

Algoritmos y Estructura de Datos III

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Práctico 3

Grupo 1

Integrante	LU	Correo electrónico
Hernandez, Nicolas	122/13	nicoh22@hotmail.com
Kapobel, Rodrigo	695/12	rok_35@live.com.ar
Rey, Esteban	657/10	estebanlucianorey@gmail.com
Tripodi, Guido	843/10	guido.tripodi@hotmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Contents

1	Explicación del problema	3
2	Informe de correcciones	4
2.0.1	Ejercicio 1	4
2.0.2	Ejercicio 2	4
2.0.3	Ejercicio 3	4
2.0.4	Ejercicio 4	4
2.0.5	Ejercicio 5	4
3	Ejercicio 1	5
3.1	Explicación de resolución del problema	5
3.2	Pseudocódigo	5
3.2.1	Estructura interna para chequear elecciones y estados ocurridos	8
3.3	Análisis de complejidades	8
3.4	Experimentos y conclusiones	8
3.4.1	3.4.1 1.5	8
4	Ejercicio 2	14
4.1	Explicación de resolución del problema	14
4.2	Pseudocódigo	15
4.2.1	Nota 1	18
4.2.2	Estructura interna para chequear elecciones y estados ocurridos	18
4.3	Análisis de complejidades	18
4.4	Experimentos y conclusiones	18
4.4.1	4.4.1 Familias random y Gimnasios por grupos	27
5	Ejercicio 3	33
5.1	Explicación de resolución del problema	33
5.2	Pseudocódigo	36
5.3	Experimentos y conclusiones	40
5.3.1	5.3.1 2.5	40
6	Ejercicio 4	47
6.1	Explicación de resolución del problema	47
6.2	Pseudocódigo	48
6.3	Análisis de complejidades	51
6.4	Experimentos y conclusiones	51
6.4.1	6.4.1 2.5	51
7	Ejercicio 5	70
7.1	Descripción de nuevas instancias y experimentación	70
7.1.1	7.1.1 Comparaciones de tiempo entre heurísticas	71
7.1.2	7.1.2 Comparaciones entre los errores	74
7.1.3	7.1.3 Conclusiones	75
8	Aclaraciones	76
8.1	8.1 Aclaraciones para correr las implementaciones	76

1 Explicación del problema

A mitad de año se realizó un lanzamiento en Argentina de un juego conocido como *Pokemon Go*. Dicho juego fue noticia en varios medios de comunicación ya que se veía a muchas personas persiguiendo los denominados **pokemones**.

En este problema, ayudaremos a Brian, un muchacho que desea convertirse en maestro Pokémón derrotando todos los **gimnasios** que presenta el juego. Para aclarar, cada gimnasio presenta un conjunto de pokemones que deben ser vencidos mediante batallas entre pokemones para poder tomar control de dicho gimnasio. Cuando se produce una batalla exitosa, el gimnasio se va debilitando, por lo tanto cuando el gimnasio se debilita lo suficiente, el mismo es vencido.

Brian elegirá su conjunto de pokemones acorde lo cual denominaremos equipo, para disputar las batallas en cada uno de los gimnasios y así convertirse en **Maestro Pokémón**.

Para poder realizar múltiples batallas y llegar a debilitar lo suficiente a un gimnasio para así, ser vencido, son necesarias las denominadas *pociones*, las cuales sirven para recuperar a los pokemones y librarse de varias batallas. Dichas pociones se conseguirán en lo denominado **poke paradas**, las cuales al pasar por las mismas otorgarán 3 pociones cada una. Y, además, cada poke parada podrá ser utilizada una única vez.

Brian, tendrá una mochila donde guardará las pociones, dicha mochila tendrá una capacidad máxima para albergar las mismas.

En cada gimnasio, es posible saber, de antemano, cuantas pociones se necesitarán para vencerlo.

Dada dichas situaciones, Brian nos solicitó su ayuda, la cual consistirá en obtener el camino mínimo a recorrer para vencer a todos los gimnasios teniendo todas las pociones necesarias para ganar en cada uno de los gimnasios y así convertirse en **Maestro Pokemon** en la menor cantidad de tiempo posible.

2 Informe de correcciones

2.0.1 Ejercicio 1

- Se realizó una explicación de la implementación del algoritmo adicional al pseudocódigo
- Se quitaron las podas al código implementado y se realizó una nueva medición de tiempos con la respectiva comparación del algoritmo con y sin podas.

2.0.2 Ejercicio 2

- Se realizó una explicación de la implementación del algoritmo adicional al pseudocódigo
- Se creó un nuevo conjunto de instancias random y se introdujo la familia **grupos de gimnasios**
- Se realizaron nuevas experimentaciones con un total de 470 elementos (pokeparadas + gimnasios) obteniendo nuevas conclusiones

2.0.3 Ejercicio 3

- Se realizó una explicación de la implementación del algoritmo adicional al pseudocódigo
- Se realizaron nuevas experimentaciones con un total de 470 elementos obteniendo nuevas conclusiones

2.0.4 Ejercicio 4

- Se realizó una explicación de la implementación del algoritmo adicional al pseudocódigo
- Se realizaron nuevas experimentaciones con un total de 470 elementos obteniendo nuevas conclusiones

2.0.5 Ejercicio 5

- Se creó un nuevo conjunto de instancias random y la familia **grupos de gimnasios**
- Se realizaron nuevas experimentaciones con un total de 470 elementos obteniendo nuevas conclusiones

3 Ejercicio 1

3.1 Explicación de resolución del problema

Siendo el objetivo del maestro pokemon ganar todos los gimnasios recorriendo la menor distancia posible definimos como posible solución a una secuencia de lugares (gimnasios o pokeparadas) en donde sea posible pasar de un lugar al siguiente bajo las restricciones impuestas en el problema. La solución buscada propiamente dicha es aquella que forme el camino de menor longitud dentro de todas las soluciones posibles halladas.

Para esto se resuelve aplicar la técnica de backtracking sobre el conjunto total de caminos posibles. Las podas elegidas son las siguientes:

- Si la suma de pociones de todas las pokeparadas es inferior a la suma de pociones necesarias para derrotar a todos los gimnasios, entonces no hay solución.
- No se pueden repetir lugares ya visitados: ésta es una restricción por consigna.
- No se puede visitar un gimnasio sin la cantidad necesaria de pociones: de no contemplar esta poda se analizan caminos donde se pierde en gimnasios, los cuales por la primera restriccion, quedan como perdidos, y no generan solución.
- No se recorren pokeparadas si la mochila está llena: el objetivo de una pokeparada es llenar la mochila de pociones. Al estar llena la mochila, se desperdicia la pokeparada ya que no se la podrá volver a visitar. En adición, visitar una pokeparada innecesariamente, aumenta obligatoriamente la distancia recorrida por el maestro pokemon, ya que en vez de estar tomando un camino directo entre 2 lugares, accede a la pokeparada, aumentando así la distancia euclidea.
- Los caminos se continúan construyendo siempre y cuando no superen la distancia mínima lograda anteriormente por alguna solución previa.
- Si algún gimnasio requiere mayor cantidad de pociones que las que caben en la mochila, entonces no hay solución.

3.2 Pseudocódigo

Nuestro algoritmo cuenta con 3 secciones marcadas. En la cual se chequean las podas sin solución, en la segunda se consigue el camino, y en la última donde se devuelve dicho camino con la cantidad de nodos y distancia recorrida.

En detalle, para nuestra primer etapa, desarrollamos un ciclo para poder ver el poder que posee cada uno de los gimnasios, chequeando si la capacidad de la mochila soporta el poder de los mismos. A su vez, se va sumando el poder de cada gimnasio para luego chequear si la cantidad de pokeparadas alcanza para ganar en todos los gimnasios.

Luego, en la segunda sección, se realiza el backtraking propiamente dicho, en donde un ciclo itera hasta obtener el camino con distancia recorrida óptima. Dicho camino es armado tomando ciertas elecciones posibles y en caso que se encuentre algún camino mejor se vuelve para atrás en las elecciones necesarias para tomar otro camino donde la distancia recorrida sea menor. Esto se realizará hasta obtener el camino de distancia mínima. Por último, nuestra tercer sección devolverá el camino obtenido con la distancia recorrida y la cantidad de nodos del camino.

Se muestra el pseudocódigo que ejemplifica lo comentado:

función *EJ1()*

```

    crear cola decisiones para guardar elecciones
    eleccion contiene un id, posicion de la misma y distancia desde el punto anterior hasta el
    como tambien la cantidad de pociones hasta el momento
    sePudoDeshacerEleccion ← verdadero
    minimo ← -1
    Mientras sePudoDeshacerEleccion hacer
        si lesGaneATodos(MaestroPokemon) ∧ tiempo(MaestroPokemon) < minimo entonces
            guarda: O(1)
            minimo ← tiempo(MaestroPokemon)
            caminoRecorrido ← caminoRecorrido(MaestroPokemon)
        fin si
        eleccion ← eleccionPosible(MaestroPokemon)
        si ∃ eleccion entonces
            aplicarElección(eleccion, MaestroPokemon)
        fin si
        de lo contrario
            sePudoDeshacerEleccion ← deshacerEleccion(MaestroPokemon)
        fin si
    fin ciclo
    devolver minimo, tamaño(caminoRecorrido) y caminoRecorrido
fin función

```

total: $O((n + m)!)$

función *aplicarElección(eleccion, maestroPokemon)*

```

    encolar en decisiones la elección tomada
    actualizar estado del sistema en base a la decisión tomada:
    si tipo(eleccion) == GIMNASIO entonces
        guarda: O(1)
        decrementar maestroPokemon.cantidadGymFaltantes en 1
        maestroPokemon.cantidadPociones ← pociones(maestroPokemon) -
            pocionesNecesarias(eleccion)
    fin si
    de lo contrario
        maestroPokemon.cantidadPociones ← pociones(maestroPokemon) + 3
    fin si
    sumar a maestroPokemon.tiempo tiempo(eleccion)
    guardar estado como efectuado
    actualizar elecciónActual con la nueva elección
fin función

```

Complejidad total: $O(1)$

```

función deshacerUltimaElección(maestroPokemon)
    si hay decisiones para desencolar entonces
        desencolar de decisiones una elección
        actualizar estado del sistema en base a la decisión desencolada: O(1)
        si tipo(elección) == GIMNASIO entonces O(1)
            incrementar maestroPokemon.cantidadGymFaltantes en 1 guarda: O(1)
            maestroPokemon.cantidadPociones ← pociones(maestroPokemon) +
            pocionesNecesarias(elección) O(1)
        fin si
        de lo contrario
            maestroPokemon.cantidadPociones ← pociones(maestroPokemon) - 3 O(1)
        fin si
            restar a maestroPokemon.tiempo tiempo(elección) O(1)
            actualizar elecciónActual con la elección anterior a la que acabamos de sacar O(1)
            devolver verdadero O(1)
        fin si
        de lo contrario
            devolver falso O(1)
        fin si
    fin función

```

Complejidad total: $O(1)$

```

función elecciónPosible(maestroPokemon)
    Mientras Hay elecciones disponibles hacer Ciclo:  $O(n+m)$ 
        si elección.tipo == POKEPARADA entonces
            guardar maestroPokemon.pociones = maestroPokemon.capacidadMochila O(1)
            si maestroPokemon.pociones == maestroPokemon.capacidadMochila entonces guarda: O(1)
                recalcularElección en base a ultima posición correcta O(1)
            fin si
            de lo contrario
                devolver elección O(1)
            fin si
        fin si
        de lo contrario
            si maestroPokemon.pociones < pocionesNecesarias(elección) entonces guarda: O(1)
                recalcularElección en base a ultima posición correcta O(1)
            fin si
            de lo contrario
                devolver elección O(1)
            fin si
        fin si
    fin ciclo
fin función

```

Complejidad total: $O(n + m)$

3.2.1 Estructura interna para chequear elecciones y estados ocurridos

Se trabajó con dos estructuras internas denominadas *MaestroPokemon* y *Elección*, donde la primera tiene la información sobre la cantidad de gimnasios y pokeparadas, la cantidad de gimnasios ya derrotados, la posición actual, la cantidad de posiciones que posee en cada momento como también la capacidad máxima de la mochila, y dos arreglos los cuales poseen todos los destinos posibles visitados o no y otra con solo los visitados. Mientras que nuestra segunda estructura, como el nombre lo indica, nos otorga toda la información necesaria para poder realizar la elección, como es el id de la misma para no ser repetida, la posición en donde se encuentra, la cantidad de pociones necesarias en caso de ser un gimnasio y la distancia para llegar a dicha posición desde cierto punto.

3.3 Análisis de complejidades

Considerando el arbol de todas las decisiones posibles para formar un camino, la cantidad de pasos que efectúa el algoritmo se ve reducida por las podas que sean aplicadas:

Dado que en un camino no pueden repetirse lugares, la longitud está acotada por $n + m$, que representaría al camino que pasa una vez por cada lugar. La cantidad de caminos de este estilo está dada por la permutación de $n + m$ (pokeparadas + gimnasios) elementos en $n + m$ lugares, es decir $O((n + m)!)$.

La poda de distancia actúa en la medida del orden en que se evalúan los caminos. En el caso en que el orden fuese decreciente y se evalúen los caminos con mayor longitud primero, entonces la poda nunca se efectuará.

La poda de cantidad de pociones necesarias elimina todos los casos sin solución en $O(n)$ que es lo que se demora en recorrer los gimnasios, aportandonos de esta forma el mejor caso para nuestro algoritmo ($\Omega(n)$).

La poda que evita ir a una poke-parada de tener todas las pociones posibles actuará de la mejor forma en el caso en que se tenga una mayor cantidad de poke paradas que de gimnasios, evitando caminos innecesariamente largos a través de sucesivas pokeparadas. En el caso en que haya muchos gimnasios y pocas pokeparadas, la poda de acceder a gimnasios sólo si se tiene la cantidad necesaria de pociones recortará todos los caminos que recorran innecesariamente gimnasios, sin poder ganar.

Se puede observar que la poda que evita ir a una pokeparada de tener la mochila de capacidad k completa nos indica que entre cada viaje entre gimnasios, en el peor de los casos se visitan $\lceil \frac{k}{3} \rceil$ pokeparadas, lo que nos permite acotar a la complejidad por $O((\lceil \frac{k}{3} \rceil n + n)!)$ en el caso en que $m \geq \frac{k}{3}n$ $O(\min\{(n + m)!, (\frac{k}{3}n + n)!\}) \in O((n + m)!)$. Dado que esto se repite una vez por lugar entonces quedaría $O((n + m) * (n + m)!)$

3.4 Experimentos y conclusiones

3.4.1 Testeo y performance del algoritmo

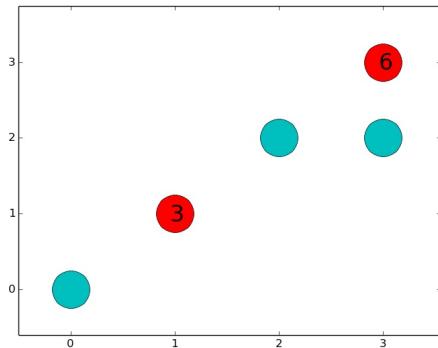
Para estudiar el comportamiento de nuestros algoritmos con distintos tipos de entradas, se generaron 7 familias, cada una con distintas características, las cuales nos permitirán evaluar las podas y estrategias tomadas en los algoritmos. Las mismas son las siguientes:

1. **Familia 1:** Sin solución. La capacidad de la mochila no es apta para ganar ciertos gimnasios
2. **Familia 2:** Sin solución. La cantidad de pokeparadas no es suficiente para ganar todos los gimnasios
3. **Familia 3:** Con solución. Ningún gimnasio necesita pociones para ser vencido.

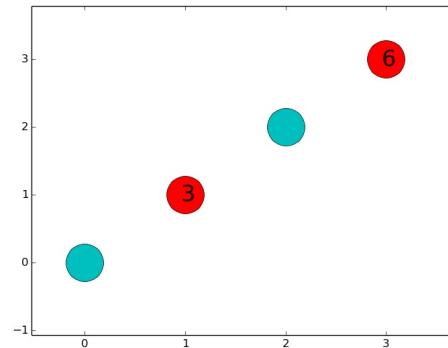
4. **Familia 4:** Con solución. Algún gimnasio necesita pociones para ser vencido y las pociones disponibles son suficientes.
5. **Familia 5:** Con solución. La solución corresponde a ordenar los elementos recibidos por distancia.
6. **Familia 6:** Con solución. Entrada fuera de orden, caso opuesto a la familia 5.
7. **Familia 7:** Con solución. Se busca agrupar los elementos de forma que haya muchas soluciones cercanas a la óptima. Esto se logra formando anillos concéntricos de gimnasios y pokeparadas.

Ejemplificaciones de las familias

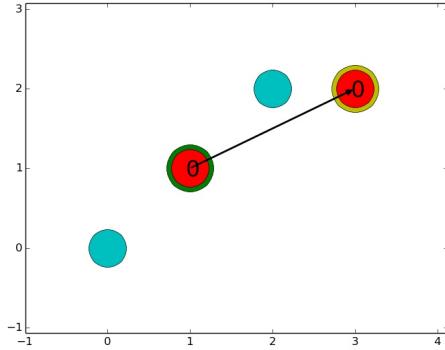
Para las representaciones de los casos se utilizan las siguientes referencias:



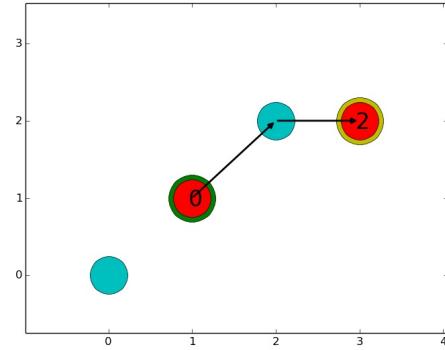
(a) Familia 1: $K=3$ (K = cantidad pokeparadas),



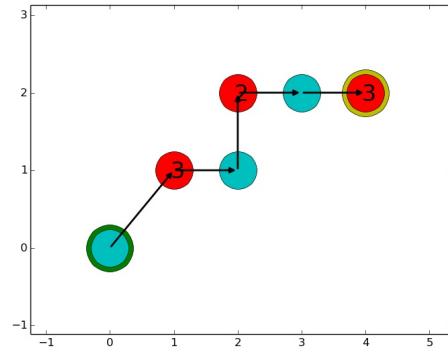
(b) Familia 2: No hay suficientes pociones



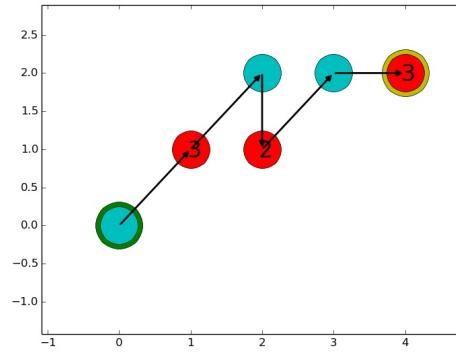
(a) Familia 3: Gimnasios de 0 pociones



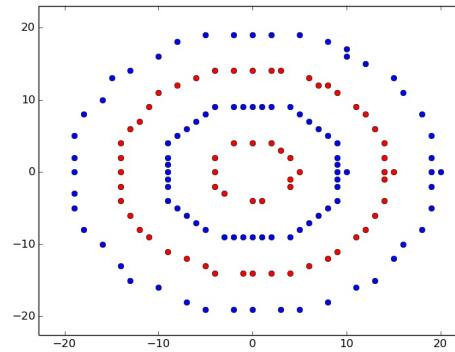
(b) Familia 4: Caso normal



(a) Familia 5: Solución ordenada



(b) Familia 6: Solución desordenada



(c) Familia 7: Anillos

Evaluación del algoritmo sobre las familias

Veremos entonces que es lo que sucede al ir variando los parámetros de entrada para cada familia a medida que N y M (pokeparadas y gimnasios) crecen linealmente.

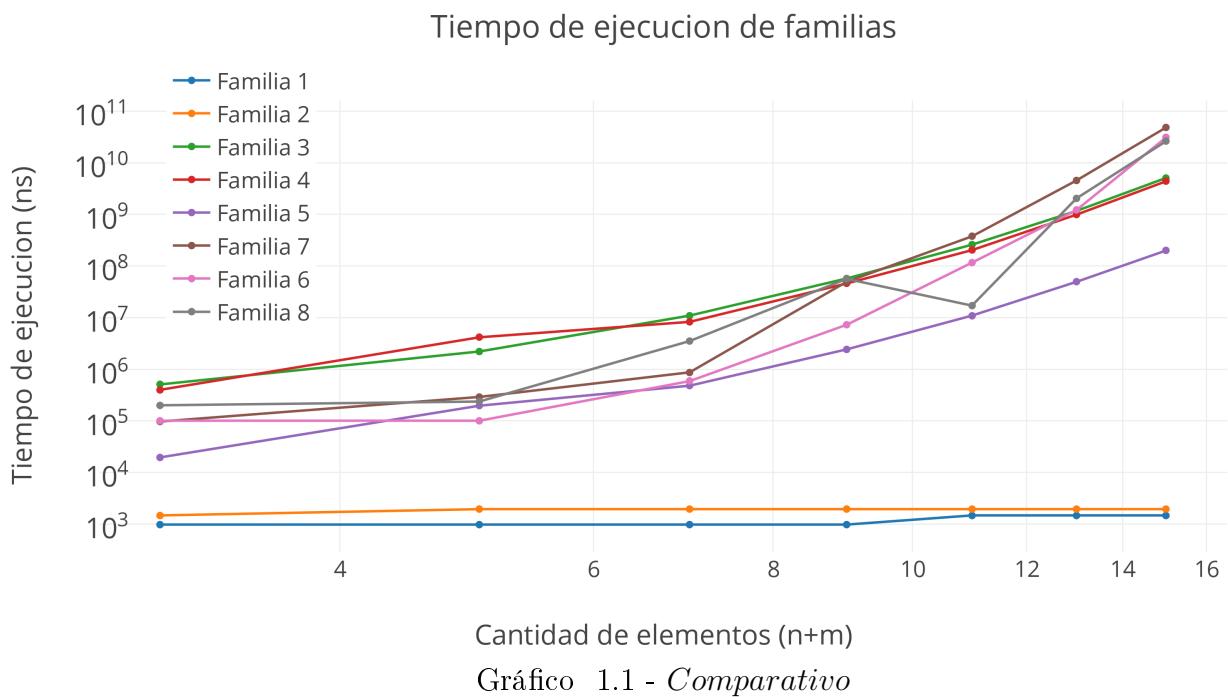
Se comienza con una entrada de un gimnasio y una pokeparada y se la aumenta en cada test en uno hasta llegar a un total de 20 elementos entre ambos conjuntos (Debido al poder de computo, no es posible trabajar con instancias de mayor tamaño).

Para obtener las mediciones se realizaron aproximadamente unas 20 corridas sobre cada una de las instancias y se tomó el promedio de las mediciones resultantes.

Dado que estas familias están pensadas para mostrar el funcionamiento del algoritmo y como funcionan las podas, fué difícil crear varios casos de tests para cada tamaño ya que sus parámetros están totalmente controlados para asegurarse de estar dentro de la familia deseada.

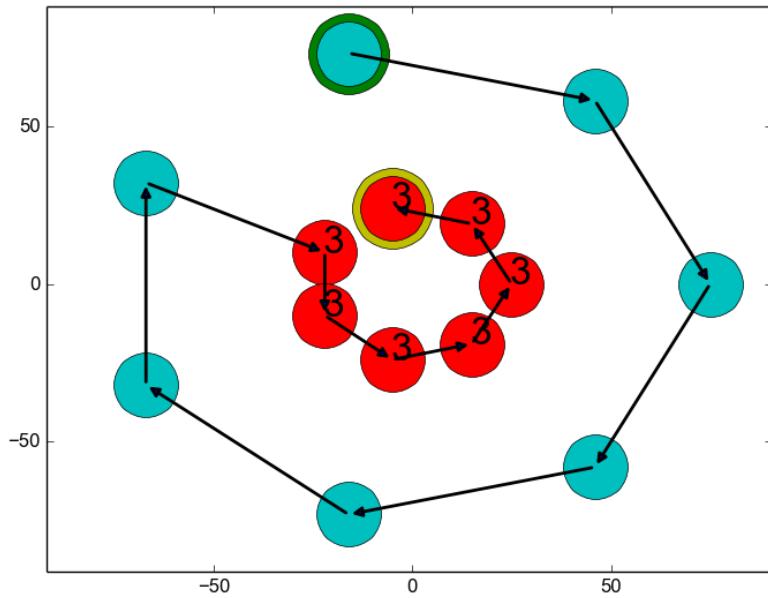
Por este motivo, en el ejercicio dos, serán presentadas dos familias nuevas, con las que se mostrará que sucede en promedio cuando se hace crecer el tamaño de entrada.

Se puede observar en el gráfico 1.1, ocho funciones, que representan el tiempo de ejecución de las familias de casos mencionadas en el apartado anterior, utilizando una escala logarítmica para una mejor visualización de cada función:



Se puede notar que la familia 1 y la número 2, presentan una mejor performance en relación a las otras. Esto se debe a que se realizan dos podas para chequear si la entrada posee o no solución. Puntualmente, la familia 1 es un poco mejor que la 2 debido a que en la implementación de las podas, la que evalúa si la mochila es de capacidad suficiente, corta apenas encuentra un gimnasio que no se pueda derrotar. En cambio la otra (chequeo de la cantidad de pociones disponibles vs necesarias) debe necesariamente chequear todos los gimnasios y pokeparadas, tardando más tiempo.

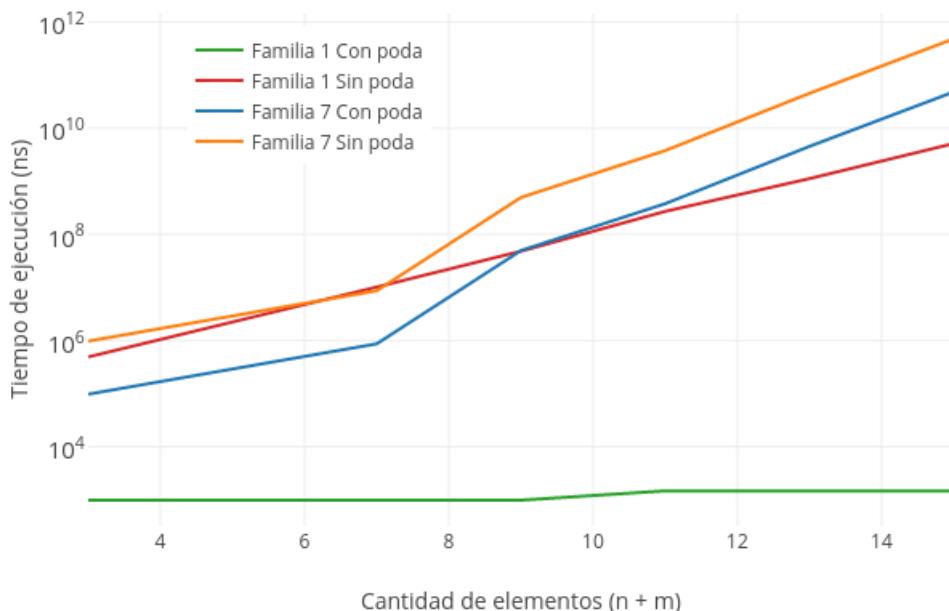
Uno de los peores casos para nuestro algoritmo es la familia 7, que sucede cuando todos los caminos tienen igual longitud. Esto se da así ya que nuestro algoritmo chequea todas las ramas posibles y, como todas pueden generar una solución posible, avanza por ellas, llegando al final de cada una con el mismo valor.



Ejemplo n y m iguales en 7 elementos cada uno

Esto genera, al comparar con un algoritmo de fuerza bruta sin podas, que los tiempos se asemejen, dado que la acción de las mismas es mínima a la hora de cortar casos no viables. El mejor caso, en cambio, sufre una mejora mucho más drástica en comparación a la fuerza bruta sin podas, ya que las mismas detectan las opciones inválidas de forma inmediata.

Medicion con y sin poda



Con y sin poda

Cota al tiempo de ejecución

Como se trabaja con pocos casos y la complejidad teórica es muy grande, no será posible graficar tiempos ya que los mismos son muy elevados, y dado que $N+M$ es pequeño (1 a 20), la muestra no es significativa. De querer trabajar con $N+M > 20$, la capacidad de cómputo de los dispositivos físicos disponibles es inferior a la necesaria, haciendo impracticable su análisis.

