DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

TP 2: Problema de coloreo de máximo impacto

Algoritmos y estructuras de datos III

Integrante	LU	Correo electrónico
Lucas Di Salvo	446/18	lucasdisalvo@gmail.com
Muriel Picone Farías	***	***
Gabriel Sac Himelfarb	***	***
María Sol Sarratea	***	***

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
https://exactas.uba.ar

1. Introducción

Este trabajo tiene como objetivo proponer y evaluar empíricamente distintas soluciones a un problema de optimización combinatoria, el Problema de Coloreo de Máximo Impacto (en adelante, PCMI).
En los problemas de optimización combinatoria se busca obtener una solución que sea óptima según ciertos criterios que dependen de la definición del problema. En algunos casos, es posible encontrar la solución óptima en tiempo polinomial, pero hay otros problemas para los que no se conocen algoritmos polinomiales que permitan encontrarla, como los problemas *NP-hard*. Dado que es muy habitual encontrar problemas de este tipo en el mundo real, es necesario utilizar técnicas que permitan encontrar buenas soluciones y evaluarlas sin conocer la solución óptima. Esto se puede lograr mediante la utilización de heurísticas y metaheurísticas.
El PCMI pertenece a este grupo de problemas. Dados dos grafos $G = (V, E_G)$ y $H = (V, E_H)$, que tienen el mismo conjunto de vértices V pero diferentes conjuntos de aristas, y un conjunto de colores C, se define el impacto $I(c)$ sobre H de un coloreo $c : V \longrightarrow C$ de G como la cantidad de aristas $(i, j) \in E_H$ tales que $c(i) = c(j)$. Más informalmente, el problema consiste en hallar la máxima cantidad de aristas de H (el impacto) a cuyos extremos (vértices) se les puede asignar el mismo color, cumpliendo la restricción de que ese mismo coloreo debe ser válido en G.
El PCMI se utiliza para modelar problemas de asignación. Por ejemplo, consideremos la necesidad de organizar las reuniones en una empresa. Dado un conjunto de equipos de trabajo, que necesitan reunirse en determinado horario, puede ser deseable asignar las salas de reuniones de manera tal que cada equipo se reúna la mayor cantidad de veces posible en la misma sala. En este escenario, V representa el conjunto de reuniones, las aristas de G indican qué reuniones ocurren en el mismo horario y las aristas de H conectan las reuniones del mismo equipo. Los colores representan las salas.
Otra situación que puede modelarse en términos del PCMI es la organización de los horarios de un torneo deportivo. Por ejemplo, dado un conjunto de partidos, cuyos árbitros ya fueron asignados ya que por lo general solo se cuenta con un pequeño grupo de árbitros y hay diversas restricciones para su asignación, hay varios partidos que es deseable que se jueguen simultáneamente para reducir la posibilidad de especulación con los resultados y también poder organizarlos en una franja horaria acotada. En este caso, V representa el conjunto de partidos, y las aristas de G relacionan los partidos que tienen el mismo árbitro asignado (y por lo tanto no podrían jugarse en el mismo horario). Las aristas de H, por su parte, representan los partidos que se quiere que se jueguen simultáneamente. El objetivo es que la máxima cantidad de partidos cuyos resultados podrían influirse mutuamente se jueguen en el mismo horario, lo que es equivalente a maximizar la cantidad de aristas de H cuyos extremos están pintados del mismo color.
Con el fin de encontrar soluciones a estos problemas, se diseñaron dos heurísticas constructivas golosas y dos metaheurísticas que toman las heurísticas como punto de partida para ir haciendo los cambios necesarios que permitan mejorar las soluciones iniciales. En las secciones que se encuentran a continuación, se describen estos algoritmos, presentando tanto su funcionamiento general como sus detalles de implementación y su complejidad, para luego dar lugar a la sección de experimentación, donde se presentan y discuten los resultados del análisis empírico de los resultados obtenidos con las heurísticas y metaheurísticas propuestas.

2. Estructura de Representación

Como ya hemos mencionado, el PCMI se define sobre dos grafos, G y H, con un único n que representa la cantidad de vértices y dos valores que representan la cantidad de aristas, m_G y m_H. Cada instancia de problema está definida en un archivo de texto, cuya primera línea contiene los valores e n, m_G y m_H, separados por un espacio. En las líneas siguientes se incluyen las aristas, primero las correspondientes a G y luego las de H, con las coordenadas también separadas por espacio.
Como los algoritmos diseñados requerían recorrer todos los vecinos de cada vértice pero también

consultar si dos vértices estaban relacionados, para contar con esta flexibilidad y mantener la complejidad se decidió representar ambos grafos tanto con una lista de adyacencia como con una matriz.

3. Heurísticas Constructivas Golosas

A continuación se describen las dos heurísticas constructivas golosas implementadas para encontrar soluciones factibles para el PCMI. Estas heurísticas generan soluciones factibles construyéndolas paso a paso. En cada momento se enfrentan a una decisión (de qué color pintar un determinado nodo), y esta se toma siguiendo un criterio goloso. Vale la pena recordar que, como se trata de heurísticas, las soluciones generadas no serán necesariamente óptimas, pero se buscará seguir criterios que puedan brindar soluciones buenas en general.

3.1. Heurística 1

3.1.1. Descripción del algoritmo

1. Se pre-procesa el grafo H para obtener un grafo H', eliminando aquellas aristas que pertenecen también a G. Es decir, $E_{H'} = E_H - E_G$. Si dos nodos están conectados en G, no será posible pintarlos del mismo color en H y por lo tanto esa arista se puede descartar ya que no sumará al impacto.

2. Se ordenan los nodos por grado en H, de mayor a menor, utilizando un criterio goloso de que será conveniente colorear los nodos que afectan a mayor cantidad de aristas primero y así sumar impacto.

3. Luego se recorren los nodos. Si el nodo no está pintado, se analiza si alguno de sus vecinos lo está. En caso de que lo esté, se intenta pintar el nodo actual de ese color, considerando las restricciones que impone el coloreo de G. Si no, se pinta de un nuevo color.

4. Para elegir un nuevo color, se comienza por el color 0 y, para cada uno, se analiza si es posible pintar el nodo de ese nuevo color en G. Se selecciona el color con el mínimo valor.

5. Luego de pintado el nodo principal (el que se está recorriendo en la iteración actual), se recorren sus vecinos para ver si se pueden pintar del mismo color. Si ya están pintados, no se hace nada, y si no, se intentan pintar, revisando que no haya restricciones en G. Si no se pueden pintar, se dejan en blanco (y serán pintados por ser vecinos de algún otro nodo o cuando les llegue el turno como nodo principal en el recorrido).

A continuación, se incluye el pseudocódigo del algoritmo:

Algorithm 1 Heurística 1

```
1: function heuristica1(colores, G, H)
2:     preprocesar(G, H);
3:     ordenarNodosPorGrado(H)
4:     for V ∈ H do
5:         if coloreo(V) = UNDEFINED then
6:             coloresVecinos ← []
7:             for vecino ∈ vecinos(V) do
8:                 if coloreo(V) ≠ UNDEFINED then
9:                     coloresVecinos.agregar(coloreo(V))
10:            if vacio(coloresVecinos) then
11:                pintarDeNuevoColor(V)
12:            else
13:                color ← colorVecinoDelQueSePuedePintar(V, coloresVecinos)
14:                if color ≠ null then
15:                    coloreo(V) ← color
16:                    for vecino ∈ vecinos(V) do
17:                        if coloreo(vecino) = UNDEFINED ∧ sePuedePintarEnG(color, vecino) then
18:                            coloreo[vecino] = color
19:                else
20:                    pintarDeNuevoColor(V)
21:        else
22:            color ← coloreo(V)
23:            for vecino ∈ vecinos(V) do
24:                if coloreo(vecino) = UNDEFINED ∧ sePuedePintarEnG(color, vecino) then
25:                    coloreo(vecino) ← color
```

3.1.2. Análisis de complejidad

1. Preprocesamiento de H: se recorre todo H (nodos y aristas) ($O(2 \cdot m_H + n)$) y se chequea si la arista existe en G, representado como matriz de adyacencia ($O(1)$). La complejidad de esta operación es, entonces, $O(m_H + n)$.

2. El ordenamiento de los nodos por grado se realiza con el algoritmo de ordenamiento *bucket sort*, que tiene una complejidad de $O(n)$. Obtener la longitud de las listas de adyacencia tiene una complejidad de $O(1)$ ya que están representadas como vectores y se obtiene con la operación `size()`.

3. El algoritmo que realiza el coloreo recorre todos los vecinos de cada nodo, es decir que en total recorre n nodos y $2 \cdot m_H$ aristas (cada una se recorre dos veces porque aparece en las listas de adyacencia de los dos nodos). Por lo tanto, la complejidad de este recorrido es $O(m_H + n)$.

4. Entre las distintas condiciones, el peor caso que se puede dar es cuando para cada vecino, hay que recorrer todos los vecinos que tiene en G para ver si se pueden pintar, lo que tiene una complejidad de $O(m_H^2 \cdot m_G)$. Como se mencionó anteriormente se realizan $O(n + m_H)$ iteraciones, por lo que si el algoritmo realizara esta operación en todos los casos, la complejidad resultante sería de $O((n + m_H) \cdot (m_H^2 \cdot m_G))$.

5. En caso de que haya que pintar el nodo actual de un nuevo color, es necesario ir recorriendo los colores (que necesariamente serán $\leq n$) y para cada uno de ellos recorrer G, lo que tiene una complejidad de $O(n+m_G)$. Esto se hace para cada nodo, por lo que si en todas las iteraciones el algoritmo tuviera que realizar esta operación, la complejidad sería de $O(n^2 + m_G)$.

Por lo tanto, la complejidad total del algoritmo es de $O(n+m_H+n+n^2+m_H^2 \cdot m_H+n^2+m_G)$, dando una complejidad total de $O(n^2 + m_H^2 \cdot m_G)$.

3.2. Heurística 2

3.2.1. Descripción del algoritmo

1. Se ordenan las aristas de H del siguiente modo: dada la arista $(u, v) \in E_H$, se le asigna el valor $d_G(u) + d_G(v)$. A continuación, se ordenan las aristas de H en orden decreciente de acuerdo a los valores calculados.

2. En el orden determinado por el paso anterior, se comienzan a colorear los extremos de las aristas de H. La idea de realizar el coloreo en ese orden es que las aristas de H que inciden en vértices de mayor grado en G, serán más difíciles de colorear si se "dejan para el final", porque tendrán muchas más restricciones. Para cada arista (u, v) de H, en el orden ya definido, se hace lo siguiente:

   - Si ambos extremos están pintados, se dejan como están.
   - Si (u, v) no está en G, y si ambos extremos están sin pintar, se pintan ambos del mínimo color que es compatible con los vecinos de u y v en G. Si (u, v) está en G, no se pintan los vértices.
   - Si solo uno de los extremos está coloreado, el otro se colorea del mismo color siempre y cuando no entre en conflicto con sus vecinos en G. En caso de haber conflicto, se deja sin pintar. La decisión de dejarlo sin pintar es que elegir un color arbitrario podría perjudicar el coloreo a futuro: dejándolo sin pintar, ese nodo aún es "utilizable" por algún otro arista.

   Una vez finalizado el proceso, es posible que queden nodos sin pintar. Para colorearlos, se los recorre y se pinta cada uno con el menor color posible que hace que el coloreo sea válido en G.

El pseudocódigo del algoritmo corresponde al Algoritmo 2.

Algorithm 2 Heurística 2

```
1: function heuristica2(colores, G, H)
2:     ordenarAristasH
3:     maximoColorActual ← 0
4:     for (u, v) ∈ E_H do
5:         if (u, v) ∉ E_G then
6:             if colores[u] = colores[v] = UNDEFINED then
7:                 colores[u] ← min{k : ∄w ∈ Vecinos_G(v) ∪ Vecinos_G(u) tal que colores[w] = k}
8:                 colores[v] ← colores[v]
9:             else if colores[u] = UNDEFINED then
10:                if ∄w ∈ Vecinos(u) tal que colores[w] = colores[v] then
11:                    colores[u] ← colores[v]
12:            else if colores[v] = UNDEFINED then
13:                if ∄w ∈ Vecinos(v) tal que colores[w] = colores[u] then
14:                    colores[v] ← colores[u]
15:    for v ∈ V do
16:        if colores[v] = UNDEFINED then
17:            colores[v] ← min{k : ∄w ∈ Vecinos(v) tal que colores[w] = k}
```

3.2.2. Análisis de complejidad

La complejidad del algoritmo viene dada por:

- ordenarAristasH: Las aristas de H se guardan como tuplas de tres componentes: las dos primeras corresponden a los extremos, mientras que la tercera es la suma de los grados de los extremos en G, es decir, el valor según el cual se efectúa el ordenamiento. El método de ordenamiento empleado fue BucketSort: se crea un bucket por cada posible valor de la suma de los grados en G. ¿Cuál es el valor máximo de esta suma? Por un lado, en el peor caso cada uno de los nodos tiene grado $n-1$ en G, con lo que la suma es $O(2n) = O(n)$. Pero por otro lado, sabemos que en la suma, cada arista de G se cuenta a lo sumo una vez, salvo quizás una arista que una los dos vértices, que se cuente dos veces. De cualquier modo, es $O(m_G)$. Entonces, el bucketsort es en total $O(min\{n, m_G\} + m_H)$.

- Analicemos ahora la complejidad del for de la línea 4. Para en la primera guarda calcular el mínimo del conjunto especificado, se emplea un vector de valores booleanos (de tamaño el máximo color usado hasta el momento), inicializado en false en cada posición. Se recorren los vecinos de u y v y se colocan en true aquellos colores empleados. Finalmente, se recorre el arreglo buscando el mínimo color no usado hasta el momento (y si están todos usados se agrega un color nuevo). En la segunda y tercera guarda simplemente se recorren los vecinos y se verifica la condición. Podemos entonces identificar que tenemos en este for dos "tipos de acciones": recorrer los vecinos (realizado en las tres guardas) y crear y recorrer los arreglos booleanos de colores, solo en la primer guarda. El costo **total** a lo largo de todo el for de recorrer los vecinos es $2m_G = O(m_G)$, pues cada arista se chequea a lo sumo dos veces.

  Por otro lado, la primera guarda se ejecuta a lo sumo $\frac{n}{2}$ veces (ya que se ejecuta cuando los dos extremos están sin pintar, y además, cada vez que se ejecuta, pinta ambos extremos), y cada una de esas veces se recorre el arreglo booleano, que tiene a lo sumo n elementos (en realidad se puede mejorar la estimación, porque la cantidad de colores no es siempre la misma, pero el orden asintótico es el mismo), por lo que el recorrido de los arreglos lleva en total $O(n^2)$ operaciones. Por otro lado, otra posible cota es la siguiente: se realizan en total m_H iteraciones, y en cada iteración la cantidad de colores aumenta en 1 a lo sumo, por lo que en todas las iteraciones se terminan usando como mucho m_H colores, con lo que el recorrido de los arreglos es $O(m_H^2)$. Juntando las dos cotas, obtenemos que el procesamiento de los arreglos booleanos es $O(min\{n, m_H\}^2)$.

