

Metaheurística - Práctica 1.a

Técnicas de Búsqueda Local y Algoritmos
Greedy para el Problema de la Asignación
Cuadrática

3º Grado Ingeniería Informática, Grupo 3 (Miércoles)

Salvador Corts Sánchez, 75935233C

salvacorts@correo.ugr.es

Índice

1. Descripción del problema	3
2. Consideraciones comunes a los algoritmos utilizados	4
3. Algoritmo Greedy	6
4. Algoritmo de Búsqueda Local	7
5. Procedimiento considerado para desarrollar la práctica	10
6. Experimentos y análisis de resultados	12

1. Descripción del problema

El problema de asignación cuadrática (en inglés, quadratic assignment problem, QAP) es uno de los problemas de optimización combinatoria más conocidos. En él se dispone de n unidades y n localizaciones en las que situarlas, por lo que el problema consiste en encontrar la asignación óptima de cada unidad a una localización. La nomenclatura “cuadrático” proviene de la función objetivo que mide la bondad de una asignación, la cual considera el producto de dos términos, la distancia entre cada par de localizaciones y el flujo que circula entre cada par de unidades. El QAP se puede formular como:

$$QAP = \min_{\pi \in \Pi_N} \left(\sum_{i=1}^n \sum_{j=1}^n f_{ij} d_{\pi(i)\pi(j)} \right)$$

donde:

- π es una solución al problema que consiste en una permutación que representa la asignación de la unidad i a la localización $\pi(i)$.
- f_{ij} es el flujo que circula entre la unidad i y la j .
- d_{kl} es la distancia existente entre la localización k y la l .

2. Consideraciones comunes a los algoritmos utilizados

Esta práctica ha sido diseñada como una librería de metaheurísticas per se. Es decir, existe un tipo de objeto **Solution** y un tipo de objeto **Solver** del cual heredarán los objetos que implementan las diversas metaheurísticas. Cada metaheurística deberá implementar la función *Solve* que devuelve un objeto **Solution**.

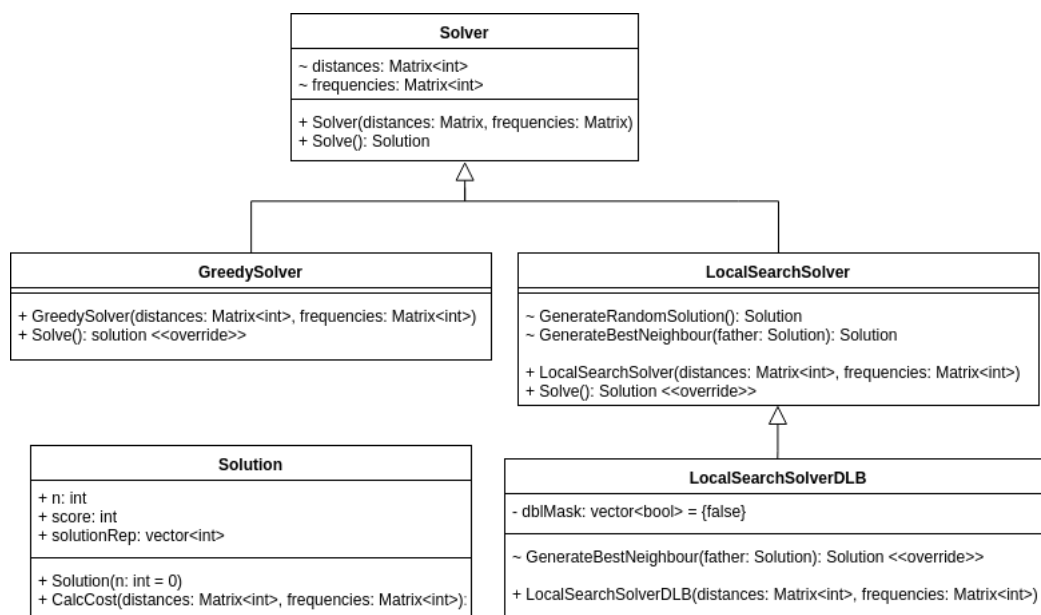


Figura 1: Diagrama de clases

Clase Solver

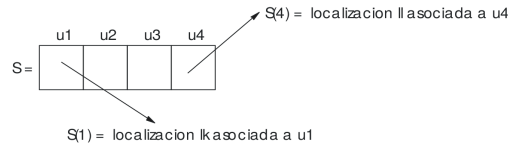
Esta clase debe ser heredada por las metaheurísticas a implementar. Su representación consta de dos matrices:

- **Distancias:** Matriz de distancias entre un punto i y otro j .
- **Frecuencias:** Matriz de flujo entre un objeto i y otro j .

