Metaheurísticas

Práctica 1.b – Algoritmos GRASP e ILS

Grupo 3 – Martes 12:30-14:30

43180612K - Abad Vich, David
DAV00004@RED.UJAEN.ES
GRADO EN INGENIERÍA INFORMÁTICA Curso 2014/2015

Índice

1 CONTENIDOS DEL ARCHIVO

2 Descripción del problema 2
3 Aplicación de los algoritmos 3
4 Algoritmos utilizados 5
4.1 Búsqueda local 5
4.2 GRASP 7
4.3 ILS 9
5 Algoritmo de comparación – Greedy 10
6 Procedimiento considerado para el desarrollo de la práctica 11
6.1 Pequeño manual de ususario 11
7 Experimentos y análisis de resultados 12
7.1 Resultados Greedy 13
7.2 Resultados Búsqueda Local 14
7.3 Resultados GRASP 15
7.4 Resultados ILS 16
7.5 Resultados Globales 17
8 Bibliografía 19

2 DESCRIPCIÓN DEL PROBLEMA

De forma general, se trata de encontrar una solución al problema de asignación cuadrática (del inglés, Quadratic Assignation Problem o QAP).

Para este problema, disponemos de *n* unidades y localizaciones, de modo que debemos asignar, de manera óptima, las unidades a las localizaciones. Para ello se dispone de un método para evaluar cómo de óptima es una solución y es mediante el flujo que hay entre diferentes unidades y la distancia que hay entre las localizaciones.

Sin embargo la descripción de encontrar una solución es bastante genérica. Debido a la cantidad de tiempo que nos puede llevar el conseguir la solución más óptima, dependiendo del tamaño del problema, en ocasiones nos tendremos que conformar con encontrar una solución, óptima por supuesto, pero no la más óptima que exista.

Entonces, el coste de una solución se puede formular como:

$$\pi \in \prod_{N}^{min} \left(\sum_{i=1}^{n} \sum_{j=1}^{n} f_{ij} \cdot d_{\pi(i)\pi(j)} \right)$$

donde:

- π es una permutación solución al problema, que representa la asignación de la unidad *i* a la localización π(i).
- f_{ij} es el flujo que hay de la unidad *i* a la *j*.
- d_{kl} es la distancia existente entre la localización *k* y la *l*.

Para este problema se usarán 20 casos diferentes, seleccionados de varios conjuntos de instancias disponibles en la QAPLIB (la biblioteca de problemas de asignación cuadrática) donde se puede encontrar además una descripción de lo que representa el problema.

3 APLICACIÓN DE LOS ALGORITMOS

Al aplicar los algoritmos al problema, hay que tener que tienen en común varios aspectos:

- Representación del problema: El problema se representa como dos matrices, una que contiene los flujos y otra las distancias, de manera que se puede ver el flujo que hay entre dos unidades y la distancia entre dos localizaciones de forma inmediata. Hay que tener en cuenta que mientras que la distancia entre dos localizaciones es la misma en ambos sentidos, puede que el flujo sea distinto entre dos unidades, dependiendo del sentido.

- Representación de las soluciones: Hemos mencionado antes la representación para la solución y es utilizando una permutación de *n* elementos (siendo *n* el tamaño del problema) a modo de que cada posición de la permutación representa una unidad y cada elemento de la permutación representa su localización asignada:

$$solución: \begin{matrix} unidad \\ localización \end{matrix} \begin{pmatrix} 1\ 2 \cdots n \\ i\ \ j\ \cdots k \end{pmatrix}\ i,j,k \in \{1,\dots,n\}$$

- Función objetivo: La función que se va a optimizar, es la suma del producto entre el flujo de dos unidades y la distancia de sus correspondientes localizaciones asignadas. Su pseudocódigo[1] es:

```
coste ← 0
tamaño ← tamaño(solución)
Para i ← 0 Hasta tamaño-1 Hacer
        Para j ← 0 Hasta tamaño-1 Hacer
                coste ← distancia[solución[i]][solución[j]] * flujo[i][j]
        Fin_Para
Fin_Para
```

- Generación de vecino: Para generar el entorno, se generarán los vecinos mediante el intercambio de dos elementos de la permutación solución.

