DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

The Travelling Salesman Problem

Integrante	LU	Correo electrónico
Sosa, Patricio	218/16	patriciososa91@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

La teoría de grafos, es una rama de las matemáticas y las ciencias de la computación que estudia las propiedades de los grafos. Un grafo es una construcción matemática compuesta por dos conjuntos: uno de vértices o nodos y uno de aristas que unen dichos vértices. Su estructura los hace aptos para el modelado de numerosos problemas, pues la existencia de entidades conectadas entre sí es frecuente en la realidad.

La función principal de un grafo es la abstracción o modelado de un problema. Sobre un problema abstraído como un grafo se pueden aplicar propiedades y algoritmos con el fin de analizarlo o resolverlo. Existen varios problemas modelados con grafos que requieren diferentes conceptos de otras áreas tales como la Optimización Combinatoria.

En optimización combinatoria se busca la mejor manera de resolver un problema entre un conjunto de posibles soluciones al mismo. Para hablar de la mejor manera, se debe asociar cada una de las posibles soluciones a una función objetivo que nos permita evaluarlas y compararlas entre si. Estos problemas son de mucho interés en la práctica porque pueden modelar situaciones reales, en donde tenemos que tomar una decisión sobre una tarea y la valuación puede representar una métrica que nos resulte de interés, como una ganancia asociada o un costo a pagar.

Existen una gran variedad de problemas reales que se pueden resolver mediante optimización combinatoria. En este caso abordaremos el Problema del Viajante de Comercio (TSP, por sus siglas en inglés) que consiste en encontrar un circuito hamiltoniano de costo mínimo, es decir, un circuito que visite todos los vértices exactamente una vez. A continuación detallamos el problema.

1.1. El problema: TSP

El Problema del Viajante de Comercio (The Travelling Salesman Problem-TSP) resuelve el siguiente problema: dada una lista de ciudades y las distancias entre cada par de ellas, ¿Cuál es la ruta más corta posible que visita cada ciudad exactamente una vez y al finalizar regresa a la ciudad origen?.

El Problema del Viajante de Comercio (TSP) se puede definir formalmente de la siguiente manera:
Dado un grafo completo $G = (V, X)$, donde cada arista $(i, j) \in$ X, tiene un costo asociado $c_{i,j}$. Se define el costo de un camino p como la suma del costo de las aristas que componen el camino p. Es decir:

- $c_p = \sum_{(i,j)\in P} c_{i,j}$

El problema consiste en encontrar un ciclo hamiltoniano de costo mínimo.

Ejemplo 1.1. *En la Figura 1 se presenta un grafo de 4 nodos en el cual una solución óptima es el ciclo conformado por las aristas marcadas en azul.*

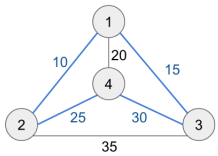


Figura 1: Ejemplo de TSP con un grafo de 4 nodos

2. Heurísticas

2.1. Heurística Constructiva Golosa basada en el Vecino Mas Cercano-VMC

Es la heurística mas intuitiva para este problema y lo que haríamos naturalmente. En cada paso elegimos como siguiente lugar a visitar el que, entre los que nodos todavía no visitados, se encuentre mas cerca de donde nos encontramos. En la Figura 2 se muestran los pasos realizados por esta heurística.

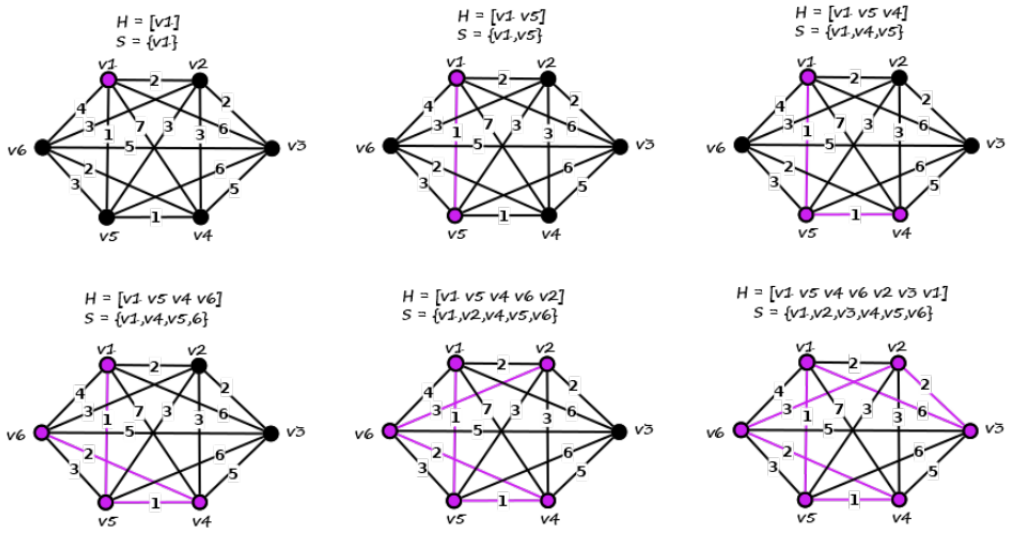


Figura 2: VMC: En cada paso elige el vértice mas cercano, que tiene como vecino el ultimo vértice visitado, hasta finalmente formar un ciclo.

Algorithm 1 Algoritmo de la Heurística Constructiva Golosa VMC.

```
1:  function VMC(g : Grafo)
2:      costoCiclo ← 0                                          ▷ O(1)
3:      visitados ← vector(n, false)(g)                         ▷ O(n)
4:      vecino ← 1                                              ▷ O(1)
5:      visitados[vecino] ← true                                ▷ O(1)
6:      ciclo ← vectorVacio()(g)                                ▷ O(1)
7:      while queden nodos por visitar do                       ▷ O(n − 1)
8:          vmc ← verticeMasCercano(ciclo, vecino)              ▷ O(n)
9:          visitados[vmc] ← true                               ▷ O(1)
10:         ciclo.push_back(vmc)                                ▷ O(1)
11:         vecino ← vmc                                        ▷ O(1)
12:         costoCiclo ← costoCiclo + grafo[vecino][vmc]        ▷ O(1)
13:     return ciclo.
```

La complejidad de la heurística en el peor caso es $O(n^2)$, donde n es la cantidad de vértices, es decir es lineal en el tamaño de la entrada dado que el grafo es completo.

