1. Heurística de búsqueda local

1.1. Introduccion

¿Qué es una **heurística de búsqueda local**? Una heurística de busqueda local, como su nombre indica, es un **procedimiento que busca, partiendo de una solución inicial s, una que sea mejor** que s a través de "vecinos".

¿Qué son los vecinos de una solución? **Los vecinos de s, denominados $N(s)$** siendo s una solución factible al problema, son aquellos que puedo **obtener realizando una modificación a s**. Esa modificación es ad hoc al problema en sí, y, de hecho, la vecindad de una solución está definida por esa modificación. Por ejemplo, en k-PMP, si la vecindad está definida por tomar un nodo v de un conjunto y colocarlo en otro conjunto, una solución s tiene $s' \in N(s) \Leftrightarrow$ puedo obtener a s' tomando un nodo v de un conjunto y colocarlo en otro. Para esta vecindad, ocurre que $s \in N(s')$ ya que basta con colocar v en su conjunto original.

¿Mejorar en cuánto a qué? Los problemas de **optimización combinatoria**, como el que es analizado en este trabajo práctico, son los cuáles se busca **optimizar una función $f : S \to \mathbb{R}$**, donde S es el conjunto de soluciones factibles para su correspondiente instancia I. La función f para este problema es $f(s) =$ peso de solución s, con $s \in S$ para la instancia I asociada. Por ende, **una solución s' es mejor que una solución $s \Leftrightarrow f(s') < f(s)$**. **La factibilidad de una solución está atada a reglas del problema**. Por ejemplo, en el k-PMP, una solución no puede distribuir menos nodos, en la partición, de los que tiene el grafo de entrada, tampoco podría describir una partición P tal que $|P| > k$, etc.

¿Que nos detiene de **obtener la mejor solución**? **Encontrar la óptima es, efectivamente, resolver el problema**. La necesidad de tener heurísticas, de cualquier tipo, surge de **no conocer algoritmos que resuelvan la problemática en tiempo polinomial**, como es el caso del k-PMP. Por ende, buscamos algoritmos que nos permitan obtener soluciones "buenas" en un tiempo más razonable.

En particular, ¿por qué existen las busquedas locales? Dichas búsquedas proveen un servicio de **mejoramiento a otros algoritmos que generen soluciones**. Supongamos que una instancia I fue puesta a través de un algoritmo heurístico goloso HG. Si esa instancia I era particularmente mala para dicha heurística, la solución obtenida podría estar muy alejada de la óptima, que es lo que queremos tratar de evitar. Una buena idea es someter esa solución a un algoritmo de búsqueda local HBL que atente a mejorarla. Al ser un algoritmo distinto, explora distintas posibilidades que la HG no pudo ver.

¿**Siempre se puede llegar a una solución óptima de esta manera**? No siempre. **Si fuese así, tendríamos un algoritmo que resuelva el problema en tiempos razonables**. Por tiempos razonables nos referimos a tiempos no exponenciales, no podemos asegurar que siempre sean polinomiales ya que no toda heurística tiene una cota polinomial definida. Hasta puede ocurrir que para ciertas instancias sea imposible alcanzar el óptimo, por como fue definida vecindad, por como es la solución, o una mezcla de ambas. Hay heurísticas, denomidadas aproximadas, que tienen cotas de cuán mala puede ser la solución encontrada por dicho algoritmo. Sabiendo eso, si la cota no es muy holgada, combinar esa heurística con una de búsqueda local tendría altas probabilidades de generar muy buenas, o inclusive óptimas, soluciones. Pero, de nuevo, ni siquiera en esos casos podemos asegurar que siempre vamos a encontrar una óptima. **Esta es la razón por la cuál se denominan busquedas *locales*. Buscan la mejor solucion, denominada óptimo local, en un conjunto S' de soluciones tal que $S' \subseteq S$, donde S es el conjunto total de soluciones**, que depende de la solución inicial, de la vecindad planteada y de la instancia.

