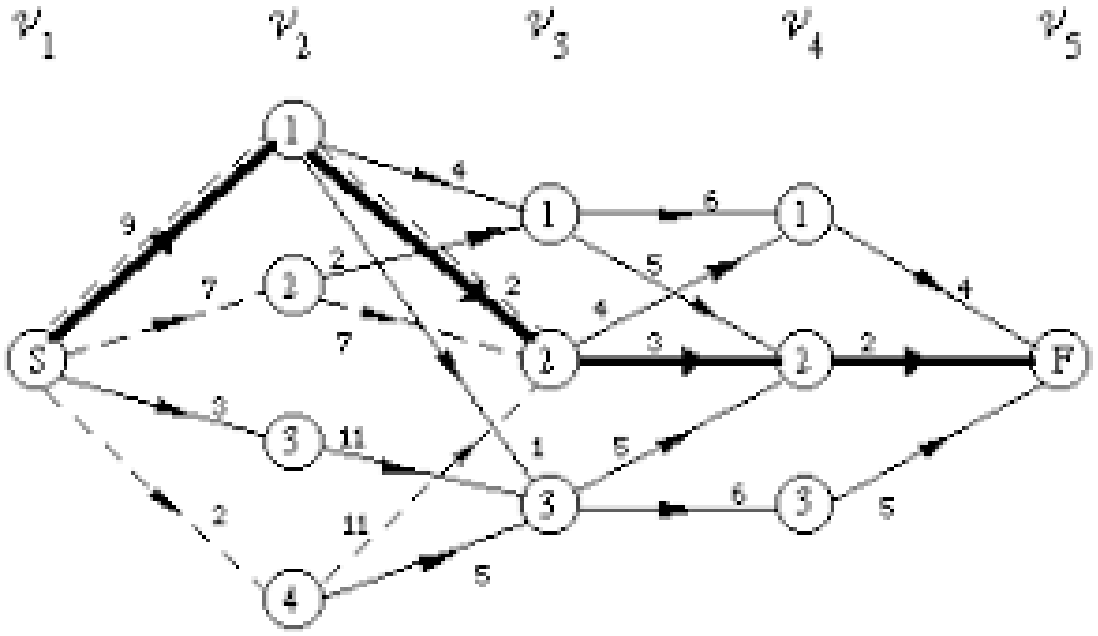
PRÁCTICA 4:

ALGORITMOS DE PROGRAMACIÓN DINÁMICA



Por:

Carmona Molina, José Antonio
Manzano Mata, Nuria
Rodríguez Rodríguez, Antonio

Algoritmo 1:

Diseño de resolución por etapas y ecuación recurrente:

- Resolución por etapas: El problema se puede resolver por etapas. En cada etapa se buscaría la relación entre los diferentes diccionarios para encontrar la traducción entre los dos idiomas en cuestión. Suponemos que la traducción de un idioma a sí mismo no se contemplaría (dándole el valor 0), ya que, no necesitaríamos diccionario adicional.
- Ecuación recurrente: La solución depende de las etapas (búsqueda de la relación entre diccionarios para la traducción de un idioma a otro), por lo que, dispondremos de una relación de idiomas que notaremos con el valor i y otra relación de idiomas que notaremos con el valor j. Llamaremos al valor T(i,j) a la relación entre dos idiomas mediante la traducción en diccionarios, por lo que, tendremos un camino de diccionarios a seguir para la llegada a la traducción desde el idioma_1 al idioma_2.

  En la etapa i consideramos que caben dos posibles decisiones:
  - Encontrar traducción directa: En tal caso, el valor que se almacena en la solución es el que le corresponda a la i,j y sus respectivas etiquetas (nombres de los diccionarios).
  - No encontrar traducción: En tal caso, puede ocurrir porque no haya traducción directa y buscando las traducciones mediante los diferentes diccionarios, tampoco se encuentre.
  - Encontrar traducción indirecta: Sucede cuando, al buscar primeramente la traducción directa y no la encuentre, observa parcialmente las traducciones que puede tener hasta llegar al idioma que se quiere traducir, obteniendo una relación de diccionarios por los que si se consigue llegar a traducir dicho idioma principal.

Con estas tres decisiones posibles, y dado que el problema es de minimización, tendríamos que

$$T_k(i, j) = \min \{T_k{-}1(i , j), T_k{-}1(i ,k) + T_k{-}1(k, j) \}, k \geq 1$$

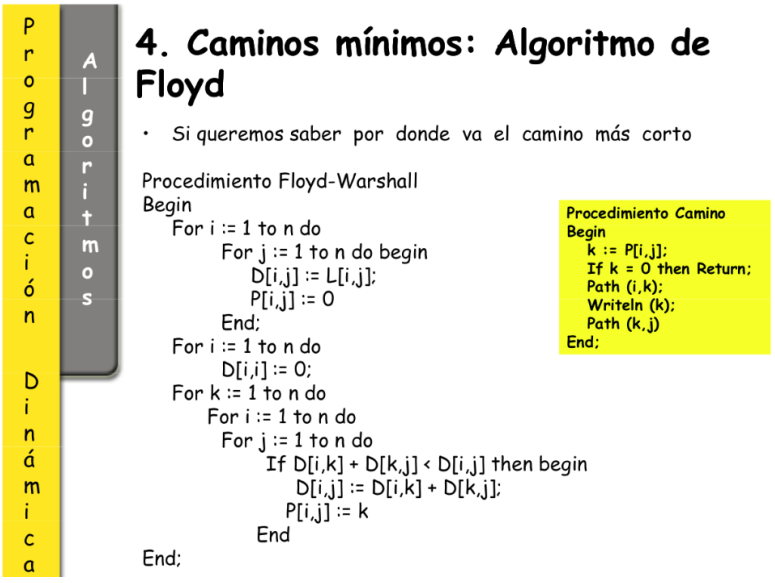
En la iteración k, el algoritmo tiene que chequear, para cada par de nodos (i,j), si existe o no un camino que pase a través de k que sea mejor que el actual camino minimal que solo pasa a través de los nodos {1,2,...,k-1}.
Hemos hecho uso del POB para calcular la longitud del camino más corto que pasa a través de k.

