DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

TP2: PCMI

Algoritmos y estructuras de datos III

Integrante	LU	Correo electrónico
Rabinowicz, Julia	48/18	julirabinowicz@gmail.com
Phoenix, Nathaniel	151/18	nathanphoenix.arg@gmail.com
Otouzbirian, Leandro Ariel	5/18	lotouzbirian@gmail.com
Tarsia, Luciano Fabrizio	84/19	lucianoftarsia@gmail.com

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

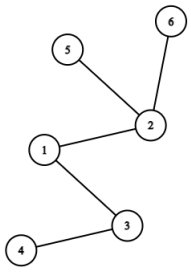
Índice

1. Introducción

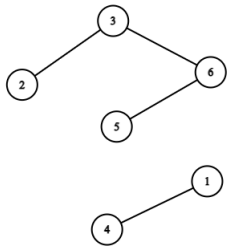
En el siguiente trabajo nos proponemos intentar resolver el Problema de Coloreo de Máximo Impacto (PCMI) desarrollando e implementando diferentes algoritmos que busquen devolver soluciones óptimas. PCMI es un problema de optimización combinatoria, es decir, es un problema en donde se busca, bajo cierto criterio, encontrar la mejor solución dentro de un conjunto de soluciones factibles. Dicho conjunto suele ser de gran tamaño, por lo que los algoritmos exhaustivos pueden llevar demasiado tiempo. Por esta razón, los algoritmos que utilizaremos son heurísticas y metaheurísticas: obtendrán soluciones de manera más rápida, pero estas no serán necesariamente las óptimas.

El problema planteado consiste en encontrar, dado dos grafos con idénticos vértices, un coloreo para el primero que maximice el impacto en el segundo. Dicho de otra manera, se tienen dos grafos $G = (V, E_G)$ y $H = (V, E_H)$, y lo que se busca es un coloreo válido de los vértices de G, tal que se tenga la mayor cantidad posible de aristas en H tales que los dos vértices en los que inciden tienen el mismo color. Más formalmente, dado un conjunto de colores C y un coloreo $c : V \to C$ de G, queremos maximizar la función $I(c)$ definida como la cantidad de aristas $(i, j) \in E_H$ que cumplan que $c(i) = c(j)$.

A continuación presentamos un ejemplo del problema a resolver, con una cantidad $n = 6$ de vértices, y algunas posibles soluciones.



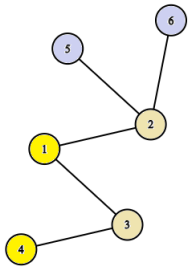
(a) Grafo $G = (V, E_G)$



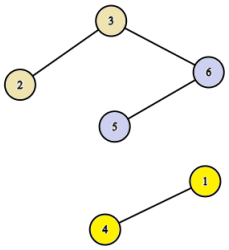
(b) Grafo $H = (V, E_H)$

Figura 1: Ejemplo de grafos para buscar un Coloreo de Máximo Impacto.

Puede verse en la siguiente figura que con una solución que utiliza 3 colores, se obtiene un coloreo válido en G y un impacto óptimo de valor 3 en H, que surge de las aristas (2,3), (1,4) y (6,5). Esta solución es óptima, pues el mayor impacto posible está acotado por 4 (la cantidad de aristas del grafo H), pero el valor 4 no es posible de alcanzar: para ello los nodos 2, 3, 5 y 6 deberían tener el mismo color, dejando en G una adyacencia inválida entre el vértice 2 y el 6.



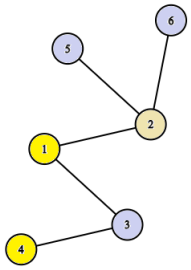
(a) Grafo $G = (V, E_G)$



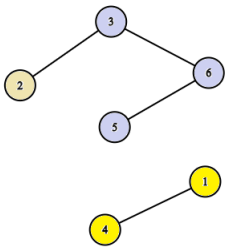
(b) Grafo $H = (V, E_H)$

Figura 2: Posible coloreo de los vértices del que se obtiene un impacto óptimo.

A continuación se muestra un coloreo diferente para los vértices, también de 3 colores, donde también se obtiene un impacto máximo. En este caso, influyen en el impacto las aristas (1,4), (3,6) y (6,5).



(a) Grafo $G = (V, E_G)$



(b) Grafo $H = (V, E_H)$

Figura 3: Otro posible coloreo con impacto óptimo.

Finalmente, podemos observar que con el siguiente coloreo se obtiene un impacto de valor 0.

(a) Grafo $G = (V, E_G)$

(b) Grafo $H = (V, E_H)$

Figura 4: Coloreo válido de los vértices para el cual el impacto obtenido es 0.

Es importante destacar, que el coloreo de los vértices buscado debe de ser válido para el grafo G, pero no necesariamente óptimo. Es decir que, por más que existe un coloreo de G con 2 colores, lo que se busca en PCMI es un coloreo que maximice el impacto en H, lo que no implica que deba obtenerse con un coloreo óptimo de G.

Los problemas de optimización combinatoria se destacan por ser muy útiles en la práctica dado que sirven para modelar diversas situaciones de la realidad. En particular, PCMI tiene diferentes aplicaciones prácticas.

Un ejemplo de un problema real que se puede modelar para resolverlo utilizando PCMI es el siguiente. Un colegio está organizando viajes de estudio con todos los alumnos de 4to año. La escuela cuenta con una cierta cantidad de micros, cada uno con capacidad para que viajen los alumnos de diferentes divisiones. Previamente al viaje, se le consultó a cada división por otras con las que tengan conflicto por lo que prefieren no viajar en el mismo micro. Se quiere entonces asignar cada división a un micro respetando las respuestas de los alumnos, pero a la vez priorizando que la mayoría de cursos del mismo turno viajen juntos. De esta manera, podemos definir el conjunto de vértices como las diferentes divisiones. Dos vértices serán adyacentes en G si alguna de las dos divisiones a las que representan manifestó que tenía conflicto con la otra, por lo que un coloreo válido en G indicará una posible distribución de los cursos en los micros. Por otro lado, dos divisiones tendrán una arista entre ellos en el grafo H si pertenecen al mismo turno.

Otra situación en la que resulta de gran utilidad el problema en cuestión es si, por ejemplo, el flamante nuevo ministro de relaciones exteriores quiere reunirse con todos los embajadores extranjeros, buscando causar la mejor impresión. Podría invitar a todos a una gran reunión, pero como quiere quedar en buenos términos, deberá fijarse de no invitar a la misma reunión a dos embajadores de países con conflictos (actuales o históricos) y por otro lado, le conviene invitar juntos a embajadores de países aliados. En este caso, los embajadores pueden ser representados como vértices, los conflictos entre ellos como las aristas de G, y sus alianzas como aristas en H. Hecho esto, un coloreo válido en G con alto impacto en H determinará entonces los invitados a cada reunión, evitando conflictos y maximizando alianzas.

Veamos un último ejemplo: supongamos que un turista quiere planificar las atracciones que visitará cuando recorra una determinada cuidad, para saber cuántos días se quedará allí antes de continuar con su viaje. Investigando descubre que la mayoría está disponible únicamente en un cierto horario, por lo que si dos atracciones deben ser visitadas en el mismo horario, sí o sí deberá ir en días diferentes. Por otro lado, para ahorrar en viaje, prefiere ir el mismo día a las que se encuentren más cerca geográficamente. Esta situación se puede modelar tomando el conjunto V como las atracciones que se quieren visitar. Las adyacencias en el grafo G representarán las que deben ser visitadas en el mismo horario. Un coloreo válido de G determinará entonces los días en los que se asistirá a cada una, y las adyacencias en H serán los sitios cercanos entre sí.

