DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

Algoritmos y Estructuras de Datos III
Segundo Cuatrimestre de 2016

Integrante	LU	Correo electrónico
Nicolás Bukovits	546/14	nicobuk@gmail.com
Kevin Frachtenberg	247/14	kevinfra94@gmail.com
Julián Len	467/14	julianlen@gmail.com
Nicolás Len	819/11	nicolaslen@gmail.com

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

Índice

1. Introducción

Este trabajo es el resultado del estudio de una variante del *Traveling Salesman Problem*. El TSP (por sus siglas) es conocido en español como "Problema del viajante", el cual sitúa a un viajero en búsqueda del camino más corto que cumpla ciertas condiciones las cuales, en general, son que debe recorrer distintas ciudades interconectadas sin pasar más de una vez por cada una. En nuestro caso de estudio, tenemos un jugador de *Pokémon GO* en una ciudad en la que hay distintos puntos con 2 posibles características: los **gimnasios** y las **pokeparadas**. Los primeros son lugares que el jugador desea conquistar dado que están, en un comienzo, bajo control del enemigo y para ello debe vencer a los pokemones enemigos utilizando los propios. Un pokemon es una criatura virtual que tiene un poder de ataque y de vida. Para vencer a los enemigos es necesaria una cierta cantidad de posiones, las cuales se obtienen de las pokeparadas. Cada una de ellas le otorga 3 posiones al jugador y la forma en que éste las guarda es en una mochila que tiene una capacidad máxima; si la mochila está llena y el jugador pasa por una pokeparada, las posiones son descartadas. Relacionando ambas ideas, estudiamos nuestra problemática considerando que las restricciones para hallar el camíno mínimo es que dados n la cantidad de gimnasios, m la cantidad de pokeparadas y k el tamaño de la mochila, queremos pasar por todos los gimnasios siempre que tengamos la cantidad de posiones necesarias para capturarlo, en ningún momento se pueden tener más de k posiones y, obviamente, no pasar dos veces por el mismo lugar.

Para obtener la solución a este problema, se plantearon 4 formas distintas, utilizando las técnicas algorítmicas vistas en la materia y modelando los mismos mediante grafos, siendo cada nodo una pokeparada o un gimnasio. Estos presentaron cada uno un desafío distinto, teniendo que aplicar métodos diferentes para resolverlos y cumplir con los requisitos exigidos.

Además de la resolución de los mismos, se procedió a demostrar la correctitud de cada implementación. Esto fue acompañado a su vez de una justificación de la complejidad temporal.

Cada ejercicio contó con su respectiva experimentación para corroborar que la complejidad temporal teórica se cumpliera y en los casos donde el algoritmo podía comportarse mejor, verlo reflejado de alguna manera.

Los experimentos contaron con diversas medidas para asegurar su efectividad de las cuales las siguientes fueron iguales para los tres problemas:

- Sólo se midió el costo temporal de generar la solución, no de lectura y escritura del problema.
- Para la medición del tiempo se utilizó la biblioteca `chronos` con unidad de tiempo en microsegundos.

Los expermientos en los cuales se utilizaron instancias aleatorias fueron desarrollados usando la librería Random de la STL de C++ y bajo las siguientes condiciones (salvo en el ejercicio 2 que además se uso otro caso aleatorio): la cantidad de gimnasios varía entre 1 y 40; las pociones que requieren los gimnasios para vencerlos varían entre 0 y 20; el tamaño de la mochila se configuró como la suma de las pociones que necesita cada gimnasio y la cantidad de pokeparadas es el tamaño de la mochila dividido por tres, de la cual se toma la parte entera y se le suma uno. Las posiciones de las estaciones (tanto x como y) varían entre 0 y 1000.

En todos los casos, el input del programa se dió de la siguiente manera: Una línea con tres valores enteros positivos, n, m y k, separados por espacios; los valores n y m indican la cantidad gimnasios y paradas, respectivamente, mientras que k indica el tamaño de la mochila. A continuación, siguen n líneas, representando los gimnasios. Cada una contiene 3 enteros, x, y y p, donde x e y indican la posición, y p la cantidad de pociones necesarias para vencer el gimnasio. Las siguentes m lineas indican la posición de las pokeparadas, cada una contiene dos enteros x e y. En el caso de Búsqueda Local, se agrega el input para indicar el vecindario que será recorrido dentro de la búsqueda.

2. Ejercicio 1: Solución exacta

2.1. Descripción del problema y solución propuesta

En este primer acercamiento, se pidió realizar un algoritmo exacto que resuelva el problema en cuestión. El mismo debía contener podas y estrategias que permitan mejorar el tiempo de ejecución pero que aseguren devolver la solución correcta (en caso de haber).

Una solución existe cuando se pueden conquistar todos los gimnasios y además es exacta cuando se conquistan todos ellos recorriendo la menor distancia posible.

Dado que nuestro problema es una instancia del TSP mencionado en la introducción, resolvimos encontrar la solución exacta utilizando la técnica de *backtracking*. Esto es probar todas las soluciones posibles descartando la mayor cantidad que no sean correctas a la vez mediante el uso de *podas*, para así obtener lo más rápido posible la solución correcta. Cada solución se construye de manera tal que en cada iteración se agrega parte de una posible solución. En este caso, lo que se hace es recorrer el grafo desde cada nodo probando todos los caminos posibles, es decir, partiendo de cada gimnasio o pokeparada intentar conquistar todos los gimnasios de todas las formas posibles sin pasar dos veces por el mismo lugar. Las podas utilizadas fueron las siguientes:

- Verificar al principio que dado pociones(g) el número de pociones para un gimnasio g, $\sum_{g \in Gimnasios} pociones(g) \leq m * 3$ con m cantidad de pokeparadas no recorridas
- Verificar al principio que $\forall g \in Gimnasios$ ($k \geq$ pociones(g)) con k el tamaño de la mochila
- Chequear que en todo momento el recorrido que se está llevando a cabo tiene distancia menor o igual a la solución actual si es que ya se encontró una
- No pasar por pokeparadas si la mochila está llena

En cada iteración, se visita un nuevo punto (gimnasio o pokeparada), agregando la distancia correspondiente a la posible solución.

2.2. Detalles implementativos

Para modelar las pokeparadas y gimnasios, implementamos la clase *Estacion*, la cual consiste en: un booleano para decidir si es gimasio o pokeparada, un entero que indica la cantidad de pociones y un id único para cada estación (el cual se da segun el número de linea de la entrada). Para determinar el valor de pociones, en el caso de las pokeparadas es simplemente 3 ya que es la cantidad que entrega cada una, mientras que en el caso de los gimnasios es 0 menos el número de pociones necesarias para ganarlo, que se pasa en el input. Esto es justamente porque los gimnasios consumen pociones de la mochila.

Para trabajar las distancias entre cada punto, construímos una matriz formada con vectores de vectores, en los que cada índice es el ID del primer nodo con el ID del segundo y para cada posicion (i, v) se encuentra la distancia entre los nodos i y v, la cual se calcula mediante pitágoras.

La función principal de Backtracking recibe como parámetros un vector de Estaciones, que consiste en todas las estaciones aún no visitadas (*estaciones*), la matriz de distancias (*distancias*), el tamaño de la mochila (k), un vector de Estaciones visitadas (*visitados*), la cantidad de pociones actuales (*potasActuales*) y una tupla de *double*, *int* y vector de *int* que representa la solución actual (*solucion*). La misma realiza la lógica descripta en el siguiente pseudocódigo:

```
BT_capturar_gimnasios(lista<Estacion> estaciones, lista< lista<double> > distancias,
entero k, lista<Estacion> visitados, entero potasActuales,
tupla<double,entero,lista<IDs> > soluciones):
  si es solucion(estaciones):
    distancia = distancia_acumulada(visitados,distancias)
    lista<IDs> camino                                    //(un ID es un numero entero)
    para i = 0 hasta tamaño(visitados):
      agregar id(visitados[i]) a camino
    fin para
    soluciones = tupla(distancia, tamaño(visitados), camino)
  si no:
    para i = 0 hasta tamaño(estaciones):
      nueva_distancia = 0
      si tamaño(visitados) > 0:
        id_ultimo_visitado = id(ultimo(visitados))
        proxima_distancia = distancias[id_ultimo_visitado][id(estaciones[i])]
        nueva_distancia = proxima_distancia + distancia_acumulada(visitados,distancias)
      fin si
      si (soluciones[0] >= 0 y nueva_distancia < soluciones[0] ó soluciones[0] < 0):
        si (puede_ganar_gimnasio(estaciones[i],potasActuales) ó
            puede_recibir_potas(estaciones[i],potasActuales,k)):
          agregar estaciones[i] a visitados
          si (esGimnasio(estaciones[i]) ó potasActuales + 3 <= k)
            potasActuales = potasActuales + estaciones[i].potas
          si no
            potasActuales = k
          fin si
          borrar elemento con índice i de estaciones
          BT_capturar_gimnasios(estaciones,distancias,k,visitados,potasActuales,soluciones)
          ultimaEstacion = ultimo(visitados)
          eliminar el último de visitados
          insertar ultimaEstacion en posición i de estaciones
        fin si
      fin si
    fin para
  fin si
  devolver soluciones
fin BT_capturar_gimnasios
```

En cada iteración de la función, se chequea si el estado actual de los parametros es una solución al problema. Para esto, se revisa todo el vector de estaciones en búsqueda de algúna Estación que sea gimnasio. Si no hay, entonces efectivamente es una solución y se construye la tupla con lo que devuelve el algoritmo para luego imprimir el output correspondiente. Caso contrario, se recorre todo el vector de estaciones para chequear todos los posibles lugares a los que se puede ir desde la ubicación actual e ir a todos ellos.

Para decidir si podemos ir o no a una estación chequeamos la cantidad de pociones actuales que se tienen junto con el tamaño de la mochila y las pociones de la estación a la que se desea ir. Si es un gimnasio, analizamos si $pociones + pociones(gimnasio) > 0$. Dado que el gimnasio tiene la cantidad de pociones en negativo (como mencionamos anteriormente), si esta suma deciende de 0 quiere decir que las pociones que tenemos en la mochila no son suficientes para conquistar ese gimnasio, por lo que no tendría sentido ir ya que esto solamente aumentaría la distancia recorrida (y esto último ocurre porque como todas las estaciones están conectadas, $\forall v, w, r \in Estaciones : distancia(v, w) \leq distancia(v, r) + distancia(r, w)$). Si es una pokeparada, observamos si $pociones = k$ con k tamaño de la mochila. La razón es que cada pokeparada otorga 3 pociones pero si la mochila se llena entonces se descartan las excedentes. En particular, si la mochila esta totalmente llena, se descartan todas las pociones de esa pokeparada; y teniendo en cuenta lo dicho recientemente, si hay solución debería existir otra estación tal que se pueda

visitar sin agregar innecesariamente distancia recorrida a la solución actual.

Antes de entrar en la función descripta, se realizan dos chequeos (las podas). Primero, que no exista un gimnasio cuya cantidad de pociones sea mayor al tamaño de la mochila, ya que si esto ocurriera, incluso pasando por todas las pokeparadas posibles nunca alcanzaríamos la cantidad necesaria para conquistar ese gimnasio. Un ejemplo puede ser 2 pokeparadas, 1 gimnasio que necesita 10 pociones y tamaño de mochila 9. En segundo lugar, que la suma total de pociones necesarias para conquistar a cada gimnasio sea menor o igual a la cantidad de $pokeparadas*3$. Esto se hace para descartar los casos en los que incluso teniendo el tamaño de mochila suficiente para guardar todas las pociones de todas las pokeparadas, las mismas no alcancen para conquistar todos los gimnasios. Un caso ejemplo sería 1 pokeparada, 2 gimnasios que necesitan 3 pociones y tamaño de mochila 3.

2.3. Complejidad teórica

La primeras dos podas, que son los chequeos que se realizan antes de llamar a la función principal, consisten en recorrer el vector de Estaciones sumando y restando valores de pociones, por lo que es $O(n+m)$ temporal y $O(1)$ espacial ya que solo crean 2 variables enteras.

El algoritmo principal lo primero que hace es chequear si el estado actual de los parámentros, como se dijo anteriormente, corresponden a una solución del problema. Este análisis toma $O(n+m)$ temporal y $O(1)$ espacial ya que solo es recorrer el vector de Estaciones para chequear si existen gimnasios no visitados. Si es solución, se obtiene la distancia acumulada, que toma $O(n+m)$ ya que es solamente recorrer el vector de visitados (y por lo tanto $O(1)$ espacial) y luego se recorre nuevamente el mismo vector para obtener el camino que se realizó, costando $O(n+m)$ temporal y $O(1)$ espacial. Si no es solución se recorren todas las estaciones del vector estaciones (que tomará $O(n+m)$) realizando los siguientes pasos para cada una: Si la cantidad de Estaciones visitadas es mayor a 0 (cosa que no ocurrirá solamente en la primera iteración) se obtiene la distancia acumulada más la proxima distancia a sumar, que como dijimos en el párrafo anterior cuesta $O(n+m)$. Luego, si esa distancia es menor que la distancia de la solución encontrada hasta ahora (si la hubiera) y puede ir a conquistar el gimnasio (si el la próxima posición lo es) o la cantidad de pociones es menor al tamaño de la mochila (y la próxima posición es una pokeparada), se agrega al vector de visitados la próxima estación ($O(1)$ amortizado), se actualiza el valor de pociones ($O(1)$), se elimina la estación del vector estaciones ($O(n+m)$), se llama a la función recursiva (luego explicado) y por último se vuelve al estado original los vectores de visitados y estaciones ($O(n+m)$ temporal y $O(1)$ espacial cada uno). El llamado a la función recursiva hace que para cada Estación del vector de estaciones se recorran todas las demás en búsqueda de una solución, por lo que una iteración tomaría $O(n+m)$ temporal y como todos los vectores se pasan siempre por referencia, solo tomaría $O(1)$ espacial (sin contar el overhead en el *stack* de llamadas recursivas). Luego, como en cada iteración se realiza todo lo mencionado y en total son $(n+m)$ iteraciones, en total costará

$$O(\prod_{i=0}^{n+m} i) * O(n+m)$$

$$\in O((n+m)! * (n+m))$$

Por lo que el algoritmo tiene complejidad temporal $O((n+m)! * (n+m))$ y espacial $O((n+m)^2)$ por la matriz de distancia que se construye antes de llamar a la función principal.

2.4. Experimentación

Los experimentos que realizamos para observar los tiempos de ejecución del algoritmo en función del tamaño de entrada y de las podas implementadas consistieron en el análisis de casos que contemplan:

- El comportamiento de las podas
- Configuraciones de valores que no están en la cota de complejidad
- Configuraciones de valores que sí están en la cota de complejidad, dejando fijos los demás

Para todos los casos, el algoritmo se ejecutó 30 veces con cada instancia para intentar tener la mayor presición posible en cuanto a tiempo de ejecución.

2.4.1. Experimento 1

En esta instancia generamos entradas que aumentaban linealmente de 1 a 10 la cantidad de pokeparadas y gimnasios, variando el tamaño de la mochila y marcando a cada gimnasio con la cantidad de pociones necesarias igual al numero de linea en el input (para el gimnasio k, se piden k pociones). Además, las ubicaciones de cada estación se setearon cada una a distancia 1 de la anterior y de forma que para la estación i la ubicación es (i,i).

Algunos ejemplos para el input de este experimento son los siguientes:

```
8 8 36          9 9 45          10 10 55
1 1 1           1 1 1           1 1 1
2 2 2           2 2 2           2 2 2
3 3 3           3 3 3           3 3 3
4 4 4           4 4 4           4 4 4
5 5 5           5 5 5           5 5 5
6 6 6           6 6 6           6 6 6
7 7 7           7 7 7           7 7 7
8 8 8           8 8 8           8 8 8
9 9             9 9 9           9 9 9
10 10           10 10           10 10 10
11 11           11 11           11 11
12 12           12 12           12 12
13 13           13 13           13 13
14 14           14 14           14 14
15 15           15 15           15 15
16 16           16 16           16 16
                17 17           17 17
                18 18           18 18
                                19 19
                                20 20
```

Las primeras 10 entradas se seteó para que el tamaño de la mochila sea $n * m$, es decir que para cada entrada, si la cantidad de gimnasios es n y la de pokeparadas m, entonces el tamaño de la mochila es $n * m$ (eg, 3 gimnasios y 3 pokeparadas, la mochila tiene tamaño 9). Para las siguientes 10 entradas el tamaño de la mochila es $\sum_{g \in gimnasios} pociones(g)$. En las últimas 10, el tamaño se seteó como $n + m$. Para este experimento, esperabamos que para las primeras 20 instancias el algoritmo tome más tiempo que para las últimas 10, ya que si el tamaño de la mochila $k \geq 3 * m$ entonces se pueden guardar todas las pociones de todas las pokeparadas antes de ir a algun gimnasio, mientras que en caso contrario, habrá muchos caminos en los que primero se pasen por pokeparadas sin ir a gimnasios que no se podrían realizar porque siguiendo la poda de que si la mochila está llena no vamos a una pokeparada, la misma se llenaría mucho más rápido y esto impediría que sigamos por esos caminos. Aprovechamos este mismo exprimento para correrlo sin y con la poda de chequeo que el total de pociones de todos los gimnasios debe ser mayor o igual a $3 * m$ para mostrar la diferencia de comportamiento. Los resultados de tiempo en funcion de la cantidad de estaciones según el tamaño de la mochila para el algoritmo sin la poda puede verse en el siguiente gráfico, el cual tiene además una función que sigue la cota de complejidad que es solo para dar una idea de cómo es la misma.