2.2. Heurística Constructiva Golosa Por Inserción

Esta heurística constructiva toma como entrada un ciclo hamiltoniano formado por tres vértices cualesquiera, y en cada paso la heurística selecciona un vértice que minimice la distancia entre el ciclo y el nuevo nodo(*ELEGIR*), luego inserta el mismo entre dos vértices consecutivos de forma tal que la suma de las nuevas aristas que se incorporen al ciclo sea mínima(*INSERTAR*).

Algorithm 2 Heurística constructiva golosa de inserción.

```
1: function HI(g : Grafo)
2:     H ← [v1, u, w]
3:     while |H| < n do                          ▷ O(n − 3)
4:         ELEGIR un nodo de v ∈ V \ H            ▷ O(n ∗ |H|)
5:         INSERTAR v en H                        ▷ O(|H|)
6:     return H.
```

ELEGIR revisa todos los vecinos de los vértices que se encuentran en H para encontrar el vértice mas cercano que aun no esta en H, la complejidad de esta operación es $O(|H| * n)$, con $|H|$ la longitud del ciclo (en el peor caso $|H| = n$). Por lo tanto *ELEGIR* en el peor caso es $O(n^2)$, a su vez, *INSERTAR* toma $O(|H|)$ que como dijimos anteriormente en el peor caso $O(n)$. Luego la complejidad total del algoritmo es el ciclo principal que en el peor caso es $O(n)$ y en cada iteración tenemos *ELEGIR* + *INSERTAR* que es $O(n^2)$, por lo tanto la complejidad total es de $O(n * n^2) = O(n^3)$.

2.3. Heurística Basada en Árbol Generador Mínimo

La idea es usar el AGM del grafo para generar un ciclo. Lo primero que hace el algoritmo es obtener el AGM T del grafo, luego ejecuta el algoritmo DFS sobre T con el objetivo de guardarse el recorrido sobre el árbol T. Finalmente construye el ciclo siguiendo el recorrido DFS pero salteando los vértices ya visitados.

La complejidad del algoritmo es $O(n^2 log(n) + n + n) = O(n^2 log(n))$. Para obtener el AGM se utiliza un algoritmo de complejidad $O(mlog(n))$ y como el grafo es completo, m es $O(n^2)$.

Algorithm 3 Heurística constructiva basada en árbol generador mínimo.

```
1: function AGMH(g : Grafo)
2:     T ← AGM(g)                                 ▷ O(n^2 log(n))
3:     recorrido_dfs ← obtenerRecorridoDfs(T)     ▷ O(n)
4:     ciclo ← crearCiclo(recorrido_dfs)          ▷ O(n)
5:     return ciclo.
```

En la Figura 3 se muestran los pasos realizados por esta heurística.

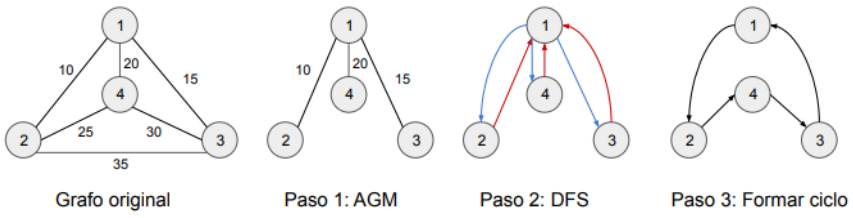


Figura 3: Ej. Heurística basada en AGM

Notar que el peso del AGM T es una cota inferior de la solución de TSP, esto vale porque la solución de TSP incluye un árbol generador, y T es un árbol generador mínimo.

Además, si la función que asigna pesos a las aristas cumple la desigualdad triangular, entonces el ciclo C_3 (paso 3) es menos costoso que el de C_2 (Paso 2).

Finalmente, $peso(T) \leq peso(OPT) \leq peso(C_3) \leq peso(C_2) \leq 2peso(T)$.

Donde OPT es una solución óptima del problema. Con lo cual, se puede asegurar que C_3 esta a menos de $peso(T)$ de la solución óptima.

3. Metaheurísticas

Las heurísticas constructivas nos permiten obtener una solución de calidad aceptable.

Para intentar obtener una solución mejor, se puede usar un algoritmo de búsqueda local, que toma una solución factible s del problema, en este caso un circuito hamiltoniano, y se encarga de recorrer soluciones parecidas con el objetivo de encontrar la mejor de estas soluciones, según la función objetivo.

Dada una solución s del problema, al conjunto de soluciones parecidas a s se lo conoce como vecindad de s, y es el resultado de modificar levemente a s.

Las metaheurísticas son métodos que se encargan de diversificar el espacio de búsqueda propuesto por los algoritmos de búsqueda local, con el objetivo de encontrar mejores soluciones y evitar quedarse estancado en una solución óptima local. En este trabajo se estudiaran 2 versiones de la metaheuristica tabu search.

3.1. Tabu Search

El método comienza con una solución inicial y luego, iterativamente se mueve a una solución mejor en su vecindad. Se utiliza una lista tabú para evitar explorar soluciones ya visitadas o movimientos realizados. Los siguientes elementos son utilizados para la implementación del método:

- **memoria:** Es la lista tabu, se le fija un tamaño t. Se utiliza 2 tipos de memoria, una basada en ultimas soluciones exploradas (ciclos), y otra basada en estructuras (aristas). El objetivo de la memoria es no repetir soluciones visitadas o movimientos realizados. Cada memoria define una implementación distinta de tabu search.
- **vecindad:** Se utiliza la vecindad 2-OPT. La vecindad 2-OPT de un ciclo S esta compuesta por todos los ciclos que se forman tomando 2 aristas en S, e intercambiando sus extremos. En la implementación de tabu search se agrega un parámetro para obtener solo un porcentaje de la vecindad. Los elementos en la subvecindad serán elegidos de manera aleatoria hasta completar el porcentaje requerido.
- **criterio de parada:** En este caso se define como la cantidad de iteraciones a realizar.

3.2. Metaheurística Tabu-Search Basada en las ultimas soluciones exploradas

La complejidad es $O(S + n + t + iteraciones * max\{n^2, |vecinos| * t * n\})$, donde:

- **S** es el costo de obtener la solución inicial, el mismo depende de la heurística utilizada.
- $|vecinos|$ es el tamaño de la vecindad.
- **t** es el tamaño de la memoria.

3.3. Metaheurística Tabu-Search Basada en estructuras (aristas)

La complejidad es $O(S + n + t + iteraciones * max\{n^2, |vecinos| * t\})$.

Algorithm 4 Algoritmo tabu search basado en ciclos.

