DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

TP 2: Heurísticas aplicadas al TSP

27 de enero de 2021 — Algoritmos y estructuras de datos III

Integrante	LU	Correo electrónico
Chami, Uriel Alberto	157/17	uriel.chami@gmail.com
Curti, Felipe Mateo	71/17	fmcurti@gmail.com
Mamani Aleman, J. Martín	630/17	mr.tinchazo@gmail.com
Terrén Alonso, Pablo G.	271/08	pablo.scrp@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 el presente trabajo práctico abordaremos un problema conocido en el mundo de la computación: el problema del viajante de comercio (TSP de aquí en adelante, por sus siglas en inglés). Este es un problema clásico que pertenece a una categoría de problemas llamados de optimización combinatoria, en los cuales tenemos un conjunto de soluciones factibles, y buscamos encontrar la mejor de entre todas ellas de acuerdo con un criterio o *función objetivo* determinada.

La dificultad en los problemas de optimización combinatoria radica generalmente en el hecho de que no conocemos un algoritmo eficiente para hallar la mejor solución, y por lo tanto debemos construirlas todas y luego verificarlas. Por otro lado, al tratarse de un problema combinatorio, el número total de posibles soluciones crece muy rápido con el tamaño de la instancia a considerar.

En particular trabajaremos con el TSP simétrico, para el cual tenemos un conjunto de lugares a visitar y un "costo" asociado a moverse de un lugar a otro. El problema consiste en hallar el recorrido cerrado más barato que pase por todos los lugares exactamente una vez (retornando al punto de inicio). Consideramos el caso simétrico, donde el costo de moverse de un lugar A a un lugar B es el mismo que el de moverse desde B hasta A. Este problema se modela mediante un grafo pesado completo, donde los nodos representan cada uno de los lugares que debemos visitar y los pesos de las aristas los costos. El recorrido será un camino Hamiltoniano H sobre el grafo y la función objetivo, que llamamos simplemente *costo*, estará dada por $costo(H) = \sum_{c \in H} peso(c)$, donde c representa una arista particular.

Para ejemplificar el problema, veamos la figura 1 en la que tenemos un grafo de 4 vértices y 6 aristas. Al ser completo, podemos ir desde cualquier nodo hasta cualquier otro del grafo. Esto significa que podemos pensar el problema como encontrar un orden de visita de los nodos ya que todas las conexiones son posibles. Vemos así que la solución óptima comenzando por el nodo 1 es la dada por el recorrido [1, 2, 4, 3], que tiene costo 67. Cualquier otra solución posee un costo mayor, pero observemos que podría haber varias soluciones óptimas con el mismo costo.

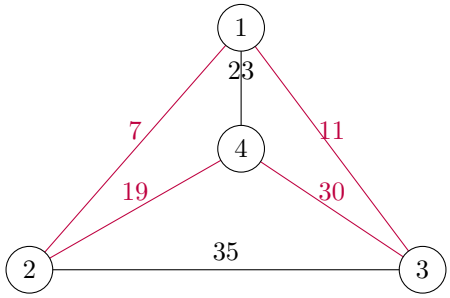

Figura 1: Ejemplo de instancia para el TSP.
En violeta, la solución óptima.

1.1. El TSP en la vida real

El estudio del problema del TSP tiene una gran relevancia práctica, dado que modela bien situaciones de la vida real. Por ejemplo, una empresa encargada de recolectar los residuos de la Ciudad de Buenos Aires posee un camión de basura por comuna. Este debe recolectar la basura depositada en los distintos contenedores repartidos a lo largo de la comuna y luego volver a la central. En este problema, no nos resulta de interés el tiempo total de recorrido sino que buscamos minimizar el combustible consumido. Este escenario modelado con grafos nos da uno cuyos vértices son los contenedores de basura de una comuna y están todos conectados (el grafo es completo). El costo c_{ij} de una arista (i, j) será la distancia entre los nodos i y j, y queremos minimizar la suma de estos costos. En definitiva, estamos buscando resolver el problema del TSP para un camión de basura que recorre distintas ubicaciones de una comuna.

Otro ejemplo muy actual que puede modelarse con el TSP: el operativo DetectAR realiza una búsqueda activa de personas sospechosas de tener Coronavirus. El objetivo es visitar los hogares de personas que tuvieron contacto estrecho con alguien que haya dado positivo o tenga síntomas compatibles con Coronavirus. De este modo, podemos crear un grafo en el que los nodos son los hogares de los distintos ciudadanos a los que apunta el operativo. Como la ciudad está interconectada, podemos llegar de una dirección a cualquier otra sin problemas. Esto provoca que el grafo sea completo. El equipo de salud quiere recorrer todos los hogares sospechosos, pero es vital hacerlo en el menor tiempo posible ya que estos trabajadores de la salud poseen otras obligaciones. De esta manera, siendo el costo de una arista el tiempo que toma trasladarse de un hogar a otro, estamos

buscando el circuito hamiltoniano de menor costo. Es decir, tenemos el problema del viajante de comercio versión cuarentena.

Sin embargo, en la gran mayoría de estos escenarios, resulta imposible escribir un algoritmo que busque la solución óptima porque el problema del TSP es *NP-hard*. Por eso, computaremos soluciones mediante heurísticas. Los algoritmos que son de tipo heurístico brindan soluciones que no necesariamente son óptimas. Sin embargo, nos garantizan que brindan soluciones en costos polinomiales, de modo que son "buenas" soluciones. De esta manera, evitamos complejidades temporales impracticables. Como veremos más adelante, hay varias formas de implementar heurísticas. En particular, implementaremos 5 técnicas que serán explicadas en detalle.

1.2. Información general

En este trabajo práctico, tomamos la decisión de representar un grafo con una matriz de adyacencias. Es por esto que el grafo g de n vértices será una matriz g $\in \mathbb{R}^{nxn}$, donde el valor de $g[i][j]$ tendrá el peso de la arista que conecta al nodo i con el nodo j. Si dos nodos no son adyacentes, entonces su arista se representará con peso -1 dado que aquí los pesos son distancias y este valor no es fáctico. De todas maneras, esto no ocurrirá en el problema del TSP ya que todos los grafos recibidos son completos por precondición.

Además, elegimos que las soluciones brindadas por las distintas heurísticas sean del tipo de datos Hamiltoniano, que es simplemente un vector de enteros. De esta forma, el valor guardado en la primera posición es el nodo inicial del circuito y luego el circuito continúa por el nodo que se encuentra en la segunda posición del vector. Esto se repite hasta el nodo que se encuentra en el último índice $(n-1)$, que se conecta con el nodo de la primera posición del vector.

Para ordenar las distintas implementaciones, juntamos a las funciones en distintos archivos de extensión .cpp:

- grafo: aquí se encuentran utilidades generales para un grafo, como por ejemplo leer uno, verificar si dos grafos son iguales, conectar o desconectar dos nodos de un grafo, hallar el árbol generador mínimo y realizar una exploración Depth First Search.
- hamiltoniano: en este archivo tenemos funciones pertinentes al tipo de datos *Hamiltoniano*, como calcular el costo de un circuito hamiltoniano y ver si dos hamiltonianos son iguales.
- heuristicas: las funciones más importantes del trabajo práctico se encuentran aquí. En particular, tenemos las 5 implementaciones de las heurísticas y sus respectivas funciones auxiliares necesarias para poder procesar una solución.
- tp: aquí está la función *main*, que contiene todo lo necesario para procesar los parámetros de configuración del programa, leer el grafo recibido por stdin y calcular el tiempo de ejecución con ayuda de la biblioteca *chrono*.

El trabajo práctico consta de 6 secciones, donde en las secciones 2, 3 y 4 desarrollamos todo lo pertinente a las heuristicas golosas, basada en AGM y metaheurísticas tabú, respectivamente. En cada una de ellas, explicamos detalladamente la idea del algoritmo, las variables utilizadas, su desarrollo, la complejidad temporal y ejemplos de instancias que muestran que las heurísticas no siempre devuelven la solución óptima. Luego, la sección 5 corresponde a los experimentos computacionales que realizamos para evaluar nuestros algoritmos. Finalmente, en la sección 6 sacaremos nuestras conclusiones de este trabajo.

2. Heurísticas constructivas golosas

2.1. Del vecino más cercano

2.1.1. El algoritmo

La forma de proceder en esta heurística es, dado un nodo inicial que será agregado a la solución al comienzo del algoritmo, ir agregando un nuevo nodo en cada paso con un criterio intuitivo: tomamos el nodo adyacente más cercano al último agregado a la solución. Es decir, si el nodo v_i fue el último agregado, debemos elegir v_{i+1} tal que la arista (v_i, v_{i+1}) minimice el costo agregado al circuito solución. Esto último es lo que le brinda la naturaleza golosa al algoritmo dado que en cada paso elegimos el mejor vecino posible y una vez que lo agregamos no deshacemos esta acción.

Esta idea se ve reflejada en la función *heuristicaVecinoMasCercano*, que recibe por parámetro el grafo en cuestión y el nodo inicial a partir del cual exploramos el resto de los nodos. En nuestra implementación, la solución será *circuitoHamiltoniano*, una variable de tipo Hamiltoniano. También contaremos con un vector de

n booleanos al cual llamaremos *visitados*. Este sirve para saber qué nodos ya han sido visitados y agregados a la solución con el fin de no pasar dos veces por un mismo vértice. La i-ésima posición de *visitados* vale *true* si y solo si el i-ésimo nodo (según el orden dado en la matriz de adyacencia de g) está en *circuitoHamiltoniano*. Además usaremos la variable *actual*, que sirve para distinguir qué nodo fue el último agregado a la solución y por ende cuál estamos viendo para explorar sus vecinos.

Al comienzo, agregamos *nodoInicial* a la solución, incializamos *actual* con el valor del nodo inicial y marcamos que este nodo ya ha sido visitado.

Acto seguido, tenemos un ciclo que se ejecutará hasta que todos los nodos hayan sido visitados. Para verificar esta condición, definimos la función *todosVisitados* en grafo.cpp. Esta recibe el vector *visitados* y devuelve true si y solo si todas las posiciones de este valen true.

En cada iteración, obtenemos el vecino más cercano al nodo *actual* y lo guardamos en la variable *elMasCercano*, lo agregamos a la solución, actualizamos el valor de *actual* asignándole el valor de *elMasCercano* y marcamos que este nodo ya ha sido visitado. Una vez que ya han sido visitados todos los nodos, devolvemos la solución *circuitoHamiltoniano*.

Para saber cuál es el vecino más cercano al nodo *actual*, definimos en heuristicas.cpp la función *obtenerElVecinoNoVisitadoMasCercano*. Esta consiste en recorrer los vecinos de *actual* (es decir, recorrer la fila g[actual] de la matriz g) y en la i-ésima iteración actualizar el valor de la variable *elMasCercano* si el i-ésimo nodo aún no ha sido visitado y la arista g[actual][i] pesa menos que g[actual][elMasCercano]. Luego de recorrer toda la fila, devolvemos *elMasCercano*.