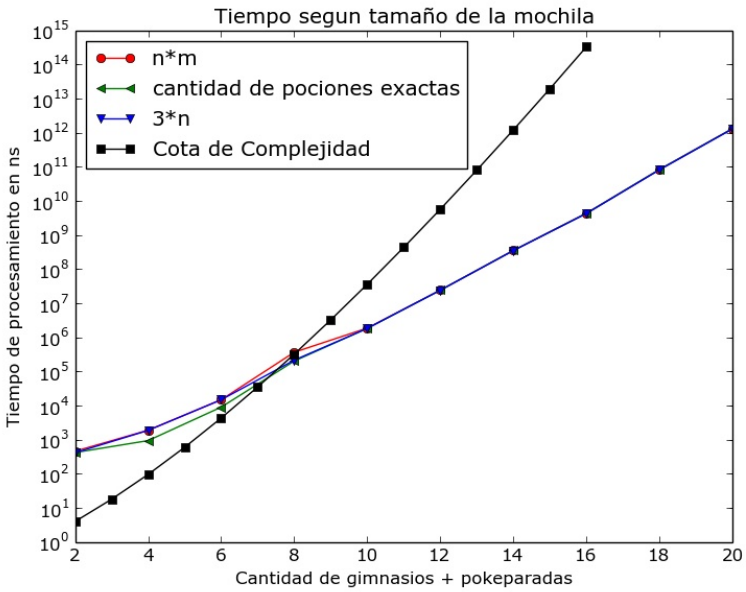


Figura 1

Mientras que los resultados con la poda son los siguientes:

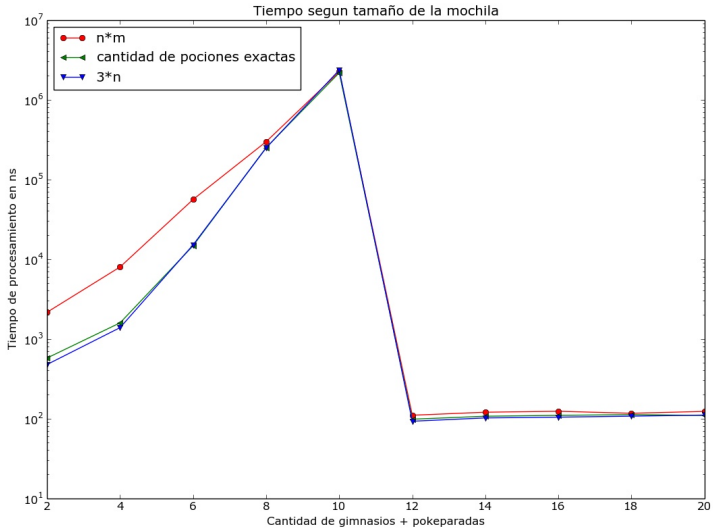


Figura 2

En cuanto al primer gráfico, nos llevamos la sopresa de que las podas que cortan las posibles soluciones que sean peores que la solución actual y que aseguran no ir a una pokeparada si la mochila está llena no fueron muy útiles, pues notamos que los resultados en tiempo son muy similares en cada caso. Tanto, que en el gráfico las líneas de cada instancia están solapadas, pudiendo notarlo en los puntos marcados, que están los marcadores correspondientes a cada línea uno encima del otro.

Respecto al segundo gráfico, notamos que a partir de las 12 estaciones (6 pokeparadas y 6 gimnasios) las podas funcionaron adecuadamente, ya que para esas instancias la cantidad de gimnasios no alcanzaba para cubrir todos los gimnasios (había entre 21 pociones necesarias con 6 pokeparadas hasta 110 pociones necesarias con 10 pokeparadas). Tambien observamos que los tiempos para cada tamaño de mochila no

variaba mucho de los otros, por lo que decidimos hacer otro experimento en el que se observe el tiempo de ejecución cuando lo que varía es la cantidad de pociones necesarias para cada gimnasio.

De ambos resultados obtuvimos la información necesaria para notar que para las instancias que tienen solución, el tiempo de ejecución respecto del tamaño de la entrada crece con una forma similar a la cota de complejidad. También concluímos que las podas implementadas sirven son de mucha utilidad en instancias no necesariamente grandes (con 12 estaciones recortó mucho el tiempo de ejecución).

2.4.2. Experimento 2

Para este experimento, como mencionamos antes, construimos 16 instancias en las que aumentamos linealmente entre 1 y 8 la cantidad de pokeparadas y gimnasios (ambas al mismo tiempo), fijamos en forma relativa a la cantidad de pokeparadas el tamaño de la mochila y colocamos a cada gimnasio un valor entre 1 y 3 pociones para las primeras 8 instancias y fijamos en 3 cada una en las 8 siguientes. Para este experimento esperabamos que las segundas 8 instancias tomen más tiempo que las primeras, ya que en las primeras 8 hay soluciones en las que no se pasan por todas las pokeparadas, teniendo más chances de encontrar una solución mejor y que se utilice la poda que saca los casos en los que se está contruyendo una solución peor a la mejor encontrada hasta el momento. Esto ocurre porque en dos instancias con misma cantidad de pokeparadas y gimnasios, en el segundo caso es necesario pasar por todas las pokeparadas, ya que en cada gimnasio se gastan las 3 pociones que se recolectaron en alguna pokeparada, mientras que en el primer caso hay algunos gimnasios que permiten mantener algunas pociones en lugar de gastarlas todas, permitiendo ahorrar pociones en algún gimnasio para usarlas en otro.

Los resultados obtenidos de este experimento se plasmaron en los siguientes gráficos, los cuales tienen una línea que representa (a modo de referencia) la cota de complejidad y muestran de manera aproximada el funcionamiento para entre 2 y 6 estaciones (el primero) y entre 2 y 16 estaciones (el segundo):

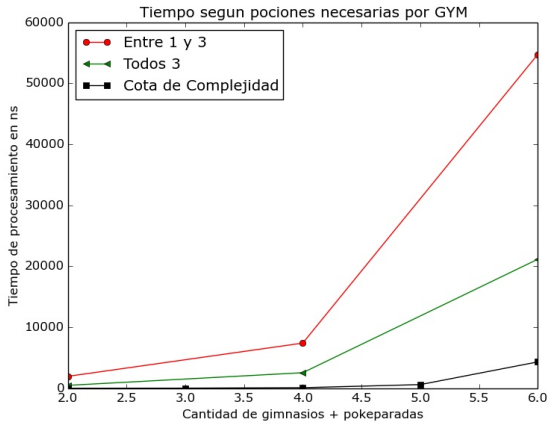


Figura 3: 2 a 6 estaciones

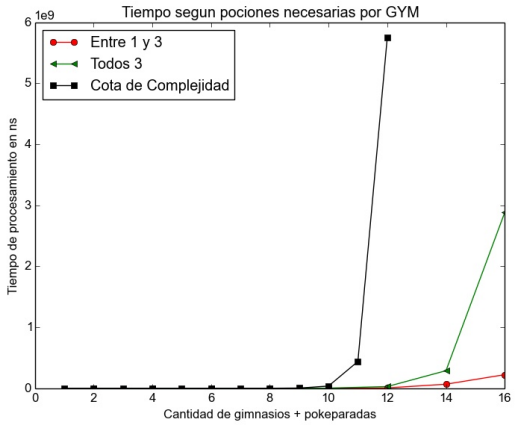


Figura 4: 2 a 16 estaciones

Observamos que, como esperabamos, a igual tamaño de entrada, las instancias que necesitan menor cantidad de pociones por gimnasio toman menos tiempo que las instancias que necesitan 3 o más pociones por gimnasio. Esto muestra que nuestra hipótesis funciona para los resultados de este experimento, por lo que concluímos que la poda de eliminar ramas en las que se construye una solución peor que la mejor hallada hasta el momento también resulta de utilidad cuando hay mayor cantidad de soluciones posibles.

Tomando los datos de los experimentos 1 y 2, pudimos notar que a pesar de que hay ciertos valores que no aparecen en la cota de complejidad, los mismos pueden cambiar el tiempo de ejecución del algoritmo, sin embargo estos cambios ocurren en gran parte gracias a las podas que sí dependen de estos valores, por lo que teóricamente la cota de complejidad esta bien planteada pero en la práctica la número de operaciones (y por lo tanto, el tiempo de ejecución) puede ser mucho menor.

2.4.3. Experimento 3

En este último experimento quisimos observar el tiempo de ejecución del algoritmo cuando varía la cantidad de pokeparadas. Para eso, creamos 20 instancias en las que se fija la cantidad de gimnasios (en 4 para las primeras 10 y 5 para las últimas), se setea un tamaño fijo para la mochila suficiente para que se puedan guardar todas las pociones al mismo tiempo (100 en este caso) y aumentamos linealmente la cantidad de pokeparadas entre 1 y 10. Habiendo realizado el análisis de los experimentos anteriores y teniendo en cuenta el marco teórico de este ejercicio, propusimos como hipótesis que el tiempo de ejecución aumentaría de acuerdo a la cantidad de estaciones totales, es decir, para las primeras 10 instancias el tiempo de ejecución será menor que para las últimas 10 y además para cada instancia dentro de esas 10, la siguiente tomaría más tiempo que la anterior, ya que se agrega una pokeparada cada vez. El gráfico con los resultados es el siguiente:

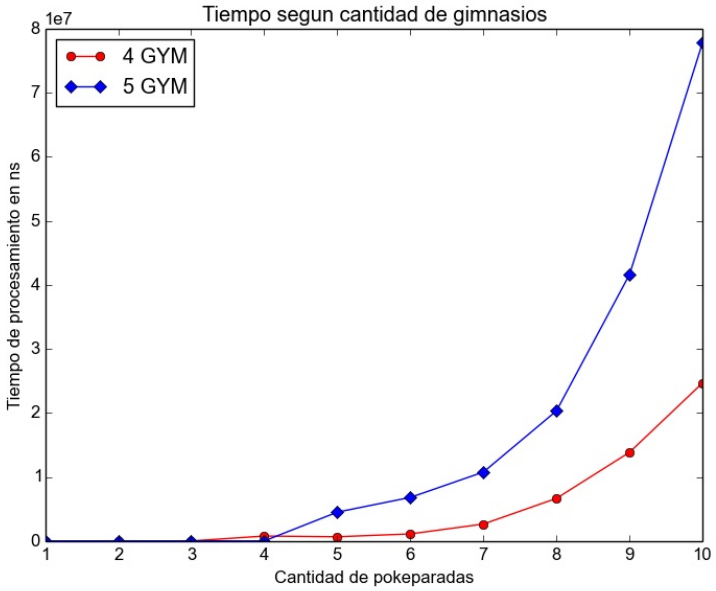


Figura 5

Confirmando nuestra hipótesis, el tiempo de ejecución aumentó en cada instancia y los casos con 5 gimnasios tomaron más tiempo que los de 4. Lo que sí nos llamó la atención fue que las líneas de los graficos no tienen una forma muy similar a la de la cota de complejidad, pero por lo dicho anteriormente, esto puede deberse a que gracias a las podas los tiempos de ejecución pueden ser mucho menores que lo que la teoría puede asegurar.

3. Ejercicio 2: Heurística constructiva golosa

3.1. Descripción del problema y solución propuesta

En este punto, se pidió realizar un algoritmo basado en una heurística constructiva golosa que resuelva el problema en cuestión. El mismo al tratarse de una heurística puede no dar la solución óptima, y puede no devolver una solución aunque la misma exista.

Se decidió utilizar como heurística golosa para resolver esta variante del TSP, la técnica del vecino más cercano. La misma consiste en lo siguiente: en cada momento, el algoritmo selecciona como próximo nodo a visitar el que se encuentre a menor distancia del actual. Como en este caso puede ocurrir que no se pueda ir al nodo más cercano debido a que no se poseen las suficientes pociones (para enfrentarlo en caso de que sea un gimnasio), el algoritmo selecciona al nodo más cercano que se pueda ir. El funcionamiento es el siguiente: se busca un nodo inicial al que se pueda ir, ya sea porque alcanzan las pociones (para el caso de un gimnasio) o porque se pueden recibir pociones (porque la mochila no está llena). Una vez identificado un nodo inicial se busca la estación (gimnasio o pokeparada) más cercana a la que se pueda ir. Este procedimiento se repite sucesivamente hasta que se visiten todos los gimnasios o hasta que no se pueda ir a ningún nodo (porque la mochila está llena y no alcanza para ir a ningun gimnasio o porque no quedan mas pokeparadas y no se puede vencer a ningún gimnasio). Este método no garantiza que la solución encontrada sea óptima ya que puede pasar que se recorran pokeparadas innecesariamente sólo por el hecho de estar próximas entre sí y además puede ser que no se encuentre una solución aunque si haya alguna porque se pueden desperdiciar pociones en una instancia similiar a la anterior mencionada. Basándonos en la técnica del vecino más cercano, las instancias en los cuales los gimnasios estén cerca uno de los otros y se tenga la cantidad suficiente de pociones para ir a ellos, van a ser resueltas de manera muy cercana a la óptima. Los casos en los cuales las pokeparadas estén muy cercas unas de otras, van a tener peores resultados con esta técnica ya que se van a recorrer nodos que en su mayoría son innecesarios, ya que se podría directamente pasar por los gimnasios.

3.2. Detalles implementativos

Las clases y estructuras utilizadas en el algoritmo son las mismas que las del primer ejericio. El algoritmo propuesto para la heurísitica golosa es el siguiente:

```
solverEj2(lista<Estacion> estaciones, matriz<double> distancias, cantidad_gimnasios,
          cantidad_pokeparadas, mochila_size)

  lista<int> camino_nulo
  solucion = (-1,-1,camino_nulo)
  si puedo ir a algun nodo inicial
      lista<Estacion> visitados
      id = donde_voy(estaciones, 0, mochila_size)
      index = indice_estacion_con_id(id, estaciones)
      visitados.agregar(estaciones[index])
      potasActuales = (si es gimnasio o mochila_size >3)
                          estaciones[index].potas
                      si no
                          mochila_size
      estaciones.borrar(index)
      greedy_capturar_gimnasios(estaciones,distancias,cantidad_gimnasios,cantidad_pokeparadas,
                                mochila_size,visitados,potasActuales,id,solucion)

  devolver solucion
```

La función que resuelve el problema es *solverEj2*. Para la implementación de las listas se utilizó la clase vector y para la matriz que contiene las distancias se utilizó un vector de vector. El algoritmo lo primero que realiza es obtener el id de una estación inicial. El mismo es obtenido llamando a una función

donde voy. Esta función recibe por parámetros una lista con las estaciones, la cantidad de pociones que se posee y el tamaño de la mochila, y recorre las estaciones y por cada una de ellas se fija si alcanzan las pociones para enfrentarse al gimnasio (en caso que la estación sea un gimnasio) o si puede recibir las pociones (en caso de ser una pokeparada y no tener la mochila llena). De haber alguna, devuelve el id de la primera estación que encuentra. En caso contrario la función devuelve -1, indicando que no hay ninguna estación a la cual se puede ir. Si la función devuelve -1 no se puede tomar ninguna estación inicial por lo que simplemente se devuelve lo solicitado en el enunciado (-1). En caso contrario se utiliza el id y se recupera el índice de la estación en la lista de estaciones llamando a la función *indice estación con id* para luego agregar a la lista de visitados la estación correspondiente a este índice. De esta forma, tenemos el primer nodo (estación) visitado. Se definen las pociones actuales como las pociones de esta estación (en el caso de que sea un gimnasio o que el tamaño de la mochila sea mayor a 3) o como el tamaño de la mochila (en caso contrario). Se borra de la lista de estaciones la estación visitada, para indicar que la misma ya fue visitada. Finalmente se llama a la función *greedy capturar gimnasios* la cual realiza un procedimiento similar al anteriormente mencionado hasta que termine de recorrer todos los gimnasios (en caso de que pueda). La función *greedy capturar gimnasios* recibe la lista de estaciones, una matriz de distancias (que contiene todas las distancias desde todas las estaciones a todas las demás, el mismo que se utilizó en el primer ejercicio), la cantidad de gimnasios (n), la cantidad de pokeparadas (m), el tamaño de la mochila (k), la lista de estaciones visitadas, las pociones actuales que se posee, el id de la estación actual y una tupla que contiene la distancia total, la cantidad de estaciones que hay que recorrer, y una lista con los ids en orden de las estaciones que hay que recorrer.

```
greedy_capturar_gimnasios(lista<Estacion> estaciones, matriz<double> distancias,
                          cantidad_gimnasios, cantidad_pokeparadas, mochila_size,
                          lista<Estacion> visitados, potasActuales, id_estacion_actual,
                          tupla<double,int,lista<int> > soluciones)
  i = 0
  mientras i < (n+m) y no es_solucion(estaciones)
    ordenar(estaciones, distancias, id_estacion_actual)
    id = donde_voy(estaciones, potasActuales, k)
    si id es igual a -1
       i = n + m
    si no
       index = indice_estacion_con_id(id, estaciones)
       visitados.agregar(estaciones[index])
       potasActuales += estaciones[index].potas
       estaciones.borrar(index)
       id_estacion_actual = id
       i = i + 1
  fin mientras
  si es_solucion(estaciones)
      distancia = distancia_acumulada(visitados,distancias)
      lista<int> camino

      for (u = 0 hasta visitados.size() - 1)
        camino.agregar(visitados[u].id)

      soluciones = (distancia, visitados.size(), camino)
```