- Función de factorización: Para un cálculo eficiente del coste de una solución, se extraerá del coste de la solución original los costes relacionados con las dos posiciones que se van a intercambiar para generar esa solución.

- Criterio de parada: Tras evaluar 10000 soluciones distintas, se detendrá la ejecución del algoritmo.

- Generación de la solución inicial aleatoria: En ambos algoritmos se utiliza la misma función para crear una solución inicial aleatoria de la que partir. El pseudocódigo es:

[1] Nota de programación: Hay que tener en cuenta que en el código los vectores y arrays empiezan desde la posición 0, por lo que queda representado en el pseudocódigo.

```
Función generarSolucionAleatoria() Devuelve un vector con la permutación
        tamaño ← tamaño del problema
        Para i ← 0 Hasta tamaño Hacer
                encontrada ← falso
                Mientras (no encontrada) Hacer
                        crear localización aleatoria
                        Para j ← 0 Hasta tamaño Hacer
                                Si (localización está en el vector) Entonces
                                        encontrada ← false
                                Fin_Si
                        Fin_Para
                        vector[i] ← localización
                Fin_Mientras
        Fin_para
        devuelve vector
Fin_Función
```

4 ALGORITMOS UTILIZADOS

4.1 BÚSQUEDA LOCAL

Generamos una solución inicial aleatoria y exploramos el entorno, siguiendo la técnica del primer mejor, por lo que en cuanto encontramos una solución mejor, nos movemos a ella. El entorno se recorre además de forma secuencial con la ayuda de la máscara *don't look bits*, por lo que se exploran sólo las unidades que están activas. Pseudocódigo:

```
Procedimiento buscar()
        Generar solución inicial aleatoria
        tamaño ← tamaño(solución)
        Crear e inicializar array don't look bits dlb
        i ← 0
        mejora ← true
        Mientras (mejora AND soluciones_comprobadas < 10000) Hacer
                j ← i+1
                solución encontrada ← false
                Si (dlb[i] != 0) Entonces
                        //Si el elemento está desactivado, no entra en el siguiente bucle
                        encontrada ← true
                Fin_Si
                Mientras (j < tamaño AND no encontrada) Hacer
                        soluciones_comprobadas++
                        coste ← calcular el coste de cambiar las posiciones i y j
                        Si (coste < coste de la solución) Entonces
                                Se activa el elemento con dlb[i] ← 0
                                Se cambia la solución actual por la encontrada
                                encontrada ← true
                                i ← 0
                        Fin_Si
                        j++
                Fin_Mientras
                Si (no encontrada) Entonces
                        Se desactiva el elemento con dlb[i] ← 1
                Fin_Si
                Si (i == tamaño - 1) Entonces
                        //Si ya ha recorrido todos los elementos, es que no hay mejora
                        mejora ← false
                Fin_Si
                i++
        Fin_Mientras
Fin_Procedimiento
```

En la búsqueda local, además de parar tras comprobar 10000 soluciones, también hay otro criterio de parada y es cuando no hay posibilidad de mejora en el entorno.

No habrá posibilidad de mejora cuando ya se han comprobado todos los cambios posibles, por lo que también la máscara se encontrará a 1 en todos sus bits (todos activados ya que no ha habido mejora para ninguno de los intercambios posibles).

4.2 GRASP

Generando una solución inicial Greedy aleatoria, se realiza una búsqueda local sobre la misma. Se repite este procedimiento hasta llegar al criterio de parada, que en este caso es evaluar 25 soluciones Greedy aleatorias (con su correspondiente búsqueda local).

El algoritmo BL utilizado es el antes descrito, de primer mejor, ya que no sólo conlleva un menor coste computacional (que en este problema no es algo por lo que haya que preocuparse) sino que además, tiene más probabilidades de evitar caer en óptimos locales[2].

