DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

TP2: Problema del Viajante de Comercio (TSP)

Algoritmos y estructuras de datos III

Integrante	LU	Correo electrónico
Chimenti, Tomas Agustin	99/18	tach.365@gmail.com
Laconte, Rodrigo	193/18	rola1475@gmail.com
Piaggio, Macarena	212/18	piaggiomacarena@gmail.com
Sorondo, Amalia	281/18	sorondo.amalia@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

El problema del viajante de comercio, o TSP, consiste en determinar dada una lista de ciudades a las cuales se debe transportar mercancías y costos asociados a los caminos entre cada par de ciudades, cuál es el mejor orden para recorrerlas. La definición del mejor recorrido depende del contexto en el que nos encontremos; puede tratarse del camino más rápido, el más económico o el más seguro. El concepto de "mejor"se define entonces por la función objetivo que permite evaluar un conjunto de soluciones posibles con el fin de compararlas y elegir la solución óptima. Se trata entonces de un problema de optimización combinatoria.

El problema puede modelarse mediante un grafo completo donde cada vértice representa una ciudad, cada arco el camino entre cada par de ciudades y cada peso asociado el costo de recorrer dicho camino. Luego, se busca un circuito hamiltoniano, es decir un circuito que recorre todos los vértices del grafo una única vez, cuyo costo sea mínimo entre todos los circuitos posibles.

Para ilustrar un ejemplo, consideremos el grafo completo de cuatro vértices con los siguientes costos asociados a las aristas:

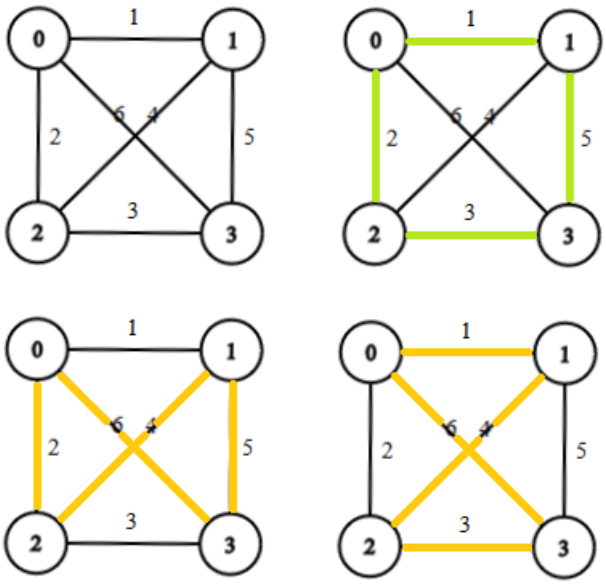

Figura 1: Circuitos hamiltonianos del grafo

El primer recuadro de la figura 1 presenta el grafo con sus costos asociados, en los recuadros siguientes se remarcan en color sus distintos circuitos hamiltonianos. La diferencia entre los circuitos consiste en el orden elegido para recorrer los vértices del grafo, esto implica utilizar distintas aristas y en consecuencia obtener diferentes costos totales del circuito. Podemos observar que el circuito remarcado en verde tiene un costo total igual a 11 mientras que los circuitos naranjas poseen costos iguales a 17 y 14. El primer circuito mencionado representa entonces el circuito hamiltoniano de costo mínimo y por lo tanto es éste el circuito que el problema del TSP intentará devolver o aproximar.

Como mencionamos antes, TSP se basa en un grafo pesado completo, con las ciudades como nodos y las distancias entre estas mismas como pesos de las aristas. Sin embargo, podríamos

utilizar un grafo de este estilo para modelar otros problemas que aparecen en la vida real, por ejemplo:

- cableo de computadoras
- serición arqueológica
- secuencias de genomas
- organización (de tareas, por ejemplo, minimizando tiempo de preparación entre cada una)
- empapelado de paredes[1] (con las posiciones de la pared como las ciudades, y la perdida de papel si empapelamos una posición luego de la otra como pesos de las aristas).

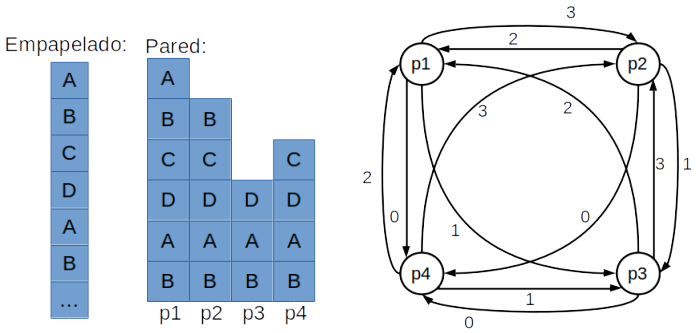

Figura 2: Empapelado como instancia del TSP

Podemos ver, en la figura 2, que el grafo que modela el problema del empapelado no es simétrico, ya que el orden del empapelado de las distintas áreas de la pared nos importa. Si empapelamos $p2$ y luego $p4$ no tenemos perdida de papel, distinto a si lo hacemos en el orden inverso.

Flinders Petrie (1853-1942), arqueólogo británico, trajo la teoría de grafos al mundo de la arqueología. En una de sus grandes excavaciones en Egipto, utilizó una "función de distancia"(la hoy conocida como "distancia de Hamming") para describir la cercanía de las tumbas en el tiempo. Petrie no estaba al tanto de la lógica matemática que estaba usando, pero llevó el problema a una instancia del TSP, incluyendo conceptos matemáticos/computacionales complejos, data mining, heurísticas, y todo en un pizarrón.[2]

Por otro lado, también podemos encontrar aplicaciones del TSP en áreas como la de biología. Conocer la secuenciación completa de ADN del genoma de un individuo proporciona información clínica útil.[3] El método del problema del viajante de comercio entra en juego cuando queremos formar una secuencia de genomas, de manera tal que se maximice la superposición entre los mismos, ya que están formados como una combinación de nucleótidos en línea (ej: CTGTAAT, donde C, T, G y A son nucleótidos).

[1] Garfinkel, Robert S. "Minimizing Wallpaper Waste, Part 1: A Class of Traveling Salesman Problems." Operations Research, vol. 25, no. 5, 1977, pp. 741–751. JSTOR, www.jstor.org/stable/169475. Accessed 23 June 2020. www.jstor.org/stable/169475

[2] https://www.researchgate.net/publication/267974382_Flinders_Petrie_the_travelling_salesman_problem_and_the_beginning_of_mathematical_modeling_in_archaeology

[3] https://en.wikipedia.org/wiki/Whole_genome_sequencing

El problema de encontrar un circuito hamiltoniano en un grafo no está bien resuelto computacionalmente. Esto es debido a que el problema esta categorizado como una clase NP-Hard, por lo tanto no se sabe si existen algoritmos polinomiales para el caso particular de este problema que representa el TSP. Sin embargo, vimos que este problema tiene numerosas y variadas aplicaciones, la mayoría de éstas representan instancias grandes del problema y por lo que no pueden ser resueltas en tiempo exponencial. Se utilizan entonces algoritmos heurísticos para aproximar la solución óptima del TSP. Estos algoritmos brindan soluciones suficientemente buenas y tienen una complejidad polinomial que permite su utilización.

A lo largo de este trabajo estudiaremos distintas heurísticas para resolver el problema del TSP analizando su comportamiento así como la calidad de sus resultados. Presentaremos primero tres heurísticas constructivas: la heurística del vecino más cercano, la de la arista más corta y por último la heurística basada en árbol generador mínimo (AGM). Luego estudiaremos como se puede mejorar una solución obtenida gracias a la metaheurística de búsqueda tabú bajo dos implementaciones distintas: Basado en las ultimas soluciones y basada en aristas.

2. Desarrollo

En esta sección presentaremos los algoritmos implementados para cada heurística, sus complejidades e instancias patológicas. En nuestros algoritmos, decidimos representar a los grafos mediante matrices de adyacencia ya que sabíamos que las instancias con las que ibamos a trabajar eran modeladas por grafos completos. Es decir, las matrices resultantes son densas por lo que no se desperdicia espacio y recorrer todos los vecinos de un vértice tiene, en este caso, el mismo costo que en una representación con listas enlazadas. Además, una gran ventaja de esta representación es la consulta del costo de una arista en tiempo constante. No obstante, la heurística basada en AGM devuelve un árbol que, al recorrerlo utilizando DFS representa el circuito hamiltoniano buscado. Representar un árbol con una matriz resultaba ineficiente por lo que en este caso utilizamos listas de adyacencia.

2.1. Heurística del vecino más cercano

La heurística del vecino más cercano es un procedimiento goloso. Se construye la solución partiendo desde un vértice inicial y en cada paso se agrega al recorrido el nodo que resulte menos costoso de acceder desde el vértice actual y que aún no haya sido visitado.