En la función *greedy capturar gimnasios* hay un ciclo que se ejecuta n + m veces (la cantidad de estaciones total), mientras no se haya encontrado una solución. Para saber si se encontró una solución o no, se recorren todas las estaciones para verificar que no haya ningún gimnasio sin visitar. En el cuerpo del ciclo se realiza el ordenamiento de las estaciones, la parte fundamental de la heurística greedy. El mismo consiste en una función *ordenar* que recibe por parámetro la lista de estaciones, la matriz de distancias, y el id de la estación actual en la cual se está analizando. Luego del ordenamiento se llama

nuevamente a la función que devuelve el id de la siguiente estación que hay que visitar. Como ahora la lista de estaciones está ordenada y se recorre de adelante para atrás, esta función va a devolver el id de la estación más cercana a la que se puede ir. Si devuelve -1, se actualiza el índice del ciclo con el valor n + m para que termine ya que no se puede avanzar. Si no, se recupera el índice de la estación con ese id, se agrega la estación nueva a la lista de visitados y se actualizan las pociones actuales, de acuerdo al tipo de estación que se visitó. Se borra la estación de la lista de estaciones para que no sea considerada nuevamente y se actualizan el id de la estación actual con el nuevo id y se incrementa el índice del ciclo. Luego del ciclo, se verifica que sea una solución lo que se haya encontrado y en caso de serlo se suma las distancias de todos los nodos visitados y se actualiza la tupla de solución con la distancia total, la cantidad de estaciones visitadas y el camino realizado, para que al retornar de la función la tupla solución contenga la solución al problema.

```
ordenar(lista<Estacion> estaciones, matriz<double> distancias, id_estacion_actual)

  lista<int> ids_vistos
  ids_vistos.agregar(id_estacion_actual)
  for (i = 0 hasta estaciones.size() - 1)
    id_mas_cercano = id_mas_cercano_que_no_viste(distancias[id_estacion_actual],
                                                 ids_vistos, estaciones)
    ids_vistos.agregar(id_mas_cercano)
    swap(id_mas_cercano, i, estaciones)
```

Lo que hace la función *ordenar* es recorrer la lista de distancias de la estación actual y ordena la lista de estaciones de acuerdo a esta lista de distancias. De esta manera en cada iteración la lista de estaciones queda ordenado por distancia relativa a la estación actual que se está analizando.

3.3. Complejidad teórica

La función que resuelve el problema (solverEj2) realiza las siguientes operaciones con el siguiente costo temporal: La inicialización de la lista y la tupla se realizan en $O(1)$. La función que es llamada *donde voy* tiene complejidad temporal $O(n+m)$ ya que recorre todas las estaciones analizando en $O(1)$ si se puede ir a alguna de ellas. La función que devuelve el índice de la estación con un id determinado también tiene complejidad $O(n+m)$ ya que recorre toda la lista de estaciones en el peor caso. El agregar al final de la lista (implementada sobre vector) se realiza en $O(1)$ amortizado y el borrar (en cualquier posición) se realiza en $O(n+m)$. Finalmente falta sólo analizar la complejidad de la función *greedy capturar gimnasios* y con ella determinar la complejidad resultante de solverEj2. Esta última función tiene un ciclo que se ejecuta en el peor caso $n+m$ veces. En cada una de las iteraciones se pregunta si se encontró una solución o no. Para ello se recorre la lista de estaciones para verificar que no quede ningun gimnasio sin visitar. Esto por lo tanto tiene una complejidad $O(n+m)$ en el peor caso. En el cuerpo del ciclo se llama a la función ordenar la cual recorre la lista de estaciones (en $O(n+m)$) y en cada iteración recupera el id del más cercano. Para recuperar este id, la función *id mas cercano que no viste* recorre todas las distancias de las demas estaciones en $O(n+m)$, preguntando en cada caso si esa estación se vio o no (en $O(n+m)$). Luego ordenar en cada iteración realiza el swap correspondiente de estaciones, que para el mismo necesita recorrer nuevamente todas las estaciones ($O(n+m)$) para poder recuperar el id de la estación que quiere intercambiar. En conclusión, ordenar realiza en el peor caso $O(n+m)*O(n+m)*O(n+m)$ operaciones resultanto en una complejidad temporal de $O(n+m)^3$. Después de ordenar, la función greedy llama a la función donde voy ($O(n+m)$). Se realiza una comparación con el id devuelto de esta función en $O(1)$ y si es distinto de -1 se llama a la función que devuelve el índice de la estación ($O(n+m)$). Las otras operaciones del ciclo se realizan en tiempo cosntante ($O(1)$). Después del ciclo se pregunta si se encontró o no solución chequeando la lista de estaciones ($O(n+m)$ en el peor caso). La distancia acumulada se calcula iterando todos las estaciones vistas ($O(n+m)$ en el peor caso) y se arma el camino ($O(n+m)$ en el peor caso). En definitiva el ciclo es el que domina la complejidad de la función ya que por lo dicho realiza $O(n+m)*O(n+m)*O(n+m)*O(n+m)*O(n+m) = O(n+m)^5$ operaciones. Por lo tanto la función solverEj2 tiene una complejidad temporal teórica de $O(n+m)^5$. La complejidad espacial es la

misma que la del ejercicio 1 ya que se utiliza para representar las distancias una lista con tamaño igual a la cantidad de estaciones en el cual en cada posición se tienen las distancias a todas las demás estaciones, lo que da un total de $O((n+m)^2)$.

3.4. Comportamiento del algoritmo

El algoritmo al tratarse de una heurística no devuelve una solución exacta en todos los casos. A continuación se procede a realizar un análisis del comportamiento del algoritmo para distintos tipos de instancias. El peor caso es que el algoritmo no devuelva ninguna solución aunque la misma exista. Esto puede suceder en instancias en las cuales las pokeparadas están muy próximas entre sí por lo que el algoritmo al basarse en el vecino más cercano las va a recorrer a todas (siempre y cuando el tamaño de la mochila sea lo suficientemente grande como para recuperar pociones en todas las pokeparadas) antes que los gimnasios. De esta forma puede suceder que haya pociones que se desperdicien porque cada pokeparada devuelve 3 pociones y si en la mochila caben sólo 1 o 2 pociones se van a descartar 2 o 1 poción respectivamente. De esta forma puede suceder que debido a ese descarte no alcancen para vencer a un gimnasio. La siguiente instancia es un ejemplo que cumple con esas características:

```
2 3 7
1 1 2
2 2 6
7 7
8 8
9 9
```

Para este caso el algoritmo propuesto no devuelve solución. Lo que realiza es lo siguiente: toma la primer estación a la que puede ir que en este caso es la que se encuentra en (7,7). Al tratarse de una pokeparada recibe 3 pociones. Luego la estación mas cercana a la que puede ir es (8,8) que es otra pokeparada en donde recibe otras 3 pociones (en total tiene 6). Luego la próxima estación más cercana es otra pokeparada (9,9) de la cual solo puede recibir 1 poción (porque el tamaño de la mochila es 7 y ya tenía 6). Estas dos pociones que desperdicia son las que hacen que despúes no le alcance para vencer a los dos gimnasios. La última estación a la que visita es (2,2) por proximidad y en esta gasta 6 pociones, quedandole una sola disponible en la mochila. Finalmente le queda una única estación (el gimnasio que se encuentra en 1,1) pero que no puede vencer porque le falta una poción y no quedan mas pokeparadas sin recorrer. De esta forma el algoritmo retorna que no hay solución cuando si existe y la óptima es la siguiente:

D = 25.4558 k = 5 i = 5 4 2 3 1

Es decir se pueden recorrer las estaciones (8,8) y (9,9) obteniendo 6 pociones, enfrentar el gimnasio que se encuentra en (2,2), usando las 6 pociones, ir a la pokeparada (7,7) y finalmente venciendo al gimnasio en (1,1).

Además de no devolver la solución hay casos en donde el algoritmo si devuelve una solución pero la misma no es óptima y hace que se recorran todas las estaciones, como por ejemplo el siguiente:

```
3 3 7
1 1 1
2 2 1
3 3 1
7 7
8 8
9 9
```

En este caso el algoritmo devuelve lo siguiente:

D= 14.1421 k= 6 i= 4 5 6 3 2 1

Es decir recorre todas las pokeparadas y después todos los gimnasios cuando el óptimo es ir a la pokeparada en (7,7) y despues a los gimnasios (3,3),(2,2) y (1,1).

Hay casos igualmente que el algoritmo devuelve la solución óptima como por ejemplo en el siguiente:

```
3 3 6
1 1 1
2 2 1
3 3 1
4 4
8 8
9 9
```

En este caso el algoritmo primero recorre la estación (4,4) que es una pokeparada de donde obtiene 3 pociones y después por proximidad visita las estaciones (3,3),(2,2) y (1,1) ya que puede enfrentar a todos los gimnasios porque todos requieren sólo 1 poción.

3.5. Experimentación

Los experimentos que realizamos para observar los tiempos de ejecución del algoritmo en función del tamaño de entrada consistieron en el análisis de casos que contemplan:

- Configuraciones de valores en los cuales se evaluan distintas cantidades de estaciones, tomando como capacidad de la mochila, un tamaño superior a todo lo que se puede obtener y un valor menor a todo lo que se puede obtener.
- Casos random generados a partir de la librería Random de la STL de C++ utilizando distribución uniforme

Para todos los casos, el algoritmo se ejecutó 150 veces con cada instancia para intentar tener la mayor presición posible en cuanto a tiempo de ejecución.

3.5.1. Analisis de configuraciones de valores que están en la cota de complejidad

En esta instancia generamos entradas que aumentaban linealmente de 1 a 100 la cantidad de pokeparadas, fijando el tamaño de la mochila en 100 y marcando a cada gimnasio con 3 pociones necesarias para vencerlo. Además, las ubicaciones de cada estación se configuraron cada una a distancia 1 de la anterior y la siguiente: para la estación i la ubicación es (i,i). Se probaron estos casos con 5, 10 y 15 gimnasios, en donde cada gimnasio requiere 3 pociones para vencerlo. Se esperó que a medida que se aumenta la cantidad de gimnasios, aumente también el tiempo de ejecución del mismo. Las primeras instancias no tienen solución ya que la cantidad de pokeparadas no alcanza para enfrentar a los gimnasios.

A continuación en el siguiente gráfico se puede observar los tiempos de ejecución:

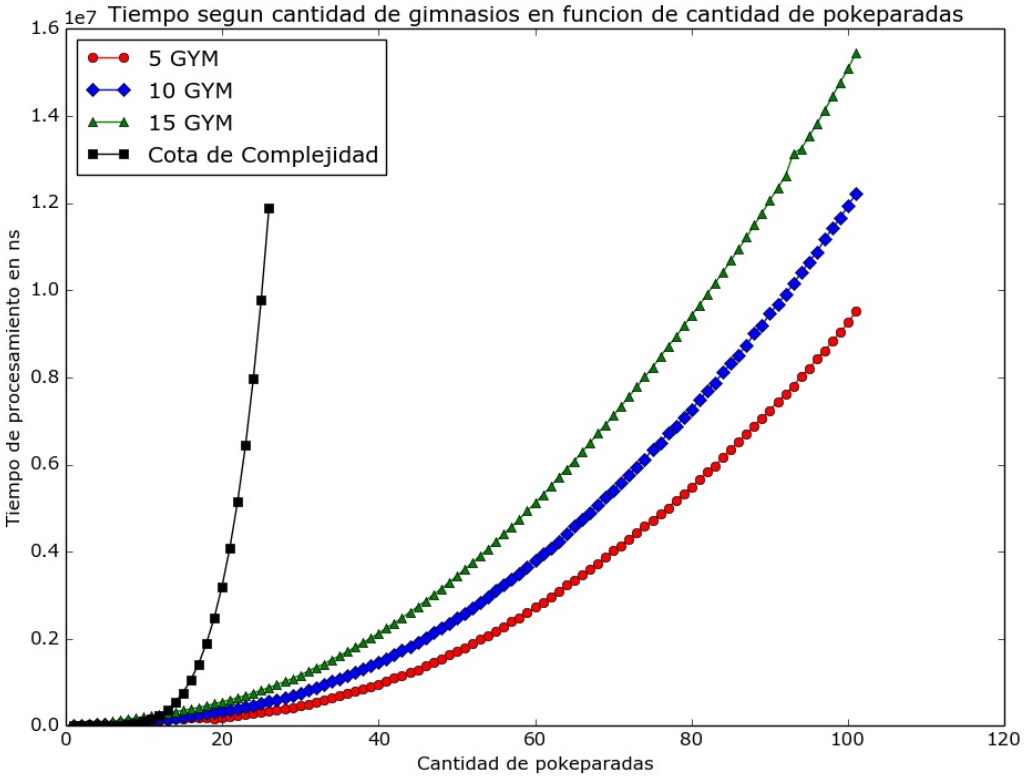


Figura 6

Como era de esperar, el caso de los 15 gimnasios demandó más tiempo de ejecución que los otros. Para las primeras instancias los tiempos fueron muy similares ya que no hay solución posible y el algoritmo corta rápido la ejecución. En el gráfico además se agregó una línea que representa (a modo de referencia) la cota de complejidad. Se puede también observar que a medida que aumenta el número de pokeparadas también aumenta el tiempo de ejecución. Esto se debe a que debido a la construcción de las intancias, el algoritmo recorre las pokeparadas primero, ya que están muy próximas entre sí. Cabe destacar, que al colocar un tamaño de mochila en 100, una vez que se recorren 34 pokeparadas la mochila se llena, por lo que el algoritmo deja de recorrer las pokeparadas aunque estén próximas y comienza a recorrer los gimnasios. Por lo tanto se puede observar que a partir de 34 pokeparadas el tiempo de ejecución no cambia significativamente, ya que no se visitan las demás pokeparadas. El cambio en el tiempo de ejecución se debe a que si bien no se recorren más pokeparadas, si se aumenta su número aumenta también el tiempo en ordenar las estaciones por distancia en cada estación, además de los tiempos de busquéda en las estaciones.

Se realizó el mismo experimento pero utilizando un tamaño de mochila superior a todo lo que se puede obtener (10000) para poder comprobar que el tiempo de ejecución cambia significativamente en comparación con el experimento anterior ya que ahora el algoritmo recorre todas las pokeparadas por proximidad. Los resultados son los siguientes:

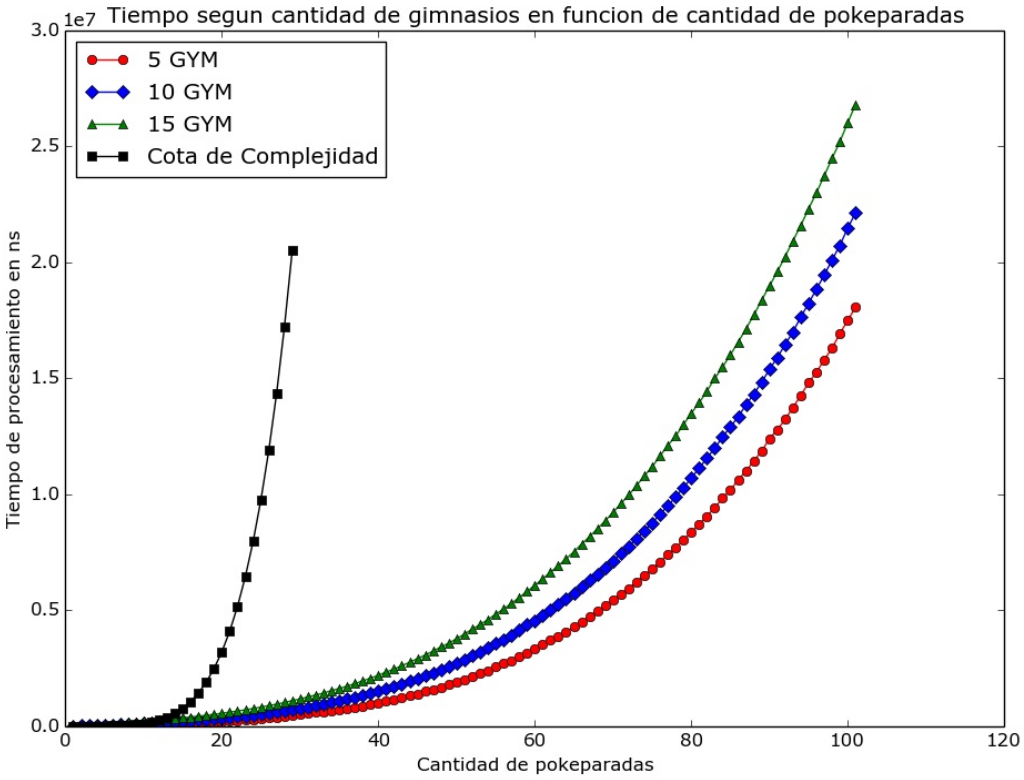


Figura 7

Del gráfico se puede sacar como conclusión que al usar un tamaño de mochila superior a todo lo que se puede obtener, cambia notablemente el tiempo de ejecución en función de la cantidad de pokeparadas, ya que se recorren todas y no se para ya que la mochila no se puede llenar. Se mantiene que también al aumentar la cantidad de gimnasios cambia el tiempo considerablemente.

