Práctica 3. Divide y vencerás

Jose Manuel Barba Gonzalez
josemanuel.barbagonzalez@alum.uca.es
Teléfono: xxxxxxxx
NIF: 48899329H

19 de diciembre de 2021

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

   La estructura para englobar los datos a ordenar y para seleccionar la posición de la lista de defensas es la clase "ValueList" con los atributos "value" y "position", dicha clase se usa para construir un vector de la biblioteca standard "std::vector<ValueList>" en cada función de ordenación y ordenar el vector previamente a seleccionar su posición en el mapa.

   Se sobrecargan los operadores "<" ">" "<=" ">=" para las comparaciones en los algoritmos de ordenación.

```
class ValueList
{
        public:
                ValueList(float vl=0, Vector3 ps=0): value(vl), position(ps){};
                float value;
                Vector3 position;
};
bool operator <(const ValueList &A, const ValueList &B){ return A.value < B.value; }
bool operator >(const ValueList &A, const ValueList &B){ return A.value > B.value; }
bool operator <=(const ValueList &A, const ValueList &B){ return A.value <= B.value; }
bool operator >=(const ValueList &A, const ValueList &B){ return A.value >= B.value; }
```

   A continuación se muestra la función que crea la matriz que da valores a las celdas del mapa mediante la funcion "defaultCellValue()", para luego ser introducidas en el vector "std::vector<ValueList>" y ser ordenadas en su caso.

```
/**
* Funcion que crea una matriz de valores correspondientes a cada celda del mapa dados por la
    funcion cellValue, siendo
* la funcion Candidatos del algoritmo Voraz
* @param freeCells matriz de numero de celdas-ancho por numero de celdas-alto, contiene true
    si el centro de la celda
*                   esta libre y false si el centro de la celda esta ocupado por un obstaculo.
* @param nCellsWidth numero de celdas en anchura.
* @param nCellsHeight numero de celdas en altura.
* @param mapWidth ancho total del mapa preestablecido.
* @param mapHeight alto total del mapa preestablecido.
* @param obstacles Lista actual de Obstaculos creados y situados.
* @param defenses Lista actual de Defensas creadas, no situadas.
* @return Devuelve una matriz rellena con los valores de las posiciones candidatas
    prometedoras.
*/
float** matrixValues(bool** freeCells, int nCellsWidth, int nCellsHeight
, float mapWidth, float mapHeight, const List<Object*> &obstacles, const List<Defense*> &
    defenses)
{
        float **matrixOfValues; matrixOfValues = new float* [nCellsWidth];
        float MAXV = 100.0f;
        float MEDV = MAXV * 0.9f;

        for (int i = 0; i < nCellsWidth; ++i)
                matrixOfValues[i] = new float [nCellsHeight];
```

```
        for(int iW = nCellsWidth; iW < nCellsWidth; iW++)
                for(int iH = nCellsHeight; iH < nCellsHeight; iH++)
                        matrixOfValues[iW][iH] = defaultCellValue(iW, iH, freeCells,
                            nCellsWidth, nCellsHeight, mapWidth,
                        mapHeight, obstacles, defenses);

return matrixOfValues;
}
```

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

   Método de ordenación por Fusión descrito en la teoría por pseudocódigo y transcrita a código C++.

```
/**
 * Funcion secundaria de Ordenacion por Fusion que realiza los intercambios indicodos por los
      parametros de entrada.
 * @param v Variable que tiene el estado actual del vector pasado por referencia.
 * @param i Variable que recibe el inicio de tramo a evaluar.
 * @param k Variable que recibe el tramo de posiciones a evaluar.
 * @param j Variable que recibe el final de tramo a evaluar.
 */
void Fusion(std::vector<ValueList>& v, size_t i, size_t k, size_t j)
{
        size_t n = j - i + 1;
        size_t p = i; size_t q = k + 1;
        std::vector<ValueList> w;

        for(int it = 0; it < n; ++it)
        {
                if (p <= k and (q > j or v[p].value <= v[q].value))
                {
                        w.push_back(v[p]);
                        p++;
                }
                else
                {
                        w.push_back(v[q]);
                        q++;
                }
        }

        for (int it = 0; it < n; ++it)
                v[i + it] = w[it];
}

/**
 * Funcion que ordena por el metodo "Fusion", en este caso, un vector de la clase "ValueList"
      por su atributo "value".
 * @param orderCells Variable del vector inicial pasado por referencia.
 * @param pos_i Variable con la posicion inicial del vector.
 * @param pos_j Variable con la posicion final del vector.
 */
void orderFusion(std::vector<ValueList>& orderCells, size_t pos_i, size_t pos_j)
{
        size_t n = pos_j - pos_i + 1;
        size_t n0 = 3, pos_k;

        if (n <= n0)
                std::sort(orderCells.begin() + pos_i, orderCells.begin() + pos_j, std::
                    greater<ValueList>());
        else
        {
                pos_k = pos_i - 1 + n / 2;
                orderFusion(orderCells, pos_i, pos_k);
                orderFusion(orderCells, pos_k + 1, pos_j);
                Fusion(orderCells, pos_i, pos_k, pos_j);
        }
}
```

