Práctica 4 Algorítmica:

PROGRAMACIÓN DINÁMICA

Por:

José A. Carmona Molina

Nuria Manzano Mata

Antonio Rodríguez Rodríguez

Algoritmo 1: Traducción de textos

Diseño de resolución por etapas y ecuación recurrente

- En cada etapa se busca la relación entre los distintos diccionarios para encontrar la traducción de los dos idiomas en cuestión, sin contemplar la traducción de un idioma a sí mismo.

- Ecuación recurrente, posibles decisiones por etapa:
  - Encontrar traducción directa
  - No encontrar traducción
  - Encontrar traducción indirecta

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

Diseño de la memoria

```
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 i filas y j columnas
- Valores entre 1 y 1000 (INF) :
    - 1 -> Casillas con traducción directa
    - 1000 -> Casilla sin traducción.

Verificación del P.O.B.

```
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
```

Diseño del algoritmo de cálculo de coste óptimo

```
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;
}
```

Diseño del algoritmo de recuperación de la solución

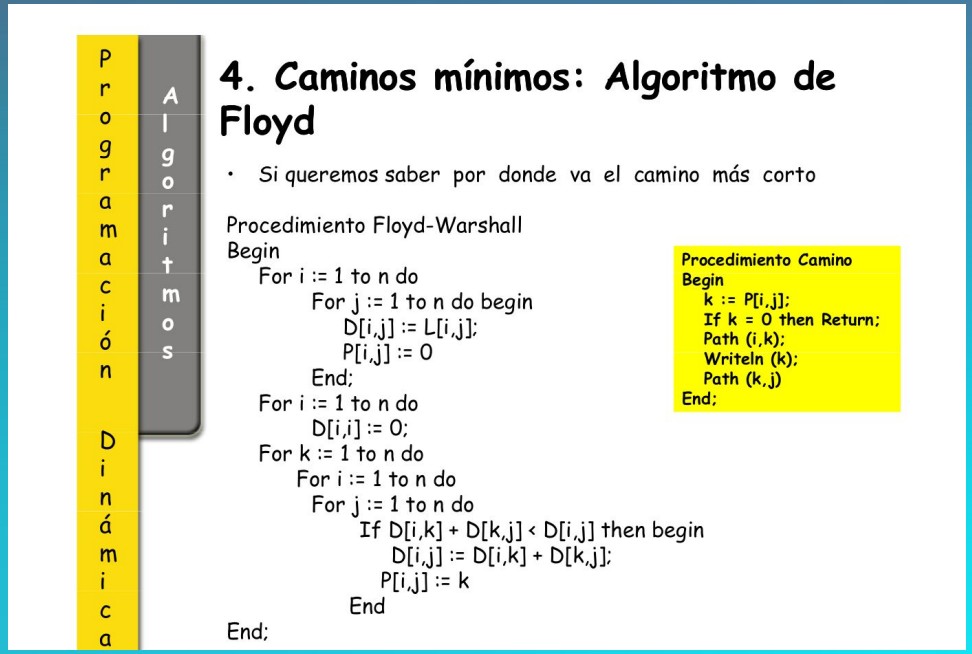
```
vector<int> Camino(int **valores, int pos_idioma_1, int pos_idioma_2) {
    vector<int> solucion;
    int valor = valores[pos_idioma_1][pos_idioma_2];
    if (valor != -1) {
        solucion = Camino(valores, pos_idioma_1, valor);
        solucion.push_back(valor);
    }
    return solucion;
}
```

Implementación de los algoritmos

```
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;
}
```

Implementación de los algoritmos



Algoritmo 2: Videojuego

Diseño de resolución por etapas y ecuación recurrente

- En cada etapa buscaremos la casilla que más oro proporcione al jugador, indicada con unos valores de preferencia a la hora de escogerla, pues la mejor decisión individual no garantiza el mejor camino total.

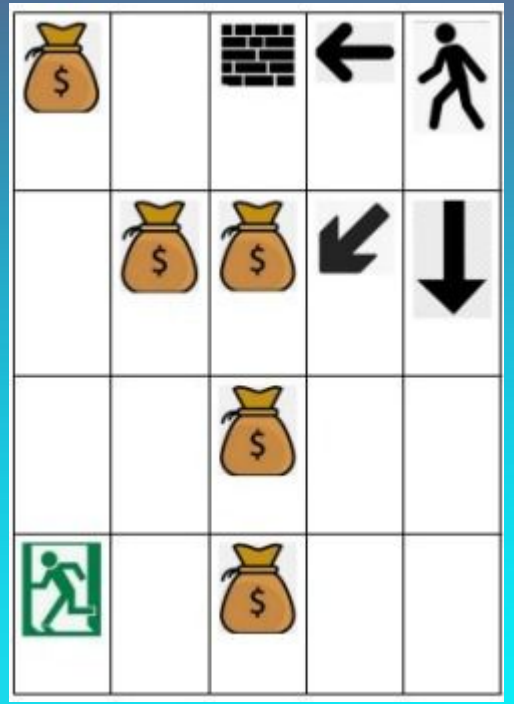
- Ecuación recurrente, posibilidades por etapa:

  - Encontrar un muro, el valor del camino asciende a infinito, se descarta.
  - Encontrar una casilla vacía, se baraja pero se da preferencia a las que tienen oro.
  - Encontrar una casilla con oro, se le dará siempre prioridad.

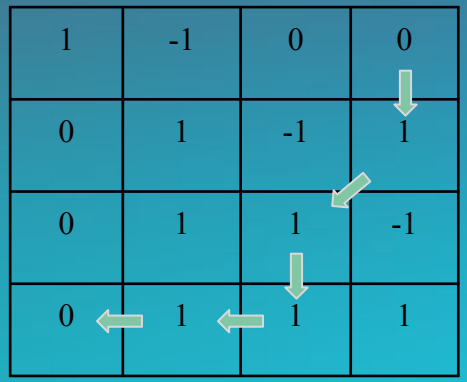
$$T(i,j)= \max\{T(i-1, j), T(i-1, j+1), T(i,j+1)\}$$

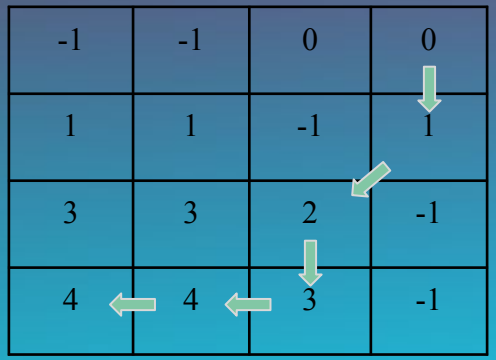
Diseño de la memoria

- Matriz de i filas y j columnas
- Valores entre -1 y 1:
  - 0 -> Casilla de entrada
  - 1 -> Casilla con bolsa de oro
  - 0 -> Casilla vacía
  - 0 -> Casilla de salida
  - -1 -> Muro



Verificación del P.O.B.