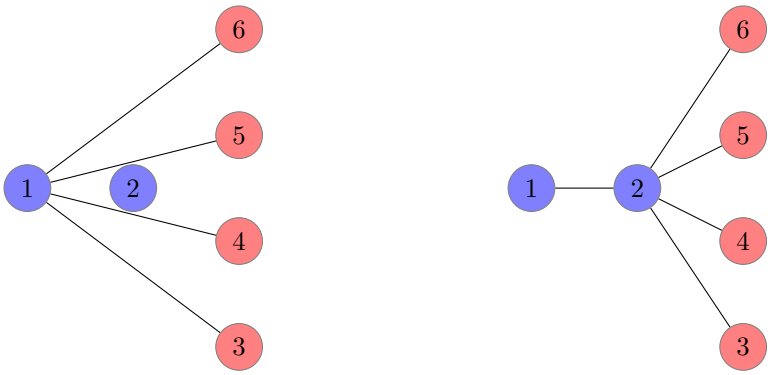


Figura 1: Solución de la Heurística 2 para instancia de peor desempeño. Izquierda: G, Derecha: H

- Para el for de la línea 15, que colorea los vértices que aún no fueron pintados, por un lado podemos estimar igual que antes que en los chequeos de vecinos, cada arista se revisa a lo sumo dos veces, luego en total esas verificaciones suman m$O(m_G)$; y por otro lado, nuevamente la búsqueda de los mínimos colores utilizables en cada iteración es $O(n)$, y a lo sumo se hacen n iteraciones, con lo que es $O(n^2)$ (nuevamente, se podría hacer una estimación más fina porque no puede haber n colores en todo momento, pero la complejidad resultará igual asintóticamente).

Juntando todos los resultados, obtenemos una complejidad de:
$O(min\{n, m_G\} + m_H + m_G + min\{n, m_H\}^2 + n^2) = O(n^2 + m_G + m_H)$

3.2.3. Casos de peor desempeño

Es posible encontrar instancias en las cuales la Heurística proporciona un coloreo de calidad tan baja como se desee. Para ver esto, se puede considerar una familia de grafos G_k, H_k, donde para cada k, G_k y H_k tienen $k+2$ nodos. $E_{G_k} = \{(1,3),(1,4),...,(1,k+2)\}$ y $E_{H_k} = \{(1,2),(2,3),(2,4),...,(2,k+2)\}$. En las Figuras 1 y 2 aparecen represantados G_4 y H_4.

El coloreo óptimo en G_k y H_k tiene impacto k, y se consigue pintando los vértices $2,3,...,k+2$ del mismo color, y 1 de un color diferente.

Sin embargo, la heurística comienza a colorear por la arista $(1,2)$ (dado que la suma de los grados de sus extremos en G es k, la mayor en todo el grafo, si $k > 1$), y el impacto máximo que se alcanza es 1.

Entonces esta familia de grafos cumple que el impacto de la solución brindada por la heurística es 1 en cualquier caso, y el impacto de una solución óptima es k, con k arbitrariamente grande.

4. Metaheurísticas

4.1. Metaheurística 1

La primer metaheurística utiliza una memoria basada en soluciones previamente exploradas, haciendo uso de la misma para la selección de futuras soluciones que pueda conducir a resultados útiles prohibiendo el tomar una solución que se encuentre en ella, pero con ciertas libertades a través del uso de una función de aspiración.

4.1.1. Generación de la vecindad

La generación de la vecindad se da en base a una solución previamente encontrada de la siguiente manera:

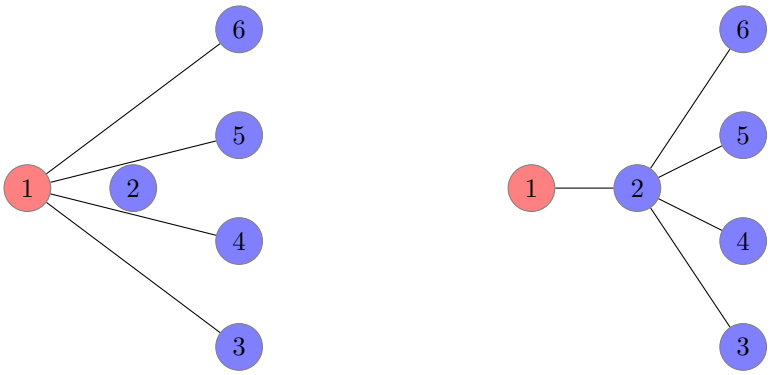


Figura 2: Solución óptima para instancia de peor desempeño. Izquierda: G, Derecha: H.

- Si el algoritmo se encuentra en la *primer iteración* del mismo, la vecindad se genera en base a la solución inicial generada por la heurística golosa utilizada.
- Si el algoritmo se encuentra en una iteración posterior a la primera, la vecindad se genera en base a la última mejor solución de una vecindad previa, la cual no necesariamente es la iteración previa (por ejemplo, si en la iteración previa ninguna de las soluciones de la vecidnad tenía un coloreo válido).

La vecindad contendrá soluciones que varían unicamente el color de un nodo en particular entre las mismas y la solución sobre la que se crea la vecindad.

En el algoritmo se decide una *cantidad* a drede de nodos que verán su color afectado, donde la elección de *cuáles* se veran afectados sucede de manera aleatoria. Para cada uno de estos nodos se generan tantas soluciones como colores posibles haya para colorear dicho nodo, exceptuando el color que originalmente tenía (la cantidad de colores disponibles es a su vez un parámetro del algoritmo).

De este modo, cada uno de estos coloreos hacen a la vecindad, siempre y cuando su coloreo sea válido, donde potencialmente habrá (aunque probablemente no tan frecuentemente) *cantidad de nodos afectados* $\times$ *colores disponibles* soluciones.

4.1.2. Memoria de soluciones

Para la memoria se decidió implementar un diccionario a través arbol de busqueda (más especificamente, un Trie), donde las claves serán cada una de las mejores soluciones encontradas (si las hubiera) en las iteraciones pasadas, y el significado, la iteración del algoritmo en la que se guardo.

Notar que esto permite que la busqueda e inserción de una clave se de en $O(n)$ (donde n es la cantidad de nodos del grafo), y el costo en memoria de la estructura es de $O(n \times k)$, donde k es la cantidad de claves ya almacenadas.

4.1.3. Función de aspiración

Durante la elección de la mejor solución de la vecindad generada en una iteración, se quiere ver si la misma provee un mayor impacto que la mejor solución encontrada hasta el momento. En esta situación puede haber dos escenarios, por un lado, que la solución no esté en la memoria, o por el otro, que sí lo esté.

En el primer caso, la función de aspiración no resulta relevante, pues la solución encontrada será guardada en memoria sin más. Es en el segundo caso donde toma relevancia; se decidió tomar por función de aspiración la cantidad de iteraciones pasadas desde que se guardo la solución hasta que fue encontrada nuevamente, si esta cantidad supera determinado umbral (el cual es un parámetro del algoritmo), la misma se vuelve a considerar para la iteración actual.

Posteriormente se procede a comparar su impacto con la mejor solución encontrada hasta el momento al igual que en todas las iteraciones.

4.1.4. Parámetros del algoritmo

Se consideraron los siguientes parámetros posibles de configurar para la primer metaheurística:

- `heuristica` : Permite decidir qué heurística utilizar apra la solución inicial.
- `max_it` : Permite decidir cuantas iteraciones durará el algoritmo.
- `max_it_sin_mejora` : Permite decidir cuantas iteraciones sin modificación de la mejor solución encontrada se permiten, si se llega a este valor la ejecución termina prematuramente.
- `max_cant_colores` : Permite decidir cuántos colores distintos son posibles de utilizar en los coloreos de las soluciones de una vecindad.
- `aspiracion` : Permite definir el umbral de iteraciones que utiliza la función de aspiración.
- `num_factor` : Permite definir la cantidad de nodos que serán modificados en la generación de soluciones vecinas.

4.1.5. Algoritmo

Algorithm 3 Metaheurística 1

function $metaheuristica1(colores, G_{lista}, G_{matriz}, H_{lista}, aristas_H, m_G)$
  $n \leftarrow |G_{lista}|$
  if `heuristica` **then**
    $colores \leftarrow heuristica1(colores, G_{lista}, G_{matriz}, H_{lista})$
  else
    $colores \leftarrow heuristica2(colores, G_{lista}, G_{matriz}, H_{lista}, aristas_H, m_G, n)$
  $S \leftarrow colores$ ▷ Mejor solución encontrada hasta el momento
  if `max_it`$= 0$ **then**
    $cantidadIteraciones \leftarrow n$
  else if `max_it`$= 1$ **then**
    $cantidadIteraciones \leftarrow n^2$
  else
    $cantidadIteraciones \leftarrow n^3$
  $iteracionesSinCambios \leftarrow 0$
  if `max_it_sin_mejora`$= 1$ **then**
    $maxSinMejora \leftarrow \sqrt{cantidadIteraciones}$
  else
    $maxSinMejora \leftarrow \frac{cantidadIteraciones}{2}$
  $colorMaximoAUtilizar \leftarrow 0$
  if `max_cant_colores`$= 0$ **then**
    $colorMaximoAUtilizar \leftarrow \#\{c : \text{c es un color de S}\}$
  else if `max_cant_colores`$= 1$ **then**
    $color \leftarrow max\{c : \text{c es un color de S}\}$
    $colorMaximoAUtilizar \leftarrow color + \sqrt{n - color}$
  else
    $colorMaximoAUtilizar \leftarrow n + 1$
  $memoria \leftarrow map < string, int > ()$
  if `aspiracion` $= 0$ **then**
    $iteracionesPermitidas \leftarrow n$
  else if `aspiracion` $= 1$ **then**
    $iteracionesPermitidas \leftarrow \frac{n}{2}$
  else
    $iteracionesPermitidas \leftarrow \sqrt{n}$
  if `num_factor`$= 0$ **then**
    $factor \leftarrow \frac{n}{2}$
  else
    $factor \leftarrow \sqrt{n}$
  $S_{aux} = colores$ ▷ Mejor solución de la iteración
  $i = 0$

while $i < cantidadIteraciones$ **do**
  $vecindad \leftarrow \{\}$
  $vecina = s_{aux}$
  $aCambiar \leftarrow \{0...n-1\}$
  $noIncluidos \leftarrow \{i \ : \ 0 \leq i \leq n-1\}$ ▷ $\#noIncluidos = n - factor$
  $aCambiar \leftarrow aCambiar \setminus noIncluidos$
  for $j \in \{0...|aCambiar| - 1\}$ **do**
    $original \leftarrow vecina[aCambiar[j]]$
    for $c \in \{0...|colorMaximoAUtilizar| - 1\}$ **do**
      if $c \neq original$ **then**
        $vecina[aCambiar[j]] \leftarrow c$
      if $valido(vecina, G_{lista})$ **then**
        $vecindad \leftarrow vecindad \cup \{vecina\}$
    $vecina[aCambiar[j]] \leftarrow original$
  $indiceS_{aux} \leftarrow -1$ ▷ Por si toda la vecindad está en memoria
  $impacto_{aux} \leftarrow 0$
  for $j \in \{0...|vecindad| - 1\}$ **do**
    $enMemoria \leftarrow memoria \cap \{vecindad[j]\} \neq \emptyset$
    $permitida \leftarrow enMemoria \wedge (i - memoria[vecindad[j]] > iteracionesPermitidas)$
    if $permitida \vee (\neg enMemoria \wedge imapcto_{aux} \leq impacto(H_{lista}, vecindad[j]))$ **then**
      $indiceS_{aux} \leftarrow j$
      $impacto_{aux} \leftarrow impacto(H_{lista}, vecindad[j])$
  if $indiceS_{aux} \neq -1$ **then**
    $S_{aux} \leftarrow vecindad[indiceS_{aux}]$
    $memoria[vecindad[indiceS_{aux}]] \leftarrow i$
  if $indiceS_{aux} \neq -1 \wedge impacto_{aux} \geq impacto(H_{lista}, S)$ **then**
    $s \leftarrow vecidnad[indiceS_{aux}]$
    $iteracionesSinCambios \leftarrow 0$
  else
    if `max_it_sin_mejora`$\neq 0$ **then**
      $iteracionesSinCambios \leftarrow iteracionesSinCambios + 1$
      if $iteracionesSinCambios \geq maxSinMejora$ **then**
        $i \leftarrow cantidadIteraciones$
  $i \leftarrow i + 1$
$colores = S$

4.1.6. Complejidad temporal

La complejidad temporal ha de medirse teniendo en cuenta lo siguiente:

- Los computos previos al ciclo principal están dominados temporalmente por el costo temproal de calcular la solución inicial, lo cual es $O(n + m_G \times m_H)$ en el caso de la primer heurística, y $O(n^2 + m_G + m_H)$ en el caso de la segunda.
- El valor de *cantidadIteraciones* vale a lo sumo n^3.
- La creación de la vecindad está ligada a los valores de $factor$ (el cual no excederá a n) y de $colorMaximoAUtilizar$ (el cual no excederá $n + 1$), es decir, se pueden llegar a crear n^2 soluciones vecinas. Dentro de esta generación, ha de verificarse si un coloreo es válido, y la complejidad temproal de esto pertenece a $O(m_G)$. En total, la complejidad temporal de la generación de la vecindad pertenece a $O(n^2 \times m_G)$.
- La elección de la mejor solución de la vecindad requiere que para cada solución se le calcule el impacto (la complejidad temporal de esto pertenece a $O(m_H)$). Teniendo en cuenta que habrá a lo sumo n^2 soluciones vecinas, la complejidad de la elección de la mejor solución de la vecindad pertenece a $O(n^2 \times m_H)$.

Dado que la cantidad de iteraciones serán a lo sumo n^3, se tendrá que la complejidad temporal total del algoritmo es $O(h + n^3 \times (n^2 \times m_G + n^2 \times m_H)) = O(h + n^5 \times (m_G + m_H))$, donde h representa el costo temporal de la heurística utilizada.

4.1.7. Complejidad espacial

La complejidad espacial ha de medirse teniendo encuenta lo siguiente:

- El vector de colores tiene una complejidad espacial perteneciente a $O(n)$.
- La mejor solución actual tiene una complejidad espacial perteneciente a $O(n)$.
- La mejor solución auxiliar (de cada iteración) tiene una complejidad espacial perteneciente a $O(n)$.
- Los cálculos con un vector que representa un coloreo tiene una complejidad espacial perteneciente a $O(n)$.
- Cada solución vecina tiene una complejidad espacial perteneciente a $O(n)$.
- El espacio utilizado por la vecindad está ligado al valor del $factor$ (el cual no excederá a n) y el valor del $colorMaximoAUtilizar$ (el cual no excederá a $n + 1$),dado cómo se generan las soluciones vecinas, puede llegar a a haber tantas como nodos con cada color (siempre cambiando un nodo a la vez), de modo que la complejidad espacial de la vecindad pertenece a $O(n^3)$.
- El espacio utilizado por la memoria está ligado al valor de *cantidadIteraciones* (el cual no excederá a n^3), el peor caso de complejidad para la memoria sería si nuinguna solución encontrada ya estaba guardada en la memoria, de modo que la complejidad espacial de la memoria eprtenece a $O(n^4)$.

Finalmente, la complejidad espacial del algoritmo está determianda por el espacio que ocupa la memoria, es decir, en el peor caso, sería $O(n^4)$.

4.2. Metaheurística 2

La segunda metaheurística implementada consiste en realizar Taboo search empleando memoria de estructura. A continuación, se detalla el desarrollo del algoritmo.

Figura 3: Instancia en que es necesario efectuar un Swap. G a izquierda, H a derecha

Figura 4: Instancia en que no bastaría cambio de color ni swap. G a izquierda, H a derecha

4.2.1. Definición de soluciones vecinas

Al momento de desarrollar la metaheurística, se debió optar por algún modo de determinar cuándo dos soluciones son vecinas. En un principio, se optó por la siguiente definición:

Dos soluciones son vecinas si difieren en el color de exactamente uno de sus nodos

Sin embargo, tomar como definición de vecindad únicamente este criterio provoca la aparición de problemas que impiden en ciertos casos que el grafo de soluciones sea conexo (y por lo tanto, es posible nunca obtener una solución óptima). Un ejemplo inmediato (pero que no es muy útil) es el caso de grafos completos: en un grafo completo, todo coloreo válido tendrá exactamente n colores. Hay exactamente $n!$ coloreos válidos (suponiendo que permitimos coloreos de hasta n colores y no más), que resultan de renombrar los colores (por lo que a efectos prácticos, son el mismo). Se observa que todas las soluciones son aisladas: si queremos pasar de una a la otra, no es posible hacerlo cambiando el color de un nodo por vez. Una forma de solucionar esto sería además permitir que dos coloreos que difieren en un "swap"sean vecinos.

