DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Coloreo de impacto máximo

Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Gómez, Bruno Agustín	428/18	brunolml99@outlook.es

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

En el presente trabajo, vamos a tratar el *Problema de Coloreo de Máximo Impacto (PCMI)*. El dominio del mismo consiste en dos grafos con el mismo conjunto de vértices, $\mathbf{G} = (V, E_G)$ y $\mathbf{H} = (V, E_H)$, donde $|V| = n$, $|E_G| = m_G$ y $|E_H| = m_H$. Por otro lado, se define el impacto $I(c)$ sobre H de un coloreo $c : V \longrightarrow C$ de G como el número de aristas $(i,j) \in E_H$ tales que $c(i) = c(j)$. El objetivo de PCMI es encontrar aquel coloreo válido de G tal que se obtenga el máximo impacto sobre H.

Durante el desarrollo, denominaremos una *solución* a las posibles distribuciones de colores sobre el grafo G, mientras que una *solución factible* consistirá de los posibles *coloreos* sobre G, es decir, aquellas distribuciones en las que no exista ninguna arista que una dos vértices que tengan el mismo color. En la figura 1 se visualizan distintas soluciones para un ejemplo de PCMI con V = {1,2,3,4,5,6}, E_G = {<1,2>, <1,3>,<2,4>,<2,6>, <3,5>} y E_H = {<1,4>, <3,4>,<2,5>, <3,6>, <5,6>}.

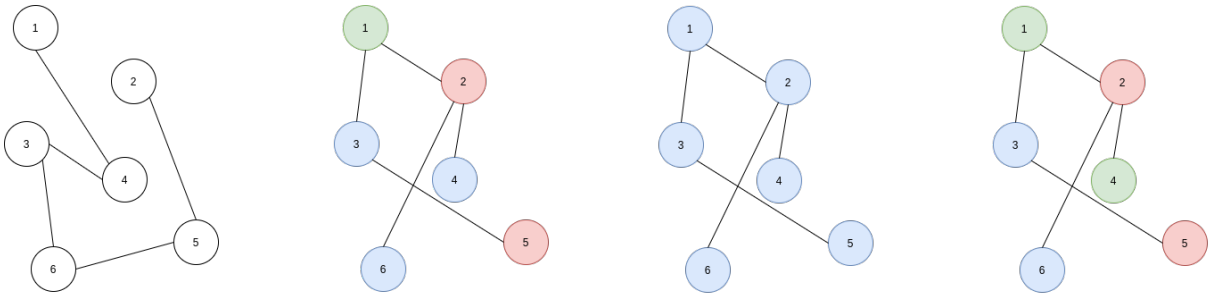

Figura 1: Ejemplo de soluciones posibles al problema. De izquierda a derecha se muestran el grafo H y tres coloreos sobre G: una solución óptima, otra óptima pero no factible y finalmente una factible pero no óptima.

Una de las aplicaciones del PCMI es el problema de asignar aulas a clases del mismo curso, considerando que las clases que se superponen en horario se dan en aulas distintas. En este caso, los vértices de G van a representar las clases y las aristas del grafo van a unir los pares que no pueden darse en la misma aula, pues se solapan los intervalos de tiempo en los que son dados. Por otro lado, los colores van a representar las aulas asignadas y el grafo H tendrá como conjunto de aristas a las clases que sean del mismo curso; con este planteo, maximizar el impacto de G en H nos permitiría saber cómo tener la mayor cantidad de clases del mismo curso en el mismo aula, de modo que los alumnos no tengan que moverse una vez finalizada la clase.

Este problema tiene la particularidad de pertenecer a la clase *NP-Hard* lo cual nos indica que como mínimo puede ser tan difícil como un problema de NP. Gracias a esto, no podemos optar por un algoritmo de resolución exacta para este problema dado que en principio su complejidad asociada va a ser potencialmente prohibitiva. Acá es donde entran los algoritmos heurísticos, los cuales por definición son aquellos que están diseñados para resolver un problema de una manera más rápida y eficiente que los métodos tradicionales sacrificando la optimización, la precisión o la integridad por la velocidad. Considerando esto para llevar a cabo la resolución de PCMI vamos a análizar enfoques *heurísticos* para construir las soluciones y *metaheurísticos* para explorar las soluciones en busca de una mejor aproximación para la solución óptima.

A continuación vamos a presentar tres heurísticas constructivas golosas distintas para obtener soluciones factibles y una metaheurística *Tabú Search* para lograr una búsqueda más amplia en el espacio de soluciones obtenidas y así poder acercarnos más a la solución óptima.

2. Desarrollo

En esta sección, explicaremos las distintas heurísticas implementadas para la resolución de *PCMI*. En primer lugar, se profundizará sobre el conjunto de métricas elegidas al momento de evaluar el desempeño de cada algoritmo y el contexto de uso de las mismas. Más adelante, se detallarán funciones auxiliares y variables globales utilizadas para la modularización durante el desarrollo en completo. Para finalizar, se presentarán las heurísticas implementadas, junto con el respectivo desglose de las mismas y sus variantes, el estudio de la complejidad algorítmica y un análisis del desempeño bajo los indicadores elegidos en la Sección 2.1.

2.1. Métricas para evaluación de desempeño

Los algoritmos presentados ofrecerán una solución aproximada al problema del *PCMI*, dado que este es perteneciente a la clase *NP-hard* cuyos problemas son “al menos” tan complejos como los de *NP*. Para ello, utilizaremos la medición *gap*, o diferencial.
Se define gap absoluto como el porcentaje que representa la diferencia entre el impacto real y el impacto obtenido por una heurística para una misma instancia. Esta métrica se utilizará en los casos donde el impacto real esté precalculado, siendo estos los provistos por la cátedra. Al ser un conjunto reducido y exacto, se utilizarán en las mediciones de calidad de cada heurística.
Por otro lado, el grueso del análisis en la sección 3 se realizará sobre instancias generadas en lotes sin un precálculo disponible del impacto exacto y ,dado el índole de *PCMI*, realizar el cálculo en tiempo real es potencialmente prohibitivo dada la índole del problema. Para ello, se define gap relativo como la diferencia entre una cota superior para el impacto posible y el impacto obtenido por una heurística.
Considerando el impacto como la cantidad de aristas de H cuyos vértices están coloreados del mismo color en G, la cota superior

para el impacto exacto será $E_H \setminus E_G$, ya que una arista de H que también pertenezca a G no podrá tener sus vértices coloreados con el mismo color.

2.2. Variables globales y funciones auxiliares

En esta sección se describirán nomenclaturas y funciones auxiliares comunes a los enfoques presentados, a modo de unificar la notación durante los mismos.

En primer lugar, al final de la ejecución de cada heurística, en la variable **ordenCol** se salvará el orden de coloreo correspondiente con el impacto devuelto. Así, el color resultante del i-ésimo vértice se corresponderá con el color asignado por $ordenCol[i-1]$.
En segundo lugar, la función **ImpactoDeColoreo** es utilizada para calcular el impacto referido por un determinado coloreo y un grafo a comparar. Como parámetros de la función, tomará un coloreo de vértices y el grafo H. Posteriormente, recorrerá secuencialmente las aristas de H llevando el conteo de las que son incidentes a vértices que poseen el mismo color.
Respecto a su complejidad, el ciclo principal realiza comparaciones, asignaciones y sumas, operaciones cuya complejidad temporal es constante. Como resultado, esta función auxiliar es de orden $O(m_H)$.

Para realizar el ordenamiento de vértices que alimentarán luego a coloreos secuenciales, se utiliza la función **sort** proveniente de la *C++ standard library*. Este es un algoritmo híbrido, utilizando internamente múltiples algoritmos de *sorting*, y recibe un vector a ordenar, junto con una función de comparación de manera opcional. Para el presente trabajo, se utilizará la función para ordenar el conjunto de vértices, bajo los criterios de comparación de grado en G y grado en H. Dichas comparaciones, contando con las listas de adyacencia en cada grafo, resultan constantes. Bajo estas condiciones, la complejidad de este algoritmo es de O(D*log(D))[1], donde D representa la distancia entre el primer y último elemento del arreglo a ordenar. En nuestro contexto de aplicación D va a ser igual a la cantidad de vértices en los grafos, es decir, n. Así, la complejidad para estos casos en específico, será de $O(n * log(n))$.

2.3. Heurística golosa secuencial

El primer algoritmo desarrollado tiene como esencia una heurística secuencial. Ésta, dado un orden específico de los vértices pertenecientes al grafo a estudiar, realiza un coloreo sobre el grafo con un enfoque intuitivo: se recorre linealmente dicho orden, coloreando cada vértice con el mínimo color disponible que resulte en un *coloreo* válido, es decir, que ningún vértice adyacente al actual tenga el mismo color.

De esta forma, si bien todo criterio de ordenamiento recibido por la heurística secuencial producirá un coloreo, un aspecto clave para la resolución de este problema en particular recae en la selección de este criterio.
En la línea 2 elige entre dos criterios posibles que luego alimentarán a la heurística golosa: como primera opción, un orden determinado de mayor a menor respecto al grado de cada vértice en el grafo G; en segundo lugar, un orden también por grado de cada vértice de mayor a menor en el grafo H. Ambos criterios en caso de empate decidirán por el número de vértice más grande entre los pares a comparar.

Estos criterios de ordenamiento fueron seleccionados de modo que siempre coloreemos primero los vértices de mayor grado, aumentando así, las posibilidades de que dos vértices en H tengan el mismo color. Cada orden lo logra por distintas intuiciones generadas sobre el problema, en el caso del orden por grado en G al usar el *Largest First Order*, el cual como vimos en la teorica, genera un coloreo con una utilización menor de colores que un orden tradicional. Al utilizar menos colores, este orden incrementa las chances de que haya dos vértices en H que posean el mismo color y sean incidentes. Sin embargo, por otro lado el segundo orden establecido, como se mencionó anteriormente, utiliza un ordenamiento por grado en H el cual incrementa las probabilidades de que dos vértices adyacentes en H tengan un mismo color asignado. Ambos criterios intentan lograr el objetivo con diferentes enfoques.

Por otra parte, en la línea 5 se introduce una mejora extra, para una pregunta que nos surgió posteriormente. Para la comparación entre una búsqueda local golosa y una búsqueda por *Tabú search*, que tiene cómo objetivo mostrar el impacto que posee la utilización de una *Metaheurística* como lo es *Tabú search* en la exploración del espacio de soluciones. Sin embargo, esta mejora no forma parte de la ejecución tradicional de las heurísticas secuenciales golosas dado que de lo contrario dejarían de ser *Heurísticas constructivas*.

Algorithm 1 Algoritmo de Heurística golosa secuencial.

```
1: function heuristicaSecuencial(loc, gr)
2:     if gr then crit ← gradoG else crit ← gradoH
3:     ordenar(V, crit)
4:     mCol ← coloreoSecuencial(V,ordenCol,G)
5:     if loc then
6:         max ← impactoDeColoreo(ordenCol,H)
7:         busquedaLocalGolosa(mCol,max,ordenCol,H,G)
8:     return impactoDeColoreo(ordenCol,H)
```

Para el peor caso de esta heurística, se debe considerar las dos subdivisiones del algoritmo.

