25/04/2015

Práctica 3 Metaheurísticas basadas en poblaciones

Universidad de Córdoba

Miguel Morales Carmona
Grado de Ingeniería Informática 3er curso
Escuela Politécnica Superior
I12mocam@uco.es
45886851-B

Índice

Descripción problema CPH

Este problema consiste en una cantidad "n" de centros que pueden ser clientes o hubs. Tendremos un máximo de hubs y todos los centros son clientes a su vez. Cada cliente estará asociado a un solo hub.

Dada esta situación, el objetivo es minimizar la suma de las distancias entre clientes y sus respectivos hubs. Cada hub tiene unos recursos limitados, y es una restricción a tener en cuenta.

Nuestro vector solución será una permutación de conexiones entre clientes y hubs, mientras que para evaluarlo observaremos la matriz distancia para realizar el sumatorio. Al acabar cada iteración se evaluará si es una nueva mejor solución.

función generarSolución

Desde 0 a P (donde P son los concentradores o hubs)
VectorHubs[i]=PosiblesClientes[Aleatorio()%N-1)

Eliminar cliente de PosiblesClientes

Desde 0 a N

Asignar sin sobrepasar la capacidad de los hubs

Devuelve solucion

función evaluarSolución

Desde 0 hasta N

actual=Solución[i]

mínimo=mínimo+distancias[i][actual]

Devuelve mínimo

Algoritmos genéticos

La filosofía básica de estos algoritmos estocásticos es que las soluciones previas o padres, puedan reproducirse o variar en individuos distintos entre sí. Podemos someterlos a selecciones según el problema al que nos enfrentemos entre el total de soluciones generadas y los padres, o sólo estas primeras.

El conjunto de estas soluciones se llaman poblaciones que inicialmente son elegidas de manera aleatoria. En nuestro problema contienen información sobre su rol (cliente o servidor), capacidad y la aptitud de esta solución. Para realizar este proceso, podemos reutilizar el código de generación y evaluación, aunque se tuvieron que incluir algunas mejoras en el control de evitar sobrepasar la capacidad de los hubs. Aunque inicialmente funcionaba en la práctica 1, por alguna razón en esta causaba problemas.

Para aplicar los algoritmos genéticos necesitamos primero dos tipos de operadores ; Selección y reproducción. Los de selección son los siguientes:

- Torneo : Realizamos un torneo entre 2 individuos del total y los evaluamos para ver quién se acerca más a la función objetivo. Se repetirá tantas veces como individuos tenga la población de la siguiente generación. El pseudocódigo es el siguiente :

```
Desde 0 hasta N
        indice1 = aleatorio()
        indice2 = aleatorio()
        Comparo aptitudes y elijo el ganador.
return ganador
```

- Ruleta : Se crea una ruleta con áreas según sus aptitudes acumuladas, entonces el que tenga mayor aptitud tendrá mayor probabilidad.

```
Desde 0 hasta N
        poblacion[i] = 1/evaluarSolucion(poblacion[i])+F;
        F = F+evaluarSolucion(poblacion[i]) //Acumulada
FinPara
Desde 0 hasta N
        poblacion[i]=población[i]/F
finPara
Desde 0 hasta seleccionados
        select = aleatorio()
        Desde 0 hasta N
```

si poblacion[j] >= select

poblacionauxiliar[i]=poblacion[j] //Incluida la aptitud

finsi

finpara

finpara

devolver poblacionauxiliar;

Los dos operadores de reproducción son :

- Mutación : Se basa simplemente en variar una solución ligeramente, controlando que no se incumpla las restricciones del problema. Aunque en mi código lo controla en la misma evaluación.

indice1 = aleatorio()%n

indice2 = aleatorio()%n //Deben ser distintos los indices.

intercambio(indice1, indice2) //excepto aptitudes

aptitudPoblacion = -1 //Para identificar el mutado

- Cruce : Se basa en generar un descendiente mezcla de dos padres. Al igual que en el anterior, mi evaluador controlará las restricciones del problema.

Desde 0 hasta N

Si ind1->VectorSolucion[i]<data.p && ind2->VectorSolucion[i]<data.p entonces

select=aleatorio()

