Práctica 1. Algoritmos devoradores

Sergio Rodríguez Vázquez
sergio.rodriguezvazquez@alum.uca.es
Teléfono: +634471641
NIF: 77171745Y

8 de noviembre de 2020

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.

   En mi caso, la función diseñada para darle un valor a los candidatos (las celdas) es la siguiente:

   a) Creo la lista de Candidatos mediante la función cellValue, esta función devuelve un valor tipo float que se calcula midiendo la distancia que existe entre la posición del candidato y la esquina más cercana a este.

   b) Una vez tengo la lista de Candidatos completa, la ordeno de mayor a menor, siguiendo esta lógica, las posiciones más cercanas a las esquinas estarían en las primeras posiciones de la lista y lás más lejanas al final de la lista, en concreto, la posición central del mapa estará en la última posición de la lista de Candidatos

   c) A continuación, vamos recogiendo los úlltimos valores de la lista de Candidatos comprobando que sea una posición factible, cuando lo sea, colocamos la defensa y ya habríamos colocado la primera defensa en el mapa, que en nuestro caso se trata del centro de extracción de minerales.

   Como podemos observar, mi estrategia colocará la defensa en el centro dando los valores a las celdas desde las esquinas.

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

   La función de factibilidad comprueba si una posición está ocupada por otro obstaculo u otra defensa para conocer si es factible poner en la posición elegida una nueva defensa. La función de factibilidad recibirá los siguientes parámetros:

   - La lista de defensas completa
   - La lista de obstáculos completa
   - La anchura del mapa
   - La altura del mapa
   - La posición elegida para comprobar si se puede poner la defensa o no
   - El radio de la defensa que se desea colocar
   - La matriz de freeCells, la cual es una matriz booleana que devuelve true si el centro de la celda no está ocupado y false en el caso contrario
   - La anchura de una celda
   - La altura de una celda
   - La última defensa colocada en el mapa, esto nos servirá después para que el algoritmo sea más eficiente y no tengamos que recorrer la lista de defensas entera, sólo lo justo y necesario

   Pues bien, en este caso, la función factible deberá comprobar cuatro casos distintos:

a) Para empezar, debemos comprobar si la posición elegida está dentro del mapa, esto quiere decir que la defensa no puede sobresalir por el borde del mapa, esto lo podemos comprobar de forma sencilla con el siguiente condicional, comprobamos si la posicion.x + radio es mayor que la anchura del mapa, si la posicion.y + radio es mayor que la altura del mapa, si la posicion.x - radio es menor que 0 y por último si la posicion.y - radio es menor que 0.

```
if (position.x + radio > mapWidth || position.y + radio > mapHeight ||
position.x - radio < 0 || position.y - radio < 0)
```

b) Luego comprobamos que la posición elegida en la matriz freeCells devuelve true, es decir, el centro de la celda elegida está libre

```
if (!freeCells[candidaterow, candidatecol])
```

c) En tercer lugar, recorremos la lista de defensas (sólo las que estén ya colocadas), para ello comprobamos que la distancia entre el centro de la celda elegida y el centro de la defensa que estamos observando en el bucle sea menor o igual que la suma de sus radios.

```
if(_distance(position, (*defactual)->position) <= radio +
(*defactual)->radio)
```

d) En último lugar, recorremos la lista de obstáculos para ver si choca con algún obstáculo en la posición elegida, es una comprobación análoga a la de las defensas.

```
if(_distance(position, (*actual)->position) <= radio + (*actual)->radio)
```

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;
    float x, y;
    int ultima_colocada = -1;
    List<C> Candidates;

    //Hacemos lista de celdas (candidatos) para colocar el resto de defensas
    for(int i = 0; i < nCellsHeight; i++)
    {
        for(int j = 0; j < nCellsWidth; j++)
        {
            C c({cellCenterToPosition(i, j, cellWidth, cellHeight)},
            cellValue(i, j, nCellsWidth, nCellsHeight, cellWidth, cellHeight));
            Candidates.push_back(c);
        }
    }

    Candidates.sort(isMinor);
    C cfirstpromising;

    List<Defense*>::iterator currentDefense = defenses.begin();
    while(!Candidates.empty() && ultima_colocada == -1 )
    {
        cfirstpromising = Candidates.back();
        Candidates.pop_back();
        if(factible(defenses, obstacles, mapWidth, mapHeight,
        cfirstpromising.position, (*currentDefense)->radio, freeCells,
        cellWidth, cellHeight, -1))
        {
            (*currentDefense)->position.x = cfirstpromising.position.x;
            (*currentDefense)->position.y = cfirstpromising.position.y;
            (*currentDefense)->position.z = 0;
            ultima_colocada++;
```

```
        }
    }
```

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

   Las características del algoritmo voraz en este caso son:

   - Candidatos: representados por una estructura que guarda un Vector3 position de la celda y el valor de la celda (la puntuación).
   - Candidatos seleccionados: la lista de candidatos se ordena y seleccionamos los candidatos con la mejor puntuación, en nuestro caso la mejor puntuación para la primera defensa sería la mayor distancia a las esquinas, y la mejor posición para las demás defensas sería la menor distancia a la defensa extractora.
   - Solución: en nuestro caso, no tenemos función solución explícita sino que tenemos que comprobar que todas las defensas que tenemos se hayan colocado, en ese caso habríamos llegado a la solución.
   - Selección: en nuestro caso la función de selección sería escoger al mejor candidato de la lista ya ordenada
   - Función factibilidad: tenemos la función factible que comprueba si una posición es válida o no para una defensa
   - Función objetivo: la función objetivo sería la función de cellValue, que puntúa las distintas posiciones según la distancia a las esquinas, o a la defensa extractora
   - Objetivo del algoritmo: colocar en las defensas en las mejores posiciones para que aguante el mayor tiempo posible en los distintos niveles

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.

   La función cellValue en el caso de las demás defensas recibe los siguientes parámetros:

   - La fila de la posición a puntuar
   - La columna de la posición a punturar
   - La anchura de una celda
   - La altura de una celda
   - La posición de la defensa extractora

   Se puntúa mejor o peor una defensa en función de la distancia de la posición elegida a la posición de la defensa extractora de minerales.

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.

```
//Codigo anterior incluido en el ejercicio 3

    Candidates = {};

    //Hemos colocado la primera defensa
    for(int i = 0; i < nCellsHeight; i++)
    {
        for(int j = 0; j < nCellsWidth; j++)
        {
            C c({cellCenterToPosition(i, j, cellWidth, cellHeight)}, cellValue(i, j,
                cellWidth, cellHeight, (*currentDefense)->position));
            Candidates.push_back(c);
        }
    }
```

```
    ++currentDefense;
    Candidates.sort(isMinor);
    //Ponemos las demas defensas
    C cpromising;
    while(currentDefense != defenses.end() && !Candidates.empty())
    {
        cpromising = Candidates.front();
        Candidates.pop_front();
        if(factible(defenses, obstacles, mapWidth, mapHeight, cpromising.position, (*
            currentDefense)->radio, freeCells, cellWidth, cellHeight, ultima_colocada))
        {
            (*currentDefense)->position.x = cpromising.position.x;
            (*currentDefense)->position.y = cpromising.position.y;
            (*currentDefense)->position.z = 0;
            ++currentDefense;
            ultima_colocada++;
        }
    }
```

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.