Práctica final – Jugador autómatico para “¿Quién es quién?”

Esta práctica consiste en implementar un jugador autómatico para el juego “¿Quién es quién?” que podrá adivinar el personaje de un tablero de personajes según las respuestas que le vayamos dando. La solución propuesta para este problema viene conjunta con este documento. Se ha pedido que se implementen 4 métodos obligatorios (además de eliminar_nodos_redundantes) y 3 métodos opcionales. A continuación, pasaré a explicar como funciona cada uno de ellos:

void crear_arbol()

Este método crea un árbol de preguntas para todos los personajes del tablero. Una vez leído el tablero de un fichero y guardado en la variable tablero de la clase QuienEsQuien, lo usaremos para ir creando los nodos del árbol. En primer lugar, creamos una Pregunta con el atributo 0 y el número de personajes total y creamos un árbol con esta Pregunta como etiqueta del nodo raíz. A partir de aquí el procedimiento sigue un patrón.

```
vector<int> vector_l;
vector<int> vector_r;

for (int j = 0; j < tablero.size(); ++j) {

        if (tablero [j][0] == 1)
                vector_l.push_back (j);

        else
                vector_r.push_back(j);
}
```

Con este trozo de código separamos en un “vector izquierda” los índices de los personajes que contestan que sí a tal atributo y en un “vector derecha” los de los personajes que no. Luego creamos una pregunta con el siguiente atributo y el tamaño del “vector izquierda” e insertamos un nodo a la izquierda del nodo raíz con esta pregunta como etiqueta. Tras esto, llamamos a una función auxiliar llama “crear_nodo” que hemos creado como nueva y que trata sobre ir creando nodos de una manera recursiva teniendo en cuenta el tablero. A esta función le pasamos el vector con los índices de los personajes para el nodo (ya sea “vector izquierda” o “vector derecha”), el nuevo nodo que hemos creado y la profundidad del mismo. Luego hacemos lo mismo con “vector derecha”.

Dentro de esta función se trata de básicamente lo mismo, separar en derecha o izquierda según si el personaje responde si o no, solo que si queda todavía más de un personaje restante, se vuelve a llamar a la misma función con nuevos argumentos. En caso de que quede solo uno, únicamente se insertaría el nodo.

void iniciar_juego()

Esta función implementa el desarrollo de una partida. Simplemente el jugador automático le pide una respuesta al usuario humano por pantalla y, en relación con esto, se va desplazando por el árbol hasta llegar al personaje correcto. Si no hay personajes con tales atributos, muestra por pantalla que ese personaje no existe.

set<string> informacion_jugada (bintree<Pregunta>::node jugada)

Este método devuelve un set<string> con los personajes que no han sido descartados todavía según las respuestas que se han dado. El algoritmo es sencillo. Primero creamos un subárbol a partir del árbol general y el nodo dado como argumento. Y a partir de ahí con un iterator, vamos guardando los personajes de los nodos hoja en un set<string> local.

float profundidad_promedio_hojas()

Este método calcula una profundidad media de los nodos hoja del árbol. Consiste en recorrer el árbol y, si el nodo actual es un nodo hoja, se calcula la profundidad de dicho nodo, se le suma a una variable “suma” y se aumenta contador en 1. Al final se devuelve dicha suma dividida entre el número de nodos hoja (contador).

void aniade_personaje (string nombre, vector<bool> caracteristicas)

Añade un personaje nuevo en el árbol según las características de dicho personaje. El algoritmo consiste en ir avanzando en el árbol según las características del nuevo personaje (a medida que se va avanzando, se va actualizando el número de personajes de cada nodo, aumentándolo en 1), hasta llegar a un nodo hoja (o a un nodo nulo).

Si llegamos a un nodo hoja existente, tendremos que distinguir el personaje de dicho nodo del nuevo. Para ello, guardamos este personaje en una variable auxiliar para no perderlo e introducimos como nueva etiqueta del nodo hoja una pregunta con el siguiente atributo a tener en cuenta y con num_personajes = 2. Y dependiendo de las características de cada uno, lo insertamos a la izquierda o a la derecha de este nodo.

Si no llegamos a ningún nodo hoja se inserta el personaje directamente donde le corresponde.

Además, se actualizan los vectores “personajes” y “tablero”.

void elimina_personaje (string nombre)

Es similar al método anterior pero al contrario. Primero se comprueba que el personaje existe en el vector de personajes. Si no, muestra un mensaje por pantalla.

Si existe, se vuelve a hacer lo de antes. Se va recorriendo el árbol hasta llegar al nodo del personaje, actualizando a medida que se avanza el número de personajes restantes (ahora se les resta 1).

Tras llegar al nodo hoja, le hacemos un prune_right() o un prune_left() al padre (guardamos el subarbol en una variable) y se actualizan los vectores “personajes” y “tablero”.

void QuienEsQuien::eliminar_nodos_redundantes()

Método para eliminar preguntas inútiles dentro del árbol.

Comprueba si cada nodo que no sea la raíz tiene solo un hijo, y acopla el subarbol del único hijo como el nuevo padre. Si es la raíz, comprueba si tiene una pregunta inútil, y si es así coge como nuevo árbol el subarbol del hijo válido de la raíz.

void preguntas_formuladas (bintree<Pregunta>::node jugada)

Este método te muestra las preguntas que ya se han hecho y las respuestas que ha dado el jugador humano.

Primero creamos dos nodos, uno lo igualamos a jugada y el otro a jugada.parent() (evitamos que jugada sea igual a root()) y creamos también un set<string>. Si jugada es el hijo izquierdo de su padre, insertamos en el set la pregunta + “ - si”. Si es el hijo derecho, en vez de “ - si”, metemos “ - no”. Y si el padre de jugada.parent() no es nulo, igualamos este a su padre y jugada a su padre también, entrando de nuevo en el bucle. Finalmente, mostramos por pantalla el resultado.