La pregunta que surge es si esto no es un problema exclusivo de grafos completos; en ese caso, no habría inconvenientes en dejar la noción de vecindad previa. Sin embargo, el grafo de la Figura 3 muestra que no es así, y que puede ocurrir en otros casos.

Si no se permiten usar colores adicionales (ver Sección 4.2.5 más adelante) entonces no se puede modificar el color de ningun nodo. El impacto de este coloreo es 0, y el impacto óptimo es 1. Entonces es imposible llegar de la solución actual a la óptima! Esto motiva la siguiente definición:

Dos soluciones son vecinas si difieren en el color de exactamente uno de sus nodos, o si una se obtiene de la otra swapeando los colores de dos nodos

Supongamos que dos soluciones se pueden obtener una de la otra mediante el swapeo de los colores de los nodos u y v. Si $(u, v) \notin E_G$, entonces el swapeo se podía realizar como composición de la operación modificar el color de un nodo, dos veces (como el swapeo es factible, significa que ningún vecino de u tiene el color de v, porque además v no es vecino de u en G. Entonces, se puede cambiar el color de u al de v sin que aparezcan conflictos, y análogamente, luego se cambia el color de v al de u). Entonces, se optó finalmente por la definición:

Dos soluciones son vecinas si difieren en el color de exactamente uno de sus nodos, o si una se obtiene de la otra swapeando los colores de dos nodos vecinos en G

El último requisito se agregó con la idea de que al realizar un swap se esté realizando efectivamente una operación que sería imposible llevar a cabo de otro modo.

Se puede observar en la Figura 4 una instancia en la que si no permitimos agregar colores, entonces es posible que las definición de vecindad anterior siga sin ser suficiente. Sin embargo, se decidió mantener la definición, como tradeoff entre performance y complejidad del código.

Notar que todas las observaciones anteriores tienen sentido si no se permite el agregado de

cierta cantidad de colores. Si se permitiera usar nuevos colores en forma arbitraria, siempre se pueden realizar los swaps (se usarían los nuevos colores como "colores auxiliares").

4.2.2. Memoria de estructura

Se decidió almacenar en memoria aquellos nodos que fueron los últimos en ser modificados (donde el tamaño de la memoria es un parámetro que se ajustó durante la experimentación).

Se decidió guardar en la memoria únicamente aquellos nodos que fueron modificados siguiendo la noción de vecindad de cambiar el color de un vecino por vez, y no aquellos cuyo cambio provino de efectuar un swap. Esta decisión se tomó pensando que la cantidad de cambios por swap que se efectuarán será más chica (la idea sería que el cambio de color es un movimiento más crucial en el espacio de soluciones que el swap).

La idea detrás de impedir que nodos recientemente modificados lo sean nuevamente (es decir, volver taboo su cambio de color), es que se querría tener cierta "estabilidad" en la exploración del conjunto de soluciones. Si en todo momento se permitiera modificar cualquier nodo, el recorrido de las soluciones sería más caótico. Sería mejor intuitivamente fijar el color de algunos nodos para explorar una región determinada del grafo de soluciones en forma un poco más exhaustiva. Esto no ocurre si todos los nodos cambian constantemente de color.

Para almacenar los nodos taboo, se emplearon dos estructuras de memoria: una cola (FIFO) basada en lista (de la STL), y un arreglo booleano de tamaño n que indica en cada posición si el nodo está o no en la cola. La cola tiene como objetivo hacer eficiente el ingreso y salida de los nodos a la memoria, y el arreglo busca facilitar el recorrido de los nodos al momento de generar las vecindades.

El mantenimiento de ambas estructuras es como sigue: cuando se debe ingresar un nodo a la memoria, se coloca su posición en el arreglo como verdadera, se extrae el primer elemento de la cola y se pone su posición en falsa, y se ingresa el nuevo elemento a la cola. Observar que esto es $O(1)$.

4.2.3. Generación de vecindades

Para facilitar la generación de vecindades, se empleó una matriz de tamaño $n \times \text{maxColor}$ (donde maxColor es el mayor color que se permitirá utilizar). En la posición (i, j) se mantiene la información de la cantidad de vecinos en G que tiene el nodo i tales que están pintados del color j. A partir de esta estructura, es posible deducir en tiempo constante que:

- Si en la posición (i, j) hay un 0, entonces el nodo i es coloreable con el color j. Esto se puede verificar en $O(1)$.
- Si el nodo i_1 está pintado del color j_1, y el nodo i_2 está pintado del color j_2, con $(i_1, i_2) \in E_G$, entonces si en la posición (i_1, j_2) hay un 1 y en la posición (i_2, j_1) hay un 1, resulta que i_1 e i_2 son swapeables. Esto se puede verificar en $O(1)$.

El mantenimiento de la estructura es como sigue:

- Si se cambia el color de un solo nodo, se recorren sus vecinos en G, y se aumenta en 1 la posición del color nuevo, y se disminuye en 1 la posición del color anterior. (Esto es $O(d_G(v))$ si v es el nodo cambiado).
- Si se realiza un swap, se recorren los vecinos de los dos nodos involucrados, aumentando en 1 y disminuyendo en 1 las posiciones adecuadas. (Esto es $O(d_G(v_1) + d_G(v_2))$ donde v_1 y v_2 son los nodos swapeados).

Las vecindades se generarán de dos formas diferentes:

- Primera forma: Se generarán vecindades pequeñas, pero en forma "guiada", siguiendo ciertos criterios. Corresponde a la descripción que se da a continuación **Vecindad por cambio de color guiada** y **Vecindad por swap**. Qué tipo de vecindad se emplea en cada iteración está controlado por un parámetro de la experimentación, como se detalla en la Sección 4.2.4.

- Segunda forma: **Vecindad por cambio de color variable**.Se generarán vecindades de tamaños variables (ese tamaño será un parámetro para la experimentación), y no habrá un criterio elaborado que guíe las elecciones: para cada nodo, se prueba cambiar su color a cada uno de los k colores factibles más pequeños; y se calcula cuál de todos los cambios es el que genera un mayor impacto.

Vecindad por cambio de color Guiada Se recorren todos los nodos, y para cada uno que no esté en la lista taboo se selecciona un color factible con el criterio que se explica a continuación. Se analiza cuál de todos los cambios tiene lleva a un mayor impacto, y se hace el recoloreo. Dado un nodo, para elegir un color factible se siguen los siguientes pasos:

1. Se recorren sus vecinos en H, y se intenta pintarlo del color de alguno de ellos (la estructura descripta previamente permite hacer los chequeos fácilmente).

2. En caso de ser imposible, se intenta usar el menor color no usado (que sea menor que el máximo color permitido predeterminado como se explica en la Sección 4.2.5).

3. En caso de ser imposible, se elige al azar un color en forma uniforme entre los colores entre 0 y el máximo permitido, y se escoge el menor color que sea mayor que ese, de tal forma que el nodo sea coloreable. La decisión de que el algoritmo tuviese una parte randomizada es que se consideró que permitiría realizar una exploración del espacio de soluciones más rica, y potencialmente, obtener mejores resultados.

4. En caso de no funcionar esta última opción, no se considera recolorear el nodo.

Notar que el tamaño de la vecindad es a lo sumo n (en realidad, n menos el tamaño de la memoria), dado que para cada nodo, se busca elegir un único color (de forma 'inteligente') al que cambiarlo.

Vecindad por swap Se recorren los nodos, y para cada uno de ellos, se ve si es swapeable con alguno de sus vecinos en G (con la estructura el chequeo es $O(1)$). Para cada nodo se considera un único swapeo (el primero que se encuentra factible), y se determina qué swapeo entre los de todos los nodos, es el que lleva a un mayor impacto.

4.2.4. Parámetros de experimentación

Se consideraron los siguientes parámetros ajustables para la metaheurística 2:

- `max_it`: máximo número de iteraciones. Se fijará en n si vale 0, y n^2 si vale 1.

- `max_it_sin_mejora`: máximo número de iteraciones sin mejora que se permite seguir al algoritmo. Será $\sqrt{\text{cantidadIteraciones}}$ si vale 1; cantidadIteraciones/2 si vale 2; y no se tendrá en cuenta si vale 0.

- `max_cant_colores`: máximo número de colores que se intentará usar en los coloreos. Si vale 0 se usará la misma cantidad de colores que tenga el coloreo dado por la heurística golosa; si vale 1 se usarán $\text{maxColorHeuristica} + \sqrt{n - \text{maxColorHeuristica}}$; y si vale 2 se usarán n colores.

- `vecindad-size`: Si vale 0 se generarán las vecindades usando Vecindad por Cambio de Color Guiada. Si vale $k > 0$, se prueba colorear cada nodo con los menores k colores factibles. Si vale -1 se prueba colorear cada nodo con los menores $\sqrt{n}$ colores factibles, y si vale -2, con los menores $n/2$ colores factibles.

- `tipo_vecindad`: Si vale 0 no se emplean swaps. Si vale 1, se realiza un swap cada 5 iteraciones. Si vale 2 se realiza un swap cada 10 iteraciones.

- `memory_size`: El tamaño de la memoria de estructura. Si vale p, tendrá tamaño $p\,\%$ de la cantidad total de nodos.

Figura 5: Un posible coloreo con impacto cero y dos colores. G a izquierda, H a derecha.

Figura 6: Coloreo con impacto uno y tres colores

4.2.5. Máxima cantidad de colores

Uno de los parámetros tanto de la metaheurística 1 como la 2 es la máxima cantidad de colores que se emplearán al momento de buscar soluciones.

La motivación para tal parámetro es la siguiente: dada una solución brindada por una heurística golosa, que emplee k colores, se puede plantear la siguiente pregunta. ¿Es posible que la solución óptima tenga mejor impacto, y aun así emplee más colores? La respuesta es sí, y un ejemplo de tal situación se observa en las Figuras 5 y 6. Notar que si se quieren emplear únicamente dos colores, entonces los nodos 1 y 3 deben estar pintados del mismo color, y los nodos 2 y 4 también, siendo el color de 1 y 2 distintos, con lo que el impacto nunca podrá ser 1. Con tres colores se logra mejorar el impacto a uno.

Esto dice que puede ser necesario incorporar nuevos colores a la solución obtenida por la heurística, para obtener mejores impactos. La pregunta entonces es ¿cuántos colores adicionales es razonable agregar? Muy pocos colores puede reducir el espacio de búsqueda demasiado e impedir obtener soluciones cercanas al óptimo; demasiados colores puede volverlo demasiado grande y además, innecesariamente. Salvo que G sea completo, nunca usaremos n colores si queremos obtener un impacto no nulo. Se decidió por lo tanto agregar como parámetro en la experimentación el número de colores a usar.

4.2.6. Algoritmo

El pseudocódigo del algoritmo se encuentra en los Algoritmos 4, 5,6 y 7. No se indican los parámetros de entrada, como los grafos, y las variables de configuración, para facilitar la lectura.

Algorithm 4 Metaheurística 2

```
1: function metaheuristica2
2:     colores = heuristicaGolosa()
3:     mejorImpactoActual ← impacto(colores)
4:     mejorSolucionActual ← colores
5:     nodosProhibidos ← vector(n, false)
6:     inicializarMemoria(size)
7:     inicializarColoresPorNodo(colores, maxColor)
8:     menorColorNoUsado = maximoColorUsado(colores)
9:     for i = 1...cantidadIteraciones do vecindadCorrespondiente()
```

Algorithm 5 Vecindad por Cambio de Color Guiada

```
1: function vecindadChangeColorGuiada
2:     mejorImpactoVecindad ← 0
3:     mejorNodo ← 0
4:     mejorColor ← colores[0]
5:     for v ∈ V do
6:         if not nodosProhibidos[v] then
7:             impactoPosible ← 0
8:             eligioColor ← false
9:             for u ∈ Vecinos_H(v) do
10:                if coloreable(v, colores[u]) then
11:                    colorAnterior = colores[v]
12:                    colores[v] = colores[u]
13:                    eligioColor ← true
14:                    break
15:            if not eligioColor then
16:                if menorColorNoUsado < maxColor then
17:                    colorAnterior = colores[v]
18:                    colores[v] = menorColorNoUsado
19:                    eligioColor ← true
20:                else
21:                    colorAnterior = colores[v]
22:                    colores[v] = colorRandomFactible (a veces no es posible)
23:                    eligioColor ← true
24:            impactoPosible ← impacto(colores)
25:            if impactoPosible > mejorImpactoVecindad then
26:                mejorImpactoVecindad ← impactoPosible
27:                mejorColor ← colores[v]
28:                mejorNodo ← v
29:            if eligioColor then
30:                colores[v] ← colorAnterior
31:    colorAnterior ← colores[mejorNodo]
32:    colores[mejorNodo] ← mejorColor
33:    menorColorNoUsado ← maximoColorUsado(colores) + 1
34:    actualizarColoresPorNodo
35:    memoria.push(mejorNodo)
36:    nodosProhibidos[mejorNodo] ← true
37:    front ← memoria.pop()
38:    nodosProhibidos[front] ← false
39:    if mejorImpactoVecindad > mejorImpactoActual then
40:        mejorImpactoActual ← mejorImpactoVecindad
41:        mejorSolucionActual ← colores
```

Algorithm 6 Vecindad por Cambio de Color Variable

```
1: function vecindadChangeColorGuiada(cantidadColoresVecindad)
2:     mejorImpactoVecindad ← 0
3:     mejorNodo ← 0
4:     mejorColor ← colores[0]
5:     for v ∈ V do
6:         if not nodosProhibidos[v] then
7:             impactoPosible ← 0
8:             k ← 0
9:             cantidadColoresProbados ← 0
10:            while k < maxColor y cantidadColoresProbados < cantidadColoresVecindad do
11:                if coloreable(v, k) then
12:                    colorAnterior = colores[v]
13:                    colores[v] = k
14:                    impactoPosible ← impacto(colores)
15:                    if impactoPosible > mejorImpactoVecindad then
16:                        mejorImpactoVecindad ← impactoPosible
17:                        mejorColor ← k
18:                        mejorNodo ← v
19:                    colores[v] ← colorAnterior
20:                    cantidadColoresProbados + +
21:                k + +
22:    colorAnterior ← colores[mejorNodo]
23:    colores[mejorNodo] ← mejorColor
24:    menorColorNoUsado ← maximoColorUsado(colores) + 1
25:    actualizarColoresPorNodo
26:    memoria.push(mejorNodo)
27:    nodosProhibidos[mejorNodo] ← true
28:    front ← memoria.pop()
29:    nodosProhibidos[front] ← false
30:    if mejorImpactoVecindad > mejorImpactoActual then
31:        mejorImpactoActual ← mejorImpactoVecindad
32:        mejorSolucionActual ← colores
```

Algorithm 7 Vecindad por Swap

```
1: function vecindadSwap
2:     mejorImpactoVecindad ← 0
3:     mejorNodo1 ← 0
4:     mejorNodo2 ← 0
5:     colorAnterior1 ← 0
6:     colorAnterior2 ← 0
7:     for v ∈ V do
8:         impactoPosible ← 0
9:         eligioColor ← false
10:        for u ∈ Vecinos_G(v) do
11:            if swapeables(v, u) then
12:                colorAnterior1 ← colores[v]
13:                colorAnterior2 ← colores[u]
14:                colores[v] ← colorAnterior2
15:                colores[u] ← colorAnterior1
16:                swapped ← true
17:                break
18:        if swapped then
19:            impactoPosible ← impacto(colores)
20:            if impactoPosible > mejorImpactoVecindad then
21:                mejorImpactoVecindad ← impactoPosible
22:                mejorNodo1 ← v
23:                mejorNodo2 ← u
24:            colores[v] ← colorAnterior1
25:            colores[u] ← colorAnterior2
26:    colorAnterior1 ← colores[mejorNodo1]
27:    colorAnterior2 ← colores[mejorNodo2]
28:    colores[mejorNodo1] ← colorAnterior2
29:    colores[mejorNodo2] ← colorAnterior1
30:    actualizarColoresPorNodo
31:    if mejorImpactoVecindad > mejorImpactoActual then
32:        mejorImpactoActual ← mejorImpactoVecindad
33:        mejorSolucionActual ← colores
```

4.2.7. Complejidad temporal

Sea k el número de iteraciones que se efectuarán en total. Veamos en primer lugar las complejidades de las rutinas de Vecindad por Cambio de Color Guiada y Variable, y de Vecindad por Swap:

Vecindad por Cambio de Color Variable Se recorren todos los nodos; gracias a las estructuras empleadas, determinar si un nodo es o no taboo es $O(1)$. Si c es el valor $cantidadColoresVecindad$ (es decir, el dado por el parámetro `vecindad_size`, para cada nodo se prueban a lo sumo c colores. Para cada uno de ellos, se calcula el impacto obtenido en $O(m_H)$ (Esto se puede optimizar del siguiente modo: se mantiene el impacto actual, y al momento de recalcularlo, se tiene en cuenta solo el nodo que fue modificado y las aristas que inciden en él en H. Se decidió en un principio no implementar esta optimización para mantener mayor simplicidad en el código. Sin embargo, a posteriori, durante la etapa de experimentación, esta modificación fue incorporada dadas la sustancial mejora en el tiempo de ejecución). Todas las demás operaciones realizadas tienen complejidad $O(1)$. Notar que en realidad, al probar los colores para cada nodo, se realizan como mínimo $maxColor$ iteraciones del While, y en aquellas en que no se prueba colorear, las operaciones son $O(1)$. Se deduce entonces que el costo es $O(n.maxColor + n.c.m_H) = O(n^2 + n.c.m_H)$.