4 Ejercicio 2

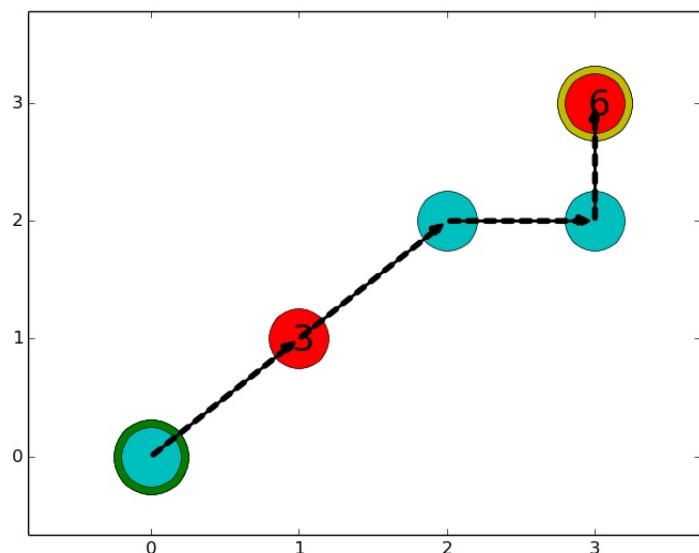
4.1 Explicación de resolución del problema

El problema propuesto plantea encontrar el menor camino a recorrer. En base a esto se decide buscar en cada paso el menor camino que una a dos puntos, generando una aproximación de heurística golosa.

Para generar la heurística se toma de base la lógica restrictiva de las podas del backtracking, lo que nos asegurará que de encontrar una solución será correcta. Para el paso goloso se tienen en cuenta los siguientes puntos:

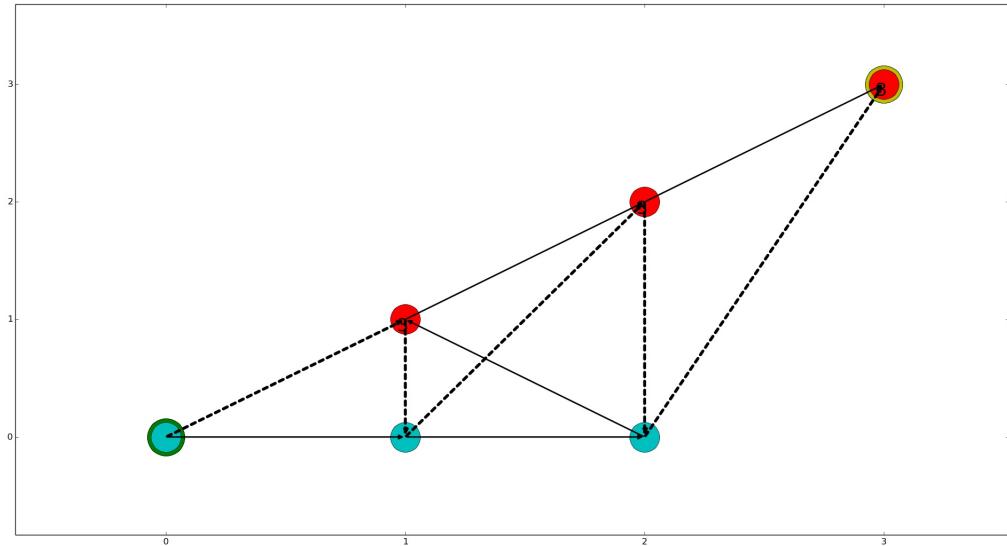
- Se prioriza ganar gimnasios, por lo tanto de poder ganar un gimnasio se procederá a hacerlo.
- La elección de movimiento será aquella que este más proxima al lugar actual del maestro pokémon: Se busca la mínima distancia entre nodos.
- En cada iteración de búsqueda de un nuevo destino, primero se evaluarán los gimnasios y luego las poke-paradas: Si ya se cuenta con una cantidad suficiente de pociones para derrotar a un gimnasio entonces se procede a vencerlo.
- Al finalizar una rama completa, es decir habiendo recorrido todos los gimnasios de haber sido posible desde una pokeparada determinada, se vuelve a correr el algoritmo desde otra pokeparada diferente, intentando generar así una mejor solución. De encontrarla se la tomará como la mejor solución hasta el momento.

La aproximación a travez de una heurística nos permite obtener soluciones en forma mucho más rápida que el método de backtracking. El paso goloso nos permite ahorrar evaluaciones de casos en que es trivial su invalidez. Un ejemplo de una solución correcta alcanzada por el algoritmo es la siguiente: La solución óptima esta representada por el camino 1, y la obtenida, por el camino 2 (se encuentran ensimados):



La solución obtenida es la óptima (ambos caminos se solapan)

No obstante, la aproximación golosa tiene como desventaja de que ya no asegura que la solución a obtener sea la óptima: un ejemplo de esto se puede ver en el siguiente escenario, en donde la capacidad de la mochila es 9 y todos los gimnasios se vencen con 3 pociones:



*La solución obtenida no es la óptima
camino punteado = goloso, camino continuo = exacto*

Ambos caminos empiezan en la misma pokeparada: el camino exacto recorre primero todas las pokeparadas y luego con todas las pociones, recorre los gimnasios resultando este camino mínimo. En cambio, el camino hecho por el goloso (punteado), al tener en la primera pokeparada las pociones necesarias para ganar un gimnasio, toma esta opción: al no tener más pociones, pasa a buscar más pociones que, al obtenerlas, vuelve a dirigirse hasta el próximo gimnasio. La distancia lograda de esta manera, resulta mucho mayor que la mínima necesaria.

4.2 Pseudocódigo

Nuestro algoritmo cuenta con 3 secciones marcadas, la primera en la cual chequeamos las podas sin solución, la segunda donde se consigue el camino, y la última donde se devuelve dicho camino con la cantidad de nodos y distancia recorrida.

En detalle, para nuestra primer etapa, desarrollamos un ciclo para poder ver el poder que posee cada uno de los gimnasios, chequeando si la capacidad de la mochila soporta el poder de los mismos. A su vez, se va sumando el poder de cada gimnasio para luego chequear si la cantidad de pokeparadas alcanza para ganar en todos los gimnasios.

Luego, en la segunda sección, se realiza un ciclo el cual realizan $\#Poke\text{-paradas} + \#Gimnasios$ de vueltas. Este ciclo se realizó para que exista un camino con cada uno de los nodos como inicio. Dentro de dicho ciclo por cada iteración del mismo creamos un nuevo *maestroPokemon* para realizar sucesivas elecciones golosas para armar un camino posible. Dicha elección golosa tomará un gimnasio siempre y cuando se lo pueda vencer y se encuentre a mínima distancia entre los que puedan ser vencidos. En el caso de que esto no sea posible, se tomará a la pokeparada más cercana. Se guardará la distancia que será recorrida para ir a esta nueva posición y las posiciones del nuevo nodo para

ir formando el camino. Esto se realizará hasta que se gane en todos los gimnasios. Luego, si la distancia recorrida es menor a una anterior iteración se reemplaza la anterior por la nueva al igual que el camino recorrido. Por último, nuestra tercer sección devolverá el mejor camino obtenido con la distancia recorrida y la cantidad de nodos del camino.

A continuación, mostraremos el pseudocódigo que exemplifica lo enunciado.

función *EJ2()*

```

crear cola decisiones para guardar elecciones
crear cola opciones para guardar todas las opciones posibles
eleccion contiene un id, posicion de la misma y distancia desde el punto anterior hasta el
como tambien la cantidad de pociones hasta el momento
se crea esPosible ← verdadero                                         O(1)
se crea minimo ← -1                                                 O(1)

Para cada pokeparada y gimnasio hacer                                         ciclo: O(n+m)
    asignar pokeparada o gimnasio como inicio de camino
    Mientras esPosible hacer                                         O(1)
        si lesGaneATodos(MaestroPokemon) ∧ tiempo(MaestroPokemon) < minimo
        entonces                                         ciclo1: O(n+m)
            minimo ← tiempo(MaestroPokemon)
            caminoRecorrido ← caminoRecorrido(MaestroPokemon)
            guarda: O(1)
            fin si                                         O(1)
            esPosible ← eleccionGolosa(MaestroPokemon)          O(n + m)
            fin ciclo
        fin para                                         O(n + m)
        devolver minimo, tamaño(caminoRecorrido) y caminoRecorrido
    fin función                                         total: O((n + m)3)

```

```

función eleccionGolosa(maestroPokemon)
    creo eleccion                                         O(1)
    Para cada opcion hacer
        eleccion ← opcion                               ciclo: O(n+m)
        recalcular(eleccion)                            O(1)
        si distancia(eleccion) < minima ∧ esValida(eleccion) entonces
            si ¬ (minimo_gym ∧ tipo(eleccion) == pokeparada) entonces
                minimo ← distancia(eleccion)           O(1)
                minimo_gym ← tipo(eleccion) ≠ pokeparada O(1)
            fin si
        fin si
    fin para
    encolar en decisiones la eleccion tomada          O(1)
    eliminar opcion de cola de opciones               O(1)
    actualizar estado del sistema en base a la decision tomada: O(1)
    si tipo(eleccion) == GIMNASIO entonces
        decrementar maestroPokemon.cantidadGymFaltantes en 1      guarda: O(1)
        maestroPokemon.cantidadPociones ← pociones(maestroPokemon) - O(1)
        pocionesNecesarias(eleccion)                                O(1)
    fin si
    de lo contrario
        maestroPokemon.cantidadPociones ← pociones(maestroPokemon) + 3 O(1)
    fin si
    sumar a maestroPokemon.tiempo tiempo(eleccion)           O(1)
    guardar estado como efectuado                      O(n+m)
    actualizar eleccionActual con la nueva eleccion       O(1)
fin función

Complejidad total:  $O(n + m)$ 

función esValida(eleccion)
    si eleccion.tipo == POKEPARADA entonces
        si maestroPokemon.pociones == maestroPokemon.capacidadMochila entonces
            guarda: O(1)
            devolver falso
        fin si
    fin si
    de lo contrario
        si maestroPokemon.pociones < pocionesNecesarias(eleccion) entonces
            devolver falso
        fin si
    fin si
    devolver verdadero                                 O(1)
fin función

Complejidad total:  $O(1)$ 
```

4.2.1 Nota 1

El ciclo con guarda *esPossible* iterará $n+m$ veces (pokeparadas + gimnasios) dado que esta es la longitud del camino más largo posible. Cada iteración realizará en el peor de los casos $n+m$ comparaciones, buscando el siguiente destino para formar el camino.

Como se evalúan todos los inicios de caminos posibles, entonces el ciclo descripto arriba se ejecutará $n+m$ veces.

4.2.2 Estructura interna para chequear elecciones y estados ocurridos

Se trabajó con dos estructuras internas denominadas *MaestroPokemon* y *Elección*, donde la primera tiene la información sobre la cantidad de gimnasios y pokeparadas, la cantidad de gimnasios ya derrotados, la posición actual, la cantidad de posiciones que posee en cada momento como también la capacidad máxima de la mochila, y dos listas que poseen todos los destinos posibles visitados o no y otra con solo los visitados. Mientras que nuestra segunda estructura, como el nombre lo indica, nos otorga toda la información necesaria para poder realizar la elección, como es el id de la misma para no ser repetida, la posición en donde se encuentra, la cantidad de pociones necesarias en caso de ser un gimnasio y la distancia para llegar a dicha posición desde cierto punto. La función *recalcular*, nos dará el valor de la distancia al cuadrado de las diferencias entre el nodo donde se está parado y la potencial elección.

4.3 Análisis de complejidades

Como ya fué analizado, la longitud de cada solución (entendida como la cantidad de lugares visitados que la componen) se encuentra acotada por $n + m$ siendo n la cantidad de gimnasios y m las pokeparadas en cada instancia. El procedimiento, se aplica una vez por cada elemento, es decir $O(n + m)$ veces.

En cada una iteración, a diferencia del backtracking que analiza todo un sub-arbol de posibilidades, la heurística golosa solo se queda con una sola rama. Con esto último podemos ver lo siguiente: agregar un elemento a una rama demanda $O(n+m)$, siendo la cantidad de elementos en ella $O(n+m)$. Su construcción se efectúa en $O((n+m)^2)$ dejando la totalidad del procedimiento ($n+m$ ramas a construir) en una complejidad de $O((n+m)^3)$.

Siendo que se siguen aplicando todas las podas que se efectuaban en el backtracking, el mejor caso de este algoritmo sucede al no haber solución, ya que la única operación que se realiza es el chequeo de la poda que preevalúa la instancia, con lo cual el algoritmo queda inferiormente acotado por $\Omega(n)$.

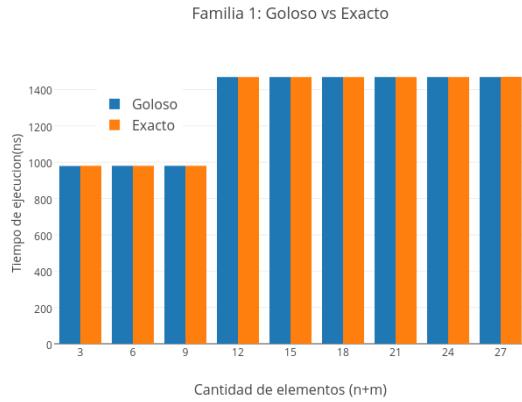
4.4 Experimentos y conclusiones

Nuestro algoritmo chequea en cada paso si existe algún gimnasio capaz de ser vencido y, si existe, busca cual es el más cercano, por lo tanto existirán casos en los cuales la solución obtenida para los mismos sea la óptima pero para algunos no lo será.

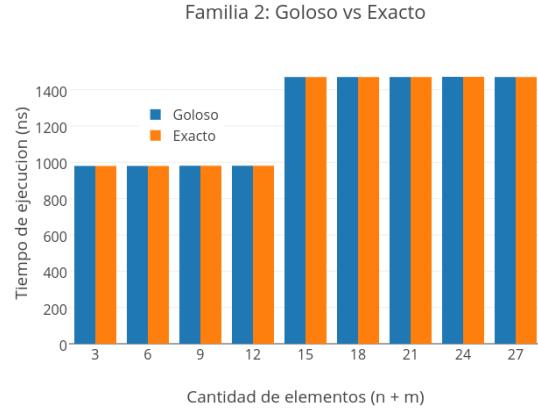
Familias con solución obtenida igual a la óptima

Familia 1 y 2

Ambas familias devolverán -1 ya que como se explicó anteriormente tanto el greedy como el exacto presentan podas para estos casos sin solución por lo tanto, su tiempo de ejecución será aproximadamente el mismo.



(a) Familia 1: actua poda 1



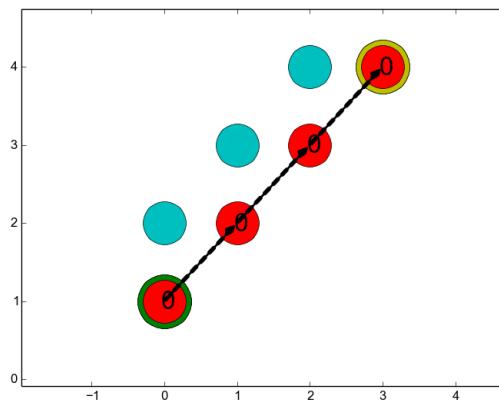
(b) Familia 2: actua poda 2

Como se observa en los últimos gráficos, las funciones resultantes para cada familia en ambos algoritmos presentan el mismo tiempo por lo comentando sobre las podas realizadas.

Familia 3

En este caso, como nuestro greedy chequea si hay algún gimnasio a ser vencido con la cantidad de pociones que se tienen en el momento (se inicia con 0), y como todos necesitan 0, recorre los gimnasios sin necesidad de pasar por las pokeparadas, obteniendo la mejor solución posible.

A continuación mostraremos el camino obtenido tanto para el algoritmo exacto como el goloso de un caso en el que se trabaja con 8 elementos en total para exemplificar lo enunciado anteriormente:

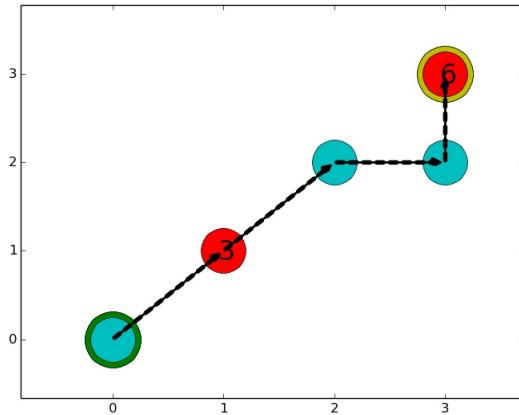


Punteado = resultado exacto, continua = resultado goloso

Como se observa en el ejemplo el camino obtenido es exactamente el mismo.

Familia 5

Se obtendrá la solución óptima para esta familia ya que se reciben primero pokeparadas para vencer a un gimnasio cerca del mismo y luego más pokeparadas para vencer a otros gimnasios que se encuentren cerca de los mismos. Se mostrará a continuación un dibujo que exemplifica lo dicho:



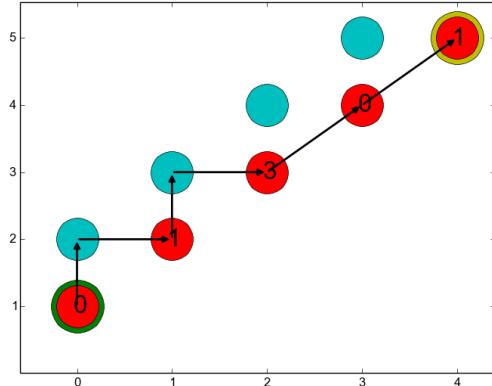
Punteado = resultado exacto, continua = resultado goloso

Familia con solución no óptima

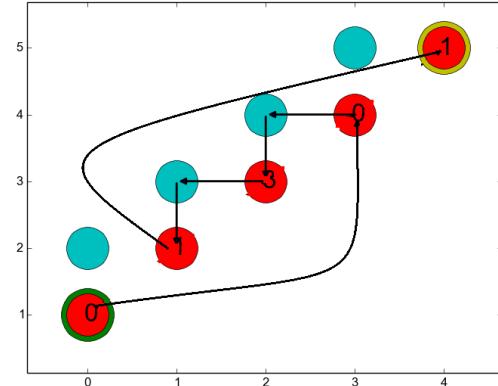
Familia 4

En este tipo de familia existan gimnasios que no necesiten pociones para ser vencidos y otros que sí. Nuestro algoritmo, por cada iteración chequea si puede elegir un gimnasio que se encuentre a una distancia mínima en relación a los demás, y además corrobora si posee las pociones necesarias para vencerlo, decide inicialmente ir a vencer a los gimnasios que posean cero poder, lo cual puede no ser óptimo para el resultado final.

Este es un ejemplo del algoritmo exacto y el goloso con un total de 10 elementos:



(a) Algoritmo exacto

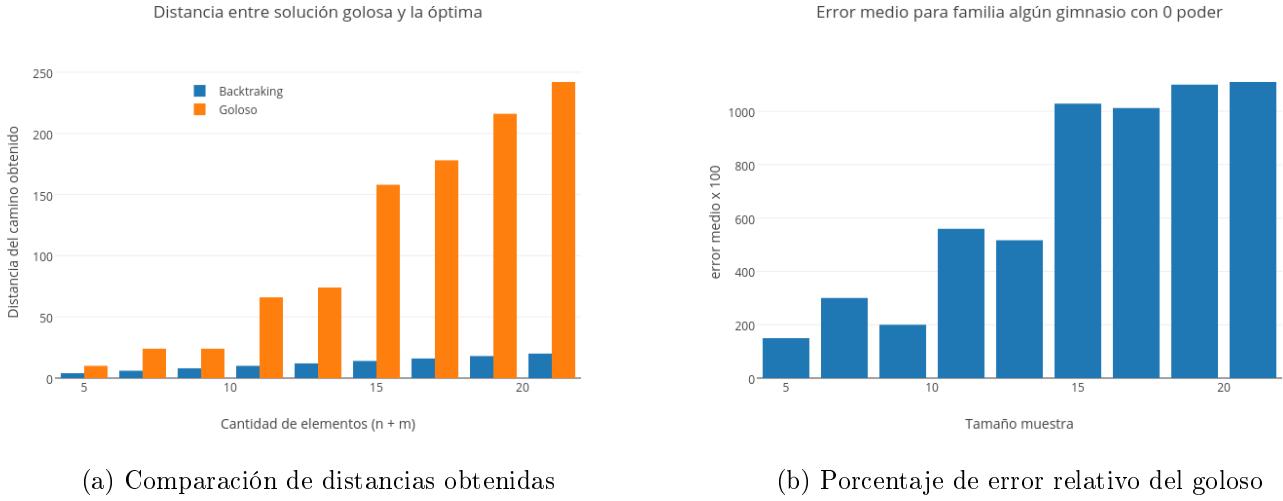


(b) Algoritmo goloso

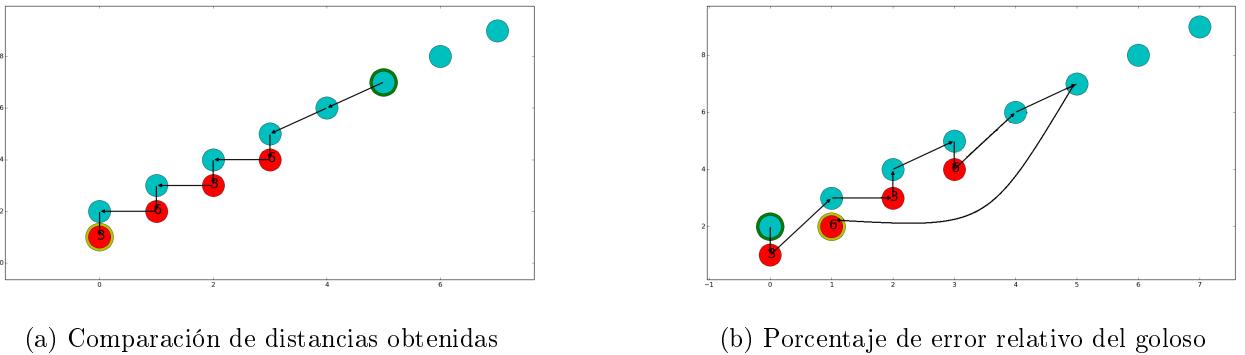
Con respecto a la diferencia entre la soluciones que se obtienen en relación a las óptimas elaboramos las siguientes comparaciones:

Debido al poder de cómputo utilizado para realizar los tests, solo pudo testearse el algoritmo exacto hasta con 20 elementos teniendo que bajar inclusive hasta 15 elementos para algunas familias, mientras que el goloso puede tomar una mayor cantidad de elementos con tiempos de ejecución considerablemente menores.

Familia 6



Este estilo de familia presenta a los gimnasios y pokeparadas desordenados en referencia a las posiciones, es decir, para ganar a cierto gimnasio es necesario pasar por una cantidad puntual de pokeparadas las cuales estan de un lado y del otro de dicho gimnasio.

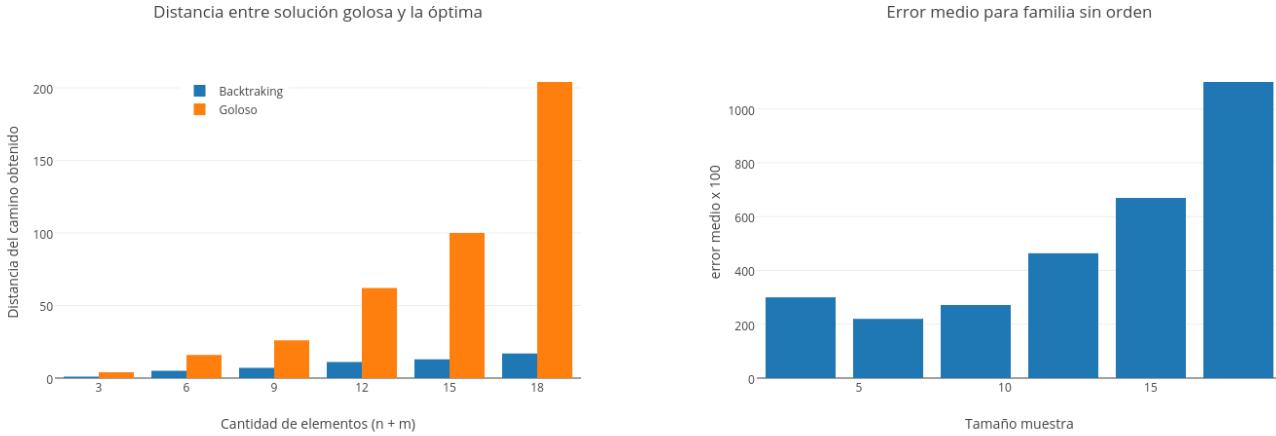


Se puede observar en el ejemplo como nuestro algoritmo goloso va a la primer pokeparada y de ahí a vencer al gimnasio más cercano en vez de ir a la pokeparada consecutiva. Esto lo hace hasta vencer a todos los gimnasios.

La diferencia entre soluciones se pueden apreciar en los siguientes gráficos:

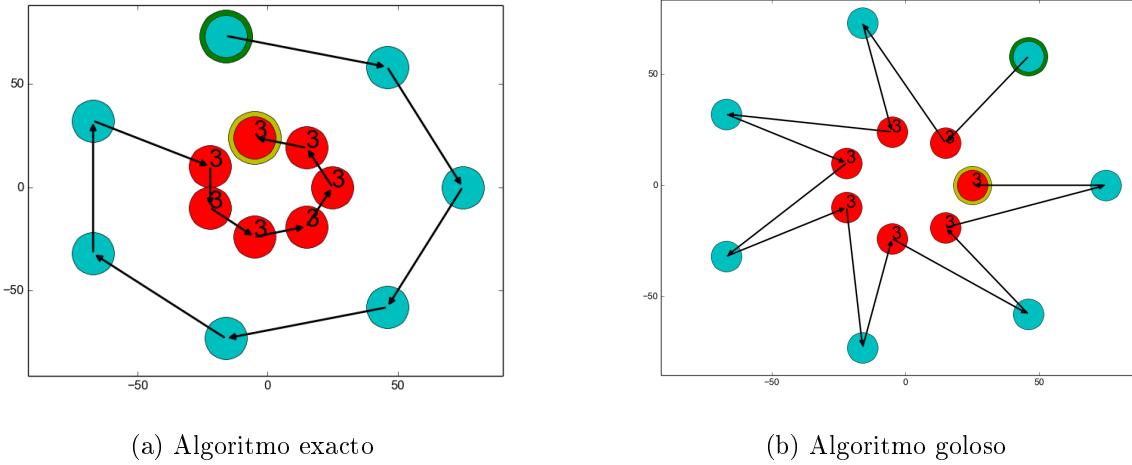
Familia 7

El objetivo de esta familia es plantear un caso en donde, de forma controlada, la respuesta del algoritmo goloso nunca fuera la óptima. Para ello, se genera una instancia agrupando los gimnasios y pokeparadas en 2 anillos de radios diferentes. Dado que nuestro algoritmo siempre y cuando pueda vencer a algún gimnasio buscará el mínimo en distancia para vencerlo, para esta familia resultará contraproducente: es preferible adquirir más pociones para luego ir a varios gimnasios juntos, que ganar apenas exista la posibilidad.



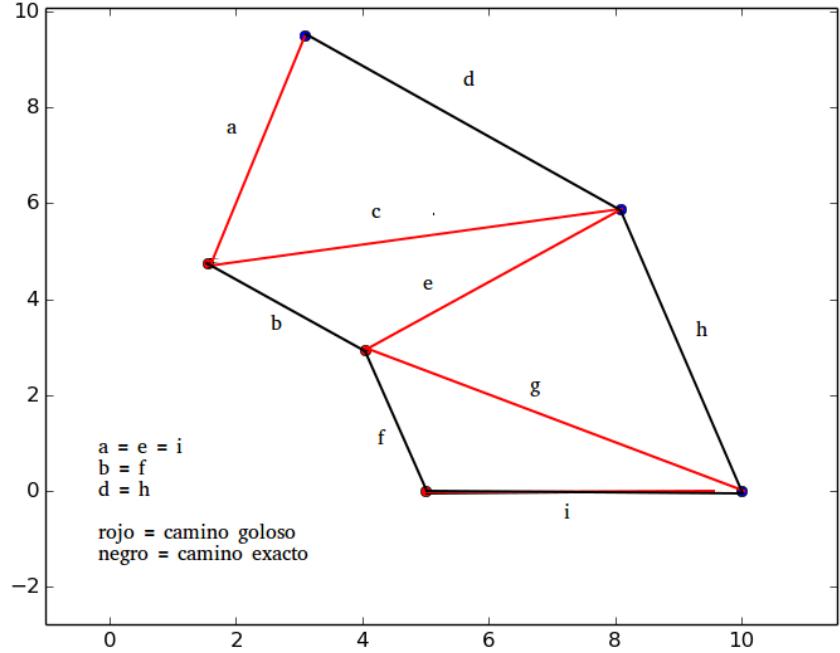
(a) Comparación de distancias obtenidas

(b) Porcentaje de error relativo del goloso



Se puede observar en el último gráfico como los resultados tienen valores diversos para casos pequeños. En las instancias de mayor tamaño, la solución óptima resulta de recorrer primero todas las pokeparadas y luego los gimnasios, ya que la cantidad de pociones necesarias para vencer a todos los gimnasios es igual a la cantidad total de pociones presentes en el mapa, sumado a que la distancia entre pokeparadas es muy inferior que entre pokeparadas y gimnasios.

