede volver arbitrariamente mala:



Podemos ver en el grafo anterior que si tomamos n y c tal que $1 < c < \frac{n+1}{2}$, podemos hacer que el peso del circuito hamiltoniano calculado por el algoritmo anterior sea arbitrariamente malo. Esto resulta porque si $c, n > 1$ el AGM siempre va a consistir de las aristas de peso 1, y al aplicar DFS y cerrar el circuito se obtendrá un camino de peso $3 + n$, el cual puede ser mucho peor que $2c + 2$ (el peso de un camino hamiltoniano alternativo) dependiendo de como se elijan estos parámetros.

Adicionalmente, vamos a definir una familia de instancias que creemos debería llegar a malos resultados con la heurística basada en AGM pero no así con la heurística del vecino más cercano. La instancia se define de la siguiente manera: se empieza con un árbol de 2 niveles, la raíz y todos los demás vértices del grafo (los mismos deben ser por lo menos 3) son hijos de la raíz. Luego, conectamos todos los vértices del segundo nivel en serie con aristas de peso 1, a excepción de la que conecta el primer hijo de la raíz al segundo, que tendrá peso n. Adicionalmente, se agrega una arista de peso 1 entre el primer hijo de la raíz y el último. Todas las demás aristas que no existan hasta el momento tendrán peso $n+1$. La idea es que una vez planteado el árbol generador mínimo (que es la raíz con todos los vértices como hijos), la heurística basada en AGM deberá recorrer la arista de peso n para pasar de un hijo de la raíz al siguiente, mientras que la heurística del vecino más cercano podrá pasar por la arista de peso 1. Es importante resaltar que este caso resulta patológico en nuestra implementación de estos algoritmos, donde el orden en el que se recorren los vértices está dado por el índice de cada uno. Pero un caso patológico de la misma clase podría plantearse para cualquier implementación de la heurística AGM que recorra los vértices en un orden específico. A continuación se muestra un ejemplo de la instancia descripta para 4 vértices:



Formalizando, podemos plantear el siguiente pseudocódigo para el algoritmo:

Algorithm 4 Heurística del camino hamiltoniano utilizando AGM

```
1: function HEURISTICAAGM(G, v_0)
2:     T = AGM_PRIM(G)                          ▷ O(n^2)
3:     R = verticesDFS(T)                       ▷ O(n^2)
4:     return eliminarRepetidos(R)              ▷ O(n)
   Complejidad: O(n^2)
```

Si bien el algoritmo es simple, su complejidad depende fuertemente de como se implemente la búsqueda del AGM y la DFS. Para la implementación de AGM usamos el algoritmo de Prim,en su versión para grafos densos que tiene una complejidad de $\mathcal{O}(n^2)$. Para generar el camino en DFS, la complejidad es $\mathcal{O}(n + m)$, dado que el grafo es denso, también resulta igual a $\mathcal{O}(n^2)$. Como podemos observar la complejidad de este algoritmo resulta muy acotada y en el caso de grafos que cumplan la desigualdad triangular (como sucede cuando trabajamos sobre grafos que modelen mapas, distancias, etc.) nos permite saber que nuestra solución es como máximo el doble de pesada que la solución óptima. Si bien un error del 100 % con respecto a la solución óptima no nos resulta satisfactoria, por lo menos nos permite saber que esta heurística, a diferencia de las anteriores, no puede ser arbitrariamente mala (de nuevo, suponiendo un grafo que cumpla con las característica deseadas).

5. Metaheurística tabú search

Los algoritmos que se presentan a continuación se basan en una meta-heurística conocida como *Tabú Search*. La misma se basa en una búsqueda de óptimo local. Para esto definimos el espacio de soluciones factibles, y luego definimos un concepto de "vecindad" entre soluciones. Luego, empezando por una solución arbitraria, vamos moviéndonos a la mejor de las soluciones vecinas.

Este tipo de búsquedas nos provee una forma conveniente de encontrar mínimos locales, dado que siempre nos vamos a estar moviendo a la mejor solución candidata de un vecindario dado. Para evitar caer en un caso donde nuestra búsqueda se limita a encontrar un óptimo local ignorando otros vecindarios de búsqueda (donde podría haber otro óptimo local más bajo, o inclusive el óptimo global) vamos a limitar el tamaño de la vecindad que tomamos a un porcentaje dado (el mismo pasará a ser un parámetro del algoritmo). Luego, las soluciones vecinas que exploraremos van a ser seleccionadas aleatoriamente entre todas las vecinas y esto nos permitirá movernos en direcciones que no necesariamente tienden al mínimo local más cercano. Al introducir este cambio podemos llegar a recaer en una situación donde se explora un subconjunto del vecindario donde se encuentra una solución óptima, y luego se explora un subconjunto del vecindario en el cual la solución óptima era la anterior a la actual. Estas situaciones podrían hacer que nuestro algoritmo oscile entre dos soluciones cercanas sin lograr ningún progreso real. Para evitar este tipo de situaciones también vamos a implementar una estructura de memoria, en la cual vamos a "recordar" las soluciones por las cuales vamos pasando para evitar repetirlas en el futuro. Claramente el tamaño de la estructura de memorización es acotado y resulta necesario olvidar soluciones previas eventualmente (el tamaño de la memoria también pasa a ser un parámetro del algoritmo, dado que no resulta inmediatamente obvio que utilizar más memoria nos permita obtener mejores soluciones).

En nuestro caso, las soluciones factibles son permutaciones de los n vértices del grafo que debemos recorrer. Dada una solución factible, las soluciones vecinas son las que surgen de tomar dos aristas $e = (u, v)$ y $e' = (u', v')$ que se encuentren en nuestro circuito y rearmarlo tal que la nueva solución incluya las aristas (u, u') y (v, v'). Veamos esto con un ejemplo más concreto: Sea C el camino $C = v_0v_1v_2v_3v_4v_5$, si intercambiamos las aristas (v_0, v_1) y (v_3, v_4), el camino resultante es $C = v_0v_3v_2v_1v_4v_5$. Dado el vecindario de una solución factible, vamos a decidir si tener en cuenta a cada vecino en la búsqueda del óptimo dentro del vecindario con una probabilidad p (recibida por parámetro) para evitar tener en cuenta a todo el vecindario en cada iteración. Algo importante a destacar es que sin un criterio de parada externo (independiente de las soluciones visitadas, la memoria etc.) no hay nada que garantice que el algoritmo no cicle indefinidamente (por ejemplo,

visitando vecinos nuevos hasta olvidar el primero por el que pasó y luego volver a visitarlo), por lo cual resulta necesario establecer un criterio de parada fijo para garantizar la finalización del algoritmo. Por esta razón decidimos establecer un límite para la cantidad de iteraciones, este es otro parámetro a definir al momento de ejecutar el algoritmo. Lo único que resta definir es el modo en el cual se memorizarán los pasos dados por el algoritmo para evitar repetirlos. Para esto definimos dos criterios posibles, en ambos casos el tamaño de la memoria también resulta ser un parámetro del algoritmo.

5.1. Memoria por soluciones

En esta versión del algoritmo vamos a memorizar cada circuito hamiltoniano por el cual pasamos (cada uno que elegimos de entre sus vecinos) y agregarlo a una lista de memorización. Luego, al momento de recorrer un vecindario de una solución factible dada, vamos a ignorar a todos los elementos que se encuentren en esta lista. Cuando se llene esta lista vamos a remover al elemento que haya sido ingresado hace más tiempo (en orden FIFO).

5.2. Memoria por estructura

En esta segunda versión vamos a memorizar cuales aristas intercambiamos para pasar de una solución a su vecina, y luego evitaremos repetir esos mismos cambios. Esta segunda forma de memorizar puede resultar más contraintuitiva, dado que al memorizar solo los cambios es posible que lleguemos a visitar la misma solución dos veces, siempre y cuando sea posible volver a una solución anterior realizando intercambios diferentes. Sin embargo, esta forma de memorizar nos resulta interesante dado que al restringir únicamente los cambios que realizamos tenemos un criterio alternativo para explorar el espacio de soluciones. La efectividad de este método será explorada en profundidad en las siguientes secciones.

