UNIVERSIDAD DE GRANADA

Cuaderno de prácticas.
Práctica 3. Greedy y Programacion Dinamica.

Enrique Pinazo Moreno

28 de mayo de 2025

Índice

1. Algoritmo Greedy (Algoritmo Voraz).

1.1. Diseño del Algoritmo Greedy(Voraz) e Implentación.

- **Componentes del Diseño Greedy.**

- Cometido: Para resolver el problema de encontrar el camino mas corto para atravesar una matriz de enteros, me ha basado en un algoritmo voraz el cual selecciona el camino con menos coste en cada paso, utilizando una matriz de enteros donde quedará registrado el mapa y el coste de cada celda, el uso de esta matriz radica en el metodo algortimoGreedy(), donde se inicializamos la suma de los costes y el numero con menor coste de esta columna, cuyo valor inicial segun la columna, es el ubicado en la posicion media de esta.

```
int suma = 0;
int numeroMenor = v[f/2][0];
```

A continuacion se recorre la matriz de enteros, empezando desde la mitad de la columna j, y se aplica la heurística desarrollada para obtener el camino mas corto. En el recorrido, se compara como primer valor el numeroMenor inicializado previamente como el valor de la mitad de la columna + la suma de los costes acumulados, con el valor de la celda actual + la suma de los costes acumulados, si el numeroMenor es mayor o igual al valor de la celda actual, se actualiza el numeroMenor.
El ultimo paso es añadir el numeroMenor al vector rGreedy, cuyo cometido es almacenar el camino mas corto en orden, y actualizar la suma de los costes acumulados.

```
for (int i=0; i < c; i++){
    numeroMenor = v[f/2][i];
    for (int j=0; j < f; j++)
    {
        if(numeroMenor+suma > v[j][i]+suma)
        {
            numeroMenor = v[j][i];
        }
    }
    rGreedy.push_back(numeroMenor);
    suma += numeroMenor;
}
```

- **Heuristica, Ejemplo y Demostración de Solución Óptima.**

  - Heurística: La heurística desarrollada en este algoritmo es seleccionar el camino con menor coste en cada paso, lo que significa que en cada iteración se elige la celda con el valor más bajo de la columna actual, sumando este valor al total acumulado. Este enfoque garantiza que sea minimo el coste total del camino a medida que se avanza por la matriz.
  Por ejemplo, si tenemos una matriz de enteros como la que vimos en el enunciado::

$$
\begin{array}{|cccccc|}
2 & 8 & 9 & 5 & 8 & 1 \\
4 & 4 & 6 & 2 & 3 & 3 \\
5 & 1 & 4 & 6 & 1 & 5 \\
3 & 2 & 5 & 4 & 8 & 2 \\
4 & 2 & 3 & 3 & 2 & 3 \\
\end{array}
$$

  La heurística, selecciona desde la columna 0 pos [f/2,0] con el valor 5, hasta la columna c-1 pos [f/2,c-1] con el valor 5, se revisa si en su misma columna hay un valor menor, si lo hubiese, se actualiza la variable numeroMenor, si no, se mantiene dicho valor. Posteriormente, se sumaria el valor de la celda seleccionada al total acumulado y se añade al vector con el resultado rGreedy.
  En su esencia, sin importar el tamaño de la matriz, la heuristica de este algoritmo voraz seleccioanra el camino con menor coste sin importar el camino que se haya tomado anteriormente, siempre y cuando se mantenga el criterio de seleccionar el valor mas bajo en cada iteración , el resultado es obtener el verdadero camino con menos coste.

- Ejemplo: Supongamos que tenemos la siguiente matriz de enteros:

$$\begin{vmatrix} 2 & 8 & 9 & 5 & 8 & 1 \\ 4 & 4 & 6 & 2 & 3 & 3 \\ 5 & 1 & 4 & 6 & 1 & 5 \\ 3 & 2 & 5 & 4 & 8 & 2 \\ 4 & 2 & 3 & 3 & 2 & 3 \end{vmatrix}$$