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

   Método de ordenación por QuickSort (método propio de Rápida) transcrita a código C++.

```
/**
* Funcion que ordena por el metodo "QuickSort", en este caso, un vector de la clase "
    ValueList" por su atributo "value".
* @param orderCells Variable del vector inicial pasado por referencia.
* @param izq Variable con la posicion inicial del vector.
* @param dch Variable con la posicion final del vector.
*/
void quickSort(std::vector<ValueList>& orderCells, int izq, int dch)
{
        float piv;
        int i=izq, d=dch;

        if (izq >= dch) return;
        piv = orderCells[(izq + dch) / 2].value;

        while (i < d)
        {
                for (; orderCells[i].value < piv; ++i);
                for (; orderCells[d].value > piv; --d);
                if (i <= d)
                {
                        std::swap(orderCells[i], orderCells[d]);
                        ++i;
                        --d;
                }
        }

        quickSort(orderCells, izq, d);
        quickSort(orderCells, i, dch);
}
```

   Método de ordenación Rápida descrito en la teoría por pseudocódigo y transcrita a código C++.

```
/**
* Funcion que dado un pivote ordena las posiciones adyacentes.
* @param orderCells Variable que tiene el estado actual del vector pasado por referencia.
* @param pos_i Variable que recibe el inicio de tramo a evaluar.
* @param pos_j Variable que recibe el final de tramo a evaluar.
* @return Devuelve el pivote actual.
*/
int pivote(std::vector<ValueList>& orderCells, int pos_i, int pos_j)
{
        int p = pos_i;
        ValueList x = orderCells[pos_i], aux;

        for (int k = pos_i + 1; k < pos_j; ++k)
        {
                if (orderCells[k].value <= x.value)
                {
                        p++;
                        aux = orderCells[p];
                        orderCells[p] = orderCells[k];
                        orderCells[k] = aux;
                        //std::swap(orderCells[p], orderCells[k]);
                }
                orderCells[pos_i] = orderCells[p];
                orderCells[p] = x;
        }
        return p;
}

/**
* Funcion que ordena por el metodo "Rapida", en este caso, un vector de la clase "ValueList"
    por su atributo "value".
* @param orderCells Variable del vector inicial pasado por referencia.
* @param pos_i Variable con la posicion inicial del vector.
```

```
* @param pos_j Variable con la posicion final del vector.
*/
void orderRapida(std::vector<ValueList>& orderCells, int pos_i, int pos_j)
{
        int n = pos_j - pos_i + 1;
        int n0 = 3;

        if(n <= n0)
                std::sort(orderCells.begin() + pos_i, orderCells.begin() + pos_j + 1, std::
                    greater<ValueList>());
        else
        {
                int p = pivote(orderCells, pos_i, pos_j);
                orderRapida(orderCells, pos_i, p - 1);
                orderRapida(orderCells, p + 1, pos_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.

   Para las pruebas de caja negra se ha usado el método estándar, se realizan cuatro bucles, uno por cada llamada a las diferentes ordenaciones, y se tomas muestras hasta los parámetros indicados en $e_abs/(e_rel + e_abs)$.

```
cronometro cNOrdenado;
long int rNOrdenado = 0;
cNOrdenado.activar();
do
{
        //codigo de placedefenses de la p1 sin ordenar, o llamar a la funcion
        placeDefensesNoOrdenacion(freeCells, nCellsWidth, nCellsHeight, mapWidth, mapHeight,
            obstacles, defenses);
        ++rNOrdenado;
} while(cNOrdenado.tiempo() < e_abs / (e_rel + e_abs));
cNOrdenado.parar();

cronometro cFusion;
long int rFusion = 0;
cFusion.activar();
do
{
        //codigo de placedefenses de la p1 sin ordenar, o llamar a la funcion
        placeDefensesFusion(freeCells, nCellsWidth, nCellsHeight, mapWidth, mapHeight,
            obstacles, defenses);
        ++rFusion;
} while(cFusion.tiempo() < e_abs / (e_rel + e_abs));
cFusion.parar();