- Si el orden elegido es por grado en G, se genera un grafo G al azar. Luego, ordenamos los vértices de mayor a menor según su grado en G y, si llamamos a este orden $\{v_1, v_2, ..., v_n\}$, generamos aristas en H tal que $(v_i, v_{n-i}) \in E_H$, donde $1 \leq i \leq n/2$.

[1] https://en.cppreference.com/w/cpp/algorithm/sort#Complexity

Al ser el coloreo secuencial con un orden en específico, tener una distancia de $\frac{n}{2}$ para cada par de vértices conectados por una arista en H aumenta ampliamente la probabilidad de que no sean coloreados igualmente en G. Esto reducirá entonces el impacto final.

- Como contraparte, si el orden elegido es por grado en H, el procedimiento es similar. Se genera un grafo H al azar y se ordenan los vértices de mayor a menor, de acuerdo a su grado en el mismo, resultando en un orden $\{v_1, v_2, ..., v_n\}$. Así, generamos aristas en G tal que $(v_i, v_{i+1}) \in E_G$, donde $1 \leq i \leq n-1$. Cuanto más cercanos sean en el orden a colorear, más propenso será que el coloreo secuencial los colore de distinta manera, reduciendo el impacto generado por esta resolución.

De manera análoga, para el mejor caso del algoritmo se consideran ambos ordenes por separado:

- Si el orden elegido es por grado en G, generamos un G de manera aleatoria y, si llamamos a este orden $\{v_1, v_2, ..., v_n\}$, generamos aristas en H tal que $(v_i, v_{i+1}) \in E_H$, donde $1 \leq i \leq n-1$. Aprovechando las propiedades del coloreo secuencial, esto hace que las aristas resultantes en H sean más propensas a tener el mismo color. De esta manera el impacto se va a ver incrementado.

- Mientras que si el orden es por grado en H, generamos un H de manera aleatoria y, si llamamos a este orden $(v_i, v_{n-i}) \in E_G$, donde $1 \leq i \leq n/2$. Al ser el coloreo secuencial, los vértices más cercanos en el orden de coloreo van a ser más propensos a colorearse con el mismo color. Y gracias a que las aristas en G, son incidentes a vértices que se encuentran mayormente distanciados dentro del orden de coloreo. Considerando esto, los primero vértices a colorear van a aumentar sus chances de producir coloreos válidos dado que en principio no serían aristas adyacentes en G y por igual, al tener el mismo color en H el impacto aumentará significativamente.

2.3.1. Calidad de la solución

Para la evaluación, se utilizaron los métodos **HS1**, **HS1B**,**HS2** y **HS2B**: El primer par evaluará la heurística con un ordenamiento por grado de mayor a menor de G, sin y con búsqueda local respectivamente, mientras que el siguiente par ordena de la misma forma por grado en H, también sin y con búsqueda local.
En el siguiente cuadro se visualizan los resultados para cada una de las 30 instancias utilizadas.

Instancia	Impacto optimo	Impacto HS1	Impacto HS1B	Impacto HS2	Impacto HS2B
n6	1	0	1	1	1
n8	6	5	5	1	3
n10	3	1	3	1	3
n12	16	8	8	7	10
n14	12	6	7	6	6
n16	20	6	8	6	9
n18	27	7	11	9	12
n20	25	5	9	10	10
n22	26	10	14	9	13
n24	33	6	11	10	15
n26	38	7	13	9	16
n28	48	15	16	14	16
n30	47	10	14	11	17

Como se puede visualizar, el gap producido por el diferencial entre impactos en todas las ejecuciones es considerablemente alto. La ejecución que más nos aleja del óptimo es HS1, con un gap en promedio de **67,678 %**. A ello le sigue HS2, con un diferencial de **62,441 %**. Al introducir las mejoras por búsqueda local, podemos ver una mejora significativa, resultando en HS1B con un gap de **46,698 %**, y HS2B como el más cercano, con un diferencial de **46,230 %**. Por otro lado, en la Sección 3 vamos a ahondar más en esta discusión sobre la calidad de estas heurísticas. Ejecutándolas sobre distintas familias de instancias.

2.3.2. Complejidad

En cuanto a complejidad, el algoritmo presentado consta de cuatro llamados a funciones externas, sumadas a operaciones con tiempo insumido constante, como comparaciones y asignaciones.
En la línea 3 se ordenan los vértices del grafo H que, como se mencionó en la sección 2.2, tiene un costo asociado a la distancia entre el mayor y menor elemento. En este caso, al evaluar una representación vectorial en enteros de los vértices, dicha distancia será n. De esta forma, retomando la fórmula de complejidad, esto resulta en un orden $\mathbf{O(}n * log(n)\mathbf{)}$ en ambas ramas, ya que las funciones de comparación utilizadas son de costo constante y el conjunto de vértices evaluado es el mismo.

Por otro lado, en la línea 4 se utiliza un coloreo secuencial de los vértices del problema, tomando el orden previamente aplicado en base al parámetro gr. Este, según lo evaluado en la sección 2.2, nos proporciona un costo $\mathbf{O(}n^2 + m_G\mathbf{)}$.

Asimismo, los llamados a *impactoDeColoreo* en las líneas 6 y 8 aportan cada una una complejidad temporal de $\mathbf{O(}m_H\mathbf{)}$.

Para finalizar, el llamado de la línea 7 refiere a una búsqueda local golosa. Ésta realiza un intento de coloreo distinto para cada arista en H. En el peor caso, si la arista no tiene ambos vértices del mismo color en G, recalcula el impacto en base a este, llamando a la función *impactoDeColoreo*. Las operaciones que rodean tanto a este llamado como a su ciclo son de orden constante,

como comparaciones y asignaciones. Entonces, la cota superior de complejidad temporal ofrecida será considerando un llamado a impactoDeColoreo por cada arista de H, resultando en $\mathbf{O(}m_H * (m_H + m_G)\mathbf{)}$.

En conjunto, la complejidad temporal asintótica de esta heurística tomando la cota superior será de $\mathbf{O(}n*log(n)+m_H+n^2+m_G\mathbf{)}$ que se puede acotar "burdamente" por una complejidad cuadrática dependiendo del tamaño del conjunto de vértices. Sin embargo, es importante resaltar que esta complejidad de peor caso se espera que no ocurra por lo general en la práctica resultando en una cota muy burda para el contexto de uso del problema. Dado que estos ordenes de magnitud van a ser reflejados en en grafos con densidades altas de aristas y llevándolo al extremo de grafos completos no tiene sentido su aplicación en este problema. Mientras que al utilizar el método de búsqueda local, dicha cota resulta $\mathbf{O(}n*log(n)+m_H*(1+m_H+m_G)+n^2+m_G\mathbf{)}$, acotado "burdamente" por $O(n^4)$.

2.4. Heurística golosa según vértices en H

En este segundo algoritmo se optó por una heurística que colorea G, basándose en H. Se obtienen primero las adyacencias de cada uno de los vértices pertenecientes al grafo H, para luego de ordenarlos según su grado,es decir, según con cuántos otros vértices comparte una arista, esto se realiza en la línea 4.

A continuación, para cada vértice con aristas del grafo H, se lo añade en el vector de color inicializado en la línea 5 y posteriormente se lo colorea, de forma tal que éste sea válido. Luego de esto, para cada vértice que comparta una arista con éste, y que no se encuentre en el grafo G ni en el vector, también es añadido y coloreado según corresponda. Luego, si el vértice actual es el último en el vector de color, lo cual significa que este no tiene adyacentes que no estan en G, se elimina del vector de coloreo y se asigna $\perp$ como su color en el vector **ordenCol**, que es el que se utiliza como coloreo final.

Finalmente, en la línea 18, llamamos a la función **coloreoSecuencial**, que se encarga de dar color a los nodos que no fueron coloreados por el paso anterior.

Algorithm 2 Algoritmo de Segunda Heurística golosa.

```
1:  aH = v/v ∈ V ∧ d(v) ≠ 0
2:  N_Y(v) = w/w ∈ V ∧ (v, w) ∈ E_Y
3:  function segundaHeuristica
4:      ordenar(aH,gradoH)
5:      vColor ←<>
6:      cr←0
7:      for v ∈ aH do
8:          agregar(vColor,v)
9:          ordenCol[v-1] ← cr
10:         colorSH(vColor, cr,ordenCol,H,G)
11:         for k ∈ N_H(v) do
12:             if k ∉ N_G(v), vColor and cr < n then
13:                 agregar(vColor,k)
14:                 colorSH(vColor, cr,ordenCol,H,G)
15:         if ultimo(vColor) = v then
16:             eliminarUltimo(vColor)
17:             ordenCol[v-1] ← ⊥
18:     coloreoSecuencial(V_H, ordenCol,G)
19:     return impactoDeColoreo(ordenCol,H)
```

Esta heurística con respecto a las anteriores, realiza un enfoque más complejo sobre ambos grafos en su totalidad. Lo cual esperamos que resulte en un incremento significativo en la calidad de las soluciones presentadas por la misma. Evidenciando el impacto que puede tener en una heurística las consideraciones sobre las propiedades del problema. En la Sección 3 vamos a profundizar más en este análisis, comparando esta heurística con el resto de las heurísticas golosas presentadas.

2.4.1. Calidad de la solución

En esta ocasión, se evaluará la calidad de la heurística con una única configuración, la ejecución HGV.

Instancia	Impacto optimo	Impacto HGV
n6	1	1
n8	6	6
n10	3	3
n12	16	14
n14	12	12
n16	20	16
n18	27	21
n20	25	20
n22	26	22
n24	33	26
n26	38	31
n28	48	31
n30	47	31

Como se puede apreciar en el cuadro, la diferencia con respecto a los impactos óptimos es considerablemente menor con respecto a la heurística anterior. Esto nos da, en promedio, un gap absoluto de **15,323 %**

2.4.2. Complejidad

La complejidad del algoritmo puede determinarse teniendo en cuenta que lo primero que se realiza es la inicialización del vector de vértices que poseen grado > 1 en el grafo H, debido a que corroboramos no agregar dos veces cada vértice, este paso resulta ser $\mathbf{O}(n^2)$ en el peor caso. Ya que no agregamos los vértices más de dos veces en el vector de vértices a colorear, la complejidad que obtenemos es de orden $\mathbf{O}(n)$.

Los dos ciclos **for** que se encuentran en el algoritmo se ejecutan $\mathbf{O}(m_H)$ veces. Dentro de estos ciclos, se verifica que el vértice no se encuentre en el vector de vértices a colorear, lo cual cuesta a lo sumo $\mathbf{O}(n^2)$.

En cuanto al coloreo de vértices, como el ciclo se ejecuta $\mathbf{O}(n^2)$ veces, y en cada iteración el costo del peor caso es de $\mathbf{O}(m_H + m_G)$, el costo total de este algoritmo sería de $\mathbf{O}(n^2 * (m_H + m_G))$, pudiendo tomar como cota burda $\mathbf{O}(n^4)$. El segundo ciclo **for**, que se encarga de chequear las adyacencias de los vértices tiene complejidad $\mathbf{O}(m_H * (m_H + n))$, luego, nuevamente el coloreo, $\mathbf{O}(n^4)$.

