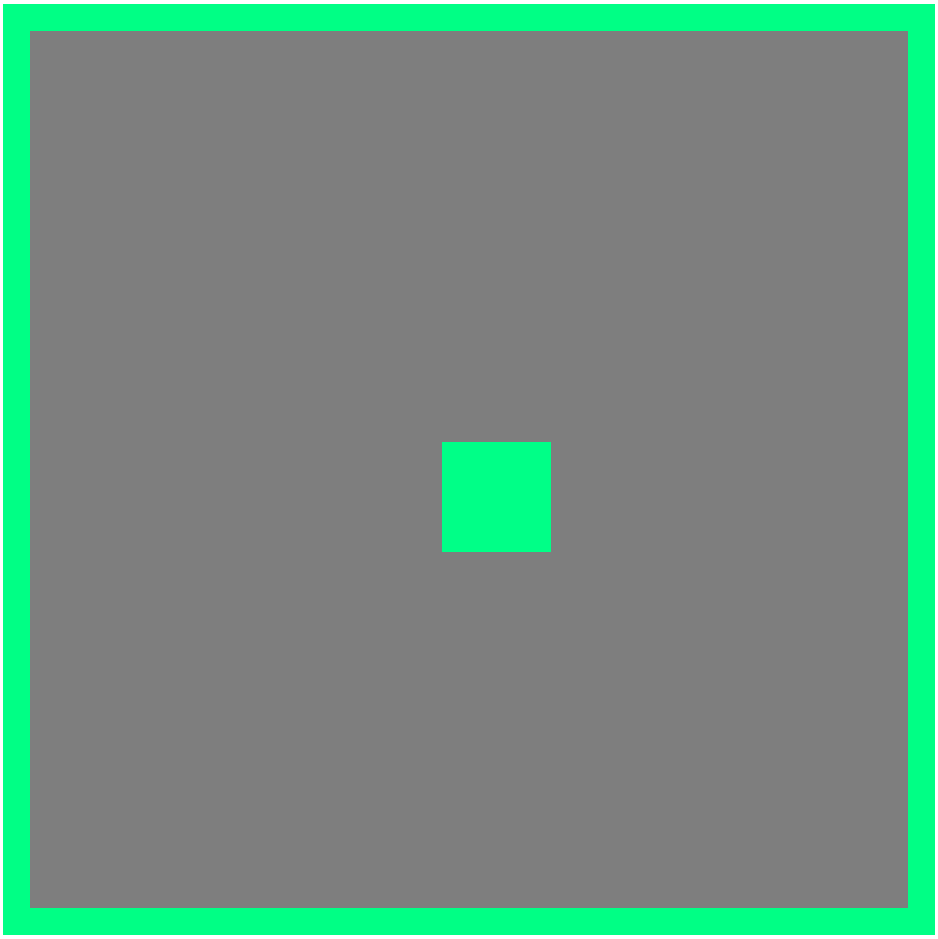

Figura 1: Estrategia devoradora para la mina

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.

```
/**
* Dada una celda o posición devuelve un valor para crear en otra función la matriz de
    valores de las celdas.
* @param row Fila a evaluar.
* @param col Columna a evaluar.
* @param freeCells matriz de numero de celdas-ancho por número de celdas-alto, contiene
    true si el centro de la celda
*                 esta libre y false si el centro de la celda esta ocupado por un
    obstáculo.
* @param nCellsWidth número de celdas en anchura.
* @param nCellsHeight número de celdas en altura.
* @param mapWidth ancho total del mapa preestablecido.
* @param mapHeight alto total del mapa preestablecido.
* @param obstacles Lista actual de Obstáculos creados y situados.
* @param defenses Lista actual de Defensas creadas, no situadas.
* @return Devuelve un valor determinado para una posición determinada del tablero.
*/
float cellValue(int row, int col, placePosition place, bool** freeCells, int nCellsWidth,
    int nCellsHeight
, float mapWidth, float mapHeight, List<Object*> obstacles, List<Defense*> defenses)
```

Otorga 100 puntos a las posiciones más valiosas, en este caso las centrales, las segundas más valiosas de 90 puntos a los bordes, 20 puntos a las posiciones cercanas al centro y 10 puntos al resto de posiciones.