2. Algoritmos implementados

Para todos los algoritmos decidimos representar los grafos utilizando matrices de adyacencia. Llamaremos MG a la matriz correspondiente al grafo G y MH a la del grafo H. Por otro lado, los colores los representamos como enteros mayores a cero, y el coloreo se va guardando en un vector de tamaño n llamado `coloreo` donde el valor de la posición i-ésima corresponde al color del vértice $i+1$. Comienza con 0 en todas las posiciones. A continuación explicaremos cada algoritmo en detalle.

2.1. Heurística 1

La primera heurística constructiva golosa pinta los vértices uno a uno, chequeando en cada caso si se puede (o no) generar un punto más de impacto pintando el vértice en cuestión de algún color en particular.

Lo primero que debe realizar la heurística es guardar un vértice de cada componente conexa de G con el fin de poder recorrerlo en su completitud. Para ello creamos una función llamada `representantes` que recorre el grafo G mediante múltiples BFS (uno por cada componente conexa), devolviendo un conjunto de vértices tal que cada uno representa una componente conexa.

Algorithm 1 Algoritmo de `representantes`

```
1:  function Representantes(Matriz)
2:      ConjRep ← {}
3:      visitados ← vector(tamaño : Matriz.size(), valor : false)
4:      cola ← queue({})
5:      for i in range(Matriz.size()) do
6:          if !visitados[i] then
7:              cola.push(i)
8:              visitados[i] ← true
9:              ConjRep.insert(i)
10:             while !cola.empty() do
11:                 v ← cola.front()
12:                 cola.pop()
13:                 for w in ady(Matriz, v) do
14:                     if !visitados[w] then
15:                         visitados[w] ← true
16:                         cola.push(w)
17:     return ConjRep
```

Luego, para cada una de estas componentes conexas, y comenzando desde su vértice representante, el algoritmo colorea cada vértice de la misma con el color indicado por la función `colorPosible`, siguiendo el orden definido por BFS.

Algorithm 2 Algoritmo de la Heurística 1.

```
1:  function H1(rep, coloreo)
2:      color ← MG.size() − 1
3:      coloreo[rep] ← MG.size()
4:      cola ← queue({})
5:      cola.push(rep)
6:      while !cola.empty() do
7:          v ← cola.front()
8:          cola.pop()
9:          for w in ady(MG, v) do
10:             if !coloreo[w] then
11:                 colorPos ← colorPosible(w, coloreo)
12:                 if !(ady(MH, w).empty()) && (colorPos! = −1) then
13:                     coloreo[w] ← colorPos
14:                 else
15:                     coloreo[w] ← color
16:                     color ← color − 1
17:                 cola.push(w)
```

Para cada vértice $w \in V$ su color es elegido verificando si existe un vértice $u \in V$ que ya haya sido coloreado (de un color c) y que se cumpla:

- $(w, u) \in E_H$ (w y u son adyacentes en H)
- $(w, u) \notin E_G$ (w y u no son adyacentes en G)
- $\nexists x \in V, (w, x) \in E_G$ tal que x esté pintado de c (w no tiene ningún vértice adyacente en G que esté pintado de color c).

Si todo esto se cumple, entonces w y u pueden pintarse del mismo color y así aumentar en 1 el impacto, por lo que se colorea w con c. En caso de no existir un vértice u que cumpla lo anterior, se pinta a w del máximo color entre 1 y n que aún no se haya utilizado. Lo explicado es también lo que realiza la función `colorPosible` (línea 11).

Algorithm 3 Algoritmo de colorPosible.

```
1:  function colorPosible(vertex, coloreo)
2:      color ← −1
3:      adyVertex ← ady(MG, vertex)
4:      for i ∈ [0, n − 1] do
5:          if MG[vertex][i] = 0 ∧ MH[vertex][i] = 1 ∧ coloreo[i] ≠ 0 then
6:              flagAux ← true
7:              for ady in adyVertex do
8:                  if coloreo[i] = coloreo[ady] then
9:                      flagAux ← false
10:             if flagAux then
11:                 color ← coloreo[i]
12:                 break
13:     return color
```

Complejidad

Como se puede ver en el algoritmo 1, se realizará BFS una vez por cada componente conexa del grafo. Por este motivo, su complejidad es $O(n^2)$. Es importante destacar que para recorrer el grafo, debe buscar los vecinos de cada vértice y al utilizar la matriz de adyacencia esto tiene complejidad lineal en n.

En el algoritmo 3 cada componente conexa se recorre con una modificación de BFS, donde para verificar si un vértice v fue visitado, se fija si en su posición del vector `coloreo` hay un 0 o si ya fue coloreado. En caso de que haya un 0, se debe elegir un color según lo explicado en la sección anterior. La complejidad de la función `colorPosible` es entonces $O(n^2)$.

En conclusión, para cada vértice del grafo se termina haciendo lo siguiente: se verifica en $O(1)$ si fue visitado, se buscan sus adyacentes en $O(n)$ y se elige su color en $O(n^2)$. Por lo tanto, la heurística tiene complejidad $O(n^3)$.

Casos patológicos

Dado que el algoritmo presentado es una heurística, no podemos asegurar que siempre se obtenga un coloreo que derive en un impacto máximo. Estudiando su comportamiento encontramos ciertos casos para los cuales no podrá llegar a la solución óptima. Estos casos se relacionan con el hecho de que la heurística es golosa, por lo que no revisará vértices que ya hayan sido coloreados. Analicemos la siguiente situación: supongamos que hay dos vértices v y w tal que $(v, w) \notin E_H$ y $(v, w) \notin E_G$. Además, v y w tienen un vértice adyacente en común en H, pero no en G. Es decir, $\exists x$ tal que $(v, x), (w, x) \in E_H$ y $(v, x), (w, x) \notin E_G$. Para obtener un impacto máximo entonces, los tres vértices deberían pintarse del mismo color. Sin embargo, dependiendo del orden en el que son coloreados estos vértices, podría no llegarse a ese resultado. Si primero se colorea v y luego w, al momento de elegir el color de w, se verificará si tiene adyacentes en H ya coloreados, pero como v y w no son adyacentes en H, el algoritmo les asignará colores diferentes. Luego cuando llegue el momento de pintar x, se elegirá uno de los colores de v o w, pero nunca se llegará a una solución en donde los tres tengan el mismo color.