si select < 0.5 entonces

intercambio(ind1->VectorSolucion[i],ind2->VectorSolucion[i])
//Excepto aptitudes

ind1Aptitud = ind2Aptitud = -1

finsi

finsi

finpara

Pseucódigo algoritmo genético

Con el uso de las funciones anteriores, podemos realizar esta función para llevar a cabo tanto AGg (no estacionario) como AGe (estacionario). La diferencia el primero respecto del segundo es que los padres son desechados.

```
Desde 0 hasta Npoblacion
        pob[i]←generarSolucionRandom()
        pob[i]->aptitud←evaluarSolucion(pob[i])
        cont++
finPara
Desde 0 hasta Npoblacion+2
        aux[i]←reservaMemoriaSolucion()
finPara
Si es AGe
        Hacer durante 5000*N iteraciones
                auxmejor=mejorSolucionPoblacion(pob, Npoblacion)
                Desde 0 hasta N
                        mejor->VectorSolucion[i]=auxmejor->VectorSolucion[i]
                finPara
                mejor->aptitud=auxmejor->aptitud //Guardo tambien actitud
                pob2<-seleccionPorRuleta(pob, Npoblacion, 2)
                select = aleatorio();
                Si aleatorio <= pCruce entonces
                        operadorCruce(pob2[0],pob2[1])
                finSi
                Desde 0 hasta hasta 2
                        select = aleatorio();
                        Si aleatorio <= pMutacion entonces
                                operadorMutacion(pob2[i]);
                        finSi
```

```
finPara
Desde 0 hasta hasta 2
        Si pob2[i]->aptitud == -1 //Ha habido cambio
                pob2[i]->aptitud←evaluarSolucion(pob2[i])
        finSi
finPara
Desde 0 hasta Npoblacion
        Desde 0 hasta N
                aux[i]->VectorSolucion[j]=pob[i]->VectorSolucion[j]
        FinPara
        aux[i]->aptitud=pob[i]->aptitud
FinPara
Desde 0 hasta N
        aux[Npoblacion]->VectorSolucion[i]=pob2[0]->VectorSolucion[i]
FinPara
aux[Npoblacion]->aptitud=pob2[0]->aptitud
Desde 0 hasta N
        aux[Npoblacion+1]->VectorSolucion[i]=pob2[1]->VectorSolucion[i]
FinPara
aux[Npoblacion+1]->aptitud=pob2[1]->aptitud
pob2=seleccionPorTorneo(aux,2,(Npoblacion+2),Npoblacion-1)
Desde 0 hasta Npoblacion-1
        SI mejor->aptitud >  pob2[i]->aptitud
                mejor=TRUE
        FinSi
FinPara


Si mejora = True
```

```
                nGenSinMejorar=0
            Finsi
            Sino
                nGenSinMejorar++
            Finsi
            SI nGenSinMejorar >= nEval
                Desde 0 hasta Npoblacion-1
                    pob2[i]=generarSolucionRandom()
                    pob2[i]->aptitud=evaluarSolucion(pob2[i])
                finPara
                nGenSinMejorar=0
            FinSi
            Desde 0 hasta Npoblacion-1
                Desde 0 hasta N
                    pob[i]->VectorSolucion[j]=pob2[i]->VectorSolucion[j]
                FinPara
            FinPara
            Desde 0 hasta N
                pob[Npoblacion-1]->VectorSolucion[i]=mejor->VectorSolucion[i]
            FinPara
            pob[Npoblacion-1]->aptitud=mejor->aptitud
        finWhile
        Mejor ←mejorSolucionPoblacion(pob,Npoblacion)
Finsi
```

```
Sino
        Hacer durante 5000*N iteraciones
            auxmejor=mejorSolucionPoblacion(pob, Npoblacion)
            Desde 0 hasta N
                mejor->VectorSolucion[i]=auxmejor->VectorSolucion[i]
            FinPara
            mejor->aptitud=auxmejor->aptitud
            pob2=seleccionPorRuleta(pob, Npoblacion, Npoblacion-1)
            Desde 0 hasta Npoblacion-1
                select = aleatorio()
                Si aleatorio <= pCruce
                    operadorMutacion(pob2[i])
                finSi
            FinPara
            Desde 0 hasta Npoblacion-1
                Si pob2[i]->aptitud == -1 //Ha habido cambio
                    pob2[i]->aptitud=evaluarSolucion(pob2[i])
                Finsi
            Finpara
            Desde 0 hasta Npoblacion-1
                Si pob2[i]->aptitud < mejor->aptitud
                    Mejora=TRUE
                Finsi
            Finpara
            Si mejora==True
                nGenSinMejorar=0
            Finsi
            Sino
```

```
                nGenSinMejorar++
            Finsi
            SI nGenSinMejorar >= nEval
                Desde 0 hasta Npoblacion-1
                    pob2[i]=generarSolucionRandom()
                    pob2[i]->aptitud=evaluarSolucion(pob2[i])
                FinPara
            FinSi
            Desde 0 hasta Npoblacion-1
                Desde 0 hasta N
                    pob[i]->VectorSolucion[j]=pob2[i]->VectorSolucion[j]
                FfinPara
                pob[i]->aptitud=pob2[i]->aptitud
            FinPara
            desde 0 hasta N
                pob[Npoblacion-1]->VectorSolucion[j]=mejor->VectorSolucion[j]
            FinPara
            pob[Npoblacion-1]->aptitud=mejor->aptitud
        finWhile
        mejor=mejorSolucionPoblacion(pob,Npoblacion)
FinSi
Devuelve mejor
```

Análisis de resultados

Los resultados tras la ejecución del script han sido :

