Práctica 1. Algoritmos devoradores

Iván Alba Gómez
ivan.albagomez@alum.uca.es
Teléfono: XXXXXXXXX
NIF: 49302616T

12 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.

   La función dedicada a establecer un determinado valor a cada una de las celdas del terreno de batalla para el caso del centro de extracción de minerales la hemos llamado cellValue().

   - Calculamos el tamaño de las celdas.
   - Calculamos el centro del mapa.
   - Devolvemos la distancia entre el centro del mapa y el centro de la celda que estamos evaluando.

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

   - Calculamos el centro de la celda a tratar.
   - Comprobamos que la defensa no se sale de los límites del mapa.
   - Comprobamos que la defensa no colisiona con el resto de defensas ya colocadas.
   - Comprobamos que la defensa no colisiona con ningún obstáctulo.
   - Devolvemos la distancia entre el centro del mapa y el centro de la celda que estamos evaluando.

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.

```
class Cell {    // Usado en placeDefenses()
    public:
        Cell(int row, int col, float value) : row_(row), col_(col), value_(value) {}
        int row() { return row_; }
        int col() { return col_; }
        float value() { return value_; }
    private:
        int row_, col_;
        float value_;
};

Cell selection(std::list<Cell> cellList) {  // Susado en placeDefenses()
    return cellList.front();
}

float cellValue(int row, int col, bool** freeCells, int nCellsWidth, int nCellsHeight
        , float mapWidth, float mapHeight, List<Object*> obstacles, List<Defense*> defenses)
            {
    float cellWidth = mapWidth / nCellsWidth;
    float cellHeight = mapHeight / nCellsHeight;
    float middleWidth = mapWidth / 2;
    float middleHeight = mapHeight / 2;
    return _distance(Vector3(middleHeight, middleWidth, 0), cellCenterToPosition(row, col,
        cellWidth, cellHeight));
}
```

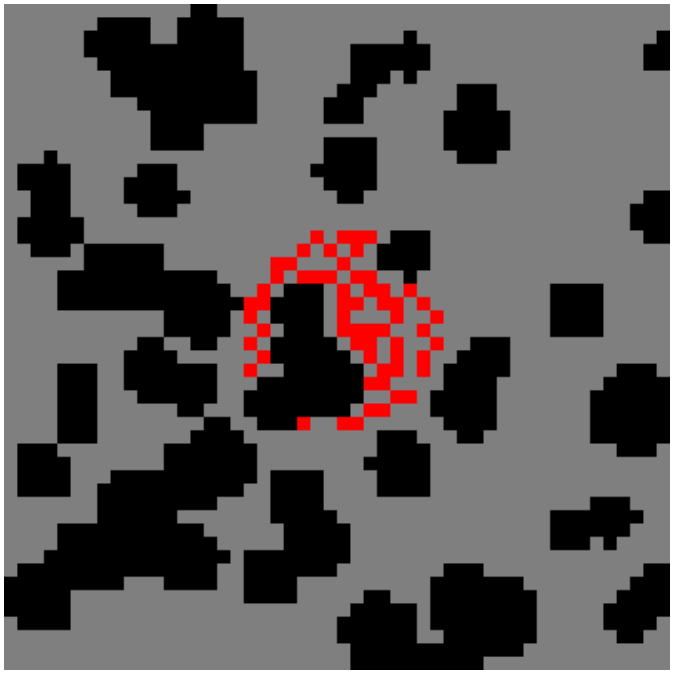

Figura 1: Estrategia devoradora para la mina

```
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 values[nCellsHeight][nCellsWidth];    // Matriz que almacena los valores de cada
        celda
    List<Cell> cellList;
    // Rellenamos la matriz de valores para colocar la primera defensa (Centro de extracci n
         de minerales)
    for(int i = 0; i < nCellsHeight; i++) {
        for(int j = 0; j < nCellsWidth; j++) {
            values[i][j] = cellValue(i, j, freeCells, nCellsWidth, nCellsHeight, mapWidth,
                mapHeight, obstacles, defenses);
        }
    }
    // Agrupamos las celdas en una lista
    List<Cell>::iterator itCells = cellList.begin();
    for(int i = 0; i < nCellsHeight; i++) {
        for(int j = 0; j < nCellsWidth; j++) {
            Cell c(i, j, values[i][j]);
            cellList.insert(itCells, c);
            itCells++;
        }
    }
    // Ordenamos la lista de menor a mayor valor
    cellList.sort([](Cell& c1, Cell& c2) { return c1.value() < c2.value(); });
    // Colocamos la primera defensa
    List<Defense*>::iterator itDefense = defenses.begin();
    bool placed = false;
    while(!placed) { // Mientras no est  colocada la primera defensa...
        // Seleccionamos la celda m s prometedora, que es el primer elemento de cellList
        Cell promisingCell = selection(cellList);  // promisingCell guarda la celda m s
            prometedora
        // Si es factible...
        if(factibility(*itDefense, promisingCell.row(), promisingCell.col(), obstacles,
            defenses, cellWidth, cellHeight, mapWidth, mapHeight)) {
            // Asignamos a la posici n de la defensa la posici n del centro de la celda
                m s prometedora
            (*itDefense)->position = cellCenterToPosition(promisingCell.row(), promisingCell.
```

```
                col(), cellWidth, cellHeight);
            placed = true;
        }
        // Sacamos de la lista la celda procesada
        cellList.pop_front();
    }
}
```