2.1.2. Complejidad temporal

Los costos de inicializar los vectores *circuitoHamiltoniano* y *visitados* son $O(n)$. Agregar el nodo inicial a la solución, marcarlo como visitado y actualizar el valor de *actual* son operaciones de tiempo constante. El ciclo se ejecuta n veces ya que en cada iteración se agrega exactamente un nodo a la solución. Aquí, como tanto verificar la guarda como obtener el vecino más cercano cuestan $O(n)$ (el resto de las operaciones son asignaciones y una inserción en un vector), estaremos pagando este costo n veces. Es por esto que la solución dada por la heurística del vecino más cercano se calcula en $O(n^2)$.

2.1.3. Propiedades de las instancias para las cuales el método es efectivo y para las cuales no lo es

Este algoritmo puede dar resultados tan lejanos del óptimo como uno desee: en la práctica lo que podría suceder (en un espacio euclideano) es que el algoritmo empiece en el nodo 1, luego se aleje más y más de dicho nodo y cuando ya recorrió todo el grafo con un camino muy eficiente y de bajo costo, la vuelta del último nodo al nodo 1 sea extremedamente costosa.

En el marco teórico esto puede volverse aún más pronunciado cuando generamos un grafo con la intención de obtener el peor caso de esta heurística. Si quitamos la restricción de que nuestro espacio sea euclideano las posibilidades son aún más grandes. Simplemente nos proponemos un costo mínimo λ y se plantea el grafo completo que se puede observar en la figura 2.

Se puede ver que empezando en el nodo 1, la heurística del vecino más cercano se moverá hacia el nodo 2, siguiendo al 3, luego al 4 y por último cuando ya ha recorrido todos los nodos deberá volver al nodo 1 pasando por la única arista que no puede discriminar, allí asociamos nuestro costo λ.

Luego podemos ver que el costo total del circuito hamiltoniano propuesto por esta heurística es mayor a λ ya que además de pasar por la arista (4,1) de costo λ debe recorrer el camino hasta el nodo 4. Notemos que λ es completamente arbitrario y por lo tanto el costo también lo es.
Esta familia de heurísticas no acotadas se denominan *no aproximadas* y nos referiremos a ellas de esta manera más adelante.

Por último vale aclarar que aunque el planteado sea el peor caso, es poco común encontrarse con situaciones tan extremas, y además este estudio no permite medir cuán mal se va a comportar nuestra heurística para cierta instancia más homogenea. Mediante un enfoque probabilistico, asumiendo cierta homogeneidad, podemos mirar esta familia de casos en donde obtenemos resultados muy lejanos al óptimo analizando la *dispersión* de los pesos de las aristas. Este es un concepto que trabajaremos más en detalle en la sección 5, nos referimos a *dispersión* como la cantidad de pesos de distinto valor dividido por la cantidad total de aristas. Por ejemplo en el grafo de la figura 2 hay 3 valores distintos, 3, λ y 20 y la cantidad de aristas es 6. La dispersión entonces será $\frac{3}{6} = 0{,}5$. Esta medida nos da un indicador de cuánto pueden variar los valores, si la dispesión es muy baja (cercana a 0), hay muchos valores iguales lo que significa que el óptimo es más obvio de encontrar ya que se eliminan

ciertas combinaciones, por el contrario si la dispersión es muy cercana a 1, la heurística de *vecinoMasCercano* probablemenete provea resultados poco satisfactorios.

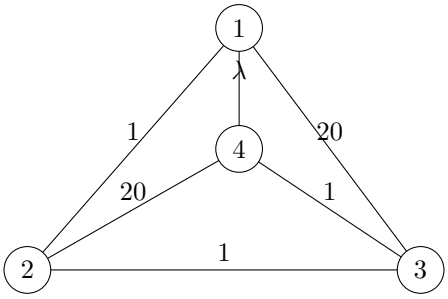

Figura 2: Posbile forma de generar un peor caso para la heurística del vecino más cercano

2.2. De inserción

2.2.1. El algoritmo

En esta heurística, la idea es formar un ciclo hamiltoniano inicial con 3 vértices cualesquiera para luego ir agregando un vértice al ciclo en cada paso. De esta forma, empezaremos con 3 nodos agregados a la solución y luego en cada paso elegir un nodo e insertarlo. Este algoritmo es goloso porque para elegir el nodo a insertar, tomaremos el más cercano a un nodo que ya está en el circuito; además, para insertarlo, buscaremos dos nodos consecutivos i e $i+1$ que minimicen el costo resultante de dicha inserción. Notemos que en todos los pasos tenemos un ciclo que se va extendiendo, a diferencia de la heurística del vecino más cercano en la cual se formaba un ciclo recién en el último paso.

La función que implementa esta idea es *heuristicaDeInsercion*, que simplemente recibe el grafo g a procesar. Como en la heurística anterior, inicialmente definimos un vector de booleanos llamado *visitados* que registra qué nodos ya han sido visitados (se inicializan todas sus posiciones en false) y también definimos la variable que representa la solución: *insertados*, de tipo Hamiltoniano.

Con el objetivo de armar un ciclo inicial cualquiera, tomamos los primeros 3 nodos según el orden dado por la matriz g, los marcamos como visitados y los agregamos a la solución, es decir, los "pusheamos" al vector *insertados*.

El grueso de este algoritmo se encuentra en el ciclo while, aunque su cuerpo simplemente consiste en elegir un nodo, marcarlo como visitado e insertarlo en la solución. Esto lo repetimos hasta que todos los nodos hayan sido visitados y lo verificamos utilizando, como en la heurística anterior, la función *todosVisitados*.

A la hora de elegir el nodo a insertar, definimos la función auxiliar *elegirNodo*, que se encuentra en este archivo. En esta función, queremos devolver el vértice más cercano a un vértice que ya está en el circuito. Este vértice lo guardaremos en la variable *elegido*. Además, tenemos las variables *distanciaAlElegido* y *distanciaAlActual* para poder comparar distancias cómodamente, y la variable *masCercanoAlActual*. En la i-ésima iteración del ciclo, que recorre todos los nodos insertados hasta el momento, tomamos el vecino que aún no haya sido visitado y que esté a menor distancia del i-ésimo nodo de *insertados*. Para esto, reutilizamos la función *obtenerElVecinoNoVisitadoMasCercano* y su valor lo guardamos en *masCercanoAlActual*. Luego de esto, comparamos *distanciaAlActual* (que no es más que g[i][masCercanoAlActual]) con *distanciaAlElegido*, que inicialmente valía infinito. Si la distancia al nodo mirado actualmente es menor que la distancia al nodo elegido hasta el momento, actualizamos el valor de *elegido*, asignándole el valor de *masCercanoAlActual* y también actualizamos el valor de *distanciaAlElegido*.

Por otra parte, para insertar el nodo elegido previamente, tenemos la función *insertarNodo*. Esta función elige dos nodos, llamados *izquierda* y *derecha* en el algoritmo, entre los cuales insertaremos el nodo *elegido*. También tendremos las variables *costoResultante* y *costoActual* para poder comparar los costos de inserción y así quedarnos con el par de nodos que minimice esta operación. En el ciclo recorremos todos los nodos en *insertados*, y en la i-ésima iteración vemos el costo de insertar *elegido* entre los nodos i e $i+1$ de *insertados*. Es decir, a *costoActual* le asignamos el peso de insertar una arista entre i y *elegido* y otra entre *elegido* e $i+1$ pero quitando el peso de la arista entre i e $i+1$. Si este costo calculado es menor que *costoResultante*, que inicialmente vale infinito, entonces actualizamos el valor de *costoResultante* y también actualizamos *izquierda* y *derecha*, que ahora pasan a valer i e $i+1$ respectivamente. Este mismo procedimiento lo realizamos una última vez pero fuera del ciclo para calcular el costo de insertar *elegido* entre el último y el primer nodo de *insertados*, es decir, viendo las posiciones $n-1$ y 0 del vector *insertados*. Una vez que sabemos entre qué nodos insertar al elegido, efectuamos esta operación mediante la función auxiliar *insertarElementoEnPosicion*. Esta se encarga

de preservar el vector *insertados* correctamente: "pushea" el nodo elegido y luego lo corre hacia su posición correcta, realizando permutaciones entre los nodos que se encuentran en las posiciones i e $i-1$ en *insertados*.

2.2.2. Complejidad temporal

En principio, inicializamos las variables utilizadas en este algoritmo en $O(n)$. Luego de esto, crear el ciclo hamiltoniano inicial (agregar 3 nodos a la solución y marcarlos como visitados) consta de 6 operaciones de tiempo constante. Acto seguido, tenemos un ciclo que se ejecuta hasta que estén todos los nodos visitados. En cada iteración agregamos un solo nodo, por lo que todos estarán visitados en n iteraciones. Como mencionamos en la subsección 2.2.1, cada iteración consiste en llamar a las funciones *elegirNodo* y *insertarNodo*, además de marcar el nodo elegido como visitado en tiempo constante. Debemos ver el costo de estas dos llamadas.

Por su parte, *elegirNodo* consiste en un ciclo que recorre el vector *insertados*, que en el peor caso tendrá tamaño n. En su cuerpo, realizamos asignaciones y una comparación, que son operaciones constantes, pero también llamamos a *obtenerElVecinoNoVisitadoMasCercano*, que es lineal. Es por esto que *elegirNodo* es una función que cuesta $O(n^2)$.

Por otro lado, *insertarNodo* es una función en la que definimos 4 variables de tipo int, claramente en tiempo constante, y luego tenemos un ciclo que también recorre *insertados*. Como en cada iteración se realizan asignaciones, comparaciones de enteros y accesos a memoria de una matriz, esto se realiza en $O(1)$ y se ejecuta a lo sumo n veces. Es por esto que el ciclo que recorre *insertados* demora $O(n)$. A continuación del ciclo, tenemos una vez más las operaciones de una iteración del ciclo, por lo que también se ejecutan en tiempo constante. La función finaliza llamando a *insertarElementoEnPosicion*, que contiene un ciclo que en el peor caso ejecuta n iteraciones. Tenemos entonces que *insertarNodo* demora dos veces $O(n)$, es decir, $O(n)$.

Volviendo al ciclo dentro de *heuristicaDeInsercion*, este se ejecuta n veces, y cada iteración cuesta $O(n^2) + O(n) = O(n^2)$. Finalmente, la heurística posee una complejidad cúbica. La función *heuristicaDeInsercion* demora $O(n^3)$.

2.2.3. Propiedades de las instancias para las cuales el método es efectivo y para las cuales no lo es

En este caso, al igual que para la heurística anterior las instancias podrían estar tan lejos como uno desee del óptimo, lo cuál nuevamente no nos dice mucho para poder comparar entre heurísitcas.
Una observación valiosa es que al buscar el nodo "más cercano" al circuito (llamemoslo w) se busca el nodo más cercano a algún v_i del circuito, luego para poder agregarlo se hace otra búsqueda minimizando $c_{v_j,w} + c_{v_{j+1},w} - c_{v_j,v_{j+1}}$, este costo resultante no necesariamente está relacionado con la distancia a la que estaba w del v_i. Para grafos donde vale la desigualdad triangular esta técnica se comportará mejor debido a lo recién mencionado, ya que si existe una arista (w, v_i), ese nodo está realmente "cerca" del circuito y cualquier nodo v_j cumplirá $c_{v_j,w} \leq c_{v_i,v_j} + c_{v_i,w}$.

