Metaheurísticas

Práctica 2.a: Técnicas de trayectorias para el problema de asignación cuadrática

Curso 2016-2017. Grado en Ingeniería Informática

Alumno:
Sergio Carrasco Márquez DNI:76590869h

Contenido

1. Descripción del problema

El problema consiste en la asignación de unidades que tiene un flujo asociado entre ellas a localizaciones con un valor de distancia que las separa, de forma que las unidades con más flujo entre ellas estén separadas por distancias más cortas. Expresado de forma matemática el problema consiste en reducir el resultado de la función $\sum_{i=1}^{n}\sum_{j=1}^{n}(f_{ij}\ d_{\pi(i)\pi(j)})$. De tal forma que f_{ij} Simboliza el flujo entre las unidades i y j y $d_{\pi(i)\pi(j)}$ la distancia entre la localización a la que se asignan dichas unidades.

2. Descripción de la aplicación de los algoritmos

2.1 Función de evaluación

La solución al problema se representa como un vector en el que cada casilla indica la unidad a la que se hace referencia y el contenido de dicha casilla es la localización de dicha unidad. Para evaluar una solución se llama a una función que calcula el valor de la función explicada en el apartado anterior, $\sum_{i=1}^{n}\sum_{j=1}^{n}(f_{ij}\ d_{\pi(i)\pi(j)})$, dependiendo de los valores asignados a cada posición del vector. La función se implementa de la siguiente manera:

```
Evaluación = 0
para i=0 hasta tamaño_del_vector
        para j=0 hasta tamaño_del_vector
                //v es el vector que almacena la solución
                evaluación += FlujoEntre(i,j)*Distancia(v[i],v[j])
return evaluación
```

2.2 Soluciones aleatorias para ILS, BMB y BL

La generación de soluciones aleatorias se realiza de forma simple, pero cumpliendo la restricción del problema sobre las soluciones, que no se asigne la misma localización a dos unidades distintas. En nuestro caso esa restricción implica que el en vector solución no pueden aparecer dos valores repetidos. Para generar vectores que cumplan esta restricción se crea un vector ordenado con todos los valores posibles, es decir desde 0 hasta el tamaño del vector y luego se baraja.

```
Para i = 0 hasta tamaño_del_vector
        V[i] = i
//Barajar el vector
Para i = 0 hasta tamaño_del_vector
        position = rand(i,tamaño_del_vector-1)//aleatorio entre i el el tamaño del vector -1
        swap = v[i]
        v[i] = v[posicion]
        v[posicion] = swap
```

3. Descripción de los algoritmos implementados

3.1 Búsqueda local

La búsqueda local se basa en generar vecinos de una solución de partida aleatoria y en este caso escoger el primer vecino mejor que dicha solución. Una vez escogido un vecino se repite el proceso de generar vecinos hasta que o bien no existan vecinos mejores, lo que hace que encontremos un óptimo local o global, o bien hasta que se realicen un número de iteraciones o evaluaciones concretos. La generación de vecinos se realiza intercambiando una posición por otra en el vector. En este caso se usa una máscara don´t look bits que indica si intercambiar una posición es o no prometedor, lo que ahorrara evaluaciones poco prometedoras y permite una menor exploración del vecindario de una solución.

El proceso de encontrar al primer vecino mejor se realiza de la siguiente forma:

```
Procedure EncuentraMejorVecino
para i = 0 hasta tamaño_vector && encontrado == false
        posi = orden[i]
        si(DLB[posi] == true)
                para j = 0 hasta tamaño_vector && encontrado == false
                        posj = orden[j]
                        si(i ¡= j)
                                swap = v[posi]
                                v[posi] = v[posj]
                                v[posj] = swap
                                evaluacion = FactorizacionVector(v,solucion_actual,posi,posj)
                                iteraciones++
                        fin si
                        si evaluacion < solucion_actual
                                encontrado = true
                                DLB[posi] = true
                                DLB[posj] = true
                        sino
                                swap = v[posi]
                                v[posi] = v[posj]
                                v[posj] = swap
                fin para
        fin si
        si(found == false)
                DLB[posi] = false
        fin si
fin para
return found
```