Diseño de la memoria: Para resolver el problema, T(i,j) se representa como una matriz i, j, cuyos valores son las diferentes asociaciones entre la traducción a través de diccionarios, que mediante un archivo txt, los añadimos a nuestro código. Se crea la matriz de adyacencia

suponiendo que las traducciones directas entre diccionarios se guardarán con 1 en la casilla correspondiente y el valor de 1000 para las casillas de la matriz de adyacencia que no posee traducción directa entre diccionarios.

Verificación del P.O.B.: En este algoritmo de programación dinámica observamos cómo se cumple el principio de optimalidad de Bellman, pues la función devuelve una respuesta tan óptima como la de todas las n etapas en las que se subdivide el problema. En todos los casos encontraremos las traducciones directas o indirectas entre idiomas, ya sea a través de un solo diccionario o haciendo uso de diccionarios puente, que se agregarán a un vector con camino más directo posible.

Diseño del algoritmo de cálculo de coste óptimo: Para realizar el cálculo de coste óptimo se procederá buscando el camino que menos traducciones requiera, para ello tomamos como caso base la posibilidad de traducción directa entre dos idiomas que se vería reflejado en nuestra matriz de adyacencia con un 1 entre ambos idiomas. Otro caso que permite acabar el problema rápido es la comprobación de que ambos idiomas se encuentran entre los disponibles, pues independientemente de si luego pudiera haber o no traducción, si no contamos con uno de los dos idiomas que buscamos traducir no sería posible la traducción.
Una vez comprobadas estas dos casuísticas entraría en juego realmente nuestro algoritmo con Programación Dinámica (traduccionOptima), cuya implementación veremos posteriormente y para el cual nos hemos basado en el algoritmo de camino mínimo de Floyd, con el que calcularemos la distancia mínima entre todos los puntos de nuestra matriz de adyacencia o grafo (traducciones directas) y luego mediante la función de recuperación buscaremos la distancia mínima entre las posiciones de los idiomas que buscamos en nuestra matriz de adyacencia, que nos dará finalmente la traducción óptima que requiere del uso de menos diccionarios intermediarios.



Diseño del algoritmo de recuperación de la solución: Como he comentado en el apartado anterior ahora buscaremos entre todas las distancias la mínima distancia entre los idiomas que queremos traducir con lo que obtendremos la traducción óptima. La función en cuestión sería "Camino" y podemos ver su implementación a continuación.
En todo momento sigo el algoritmo de camino mínimo de Floyd, en este caso el mismo que permite la recuperación del camino.