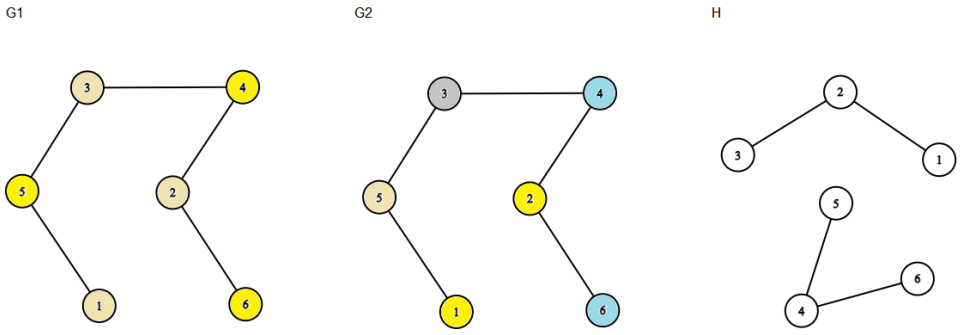
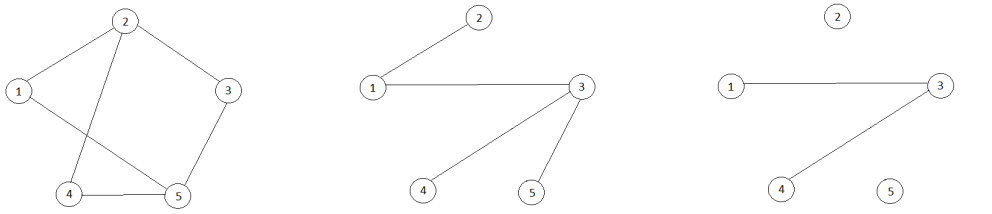


Figura 5: Ejemplo de un caso patológico de la Heurística 1

En la figura 5 G1 muestra el coloreo óptimo y G2 el obtenido por el algoritmo. Se puede observar que para cada componente conexa de H se tiene un caso como el mencionado, ya que la heurística comenzará recorriendo el vértice 1, y el vértice 3 lo recorrerá antes que el 2, y lo mismo para los vértices 4, 5 y 6.

2.2. Heurística 2

Para el caso de la segunda heurística constructiva golosa, lo que se hace es, a partir de los grafos G y H, realizar una resta entre las aristas de H y las de G. Eso define un nuevo grafo, digamos R (resta), que cumple $R = (V, E_H \setminus E_G)$. Esto se debe a que si dos vértices son adyacentes en H y también en G, nunca se podría aumentar el impacto pintándolos del mismo color dado que no resultaría un coloreo válido en G. Por esta razón, esta adyacencia en H no será de importancia para este problema, por lo que podemos omitirla. A continuación presentamos un ejemplo de cómo quedaría el grafo R para ciertos grafos G y H.



(a) Grafo $G = (V, E_G)$ (b) Grafo $H = (V, E_H)$ (c) Grafo $R = (V, E_H \setminus E_G)$

Figura 6: Ejemplo de la resta entre grafos.

Como puede verse en la figura, en el grafo R no aparecen las aristas (1,2) y (3,5) ya que son parte tanto de G como de H.

Luego de realizar la resta mencionada, pintamos cada componente conexa de R en G de un color distinto. Teniendo el grafo G totalmente pintado, se corrigen los posible errores de coloreo (por error nos referimos a dos vértices adyacentes pintados del mismo color). Esto se hace modificando primero el color del vértice v que tenga más adyacencias inválidas en G, pintándolo de un color que no haya sido utilizado hasta el momento. Esto último se realiza hasta que no haya más vértices con adyacencias inválidas. Volviendo al ejemplo de la sección 1, el algoritmo devolverá un coloreo como el de la figura 3. Esta es la parte golosa de la heurística constructiva.

La complejidad del algoritmo es de $O(n^3)$. Esto se debe a que el algoritmo primero crea un vector que será el coloreo resultante, el cual se inicializa con todos 0. Luego de eso, realiza la resta de H y G especificada anteriormente, haciendo una resta de las matrices de adyacencias de H y G (colocando 0 en la posición (i,j) sii $H[i][j] - G[i][j] \leq 0$, y asignando 1 en cualquier otro caso). Dicho procedimiento tiene complejidad $O(n^2)$. Luego se consigue un vértice de cada componente conexa usando la función `representantes` (compartida con la Heurística 1) lo que cuesta $O(n^2)$. A partir del conjunto de vértices representantes de cada componente conexa, se agarra de a un representante a la vez y se pinta cada componente conexa usando la función `pintarComponente`, que es un BFS que en lugar de guardar los visitados, los pinta, y por lo tanto la complejidad es $O(n^2)$.

Finalmente se arregla el coloreo. Esto se hace usando la función `arreglarColoreo`, que busca el vértice que tiene más vecinos con coloreo inválido en $O(n^2)$, y cambia el color de ese vértice por un color no usado anteriormente. Esta función tiene complejidad $O(n^3)$, ya que va a haber vértices con vecinos con colores inválidos hasta la iteración $n/2$, que sería el caso de que en cada iteración, se arregla solo un vecino con color inválido. De esta manera, la complejidad del algoritmo entero termina siendo $O(n^3)$.

Casos patológicos

Este algoritmo descripto no deja de ser una heurística, y como tal no asegura conseguir el impacto óptimo, de hecho, dadas ciertas características, es seguro que el óptimo no es conseguido. A esos casos los llamamos casos patológicos.

Para este algoritmo en particular, hay que notar que cuando se cambia el color de un vértice v que tiene vecinos con coloreo inválido, se cambia por un color que no sea usado por ningún otro

vértice del grafo. Esto puede causar que para dos vértices v y w con vecinos con coloreo inválido, adyacentes en H y no en G, se cambien sus colores sin considerar el hecho de usar el mismo para ambos, así perdiendo un punto de impacto.

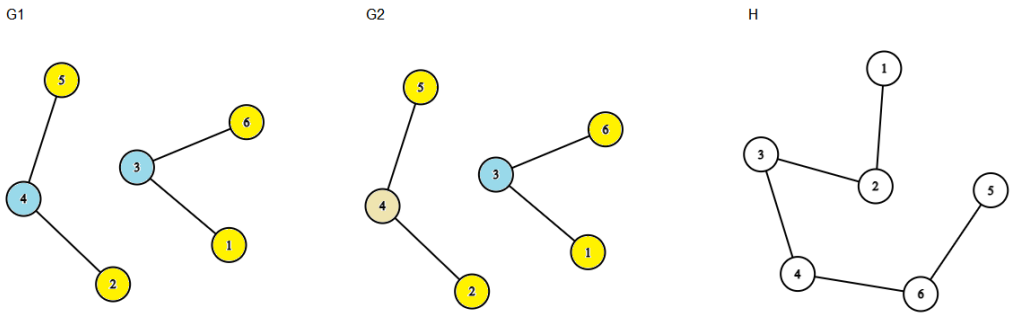


Figura 7: Ejemplo de un caso patológico de la Heurística 2.

En la figura 7 se puede observar G1 (izquierda), el coloreo de máximo impacto, G2 (medio), el coloreo dado por la heurística 2, y H (derecha). En este caso H es igual a R, por esto todos los vértices son pintados del mismo color en G. Luego, se corrigen los vértices 3 y 4, que son los que tienen más vecinos con colores inválidos. Es claro que de haber coloreado el vértice 3 del mismo color que el 4, la heurística 2 habría ganado un punto de impacto.

Esto se puede arreglar fácilmente, pero hacerlo causa que el algoritmo sea más complejo y por lo tanto decidimos dejar este "error" y que sea corregido luego por la meta-heurística.

2.3. Tabú Search

Tabú Search recibe parámetros que definirán el camino a tomar del algoritmo y la información que usará. Estos son:

- **flagHeurística**: Flag booleano que indica si se utiliza la heurística 1 o 2 para inicializar la solución inicial.
- **flagMemoria**: Flag booleano que indica que tipo de memoria se utiliza. Determinará el tamaño máximo de la lista tabú: n en el caso de memoria basada en soluciones y $\frac{n}{2}$ en el de memoria basada en estructura.
- **iterMax**: Un entero que indica la cantidad de iteraciones a realizar.

