DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

PCMI

Algoritmos y Estructura de Datos III
Segundo Cuatrimestre de 2020

Grupo 15

Integrante	LU	Correo electrónico
Cerdeira, Elías Nahuel	692/12	eliascerdeira@gmail.com
Gianatiempo, Octavio	280/10	ogianatiempo@gmail.com
Panichelli, Manuel	72/18	panicmanu@gmail.com
Schuster, Martín	208/18	m.a.schuster98@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Introducción

En la vida cotidiana es común tener situaciones donde se deseen asignar, disponer o distribuir elementos según algunas restricciones y/o preferencias. Algunos problemas complejos pueden resolverse con algoritmos de optimización combinatoria. Generalmente, estos algoritmos resuelven instancias de problemas que se consideran difíciles de resolver, explorando el espacio de soluciones (usualmente grande) utilizando algunas heurísticas. Estos algoritmos, además, lo logran reduciendo el tamaño efectivo del espacio ya que exploran el espacio de búsqueda eficientemente.

Un problema conocido de optimización combinatoria es el problema de *coloreo mínimo*, donde se explora el espacio de soluciones válidas (coloreos) y se selecciona aquel que utilice el menor número de colores. En este trabajo, se estudiará un problema alternativo, en el que se desea encontrar un coloreo factible para el grafo G buscando maximizar un función I (impacto) de acuerdo a las relaciones establecidas por otro grafo H que presenta los mismos vértices pero no necesariamente la misma distribución de aristas.

Formalmente, dados dos grafos $G = (V, X_G)$ y $H = (V, X_H)$ definidos sobre el mismo conjunto de vértices V y dado un conjunto de C de colores, se define el impacto $I(c)$ sobre H de un coloreo $c : V \to C$ de G como el número de aristas $e = (i, j) \in X_H$ tales que $c(i) = c(j)$. Se define entonces el Problema de Coloreo Máximo Impacto (PCMI) como un problema de optimización combinatoria donde se busca maximizar el valor de impacto ($I(c)$) obtenido. Para simplificar, se considera *solución factible* a todo coloreo válido de G. Cada coloreo se representa como un arreglo de colores de longitud n donde la posición i-ésima corresponde al color del vértice i.

A continuación se exhibe un ejemplo con su correspondiente respuesta esperada.

- $n = 5$ con los grafos G y H representados en la fig. 1. Las soluciones factibles son todos los coloreos válidos de G, como pueden ser [*rojo, verde, amarillo, azul, violeta*] o [*rojo, rojo, azul, azul, rojo*]. El primero corresponde a un coloreo trivial, asignar colores distintos a cada vértice, y presenta un impacto de 0 debido a que ningún par de vértices comparte colores. El segundo, es el coloreo óptimo ya que todas las aristas de H que no están en G, es decir todas las que podrían contribuir al impacto sin invalidar el coloreo si unieran dos vértices del mismo color, efectivamente unen dos vértices del mismo color.

(a) Grafo G (b) Grafo H

Figura 1: Instancia de ejemplo para el problema de PCMI. El grafo G se utiliza para verificar que el coloreo sea válido, establece restricciones entre nodos, mientras que el grafo H establece las relaciones de impacto.

Algunas situaciones de la vida cotidiana donde se puede encontrar un problema de PCMI podría ser el caso de distribución de invitados en las distintas mesas en un casamiento, cumpleaños o demás fiestas. Cada invitado se representa con un nodo $v \in V$. Luego, como estamos modelando personas reales, existirán enemistades, peleas o cualquier tipo de restricción que impida o sugiera que no es correcto sentar a dos personas en la misma mesa. Estas restricciones se representarán como una arista en el grafo G. Luego, tendremos gente que se desee agrupar preferencialmente por algún criterio, como pueden ser parejas, amigos, familiares directos o grupos etarios. Estas relaciones deseables, pero no restrictivas, se modelan como una arista en el grafo H. Luego, las distintas mesas (que pueden considerarse de tamaño infinito para simplificar el problema) se pueden representar a través de los colores. Así, es fácil ver que se

puede utilizar una solución al problema de PCMI para encontrar una asignación óptima para las personas en las distintas mesas, dado que el coloreo obtenido permite evitar juntar gente que no quiere reunirse y maximiza la cantidad de personas que se quieren poner juntas.

El objetivo de este trabajo es implementar una solución para PCMI utilizando 3 heurísticas golosas constructivas y 2 metaheurísticas y evaluar la efectividad de cada una para distintas instancias. En primer lugar se utiliza el algoritmo de *heurística secuencial de coloreo con orden largest first* (S-LF) que consiste en colorear secuencialmente los vértices del grafo G comenzando por alguno de grado mayor ($\triangle_G(v)$) y asignando el menor número de color posible. Para el algoritmo de *Wyrnística diferencial* (W) se genera un conjunto de aristas W tales que $e \in W \iff e \in X_H \wedge e \notin X_G$. Luego, para cada arista en W, se colorean sus extremos del mismo color. En caso de que uno de los dos esté ya coloreado, ignora ambos vértices y, al terminar, asigna un color distinto a cada vértice no coloreado. Finalmente, la heurística *Wyrnística power* (WP) consiste en realizar el mismo procedimiento anteriormente descrito, excepto que al intentar colorear el par $(v, w) \in W$, si w ya está coloreado, en vez de saltearlo intenta colorear a v del mismo color, siempre y cuando sea válido.

Para el caso de las metaheurísticas, se utilizan para construir una solución inicial todas las heurísticas golosas descritas anteriormente. Por un lado, para el algoritmo *Tabu Search con Swap* (TSS) se explora la vecindad de una solución intermedia intercambiando el color de dos vértices del grafo, mientras que para el algoritmo *Tabu Search con Change* (TSC), se cambia el color de un vértice por algún color válido presente en el coloreo observado. En ambos casos, se utiliza una memoria o *lista tabú* para almacenar soluciones intermedias ya recorridas. En un caso, se almacena de forma *estructural* (E), definiendo qué vértices se intercambiaron o qué color se le asignó al vértice cambiado. En otro, se almacena directamente el *coloreo* (C) completo de la solución revisada.

En la sección de metodología se introducen y explican los algoritmos y las diferencias entre las distintas técnicas junto con las respectivas demostraciones de correctitud y complejidad. Luego, se exponen los experimentos realizados con sus resultados y la respectiva discusión. Por último, se detallan las conclusiones finales del trabajo.

2. Metodología

Dado que PCMI pertenece a la categoría de problemas $\mathcal{NP}$-hard, en este trabajo se busca dar soluciones de la mejor calidad posible, pero no las óptimas. A continuación se presentan los distintos métodos utilizados para desarrollar algoritmos que lo resuelvan.

3. Heurísticas constructivas golosas

Las **heurísticas constructivas** son métodos que construyen iterativamente una solución factible para una instancia de un problema dado. Se suelen utilizar procedimientos golosos, que si bien no garantizan que las soluciones sean óptimas, retornan soluciones de una calidad aceptable por un costo computacional comparativamente menor. En este trabajo se implementa una heurística existente para el problema de coloreo *(Secuencial - Largest First)* y se definen dos nuevas específicas para PCMI (*Wyrnística Diferencial y Wyrna Power*).

3.1. Secuencial - Largest First (S-LF)

La heurística **secuencial** consiste en recorrer los vértices de forma secuencial y colorear a cada uno con el mínimo posible. La implementación **Secuencial - Largest First (LF)** los recorre en forma descendiente según su grado, lo cuál hace que genere coloreos más cercanos al óptimo. Es la heurística más natural para colorear los vértices de un grafo y su correctitud es trivial. Es fácil verificar que no genera siempre el coloreo óptimo, lo cual no resulta importante para PCMI dado que sólo interesa optimizar el impacto. En la fig. 2 se presenta el coloreo obtenido con esta heurística.

El Algoritmo 1 muestra una implementación posible. Es importante notar que es un algoritmo que solo toma en cuenta el grafo G y no considera el impacto que genera. Sin embargo, resulta intuitivo que a menor cantidad de colores, mayor es la probabilidad de que haya un impacto alto.

Algorithm 1 Algoritmo de *Secuencial con ordenamiento Largest First* para PCMI.

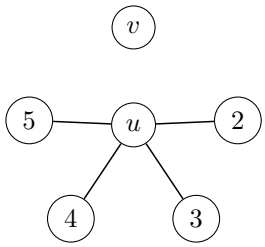
1: **Entrada:** los grafos G y H en representación de listas de adyacencias
2: **Salida:** coloreo de G e impacto
3:
4: **function** $secuencialLF(G, H)$:
5: $vertices \leftarrow \forall v \in G$ pares de $(d_G(v), v)$ ▷ $O(n)$
6: $sort(vertices)$ ▷ $O(n \times log(n))$
7: $coloreo \leftarrow array_n[undefined]$ ▷ $O(n)$
8: **for** (g, v) **in** $vertices$ **do**: ▷ $O(n^2)$
9: **for** $color\ c$ **in** $1..n \wedge coloreo[v]\ is\ undefined$ **do**:
10: **if** $\nexists\ u \in N_G(v) \mid coloreo[u] = c$ **then** ▷ $O(n)$
11: $coloreo[v] = c$
12: **return** $coloreo, impacto(H, coloreo)$ ▷ $O(m_H)$
13:
14: **Complejidad:** $O(n^2 + n \times log(n) + n + m_H) = O(n^2)$

(a) Grafo G (b) Grafo H

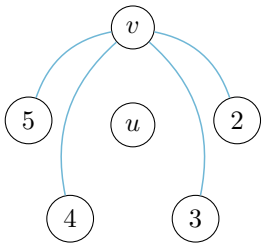
Figura 2: Instancia de ejemplo para el problema de PCMI. Se presenta el coloreo esperado obtenido al correr el algoritmo presentado para la heurística S-LF.

Dado que esta heurística colorea a cada vértice con el mínimo que pueda asignarle sin tener en cuenta H, dados 3 vértices, u, v, w tales que $(v, w) \in X_H, (u, v) \in X_G$ y $d_G(u) > d_G(v) > d_G(w)$, se colorea primero a u, luego a v de un color diferente a u y finalmente a w del mismo color de u. Como resultado, u tiene un color diferente al de v, reduciendo así el impacto.

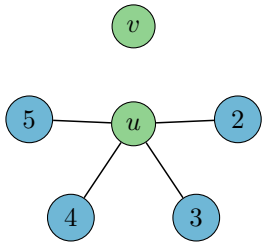
Esto puede extenderse a clases de instancias, donde dados u y v, u es universal en $G - v$ (pero aislado en H) y v universal en $H - u$ (pero aislado en G). u no es adyacente a v. Se colorea primero a v por ser el de grado máximo en G del color 1, luego a todos menos v del color 2, y finalmente a v del color 1, llevando a un impacto 0 sin importar que tan grande sea el grafo, cuando en realidad el impacto máximo es $n - 1$.



