etsinf

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Advent Of Code

2024

Miembros:

Marcos Sellés Solà

Francesc Loeches Nogués

Índice

Reto del Día 1: Advent of Code - Reconciliación de Listas de IDs

Se nos pedía reconciliar dos listas de IDs numéricos representando ubicaciones. El objetivo era calcular la distancia total entre ambas listas tras emparejar sus elementos ordenados de menor a mayor. Esto implicaba leer datos desde un archivo, ordenar las listas y sumar las distancias absolutas entre los elementos emparejados.

Solución

La solución de este problema usa un algoritmo de Divide y Vencerás para ordenar las tablas

```
 8
 9 using namespace std;
10
11 // Function to read file
12 bool readFile(const string& fileName, vector<pair<int, int>>& pairs) {
13     ifstream in{fileName};
14     if (!in) {
15         cerr << "Cannot open file " << fileName << '\n';
16         return false;
17     }
18     string line;
19     while (getline(in, line)) {
20         istringstream iss{line};
21         int i1{}, i2{};
22         if (iss >> i1 >> i2) {
23             pairs.emplace_back(i1, i2);
24         }
25     }
26     return true;
27 }
28
29 int main(int argc, char* argv[]) {
30     if (argc != 2) {
31         cerr << "Usage: " << argv[0] << " <input_file>\n";
32         return EXIT_FAILURE;
33     }
34
35     vector<pair<int, int>> pairs;
36     if (!readFile(argv[1], pairs)) {
37         return EXIT_FAILURE;
38     }
39
40     vector<int> c1; //vector Columna 1
41     vector<int> c2; // Vector columna 2
42     for (const auto& [l, r] : pairs) {
43         c1.push_back(l);
44         c2.push_back(r);
45     }
46
47     sort(c1.begin(), c1.end());
48     sort(c2.begin(), c2.end());
49 S
50     int DistanciaTotal = 0;
51     for (size_t i = 0; i < c1.size(); ++i) {
52         DistanciaTotal += abs(c1[i] - c2[i]);
53     }
54     cout << "Distancia Total: " << DistanciaTotal << '\n';
55
56     return EXIT_SUCCESS;
57 }
```

1. **Lectura del archivo:** Se utiliza la función readFile para leer un archivo que contiene pares de números separados por espacios y almacenarlos en un vector de pares.
2. **Separación en dos columnas:** Los valores se dividen en dos vectores (c1 y c2) que representan las dos listas.
3. **Ordenación:** Ambos vectores se ordenan usando std::sort.
4. **Cálculo de distancias:** Se recorre cada lista y se calcula la distancia absoluta entre los elementos emparejados usando std::abs.
5. **Suma de distancias:** Las distancias calculadas se acumulan en la variable DistanciaTotal.
6. **Impresión del resultado:** Finalmente, se imprime la distancia total calculada.

Reto del Día 4: Advent of Code - Búsqueda de Palabra "XMAS"

El día de hoy, se nos pedía encontrar todas las apariciones posibles de la palabra "XMAS" dentro de una matriz de letras, considerando todas las direcciones posibles: horizontal, vertical, diagonal y al revés. Esto implicaba recorrer la matriz y verificar las coincidencias en las 8 direcciones desde cada posición.

Solución

Para resolver este problema implementamos un código que emplea grafos en C++ para la búsqueda de la palabra

```cpp
using namespace std;

// Dirección de exploración: [vertical, horizontal, diagonal]
int dirs[8][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}, {-1, -1}, {1, 1}, {-1, 1}, {1, -1}};

bool isValid(int x, int y, int rows, int cols) {
    return x >= 0 && y >= 0 && x < rows && y < cols;
}

int countXMAS(const vector<string>& board, int rows, int cols) {
    int count = 0;
    string target = "XMAS";

    // Iterar por cada celda en el tablero
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            // Si encontramos una 'X', intentamos encontrar la palabra "XMAS" en las 8 direcciones
            if (board[i][j] == 'X') {
                // Probar todas las direcciones
                for (int d = 0; d < 8; ++d) {
                    int nx = i, ny = j;
                    bool match = true;

                    // Verificar si las siguientes letras forman "XMAS"
                    for (int k = 0; k < target.length(); ++k) {
                        if (!isValid(nx, ny, rows, cols) || board[nx][ny] != target[k]) {
                            match = false;
                            break;
                        }
                        // Mover en la dirección
                        nx += dirs[d][0];
                        ny += dirs[d][1];
                    }

                    if (match) {
                        ++count;
                    }
                }
            }
        }
    }

    return count;
}

int main() {
    string filename = "PuzzleInput";  // Nombre del archivo de entrada
    ifstream inputFile(filename);

    if (!inputFile) {
        cerr << "Error al abrir el archivo." << endl;
        return 1;
    }

    vector<string> board;
    string line;
     // Leer las líneas del archivo y almacenarlas en el vector board
     while (getline(inputFile, line)) {
         board.push_back(line);
     }

     inputFile.close();

     int rows = board.size();
     int cols = board.empty() ? 0 : board[0].size();

     // Contar las ocurrencias de "XMAS"
     int result = countXMAS(board, rows, cols);

     // Mostrar el resultado
     cout << "La palabra 'XMAS' aparece " << result << " veces." << endl;
```