Habiendo dicho esto, **el procedimiento de la heurística de busqueda local es, dada una solución s, elegir el óptimo local o dentro del conjunto $S' \subseteq S$ que forma la vecindad**. Entonces, esta solución o es alguna una tal que $o \in S'$ y $f(o) \leq f(s')$, $\forall s' \in S'$, donde f es la función a optimizar.

1.2. Busqueda Local 1

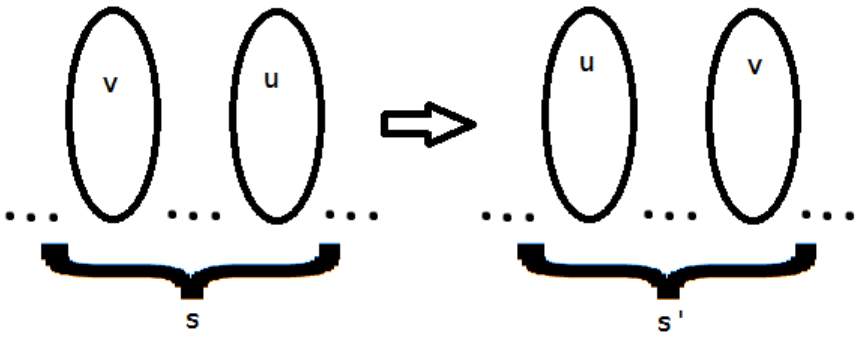
1.2.1. Desarrollo

Explicación de vecindad de la búsqueda local 1

El modo en que este algoritmo de búsqueda local va a operar es **revisando todos los vecinos de una solución y eligiendo el mejor**. Elegimos en el conjunto $S' = \{s\} \cup N(s)$ el óptimo local. Para toda solución s, la vecindad está definida por la siguiente función:

$N1(s)$ = intercambiar un nodo v en un conjunto, por otro nodo u en otro conjunto distinto dentro la partición.

En una solución s genérica:



El tamaño de la vecindad depende de cuántos nodos puedo intercambiar, dejando de lado los nodos que estén en un mismo conjunto, ya que intercambiar 2 nodos dentro de un mismo conjunto deja el peso intrapartición intacto, por ende dejando el peso total intacto. En peor caso, en cuánto a cantidad de intercambios, podría pasar que todos los nodos estén en conjuntos distintos, por ende teniendo $(n-1)$ intercambios posibles para cada vértice, lo cuál genera un total $n*(n-1)$ posibles intercambios. Entonces, **para toda solución s, $|N1(s)| \in O(n*(n-1)) = O(n^2)$**. Esto nos dice que, **como mínimo, en peor caso, la función de complejidad temporal de dicho algoritmo debe ser de orden cuadrático en n ya que esa es la cantidad de vecinos, como máximo, que podría tener una solución** y el algoritmo explora toda la vecindad.

Pseudo código búsqueda local 1

```
1  Solucion BusquedaLocal1(Grafo G = {v_1 ... v_n}, X), int k, Solucion s)
2    Particion particion_actual = particion generada por s
3    Particion mejor_particion = particion_actual
4    para todo nodo v_i con i desde 1 a n:
5      para todo nodo v_j con j desde 1 a n:
6        si v_i esta en un conjunto distinto al v_j:
7          colocar v_i en el conjunto donde se encuentra v_j dentro de la
               particion_actual
8          colocar v_j en el conjunto donde se encontraba v_i dentro de la
               particion_actual
9          si el peso total disminuyo:
10           mejor_particion = particion_actual
11         coloco v_i en su conjunto original dentro de la particion actual
12         coloco v_j en su conjunto original dentro de la particion actual
13         fin si
14       fin si
15     fin para
16   fin para
17
18   Solucion mejor_solucion = solucion asociada a la mejor_particion
19   return mejor_solucion
```

Detalles a notar:

- No hace falta preguntar si $v_i \neq v_j$ ya que si están en conjuntos distintos, lo son.

1.2.2. Complejidad

Comentarios preliminares

- Decidimos utilizar el código fuente para el cálculo de complejidad ya que brinda un cálculo más preciso y sin ambiguedades. Consideramos la línea 1 de un algoritmo como la línea donde, en el código fuente, se encuentra el return type, la declararción y los parámetros de una función.
- Las estructuras utilizadas en el código son set y vector, ambas de la STL de c++.
- Definimos *Grafo* como vector<vector<float>>, *Solución* como vector<int>, *Conjunto* como set<int>y *Partición* como vector<Conjunto>.