Vecindad por Cambio de Color Guiada

Se recorren todos los nodos. Verificar que son o no taboo se hace en $O(1)$ gracias al arreglo booleano $nodosProhibidos$. Para cada nodo no taboo, se revisan sus vecinos en H, y se intenta pintarlo del color de ese vecino. Saber si es o no pintable del color del vecino en H es $O(1)$ gracias a la matriz $coloresPorNodo$. Teniendo en cuenta que esta acción se realiza para todos los nodos, y que en total, cada arista de H se revisa a lo sumo dos veces, esta operación es $O(m_H)$ en total. En caso de no poder pintar el nodo del color de un vecino de H, se intenta utilizar un color nuevo (esto se hace en $O(1)$), y en caso de no ser posible, un color random (por la forma en que esto está implementado, se elige un color al azar, y se busca el menor color factible mayor que ese. Entonces es $O(maxColor) = O(n)$. Observar que en realidad esto no siempre se ejecuta).

Calcular el impacto del recoloreo se hace en $O(m_H)$. (Se puede optimizar como se describió antes). Las demás operaciones que se realizan en el for principal son $O(1)$. El for principal tiene un costo total de $O(n.m_H + n^2)$. Observar que con la optimización antes descripta, como se revisa una vez cada nodo para analizar la vecindad, el costo total entre todas las iteraciones de actualizar los impactos es de $O(m_H)$, con lo cual se obtendría una mejor complejidad $O(m_H + n^2) = O(n^2)$.

Fuera del for, las operaciones no constantes son obtener el máxmo color usado por el coloreo, que es $O(n)$; actualizar en caso de ser necesario la mejor solución actual, que es $O(n)$; y actualizar coloresPorNodo. El costo de este último es $O(d_G(v))$ donde v fue el nodo modificado. Si la memoria de estructura fuera grande (por ejemplo, de tamaño exactamente $n-1$), se modificarían todos los nodos, y recién ahí se podría repetir un nodo a modificar; entonces, en esas n iteraciones, la actualización sería en total $O(\sum_v d_G(v)) = O(m_G)$. Sin embargo, si la memoria es muy chica (en el peor caso de tamaño 0), se puede repetir siempre el mismo nodo, y podría ocurrir que ese nodo tenga grado exactamente m_G.

En conclusión, la complejidad en peor caso de Vecindad por Cambio de Color es $O(n^2 + n.m_H + m_G)$ (donde el término $n.m_H$ puede ser m_H con la optimización).

Vecindad por Swap El for principal se ejecuta n veces. Por cada nodo, se recorren sus vecinos en G y se verifica en $O(1)$ si son swapeables. Esto en total, entre todas las iteraciones, verifica a lo sumo dos veces cada arista en G, con lo que es $O(m_G)$. Las demás operaciones dentro del for son $O(1)$, a excepción del cálculo del impacto, que es $O(m_H)$, con lo que la complejidad total de este ciclo es $O(m_G + n + n.m_H) = O(m_G + n.m_H)$.

Las operaciones fuera del ciclo son $O(1)$, a excepción de actualizar la mejor solución actual (que es $O(n)$) y de actualizar los colores por nodo. Esta operación vimos que es $O(d_G(u) + d_G(v))$ donde u y v son los nodos a swapear. Esto es $O(m_G)$. Entonces, en total es $O(m_G + n.m_H)$. Nuevamente se puede reducir el término $n.m_H$ a m_H con la optimización descripta anteriormente.

Metaheurística 2 El costo total de la metaheurística, en caso de usar Vecindad por Cambio de Color Guiada y Swap, viene dado por:

- Complejidad de la heurística 1 o 2. En caso de usar la 2, es $O(n^2 + m_G + m_H)$.

- El cálculo del impacto de la heurística es $O(n)$.
- La inicialización de nodosProhibidos y de la memoria es $O(n)$.
- La inicialización de coloresPorNodo es $O(n.maxColor + m_G) = O(n^2 + m_G)$. El primer término corresponde crear la matriz, y el segundo, a mirar para cada nodos sus vecinos.
- Determinar el menor color no usado es $O(n)$.
- Si se realizan k iteraciones, y cada vecindad se emplea una proporción lineal en k de las veces (por ejemplo, 10 % de las veces la vecindad por swap, y el resto vecindad por cambio de color), entonces la complejidad es $O(k.(n^2 + n.m_H + m_G) + k.(m_G + n.m_H)) = O(k.n^2 + k.n.m_H + k.m_G)$.

La complejidad queda dominada por el último item, con lo que es: $O(k.n^2 + k.n.m_H + k.m_G)$. Para el caso particular de $k = n^2$, es igual a: $O(n^4 + n^3.m_H + n^2.m_G) = O(n^4 + n^3.m_H)$ (usamos que $m_G = O(n^2)$).

La complejidad en caso de usar Vecindad por Cambio de Color Variable, sería (en caso de usar la Heuristica golosa 2)$O(n^2 + m_G + m_H + n + n^2 + m_G + k.(n^2 + n.c.m_H))$, que es $O(kn^2 + k.n.c.m_H + m_G)$. En caso de tener $k = n^2$, y de fijar c constante, obtenemos: $O(n^4 + n^3.m_H + m_G) = O(n^4 + n^3.m_H)$.

4.2.8. Complejidad espacial

- Arreglo booleano de tamaño n. Es $O(n)$.
- Cola FIFO basada en lista, cuyo tamaño depende del parámetro `memorySize`, pero en cualquier caso es siempre menor que n. Es $O(n).$
- La estructura `coloresPorNodo` es una matriz de enteros de tamaño $n \times$maxColor . La cantidad de colores máxima a usar es n, con lo cual es $O(n^2)$.

Entonces la complejidad espacial es de $O(n^2)$.

4.2.9. Situación de peor desempeño

Una situación de peor desempeño para la metaheurística 2 ocurre cuando se corre del siguiente modo:

- La solución inicial se obtiene con la heurística 2
- No se permiten agregar más colores de los empleados por la solución de la heurística.
- G y H son como los mostrados en la Figura 7. Notar que se trata de una pequeña variante sobre los grafos de la Figura 1.
- Se emplea vecindad por cambio de color únicamente

El coloreo mostrado corresponde al realizado por la Heurística 2. Observar que si no se permite el uso de colores adicionales a los utilizados por la Heurística 2, entonces es imposible, mediante la modificación del color de un nodo a la vez, obtener un coloreo diferente.

Se observa que en forma análoga a la definición de los grafos G_k y H_k, se pueden obtener grafos en los cuales la metaheurística devuelva una solución de impacto tan malo como se desee (de impacto 1 en contraposición con el impacto óptimo k, con k arbitrariamente grande). El nodo 7, que no estaba presente antes, es el que impide que se cambie el color del nodo 2.

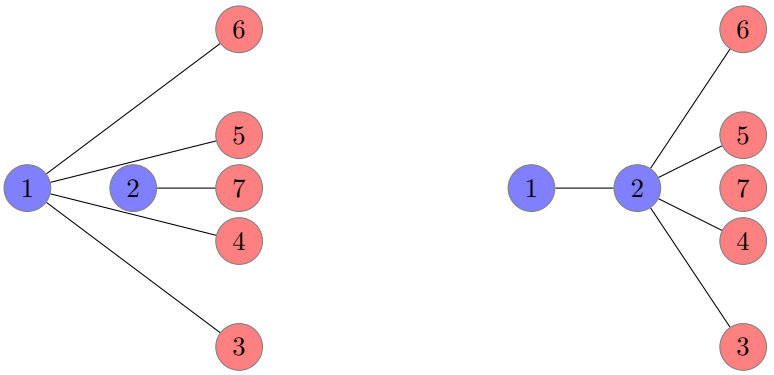


Figura 7: Instancias de Peor desempeño Metaheurística 2. Coloreo dado por el algoritmo.

5. Experimentación

5.1. Evaluación de las heurísticas golosas

5.1.1. Calidad de las soluciones con respecto al óptimo

Para evaluar la calidad de las soluciones obtenidas con las heurísticas, se evaluó el rendimiento de los algoritmos sobre un conjunto de instancias provistas por la cátedra, cuyo óptimo se conoce. La figura 8 muestra el error, en términos porcentuales, respecto del impacto óptimo, para cada instancia (identificadas en el eje x por el valor de n, ya que hay una única instancia para cada n).

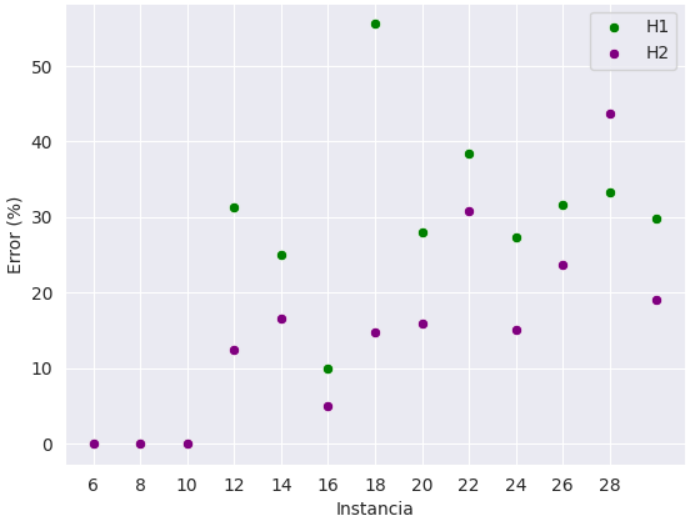
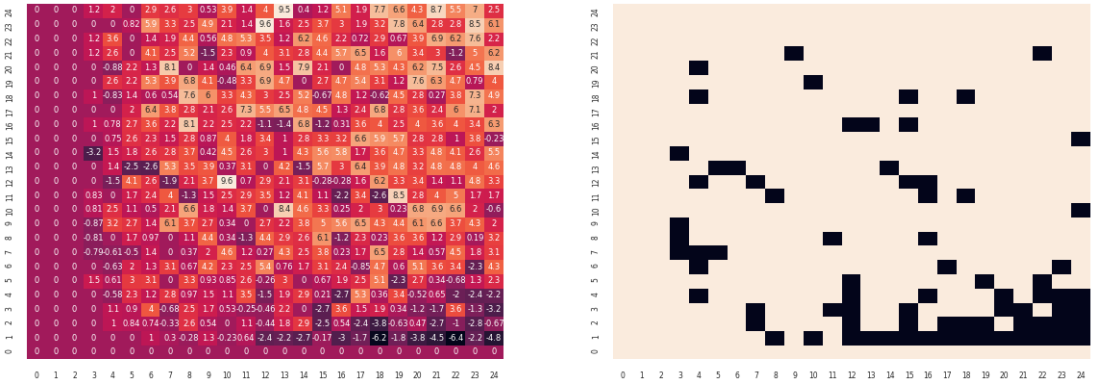


Figura 8: Comparación del error (respecto del impacto óptimo) para las heurísticas 1 y 2

Como se puede observar, en la mayoría de los casos la heurística 2 produjo un resultado de mejor calidad que la heurística 1, excepto para el caso $n = 28$. Para las instancias $n = 6$, $n = 8$ y $n = 10$ ambos algoritmos hallaron el óptimo. Otro dato que vale la pena mencionar es que para la heurística 2 en ningún caso el error supera el 50 % (de hecho, en todos los casos excepto en 22 se mantiene por debajo del 30 %) y que para la heurística 1 solo lo supera para $n = 18$.

5.1.2. Comparación de las heurísticas 1 y 2 para distintos valores de m_G y m_H

Dado que el conjunto de instancias cuyo óptimo se conoce es acotado, no solo por la disponibilidad de datos sino por las características del problema, se realizó una evaluación del rendimiento de ambas heurísticas sobre nuevas instancias.
Al no conocer el óptimo, en este caso no es posible medir el error de la misma forma que en el caso anterior. Por este motivo, se decidió comparar el rendimiento de las dos heurísticas sobre un conjunto de instancias con un n fijo, variando m_G y m_H, cono el objetivo de ver si alguna de las heurísticas tiene un mejor rendimiento de acuerdo con la variación de estos valores. Basándonos en los datos del experimento anterior, tomamos $n = 12$ por ser el n más pequeño para el que las heurísticas presentan diferencias.
Para este experimento, entonces, se generaron instancias para todas las combinaciones posibles de m_G y m_H entre 0 y $2n$, es decir, entre 0 y 24, y por cada una de estas combinaciones se crearon 50 instancias con aristas generadas al azar (pero sin repeticiones, para garantizar que la cantidad de aristas fueran efectivamente el m_G y el m_H esperados). Por lo tanto, se generó un total de 31250 instancias. Para cada subgrupo de instancias, es decir, los grupos de 50 instancias con el mismo m_G y m_H, se obtuvo el promedio.
A continuación se presentan los resultados del experimento. En el mapa de calor de la figura 9a se puede observar, para cada combinación de m_G y m_H, la diferencia porcentual entre el impacto obtenido por las heurísticas 1 y 2. Un valor negativo indica que la heurística 1 tuvo un mejor rendimiento, mientras que un valor positivo muestra que la solución arrojada por la 2 fue mejor. Los valores de m_H se representan sobre el eje x y los de m_G sobre el y. La figura 9b presenta estos mismos datos pero en forma discreta, indicando con color oscuro los casos en los que la heurística 1 funciona mejor que la 2 y en color claro el caso contrario.



(a) Diferencia porcentual entre $H1$ y $H2$ para distintas combinaciones de m_G y m_H

(b) Mapa de la heurística con mejor rendimiento para distintas combinaciones de m_G y m_H

Figura 9: Comparación del rendimiento de las heurísticas 1 y 2 para n fijo y distintos valores de m_G y m_H

A partir de lo que se puede observar en los gráficos, se confirma que la heurística 2 presenta un mejor funcionamiento general, pero se vislumbra una tendencia hacia un mejor funcionamiento de la 1 en los casos en los que H es más denso que G.

