DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

El Problema del Viajante de Comercio (TSP)

Algoritmos y Estructuras de Datos III

Primer Cuatrimestre de 2021

Integrante	LU	Correo electrónico
Matias Nahuel Castro Russo	203/19	castronahuel14@gmail.com
Juan Manuel Torsello	248/19	juantorsello@gmail.com
Maximo Yazlle	310/19	myazlle99@gmail.com
Gianluca Capelo	83/19	gianlucacapelo@gmail.com

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

http://www.fcen.uba.ar

Índice

1. Introducción

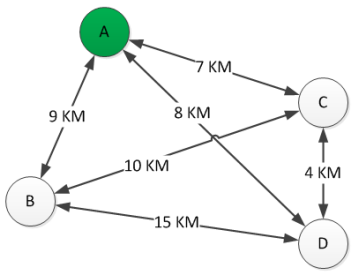
El problema del viajante o TSP por sus siglas en inglés (Travelling Salesman Problem) es uno de los problemas más conocidos en computación. El problema original fue planteado por el matemático irlandes irlandés William Rowan Hamilton y desde su postulación en el siglo XIX hasta hoy no ha perdido importancia, en cambio, es cada vez más relevante. Los desarrolladores siguen investigando cómo mejorar el algoritmo que resuelve este problema dado que es un problema *NP-HARD*. Esto hace referencia a que no existe una solución que tarde tiempo polinomial para resolver el problema. Para dicho problema existen distintos acercamientos en su implementación. En el presente trabajo, implementamos el algoritmo utilizando heurísticas y una metaheurísticas como *Tabú Search* con dos implementaciones. Además experimentamos sus tiempos con distintas instancias para comparar cuál es más eficiente y en qué situaciones.

Como mencionamos antes, el problema del viajante o TSP, es un problema importantísimo en computación. Esto se debe a que no es únicamente un problema abstracto, sino que contamos con numerosas aplicaciones en la vida cotidiana. A continuación vamos a postular algunos de estos problemas:

- Dado una empresa distribuidora de algún bien, producto o servicio, que debe visitar distintos puntos de distribución a lo largo de una ciudad, cuál es la manera de recorrer estos puntos de forma que se tarde el menor tiempo posible o el costo de traslado sea el menor posible.
- En la fabricación de semiconductores, dados circuitos integrados, ver cuál ruta es la mejor para minimizar el costo de tiempo y consumo.
- Dado un avión y distintos aeropuertos con sus rutas aéreas, encontrar el camino que minimice el tiempo o costo de combustible.

Veamos un ejemplo del primer item. Donde hay un Comerciante que se encuentra en su negocio en el punto A, y debe distribuir su mercadería en cada uno de los puntos donde se encuentran sus clientes, por último tiene que volver a su negocio. También sabemos que su vehículo consume bastante combustible, por lo que quiere minimizar los kilómetros a recorrer, con el fin de reducir su costo del recorrido

El siguiente grafo representa los puntos relevantes como nodos, en el nodo A se encuentra su negocio, y en el resto de nodos, cada cliente. Las aristas representan la distancia en km entre dos vértices.



Como podemos observar, hay múltiples recorridos posibles, en general existen $(n-1)!$ rutas posibles, siendo n la cantidad de vértices. En efecto a este caso, hay $(4-1)! = 6$ rutas posibles:

En este caso, el camino más corto es A – B – C – D – A, donde recorre 31 km.

Algo fundamental que observar es que, en un caso chico como este, es relativamente sencillo ver todas las rutas y elegir la mejor. Pero esta dificultad para probar todas las opciones crece exponencialmente con respecto a la cantidad de nodos.

Por ejemplo, agregando 3 clientes más, quedaría $n = 7$. Por lo cual hay $(7-1)! = 5040$ rutas posibles. Si agregamos otros 10 clientes, $n = 17$, por lo que existen $2{,}09227898910^{13}$ rutas. Dado a este crecimiento exponencial, es que para casos que no son chicos, no es factible evaluar todas las posibilidades en busca de la solución óptima, y se opta por desarrollar algoritmos que encuentren soluciones buenas o aceptables, las cuales no necesariamente son soluciones óptimas. Algunos de estos algoritmos son los que implementaremos y analizaremos con detención a lo largo de este trabajo.

2. Metodos

En esta sección se va a presentar cada método implementado, dando no solo un detalle del funcionamiento del algoritmo, sino además, planteando una hipótesis de complejidad para cada uno, acompañado de un pseudocódigo. Para la implementación de todos los métodos se usó la representación de grafos mediante una lista de adyacencias.

2.1. Vecino más cercano

Esta primera heurística es la manera más intuitiva para resolver este problema, en cada paso elegimos como siguiente lugar a visitar, el que se encuentre más cerca del nodo actual, pero que todavía no haya sido visitado. Como se puede notar este es un algoritmo goloso, ya que en cada iteración se toma una decisión solamente teniendo el camino más eficiente para ese momento.

Algorithm 1 Algoritmo de VMC para TSP.

```
1: function VMC(Grafo g)
2:     costo ← 0
3:     v ← 0
4:     H.agregarAtras(v)
5:     while —H— ≤ n do:
6:         w ← buscarmin(g)
7:         Costo ← costo+w.peso
8:         H.agregarAtras(w.dst)
9:     return <H, costo>
```

Una de las ventajas de esta heurística, es que al ser golosa, tiene una implementación bastante sencilla y con un buen tiempo de ejecución, pero al mismo tiempo no nos garantiza nada con respecto a la optimalidad de la solución encontrada. Más aún, podemos encontrar una solución tan mala como quisiésemos modificando el valor de la arista de cierre. Cabe notar que en la realidad todas las aristas de un grafo están dentro de un mismo orden, por lo cual esto tampoco es un gran problema.