La clase BusquedaLocal1

Decidimos colocar la heurística dentro de una clase llamada BusquedaLocal1 que consta de:

Miembros privados de la clase

- *Solución solución_inicial*: Contiene una copia de la solución a ser mejorada por la heurística.
- *Solución mejor_vecino*: Contiene una copia del resultado de la heurística aplicada a la solución_inicial o una copia de la solución_inicial.
- *float peso_mejor_vecino*: Contiene el peso de mejor_vecino si es que se usó previamente la función *resolver*.

Funciones relevantes públicas de la clase

- *Constructor BusquedaLocal1(const Solución& solución)*: Constructor que toma una solución como parametro y crea un objeto de la clase *BusquedaLocal1* copiando la solución pasada como parámetro a *solución_inicia*l y a *mejor_vecino*. Operación que toma $O(n)$ dado que es copiar un vector de ints de tamaño n, y copiar un int tiene complejidad $O(1)$.
- *Solución resolver(const Grafo& grafo, int k, int n)*: Función que aplica la heurística de búsqueda local 1 a *solución_inicial*, almacenando el resultado en *mejor_vecino* y devolviéndolo. Complejidad analizada más abajo. **Esta función corresponde al pseudo código de la busqueda local 1 hecho más arriba.**
- *float getPeso()*: Función que devuelve peso_mejor_vecino. Complejidad $O(1)$.

Complejidad algoritomo de BusquedaLocal1::resolver

- Líneas 2 - 3: 2 copias de objetos *Partición*. Copiar un *Conjunto* tiene complejidad $O(n)$, ya que en peor caso puede contener a todos los vértices, y hay k conjuntos, por ende la complejidad es $O(2(n*k)) = O(n*k)$.
- Líneas 3 - 18: *For* que itera sobre la cantidad de nodos $\Rightarrow O(n * cuerpoFor1) = O(n^4 * k)$.
- $cuerpoFor1$:
  - Líneas 5 - 17: *For* que itera sobre la cantidad de nodos $\Rightarrow O(n*cuerpoFor2) = O(n*n^2*k) = O(n^3*k)$.
  - $cuerpoFor2$:
    - Línea 9: Complejidad de *swapear_nodos* $\in O(log(n))$ (Ver Complejidad Algoritmos Usados).
    - Línea 10: Compejidad de *peso_de_partición* $\in O(n^2 * k)$ (Ver Complejidad Algoritmos Usados).
    - Línea 11: Copia de una *Partición* $\in O(n * k)$.
    - Línea 15: Complejidad de *swapear_nodos* $\in O(log(n))$ (Ver Complejidad Algoritmos Usados).
  - Complejidad $cuerpoFor2 \in O(n^2 * k + n * k + log(n)) = O(n^2 * k)$, por propiedades de la función O.
- Complejidad $cuerpoFor1$ $O(1 + n^2 * k) = O(n^3 * k)$, por propiedades de la función O.
- Línea 19: Copiar un float $O(1)$ y Complejidad de *peso_de_partición* $\in O(n^2 * k)$ (Ver Complejidad Algoritmos Usados) $\Rightarrow \in O(n^2 * k)$.
- Línea 20: Copiar una *Solución* tiene complejidad $O(n)$ y Complejidad de *convertir_particion_en_solucion* $\in O(n * k)$ (Ver Complejidad Algoritmos Usados) $\Rightarrow \in O(n + n * k) = O(n * k)$.
- Línea 21: Devolver una *Solución* por copia tiene complejidad $O(n)$.

Complejidad Total: $O(n * k + n^4 * k) = O(n^4 * k)$ la cuál es pseudo-polinomial en el tamaño de la entrada.

Un pequeño análisis más en detalle de la complejidad nos muestra que lo que habíamos mencionado del tamaño de la vecindad es cierto. $O(n^3 * k) = O((n^2) * (n^2 * k))$, donde $O(n^2)$ es el costo de recorrer los vecinos y $O(n^2 * k)$ es el costo de "armar" y obtener el valor de f, la función a optimizar, de cada vecino.

