DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Travelling Salesman Problem

Algoritmos y estructuras de datos III

Integrante	LU	Correo electrónico
Hayon, Gabriel David	701/17	gabrielhayonort@gmail.com
Lopes Perera, Pablo	007/18	plopesperera@dc.uba.ar
Morales, Matias	504/18	99moralesmatias@gmail.com
Rosinov, Gaston Einan	037/18	grosinov@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

Resumen

Este informe presenta resultados del análisis de algoritmos heurísticos y meta-heurísticos para el problema del viajante de comercio. Se experimentó con dos heurísticas constructivas golosas, árbol generador mínimo, y con la meta-heurística tabú search, implementando una memoria por caracterizaciones y otra basada en estructura. Se vió como los tiempos de ejecución de las heurísticas se ajustan a sus cotas temporales calculadas. También se vió como y cuanto tabú search optimiza soluciones iniciales y cual es la configuración óptima (de forma empírica), haciendo válidas las hipótesis presentadas.

1. Introducción

1.1. Travelling Salesman Problem

El problema del viajante de comercio(TSP) está basado en la siguiente pregunta, *"Dada una lista de ciudades con las distancias entre ellas, cual es el camino mínimo de forma tal de visitar todas las ciudades solo una vez, y volver al inicio?"*. En términos de teoría de grafos, esto representa la búsqueda del circuito Hamiltoniano de menor peso con respecto a sus aristas. Notar que en este informe, contaremos con grafos completos. Utilizamos grafos completos, ya que de esta forma podemos garantizar que tiene un ciclo Hamiltoniano.

Definición formal: Dado un grafo $G = (V, E)$ con pesos (positivos) asignados a todas las aristas de E, determinar un circuito Hamiltoniano de peso mínimo. Teniendo la función longitud $l : E \to \mathbb{R}^{>0}$, hallar C^0 tal que

$$l(C^0) = \text{mín}(l(C) \,|\, C \text{ es un circuito Hamiltoniano de } G)$$

La función l también la podemos definir para que tome un grafo y devuelva un camino, $l : G = (V, E) \to \mathbb{R}^{>0}$. A lo largo del informe se podrán usar ambas de forma intercambiable.

Este problema esta clasificado (en términos de complejidad computacional) como NP-hard, por lo tanto no se conocen algoritmos polinomiales para su resolución óptima. Lo que se puede realizar son aproximaciones, utilizando algoritmos heurísticos o meta-heurísticos, los cuales no necesariamente brindan la solución óptima, pero con ellos se pueden obtener buenos resultados. Estos serán enunciados y analizados mas adelante.

1.2. Aplicaciones en el mundo real del TSP

TSP se puede emplear en cualquier situación que requiere seleccionar nodos en cierto orden que reduzca los costos, como por ejemplo:

- Reparto de productos: Mejorar una ruta de entrega para seguir la más corta.
- Transporte: Mejorar el recorrido de caminos buscando la menor longitud posible.
- Robótica: Resolver problemas de fabricación para minimizar el número de desplazamientos al realizar una serie de perforaciones en un circuito impreso.
- Turismo y agencias de viajes. Aun cuando los agentes de viajes no tienen un conocimiento explícito del Problema del Agente Viajero, las compañías dedicadas a este giro utilizan un software que hace todo el trabajo. Estos paquetes son capaces de resolver instancias pequeñas del TSP. Horarios de transportes laborales y/o escolares: Estandarizar los horarios de los transportes es claramente una de sus aplicaciones, tanto que existen empresas que se especializan en ayudar a las escuelas a programarlos para optimizarlos en base a una solución del TSP.

- Inspecciones a sitios remotos: Ordenar los lugares que deberá visitar un inspector en el menor tiempo.
- Secuencias: Se refiere al orden en el cual n trabajos tienen que ser procesados de tal forma que se minimice el costo total de producción.

2. Técnicas de resolución

2.1. Heurísticas constructivas golosas

Las heurísticas constructivas son métodos que construyen una solución factible de una instancia del problema. Por simpleza en su implementación, se suelen utilizar procedimientos golosos, si bien estas no aseguran soluciones óptimas, podemos obtener soluciones aceptables, dejando de lado tanta complejidad algorítmica.

2.1.1. Heurística del vecino más cercano

Esta heurística consiste en comenzar con un vértice, e ir avanzando por aquellos cuyas aristas tengan menor peso. En cada paso elegimos a cual avanzar, de los cuales no fueron previamente visitados. Cuando se visitan todos los vértices concluye el proceso. Evidentemente este es un proceso goloso.

Algorithm 1 Procedimiento del Vecino más Cercano

```
function nearestNeighbour(G)
    Input: G = (V, E) de n vértices
    Output: H circuito Hamiltoniano
    v ← algún vértice
    verticesVisitados ← [v]
    H ← [v]
    while |H| ≤ n do                                   // O(n)
        w ← masCercano(G, v, verticesVisitados)        // O(n)
        verticesVisitados ← verticesVisitados + w
        H ← H + w
        v ← w
    return H
Orden de Complejidad: O(n^2)
```

Algorithm 2 Función auxiliar para el Vecino más Cercano

```
function masCercano(G, v, verticesVisitados)
    Input: G = (V, E), v vértice, verticesVisitados lista
    Output: w vértice mas cercano a v que no fue visitado
    w ← vértice vecino de v cualquiera
    for t vecino de v do                               // O(n)
        if t no fue visitado y esta mas cerca a v que w then
            w ← t
    return w
Orden de Complejidad: O(n)
```

2.1.2. Heurística de la arista más corta

Para esta heurística podemos modificar el algoritmo de Kruskal, para que devuelva un circuito Hamiltoniano. De esta forma tenemos que ser precavidos para no formar ciclos, en la solución que armamos con el mismo.

Agregamos la arista de menor peso entre todas las aristas, cumpliendo que no genere ciclos ni vértices de grado mayor a 3.

Utilizaremos una estructura *UnionFind*, que nos permite manejar conjuntos disjuntos de elementos. Cada conjunto tiene un representante que lo identifica. Esto nos ayuda a disminuir la complejidad de nuestro algoritmo.

