INFORME

Trabajo Final

COMPLEJIDAD TEMPORAL Y ESTRUCTURAS DE DATOS

AUTOR

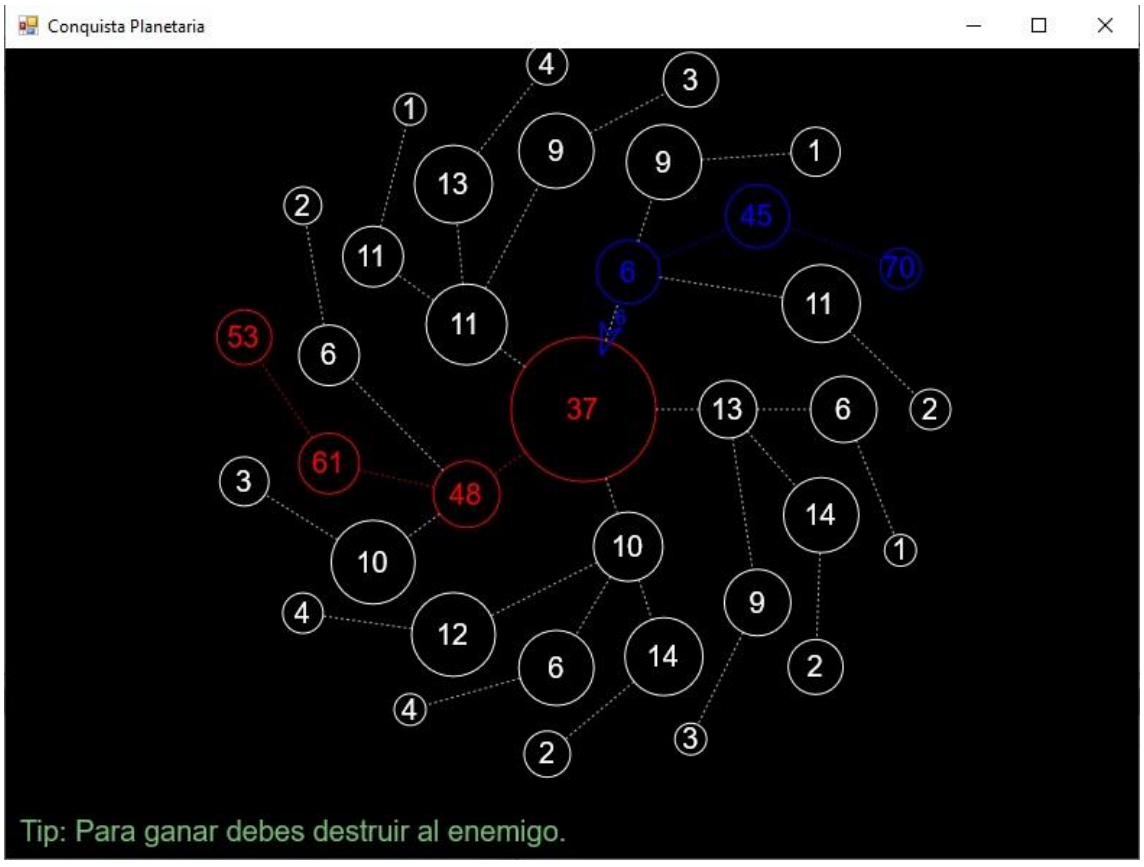
Andrés Báez

Contenido

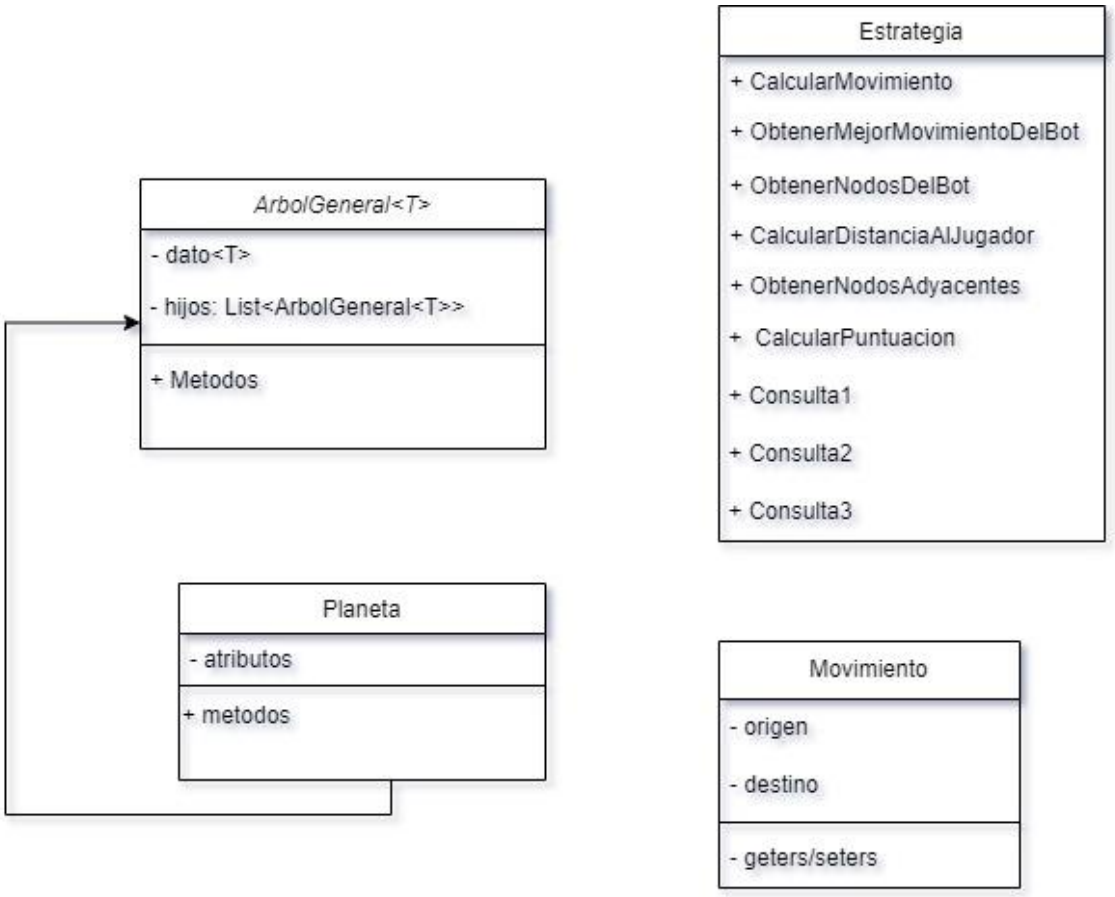
Introducción

El proyecto consiste en un juego realizado mediante una aplicación en C# que consiste en un jugador y un bot que se mueven conquistando diferentes planetas dispuestos en forma de árbol. Los diferentes planetas pueden ser del jugador (en rojo), del bot (en azul) o neutrales (en blanco). Cuando el bot o el jugador ataca a otro planeta, la mitad de la población de dicho planeta viaja en las naves, a la población del planeta atacado se le resta dicha mitad la población del planeta atacante y el planeta llega a ser conquistado cuando la población llega a cero, pasando a convertirse en un planeta del atacante (del bot o del jugador según el caso). Cuando un planeta es conquistado sea tanto del bot o del jugador, cambia al color correspondiente y comienza a aumentar su población según una taza de crecimiento del planeta, además un planeta puede enviar población a otro planeta suyo.

Captura del juego (el centro es la raíz del árbol):



Diagramas UML de la aplicación



- Estrategia es la clase principal que representa la estrategia del juego del bot.
- ArbolGeneral<T> es una clase genérica que implementa un árbol general y se utiliza para modelar la jerarquía de planetas.
- Nodo<T> es una clase que representa los nodos del árbol y contiene un dato y una lista de hijos.
- Planeta es una clase que representa un planeta en el juego.
- Movimiento representa un movimiento de un planeta a otro

Las relaciones entre las clases están representadas por las líneas en el diagrama. La clase ArbolGeneral<T> contiene nodos, y los nodos pueden contener datos de cualquier tipo (Planeta en este caso).
En cada uno de los métodos de la clase estrategia pasa un árbol con los planetas, que

representa el estado del juego.

