Práctica 3. Divide y vencerás

Alejandro Guitarte Fernández
alejandro.guifer@alum.uca.es
Teléfono: 601048359
NIF: 32092444S

15 de diciembre de 2022

1. Describa las estructuras de datos utilizados en cada caso para la representación del terreno de batalla.

   En mi caso he usado la misma estructura para todos los casos, llamada tipoCelda. Esta contiene 2 campos de números enteros (fila y columna) y un flotante (valor de la celda). Tiene constructor por defecto y uno que recibe 3 parámetros, uno para cada atributo. Por último, se define el operador de compración $<$ como la comparación $<$ entre los campos *valor* de ambos objetos, para poder usar montículos.

2. Implemente su propia versión del algoritmo de ordenación por fusión. Muestre a continuación el código fuente relevante.

```
void fusion(std::vector<tipoCelda>& L, int i, int k, int j){
    int n = j-i+1, p = i, q = k+1;
    std::vector<tipoCelda> w;
    w.reserve(L.size());
    for(int l = 0; l < n; l++){
        if(p <= k && (q > j || L[p].value <= L[q].value)){
            w[l] = L[p];
            p++;
        }
        else{
            w[l] = L[q];
            q++;
        }
    }
    for(int l = 0; l < n; l++)
        L[i + l] = w[l];
}

void OrdenaFusion(std::vector<tipoCelda>& L, int i, int j){
    int n = j-i + 1;
    if(n <= 3){
        //ORDENACION POR INSERCION
        for(int k = i; k <= j; k++){
            int pos = k;
            tipoCelda aux = L[k];
            while(pos > 0 && L[pos-1].value > aux.value){
                L[pos] = L[pos-1];
                pos--;
            }
            L[pos] = aux;
        }
    }
    else{
        int k = i + n/2;
        OrdenaFusion(L, i, k);
        OrdenaFusion(L, k+1, j);
        fusion(L, i, k, j);
    }
}
```

3. Implemente su propia versión del algoritmo de ordenación rápida. Muestre a continuación el código fuente relevante.

```
int pivote(std::vector<tipoCelda>& L, int i, int j){
    int p = i;
    tipoCelda x = L[i];
    for(int k = i+1; k < j; k++){
        if(L[k].value <= x.value){
            p++;
            tipoCelda aux = L[p];
            L[p] = L[k];
            L[k] = aux;
        }
    }
    L[i] = L[p];
    L[p] = x;

    return p;
}

void OrdenaRapida(std::vector<tipoCelda>& L, int i, int j){
    int n = j-i + 1;
    if(n <= 3){
        //ORDENACION POR INSERCION
        for(int k = i; k <= j; k++){
            int pos = k;
            tipoCelda aux = L[k];
            while(pos > 0 && L[pos-1].value > aux.value){
                L[pos] = L[pos-1];
                pos--;
            }
            L[pos] = aux;
        }
    }
    else{
        int p = pivote(L, i, j);
        OrdenaRapida(L, i, p-1);
        OrdenaRapida(L, p+1, j);
    }
}
```

4. Realice pruebas de caja negra para asegurar el correcto funcionamiento de los algoritmos de ordenación implementados en los ejercicios anteriores. Detalle a continuación el código relevante.

```
int main(){
    std::vector<tipoCelda> Vf, Vr;

    //Llenamos los vectores con numeros aleatorios (Los mismos)
    for(int i = 0; i < 100; i++){
        int r = rand();
        tipoCelda c(Vector3(), 0, 0, r);
        Vf.push_back(c);
        Vr.push_back(c);
    }
    OrdenaFusion(Vf, 0, 99);
    OrdenaRapida(Vr, 0, 99);

    std::cout << "-----------------------------------------PRUEBAS DE CAJA NEGRA
        -----------------------------------------" << std::endl;
    std::cout << "FUSION - 100 elementos aleatorios: ";
    bool success = true;
    for(int i = 1; i < 100 && success; i++){
        if(Vf[i].value < Vf[i-1].value)
            success = false;
    }
    std::cout << (success ? "PASS" : "FAIL") << std::endl;

    std::cout << "RAPIDA - 100 elementos aleatorios: ";
    success = true;
    for(int i = 1; i < 100 && success; i++){
```

```
        if(Vr[i].value < Vr[i-1].value)
            success = false;
    }
    std::cout << (success ? "PASS" : "FAIL") << std::endl;
    std::cout << "
        ----------------------------------------------------------------------------------------------------
        " << std::endl;

    return 0;
}
```

5. Analice de forma teórica la complejidad de las diferentes versiones del algoritmo de colocación de defensas en función de la estructura de representación del terreno de batalla elegida. Comente a continuación los resultados. Suponga un terreno de batalla cuadrado en todos los casos.

   Para el caso primero, en el que no ordenamos las defensas, obtendremos que el $t(n_{defs}, n_{cells}) \in \mathcal{O}(n_{defs} \cdot n_{cells}^2)$, ya que en el bucle más externo recorremos todas las defensas, para colocarlas, y dentro de éste calculamos la lista (n_{cells} para recorrer todas las celdas y n_{cells} de nuevo en el peor caso para insertarla en la lista de manera ordenada. Esto es porque los elemenos se insertan de manera ordenada crecientemente, de manera que la función de selección simplemente elegirá el último elemento.