5.3. Pseudo-código

Algorithm 5 Meta-heurística Tabú Search

```
1:  function HEURISTICATABUSEARCH(G, v_0, mem_max, vec_max, i_max)
2:      actual ← heuristicaInicial(G, v_0)                          ▷ O(n^2 log n)
3:      opt ← actual                                                ▷ O(1)
4:      memoria ← lista(mem_max)                                    ▷ O(mem_max)
5:      while (i < i_max) do                                        ▷ O(i_max)
6:          vec ← vecindario(actual, vec_max)                       ▷ O(n^2)
7:          vecino ← nuevoOptimo(vec, mem, mem_max)                 ▷ O(n^3)
8:          if memoria por solucion then
9:              recordarSwap(actual, vecino, mem, mem_max)          ▷ O(mem_max)
10:         else
11:             recordarSolucion(vecino, mem, mem_max)              ▷ O(mem_max)
12:         actual ← vecino                                         ▷ O(1)
13:         if actual < opt then
14:             opt ← actual                                        ▷ O(1)
15:         i ← i + 1
16:     return opt
    Complejidad: O(i_max(n^3 + mem_max))
```

Varios detalles implementativos fueron omitidos del pseudo-código 5 por claridad, a continuación vamos a destacar los más importantes:

- **Heurística inicial**: Es necesario partir de algún circuito hamiltoniano, por lo cual nosotros decidimos ejecutar las 3 heurísticas anteriores y elegir la de menor costo. Esto significa que

la cota teórica de la complejidad de la heurística inicial en el pseudo-código es la peor de las cotas teóricas de las heurísticas.

- **Selección del vecindario**: Para evitar recorrer todo el vecindario de una solución innecesariamente, decidimos utilizar una variable aleatoria que nos dicta cuales de los vecinos debemos omitir. De esta manera podemos explorar parcialmente la vecindad de manera aleatoria sin recurrir a técnicas más costosas. Esto tiene la desventaja de que en el peor caso se evalúa toda la vecindad, lo cual tiene complejidad $\mathcal{O}(n^2)$ (aunque este caso es prácticamente imposible si el porcentaje de la vecindad a visitar no es 100 %).

- **Complejidad de buscar al mejor vecino**: Por cada vecino posible (es decir, por cada par de aristas a intercambiar) debemos reconstruir un circuito hamiltoniano con las nuevas aristas. Dado que reconstruir un circuito tiene complejidad $\mathcal{O}(n)$ una vez determinadas las aristas a utilizar, la complejidad de buscar al mejor vecino implica, en el peor caso, evaluar a los n^2 vecinos posibles realizando n operaciones por cada uno. Es por esta razón que decimos que *nuevoOptimo* tiene complejidad $\mathcal{O}(n^3)$.

- **Olvidar pasos**: En el pseudo-código la función *recordarSolucion* toma por parametro mem_{max}. Esto es porque estamos asumiendo que esta función olvida al primer vecino visitado en caso de que el tamaño de la estructura exceda este último parámetro. Esto explica también porque decimos que su complejidad es $\mathcal{O}(mem_{max})$, dado que decidimos implementarlo sobre un vector.

6. Experimentación

6.1. Toma de mediciones

Las mediciones fueron realizadas en una PC con CPU Intel Core i7 @ 2.6 GHz que cuenta con 6 cores y 12 hilos ya que el mismo cuenta con *hyperthreading* y 16 GB de memoria RAM, utilizando el lenguaje de programación *C++* y corriendo Ubuntu 20.04.

Para medir el gap relativo entre el peso del circuito hamiltoniano encontrado y el óptimo utilizamos los datos disponibles en TSPLIB y los contrastamos con el óptimo obtenido mediante nuestros algoritmos (comparamos contra el óptimo en la mayoría de los casos, y en las 2 instancias en las cuales no se conocía el óptimo tomamos una cota inferior provista por dicha biblioteca).

A la hora de medir el tiempo de ejecución utilizamos la biblioteca *chrono* de *C++* para tomar el tiempo insumido por cada algoritmo, y luego tomamos el promedio de ejecutar 5 veces cada medición para intentar minimizar el ruido producido por tareas programadas del sistema operativo, procesos ejecutandose en el fondo, etc.

6.2. Instancias utilizadas

Para llevar a cabo la experimentación utilizamos tanto las instancias disponibles en la biblioteca de instancias TSPLIB como instancias sintéticas (las cuales serán descriptas más adelante) generadas para exhibir los que nosotros creemos que serán los peores casos de cada algoritmo. Como mencionamos brevemente en la sección 1, este problema nos presenta un espacio de soluciones factibles tan vasto que inclusive para instancias muy pequeñas los tiempos computacionales pueden volverse prohibitivos muy rápidamente. Es por esta razón que decidimos trabajar mayoritariamente con instancias que estén por debajo de los 490 vértices para experimentar con diferentes parámetros y llevar a cabo las comparaciones entre algoritmos. Más adelante se evaluará también si estos resultados obtenidos en instancias comparativamente pequeñas generalizan a instancias de mayor tamaño (tomando instancias de tamaño mayor a 490 pero menor a 3000).

6.2.1. Training vs Validation

Dado que debemos buscar parámetros óptimos para los algoritmos basados en Tabú Search, nos vemos en la necesidad de poner a prueba los parámetros óptimos obtenidos sobre un dataset diferente al utilizado para establecer dichos valores. Es por esto que dividimos el dataset de instancias pequeñas anteriormente descripto en un 70 % que llamaremos **training** y un 30 % que denominaremos **validation** (estos conjuntos fueron elegidos de manera tal que en **training** haya instancias de tamaños variados y que resulten representativos). A la hora de encontrar la combinación óptima de parámetros de Tabú Search vamos a buscar los que minimicen el *gap* entre la solución encontrada y la óptima en las instancias **training**. Luego pondremos a prueba la generalidad de estos parámetros ejecutando dicha parametrización del algoritmo sobre las instancias **validation**.

A continuación se listan las instancias incluidas en cada dataset:

- **training**: gr96, ulysses16, gr202, gr229, eil51, pr226, rat195, pr439, gil262, pr264, berlin52, a280, kroB150, ch150, d198, kroA100, eil76, lin318, pr152, rat99, kroE100, lin105, ts225, bier127, u159, ch130, kroB100, kroA150, pcb442, pr76, eil101, pr136, st70, kroC100, att48.
- **validation**: pr124, kroB200, pr299, pr107, kroD100, burma14, kroA200, pr144, gr137, gr431, ulysses22.tsp, rd400, rd100, fl417, tsp225.

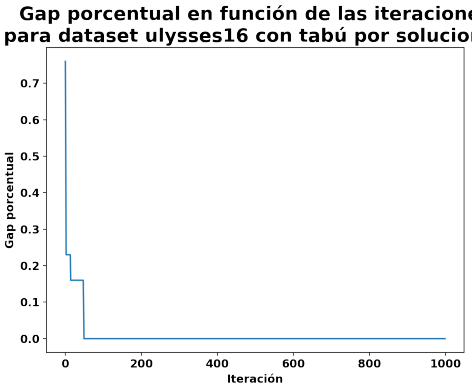
6.3. Parametrización de Tabú Search

Antes de poder hacer un análisis de la calidad de las soluciones encontradas por cada algoritmo o de los tiempos insumidos por cada uno, nos avocaremos a encontrar los parámetros óptimos para los algoritmos basados en la meta-heurística Tabú Search.

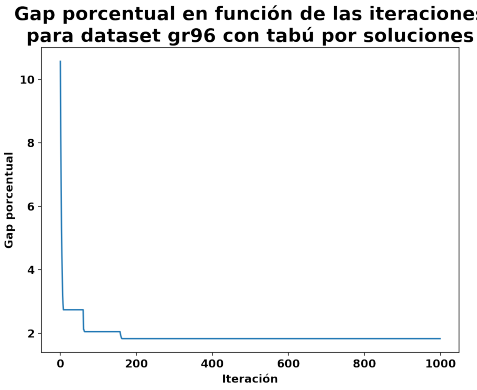
6.3.1. Cantidad de iteraciones

El primer parámetro que buscamos establecer es la cantidad de iteraciones máxima permitidas. Para lograr determinarlo ejecutamos cada versión del algoritmo basado en Tabú Search registrando el mínimo histórico en cada iteración. Para correr este experimento, el tamaño de la memoria fue fijado en 1000 y el porcentaje de la vecindad en 100 % (elegimos estos valores ya que creemos que no influirán en gran medida a la hora de determinar la cantidad necesaria de iteraciones para que el óptimo encontrado por el algoritmo se estabilice). Por otra parte, los algoritmos serán aplicados sobre instancias de distintos tamaños (ulysses16, gr96, ts225, pcb442), con el objetivo de conseguir resultados más representativos.

