Práctica 1. Algoritmos devoradores

Moises Guerrero Lopez
moises.guerrerolopez@alum.uca.es
Teléfono: xxxxxxxx

November 17, 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.

   Estas funciones están implementadas de tal forma que cuanto menor sea el valor de la celda mejor posicionada esta, de esta forma la celda con valor 0 sera la mejor posible y con el valor máximo que puede alcanzar el tipo float sera la peor celda posible. Mi función para colocar el centro de extracción de minerales comienza aplicando el peor valor posible a las casilla que se encuentran en los bordes de la matriz, por aquí salen los UCOs, luego solo me interesa que se coloque ahí si no hay otro lugar posible. Después de esto calculamos la casilla central de la matriz, dividiendo el numero de casillas de la altura y de la anchura del mapa, si alguno de estos valores es par ocurre que hay dos posibles centros, en este caso tenemos en cuenta los dos centros posibles. Es cierto que se podría haber elegido dividiendo entre 2 la longitud de la altura y la anchura y después convertir estos valores a los valores enteros de una fila y columna con el método proporcionado, pero encontré la otra forma algo más precisa cuando estaba realizando pruebas.Justo después sumamos las distancias desde nuestra celda hasta los obstáculos, pero solo aquellos que estén a una distancia lo suficientemente cerca, de esta forma discriminamos las celdas de alrededor del centro que estén más alejadas de los obstáculos. Haciendo pruebas he visto que la distancia mas adecuada es de alrededor de 40, apenas hay diferencia si variamos un poco este valor, pero si que hay diferencia si es demasiado pequeña o demasiado grande y tras acabar con los obstáculos miramos que distancia a los posibles centros que tenemos es mas corta y la sumamos a la variable donde hemos almacenado las distancias a los obstáculos, si la del segundo es menor guardamos su fila y columna para más adelante, si no lo es, guardamos la fila y columna del primer centro. Ahora hacemos uso la fila y columna de nuestro centro para penalizar las celdas que se encuentren mas alejadas del centro, dependiendo si la celda esta a la derecha o izquierda según la columna y si esta encima o debajo según la fila, la penalización consiste en restar nuestra fila o columna al indice(numero de fila o columna) máximo de la fila y columna si esta a la derecha para la columna o debajo y si esta a la izquierda o encima se resta a la fila o columna del centro nuestra fila o columna. Una vez acabado esto se devuelve la variable en la que hemos estado sumando las distancias y aplicando las penalizaciones.

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

   Para la función de factibilidad recibe la fila y la columna a comprobar, el numero de celdas y columnas que hay, la longitud la altura y de la anchura, la lista de obstáculos, la lista de defensas y el id de la defensa a colocar. La función primero comprueba que el id recibido existe en la lista de defensas y si no se encuentra devuelve false,esta parte no es realmente necesaria si se vigila que parámetro se le envía a la función, pero mejor prevenir que curar.Después comprobamos que la defensa que vamos a colocar no se sale del mapa, para ello sumamos el radio de la defensa a las posiciones de la celda correspondiente a la fila y columna que enviamos a la función, si es mayor o igual que la longitud de la anchura y la altura del mapa significa que se ha salido del mapa, ahora hacemos lo mismo pero restando en lugar de sumar, si el resultado es menor o igual que cero significa también que se ha salido del mapa, si cumple alguna de estas condiciones devuelve false. El

```
//Cuanto menor sea el valor devuelto mejor
float cellValue_centro(int row,int col,int nCellsWidth, int nCellsHeight,List<Object*> obstacles, List<Defense*> defenses,
  float cellWidth,float cellHeight){
```

Figure 1: Estrategia devoradora para la mina

siguiente paso es comprobar que no se sobrepone en una zona ya ocupada por una defensa u obstáculo, son básicamente iguales excepto que en las defensas debemos comprobar solo las defensas que hemos colocado las que tengan un id menor al que recibimos de la función. Para comprobar que no choca con nada calculamos las distancia desde la posición correspondiente a la fila y columna que recibimos en la función con la posición del objecto que queremos comprobar, si esta distancia es menor que la suma de los radios de la defensa que queremos colocar y la del objeto que estamos comprobando significa que están colisionando y devolverá false, sino pues continua. Después de comprobar todos los obstáculos no hay nada más que comprobar y si se ha llegado hasta este punto devolvemos true indicando que la celda es factible