	Algoritmo AG g		Algoritmo AG e	
Fichero	Desv	Tiempo	Desv	Tiempo
phub_100_10_10.txt	12.56	73.998	15.01	79.845
phub_100_10_1.txt	14.77	73.970	15.21	79.895
phub_100_10_2.txt	21.03	73.905	24.62	79.679
phub_100_10_3.txt	16.92	73.769	17.76	79.446
phub_100_10_4.txt	14.31	73.829	16.68	79.742
phub_100_10_5.txt	13.37	74.201	16.13	79.819
phub_100_10_6.txt	15.19	73.918	15.09	79.883
phub_100_10_7.txt	15.07	73.963	12.74	79.714
phub_100_10_8.txt	18.02	73.976	15.16	79.752
phub_100_10_9.txt	15.65	73.913	15.76	79.442
phub_50_5_10.txt	5.73	14.755	8.83	16.294
phub_50_5_1.txt	10.65	14.637	15.67	16.264
phub_50_5_2.txt	12.51	14.682	9.35	16.253
phub_50_5_3.txt	12.60	14.702	16.14	16.246
phub_50_5_4.txt	7.06	14.731	11.34	16.245

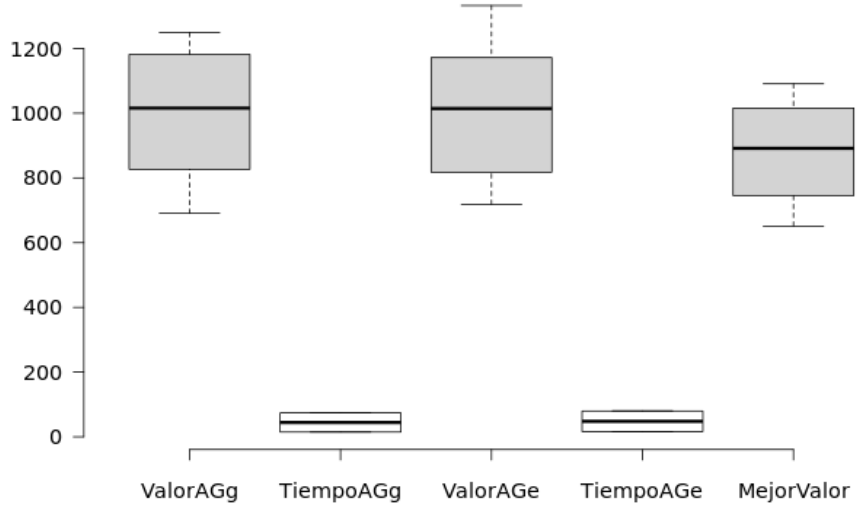
phub_50_5_5.txt	22.28	14.642	15.03	16.348
phub_50_5_6.txt	9.35	14.656	8.78	16.301
phub_50_5_7.txt	7.85	14.665	7.15	16.324
phub_50_5_8.txt	7.43	14.677	5.42	16.313
phub_50_5_9.txt	10.76	14.738	7.34	16.278
	CPH			
Media	13.16	44.316	13.46	48.004
Desviación Típica	4.48	30.398	4.58	32.542
Máximo	22.28	74.201	24.62	79.895
Mínimo	5.73	14.637	5.42	16.245

Los tiempos de ejecución tienen un factor multiplicativo añadido debido a que genero hijos sin comprobar las restricciones, y las compruebo posteriormente al evaluar. Esto implica un coste computacional mayor que comprobándolo al generar así evitando hijos no aptos. El error fue corregido en una versión posterior a este documento y que no he podido adjuntar con sus soluciones en esta versión de la práctica.

Este multiplicativo nos acompleja la observación de los tiempos de ejecución ya que empequeñece las diferencias entre los tiempos pero podemos observar que es ligeramente inferior los tiempos de AGg. Podemos presuponer que esto es porque AGe realiza una selección de la población que implica más coste computacional que AGg, y que al no verse afectada por la reproducción de hijos, no crece esa diferencia de tiempos y se mantiene constante.

Las medias podemos observar que son similares aunque hay que tener en cuenta dos detalles. El máximo de AGg es menor que en AGe, por lo que en líneas generales obtendrá peores resultados. Para ver en qué momento sucede esto podemos mirar la tabla y ver que en instancias pequeñas es mejor AGg y en grandes AGe, en líneas generales, excepto algunos casos que pueden ser causados por la aleatoriedad del algoritmo. El algoritmo tiene aleatoriedad en casi todas las partes del mismo; reproducción, selección, generación de soluciones...etc.

Los boxplots de estas instancias son los de la figura siguiente, dónde podemos observar que los resultados son parecidos en AGe y AGg aunque su máximo y su mínimo no, como comentábamos anteriormente. También podemos observar sus diferencias respecto al boxplot del mejor valor.



Box plot statistics

	ValorAGg	TiempoAGg	ValorAGe	TiempoAGe	MejorValor
Upper whisker	1249.76	74.27	1332.88	80.24	1091.00
3rd quartile	1181.80	73.92	1172.11	79.73	1016.00
Median	1015.90	44.29	1014.44	47.81	891.50
1st quartile	826.92	14.69	817.92	16.29	745.50
Lower whisker	690.43	14.59	718.20	16.15	651.00
Nr. of data points	60.00	60.00	60.00	60.00	60.00