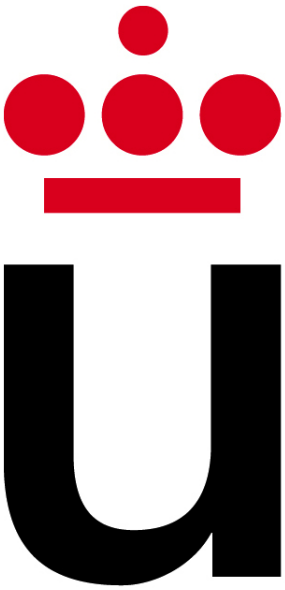
Práctica II

Simulación y Metaheurísticas

Búsquedas locales

José Ignacio Escribano



Móstoles, 14 de mayo de 2016

Índice de tablas

Índice de figuras

Índice

1. Introducción

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

2. Resolución de la práctica

A continuación, resolveremos todos 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 $[0, \text{número de elementos vecindad})$ 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`. El código del algoritmo A2 se muestra a continuación:

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

```
3       Solución mejor = sol;
4       long inicio = System.currentTimeMillis();
5       long fin = inicio + tiempo;
6       List<Solución> vecindad = PHub.generarVecindad(sol, instancia);
7
8       boolean noEsMejor = true;
9       long actual;
10      while ((actual = System.currentTimeMillis()) <= fin) {
11          Solución result = Utils.busquedaLocal("lexicográfico", vecindad,
    ↪ mejor);
12          if (result.getObjetivo() < mejor.getObjetivo()) {
13              mejor = result;
14              vecindad = PHub.generarVecindad(mejor, instancia);
15          } else {
16              noEsMejor = true;
17          }
18      }
19      return mejor;
20  }
```

El algoritmo genera la vecindad a partir de la solución que se pasa como parámetro, se comprueba si la solución es mejor, y si lo es se guarda como mejor solución y se genera su vecindad. Este proceso sigue hasta que finaliza el tiempo que se le pasa a la función como parámetro.

El código del algoritmo A3 se idéntico al anterior salvo en la línea 11, donde se cambia el orden lexicográfico por 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.

Se puede observar que el valor de la función objetivo de la búsqueda best improvement siempre es el mismo, tanto si se hace con 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 0.00232, mientras que en la búsqueda first improvement con orden aleatorio

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	952.877	[24, 28, 31, 37, 49]	952.877	[24, 28, 31, 37, 49]	953.776	[24, 28, 31, 37, 49]	952.877	[24, 28, 31, 37, 49]
2	917.405	[7, 23, 33, 40, 50]	917.405	[7, 23, 33, 40, 50]	927.784	[7, 23, 33, 40, 50]	919.901	[7, 23, 33, 40, 50]
3	987.576	[10, 18, 34, 40, 49]	987.576	[10, 18, 34, 40, 49]	987.576	[10, 18, 34, 40, 49]	987.576	[10, 18, 34, 40, 49]
4	889.546	[4, 11, 16, 42, 50]	889.546	[4, 11, 16, 42, 50]	909.138	[4, 11, 16, 42, 50]	911.626	[4, 11, 16, 42, 50]
5	777.419	[7, 11, 16, 22, 50]	777.419	[7, 11, 16, 22, 50]	800.094	[7, 11, 16, 22, 50]	790.070	[7, 11, 16, 22, 50]
6	714.157	[4, 14, 18, 27, 49]	714.157	[4, 14, 18, 27, 49]	717.988	[4, 14, 18, 27, 49]	717.988	[4, 14, 18, 27, 49]
7	863.546	[4, 11, 14, 38, 49]	863.546	[4, 11, 14, 38, 49]	961.614	[4, 11, 14, 38, 49]	863.546	[4, 11, 14, 38, 49]
8	873.628	[19, 22, 37, 38, 50]	873.628	[19, 22, 37, 38, 50]	873.628	[19, 22, 37, 38, 50]	874.776	[19, 22, 37, 38, 50]
9	791.706	[2, 4, 13, 24, 50]	791.706	[2, 4, 13, 24, 50]	791.923	[2, 4, 13, 24, 50]	797.231	[2, 4, 13, 24, 50]
10	827.279	[2, 20, 32, 33, 50]	827.279	[2, 20, 32, 33, 50]	837.145	[2, 20, 32, 33, 50]	805.421	[2, 20, 32, 33, 50]

hemos obtenido 0.00221 y con orden lexicográfico 0.00230.

Por tanto, el mejor valor de `dev` se ha obtenido con la búsqueda first improvement con orden aleatorio, aunque con muy poca diferencia.

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.