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.

```
// sustituya este codigo por su respuesta
void 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;

        float greatest=std::numeric_limits<float>::max();
        int iaux,jaux;
        //Colocamos el centro de recursos
        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] = cellValue_centro(i,j,nCellsWidth,nCellsHeight,
                            obstacles,defenses,cellWidth,cellHeight);
                        if(cellValues[i][j]<greatest && factible(i,j,nCellsWidth,nCellsHeight
                            ,mapWidth,mapHeight,obstacles,defenses,0))
                            {
                                    iaux=i;
                                    jaux=j;
                                    greatest=cellValues[i][j];
                            }
                }
        }

        Vector3 centro_de_recoleccion= cellCenterToPosition(iaux,jaux,cellWidth,cellHeight);
        List<Defense*>::iterator currentDefense=defenses.begin();
        (*currentDefense)->position.x=centro_de_recoleccion.x;
        (*currentDefense)->position.y=centro_de_recoleccion.y;
        (*currentDefense)->position.z=0;

        //Aqui va el codigo para el resto de las defensas
}
```

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

   Se trata de un algoritmo voraz porque se trata de un algoritmo cuyo objetivo es optimizar un problema. Siguiendo el esquema de las transparencias, el problema acabara cuando se coloquen todas las defensas, este algoritmo tiene dos algoritmos voraces, uno para el centro de recursos y otro para el resto de defensas. Ambos siguen el mismo esquema, primero seleccionamos la defensa y después a cada celda se le asigna un determinado valor, comprobamos si es menor que que el anterior valor mas bajo,esta sería nuestra función objetivo, y si es una posición factible y si lo es se asigna este nuevo valor como el mas bajo y guardamos su fila y columna. Cuando calculamos todos los valores de todas las celdas asignamos la posición de la fila y la columna que tiene menor valor de todas las celdas y se lo asignamos a la defensa que estamos comprobando. Cuando se acaban las defensas ya hemos encontrado todas las soluciones y el conjunto de candidatos queda vació. Nuestro algoritmo voraz no tendría función de selección ya que tenemos que asignar un valor a todas las defensas, el conjunto de candidatos y candidatos seleccionados sería el mismo y tampoco tiene función solución por la misma razón que la de la función de selección

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.

Esta función no es muy diferente de la que usamos para designar la posición del centro de recursos. Primero asignamos el valor mayor posible de float a los bordes de la matriz, para que solo se coloque aquí si no hay otra posición posible, después obtenemos la posición del centro de recursos y calculamos la distancia desde la posición de la celda que estamos comprobando hasta la del centro de recursos y se lo asignamos a una variable, en el siguiente paso restamos ha esta variable la distancia desde nuestra posición hasta los obstáculos, siempre y cuando la distancia entre estos dos no sea mayor que 20. Al parecer hay un poco de mejora si se coloca esa condición. Y finalmente sumamos a nuestra variable la distancia desde nuestra celda al centro del mapa. Haciendo esto ultimo noto una mejora considerable en algunos mapas y apenas se ven afectados otros mapas. Pensé en añadir la penalización por esta alejado del centro, pero los resultados se en algo mermados con esa opción, así que preferí eliminarla. Y por ultimo devolvemos esta variable.

