Práctica 3. Divide y vencerás

Alejandro Serrano Fernandez
ale.serranofer@alum.uca.es
Teléfono: 640217690
NIF: 20501318S

21 de diciembre de 2020

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

   Para representar el terreno de batalla, primero he hecho uso de un un struct, llamado Celda donde almacenaremos las posiciones de la fila y columna, junto con el valor determinado por la función defaultCellValue. Estas celdas, serán posteriormente almacenadas en un vector de longitud N*M, siendo N el número de celdas a lo ancho y M el número de celdas a lo alto.

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

```
void ordenacion_fusion(std::vector<Celda>& v, int i, int j)
{
        int n = j - i + 1;
        int k = 0;

        if(n <= 3)
                ordenacion_insercion(v,i,j);
        else
        {
                k = i - 1 + n/2;
                ordenacion_fusion(v,i,k);
                ordenacion_fusion(v, k+1, j);
                fusion(v,i,k,j);
        }
}



void fusion(std::vector<Celda>& v, int i, int k, int j)
{
        int n = j - i + 1;
        int p = i;
        int q = k;
        std::vector<Celda> w;

        for(int l = 0; l < n; l++)
        {
                if(p < k && (q > j-1 || v[p] <= v[q]))
                {
                        w.push_back(v[p]);
                        p = p + 1;
                }
                else
                {
                        w.push_back(v[q]);
                        q = q + 1;
                }
        }

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

```
}


void ordenacion_insercion(std::vector<Celda>& v, int i, int j)
{
        int k;
        Celda aux;

        for(int t = i; t <= j; t++)
        {

                aux = v[t];
                for(k = t; k > 0 && (aux < v[k-1]); k--)
                {
                        v[k] = v[k-1];
                }
                v[k] = aux;
        }
}
```

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

```
void ordenacion_rapida(std::vector<Celda>& v, int i, int j)
{
        int p;
        int n = j - i + 1;

        if(n <= 3)
        {
                ordenacion_insercion(v,i,j);
        }
        else
        {
                p = pivote(v,i,j);
                ordenacion_rapida(v,i,p-1);
                ordenacion_rapida(v,p+1,j);
        }
}

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

        v[i] = v[p];
        v[p] = x;

        return p;
}

void ordenacion_insercion(std::vector<Celda>& v, int i, int j)
{
        int k;
        Celda aux;

        for(int t = i; t <= j; t++)
        {
```

```
                aux = v[t];
                for(k = t; k > 0 && (aux < v[k-1]); k--)
                {
                        v[k] = v[k-1];
                }
                v[k] = aux;
        }
}
```

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.

```
void caja_negra()
{
        std::vector<Celda> v;
        std::vector<Celda> w;
        std::vector<Celda> v2;


        for(int i = 0; i != 6; i++)
        {
                v.push_back(Celda(0,0,i));
                w.push_back(Celda(0,0,i));
        }

        /*      ORDENACION POR FUSION   */
        do{
                v2 = v;
                ordenacion_fusion(v2, 0, v2.size()-1);
                if(!comprobar_ordenado(v2))
                        std::cout<<"ERROR: Con fusion no esta ordenado"<<std::endl;

        }while(std::next_permutation(v.begin(),v.end()));


        /*      ORDENACION RAPIDA       */
        do{
                v2 = w;
                ordenacion_rapida(v2, 0, v2.size()-1);
                if(!comprobar_ordenado(v2))
                        std::cout<<"ERROR: Con ord.rapida no esta ordenado"<<std::endl;

        }while(std::next_permutation(w.begin(),w.end()));

