Práctica 1. Algoritmos devoradores

Juan Manuel Grondona Nuño
juanmanuel.grondonanu@alum.uca.es
Teléfono: 656485032
NIF: 49193526E

11 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.

   En este caso la función se divide en dos partes.

   - Valoracion por lo centrado que esté en el tablero.
   - Valoracion por lo centrado que esté con respecto el número de obstáculos alrededor.

   Para el primer tipo, doy una valoración positiva a la celda central, y voy disminuyendo según se aleje. La valoración más alta de las celdas es el de mínimo ente la longitud y la altitud del mapa dividida ente 3, de este modo se genera un "cuadrado" en torno a esta celda central y después de 0, otorga valores negativos.

   Para la segunda parte de la valoración, comprobamos alrededor de la celda, los obstáculos que hay, después esta se valora según la media de estas rocas, cuanto mayor sea esta, mejor puntuación, ya que con un mayor media existe una mayor distancia entre estas, y de esta forma se puede posicionar bien la torreta principal.

   Además, según el tamaño del mapa, el valor máximo que pueden aportas estos se ven modificados, de esta manera podemos encontrar la posición más segura que puede estar más alejada.

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

   Para la función factibilidad simplemente haremos tres observaciones.

   - La defensa no se colocara fuera del mapa.
   - La defensa no se colocará donde coincida con algún obstáculo.
   - La defensa no se colocará donde coincida con otra defensa.

   Si una de estas condiciones se cumpliera, la función devolverá false, y por el contrario devolverá true.

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.

```
struct celda{
    int x_;
    int y_;
    int valor;
    Vector3 posicion;

    celda(int x, int y, int val, double cellW, double cellH): x_(x), y_(y), valor(val){
        posicion = Vector3(x * cellH + cellH * 0.5f, y * cellW + cellW * 0.5f, 0);
    }
};

// funcion que valorea las celdas

int cellValueBase(int row, int col, int nCellsWidth, int nCellsHeight
```

```
        , float mapWidth, float mapHeight, List<Object*> obstacles, List<Defense*> defenses)
            {

    int RadioPiedras = 13;
    double densidad = 0;

    int valor, inter;
    if (nCellsHeight <= 24 || nCellsWidth <= 24){
        inter = std::min(ceil(nCellsWidth / 3), ceil(nCellsHeight / 3));
    }
    else {
        inter = std::min(ceil(nCellsWidth / 2), ceil(nCellsHeight / 2));
    }

    int distCenWid = fabs(row - ceil(nCellsHeight / 2));
    int distCenHei = fabs(col - ceil(nCellsWidth / 2));
    valor = std::min(inter - distCenHei, inter - distCenWid);

    if (valor > 0 ){
        double distancia, media = 0;
        int maxva, aux = 0;
        for (auto it = obstacles.begin(); it != obstacles.end(); it++) {
            distancia = _distance((*it)->position, Vector3((double)row / nCellsHeight *
                mapHeight, (double)col / nCellsWidth * mapWidth, 0));
            if (distancia <= RadioPiedras) {
                media += distancia;
                aux++;
            }
        }
        valor += inter * media / aux;

    }
    return valor;
}

//funcion que considera posible esa celda

bool factible(Object* defensa, celda *cell, float mapWidth, float mapHeight, List<Object*>
    obstacles, List<Defense*> defenses) {
    bool res = true;
    if ((defensa->radio >= cell->posicion.x || cell->posicion.x >= mapWidth - defensa->radio)
         || (defensa->radio >= cell->posicion.y || cell->posicion.y >= mapHeight - defensa->
        radio) ) {
        res = false;
    }
    for (auto it = obstacles.begin(); it != obstacles.end() && res; it++){
        if (distObjeRad(cell->posicion, defensa->radio, (*it)->position, (*it)->radio) <= 5)
            {
            res = false;
        }
    }
    for (auto it = defenses.begin(); it != defenses.end() && res; it++){
        if (distObjeRad(cell->posicion, defensa->radio, (*it)->position, (*it)->radio) <= 0)
            {

            res = false;
        }

    }
    return res;
}

//algoritmo devorador

void colocaBase(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::list<celda*> valoresbase;
    for(int i = 0; i < nCellsHeight; ++i) {
        for(int j = 0; j < nCellsWidth; ++j) {
            valoresbase.push_back(new celda(i, j, cellValueBase(i, j, nCellsWidth,
                nCellsHeight, mapWidth, mapHeight, obstacles, defenses), cellWidth,
                cellHeight));
        }
    }
    valoresbase.sort([](celda* a, celda* b) -> bool{return a->valor > b->valor;});

    std::list<Defense*> posicionadas;
    auto defAct = defenses.begin();
    auto cellAct = valoresbase.begin();
    bool fin = true;
    while (cellAct != valoresbase.end() && fin){
        if (factible(*defAct, (*cellAct), mapWidth, mapHeight, obstacles, defenses)){
            (*defAct)->position = (*cellAct)->posicion;
            fin = false;
            defAct++;
        }
        else{
            cellAct++;
        }
    }

}
```

