Práctica 3. Divide y vencerás

Federico Carrillo Chaves
federico.carrilloch@alum.uca.es
Teléfono: +34615158732
NIF: 32095180Z

5 de diciembre de 2020

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

   El terreno de batalla está compuesto por celdas. Esas celdas se han codificado en una clase Celda únicamente con los siguientes miembros públicos:

   a) Atributo valor_ de tipo double.

   b) Atributo vector3_ de tipo Vector3 que nos viene ya declarado en una librería.

   c) Constructor que recibe ambos parámetros y constructor predeterminado.

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

   Mi implementación del algoritmo de ordenación por fusión (decreciente) para objetos Celda es la siguiente:

```
// Algoritmo basado en el pseudocodigo del libro "Introduction to Algorithms 3era edicion
    pagina 31"
void merge(std::vector<Celda> &V, int p, int q, int r) {
  int n1 = q - p + 1;
  int n2 = r - q;
  std::vector<Celda> L(n1), R(n2);
  for (int i = 0; i < n1; ++i) {
    L[i].valor_ = V[p + i].valor_;
  }
  for (int j = 0; j < n2; ++j) {
    R[j].valor_ = V[q + j + 1].valor_;
  }
  int i = 0;
  int j = 0;
  int z = p;
  while (i < n1 && j < n2) {
    if (L[i].valor_ >= R[j].valor_) { // ahora ordena de forma decreciente.
      V[z].valor_ = L[i].valor_;
      i++;
    } else {
      V[z].valor_ = R[j].valor_;
      j++;
    }
    z++;
  }
  while (i < n1) {
    V[z].valor_ = L[i].valor_;
    i++;
    z++;
  }
  while (j < n2) {
    V[z].valor_ = R[j].valor_;
    j++;
    z++;
  }
}
```

```
// Algoritmo basado en el pseudocodigo del libro "Introduction to Algorithms 3era edicion
    pagina 31"
void mergeSort(std::vector<Celda> &V, int p, int r) {
  if (p < r) {
    int q = (p + r) / 2;
    mergeSort(V, p, q);
    mergeSort(V, q + 1, r);
    merge(V, p, q, r);
  }
}
```

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

   Mi implementación del algoritmo de ordenación rápida (decreciente) para objectos Celda es la siguiente:

```
// Algoritmo basado en el pseudocodigo del libro "Introduction to Algorithms 3era edicion
    pagina 171"
int partition(std::vector<Celda> &V, int p, int r) {
  int x = V[r].valor_;
  int i = p - 1;
  for (int j = p; j <= r - 1; ++j) {
    if (V[j].valor_ >= x) { // ahora ordena de forma decreciente.
      i = i + 1;
      std::swap(V[i], V[j]);
    }
  }
  std::swap(V[i + 1], V[r]);
  return i + 1;
}
```

```
// Algoritmo basado en el pseudocodigo del libro "Introduction to Algorithms 3era edicion
    pagina 171"
void quickSort(std::vector<Celda> &V, int p, int r) {
  if (p < r) {
    int q = partition(V, p, r);
    quickSort(V, p, q - 1);
    quickSort(V, q + 1, r);
  }
}
```

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.

```
bool funcionComparacion(Celda const &celda1, Celda const &celda2) {
  return celda1.valor_ > celda2.valor_;
}
```

```
void pruebasCajaNegra() {
  // Parametros que recibe celda es valor_ y Vector3_ respectivamente.
  Celda c1(10, 0), c2(20, 0), c3(30, 0), c4(40, 0);
  Celda c5(50, 0), c6(60, 0), c7(70, 0);
  std::vector<Celda> v{c7, c6, c5, c4, c3, c2, c1};
  std::vector<Celda> copia_de_v(v.size());
  copia_de_v = v;
  std::vector<Celda> aux_mergeSort(v.size());
  std::vector<Celda> aux_quickSort(v.size());
  unsigned int ok = 0;
  unsigned int mal = 0;
  bool b;
  do {
    aux_mergeSort = v;
    aux_quickSort = v;
    mergeSort(aux_mergeSort, 0, v.size() - 1);
    quickSort(aux_quickSort, 0, v.size() - 1);
    b = true;
    for (int i = 0; i < v.size(); ++i) {
```

```
      if (aux_mergeSort[i].valor_ != copia_de_v[i].valor_ || aux_quickSort[i].valor_ !=
          copia_de_v[i].valor_)
        b = false;
    }
    ++(b ? ok : mal);
  } while (next_permutation(v.begin(), v.end(), funcionComparacion));
  std::cout << "Permutaciones OK: " << ok << ", Permutaciones Error: " << mal << std::endl;
}
```

Permutaciones OK: 5040, Permutaciones Error: 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.