5.2. Experimentación sobre Metaheurística 1

En esta sección se desarrollan un par de experimentos que buscan ofrecer información que pueda ser de utilidad sobre la primer metaheurística (la cual utiliza una memoria basada en las

soluciones previamente exploradas).

Al momento de decidir con qué experimentar, nos encontramos con un ligero impedimento: La cantidad de posibles combinaciones de los parámetros del algoritmo resultaban demasiadas, y si se quería efectuar un análisis exhaustivo de los mismos, dichos experimentos consumirían un tiempo absurdo.

Recordemos los parámetros configurables del mismo :

- `heuristica` : Permite decidir qué heurística utilizar para la solución inicial.
- `max_it` : Permite decidir cuantas iteraciones durará el algoritmo.
- `max_it_sin_mejora` : Permite decidir cuantas iteraciones sin modificación de la mejor solución encontrada se permiten, si se llega a este valor la ejecución termina prematuramente.
- `max_cant_colores` : Permite decidir cuántos colores distintos son posibles de utilizar en los coloreos de las soluciones de una vecindad.
- `aspiracion` : Permite definir el umbral de iteraciones que utiliza la función de aspiración.
- `num_factor` : Permite definir la cantidad de nodos que serán evaluados en la generación de soluciones vecinas.

De la misma manera, se decidió priorizar el analizar lo que quizás podía proveer mayor información sobre la eficacia y eficiencia del algoritmo, que a su vez era lo que resultaba de mayor interés.

Para ello, los siguientes parámetros se fijaron en un valor predeterminado :

- `heuristica = 0` : El valor `0` representa la segunda heurística golosa que se diseño (el valor `1` representa la primera). La razón detrás de esto es que en las instancias ofrecidas, la segunda heurística tendía a dar un mejor resultado inicial, con lo que para reducir la cantidad de experimentos, se decidió tomar este valor como predeterminado.
- `max_it = 1` : El valor `1` representa n^2 iteraciones (`0` representa n y `2` representa n^3). La razón detrás de esta decisión es que tomar n^3 consumía una cantidad no trivial de tiempo por instancia y esto iba a afectar el cómo podríamos con los tiempos para los experimentos. Por otro lado, el no utilizar n iteraciones nada más se debió a que había situaciones donde tal cantidad no alcanzaba para encontrar una solución notablemente mejor que la encontrada por la heurística golosa (es decir, no había suficientes iteraciones para analizar el espacio de búsqueda disponible).

Los experimentos realizados en sí, exploran en cierta medida cómo las instancias ofrecidas ven afectada si eficiencia y eficacia a medida que estos parámetros cambian. Particularmente, se decidió explorar el comportamiento de la memoria, y el comportamiento de la vecindad.

Los siguientes experimentos fueron realizados en una computadora de la siguientes especificaciones:

- CPU AMD® Ryzen 3 2200g 3.5GHz y 8GB de memoria RAM,

y utilizando el lenguaje `C++` para los algoritmos, y scripts en `Python 3` para correr los mismos.

Se decidió efectuar 20 instancias de cada combinación de parámetros elegida para obtener resultados promedios en los que confiar (un número mayor podría no arrojar nada más de interés y consumir demasiado tiempo, y con un número menor se corre el riesgo de que los valores hallados no sean realistas).

5.2.1. Experimento 1 : Comportamiento de la memoria

En el primer experimento se busca explorar el comportamiento de la memoria de la búsqueda tabú, la cual almacena soluciones previamente exploradas. Más precisamente, durante el transcurso de las iteraciones, en cada una de ellas existe la posibilidad de que se genere una vecindad de soluciones factibles, de ellas se elegirá la que provea un mayor impacto siempre y cuando suceda que esta no está en la memoria, o está y la función de aspiración permite que la misma vuelva a ser visitada (esto último toma lugar cuando pasaron suficientes iteraciones desde que dicha solución fue guardada en la memoria).

Para este experimento, se decidió asignar un valor predeterminado a los siguientes parámetros que maximizaran el espacio de búsqueda dentro de los valores disponibles (y porque que no intervenían directamente en el comportamiento de la memoria):

- `max_cant_colores = 2` : El valor `2` representa que se pueden utilizar hasta $n + 1$ colores.
- `factor = 0` : El valor `0` representa que se elegirán $\frac{n}{2}$ nodos en cada iteración sobre los que probar colores para las soluciones vecinas.

Por otro lado, los parámetros que sí resultan de interés para las combinaciones son :

- `max_it_sin_mejora` : Si bien este parámetro no influye en cómo se crea la memoria, sí interactúa con cómo se la revisa, o con la función de aspiración directamente, pues el que se permita una cantidad muy chica de iteraciones sin mejora de la mejor solución (global), podría impedir que la función de aspiración revise soluciones que ya estaban en la memoria, y llegar a una solución con un mayor impacto. Es por eso que es interesante ver cómo la combinación de este parámetro interactúa con la función de aspiración y que resultados se obtienen. Los valores del mismo que se utilizaron en este experimento son los siguientes :
  - `max_it_sin_mejora = 0` : Hace que no se tenga en cuenta la cantidad de iteraciones sin mejora, con lo que el algoritmo no se detendrá prematuramente en ningún caso.
  - `max_it_sin_mejora = 2` : La cantidad de iteraciones sin mejora permitidas serán $\frac{max\_it}{2}$.
- `aspiracion` : Este parámetro trabaja directamente con la memoria, y representa la función de aspiración del algoritmo. Definirá la cantidad de iteraciones necesarias para que una solución encontrada sea vuelta a considerar si ya estaba previamente en la memoria. Sus valores son los siguientes :
  - `aspiracion = 0` : La cantidad de iteraciones necesarias es n.
  - `aspiracion = 1` : La cantidad de iteraciones necesarias es $\frac{n}{2}$.
  - `aspiracion = 2` : La cantidad de iteraciones necesarias es $\sqrt{n}$.

Los resultados encontrados fueron graficados en una primer imagen (Figura 10), muestran una ligera diferencia entre el error cometido por `max_it_sin_mejora = 0` y `max_it_sin_mejora = 2` (siendo este último menor en una cantidad mayor de ocasiones).

Adicionalmente, la combinación de `max_it_sin_mejora = 2` junto con `aspiracion = 1` pareciera tener la menor cantidad de instancias donde los resultados hallados se alejaron del óptimo (esto también puede ser visualizado en la Figura 11)

Por último, se quiso saber qué valores de la combinación ofrecían mejores soluciones para futuros experimentos, esto se hizo calculando la suma del error total acumulado (que es un reescalamiento del promedio) del experimento para cada combinación de parámetros y se puede visualizar en la Figura 12.

Los resultados arrojados resultaron particularmente interesantes : Cuando efectivamente se utiliza un límite de iteraciones sin mejoras, el error total no varía tan drásticamente y de hecho es menor que cuando este límite no se tiene en cuenta.

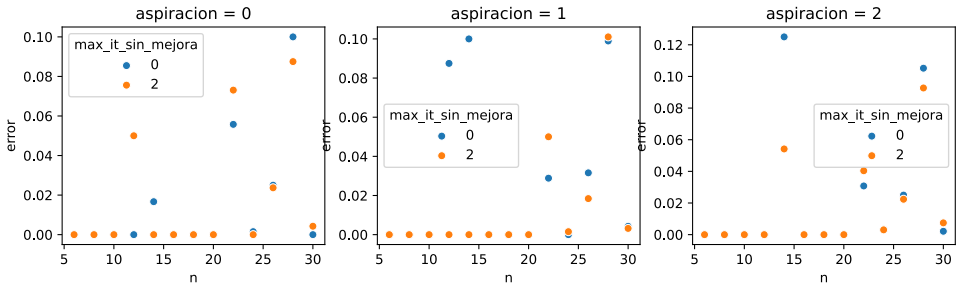


Figura 10: Resultados en funcion de la función de aspiración.

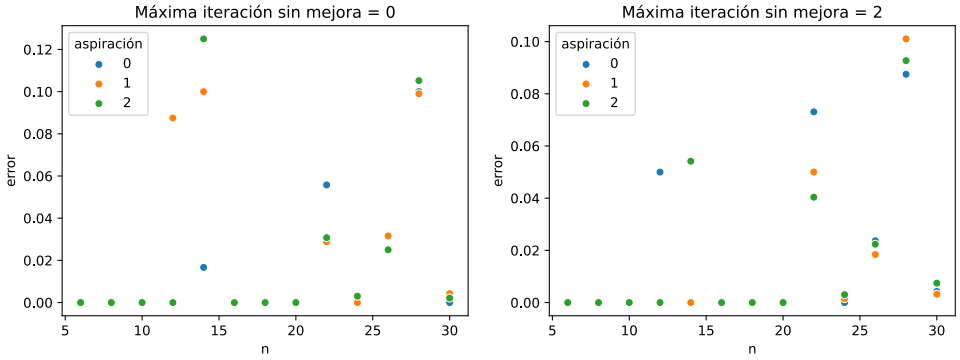


Figura 11: Resultados en función de la máxima cantidad de iteraciones sin mejora

Es más, hay una correlación particular: si se tiene en cuenta la aspiración, el error total correspondiente a `aspiracion = 0` es el mayor de cuando se tiene en cuenta este límite, y el menor cuando no se lo tiene en cuenta. Esto puede interpretarse como que si no existe este límite, el valor asociado al parámetro de la función de aspiración (n iteraciones), alcanza a revisar un espacio de soluciones más amplio y ofrece mejores resultados, mientras que si se tiene en cuenta el límite de iteraciones sin mejora (donde el valor asociado es $\frac{\texttt{max\_it}}{2}$) puede no ser suficiente para revisar soluciones que estuvieran previamente en la memoria y que pudieran conllevar el descubrimiento de una mejor solución. Este fue un descubrimiento no previsto en el planeamiento del experimento.

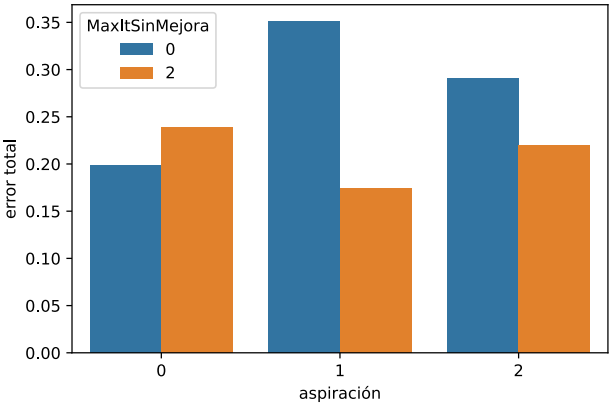


Figura 12: Error total de las combinaciones para el experimento 1

5.2.2. Experimento 2 : Comportamiento de la vecindad

En este segundo experimento se busca analizar el comportamiento de la vecindad para el algoritmo diseñado, la misma se crea frente a agarrar la última mejor solución de una iteración pasada, y modificar un nodo a la vez con un color a la vez, de esta manera si tengo 2 nodos a modificar, con 4 colores disponibles, habrá 6 soluciones resultantes (3 soluciones que modificaban el color del primer nodo, y 3 que modificaban el segundo, hay que tener en cuenta que uno de los colores disponibles era el que previamente ya tenía el nodo y esto no resulta en una solución vecina), que se distinguirán de la solución sobre la cual se genera la vecindad por un nodo con el color modificado, cada una.

Para este experimento, se decidió asignar un valor predeterminado a los siguientes parámetros que el impacto obtenido, frente a observar los resultados obtenidos en el primer experimento:

- `max_it_sin_mejora = 2` : El valor 2 representa que se va a considerar hasta $\frac{\texttt{max\_it}}{2}$ iteraciones sin mejora para detener la ejecución del algoritmo prematuramente.
- `aspiracion = 1` : El valor `1` representa que luego de $\frac{n}{2}$ iteraciones, si una solución vecina encontrada ya estaba previamente en la memoria, se la puede volver a considerar.

Por otro lado, los parámetros que sí resultan de interés para las combinaciones son :

- `max_cant_colores` : Este parámetro determina la cantidad disponible de colores a utilizar para colorear cada nodo que lo necesite. Donde sus valores elegidos a probar son los siguiente manera :
  - `max_cant_colores = 0` : La cantidad de colores disponibles es la misma que colores distintos hallados en la solución inicial ofrecida por la heurística golosa.
  - `max_cant_colores = 2` : La cantidad de colores disponibles es igual a $n + 1$, esto garantiza que por nodo siempre haya una solución válida (por ejemplo, cuando se tiene un grafo completo y no se quiere tampoco la solución con el mismo color que ya tenía el nodo, pues sino no sería una solución vecina)
- `factor` : Este parámetro determina la cantidad de nodos a modificar en el algoritmo, la elección de cuales se modificaran sucede de forma aleatoria, y los valores de este parámetro son los siguientes:
  - `factor = 0` : Se van a colorear $\frac{n}{2}$ nodos.
  - `factor = 1` : Se van a colorear $\sqrt{n}$ nodos.

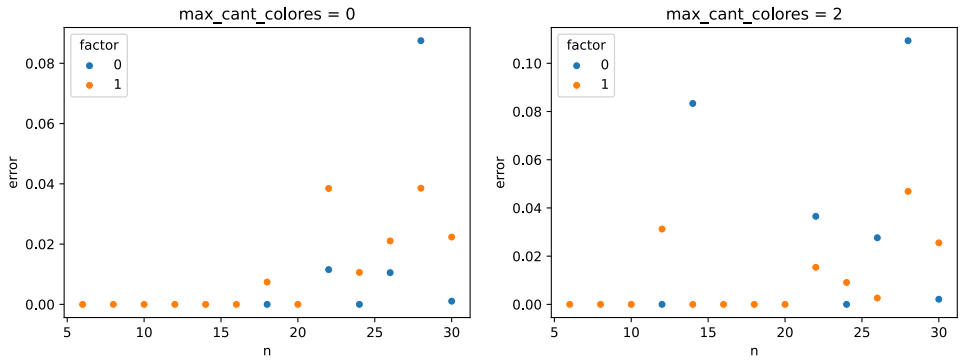


Figura 13: Resultados en función de `max_cant_colores`

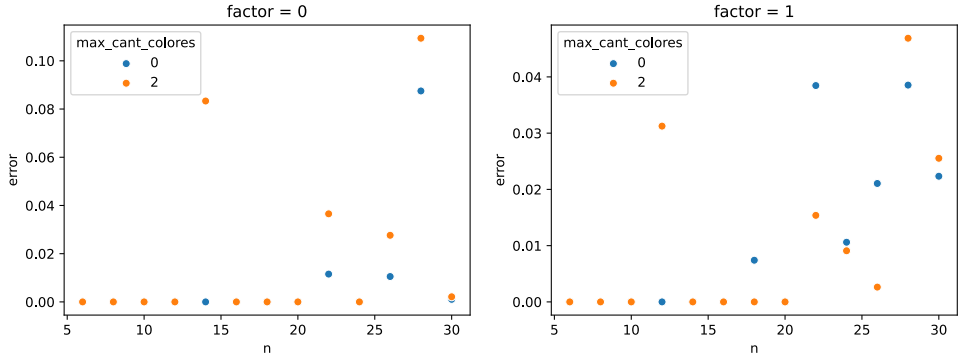


Figura 14: Resultados en función de `factor`

En los resultados arrojados por las combinaciones de parámetros, se puede visualizar tanto en la Figura 13 como en la Figura 14 que `factor = 0` junto con `max_cant_colores = 0` produce menos error, y que por el contrario el uso de `max_cant_colores = 2` ofrece en general un error mayor.

