MEMORIA PRÁCTICA FINAL

ANÁLISIS Y DISEÑO DE ALGORITMOS

Descripción

En este documento se describen las estructuras de datos empleadas, las estrategias de búsqueda, los mecanismos de poda y las cotas utilizadas.

Noel Martínez Pomares
48771960T

Índice de Contenido

1. Estructura de datos

1.1. Nodo

El nodo se define usando un Struct "Node", que contiene cuatro variables principales y un constructor que las inicializa. Además, se sobrecarga el operador < para ordenar los nodos en la cola de prioridad según la estrategia de búsqueda.

```
// Estructura para almacenar los datos de un nodo
struct Node {
    int cota_optimista, x, y, valor_actual;
    Node(int co, int xi, int yi, int va) : cota_optimista(co), x(xi), y(yi), valor_actual(va) {} // Constructor
    bool operator<(const Node &nodo) const {
        if (cota_optimista != nodo.cota_optimista) // Cota optimista como criterio de ordenación
            return cota_optimista > nodo.cota_optimista;
        if (valor_actual != nodo.valor_actual) // Valor actual como criterio de ordenación
            return valor_actual > nodo.valor_actual;
        return (valor_actual / (x + y + 1)) > (nodo.valor_actual / (nodo.x + nodo.y + 1)); // Coste promedio por casilla como criterio de ordenación
    }
};
```

- cota_optimista: Esta variable guarda el valor de la cota pesimista calculada para cada nodo que realiza una estimación del coste mínimo necesario para alcanzar el objetivo desde el nodo actual.
- x, y: Estas son las coordenadas del nodo en la matriz, que identifican la posición del nodo.
- valor_actual: Representa el coste acumulado hasta el nodo actual desde el inicio.
- operator<: Sobrecarga del operador < para ordenar los nodos en la cola de prioridad. La prioridad se determina primero por la cota_optimista, luego por el valor_actual, y por último por el coste promedio por casilla.

1.2. Lista de nodos vivos

La lista de nodos vivos se gestiona usando una cola de prioridad (priority_queue), pues es adecuada para extraer el nodo más prometedor de manera eficiente. La cola de prioridad se ordena según los criterios definidos en el operador < de la estructura Node.

```
priority_queue<Node> pq;
pq.emplace(calcular_cota_optimista(0, 0, matriz_datos[0][0], n, m), 0, 0, matriz_datos[0][0]);
```

- cota_optimista: Este criterio hace que se exploren primero los nodos con la menor cota_optimista.

- valor_actual: Si los nodos tienen la misma cota_optimista, se selecciona el nodo con el menor coste acumulado.
- coste promedio por casilla: Considera la eficiencia del recorrido en términos del coste por movimiento, escogiendo caminos que mantengan un coste bajo.

2. Mecanismos de poda

2.1. Poda de nodos no factibles

La poda de nodos no factibles se hace con la verificación de las coordenadas del nodo en la matriz y su estado de visitado. Si el nodo tiene coordenadas fuera de los límites de la matriz o ya ha sido visitado, se descarta. Esta poda asegura que no se exploren caminos imposibles de seguir o que ya han sido explorados previamente.

```
if (es_valida(new_x, new_y, n, m) && !visitados[new_x][new_y]) {
    visitados[new_x][new_y] = true;

    int coste_nuevo = valor_actual + matriz_datos[new_x][new_y]; // Coste acumulado hasta la nueva posición

    if (coste_nuevo < mejor_valor) { // Si es mejor que el mejor valor encontrado hasta ahora
        if (coste_nuevo < coste_minimo[new_x][new_y]) { // Si es mejor que el coste mínimo encontrado hasta ahora para esa casilla
            coste_minimo[new_x][new_y] = coste_nuevo;
            pq.emplace(calcular_cota_optimista(new_x, new_y, coste_nuevo, n, m), new_x, new_y, coste_nuevo); // Se añade a la cola de prioridad
            padres[new_x][new_y] = make_pair(x, y); // Se actualiza el padre de la nueva casilla
        }
        else {
            PROM_DESCARTADOS++;
        }
    } else {
        NO_PROMETEDORES++;
    }

    visitados[new_x][new_y] = false;
} else {
    NO_FACTIBLES++;
}
```

- es_valida(new_x, new_y, n, m): Esta función verifica si las coordenadas new_x y new_y están dentro de los límites de la matriz.
- !visitados[new_x][new_y]: Verifica que la casilla en new_x, new_y no haya sido visitada anteriormente, previniendo ciclos y exploraciones redundantes.