1: **function** $tabuSearchC(g : Grafo, iteraciones : Entero, t : Entero, porcentaje : Entero)$
2: $ciclo \leftarrow solucionInicial(g)$ ▷ $O(S)$
3: $mejorCiclo \leftarrow ciclo$ ▷ $O(n)$
4: $memoria \leftarrow listaVacia(tamanoMaximo = t)$ ▷ $O(t)$
5: **while** criterio de parada **do** ▷ $O(iteraciones)$
6: $vecinos = obtenerSubVecindad(ciclo, porcentaje)$ ▷ $O(n^2)$
7: $ciclo = obtenerMejor(vecinos, memoria)$ ▷ $O(|vecinos| * t * n)$
8: $memoria.recordar(ciclo)$ ▷ $O(n)$
9: **if** $costo(ciclo) < costo(mejorCiclo)$ **then** ▷ $O(n)$
10: mejorCiclo = ciclo ▷ $O(n)$
11: **return** $mejorCiclo$.

Algorithm 5 Algoritmo tabu search basado en estructuras.

1: **function** $tabuSearchC(g : Grafo, iteraciones : Entero, t : Entero, porcentaje : Entero)$
2: $ciclo \leftarrow solucionInicial(g)$ ▷ $O(S)$
3: $mejorCiclo \leftarrow ciclo$ ▷ $O(n)$
4: $memoria \leftarrow listaVacia(tamanoMaximo = t)$ ▷ $O(t)$
5: **while** criterio de parada **do** ▷ $O(iteraciones)$
6: $vecinos = obtenerSubVecindad(ciclo, porcentaje)$ ▷ $O(n^2)$
7: $swap = obtenerMejor(vecinos, memoria)$ ▷ $O(|vecinos| * t)$
8: $ciclo = reconstruirCiclo(ciclo, swap)$ ▷ $O(n)$
9: $memoria.recordar(swap)$ ▷ $O(1)$
10: **if** $costo(ciclo) < costo(mejorCiclo)$ **then** ▷ $O(n)$
11: mejorCiclo = ciclo ▷ $O(n)$
12: **return** $mejorCiclo$.

4. Experimentación

4.1. Métodos

Los métodos utilizados durante la experimentación son los siguientes:

- **VMC**: heurística de vecino mas cercano.
- **HI**: heurística de inserción.
- **AGMH**: heurística basada en árbol generador mínimo.
- **TSC**: tabu search con memoria basada ultimas soluciones exploradas (ciclos).
- **TSE**: tabu search con memoria basada en estructura (aristas).
- Según la heurística que se utilice para obtener una solución inicial de tabu search, se obtienen los siguientes métodos: **TSE-VMC, TSE-HI, TSE-AGM, TSC-VMC, TSC-HI, TSC-AGM**.

4.2. Métricas

Para medir la calidad de una solución obtenida se utiliza el **gap relativo** con respecto a una solución óptima conocida.

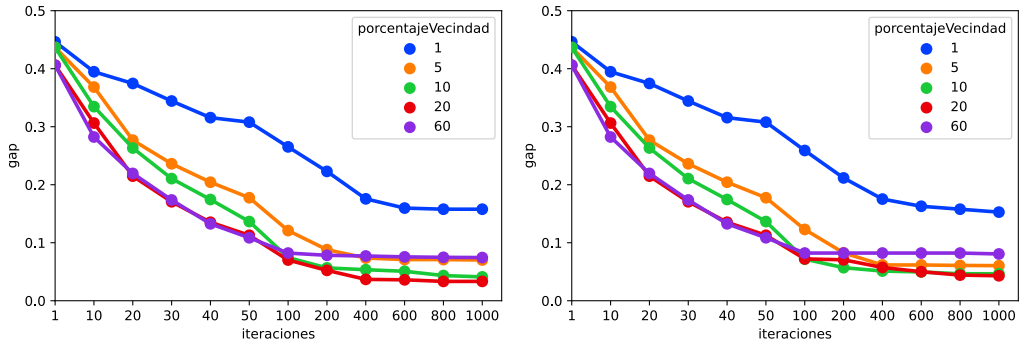
$$\text{gap relativo} = \frac{f(x) - f(x^*)}{f(x^*)},$$

donde f calcula el costo de una solución, x es una solución y x^* es una solución óptima conocida.

4.3. Experimento Iteraciones Vecinos:

En este experimento se quiere analizar cómo varían los resultados de tabu search al considerar distintos tamaños de vecindades.

Para el dataset a280.tsp, se ejecutaron los algoritmos TSE-AGM y TSC-AGM para distintos porcentajes de vecindades y variando la cantidad de iteraciones.



(a) Gap según vecindad TSE-AGM.

(b) Gap según vecindad TSC-AGM.

Figura 4: Análisis de vecindad para los métodos TSE-AGM y TSC-AGM. El dataset utilizado es a280.tsp y los parámetros son: memoria = 200, porcentaje de vecindad: 1, 5, 10, 20, 60. Iteraciones: 1, 10, 20, 30, 40, 50, 100, 200, 400, 600, 800, 1000.

En la figura 4 presentan los resultados, en el eje x se tienen las iteraciones realizadas y, en el eje y, el gap relativo con respecto a la solución óptima conocida.

Cuadro 1: GAP relativo respecto a la solución óptima, para a280.tsp usando 1000 iteraciones.

	TSE-AGM				
Porcentaje vecindad	1	5	10	20	60
GAP aprox.	15,7 %	7 %	4,1 %	3,3 %	7,4 %

TSC-AGM				
1	5	10	20	60
15,2 %	6 %	4,6 %	4,3 %	8 %

Para ambos gráficos, al variar entre el 5 % y el 20 % de la vecindad se observan mejoras en los resultados obtenidos, logrando las mejores soluciones usando un 20 %.

Al aumentar al 60 % los resultados empeoraron. La hipótesis es que al aumentar el tamaño de la vecindad a valores tan grandes, tabú search tiende a comportarse como búsqueda local y por lo tanto será más propenso a estancarse en un óptimo local.

Por ultimo, al aumentar la cantidad de iteraciones, los métodos aumentan la cantidad de soluciones exploradas y por lo tanto tienen mas oportunidad de obtener mejores soluciones, esto se comprueba en la Figura 4. En el Cuadro 1 se muestra en detalle los valores de gap que se obtuvieron con 1000 iteraciones para los distintas metaheuristicas.