Algorithm 4 TabuSearch para PCMI

```
1: function tabuSearch(flagMemoria, flagHeuristica, iterMax)
2:     coloreoMax ← heurística()                                  ▷ inicializamos el coloreo
3:     impactoMax ← impacto(coloreoMax)
4:     solucionActual ← <-1, coloreoMax>          ▷ <ultimoVerticeModificado, ColoreoActual>
5:     listaTabu ← ∅
6:     cantIt ← 0
7:     while cantIt < iterMax do
8:         solucionActual ← crearVecindad(solucionActual, listaTabu, flagMemoria)
9:         impactoActual ← impacto(solucionActual)
10:        listaTabu.push(solucionActual)
11:        if impactoActual > impactoMax then
12:            impactoMax ← impactoActual
13:            coloreoMax ← solucionActual.second()
14:        if listaTabu.size() = tamañoMaximo then
15:            listaTabu.pop()
16:        cantIt ← cantIt + 1
17:    return coloreoMax
```

Primero se crea la solución inicial ($coloreoMax$) usando la heurística 1 o la heurística 2 según corresponda. Luego se calcula el impacto de esta solución inicial y se guarda esta información en $impactoMax$, para más tarde compararlo con el de otras soluciones. El impacto de una solución es calculado recorriendo la parte triangular superior de la matriz de adyacencia de H (sin incluir a la diagonal), sumando la cantidad de vértices adyacentes que tienen el mismo color.

Después de esto, se entra al ciclo principal, que va hacer tantas iteraciones como indique $iterMax$. Dentro de este ciclo se crea una nueva solución en cada iteración. Esta nueva solución se crea usando la función `crearVecindad`, que crea un subconjunto de la vecindad y devuelve la solución de mayor impacto. El pseudocódigo de esta función es el siguiente:

Algorithm 5 crearVecindad

```
1: function crearVecindad(solucionActual, listaTabu, flagMemoria, impactoActual)
2:     mejor ← tupla < int, vector < int >>
3:     impactoMejor ← 0
4:     coloreoAux ← solucionActual.second()
5:     for k in range(MG.size() ∗ 3) do
6:         i ← random i uniforme entre 1 y n
7:         j ← random j uniforme entre 1 y n
8:         coloreoAux[i] ← j
9:         aux ←< i, coloreoAux >
10:        impactoAux ← impacto(coloreoAux)
11:        if valida(aux, listaTabu, flagMemoria) && impactoAux > impactoMejor then
12:            mejor ← aux
13:            impactoMejor ← impactoAux
14:        coloreoAux ← solucionActual.second()
15:    impactoActual ← impactoMejor
16:    return mejor
```

Para generar las instancias de la vecindad decidimos utilizar la operación *change*, cambiando el color de un solo vértice. Las soluciones se representarán con una tupla <vértice modificado, coloreo>, y la lista tabú será una cola de tuplas como la mencionada. Para ahorrar tiempo optamos por no crear toda la vecindad, sino solo un subconjunto de la misma de tamañored3n $n * 3$. Cabe destacar que el color elegido para el vértice modificado puede ser uno que ya se haya utilizado

para otros vértices, como también puede ser uno nuevo.

Cada vez que se aplica el *change*, se verifica que la solución sea "válida" (línea 11). Con esto nos referimos a que el coloreo generado no deje dos vértices adyacentes en G con el mismo color y que la solución o el vértice no se encuentre en la lista tabú, según indique el flag correspondiente. En caso de que la solución sea válida, se calcula su impacto, y se compara con el de la mejor solución obtenida hasta el momento, para finalmente devolver únicamente la mejor solución del subconjunto de la vecindad.

La nueva solución creada por `crearVecindad` es añadida a la lista tabú, y si es mejor que la solución actual, entonces se cambia la actual por la nueva. Además, si la lista tabú se llenó (su tamaño es igual a $tamañoMaximo$), se saca el elemento más antiguo. Finalmente, se aumenta en 1 la cantidad de iteraciones realizadas.

Complejidad

Inicializar la solución $coloreoMax$ es $O(n^3)$, ya que esa es la complejidad de ambas heurísticas. Luego calculamos el impacto de $coloreoMax$, que como dijimos anteriormente, consiste en recorrer media matriz de $n * n$ (su triángulo superior), lo cual tiene complejidad $O(n^2)$.

Ya con esto hecho, entramos al ciclo principal. En este se crea la $soluciónActual$, usando la función `crearVecindad`. Esta función realiza $O(n)$ iteraciones, creando una nueva solución en cada una y chequeando que sea válida. Comprobar que el coloreo resultante sea válido se logra en $O(n)$ ya que al haber modificado un solo vértice, basta con recorrer sus adyacentes. Al recorrer la lista tabú, hay dos posibilidades:

- Si la memoria es basada en soluciones, se deben comparar coloreos, es decir, vectores, por lo que la complejidad es $O(n^2)$. Esto surge de que compararar dos vectores tiene complejidad $O(n)$, y acá lo hacemos $O(n)$ veces (una vez para cada elemento de la lista tabú).
- Si la memoria es basada en estructura, alcanza con comparar los primeros elementos de las tuplas en $O(1)$. Esto resulta entonces $O(n)$ debido a la capacidad de la lista tabú.

Dado que el tamaño de la subvecindad elegido es $3*n$, la complejidad de cada iteración es $O(n^2)$ en el caso de memoria elegida por estructura y $O(n^3)$ en el de memoria basada en soluciones. Por más que implementamos un mismo algoritmo para ambos casos, tomamos como dos situaciones diferentes el hecho de basar la memoria de la lista tabú en una opción o en la otra, y por eso analizamos las complejidades por separado.

La complejidad total del algoritmo dependerá de la cantidad de iteraciones definidas como criterio de parada, quedando entonces $O(n^3 + cantIteraciones * n^3) = O(cantIteraciones * n^3)$ en un caso, y $O(n^3 + cantIteraciones * n^2)$ en el otro.

3. Experimentación

La experimentación se divide en 3 secciones:

- Las heurísticas golosas y su desempeño.
- Tabú Search.
- Instancias nuevas y patológicas.

En la segunda sección se analiza la performance del algoritmo usando distintos criterios de corte (la cantidad de iteraciones que se realizan en el ciclo principal de Tabú Search), los cuales son: 50, 500, 1000 y 2500 iteraciones.

3.1. Heurísticas

El objetivo de esta sección es analizar el desempeño de cada heurística con el fin de encontrar diferencias entre ellas, a partir de las cuales se pueda decidir y argumentar cual de ellas es mejor y/o preferible para distintos escenarios.

En particular, lo que nos interesa ver y comparar es la calidad de las soluciones generadas y el tiempo que tarda cada heurística en generarlas.