Finalmente, el llamado a **coloreo secuencial** en la línea 18 tiene un costo de $\mathbf{O}(n^2)$, luego, el cálculo del impacto tiene una complejidad de $\mathbf{O}(m_H)$. Consecuentemente, el costo total del algoritmo es de $\mathbf{O}(n^2 + n^4 + n^2 + m_H)$, lo cual podemos acotar como $\mathbf{O}(n^4)$. Sin embargo, como mencionamos anteriormente, en la práctica este tipo de cotas pueden resultar demasiado “burdas”.

2.5. Tabú Search (TS)

La búsqueda Tabú es un enfoque metaheurístico cuyos cimientos están basados en el uso de una memoria adaptativa y estrategias particulares de resolución de problemas. Así, su filosofía se centra en la explotación de estrategias puntuales basadas en procedimientos de aprendizaje y memorización.
Este enfoque presenta una estructura delimitada. En primer lugar, se obtiene una solución base para la instancia a evaluar. Luego, considerando un criterio de parada estático, se analizan mejoras mediante la generación de soluciones llamadas "vecinas"(soluciones similares a mi solución base), obviando aquellos ya explorados y prohibidos por la memoria del algoritmo, soluciones ”tabú”. Dicha generación es el resultado de aplicar los operadores presentados en la sección 2.5.1, para más tarde evaluar el mejor del conjunto como posible candidato a la solución más óptima. En simultáneo, se almacena este candidato en la *memoria tabú*, para evitar repetir exploraciones al buscar nuevas vecindades, considerándolo como un camino ya recorrido. Además, se introduce una función de aspiración: Esta cumple la función de exceptuar soluciones marcadas como ”tabú”, bajo algún criterio basado en el supuesto de que repasar a aquellas que la cumplan puede llevar a una solución mejor que la conocida.
En primer lugar, se ofrecen dos opciones para la solución por la cual comenzar: Por un lado, la solución dada por la segunda heurística (la más óptima) y, por el otro, la dada por la primer heurística con búsqueda local y con un orden de grado en G (la menos óptima de las dos heurísticas). Esto se determina llamando a la función con el parámetro `orden` vacío o no. Luego, se realiza el ciclo de mejoras una cantidad finita de veces: El criterio de parada elegido, representado en la línea 5, evaluará el número de iteraciones e iteraciones sin mejora realizadas y, si alguna supera respectivamente a los parámetros `l1` y `l2`, finalizará la ejecución del ciclo. En caso contrario el ciclo continuará, en primer lugar, generando la subvecindad de la solución actual.

Una vez obtenida la vecindad, se busca la solución con impacto más óptimo entre ellas con un llamado en la línea 7, que calculará linealmente cada impacto y seleccionará el mejor de ellos. Además, lo guardará de manera modular (es decir, el siguiente *slot* a asignar luego de utilizar la última posición será la primera) a la memoria de tamaño fijo M_T. Esto nos indica que poseemos una memoria acotada en donde registrar soluciones previamente exploradas, con lo cual una variable relevante a la optimalidad del algoritmo es su tamaño, t. En la implementación presentada, específicamente la línea 8, existen dos variantes de almacenamiento de soluciones exploradas: Por un lado, una *memoria por últimas soluciones*, almacenando el mejor vecino en su representación de vértice. En segundo lugar, una *memoria por estructura*, donde se representarán las soluciones memorizadas como el conjunto de

operadores swap y change que le fueron aplicadas a la solución original para llegar a la misma.
Finalmente, para cada ciclado se guarda el mayor de los impactos obtenidos, así como el orden de coloreo que le corresponda.

Algorithm 3 Algoritmo de Tabu Search.

```
1: M_T : |M_T| = t ∧ i ∈ 0...t − 1 → M_T[i] = ⊥
2: function tabuSearch(orden, i1, i2, pVecinos)
3:     if tam(orden) = 0 then imax ← segundaHeurística() else imax ← primerHeuristica(orden,true,true)
4:     actual ← ordenCol
5:     while continuarMejoras(i1,i2) do
6:         vecinos ← subVecindad(actual,H,G, M_T, pVecinos)
7:         actual← mejorVecino(vecinos)
8:         if memoriaSol then agregarModular(M_T, actual) else  agregarModular(M_T,swapYChange(actual))
9:         iactual ← impactoDeColoreo(actual)
10:        if iactual ≥ imax then
11:            imax ← iactual
12:            ordenCol ← actual
13:    return imax
```

2.5.1. Exploración de soluciones vecinas

Para la generación de vecindades en la metaheurística se introdujeron dos variaciones.
En primer lugar, se presenta la quizás más frecuente al hablar de Tabú Search, la generación de soluciones vecinas por *swap* y *change*. En segundo lugar optamos por una exploración de soluciones más intuitiva en el contexto de nuestro problema, el intento de colorear dos vertices incidentes a una arista en H del mismo color para incrementar el impacto, a continuación ambas van a ser descriptas con más profundidad.

Vecindad por *swap* y *change* : Para la exploración de soluciones, definimos el conjunto de soluciones vecinas generadas con estos operadores de la siguiente manera:

$$N(\mathcal{S})_{swap} = \{\textbf{ordenCol}_{N(\mathcal{S})} | \textbf{ordenCol} \in \mathcal{S} \wedge \{\mu, \omega\} \in V \wedge \textbf{ordenCol}_{N(\mathcal{S})}[\mu] = \textbf{ordenCol}[\omega] \wedge \textbf{ordenCol}_{N(\mathcal{S})}[\omega] = \textbf{ordenCol}[\mu]\}$$

$$N(\mathcal{S})_{change} = \{\textbf{ordenCol} \in \mathcal{S} \wedge (\mu, \omega) \in E_H \wedge \textbf{ordenCol}[\mu] \neq \textbf{ordenCol}[\omega] \textbf{ordenCol}_{N(\mathcal{S})}[\mu] = \textbf{ordenCol}_{N(\mathcal{S})}[\omega]$$
$$\wedge max\{c_i \in \textbf{ordenCol}\} + 1 = max\{c_i \in \textbf{ordenCol}_{N(\mathcal{S})}\}\}$$

*Donde $\mathcal{S}$ representa la solución a partir de la cual se genera la vecindad, V representa al conjunto de vértices y para continuar con la notación **ordenCol** refleja el coloreo realizado en la solución $\mathcal{S}$. Reflejando que en change generamos solamente una solución a partir de esta, para luego aplicar swap sobre esta nueva solución vecina*

La forma en la que utilizamos estos operadores es la siguiente :

- Mientras la generación de vecindades por *swap* genere mejoras generamos solo vecindades utilizando sólo este operador
- En el caso de que la vecindad por *swap* no presente ninguna mejora, en la siguiente iteración antes de generar la vecindad, aplicamos el operador *change* sobre la solución de la que partimos. Y luego a partir de esta generamos una vecindad por *swap*.

Vecindad por coloreo de aristas en H : En este caso vamos a introducir un operador particular para este problema, para luego contrastar la eficiencia obtenida con respecto a los operadores tradicionales. Este operador en particular se basa en partir desde una solución inicial y a partir de esta generar todos las soluciones que utilizando un color nuevo o uno ya usado pueden tener una nueva arista incidente a dos vértices del mismo color. Este operador se define de la siguiente manera:

$$N(\mathcal{S})_{custom} = \{\textbf{ordenCol}_{N(\mathcal{S})} | \textbf{ordenCol} \in \mathcal{S} \wedge (\mu, \omega) \in E_H \wedge \textbf{ordenCol}[\mu] \neq \textbf{ordenCol}[\omega] \textbf{ordenCol}_{N(\mathcal{S})}[\mu] = \textbf{ordenCol}_{N(\mathcal{S})}[\omega]$$
$$\wedge max\{c_i \in \textbf{ordenCol}\} \leq max\{c_i \in \textbf{ordenCol}_{N(\mathcal{S})}\}\}$$

Se define de una manera similar al operador change mencionado anteriormente pero al contrario que este, genera todas las posibles soluciones que incrementan el impacto actual y además no necesariamente siempre agrega un color nuevo.

2.5.2. Memoria con últimas soluciones exploradas

Un *approach* bajo una memoria que almacene t-últimas soluciones exploradas, también conocido como memoria explícita, es quizás la estructura de memorización por defecto al idear los pasos a seguir del algoritmo. Este enfoque para la memoria de la metaheurística tabú representa el enfoque más naive e intuitivo: Almacenar, de una manera directa, la solución a marcar como explorada, en su representación final. En este caso, esto es representado como el mismo arreglo de coloreo para cada uno de los vértices del problema. Así, a la hora de seleccionar el siguiente candidato referente a una vecindad específica, basta con tener

en cuenta que la misma no haya sido agregada al vector de soluciones ”tabú”. Esto, si bien resulta simple a la hora de modelar e implementar el algoritmo, puede dejar paso a soluciones que, si bien no son exactamente iguales a las memorizadas, fueron cambiadas con acciones ya conocidas, que podríamos saber de antemano no mejorarán la solución actual. En el siguiente párrafo se profundizará sobre una alternativa a esta.
Si bien por el tipo de memoria, este problema mencionado no puede ser subsanado, esta puede complementarse con una a largo plazo, teniendo en cuenta la frecuencia de aparición de una solución prohibida, combinada con una función de aspiración que considere como excepciones aquellas consultadas con una frecuencia proporcional a las iteraciones límite utilizadas en la condición de corte.

2.5.3. Memoria con estructura de vértices

Como contraparte, otro de los enfoques más populares a la hora de representar las soluciones almacenadas en la memoria es el conjunto de operaciones que llevaron a construir la misma. Así, se evita realizar las mismas acciones para generar una solución, al momento de generar la subvecindad. Como fue adelantado en la subsección anterior, esta implementación requiere una complejización en el concepto de solución, para obtener de manera eficiente las acciones mencionadas. De caso contrario, el costo de recuperar las operaciones realizadas para llegar a la solución podría llegar a superar los beneficios que almacenar dichas operaciones trae.
En este lineamiento, se introdujo una estructura *solución*, en donde se guardarán por un lado, el coloreo de la solución misma y, por otro, las operaciones realizadas para llegar a la misma: Tanto swaps como changes. Así, opuesto a comparar la solución presentada como coloreo con cada elemento de la memoria, se compara tanto la aplicación de change como el índice de swap de los elementos ”prohibidos” con la solución actual, considerando ”movimientos prohibidos” en lugar de ”soluciones prohibidas”. Esto, efectivamente, nos brinda una mejor precisión a la hora de memorizar, ya que ofrece un criterio más restrictivo: Sin importar de qué solucion se parta ni a cual se llegue, se prohiben todas las soluciones cuyas acciones de generación ya hayan sido probadas. Teniendo en cuenta una memoria finita y circular, esta alternativa hace mejor uso de los recursos.

2.5.4. Calidad de la solución

Esta metaheurística en particular cuenta con generación de números aleatorios, lo cual no produce resultados iguales en serie. Para obtener un mejor reflejo de la realidad, se realizarán 50 iteraciones en cada instancia, tomándose el impacto promedio redondeado de ellas.
Para esta medición, se tomarán seis configuraciones posibles: TSS (con memoria por últimas soluciones exploradas utilizando la heurística 2 como base), TSE (con memoria por estructura de vértices utilizando la heurística 2 como base), TSSA (con memoria por últimas soluciones exploradas y vecindario por coloreo en H utilizando la heurística 2 como base), TSSB (con memoria por últimas soluciones exploradas utilizando la heurística 1 como base), TSEB (con memoria por estructura de vértices utilizando la heurística 1 como base), TSSAB (con memoria por últimas soluciones exploradas y vecindario por coloreo en H utilizando la heurística 1 como base).
Además, se fijarán los parámetros **t** = 1000, **pVecinos** = 20 %, **l1** = **l2** = 1000 para este análisis. Más adelante, en la sección 3 se estudiará el impacto y optimalidad de variar dichos parámetros.