4.4. Experimento Iteraciones Memoria:

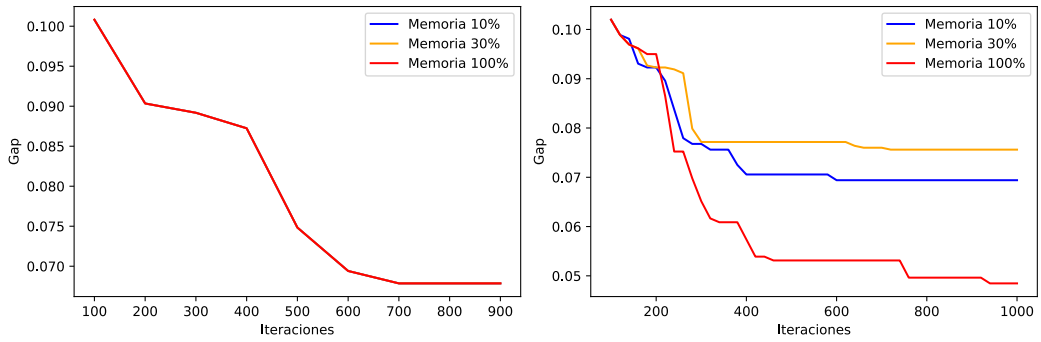
Este experimento tiene como fin analizar el comportamiento de Tabu Search cuando variamos el tamaño de la lista tabu y el valor de las iteraciones.

Para tal fin comparamos el gap obtenido por las metaheurísticas TSC-AGM y TSE-AGM usando el dataset *a280.tsp* que consta de 280 vértices. Se toman valores de memoria proporcionales al tamaño de los vértices de la entrada y las iteraciones entre el rango de 100 a 1000.

En particular para la experimentación las longitudes de memoria usada fueron 28, 84 y 280 que corresponden al 10 %, 30 % y 100 % de la cantidad de vértices. Por último se fijaron los porcentajes de vecindad en 10 % y 20 % respectivamente.

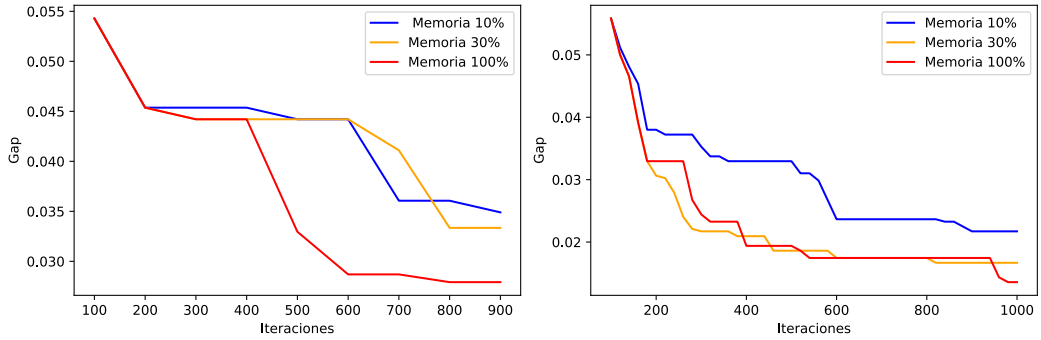
En la Figura 5a podemos observar que cuando corremos la metaheurístca TSC-AGM con el 10 % de la vecindad, los tres graficos se superponen, en este caso, variar el tamaño de la memoria no genera una mejora en el gap. Por el contrario, con la metaheurístca TSE-AGM (Figura 5b), se ve claramente como mejora el gap al aumentar el tamaño de la memoria. Podemos suponer que al usar solo el 10 % de la vecindad, el método TSC-AGM *"explora poco"* por lo cual variar la memoria no mejora las soluciones guardadas en la lista tabu.

Por otro lado, cuando tenemos una vecindad del 20 % (Figura 6), variar el tamaño de la memoria representa una mejora en el gap para ambas metaheurísticas, siendo mas visible esta mejora en el método TSC-AGM(Figura 6a), y obteniendo los mejores valores de gap para la metaheuristica TSE-AGM.



(a) Gap según memoria TSC-AGM. (b) Gap según memoria TSE-AGM..

Figura 5: Comparación del gap de los métodos TSC-AGM y TSE-AGM tomando como entrada el dataset *a280*,manteniendo fijo el valor de vecindad en un 10 % y variando los porcentajes de memoria e iteraciones.



(a) Gap según memoria TSC-AGM. (b) Gap según memoria TSE-AGM..

Figura 6: Comparación del gap de los métodos TSC-AGM y TSE-AGM tomando como entrada el dataset *a280*,manteniendo fijo el valor de vecindad en un 20 % y variando los porcentajes de memoria e iteraciones.

Como conclusión podemos asegurar que al tomar una vecindad del 20 % el aumentar el número de iteraciones y el tamaño de la memoria produce una mejora notable en el gap, siendo la metaheurística TSE-AGM la que obtiene mejores resultados. Además el mejor gap se obtuvo cuando el tamaño de memoria (proporcional al tamaño de los vértices de la entrada) se corresponden con el 100 %.

4.5. Experimento: Comparación Gap y Tiempos de soluciones óptimas

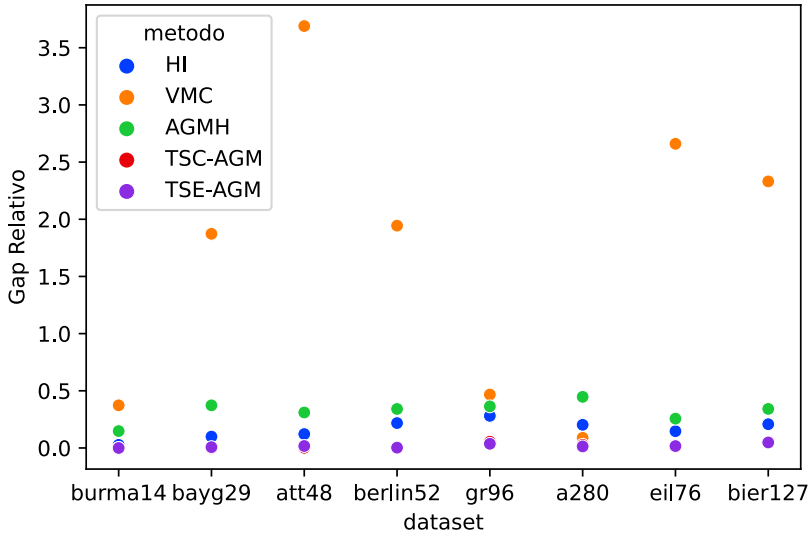
En este experimento nos proponemos ver el comportamiento de las diferentes heurísticas utilizando como dataset las instancias burma14, bayg29, att48, berlin52, eil76, gr96, bier127, a280 para las cuales se conocen los valores óptimos y fijando como parámetros para las metaherísticas Tabu Search, 1000 iteraciones, memoria igual a la cantidad de vértices y una vecindad del 20 %, que resultaron ser los mejores parámetros para tabu search, según los experimentos previos.

