Práctica 2

Problema QAP:

Técnicas de Búsqueda basadas en Poblaciones para el Problema de la Asignación Cuadrática.

Óscar David López Arcos, 75571640-B
odlarcos@correo.ugr.es
Grupo A2: Martes 17:30-19:30

Índice

Descripción del problema

Este problema consiste en encontrar la asignación óptima de de n unidades a n localizaciones, conociendo la distancia entre estas últimas y el flujo existente entre las primeras. Para ello, se considerará un costo asociado a cada una de las asignaciones, que dependerá simultáneamente de la distancia y del flujo. De este modo se buscará que este costo, en función de las distancias y los flujos, sea mínimo

Es considerado como uno de los problemas de optimización más costosos (NP-completo), ya que la función de coste a minimizar es cuadrática:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} F_{ij} \cdot D_{\pi(i)\pi(j)}$$

Debemos encontrar el π (solución al problema) que genere el menor coste total posible, siendo F_{ij} el flujo que circula entre la unidad i y la j y D_{kl} la distancia existente entre la localización k y la l. Del mismo modo, el valor $\pi(i)$ representa la localización asignada a la unidad i en una determinada solución π.

Representación de soluciones

La representación interna de este problema está formada por dos matrices F y D, que representan los flujos y distancias entre cada unidad/localización (filas) y las demás (columnas). Las dimensiones de las matrices serán por tanto $n \times n$ y las diagonales principales valdrán 0. Todo esto quedará encapsulado dentro de la clase QAP junto a la implementación de los futuros algoritmos.

Cada solución π será representada por un vector, donde las posiciones identifican las unidades y el contenido las localizaciones asociadas a estas.

Función Objetivo

El coste de una solución, se calcularía de la siguiente forma:

```
S = vector de tamaño n, que representa las unidades asociadas a las localizaciones
F / D = Matriz de flujos/distancias (Explicadas en apartado anterior)

CalcularCoste(Solucion S){
        M = (Matriz n x n dimensiones inicializada a 0)
        para cada fila i de M{
                elemento de la columna S[i] = 1
        }
        Mt = Traspuesta(M)
        coste = multiplica_suma(F, M*D*Mt)
        return coste
}
```

(Función auxiliar para cálculo de Coste)

```
multiplica_suma(Matriz M1,  Matriz M2){
        if ( dimensiones M1 ≠ dimensiones M2)
                error
        resultado = 0
        para cada fila i de ambas matrices
                para cada columna j de ambas matrices
                        resultado = resultado + M1(i,j) * M2(i,j)
}
```

De todas las soluciones existentes en nuestra población, elegiremos aquella que menos coste posea.

Operadores de un Algoritmo Genético

Generación de soluciones aleatorias

Antes de comenzar el proceso de "evolución" de nuestro algoritmo, necesitamos partir de una población (conjunto de soluciones aleatorias). El encargado de crear esta población es el método "generarPoblacion", que funciona de la siguiente forma:

```
Poblacion = Almacena cada una de las soluciones de nuestra población junto al coste de estas.
tamañoPoblacion = Tamaño de la población a generar

generarPoblacion ( Poblacion, tamañoPoblacion){

        Hasta  alcanzar el tamañoPoblacion{
                v = {1,…,n}  genero solución v con los valores [1,n]
                desordeno v aleatoriamente, para conseguir una solución válida y aleatoria.
                calculo el coste de la solución v
                Añado v a la población con su coste asociado
        }
}
```

Mecanismo de selección (Torneo Binario)

Para elegir los diferentes padres que se emparejarán en la generación actual, el método "seleccionarPadres" devolverá un vector con las posiciones de las soluciones que intervendrán para generar la población de hijos.

```
NumeroPadres = Número de padres a elegir de la Población.
```
Devuelve: Padres = Vector con las posiciones de los padres elegidos

```
seleccionarPadres( Poblacion, NumeroPadres){
        para i=1 hasta i=NumeroPadres {
                p1 = Padre aleatorio de la Poblacion
                p2 = Padre aleatorio de la Poblacion

                si Coste(p1) < Coste(p2)
                        Añado p1 a Padres
                si no
                        Añado p2 a Padres
        }
        return Padres
}
```

Como la obtención de padres ha sido aleatoria, se emparejarán de dos en dos en el mismo orden que se han generado.