Al generarse un camino alternado para el goloso entre gimnasio y pokeparada, se le dan a todos los gimnasios la dificultad igual a la cantidad de pociones que aporta cada pokeparada y se crean la misma cantidad de gimnasios como de pokeparadas. En cuanto a la distribución de los mismos, se buscó una forma de estrella, en la cual cada pokeparada está alineada con 2 gimnasios y una segunda pokeparada. Dadas estas restricciones se observa lo siguiente:



Comparación solución golosa vs exacta

El camino goloso realiza un camino de longitud $\sum_{i=1}^k a + \sum_{i=1}^j c$ y el exacto $\sum_{i=1}^{n-1} b + \sum_{i=1}^{m-1} d + a$. Siendo k el número de aristas como a y j como c , se tiene que $k + j = n + m - 1$ (aristas en un camino de $n+m$ elementos) y que $k = j + 1$. Para que el algoritmo goloso difiera del exacto, se debe cumplir entonces la siguiente desigualdad:

$$\sum_{i=1}^k a + \sum_{i=1}^j c \geq \sum_{i=1}^{n-1} b + \sum_{i=1}^{m-1} d + a$$

La cual equivale a decir:

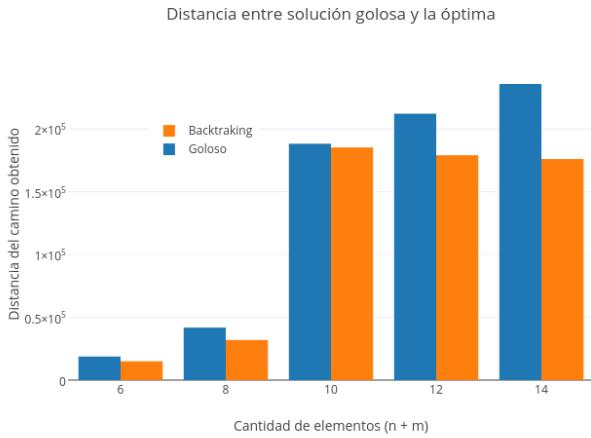
$$\begin{aligned} \sum_{i=1}^k a + \sum_{i=1}^j c - a &\geq (n-1)b + (m-1)d \\ \sum_{i=1}^j a + \sum_{i=1}^j c &\geq (n-1)b + (m-1)d \\ j(a+c) &\geq (n-1)b + (m-1)d \end{aligned}$$

Siendo que $n = m$ por como se construye la instancia, entonces:

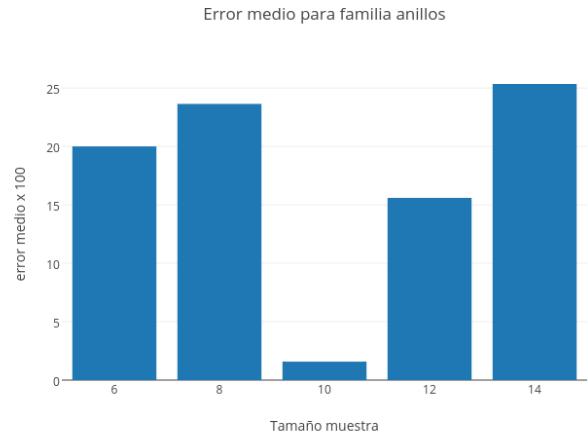
$$j(a+c) \geq (n-1)(b+d)$$

Finalmente, como $k = j + 1 \wedge k + j = n + m - 1 \Leftrightarrow j = n - 1$, se resolverán distintos caminos siempre y cuando $a + c \geq b + d$. Con lo cual, para generar correctamente las instancias para esta familia, se consideró dicha desigualdad.

Podemos ver un ejemplo de la desigualdad de las soluciones presentes en las siguientes instancias corridas con el algoritmo goloso y el backtracking:



(a) Algoritmo exacto

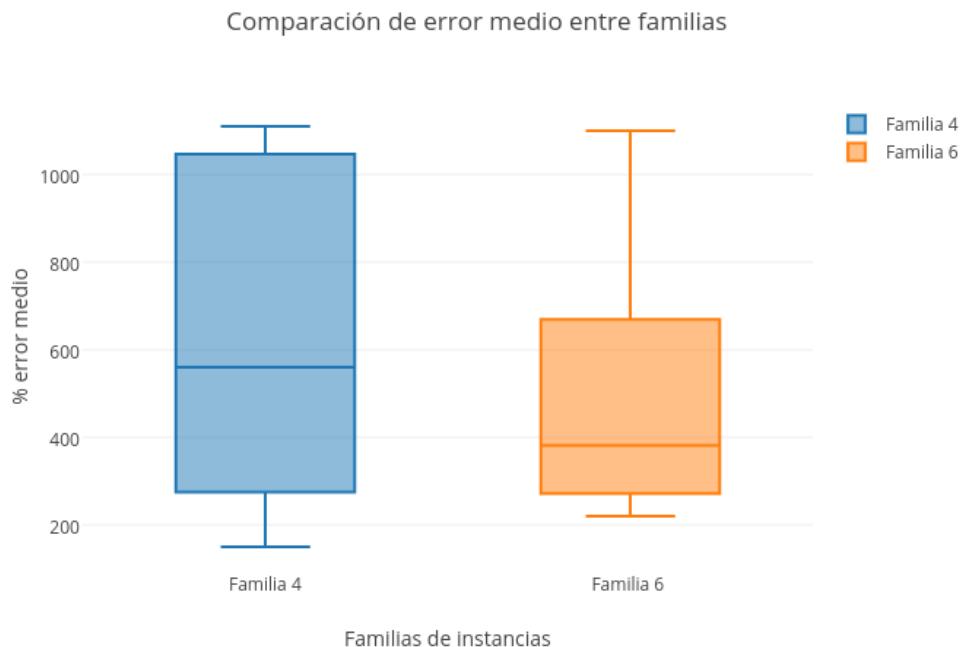


(b) Algoritmo goloso

Los radios para entradas de hasta 8 elementos son distintos al resto de las instancias. En síntesis se puede ver para los tamaños de 10 a 14 elementos que se mantuvo la distancia recorrida en el algoritmo exacto: esto se debe al carácter del camino, que resulta de recorrer primero una circunferencia, luego dirigirse de forma recta a la circunferencia interior y recorrerla. Al no haber variabilidad en los radios, el tiempo insumido es aproximadamente el mismo. No así para el algoritmo goloso, ya que un incremento en la cantidad de elementos, conyeva un incremento de las idas y vueltas a realizar; aumentando notoriamente la distancia del recorrido.

Comparación entre errores de familias

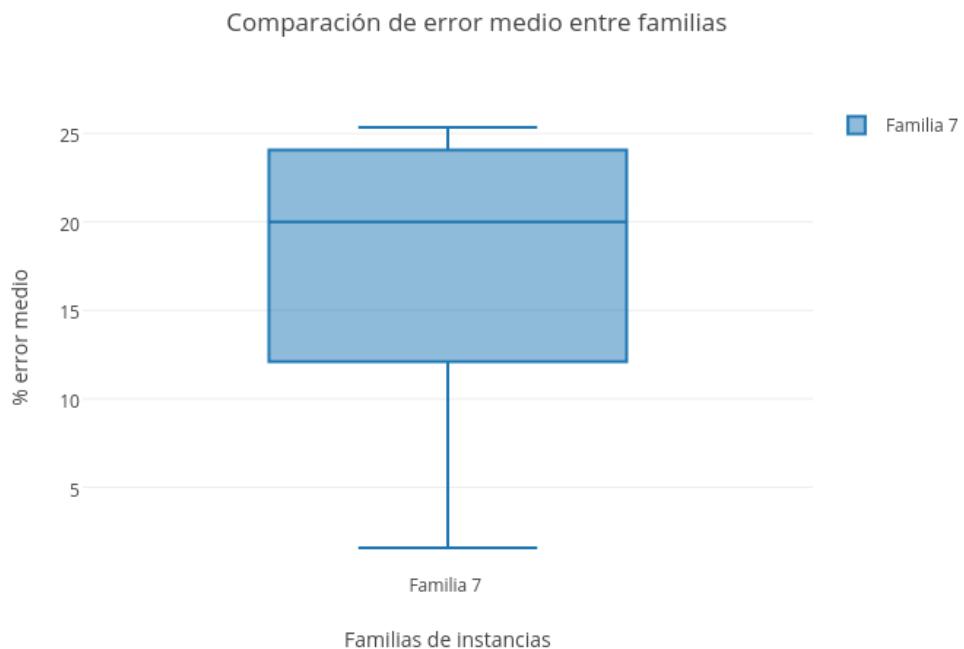
Teniendo un grupo de familias que provocan soluciones con errores al aplicar el algoritmo goloso, podemos ver lo siguiente entre la Familia 4 y la Familia 6:



Se evidencia el mejor desempeño del algoritmo sobre la familia 6, provocando un error medio mucho menor que la familia 4 (alrededor de 110% mejor) y con una dispersión del mismo mucho más

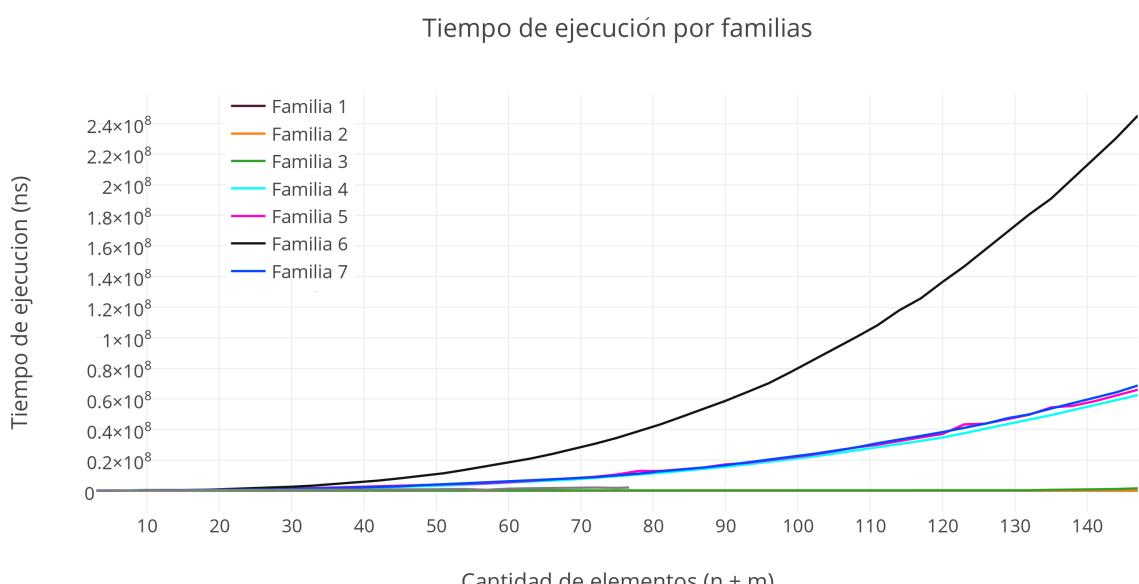
baja.

De forma separada, debido a los rangos que se manejan, se decidió analizar la distribución de la Familia 7. Vale destacar que el error de la misma es muy inferior al de las dos anteriores y posee la menor varianza de todas. Esto se debe al carácter controlado de la misma, y su característica de posicionamiento:

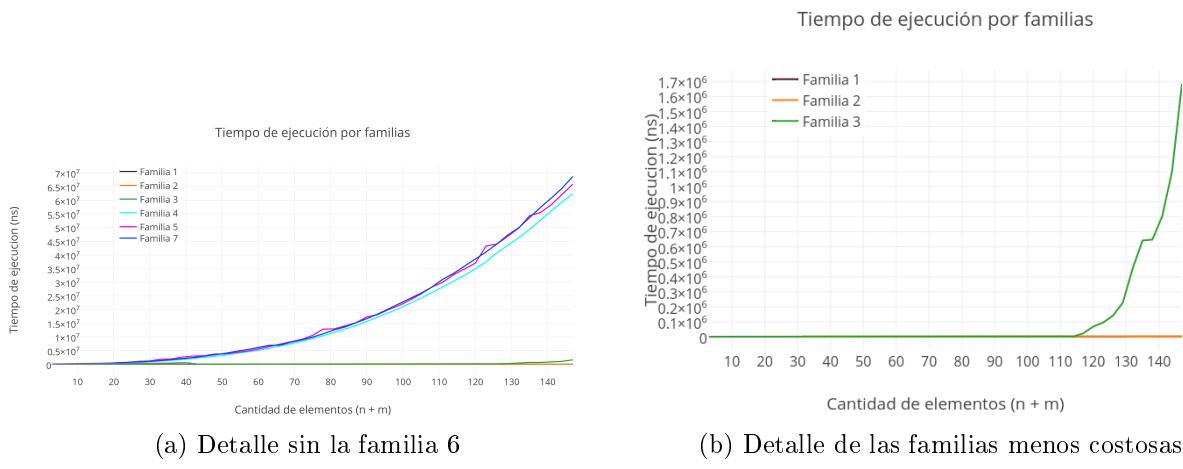


Comparación entre tiempos de familias

Veremos como se comporta cada familia en función del tiempo al ir aumentando la cantidad de elementos manteniendo las condiciones para que sigan perteneciendo cada uno a su respectiva familia.

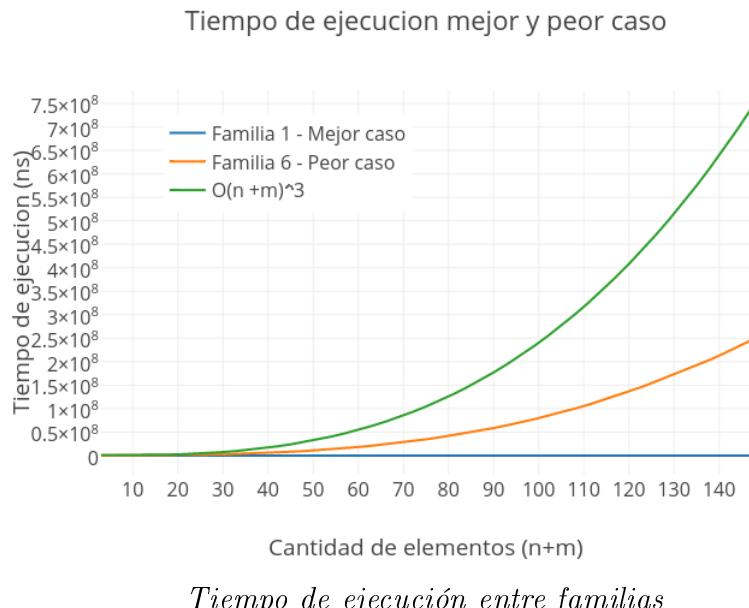


Tiempo de ejecución entre familias



Se puede observar como la familia número 6 presenta una peor performance en comparación al resto mientras que tanto la familia 1 como la 2 presentan un tiempo que se torna constante dando una mejor performance en relación al resto, lo cual se debe a las podas utilizadas para estas familias como mencionamos anteriormente. Mientras que la número 6 presenta la dificultad en la cual todos los elementos del mapa se encuentran desordenados, por lo tanto nuestro algoritmo tiene que llegar a hacer hasta un doble viaje para poder vencer a un gimnasio, ya que se dan instancias en las cuales los gimnasios de poder menor o igual a 3 se encuentran muy lejos de las pokeparadas en relación a los gimnasios de poder mayor, los cuales estan "pegados" a las pokeparadas insumiendo así un tiempo mayor de decisión y ejecución. Como habíamos visto, nuestro algoritmo siempre que puede vencer a un gimnasio va a vencerlo.

Luego, mostraremos como se comporta nuestro algoritmo en base a la complejidad calculada anteriormente:



Es posible observar como las funciones resultantes del mejor y peor caso se encuentran por debajo de la cota de complejidad. Dicha complejidad fué calculada utilizando el método de cuadrados

mínimos generando una función que es tomada como cota dentro de nuestro orden de complejidad ($O(n + m)^3$).

4.4.1 Familias random y Gimnasios por grupos

Como se comentó previamente en el ejercicio uno, las anteriores familias fueron de utilidad para analizar que sucede con el algoritmo goloso dado cierto tipo de entradas y porque no obtiene solución óptima en algunos casos, además de servir para analizar las podas tanto del algoritmo exacto como el goloso. Lo que no fué posible con estas familias es analizar que sucede en el promedio de casos con soluciones aleatorias, dado que eran familias totalmente controladas para poder observar particularidades. Por este motivo introducimos aquí dos familias nuevas con el objetivo de analizar casos promedio dentro de conjuntos de un cierto tamaño.

La primera es la familia Random, que como el nombre lo indica, todo es absolutamente random en los parámetros de entrada. Con este tipo de instancias podremos observar cual es el desempeño de los algoritmos en promedio.

La segunda familia también tiene parámetros aleatorios, pero organiza gimnasios por cuadrantes o grupos, donde $\forall C_i$ con $i \in 1..4$, todos los gimnasios en el cuadrante C_j tienen el mismo poder. Otra regla que trató de aplicarse es que el poder de los gimnasios sea diferente para cada cuadrante C_i, C_j con $i \neq j$. Esto será posible siempre y cuando no sea posible asignar más de 1 (uno) de poder a los gimnasios, en cuyo caso, puede haber cuadrantes que tienen el mismo poder para sus gimnasios.

Esta familia será como una suerte de mapa por niveles. Veremos de esta manera como se desenvuelve sobre todo el algoritmo goloso para obtener un camino dentro de estas circunstancias.

Para el caso del valor de la mochila, con más de 15 elementos solo se analizará tomando el mínimo de mochila posible, y hasta con 15 elementos, podremos comparar contra el exacto que resulta de tomar la peor o la mejor mochila. La mochila mínima será el valor del gimnasio con más poder más un extra de 3 ya que si el gimnasio más poderoso tiene valor menor a 3 podría generarse un caso sin solución. La mochila más grande será la suma de los poderes de todos los gimnasios. Mochilas con mayor tamaño harán que sobre capacidat.

Con respecto a la creación de casos para cada familia se idearon dos rangos iniciales:

- Rango 1: de 5 a 15 elementos. Por cada instancia de tamaño n se tomaron $n * 5$ instancias aleatorias.
- Rango 2: desde 70 hasta 470 elementos en intervalos de 50 elementos. Por cada instancia de tamaño n se tomó un 10% del tamaño de la entrada.

La cantidad de intancias dentro de cada tamaño será utilizada para promediar distancias y tiempos. En el caso del Rango 1 podremos calcular el error relativo de la solución del algoritmo goloso contra el exacto.

El calculo del error relativo se realiza tomando el promedio de todas las distancias obtenidas por el backtracking y el promedio de todas las distancias de las soluciones del algoritmo goloso para instancias del mismo tamaño. Sean $promedio_{exacto}$ y $promedio_{greedy}$ los valores respectivos, el error relativo se calcula como:

$$Error_{abs} = |promedio_{greedy} - promedio_{exacto}| \quad (1)$$

$$Error_{rel} = \frac{Error_{abs}}{promedio_{exacto}} * 100 \quad (2)$$

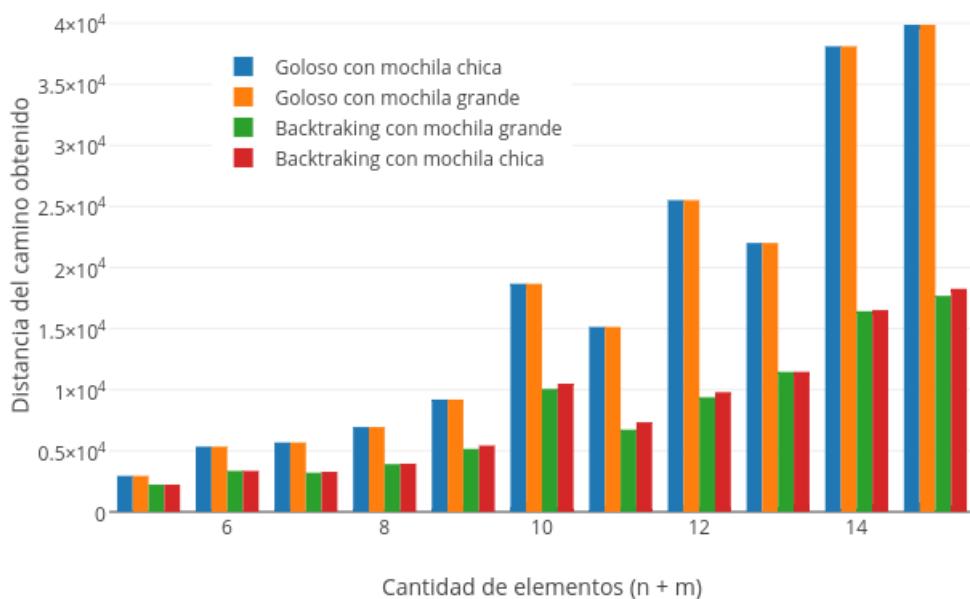
La idea del Rango 1 era poder calcular hasta con 20 elementos utilizando backtracking, pero debido a los elevados tiempos de resolución se fué achicando el tamaño de las instancias hasta 15 elementos. La cantidad de instancias tomadas por cada tamaño permite obtener promedios más consistentes para que las comparaciones contra el backtracking sean de la mejor calidad posible y puedan aportar información relevante.

Dentro del Rango 2 los intervalos de 50 elementos podrían parecer ser demasiado, pero cuanto más se discretiza más tiempo de espera se necesita para correr todos los algoritmos que se verán en este informe. Como el objetivo principal es observar que sucede a medida que la entrada crece, se priorizó testear casos grandes resignando discretización. El 10% como elección en la cantidad de ejemplos tomados en cada tamaño fué suficiente para obtener resultados consistentes y mantener controlados los tiempos de experimentación.

Repercusión del tamaño de la mochila: hasta 15 elementos (pokeparadas + gimnasios)

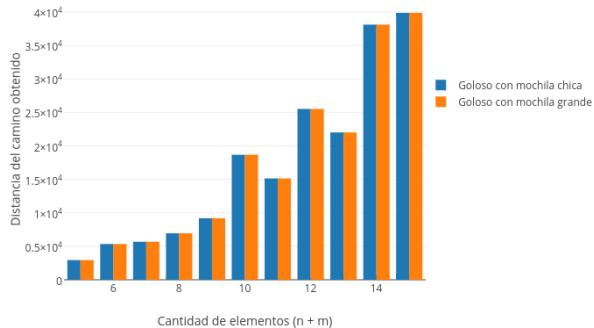
Se realizaron dos experimentos en base a la familia Random y a Gimnasios por grupos para ver como repercute el tamaño enunciado, con la mochila denominada grande y la chica. Luego de dicha experimentación, se desarrollaron los siguientes gráficos en los cuales se puede observar como la heurística golosa no presenta cambios en su solución al tener una mochila más grande, mientras que el backtracking al tener una mochila con mayor capacidad logra obtener un camino diferente y de menor longitud.

Repercusión del tamaño de la mochila en la solución



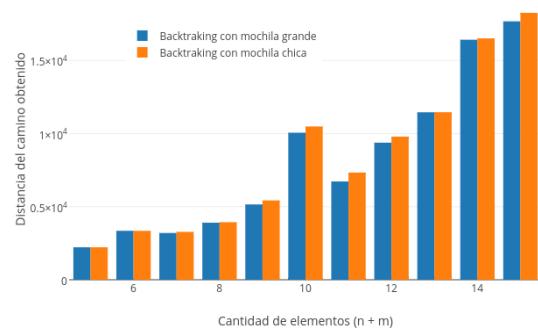
Gimnasios por grupos

Repercusión del tamaño de la mochila en la solución



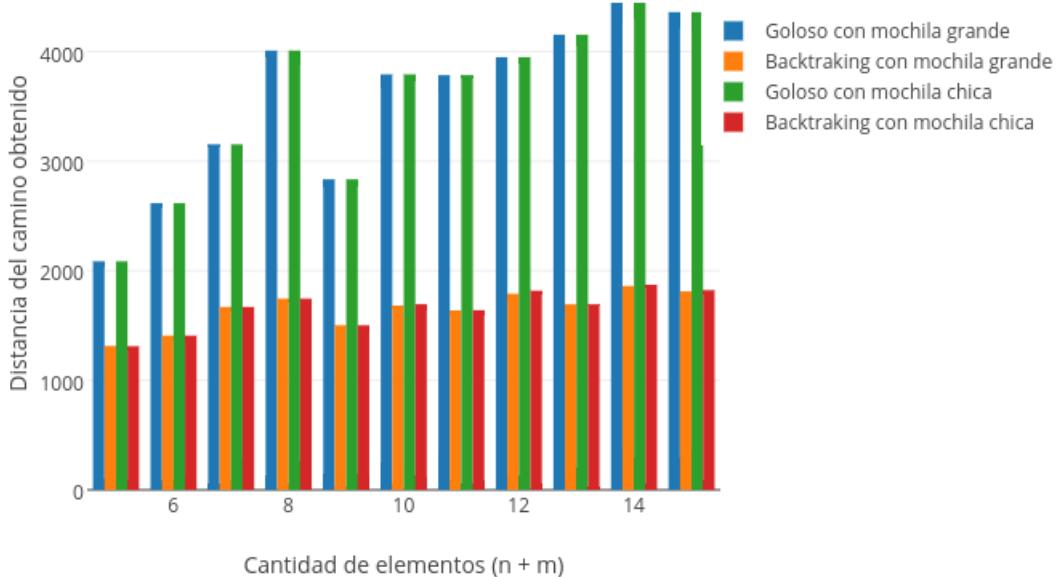
(a) Repercusión en goloso

Repercusión del tamaño de la mochila en la solución



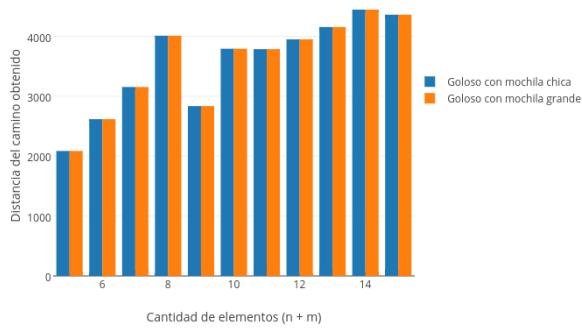
(b) Repercusión en backtracking

Repercusión del tamaño de la mochila en la solución



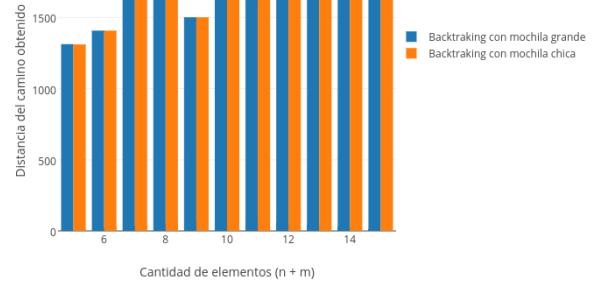
Random

Repercusión del tamaño de la mochila en la solución



(a) Repercusión en goloso

Repercusión del tamaño de la mochila en la solución



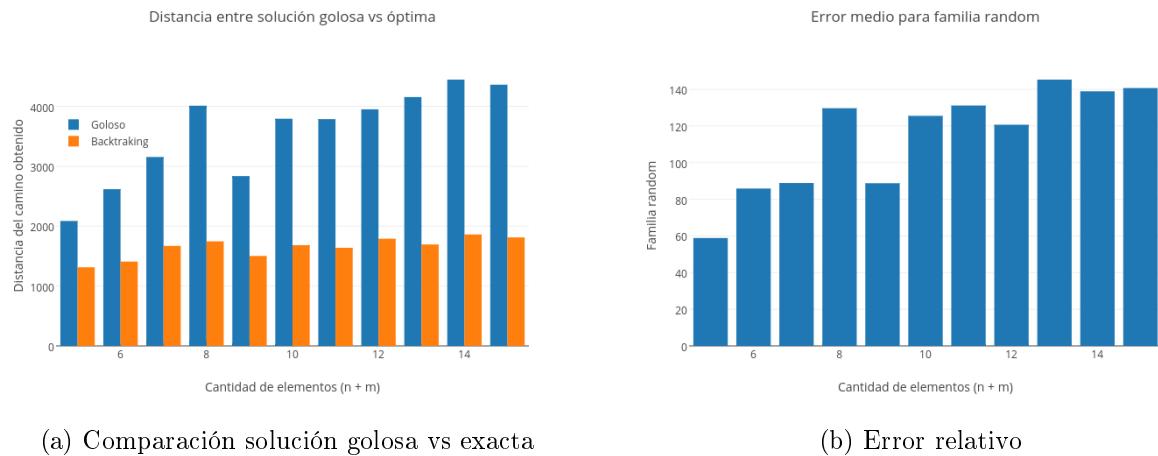
(b) Repercusión en backtraking

El motivo principal de que no haya cambios se debe a que siempre que existe la posibilidad de vencer a un gimnasio el algoritmo toma esa decisión. Podemos notar sin embargo, algunas mejorías en cuanto a las mochilas grandes para los resultados del backtracking. Siempre se podrá obtener un resultado exacto mejor si se dispone de más capacidad de carga. Si se dispone de las pokeparadas suficientes, con la mochila más grande posible podríamos primero cargar todas las pokeparadas necesarias y luego ir a vencer a todos los gimnasios. Si los gimnasios están lejos de todas las pokeparadas, esto termina beneficiando a la distancia recorrida ya que no será necesario volver a recargarse.