Implementación de los algoritmos de cálculo de coste óptimo y recuperación de la solución:

```
/**
 * Línea de ejecución: ./ejecutable prueba2.txt español aleman
 */

#include <fstream>
#include <iomanip>
#include <iostream>
#include <vector>

using namespace std;

class Matrix {
private:
    int ** _data; //array of double
    vector<string> rowLabels; //Language for row labels
    vector<string> colLabels; //Language for col labels



public:

    /**
     * @brief Set the Values object (default to 0)
     *
     * @param value
     */
    void setValues(int value = 1000) {
        for (int i = 0; i < rowLabels.size(); i++) {
            for (int j = 0; j < colLabels.size(); j++) {
                _data[i][j] = value;
            }
        }
    }

    /**
     * @brief Set languages availables (without repetitions)
     *
     * @param labels
     */
    void setLabels(vector<string> labels) {
        for (int i = 0; i < labels.size()-2; i++) {
            if (insertado(labels[i]) == -1) {
                rowLabels.push_back(labels[i]);
                colLabels.push_back(labels[i]);
```

```
        }
    }
}

void getLabels() {
    for (int i = 0; i < rowLabels.size(); i++)
        cout << rowLabels[i] << " ";
    cout << endl;
    for (int i = 0; i < colLabels.size(); i++)
        cout << colLabels[i] << " ";
    cout << endl;
}

const vector<string> & gettRowLabels() {
    return rowLabels;
}

const vector<string> & gettColLabels() {
    return colLabels;
}

/**
 * @brief Check if exist an element.
 *
 * @param language to search
 * @return int Pos of the language.
 */
int insertado(string & language) {
    for (int i = 0; i < rowLabels.size(); i++) {
        if (rowLabels[i] == language) return i;
    }
    return -1;
}

/**
 * @brief this method reserve memory to allocate a 2D matrix of size r x c.
 * @param r number of rows
 * @param c number of cols
 * @return the pointer to the memory block containing the data
 */
int ** allocate(int r, int c) {
    int ** block;
    // allocate memory into block
    block = new int * [r];
    for (int i = 0; i < r; i++) {
        block[i] = new int [c];
    }
    return block;
}

/**
 * @brief Free the memory allocated for the matrix, i.e., pointed by _data.
 * @post _data is set to nullptr
 */
void deallocate() {
    if (_data != nullptr) {
        for (int i = 0; i < rowLabels.size(); i++) {
```

```
                delete[] _data[i];
                _data[i] = nullptr;
            }
            delete[] _data;
        }
        _data = nullptr;
    }

    /**
     * @brief Create de adyaceny matrix
     *
     * @param labels Vector with the diferents languages.
     * @return Matrix
     */
    void MatrizAdy(vector<string> labels) {
        setLabels(labels); // Inserto los idiomas en las filas y columnas
        _data = allocate(rowLabels.size(), colLabels.size()); // Reservo memoria para la matriz.
        setValues(); // Inicializo toda la matriz a 0.
        int row, col;
         for (int i = 0; i < labels.size(); i+=2) { // Avanzo de 2 en 2 por la forma en la que leeré
del fichero de entrada.
            row = insertado(labels[i]);
            col = insertado(labels[i+1]);
            if (col != -1 && row != -1) {
                _data[row][col] = 1;
                _data[col][row] = 1;
            }
        } // Pongo a 1 las posiciones en las que existen diccionarios (bidireccional).
        //return *this;
    }

    int getValue(int row, int col) {
        return _data[row][col];
    }

    /**
     * @brief Copies the values in a 2D matrix org with nrows and ncols to an also 2D matrix dest.
     * It is assumed that org and dest have the memory properly allocated
     * @param dest destination matrix
     * @param org  source matrix
     * @param nrows number of rows
     * @param ncols number of cols
     * @param aux La he añadido para ahorrarme un bucle dentro del algoritmo.
     */
    void copy(int **dest, int **org, int nrows, int ncols, int ** aux) {
        for (int i = 0; i < nrows; i++) {
            for (int j = 0; j < ncols; j++) {
                dest[i][j] = org[i][j];
                aux[i][j] = -1;
            }
        }
    }

    vector<int> Camino(int **valores, int pos_idioma_1, int pos_idioma_2) {
        vector<int> solucion;
        int valor = valores[pos_idioma_1][pos_idioma_2];
        if (valor != -1) {
```

```
            solucion = Camino(valores, pos_idioma_1, valor);
            solucion.push_back(valor);
        }
        return solucion;
    }

    vector<string> traduccionOptima(int pos_idioma_1, int pos_idioma_2) {
        vector<string> solucion;
        vector<int> solucion_num;
        int **valores = allocate(rowLabels.size(), colLabels.size());
        int **copia = allocate(rowLabels.size(), colLabels.size());
        copy(copia, _data, rowLabels.size(), colLabels.size(), valores);
        for (int i = 0; i < rowLabels.size(); i++)
            copia[i][i] = 0;

        for (int k = 0; k < rowLabels.size(); k++) {
            for (int i = 0; i < rowLabels.size(); i++) {
                for (int j = 0; j < colLabels.size(); j++) {
                    if (copia[i][k] + copia[k][j] < copia[i][j]) {
                        int valor = copia[i][k] + copia[k][j];
                        copia[i][j] = valor;
                        valores[i][j] = k;
                    }
                }
            }
        }
        ////////////////////////////////////// IMPRIME LA MATRIZ DE CAMINOS
        cout << "Matriz de caminos: " << endl;
        for (int k = 0; k != rowLabels.size(); k++) {
            for (int i = 0; i < rowLabels.size(); i++) {
                cout << valores[k][i] << " ";
            }
            cout << endl;
        }
        cout << endl;
        //////////////////////////////////////
        solucion_num = Camino(valores, pos_idioma_1, pos_idioma_2);
        for (int i = 0; i < solucion_num.size(); i++) {
            solucion.push_back(colLabels[solucion_num[i]]);
        }

        return solucion;
    }

    friend std::ostream & operator<<(std::ostream & os, const Matrix & m);

};

////////////////////////////////////////////////////////////////////////////////////////////////////
////////

ostream & operator<<(ostream & flujo, const Matrix& m) {
    flujo << m.rowLabels.size() << " " << m.colLabels.size();
    flujo << setw(6) << std::right << "|" << "  ";
    for (int i = 0; i < m.colLabels.size(); i++)
        flujo << setw(6) << std::right << m.colLabels.at(i) << " ";
    flujo << endl << " ";
```

```
    for (int i = 0; i < m.rowLabels.size(); i++) {
        if (m.rowLabels.size() > 0)
            flujo << setw(9) << std::right << m.rowLabels.at(i);
        for (int j = 0; j < m.colLabels.size(); j++) {
            flujo << setw(7) << setprecision(2) << std::right << m._data[i][j] << " ";
        }
        flujo << endl;
    }
    return flujo;
}

////////////////////////////////////////////////////////////////////////////////////////////////////
/////

int main(int argc, char * argv[]) {
    // Comprobamos los argumentos de entrada.
    /**
     * El fichero de entrada debería tener una estructura tal que:
     *  Español Inglés
     *  Francés Aleman
     *  Portugués Italiano
     *  ...
     */
    if (argc < 4) {
         cout << "Error en los argumentos: ./ejecutable fichero_Diccionarios idioma_1 idioma_2" <<
endl;
        exit(-1);
    }

    // Hacemos la lectura de los datos de fichero en un vector.
    vector<string> labels;
    ifstream File1;
    File1.open(argv[1]);
    if (File1.fail()) {
        cout << "Error al abrir el archivo" << endl;
        exit(1);
    }
    char delimitador = ' ';
    while (!File1.eof()){
        string input1, input2;
        getline(File1, input1, delimitador);
        getline(File1, input2);
        labels.push_back(input1);
        labels.push_back(input2);
    }

    // Almacenamos nuestros datos en la matriz de adyacencia.

    Matrix matriz;
    matriz.MatrizAdy(labels);

    // Imprimimos nuestra matriz de adyacencia.
    cout << matriz << endl;

    // Comprobamos que ambos idiomas están entre los disponibles.
    string idioma_1 = argv[2];
```

```
    string idioma_2 = argv[3];
    int pos_idioma_1 = matriz.insertado(idioma_1);
    int pos_idioma_2 = matriz.insertado(idioma_2);
    // Caso base 1.
    if (pos_idioma_1 == -1 || pos_idioma_2 == -1) {
          cout << "No existe traducción posible, alguno de los lenguajes seleccionados no aparece en
ninguno de los diccionarios disponibles" << endl;
        matriz.deallocate();
        exit(-1);
    }

      // Comprobamos si hay traducción directa con el if, en caso contrario comprobaremos si es
posible la indirecta.
    vector<string> solucion;
    int value = matriz.getValue(pos_idioma_1, pos_idioma_2);
    // Caso base 2.
    if (value == 1)
          cout << "La traducción es posible y además es directa entre: " << idioma_1 << " y " <<
idioma_2 << endl;
    else {
        // Caso general.
        solucion = matriz.traduccionOptima(pos_idioma_1, pos_idioma_2);
        if (!solucion.empty()) {
            cout << "Resultando la traducción más corta posible mediante: " << idioma_1 << " ";
            for (int i = 0;i < solucion.size(); i++) {
                cout << solucion[i] << " ";
            }
            cout << idioma_2 << endl;
        } else
            cout << "No hay traducción posible mediante los diccionarios disponibles" << endl;
    }

    matriz.deallocate();
}
```