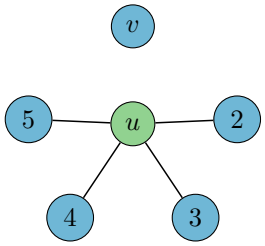
(a) Grafo G



(b) Grafo H



(c) Grafo G coloreado según S-LF. Impacto 0



(d) Grafo G coloreado de forma óptima. Impacto 4

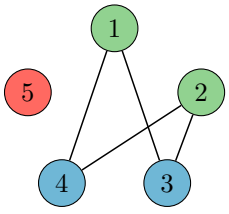
Figura 3: Ejemplo de instancia que funciona mal para S-LF.

Como se ve en la fig. 3 el algoritmo primero colorea u del color 1, luego todos los adyacentes a el del color 2, y finalmente a v del color 1 nuevamente, llevando a un impacto de 0.

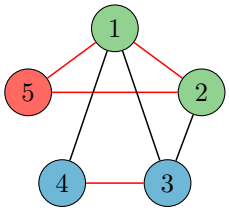
3.2. Wyrnística Diferencial (W)

El algoritmo de la *Wyrnística diferencial* (W) consiste en dados X_G y X_H conjunto de aristas, generar un conjunto $W = \{e \mid e \in X_H \wedge e \notin X_G\}$ y para cada $(u, v) \in W$ intentar pintar los extremos del mismo color. Así, si ninguno de los dos vértices está coloreado, se pone en ambos vértices el color de aquel con menor numeración. En caso de que alguno de los dos tenga color definido, para evitar tener que chequear si es posible asignarles el mismo color, se ignoran. Finalmente, para todo vértice que no tenga color definido, se le asigna un color distinto de forma incremental. De esta manera, este algoritmo no permite tener más de dos vértices con el mismo color.

Por ejemplo, en el caso presentado en la fig. 4 se puede observar una instancia de ejemplo coloreado según el algoritmo 2 descrito a continuación. Al recorrer los nodos adyacentes al nodo 1 en W se encuentran 2 y 5. En la primera iteración observa al nodo 2, como ninguno se encuentra coloreado aún, le asigna el primer color. Luego, en la siguiente iteración se verifica que v fue coloreado, por lo que se cancela el ciclo y se pasa a observar el siguiente vértice (2). Al finalizar, como 5 no fue coloreado, se le asigna un color arbitrario no utilizado aún.



(a) Grafo G



(b) Grafo H

Figura 4: Instancia de ejemplo para el problema de PCMI. Se presenta el coloreo esperado obtenido al correr el algoritmo presentado para la heurística W. El conjunto W está pintado de rojo sobre H.

Algorithm 2 Algoritmo de *Wyrnística diferencial* para PCMI.

```
1: Entrada: los grafos G y H en representación de listas de adyacencias
2: Salida: coloreo de G e impacto
3:
4: function WyrnisticaDiferencial(G, H):
5:     coloreo ← array_n[undefined]                                  ▷ O(n)
6:     matrizG ← listaAMatrizDeAdyacencia(G)                         ▷ O(m_G)
7:     for vertice v in H do:                                        ▷ O(m_H)
8:         for w ∈ H[v] ∧ coloreo[v] = undefined do:
9:             if matrizG[v][w] then continue
10:            if coloreo[w] ≠ undefined then continue
11:
12:            coloreo[w] ← coloreo[v] ← min(v, w)                   ▷ O(1)
13:    for v ∈ G do:                                                 ▷ O(n)
14:        if coloreo[v] = undefined then
15:            coloreo[v] = n + v          ▷ para asegurarse que sean distintos
16:    return coloreo, impacto(H, coloreo)                           ▷ O(m_H)
17:
18: Complejidad: O(m_G + m_H + n)
```

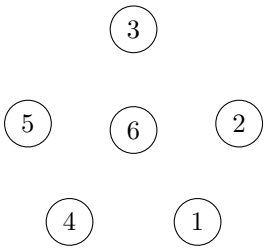
Se desprende del algoritmo 2 que cada vértice se recorre de tres maneras posibles.

- Al comenzar la iteración v no está coloreado. Presenta un adyacente en H no coloreado. Ambos se colorean del mismo color. Se deja de recorrer la vecindad de v.
- Al comenzar la iteración v no está coloreado. Se observa que v tiene todos sus vecinos en H coloreadas. Al finalizar el recorrido de H se colorea v con un color exclusivo.
- Se observa que v no presenta vecinos en H. Al finalizar el recorrido de H se colorea v con un color exclusivo.

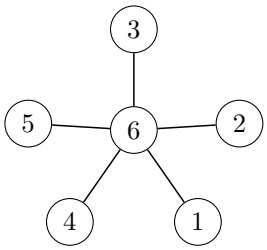
La correctitud del algoritmo depende de que ambos valores devueltos sean correctos.

- **Coloreo válido:** el coloreo obtenido es válido, pues sólo se pintan del mismo color aquellos nodos que presenten arista en el grafo H y no en G, de esta manera, cualquier restricción presente en G definirá que esos vértices tengan colores distintos.
- **Impacto válido:** el impacto se calcula antes de finalizar el algoritmo. La función $impacto(H, coloreo)$ recorre las aristas de H y verifica cuántas tienen sus extremos del mismo color. Por lo tanto, el impacto retornado también es válido.

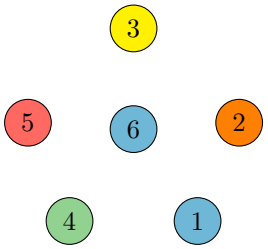
Se considera el caso de una arista $e = (u, v)$ cuyos extremos presentan el mismo color en el óptimo coloreo para PCMI. Sin embargo, si en toda iteración del algoritmo descrito para W al menos uno de ellos ya se encuentra coloreado, no se les asignará nunca el mismo color. Esto sucederá para todas las instancias en las que G sea $n \times K_1$ (grafo de todos nodos aislados) y H un grafo *estrella* $K_{1,n}$, en los cuales el óptimo es pintar a todos del mismo color pero solamente pintaremos a dos.



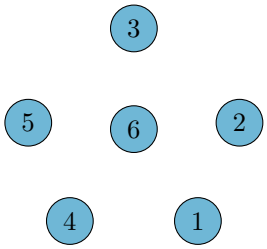
(a) Grafo G, todos nodos aislados



(b) Grafo estrella $H = K_{1,5}$



(c) Grafo G coloreado según W. Impacto 1



(d) Grafo G coloreado de forma óptima. Impacto 5

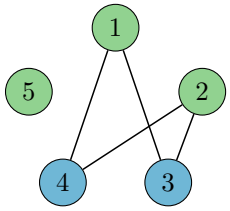
Figura 5: Ejemplo de instancia que funciona mal para W.

El coloreo óptimo sería simplemente colorear a todos lo vértices del mismo color. Sin embargo, el algoritmo colorea a 6 y 1 del mismo color, y para el resto de los vértices no puede ya que 6 está coloreado.

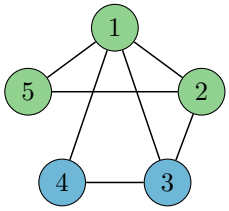
3.3. Wyrnística Power (WP)

La técnica de *Wyrnística power* (WP) constituye una mejora por sobre la técnica W. En este caso también se construye el conjunto de aristas W ya descrito y se recorren los vértices de acuerdo a los adyacentes encontrados en W. Dada una arista $e = (u, v)$ si ninguno de los dos vértices tiene color, se pone a ambos vértices el color de aquel con menor numeración. Al recorrer los vecinos de u, es decir, u aún no está coloreado. En ese caso, si v vecino de u está coloreado, verifica que u pueda tener ese color revisando todos sus adyacentes. Si ningún vecino de u tiene ese color, le asigna el color de v a u. Caso contrario, se ignora la iteración y continúa.

En fig. 7 se observa un ejemplo de coloreo obtenido luego de una iteración de WP. En la primera iteración se colorean 1 y 2. Al recorrerse los vecinos de 5 se encuentra con que 1 está coloreado, verifica en los vecinos de 5 en G y como no tiene vecinos, puede colorearlo del mismo color que 1. Al finalizar, 1, 2 y 5 quedan del mismo color.



(a) Grafo G



(b) Grafo H

Figura 6: Instancia de ejemplo para el problema de PCMI. Se presenta el coloreo esperado obtenido al correr el algoritmo presentado para la heurística WP.

Para verificar la correctitud, es necesario comprobar que la línea modificada en el algoritmo 3 no genere soluciones inválidas.

- **Coloreo válido:** para asignar el color del adyacente en H ya coloreado a un nodo v se recorren todos los vecinos de v en G. En caso de encontrar uno que tenga ese color, no se asigna. Caso

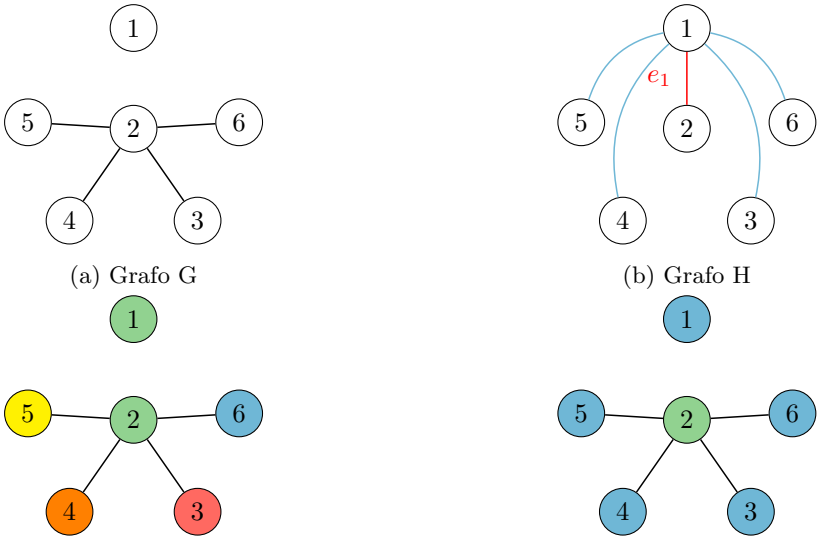
contrario, se pinta del mismo color que el adyacente en H. De esta manera, no puede resultar en un coloreo inválido, pues se verifican las relaciones en G.

- **Impacto válido:** como se calcula al final del código mantiene correctitud al igual que en W

Algorithm 3 Modificación al algoritmo de *Wyrnística diferencial* para PCMI en la línea 10. El algoritmo resultante es el que corresponde a *Wyrnística power*.

```
1: if coloreo[w] ≠ undefined then
2:     if ∄ u ∈ N_G(v) | coloreo[u] = coloreo[w] then                ▷ O(n)
3:         coloreo[v] = coloreo[w]
4:
5: Complejidad resultante: O(m_G + m_H × n + n)
```

Si bien es una mejora por sobre W, sigue teniendo clases de instancias en las que no da soluciones óptimas. La idea es que un color se bloquee siendo usado por un nodo universal.