cronometro cRapida;
long int rRapida = 0;
cRapida.activar();
do
{
        //codigo de placedefenses de la p1 sin ordenar, o llamar a la funcion
        placeDefensesRapida(freeCells, nCellsWidth, nCellsHeight, mapWidth, mapHeight,
            obstacles, defenses);
        ++rRapida;
} while(cRapida.tiempo() < e_abs / (e_rel + e_abs));
cRapida.parar();

cronometro cMonticulo;
long int rMonticulo = 0;
cMonticulo.activar();
do
{
        //codigo de placedefenses de la p1 sin ordenar, o llamar a la funcion
        placeDefensesMonticulo(freeCells, nCellsWidth, nCellsHeight, mapWidth, mapHeight,
            obstacles, defenses);
        ++rMonticulo;
} while(cMonticulo.tiempo() < e_abs / (e_rel + e_abs));
cMonticulo.parar();
```

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.

   Teniendo en cuenta que:

   a) N es el numero de celdas del terreno de batalla.

   b) O el número de obstáculos del terreno de batalla.

   c) D el numero de defensas a colocar en el terreno de batalla.

   Sin Ordenación

   Complejidad Espacial Vector de STL std::vector<ValueList> $\Theta(N)$

   Complejidad Valoración y Selección $\Theta(DN^2) + \Theta(O)$

   Complejidad Temporal $\Theta(NO) + \Theta(DN^2)$

   Ordenación Fusión

   Complejidad Espacial Vector de STL std::vector<ValueList> $\Theta(N)$

   Complejidad Valoración y Selección $\Theta(DN^2) + \Theta(O)$

   Complejidad Temporal $\Theta(NO) + \Theta(logN) + \Theta(DN)$

   Ordenación Rápida

   Complejidad Espacial Vector de STL std::vector<ValueList> $\Theta(N)$

   Complejidad Valoración y Selección $\Theta(DN^2) + \Theta(O)$

   Complejidad Temporal $\Theta(NO) + \Theta(NlogN) + \Theta(DN)$

   Ordenación Montículo

   Complejidad Espacial Vector de STL std::vector<ValueList> $\Theta(N)$

   Complejidad Valoración y Selección $\Theta(DC^2) + \Theta(P)$

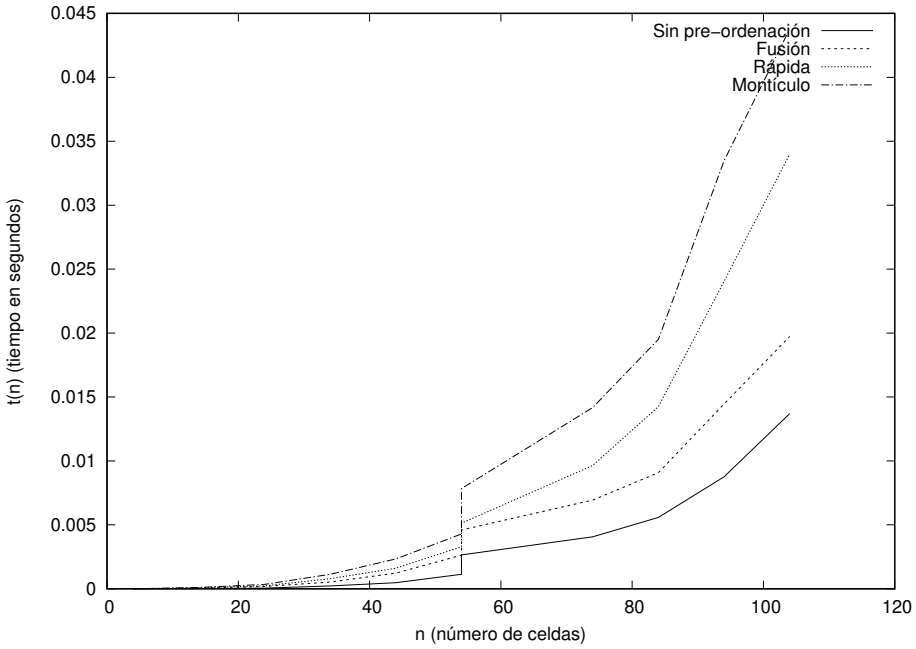

Figura 1: Tiempos de Ordenación

Complejidad Temporal $\Theta(NO) + \Theta(NlogN) + \Theta(DN)$

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.

   Gráfica de resultados de las pruebas realizadas recogidas en "data.txt".

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.