DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 3

Camino mínimo a Maestro Pokemon

Algoritmos y Estructuras de Datos III
Segundo Cuatrimestre de 2016

Integrante	LU	Correo electrónico
Badell, Luis	246/13	luisbadell@gmail.com
Borgna, Agustín	079/15	aborgna@dc.uba.ar
Corleto, Alan	790/14	corletoalan@gmail.com
Lancioni, Gian Franco	234/15	glancioni@dc.uba.ar

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

El problema del cual nos ocuparemos en este trabajo se presenta desde el siguiente contexto narrativo: queremos encontrar el tiempo mínimo para recorrer un conjunto de ubicaciones en un mapa (que llamaremos *gimnasios*) pero manteniendo una cantidad no-negativa de ciertos objetos (*pociones*) que disminuyen cada vez que recorremos un gimnasio. La cantidad de pociones presentes estará en todo momento acotada por una constante que consideraremos el *tamaño de mochila*. Las pociones se recargan de a 3 unidades (si la recarga supera el tamaño de la mochila, el valor final será el tope mismo de la mochila) visitando ubicaciones que llamaremos *paradas* y no pueden visitarse cada una más de una vez, sin embargo pueden no visitarse todas.

Recibimos por input n (parámetro) gimnasios de la forma (X_k, Y_k, P_k) con las primeras y segundas coordenadas como su ubicación en el mapa (un plano euclideano) y las terceras como la cantidad de pociones que consumen. También recibimos m (también parámetro) paradas de la forma $(X_{k'}, Y_{k'})$ que convertimos en tuplas $(X_{k'}, Y_{k'}, -3)$. Esto se debe a que vamos a considerar la secuencia de mínima distancia (definida más adelante), correspondiente al camino buscado, $orden = \langle (X_0, Y_0, P_0)\ ..\ (X_{|orden|-1}, Y_{|orden|-1}, P_{|orden|-1}) \rangle$ que contenga solamente algunas paradas y todos los gimnasios (apareciendo una única vez cada uno) del input y definiremos la cantidad de pociones acumuladas al momento de realizar la i-ésima ubicación de la secuencia como:

$$f(i) = \begin{cases} -P_0 & si \quad i = 0 \\ \text{mín}(f(i-1) - P_i, \text{tamaño de mochila}) & si \quad 0 < i < |orden| \end{cases}$$

De modo que para todo i valga $f(i) > 0$, manteniendo el requisito de la cantidad no-negativa de pociones acumuladas. Nos queda especificar la noción de distancia total de la secuencia:

$$distancia(orden) = \sum_{i=1}^{|orden|-1} \sqrt{(X_i - X_{i-1})^2 + (Y_i - Y_{i-1})^2}$$

Por ejemplo, si se nos presenta una mochila con capacidad 4, dos gimnasios correspondientes a nodos $(0, 0, 2)$ y $(1, 1, 4)$ y dos paradas (0,1) y $(1, 0)$, una solución óptima es aquella de distancia 3 dada por la secuencia $orden = \langle (0, 1), (0, 0, 2), (1, 0), (1, 1, 4) \rangle$ (no es la única óptima porque $orden = \langle (1, 0), (0, 0, 2), (0, 1), (1, 1, 4) \rangle$ también es solución con distancia 3).

El problema es de la clase NP-Hard, esto se puede ver dado que se puede transformar una instancia del problema conocido como *Shortest Hamiltonian Path* sobre grafos euclideanos (NP-completo) [1]) en una instancia de nuestro problema solamente tomando por cada nodo un gimnasio que consuma 3 pociones y una parada superpuesta al gimnasio. Permitiéndonos cierta informalidad, un supuesto algoritmo polinomial que resolviera nuestro problema recorrería de a pares superpuestos dado que, para mantener la cantidad de pociones no-negativa, hay que alternar paradas y gimnasios. Siendo claramente los gimnasios superpuestos a cada una de las paradas la forma óptima de tomar estos gimnasios, porque de lo contrario el camino estaría no solamente pagando la distancia entre gimnasios sino también entre gimnasios y paradas. Que sea de la clase NP implica que los algoritmos necesarios para conseguir una solución exacta al problema son del tipo de los que recorren el espacio de soluciones del problema buscando el orden apropiado para las ubicaciones. Mencionaremos más sobre esto en la sección correspondiente a dichas implementaciones.

[1] Christos H. Papadimitriou, Computational Complexity, Page 190, Addison-Wesley Publishing Company, Inc.,1994.

1.1. Funciones auxiliares recurrentes y representación del grafo mapa

Antes de proceder específicamente sobre las implementaciones es conveniente explicar cómo representamos y tratamos las ubicaciones del mapa del problema. Si bien, como dijimos, las ubicaciones se corresponden a tuplas de la pinta (X_k, Y_k, P_k), las enumeraremos como índices de 0 a $(n + m - 1)$ en el orden que se pasan por parámetro (primero n gimnasios y luego m paradas) de modo que para saber si una ubicación es gimnasio o parada basta con comparar su valor numérico contra n y m.

De esta manera, el grafo que sobre el cual trabajaremos consiste en dos secuencias de nodos (X, Y, P) para gimnasios y paradas (el índice número i es el nodo gimnasio i-ésimo o la parada (i-n)-ésima). Este grafo es completo dado que no hay restricción sobre qué ubicaciones se pueden visitar consecutivamente.

También por conveniencia, presentamos el pseudocódigo de algunas funciones auxiliares que utilizaremos regularmente en las subsecciones de las implementaciones. Estas son aquellas que definen la validez de una secuencia como camino dado un tamaño de mochila y una cantidad acumulada de pociones hasta el momento (actualizando además dicha cantidad), y la distancia de un camino dado hasta su último gimnasio. Más adelante se detallará por qué, pero para buscar caminos mínimos en nuestro problema nos alcanza con saber distancias hasta el último gimnasio, sin contar las paradas restantes.

```
Def esCaminoValido(camino, mochila, pocionesAcum) → bool:
    Para i en camino:
        // Las paradas siempre valen -3
        nodo ← nodos_gimnasios[i] si i < #Gimnasios sino nodos_paradas[i − #Gimnasios]
        pocionesAcum ← pocionesAcum − nodo.p
        Si pocionesAcum < 0:
            terminar ciclo
        Sino si pocionesAcum > mochila:
            pocionesAcum ← mochila

    Retornar pocionesAcum ≥ 0

Def esCaminoValido(camino, mochila) → bool:
    Retornar esCaminoValido(camino, mochila, 0)
```

Es fácil ver que la complejidad es $\mathcal{O}(|camino|)$, dado que el cuerpo del ciclo solamente realiza comparaciones y asignaciones numéricas o de nodos (tuplas numéricas).

```
Def distanciaCamino(camino) → float:
    // |camino| > 0
    dist ← 0
    ultimo ← camino[0]
    desdeUltimoGim ← 0
    Para i en camino:
        desdeUltimoGim ← desdeUltimoGim + distancia(ultimo, i)
        Si i < #Gimnasios:
            dist ← dist desdeUltimoGim
            desdeUltimoGim ← 0
        ultimo ← i

    Retornar dist

Def distancia(i, j) → float:
    nodo_i ← nodos_gimnasios[i] si i < #Gimnasios sino nodos_paradas[i − #Gimnasios]
    nodo_j ← nodos_gimnasios[j] si j < #Gimnasios sino nodos_paradas[j − #Gimnasios]
    dx ← nodo_i.x − nodo_i.x
    dy ← nodo_i.y − nodo_i.y
    Retornar sqrt(dx^2 + dy^2)
```

Para $distanciaCamino$ se vuelve a proponer una complejidad temporal peor caso de $\mathcal{O}(|camino|)$ dado que las asignaciones y comparaciones del resto del algoritmo son $\mathcal{O}(1)$ (se asume esta complejidad para la operación $distancia$ si consideramos $sqrt$ constante en peor caso independientemente de la

arquitectura, considerando una cantidad de $ticks$ de CPU necesaria acotada [2])

1.2. Generadores

Para poder probar nuestras heurísticas diseñamos una serie de generadores de instancias aleatorios que se detalla a continuación.

Cada uno recibe como variable la cantidad de gimnasios y paradas, y el tamaño de la mochila deseados (y opcionalmente la seed a utilizar en el generador de números aleatorios).

1.2.1. Generador `random`

El generador más general, que usaremos en la mayoría de las mediciones. Es capaz de generar cualquier instancia válida (con las posiciones x e y dentro de los límites dados).

Comienza ubicando cada uno de los gimnasios y paradas en una posición aleatoria con $0 \leq x, y \leq 65535$. A cada uno de gimnasios le asigna una cantidad de pociones aleatoria entre 0 y 10, cuidando que la cantidad total nunca supere 3 veces la cantidad de paradas.

A continuación puede verse el pseudocódigo:

```
Def randomGenerator(int nGyms, int nStops, int bagSize) → {[Gimnasio], [Parada]}:
    gyms ← {}
    stops ← {}
    pocionesRestantes ← nStops * 3

    Para i de 0 a nGyms:
        int gymsRestantes ← nGyms − i − 1
        int potas ← min(random(0,10), pocionesRestantes − gymsRestantes)
        potas = min(potas, bagSize)

        int x ← random(0,65535)
        int y ← random(0,65535)
        agregar a gyms un gimnasio con {x,y,potas}
        pocionesRestantes ← pocionesRestantes − potas

    Para i de 0 a nStops:
        int x ← random(0,65535)
        int y ← random(0,65535)
        agregar a stops una parada con {x,y}

    Retornar {gyms, stops}
```

Es importante notar que existen casos donde se podría generar instancias sin solución. Por ejemplo, si deseamos una instancia con 2 gimnasios, 3 paradas y tamaño de mochila 5 podría generarse el problema descripto en el cuadro 1.

Nodo	X	Y	potas
Gimnasio 1	0	0	5
Gimnasio 2	1	0	4
Parada 1	2	0	-3
Parada 2	3	0	-3
Parada 3	4	0	-3

Cuadro 1: Instancia sin solución

1.2.2. Generador `separated`

La idea de este generador es agrupar los gimnasios y las paradas separando unos de otros, y generando dos *cúmulos* de puntos.

El algoritmo es similar al del `random`, descripto en la sección 1.2.1, pero restringiendo las posiciones de los gimnasios a $0 \leq x, y \leq 16383$, y la de las paradas a $0 \leq y \leq 16383$ y $32768 \leq x \leq 49152$.

[2] http://stackoverflow.com/questions/6884359/c-practical-computational-complexity-of-cmath-sqrt

1.2.3. Generador `zigzag`

Este generador intenta generar instancias donde el camino solución deba ir zigzageando entre dos columnas. Para ello fuerza el tamaño de la mochila a 3, la cantidad de pociones a 3, y coloca los gimnasios y las paradas en dos columnas separadas.

El algoritmo nuevamente es una modificación del `random` de la sección 1.2.1, esta vez fijando $x = 0$ para los gimnasios e $x = 16383$ para las paradas y variando la coordenada y de ambos entre 0 y 16383. Además, setea el tamaño de la mochila a 3 y la cantidad de pociones de cada gimnasio a 3 (si hay suficientes paradas, sino debe dejar algunos en 0 para asegurar que haya solución).

Este nuevo pseudocódigo se muestra a continuación:

```
Def zigzagGenerator(int nGyms, int nStops, int bagSize) → {[Gimnasio], [Parada]}:
    gyms ← {}
    stops ← {}
    setear el tamano de la mochila a 3

    Para i de 0 a nGyms:
        int potas ← 3 if i < nStops else 0
        int y ← random(0,16383)
        agregar a gyms un gimnasio con {0,y,potas}

    Para i de 0 a nStops:
        int y ← random(0,16383)
        agregar a stops una parada con {16383,y}

    Retornar {gyms, stops}
```

2. Algoritmo exacto

Nuestro primer acercamiento al problema consiste encontrar una solución exacta al problema, es decir, alguna de las secuencias de menor distancia (no necesariamente existe una única) con el orden de ubicaciones a recorrer correspondiente al camino deseado. Como anticipamos en la introducción, para encontrarla vamos a tener que recorrer potencialmente todo el espacio de soluciones posibles para hallar la exacta.

2.1. Fuerza bruta sobre permutaciones totales

La forma más simple de implementar un algoritmo exacto es iterando sobre cada permutación posible de las ubicaciones y comparar cada una contra la de menor distancia vista hasta al momento. Como el problema no requiere recorrer todas las paradas pero sí los gimnasios, solamente consideraremos la distancia hasta el último gimnasio de cada secuencia. No podría suceder que una solución fuera óptima recorriendo más paradas tras haber recorrido todos los gimnasios de no ser que estas estén superpuestas en el plano con el último gimnasio, sumando distancias nulas, pero aún así nuestro algoritmo tomaría aquella que termina en este último gimnasio dado que ya recorrió todos y tendrá distancia menor o igual a aquellas secuencias de las que es prefijo.