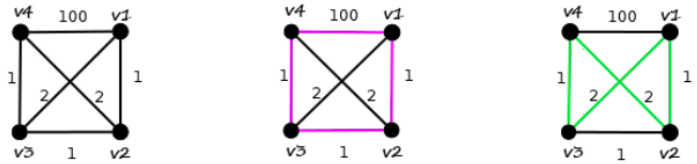


Figura 1: En el grafo central se muestra la solución obtenida por VMC.

En la figura 1 se ve como el método Vecino más cercano obtiene una solución de costo 103, mientras que el costo óptimo del grafo es 6. Además, podemos ver que cambiando el valor de la arista (v_4, v_1) se obtiene una solución tan mala como quisiésemos.

En cuanto a la complejidad de este algoritmo podemos notar que es lineal con respecto a la cantidad de aristas del grafo, ya que en la búsqueda del vecino más cercano se termina recorriendo dos veces cada arista del grafo. Teniendo en cuenta que estamos trabajando sobre grafos completos, podemos analizar la complejidad del algoritmo también en relación con la cantidad de nodos. Podemos notar que por cada vértice, se verifica la distancia con sus n vecinos, por lo tanto la complejidad quedaría en $O(n^2)$. Ahora bien, un grafo completo tiene $(n*(n-1))/2$ aristas por lo tanto es equivalente decir que la complejidad es $O(m)$ o $O(n^2)$. Por último podemos notar que sin importar sea el grafo en cuestión, no podemos ahorrar ninguna búsqueda, por lo tanto la complejidad del algoritmo es $\Theta(n^2)$.

2.2. Inserción

Las heurísticas de inserción consisten en incrementar un ciclo inicial e ir eligiendo, mediante un criterio, el nuevo vértice a insertar e insertarlo en el ciclo hasta haber incorporado todos los vértices a la solución. En general la selección del próximo vértice a incorporar y donde insertarlo son decisiones golosas.

Las implementaciones de inserción varían de acuerdo a la implementación de la función selectora y de inserción. En nuestro caso, comenzamos con un ciclo entre los primeros tres vértices. La función selectora elige el vértice de menor longitud entre los no insertados y los ya insertados. Una vez elegido el vértice v, se inserta en el circuito entre dos vértices consecutivos vi y $vi+1$ de modo tal que:

$$l((vi, v)) + l((v, vi+1)) \;\; l((vi, vi+1)) \text{ sea mínimo.}$$

Algorithm 2 Algoritmo de Inserción para TSP.

```
1:  function INSERCIÓN(Grafo g)
2:      costo ← 0
3:      n ← g.size
4:      insertarPrimerosVertices(g, conexiones, costo, insertado)
5:      while i ≤n do:
6:          w ← elegir(g, insertado)
7:          costo ← costo + insertar(g, w, longitudes, conexiones, insertado)
8:          i ← i+1
9:      agregarVerticesAH(H, g, conexiones)
10:     return <H, Costo>
```

longitudes es un map de par de enteros a enteros instanciado con todas las longitudes entre cada vértice. *insertado* es un vector de longitud n instanciado en false que representa los vértices

insertados. *conexiones* es un map de entero a entero que indica la conexión de un vértice a otro. *agregarVerticesAH* le asigna los vértices al resultado H.

La complejidad de este algoritmo es cúbico con respecto a los vértices. Esto es, $O(n^3)$, ya que en el peor caso el insertar va a recorrer todos los vecinos de dos vértices y esto lo voy a hacer por n vértices. Por otro lado, *insertarPrimerosVertices* son operaciones lineales que se realiza constante veces. Elegir, por otro lado, recorre los insertados, que en el peor caso es $n-1$, y para todo insertado la vecindad de un vértice que es n esto nos da cuadrático. Entonces sumando esto nos queda, como dijimos, $O(n^3)$.

2.3. Árbol generador mínimo

La última de las heurísticas que vamos a ver es basada en un árbol generador mínimo. Esta es 1-aproximada en grafos euclidianos como los nuestros. Este método se divide en 3 partes principales, en primer lugar, usamos un algoritmo para obtener un árbol generador mínimo, para esto decidimos usar Prim, el cual se basa en ir extendiendo un subárbol, buscando el siguiente vértice a menor distancia del mismo, hasta llegar al árbol generador mínimo. El próximo paso consiste en usar DFS el cual recorre el árbol verticalmente dando el orden del camino con el cual armamos el camino hamiltoniano. Por último, buscamos las aristas que unen el camino obtenido por DFS determinando el costo del ciclo obtenido, el cual se termina uniendo el último nodo con el primero, obteniendo el circuito hamiltoniano.

Algorithm 3 Algoritmo de AGM para TSP.

```
1: function AGM(Grafo g)
2:     grafo_agm ← prim(g,0)
3:     camino ← dfs(grafo_agm,0)
4:     costo ← costo_camino(g,camino)
5:     return <camino, costo>
```

En el peor caso, la complejidad del algoritmo es $O(n^2)$. Para ver esto, vamos a hacer un breve analisis de los algoritmos utilizados.

- *Prim:*Dado que estamos experimentando sobre grafos completos, optamos por utilizar la implementación básica de Prim, sin utilizar colas de prioridad. Para esto implementamos el algoritmo visto en la clase del laboratorio, el cual tiene complejidad $O(n^2)$.
- *DFS:* Sabemos que la complejidad de este algoritmo es $O(m)$, siendo m la cantidad de aristas, y como estamos trabajando sobre grafos completos, la cantidad de aristas está acotada superiormente por n^2. Dicho esto, podemos concluir que la complejidad del algoritmo es $O(n^2)$.
- *Costo_camino:* Dado el orden de los n nodos, para cada uno, tenemos que buscar en la lista sus vecinos el próximo nodo del ciclo, lo que tiene un costo de n. Por lo tanto, la complejidad total del algoritmo termina quedando $O(n^2)$.

En conclusión, la complejidad total del metodo es $O(n^2)$. En particular, podemos notar que tanto la complejidad de Prim, como DFS en su mejor caso, es igual a la del peor caso, esto nos permite concluir que la complejidad total del algoritmo es $\Theta(n^2)$.