Repercusión del tamaño de la mochila: más de 15 elementos

Como se mencionó en repetidas oportunidades, al algoritmo goloso no saca partida de la capacidad de la mochila. Como con más de 15 elementos nos fué imposible calcular el backtracking en el ordenador disponible en tiempos razonables (más de 30 minutos para 16 elementos y era necesario correr al menos la mitad del tamaño en ejemplos para obtener promedios), no podremos comparar que sucede al utilizar la peor y mejor mochila contra el resultado exacto.

Comparación de bactracking y algoritmo goloso para la familia Random



(a) Comparación solución golosa vs exacta

(b) Error relativo

Se observa para esta familia un error considerable cuanto más grande es la instancia, llegando a estar por arriba del 100% con respecto al resultado exacto para entradas de 13 elementos. El motivo puede deberse a la variabilidad de los parámetros de entrada que hacen que el algoritmo goloso tenga que tomar las peores decisiones. Es decir, la mayoría de las veces se recorren grandes distancias para vencer a un gimnasio luego de juntar pokeparadas.

Comparación de bactracking y algoritmo goloso para la familia de Gimnasios por grupos

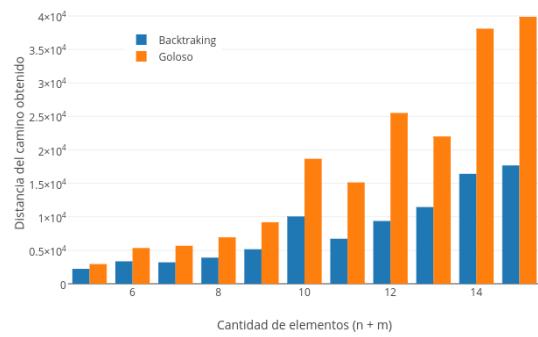
Podemos observar en las figuras el mismo mapa, donde (a) indica el recorrido exacto y (b) el recorrido goloso. Hay dos grupos de gimnasios. Un grupo con tres gimnasios de poder ocho y otro grupo con dos gimnasios de poder cero.

Con respecto a la diferencia entre la soluciones que se obtienen en relación a las óptimas elebaramos los siguientes gráficos:

Podemos observar que habria que retestear tomando mas muestras!!

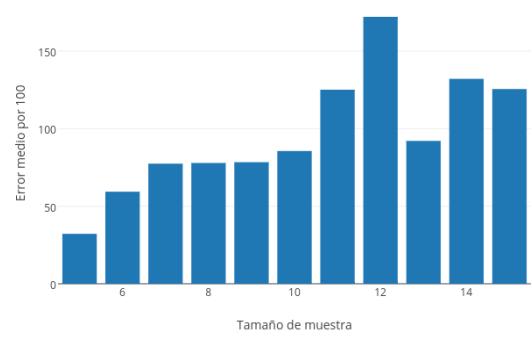
BOX: —>

Repercusión del tamaño de la mochila en la solución



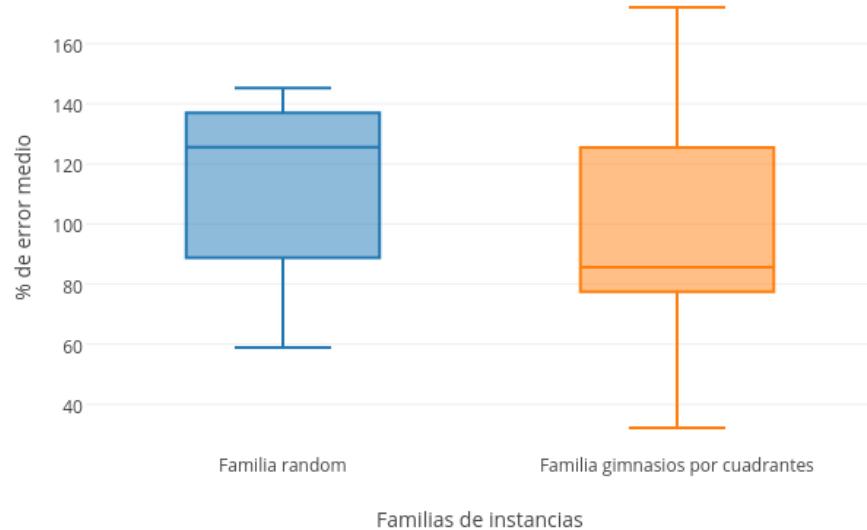
(a) Comparación solución golosa vs exacta

Error medio para familia gimnasios por cuadrantes



(b) Error relativo

Comparación de error medio entre familias



Errores absolutos por familia

5 Ejercicio 3

5.1 Explicación de resolución del problema

Dado que una solución exacta al problema de nuestro entrenador pokémon es costosa, ya que debe recurrirse al backtracking por ser un problema que busca un circuito mínimo en cantidad de pokeparadas pasando una sola vez por las mismas y sin repetir gimnasios. Esto es similar a buscar un camino hamiltoniano, pero sin recorrer absolutamente todos los nodos, con lo que se generan aun más combinaciones posibles.

En el punto anterior se decidió implementar una solución basada en un algoritmo goloso. Dado que la misma puede no ser exacta, nos interesa tratar de mejorar sus resultados.

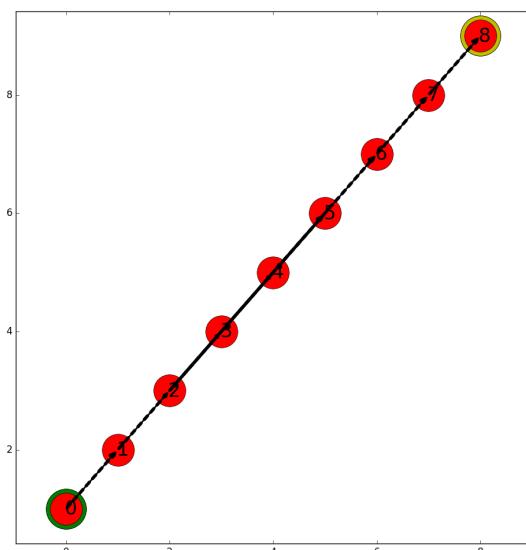
Para esto, dado que una solución es representada como una sucesión de nodos, tal que dos nodos consecutivos identifican una arista de nuestro camino, si intercambiamos el orden de ciertos nodos, estaremos modificando la solución reemplazando aristas de la solución original (De ahora en más S_o) y agregando nuevas a la misma.

De esta manera existen una cierta cantidad y tipo de movimientos que podemos realizar y que generan nuevas soluciones a la que denominamos *vecindad* de S_o . Para que un movimiento sea válido, cada vez que se realiza un cambio de aristas, se tiene que asegurar que el recorrido obtenido sea justamente un recorrido. Si cada vez que observamos una vecindad, nos quedamos con la mejor solución posible en la misma y seguimos analizando los vecinos de esta, hasta que no se produzcan mejoras, podremos refinar el resultado aún más.

Esta técnica de optimización se denomina heurística de búsqueda local.

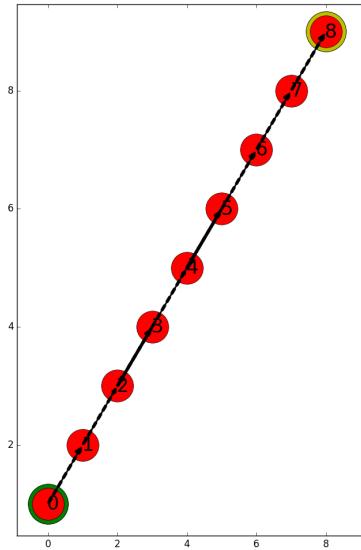
Se denomina $k-opt$ cuando específicamente se modifican k aristas de la solución. Generalmente se suele utilizar un k de 2 o 3 ya que para valores mayores se pierde granularidad y por lo tanto es posible perder mejores resultados. Aunque esto último depende mucho del problema analizado y las entradas del algoritmo. Para este informe elegimos $2-opt$.

En el informe previo a este recuperatorio se había trabajado además con $3-opt$, pero debido a los tiempos de cómputo requeridos por esta herística y los tiempos de entrega, se decidió no incluir la misma en la experimentación. Sólo se dejará el pseudo-código a modo informativo.



Movimiento 2-opt

Realizar un movimiento $2 - opt$ como se explicó, es cambiar dos aristas de un recorrido por otras dos aristas diferentes, de manera tal que el resultado siga siendo un recorrido simple. En el ejemplo puede observarse el recorrido $1 \rightarrow 2, 3, 4, 5 \rightarrow 6, 7, 8$ y el movimiento realizado cambia las aristas $(1,2)$ y $(5,6)$ por las aristas $(2,6)$ y $(1,5)$, lo que invierte el tramo $2, 3, 4, 5$ a $5, 4, 3, 2$ obteniendo el recorrido $1, 5, 4, 3, 2, 6, 7, 8$.



Movimiento 3-opt

Realizar un movimiento $3 - opt$ genera más de una opción para las tres aristas elegidas. En el ejemplo podemos ver el recorrido $1 \rightarrow 2, 3 \rightarrow 4, 5 \rightarrow 6, 7, 8$ y se eligen las aristas $(1,2)$, $(3,4)$ y $(5,6)$ intercambiandolas por $(1,3)$, $(2,5)$ y $(4,6)$ y obteniendo el recorrido $1 \rightarrow 3, 2 \rightarrow 5, 4 \rightarrow 6, 7, 8$.

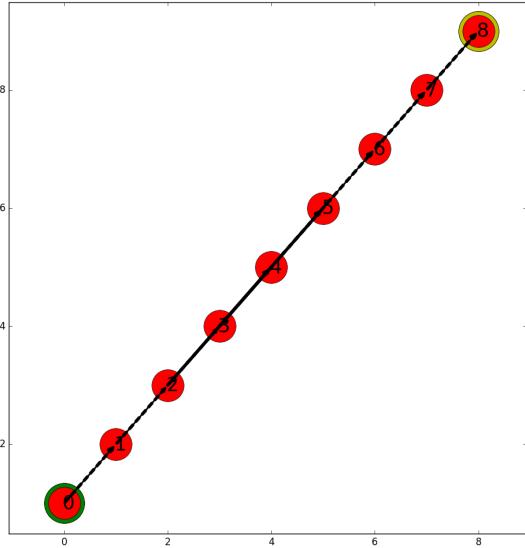
Aunque existen otras seis posibilidades, tres más son $3 - opt$ y otras tres que incluyen dos movimientos $2 - opt$ cada una. Si lo que se busca es $3 - opt$ puro, deben descartarse las que sean $2 - opt$.

Los movimientos $3 - opt$ restantes serán:

1. $1 \rightarrow 4, 5 \rightarrow 2, 3 \rightarrow 6, 7, 8$
2. $1 \rightarrow 5, 4 \rightarrow 2, 3 \rightarrow 6, 7, 8$
3. $1 \rightarrow 4, 5 \rightarrow 3, 2 \rightarrow 6, 7, 8$

$3 - opt$ requerirá que se tomen todas estas posibilidades si se busca no dejar ningún caso afuera.

Además, realizando un simple swap de nodos, podemos obtener muy fácilmente soluciones que intercambian 2 o 4 aristas. Serán dos si los nodos intercambiados de posición son consecutivos y cuatro si no lo son.



Movimiento swap

En el ejemplo se toma el camino 1,2,3,4,5,6,7,8 y se intercambian 2 por 6 generando 1,6,3,4,5,2,7,8 y generando así 4 aristas nuevas (1,6), (6,3), (5,2) y (2,7).

Además tenemos que tener en cuenta para nuestro problema, que al permutar una solución con alguno de los métodos mencionados, y la solución sea válida, pueden quedar pokeparadas al final del recorrido, por lo cual es necesario eliminarlas del mismo, ya que al considerar el recorrido hasta las mismas, se podría sumar distancia a la solución que ya no aporta, debido a que todos los gimnasios fueron derrotados. Además, podría desestimarse como solución candidata si luego se obtiene otra que tiene menor distancia solo porque se están contando pokeparadas de más.

De esta manera, podremos movernos a través del espacio de soluciones locales a S_o y tal vez mejorar la solución, aunque que de esto, por ser una heurística, no tendremos ninguna garantía.

En este punto nos centraremos en estudiar y tratar de concluir cual de las heurísticas consideradas en este punto es mejor utilizar para mejorar los resultados obtenidos por el algoritmo goloso del punto anterior. Es decir, dado un tipo de entrada, que en este caso se corresponde con un mapa de pokeparadas y gimnasios, que tendrá alguna particularidad que hará que el algoritmo goloso produzca un resultado bueno o malo, veremos que tipo de búsqueda local será mejor aplicar para mejorar la solución o si no conviene aplicar ninguna de las heurísticas de este punto ya que a pesar de ser la solución buena o mala, no se obtienen mejores resultados.

Primero veremos los pseudocódigos de las tres heurísticas y analizaremos sus complejidades. Luego realizaremos un análisis cualitativo de las mismas aplicadas a los resultados de cada tipo de entrada, tratando de abordar las características de las mismas y explicar porque una heurística resulta mejor o no en cada caso. Luego de comparar los resultados para cada tipo de entrada intentaremos elegir, si es posible, el tipo de búsqueda local que obtiene los mejores resultados en la mayoría de los casos.

5.2 Pseudocódigo

En la explicación de las heurísticas ya se mencionó como son los intercambios de aristas válidos para cada búsqueda local. Un camino será representado mediante un vector S_{actual} el cual se inicializa con el camino a mejorar S_o .

Cada posición del vector indica el *id* de una pokeparada o gimnasio y cada par de *id's* conforma una arista del mapa. Luego de los algoritmos se explicará que es un *id* para una pokeparada y gimnasio.

Se itera sobre toda la vecindad (todas las posibles reconexiones de aristas que sean válidas dentro de la heurística realizada) hasta que no se producen más mejoras. Cada mejora se guarda en un vector S_{final} que es el camino mejorado a devolver.

Como las distancias son enteros positivos y las mejoras son enteras, como mínimo se mejora la distancia en una unidad. Por lo tanto, o la distancia llega a cero en cuyo caso ya no habrá mejoras y el algoritmo finaliza su ejecución o podría ser que se llegue a un mínimo local y por lo tanto no pueda seguir mejorandose la distancia, con lo cual el algoritmo también da por finalizada su ejecución.

```

función swap()
     $S_o$  es un Recorrido solución que brinda el algoritmo goloso
    n es cantidad de nodos de  $S_o$ 
    Recorrido  $S_{actual} \leftarrow S_o$                                      O(n)
    entero costoAnterior  $\leftarrow$  calcularCosto( $S_o$ )                  O(1)
    Recorrido  $S_{final} \leftarrow S_o$                                      O(n)
    hayMejora  $\leftarrow$  true                                         O(1)
    Mientras hayMejora hacer                                         ciclo: O(n - longitud solucion optima)
        entero costoActual  $\leftarrow$  -1
        Para cada i de 1 a n hacer                                         ciclo: O(n)
            Para cada j de i+1 a n hacer                                         ciclo: O(n)
                intercambiar posiciones i con j en  $S_{actual}$           O(1)
                optimizarS( $S_{actual}$ )                                    O(n)
                costoActual  $\leftarrow$  calcularCosto( $S_{actual}$ )              O(n)
                si costoActual <> -1  $\wedge$  costoActual < costoAnterior entonces
                    costoAnterior  $\leftarrow$  costoActual                      O(1)
                     $S_{final} \leftarrow S_{actual}$                                 O(n)
                fin si
                intercambiar posiciones i con j en  $S_{actual}$           O(1)
            fin para
        fin para
        si costoActual = -1  $\vee$  costoActual  $\geq$  costoAnterior entonces
            hayMejora  $\leftarrow$  false                                 O(1)
        fin si
    fin ciclo
    devolver  $S_{final}$                                               complejidad total: O( $n^3$ )
fin función

```

función $2opt()$

Recorrido S_o es la solución que brinda el algoritmo goloso
 n es cantidad de nodos de S_o

Recorrido $S_{actual} \leftarrow S_o$

$O(n)$

entero costoAnterior \leftarrow calcularCosto(S_o)

$O(1)$

Recorrido $S_{final} = S_o$

$O(n)$

Mientras hayMejora hacer

ciclo: $O(n - \text{longitud solucion optima})$

entero costoActual $\leftarrow -1$

Para cada i de 1 a n hacer

ciclo: $O(n)$

Para cada j de $i+1$ a n hacer

ciclo: $O(n)$

invertir rango de i a j en S_{actual}

$O(n)$

optimizarS(S_{actual})

$O(n)$

costoActual \leftarrow calcularCosto(S_{actual})

$O(n)$

si $costoActual <> -1 \wedge costoActual < costoAnterior$ entonces

costoAnterior \leftarrow costoActual

$O(1)$

$S_{final} \leftarrow S_{actual}$

$O(n)$

fin si

invertir rango de i a j en S_{actual}

$O(n)$

fin para

fin para

si $costoActual = -1 \vee costoActual \geq costoAnterior$ entonces

$O(1)$

hayMejora \leftarrow false

fin si

fin ciclo

devolver S_{final}

complejidad total: $O(n^3)$

fin función

función $\beta_{opt}()$

Recorrido S_o es la solución que brinda el algoritmo goloso
n es cantidad de nodos de S_o

Recorrido $S_{actual} \leftarrow S_o$

entero costoAnterior \leftarrow calcularCosto(S_o)

Recorrido $S_{final} = S_o$

hayMejora \leftarrow true **Mientras** hayMejora **hacer**

ciclo: $O(n - \text{longitud solucion optima})$

entero costoActual $\leftarrow -1$

Para cada i de 1 a n-3 **hacer**

ciclo: $O(n)$

Para cada j de i+1 a n-2 **hacer**

ciclo: $O(n)$

Para cada k de j+2 a n **hacer**

ciclo: $O(n)$

caso 1:

invertir rango de i a j en S_{actual}

$O(n)$

invertir rango de j+1 a k en S_{actual}

$O(n)$

optimizarS(S_{actual})

$O(n)$

entero costoActual \leftarrow calcularCosto(S_{actual})

$O(n)$

si costoActual $<> -1 \wedge$ costoActual $<$ costoAnterior **entonces**

costoAnterior \leftarrow costoActual

$O(1)$

$S_{final} = S_{actual}$

$O(n)$

fin si

invertir rango de j+1 a k en S_{actual}

$O(n)$

invertir rango de i a j en S_{actual}

$O(n)$

caso 2:

intercambiar rango de i a j con el de j+1 a k en S_{actual}

$O(n)$

optimizarS(S_{actual})

$O(n)$

entero costoActual \leftarrow calcularCosto(S_{actual})

$O(n)$

si costoActual $<> -1 \wedge$ costoActual $<$ costoAnterior **entonces**

costoAnterior \leftarrow costoActual

$O(1)$

$S_{final} = S_{actual}$

$O(n)$

fin si

intercambiar rango de i a j con el de j+1 a k en S_{actual}

$O(n)$

caso 3:

es igual al caso 2 pero además invirtiendo el rango i a j

$O(4*n)$

caso 4:

es igual al caso 2 pero además invirtiendo el rango j+1 a k

$O(4*n)$

fin para

fin para

fin para

fin ciclo

si costoActual = -1 \vee costoActual \geq costoAnterior **entonces**

hayMejora \leftarrow false

$O(1)$

fin si

devolver S_{final}

complejidad total: $O(n^4)$

fin función

```

función optimizarS( $S_o$ )
  Mientras  $back(S_o).tipo = pokeparada$  hacer
    pop_back( $S_o$ )
  fin ciclo
fin función

función calcularCosto(Recorrido camino)
  entero costo = 0 O(1)
  entero capacidadParcial = 0 O(1)
  Para cada  $i$  desde 2 hasta /camino/ hacer ciclo: O(n)
    si pasoPosible(camino[i], capacidadParcial) entonces guarda: O(1)
      <entero, entero> pOrigen
      <entero, entero> pDestino
      entero origen  $\leftarrow$  camino[i-1] O(1)
      entero destino  $\leftarrow$  camino[i] O(1)
      bool destinoEsPP  $\leftarrow$  false O(1)
      si origen < cantGyms entonces O(1)
        pOrigen  $\leftarrow$  gimnasiosArr[origen].coord
      fin si
      de lo contrario O(1)
        pOrigen  $\leftarrow$  pokeParadasArr[origen-cantGyms]
      fin si
      si destino < cantGyms entonces O(1)
        pDestino  $\leftarrow$  gimnasiosArr[destino].coord
      fin si
      de lo contrario O(1)
        pDestino  $\leftarrow$  pokeParadasArr[destino-cantGyms]
        destinoEsPP  $\leftarrow$  true O(1)
      fin si
      costo  $\leftarrow$  costo + distanciaEuclidea(pOrigen, pDestino) O(1)
      si destinoEsPP entonces O(1)
        capacidadParcial += 3
        si capacidadParcial > capMochila entonces O(1)
          capacidadParcial  $\leftarrow$  capMochila
        fin si
      fin si
      de lo contrario O(1)
        capacidadParcial  $\leftarrow$  capacidadParcial - gimnasiosArr[destino].poder
      fin si
    fin si
    de lo contrario O(1)
      devolver -1
    fin si
  fin para complejidad total: O(n)
  devolver costo
fin función

```

```

función pasoPosible(entero destino, entero capacidadParcial
    entero poderGym ← 0
        si destino < cantGyms entonces
            poderGym ← gimnasiosArr[destino].poder
        fin si
        si poderGym = 0 ∨ capacidadParcial > poderGym entonces
            devolver true
        fin si
        devolver false
    fin función

```

complejidad total: O(1)

Detalles de los algoritmos

- Justificación de la cota del ciclo principal: El algoritmo comienza con una solución de n nodos. Cada mejora en el peor de los casos se realiza en una unidad, ya que se trabaja con distancias enteras. Como se busca llegar al óptimo global, en el peor de los casos habrá $O(n\text{-longitud de la solución óptima})$ iteraciones para el ciclo principal.
- Recorrido = Lista de id 's
- $n = |S_o|$
- Todos los id 's en un Recorrido son enteros entre 1 y $M + N$, con M la cantidad de gimnasios y N la cantidad de pokeparadas.
- Los primeros M indices corresponden a los gimnasios y los restantes N a pokeparadas.
- Gimnasio = $\langle \langle$ entero x, entero y $\rangle, \text{entero poder} \rangle$
- PokeParada = $\langle \text{entero x, entero y} \rangle$
- gimnasiosArr es un arreglo de Gimnasio
- pokeParadasArr es un arreglo de PokeParada

Podemos ver que todos los algoritmos iteran sobre la solución S_o , que en el peor caso puede contener todos los nodos del mapa, osea, $n = M + N$.

Las operaciones *invertir rango* o *intercambiar rango* en el peor caso serán realizadas sobre los n nodos de S_o .

La operación costo es $O(n)$ ya que requiere recorrer S_o hasta la última posición observando si un movimiento es válido. Recordemos que la validez de un movimiento se observa cuando se avanza hacia un gimnasio. Este movimiento será válido si y solo si se puede vencer al gimnasio. Esto último es un chequeo que puede realizarse en tiempo constante.

Luego, realizar búsquedas locales con las vecindades planteadas es, en el peor caso, de complejidad polinómica.

5.3 Experimentos y conclusiones

5.3.1 Test y performance de los algoritmos

En el ejercicio dos se introdujeron dos familias nuevas: **Random** y **Gimnasios por grupo** que fueron explicadas apropiadamente. Como la idea es estudiar en promedio que sucede con las distancias al crecer la cantidad de elementos total del mapa (pokeparadas + gimnasios), las experimentaciones de este test se centrarán en tratar de mejorar esas distancias mediante las heurísticas introducidas en este punto ($2 - opt$ y *Swap*).

Para realizar los tests se tomaron dos rangos de tamaños:

- Rango 1: 5 a 15 elementos (pokeparadas + gimnasios)
- Rango 2: 70 a 470 en intervalos de 50 elementos.

Para el Rango 1 se tomó cinco veces el total de elementos como cantidad de instancias aleatorias por cada tamaño y para el Rango 2 se tomó un 10% del total de elementos como cantidad de instancias aleatorias.

Se realizan las búsquedas locales promediando los tiempos y distancias para todas las instancias de cada tamaño. Y se obtienen los porcentajes de mejora relativos a cada tamaño con respecto al promedio de las distancias del algoritmo goloso.

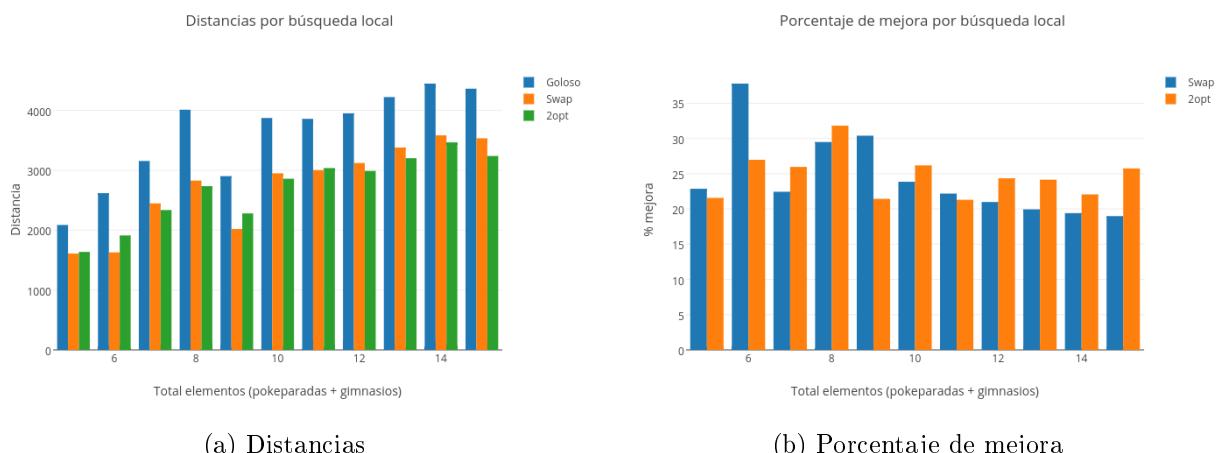
Al igual que el error relativo, el porcentaje de mejora también se calcula en base a los promedios de las distancias. En este caso el promedio de las distancias del algoritmo goloso y el promedio de las distancias de las búsquedas locales. Sean $promedio_{greedy}$ y $promedio_{ls}$ respectivamente. El porcentaje de mejora relativo se calcula de la siguiente manera:

$$\%Mejora_{rel} = \frac{promedio_{ls}}{promedio_{greedy}} * 100 \quad (3)$$

En el rango 1 podrá calcularse el error relativo para comparar la solución exacta con las heurísticas de búsqueda local.