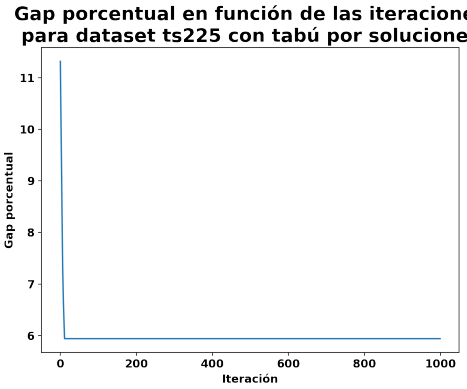
Creemos que mientras no se halle el óptimo, los algoritmos siempre irán encontrando mejoras, aunque sean muy pequeñas. Sin embargo, estamos interesados en determinar el punto en el que dejan de verse mejoras sustanciales. Por otra parte, si bien esperamos obtener resultados heterogéneos, esperamos poder determinar una buena cota para el valor del parámetro en cuestión de forma tal que permita que la mayor parte de las instancias logren obtener soluciones cercanas a sus óptimos, a pesar de que estas puedan mejorar con un número mayor de iteraciones.



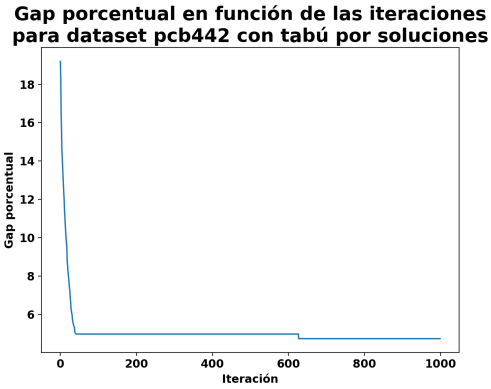
(a) Gap vs cantidad de iteraciones en búsqueda tabú con memoria por soluciones para un grafo de 16 nodos.



(b) Gap vs cantidad de iteraciones en búsqueda tabú con memoria por soluciones para un grafo de 96 nodos.

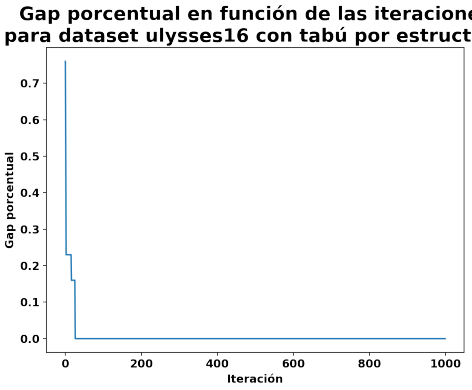


(c) Gap vs cantidad de iteraciones en búsqueda tabú con memoria por soluciones para un grafo de 225 nodos.

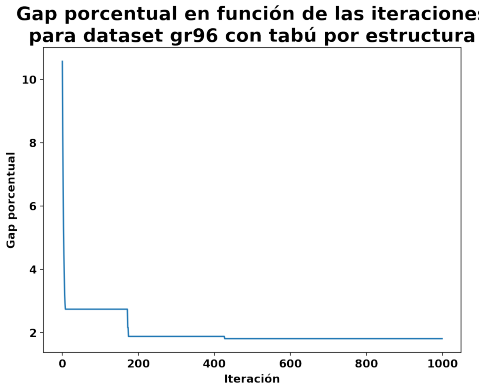


(d) Gap vs cantidad de iteraciones en búsqueda tabú con memoria por soluciones para un grafo de 442 nodos.

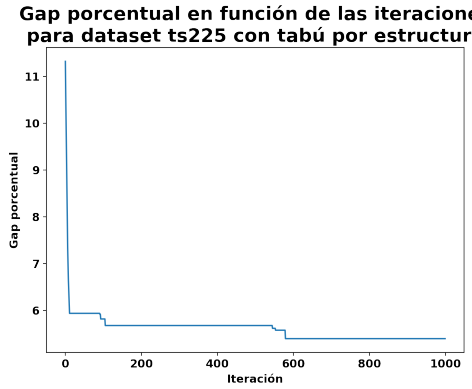
Figura 1: Comportamiento de las búsqueda tabú con memoria por soluciones para distinta cantidad de iteraciones.



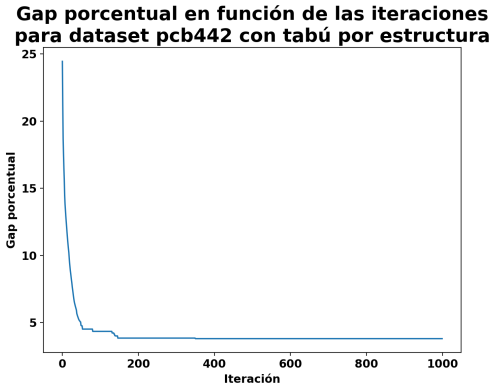
(a) Gap vs cantidad de iteraciones en búsqueda tabú con memoria por estructura para un grafo de 16 nodos.



(b) Gap vs cantidad de iteraciones en búsqueda tabú con memoria por estructura para un grafo de 96 nodos.



(c) Gap vs cantidad de iteraciones en búsqueda tabú con memoria por estructura para un grafo de 225 nodos.



(d) Gap vs cantidad de iteraciones en búsqueda tabú con memoria por estructura para un grafo de 442 nodos.

Figura 2: Comportamiento de las búsqueda tabú con memoria por estructura para distinta cantidad de iteraciones.

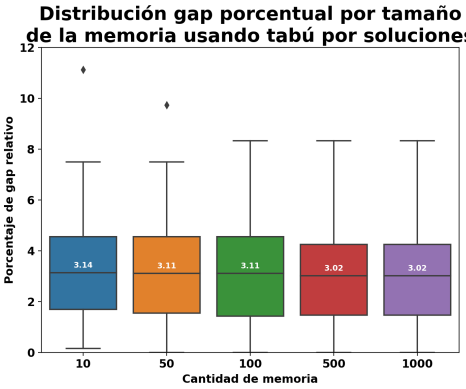
Como podemos observar, tanto en la figura 1 como en la figura 2, ambos algoritmos encuentran las mejoras más sustanciales en las primeras iteraciones y a medida que aumentan las iteraciones, continúan hallando soluciones que se acercan aún más al óptimo. No obstante, vemos que el porcentaje de gap que se logra disminuír con un número grande de iteraciones, no resulta lo suficientemente significativo, según nuestros objetivos, como para que valga la pena setear el parámetro a determinar cercano a estos valores. Por esta razón, decidimos continuar los experimentos corriendo los algoritmos de Tabú Search con 300 iteraciones.

6.3.2. Tamaño de la memoria

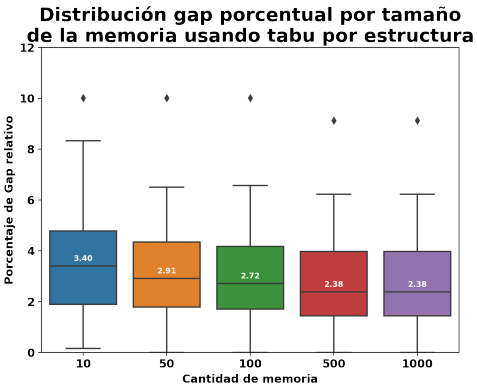
Como se explicó en la sección 5, la relación entre el tamaño de la memoria y la calidad de la respuesta no resulta inmediatamente obvia. Este fenómeno se acentúa aún más cuando ejecutamos la búsqueda tabú con memoria por estructura, dado que en ese caso resulta más contra intuitivo pensar cómo se comportará el algoritmo. Otro factor a tener en cuenta es que incrementar el tamaño de la memoria tiene la desventaja de aumentar la complejidad de los tiempos de ejecución, dado que es necesario inicializar y actualizar una estructura más grande. Para buscar el tamaño de memoria óptimo fijamos la cantidad de iteraciones en 300 basándonos en los resultados anteriores y el tamaño de la vecindad en 100 %. Luego, ejecutamos el algoritmo variando únicamente el tamaño de la memoria entre los valores 10, 50, 100, 500, 1000.

Es importante destacar que, debido a que buscamos el tamaño de memoria que minimice el gap en el caso promedio, vamos a ejecutar el algoritmo sobre todas las instancias comprendidas en

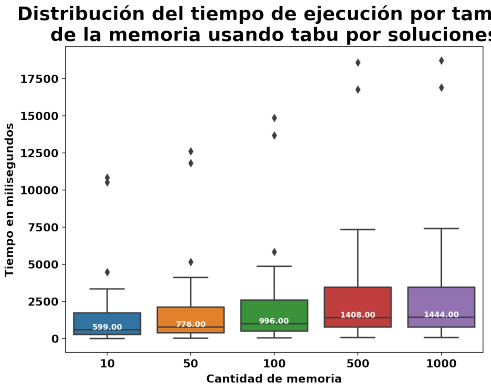
training. Como este dataset contiene instancias muy heterogéneas y de tamaños muy variados, esperamos que los resultados tengan un nivel importante de dispersión, pero no vamos a agrupar los resultados de ninguna manera en particular debido a que buscamos los parámetros que mejoren el caso promedio.



