Práctica 1. Algoritmos devoradores

Federico Carrillo Chaves
federico.carrilloch@alum.uca.es
Teléfono: +34615158732
NIF: 32095180Z

6 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 el centro de extracción de minerales, en adelante C.E.M, las celdas del terreno del mapa estarán mejor valoradas cuánto más cerca se encuentren a la posición central del mapa.

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

   La función de factibilidad implementada tiene una serie de criterios evaluados en el siguiente orden:

   a) Determina si la posición candidata está dentro de los límites del mapa.

   b) Determina si la posición candidata no choca con ninguna de las defensas anteriormente colocadas.

   c) Determina si la posición candidata no choca con ningún obstáculo.

   Una vez que la posición candidata ha pasado esos 3 criterios, se considera que la posición es válida.

   También comentar que la función de factibilidad catalogará a una celda como no válida en cuanto no cumpla uno de los tres criterios.

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.

   Nuestro algoritmo devorador va a usar una cola de prioridad Q para almacenar las celdas candidatas. Recorreremos todas las posiciones del tablero y le aplicaremos la función de evaluación. Ahora sólo nos queda ir sacando de la cola de prioridad una por una las celdas hasta que demos con la primera que sea factible. Una vez encontrada se coloca el CEM en esa posición y se almacena también en un vector de defensas ya colocadas.

```
  // Estructura que representa una celda candidata
  // valor_        - valor de la celda
  // vector3_      - coordenadas de la celda en Vector3
class Celda {
public:
  double valor_;
  Vector3 vector3_;

  Celda(double valor, const Vector3 &vector3) : valor_(valor), vector3_(vector3) {}
};

// this is an strucure which implements the
// operator overlading
struct ComparaValor {
  bool operator()(Celda const &celda1, Celda const &celda2) {
    // Cuanto mayor valor, mayor prioridad.
    // Con esto conseguimos que las celdas con mas cerca del centro tengan mayor prioridad.
    return celda1.valor_ > celda2.valor_;
  }
};

// cola de prioridad para las celdas candidatas
  std::priority_queue<Celda, std::vector<Celda>, ComparaValor> Q;
```

```
int cellValue(int row, int col, bool **freeCells, int nCellsWidth, int nCellsHeight, float
    mapWidth, float mapHeight, const std::list<Object *> &obstacles, std::list<Defense *>
    defenses, bool esCEM) {
  float cellWidth = mapWidth / float(nCellsWidth);
  float cellHeight = mapHeight / float(nCellsHeight);
  if (esCEM) {
    Vector3 pos_actual = cellCenterToPosition(row, col, cellWidth, cellHeight);
    Vector3 centroDelMapa_v3 = Vector3(mapWidth / 2, mapHeight / 2, 0);
    return abs(_distance(pos_actual, centroDelMapa_v3));
  } else {
    .
    .
    .
  }
}
```

```
void DEF_LIB_EXPORTED
placeDefenses(bool **freeCells, int nCellsWidth, int nCellsHeight, float mapWidth, float
    mapHeight, std::list<Object *> obstacles, std::list<Defense *> defenses) {
.
.
.
  Vector3 aux_vector3;
    for (int i = 0; i < nCellsHeight; i++) {
      for (int j = 0; j < nCellsWidth; j++) {
        aux_vector3 = cellCenterToPosition(i, j, cellWidth, cellHeight);
        Celda aux = Celda(
              cellValue(i, j, freeCells, nCellsWidth, nCellsHeight, mapWidth, mapHeight,
                  obstacles, defenses, true),
              aux_vector3
              );
          Q.push(aux);
      }
    }

  std::vector<Defense *> defensesPlaced;
  auto defensaCandidata = defenses.begin();
  bool CEM_colocado = false;
// CEM
  while (defensaCandidata != defenses.end() && !CEM_colocado) {
    Celda celda = Q.top();
    Q.pop();
    (*defensaCandidata)->position = celda.vector3_;
    if (factibilidad(defensesPlaced, mapWidth, mapHeight, (*defensaCandidata), obstacles)) {
        defensesPlaced.push_back((*defensaCandidata));
        ++defensaCandidata;
        CEM_colocado = true;
    }
  }
.
.
.
}
```

