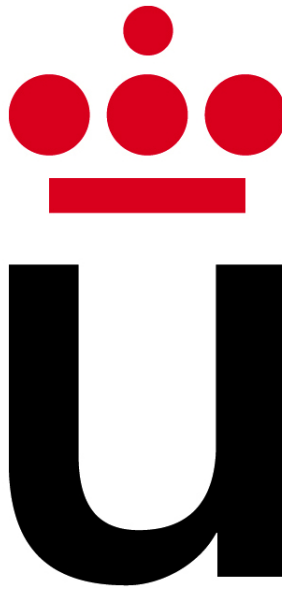


PRÁCTICA II
Simulación y Metaheurísticas
BÚSQUEDAS LOCALES

José Ignacio Escibano



MÓSTOLES, 14 DE MAYO DE 2016

Índice de tablas

1. Comparativa de resultados 4

Índice de figuras

Índice

1. Introducción	1
2. Resolución de la práctica	1
2.1. Definir una búsqueda local	1
2.2. Programación de los dos algoritmos	3
2.3. Análisis de los resultados	4
3. Conclusiones	5

1. Introducción

En esta práctica realizaremos búsquedas locales para el problema del p-hub.

2. Resolución de la práctica

A continuación, resolveremos todas las cuestiones planteadas de la práctica.

2.1. Definir una búsqueda local

El código para realizar una búsqueda local a partir de su vecindad y el tipo de orden (lexicográfico o aleatorio) se puede ver a continuación (función `busquedaLocal` de la clase `Utils`):

```
1 public static Solución busquedaLocal(String tipo_orden, List<Solución>
  ↳ vecindades, Solución actual) {
2     Solución mejor_solución = actual;
3
4     if (tipo_orden == "lexicográfico") {
5         for (int i = 0; i < vecindades.size(); i++) {
6             if (vecindades.get(i).getObjetivo() <
  ↳ mejor_solución.getObjetivo()) {
7                 mejor_solución = vecindades.get(i);
8             }
9         }
10    } else if (tipo_orden == "aleatorio") {
11        // Generamos un número aleatorio entre 0 y número de vecindades
12
13        // Copiamos la lista con las vecindades
14        List<Solución> vecindades_copia = new
  ↳ ArrayList<Solución>(vecindades);
15
16        int n = vecindades_copia.size();
17
18        Solución s = null;
19
20        while (n > 1) {
21            n = vecindades_copia.size();
22            Random r = new Random();
23            int indice = r.nextInt(n);
24            s = vecindades_copia.get(indice);
25
26            if (s.getObjetivo() < mejor_solución.getObjetivo()) {
```

```

27         mejor_solución = s;
28     }
29
30     // Eliminamos de la vecindad
31     vecindades_copia.remove(s);
32 }
33
34 }
35 return mejor_solución;
36 }

```

En las líneas 4 a 9 se realiza la búsqueda por orden lexicográfico y de las líneas 10 a 32 se realiza por orden aleatorio.

En el caso de la búsqueda lexicográfica se comparan las soluciones de la vecindad con la mejor solución actual, se selecciona ésta como mejor solución actual y se continúa el proceso hasta que se acaban las soluciones de las vecindad.

Notar que seguimos el orden que viene en la lista de soluciones vecinas, puesto que éstas vienen en ese orden.

En el caso del orden aleatorio, se genera un número aleatorio en el intervalo discreto [0, número de elementos vecindad) y se selecciona el índice que corresponde con el número aleatorio, se compara con la mejor solución actual y se elimina de la vecindad.

El código para obtener la vecindad a partir de una solución y su instancia está en la función `generarVecindad` de la clase `PHub`. Esta función busca un cliente y un servidor, intercambia sus papeles y genera una nueva matriz de adyacencia con esa configuración.

```

1  static List<Solución> generarVecindad(Solución s, InstanciaPHub
   ↪ instancia) {
2      List<Solución> vecindad = new ArrayList<>();
3      boolean[] sol = s.getSolucion();
4      boolean[] sol2 = new boolean[sol.length];
5
6      int nodos = sol.length;
7
8      // Generamos las permutaciones
9      for (int i = 0; i < sol.length; i++) {
10         for (int j = 0; j < sol.length; j++) {
11             // Buscamos un true y un falso entre un servidor y un
   ↪ cliente
12             if (sol[i] == true && sol[j] == false) {
13                 // Copiamos el array sol
14                 System.arraycopy(sol, 0, sol2, 0, sol.length);
15
16                 // Intercambiamos las posiciones

```

```

17         sol2[i] = false;
18         sol2[j] = true;
19
20         // Creamos la matriz de adyacencia de la nueva solución
21         boolean[][] ady = new boolean[nodos][nodos];
22         for (int z = 0; z < nodos; z++) {
23             if (!sol2[z]) {
24                 // Seleccionamos el servidor más cercano que nos
25                 // encontremos
26                 int serv = Utils.seleccionarServidor(z, sol2,
↪ instancia.getDistancia());
27
28                 ady[z][serv] = true;
29                 ady[serv][z] = true;
30             }
31
32         }
33
34         Solución s1 = new Solución(sol2, ady,
↪ instancia.getDistancia());
35
36         if (Utils.esSoluciónVálida(s1, instancia)) {
37             vecindad.add(s1);
38         }
39     }
40 }
41 }
42 return vecindad;
43 }

```

En las líneas 9 a 12 se busca un servidor y un cliente y se intercambian sus posiciones. De la línea 21 a la 32, se genera la matriz de adyacencia de la misma forma que para generar una solución aleatoria. Se crea la solución y se comprueba si es válida. En caso afirmativo se añade a la lista de vecinos. Por último, se devuelve la lista de vecinos.