En cuanto a la optimalidad, en caso de que todas las aristas del grafo cumplan con la desigualdad triangular, podemos dar una cota superior definiendo un valor en el ”peor caso”, el cual corresponde con el doble del valor del árbol generador mínimo. Para ver esto con más

claridad podríamos pensar como se transforma el AGM en un ciclo. Si duplicásemos las aristas del AGM, nos ayudaría a comprender que cada vez que se quiera conectar una hoja del árbol, con el nodo siguiente en el ciclo, el costo de ir directamente al nodo, sería menor o igual que pasar por cualquier nodo intermedio, en particular, haciendo el camino de vuelta por las aristas duplicadas. Teniendo en cuenta que vale la desigualdad triangular, podríamos concluir que el costo va a estar acotado por el doble del costo del árbol generador mínimo.

2.4. Tabu search

Por último, presentamos la metaheurística *Tabu Search*. Esta guía una heurística de búsqueda local para explorar el espacio de soluciones, con el principal objetivo de evitar quedarse atascados en un óptimo local. La base del método es similar a la búsqueda local, ya que iterativamente se mueve de una solución a otra hasta que se cumple algún criterio de terminación. En nuestro caso, la búsqueda local que utilizamos es 2-opt, es decir, que las soluciones se obtienen reemplazando dos aristas del ciclo por otras dos. Este intercambio de aristas es lo que llamamos movimientos.

Cada solución $s \in N(s)$ es alcanzada desde s realizando movimientos, donde $N(s)$ es la vecindad de s. Esto es, un conjunto de partes de las soluciones factibles de s. Nuestra vecindad, es la que nos devuelve 2-opt. Estas son, todas las formas de reemplazar dos aristas en nuestro ciclo actual.

Tabu Search restringe la definición del problema de optimización local a un subconjunto $V \subset N(s)$ con $|V| << |N(s)|$ en una forma estratégica, haciendo uso sistemático de memoria para explotar el conocimiento ya adquirido de la función objetivo $f(s)$ y la vecindad $N(s)$. En el algoritmo que realizamos, el subconjunto de vecinos que elegimos es un porcentaje pasado por parámetro y dichos vecinos son seleccionados aleatoriamente. Luego, en la experimentación, veremos qué selección es la óptima.

La metaheurística recibe ese nombre por la lista que implementa llamada *lista tabú* la cual es la memoria que almacena resultados o movimientos a ser penalizados. Es decir, si un movimiento o atributo está almacenado en esta lista es porque se desea evitar que se repita y el algoritmo no debería repetirlo. Sin embargo, esta técnica cuenta con una *función aspiración* la cual permite, bajo ciertas condiciones definidas, realizar el movimiento o atributo tabú. Particularmente, nuestra función aspiración permite volver a hacer un movimiento ya realizado si este produce un menor costo que el mejor ciclo visto.

Asimismo, contamos con un criterio de parada que nos evita seguir iterando infinitamente, este criterio esta dado por un parámetro que se le pasa a la función.

Los parámetros utilizados son:

- T = Tamaño de la memoria
- max_iter = cantidad de iteraciones máximas
- rango_iter = cantidad de iteraciones máximas en las que no se produjo cambios
- percent = tamaño del subconjunto de la vecindad que explora el algoritmo

En el presente informe vamos a exponer dos implementaciones. Nuestra primera implementación, TS1, guarda en la lista tabú los ciclos. Mientras que la segunda implementación, a la cual haremos referencia como TS2, guarda los movimientos. Es decir, almacena en memoria el par de aristas que fueron extraídas del ciclo.

3. Experimentos

En los experimentos posteriores se ejecutaron los siguientes métodos:

- **VMC:** Algoritmo de Vecino más cercano de la sección 2.1.
- **I:** Algoritmo de Inserción de la sección 2.2.
- **AGM:** Algoritmo de Árbol generador mínimo de la sección 2.1.
- **X-TS1:** Algoritmo de Tabú Search, donde la memoria almacena ciclos, partiendo de la heurística X. Sección 2.4.
- **X-TS2:** Algoritmo de Tabú Search sin memoria, donde la memoria almacena movimientos, partiendo de la heurística X. sección 2.4.

Los métodos de X-TS1 y X-TS2, también recibe los parámetros T, max_iter, rango_iter, percent

3.1. Instancias

Para evaluar y analizar los algoritmos, es preciso usar instancias conocidas, las cuales modelen escenarios reales y sepamos su costo óptimo, para así poder sacar mejores conclusiones. Para esto usamos tres instancias euclidianas conocidas: **berlin52**, **rd100**, **ch150** con 52, 100 y 150 nodos respectivamente. Estas fueron obtenidas de:http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/index.html, y los valores óptimos de http://elib.zib.de/pub/mp-testdata/tsp/tsplib/stsp-sol.html

También definimos familias de instancias conformadas con distintas características. Presentamos los siguientes datasets:

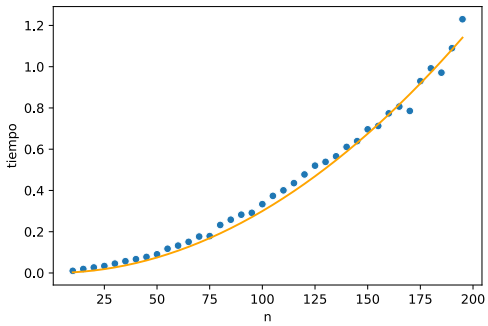
- **instancias-aleatorias**: Grafo completo con n vértices, en el que cada arista tiene un costo aleatorio entre 1 y 300, por lo que puede fácilmente no ser euclidiano.
- **instancias-vmc**: Grafo completo con n vértices, en el que el costo de cada arista es 1, excepto por la arista $(1, n)$ que denominamos arista de cierre.