3. Heurística basada en AGM

3.1. El algoritmo

Este método se basa en dos grandes pasos: calcular el árbol generador mínimo (AGM de aquí en adelante) del grafo recibido y luego recorrer dicho AGM por Depth First Search (DFS). El hecho de calcular el AGM nos permite quedarnos con las aristas que en total acumulen el menor costo posible en un árbol, mientras que el hecho de recorrerlo mediante DFS nos permite conectar dos descendientes de un nodo v_i pero sin volver a pasar por v_i ni ningún nodo ya visitado. Esto garantiza formar un circuito hamiltoniano. Sin embargo, para que dichas conexiones dadas por DFS sean posibles y no empeoren el costo total de la solución, necesitamos que valga una propiedad: el grafo $G = (V, X)$ recibido por parámetro debe ser euclideano. Esto significa que, siendo c: $X \to \mathbb{N}$ la función de costo de una arista, debe cumplirse que c((i,k)) $\leq$ c((i,j)) + c((j,k)) para todo i, j, k $\in V$.

La función que implementa esta heurística es *heuristicaAGM*. Aquí simplemente definimos t, que es un grafo que representa el AGM del grafo g empezando por el nodo 0, y luego definimos *ordDFS*, que será el circuito hamiltoniano resultante de aplicarle DFS a t. Es decir, la solución computada por *heuristicaAGM* es *ordDFS*.

Cabe destacar que la función *AGM* se encuentra en grafo.cpp y es la implementación del algoritmo de Prim. Este código está fuertemente basado en lo visto en la clase 05 "Representaciones de grafos y árbol generador mínimo" del laboratorio. *AGM* comienza marcando al nodo inicial (el número 0 según el orden dado por la matriz g) como visitado y luego guardando las distancias de los demás nodos al inicial y marcando al nodo inicial como padre del resto de los nodos. Después de esto, mientras no hayan sido visitados todos los nodos

de g, tomamos el nodo v más cercano a alguno de los que ya están en la solución, lo marcamos como visitado, actualizamos las distancias de los nodos v_i si es que ahora con v agregado es menor la distancia a estos nodos v_i y si corresponde, marcamos a v como padre de estos v_i.

Además, *DFS* también está implementada en grafo.cpp y consiste en definir un vector de booleanos llamado *visitados* y la solución *orden*. Para explorar en profundidad, utilizamos una función auxiliar recursiva llamada *DFSAux*. En esta, primero marcamos al nodo inicial como visitado y lo agregamos a la solución. Esto es posible porque los parámetros *orden* y *visitados* se pasan por referencia, permitiendo así modificar los vectores originales en lugar de copias. Luego, se llama a sí misma con el primer hijo del nodo inicial que encuentre, permitiendo así computar primero todos los hijos de una misma rama desde la raíz e ir "subiendo" solo cuando sea inevitable dado que no encuentra más hijos para explorar en profundidad esta rama.

3.2. Complejidad temporal

Dado que *AGM* es una implementación del algoritmo de Prim sin usar colas de prioridad, la complejidad de esta función es cuadrática. Por su parte, *DFS* consiste en inicializar *orden* y *visitados*, cuyo costo es $O(n)$, y luego llamar a *DFSAux*. Esta función auxiliar se ejecuta una vez por cada nodo, y en cada ejecución recorre todos los vecinos del nodo en cuestión que aún no hayan sido visitados. Es decir que no se pueden recorrer antecesores dado que estos ya habrán sido marcados como visitados. De esta forma, terminamos recorriendo una sola vez cada arista en cuestión. Por esto, *DFSAux* cuesta $O(m)$

Tenemos entonces que la complejidad temporal de *DFS* es $O(m)$. Sin embargo, como el grafo es completo, vale que $m = \frac{n.(n-1)}{2}$, por lo que *DFS* cuesta $O(n^2)$.

Dadas estas 2 complejidades, y sabiendo que se ejecutan *AGM* y *DFS* una al término de la otra, tenemos que *heuristicaAGM* cuesta $O(n^2)$.

3.3. Propiedades de las instancias para las cuales el método es efectivo y para las cuales no lo es

A priori, *heuristicaAGM* puede brindarnos resultados tan malos como sean posibles dado que la heurística es no aproximada. De este modo, es evidente que existen instancias que no conducen a una solución óptima mediante esta técnica: basta que el orden de visita dado por Depth First Search nos lleve a reemplazar 2 aristas e_i, e_j en el árbol generador mínimo por una nueva e_k perteneciente al grafo original tal que $c(e_i) + c(e_j) < c(e_k)$. Esta arista e_k podría ser lo suficientemente costosa como para que el costo de la solución de nuestra heurística sea varios órdenes de magnitud mayor que el costo de la solución óptima. En particular, si la función objetivo nos indica que λ es el costo de la solución óptima, una instancia en la cual $c(e_k) = \lambda$ ya nos permite visualizar una instancia para la cual nuestro método no es totalmente efectivo.

Sin embargo, podríamos pedir que se cumpla una hipótesis sobre el grafo a procesar: que este sea euclideano, o sea, que cumpla la desigualdad triangular mencionada en la sección 3.1. Si esto se cumple, entonces es seguro que el costo del circuito C hallado es menor que el costo del grafo E resultante de recorrer el AGM (llamémoslo T) mediante DFS. Esto se debe a que este último grafo duplica todas las aristas de T porque, por cada nodo v que visita en profundidad, luego tiene que volver al antecesor y así genera una conexión extra entre v y su padre. Esto significa que, siendo f la función objetivo, se cumple que $f(\mathrm{C}) \leq f(\mathrm{E}) \leq 2.f(\mathrm{T})$.

Al mismo tiempo, sabemos que la solución óptima C_{opt} es un circuito y que un circuito incluye un árbol. Como por definición este árbol seguro tiene costo total menor o igual que un árbol generador mínimo y justamente T es uno, entonces $f(\mathrm{T}) \leq f(\mathrm{C}_{opt})$. Juntando esta cota con la mencionada en el párrafo anterior, tenemos que $f(\mathrm{T}) \leq f(\mathrm{C}_{opt}) \leq f(\mathrm{C}) \leq 2.f(\mathrm{T})$. Es decir, tenemos garantizado que la solución C computada por la heurística basada en AGM se aleja a lo sumo $f(\mathrm{T})$ de la solución óptima.

4. Metaheurísticas Tabú

Las heurísticas presentadas anteriormente pueden darnos soluciones significativamente distantes de la solución óptima, por lo que surge naturalmente la idea de mejorar una solución inicial S. Para esta sección, implementaremos un tipo de búsqueda local: 2-opt. Esta es una búsqueda que explora soluciones "parecidas" a la inicial. A estas las llamaremos vecinos. 2-opt nos brinda una vecindad que se forma de la siguiente manera: mientras sea posible mejorar, se van formando ciclos hamiltonianos producto de tomar 2 aristas e intercambiar sus extremos.

El problema de 2-opt es que solo explora si inmediatamente está mejorando una solución actual. Esto provoca que con 2-opt a secas, nos estanquemos en óptimos locales. Si el óptimo global podía formarse primero empeorando la solución actual y luego mejorándola, 2-opt jamás podría computarla. Es por esto que existen las

metaheurísticas. Estas buscan explorar más allá de los óptimos locales. En particular, la metaheurística tabú consiste en explorar soluciones buenas pero también soluciones malas, que estarían prohibidas con el método 2-opt tradicional.

Para poder empeorar soluciones sin caer en repetir ciclos ni extender indefinidamente la búsqueda, tabú se apoya en el uso de una memoria de últimas acciones realizadas y un criterio de parada que pondrá fin a la exploración. En este trabajo práctico, implementamos 2 búsquedas tabú según cómo está implementada la memoria.

4.1. Basada en las últimas soluciones exploradas

4.1.1. Parámetros de entrada

Esta metaheurística la hemos implementado en la función *heuristicaTabuSolucionesExploradas*, que recibe los siguientes parámetros:

- g: el grafo a procesar.
- solucionInicial: función pasada por parámetro utilizada para calcular la solución inicial a partir de la cual haremos la búsqueda local. Esta función recibe un grafo y devuelve un Hamiltoniano.
- criterioDeParada: un string que representa el criterio elegido para terminar la búsqueda. Este string puede ser "cantIteraciones" o "cantIteracionesSinMejora". El primero lleva la cuenta de la cantidad total de iteraciones, hayan encontrado una mejora o no, mientras que el segundo solo cuenta la cantidad de intentos fallidos, es decir, iteraciones en las que no se encontró un circuito hamiltoniano de menor costo que el mejor actualmente.
- umbral: un valor de tipo int que representa el límite de iteraciones o de iteraciones sin mejora, según el criterio de parada pasado. De esta manera, el ciclo principal se ejecuta mientras no se supere este límite.
- tamanioMemoria: un int que indica cuál es el tamaño de la memoria utilizada por tabú search para evitar repetir soluciones.
- probabilidadDeDescarte: valor de tipo float que vale entre 0 y 1. Este número nos indica qué porcentaje de la subvecindad descartaremos. De esta forma, con una probabilidad de descarte de 0.7 por ejemplo, estaremos tomando el 30 % de la vecindad para explorar.

4.1.2. Variables declaradas en la función

El algoritmo comienza declarando y definiendo una serie de variables. La primera es *ciclo*, de tipo Hamiltoniano, que se inicializa con el valor devuelto por solucionInicial. La variable *mejor* también es de tipo Hamiltoniano, y la usaremos para guardar la mejor solución que vayamos encontrando y devolverla al final del algoritmo. Luego tenemos *memoria*, que es un vector de hamiltonianos en el que guardaremos los últimos *tamanioMemoria* ciclos visitados. También tenemos *indiceMasViejoDeLaMemoria*, que usaremos para saber en qué posición del vector de memoria corresponde guardar una nueva solución explorada. Luego incializamos *cantIteracionesSinMejora* y *cantIteraciones* en 0 y definimos *criterio*, un puntero a int. Este puntero lo usaremos para comparar lo referenciado por él con *umbral* y así evitar tener dos ciclos while: uno para ver si *cantIteracionesSinMejora* $<$ *umbral* y otro exactamente igual repitiendo código salvo que la guarda es ver si *cantIteraciones* $<$ *umbral*.

De esta forma, si nos pasaron por parámetro que *criterioDeParada* es "cantIteracionesSinMejora" entonces *criterio* apunta a *cantIteracionesSinMejora*. En caso contrario, apunta a *cantIteraciones*.

4.1.3. El ciclo *while*

Como mencionamos anteriormente, tendremos un ciclo que se ejecuta hasta que lo referenciado por *criterio* sea menor que *umbral*. En cada iteración, creamos la variable *vecinos*, que es un vector de circuitos hamiltonianos. Aquí guardaremos al subconjunto de los vecinos de *ciclo* que exploraremos, y para esto llamamos a la función *obtenerSubVecindad* que explicaremos más adelante.

Una vez que tenemos la subvecindad guardada en *vecinos*, debemos quedarnos con la mejor solución aquí dentro teniendo en cuenta que dicha solución no esté en la memoria de soluciones exploradas (existe la posibilidad de que todos los vecinos posibles estén en la memoria, en cuyo caso es devuelto un Hamiltoniano vacío). Para esto, invocamos a la función *obtenerMejorConMemoriaDeSoluciones* y lo devuelto se lo asignamos a *nuevo* una

variable de tipo Hamiltoniano. Si nuevo tiene un Hamiltoniano válido asignamos $ciclo = nuevo$. Actualizando así la solución actual que estamos teniendo en cuenta.