Descripción del esquema de búsqueda:

```
Procedimiento buscar()
        mejor_solución ← Generar solución Greedy aleatoria
        Búsqueda local (mejor_solución)
        Para i←1 Hasta 25 Hacer
                solución ← Generar solución Greedy aleatoria
                Búsqueda local (solución)
                Si coste(solución) mejor que coste(mejor_solución) Entonces
                        mejor_solución ← solución
                Fin_Si
        Fin_Para
Fin_Procedimiento
```

El esquema de búsqueda es bastante sencillo, se genera una primera solución que se tomará como base para comparar con el resto, a partir de ahí se repite 24 veces (de forma que junto con la inicial, se evalúan un total de 25 soluciones) y se compara con la inicial, de manera que si se encuentra una mejor, se sustituye.

La parte principal del algoritmo la constituye la generación de la solución Greedy aleatoria (o probabilística), que sigue las transparencias del seminario 2b (dos etapas).

Una vez generada la lista de candidatos, la lista restringida de candidatos se calcula a partir del umbral. Después de una asignación hay que recalcular la lista de candidatos, ya que los costes se ven afectados por las soluciones que ya han sido asignadas (siguiendo la función de selección de la segunda etapa).

[2] Siguiendo la publicación [1] de la bibliografía.

```
Función greedyAleatorio() Devuelve solución
        Inicializar vectores booleanos de elementos disponibles
        Calcular costes siguiendo la función de selección
        Obtener costes máximo y mínimo
        Calcular umbral_flujos y umbral_distancias
        Crear listas restringidas
        Escoger dos candidatos de forma aleatoria
        Calcular lista candidatos (LC)
        Repetir Hasta candidatos = 1
                Calcular umbral LC
                Crear lista restringida (LRC)
                Escoger un elemento aleatorio de la LRC
                Añadir el elemento al vector solución
                Actualizar vectores de elementos disponibles
                Calcular lista candidatos
        Hasta candidatos = 1
        Añadir el último candidato a la solución

        Devolver solución
```

Los vectores booleanos controlan los elementos, tanto unidades como localizaciones, que están disponibles y no han sido asignados aún. Esto sirve para el proceso de creación de la lista de candidatos, una función a parte:

```
Función generarListaCandidatos (vector booleano flujos_disponibles, loc_disponibles, vector
solución_parcial) Devuelve vector candidatos
        Para i ← 0 Hasta n Hacer
                Si (flujos_disponibles[i] == true) Entonces
                        Para j ← 0 Hasta n Hacer
                                Si (loc_disponibles[j] == true) Entonces
                                        coste ← 0
                                        Para k ← 0 Hasta n Hacer
                                                Si (solución_parcial[k] == -1) Entonces
                                                        coste ← flujo[i][k] *distancia[j][sol_par[k]]
                                                Fin_Si
                                        Fin_Para
                                        añade candidato (i, j, coste)
                                Fin_Si
                        Fin_Para
                Fin_Si
        Fin_Para
        Devuelve candidatos
Fin_Funcion
```

4.3 ILS

Se crea una solución inicial aleatoria, se aplica la búsqueda local y a partir de ahí se repite el proceso de aplicar una mutación a la mejor solución encontrada y comprobar si se obtiene mejor resultado. En caso positivo, se cambia la mejor solución, en caso negativo se aplica otra mutración a la mejor solución.