Utilizaremos un grafo implementado con listas de incidencia. Esto facilita la escritura del algoritmo ya que nos combine iterar por las aristas. Para ello ordenamos el grafo por los pesos de dichas aristas

Algorithm 3 Procedimiento de la Arista mas corta

```
function shortestEdge(G)
    Input: G = (V, E)
    Output: H circuito Hamiltoniano
    uf ← UnionFind de tamaño n                                   // O(n)
    grados ← lista de tamaño n con valores en 0                  // O(n)
    sort(aristas de G)                                           // O(n · log(n))
    H un camino Hamiltoniano vació
    for e arista de G.aristas do                                 // O(m)
        if los vértices que une la arista e están en componentes distintas then  //
O(m · log(n))
            if los vértices que une la arista e tienen grado menor o igual a 1 then
                H ← H + la arista (y sus nodos) e
                grados[e_v] ← grados[e_v] + 1
                grados[e_w] ← grados[e_w] + 1
                uf.union(e_v, e_w)                               // O(n)
    H ← H + unir el ultimo vértice con el primero
    return H
Orden de Complejidad: O(m² · log(n))
```

2.2. Heurística basada en Árbol Generador Mínimo

En este algoritmo creamos el árbol generador mínimo de G, luego recorrer el árbol con DFS. Con el orden que devuelve DFS, podemos recrear el camino Hamiltoniano. Luego con unir el ultimo con el primer vértice obtendremos un ciclo Hamiltoniano.

El árbol generador mínimo se genera con el algoritmo de Kruskal (1; 2).

Algorithm 4 Procedimiento con el Árbol Generador Mínimo

function $heurisitcaArbolGeneradorMinimo(G)$
Input: $G = (V, E)$
Output: H circuito Hamiltoniano
$T \leftarrow$ Kruskal(G) // $O(m \cdot log(n))$
$order \leftarrow$ DFS(T) // $O(n^2)$
H camino vacío
for (v, w) vértices en $order$ **do** // $O(T)$
Notar que se van tomando vértices de a pares
$H \leftarrow H +$ arista (v, w)
$H \leftarrow H +$ uno el ultimo vértice con el primer vértice
return H
Orden de Complejidad: $O(m \cdot log(n) + n^2)$. Siendo $O(T) = O(\#Aristas(T)) = O(n-1) = O(n)$

Algorithm 5 Algoritmo de Kruskal

function KRUSKAL(G)
Input: $G = (V, E)$
Output: T árbol generador mínimo de G
$uf \leftarrow$ UnionFind de tamaño n
sort(aristas de G) // $O(m \cdot log(m))$
for e arista de G.aristas **do** // $O(m)$
if los vértices que une e están en componentes distintas **then**
uno las componentes de e en la estructura uf // $O(log(n))$
return T
Orden de Complejidad: $O(m \cdot log(n))$

2.3. Problemas de las heurísticas

Las heurísticas utilizadas en este informe tienen una familia de instancias que pueden hacer que el óptimo se aleje del resultado infinitamente.
La siguiente familia cumple lo dicho anteriormente para las heurísticas de NN, SE y AGM: Sea G un grafo completo, y T el árbol formado por la heurística, llamaremos v_1 al nodo inicial y v_{n-1} al último nodo del árbol. Tomando la arista e tal que une a los nodos v_1 y v_{n-1}, con el peso que se desea, se consigue así alejar el resultado de la heurística del resultado óptimo infinitamente.

2.4. Metaheurísticas

Las metaheurísticas son métodos generales para la resolución de problemas. Estas no están limitadas a un problema en especifico, sino que al ser un método, se pueden aplicar a una diversa cantidad de problemas.

Para la resolución del problema del *TSP* optamos por utilizar una metaheurística llamada **Tabu Search** (Búsqueda Tabú). La búsqueda tabú implementa un algoritmo que utiliza una búsqueda local para buscar soluciones vecinas a una dada solución. Esto se puede pensar como moverse o realizar perturbaciones dentro de una solución para avanzar a otros óptimos

locales, para no quedar atascado en la misma rama de soluciones. También podemos pensar esto como un proceso de optimización en un espacio, ya que estamos buscando optimizar la función $l : E \to \mathbb{R}^{>0}$ para nuestro circuito Hamiltoniano.

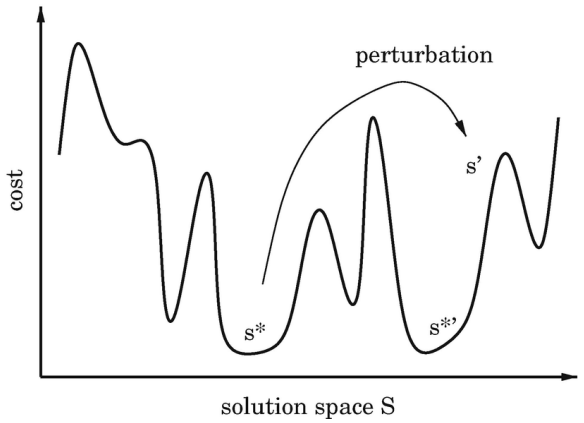


Figura 1: Ejemplo de función donde se aplica local search

Denominamos $N(s)$ la vecindad de la solución s. Cada solución $s' \in N(s)$ es alcanzada desde s realizando estas perturbaciones o movimientos.

Tabu Search restringe al local search a un subconjunto $V' \subset N(s)$ con $|V'| \ll |N(s)|$ de forma estratégica tal que utiliza de forma sistemática una memoria para explotar el conocimiento adquirido de dicha función $f(s)$ para la vecindad $N(s)$.

Con Tabu Search limitamos la creación de ciclos (al momento del descenso) ya que con la memoria evitamos visitar soluciones de mas. Pero en el caso de habernos pasado por una solución que a primera vista no es mejor que las actuales, aplicamos una función llamada *función de aspiración*. Esta función nos permite revisar soluciones que son consideradas tabú. Esta función se puede activar cuando toda la vecindad de una solución es tabú, eso produciría un ciclo, y con esta función se podría llegar a salir de ese mínimo.

A continuación se encuentra el esquema general del algoritmo de tabu search.