Explicación y Complejidad Algoritmos Usados

- *void swapear_nodos(int nodo_i, int conjunto_nodo_i, int nodo_j, int conjunto_nodo_j, Partición& partición_actual)*:
  - Función que, dados los parámetros, toma el *nodo_i* y lo coloca en el *conjunto_nodo_j*, y luego toma el *nodo_j* y lo coloca en el *conjunto_nodo_j*.
  - Consta de 2 llamados de función de *set::erase*, que tiene complejidad $O(log(n))$, y 2 llamados de función *set::insert*, que también posee complejidad $O(log(n))$, con n la cantidad de elementos del *set* $\Rightarrow \in O(4 * log(n)) = O(log(n))$.
- *float peso_de_particion(Partición& partición, int k, Grafo& grafo))*:
  - Función que, dado el *Grafo* correspondiente, devuelve el peso total de la *Partición* pasada como parámetro.
  - Consta de un *For* que itera sobre k donde cada iteración tiene un *For* que itera sobre el tamaño del *Conjunto*, denominado n, con un ciclo dentro que itera sobre el tamaño del conjunto, realizando operaciones con costo temporal $O(1) \Rightarrow$ la complejidad de la función es $O(k * n * n * 1) = O(n^2 * k)$.
- *Solución convertir_particion_en_solucion(Partición& partición, int k)*:
  - Función que transforma una *Partición* en una *Solución* válida.
  - Consta de un *For* que itera tantas veces como k que contiene otro *For* que itera sobre n, la cantidad de elementos del *Conjunto*, realizando operaciones de complejidad constante $\Rightarrow \in O(n * k)$.

Costo total de aplicar la heurística sobre una solución

- Tomar el input (no se tiene en cuenta en el cálculo de Complejidad).
- Creación del objeto *BusquedaLocal1* con la solución a mejorar $\in O(n)$.
- Llamado a *BusquedaLocal1::resolver* sobre ese objeto $\in O(n^4 * k)$.

Complejidad Total: $O(n^4 * k)$.

1.3. Busqueda Local 2

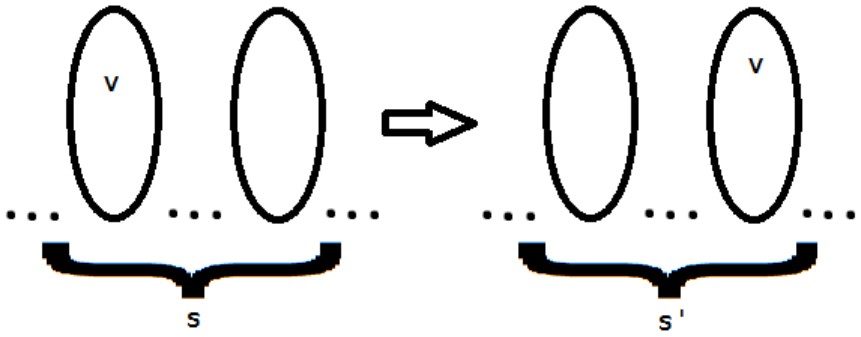
1.3.1. Desarrollo

Explicación de vecindad de la búsqueda local 2

El modo en que este algoritmo de búsqueda local va a operar es **revisando todos los vecinos de una solución y eligiendo el mejor**. Elegimos en el conjunto $S' = \{s\} \cup N(s)$ el óptimo local. Para toda solución s, la vecindad está definida por la siguiente función:

$N2(s)$ = colocar un nodo v en un conjunto distinto al que pertenece dentro de la partición.

En una solución s genérica:



El tamaño de esta vecindad depende de en cuántos conjuntos distintos dentro de la partición puedo colocar a un vértice, y eso es siempre $(k-1)$, todos los conjuntos menos el mismo al que pertenece. Esto es cierto para todo vértice, **entonces para toda solución s se cumple que $|N2(s)| = O(n*(k-1)) = O(n*k)$**. Como para la anterior búsqueda local, se debe cumplir que el **algoritmo, como mínimo, perteneza a $O(n*k)$ dado que el algoritmo revisa toda la vecindad en busca del mejor vecino**.

