Universidad de Granada

Práctica 4 - Metaheurísticas
Problema de la Máxima Diversidad

Bumblebees

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.

Algoritmo

En esta práctica se estudia la metaheurística *Bumblebee*, descritas en "Bumblebees: A Multiagent Combinatorial Optimization – Algorithm Inspired by Social Insect Behaviour" por Francesc Comellas y Jesús Martínez-Navarro de la Universitat Politècnica de Catalunya (UPC).

La metaheurística presentada, definida a grandes rasgos, simula una colmena de abejorros donde cada abejorro tiene una solución del problema y se mueven a través de una malla toroidal (cuando se "sale" por un borde de la malla, se "entra" por el opuesto). Cada abeja tiene una esperanza de vida y al agotarse mueren. La metaheurística espera encontrar una solución a través de la mutación de estas soluciones y dando más esperanza de vida a los abejorros con mejores soluciones. Además, si un abejorro se mueve a una casilla con comida, su esperanza de vida aumenta, evitando así que se pierdan soluciones "malas" que tienen potencial a mejorar. También nace un abejorro con una solución buena cada pocas generaciones, por lo que se asegura que, aunque las buenas soluciones hayan mutado a soluciones malas, se mantiene la malla con al menos alguna solución buena.

Este enfoque se podría resumir en una mezcla de **algoritmo genético** y ***angels & mortals***. Del algoritmo genético se toma el operador de mutación, que consiste en que dos genes al azar con valores distintos se intercambian, produciendo una solución nueva. *Angels & mortals* consiste en que unos mortales tienen una esperanza de vida y esta va decrementando hasta que mueren, a no ser que se encuentren con un ángel que alargue su vida. En el caso de este algoritmo, los abejorros serían los mortales y la comida que hay en las celdas serían los ángeles.

El pseudocódigo, extraído del propio documento indicado arriba, es así:

```
Bumblebees Algorithm():
Begin
Initialize random solutions;
    Generate N random solutions of the problem;
Set MaxGenerations,
Initialize an n × m cells world;
    Create the colony nest with N bumblebees;
    Put F food cells at random;
    Associate a random solution to each bumblebee;
    Assign the lifespan of each bumblebee;
        accordingly to its solution fitness;
Repeat Until (currentGeneration > MaxGenerations) Do
    Look for the best bumblebee;
    If (its solution is the global optimum) Then
        Report solution and exit algorithm;
    endIf
    Decrease life counters;
    Reap bumblebees;
    Create a new bumblebee in the nest
        every G generations;
    Move randomly every bumblebees to a neighbor cell;
    If (a bumblebee finds food) Then
        Decrease food counter;
        Increase lifespan of the bumblebee;
        Move bumblebee back to the nest;
    endIf
    Mutate bumblebees solutions;
    Recalculate fitnesses and assign new lifespans;
    Increment currentGeneration;
endDo
End.
```

El algoritmo que he implementado crea una malla de 20x20 donde 40 de estas celdas tienen 5 unidades de fruta cada una. Genera también 200 abejorros al comienzo con soluciones aleatorias cada una y se colocan en una casilla al azar de la malla. La solución se representa con un vector binario de 1 y 0, donde 1 significa que ese elemento es parte de la solución y 0 significa que no. Guarda los 5 mejores abejorros encontrados hasta el momento y se asigna la esperanza de vida en relación al mejor coste encontrado. El abejorro con el mejor coste tendrá 100 generaciones como esperanza de vida, mientras que aquel que tenga ese coste–1000 tendrá esperanza de vida 0. Es decir, los abejorros con valores en el intervalo (maxcoste, maxcoste-1000) se mapean conforme a (100, 0). Aquellas con esperanza de vida 0, mueren.

Una vez creada esta generación inicial, se empiezan las siguientes. Se empieza bajando el lifespan de todas las abejas en 1 y borra las que han agotado su lifespan. Cada 40 generaciones, crea una abeja nueva con una de las 5 soluciones que se guardaron anteriormente y se coloca en la malla. Ahora, todos los abejorros se mueven a una casilla al azar que no esté ocupada dentro del rango (pos-2, pos+2) tanto en la “x” como en la “y”. Si encuentran comida se comen una unidad y aumenta su esperanza de vida en 6. Después, todos los abejorros mutan intercambiando un elemento del vector solución que esté a 1 por otro que esté a 0 al azar. Si el nuevo coste de la función objetivo ha mejorado, su esperanza de vida aumenta en 2, pero si empeora baja en 1. Vuelve a guardar las 5 mejores soluciones encontradas hasta ahora y pasa a la siguiente generación.