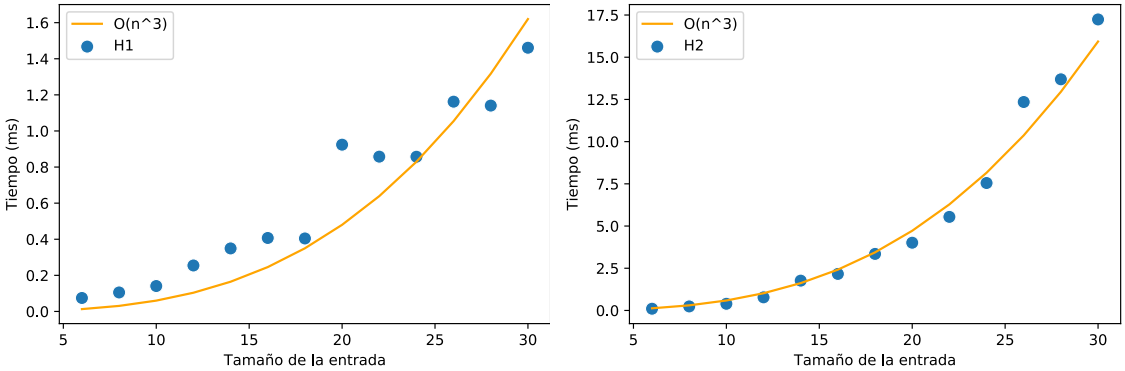
3.1.1. Complejidad

Hipótesis

En primer lugar, verificaremos la complejidad de estos algoritmos contrastándola con la calculada teóricamente. Como se vio en la sección 2, nuestra hipótesis es que ambas tienen complejidad $O(n^3)$.

Experimento

El siguiente gráfico muestra los resultados de correr ambas heurísticas para las instancias provistas, y la comparación del tiempo de ejecución de las mismas contra el esperado por nosotros.



(a) Heurística 1

(b) Heurística 2

Figura 8: Gráficos correspondientes al tiempo de ejecución de las heurísticas.

Conclusión

Como se puede observar en la figura 8, los tiempos de ejecución de ambos algoritmos se corresponden con las complejidades esperadas. En el caso de la heurística 1, se puede ver que la escala es menor que en la segunda. Por este motivo los puntos parecen más dispersos que los de la heurística 2, ya que variaciones pequeñas son más notorias a menor escala. Podemos decir entonces que nuestra hipótesis se verifica.

3.1.2. Comparación de los tiempos y calidad de las soluciones

En esta parte tenemos como objetivo obtener información a partir de los experimentos. Con este fin, analizaremos los resultados de los mismos y los tendremos en cuenta para entender mejor el comportamiento de la heurísticas.

Primero comparamos los tiempos de ejecución de las heurísticas. Esto lo hicimos usando un scatterplot, cuyo eje y representa al tiempo de ejecución y el eje x el tamaño de la instancia.

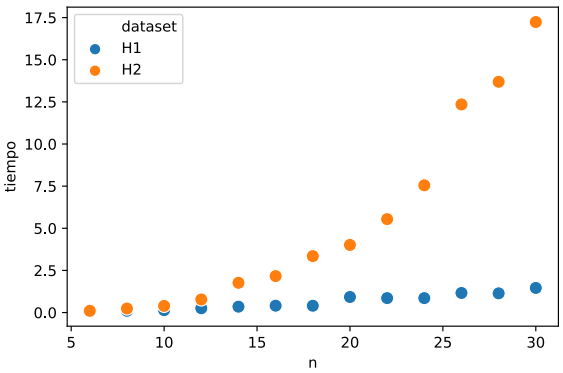


Figura 9: Comparación entre los tiempos de ejecución de las heurísticas.

Como se puede ver, los datapoints que representan a las corridas de la heurística 1 se encuentran siempre por debajo de los de la heurística 2. Esto es indicativo que la heurística 1 tiene un menor tiempo de ejecución. Además, la brecha entre los puntos de la heurística 1 y 2 tiende a acrecentarse a medida que el n crece.

En segundo lugar, comparamos la calidad de las soluciones. Para esto, usamos lo que llamamos GAP, que está definido por la siguiente fórmula:

$$\text{GAP} = \frac{impactoOptimo(G) - impactoConseguido(G)}{impactoOptimo(G)}$$

Un GAP más pequeño es sinónimo de una mejor calidad de respuesta.
Para contrastar la calidad de las soluciones, usamos el heatmap de la figura 10, en donde:

- Cada columna representa una heurística distinta.
- El eje y denota el tamaño de las instancias.
- El color de cada casillero indica la calidad de la solución obtenida.

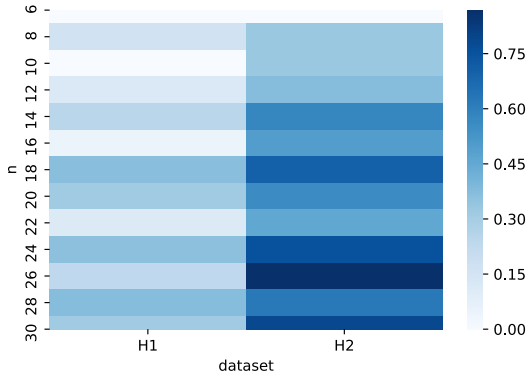


Figura 10: Comparación entre los GAP's de las heurísticas. Cuánto más claro el color, mejor la solución.

En el gráfico se puede observar que la calidad de las soluciones de la heurística 1 es mejor que la de la 2. Esto se nota viendo que para la misma instancia (fila) se tiene un casillero más claro para la heurística 1, lo cual simboliza una mejor calidad de solución (un menor GAP).

En conclusión, se puede argumentar que la heurística 1 consigue mejores resultados y en menos tiempo que la 2. Sin embargo, realizaremos el análisis de la performance de Tabú Search variando entre ellas, ya que no sabemos si un mejor punto de partida afecta positivamente o negativamente al algoritmo.

Es importante destacar que luego de las corridas realizadas, para ciertas de las instancias con las que fueron corridos los algoritmos, al finalizar no lograron llegar al óptimo debido a que tenían situaciones como las mencionadas en los casos patológicos de los mismos. Es el caso de la instancia con $n = 8$ para las dos heurísticas.

3.2. Tabú Search

A continuación estudiaremos el comportamiento del algoritmo de Tabú Search. Veremos su performance al inicializarlo con la heurística 1 o con la 2, variando la cantidad máxima de iteraciones que realizará y modificando el criterio para guardar soluciones en la memoria.

Los nombres de los experimentos realizados se dividen en cuatro:

1. Algoritmo: en este caso será únicamente Búsqueda Tabú (TB)
2. Heurística utilizada (H1 o H2)
3. Memoria utilizada: basada en soluciones exploradas o en estructura (MSE o ME)
4. Cantidad de iteraciones: MBI para muy bajas (50), BI para bajas (500), MI para medias (1000) y AI para altas (2500).

De esta manera, por ejemplo, TB-H1-MSE-BI significa que el experimento es correr Tabú Search inicializado con la Heurística 1, utilizando memoria por soluciones exploradas con 500 iteraciones.

3.2.1. Tiempo de ejecución

A continuación estudiaremos el tiempo de ejecución de Tabú Search al correrlo con las distintas combinaciones mencionadas.

