PRÁCTICA 2
ALGORITMOS GENÉTICOS PARA QAP

JULIO A. FRESNEDA GARCÍA – 49215154F – JULIOFRESNEDAG@CORREO.UGR.ES – ALGORITMOS GENÉTICOS – GRUPO 1 – LUNES 5:30-7:30

DESCRIPCIÓN DEL PROBLEMA

El problema de Asignación Cuadrática (QAP) es uno de los problemas de optimización combinatoria más complejos. Es NP-completo.

En el QAP, lo que tratamos de hacer es asignar n unidades a una cantidad n de localicaziones, donde se considera un coste aosicado a cada una de las asignaciones. Este coste dependerá de la distancia entre localizaciones y del flujo entre unidades.

Lo que se busca es que el coste total, en función de la distancia y el flujo, sea mínimo.

En este problema contamos con dos matrices, la matriz de flujos y la matriz de distancias, que nos sirven para obtener el flujo entre unidades y la distancia entre localizaciones.

DESCRIPCIÓN DE LA APLICACIÓN DE LOS ALGORITMOS AL PROBLEMA

La función objetivo es la siguiente:

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

donde f es la matriz de flujos, d la matriz de distancias y S el vector solución.

Vamos a considerar lo siguiente:

Tenemos dos matrices, una para representar los flujos entre unidades, y otra matriz para representar la distancia entre localizaciones.

La distancia entre las localizaciones i y j se comprueba en la matriz de distancias, en la posición [i][j]. De igual forma, el flujo entre las unidades k y l se comprueba en la matriz de flujos, en la posición [k][l].

Las bases de datos que usaremos para comprobar nuestros algoritmos son archivos .dat que tienen el siguiente formato:

Encabeza un número que representa el tamaño del problema. Le sigue una línea en blanco, y después la matriz de flujos. Le sigue una línea en blanco (en algunos archivos), y la matriz de distancias.

Para representar las posibles soluciones, usaremos un vector de enteros, en el cual las posiciones del vector representan las unidades, y el contenido de cada posición representa a la localización asignada a esa unidad. Por ejemplo, si tenemos como solución {2,3,1,0} significa que la unidad 0 está asociada a la localización 2; la unidad 1, a la localización 3; etc.

Para calcular el coste de una solución en concreto, usamos la siguiente fórmula:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} f_{ij} \cdot d_{S(i)S(j)}$$

En pseudocódigo, el cálculo de coste de una solución sería así:

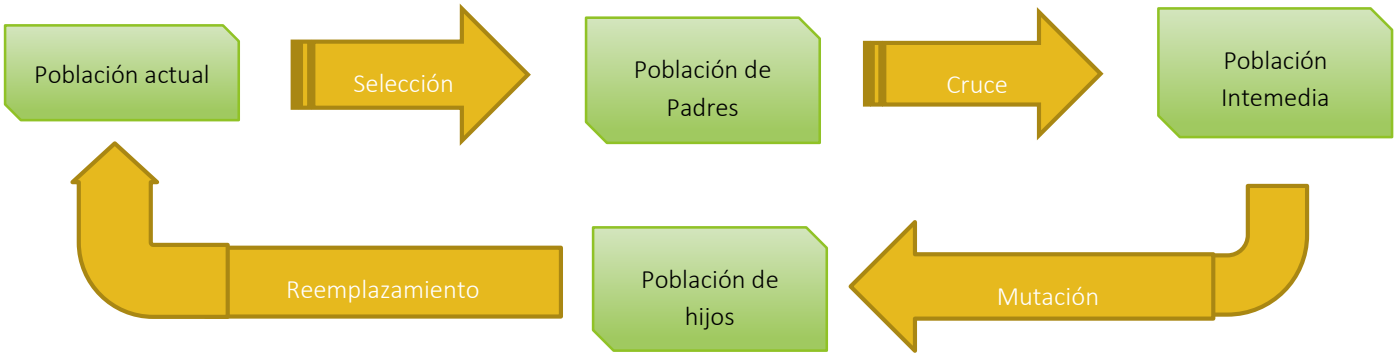
```
Coste ← 0
for i=0 to n
        for j=0 to n
                if i != j
                        coste = coste + flujos[i][j]*distancias[solución[i]][solución[j]]
                endif
        endfor
endfor
```

Para llegar a la solución S mínima, vamos a implementar algoritmos genéticos y meméticos: Dos algoritmos genéticos generacionales, dos algoritmos genéticos estacionarios y tres algoritmos meméticos.

CONSIDERACIONES COMUNES A LOS ALGORITMOS

En este apartado, se van a describir todas las consideraciones comunes a los algoritmos.

