Universidad de Granada

Práctica 2 - Metaheurísticas
Problema de la Máxima Diversidad

Algoritmo Genéticos y Meméticos

Juan Pablo García Sánchez
3º GII – Universidad de Granada

DNI: 77635249-Z
e-mail: juanpabloyeste@correo.ugr.es
Grupo 1 (Miércoles 17:30-19:30)

Índice

Descripción del problema

El problema de la máxima diversidad consiste en seleccionar m elementos de un conjunto de n elementos de tal manera que se maximice la diversidad entre los elementos escogidos. La diversidad en este caso la medimos como la distancia que hay entre los elementos. Por ello la función objetivo a maximizar es la distancia que hay entre los elementos escogidos en el vector solución.

Existen varios enfoques distintos para resolver este problema:

- MaxSum: la diversidad es la suma de las distancias entre cada par de los elementos elegidos.
- MaxMin: la diversidad es la distancia mínima entre los pares de elementos elegidos.
- Max Mean Model: la diversidad es el promedio de la distancia entre cada par de los elementos elegidos.
- Generalized Max Mean: igual que el modelo max mean, pero esta vez hay unos pesos asociados a cada elemento. Hay un orden de importancia de los elementos.

En este caso utilizamos el método MaxSum para resolver el problema.

Algoritmos

En esta práctica se estudian algoritmos genéticos y algoritmos meméticos. Yo solo he programado los genéticos. Dentro de estos pueden ser según su esquema generacional con elitismo (AGG) o estacionario (AGE), y según su operador de cruce puede utilizar cruce uniforme o cruce por posición. En total, tenemos 4 algoritmos genéticos distintos.

Básicamente, el funcionamiento del algoritmo es el siguiente: coge una población de vectores soluciones válidos (cromosomas) iniciales, donde cada cromosoma está formado por el mismo número de genes. Selecciona de esta población los que considera que son mejores, que serán los padres de la siguiente generación. Estos padres se cruzan (según una probabilidad) dando lugar a los hijos. Luego, los hijos tienen posibilidad de mutar sus genes y finalmente reemplazan a la población anterior.

Ahora en más profundidad: el programa lee un archivo de datos donde se encuentra el valor de n, m y las distancias entre todas las parejas de elementos.

En este caso, la población tiene 50 cromosomas de n genes. Lo representamos con un vector binario donde cada gen puede tener un “0” que significa que el elemento n no está escogido y un “1” que significa que sí lo está. El número de genes con 1 nunca puede ser mayor a m.

¿Qué cromosoma se considerará mejor que el resto? Aquel que maximice la función objetivo, es decir, cuya evaluación sea mayor. La función evaluación es simplemente la suma de todas las parejas de elementos seleccionados:

```
valor = 0
for i = 0:n
begin
    if cromosoma[i] == 1
    for j = i:n
    begin
        if cromosoma[j] == 1
```

```
        valor += distancia[i][j]
    end
end
```

Lo que hace es buscar los elementos del cromosoma que están seleccionados y calcular las distancias de ese elemento al resto. En el segundo bucle empieza desde el elemento que está calculando para evitar calcular las mismas parejas dos veces.

Para crear la primera población, toma un vector binario de tamaño n donde los primeros m elementos son 1 y el resto son 0. Después, basta con barajar el vector 50 veces para crear 50 cromosomas aleatorios válidos. Estos conformarán la población inicial.

```
cromosoma = Ø
for i = 0:n
begin
    if (i < m)
        cromosoma = 1
    else
        cromosoma = 0
end

for i = 0:50
begin
    shuffle(cromosoma)
    poblacion[i] = cromosoma
end
```

Ahora calcula la evaluación de cada cromosoma y pasamos a evolucionar esta población. En cada generación se hacen 50 evaluaciones a la función objetivo en el algoritmo **generacional** y 2 en el **estacionario**. Se para cuando se hayan hecho 100000 evaluaciones.

El primer paso, si se trata de un algoritmo **generacional** es hallar el mejor cromosoma de la población actual. Esto nos servirá al final cuando hagamos la sustitución de una población por otra ya que en el algoritmo generacional hay **elitismo**. Esto quiere decir que en caso de que el mejor cromosoma de esta generación no sobreviva, reemplaza al peor cromosoma de la generación siguiente.

Copia la población actual en otra variable `poblaciónAnterior`, así podemos modificarla sin preocupación de perder la generación anterior.