   En el caso de la ordenación por fusión, el resultado será $t(n_{defs}, n_{cells}) \in \mathcal{O}(n_{defs} \cdot n_{cells} \cdot \log n_{cells})$. Se realiza el algorito de ordenación por fusión (complejidad $n \cdot \log n$ en el peor caso, y esto n_{defs} veces.

   En el caso de la ordenación rápida, el resultado será $t(n_{defs}, n_{cells}) \in \mathcal{O}(n_{defs} \cdot n_{cells} \cdot \log n_{cells})$. Se realiza el algorito de ordenación rápida (complejidad $n \cdot \log n$ en el caso promedio, y esto n_{defs} veces.

6. Incluya a continuación una gráfica con los resultados obtenidos. Utilice un esquema indirecto de medida (considere un error absoluto de valor 0.01 y un error relativo de valor 0.001). Es recomendable que diseñe y utilice su propio código para la medición de tiempos en lugar de usar la opción *-time-placeDefenses3* del simulador. Considere en su análisis los planetas con códigos 1500, 2500, 3500,..., 10500, al menos. Puede incluir en su análisis otros planetas que considere oportunos para justificar los resultados. Muestre a continuación el código relevante utilizado para la toma de tiempos y la realización de la gráfica.

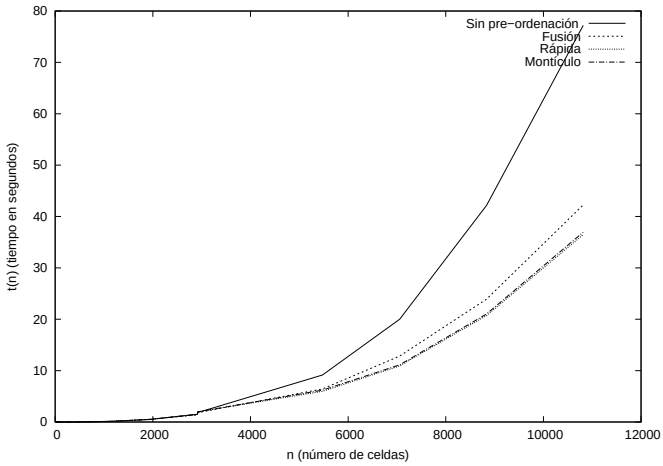

Figura 1: Tiempos de ejecución para cada estrategia

Vemos como la peor es claramente la que no tiene pre-ordenación, y después la de fusión. Después, aunque muy similares, el montículo parece ser algo mejor que la ordenación rápida. Para la medición de tiempos se ha usado el siguiente código:

```
void DEF_LIB_EXPORTED placeDefenses3(bool** freeCells, int nCellsWidth, int nCellsHeight,
    float mapWidth, float mapHeight
              , List<Object*> obstacles, List<Defense*> defenses) {

    float cellWidth = mapWidth / nCellsWidth;
    float cellHeight = mapHeight / nCellsHeight;
```

```
        cronometro c;
    long int r = 0;
    double e_abs = 0.01, e_rel = 0.001;

    c.activar();
    do {
                algoritmoVorazSinOrdenar(nCellsWidth, nCellsHeight, freeCells, mapWidth,
                    mapHeight, obstacles, defenses);

                ++r;
    } while(c.tiempo() < e_abs/e_rel + e_abs);
    c.parar();
    double t1 = c.tiempo() / r;

    r = 0;
    c.activar();
    do {
            algoritmoVorazFusion(nCellsWidth, nCellsHeight, freeCells, mapWidth, mapHeight,
                obstacles, defenses);

                ++r;
    } while(c.tiempo() < e_abs/e_rel + e_abs);
    c.parar();

    double t2 = c.tiempo() / r;

    r = 0;
    c.activar();
    do {
                algoritmoVorazRapida(nCellsWidth, nCellsHeight, freeCells, mapWidth,
                    mapHeight, obstacles, defenses);

                ++r;
    } while(c.tiempo() < e_abs/e_rel + e_abs);
    c.parar();

    double t3 = c.tiempo() / r;

    r = 0;
    c.activar();
    do {
                algoritmoVorazMont(nCellsWidth, nCellsHeight, freeCells, mapWidth, mapHeight,
                     obstacles, defenses);

                ++r;
    } while(c.tiempo() < e_abs/e_rel + e_abs);
    c.parar();

    double t4 = c.tiempo() / r;

    std::cout << (nCellsWidth * nCellsHeight) << '\t' << t1 << '\t' << t2 << '\t' << t3 << '\
        t' << t4 << std::endl;
}
```

Esta salida es la que necesita el comando *make data*, que es el que he usado, y a continuación *make plot* para generar la gráfica. Posteriormente se convirtión a pdf para poder integrarla en la memoria.

Todo el material incluido en esta memoria y en los ficheros asociados es de mi autoría o ha sido facilitado por los profesores de la asignatura. Haciendo entrega de este documento confirmo que he leído la normativa de la asignatura, incluido el punto que respecta al uso de material no original.