MEMORIA

OBTENCIÓN DEL CAMINO MÍNIMO
MEDIANTE RAMIFICACIÓN Y PODA

UNIVERSIDAD DE ALICANTE

ANALISIS Y DISEÑO DE ALGORITMOS

ILYAS UMALATOV

X7278165E

INDICE

1. ESTRUCTURA DE DATOS

1.1 Nodo

En este código, la estructura Node representa cada nodo en el proceso de ramificación y poda. Aquí está la estructura Node:

```
struct Node
{
    int x, y;
    int cost;
    vector<pair<int, int>> path;
    Node(int x, int y, int cost, vector<pair<int, int>> path) : x(x), y(y), cost(cost), path(path) {}
};
```

Las variables que componen un nodo son:

- **x e y:** representan las coordenadas del nodo en la matriz.
- **cost:** representa el costo acumulado desde el nodo de inicio (0, 0) hasta este nodo.
- **path:** es un vector de pares de enteros que almacena la ruta tomada para llegar a este nodo.

Cada nodo almacena toda la información necesaria para poder evaluar su viabilidad como parte del camino mínimo y para reconstruir la ruta una vez que se alcanza la solución óptima.

1.2 Lista de nodos vivos

La lista de nodos vivos se implementa utilizando una cola de prioridad (priority_queue) con un criterio de comparación personalizado. La cola de prioridad se usa para siempre seleccionar el nodo con el menor costo acumulado primero, lo que corresponde a una estrategia de búsqueda voraz.
Aquí está la definición de la cola de prioridad y el criterio de comparación:

```
struct CompareCost
{
    bool operator()(Node const &n1, Node const &n2)
    {
        return n1.cost > n2.cost;
    }
};

priority_queue<Node, vector<Node>, CompareCost> liveNodes;
```

La cola de prioridad liveNodes mantiene los nodos vivos ordenados por su costo. El nodo con el costo más bajo se procesa primero.

Otros enfoques que podrían incluirse son búsqueda en profundidad, búsqueda en anchura y A*. El programa implementado usa el criterio basado en el costo acumulado porque es simple y eficiente.

2. MECANISMOS DE PODA

2.1 Poda de nodos no factibles

La poda de un nodo no factible se da durante la expansión de un nodo. Se verifica si las coordenadas de los nodos hijos (nuevas posiciones posibles) están dentro de los límites de la matriz. Si no lo están, se consideran no factibles y se descartan

Este tipo de poda se implementa en el siguiente fragmento de código dentro del bucle que explora los vecinos del nodo actual:

```
if (new_x < rows && new_y < cols && new_x >= 0 && new_y >= 0)
{
    int new_cost = node.cost + matrix[new_x][new_y];
    int new_optimistic_bound = mcp_optimistic(matrix, new_x, new_y, rows, cols, min_val);

    if (new_cost + new_optimistic_bound < solution)
    {
        nexplored++;
        vector<pair<int, int>> new_path = node.path;
        new_path.push_back({new_x, new_y});
        liveNodes.push(Node(new_x, new_y, new_cost, new_path));
    }
    else
    {
        nnot_promising++;
    }
}
else
{
    nunfeasible++;
}
```

En este bloque de código, new_x y new_y representan las coordenadas de los nodos hijos. Si new_x o new_y están fuera de los límites (es decir, new_x < 0, new_y < 0, new_x >= rows, new_y >= cols), el nodo hijo es considerado no factible y se descarta incrementando el contador **nunfeasible**.

2.2 Poda de nodos no prometedores

Un nodo se considera no prometedor si el costo acumulado hasta el nodo más la estimación optimista del costo restante (mcp_optimistic) excede la mejor solución conocida (solution). Este tipo de poda se implementa en el siguiente fragmento de código:

```
if (new_cost + new_optimistic_bound < solution)
{
    nexplored++;
    vector<pair<int, int>> new_path = node.path;
    new_path.push_back({new_x, new_y});
    liveNodes.push(Node(new_x, new_y, new_cost, new_path));
}
else
{
    nnot_promising++;
}
```

En este bloque de código:

- **node.cost:** es el costo acumulado hasta el nodo actual.
- **mcp_optimistic(matrix, node.x, node.y, rows, cols, min_val):** es una función que calcula una estimación optimista del costo restante desde el nodo actual hasta la meta.
- **solution**: es el costo de la mejor solución encontrada hasta el momento.

Si la suma del costo acumulado y la estimación optimista supera el costo de la mejor solución conocida, el nodo actual se descarta como no prometedor y se incrementa el contador **npromising_but_discarded**.