3.5.2. Analisis de casos aleatorios

En esta instancia se generaron casos aleatorios usando la librería Random de la STL de C++, variando la cantidad de gimnasios y pokeparadas entre 1 y 8 y el tamaño de la mochila entre 0 y 48, la cantidad de pociones entre 0 y 24. Las posiciones de las estaciones varían entre 0 y 100. Para la mayoría de los casos el algoritmo no retornó solución. En el siguiente gráfico se pueden observar los resultados:

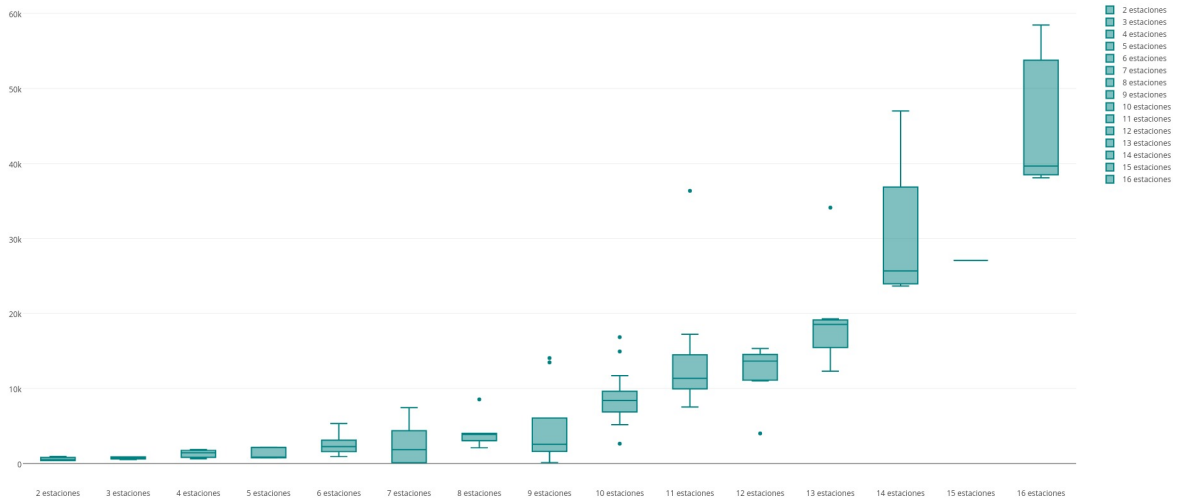


Figura 8

El gráfico es un boxplot el cual tiene agrupado los tiempos de ejecución por cantidad de estaciones. Se optó por realizar de esta forma ya que al tratarse de casos random, hubo varias instancias con la misma cantidad de estaciones generadas. Como era de esperarse, las instancias que más tiempo demandaron fueron las que más estaciones tenian. En los primeros casos, con pocas estaciones, los tiempos estuvieron mucho menos dispersos y no hubo prácticamente outliers. Para las instancias más grandes se observa una dispersión mucho más amplia (longitud de la caja) debido a que al tener más estaciones y depender fuertemente de su distribución espacial, el algortimo en algunos casos pudo recorrer todos los gimnasios (cuando encontró solución) y en otros casos, que fueron la mayoría no encontró solución, deteniendosé en su búsqueda en distintas iteraciones del mismo. En algunos casos también recorrió todas las estaciones tomando más tiempo de ejecución. Los puntos aislados observados se tratan de outliers, instancias que demandaron un tiempo muy superior o menor al resto de las demás. El unico caso particular es cuando las estaciones fueron 15 que al ser pocas las instancias generadas con random, presentó un comportamiento similar al de las que tenian pocas estaciones.

Finalmente se corrió el expermiento aleatorio mencionado en la introducción. Los resultados obtenidos se pueden visualizar en el siguiente gráfico:

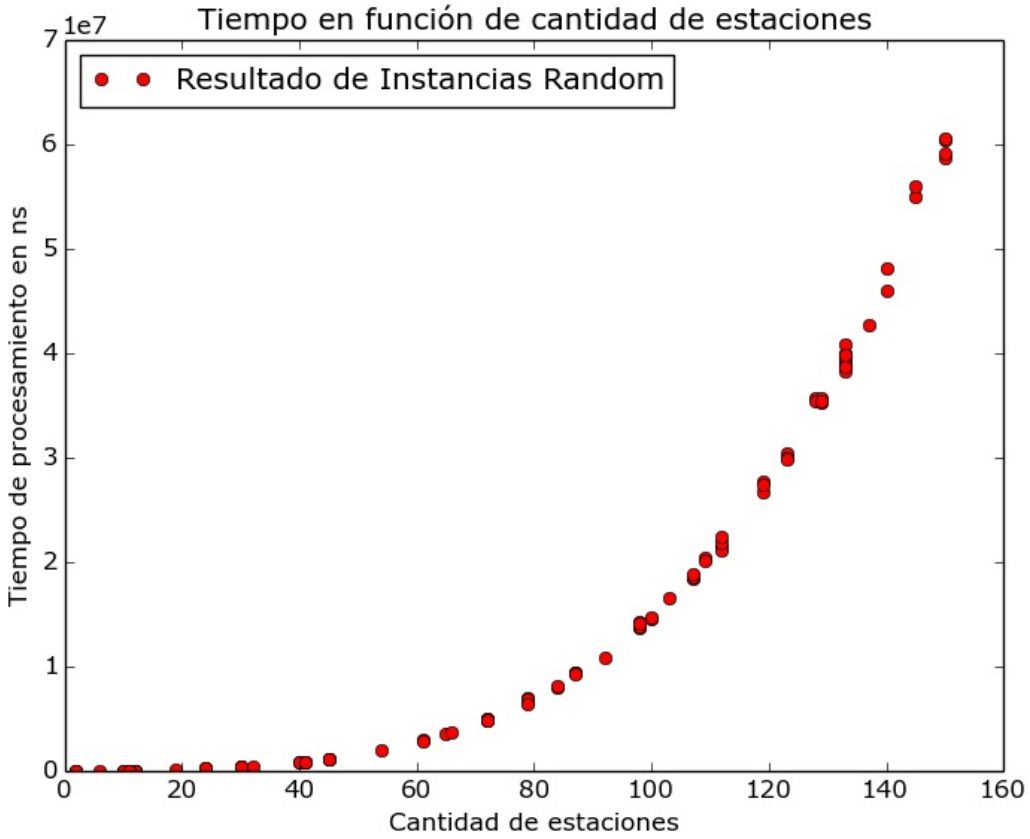


Figura 9

Se puede observar como en los demás experimentos que al aumentar la cantidad de estaciones también aumenta el tiempo de ejecución. La mayoría de los casos pudieron ser resueltos por el algoritmo.

4. Ejercicio 3: Búsqueda Local

4.1. Descripción del problema y solución propuesta

Una vez obtenida una heurística greedy para resolver nuestra variante de TSP, utilizamos una **Búsqueda Local** para analizar una mayor cantidad de soluciones posibles. Es decir, dada una solución inicial S (obtenida de aplicar a una instancia de entrada, el algoritmo heurístico implementado en el Ejercicio 2) se obtienen los vecinos de S, es decir, un "vecindario" de soluciones. Y de este vecindario, se obtiene el vecino cuya solución mejore a S. Luego se realiza una búsqueda local sobre este vecino.

Sea V este vecindario, $\forall S^* \in V, S^* = h(S)$ h es una función que obtiene un vecino a partir de S, perteneciente a V. En otras palabras, h altera la solución S para encontrar una solución parecida que pueda mejorar el resultado. Por otro lado, se cuenta con una función f que da valor a cada una de las soluciones. En nuestro caso, sean S y $S^{'}$, diremos que una solución S es mejor que $S^{'}$ si $f(S) < f(S^{'})$. Luego para cada $S^* \in V$, se evalúa f(S^*).

Dado V el conjunto de vecinos del Vecindario y S la solución original, si $\forall S^* \in V$, $f(S) < f(S^*)$ se devuelve la solución S. En caso contrario, se recorre el vecindario de soluciones y se aplica nuevamente la búsqueda local sobre el $S^* \in Vecindario$ cuyo valor sea el mínimo, y se reemplaza a S por S^*. Esto se repite hasta que no haya un S^* que cumpla $f(S^*) < f(S)$. Una vez que esto ocurre, se devuelve S.

De esta manera, a partir de una solución se estará explorando sus vecinos, y la búsqueda local se detendrá al hallar una mínimo local.

4.1.1. Vecindad A

Se realiza una búsqueda local aplicado a nuestro problema de la siguiente manera. La solución inicial S, se obtiene de aplicar la heurística del vecino más cercano (explicada en el Ejercicio 2) a los parámetros pasados por entrada. En nuestro caso, S consiste en una tupla con distancia, recorrido y cantidad de estaciones.

Una vez obtenida esta solución, contamos con la función **h** la cual generará la primer vecindad. En este caso, la función h se denomina *dameVecindario*, y se encarga de intercambiar únicamente las *pokeparadas* dentro del recorrido de S. Por cada intercambio generado, se guarda este nuevo recorrido en un vector.

```
vector<vector<int>> dameVecindario(lista<Estacion> estaciones, lista<int> recorrido)
    vector<vector<int>> vecindario;
    Para i = 0...recorrido.size() - 1
        Si la estacion pasada en el recorrido[i] no es gimnasio
            Para j = i+1 ... recorrido.size()
                Si la estacion pasada en el recorrido[j] no es gimnasio
                    Creo un vector estadoAux <-- recorrido
                    Hago swap entre i y j en estadoAux
                    vecindario.push_back(estadoAux);
                Fin si
            Fin
        Fin Si
    Fin
    Devolver vecindario
```

En nuestro caso, una solución es mejor que otra si una logra conquistar todos los Gimnasios recorriendo menor distancia que la otra, manteniendo siempre el invariante de no pasar por la misma estación más de una vez. Una vez obtenido el vecindario, la función f evalúa cada solución. En nuestro caso f se llama "dameDistancia" y se encarga de calcular la distancia de cada solución obtenida, guardándola en el vector vecindario. Luego compara cada una con S.

Si la distancia de algún recorrido nuevo en el vecindario (por ejemplo S^*) es menor que la distancia que recorrió S, se repite el ciclo de búsqueda local sobre algún S^*. Caso contrario, se devuelve la solución S con su recorrido, su distancia y la cantidad de estaciones por donde pasó.

4.1.2. Vecindad B

A diferencia de la Vecindad A, la función h del vecindario B, se llama "dameVecindario2" y se encarga de intercambiar únicamente los *gimnasios* dentro del recorrido S, siempre que cada intercambio de gimnasios en el recorrido no cause que alguno de los gimnasios no pueda ser conquistado.

```
vector<vector<int>> dameVecindario2(lista<Estacion> estaciones, lista<int> recorrido,
                                    matriz<double, double> distancias, int k)
    vector<vector<int>> vecindario;
    Para i = 0...recorrido.size() - 1
        Si la estación pasada en el recorrido[i] es gimnasio
            Para j = i+1 ... recorrido.size()
                Si la estación pasada en el recorrido[j] es gimnasio
                    Creo un vector estadoAux <-- recorrido
                    Hago swap entre i y j en estadoAux
                    Si el recorrido sigue siendo válido (se conquistan todos los gimnasios)
                        vecindario.push_back(estadoAux);
                    Fin si
                Fin si
            Fin
        Fin Si
    Fin
    Devolver vecindario
```

El resto del procedimiento es equivalente al explicado para la búsqueda local que incluye el Vecindario A.

4.2. Complejidad teórica

Para la complejidad teórica, debemos mostrar por cada iteración de la búsqueda local cual será la cota asintótica.

Para empezar, hablaremos de la **Vecindad A**. Como fue explicado anteriormente, se encarga de crear las posibles combinaciones de intercambiar la posición de las pokeparadas en el recorrido. Para calcular la complejidad en cada iteración de la búsqueda local, sean N los nodos gimnasio y M los nodos pokeparadas, se deben tener en cuenta:

- Para ingresar a la búsqueda local, se copia el vector recorrido, de la solución inicial S, esto cuesta O(N+M).
- Luego se entra al ciclo principal de búsqueda local que en el peor de los casos tendrá M! iteraciones. Esto se debe a que, dado que en cada iteración se permutan ciertas pokeparadas, en el peor de los casos, siempre encuentra una solución mejor. Va a tomar O(M!) iteraciones recorrer todas las permutaciones posibles.
- Dentro del ciclo principal, se busca el vecindario de una solución. Para esto se llama a la función *dameVecindario* con el vector *recorrido*. La misma función, recorre el vector hasta que el i-ésimo elemento sea pokeparada. Luego recorre el vector restante, desde i+1 hasta recorrido.size()-1. Es decir, la complejidad del ciclo, siendo r la cantidad de estaciones del recorrido, es de

$$O(\sum_{i=0}^{r-1} i)$$

Que dada la progresión aritmética,

$$\sum_{i=0}^{r-1} i = r * (r-1)$$

Esto quedaría,

$$O(r * (r-1)/2)$$

Además, como mencionamos anteriormente, el tamaño del recorrido en el peor caso es O(N+M). Luego,

$$O((N+M)) * (N+M-1)/2)$$

Luego, dentro del ciclo, se copia el vector recorrido en un vector *estadoAux*, la copia del vector es O(N+M), y si esto se realiza en cada ciclo,

$$O(((N+M)) * (N+M-1)/2) * (N+M)) = O((N+M)^3)$$

- Una vez obtenido el vecindario se recorre el vector *vecindario* y por cada vecino se itera sobre su recorrido calculando la distancia. Dado que las iteraciones dentro de *dameVecindario*, empieza en la i-ésima pokeparada, y por cada elemento a partir del i+1, que también sea pokeparada, se genera un intercambio de elemento dentro del vector, entonces la cantidad de vecinos distintos serán,

$\sum_{i=0}^{M-1} i = M(M-1)/2$

Luego la cantidad de vecinos diferentes, es decir el tamaño total del vector vecindario es de $M(M-1)/2$. Calcular la distancia, implica por cada vecino, recorrer todo el vector *recorrido*, esto es O(N+M). Esto lo realiza para los M(M-1)/2 vecinos. Es decir

$$O((M(M-1)/2) * (N+M)) = O(M^2 * (N+M))$$

- Finalmente, dada la solución, se genera una tupla en O(1) y se devuelve esta solución.

Es decir, la complejidad quedaría

$$O(N+M) + O(M!) * (O((N+M)^3) + O(M^2 * (N+M)) =$$

$$O(M!) * (O((N+M)^3) + O(M^2 * (N+M)) =$$

$$O(M! * ((N+M)^3 + M^2 * (N+M))) =$$

$$O(M! * ((N+M)^3))$$

Pues $O(M^2 * (N+M)) = O((N+M)^3)$

Finalmente, la complejidad total es de

$$O(M! * ((N+M)^3))$$

Por otro lado, hablaremos de la complejidad del vecindario B. La misma, comparte gran parte de la complejidad del vecindario A. Por lo que se nombrarán las partes relevantes del algoritmo de búsqueda local con el vecindario B.

- Para ingresar a la búsqueda local, se copia el vector recorrido, de la solución inicial S, esto cuesta O(N+M).
- Luego, busca las permutaciones de los gimnasios, por lo que, el ciclo principal de la búsqueda local, tendrá en el peor de los casos N! iteraciones. Es decir, si intercambiar los gimnasios siempre genera una solución posible, como en cada iteración se permutan algunos gimnasios, en el peor de los casos, siempre encuentra una solución mejor dentro del vecindario. Luego esto tomará O(N!) iteraciones recorrer todas las permutaciones posibles.

- Dentro del ciclo principal, se busca el vecindario de la solución. Para esto, se llama a la función *dameVecindario2* con el vector *recorrido*. Hace las mismas iteraciones que en el vecindario A, es decir

$$O((N+M)*(N+M-1)/2)$$

Con la diferencia que, sea r = recorrido.size(), copiar el vector*recorrido* en *estadoAux* cuesta O(r), y además debe chequear que el nuevo recorrido es válido. Para esto, recorre nuevamente el vector *estadoAux*, viendo que se cumplan diferentes condiciones. Luego como fue mencionado, dado que se copia *recorrido* en *estadoAux* y $r = O(N+M)$, entonces $estadoAux.size() = r = O(N+M)$. Luego copiar el vector es O(N+M) y recorrer el vector estadoAux, es

$$O(2*(N+M)) = O(N+M)$$

Esto quiere decir que, por cada ciclo de *dameVecindario2*, debe copiar el vector y chequear que sea válido. Y las iteraciones de esta función eran O((N+M)*(N+M-1)/2). Luego quedaría O((N+M)*(N+M-1)/2*(N+M)), entonces

$$O((N+M)*(N+M-1)/2*(N+M)) = O((N+M)^3)$$

- Luego una vez obtenido el vecindario, se calcula al igual que en el **vecindario A** las distancias de cada vecino en $O(M^2*(N+M))$
- Por último en O(1) se genera una tupla como solución y se devuelve.