}
```

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.

   -Complejidad Espacial: Para todos los algoritmos de ordenación usados en esta práctica (incluido el método sin ordenación), he hecho uso de dos vectores de Celdas, de tal manera que en uno se almacenará todas la celdas ya valoradas, y otro donde copiaré el contenido del anterior vector por motivos de eficiencia, para no evaluar de nuevo todas las celdas, pues la evaluación tiene un coste temporal de n (número de celdas). En este caso, estaríamos hablando de una complejidad espacial de 2*n, siendo n el número de celdas del mapa.

   -Complejidad Temporal: Siendo n el numero de celdas y p el número de defensas a colocar, para el método de colocación de defensas sin preordenación obtendríamos una complejidad de n + p*(n*n), es decir, el coste de dar valor a cada celda, más el número de defensas a colocar por el coste de buscar el mayor elemento (orden n) por el coste de recorrer todas las celdas hasta encontrar una factible.

   Para el algoritmo de ordenación por fusión obtendríamos una complejidad temporal de orden n*log n, luego la complejidad temporal del algoritmo de colocación de defensas sería de n + n*log n + n*p, es decir, el coste de valorar cada celda, más ordenar las celdas más el coste de colocar una defensa por el el coste de recorrer todas las celdas hasta encontrar una celda factible.

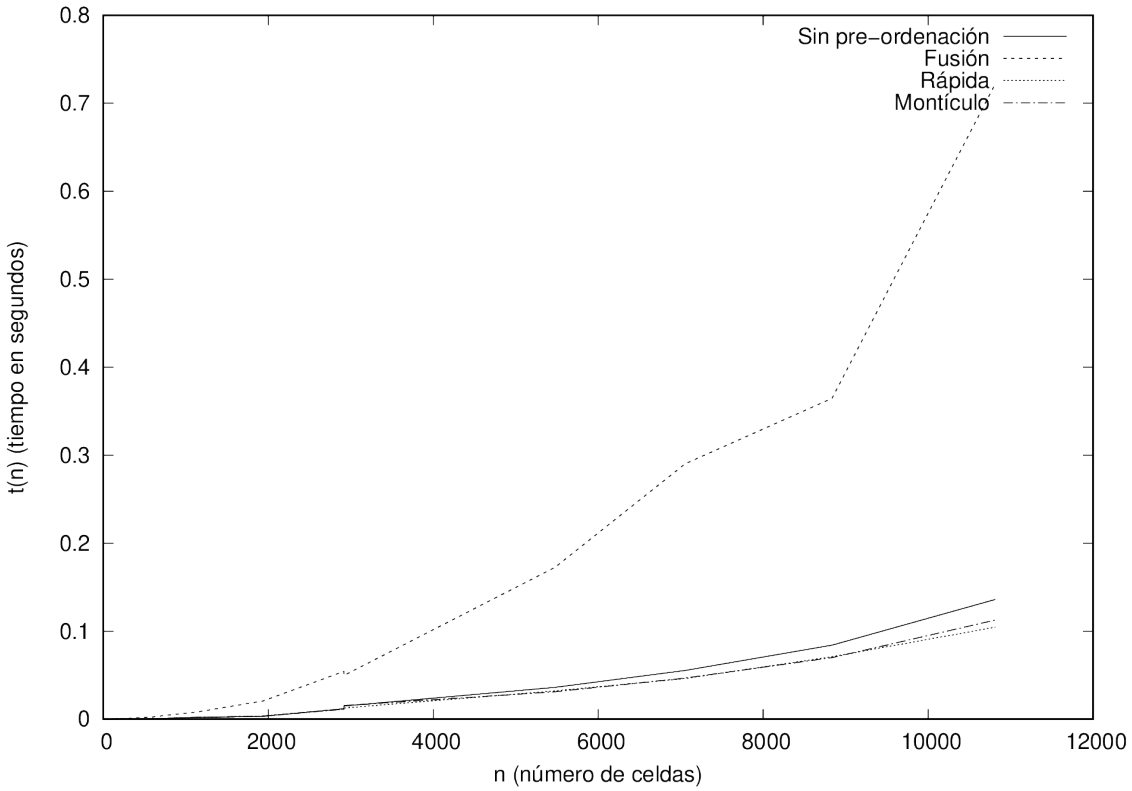
Para el algoritmo de ordenacion rápida obtendríamos en el peor caso una complejidad temporal de orden n^2, luego la complejidad temporal del algoritmo de colocación de defensas sería de n + n^2 + n*p.

En el mejor caso del algoritmo de ordenación rápida obtendríamos una complejidad temporal de orden n*log n, luego la complejidad temporal del algoritmo de colocación de defensas sería de n + n*log n + n*p

Para el algoritmo de ordenación por montículo obtendríamos una complejidad temporal de orden n*log n, luego la complejidad temporal del algoritmo de colocación de defensas sería de n(valorar las celdas) + n*log n (su creación) + n*log n(ordenar) + n*p

A la vista de los resultados obtenidos, el algoritmo que menos tiempo de ejecución ha necesitado, ha sido el algoritmo de ordenación rápida, aunque con el algoritmo de ordenación por montículo obtenemos resultados parecidos aunque tarda ligeramente más que el de ordenacion rapida. El resultado que más me ha impactado ha sido el del algoritmo de fusión, pues con éste obtenemos unos resultados desastrosos en comparación a los demás algoritmos. Vease en el gráfico que se muestra en el ejercicio 6.

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.