2.1.1. Pseudocódigo del algoritmo

Si bien cada vez que encontramos una mejor secuencia podríamos ir pisando una variable auxiliar, para ahorrarnos esa variable y sus asignaciones guardamos el *'índice' de la permutación* (la std de C++ provee funciones para avanzar y retroceder entre permutaciones respecto de un orden lexicográfico sobre la secuencia, que volveremos a mencionar más adelante para tratar la complejidad del algoritmo). Una vez iteradas todas las permutaciones, retrocedemos desde la última hasta aquella con el índice deseado. La decisión no afecta la complejidad final del algoritmo.

```
mejorDist ← ∞
mejorComb, comb ← 0

Para cada orden ← permutacion de ⟨0... #gims + #paradas − 1⟩:
    Si esCaminoValido(orden, tam_mochila, 0):
        dist ← distanciaCamino(orden)
        Si dist < mejorDist:
            mejorDist ← dist
            mejorComb ← comb
    comb++

Mientras comb > mejorComb:
    comb−−
    anteriorPermutacion(orden)

ultimoGim ← indice del ultimo gimnasio de orden

orden ← orden[0..ultimoGim]

Retornar ⟨mejorDist, |orden|, orden⟩
```

2.1.2. Complejidad del algoritmo

Primero consideramos el costo de inicializar las estructuras que usamos, que es $\mathcal{O}(\#gimnasios + \#paradas)$ dado que orden es la única variable cuya asignación no es en $\mathcal{O}(1)$.

Las iteraciones sobre permutaciones se realizan un total de $(\#gimnasios + \#paradas)!$ veces (cantidad de permutaciones de la secuencia orden, considerando que es creciente estricta y todos sus elementos distintos por lo tanto), con el costo en cada iteración de conseguir la próxima permutación en $\mathcal{O}(1)$ amortizado [3], evaluar si el camino generado es válido en $\mathcal{O}(\#gimnasios+\#paradas)$ peor caso (como se menciona en la introducción del informe), y comparando y actualizando cada variable al conseguir mejores soluciones, ambos valores numéricos por lo que consideramos su comparación y asignación $\mathcal{O}(1)$. Por lo tanto nos queda un ciclo con complejidad $\mathcal{O}((\#gimnasios + \#paradas) * (\#gimnasios + \#paradas)!) = \mathcal{O}((\#gimnasios + \#paradas)!)$

Luego tenemos el costo de retroceder en las permutaciones hasta la mejor combinación vista, en peor caso hace falta recorrer todas de vuelta: $\mathcal{O}((\#gimnasios + \#paradas)!)$ (se asume la misma complejidad para retroceder que para avanzar entre permutaciones).

Finalmente encontrar el índice de la última ubicación correspondida a un gimnasio nos cuesta siempre el largo del vector en peor caso, $\mathcal{O}(\#gimnasios + \#paradas)$, dado que en su implementación iteramos de fin a principio el vector buscando la 'primer' aparición [4]. Luego hacemos el recorte, que en su implementación correspondiente se hace con la operación $resize$ sobre el tipo $vector$, con complejidad equivalente a la cantidad de elementos recortados [5], es decir, $\mathcal{O}(\#gimnasios+\#paradas)$ peor caso.

Sumando las complejidades de cada porción de código llegamos a una complejidad total de $\mathcal{O}((\#gimnasios+\#paradas)!)$.

2.2. Backtracking con podas

Considerando que la complejidad de peor caso del algoritmo de fuerza bruta anterior es similar a la de todos los casos porque siempre se iteran las $(\#gimnasios + \#paradas)!)$ permutaciones posibles, surge la necesidad de poder optimizar utilizando podas u otras estrategias para mejorar rendimiento. Para esto hace falta reformular el algoritmo como un algoritmo sobre backtracking que permita ejecutar de forma ramificada en DFS y abortar ramas de ejecución según sea conveniente.

2.2.1. Pseudocódigo del algoritmo

En nuestra formulación recursiva del backtracking se empieza desde la posición $pos = 0$ de un vector 'orden' (donde $orden[0..pos]$ es el camino generado hasta el momento) y se van eligiendo las ubicaciones numeradas del 0 a $(\#gimnasios + \#paradas - 1)$ para la posición siguiente que no hayan sido ya utilizadas y que además formen un camino válido, es decir, que mantengan no negativa la cantidad de pociones, de modo que no se siga trabajando sobre caminos cuyos prefijos ya son inválidos (lo que los invalida a ellos también). Esta poda es la más básica de las que vamos a realizar, luego presentaremos algunas más con la idea de poder contrastar su rendimiento combinado.

En cada iteración se actualizan, en función de la ubicación insertada en pos (si $pos \neq 0$), variables como un contador de gimnasios recorridos (cuando se recorren todos se compara la distancia total contra la mínima hasta el momento y se procede a otra rama) y la distancia actual.

[3] http://stackoverflow.com/questions/4973077/the-amortized-complexity-of-stdnext-permutation

[4] En realidad, en la implementación la búsqueda devuelve un iterador sobre el cual luego se recorta llamando a $resize$ del vector 'orden' con la distancia entre el iterador devuelto y $orden.begin()$. Pero a efectos de complejidad es equivalente y se simplifica la lectura del pseudocódigo considerando índices.

[5] http://en.cppreference.com/w/cpp/container/vector/resize#Complexity

```
    //pow = cantidad de pociones
Def recursiva(pos, gymCounter, powAcumulado, dist) → void:

    // Calculo distancia para orden[0..pos]
    Si pos ≥ 1:
        dist ← dist + distancia(orden[pos − 1], orden[pos])

    // Vemos si orden[0..pos] es solucion
    Si esGimnasio(orden[pos]):
        gymCounter++

    Si gymCounter = ngyms:
        // Llegamos a una posible solucion
        Si distancia < mejorDist:
            mejorDist ← dist
            mejorOrden ← orden
            mejorOrdenLen ← pos
        Retornar
    }

    Si pos + 1 = #gimnasios + #paradas:
        Retornar

    Para i ← 0 ... #gimnasios + #paradas − 1:
        Si usado[i]:
            continuar
        orden[pos+1] ← i

        nuevoPowAcumulado ← powAcumulado
        Si ¬esCaminoValido(orden[pos + 1..pos + 2), tam_mochila, nuevoPowAcumulado):
            continuar

        usado[i] ← true
        recursiva(pos + 1, gymCounter, nuevoPowAcumulado, dist);
        usado[i] ← false
```

Debido a nuestra formulación, el algoritmo requiere un *launcher* que se encargue de inicializar las estructuras y variables (globales) para luego hacer los llamados a la función para $pos = 0$:

```
orden ← [ngyms + nstops] × (−1)                    // orden = [-1,-1...-1]

mejorDist ← ∞
mejorOrdenLen ← 0
mejorOrden ← orden
used ← [#gimnasios + #paradas] × (false)

Para i ← 0 ... #gimnasios + #paradas − 1:
    nodo_i ← nodos_gimnasios[i] si esGimnasio(i) sino nodos_paradas[i − #gimnasios]
    Si nodo_i.p > 0:
        //No se puede comenzar con cantidad negativa de pociones
        continuar
    orden[0] ← i;
    used[i] ← true;
    recursiva(0, 0, −nodo_i.p, 0);
    used[i] ← false;
orden ← mejorOrden

Retornar ⟨mejorDist, |orden|, orden⟩
```

2.2.2. Complejidad del algoritmo

Empezando por el *launcher*, asignar los vectores nos cuesta $\mathcal{O}(\#gimnasios + \#paradas)$, el resto de las asignaciones $\mathcal{O}(1)$. El ciclo itera $n + m = \#gimnasios + \#paradas$ veces realizando únicamente comparaciones y asignaciones numéricas o booleanas, lo cual es $\mathcal{O}(1)$, dejando una complejidad total

de $\mathcal{O}(n+m)$. Ahora veamos el costo de la función $recursiva$, para luego calcular cuántas veces es llamada.

Las guardas de los condicionales antes del ciclo son todas comparaciones numéricas (la representación del grafo, presentada en la introducción, hace que saber si un nodo es gimnasio o no sea consultar si su numeración es menor a la cantidad de gimnasios totales) por lo que su costo es $\mathcal{O}(1)$. A su vez sus cuerpos solamente realizan asignaciones, algunas dentro de condicionales anidados con guardas también $\mathcal{O}(1)$, de a lo sumo $\mathcal{O}(n+m)$ para actualizar un nuevo mejor orden. Por lo tanto, antes del ciclo de la función, tenemos un costo de $\mathcal{O}(n+m)$.

Dentro del ciclo *for*, que se ejecuta $n+m$ veces, tenemos asignaciones de variables y elementos de vectores numéricos/booleanos con costo $\mathcal{O}(1)$ además del costo de consultar si el nuevo camino formado es válido con el tamaño de mochila y la cantidad de pociones acumuladas hasta el momento lo cual es $\mathcal{O}(1)$ porque se calcula asumiendo que el camino hasta la ubicación anterior era válida (de lo contrario se hubiera realizado una poda) y solamente se chequea si la nueva ubicación es válida. Por lo tanto el ciclo tiene un costo $\mathcal{O}(n+m)$.

Por último, queremos una cota de peor caso para la cantidad de veces que es llamada la función recursivamente. Si bien a primera vista podría suponerse que en cada llamado se abren $n+m$ ramas posibles, teniendo $n+m$ posiciones para llenar a lo sumo (quedando $(n+m)^{n+m}$ llamados), esto no considera la poda que realiza la guarda que consulta si la ubicación escogida para $orden[pos+1]$ ya fue utilizada. Esto significa que en cada iteración descartamos una ubicación para el camino, realizando n llamados para $pos=0$, $n-1$ para $pos=1$, hasta llegar a una única ubicación libre en el llamado con $pos=n-1$. Esto nos deja un total de $(n+m)!$ potenciales llamados descartando podas por invalidez de caminos (en peor caso son todos válidos) con costo $\mathcal{O}(n+m)$.

Por lo tanto, sumando el costo de la inicilización del *launcher* al producto entre la cantidad espectulada de llamadas y tu respectivo costo tenemos una complejidad $\mathcal{O}(n+m+(n+m)*(n+m)!) = \mathcal{O}((n+m)!)$. Que es, en potencial y teórico peor caso, equivalente a la de fuerza bruta, sin considerar podas que disminuyan tiempos de ejecución en casos promedio.

2.2.3. Poda por distancias

Una poda más interesante es la resultante de abortar aquellas ramas de ejecución cuyos prefijos de la secuencia de ubicaciones visitadas llevan una distancia acumulada mayor o igual a la mejor distancia vista hasta el momento.

Esto funciona porque agregando ubicaciones al final a un camino que ya tiene distancia mayor que la solución actual, solamente pueden aumentar.

El código del algoritmo es esencialmente el mismo, solo que en el if que actualiza la distancia también se compara la distancia con la mínima:

```
[...]
Si pos ≥ 1:
    dist ← dist + distancia(orden[pos − 1], orden[pos])
    Si dist ≥ mejorDist:
        Retornar
[...]
```

La complejidad peor caso del algoritmo sigue siendo la misma dado que considera el caso en que no se poda ninguna rama y se tienen que explorar todas de principio a fin.

2.2.4. Poda por distancias partiendo de un greedy

La poda anterior mejora su rendimiento cuanto antes encuentre el backtracking una solución buena que permita recortar la mayor cantidad de ramas posibles. Si las primeras soluciones encontradas no

dan distancias cercanas a la mínima entonces vamos a profundizar sobre varias ramificaciones que no proporcionan soluciones óptimas.

Para mejorar esto, propusimos utilizar la implementación greedy de *vecino más cercano* (que presentaremos más adelante) para conseguir previo al backtracking una distancia máxima inicial que no dependa necesariamente de las primeras ramas exploradas.

Si bien ese llamado tiene un costo temporal en peor caso cuadrático en función de la cantidad de ubicaciones en el mapa (la cota viene del análisis que haremos en dicha sección), en notación $\mathcal{O}$ el costo del backtracking lo 'absorbe'. A nivel experimental nos interesaría ver si las podas iniciales gracias a una potencial cota decente del greedy justifican el costo del llamado a dicha implementación.

También podría suceder que el greedy no encuentre una solución y devuelva distancia infinita, en cuyo caso el backtracking funcionaría igual que en con la poda de la subsección anterior pero con la penalidad temporal del llamado al greedy.

3. Heurística constructiva-golosa

Como vimos en la sección anterior, el costo en complejidad temporal de un algoritmo que nos asegure una solución exacta a nuestro problema no es desdeñable incluso aplicando podas y estrategias para aminorar tiempos de ejecución. En esta sección nos encargaremos de presentar técnicas heurísticas para el problema, particularmente del tipo *greedy*, que proporcionen soluciones en tiempo polinomial aún sin proveer garantías de soluciones óptimas.

3.1. Gimnasio más cercano

Ya mencionamos en la introducción cierto paralelismo en la naturaleza del problema con el de buscar un camino hamiltoniano mínimo, el cual a su se asemeja al *TSP*[6]. Nuestro *approach* a una heurística *greedy* está fuertemente basada en el homólogo *Nearest neighbour algorithm* [7] pensada para encontrar soluciones al *TSP*.

La idea es tomar en cada momento, de todos los gimnasios con requisitos de pociones menores a la cantidad acumulada, aquel que sea más cercano a la ubicación 'actual'. De no contar con suficientes pociones para ninguno de los gimnasios restantes, se busca la parada más cercana. En caso de que no queden paradas sin visitar y falten gimnasios por visitar el algoritmo terminará, devolviendo que no hay solución.