Por último se quiso ver el error total acumulado al igual que en el primer experimento, para las combinaciones tratadas. Esto pudo revelar que al utilizar una menor cantidad de colores, el error no tuvo variaciones abruptas en los resultados, mientras que en el caso donde se tienen más, el error varía entre una mayor cantidad de errores cuando se tienen más nodos para colorear, y un error menor cuando se tienen menos

Los resultados arrojados muestran que la configuración `max_it_sin_mejora = 0` y `aspiracion = 0` es la que provee mejores resultados.

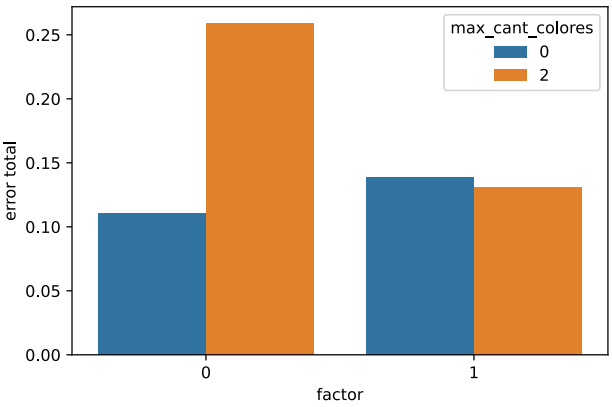


Figura 15: Error total de las combinaciones para el experimento 2

5.3. Experimentación Sobre Metaheurística 2

Al momento de realizar experimentos sobre la Metaheurística 2 con las instancias dadas, con el objetivo de escoger una combinación para sus parámetros que resultara en un mayor rendimiento, se encontró la siguiente dificultad: la cantidad de instancias disponibles de óptimo conocido era demasiado pequeña como para permitir la obtención de información representativa que guiara la elección de parámetros. En los experimentos inicialmente realizados sobre estas instancias, en los cuales se mantenían todos los parámetros fijos, excepto uno, que se variaba, se obtuvieron resultados de extrema calidad para todos los valores del parámetro. De las 13 instancias disponibles, se obtenían resultados óptimos entre 8 a 11 de ellas aproximadamente; y en las restantes, el error era muy pequeño. Por lo tanto, al momento de hacer comparaciones, no se contaba con diferencias sustanciales, que en parte podrían llegar a deberse a que realmente no hubiera relación entre el parámetro modificado y el impacto, o bien, a la poca representatividad de la muestra empleada.

Se optó entonces por realizar experimentos con instancias nuevas.

Los experimentos a continuación fueron realizados en una computadora con las siguientes especificaciones: CPU Intel Core i5-4210U 1.7 GHz, 8GB de memoria RAM.

5.3.1. Experimentación con instancias de peor desempeño

Se experimentó con la instancia de peor desempeño descripta en la Sección 4.2.9, y se verificó que en efecto el mayor impacto alcanzado por la metaheurística fue 1.

5.3.2. Experimentación con instancias nuevas generadas

El principal inconveniente al momento de evaluar las metaheurísticas empleando instancias generadas, es que se desconoce su valor de impacto óptimo, y por lo tanto no es posible calcular el error relativo de la solución encontrada.

Notar que si se desea experimentar con instancias de diferentes tamaños y características, tampoco tendría sentido emplear una escala de impacto absoluto, pues este puede variar radicalmente entre una instancia y otra.

Para evaluar entonces la performance de la metaheurística sobre estas instancias, se hizo lo siguiente: una cota conocida para el impacto de una solución es m_H. Medir el error de una solución respecto de este valor no tiene demasiado sentido en principio, pues el óptimo puede diferir de m_H tanto como se desee. Una mejora respecto de esta cota se obtiene preprocesando el grafo, eliminando aquellas aristas de H que también pertenezcan a G. Llamamos m'_H a la cantidad de aristas en H luego de esta transformación. Además, se decidió agrupar las instancias generadas dentro de dos grupos: instancias con G denso e instancias con G ralo. La idea es que intuitivamente,

para G denso, la diferencia entre el óptimo y m'_H (incluso después del procesamiento) puede ser grande, mientras que para G ralo se esperaría que la diferencia fuese más pequeña. La idea es que si mezcláramos instancias de ambos tipos, calcular el promedio de los errores relativos a m'_H para instancias del mismo n no daría información muy significativa.

5.3.3. Generación de instancias

Se generaron instancias con n variando entre 20 y 70 del siguiente modo: para cada n se generan una instancia con G denso y una instancia con G ralo. El valor m_H es un entero elegido al azar uniformemente en el intervalo $[1, \frac{n(n-1)}{2}]$, mientras que el valor de m_G se elige al azar uniformemente en el intervalo $[1, n]$ para el caso ralo, y $[\frac{n^2}{4}, \frac{n(n-1)}{2}]$ para el caso denso.

5.3.4. Realización de experimentos y componente de azar

La generación de vecindades por Cambio de Color Guiado tiene una componente azarosa. Por lo tanto, al correr los experimentos, lo ideal sería hacerlo una cantidad de veces suficiente para que los resultados obtenidos sean signficativos. Por cuestiones de tiempo, la cantidad de veces que se ejecutó cada instancia fue solo de 5. Sin embargo, se debe tener en cuenta que además se generaron 2 instancias por cada n: con la posibilidad G denso/ralo; y que los análisis realizados no son para instancias específicas, sino para conjuntos de instancias (se agruparon las instancias de n en $[20, 30)$,$[30, 40)$, etc).

5.3.5. Experimento 1: Tamaño de memoria de estructura

Se fijaron los parámetros de experimentación del siguiente modo:

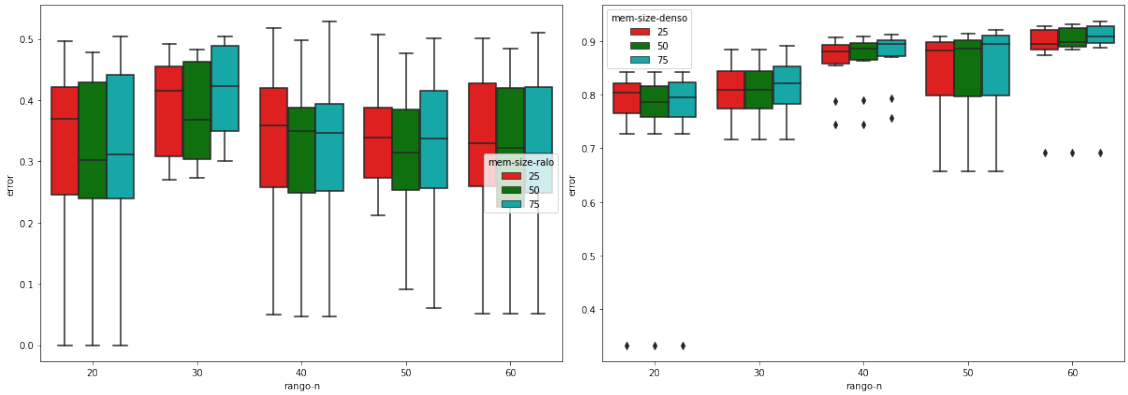
- Heurística golosa para la solución inicial: 2
- Máximo número de iteraciones: n^2
- Máximo número de iteraciones sin mejora: no se tiene en cuenta
- Máxima cantidad de colores: n
- Tipo de vecindad: vecindad por cambio de color guiada
- Vecindad size: sin efecto

y se hizo variar el parámetro de `memory_size` tomando los valores: 25, 50 y 75 %.

Los resultados obtenidos de impacto se muestran en la Figura 16. Se puede notar que en el caso de instancias con G denso el valor de los errores es mucho más elevado que en el caso de las instancias con G ralo, como se observó en la Sección previa.

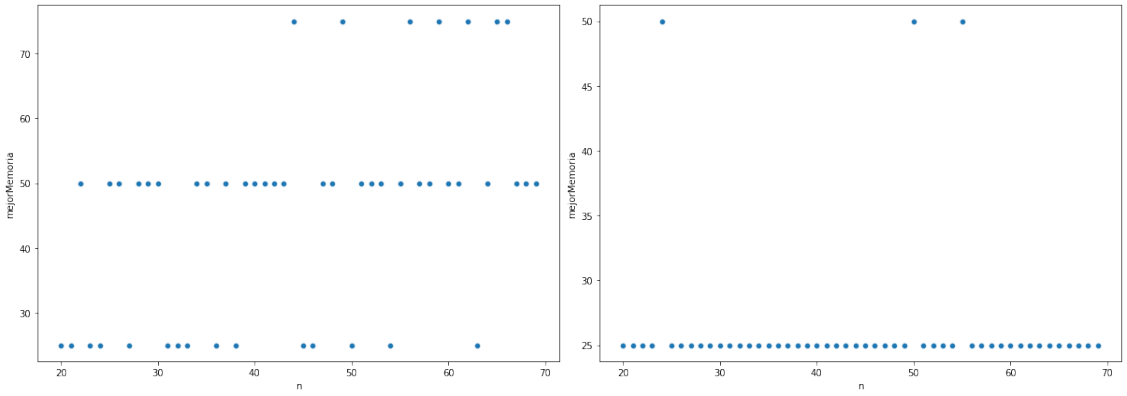
Las diferencias obtenidas en error relativo para los diferentes tamaños de memoria no son en general demasiado pronunciadas. Se observa una ligera tendencia en las instancias ralas, a una mejor performance para un tamaño de memoria de 50 %, y peor performance para memorias de tamaño 25 % (es importante notar que los rangos de los errores son muy similares para los tres valores del parámetro. Sacar conclusiones entonces no es muy seguro, dado que en muchos de los casos, los errores varían en su distribución, y no tanto en su rango. Que los rangos sean tan amplios se debe probablemente a que m'_H puede ser una cota mala para el impacto). Para el caso de las instancias densas, y con valores de n a partir de 30, la performance es ligeramente mejor para memorias más pequeñas. Una posible explicación para este fenómeno es que para instancias densas los nodos "buenos" que tienen la capacidad de mejorar el impacto son pocos, y por lo tanto revisitar nodos "buenos" sería difícil dado que hay que esperar más iteraciones a que deje de ser taboo si la memoria es grande.

Otra forma de visualizar los datos consiste en observar, para cada instancia qué tamaño de memoria fue el que produjo el mayor impacto (en promedio entre las 5 ejecuciones). Los resultados son los que se observan en la Figura 17. Notar que coinciden con las conclusiones sacadas



(a) (b)

Figura 16: Error relativo a m'_H en función de n variando el tamaño de la memoria para instancias con a) G ralo b) G denso



(a) (b)

Figura 17: Tamaño de memoria que brindó mayor impacto para cada n. a)Instancias con G ralo b)Instancias con G denso

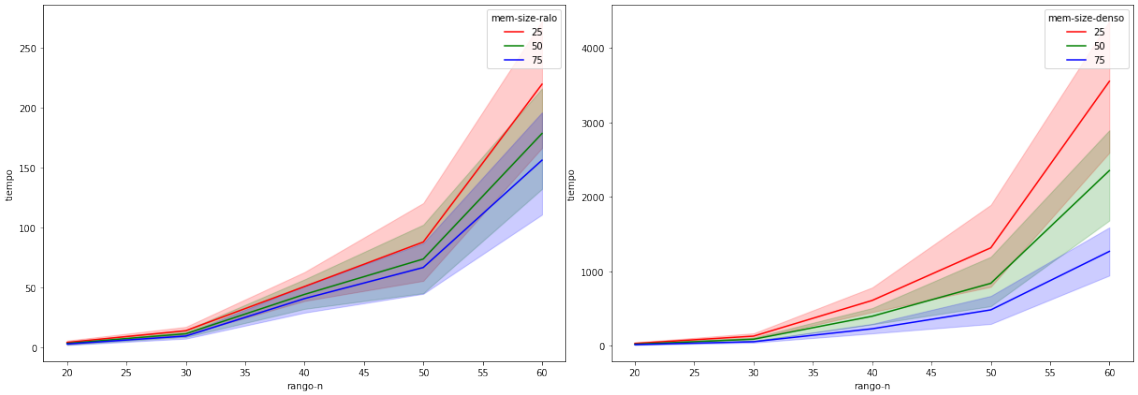
previamente a partir de los gráficos anteriores: para el caso G ralo, es más conveniente un tamaño de memoria de 50 %, mientras que para el caso G denso, es mejor 25 %. La principal desventaja de esta forma de visualización de los datos es que no se tiene una noción de cuánto mejores son las soluciones dadas por cada tamaño de memoria.

Respecto a tiempos de ejecución, previo a la realización del experimento, se consideró que dado que las operaciones de manejo de memoria de estructura son de tiempo constante, entonces el tiempo de ejecución no sería decisivo a la hora de escoger un tamaño sobre otro. Sin embargo, esta suposición era errada, como se aprecia en la Figura 18. Emplear tamaños de memoria menores incrementa el tiempo de ejecución, no porque las operaciones sobre la estructura de memoria aumenten de costo, sino porque las vecindades son mayores al haber menor porcentaje de nodos taboo.

Como decisión de compromiso entre tiempo de ejecución, e impacto, se decidió tomar un memory size de 50 % de la cantidad de nodos.

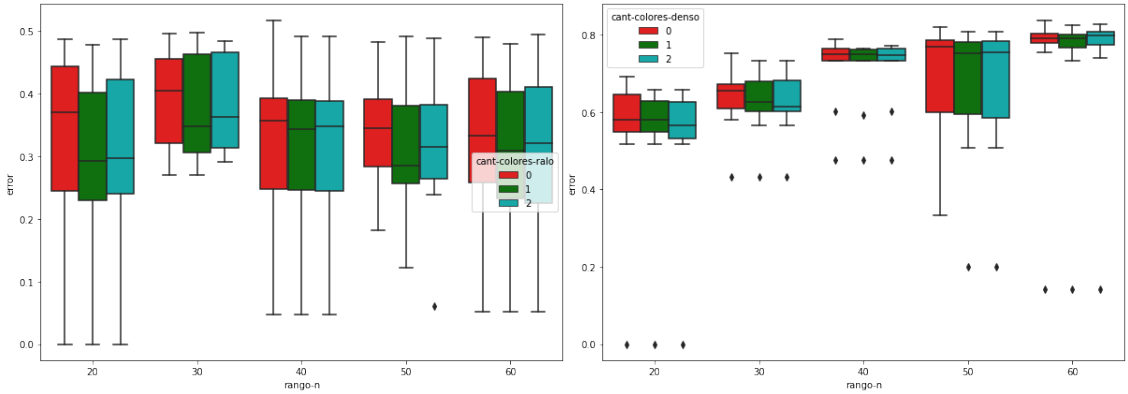
5.3.6. Experimento 2: Máxima cantidad de colores

Se fijaron los parámetros de experimentación del siguiente modo:



(a) (b)

Figura 18: Tiempo de ejecución en ms en función de n para instancias con a) G ralo b) G denso

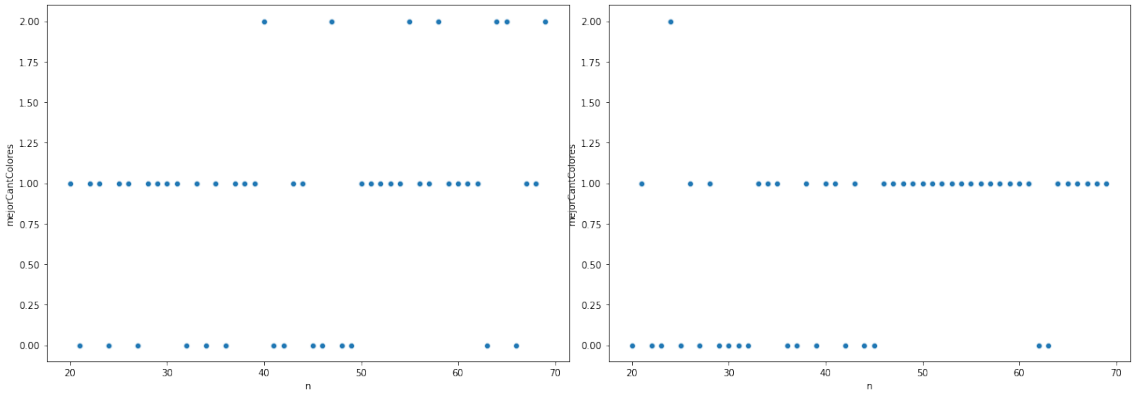


(a) (b)

Figura 19: Error relativo a m'_H en función de n variando la cantidad de colores usados para instancias con a) G ralo b) G denso

- Heurística golosa para la solución inicial: 2
- Máximo número de iteraciones: n^2
- Máximo número de iteraciones sin mejora: no se tiene en cuenta
- Tamaño de la memoria: 50 %, fijado en el experimento anterior
- Tipo de vecindad: vecindad por cambio de color guiada
- Vecindad size: sin efecto

y se hizo variar el parámetro `max_cant_colores` tomando los valores 0, 1, y 2 (ver Sección 4.2.4). En la Sección 4.2.5 se hipotetizó que para los valores extremos 0 y 2 -es decir, no agregar ningún color a los usados por la heurística golosa, o permitir el uso de n colores- el desempeño sería peor que para una cantidad de colores intermedia. En la Figura 19 se puede apreciar que para instancias ralas hay una ligera tendencia a obtener errores menores para una cantidad de colores intermedia (observando el percentil 50). En las instancias de G denso, se observa que para instancias pequeñas, hay una ligera ventaja para el valor 2, mientras que para instancias grandes hay ventaja del valor 1.