Todos los algoritmos genéticos siguen el siguiente esquema:



La etapa en la que empiezan los algoritmos es en la etapa de Población actual. En esta etapa, la primera población la componen un determinado número de objetos de tipo QAP (cromosomas). Los objetos QAP contienen información de una asignación de localizaciones en unidades determinada (genes), y de su coste, entre otras cosas. Estos objetos QAP se inicializan por primera vez con una asignación de localizaciones en unidades aleatoria. Esta solución aleatoria se inicializa de la siguiente forma:

```
for i=0 to n
        ordenados.Add( i )
endfor

while ordenados.Length != 0
        i ← random(0,ordenados.Length)
        solución.Add( ordenados[i] )
        remove ordenados[i]
endwhile
return solucion
```

Cada objeto QAP se inicializan de forma aleatoria si los creamos pasando como argumento una ruta del archivo de datos. Por lo tanto, para crear una primera población inicial con cromosomas aleatorios, basta con hacer:

```
for i=0 to numCromosomas
        población.Add( new QAP(ruta) )
endfor
```

SELECCIÓN

Una vez tenemos la población inicial, debemos hacer la selección. El número de cromosomas que seleccionaremos dependerá del algoritmo, pero todos los algoritmos tienen algo en común:

Para seleccionar un cromosoma realizaremos un torneo binario entre dos cromosomas aleatorios.

El torneo binario es un método que a partir de dos enteros (que corresponden a los índices del cromosoma en la lista PoblaciónInicial) compara los dos cromosomas y devuelve el mejor. En pseudocódigo, el método de torneo binario sería así:

```
QAP TorneoBinario( pos1, pos2 )
        if( población[pos1].coste < población[pos2].coste
                return población[pos1]
        else return población[pos2]
end TorneoBinario
```

Una vez tenemos los cromosomas seleccionados, tenemos la población de padres.

CRUCE

La siguiente etapa consiste en cruzar estos padres entre ellos. Al igual que con la selección, el número de cromosomas de la población de padres que cruzaremos será distinto para cada algoritmo.

Para cada cruce, obtendremos dos hijos.

Para cruzar dos cromosomas, podemos usar dos métodos distintos: Cruce por posición y cruce PMX.

Recordemos que para representar las asignaciones de localizaciones en unidades se ha usado una lista. Cada localización está en una determinada posición de esta lista, y representa a un gen.

CRUCE POR POSICIÓN

El cruce por posición consiste en mantener los genes que coincidan en valor y posición en ambos cromosomas, y permutar el resto. En pseudocódigo, el cruce por posición se realiza así:

```
pair<QAP,QAP> CruzarPosicion( QAP crom1, QAP crom2 )
        genesCrom1 ← crom1.LocalizacionesEnUnidades
        genesCrom2 ← crom2.LocalizacionesEnUnidades

        for i=0 to numGenes
                if genesCrom1[i] = genesCrom2[i]
                        genesComunes.Add( genesCrom1[i] )
                endif
                else
                        genesComunes.Add( -1 )
                        restos.Add( genesCrom1[i] )
                endelse
        endfor

        nuevosgenesCrom1 ← genesComunes
        nuevosgenesCrom2 ← genesComunes

        Permutar(restos)
        j ← 0
        for i=0 to numGenes
                if nuevosgenesCrom1[i] = -1
                        nuevosgenesCrom1[i] = restos[j]
                        j++
                endif
        endfor

        Permutar(restos)
        j ← 0
        for i=0 to numGenes
                if nuevosgenesCrom2[i] = -1
                        nuevosgenesCrom2[i] = restos[j]
                        j++
                endif
        endfor

        parQAPs ← new QAP(nuevosgenesCrom1), new QAP(nuevosgenesCrom2)

        return parQAPs

end CruzarPosicion
```

CRUCE PMX

El cruce PMX consiste en lo siguiente. Se elige una subcadena central y se establece una correspondencia por posición entre las asignaciones contenidas en ellas.Cada hijo contiene la subcadena central de uno de los padres y el mayor número posible de asignaciones en las posiciones definidas por el otro padre. Cuando se forma un ciclo, se sigue la correspondencia fijada para incluir una asignación nueva.

En pseudocódigo, el cruce PMX funciona de la siguiente forma:

```
pair<QAP,QAP> CruzarPMX( QAP crom1, QAP crom2 )

    genesCrom1 ← crom1.LocalizacionesEnUnidades
    genesCrom2 ← crom2.LocalizacionesEnUnidades

    corteDcha ← Random( 0, numGenes )
    corteIzq ← Random( 0, corteDcha )

    for i=0 to crom1.numGenes
        if i ≥ corteIzq and i ≤ corteDcha
            genesHijo1.Add( genesCrom2[i] )
            genesHijo2.Add( genesCrom1[i] )
            cadenaCentralH1.Add( genesCrom2[i] )
            cadenaCentralH2.Add( genesCrom1[i] )
        endif
        else
            genesHijo1.Add( -1 )
            genesHijo2.Add( -1 )
        endelse
    endfor

    for i=0 to crom1.numGenes
        if i < corteIzq or i > corteDcha

            genesHijo1[i] = genesCrom1[i]
            while cadenaCentralH1.Contains( genesHijo1[i] )
                genesHijo1[i] = cadenaCentralH2[ index of genesHijo1[i] in cadenaCentralH1 ]
            endwhile

            genesHijo2[i] = genesCrom2[i]
            while cadenaCentralH2.Contains( genesHijo2[i] )
                genesHijo2[i] = cadenaCentralH1[ index of genesHijo2[i] in cadenaCentralH2 ]
            endwhile
        endif
    endfor


    parQAPs ← new QAP(genesHijo1), new QAP(genesHijo2)

    return parQAPs
```

MUTACIÓN

Una vez hemos cruzado nuestros cromosomas, obtenemos nuestra población intermedia.