3.2. Experimento 1: Complejidad de los métodos

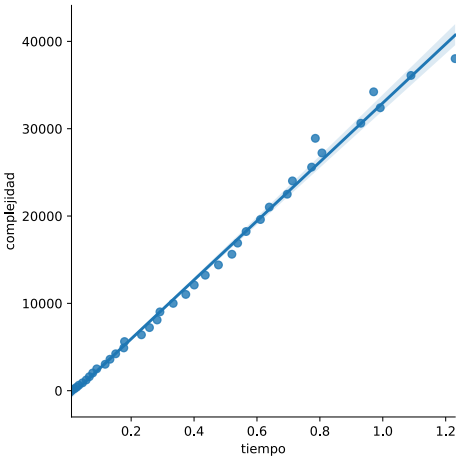
En esta sección vamos a constatar empíricamente las hipótesis de complejidad planteadas en la sección 2. Para esto generamos un dataset de instancias aleatorias, con el objetivo de poder analizar el comportamiento de cada algoritmo a medida que aumenta la cantidad de nodos del grafo. Cabe aclarar que cuando hablemos de n nos referimos a la cantidad de nodos del grafo.

3.2.1. Vecino más cercano

En primer lugar vamos queremos constatar la de complejidad de la heurística Vecino más cercano, en la cual planteamos que esta debería ser $O(n^2)$. En la figura 2a se puede ver que los tiempos de ejecución se comportan de la misma manera de la curva de complejidad esperada.



(a) Tiempo de ejecución de *instancias aleatorias* con el método VMC contra cota teórica n^2.



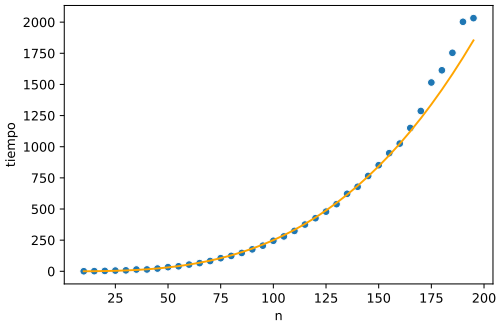
(b) Correlación entre tiempo de ejecución y la complejidad esperada de VMC.

Figura 2: Complejidad heurística Vecino más cercano.

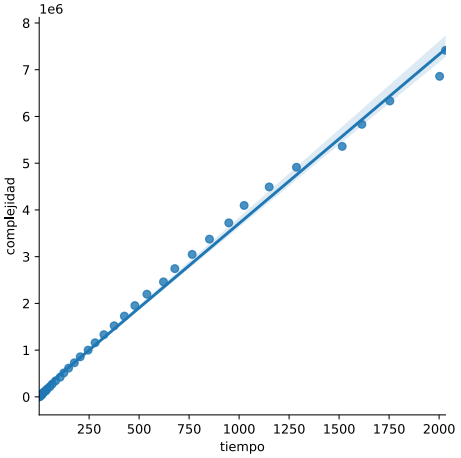
Para ver esto con más detenimiento, en la figura 2b se puede ver la correlación entre el tiempo de ejecución y la complejidad esperada, parecería que ambas están muy correlacionadas, lo cual se confirma al calcular el coeficiente de Pearson cuyo valor es 0.9970. Esto nos permite afirmar que la cota teórica de complejidad que se anunció en la sección 2.1 es correcta.

3.2.2. Inserción

En esta sección queremos ratificar la hipótesis de complejidad del método goloso, Inserción. En la sección 2.2 nuestra hipótesis fue que el algoritmo tiene una cota superior de complejidad $O(n^3)$. La figura 3a muestra como la curva de tiempo de ejecución se acopla con la curva correspondiente a la complejidad esperada, en este caso n^3. Para ratificar esto calculamos el coeficiente de Pearson entre el tiempo de ejecución y la complejidad esperada, cuyo valor es de 0.9937.



(a) Tiempo de ejecución de *instancias aleatorias* con el método I contra cota teórica $O(n^3)$.

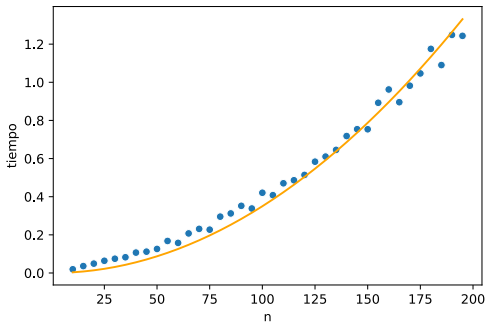


(b) Correlación entre tiempo de ejecución y la complejidad esperada de I.

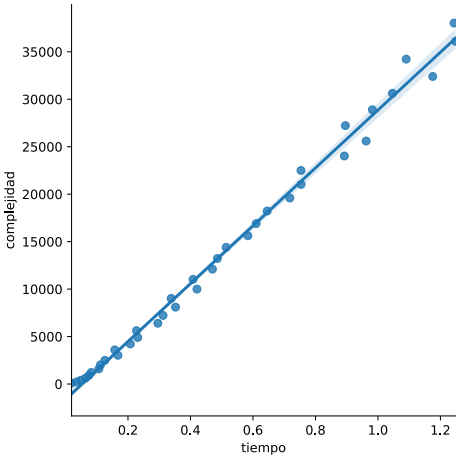
Figura 3: Complejidad heurística Inserción.

3.2.3. Árbol generador mínimo

Por último vamos a analizar la complejidad el método del árbol generador mínimo, para este caso planteamos una cota superior de complejidad $O(n^2)$. En la figura 4a se puede ver que los tiempos de ejecución crecen cuadráticamente.



(a) Tiempo de ejecución de *instancias aleatorias* con el método AGM contra cota teórica n^2.



(b) Correlación entre tiempo de ejecución y la complejidad esperada de AGM.

Figura 4: Complejidad heurística Árbol generador mínimo.

Una vez más, ratificamos esto calculando el coeficiente de Pearson entre el tiempo de ejecución y la complejidad esperada, el cual vale 0.9967. Con esto corroboramos nuestra hipótesis de complejidad planteada en la sección 2.3.