En el Cuadro 2 y la Figura 7 se compara el gap relativo un función del tamaño de la entrada(cantidad de vértices) para todas las heurísticas(HI, VMC, AGMH, TSC-AGM, TSE-AGM).

Dataset	Heurísticas				
	HI	VMC	AGMH	TSC-AGM	TSE-AGM
Burma14	0.027084	0.372856	0.147758	0.000000	0.000000
Bay29	0.099379	1.872671	0.372671	0.000000	0.007453
Att48	0.121566	3.689499	0.310218	0.001882	0.017313
Berlin52	0.217847	1.944179	0.341024	0.000000	0.003050
Eil76	0.146840	2.659851	0.256506	0.011152	0.016729
Gr96	0.280769	0.467279	0.363926	0.054846	0.036733
Bier127	0.207749	2.330929	0.341531	0.049568	0.048148
A280	0.202404	0.088794	0.447073	0.027918	0.013571

Cuadro 2: Gap relativo respecto a la solución óptima para los métodos HI, VMC, AGMH, TSC-AGM, TSC-AGM tomando como entrada los datasets burma14, bay29, att48, berlin52, eil76, gr96 bier127 y a280.

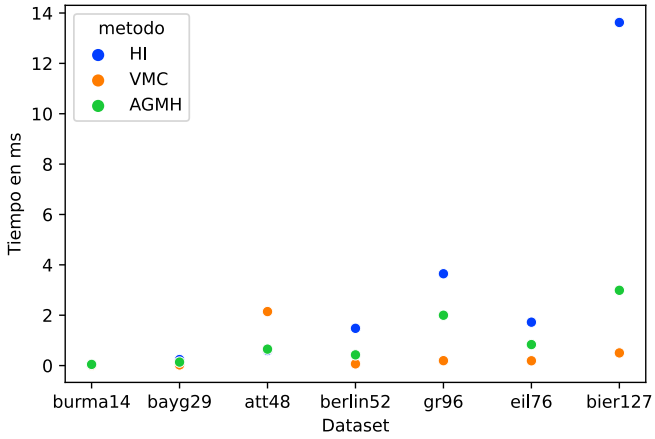
Podemos observar que la metaheurística TSE-AGM obtiene los mejores valores para todas las instancias, lo cual era de esperarse, ya que este método toma una como entrada una solución inicial cualquiera y luego la mejora(en términos del gap). Además observamos que el método VMC, en varios casos, obtiene los resultados con mayor gap relativo, una posible explicación para este comportamiento es que esas instancias son similares al peor caso para heurística.



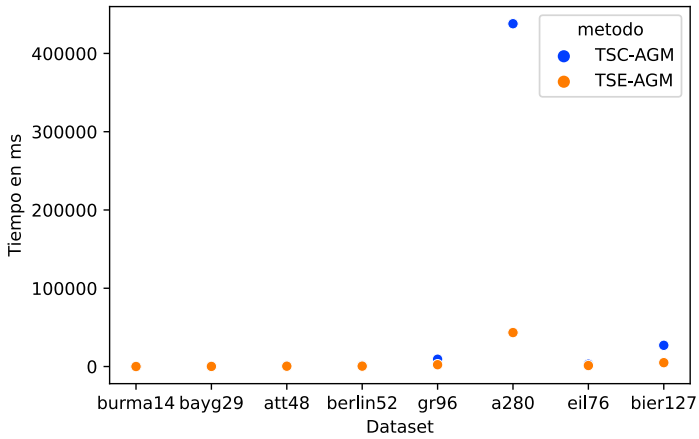
(a) Gap relativo para las diferentes instancias.

Figura 7: Análisis del gap relativo para las heurísticas VMC, HI, AGMH, TSC-AGM y TSE-AGM.

Por otro lado en la Figura 8 podemos ver los tiempos de ejecución de todos los métodos utilizando las mismas entradas. Se observa que la performance de las heurísticas constructivas(Figura 8a) es mejor, en cuanto a tiempos, comparado a las metaheurísticas Tabu Search(8b).



(a) Tiempo de ejecución para las heurísticas constructivas.



(b) Tiempo de ejecución para las metaheurísticas Tabu Search.

Figura 8: Análisis de tiempo de ejecución para todos los métodos.

Por lo tanto podemos concluir que si bien Tabu Search obtiene mejores resultados para calcular el costo del los circuitos hamiltonianos, estas metaheurísticas introducen el costo adicional de tener un tiempo de ejecución superior a las heurísticas constructivas, y que según el problema a resolver, esto puede resultar bastante restrictivo, por lo tanto no podemos afirmar que alguno de los métodos sea mejor a los demás, la elección de un método sobre otro debe tener en cuenta el tiempo que estamos dispuestos a esperar por un resultado y la calidad de la solución.

4.6. Experimento: Comparación Calidad entre Algoritmos

4.6.1. Comparación Calidad Heurísticas

En este experimento preliminar se busca comparar la calidad de las distintas heurísticas sobre datasets generados aleatoriamente, de los cuales no se conoce una solución óptima.

En la Figura 9 se presenta un gráfico con boxplots, en los cuales cada boxplot representa las distintas instancias de tamaño n. Como podemos observar la heurística HI es la que logra obtener mejores soluciones(óptimas) para este tipo de entrada, para el caso de AGMH se puede observar

que para tamaños de entradas pequeños, la diferencia es poca contra HI pero a medida que aumenta se va deteriorando. Finalmente tenemos que la heurística VMC es la que peor soluciones genera, y a medida que aumenta el tamaño de entrada sigue empeorando dando muy malos resultados en comparación al resto de las heurísticas.