En muchas ocasiones el algoritmo puede no encontrar una solución factible incluso frente a la existencia de una. Particularmente por la 'saturación' de pociones respecto del tamaño de la mochila, tomando menos pociones de las que ofrece una parada (no pudiendo cumplir eventualmente el requisito de algún gimnasio posterior). Por ejemplo, sea un tamaño de mochila 5 y los siguientes 5 gimnasios y paradas:

(2, 2, 4)	(0, 0)
(2, 5, 5)	(1, 2)
(10, 10, 4)	(2, 3)
(0, 1, 1)	(2, 4)
(1, 1, 1)	(3, 5)

El algoritmo *exacto* toma la secuencia $\langle$(0, 0), (0, 1, 1), (2, 4), (2, 5, 5), (2, 3), (1, 1, 1), (1, 2), (2, 2, 4), (3, 5), (10, 10, 4)$\rangle$ con distancia aproximada $23,60$. En cambio, el algoritmo *greedy* comienza por la primer parada enumerada (como se verá más adelante en el pseudocódigo) $(0, 0)$ dado que no puede acceder a ningún gimnasio por requisitos de pociones, para luego ir a los gimnasios más cercanos $(0, 1, 1)$ y $(1, 1, 1)$ quedando con una única poción. En este punto accede a la parada $(1, 2)$ para luego visitar (con 4 pociones acumuladas) el gimnasio $(2, 2, 4)$. Como ahora los dos únicos gimnasios que quedan requieren más de 3 pociones tiene que volver a dos paradas, pero al tener una mochila de tamaño 5 va a saturar las 3 potenciales pociones en 5 necesariamente. Yendo al gimnasio más cercano $(2, 5, 5)$ queda sin pociones y con una única parada restante por visitar y un gimnasio de 4 pociones por requisito. Por lo tanto no puede terminar el camino.

Incluso no solo no provee garantías de llegar a un resultado válido, de existir, sino que tampoco podemos especular con un 'error máximo' de precisión respecto de la solución real. Supongamos que existe alguna cota k que caracterice çuántas veces la distancia mínima real"puede a lo sumo llegar a ser aquella devuelta por nuestro algoritmo. Pero entonces podríamos suponer un escenario con tamaño de mochila $3 * k$, n paradas (0,0) superpuestas y n gimnasios (k, 0, 3) también superpuestos.

El algoritmo *exacto* devolvería un orden que tome primero todas las paradas recorriendo distancia 0 y luego todos los gimnasios, con distancia k.

[6] Travelling Salesman Problem: https://en.wikipedia.org/wiki/Travelling_salesman_problem
[7] https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm

El *greedy*, en cambio, va a buscar la primer parada para tener al menos 3 pociones (necesarias para visitar cualquier gimnasio) para luego visitar alguno. Al quedar sin pociones, va a tener que reiterar el mecanismo repetidas veces hasta cubrir todas las paradas y gimnasios de manera alternada, haciendo $2k$ idas y $2k-1$ vueltas. Por lo tanto va a recorrer $k(2k-1)$ que es $2k-1$ veces la distancia del exacto, superando la supuesta cota k.

3.1.1. Pseudocódigo del algoritmo

Haciendo uso de conjuntos, implementados sobre estructuras de acceso logarítmico en su cardinal [8], guardamos por un lado el total de nodos paradas (numerados desde $\#gimnasios$ hasta $(\#gimnasios + \#paradas - 1)$ y, por el otro lado, por cada gimnasio (van de 0 a $\#gimnasios - 1$) una tupla de $\langle pociones\ requeridas,\ gimnasio \rangle$ de modo que tengamos acceso logarítmico al gimnasio con menos pociones requeridas (el tipo set ordena primero por primer componente en pares). Esto nos va a permitir recortar búsquedas lineales cuando, al recorrer de menor a mayor en requerimientos de pociones, veamos el primer gimnasio que supera nuestra cantidad acumulada.

Como el esquema del vecino más cercano nos permite elegir cualquier ubicación inicial válida, tomamos como punto de partida del camino el gimnasio 'gratuito' de menor numeración (correspondiente al mínimo sobre el conjunto, dado que es el elemento con menor segunda componente de aquellos con pociones requeridas igual a 0), sino, la primer parada enumerada.

```
distRecorrida ← 0
orden ← ⟨⟩

gims ← ∅
Para i ← 0.. #gimnasios:
    gims ← gims ∪ {⟨P(i), i⟩}

paradas ← ∅
Para i ← 0.. #paradas:
    paradas ← paradas ∪ {i + #gimnasios}
pociones_actuales ← 0

Si min(gims).first > 0:
    nodo_actual ← min(paradas)
    paradas ← paradas − min(paradas)
    pociones_actuales ← min{3, tam_mochila}
Sino:
    nodo_actual ← min(gims).second
    gims ← gims − min(gims)
orden ← orden ++ [nodo_actual]

Mientras gims ≠ ∅:
    Si min(gims).first ≤ pociones_actuales:
        gim_candidato ← (g ∈ gims | g.first ≤ pociones_actuales ∧
          (∀ g' ∈ gims, g'.first≤ pociones_actuales) distancia(nodo_actual, g.second) ≤
                                                    distancia(nodo_actual, g'.second))

        dist_candidato ← distancia(nodo_actual, gim_candidato.second)

        distRecorrida ← distRecorrida + dist_candidato
        nodo_actual ← gim_candidato.second
        pociones_actuales ← pociones_actuales − gim_candidato.first
        gims ← gims − {gim_candidato}

    Sino si paradas ≠ ∅:
        parada_mas_cercana ← (p ∈ paradas | ∀ p' ∈ paradas, distancia(nodo_actual,p) ≤
                                                    distancia(nodo_actual,p'))

        dist_candidato ← distancia(nodo_actual, parada_mas_cercana)

        distRecorrida ← distRecorrida + dist_candidato
```

[8] http://en.cppreference.com/w/cpp/container/set

```
            nodo_actual ← parada_mas_cercana
            pociones_actuales ← min(pociones_actuales + 3, tam_mochila)
            paradas ← paradas − {nodo_actual}

        Sino:
            distRecorrida ← −1
            Terminar
        orden ← orden ++ [nodo_actual]
    Retornar {distRecorrida, orden.size()}
```

3.1.2. Complejidad del algoritmo

El costo de inicializar ambos conjuntos con todos sus elementos iniciales es $\mathcal{O}(\#gimnasios * log(\#gimnasios) + \#paradas * log(\#paradas))$ acotando para ambos casos el costo de cada inserción como aquel en el que ya se encuentran todos los elementos en la estructura. Además, conseguir los requisitos de pociones de cada gimnasio es $\mathcal{O}(1)$ considerando que esa información está contenida en las tuplas que representan al grafo, mencionadas en la sección de introducción, las cuales se coleccionan en vectores de acceso $\mathcal{O}(1)$ desde índices.

Para determinar la posición inicial hace falta consultar la cantidad de pociones que pide el gimnasio de menor requisitos, esto tiene un costo asociado de $\mathcal{O}(log(\#gimnasios))$. Si no es 'gratuito' entonces tenemos que asignar la ubicación actual como la menor parada numerda y eliminarla del conjunto de paradas candidatas, ambas operaciones en $\mathcal{O}(log(\#paradas))$, y luego incrementar la cantidad acumulada de pociones tomando mínimo entre las 3 que otorga la parada y el tamaño total de mochila en $\mathcal{O}(1)$. De ser gratuito, se asigna el mínimo gimnasio como ubicación actual y se lo elimina de la lista de candidatos en $\mathcal{O}(log(\#gimnasios))$. En ambos casos, agregar la nueva ubicación a la lista de orden es $\mathcal{O}(1)$ amortizado, implementado en el tipo vector de la STL de C++ [9]. Por lo tanto asignar la primer ubicación tiene un costo $\mathcal{O}(log(\#gimnasios) + log(\#paradas))$ sumando ambas ramas y la guarda.

Nos resta ver el costo del ciclo. El ciclo itera hasta cubrir todos los gimnasios candidatos, pero en peor caso necesita para ello cubrir también todas las paradas. Entonces el ciclo itera, en peor caso, un orden de $(\#gimnasios + \#paradas)$ veces.

En cada iteración se consulta si el conjunto de gimnasios candidatos es vacío, en tiempo constante [10]. Y además se inserta, como vimos ya en $\mathcal{O}(1)$ amortizado, la ubicación elegida.

En la rama en que no alcanzan las pociones actuales y quedan paradas hace falta recorrer linealmente todas las paradas en busca de aquella más cercana, como la operación de distancia tiene complejidad $\mathcal{O}(1)$ (visto en la introducción) el costo de buscar la más cercana tiene en el peor caso asociado a recorrer todas las paradas sin considerar borrados de $\mathcal{O}(\#paradas)$. Luego se actualizan variables numéricas como pociones actuales y distancia recorrida en $\mathcal{O}(1)$ y finalmente se elimina la parada elegida en $\mathcal{O}(log(\#paradas))$. La rama tiene un costo peor caso, entonces, de $\mathcal{O}(\#paradas)$ considerando también el costo de consultar en la guarda si el conjunto de candidatos está vacío que, como ya vimos, es constante.

En la rama en que existe al menos un gimnasio recorrible con la cantidad de pociones acumuladas, también hace falta buscar linealmente en la lista de gimnasios a aquel con menor distancia y que además pida menos pociones que las acumuladas, estas dos operaciones se realizan en tiempo constante y por lo tanto la búsqueda tiene un costo peor caso (al igual que en las paradas, considerando que siempre van a estar todos los gimnasios sin considerar que se van eliminando) de $\mathcal{O}(\#gimnasios)$. Nuevamente se actualizan las pociones, distancias recorridas, y ubicación actual en tiempo constante para finalmente eliminar el gimnasio elegido del conjunto de candidatos en $\mathcal{O}(log(\#gimnasios))$. Sumando todos estos costos al de consultar en la guarda si el gimnasio de requisito mínimo pide menos pociones

[9] http://www.cplusplus.com/reference/vector/vector/push_back/#complexity
[10] http://www.cplusplus.com/reference/vector/vector/empty/#complexity

de las que tenemos en $\mathcal{O}(log(\#gimnasios))$ peor caso, nos queda una complejidad total de la rama de $\mathcal{O}(\#gimnasios)$.

Despreciando el costo de la rama en que se aborta la ejecución porque solamente se actualiza la distancia recorrida y se retorna. Nos queda un costo total del ciclo que ejecuta $(\#gimnasios + \#paradas)$ con un costo por cuerpo de $(\#gimnasios + \#paradas)$, quedando la complejidad $\mathcal{O}((\#gimnasios + \#paradas) * (\#gimnasios + \#paradas)) = \mathcal{O}((\#gimnasios + \#paradas)^2)$.

3.2. Gimnasio más lejano

Como nos interesaba poder comparar el *greedy* anterior con otra heurística constructiva golosa a la hora de contrastar rendimientos con otras, implementamos otra heurística similar. El código es idéntico al de 'Gimnasio más cercano' salvo que en cada búsqueda lineal del siguiente candidato, se elije aquel que maximice la distancia al nodo actual.

El algoritmo está basado en el esquema de inserción más lejana. La idea es que, como en muchos grafos hay ubicaciones costosas de recorrer, estas sean visitadas cuanto antes con la esperanza de que las remanentes formen un grafo más compacto.

La complejidad es exactamente la misma y la implementación solo cambia la comparación de menor estricto a mayor estricto para las distancias de los candidatos.

3.3. Elección aleatoria sobre un subconjunto de gimnasios

Para poder abordar el problema más adelante con una metodología *GRASP* [11], necesitamos implementar un algoritmo de tipo $greedy$ pero que devuelva soluciones diversas al ser llamado múltiples veces sobre el mismo problema.

La estrategia de este algoritmo es similar a la del gimnasio más cercano. Su diferencia se basa en la elección de gimnasios y paradas. En vez de elegir el nodo más cercano a la ubicación 'actual', se elige uno aleatoriamente entre la mitad de los nodos más cercanos a la misma ubicación.

La lógica de elección entre ir a un gimnasio o a una parada es la misma: si no se puede ir a ningún gimnasio, entonces se busca una parada con el mismo criterio mencionado en el párrafo anterior, y luego se continua con el procedimiento desde la nueva posición 'actual'.

Al mismo tiempo, así como el algoritmo anterior, este también puede no encontrar una solución válida aún en el caso de haberla (bastaría con que el random tome siempre el elemento más cercano para que el resultado fuera el mismo).

La estrategia de este algoritmo se basa en aumentar las posibilidades de elecciones al tomar uno aleatoriamente entre la mitad de los más cercanos, ya que existen situaciones en las cuales tomar el más cercano de todos no es la mejor solución.

La sección que cambia de manera más significativa el código es en las guardas a la hora de buscar gimnasios o paradas.

```
            Si min(gims).first ≤ pociones_actuales:
                gims_candidatos ← vector de gimnasios que se pueden visitar

                sort(gims_candidatos)          //Se ordenan segun la distancia al nodo actual

                gim_candidato ← random(primera_mitad(gims_candidatos))
```

[11] https://en.wikipedia.org/wiki/Greedy_randomized_adaptive_search_procedure

```
            [...]

        Sino si paradas ≠ ∅:
            paradas_mas_cercanas ← vector de todas las paradas

            sort(paradas_mas_cercanas)      //Se ordenan segun la distancia al nodo actual

            parada_mas_cercana ← random(primera_mitad(paradas_mas_cercanas))

            [...]
```

El primer paso en cada entrada de la guarda es recorrer de manera lineal tanto los gimnasios como las paradas, con lo cual la complejidad se mantiene igual. El segundo paso es ordenar. Esto cambia la complejidad, ya que anteriormente se recorrían linealmente los gimnasios o las paradas respectivamente dependiendo de en cual de las dos condiciones de la guarda se ejecutaba. Ahora, al realizar un sort, esa parte del algoritmo pasa de ser $\mathcal{O}(\#gimnasios + \#paradas)$ a ser $\mathcal{O}(\#gimnasios \cdot log(\#gimnasios) + \#paradas \cdot log(\#paradas))$. Obtener un elemento random de un arreglo tiene complejidad $O(1)$ ya que se puede implementar con la función rand que provee C++[12], cuya complejidad es la mencionada aquí. Siguiendo la linea de razonamiento para el algoritmo del gimnasio más cercano y aplicando esta modificación, la complejidad nueva pasa a ser $\mathcal{O}((\#gimnasios + \#paradas) \cdot (\#gimnasios \cdot log(\#gimnasios) + \#paradas \cdot log(\#paradas)))$

Como la finalidad de esta implementación es únicamente en el contexto de las implementaciones de *GRASP* y, entendiendo que las soluciones que devuelve van a ser relativamente similares a la del gimnasio más cercano (seguramente con mayor varianza) al elegir siempre ubicaciones cercanas, no constituye un caso experimental en nuestro informe.