Algorithm 6 Esquema general del algoritmo de Tabu Search.

function TABUSEARCH($\cdots$)
  Input: $\cdots$
  Output: H camino Hamiltoniano solución
  $s \leftarrow$ solución inicial encontrada con alguna Heurística // $O(h(G))$
  $s' \leftarrow$ s
  inicializo la memoria Tabu
  inicializo la función de aspiración
  while No valga T (condicion de terminacion) **do** // $O(T)$
    $V' \subset N(s)$ // $O(n^2)$
    elijo $s^* \in V' \,/\, f(s^*) < f(\hat{s}) \forall \hat{s} \in V'$ // $O(V')$
    actualizo la memoria // $O(1)$
    actualizo la función de aspiración // $O(1)$
    if $f(s^*) < f(s')$ **then**
      $s' \leftarrow s^*$ // $O(1)$
  return s^*
Orden de Complejidad: $O(T \cdot (n^2 + V'))$

En dicho algoritmo utilizamos dos implementaciones distintas de memoria. Una que guarda caracterizaciones de las soluciones, y la otra almacena en estructuras previamente visitadas.

Sobre criterios de parada, eso depende plenamente de la implementación. Algunos ejemplos serian: cantidad de iteraciones, tiempo, cantidad de iteraciones sin mejora, etc.

Un tema importante al momento de implementar tabu search, es la forma de búsqueda de la vecindad de s. Para dicho algoritmo, en nuestra implementación, hemos creado un buscador de vecindad. Esta se basa en buscar una arista aleatoria encontrada en s, y a esa realizarle swaps[1] con todas las otras aristas (con las que sea posible, es decir, que siga siendo una solución valida luego de aplicar el swap). De las soluciones vecinas creadas, nos quedamos con las k mejores. El numero k es un parámetro del algoritmo de Tabu Search. Una vez que obtenemos la vecindad con los k mejores, se selecciona uno al azar, y se verifica que este no pertenezca a la memoria. El efecto de la aleatoriedad nos quita el sentido deterministico del algoritmo, al introducir incertidumbre, hay casos en lo que nos permite retroceder para volver a avanzar (Mejor apreciable en la Figura 1).

Si el algoritmo se queda estancado en un mínimo local durante varias iteraciones y al buscar una nueva solución esta se encuentra en la memoria (o se encuentra que el swap realizado), se aplica la función de aspiración cuya función es que en vez de descartar esta solución se aplica para buscar en una vecindad distinta y poder salir de ese mínimo local.

En la implementación de Tabú Search realizada, se han tomado distintos parámetros para la ejecución del algoritmo, de forma tal que realicen condiciones para distintas partes del algoritmo. Estos parámetros son

[1] Definimos a un **swap** como intercambio de vértices de forma tal que se mantenga la cantidad de componentes conexas y las aristas que unan a esos vértices hayan cambiado

- **MEM_SIZE** este es el parámetro que modifica el tamaño de la memoria. Esto altera a la cantidad de soluciones que guarda el algoritmo en la memoria Tabú. La hipótesis ideada en dicho parámetro es que el algoritmo debería tener una buena performance (e n términos de costo de la solución) con memorias de tamaño mediano. Es decir, memorias muy pequeñas no almacenarían una cantidad considerable de soluciones (ya sean características o por estructura) para que el algoritmo optimice los resultados.
- **ASPIRATION_STALL** este parámetro es el que realiza la activación de la función de activación. Dentro de la implementación, se encuentra un contador el cual guarda la cantidad de iteraciones que ocurren sin que el algoritmo mejore una solución. Este parámetro se utiliza como control contra ese valor. Cuando el valor de *stall* supera el valor del parámetro, se procede con la activación de la función de aspiración. Mientras mas grande sea el valor del parámetro, mas tiempo le estamos dando al algoritmo para que encuentre otras soluciones, de forma contratia, mientras mas chico es el valor estamos siendo estrictos sobre la "velocidad.[a] la cual encuentra otras soluciones.
- **MAX_ITERATIONS** es la cantidad máxima de iteraciones del algoritmo.
- **MAX_VICINITY_SIZE** es el tamaño máximo de la vecindad. Este parámetro fue creado para limitar la cantidad de sub-soluciones factibles que se devuelven de vecindad de una solución. En la implementación, V es un subconjunto de mejores sub-soluciones de $N(s)$, cuyo tamaño es el valor del parámetro. A medida que este valor vaya incrementando, estamos dando lugar al análisis mas "mejores sub-soluciones"de la vecindad. Esto es importante, ya que la complejidad esta ligada con dicha variable, por ende a mayor valor, mayor tiempo de ejecución. Aquí se hipotetizamos de forma similar al parámetro MEM_SIZE, diciendo que no necesitamos un tamaño de V muy grande. Esto es análogo con valores demasiado chicos, mientras mas chico sea el parámetro menor flexibilidad nos dará.
- **TERMINATION_CONDITION** cuenta la cantidad de veces que fue activada la función de aspiración, y el resultado no obtuvo una solución mejor que la óptima hasta ese momento. Este parámetro indica cuando el algoritmo queda trabado sin poder mejorar luego activar la función de aspiración.

2.4.1. Tabú Search: Memoria basada en ultimas soluciones exploradas (caracterizaciones)

Con el tipo de memoria que almacena las ultimas soluciones exploradas/caracterizaciones lo que guarda, propiamente dicho, son las caracterizaciones de las soluciones que visito en el local search. Estas caracterizaciones pueden ser tales como el peso total de s', etc.

2.4.2. Tabú Search: Memoria basada en estructura (aristas)

Este tipo de memoria guarda características estructurales de las soluciones como, en nuestro caso, el swap realizado o cualquier cambio estructural que realiza el algoritmo en cuestión.

3. Experimentación

En esta sección evaluaremos la efectividad y la performance de los algoritmos detallados, con su implementación en un lenguaje de programación. El objetivo principal es analizar las

cotas temporales dadas y (en aquellos que se conozca) la relación entre el resultado óptimo de un test y el resultado obtenido por los algoritmos.

