Juego del Quién es Quién

Estructura de datos

Realizado por: Daniel Díaz Pareja
Fecha: 07/01/2018
Universidad de Granada

Índice

1. Introducción....3

2. Métodos básicos....3

2.1 Construcción básica del árbol de preguntas....3

2.2 Omitir preguntas inútiles....6

2.3 Iniciar juego....7

2.4 Obtener información de la jugada....8

2.5 Profundidad promedio de las hojas (calidad del árbol)....9

3. Bibliografía....11

1. Introducción.

En este documento se describe la solución propuesta al problema de realizar un jugador automático para el juego "¿Quién es Quién?" mediante un árbol binario de búsqueda. Se discutirá el funcionamiento de la implementación de los métodos más relevantes del mismo.

2. Métodos básicos.

2.1 Construcción básica del árbol de preguntas.

Para construir el árbol de preguntas se ha seguido la siguiente estrategia:

Primero, para saber el número de personajes que quedan por levantar (el campo "string num_personajes" de la clase Pregunta), se ha creado una función que, dada una sucesión de respuestas a las preguntas sobre los atributos de personajes (true, false, true, true...), devuelve un set<string> con los personajes que quedan por levantar, es decir, los que coinciden con dicho camino.

El funcionamiento es simple, un bucle que recorre el tablero de personajes y atributos por cada personaje. Si al acabar un personaje este coincide con la sucesión de respuestas, se añade, si no, no se añade.

Código fuente:

```
1  set<string> QuienEsQuien::personajes_coincidentes(const vector<bool> &
2  respuestas)
3  {
4          set<string> personajes_coincidentes;
5          for (int personaje = 0; personaje < personajes.size(); personaje++){
6                  int atrib = 0;
7                  bool coincidencia = true;
8                  while (coincidencia && atrib < respuestas.size()){
9                          if (respuestas[atrib] != tablero[personaje][atrib])
10                                 coincidencia = false;
11                         atrib++;
12                 }
13
14                 if (coincidencia)
15                         personajes_coincidentes.insert(personajes[personaje]);
16         }
17
18         return personajes_coincidentes;
   }
```

Por otra parte, para construir el árbol, se ha construido un algoritmo recursivo con la siguiente cabecera:

```
1 void QuienEsQuien::construccion_basica(bintree<Pregunta> & arbol,
2         const bintree<Pregunta>::node & nodo_actual,
3         vector<bool> & camino, const int & num_atributo)
```

Donde “arbol” y “nodo_actual” son el árbol y el nodo a partir del cual se creará el árbol de preguntas. Camino es la sucesión de respuestas por la que vamos actualmente (true, false, true...), y num_atributo es el atributo por el que se está preguntando actualmente.

Funciona de la siguiente manera:

Primero, el camino está vacío. Se empieza a procesar la parte izquierda, es decir, la parte del sí, por lo que se inserta el valor “true” al vector “camino”.

A continuación, se comprueba el número de personajes que coinciden con el método anteriormente comentado. Si el número de coincidencias es mayor que 1, significa que queda más de un personaje por averiguar. Entonces, se añade el nodo con la pregunta por la que vamos y con dicho número. Una vez hecho esto, hay que seguir expandiendo el nodo, es decir, se llama recursivamente a este método construccion_basica con el árbol, con el nuevo nodo (nodo_actual.left()), con el nuevo camino y con el siguiente número de atributo (hay que preguntar por otra cosa).

Si el número de personajes coincidentes es 1, hemos llegado a un personaje y no hay que actuar recursivamente, se añade el nodo con el único valor del set<string> que devuelve el método preguntar_coincidencias.

Una vez pasamos este tramo, se saca el último valor del vector<bool> camino, ya que estamos volviendo al padre.

Ahora, para procesar la parte derecha (la del no), se sigue la misma estrategia, pero obviamente se inserta el valor “false” en el camino, al añadir los nodos se hace en la parte derecha y al hacer la recursividad se hace con nodo_actual.right().

Al terminar este algoritmo, el árbol queda construido.

Código fuente:

```
 1 void QuienEsQuien::construccion_basica(bintree<Pregunta> & arbol,
 2         const bintree<Pregunta>::node & nodo_actual,
 3         vector<bool> & camino, const int & num_atributo)
 4 {
 5
 6         // rama izquierda
 7         camino.push_back(true);
 8         set<string> coincidencias = personajes_coincidentes(camino);
 9         int num_coincidencias = coincidencias.size();
10         Pregunta nuevo_nodo;
11
12         if (num_coincidencias > 1) // Queda más de un personaje por averiguar
13         {
14            nuevo_nodo = Pregunta(atributos[num_atributo], num_coincidencias);
15            arbol.insert_left(nodo_actual, nuevo_nodo);
16            // Seguimos procesando por la izq
17            construccion_basica(arbol, nodo_actual.left(), camino, num_atributo+1);
18         }
19         else if (num_coincidencias == 1) // Queda 1 personaje, es una hoja
20         {
21            nuevo_nodo = Pregunta(*coincidencias.begin(), 1);
22            arbol.insert_left(nodo_actual, nuevo_nodo);
23         }
24
25         camino.pop_back();
26
27         // rama derecha
28         camino.push_back(false);
29         coincidencias = personajes_coincidentes(camino);
30         num_coincidencias = coincidencias.size();
31
32         if (num_coincidencias > 1) // Queda más de un personaje por averiguar
33         {
34            nuevo_nodo = Pregunta(atributos[num_atributo], num_coincidencias);
35            arbol.insert_right(nodo_actual, nuevo_nodo);
36            // Seguimos procesando por la der
37            construccion_basica(arbol, nodo_actual.right(), camino, num_atributo+1);
38         }
39         else if (num_coincidencias == 1) // Queda 1 personaje, es una hoja
40         {
41             nuevo_nodo = Pregunta(*coincidencias.begin(), 1);
42             arbol.insert_right(nodo_actual, nuevo_nodo);
43         }
44
45         camino.pop_back();
46 }
```

