Práctica 1. Algoritmos devoradores

Antonio Roldan Andrade
antonio.roldanandrade@alum.uca.es
Teléfono: +34611404497
NIF: 49562495W

19 de mayo 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.

   Para realizar este ejercicio he determinado usar el criterio de cercania con respecto a los obstaculos distribuidos en el mapa, es decir, cuanto mas cerca a los obstaculos este la celda dada mejor puntuacion. La puntuación se calcula en base a la suma de las distancias que separan a posicion creada a partir de un objeto tipo Vector3 y la funcion cellCenterToPosition, con esto podremos calcular la distancia con respecto a la lista de obstaculos que se reciben como parametro.Por tanto la puntuación sera el inverso de la suma de las distancias. Anotación: El código de esta función esta "incrustado" en el ejercicio nº 5.

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

```
bool funcion_factibilidad(List<Defense *> defenses, const Defense &d, List<Object *>
    obstacles, float mapHeight,
               float cellWidth, float cellHeight, float mapWidth, int row, int col, bool **
                   freeCells)
{

    //definicion de variables
    Vector3 p = cellCenterToPosition(row,col,cellWidth,cellHeight); //Creamos la posicion con
         los datos dados
    bool token = true;

    //Primeiro comprobamos que la celda no esta en ninguna posicion limite
    if(p.x + d.radio > mapWidth ||p.x - d.radio < 0 ||
        p.y + d.radio > mapHeight ||p.y - d.radio < 0)
    {
        return false; //si se cumple alguna condicion la defensa no cabe al alcanzar
            posiciones limite del mapa
    }
        //Comprobaremos que no colisionan con las defensas/obstaculos que ya estan colocadas
        for(auto i : obstacles){
            //Colisionara en caso de que las distancias entre puntos centrales de los
                obstaculos
            //sea menor que los radios de la defensa a colocar y el obstaculo
            if(_distance(p,i->position) < (d.radio + i->radio ))
                token = false;
        }
        if(token){
            //Se comprobara de forma similar a los obstaculos con las defensas
            for (auto i: defenses){
                if(_distance(p,i->position) < (d.radio + i->radio ))
                    token = false;
            }
        }
```

