

Práctica 1 QAP

2016/2017

Algoritmos: Greedy, BL, AGG-Pos, AGG-OX, AGE-Pos,
AGE-OX, Memetico1.0, Memetico0.1, Memetico0.1Mejor

Antonio Manuel Milán Jiménez

antoniomj@correo.ugr.es

Grupo 3 Lunes

índice

Descripción del problema-----	pag 3
Representación de las soluciones-----	pag 3
Función objetivo-----	pag 3
Operador de generación de vecino/mutación-----	pag 3
Factorización de la búsqueda local-----	pag 4
Generación de soluciones aleatorias-----	pag 4
Selección en los algoritmos genéticos-----	pag 5
Cruce en los algoritmos genéticos-----	pag 6
Mutación en los algoritmos genéticos-----	pag 7
Mascara “don’t look bits” y exploración en BL-----	pag 7
Evolución y reemplazamiento en AGs-----	pag 8
Búsqueda en los algoritmos memeticos-----	pag 8
Algoritmo Greedy-----	pag 9
Procedimiento y manual-----	pag 9
Análisis-----	pag 10

Descripción del problema

El problema de la asignación cuadrática, QAP, nos presenta el problema de determinar la asignación óptima de una serie de unidades en una serie de localizaciones que tenga un costo mínimo. Se nos proporciona las distancias entre las localizaciones y los flujos entre las unidades. Con esta información se nos pide construir diferentes algoritmos con diferentes técnicas con el fin de obtener buenas soluciones en un tiempo razonable.

Los algoritmos que hemos construido para resolver el problema han sido un algoritmo “greedy”, muy rápido pero con mala soluciones como veremos más adelante; una búsqueda local con técnica de “primero el mejor”, un algoritmo genético con dos tipos de selección: generacional y estacionario, y dos tipos de cruce: por posición y OX; y finalmente un algoritmo memético con 3 formas de incluir la búsqueda local: aplicando búsqueda local a toda la población, aplicarla aleatoriamente a un número de individuos y aplicarla a los mejores de la población.

Representación de las soluciones:

Una solución al problema se representa como una permutación de un vector desde el 1 hasta el número de unidades. Así, en cada uno de estos vectores, soluciones, la posición del vector se corresponde con el número de la unidad y el contenido de dicha posición en el vector es la localización.

La valoración de una solución se hace a partir de las matrices de distancia y flujo de cada caso, que nos indica la distancia entre todas las localizaciones y el flujo entre todas las unidades. Dado que intentamos minimizar este valor, diremos que una solución es mejor que otra cuando su valoración sea menor.

Función objetivo:

Este es el pseudocódigo de la función objetivo que se encarga de calcular la valoración de una solución a partir de las matrices cómo hemos descrito anteriormente:

```
(for i < 0...nºunidades)
    (for j < 0...nºunidades)
        a=solución[i]-1
        b=solución[j]-1
        valor += flujo[i][j] + distancia[a][b]
```

Operador de generación de vecino/mutación:

Para crear un vecino en la búsqueda local o simular una mutación en el genético, básicamente seleccionaremos dos unidades y las intercambiaremos. El qué unidades seleccionar lo describiremos más adelante con la máscara de “Don’t look bits” pero este es el pseudocódigo referente a la generación/mutación:

```

valoración = evaluacionFactorial(pos1, pos2, solución, solución.valor)
if(valoración < solución.valor)
    Intercambio(solución, pos1, pos2)
    solución.valor = valoración

```

“Intercambio” se encarga de intercambiar en la solución las 2 posiciones ya que ahora es nuestra nueva mejor solución.

En el algoritmo genético, la mutación sigue un esquema similar aunque no se hace comparaciones de valoraciones ya que no importa que sea mejor o peor la mutación para que mute.

Factorización de la búsqueda local:

Hemos visto en el anterior operador que se hace una valoración factorizada al generar un vecino. Esto se hace porque teniendo ya la valoración de la solución, sería inefficiente volver a calcular el valor del vecino cuando sólo han cambiado dos posiciones. Por eso, para que resulte mas eficiente nos apoyamos en la valoración de la solución para calcular el nuevo valor en menos tiempo. Aquí tenemos el pseudocódigo:

```

a= solucion[pos1]-1
b=solucion[pos2]-1
for(i ← 0...nºunidades)
    c=solucion[i]-1
    valor += flujo[pos1][i]*(distancia[b][c]-distancia[a][c]) +
    flujo[pos2][i]*(distancia[a][c]-distancia[b][c]) +
    flujo[i][pos1]*(distancia[c][b]-distancia[c][a]) +
    flujo[i][pos2]*(distancia[c][a]-distancia[c][b])
valor += solucion.valor