La búsqueda local hace llamadas a esta función de manera reiterada mientras no se cumpla la condición de parada

```
Procedure BusquedaLocal
DLB = true //Dont look bits inicializado a true en todas las posiciones
mientras(iteraciones <max_iters  and encontrado )
        encontrado = EncuentraMejorVecino(v,iteraciones,DLB)
return v
```

Por último queda explicar cómo se evalúa la solución vecina usando factorización.

```
//Primero se invierte el cambio realizado en el vector
Swap = v[p1]
V[p1] = v[p2]
 v[p2] = swap

incremento = 0
para i = 0 hasta tamaño_del_vector
        si(i != p1 and i != p2)
                incremento += flujo(p1,i)*(distancia(v[p1],i) – distancia(v[p2],v[i] )) +
                flujo(p1,i)*( distancia( v[p2],v[i] ) – distancia( v[p1],v[i] ) ) +
                flujo(i,p2)*( distancia( v[i],v[p1] )-distancia(v[i],v[p2]) )+
                flujo(i,p1)*( distancia(v[i],v[p2])-distancia(v[i],v[p1]) )
        fin si
fin para
//Deshacer el cambio en el vector
 Swap = v[p1]
V[p1] = v[p2]
 v[p2] = swap

eval+=incremento
return eval
```

3.2 Enfriamiento Simulado

El algoritmo de enfriamiento simulado se basa en una búsqueda local que es capaz de escapar de óptimos locales escogiendo soluciones peores que la mejor encontrada. La probabilidad de escoger una peor solución depende de un parámetro que indica la temperatura del sistema. En cada iteración la temperatura disminuye otorgando al algoritmo una mayor convergencia en las etapas finales.

La implementación es pseudocódigo es la siguiente:

```
solucion_actual = Solucion_Aleatoria
evaluacion_actual = evaluar(solucion_actual)
mejor_solucion = solucion_actual
mejor_evaluacion = evaluacion_actual
//Esquema de enfriamiento
t0 = mu*evaluacion_actual/-log(fi)
```

```
tf = 10^-3
temperatura = t0
m = max_evaluaciones/max_vecinos
b = (t0-tf)/(m*t0*tf)

Mientras(n_evals < max_evals && exito)

        Mientras(vecinos_generados < max_vecinos && exitos < max_exitos)
                exitos = 0
                //Generación de vecino
                p1 = aleatorio(0,n)
                p2 = aleatorio(0,n)
                swap = soluicon_actual[p1]
                solucion_actual[p1] = solucion_actual[p2]
                solucion_actual[p2] = swap
                //Evaluación del vecino
                evaluacion_nueva = evaluar(solucion_actual)
                si(evaluacion_nueva < evaluacion_actual)
                        exitos++
                        si(evaluacion_nueva < mejor_evaluacion)
                                mejor_solucion = solucion_actual
                                mejor_evaluacion = evaluacion_actual
                        evaluacion_actual = evaluacion_nueva
                sino
                        //Se puede aceptar esta solución aunque sea peor
                        si(exp((-1 * (evaluacion_nueva - evaluacion_actual)) /
                                (iteracion * t))
                                exitos++
                                si(evaluacion_nueva < mejor_evaluacion)
                                        mejor_solucion = solucion_actual
                                        mejor_evaluacion = evaluacion_actual
                                evaluacion_actual = evaluacion_nueva
                                sino
                                //Si la solucion no se escoge, se restarua la solución
                                //actual
                                        p1 = aleatorio(0,n)
                                        p2 = aleatorio(0,n)
                                        swap = soluicon_actual[p1]
                                        solucion_actual[p1] = solucion_actual[p2]
                                        solucion_actual[p2] = swap
                vecinos_generados++
        Fin Mientras
        si(exitos == 0)
                exito = false
        //Actualización de la temperatura y de la iteración
        iteracion++
        temperatura = temperatura/(1+(b*temperatura));
Fin Mientras
return mejor_solucion
```

3.3 GRASP

El algoritmo GRASP se basa en una búsquela local multiarranque en la que se lanza una búsqueda local para un conjunto de soluciones iniciales generadas por un algoritmo voraz con una componente aleatoria. El algoritmo voraz hace que la solución de partida sea mejor que una generada aleatoriamente, por lo que la exploración de las búsquedas locales se realiza en zonas más prometedoras.