Para obtener nuestra población de hijos, sólo nos queda mutar una parte de esta población intermedia, bajo una probabilidad. El modo en el que esta probabilidad actúa depende del algoritmo, y se explicará más adelante.

La mutación de un cromosoma consiste simplemente en intercambiar dos genes de lugar. Es decir, intercambiar dos localizaciones de posición.

El método para intercambiar las posiciones i y j es el siguiente.

```
Intercambiar( pos1, pos2, genes )
        temp ← genes[pos1]
        genes[pos1] ← genes[pos2]
        genes[pos2] ← temp
end Intercambiar
```

REEMPLAZAMIENTO

Una vez completemos la etapa de mutación, tendremos nuestra población de hijos. Esta población, reemplazará a la población inicial, dependiendo del algoritmo, en parte o completamente.

Cuando la población de hijos haya reemplazado a la de padres, volveremos a empezar el ciclo. De esta forma, en cada generación tendremos mejores soluciones que en la anterior.

EXPLICACIÓN DE LOS ALGORITMOS ESPECÍFICAMENTE: AGG

Los distintos algoritmos que vamos a explicar son los siguientes:

- Algoritmos Genéticos Generacionales
- Algoritmos Genéticos Estacionarios
- Algoritmos Meméticos

Los AGG, al igual que el resto de algoritmos, se componen de cuatro etapas: Selección, cruce, mutación y reemplazamiento.

SELECCIÓN

La selección de padres que se hace en los AGG consiste en realizar tantos torneos binarios entre dos cromosomas aleatorios, como número de cromosomas tengamos en la población. De esta manera, el número de cromosomas de la población de padres es igual al número de cromosomas de la población inicial.

En pseudocódigo, la selección es así:

```
for i=0 to numCromosomas
        rand1 ← Random( 0, numCromosomas )
        rand2 ← Random( 0, numCromosomas )

        while( rand1 = rand2 ) rand2 ← Random( 0, numCromosomas )

        poblacionPadres.Add( TorneoBinario( rand1, rand2 ) )
endfor
```

CRUCE

Ya tenemos la población de padres, con el mismo número de cromosomas que la población inicial.

Ahora debemos cruzar estos padres. En los Algoritmos Genéticos Generacionales cruzamos con una probabilidad de mutación, pero como el coste de generar un aleatorio para cada cromosoma es computacionalmente alto, estimamos el número de cruces aproximado, y cruzamos ese número de cromosomas. En pseudocódigo, el cruce se realiza así:

```
numCruces = probCruce * numCromosomas

for ( i=0, i<numCruces, i+=2 )
        if pmx cruzados ← CruzarPmx( i, i+1 )
        else if pos cruzados ← CruzarPosicion( i, i+1 )
        poblacionIntermedia.Add( cruzados )
endfor

for i=numCruces to numCromosomas
        poblacionIntermedia.Add( poblacionPadres[i] )
endfor
```

Como la aleatoriedad de cromosomas ya se ha aplicado en la selección, podemos cruzar siguiendo un orden, en este caso, cruzamos cada cromosoma de la población de padres con el cromosoma siguiente.

MUTACIÓN

Ya tenemos la población intermedia con los cromosomas cruzados. Para obtener la población de hijos, nos queda mutar estos cromosomas.

Al igual que con los cruces, la mutación de un gen se somete a una probabilidad. El pseudocódigo de la mutación es el siguiente.

```
numMutaciones = probMutacion * numCromosomas * numGenes

for i=0 to numMutaciones
        c ← Random( 0, numCromosomas )
        g1 ← Random( 0, numGenes )
        g2 ← Random( 0, numGenes )
        while( g1 = g2 ) g2 ← Random( 0, numGenes )

        Intercambiar( g1, g2, poblacionIntermedia[c] )
        poblacionIntermedia[c].CalcularNuevoCoste
endfor

poblacionHijos = poblacionIntermedia
```

REEMPLAZAMIENTO

Una vez obtenida la población de hijos, nos queda reemplazar por la población inicial. En los AGG, la totalidad de la población de hijos reemplaza a la población inicial por completo, excepto en 1 cromosoma:

Para conservar el elitismo, conservamos el mejor cromosoma de la población inicial, y si es mejor que el peor cromosoma de la población de hijos, lo sustituimos.