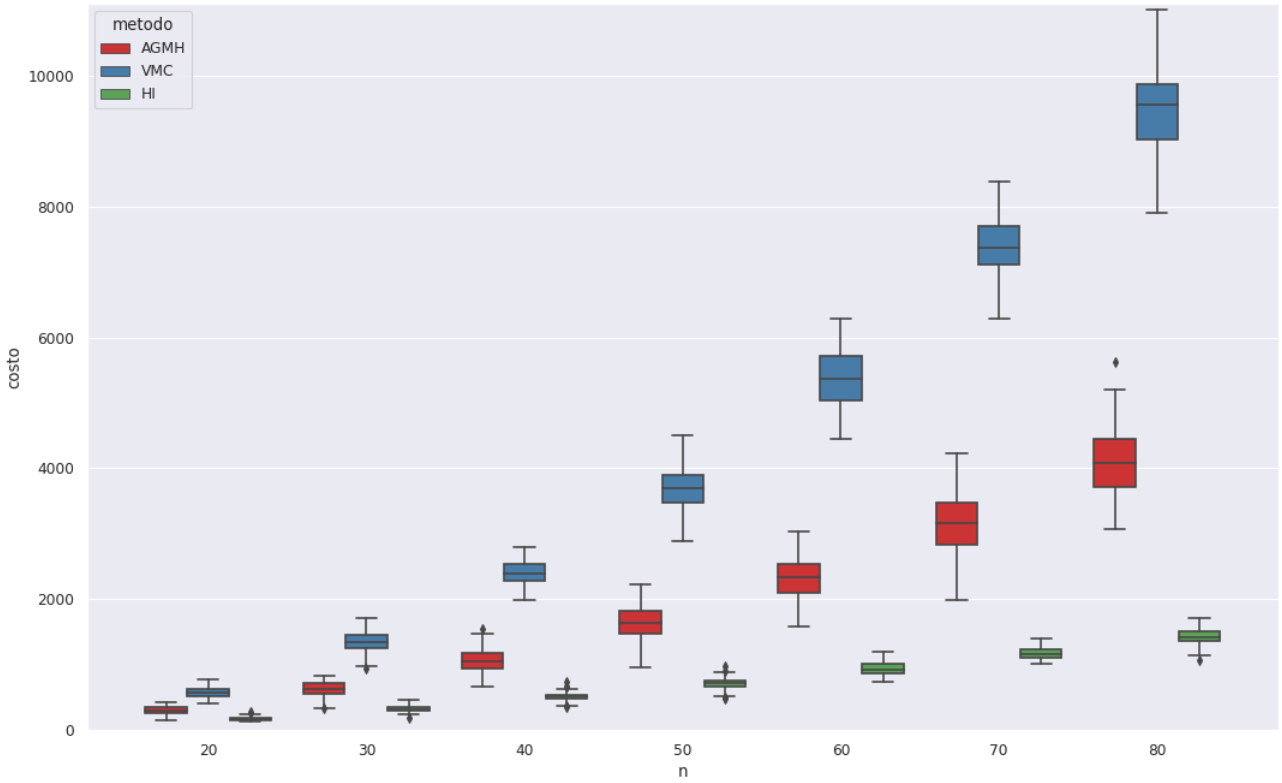


Figura 9: Comparación Calidad en Heurísticas. Para cada entrada de tamaño n, se generaron 100 instancias distintas, para los costos de cada instancia se utilizo la función random(randint), el cual para cada instancia de tamaño n, generaba un array de costos, donde cada valor equiprobable esta entre $[0, 3 * n]$.

4.6.2. Comparación Calidad Metaheuristicas

En este experimento preliminar se busca comparar la calidad entre las metaheurísticas sobre datasets generados aleatoriamente, de los cuales no se conoce una solución óptima.

En la Figura 10 se comparan las metaheurísticas basadas en soluciones exploradas(TSC-AGM) vs la basada en estructuras(TSE-AGM), en este caso fijando en ambas la heurística AGMH.

En este caso se observa que si bien para valores pequeños del tamaño de entrada no se ve diferencia entre ambas metaheuristicas, a medida aumenta se puede apreciar una diferencia en la media, la cual es a favor para la metaheurística basada en ultimas soluciones exploradas. Además contamos con la presencia de varios outliers a medida que aumenta la cantidad instancias, esto es debido a la naturaleza de las metaheuristicas ya que nunca pueden garantizar que la solución siempre sea la mejor.

Con estos resultados podemos decir que para esta familias de instancias, en cuanto a Heurísticas, una muy buena opción es la de inserción, mientras que para las metaheuristicas, la basada en

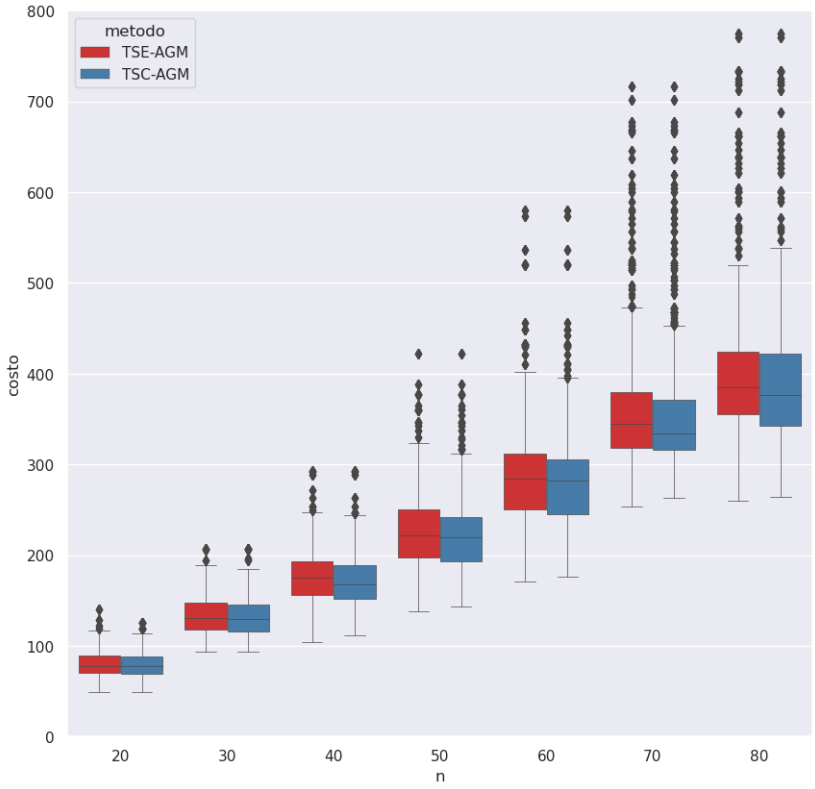


Figura 10: Comparación Calidad en Metaheurísticas. Para cada entrada de tamaño n, se generaron 10 instancias distintas, para los costos de cada instancia se utilizo la función random(randint), el cual para cada instancia de tamaño n, generaba un array de costos, donde cada valor equiprobable esta entre $[0, 2 * n]$. Luego para cada input se ejecutaron las metaheristicas con las siguientes combinación de parámetros: porcentaje de vecindad p, con p $[10, 20, 30, 40, 50]$,una memoria de tamaño t entre $[20, 40, 60, 80, 100]$,i iteraciones con i entre $[20, 500]$.

ultimas soluciones exploradas es la que mejor soluciones genera.

4.7. Experimento :Comparación Tiempo entre algoritmos

4.7.1. Comparación Tiempo Heurísticas

En este experimento preliminar se busca comparar el tiempo (milisegundos) que le toma a cada heurística encontrar una solución.