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.

```
// sustituya este codigo por su respuesta
void DEF_LIB_EXPORTED placeDefenses(bool** freeCells, int nCellsWidth, int nCellsHeight,
    float mapWidth, float mapHeight,std::list<Object*> obstacles, std::list<Defense*>
    defenses) {

        //Aqui estaria el codigo del ejercicio 3

        int best_row;
        int best_col;

        Vector3 centro_mapa;
        centro_mapa.x=mapWidth/2;
        centro_mapa.y=mapHeight/2;
        centro_mapa.z=0;

        //Colocamos el resto de defensas

        for(int k=1;k<defenses.size();++k)
        {
                ++currentDefense;
                greatest=std::numeric_limits<float>::max();
                for(int i=0;i<nCellsWidth;++i)
                {
                        for(int j=0;j<nCellsHeight;++j)
                        {
                                cellValues[i][j]=cellValue(i,j,freeCells,nCellsWidth,
                                    nCellsHeight,mapWidth,mapHeight,obstacles,defenses,
                                    centro_mapa);
                                if(cellValues[i][j]<=greatest && factible(i,j,nCellsWidth,
                                    nCellsHeight,mapWidth,mapHeight,obstacles,defenses,k))
                                {
                                        best_row=i;
                                        best_col=j;
                                        greatest=cellValues[i][j];
                                }
                        }
                }

                Vector3 best=cellCenterToPosition(best_row,best_col,cellWidth,cellHeight);
                (*currentDefense)->position.x=best.x;
                (*currentDefense)->position.y=best.y;
                (*currentDefense)->position.z=0;
        }

        #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

}

float cellValue(int row, int col, bool** freeCells, int nCellsWidth, int nCellsHeight, float
    mapWidth, float mapHeight, List<Object*> obstacles, List<Defense*> defenses,Vector3
    centro_mapa) {

        float cellWidth = mapWidth / nCellsWidth;
        float cellHeight = mapHeight / nCellsHeight;
        float maximus=std::numeric_limits<float>::max();
        if(row==0 || row==nCellsHeight-1 || col==0 || col==nCellsWidth-1) return maximus;
        else
        {
                List<Defense*>::iterator centro_rec=defenses.begin();
                int rec_fila,rec_col;
                positionToCell((*centro_rec)->position,rec_fila,rec_col,cellWidth,cellHeight)
                    ;
                Vector3 celda=cellCenterToPosition(row,col,cellWidth,cellHeight);
                float distance= _distance((*centro_rec)->position,celda);

                List<Object*>::iterator obs=obstacles.begin();
                float d_aux;

                while(obs!=obstacles.end())
                {
                        d_aux=_distance((*obs)->position,celda);
                        if(d_aux<=20)distance-=d_aux;
                        ++obs;
                }

                distance+=_distance(celda,centro_mapa);
                return distance;
        }
}

float cellValue_centro(int row,int col,int nCellsWidth, int nCellsHeight,List<Object*>
    obstacles, List<Defense*> defenses,float cellWidth,float cellHeight){

        float maximus=std::numeric_limits<float>::max();
        float d=0;

        if(row==0 || col==0 || row==nCellsHeight-1 || col==nCellsWidth-1)return maximus;//No
            queremos la frontera de la matriz
        else
        {
                //Encontramos el centro de la matriz
                //puede haber 2, cuando resto==0
                int fila1=-1;
                int fila2=-1;
                int col1=-1;
                int col2=-1;
                if((nCellsWidth-1)%2!=0)
                {
```

```
        col1=(nCellsWidth-1)/2;
        col2=(nCellsWidth-1)/2+1;
}
else
{
        col1=   (nCellsWidth-1)/2;
}

if((nCellsHeight-1)%2!=0)
{
        fila1=(nCellsHeight-1)/2;
        fila2=(nCellsHeight-1)/2+1;
}
else
{
        fila1=  (nCellsHeight-1)/2;
}

float dist1;
float dist2;

int iaux;
int jaux;
Vector3 cell=cellCenterToPosition(row,col,cellWidth,cellHeight);

//suma de las distancia hasta los obstaculos
List<Object*>::iterator checkObstacle=obstacles.begin();
float d_aux;
while(checkObstacle!=obstacles.end())
{
        d_aux= _distance((*checkObstacle)->position,cell);
        if(d_aux<=39)d+= d_aux;//No nos interesan los que estan lejos,39
            parece correcto
        ++checkObstacle;
}

Vector3 center1=cellCenterToPosition(fila1,col1,cellWidth,cellHeight);
//distancia hasta el centro 1
dist1= _distance(cell,center1);

//si hay dos centros;
if(col2!=-1 && fila2!=-1)
{