El algoritmo implementado sigue el siguiente pseudocódigo:

Algorithm 1 Heurística del vecino más cercano

```
function HeuristicaNN(G : KGrafo)
    n ← cantidadVertices(G)
    inicial ← verticeRandom(G)
    actual ← inicial
    agregarVertice(solucion, actual)
    agregarVertice(verticesRecorridos, actual)
    contador ← 1
    while contador < n do
        for all vertice ∉ verticesRecorridos do
            if costo(G, actual, vertice) < costo(G, actual, min) then
                min ← vertice
        actual ← min
        agregarVertice(verticesRecorridos, min)
        agregarVertice(solucion, min)
        contador ← contador + 1
    return solucion
```

La *solucion* y el conjunto de *verticesRecorridos* contienen los mismos vértices, aunque están implementados con estructuras diferentes. Por un lado, la solución se representa con un vector de enteros ya que modelamos los nodos con números del 0 al $n-1$. Mientras que al conjunto de vértices recorridos se encuentra representado con un vector de booleanos, siendo *true* en el caso que el índice coincida con un vértice que ya hayamos agregado a la solución. De esta manera podemos verificar por qué vértices ya pasamos en O(1).

Analicemos la complejidad de la heurística; el ciclo principal utiliza un contador que se incrementa en cada iteración y equivale a la cantidad de vértices ya recorridos, consta entonces de n iteraciones. En cada iteración, se recorren todos los vértices realizando los chequeos correspondientes y actualizando el costo mínimo de ser necesario, esto se realiza en tiempo constante. Luego, el algoritmo tiene complejidad temporal de O(n^2).

Como se trata de una heurística golosa, el algoritmo no revisa las decisiones tomadas. Esto puede impactar negativamente en la calidad de la solución. Por como está definida la heurística, la solución puede ser tan mala como se imagine, dependiendo de como sea la estructura del grafo. Por esta razón, elegimos un vértice inicial aleatorio entre todos para elegir un inicial distinto en otra corrida, en caso de que tomemos eventualmente un vértice que genere una solución deficiente.

2.2. Heurística de la arista más corta

La presente heurística tiene por estrategia recorrer los nodos en relación a las aristas de menor costo. Partiendo de dos vértices conectados con el menor eje del grafo, se empieza un ciclo donde en cada iteración se elige el siguiente nodo respetando las siguientes condiciones:

- No se forma un ciclo.
- Que el vértice analizado no tenga aristas de grado mayor a 3.

Si agregamos n-1 aristas, siendo n la cantidad de vértices, habremos formado un camino hamiltoniano. Para obtener un ciclo, debemos agregar la arista entre los nodos extremos del este camino. El procedimiento de este algoritmo también puede ser categorizado de goloso, por el hecho de no revisar las decisiones tomadas con respecto al vértice agregado.

Una posible implementación radica en una modificación del algoritmo de Kruskal para generación de arboles generadores mínimos.

Algorithm 2 Heurística de la arista más corta

function $HeuristicaAMC(G : KGrafo)$
$X_T \leftarrow \emptyset$
$i \leftarrow 1$
$Cand \leftarrow X$
$ordenar(Cand)$
$i \leftarrow 1$
while $i < n - 1$ **do**
$(u, w) \leftarrow min(Cand)$
$Cand \leftarrow Cand\{(u, w)\}$
si u y w no pertenecen a la misma componente conexo de T=(V, X_T) y el grado de u y w sea menor a 2, hacer:
$X_T \leftarrow X_T \bigcup (u, w)$
$i \leftarrow i + 1$

La modificación que se realiza a Kruskal, es comprobar en cada paso si el grado de las aristas es menor a 2, si no lo es, se procede a rechazar ese vértice. Los vértices y aristas aceptadas serán guardados en un grafo nuevo y se procederá a recorrerlo linealmente para poder guardar el camino hamiltoniano

La complejidad el algoritmo es $O(m * log(m))$ ya que el algoritmo de Kruskal tiene esa complejidad y los cambios agregados no modifican el análisis asintótico.

2.3. Heurística basada en AGM

La siguiente heurística se basa en la construcción de un árbol generador mínimo (ahora en mas AGM) a partir del grafo dado y luego se procede a recorrerlo a partir del criterio DFS para poder conseguir un camino hamiltoniano asociado al grafo. El siguiente pseudo-código sintetiza la idea de la heurística.

Algorithm 3 Heurística del árbol generador mínimo

function $HeuristicaAGM(G : KGrafo)$
caminoDFS es vector de ints
AGM es un grafo con estructura de lista
$AGM \leftarrow G.getAGM()$
$caminoDFS \leftarrow n.getDFS()$
$caminoDFS.push\_back(caaminoDFS[0])$
return $caminoDFS$

Un AGM se basa en la idea de construir un grafo sin circuitos de menor costo posible. Una ilustración de esta idea es la figura 3. Este árbol siempre es posible construirlo partiendo de un grafo conexo, que los utilizados en este trabajo lo son al ser completos.

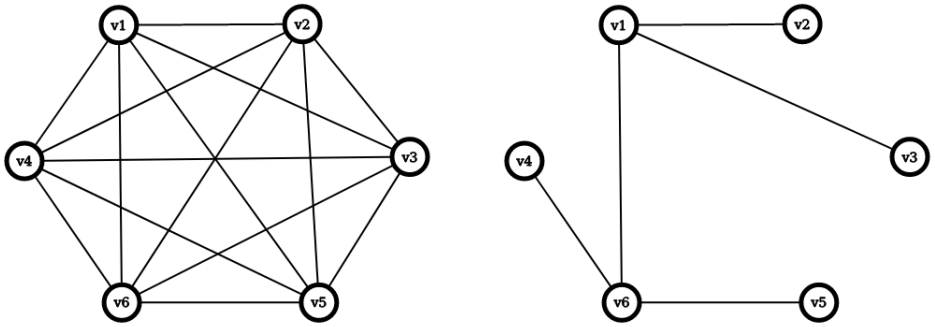

Figura 3: Ejemplo de AGM a partir de un grafo completo de 6 aristas

Luego, al árbol generado se buscar recorrerlo con `Búsqueda en profundidad` (DFS por sus siglas en ingles). Su funcionamiento consiste en ir expandiendo todos y cada uno de los nodos que va localizando, de forma recurrente, en un camino concreto. Cuando ya no quedan más nodos que visitar en dicho camino, regresa (Backtracking), de modo que repite el mismo proceso con cada uno de los hermanos del nodo ya procesado. La siguiente imagen es una continuación del ejemplo anterior aplicando DFS

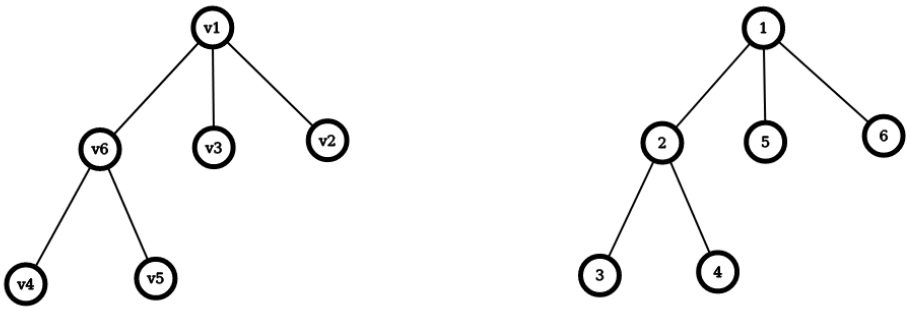

Figura 4: Recorrido DFS del AGM anterior visto como un arbol, los numeros de los vertices indican el camino elegido por DFS

Por ultimo, se procede a construir el camino hamiltoniano siguiendo los vértices marcados por DFS, con el detalle de conectar el ultimo nodo con el primero.

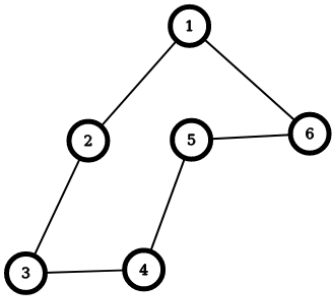

Figura 5: Camino Hamiltoneano

La complejidad de esta heurística depende mucho del algoritmo a usar para construir el AGM. En este trabajo vamos a utilizar el algoritmo de Kruskal en su variante con la estructura de datos Disjoint-set con ranking y compresión de caminos. También se utilizara QuickSort con pivote elegido al azar[4] para el ordenamiento de aristas. La complejidad es $O(m*log(m))$[5] donde m denota la cantidad de aristas y n la cantidad de vértice. Ahora, tenemos que sumarle el costo de llamar a la función DFS que cuya complejidad es $O(m+n)$, que nos da un total de $O(m*log(m)+n+m) = O(m*log(m)+n)$.

2.4. Metaheurística Tabu Search

Por último, vamos a enunciar una metaheurística que parte de una solución inicial que puede ser provista por cualquiera de las heurísticas nombradas anteriormente. Este algoritmo implementa una búsqueda a partir de una solución con ayuda de una estructura auxiliar, la cual restringe ciertas características a la hora de elegir una nueva solución. Por esta razón se le da el nombre de Búsqueda Tabú a la metaheurística.

[4]Si bien la complejidad en el peor caso es $O(n^2)$, las probabilidades de tener ese caso son prácticamente nulas, entonces vamos a considerar que es $n*log(n)$

[5]En realidad la complejidad es $O(m*log(m)*\alpha(m))$ con $\alpha(m)$ la función inversa de Ackermann, pero debido al lento crecimiento de la función, en la practica podemos considerar acotarlo por alguna constante.