Pseudo código de la búsqueda local 2

```
1  Solucion BusquedaLocal1(Grafo G = {v_1 ... v_n}, X), int k, Solucion s)
2    Particion particion_actual = particion generada por s
3    Particion mejor_particion = particion_actual
4    para todo nodo v_i con i desde 1 a n:
5      para todo conjunto c_j desde 1 hasta k:
6        si el nodo v_i no pertenece al conjunto c_j
7          quitar v_i de su conjunto actual
8          colocar v_i en el conjunto c_j
9          si el peso total disminuyo:
10           mejor_particion = particion_actual
11         quitar v_i de c_j
12         colocar v_i en su conjunto original correspondiente
13         fin si
14       fin si
15     fin para
16   fin para
17
18   Solucion mejor_solucion = solucion asociada a la mejor_particion
19   return mejor_solucion
```

1.3.2. Complejidad

Comentarios preliminares

- Decidimos utilizar el código fuente para el cálculo de complejidad ya que brinda un cálculo más preciso y sin ambiguedades. Consideramos la línea 1 de un algoritmo como la línea donde, en el código fuente, se encuentra el return type, la declararción y los parámetros de una función.
- Las estructuras utilizadas en el código son set y vector, ambas de la STL de c++.

- Definimos *Grafo* como vector<vector<float>>, *Solución* como vector<int>, *Conjunto* como set<int>y *Partición* como vector<Conjunto>.

La clase BusquedaLocal2

Decidimos colocar la heurística dentro de una clase llamada *BusquedaLocal2* que consta de:

Miembros privados de la clase

- *Solución solución_inicial*: Contiene una copia de la solución a ser mejorada por la heurística.
- *Solución mejor_vecino*: Contiene una copia del resultado de la heurística aplicada a la solución_inicial o una copia de la solución_inicial.
- *float peso_mejor_vecino*: Contiene el peso de mejor_vecino si es que se usó previamente la función *resolver*.

Funciones relevantes públicas de la clase

- *Constructor BusquedaLocal1(const Solución& solución)*: Constructor que toma una solución como parametro y crea un objeto de la clase *BusquedaLocal1* copiando la solución pasada como parámetro a *solución_inicia*l y a *mejor_vecino*. Operación que toma $O(n)$ dado que es copiar un vector de ints de tamaño n, y copiar un int tiene complejidad $O(1)$.
- *Solución resolver(const Grafo& grafo, int k, int n)*: Función que aplica la heurística de búsqueda local 1 a *solución_inicial*, almacenando el resultado en *mejor_vecino* y devolviéndolo. Complejidad analizada más abajo. **Esta función corresponde al pseudo código de la busqueda local 2 hecho más arriba.**
- *float getPeso()*: Función que devuelve peso_mejor_vecino. Complejidad $O(1)$.

Complejidad algoritomo de BusquedaLocal2::resolver

- Líneas 2 - 3: Líneas 2 - 3: 2 copias de objetos *Partición*. Copiar un *Conjunto* tiene complejidad $O(n)$, ya que en peor caso puede contener a todos los vértices, y hay k conjuntos, por ende la complejidad es $O(2(n*k)) = O(n*k)$.
- Líneas 3 - 18: *For* que itera sobre la cantidad de nodos $\Rightarrow O(n * cuerpoFor1) = O(n * (n^2 * k^2)) = O(n^3 * k^2)$
- *cuerpoFor1*:
  - Líneas 4 - 16: *For* que itera sobre la cantidad de conjuntos $\Rightarrow O(k * cuerpoFor2) = O(k * (n^2 * k)) = O(n^2 * k^2)$
  - *cuerpoFor2*
    - Línea 8: Complejidad de operación *mover_nodos* $\in O(log(n))$ (Ver Complejidad Algoritmos Usados).
    - Línea 9: Compejidad de *peso_de_partición* $\in O(n^2 * k)$ (Ver Complejidad Algoritmos Usados).
    - Línea 10: Copia de una *Partición* $\in O(n * k)$.
    - Línea 14: Complejidad de operación *mover_nodos* $\in O(log(n))$ (Ver Complejidad Algoritmos Usados).
  - Complejidad *cuerpoFor2* $\in O(n^2 * k + n * k + 2log(n)) = O(n^2 * k)$.
- Complejidad *cuerpoFor1* $\in O(n^2 * k^2)$.
- Línea 20: Copiar un float $O(1)$ y Complejidad de *peso_de_partición* $\in O(n^2 * k)$ (Ver Complejidad Algoritmos Usados) $\Rightarrow \in O(n^2 * k)$.
- Línea 21: Copiar una *Solución* tiene complejidad $O(n)$ y Complejidad de *convertir_particion_en_solucion* $\in O(n * k)$ (Ver Complejidad Algoritmos Usados) $\Rightarrow \in O(n + n * k) = O(n * k)$.
- Línea 22: Devolver una *Solución* por copia tiene complejidad $O(n)$.