Pruebas de ejecución:

```
antonio@antonio-IdeaPad-Gaming-3-15ARH05:~/Descargas/PracticaPD/Programas_def$ g++ Ejercicio1.cpp -o ejer1
antonio@antonio-IdeaPad-Gaming-3-15ARH05:~/Descargas/PracticaPD/Programas_def$ ./ejer1 prueba2.txt español aleman
10 10     |  español frances ingles aleman portugues   ruso japones  chino hungaro italiano
  español   1000       1    1000    1000    1000    1000    1000    1000    1000    1000
  frances      1    1000    1000       1       1    1000    1000    1000    1000    1000
   ingles   1000    1000    1000       1    1000    1000    1000    1000    1000    1000
   aleman   1000       1       1    1000    1000    1000    1000    1000    1000       1
portugues   1000       1    1000    1000    1000    1000    1000    1000    1000    1000
     ruso   1000    1000    1000    1000    1000    1000       1    1000    1000    1000
  japones   1000    1000    1000    1000    1000       1    1000    1000    1000    1000
    chino   1000    1000    1000    1000    1000    1000    1000    1000       1    1000
  hungaro   1000    1000    1000    1000    1000    1000    1000       1    1000    1000
 italiano   1000    1000    1000       1    1000    1000    1000    1000    1000    1000

Matriz de caminos:
-1 -1 3 1 1 -1 -1 -1 -1 3
-1 -1 3 -1 -1 -1 -1 -1 -1 3
3 3 -1 -1 3 -1 -1 -1 -1 3
1 -1 -1 -1 1 -1 -1 -1 -1 -1
1 -1 3 1 -1 -1 -1 -1 -1 3
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1
3 3 3 -1 3 -1 -1 -1 -1 -1

Resultando la traducción más corta posible mediante: español frances aleman
```

Algoritmo 2:

Diseño de resolución por etapas y ecuación recurrente:

- Resolución por etapas: El problema se puede resolver por etapas. En cada etapa se buscaría el camino donde se consiga la mayor suma de dinero posible. Suponemos que partimos desde la posición superior derecha de un mapa y hay que esquivar los muros hasta llegar a la salida que se encontraría en la esquina inferior izquierda .
- Ecuación recurrente: La solución depende de las etapas (obtener el camino con la mayor suma posible de dinero hasta llegar a la salida), por lo que, dispondremos de un mapa con diferentes casillas en las que se encuentran en algunas unas bolsas de dinero, un muro o vacío. Donde las filas de dicho mapa lo notaremos como i y las columnas como j. Llamaremos al valor T(i,j) al camino obtenido como el que más número de bolsas de oro pueda conseguir.
  En la etapa i consideramos que caben tres posibles decisiones:
  - Encontrarnos un muro: En tal caso, no podremos pasar por esa casilla y tendríamos que buscar porque otra casilla nos conviene más acceder.
  - Encontrarnos una casilla vacía: En tal caso, es necesario evaluar si en las demás casillas hay una bolsa de oro. por lo que, desecharíamos esta casilla vacía o si no es el caso, observar qué ocurre si elegimos una u otra, guardando los valores finales y comparándolos.
  - Encontramos una bolsa de oro: Automáticamente sabemos que hay que pasar por esa casilla sí o sí, sin necesidad de comprobación futura.

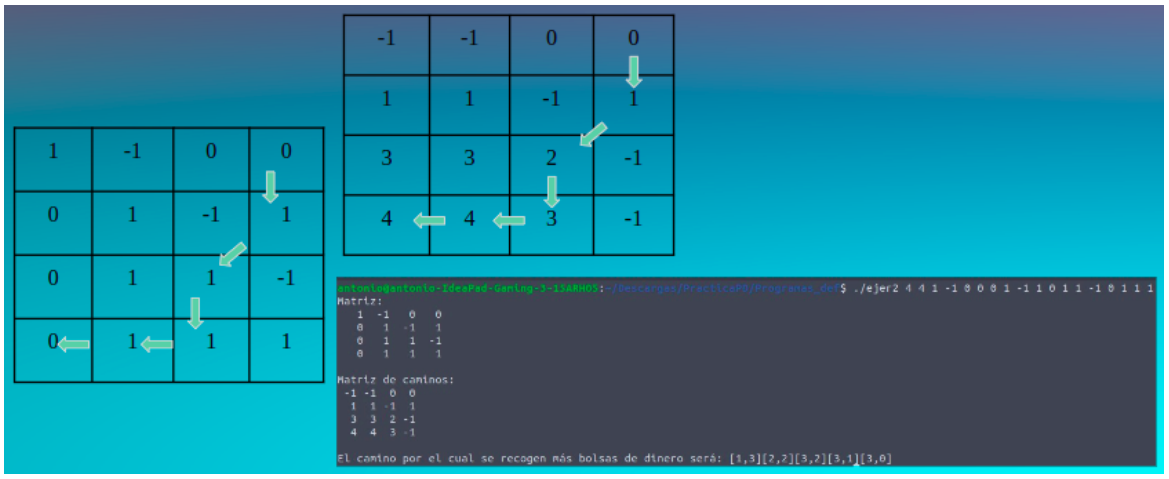
Con estas tres decisiones posibles, y dado que el problema es de maximización, tendríamos que la ecuación de recurrencia sería:

$$T(i,j)= \max\{T(i-1, j), T(i-1, j+1), T(i,j+1)\}$$

Diseño de la memoria: Para resolver el problema, T(i,j) se representa como una matriz con i filas y j columnas, que utilizaremos como tablero para hallar los posibles caminos del jugador. En cada casilla de la matriz habrá un valor entre -1 y 1, representando cada uno un objeto dentro del mapa. Las casillas superior derecha e inferior izquierda se marcarán con el valor 0, pues son el lugar de partida y la línea de meta. Las casillas vacías se marcarán con el valor 0 y las que contengan una bolsa de oro tendrán un 1. Finalmente decidimos darle el valor -1 a los muros para que nuestro algoritmo detecte los caminos con muros siempre menos eficientes que el resto, donde se priorizará recoger tantas bolsas de oro como sea posible.

Verificación del P.O.B.: Como vemos en la imagen, el camino óptimo estará compuesto por subcaminos donde se tomará la mejor solución, es decir donde se sume el valor máximo,

pues se entiende como un camino más beneficioso aquél en el que recojamos más bolsas de oro, las cuales tienen valor 1.