Operadores de cruce: Posición

El operador de cruce basado en posición genera por defecto un único hijo. Por suerte esta carencia puede suplirse fácilmente gracias a que, a excepción de los elementos que coinciden en ambos padres, el resto se introducen de forma aleatoria. De esta forma podemos generar dos hijos y que no sean completamente iguales.

padre1/padre2: Soluciones al problema representadas por un vector, que usaremos para generar el hijo.
Devuelve: Hijo: Elemento de la población, es decir, vector solución ligado a su coste.

```
crucePosicion(padre1, padre2){

        para cada posicion i de las Soluciones padres{
                Si ambos padres coinciden en el valor de la posición i {
                        El hijo guardará en esa posición el mismo valor que sus padres
                        Se marca la posición i del  hijo como ocupada
                } si no{
                        Almaceno el valor como valor distinto
                }
        }

        para i=1 hasta i=n {
                Si la posicion i no está ocupada {
                        Introduzco en la posición i un valor de los valores distintos (aleatorio)
                        Elimino el valor introducido de la lista de valores distintos
                }
        }
        return (hijo, Coste(hijo))
}
```

Operadores de cruce: Orden

En este caso el operador sí genera dos hijos directamente, por lo que servirá con una única llamada para conseguir nuestro objetivo. Para resolver este tipo de cruce de una forma eficiente, haré uso de dos variables (vectores de tamaño n) que almacenarán las correspondencias de los valores situados entre los pivotes de ambos padres.

Explico el funcionamiento a continuación:

```
Hijos cruceOrden(Padre1,  Padre2){

        Genero dos pivotes con valores aleatorios entre [1, n]

        // Rellenar elementos centrales
        para cada valor situado entre ambos pivotes {
                Copio en las mismas posiciones de Hijo1 los valores del Padre2
                Copio en las mismas posiciones de Hijo2 los valores del Padre1
                Almaceno las correspondencias de cada valor incluido en ambos hijos
        }

        // Rellenar extremos salvando incongruencias
        para cada una de las posiciones i no situadas entre ambos pivotes{

                Compruebo si puedo introducir en cada Hijo el valor de la posición i del Padre contrario
                (Es decir, si puedo rellenar Hijo1 con los valores de Padre1 e Hijo2 con los valores de Padre2)

                En caso afirmativo
                        Introduzco ese valor.
                En caso negativo (ese valor ya ha sido utilizando entre los pivotes)
                        Sustituyo el valor por su correspondencia y vuelvo a comprobar
        }

        return Hijos( (Hijo1, Coste(Hijo1)) , (Hijo2, Coste(Hijo2)) )
}
```

Operador de mutación

Para la mutación, estimaremos el número de mutaciones a priori para evitarnos generar una cantidad abusiva de números aleatorios. Una vez estimada, obtendremos de forma aleatoria el cromosoma y genes a los cuales aplicarla.

```
Mutate( Poblacion, ProbabilidadMutacion){

        TotalCromosomas = tamaño de la Población
        GenesPorCromosoma = n
        TotalMutacionesEstimadas = TotalCromosomas* GenesPorCromosoma*ProbabilidadMutacion
```

```
        para el total de TotalMutacionesEstimadas{
                Elijo un cromosoma aleatorio
                Elijo dos posiciones aleatorias para intercambiar (i, j)
                Calculo la diferencia de Coste al intercambiar ambos genes (Práctica 1)
                Intercambio ambas posiciones
                Modifico el coste de la solución
        }
}
```

En el caso del Algoritmo Genético Estacionario la población es muy pequeña para estimar el total de mutaciones a priori. Por ello, y para evitar utilizar demasiados números aleatorios, se calcularán únicamente 2 y en función del valor decidiremos cuantas mutaciones habrá. El pseudocódigo sería de la siguiente forma:

```
        P = (1/ProbabildadMutacion)
        TotalMutacionesEstimadas = 0
        para el tamaño de la poblacion{     // Tamaño 2 en nuestro caso
                Genero numero aleatorio entre 0 y P
                Si el número generado es menor que n
                        TotalMutacionesEstimadas = TotalMutacionesEstimadas +1
        }
```