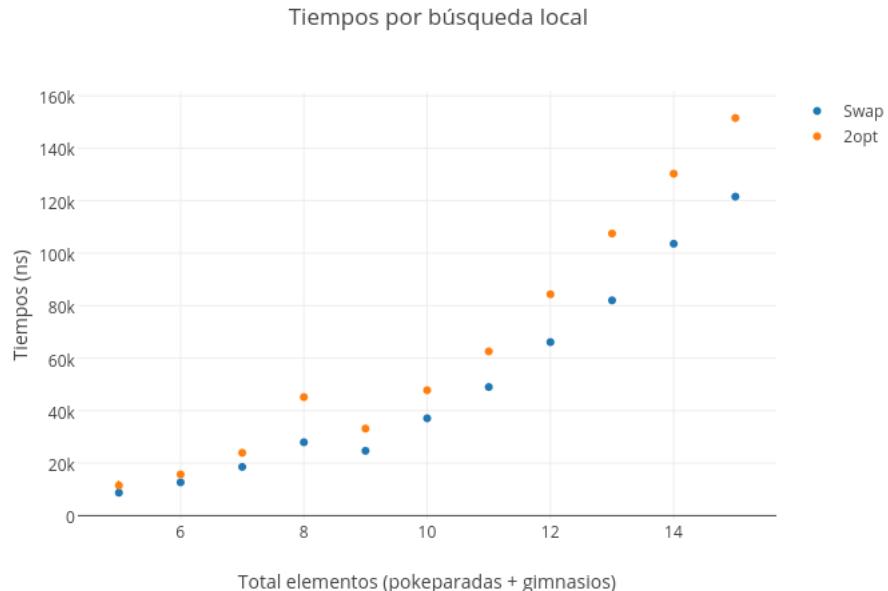
Random

Comenzando con el Rango 1 puede verse en los siguientes gráficos las mejoras realizadas por las búsquedas locales con respecto al algoritmo goloso. En promedio para cada tamaño $2 - opt$ logra entre un 3% y un 4% de mejora más que *Swap*. Las mejoras se observan entre un 20% y un 27% para la mayoría de los casos.



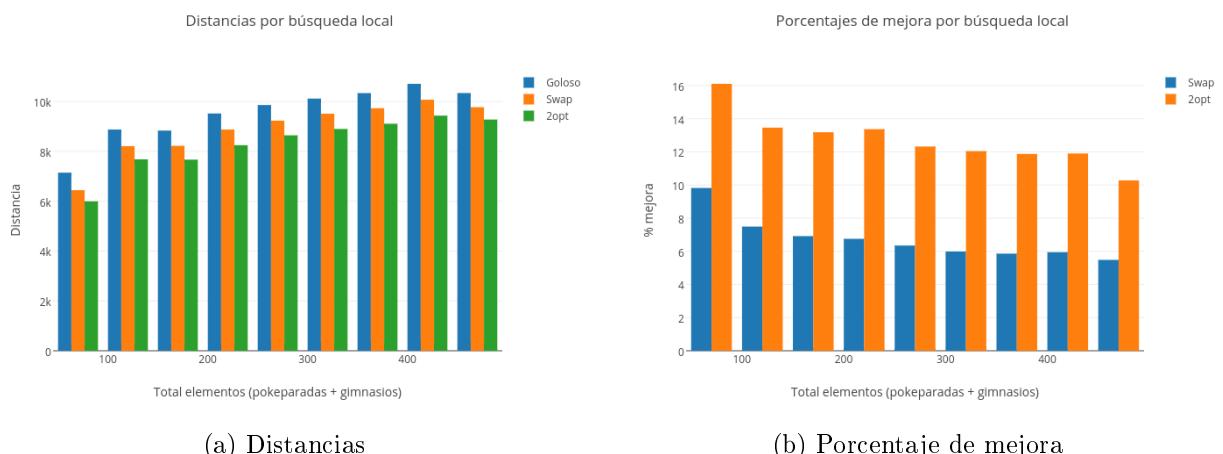
(a) Distancias

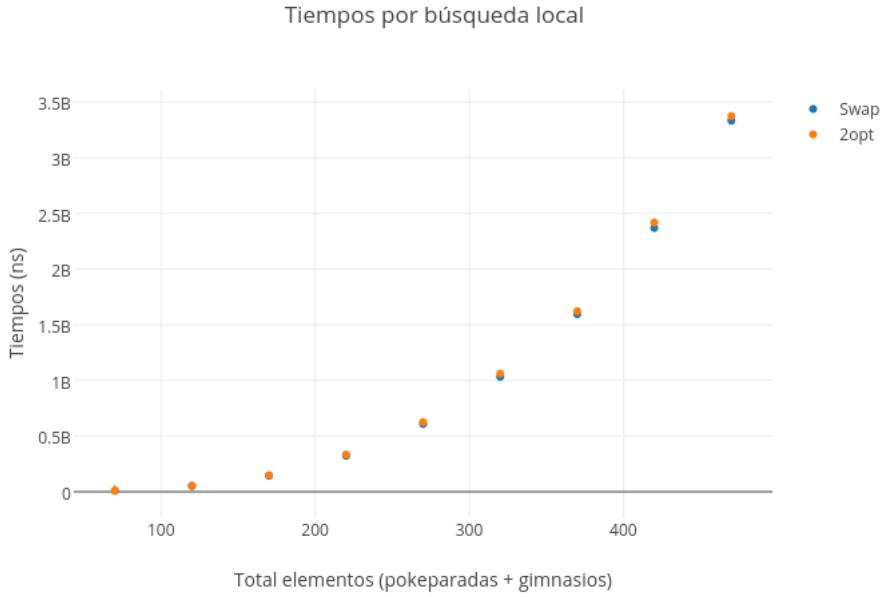
(b) Porcentaje de mejora



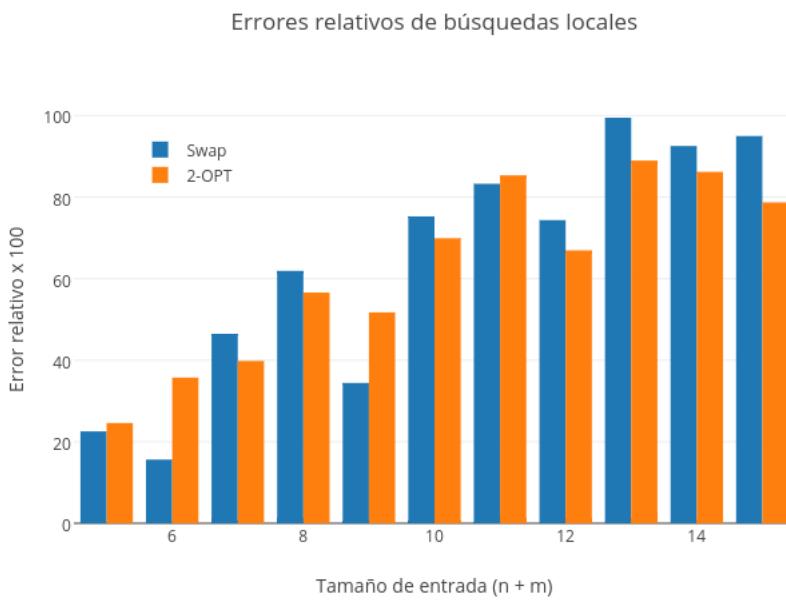
Tiempos

En relación a los tiempos *Swap* insume aproximadamente un entre un 15% y un 20% menos de tiempo. Esto no es considerable como para no tomar las mejoras realizadas por *2-opt* como un buen resultado en relación calidad-performance.





Tiempos



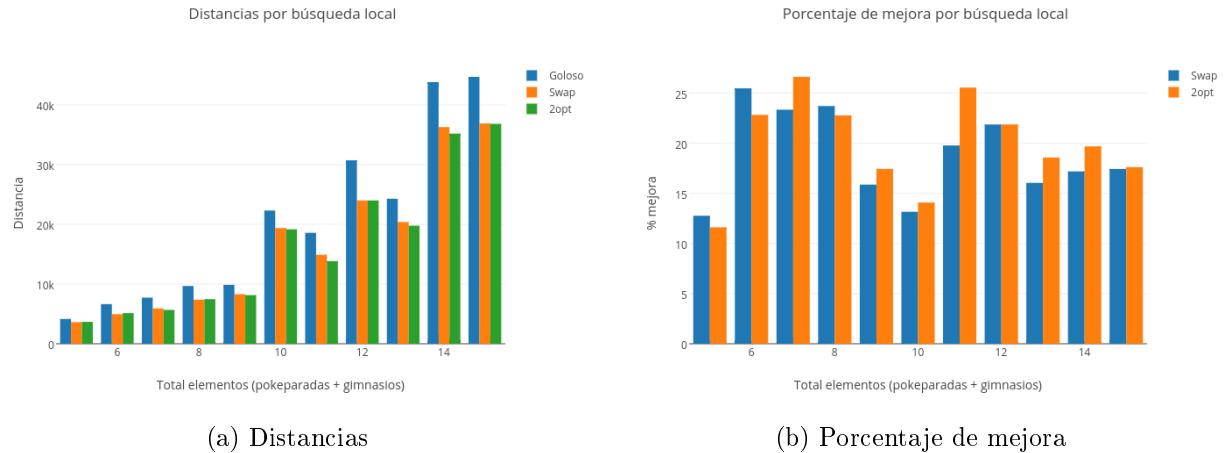
Error Relativo de cada búsqueda

Para el Rango 2, podemos observar que a medida que el tamaño crece se observa una ligera tendencia de disminución en la mejora realizada por las búsquedas locales. Esto puede deberse al hecho de que cuanto más grande es la instancia más local se vuelve la solución realizada por el goloso y así mismo la búsqueda local solo llega a mínimos locales. Se observa igualmente un buen desempeño de la búsqueda local $2 - opt$ pero con mejoras entre un 12% y un 14% en comparación al Rango 1. A su vez como se mencionó, para instancias mayores este porcentaje disminuye llegando a estar incluso por debajo del 12%

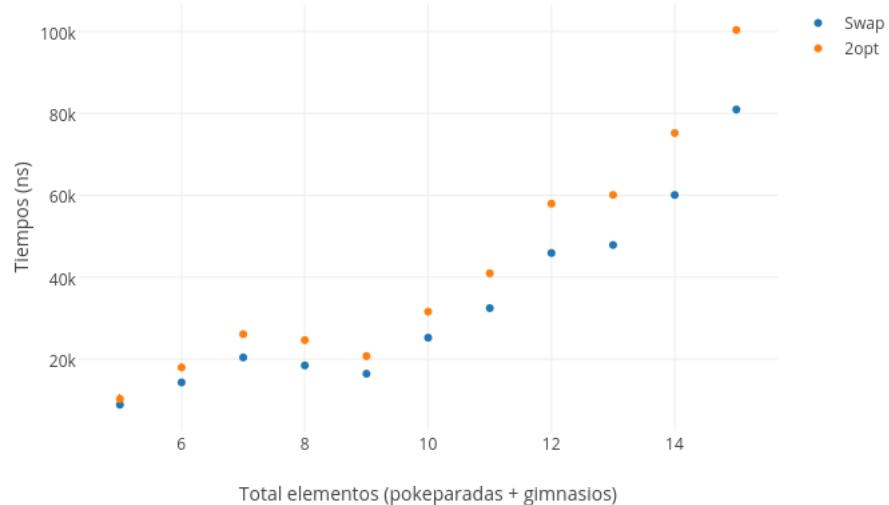
Con respecto a los errores relativos de las búsquedas enunciadas, teniendo en cuenta la cantidad de instancias tomadas para cada uno de los tamaños, podemos concluir que, para tamaños grandes (tomando $n + m \geq 12$), 2-OPT presenta una ventaja considerable. Mientras que para tamaños pequeños Swap tiene un error inferior en la mayoría de los casos. Como la cantidad de instancias de los casos chicos son pocos en relación a los grandes, y nuestro interés se centra cuando $n+m$ tiene un valor grande, podemos concluir que la heurística 2-OPT presenta una mejora superior a la otra.

Gimnasios por grupos

Como ya sabemos, esta familia intenta organizar los gimnasios en grupos de hasta cuatro poderes distintos. Veamos que sucede al incrementar la cantidad de elementos en el mapa.



Tiempos por búsqueda local

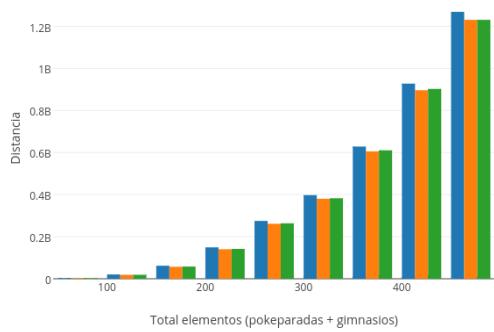


Tiempos

Podemos observar en el Rango 1 como a medida que las instancias crecen en tamaño aumenta considerablemente la distancia del algoritmo goloso y las mejoras realizadas por ambas búsquedas locales son muy parecidas estando ambas por encima del 15% en casi todos los casos. Para instancias pequeñas se hace particularmente difícil decidir que búsqueda local es la ganadora.

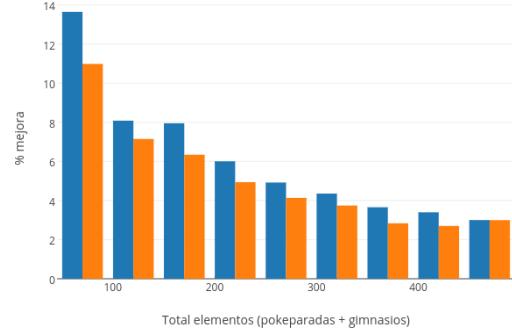
En relación a los tiempos se observa que *Swap* se encuentra entre un 15% y un 20% por debajo de *2-opt*. En relación a las mejoras obtenidas es sugerente que *Swap* parece una buena opción para casos particularmente pequeños en relación calidad-performance.

Distancias por búsqueda local



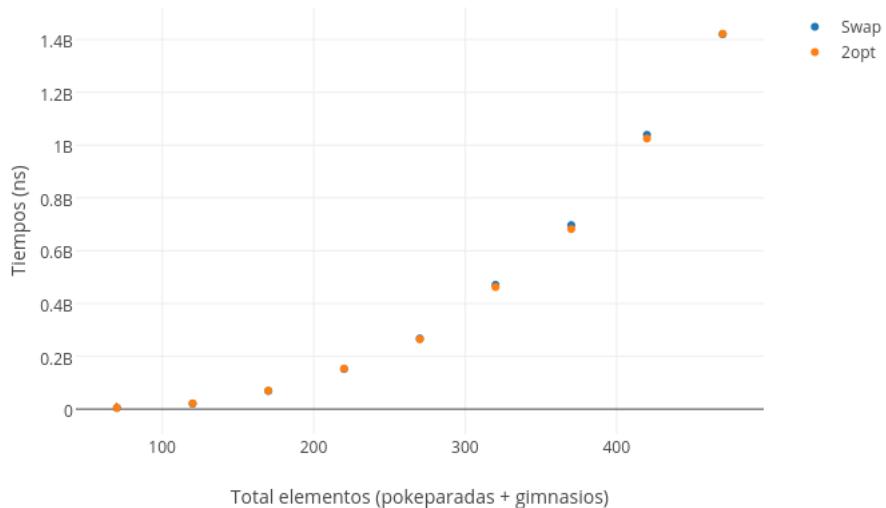
(a) Distancias

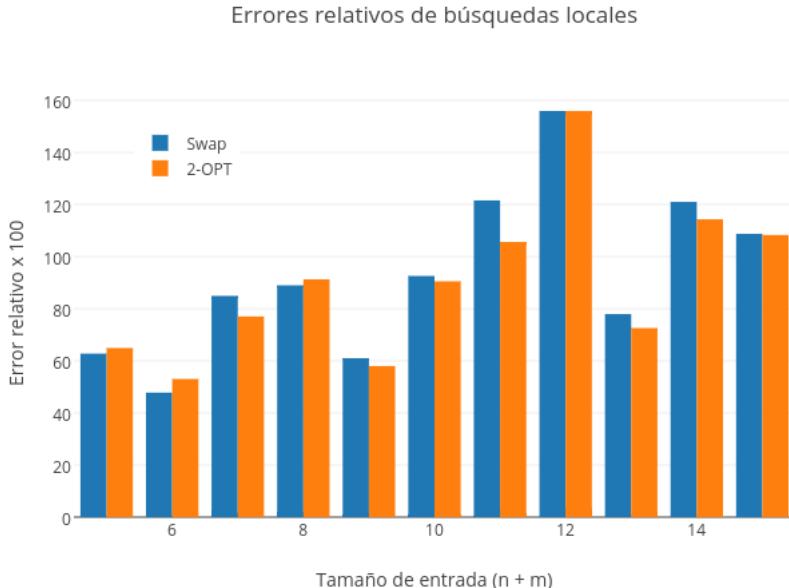
Porcentaje de mejora por búsqueda local



(b) Porcentaje de mejora

Tiempos por búsqueda local

*Tiempos*



Error Relativo de cada búsqueda

Para el Rango 2 se sigue observando un incremento extremadamente alto en las distancias obtenidas por el goloso. Y a su vez las mejoras realizadas por ambas búsquedas locales disminuyen conforme aumenta el tamaño. En principio se observa que no hay diferencias para instancias de 470 elementos. Debido a la discretización utilizada, este patrón podría repetirse incluso a partir de los 400 elementos.

Creemos que las distancias observadas son particularmente altas debido a como están organizados los gimnasios. Al estar las pokeparadas distribuidas por todo el mapa y los gimnasios más agrupados, el algoritmo goloso deberá recorrer enormes distancias para vencer gimnasios y volver a buscar pokeparadas en la mayoría de los casos. Además debemos recordar que el algoritmo goloso no hace particular uso de la mochila ya que su premisa es siempre vencer un gimnasio cuando sea posible.

Con respecto a la disminución en los porcentajes de mejora puede observarse lo mismo que en la familia *Random*, a medida que la instancia crece, la solución se hace cada vez más local y por lo tanto las mejoras tienden a ser mínimos locales.

Los tiempos de ambas búsquedas locales son relativamente iguales. Teniendo en cuenta la mínima ventaja de *Swap* en cuanto a mejoras, podríamos decir que ambas búsquedas locales funcionan bien para intancias grandes.

Refiriéndonos a los errores relativos de las búsquedas enunciadas, no será posible concluir cual será mejor que la otra ya que ambas presentan errores similares, no se obtiene una marcada diferencia en varios casos entre ambas. Concluimos que para este tipo de caso, no existirá una preferencia entre las dos heurísticas en base a un mejor o peor error relativo.

Será particularmente importante observar en el siguiente ejercicio cuanto puede mejorar la distancia la meta heurística que será presentada, ya que queda bastante claro que los resultados del algoritmo goloso para este tipo de familia pueden ser extramadamente malos.

6 Ejercicio 4

6.1 Explicación de resolución del problema

Como su nombre lo indica, búsqueda local analiza una vecindad local a la solución inicial S_0 . Por lo que generalmente, la mejora obtenida puede no ser global, si no, la mejor solución dentro de la vecindad analizada.

Para salir de un óptimo local, existen meta heurísticas que pueden o no proveer una mejor solución observando otras vecindades, y en algunos casos acercarse lo suficiente u obtener el óptimo global. Una de ellas es la elegida para este informe, denominada, *tabú search*.

La idea de esta meta heurística es ir moviéndose por las vecindades adyacentes a una vecindad analizada. Es decir, las vecindades de las soluciones que conforman una vecindad. Pero no todas ellas, si no, la vecindad de una solución elegida que cumpla con ciertos atributos, o mejor dicho, que no posea ciertos atributos o características. Esto es así, dado que se tiene que buscar una manera de descartar soluciones que no se consideren adecuadas para ser analizadas, si no, caeríamos en el problema de backtracking, donde se consideran todas las posibilidades, lo cual, puede ser impracticable.

Los atributos elegidos como no adecuados para elegir una solución son los denominados atributos *tabú*. Existen muchas posibilidades según el problema estudiado. Para el problema del maestro pokemon elegiremos como atributos *tabú* las aristas que sean modificadas al moverse de una solución a otra. También podría tomarse como *tabú* las aristas nuevas en la solución y ver si se obtienen mejores resultados (Para este informe no será tenida en cuenta esta posibilidad por cuestiones de tiempo). Por lo tanto, se define un conjunto que alojará atributos *tabú*.

Los métodos para encontrar vecindades serán los mismos analizados en el ejercicio tres. Es decir, a través de las búsquedas locales estudiadas: *swap*, *2-opt* (*3-opt* fué descartada para esta re-entrega por cuestiones de tiempos). Solo que para este algoritmo se filtrarán aquellos recorridos que no sean válidos. Luego una vecindad $V(s)$ para una solución s , será un conjunto de recorridos de soluciones válidas para el problema.

Debido a que la memoria tiene un límite, y los problemas podrían ser extremadamente grandes, se suele definir lo que se denomina *tenor tabú* que es el tamaño máximo que el conjunto *tabú* tiene para alojar atributos. Cuando el tamaño máximo es alcanzado, se tiene que determinar una manera de desalojar atributos para obtener espacio libre que pueda ser usado luego. El motivo es tener una lista de tamaño acotado, pero que sea dinámica en contenido de atributos a lo largo de una corrida del algoritmo, dado que si no, al alcanzarse el tamaño máximo, la lista dejaría de crecer y los atributos dejarían de cambiar, acotando el universo de posibles soluciones a la unión de algunas vecindades. Los atributos que serán desalojados serán aquellos que tengan más tiempo dentro del conjunto, por lo que además, el conjunto *tabú* tendrá la característica de poder contener esa información y funcionar internamente como una pila. La cantidad de atributos desalojados será determinada por la cantidad de atributos que se quiera alojar en el conjunto. En el peor caso, todos los atributos serán nuevos.

Además, si el tenor definido en el punto anterior es lo suficientemente grande, indefectiblemente, en algún punto tendremos todos los atributos alojados. Por lo cual tendremos que tomar alguna decisión para elegir una nueva solución en la vecindad analizada. Esta decisión se conoce como función de aspiración $A(V(s)) \rightarrow s'$ que dada una vecindad $V(s)$ a una solución s , determina qué solución tomar. La decisión puede tomarse en base a la cantidad de atributos *tabú* que posee la solución analizada. Luego puede elegirse la más tabú o la menos tabú. Para este informe se elige la estandar que es la menos tabú.

Algo que hasta aquí no fue definido es el criterio de parada. Dado que el algoritmo se mueve entre vecindades sin marcar soluciones como ya visitadas, podría darse el caso de que se esté iterando sobre un conjunto de soluciones. Para poder finalizar el algoritmo en cierto punto, se utilizan diferentes estrategias. La más usual es cantidad de iteraciones límite, pero existe otra posibilidad y es tomar cantidad de iteraciones sin mejora. Esta última funciona correctamente dado que como las distancias son enteros positivos y las mejoras son enteras, el mínimo posible que puede disminuir la distancia es en una unidad. Además el algoritmo siempre compara contra la mejor solución en cada iteración, por lo tanto, o bien no se producen mejoras y el algoritmo finaliza en una cantidad x de iteraciones o bien la distancia llega a cero y deja de disminuir finalizando en ambos casos la ejecución del algoritmo.

Hasta aquí pudimos describir las motivaciones para usar *tabú search* y de que manera funciona a grandes rasgos. En lo que sigue podremos ver el pseudo código de la implementación realizada y luego abordaremos los casos de tests para luego analizar los resultados. La idea será trabajar sobre los casos de entrada del ejercicio 2 y ver que optimizaciones logra *tabú search* con respecto a las soluciones provistas por el algoritmo goloso, las búsquedas locales y cuando sea posible contrastar los resultados contra la solución exacta para ver cuánto pudimos acercarnos.

6.2 Pseudocódigo

Como comentario inicial, se presenta el algoritmo *Tabu Search* con ambos criterios de parada en el ciclo principal (cantidad de iteraciones límite y cantidad de iteraciones sin mejora) ya que ambos pueden aplicarse a la vez o por separado que es como serán estudiados en la experimentación.

El algoritmo itera sobre una solución inicial S_0 que es un vector con los *id's* de las pokeparadas y gimnasios como fue definido en el ejercicio tres. 5.2 Se utiliza el vector *solucionActual* para guardar el recorrido que se utiliza para recorrer vecindades. Mientras que *mejorSolucion* guarda el mejor recorrido encontrado en todas las vecindades analizadas y será este el encargado de hacer valer la condición de corte de cantidad de iteraciones sin mejora.

El algoritmo procede a calcular la vecindad de *solucionActual* filtrando aquellos recorridos no válidos para el problema en cuestión (las condiciones de validez de una solución fueron dadas en el ejercicio dos) y luego busca el mejor candidato dentro de esta vecindad, que será aquella con menor costo y que no sea un recorrido marcado como tabú (es decir que previamente ya fue tomado).

Si no obtuvo un candidato, procede a utilizar la función de aspiración, que será elegir aquel recorrido en el vecindario que sea menos tabú.

En cualquier caso, siempre se obtienen las aristas modificadas para llegar a este recorrido dentro de la vecindad (esto depende de la búsqueda local utilizada) y se agregan las mismas a la lista tabú.

Una vez hecho esto se actualiza el *mejorCosto* y la *mejorSolucion* si el recorrido encontrado mejora el *mejorCosto* actual. Si esto no sucede, se aumenta en una unidad la cantidad de iteraciones sin mejora.

El algoritmo finaliza cuando se alcanza la cantidad de iteraciones límite o la cantidad de iteraciones sin mejora límite. Una u otra dependen de la elección del usuario.

función *tabuSearch()*

S_o es la solución provista por el algoritmo greedy

ConjuntoTabu attributosTabu $\quad \quad \quad O(n)$

Recorrido mejorSolucion $\leftarrow S_o$ $\quad \quad \quad O(n)$

Recorrido solucionActual $\leftarrow S_o$ $\quad \quad \quad O(n)$

entero mejorCosto \leftarrow calcularCosto(mejorSolucion) $\quad \quad \quad O(n)$

entero iter = 0

entero MaxIter = cantidad pokeparadas+cantidad gimnasios

entero TenorTabu = MaxIter

entero MaxNoMejora = 4

Para cada *iter < MaxIter* \wedge *entero noMejora < maxNoMejora* **hacer** $\quad \quad \quad$ ciclo: $O(MaxIter)$

- Recorrido mejorCandidato $\quad \quad \quad O(1)$
- Lista<Arista> aristasModificadas $\quad \quad \quad O(1)$
- entero costoMejorCandidato $\leftarrow -1$ $\quad \quad \quad O(1)$
- Conjunto<Recorrido, Lista<Arista>> vecindadActual \leftarrow vecindadFiltrada(solucionActual) $\quad \quad \quad O(n^4)$
- Para cada** *par en vecindadActual* **hacer** $\quad \quad \quad$ Ciclo: $O(n^4)$
 - Recorrido candidatoActual \leftarrow par.first $\quad \quad \quad O(n)$
 - entero costoActual \leftarrow calcularCosto(candidatoActual) $\quad \quad \quad O(n)$
 - costoMejorCandidato \leftarrow calcularCosto(mejorCandidato) $\quad \quad \quad O(n)$
 - si** *costoActual < costoMejor* \vee (*!tabuCount(atributosTabu, candidatoActual)*) \wedge (*costoActual < costoMejorCandidato* \vee *costoMejorVecino = -1*) **entonces** $\quad \quad \quad$ Condicion: $O(n)$
 - aristasModificadas \leftarrow par.second $\quad \quad \quad O(1)$
 - mejorCandidato \leftarrow candidatoActual $\quad \quad \quad O(n)$
 - fin si**
- fin para** $\quad \quad \quad$ Total del ciclo: $O(n^5)$
- si** *no se encontro mejorCandidato* **entonces** $\quad \quad \quad$ Condicion: $O(1)$
 - <Recorrido, Lista<Arista>> menosTabu \leftarrow funcionAspiracion(atributosTabu, vecindadActual) $\quad \quad \quad O(n^5 * log(TenorTabu))$
 - mejorCandidato \leftarrow menosTabu.first $\quad \quad \quad O(n)$
 - aristasModificadas \leftarrow menosTabu.second $\quad \quad \quad O(1)$
- fin si**
- costoMejorCandidato = calcularCosto(mejorCandidato); $\quad \quad \quad O(n)$
- solucionActual \leftarrow mejorCandidato $\quad \quad \quad O(n)$
- si** *costoMejorCandidato < mejorCosto* **entonces** $\quad \quad \quad$ Condicion: $O(1)$
 - mejorSolucion \leftarrow mejorCandidato $\quad \quad \quad O(n)$
 - mejorCosto \leftarrow costoMejorCandidato $\quad \quad \quad O(1)$
 - noMejora=0
- fin si**
- de lo contrario**
- noMejora+=1
- fin si**
- Para cada** *Arista a en aristasModificadas* **hacer** $\quad \quad \quad$ Ciclo: $O(1)$
 - atributosTabu.push(a) $\quad \quad \quad O(log(TenorTabu))$
- fin para**
- Mientras** *|atributosTabu| > TenorTabu* **hacer** $\quad \quad \quad$ Ciclo: $O(1)$
 - atributosTabu.pop() $\quad \quad \quad O(log(TenorTabu))$

Complejidad final: $O(\text{MaxIter} * (n^5 * \log(\text{TenorTabu}) + \text{TenorTabu})) = O(N + M * (n^5 * \log(N + M) + N + M)) \subset O(N + M * ((N + M)^5 * \log(N + M) + N + M))$
 Siendo N = cantidad pokeparadas y M = cantidad gimnasios.