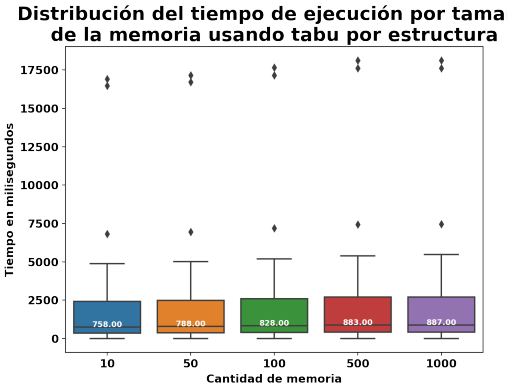
(a) Gap vs tamaño de memoria en búsqueda tabú con memoria por soluciones.



(b) Gap vs tamaño de memoria en búsqueda tabú con memoria por estructura.



(c) Tiempo de ejecución vs tamaño de memoria en búsqueda tabú con memoria por soluciones.



(d) Tiempo de ejecución vs tamaño de memoria en búsqueda tabú con memoria por estructura.

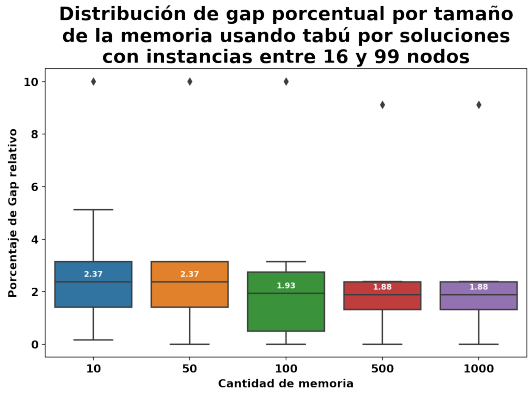
Figura 3: Comportamiento de los algoritmos de búsqueda tabú con distintos tamaños de memoria.

Vemos en la figura 3 que hay una clara tendencia del gap porcentual a disminuir a medida que tomamos tamaños de memoria progresivamente más grandes. Sin embargo, esta tendencia resulta muy poco pronunciada en el caso de la búsqueda tabú con memoria por soluciones. Es importante destacar que es en ese último caso donde el tamaño de la memoria tiene un mayor impacto en los tiempos de ejecución.

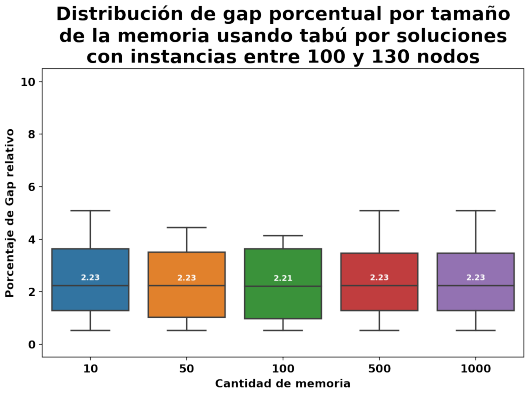
Adicionalmente, podemos observar que encontramos muchos outliers en cuanto al gap porcentual para un mismo tamaño de memoria (lo cual coincide con lo que estábamos esperando). Esto nos resulta poco preocupante dado que estamos realizando mediciones sobre grafos de tamaños muy variados (las instancias incluidas en **training** tienen tamaños desde 16 hasta 442), por lo cual resulta esperable que los comportamientos no sean uniformes a lo largo de todas las instancias. Lo mismo ocurre con los tiempos de ejecución, aquí no solo interviene el tamaño de las instancias, sino que además el experimento ejecutó durante algunas horas, por lo que resulta natural que alguna tarea programa o una interacción mínima con el sistema haya afectado a las mediciones.

Creemos que los resultados expuestos por la figura 3a no son lo suficientemente concluyentes como para poder establecer cuál es la cantidad de memoria óptima para correr Tabú Search con memoria por soluciones. Por otra parte, sospechamos que el tamaño de la memoria comienza a tener una mayor influencia cuando trabajamos con grafos más grandes. Por esta razón, correremos un

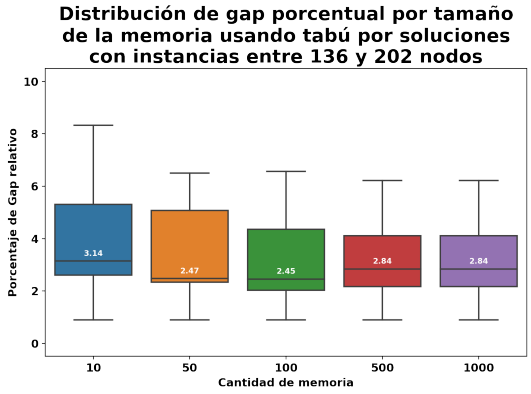
nuevo experimento, en donde analizaremos cómo se comporta el algoritmo con los mismos valores posibles para la cantidad de memoria a utilizar si se corre sobre instancias cuyo tamaños sean más homogéneos, con el objetivo de verificar (o refutar) nuestra nueva hipótesis.



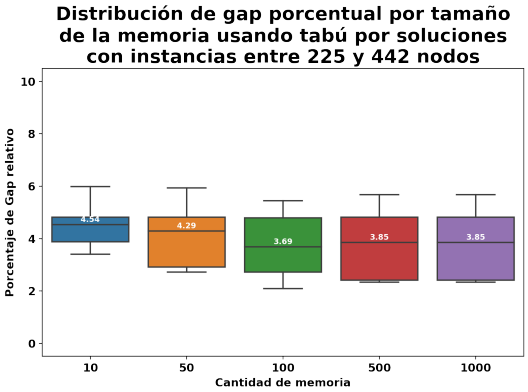
(a) Gap vs tamaño de memoria en búsqueda tabú con memoria por soluciones con grafos pequeños.



(b) Gap vs tamaño de memoria en búsqueda tabú con memoria por soluciones con grafos medianos.



(c) Gap vs tamaño de memoria en búsqueda tabú con memoria por soluciones con grafos medianamente grandes.



(d) Gap vs tamaño de memoria en búsqueda tabú con memoria por soluciones con grafos grandes.

Figura 4: Comportamiento de las búsqueda tabú con memoria por soluciones con distintos tamaños de memoria sobre instancias de tamaño similar.

Analizando los gráficos de la figura 4, notamos que nuestra hipótesis resultó no ser cierta, ya que no logramos ver una diferencia significativa entre las relaciones cantidad de memoria-gap de los distintintos tamaños. Luego, vamos a elegir este parámetro basándonos únicamente en el tiempo de ejecución que implican.

En vista de estos resultados, llegamos a la conclusión de que aumentar el tamaño de la memoria atiene un fuerte impacto en el tiempo de cómputo requerido para la búsqueda tabú con memoria por soluciones, mientras que los aumentos en la calidad de la solución que logra son marginales. Por esta razón decidimos establecer provisionalmente el tamaño de la memoria en 50 para el caso de la memoria por soluciones. Con respecto a la versión que utiliza memoria por estructura, vemos que aumentar el tamaño de la memoria tiene un efecto despreciable en el tiempo de ejecución, mientras que nos provee con respuestas mucho más cercanas a la óptima. Es por esta razón que nos permitimos fijar provisionalmente el tamaño de la memoria por estructura en 500.

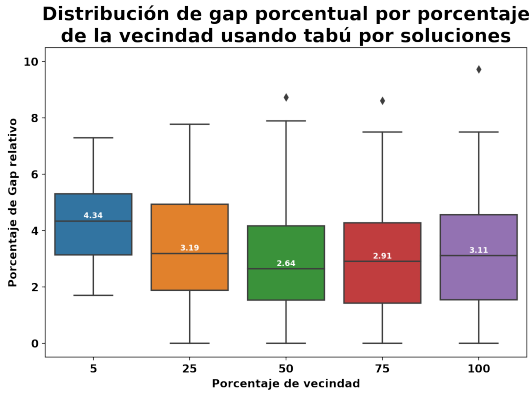
Esto lo hacemos con el fin de proseguir con la parametrización, y estos parámetros serán evaluados más a fondo en la sección 6.3.4. También vale destacar que estas decisiones que tomamos estuvieron fuertemente influenciadas por los beneficios marginales que lográbamos al pagar un costo computacionalmente mayor. Pero en aplicaciones críticas donde inclusive pequeños porcen-

tajes de mejora sean altamente deseable sin importar su costo computacional, estos parámetros necesitarían ser redefinidos en base a un criterio más adecuado.