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

   - Conjunto de candidatos: Cada una de las celdas del mapa.
   - Conjunto de candidatos seleccionados: Cada celda donde colocamos las defensas.
   - Función selección: Selecciona la celda más prometedora, en nuestro caso devuelve la primera posición de la lista cellList ya ordenada anteriormente. (*'selection()'*)
   - Función de factibilidad: Comprueba si es posible colocar la defensa en la celda dada. (*'factibility()'*)

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 dedicada a establecer un determinado valor a cada una de las celdas del terreno de batalla para el caso del resto de defensas la hemos llamado cellValueRest().

   - Calculamos el tamaño de las celdas.
   - Obtenemos la posición del centro de extracción de minerales. Devolvemos la distancia entre el centro de extracción de minerales y el centro de la celda que estamos evaluando.

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.

```
(La clase 'Cell' y la funci n 'selection' se encuentran implementadas en el ejercicio 3)

float cellValueRest(int row, int col, bool** freeCells, int nCellsWidth, int nCellsHeight
        , float mapWidth, float mapHeight, List<Object*> obstacles, List<Defense*> defenses)
            {
    float cellWidth = mapWidth / nCellsWidth;
    float cellHeight = mapHeight / nCellsHeight;
    List<Defense*>::iterator itDefense = defenses.begin();  // El primer elemento de la lista
         de defensas es el centro de extracci n
        return _distance(cellCenterToPosition(row, col, cellWidth, cellHeight), (*itDefense)
            ->position);
}

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 values[nCellsHeight][nCellsWidth];    // Matriz que almacena los valores de cada
        celda
    List<Cell> cellList;
    // Rellenamos la matriz de valores para colocar el resto de defensas
    for(int i = 0; i < nCellsHeight; i++) {
        for(int j = 0; j < nCellsWidth; j++) {
            values[i][j] = cellValueRest(i, j, freeCells, nCellsWidth, nCellsHeight, mapWidth
                , mapHeight, obstacles, defenses);
        }
    }
    // Vaciamos la lista
    while(!cellList.empty()) { cellList.pop_front(); }
    // Agrupamos las celdas en la lista anterior
```

```
    itCells = cellList.begin();
    for(int i = 0; i < nCellsHeight; i++) {
        for(int j = 0; j < nCellsWidth; j++) {
            Cell c(i, j, values[i][j]);
            cellList.insert(itCells, c);
            itCells++;
        }
    }
    // Ordenamos la lista de menor a mayor valor
    cellList.sort([](Cell& c1, Cell& c2) { return c1.value() < c2.value(); });
    // Colocamos el resto de defensas
    itDefense = defenses.begin();
    itDefense++;    // Empezamos a colocar la segunda defensa, puesto que la primera es el
        centro de extracci n
    while(itDefense != defenses.end()) {    // Mientras haya defensas sin colocar...
        placed = false;
        while(!placed) {
            // Seleccionamos la celda m s prometedora, que es el primer elemento de cellList
            Cell promisingCell = selection(cellList);  // promisingCell guarda la celda m s
                prometedora
            if(factibility(*itDefense, promisingCell.row(), promisingCell.col(), obstacles,
                defenses, cellWidth, cellHeight, mapWidth, mapHeight)) {
                // Colocamos la defensa en el centro de la celda m s prometedora
                (*itDefense)->position = cellCenterToPosition(promisingCell.row(),
                    promisingCell.col(), cellWidth, cellHeight);
                placed = true;
            }
            // Sacamos de la lista la celda procesada
            cellList.pop_front();
        }
        itDefense++;
    }
}
```

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.