Ahora que tenemos esta solución explorada, la guardamos en *memoria*, justamente en el índice *indiceMasViejoDeLaMemoria*. Después de esto, incrementamos en 1 el índice correctamente y actualizamos el valor de *mejor* si es que el costo de *ciclo* es menor que la mejor solución guardada hasta el momento. Además, en cada iteración del ciclo *while* incrementamos en 1 a *cantIteraciones*; y si *ciclo* no tenía mejor costo que *mejor*, también incrementamos en 1 a *cantIteracionesSinMejora*.

4.1.4. Obtención de la subvecindad

La función *obtenerSubVecindad* recibe un Hamiltoniano *solucionParcial* y el Grafo g y utilizando la función 2*opt* obtiene la vecindad de *solucionParcial*. Además esta función discrimina ciertos vecinos y los ignora, de esta manera le permitimos a nuestra heurística que empeore y así que no se estanque en óptimos locales.

Iniciamos definiendo *vecindad*, variable en la cual guardaremos a todos los vecinos que no discriminemos de *solucionParcial*. Luego, creamos una variable de tipo bernoulli_distribution de la biblioteca std que nos permite **tirar una moneda cargada con la probabilidad recibida por parámetro** *probabilidadDeDescarte*.

Luego se inicia un ciclo que recorre i desde 0 hasta n, y un subciclo del mismo que recorre j entre $i+1$ y n. Las variables i y j van a representar las posiciones en *solucionParcial* de los nodos que vamos a intercambiar, es por esto que no tiene sentido que j pueda valer desde 0 hasta n ya que los intercambios son simétricos. En cada iteración del ciclo interno, tiramos la moneda cargada y si la respuesta es $false$ (es decir, que no descartamos), ejecutamos la funcion 2*opt* (que simplemente invierte el orden de los nodos de *solucionParcial* entre i y j). Y agregamos este vecino al vector *vecindad*.

Por ley de grandes números, después de suficientes tiradas de la moneda, deberíamos estar cerca de descartar el porcentaje indicado de los vecinos y ciertamente tiraremos la moneda muchas veces, precisamente $cantIteraciones * n^2$ veces.

4.1.5. Obtención del mejor de la subvecindad

La función *obtenerMejorConMemoriaDeSoluciones* obtiene mediante los *vecinos*, la *memoria* y el Grafo g, el vecino con menor costo que no esté presente en la memoria. Esto lo logra recorriendo cada uno de los vecinos con un j y por cada uno de ellos verificando que para ningún i valga $sonIguales(vecinos[j], memoria[i])$.
sonIguales es una función definida en hamiltoniano.cpp que recorre posición por posición ambos Hamiltonianos y verifica si son o no iguales.

Mediante este proceso se llena el vector de Hamiltonianos *vecinosFiltrados*, dicho vector podría terminar vacío, la función *obtenerMejor* a la que llamamos luego se encarga entonces de dos cosas:

- Devolver un Hamiltoniano inválido en caso de que el vector de Hamiltonianos esté vacío.
- Obtener el grafo de menor costo del vector de Hamiltonianos que recibe en caso contrario.

4.2. Basada en una estructura

La metaheurística tabú basada en estructura cambia uno de los conceptos básicos respecto de su pariente que es la información que es guardada en la memoria, recordemos que la memoria es el artilugio que evita que naveguemos la misma solución más de una vez. En este caso guardamos la información de qué aristas fueron intercambiadas.

Esto no necesariamente significa que la solución a la que se transicionaría si se hiciese dicho intercambio es necesariamente la misma que antes (como sí pasaba con la memoria de la heurística previa). Esta interpretación más libre de memoria ahorra capacidad de RAM requerida, ya que en vez de guardar un vector de vectores de enteros (Hamiltonianos), ahora guardamos un vector de pares de enteros lo cual es mucho menos costoso. También ahorra comparaciones para decidir si cierto Hamiltoniano está presente o no en la memoria.

Existen sin embargo algunas dificultades en la implementación que tienen como consecuencia que los códigos de los algoritmos estén duplicados, siendo que la lógica que siguen es exactamente la misma.

Dichas complicaciones están relacionadas con el hecho de que además de guardar cada posible vecino a la hora de generar la subVecindad, ahora debemos guardar también el par (i, j) de índices que ese vecino intercambia respecto de su *solucionParcial* asociada.
Utilizaremos la estructura pair<Hamiltoniano, pair<int,int>> para representar a un circuito hamiltoniano durante toda la implementación y llamaremos a dicho tipo de dato **HamiltonianoConInfo** (aunque en el código este renombre no exista). La implementación de lo aquí mencionado se encuentra en el archivo heuristicas.cpp en

la función *heuristicaTabuAristasIntercambiadas*. Aclaremos rápidamente diferencias y similitudes entre ambos métodos.

4.2.1. Parámetros de entrada

Los paramétros de entrada son exactamente iguales que los de la sección 4.1.1.

4.2.2. Variables declaradas en la función

Las variables definidas son equivalentes a las de la sección 4.1.2, con la excepción de que cuando se trataba con un Hamiltoniano se trata con un HamiltonianoConInfo y que la memoria ahora es un vector de pair<int,int>.

4.2.3. El ciclo *while*

Aplica lo desarrollado en la sección 4.1.3 excepto:

- Se trata con HamiltonianoConInfo en vez de con Hamiltoniano
- La función $obtenerMejorConMemoriaDeSoluciones$ pasa a llamarse $obtenerMejorConMemoriaDeAristas$.
- La función $obtenerSubVecindad$ pasa a llamarse $obtenerSubVecindadConInfoIntercambios$.

4.2.4. Obtención de la subvecindad

La función *obtenerSubVecindadConInfoIntercambios* es equivalente a lo desarrollado en la seccion 4.1.4 con la excepción de que a la hora de guardar el Hamiltoniano en *vecindad*, incluye tambien el *intercambio* ejecutado, que no es más que un pair<int,int>. Esta información nos permitirá luego saber si el vecino se encuentra o no en la memoria y también para el vecino que consideremos óptimo, poder guardarlo en la misma.

4.2.5. Obtención del mejor de la subvecindad

El desarrollo es muy similar al de la sección 4.1.5 salvando la comparación para decidir si cierta solución pertenece o no a la memoria. La validación ejecutada para verificar que el vecino no pertenezca a la memoria es $(memoria[\alpha]_1 \neq vecinos[\beta]_{2_1} \vee memoria[\alpha]_2 \neq vecinos[\beta]_{2_2})$ donde α y β son enteros que van entre 0 y el tamaño de la memoria y los vecinos respectivamente. Recordemos que vecinos es un vector de HamiltonianoConInfo, por lo tanto $vecinos[i]_2$ es el par (i, j) intercambiado. Por último recordemos que la segunda coordenada de los pares intercambiados siempre es mayor a la primera ya que el algoritmo elige los j siempre mayores a los i. Por lo tanto no es necesario comparar $memoria[i]_1$ con $vecinos[j]_{2_2}$ ni visceversa.

4.3. Propiedades de las instancias para las cuáles el método es efectivo y para las cuales no lo es

El método de tabú no dista de otra búsqueda local, es decir que nada nos asegura que las iteraciones en busca de los mejores vecinos necesariamente deban mejorar la solución inicial. Existe la posibilidad (en peor caso) de que las heurísticas tabú no puedan mejorar en nada la solución inicial que se les proveyó.

De lo que tenemos seguridad es que es imposible, por como está planteada la heurística, que se empeore la solución inicial. Ya que si no logra una mejora no recuerda a ese vecino.

Podemos decir entonces que a nivel teórico, **Tabú search será tan bueno como su solución inicial**. Por esto, si nuestro grafo es euclideano y utilizamos como algoritmo para proveer la solucion inicial la heurística basada en AGM, podemos asegurar que por lo explicado en la sección 3.3 podemos asegurar que el resultado será 1-aproximado.

5. Experimentación

Esta sección está dedicada al estudio empírico de los algoritmos implementados. Analizamos cuantitativamente las características de las rutinas mediante dos estrategias: exacta (donde el camino de costo mínimo es conocido) y estadística. En el primer caso nos interesamos por los valores finales de los caminos hallados ya que tenemos acceso a un contraste exacto con la solución óptima del problema. En el caso estadístico, nos valemos de un análisis combinatorio para estimar un valor óptimo probable, y cuantificar la calidad de la solución hallada en términos de la población esperada de caminos con ese costo en el grafo.

Los experimentos fueron realizados en una computadora con procesador Intel Core i5 (mod. 7200U) @2.50GHz con 8Gb de memoria RAM disponible. La implementación de los algoritmos se realizó en el lenguaje de programación C++. Compilado con g++ y ejecutado sobre una plataforma linux de 64bits.

5.1. Grafos utilizados en los experimentos

Para las comparaciones y testeo de rendimiento exacto de las rutinas implementadas utilizamos los grafos con soluciones conocidas que fueron dados por la cátedra. En los experimentos nos referimos a estos grafos mediante el mismo nombre con el cual están publicados en el *url* provisto (ej: *a280*, *gr21*, *gr202*, etc).

Para los estudios cualitativos requerimos un control más específico sobre las instancias del problema, por lo cual trabajamos con tres tipos de grafos con costos aleatorios sobre K_n que denominamos genéricamente *grafo aleatorio*, *grafo aleatorio euclídeo* y *grafo desconectado* a lo largo de toda esta sección.

En el caso de los grafos aleatorios, elegimos los $\frac{n(n-1)}{2}$ costos (enteros) de arista mediante una distribución uniforme definida por un intervalo $[n_i, n_f]$.

Por otro lado, los grafos aleatorios euclídeos los construímos a partir de muestrear al azar y sin repetición una grilla bidimensional de puntos (x, y) de coordenadas enteras, utilizando un cuadrado de tamaño $D \times D$ (con $D^2 > n$). Construímos el grafo asociando cada uno de los n puntos elegidos con un nodo, y calculando los costos de la aristas mediante la norma uno sobre la grilla, es decir $costo(i, j) = |x_i - x_j| + |y_i - y_j|$. Como la norma uno define una distancia, nos aseguramos que la desigualdad triangular se cumple en el grafo.

Finalmente, el grafo desconectado tiene dos tipos de costo de arista: las que llamamos *caras* (costo alto) y las *baratas* (costo bajo respecto de las anteriores). Estos grafos los construímos de manera aleatoria, pero mediante un parámetro de proporción de aristas α. Elegimos $\alpha\frac{n(n-1)}{2}$ aristas del conjunto de costo alto y las restantes del conjunto barato. Así decimos que un grafo con α cercano a 1 es caro, mientras que con α cercano a 0 será barato.

Para realizar estas operaciones empleamos el paquete *random* de *Python*.

5.2. Breve análisis estadístico de K_n

Reservamos esta subsección para introducir brevemente las cualidades estadísticas en las que estaremos interesados para el análisis. Consideramos primero el grafo K_n de aristas no pesadas. Sabemos que:

- el número de vértices es n.
- el número total de aristas es $\frac{n(n-1)}{2}$.
- la cantidad de ciclos Hamiltonianos *distintos* es $\frac{(n-1)!}{2}$. Asumiremos a partir de ahora que todos los ciclos comienzan en el mismo vértice v_1 y circulan en un sentido particular.