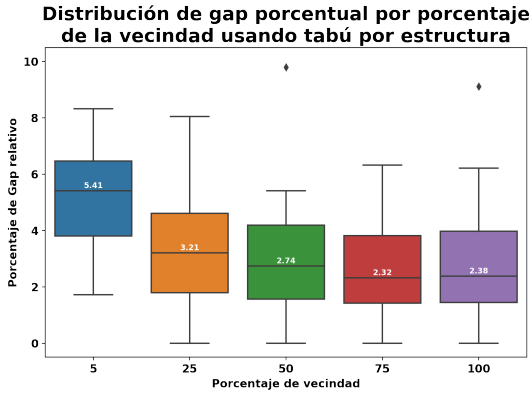
6.3.3. Porcentaje de la vecindad

El último punto a considerar, luego de haber establecido la cantidad óptima de memoria e iteraciones, es el porcentaje de la vecindad el cual vamos a evaluar. De manera análoga a las secciones anteriores vamos a ejecutar los algoritmos sobre las instancias **training** con los parámetros previamente definidos de memoria e iteraciones, variando el porcentaje de vecindad a considerar.

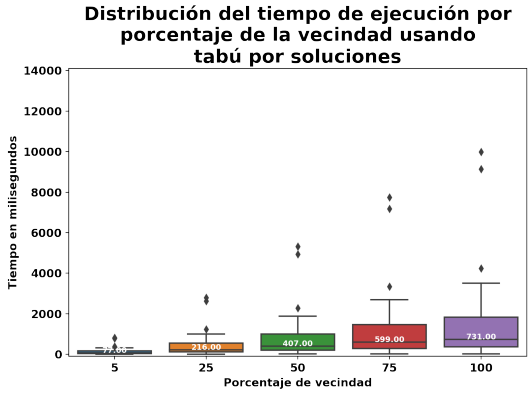
Los resultados esperados son análogos a los de la sección 6.3.2. Aumentar el porcentaje de vecinos a considerar debe impactar de manera significativa en los tiempos de ejecución, análogamente, esperamos que haya muchos outliers por la misma razón que en la sección anterior. A continuación se muestran los resultados.



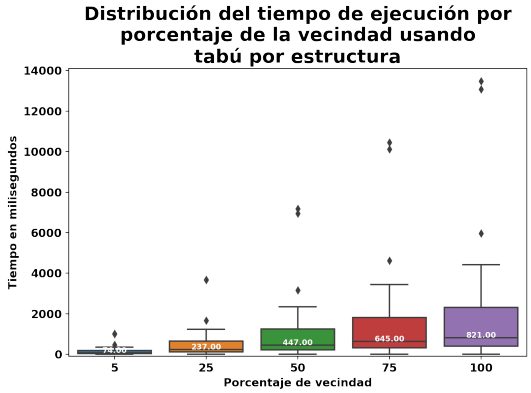
(a) Gap vs porcentaje de vecindad en búsqueda tabú con memoria por soluciones.



(b) Gap vs porcentaje de vecindad en búsqueda tabú con memoria por estructura.



(c) Tiempo de ejecución vs porcentaje de vecindad en búsqueda tabú con memoria por soluciones.



(d) Tiempo de ejecución vs porcentaje de vecindad en búsqueda tabú con memoria por estructura.

Figura 5: Comportamiento de las búsqueda tabú con distintos porcentajes de vecindad.

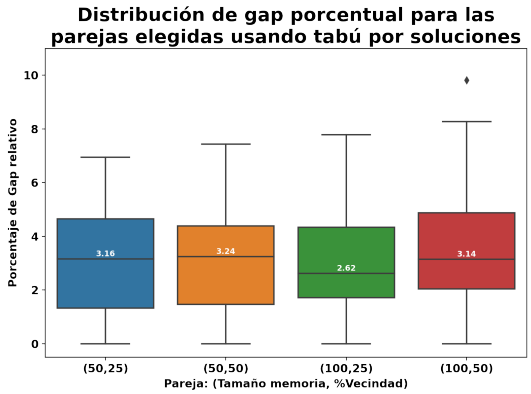
Podemos ver que, a diferencia de lo que ocurría con el tamaño de la memoria, aumentar el porcentaje de la vecindad visitado no siempre nos provee soluciones mejores. Esto es razonable, dado que es justamente lo que buscamos que ocurra al implementar Tabú Search y no simplemente una búsqueda del óptimo local. Podemos ver también que al incrementar el porcentaje de vecinos visitados el tiempo de ejecución aumenta considerablemente, lo cual también resulta razonable. Análogamente a la sección anterior, fijamos de manera provisional la vecindad de la memoria en 25 % y 50 % para la búsqueda tabú con memoria por soluciones y por estructura, respectivamente.

6.3.4. Validación de parámetros

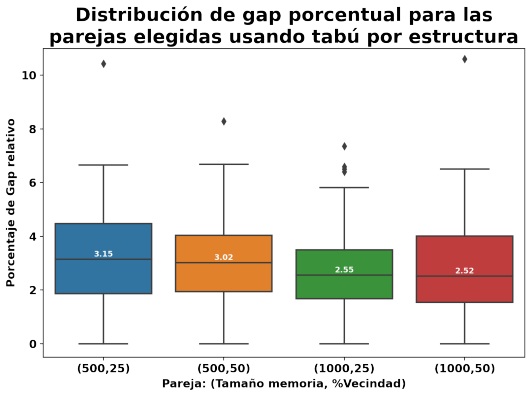
Habiendo analizado en los experimentos anteriores los cambios en la performance de las metaheurísticas al modificar sus parámetros, vemos que establecer valores para los mismos no es tan simple. Para asegurarnos de elegir los parámetros que mejor se ajusten a nuestros objetivos, decidimos evaluar cómo se comportan los algoritmos al trabajar con todas las combinaciones posibles entre dos valores para el tamaño de memoria y dos para el porcentaje de vecindad, para elegir así la pareja mś adecuada.

En el caso de la búsqueda tabú con memoria por soluciones, los valores de tamaño de memoria a probar fueron 50 y 100, y los porcentajes de vecindad fueron 25 y 50. Por otra parte, en el caso con memoria por estructura los tamaños de memoria elegidos fueron 500 y 1000, mientras que los porcentajes de vecindad fueron 25 y 50.

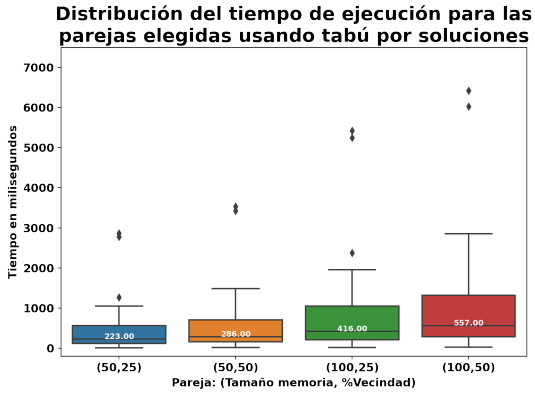
Para seleccionar estos valores tomamos aquellos que minimizaban el gap de la solución obtenida insumiendo un tiempo que nosotros consideramos adecuado, y luego tomamos al valor más cercano dentro de estos mismos limites a fin de evaluar si el cambio en la calidad de la solución se veía compensado por el cambio en los tiempos de ejecución. Es decir, la elección de estos valores no se dio exclusivamente por la calidad de los resultados obtenidos, ya que por ejemplo con un porcentaje de vecindad del 75 % para la búsqueda tabú con memoria por estructura, creemos que la mejora la solución obtenida no justifica la diferencia en el tiempo que insume, ya que el objetivo no es encontrar la mejor solución posible sino una solución lo suficientemente buena en un tiempo razonable.



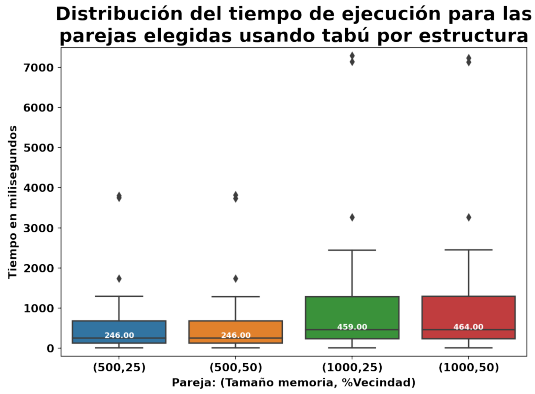
(a) Gap para las parejas de parámetros con memoria por solución.



(b) Gap para las parejas de parámetros con memoria por estructura.



(c) Tiempo de ejecución para las parejas de parámetros con memoria por solución.



(d) Tiempo de ejecución para las parejas de parámetros con memoria por estructura.