El algoritmo termina cuando se hacen 1000 generaciones o cuando ya no quedan abejorros. El abejorro guardado en el primer elemento del vector de mejores abejorros es el que devuelve el programa.

Al darle más peso a las soluciones buenas y penalizar las malas, se espera encontrar un máximo a partir de mutaciones sobre estas soluciones buenas para que mejoren aún más. Con la comida prevenimos que una solución con mal coste, pero con potencial se pierda y con el nacimiento de abejas prevenimos que solo haya soluciones con malos costes.

Método de búsqueda y operadores relevantes

Para esta implementación he hecho uso de dos clases: la clase **Bumblebee** y la clase **Cell**.

La clase Bumblebee guarda todos los datos necesarios para describir un abejorro: su vector solución, la evaluación de la función objetivo con esta solución (el coste), su esperanza de vida y la casilla en la que se encuentra. Tiene los métodos públicos necesarios para consultar y editar estos valores, entre ellos el método de **mutate** explicado abajo y **changeLifespan** que se utiliza cuando encuentra fruta, mejora o empeora su coste al mutar o se ha pasado a una generación nueva, para modificar la esperanza de vida del abejorro..

La clase Cell guarda un par entero-entero para saber que celda de la malla representa, un bool que indica si la casilla está ocupada o no y un contador con la fruta que hay en esa celda.

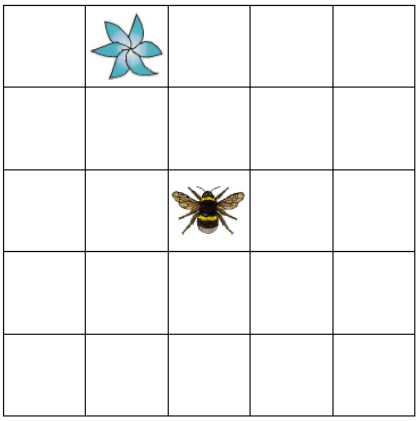
Todos los números generados aleatoriamente utilizan el mismo generador mt19937 (Mersenne Twister 19937 generator) con la *seed* puesta a 0. Para generar las soluciones iniciales, en un vector binario se guardan m elementos con 1 y el resto a 0. Luego se baraja y se asigna a un abejorro.

```
solucion = [1]*m + [0]*n-m
for i in 0..numBumblebees:
    shuffle(solucion)
    Bumblebee[i] = solucion
end
```

Esta metaheurística explora el vecindario usando el operador de mutación, que intercambia un elemento del vector solución que tiene un 1 por otro que tiene un 0, ambos elegidos al azar.

```
indice1 = rand()
indice2 = rand()
if solucion[indice1] != solucion[indice2]
    swap(solucion[indice1], solucion[indice2])
```

A parte, se usan un operador de movimiento para recorrer la malla que, aunque no cambia la solución del abejorro en sí, si da la posibilidad de aumentar su esperanza de vida al encontrar comida y así tener más generaciones donde explorar el vecindario. El abejorro se puede mover a 24 casillas diferentes alrededor de él.



Con el movimiento 1, se llega a la esquina superior izquierda de la imagen. Con el movimiento 2, a la celda a la derecha de la esquina superior izquierda y así hasta llegar al movimiento 24 que lleva a la esquina inferior derecha. Se genera un número aleatorio entre 1 y 24 para elegir el movimiento que hará el abejorro. La función **newCell** calcula, dada la posición del abejorro y el movimiento que debe hacer, la nueva celda donde estaría. Si "sale" por uno de los bordes de la malla, entrará por el opuesto. Luego se comprueba si la celda a la que debería estar ocupada. Si no lo está, se mueve a ella. Si está ocupada, se genera otro movimiento hasta encontrar una casilla libre.

```
switch(mov)
    case 1:
        pos.x += -2;
        pos.y += -2;
    case 2:
        pos.x += -2;
        pos.y += -1;
    case 3:
        pos.x += -2;
    ...
    case 24:
        pos.x += 2;
        pos.y += 2;

pos.x = pos.x mod columnas;
pos.y = pos.y mod filas;
```

