Práctica 1. Algoritmos devoradores

AGUSTIN ENEAS HARISPE LUCARELLI
correo@servidor.com
Teléfono: xxxxxxxx
NIF: 55080402W

14 de junio de 2023

1. Describa a continuación la función diseñada para otorgar un determinado valor a cada una de las celdas del terreno de batalla para el caso del centro de extracción de minerales.

   La estrategia a seguir en esta práctica para el centro de extracción de minerales se basa en colocarla lo más cerca del centro posible. La función debe darle un mayor valor a la celda que más cerca este del centro del mapa. Donde x es la diferencia entre la posición del mapa y el centro de la celda a evaluar podemos decir que nuestra función es racional, o sea del tipo f(x)=k/x, donde k es un entero positivo para ajustar nuestra escala.

2. Diseñe una función de factibilidad explicita y descríbala a continuación.

   Recibimos la posicion x,y del centro de la celda que evaluamos además del tamaño del mapa junto con la lista de denfensas y la defensa que queremos colocar. La funcion devuelve True en caso de que sea factible, False en caso contrario

```
bool factibilidad(float x, float y, float mapWidth, float mapHeight, List<Defense*>::iterator
     defensa,
List<Defense*> defenses, List<Object*> obstacles){
        bool esFactible = true;

        float radio = (*defensa)->radio;

        if(x<radio || y<radio || x + radio > mapWidth || y + radio > mapHeight){ esFactible =
             false;}

        List<Object*>::iterator currentObstacle = obstacles.begin();
        while(esFactible && currentObstacle != obstacles.end()){
                Vector3 diferencia = Vector3(x,y,0) - (*currentObstacle)->position;
                if(diferencia.length() < radio + (*currentObstacle)->radio) {esFactible =
                    false;}
                ++currentObstacle;
        }

        List<Defense*>::iterator currentDefense = defenses.begin();
        while(esFactible && (*currentDefense)->id != (*defensa)->id){
                Vector3 diferencia = Vector3(x,y,0) - (*currentDefense)->position;
                if(diferencia.length() < radio + (*currentDefense)->radio) {esFactible =
                    false;}
                ++currentDefense;
        }
        return esFactible;
}
```

3. A partir de las funciones definidas en los ejercicios anteriores diseñe un algoritmo voraz que resuelva el problema para el caso del centro de extracción de minerales. Incluya a continuación el código fuente relevante.

```
float cellValue(int row, int col, bool** freeCells, int nCellsWidth, int nCellsHeight
        , float mapWidth, float mapHeight, List<Object*> obstacles, List<Defense*> defenses)
            {

    float cellWidth = mapWidth / nCellsWidth;
    float cellHeight = mapHeight / nCellsHeight;
    float centro_x = col*cellWidth+cellWidth*0.5f;
    float centro_y = row*cellHeight+cellHeight*0.5f;
    float mitad = mapWidth/2;
    float resultado = ((1000)/(abs(centro_x - mitad) + abs(centro_y - mitad) + 0.001f)); //
        offset para que no de 0
    if(!freeCells) resultado = 0.0f;
        return resultado;
}
```

4. Comente las características que lo identifican como perteneciente al esquema de los algoritmos voraces.

   Podemos ver como el programa sigue el esquema de un algoritmo voraz:

   - Conjunto de candidatos: celdas.
   - Función factibilidad: la función factibilidad.
   - Funcion de selección: extraerValores;
   - Solución: colocar defensas.
   - Objetivo: maximizar los segundos que sobrevive el centro de extracción.

5. Describa a continuación la función diseñada para otorgar un determinado valor a cada una de las celdas del terreno de batalla para el caso del resto de defensas. Suponga que el valor otorgado a una celda no puede verse afectado por la colocación de una de estas defensas en el campo de batalla. Dicho de otra forma, no es posible modificar el valor otorgado a una celda una vez que se haya colocado una de estas defensas. Evidentemente, el valor de una celda sí que puede verse afectado por la ubicación del centro de extracción de minerales.

   En este caso, una vez colocado el centro de extracción de minerales, la estrategia es homónima a la anterior. El objetivo será minimizar la distancia entre el centro de extracción y las defensas a colocar.

   Para conseguir esto volvemos a una función racional k/x donde k es un entero positivo que ajusta la escala y x la distancia entre el centro de la celda y la posicion del centro de extracción.

6. A partir de las funciones definidas en los ejercicios anteriores diseñe un algoritmo voraz que resuelva el problema global. Este algoritmo puede estar formado por uno o dos algoritmos voraces independientes, ejecutados uno a continuación del otro. Incluya a continuación el código fuente relevante que no haya incluido ya como respuesta al ejercicio 3.

```
void DEF_LIB_EXPORTED placeDefenses(bool** freeCells, int nCellsWidth, int nCellsHeight,
    float mapWidth, float mapHeight
              , std::list<Object*> obstacles, std::list<Defense*> defenses) {

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

                int N = nCellsWidth*nCellsHeight;
                float** valores = new float* [N];
                for(int i=0; i<N;i++){
                        valores[i] = new float[2]; // [][0] = pos [][1] valores cellValue
                        valores[i][0] = i;
                        valores[i][1] = cellValue(i/nCellsWidth, i%nCellsWidth, freeCells,
                            cellWidth,cellHeight
, mapWidth, mapHeight, obstacles, defenses); // i/ncellswidth=row y i%ncellsheight=col
                }

                int aux0; float aux1;
                for(int i=0;i<N-1;i++){
```

```
                        for(int j=0; j<N-1;j++){
                                if(valores[j][1]<valores[j+1][1]){
                                        aux0 = valores[j][0];
                                        aux1 = valores[j][1];
                                        valores[j][0] = valores[j+1][0];
                                        valores[j][1] = valores[j+1][1];
                                        valores[j+1][0] = aux0;
                                        valores[j+1][1] = aux1;
                                }
                        }
                } //preordena


    List<Defense*>::iterator currentDefense = defenses.begin();
                float posx, posy; int row, col, i = 0;

    while(currentDefense != defenses.end()) {
                        row = (int)valores[i][0]/nCellsHeight;
                        col = (int)valores[i][0]%nCellsWidth;
                        posx= col * cellWidth + cellWidth * 0.5f;
                        posy= row * cellHeight + cellHeight * 0.5f;
                        if(factibilidad(posx, posy, mapWidth,mapHeight,currentDefense,
                            defenses,obstacles)){
                                (*currentDefense)->position.x = posx;
                                (*currentDefense)->position.y = posy;
                                (*currentDefense)->position.z = 0;
                                ++currentDefense;

                        }
                        i++;
}
```

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.