(a) Grafo G (b) Grafo H

(c) Grafo G coloreado según WP. Impacto 1 (d) Grafo G coloreado de forma óptima. Impacto 4

Figura 7: Ejemplo de instancia que funciona mal para WP. En rojo la arista que primero se visita de H.

Visitando primero e_1, se pintan 1 y 2 del mismo color. Como 2 es universal en G, no se puede pintar a ninguno de sus vecinos del mismo color que 1, lo que llevaría al máximo impacto.

4. Búsqueda local y Metaheurísticas

Si bien las heurísticas constructivas golosas dan soluciones razonables, se pueden mejorar. Una forma es tomarlas de solución *base*, para luego iterativamente explorar soluciones *cercanas*, resultantes de cambiar levemente el coloreo, con las esperanzas de así acercarse más al óptimo.

Más formalmente, dado un problema específico y una instancia con un conjunto de soluciones factibles S, una **vecindad** es una función $N : S \to \mathcal{P}(S)$, donde $\mathcal{P}(S)$ es el conjunto de partes de S. Para $s \in S$ decimos que $N(s)$ es su vecindad. Luego, un **óptimo local** será la solución con máximo impacto de una vecindad dada.

Los algoritmos de **búsqueda local** exploran este espacio de vecindad partiendo desde una solución inicial, moviendose por óptimos locales, con esperanzas de así llegar a un óptimo global o terminar cerca de él. Pero tienen un problema fundamental: quedan estancados en óptimos locales siempre que tengan que empeorar temporalmente para llegar a un óptimo global

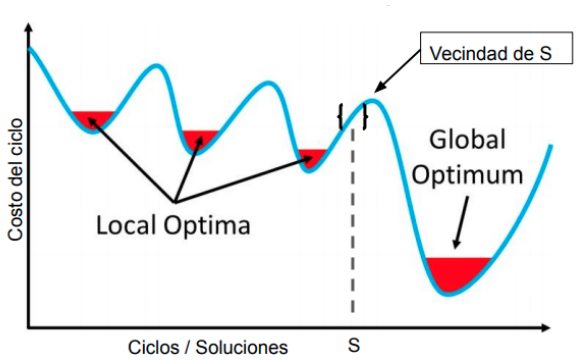


Figura 8: Esquema representativo del costo de cada ciclo en función de la relación ciclos sobre soluciones. Se puede observar la representación de óptimos locales (local optima), el óptimo global (global optimum) y las soluciones vecinas de S.

Para intentar vencer esa limitación se desarrollaron las **metaheurísticas**, cuyo objetivo es realizar una búsqueda más amplia del espacio de soluciones, para así evitar caer en óptimos locales. Estas no son específicas para un problema dado, sino que son técnicas más generales utilizadas para construir heurísticas de búsqueda particulares para cada uno. La metaheurística implementada en este trabajo es **Tabú Search**

4.1. Tabú Search

La base del método es parecida a búsqueda local, comienza con una solución inicial y se mueve por su vecindad, pero contempla que a veces se necesita pasar por soluciones peores para llegar a soluciones mejores. El problema con esto es que puede llevar a ciclos, entonces para evitarlo hace uso de una **lista tabú**, que recuerda las soluciones previamente visitadas donde fue y no las exploras nuevamente. El esquema general del método es el descrito en el algoritmo 4.

Algorithm 4 Esquema general de tabú search, en **negrita** cosas a definir. Las complejidades suponen accesos en tiempo constante a memoria

```
1:  Entrada: los grafos G y H en representación de lista de adyacencia.
2:  Salida: coloreo y su impacto
3:
4:  function TabuSearch(G, H, ...):
5:      actual ← solucionInicial(G, H)                          ▷ O(I)
6:      mejor ← actual
7:      T ← inicializarMemoria(...)                             ▷ O(T)
8:      while No se cumple el criterio de parada do             ▷ O(#it)
9:          actual ← max{ filtrarTabu(vecindad(actual), T) }    ▷ O(|V| + N(s))
10:         T ← T + actual                                      ▷ O(1)
11:         if impacto(actual) > impacto(mejor) then
12:             mejor ← actual
13:     return mejor, impacto(mejor)
```

La complejidad entonces será $O(I + T + \#it \times (|V| + N(s))$, donde

- **I** es el costo de obtener la solución inicial,
- **T** es el costo de inicializar la memoria,
- $|V|$ es el tamaño de la vecindad, max y $filtrarTabu$ son lineales en cuanto a ella,
- **N(s)** es el costo de encontrar la vecindad.

4.1.1. Parámetros

Como se puede ver en esquema general presentado en el Algoritmo 4, el método varía según la elección de **solución inicial**, **memoria**, **criterio de parada** y **vecindad**. Se describen las versiones que se contemplan de cada uno.

- **Solución inicial**: Se consideran como procedimientos para generar la solución inicial todas heurísticas constructivas golosas presentadas previamente: W, WP y S-LF.
- **Criterio de parada**: Se toma una cantidad máxima de iteraciones para el algoritmo.
- **Memoria**: Existen dos tipos. Por **solución** (i.e un coloreo) y **estructura**, en lugar de recordar las soluciones, se recuerdan los *cambios estructurales* que llevaron de una a otra. En el segundo caso, podría causar que más de una solución que se obtenga con el mismo cambio se marque como tabú y no se visite. Incluso las no exploradas que podrían llegar a ser mejores, dado que el coloreo resultante de aplicar un cambio estructural depende del *contexto* en el cuál se aplicó. El mismo cambio engloba a más de un coloreo. Para evitar perder buenas solucionas, se introduce la **función de aspiración**. Si una solución es tabú pero es mejor que lo visto hasta el momento, *se aspira* y se la considera como solución factible. Esto es contemplado en *filtrarTabu* del Algoritmo 4.

  Es importante notar que la función de aspiración no tiene sentido junto con la memoria por soluciones, ya que si el algoritmo ya pasó por esa solución, fue contemplada ella y su vecindad en su totalidad, y no tendría sentido volver a verla.

  Además, la memoria tiene un tamaño fijo, ya que guardar el espacio entero de soluciones sería demasiado costoso.
- **Vecindad**: Se definen dos movimientos o cambios estructurales para pasar de una solución a otra: **change** (cambiar un color de un vértice por uno ya usado en el coloreo) y **swap** (intercambiar los colores de dos vértices). Finalmente, la vecindad de una solución será el conjunto de variaciones que presenten un coloreo válido luego de realizar un solo cambio. Opcionalmente, se puede tomar una parte de la vecindad para restringir la búsqueda e influenciar la toma de decisiones subóptimas. La representación de la estructura utilizada para almacenar en la memoria es en tuplas $(v, c) \mid v \in V, c$ color para *change* y $(v, w) \mid v, w \in V$ para *swap*.

4.2. Vecindades

Los procedimientos para las vecindades a ser usados en las implementaciones concretas de Tabú Search son los siguientes:

Algorithm 5 Procedimientos para distintas vecindades

```
1:  function VecinosChange(G = (V, X_G), H, coloreo, ...)          ▷ O(n^2 + n × (m_G + m_H))
2:      vecinos ← {}
3:      for v ∈ V do                                               ▷ O(n^2 + n × (m_G + m_H))
4:          coloresAdy ← {coloreo[w] | (w, v) ∈ X_G} ∪ {coloreo[v]}   ▷ O(m_G)
5:          coloresFactibles ← coloreo \ coloresAdy                ▷ O(n)
6:          for c ∈ coloresFactibles do
7:              vecinos ← vecinos ∪ {(coloreo[v] ↞ c, impacto(coloreo) + Δ_I}   ▷ O(m_H)
8:      return vecinos
9:
10: function VecinosSwap(G = (V, X_G), H, coloreo, ...)            ▷ O(n^2 + n × (m_G + m_H))
11:     vecinos ← {}
12:     for v ∈ V do                                               ▷ O(n^2 + n × (m_G + m_H))
13:         for w ∈ V do
14:             swapValido ← ∄ u ∈ N(v), coloreo[u] = coloreo[w] ∧   ▷ O(m_G)
                             ∄ u ∈ N(w), coloreo[u] = coloreo[v]
15:             if swapValido then
16:                 vecinos ← vecinos ∪ {(swap(coloreo, v, w), impacto(coloreo) + Δ_I}   ▷ O(m_H)
17:     return vecinos
18:
```

Donde ↞ en la línea 7 de Change asigna el color al coloreo y devuelve el coloreo resultante. Δ_I es la diferencia de impacto resultante del cambio estructural. Para ambos casos, se calcula en $O(m_H)$ y está definido de la siguiente manera

$$\Delta_I = \#\{w \in N_H(v) \wedge c_w = c\} - \#\{w \in N_H(v) \wedge c_w = c_v\},$$

donde c_v es el color del vértice v, $N_H(v)$ son los vértices adyacentes a él en H y c el nuevo color del vértice v.

El tamaño de ambas vecindades $|V|$ es $O(n^2)$, ya que para cada vértice se puede cambiar el color por todos los usados hasta el momento (a lo sumo n) o intercambiar con el resto (también n).

Para ambas, opcionalmente se puede especificar un **porcentaje** a tomar de la vecindad, para lo cual se mezcla la vecindad de forma aleatoria y se toma la primera fracción cuyo tamaño alcanza dicho porcentaje. Esto puede ayudar a que tabú explore más y tome soluciones peores que los óptimos locales.

4.3. Variaciones

Finalmente, se presentan las variaciones de *tabú search* implementadas en este trabajo. Para la complejidad, se deja sin determinar aquello que depende de meta parámetros no definidos, cada uno de los cuales será optimizado durante la experimentación. Por lo tanto, la complejidad es la misma para todos: $O(I + T + \#it \times (|V| + N(s)) = O(I + |T| + \#it \times (n^2 + n \times (m_G + m_H))$

- **TSC-E**: Tabú Search con vecindades de *change* y memoria de tipo estructura.
- **TSC-C**: Tabú Search con vecindades de *change* y memoria de tipo solución (coloreo).
- **TSS-C**: Tabú Search con vecindades de *swap* y memoria de tipo solución (coloreo).
- **TSS-E**: Tabú Search con vecindades de *swap* y memoria de tipo estructura.

Cada clase de vecindad tiene una desventaja. Para **change**, como se cambia el color de un vértice por otro ya utilizado en el coloreo, en cada paso la cantidad de colores pasa a ser menor o igual. Esto

hace que el espacio de soluciones al que se puede llegar desde la vecindad de la solución actual se reduzca en cada iteración hasta que no haya más vecinos. Por lo tanto, ciertas soluciones muy cerca del óptimo no se podrían explorar mucho y el algoritmo terminaría cortando luego de pocas iteraciones. En cambio, resulta evidente que algoritmos que den soluciones razonables, pero no con la mínima cantidad de colores, pueden hacer que el algoritmo performe mejor. Por otro lado, en **swap** pasa lo contrario. Al intercambiar los colores de dos vértices, la cantidad de colores se mantiene luego de cada iteración, por lo que se cree que puede ser más conveniente tener como solución inicial un coloreo de pocos colores. Visto en términos de los algoritmos descritos, probablemente sea conveniente que comience con $S - LF$ o WP en vez de W, ya que dan soluciones con menos colores. Estas suposiciones se validarán experimentalmente.