```
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]
```

Diseño del algoritmo de cálculo de coste óptimo

```
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 - 1; j >= 0; --j) {
        caminos[0][j] = matriz[0][j];
        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

    for (int i = 0; i < filas; ++i) {
        caminos[i][columnas - 1] = matriz[i][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]);
            }
        }
    }
}
```

Diseño del algoritmo de recuperación de la solución

```
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;
}
```

Implementación de los algoritmos

```
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 - 1; j >= 0; --j) {
        caminos[0][j] = matriz[0][j];
        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

    for (int i = 0; i < filas; ++i) {
        caminos[i][columnas - 1] = matriz[i][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;
}
```

Algoritmo 3: Sonda Espacial

Diseño de resolución por etapas y ecuación recurrente

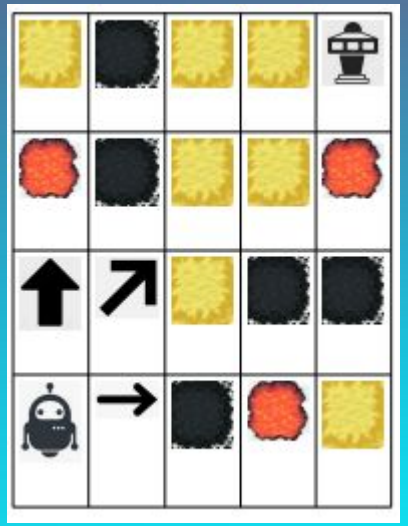
- En cada etapa buscaremos la casilla que menos batería cueste a la sonda, indicada con unos valores de preferencia a la hora de escogerla, pues la mejor decisión individual no garantiza el mejor camino total.

- Ecuación recurrente:

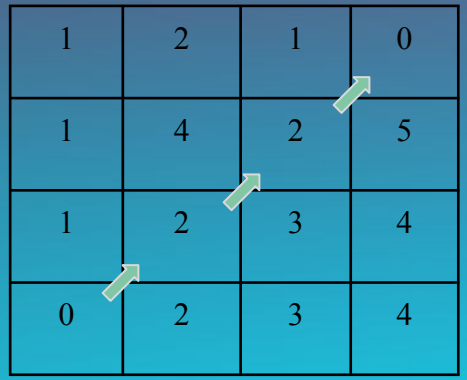
$$T(i,j)= \max\{T(i+1, j), T(i+1, j-1), T(i, j-1)\}$$

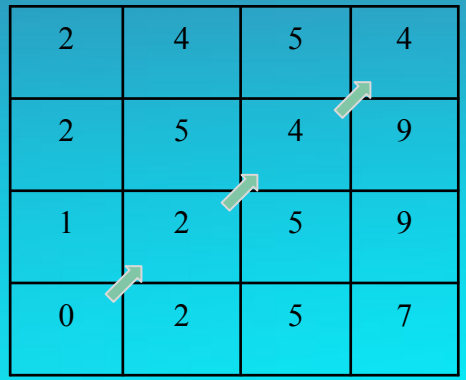
Diseño de la memoria

- Matriz de i filas y j columnas
- Valores entre 0 y n:
  - 0 -> Casilla de entrada
  - 0 -> Casilla de salida
  - n -> Casillas terrenos



Verificación del P.O.B.






```
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]
```

Diseño del algoritmo de cálculo de coste óptimo

```
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;
}
```

```
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;

}
```

Diseño del algoritmo de recuperación de la solución

```
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;
}
```

Implementación de los algoritmos

```cpp
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;
}
```

```cpp
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;
}
```

Implementación de los algoritmos

```
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;

}
```