Práctica 3. Divide y vencerás

Nombre Apellido1 Apellido2
correo@servidor.com
Teléfono: xxxxxxxx
NIF: xxxxxxxxm

December 18, 2022

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

   Escriba aquí su respuesta al ejercicio 1.

   EN la estructura almacenamos la fila y columna de cada celda y ambos son enteros. También almacenamos el valor que tiene la celda de tipo float.

   struct CellValue int row, col; float value; CellValue(int r = 0, int c = 0, float v = 0) : row(r), col(c), value(v) ;

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

   Escriba aquí su respuesta al ejercicio 2.

   void insertion$_sort(std :: vector < CellValue > v, intfirst, intlast)inti, j; CellValuec; for(i = first + 1; i <= last; i + +)c$

void fusion(std::vector¡CellValue¿ v, int i, int k, int j) int n = j - i + 1, p = i, q = k + 1, l; std::vector¡CellValue¿ c; c.resize(v.size());

for (l = 0; l ¡ n; l++) if (p ¡= k (q ¿ j —— v[p].value ¡= v[q].value)) c[l] = v[p]; p++; else c[l] = v[q]; q++; for (l = 0; l ¡ n; l++) v[i + l] = c[l];

void fusionSort(std::vector¡CellValue¿ v, int i, int j) int n = j - i + 1; int k;

   if (n ¡= 3) insertion$_sort(v, i, j); elsek = i - 1 + n/2; fusionSort(v, i, k); fusionSort(v, k + 1, j); fusion(v, i, k, j);$

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

   Escriba aquí su respuesta al ejercicio 3.

   void insertion$_sort(std :: vector < CellValue > v, intfirst, intlast)inti, j; CellValuec; for(i = first + 1; i <= last; i + +)c$

int pivot(std::vector¡CellValue¿ v, int i, int j) int n = i, k; CellValue c = v[i];

for (k = i + 1; k ¡= j; k++) if (v[k].value ¡= c.value) n++; std::swap(v[n], v[k]); v[i] = v[n]; v[n] = c;

return n;

void quickSort(std::vector¡CellValue¿ v, int i, int j) int n = j - i + 1; int p;

   if (n ¡= 3) insertion$_sort(v, i, j); elsep = pivot(v, i, j); quickSort(v, i, p - 1); quickSort(v, p + 1, j);$

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.

   Escriba aquí su respuesta al ejercicio 4.

   bool prueba$_vec$1()

bool ordenado = true; std::vector¡int¿v1=3,6; std::vector¡int¿v$_ordenado = 6, 3;$

quickSort(v1,0,v1.size()-1);

for(size$_ti = 0; i < v1.size(); i + +)if(v1[i]! = v_ordenado[i])ordenado = false;$

return ordenado;

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.

   Escriba aquí su respuesta al ejercicio 5.

   1)Ordenación por fusión.

   La idea detrás del divide y vencerás, es como el mismo nombre lo indica, dividir en sub-problemas que más adelante resuelvan el problema final. En el caso del Merge Sort, el lista será dividido en dos sub-listados, luego estos sub-listados serán nuevamente divididos y así sucesivamente, hasta que llega a un listado de dos elementos que puede ser fácilmente ordenado. Luego de llegar a estos sub-listados pequeños de 1 o 2 elementos, se comienzan a mezclar los sub-listados formando nuevos listados ordenados. Este proceso se repite hasta llegar al listados original que en última instancia, quedará ordenado.

   Todo este proceso, tiene una complejidad O(nlog2n) dado que la altura del árbol que se forma al dividir en sublistas el listado original es log2n donde n es la cantidad de elementos y en cada nivel del árbol hay que iterar sobre todos los elementos, la complejidad es O(n*log2n)

   2)Ordenación rápida.

   Elegimos un elemento del conjunto de elementos a ordenar que llamaremos pivote. Situamos los demás elementos de la lista a cada lado del pivote, a un lado los menos que él y al otro los mayores, los que sean iguales se pueden colocar en cualquiera de los lados, ocupando así el pivote su lugar correspondiente. La lista se queda separada en dos sublistas, este proceso se repite de forma recursiva, la eficiencia depende de la posición en la que se termine el pivote elegido.

   Mejor caso -¿ pivote termina en el centro de la lista, O(n*logn). Peor caso -¿ pivote termina en un extremo de la lista, $O(n^2)$.$Casopromedio- > O(n*logn).floatdefaultCellValue(introw, intcol, bool**freeCells, intnCellsWidth, intnC$
   $Object* > obstacles, List < Defense* > defenses)$

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

Vector3 cellPosition((col * cellWidth) + cellWidth * 0.5f, (row * cellHeight) + cellHeight * 0.5f, 0);

float val = 0; for (List¡Object*¿::iterator it=obstacles.begin(); it != obstacles.end(); ++it) val += $_distance(cellPosition, (*it)- >$
$position)$;

return val;

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). Considere en su análisis los planetas con códigos 1500, 2500, 3500,..., 10500. Incluya en el análisis los planetas que considere oportunos para mostrar información relevante.

   Escriba aquí su respuesta al ejercicio 6. ../graphic.eps

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.