5. Experimentación

5.1. Metodología experimental

El objetivo de la experimentación es lograr comparar el desempeño de los distintos algoritmos para diferentes parámetros, contrastando tiempo de ejecución con calidad de soluciones. Para ello, se define un conjunto de instancias de tamaños variados ($n = [6, 8..., 30]$) para las cuales se conoce el valor óptimo, separado en instancias de *train* (entrenamiento, $n \not\equiv 0 \pmod 4$) y *test* (verificación, $n \equiv 0 \pmod 4$). De esta forma, se buscan los mejores parámetros en las instancias de *train* y luego se verifica si efectivamente son los mejores comparándolos con instancias del conjunto de *test*.

5.1.1. Rangos para parámetros de Tabú Search

Para correr todas las combinaciones de las variaciones descritas en la sección 4.3 sobre las instancias de *train*, se definen los siguientes rangos para los parámetros de Tabú Search:

- % vecindad $\in \{5, 10, \ldots, 30, 40, \ldots, 100\}$
- $|T|$ (*tamaño de la memoria*) $\in \{5, 20, \ldots, 65, 100, 200, \ldots 600\}$
- $\#iteraciones \in \{100, 300, \ldots, 900\}$
- aspirar $\in \{si, no\}$.

5.1.2. Métricas

Para medir la eficiencia se usa el tiempo de ejecución, y para la calidad de las soluciones el *gap relativo* con respecto de la solución óptima, definido de la siguiente manera:

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

donde I calcula el impacto de una solución x y x^* es alguna óptima.

5.1.3. Especificaciones técnicas

Los algoritmos fueron implementados utilizando el lenguaje de programación $C++$, y fueron ejecutados en una computadora con un Intel(R) Core(TM) i5-4460 CPU @ 3.20GHz (32K cache lvl 1, 256K lvl 2 y 6MB lvl 3) y 16GB de RAM.

5.2. Heurísticas constructivas golosas

Se comenzó evaluando el desempeño de las distintas heurísticas constructivas golosas para distintas instancias del problema PCMI. En la fig. 9 izquierda se puede observar que, en líneas generales, el *gap*

relativo promedio aumenta con el tamaño de la instancia para todas las heurísticas constructivas golosas implementadas. Este resultado es esperable porque a mayor tamaño de instancia es más probable que un algoritmo goloso tome una decisión local que afecte negativamente la calidad de la solución futura y, por su naturaleza, esta decisión no es reevaluada. Se observan también oscilaciones en la calidad de la solución que pueden deberse a particularidades topológicas de cada instancia.

Como se menciona en la Sección 2, la heurística WP es una mejora de la heurística W. Los resultados obtenidos en la fig. 9 soportan está observación, dado que para todas las instancias evaluadas el gap relativo promedio es menor para WP que para W. A su vez, como la mejora de WP respecto de W está asociada a intentar colorear vértices que W no colorearía, esto requiere corroborar la factibilidad de dicha decisión en cada caso. En la fig. 9 derecha, se observa que esto implica un aumento en el tiempo promedio de ejecución de WP respecto de W. Sin embargo, la diferencia parece ser despreciable.

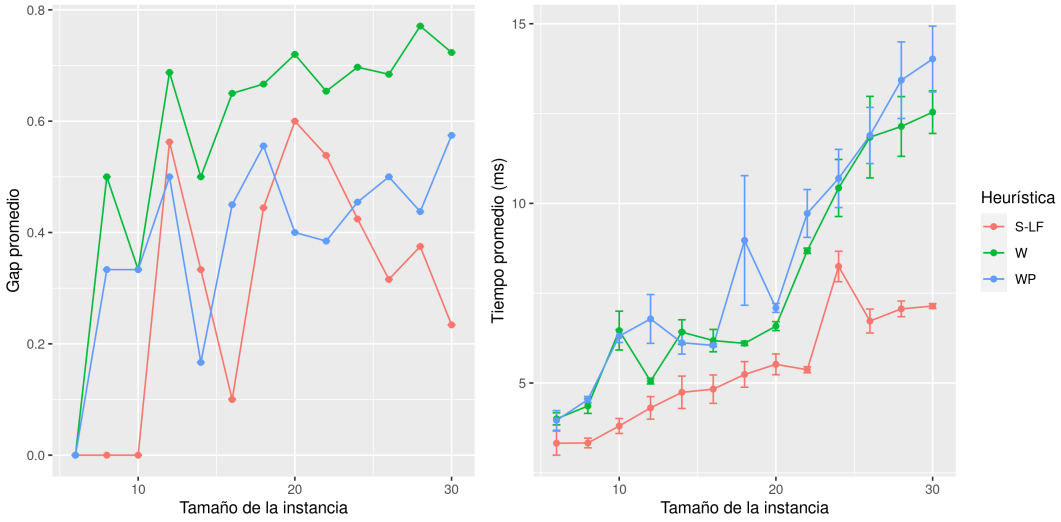


Figura 9: Gap relativo promedio en función del tamaño de la instancia (izq.) y tiempo promedio de ejecución en función del tamaño de la instancia. Los puntos representan el valor medio y las barras el error estándar. Se realizaron 5 repeticiones por instancia.

Por otro lado, también se observa que para muchas instancias el desempeño de WP y SL-F parece ser comparable. Sin embargo, para tamaños de instancia muy chicos y muy grandes S-LF parece ser mejor. Una posible explicación para las instancias de mayor tamaño es que como WP busca aumentar el impacto evaluando vértices de a pares, dependiendo del orden de evaluación de los vértices conectados en H, podría ocurrir que se generen subgrupos de vértices con colores distintos. Esto no pasaría en S-LF ya que minimiza la cantidad de colores, aumentando así el impacto como consecuencia y desconociendo las relaciones presentes en H. Finalmente, se puede observar que el tiempo de ejecución de SL-F es mucho menor al de las otras dos heurísticas constructivas golosas.

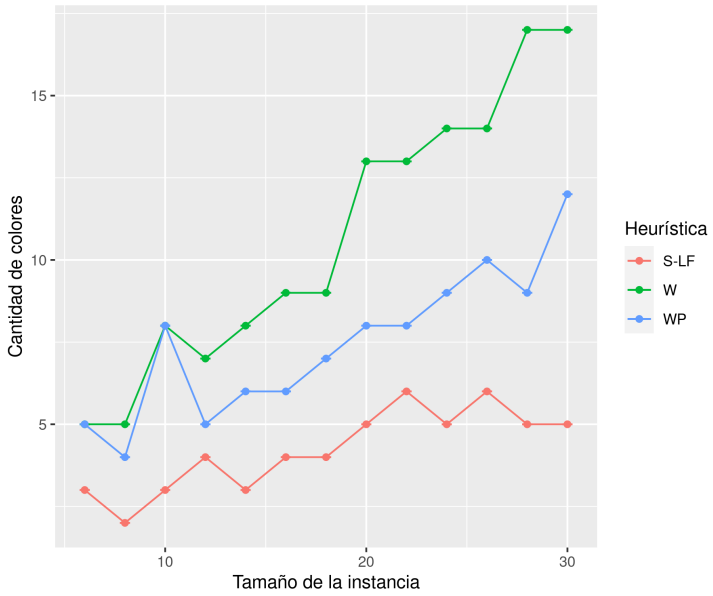


Figura 10: Cantidad de colores de la solución obtenida en función del tamaño de la instancia para las distintas heurísticas constructivas golosas. Se realizaron 5 repeticiones por instancia.

En la fig. 10 se corrobora que, para cada tamaño de instancia, la cantidad de colores de la solución obtenida es mayor para W, intermedia para WP y menor para S-LF.

Teniendo en cuenta los resultados de esta sección, se concluye que S-LF es la mejor heurística constructiva golosa en términos de distancia respecto del óptimo y tiempo de ejecución. Sin embargo, esto no implica que necesariamente sea la mejor heurística inicial para todas las metaheurísticas y parámetros posibles porque la cantidad de coloreos de la solución inicial podría tener un impacto importante sobre la posibilidad de generar vecinos, dependiendo de la estrategia empleada por cada metaheurística.

5.3. Impacto de parámetros

De acuerdo a lo concluido en la sección anterior, se evaluó a cada metaheurística partiendo de la solución proporcionada por cada una de las heurísticas constructivas golosas. Adicionalmente, para cada una de estas combinaciones, se variaron los parámetros de las metaheurísticas como se describe en la sección 5.1 con el objetivo de poder encontrar una configuración óptima. Para esto sólo se utilizaron las instancias de entrenamiento.