3.3. Experimento 2: Búsqueda de parámetros óptimos de Tabu Search

El objetivo de esta experimentación es obtener los parámetros óptimos de **TS-1** y **TS-2** para las diferentes heurísticas. Una aclaración importante es que por parámetros óptimos no nos referimos a los que consigan el ciclo de costo menor, sino a los que muestren la mejor relación tiempo de ejecución - calidad. Se experimentó sobre el dataset **berlin52** cada heurística, utilizando los siguientes valores para cada parametro:

- *largolista* :[50 , 100 ,200]
- *max_iter*: [50 ,200 ,500 , 1000]
- *rang_iter*: [50 , 100 , 300]
- *percentage*:[5 , 30 , 50 , 100]

En la tabla 1 se detallan los parámetros que encontraron los ciclos de menor costo.

Método	Memoria	%Vecinos	Iter	Rango Iter	gap
AGM-TS1	200	100	1000	50	0.0336
AGM-TS2	200	100	1000	50	0.0212
VMC-TS1	200	100	1000	300	0.0312
VMC-TS2	100	100	1000	50	0.0298
I-TS1	50	100	1000	300	0.0319
I-TS2	200	100	1000	300	0.0356

Cuadro 1: Parámetros que dan como resultado el camino con menor costo.

Como podemos ver en los gráficos 5, los dos parámetros que más afectan a la complejidad son la cantidad de iteraciones y la cantidad de vecinos visitados. Por esta razón nos dedicaremos a analizar cómo se comportan nuestras meta-heurísticas al variar estos parámetros, dado que los otros no afectan notoriamente el tiempo de ejecución para los valores con los que se experimentó.

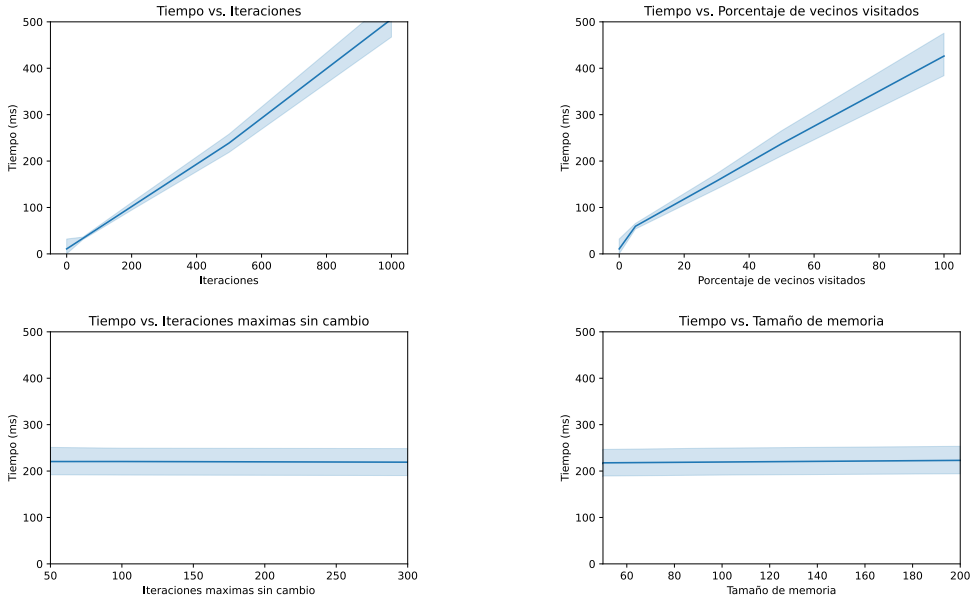


Figura 5: Variación del tiempo respecto de los distintos parámetros de Tabú Search.

En la figura 6 se observa una correlación positiva entre el porcentaje de vecinos visitados y el gap para cada método, En particular las mejores soluciones fueron cuando se visitan el cien por ciento de los vecinos.