En este caso observamos que cada avance del jugador por una casilla siempre es a una casilla con oro, si es posible, y siempre evitando los muros. La mejor solución para avanzar a la primera casilla del camino solución es hacia abajo, en diagonal para la segunda pues las demás opciones implican un muro, la tercera corresponderá al camino hacia abajo; y así sucesivamente, para llegar a cualquier etapa se sigue un camino óptimo.

Diseño del algoritmo de cálculo de coste óptimo: Para calcular un camino de coste óptimo para llegar desde la casilla de entrada a la de salida maximizando el número de bolsas de oro, haríamos uso de una matriz solución en la que almacenaríamos la suma de los valores máxima por un camino a cada casilla seleccionada, inicializando a 0 la casilla de salida y la de llegada, a -1 las casillas que son inaccesibles dado a que existe un muro y con 1 las bolsas de oro. Vamos comprobando las casillas colindantes y según el valor que tengan y en función el valor avanzamos o no por ellas, obteniendo así el cálculo óptimo que nos daría la solución final. Esto se realiza a través de comprobaciones dado que, si nos encontramos un -1 no pasamos por esa casilla, en caso contrario, si nos encontramos un 1 avanzamos y sumamos en la matriz donde almacenamos los caminos.
De hecho si pensamos, si nos encontramos un muro en una fila o columna, sabemos que por esa fila o columna respectivamente no encontraremos la solución, pues recordemos que solo podemos avanzar hacia la izquierda, hacia abajo o en diagonal en dirección a la salida.
Nuestra casilla va comprobando los valores que podría haber tomado antes de llegar a esa casilla comprobando los valores anteriores (de los que puede provenir) y quedándose con el mejor (mayor valor), téngase en cuenta que ahora estaremos comprobando en la matriz solución.
A continuación podemos ver el código del algoritmo “CaminoOptimo”, donde se puede ver más claramente el funcionamiento del programa.
Cabe mencionar que en el diseño de este algoritmo no nos hemos basado en ningún algoritmo conocido y lo hemos hecho de forma propia, tal y como nos recomendó nuestro profesor de

prácticas. Además se verifica que es un algoritmo de programación dinámica, pues en cada iteración el tamaño del programa aumenta en 1.

Diseño del algoritmo de recuperación de la solución: Este algoritmo lo que busca es mediante comprobaciones en la matriz solución donde hemos ido almacenando el máximo valor posible en cada casilla, recogemos el camino que permita optimizar la solución, es decir, el que nos dé el mayor valor, que hemos ido recogiendo; el mayor número posible de bolsas de dinero. La función en cuestión es GuardarCamino y podemos ver su implementación a continuación.
Finalmente nos quedamos con un vector de pares de enteros que contiene las posiciones por las que deberá pasar el camino que optimiza la solución.

Implementación de los algoritmos de cálculo de coste óptimo y recuperación de la solución:

```
/**
* Línea de ejecución: ./ejecutable 4 5 1 -1 0 0 0 0 1 -1 1 -1 0 1 1 1 1 0 1 1 1 1
*/

#include <iostream>
#include <vector>
#include <iomanip>

using namespace std;

/**
* Elementos de la matriz:
* Bolsa de dinero --> 1.
* Casilla vacia --> 0.
* Muro --> -1.
* Entrada --> -0.
* Salida --> 0.
*/
////////////////////////////////////////////////////// COMPROBAMOS LOS DATOS.

void ComprobarEntradasSalidas(vector<vector<int>> matriz, int filas, int columnas) {
   for (int i = 0; i < filas; i++) {
       for (int j = 0; j < columnas; j++) {
           int elemento = matriz[i][j];
           if(elemento < -1 || elemento > 1) {
                 cout << "Error: Hay algún elemento que no se corresponde a ninguno de los elementos
posibles" << endl;
               cout << "Intervalo 1,1" << endl;
               exit(-1);
           }
       }
   }
}
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////
//////

int ** allocate(int r, int c) {
   int ** block;
   // allocate memory into block
   block = new int * [r];
   for (int i = 0; i < r; i++) {
       block[i] = new int [c];
   }
   return block;
}

int MejorCasilla(int lado, int diagonal, int abajo) {
   int elegido;
   if (lado > diagonal) {
       if (lado > abajo)
           elegido = lado;
       else
           elegido = abajo;
   } else {
       if (abajo > diagonal)
           elegido = abajo;
       else
           elegido = diagonal;
   }
   return elegido;
}

void InicializaraCero(int ** caminos, int filas, int columnas) {
   for (int i = 0; i < filas; i++) {
       for (int j = 0; j < columnas; j++) {
           caminos[i][j] = -1;
       }
   }
}

void CaminoOptimo(vector<vector<int>> matriz, int ** caminos, int filas, int columnas) {

   InicializaraCero(caminos, filas, columnas);

////////////  Si encontramos un muro en la fila sabemos que las posiciones que le siguen son
inaccesibles
   for (int j = columnas - 2; j >= 0; --j) {
       caminos[0][j] = matriz[0][j] + caminos[0][j-1];
       if (matriz[0][j] == -1) {
           for (; j >= 0; j--) {
               caminos[0][j] = -1;
           }
       }
   }

////////////  Si encontramos un muro en la columna sabemos que las posiciones que le siguen son
inaccesibles
   caminos[0][columnas-1] = matriz[0][columnas-1];
   for (int i = 1; i < filas; ++i) {
       caminos[i][columnas - 1] = matriz[i][columnas - 1] + caminos[i-1][columnas-1];
```

```
        if (matriz[i][columnas - 1] == -1) {
            for (; i < filas; i++) {
                caminos[i][columnas - 1] = -1;
            }
        }
    }
    cout << endl;

////////////// Vamos iterando fila a fila y vamos sumando a la casilla en la que nos encontramos los 3
valores de los que podemos venir.

    for (int i = 1; i < filas; ++i) {
        for (int j = columnas - 2; 0 <= j; --j) {
            if (matriz[i][j] == -1) {
                caminos[i][j] = -1;
            } else if (caminos[i - 1][j] == -1 && caminos[i - 1][j + 1] == -1 && caminos[i][j + 1] ==
-1) {
                caminos[i][j] = -1;
            } else {
                caminos[i][j] = matriz[i][j] + MejorCasilla(caminos[i - 1][j], caminos[i - 1][j + 1],
caminos[i][j + 1]);
            }
        }
    }
}

vector<pair<int, int>> GuardarCamino(int** caminos, int filas, int columnas) {
    vector<pair<int, int>> solucion;
    pair<int, int> aux;
    int i = filas - 1, j = 0;
    aux.first = i;
    aux.second =j;
    solucion.push_back(aux);
    while (i != 0 && j != columnas - 1) {
        if (caminos[i - 1][j] > caminos[i - 1][j + 1]) {
            if (caminos[i - 1][j] > caminos[i][j + 1]) {
                i--;
            } else {
                j++;
            }
        } else {
            if (caminos[i][j + 1] > caminos[i - 1][j + 1]) {
                j++;
            } else {
                i--;
                j++;
            }
        }
        aux.first = i;
        aux.second =j;
        solucion.push_back(aux);
    }
    return solucion;
}

int main(int argc, char * argv[]) {
    if (argc < 3) {
```

```
        cout << "Error en los argumentos: ./ejecutable filas * columnas" << endl;
        exit(-1);
    }
    int filas = stoi(argv[1]);
    int columnas = stoi(argv[2]);

    if (filas * columnas != argc - 3) {
        cout << "Número de elementos del matriz incorrecto: filas * columnas" << endl;
        exit(-1);
    }

    vector<vector<int>> matriz;
    vector<int> aux;

    int cont = 0;
    int n = 0;
    for (int i = 3; i < argc; i++) {
        aux.push_back(stoi(argv[i]));
        cont++;
        if (cont == columnas) {
            cont = 0;
            matriz.push_back(aux);
            aux.clear();
            n++;
        }
    }

    ComprobarEntradasSalidas(matriz, filas, columnas);

    cout << "Matriz: " << endl;
    for (int i = 0; i < filas; i++) {
        for (int j = 0; j < columnas; j++)
            cout << " " << setw(3) << setprecision(2) << matriz[i][j];
        cout << endl;
    }

    int** caminos = allocate(filas, columnas);
    CaminoOptimo(matriz, caminos, filas, columnas);

    cout << "Matriz de caminos: " << endl;
    for (int i = 0; i < filas; ++i) {
        for (int j = 0; j < columnas; ++j) {
            cout << " " << setw(2) << setprecision(2) << caminos[i][j];
        }
        cout << endl;
    }
    cout << endl;

    vector<pair<int, int>> posiciones = GuardarCamino(caminos, filas, columnas);

    cout << "El camino por el cual se recogen más bolsas de dinero será: ";
    for (int i = posiciones.size()-1; i >= 0; i--)
        cout << "[" << posiciones[i].first << "," << posiciones[i].second << "]";
    cout << endl;

    return 0;
}
```