```

Generación de soluciones aleatorias:

```

solucion ← [1...nºunidades]
for(i ← 0...nºunidades)
    r=randInt(0,nºunidades-1)
    Intercambio(solucion, i, r)

```

Selección en los algoritmos genéticos:

Hemos utilizado dos tipos de selección: generacional donde seleccionamos tantos padres como individuos en la población, y estacionario donde únicamente seleccionamos 2 padres:

Selección generacional:

```
for(i < 0...tam_poblacion)
    n1 = randint(0,tam_poblacion-1)
    do
        n2 = randint(0,tam_poblacion-1)
    while(n1==n2)
    ganador = torneoBinario(n1,n2)
    padres.add(población(ganador))
```

Selección estacionario:

```
for(i < 0...2)
    n1 = randint(0,tam_poblacion-1)
    do
        n2 = randint(0,tam_poblacion-1)
    while(n1==n2)
    ganador = torneoBinario(n1,n2)
    padres.add(población(ganador))
```

“torneoBinario” compara los dos individuos de las posiciones n1 y n2, y devuelve la posición del mejor.

Cruce en los algoritmos genéticos:

Hemos implementado cruce por posición y el cruce OX:

Cruce por posición:

```
random ← random([1...tam_cromosoma])
for(i ← 0...tam_cromosoma)
    if(cromosoma1[i] == cromosoma2[i])
        a=cromosoma1[i]
        b=random[i]
        if(a != b)
            c = hijo.buscar(a)
            if(c != -1) //encontrado
                hijo.set(c,b)
            else
                c = random.buscar(a)
                random.set(c,b)
                random.set(l,a)
            hijo.añadir(a)
        else
            hijo.añadir(random[l])
```

Cruce OX

```
min= tam_Crom/2 – tam_Crom/4
max= tam_Crom/2 + tam_Crom/4
hijo ← [-1,-1,..,cromosoma1[min]...cromosoma1[max],..-1,-1]
v ← [cromosoma1[0]...cromosoma1[min -1],
      cromosoma1[max],...,cromosoma1[tam_Crom-1]]

for(i ← 0...v.size())
    orden.añadir(cromosoma2.buscar(v[i])) //Orden del cromosoma 2

for(i ← 0...tam_Crom)
    a = orden.buscar(i)
    if(a != -1)
        max=max%tam_Crom
        hijo.set(v[a],max)
        max++
```

Mutación en los algoritmos genéticos:

```
while(n<n_mutaciones)
    gen1 = randint(1,tam_Crom)-1
```

```

do
    gen2 = randInt(1,tam_Crom)-1
    while(gen1 == gen2)
        cromMuta = randInt(1,hijos.size())-1
        hijo = hijos[cromMuta]
        Intercambio(hijo,gen1,gen2)
    n++

```

Máscara “don’t look bits” y exploración de la búsqueda local:

Utilizamos la máscara “don’t look bits” para generar los nuevos vecinos de la solución actual. Tenemos inicialmente un vector de ceros, uno por cada unidad. Si vemos que todos los vecinos posibles generados al intercambiar una unidad no mejoran a la solución actual, en el vector ponemos en su posición un 1, indicando que no se han conseguido mejora con él y que no se tenga en cuenta para generar futuros vecinos.

Para evitar que en la búsqueda “primero el mejor” le demos prioridad a los vecinos de las primeras posiciones, barajamos un vector de [0...nºunidades-1] que nos indicará en cada iteración la próxima unidad con la que probar los intercambios y generar los vecinos. Este es el pseudocódigo de la exploración de la búsqueda local utilizando la máscara de “don’t look bits”:

```

aux=random[0...nºunidades]
bits=[0...0]
for(i ← 0...nºunidades && evaluaciones<max)
    pos=aux[i]
    if(bits[pos]==0)
        for(j ← 0...nºunidades && !encontrado)
            if(j!=pos)
                valor= evaluacionFactorial(solucion, j, pos,
valor_ini)
                evaluaciones++
                if(valor < valor_ini)
                    encontrado=true
                    i=-1
                    Intercambiar(solucion,j,pos)
                    valor_ini = valor
                    aux=random[0...nºunidades]
                if(!encontrado)
                    bits[pos]=1

```

Esquema de evolución y reemplazamiento de los algoritmos genéticos

```

while(iteraciones<max)
    padres=selección() //Generacional o estacionaria