Instancia	Impacto optimo	Impacto aproximado TSS	Impacto aproximado TSE	Impacto aproximado TSSA	Impacto aproximado TSSB	Impacto aproximado TSEB	Impacto aproximado TSSAB
n6	1	1	1	1	1	1	1
n8	6	6	6	6	5	5	5
n10	3	3	3	3	3	3	3
n12	16	14	14	16	9	9	16
n14	12	12	12	12	7	7	12
n16	20	16	16	20	9	8	20
n18	27	21	21	27	11	11	27
n20	25	20	20	25	9	9	25
n22	26	22	22	26	16	15	26
n24	33	26	26	33	11	11	33
n26	38	31	31	38	13	13	38
n28	48	31	31	46	20	18	48
n30	47	31	31	46	18	16	37
Gap promedio	**-**	**15,17 %**	**15,26 %**	**0,65 %**	**55,3 %**	**59,21 %**	**0,81 %**

Como se puede apreciar, la diferencia en configuraciones afecta a las métricas de manera abismal. La configuración que más se aleja del óptimo en promedio es **TSEB**, muy a la par con **TSSB**. Con entre de 16 y 15 % de diferencial se ubican **TSS** y **TSE**. Finalmente, con menos del 1 % de gap, encontramos a las configuraciones **TSSA** y **TSSAB**.

2.5.5. Complejidad

En primer lugar, se considera la generación de la solución base por la heurística de peor complejidad, es decir, $O(n^4)$. A esto se le suma la inicialización de la memoria, en donde se almacenarán soluciones ya exploradas, con valor indefinido en cada slot. Esto

nos agrega una complejidad de $O(t)$.
En segundo lugar, el ciclo principal se ejecutará una cantidad $max(l1, l2)$ de veces, con los siguientes llamados:

- Generación de vecindades. Si la generación es por swapping, se verifica si es requerido aplicar el operador $change$ y, si es así, la función de aplicación del mismo agrega un costo de $O(3n)$ dados sus tres ciclos de tamaño cantidad de vértices. Además, la generación propia por swapping constará de dos ciclos anidados de longitud n, en cuyo interior se realizarán dos verificaciones: En primer lugar, si la solución a agregar presenta un coloreo válido (es decir, recorrer las aristas de G) y, en segundo, si la solución a agregar no ha sido salvada en la memoria. En total, por cada iteración el costo será de $O(m_G + t)$, lo cual resultará en la función generadora de swapping en $O(n^2 * (m_G + t))$.
  Como contraparte, si se opta por la generación alternativa o por aristas de H, se comienza inicializando y completando un vector de tamaño n, resultando en 2*n operaciones. Luego, mediante un ciclo, se recorren las aristas de H. Por cada iteración, se verifica adyacencia en G recorriendo la lista de adyacencia de uno de los vértices y, además, se verifican los coloreos producidos junto a un ciclo extra de n iteraciones para colorear ambos extremos de un color no utilizado. En total, esta alternativa nos da un costo de $O(2 * n + m_H * (m_G + n))$. En ambos casos, la selección randomizada del porcentaje de subvecindad seleccionada, agrega un costo de $O(pVecinos * vecinos\ generados/100)$.

- Selección del mejor vecino. Para ello, se recorre el tamaño resultante de la vecindad, es decir, $pVecinos * vecinos\ generados/100$ y, para cada vecino, verifica si pertenece al vector de soluciones exploradas, de tamaño t y además si supera el impacto mayor que tenemos hasta ahora. Entonces, esto agrega un costo total de $O(pVecinos * vecinos\ generados/100 * t * 2 * m_H)$.

- Calculo del impacto de coloreo. A partir de lo mencionado en la sección 2.2, aporta una complejidad de $O(2 * m_H)$

Notar que, al incluir simulación de aleatoriedad al seleccionar el porcentaje de la vecindad, su complejidad se calcula de manera empírica, considerando la generación sin colisiones.
Así, la complejidad total resulta en $O(t + max(l1, l2) * (n + n^2 * (m_G + t) + pVecinos * vecinos\ generados/100(1 + t) * m_H + m_H))$ en el caso de vecindades por swap, y $O(t + max(l1, l2) * (n + m_H * (m_G + n) + pVecinos * vecinos\ generados/100(1 + t) * m_H + m_H))$ para vecindades por aristas de H. Por otra parte, todas las operaciones no mencionadas cuentan con una complejidad constante, como comparaciones, operaciones aritméticas y asignaciones.

3. Experimentación

A continuación se presentarán distintos experimentos computacionales, utilizados para evaluar el rendimiento, costo y funcionamiento de los enfoques presentados en la sección 2. Estos fueron realizados con una serie de ejecuciones en un equipo Intel(R) Core(TM) i5-3570 CPU @ 3.40GHzen con 4 núcleos, el lenguaje de programación *C++*.

3.1. Métodos

Las configuraciones y métodos utilizados durante la experimentación son los siguientes:

- **HS1**: Heuristica golosa constructiva secuencial por grado en G
- **HS1B**: Heuristica golosa constructiva secuencial por grado en G con busqueda local
- **HS2**: Heuristica golosa constructiva secuencial por grado en H
- **HS2B**: Heuristica golosa constructiva secuencial por grado en H con busqueda local
- **HGA**: Heuristica golosa coloreando G por las aristas de H
- **HGV**: Heuristica golosa coloreando G por los vertices de H
- **TSS**: Metaheuristica Tabú search con memoria de vertices utilizando la mejor heurística golosa
- **TSE**: Metaheuristica Tabú search con memoria de soluciones utilizando la mejor heurística golosa
- **TSSA**: Metaheuristica Tabú search con memoria de vertices con soluciones vecinas alternativas utilizando la mejor heurística golosa
- **TSSB**: Metaheuristica Tabú search con memoria de vertices utilizando la peor heurística golosa
- **TSEB**: Metaheuristica Tabú search con memoria de soluciones utilizando la peor heurística golosa
- **TSSAB**: Metaheuristica Tabú search con memoria de vertices con soluciones vecinas alternativas utilizando la peor heurística golosa

3.2. Instancias

Para los distintos experimentos se utilizaran las siguientes instancias:

- **peor-caso-HS1**: Instancias de peor caso para la heuristica golosa constructiva secuencial por grado en G, descriptas en la sección 2.3.

- **peor-caso-HS2**: Instancias de peor caso para la heuristica golosa constructiva secuencial por grado en H, descriptas en la sección 2.3.

- **mejor-caso-HS1**: Instancias de mejor caso para la heuristica golosa constructiva secuencial por grado en G, descriptas en la sección 2.3.

- **mejor-caso-HS2**: Instancias de mejor caso para la heuristica golosa constructiva secuencial por grado en H, descriptas en la sección 2.3.
- **instancias-costo**: Instancias generadadas de forma aleatoria para medir la perfomance de los algoritmos.
- **instancias-calidad**: Instancias provistas por la cátedra para medir la calidad de los algoritmos

3.3. Experimento 1: Complejidad de Heuristicas secuenciales

> ***Hipótesis 1.*** *En la práctica, la complejidad temporal, tanto en su configuración por orden de adyacencia en G como en H, será perteneciente al orden* n^4.

Uno de los aspectos centrales de una heurística secuencial es su simpleza, tanto en el aspecto intuitivo como en su implementación. Como contraparte, nos encontramos con un diferencial para con la solución óptima que pone en duda la utilidad de este algoritmo. Sin embargo, otro factor determinante a la hora de medir un algoritmo aproximado es la complejidad del mismo: En ciertos contextos, realizar un *tradeoff* de optimalidad por complejidad puede ser no solo funcional, sino hasta necesario al incursionar en instancias con un tamaño considerable.
En este experimento, analizaremos en profundidad el costo temporal de dicha heurística.
Para ello, se utilizarán las configuraciones **HS1** y **HS2**, junto a las familias de instancias **mejor-caso-HS1**, **mejor-caso-HS2**,**peor-caso-HS1** y **peor-caso-HS2**.
En primer lugar, se analizarán más detenidamente los índices de calidad obtenidos para ambas configuraciones, junto a sus familias de mejor y peor caso.

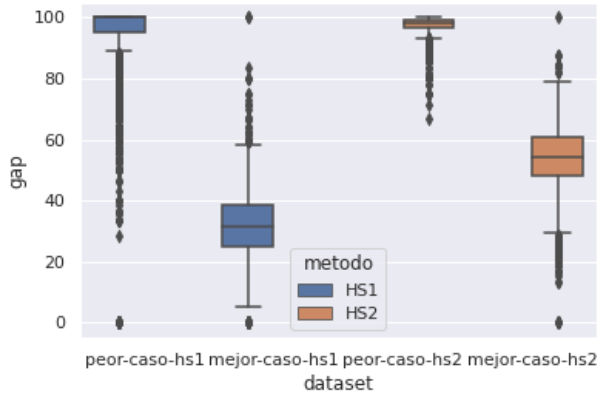

Figura 2: Calidad de soluciones ofrecidas en peor y mejor caso por cada heurística secuencial

Como se puede apreciar en la figura 2, ambas configuraciones acarrean un diferencial destacable entre sus peores y mejores casos en cuanto al gap. Por otro lado, la solución aproximada de más cercanía a la óptima fue, por alrededor del 20 % de diferencia, la configuración por orden de adyacencia en G.
Este diferencial nos lleva a la siguiente pregunta: ¿Puede ser que lo ganado en calidad se vea opacado por pérdidas en cuanto a performance?

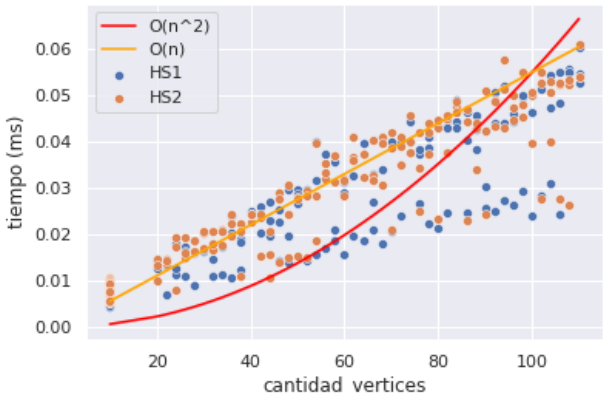

Figura 3: Comparación entre complejidades matemáticas y complejidades obtenidas para cada heurística secuencial

En la figura 3 podemos observar la complejidad obtenida para instancias inicializadas de manera aleatoria. Además de no notar una diferencia sustancial entre el tiempo insumido de cada configuración de la heurística, podemos resaltar algo de suma importancia: En los casos promedio, la cota temporal se aleja bastante de la hipótesis planteada.
Ahondamos, una vez más en la complejidad temporal, esta vez bajo el peor caso de calidad.