Complejidad Total: $O(n^3 * k^2)$.

Un pequeño análisis más en detalle de la complejidad nos muestra que lo que habíamos mencionado del tamaño de la vecindad es cierto. $O(n^2 * k^2) = O((n * k) * (n^2 * k))$, donde $O(n * 2)$ es el costo de recorrer los vecinos y $O(n^2 * k)$ es el costo de "armar" y obtener el valor de f, la función a optimizar, cada vecino.

Explicación y Complejidad Algoritmos Usados

- *void mover_nodos(int nodo_i, int conjunto_nodo_i, int conjunto_j, Partición& partición_actual)*:
  - Función que, dados los parámetros, toma el *nodo_i* y lo coloca en el *conjunto_j*, y luego quita el *nodo_i* del *conjunto_nodo_i*.

- Consta de 1 llamado a la función de *set::erase*, que tiene complejidad $O(log(n))$, y 1 llamado a la función *set::insert*, que también posee complejidad $O(log(n))$, con n la cantidad de elementos del *set* $\Rightarrow \in O(2 * log(n)) = O(log(n))$.

- *float peso_de_particion(Partición& partición, int k, Grafo& grafo))*:
  - Función que, dado el *Grafo* correspondiente, devuelve el peso total de la *Partición* pasada como parámetro.
  - Consta de un *For* que itera sobre k donde cada iteración tiene un *For* que itera sobre el tamaño del *Conjunto*, denominado n, con un ciclo dentro que itera sobre el tamaño del conjunto, realizando operaciones con costo temporal $O(1) \Rightarrow$ la complejidad de la función es $O(k * n * n * 1) = O(n^2 * k)$.

- *Solución convertir_particion_en_solucion(Partición& partición, int k*):
  - Función que transforma una *Partición* en una *Solución* válida.
  - Consta de un *For* que itera tantas veces como k que contiene otro *For* que itera sobre n, la cantidad de elementos del *Conjunto*, realizando operaciones de complejidad constante $\Rightarrow \in O(n * k)$.

Costo total de aplicar la heurística sobre una solución

- Tomar el input (no se tiene en cuenta en el cálculo de Complejidad).
- Creación del objeto *BusquedaLocal1* con la solución a mejorar $\in O(n)$.
- Llamado a *BusquedaLocal1::resolver* sobre ese objeto $\in O(n^3 * k^2)$.

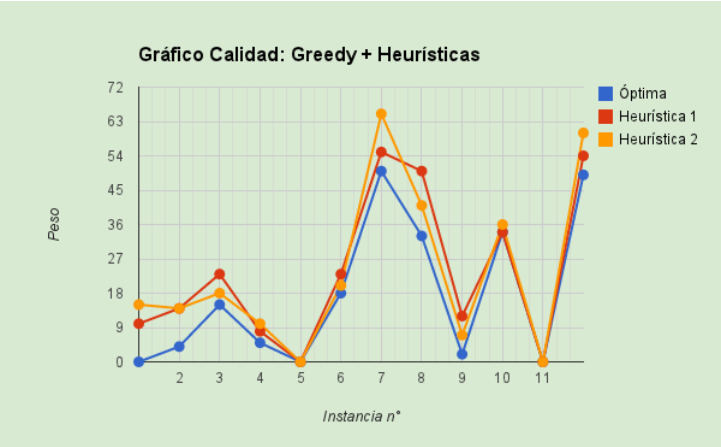
Complejidad Total: $O(n^3 * k^2)$.