Consideramos ahora un subgrafo generador K_r^n de K_n construido con $(n/2).r$ aristas de K_n tomadas al azar (sin reposición). Como cada arista aumenta en 1 el grado de dos nodos de K_r^n, vale $\sum_{v \in K_r^n} gr(v) = 2.(n/2).r = n.r$. Finalmente como son tomadas al azar, esperamos que en promedio cada nodo tenga grado $r = \frac{n.r}{n}$.

Con este resultado calculamos el número esperado de caminos Hamiltonianos distintos en K_r^n. Sean $C = \{c_1, c_2, ..., c_n\}$ con $c_k = (v_k, v_{k+1})$ las aristas que determinan un ciclo hamiltoniano particular de K_n. Entonces la probabilidad de que c_1 esté en K_r^n es $p(c_1 \in G_r) = \frac{r}{n-1}$, porque asumimos que el vértice v_1 tiene r aristas de las $n-1$ que le corresponden en K_n. Luego, la probabilidad de que c_2 esté en K_r^n dado que $c_1 \in K_r^n$ resulta $p(c_2 \in K_r^n | c_1 \in K_r^n) = p(c_1 \in K_r^n)\frac{r-1}{n-2}$, puesto que c_1 ya determina una de las r aristas de v_2. Lo mismo ocurre con $c_3, c_4, ..., c_n$. Teniendo en cuenta estos valores, calculamos la probabilidad de que C esté incluído en K_r^n como

$$p(C \in K_r^n) = \frac{r}{n-1}\left(\frac{r-1}{n-2}\right)^{n-1}. \tag{1}$$

Luego, el número esperado de caminos Hamiltonianos en el subgrafo K_r^n, que llamamos $E_c(K_r^n)$ lo podremos estimar como la probabilidad de que un ciclo esté incluído, multiplicado por la cantidad total de ciclos de K_n, es decir

$$\#\{\text{ciclos Hamiltonianos en } K_r^n\} = E_c(K_r^n) = \frac{(n-1)!}{2}\frac{r}{n-1}\left(\frac{r-1}{n-2}\right)^{n-1}. \tag{2}$$

A partir de la ecuación (2) podemos hallar un r' tal que $E_c(K_{r'}^n) = 1$. Este valor determinará el mínimo número de aristas $(n.r'/2)$ que deberemos tomar de K_n para, en promedio, tener un único ciclo en un grafo de n vértices.

Sea ahora un grafo G completo con costos asociados en sus aristas. Por la cuenta anterior, sabemos que si esperamos tener un ciclo, debemos incluír $n.r'/2$ aristas de este grafo. Consideramos entonces las $n.r'/2$ aristas de menor costo, lo que nos permite inferir que habrá n de ellas que formen un ciclo. Dado que en general no podemos saber cuáles, simplemente tomamos un valor promedio entre todas ellas, y multiplicamos por n (número de aristas que conforman el ciclo). Por ejemplo, si en G las $n.r'/2$ aristas más baratas son de costo 3, la estimación dará un ciclo de tamaño $3n$. Si la mitad de ese conjunto cuestan 1 y la otra mitad 3, la estimación dará un costo de ciclo de $\frac{1.n.r'/4+3.n.r'/4}{n.r'/2}n = (0{,}5 + 1{,}5)n = 2n$.

La Fig. 3 muestra el error relativo entre la estimación del costo del ciclo mínimo y la solución al TSP para los grafos donde ésta es conocida. Se observa que hay una cantidad no despreciable de puntos aglomerados alrededor del 0, aunque también pueden verse casos *outliers* donde el error en la estimación es superior al 50 %. También se observa que en general el valor estimado es menor a la solución, con lo cual como regla conviene, aunque no exactamente, tomarlo como una cota inferior estricta.

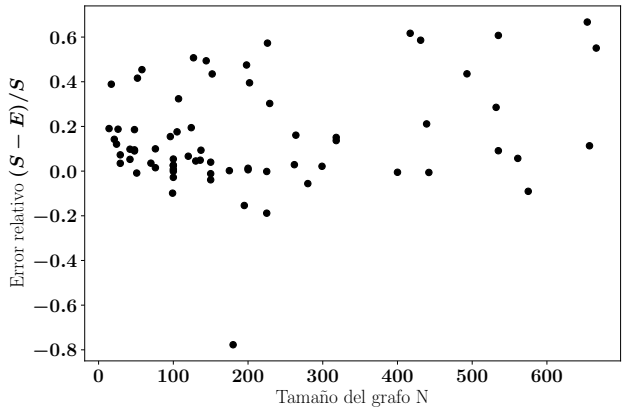

Figura 3: Error relativo de la estimación del ciclo de costo mínimo para los grafos con solución conocida.

5.3. Algoritmos golosos - complejidad temporal y comportamiento

En la Fig. 4 se observa el tiempo de cómputo empleado por los algoritmos golosos implementados como función del tamaño del grafo de entrada n en escala logarítmica. Los ajustes correspondientes muestran que la complejidad asintótica empírica se encuentra en total acuerdo con el análisis teórico.

Por su parte la Fig. 5 muestra el costo medio por arista en los ciclos hallados por los algoritmos golosos. Los grafos utilizados en este experimento son aleatorios, con pesos de aristas tomados uniformemente en el intervalo $I = [1, 500]$. Notar que el costo de ciclo más bajo posible en estos grafos es n, o en términos relativos al tamaño del grafo, el costo por arista más bajo posible es $n/n = 1$.

Es esperable que dado un costo de arista fijado por un intervalo como en este caso, los algoritmos golosos encuentren soluciones cada vez mejores (más parecidas a 1 en el caso de la Fig. 5) a medida que aumenta el tamaño del grafo. Al tener más aristas disponibles, es más probable hallar alguna de costo bajo en cada paso de la construcción de la solución y que en el futuro todavía queden aristas de costo bajo disponibles.

Agregamos en la Fig. 5, a modo de comparación, el costo hallado por el algoritmo AGM para grafos aleatorios euclideos de igual tamaño. Observamos que la calidad de los ciclos obtenidos mediante AGM es muy inferior con respecto a los otros algoritmos golosos si los grafos poseen pesos completamente aleatorios. Sin embargo podemos observar un rendimiento similar en el caso de tratarse de grafos euclídeos. Notar que la comparación no puede hacerse de manera directa pues la distribución de costos en un grafo aleatorio y en uno aleatorio euclídeo no es igual. Sin embargo es un indicio claro del tipo de grafos en los que AGM tiene relevancia a la hora de proveer una solución equiparable con los otros algoritmos golosos.

5.4. Algoritmos golosos - análisis de calidad

En las Figs. 6 y 7 pueden verse los costos de ciclo hallados por las heurísticas golosas, normalizados por el número de aristas del ciclo ($n = 100$ y $n = 30$ respectivamente) para grafos aleatorios y aleatorios euclídeos.

En el caso de la Fig. 6 analizamos grafos completamente aleatorios de tamaño 100 con costos de aristas tomados del intervalo $[1, n_f]$. Al aumentar el límite n_f se observan principalmente dos características: por un

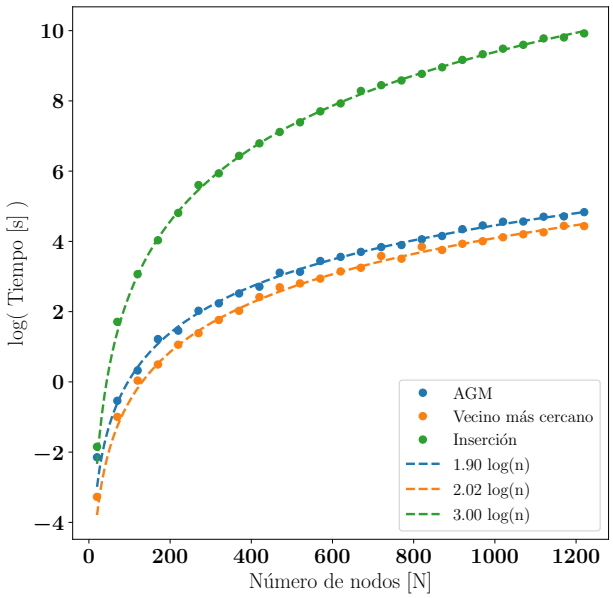

Figura 4: Complejidad temporal de los algoritmos golosos como función del tamaño del grafo

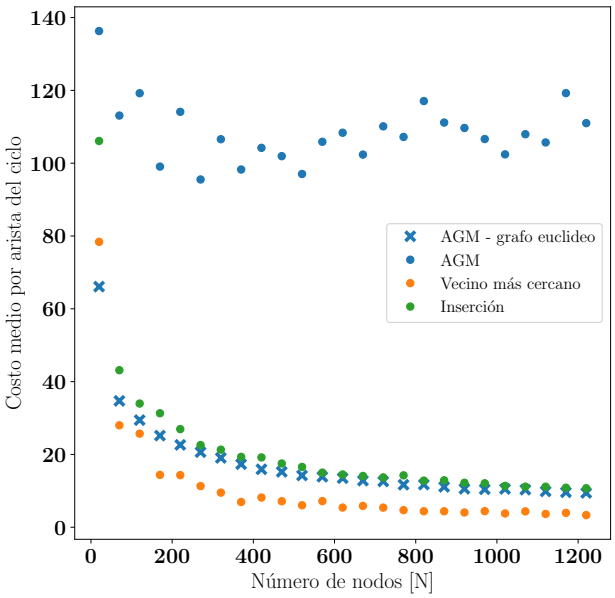

Figura 5: Costo de los ciclos hallados para los mismos grafos utilizados en la Fig. 4

lado, al incorporar aristas más pesadas al grafo es esperable que el valor óptimo del TSP aumente, esto se ve reflejado tanto en la estimación como en los resultados de las heurísticas en el crecimiento lineal de los valores.

Por otro lado, el efecto de agregar diversidad de costos de aristas al grafo hace que cada vez sea más improbable acercarse a una solución óptima mediante un criterio local. Esto se observa en el hecho de que los puntos se alejan cada vez más de los valores óptimos esperados. Una forma de entender este comportamiento es a través de un razonamiento estadístico. La variedad de costos de aristas genera que los ciclos se diferencien entre ellos (en un caso extremo, si hay sólo un tipo de costo, todos los ciclos tendrán igual valor). El efecto es el contrario al que se observaba en la Fig. 5 donde lo que se aumentaba era el número de nodos, con lo cual la población de costos de arista era cada vez menos diversa en términos de n.

En la Fig. 7 analizamos una situación análoga a la presentada en la Fig. 6 pero sobre grafos aleatorios euclídeos, donde el parámetro que crece es el ancho de la grilla de sampleo, D. El comportamiento general de las heurísticas es similar al anterior, sin embargo se observa una diferencia fundamental en el comportamiento de los valores de AGM. En este caso la desviación respecto de la solución esperada óptima es menor, pues sabemos que para un grafo que cumple la desigualdad traingular, la heurística de AGM es 2-aproximada.

En las Figs. 8 y 9 estudiamos las condiciones generales que cumplen las instancias para las cuales es esperable que las heurísticas golosas arrojen resultados alejados de los valores óptimos. Se muestran los costos de los ciclos hallados por las heurísticas de Inserción y VecinoMasCercano, normalizados al valor óptimo estimado por el modelo probabilístico presentado en la sección 5.2.