Entonces, la complejidad quedaría

$$O(N+M) + O(N!)*(O((N+M)^3) + O(M^2*(N+M))) =$$

$$O(N!)*(O((N+M)^3) + O(M^2*(N+M))) =$$

$$O(N!*((N+M)^3 + M^2*(N+M))) =$$

$$O(N!*((N+M)^3))$$

Pues $O(M^2*(N+M)) = O((N+M)^3)$

Finalmente, la complejidad total es de

$$O(N!*((N+M)^3))$$

4.3. Detalles Implementativos

Las clases y estructuras utilizadas en el algoritmo son las mismas que las del primer ejercicio. El algoritmo propuesto para las búsquedas locales es el siguiente:

```
solucion busquedaLocal(vector<int> recorridoInicial, vector<Estacion> estaciones,
    matriz<double> distancias, double distancia, bool elegirVecindario)

    matriz<int> vecindario;
    double distanciaMinima;
    double distanciaAux;
    int proximoEstado;
    bool hayanSoluciones = true;
    vector<int> recorrido = recorridoInicial;

    mientras hayanSoluciones
        vecindario = dameVecindario(estaciones, recorrido, elegirVecindario)
        //obtiene un vecindario de soluciones

        proximoEstado = -1;

        si el vecindario no es vacío

            distanciaMinima = distancia;

            //Busco algún vecino con distancia mas chica
            para cada vecino entre 0 y tam(vecindario)

                distanciaAux = dameDistancia(vecindario[vecino], distancias);
                //obtiene las distancia recorridas de la solución vecino-ésima

                si distanciaAux < distanciaMinima
                    distanciaMinima = distanciaAux;
                    proximoEstado = vecino;
                fin si
            fin para cada
        fin si

        si proximoEstado > -1
            recorrido = vecindario[proximoEstado];
            distancia = distanciaMinima;
        sino
            hayanSoluciones = false;
        fin si

    fin mientras

    devolver solucion (distancia, tam(recorrido), recorrido);
fin busquedaLocal
```

El algoritmo obtiene una primera solución **recorridoInicial** en base al algoritmo greedy goloso implementado en el Ejercicio 2.

Luego, busca un vecindario de soluciones en base a la solución obtenida (dado el parámetro **elegirVecindario**, se elige uno de los dos tipos de vecindarios).

Se obtiene la mejor solución del vecindario, es decir, la que menor distancia recorre. Llamémosla S.

Por último se vuelve a repetir el procedimiento, buscando un vecindario de soluciones a partir de la solución S.

Todo este procedimiento se repite hasta que el vecindario de soluciones no contenga ninguna solución mejor que la solución ya obtenida.

4.4. Experimentación

Los experimentos realizados permiten observar las diferentes calidades de solución para ambos vecindarios, es decir, en el primer experimento se comparó junto al resultado exacto dado por backtracking, qué vecindario devolvía la menor distancia posible, mientras que en el segundo experimento, se comparó las soluciones devueltas por búsqueda local contra las devueltas por el algoritmo greedy. Además en ambos experimentos, se comparó la performance de cada vecindario, es decir, se calculó el tiempo en ns por cada uno. En cuanto a instancias, en el primer experimento, utilizamos instancias particulares que serán explicadas en él. Mientras que en el segundo, se generó un mayor número de instancias random, asegurándonos que tengan solución.

4.4.1. Experimento 1

Como hipótesis mantuvimos que, dado que en el vecindario A se intercambian pokeparadas, el campo de soluciones recorridas por él es mayor al del vecindario B, debido a que, al intercambiar pokeparadas de una solución esta sigue siendo solución, mientras que en el vecindario B, por cada intercambio de gimnasios, debe filtrar aquellos que no devuelven una solución válida, teniendo menos margen para generar vecinos. Entonces, **en general**, esto permite que el vecindario A, obtenga mejores soluciones que el B.

4.4.2. Experimento 1: Calidad de resultados

Para experimentar acerca de la calidad de los resultados de ambas vecindades, se generaron distintas comparaciones.

- Por un lado, se comparó cada vecindario por separado con backtracking. Para esto se generaron 66 instancias, de las cuales, las primeras 20 instancias cumplen las mismas condiciones mencionadas en **Ejercicio 1: Experimento 1**. Por otro lado las siguientes 16 instancias cumplen con las mismas condiciones que menciona el **Ejercicio 1: Experimento 2** y por último, fueron generadas 30 instancias que por cada una se aumenta en 1 la cantidad de gimnasios y pokeparadas, manteniendo siempre que la distancia entre cada estación es 1.

- Por otro lado, se compararon los tres algoritmos juntos, corriendo 16 instancias creadas también manualmente. Por cada instancia, aumentamos de la misma manera la cantidad de gimnasios, pokeparadas y capacidad de las mochilas. Al igual que la anterior comparación, se mantuvo que la distancia entre cada estación es 1.

En ambas comparaciones, por cada algoritmo, se corrió 30 veces la misma instancia, sacando un promedio de la distancia devuelta. Luego, en función de la cantidad de estaciones (n+m), se comparó qué distancias devolvían. Los demás valores de entradas son los mismos que se utilizaron para el experimento de performance realizado en el Ejercicio 2.

A continuación en los siguientes gráficos se puede observar la calidad de resultado de cada vecindario en comparación con el algoritmo exacto.

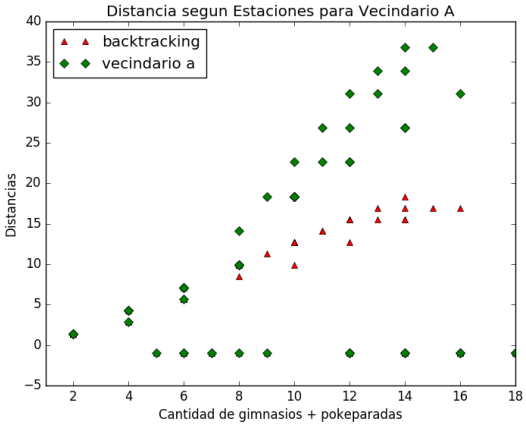


Figura 10: Vecindario A

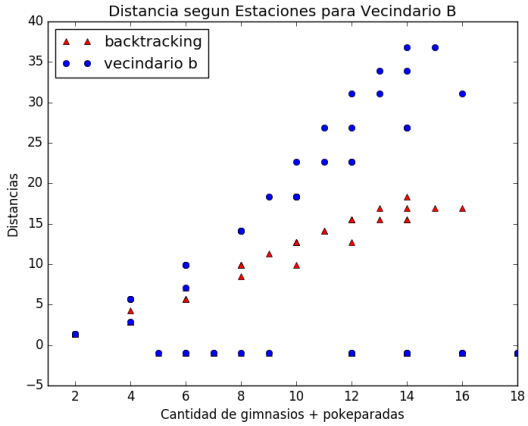


Figura 11: Vecindario B

Para estas instancias, se puede observar que a medida que crece la cantidad de estaciones, crece la brecha entre la solución heurística con búsqueda local, y la solución exacta con backtracking. Esto se debe principalmente a que, a mayor cantidad de estaciones, el campo de soluciones crece. Es decir, mientras que backtracking recorre absolutamente todo el campo de soluciones obteniendo un mínimo global, la búsqueda local a partir de un resultado, recorre únicamente sus vecinos, deteniéndose en los mínimos locales.

Además, no esta demás mencionar, que previo a la búsqueda local, se corre un algoritmo greedy y cuya solución devuelta, dada la configuración de nuestras instancias (las cuales aumentan linealmente la cantidad de estaciones, y por cada estación que se agrega se incrementa su ubicación), en la cual se basará la búsqueda local tiende a empeorar a medida que hicimos crecer la cantidad de estaciones (esto es explicado en el **Ejercicio 2**). Entonces, esto influye directamente en las soluciones encontradas por la búsqueda local, pues debido a que la búsqueda comienza desde una solución la cual puede llegar a diferir de la mejor posible, y que la búsqueda local se detiene en un mínimo local, la mejor solución que encontrará la búsqueda local para cualquier vecindario será peor a la que devuelve el algoritmo exacto, que en nuestro caso, será el mínimo global.

Luego dado el vecindario A, se observa que para instancias chicas, las soluciones se asemejan a las exactas pero sus distancias empiezan a diferenciarse a partir 8 estaciones. Mientras que en el vecindario B, desde un primer momento Backtracking encuentra mejores soluciones. Esto nos asegura de que para ciertas instancias, intercambiar pokeparadas hace una mejor búsqueda en el vecindario de soluciones.

Para poder afirmar nuestra hipótesis, se mostrará a continuación una comparación entre las vecindades y el algoritmo exacto.

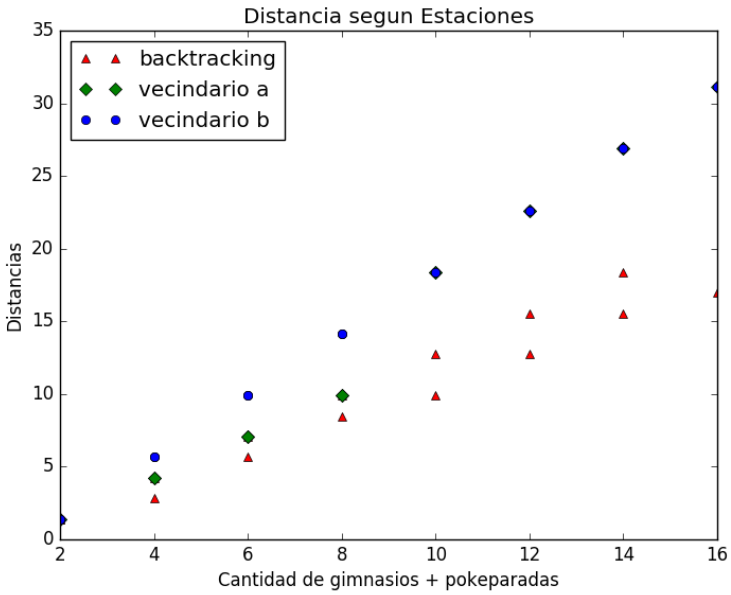


Figura 12: Comparación vecindario y backtracking

Para este caso, se usaron menos instancias para observar el comportamiento de los distintos vecindario en comparación con Backtracking. Como se puede observar, para instancias chicas las distancias obtenidas por el vecindario A se asemejan a las de backtracking, es decir, a las soluciones exactas. Pero a medida que la cantidad de estaciones crecen, la brecha entre Backtracking y vecindario A crece. Esto mismo no sucede con el vecindario B, donde desde un principio mantiene una brecha creciente con el resultado exacto. Por lo tanto, afirma la hipótesis anteriormente mencionada.

4.4.3. Experimento 1: Performance

Para experimentar acerca de la performance de los resultados de ambas vecindades, se generaron distintas comparaciones.

- Por un lado se realizaron comparaciones para la vecindad A, en la cual en cada instancia se intercambian únicamente pokeparadas.
- Por otro lado, se realizaron comparaciones para la vecindad B, en la cual se intercambian gimnasios (siempre y cuando la solución siga siendo válida).

En ambos casos, se usaron tres tipos de instancias diferentes donde se fijan 4, 5 y 6 gimnasios respectivamente y se aumenta la cantidad de pokeparadas. Por ejemplo, para el primer tipo de instancia se fijan 4 gimnasios y se van agregando pokeparadas.

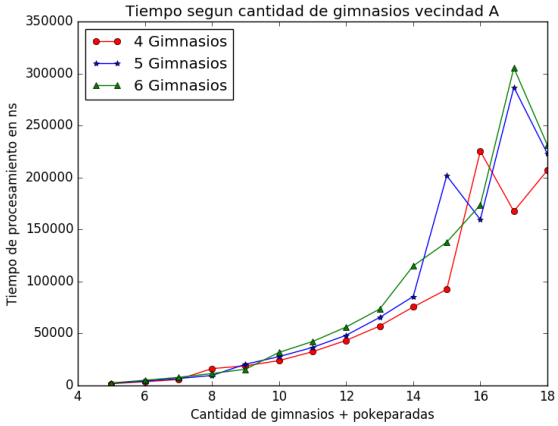


Figura 13: Vecindario A

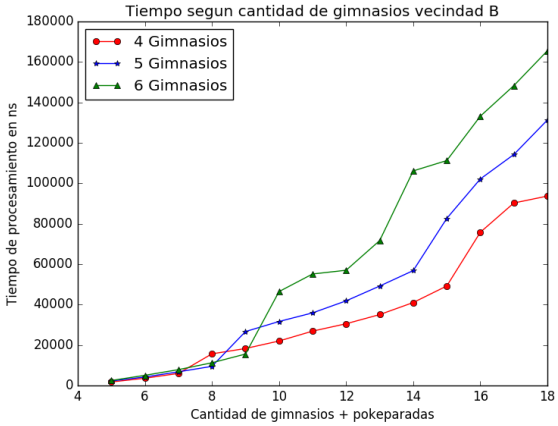


Figura 14: Vecindario B

Para estas instancias, se puede observar que en ambos casos, a medida que crece la cantidad de estaciones, aumenta el tiempo de procesamiento de manera muy similar.

Se puede suponer que para la vecindad A el tiempo de procesamiento debería crecer más rápido que para la vecindad B, debido a que para la vecindad A se pueden intercambiar cualquier par de pokeparadas, mientras que en el vecindario B sólo se pueden intercambiar algunos gimnasios entre sí (además de que para estos experimentos, la cantidad de pokeparadas tiende a ser mucho mayor que la cantidad de gimnasios)

Sin embargo, el tiempo de procesamiento es similar ya que para el vecindario B, para cada intercambio de gimnasios se debe chequear que la nueva solución generada sea válida. Y esto implica recorrer cada estación del recorrido, chequeando que para cada gimnasio por el que se pasa, se pase con la cantidad de pokebolas necesarias.

4.4.4. Experimento 2

En el próximo experimento, comparamos la calidad de solución y el tiempo de corrida de los vecindarios entre sí, y por otro de los vecindarios y el algoritmo greedy, y finalmente los tres juntos. Para esto, se generaron 50 instancias rándom, para cada instancia se corrió el algoritmo 20 veces. Luego se obtuvo un promedio de las 20 corridas hechas para cada instancia (promedio del tiempo de ejecución y de la calidad de la solución).

Para el siguiente experimento queremos probar que realmente, hacer una búsqueda local mejora la solución obtenida del algoritmo heurístico. Esta hipótesis es tomada debido a que, búsqueda local se basa en la solución devuelta por greedy y busca, a partir de ella, encontrar una solución mejor. Por otro lado, queremos ver como se comportan los vecindarios para cualquier instancia, para ver si realmente, dado que el vecindario A tiene más posibilidades de intercambias pokeparadas, esto produce mejores soluciones que el vecindario B, el cuál intercambia gimnasios y tiene menos posibilidades de soluciones válidas.

4.4.5. Experimento 2: Calidad de resultados

Para experimentar acerca de la calidad de los resultados, se generaron distintas comparaciones.

- Por un lado, se compararon los resultados obtenidos ejecutando una búsqueda local haciendo uso del vecindario A, con los obtenidos del vecindario B. Dado que el vecindario A da lugar a muchos intercambios posibles, ya que intercambiar dos pokeparadas cualesquiera mantiene una solución válida, mientras que el vecindario B debe descartar varios intercambios de gimnasios ya que algunos pueden dejar un recorrido inválido, nuestra hipótesis fue que la búsqueda local sobre el vecindario A iba a retornar mejores soluciones que la búsqueda local sobre el vecindario B.

- Por otro lado, se compararon los resultados obtenidos ejecutando una búsqueda local haciendo uso del vecindario A, con los obtenidos a partir del algoritmo greedy (implementado en el Ejercicio 2). El objetivo fue poder visualizar que dado una solución obtenida por el algoritmo greedy, ejecutar

sobre ésta una búsqueda local, la mejora. Elegimos el vecindario A para la búsqueda local ya que es la que más soluciones posibles retorna, debido a la cantidad de intercambios de pokeparadas que se pueden hacer, sobre la cantidad de intercambios de gimnasios que retornan soluciones válidas.

En ambas comparaciones, se corrió 20 veces el algoritmo sobre cada instancia de 50 obtenidas de forma aleatoria. Luego se sacó un promedio de las 20 distancias devueltas para cada instancia, obteniendo 50 resultados. Y por último, se realizó una comparación de cada resultado en función de la cantidad de estaciones (n+m).

A continuación en los siguientes gráficos se pueden observar la calidad de resultados obtenidos en ambas comparaciones.