no es necesario especificar la extensión del archivo que contiene la imagen

Figura 1: Estrategia devoradora para la mina

```
    return token;

}
```

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.

```
struct celda_valoracion
{
    int row, col;
    float value;
    celda_valoracion(int r = 0, int c = 0, float v = 0) : row(r), col(c), value(v) {}

    // Overload ctor de copia
    celda_valoracion operator=(const celda_valoracion &c)
    {
        this->row = c.row;
        this->col = c.col;
        return *this;
    }

    // Overload operator < (En caso de ordencacion por lista)
    bool operator<(const celda_valoracion &c)
    {
        return value < c.value;
    }
};

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;
    std::vector<celda_valoracion> celdas_valoradas;
    int i, j, k;
    float aux;
    celda_valoracion auxCell;

    /* ------------------------------------------------------------------------ */
    /*                  ASINGACION DEL CENTRO DE EXTRACCION (DEF 0)             */
    /* ------------------------------------------------------------------------ */

    // 1) Obtenemos que celdas estan libres tras la colocacion de los obstaculos
    for (int i = 0; i < nCellsHeight; i++)
    {

        for (int j = 0; j < nCellsWidth; j++)
        {
            if (freeCells[i][j] != false)
            {

                celdas_valoradas.push_back(celda_valoracion(i, j, cellValueExtractionCenter(i
                    , j, freeCells, nCellsWidth, nCellsHeight, mapWidth, mapHeight, obstacles
                    )));
            }
        }
    }

    // Ordenacion de las celdas valoradas aplicando los constructores de vector y lista

    std::list<celda_valoracion> aux_lista(celdas_valoradas.begin(), celdas_valoradas.end());
        // Creo una lista con los elementos de celdas
```

```
    aux_lista.sort();
        // Ordeno los elementos O(n) = n    log n
    celdas_valoradas = std::vector<celda_valoracion>(aux_lista.begin(), aux_lista.end());
        // los copio a la lista

    bool placed = false;
    celda_valoracion solution;
    List<Defense *>::iterator currentDefense = defenses.begin();

    // Algoritmo devorador para centro de extraccion
    while (!placed && !celdas_valoradas.empty())
    {
        solution = celdas_valoradas.back();
        celdas_valoradas.pop_back();
        if (funcion_factibilidad(defenses, *(*currentDefense), obstacles, mapHeight,
                                 cellWidth, cellHeight, mapWidth, solution.row, solution.col,
                                      freeCells))
        {
            placed = true;
            freeCells[solution.row][solution.col] = false;
            (*currentDefense)->position = cellCenterToPosition(solution.row, solution.col,
                cellWidth, cellHeight);
        }
    }
    // Copia de los candidatos

    std::vector<celda_valoracion> celdas_libres_aux;

    // Insertamos las celdas aun libres en el vector
    for (i = 0; i < nCellsHeight; i++)
        for (j = 0; j < nCellsWidth; j++)
        {
            if (freeCells[i][j])
                celdas_libres_aux.push_back(celda_valoracion(i, j, defaultCellValue(i, j,
                    freeCells, nCellsWidth, nCellsHeight, mapWidth, mapHeight, obstacles,
                    defenses)));
        }

    // Ordenamos los elementos de forma similar a lo hecho con anterioridad
    std::list<celda_valoracion> aux_lista2(celdas_libres_aux.begin(), celdas_libres_aux.end()
        ); // Creo una lista con los elementos de celdas
    aux_lista2.sort();
                                                                                    // Ordeno
        los elementos O(n) = n    log n
    // Copiamos el contenido a las celdas originales para la insercion en el mapa de las
        defensas
    celdas_valoradas = std::vector<celda_valoracion>(aux_lista2.begin(), aux_lista2.end());
    ;

    std::vector<celda_valoracion>::iterator it;

    currentDefense++; // pasamos al segundo elemento ya que el primero ya se ha colocado (
        centro de extraccion)

    while (currentDefense != defenses.end())
    {

        placed = false;
        it = celdas_libres_aux.end();
        while (!placed && !celdas_valoradas.empty())
        {

            solution = celdas_valoradas.back();
            celdas_valoradas.pop_back();
            if (funcion_factibilidad(defenses, *(*currentDefense), obstacles, mapHeight,
                cellWidth, cellHeight, mapWidth, solution.row, solution.col, freeCells))
            {
                placed = true;
                (*currentDefense)->position = cellCenterToPosition(solution.row, solution.col
                    , cellWidth, cellHeight);
                it--;
```

```
                celdas_libres_aux.erase(it);
            }
        }
        celdas_valoradas = celdas_libres_aux;
        currentDefense++;
    }
}
```

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

   Las características que tiene este algoritmo son: 1)Sigue la extructura de un algoritmo devorador (recibe un conjunto a tratar, bucle while para el tratamiento del mismo, ...)

   2) Funcion de seleccion: la cual esta intrinseca en el codigo de lista (C.front()), ya que esta nos devolveria el elemento en el frente de la lista, la cual ha sido previamente ordenada, teniendo en ella la celda de mayor valoracion.

   3) Funcion de Factibilidad: La cual nos permite saber si es posible colocar en esa posicion un elemento (defensa/centro de extraccion)

   4)Funcion Objetivo: La misma funcion de factiblidad nos permite obtener el candidato mas prometedor ya que si esta es correcta se insertará en el conjunto solucion.

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.

```
float defaultCellValue(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;

    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;
}

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

    float cellWidth = mapWidth / nCellsWidth; //anchura de la celula
    float cellHeight = mapHeight / nCellsHeight;//altura de la celula
    //Aplicamos el criterio ==> cuanto mas cerca de un obstaculo mejor
    //Usamos el tipo vector 3 para una comparativa (si esta vacia o no de forma mas sencilla)
    Vector3 t_posicion = cellCenterToPosition(row,col,cellWidth,cellHeight);
    float value = 0;
    for(auto i: obstacles){
        value+=i->position.subtract(t_posicion).length();
        //value+=_distance(t_position,i.position);
        //me da fallos a pesar de que esto representa lo mismo que arriba
    }

    //el que tenga menor valor tendra mas obstaculos cerca ==> mayor puntuacion
    //por tanto lo invierto para tener mayor puntuacion
    return 1/value; // implemente aqui la funcin que asigna valores a las celdas
}
```

