Practica 3 Greedy

Francisco Javier Merchán

El problema

Dada una secuencia de palabras p_1, p_2,..., p_n de longitudes l_1, l_2,..., l_n se desea agruparlas en líneas de longitud L. Las palabras están separadas por espacios cuya amplitud ideal (en milímetros) es b, pero los espacios pueden reducirse o ampliarse si es necesario (aunque sin solapamiento de palabras), de tal forma que una línea conteniendo las palabras p_i, p_{i+1},..., p_j tenga exactamente longitud L. Sin embargo, existe una penalización por reducción o ampliación en el número total de espacios que aparecen o desaparecen. El costo de fijar la línea p_i, p_{i+1},..., p_j es $(j - i)\,|b' - b|$, siendo b' el ancho real de los espacios, es decir $(L - l_i - l_{i+1} -... - l_j)/(j - i)$. No obstante, si j = n (la última palabra) el costo será cero a menos que b' < b (ya que no es necesario ampliar la última línea). Diseñar un algoritmo greedy que divida un conjunto de palabras en líneas minimizando el coste total de arreglar los espacios en cada línea.

Para resolver este problema vamos a pensar como lo haríamos en la vida real. Lo lógico sería ir metiendo palabras hasta que la suma de las longitudes con el tamaño óptimo de espacio hasta que la siguiente palabra no cupiese. En ese caso tendríamos dos opciones, o modificamos los espacios para ver si la palabra nueva pudiera caber, o aumentamos los espacios para que lleguen al final de la línea. Aquí es donde se da el comportamiento '*voraz*' del programa, el cual elige siempre la opción de estas que tenga menos coste, sin recordar lo anterior o lo más óptimo, de cara al futuro.

¿Se puede resolver por la estrategia greedy?

Como podemos ver es un problema de minimización de costo, por lo que es un firme candidato para poder resolver este problema con esta técnica. También tenemos en cuenta los siguientes aspectos para hacerlo apto, como son:

- Construir el problema con etapas.
- Seleccionar en cada iteración la mejor opción.
- No volver a considerar las iteraciones ya hechas.

Ahora veremos si los elementos necesarios para aplicar este diseño se encuentran en el problema:

- Lista de candidatos: Lista de palabras
- Lista de candidatos ya utilizados: Lista de palabras
- Función Solución: Si el conjunto de longitudes de palabras y sus espacios es igual que el tamaño máximo de la línea.
- Criterio de factibilidad: Si el conjunto de palabras, las longitudes de estas juntos con sus espacio mas la palabra a añadir, no supera el tamaño máximo de la línea
- Función de selección: Siguiente palabra que quepa en la línea, si no, crea una línea nueva.
- Función objetivo: Minimizar el coste de los arreglos de los espacios.

Esquema general de greedy

```
Función S= Voraz(vector candidatos C)
    S_parcial = 0;
    S_final =0;
    Coste_linea = 999999;
    Mientras (C != 0) hacer:
        x = Seleccion dela siguiente palabra
        Si (S_pacial U {x} <=  Longitud Max de línea) y (Calcular coste de añadir la palabra)<
        coste  entonces
            S_parcial=S U {x};
        Si no entonces
            S_final = S_parcial U S_final;
            S_parcial =0;
            Coste_linea=Calculamos el coste al añadir la palabra;
         Fin-Sino
```

Fin-Mientras
Devolver *"No hay solución"*

El diseño es muy simple, vamos añadiendo a la línea palabras y calculando el coste de redimensionar los espacios, si este coste es menor y la palabra cabe, lo añadimos a la línea, si no pues guardamos la línea, y añadimos la palabra a la línea siguiente y reiniciamos los contadores y recalculamos el coste.

Optimalidad

Esta solución no encuentra el óptimo, para ello lo podemos ver con este contra ejemplo:

Supongamos que la longitud máxima de línea, L, es de 26, y el espacio en palabras, b, es de 2. También disponemos de una entrada de 7 palabras(n), la longitudes de estas son {10, 10, 4, 8, 10, 12, 12}.

Por la plantilla anterior, tras meter las dos primeras palabras en la primera línea, tiene que tomar una decisión en cuanto a si la tercera palabra (de longitud 4) debe estar en la primera línea o no. Si el coste de comprimir los espacios que es de 2, ocasiona más que el coste de redimensionarlos los espacios para ocupar el hueco que no se puede rellenar que es de 4, lo que el programa incluye la palabra en la primera línea. Esto lo hace para las palabras consiguientes, quedando las palabras separadas de esta manera:

Línea 1-> (10, 10,4)

Línea 2-> (8, 10)

Línea 3 -> (12, 12)

La suma del coste de redimensionar los espacios en esta configuración es de 8. La primera línea tiene un costo de 2, la segunda de 6 y la tercera de 0.