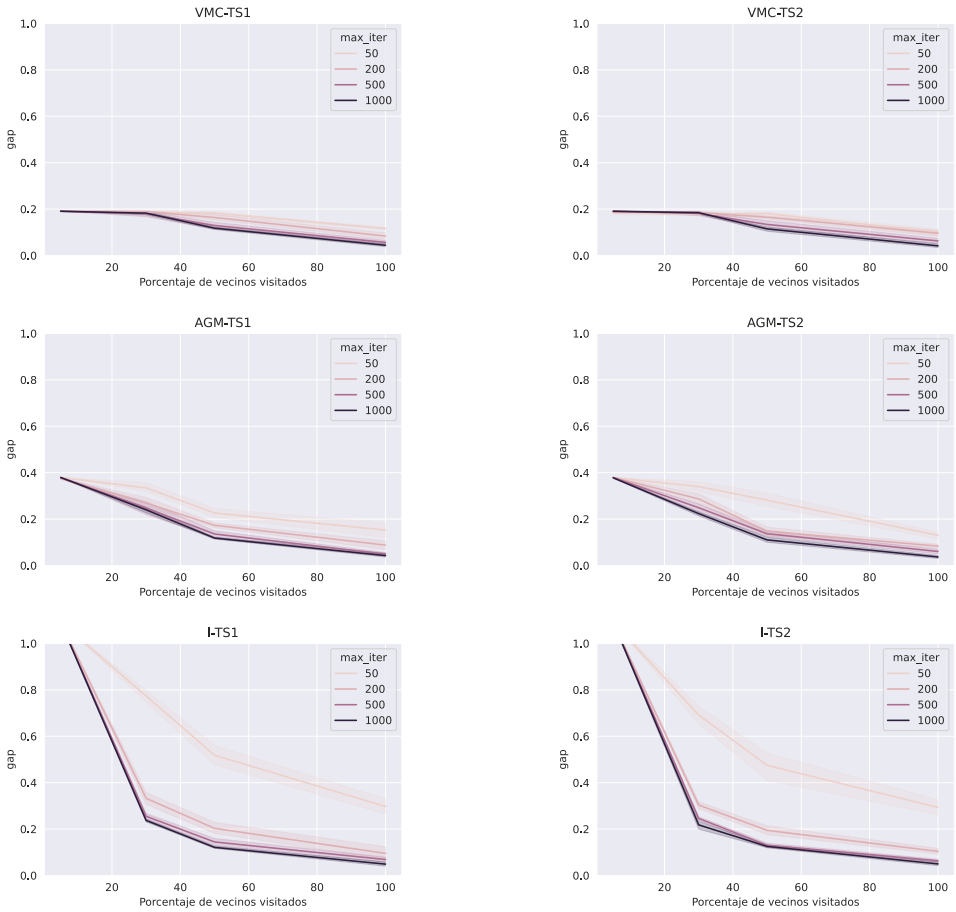


Figura 6: Relación gap-porcentaje de vecinos distinguiendo la cantidad de iteraciones.

También se puede observar en la Figura 7 una relación positiva entre la cantidad de iteraciones y el gap, pero no es tan fuerte como la primera enunciada. En particular en ninguno de los métodos no se notan cambios significativos a partir de las 500 iteraciones.

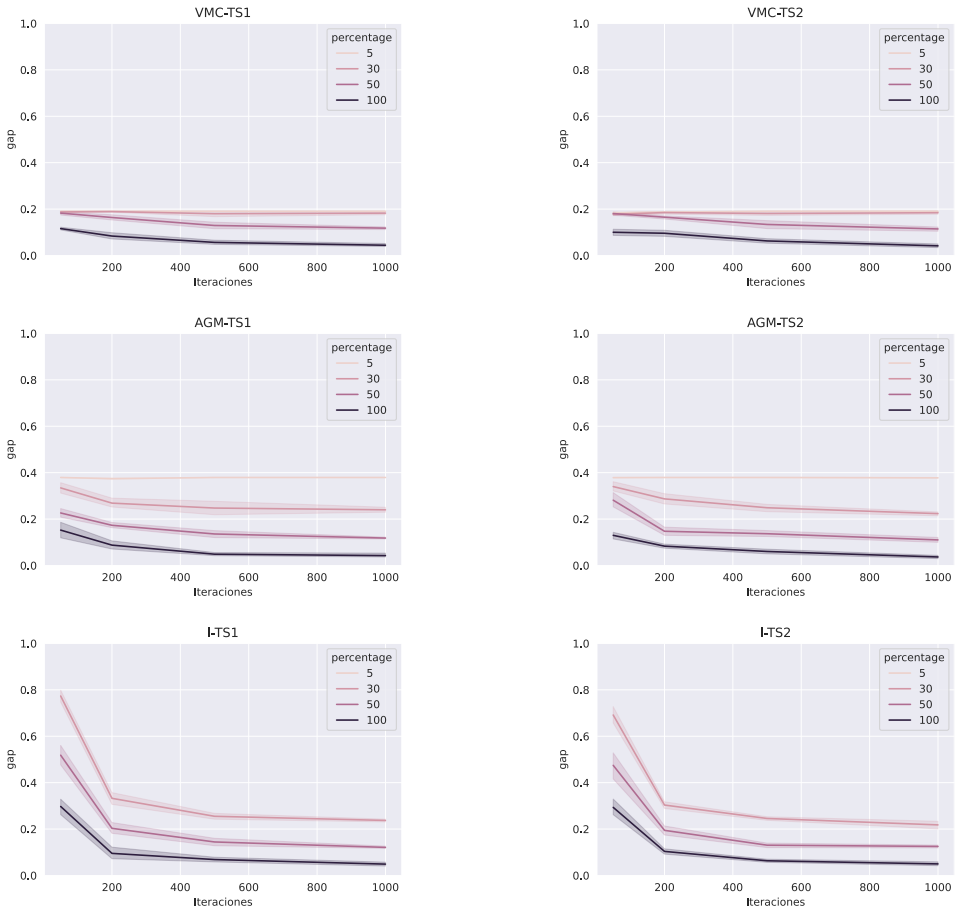


Figura 7: Relación de gap-iteraciones distinguiendo los porcentajes de vecinos.

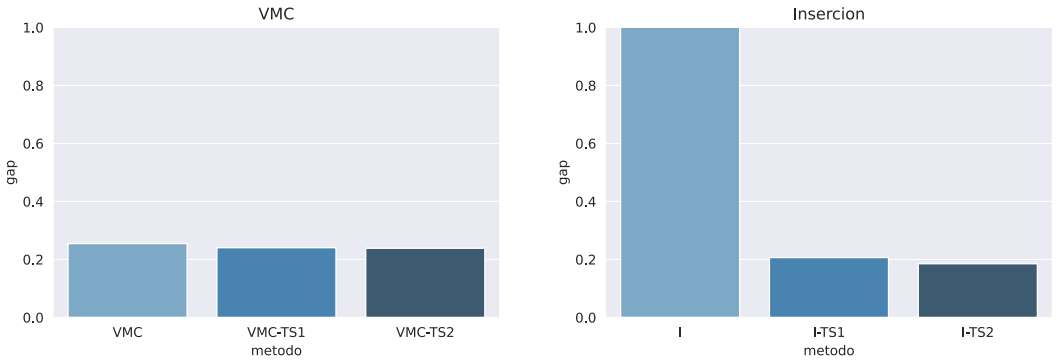
Como se enunció al principio buscamos los parámetros que tenga el mejor trade-off calidad -tiempo, por esta razón se definió un criterio para determinar si era necesario ver la cantidad máxima de vecinos o si viendo menos el resultado ya era suficientemente bueno. El criterio de selección del parámetro *porcentaje de vecinos* es el siguiente: si la diferencia entre gap obtenido viendo el $100\,\%$ de los vecinos y una cantidad menor es a lo **0.06**, entonces nos quedamos con este último. Por último en los gráficos 6 y 7 no se notan grandes diferencias a partir de las 500 iteraciones en cada uno de los métodos, por esta razón utilizaremos como parámetros óptimos de nuestros métodos aquellos con 500 iteraciones. En la tabla 2 se definen los parámetros óptimos bajo nuestros criterios.

