Universidad de Granada

Práctica 1 - Metaheurísticas
Problema de la Máxima Diversidad

Algoritmo Greedy y Búsqueda Local

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 utilizan dos algoritmos distintos: greedy y búsqueda local. Ambos tienen como input el número de elementos total n que hay en los datos, el número de elementos que deben escoger m y las distancias entre cada par de elementos. Para guardar las distancias, se utiliza una matriz completa a pesar de ser simétrica (se podría utilizar solo una diagonal) por simple comodidad a la hora de acceder a ella. Los dos devuelven un vector con los elementos seleccionados.

Lo primero que hacen los programas es leer una ruta al archivo con el fichero de datos. Consideré pasarle la ruta como argumento al lanzar el ejecutable, pero leyendo la ruta se pueden introducir varios archivos sin tener que volver a lanzar el programa.

Si la ruta es correcta, se abrirá el archivo y leerá la primera línea para obtener la n y la m. Luego, lee las distancias de ambos elementos y las guarda en una matriz simétrica, con la diagonal inicializada al 0. Esta matriz es una matriz de punteros para que los accesos sean más rápidos y porque con n = 2000 no cabe toda la matriz en el “heap”.

Para calcular el tiempo de ejecución de cada algoritmo utilizo la librería chrono porque tiene muy buena precisión y se necesita para estos casos en los que cada iteración tarda unos pocos milisegundos.

Como generador de números pseudo-aleatorios he utilizado el generador de pseudo-aleatorios Mersene Twister 19937 que viene en C++. Este generador se utiliza para barajar una lista con todos los elementos y así coger los m primeros como seleccionados y el resto como no seleccionados.

Algoritmo Greedy

El algoritmo greedy es bastante simple. Comienza buscando el elemento que más alejado está del resto, es decir, aquel elemento cuya suma de las distancias al resto de elementos sea máximo.

Este elemento es el primero que se añade al grupo de seleccionados, y se elimina del conjunto de no seleccionados.

A partir de ahí, calcula la distancia de cada elemento en el conjunto de no seleccionados con cada elemento del conjunto de seleccionados. El elemento del grupo de no seleccionados que maximice esa distancia calculada, se añade al conjunto de seleccionados y se elimina del de no seleccionados. Esta operación se repite hasta que el conjunto de seleccionados tenga m elementos y se devuelve como solución.

Siendo Sel el conjunto de elegidos y S el de no elegidos:

```
Sel = Ø
DistAc(s_i) = for s_j in S sum(d[s_i][s_j]) end
s_i* = max(DistAc(S))
Sel = Sel + {s_i*}
S = S – {s_i*}

while(|Sel| < m)
      for s_i in S
            for s_j in Sel s_i* = max(d[s_i][s_j]) end
      end

      Sel = Sel + {s_i*}
      S = S – {s_i*}
end
return Sel
```

Algoritmo Búsqueda Local

El algoritmo de búsqueda local comienza generando un vector solución aleatorio. El vector solución obviamente no puede tener elementos repetidos y deben ser seleccionados de entre el conjunto de elementos inicial. Estos elementos seleccionados se deben eliminar después del grupo de no seleccionados.

Después, calcula la distancia de cada elemento del conjunto de seleccionados al resto de los elementos de este conjunto. Esto es lo que se llama como contribución de ese elemento. Se busca el elemento que menor contribuya en el vector solución.

Luego, para cada elemento del conjunto de no seleccionados, evaluamos la función objetivo como si hubiéramos intercambiado el elemento que menos contribuye en la solución con este que hemos tomado del conjunto de no seleccionados. Así es como genera un “vecino”.

Para calcular la nueva distancia total, no hace falta volver a calcular la distancia de todos los pares de elementos. Basta con sumar la distancia del nuevo elemento al resto de elementos de la solución y restar la distancia del anterior elemento al resto de elementos de la solución. Esta es la factorización del movimiento de intercambio.