Algorithm 4 Metaheurística búsqueda tabú

```
function TabuSearch(G : KGrafo, tamMem : int, iterMax : int, porcentajeVecindad : float)
    n ← cantidadVertices(G)
    ciclo ← solucionInicial(G)
    mejorCiclo ← ciclo
    memoria ← ∅
    while i < iterMax do
        vecindad ← definirVecindad(n, porcentajeVecindad)
        aristas ← ⟨−1, −1⟩
        mejorVecino ← buscarMejorVecino(G, ciclo, vecindad, aristas)
        if ¬tabu(mejorVecino, aristas) ∨ aspiracion(mejorVecino, aristas) then
            ciclo ← mejorVecino
            if tam(memoria) = tamMem then
                olvidar(memoria)
            agregar(memoria, mejorVecino, aristas)
            if costo(G, ciclo) < costo(G, mejorCiclo) then
                mejorCiclo ← ciclo
        i ← i + 1
    return mejorCiclo
```

Analicemos la complejidad de la metaheurística. Como se puede apreciar en el pseudocódigo, primero se toma una solución inicial. Allí tenemos un costo de $O(heurística)$, que se puede reemplazar por algunas de las complejidades antes analizadas.

Algorithm 5 definirVecindad

```
function definirVecindad(cantVertices : int, porcentajeVecindad : float)
    indices ← vectorVacio()
    for all 0 ≤ i < cantVertices do
        for all i + 1 ≤ j < cantVertices do
            agregarAtras(indices, ⟨i, j⟩)
    randomShuffle(indices)
    porcion ← slice(indices, 0, tam(indices) ∗ porcentajeVecindad − 1)
    return porcion
```

Dentro del cuerpo del ciclo principal, se puede ver que la función define una vecindad (5). Genera todos los pares de índices necesarios en $O(\binom{cantVertices}{2})$, que equivale a todos los movimientos posibles a partir de la solución parcial mediante el método de 2-opt. Luego, se mezclan los elementos de la vecindad de manera aleatoria y procede a elegir un subconjunto de la misma, lo cual está implementado con una complejidad lineal en el tamaño de los vectores.

Seguimos con la búsqueda del mejor vecino en la vecindad, o porción de la misma (6). Tenemos un ciclo de $O(|porcionVecindad|)$ iteraciones donde dentro se realizan operaciones de complejidad lineal en la longitud de los circuitos (cálculo de costos) y comparaciones $O(1)$. En otras palabras, aquí se realiza la búsqueda local mediante 2-opt.

Luego, procedemos a realizar algunos chequeos: comprobar si debemos rechazar la solución por cumplir alguna condición de la lista tabú y si tenemos que considerarla por cumplir con la función de aspiración. Acá se podría realizar la distinción que tenemos entre las dos versiones de Tabú Search:

Algorithm 6 buscarMejorVecino

```
function buscarMejorVecino(G : KGrafo, ciclo : vector, vecindad : vector, indices : par)
    mejorCicloVecino ← ciclo
    for all par ∈ vecindad do
        i ← par[0]
        j ← par[1]
        costoActual ← costo(G, ciclo[i], ciclo[i + 1]) + costo(G, ciclo[j], ciclo[j + 1])
        costoCruce ← costo(G, ciclo[i], ciclo[j]) + costo(G, ciclo[i + 1], ciclo[j + 1])
        if costoActual > costoCruce then
            cruzado ← cruzarAristas(ciclo, par)
            if costo(G, cruzado) < costo(G, mejorCicloVecino) then
                indices ← par
                mejorCicloVecino ← cruzado
    return mejorCicloVecino
```

- Por un lado podemos memorizar en la lista tabú las soluciones completas, de manera tal que rechazaremos un resultado si se encuentra en la lista. Chequearlo tendría un costo $O(|tabu| * cantVertices)$, y $O(cantVertices)$ por agregar un elemento a la lista. Podría considerarse también alguna función de aspiración, tal como aceptar soluciones un 5 % peores, aunque no lo implementamos en nuestro código,
- Por el otro, memorizar algún elemento relacionado con la estructura de las soluciones, nosotros elegimos las aristas participantes en el 2-opt. Las complejidades serían similares, con $O(|tabu|)$ y $O(1)$ para el chequeo y manejo de la lista. Para esta versión sí implementamos una función de aspiración, considerar soluciones que tengan un costo mejor (a pesar de ser rechazadas por la lista tabú). El precio de esto último tiene la misma complejidad que calcular el costo de un circuito.

A esto se le suma el precio a pagar por el cálculo de los costos de los ciclos, que termina siendo $O(cantVertices)$, y por asignar *mejorCiclo* a *ciclo*, el copiado también resulta $O(cantVertices)$. No olvidemos tampoco que este proceso se realiza $O(iterMax)$ veces.

Entonces, si juntamos todas las complejidades, nos queda que la función de TabuSearch tiene orden $O(heuristica + iterMax * (\binom{n}{2} + \binom{n}{2} * \%V * n + |T| * n))$, donde n es la cantidad de vértices, T es la lista tabú y $\%V$ es el porcentaje de la vecindad ($0 \leq \%V \leq 1$). Lo que resulta en $O(heuristica + iterMax * (\binom{n}{2} * n + |T| * n)) = O(heuristica + iterMax * n * (\binom{n}{2} + |T|))$

3. Experimentación

En esta sección presentaremos los experimentos realizados con el objetivo de evaluar cada heurística en cuanto a la calidad de sus soluciones así como el tiempo de ejecución insumido. En un primer lugar, estudiaremos para cada método las instancias que producen mejores y peores resultados. La calidad del resultado de una heurística la mediremos en función de cuánto mayor es en proporción a la solución óptima. En un segundo lugar, analizaremos si existen características que impacten en el tiempo de ejecución de las heurísticas. Luego experimentaremos con los parámetros de la metaheurística de búsqueda tabú evaluando cómo varía la performance del método y analizar su tiempo de ejecución. Finalmente, evaluaremos la calidad de las soluciones de la metaheurística al utilizar los parámetros óptimos encontrados.

Se realizaron las ejecuciones utilizando el lenguaje de programación C++ sobre una máquina con las siguientes características: Procesador Intel I7-8550U 1.80GHz, 12GB de RAM, SO: Windows 10 utilizando WSL.

Para la mayor parte de los experimentos planteados decidimos generar nuestros propios conjuntos de instancias. De esta forma pudimos fijar la mayor cantidad de parámetros, como el tamaño o las características particulares de las instancias con las que trabajamos. Esto nos pareció útil al estudiar el impacto de algún parámetro o característica particular.

3.1. Calidad de la solución

Como mencionamos, comenzaremos con el análisis de la calidad de la soluciones. Definimos la *calidad* de la siguiente manera:

$$C = \frac{\acute{o}ptimo - obtenido}{\acute{o}ptimo} * 100.$$

La distancia entre el el *óptimo* y el resultado *obtenido* no es siempre positiva, ya que nuestra mejor solución la encontramos con algunas corridas de la heurística de NN, tomando la mejor entre ellas. Es decir, podríamos tener *calidad* "negativa", lo que significaría un costo menor en el circuito hamiltoniano encontrado.

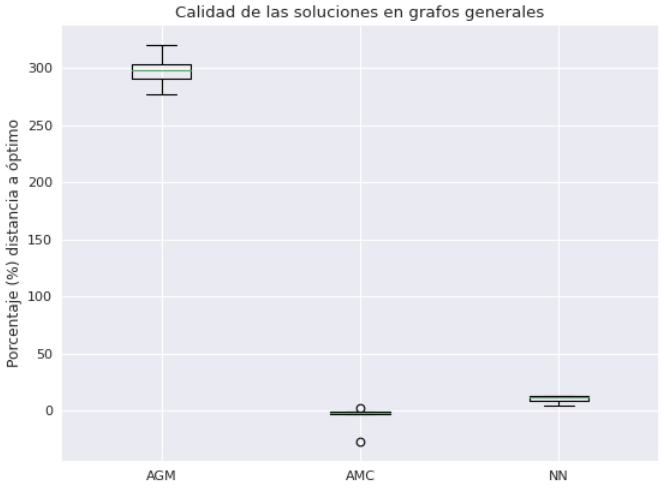

Figura 6: Calidad de soluciones - grafos random.

En primer lugar vamos a comparar las tres heurísticas en grafos generados aleatoriamente por nosotros, estableciendo un tamaño fijo de 100 vértices. Con aleatorios nos referimos también a que no se especifica si son modelos que respeten distancias euclidianas o no.

Planteamos este experimento con la idea de empezar a comprender cómo reaccionan los distintos métodos a cada tipo y forma de instancia. Previo a realizar las corridas, se nos ocurrió que los resultados iban a ser bastante variados, así como lo son las instancias. Pero podemos ver que la heurística basada en AGM se vio totalmente desfavorecido, suponemos que por el hecho de que no aseguramos distancias euclidianas. Por otro lado, las heurísticas de AMC y NN se mantuvieron muy cerca de las soluciones óptimas, en comparación con la heurística de árbol.