Estructura de un Algoritmo Genético

Algoritmo Genético Generacional

```
SolucionAGG (TamañoPoblacion, MáximoFitness){
        evaluaciones = 0
        Genero Población Aleatoria
        GenerarPoblacion(poblacion, TamañoPoblacion)
        evaluaciones = evaluaciones + TamañoPoblacion

        mientras evaluaciones < MáximoFitness {
                Selecciono los padres
                Padres = seleccionarPadres(poblacion)
                Genero hijos
                hijos = cruzarPoblacion (Padres, ProbabilidadCruce);
                evaluaciones = evaluaciones + ProbabilidadCruce*tamaño(Padres).
                Genero mutaciones
                Mutate(hijos, ProbabilidadMutación);
                Mantengo elitismo
                Cambio mejor padre (menor coste) por el peor hijo (mayor coste)
                Reemplazo
                poblacion = hijos
        }
```

```
    solucion = Mejor Solución de la Población
    return solucion;
}
```

Como se puede apreciar en el pseudocódigo, la implementación sigue un esquema básico de Algoritmo Genético, donde quizás el detalle de más relevancia sea la generación de los hijos, con un cruce del 70% de los padres (el resto se incluirán directamente en la población de Hijos).

```
Hijos cruzarPoblacion( Poblacion, total_parejas, probabilidad_cruce, cruce){

    Calculo el total de cruces (total_parejas * probabilidad_cruce)
    para cada posición i impar menor que el total de cruces{
        hijo1 = cruce( Elemento i de la población, Elemento i+1 de la población )
        hijo2 = cruce( Elemento i de la población, Elemento i+1 de la población )
        Añado hijo1 a Hijos
        Añado hijo2 a Hijos
    }
    para cada posicion i de soluciones no cruzadas {
        Añado "Solución i" a Hijos
    }
    return Hijos;
}
```

El reemplazo en este caso es tan sencillo como convertir la población de hijos en la población de trabajo para la siguiente iteración.

Algoritmo Genético Estacionario

La única diferencia para este algoritmo respecto al anterior (además de que la probabilidad de cruce es total y sólo se escogen dos padres por generación) sería la forma de reemplazar. Nuestra población de hijos contiene únicamente dos soluciones y debemos comprobar si estas son mejores que las dos peores de la población actual. El método "reemplazoAGE" resuelve este problema:

```
reemplazoAGE( Poblacion,  hijos){

    Comparo el mejor hijo con el segundo peor padre, y el segundo mejor hijo con el peor padre

    Si en ambos casos es posible sustituir
        Realizo ambas sustituciones
    Si sólo se ha cumplido una
        Sustituyo el mejor hijo por la peor solución
    Si no es posible sustituir ningún caso
        Compruebo si el peor padre tiene un coste mayor que el mejor hijo
            En caso afirmativo, intercambio esos dos
}
```

Algoritmos Meméticos

Bajo el esquema del Algoritmo Genético Generacional, se nos pide aplicar cada 10 generaciones la Búsqueda Local realizada en la práctica 1 a un conjunto de soluciones.

Esto simplemente consiste en añadir un contador "generaciones" y entrar en el AM al principio de cada iteración cuando este contador sea múltiplo de 10. Las diferentes implementaciones son:

Aplicado a todos los cromosomas

```
AM1(Poblacion){
        para i en (1… TamañoPoblacion){
                new_solution = BusquedaLocal(Solución i)
                Cambio "Solución i" por new_solution
        }
}
```

Aplicado aleatoriamente a un 10% de los cromosomas

```
AM2(Poblacion){
        Pls = 0.1;
        Desordeno cromosomas de forma aleaoria
        maxIndex = tamaño(Poblacion)*Pls
        para i=1 hasta i=maxIndex{
                new_solution = BusquedaLocal(Solución desordenada i)
                Cambio "Solución i" por new_solution
        }
}
```

Aplicado al 10% de los mejores cromosomas

```
AM3(Poblacion){
        Pls = 0.1
        Ordeno cromosomas según Coste de menor a mayor
        maxIndex = tamaño(Poblacion)*Pls;
        para i=1 hasta i=maxIndex{
                new_solution = BusquedaLocal(Solución ordenada por Coste i)
                Cambio "Solución i" por new_solution
        }
}
```

Procedimiento

La práctica ha sido desarrollada en C++, compilable por medio de un makefile (ver archivo "Leeme.txt"). Ejecutando las órdenes "make" desde el directorio Software y posteriormente "./bin/main" se ejecutarán todos los algoritmos explicados anteriormente.