Para la implementación de la clase estrategia se requirieron muchos métodos de la clase, es decir privados, que permitieran resolver una parte del problema y dieran mejor legibilidad al código.
Para la toma de decisión de cada movimiento, el código de estrategia, toma todos los nodos del bot y para cada uno de ello le da una puntuación a cada adyacente. Cada adyacente a un nodo del bot es un potencial planeta atacado.
El planeta con mejor puntaje se convierte en el objetivo del bot.
Para calcular el puntaje se tomaron en cuenta varios factores:

- Si el planeta adyacente es otro planeta del bot, debe tener poco puntaje.
- Si el planeta adyacente es un planeta del enemigo (el jugador), debe tener una muy alta puntuación.
- Si el planeta adyacente es un planeta neutral, se toma en consideración si está en el camino hacia un nodo enemigo y si la posibilidad de ser conquistado comparando las poblaciones es alta.

Decisión de mejor planeta:

En la siguiente captura se puede ver la parte de código correspondiente a el recorrido de los adyacentes a un nodo del bot (pueden ser varios nodos del bot). Y la toma de decisión del mejor nodo según su puntuación.

```
foreach (ArbolGeneral<Planeta> nodoAdyacente in nodosAdyacentes)
{
    // Calcular la distancia al jugador
    ArbolGeneral<Planeta> nodoJugadorCercano = ObtenerJugadorMasCercano(arbol, nodoAdyacente);
    int distanciaAlJugador = CalcularDistanciaAlJugador(arbol, nodoBot, nodoJugadorCercano);
    puntuacion = 0;

    if(nodoAdyacente.getDatoRaiz().EsPlanetaNeutral()){
        int poblacionAdyacente = nodoAdyacente.getDatoRaiz().Poblacion();
        int poblacionBot = nodoBot.getDatoRaiz().Poblacion();
        poblacionObtenida = poblacionBot - poblacionAdyacente ;
        // Calcular la puntuación considerando la distancia al jugador y la población del planeta enemigo
        puntuacion = CalcularPuntuacion(distanciaAlJugador, (poblacionObtenida));
    }

    if (nodoAdyacente.getDatoRaiz().EsPlanetaDeLaIA()){
        puntuacion = puntuacion-int.MaxValue; //Si es planeta de la IA no es tan importante
    }
    if (nodoAdyacente.getDatoRaiz().EsPlanetaDelJugador()){
        poblacionObtenida = 9999999;    //Forzamos que el criterio de la poblacion para que deje de ser tan relevante.
        // Calcular la puntuación considerando la distancia al jugador y la población del planeta enemigo
        puntuacion = CalcularPuntuacion(distanciaAlJugador, (poblacionObtenida));
    }
    // Elegir el adyacente con mayor puntuación
    if (puntuacion >= mejorPuntuacion)
    {
        objetivo = nodoAdyacente;
        mejorPuntuacion = puntuacion;
        mejorNodo = nodoBot;
    }
```

Cálculo de puntuación del planeta:

Para calcular la puntuación se utiliza un método que recibe que distantica tiene el nodo adyacente hacia un nodo del jugador y cuanto sería la población obtenida de la resta del nodo del bot y el nodo adyacente. Con eso dos datos, población obtenida y distancia, se calcula la puntuación del planeta. Si la distancia es muy chica y la población obtenida es muy alta, ese será el planeta con mejor puntuación. Se le da prioridad a la distancia con un 60% y la población obtenida un 40%

```
private double CalcularPuntuacion(int distanciaAlJugador, int poblacionObtenida)
{

    // Ajusta los pesos según tus necesidades, en este caso se prioriza la distancia en 60%
    double pesoDistancia = 0.6;  // Puedes experimentar con diferentes valores
    double pesoPoblacion = 0.4;  // Puedes experimentar con diferentes valores

    // Calcula la puntuación combinando la distancia y la población
    // En este caso, se suma la población, ya que queremos que una población más grande incremente la puntuación
    return pesoDistancia / (1.0 + distanciaAlJugador) + (pesoPoblacion * poblacionObtenida);
}
```