2.2. Poda de nodos no prometedores

La poda de nodos no prometedores se basa en dos criterios principales:

- Si el valor_actual es mayor o igual al mejor_valor encontrado hasta el momento, el nodo se descarta, ya que no puede dar una solución mejor.
- Si la cota_optimista es mayor o igual al mejor_valor, el nodo también se descarta, porque no puede mejorar la solución óptima conocida.

```
// Poda si el valor actual es mayor o igual que el mejor valor encontrado hasta ahora
if (valor_actual >= mejor_valor) {
    NO_PROMETEDORES++;
    continue;
}

if (cota_optimista >= mejor_valor) { // Poda por cota optimista
    NO_PROMETEDORES++;
    continue;
}
```

- valor_actual >= mejor_valor: Este criterio asegura que solo se consideren caminos que tienen el potencial de mejorar la solución actual.
- cota_optimista >= mejor_valor: Este criterio utiliza la cota_optimista para descartar caminos que no pueden llevar a una solución mejor.

Además, cuando se evalúa un nodo hijo, si el coste acumulado coste_nuevo es menor que el mejor_valor actual, y mejor que el coste mínimo registrado para esa casilla, se actualiza el coste_minimo y se añade el nodo hijo a la cola de prioridad. Si el coste_nuevo no es mejor que el coste_minimo, se descarta ese nodo hijo específico.

```
if (coste_nuevo < mejor_valor) { // Si es mejor que el mejor valor encontrado hasta ahora
    if (coste_nuevo < coste_minimo[new_x][new_y]) { // Si es mejor que el coste mínimo encontrado hasta ahora para esa casilla
        coste_minimo[new_x][new_y] = coste_nuevo;
        pq.emplace(calcular_cota_optimista(new_x, new_y, coste_nuevo, n, m), new_x, new_y, coste_nuevo); // Se añade a la cola de prioridad
        padres[new_x][new_y] = make_pair(x, y); // Se actualiza el padre de la nueva casilla
    }
    else {
        PROM_DESCARTADOS++;
    }
} else {
    NO_PROMETEDORES++;
}
```

- coste_nuevo < mejor_valor: Verifica que el nuevo coste sea menor que el mejor valor encontrado hasta ahora, asegurando que solo se consideran caminos prometedores.
- coste_nuevo < coste_minimo[new_x][new_y]: Si el nuevo coste es mejor que el coste mínimo registrado para esa casilla, se actualiza el coste mínimo y se añade el nodo hijo a la cola de prioridad.

3. Cotas pesimistas y optimistas

3.1. Cota pesimista inicial (inicialización)

La cota pesimista inicial se establece utilizando numeric_limits<int>::max().

```
int mejor_valor = numeric_limits<int>::max(); // Cota pesimista inicial
```

- numeric_limits<int>::max(): Representa el mayor valor posible para un entero, asegurando que cualquier camino encontrado inicialmente será una mejora.

3.2. Cota pesimista del resto de nodos

NO IMPLEMENTADO

3.3. Cota optimista

La cota optimista se calcula utilizando la distancia Chebyshev desde el nodo actual hasta el objetivo. Esta distancia representa el número mínimo de movimientos necesarios para alcanzar el objetivo, lo cual proporciona una estimación optimista del coste.

```
// Función para calcular la cota optimista
int calcular_cota_optimista(int x, int y, int valor_actual, int n, int m) {
    int distancia_restante = max(n - 1 - x, m - 1 - y);
    return valor_actual + distancia_restante; // Distancia Chebyshev como cota optimista
}
```

- distancia_restante = max(n - 1 - x, m - 1 - y): Calcula la distancia Chebyshev, que es el número máximo de pasos necesarios en una dirección para alcanzar el borde de la matriz.
- valor_actual + distancia_restante: Suma el coste acumulado hasta el nodo actual con la distancia restante.