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

```
        /**
        * Funcion que comprueba la factibilidad de posicionar una defensa en una posicion dada del
            mapa.
        * @param currentDefense Defensa actual a evaluar su posicion.
        * @param defenses Lista actual de Defensas creadas y situadas.
        * @param obstacles Lista actual de Obstaculos creados y situados.
        * @param mapWidth ancho total del mapa preestablecido.
        * @param mapHeight alto total del mapa preestablecido.
        * @return Devuelve true si se puede colocar y false si no se puede colocar.
        */
        bool factibility(Defense* currentDefense, std::list<Defense*> defenses, std::list<Object*>
            obstacles, float mapWidth, float mapHeight)
```

Comprueba que no se salga de los bordes y que no choque contra un obstáculo u defensa.

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.

```
/**
* Funcion que posiciona las defensas y que tiene implicita la funcion objetivo del algoritmo Voraz
* @param freeCells matriz de numero de celdas-ancho por numero de celdas-alto, contiene true si el
    centro de la celda
*                   esta libre y false si el centro de la celda esta ocupado por un obstaculo.
* @param nCellsWidth numero de celdas en anchura.
* @param nCellsHeight numero de celdas en altura.
* @param mapWidth ancho total del mapa preestablecido.
* @param mapHeight alto total del mapa preestablecido.
* @param obstacles Lista actual de Obstaculos creados y situados.
* @param defenses Lista actual de Defensas creadas, no situadas.
*/
void DEF_LIB_EXPORTED placeDefenses(bool** freeCells, int nCellsWidth, int nCellsHeight, float
    mapWidth, float mapHeight
, std::list<Object*> obstacles, std::list<Defense*> defenses)
```

En mi caso, el algoritmo es voraz porque crea una matriz de valores entre 100 y 10 puntos a las casillas, para luego ordenarlas en una lista de mayor a menor valor, y terminar colocando las defensas, primero el centro de extracción en el centro del mapa, como segunda opción colocar ciertas defensas en los bordes del mapa o lo más cercano posible, por último, en casillas cercanas al centro del mapa.

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

```
        /**
        * Funcion que selecciona la mejor posicion dado los valores de la matriz cellsvalues.
        * @param valuelist iterador que contiene el valor actual de una posicion dada del mapa.
        * @param freeCells matriz de numero de celdas-ancho por numero de celdas-alto, contiene
            true si el centro de la celda
        *                   esta libre y false si el centro de la celda esta ocupado por un
            obstaculo.
        * @param nCellsWidth numero de celdas en anchura.
        * @param nCellsHeight numero de celdas en altura.
        * @param extraction Bit que indica si es o no el centro de extraccion.
        * @return Devuelve un tipo Vector3 con la posicion mas prometedora para la defensa.
        */
        Vector3 cellSelect(std::list<ValueList>::iterator valuelist, bool** freeCells, int
            nCellsWidth, int nCellsHeight, int extraction = 0)
```

En mi caso, la función "cellSelect" se comporta eligiendo paulatinamente la primera posición más valiosa de la matriz de valores, sin tener en cuenta la posición que ocupa en el mapa dicho valor, y comportándose de forma voraz.

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.

```
        /**
        * Dada una celda o posición devuelve un valor para crear en otra función la matriz de
            valores de las celdas.
        * @param row Fila a evaluar.
        * @param col Columna a evaluar.
        * @param freeCells matriz de numero de celdas-ancho por número de celdas-alto, contiene
            true si el centro de la celda
        *                   esta libre y false si el centro de la celda esta ocupado por un
            obstáculo.
        * @param nCellsWidth número de celdas en anchura.
        * @param nCellsHeight número de celdas en altura.
        * @param mapWidth ancho total del mapa preestablecido.
        * @param mapHeight alto total del mapa preestablecido.
        * @param obstacles Lista actual de Obstáculos creados y situados.
        * @param defenses Lista actual de Defensas creadas, no situadas.
        * @return Devuelve un valor determinado para una posición determinada del tablero.
        */
        float cellValue(int row, int col, placePosition place, bool** freeCells, int nCellsWidth,
            int nCellsHeight
        , float mapWidth, float mapHeight, List<Object*> obstacles, List<Defense*> defenses)
        .
        .
        .
        .
        case bordered:
                valueOfCell = valueOfCell * 0.5f;
                break;
        case other:
                valueOfCell = valueOfCell * 0.2f;
                default:
        valueOfCell = valueOfCell * 0.1f;
                break;
```

En mi caso, la segunda parte del algoritmo "cellValue" otorgo 50 puntos a las posiciones cercanas al centro, 20 puntos a posiciones alejadas del centro del mapa, y 10 puntos al resto de posiciones, para que estén cerca del centro de extracción, y colocar el resto de las defensas.