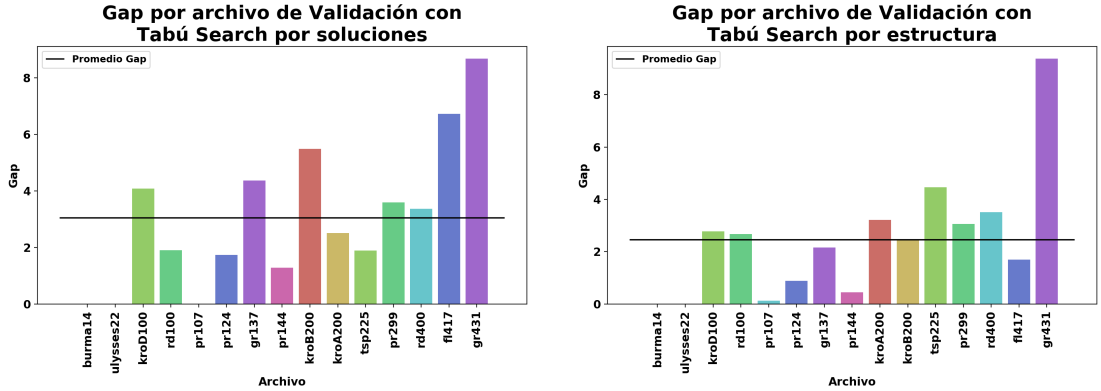
Figura 6: Comparación de parejas de parámetros para búsqueda tabú.

Con estos resultados decidimos que los parámetros que mejor se ajustan a nuestros criterios para

la búsqueda tabú con memoria por soluciones son un tamaño de memoria de 50 y un porcentaje de vecindad de 25, ya que viendo los gráficos de las figuras 6a y 6c, notamos que, si bien la mediana de la pareja $(100, 25)$ o de $(100, 50)$ son menores, la dispersión de los datos es mucho mayor, y en algunos casos, los valores obtenidos de gap resultan peores. Por otra parte, la cantidad de tiempo que insumen es más que el doble de lo que nos toma correr el alogritmo con los parámetros elegidos.

En el caso de búsqueda tabú con memoria por estructura, decidimos que los mejores parámetros los conseguimos tomando un tamaño de memoria de 500 y un porcentaje de vecindad de 50. Las figuras 6b y 6d nos muestran que, si bien pareciera que podemos obtener soluciones de mayor calidad tomando la pareja $(1000, 25)$, los tiempos de ejecución de la misma son muy altos comparados a los de la pareja elegida, y consideramos que las mejoras obtenidas no son los suficientemente considerables como para que valga la pena la cantidad de tiempo extra que nos consume.

A continuación validaremos los parámetros elegidos con el conjunto de instancias de validación. Esperamos ver que la pareja elegida logra generalizar bien cuando la utilizamos sobre instancias que no pertenecían al conjunto de training.



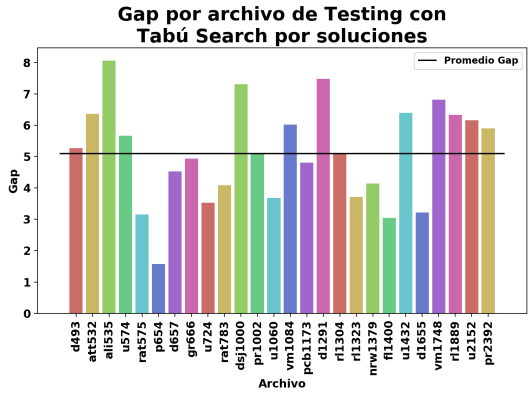
(a) Resultados de validación para memoria por solución tomando al 25 % de la vecinidad y utilizando memoria de tamaño 50.

(b) Resultados de validación para memoria por estructura tomando al 50 % de la vecinidad y utilizando memoria de tamaño 500.

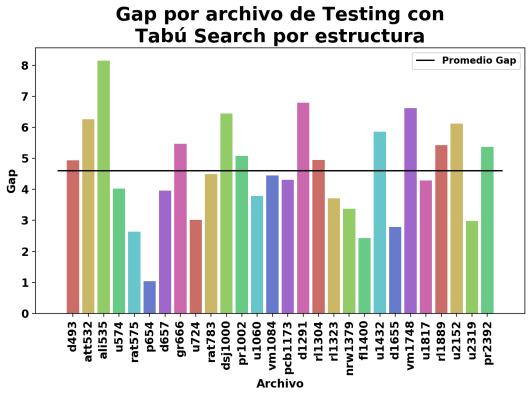
Figura 7: Gap porcentual para archivos de validación para búsqueda tabú utilizando los parámetros óptimos.

La figura 7 nos permite ver que los valores de gap obtenidos para las instancias de validación se condicen con los obtenidos para las instancias de training de la figuras 6a y 6b.

Sin embargo, para terminar de convencernos de que estos parámetros funcionan bien sobre todas nuestras instancias, los testearemos utilizando instancias de tamaños mayores (entre 490 y 3000 vértices) esperando obtener resultados similares a los obtenidos en las instancias de training y validación.



(a) Resultados de instancias grandes para memoria por solución tomando al 25 % de la vecindad y utilizando memoria de tamaño 50.



(b) Resultados de instancias grandes para memoria por estructura tomando al 50 % de la vecindad y utilizando memoria de tamaño 500.

Figura 8: Gap porcentual para archivos de instancias de mayor tamaño para búsqueda tabú utilizando los parámetros óptimos.

Para el caso de las instancias con cantidad de nodos mayor, se puede ver en la figura 10 que si bien los resultados son ligeramente peores a las instancias más pequeñas no es preocupante ya que la diferencia es poca y además esperable: por un lado, al tener más nodos el espacio de soluciones crece exponencialmente, lo que hace que encontrar una buena solución sea aún más díficil; y por otro lado, los parámetros fueron elegidos utilizando instancias mucho más pequeñas, y es razonable que no se ajusten tan bien a estas instancias que no fueron tomadas en cuenta para entrenar al modelo.

En conclusión, consideramos que la pareja obtenida para cada metaheurística logra que cada modelo funcione para una variedad de instancias grande, consiguiendo buenos resultados con tiempos de ejecución razonables. Por este motivo, las utilizaremos para establecer los parámetros de Tabú Search con memoria por soluciones y por estructura al momento de comparar todos los algoritmos en la próxima sección.

6.4. Comparación de la performance de los algoritmos

En esta sección, nos avocaremos a comparar los distintos algoritmos desarrollados en este trabajo. Es importante aclarar que Tabú Search con memoria por soluciones y por estructura serán corridos utilizando los parámetros óptimos hallados en la sección 6.3.4, los cuales resultaron ser porcentaje de vecindad 25 y cantidad de memoria 50 para la primera, y 50 y 500 para la segunda.

Como primer paso, comenzaremos por graficar el gap promedio conseguido por cada algoritmo. Esperamos ver que las heurísticas golosas, al igual que la heurística de AGM, arrojan un gap mucho mayor al obtenido con la búsqueda Tabú utilizando las dos variantes de memoria. Por otra parte, dentro de las heurísticas, creemos que la que peor funcionará será VMC, ya que pensamos que el hecho de que sea tan simple y rápida hace que la calidad de las soluciones obtenidas no sea tan buena.

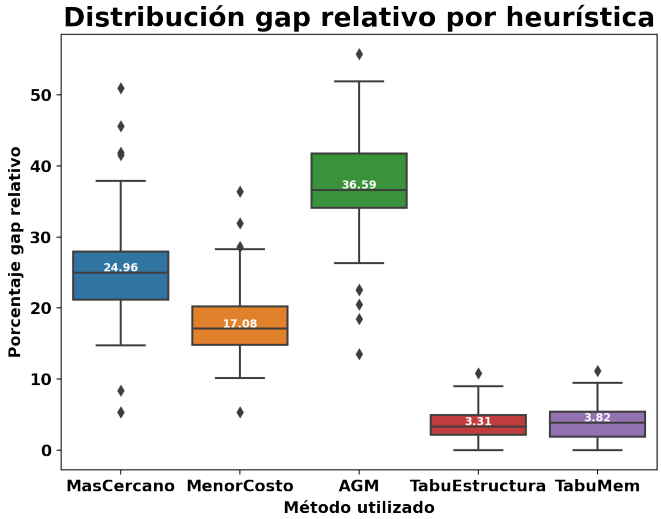
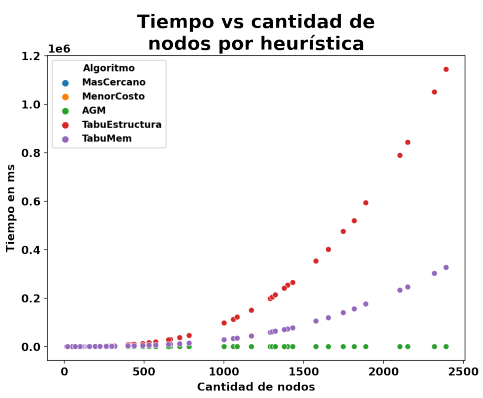


Figura 9: Gap promedio arrojado por los distintos algoritmos.