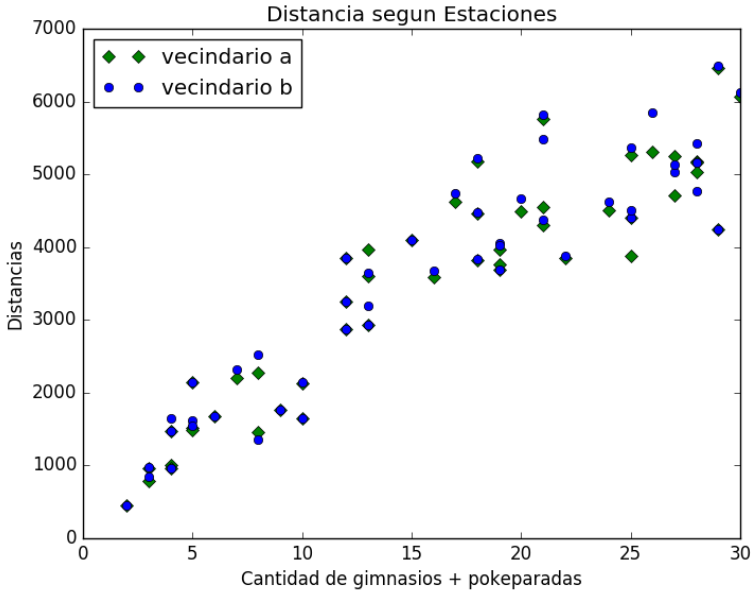


Figura 15: Vecindario A vs Vecindario B

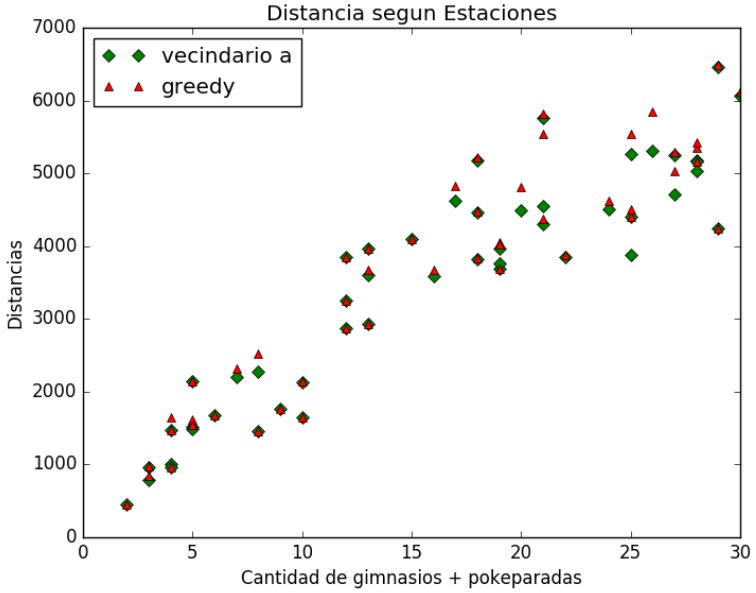


Figura 16: Greedy vs Vecindario A

Para el gráfico que compara ambos vecindarios se puede ver que, como suponíamos, las soluciones obtenidas a partir del vecindario A tienden a ser mejores que las soluciones obtenidas a partir del vecindario B.

Y de igual manera, para el gráfico que compara la solución obtenida a partir del algoritmo greedy con aplicarle a ésta la búsqueda local usando el vecindario A, se puede ver que, como suponíamos, la búsqueda local mejora las soluciones obtenidas a partir del algoritmo greedy.

4.4.6. Experimento 2: Performance

Para experimentar acerca de la performance de los resultados de ambas vecindades y del algoritmo greedy sin búsqueda local, se corrió 20 veces cada algoritmo sobre cada instancia de 50 obtenidas de forma aleatoria. Luego se sacó un promedio del tiempo de ejecución (en ns) de las 20 corridas para cada instancia, obteniendo así 50 resultados. Y por último, se realizó una comparación de cada resultado en función de la cantidad de estaciones (n+m).

Nuestra hipótesis fue que en orden de menor a mayor tiempo de procesamiento estaba el algoritmo greedy, el algoritmo con búsqueda local con el Vecindario B, y por último el algoritmo con búsqueda local con el vecindario A.

El algoritmo greedy tenía que ser el más rapido simplemente porque ambas búsquedas locales comienzan ejecutando el algoritmo greedy y luego realizan los intercambios válidos. Al realizar estos intercambios, la diferencia del tiempo de procesamiento aumenta en comparación con el algoritmo greedy.

Y luego, al comparar el tiempo de procesamiento entre ambas vecindades, suponemos que la vecindad A tardará más que la vecindad B, debido a que para la vecindad A se pueden intercambiar cualquier par de pokeparadas, mientras que en el vecindario B sólo se pueden intercambiar algunos gimnasios entre sí (además de que para estos experimentos, la cantidad de pokeparadas tiende a ser mucho mayor que la cantidad de gimnasios). Mas allá de que para la búsqueda local con el vecindario B (a diferencia del vecindario A), se deba verificar que para cada intercambio, la solución siga siendo válida (y sino, descartarla), el tiempo sigue siendo mayor debido a que la cantidad de intercambios de pokeparadas es mucho mayor a la cantidad de intercambios de gimnasios que se pueden realizar.

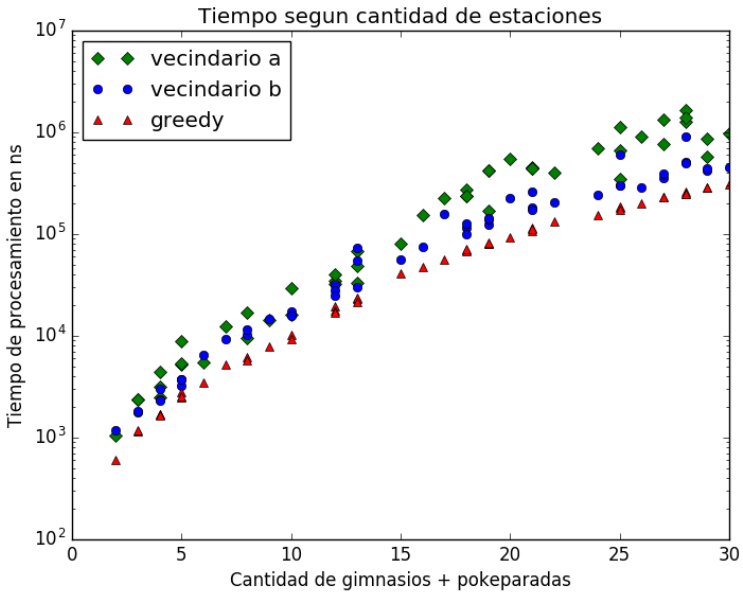


Figura 17: Greedy vs Vecindario A

En este gráfico se puede observar que el tiempo de procesamiento para cada algoritmo coincide con la hipótesis. La que menos tiempo de procesamiento tiene es la greedy. Luego la búsqueda local con el vecindario B y luego la búsqueda local con el vecindario A.

5. Ejercicio 4: Metaheurística

5.1. Descripción del problema y solución propuesta

En este punto, se pidió realizar un algoritmo basado en una metaheurística que resuelva el problema en cuestión. El mismo al tratarse de una metaheurística puede no dar la solución óptima, y puede no devolver una solución aunque la misma exista.

Se decidió utilizar como metaheurística para resolver esta variante del TSP, GRASP. De esta forma se explora el espacio de soluciones, partiendo de diferentes instancias y tratando de mejorarlas mediante búsqueda local. La idea de GRASP es que cada búsqueda va a converger en caso de ser posible a un mínimo local. Como se comienza de distintas instancias se van a explorar varios mínimos locales distintos, con lo cual la solución se puede obtener tomando la mejor de estas buenas soluciones. Como la eficacia de GRASP depende de empezar de soluciones lo suficientemente diferentes utiliza una heurística greedy a la que le suma aleatoridad, en la cual en vez de seleccionar siempre al mejor candidato para una solución, selecciona uno al azar de un grupo de candidatos (el RCL), cuyo tamaño o criterio de selección es configurable. La otra parte configurable de GRASP es la determinación de cuánto se va a buscar, es decir, cuanto tiempo se va a invertir en generar soluciones greedy aleatoria y tratar de mejorarlas. Para este caso particular de este problema, la implementación de GRASP elegida consiste en lo siguiente: se obtiene una solución (en caso de ser posible) mediante una heurística greedy. La misma es similar a la desarrollada en el ejercicio 2. La diferencia es que se le agrega aleatoridad, ya que no se toma a la estación más cercana sino que se eligen k estaciones más cercanas (siendo k un parámetro configurable) y se elige una al azar de esas estaciones. La misma estación elegida tiene que ser factible de ser visitada. En caso contrario, el algoritmo devuelve que no encontró solución. Si se toma como parámetro $k = 1$, este algoritmo greedy es exactamente igual al desarrollado en el ejercicio 2 ya que en todas las iteraciones va a elegir al más cercano. Si se toma como k la cantidad de estaciones total, el algoritmo se transforma en uno completamente aleatorio ya que en cada iteración se toma cualquiera estación. Luego de obtenerse una solución a la misma se le aplica la búsqueda local desarrollada en el ejercicio 3 para tratar de mejorarla. Este procedimiento se repite varias veces hasta que se cumple la condición de parada (parámetro configurable). Se usaron dos condiciones de parada distintas: una es determinar una cantidad de repeticiones que se desea realizar el procedimiento (condición sobre la cantidad de repeticiones del ciclo) y la otra es una cantidad de repeticiones en la cual no se mejoró una solución (condición sobre la cantidad de repeticiones sin mejorar). En ambos casos siempre el algoritmo termina ya que en el primer caso se fija la cantidad de repeticiones (por lo tanto el ciclo corre un número finito de veces) y en el otro aunque se elija un número muy grande, siempre a partir de un momento la solución no se va a poder mejorar (cuando se encuentra la óptima), por lo que en el peor caso si la óptima es encontrada rápidamente por el algoritmo, se va ejecutar una vez encontrada, la cantidad de iteraciones fijada, ya que no se va a poder encontrar ninguna mejor. En cada paso, se va guardando la solución mejor que se encuentra. Es decir, se mantiene una solución, que se actualiza en caso de que se encuentre una mejor en cada iteración de este procedimiento. Al finalizar se retorna la mejor solución encontrada.

El algoritmo propuesto es el siguiente:

```
enum crit = { A, B }

solverEj4(vector<Estacion> estaciones, vector<vector<double> > distancias, n, m,
          k, grasp, crit criterioTipo, int criterioCant, bool vecindarioBusqLocal) {
    solucion s1
    solucion s2
    solucion best
    i = 0
    bool mejor
    double nuevaSol

    mientras (i < criterioCant)
        s1 = greedyRandomized(estaciones, distancias, n, m, k, grasp)
        s2 = solucionEj3(s1, estaciones, distancias, k, vecindarioBusqLocal)

        mejor = False
        nuevaSol = s2.first
        si i == 0 || (nuevaSol > -1 && nuevaSol < best.first)
            best = s2
            si i > 0
                mejor = True
            fin si
        fin si


        si criterioTipo == A
            i++
        fin si

        si criterioTipo == B && mejor == False
            i++
        fin si

    fin mientras

    devolver best
```

El algoritmo consiste en un ciclo, el cual se ejecuta mientras no se haya llegado a la condición de terminación. La condición de terminación está dada por una tupla (denominada criterio) que contiene en su primer coordenada un entero, que es el que se usa en la condición del ciclo para saber si hay que cortar o no. La segunda coordenada es un booleano y sirve para identificar cual de los dos criterios que definimos se está utilizando. El booleano si es verdadero indica que se usa el criterio de parar cuando no se pudo mejorar en k veces la solución siendo k el entero de la primera coordenada. Si es falso, significa que se usa el criterio de correr directamente k veces el ciclo, siendo k también el entero de la primer coordenada. En el cuerpo del ciclo lo que se realiza es obtener una solución llamando a la función *greedyRandomized*, la cual realiza lo mismo que la implementada en el ejercicio 2, salvo que en vez de seleccionar a la estación más cercana a la que se puede ir en cada paso, selecciona una al azar de n estaciones cercanas, siendo n el entero *grasp* que se le pasa a la función. Luego de eso se aplica búsqueda local, llamando a la función *solucionEj3*, la cual es exactamente la misma que la desarrollada en el ejercicio 3. Si nos encontramos en la primera iteración del ciclo o la solución obtenida es mejor que la anterior que teniamos, entonces se actualiza la nueva mejor solución obtenida con la encontrada en esta iteración. Finalmente se pregunta si se está usando el criterio de ejecutar k veces fijo (con lo cual la variable *criterio.second* sería falsa) y en caso de ser así se suma uno al contador del ciclo. Sino, se trata del otro criterio y sólo se actualiza el contador si no se encontró una mejor solución en la iteración (hecho representado por el valor de la variable booleana *mejor*), sumándole 1. Una vez terminado el ciclo se devuelve la mejor solución obtenida.

5.2. Experimentación

Para poder comparar cada configuración posible y obtener de ahí la configuración óptima, se generaron 30 instancias random, las cuales nos aseguramos que tengan solución. Luego se realizaron dos experimentos. En el primero, se iteró sobre la cantidad de vértices que entran en la RCL. Mientras que en el segundo, se iteró sobre el límite de cada criterio de parada. Para ambos al igual que en el **Ejercicio 3**, se buscó comparar por un lado, qué configuración devolvía mejores distancias, y por otro qué configuración corría en menos tiempo. En cada gráfico se usaron las siguientes referencias, P = Criterio de paradas, i = Iteraciones del criterio, V = vecindario de búsqueda local, RCL = K vértices que accederan a la RCL.

5.2.1. Experimento 1: RCL

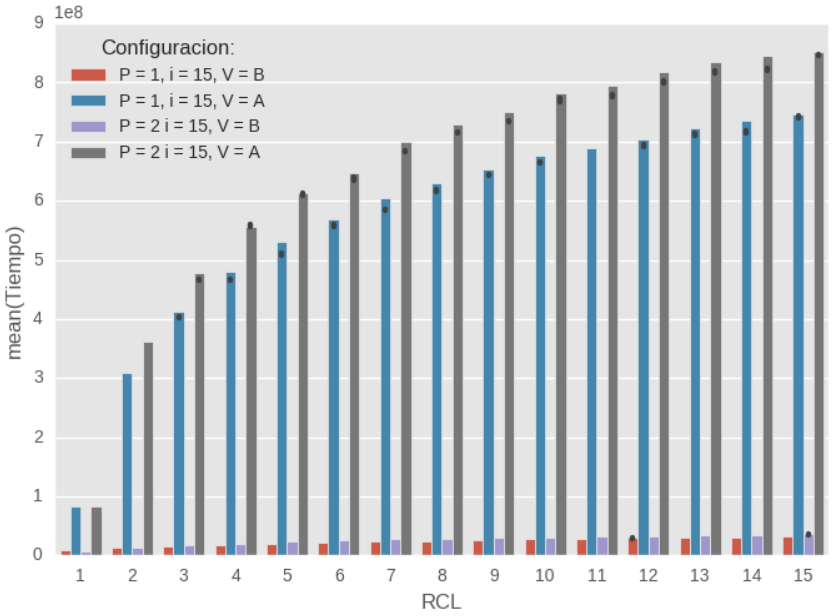
En el siguiente experimento, se crearon 30 instancias random. Las mismas tenían entre 1 y 15 gimnasios. Los mismos requerían entre 0 y 20 pociones. Las posiciones de las estaciones también fueron random. Además, estas 30 instancias se copiaron 4 veces, cada una con una configuración distinta. En cada una de las 4 configuraciones se modificó por un lado el vecindario utilizado en la Búsqueda Local y por otro, el criterio de parada en el cual la cantidad de iteraciones del criterio se fijó en 15.

De esta manera las siguientes 4 configuraciones son

Instancias	Busqueda Local	Criterio de parada
Configuración 0	Vecindario A	Iterar K veces
Configuración 1	Vecindario A	Iterar hasta K peores
Configuración 2	Vecindario B	Iterar K veces
Configuración 3	Vecindario B	Iterar hasta K peores

Cuadro 1: Configuraciones experimento 1

Luego por cada instancia, con cada configuración, se iteró el K de RCL, de 1 a 15. Por cada K, se corrió 20 veces el mismo algoritmo, y se sacó un promedio del tiempo y de la distancia. Además, se sacó un promedio de la distancia y el tiempo entre las 30 instancias con la misma configuración y el mismo K. De esta manera se contaba con un solo promedio de tiempo y distancia por cada K, y por cada configuracion.



Luego, a partir de esta experimentación, dado que a medida que crecía la lista RCL, crecía la cantidad

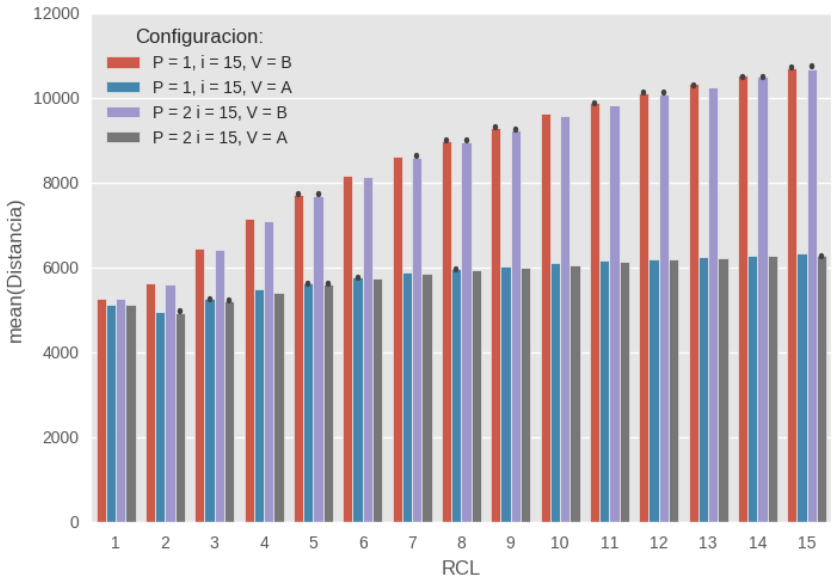
de elementos a randomizar, pudimos observar cómo esto impactaba directamente en el tiempo de ejecución. En el gráfico esto se observa por el crecimiento del tiempo con respecto al crecimiento del tamaño de la RCL.