1.4. Experimentacion

Análisis de Calidad

Medición de calidad con Solución Greedy

Para esta medición, **generamos instancias con $k < n$** ya que el algoritmo greedy devuelve una respuesta exacta en caso contrario. Si $k \geq n$, basta con colocar un nodo por conjunto y obtenemos una solución óptima con peso 0. Veamos el gráfico de dichas mediciones:

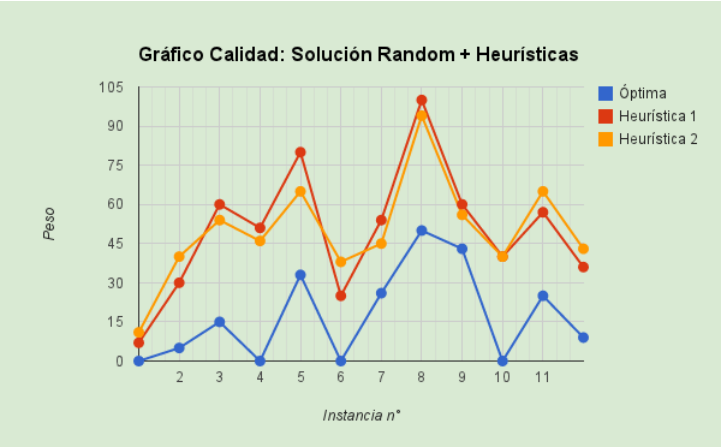


En estas condiciones, ninguno es claro vencedor. Esto puede deberse al hecho de que las vecindades son "similares" en cuánto a cómo son sus vecinos, realizar un intercambio de nodos entre 2 conjuntos distintos es lo mismo que tomar un nodo, colocarlo en un conjunto y luego tomar otro nodo distinto y colocarlo en el conjunto donde se encontraba el vértice anteriormente mencionado.

Lo que si se puede apreciar del gráfico es que las soluciones obtenidas por las heurísticas no son excesivamente lejanas a las óptimas. Dicho fenómeno no ocurre espontáneamente, **mientras mejor es la solución inical, la solución encontrada por la heurística también lo va a ser.** Formalmente, la respuesta de una heurística, a lo sumo, es igual de buena a la solución inicial, nunca peor, sino caería fuera de la definición de heurística. En consecuencia, tener buenas soluciones iniciales favorece que la heurística acabe en una buena solución final.

Medición de calidad con Solución Aleatoria

El siguiente gráfico muestra **instancias sin ningún tipo de particularidad**. La diferencia con el caso anterior es que, inclusive con $k \geq n$, como la solución es generada al azar, no necesariamente la solución obtenida es óptima. Por eso, en estas condiciones, es relevante analizar todos los casos, y no restringirlo a $k < n$.



Como en el análisis de calidad anterior, **ninguno de los 2 se destaca claramente.** Por otra parte, es claro que **las heurísticas entregaron soluciones peores que las del análisis previo**. Las soluciones aleatorias pueden ser excesivamente malas, inclusives las peores, y eso se ve reflejado en los resultados obtenidos. Es fácil ver que, para estas heurísticas, **siempre conviene tener inicialmente una solución púlida.** Por esto es que, en la introducción a esta sección del trabajo práctico, mencionamos que las **heurísticas de búsqueda local brindan un servicio de complemento a otros algoritmos, incluyendo heurísticas, de generación de soluciones.**