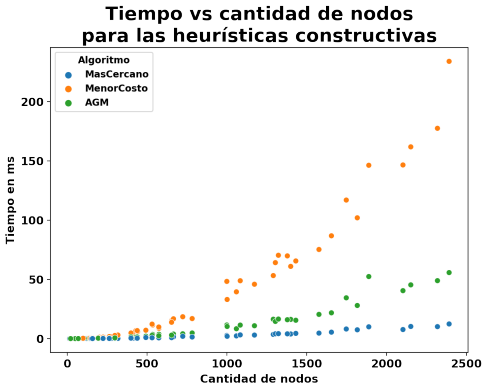
Como podemos observar en la figura 9, las metaheurísticas logran obtener resultados significativamente mejores que las heurísticas, como habíamos teorizado anteriormente. No obstante, nuestra teoría no resulta cierta con respecto a lo hipotetizado acerca del comportamentiendo de la heurística AGM, cuyas soluciones nos dieron un gap mayor a los de las demás heurísticas. Este último resultado nos hace cuestionarnos qué pudo haber ocurrido, ya que teníamos bastante confianza en nuestra hipótesis, por esta razón decidimos revisar el código para asegurarnos de no haber introducido algún bug durante la codificación del algoritmo.

Luego de auditar el código en busca de errores y comparar las distintas heurísticas manualmente para instancias de tamaño razonable (*burma14* y *berlin52*) pudimos apreciar que, en el caso de la heurística basada en AGM, el AGM se estaba construyendo de forma correcta, y el circuito hamiltoniano también era correcto. Esto nos llevó a revisar la implementación de las demás heurísticas (empezamos por revisar la implementación de la heurística AGM porque era la más compleja, y por lo tanto la más propensa a tener bugs) y, para nuestra sorpresa, tampoco pudimos hallar errores ni en el código, ni en los resultados obtenidos para estas instancias. Todo esto sugiere que potencialmente las heurísticas más simples pueden funcionar mejor que la basda en AGM en algunos casos. Dependiendo de en cuántos casos ocurra esto, y qué tan buena sea la respuesta arrojada por estas heurísticas, esto podría tener un fuerte impacto en la media de los gaps.

Continuaremos con la comparación de los algoritmos analizando ahora la complejidad temporal de cada uno. Creemos que habrá una gran diferencia entre los tiempos de ejecución de las heurísticas y los de las metaheurísticas, en donde los tiempos de las segundas serán significativamente mayores que los de las primeras.



(a) Tiempo de ejecución obtenido en función de la cantidad de nodos para los distintos algoritmos.



(b) Tiempo de ejecución obtenido en función de la cantidad de nodos enfoncándonos en las heurísticas golosas constructivas.

Figura 10: Tiempo de ejecución de cada algoritmo por cantidad de nodos.

Observando la figura 10a, notamos que efectivamente la complejidad temporal de las metaheurísticas supera ampliamente a la de las heurísticas. Además, si vemos el gráfico de la figura 10b, en la que se exponen con más detalle los tiempos de ejecución de las heurísticas, notamos que MenorCosto resulta ser la más lenta. esto se debe a que realiza un sort de las aristas, lo que hace que la complejidad del algoritmo sea $\mathcal{O}(n^2 logn)$ (como vimos en la sección 3), mientras que las otras son cuadráticas en la cantidad de nodos.

Por último, vamos a ver cuál es la relación entre el gap obtenido y la cantidad de tiempo que se ejecutan los algoritmos. Nuestra hipótesis es que las heurísticas tardarán poco tiempo en arribar a una solución, solución que será mala comparada con las de las dos implementaciones de Tabú Search. Para estas, esperamos que se vea una mayor cantidad de tiempo de ejecución y que el gap obtenido cuando son aplicadas sea pequeño.

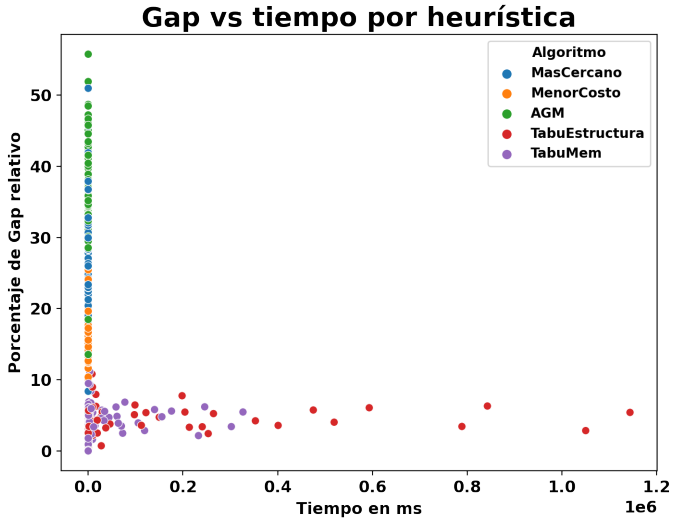


Figura 11: Relación entre el Gap promedio arrojado por los distintos algoritmos y su tiempo de ejecución.

Analizando la figura 11, confirmamos nuesta teoría: podemos ver que los porcentajes de gap correspondientes a las heurísticas son muy altos, mientras que el tiempo que les toma arribar a una solución es muy pequeño. Contrariamente, las metaheurísticas hallan soluciones mucho mejores

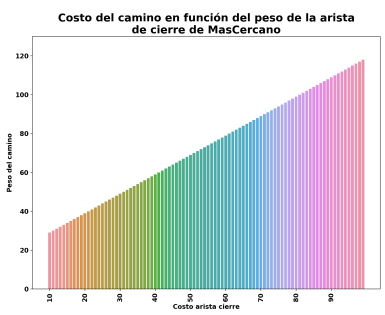
que las anteriores insumiendo un mayor tiempo de ejecución, que de todas formas, en la mayoría de los casos no resulta inconmesurable en vistas de la calidad de las soluciones que hallan.

6.5. Análisis de distintas familias patológicas

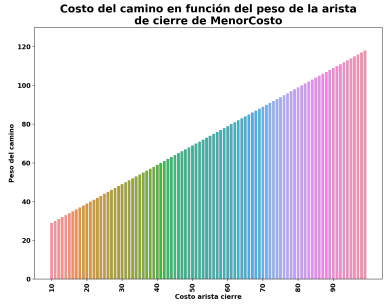
Ahora, nos encargaremos de ver qué sucede con las instancias patológicas que hallamos.

Para esta sección de la experimentación es necesario crear instancias artificiales ya que no poseemos ningún dataset con las características que buscamos. Para ello, generamos 90 grafos de cada familia. Para la familia que rompe las tres heurísticas (grafos que tienen un peso alto en la arista que cierra el circuito hamiltoniano cuyo costo es mínimo si solo se tienen en cuenta las aristas que no cierran el circuito), todos los grafos artificiales son completos y tienen 20 nodos. Elegimos un circuito entre nodos que comience en 1, que en este caso es $C = [1, 2, ..., 20]$. Asignamos a todas las aristas que no pertenecen a C peso 5, a las aristas del camino distintas a $(20, 1)$ les asignamos peso 1 y a $(20, 1)$ le asignamos peso $10 \leq i < 100$, es decir que en todos los grafos tiene un peso diferente. Por otro lado, para generar las instancias de la familia que no se comporta bien con AGM, generamos grafos que varíen en cantidad de nodos de 10 a 100. Los pesos de sus aristas son todos aleatorios de 20 a 40, exceptuando las aristas que se conectan al primer nodo, cuyo peso es 1.

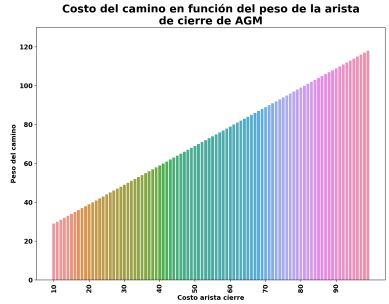
Comenzaremos analizando la familia de grafos que logra obtener resultados tan malos como queramos si aplicamos cualquiera de las heurísticas. Nos gustaría ver que a medida que vamos aumentando el peso de la arista que cierra el circuito, el costo del camino hamiltoniano arrojado por estos algoritmos crece en función de ese peso. Contrariamente, queremos ver que aplicando a estas mismas instancias las metaheurísiticas, los resultados obtenidos no varían, ya que esperamos que la exploración que realizan les permita sortear los caminos elegidos por las heurísticas.