3.1. Detalles Experimentales

Los experimentos realizados fueron corridos en una computadora que cuenta con un CPU Intel Core i7@4.50Ghz y 32gb de memoria RAM. Los algoritmos fueron implementados en `C++ 14`.

3.2. Casos de test

En este informe se realizo la experimentación con distintos grafos (completos), de los cuales se conoce el resultado óptimo de una gran parte de ellos (3; 4; 5). Las instancias con óptimo conocido instancias fueron obtenidas en este sitio web[2], el cual contiene una colección de grafos. El resto de la instancias fueron creadas por un script de forma random.

Para la experimentación de las heurísticas se han utilizado 15 grafos distintos, de los cuales 7 se conocen el óptimo.

En la experimentación de Tabú Search solo se seleccionaron 5 grafos distintos de la colección. Esta decisión fue tomada ya que el algoritmo de Tabú Search se vuelve muy pesado en términos temporales y para hacer mas eficiente el proceso de análisis.

Para tabú search se definieron como *parámetros estándar* una memoria implementada sobre características y los parámetros como:

- **MEM_SIZE** = 300
- **ASPIRATION_STALL** = 150
- **MAX_ITERATIONS** = 3500
- **MAX_VICINITY_SIZE** = 200
- **TERMINATION_CONDITION** = 300

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

3.3. Resultados Iniciales

Aquí se detallaran los resultados obtenidos para la ejecución de las heurísticas, luego se procederá al análisis de las mismas. Notar los casos donde el prefijo es 'r' o 'c', son casos de test random (creados con un script) o creado manualmente, respectivamente

Test	Óptimo	AGM		NN		SE	
		Resultado	Tiempo	Resultado	Tiempo	Resultado	Tiempo
c4	95	95	0.038	95	0.010	95	0.014
c6	14	16	0.040	14	0.017	15	0.012
r10	-	166	0.067	168	0.031	169	0.019
att48	33524	45740	0.842	40503	0.447	40138	0.280
berlin52	7542	10713	1.326	8962	0.534	9937	0.324
gr96	512	665	2.977	661	2.489	590	1.009
r100	-	3174	3.451	3411	1.929	3328	1.236
eil101	629	849	3.277	810	1.979	760	1.134
pr152	73682	90329	7.608	85640	7.697	84643	2.979
r200	-	9189	13.297	9401	7.815	8830	6.989
a280	2579	3682	26.141	3425	20.300	3177	9.901
r300	-	16740	32.856	16930	19.489	15675	12.129
r350	-	20431	41.908	20523	36.302	19088	17.821
r400	-	21319	53.839	19355	33.562	17791	21.826
pr439	107217	149176	66.630	129916	38.672	128690	31.725

Cuadro 1: Resultados de la ejecución de cada test en cada heurística.

A continuación se mostraran dos tablas, la primera para los resultados obtenidos con el algoritmos de Tabu Search con memoria por características, y la segunda con memoria por estructuras.

Test	Opt	AGM					NN					SE				
		x_0	$\bar{x}$	$\bar{t}$	x_b	t_b	x_0	$\bar{x}$	$\bar{t}$	x_b	t_b	x_0	$\bar{x}$	$\bar{t}$	x_b	t_b
att48	33524	45740	35053	10.63	33809	9.50	40503	34764	11.39	34004	13.26	40138	35411	11.74	34324	15.79
berlin52	7542	10713	8173	18.01	7588	20.63	8962	8081	13.85	7781	15.40	9937	8222	17.42	7659	30.41
gr96	512	665	545	64.70	513	55.44	661	538	48.09	512	57.40	590	514	36.24	513	14.27
r100	-	3174	2647	85.87	2504	96.76	3411	2661	65.63	2492	68.45	3328	2661	66.60	2465	68.83
eil101	629	849	700	72.04	669	72.29	810	702	52.70	672	68.74	760	679	49.92	643	65.45
pr152	73682	90329	86030	166.94	80944	143.20	85640	79973	142.52	77979	142.28	84643	79587	143.79	77543	143.37

Cuadro 2: Resultados obtenidos para Tabu search con memoria basada en características

Test	Opt	AGM					NN					SE				
		x_0	$\bar{x}$	$\bar{t}$	x_b	t_b	x_0	$\bar{x}$	$\bar{t}$	x_b	t_b	x_0	$\bar{x}$	$\bar{t}$	x_b	t_b
att48	33524	45740	35104	15.18	33840	15.39	40503	34736	13.83	33880	13.64	40138	35436	13.02	34434	16.36
berlin52	7542	10713	8205	19.60	7607	18.38	8962	8055	14.86	7730	16.37	9937	8147	16.08	7631	12.32
gr96	512	652	534	63.14	513	62.87	661	517	63.07	514	63.38	553	536	62.85	520	62.63
r100	-	3174	2639	68.25	2461	68.22	3411	2638	67.64	2484	67.52	3311	2649	68.07	2480	67.96
eil101	629	837	684	69.48	662	69.66	810	685	68.34	664	68.71	760	661	69.38	641	69.41
pr152	73682	90329	86058	141.36	82001	141.58	85640	79787	141.22	75568	141.12	84778	79536	141.52	77065	141.93

Cuadro 3: Resultados obtenidos para Tabu search con memoria basada en estructuras

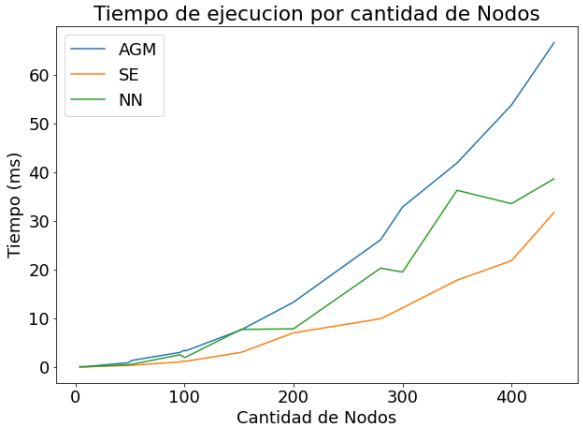
Nota, x_0 es la solucion inicial del Tabu search, basado en la heurisica (Columna), $\bar{(x)}$ es el promedio de resultados de la ejecucion del algoritmo de Tabu search, $\bar{t}$ es el promedio de tiempo de ejecucion, x_b es el mejor resultado obtenido en el algoritmo y t_b es el tiempo asociado a esa solucion.