El parámetro de parada es aplicar 25 búsquedas locales y la mutación consiste en barajar los elementos de una sublista de tamaño *n*/4 (siendo *n* el tamaño del problema).

```
Procedimiento buscar()
        mejor_solución ← generaSolucionAleatoria()
        mejor_coste ← coste(mejor_solución)
        búsqueda_local(mejor_solución)
        tamaño_mutación ← tamaño(mejor_solución)/4
        solución ← mejor_solución
        Para i ← 1 Hasta 25 Hacer
                Coger elemento aleatorio entre 0 y (tamaño(mejor_solución) – tamaño_mutación)
                Barajar solución desde elemento hasta elemento + tamaño_mutacion
                búsqueda_local(solución)
                coste ← coste(solución)
                Si (coste < mejor_coste) Entonces
                        mejor_coste ← coste
                        mejor_solución ← solución
                Si_No
                        coste ← mejor_coste
                        solución ← mejor_solución
                Fin_Si
        Fin_Para
Fin_Procedimiento
```

La función usada para barajar la solución es una función estándar incluida en la librería *<algorithm>* de *C++*, llamada *random_shuffle*[3]. Su funcionamiento interno consiste en intercambiar, empezando desde el último elemento indicado hasta el primero, los elementos de forma aleatoria.

```
Procedimiento random_shuffle (inicio, fin)
        n ← fin – inicio
        Para i ← n-1 Hasta 0 Decrementando i Hacer
                intercambiar(elemento[i], elemento[aleatorio entre i+1 y fin])
        Fin_Para
Fin_Procedimiento
```

El algoritmo de búsqueda local utilizado es el mismo realizado en la práctica anterior (aquí incluído anteriormente).

[3] Referencia: http://www.cplusplus.com/reference/algorithm/random_shuffle/

5 ALGORITMO DE COMPARACIÓN – GREEDY

Este algoritmo está basado en la búsqueda del primer mejor. Consiste en realizar el sumatorio de cada fila de ambas matrices (flujos y distancias) y asignar a los mayores flujos, las menores distancias.

Es un algoritmo de bajo coste computacional, pues genera una solución (que variará según se asignen las localizaciones a las unidades) que normalmente no llegará a ser un óptimo, pero es probable que se encuentre cerca de uno, por lo que puede tener una gran aplicación si se utiliza para generar soluciones iniciales al problema, de las cuales explorar el entorno para encontrar una mejor solución.

```
Procedimiento CalcularSolucion()
        tamaño ← tamaño(distancias)
        Para i ← 0 Hasta tamaño-1 Hacer
                Para j ← 0 Hasta tamaño-1 Hacer
                        suma_distancias[i] ← suma_distancias[i] + distancias[i][j]
                        suma_flujos[i] ← suma_flujos[i] + flujos[i][j]
                Fin_Para
        Fin_Para
        Para i ← 0 Hasta tamaño-1 Hacer
                Para j ← 0 Hasta tamaño-1 Hacer
                        Si (suma_flujos[i] > max) Entonces
                                Coge nuevo flujo máximo
                        Fin_Si
                        Si (suma_distancias[i] < min AND suma_distancias[i] >= 0) Entonces
                                Coge nueva  distancia mínima
                        Fin_Si
                Fin_Para
                solución[flujo máximo] = distancia mínima
                suma_flujos[flujo máximo] = -1
                suma_distancias[distancia mínima] = -1
        Fin_Para
Fin  Procedimiento
```

Para generar la solución simplemente se suman los flujos y distancias y se asigna al mayor flujo, la menor distancia.

Para evitar volver a coger un elemento que ya ha sido asignado, se le asigna un valor negativo que pasará desapercibido cuando se busquen los siguientes valores máximo y mínimo.

6 PROCEDIMIENTO CONSIDERADO PARA EL DESARROLLO DE LA PRÁCTICA

Para el desarrollo de la práctica se ha utilizado código desarrollado por el propio alumno, con la ayuda de las transparencias de la asignatura y diferentes consultas al profesor.

Además, se han incorporado opciones extra para la práctica que pueden ayudar en el análisis de los resultados, tales como número de iteraciones o la posibilidad de usar la solución del algoritmo Greedy como solución inicial para ambas búsquedas.

