Práctica 1. Algoritmos devoradores

Jesús Rodríguez Heras
jesus.rodriguezheras@alum.uca.es
Teléfono: 628576107
NIF: 32088516C

17 de noviembre de 2018

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 función que evalúa las celdas para el caso del centro de extracción de minerales se llama `cellValue` y recibe los siguientes parámetros:

   - `int row`: Número de fila de la celda de la matriz del terreno de batalla.
   - `int col`: Número de columna de la celda de la matriz del terreno de batalla.
   - `bool** freeCells`: Matriz de booleanos que indica si una celda de la matriz está libre (`true`) o no (`false`).
   - `int nCellsWidht`: Número de celdas de ancho.
   - `int nCellsHeight`: Número de celdas de alto.
   - `float mapWidth`: Tamaño de la anchura del mapa.
   - `float mapHeight`: Tamaño del alto del mapa.
   - `flaot cellWidht`: Tamaño de la anchura de una celda.
   - `float cellHeight`: Tamaño del alto de una celda.
   - `std::list<Object*> obstacles`: Lista de obstáculos del juego.
   - `std::list<Defense*> defenses`: Lista de defensas del juego.

   Para dar valores a las celdas de la matriz del terreno de batalla la función comprueba primeramente que la celda de la matriz no está ocupada. Si está ocupada, devuelve `false`.

   Si no está ocupada creamos una variable de tipo `Vector3` llamada `posibleDefensa` con la posición de esa celda y luego vamos recorriendo la lista de obstáculos mientras que vamos guardando en la variable `maxDistancia` (de tipo `float`) la máxima de la inversa de la distancia euclídea (proporcionada por la función `_distance()`) entre la posición de `posibleDefensa` y todos los obstáculos del terreno de batalla.

   Finalmente devuelve la variable `maxDistancia` que será el valor que tenga esa celda.

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

   La función de factibilidad diseñada recibe los siguientes parámetros de entrada:

   - `float x`: Posición en el eje "X" donde se va a comprobar la factibilidad de colocar una defensa.
   - `float y`: Posición en el eje "Y" donde se va a comprobar la factibilidad de colocar una defensa.
   - `Defense* defensa`: Defensa que se va a comprobar.
   - `std::list<Object*> obstaculos`: Lista de obstáculos del terreno de juego.
   - `float mapWidth`: Tamaño de la anchura del mapa.
   - `float mapHeight`: Tamaño del alto del mapa.
   - `std::list<Defense*> defensas`: Lista de defensas del juego.
   - `float cellWidth`: Ancho de las celdas.

- `float cellHeight`: Alto de las celdas.

Primeramente se declara una variable de tipo lógica llamada `entra` inicializada a `true`, que será lo que retornará esta función.

Luego se comprueban los siguientes factores:

a) La defensa entra en el mapa. Es decir, no se sale por los bordes del mismo. Lo cual es calculado teniendo en cuenta el radio de la defensa y las dimensiones (ancho y alto) del mapa. En caso contrario, la variable `entra` tomaría el valor `false`.

b) Colisión con obstáculo: Se comprueba que la defensa no colisiona con ninguno de los obstáculos del terreno de juego. En caso contrario, la variable `entra` tomaría el valor `false`.

c) Colisión con otra defensa anteriormente colocada: Se recorre la lista de defensas comprobando que, de las ya colocadas, es decir, las $n-1$ defensas anteriores, no colisionen con la defensa que queremos colocar. En caso contrario, la variable `entra` tomaría el valor `false.`

Finalmente se devuelve el valor de la variable `entra`.

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.

```
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 maxAttemps = 1000;
    int fila = 0, columna = 0;
    float x = 0, y = 0;

        float** mapa = new float*[nCellsHeight];
        for (size_t i = 0; i < nCellsHeight; ++i) {
                mapa[i] = new float[nCellsWidth];
        }

        for (size_t i = 0; i < nCellsHeight; ++i) {
                for (size_t j = 0; j < nCellsHeight; ++j) {
                        mapa[i][j] = cellValue(i, j, freeCells, nCellsWidth, nCellsHeight,
                            mapWidth, mapHeight, cellWidth, cellHeight, obstacles, defenses);
                }
        }

        std::list<Defense*>::iterator currentDefense = defenses.begin();
        while(currentDefense == defenses.begin() && maxAttemps > 0){
                seleccion(mapa, nCellsWidth, nCellsHeight, &fila, &columna);
                x = fila*cellWidth + cellWidth*0.5f;
                y = columna*cellHeight + cellHeight*0.5f;
                if(factibilidad(x, y, (*currentDefense), obstacles, mapWidth, mapHeight,
                    defenses, cellWidth, cellHeight)){
                        (*currentDefense)->position.x = x;
                        (*currentDefense)->position.y = y;
                        (*currentDefense)->position.z = 0;

                        cellValueCentro(mapa, nCellsWidth, nCellsHeight, cellWidth,
                            cellHeight, mapWidth, mapHeight, defenses);

                        ++currentDefense;
                }
                --maxAttemps;
        }
    //El resto del codigo no pertenece al caso del centro de extraccion de minerales
}
```

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

El algoritmo diseñado contiene los siguientes elementos que se corresponden con los de un algoritmo voraz:

- **Conjunto de candidatos:** Lista de defensas a colocar en el terreno de batalla.
- **Función de selección:** Indica la celda más prometedora de las que quedan disponibles en el terreno de batalla.
- **Función de factibilidad:** Comprueba si una defensa dada puede colocarse en la celda elegida por la función de selección.