[12] http://stackoverflow.com/questions/8658784/rand-function-in-c

4. Heurística de búsqueda local

Como vimos en la sección anterior, la heurística golosa no nos provee ninguna garantía de error máximo contra soluciones reales. Una manera de 'amortizar' esto, es realizar optimizaciones por búsqueda local sobre los resultados arrojados por dicha heurística. Dada una instancia de solución al problema y un vecindario definido de otras soluciones, tomamos la vecina de menor distancia total y reiteramos con algún criterio que determine cuándo parar. Si ninguna vecina tenía menor distancia, entonces no es posible ninguna optimización desde esa misma instancia.

Presentaremos dos vecindarios experimentales con el fin de contrastar su rendimiento temporal y sus respectivos errores respecto de la solución real. Al igual que en el caso de la heurística golosa, decidimos basarnos en heurísticas clásicas utilizadas para *TSP*.

4.1. Búsqueda local por swaps de nodos

La idea es sencilla: para cualquier par dado de nodos en un camino, supongamos v y w con w recorrido posteriormente a v ¿representa una mejora en la distancia total intercambiarlos de manera que a la hora de visitar v en su lugar recorra w y viceversa?

Así, el vecindario de una solución son todas aquellas resultantes de swapear dos ubicaciones en el vector donde se encuentran por orden.

4.1.1. Pseudocódigo del algoritmo

Claramente es necesario recalcalcular la validez de un camino tras swapear dos de sus ubicaciones antes de considerarlo como candidato.

```
Def iterar_swap(orden) → ⟨float, bool⟩

    huboMejora ← false
    dist ← distanciaCamino(orden, graph)

    mejor_vecino ← orden
    Para i ← 0...|orden|−1:
        Para j ← i + 1...|orden|−1:
            orden_vecino ← orden
            orden_vecino[i] ← orden[j]
            orden_vecino[j] ← orden[i]

            Para k ← 0...|orden|−1:
                Si esGimnasio(orden[k]):
                    ultimo_gim ← k

            pow ← 0
            valido ← esCaminoValido(orden_vecino[0..ultimo_gim], tam_mochila, pow)

            Si ¬valido:
                Proximo j

            dist_vecino ← distanciaCamino(orden_vecino)

            Si dist_vecino < dist:
                huboMejora ← true
                dist ← dist_vecino
                mejor_vecino ← orden_vecino
    orden ← mejor_vecino
    Retornar ⟨ dist, huboMejora ⟩
```

Veamos ahora el contexto de búsqueda en que se invoca la función. Recibimos por entrada el orden de recorrido de alguna solución válida desarrollada por la heurística *greedy*, de no ser válida devolvemos un resultado anunciándolo pues no podríamos optimizar una solución inválida, y luego procedemos a

insertarle al final todas las paradas que no contenga dicho recorrido. Esto se debe a que, aún cuando no se hayan usado en la solución original, nos interesa maximizar el vecindario de la instancia de modo que podamos explorar la mayor cantidad posible de swapeos. Si la solución que devolveremos no necesita esas paradas para recorrer todos los gimnasios, serán recortadas nuevamente.

Un tema de interés respecto a la optimización por búsqueda local es el de escoger la cantidad necesaria de iteraciones a realizar. Por ejemplo, una opción sería iterar hasta que no se encuentre ninguna mejora en el último vecindario. Si bien maximiza posibilidades de encontrar una buena solución, es potencialmente costosa dado que si en cada llamado a $iterar_swap$ encontramos una nueva mejor solución podría llegar a ser necesario mejorar cada una de las permutaciones posibles del arreglo de ubicaciones (de cardinal factorial en su longitud) en un supuesto peor caso. Por lo tanto, buscando un comportamiento polinómico, iteramos una cantidad fija de veces de acuerdo a un parámetro de entrada (corridas). Si corridas es 0 se itera como describimos antes, hasta que no haya mejoras, de lo contrario se itera $corridas$ veces o hasta llegar a un vecindario sin mejores soluciones.

```
Def local_swap(orden, corridas) → ⟨float, int, vector⟩

    Si |orden| = 0:
        Retornar ⟨ ∞, 0, [] ⟩

    usada ← [nstops] × (false)          // usada = [false.. false]
    Para i en orden:
        Si i ≥ #gimnasios:
            usada[i − #gimnasios] ← true

    Para parada ← 0..#paradas:
        Si ¬usada[parada]:
            orden ← orden ++ [parada + #gimnasios]

    seguir ← true
    Mientras seguir:
        ult_corrida ← iterar_swap(orden);
        seguir ← ult_corrida.second
        dist ← ult_corrida.first
        Si corridas > 0:
            corridas ← corridas − 1
            seguir ← (seguir ∧ (corridas > 0))

    Para k ← 0...|orden|−1:
        Si esGimnasio(orden[k]):
            ultimo_gim ← k

    orden ← orden[0..ultimo_gim]

    Retornar ⟨ dist, |orden|, orden ⟩
```

4.1.2. Complejidad del algoritmo

Empecemos por calcular la complejidad de $iterar\_swap$. Arranca calculando la distancia del camino conformado por $orden$ con un costo de $\mathcal{O}(|orden|)$ que es $\mathcal{O}(\#gimnasios + \#paradas)$ considerando que se rellenan todas las paradas faltantes desde $local\_swap$. Luego se copia el contenido de $orden$ a $mejor\_vecino$, en $\mathcal{O}(\#gimnasios + \#paradas)$ en el mismo peor caso que ya mencionamos.

Luego se ejecuta un ciclo que itera $\mathcal{O}((\#gimnasios + \#paradas)^2)$ veces.

Dentro del ciclo se hace una copia de $orden$ en $\mathcal{O}(\#gimnasios + \#paradas)$, se hace el swap de ubicaciones en $\mathcal{O}(1)$ (asignar índices de nodos en un vector es constante), se busca el índice del gimnasio en $\mathcal{O}(\#gimnasios + \#paradas)$ [13]. Luego, se calcula si el camino nuevo es válido con costo

[13] Como ya ha sucedido en otras secciones se aclara que en nuestra implementación algunas operaciones como esta se realizan sobre iteradores pero se usan versiones sobre índices equivalentes en complejidad por simplicidad de lectura de pseudocódigo.

$\mathcal{O}(\#gimnasios + \#paradas)$ peor caso (que el último gimnasio sea la última ubicación del arreglo de órdenes), se calcula también su longitud hasta el último gimnasio, de ser válido el camino, en $\mathcal{O}(\#gimnasios + \#paradas)$ y por último se actualizan las variables que guardan los mejores resultados vistos hasta el momento en $\mathcal{O}(\#gimnasios + \#paradas)$, costo correspondiente al copiado de $orden\_vecino$ en el peor caso de contener todas las paradas además de gimnasios.

Por lo tanto tenemos un costo total del ciclo de $\mathcal{O}((\#gimnasios + \#paradas)^3)$. Que se corresponde también con el costo total de la función considerando que absorbe los costos del primer cálculo de distancia y asignar a $orden$ la mejor solución hallada (ambos lineales en el tamaño de orden).

Para $local\_swap$ empezamos corroborando que $orden$ sea una secuencia de tamaño mayor que 0 y retornando un valor específico en $\mathcal{O}(1)$. Inicializamos el vector de *bool* sobre paradas usadas en $\mathcal{O}((\#paradas))$ y luego se marcan en $\mathcal{O}(|orden|)$, que en peor caso es $\mathcal{O}(\#gimnasios + \#paradas)$ por contener todas las ubicaciones, para después agregar en $\mathcal{O}(1)$ todas las paradas recorridas linealmente que no estén usadas ($\mathcal{O}(\#paradas)$).

Para el ciclo que convoca a $iterar\_swap$ aparte del llamado a esta función solamente realizamos asignaciones y comparaciones numérico-booleanas, por lo que prepondera la complejidad de $iterar\_swap$ que es, como vimos antes, cúbica en la cantidad total de ubicaciones.

Como la cantidad de veces que ejecuta el ciclo depende de si $corridas$ es mayor a 0 o igual, se separan dos casos: ejecutar el ciclo $corridas$ veces o ejecutar hasta que no se encuentren mejoras. Este último caso resulta ser de orden factorial hipotetizando algún peor caso en el que se realice una actualización (produciendo mejora) por cada permutación posible de $orden$. Aún si en cada llamado a $iterar\_swap$ cubrimos la cuota superior de mejorar un orden cuadrático de veces, aún faltaría hacer $\mathcal{O}((\#gimnasios + \#paradas)!/(\#gimnasios + \#paradas)^2) = \mathcal{O}((\#gimnasios + \#paradas)!)$ llamados hasta dejar de ver mejoras.

El ciclo entonces queda, con complejidad $\mathcal{O}(corridas * (\#gimnasios + \#paradas)^3)$ si $corridas > 0$, o $\mathcal{O}((\#gimnasios + \#paradas)! * (\#gimnasios + \#paradas)^3) = \mathcal{O}((\#gimnasios + \#paradas)!)$ si se debe iterar hasta dejar de ver mejoras. En ambos casos la complejidad de ubicar linealmente el índice del último gimnasio y recortar $orden$ es, en peor caso, $\mathcal{O}(\#gimnasios + \#paradas)$.

Por lo tanto, sumando todas las complejidades, donde preponderan las del ciclo, queda $\mathcal{O}(corridas * (\#gimnasios + \#paradas)^3)$ si $corridas > 0$, o $\mathcal{O}((\#gimnasios + \#paradas)!)$ si $corridas = 0$. Esta última cota parecería indicar que iterar de esta manera sería similar en costo a la del backtracking o fuerza bruta, pero previo a realizar experimentaciones podemos especular con que realmente no sucedan por lo general casos donde se prolongue significativamente el hallazgo continuo de mejoras. Más adelante contrastaremos empíricamente si esto efectivamente es así.

4.1.3. Variación alternativa del algoritmo

Originalmente la idea que se nos había ocurrido era, en vez de tomar el mejor de los vecinos de la instancia en cada iteración, tomar el primer vecino que fuera mejor que el actual y seguir luego sin repetir pares (i, j) aún tras haber actualizado la instancia sobre la cual se itera su vecindario.

Es decir, cada vez que vemos una solución que mejore a la actual, la pisamos y seguimos con los pares (i, j) restantes. Esto último se debe a que si volvieramos a tomarlos desde $(0, 0)$ cada vez que se actualiza la solución, caeríamos en una situación similar a la del peor caso con $corridas = 0$ del algoritmo anterior donde potencialmente se recorrían todas las permutaciones (incluso cuando $corridas$ fuera distinta de 0).

Al no repetir pares (i, j) dentro de un mismo llamado y siendo el código extremadamente similar (en vez de actualizar $mejor\_vecino$ se pisa $orden$ con $orden\_vecino$) la complejidad se mantiene igual que en la subsección anterior. Por los motivos anteriores no se considera necesario repetir el pseudocódigo.

Aunque no tome necesariamente el mínimo de los vecinos como sí lo hace la implementación que mostramos anteriormente, nos interesó la cuestión de, si al tomar siempre la primer solución 'buena' de cada vecindario y pudiendo actualizar más de una solución por llamado a $iterar\_swap$, se produjeran mejoras en rendimiento. Esto lo pondremos a prueba en la sección de experimentación.

4.2. Búsqueda local por 2opt

Dado un camino, ¿existen dos pares continuos de ubicaciones $(orden_i,\ orden_{i+1})$ y $(orden_k,\ orden_{k+1})$ tal que $distancia(orden_i,\ orden_{i+1})$ + $distancia(orden_k,\ orden_{k+1})$ $<$ $distancia(orden_i,\ orden_k)$ + $distancia(orden_{i+1},\ orden_{k+1})$?

De ser así, podemos 'swapear' las aristas $(orden_i,\ orden_{i+1})$ y $(orden_k,\ orden_{k+1})$ por $(orden_i,\ orden_k)$ y $(orden_{i+1},\ orden_{k+1})$ obteniendo un camino de mejor longitud aunque no sea necesariamente válido, por lo que haría falta corroborar su validez. Esta estrategia se llama $2opt$ [14] y es un caso particular de la heurística *Lin–Kernighan* [15] usada generalmente para *TSP*.

Cabe destacar que en grafos dirigidos esto implicaría recorrer en sentido inverso $orden[i+1..k]$, lo cual no afecta la distancia del subcamino. Entonces nos alcanza con realizar la transformación $orden \leftarrow orden[0..i]{+}{+}reverso(orden[i{+}1..k]){+}{+}orden[k{+}1..|orden|{-}1]$ para realizar el swap de aristas deseado.