Sin embargo hay otra configuración que da menor coste global, a cambio de sacrificar el primer coste, esta configuración es:

Línea 1-> (10,10)

Línea 2-> (4, 8, 10)

Línea 3 -> (12, 12)

El coste de este caso es de 4, ya que la segunda línea tiene coste 0, por que la suma de estas da 22 que con los espacios hace una suma de 26, que es justo el máximo de línea, por lo que no hay que dimensionar nada. Y la última línea que tiene siempre coste 0. Haciendo este caso el más óptimo en cuestión de coste.

Eficiencia

La eficiencia de este algoritmo es de n, que es número de palabras del texto. Básicamente hace una vuelta en el mientras por cada palabra, que las otras operaciones son elementales. Es de orden exacto porque vas a recorrer todas las palabras tanto en el mejor como en el peor de los casos.

Implementación

```
const int MAXPALABRAS = 100;
const int MAXLINEAS = MAXPALABRAS;  //En el peor caso es una palabra por linea

struct registro{
    int primera, ultima;
    double espacio, coste;

}
```

```
int Parrafo(int l, int n, int b, vector<int> longPalabras, vector<registro> solucion){
    /* L es la longitud de la linea, n el numero de palabras,
    b el tamano optimo de los espacios,
    l es el vector con las longitudes de las n palabras,
    y en sol almacena la solucion.
    Devuelve el numero de lineas que ha necesitado */
    int tamanoPalabra; // long de palabras de la linea
    int tamanoLinea; // tamaño de la linea en curso
    int nLinea; // Linea en curso
    int nPalabra; // palabra en curso
    int nEspacio; // numero de espacios linea en curso

    nLinea=1;
    ResetContadores(nlinea,1, solucion, longPalabras, tamanoPalabra, tamanoLinea, nEspacio); //Metemos la primera palabra
    nPalabra =2;
    while(nPalabra <= n){
        if(tamanoLinea+b+longPalabras[nPalabra] <= l){ //Cabe
            tamanoLinea =+ (b+longPalabras[nPalabra]);
            tamanoPalabra =+ longPalabras[nPalabra];
            nEspacio++;
        }else{ //No cabe de forma optima
            if((tamanoPalabra+longPalabras[nPalabra]+nEspacio+1) > L){
                //no cabe asi qeu la pasamos a otra linea
                CerrarLinea(nLiena, nPalabra);
                nLinea++; //Reiniciamos contadores, osea, añadimos otra linea
                ResetContadores(nLinea, nPalabra, solucion, longPalabras, tamanoPalabra, tamanoLinea, nEspacio);
            }else{ // Puede que quepa si movemos los espacios
                if(Coste(l, b, tamanoPalabra, nEspacio) >= Coste(l, b, tamanoPalabra+longPalabras[nPalabra], nEspacios+1)){
                    nPalabra++; //Cabe y es mejor que lo anterior, asi que la metemos
                }
                //No cabe la pasamos a otra linea
                CerrarLinea(nLinea, nPalabra);
                nLinea++;
                ResetContadores(nLinea, nPalabra, solucion, longPalabras, tamanoPalabra, tamanoLinea, nEspacio );
            }
        }
        nPalabra++;
    }
    if(solucion[nLinea].primera == 0){
        return nLinea-1
    }
    if(solucion[nLinea].ultima == 0){
        CerrarLinea(nLinea, nPalabra, solucion, tamanoPalabra, nEspacio, l, b);
    }
    return nLinea;
}
```

```
double Espacio(int l, int tamanoPalabra, int nesp){
    /* devuelve cero si nesp = 0, o bien un numero mayor que 1 */
    if(nesp==0){
        return 0.0;
    }
    return (l-tamanoPalabra)/(nesp);
}

void ResetContadores(int linea ,int npal, vector<registro> &solucion, vector<int> longPalabras, int tamanoPalabra, int tamanoLinea, int nEspacio ){
    if(npal <= longPalabra.size() ){ //Para que la ultima palabra no lo haga nada
        solucion[linea].primera =npal;
        solucion[linea].coste=0.0;
        tamanoPalabra = longPalabras[npal];
        tamanoLinea = longPalabras[npal];
        nEspacio=0;
    }
}

double Coste(int l, int b, int tampalabras, int nesp){
    double bprima= Espacio(l,tampalabras, nesp):
    if(bprima > b){
        return (nesp*(bprima-b))
    }else{
        return (nesp*(b -bprima))
    }
}

void CerrarLinea(int linea, int npal, vector<registro> &solucion, int tamanoPalabra, int nEspacio, int l, int b ){
    solucion[linea].ultima= npal-1;
    solucion[linea].espacio = Espacio(l, tamanoPalabra, nEspacio)
    solucion[linea].coste = Coste(l, b, tamanoPalabras, nEspacio);
}
```