Práctica 1. Algoritmos devoradores

Miguel Angel Fernandez Jimenez
miguel.ferjimenez@alum.uca.es
Teléfono: 601101415
NIF: 77497367V

13 de noviembre de 2022

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.

   Escriba aquí su respuesta al ejercicio 1.

   Realizo una ponderación en caso de lo que considere de mayor valor o menor sobre un 100 por ciento, como podemos observar en las celdas correspondientes a los centros de extracción lo valoraremos en un 1 por ciento

```
float cellValueExtr(int row, int col, bool **freeCells, int nCellsWidth, int nCellsHeight,
    float mapWidth, float mapHeight, List<Object *> obstacles)
{
    float cellWidth = mapWidth / nCellsWidth;
    float cellHeight = mapHeight / nCellsHeight;
    int bestRow = nCellsHeight / 2, bestCol = nCellsWidth / 2;
    Vector3 posicion = cellCenterToPosition(row, col, cellWidth, cellHeight);
    Vector3 bestPos = cellCenterToPosition(bestRow, bestCol, cellWidth, cellHeight);
    float value;

    value = bestPos.length() - _distance(posicion, bestPos);

    if (row < 3 || row > nCellsHeight - 3 || col > nCellsWidth - 3 || col < 3)
        value = value * 0.01;

    return value;
}
```

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

   Escriba aquí su respuesta al ejercicio 2.

```
bool factible(List<Defense *> defenses, const Defense &currentDef, List<Object *> obstacles,
    float mapHeight,
              float cellWidth, float cellHeight, float mapWidth, int row, int col, bool **
                  freeCells)
{
    bool fact;

    Vector3 posicion = cellCenterToPosition(row, col, cellWidth, cellHeight);

    if (posicion.x + currentDef.radio > mapWidth ||
        posicion.x - currentDef.radio < 0 ||
        posicion.y + currentDef.radio > mapHeight ||
        posicion.y - currentDef.radio < 0)
        fact = false;
    else
        fact = true;

    List<Object *>::iterator i;
```

Figura 1: Estrategia devoradora para la mina

```
    for (i = obstacles.begin(); i != obstacles.end(); i++)
    {
        //Si la distancia entre los centros es menor a la suma de los radios de la defensa y
            el obst culo, colisionan
        if (_distance(posicion, (*i)->position) < (currentDef.radio + (*i)->radio))
            fact = false;
    }

    List<Defense *>::iterator j;

    for (j = defenses.begin(); j != defenses.end(); j++)
    {
        //Si la distancia entre los centros es menor a la suma de los radios de la defensa
            actual y la que est  colocada, colisionan
        if (_distance(posicion, (*j)->position) < (currentDef.radio + (*j)->radio))
            fact = false;
    }

    return fact;
}
```

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;

        int maxAttemps = 1000;
        std::vector<Cell> Cells;
        int i, j, k;
        float aux;
        Cell auxCell;

        //Algoritmo centro extracci n
        for (i = 0; i < nCellsHeight; i++)
            for (j = 0; j < nCellsWidth; j++)
            {
                if (freeCells[i][j])
                    Cells.push_back(Cell(i, j, cellValueExtr(i, j, freeCells, nCellsWidth,
                        nCellsHeight, mapWidth, mapHeight, obstacles)));
            }

        for (i = 0; i < Cells.size(); i++)
            for (k = i + 1; k < Cells.size(); k++)
            {
                if (Cells[k].value < Cells[i].value)
                {
                    auxCell = Cells[i];
                    Cells[i] = Cells[k];
                    Cells[k] = auxCell;
                }
            }

        bool placed = false;
        Cell solution;
        List<Defense *>::iterator currentDefense = defenses.begin();

        //Algoritmo devorador para centro de extraccion
        while (!placed && !Cells.empty())
        {
            solution = Cells.back();
            Cells.pop_back();
            if (factible(defenses, *(*currentDefense), obstacles, mapHeight,
                         cellWidth, cellHeight, mapWidth, solution.row, solution.col,
                             freeCells))
```

```
            {
                placed = true;
                freeCells[solution.row][solution.col] = false;
                (*currentDefense)->position = cellCenterToPosition(solution.row, solution.col
                    , cellWidth, cellHeight);
            }
        }

        std::vector<Cell> AuxCells;

        for (i = 0; i < nCellsHeight; i++)
            for (j = 0; j < nCellsWidth; j++)
            {
                if (freeCells[i][j])
                    AuxCells.push_back(Cell(i, j, cellValue(i, j, solution.row, solution.col,
                         freeCells, nCellsWidth, nCellsHeight, mapWidth, mapHeight, obstacles
                        , defenses)));
            }

        k = 0;

        for (i = 0; i < AuxCells.size(); i++)
            for (k = i + 1; k < AuxCells.size(); k++)
            {
                if (AuxCells[k].value < AuxCells[i].value)
                {
                    auxCell = AuxCells[i];
                    AuxCells[i] = AuxCells[k];
                    AuxCells[k] = auxCell;
                }
            }