6.1 PEQUEÑO MANUAL DE USUSARIO

La ejecución del programa es sencilla:

1. Ejecutar el archivo Meta1a.exe situado en la carpeta Deployment.
2. Elegir el archivo a ejecutar.
3. Introducir los diferentes parámetros (opcional, se puede dejar por defecto):
    a. Matriz que se lee primero (por defecto la de flujos).
    b. Valor de la semilla para la generación de números aleatoria (por defecto 123456).
    c. (Opcional) Usar la solución Greedy como solución inicial de la búsqueda local.
    d. (Opcional) Elegir el número de soluciones a comprobar para la búsqueda local.
4. Hacer clic en ejecutar.

El programa incluye una pequeña barra de progreso, introducido de forma manual, para mostrar que la ejecución sigue activa.

También cabe la posibilidad de ejecutar el programa usando Qt, para ello, una vez instalado Qt, hay que abrir el proyecto que desde la pantalla de bienvenida de Qt, es haciendo clic en la opción Open Project, seleccionando la carpeta donde se encuentra el proyecto y seleccionando el archivo .pro que hay ahí.

Una vez abierto, construir y ejecutar. Saldrá entonces la misma interfaz que con el ejecutable y ya hay que seguir los mismos pasos que con el mismo.

NOTA: Es posible que con archivos de gran tamaño (como lipa90) el programa parece no responder, pero acaba la ejecución transcurridos unos segundos.

7 EXPERIMENTOS Y ANÁLISIS DE RESULTADOS

Para todos los problemas se ha considerado siempre el mismo valor para la semilla: 123456.

Pasamos a comentar las instancias usadas en esta práctica[4], todas partes de la QAPLIB:

- bur*: Tenemos una matriz que contiene el tiempo medio de escritura de una máquina de escribir y otra con la frecuencia de los pares de letras en diferentes idiomas.
- chr*: Nos dan una matriz de adyacencia de un árbol balanceado y otra matriz con grafos completos.
- els19: Las distancias entre 19 instalaciones diferentes de un hospital y el flujo de pacientes entre ellas. Hay que notar que ésta es de las instancias que tendrá que leerse primero la matriz de distancias.
- esc32a: Ejemplos de una aplicación, usados para circuitos secuenciales auto-comprobables.
- kra32: Contienen datos reales usados para planear el Hospital Regensburg en Alemania.
- lipa90a: Esta instancia proviene de generadores de problemas, que proveen de instancias asimétricas con soluciones óptimas conocidas.
- nug25: La matriz de distancias contiene las distancias de cuadrículas rectangulares de Manhattan. También hay que leer primero la matriz de distancias en esta instancia.
- sko*: Las distancias de éstos problemas son rectangulares y los flujos son número pseudoaleatorios. Otra más para la que se lee primero la matriz de distancias.
- tai*: Instancias uniformemente generadas. Asimétricas y aleatorias, suceden en la generación de patrones de grises.
- tho40: Las distancias de este problema son rectangulares, por lo que la primera matriz que se lee es la de distancias.

[4] En caso de no indicar nada, la matriz que se ha de leer primero es la de flujos.

7.1 RESULTADOS GREEDY

Algoritmo Greedy			
Caso	**Coste obtenido**	**Desv**	**Tiempo**
Els19	38627698	124,42	0,00
Chr20a	8632	293,80	0,00
Chr25a	17556	362,49	0,00
Nug25	4462	19,18	0,00
Bur26a	5968117	9,98	0,00
Bur26b	4273373	11,93	0,00
Tai30a	2098700	15,43	0,00
Tai30b	1387185541	117,73	0,00
Esc32a	342	163,08	0,00
Kra32	117130	32,05	0,00
Tai35a	2952758	21,91	0,00
Tai35b	506552163	78,79	0,00
Tho40	312578	29,96	0,00
Tai40a	3730054	18,82	0,00
Sko42	18902	19,54	0,00
Sko49	27378	17,07	0,00
Tai50a	5804880	17,54	1,00
Tai50b	788404422	71,83	0,00
Tai60a	8345630	15,82	1,00
Lipa90a	367797	1,99	1,00