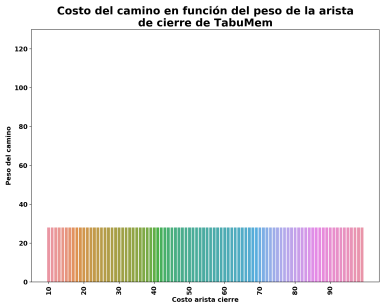
(a) Peso del camino hamiltoniano obtenido por la heurística VMC para grafos de la familia analizada.



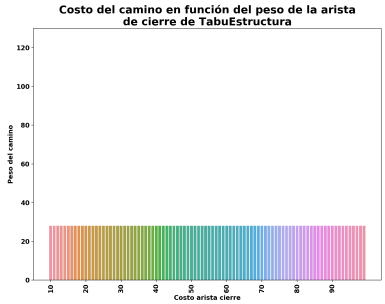
(b) Peso del camino hamiltoniano obtenido por la heurística MenorCosto para grafos de la familia analizada.



(c) Peso del camino hamiltoniano obtenido por la heurística AGM para grafos de la familia analizada.



(d) Peso del camino hamiltoniano obtenido por Tabú Search con memoria por soluciones para grafos de la familia.



(e) Peso del camino hamiltoniano obtenido por Tabú Search con memoria por estructura para grafos de la familia.

Figura 12: Pesos de los caminos hamiltonianos obtenidos en función del peso de la arista $(20, 1)$ por algoritmo.

Como se puede ver en los barplots de la figura 12, los costos de las soluciones de las heurísticas crecen de manera lineal en función del peso de la arista que cierra el circuito, mientras que para las metaheurísticas, este comportamiento no ocurre, ya que se mantienen constantes, es decir que siempre encuentran un camino que no tiene en cuenta a la arista conflictiva. Esto se condice con nuestra hipótesis y nos muestra que realmente podemos hacer que las heurísticas funcionen tan mal como queramos, basta con establecer un peso arbitrariamente grande para la arista que cierra el circuito de estos grafos para obtener caminos de costos extremadamente altos.

Procederemos a analizar ahora a la segunda familia de instancias. En este caso, vamos a analizar cuál es la distribución de los pesos obtenidos por cada algoritmo para todos los grafos generados. Queremos ver que AGM encuentra soluciones considerablemente peores que los demás algoritmos, que no deberían verse particularmente afectados al trabajar sobre esta familia de grafos.

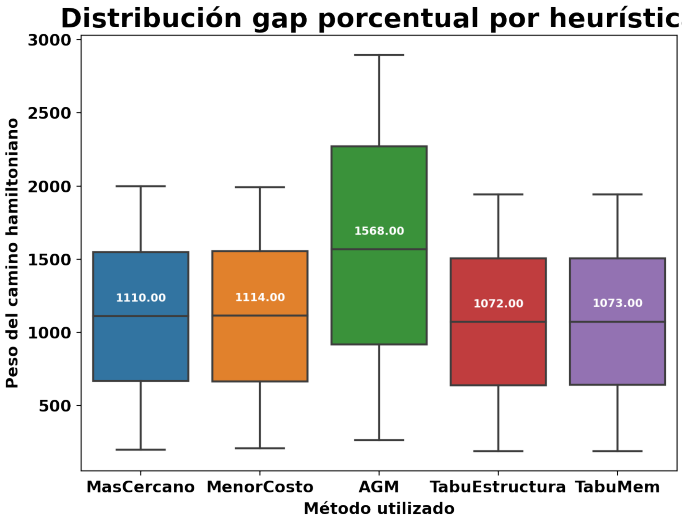
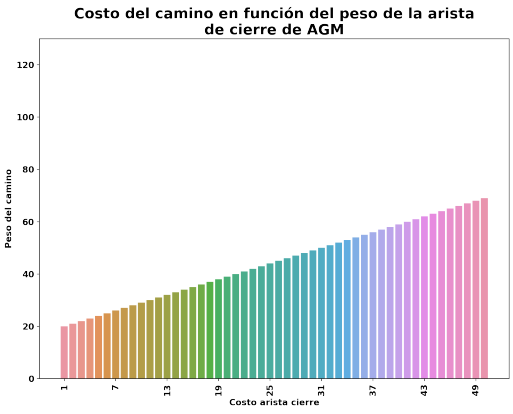


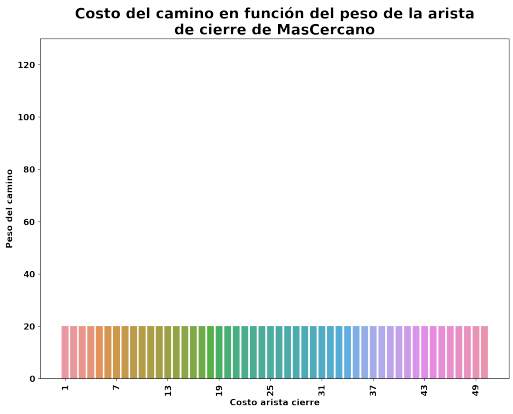
Figura 13: Pesos de los caminos hamiltonianos obtenidos por los algoritmos.

La figura 13 nos muestra que efectivamente la heurística AGM resulta ser la peor entre todos los algoritmos analizados. Sin embargo, dado a los resultados inconclusos que obtuvimos en la sección 6.4, no podemos determinar con seguridad si esta mala performance se debe al efecto que causa esta familia de instancias sobre la heurística AGM o si se trata simplemente de que este algoritmo nos provee de soluciones lejanas al óptimo en todos los casos.

A continuación se muestran los resultados de ejecutar también las heurísticas sobre una instancia como la descripta en la sección 4 (la instancia que creemos que funcionará peor en AGM que en vecinos más cercanos), con 20 vértices y aumentando el peso de la arista entre los vértices 1 y 2:



(a) Costo de la solución encontrada con la heurística AGM.



(b) Costo de la solución encontrada con la heurística VMC.

Figura 14: Comparación de heurísticas para un grafo patológico para distintos valores de n.

Podemos ver en la figura 14 que, como suponíamos, el costo del camino encontrado por la heurística AGM va creciendo de manera lineal con el peso de la arista. De esta forma logramos comprobar nuestra hipótesis acerca del comportamiento de la heurística AGM cuando es aplicada a grafos de esta familia.

7. Conclusiones

La experimentación llevada a cabo en el presente trabajo nos permitió explorar una familia de algoritmos para atacar el problema de TSP (o, en general, problemas de optimización NP-completos) que cumplen las siguientes características:

- Nos provee de soluciones cercanas a la óptima.
- Son relativamente fáciles de programar.
- Su complejidad teórica es polinomial.

Todas las cuales son características altamente deseables al tratar con este tipo de problemas. No obstante, vimos que la elección de los parámetros óptimos para una búsqueda tabú (junto con su forma de memorización) resulta una tarea no-trivial. Otro factor a tener en cuenta, es que los métodos de comparación utilizados en este trabajo (gap relativo entre el óptimo y la solución obtenida) no generalizan a instancias donde no se conoce el resultado óptimo de ante mano. Esto nos obliga a trabajar en un entorno donde constantemente se baraja cierto nivel de incertidumbre, lo que complejiza aún más la situación.

También pudimos apreciar que la elección de los parámetros, criterios de parada y criterios de memorización óptimos pueden variar en función de la aplicación sobre la cual se esté trabajando. A la hora de implementar algoritmos de estas características en la práctica, debe ser tomada en cuenta la relación entre el costo computacional y la calidad de la solución, o también entre la cantidad de memoria requerida por el algoritmo y la calidad de la solución, etc. Ninguna de estas decisiones tiene una respuesta obvia *a priori*, y deben ser analizadas dependiendo del dominio del problema que se esté atacando.

Es importante destacar que en nuestro análisis le dimos poca importancia a la complejidad teórica de dichos algoritmos, esto es así porque dada la magnitud del problema que se está tratando, prácticamente cualquier algoritmo de complejidad polinomial nos resulta satisfactorio. Si bien los algoritmos (o distintas parametrizaciones del mismo algoritmo) pueden ser más o menos

eficientes, estas diferencias resultan poco significativas comparadas con el costo prohibitivo de obtener una solución exacta en el caso general mediante el uso de algoritmos basados en backtracking o programación dinámica.

Por último, vale destacar que en este trabajo se experimentó sobre un subconjunto del universo de posibilidades ofrecidas por la meta-heurística Tabú Search, dado que al tratarse de un esquema tan general hay una amplia cantidad de variaciones que podrían implementarse sobre la misma.