        Cells.clear();
        Cells = AuxCells;

        std::vector<Cell>::iterator it;
        int iterations;
        currentDefense++;

        while (currentDefense != defenses.end())
        {
            iterations = 0;
            placed = false;
            it = AuxCells.end();
            while (!placed && !Cells.empty())
            {
                iterations++;
                solution = Cells.back();
                Cells.pop_back();
                if (factible(defenses, *(*currentDefense), obstacles, mapHeight, cellWidth,
                    cellHeight, mapWidth, solution.row, solution.col, freeCells))
                {
                    placed = true;
                    (*currentDefense)->position = cellCenterToPosition(solution.row, solution
                        .col, cellWidth, cellHeight);
                    while (iterations > 0)
                    {
                        it--;
                        iterations--;
                    }
                    AuxCells.erase(it);
                }
            }
            Cells.clear();
            Cells = AuxCells;
            currentDefense++;
        }
    }
```

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

Escriba aquí su respuesta al ejercicio 4.

Este algoritmo se comporta como un algoritmo devorador al recibir un conjunto de candidatos (en este caso los centros de extracción), y de estos selecciona los más prometedores. Una función de selección para realizar lo dicho antes, que nos devolverá la celda de mayor valoración. Una función de factibilidad para que una vez seleccionado dicho centro, comprobemos si este cumple los requisitos para ser una solución. Un conjunto de soluciones donde introducir los centros comprobados por la función de factibilidad y una función objetivo ya definida gracias a la función factibilidad, que comprobará que centro de extracción es más prometedor y lo introducirá en el conjunto de soluciones previamente mencionado.

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.

   Escriba aquí su respuesta al ejercicio 5.

   Como se ha explicado en el ejercicio 1, la ponderación que he decidido asignar a las celdas del campo con alguna defensa asignada es del 25 por ciento.

```
float cellValue(int row, int col, int exRow, int exCol, bool **freeCells, int nCellsWidth,
    int nCellsHeight, float mapWidth, float mapHeight, List<Object *> obstacles, List<Defense
     *> defenses)
{

    float cellWidth = mapWidth / nCellsWidth;
    float cellHeight = mapHeight / nCellsHeight;
    Vector3 posicion = cellCenterToPosition(row, col, cellWidth, cellHeight);
    Vector3 bestPos = cellCenterToPosition(exRow, exCol, cellWidth, cellHeight);
    float value;

    value = bestPos.length() - _distance(posicion, bestPos);

    if (_distance(bestPos, posicion) > 3 * (*(defenses.back())).radio)
        value = value * 0.25;

    return value;
}
```

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)
    {

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

        int maxAttemps = 1000;
        std::vector<Cell> Cells;
        int i, j, k;
        float aux;
        Cell auxCell;

        //Algoritmo centro extracci n
        for (i = 0; i < nCellsHeight; i++)
            for (j = 0; j < nCellsWidth; j++)
            {
                if (freeCells[i][j])
                    Cells.push_back(Cell(i, j, cellValueExtr(i, j, freeCells, nCellsWidth,
                        nCellsHeight, mapWidth, mapHeight, obstacles)));
            }
```

```
        for (i = 0; i < Cells.size(); i++)
            for (k = i + 1; k < Cells.size(); k++)
            {
                if (Cells[k].value < Cells[i].value)
                {
                    auxCell = Cells[i];
                    Cells[i] = Cells[k];
                    Cells[k] = auxCell;
                }
            }

        bool placed = false;
        Cell solution;
        List<Defense *>::iterator currentDefense = defenses.begin();

        //Algoritmo devorador para centro de extraccion
        while (!placed && !Cells.empty())
        {
            solution = Cells.back();
            Cells.pop_back();
            if (factible(defenses, *(*currentDefense), obstacles, mapHeight,
                         cellWidth, cellHeight, mapWidth, solution.row, solution.col,
                             freeCells))
            {
                placed = true;
                freeCells[solution.row][solution.col] = false;
                (*currentDefense)->position = cellCenterToPosition(solution.row, solution.col
                    , cellWidth, cellHeight);
            }
        }

        std::vector<Cell> AuxCells;

        for (i = 0; i < nCellsHeight; i++)
            for (j = 0; j < nCellsWidth; j++)
            {
                if (freeCells[i][j])
                    AuxCells.push_back(Cell(i, j, cellValue(i, j, solution.row, solution.col,
                         freeCells, nCellsWidth, nCellsHeight, mapWidth, mapHeight, obstacles
                        , defenses)));
            }

        k = 0;

        for (i = 0; i < AuxCells.size(); i++)
            for (k = i + 1; k < AuxCells.size(); k++)
            {
                if (AuxCells[k].value < AuxCells[i].value)
                {
                    auxCell = AuxCells[i];
                    AuxCells[i] = AuxCells[k];
                    AuxCells[k] = auxCell;
                }
            }

        Cells.clear();
        Cells = AuxCells;

        std::vector<Cell>::iterator it;
        int iterations;
        currentDefense++;

        while (currentDefense != defenses.end())
        {
            iterations = 0;
            placed = false;
            it = AuxCells.end();
            while (!placed && !Cells.empty())
            {
                iterations++;
                solution = Cells.back();
```

```
                Cells.pop_back();
                if (factible(defenses, *(*currentDefense), obstacles, mapHeight, cellWidth,
                    cellHeight, mapWidth, solution.row, solution.col, freeCells))
                {
                    placed = true;
                    (*currentDefense)->position = cellCenterToPosition(solution.row, solution
                        .col, cellWidth, cellHeight);
                    while (iterations > 0)
                    {
                        it--;
                        iterations--;
                    }
                    AuxCells.erase(it);
                }
            }
            Cells.clear();
            Cells = AuxCells;
            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.