7.2 RESULTADOS BÚSQUEDA LOCAL

Algoritmo BL			
Caso	**Coste obtenido**	**Desv**	**Tiempo**
Els19	23734354	37,89	1,00
Chr20a	3912	78,47	0,00
Chr25a	8028	111,49	2,00
Nug25	3962	5,82	1,00
Bur26a	5444522	0,33	2,00
Bur26b	3851727	0,89	2,00
Tai30a	1916134	5,39	2,00
Tai30b	691972163	8,61	3,00
Esc32a	170	30,77	3,00
Kra32	98630	11,20	5,00
Tai35a	2585098	6,73	5,00
Tai35b	353867912	24,90	5,00
Tho40	264608	10,02	9,00
Tai40a	3339928	6,39	8,00
Sko42	16234	2,67	11,00
Sko49	24972	6,78	22,00
Tai50a	5265454	6,61	19,00
Tai50b	487830146	6,32	34,00
Tai60a	7717196	7,09	37,00
Lipa90a	363582	0,82	227,00

7.3 RESULTADOS GRASP

Algoritmo GRASP			
Caso	**Coste obtenido**	**Desv**	**Tiempo**
Els19	19968092	16,01	20,00
Chr20a	2900	32,30	15,00
Chr25a	5920	55,95	37,00
Nug25	3880	3,63	36,00
Bur26a	5442765	0,30	64,00
Bur26b	3828169	0,27	64,00
Tai30a	1916702	5,42	66,00
Tai30b	661387279	3,81	79,00
Esc32a	160	23,08	91,00
Kra32	94500	6,54	119,00
Tai35a	2548854	5,24	187,00
Tai35b	292699737	3,31	169,00
Tho40	251190	4,44	292,00
Tai40a	3318068	5,69	234,00
Sko42	16272	2,91	392,00
Sko49	24440	4,51	571,00
Tai50a	5178322	4,85	616,00
Tai50b	477862516	4,15	886,00
Tai60a	7606744	5,56	1063,00
Lipa90a	363482	0,79	5965,00

7.4 RESULTADOS ILS

Algoritmo ILS			
Caso	**Coste obtenido**	**Desv**	**Tiempo**
Els19	20508428	19,15	11,00
Chr20a	2872	31,02	10,00
Chr25a	5260	38,57	23,00
Nug25	3818	1,98	25,00
Bur26a	5432068	0,10	29,00
Bur26b	3826062	0,22	30,00
Tai30a	1895080	4,23	48,00
Tai30b	683045735	7,21	44,00
Esc32a	152	16,92	62,00
Kra32	95170	7,29	75,00
Tai35a	2525558	4,28	132,00
Tai35b	306755533	8,27	89,00
Tho40	245686	2,15	164,00
Tai40a	3305390	5,29	150,00
Sko42	16160	2,20	232,00
Sko49	23874	2,09	336,00
Tai50a	5125506	3,78	387,00
Tai50b	478785392	4,35	365,00
Tai60a	7534924	4,57	747,00
Lipa90a	363328	0,75	3773,00

7.5 RESULTADOS GLOBALES

Algoritmo	*Desviación*	*Tiempo (en ms.)*
Greedy	72.17	0.15
BL	18.46	19.90
GRASP	9.44	548.30
ILS	8.22	336.60

En la tabla con los resultados globales se ve más claro la mejora de las búsquedas iterativas. Aunque ya el tiempo de ejecución empieza a notarse para plantearse el uso de un algoritmo u otro.

Las soluciones del algoritmo Greedy no han sido tan buenas como para decidir quedarse con ellas sin recurrir a un algoritmo de búsqueda.

