

# Práctica 1. Algoritmos devoradores

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

3 de diciembre de 2021

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 cercanía con respecto a los obstáculos distribuidos en el mapa, es decir, cuanto mas cerca a los obstáculos este la celda dada mejor puntuación. La puntuación se calcula en base a la suma de las distancias que separan a posición creada a partir de un objeto tipo Vector3 y la función cellCenterToPosition, con esto podremos calcular la distancia con respecto a la lista de obstáculos que se reciben como parametro. Por tanto la puntuación será el inverso de la suma de las distancias. Anotación: El código de esta función está "incrustado" en el ejercicio nº 5.

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

```
/*
FUNCION DE FACTIBILIDAD
Dise e una funcion de factibilidad explicita y descr bala a continuacion.
Entiendo que la funcion sirve para comprobar si es posible que se coloque algo en una
celda
(una defensa en x lugar)
Parametros:
Row-> entero que representa una fila en la matriz del mapa
col-> entero que representa una col en la matriz del mapa
obstacles-> Recibe la lista de defensas colocadas en el mapa para iterar y comprobar
que no coincidan
Defenses-> Recibe la lista de obstacles en el mapa para iterar y comprobar que no
coincidan
freeCells -> Matriz con el numero de celdas libres
mapHeight -> altura del mapa (eje z)
mapWidth -> anchura del mapa (eje x)
int nCellsWidth-> numero de celdas en total del eje x
int nCellsHeight-> numero de celdas en total del eje z
Defense d ==> Recibe una referencia a la defensa a colocar en la celda
PREGUNTAR QUE SIGNIFICA QUE SEA UNA FUNCION DE FACTIBILIDAD EXPLICITA
*/
bool funcion_factibilidad(int row, int col, List<Object*> obstacles, List<Defense*>
defenses,
bool **freeCells, float mapHeight, float mapWidth, int nCellsWidth, int nCellsHeight, const
Defense& d){
//definicion de variables
float cellWidth = mapWidth / nCellsWidth; //anchura de la celula
float cellHeight = mapHeight / nCellsHeight; //altura de la celula
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 ||
```

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

Figura 1: Estrategia devoradora para la mina

```

        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
    }
    else{//Si no cumple el primer requisito no sera necesario continuar, en cambio , si
        lo cumple entonces:
        //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((d.radio + i->radio)>_distance(p,i->position))
                token = false;
        }
        if(token){//si ya se ha detectado que no es posible colocarla en un obstaculo
            //para que comprobar las defensas
            //Se comprobara de forma similar a los obstaculos con las defensas
            for (auto i: defenses){
                if((d.radio + i->radio)>_distance(p,i->position))
                    token = false;
            }
        }
        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.

```

//Para este algoritmo se tiene que tener en cuenta todas las celdas y la de mejor valoracion
//sera el centro de extracci n
//Asigna valor a las celdas (mejor valor, usando la funcion cellvalue)

//Estructura auxiliar para realizar la valoracion (se podria usar un map pero con esto
podemos llamar al sort de la lista)
struct defensa_valoracion{
    Defense* d;
    float valoracion;
    defensa_valoracion(Defense *d_,float v=-1):d(d_),valoracion(v){}
    bool operator <(defensa_valoracion & b){//para la ordenacion de la lista
        return valoracion < b.valoracion;
    }
};

std::list<defensa_valoracion> asignar_valoracion_celda(int row, int col, bool** freeCells,
    int nCellsWidth, int nCellsHeight
    , float mapWidth, float mapHeight, List<Object*> obstacles, List<Defense*> defenses){
    std::list<defensa_valoracion> res;
    for(auto i: defenses){
        res.push_back(defensa_valoracion(i,
            cellValue(row,col,freeCells,nCellsWidth,nCellsHeight,mapWidth,mapHeight,obstacles,
                defenses)));//insertamos todas las valoraciones
    }
    return res;
}