3.1.1. Vecino más cercano

En este experimento intentaremos determinar si existen instancias para las cuales al aplicar la heurística de vecino más cercano ésta devuelve peores resultados que para otras. Nuestra hipótesis es que el resultado retornado por el método puede estar tan alejado de la solución óptima como uno quiera si se fija el vértice inicial. Tomando como vértice inicial el 0, consideremos el siguiente caso donde el vecino más cercano del 0 es el vértice 1, el del 1 es el 2 y así hasta el vértice n. Construimos así un camino hamiltoniano de costo mínimo, para completar el circuito estamos obligados a unir los vértices n y 0 pero esta arista puede tener un costo arbitrariamente grande. Luego, la solución puede ser arbitrariamente mala.

Para contrastar nuestra hipótesis con los resultados experimentales fabricamos el peor caso descripto anteriormente asignando costos a las aristas de la siguiente forma. Si la arista unía los vértices i e $i+1$ con $i = 0, ..., n$ siendo n la cantidad de vértices entonces se asigna el costo i. Si la arista une los vértices 0 y n entonces le asignamos costo $3n$, éste será el costo arbitrariamente grande. Si la arista une cualquier otro par de vértices entonces su costo será $n+1$. Esta asignación de costos produce al aplicar la heurística las condiciones que buscábamos en el peor caso. Para ejecutar el experimento modificamos nuestro código para que fije el vértice inicial al vértice 0.

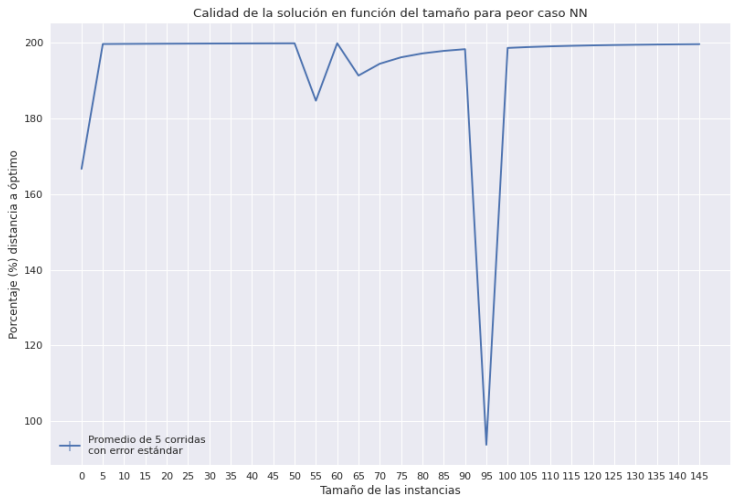

Figura 7: Peor caso de vecino más cercano

Para realizar el experimento promediamos 5 corridas de cada instancia de peor caso para los tamaños de grafo entre 0 y 145 vértices. La figura 7 presenta los resultados obtenidos.

Para casi todos los tamaños el porcentaje de distancia al óptimo es aproximadamente del 200 %, esto representa soluciones considerablemente peores en calidad que las obtenidas en el experimento asociado a la figura 6. Por lo tanto se comprueba nuestra hipótesis; la instancia generada representa un peor caso en la calidad de la solución de la heurística de vecino más cercano. Estos peores casos se pueden amortizar eligiendo el vértice inicial aleatorio y tomando el promedio de diferentes corridas.

3.1.2. Arista más corta

Procedemos a experimentar con la heurística de AMC. Hipotetizamos que este algoritmos tiene un pobre rendimiento en cuestión de soluciones obtenidas cuando es sometido a una familia de grafos con ciertas particularidades. Este conjunto de grafos es idéntico al que fue usado por la heurística de vecino mas cercano mas arriba, dando a entender que tanto este algoritmo como el de NN comparten casos donde su perfomance en materia de solución buscada es bajo. Esto se explica debido a la naturaleza de AMC, donde se ordenan todas las aristas en relación a su costo en orden creciente. Esto tiene el problema que si existe un camino hamiltoniano desde el vértice 1 hasta el vértice n (donde n es la cantidad de vértices) de costo mínimo posible, pero la arista faltante pare completar el circuito, es decir, aquella que conecta el nodo n con el primero, es demasiado elevada su costo, entonces ese circuito hamiltoniano puede estar lejos del óptimo. Para contrastar nuestras hipótesis, utilizaremos los grafos generados (que están explicados previamente) y comparamos con la solución óptima de los mismos.

Realizamos el mismo experimento que el de la sección anterior pero aplicando la heurística de arista más corta. La figura 8 presenta los resultados del experimento, éstos son idénticos a los obtenidos en el experimento anterior. Esto se debe a que ambas heurísticas por cómo están construidas tomaron los mismos circuitos para cada instancia y por lo tanto obtuvieron los mismos resultados. Al igual que en el experimento del vecino más cercano los resultados respaldan nuestra hipótesis de asumir que el algoritmo toma más tiempo en ejecutar la instancia de peor caso que construimos.

3.1.3. Árbol Generador Mínimo

Ahora pasamos a enfocarnos en la heurística de AGM. Luego de analizar la figura 6, pasamos a suponer que su bajo rendimiento se debió a que no podíamos asegurar que las distancias entre los vértices sean euclidianas. Lo que nos dio pie a este experimento, en el cual comparamos la calidad de las soluciones de AGM para grafos con distancias euclidianas y no euclidianas.

En el gráfico de la figura 9 podemos observar la diferencia. Vemos que si las distancias son euclidianas las soluciones obtenidas están rozando el óptimo con una *calidad* menor al 40∼30 %. Lo cual tiene bastante sentido si pensamos en lo visto en la teórica: *si las distancias del grafo son euclidianas, la heurística del árbol generador es un algoritmo 1-aproximado.* En cambio, no podemos decir lo mismo sobre las soluciones obtenidas en el otro tipo de instancias, que escalaron a una calidad de 300 %.

3.2. Tiempo de ejecución

El objetivo de este experimento es determinar si existen casos donde el tiempo de ejecución de los algoritmos es mayor que otros. Para esto tomamos los diferentes tipos de instancias que construimos en los experimentos anteriores: las instancias random, las instancias euclidianas y las que representan el peor caso de las heurísticas de vecino más cercano y arista más corta.

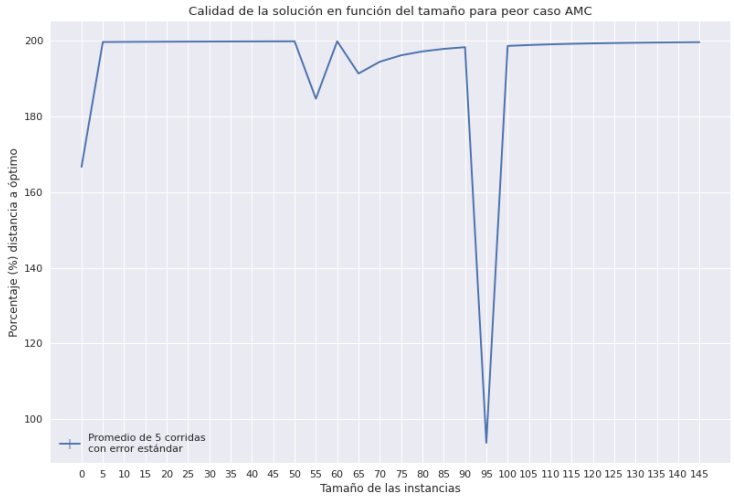

Figura 8: Peor caso de arista más corta

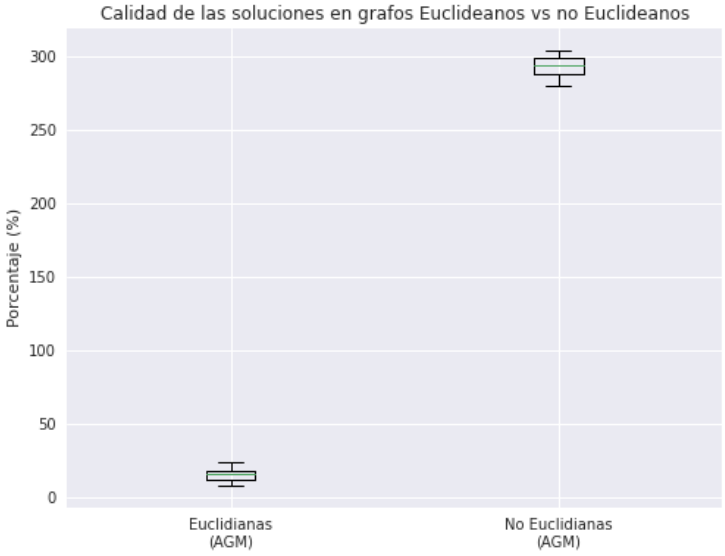

Figura 9: AGM en grafos euclidianos vs no euclidianos

Nuestra hipótesis es que no habrán para ninguna heurística instancias que impliquen más tiempo de ejecución que otras. Esto se debe a que todas las heurísticas recorren todo el conjunto que están considerando sin importar las características de la instancia que están considerando. Es decir, para cada vértice el vecino más cercano evalúa los costos de la arista que une el vértice actual con todos los demás vértices no pertenecientes al circuito. No existen instancias determinadas para las cuales se realicen menos chequeos. Las heurísticas de arista más corta y de árbol generador mínimo utilizan el algoritmo de Kruskal. Por cómo implementamos este algoritmo siempre se realizan

la misma cantidad de operaciones sin importar las características particulares de la instancia. Como AGM utiliza el algoritmo de Kruskal y luego únicamente realiza un recorrido DFS del grafo resultante podemos afirmar que el tipo de instancia no impactará en el tiempo de ejecución total.