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

   En lo primero en lo que se parece es en que tenemos un primer conjunto de Soluciones que serian las posibles celdas en las que colocarse. A estas se les da una valoración, y se van comprobando de una a una, de mayor a menor, como en un algoritmo devorador. Además de esto, tenemos que comprobar todas las celdas seleccionadas con la función factible, que también está en los algoritmos devoradores y se encarga de comprobar que ese caso sea válido, en nuestro caso, comprueba que se puede colocar en esa celda en concreto una torreta y que no hay nada allí que haga que colisionen o que no se salga del mapa.
   La única diferencia de este algoritmo con respecto a los algoritmos voraces, es que ese no devuelve el conjunto solución, ya que es un problema de colocación de objetos, y se pueden colocar directamente cada objeto dentro de la función, y de este modo no tener que devolver nada.

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.

   Esta función en mi caso la he hecho mucho más simple, simplemente cuanto más cerca de la torre de extracción minera este, más puntuación.
   De esta manera toda las torretas están cerca de la central y están bástente pegadas unas a otras, lo cual tiene una ventaja, y es que los radios se superponen, eso significa que varias torretas golpean a un mismo objetivo, acabando así antes con estos.

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.

```
struct celda{
    int x_;
    int y_;
    int valor;
    Vector3 posicion;

    celda(int x, int y, int val, double cellW, double cellH): x_(x), y_(y), valor(val){
        posicion = Vector3(x * cellH + cellH * 0.5f, y * cellW + cellW * 0.5f, 0);
    }
};
```

```
// funcion que valorea las celdas

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

    int valor;
    int maxVal = std::min(ceil(nCellsWidth / 3), ceil(nCellsHeight / 3));
    int distCenWid = fabs(row - defenses.front()->position.x / (mapHeight / nCellsHeight) );
    int distCenHei = fabs(col - defenses.front()->position.y / (mapWidth / nCellsWidth) );
    valor = std::min(maxVal - distCenHei, maxVal - distCenWid);
    return valor;
}

//funcion que considera posible esa celda

bool factible(Object* defensa, celda *cell, float mapWidth, float mapHeight, List<Object*>
    obstacles, List<Defense*> defenses) {
    bool res = true;
    if ((defensa->radio >= cell->posicion.x || cell->posicion.x >= mapWidth - defensa->radio)
         || (defensa->radio >= cell->posicion.y || cell->posicion.y >= mapHeight - defensa->
        radio) ) {
        res = false;
    }
    for (auto it = obstacles.begin(); it != obstacles.end() && res; it++){
        if (distObjeRad(cell->posicion, defensa->radio, (*it)->position, (*it)->radio) <= 5)
            {
            res = false;
        }
    }
    for (auto it = defenses.begin(); it != defenses.end() && res; it++){
        if (distObjeRad(cell->posicion, defensa->radio, (*it)->position, (*it)->radio) <= 0)
            {

            res = false;
        }

    }
    return res;
}

//algoritmo devorador

void colocaTorre(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::list<celda*> valorestorre;
    for(int i = 0; i < nCellsHeight; ++i) {
       for(int j = 0; j < nCellsWidth; ++j) {
           valorestorre.push_back(new celda(i, j, cellValueTorretas(i, j, freeCells,
               nCellsWidth, nCellsHeight, mapWidth, mapHeight, obstacles, defenses),
               cellWidth, cellHeight));
       }
    }
    valorestorre.sort([](celda* a, celda* b) -> bool{return a->valor > b->valor;});

    auto defAct = ++defenses.begin();
    auto cellAct1 = valorestorre.begin();
    bool fin1 = true;
    while (defAct != defenses.end()){
        while (cellAct1 != valorestorre.end() && fin1){
            if (factible(*defAct, (*cellAct1), mapWidth, mapHeight, obstacles, defenses)){
                (*defAct)->position = (*cellAct1)->posicion;
                fin1 = false;
                valorestorre.erase(cellAct1);
                cellAct1 = valorestorre.begin();
```

```
            }
            else{
                cellAct1++;
            }
        }
        fin1 = true;
        defAct++;
    }
}

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;

    colocaBase(freeCells, nCellsWidth, nCellsHeight, mapWidth, mapHeight, obstacles, defenses
        );

    colocaTorre(freeCells, nCellsWidth, nCellsHeight, mapWidth, mapHeight, obstacles,
        defenses);

}
```

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.