Tiene una función virtual llamada *Solve* que ha de ser implementada por los objetos que hereden de **Solver**. Es la interfaz común a todos los objetos de tipo Solver para obtener una Solución.

Clase Solution

Sirve para representar una solución, la cual, se implementa como un vector donde cada posición i representa un objeto y alberga la localización j donde debe ser colocado dicho objeto i .



Existe una función *CalcCost* que calcula el coste de dicha solución como:

$$cost = \sum_{i=1}^n \sum_{j=1, j \neq i}^n f_{ij} d_{\pi(i)\pi(j)}$$

donde:

- π es la solución al problema.
- f_{ij} es el flujo que circula entre la unidad i y la j .
- d_{kl} es la distancia existente entre la localización k y la l .

Dado que el cálculo del coste de la solución es bastante costoso, $O(n^2)$, Esta función debe llamarse manualmente al menos una vez para obtener el coste y que este se guarde en la representación de la clase.

3. Algoritmo Greedy

Se basa en el cálculo de los potenciales de flujo y distancia definidos como:

$$\hat{f}_i = \sum_{j=1}^n f_{ij} \quad \hat{d}_i = \sum_{j=1}^n d_{ij}$$

El algoritmo irá seleccionando la unidad i libre con mayor \hat{f}_i y le asignará la localización j libre con menor \hat{d}_j . Su implementación en pseudocódigo es la siguiente:

```

1  # Calcula los potenciales
2   $dp = fp = \text{vector}(n)$ 
3  for  $i = 0$  to  $n$  do
4       $\hat{f}_i = \hat{d}_i = 0$ 
5      for  $j = 0$  to  $n$  do
6           $\hat{f}_i = \hat{f}_i + f_{ij}$ 
7           $\hat{d}_i = \hat{d}_i + d_{ij}$ 
8      end
9       $dp_i = \hat{d}_i$ 
10      $fp_i = \hat{f}_i$ 
11 end

12 # Calcula la mejor combinación.  $\pi$  es la representación de la solución
13  $locAssigned = unitAssigned = \text{vector}(n)\{0\}$ 
14 for  $i = 0$  to  $n$  do
15      $best\hat{f} = -\infty$ ;  $best\hat{f}_{index} = 0$ 
16      $best\hat{d} = \infty$ ;  $best\hat{d}_{index} = 0$ 
17     for  $j = 0$  to  $n$  do
18          $\hat{f}_i = fp_j$ ;  $\hat{d}_i = dp_j$ 
19         if  $\hat{f}_i > best\hat{f}$  and  $unitAssigned_j \neq 1$  then
20              $best\hat{f} = \hat{f}_i$ ;  $best\hat{f}_{index} = j$ 
21         end
22         if  $\hat{d}_i < best\hat{d}$  and  $locAssigned_j \neq 1$  then
23              $best\hat{d} = \hat{d}_i$ ;  $best\hat{d}_{index} = j$ 
24         end
25     end
26      $\pi(best\hat{f}_{index}) = best\hat{d}_{index}$ 
27      $unitAssigned_{best\hat{f}_{index}} = locAssigned_{best\hat{d}_{index}} = 1$ 
28 end

```

4. Algoritmo de Búsqueda Local

Vamos a utilizar una **búsqueda local del primer mejor**. Cuando se genera una solución vecina que mejora a la actual, se toma esta como solución y se pasa a la siguiente iteración. Se detiene la búsqueda cuando no se genera ningún vecino mejor que la solución actual. La implementación de dicha idea, que será la función *Solve*, se puede ver como:

```

1  $\pi = \text{GenerateInitialSolution}()$  # Será aleatoria
2 do
3    $\pi' = \text{GenerateBestNeighbour}(\pi)$ 
4   if  $\exists \pi'$  then  $\pi = \pi'$ ;
5 while  $\exists \pi'$ ;