Por otro lado, pudimos notar que el factor determinante para el tiempo de corrida, fue el vecindario utilizado.

Esto se debe a que, como fue mencionado en el **Ejercicio 3**, intercambiar pokeparadas genera más soluciones que intercambiar gimnasios en los recorridos, pues encontrar un recorrido válido intercambiando gimnasios con distintas pociones es muy poco probable, mientras que intercambiar pokeparadas siempre resulta un recorrido válido. Por lo tanto, la búsqueda local de gimnasios, al no encontrar recorridos válidos, termina. Mientras que la búsqueda local de pokeparadas, dado que no puede suceder que caiga en un recorrido inválido, siempre puede llegar a obtener una posible mejor solución o mínimo local.

Finalmente, otro aspecto que impactó directamente en el tiempo de ejecución, fue el criterio de parada. Si bien la diferencia de tiempo es mínima, dentro del vecindario A se puede observar que en todo momento, el criterio de parada 1 con 15 iteraciones, era aquel que menos tardaba. Concluyendo de esta manera, que en cuanto a tiempos, la configuración más optima, se da con el vecindario B y con el criterio de parada 1, además de un RCL de un elemento. La comparación en cuanto a criterios de parada, se da en la sección de **Experimento 2**.

Un segundo aspecto a comparar es la calidad de solución,



Para este entonces, volvemos a notar como el vecindario, es la sección fundamental de la metaheurística, dado que, comparando a partir de los vecindarios, el vecindario A es aquel que mejor distancia consigue, para todo criterio y para todo valor de RCL. La justificación de por qué las soluciones del vecindario A son mejores que las del vecindario B, está explicado en el **Ejercicio 3**.

Para sacar una mejor conclusión, comparamos el vecindario A y el vecindario B independientemente. Por un lado, usando el vecindario A notamos que, para un RCL de tamaño 2 y para cualquier criterio, se consigue la mejor distancia. Esto podría deberse a que, a mayor RCL, las posibilidades de obtener una próxima estación con una distancia más corta, es menor, debido al factor random que obtiene una próxima estación de la RCL. Mientras que para un RCL de tamaño 1, el algoritmo metaheurístico pasaría a ser similar a un algoritmo greedy, junto con una búsqueda local, con lo cual recorrería menos soluciones.

Por otro lado, dentro del vecindario A, no encontramos una mejora lo suficientemente notable como para decidir acerca de cuál criterio de parada es óptimo.

También pudimos ver que con el vecindario B la calidad de solución empeora a medida que crece el RCL. Esto se debe a que cuanto más tamaño tenga RCL, en cada iteración donde se construye un recorrido, se elige una estación al azar entre las que entren dentro de la RCL. Cuanto más grande sea la RCL, más estaciones posibles serán las candidatas a ser la próxima a recorrer, y por lo tanto, mayor

probabilidad habrá de que no se elija la óptima. También influye que en la búsqueda local, el vecindario B tiene una peor calidad de solución.

También cabe destacar que, al igual que con el vecindario A, no notamos ninguna mejora a partir del criterio de parada.

Luego de este experimento, concluímos que el algoritmo que utiliza el vecindario A, es quien devuelve una mejor solución, sin importar el criterio de parada. Y que al comparar en base al tamaño de la RCL, cuánto menor sea, mejor solución se va a obtener.

Finalmente, obtuvimos como conclusión de este experimento que, en principio, si se depende de la RCL, no se puede obtener un algoritmo eficiente en cuanto a tiempo y calidad de la solución a la vez. Por otro lado, pudimos comprender la fuerte influencia que tienen los vecindarios utilizados en la búsqueda local dentro del algoritmo, tanto para tiempos como para calidad de solución. Es decir, el vecindario A devolverá una mejor solución, en mayor tiempo, y el vecindario B, devolverá una peor solución, en menor tiempo.

El tamaño de la RCL fue el segundo factor de crecimiento, tanto para distancia como para tiempos. En ambas comparaciones (exceptuando que en un RCL de tamaño 2, la solución mejoró), tanto el tiempo como la calidad de solución, empeoraban a medida que crecía. Luego, se buscará en próximos experimentos ver cuánto influye el criterio de parada y sus iteraciones.

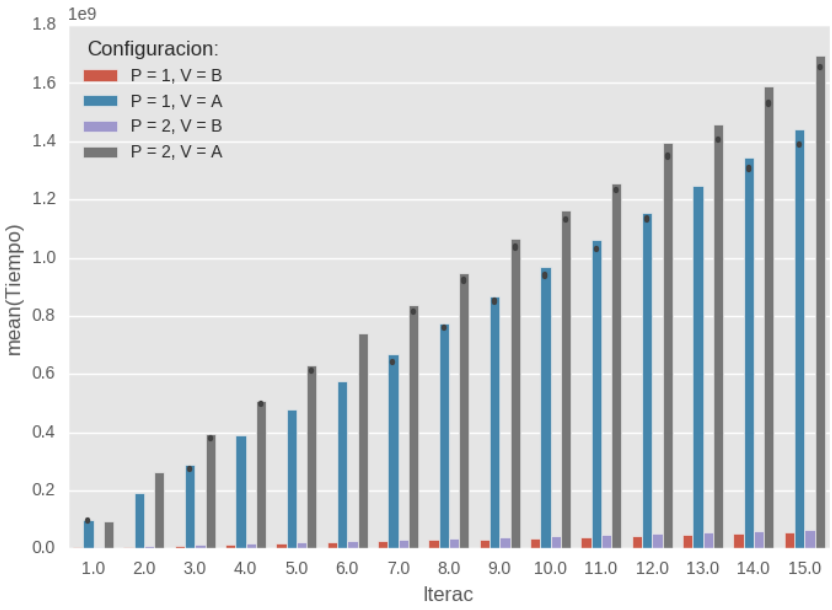
5.2.2. Experimento 2: Iterar el límite de los criterios de parada

Para este experimento se usaron las mismas instancias que en el **Experimento 1**, pero esta vez en vez de fijar el criterio de parada e iterar el tamaño del RCL, se fijó el tamaño del RCL en la mitad de la cantidad de estaciones y se iteró de 1 a 15 el límite de ambos criterios de parada. De esta manera las 4 configuraciones son,

Instancias	Busqueda Local	Criterio de parada	RCL
Configuración 0	Vecindario A	Iterar K veces	Estaciones/2
Configuración 1	Vecindario A	Iterar hasta K peores	Estaciones/2
Configuración 2	Vecindario B	Iterar K veces	Estaciones/2
Configuración 3	Vecindario B	Iterar hasta K peores	Estaciones/2

Cuadro 2: Configuraciones experimento 2

Luego, se corrió cada instancia 20 veces, y se calculó el promedio del tiempo y las distancias. Después por cada instancia cuyo vecindario de búsqueda local, criterio de parada y cantidad de iteraciones eran el mismo, se sacó el mismo promedio. Luego, por cada una de las 4 configuraciones, se obtuvieron 15 promedios de tiempo y distancia.

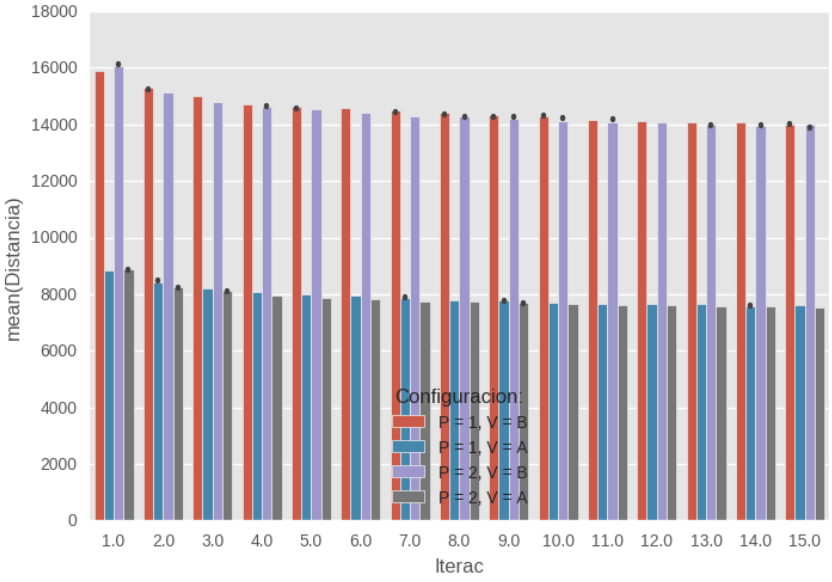


Al igual que en el **Experimento 1**, la característica determinante de las instancias, que influye en el tiempo, es el vecindario utilizado en la búsqueda local. Luego, comparando cada vecindario en particular a medida que avanza la cantidad de iteraciones, en ambos criterios crece el tiempo de ejecución. Esto se debe claramente a que, a mayor cantidad de veces que se ejecuta un algoritmo, el tiempo de corrida crece.

Luego, si se observa el vecindario A, hay una gran diferencia de tiempos entre cada criterio de parada del algoritmo. Para entender esto, se debe especificar en detalle cada criterio de parada. Es decir, esto se debe a que el **criterio 1** hace exactamente la cantidad de iteraciones que se le indique, mientras que el **criterio 2**, sea K la cantidad de iteraciones, se ejecuta hasta encontrar K soluciones peores. O sea, siendo i la cantidad de mejores soluciones que se encuentran, se harán K + i iteraciones, lo cual aumenta el tiempo de ejecución. Lo mismo se puede observar en el vecindario B. Aunque es mínima la diferencia de tiempos entre criterios, es el segundo factor que genera que tarde más, por el mismo motivo que en el vecindario A.

Como primer conclusión, si nos basamos en vecindarios de la búsqueda local, el vecindario que menos tiempo toma es el B. Y a la hora de elegir un criterio de parada, si se busca un menor tiempo, se debe hacer una sola iteración para cualquier criterio. Pero si es necesario iterar en alguno, se tomaría el **criterio 1**, donde la cantidad de iteraciones que se indique son las que se van a realizar.

Luego para analizar la calidad de solución,



En primer lugar, el objetivo del criterio de parada es mejorar la solución. Cuantas más iteraciones se hace del algoritmo, mejor soluciones se encuentra. Es por esto que sin importar el vecindario ni el criterio, a mayor cantidad de iteraciones, mejores soluciones se encontrarán.

Luego, el vecindario fue el factor que más influyó en el resultado, es decir, las soluciones del vecindario A son claramente mejores que las soluciones del vecindario B. Además, por cada vecindario, se observa que el criterio 1 está por debajo, en cuanto a calidad de solución, que el criterio 2. Esto se debe, como mencionamos anteriormente, a que si se itera K veces el **criterio 2**, cada vez que encuentra una mejor solución no se cuenta como iteración. Por lo tanto, sea i la cantidad de veces que se encontró una mejor solución, la cantidad de iteraciones será de K+i. Al haber mayor cantidad de iteraciones, habrá mas probabilidad de mejorar la solución. Mientras que el **criterio 1** únicamente iterará K veces el algoritmo, sin importar la mejora de la solución.

De esta manera, concluímos que el criterio 2 es aquel que, a mayor cantidad de iteraciones, mejores soluciones encuentra (siempre y cuando esté complementado con el vecindario A de la búsqueda local).

Finalmente, como conclusión del segundo experimento, los resultados fueron similares al experimento 1. Debido a que si se busca una mejor solución, es necesario mayor tiempo de ejecución. Luego, si se busca una mejor solución, se debe usar el criterio de parada 2 iterando la mayor cantidad de veces posible, junto con el vecindario A. Mientras que si se busca una solución poco óptima en el menor tiempo posible, es necesario pocas iteraciones del criterio de parada 1, junto con el vecindario B.

5.2.3. Conclusión

Luego de comparar cada configuración posible de la metaheurística, pudimos concluir que, uno de los principales problemas que se enfrentan para este tipo de algoritmo, es el criterio entre calidad de solución y tiempo de corrida. En ambos experimentos encontramos la misma problemática. Luego, si se busca optimizar más la solución, obtuvimos que es necesario elegir el vecindario A, junto con un RCL de tamaño 2, con el criterio de parada 2 haciendo la mayor cantidad de iteraciones posibles. Mientras que si se busca es un algoritmo metaheurístico rápido, es necesario el vecindario B, junto con un RCL de tamaño 1, con pocas iteraciones del criterio de parada 1.

Dado que una metaheurística tiene un factor aleatorio que influye en la calidad del resultado, si el algoritmo se ejecuta pocas veces, es difícil analizar la calidad del resultado. Sin embargo, al ejecutar el algoritmo una cantidad considerada de veces, es más probable que la calidad del resultado mejore. Pero como se sabe, cuantas más veces se ejecute el algoritmo, más alto va a ser el tiempo de ejecución.

Por lo tanto, creemos que antes de decidir la configuración de un algoritmo con una metaheurística, se debe analizar, y poner en una balanza, qué es más relevante: si obtener una mejor calidad de soluciones,

o si obtener una solución lo antes posible.

Vale también destacar que creemos que la configuración de cuánto va a influir el factor aleatorio del algoritmo, se debe ir modificando y decidiendo en base a la calidad de resultados que se va obteniendo. Por ejemplo, si el tamaño de la RCL es de un único elemento (es decir, el algoritmo se asemeja al del Ejercicio 2) y la solución obtenida es mala comparada con la mejor solución que se podría obtener, se tiende a tomar la decisión de aumentar el tamaño de la RCL con la esperanza de obtener mejores soluciones.

6. Ejercicio 5: Comparación

6.1. Hipótesis

Para realizar un análisis exhaustivo y poder realizar conclusiones generales sobre las distintas implementaciones de nuestro problema, se pidió por último hacer una experimentacion acorde para poder compararlas, en donde se usaron las mejores configuraciones halladas (particularmente para los ejercicios 3 y 4) en cuanto a calidad de la solución.

Se realizaron dos experimentos. El primero consistió en un análisis y comparación de la calidad de los resultados obtenidos y del tiempo de procesamiento que demandaron todas las implementaciones realizadas. El segundo experimento consistió en el mismo análisis y comparación pero sólo de las implementaciones no exactas (ejercicios 2, 3 y 4). Para ambos experimentos se utilizó una configuración considerada favorable en cuanto a calidad de solución para las heurísticas (Vecindario A, RCL = 2, criterio de GRASP de 30 iteraciones en las cuales no se haya encontrado ninguna solución mejor). Las instancias fueron generadas de manera aleatoria sin repetir cantidad de estaciones totales (gimnasios + pokeparadas), y se intentó acercarse a que haya al menos una posible solución colocando en cada instancia la cantidad de pokeparadas igual a $\sum_{g \in Gimnasios} pociones(g)$, con una cantidad random entre 1 y 10 pociones por gimnasio para el primer experimento y entre 1 y 50 para el segundo, y un tamaño de mochila igual a 3 veces la cantidad de pokeparadas de manera que alcance para guardar todas las pociones de todas las pokeparadas. Las ubicaciones de las pokeparadas y los gimnasios también fueron seleccionadas de manera random con cada par (x, y) x e y entre 1 y 1000. Para el primer experimento se utilizaron 15 instancias aleatorias, en la cual la cantidad de gimnasios se varió entre 1 y 7 y la suma de gimnasios más pokeparadas sea menor o igual a 16, mientras que para el segundo experimento se realizaron 99 instancias donde la cantidad de gimnasios varió entre 1 y 50 y la suma de gimnasios más pokeparadas sea menor o igual a 100.

Teniendo en cuenta la experimentación realizada a lo largo de este trabajo, esperábamos que la metaheruística tenga mejores resultados que la heurística greedy y la búsqueda local, pero con una significativa diferencia temporal entre ellas ya que GRASP realiza reiteradas veces una heurística greedy junto a una búsqueda local.

6.2. Experimento 1

En este experimento se utilizaron instancias de hasta 16 estaciones para que los tiempos y soluciones puedan ser comparados con el algoritmo de Backtracking. En una comparación entre los 4 algoritmos incluyendo las 16 instancias, construimos la siguiente tabla con estadísticas al respecto. En la misma, el promedio realizado es sobre los resultados de las 16 instancias evaluadas 30 veces cada una.