Obtener Adyacentes:

Para obtener los adyacentes al nodo del bot se utiliza un código que devuelve una lista con el padre y/o los hijos del nodo si es que los tiene.

```
private List<ArbolGeneral<Planeta>> ObtenerNodosAdyacentes(ArbolGeneral<Planeta> arbol, ArbolGeneral<Planeta> nodo)
{
    // Implementar la lógica para obtener todos los nodos adyacentes (padre e hijos) al nodo dado
    // Puedes utilizar algoritmos de recorrido en el árbol.

    List<ArbolGeneral<Planeta>> nodosAdyacentes = new List<ArbolGeneral<Planeta>>();

    if (nodo != null)
    {
        // Agregar los hijos
        if(nodo.getHijos() != null){
            nodosAdyacentes.AddRange(nodo.getHijos());
        }
        // Agregar el padre
        ArbolGeneral<Planeta> padre = ObtenerPadre(arbol, nodo);
        if (padre != null)
        {
            nodosAdyacentes.Add(padre);
        }
    }

    return nodosAdyacentes;
}
```

Consultas

Además, la aplicación presenta tres consultas que se pueden ejecutar antes de iniciar la partida, que corresponden a recorridos del árbol.

Consulta 1:

La consulta 1 obtiene todos los nodos del bot y luego por cada uno de ellos imprime sus distancias hasta la raíz.

```
public string Consulta1(ArbolGeneral<Planeta> arbol)
{
    //Calcula y retorna un texto con la distancia del camino que existe entre el planeta del Bot y la raíz.
    List<ArbolGeneral<Planeta>> nodosDelBot = ObtenerNodosDelBot(arbol);
    string resultado = "";

    foreach (ArbolGeneral<Planeta> nodoBot in nodosDelBot)
    {
        // Calcular la distancia desde el nodo del bot hasta la raíz
        int distancia = ObtenerProfundidad(arbol, nodoBot);

        // Agregar la información al resultado
        resultado += "Distancia desde la raíz hasta el planeta del Bot: " + distancia + "\n";
    }

    return resultado;
}
```

Para obtener los nodos del bot se recorre todo el arbol y si el nodo es un nodo del bot se agrega a una lista que luego se retorna como parametro.

```
private List<ArbolGeneral<Planeta>> ObtenerNodosDelBot(ArbolGeneral<Planeta> arbol)
{
    List<ArbolGeneral<Planeta>> nodosDelBot = new List<ArbolGeneral<Planeta>>();

    if (arbol == null)
    {
        return nodosDelBot;
    }

    Cola<ArbolGeneral<Planeta>> cola = new Cola<ArbolGeneral<Planeta>>();
    cola.encolar(arbol);

    while (!cola.esVacia())
    {
        ArbolGeneral<Planeta> nodoActual = cola.desencolar();

        // Verificar si el nodo actual pertenece a la IA (Bot)
        if (nodoActual.getDatoRaiz().EsPlanetaDeLaIA())
        {
            nodosDelBot.Add(nodoActual);
        }

        // Agregar todos los hijos del nodo actual a la cola
        foreach (ArbolGeneral<Planeta> hijo in nodoActual.getHijos())
        {
            cola.encolar(hijo);
        }
    }

    return nodosDelBot;
}
```

Para obtener la profundidad, se comienza a contar en cero y luego se cuentan los padres de cada nodo hasta llegar a un nodo que no tiene padre, o sea, la raíz.

```
private int ObtenerProfundidad(ArbolGeneral<Planeta> arbol, ArbolGeneral<Planeta> nodo)
{
    int profundidad = 0;

    while (nodo != null && nodo != arbol)
    {
        nodo = ObtenerPadre(arbol, nodo);
        profundidad++;
    }

    return profundidad;
}
```