```

A fin de minimizar el riesgo de quedarnos en un óptimo local, vamos a partir de una solución aleatoria en vez de partir de una solución greedy. Dicha solución aleatoria se genera de la siguiente manera:

```

1  $assigned = \text{vector}(n)0$ 
2 for  $i = 0$  to  $n$  do
3   do
4      $r = \text{random}() \bmod n$ 
5     while  $assigned_r \neq 0$ ;
6      $\pi(i) = r$ 
7      $assigned_r = 1$ 
8 end

```

Como se comentó anteriormente, el proceso de cálculo del coste de la solución es de orden cuadrático por lo que realizar dicho calculo con cada vecino es sumamente costoso; En su lugar, vamos a considerar una **factorización** (con eficiencia $O(n)$) teniendo en cuenta solo los cambios realizados por el movimiento de intercambio para generar el vecino. El incremento del coste de cambiar el elemento en la posición r por el de s se define como:

$$\Delta C(\pi, r, s) = \sum_{k=1, k \neq r, s}^n \left[f_{rk}(d_{\pi(s)\pi(k)} - d_{\pi(r)\pi(k)}) + f_{sk}(d_{\pi(r)\pi(k)} - d_{\pi(s)\pi(k)}) + \right. \\ \left. f_{kr}(d_{\pi(k)\pi(s)} - d_{\pi(k)\pi(r)}) + f_{ks}(d_{\pi(k)\pi(r)} - d_{\pi(k)\pi(s)}) \right]$$

Si $\Delta C(\pi, r, s) < 0$, el resultado de cambiar r por s es favorable, es decir, el costo es menor por lo que tomaremos el vecino resultante de este cambio como solución actual y generamos nuevos vecinos a partir de este.

La función que hace uso de esta factorización para explorar los vecinos de una solución se implementaría como:

```
1 def GenerateBestNeighbour( $\pi$ ):  
2   for  $r = 0$  to  $n/2$  do  
3     for  $s = r + 1$  to  $n$  do  
4       if  $\Delta C(\pi, r, s) < 0$  then  
5          $\pi' = \pi$   
6          $t = \pi'(r)$   
7          $\pi'(r) = \pi'(s)$   
8          $\pi'(s) = t$   
9         return  $\pi'$   
10      end  
11    end  
12  end  
13 end
```

Como vemos, podemos reducir considerablemente el numero de iteraciones totales iterando en r en el primer bucle hasta $n/2$ y en el segundo desde $r + 1$ hasta n , ya que así podemos evitar comparar dos veces el mismo movimiento. Es lo mismo cambiar r por s que s por r .

Búsqueda Local con *Don't Look Bits*

Como estamos utilizando una **búsqueda local del primer mejor**, podemos definir una lista de candidatos a la que llamamos ***Don't Look Bits*** que reducirá significativamente el tiempo de ejecución.

Se trata de un vector de bits inicialmente a 0, esto nos indica que todos los movimientos pueden ser considerados. Si tras probar todos los movimientos asociados un bit no hemos encontrado ninguna mejora, cambiaremos el valor de dicho bit a 1, indicando que esta unidad no debe ser tenida en cuenta hasta que dicha unidad asociada a ese bit se vea implicada en un movimiento que mejora la solución actual, en cuyo caso el bit será nuevamente 0.