Inicialmente, el algoritmo contiene un bucle cuya condición de parada contiene dos factores: un número de intentos (establecido a 1000 por defecto) y una condición booleana que se correspondería con la solución de la estructura de los algoritmos voraces.

También contiene una función de selección que, en su interior impondrá un valor de 0 a las celdas que hayan sido eliminadas de la solución para que no puedan ser seleccionadas de nuevo.

A continuación, tenemos una función booleana de factibilidad que nos devolverá `true` si la es posible incluir la selección realizada por la función de selección puede incluirse en la solución del problema. En caso contrario, devolverá `false`.

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 esta función se intenta puntuar las celdas más cercanas al centro con un valor superior a las demás con la intención de colocarlas rodeando el centro de extracción de minerales para defenderlo mejor.

   La función que evalúa las celdas para el caso del centro de extracción de minerales se llama `cellValueCentro` y recibe los siguientes parámetros:

   - `float** mapa`: Matriz que representa el mapa de juego.
   - `int nCellsWidth`: Número de celdas de ancho que tiene el mapa.
   - `int nCellsHeight`: Número de celdas de alto que tiene el mapa.
   - `float cellWidth`: Ancho de las celdas.
   - `float cellHeight`: Alto de las celdas.
   - `float mapWidth`: Tamaño de la anchura del mapa.
   - `float mapHeight`: Tamaño del alto del mapa.
   - `std::list<Defense*> defensas`: Lista de defensas del juego.

   Para dar valores a las celdas de la matriz del terreno de batalla la función calcula la posición del centro de extracción respecto de la diferencia con los bordes del mapa en una variable de tipo `Vector3` llamado `posicionDefensa`.

   Luego creamos una variable de tipo `Vector3`, llamado `posMax`, que estará formada por el tamaño total del mapa en sus coordenadas "X" e "Y".

   A continuación, se asigna el valor de cada celda como la resta de los módulos de ambos vectores, `posMax` y `posicionDefensa`.

   El valor de las celdas es devuelto por la misma variable `mapa` por referencia.

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.

   Continuación de la función `placeDefenses` del 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 maxAttemps = 1000;
    int fila = 0, columna = 0;
        float x = 0, y = 0;

    // ...
        // Ejercicio 3
    // ...
    // Continuacion del ejercicio 3

        while (currentDefense != defenses.end() && maxAttemps > 0) {
                seleccion(mapa, nCellsWidth, nCellsHeight, &fila, &columna);
                x = fila*cellWidth + cellWidth*0.5f;
                y = columna*cellHeight + cellHeight*0.5f;
                if(factibilidad(x, y, (*currentDefense), obstacles, mapWidth, mapHeight,
                    defenses, cellWidth, cellHeight)){
                        (*currentDefense)->position.x = x;
                        (*currentDefense)->position.y = y;
                        (*currentDefense)->position.z = 0;
                        ++currentDefense;
                }
                --maxAttemps;
        }

    // Resto de codigo de PRINT_DEFENSE_STRATEGY que no hemos modificado

}
```

Función `factibilidad`:

```
bool factibilidad(float x, float y, Defense* defensa, std::list<Object*> obstaculos, float
    mapWidth, float mapHeight, std::list<Defense*> defensas, float cellWidth, float
    cellHeight){
        bool entra = true;

        if (x-defensa->radio < 0 || x+defensa->radio > mapWidth || y-defensa->radio < 0 || y+
            defensa->radio > mapHeight) {
                entra = false; //No cabe porque se sale de los limites del mapa
        }

        std::list<Object*>::const_iterator iterObst = obstaculos.begin();
        std::list<Defense*>::const_iterator iterDef = defensas.begin();
        Vector3 posicionDefensa(x, y, 0);
        while (iterObst!=obstaculos.end()) {
                if ((defensa->radio + (*iterObst)->radio) > (_distance(posicionDefensa, (*
                    iterObst)->position))) {
                        entra = false; //Se choca con un obsaculo
                }else{
                        while ((*iterDef)!=defensa) {
                                if ((defensa->radio + (*iterDef)->radio) > (_distance(
                                    posicionDefensa, (*iterDef)->position))) {
                                        entra = false; //Se choca con una defensa
                                }
                                ++iterDef;
                        }
                }
                ++iterObst;
        }

        return entra;
}
```

Función `selección`:

```
void seleccion(float** mapa, int nCellsWidth, int nCellsHeight, int* fila, int* columna){
        float maxi = 0;
        for (size_t i = 0; i < nCellsWidth; ++i) {
                for (size_t j = 0; j < nCellsHeight; ++j) {
                        if (mapa[i][j] > maxi) {
                                maxi = mapa[i][j];
                                *fila = i;
                                *columna = j;
```

```
                        }
                }
        }
        mapa[*fila][*columna] = 0;
}
```

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.