(a) (b)

Figura 20: max_cant_colores que brindó mayor impacto para cada n. a) Instancias con G ralo b)Instancias con G denso

En la Figura 20 se puede observar para cada n, cuál fue la configuración que llevó a obtener un mayor impacto. Para instancias ralas, se observa que efectivamente colocar el parámetro en 1 dio mejor impacto una mayor cantidad de veces. Para instancias densas se observa también la misma prevalencia, aunque para instancias pequeñas el valor del parámetro que da más impacto es 0, como se observó antes.

5.3.7. Experimento 3: Tipo de vecindad

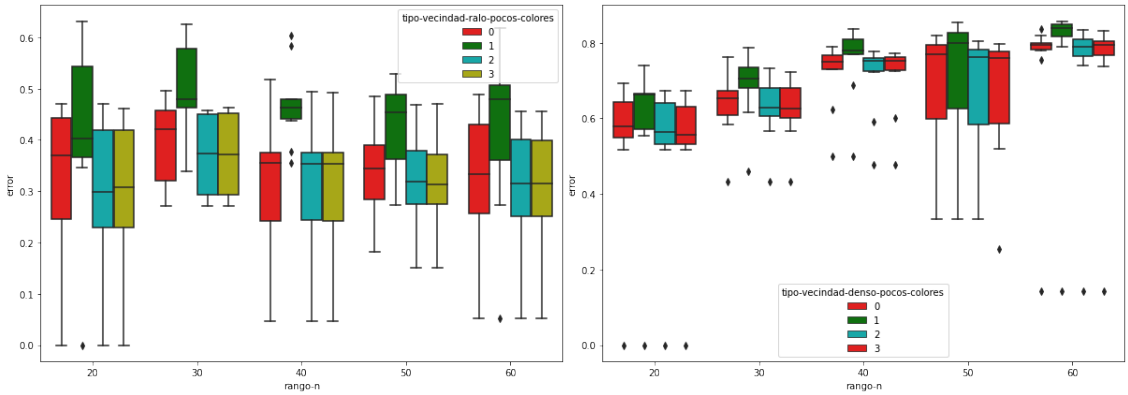
El objetivo de este experimento es comprobar o refutar la hipótesis planteada en la Sección 4.2.1, que afirma que si no se permite emplear demasiados colores, entonces efectuar swaps tiene el potencial de mejorar la calidad de las soluciones. Para este experimento, se fijaron los parámetros a los valores siguientes:

- Heurística golosa para la solución inicial: 2
- Máximo número de iteraciones: n^2
- Máximo número de iteraciones sin mejora: no se tiene en cuenta
- Tamaño de la memoria: 50 %, fijado en el experimento anterior
- Máxima cantidad de colores: 0 (sin agregar colores a la solución de la heurística golosa)
- Vecindad size: sin efecto

y se varió el parámetro de tipo de vecindad. Los resultados pueden observarse en la Figura 21.

Se aprecia que para el caso `tipo_vecindad` igual a 1 (swaps cada dos iteraciones), los resultados obtenidos son siempre de peor calidad, mientras que para `tipo_vecindad` igual a 2 y 3 (swaps cada 4 y 6 iteraciones respectivamente), los resultados tienden a ser mejores que para el caso igual a 0 (sin swaps).

Es decir, hacer swaps con una frecuencia demasiado elevada provoca una sustancial disminución de la performance, mientras que hacer swaps a intervalos más espaciados trae cierto beneficio. El primero de los efectos puede deberse a que los swaps resultan beneficiosos para ciertas circunstancias, pero el movimiento que permiten en el espacio de soluciones es más limitado que el de cambio de color (si solo se realizaran swaps, el multiconjunto de colores empleados siempre es el mismo); por lo tanto, realizarlo con demasiada frecuencia puede ser perjudicial. Otro posible factor es que para los swaps no se empleó la memoria de estructura (porque originalmente el swap se había pensado para hacerse con menos frecuencia). Un posible experimento adicional a realizar sería incorporar a la vecindad por swap el uso de la memoria.



(a) (b)

Figura 21: Error relativo a m'_H en función de n variando el tipo de vecindad para instancias con a) G ralo b) G denso

5.3.8. Experimento 4: Vecindad Size

En este experimento se busca determinar si la modificación del tamaño de la vecindad para la Metaheurística 2 introduce o no mejoras significativas. Se hipotetizó que para vecindades de tamaños mayores, debía obtenerse una mejor performance, aunque posiblemente para vecindades excesivamente grandes, el desempeño fuera peor, dado que se asemejaría más a una búsqueda local y perdería las ventajas de taboo search.

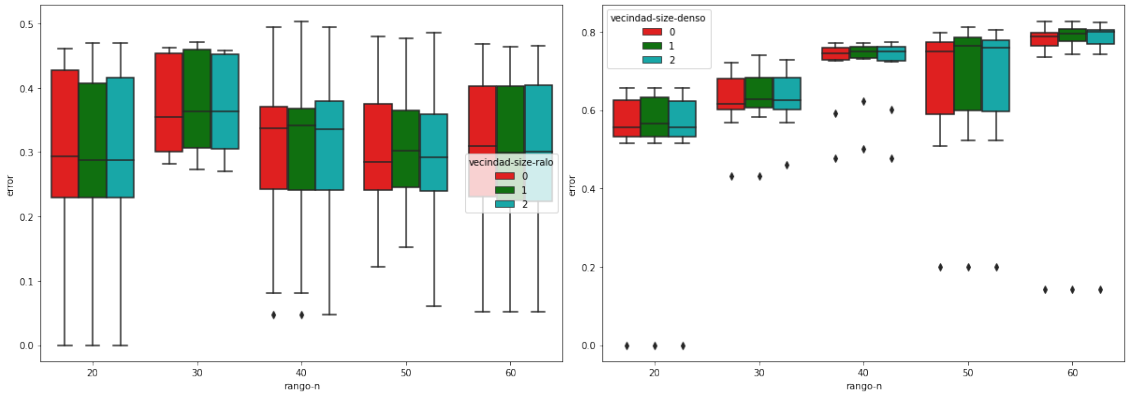
Los parámetros del experimento se fijaron en:

- Heurística golosa para la solución inicial: 2
- Máximo número de iteraciones: n^2
- Máximo número de iteraciones sin mejora: no se tiene en cuenta
- Tamaño de la memoria: 50 %
- Máxima cantidad de colores: 1 (es decir, maxColorHeuristica + $\sqrt{n - \text{maxColorHeuristica}}$ colores)
- Tipo de vecindad: 2 (es decir, en caso de Vecindad por Cambio de Color Guiado, se realizan swaps cada cuatro iteraciones).

Los resultados se exhiben en las Figuras 22 (donde se usaron los valores 0, 1 y 2 para `vecindad_size`, es decir, Vecindad Por Cambio de color guiada, Vecindad por cambio de Color Variable probando con 1 color por nodo, y Vecindad por cambio de color Variable probando con 2 colores por nodo)y 23 (donde se usaron además los valores -2 y -1, es decir, probando con $n/2$ y $\sqrt{n}$ colores respectivamente). (Aclaración: por cuestión de tiempos, el experimento con valores de `vecindad_size` iguales a -1 y -2, solo se corrió para el caso G ralo).

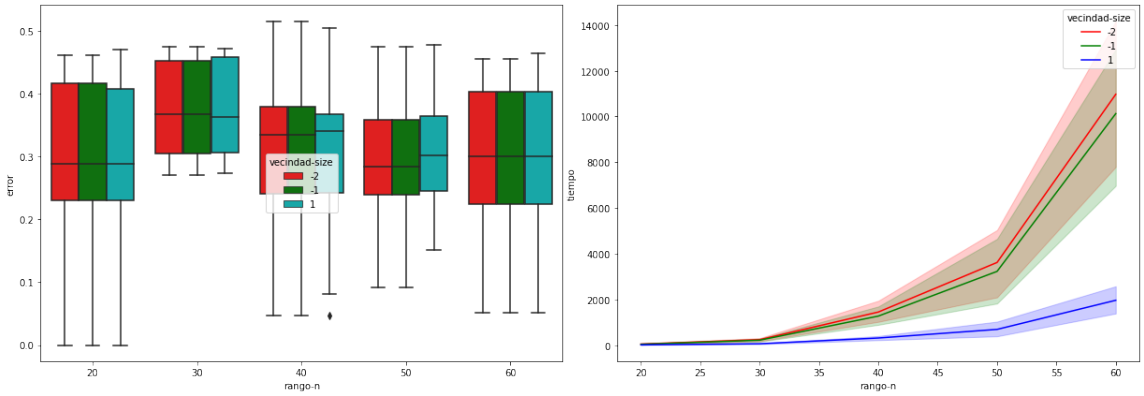
Se puede observar que contrariamente a lo supuesto, el aumento del tamaño de la vecindad no presenta mejora alguna. La explicación de este hecho es el modo en que se eligen los colores para intentar pintar cada nodo: es una elección determínistica, en la cual se escogen los menores colores que son viables. El efecto de esta forma de selección es que probar pintar con colores mayores a los usados inicialmente por la heurística golosa nunca tendrá un mejor impacto que los colores más chicos, y por lo tanto, no serán en general empleados.

Una posibilidad para conseguir cambios en la performance con modificaciones del tamaño de la vecindad sería incorporar a la selección de los colores candidatos una componente de azar. No fue implementado ni experimentado en el presente trabajo, pero sería interesante verificar si la hipótesis es válida.



(a) (b)

Figura 22: Error relativo a m'_H en función de n variando el tamaño de la vecindad 0,1 y 2 para instancias con a) G ralo b) G denso



(a) (b)

Figura 23: a) Error relativo a m'_H en función de n variando el tamaño de la vecindad para instancias con G ralo b) Tiempo de ejecución en ms

Debido a los resultados obtenidos, se escogió fijar el tipo de vecindad en Vecindad por Cambio de Color Guiada y Swaps, y no utilizar la Vecindad por Cambio de Color Variable, dejando el parámetro `vecindad_size` en 0.

En la Figura 23 se aprecia además que el aumento del tamaño de las vecindades trae aparejado un aumento en el tiempo de ejecución, que en la versión actual de los algoritmos, no se compensa con ninguna mejora.

5.3.9. Experimento 5: Impacto Vs Iteraciones

El objetivo de este experimento es observar el comportamiento de la metaheurística 2, para instancias específicas, en lo que respecta al impacto de las soluciones en cada iteración. Se seleccionaron las instancias siguientes:

- 60 Nodos, G ralo: $m_G = 12$, $m_H = 701$
- 68 Nodos, G denso:$m_G = 1204$, $m_H = 911$.

Para los experimentos se varió el tipo de Vecindad, tomando los valores 0, 1 y 2, y se fijaron los siguientes parámetros:

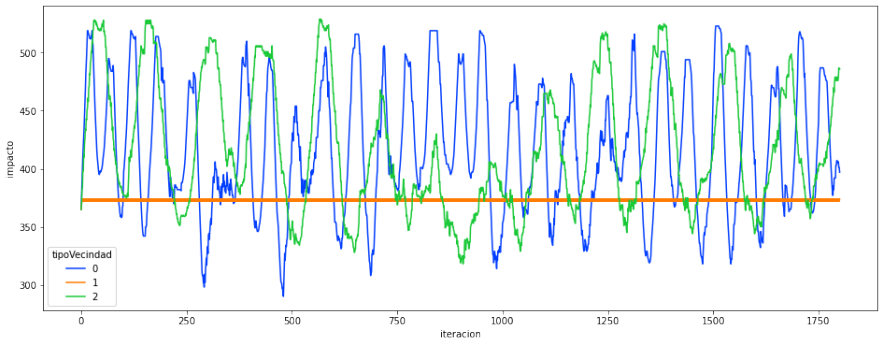


Figura 24: Impacto actual en cada iteración para Instancia 60 Nodos G ralo, max_cant_colores = 1

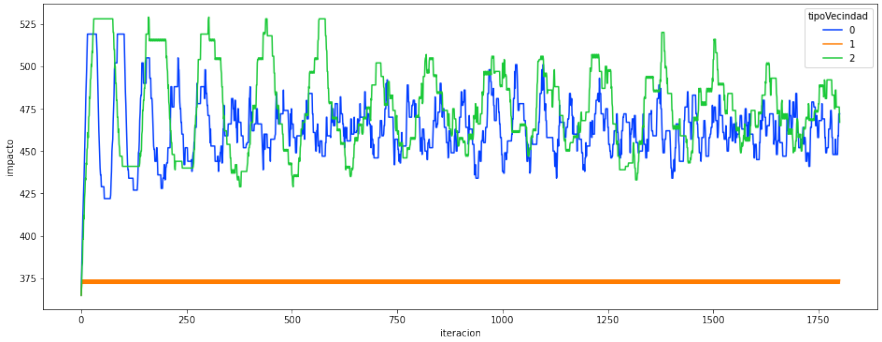


Figura 25: Impacto actual en cada iteración para Instancia 60 Nodos G ralo, max_cant_colores = 0

- Heurística golosa para la solución inicial: 2
- Máximo número de iteraciones: n^2
- Máximo número de iteraciones sin mejora: no se tiene en cuenta
- Tamaño de la memoria: 50 %
- Máxima cantidad de colores: 1 (es decir, maxColorHeuristica + $\sqrt{n - \text{maxColorHeuristica}}$ colores), excepto para el experimento de la Figura 25, en el cual se usa el valor 0 (no se agregan colores).
- Tipo de vecindad: 2 (es decir, en caso de Vecindad por Cambio de Color Guiado, se realizan swaps cada cuatro iteraciones).

Una primera observación común a las Figuras 24, 25 y 26 es que si el valor de `tipo_vecindad` es 1, es decir, cada dos iteraciones se utiliza swap, el impacto toma valores en un intervalo extremadamente pequeño y se mantiene casi constante a lo largo de toda la ejecución. Estos resultados se condicen con los descriptos en la Sección 5.3.7.