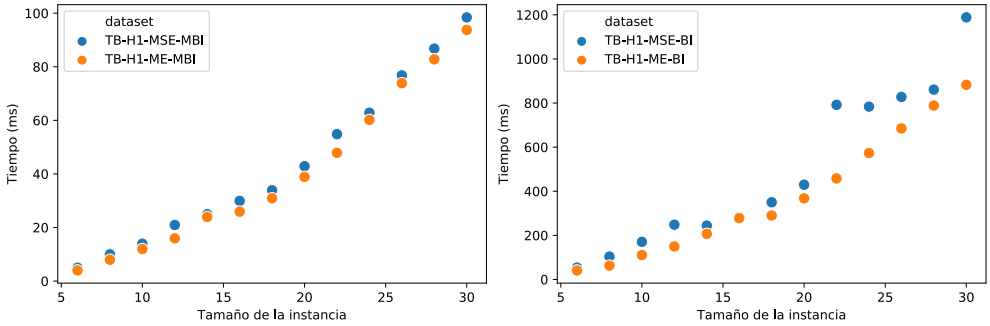
Hipótesis

Nuestra hipótesis para este experimento es que no debería haber diferencia (en el tiempo de ejecución) entre inicializar el algoritmo con la heurística 1 o con la 2. Esta idea se fundamente en el hecho de que ambos algoritmos tienen la misma complejidad, y que además el mayor peso en el tiempo que tarda el algoritmo surge de la parte de búsqueda local, de la cual no forman parte las heurísticas.

Por otro lado, sí esperamos ver una diferencia entre el uso de memoria por soluciones exploradas o por estructura. En este caso, la complejidad entre las dos opciones varía, como bien explicamos en la sección 2.3, y se hacen presentes en la búsqueda local. Por esta razón creemos que llevará más tiempo correr el algoritmo con memoria basada en soluciones que al correrlo con memoria basada en estructura.

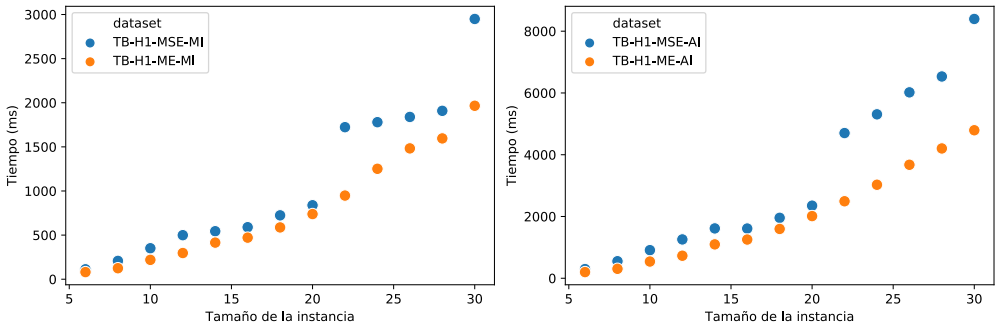
Experimento

Las figuras 11 y 12 muestran el tiempo que tardó la función Tabú Search en devolver una solución al correrla con distintos valores de cantidad máxima de iteraciones. Para cada uno de dichos valores, se corrió el algoritmo con ambas opciones de memoria.



(a) Cantidad de iteraciones muy baja: 50

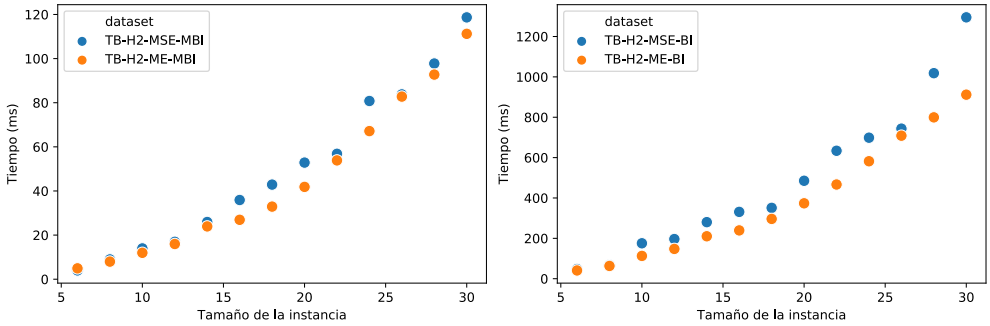
(b) Cantidad de iteraciones baja: 500



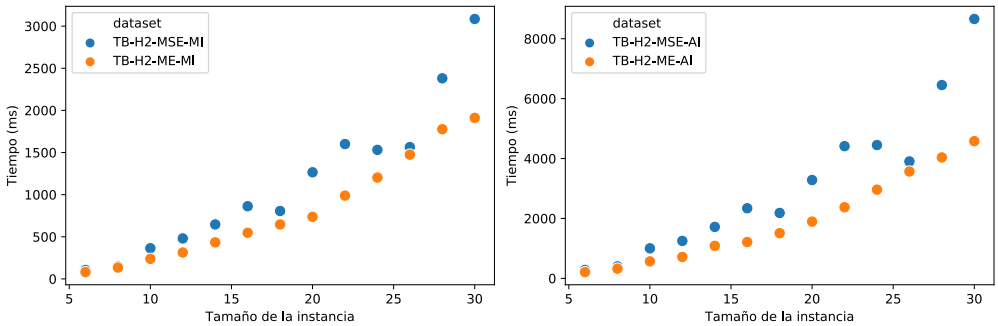
(c) Cantidad de iteraciones mediana: 1000

(d) Cantidad de iteraciones alta: 2500

Figura 11: Tiempo de ejecución en función del tamaño de la entrada para Tabú Search al inicializarlo con la Heurística 1 con diferentes cantidades de iteraciones.



(a) Cantidad de iteraciones muy baja: 50 (b) Cantidad de iteraciones baja: 500



(c) Cantidad de iteraciones mediana: 1000 (d) Cantidad de iteraciones alta: 2500

Figura 12: Tiempo de ejecución en función del tamaño de la entrada para Tabú Search al inicializarlo con la Heurística 2 con diferentes cantidades de iteraciones.

Conclusión

En primer lugar, se verifica lo dicho en la hipótesis: el algoritmo usando MSE tarda más que usando ME. Es importante notar que los tiempos del algoritmo al utilizar una memoria basada en soluciones, al aumentar el tamaño de las instancias, crecen en mayor medida que al utilizar una memoria por estructura. Esta brecha se ve con más claridad en altas iteraciones.

Si en cambio comparamos los gráficos de la figura 11 con los de la 12, podemos observar que usan la misma escala en su eje y, siendo más notable la diferencia en bajas iteraciones y prácticamente idéntica en altas iteraciones, lo cual es congruente con nuestra hipótesis: el tiempo de ejecución es, en la práctica, independiente de la solución inicial, y la única diferencia radica en los (aproximadamente) 20 ms. de diferencia entre la primer heurística y la segunda.

Finalmente, como era esperable, para ambas heurísticas el algoritmo tardó más en los casos de las entradas de mayor tamaño.

3.2.2. Calidad de las soluciones

Con el objetivo de analizar la mejor combinación de los parámetros configurables de la Búsqueda Tabú realizamos un experimento para contrastar la calidad de las soluciones obtenidas con cada una. Como ya dijimos, la calidad la medimos en función del gap.