Estos datos serán analizados en secciones posteriores.

3.4. Heurísticas: Shortest Edge vs Nearest Neighbour vs AGM

A partir de los datos obtenidos en la Tabla 1, fue calculado el promedio de los gaps relativos[3] de nuestros algoritmos. Los resultados obtenidos fueron

Como se puede ver en la tabla (Cuadro 1), los datos varían dependiendo el caso y el algoritmo. En la siguiente figura se puede apreciar el tiempo de ejecución de cada algoritmo para cada instancia

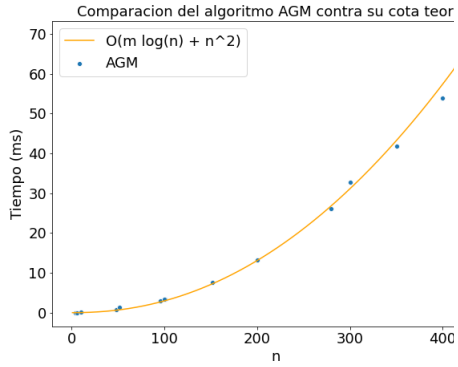


[3]El GAP relativo es el valor porcentual de cuan cerca se encuentra una solución sobre el resultado óptimo

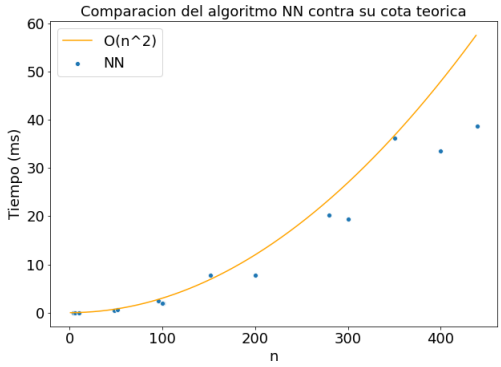
Algoritmo	Promedio del gap Relativo
AGM	0,27
NN	0,18
SE	0,16

Cuadro 4: Gap relativo por algoritmo

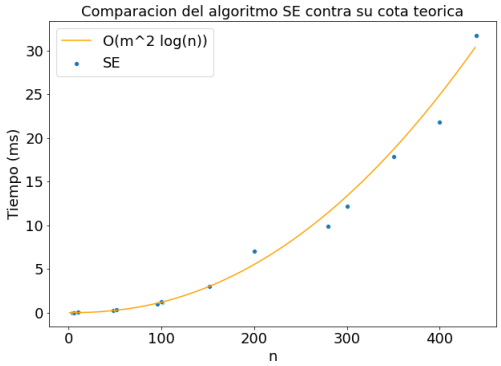
A continuación se mostraran 3 gráficos, los cuales representan los tiempos de ejecución de cada algoritmo con su cota temporal calculada de forma teórica para el peor caso. Como la cota fue calculada para el peor caso, no implica que en todas las instancias la hayan cumplido. Estos gráficos nos sirven para corroborar o verificar la correctitud de las cotas calculadas



(a) Comparación de la cota teórica de AGM con la experimentación



(b) Comparación de la cota teórica de NN con la experimentación



(c) Comparación de la cota teórica de SE con la experimentación

Figura 2: Gráficos obtenidos de la ejecución de nuestras heurísticas midiendo su tiempo de ejecución y comparándolos con su costo teórico

Por estos análisis, podemos afirmar que Shortest Edge tiene la mejor performance en nuestros experimentos , seguida de Nearest Neighbour y luego AGM. Esto también lo podemos confirmar calculando el coeficiente de Pearson, los cuales dieron $p_{agm} = 0{,}965$, $p_{nn} = 0{,}957$ y $p_{se} = 0{,}957$.

3.5. Heurísticas: Grafos problemáticos

Como se menciono en 2.3 (hacer referencia) hay una familia de grafos que permiten alejar la distancia del óptimo y el resultado. Los siguientes grafos son ejemplos de esta:

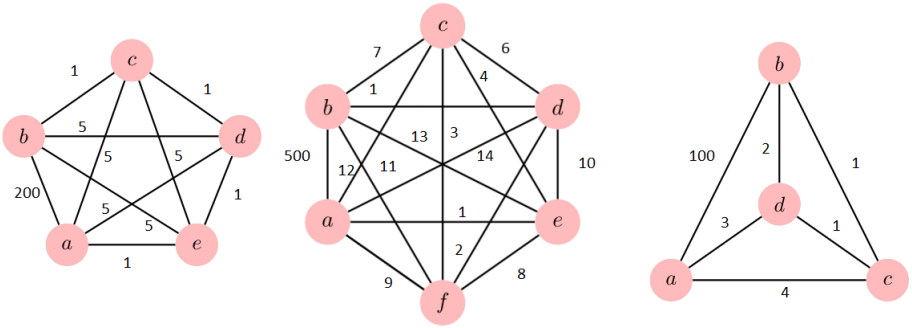


Figura 3: Ejemplos de grafos que nos permiten alejar la distancia entre nuestro resultado y el óptimo

Nodos	Algoritmo		
-	AGM	NN	SE
4	105	105	105
5	204	204	204
6	511	511	511

Cuadro 5: Pesos resultantes de correr las heurísticas con los grafos anteriores

En estos grafos y en sus resultados se puede ver que si nuestra arista inicial es b (para las heurísticas de NN y AGM), las heurísticas tienen una única opción para cerrar el árbol que construyen, la arista (a,b), teniendo así un control de la distancia entre el óptimo y el resultado de las heurísticas.

3.6. Tabú Search

3.6.1. Comparación de performance contra heurísticas

En esta sección vemos el grado de mejoría que tiene Tabú Search con respecto a las heurísticas comparándolas con los resultados de Tabú Search usando como solución inicial, la heurística SE. Para ello observamos el peso obtenido con respecto al tiempo para cada instancia, con un *conjunto estándar* de parámetros. Las instancias, son 15 instancias de 40 vértices tal que los pesos de los vértices son valores "random".