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.

```
/**
* Funcion que posiciona las defensas y que tiene implicita la funcion objetivo del algoritmo Voraz
* @param freeCells matriz de numero de celdas-ancho por numero de celdas-alto, contiene true si el
    centro de la celda
*                   esta libre y false si el centro de la celda esta ocupado por un obstaculo.
* @param nCellsWidth numero de celdas en anchura.
* @param nCellsHeight numero de celdas en altura.
* @param mapWidth ancho total del mapa preestablecido.
* @param mapHeight alto total del mapa preestablecido.
* @param obstacles Lista actual de Obstaculos creados y situados.
* @param defenses Lista actual de Defensas creadas, no situadas.
*/
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 **cellsValues = new float*[nCellsWidth];
        /**
        * Matriz de valores del algoritmo voraz.
        */
        cellsValues = matrixValues(freeCells, nCellsWidth, nCellsHeight, mapWidth, mapHeight,
            obstacles, defenses);
        /**
        * Lista para almacenar la matriz de valores y ordenarlos
        */
```

```
std::list<ValueList> VectorValues;
Vector3 v3tmp{}, cellSelection{};
int maxAttemps = 1000, n = 0;

for (int i = 0; i < nCellsWidth; ++i)
{
        for (int j = 0; j < nCellsHeight; ++j)
        {
                v3tmp.x = (float)i;
                v3tmp.y = (float)j;
                VectorValues.emplace_back(cellsValues[i][j], v3tmp);
        }
}

/**
* Ordenacion de los valores de la lista.
*/
VectorValues.sort(std::greater<ValueList>());

std::list<ValueList>::iterator currentValue = VectorValues.begin();

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

while(currentDefense != (defenses.end()) and maxAttemps > 0)
{
        /**
        * funcion select, primera defensa en el centro
        */
        if(currentDefense == defenses.begin())
                cellSelection = cellSelect(currentValue, freeCells, nCellsWidth,
                    nCellsHeight, 1);
        else
        {
                cellSelection = cellSelect(currentValue, freeCells, nCellsWidth,
                    nCellsHeight);
        }

        (*currentDefense)->position.x = (cellSelection.x * cellWidth) + (cellWidth * 0.5f);
             //(int)(_RAND2(nCellsWidth))* cellWidth) + cellWidth * 0.5f
        (*currentDefense)->position.y = (cellSelection.y * cellHeight) + (cellHeight * 0.5f
            ); //(int)(_RAND2(nCellsHeight)) *cellHeight) + cellHeight *0.5f
        (*currentDefense)->position.z = 0;// cellSelection.z;
        if(factibility(*currentDefense, defenses, obstacles, mapWidth, mapHeight))
        {
                (*currentDefense)->health = DEFAULT_DEFENSE_HEALTH;
                ++currentDefense;
                ++currentValue;
                //quitar de la lista
                VectorValues.pop_front();
        }
        else
        {
                ++currentValue;
                //quitar de la lista
                VectorValues.pop_front();
        }
}

#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)(cellsValues[i][j]) % 256;
        }
}
dPrintMap("defenseValueCellsHead.ppm", nCellsHeight, nCellsWidth, cellHeight, cellWidth,
    freeCells
```

```
        , cellValues, std::list<Defense*>(), true);

        for(int i = 0; i < nCellsHeight ; ++i)
        delete [] cellValues[i];
        delete [] cellValues;
        cellValues = nullptr;

        #endif
        for(int i = 0; i < nCellsHeight ; ++i)
        delete [] cellsValues[i];
        delete [] cellsValues;
        cellsValues = nullptr;
}
```

Mi estrategia está formada por un algoritmo voraz que selecciona las celdas más valiosas en primer lugar mediante una lista rellenada por una matriz de puntuaciones, apoyándose en la función de factibilidad para descartar las posiciones del mapa ya ocupadas y en segundo lugar lo más cercano al centro del mapa mas varias en los bordes como protección de avanzadilla.

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.