Lo siguiente es seleccionar los padres para lo que será la siguiente generación. Tanto si es **generacional** o **estacionario**, los padres se seleccionan con un torneo binario, es decir, se eligen dos cromosomas al azar de entre toda la población y selecciona el mejor de ellos dos. La diferencia es que en el algoritmo **generacional** selecciona tantos padres como cromosomas hay en la población (50) y el algoritmo **estacionario** selecciona tan solo dos padres.

```
for j = 0:padres
begin
    indice1 = random(0:50)
    indice2 = random(0:50)
    if eval[indice1] > eval[indice2]
        poblacion[j] = poblacionAnterior[indice1]
    else
        población[j] = poblacionAnterior[indice2]
end
```

Después, estos padres se cruzan. En el caso del **generacional**, tenemos una probabilidad de que se crucen o no, que es de 0,7. Para no generar más números aleatorios, se calcula el número de cruces esperados que es *probabilidad * nº Cromosomas / 2*. Para el **estacionario**, hay una probabilidad de 1, es decir, siempre se cruzan los 2 padres que hemos seleccionado. Ambos padres seleccionados para el cruce se cruzan 2 veces para generar 2 hijos distintos. Contamos con dos operadores de cruce distintos: **uniforme** o por **posición**.

- **Uniforme**: busca los genes que sean iguales en ambos padres y el hijo los heredera. El resto de genes se generan aleatoriamente. Esto puede dar lugar a cromosomas con menos genes con 1 que m o más y necesitamos que tenga exactamente m genes seleccionados. Para eso contamos con la función **reparar** que lo solventa. Si el cromosoma tiene pocos genes seleccionados, selecciona aquellos que contribuyen más a la solución; si el cromosoma tiene muchos genes seleccionados, elimina los que más contribuyan a la solución (por aumentar la variedad de resultados).

```
for i = 0:n
begin
    if cromosoma1[i] == cromosoma2[i]
        hijo[i] = cromosoma1[i]
    else
        hijo[i] = random(0:1)
end

function reparar
begin
    Contador = 0
    for i = 0:n
    begin
        if cromosoma[i] == 1
            contador++
    end
    if contador > m
    begin
        diferencia = contador – m
        while diferencia > 0
        begin
            cromosoma[mejorIndice] = 0
            diferencia = diferencia - 1
        end
    end
    if contador < m
    begin
        diferencia = m – contador
        while diferencia > 0
        begin
            cromosoma[mejorIndice] = 1
            diferencia = diferencia – 1
        end
    end
end
```

- **Posición**: al igual que en el **uniforme**, se buscan los genes que son iguales en los 2 padres y se transmiten al hijo. El resto de genes es una combinación aleatoria de los genes restante de uno de los padres. Así, siempre se confirma que si los padres tenían m genes a 1, el hijo también los tendrá.

```
for i = 0:n
begin
    if cromosoma1[i] == cromosoma2[i]
        hijo1[i] = cromosoma1[i]
        hijo2[i] = cromosoma1[i]
        cromosoma1 – {cromosoma1[i]}
        cromosoma2 – {cromosoma2[i]}
end
shuffle(cromosoma1)
hijo1.insertar(cromosoma1)
shuffle(cromosoma1)
hijo2.insertar(cromosoma1)
```

  Para insertar, se podría haber usado un vector con posiciones vacías e insertar los elementos del cromosoma en esos genes, utilizar algún valor que indique que se tiene que reemplazar (por ejemplo el –1), pero en mi caso he guardado un vector con las posiciones que había que cambiar he ido insertando los elementos del padre en esas posiciones.

Una vez generados los hijos, queda ver si mutan o no. Al igual que con los cruces, hay una probabilidad de 0,001 de que cada gen mute. En vez de seleccionar genes al azar, se calcula de antemano como *probabilidad * n * numHijos*, por lo que en la **generacional** numHijos será 50 y en la **estacionaria** será 2. Para mutar, se selecciona un gen al azar y se cambia por otro gen aleatorio del mismo cromosoma que tenga el valor contrario.

```
M = random(0:numHijos)
n1 = random(0:n)
n2 = random(0:n)
while hijos[M][n1] == hijos[M][n2]
begin
    n2 = random(0:n)
end
hijos[M][n1] = !hijos[M][n1]
hijos[M][n2] = !hijos[M][n2]
```

Para acabar esta generación, tiene que reemplazar a la anterior. Para el algoritmo **generacional**, los hijos reemplazan directamente a la población anterior, con la condición de que si el mejor caso no sobrevive de la población anterior no sobrevive, ocupa el lugar del peor caso de la población actual. Para el algoritmo **estacionario**, los 2 hijos que hemos generado reemplazan a los 2 peores cromosomas de la generación anterior. Hecho esto, se vuelven a evaluar los cromosomas y comienza la siguiente generación.