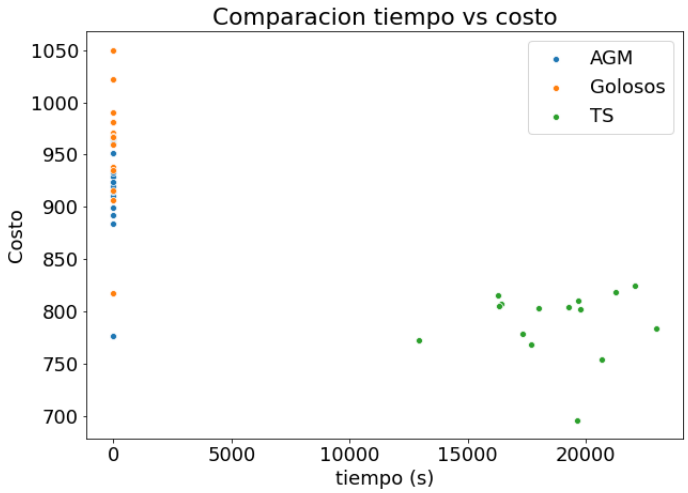


Figura 4: Comparación de tiempo vs costo entre AGM, golosos y Tabu search

La Figura muestra como evoluciona el costo de las soluciones a medida del paso del tiempo. Lo que se puede extraer, es que, en general, tabú search performa mejor , en términos de costo, que los algoritmos golosos y que la heurística de AGM. Esto

Resultados comparativos con el test *att48* Tomando el caso de test *att48* (Cuadro 1), podemos ver como el gap relativo mejora sustancialmente contra las heurísticas. Tomamos como comparación en TS el mejor costo devuelto.

- **AGM**: En este caso el gap relativo es de $0,364$, luego de aplicar TS al resultado inicial de la heurística tenemos que es $0,008$.
- **NN**: En este caso el gap relativo es de $0,208$, luego de aplicar TS al resultado inicial de la heurística tenemos que es $0,014$.
- **SE**: En este caso el gap relativo es de $0,197$, luego de aplicar TS al resultado inicial de la heurística tenemos que es $0,023$.

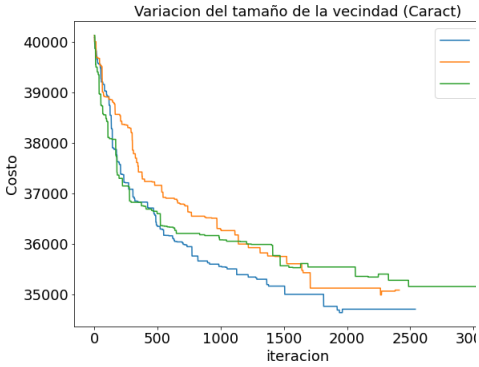
Por lo visto en las instancias *"random"* y en *att48*, siendo un caso real, podemos concluir que Tabú Search puede mejorar considerablemente las soluciones iniciales provistas por heurísticas golosas dado el tiempo necesario.

Notar que el gap relativo lo calculamos como $(x_b - Opt)/Opt$

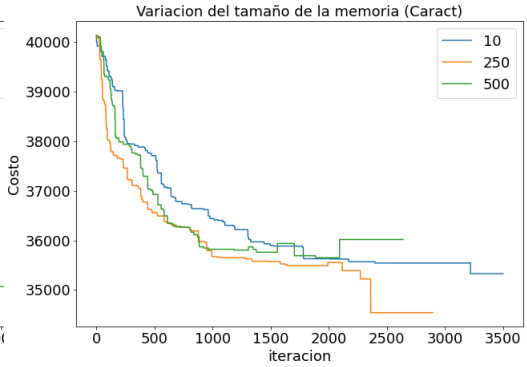
3.6.2. Comparación de performance con distintos parámetros de entrada

Por otro lado, no solo queremos ver la mejoría de Tabú Search sobre las heurísticas, sino que queremos observar como varían los resultados alterando los parámetros de entrada. Para ello realizamos modificaciones en la memoria, en la condición de activación de la función de aspiración, en el tamaño de la sub-vecindad y en la condición de terminación, obteniendo cuatro gráficos por cada implementación de memoria. En ellos se contempla la modificación de cada uno de estos aspectos para así ver que cambios genera en los resultados. Para este experimento utilizamos la heurística de la arista más corta y un set de valores estándar usando como caso de test *att48*. Notar que los gráficos tienen periodos de crecimiento. Esto creemos que se debe a que, como por cada caso corrimos muchas veces y luego hicimos el promedio del resultado, y como las corridas tienen distinta cantidad de iteraciones, hubieron

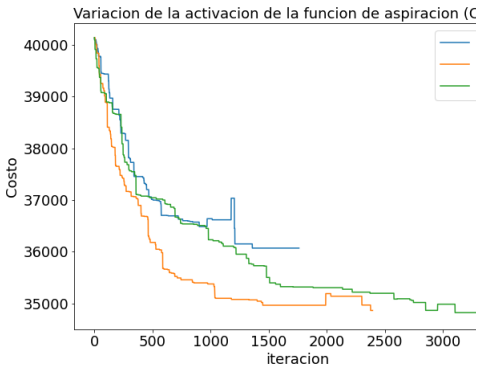
pocos casos en los que hubieron mas iteraciones y en estos casos, la mejor solución no fue tan óptima como otras ejecuciones en las que hubieron menos iteraciones



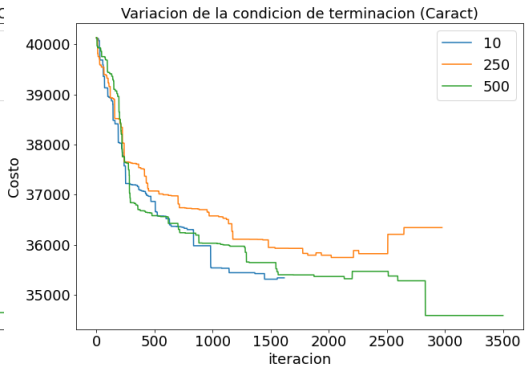
(a) Evolución con distintos tamaños de sub-vecindad