En general a la hora de implemetar y experimentar, serán tomados porcentajes de $N + M$ para la cantidad de iteraciones límite y el tenor tabú ya que pueden ser valores muy grandes.

Detalles del algoritmo

- Justificación de la cota de complejidad del ciclo principal: El algoritmo finaliza si se producen *maxNoMejora* cantidad de iteraciones sin mejora de *costoInicial* o si se alcanza *MaxIter*.
- En el peor caso $n = N + M$ es decir, que la solución inicial del algoritmo tiene todas las pokeparadas y gimnasios del mapa.
- $\text{MaxIter} = (N + M)\%$ y $\text{TenorTabu} = (N + M)\%$
- Arista = < Punto a , Punto b >
- Punto = < entero x, entero y >
- La función *costoTotal* ya fue definida en el apartado del algoritmo del ejercicio 3.
- La función *vecindadFiltrada* devuelve una lista de tuplas. Cada tupla posee una de las soluciones vecinas (obtenida a partir de algún movimiento: 2opt, 3opt o swap) y las aristas que se modificaron para llegar a esa solución. Las soluciones han sido previamente "filtradas"; es decir, una solución solo puede estar en esta lista si es una solución válida para nuestro problema. Generar vecindades 2opt y swap tiene complejidad $O(n^3)$, en cambio las vecindades 3opt toman $O(n^4)$. La cantidad de soluciones de cada vecindad está acotada de la misma manera que sus complejidades. En peor caso, se encontró una solución válida en cada iteración.
- La función *funcionAspiracion* devuelve dada una vecindad y los atributos tabú, el recorrido que menos atributos tabú tenga y además las aristas modificadas para obtener ese recorrido. Implica iterar sobre la vecindad y sobre los atributos tabú.
- La función *tabuCount* sirve para decidir si una solución es tabú o no y cuenta cuantos atributos tabú posee dando de esta manera una medida de "cuán tabú" es una solución. Se itera por las aristas de la solución en tiempo lineal.
- *atributosTabu* está implementado sobre el set proveido por la STL de c++. Se observa una estructura de árbol rojo-negro que puede buscar, insertar y borrar en un tiempo $O(\log(\text{TenorTabu}))$, y se mantiene la cantidad de elementos de este por debajo de *TenorTabu*.

6.3 Análisis de complejidades

6.4 Experimentos y conclusiones

6.4.1 Test y performance del algoritmo

Queremos observar que tan bueno es el algoritmo propuesto en la práctica. Para estudiar el desempeño del algoritmo utilizaremos las familias **Random** y **Gimnasios por grupos**. La idea, además,

es tratar de encontrar la configuración de Tabú Search ideal para que en promedio el algoritmo resuelva la mayoría de los casos eficientemente. Para lograr esto, se experimenta variando distintos parámetros del algoritmo para cada entrada.

Los parámetros de estudio serán:

- **Cantidad de iteraciones:** Tenemos que observar en promedio, cuantas iteraciones conviene tomar para obtener un buen resultado.
- **Tenor tabú:** Tenemos que observar que sucede al aumentar el tenor tabú, y hasta cuánto es conveniente hacerlo para obtener un buen resultado.

Debemos aclarar, que tenor tabú fué establecido en cantidad de pokesparadas mas gimnasios. Este valor puede ser relativamente grande, pero cuanto más grande, más tiempo de resolución es necesario, con lo cual al final no es una cuestión de memoria, si no, una cuestión de tiempo de computo necesaria para correr el algoritmo. También se trabajó probando diferentes configuraciones de atributos tabú y función de aspiración, obteniendo en todos los casos los mismos resultados, por lo que al final se decidió dejar prefijado el estandar para tabú search que es aristas viejas como atributos tabú y solución menos tabú para la función de aspiración.

Se tomaron 20 mediciones por cada tipo de test y se tomó una media alfa podada de las mismas con $\alpha = 0.5$ de manera de podar un 25% de los datos a cada lado. De esta forma se reduce la posibilidad de outliers en las muestras consideradas.

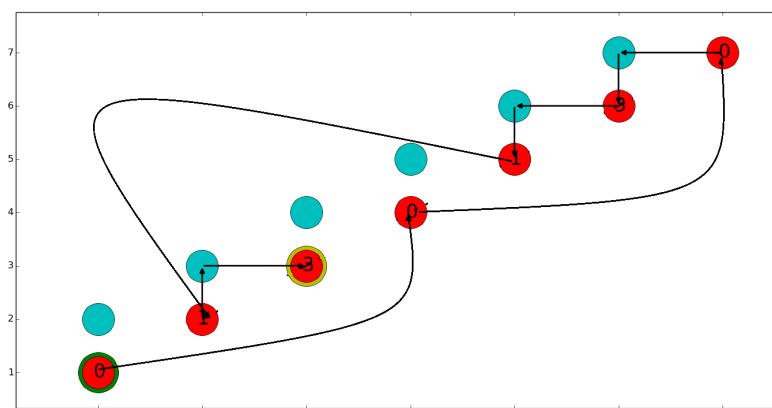
Como fue enunciado en incisos anteriores de este trabajo las 4 familias en las que la heurística golosa no otorga una solución óptima son:

1. Familia 4
2. Familia 6
3. Familia 7
4. Familia 8

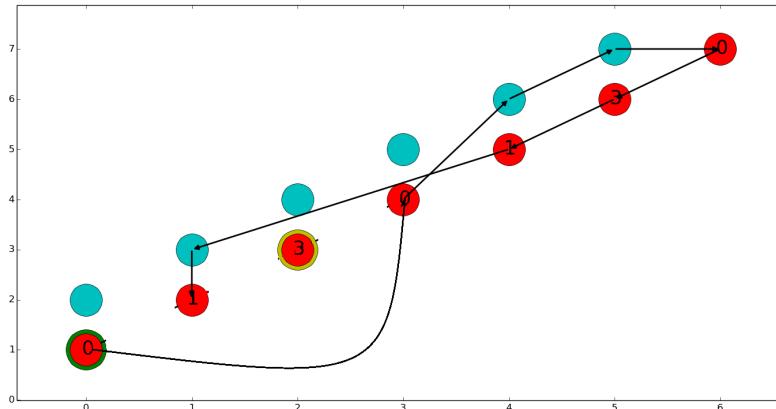
Estas serán las únicas a ser analizadas, ya que el resto son el resultado exacto o no tienen solución, con lo cual no aportan mayor relevancia.

Familia 4

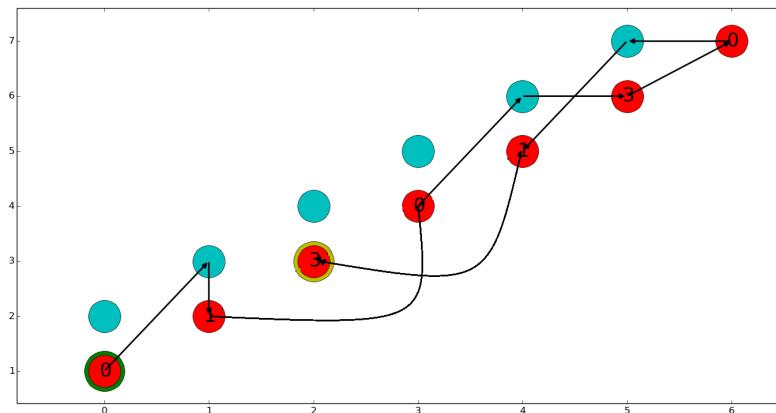
Veamos algunos resultados de aplicar cada versión de tabú search:



Solución Golosa



Solución 2-OPT



Solución 3-OPT

Nos parece interesante comparar los resultados de mejora y tiempo de las búsquedas locales y tabú search que utilicen la misma vecindad, para ver si tabú search mejora lo logrado por la búsqueda local o empeora el resultado. Esto será realizado para cada familia mencionada.

Veamos como se comporta tabú 2-OPT con respecto a la heurística de búsqueda local 2-OPT dentro de la familia 4:

Busqueda 2-OPT y TABU 2-OPT

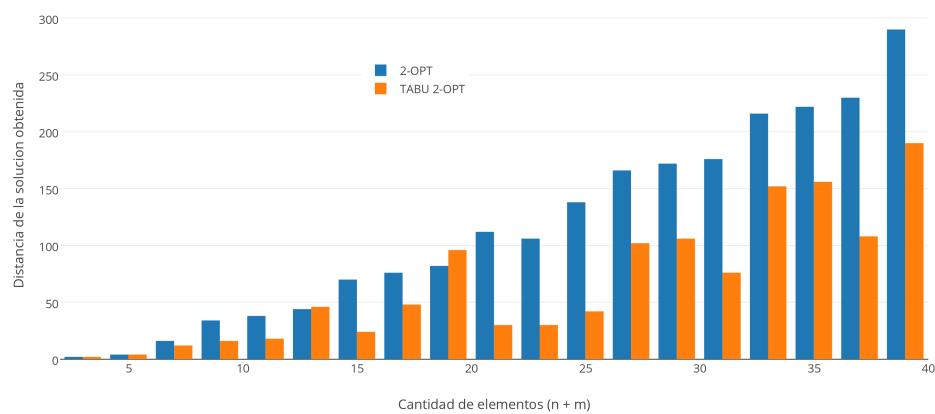


Gráfico 4.1 - 2-OPT vs Tabu 2-OPT sobre Familia 4

En cuanto a tiempo insumido vemos lo siguiente:

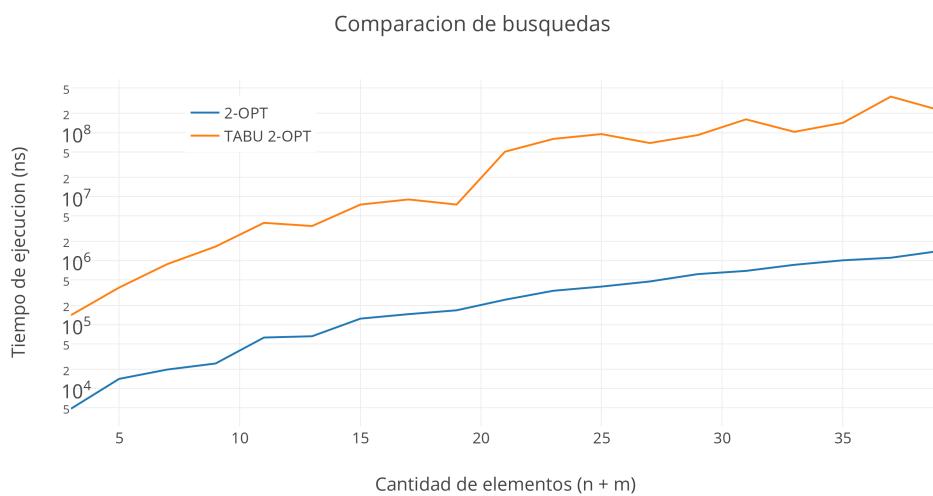


Gráfico 4.2 - 2-OPT vs Tabu 2-OPT sobre Familia 4

Aplicando 3-OPT:

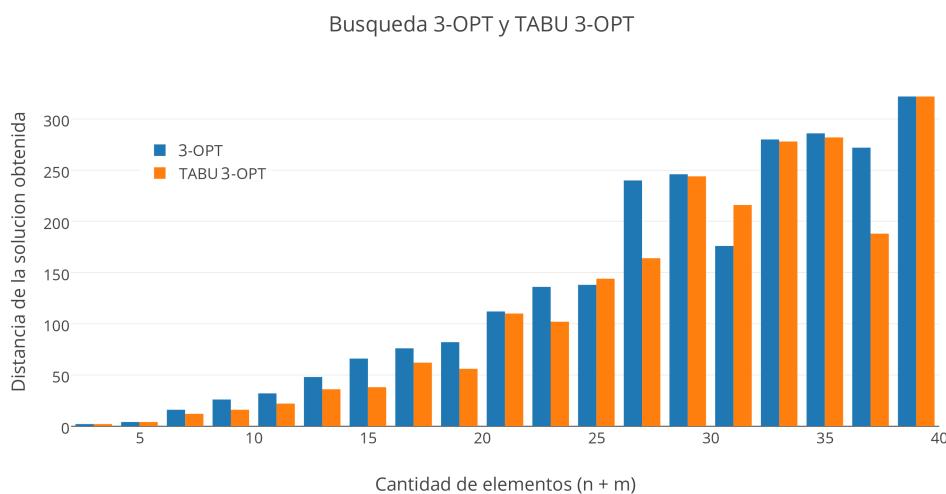


Gráfico 4.3 - 3-OPT vs Tabu 3-OPT sobre Familia 4

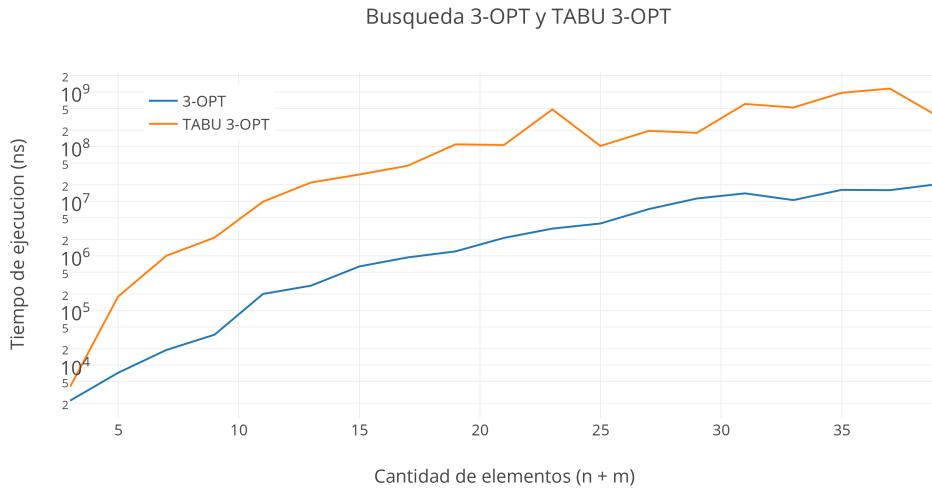


Gráfico 4.4 - 3-OPT vs Tabu 3-OPT sobre Familia 4

Podemos observar que tabú search 2-OPT mejora lo realizado por la heuristica de búsqueda local 2-OPT. Los tiempos insumidos para lograrlo son elevados, pero en relación a la mejora es un resultado aceptable en la práctica. No así tabú search 3-OPT que apenas mejora lo realizado por búsqueda local 3-OPT y requiere tiempos elevados de corrida del algoritmo.

Para decidir que vecindad es mejor utilizar en tabú search, se comparará conjuntamente el tiempo de ejecución con la calidad de la solución. Para esta última tendremos en cuenta que los algoritmos, de devolver un resultado, serán válidos: esto quiere decir que cuanto menor distancia recorran las soluciones, mejor serán las mismas:

Las soluciones obtenidas fueron las siguientes:

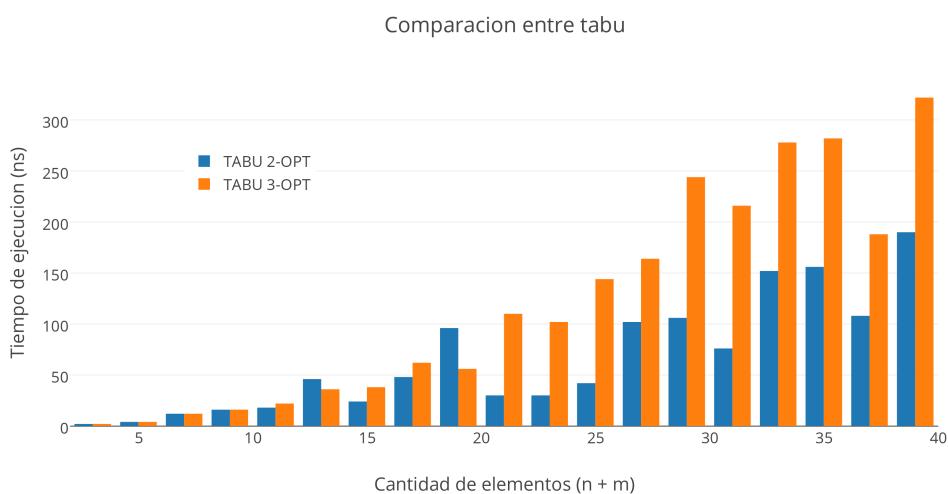


Gráfico 4.5 - Tabu 2-OPT vs Tabu 3-OPT sobre Familia 4

En cuanto a tiempo insumido vemos lo siguiente:

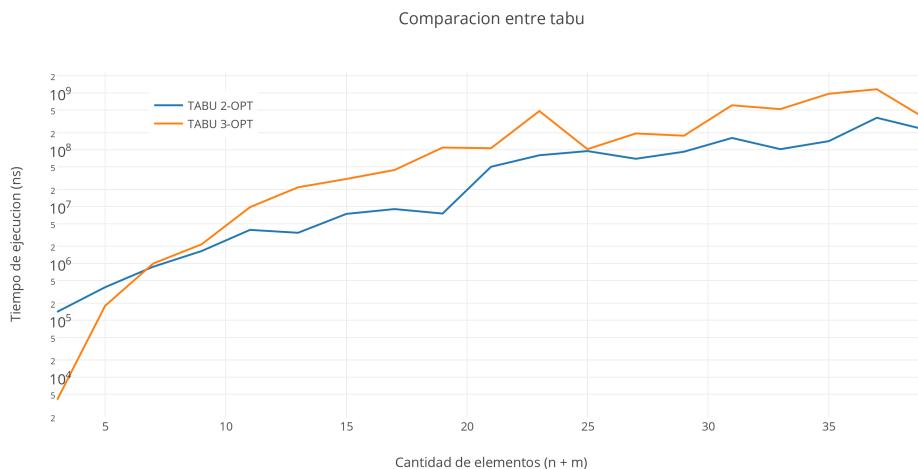


Gráfico 4.6 - Tabu 2-OPT vs Tabu 3-OPT sobre Familia 4

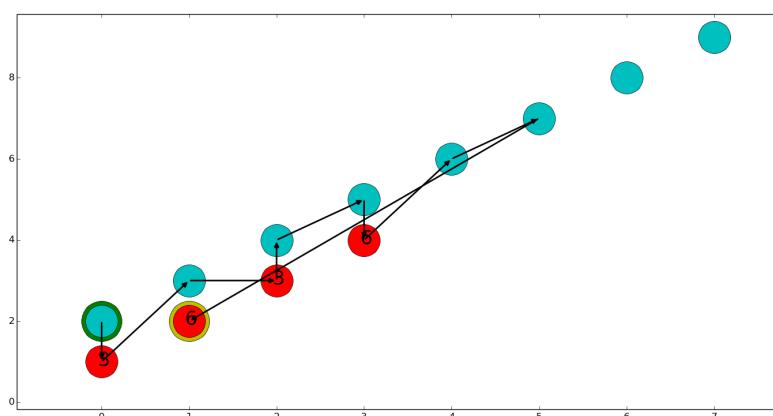
Se pudo observar como la heurística Tabu 2-OPT mejoró considerablemente las soluciones para 2-OPT, mientras que en relación a Tabu 3-OPT y 3-OPT las soluciones obtenidas fueron similares. Teniendo en cuenta además, que para esta última el tiempo insumido fue considerablemente mayor que la búsqueda local. Por lo tanto, para esta familia puntualmente, a la hora de elegir una entre los dos Tabu, será más eficiente en relación tiempo - calidad de solución la Tabu 2-OPT.

Familia 6

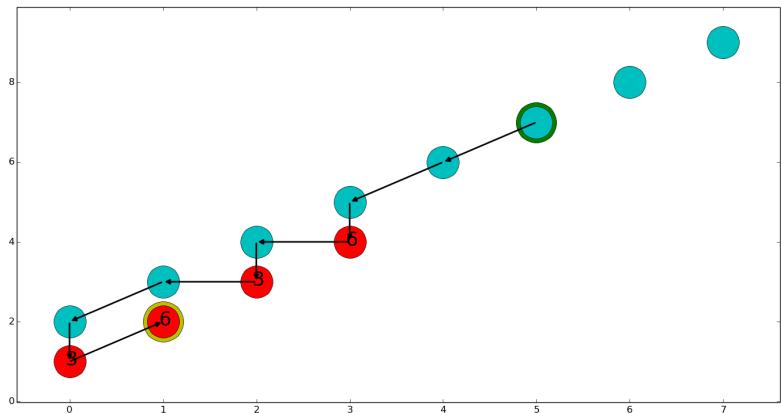
Como enunciábamos anteriormente esta familia tiene como característica principal tener pockeparadas a ambos lados de gimnasios, generando que el resultado de la heurística golosa de una solución muy alejada de la óptima, por esta razón luego de haber realizado nuevas heurísticas de búsquedas locales, puntualmente 2-OPT y 3-OPT, pudimos observar que se vio mejorada la solución.

Procederemos a ver, si con las nuevas heurísticas implementadas Tabu 2-OPT y 3-OPT la solución que obtenemos se acerca o iguala a la óptima.

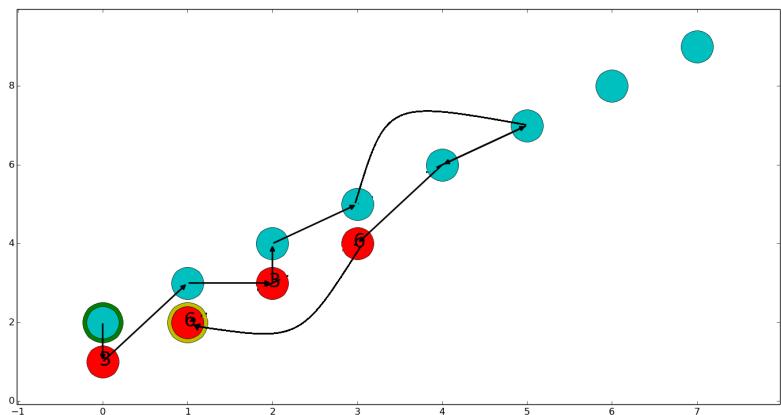
Veamos un ejemplo de como se mejora el camino obtenido por el goloso al utilizar Tabu 2-OPT y Tabu 3-OPT:



Solución Golosa



Solución TABU 2-OPT



Solución TABU 3-OPT

Luego de haber ejemplificado una instancia posible queremos ver como se comporta Tabu 2-OPT con respecto a la heurística de búsqueda local 2-OPT:

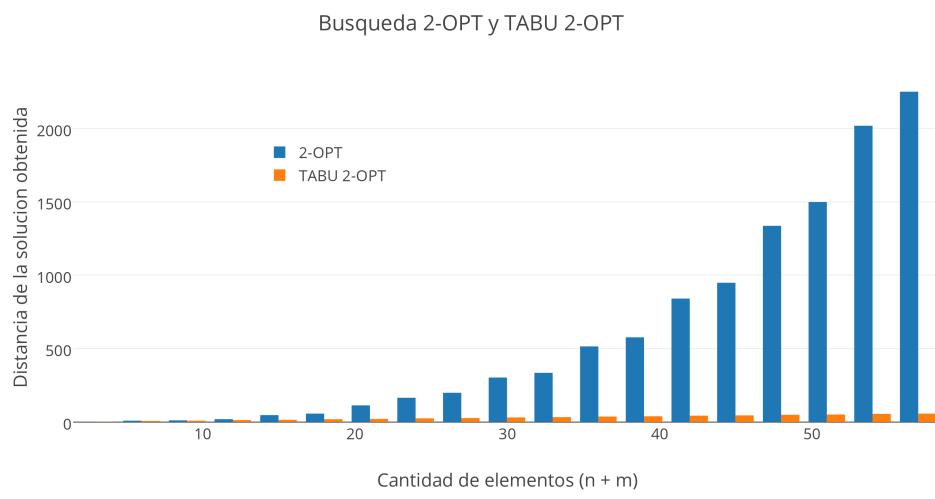


Gráfico 4.7 - 2-OPT vs Tabu 2-OPT sobre Familia 6

Busqueda 2-OPT y TABU 2-OPT

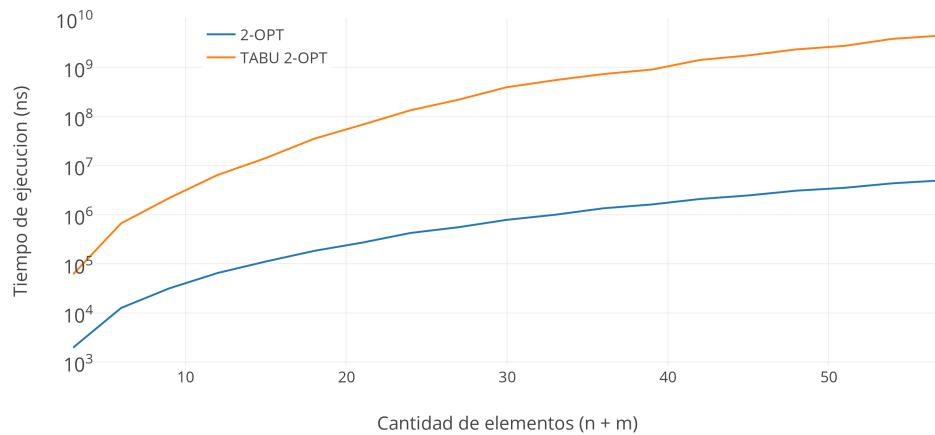


Gráfico 4.8 - 2-OPT vs Tabu 2-OPT sobre Familia 6

Luego, para 3-OPT:

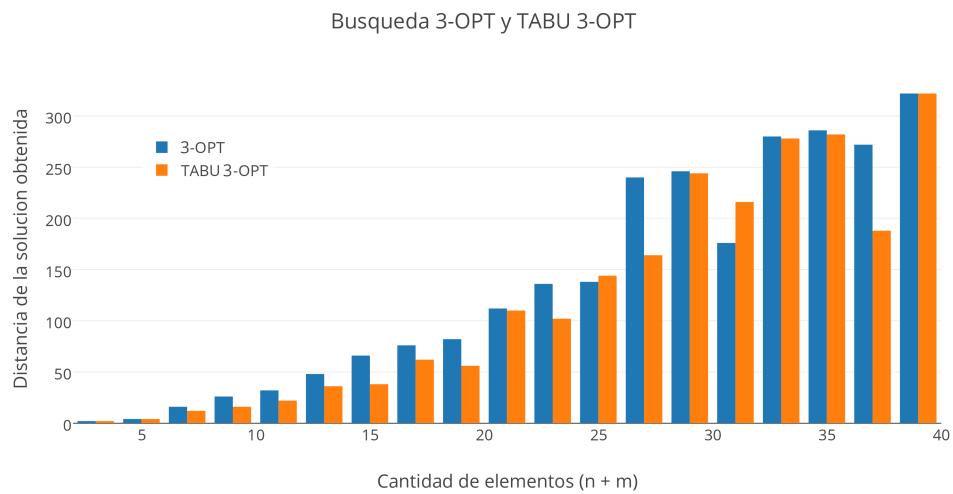


Gráfico 4.9 - 3-OPT vs Tabu 3-OPT sobre Familia 6

Busqueda 3-OPT y TABU 3-OPT

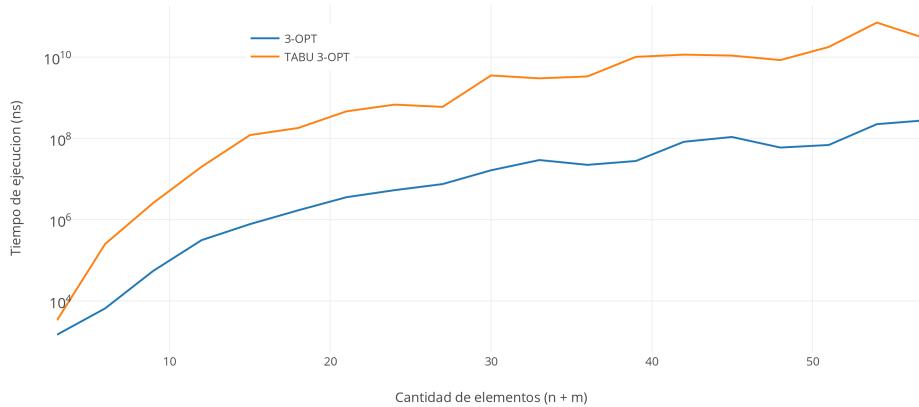


Gráfico 4.10 - 3-OPT vs Tabu 3-OPT sobre Familia 6

Habiendo chequeado dichos experimentos, pudimos observar que utilizando Tabu 2-OPT mejoramos considerablemente la solución obtenida por 2-OPT. A pesar de tener en cuenta que, en la medición de tiempo insumido el tabu tarda más, la calidad de solución obtenida es hasta 10 veces mejor que en la búsqueda, es por esto que, para esta familia puntual es preferible trabajar con Tabu 2-OPT que con la búsqueda en cuestión. Por otro lado, el otro Tabu, en vez de mejorar la solución en muchos casos nos otorga una solución inferior a la que ya teníamos.