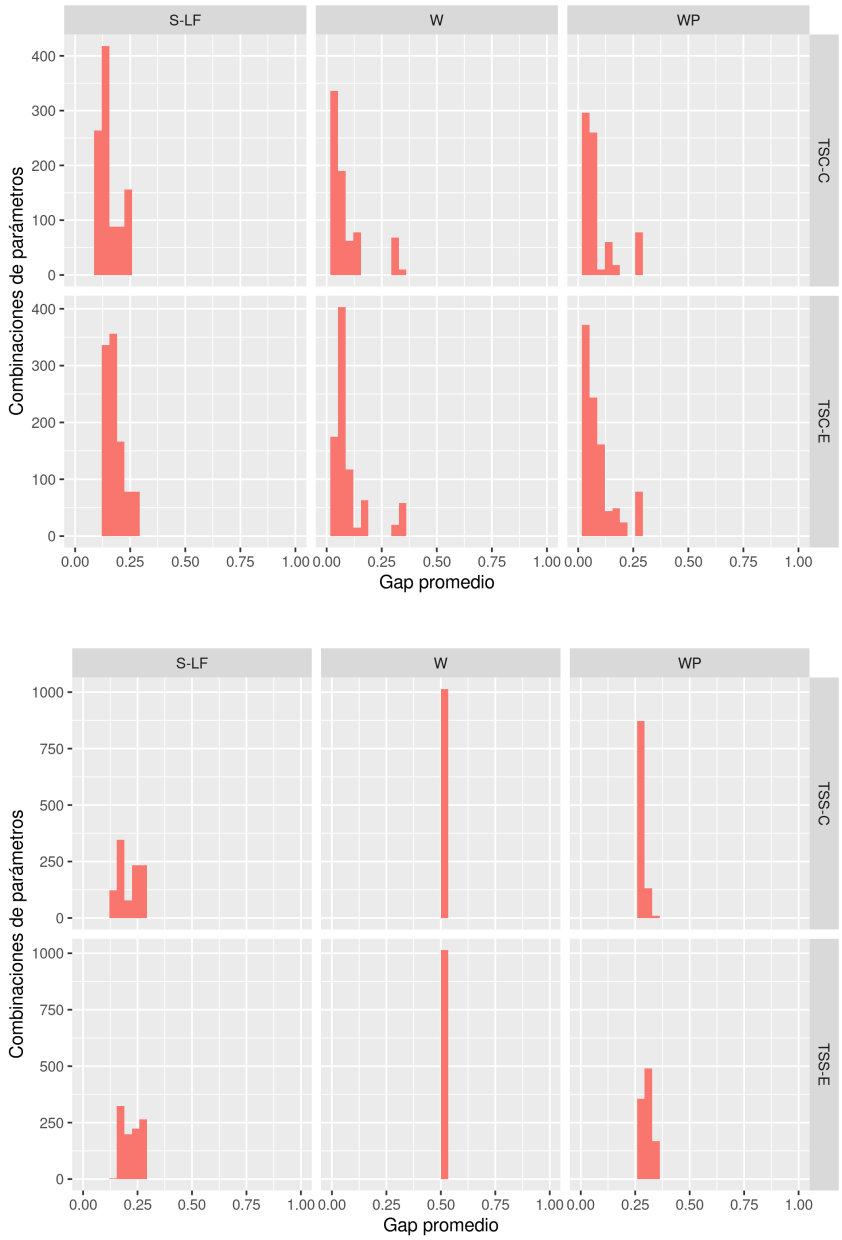


Figura 11: Histogramas que indican la cantidad de combinaciones de parámetros que resultan en un mismo valor de gap relativo promedio. En la parte superior, se muestran los resultados obtenidos para las variaciones de Tabú search que usan vecindades generadas por la metodología **change** (TSC) y, en parte inferior, los resultados obtenidos para las variaciones que usan vecindades generadas por la metodología **swap** (TSS). A su vez, cada una está dividida en paneles según el tipo de memoria usada (C por coloreo y E por estructura) y la heurística constructiva inicial empleada (S-LF, W o WP).

En la fig. 11 se muestran, en la parte superior, los resultados obtenidos para las variaciones de Tabú search que usan vecindades generadas por la metodología **change** (TSC) y, en parte inferior, los resultados obtenidos para las variaciones que usan vecindades generadas por la metodología **swap** (TSS). A su vez, cada una está dividida en paneles según el tipo de memoria usada (C por coloreo y E por estructura) y la heurística constructiva inicial empleada (S-LF, W o WP). En cada panel se grafica un histograma que indica la cantidad de combinaciones de parámetros que resultan en un mismo valor de gap relativo promedio.

En líneas generales, a pesar de los amplios rangos de los parámetros evaluados, se observa una gran cantidad de combinaciones de los mismos que dan resultados equivalentes. En la siguientes secciones se

intentará comprender por qué ocurre esto.

Más allá de esto, para TSC (figura superior) se observa un rango de variación en la calidad de la solución ocasionado por la variación de los parámetros independientemente del tipo de memoria y de la heurística constructiva inicial. Adicionalmente, para esta metodología de construcción de vecindad, parece ser peor la heurística constructiva S-LF que WP y W, ya que para estás últimas dos el mejor gap está más cerca de 0. Sin embargo, existen combinaciones de parámetros que permiten obtener resultados comparables entre todos los tipos de heurísticas constructivas golosas para TSC.

En el caso de TSS (figura inferior) la variación en la calidad de la solución es más reducida y es dependiente de la heurística constructiva inicial. Se observan grandes diferencias de gap promedio que parecen depender más de la heurística inicial que de los parámetros. A pesar de esto, se puede concluir que, en el caso de emplear la heurística constructiva S-LF, parece haber lugar para la optimización de parámetros y posiblemente en el caso de WP también. Para esta metodología de generación de vecinos, parece ser mejor la heurística S-LF que W y WP ya que el mejor *gap* está más cerca de 0 para S-LF. Adicionalmente, no existe combinación de parámetros que logre resultados competitivos para W y WP respecto de S-LF.

Para analizar el impacto de variar cada parámetro sobre el desempeño de las metaheurísticas y tratar de comprender por qué distintas combinaciones de los mismos pueden resultar equivalentes, se decidió empezar evaluando el efecto de variar el porcentaje de vecindad.

5.4. Impacto del porcentaje de vecindad

Dados los resultados vistos en la sección anterior, resultó interesante comenzar a analizar cada parámetro en particular para intentar dilucidar el impacto de cada uno en el desempeño de los algoritmos. Se comenzó analizando el efecto de tomar distintos tamaños de vecindad, ya que esto podría implicar cambios notables en el comportamiento del algoritmo. Para tamaños de vecindad baja, podría suceder que se ignoren mejoras de impacto que formen parte de la vecindad pero hayan quedado fuera por estar tomando una porción demasiado pequeña. Por el contrario, tomar la totalidad de la vecindad podría resultar en un estancamiento en óptimos locales.

En la fig. 12 se puede observar una fuerte variación en los resultados al variar el porcentaje de vecindad. Respecto a TSC se puede ver que tomar una vecindad más grande resulta más provechoso, encontrando sus óptimos en los valores del rango $70 - 100$. Por el contrario, tomar una vecindad demasiado pequeña parece restringir demasiado las posibilidades de movimiento, inhabilitando posibles soluciones vecinas mejores.

Respecto de TSS se puede observar que algunos algoritmos como TSS-C-WP y TSS-E-WP encuentran óptimos para porcentajes de vecindad medianos. Se corrobora que TSS sólo varía el gap promedio de forma relevante cuando la heurística inicial es S-LF. Para la heurística WP la variación es mucho menor y para W es inexistente. Esto lleva a pensar junto con lo visto en la subfig. **??**, que los algoritmos de *Tabú Search Swap* no ofrecen mejoría alguna para el algoritmo inicial W, estancándose siempre en el impacto obtenido por la solución inicial. Sólo ofrecen mejoras leves para el algoritmo inicial WP.

Lo observado en esta sección permite concluir que el cambio del porcentaje de vecindad puede explicar en gran medida la variación del gap vista en la fig. 11, mostrando así la injerencia de este parámetro en la calidad de las soluciones obtenidas por el algoritmo. Por este motivo, se decidió evaluar el impacto de los parámetros restantes teniendo en cuenta el porcentaje de vecindad empleado.

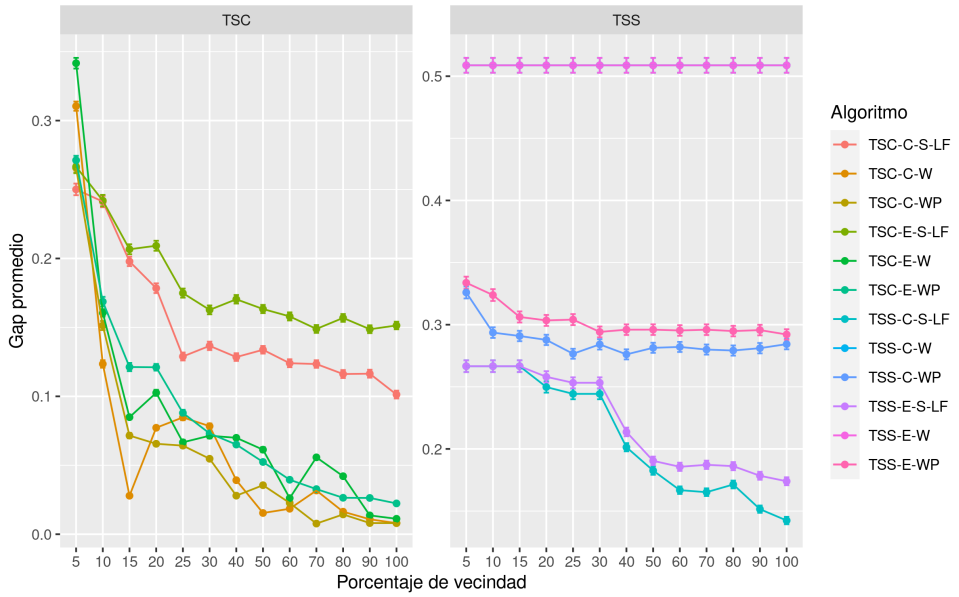


Figura 12: Gap promedio según combinación de algoritmo de Tabú Search con algoritmo goloso inicial para diferentes porcentajes de vecindad.

5.5. Impacto de aspirar

Otro de los metaparámetros cuyo análisis resultó importante fue la utilización de una función de aspiración. Para ello, se corrieron los diferentes métodos de Tabú con un algoritmo goloso inicial óptimo para cada uno. Los resultados de TSS no presentaron diferencia alguna en la utilización de la función de aspiración para ninguna estructura de memoria.

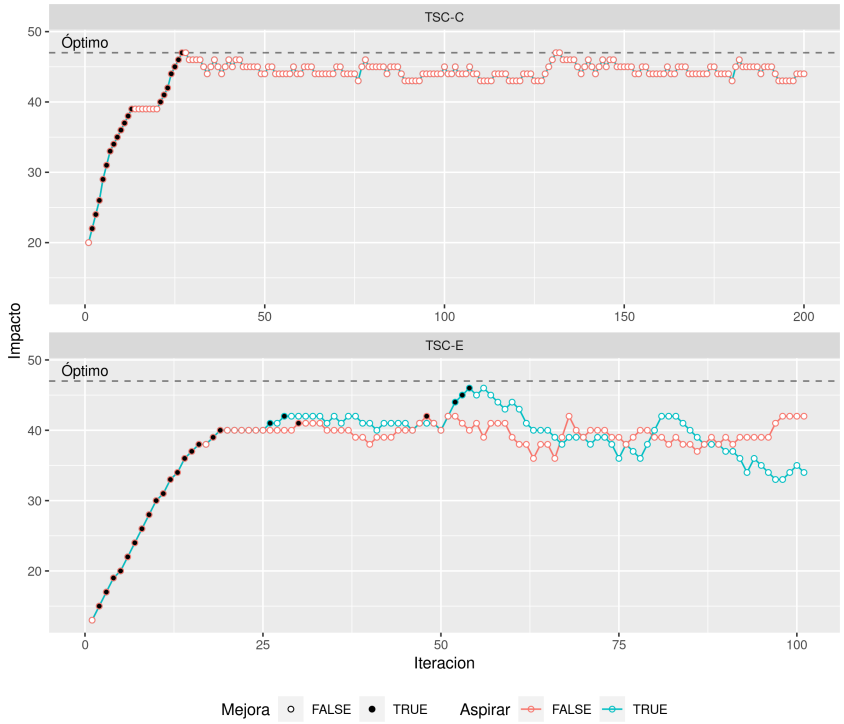


Figura 13: Se observa el impacto en función del número de iteraciones para los algoritmos TSC-C Y TSC-E. Se observan en rojo los casos evaluados con función de aspiración y, en azul, aquellos que no. Los círculos rellenos implican que se observa una mejora en el impacto. Se realizaron 5 repeticiones por instancia.