Método	Memoria	%Vecinos	Iter	Rango Iter	gap
VMC-TS1	100	50	500	50	0.0845
VMC-TS2	100	50	500	300	0.0808
AGM-TS1	200	50	500	50	0.0926
AGM-TS2	100	100	500	300	0.0383
I-TS1	100	100	500	300	0.0396
I-TS2	50	100	500	50	0.0470

Cuadro 2: Parámetros óptimos de Tabú search.

3.4. Experimento 3: Comparación

En esta sección analizaremos cómo se comportan nuestros métodos utilizando los parámetros definidos en la sección 3.3 con un nuevo dataset **ch150**. Nos interesa saber si es que mejora y cuanto, para ambas implementaciones de Tabú Search con cada método. Además, queremos ver si la elección de parámetros generaliza bien a esta nueva instancia.

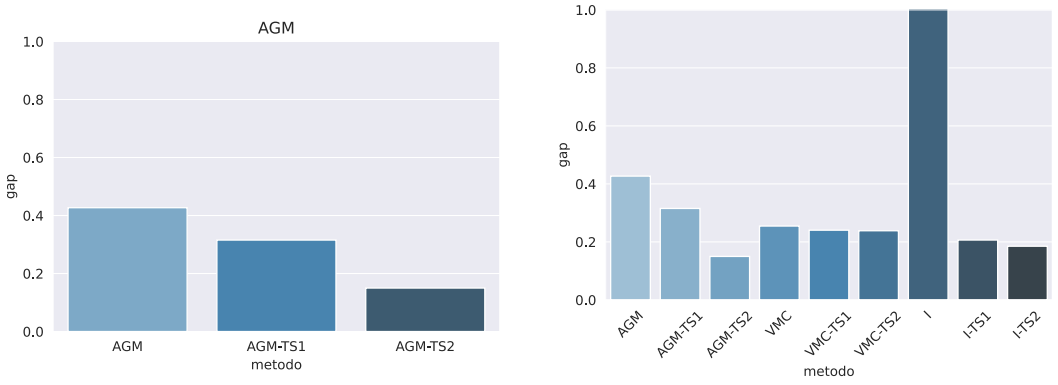


(a) Gap para métodos que utilizan VMC. (b) Gap para métodos que utilizan Inserción.

Figura 8

Un resultado interesante es que la heurística del vecino más cercano a pesar de ser la más simple fue la que mejor ciclo obtuvo con respecto a AGM e Inserción. Pero al aplicar la heurística de mejoramiento(Tabu Search) no se notaron grandes mejoras como podemos ver en la figura 8a. Una de nuestras hipótesis, es que esto sucede por como construye su solución. Creemos que aplicando Tabú Search, por como es la solución obtenida por VMC, se estanca muy rápido en un óptimo local, no permitiendo salir en búsqueda de un óptimo global. De no ser verdadera también se podría asumir que la elección de nuestros parámetros no generaliza bien para VMC.

En inserción sucede exactamente lo contrario a vecino más cercano, esta comienza con una solución muy pobre con respecto a sus pares, pero es la que tabú mas logra mejorar. Más aún, se puede ver que este método, con ambas implementaciones de Tabú Search logra una solución más óptima que todas las de VMC. Viendo la figura 8b, se ilustra la diferencia entre la solución inicial, y luego de aplicarle nuestra meta-heurística.



(a) Gap para métodos que utilizan AGM. (b) Comparación gap entre todos los métodos para la instancia ch150.

Figura 9

Por último, pero no menos importante, en la heurística de árbol generador mínimo podemos observar en 9b que la implementación de Tabú Search que guarda movimientos es el método que más se acerca al óptimo. Sin embargo en la figura 9a se ve que **TS-1** no logra optimizar la solución de la misma manera que la otra.

A partir de esto planteamos el siguiente experimento para dilucidar si esto se debe a una característica intrínseca de este método, o si es causa del criterio tomado en la sección previa (3.3). Recordamos que en la misma se decidió en base a los gaps obtenidos, que este método recorra el 50 por ciento de los vecinos. El experimento consta de comparar la solución obtenida previamente para la misma instancia, con el método **AGM-TS1** recorriendo el $100\,\%$ de los vecinos.

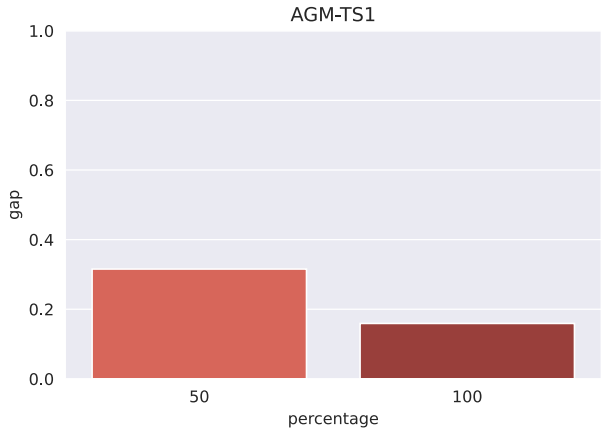


Figura 10: Comparación de gap entre TS1 con $50\,\%$ y $100\,\%$ de los vecinos.

Los resultados de este se pueden observar en la figura 10, donde notamos una gran mejora en el método cuando se recorren todos los vecinos, por ende se puede asumir que el criterio tomado en este caso no aplica correctamente en esta instancia.

3.5. Experimento 4: AGM en grafos no euclidianos

En la sección 3.4 no nos centramos en la comparación de las tres heurísticas sin la utilización de Tabú Search. No obstante, parecería que AGM y VMC obtienen una solución bastante más cercana al óptimo que Inserción. En la figura 11 se puede corroborar lo dicho previamente para tres instancias de grafos euclidianos: **berlin52**, **rd100** y **ch150**.