Comparando las soluciones de cada versión de tabú search podemos observar lo siguiente:

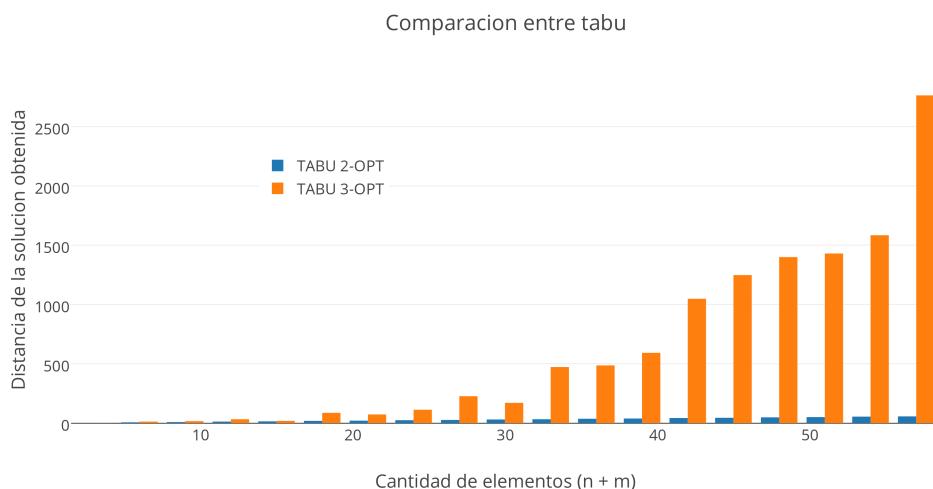


Gráfico 4.11 - Tabu 2-OPT vs Tabu 3-OPT sobre Familia 6

En cuanto a tiempo insumido vemos lo siguiente:

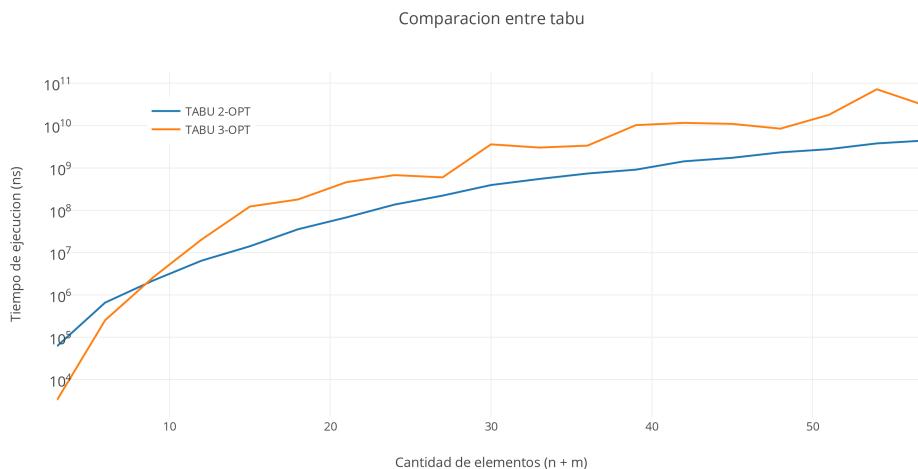


Gráfico 4.12 - Tabu 2-OPT vs Tabu 3-OPT sobre Familia 6

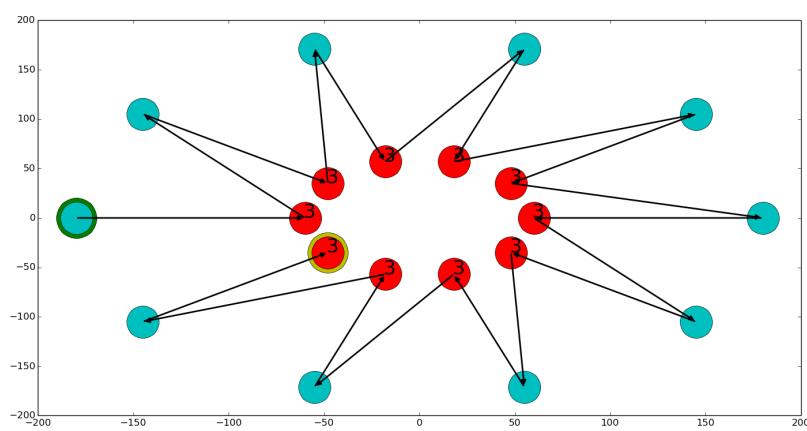
Como mencionamos en las comparaciones de cada tabu con la respectiva búsqueda local, Tabu 2-OPT tendrá un mejor desempeño en consideración al otro para esta familia analizada.

Familia 7

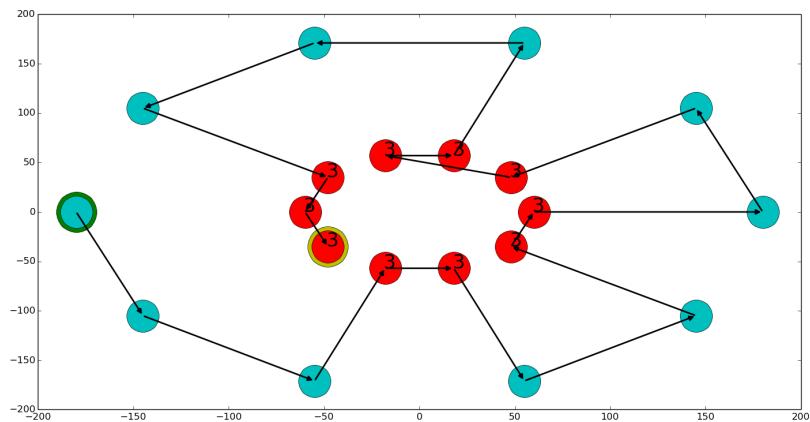
Esta familia de instancias como fue mencionada presenta a los conjuntos de gimnasios y pokeparadas en dos anillos distintos. Como la solución de la heurística golosa irá de una pokeparada a un gimnasio todo el tiempo, la solución obtenida distará considerablemente de la óptima ya que la misma se obtiene recorriendo primero el anillo de pokeparadas y luego el de gimnasios.

Luego de haber obtenido una mejora en la solución con las búsquedas locales, intentaremos mejorar más la solución del mismo utilizando las meta-heurísticas Tabu 2-OPT y Tabu 3-OPT.

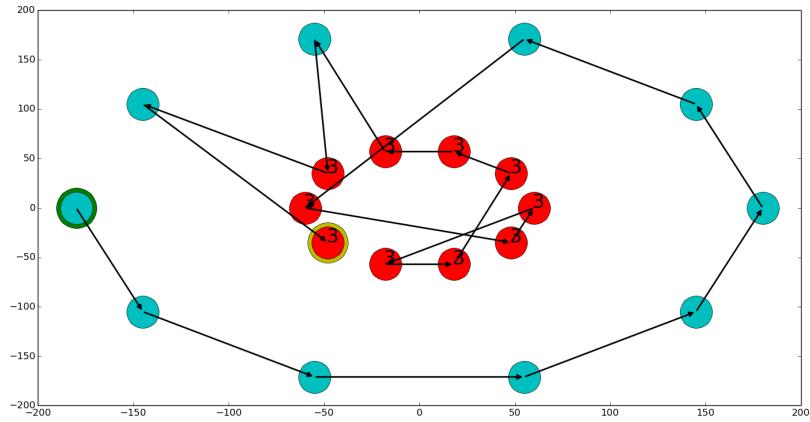
Una exemplificación de cómo es transformado el camino resultante para esta familia es el siguiente:



Solución Golosa



Solución TABU 2-OPT



Solución TABU 3-OPT

Veamos como se comporta Tabu 2-OPT con respecto a la heurística de búsqueda local 2-OPT:

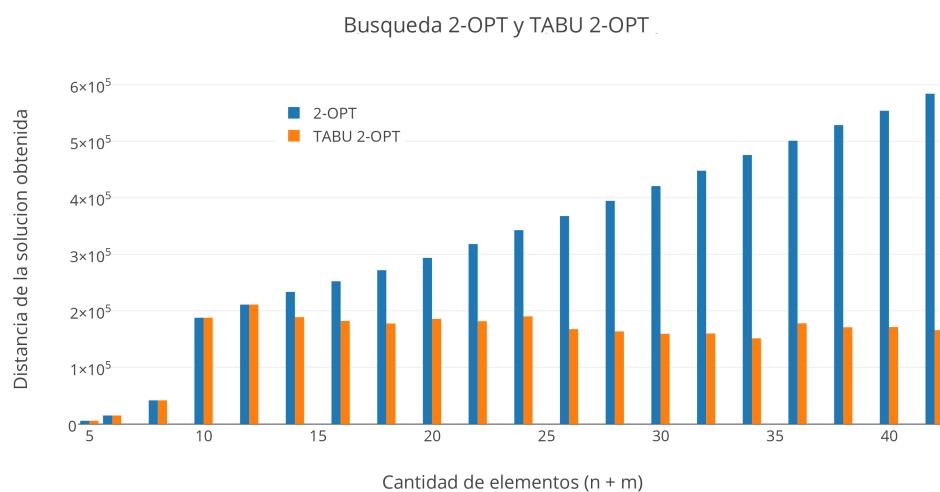


Gráfico 4.7 - 2-OPT vs Tabu 2-OPT sobre Familia 7

Busqueda 2-OPT y TABU 2-OPT

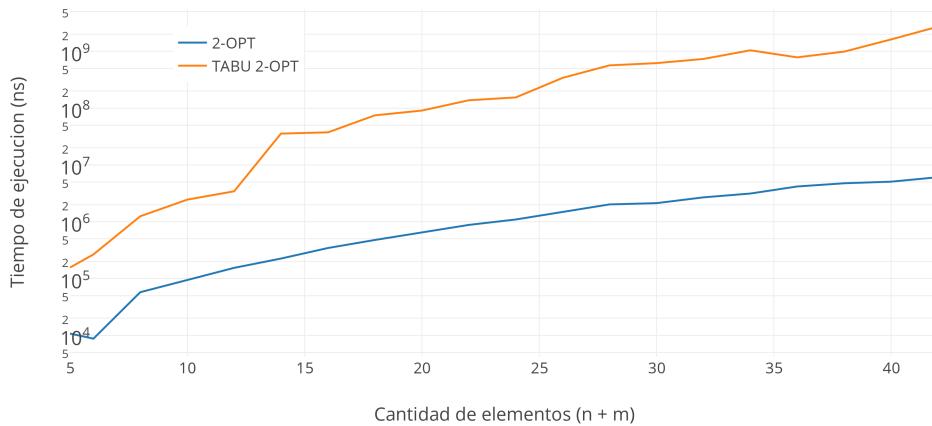


Gráfico 4.8 - 2-OPT vs Tabu 2-OPT sobre Familia 7

Luego, para 3-OPT:

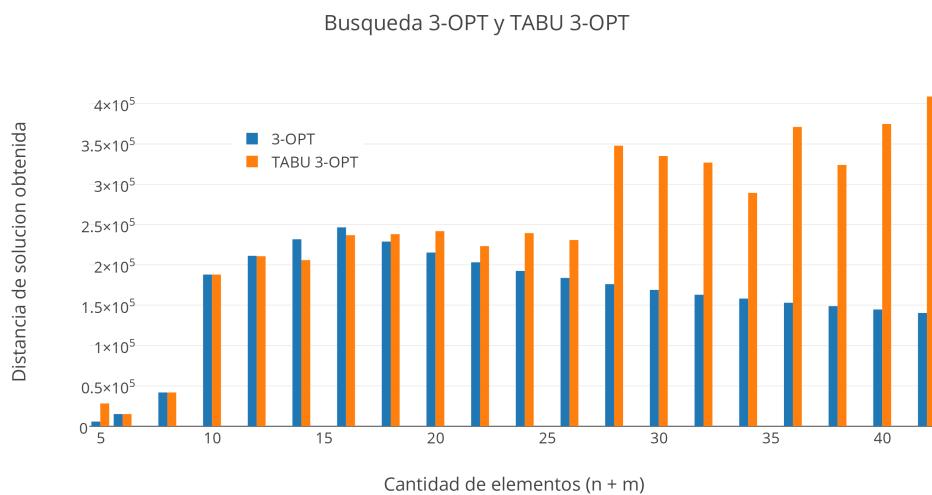


Gráfico 4.9 - 3-OPT vs Tabu 3-OPT sobre Familia 7

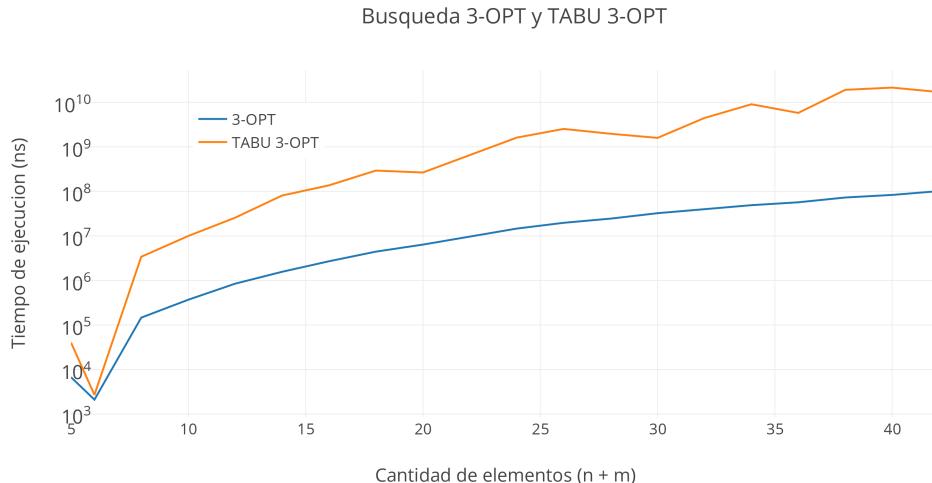


Gráfico 4.10 - 3-OPT vs Tabu 3-OPT sobre Familia 7

Habiendo chequeado dichos experimentos, pudimos observar que para esta familia sucede algo muy similar que en la número 6 donde al utilizar Tabu 2-OPT la solución obtenida es considerablemente mejor que la obtenida por 2-OPT. Y al igual que en la familia anterior, el otro Tabu, presenta un peor desempeño. Ya que las soluciones obtenidas son hasta 3 veces peores que las obtenidas por la búsqueda local 3-OPT.

Comparando las soluciones de cada versión de tabú search podemos observar lo siguiente:

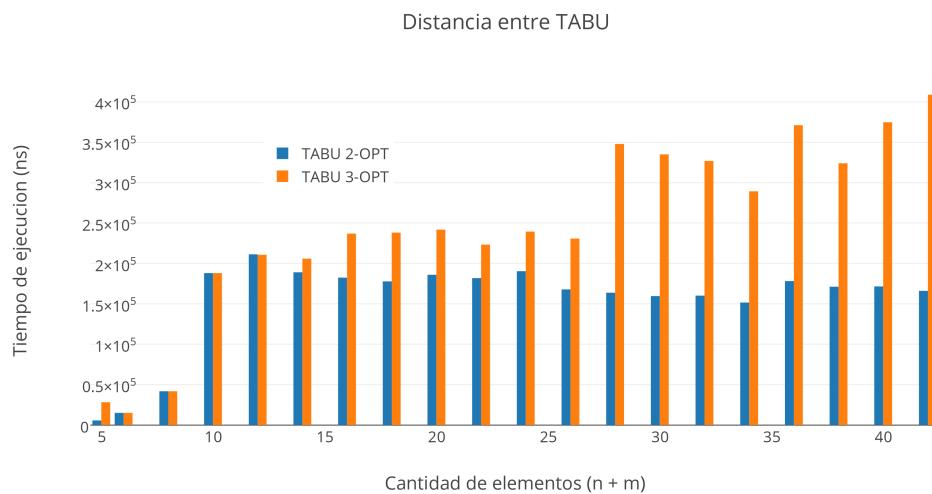


Gráfico 4.11 - Tabu 2-OPT vs Tabu 3-OPT sobre Familia 7

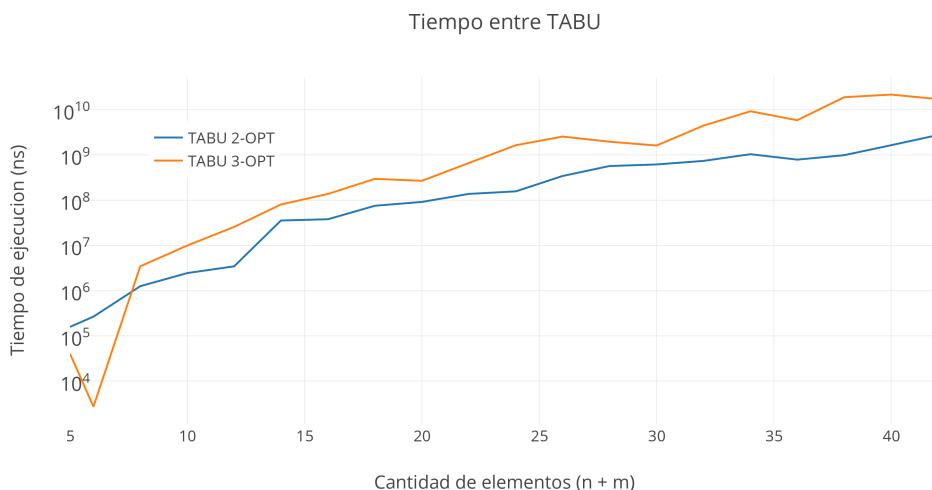


Gráfico 4.12 - Tabu 2-OPT vs Tabu 3-OPT sobre Familia 7

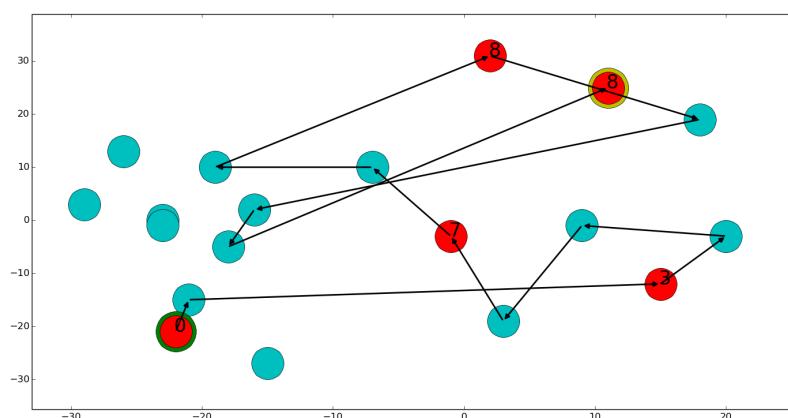
Dando por finalizada la comparaciones entre Tabu, como mencionamos, a la hora de trabajar con alguna de las dos para esta familia, la mejor en relación tiempo-calidad de solución será Tabu 2-OPT.

Familia 8

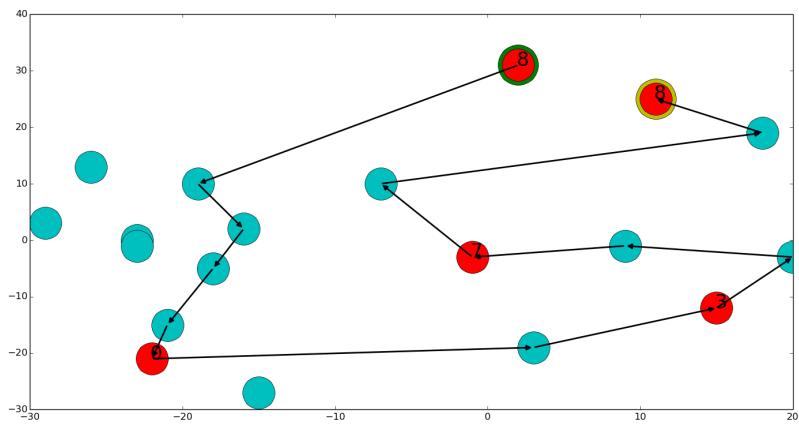
Esta familia, como se denominó, es random, la misma fue implementada de la forma en la que siempre se obtenga solución, o sea, siempre exista la cantidad necesaria de Pokeparadas para vencer a todos los gimnasios y la capacidad de la mochila sea acorde a la cantidad de pociones necesarias para poder vencer al gimnasio mas poderoso.

Por la particularidad de la misma, las soluciones y el tiempo insumido para obtener las mismas suele ser variable. Se intentará por medio de las meta-heurísticas trabajadas mejorar en caso de ser posible la solución obtenida por el goloso.

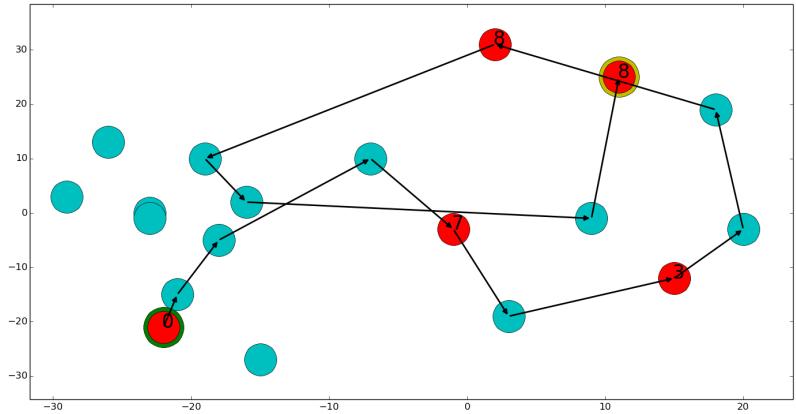
Un ejemplo de como es cambia el camino del goloso por las meta heurísticas es el siguiente:



Solución Golosa



Solución TABU 2-OPT



Solución TABU 3-OPT

Veamos como se comporta Tabu 2-OPT con respecto a la heurística de búsqueda local 2-OPT:

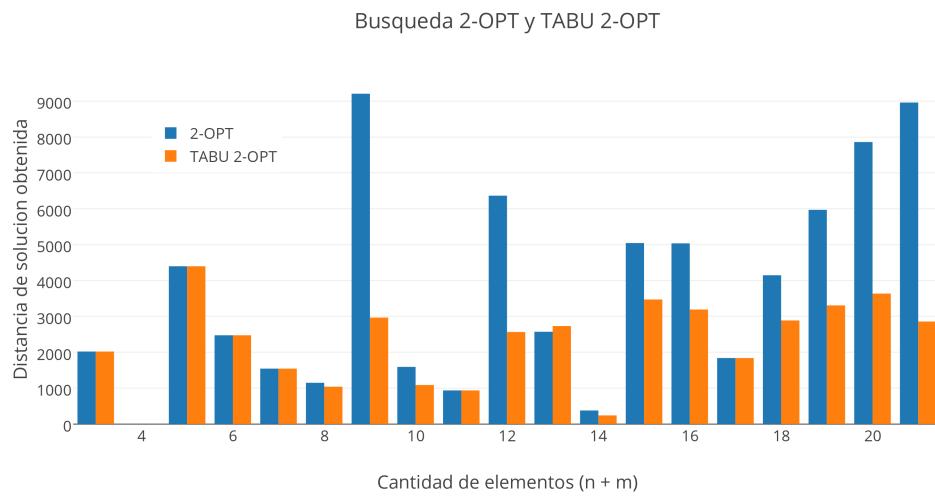


Gráfico 4.7 - 2-OPT vs Tabu 2-OPT sobre Familia 8

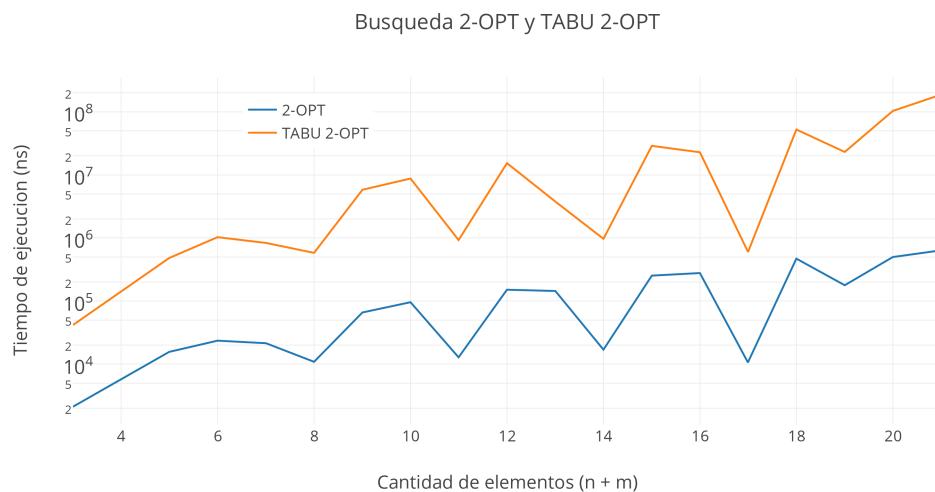


Gráfico 4.8 - 2-OPT vs Tabu 2-OPT sobre Familia 8

Luego, para 3-OPT:

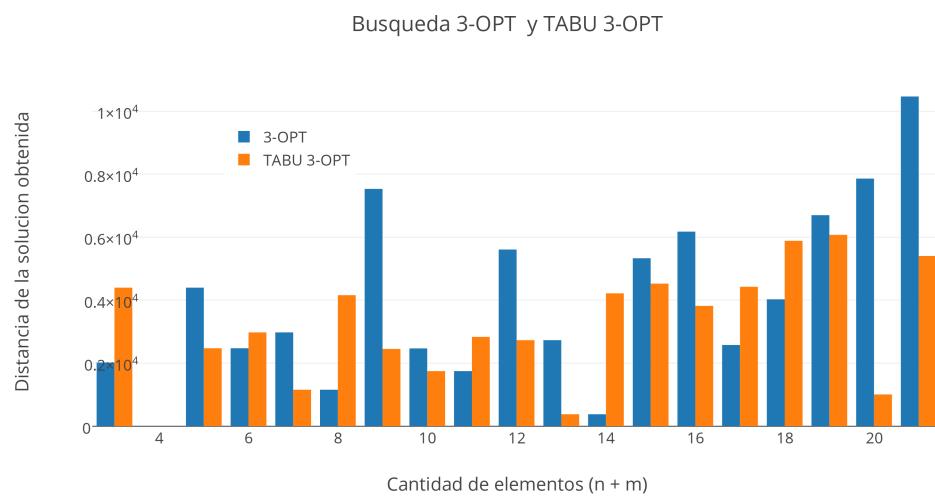


Gráfico 4.9 - 3-OPT vs Tabu 3-OPT sobre Familia 8

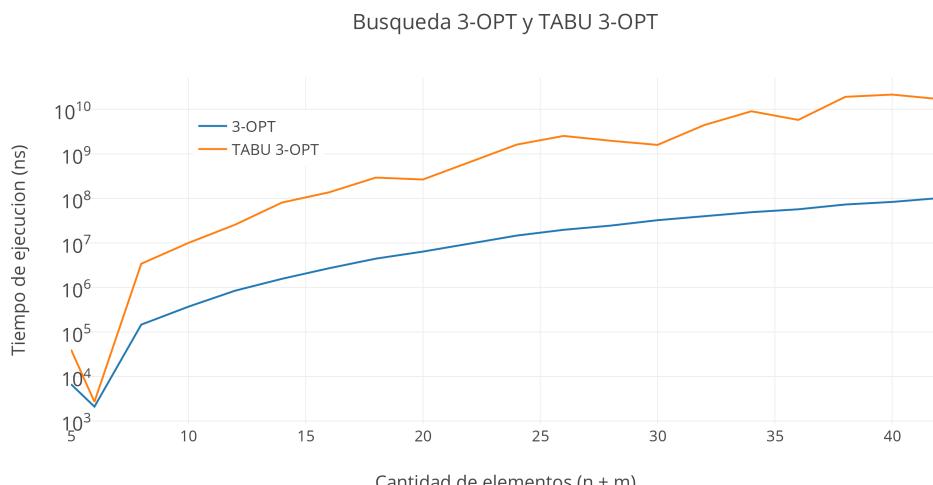


Gráfico 4.10 - 3-OPT vs Tabu 3-OPT sobre Familia 8

Dada la particularidad de este caso, las soluciones obtenidas y el tiempo insumido es variante tanto para las heurísticas como para las meta-heurísticas. A pesar esto tanto Tabu 2-OPT como Tabu 3-OPT obtienen soluciones mejores.