En la Figura 24 se puede apreciar que para los tipos de vecindad 0 y 2, se producen variaciones muy pronunciadas en los valores de impacto actuales, a diferencia de la Figura 26 en la que hay variaciones pero de menor amplitud. En la Figura 25 se modificó el parámetro de cantidad de colores: no se permite agregar colores a la solución hallada por la heurística golosa. Ahora las variaciones se tornaron mucho menores. La pregunta que surge entonces es por qué reducir la cantidad de colores a usar en esta instancia aumenta la estabilidad en la búsqueda de las soluciones (notar que más allá de la estabilidad, los impactos obtenidos en ambos casos no difieren

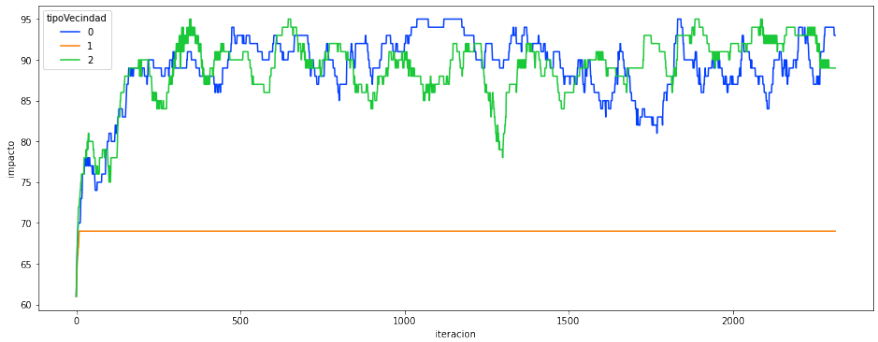


Figura 26: Impacto actual en cada iteración para Instancia 68 Nodos G denso

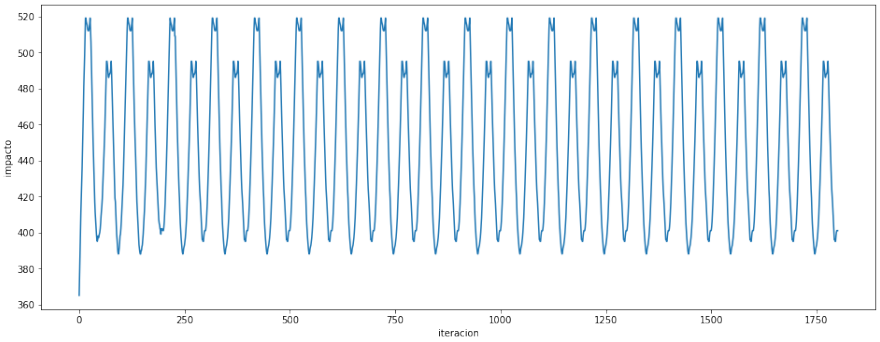


Figura 27: Impacto actual en cada iteración para Instancia 60 Nodos G ralo, vecindad_size igual a -1

demasiado). Este fenómeno tiene sentido: tener menor cantidad de colores disponibles reduce el espacio de búsqueda.

También surge la pregunta de cuánta estabilidad es deseable: demasiada estabilidad puede impedir que se hallen soluciones mejores, pero demasiado variación hace que la búsqueda sea demasiado caótica. Además, estos resultados sugieren que no existe necesariamente una única combinación de parámetros que sea la mejor para todas las instancias, y que dependa de las características de estas.

En las Figuras 27 y 28 se modificó `vecindad_size` a -1, con lo que se eliminó el factor de azar, y se pasó a la elección determinística de los colores. Lo que se observa va un pasó más allá que las conclusiones obtenidas en la Sección 5.3.8: no solo no se obtienen mejores soluciones por los motivos ya explicados, sino que el comportamiento es ¡aproximadamente periódico! Esto impide salir de una sucesión de valores específica, y explorar en forma más exhaustiva el espacio de soluciones; y resulta indeseable. La única ventaja que podría traer aparejado es que si se pudiera calcular el período de oscilación, se podría limitar el número de iteraciones y se sabría con certeza que la metaheurística con esa configuración específica nunca encontrará soluciones mejores. (Notar igualmente que a pesar de tener carácter oscilatorio, la calidad de la solución hallada es similar a la obtenida con los otros métodos). En la Figura 28 además se redujo el tamaño de la memoria al 10 % de n, y se observa cómo tiene un efecto en la amplitud de las oscilaciones. Esto se puede deber a que al haber menos nodos taboo, se puede volver más rápidamente a aquellos nodos que lograron un mejor impacto, y las variaciones son entonces menores.[1]

En el caso de la Figura 25 se observa además que el uso de swaps proveyó mejores soluciones.

[1]Este comportamiento trae cierta reminiscencia a las aplicaciones logísticas de modelado de poblaciones, ver `https://en.wikipedia.org/wiki/Logistic_map`. ¿Habrá relación? Bonus track: se adjunta un Gif en la carpeta Caos, que muestra la evolución del comportamiento variando el tamaño de la memoria. ¡A partir de cierto valor pareciera que el orden deja lugar al caos!

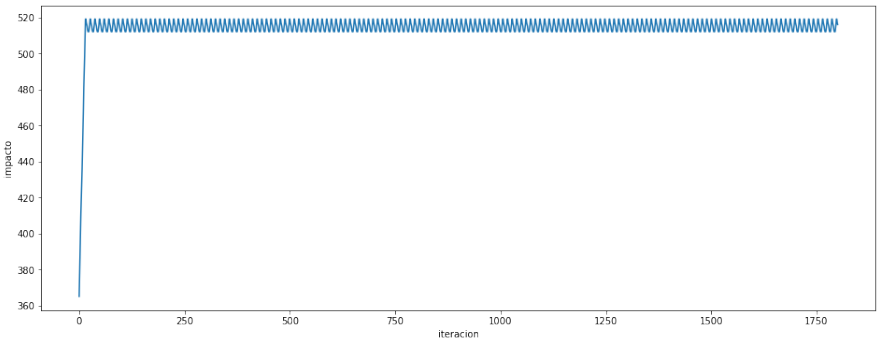


Figura 28: Impacto actual en cada iteración para Instancia 60 Nodos G ralo, vecindad_size igual a -1 y memory_size en 10 %

5.3.10. Carácter oscilatorio y Máximo número de iteraciones sin mejora

No se realizaron experimentos modificando el parámetro de maximo número de iteraciones sin mejora. Sin embargo, es pertinente hacer algunos comentarios al respecto en base a los resultados de los experimentos previos: El comportamiento de la metaheurística, al menos para las instancias y configuraciones de la Sección previa, es oscilatorio. Es posible que los valores de los picos correspondan a máximos locales (aunque también está la posibilidad de que el algoritmo, al estudiar solo una fraccion de la vecindad, no detecte una solución vecina que sea mejor). Que no se estabilice en un entorno acotado de un valor puede deberse a las características del algoritmo, y que no sea capaz de encontrar mejoras sustanciales más allá de ciertos máximos locales; o que efectivamente esos máximos locales estén muy cercanos a un máximo global. Esto no puede ser verificado con estas instancias dado que su impacto óptimo es desconocido.

El carácter oscilatorio sugiriría que más que medir cantidad de iteraciones sin mejora, podría ser más adecuado medir cantidad de picos que se alcanzaron a lo largo de la ejecución; y decidir qué valor sería adecuado para detener el algoritmo.

5.3.11. Máximo número de iteraciones

Se fijaron los valores de los parámetros a:

- Heurística golosa para la solución inicial: 2
- Máximo número de iteraciones sin mejora: no se tiene en cuenta
- Tamaño de la memoria: 50 %, fijado en experimento anterior
- Tipo de vecindad: vecindad por cambio de color guiada + swap cada 4 iteraciones
- Vecindad size: sin efecto

Se hipotetizó que debería haber mejora entre ejecutar el algoritmo n veces y n^2 iteraciones, cosa que efectivamente sucedió. También, que la mejora traería aparejado un incremento de la complejidad temporal. Los resultados se observan en las Figuras 29 y 30.

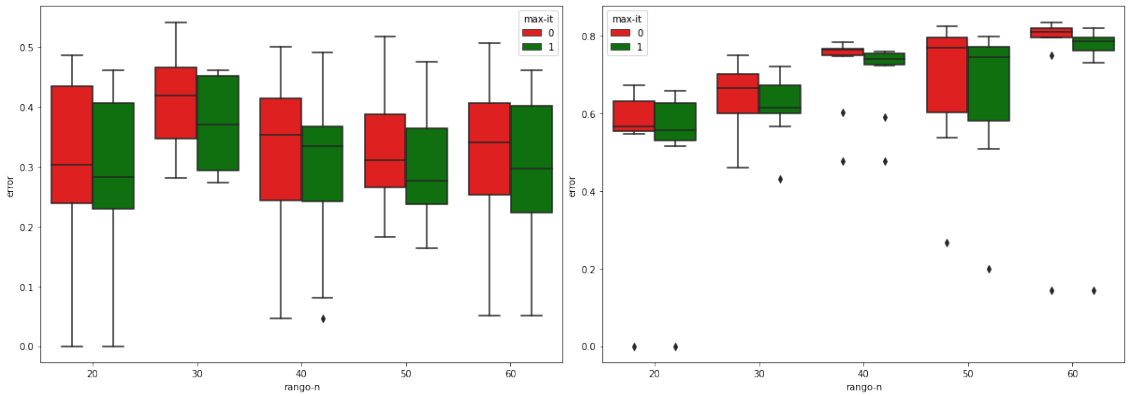
Dependiendo de la aplicación, si se necesita priorizar el tiempo de ejecución, o la calidad de la solución, convendrá una opción o la otra, o incluso, algún punto intermedio entre ambas.

Debido a las limitaciones temporales, no se realizaron ejecuciones con n^3 iteraciones.

5.4. Comparación entre metaheurísticas y heurísticas golosas

Este experimento se realizó empleando las instancias de óptimo conocido.

Los parámetros de la metaheurística 1 se fijaron a:



(a) (b)

Figura 29: Ejecución con n y n^2 iteraciones, a) Error relativo a m'_H en función de n para instancias con G ralo b) Error para instancias con G denso

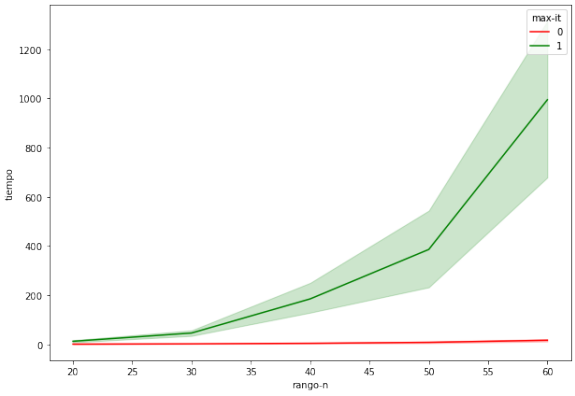
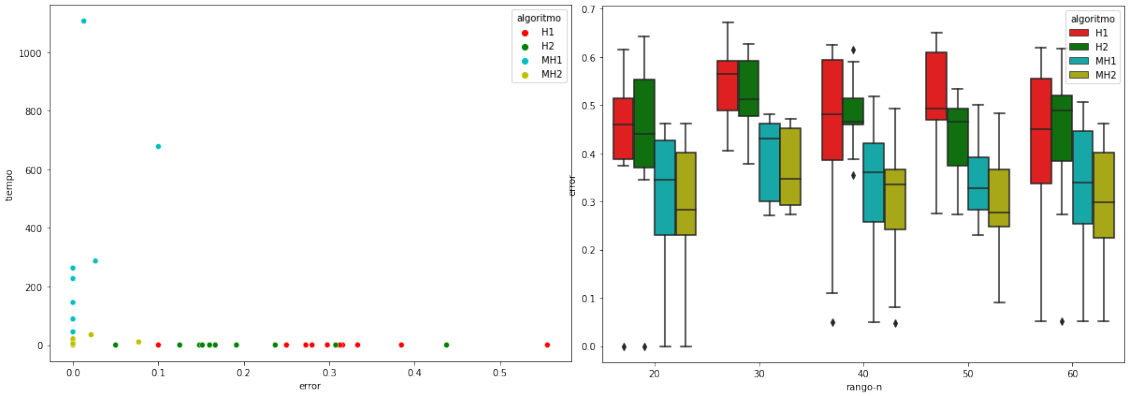


Figura 30: Tiempo de ejecución para n y n^2 iteraciones, G ralo



(a) Error relativo al impacto óptimo y Tiempo de ejecución para Heurísticas golosas 1 y 2, y Metaheurísticas 1 y 2

(b) Error relativo a m'_H con instancias generadas ralas, para Heurísticas golosas 1 y 2, y Metaheurísticas 1 y 2

Figura 31: Comparación Heurísticas y Metaheurísticas

- Heurística golosa para la solución inicial: 2
- Máximo número de iteraciones: n^2.
- Máximo número de iteraciones sin mejora: 2
- Aspiración: 1
- NumFactor: 0

Los parámetros de la metaheurística 2 se fijaron en base a los experimentos previos:

- Heurística golosa para la solución inicial: 2
- Máximo número de iteraciones: n^2.
- Máximo número de iteraciones sin mejora: no se tiene en cuenta
- Tamaño de la memoria: 50 %, fijado en experimento anterior
- Tipo de vecindad: vecindad por cambio de color guiada + swap cada 4 iteraciones
- Vecindad size: sin efecto

Los resultados de la ejecución para las instancias dadas se observan en la Figura 31a. Como se puede apreciar, tanto la Metaheurística 1 como la 2 presentan errores mucho más bajos que aquellos dados por las heurísticas golosas, observándose a su vez un incremento considerable en el tiempo de ejecución.

En la Figura 31b se observan los resultados para las instancias con G ralo generadas. En general, se observa que ambas metaheurísticas tienen mejor performance que ambas heurísticas golosas. Además, la heurística golosa 2 parece tener en general un ligero mejor desempeño que la 1; y lo mismo se observa para ambas metaheurísticas.

Las diferencias entre ambas metaheurísticas pueden deberse a dos factores: o bien al tipo de memoria escogido, o bien al tipo de vecindad empleado. Con los experimentos realizados no es posible determinar con certeza a qué se debe exactamente: si al uso de memoria de estructura en contraposición a memoria de soluciones; o a la forma de generar vecindades (además, las metaheurísticas no solo generan las vecindades de formas diferentes, sino que estas son de tamaños muy distintos: para el caso de la metaheurística 2 son mucho más pequeñas). Para determinar con certeza a qué se deben las ventajas, se debería experimentar el funcionamiento de cada metaheurística cambiando su memoria.

6. Conclusiones

En el presente trabajo se desarrollaron algoritmos para la resolución del problema de coloreo de máximo impacto, que es NP-hard. Por lo tanto, no se esperaba obtener soluciones exactas polinomiales, y sí hallar algoritmos que fueran capaces de encontrar soluciones de calidad en un tiempo razonable. Los algoritmos desarrollados entran dentro de dos categorías: heurísticas golosas, cuya dificultad de implementación fue menor; y metaheurísticas, en particular, taboo search, que implicaron una dificultad extra de diseño, pero fueron capaces de mejorar las soluciones obtenidas por las heurísticas golosas. Quedaron puntos que explorar para llegar a una comprensión más exhaustiva de los algoritmos. Algunos posibles desarrollos y experimentaciones futuros serían:

- A lo largo del trabajo se observó que es posible que no exista una única combinación de parámetros que sea la mejor para todas las instancias, y que dependa de las características de estas. Una posible línea de experimentación sería intentar hallar relaciones más precisas entre características de las instancias y combinaciones buenas de parámetros.
- Sería deseable también realizar experimentos adicionales variando en las metaheurísticas 1 y 2 el tipo de memoria empleado, para determinar si las ventajas detectadas se deben al tipo de vecindad empleado o al tipo de memoria, o a ambos factores.
- Quizás es posible encontrar mejores cotas para el impacto, de manera de poder comparar los algoritmos en forma más precisa. Otra posibilidad sería, usando los algoritmos ya desarrollados, tomar como valor de referencia los óptimos obtenidos por estos, para posibles desarrollos a futuro en que se intente mejorar las metaheurísticas.