El pseudocódigo de dicho algoritmo consta de dos partes, la primera que sería la generación de las soluciones voraces aleatorias y la segunda que consta de la búsqueda local multiarranque. Para la generación de soluciones voraces se usa un umbral de calidad que determina si una localización puede ser asignada a una unidad concreta. Si la asignación cumple con el umbral establecido entonces dicha asignación entra dentro de las asignaciones candidatas para ser escogida de forma aleatoria. En este caso particular se han establecido las dos primeras asignaciones de la misma forma que lo haría un algoritmo voraz determinista .

Procedure RandomGredy

```
para i = 0 hasta n
        val = 0
        para j = 0 hasta n
                val+=flujoEntre(i,j)
        flujos[i].unidad = unidad
        flujos[i].flujo = val
fin para
ordenaMayorMeno(flujos)

para i = 0 hasta n
        val = 0
        para j = 0 hasta n
                val+=DistanciaEntre(i,j)
        distancias[i].unidad = unidad
        distancias[i].flujo = val
fin para
ordenaMenorMayor(distancias)

para i = 0 hasta n
        solucion[i] = -1
fin para
solucion[flujos[0].unidad] = distancias[0].unidad
solucion[flujos[1].unidad] = distancias[1].unidad

localizaciones_usadas[distancias[0].unidad] = false
localizaciones_usadas[distancias[1].unidad] = false
localizaciones_usadas = 2
coste = flujoEntre(flujos[0].unidad,flujos[1].unidad)*
DistanciaEntre(distancias[0].unidad,distancias[1].unidad)
para i = 2 hasta n
```

```
        unidad = flujos[i].unidad
        indice_localizaciones = 0
        para j = 0 hasta n
                localizacion = distancias[j].unidad
                si(localizaciones_usadas[localizacion] == false)
                        //Si la localización no ha sido usada, se prueba como candidata
                        coste_unidad = coste
                        //Calculo del coste de localización
                        para k = 0 hasta n
                                s = solucion[k]
                                si(s != -1)
                                        distancia =
                                        DistanciaEntre(solucion[k],localizacion)
                                        flujo = flujoEntre(k,unidad)
                                        coste_unidad = distancia*flujo
                                fin si
                        fin para
                        ucs.cost = coste_unidad
                        ucs.unidad = localuzacion
                        localizaciones_candidatas[indice_localizaciones] = ucs
                        indice_localizaciones++
                fin si
        fin para
        ordenaMenorMayor(localizaciones_candidatas)
        //Calculo del umbral
        coste_mejor = localizaciones_candidatas[0].cost
        coste_peor = localizaciones_candidatas[n-1-n_localizaciones usadas].cost
        umbral = coste_mejor + umbral_calidad*(coste_peor-coste_mejor)
        indice_corte = 0
        encontrado = false
        para j = 0 hasta n-n_localizaciones_usadas and found == false
                si(localizaciones_candidatas[j].coste > umbral)
                        indice_corte = j-1
                        encontrado = true
                fin si
        fin para
        asignacion = Aleatorio(0,indice_corte)
        solucion[unidad] = localizaciones_candidatas[asignacion].unidad
        n_localizaciones_usadas++
        cost += localizaciones_candidatas[asignacion].cost
fin para

Procedure GRASP
solucion_actual = RandomGreedy
evaluacion_actual = EvaluarSolucion(solucion_actual)
mejor_solucion = solucion_actual
mejor_evaluacion = evaluacion_actual
para i = 0 hasta n_iteraciones
        BusquedaLocal(solucion_actual)
        si(evaluacion_actual < mejor_Evaluacion)
```

```
            mejor_solucion = solucion_actual
            mejor_evaluacion = evaluacion_actual
        fin si
        solucion_actual = RandomGreedy
fin para
return mejor_solucion
```

3.4 ILS

El algoritmo ILS se basa en una búsqueda que escapa de óptimos locales usando un operador de mutación sobre dicho óptimo para relanzar la búsqueda. Se han implementado dos esquemas distintos de ILS, uno que utiliza la búsqueda local y otro que utiliza el enfriamiento simulado. El operador de mutación usado en ambos esquemas consiste en escoger una sección de la solución y realizar un reordenamiento aleatorio de dicha sección.