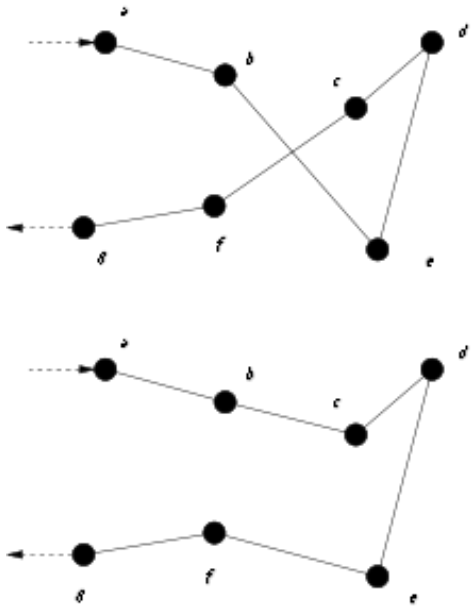

Figura 1: Sea (b, e) la arista $(orden_i,\ orden_{i+1})$ de nuestro ejemplo y (c, f) nuestra $(orden_k,\ orden_{k+1})$, se puede apreciar que se recorre de manera reversa $orden[i + 1..k] = [e, d, c]$ tras el swap 2opt.

Definimos entonces el vecindario de una solución para esta heurística como todas aquellas secuencias de ubicaciones resultantes de invertir una subsecuencia.

[14] https://en.wikipedia.org/wiki/2-opt
[15] https://en.wikipedia.org/wiki/Lin-Kernighan_heuristic

4.2.1. Pseudocódigo del algoritmo

Al igual que con la búsqueda local con swaps de nodos, se hacen reiterados llamados a la función que realiza una iteración de $2opt$, pudiendo elegir si se hace una cantidad $corridas$ de iteraciones o si se itera hasta dejar de ver mejoras (en caso de que se itere una cantidad definida de veces y que se dejen de ver mejoras antes de completar dicha cantidad, también se deja de iterar).

El pseudocódigo de la función $local\_dos\_opt$ es idéntico a $local\_swap$ solamente que llama a $iterar2opt$ mientras itera. Por lo tanto nos abstenemos de escribir sus pseudocódigo.

Nuevamente es necesario rellenar las paradas que falten y recortarlas al final, para maximizar los vecindarios de las instancias.

Para esta búsqueda local, decidimos ir pisando la variable $orden$ sobre la cual se explora su vecindario al momento de encontrar un vecino mejor. Esto, como ya comentamos, significa no recorrer potencialmente todo el vecindario pero también buscar en más vecindarios.

```
Def iterar2opt(orden, verbose) → ⟨float, bool⟩:

    dist ← distanciaCamino(orden)

    huboMejora ← false

    Para i ← 0..|orden|−1:
        Para j ← i + 1..orden.size()−1:
            orden_vecino ← orden[0..i] ++ reverso(orden[i+1..j]) ++ orden[j+1..|orden|−1]

            Para k ← 0...|orden|−1:
                Si esGimnasio(orden[k]):
                    ultimo_gim ← k

            pow ← 0
            valido ← esCaminoValido(orden_vecino[0..ultimo_gim], tam_mochila, pow)

            Si ¬valido:
                Proximo j

            dist_vecino ← distanciaCamino(orden_vecino)

            Si dist_vecino < dist:
                huboMejora ← true
                dist ← dist_vecino
                orden ← orden_vecino
    Retornar ⟨ dist, huboMejora ⟩
```

4.3. Complejidad del algoritmo

El algoritmo tiene la misma complejidad temporal que $iterar\_swap$, dado que en lo único que difieren es en la función que construye a los vecinos, en este caso $\mathcal{O}(|orden|)$ dado que se copian 3 tercios, de los cuales uno se invierte también en $\mathcal{O}(|orden|) = \mathcal{O}(\#gimnasios + \#paradas)$ porque se agregan todas las paradas en $local\_dos\_opt$.

Como dijimos al analizar $iterar\_swap$ (que tiene las mismas guardas), el resto del cuerpo del ciclo tiene complejidad también $\mathcal{O}(\#gimnasios + \#paradas)$. Se ejecuta dicho cuerpo un orden cuadrático (en la cantidad de ubicaciones) de veces, dejando un ciclo for anidado con costo $\mathcal{O}((\#gimnasios + \#paradas)^3)$ que además es mayor que el costo de inicializar $dist$ llamando a $distanciaCamino$ en $\mathcal{O}(\#gimnasios + \#paradas)$, haciendo que la complejidad final de la función sea cúbica en la cantidad de ubicaciones.

Siguiendo lo mencionado sobre $local\_dos\_opt$ siendo idéntica a $local\_swap$, se podría especular con

que tengan además las mismas complejidades. Como lo único que podría llegar a cambiar es la cantidad de veces necesarias a iterar hasta no encontrar mejoras (si $corridas$ ¿0 entonces la complejidad es la misma que su homóloga pues se realizan $corridas$ veces llamados a una función cúbica y el resto del código es el mismo), queremos ver si esta cantidad es la misma que en su caso homólogo.

Esto es así porque a lo sumo podemos tener que ver $\mathcal{O}((\#gimnasios + \#paradas)!)$ mejoras (una por cada permutación). Por cada iteración se cubre a lo sumo una cantidad cuadrática de mejoras, quedando $\mathcal{O}((\#gimnasios + \#paradas)!/(\#gimnasios + \#paradas)^2) = \mathcal{O}((\#gimnasios + \#paradas)!)$ llamados peor caso a la función $iterar2opt$ de complejidad cúbica sobre las ubicaciones totales.

Por lo tanto, al igual que $local\_swap$, $local\_dos\_opt$ tiene complejidad $\mathcal{O}((\#gimnasios + \#paradas)! * (\#gimnasios + \#paradas)^3) = \mathcal{O}((\#gimnasios + \#paradas)!)$ al realizar un orden factorial de llamados a una función cúbica con el resto del código con menor complejidad peor caso. Pero en los casos en que $corridas > 0$ la complejidad es idéntica a la del mismo caso para $local\_swap$, $\mathcal{O}(corridas * (\#gimnasios + \#paradas)^3)$ por realizar $corridas$ veces una función cúbica.

5. GRASP

Nuestra última optimización se trata de una metaheurística [16] de tipo *GRASP (Greedy randomized adaptive search procedure)*, que consiste en mejorar sucesivamente mediante búsqueda local una lista de soluciones candidatas arrojadas por nuestra función greedy aleatoria y luego tomando el mínimo de todas las soluciones ya optimizadas.

Si bien el esquema de esta metaheurística es sencillo y concreto, hay mucho lugar para variaciones en la manera en que se toman las listas de candidatos (e.g tamaño, si se filtran con algún criterio, etc), criterios de parada o métodos de búsqueda local.

5.1. Búsquedas locales alternadas

Nuestra primer combinación de variantes toma como parámetro un exponente i que indica que se va a considerar un total de $(\#gimnasios + \#paradas)^i$ candidatos construídos por el *greedy random*, otro exponente j que indica que se van a realizar $(\#gimnasios + \#paradas)^j$ iteraciones de búsqueda local sobre los candidatos.

Estas variables definen un criterio de parada que asegura un comportamiento polinómico sobre la cantidad de gimnasios y paradas totales.

Además, para obtener mayor diversidad de características entre los candidatos, alternamos las búsquedas locales entre *swap de nodos* y *2opt* para optimizarlos.

```
Def grasp_alternado(orden, i, j) → ⟨float, int⟩:
    mejorRes ← ∞

    n ← #gimnasios + #paradas

    cant_candidatos ← n^i
    limite ← n^j

    Para i ← 0..cant_candidatos−1:
        resGreedy ← greedy_random(orden_actual)

        Si resGreedy.first = ∞:
            Proximo i

        res ← local_dos_opt(orden_actual, limite) Si i es par
              local_swap(orden_actual, limite) Sino

        Si res.first < mejorRes:
            mejorRes ← res.first
            orden ← orden_actual

    Retornar ⟨mejorRes, |orden|⟩
```

5.2. Mejores candidatos

Mientras en el caso anterior nos permitíamos mejorar indiscriminadamente todas soluciones que nos devolvía el *greedy random* una manera de reducir tiempos de ejecución es también ser más restrictivos sobre qué candidatos vamos a considerar y cuáles no.

Todavía alternando búsquedas locales, presentamos uan segunda implementación de *GRASP* para nuestro problema donde, todavía alternando búsquedas locales, solamente vamos a mejorar un grupo selecto de candidatos.

[16] https://en.wikipedia.org/wiki/Metaheuristic

En particular, nos limitamos únicamente a hacer búsquedas locales sobre a lo sumo la 'mejor mitad' (en cuanto a distancias de cada solución refiere) de las que anteriormente considerábamos. La intuición detrás de esto es que estas soluciones tendrían que iterar menos hasta encontrar un mínimo local o que den mejores resultados tras ser mejoradas.

Para no tener que realizar ordenamientos, directamente agregamos los candidatos en un set de tuplas ($distancia$, $orden$) que, al ordenar por primer componente, nos garantice acceso al mínimo y borrado logarítmico en sus elementos.

```
Def grasp_trim(orden, expLimite, expInicios) → ⟨float, int⟩
    mejorRes ← ∞

    n ← #gimnasios + #paradas

    cant_candidatos ← n^i
    limite ← n^j

    candidatos ← ∅
    Para i ← 0...cant_candidatos−1:
        orden_candidato ← [ ]
        resGreedy ← greedy_random(orden_candidato)
        candidatos ← candidatos ∪ {⟨resGreedy.first, orden_candidato⟩}

    hasta ← ⌈#candidatos/2⌉

    Para i ← 0..hasta−1:
        actual ← min(candidatos)
        orden_actual ← actual.second
        candidatos ← candidatos − min(candidatos)

        Si actual.first = ∞:
            Proximo i

            res ← local_dos_opt(orden_actual, limite) Si i es par
                  local_swap(orden_actual, limite) Sino

            Si res.first < mejorRes:
                mejorRes ← res.first
                orden ← orden_actual

    Retornar ⟨mejorRes, |orden|⟩
```

5.3. Con una única búsqueda local

Por último hicimos otra implementación que, sin recortar candidatos, emplea una única búsqueda local (es decir, no alternan como en las implementaciones que ya mostramos) y mejora cada instancia hasta encontrar un mínimo local.

La búsqueda local que elegimos fue $2op$ dado que, como se verá más adelante en su apartado experimental, tiene un comportamiento temporal mejor que $swap$ y, para casos grandes, tiene mayor precisión.

La motivación detrás de esta GRASP es, a costa de potencialmente empeorar el rendimiento temporal (ya hemos visto que cuando $corridas = 0$ para las búsquedas locales la performance es potencialmente factorial), mejorar la precisión dado que se mejoran todos los candidatos hasta que que no sea posible seguir.

Para cada instancia hacemos un llamado a la respectiva búsqueda local de la función con $corridas = 0$, recordando que este valor de parámetro seteaba la búsqueda para que realice iteraciones hasta no encontrar ningún vecino de la instancia final que presente menor distancia.

```
Def grasp_2opt(orden, i) → ⟨float, int⟩:
    mejorRes ← ∞

    n ← #gimnasios + #paradas
    cant_candidatos ← n^i

    Para i ← 0..cant_candidatos−1:
        resGreedy ← greedy_random(orden_actual)

        Si resGreedy.first = ∞:
            Proximo i

        res ← local_2opt(orden_actual, 0)

        Si res.first < mejorRes:
            mejorRes ← res.first
            orden ← orden_actual

    Retornar ⟨mejorRes, |orden|⟩
```

6. Experimentación

Para las diferentes mediciones necesarias para los experimentos presentados a continuación fueron realizadas sobre una CPU Intel 5200U en un sistema con 8GB de memoria RAM. Los binarios fueron compilados por `gcc 6.2.1 20160830` utilizando los flags `-O3 -std=c++11 -march=native`. El toolchain utilizado para correr las mediciones y realizar los gráficos puede encontrarse junto al código en `tools/data_analisis`.

Para las mediciones de tiempos se corre repetidamente el mismo problema no menos de 4 veces y 100ms para caso, y luego se almacena la media de las mediciones.

6.1. Runtime de los algoritmos exactos

6.1.1. En función de la cantidad de nodos

En la figura 2 comparamos el tiempo de ejecución del algoritmo exacto de fuerza bruta con el de backtracking. Para ello medimos con ambos 5 casos distintos para cada combinación de cantidad de gimnasios y paradas entre 2 y 5. Además usamos siempre un tamaño de mochila mayor a tres veces la cantidad de paradas.

Como podemos observar, ambos se comportan exponencialmente pero el backtracking resulta ser cerca de un orden de magnitud mas rápido.

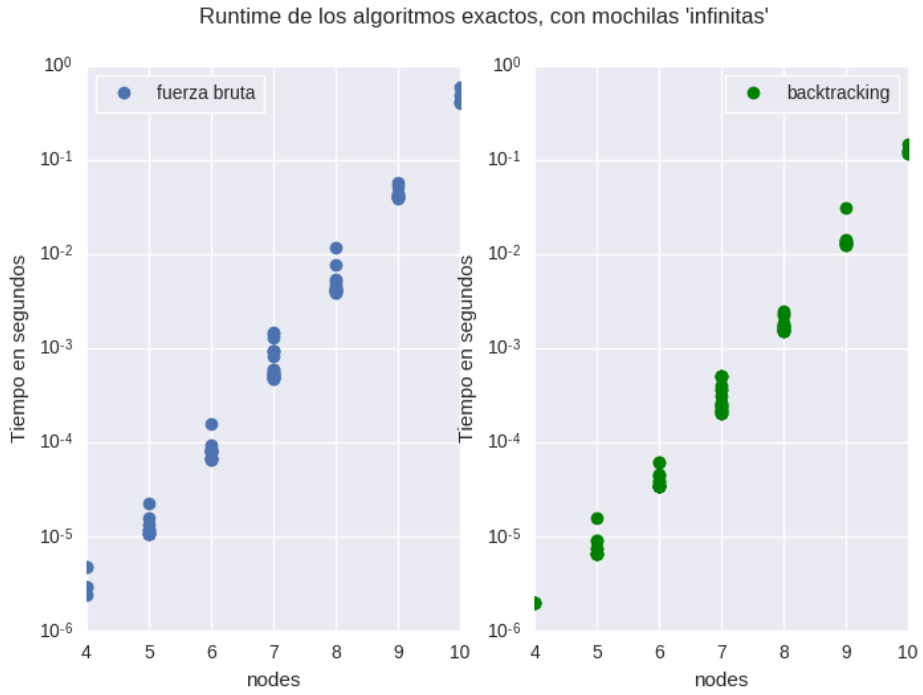

Figura 2: Tiempo de ejecución, en escala logarítmica, de los algoritmos exactos.