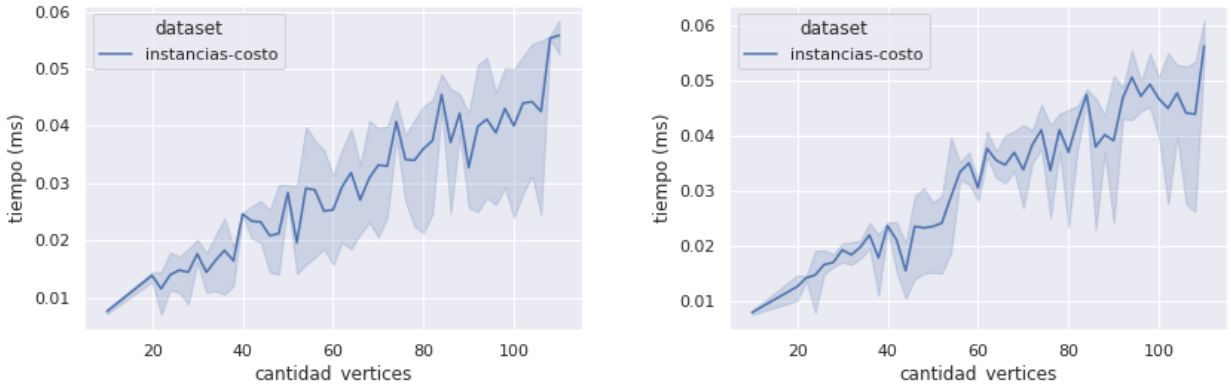

Figura 4: Medición de costo temporal para peor caso en HS1 (izquierda) y HS2 (derecha)

Notamos una leve suba del tiempo insumido, pero no es suficiente como para adecuarse a la hipótesis planteada originalmente. Esta diferencia entre la cota teórica se debe a lo mencionado previamente: Plantear el problema del PCMI en las instancias que representan a los peores casos teóricos (grafos completos) no es un sinsentido.

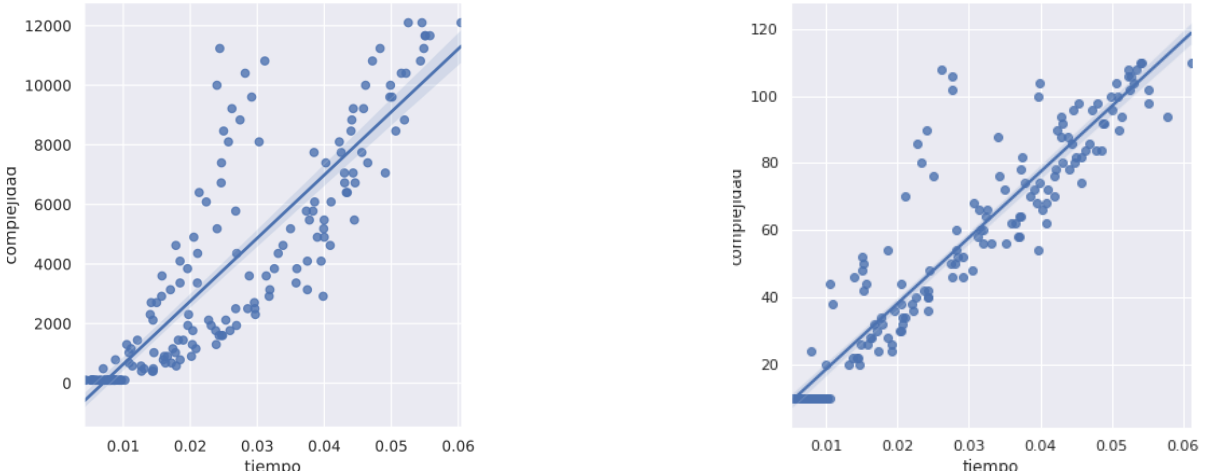

Figura 5: Correlación entre la complejidad matemática y complejidad obtenida para HS1 (izquierda) y HS2 (derecha) en peor caso

Finalmente, en la figura 5, se contrasta la correlación entre la función $f(n) = n$, complejidad matemática y la complejidad obtenida. Para la primer configuración, el coeficiente de correlación de Pearson resultante es de 0.83, mientras que para la segunda se obtuvo un más cercano 0.88. Esto nos da a entender que la complejidad temporal como cota superior en la práctica es más cercana al orden n^2 que a la planteada de manera teórica, n^4

3.4. Experimento 2: Complejidad de Heuristica golosa de coloreo según H

Hipótesis 2. *En la evaluación de los peores casos, la cota de complejidad temporal se volverá polinómica.*

En esta ocasión mediremos la heurística golosa de coloreo según vértices en H. En un previo análisis, se constató la amplia mejora que introdujo esta heurística con respecto a la anterior. Resta, así, una profundización en su complejidad.
Para ello, se utilizará una única configuración **HGV**, junto a las familias **peor-caso-HGV** e **instancias-costo**.

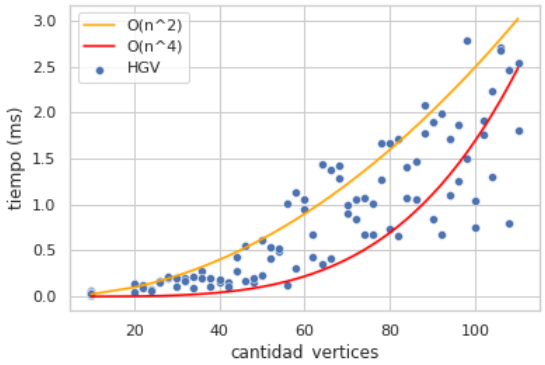

Figura 6: Contraste entre complejidades matemáticas y complejidad obtenida para cota superior

Como podemos observar en la figura 6, la complejidad obtenida para las instancias de peor caso verifica nuestra hipótesis, ya que el crecimiento de la complejidad puede ser acotado por n^4, e incluso, pudiendo tomarse como cota más fina n^2.

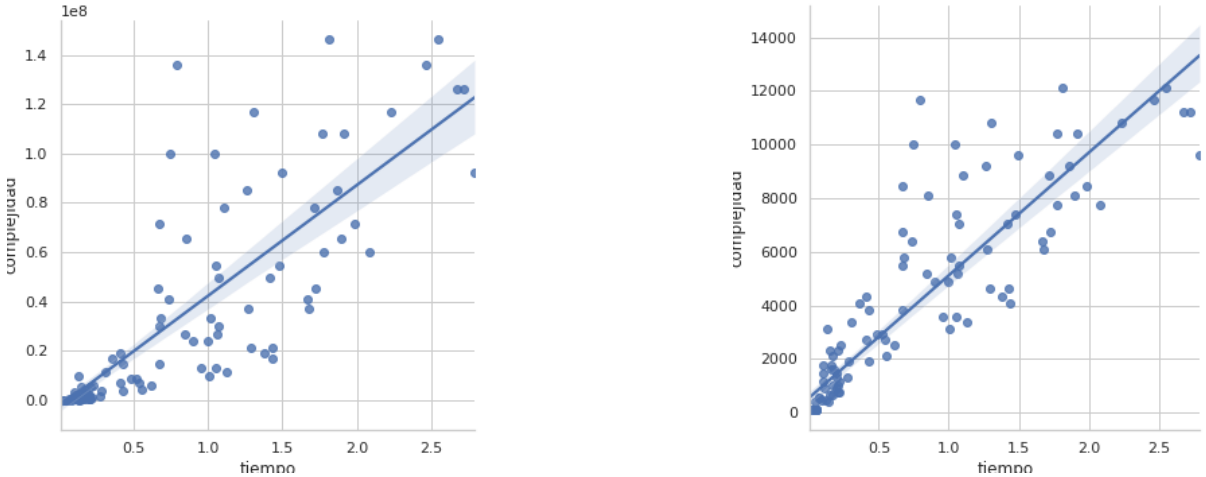

Figura 7: Correlación entre la complejidad matemática y complejidad obtenida para peor caso (izquierda) y caso promedio (derecha)

En la figura 7 contrastamos la correlación entre la complejidad obtenida y la complejidad matemática, con la función $f(n) = n$. En el peor caso obtenemos un coeficiente de correlación de Pearson de 0.84, y para el caso promedio, el coeficiente obtenido es de 0.89. Por esto podemos entender que la complejidad temporal, al igual que en el experimento anterior, la cota superior en la práctica resulta ser más cercana a n^2 que a la teórica n^4.

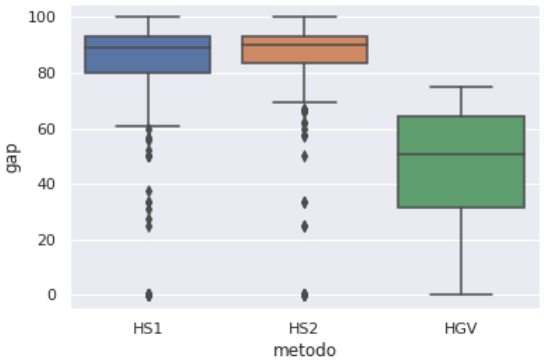

Figura 8: Comparación de calidad entre HS1, HS2 y HGV

Finalmente, en la figura 8 podemos ver la comparación de calidad entre **HGV**, **HS1** y **HS2**. Aquí podemos asegurar que la segunda heurística es mejor que la primera y su variante, presentando un gap cercano a 55, mientras que las dos primeras heurísticas tienen un gap cercano a 90.

3.5. Experimento 3: Impacto del método utilizado para generar vecinos en Tabú search

> ***Hipótesis 3.*** *Una generación de soluciones vecinas más adaptada al problema va a presentar un incremento significativo en la calidad de las soluciones del mismo.*

Como se evidenció en el estudio de calidad, la implementación alternativa a la hora de generar las soluciones vecinas nos otorgó la mejor solución en cuanto a diferencial en contraste con la solución óptima. En este experimento, acompañando, se compararán las complejidades de la heurística con la generación tradicional y la generación alternativa de vecinos, además de profundizar en su calidad, ambas con memoria por últimas soluciones exploradas, en casos aleatorios.
Para ello, se utilizarán las configuraciones **TSS** y **TSSA**, junto a las familias **instancias-costo**.

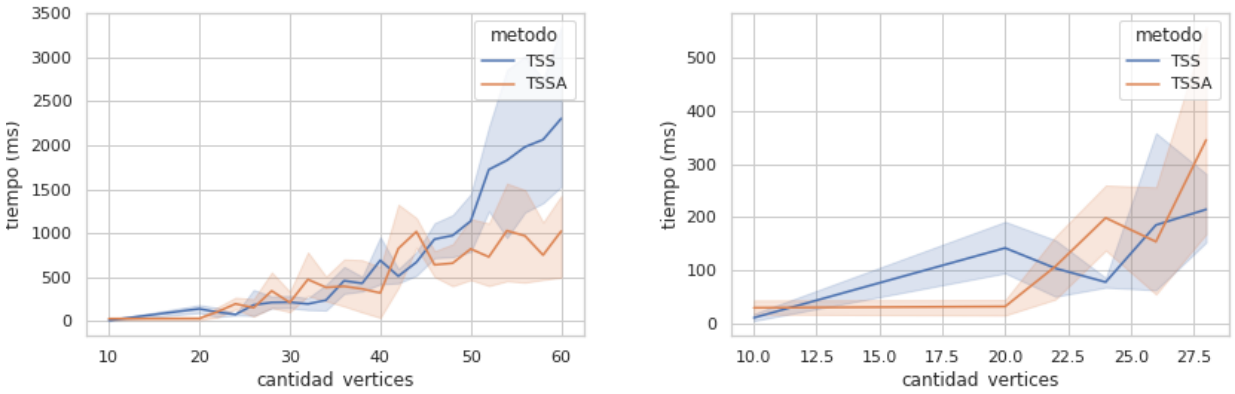

Figura 9: Comparación de costo temporal entre heurísticas variando la selección de vecindad. A la derecha, una ampliación

En la figura 9 podemos apreciar un diferencial mínimo en costo temporal entre cada configuración. Sin embargo, a partir de instancias con más de 50 nodos, la historia es otra: La generación de vecindades por operadores clásicos insume más tiempo que la generación alternativa. De esto surge de nuevo el interrogante acerca de la calidad que nos acercará esta alternativa a una vecindad generada por *swaps* y *changes*.

