Práctica 1. Algoritmos devoradores

Ignacio Rodríguez Pérez
ignacio.rodriguezperez@alum.uca.es
Teléfono: +34 697 469 718

9 de noviembre de 2020

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 asignarle un valor a cada celda, con el objetivo de determinar cuál es más prometedora, se ha tenido en cuenta dos cosas:
   -La primera de ellas es que, como el sentido común nos dice, cuanto más centrada esté, más valor tendrá; o dicho de otra forma, cuanto más alejado del centro (y por ende, más cerca de los bordes) menor valor.
   -Por otro lado, se ha tenido en cuenta los obstáculos que hay, ya que, a igualdad de lejanía, los obstáculos pueden dificultar a los ucos y, por lo tanto, proteger mejor a la defensa que queremos proteger; el centro de extracción de minerales.

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

   Una función de factibilidad deberá comprobar que la defensa se puede colocar en el mapa en cuestión.
   Problemas para ésto podrían ser que se colocase en una posición que no pertenezca al mapa (o que su centro si, pero el radio de ésta sobrepase estos límites) o que chocase con un objeto; es decir, que hubiese alguna coordenada que coincidiese con alguna de un obstáculo o alguna defensa (ya sea el centro, o una pequeña parte del radio de ambas).

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.

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

    List<Defense*> colocadas;
    float aux, vMax=-1.0;
    Celda cPrometedora;
    size_t x = 0, y = 0;
    for( size_t x = 0; x< nCellsWidth; ++x)
        for(size_t y = 0; y< nCellsHeight; ++y)
         if((aux=cellValue1(x,y,nCellsWidth,nCellsHeight,obstacles)) > vMax)
            {
                float auxX = (*defenses.begin())->position.x,
                 auxY=(*defenses.begin())->position.y;
                (*defenses.begin())->position=cellCenterToPosition(x,y,cellWidth,cellHeight);
                if(factible(*defenses.begin(), colocadas, obstacles, mapWidth, mapHeight))
                    vMax = aux,  cPrometedora = Celda(x,y,vMax);
                else
                    (*defenses.begin())->position.x=auxX,(*defenses.begin())->position.y=auxY
            }
    colocadas.push_back(*defenses.begin());
    Defense* centroExtraccion =  (*defenses.begin());
    defenses.pop_front();
```

   Se evalua cada celda y, si el valor de dicha celda es mayor que la que tenemos como más prometedora, miramos si es factible (para ello debemos tener una defensa, por lo que guardamos sus coordenadas para deshacer los cambios si fuese necesario). Si es factible, actualizamos los datos, sino deshacemos los cambios.
   Una vez tenemos la defensa colocada en la celda más prometedora, la añadimos a la lista de defensas colocadas, y la sacamos de la lista de defensas disponibles. La variable *centroExtraccion* nos facilitará el código después.
   Nota: falta un ; al final del else, omitido por claridad.

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

   Los elementos que definen un algoritmo voraz son los siguientes:
   Conjunto de candidatos iniciales: Cada una de las posiciones/celdas en el mapa (exista o no otro objeto ahí; de ésto se encarga la función factibilidad).
   Función solucion: Colocar el centro de extracción.
   Función de selección: Selecciona la celda que la función cellValue1 da mejor resultado; es decir, la más prometedora.
   Función de factibilidad: Verifica si el centro de extracción (u otra defensa) se puede colocar en una determinada celda, ya que puede ocurrir que otra defensa colisione.
   Función objetivo: El valor de cada celda es el producto de la distancia inversa al centro (es decir, cuanto más lejos del centro, menos valor) por el número de obstáculos que se encuentren en la zona a tratar (una celda con obstáculos, a la misma distancia que otra sin obstáculos tendrá más valor que ésta segunda).
   Objetivo: maximizar el valor.

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.

   Para el resto de defensas, se colocarán alejadas de la defensa principal un cierto número de celdas (con un estudio previo del escalar más adecuado), pero **en rango** con ésta; con el objetivo de que las defensas empiecen a disparar antes. Como no se puede reevaluar, el rango que se tomará será el de la defensa con menos rango (para ello, previamente se ordenan el resto de 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.

```
 colocadas.push_back(*defenses.begin());
    Defense* centroExtraccion =  (*defenses.begin());
    defenses.pop_front();
//Esto estaba en el ejercicio 3; no se hace dos veces, pero para saber bien donde se ubica
    defenses.sort(DefenseSort);

    float rangemin = (*defenses.begin())->range;

    std::vector<Celda> vCeldas;
    for( size_t x = 0; x< nCellsWidth; ++x)
        for(size_t y = 0; y< nCellsWidth; ++y)
            vCeldas.push_back(Celda(x,y, cellValue2(x, y, centroExtraccion, rangemin,
                nCellsWidth, nCellsHeight, obstacles)));

    std::sort(vCeldas.begin(), vCeldas.end(), std::less<Celda>());

    size_t indice=0;

    while(defenses.size()>0)
    {
        Celda c = vCeldas[indice];
        (*defenses.begin())->position = cellCenterToPosition(c.x,c.y,cellWidth,cellHeight);
        if(factible(*defenses.begin(), colocadas, obstacles, mapWidth, mapHeight))
            colocadas.push_back(*defenses.begin()), defenses.pop_front();
        indice++;
    }
```

Se ordenan las defensas para que se coloquen primero las de menos rango, y, en caso de igualdad, las que más daño hagan(código a continuación). Tras esto, como podemos ver en el código inmediatamente anterior a este texto, obtenemos el rango menor, evaluamos cada celda con el segundo algoritmo, ordenamos las celdas por valores, y asignamos posiciones a cada defensa, siempre que sea factible hacerlo.

```
bool DefenseSort(Defense* d1, Defense* d2) {

    return (d1->range != d2->range? d1->range < d2->range:
    d1->dispersion*d1->damage*d1->attacksPerSecond < d2->dispersion*d2->damage*d2->
        attacksPerSecond);
}
```

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 u otras. 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.