Práctica 1. Algoritmos devoradores

Luis José Quintana Bolaño
luisjquintana@gmail.com
Teléfono: 956535843
NIF: 49073584w

23 de noviembre de 2014

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.

   Para el caso del centro de extracción de minerales, la función otorga un valor a cada celda en dos partes.

   La primera parte consiste en un valor de 0 a 1 que describe la proximidad de la celda al centro del mapa, calculado como:

   $$P_{C_{x,y}} = \frac{t.mapa/2 - d.euclidea(C_{x,y}, Centro)}{t.mapa/2}$$

   La segunda parte otorga un valor de 0 a 0.5 que es sumado al anterior, y que representa la cercanía de los obstáculos del mapa a la posición, calculada para cada obstáculo como:

   $$O_{C_{x,y}} = \frac{t.mapa - d.euclidea(C_{x,y}, Obstaculo)}{n.obstaculos}$$

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

   Una posición es factible siempre que la distancia entre la posición y las posiciones de cada obstáculo y defensa ya posicionadas sea menor que la suma de los radios de ambos objetos.

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.

```
std::vector <std::pair <std::pair<int, int>, float> > evaluateMain (int nCellsWidth, int
    nCellsHeight, float cellWidth, float cellHeight, std::list<Object*> obstacles){

        std::vector <std::pair <std::pair<int, int>, float> > cellVal (nCellsWidth*
            nCellsHeight);
        float maxDistance = euclideanDistance(0,(nCellsWidth-1)*cellWidth,0,(nCellsHeight-1)*
            cellHeight);
    float maxDistanceCenter = euclideanDistance(0,((nCellsWidth-1)*cellWidth)/2,0,((
        nCellsHeight-1)*cellHeight)/2);

        int i=0;
        for (int x=0; x<nCellsWidth; ++x){
                for (int y=0; y<nCellsHeight;++y){
                        //First values are assigned to center the turret
                        float centerValue=(maxDistanceCenter-euclideanDistance(x*cellWidth,((
                            nCellsWidth-1)*cellWidth)/2,y*cellHeight,((nCellsHeight-1)*
                            cellHeight)/2))/maxDistanceCenter;

                        //Now a modifier is added according to the number of obstacles
                            surrounding the position
                        //More obstacles mean a higher value
                        float obstacleValue=0;
                        for (List<Object*>::iterator currentObstacle = obstacles.begin();
                            currentObstacle != obstacles.end();++currentObstacle){
                                float objectDistance = euclideanDistance(x*cellWidth+
                                    cellWidth*0.5f,(*currentObstacle)->position.x,y*
                                    cellHeight+cellHeight*0.5f,(*currentObstacle)->position.y
                                    );
```

```
                                obstacleValue+=((maxDistance-objectDistance)/maxDistance)/
                                    obstacles.size();
                            }

                            //The sum of both values is assigned to the cell
                            cellVal[i++]=std::make_pair(std::make_pair(x,y),centerValue+
                                obstacleValue);
                    }
            }

            return cellVal;
}

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;
    List<Defense*>::iterator currentDefense = defenses.begin();
    std::vector <std::pair <std::pair<int, int>, float> > cellVal;
    std::vector <std::pair <std::pair<int, int>, float> >::iterator currentCell;

//MAIN TOWER=============================================================================
    //Cell values for the position of the main tower are calculated
        //A higher value represents a better position.
        cellVal = evaluateMain(nCellsWidth,nCellsHeight,cellWidth,cellHeight,obstacles);

        //The values get sorted descendingly to try and place the tower in the best position
        std::sort(cellVal.begin(), cellVal.end(), comparePair);

        //We try every position from highest value to lowest
        //The tower is placed in the first feasible position
        bool positioned = false;
        currentCell = cellVal.begin();
        while(currentCell != cellVal.end() && !positioned){
                if(feasibility(cellWidth,cellHeight,currentCell->first.first,currentCell->
                    first.second,currentDefense,obstacles,defenses)){
                        (*currentDefense)->position.x = currentCell->first.first * cellWidth
                            + cellWidth * 0.5f;
                (*currentDefense)->position.y = currentCell->first.second * cellHeight +
                    cellHeight * 0.5f;
                (*currentDefense)->position.z = 0;
                positioned=true;
                } else {
                        ++currentCell;
                }
        }
```

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

   El algoritmo es categorizado como algoritmo voraz, puesto que busca un resultado local aplicando una función de factibilidad y descarta los elementos ya comprobados (en este caso, las posiciones).

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.

   En el caso del resto de las defensas, simplemente se ha calculado un valor de 0 a 1 que describe la proximidad de cada posición a la base. La función es similar a la usada en la primera parte del ejercicio 1 para centrar la torre principal, siendo en este caso 1 el valor correspondiente a la máxima distancia del mapa.

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.

```
std::vector <std::pair <std::pair<int, int>, float> > evaluateDefenses (int nCellsWidth, int
    nCellsHeight, float cellWidth, float cellHeight, Defense* mainTower){

        std::vector <std::pair <std::pair<int, int>, float> > cellVal (nCellsWidth*
            nCellsHeight);
        float maxDistance = euclideanDistance(0,(nCellsWidth-1)*cellWidth,0,(nCellsHeight-1)*
            cellHeight);

        int i=0;
        for (int x=0; x<nCellsWidth; ++x){
                for (int y=0; y<nCellsHeight;++y){
                        //First values are assigned according to proximity to the main tower
                        cellVal[i++]=std::make_pair(std::make_pair(x,y),(maxDistance-
                            euclideanDistance(x*cellWidth,mainTower->position.x,y*cellHeight,
                            mainTower->position.y))/maxDistance);
                }
        }

        return cellVal;
}


//DEFENSES=============================================================================
        //Cell values for the position of the defensive towers are calculated
        //A higher value represents a better position.
        cellVal = evaluateDefenses (nCellsWidth,nCellsHeight,cellWidth,cellHeight,*
            currentDefense);

        //The values get sorted descendingly to try and place the towers in the best
            positions
        std::sort(cellVal.begin(), cellVal.end(), comparePair);

        //We try every position from highest value to lowest
        //Each tower is placed when a feasible position is find
        currentCell = cellVal.begin();
        while(currentCell != cellVal.end() && currentDefense != defenses.end()){
                if(feasibility(cellWidth,cellHeight,currentCell->first.first,currentCell->
                    first.second,currentDefense,obstacles,defenses)){
                        (*currentDefense)->position.x = currentCell->first.first * cellWidth
                            + cellWidth * 0.5f;
                (*currentDefense)->position.y = currentCell->first.second * cellHeight +
                    cellHeight * 0.5f;
                (*currentDefense)->position.z = 0;
                ++currentDefense;
                }
                ++currentCell;
        }
```

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.