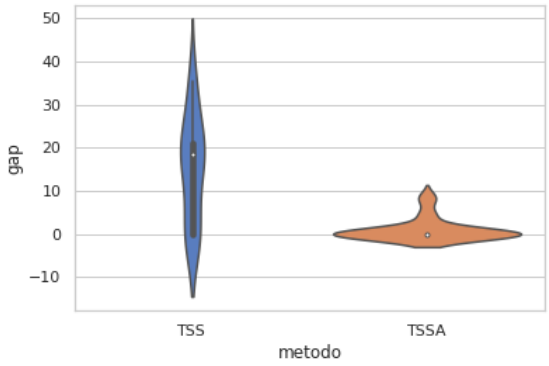

Figura 10: Comparación de calidad entre heurísticas variando la selección de vecindad.

Por contrario a lo esperado en las problematicas que abordamos, donde una ganancia en complejidad puede suponer una pérdida en calidad, en la figura 10 se observa cómo el foco alternativo presentado a la generación clásica presenta un diferencial gap más cercano a 0, en promedio, y con una varianza sorprendentemente menor.

3.6. Experimento 4: Impacto de la solución inicial en Tabú search

> ***Hipótesis 4.*** *El impacto de la solución a partir de la cual se realiza la búsqueda Tabú va a afectar tanto en tiempos de ejecución como en calidad de las soluciones. Considerando esto, partiendo de una solución menos óptima el costo de exploración se va a ver incrementado y además la calidad obtenida.*

Tal como la selección de vecinos, el punto de partida de la metaheurística representa un punto central en cuanto a lo que la métrica analizada previamente respecta: Los métodos que parten desde la heurística con mejor gap, presentan aproximadamente una tercera parte de diferencial con respecto al de los métodos que parten desde la peor. En esta ocasión se contrastarán ambos

métodos, profundizando en la calidad de los mismos junto a un análisis de la complejidad que estos atraen.
Para ello, se utilizarán las configuraciones **TSS**, **TSSA**, **TSSB** y **TSSAB**, junto a las familias **instancias-costo** e **instancias-calidad**.

Figura 11: Comparación en costo temporal entre tabú search con distintas soluciones de partida

Como podemos observar en la figura 11, el método **TSSB** es el que tiene mayor crecimiento, los métodos **TSS**, **TSSA** y **TSSAB** muestran un crecimiento bastante similar. En el acercamiento podemos observar como en un entorno acotado los cuatro métodos se comportan de forma similar. Podemos afirmar que **TSSB** al utilizar la peor heurística golosa y no contar con la memoria de vértices con soluciones vecinas alternativas como si ocurre con **TSSAB** termina siendo el peor método utilizado en este experimento en cuanto a velocidad.

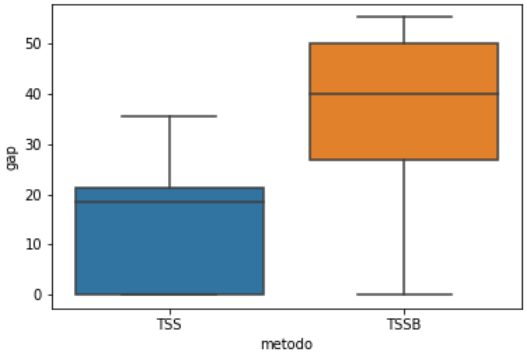

Figura 12: Comparación en calidad entre tabú search con distintas soluciones de partida

Al realizar un análisis mas fino entre **TSS** y **TSSB** podemos ver que el gap obtenido con estos métodos se diferencia en al menos 20 unidades, teniendo la ventaja **TSS**, y como este también le gana en tiempo, podemos afirmar que es mejor.

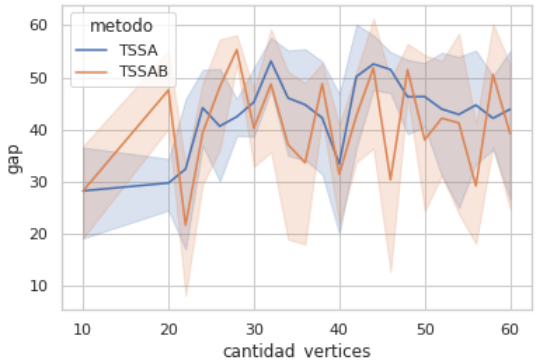

Figura 13: Comparación en calidad entre tabú search con distintas soluciones de partida

Por otro lado, si comparamos **TSSA** con **TSSAB**, vemos que los gaps obtenidos fluctúan entre 20 y 60, pero, al ver como la complejidad temporal de **TSSA** es apenas levemente mayor, no podríamos elegir **TSSAB** como un método totalmente mejor. Por lo tanto, podemos decir que en este caso no nos vemos afectados por la solución inicial.

3.7. Experimento 5: Impacto del tipo de memoria utilizada en Tabú search

> ***Hipótesis 5.*** *La memorizacion por soluciones va a tener un costo significativamente mayor a la que solo memoriza la estructura de la solucion. Mientras que ambos tipos de memorizacion van a presentar la misma calidad en sus soluciones*

Otro de los puntos centrales a debatir en la metaheurística es la memoria elegida, temática abordada en la sección 2.5. En este experimento profundizaremos en las ventajas y desventajas de cada una de las memorias elegidas.
Para ello, utilizaremos la configuración **TSE** y **TSS** fijaremos los párametros de tamaño de memoria y cantidad de iteraciones a realizar en 1000; el porcentaje de subvecindad en 30 y el límite de iteraciones sin mejora en 100, utilizando las instancias de **instancias-costo** e **instancias-calidad**.
Comenzamos con un análisis del costo temporal para cada memoria elegida.

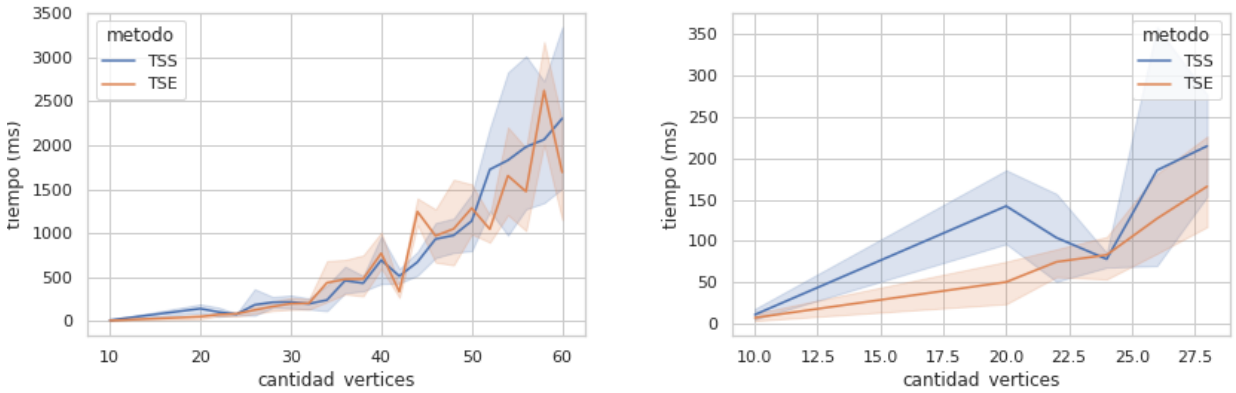

Figura 14: Comparación en costo temporal entre tabú search con distintas estructuras de memoria

Intuitivamente, al variar solo en el salvado de la solución, la complejidad temporal no se ve afectada: En ambos casos se genera el mismo arreglo a utilizar como memoria, y lo que se almacena son una o dos componentes de la estructura solución: O bien el coloreo, o bien los operadores que nos llevaron a ese coloreo partiendo de la solución anterior. En ambos casos, el costo termina siendo un conjunto de asignaciones finitas.

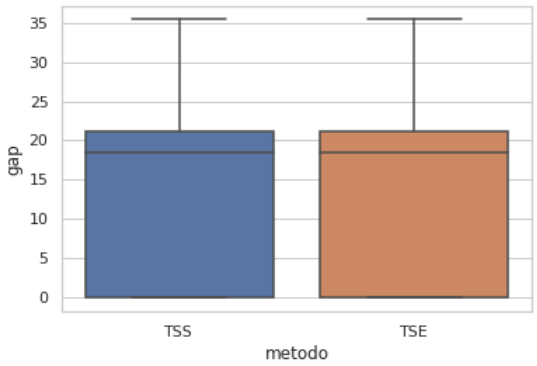

Figura 15: Comparación en calidad entre tabú search con distintas estructuras de memoria

En cuanto a la medición de calidad, podemos ver un claro empate entre ambos métodos a evaluar: Para una misma generación de vecindades, la diferencia que resulta de guardar operadores o soluciones es mínima para las instancias en donde el problema es medido.

3.8. Experimento 6: Configuración de los parámetros de Tabú search

> ***Hipótesis 6.*** *Variando los distintos parámetros de Tabú Search podremos observar variaciones en los resultados y en los costos temporales.*

Uno de los aspectos en la contracara de las mejoras que acerca Tabú Search son los numerosos parámetros con los que cuenta. La capacidad de mejora de este algoritmo está inherentemente ligada a hasta dónde se le deje optimizar con los límites establecidos. Es por esto que en este experimento nos dedicaremos al análisis de las configuraciones de la heurística, y qué impacto tienen en la

solucion final.
Para ello utilizaremos las configuraciones **TSS** y **TSSA** con las instancias **instancias-calidad** e **instancias-costo** variando el tamaño de memoria, el porcentaje de subvecindad, el límite de iteraciones sin mejora y la cantidad de iteraciones a realizar.
No utilizamos **TSE**, ya que como pudimos observar en el experimento anterior, la calidad de soluciones obtenidas es igual a **TSS** e incluso el costo temporal llega a ser menor en algunos casos. En primer lugar, analizaremos el tamaño de memoria que nos acerca a una relación tiempo-calidad suficiente para la índole del problema.