Podemos ejemplificar este algoritmo con el siguiente pseudocódigo:

```

1 dlbMask = vector(n){0} # Don't look bits
2 def GenerateBestNeighbour( $\pi$ ):
3     for r = 0 to n do
4         if dlbMaskr ≠ 0 then continue;
5         for s = 0 to n do
6             if  $\Delta C(\pi, r, s) < 0$  then
7                  $\pi' = \pi$ 
8                  $t = \pi'(r)$ 
9                  $\pi'(r) = \pi'(s)$ 
10                 $\pi'(s) = t$ 
11                dlbMaskr = dlbMasks = 0
12                return  $\pi'$ 
13            end
14        end
15        dlbMaskr = 1
16    end
17 end

```

5. Procedimiento considerado para desarrollar la práctica

Esta práctica ha sido desarrollada en **C++** como una librería de metaheurísticas. La estructura del proyecto es la siguiente:

```

/Software
├── bin/ ..... Archivos ejecutables
│   └── practical ..... Ejecutable principal de la práctica
├── build/ ..... Directorio para compilación con CMake
├── doc/ ..... Otra documentación y código LaTeX de este documento
├── include/ ..... Cabeceras
├── instancias/ ..... Instancias de problemas sobre QAP
│   ├── *.dat ..... Definición de un problema
│   └── *.sln ..... Solución a un problema
├── src/ ..... Código fuente
├── CMakeLists.txt ..... Instrucciones de compilación para CMake
└── readme.txt ..... Instrucciones de uso

```

Para compilar este proyecto necesitamos las herramientas *g++* y *CMake*. Para instalarlas en un sistema **Ubuntu** o derivado, ejecutamos:

```
sudo apt-get install g++ cmake
```

Podemos compilar el proyecto con dos niveles de optimización:

- **Debug:** Sin optimización y con símbolos de depuración. Ejecutar CMake con opción `-D CMAKE_BUILD_TYPE=Debug`
- **Release:** Máxima optimización en la compilación. Por defecto.

Se compila con las siguientes instrucciones:

```

cd build/
# Para debug: cmake -D CMAKE_BUILD_TYPE=Debug ..
cmake ..
make clean
make
cd ..

```

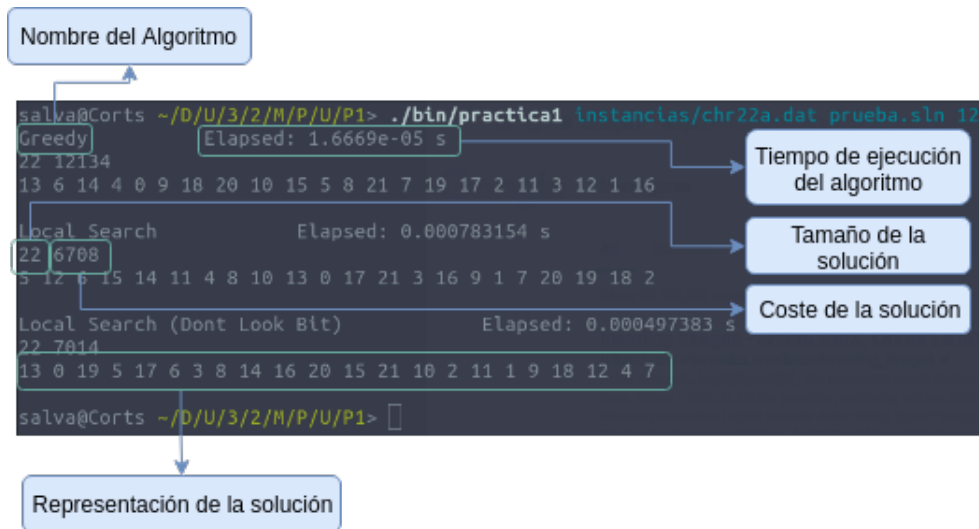
La sintaxis de ejecución es la siguiente¹:

```
./bin/practica1 instancias/<problema>.dat <semilla>
```

Por ejemplo:

```
./bin/practica1 instancias/chr22a.dat 12
```

La salida del programa tiene la siguiente estructura:



¹El parámetro semilla es opcional, por defecto *semilla* = 7

6. Experimentos y análisis de resultados

Con el fin de comparar los algoritmos implementados con los ya existentes, vamos a calcular para cada algoritmo los siguientes parámetros:

- **Desv**: Media de las desviaciones en porcentaje, del valor obtenido por cada método en cada instancia respecto al mejor valor conocido para ese caso.

$$\frac{1}{|\text{casos}|} \sum_{i=1}^{\text{casos}} 100 \frac{\text{valoralgoritmo}_i - \text{mejorValor}_i}{\text{mejorValor}_i}$$

Obtendremos los mejores valores conocidos de *QAPLIB*²

- **Tiempo**: se calcula como la media del tiempo de ejecución empleado por el algoritmo para resolver cada caso del problema.