En pseudocódigo, el reemplazamiento se puede describir así:

```
peorHijo ← poblacionHijos[0]
mejorInicial ← poblacionInicial[0]

for i=0 to poblacionHijos.Length
        if  peorHijo.Coste > poblacionHijos[i].Coste
                peorHijo ← poblacionHijos[i]
        endif

        if mejorInicial.Coste < poblacionInicial[i].Coste
                mejorInicial ← poblacionInicial[i]
        endif
endfor

poblacionHijos[ index of peorHijo ] ← mejorInicial
poblacionInicial ← poblacionHijos
```

Una vez hemos completado la fase de reemplazamiento, podemos volver a comenzar el ciclo.

EXPLICACIÓN DE LOS ALGORÍTMICOS EXPLÍCITAMENTE: AGE

Al igual que con los AGG, se van a explicar las peculiaridades que tienen los AGE en sus cuatro etapas.

SELECCIÓN

En la etapa de selección, no vamos a seleccionar el mismo número de padres que de cromosomas iniciales. Sólo vamos a seleccionar dos padres. Para ello, haremos dos veces el torneo binario con cuatro cromosomas aleatorios de la población inicial.

En pseudocódigo, la selección es así:

```
for i=1 to 2
        rand1 ← Random( 0, numCromosomas )
        rand2 ← Random( 0, numCromosomas )

        while( rand1 = rand2 ) rand2 ← Random( 0, numCromosomas )

        poblacionPadres.Add( TorneoBinario( rand1, rand2 ) )
endfor
```

Como vemos, a la población de padres sólo se agregan dos cromosomas.

CRUCE

En los AGE, el cruce no lo realizamos bajo una probabilidad, lo realizamos siempre. Al sólo tener dos padres, sólo cruzamos una vez.

```
if pmx cruzados ← CruzarPmx( i, i+1 )
else if pos cruzados ← CruzarPosicion( i, i+1 )

poblacionIntermedia.Add( cruzados )
```

MUTACIÓN

La siguiente fase es la mutación. En este caso, al tener sólo dos cromosomas en la población intermedia, no vamos a estimar el número aproximado de mutaciones a realizar. Simplemente mutaremos estos cromosomas bajo una probabilidad.

```
max ← 1 / (probMutacion * numGenes * 2 )

for i=1 to 2
        c ← Random( 1, max )
        if c=1
                g1 ← Random( 0, numGenes )
                g2 ← Random( 0, numGenes )
                while( g1 = g2 ) g2 ← Random( 0, numGenes )

                Intercambiar( g1, g2, poblacionIntermedia[0] )
                poblacionIntermedia[0].CalcularNuevoCoste
        endif
endfor

poblacionHijos ← poblacionIntermedia
```

REEMPLAZAMIENTO

Ya hemos mutado y cruzado estos dos padres, obteniendo dos hijos. Ahora falta el reemplazamiento.