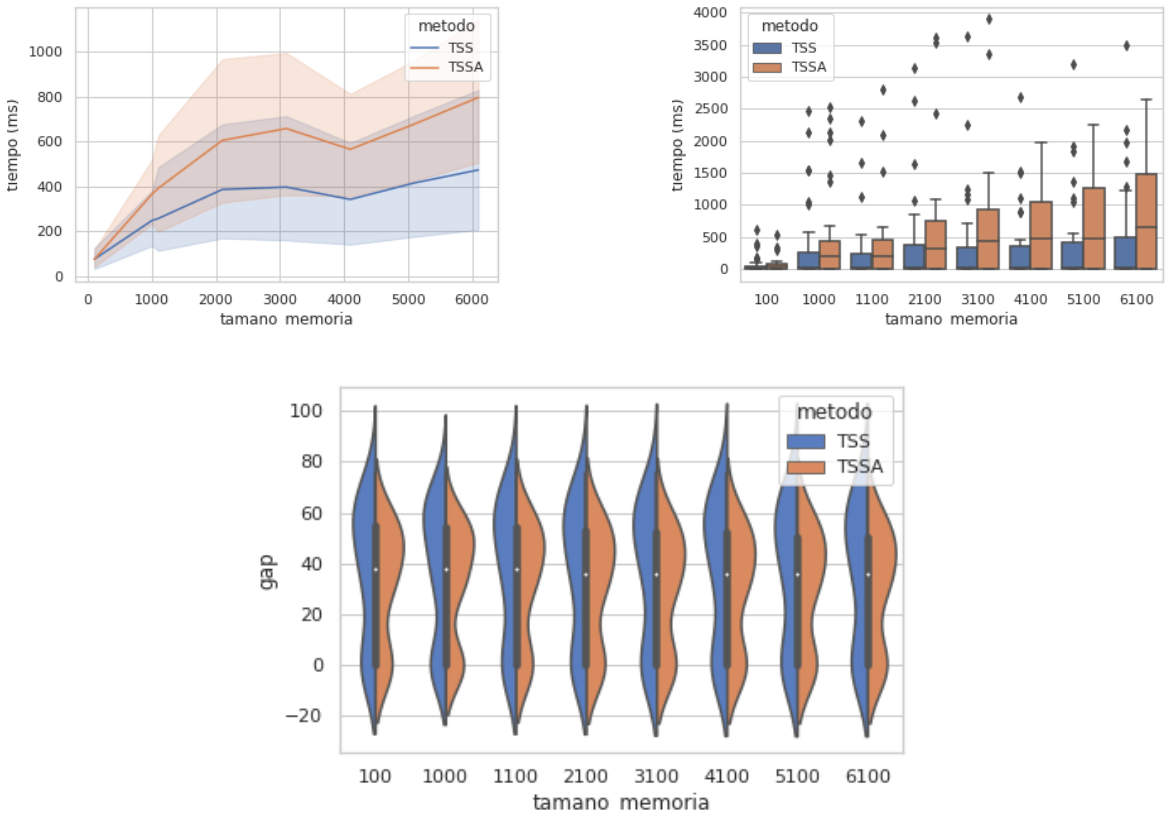

Figura 16: Variación de tamaño de memoria

Como podemos observar en la figura 16, la variación en cuanto al tamaño de la memoria viene acompañado de un aumento en el costo temporal sublineal, lo cual no representa un crecimiento significativo.
En contraparte, en cuanto al análisis de calidad, tanto las iteraciones como las iteraciones sin mejora fueron fijadas en 1000 y en 500 respectivamente, permitiéndonos así notar que la variación de estos parámetros no altera los resultados, ya que los gráficos obtenidos no presentan variación alguna en cuanto al gap obtenido.

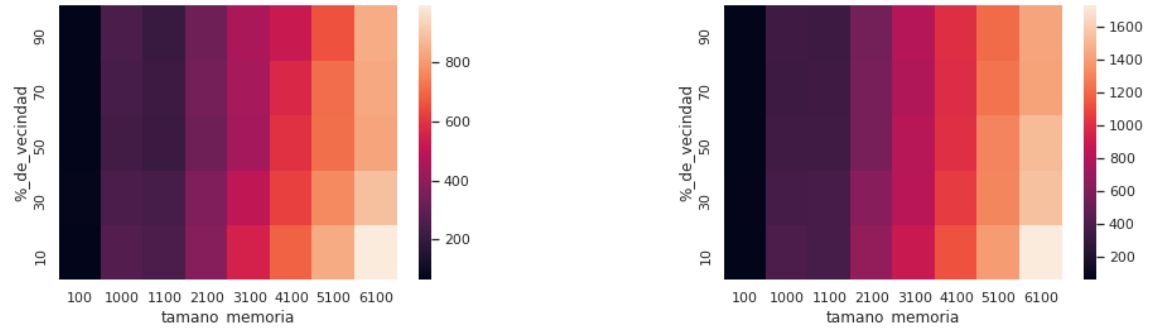

Figura 17: Variación de memoria y vecindad en **TSS** (izquierda) y **TSSA** (derecha)

En la figura 17 podemos observar como lo que realmente nos marca una diferencia notable de tiempo de ejecución es la variación del tamaño de la memoria. La variación de porcentaje de vecindad no produce cambios en tiempo de ejecución tan marcados como si lo hace el tamaño de memoria. Analizaremos cada método por separado a continuación.

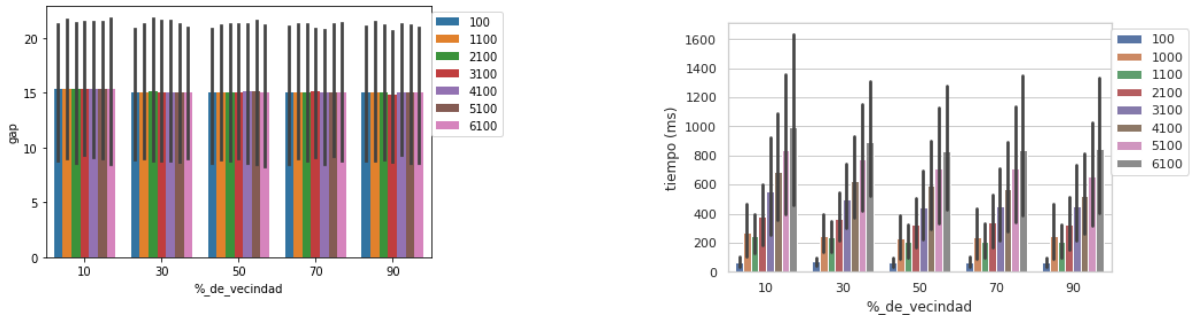

Figura 18: Variación de vecindad en **TSS**, calidad (izquierda) y costo (derecha)

Podemos notar en la figura 18 como la calidad obtenida no se ve alterada ni por la variación de la vecindad ni por la variación del tamaño de memoria, representado con las distintas barras que componen el gráfico. En el gráfico de la derecha, podemos constatar lo mencionado anteriormente, la variación del porcentaje de vecindad no marca una diferencia notable en el tiempo de ejecución como si lo hace la variación del tamaño de memoria.

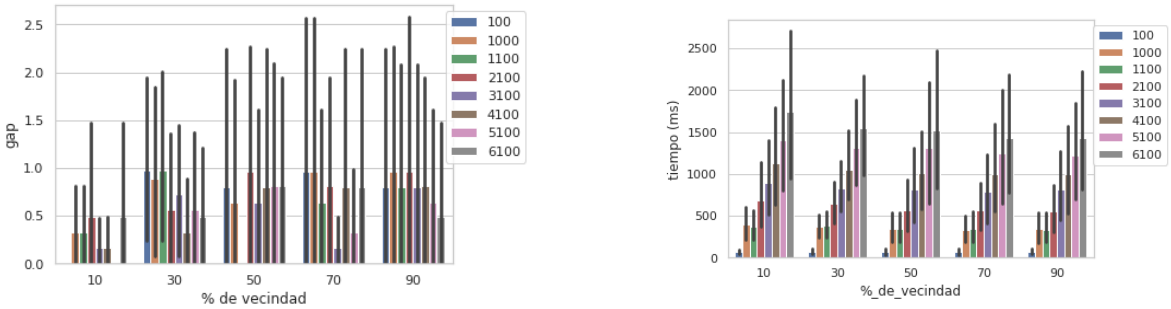

Figura 19: Variación de vecindad en **TSSA**, calidad (izquierda) y costo (derecha)

Como podemos ver en la figura 19, en el método **TSSA** el resultado obtenido no se ve alterado por porcentaje de vecindad, pero sí por la variación en el tamaño de memoria, podemos notar que a medida que la memoria crece, el gap disminuye. Por otro lado, podemos ver como el costo temporal no se ve afectado al cambiar este parámetro, variando más que nada por el cambio del tamaño de memoria, representado por los distintos colores en el gráfico de barras.

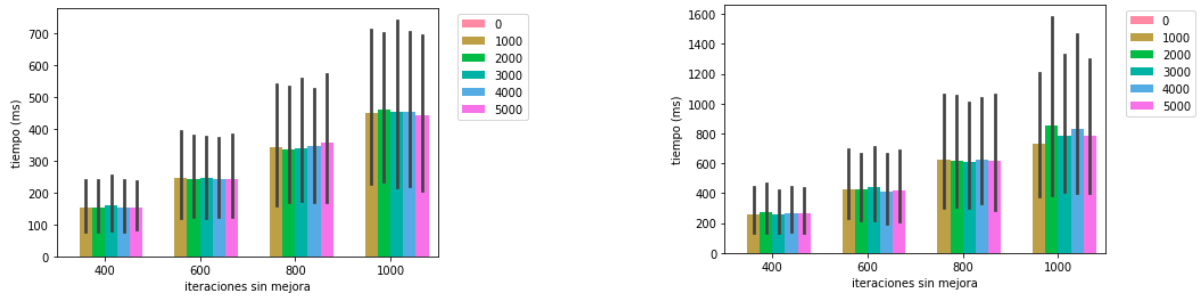

Figura 20: Variación de iteraciones sin mejora para **TSS**(izquierda) y **TSSA** (derecha) en cuanto a costo

La figura 20 nos permite afirmar que, si bien los distintos tamaños de memoria afectan el comportamiento, esto es mínimo, y lo que efectivamente altera el costo temporal de ambos métodos son las iteraciones sin mejora.
Notar que en los gráficos se puede evidenciar una varianza superior a las demás mediciones presentadas en este trabajo. Esto se puede deber a posibles ruidos generados en el equipo que generó las mediciones.