2.2. Programación de los dos algoritmos

Elegimos la construcción aleatoria y mejora con orden de exploración lexicográfico (A2) y orden de exploración aleatorio (A3). El código de estos dos algoritmos se implementa en la clase `Práctica2`.

```

1 public static Solución constAleatoriaYMejoraLexicográfica(Solución sol,
↪ InstanciaPHub instancia){
2

```

```

3      Solución result = null;
4      List<Solución> vecindad = PHub.generarVecindad(sol, instancia);
5      result = Utils.búsquedaLocal("lexicográfico", vecindad, sol);
6      return result;
7  }
8
9  public static Solución constAleatoriaYMejoraAleatorio(Solución sol,
10     ↪ InstanciaPHub instancia){
11
12      Solución result = null;
13      List<Solución> vecindad = PHub.generarVecindad(sol, instancia);
14      result = Utils.búsquedaLocal("aleatorio", vecindad, sol);
15      return result;
16  }

```

En ambos casos se genera la vecindad a partir de la solución que se pasa como parámetro, se genera la vecindad, y se realiza la búsqueda según el tipo (lexicográfico o aleatorio).

2.3. Análisis de los resultados

En la Tabla 1 se muestran los resultados de la ejecución de los algoritmos, tanto por tipo de búsqueda (best o first improvement) y por tipo de orden (aleatorio y lexicográfico). El detalle completo de las soluciones puede verse en el fichero `busqueda_local.txt` que se encuentra en el mismo directorio que esta memoria.

Tabla 1: Comparativa de resultados

Instancia	Best improvement				First improvement			
	Orden lexicográfico		Orden aleatorio		Orden lexicográfico		Orden aleatorio	
	Función objetivo	Servidores	Función objetivo	Servidores	Función objetivo	Servidores	Función objetivo	Servidores
1	839.262	[5, 30, 39, 43, 50]	839.262	[5, 30, 39, 43, 50]	940.312	[5, 30, 39, 43, 50]	900.653	[5, 30, 39, 43, 50]
2	842.792	[10, 26, 29, 40, 50]	842.792	[10, 26, 29, 40, 50]	878.318	[10, 26, 29, 40, 50]	870.425	[10, 26, 29, 40, 50]
3	814.225	[10, 26, 31, 38, 50]	814.225	[10, 26, 31, 38, 50]	940.431	[10, 26, 31, 38, 50]	926.313	[10, 26, 31, 38, 50]
4	951.370	[1, 8, 16, 18, 50]	951.370	[1, 8, 16, 18, 50]	1187.107	[1, 8, 16, 18, 50]	974.978	[1, 8, 16, 18, 50]
5	683.575	[9, 13, 22, 27, 50]	683.575	[9, 13, 22, 27, 50]	716.234	[9, 13, 22, 27, 50]	683.659	[9, 13, 22, 27, 50]
6	732.835	[13, 19, 20, 23, 50]	732.835	[13, 19, 20, 23, 50]	870.756	[13, 19, 20, 23, 50]	807.549	[13, 19, 20, 23, 50]
7	872.203	[6, 9, 20, 46, 50]	872.203	[6, 9, 20, 46, 50]	913.547	[6, 9, 20, 46, 50]	937.001	[6, 9, 20, 46, 50]
8	861.647	[14, 16, 34, 36, 50]	861.647	[14, 16, 34, 36, 50]	968.366	[14, 16, 34, 36, 50]	909.729	[14, 16, 34, 36, 50]
9	859.900	[13, 14, 17, 23, 50]	859.900	[13, 14, 17, 23, 50]	1083.282	[13, 14, 17, 23, 50]	1083.282	[13, 14, 17, 23, 50]
10	793.365	[8, 14, 21, 28, 50]	793.365	[8, 14, 21, 28, 50]	970.672	[8, 14, 21, 28, 50]	924.900	[8, 14, 21, 28, 50]

Se puede observar que el valor de la función objetivo de la búsqueda best improvement siempre es el mismo, tanto si se hace un orden aleatorio como lexicográfico. Esto es debido a que se explora toda la vecindad y se elige la mejor solución, por lo que el orden no importa. Por el contrario, con la búsqueda first improvement, éste orden sí importa y, en este caso, se obtienen mejor resultados con la búsqueda aleatoria.

En cuanto a la medida del valor `dev` hemos obtenido en la búsqueda best improvement un valor de 0.001363, mientras que en la búsqueda first improvement con orden aleatorio

hemos obtenido $6.527 \cdot 10^{-4}$ y con orden lexicográfico $2.792 \cdot 10^{-4}$.

Por tanto, el mejor valor de `dev` se ha obtenido con la búsqueda first improvement con orden lexicográfico.

3. Conclusiones

En este caso práctico hemos visto cómo implementar búsquedas locales (best y first improvement). Además, hemos visto que el orden de exploración de la vecindad puede suponer grandes diferencias de la función objetivo. En nuestro caso hemos visto que el orden aleatorio produce mejores resultados que el lexicográfico.