```

```

hijos = cruce(padres) //Posicion o OX
Mutar(hijos)
for(i ← 0...hijos.size())
    hijo[i].valor = valorar(hijo[i])

iteraciones += hijos.size()
//A partir de aquí hay dos formas de realizar el reemplazamiento:
//Si ha sido una selección generacional;
for(i ← hijos.size()...tam_poblacion)
    hijos.add(padres[i])

OrdenarPoblacion(hijos)
If(hijos.contiene(población[0]))
    hijos.set(hijos.size()-1,población[0])
OrdenarPoblacion(hijos)
población=hijos

//Si ha sido una selección estacionaria
OrdenarPoblacion(hijos)
If(hijos[0].valor < población[tam_Pob-2])
    población[tam_pob-2] = hijos[0]
    If(hijos[1].valor < población[tam_Pob-1])
        población[tam_pob-1] = hijos[1]
    else(hijos[0].valor < población[tam_Pob-1])
        población[tam_pob-1] = hijos[0]

```

Búsqueda en los algoritmos meméticos:

```

población = GenerarPoblInicialGenetico()
while(iteraciones < max)
    //Si aplicamos BL a toda la población
    for(i ← 0...tam_Pob)
        BL.setSolucion(población[i])
        solucion = InicioBL()
        cromosoma = solucion
        población[i] = cromosoma

```

```

//Si aplicamos BL a un individuo aleatorio
r= randInt(0,tam_Pob-1)
BL.setSolucion(población[r])
solucion = InicioBL()

```

```

cromosoma = solucion
población[r] = cromosoma

//Si aplicamos BL al mejor individuo

BL.setSolucion(población[0])
solucion = InicioBL()
cromosoma = solucion
población[0] = cromosoma

//Tras haber aplicado la búsqueda local
población = InicioGeneticoGeneracional() //Cruce Posicion o OX
iteraciones += genético.evaluaciones

```

Controlamos que se hagan sólo 10 generaciones en el genético modificando el “max_iters” del bucle externo del genético con “max_iters=(4*tamPob)+9”.

Algoritmo Greedy:

Nuestra solución busca asignar las unidades de mayor flujo a las localizaciones más céntricas, es decir, con menos distancia a las demás localizaciones:

```

for(i < 0...nºunidades)
    potencial_flujo[i] += flujo[i][0...nºunidades]
for(i < 0...nºunidades)
    potencial_distancia[i] += flujo[i][0...nºunidades]
while(k<nºunidades)
    pos_max = potencial_flujo.maximo.posicion
    potencial_flujo[pos_max]=-1

    pos_min = potencial_distancia.minimo.posicion
    potencial_distancia[pos_min] = 1000000000
    solucion.set(pos_max,pos_min+1)
    k++

```

Procedimiento y manual de la práctica:

Para desarrollar la práctica y construir los diferentes algoritmos he utilizado como base las explicaciones y pseudocódigos proporcionados en las transparencias y seminarios de la asignatura.

Manual:

Tal y como está construido el “main” de la práctica, para cada uno de los 20 casos se lanzan los 9 algoritmos (greedy, BL, Genético generacional por posición, Genético

generacional OX, Genético estacionario por posición, Genético estacionario OX, memético toda la población, memético aleatorio y memético los mejores) 10 veces con diferentes semillas. Se calcula la media de la desviación y del tiempo para cada algoritmos en esas 10 ejecuciones.

Estos son los parámetros de construcción de los objetos relevantes:

-Evaluador(matrizFlujos,matrizDistancias)

-generadorAleatorio(semilla, mascara, prime, escala)

-Greedy(matrizFlujos, matrizDistancias, evaluador, nºunidades)

-BusquedaLocal(nºunidades, generadorAleatorio,solucionInicial, evaluador, maxIteraciones, boolean para iniciar el algoritmo)

-Genetico(nºunidades,tamPoblacion, generadorAleatorio, evaluador, tipoSeleccion, tipoCruce, probCruce, probMutar, maxIteraciones, boolean de generar población inicial)

-Memetico(nºunidades, tamPoblacion, generadorAleatorio, evaluador, tipoCruce, probCruce, probMutar, tipoMemetico*)

***1→toda la población**

***2→aleatorio**

***3→los mejores**

Si queremos utilizar la línea de comandos para establecer los parámetros de los algoritmos, los tendremos que proporcionar en el orden de: probCruceGeneracional, probCruceEstacionario, probMutar, tamañoPoblacionMemetico y tamañoPoblacionGenetico.

Si se quiere ejecutar sólo un caso, como sexto argumento pondremos el índice del caso en concreto.