6.1.2. En función del tamaño de la mochila

A continuación realizamos pruebas fijando la cantidad de gimnasios y paradas en 5 y variando el tamaño de la mochila.

Vemos en la figura 3 que dentro del intervalo de tamaños de mochila interesantes, limitado por la cantidad de nodos que puede tener un problema a ser resuelto en un tiempo factible (ya que si el tamaño es mayor a tres veces la cantidad de paradas, no influirá en el comportamiento del algoritmo) la gran diferencia de rendimiento contra la implementación de fuerza bruta a favor del backtracking.

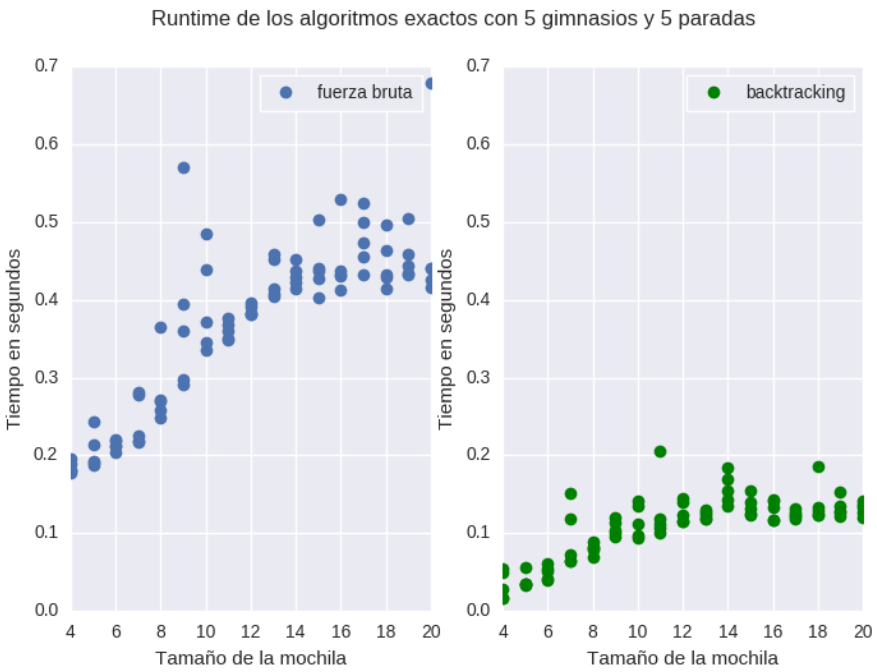

Figura 3: Tiempo de ejecución de los algoritmos exactos al variar el tamaño de la mochila

6.2. Runtime de las heurísticas greedy

6.2.1. En función de la cantidad de nodos

Medimos el tiempo en la figura 4 generando 5 casos para cada combinación de entre 1000 y 10000 gimnasios y paradas, avanzando de a 1000, y tamaño de mochila mayor a 30000.

Vemos en la figura 5 que estos tiempos tiempos se correlaciona fuertemente con un comportamiento cuadrático en función de la cantidad de nodos.

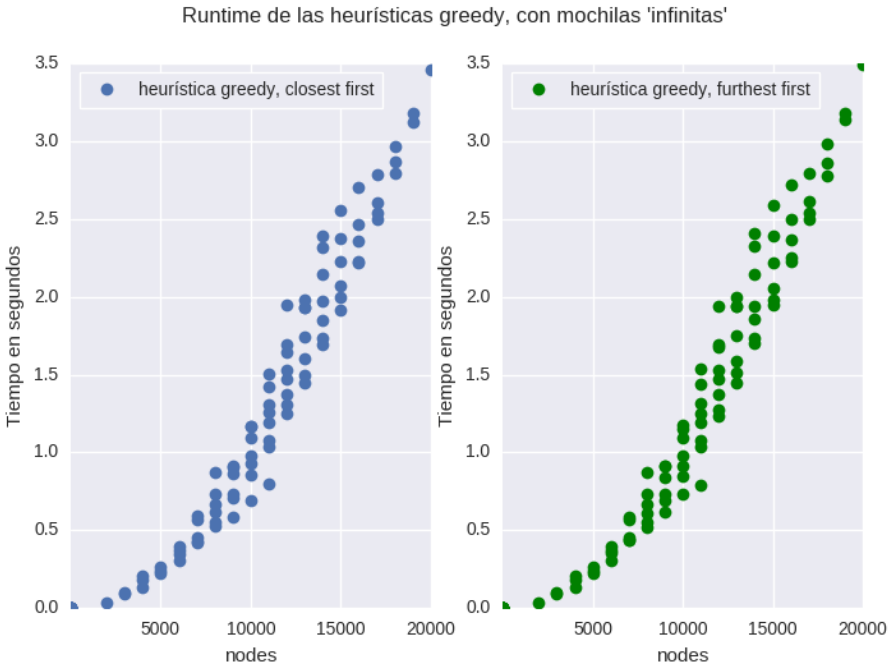

Figura 4: Tiempo de ejecución de las heurísticas greedy

Las implementaciones *Closest First* y *Furthest First* mostraron tiempos de ejecuciones notablemente similares, como se aprecia en la figura 4. Esto se debe a que el algoritmo es indentico pero modificando el tipo de comparación para elegir el próximo gimnasio (menor distancia por mayor distancia).

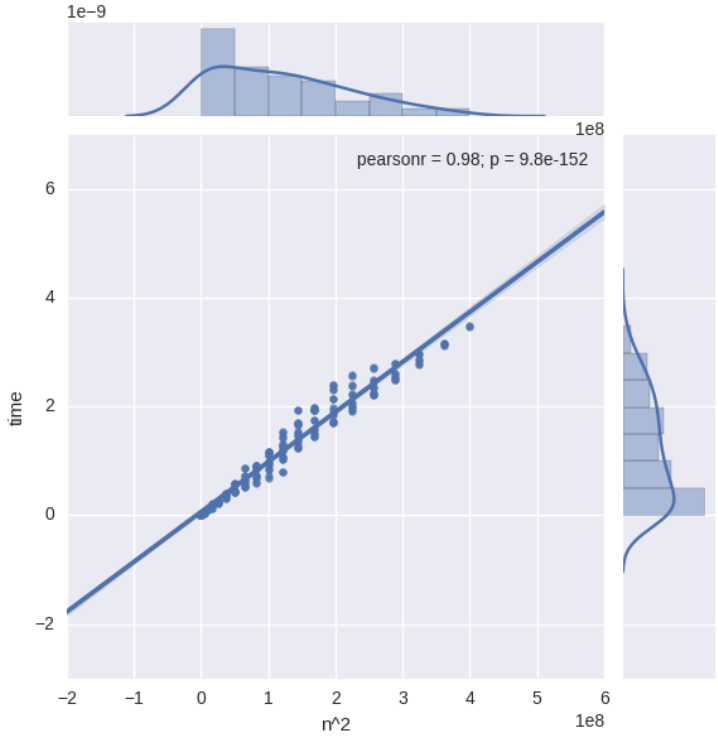

Figura 5: Correlación entre el tiempo de ejecución del greedy y una complejidad cuadrática

6.2.2. En función del tamaño de la mochila

Realizamos también pruebas fijando la cantidad de gimnasios y paradas en 2500 y variando el tamaño de la mochila entre 100 y 7500 en pasos de a 100, y nos encontramos con un comportamiento constante y con poco ruido para ambos tipos de greedy (con una correlación entre tiempo medido y tamaño de mochila de $0{,}170856$).

En el cuadro 2 se encuentra una descripción de los datos medidos.

count	76.000000
mean	0.215175
std	0.005658
min	0.209863
25 %	0.212328
50 %	0.213131
75 %	0.215283
max	0.239380

Cuadro 2: Descripción de las mediciones en función del tamaño de mochila

6.3. Runtime de las heurísticas locales

6.3.1. En función de la cantidad de nodos

Para las heurísticas locales medimos nuevamente los tiempos generando 5 casos para cada combinación de entre 10 y 100 gimnasios y paradas, avanzando de a 10, y tamaño de mochila mayor a 300. Los

resultados se pueden apreciar en la figura 6. Se pasa $corridas = 0$, por lo que se itera hasta encontrar un mínimo local.

Podemos corroborar en la figura 7 que ambas variaciones se corresponden con un comportamiento de orden cuarto en función de la cantidad de nodos.

Esto significaría que se hizo una cantidad lineal (también en función de la cantidad de nodos) de llamados a cada método de optimización dado que estos eran cúbicos en complejidad. Magnitud muy distinta a la cota factorial que supusimos en dicha sección, aún cuando considerábamos que era realmente muy poco probable que se tuviera un orden factorial de llamados antes de encontrar un mínimo local.

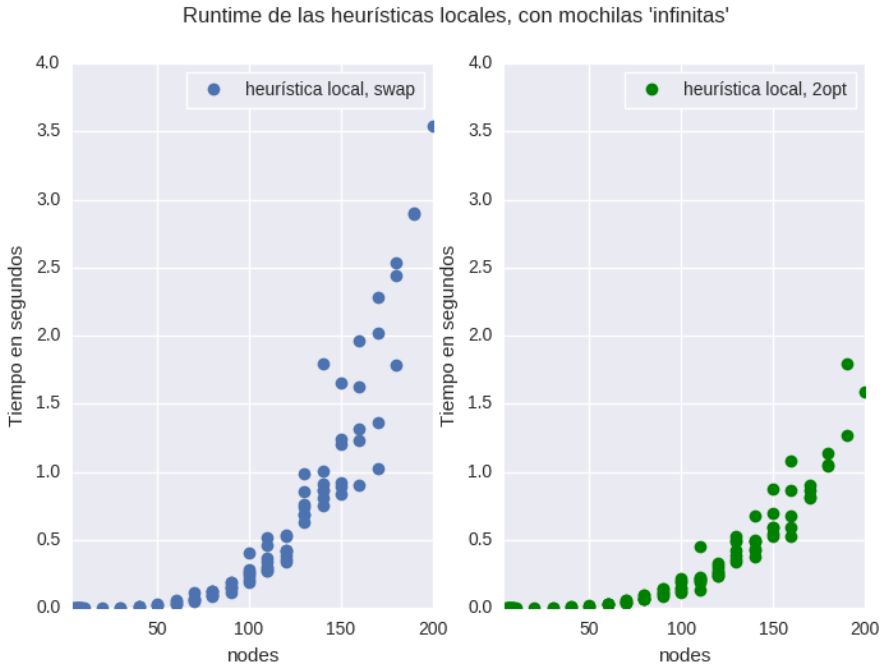

Figura 6: Tiempo de ejecución de las heurísticas locales

Sobre las instancias más grandes se observa una diferencia favorable hacia $2opt$ respecto de $swap$.

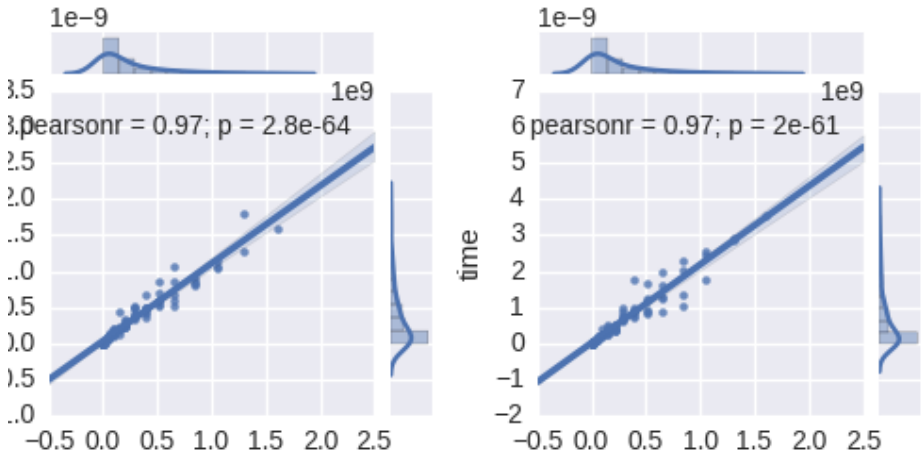

Figura 7: Correlación entre el tiempo de ejecución de las heurísticas locales 2opt (izquierda) y swap (derecha), con una complejidad n^4

El $swap$ que se tomó para comparar con $2opt$ es aquel que no tomaba el mínimo del vecindario de cada instancia sino que tomaba el primero que tuviera menor distancia a la actual (realizando el resto

de las iteraciones sobre el último mínimo encontrado). Como especulamos en su momento, al recorrer potencialmente más de un vecindario por iteración (aunque no de manera completa), su performance terminó siendo temporalmente mejor. Esto se ve en la figura 8 donde se observa un comportamiento de complejidad n^5, peor al n^4 apreciado en la figura 6 correspondiente al $swap$ que se queda con el primer vecino que mejore la distancia.

