Práctica 1. Algoritmos devoradores

Alejandro Segovia Gallardo
alejandro.segoviagallardo@alum.uca.es
Teléfono: 608842858
NIF: 32083695Y

17 de noviembre de 2017

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 que he seguido se ha basado en la evaluación de la fila y columna que reciben la función cellValue por parámetros y su puntuación en funcion de su distancia al centro del mapa concreto(a mayor distancia menor puntuación) y aumentando su puntuacion en función de su cercania a los obstaculos que puedan existir en el ya mencionado mapa.

```
float cellValue(int row, int col, bool** freeCells, int nCellsWidth,
                int nCellsHeight, float mapWidth, float mapHeight,
                float cellWidth, float cellHeight,
                List<Object*> obstacles, List<Defense*> defenses)
{
    float proximidad = 0;
    Vector3 dst,obj;
    dst.x = (nCellsWidth/2) * cellWidth + (cellWidth/2) - row * cellWidth +
              (cellWidth/2);
    dst.y = (nCellsHeight/2) * cellHeight + (cellHeight/2) - col * cellHeight +
            (cellHeight/2);

    for(std::list<Object*>::const_iterator it = obstacles.begin();
        it != obstacles.end();it++)
        {
        obj.x = (*it)->position.x - row * cellWidth + cellWidth/2;
        obj.y = (*it)->position.y - col * cellHeight + cellHeight/2;
        if((*it)->radio * 1.5 < obj.length())
          proximidad += 1;
        if((*it)->radio * 2 < obj.length())
          proximidad += 0.5;
        if((*it)->radio * 1.1 < obj.length())
          proximidad += 0.5;
        }
    return  std::max(mapWidth,mapHeight) - dst.length() + proximidad;
}
```

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

   La función de factibilidad de mi algoritmo está basada en que dada una posición del mapa concreta dada observar si pertenece a los limites del mapa dado o si presenta algún conflicto de posición con algun obstaculo/defensa colocada ya en el tablero. En el caso de que no peresente alguna incidencia de las ya mencionadas, retornara un valor booleano verdadero, en caso contrario, será falso

```
bool factible(float x, float y, float rDef, float mapWidth,
                  float mapHeight, float cellWidth,
                  float cellHeight,std::list<Object*> obstacles,
                  std::list<Defense*> defenses)
{

    if (x - rDef  < 0 || x + rDef > mapWidth ||y - rDef< 0 ||
        y + rDef > mapHeight)
          return false;
```

```
    Vector3 dst;
    for(std::list<Object*>::const_iterator obs = obstacles.begin();
        obs != obstacles.end();obs++)
    {
      dst.x = (*obs)->position.x - x;
      dst.y = (*obs)->position.y - y;
      if(dst.length() - (*obs)->radio - rDef <= 0)
        return false;
    }

    for(std::list<Defense*>::const_iterator defensaconst = defenses.begin();
        defensaconst != defenses.end();defensaconst++)
    {
      dst.x = (*defensaconst)->position.x - x;
      dst.y = (*defensaconst)->position.y - y;
      if(dst.length() - (*defensaconst)->radio - rDef <= 0)
        return false;
    }

    return true;
}
```

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)
{
    int fila = 0, columna = 0;
    float x = 0, y = 0;
    float** cellValues = new float*[nCellsHeight];

                for(int i = 0; i < nCellsHeight; ++i)
            cellValues[i] = new float[nCellsWidth];

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

    for(int i = 0; i < nCellsHeight; i++)
                for(int j = 0; j < nCellsWidth; j++)
                        cellValues[i][j] = cellValue(i, j, freeCells, nCellsWidth,
                            nCellsHeight,
                                mapWidth,mapHeight,cellWidth, cellHeight, obstacles, defenses
                                    );

    int maxAttemps = 1000;

    List<Defense*>::iterator currentDefense = defenses.begin();

    while(currentDefense == defenses.begin() and maxAttemps > 0)
    {
        seleccion(cellValues, nCellsWidth, nCellsHeight, &fila, &columna);

        x = fila*cellWidth + cellWidth/2;
        y = columna*cellHeight + cellHeight/2;

        if(factible(x, y, (*currentDefense)->radio, mapWidth, mapHeight,
           cellWidth,cellHeight,obstacles, defenses))
        {
            (*currentDefense)->position.x = x;
            (*currentDefense)->position.y = y;
            (*currentDefense)->position.z = 0;

            cellValueDefensas(cellValues, nCellsWidth, nCellsHeight,cellWidth,
              cellHeight,mapWidth,mapHeight,defenses);

            currentDefense++;
```

```
        }
        maxAttemps--;
    }
    //Resto de codigo no perteneciente al centro de extraccion.
```

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

   Mi algoritmo pertenece al esquema de los algoritmos voraces ya que peresenta todos los elementos necesarios en su estructura, presenta un bucle, con una condición de parada basada en dos factores, un numero de intentos y también una condición booleana, que en la estructura voraz equivaldria a la solución. Presenta tambien una función de seleccion, que una vez extraida elimina de las siguiente iteraciones esa solucion, asignandole un valor inalcanzable, ya sea factible o no, y por último tambien presenta una funcion de factibilidad que observa si la mejor seleccion encontrada en cada iteracion puede incluirse en nuestra solución del problema.

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 he intentado puntuar con mayor valor las celdas mas cercanas en linea recta al centro de extracción ya colocado, de manera que estaría anillando las posiciones cercanas al centro, tratando de conseguir que las defensas rodeen el ya mencionado centro.

```
void cellValueDefensas(float** cellValue, int  nCellsWidth, int nCellsHeight,
              float cellWidth, float cellHeight,float mapWidth, float mapHeight,
              std::list<Defense*> defenses)
{
    List<Defense*>::const_iterator centroExtraccion = defenses.begin();
    Vector3 dst;
    for(int i = 0; i < nCellsWidth; ++i)
        for(int j = 0; j < nCellsHeight; ++j)
        {
          dst.x = i*cellWidth + cellWidth/2 - (*centroExtraccion)->position.x;
          dst.y = j*cellHeight + cellHeight/2 - (*centroExtraccion)->position.y;
          cellValue[i][j] = std::max(mapWidth, mapHeight) - dst.length();
        }
}
```

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)
{
    //Continuacion directa del codigo del ejercicio 3.

    maxAttemps = 1000 * std::max(nCellsWidth,nCellsHeight);

    while(currentDefense != defenses.end() && maxAttemps > 0)
    {
        seleccion(cellValues, nCellsWidth, nCellsHeight, &fila, &columna);

        x = fila*cellWidth + cellWidth/2;
        y = columna*cellHeight + cellHeight/2;

        if(factible(x, y, (*currentDefense)->radio, mapWidth, mapHeight,
        cellWidth,cellHeight, obstacles, defenses))
        {
            (*currentDefense)->position.x = x;
            (*currentDefense)->position.y = y;
            (*currentDefense)->position.z = 0;
```

```
            currentDefense++;
        }
        maxAttemps--;
    }

}
```

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.