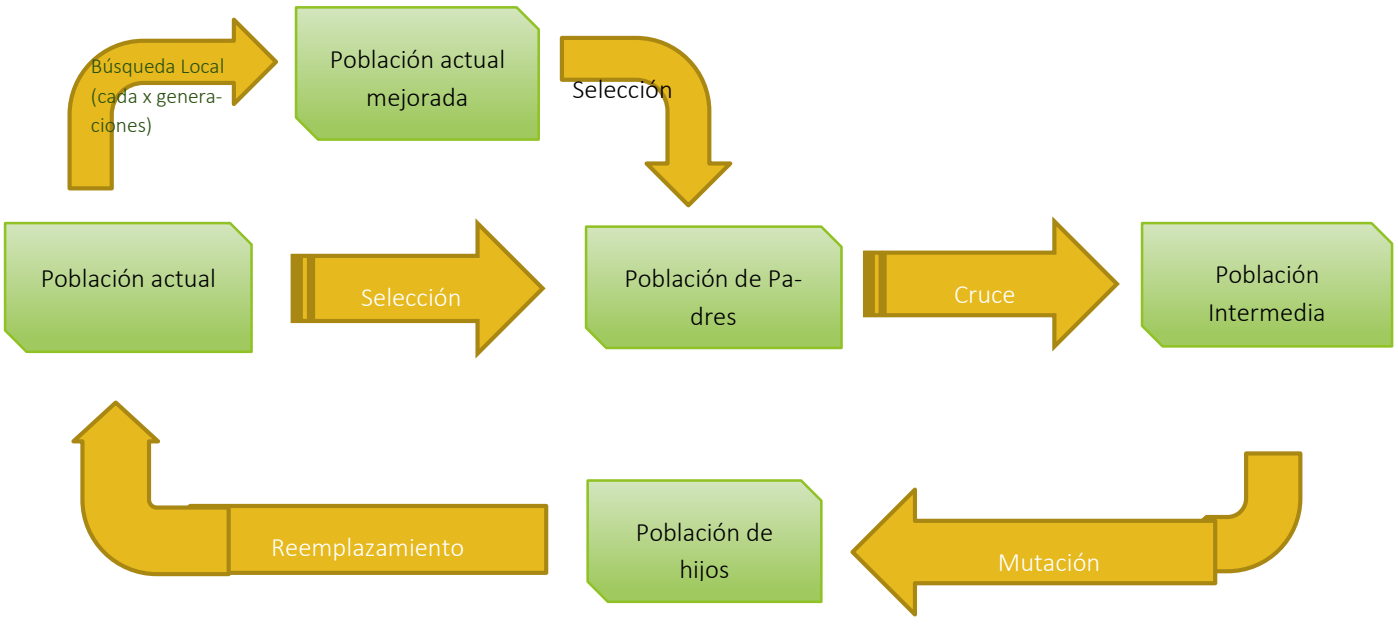
El reemplazamiento consiste en sustituir los peores cromosomas de la población inicial por estos dos nuevos cromosomas, siempre y cuando estos últimos sean mejores.En pseudocódigo, el reemplazamiento se puede llevar a cabo así:

```
peorInicial ← BuscarPeor( poblacionInicial )
segundoPeorInicial ← BuscarSegundoPeor( poblacionInicial )

if poblacionHijos[0].Coste < peorInicial.Coste and poblacionHijos[1].Coste < segundoPeorInicial.Coste
        poblacionInicial[ index of peorInicial ] ← poblacionHijos[0]
        poblacionInicial[ index of segundoPeorInicial ] ← poblacionHijos[1]
endif

else
        if poblacionHijos[0].Coste < peorInicial.Coste
                poblacionInicial[ index of peorInicial ] ← poblacionHijos[0]
        endif
        else if poblacionHijos[0].Coste < segundoPeorInicial.Coste
                poblacionInicial[ index of segundoPeorInicial ] ← poblacionHijos[0]
        endif
        else if poblacionHijos[1].Coste < peorInicial.Coste
                poblacionInicial[ index of peorInicial ] ← poblacionHijos[0]
        endif
        else if poblacionHijos[1].Coste < segundoPeorInicial.Coste
                poblacionInicial[ index of segundoPeorInicial ] ← poblacionHijos[0]
        endif
endelse
```

EXPLICACIÓN DE LOS ALGORÍTMOS ESPECÍFICA: AM

Los algoritmos meméticos son algoritmos genéticos generacionales en los cuales cada x generaciones, se aplica búsqueda local sobre todos o un porcentaje de la población, con un máximo de exploración de 400 vecinos en este caso.

El esquema evolutivo de los algoritmos meméticos es el siguiente:



Excepto la etapa en la que aplicamos búsqueda local, el resto de etapas son exactamente iguales a un Algoritmo Genético Generacional.

El AM tiene tres variantes: Búsqueda local con toda la población inicial, búsqueda local un porcentaje de la población inicial, y búsqueda local con el mejor porcentaje de población inicial.

BÚSQUEDA LOCAL

En pseudocódigo, podemos definir esta etapa de la siguiente manera:

```
if mejorPorcentaje
        poblacionInicial.OrdenarPorCoste( )
endif

for i=0 to poblacionInicial.Length*porcentajeMeme
        poblacionInicial[i] ← BusquedaLocal( poblacionInicial[i] )
endfor
```

Recordemos que el método de búsqueda local es el siguiente

```
hayMejora ←true

for i=0 to n
    dlb[i] ←false
endfor

while hayMejora
    hayMejora ←false
    for i=0 to n
        if dlb[i] = false
            improveFlag ←false
            for j=0 to n
                if i≠j
                    if CosteIntercambio(i,j) < 0
                        intercambiar(i,j)
                        dlb[i] ←false
                        dlb[j] ←false
                        improveFlag ←true
                        hayMejora ←true
                    endif
                endif
            endfor
        endif
    endfor
endwhile
```

Y el método que calcula el coste después de un intercambio, de forma factorizada, para ahorrarnos tiempo de computación, es el siguiente.

```
costeIntercambio r, s
    coste ←0
    for k=0 to n
        if k ≠r and k ≠s
            coste←coste + flujos[r][k]*(distancias[solucion[s]][solucion[k]] -
distancias[solucion[r]][solucion[k]])
            coste←coste + flujos[s][k]*(distancias[solucion[r]][solucion[k]] -
distancias[solucion[s]][solucion[k]])
            coste←coste + flujos[k][r]*(distancias[solucion[k]][solucion[s]] -
distancias[solucion[k]][solucion[r]])
            coste←coste + flujos[k][s]*(distancias[solucion[k]][solucion[r]] -
distancias[solucion[k]][solucion[s]])
        endif
    endfor

    return coste
end
```

CALCULAR LA BONDAD DE UN ALGORITMO

Para calcular la bondad de un algoritmo, vamos a usar la media de desviaciones típicas de los costes obtenidos por este algoritmo respecto los mejores costes conocidos. Cuanto menor sea la desviación típica, mejor es el algoritmo.