Para el algoritmo TSC se observan dos situaciones diferentes. En el caso de utilizar una memoria por estructura se pueden obtener mejores resultados a partir de la función de aspiración. Esto se debe a que volver a utilizar un cambio de color en un vértice específico en un futuro coloreo distinto puede permitir una mejoría en el impacto de la solución, ya que dos coloreos radicalmente distintos (y con distinto impacto) pueden ser vecinos del mismo cambio estructural. En este sentido, la utilización de la función de aspiración permite desbloquear muchas soluciones posibles que permiten impactos distintos.

En el caso de utilizar una memoria por coloreo, aspirar no presenta ninguna diferencia con no hacerlo. Esto se condice con lo dicho anteriormente, ya que al haber pasado por un coloreo específico, se contemplan esa solución y sus vecinas. Luego, volver a pasar por esa situación no presenta nuevas oportunidades de coloreo que permitan una mejoría en el impacto. A su vez, resulta muy poco probable que un coloreo sufra los suficientes cambios como para pasar por una solución y luego volver exactamente al mismo coloreo. En este sentido, la probabilidad de volver a un coloreo exacto anterior resulta muy baja y no presenta ninguna mejoría, por lo que la función de aspiración no genera cambios en el impacto obtenido por esta implementación.

Finalmente, considerando la mejoría que brinda la función de aspiración, el costo temporal que ésta requiere resulta tolerable, por lo que resulta provechoso utilizarla. Esto puede verse en mayor profundidad en las figuras 21, 22, 19 y 20.

5.6. Impacto de tamaño de memoria

En el paradigma computacional, los recursos de memoria y tiempo suelen ser elementos cuya optimización puede llevar a resultados considerablemente mejores. En el caso de los algorítmos metaheurísticos, la memoria y el tiempo resultan de vital importancia para un correcto funcionamiento. Sin embargo, se debe evitar el abuso de los mismos para no sumar gastos innecesarios. Por ello, se decidió establecer un rango amplio de memoria 2 y luego verificar cuánta era realmente necesaria para llegar a una buena

solución.

Se corrió cada combinación de algoritmos Tabú con heurísticos iniciales y se graficó cada uno según porcentaje de vecindad. A su vez, se agregó una línea vertical con la cantidad de iteraciones efectivas[1] realizadas y sus respectivas barras de error, lo que permitió analizar en detalle la real utilización de la memoria.

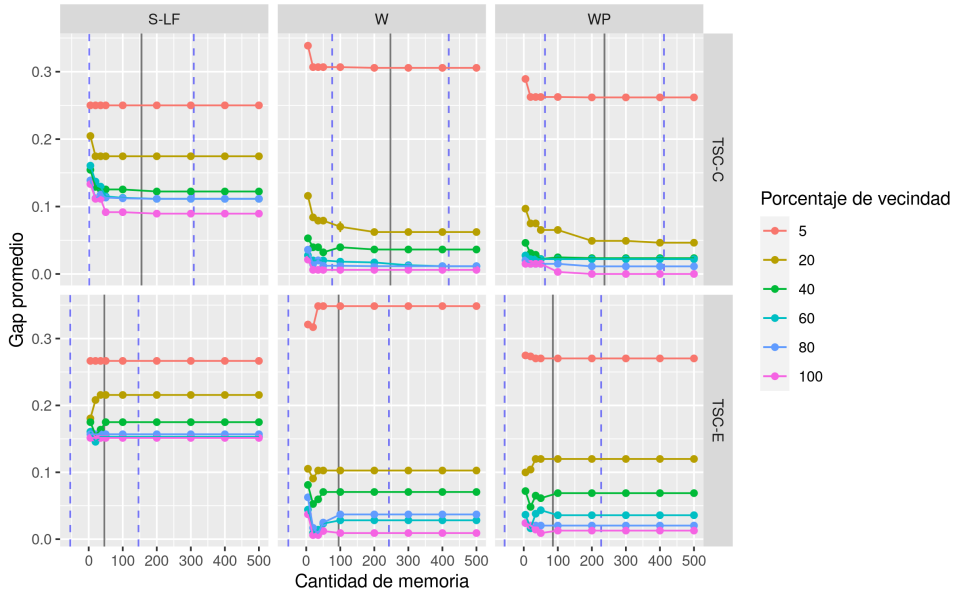


Figura 14: Gap promedio según cantidad de memoria asignada para cada combinación de algoritmo. Se grafican líneas verticales con iteraciones efectivas realizadas por cada algoritmo.

En el caso de TSC, se puede observar que las iteraciones efectivas hacen que memoria no se use, lo que indica que no hace falta tanta memoria en algunos algoritmos. A su vez, no se notan cambios significativos en el gap para memoria mayor a 200 aproximadamente, lo que parecería mostrar que no es necesaria una gran cantidad de memoria para mantener una zona tabú efectiva.

A su vez, entre las dos diferentes estructuras de TSC (TSC-E y TSC-C) se ven diferencias en las iteraciones efectivas, realizando TSC casi el doble en promedio. Esto podría deberse a lo explicado anteriormente, ya que al no recortar tanto la vecindad por la zona tabú definida en base a coloreo, se precisan más iteraciones para analizar la vecindad hasta quedarse sin vecinos o terminarse las iteraciones.

Respecto a los algoritmos iniciales elegidos, se puede observar que W y WP funcionan de manera similar en TSC-C y TSC-E pero S-LF no. Esto se debe a que la lógica del *change* genera un nuevo vecino a partir de colores pre-existentes en el grafo no presentes en sus adyacentes. Por lo tanto, sólo puede mantener o disminuir el número de colores en el grafo. En consecuencia, al partir de una heurística golosa que minimiza la cantidad de colores[2], la cantidad de soluciones posibles que puede explorar el algoritmo se ve reducida afectando negativamente a su performance.

[1] Iteraciones realmente efectuadas que terminaron antes de llegar al tope de iteraciones asignadas al algoritmo.

[2] Notar que minimizar la cantidad de colores pensando únicamente en G puede ser contraproducente porque tener impacto implica repetir colores en H.

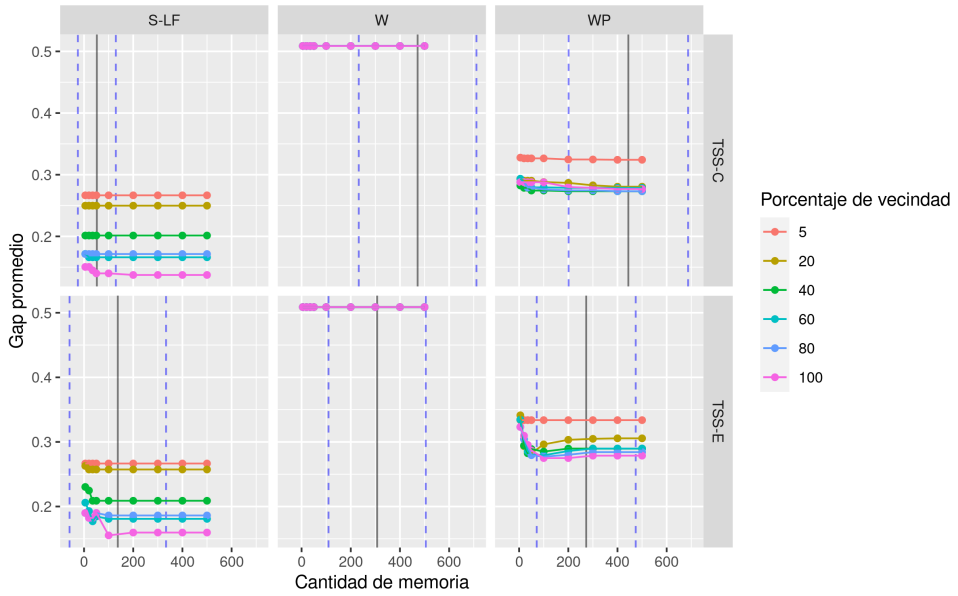


Figura 15: Se observa el gap promedio en función de la cantidad de memoria para todas las combinaciones de tabú search swap y los algoritmos heurísticos iniciales. En todos los casos, se grafican los resultados obtenidos para valores de porcentaje de vecindad en [5, 20, 40, 60, 80, 100]. Se realizaron 5 repeticiones por instancia.

A primera vista, para TSS utilizando los algoritmos iniciales W y WP parecería tener sentido usar una gran cantidad de memoria, ya que no estaría siendo anulada por la cantidad de iteraciones. Sin embargo, para W la performance no cambia respecto de este parámetro, descartando esta posibilidad. En este caso, lo que sucede es que hay gran vecindad que la memoria no es capaz de cubrir completamente, por lo que siempre habrá vecinos para analizar (aunque no mejoren el impacto). Esto sucede porque la heurística golosa W utiliza muchos colores. Luego, como TSS sólo intercambia colores (no introduce nuevos) de vértices y W utiliza muchos colores, la vecindad resulta muy grande y la memoria no logra acapararla toda, por lo que el algoritmo prácticamente se detiene cuando se queda sin iteraciones. A su vez, es por este mismo motivo que el desempeño del algoritmo es malo. Al no tener muchos colores repetidos y no puede introducir nuevos, no logra tener un alto impacto ni tampoco mejorarlo, llegando a su óptimo rápidamente y estancándose en él.

En el caso de WP, la heurística golosa reduce la cantidad de colores en las componentes conexas de H pero no limita la cantidad en los vértices aislados en H, generando una cantidad de colores intermedia que da una vecindad de tamaño intermedio. Por otro lado S-LF reduce la cantidad de colores en general, causando una mayor reducción del número de colores y en consecuencia de la vecindad. Esto explica por qué las iteraciones efectivas decrecen en el orden W, WP, S-LF y la performance aumenta en el mismo sentido.

5.7. Impacto de cantidad de iteraciones

Según lo visto en la sección anterior, resultaría lógico ver una gran diferencia entre las iteraciones asignadas a cada algoritmo y las efectivas. Para verificar esto se analizaron las diferentes combinaciones de algoritmos y se graficó el gap promedio según la cantidad de iteraciones. Al igual que antes se agregó una línea vertical que indica las iteraciones efectivas.