Utilizamos instancias de grafos desconectados, pues nos permiten un control selectivo de la característica que distingue a estos algoritmos: elegir siempre la opción barata local. Debido a esto, sabemos que los algoritmos intentarán construír un ciclo Hamiltoniano teniendo en cuenta solo las aristas del conjunto de costos baratos, e ignorando las del conjunto caro. Como se observa en dichas figuras, estudiamos los efectos como función del parámetro α anteriormente definido. Para este experimento se eligieron aristas baratas en el intervalo $[1, 5]$ y aristas caras en el intervalo $[100, 105]$. Estos valores generan un contraste significativo en los resultados de las heurísticas (observar la estructura de tipo escalón en las figuras). Pueden distinguirse claramente los casos donde por ejemplo sólo una, dos o tres aristas caras fueron elegidas.

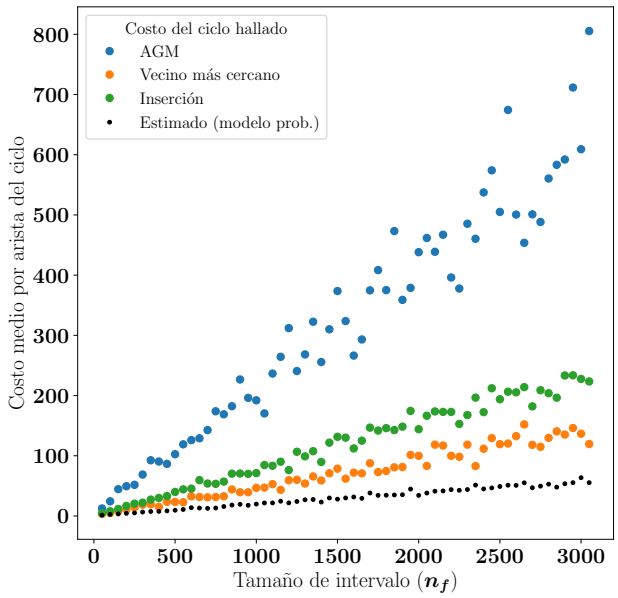

Figura 6: Costo grafo random. $n = 100$ en este ejemplo

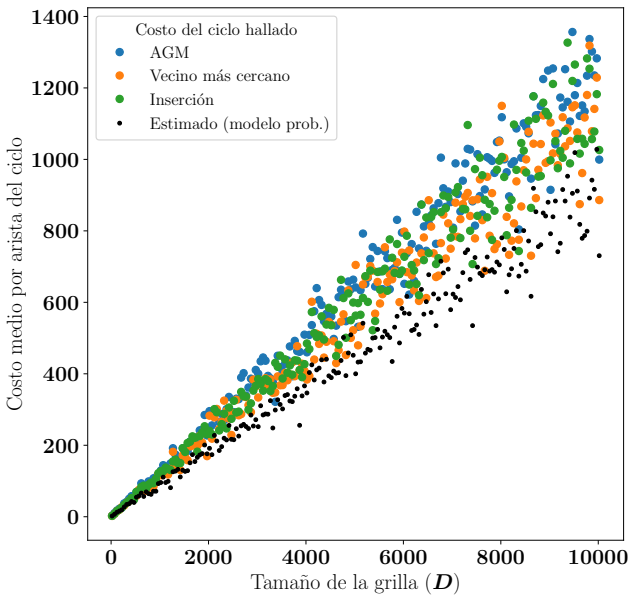

Figura 7: Costo grafo euclideo. $n = 30$ en este otro.

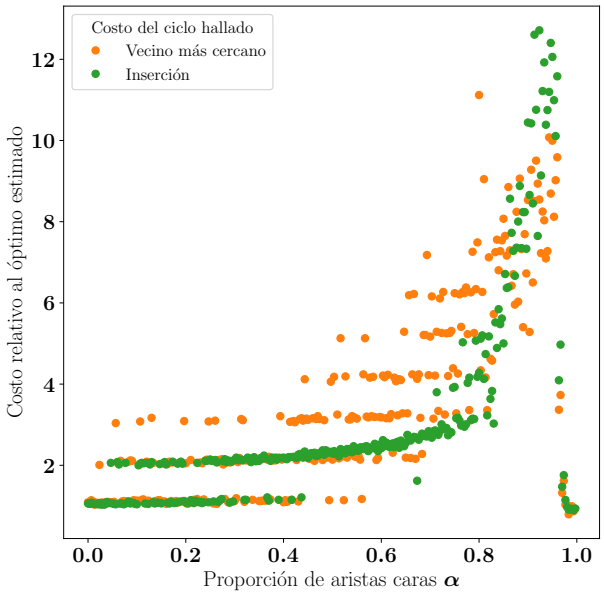

Figura 8: Costos grafo desconectado. $n = 100$

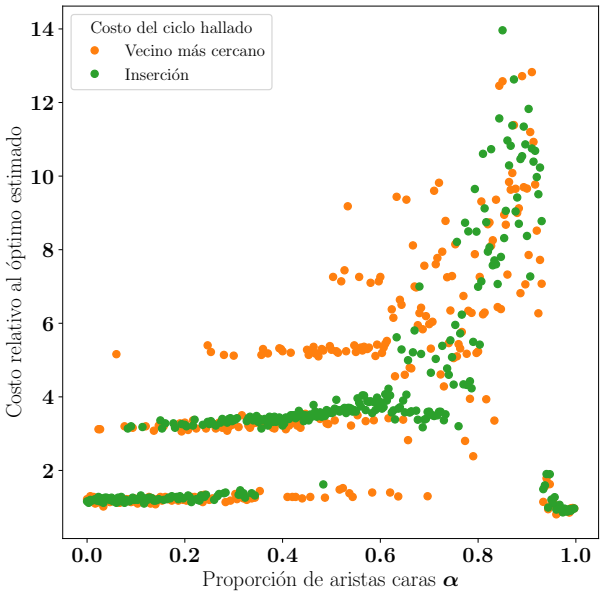

Figura 9: Costos grafo desconectado. $n = 50$

Observamos en la Fig. 8 que si existe una población grande de aristas baratas ($\alpha \sim 0$), el ciclo que encuentran las heurísticas es en general cercano al óptimo, pues es esperable que continúe habiendo disponibilidad de aristas baratas mientras se va contruyendo la solución. Por otro lado, a medida que el grafo incorpora mayor proporción de aristas caras, la elección golosa trae consecuencias en los costos posteriores; al aumentar el parámetro α cada vez es menos probable que el costo óptimo se alcance mediante este tipo de elecciones. Finalmente, cuando $\alpha \sim 1$ vemos que los resultados golosos vuelven a acercarse a los valores óptimos estimados. Esto responde simplemente al hecho de que se pierde la diferenciación de costos de aristas, y el problema se vuelve fácil como al principio.

La Fig. 9 estudia el mismo fenómeno con un número de nodos menor. Notamos que el efecto descripto por la Fig. 6 también se observa en este caso: Al reducir el número de aristas el problema se vuelve "mas difícil"porque el número de aristas baratas respecto de la cantidad de nodos se ve deteriorado (recordar que el número de aristas crece como n^2, luego duplicar los nodos implica cuadrupicar la cantidad de aristas). Como resultado de este efecto vemos que los algoritmos golosos comienzan a discrepar de la solución óptima esperada con α más pequeño y en un rango más amplio.

Aclaración: omitimos el estudio de la heurística AGM en estos experimentos por tratarse de instancias de grafos no euclideos, y por lo tanto los resultados son de una calidad considerablemente inferior, al igual que ocurre en la Fig. 6.

5.5. Tabú - Calidad de las soluciones según tipo de memoria

Estudiaremos en esta sección el comportamiento de ambas variantes de Tabú search (con memoria de arisas y de soluciones), realizamos dos experimentos en ambos utilizamos para hacer este análisis las instancias provistas por la cátedra *gr48* y *gr202* para las cuales sabemos con seguridad cuál es su valor óptimo.

5.5.1. 1^{er} experimento - variando cantidad de iteraciones

Para este experimento nuestra variable libre son la cantidad de iteraciones, podemos ver en la Figura 10 cómo la memoria de soluciones performa notoriamente mejor para la instancia *gr48* mientras que para *gr202* el comportamiento es exactamente el mismo y por eso no se puede apreciar la diferencia.

Esto puede deberse a la baja cantidad de iteraciones. Un grafo con 202 nodos tiene $\frac{202*201}{2} = 20301$ aristas, y nosotros ejecutamos 100 iteraciones, recordar que en cada iteraciones el mejor vecino/intercambio es guardado en la memoria, es decir que al cabo de 100 iteraciones solo se guardaron 100 vecinos. Recordar también que la única diferencia entre estas dos técnicas es precisamente la memoria y por lo tanto el único escenario en el que puede haber diferencia entre ellas es cuando la memoria filtra casos significativos. La probabilidad de que algun otro vecino que intercambia las mismas aristas que alguno de los guardados en la memoria sea el mejor de toda la subvecindad a la que pertenece es $\frac{100}{20301}$.

Notemos también lo mucho que mejora tabú en 40 iteraciones respecto de la solución incial, llegando prácticamente al valor óptimo con mejoras de entre 20 % y 40 % respecto de las soluciones iniciales dependiendo el tipo de memoria y la instancia particular.

Estancamiento

Debido a la naturaleza de las búsquedas locales, llegado cierto punto se estancan y no pueden mejorar más la solución con la que trabajan. Quizás porque el óptimo local en el que caen y sus subvecindades asociadas tienen cierta "gravedad"hacia el óptimo local en cuestión. Es por eso que podemos ver cómo la línea sólida azul (que representa la memoria de aristas para la instancia *gr48*) en la Figura 10 se queda en la solución que es un 15 % más pesada que el óptimo, y no importa si se le dan 40 iteraciones más, no logra escapar de dicho óptimo local. Este factor es bastante importante, y creemos por este motivo que el criterio de parada que depende de la cantidad de iteraciones sin mejora es la manera más acertada de proceder si lo que se busca es obtener la mejor solución posible en un tiempo razonable. Sin embargo si se desea poder tener una cota previa de cuánto va a tomar el procedimiento, es necesario darle un marco a esta medida, podría simplemente fijarse una cantidad de iteraciones máxima más allá de la última mejora hallada, o podría establecerse un porcentaje de mejora mínimo para considerar que lo encontrado es realmente una mejora, estos son posibles agregados que podrían hacerse a nuestras heurísticas actuales.

5.5.2. 2^{do} experimento - variando el tamaño de memoria

En la Figura 11 podemos observar los resultados comparativos de ejecutar 50 iteraciones de tabú search para memoria de aristas y de soluciones variando el tipo de memoria. Lo primero que uno observa es el comportamiento errático de la memoria de aristas con memoria chica, nuestras hipótesis sobre esa particularidad es que el comportamiento errático no depende de la memoria en sí. Algo que es bastante claro es que a partir del tamaño de memoria 50, la diferencia se vuelve nula, esto se debe a que la cantidad de iteraciones es 50, con lo cual nunca llegará a llenarse una memoria de tamaño 60, por lo que obtendrá el mismo resultado. Este resultado poco satisfactorio nos lleva a analizar cuáles podrían ser los potenciales parámetros que generan un comportamiento tan errático. Nuestra primer teoría para el caso de memoria de aristas, es que la memoria de aristas es un filtro demasiado exigente, y deja de lado demasiadas soluciones, soluciones para las cuales no estamos validando nada.