Pruebas de ejecución:

```
antonio@antonio-IdeaPad-Gaming-3-15ARH05:~/Descargas/PracticaPD/Programas_def$ ./ejer2 4 5 1 -1 0 0 0 0 1 -1 1 -1 0 1 1 1 1 0 1 1 1 1
Matriz:
   1  -1   0   0   0
   0   1  -1   1  -1
   0   1   1   1   1
   0   1   1   1   1

Matriz de caminos:
 -1 -1  0  0  0
  1  1 -1  1 -1
  4  4  3  2 -1
  5  5  4  3 -1

El camino por el cual se recogen más bolsas de dinero será: [0,4][1,3][2,3][3,3][3,2][3,1][3,0]
```

```
antonio@antonio-IdeaPad-Gaming-3-15ARH05:~/Descargas/PracticaPD/Programas_def$ ./ejer2 4 4 1 -1 0 0 0 1 -1 1 0 1 1 -1 0 1 1 1
Matriz:
   1  -1   0   0
   0   1  -1   1
   0   1   1  -1
   0   1   1   1

Matriz de caminos:
 -1 -1  0  0
  1  1 -1  1
  3  3  2 -1
  4  4  3 -1

El camino por el cual se recogen más bolsas de dinero será: [1,3][2,2][3,2][3,1][3,0]
```

Algoritmo 3:

Diseño de resolución por etapas y ecuación recurrente:

- Resolución por etapas: El problema se puede resolver por etapas. En cada etapa elegimos basándonos en el resultado almacenado anteriormente, cual es el camino con menor valor de la sumatoria total de dificultad de los terrenos. Suponemos que partimos desde la esquina inferior izquierda y el lugar de llegada es la esquina superior derecha.
- Ecuación recurrente: La solución depende de las etapas (obtener el camino con la máxima batería posible de la sonda hasta llegar a la salida), por lo que, dispondremos de un mapa con diferentes casillas en las que se encuentran diferentes terrenos con una dificultad determinada. Donde las filas de dicho mapa lo notaremos como i y las columnas como j. Llamaremos al valor T(i,j) al camino obtenido como el que se pierde menos batería de la sonda, es decir, el más fácil.
  En la etapa i consideramos encontrarnos con un tipo de terreno concreto que caben posibles decisiones:
  - Encontrar un terreno con dificultad mayor: En tal caso, buscaríamos otro terreno posible con menor dificultad, siempre y cuando la sumatoria total de todos hasta llegar al destino no supere la que ya hay almacenada.
  - Encontrar un terreno con misma dificultad: En tal caso, es necesario evaluar si en los demás terrenos al llegar al destino dicha sumatoria es menor que la que ya hay almacenada, si no, siempre escogeremos la de menor sumatoria total.