Hipótesis

Contamos con tres hipótesis, que son las siguientes:

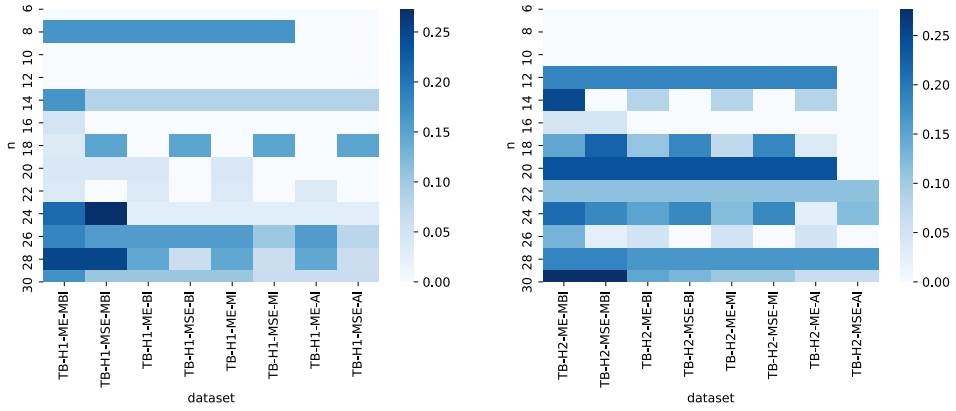
1. Creemos que un mejor punto de partida puede causar un mejor resultado final. Por este motivo, y considerando los resultados obtenidos anteriormente, suponemos que inicializar el algoritmo con la heurística 1 puede hacer que la calidad de las soluciones sean mejores (un menor GAP) con respecto de inicializarlo con la 2.

2. Suponemos que utilizando una memoria basada en soluciones exploradas se obtendrán mejores resultados, dado que es menos restrictiva que la basada en estructura. Con esto queremos decir que será más difícil llegar a la solución óptima utilizando memoria por vértices ya que puede ocurrir que los vértices que es necesario modificar para aumentar el impacto sean parte de la lista tabú por varias iteraciones. Creemos que esta situación será relativamente común debido al tamaño de la lista tabú, que en este caso es de $n/2$.

3. Por último, postulamos que más iteraciones causarán un mejor resultado final. Por lo tanto, esperamos que la variación con más iteraciones sea la que consiga mejores resultados.

Experimento

En este experimento, ejecutamos las 8 posibles combinaciones resultantes del producto cartesiano entre las 2 modalidades de memoria y los 4 criterios de corte. Exponemos dichos resultados en 2 heatmaps, uno para cada heurística. En estos, cada columna representa una combinación memoria-criterio de corte, el eje y indica el tamaño de la entrada y el color de cada cuadrado corresponde con la calidad de la solución obtenida (nuevamente, un color más oscuro indica un mayor GAP).



(a) Heurística 1 (b) Heurística 2

Figura 13: Heatmaps para comparar la calidad de las soluciones obtenidas con diferentes configuraciones del algoritmo Tabú Search.

Conclusión

Con una observación superficial, podemos ver que en la figura 13a predominan colores más claros que en la 13b. Teniendo en cuenta que la primer figura parte de mejores soluciones (pues la heurística 1 genera soluciones de mayor impacto), damos por confirmada nuestra primera hipótesis: en general, comenzar con una mejor solución inicial resulta en un mejor resultado final. Habiendo determinado esto, continuaremos el análisis considerando únicamente la figura 13a.

Luego, al observar las columnas de dos en dos, podemos comparar los resultados entre las dos configuraciones de memoria: ME (por estructura) y MSE (por soluciones exploradas), pues los otros parámetros son constantes. Haciendo este análisis, podemos ver que las soluciones generadas

usando MSE son, en algunos casos, mejores que las generadas por ME. Aunque la diferencia no se ve con tanta claridad, podemos decir que se cumple nuestra hipótesis (en este caso la segunda).

Finalmente, podemos decir que nuestra hipótesis respecto de la cantidad de iteraciones realizadas se cumple parcialmente. Por un lado, se comprueba que cuantas más iteraciones mejor calidad en los resultados, pero esta mejora se ve decrementada paulatinamente. Por ejemplo, de 50 a 500 hay una mejora más marcada que de 1000 a 2500.

Por este motivo, pensamos que la mejor opción teniendo en cuenta la relación costo-calidad es usar 1000 iteraciones (MI) y será la que nosotros usaremos en la sección de test.

Una observación que nos pareció relevante mencionar es que para el caso patológico ($n = 8$ para ambas heurísticas) el algoritmo logró mejorar la solución hasta llegar a la óptima con la heurística 2 en todos los casos pero no así con la 1. Esto indica que el "error" en la solución inicial generada por la heurística 2 es más sencillo de arreglar con la búsqueda local que el de la 1.

3.2.3. Complejidad

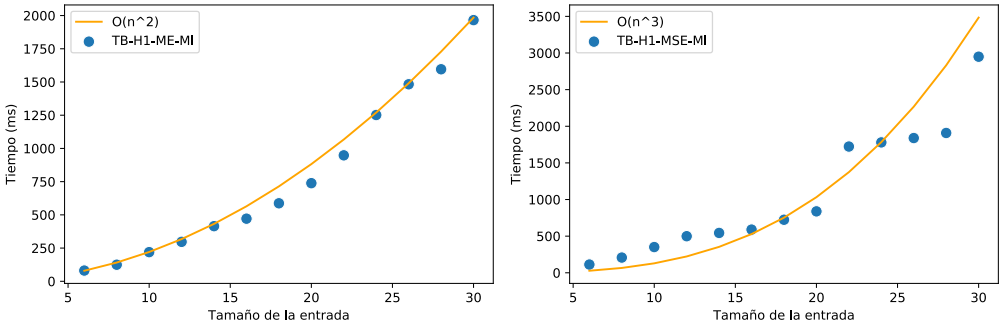
Para corroborar la complejidad del algoritmo en cuestión elegimos correrlo con la combinación que según nuestro criterio es la que mejor se adapta a nuestras condiciones. Esto es, con la que se puede obtener una solución cercana al óptimo sin un costo alto. Dicha combinación es TB-H1-MSE-MI. Sin embargo, dado que la complejidad con memoria por estructura es diferente, también correremos TB-H1-ME-MI.

Hipótesis

Como fue mencionado, esperamos ver que la complejidad sea del orden $O(n^3)$ en el caso de memoria basada en soluciones exploradas, y $O(n^2)$ en el de memoria basada en vértices.

Experimento

Para verificar nuestra hipótesis realizamos un scatterplot, donde el eje x representa el tamaño de la instancia y el y el tiempo de ejecución en milisegundos, y contrastamos los datapoints contra una función cuadrática en la figura 14a y cúbica en la 14b.



(a) Memoria por estructura

(b) Mmoria por solución explorada

Figura 14: Complejidad de Tabú Search comenzando con la solución obtenida por la heurística 1, utilizando 1000 iteraciones.

Conclusión

En la figura 14 se puede ver la complejidad de Tabú Search. En la imagen de la izquierda la linea naranja simboliza a una función cuadrática, y se puede ver que se ajusta bien a los datos.

En la de la derecha la función es cúbica, y se puede ver que se sigue la tendencia, si bien su ajuste no es tan bueno como en la anterior. Creemos que esto se debe a que la diferencia entre ME y MSE es que en la segunda se realizan comparaciones de vectores, y dado que es esperable que la comparación se corte cuando se encuentra un elemento diferente, no todas las veces se llegará a recorrer todo el vector. Esto genera que en ciertos casos la complejidad se acerque más a la función cuadrática. Esto también explica las variaciones y los saltos que se observan tanto en el gráfico 16b, como en los de las figuras 11 y 12.