Corrimos los tres métodos con instancias de tamaño variando entre 50 y 250 con un salto de 50 para los tres conjuntos de instancias distintos. Para obtener cada valor representado en el gráfico se tomó el promedio de la corrida sobre 10 instancias distintas de mismo tamaño y tipo de instancia. A continuación presentamos los resultados obtenidos para cada heurística:

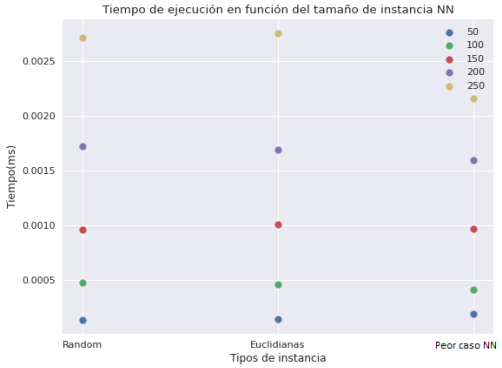

Figura 10: Tiempo de ejecución de vecino más cercano

Como podemos ver en la figura 10 los tiempos de ejecución de la heurística de vecino más cercano es casi idéntica para los conjuntos de instancias Random y Euclidianas. Para las instancias que representan el peor caso de vecino más cercano el tiempo es ligeramente inferior a los tiempos de los otros conjuntos para todos los tamaños y considerablemente inferior para el tamaño de instancia igual a 250. Sin embargo, estas diferencias no resultan muy significativas por lo que podemos atribuirlas a errores aceptables de experimentación.

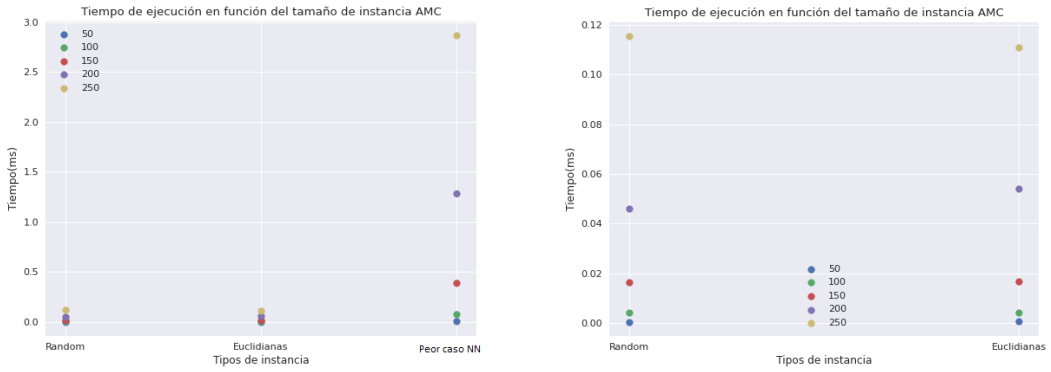

Figura 11: Tiempo de ejecución de arista más corta

Las figuras 11 y 12 nos muestran resultados que contradicen nuestra hipótesis ya que el tiempo de ejecución de las heurísticas de AMC y AGM resulta considerablemente mayor al aplicarlas a

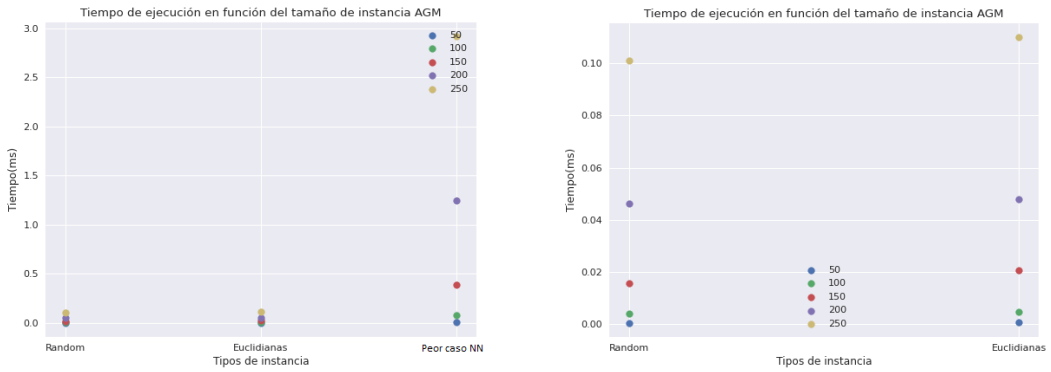

Figura 12: Tiempo de ejecución de árbol generador mínimo

instancias de tipo peor caso de NN. Para tratar de entender por qué podría estar sucediendo esto analizamos nuestro código con más detalle, especialmente el procedimiento de Kruskal ya que lo utilizaban ambas heurísticas. Nos dimos cuenta entonces que el uso de Quicksort para ordenar las aristas podía impactando negativamente en el tiempo de ejecución total ya que este algoritmo tarda más tiempo en ordenar el arreglo de entrada si éste tiene valores repetidos. Esto se debe a que el pivote separa los elementos del arreglo de manera tal que quedan dos arreglos de tamaños muy disparejos, por lo que el costo temporal termina siendo casi cuadrático. Por cómo están generadas las instancias de peor caso de NN, la mayoría de las aristas tienen costo igual al tamaño del grafo por lo que el arreglo a ordenar que recibe Quicksort tiene gran parte de los elementos iguales. Esto explicaría entonces el gran costo de tiempo de ejecución para las instancias de peor caso de NN. Podríamos utilizar otro algoritmo de ordenamiento que no admita peor y mejor caso para el tiempo de ejecución y de esta forma evitar distintos tiempos de ejecución en función de la instancia de entrada para las heurísticas de AGM y AMC.

3.3. Tabú search

En esta sección analizaremos diferentes partes del algoritmos de Tabu Search que pueden ser modificados mediantes parámetros y trataremos de ajustar los mismos para tratar de conocer aquellos que otorgan mejores soluciones para la mayoría de instancias. Para todos los experimentos de esta sección generamos 50 grafos aleatorios de 100 vértices, con pesos de arista entre 1 y 30. Generamos estas instancias propias para poder fijar la mayor cantidad de variables posibles al momento de experimentar y así observar más claramente el efecto de cada parámetro que variamos. Con esta misma motivación decidimos utilizar AGM como heurística para obtener la solución inicial de Tabú Search, ya que para las mismas instancias devuelve resultados muy similares en varias corridas.

Los gráficos de esta sección muestran, para cada parámetro variado, el promedio de los porcentajes de mejora de la solución de Tabú Search respecto a la solución inicial recibida para las 5 corridas de las 50 instancias mencionadas anteriormente. Como en el gráfico estamos trabajando con el promedio de promedios (promediamos los 5 promedios de las calidades de las instancias) decidimos graficar el *error estándar*, que se define como:

$$error\,estándar(x) = \frac{desvío\,estándar(x)}{\sqrt{longitud(x)}}.$$

Y nos permite observar la precisión del promedio de la muestra.[6]

[6]https://towardsdatascience.com/standard-deviation-vs-standard-error-5210e3bc9c04

Por último, es importante notar que como fijamos todos los parámetros que no variamos, los selección de valores para aquellos con los que no experimentamos al momento de realizar alguno de los experimentos (cantidad de iteraciones y tamaño de la Lista Tabú para porcentaje de vecinos, cantidad iteraciones para tamaño de la Lista Tabú) puede parecer arbitraria. Para elegirlos realizamos una corrida preliminar previa a la que exponemos en este informe y seleccionamos los que nos resultaron óptimos de ahí.

3.3.1. Proporción de vecindad explorada

La metaheurística de Búsqueda Tabú parte de una solución y la mejora a partir de explorar la vecindad de la solución que es considerada como actual. No se considera toda la vecindad de la solución ya que esto podría ser muy costoso temporalmente, por este motivo se toma una proporción de vecinos a considerar, éstos se toman de forma aleatoria. Queremos entonces determinar cuál es el impacto en la calidad de la solución de la proporción de la vecindad considerada y si existe un intervalo de valores que resultan óptimos.

Para realizar el experimento fijamos el criterio de parada tal que no se realicen más de 120 iteraciones y guardamos una lista tabú de tamaño igual a 40. La figura 13 muestra los resultados obtenidos para la Búsqueda Tabú que guarda las aristas y la figura 14 los resultados al guardar las últimas soluciones. Fijamos 120 iteraciones y tamaño 40 de lista tabú.