Para calcular cada coste, usaremos una base de datos con archivos de distintos tamaños.

La media de desviaciones típicas se calcula de la siguiente forma.

```
desviacion ←0
for i=0 to n
   desviacion ←desviacion + 100*((costes[i]-mejoresCostes[i])/mejoresCostes[i])
endfor

desviacion ←desviacion/n
```

Donde costes es un vector que contiene los costes obtenidos por el algoritmo, y mejoresCostes es el vector que contiene los mejores costes conocidos.

DESARROLLO DE LA PRÁCTICA

El desarrollo de esta práctica se ha llevado a cabo en C#. Hay varios archivos fuente: El lector de los archivos .dat; el código del QAP, que contiene las matrices y la solución, así como los algoritmos para encontrar soluciones; el archivo de Búsqueda Local, el archivo AG, que contiene la clase que lleva a cabo los algoritmos genéticos, los archivos AGG, AGE y AM, que contienen clases heredadas de la clase AG con constructores específicos para cada algoritmo, y el archivo Program, el cual tiene el programa que ejecuta los algoritmos genéticos a partir de unos datos de entrada.

No se han usado frameworks ni código externo.

He usado la semilla 1, aunque en los resultados obtenidos he usado los algoritmos para todos los archivos de forma seguida, por lo que los resultados nos saldrán distintos si ejecutamos el programa con un único archivo.

EXPERIMENTOS Y ANÁLISIS DE LOS RESULTADOS

Los resultados de aplicar los algoritmos son los siguientes. Hemos puesto como límite de ejecución, 50000 llamadas a la función objetivo.

El algoritmo AGG, con 50 cromosomas por población, una probabilidad de cruce de 0.7, una probabilidad de mutación de 0.001 y el operador de cruce de posición y pmx, ha obtenido los siguientes resultados

AGG-Pos

	Coste	Mejor coste	Desv.	Tiempo
chr22a.dat	6658	6156	8,154648	00:00:00.5806088
chr22b.dat	6890	6194	11,23668	00:00:00.5962901
chr25a.dat	5760	3796	51,73868	00:00:00.6911432
esc128.dat	72	64	12,5	00:00:16.7261124
had20.dat	6964	6922	0,6067581	00:00:00.5180091
lipa60b.dat	3061127	2520135	21,46679	00:00:03.6601425
lipa80b.dat	9581150	7763962	23,40542	00:00:06.4389781
nug28.dat	5516	5166	6,77507	00:00:00.9134464
sko81.dat	94440	90998	3,782501	00:00:06.5681032
sko90.dat	122062	115534	5,650284	00:00:08.1003315
sko100a.dat	160826	152002	5,805183	00:00:09.9223731
sko100f.dat	157604	149036	5,748947	00:00:09.9969672
tai100a.dat	22286072	21052466	5,859673	00:00:09.9270498
tai100b.dat	1297232717	1185996137	9,379173	00:00:10.0794470
tai150b.dat	591297080	498896643	18,52096	00:00:22.3872256
tai256c.dat	46895844	44759294	4,773422	00:01:04.7788979
tho40.dat	267522	240516	11,22836	00:00:01.7028475
tho150.dat	8976596	8133398	10,3671	00:00:22.1708595
wil50.dat	49780	48816	1,974762	00:00:02.6118190
wil100.dat	278742	273038	2,089088	00:00:09.9579921
			11,053175	

AGG-Pmx

Coste	Mejor coste	Desv.	Tiempo
chr22a.dat	6964	6156	13,1254	00:00:00.6547746
chr22b.dat	7964	6194	28,57603	00:00:00.6318429
chr25a.dat	6064	3796	59,7471	00:00:00.7826457
esc128.dat	78	64	21,875	00:00:16.2295325
had20.dat	6966	6922	0,6356506	00:00:00.5285469
lipa60b.dat	3048451	2520135	20,96379	00:00:03.7571246
lipa80b.dat	9552912	7763962	23,04172	00:00:06.5439274
nug28.dat	5354	5166	3,639183	00:00:00.9384345
sko81.dat	94564	90998	3,91877	00:00:06.6788458
sko90.dat	120360	115534	4,177124	00:00:08.3035212
sko100a.dat	157742	152002	3,776268	00:00:10.1390702
sko100f.dat	155148	149036	4,101021	00:00:10.0265331
tai100a.dat	22300354	21052466	5,927521	00:00:10.1066679
tai100b.dat	1,3E+09	1185996137	9,609818	00:00:10.1933811
tai150b.dat	538414151	498896643	7,920982	00:00:22.5435438
tai256c.dat	45854708	44759294	2,447342	00:01:05.1901418
tho40.dat	256674	240516	6,718056	00:00:01.7467643
tho150.dat	8632516	8133398	6,136642	00:00:22.4349452
wil50.dat	50384	48816	3,212059	00:00:02.6722922
wil100.dat	278328	273038	1,937462	00:00:10.0755857
			11,57434693	