        Vector3 center2=cellCenterToPosition(fila2,col2,cellWidth,cellHeight)
            ;

        //distancia hasta el centro 2 si hubiera
        dist2= _distance(cell,center2);


        if(dist2>dist1)d+=dist1;
        else
        {
                d+=dist2;
                fila1=fila2;
                col1=col2;
        }
}

//si solo hay un centro
else
{
        d+=dist1;
}

//calculamos las penalizaciones por estar cerca del border
//tanto para las filas como para las columnas


if(fila1!=row)
{
```

```
                        if(row<fila1)
                        {
                                d+=(float)(nCellsHeight-1)-(float)fila1;
                        }
                        else
                        {
                                d+=(float)row-(float)fila1;
                        }
                }

                if(col1!=col)
                {
                        if(col<col1)
                        {
                                d+=(float)(nCellsWidth-1)-(float)col1;
                        }
                        else
                        {
                                d+=(float)col-(float)fila1;
                        }
                }
        }
        return d;
}

// Devuelve la posici n en el mapa del centro de la celda (i,j)
// i - fila
// j - columna
// cellWidth - ancho de las celdas
// cellHeight - alto de las celdas
Vector3 cellCenterToPosition(int i, int j, float cellWidth, float cellHeight)
{
        return Vector3((j * cellWidth) + cellWidth * 0.5f, (i * cellHeight) + cellHeight *
            0.5f, 0);
}


// Devuelve la celda a la que corresponde una posici n en el mapa
// pos - posici n que se quiere convertir
// i_out - fila a la que corresponde la posici n pos (resultado)
// j_out - columna a la que corresponde la posici n pos (resultado)
// cellWidth - ancho de las celdas
// cellHeight - alto de las celdas
void positionToCell(const Vector3 pos, int &i_out, int &j_out, float cellWidth, float
    cellHeight)
{
        i_out = (int)(pos.y * 1.0f/cellHeight); j_out = (int)(pos.x * 1.0f/cellWidth);
}


bool factible(int row,int col,int nCellsWidth,int nCellsHeight,float mapWidth,float mapHeight
    ,List<Object*> obstacles,List<Defense*> defenses,int id)// identificador de la defensa
    para la lista de defensas
{
        //Primero buscamos la defensa con id
        List<Defense*>::iterator currentDefense = defenses.begin();
        bool found=false;
        while(currentDefense!=defenses.end() && !found)
        {
                if((*currentDefense)->id==id)
                {
                        found=true;
                }
                else
                {
                        ++currentDefense;
                }
        }

        if(!found)return false;
        else
```

```
        {
                //Usar las funciones de distancia
                float cellWidth = mapWidth / nCellsWidth;
                float cellHeight = mapHeight / nCellsHeight;

                //Comprobamos que no se sale del mapa
                Vector3 celda=cellCenterToPosition(row,col,cellWidth,cellHeight);
                float positivo_x=celda.x+(*currentDefense)->radio;
                float negativo_x=celda.x-(*currentDefense)->radio;
                float positivo_y=celda.y+(*currentDefense)->radio;
                float negativo_y=celda.y-(*currentDefense)->radio;

                if(positivo_x>=mapWidth || negativo_x<=0) return false;
                if(positivo_y>=mapHeight || negativo_y<=0) return false;


                int row2=0;
                int col2=0;;


                List<Defense*>::iterator checkDefense = defenses.begin();
                while((checkDefense!=defenses.end()) && id!=(*checkDefense)->id )
                {
                        float dist_def=_distance(celda,(*checkDefense)->position);
                        if(dist_def<=(*checkDefense)->radio+(*currentDefense)->radio)
                {
                        return false;
                }
                        ++checkDefense;
                }

                //Ni con ningun obstaculos
                //Todos los obstaculos de esta lista estan colocados
                List<Object*>::iterator checkObstacle=obstacles.begin();
                while(checkObstacle!=obstacles.end())
                {
                        float dist= _distance(celda,(*checkObstacle)->position);
                        if((*checkObstacle)->radio+(*currentDefense)->radio>=dist)
                        {
                                return false;
                        }
                        ++checkObstacle;
                }

                return true;
        }
}
```

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.