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 y mejoras posibles

Caso	Valor	Tiempo
GKD-c_11_n500_m50	16531,49	382,67
GKD-c_12_n500_m50	16420,59	372,26
GKD-c_13_n500_m50	16344,32	373,85
GKD-c_14_n500_m50	16424,14	360,04
GKD-c_15_n500_m50	16170,3	363,48
GKD-c_16_n500_m50	16873,55	363,03
GKD-c_17_n500_m50	16335,43	370,27
GKD-c_18_n500_m50	16492,92	360,54
GKD-c_19_n500_m50	16239,15	371,62
GKD-c_20_n500_m50	16299,77	364,03
MDG-b_1_n500_m50	637428,9	354,39
MDG-b_2_n500_m50	644396,9	373,9
MDG-b_3_n500_m50	642743,7	337,8
MDG-b_4_n500_m50	641654,3	376,99
MDG-b_5_n500_m50	635646,68	390,15
MDG-b_6_n500_m50	638785,8	375,37
MDG-b_7_n500_m50	636531,8	372,56
MDG-b_8_n500_m50	632645,1	340,43
MDG-b_9_n500_m50	637933,1	339,1
MDG-b_10_n500_m50	640011,1	389,21
MDG-a_31_n2000_m200	100949	5737,4
MDG-a_32_n2000_m200	100515	5824,7
MDG-a_33_n2000_m200	100874	5659,9
MDG-a_34_n2000_m200	100781	5425,1
MDG-a_35_n2000_m200	100692	5567,6
MDG-a_36_n2000_m200	100667	5597,5
MDG-a_37_n2000_m200	100837	5746,9
MDG-a_38_n2000_m200	100500	5666,9
MDG-a_39_n2000_m200	100403	5693,5
MDG-a_40_n2000_m200	100791	5657,4
Media:	**238733,5**	**2130,3**

Media mejores casos:	**303460**

	Media	Desviación	Tiempo
Greedy	296303	2,358476947	317,67
BL	297327	2,021250503	218,34
ES	294657	2,900985298	90,223
BMB	296738	2,21522847	586,9
ILS	298716	1,563413475	359,01
ILS-ES	292247	3,695290083	245,09
BB	251964	16,96973027	2130,3

Viendo los resultados podemos comprobar que los resultados de BB son mucho peores que los demás estudiados a lo largo de la asignatura, tanto los resultados que dan como los tiempos de ejecución.

Que tarde tanto tiempo se debe a que se generan muchos números aleatorios (para generar todas las soluciones, para colocar los abejorros en la malla, para colocar la fruta en la malla, para mover los abejorros, para mutar las soluciones, generar otra vez si el movimiento o la mutación no es correcta…) y como generar estos números es muy costoso, tarda mucho.

Que los resultados sean tan malos, se podría solucionar con un número mayor de abejorros iniciales ya que 200 soluciones aleatorias es una muestra pequeñísima de todas las combinaciones posibles. Otra propuesta es aumentar más el tope máximo de lifespan, para darle más tiempo a que muten y exploren más vecindario.

Creo que el principal problema es que, aunque se tome el operador de mutación de los operadores genéticos, no se adopta también el de cruce que para mí es lo más importante del algoritmo genético. El operador de cruce creaba una generación nueva con los mejores genes de los padres, y esto hacía que consiguiera resultados finales tan buenos. El operador de mutación servía en este caso para añadir variedad a las soluciones encontradas, pero no era el motivo principal por el que se encontraban mejores soluciones.

Por eso pienso que este es el fallo de Bumblebees, aunque adopte el operador de mutación del algoritmo genético y la esperanza de vida de *angels & mortals*, solo con eso no puedes garantizar que se encuentren soluciones mejores a las que ya se tienen. Cuanto más cerca se está del óptimo, menos probabilidades tiene la mutación de mejorar el resultado y más de empeorarlo, por lo que no creo que se deba depender solo en este operador para explorar el vecindario porque no es fiable. Esta misma razón es lo que hace que aunque se metan nuevos abejorros con una de las mejores soluciones, la mutación posiblemente lo único que haga es que empeore.