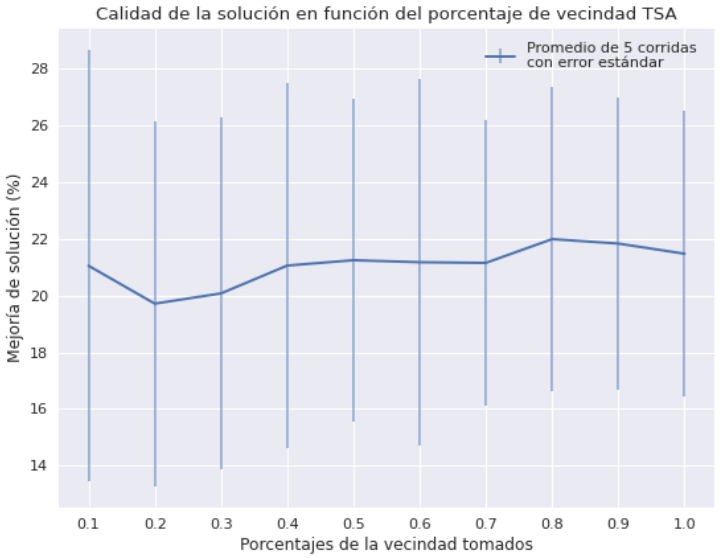

Figura 13: Variación de la proporción de vecindad tomada TSA

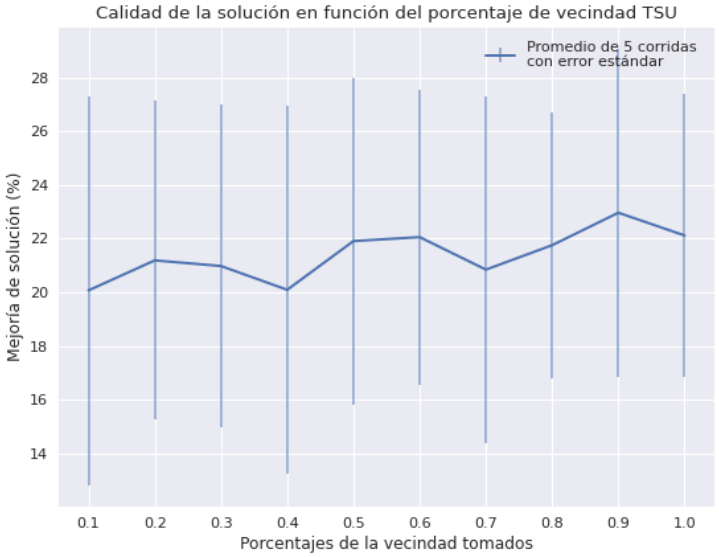

Figura 14: Variación de la proporción de vecindad tomada TSU

Podemos ver en ambos casos que la proporción de la solución óptima no varía significativamente al considerar distintas proporciones de la vecindad; en el caso de TSU mejora un 3 % y en el de TSA un 2 % de mejoría. Entonces, explorar una mayor cantidad de vecinos para los parámetros de Búsqueda tabú especificados no resulta en grandes mejoras de la calidad de la solución.

3.3.2. Tamaño de la lista tabú

Primero supeditamos bajo análisis la implementación de TabuSearch para aristas.

Analizamos que parámetro resulta conveniente para configurar el tamaño de la lista tabú. Pensamos que esto puede influir en las soluciones otorgadas por el algoritmo debido a que el propósito de la lista es, dado una solución encontrada en la búsqueda, si esta se encuentra en la memoria, entonces no se la vuelve a procesar. Cambiando el tamaño de soluciones recordadas debería tener un impacto por la naturaleza del mismo.

Para tratar de encontrar un parámetro deseable, corrimos el algoritmo sobre un conjunto de 50 instancias de mismo tamaños generadas aleatoriamente. Para analizar el rendimiento a medida que el parámetro cambia, elegimos tomar el porcentaje de la diferencia entre la solución obtenida y la inicial. Se procedió a ejecutar el algoritmo sobre cada configuración de la lista para cada grafo y tomar el promedio de cada una.

Fijamos 120 iteraciones y porcentaje de vecindad 0.8.

En la figura siguiente se muestran los resultados obtenidos.

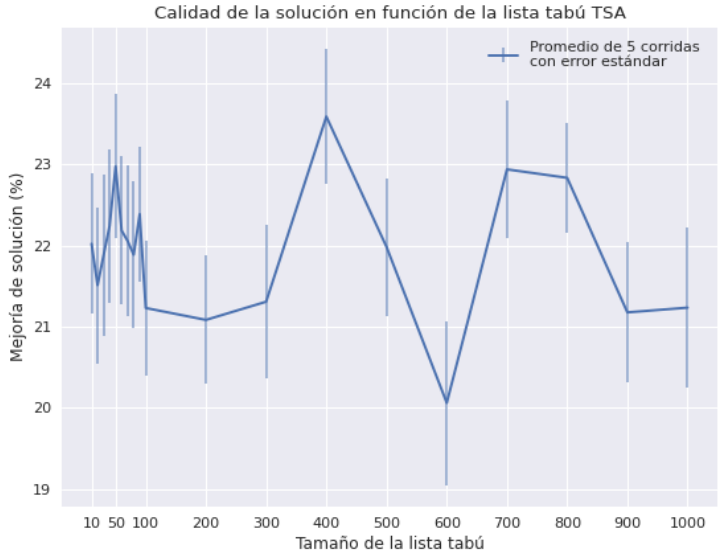

Figura 15: Variación del tamaño de la lista tabu TSA

En el gráfico se aprecia que la linea azul mas gruesa representa el promedio de porcentajes de la solución. Para cada parámetro, se muestran barras verticales que indican el error estándar que se produjo para cada promedio. La siguiente imagen muestra con mas detalle acerca del intervalo de tamaño entre 10 y 100 para el parámetro de la lista.

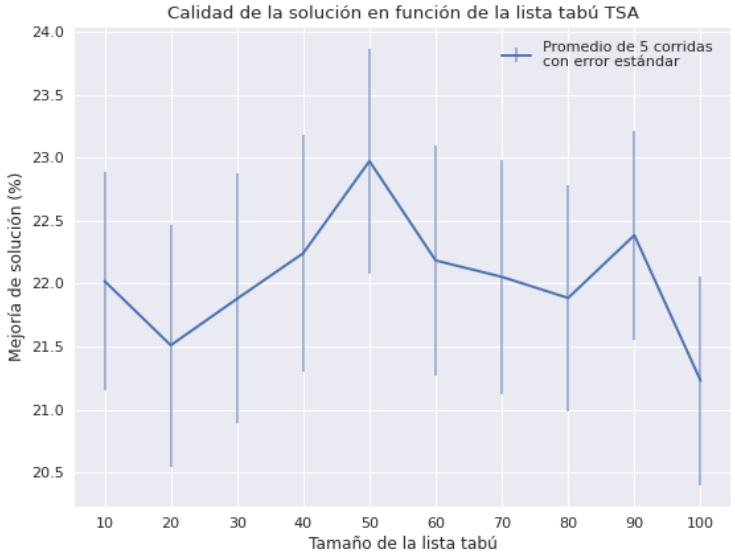

Figura 16: Variación del tamaño de la lista tabú TSU

De la información que obtuvimos a partir de la experimentación, podemos tener las siguiente conclusiones: La mejora de soluciones con respecto a la original esta en un rango entre el 20 y 24 por ciento de mejora. Esto nos dice que el tamaño de la lista tabú tiene un impacto del 4 por ciento en la variabilidad de nuestro algoritmo, que no es muy significativa. El mejor resultado parece haber sido cuando se utiliza una memoria de longitud 400 que nos otorga la mejor solución. Sin embargo, se puede ver que el parámetro configurado con el valor 50 tiene una perfomance muy parecida a la anterior, que si bien es ligeramente inferior en su porcentaje, el hecho de que el tamaño sea bastante menor, consideramos que tiene mas ventajas por el hecho de que influye de mejor manera en la ejecución del algoritmo (ver sección de complejidad).

Por ultimo, exponemos los resultados del mismo experimento pero para TSU.

