Práctica 1. Algoritmos devoradores

Lucía Atienza Olmo
lucia.atienzaolmo@alum.uca.es
Teléfono: 646767043
NIF: 32089267N

14 de noviembre 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.

   A cada celda se le ha dado el valor de:

   $$\sum_{it=obstacles.begin()}^{obstacles.end()} \_distance((*it) \rightarrow position, posicionCeldaActual)$$

   el sumatorio de las distancias a los obstáculos que hay en el mapa desde la celda actual. El valor resultante luego es ponderado según la posicion de la celda en el mapa, dándole mejores puntuaciones a las posiciones centrales del mapa. Pongamos un ejemplo de esto último con un mapa de 5x5: [Figura **??**]

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

```
bool factible(int row, int col, int nCellsWidth, int nCellsHeight, float mapWidth, float
    mapHeight, std::list<Object*> obstacles,
        std::list<Defense*> defenses, Defense* currentDefense)
{
    Vector3 defensePosition = cellToPosition(row, col, mapWidth/nCellsWidth, mapHeight/
        nCellsHeight);
    float defenseRatio = currentDefense->radio;
    //no se sale del mapa
    if(defensePosition.y + defenseRatio > mapHeight || defensePosition.x + defenseRatio >
        mapWidth ||
        defensePosition.x - defenseRatio < 0 || defensePosition.y - defenseRatio < 0)
        return false;

    //no choca con obstaculos
    for(std::list<Object*>::iterator currentObstacle = obstacles.begin(); currentObstacle !=
        obstacles.end(); currentObstacle++)
        if(_distance(defensePosition, (*currentObstacle)->position) < (defenseRatio + (*
            currentObstacle)->radio))
            return false;

    //no choca con las anteriores defensas
```

Figura 1: Estrategia valores a celdas para el centro de extracción de minerales

```
    for(std::list<Defense*>::iterator defense = defenses.begin(); *defense != currentDefense;
         defense++)
    {
        if(_distance(defensePosition, (*defense)->position) < defenseRatio + (*defense)->
            radio)
            return false;
    }
    //si llega hasta aqui es porque la posicion es valida
    return true;
}
```

La función de factibilidad (`factible`) recibe:

- La fila y la columna correspondiente a la celda que vamos a evaluar.
- El número de celdas a lo ancho y a lo alto que tiene el mapa, así como el tamaño del mismo.
- Una lista de los objetos que hay en el mapa.
- Una lista con las defensas.
- La defensa a colocar.

y devuelve un booleano que será:

- `True` en caso de que la celda pueda ser ocupada por la defensa a colocar.
- `False` en caso contrario.

Al colocar la defensa en la celda debe hacerse en el centro de la misma. De calcular esta posición se encarga la función `cellToPosition`. Sabiendo ahora la posición en la que iría la defensa, debemos controlar que al sumarle el radio de la misma:

- No se salga del mapa.
- No choque con ninguno de los obstáculos que hay en el mapa (`obstacles`).
- No choque con ninguna de las defensas que ya han sido colocadas. Las defensas se colocan en orden, por lo que solo se comprueba hasta la defensa actual (las siguientes (si hubieran) aún no están colocadas en el mapa.

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;
    std::vector<std::vector<celda> > mapa(nCellsHeight, std::vector<celda>(nCellsWidth));
    std::list<Defense*> copiaDefensas = defenses;
    std::list<Defense*>::iterator centroExtraccion = defenses.begin();
    std::list<celda> listaCeldas;
    rellenaMapa(mapa, obstacles, nCellsWidth, nCellsHeight, mapWidth, mapHeight);
    rellenaLista(mapa, listaCeldas, nCellsWidth, nCellsHeight);
    bool colocado = false;
    while(!listaCeldas.empty() && !colocado)
    {
        celda mejorPosicion = extraeCelda(listaCeldas);
        if(factible(mejorPosicion.row, mejorPosicion.col, nCellsWidth, nCellsHeight, mapWidth
            , mapHeight, obstacles, defenses, *centroExtraccion))
        {
            (*centroExtraccion)->position = cellToPosition(mejorPosicion.row, mejorPosicion.
                col, cellWidth, cellHeight);
            colocado = true;
        }
    }
}
```

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

   El algoritmo usa dos estructuras para manejar los datos, una matriz y una lista. Para la matriz se ha usado el contenedor de la STL std::vector. Se llama a la funcion rellenaMapa para asignar valores a todas y cada una de las celdas del mapa. Luego se han almacenado las celdas en una lista para poder ordenarlas por valor, permitiendo que la función extraeCelda devuelva la mejor celda de todas las disponibles en coste constante. El algoritmo sigue el siguiente esquema de los algoritmos devoradores:
   $listaCeldas \leftarrow rellenarCon(mapa)$
   listaCeldas.sort()
   centroColocado ← false
   mientras $\neg centroColocado \wedge listaCeldas \neq \emptyset$
   p ← $extraccionCelda(listaCeldas)$
   si factible(p)
   centroColocado ← true

   Se distinguen los siguientes elementos característicos de los algoritmos devoradores:

   - Conjunto de candidatos: las celdas disponibles.
   - Candidatos seleccionados: la primera celda en la que sea factible colocar el centro de extracción de minerales.
   - Funcion de factibilidad: la función factible.
   - Función de selección: la función extraeCelda().
   - Solución: colocar el centro de extracción de minerales.
   - Objetivo: colocar el centro extracción de minerales en la mejor celda para maximizar el tiempo de vida del mismo.

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.

   A cada celda se le ha dado el valor de la distancia entre dicha celda y el centro de extracción de minerales. A este valor se le ha aplicado la misma ponderación descrita en el primer ejercicio de los valores de las posiciones del mapa (cuanto más centradas estén mayores valores tienen).

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)
{
    std::list<Defense*> copiaDefensas = defenses;
    copiaDefensas.pop_front();
    copiaDefensas.sort(sortDefenses);
    std::list<Defense*>::iterator currentDefense = copiaDefensas.begin();
    copiaDefensas.push_front(defenses.front()); //volvemos a introducir el centro de
        extraccion para que no de problemas en factible...
    std::list<celda> listaCeldas2;
    modificaMapa(mapa, *centroExtraccion, nCellsWidth, nCellsHeight);
    rellenaLista(mapa, listaCeldas2, nCellsWidth, nCellsHeight);
    while(currentDefense != copiaDefensas.end() && !listaCeldas2.empty())
    {
        celda mejorPosicion = extraeCelda(listaCeldas2);
        if(factible(mejorPosicion.row, mejorPosicion.col, nCellsWidth, nCellsHeight, mapWidth
            , mapHeight, obstacles, copiaDefensas, *currentDefense))
        {
            (*currentDefense)->position = cellToPosition(mejorPosicion.row, mejorPosicion.col
                , cellWidth, cellHeight);
```

```
            ++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.