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 maxAttemps = 1000;
    std::vector<celda_valoracion> celdas_valoradas;
    int i, j, k;
    float aux;
    celda_valoracion auxCell;

    /* ------------------------------------------------------------------------ */
    /*                  ASINGACION DEL CENTRO DE EXTRACCION (DEF 0)             */
    /* ------------------------------------------------------------------------ */

    // 1) Obtenemos que celdas estan libres tras la colocacion de los obstaculos
    for (int i = 0; i < nCellsHeight; i++)
    {

        for (int j = 0; j < nCellsWidth; j++)
        {
            if (freeCells[i][j] != false)
            {

                celdas_valoradas.push_back(celda_valoracion(i, j, cellValueExtractionCenter(i
                    , j, freeCells, nCellsWidth, nCellsHeight, mapWidth, mapHeight, obstacles
                    )));
            }
        }
    }

    // Ordenacion de las celdas valoradas aplicando los constructores de vector y lista

    std::list<celda_valoracion> aux_lista(celdas_valoradas.begin(), celdas_valoradas.end());
        // Creo una lista con los elementos de celdas
    aux_lista.sort();
        // Ordeno los elementos O(n) = n    log n
    celdas_valoradas = std::vector<celda_valoracion>(aux_lista.begin(), aux_lista.end());
        // los copio a la lista

    bool placed = false;
    celda_valoracion solution;
    List<Defense *>::iterator currentDefense = defenses.begin();

    // Algoritmo devorador para centro de extraccion
    while (!placed && !celdas_valoradas.empty())
    {
        solution = celdas_valoradas.back();
        celdas_valoradas.pop_back();
        if (funcion_factibilidad(defenses, *(*currentDefense), obstacles, mapHeight,
                                 cellWidth, cellHeight, mapWidth, solution.row, solution.col,
                                     freeCells))
        {
            placed = true;
            freeCells[solution.row][solution.col] = false;
            (*currentDefense)->position = cellCenterToPosition(solution.row, solution.col,
                cellWidth, cellHeight);
        }
    }
    // Copia de los candidatos

    std::vector<celda_valoracion> celdas_libres_aux;
```

```
    // Insertamos las celdas aun libres en el vector
    for (i = 0; i < nCellsHeight; i++)
        for (j = 0; j < nCellsWidth; j++)
        {
            if (freeCells[i][j])
                celdas_libres_aux.push_back(celda_valoracion(i, j, defaultCellValue(i, j,
                    freeCells, nCellsWidth, nCellsHeight, mapWidth, mapHeight, obstacles,
                    defenses)));
        }

    // Ordenamos los elementos de forma similar a lo hecho con anterioridad
    std::list<celda_valoracion> aux_lista2(celdas_libres_aux.begin(), celdas_libres_aux.end()
        ); // Creo una lista con los elementos de celdas
    aux_lista2.sort();
                                                                                // Ordeno
        los elementos O(n) = n    log n
    // Copiamos el contenido a las celdas originales para la insercion en el mapa de las
        defensas
    celdas_valoradas = std::vector<celda_valoracion>(aux_lista2.begin(), aux_lista2.end());
    ;

    std::vector<celda_valoracion>::iterator it;

    currentDefense++; // pasamos al segundo elemento ya que el primero ya se ha colocado (
        centro de extraccion)

    while (currentDefense != defenses.end())
    {

        placed = false;
        it = celdas_libres_aux.end();
        while (!placed && !celdas_valoradas.empty())
        {

            solution = celdas_valoradas.back();
            celdas_valoradas.pop_back();
            if (funcion_factibilidad(defenses, *(*currentDefense), obstacles, mapHeight,
                cellWidth, cellHeight, mapWidth, solution.row, solution.col, freeCells))
            {
                placed = true;
                (*currentDefense)->position = cellCenterToPosition(solution.row, solution.col
                    , cellWidth, cellHeight);
                it--;
                celdas_libres_aux.erase(it);
            }
        }
        celdas_valoradas = celdas_libres_aux;
        currentDefense++;
    }
}
```

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.