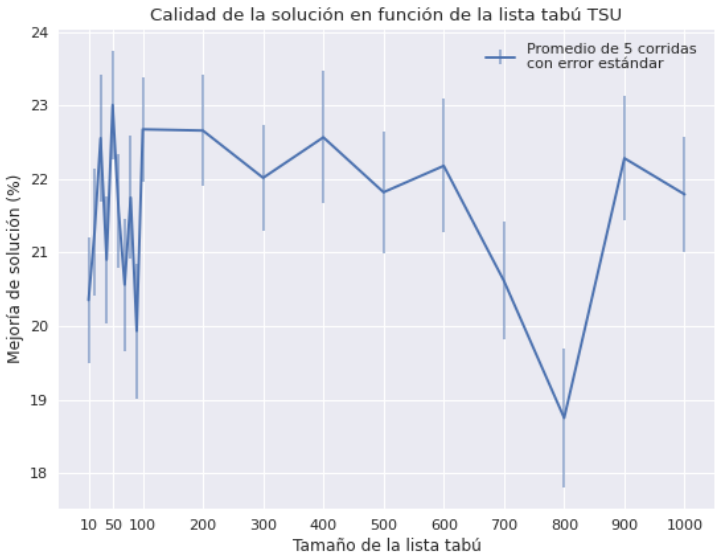

Figura 17: Variación del tamaño de la lista tabú TSU

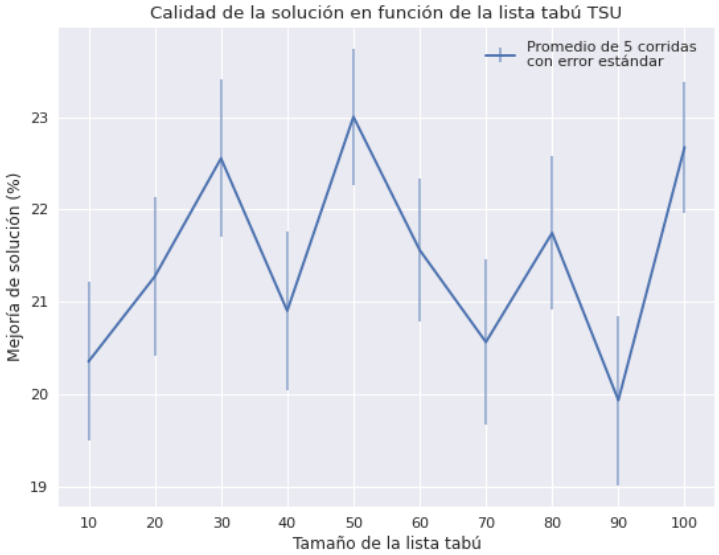

Figura 18: Variación del tamaño de la lista tabú en TSU agrandado

Estos gráficos muestran que las soluciones de esta implementación son ligeramente peores a la ofrecida por TSA. El rango de soluciones tiene una variación también del 4 por ciento pero su calidad es menor. Notar que el mejor parámetro también resulta 50 por lo tanto es igual al anterior método.

3.3.3. Cantidad de iteraciones

El criterio de parada utilizado para salir del ciclo que busca mejorar la solución actual fue fijar una cantidad máxima de iteraciones, una vez alcanzadas se devuelve la solución obtenida. Nos propusimos entonces experimentar con este valor para analizar cuánto mejora la solución al aumentar la cantidad de iteraciones. Fijamos tamaño de lista tabú 50 y porcentaje de vecindad 0.8, los gráficos de las figuras 22 y 21 muestran los resultados para los dos tipos de Búsqueda tabú.

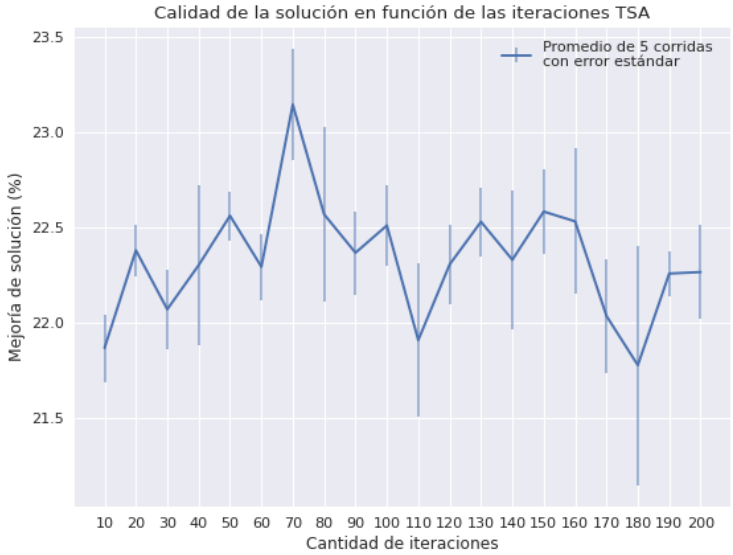

Figura 19: Variación de las iteraciones realizadas para TSA

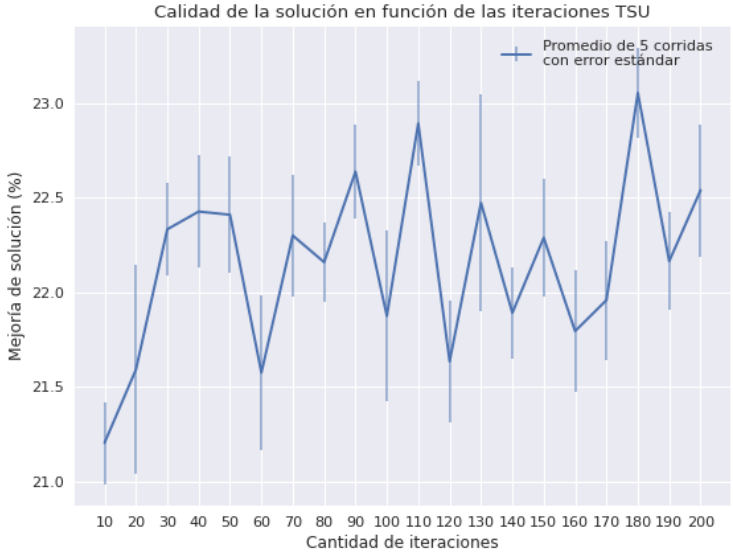

Figura 20: Variación de las iteraciones realizadas para TSU

En el caso de TSA, la mejoría de la solución varía entre 21 % y 24 % al aumentar la cantidad de iteraciones pero no parece seguir un aumento estable. Es decir, no se ve una relación entre la cantidad de iteraciones y la calidad de la solución. Podemos deducir entonces que la heurística obtiene la solución que se retornará en las primeras iteraciones del algoritmo.

En el caso de TSU tampoco se registraron modificaciones considerables en la calidad de los resultados obtenidos. Sin embargo, se puede notar una tendencia de aumento de la calidad de las soluciones al crecer la cantidad de iteraciones. Esto podría indicar que al guardar las últimas soluciones hay una mayor posibilidad de aumentar la performance del método, a diferencia de TSA que parece estancarse luego de una cierta cantidad de iteraciones.

3.4. Comparación de los métodos

En esta sección procedemos a poner a prueba los mejores parámetros obtenidos en los experimentos anteriores en un conjunto de nuevas instancias. En lugar de ser generadas aleatoriamente, se tomaran grafos bajos situaciones reales. Serán utilizadas las siguientes instancias: http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/index.html para realizar los experimentos. Una particularidad interesantes de estos grafos es que se conocen sus soluciones óptimos, con lo cual, se puede comparar las soluciones obtenidas por nuestros algoritmos con aquellas para poder realizar una análisis de perfomance. Las soluciones óptimas a las instancias pueden ser vistas en el siguiente link: enhttp://elib.zib.de/pub/mp-testdata/tsp/tsplib/stsp-sol.html..

3.4.1. Calidad de la solución

Los experimentos serán corridos sobre una subconjunto de 75 casos bajo la siguiente configuración: Para TSA, se utilizara una longitud de lista tabú de 50, el 80 por ciento de la vecindad y 70 iteraciones como máximo. Para TSU, 0.9 tamaño de vecindad, 50 lista tabú, 180 iteraciones

Los resultados están condensados en los siguiente gráfico:

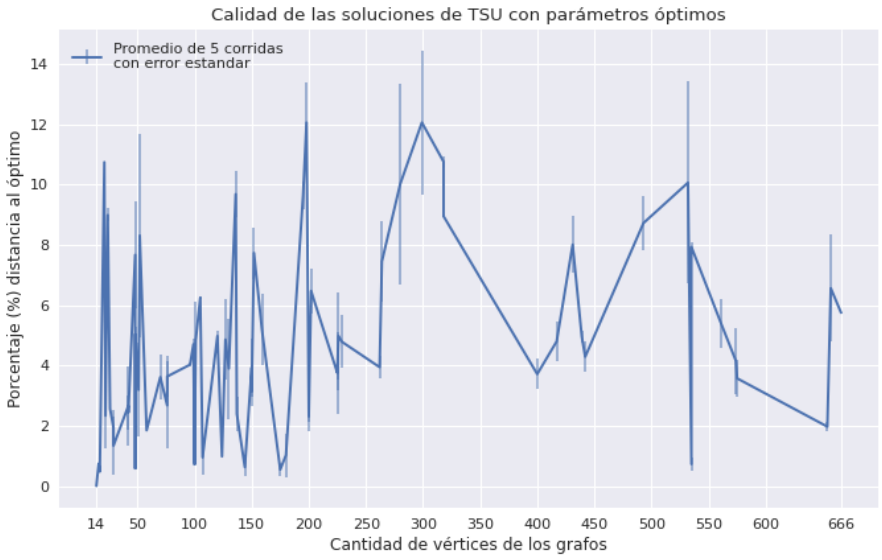

Figura 21: Comparación entre distintos porcentajes de distancia al óptimo con respecto a la cantidad de vértices