Cuanto menor es el valor de Desv para un algoritmo, mejor calidad tiene dicho algoritmo. Por otro lado, si dos métodos obtienen soluciones de la misma calidad (tienen valores de Desv similares), uno será mejor que el otro si emplea menos tiempo en media.

Parámetros del experimento:

- *El valor de la semilla para este experimento es 7.*
- *Se compilara con parámetro **debug** a fin de contrastar aún mas los resultados del tiempo.*

²<http://anjos.mgi.polymtl.ca/qaplib//inst.html>

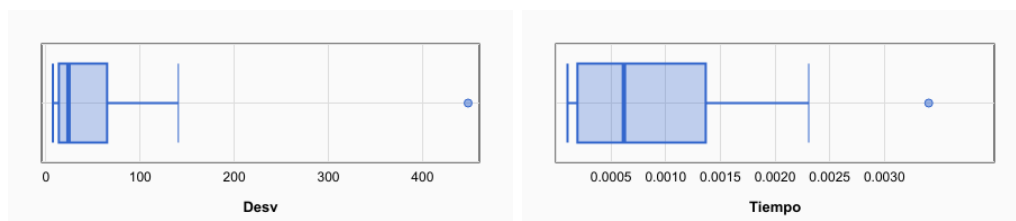
Resultados Greedy

Cuadro 1: goo.gl/yr6uN9

Algoritmo Greedy					
Caso	Desv	Tiempo	Caso	Desv	Tiempo
Chr22a	97,11	0,000126925	Sko100a	13,36	0,000630563
Chr22b	134,48	0,000125567	Sko100f	13,77	0,00230857
Chr25a	448,47	0,000175691	Tai100a	13,80	0,000616869
Esc128	140,63	0,00117301	Tai100b	32,79	0,000626035
Had20	7,71	0,000102693	Tai150b	24,97	0,00136394
Lipa60b	27,59	0,000289296	Tai256c	120,48	0,00340613
Lipa80b	28,58	0,000382927	Tho40	29,96	0,000384573
Nug28	22,88	0,00020704	Tho150	16,98	0,00204635
Sko81	15,50	0,0013694	Wil50	11,68	0,00017995
Sko90	13,61	0,00162198	Wil100	7,33	0,000619954

Como podríamos esperar, el algoritmo greedy es muy rápido, aunque las soluciones obtenidas distan bastante de la mejor conocida. Podríamos discutir su utilidad desde un punto de vista práctico; ¿En qué escenarios puede sernos útil una búsqueda greedy?

1. Si ofrecemos un servicio donde la optimalidad de la solución a un problema es lo de menos y tenemos que responder muchas solicitudes con pocos recursos de manera muy rápida.
2. Como algoritmo auxiliar de cara a obtener una solución inicial para posteriormente tratar de mejorarla mediante el uso de algoritmos mas complejos. No obstante, esto no siempre es buena idea pues corremos el riesgo de caer en óptimos locales. Hay que usarlos con algoritmos que sean capaces de esquivar con relativa facilidad óptimos locales.