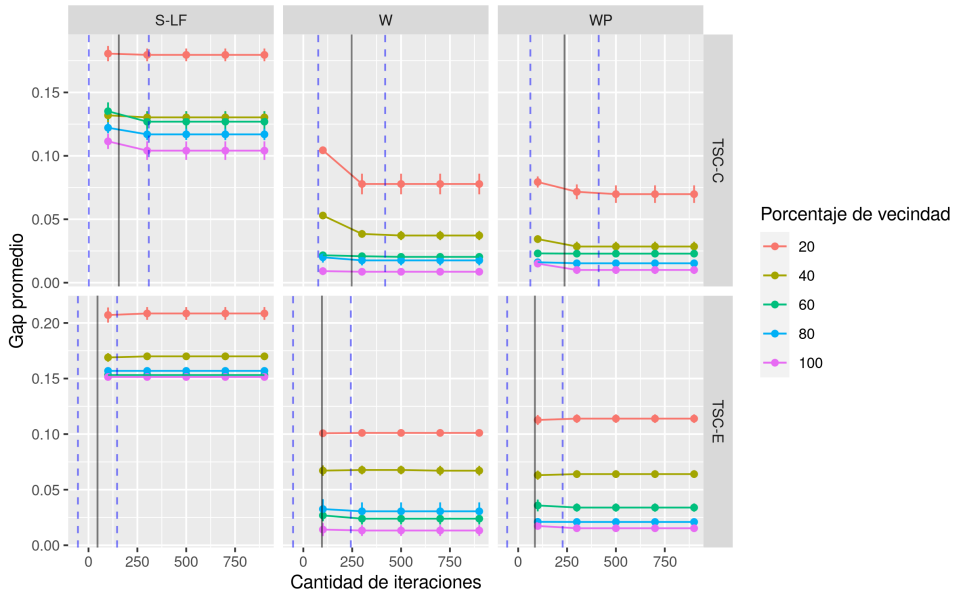


Figura 16: Se observa el gap promedio en función de la cantidad de memoria para todas las combinaciones de tabú search change y los algoritmos heurísticos iniciales. En todos los casos, se grafican los resultados obtenidos para valores de porcentaje de vecindad en [5, 20, 40, 60, 80, 100]. Se realizaron 5 repeticiones por instancia.

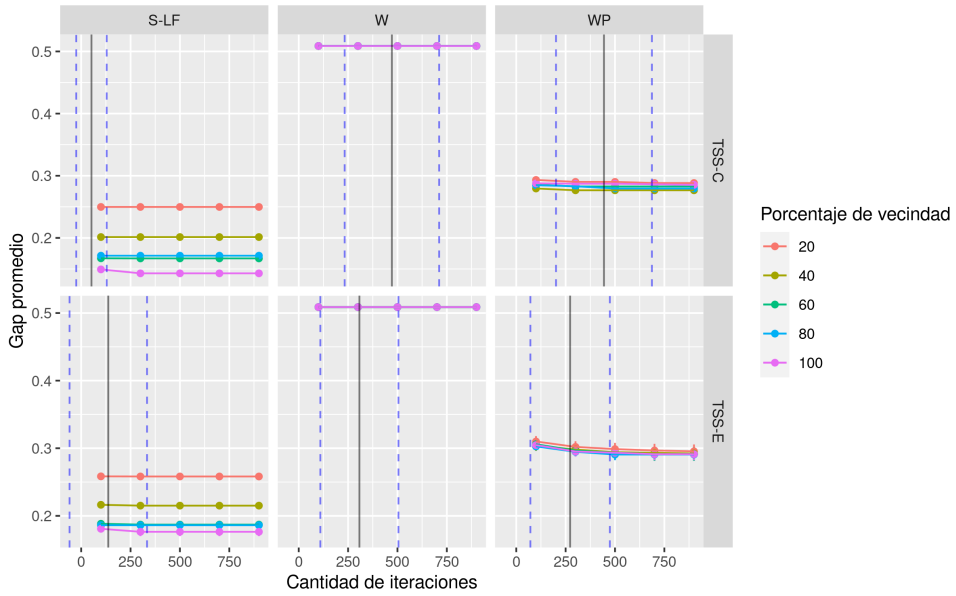


Figura 17: Se observa el gap promedio en función de la cantidad de memoria para todas las combinaciones de tabú search swap y los algoritmos heurísticos iniciales. En todos los casos, se grafican los resultados obtenidos para valores de porcentaje de vecindad en [5, 20, 40, 60, 80, 100]. Se realizaron 5 repeticiones por instancia.

Considerando está sección y la anterior, ahora resulta más claro por qué había muchas combinaciones de parámetros que dan el mismo resultado en las Figura 11. En general, se puede concluir que corridas con alta memoria o altas iteraciones terminan siendo equivalentes a las corridas que tienen valores cercanos a la cantidad de iteraciones efectivas.

5.8. Evaluación

Finalmente, para cada combinación de Tabú y heurística constructiva inicial, se eligieron las combinaciones que dieron mejor Gap en el set de datos de entrenamiento. En caso de que más de una configuración de parámetros diera resultados equivalentes se desempató por el tiempo de ejecución. Se detallan en la Tabla 1.

En lineas generales no se observó correlación entre el tiempo de ejecución y la calidad de las soluciones entre los diferentes algoritmos de *Tabú Search*. Esto puede observarse más en detalle en la fig. 23

Se puede observar que hay al menos un orden de magnitud de diferencia entre TSC usando heurística W o WP y el resto de los algoritmos, por lo que S-LF resulta la peor heurística constructiva para los TSC pero la mejor para los TSS por lo que explicado en la Sección 6.5. A su vez, notar que para TSC-E WP el número de iteraciones que da el mejor resultado es alto pero la diferencia en tiempo con las corridas que dan el resultado óptimo es muy chica, por lo que haber elegido ésta puede ser resultado de fluctuaciones experimentales. Sería interesante aumentar la cantidad de repeticiones y ver si se sigue eligiendo esta combinación de parámetros.

Por último, pareciera ser que la elección de estructura de Memoria no afecta el gap de TSS (salvo cuando usa S-LF porque tiene más posibilidades de encontrar mejorías), sólo el costo temporal. Aquí se confirma una vez más la importancia e injerencia del algoritmo inicial elegido en el desempeño del *Tabú Swap*.

	Variante de tabu	Heurística Inicial	Iteraciones	Porcentaje de Vecindad	Cantidad de Memoria	Aspirar	Gap medio	Tiempo medio (s)
1	TSC-C	WP	300	90	100	Si	0	4.50
2	TSC-E	W	100	90	35	Si	0.003	1.04
3	TSC-C	W	300	50	50	No	0.003	3.35
4	TSC-E	WP	900	100	50	Si	0.009	0.80
5	TSC-C	S-LF	300	100	50	Si	0.089	2.68
6	TSC-E	S-LF	300	70	35	Si	0.131	0.34
7	TSS-C	S-LF	300	100	50	No	0.137	3.37
8	TSS-E	S-LF	300	100	100	Si	0.154	4.47
9	TSS-E	WP	500	60	100	Si	0.271	10.56
10	TSS-C	WP	700	25	35	Si	0.271	15.29
11	TSS-E	W	100	5	20	No	0.509	1.85
12	TSS-C	W	100	5	20	Si	0.509	2.20

Cuadro 1: Cuadro comparativo que presenta la mejor combinación de metaparámetros para cada algoritmo evaluado (combinación de tabú search y heurísticas iniciales). Se ordenan las entradas según el valor de gap promedio obtenido.

Para analizar el desempeño de los algoritmos en instancias de testeo y evitar así cualquier tipo de *overfitting* se corrió cada combinación de algoritmo tabú con inicial, utilizando los parámetros óptimos encontrados en el entrenamieinito. Los resultados se detallan en la Figura 18.

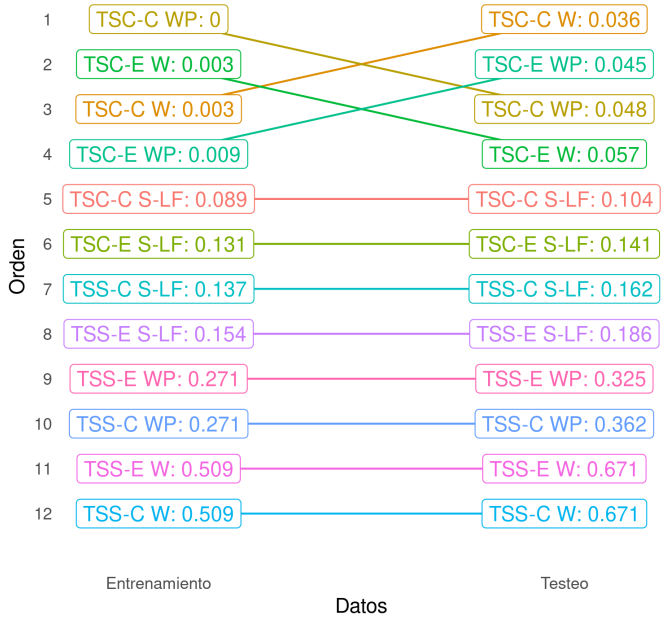


Figura 18: Se observa el orden o ranking de algoritmos según los valores obtenidos de gap medio. A la izquierda se observa el ranking calculado para las instancias de entrenamiento, mientras que a la derecha, para las instancias de testeo.

Los resultados con el set de datos de testeo son peores, lo cual es esperable pero podría estar indicando *overfitting* en el entrenamiento, sobre todo para TSC-C WP que da gap 0 en las instancias de entrenamiento. Sin embargo, se mantiene la diferencia de al menos un orden de magnitud entre TSC usando heurística W o WP y el resto de los algoritmos, por lo que el orden de eficiencia de los algoritmos se mantiene.

6. Conclusiones

En este trabajo se presentaron distintas técnicas para resolver PCMI, que resulta aplicable a problemas de la vida real. Se corrieron los algoritmos desarrollados para instancias de distintos tamaños, analizando y comparando el tiempo de ejecución y la calidad de las soluciones encontradas.

Se entrenaron los parámetros óptimos de Tabú Search (TS) en un subconjunto de instancias de *train*, y se corroboraron en instancias de *test* obteniendo buenos resultados.

Se pudo concluir que los algoritmos golosos son buenos para el costo computacional que implican, pero vale la pena invertir el tiempo extra por la calidad que brinda TS. Se probó que TS puede resolver el problema de forma satisfactoria cuando la generación de la vecindad emplea la estrategia de change. Cabe destacar que en este caso es importante emplear como heurística inicial a W o WP.

Sería interesante repetir la experimentación en un número de instancias mayor para poder determinar con mayor seguridad el ranking de las diferentes implementaciones de TSC que dieron un gap relativo muy similar.