```
cronometro c1, c2, c3, c4;
long int r1 = 0,r2 = 0,r3 = 0,r4 = 0;
const double e_abs = 0.01, // Maximo error absoluto cometido.
        e_rel = 0.001; // Maximo error relativo aceptado.

c1.activar();
do {

    colocado = false;
    //Ordenamos para obtener siempre los de mayor valor
    celdasCandidatas2 = celdasCandidatas;
```

```
    currentDefense = defenses.begin();

    //Vamos colocando las defensas restantes y comprobando aquellas que sean factibles
    while(currentDefense != defenses.end())
    {
        colocado = false;

        while(!celdasCandidatas2.empty() && !colocado)
        {
                sin_ordenacion(celdasCandidatas2);

                if(factible(celdasCandidatas2.back(), freeCells, nCellsWidth, nCellsHeight,
                    mapWidth, mapHeight, obstacles, defenses, currentDefense))
                {
                        (*currentDefense)->position.x = (celdasCandidatas2.back().row_ *
                            cellWidth) +  (0.5f * cellWidth);
                        (*currentDefense)->position.y = (celdasCandidatas2.back().col_ *
                            cellHeight) + (cellHeight * 0.5f);
                        colocado = true;
                }
                celdasCandidatas2.pop_back();
        }
        currentDefense++;
    }

    ++r1;

} while(c1.tiempo() <  e_abs / e_rel + e_abs);
c1.parar();




c2.activar();
do {

    colocado = false;
    //Ordenamos para obtener siempre los de mayor valor
    celdasCandidatas2 = celdasCandidatas;


    ordenacion_fusion(celdasCandidatas2, 0, celdasCandidatas2.size()-1);


    currentDefense = defenses.begin();

    //Vamos colocando las defensas restantes y comprobando aquellas que sean factibles
    while(currentDefense != defenses.end())
    {
        colocado = false;

        while(!celdasCandidatas2.empty() && !colocado)
        {
                if(factible(celdasCandidatas2.back(), freeCells, nCellsWidth, nCellsHeight,
                    mapWidth, mapHeight, obstacles, defenses, currentDefense))
                {
                        (*currentDefense)->position.x = (celdasCandidatas2.back().row_ *
                            cellWidth) +  (0.5f * cellWidth);
                        (*currentDefense)->position.y = (celdasCandidatas2.back().col_ *
                            cellHeight) + (cellHeight * 0.5f);
                        colocado = true;
                }
                celdasCandidatas2.pop_back();
        }
        currentDefense++;
    }

    ++r2;
```

```
} while(c2.tiempo() <  e_abs / e_rel + e_abs);
c2.parar();



c3.activar();
do {

    colocado = false;
    //Ordenamos para obtener siempre los de mayor valor
    celdasCandidatas2 = celdasCandidatas;


    ordenacion_rapida(celdasCandidatas2, 0, celdasCandidatas.size()-1);
    currentDefense = defenses.begin();

    //Vamos colocando las defensas restantes y comprobando aquellas que sean factibles
    while(currentDefense != defenses.end())
    {
        colocado = false;

        while(!celdasCandidatas2.empty() && !colocado)
        {
                if(factible(celdasCandidatas2.back(), freeCells, nCellsWidth, nCellsHeight,
                    mapWidth, mapHeight, obstacles, defenses, currentDefense))
                {
                        (*currentDefense)->position.x = (celdasCandidatas2.back().row_ *
                            cellWidth) +  (0.5f * cellWidth);
                        (*currentDefense)->position.y = (celdasCandidatas2.back().col_ *
                            cellHeight) + (cellHeight * 0.5f);
                        colocado = true;
                }
                celdasCandidatas2.pop_back();
        }
        currentDefense++;
    }

    ++r3;

} while(c3.tiempo() <  e_abs / e_rel + e_abs);
c3.parar();



c4.activar();
do {

    colocado = false;
    //Ordenamos para obtener siempre los de mayor valor
    celdasCandidatas2 = celdasCandidatas;


    ordenacion_monticulo(celdasCandidatas2);
    currentDefense = defenses.begin();

    //Vamos colocando las defensas restantes y comprobando aquellas que sean factibles
    while(currentDefense != defenses.end())
    {
        colocado = false;

        while(!celdasCandidatas2.empty() && !colocado)
        {
                if(factible(celdasCandidatas2.back(), freeCells, nCellsWidth, nCellsHeight,
                    mapWidth, mapHeight, obstacles, defenses, currentDefense))
                {
                        (*currentDefense)->position.x = (celdasCandidatas2.back().row_ *
                            cellWidth) +  (0.5f * cellWidth);
                        (*currentDefense)->position.y = (celdasCandidatas2.back().col_ *
                            cellHeight) + (cellHeight * 0.5f);
```

```
                        colocado = true;
                }
                celdasCandidatas2.pop_back();
        }
        currentDefense++;
    }

    ++r4;

} while(c4.tiempo() <  e_abs / e_rel + e_abs);
c4.parar();


std::cout << (nCellsWidth * nCellsHeight) << '\t' << c1.tiempo() / r1 << '\t' << c2.tiempo()
    / r2 << '\t' << c3.tiempo() / r3 << '\t' << c4.tiempo() / r4 << std::endl;
```

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.