Para obtener padre se verifica que el nodo “sin su padre” no es la raíz del arbol, luego se busca en los hijos del nodo arbol y si se encuentra significa que el nodo arbol es el padre, si no se encuentra, se repite el procedimiento pero pasando cada nodo hijo como raíz del arbol.

```
private ArbolGeneral<Planeta> ObtenerPadre(ArbolGeneral<Planeta> arbol, ArbolGeneral<Planeta> nodoBuscado)
{
    if (arbol == null || arbol == nodoBuscado)
    {
        return null;  // El árbol es nulo o el nodo buscado es la raíz, por lo que no tiene un padre
    }

    foreach (ArbolGeneral<Planeta> hijo in arbol.getHijos())
    {
        if (hijo == nodoBuscado)
        {
            return arbol;  // Se encontró el padre
        }

        // Buscar recursivamente en los hijos del hijo actual
        ArbolGeneral<Planeta> padreEnSubarbol = ObtenerPadre(hijo, nodoBuscado);
        if (padreEnSubarbol != null)
        {
            return padreEnSubarbol;  // Se encontró el padre en el subárbol
        }
    }

    return null;  // No se encontró el nodo buscado en los hijos del árbol actual
}
```

Consulta 2:

La consulta 2 obtiene todos los nodos del bot y por cada nodo del bot imprime sus nodos descendientes.

```
public string Consulta2(ArbolGeneral<Planeta> arbol)
{
    //Retorna un texto con el listado de los planetas ubicados en todos los descendientes del n
    string planetasDescendientes = "";
    List<ArbolGeneral<Planeta>> NodosDelBot = ObtenerNodosDelBot(arbol);
    foreach(ArbolGeneral<Planeta> Nodo in NodosDelBot)
    {
        List<Planeta> descendientes = ObtenerDescendientes(arbol, Nodo);


        foreach(Planeta p in descendientes){
            planetasDescendientes = planetasDescendientes +
            ("Planeta: (x="+ p.position.X.ToString()+"; y="+ p.position.Y.ToString()+ "); ");
        }
    }

    return "Descendientes del Bot: " + planetasDescendientes;
}
```

Consulta 3:

La consulta 3 calcula la población promedio por cada nivel del árbol y guarda la información en un string que retorna al final del programa.

```
public string Consulta3(ArbolGeneral<Planeta> arbol)
{
    //Calcula y retorna en un texto la población total y promedio por cada nivel del árbol.
    Dictionary<int, Tuple<int, int>> poblacionPorNivel = CalcularPoblacionPorNivel(arbol);

    string resultado = "Población por nivel:\n";

    foreach (var nivel in poblacionPorNivel)
    {
        resultado +="Nivel "+nivel.Key+": Población total = "+nivel.Value.Item1+", Población promedio = "+nivel.Value.Item2+"\n";
    }

    return resultado;
}
```

Posibles mejoras:

Al tener que hacer recorridos en diferentes direcciones del árbol, y obtener caminos entre nodos, quizás una posible mejora del código podría ser, implementar el árbol de estado de los planetas como un grafo. Si bien esto quizás no varie mucho en la ejecución de la aplicación si podría hacer el código más coherente y claro para representar el funcionamiento del juego.
También se podría modificar la clase árbol haciendo que algún dato del mismo no requiera un recorrido del árbol, por ejemplo, incluir un acceso directo a un nodo padre.
Otra posible mejora podría ser mejorar la interfaz gráfica incluyendo imágenes más vistosas.
Podría también tomarse otras consideraciones a la hora de elegir un planeta a donde dirigir la nave, como por ejemplo si el planeta está en peligro de ser conquistado por su baja población.

Conclusión

En el presente trabajo se pudo ver en profundidad la estructura de datos “Árbol general” y sus diferentes recorridos y operaciones sobre los mismos.
Es conveniente siempre que se utilizan estructuras de datos complejas y que requieren recursividad, la utilización de una buena modularización y nombres para los métodos y variables que sean descriptivos de su funcionalidad, junto suficientes comentarios que permita seguir correctamente la funcionalidad del código.

En el trabajo realizado muchas veces era difícil encontrar un bug en tiempo de ejecución debido a la complejidad del código, y al incluir parte del código que ya estaba implementada correctamente y de la cual no se tuvo mucha información sobre su funcionamiento e implementación desde el principio, por lo cual su implementación era preferible no modificarla.

Finalmente también puede decirse que en el desarrollo del trabajo se pudo obtener experiencia sobre códigos que implican tomar una decisión durante su ejecución.