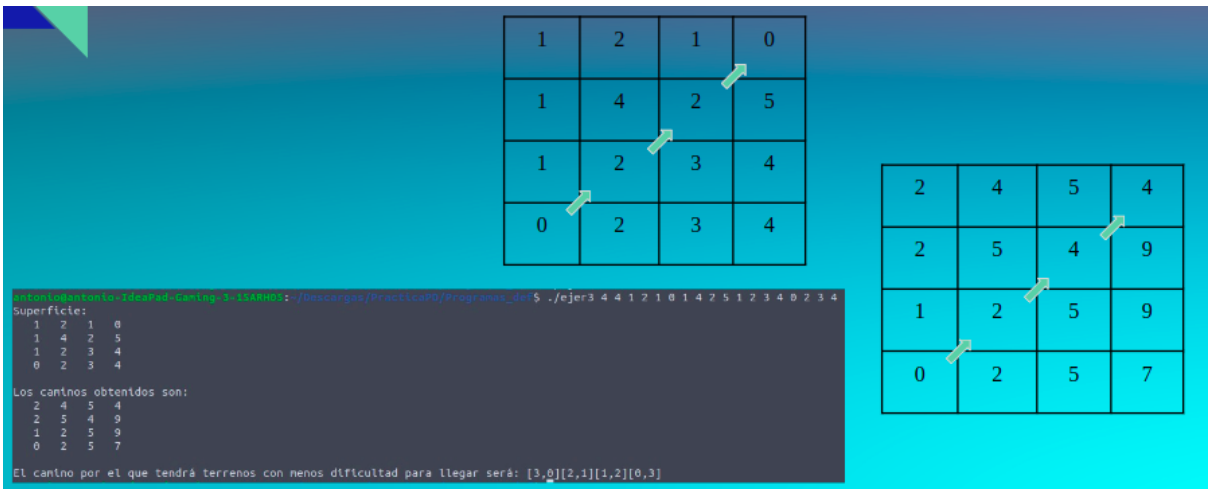
- Encontrar un terreno con dificultad menor: Automáticamente sabemos que hay que pasar por dicho terreno, ya que, en teoría será la decisión más acertada para obtener lo que buscamos.

Con estas tres decisiones posibles, y dado que el problema es de maximización, tendríamos que la ecuación recurrente se expresa como:

$$T(i,j)= \max\{T(i+1, j), T(i+1, j-1), T(i, j-1)\}$$

Diseño de la memoria: Para resolver el problema, T(i,j) se representa como una matriz con n filas y m columnas, que utilizaremos como tablero para hallar los posibles caminos de la sonda. En cada casilla de la matriz habrá un valor entre 1 e infinito, representando el grado de dificultad o coste en batería para el recorrido de la sonda. Las casillas inferior izquierda y superior derecha se marcarán con el valor 0, pues son el comienzo y el fin.
(insertar algoritmo)

Verificación del P.O.B.: En este algoritmo encontramos de nuevo la verificación del principio de optimalidad de Bellman, ya que como contemplamos en esta imagen, todos los subcaminos que conforman el camino final son óptimos. En este caso para llegar a la segunda casilla el camino más óptimo es la diagonal, a la tercera también es la diagonal y así hasta conformar el camino hasta la nave.



Diseño del algoritmo de cálculo de coste óptimo: La búsqueda del cálculo de coste óptimo se realizará a través de la maximización de la bateria de una sonda i pasando por diferentes tipos de terrenos (cada uno consume bi de batería de la sonda) hasta el destino, obteniendo la menor pérdida posible de batería por parte de la sonda, lo que se conseguiría pasando por los terrenos que ofrezcan una menor dificultad de paso.

Para la implementación de dicho programa utilizamos la función "CaminoOptimo", en primer lugar tomaríamos una matriz auxiliar que inicializamos a un valor grande (1000) y en esta vamos escribiendo la suma de las dificultades por las que tendríamos que pasar para llegar a la casilla actual, eligiendo siempre el camino que menos dificultad presente, además vamos guardando el camino por el que vamos pasando en nuestra matriz de par de enteros y en base a la cuál iremos actualizando o recuperando la solución mediante la función "recuperarCamino", quedándonos finalmente con un vector de pares de enteros que contendrá las posiciones por las que debemos de pasar para ahorrar la máxima batería al tener menos dificultad de paso.

Diseño del algoritmo de recuperación de la solución: En este diseño de algoritmo utilizamos recursividad para hacer llamadas a la propia función "recuperarCamino" en la que, dándole una posición concreta del terreno, donde se obtiene el valor de esa posición y la almacena en un vector de pares (i,j) mientras que la i no sea la última posición de las filas (dado a que sería donde parte la sonda) y que la j no sea 0, dado a que sería la salida de la sonda, el valor de estos se iría actualizando en cada llamada recursiva. Por último, devuelve el vector de pares de enteros con las posiciones que optimizan el camino y que será la solución al problema.