Experimento y análisis

Los resultados obtenidos para los diferentes conjuntos de datos y algoritmos han sido:

	AGG-POS			AGG-PMX		
Archivo	**Coste**	**Desviación**	**Tiempo**	**Coste**	**Desviación**	**Tiempo**
Chr22a	9724	57,96	2,13	8790	42,79	2,00
Chr22b	8768	41,56	2,05	8284	33,74	1,98
Chr25a	12782	236,72	2,68	11266	196,79	2,57
Esc128	242	278,13	272,84	138	115,63	265,68
Had20	7158	3,41	1,67	7064	2,05	1,65
Lipa60b	3210148	27,38	25,22	3173122	25,91	25,24
Lipa80b	10009386	28,92	56,19	9952278	28,19	56,58
Nug28	6246	20,91	3,39	6068	17,46	3,30
Sko81	104786	15,15	57,50	102262	12,38	57,20
Sko90	133494	15,55	76,96	129954	12,48	76,87
Sko100a	174090	14,53	103,68	168950	11,15	104,49
Sko100f	171082	14,79	103,77	166958	12,03	104,05
Tai100a	23685966	12,51	103,93	23464022	11,45	103,93
Tai100b	1620833176	36,66	103,73	1524904217	28,58	104,09
Tai150b	626346090	25,55	334,63	614043105	23,08	335,93
Tai256c	50106838	11,95	2634,91	48846274	9,13	2632,02
Tho40	305760	27,13	8,98	304946	26,79	8,87
Tho150	9534380	17,23	339,32	9378868	15,31	339,35
Wil50	53252	9,09	16,02	52400	7,34	15,87
Wil100	295152	8,10	105,91	292292	7,05	105,51

	AGE-POS			AGE-PMX		
Archivo	**Coste**	**Desviación**	**Tiempo**	**Coste**	**Desviación**	**Tiempo**
Chr22a	9000	46,20	2,04	8282	34,54	1,93
Chr22b	9496	53,31	1,98	9360	51,11	1,93
Chr25a	13202	247,79	2,62	11212	195,36	2,49
Esc128	234	265,63	261,01	160	150,00	256,50
Had20	7350	6,18	1,61	7320	5,75	1,60

	AGE-POS			AGE-PMX		
Lipa60b	3215268	27,58	23,91	3199593	26,96	24,01
Lipa80b	10020557	29,06	52,33	9926556	27,85	52,60
Nug28	6298	21,91	3,18	5910	14,40	3,16
Sko81	105354	15,78	54,33	103646	13,90	54,24
Sko90	133556	15,60	73,10	128852	11,53	73,20
Sko100a	173212	13,95	98,92	170832	12,39	99,05
Sko100f	171082	14,79	98,88	168174	12,84	98,75
Tai100a	23699220	12,57	98,77	23393140	11,12	99,09
Tai100b	1592317098	34,26	99,01	1498467109	26,35	98,87
Tai150b	632049552	26,69	319,14	613592829	22,99	321,37
Tai256c	50997254	13,94	2654,91	49547010	10,70	2673,52
Tho40	311744	29,61	8,54	293070	21,85	8,55
Tho150	9596044	17,98	324,51	9361614	15,10	323,31
Wil50	53440	9,47	15,13	52634	7,82	15,10
Wil100	294808	7,97	100,74	293280	7,41	100,90

Como el AGG con mejores resultados ha sido el PMX, aplico el memético sobre ese caso:

	AM-1			AM-2			AM-3		
Archivo	**Coste**	**Desv.**	**Tiem.**	**Coste**	**Desv.**	**Tiem.**	**Coste**	**Desv.**	**Tiem.**
Chr22a	6560	6,56	4,04	7348	19,36	2,39	7024	14,10	2,41
Chr22b	7040	13,66	4,08	7040	13,66	2,39	7040	13,66	2,39
Chr25a	4810	26,71	5,54	6494	71,07	3,12	6468	70,39	3,12
Esc128	74	15,63	691,47	74	15,63	336,98	74	15,63	344,62
Had20	6926	0,06	3,20	6926	0,06	1,97	6926	0,06	1,98
Lipa60b	3018676	19,78	66,48	3027004	20,11	31,95	3028112	20,16	31,94
Lipa80b	9417415	21,30	154,85	9455035	21,78	71,30	9417415	21,30	71,33
Nug28	5406	4,65	7,54	5414	4,80	4,13	5452	5,54	4,12
Sko81	92794	1,97	159,46	93078	2,29	73,84	92648	1,81	73,79
Sko90	117958	2,10	213,87	117958	2,10	99,92	117958	2,10	99,79
Sko100a	155924	2,58	293,82	155924	2,58	135,45	155924	2,58	134,34
Sko100f	151632	1,74	292,71	151632	1,74	135,03	151632	1,74	134,38

	AM-1			AM-2			AM-3		
Tai100a	21862294	3,85	291,96	21862294	3,85	134,62	21862294	3,85	134,35
Tai100b	1201987393	1,35	293,20	1238551240	4,43	135,16	1232027212	3,88	134,48
Tai150b	513423426	2,91	980,69	513423426	2,91	439,27	513423426	2,91	449,34
Tai256c	44904922	0,33	2325,71	44904922	0,33	3352,95	44904922	0,33	3350,35
Tho40	254940	6,00	20,81	254940	6,00	11,09	254940	6,00	11,09
Tho150	8339388	2,53	985,76	8339388	2,53	444,14	8339388	2,53	444,40
Wil50	49674	1,76	39,28	50076	2,58	20,03	49674	1,76	20,05
Wil100	276646	1,32	294,17	276646	1,32	137,39	276646	1,32	137,20

Haciendo la media de la desviación y tiempo, obtenemos la siguiente tabla:

Algoritmo	Desviación media	Tiempo medio
AGG-POS	45,16	217,77
AGG-PMX	31,97	217,36
AGE-POS	45,51	214,73
AGE-PMX	34,00	215,51
AM-1	6,84	356,43
AM-2	9,96	278,66
AM-3	9,58	279,27

Los algoritmos genéticos consumen bastante tiempo y, a pesar de acercarse al óptimo, nunca llegan a converger. Esto provoca que las soluciones sigan cruzándose y, finalmente, pierdan toda diversidad hasta estancarse o incluso comenzar a empeorar.

Se puede apreciar que tanto los AGGs como los AGEs obtienen soluciones similares aunque cada uno de ellos opte por un sistema evolutivo distinto. En estos algoritmos es posible que si las soluciones consiguen converger lo hagan en torno a un mínimo local, por lo que no tenemos ninguna garantía de acercarnos al óptimo. El AGE intenta evitar este problema permitiendo que sólo entren en la población aquellas soluciones con un mejor fitness, sin embargo, puede tardar demasiado tiempo hasta acercarse a una solución de calidad.

Los meméticos, por otro lado, consiguen evitar estos problemas aplicando una búsqueda local cada cierto tiempo que aproxime las soluciones a diferentes óptimos. De esta forma la convergencia sí suele realizarse en torno a un mínimo global, generando muchos mejores resultados.

Es aquí donde debemos evaluar nuestras necesidades de coste frente a tiempo. Los algoritmos meméticos funcionan muy bien pero, del mismo modo que el resultado mejora

conforme la búsqueda local se aplique a más elementos, el tiempo de cómputo también aumenta.

Por eso se debe encontrar un equilibrio entre ambas partes. En nuestro problema, es cierto que aplicando BL a todas las soluciones cada 10 generaciones hemos encontrado la mejor solución (desv. media 6,84). Hay que evaluar por tanto qué merece más la pena, ya que aplicándose únicamente al 10% de las mejores soluciones la desviación media no es mucho peor (9,58) y se obtiene un ahorro en tiempo medio de casi 1 minuto.