3.4.1 ILS-BL

```
solucion_actual = Solucion_aleatoria()
evaluacion_actual = EvaluarSolucion(solucion_actual)
mejor_solucion = solucion_actual
para i = 0 hasta n_iteraciones
        //mutación de la solución
        inicio_sublista = Aleatorio(0,tamaño-tamaño_sublista-1)
        posiciones = GeneraVectorAleatorio()
        para j = 0 hasta tamaño_sublista
            swap = solucion_actual[j+inicio_sublista]
            solucion_actual[j+inicio_sublista] = solucion_actual[j+posiciones[j]]
            solucion_actual[j+posiciones[j]] = swap
        fin para
        BusquedaLocal(solucion_actual)
        evaluacion_actual = EvaluarSolucion(solucion_actual)
        si(evaluacion_actual < mejor_evaluacion)
            mejor_solucion = solucion_actual
            mejor_evaluacion = evaluacion_actual
        sino
            solucion_actual = mejor_solucion
            evalaucion_actual = mejor_evalaucion
fin para
return mejor_solucion
```

3.4.2 ILS-ES

```
solucion_actual = Solucion_aleatoria()
evaluacion_actual = EvaluarSolucion(solucion_actual)
mejor_solucion = solucion_actual
para i = 0 hasta n_iteraciones
        //mutación de la solución
        inicio_sublista = Aleatorio(0,tamaño-tamaño_sublista-1)
        posiciones = GeneraVectorAleatorio()
        para j = 0 hasta tamaño_sublista
```

```
            swap = solucion_actual[j+inicio_sublista]
            solucion_actual[j+inicio_sublista] = solucion_actual[j+posiciones[j]]
            solucion_actual[j+posiciones[j]] = swap
        fin para
        EnfriamientoSimulado(solucion_actual)
        evaluacion_actual = EvaluarSolucion(solucion_actual)
        si(evaluacion_actual < mejor_evaluacion)
            mejor_solucion = solucion_actual
            mejor_evaluacion = evaluacion_actual
        sino
            solucion_actual = mejor_solucion
            evalaucion_actual = mejor_evalaucion
fin para
return mejor_solucion
```

3.5 BMB

El algoritmo BMB consiste en una búsqueda multiarranque simple, se generan un conjunto de soluciones iniciales y se lanza una búsqueda local sobre dichas soluciones.

```
 solucion_actual = SolucionAleatoria()
evaluacion_actual = EvaluarSolucion(solucion_actual)
mejor_solucion = solucion_actual
mejor_evalaucion = evaluacion_actual
para i = 0 hasta n_soluciones
        BusquedaLocal(solucion_actual)
        si(evalaucion_actual < mejor_evaluacion)
            mejor_solucion = solucion_actual
            mejor_evaluacion = evaluacion_actual
        fin si
        solucion_actual = SolucionAleatoria()
fin para
return mejor_solucion
```

4. Análisis de los resultados

Se han realizado una serie de experimentos para evaluar la calidad de los algoritmos descritos anteriormente. Para replicar los experimentos se indica que se ha usado como semilla aleatoria 3. Para compilar el archivo basta con usar el compilador g++ de Linux con la opción de optimización –O2. La ejecución del programa recibe como primer parámetro el nombre del archivo que se usa como conjunto de datos y como segundo parámetro recibe la semilla.