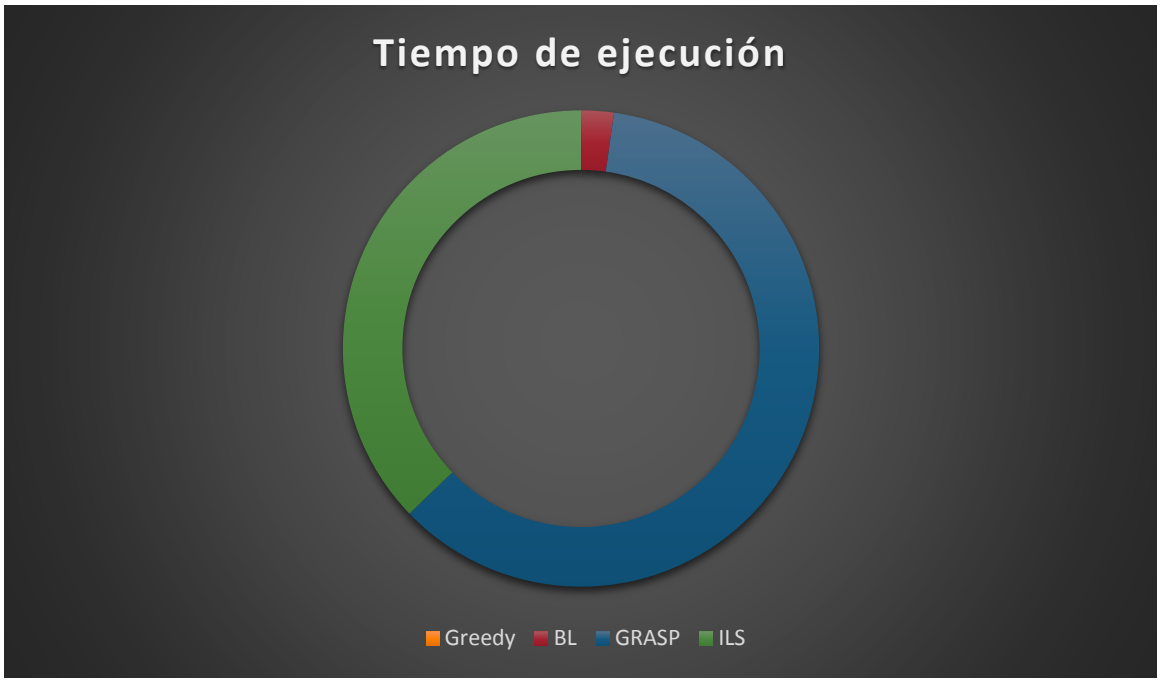
En otros problemas con mayor necesidad de recursos, es interesante tomar la decisión de usar un algoritmo más sencillo como la búsqueda local o usar uno más complejo, tomando el riesgo de llevar mayor tiempo de ejecución. Hay que tener en mente también las necesidades que se tienen y es que si se necesita una solución rápida, con el simple requisito de obtener una buena solución, podemos contar con la búsqueda local, mientras que en casos con mayores recursos y menor necesidad de una solución rápida, las búsquedas iterativas dan mejor resultado.

Dentro de las búsquedas iterativas, el algoritmo ILS ha dado mejores resultados con un menor coste en tiempo y es que la aplicación de una pequeña mutación no se puede comparar a la generación de una solución mediante un Greedy aleatorio.

Aun usando la solución Greedy como solución inicial para la búsqueda local, no se consiguen mejores resultados ya que consigue estancar la búsqueda antes de que le dé tiempo a explorar.

Cambiando el tamaño de la mutación en el ILS se obtienen diferentes resultados, para algunos ficheros mejora tener un mayor tamaño, mientras que para otros mejora tener un menor tamaño, por lo que se deduce que depende más del problema que del algoritmo implementado.


Desviación
80
70
60
50
40
30
20
10
0
Greedy
BL
GRASP
ILS


Tiempo de ejecución
Greedy
BL
GRASP
ILS

8 BIBLIOGRAFÍA

1. G. Ochoa, S. Verel y M. Tomassini. First-improvement vs. Best-improvement Local Optima Networks of NK Landscapes. *11th International Conference on Parallel Problem Solving From Nature*, Krakow, Poland, 2010.