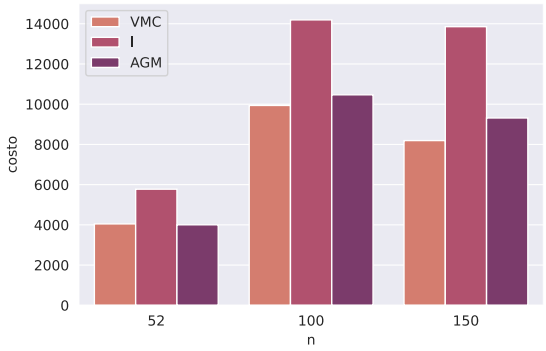


Figura 11: Costo de cada heurística para 3 instancias euclidianas.

Como se enunció en la sección 2.3 sabemos que AGM tiene una cota en grafos euclidianos la cual nos asegura una solución considerable. Ahora bien, en caso de que no estemos trabajando en este tipo de grafos, AGM debería empeorar considerablemente su rendimiento, mientras que los algoritmos golosos no deberían verse muy afectados. En este experimento vamos a ver como se comportan las heurísticas para instancias generadas aleatoriamente.

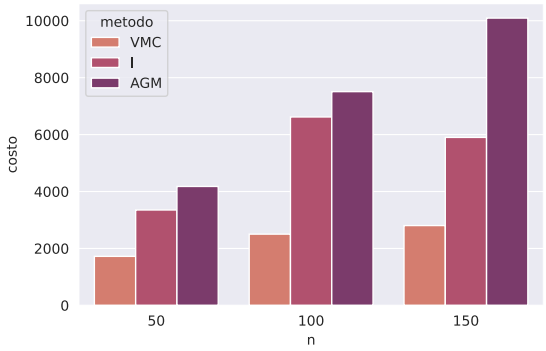


Figura 12: Costo de cada heurística para 3 instancias aleatorias.

En la figura 12 se puede ver que AGM obtiene una peor solución en comparación con los algoritmos golosos. Con respecto a los dos métodos restantes, vemos que tienen un rendimiento similar a los obtenidos en instancias euclidianas, sin VMC el de mejor solución e Inserción obteniendo una solución menos óptima.

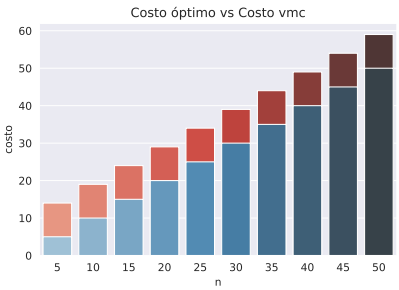
3.6. Experimento 5: Vecino más cercano

Buscamos mostrar el caso mencionado en la sección 2.1, en donde la heurística de vecino más cercano nunca encuentra el óptimo. En particular podría terminar dependiendo de una única arista, la que cierra el ciclo.

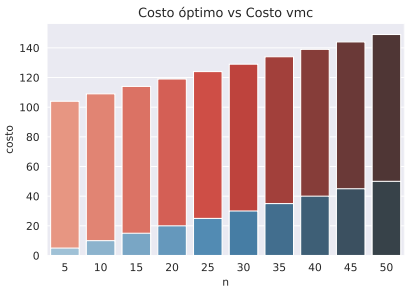
Para constatar esto creamos un dataset en el cual las instancias representan un grafo completo de n vértices, en el que el peso de todos las aristas es igual a 1, excepto por la arista (1,n) que denominamos arista de cierre.

En el gráfico 13 podemos ver la solución de la heurística para tres valores de aristas de cierre distintas. Podemos notar que siempre vecino más cercano por como es la estructura del grafo utiliza esta arista en su ciclo, de modo que el costo final siempre termina dependiendo de esta arista de cierre.

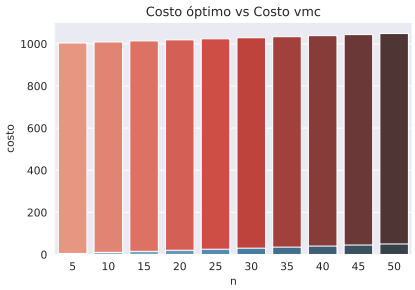
En el gráfico 13a como la arista cierre es 10 no difiere mucho de las soluciones óptimas, en cambio, en las figuras 13b y 13c se puede ver como el valor de esta heurística puede ser tan malo como esta arista en cuestión.



(a) Arista de cierre con costo 10.



(b) Arista de cierre con costo 100.



(c) Arista de cierre con costo 1000.

Figura 13: En azul se muestran los costos óptimos y en rojo los obtenidos por VMC.

4. Conclusión

Habiendo visto y analizado las distintas implementaciones para resolver el problema del viajero, podemos concluir algunas características de cada método:

- El metodo de insercion no demostro una buena performance en su solucion inicial, ademas este era el de peor complejidad entre las tres heuristicas. Pero se noto una mejora considerable en la aplicación de la metaheurística de tabú search.
- El acercamiento por vecino más cercano es una buena solución inicial, pero no es considerable la mejora con tabú search.
- De las distintas estrategias utilizadas, AGM parece ser el más robusto dando una buena solución inicial y siedno la que más cerca del óptimo está cuando se le aplica la metaheuristica Tabú search. Sin embargo, es el peor en calidad con grafos no euclidianos.

A futuro:

- Experimentar sobre AGM utilizando grafos no euclidianos provenientes de instancias reales. Dado que una de nuestras hipótesis fue que esta heurística no se desempeña bien en estas ocasiones y solo se experimentó sobre instancias aleatorias.
- Probar otros tipos de memoria, esto nos interesa porque nuestras implementaciones no lograban mejorar viendo solo una porción de los vecinos. Lo que en particular no sería viable en instancias muy grandes.