7. Apéndice

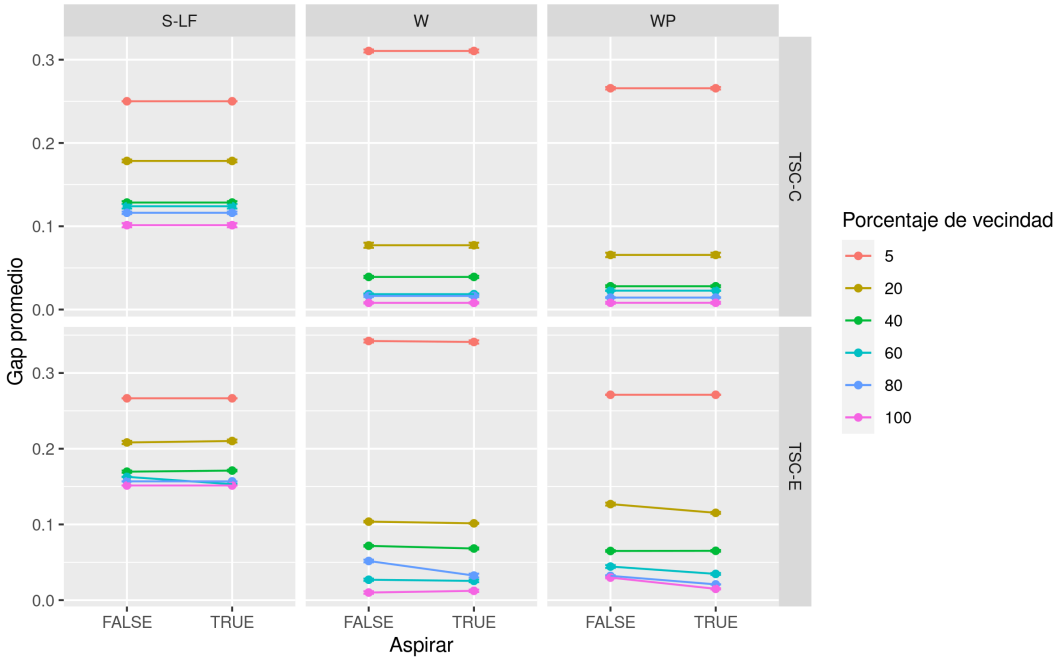


Figura 19: Gap relativo medio para ejecuciones de TSS activando o no la aspiración. Cada panel representa una combinación de heurística constructiva inicial y tipo de memoria. Los puntos representan el valor medio y las barras el error estándar. Se realizaron 5 repeticiones por instancia.

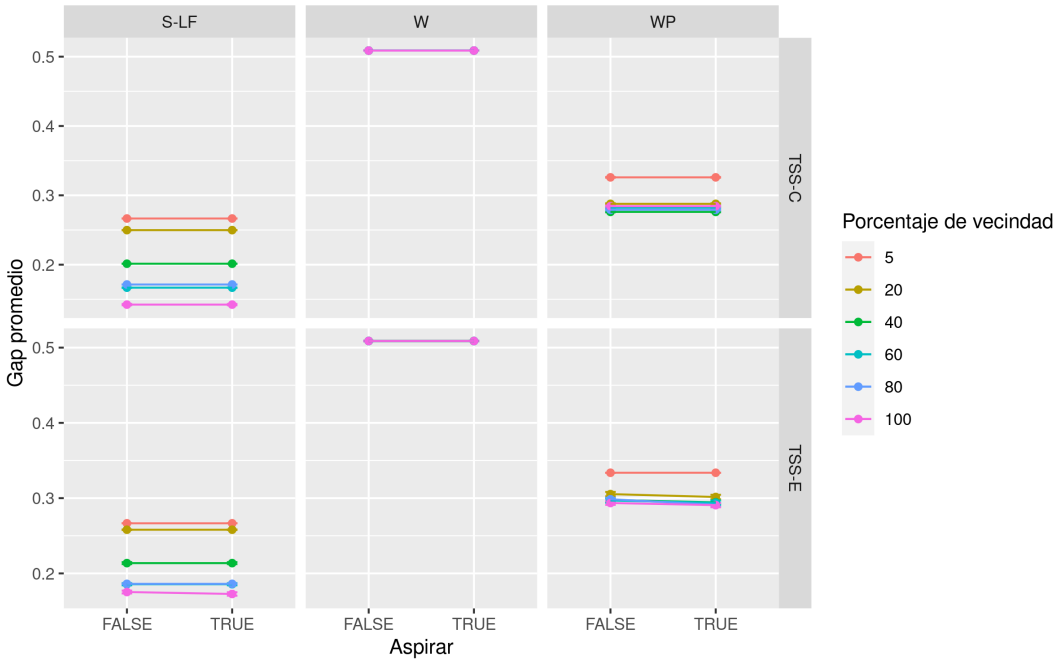


Figura 20: Gap relativo medio para ejecuciones de TSS activando o no la aspiración. Cada panel representa una combinación de heurística constructiva inicial y tipo de memoria. Los puntos representan el valor medio y las barras el error estándar. Se realizaron 5 repeticiones por instancia.

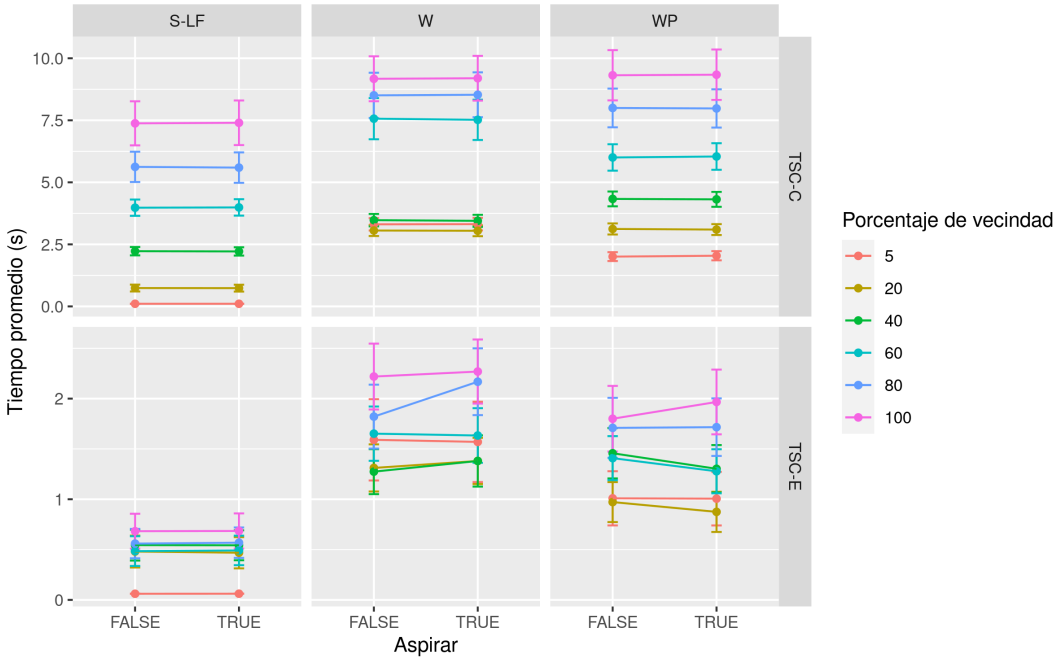


Figura 21: Tiempo de ejecución medio para ejecuciones de TSC activando o no la aspiración. Cada panel representa una combinación de heurística constructiva inicial y tipo de memoria. Los puntos representan el valor medio y las barras el error estándar. Se realizaron 5 repeticiones por instancia.

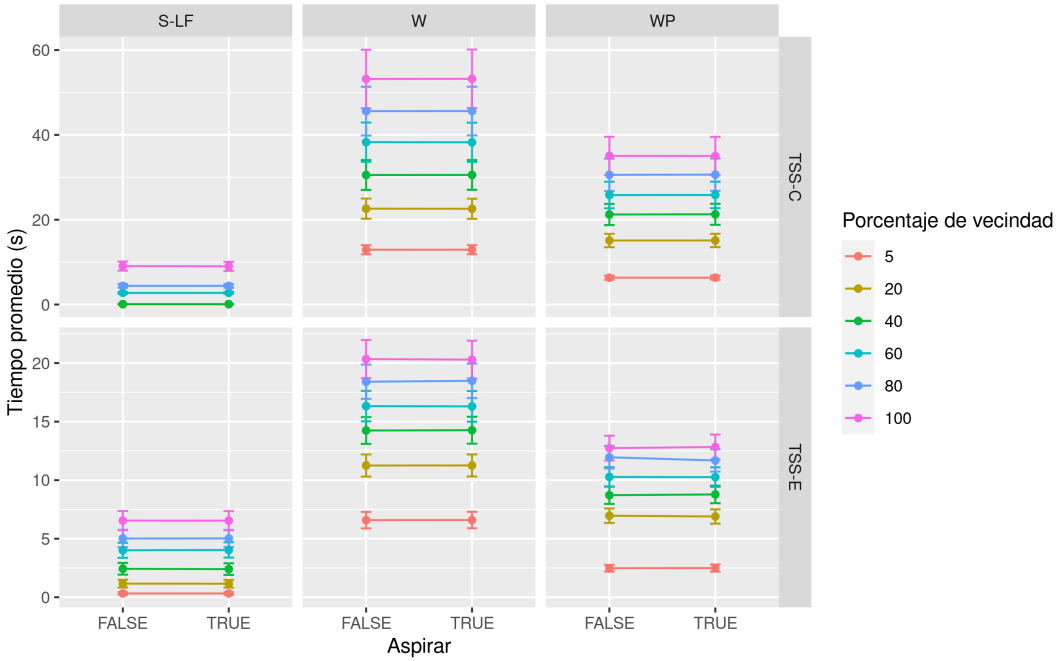


Figura 22: Tiempo de ejecución medio para ejecuciones de TSS activando o no la aspiración. Cada panel representa una combinación de heurística constructiva inicial y tipo de memoria. Los puntos representan el valor medio y las barras el error estándar. Se realizaron 5 repeticiones por instancia.

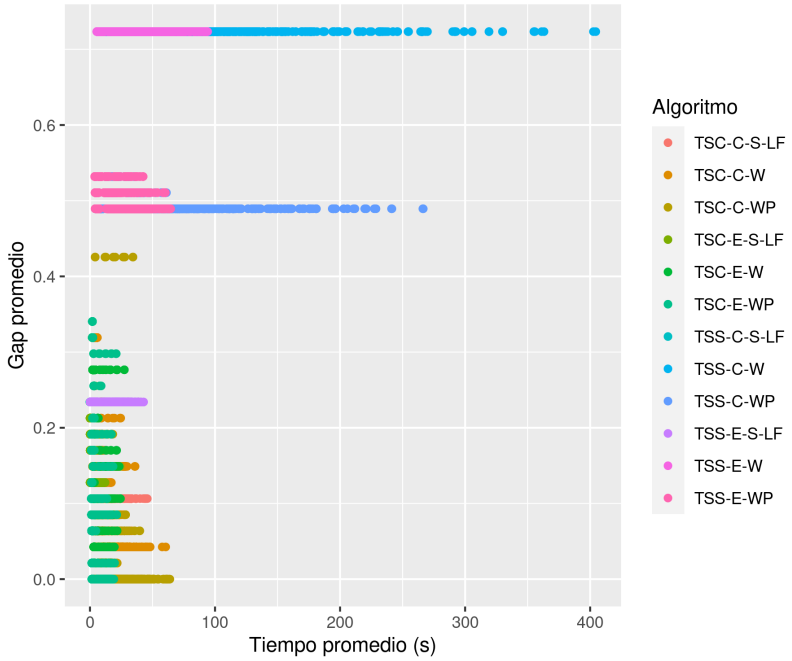


Figura 23: Gap relativo promedio en función del tiempo promedio de ejecución. Los puntos representan el valor medio. Se realizaron 5 repeticiones por instancia. No se observa correlación.