Método de búsqueda y operadores relevantes

Para la búsqueda, se han guardado las evaluaciones de cada cromosoma en todas las generaciones en un vector llamado evaluaciones. Para buscar los mejores cromosomas, simplemente había que buscar el elemento mayor del vector. El índice de este elemento coincide con el índice del cromosoma en la población.

```
mejorEvaluacion = 0
mejorIndice = 0
for i = 0:cromosomas
begin
    if evaluacion[i] > mejorEvaluacion
        mejorEvaluacion = evaluación[i]
        mejorIndice = i
```

El mismo método se utiliza para encontrar los peores cromosomas, pero cambiando la condición del if e inicializando peorEvaluacion a MAX_INT (número más grande que puede representar C++). Esto se utiliza en el algoritmo **estacionario** para encontrar el peor cromosoma y reemplazarlo por uno de los hijos generados.

En el algoritmo **generacional**, como he indicado se calcula el mejor cromosoma de la población anterior. A la hora de reemplazar, se busca a la vez el peor cromosoma o un cromosoma idéntico al mejor de la población anterior. Si se encuentra el mismo cromosoma, se para el bucle y no se modifica nada ya que el mejor caso a sobrevivido al reemplazo. Si no se encuentra, el mejor cromosoma anterior reemplaza al peor cromosoma actual.

```
sobrevive = false
for i = 0:cromosomas && !sobrevive
begin
    if actual[i] == anterior[mejorIndice]
        sobrevive = true
    if evaluación[i] < peorEvaluacion
        peorEvaluacion = evaluacion[i]
        peorIndice = i
end
if !sobrevive
    actual[peorIndice] = anterior[mejorIndice]
    evaluación[peorIndice] = anterior[mejorIndice]
```

Desarrollo de la práctica

Los códigos para los operadores siguen las pautas indicadas en el guión y las mejoras que se indican en el Seminario 3 para los cruces y mutaciones, así como la función de reparación de los cromosomas en los algoritmos generacionales.

Estos códigos se han compilado en Windows con Dev-C++ (utiliza el compilador gcc) que crea un ejecutable con el que trabajar.

Para probar los algoritmos, solo hay que lanzar el ejecutable de cada uno e introducir el archivo con los datos. Todos devuelven el mejor caso obtenido y el tiempo que tarda de media que se ha gastado en cada generación.

Análisis de resultados

Como se puede ver en los datos, dan resultados bastante peores que con Greedy o BL. Se puede notar también que los algoritmos estacionarios han dado mejores resultados que los generacionales y a su vez los que han usado el operador de cruce uniforme han dado mejor resultado que el de posición.

Creo que si el operador de cruce uniforme eliminara los peores genes en caso de que se hubieran seleccionado demasiados daría resultados iguales o mejores que el algoritmo Greedy y BL.

También los resultados malos se pueden deber a que la población es pequeña o se han hecho pocas iteraciones.

En general, cada iteración tarda mucho menos en estos algoritmos que en Greedy o BL. Si se nota que el AGG tarda más ya que tiene una población intermedia más grande y tiene que hacer más operaciones. También se puede apreciar que el operador de cruce por posición tarda **5 veces** más que el operador uniforme. Esto se debe a que para generar los genes que faltan hay que barajar los vectores de los padres y al final al cabo esto se hace generando números aleatorios que es una operación muy costosa. Se nota sobretodo la carga de tiempo con los casos de n = 2000, que tardan mucho más que los de n = 500.

Vistos estos datos, creo que podrían haber dado resultados mucho mejores si se hubieran hecho más generaciones o poblaciones más grandes y debido a que tardan menos que los algoritmos Greedy y BL no debería suponer ningún problema aumentar como he dicho el número de iteraciones. Si que es verdad que conforme el n sea más grande, más tardará y fácilmente podría superar el tiempo de los algoritmos Greedy y BL, en cuyo caso no veo recomendable usar estos algoritmos (sobretodo con operador de cruce estacionario).