El algoritmo AGE, con 50 cromosomas por población, una probabilidad de mutación de 0.001 y el operador de cruce de posición y pmx, ha obtenido los siguientes resultados

AGE-Pos

	Coste	Mejor coste	Desv.	Tiempo
chr22a.dat	9644	6156	56,66017	00:00:00.7940313
chr22b.dat	8578	6194	38,48886	00:00:00.7941073
chr25a.dat	14078	3796	270,8641	00:00:00.9297331
esc128.dat	106	64	65,625	00:00:16.1332228
had20.dat	7292	6922	5,345276	00:00:00.6681796
lipa60b.dat	3135885	2520135	24,43322	00:00:03.7899200
lipa80b.dat	9669559	7763962	24,54413	00:00:06.5190434
nug28.dat	5996	5166	16,06659	00:00:01.0828228
sko81.dat	97400	90998	7,035316	00:00:06.6513587
sko90.dat	122378	115534	5,923798	00:00:08.1601732
sko100a.dat	161288	152002	6,109131	00:00:09.9773960
sko100f.dat	161170	149036	8,141655	00:00:09.9990054
tai100a.dat	22736768	21052466	8,000496	00:00:10.0157674
tai100b.dat	1325358448	1185996137	11,75066	00:00:10.1237819
tai150b.dat	547108282	498896643	9,663658	00:00:22.3550075
tai256c.dat	45988880	44759294	2,747101	00:01:04.5562397
tho40.dat	294282	240516	22,35444	00:00:01.8696081
tho150.dat	8625662	8133398	6,052376	00:00:22.2139703
wil50.dat	51742	48816	5,993935	00:00:02.7438715
wil100.dat	282570	273038	3,491089	00:00:09.9662353
			29,96455	

AGE-Pmx

	Coste	Mejor coste	Desv.	Tiempo
chr22a.dat	8666	6156	40,77322	00:00:00.8079893
chr22b.dat	8616	6194	39,10236	00:00:00.7903406
chr25a.dat	11642	3796	206,6913	00:00:00.9325508
esc128.dat	140	64	118,75	00:00:16.1454664
had20.dat	7202	6922	4,045074	00:00:00.7107916
lipa60b.dat	3159577	2520135	25,37332	00:00:03.8906346
lipa80b.dat	9819418	7763962	26,47432	00:00:06.5989230
nug28.dat	5776	5166	11,80798	00:00:01.0994594
sko81.dat	102126	90998	12,22884	00:00:06.7468919
sko90.dat	126740	115534	9,69931	00:00:08.2113186
sko100a.dat	166362	152002	9,447243	00:00:10.0535078
sko100f.dat	161906	149036	8,635498	00:00:10.0599914
tai100a.dat	23015868	21052466	9,326225	00:00:10.1369672
tai100b.dat	1,436E+09	1185996137	21,09041	00:00:10.2856059
tai150b.dat	549363090	498896643	10,11561	00:00:22.4717572
tai256c.dat	46029484	44759294	2,837822	00:01:04.5919633
tho40.dat	292582	240516	21,64762	00:00:01.9033911
tho150.dat	9030870	8133398	11,0344	00:00:22.2508270
wil50.dat	51872	48816	6,260246	00:00:02.7949111
wil100.dat	285712	273038	4,641846	00:00:10.1103607
			29,9991322	

Como vemos, por muy poco, el operador de cruce es mejor en estos casos. Esto es muy relativo, y depende muchísimo de la semilla de aleatoriedad, pues en distintas ejecuciones, la desviación de un caso puede pasar de 10 a 2, por ejemplo.

Vemos también que el AGE es mucho peor que el AGG. Esto, según mi experiencia, es por que el AGE converge demasiado rápido, obteniendo muy pronto una población de cromosomas idénticos, por lo que los cruces dejan de tener valor y la población depende para mejorar exclusivamente de las mutaciones, lo que hace que el AGE no alcance buenas soluciones en comparación con el AGG. La ventaja del AGE respecto al AGG es que muchísimo más rápido (fijarse en el caso de tai256c).

Vamos ahora a ver cómo funcionan los AM, con 10 cromosomas por población, una probabilidad de cruce de 0.7, una probabilidad de mutación de 0.001 y llamadas a la Búsqueda Local cada 10 generaciones.