1. **Lectura del archivo:** Se carga la matriz de letras desde un archivo de texto.
2. **Búsqueda multidireccional:** Desde cada celda que contiene una 'X', se verifican las coincidencias en las 8 direcciones usando un arreglo de desplazamientos (dirs).
3. **Validación de límites:** Se asegura que las posiciones exploradas estén dentro de los límites de la matriz.
4. **Conteo de coincidencias:** Cada vez que se encuentra "XMAS" en cualquier dirección, se incrementa el contador.
5. **Uso de grafos:** El tablero puede interpretarse como un grafo, donde cada celda es un nodo y las conexiones corresponden a las 8 direcciones posibles. Este enfoque asegura que se exploren todas las rutas posibles para formar la palabra objetivo.
6. **Impresión del resultado:** Finalmente, se imprime la cantidad total de coincidencias encontradas.

Reto del Día 5: Advent of Code - Cola de Impresión

En este reto, se nos pedía verificar si las actualizaciones de un manual de seguridad están en el orden correcto según un conjunto de reglas de precedencia entre las páginas. Además, debíamos sumar el número de la página central de cada actualización que esté correctamente ordenada.

Solución

El siguiente código implementa una solución utilizando tablas hash para gestionar las reglas y validar las actualizaciones:

```cpp
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <unordered_map>
#include <unordered_set>
#include <string>
#include <algorithm>

using namespace std;

unordered_map<int, vector<int>> leer_reglas(ifstream &file) {
    unordered_map<int, vector<int>> precedencia;
    string line;
    while (getline(file, line) && !line.empty()) {
        int x, y;
        char delimiter;
        stringstream ss(line);
        ss >> x >> delimiter >> y;
        precedencia[x].push_back(y);
    }
    return precedencia;
}

vector<vector<int>> leer_actualizaciones(ifstream &file) {
    vector<vector<int>> actualizaciones;
    string line;
    while (getline(file, line)) {
        vector<int> update;
        stringstream ss(line);
        int num;
        while (ss >> num) {
            update.push_back(num);
            if (ss.peek() == ',') ss.ignore();
        }
        actualizaciones.push_back(update);
    }
    return actualizaciones;
}

bool es_orden_correcto(const vector<int> &update, const unordered_map<int, vector<int>> &precedencia) {
    unordered_set<int> seen;
    for (size_t i = 0; i < update.size(); ++i) {
        seen.insert(update[i]);
        if (precedencia.find(update[i]) != precedencia.end()) {
            for (int siguiente : precedencia.at(update[i])) {
                if (seen.find(siguiente) != seen.end()) {
                    return false;
                }
            }
        }
    }
    return true;
}

int suma_numeros_centrales(const vector<vector<int>> &actualizaciones, const unordered_map<int, vector<int>> &precedencia) {
    int suma = 0;
    for (const auto &update : actualizaciones) {
        if (es_orden_correcto(update, precedencia)) {
            suma += update[update.size() / 2];
        }
    }
    return suma;
}
```