Implementación de los algoritmos de cálculo de coste óptimo y recuperación de la solución:

```
/**
* Línea de ejecución: ./ejecutable 3 3 1 2 0 1 4 2 0 2 3
*/

#include <iostream>
#include <vector>
#include <iomanip>


using namespace std;

/**
* Elementos de la matriz:
* Salida y llegada --> 0
* Dificultad en rango de 1 a INFINITO.
*/
void ComprobarEntradasSalidas(vector<vector<int>> matriz, int filas, int columnas) {
   if (matriz[0][columnas-1] != 0 || matriz[filas-1][0] != 0) {
       cout << "Error en la posición de las casillas salida-llegada del terreno" << endl;
       exit(-1);
   }

   int control = 0;
   for (int i = 0; i < filas; i++) {
```

```
        for (int j = 0; j < columnas; j++) {
            int elemento = matriz[i][j];
            if(elemento == 0) {
                control++;
            }
            if(elemento < 0) {
                cout << "Error: No puede haber dificultad negativa, mínimo será 1" << endl;
                exit(-1);
            }
            if (control > 2) {
                cout << "Error: Se ha utilizado más de una entrada o más de una salida" << endl;
                exit(-1);
            }
        }
    }
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////
/////////////////

int ** allocate(int r, int c) {
    int ** block;
    // allocate memory into block
    block = new int * [r];
    for (int i = 0; i < r; i++) {
        block[i] = new int [c];
    }
    return block;
}

pair<int, int> ** allocate_pair(int r, int c) {
    pair<int, int> ** block;
    // allocate memory into block
    block = new pair<int, int> * [r];
    for (int i = 0; i < r; i++) {
        block[i] = new pair<int, int> [c];
    }
    return block;
}

int MejorCasilla(pair<int, int> ** temp, int lado, int diagonal, int arriba, int i, int j) {
    int elegido;
    if (lado < diagonal) {
        if(lado < arriba){
            elegido = lado;
            temp[i][j].first= i;
            temp[i][j].second= j-1;
        }else{
            elegido = arriba;
            temp[i][j].first= i+1;
            temp[i][j].second= j;
        }
    } else {
        if(arriba < diagonal) {
            elegido = arriba;
            temp[i][j].first= i+1;
            temp[i][j].second= j;
```

```
        } else {
            elegido = diagonal;
            temp[i][j].first= i+1;
            temp[i][j].second= j-1;
        }
    }
    return elegido;
}

vector <pair <int, int>> recuperarCamino(pair<int, int> ** temp, int filas, int fila_pos, int
columna_pos){
    vector <pair <int, int>> solucion;
    pair <int, int> aux;
    int i = temp[fila_pos][columna_pos].first;
    int j = temp[fila_pos][columna_pos].second;
    if(i != filas-1 || j  != 0) {
        solucion= recuperarCamino(temp, filas, i, j);
        aux.first = i;
        aux.second = j;
        solucion.push_back(aux);
    }

    return solucion;
}

void setValues(int ** caminos, int filas, int columnas, int value = 1000) {
    for (int i = 0; i < filas; i++) {
        for (int j = 0; j < columnas; j++) {
            caminos[i][j] = value;
        }
    }
    caminos[filas-1][0]= 0; // Llegada
    caminos[0][columnas-1]= 0; // Salida
}

vector <pair <int, int>> CaminoOptimo(vector<vector<int>> terreno, int filas, int columnas){
    int ** caminos = allocate(filas, columnas);
    pair<int, int> ** temp = allocate_pair(filas, columnas);
    vector <pair <int, int>> solucion;
    pair <int, int> aux;

    setValues(caminos, filas, columnas);

    for(int i=filas-2; i>=0; --i){
        caminos[i][0] = terreno[i][0] + terreno[i+1][0];
        temp[i][0].first = i+1;
        temp[i][0].second = 0;
    }

    for(int i= 1; i < columnas; i++){
        caminos[filas-1][i]= terreno[filas-1][i] + terreno[filas-1][i-1];
        temp[filas-1][i].first = filas-1;
        temp[filas-1][i].second = i-1;
    }

    for(int i = filas-2; i >= 0; i--){
        for(int j= 1; j<columnas; j++){
```

```
            caminos[i][j]= terreno[i][j] + MejorCasilla(temp, caminos[i][j-1], caminos[i+1][j-1],
caminos[i+1][j], i, j);
        }
    }

    solucion = recuperarCamino(temp, filas, 0, columnas-1);

    cout << endl << "Los caminos obtenidos son: " << endl;
    for (int i = 0; i < filas; ++i) {
        for (int j = 0; j < columnas; ++j) {
            cout << " " << setw(3) << setprecision(2) << caminos[i][j];
        }
        cout << endl;
    }

    return solucion;

}
////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////

int main(int argc, char * argv[]) {
    if (argc < 3) {
        cout << "Error en los argumentos: ./ejecutable filas * columnas" << endl;
        exit(-1);
    }
    int filas = stoi(argv[1]);
    int columnas = stoi(argv[2]);

    if (filas * columnas != argc - 3) {
        cout << "Número de elementos del terreno incorrecto: filas * columnas" << endl;
        exit(-1);
    }

    vector<vector<int>> matriz;
    vector<int> aux;

    int cont = 0;
    int n = 0;
    for (int i = 3; i < argc; i++) {
        aux.push_back(stoi(argv[i]));
        cont++;
        if (cont == columnas) {
            cont = 0;
            matriz.push_back(aux);
            aux.clear();
            n++;
        }
    }

    ComprobarEntradasSalidas(matriz, filas, columnas);

    cout << "Superficie: " << endl;
    for (int i = 0; i < filas; i++) {
        for (int j = 0; j < columnas; j++)
            cout << " " << setw(3) << setprecision(2) << matriz[i][j];
        cout << endl;
```

```
    }

    vector<pair<int, int>> posiciones = CaminoOptimo(matriz, filas, columnas);
    pair <int, int> in_out;
    if(posiciones.size()>0){
        in_out.first = filas-1;
        in_out.second = 0;
        posiciones.insert(posiciones.begin(), in_out);
        in_out.first = 0;
        in_out.second = columnas-1;
        posiciones.push_back(in_out);
    }

    cout << endl << "El camino por el que tendrá terrenos con menos dificultad para llegar será: ";
    for (int i = 0; i < posiciones.size(); i++)
        cout << "[" << posiciones[i].first << "," << posiciones[i].second << "]";
    cout << endl;

    return 0;
}
```

Pruebas de ejecución:

```
antonio@antonio-IdeaPad-Gaming-3-15ARH05:~/Descargas/PracticaPD/Programas_def$ ./ejer3 4 4 1 2 1 0 1 4 2 5 1 2 3 4 0 2 3 4
Superficie:
   1   2   1   0
   1   4   2   5
   1   2   3   4
   0   2   3   4

Los caminos obtenidos son:
   2   4   5   4
   2   5   4   9
   1   2   5   9
   0   2   5   7

El camino por el que tendrá terrenos con menos dificultad para llegar será: [3,0][2,1][1,2][0,3]
```

```
antonio@antonio-IdeaPad-Gaming-3-15ARH05:~/Descargas/PracticaPD/Programas_def$ g++ Ejercicio3.cpp -o ejer3
antonio@antonio-IdeaPad-Gaming-3-15ARH05:~/Descargas/PracticaPD/Programas_def$ ./ejer3 3 3 1 2 0 1 4 2 0 2 3
Superficie:
   1   2   0
   1   4   2
   0   2   3

Los caminos obtenidos son:
   2   3   3
   1   4   4
   0   2   5

El camino por el que tendrá terrenos con menos dificultad para llegar será: [2,0][1,0][0,1][0,2]
```