AM-Pos-(10,1)

chr22a.dat	6194	6156	0,6172867	00:00:06.5105188
chr22b.dat	6362	6194	2,712303	00:00:05.9064264
chr25a.dat	3796	3796	0	00:00:08.9036641
esc128.dat	64	64	0	00:20:18.6098114
had20.dat	6922	6922	0	00:00:05.0806438
lipa60b.dat	2985415	2520135	18,4625	00:02:19.1487868
lipa80b.dat	9369769	7763962	20,68283	00:07:24.7301140
nug28.dat	5226	5166	1,161438	00:00:12.1496815
sko81.dat	91472	90998	0,5208893	00:04:04.8828970
sko90.dat	116278	115534	0,6439667	00:05:51.6924517
sko100a.dat	152834	152002	0,5473633	00:04:44.0223325
sko100f.dat	151710	149036	1,794197	00:03:02.4459022
tai100a.dat	21538400	21052466	2,308212	00:07:32.7816169
tai100b.dat	1263349480	1185996137	6,522224	00:02:35.3160739
tai150b.dat	509424930	498896643	2,110313	00:05:19.1900770
tai256c.dat	44838404	44759294	0,1767426	01:29:17.7274516
tho40.dat	249558	240516	3,759415	00:00:32.3152688
tho150.dat	8366440	8133398	2,865242	00:06:28.7883903
wil50.dat	48848	48816	0,0655518	00:01:05.8265799
wil100.dat	275446	273038	0,8819275	00:02:21.6127443
			3,2916201	

Am-Pos-(10,0.1mej)

chr22a.dat	6426	6156	4,385963	00:00:01.2234078
chr22b.dat	6292	6194	1,582176	00:00:01.3535655
chr25a.dat	4934	3796	29,97893	00:00:01.7154341
esc128.dat	64	64	0	00:02:12.8965503
had20.dat	6922	6922	0	00:00:00.9712661
lipa60b.dat	2994415	2520135	18,81963	00:00:17.4640467
lipa80b.dat	9354854	7763962	20,49072	00:00:38.7211779
nug28.dat	5560	5166	7,626793	00:00:02.2225306
sko81.dat	92134	90998	1,248383	00:00:42.2821441
sko90.dat	116948	115534	1,223885	00:00:55.1245035
sko100a.dat	153112	152002	0,7302551	00:01:08.0964834
sko100f.dat	150928	149036	1,269493	00:01:07.2683597
tai100a.dat	21529512	21052466	2,265991	00:01:04.8420769
tai100b.dat	1214268201	1185996137	2,38382	00:01:09.0018824
tai150b.dat	504881028	498896643	1,199524	00:03:39.8875080
tai256c.dat	44845314	44759294	0,1921768	00:17:01.5615998
tho40.dat	244668	240516	1,726288	00:00:05.8484790
tho150.dat	8237884	8133398	1,284653	00:03:51.1507102
wil50.dat	49418	48816	1,2332	00:00:10.3930365
wil100.dat	274640	273038	0,586731	00:01:11.7233163
			4,9114306	

AM-Pos(10,0.1)

chr22a.dat	6426	6156	4,385963	00:00:01.2234078
chr22b.dat	6292	6194	1,582176	00:00:01.3535655
chr25a.dat	4934	3796	29,97893	00:00:01.7154341
esc128.dat	64	64	0	00:02:12.8965503
had20.dat	6922	6922	0	00:00:00.9712661
lipa60b.dat	2994415	2520135	18,81963	00:00:17.4640467
lipa80b.dat	9354854	7763962	20,49072	00:00:38.7211779
nug28.dat	5560	5166	7,626793	00:00:02.2225306
sko81.dat	92134	90998	1,248383	00:00:42.2821441
sko90.dat	116948	115534	1,223885	00:00:55.1245035
sko100a.dat	153112	152002	0,7302551	00:01:08.0964834
sko100f.dat	150928	149036	1,269493	00:01:07.2683597
tai100a.dat	21529512	21052466	2,265991	00:01:04.8420769
tai100b.dat	1214268201	1185996137	2,38382	00:01:09.0018824
tai150b.dat	504881028	498896643	1,199524	00:03:39.8875080
tai256c.dat	44872918	44759294	0,2538528	00:17:28.3069778
tho40.dat	242062	240516	0,6427841	00:00:05.5161030
tho150.dat	8328428	8133398	2,397896	00:04:18.3714378
wil50.dat	49838	48816	2,093575	00:00:09.7829878
wil100.dat	279580	273038	2,396004	00:01:22.6277890
			5,0494838	

Como vemos, añadir BL a nuestros algoritmos genéticos mejora muchísimo los resultados, encontrando incluso varios óptimos. Vemos que, aunque por muy poco, como era de esperar el algoritmo que aplica BL sobre el total de la población obtiene los mejores resultados, aunque tarda muchísimo más (el caso más acusado el de tai256c, con casi 1h30min). Aplicando BL sólo a un 10% o al 10% mejor de la población obtenemos unos resultados muy poco peores, pero disminuyendo el tiempo de computación una barbaridad. Esto es porque BL es muy potente, pero requiere mucho tiempo de cómputo.

Por último, comparando con los algoritmos Greedy y BL, vemos que todos mejoran a Greedy, pero sólo los AM mejoran la BL. Esto es relativo, ya que con los AG sólo hemos hecho una única ejecución, mientras que con BL hicimos la media de varias, y dependiendo de la suerte, podemos tener mucho mejores (o peores) soluciones en los AG.

Media desv greedy	54.98485
Media desv BL	9.233969