5.5.3. 3^{er} experimento - implementar función de aspiración

Vimos como posibilidad implementar una funcion de aspiración para tabú con memoria de aristas. Nuestra función de aspiración es muy simple, si el vecino a analizar pertenece a la memoria pero su costo es menor al mejor costo analizado hasta el momento, entonces ignoramos el hecho de que nuestra memoria nos indica descartarlo. La implementación se encuentra en el archivo heuristicas.cpp en la función *obtenerMejorConMemoriaDeAristasConAspiracion*.

Esto no nos dio ningún tipo de resultado, el gráfico resultante de ejecutar nuevamente el experimento de variar la memoria para esta implementación no varió respecto de lo que podemos observar en la Figura 11.

5.5.4. 4^{to} experimento - variando el porcentaje de descarte

Pensando en lo grandes que son los filtros para nuestras vecindades notamos que quizás la situación estaba más relacionada con las probabilidades de descarte, y que un 80 % (que es el valor que utilizamos en los experimentos previos) podría ser un exceso, generando subvecindades muy pequeñas donde el análisis se vuelve errático.

Con esta motivación ejecutamos el siguiente experimento, variar la probabilidad de descarte entre 1 % y 90 % para los grafos *g21, g48* y *g96*, los resultados se pueden observar en la Figura 12 y son bastante motivadores, podemos ver con claridad que con porcentajes tan altos las heuristicas pierden el enfoque y empiezan a actuar erráticamente como predijimos.

5.5.5. 5^{to} experimento - juntando los resultados previos

Por último analicemos entonces, cómo afecta el tamaño de la memoria a tabú search. Es decir, ejecutemos nuevamente el 2^{do} experimento (sección 5.5.2) pero bajando el porcentaje de descarte al 30 % para disminuir la incertidumbre y los comportamientos erráticos. Podemos ver en la figura 13 cómo el tamaño de la memoria no afecta la calidad de las soluciones, confirmando nuestras hipótesis.

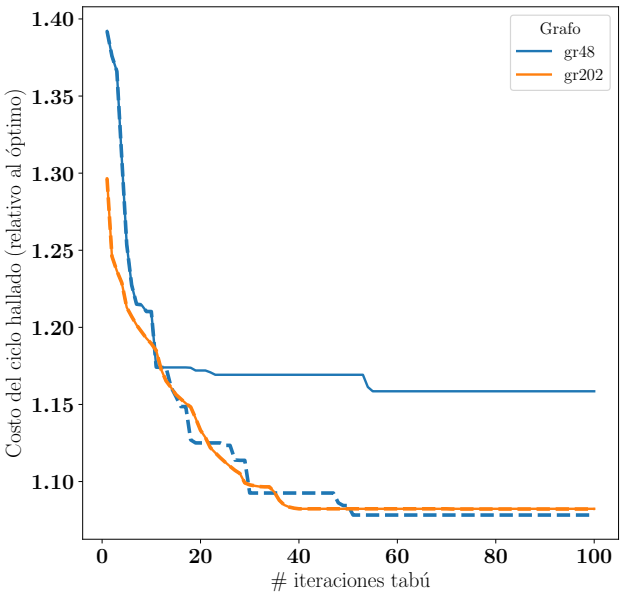

Figura 10: Costo del camino hallado por tabú en función del número de iteraciones. Linea sólida: memoria arista. Punteada: memoria soluciones.

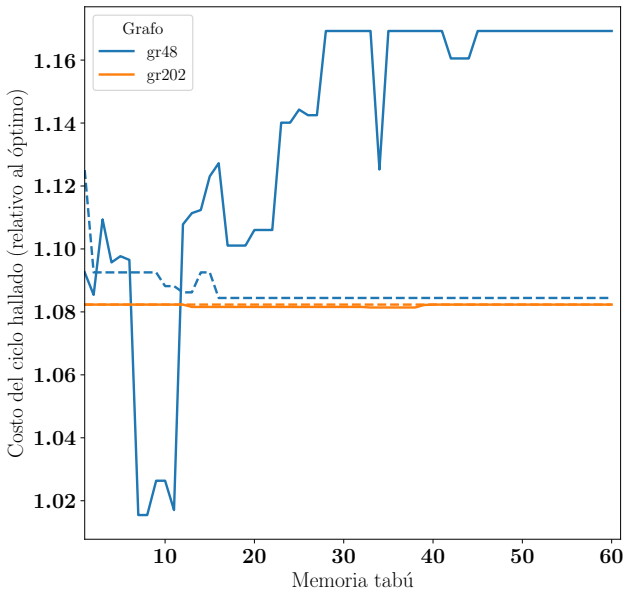

Figura 11: Costo del camino hallado por tabú en función del tamaño de memoria. Linea sólida: memoria arista. Punteada: memoria soluciones.

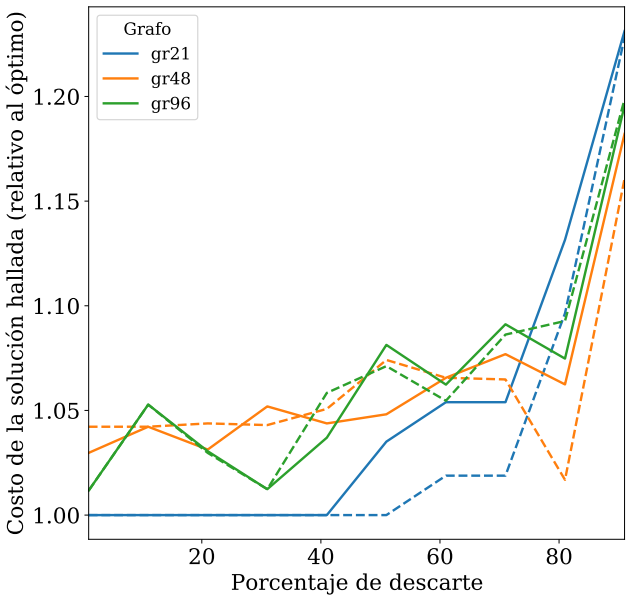

Figura 12: Costo del camino hallado por tabú en función del porcentaje de descarte. Linea sólida: memoria arista. Punteada: memoria soluciones.

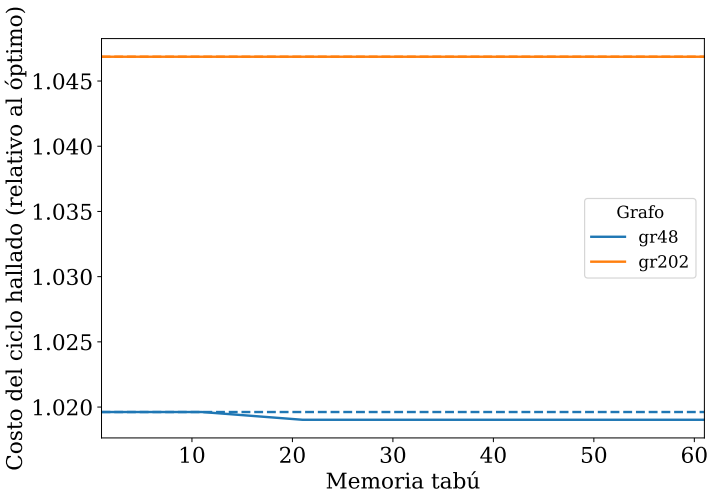

Figura 13: Costo del camino hallado por tabú en función de la memoria con porcentaje de descarte 30. Linea sólida: memoria arista. Punteada: memoria soluciones.

5.6. Tabú - Complejidad temporal

En esta subsección estudiaremos el impacto en la complejidad temporal de variar los distintos parámetros configurables de la metaheurística tabú. Por un lado, en la figura 14 vamos a medir el tiempo de ejecución del algoritmo variando la cantidad de iteraciones realizadas, mientras que en la figura 15 variaremos el tamaño de la memoria. Es factible asumir que en ambos casos, la memoria por soluciones va a tener mayor crecimiento ya que en esta se almacenan soluciones enteras, que tienen n elementos, contra la memoria por aristas, que simplemente contiene pares.

Si analizamos la complejidad de obtener la mejor solución filtrando por la memoria, vemos que en ambos casos hay que, por cada elemento de la subvecindad comprobar que este no se encuentre en la memoria. En el caso de la memoria de aristas, esto implica verificar si los valores de dos pares son iguales, y en el caso de la memoria de soluciones hay que primero alinear las 2 soluciones (que ambos circuitos comiencen a recorrerse desde el mismo nodo) y luego compararlas elemento a elemento, teniendo cada solución n elementos.

En la figura 14, utilizamos una memoria de tamaño 100. Por otro lado, en la figura 15 determinamos que el límite de la cantidad de iteraciones sea 50. En ambos casos, la solución inicial provista a tabú fue la computada por la heurística de AGM.

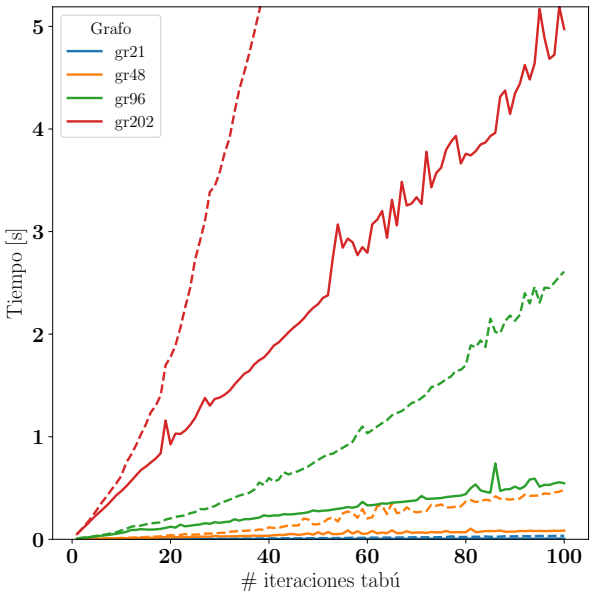

Figura 14: Complejidad de la búsqueda tabú en función del número de iteraciones. Línea punteada: memoria soluciones. Sólida: memoria aristas.

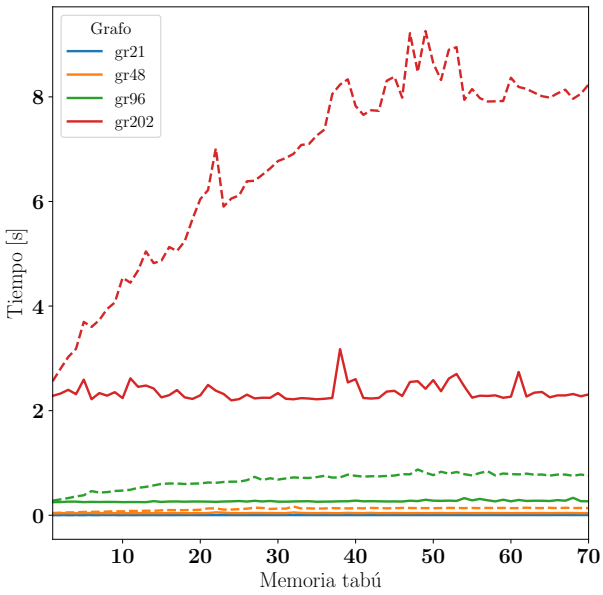

Figura 15: Complejidad de la búsqueda tabú en función del tamaño de memoria. Linea sólida: memoria aristas. Puntos: memoria soluciones.