Comparando las soluciones de cada versión de tabú search podemos observar lo siguiente:

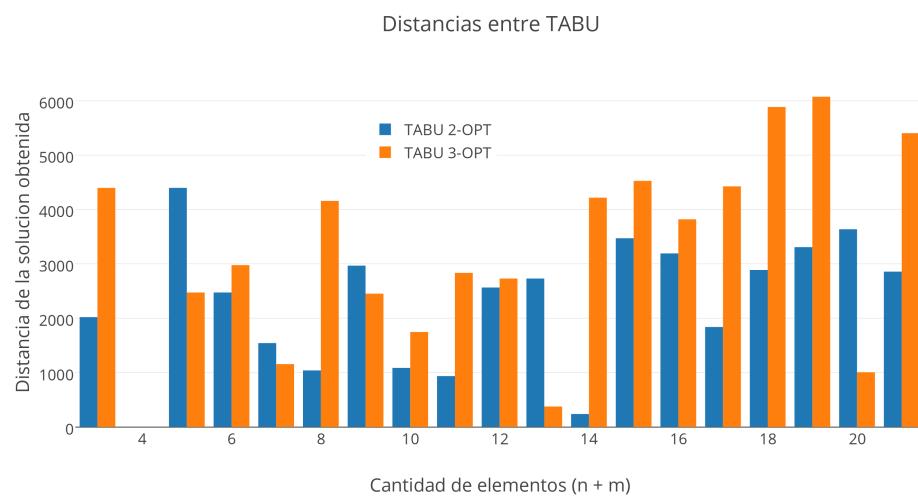


Gráfico 4.11 - Tabu 2-OPT vs Tabu 3-OPT sobre Familia 8

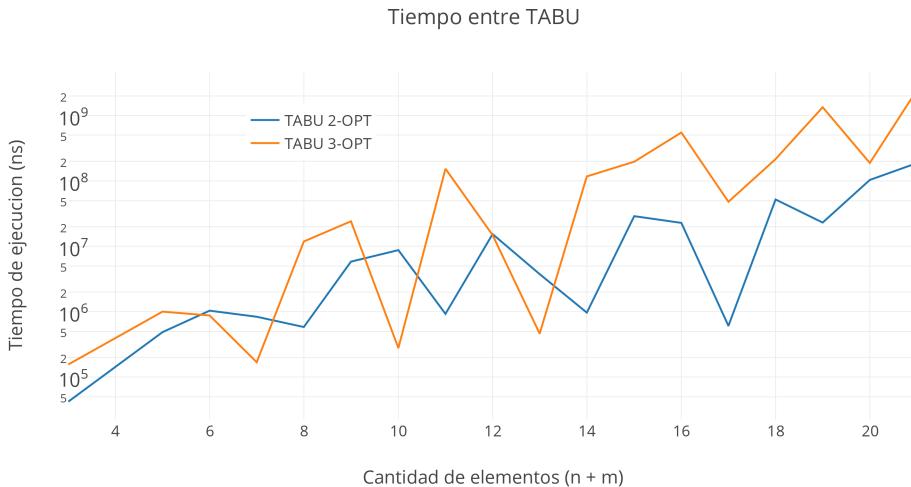


Gráfico 4.12 - Tabu 2-OPT vs Tabu 3-OPT sobre Familia 8

Dando por finalizada esta familia, a pesar de la oscilación en los resultados y mediciones Tabu 2-OPT presento un mejor desempeño que el otro.

Como pudimos ver en este informe, para la mayoría de las familias es notoriamente mejor utilizar Tabu 2-OPT que Tabu 3-OPT ya que este ultimo insume un mayor tiempo en obtener soluciones, y dichas soluciones suelen ser siempre inferiores al primero. Debido a la oscilación de la familia random podra ser elegida la busqueda local 2-OPT para esta misma ya que los valores obtenidos y el tiempo que insuma llegar a estos puede ser mayor para las meta-heuristicas trabajadas.

A la hora de trabajar con estas familias y tener que elegir entre Tabu 3-OPT y la búsqueda respectiva, como vimos será preferible trabajar con la heurística ya que esta obtendrá una mejor solución y en menor tiempo. Por el contrario, para el otro Tabu, obtendrá una solución ampliamente mejor dejando de lado que el tiempo insumido pueda llegar a ser un poco mayor.

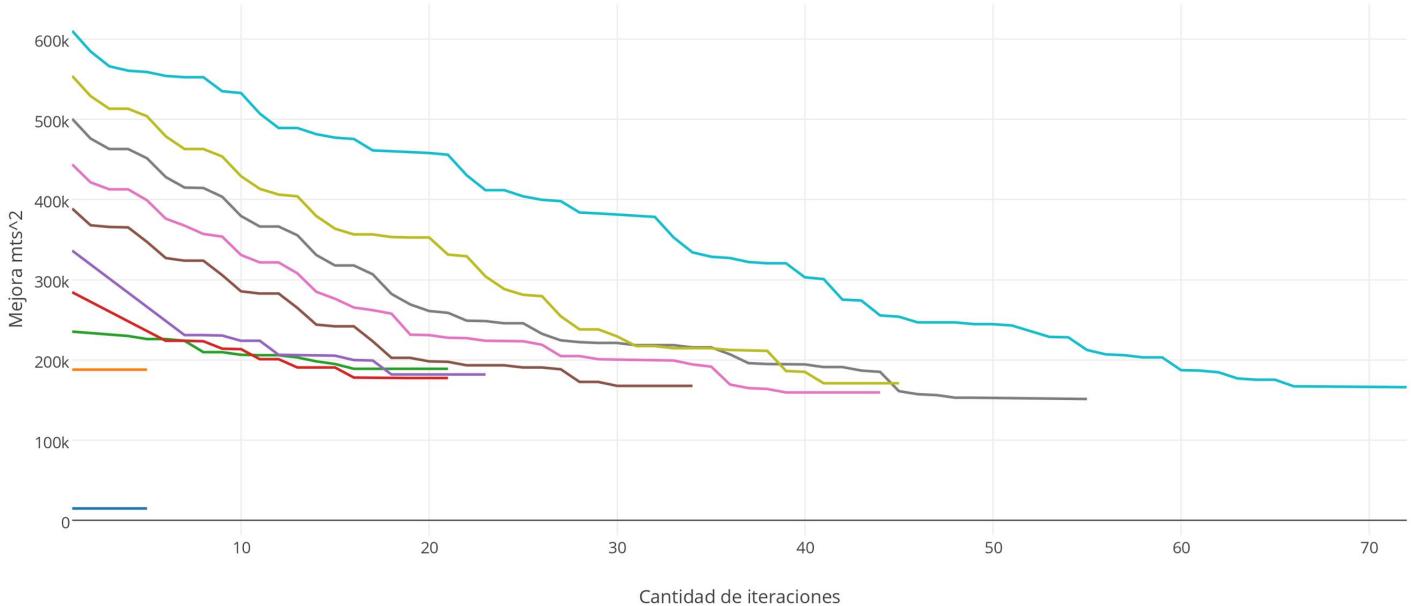
En adición a lo desarrollado, podemos ver que el tiempo de ejecución del algoritmo está ligado plenamente a la cantidad de iteraciones que realiza el algoritmo. A estas iteraciones a su vez, como también determinan a la exactitud de la solución, se las debe acotar de forma tal que se pueda obtener un trade/off beneficioso entre tiempo de computo y solución obtenida. Para este análisis amparamos 2 opciones:

- Corte por cantidad de iteraciones máxima: Esta variante implica determinar un número máximo fijo de iteraciones (por ejemplo 100), determinando un tiempo fijo de ejecución máximo.
- Corte por criterio de calidad determinado: El criterio sería detectar cierta convergencia en la distancia mínima lograda y cortar en ese límite. El criterio se basa en que si no se logra una mejora en la solución por 4 iteraciones, entonces se considera que no se puede mejorar la solución.

La elección seleccionada fue la segunda. Esto se debe a que al analizar distintos tamaños de instancias, se podía ver que en las instancias de menor tamaño, se llegaba a una "meseta" de donde no se podía mejorar más la solución, y las iteraciones restantes, simplemente desperdiciaban tiempo de computo. Por otro lado, para instancias grandes, con muchos cambios posibles, se producía un corte en la mejora de la solución, ya que el algoritmo no llegaba a encontrar esa "meseta" por que cortaba las iteraciones antes de tiempo.

Al cambiar de metodología, se logró hacer que el algoritmo se adapte al tipo de instancia, dándole el tiempo suficiente para llegar a la "meseta" de cada instancia. Cabe destacar que esta elección de corte no determinaría una solución óptima, ya que cada "meseta" simboliza un mínimo en la función de mejoras, pero no se puede asumir que ese mínimo también es un ínfimo.

Mejora vs. cantidad de iteraciones por tamaño de entrada



Instancias de distinto tamaño con cortes acordes a cada caso

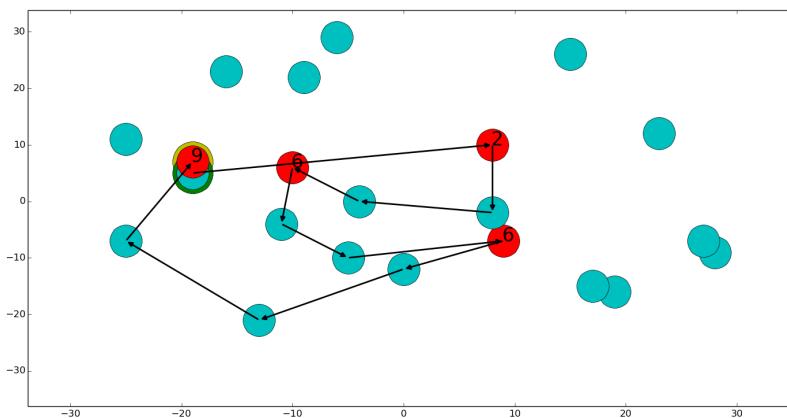
En el gráfico se puede observar el tiempo que cada instancia de un tamaño determinado (cada línea) haya su solución a travez del algoritmo, el decrecimiento implica que la solución mejora, y cuando se detecta que ya no se pudo mejorar por un período de 4 iteraciones, entonces corta y devuelve la solución hallada como la mejor. Claramente no se puede decir que se halla la mejor solución (es decir el que produce el infimo error) pero si un minímo local para cierta vecindad de soluciones.

7 Ejercicio 5

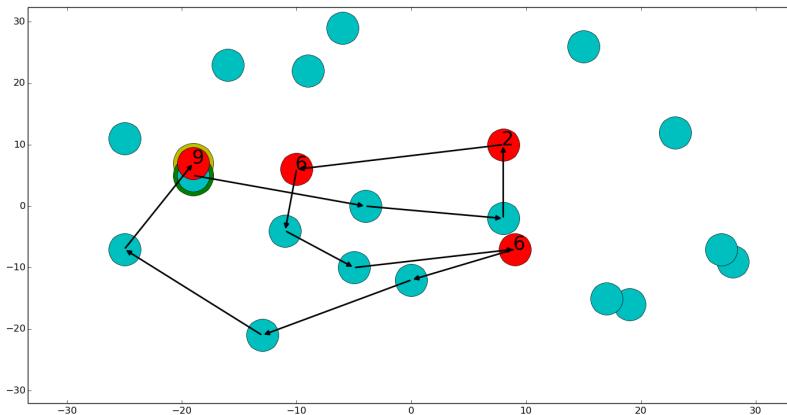
7.1 Descripción de nuevas instancias y experimentación

Para este punto se desarrollaron nuevas instancias random distintas a las anteriores. La generación de nuevos casos de prueba es debido a que los anteriormente utilizados podrían formar una muestra aleatoria que beneficiase más a ciertos algoritmos que a otros. De esta forma podremos evidenciar las diferencias que haya entre las distintas aproximaciones.

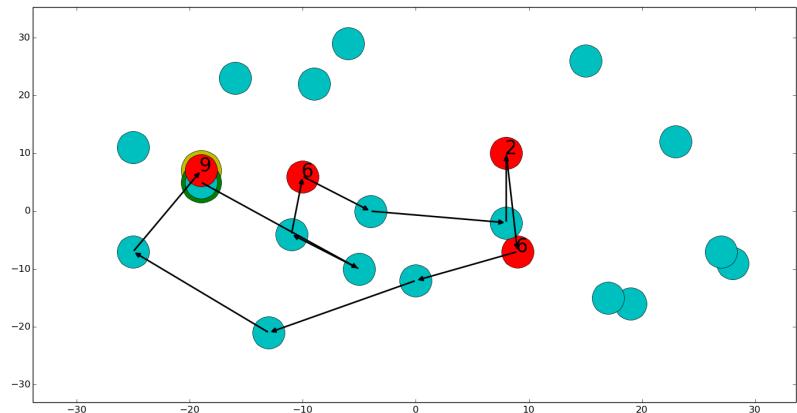
El siguiente es un ejemplo de una instancia dentro de este conjunto nuevo, ejecutada en las 5 variantes:



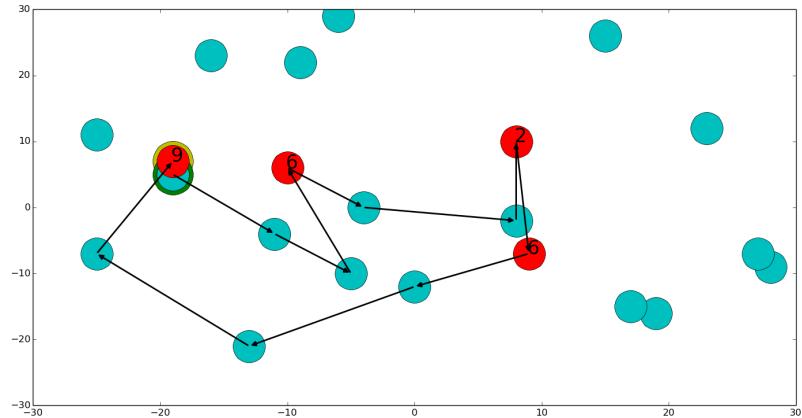
Resultado goloso



Resultado SWAP



Resultado 2-OPT



Resultado 3-OPT

7.1.1 Comparaciones de tiempo entre heurísticas

Para corroborar la performance obtenida para este grupo de instancias se tomaron los tiempos que tarda cada algoritmo de búsqueda local en obtener solución. Dicho conjunto está formado por un total de 50 instancias que van desde 5 elementos hasta 24 en total. Las mediciones de las instancias de menor cantidad de elementos se compararon con la distancia exacta tomada con el backtracking. Para los casos de mayor tamaño, siendo que el tiempo de ejecución exponencial impidió la toma de mediciones, se compararon los resultados solo entre heurísticas.

Comparacion de las heurísticas

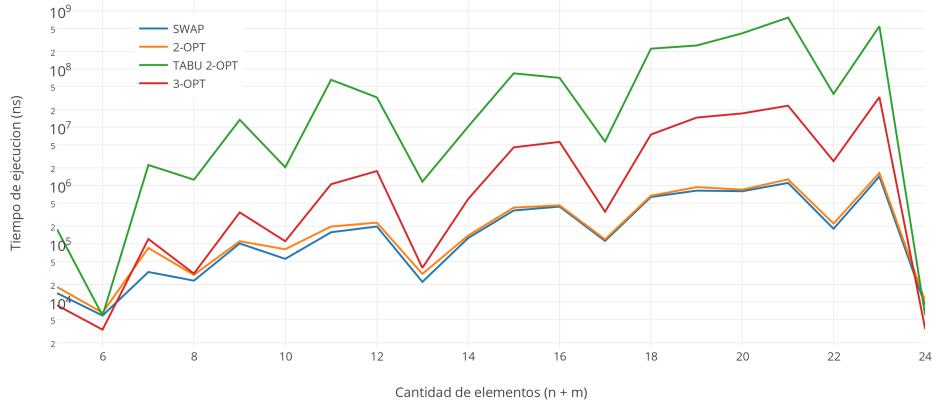


Gráfico 5.1 - Performance de las Heurísticas

Podemos ver que las búsquedas locales que se efectúan en menor tiempo son SWAP y 2-OPT. Las más caras son Tabú y 3-OPT. la comparación en contra de la exacta pierde sentido dada la complejidad de la misma.

Comparacion de soluciones entre algoritmos

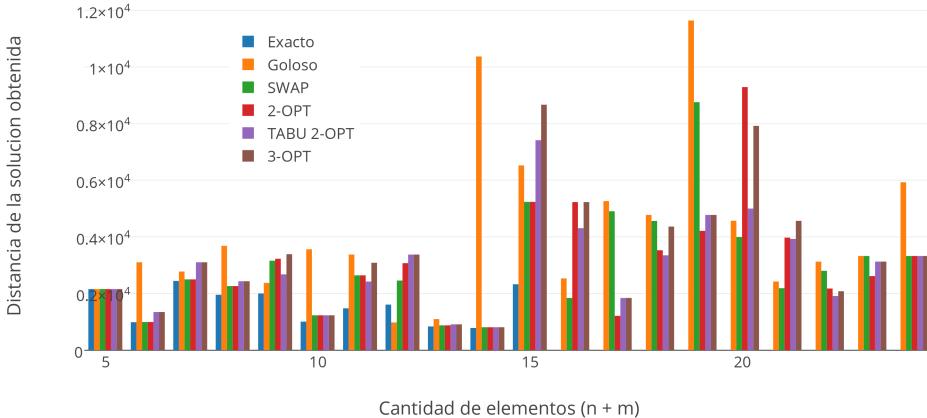


Gráfico 5.2 - Comparación de soluciones de todos los algoritmos

Al tener en cuenta la calidad de la solución, podemos ver que la golosa es mejorada por las búsquedas en la mayoría de los casos. En promedio las mejores las lleva a cabo 2-OPT y SWAP mientras que 3-OPT presenta el peor índice de mejora entre las búsquedas locales. Por su parte, Tabú al utilizar 2-OPT, acompaña los buenos resultados de la misma e incluso, en aquellos en que el desempeño de 2-OPT no fue bueno, logra mejorarlo drásticamente (instancia de 20 elementos). Partiendo el set de pruebas en 2, instancias pequeñas e instancias grandes, podemos observar lo siguiente:

Comparacion de soluciones de algoritmos

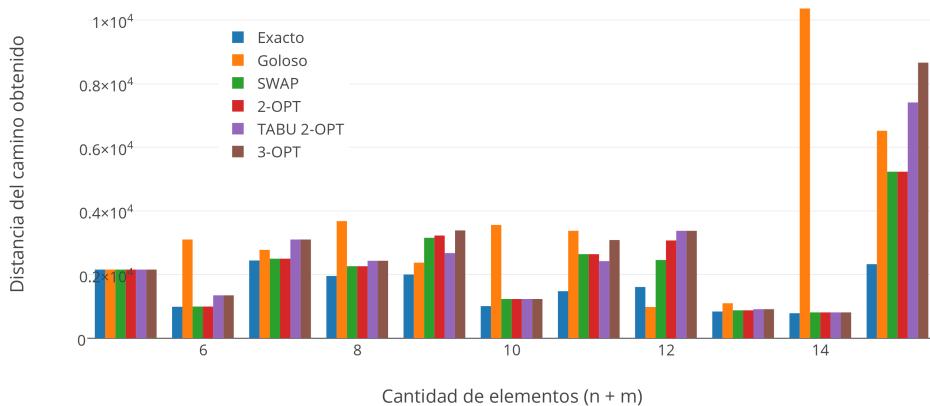


Gráfico 5.2 - Comparación de soluciones de todos los algoritmos

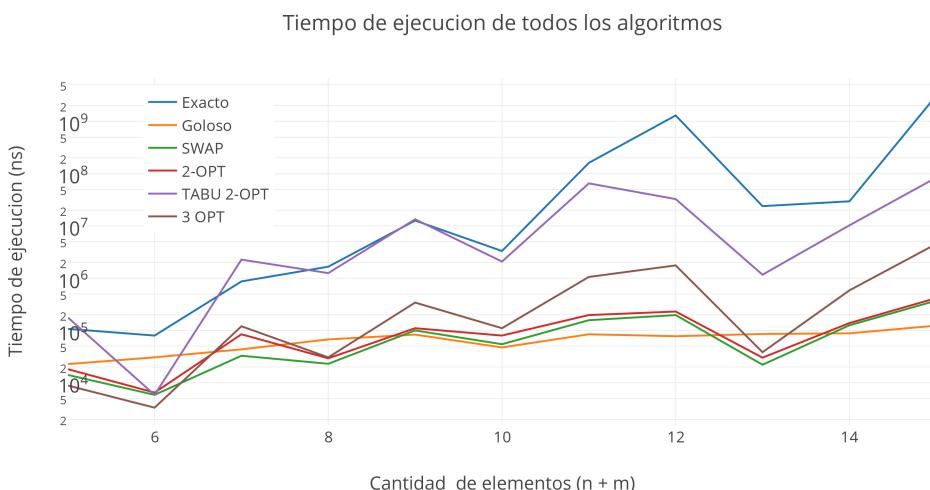


Gráfico 5.2 - Comparación de soluciones de todos los algoritmos

Las instancias de menor tamaño (de 5 a 15 elementos) producen peores soluciones al aplicarse la búsqueda local que la misma heurística golosa (considerando la instancia 14 como un outlier para el goloso). Lo mismo se puede observar entre 2-OPT y Tabú, en donde el segundo empeora todas las soluciones obtenidas por el primero. En cuestiones de tiempo, la utilización de Tabú pierde total sentido, ya que en aproximadamente el mismo tiempo puede obtenerse la solución exacta. Vale destacar que las mediciones de tiempo de las búsquedas incluyen la ejecución del algoritmo goloso, y que en los casos en que el gráfico reporta mejoras de las búsquedas versus el goloso se deben a errores de medición.

Comparacion entre las soluciones de los algoritmos

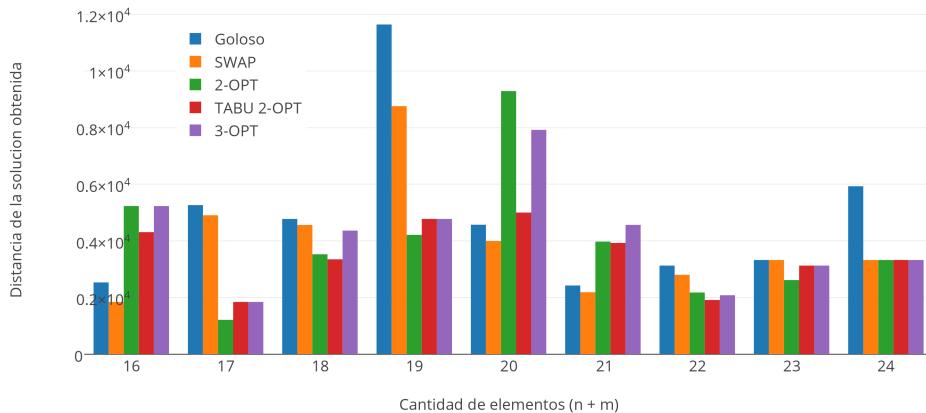


Gráfico 5.3 - Comparación de soluciones entre las heurísticas

Al observar las intancias de mayor tamaño (de 16 a 24 elementos) podemos ver que lo anterior se invierte: a partir de la instancia 16, la solución golosa es mejorada por las búsquedas locales y las soluciones de 2-OPT, a su vez por Tabú.

7.1.2 Comparaciones entre los errores

Reanalizando los valores anteriormente presentados en cuanto a calidad de solución, se analizan instancias ejecutables por el algoritmo exacto. Vale notar que lo buscado es una distribución del error con menor varianza, menor mediana (es decir que el error que realice sea lo más chico posible y lo más predecible dentro de un rango determinado), y en lo posible, la mayor simetría posible: todo esto facilitaría el análisis de la solución y la predictibilidad del error mediante intervalos de confianza o la generación de estadísticos sobre las soluciones obtenidas, dandonos un mayor control y confianza sobre las mismas. la comparación entre errores que producen las heurísticas denota lo siguiente:

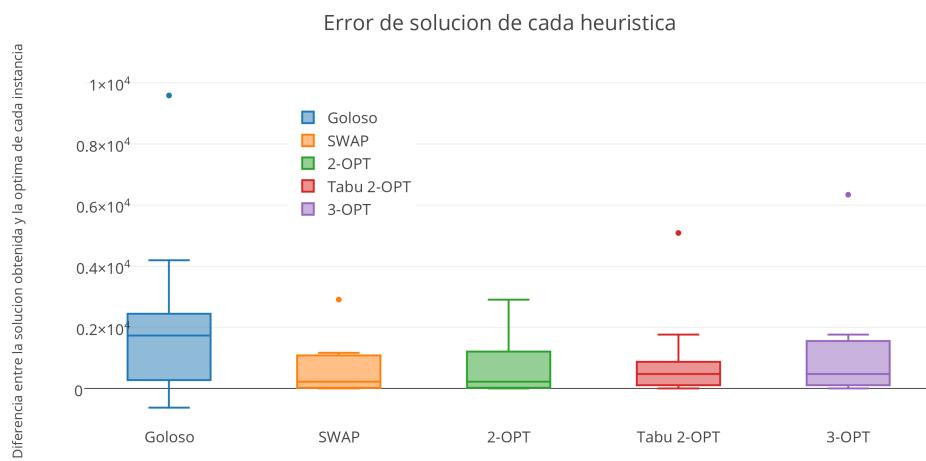


Gráfico 5.4 - Error en la soluciones de las heurísticas

El algoritmo goloso es el que presenta el mayor error medio, con una varianza muy amplia en comparación al resto de los algoritmos, presenta 2 colas en la distribución del error bastante grandes. SWAP por su parte, mejora ampliamente la variabilidad del error del goloso: disminuye determinantemente las colas de la distribución y decrementa el error medio, que en comparación con 2-OPT es

apenas menor. Siendo Tabú implementado con 2-OPT, se puede observar que mejora a distribución del error de éste, de forma que su distribución pasa a ser preferible a la de SWAP, ya que presenta menor varianza y más simetría, mejorando las colas de la de 2-OPT. Finalmente la aproximación mediante 3-OPT produce los peores resultados, con la mayor variabilidad y mediana del error.

7.1.3 Concluciones

La resolución de ciertos problemas a nivel exacto se tornan impracticables a partir de cierto tamaño del problema a analizar, con lo cual es necesario sacrificar exactitud, por obtener al menos una solución. Al intentar solucionar este problema mediante heurísticas, se recala en el hecho de que, dado que no se tiene idea de cuál es la solución buscada, es necesario obtener cierta "confianza" de las soluciones que se puedan llegar a obtener.

Al experimentar con la heurística golosa, pudimos ver que su comportamiento lo podemos mejorar mediante la aplicación de búsquedas locales, las cuales nos permitieron aumentar notoriamente la confianza de la solución obtenida: algunas heurísticas de búsqueda, no obstante, fueron descartadas ya que su costo/beneficio era muy alto en comparación a la del mismo método goloso (Refiriéndonos a 3-OPT).

Tanto la búsqueda mediante SWAP como 2-OPT nos brindaron los mejores resultados. La base de 2-OPT de Tabú nos permitió incluso mejorar lo anteriormente logrado, siendo de esta forma la metodología preferida, en cuanto a calidad de solución. No obstante, de no contar con tanto tiempo de cálculo, la decisión se deberá llevar a cabo entre SWAP y 2-OPT, las cuales se empatan bastante en calidad vs. tiempo.

Experimentaciones a realizar de interés

Dado que 2-OPT y SWAP obtuvieron resultados muy similares y que se implementó Tabú con el primero, es de interés desarrollar el mismo también en base a SWAP y poder realizar comparaciones entre las 2 implementaciones para poder observar su comportamiento y las diferencias que surgen entre ambos, buscando un mejor error.

8 Aclaraciones

8.1 Aclaraciones para correr las implementaciones

Cada ejercicio fue implementado con su propio Makefile para un correcto funcionamiento a la hora de utilizar el mismo.

El ejecutable para el ejercicio 1 sera ej1 el cual recibirá como se solicita entrada por stdin y emitirá su respectiva salida por stdout.

Tanto el ejercicio 2 como el 3 y 4, compilarán de la misma forma y podrán ser ejecutados con ej2, ej3 y ej4 respectivamente.

A su vez, para poder chequear las mediciones de tiempo y nuestros casos de testeo se creó una carpeta nueva por cada ejercicio. Dentro de cada una se cuenta con el respectivo makefile que compila todos los test pudiendo así probar cada uno por separado.