(b) Evolución con distintos tamaños de memoria



(c) Evolución con distintos valores de la activación de la función de aspiración



(d) Evolución con distintos valores de la condición de terminación

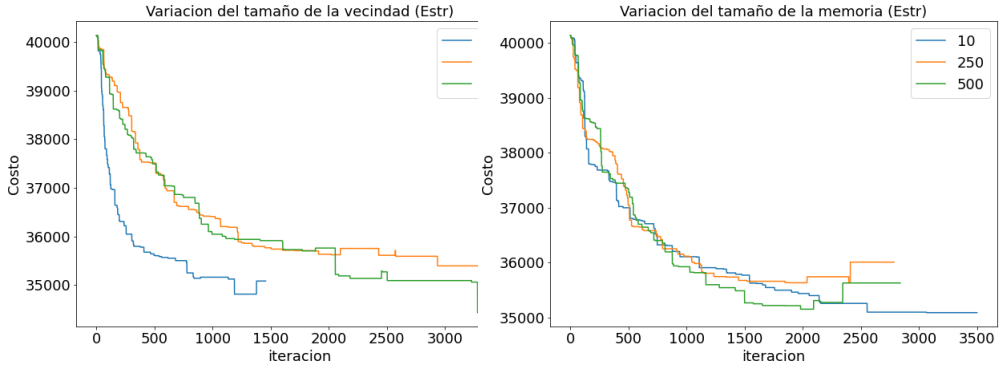
Figura 5: Gráficos obtenidos de la ejecución de Tabú Search con memoria de características con distintos parámetros de entrada

TS con memoria de características Observando los gráficos correspondientes a la implementación sobre una memoria de características (5), podemos concluir que:

- **Sub-vecindad (5a)**: Para las variaciones del tamaño de la sub-vecindad, podemos ver como la solución mas óptima fue encontrada cuando el tamaño fue 10. Para valores mas grandes, el algoritmo realizo mas iteraciones sin encontrar una solución mejor. Para valores mas grandes, se puede ver que Tabú Search termina varias iteraciones antes de poder mejorar su solución más óptima. Con esto se puede concluir que a partir de cierto tamaño de vecindad, aumentar su tamaño empeora el desempeño de Tabú Search, ya que toma demasiados valores y es menos probable obtener el vecino que te lleve al óptimo. Mientras que si es muy chica, hay pocas probabilidades de que el vecino que te lleve al óptimo siquiera este en tu vecindad. Este análisis es posible porque lo que se guarda en memoria es el valor a optimizar.
- **Tamaño de memoria (5b)**: Con las variaciones de memoria, podemos observar que

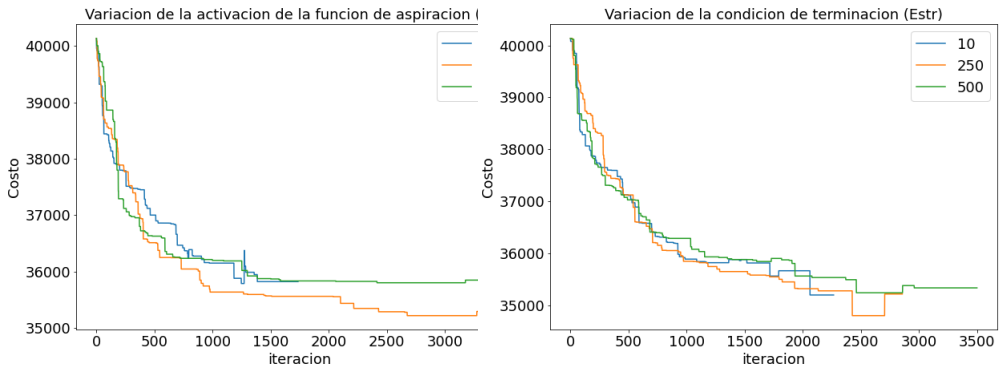
una memoria de tamaño 500 no es buena. Sin embargo vemos que el algoritmo funciona muy bien para tamaños menores. Para tamaños grandes lo que ocurre es que es mas probable encontrar una solución ya recorrida (ya que hay mas soluciones guardadas).

- **Condición de activación de la función de aspiración (5c)**: Cuando variamos el valor de activación para la función de aspiración podemos notar que para el valor 10 la función no tiene un buen desempeño, ya que revisita mucho valores y termina de ejecutarse de manera temprana por la condición de terminación. Para los demás valores podemos ver que el valor mas cercano al tamaño de la memoria (250) da mejores resultados,ya que encuentra una de las mejores soluciones mas rápido que 500, esto es porque al tener una condición de activación demasiado grande, la función de aspiración tarda demasiado en activarse y es probable que no encuentre una solución mejor antes de que termine por la cantidad máxima de iteraciones.
- **Condición de terminación (5d)**: Los gráficos muestran que el valor 500 nos da los mejores resultados, mientras también se ve que el valor de 10 funciona mejor que el de 250.



(a) Evolución con distintos tamaños de sub-vecindad

(b) Evolución con distintos tamaños de memoria



(c) Evolución con distintos valores de la activación de la función de aspiración

(d) Evolución con distintos valores de la condición de terminación

Figura 6: Gráficos obtenidos de la ejecución de Tabú Search con memoria por estructura con distintos parámetros de entrada

TS con memoria de estructura Observando los gráficos correspondientes a la implementación sobre una memoria de estructura(6), podemos concluir que:

- **Sub-vecindad (6a)**: Para las variaciones del tamaño de la sub-vecindad, podemos ver que, a pesar que con 10 obtiene soluciones mejores antes, el algoritmo termina antes de poder mejorar mas. Así mismo, vemos que con 500 logro obtener una solución de mucho menor costo, aunque haya tardado varias iteraciones mas.
- **Tamaño de memoria (6b)**: Con las variaciones de memoria, podemos observar que la mejor solución fue encontrada por la de tamaño 10 a pesar de haber tardado mas en terminar. Esto puede deberse a que es menos probable encontrar una solución ya obtenida en una memoria pequeña, por lo que la función de aspiración se activara menos veces, alargando el tiempo de ejecución. Además vemos que, como tuvo mas tiempo de ejecución, tuvo mas tiempo para buscar para, finalmente, encontrar la solución mas óptima.
- **Condición de activación de la función de aspiración (6c)**: Para las variaciones en la condición de activación de la función de aspiración podemos ver claramente que el mejor caso fue el de 250. Los motivos por los cuales pasa esto pueden ser porque para valor 10 tabú search ejecuta la función de aspiración muchas veces por lo que, como es una condición de terminación, hará que termine mucho antes. Por el contrario, para valor igual a 500,la tabú search ejecuta muy pocas veces la función de aspiración, por lo que se queda estancado en un mínimo local, evitando recorrer otras sub-vecindades.
- **Condición de terminación (6d)**: Para las variaciones en la condición de terminación, las mejores soluciones fueron encontradas cuando este valor era 250.

Concluyendo, vemos que los parámetros que optimizan los resultados de TS son:

Implementado con memoria de características

- **MEM_SIZE** = 250
- **ASPIRATION_STALL** = 250
- **MAX_ITERATIONS** = 3500
- **MAX_VICINITY_SIZE** = 10
- **TERMINATION_CONDITION** = 500

Implementado con memoria de estructuras

- **MEM_SIZE** = 10
- **ASPIRATION_STALL** = 250
- **MAX_ITERATIONS** = 3500
- **MAX_VICINITY_SIZE** = 500
- **TERMINATION_CONDITION** = 250

Observando las tablas con los resultados obtenidos con distintas instancias y memorias de TS (Cuadro 2 y 3), vemos que los diferenciales para los mismos casos de test entre ambas memorias (observar x_b) son relativamente chicos comparado con el peso total. Por lo que entonces elegimos una estructura de memoria arbitraria para ejecutar sus parámetros optimizadores, por lo que elegimos *memoria de característica*.

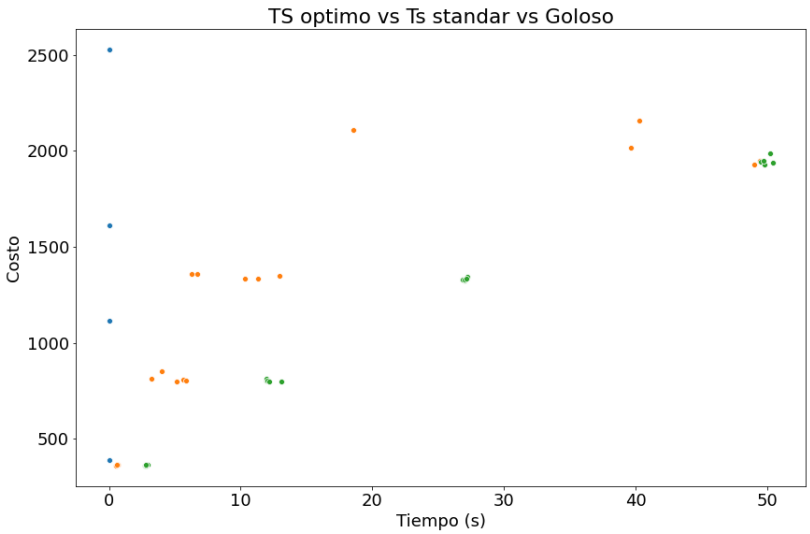


Figura 7: Comparación de Tiempo vs Costo con TS optimizado con tamaños de instancias: 20, 40, 60, 80. Azul = Goloso, Naranja = TS con parámetros estándar y memoria por características, Azul = TS con parámetros óptimos y memoria por estructura. Los cambios abruptos de altura dividen las instancias en las que corrieron (mas alto, mayor instancia)

Por último podemos concluir analizando las figura 7, como *TS* con los parámetros optimizados sobre instancias "random"de distintos tamaños resulta tener una mejoría en los costos resueltos sobre hacer *TS* con el conjunto de *parámetros estándar* llevando a que las soluciones de los parámetros no optimizados sean mayores o iguales a las de su contra parte, pero ralentiza la respuesta del algoritmo.

4. Conclusiones

En el informe pudimos experimentar distintas heurísticas y meta-heurísticas para solucionar el problema de *TSP*.
Como esperamos, las heurísticas golosas evaluadas (NN, SE) y la heurística de AGM presentan soluciones rápidas, pero con *gap relativos* bastante elevados teniendo en cuenta que una pequeña optimización puede ahorrar millones a una empresa.
Por otro lado, contrastamos las optimizaciones que genera Tabú Search sobre las soluciones de las heurísticas, resolviendo que efectivamente baja el gap entre la solución óptima y la solución resuelta, si se desea esperar tiempos que crecen rápidamente si la cantidad de nodos crece. También sobre esta meta-heurística vemos que es dependiente de la correcta decisión de rangos para los parámetros que recibe para optimizar, en caso que no se elijan parámetros dentro del mismo, Tabú Search puede dar soluciones muy costosas o tardar más para llegar a una solución cercana al óptimo.
Podemos pensar que a futuro, se puede mejorar el análisis sobre las estructuras de memoria elegidas de las implementaciones de Tabú Search, para resolver si una de las dos es realmente mejor que la otra. También es posible en pensar *otras* implementaciones de la misma.

Referencias

[1] https://campus.exactas.uba.ar/pluginfile.php/191415/mod_resource/content/1/algo3_agm.pdf

[2] https://campus.exactas.uba.ar/pluginfile.php/187684/mod_resource/content/3/4-arboles.pdf

[3] Bijaya Ketan Panigrahi, Ponnuthurai Nagaratnam Suganthan, Swagatam Das, Shubhransu Sekhar Dash. *Swarm, Evolutionary, and Memetic Computing*. 4th International Conference, SEMCCO 2013, Chennai, India, December 19-21, 2013, Proceedings, Part II Springer, 2013.

[4] David L. Applegate, Robert E. Bixby, Vašek Chvátal, William J. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, Sep 19, 2011.

[5] http://elib.zib.de/pub/mp-testdata/tsp/tsplib/stsp-sol.html