4.1 Búsqueda local

BL			
Caso	**Coste obtenido**	**Desv**	**Tiempo**
Chr20b	3270	42,30	0,00056
Chr22a	7050	14,52	0,000828
Els19	23688500	37,62	0,000641
Esc32b	228	35,71	0,002231
Kra30b	98480	7,72	0,002641
Lipa90b	15250729	22,10	0,067116
Nug25	3828	2,24	0,001493
Sko56	35348	2,58	0,018648
Sko64	49766	2,61	0,027731
Sko72	68668	3,64	0,036686
Sko100a	155542	2,33	0,138898
Sko100b	156922	1,97	0,146787
Sko100c	150352	1,68	0,136182
Sko100d	152214	1,76	0,114925
Sko100e	153946	3,22	0,087816
Tai30b	816324715	28,13	0,002339
Tai50b	479142881	4,43	0,019607
Tai60a	7531464	4,52	0,02222
Tai256c	45274196	3,25	0,90116
Tho150	8315108	9,11	0,50296
		11,57	0,11

Los resultados son buenos si nos fijamos en la relación desviación/tiempo, aunque estos resultados serán peores que los que ofrecen algunos de los algoritmos que se estudiarán a continuación. Algunos conjuntos de datos dan malas soluciones por el hecho de que la solución de partida no es prometedora o simplemente porque está cerca de un mal óptimo local. La velocidad que consigue este algoritmo se debe a la rápida convergencia del mismo, por lo que para obtener soluciones con un coste temporal bajo la búsqueda local se presenta como una buena alternativa.

4.2 Enfriamiento Simulado

ES			
Caso	**Coste obtenido**	**Desv**	**Tiempo**
Chr20b	3336	45,17	0,049464
Chr22a	7398	20,18	0,054491
Els19	29647820	72,25	0,049384
Esc32b	188	11,90	0,080671
Kra30b	96790	5,87	0,07565
Lipa90b	15208189	21,76	0,240817
Nug25	3850	2,83	0,060564
Sko56	35350	2,59	0,118894
Sko64	49162	1,37	0,16402
Sko72	67448	1,80	0,188965
Sko100a	155582	2,36	0,265453
Sko100b	156738	1,85	0,260535
Sko100c	150414	1,73	0,262196
Sko100d	152402	1,89	0,259672
Sko100e	151906	1,85	0,274095
Tai30b	729792335	14,55	0,08039
Tai50b	514682584	12,17	0,132855
Tai60a	7548822	4,76	0,159019
Tai256c	45130716	2,92	0,68441
Tho150	8329318	9,30	0,291804
		11,95	0,19

En cuanto al algoritmo de enfriamiento simulado el concepto es similar al de la búsqueda local, pero se le permite escapar de óptimos locales. Los resultados obtenidos son en primera instancia peores que los obtenidos por la búsqueda local, pero hay que tener en cuenta que la implementación de la búsqueda local hace que no se generen vecinos repetidos y usa una máscara don´t look bits, que focaliza la búsqueda hacia zonas prometedoras a cambio de una convergencia más rápida. La implementación del enfriamiento simulado no tiene elementos que focalizan la búsqueda ni se ahorra generar soluciones repetidas, por que acercarse tanto a la búsqueda local hace que en general este algoritmo sea bastante bueno. La superioridad de este algoritmo destaca más en los algoritmos ILS, dónde como se analizará posteriormente la técnica de enfriamiento simulado consigue mejores resultados.

4.3 ILS ILS-ES

ILS				ILS-ES			
Caso	Coste obtenido	Desv	Tiempo	Caso	Coste obtenido	Desv	Tiempo
Chr20b	2658	15,67	0,01125	Chr20b	2726	18,62	1,11571
Chr22a	6630	7,70	0,016832	Chr22a	6608	7,34	1,05438
Els19	23246770	35,06	0,006123	Els19	19278506	12,00	1,15977
Esc32b	192	14,29	0,029606	Esc32b	168	0,00	1,58931
Kra30b	93990	2,81	0,027664	Kra30b	93060	1,79	1,22167
Lipa90b	15125059	21,09	1,31073	Lipa90b	15144976	21,25	3,57088
Nug25	3758	0,37	0,024675	Nug25	3750	0,16	0,963254
Sko56	34646	0,55	0,396271	Sko56	34964	1,47	2,12292
Sko64	49102	1,25	0,61314	Sko64	49138	1,32	2,4263
Sko72	66924	1,01	0,82951	Sko72	66898	0,97	2,99657
Sko100a	152754	0,49	2,61422	Sko100a	153526	1,00	3,9636
Sko100b	154716	0,54	2,36375	Sko100b	155362	0,96	3,9501
Sko100c	149784	1,30	2,07401	Sko100c	149684	1,23	4,18402
Sko100d	151044	0,98	1,8798	Sko100d	151142	1,05	3,9899
Sko100e	150462	0,88	1,8836	Sko100e	150950	1,21	3,91789
Tai30b	650220622	2,06	0,038961	Tai30b	640177356	0,48	1,11134
Tai50b	469418762	2,31	0,240321	Tai50b	466996345	1,78	1,88288
Tai60a	7473572	3,71	0,270538	Tai60a	7444460	3,31	2,32724
Tai256c	44964364	2,54	15,1103	Tai256c	44955820	2,52	12,7318
Tho150	8188406	7,45	7,29499	Tho150	8287164	8,75	6,12592
		6,10	1,85			4,36	3,12