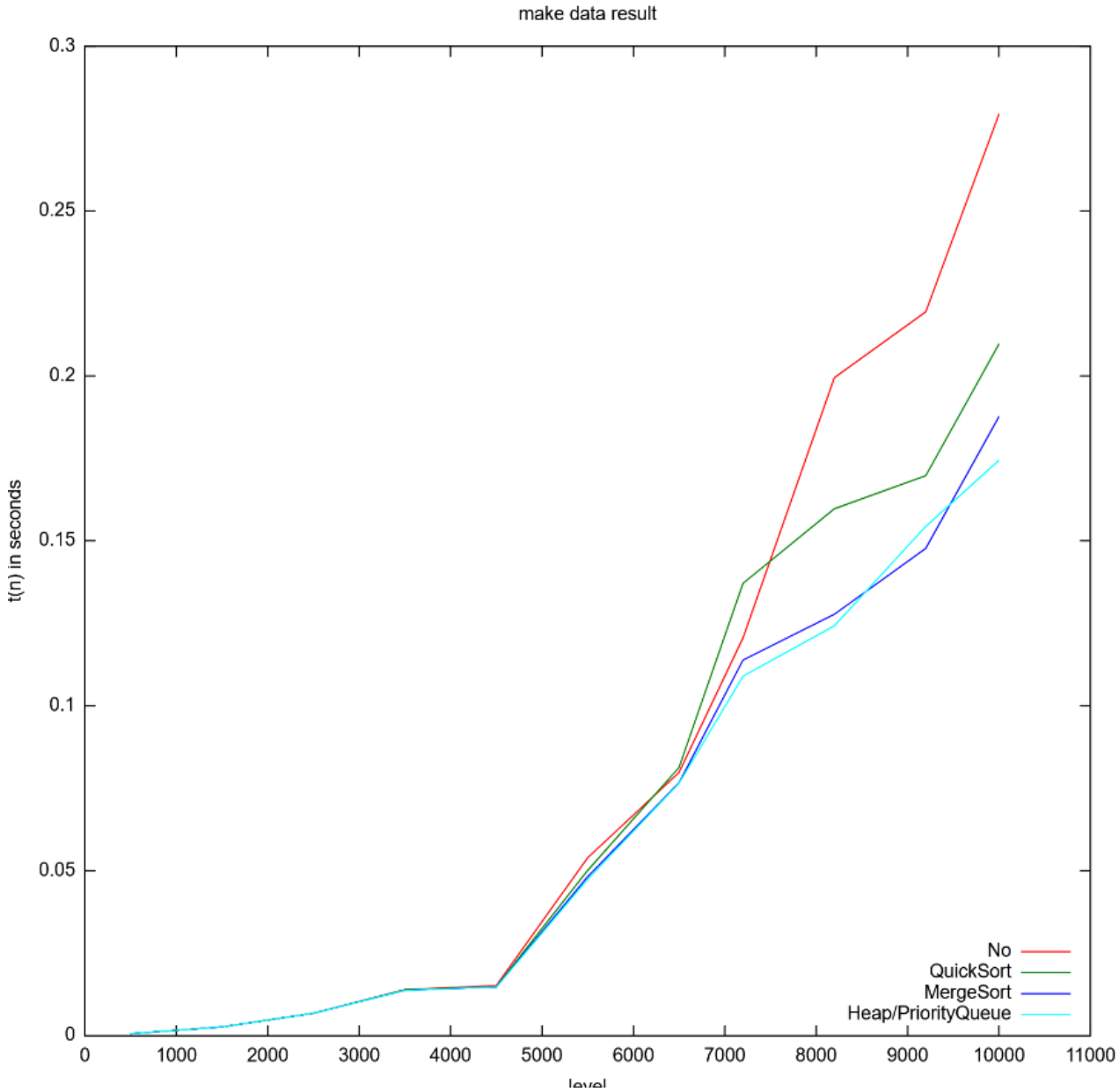
   Nuestros algoritmos de colocación de defensas tendrán que colocar las defensas en el mapa indicado. Nosotros hemos representado el terreno como una matriz cuadrada y hemos almacenados las celdas disponibles en un vector.

   - Sin preordenar. Nuestro algoritmo tendrá que ir recorriendo todas las celdas del terreno de batalla tantas veces como defensas haya disponible. Por lo tanto tendrá que recorrer ese vector en el mejor y peor caso. O(n2)
   - Usando algoritmos de ordenación por fusión y ordenación rápida. Ambos ordenarán ese vector una vez rellenado con las celdas candidatas con O(n log n), pero con al ventaja que para seleccionar una celda con mayor valor siempre es O(1) al estar ordenado.
   - Usando montículo / cola de prioridad. En esta ocasión el vector se va ordenando cada vez que se inserta y se extrae un elemento con un O(n log n). Teóricamente este método es óptimo cuando puede entrar una celda en cualquier momento y se ordenaría "sola" en lugar de tener que llamar otra vez a la función para que lo ordenase.

   Como veremos en la gráfica del ejercicio siguiente, tener que buscar un número mucho menor de defensas en algunos mapas que el número de celdas disponibles hace que sea conveniente usar un algoritmo de colocación de defensas sin ordenación y buscar las pocas veces que haga falta colocar una defensa entre las celdas disponibles en lugar de ordenar un vector enorme para no sacarle mucho partido. También hay que tener en cuenta que el algoritmo con ordenación o montícuo serán mejores opciones dependiendo de si en ese mapa va a ser dificil colocar las defensas en las celdas con mayor valoración cuando sea dificil que sean celdas factibles (ahí sí se sacaría partido al tener las celdas ordenadas e ir seleccionando en orden constante).

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.

   Como podemos observar, hay veces que no se cumplen los valores teóricos esperados. Eso es debido a que en algunos mapas, al ser las defensas mucho menores que el tamaño del mapa, sale mejor buscar pocas veces en el mapa que ordenar todo para unas pocas ocasiones. Pero también vemos como para un mapa avanzado donde haya muchas defensas o haya muchas ocasiones que la celda no pase la función de factibilidad sí es más eficiente usar una ordenación, siendo la más eficiente según mis mediciones el montículo.



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.