4. Otros medios empleados para acelerar la búsqueda

Se emplea una matriz coste_minimo para almacenar el coste mínimo encontrado hasta ahora para cada casilla. Esto evita la exploración redundante de caminos que no pueden mejorar el mejor_valor actual.

```
vector<vector<int>> coste_minimo(n, vector<int>(m, numeric_limits<int>::max()));
```

```
if (coste_nuevo < mejor_valor) { // Si es mejor que el mejor valor encontrado hasta ahora
    if (coste_nuevo < coste_minimo[new_x][new_y]) { // Si es mejor que el coste mínimo encontrado hasta ahora para esa casilla
        coste_minimo[new_x][new_y] = coste_nuevo;
        pq.emplace(calcular_cota_optimista(new_x, new_y, coste_nuevo, n, m), new_x, new_y, coste_nuevo); // Se añade a la cola de prioridad
        padres[new_x][new_y] = make_pair(x, y); // Se actualiza el padre de la nueva casilla
    }
    else {
        PROM_DESCARTADOS++;
    }
} else {
    NO_PROMETEDORES++;
}
```

- vector<vector<int>> coste_minimo(n, vector<int>(m, numeric_limits<int>::max())): Inicializa la matriz para almacenar el coste mínimo para cada casilla.
- coste_nuevo < coste_minimo[new_x][new_y]: Comprueba si el nuevo coste es menor que el coste mínimo registrado para esa casilla.
- pq.emplace(...): Añade el nodo a la cola de prioridad si el nuevo coste es mejor, asegurando que solo se exploren los caminos más prometedores.

5. Estudio comparativo de distintas estrategias de búsqueda

```cpp
bool operator<(const Node &nodo) const {
    return cota_optimista > nodo.cota_optimista;
}
```

```
72937
16142887 16142883 1 48261 2 112951918 1 0
2426.061
Nodos Vivos: 16142886
```

Esta estrategia prioriza los nodos con la menor cota optimista, por lo que siempre se escoge el nodo que parece más prometedor en cuanto a la estimación del coste total. Visita y explora más de 16 millones de nodos, lo que da un tiempo de respuesta de 2426.061 ms.

```cpp
bool operator<(const Node &nodo) const {
    return valor_actual > nodo.valor_actual;
}
```

```
72937
16000001 15999999 1 47994 0 111951999 1 0
2333.891
Nodos Vivos: 16000000
```

Esta estrategia prioriza los nodos con el menor coste acumulado hasta el momento (valor actual). Similar a la estrategia de cota optimista, visita y explora 16 millones de nodo, con un tiempo de respuesta de 2333.891 ms. El número de nodos vivos y el tiempo de ejecución son un poco menores que los de la estrategia de cota optimista.

```cpp
bool operator<(const Node &nodo) const {
    return (valor_actual / (x + y + 1)) > (nodo.valor_actual / (nodo.x + nodo.y + 1));
}
```

Esta estrategia prioriza los nodos en función del coste promedio por casilla recorrida. Sin embargo, no se puede resolver el problema con esta estrategia pues puede llevar a la selección de caminos subóptimos que minimizan el coste promedio, pero no tiene por qué llevar a la solución óptima.

```cpp
bool operator<(const Node &nodo) const {
    if (cota_optimista != nodo.cota_optimista)
        return cota_optimista > nodo.cota_optimista;
    if (valor_actual != nodo.valor_actual)
        return valor_actual > nodo.valor_actual;
    return (valor_actual / (x + y + 1)) > (nodo.valor_actual / (nodo.x + nodo.y + 1));
}
```

```
72937
16064771 16064767 1 48090 2 112405277 1 0
2761.418
Nodos Vivos: 16064770
```

Esta estrategia combina varios criterios de priorización: cota optimista, valor actual y coste promedio por casilla. Da como resultado un enfoque más equilibrado, aunque sigue visitando y explorando 16 millones de nodos. El tiempo de respuesta es de 2761.418 ms, que es mayor que el de las estrategias individuales, probablemente por la complejidad de la ordenación de los nodos según múltiples criterios.

6. Tiempos de ejecución

Fichero de test	Tiempo (ms)
060.map	0.516
090.map	0.963
201.map	4.579
301.map	11.923
501.map	32.248
700.map	64.49
900.map	105.946
1K.map	133.585
2K.map	606.191
3K.map	1507.418
4K.map	2872.34