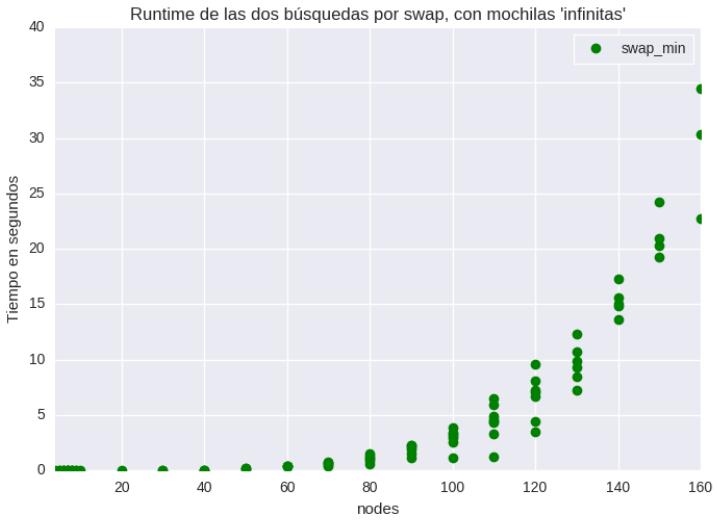

Figura 8: Tiempo de ejecución de la heurística swap tomando el mínimo de cada vecindario por iteración.

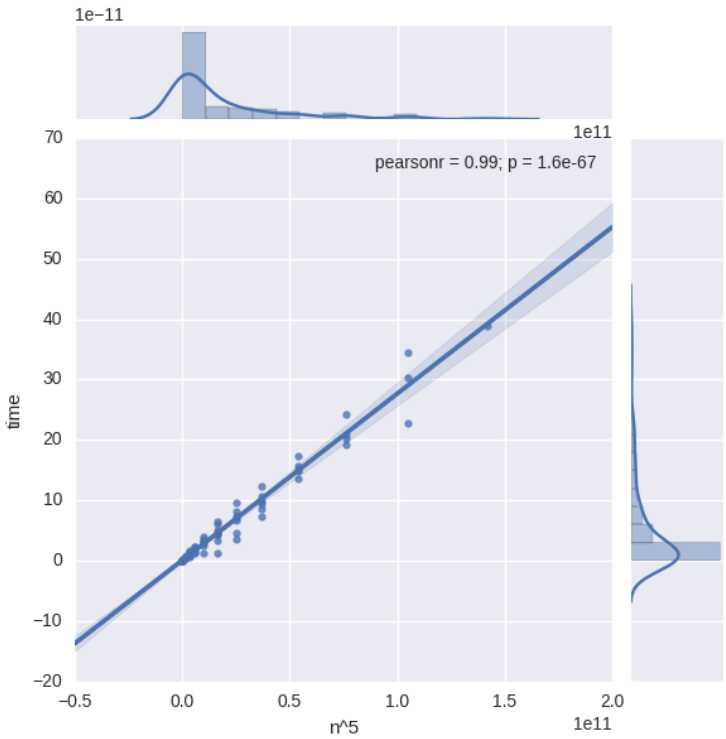

Figura 9: Correlación entre el tiempo de ejecución de la heurística swap tomando el mínimo de cada vecindario por iteración, con una complejidad n^5

Por lo tanto, utilizamos la otra implementación para el resto de los contrastes empíricos ya que tiene un rendimiento más parecido a la otra búsqueda local implementada, $2opt$, que tampoco tomaba el mínimo de cada vecindario.

6.3.2. En función del tamaño de la mochila

Nuevamente realizamos pruebas fijando la cantidad de gimnasios y paradas en 50 y variando el tamaño de la mochila entre 10 y 150, y nos encontramos, al igual que con el greedy, con un comportamiento constante en función de la mochila para ambas variaciones (con una correlación entre tiempo medido y tamaño de mochila de $0{,}259633$ para swap y de $0{,}265672$ para 2opt).

En el cuadro 3 se encuentra una descripción de los datos medidos.

	Swap	2opt
count	141.000000	141.000000
mean	0.263016	0.167931
std	0.080163	0.034468
min	0.124284	0.087849
25 %	0.214029	0.143264
50 %	0.253680	0.165566
75 %	0.297979	0.193758
max	0.566984	0.268200

Cuadro 3: Descripción de las mediciones en función del tamaño de mochila

6.4. Runtime de la metaheurística GRASP

6.4.1. En función de la cantidad de nodos

Para medir el comportamiento de GRASP generamos 4 instancias para cada combinación de gimnasios y mochilas entre 10 y 50, avanzando de a 5, con un tamaño de mochila superior a 150. Además, seteamos ambos exponentes del GRASP en 1.

En la figura 10 pueden apreciarse los resultados. Como se observa en la figura 11, el comportamiento está fuertemente correlacionado a un orden 5 sobre la cantidad de nodos.

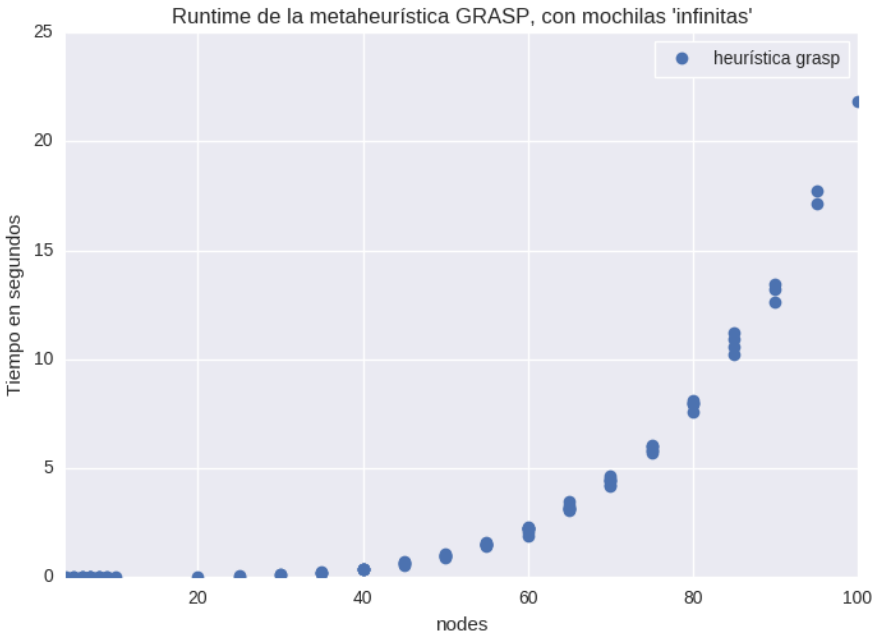

Figura 10: Tiempo de ejecución de la metaheurística grasp alternando búsquedas locales

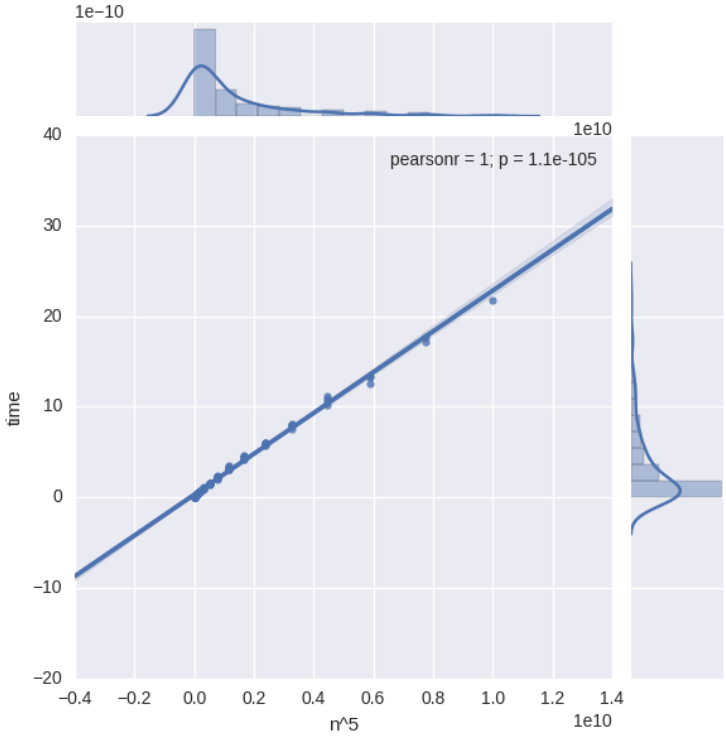

Figura 11: Correlación entre el tiempo de ejecución de grasp alternando búsquedas locales y una complejidad de orden 5 en función de la cantidad de nodos

6.4.2. En función del tamaño de la mochila

Para medir el tiempo de ejecución en función del tamaño de la mochila fijamos la cantidad de gimnasios y paradas en 20 y variamos la mochila entre 10 y 60. Como se aprecia en la figura 12, nos encontramos con que para tamaños pequeños se observa una correlación entre la mochila y el tiempo de ejecución, pero mas o menos a partir del tamaño 30 los tiempos se mantienen relativamente constantes.

Esto puede deberse a que cuando hay poco espacio el la mochila se limita la cantidad de opciones que puede tomar el algoritmo, lo que hace que termine mas temprano.

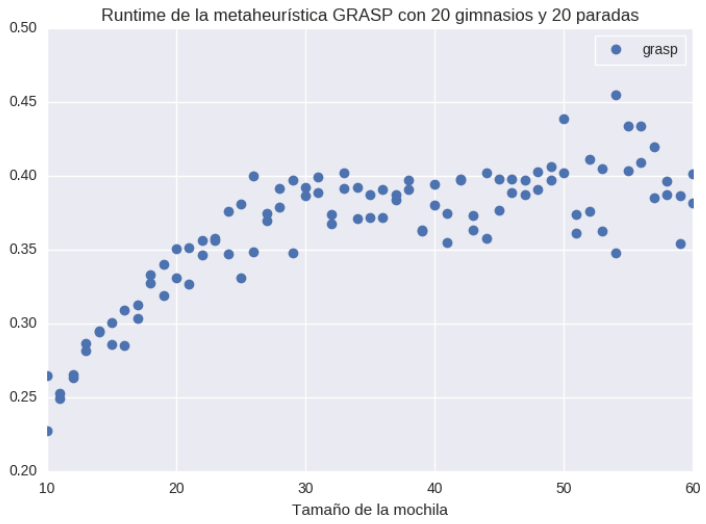

Figura 12: Correlación entre el tiempo de ejecución de grasp y una complejidad de orden 5 en función de la cantidad de nodos

6.5. Precisión de las heurísticas

Para medir que tan buenos son los resultados corrimos el mismo problema con cada heurística y comparamos con el resultado real, obteniendo la proporción entre la solución encontrada y la buscada. Como el nuestro es un problema de minimización, las proporciones siempre serán mayores o iguales a 1 (nuestras heurísticas siempre encuentran una solución posible).

Para testear problemas de mayor tamaño, tomamos como resultado de comparación el menor valor encontrado entre todas las heurísticas.

6.5.1. Precisión en casos pequeños, comparando con el exacto

Para cada combinación de gimnasios y paradas tal que su suma sea menor a 14 (y ambos ≥ 1) generamos 4 casos usando el generador `random`, uno con mochila de tamaño 3, otro con tamaño igual a la cantidad de paradas, otro con el doble de las paradas y uno con espacio igual a trues veces la cantidad de paradas. Resolvimos estos casos con cada uno de nuestros algoritmos y comparamos las soluciones.

En la figura 13 vemos las distribuciones de las precisiones para cada heurística.

Como era de esperar los algoritmos greedy suelen obtener resultados muy por encima del valor óptimo, aunque vemos que en el $75\,\%$ de los casos la variante closest first obtuvo un valor menor al doble del buscado. Las heurísticas locales, especialmente la variante swap, mejoraron bastante la precision (a costa de un tiempo de ejecución bastante mas elevado).

Y por último, el GRASP obtuvo el óptimo en un $82{,}3\,\%$ de los casos. En el cuadro 4 se describe la distribución de su precisión.

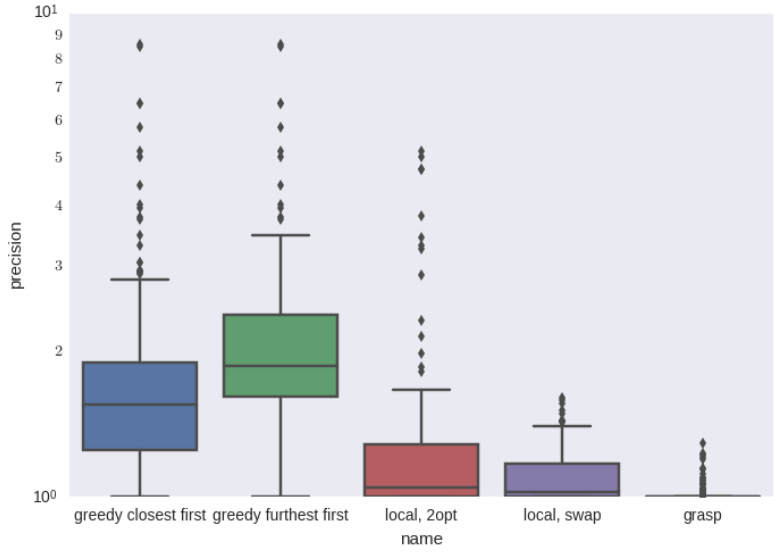

Figura 13: Precisión comparando con el valor mínimo conseguido mediante un algoritmo exacto.