En la Figura 11 podemos observar claramente que la heurística VMC es la mas rápida. Mientras que la heurística HI en cuento a tiempo es la peor de las heurísticas. Su tiempo de ejecución crece de manera muy pronunciada a medida que aumenta el tamaño de entrada. Por ultimo la heurística AGMH también apunta a un crecimiento exponencial pero mas lento que HI.

4.7.2. Comparación Tiempo Metaheurísticas

En este experimento preliminar se busca comparar el tiempo (milisegundos) que le toma a cada metaheurística en encontrar una solución. En este caso se analizaran TSE-AGM y TSC-AGM.

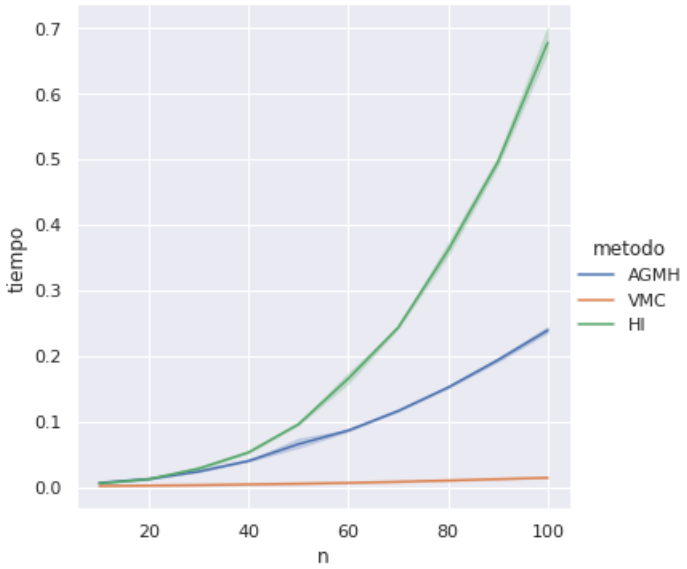


Figura 11: Comparación costo temporal Heurísticas. Para cada entrada de tamaño n, se generaron 100 instancias distintas, para los costos de cada instancia se utilizo la función random(randint), el cual para cada instancia de tamaño n, generaba un array de costos con valores entre $[0, 3 * n]$

Para el caso de la de la Figura 12 podemos observar que si bien ambos tiene un comportamiento exponencial, en el caso de la metaheurística basada en estructuras, el crecimiento es mucho mas lento que la basada en ultimas soluciones explorada.

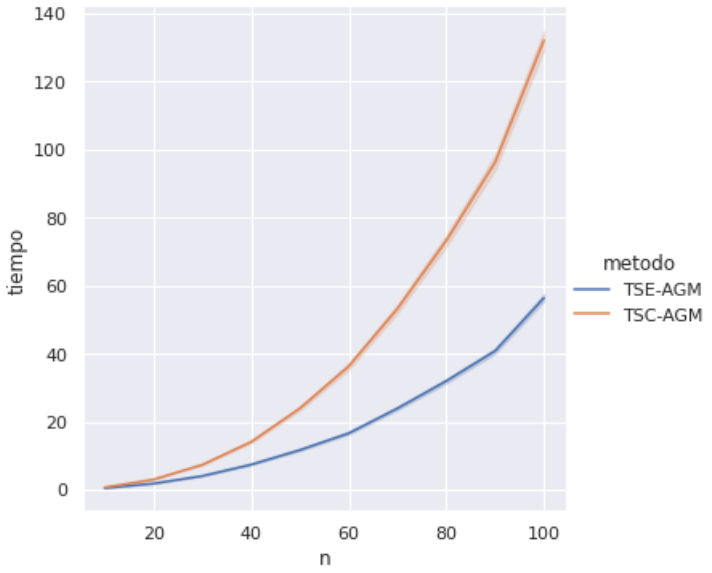


Figura 12: Comparación Tiempo Metaheuristicas Para cada entrada de tamaño n, se generaron 10 instancias distintas, para los costos de cada instancia se utilizo la función random(randint), el cual para cada instancia de tamaño n, generaba un array de costos, donde cada valor equiprobable esta entre $[0, 2 * n]$. Luego para cada una de estas entradas generadas se utilizaron los siguientes parámetros: porcentaje de vecindad p, con p $[10, 20, 30, 40, 50]$, memoria de tamaño t entre $[20, 40, 60, 80, 100]$, i iteraciones, con i entre $[20, 500]$

Con esto podemos concluir que para esta familia de instancias considerando el tiempo, la heurística VMC es una buena opción, mientras que para las metaheurística, la basada en estructuras puede dar la mejor performance.

4.8. Experimento: Comparación Tiempo Vs Costo

En este experimento preliminar buscamos comparar como se comportan las heurísticas en relación tiempo-costo.

Observando la Figura 13 podemos ver que la heurística VMC se mantiene sin cambios notables en cuanto a tiempo de ejecución, para todas las soluciones proporcionadas en esta familia de instancias. En el caso de HI, a medida que aumenta el tamaño de entrada, el costo temporal también aumenta, manteniendo buenas soluciones. En particular, HI obtuvo las mejores soluciones para estas instancias. Finalmente la heurística AGM parece mantener una relación lineal costo-tiempo para todas las entradas.

Se puede concluir que para esta familia de entradas la mejor relación tiempo-costo la mantiene AGMH ,por lo cual seria la mejor opción para esta familia de instancias cuando no tenemos certeza sobre la prioridad en cuanto tiempo o calidad de soluciones. Mientras el resto de las heurísticas se mantienen en polos opuestos.

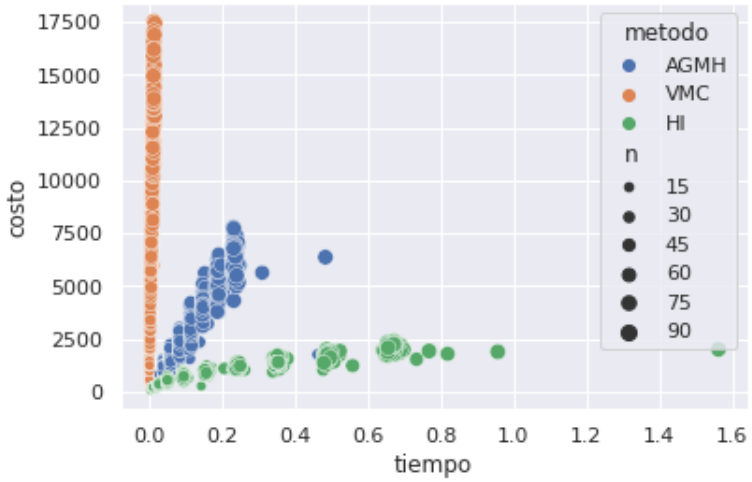


Figura 13: Tiempo vs Costo para Heurísticas, Para cada entrada de tamaño n, se generaron 100 instancias distintas, para los costos de cada instancia se utilizo la función random(randint), el cual para cada instancia de tamaño n, generaba un array de costos con valores entre $[0, 3 * n]$