Al aplicar el algoritmo voraz, comenzamos en la primera columna, con el valor inicial de su posicion en mitad "5". Luego, en cada iteración, es seleccioando el valor más bajo de la columna actual y lo sumamos al total acumulado. El resultado final sería un vector rGreedy que contiene los valores seleccionados en orden.
Primer numero de la columna <0>: 5
0+5 – 0+2
→ Hay un camino mas Factible: 2
0+2 – 0+4
0+2 – 0+5
0+2 – 0+3
0+2 – 0+4
Numero mas Factible: 2
Suma acumulada: 2

Primer numero de la columna <1>: 1
2+1 – 2+8
2+1 – 2+4
2+1 – 2+1
2+1 – 2+2
2+1 – 2+2
Numero mas Factible: 1
Suma acumulada: 3

Primer numero de la columna <2>: 4
3+4 – 3+9
3+4 – 3+6
3+4 – 3+4
3+4 – 3+5
3+4 – 3+3
→ Hay un camino mas Factible: 3
Numero mas Factible: 3
Suma acumulada: 6

Primer numero de la columna <3>: 6
6+6 – 6+5
→ Hay un camino mas Factible: 5
6+5 – 6+2
→ Hay un camino mas Factible: 2
6+2 – 6+6
6+2 – 6+4
6+2 – 6+3
Numero mas Factible: 2
Suma acumulada: 8

Primer numero de la columna <4>: 1
8+1 – 8+8
8+1 – 8+3
8+1 – 8+1
8+1 – 8+8
8+1 – 8+2
Numero mas Factible: 1
Suma acumulada: 9

Primer numero de la columna <5>: 5
9+5 – 9+1
→ Hay un camino mas Factible: 1
9+1 – 9+3
9+1 – 9+5
9+1 – 9+2
9+1 – 9+3
Numero mas Factible: 1
Suma acumulada: 10

Matriz de entrada:
| 2 8 9 5 8 1 |
| 4 4 6 2 3 3 |
| 5 1 4 6 1 5 |
| 3 2 5 4 8 2 |
| 4 2 3 3 2 3 |
Solución Greedy: [2],[1],[3],[2],[1],[1]
Suma total: 10

- **Semejanzas con el algoritmo Prim:**
  - Ambos algoritmos utilizan una estrategia voraz para construir soluciones incrementales.
  - En el algoritmo Greedy, se selecciona el número menor en cada columna para añadir a la suma total, mientras que en el algoritmo Prim se selecciona la arista de menor peso para añadir al árbol de expansión.
  - Ambos algoritmos construyen la solución de forma incremental, añadiendo un elemento (un número en Greedy y una arista en Prim) en cada paso hasta que la solución está completa.
- **¿Sería posible y conveniente abordarlo aplicándolo, o el de Kruskal:**
  - No sería conveniente aplicar el algoritmo de Kruskal en este caso, ya que Kruskal está diseñado para encontrar el árbol de expansión mínima en un grafo, mientras que el algoritmo greedy planteado es encontrar un camino con el menor coste en una matriz de enteros.
  - El algoritmo Greedy es más adecuado para este problema específico, ya que se centra en seleccionar el camino con menor coste en cada paso.
  - Además de que el algoritmo Kruskal requiere de una estructura de grafo al contrario de la estructura matricial del algorimto Greedy presentado.

- **Configuración de la ejecución del programa:**

```
./greedy_Voraz <f> <c> <elementos...>
```

Para que esto funcione, en el codigo se ha implementado un bucle que recorre los argumentos de la linea de comandos, desde el tercer argumento en adelante, se tranforman en enteros y se almacenan en un vector de vectores de enteros, la logica es la siguiente:

```
int f,c;
vector<vector<int>> v;
vector<int> rGreedy;

if(argc < 4){
    cout<<"Error: No se han introducido suficientes argumentos."<<endl;
    cout<<"Uso: "<<argv[0]<<" <f> <c> <elementos...>"<<endl;
    return 1;
}

f = atoi(argv[1]);
c = atoi(argv[2]);
v.resize(f, vector<int>(c));

for(int i = 0; i<f; i++){
    for(int j = 0; j<c; j++){
        v[i][j] = atoi(argv[3 + i*c + j]);
    }
}
```

- Ejemplo de ejecución y Tiempo de ejecución:

Ejemplo de ejecución del programa con una matriz de 3 filas y 3 columnas:

```
./greedy_Voraz 3 3 1 2 3 4 5 6 7 8 9
               --- -----------------
               f c elementos...
```