Una vez calculada esta factorización, si la diferencia de las distancias calculadas para el nuevo elemento y las del anterior es mayor que 0, es decir, es positivo, se produce el intercambio. El nuevo elemento pasa a ser parte de la solución y el anterior vuelve al conjunto de no seleccionados.

El algoritmo seguirá volviendo a calcular las contribuciones de los elementos de la nueva solución y buscando otra vez un intercambio para el elemento que menos contribuye que mejore la evaluación de la función objetivo.

Parará una vez que en una iteración el elemento que menos contribuye no tenga ningún intercambio que mejore la solución o una vez que se evalúe la función objetivo 100000 veces, lo que quiere decir que se han producido ya 100000 intercambios.

Al ser un algoritmo que depende mucho del punto de partida, se ejecuta varias veces (tantas como quiera el usuario) para estudiar mejor los resultados.

Igual que en Greedy, si S es el conjunto de no seleccionados y Sel de seleccionados:

```
Sel = random(0..n, m) // Vector con m elementos aleatorios.
S = S – Sel
while(coste(Sel') > 0 and vecinos not empty)


      for s_i in Sel
            for s_j in S
                  contribucion[] += dist[s_i][s_j]
      end


      s_i* = min(contribucion)


      for s_j in S
            Sel' = Sel – {s_i*} + {s_j}
            if (coste(Sel') > 0)
                  Sel = S
                  S = S – {s_j} + {s_i*}

      end
end
return Sel
```

Desarrollo de la práctica

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. Greedy solo necesita la ruta del archivo con los datos, pero para la Búsqueda Local ya que se ejecuta varias veces para cada caso, te pide el número de iteraciones que quieres ejecutar. Por tiempo, yo hice 10000 ejecuciones para los casos de n=500 y 100 para los de n=2000.

Greedy devuelve el resultado encontrado y el tiempo de ejecución en milisegundos y Búsqueda Local devuelve la media de los resultados encontrados, el mejor y el peor resultado encontrados y la media de tiempo de ejecución de todas las iteraciones en milisegundos.

Análisis de resultados

En el algoritmo de búsqueda local se ha utilizado como seed 0 para todos los casos. No hace falta cambiarlo para cada iteración, simplemente se vuelve a barajar el vector con los elementos utilizando el mismo generador con esa semilla.

Para el caso de búsqueda local también se ha añadido una columna con el peor caso encontrado y en la columna "coste obtenido" se ha utilizado la distancia media encontrada. En ninguna de las dos tablas se ha cambiado el mejor caso, para calcular la desviación, aunque en el caso de BL se solía encontrar el mejor resultado.

Los resultados muestran que la desviación aumenta cuantos más datos hay o cuanto más diversos son los datos. Por eso, en los casos de GKD al tener pocos elementos y tener distancias muy similares entre ellos (se encuentran cerca unos de otros), la media no se aleja mucho del óptimo y la desviación es casi nula. De hecho, hasta los peores casos se diferencian del mejor caso en unas pocas decenas como mucho.

Luego en los casos de MDG aumenta la desviación, los elementos de MDG-b tienen distancias muy distintas por lo que la media varía mucho y las peores soluciones se distinguen de las mejores soluciones en más de 10000. Aún así, la media se acerca más al mejor caso que al peor.

MDG-a tiene muchos más elementos que los otros 2 tipos de casos, por lo que el espacio de búsqueda es mucho mayor y el tiempo de ejecución aumenta considerablemente. Para este caso se han utilizado menos iteraciones por ello, pero lo ideal sería hacer más iteraciones que los otros casos porque hay muchas más combinaciones posibles en este.

Como se ve, ambos tardan más o menos lo mismo, pero Greedy tiene menos desviación en los casos de MDG. Sin embargo, en los datos se estudia con la media de las ejecuciones de la Búsqueda Local, pero la mejor solución de Búsqueda Local siempre es mejor que la solución encontrada de Greedy. Por eso, en el caso de poder ejecutar muchas iteraciones con Búsqueda Local, conviene utilizarlo porque puede encontrar una solución mejor que el algoritmo Greedy.