El algoritmo ILS ofrece mejores resultados que una búsqueda local sola, ya que consigue una gran diversidad de soluciones y consigue una mejor combinación de exploración y exploración. Pero es la versión combinada con el enfriamiento simulado la que consigue una mejora notable con respecto a la implementación con búsqueda local. Esta diferencia hace que el algoritmo de enfriamiento simulado sea una mejor opción para combinarlo con algoritmos basados en poblaciones para potenciar dichas poblaciones, que en media serán mejores si se usa este modelo. Dónde el modelo ILS-ES pierde frente al modelo ILS con búsqueda local es en el tiempo de ejecución, aunque en relación desviación/tiempo el modelo ILS-ES sigue siendo superior.

4.4 BMB

BMB			
Caso	**Coste obtenido**	**Desv**	**Tiempo**
Chr20b	2856	24,28	0,015107
Chr22a	6750	9,65	0,02261
Els19	18054452	4,89	0,018736
Esc32b	188	11,90	0,061266
Kra30b	94370	3,23	0,062008
Lipa90b	15167916	21,44	1,50617
Nug25	3790	1,23	0,037309
Sko56	35154	2,02	0,496925
Sko64	49194	1,44	0,760261
Sko72	67578	2,00	0,878613
Sko100a	154506	1,65	2,28245
Sko100b	156100	1,44	2,29263
Sko100c	150138	1,54	2,16033
Sko100d	151036	0,98	2,2295
Sko100e	151456	1,55	2,35183
Tai30b	640376356	0,51	0,051918
Tai50b	460114081	0,28	0,322851
Tai60a	7517526	4,32	0,416034
Tai256c	45014408	2,66	16,6855
Tho150	8269744	8,52	8,95915
		5,28	2,08

El algoritmo de búsqueda local multiarranque básico ofrece unos resultados muy buenos, debido a la explotación del entorno producida. Esta explotación es mayor que en el resto de algoritmos, ya que ILS usa mutaciones de una solución, mientras que BMB usa un conjunto de soluciones aleatorias que pueden ser muy distintas entre si, y por lo tanto tiene más probabilidades de llevarnos a una solución buena y explorar un mayor número de óptimos locales. Los algoritmos ILS al realizar mutaciones no se alejan lo suficiente del entorno de una solución inicial de forma que tras el proceso de mutación podemos dirigirnos al mismo óptimo local. Los algoritmos BMB pierden capacidad de explotación frente a los ILS, pero la ganancia de exploración hace que los resultados ofrecidos sean algo mejores. En cuanto al tiempo empleado en las sucesivas búsquedas la diferencia no es significativa, por lo que no pude decirse que este algoritmo sea pero en tiempo que un ILS.