El tiempo de ejecucion del programa segun el tamaño mencionado en el enunciado, y los elementos de ella misma:

```
./greedy_Voraz 5 6
2 8 9 5 8 1
4 4 6 2 3 3
5 1 4 6 1 5
3 2 5 4 8 2
4 2 3 3 2 3
```

Salida:

```
Matriz de entrada:
 _ _ _ _ _ _
|2 8 9 5 8 1|
|4 4 6 2 3 3|
|5 1 4 6 1 5|
|3 2 5 4 8 2|
|4 2 3 3 2 3|
Solucion Greedy: [2],[1],[3],[2],[1],[1]
Tiempo de ejecucion: 5.3948e-05 segundos
Suma total: 10
```

2. Programación Dinámica.

2.1. Diseño del Algoritmo de Programación Dinámica e Implementación.

- **Componentes del Diseño de Programación Dinámica.**

  - Cometido: El objetivo del algoritmo de programacion dinamica desarrolado es encontrar el camino con el mínimo coste desde una celda central de la primera columna hasta cualquier celda de la última columna de la matriz. El coste de un camino se define como la suma de los valores de las celdas por las que pasa.

  - Subproblemas: El problema se descompone en subproblemas más pequeños para calcular el coste mínimo hasta alcanzar cada celda (i,j) desde la celda inicial, dando lugar a que la solución a un subproblema se construye a partir de las soluciones de subproblemas anteriores.

  - Superposición de subproblemas: Los subproblemas se solapan entre sí, como resultado, estos mismos subproblemas se resuelven repetidamente, evitando tener que recalcular estos subproblemas, gracias a almacenar sus soluciones.

  - Relación de recurrencia: Define cómo se calcula la solución de un subproblema en función de las soluciones de subproblemas más pequeños. En este caso, para una celda (i,j), el coste mínimo puede venir de la celda adyacente izquierda (i,j-1), de la celda adyacente superior (i-1,j) o de la celda adyacente inferior (i+1,j). La función calcularCosteMinimoModificado considera estas tres direcciones para calcular el coste mínimo a la celda actual.

```
coste_minimo[i][j]=m[i][j]+min(coste_minimo[i][j-1],
    coste_minimo[i-1][j],coste_minimo[i+1][j])
```

- Casos Base: El camino comienza en la celda central de la primera columna (f/2,0). El coste mínimo para esta celda es simplemente el valor de la celda misma, ya que no hay celdas anteriores de las que depender.

```
coste_minimo[f/2][0] = m[f/2][0];
```

- Solución Óptima: La solución óptima se encuentra en la última columna de la matriz de costes mínimos, donde se selecciona el valor más pequeño, que representa el coste mínimo para llegar a cualquier celda de la última columna desde la celda inicial.

```
int minimo = *std::min_element(coste_minimo.begin(),
    coste_minimo.end());
```

- **Estructura de Datos Planteada:** Se utiliza una matriz de enteros para almacenar los costes acumulados de cada celda, cuyas dimensiones son iguales a la matariz dde entrada y se inicializa con valores muy altos(infinito) para asegurar que cualquier coste minimo sea menor que estos valores iniciales.

```
const int INFINITO = std::numeric_limits<int>::max();
std::vector<std::vector<int>> coste_minimo(f, std::
    vector<int>(c, INFINITO));
```

- **Justificación del diseño y Explicación.**

  - Justificación del diseño:

  1. El diseño del algoritmo de programación dinámica es adecuado porque el camino óptimo a una celda (i,j) se construye a partir de los caminos óptimos a las celdas adyacentes previas (izquierda, arriba, abajo). Esto permite calcular el coste mínimo de manera incremental.
  2. Al almacenar los costes mínimos calculados en la tabla coste-minimo, se evita volver a calcular los mismos subproblemas múltiples veces. es decir, para calcular el coste de una celda en la columna j, se necesita el coste de una celda en la columna j-1, una vez que coste-minimo[i][j-1] se calcula, no se necesita volver a calcularlo.
  3. Es eficiente ya que la tabla coste-minimo permite construir la solución llenando la tabla de izquierda a derecha, de manera que

las pasadas verticales dentro de cada columna son necesarias para garantizar que se realicen movimientos hacia arriba y abajo dentro de la misma columna, lo cual puede llegar a afectar el coste mínimo fianl si un camino óptimo implica moverse varias veces verticalmente antes de avanzar a la siguiente columna.