Respecto de las heurísticas golosas, comparando a partir la figura 13 podemos ver una ventaja, acorde quizás a las expectativas detrás de la intuición de cada heurística, para la implementación *Clostest First* respecto de *Furthest First*.

En el caso de las heurísticas locales, $2op$ presenta una varianza muchísimo mayor que $swap$ a pesar de que, por debajo de sus terceros cuartiles, las precisión es apenas favorable para esta última.

count	226.000000
mean	1.011650
std	0.039716
min	1.000000
25 %	1.000000
50 %	1.000000
75 %	1.000000
max	1.288691

Cuadro 4: Descripción de la precisión de la metaheurística GRASP

6.5.2. Precisión en casos grandes, comparando con el mínimo

A continuación generamos casos de prueba utilizando el generador `random` para todas las combinaciones de paradas y gimnasios tal que ambos sean mayor o iguales a 10 y su suma sea a lo sumo 40. En cada caso variamos el tamaño de la mochila de igual forma que en la sección 6.5.1.

Nuevamente observamos como la variante ***Closest First*** del greedy produce mejores resultados que la otra. Lo cual, considerando tanto las diferencias de runtimes y precisión para casos chicos, la posiciona como la de mejor rendimiento de las dos implementaciones.

Además notamos que, si bien la heurística local swap no suele producir resultados muy malos, su media es bastante peor que la variante 2opt, cuyos valores se describen en el cuadro 5. Considerando que, si bien para casos pequeños mejoraba la precisión $swap$, tanto para instancias grandes como en runtime hay clara ventaja a favor de ***2-opt***.

Por último, la metaheurística grasp obtiene el mejor resultado en el $96{,}72\,\%$ de los casos. Este número no es $100\,\%$ ya que existen casos donde los locales obtienen un buen resultado a partir del greedy closest first pero el grasp, por la característica aleatoria de su ejecución, termina en una solución peor.

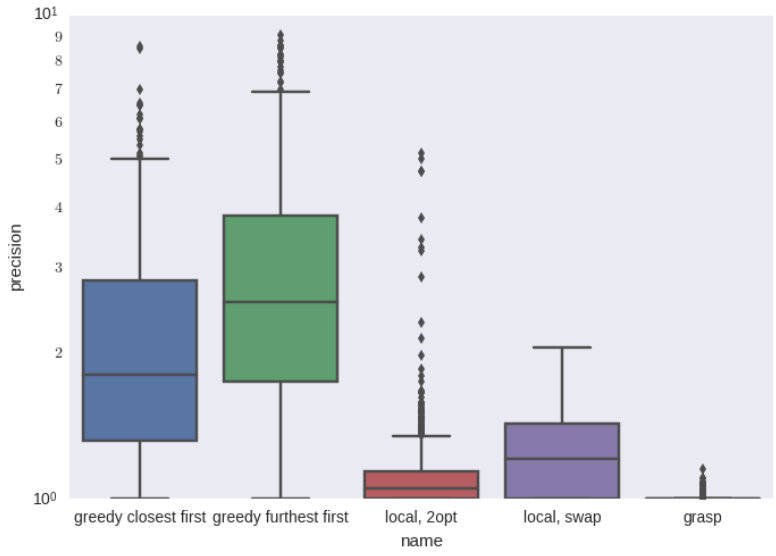

Figura 14: Precisión comparando con el valor exacto

count	2008.000000
mean	1.111849
std	0.247017
min	1.000000
25 %	1.000000
50 %	1.049093
75 %	1.139941
max	5.181492

Cuadro 5: Descripción de la precisión de la heurística local 2opt

6.5.3. Precisión en función del generador

Utilizando las mismas variables descriptas en la sección 6.5.2 generamos instancias con cada uno de los generadores disponibles, para ver si la precisión variaba con los tipos de problema.

El resultado puede verse en la figura 15. Para las instancias de `zigzag` se suele conseguir buenos resultados en comparación con las `random`. Ademas es notable que las instancias de `separated` se suelen resolver relativamente bien con las heurísticas locales y GRASP, no así con los greedy.

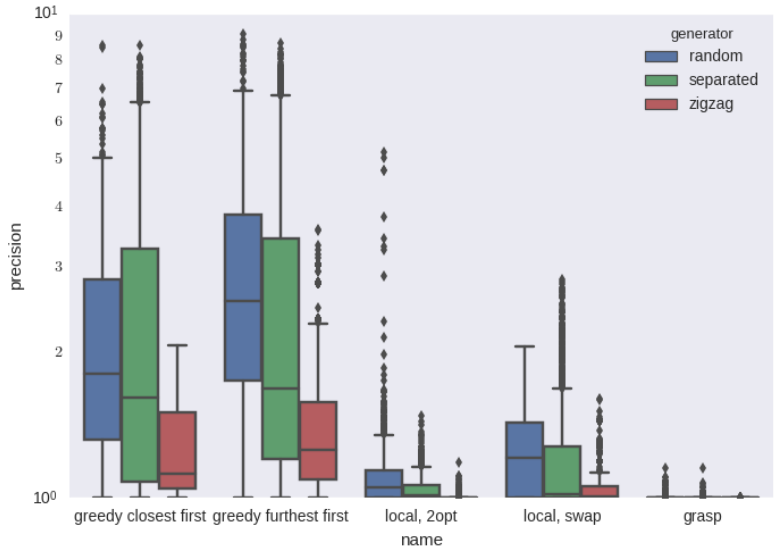

Figura 15: Precisión comparando con el mínimo, según el generador utilizado

6.5.4. Precisión en función del tamaño de la mochila

Quisimos también probar si el tamaño de las mochilas influía en la precisión de cada método. Para ello tomamos los resultados obtenidos en la sección 6.5.2 y los separamos por el tamaño de la mochila relativo a la cantidad de paradas.

El resultado, que puede observarse en la figura 16, muestra una leve correlación para las heurísticas greedy. Las heurísticas locales y GRASP, en cambio, no parecen verse afectadas.

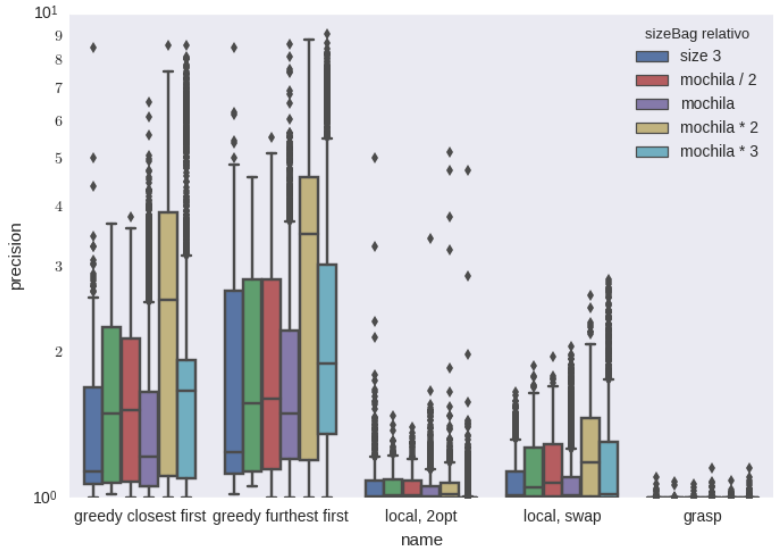

Figura 16: Precisión comparando con el mínimo, según el tamaño relativo de la mochila

6.6. Variación de las variables del GRASP

Cuando presentamos la metaheurística GRASP describimos dos variables que recibe como parámetro. Por un lado la que llamaremos $inicios$, que se usa como exponente para calcular la cantidad de puntos de partida aleatorios a utilizar. Y la otra, que llamaremos $limite$, que se usa como exponente para limitar la cantidad de pasos a dar por las búsquedas locales.

Para estudiar el comportamiento de la metaheurística al variar ambos exponentes generamos tres instancias del problema con 25 gimnasios, 25 paradas, y mochila de tamaño 100. Luego corrimos el GRASP variando cada exponente entre $0{,}2$ y $1{,}7$, en pasos de a $0{,}1$ y obtuvimos los valores medios de tiempo y precisión (comparando con el mejor resultado) entre las tres instancias.

En la figura 17 pueden apreciarse los tiempos obtenidos. Se observa un claro comportamiento exponencial en función de la cantidad de $inicios$. Por otro lado, encontramos que a partir de un $limite$ igual a 0.5 el tiempo deja de aumentar para este ya que probablemente casi todas las búsquedas locales terminen en los pasos anteriores.

Al observar los cambios de precisión en la figura 18 vemos que nuevamente los resultados se mantienen constantes al aumentar el $limite$ mas allá de 0.5. Vemos en cambio que los resultados van descendiendo siempre que aumentamos la cantidad de inicios.

Comparando ambos gráficos se aprecia que los valores 1 y 1 utilizados como exponentes de $inicios$ y $limite$ en las mediciones de la sección 6.4 brindan un buen tradeoff entre tiempo y precisión. Aunque si uno no buscara resultados tan exactos, utilizar $0{,}4$ para ambos exponentes reduciría el tiempo de corrida en un $94\,\%$ a costa de obtener un resultado un $6\,\%$ peor.

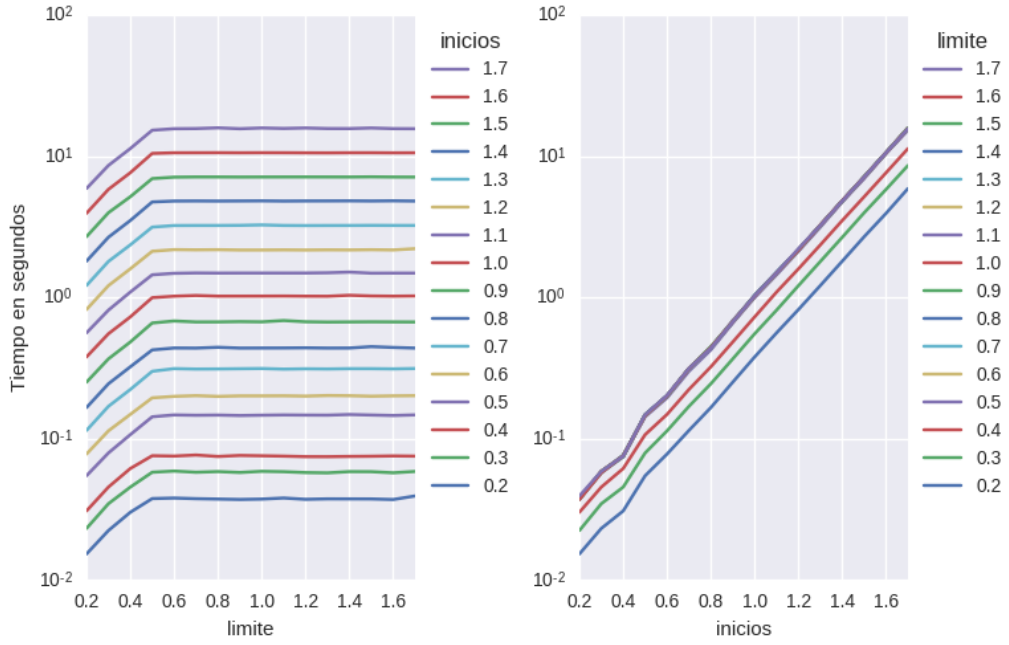

Figura 17: Tiempo de corrida medio de GRASP al variar sus exponentes, sobre tres problemas con 50 nodos

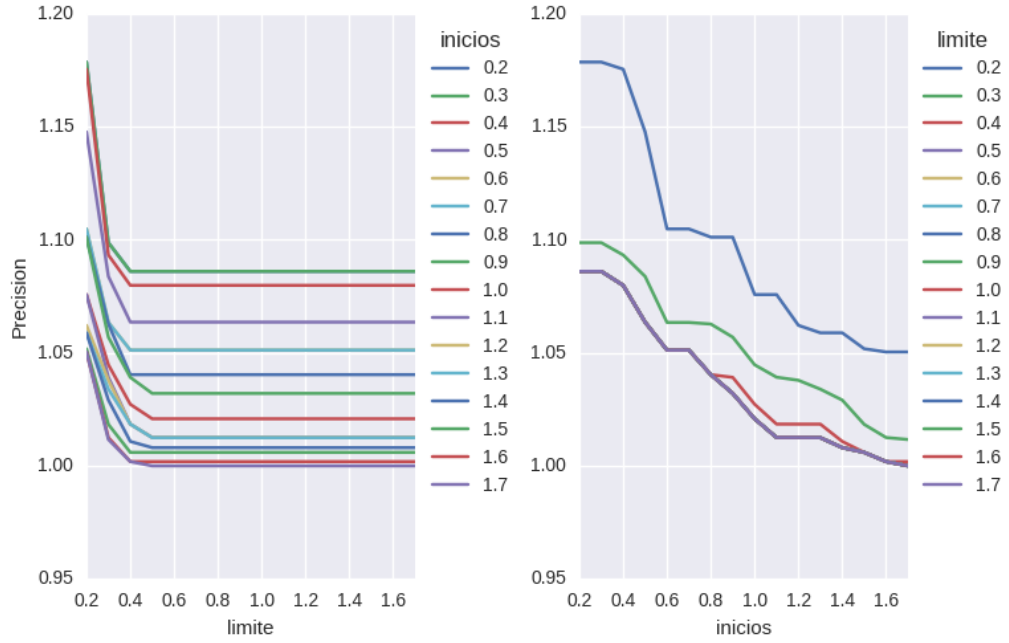

Figura 18: Precisión media de GRASP al variar sus exponentes, sobre tres problemas con 50 nodos