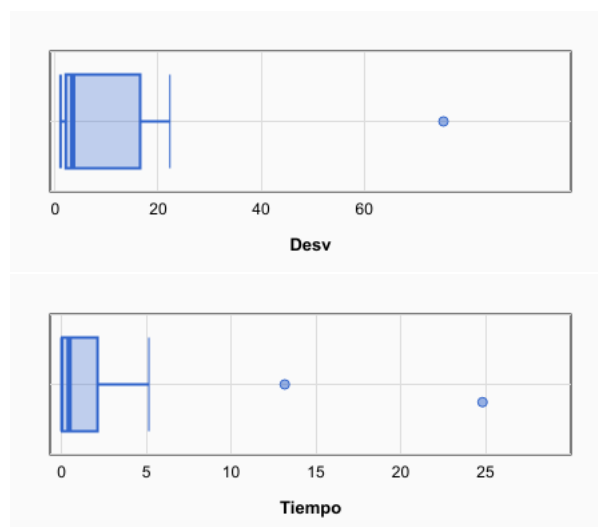
Resultados Búsqueda Local

Cuadro 2: goo.gl/SJJh1a

Algoritmo Búsqueda Local					
Caso	Desv	Tiempo	Caso	Desv	Tiempo
Chr22a	17,41	0,00255601	Sko100a	2,53	1,54504
Chr22b	15,63	0,00382979	Sko100f	2,21	2,05617
Chr25a	75,40	0,00940898	Tai100a	3,35	0,806562
Esc128	18,75	0,360157	Tai100b	4,94	3,26802
Had20	1,44	0,0058809	Tai150b	3,50	24,8278
Lipa60b	19,79	0,114542	Tai256c	1,26	5,15624
Lipa80b	22,30	0,291904	Tho40	5,34	0,0626425
Nug28	8,94	0,0114589	Tho150	2,78	13,1679
Sko81	2,70	0,540895	Wil50	1,96	0,10151
Sko90	1,86	1,46495	Wil100	1,07	2,19904

Aunque no se obtienen soluciones óptimas, estas están muy cerca de serlo. Podemos apreciar que este algoritmo trabaja muy bien con problemas relativamente pequeños pero dado que, a diferencia del greedy, el tiempo crece significativamente en función del tamaño del problema, utilizarlo como algoritmo de propósito general para solucionar problemas no es lo idóneo.

Sin embargo, en combinación con otros algoritmos como los Genéticos podemos obtener resultados aún mejores y en un tiempo mas aceptable dado que restringimos el espacio de búsqueda para este algoritmo.