	Tiempo en ns	Gyms	Pokeparadas	Mochila	Distancia	Estaciones visitadas	Nro. Ej
promedio	6,104684e+09	3,25	6,25	16,4375	1811,2025	9,0625	1
promedio	54172,085595	3,25	6,25	16,4375	2464,306785	9,298539	2
promedio	60037,843750	3,25	6,25	16,4375	2452,986250	9,298539	3
promedio	6,175643e+07	3,25	6,25	16,4375	2202,168813	9,091667	4
min	326	1	1	0	0	1	1
min	284	1	1	0	0	1	2
min	978	1	1	0	0	1	3
min	7,925460e+06	1	1	0	0	1	4
max	6,290560e+10	6	11	31	3041,12	16	1
max	314883	6	11	31	3922,82	17	2
max	531943	6	11	31	3922,82	17	3
max	1,256230e+08	6	11	31	4135,69	17	4

En cada fila se calculó el promedio, mínimo y máximo de las variables estudiadas de manera independiente para cada ejercicio. El número perteneciente a la columna de *Estaciones visitadas* corresponde al valor de estaciones visitadas de la solución obtenida.

Observamos que en cuanto a los promedios el más cercano al ejercicio 1 es, como esperábamos, el ejercicio 4; particularmente en el número de estaciones visitadas, la distancia recorrida y el tiempo. Pero además esto nos muestra que a pesar de dar una solución cercana, el tiempo que toma es mucho mayor al de los otros dos ejercicios, lo que afirma la hipótesis propuesta.

En cuanto a los mínimos obtenidos cabe destacar que es normal que el ejercicio 4 tome más tiempo que los demás, puesto que de por sí tiene la configuración de realizar 30 iteraciones que no den resultados mejores.

Por último, el máximo del ejercicio 4 fue uno de los peores en cuanto a calidad de la solución y entendemos que esto se debe a la aleatoriedad que tiene en su implementación. Los máximos tiempos que tomaron los ejercicios 2 y 3 son evidentemente los menores, lo cual era lo esperado. El ejercicio 4 si bien toma mucho más tiempo que los anteriores dos, sigue siendo aproximadamente 100 veces más rápido que el ejercicio 1.

Para mostrar cómo evoluciona el tiempo en función de la cantidad de estaciones de cada algoritmo, plasmamos los resultados obtenidos en este experimento en este gráfico:

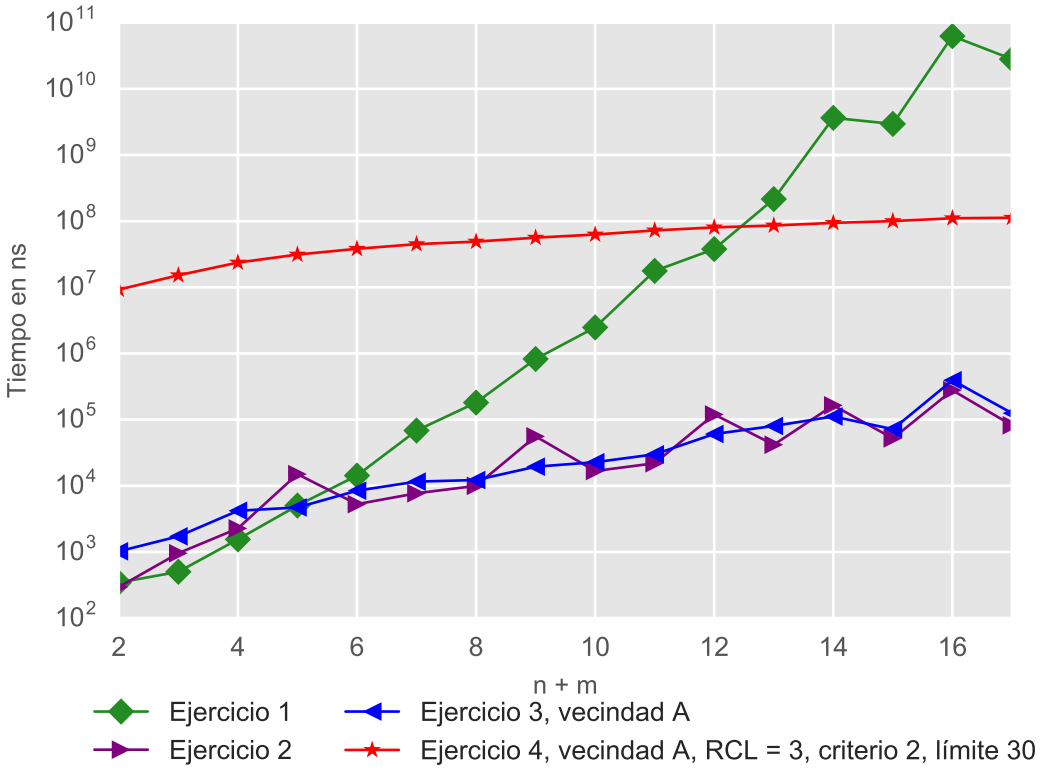


Figura 18

Observamos que como mostraba la tabla, los ejercicios 2 y 3 se mantienen cercanos en cuanto a tiempo. Por otro lado, el ejercicio 1 crece exponencialmente y el ejercicio 4 se mantiene entre 10^7 y 10^8. Notamos cómo a partir de las 12 estaciones, la implementación de GRASP toma menos tiempo que la implementación de backtracking por lo que si GRASP efectivamente diera una buena solución y estuvieramos dispuestos a renunciar a efectividad sobre tiempo, entonces sería una buena opción para instancias grandes.

Para analizar la calidad de la solución con un poco más de detalle generamos el siguiente gráfico de barras que muestra la distancia recorrida en función del tamaño de la instancia para cada implementación.

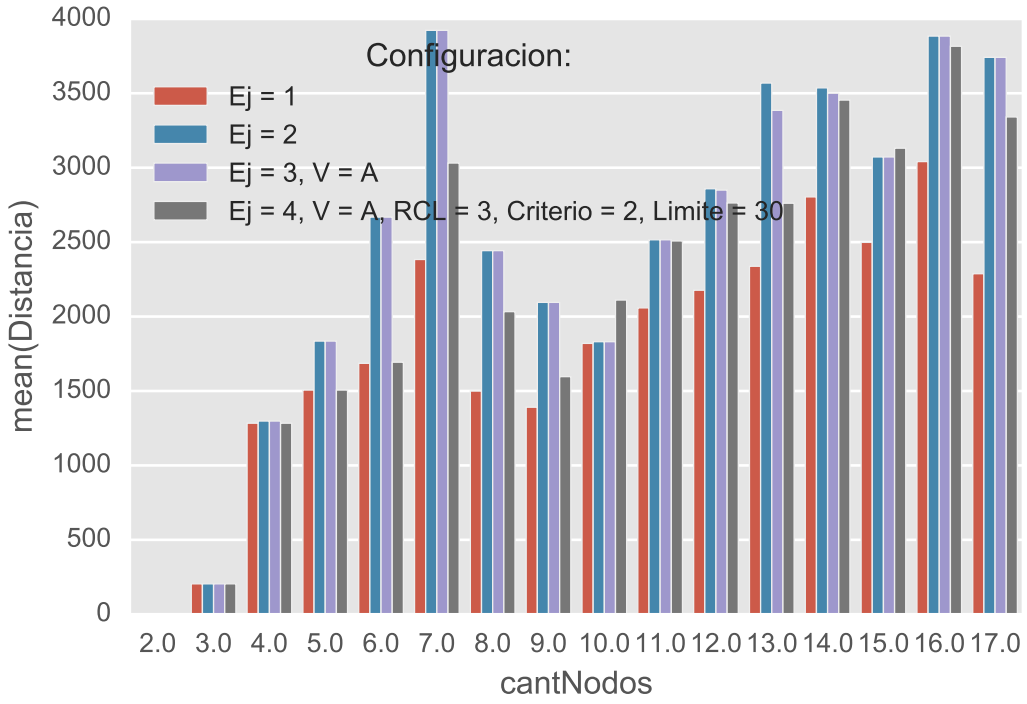


Figura 19

Como era de esperar, en general la implementación GRASP da mejores soluciones que las implementaciones golosa y de búsqueda local. En algunos casos tales como las instancias de 10 y 15 estaciones GRASP da una solución peor que los otros dos algoritmos heurísticos y esto puede deberse nuevamente a la aleatoriedad que contiene, tal como mencionamos anteriormente.

6.3. Experimento 2

Para este segundo experimento se compararon instancias de hasta 100 estaciones para dar una mejor idea del comportamiento de los algoritmos heurísticos comparándolos entre sí.

En lo que respecta a tiempo de ejecución obtuvimos el siguiente gráfico basándonos en los resultados obtenidos:

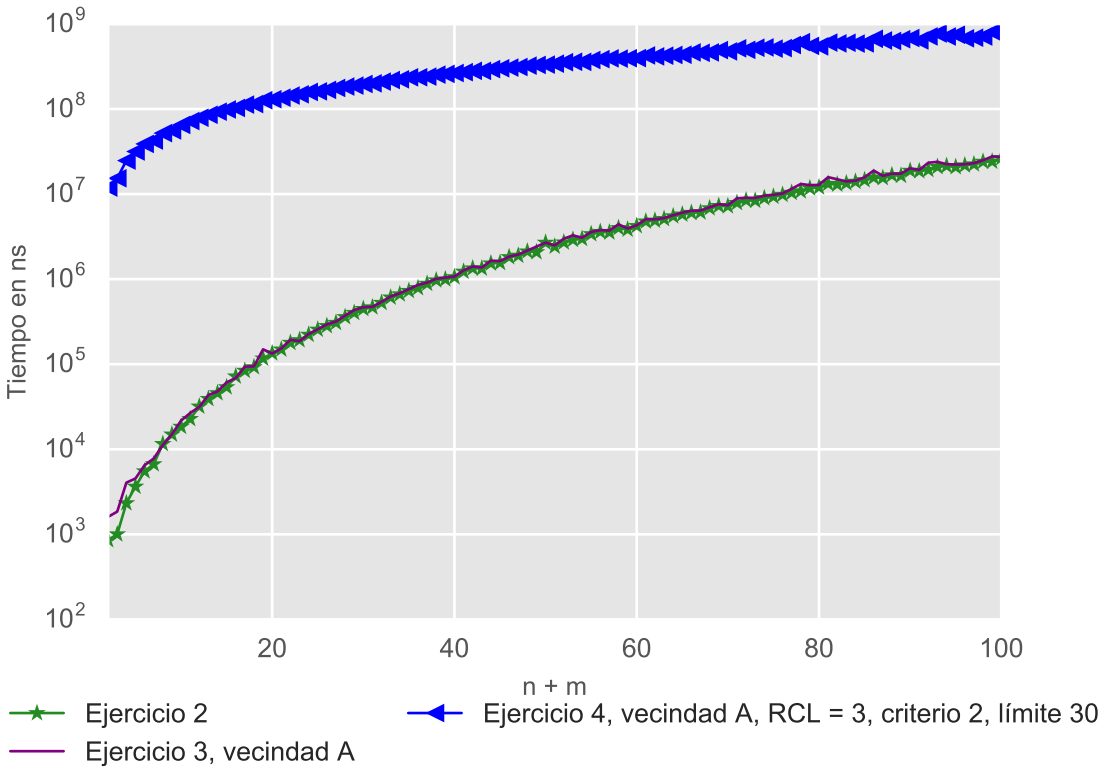


Figura 20

Al igual que en el gráfico del experimento anterior, el ejercicio 2 y 3 se mantienen casi a la par. Como podemos observar, confirmamos basándonos en nuestra hipótesis que ambos se mantienen lejanos (tiempo mucho menor) al ejercicio 4 y se verificará en el análsis de la calidad de la solución si esta diferencia en el tiempo genera una mejora en la misma. Notemos cómo para distintas instancias el ejercicio 4 toma casi un 99 % más que los otros dos ejercicios lo cual hace que este resultado afirme nuestra hipótesis.

Respecto a la calidad de la solución llegamos a construir el siguiente gráfico:

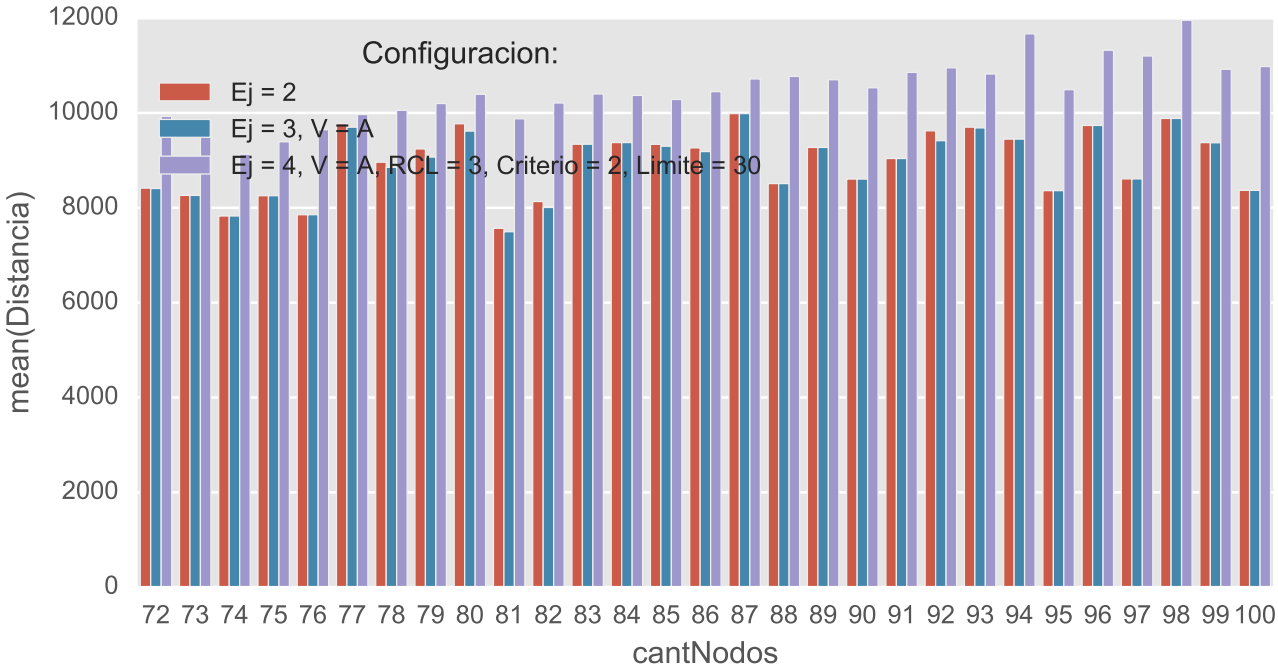


Figura 21

Luego de haber analizado los resultados de este último experimento nos llamó la atención ver que a diferencia del primer experimento, en este el ejercicio 4 se comportó de manera tal que las soluciones obtenidas con el mismo eran de aproximadamente hasta un 25 % peores que de los otros 2 ejercicios. Podemos entender que este margen de error resulta otra vez de la aleatoriedad que tiene implementado GRASP. También observamos que, con éxito en nuestra hipótesis, los resultados del ejercicio 3 se mantuvieron menores o iguales que las soluciones del ejercicio 2.

6.4. Conclusiones

Finalizando este informe, concluimos que si es necesaria una solución exacta para resolver nuestro problema entonces la única opción para asegurarnos la misma es utilizar la implementación de backtracking y esperar el tiempo que sea necesario, el cual no es mucho si son instancias de hasta 13 estaciones. En caso de que sea aceptable un posible error en la solución y no se espere siempre la óptima, es posible utilizar el algoritmo que realiza búsqueda local ya que la misma, como vimos, genera soluciones mejores o iguales que el algoritmo goloso.

Dado que la implementación de GRASP tiene cierta aleatoriedad como vimos para instancias grandes no era la mejor implementación ni en tiempo ni en calidad de la solución por lo que no sería recomendable utilizar esta implementación para resolver este ejercicio salvo que se permita utilizar un RCL más grande y una cantidad de iteraciones mucho mayor, pero esto generaría un crecimiento en el tiempo de ejecución y tal vez termine siendo mejor utilizar backtracking ya que este no tiene configuraciones. Esto último se cumple particularmente para instancias chicas (no más de 13 estaciones) ya que para ese tipo de instancias backtracking supera en tiempo (es menor) al algoritmo de GRASP.

7. Informe de modificaciones

A continuación se señalan los cambios realizados para la reentrega del trabajo:

- Ejercicio 1
  - Se realizaron correcciones de redacción en la sección de detalles implementativos.
  - Se cambiaron las escalas de algunos gráficos a logarítimica
- Ejercicio 2
  - Se realizaron correcciones de redacción en la sección de detalles implementativos.
  - Se realizaron correcciones de redacción en la sección de comportamiento del algoritmo.
  - Se dividió el pseudocódigo del algoritmo en partes, y se corrigió la explicación.
  - Se corrigió en la explicación de la complejidad, el análisis de las operaciones de las listas.
  - Se añadieron más experimentos, con más instancias y corrigiendo su explicación.
- Ejercicio 3
  - Se agregó un experimento el cual se probó con mayor número de instancias
- Ejercicio 3
  - Se experimentó con mayor número de instancias
  - Se modificó el experimento 2.
  - Se redactaron mejor las conclusiones.
- Ejercicio 5
  - Se cambiaron los experimentos por 2 nuevos con instancias nuevas generadas de manera aleatorias
  - Se agregaron conclusiones de la experimentación