3. COTA PESIMISTAS Y OPTIMISTAS

3.1 Cota pesimista inicial (inicialización).

La cota pesimista inicial es una estimación del costo máximo que puede tener el camino mínimo. Se utiliza para establecer un límite superior inicial para el costo del camino que estamos buscando. La cota pesimista inicial se calcula utilizando una estrategia voraz (greedy) que elige siempre el siguiente paso de menor costo desde el nodo actual.

Aquí está la implementación de la cota pesimista inicial:

```
int mcp_pessimistic(const vector<vector<int>> &matrix, int rows, int cols, int pos_x, int pos_y)
{
    int x = pos_x;
    int y = pos_y;
    int count = matrix[x][y];
    do
    {
        if (x == rows - 1 && y == cols - 1)
        {
            return count;
        }
        int right = INT_MAX;
        int down = INT_MAX;
        int diag = INT_MAX;
        if (y < cols - 1)
        {
            right = matrix[x][y + 1];
        }
        if (x < rows - 1)
        {
            down = matrix[x + 1][y];
        }
        if (x < rows - 1 && y < cols - 1)
        {
            diag = matrix[x + 1][y + 1];
        }
        int minimo = min(right, min(down, diag));
        if (minimo == diag)
        {
            x++;
            y++;
            count += diag;
        }
        else if (minimo == right)
        {
            y++;
            count += right;
        }
        else if (minimo == down)
        {
            x++;
            count += down;
        }
    } while (true);
}
```

Se parte del nodo inicial (0, 0) y se elige siempre el paso de menor costo entre mover a la derecha, hacia abajo o en diagonal. Este enfoque proporciona una cota pesimista inicial para el camino mínimo.

3.2 Cota pesimista del resto de nodos.

Para los nodos que no son el nodo inicial, la cota pesimista es el costo acumulado para llegar a ese nodo. Es decir, no se realiza una estimación adicional, sino que simplemente se considera el costo total acumulado hasta ese punto.

Esta cota se utiliza para comparar y actualizar la mejor solución encontrada. Si al llegar a un nodo hoja (uno que está en la última fila y última columna), el costo acumulado es menor que la mejor solución conocida, se actualiza la mejor solución.

3.3 Cota optimista.

La cota optimista es una estimación del costo mínimo que se necesitaría para alcanzar la meta desde el nodo actual. Esta cota se utiliza para decidir si un nodo es prometedor o no, comparando la suma del costo acumulado y la cota optimista con la mejor solución conocida.

En este código, la cota optimista se calcula considerando el costo mínimo en la matriz (es decir, el menor valor en cualquier celda) y multiplicándose por el número de pasos restantes para alcanzar la meta:

```
int find_min_val(const vector<vector<int>> &matrix, int rows, int cols)
{
    int min_val = INT_MAX;
    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < cols; j++)
        {
            min_val = min(min_val, matrix[i][j]);
        }
    }
    return min_val;
}

int mcp_optimistic(const vector<vector<int>> &matrix, int pos_x, int pos_y, int rows, int cols, int min_val)
{
    int steps_remaining = (rows - 1 - pos_x) + (cols - 1 - pos_y);
    return matrix[pos_x][pos_y] + steps_remaining * min_val;
}
```

En este fragmento de código:

- **pos_x y pos_y:** son las coordenadas del nodo actual.
- **steps_remaining:** calcula el número de pasos necesarios para llegar a la meta desde el nodo actual.

- **min_val:** es el valor mínimo en la matriz, utilizado como una estimación del costo de cada paso restante.

La cota optimista se suma al costo acumulado hasta el nodo actual para decidir si se debe seguir explorando ese nodo. Si la suma del costo acumulado y la cota optimista es mayor que la mejor solución conocida, el nodo se descarta como no prometedor.

4. OTROS MEDIOS EMPLEADOR PARA ACELERAR LA BÚSQUEDA

Para evitar la exploración repetida de nodos ya visitados, se utiliza un conjunto (unordered_set) para almacenar las coordenadas de los nodos que ya han sido procesados. Esto evita ciclos y redundancias, mejorando la eficiencia del algoritmo. Antes de procesar un nodo, se verifica si ya ha sido visitado:

```
string pos = to_string(node.x) + "," + to_string(node.y);
if (visited.find(pos) != visited.end())
{
    continue;
}
visited.insert(pos);
```

5. ESTUDIO COMPARATIVO DE DISTINTAS ESTRATEGIAS DE BÚSQUEDA

NO IMPLEMENTADO

6. TIEMPOS DE EJECUCIÓN

TEST	TIEMPO
060.map	8.878
090.map	22.697
201.map	203.945
301.map	648.398
501.map	2749.759
700.map	7307.517
900.map	15670.652
1K.map	20078.067
2K.map	?
3K.map	?
4K.map	?