1. **Lectura de reglas:** Las reglas de precedencia se almacenan en un unordered_map donde la clave es un número de página y el valor es una lista de páginas que deben imprimirse después.
2. **Lectura de actualizaciones:** Las actualizaciones se cargan como vectores de enteros que representan las páginas en cada actualización.
3. **Validación del orden:** Para cada actualización, se verifica que el orden cumpla las reglas usando un unordered_set para rastrear las páginas ya vistas.
4. **Suma de números centrales:** Si una actualización es válida, se suma el número de su página central.
5. **Impresión del resultado:** Se imprime la suma total de los números centrales de las actualizaciones válidas.

```
66 int main() {
67     string filename = "Puzzle5";
68     ifstream file(filename);
69 
70     if (!file.is_open()) {
71         cerr << "No se pudo abrir el archivo: " << filename << endl;
72         return 1;
73     }
74 
75     auto precedencia = leer_reglas(file);
76     auto actualizaciones = leer_actualizaciones(file);
77 
78     int resultado = suma_numeros_centrales(actualizaciones, precedencia);
79     cout << "La suma de los números de página centrales de las actualizaciones correctamente ordenadas es: " << resultado <<
   endl;
80 
81     return 0;
82 }
83 
```

Reto del Día 7: Advent of Code - Calibración del Puente

Según el enunciado debíamos determinar qué ecuaciones de calibración pueden ser ciertas al insertar operadores (+ y *) entre los números dados, evaluándose de izquierda a derecha. Además, debíamos calcular la suma de los valores de prueba de las ecuaciones que resultaron válidas.

Solución

Utilizando combinaciones de operadores y haciendo uso de memoria dinámica para optimizar el procesamiento:

```cpp
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <string>

uint64_t applyOperation(uint64_t left, uint64_t right, char op) {
    return (op == '+') ? left + right : left * right;
}

bool evaluateExpression(const std::vector<uint64_t>& numbers, const std::vector<char>& operations, uint64_t target) {
    uint64_t result = numbers[0];
    for (size_t i = 0; i < operations.size(); ++i) {
        result = applyOperation(result, numbers[i + 1], operations[i]);
    }
    return result == target;
}

bool checkEquation(const std::vector<uint64_t>& numbers, uint64_t target) {
    size_t n = numbers.size();
    if (n == 1) {
        return numbers[0] == target;  // Caso base, solo un número
    }

    // Comprobamos todas las combinaciones posibles de operadores
    size_t numOps = n - 1;
    std::vector<char> ops(numOps);
    std::vector<char> possibleOps = {'+', '*'};

    // Generamos todas las combinaciones posibles de los operadores
    for (size_t i = 0; i < (1 << numOps); ++i) {
        for (size_t j = 0; j < numOps; ++j) {
            ops[j] = possibleOps[(i >> j) & 1];  // Asignamos '+' o '*' dependiendo de la máscara
        }

```

1. **Generación de combinaciones:** Se generan todas las combinaciones posibles de los operadores + y * entre los números utilizando un enfoque basado en memoria dinámica para optimizar la generación y evaluación de estas combinaciones.

```cpp
        // Evaluamos la expresión con esta combinación de operadores
        if (evaluateExpression(numbers, ops, target)) {
            return true;
        }
    }

    return false;
}

bool readFile(const std::string& fileName, std::vector<std::string>& lines) {
    std::ifstream in(fileName);
    if (!in) {
        std::cerr << "Cannot open file " << fileName << '\n';
        return false;
    }

    std::string line;
    while (std::getline(in, line)) {
        lines.push_back(line);
    }
    return true;
}
```

2. **Evaluación de expresiones:** Cada combinación de operadores se evalúa para comprobar si produce el valor objetivo.

```
    std::vector<std::string> lines;
    if (!readFile(argv[1], lines)) {
        return EXIT_FAILURE;
    }

    uint64_t totalSum = 0;

    for (const auto& line : lines) {
        std::istringstream iss(line);
        uint64_t result;
        iss >> result;  // Leer el valor de prueba
        iss.ignore(1);  // Ignorar el ':'

        std::vector<uint64_t> ops;
        uint64_t num;
        while (iss >> num) {
            ops.push_back(num);
        }

        // Verificar si la combinación de operadores puede producir el valor de prueba
        if (checkEquation(ops, result)) {
            totalSum += result;
        }
    }

    std::cout << "Total calibration sum: " << totalSum << '\n';

    return EXIT_SUCCESS;
}

int main(int argc, char* argv[]) {
    if (argc != 2) {
        std::cerr << "Usage: " << argv[0] << " <input_file>" << std::endl;
        return EXIT_FAILURE;
    }

```

3. **Suma de resultados:** Si una ecuación es válida, se suma su valor objetivo al total.
4. **Entrada y salida:** Los datos se leen desde un archivo de texto, procesando cada línea como una ecuación separada.

