Universidad de Granada

Práctica 3 - Metaheurísticas
Problema de la Máxima Diversidad

Búsquedas por trayectorias

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 de búsqueda por trayectorias. Los 4 algoritmos que se estudian en esta práctica parten del algoritmo de búsqueda local que se vio en la práctica inicial y el algoritmo de enfriamiento simulado. Estos son:

- Enfriamiento Simulado (ES)
- Búsqueda Multiarranque Básica (BMB)
- Búsqueda Local Reiterada (ILS)
- Algoritmo Híbrido ILS-ES

Todos los algoritmos representan las soluciones de la misma manera: con un vector que guarda los elementos seleccionados.

Enfriamiento Simulado

Enfriamiento simulado comienza con una “temperatura” inicial muy alta, y se va enfriando con cada iteración. Cuando se enfría lo suficiente, para. Visto así, no se diferencia de otros algoritmos que iteran x veces, pero se diferencia de estos en que esta temperatura también se utiliza a la hora de aceptar vecinos en el llamado Algoritmo de Montecarlo.

El algoritmo de Montecarlo calcula la probabilidad de que un vecino sea aceptado o no. El vecino se aceptará siempre que mejore el coste y, en caso de que no sea así, depende de la probabilidad calculada antes.

Profundizando más: el algoritmo genera una solución inicial aleatoria, por lo que debe cumplir las restricciones fuertes impuestas. El algoritmo empieza con una temperatura inicial calculada como:

$$T_0 = \frac{\mu \cdot C(S_0)}{-ln(\phi)}$$

Donde ϕ es la probabilidad de aceptar una solución un μ por 1 peor que la inicial, y $C(S_0)$ es el coste de la solución inicial. También definimos el número de vecinos máximo que se va a explorar en cada iteración, el número de vecinos máximo que se pueden aceptar, la temperatura final, y el número de iteraciones que se van a hacer.

En cada iteración, el algoritmo explora el vecindario. Basta intercambiar un elemento de la solución aleatorio por otro al azar de los no seleccionados. Si esta mejora el coste, se acepta y en otro caso tiene **exp(-d/T)** probabilidades de aceptarlo (algoritmo de Montecarlo) donde **d** es la diferencia del coste de la solución antes y después del intercambio y **T** es la temperatura actual. Entonces, se puede observar que cuando más baja sea la temperatura actual, menos probabilidad hay de aceptar un vecino que no mejora el coste. Si lo acepta, se mantiene el intercambio.

Una vez se exploran el número máximo de vecinos o se aceptan el máximo de vecinos, se produce el enfriamiento. El esquema de enfriamiento que se utiliza es el modelo de Cauchy modificado:

$$T_{k+1} = \frac{T_k}{1+\beta \cdot T_k}; \beta = \frac{T_0 - T_f}{M \cdot T_0 \cdot T_f}$$

Donde T_0 es la temperatura inicial, T_f la final (cuando el algoritmo está más frío se para) y M es el número de enfriamientos a realizar.

Si en esa iteración la temperatura es menor que la temperatura final impuesta; el número de vecinos aceptados es 0 o hemos superado el número de iteraciones establecido, el algoritmo acaba y devuelve la mejor solución encontrada.

Búsqueda Multiarranque Básica

Es un algoritmo simple: basta con generar 10 soluciones iniciales aleatorias y aplicar una búsqueda local a cada una de ellas, devolviendo la mejor solución de las 10.

Búsqueda Local Reiterada

Parecido al algoritmo BMB, genera una solución aleatoria inicial y aplica búsqueda local. La mejor solución encontrada hasta ahora "muta" y se pasa otra vez por el algoritmo de búsqueda local. Esto se repite un total de 10 veces, acaba devolviendo la mejor solución.

El operador de mutación consiste en intercambiar **t = m * 0.1** elementos aleatorios de la solución por otro elemento no seleccionado al azar.

Algoritmo Híbrido ILS-ES

Igual que el algoritmo ILS, genera una solución aleatoria inicial pero esta vez en vez de utilizar búsqueda local, utiliza enfriamiento simulado. La mejor solución encontrada hasta entonces muta y vuelve a usarse enfriamiento simulado. El proceso se repite 10 veces y el operador de mutación es el mismo que el utilizado en ILS.

Método de búsqueda y operadores relevantes

Para generar las soluciones iniciales, en un vector se guardan todos los elementos y se baraja. Se toma el primer elemento como parte de la solución y se vuelve a barajar. Se repite hasta tomar m elementos, y el resto se guardan en un vector de "no seleccionados".

```
seleccionados = []
noSeleccionados = []
elementos = [0..n]
for i in 0..m:
    shuffle(elementos)
    seleccionados[i] = elementos[0]
    elementos.erase(0)
end
noSeleccionados = elementos
```

El generador de vecinos consiste en intercambiar un elemento aleatorio de la solución por otro no seleccionado

```
indice1 = random(0..m)
indice2 = random(0..n-m)
swap(seleccionados[indice1], noSeleccionados[indice2])
```

El operador de mutación usado en ILS es idéntico al de generador de vecinos, solo que se aplica **t** veces en la misma solución.

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

Visto los resultados podemos ver que todos tienen una desviación parecida, pero ILS y BMB (que parten de la búsqueda local) tienen menor desviación que ES y ILS-ES (que parten del enfriamiento simulado).

A pesar de que tanto BL como ES tienen la misma generación de vecinos, ES tiene mayor desviación debido al algoritmo de Montecarlo. Al principio, al tener temperatura tan alta acepta todos los vecinos que se generen. Si el vecino que se acepta el peor, y de este se genera uno peor que también se acepta y así sucesivamente, es normal que al final de la ejecución tenga mayor desviación que el BL. Si a esto le añadimos que tienen la posibilidad de mutar (ILS-ES) pudiendo dar lugar a soluciones aún peores, se incrementa la desviación.

ILS y BMB no tienen este problema y se ve en la desviación que tienen, muy bajas donde ILS supera incluso a BL. Si nos fijamos, BMB tiene mayor desviación que BL a pesar de que al fin y al cabo es un BL ejecutado 10 veces sobre 10 soluciones distintas. Esto se debe a que BL realiza 100000 iteraciones y BMB 10000 por cada solución, por lo que es normal que todas las soluciones se queden "cortas" de alcanzar el óptimo local.

En los tiempos se pueden ver que aquellas que han utilizado más la generación de números aleatorios para los intercambios y generación de soluciones iniciales (BMB y ILS) tardan más que el resto. ES es el que menos tarda por eso, y ILS-ES tarda más ya que tiene mutación, pero aún así tarda menos que su contraparte ILS.