4.5 GRASP

GRASP			
Caso	**Coste obtenido**	**Desv**	**Tiempo**
Chr20b	2878	25,24	0,016023
Chr22a	6510	5,75	0,022762
Els19	20508428	19,15	0,016737
Esc32b	192	14,29	0,060056
Kra30b	94790	3,69	0,055414
Lipa90b	15192741	21,63	1,58311
Nug25	3796	1,39	0,018871
Sko56	35252	2,30	0,335205
Sko64	49478	2,02	0,534702
Sko72	67448	1,80	0,776497
Sko100a	153402	0,92	2,37938
Sko100b	156150	1,47	2,09187
Sko100c	149752	1,28	2,4741
Sko100d	151818	1,50	2,18745
Sko100e	151698	1,71	2,32265
Tai30b	648240842	1,75	0,078126
Tai50b	473661857	3,23	0,379174
Tai60a	7465446	3,60	0,326893
Tai256c	44985852	2,59	16,4935
Tho150	8268908	8,51	8,29995
		6,19	2,02

El modelo GRASP sigue una filosofía muy similar a la del modelo BMB, pero intentando perder algo de exploración generando soluciones iniciales a partir de una solución voraz con componentes aleatorias. Sin embargo esa pérdida de exploración produce que las soluciones iniciales estén más orientadas a soluciones buenas, por lo que es ahí donde reside la potencia de este algoritmo. La diferencia en desviación media entre uno y otro es de algo más de un 1%, por lo que no se pueden sacar conclusiones claras sobre si uno es mejor que otro, o sobre si la focalización está justificada o no. También debe tenerse en cuenta cómo reacciona el problema ante soluciones voraces. A continuación se muestra una tabla con el resultado del algoritmo voraz descrito en secciones anteriores.

GREEDY			
Caso	**Coste obtenido**	**Desv**	**Tiempo**
Chr20b	10036	336,73	1,00E-05
Chr22a	12420	101,75	1,00E-05
Els19	38627698	124,42	9,00E-06
Esc32b	336	100,00	1,30E-05
Kra30b	118550	29,68	1,40E-05
Lipa90b	16183718	29,57	5,40E-05
Nug25	4446	18,75	1,10E-05
Sko56	40164	16,56	2,70E-05
Sko64	57350	18,25	3,20E-05
Sko72	76936	16,12	4,10E-05
Sko100a	172224	13,30	7,10E-05
Sko100b	174384	13,32	7,20E-05
Sko100c	167584	13,34	6,20E-05
Sko100d	170776	14,17	6,10E-05
Sko100e	171646	15,08	6,10E-05
Tai30b	1387185541	117,73	1,30E-05
Tai50b	788404422	71,83	2,30E-05
Tai60a	8393560	16,48	3,00E-05
Tai256c	83407180	90,21	0,000337
Tho150	9448176	23,98	0,000124
		59,06	0,00

Con estos resultados podemos concluir que las soluciones voraces generadas no son exesivamente malas para tratarse de algoritmos voraces y que un proceso de búsqueda sobre dichas soluciones podría dar buen resultado. De hecho los mejores resultados del algoritmo GRASP aparecen en los mismos conjuntos de datos dónde aparecen las mejores soluciones aleatorias, por lo que la eficacia del GRASP se debe a la eficacia de las soluciones voraces y dependerá de la naturaleza del problema. Un estudio previo del problema usando soluciones voraces, que se obtienen en poco tiempo puede arrojar un poco de luz sobre si usar un GRASP o si usar un BMB.

4.6 Vistazo general

Si comparamos todos los algoritmos estudiados obtenemos la siguiente tabla.

Algoritmo	desviacón	tiempo
BL	11,57	0,11
ILS	6,1	1,85
ILS-ES	4,36	3,12
BMB	5,28	2,08
GREEDY	59,06	tiende a 0
GRASP	6,19	2,02
ES	11,95	0,19

En la tabla se muestra como el mejor algoritmo presentado es el ILS-ES en cuanto a calidad de resultado se refiere, pero no dista mucho del resto de soluciones basadas en búsquedas multiarranque. Estos algoritmos mejoran notablemente a las búsquedas simples como la búsqueda local o el enfriamiento simulado, por lo que podemos concluir que añadir algo de exploración a las técnicas básicas de explotación produce buenos resultados. Esta conclusión demuestra que la mejor utilidad que tienen los algoritmos de explotación es acercar a un conjunto de soluciones a soluciones mejores y potenciar los resultados obtenidos en un tiempo más que razonable, ya que la tabla de tiempos muestra que los algoritmos no son computacionalmente muy pesados.