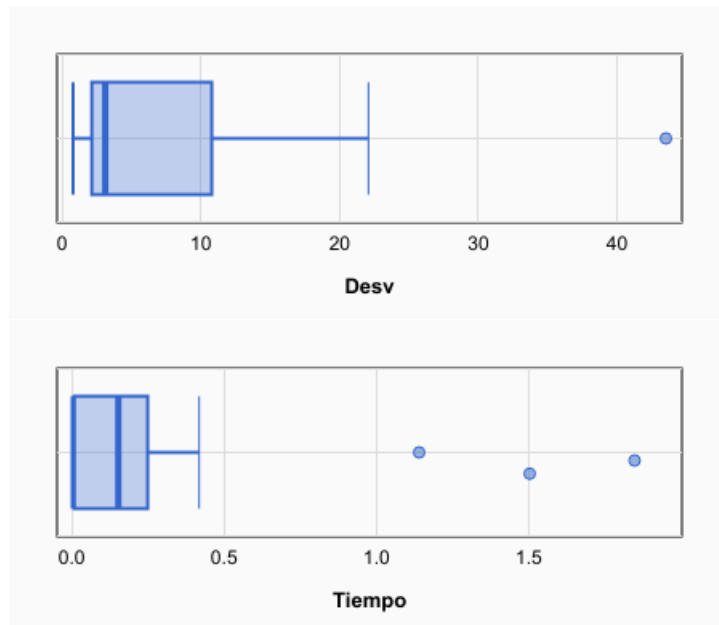


Resultados Búsqueda Local con *Don't Look Bits*

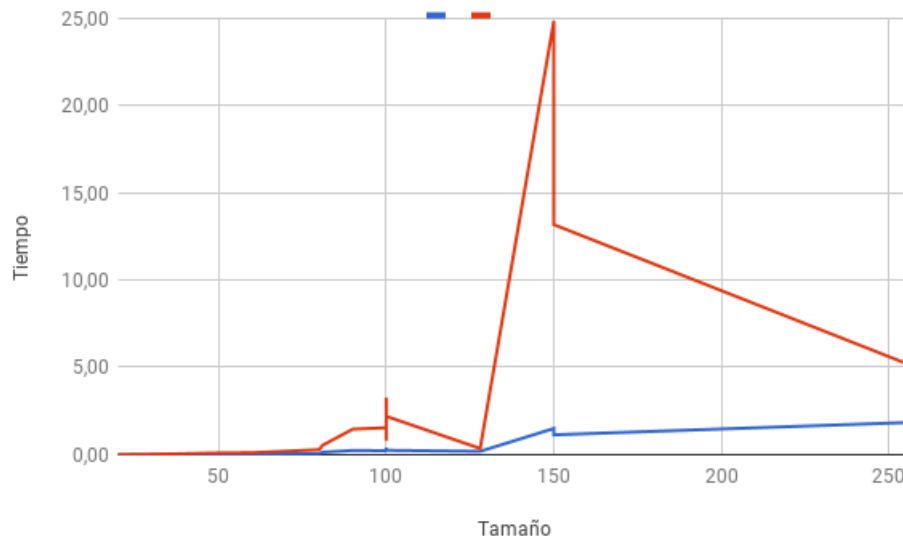
Cuadro 3: goo.gl/9frEmN

Algoritmo Búsqueda Local DLB					
Caso	Desv	Tiempo	Caso	Desv	Tiempo
Chr22a	18.23	0,0020404	Sko100a	2.78	0,229239
Chr22b	9.07	0,0059782	Sko100f	2.5	0,226626
Chr25a	43.52	0,00274717	Tai100a	4.12	0,164417
Esc128	12.5	0,193508	Tai100b	4.89	0,418418
Had20	2.11	0,00561109	Tai150b	2.59	1,5031
Lipa60b	20.23	0,0302634	Tai256c	0.79	1,84737
Lipa80b	22.08	0,0758918	Tho40	6.52	0,0116882
Nug28	3.45	0,0058682	Tho150	2.18	1,14028
Sko81	2.1	0,141866	Wil50	1.7	0,0221609
Sko90	2.67	0,246332	Wil100	1.26	0,252535

Podemos observar que la técnica de *Don't look bits* mejora significativamente los tiempos de ejecución de la versión básica de la búsqueda local.



En el siguiente gráfico podemos ver como evoluciona el tiempo de ejecución en función del tamaño tanto de la Búsqueda Local (rojo) como de su variante con *Don't Look Bits* (azul).



Se puede ver como el tamaño del problema no es lo único que afecta al tiempo de ejecución; La dificultad del problema en si (que es mucho mas difícil de medir) influye también. Por ejemplo, para la búsqueda local básica, se tarda más en *Tai150b* de tamaño 150 (24,83 segundos) que en *Tai256c* de tamaño 256 (5,15 segundos).

Comparación de los resultados. Conclusiones.

Algoritmo	Desv	Tiempo
Greedy	61,08	0,00088787315
BL	10,66	2,80
BL. Don't Look Bits	8.26	0,33

Como podemos ver, el tiempo de ejecución del algoritmo greedy es varias magnitudes menor que los tiempos de los otros dos algoritmos. Sin embargo, la poca calidad de sus soluciones hace que, por si sola, no sea una herramienta idónea de cara a resolver problemas.

Tanto la Búsqueda Local básica como su variante con *Don't look bits* nos ofrecen resultados mucho mas óptimos que el greedy.

Como la desviación de BL y BL con DLB es similar, debemos fijarnos en el tiempo medio de ejecución. La variante *DLB* es 8.5 veces más rápida³ que la original. Aunque la variante *Don't Look Bits* ofrece resultados ligeramente mejores, esto no debe ser tomado como referencia, pues la naturaleza aleatoria de cara a generar la solución de partida hace que no podamos asegurar que uno encuentra siempre una solución mejor que la otra.

Cabe destacar que el experimento ha sido compilado sin optimización (**Debug**). La realidad es que con un nivel máximo de optimización (**Release**), aún en problemas de mayor tamaño y complejidad como **Tai150b** obtenemos unos tiempos mucho menores que hace que utilizar un greedy en producción sea aun menos viable.

Un ejemplo ilustrativo de la diferencia de tiempos en función del nivel de optimización:

<i>Tai150b</i>	Nivel de Optimización	
Algoritmo	Debug	Release
BL	24.174	1.55584
BL DLB	1.54554	0.105939

³ $2,80/0,33 = 8.48$