- Explicación:

Suponiendo una matriz m de enteros de tamaño 3 x 3, con los valores:

```
|1 2 3|
|4 5 6|
|7 8 9|
```

La fila inicial central es f/2=3/2=1, por lo que comenzamos en la celda (1,0) con el valor 4, cuyo coste mínimo para esta celda es simplemente el valor de la celda misma, ya que no hay celdas anteriores de las que depender.
La matriz de costes minimos se inicializa con valores muy altos.

```
|INFINITO INFINITO INFINITO|
|INFINITO INFINITO INFINITO|
|INFINITO INFINITO INFINITO|
```

Luego, se calcula el coste mínimo para cada celda de la matriz, comenzando desde la celda (1,0) y propagandose hacia abajo y arriba.

```
coste_minimo[1][0] = 4
coste_minimo[2][0] = coste_minimo[1][0] + m[2][0] =
    4 + 7 = 11
coste_minimo[0][0] = coste_minimo[1][0] + m[0][0] =
    4 + 1 = 5

    |5 INFINITO INFINITO|
    |4        INFINITO INFINITO|
    |11 INFINITO INFINITO|
```

A continuación, se calcula el coste mínimo para la celda (1,1) considerando las celdas adyacentes (1,0), (2,0) y (0,0):

```
coste_minimo[1][1] = m[1][1] + coste_minimo[1][0] =
    5 + 4 = 9
coste_minimo[2][1] = m[2][1] + coste_minimo[2][0] =
    8 + 11 = 19
coste_minimo[0][1] = m[0][1] + coste_minimo[0][0] =
    2 + 5 = 7

        |5 7 INFINITO|
        |4 9 INFINITO|
        |11 19 INFINITO|
```

Se realiza la misma operacion para la celda (1,2), considerando el coste mínimo de las celdas adyacentes (1,1), (2,1) y (0,1):

```
coste_minimo[1][2] = m[1][2] + min(coste_minimo
   [1][1], coste_minimo[0][1], coste_minimo[2][1])
   = 6 + min(9, 7, 19) = 6 + 7 = 13
coste_minimo[2][2] = m[2][2] + min(coste_minimo
   [2][1], coste_minimo[1][1], coste_minimo[0][1])
   = 9 + min(19, 9, 7) = 9 + 7 = 16
coste_minimo[0][2] = m[0][2] + min(coste_minimo
   [0][1], coste_minimo[1][1], coste_minimo[2][1])
   = 3 + min(7, 9, 19) = 3 + 7 = 10

        |5 7 10|
        |4 9 13|
        |11 19 16|
```

Por último, se selecciona el valor mínimo de la última columna para obtener el coste mínimo total del camino:

```
int coste_final_minimo = coste_minimo[0][c - 1];
for (int i = 1; i < f; ++i) {
    if (coste_minimo[i][c - 1] < coste_final_minimo)
       {
        coste_final_minimo = coste_minimo[i][c - 1];
    }
}
```

El camino con el coste minimo en este caso es:
Desde (1,0) a (0,0) (coste 4+1=5)
Desde (0,0) a (0,1) (coste 5+2=7)
Desde (0,1) a (0,2) (coste 7+3=10)

- Ejemplo de ejecución y Tiempo de ejecución:

Ejemplo de ejecución del programa con una matriz de 3 filas y 3 columnas:

```
./programacion_dinamica <f> <c> <elementos...>
```

Ejemplo de ejecución del programa con una matriz de 5 filas y 6 columnas:

```
./programacion_dinamica 5 6 2 8 9 5 8 1 4 4 6 2 3
    3 5 1 4 6 1 5 3 2 5 4 8 2 4 2 3 3 2 3
```

El tiempo de ejecucion del programa segun el tamaño mencionado en el enunciado, y los elementos de ella misma:

```
./programacion_dinamica 5 6
2 8 9 5 8 1
4 4 6 2 3 3
5 1 4 6 1 5
3 2 5 4 8 2
4 2 3 3 2 3
```

Salida:

```
El camino de suma minima optimo es: 21
Tiempo de ejecucion del algoritmo: 4.99e-06 segundos.
```