Análisis:

Hemos utilizado los 20 casos proporcionados y los parámetros han sido: probabilidad de cruce de 0.7 y 1 para genético generacional y estacionario respectivamente. Una probabilidad de mutar de 0.001. Una población de 50 individuos en el genético y de 10 para el memético.

Las semillas utilizadas para las 10 iteraciones han sido:

486235468, 2162468, 486363, 8135461, 35468, 22235448, 87, 666468, 48625345, 25587.

Y estos son los resultados:

Algoritmo Greedy		
Caso	Desv	Tiempo
Chr20b	365,79	0,00
Chr22a	119,91	0,00
Els19	124,41	0,00
Esc32b	90,47	0,00
Kra30b	29,61	0,00
Lipa90b	29,05	0,00
Nug25	18,53	0,00
Sko56	19,29	0,00
Sko64	17,62	0,00
Sko72	15,64	0,00
Sko100a	13,23	0,00
Sko100b	13,49	0,00
Sko100c	14,52	0,00
Sko100d	12,52	0,00
Sko100e	13,25	0,00
Tai30b	117,72	0,00
Tai50b	71,83	0,00
Tai60a	15,81	0,00
Tai256c	120,48	0,00
Tho150	17,14	0,00

Algoritmo Busqueda Local		
Caso	Desv	Tiempo
Chr20b	37,62	0,00
Chr22a	14,35	0,00
Els19	34,21	0,00
Esc32b	26,66	0,00
Kra30b	6,67	0,00
Lipa90b	22,11	0,00
Nug25	4,63	0,00
Sko56	3,31	0,00
Sko64	2,86	0,00
Sko72	2,78	0,00

Sko100a	2,14	0,01
Sko100b	2,25	0,01
Sko100c	2,49	0,01
Sko100d	2,14	0,01
Sko100e	2,65	0,01
Tai30b	16,53	0,00
Tai50b	5,43	0,00
Tai60a	5,33	0,00
Tai256c	0,95	0,17
Tho150	2,56	0,06

Algoritmo GeneticoGenPos		
Caso	Desv	Tiempo
Chr20b	52,18	0,12
Chr22a	15,34	0,10
Els19	86,97	0,08
Esc32b	34,04	0,17
Kra30b	6,48	0,16
Lipa90b	22,58	1,00
Nug25	3,74	0,12
Sko56	4,16	0,46
Sko64	3,38	0,57
Sko72	3,45	0,69
Sko100a	2,73	1,21
Sko100b	3,10	1,21
Sko100c	3,31	1,21
Sko100d	2,70	1,21
Sko100e	3,20	1,22
Tai30b	17,30	0,16
Tai50b	7,19	0,38
Tai60a	5,50	0,51
Tai256c	1,22	5,74
Tho150	4,18	2,37

Algoritmo GeneticoGenOX		
Caso	Desv	Tiempo
Chr20b	50,35	0,07

Chr22a	18,79	0,08
Els19	83,57	0,06
Esc32b	50,71	0,13
Kra30b	10,92	0,11
Lipa90b	24,38	0,73
Nug25	16,78	0,09
Sko56	6,60	0,32
Sko64	5,76	0,40
Sko72	5,66	0,49
Sko100a	5,88	0,88
Sko100b	5,81	0,88
Sko100c	5,98	0,88
Sko100d	5,94	0,88
Sko100e	6,07	0,88
Tai30b	14,21	0,11
Tai50b	11,08	0,27
Tai60a	7,85	0,36
Tai256c	2,37	5,54
Tho150	8,00	1,94

Algoritmo GeneticoEstPos		
Caso	Desv	Tiempo
Chr20b	47,94	0,11
Chr22a	14,30	0,11
Els19	71,64	0,09
Esc32b	31,90	0,18
Kra30b	7,17	0,17
Lipa90b	24,70	1,04
Nug25	4,44	0,13
Sko56	4,41	0,47
Sko64	6,06	0,58
Sko72	6,18	0,71
Sko100a	7,28	1,22
Sko100b	7,20	1,23
Sko100c	7,98	1,23
Sko100d	7,26	1,22
Sko100e	7,81	1,23
Tai30b	14,21	0,17