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

   En los algoritmos devoradores se distinguen los siguientes elementos:

   a) Un conjunto de **candidatos**. En nuestro caso, las celdas del mapa.

   b) Un conjunto de **candidatos seleccionados**. En nuestro caso, las celdas del mapa seleccionadas para colocar nuestras defensas y/o el C.E.M.

   c) Una **función solución** que comprueba si un conjunto de candidatos es una solución (posiblemente no óptima). En nuestro caso, haber colocado todas las defensas disponibles, incluida la primera de todas que es el C.E.M

   d) Una **función de selección** que indica el candidato más prometedor de los que quedan. En nuestro caso, seleccionará la celda más prometedora, que es la mejor valorada extrayendo cada vez que toque la primera posición de la cola de prioridad.

e) Una **función de factibilidad** que comprueba si un conjunto de candidatos se puede ampliar para obtener una solución (no necesariamente óptima). En nuestro caso, la celda se podrá colocar cuando esté en una posición dentro del mapa y además no choque con una defensa ni con un obstáculo.

f) Una **función objetivo** que asocia un valor a una solución, y que queremos optimizar. En nuestro caso, el tiempo que emplean los ucos en destruir nuestro C.E.M.

g) Un **objetivo**. En nuestro caso, maxizar ese tiempo.

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.

   Sabemos que el C.E.M es la primera posición de la lista de defensas colocadas, por lo que obtenemos esa celda mirando la primera posición de esa lista. A través de la función positionToCell(...) conseguimos sacar la fila i y la columna j en la que está nuestro C.E.M. Y sacamos la distancia de la celda candidata a nuestro C.E.M. Posteriormente calculamos la distancia de esa celda candidata con nuestro C.E.M. Cuanto menor sea la distancia, mejor. Con esto haremos que las defensas rodeen a nuestro C.E.M protegiéndolo.

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 ComparaValor2 {
  bool operator()(Celda const &celda1, Celda const &celda2) {
    // Cuanto menor valor, mayor prioridad.
    // Con esto conseguimos que las celdas con menor distancia al CEM tengan mas prioridad.
    return celda1.valor_ < celda2.valor_;
  }
};
```

```
int cellValue(int row, int col, bool **freeCells, int nCellsWidth, int nCellsHeight, float
    mapWidth, float mapHeight, const std::list<Object *> &obstacles, std::list<Defense *>
    defenses, bool esCEM) {
  .
  .
  .
  if (esCEM) {
      .
      .
      .
  } else {
    int i_out;
    int j_out;
    std::list<Defense *>::const_iterator ci_CEM = defenses.begin();
    Vector3 aux_vector3;
    positionToCell((*ci_CEM)->position, i_out, j_out, cellWidth, cellHeight);
    aux_vector3 = cellCenterToPosition(row - i_out, col - j_out, cellWidth, cellHeight);
    return std::max(mapWidth, mapHeight) - aux_vector3.length();
  }
}
```

```
void DEF_LIB_EXPORTED
placeDefenses(bool **freeCells, int nCellsWidth, int nCellsHeight, float mapWidth, float
    mapHeight, std::list<Object *> obstacles, std::list<Defense *> defenses) {
    .
    .
    .
// rest of our defenses
std::priority_queue<Celda, std::vector<Celda>, ComparaValor2> Q2; // nueva cola de prioridad
 for (int i = 0; i < nCellsHeight; i++) {
  for (int j = 0; j < nCellsWidth; j++) {
    aux_vector3 = cellCenterToPosition(i, j, cellWidth, cellHeight);
    Celda aux_celda = Celda(
```

```
        cellValue(i, j, freeCells, nCellsWidth, nCellsHeight, mapWidth,  mapHeight, obstacles
            , defenses, false),
        aux_vector3
        );
    Q2.push(aux_celda);
    }
 }

  while (defensaCandidata != defenses.end()) {
    Celda celda = Q2.top();
    Q2.pop();
    (*defensaCandidata)->position = celda.vector3_;
    if (factibilidad(defensesPlaced, mapWidth, mapHeight, (*defensaCandidata), obstacles)) {
      defensesPlaced.push_back((*defensaCandidata));
      ++defensaCandidata;
    }
  }
}
```

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.