4.9. Casos Patológicos

4.9.1. Caso Patológico VMC

Los casos patológicos para este algoritmo, son aquellos donde el primer vecino tiene aristas muy costosas tal que al cerrar el ciclo inevitablemente se pasan por ellas. Dado que nuestro algoritmo siempre elige como primer vecino al primer elemento del grafo y le calcula el mínimo, podemos elegir como familia de instancias como casos patológicos aquellos grafos que en el primer nodo tengan n-1 aristas que sean costosas con lo cual al final del ultimo vecino recorrido si o si te toma una arista de estas con lo cual no tenemos una buena solución. En la figura 14 se expone un ejemplo.

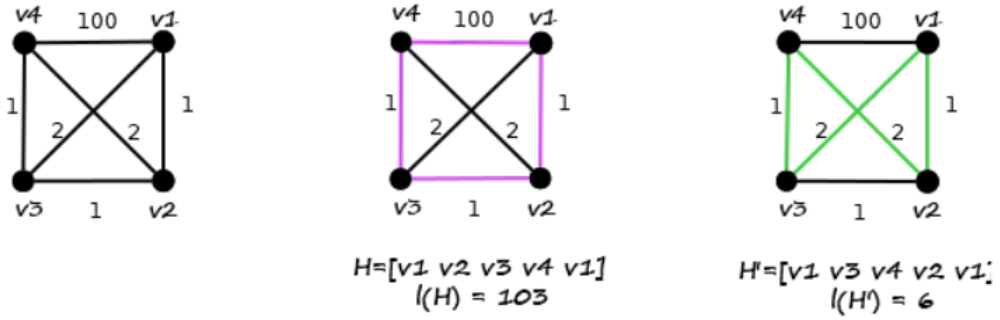


Figura 14: Ejemplo de caso patológico VMC, si se comiendo por v_1, la heurística del vecino mas cercano retorna H(el ciclo violeta) que tiene un costo 103, mientras que la solución óptima H(el ciclo verde) con costo 6.

4.9.2. Caso Patológico AGMH

Un peor caso ocurre cuando se eligen aristas muy costosas para cerrar el ciclo en el paso 3 de la heurística. Notar que las aristas que se agregan no pertenecen al AGM del grafo.

Por ejemplo, las instancias cuyo AGM T es un camino simple entre 2 vértices v y w del grafo forman un peor caso, pues se le puede asignar un peso tan grande como se quiera a la arista entre v y w. Si se define al vértice v como el vértice inicial del ciclo, la heurística elegirá la arista vw para completar el ciclo. Por lo tanto, para este tipo de instancias, la solución hallada por la heurística AGM puede tener un costo tan malo como se quiera. En la Figura 15 se muestra un ejemplo de

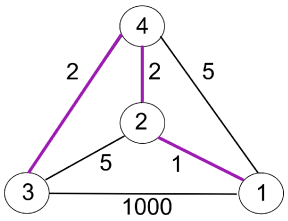


Figura 15: Ejemplo de peor caso para la heurística AGM

peor caso para la heurística AGM, en color se muestra el AGM del grafo. Comenzando desde el vértice 1, el recorrido dfs obtenido es [1,2,4,3,4,2,1]. Finalmente la heurística elige la arista de peso 1000 para cerrar el ciclo.

4.9.3. Caso Patológico HI

En el caso de esta heurística, no tenemos casos patológicos claros, dado que sin importar cual sea el ciclo inicial, en cada iteración en cada iteración determina el mejor vértice a insertar y además elige la mejor ubicación en donde insertar ese vértice, de manera tal que minimice el costo del nuevo ciclo Por lo tanto esta heurística resulta ser la mas solida de las 3.

5. Conclusión

En este trabajo presentamos 3 heurísticas y 2 metaheuristicas útiles resolver el TSP, para cada una de ellas se eligieron distintas instancias para evaluar la efectividad en cuanto a costo y tiempo y combinación de parámetros.

Se comprobó que la utilización de tabu search resulta efectiva para mejorar la calidad de soluciones iniciales al problema. Asimismo, este método introduce nuevos parámetros, de los cuales depende su efectividad como se vio en los experimentos Iteraciones Vecinos e Iteraciones Memoria. Se eligieron los mejores parámetros para tabu search, y utilizando los mismos, se comprobó sobre instancias reales que TSE-AGM obtuvo mejores soluciones que TSC-AGM.

Para el caso del Experimento Comparación Calidad entre Algoritmos, pudimos ver que para las distintas instancias de tamaño n entre $[20, 80]$,con costos entre $[0, 3 * n]$,en cuestión de calidad, podemos elegir HI para las heurísticas, mientras que para las metaheurísticas en cuanto a calidad podemos elegir tabu search basada en ultimas soluciones, usando las instancias mencionadas y las siguientes combinación de parámetros: porcentaje de vecindad entre $[10, 20, 30, 40, 50]$,tamaño de memoria entre $[20, 40, 60, 80, 100]$ e iteraciones entre $[20, 500]$.

Por otro lado en el Experimento:Comparación Tiempo entre algoritmos pudimos ver que para las distintas instancias de tamaño n entre $[20, 80]$,con costos entre $[0, 2 * n]$,para las heurísticas podemos elegir la VMC que posee la mejor performance, mientras que para las metaheurísticas usando las instancias mencionadas y las siguientes combinación de parámetros: porcentaje de vecindad entre $[10, 20, 30, 40, 50]$,tamaño de memoria entre $[20, 40, 60, 80, 100]$ e iteraciones entre $[20, 500]$. Se obtuvo que,la basada en ultimas soluciones exploradas es una buena opción dado que tarda mas en aumentar la pendiente en cuanto a tiempo en el eje y.

En el Experimento Comparación Tiempo vs Costo buscando un Trade-off entre ambas métricas (tiempo y costo) podemos apreciar que heurística AGMH, mantiene una relación lineal entre ambas y siendo así una opción viable ante ambos requerimientos.

Finalmente podemos concluir que con este informe brindamos herramientas para la toma de decisión ante la posible elección de cualquier heurística o metaheurística, basándonos en distintas métricas y entradas posibles.