Tai50b	6,32	0,39
Tai60a	7,51	0,53
Tai256c	3,64	6,82
Tho150	10,71	2,47

Algoritmo GeneticoEstOX		
Caso	Desv	Tiempo
Chr20b	57,31	0,08
Chr22a	33,88	0,09
Els19	81,45	0,07
Esc32b	53,09	0,14
Kra30b	15,02	0,13
Lipa90b	27,64	0,79
Nug25	12,65	0,10
Sko56	11,84	0,35
Sko64	8,87	0,44
Sko72	8,27	0,53
Sko100a	11,49	0,95
Sko100b	11,50	0,94
Sko100c	12,14	0,94
Sko100d	11,59	0,95
Sko100e	12,27	0,94
Tai30b	21,30	0,13
Tai50b	19,08	0,30
Tai60a	12,06	0,40
Tai256c	8,88	5,79
Tho150	15,03	2,07

Algoritmo Memetico todos 1,0		
Caso	Desv	Tiempo
Chr20b	51,88	0,09
Chr22a	15,84	0,09
Els19	26,77	0,07
Esc32b	48,57	0,16
Kra30b	10,81	0,14
Lipa90b	22,99	0,55
Nug25	7,19	0,10
Sko56	4,17	0,31

Sko64	3,68	0,33
Sko72	3,85	0,33
Sko100a	2,82	0,09
Sko100b	2,75	0,11
Sko100c	2,93	0,07
Sko100d	2,72	0,11
Sko100e	2,92	0,08
Tai30b	8,16	0,13
Tai50b	8,18	0,24
Tai60a	6,33	0,38
Tai256c	2,31	0,24
Tho150	5,80	0,09

Algoritmo Memetico 0,1 random		
Caso	Desv	Tiempo
Chr20b	43,99	0,08
Chr22a	14,21	0,08
Els19	27,07	0,06
Esc32b	37,85	0,14
Kra30b	5,37	0,12
Lipa90b	22,20	0,63
Nug25	4,22	0,09
Sko56	3,38	0,32
Sko64	2,93	0,39
Sko72	2,96	0,45
Sko100a	2,04	0,58
Sko100b	2,03	0,59
Sko100c	2,25	0,57
Sko100d	2,11	0,58
Sko100e	2,28	0,58
Tai30b	14,01	0,12
Tai50b	5,48	0,27
Tai60a	5,02	0,37
Tai256c	0,93	0,64
Tho150	2,63	0,25

Algoritmo Memetico 0,1 mejor		
Caso	Desv	Tiempo
Chr20b	43,43	0,08
Chr22a	17,76	0,08
Els19	30,19	0,06
Esc32b	28,57	0,14
Kra30b	6,07	0,12
Lipa90b	22,19	0,63
Nug25	4,22	0,09
Sko56	3,65	0,32
Sko64	2,93	0,39
Sko72	2,59	0,45
Sko100a	2,04	0,57
Sko100b	1,98	0,58
Sko100c	2,27	0,57
Sko100d	2,01	0,58
Sko100e	2,31	0,58
Tai30b	10,68	0,12
Tai50b	7,20	0,27
Tai60a	5,17	0,37
Tai256c	0,97	0,65
Tho150	2,67	0,25

Algoritmo	Desv	Tiempo
Greedy	62,01	0,00
BL	9,88	0,01
AGG-posicion	14,13	0,95
AGG-OX	17,33	0,75
AGE-posicion	14,93	1,00
AGE-OX	22,26	0,80
AM (10,1,0)	12,03	0,18
AM (10,0,1)	10,14	0,34
AM (10,0,1mej)	9,94	0,34

Analizando los resultados de los algoritmos, destaca rápidamente el algoritmo Greedy por su “inmediata” ejecución y por su pobre resultado, para cualquiera de los casos. Era previsible su poco tiempo de ejecución por su simplicidad, aunque eso le lleva también a unas malas soluciones obtenidas.

A continuación destacamos la búsqueda local de primero el mejor, puesto que siendo el siguiente algoritmo más sencillo, tal y como podemos observar por su escaso tiempo de ejecución, ha presentado el mejor resultado para el problema en general y para la mayoría de los casos.

Observando ahora los resultados de los algoritmos genéticos, vemos rápidamente una clara diferencia entre utilizar un cruce por posición o OX. El cruce por posición presenta un notable mejor resultado y un incremento en el tiempo de ejecución. Aun teniendo una componente aleatoria el cruce por posición, presenta mejores resultados. Aunque no tan destacablemente, pero la selección generacional presenta mejor resultado que la estacionaria.

Por último, analizando los meméticos, vemos que mejoran notablemente a los algoritmos genéticos al haber introducido cada cierto tiempo la búsqueda local. Además, aunque pareciese que al introducir eventualmente la búsqueda local iba a tardar más, vemos todo lo contrario ya que también hay una importante mejora en el tiempo de ejecución.