//Algoritmo Voraz para el centro (al devolver la mejor defensa para el centro de extr solo
devuelve un objeto defensa)
Defense* voraz_centro(int row, int col, List<Object*> obstacles, List<Defense*> defenses,
    float mapHeight, float mapWidth,int nCellsWidth, int nCellsHeight){
    std::list<defensa_valoracion> C = asignar_valoracion(defenses);//obtenemos la lista de
    std::list<Defense*> S;
    Defense* p;
    //inicio del algoritmo
}

```

```

C.sort(); //ordena de menor a mayor por tanto obtenemos el frente

while(C.size()>0){//tendremos que vaciar la lista en todo caso para poder comprobar su
    factibilidad
    p = C.front().d;
    C.pop_front(); //podamos el frente

    if(funcion_factibilidad(row,col,obstacles,defenses,mapHeight,
        mapWidth,nCellsWidth,nCellsHeight,p)){
        S.push_back(p);
    }
}
return S.front();//devuelve el frente ya que tiene la mejor posicion defensiva, es decir,
    el centro
}

```

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 estructura 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 factibilidad 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.

Para la definicion de esta funcion compruebo como de cerca se encuentra este del centro de extracción cuanto mas cerca mejor. Tendra los siguientes parametros: · Defense\* d ==¿ parametro de entrada al cual tendremos que evaluar. · lista de defenses ==¿ Tendra la lista de todas la defensas, para obtener el centro de extracción.

```

float defense_value(Defense* d, List<Defense*> defenses)
{
    float value=-1; //Por si acaso es la defensa[0], el centro devolvera -1
    if(d != *defenses.begin()){
        /* ----- CRITERIO: cuanto mas cerca del centro de extraccion mejor -----
        */
        Defense* centro_temp = *defenses.begin();
        value = _distance(d->position,centro_temp->position);
    }
    return value ;
}

```

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.

```

// Falla en la colocacion del algoritmo voraz
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;
    defenses = voraz_defensas(obstacles,defenses,mapHeight,mapWidth, nCellsWidth,nCellsHeight
    );
}

```

```

Defense * centro_temp = voraz_centro(obstacles, defenses, mapHeight, mapWidth, nCellsWidth,
nCellsHeight);
int maxAttempts = 1000;
List<Defense*>::iterator currentDefense = defenses.begin();

while(currentDefense != defenses.end() && maxAttempts > 0) {
    //para no complicarnos lo hacemos con parametros
    int row = ((int)(_RAND2(nCellsWidth))) * cellWidth + cellWidth * 0.5f;
    int col = ((int)(_RAND2(nCellsHeight))) * cellHeight + cellHeight * 0.5f;

    //FALTA POR IMPLEMENTAR EL VORAZ DEL CENTRO DE EXTRACCION (DEFENSA 0)

    if(funcion_factibilidad(row,col,obstacles,defenses,mapHeight,
mapWidth,nCellsWidth,nCellsHeight,(*currentDefense)){
        if (currentDefense == defenses.begin())
        {
            (*currentDefense)->position.x = centro_temp->position.x;
            (*currentDefense)->position.y = centro_temp->position.y;
            (*currentDefense)->position.z = 0;

        }
        else
        {
            (*currentDefense)->position.x = row;
            (*currentDefense)->position.y = col;
            (*currentDefense)->position.z = 0;

        }

        ++currentDefense;
    }

}

#ifdef PRINT_DEFENSE_STRATEGY

float** cellValues = new float* [nCellsHeight];
for(int i = 0; i < nCellsHeight; ++i) {
    cellValues[i] = new float[nCellsWidth];
    for(int j = 0; j < nCellsWidth; ++j) {
        cellValues[i][j] = ((int)(cellValue(i, j))) % 256;
    }
}
dPrintMap("strategy.ppm", nCellsHeight, nCellsWidth, cellHeight, cellWidth, freeCells
, cellValues, std::list<Defense*>(), true);

for(int i = 0; i < nCellsHeight ; ++i)
    delete [] cellValues[i];
delete [] cellValues;
cellValues = NULL;

#endif
}

```

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.