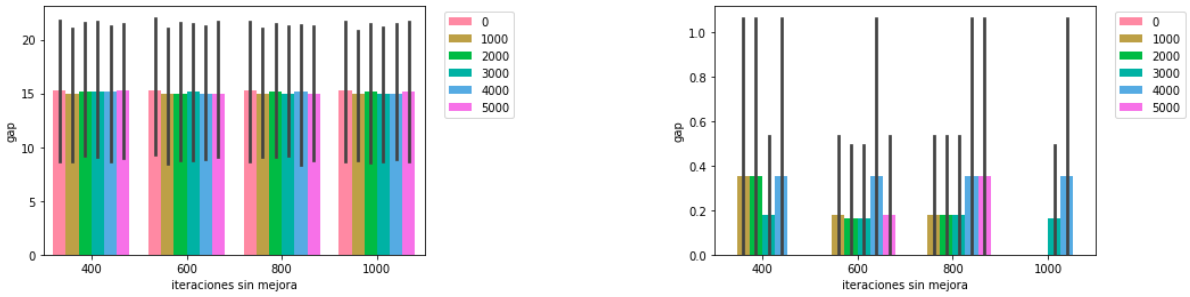

Figura 21: Variación de iteraciones sin mejora para **TSS** (izquierda) y **TSSA** (derecha) en cuanto a calidad

Podemos apreciar en la figura 21, la diferencia que implica variar las iteraciones sin mejora junto al tamaño de la memoria en el método **TSS** no representa un cambio significativo en la medición de calidad, sino que las configuraciones siguen oscilando alrededor de un mismo porcentaje, con una varianza de apertura similar. Esto no es así analizando al método **TSSA**: en este podemos observar como para cada set de iteraciones sin mejoras es notable como hay algunos tamaños de memoria que obtuvieron mejores resultados que otros, como es el caso de 1000 iteraciones sin mejoras, donde una memoria de tamaño 3000 tiene mejor resultado que una de tamaño 4000.
Vale mencionar que en la derecha de la última figura se eliminaron aquellas configuraciones que contaban con un diferencial mayor a 5, para poder realizar un análisis más minucioso de las más óptimas.

3.9. Experimento 7 : Contraste entre los distintos métodos para resolver el problema

> ***Hipótesis 7.*** *La heurística golosa en sus dos configuraciones, junto a la segunda heurística, presentarán un costo de complejidad significativamente menor que las variantes de la metaheurística. Como contraparte, las primeras traerán soluciones menos óptimas que las configuraciones de Tabú search.*

Una vez habiendo analizado de manera aislada cada heurística, y habiendo obtenido la mejor configuración para Tabú Search, nos dedicaremos a enfrentar a las soluciones. Para ello, utilizaremos la familia de **instancias-calidad**, haciendo competir los métodos de **TSS**, **TSE**, **TSSA**, **HS1**, **HS2** y **HGV**. Además utilizaremos la familia de **instancias-costo** con los métodos **TSS**, **TSE** y **TSSA** utilizando también la configuración obtenida en el experimento anterior.

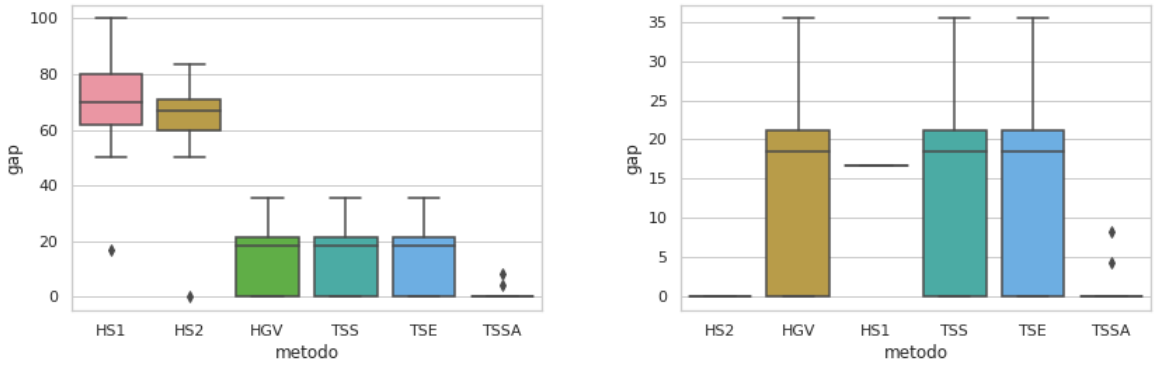

Figura 22: Comparación de la calidad obtenida en distintas soluciones. A la derecha, una ampliación

Como se puede apreciar en la figura 22, hay una delimitada diferencia entre la optimalidad de las soluciones obtenidas, acompañando un poco la idea general de que la simpleza a la hora de implementar equivale a una solución no muy exacta: Las heurísticas secuenciales cuentan con un gap elevado, mientras que la segunda heurística y tabú search con sus vecindades tradicionales disminuyen este hasta por debajo del 25 %. Finalmente, la configuración de tabú search que se puede considerar ”más compleja” presenta un casi invisible gap de menos del 10 % de manera absoluta, y en promedio menor al 1 %.

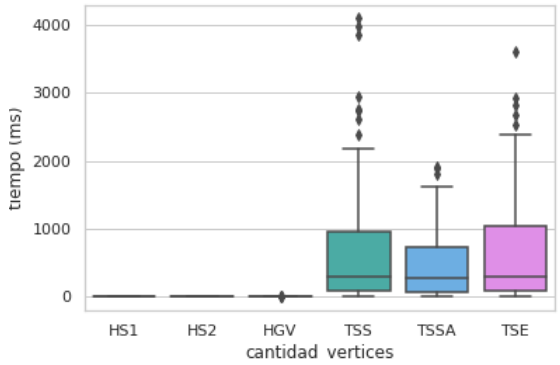

Figura 23: Comparación de costo temporal obtenido en distintas soluciones

Acompañando a lo mencionado anteriormente, una complejización en la implementación viene de la mano con un aumento en el tiempo insumido: En la figura 23 vemos como la complejidad de las primer heurística en ambas configuraciones y la segunda es imperceptible: Por debajo del segundo. En contracara, podemos observar que, si bien la mediana de las tres configuraciones de tabú search se mantiene por debajo de los 1000 ms, la diferencia es notable. Sin embargo, el comportamiento de la metaheurística con generación de vecindades alternativas presenta un costo temporal visiblemente más bajo que el de sus compañeros, resultando en una relación costo-calidad considerablemente mejor.

3.10. Experimento 8: Contraste entre búsqueda local golosa y tabú search

> ***Hipótesis 8.*** *La complejidad que acompaña a la búsqueda en el espacio de soluciones cuando utilizamos Tabú search genera un incremento en la calidad de los resultados sustancialmente mayor que las búsquedas de soluciones menos complejas. Sin embargo, el costo que representa el Tabú search va a ser significativamente mayor que el presentado por la Búsqueda local golosa.*

Finalmente vamos a abordar un análisis sobre el impacto de la búsqueda que realiza Tabú search en contraste con búsquedas locales tradicionales. En nuestro caso implementamos una búsqueda local golosa que se aplica sobre las heurísticas de construcción golosa basadas en el coloreo secuencial. A continuación vamos a contrastar el costo y la calidad de cuatro métodos, de los cuales dos realizan una búsqueda local golosa sobre las soluciones de las heurísticas secuenciales implementadas. Mientras que las otras dos realizan una Búsqueda tabú sobre las mismas soluciones.

Para llevar a cabo esto, vamos a ejecutar los métodos de **TSSB** y **TSSAB** contra **H1SB** y **H2SB** sobre la familias de **instancias-calidad** e **instancias-costo**.

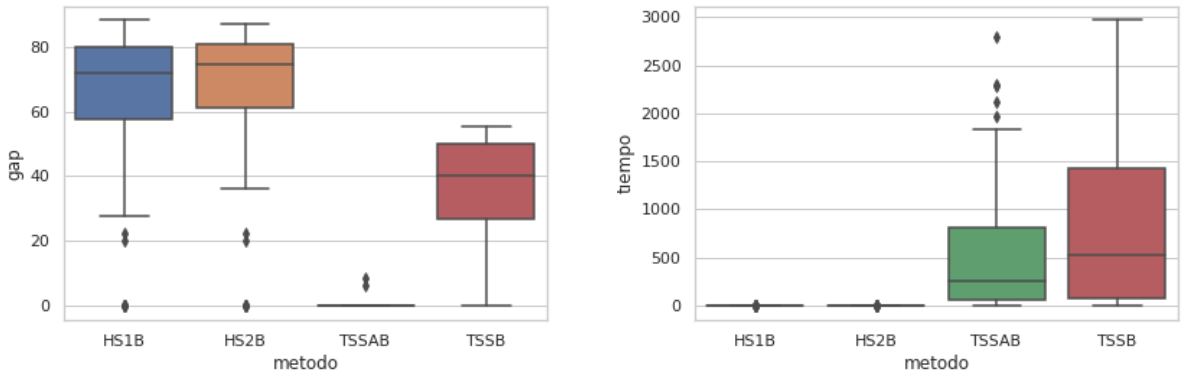

Figura 24: Medición de calidad (izquierda) y costo temporal (derecha) para los métodos golosos con búsqueda local, y tabú search con generación de vecindades alternativas y peor solucion base

Cómo podemos apreciar en el gráfico 24 , la calidad obtenida por el método **TSSB** con respecto a **HS1** y **HS2** es apróximadamente un 35 % mayor en cuanto a precisión, considerando la mediana de los gaps obtenidos. Mientras que para **TSSAB** tenemos una mejora de entre 75 % y 80 %, lo cual demuestra el poder que tiene la forma de exploración de soluciones de la Búsqueda por tabú en contraste con la búsqueda local propuesta. Sin embargo, el no siempre estamos dispuestos a *pagar* el costo necesario para lograr las soluciones óptimas.
Por otro lado, en la derecha se evidencia claramente el impacto de la búsqueda tan exhaustiva realizada por la metaheurística. Evidentemente el costo computacional de la búsqueda local es significativamente menor, y en muchos contextos de uso esto puede marcar la diferencia en cuanto a la selección de heurísticas a utilizar para resolver el problema presentado.

4. Conclusiones y trabajo futuro

Acompañando a las hipótesis presentadas, podemos ver cómo una complejización a la hora de implementar una solución trae mejoras significativas en lo que mediciones de calidad respecta: En caso de las heurísticas golosas, el funcionamiento es sencillo y su complejidad temporal es baja, pero cuentan con un diferencial en cuanto a la solución óptima considerablemente grande. En la otra cara de la moneda, podemos encontrar las variaciones de la metaheurística tabú search: El funcionamiento de estas es un poco más complejo, requiriendo estructuras adicionales y contando con un costo temporal de crecimiento más acelerado, pero cuentan con una mejora sustancial en cuanto a optimalidad. Más específicamente, se lograron diferenciales de menos del 1 %, algo muy buscado para solucionar problemas pertenecientes a la clase NP-hard.

Dependiendo del contexto de uso, como mencionamos anteriormente, podemos elegir uno por sobre otro. Siempre considerando el *trade off* entre costo y calidad de las soluciones. Por un lado, el incremento en cuanto a calidad de las soluciones obtenidas por los métodos de Tabú son significativamente mayores lo cual en muchos casos puede significar una predisposición natural a elegir este tipo de métodos. Sin embargo, por el otro lado la diferencia entre costos para obtener una calidad sustancialmente mayor en las soluciones no es algo despreciable en este caso y dependiendo el caso particular puede resultar prohibitiva la utilización de métodos tan costosos para la obtención de soluciones.

Cómo propuestas de trabajo a futuro nos quedó pendiente la exploración de nuevas heurísticas para la construcción de soluciones que no sean necesariamente golosas. En cuanto a la metaheuristica, algunas puntas de exploración puede incluir una variación entre los criterios de corte para la exploración de la metaheurística, el desarrollo de una función de aspiración, en donde se realicen excepciones a la prohibición de elementos ”tabú”, y además la búsqueda de formas alternativas para la selección de subvecindades. Consideramos que un análisis de las distintas partes que componen una metaheurística nos pueden llevar a un incremento significativo y una buena realación a nivel costo vs. calidad.