Por todo lo dicho anteriormente, consideramos que la mejor de las opciones es usar la combinación de Tabú Search inicializado por la heurística 1, usando memoria por soluciones exploradas y con medias iteraciones. Si bien el uso de memoria por soluciones exploradas hace tardar más al algoritmo, creemos que ese tiempo es aceptable considerando los beneficios que trae aparejado.

3.3. Instancias nuevas y patológicas

Para verificar que los parámetros recomendados sirvan para instancias generales del problema, creamos un nuevo conjunto de instancias y pusimos a prueba el algoritmo con ellas. Además de correrlo con los parámetros propuestos, lo corrimos con otras combinaciones con el fin de comparar la calidad de las soluciones y poder determinar si efectivamente es conveniente utilizar la combinación antes mencionada.

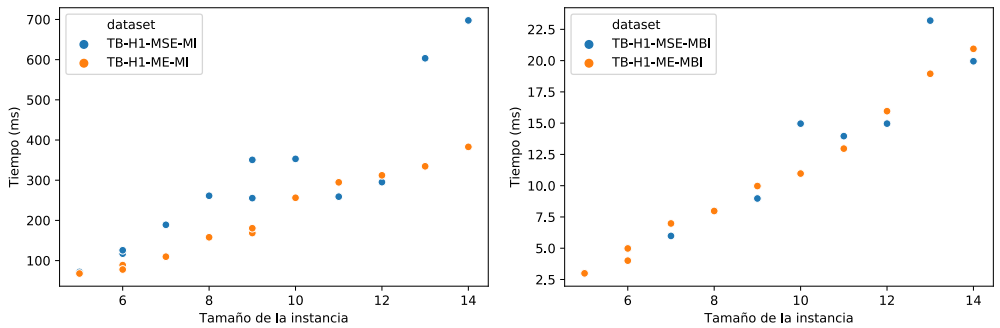
Hipótesis

Nuestra hipótesis en este caso es que la combinación TB-H1-MSE-MI presentará una mejor relación costo-calidad que las demás.

Experimento

Como ya fue mencionado, en este experimento corrimos el algoritmo de Tabú Search con el nuevo conjunto de instancias y con los parámetros que consideramos "óptimos" (heurística 1, memoria basada en soluciones y 1000 iteraciones). También lo corrimos variando la heurística y la memoria utilizada, y utilizando 50 iteraciones.

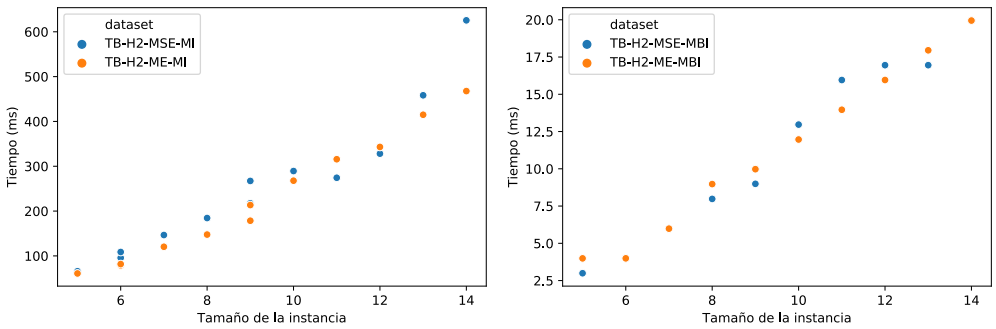
Los resultados demuestran que el algoritmo es generalizable y que consigue buenas soluciones, tal es así que consiguió para cada instancia llegar al impacto óptimo realizando 1000 iteraciones. Esto representa una mejora respecto de la calidad de las soluciones que se consiguen, ya que en 4 casos las 50 iteraciones no alcanzan para encontrar el óptimo. A continuación exponemos los gráficos a partir de los cuales analizaremos el tiempo en función del tamaño de las instancias.



(a) 1000 iteraciones

(b) 50 iteraciones

Figura 15: Tiempo de ejecución de Tabú Search comenzando con la heurística 1 para el conjunto de instancias nuevas.



(a) 1000 iteraciones

(b) 50 iteraciones

Figura 16: Tiempo de ejecución de Tabú Search comenzando con la heurística 2 para el conjunto de instancias nuevas.

Cabe destacar que en este nuevo conjunto se incluyeron instancias donde aparecían casos patológicos para ambas heurísticas, y gracias a la Búsqueda Tabú se logró mejorar la solución hasta llegar a la óptima.

Dichas instancias patológicas son las mismas que utilizamos en las explicaciones de las heurísticas en la sección 2.

Conclusión

Podemos ver en los gráficos que el algoritmo que usa memoria por estructura es (en general) más rápido que el que usa memoria por soluciones exploradas. Hay excepciones a esto, y creemos que estas se deben a que la computadora a la hora de correr experimentos, puede estar haciendo realizando otras tareas, lo cual puede causar una diferencia en los tiempos de ejecución, y en bajas iteraciones esta diferencia se nota más debido a que los tiempos son de escala muy pequeña.

En cuanto a la calidad de las soluciones, todas las generadas por el Tabú Search usando 1000 iteraciones lograron un gap igual a 0. No podemos decir lo mismo de cuando se usaron 50 iteraciones, ya que hubo casos en los que no se logró obtener un impacto óptimo. Esto indica que el aumento de iteraciones afecta positivamente al algoritmo.

Con todo esto dicho, mantenemos nuestra posición de que la combinación TB-H1-MSE-MI es la más conveniente en la mayoría de los casos. Con 1000 iteraciones se logró llegar al óptimo, por lo que no hacen falta más, y si bien en este caso con ME también se logró llegar al óptimo, vimos que MSE suele funcionar mejor.

4. Conclusiones

Durante este trabajo recorrimos y analizamos distintas maneras de resolver el problema PCMI. A continuación detallaremos las conclusiones que sacamos luego de la experimentación presentada.

Primero analizamos las heurísticas, y si bien la heurística 1 consiguió en menos tiempo mejores resultados que la heurística 2, se pudo observar con claridad que ambas están lejos de conseguir el óptimo. Esto resulta notorio viendo el gap de las soluciones obtenidas. Además, en su estudio pudimos determinar lo que llamamos casos patológicos. Estos son los que, por sus características, las heurísticas tienen un rendimiento marcadamente malo en relación a la calidad de las soluciones.

Luego estudiamos la performance de la búsqueda tabú. En este caso tuvimos varios factores a elegir: en primer lugar, la heurística utilizada para tomar como punto de partida para la búsqueda; en segundo lugar, la cantidad de iteraciones que funcionan como criterio de parada del algoritmo; y finalmente el tipo de memoria en el que basar la lista tabú. Después de correr los experimentos pudimos determinar las condiciones bajo las cuales se obtiene una mejor relación costo-calidad.

Estas condiciones son: Generar la solución inicial usando la heurística 1, correr 1000 iteraciones y usar una memoria basada en soluciones exploradas.

Una vez establecidas estas condiciones, volvimos a correr el algoritmo Tabú Search pero con un nuevo conjunto de instancias, con el fin de verificar que lo observado anteriormente se mantiene. Vimos que con la configuración de parámetros óptima se encontró la mejor solución posible con la búsqueda tabú en las instancias nuevas, pero también se llegó a este resultado con otras configuraciones. Nuestra hipótesis es que si fuera el caso de instancias de tamaño aún mayor se destacaría más el hecho de que el algoritmo funciona mejor con las condiciones mencionadas.