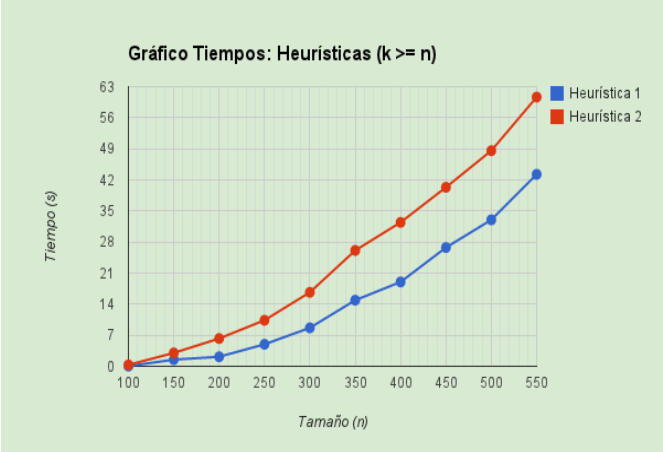
Análisis de Tiempos

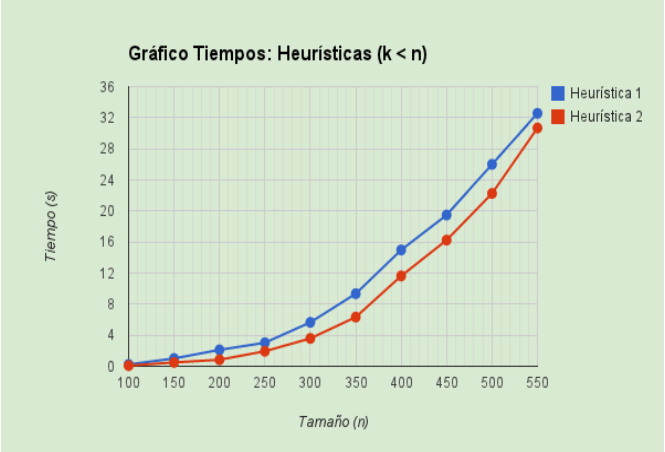
Metodología de medición

Para cada tamaño n, generamos 100 instancias. Luego, para cada instancia, medimos su tiempo de ejecución 50 veces, quedándonos con la ejecución que menor tiempo tomo. Finalmente, obtuvimos un promedio de esos valores y el resultado de dicha operación es la que fue colocada en los gráficos. Tomamos el mínimo de cada instancia debido a que el procesador atiende varios procesos en simultáneo y, por ende, es díficil obtener un valor fiel de cuánto realmente toma ejecutar el algoritmo sobre la instancia actual. Tomando el mínimo eliminamos la mayor cantidad de tiempo gastado en operaciones fuera de la ejecución de nuestro algoritmo.

Medición de tiempos

Ya que ambos algoritmos dependen, en cuánto a complejidad temporal, de k, una variable que no tiene relación directa con el tamaño de la entrada n, decidimos realizar un análisis de tiempos variando el k, esto fue lo que obtuvimos:





Los resultados son coherentes al análisis de complejidad hecho anteriormente, el algoritmo 2 está más atado al k que su contraparte. Pero, conceptualmente, **¿Cuál es la razón?** Ya habíamos mencionado **el tamaño de la vecindad de cada heurística: para la 1 el tamaño $\in O(n^2)$ y para la 2da $\in O(n*k)$**. Si $k < n$, la vecindad de la segunda heurística tiene una cota más ajustada de $O(n^2)$, por eso las mediciones son similares para dicho caso. En cambio, si $k \geq n$, ninguna se tiene una cota mejor. Si bien $O(n^2) = O(n^k)$, para este caso, eso no significa que la vecindad de la heurística 1 depende de k, nada más es una cota más "grosera" que no refleja el comportamiento adecuadamente. Veamos el siguiente caso: $n = 5$ y $k = 100$, está claro que la vecindad 2 va a ser más numerosa que la 1, independientemente de los pesos de las aristas y de la densidad del grafo. Inclusive, **uno podría tener un k tan alejado de n como quisiera y así obtener costos temporales potencialmente caros**.

De todas formas, poseemos un algoritmo polinomial que otorga una solución óptima para casos tales que $k \geq n$ y esa información no es costosa de obtener, se pregunta si eso ocurrre, en $O(1)$, inmediatamente luego de tomar el input. Por lo cuál no habría que preocuparse demasiado por este tipo de casos.