Puede verse en los gráficos que se cumplen nuestras hipotesis sobre el crecimiento temporal según estos parámetros. Tabú por memoria de aristas aumenta de manera lineal en la cantidad de iteraciones y el tamaño de la memoria parece ser poco significativo, en cambio en el caso de la memoria por soluciones en ambas figuras puede verse una curva de crecimiento mucho más pronunciada.

Estancamiento

Cabe destacar que, en el gráfico que estudia la complejidad temporal en función del tamaño de la memoria, tenemos un punto de quiebre para las 4 instancias: Al ser 50 el límite de iteraciones, inevitablemente la memoria quedará sin utilizar desde la posición 50 en adelante. Esto se debe a que durante las primeras 50 iteraciones, en cada una de ellas a lo sumo guardamos una solución (o un par de aristas, según el tipo de memoria) en la memoria. De esta forma, no estaremos aprovechando las siguientes posiciones de memoria. Esto se traduce a, salvo picos irregulares, un aplanamiento en el costo temporal cuando el tamaño de la memoria es mayor que 50.

En las 4 instancias, la evolución del costo temporal de tabú con memoria de aristas se ve menos afectada por este punto de quiebre dado que la naturaleza de esta memoria no provocaba costos altos a la hora de acceder, comparar y guardar un valor guardado en la memoria. En cambio, en tabú con memoria de soluciones, es más abrupta la diferencia antes y después de que el tamaño de la memoria es 50. Esto se condice con nuestra hipótesis ya que es más notoria la diferencia de comparar 2 soluciones (a la hora de filtrar las soluciones recientemente visitadas de la subvecindad) y asignar un nuevo valor a una posición de la memoria (a la hora de registrar una nueva solución visitada) contra no realizar ninguna de estas 2 acciones.

5.7. Experimento final

Para finalizar la etapa de experimentación computacional, procedemos a elegir las mejores configuraciones de la metaheurística Tabú Search en base a los resultados plasmados en los gráficos. Con estos valores óptimos, pondremos a prueba una vez más este algoritmo procesando nuevas instancias de las cuales conocemos sus soluciones óptimas. Este será un experimento integrador ya que acarrea los conocimientos adquiridos en las subsecciones anteriores y los combina para estudiar qué tanto podemos acercarnos a soluciones halladas por otres autores.

5.7.1. Elección de las configuraciones óptimas

Esta subsección está destinada a explicar qué configuraciones de la metaheurística Tabú elegimos para este último experimento y las motivaciones que nos llevaron a tomar estas decisiones. Explicaremos el valor de cada uno de los parámetros recibidos por el programa, que se encuentran en README.md. Antes, cabe destacar que la elección de la solución inicial brindada a Tabú nos resulta indiferente, por lo que tomamos *solInicial* = AGM.

Para empezar, vimos en la subsección 5.5.1 que Tabú utiliza por lo menos 40 iteraciones para las instancias *gr48* y *gr202* en las que la solución hallada sigue mejorando. Luego de esto, tabú no logra optimizar la solución. Es por esto que elegiremos que el umbral sea de cantidad de iteraciones sin mejoras, a costa de aumentar la complejidad temporal de las ejecuciones. Para asegurarnos de no interrumpir la ejecución prematuramente, determinamos que *criterioParada* = *cantIteracionesSinMejora* y además que *itParada* = 50.

Por otro lado, vimos que el tamaño de la memoria realmente no es un parámetro relevante a la hora de obtener un buena solución. Lo que sí pudimos ver es que, para evitar estancamientos innecesarios, el tamaño de la memoria no debe ser mayor que la cantidad de iteraciones. Esto lo podemos garantizar tomando *tamanioMemoria* = 50.

Además, en la subsección 5.5.4 vimos que si descartamos hasta un 50 % de la vecindad a explorar, la metaheurística Tabú basada en memoria de aristas nos brinda una solución mejor que la memoria de últimas soluciones exploradas. Es por esto que tomamos *tipoMemoria* = aristas y *probaDescarte* = 50.

5.7.2. Las nuevas instancias a procesar

Como en este experimento fijamos las configuraciones explicadas recientemente, podemos tomar un conjunto de más instancias que en los experimentos anteriores ya que solo ejecutaremos una vez nuestra implementación de Tabú para cada instancia. Es por esto que hemos tomado todas las instancias de http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/index.html para las cuales se conoce su solución óptima. Es decir, tomamos todos los archivos *tsp* que además contengan la extensión *.opt.tour*.

Dado que anteriormente hemos utilizado 4 instancias pertenecientes a esta biblioteca, nuestra hipótesis es que nos acercaremos significativamente a las soluciones óptimas, a una distancia no mayor al 10 %. Las configuraciones elegidas nos permiten ingresar en este rango, como se ve en las figuras de la sección 5.5, en las que es posible que el costo de nuestra solución hallada relativo al óptimo sea menor que 1.1.

5.7.3. Resultados

A continuación, presentamos en la figura 16 los resultados de haber ejecutado las 19 instancias para las que se conoce su solución óptima. En esta figura, vemos el error relativo de los costos obtenidos con respecto a los costos de las soluciones óptimas. Además, en el cuadro 1 presentamos los valores exactos correspondientes a las 5 instancias con mayor cantidad de vértices.

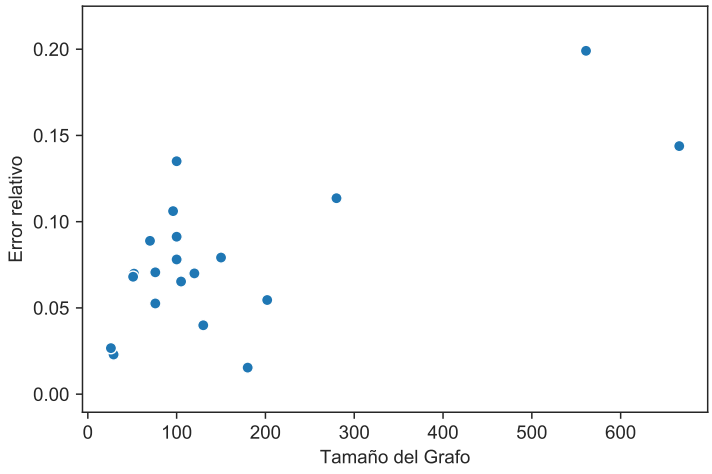

Figura 16: Error relativo de las 19 instancias con respecto a las soluciones óptimas

Instancia	Costo Heurística	Costo Óptimo	Costo en relacion al óptimo
gr666	336694	294358	1.1438
pa561	3313	2763	1.1990
a280	2872	2579	1.1136
gr202	42352	40160	1.0545
brg180	1980	1950	1.0153

Cuadro 1: Resultados de las 5 instancias más grandes

Vemos que en todas las instancias, nuestra solución tiene un costo que es a lo sumo 1.19 veces el costo de la solución óptima. Específicamente, en solo 4 de las instancias (*a280*, *kroc100*, *pa561* y *gr666*), nuestra solución tiene un costo más de 1.1 veces el costo óptimo.

Teniendo en cuenta las 19 instancias, el costo promedio de nuestra solución relativo al óptimo es 1.07, con lo que nuestra hipótesis ha sido validada. Podemos así afirmar que nuestra implementación de la metaheurística tabú brinda soluciones de calidad.

6. Conclusiones

En el presente trabajo se realizó un estudio completo del problema del viajante (TSP) modelado a través del problema de hallar circuitos Hamiltonianos en un grafo completo pesado. Se presentó el problema y se analizó la dificultad para hallar una solución óptima, exponinendo la necesidad de implementar algoritmos heurísticos y metaheurísticos de búsqueda para obtener costos de ciclos cercanos al óptimo y estimar su calidad.

Respecto de las heurísticas, se implementaron tres algoritmos del tipo goloso: AGM, Inserción y VecinoMásCercano. Se estudiaron sus complejidades temporales de manera teórica y experimental, hallando una concordancia entre ambos resultados. Se compararon los costos de los ciclos hallados para estas heurísticas en instancias de grafos de diferentes características, concluyendo que la disponibilidad de aristas de costo barato es un buen indicador de instancias favorables para estos algoritmos. Se estudió el efecto de aumentar el grado de grafo manteniendo el número de posibles costos de aristas fijos y cambiar la disponibilidad de aristas baratas manteniendo fijo el grado. En ambos casos el comportamiento de los valores obtenidos fue el esperado para las lógicas golosas. Comparativamente, el algoritmo de AGM demostró tener un rendimiento inferior en grafos completamente aleatorios pero similar a las otras heurísticas en grafos euclideanos.

Concluimos que la heurística VecinoMasCercano (complejidad de orden $\mathcal{O}(n^2)$) es la más apropiada para atacar grafos estudiados en este informe, puesto que la complejidad de la heurística de Inserción es de orden $\mathcal{O}(n^3)$ sin un beneficio significativo en el costo de la solución. En el caso de AGM para grafos euclideanos, la complejidad es de orden $\mathcal{O}(n^2)$ dando resultados similares a los de VecinoMasCercano, pero con la ventaja de ser 1-aproximado, con lo cual es preferible en esos casos.

Se estudió también la metaheurística tabú en grafos euclideanos de distintos tamaños con soluciones óptimas conocidas. En primer lugar se realizaron experimentos para determinar tanto los parámetros como los rangos relevantes del algoritmo en los grafos de prueba, para luego optimizarlos para su aplicación. Se examinaron los efectos del número de iteraciones, el tamaño de las memorias utilizadas en la listas tabú y cómo se relacionan entre ellos. También se estudió el efecto de aumentar o disminuir la vecindad mediate el parámetro de descarte. Se concluyó que, como era esperado, el número de iteraciones es en general favorable para la obtención de un mejor resultado. La memoria mostró ser un parámetro más complejo, ya que no mantuvo una tendencia clara a la hora de determinar su injerencia en los resultados de las soluciones finales. Concluimos que no es un parámetro relevante por sí solo, sino que debe estar dentro de un rango determinado principalmente por el número de iteraciones que realizará el algoritmo, y el tamaño de la vecindad fijado por el parámetro de aceptación de soluciones.

Por último, comparamos las soluciones óptimas de instancias de TSP con los valores obtenidos con la metaheurística tabú implementada utilizando parámetros optimizados. Los resultados presentados muestran un desvío con errores relativos del orden del 5 %, por lo que se concluye que la calidad de las soluciones obtenidas mediante la implementación presentada es buena.

Como ideas de implementación futuras, una posibilidad es aleatorizar la semilla en las rutinas de obtencion de la subvecindad (secciones 4.1.4 y 4.2.4), ya que actualmente la distribución implementada es determinística

lo que no permite tener diversidad en la exploración de las soluciones. De esta manera podrían realizarse diversos estudios sobre una instancia determinada para obtener una solución de mejor calidad con menos iteraciones.