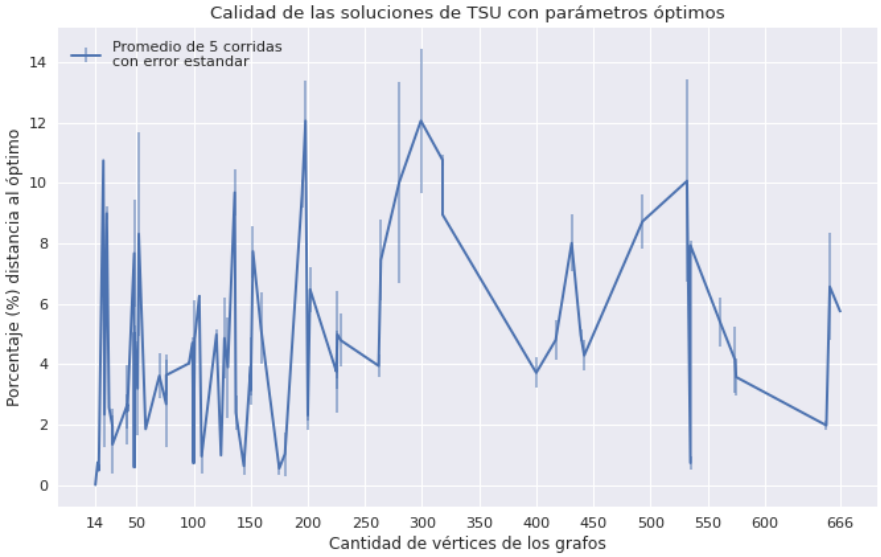

Figura 22: Comparación entre distintos porcentajes de distancia al óptimo con respecto a la cantidad de vértices

Se puede apreciar que tanto TSU, como TSA tienen un rendimiento casi igual, con lo cual no hay mucha diferencias entre ellos. Otro análisis determina que el algoritmo bajo estas configuraciones ofrece soluciones bastante próximas a la óptima. En el peor caso la diferencia radica en un 12 por ciento, pero la mayoría se encuentra en porcentajes bastante menores. También se puede notar que nuestros algoritmos es bastante sensible a la naturaleza del grafo y no depende del tamaño del mismo.

3.4.2. Tiempo de ejecución

El último experimento que presentaremos estudia la relación del tiempo de ejecución de la metaheurística de búsqueda tabú en función del tamaño de las instancias pasadas como parámetro. Esperamos que los resultados muestren una relación entre estos valores ya que la cantidad de operaciones realizadas por la búsqueda tabú depende de la cantidad de vértices contenidos en el grafo.

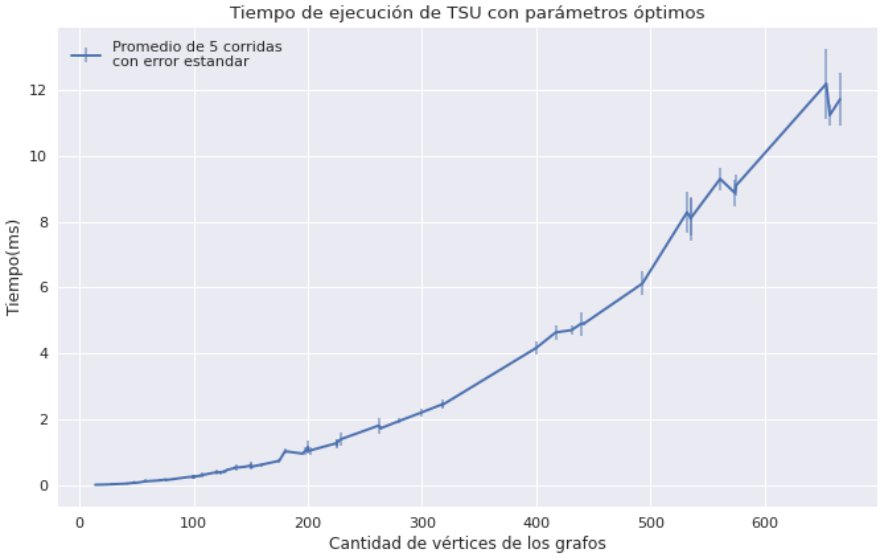

Figura 23: Tiempo de ejecución de TSU en función del tamaño

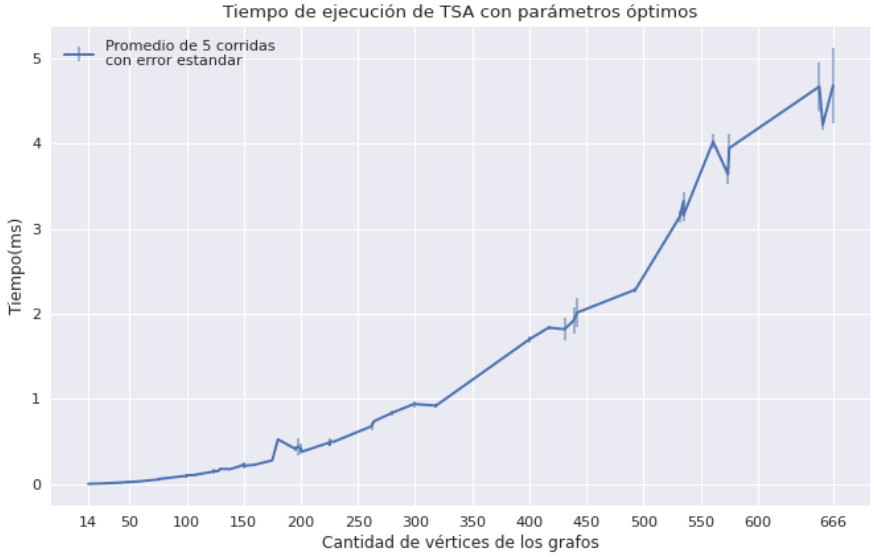

Figura 24: Tiempo de ejecución de TSA en función del tamaño

Analizamos las instancias mencionadas cuyos tamaños están entre 0 y 667 vértices y obtuvimos los resultados que se observan el las figuras 23 y 24. Como supusimos, se puede ver una fuerte relación entre el tamaño de una instancia y el tiempo de ejecución del método tanto para TSU como para TSA.

4. Conclusiones

A lo largo del trabajo implementamos heurísticas y metaheurísticas que proporcionan soluciones aproximadas para el problema de TSP. Realizamos una experimentación con el objetivo de analizar la performance de estos procedimientos en función de las características de las instancias pasadas como parámetro y de los parámetros del método.

En el caso de la heurística de Vecino Más Cercano (Nearest Neighbour) y de la Arista Más Corta pudimos observar que existe un tipo de instancias que representa el peor caso al evaluar la calidad de la solución. Estas instancias hacen que la solución obtenida sea arbitrariamente mala. Sin embargo se puede reducir el impacto de estos casos aleatorizando algunos aspectos del método y tomando promedio de varias corridas para una misma instancia. El tiempo de ejecución de ambas heurísticas no depende del tipo de instancia evaluada sino únicamente del tamaño de la misma. Aunque vimos que distintas implementaciones de las heurísticas pueden hacer que esto último no siempre sea cierto, por ejemplo la elección del algoritmo de ordenamiento utilizado en Kruskal tiene un fuerte impacto en el tiempo de ejecución de la heurística.

En cuanto a la heurística basada en Árbol Generador Mínimo, podríamos concluir que es un método bastante seguro en cuanto a la calidad de sus soluciones gracias a su 1-aproximación, aunque la condición necesaria para poder asumir esto no siempre se cumple. Hay una fuerte dependencia en que las distancias entre los vértices sean euclidianas, por lo que podría ser muy fácil descartar esta heurística ya que esto no ocurre en varios casos de la vida cotidiana. Por ejemplo, las rutas entre ciudades no necesariamente describen distancias euclidianas.

Es importante analizar el hecho que Tabú Search ofreció una mejora bastante importante con respecto a los otras heurísticas. Se pudo apreciar en los experimentos que el algoritmo mejoró alrededor del 20 por ciento las soluciones con respecto a las heurísticas anteriores, otorgando una mejora pronunciada. A pesar de esta gran mejora, tiene el inconveniente que el tiempo de ejecución se vuelve bastante mayor. Esto nos marca la pauta que Tabú Search es un algoritmo fuertemente recomendable si se busca tener una solución bastante aproximada a la óptima, siempre que el tiempo de ejecución no sea limitante. En cambio, si el aspecto en materia de tiempo es critico y la solución aproximada no es tan importante, AGM y AMC resultan heurísticas mejores para esos casos ya que ofrecen tiempos de ejecución bajos y ofrecen soluciones en la mayoría de casos razonables. Además, no observamos diferencias significativas entre Tabú Search Últimas Soluciones y Tabú Search Aristas.

Por último, si bien variando los parámetros y utilizando los parámetros óptimos para las instancias de entrenamiento obtuvimos mejoras mayores respecto a las soluciones de las que parte Tabú Search, estás mejoras resultaron ser relativamente poco significativas (4 % respecto a la solución inicial). Esto indicaría que Tabú Search es una metaheurística que brinda buena calidad de soluciones por diseño, sin necesitar ajustar sus parámetros a las características de las entradas procesadas.

Podríamos considerar algunos conceptos para trabajar en algún futuro, el uso de una función de aspiración en TSU entre ellos. En particular, se podrían tolerar soluciones de hasta un 5 % peor.