Reto del Día 19: Advent of Code - Diseño de Toallas

En el último reto que planteamos, el enunciado nos pedía verificar cuántos diseños de toallas son posibles utilizando patrones disponibles. Cada diseño se debía construir concatenando uno o más patrones exactamente en el orden especificado, sin invertirlos ni alterarlos.

Solución

Implementamos una solución utilizando un árbol binario para almacenar los patrones disponibles y una estrategia de programación dinámica para verificar los diseños:

```cpp
#include <fstream>
#include <iostream>
#include <map>
#include <sstream>
#include <string>
#include <vector>

// Definición del nodo del árbol binario
struct TreeNode {
    std::string word;
    TreeNode* left;
    TreeNode* right;

    TreeNode(const std::string& w) : word(w), left(nullptr), right(nullptr) {}
};

// Insertar una palabra en el árbol binario
TreeNode* insert(TreeNode* root, const std::string& word) {
    if (root == nullptr) {
        return new TreeNode(word);
    }
    if (word < root->word) {
        root->left = insert(root->left, word);
    } else if (word > root->word) {
        root->right = insert(root->right, word);
    }
    return root;
}

// Buscar una palabra en el árbol binario
bool search(TreeNode* root, const std::string& word) {
    if (root == nullptr) {
        return false;
    }
    if (root->word == word) {
        return true;
    }
    if (word < root->word) {
        return search(root->left, word);
    } else {
        return search(root->right, word);
    }
}

bool readFile(const std::string& fileName, std::vector<std::string>& lines)
{
    std::ifstream in{fileName};
    if (!in) {
        std::cerr << "Cannot open file " << fileName << '\n';
        return false;
    }
    std::string str;
    while (std::getline(in, str)) {
        lines.push_back(str);
    }
    return true;
}

```

1. **Estructura de datos:** Se utiliza un árbol binario para almacenar los patrones disponibles, permitiendo búsquedas rápidas.

```cpp
bool canBuild(TreeNode* root, const std::string& word, std::map<std::string, bool>& cache)
{
    if (auto it = cache.find(word); it != cache.end()) {
        return it->second;
    }
    if (search(root, word)) {
        cache[word] = true;
        return true;
    }
    bool res = false;
    for (size_t i = 1; i < word.size(); ++i) {
        const auto left = word.substr(0, i);
        if (!canBuild(root, left, cache)) {
            continue;
        }
        const auto right = word.substr(i, word.size() - i);
        if (canBuild(root, right, cache)) {
            res = true;
            break;
        }
    }
    cache[word] = res;
    return res;
}

int main(int argc, char* argv[])
{
    if (argc != 2) {
        return EXIT_FAILURE;
    }
    std::vector<std::string> lines{};
    if (!readFile(argv[1], lines)) {
        return EXIT_FAILURE;
    }

    TreeNode* root = nullptr;
    {
        std::istringstream iss{lines[0]};
        std::string p;
        while (iss >> p) {
            if (p[p.size() - 1] == ',') {
                p = p.substr(0, p.size() - 1);
            }
            if (p.empty()) {
                continue;
            }
            root = insert(root, p);
        }
    }

    std::vector<size_t> goods;

    {  // Parte 1
        int count = 0;
        std::map<std::string, bool> cache;
        for (size_t i = 2; i < lines.size(); ++i) {
            if (canBuild(root, lines[i], cache)) {
                goods.push_back(i);
                ++count;
            }
        }
        std::cout << count << '\n';
    }

    return EXIT_SUCCESS;
}
```

2. **Verificación de diseños:** La función `canBuild` utiliza programación dinámica para determinar si un diseño puede construirse concatenando patrones.

3. **Entrada y salida:** Los patrones disponibles se leen desde la primera línea del archivo, y los diseños deseados se procesan desde las líneas posteriores.

4. **Cómputo del resultado:** Se cuenta el número total de diseños que pueden construirse correctamente.