Y el código fuente del método que llama a este algoritmo recursivo:

```
 1 bintree<Pregunta> QuienEsQuien::crear_arbol()
 2 {
 3         // Creamos el primer nodo del árbol
 4         Pregunta p_raiz(atributos[0], personajes.size());
 5         bintree<Pregunta> arbol(p_raiz);
 6 
 7         // Creamos el resto del árbol de manera recursiva
 8         vector<bool> camino;
 9         construccion_basica(arbol, arbol.root(), camino, 1);
10 
11         return arbol;
12 }
```

2.2 Omitir preguntas inútiles.

Para eliminar las preguntas inútiles, es decir, para eliminar los nodos redundantes (nodos con un solo hijo) se ha optado por un algoritmo iterativo que funciona de la siguiente manera:

Partiendo de un nodo n, que al principio es la raiz del árbol de preguntas, y mientras dicho nodo no sea nulo, detectamos si tiene un solo hijo a la izquierda o a la derecha. Si este es el caso, podamos la parte no nula en un subarbol. Una vez hecho esto, reemplazamos el nodo n con la parte podada, con lo que se pierde la pregunta redundante y directamente se hace la pregunta de su hijo no nulo.

Si detectamos que tiene dos hijos, el nodo n se convierte en el siguiente nodo en preorden, que se selecciona con un algoritmo iterativo (similar al que hace el preorder_iterator).

Código fuente:

```
 1 void QuienEsQuien::eliminar_nodos_redundantes(){
 2         bintree<Pregunta>::node n = arbol.root();
 3         bintree<Pregunta> subarbol;
 4         while(!n.null())
 5         {
 6            // solo un hijo a la izquierda
 7            if (!n.left().null() && n.right().null())
 8            {
 9               arbol.prune_left(n,subarbol
10               arbol.replace_subtree(n, subarbol, subarbol.root());
11            }
12            // solo un hijo a la derecha
13            else if (!n.right().null() && n.left().null())
```

```
14              {
15                  arbol.prune_right(n,subarbol);
16                  arbol.replace_subtree(n, subarbol, subarbol.root());
17              } else {
18                  // Seleccionamos el siguiente nodo en preorden
19                  if (!n.left().null())
20                      n = n.left();
21                  else if (!n.right().null())
22                      n = n.right();
23                  else {
24                      while (!n.parent().null() &&
25                        (n.parent().right() == n || n.parent().right().null()))
26                         n = n.parent();
27
28                         if (n.parent().null())
29                             n = typename bintree<Pregunta>::node();
30                         else
31                             n = n.parent().right();
32                  }
33              }
34          }
    }
```

2.3 Iniciar juego.

Para iniciar el juego, simplemente tenemos que recorrer el árbol en función de las respuestas introducidas por teclado. La jugada actual se guarda en un atributo de clase, y va cambiando según se recorre el árbol hasta llegar a un nodo hoja. Si el jugador introduce una “s” como respuesta, el nuevo nodo será jugada_actual.left(), es decir, el nodo izquierdo, si no, será el nodo derecho.

Por ver en funcionamiento el método de obtener información de la jugada que comentaremos en el siguiente apartado, se ha incluido una llamada en este método iniciar juego que, por cada respuesta, nos dice los personajes restantes que quedan levantados.

Código fuente:

```
1 void QuienEsQuien::iniciar_juego(){
2     jugada_actual = arbol.root();
3     char respuesta;
4     set<string> personajes_restantes;
5     set<string>::const_iterator it;
6     while (!(*jugada_actual).es_personaje())
7     {
8        cout << endl << (*jugada_actual) << endl;
9        cout << "Respuesta(s/n): ";
```

```
            cin >> respuesta;
10
11          if (respuesta == 's')
12              jugada_actual = jugada_actual.left();
13          else
14              jugada_actual = jugada_actual.right();
15
16          personajes_restantes = informacion_jugada(jugada_actual);
17          cout << "Puede ser uno de estos personajes: ";
18          for (it = personajes_restantes.begin();
19              it != personajes_restantes.end(); ++it)
20               cout << *it << " ";
21
22               cout << endl;
23      }
24
25      cout << endl << "Tu personaje es: " <<
26     (*jugada_actual).obtener_personaje() << endl;
    }
```

2.4 Obtener información de la jugada.

Este método obtiene, en un set<string>, los personajes que quedan levantados en el tablero dada una jugada. Por comodidad y por hacer uso de los iteradores sobre un árbol, se ha hecho de la siguiente manera:

Primero, iteramos por el árbol de jugadas hasta encontrar el nodo cuya etiqueta sea igual a la etiqueta de la jugada que se pasa por argumento.

Una vez encontrado, creamos un sub-árbol a partir de este nodo.

A partir de aquí, encontrar los personajes es trivial usando, de nuevo, iteradores. Recorremos este sub-árbol con un iterador, y preguntamos si la etiqueta es un personaje. Si lo es, lo añadimos al set, y al terminar el recorrido devolvemos dicho set.

Código fuente:

```
1 set<string> QuienEsQuien::informacion_jugada(const bintree<Pregunta>::node
2 jugada_actual)
3 {
4
5     set<string> personajes_levantados;
6     bintree<Pregunta> sub_arbol;
7     bintree<Pregunta>::const_preorder_iterator it = arbol.begin_preorder();
8     bool encontrado = false;
9
```

```
        while (!encontrado)
10      {
11              if (*it == *jugada_actual){
12                   sub_arbol.assign_subtree(arbol, jugada_actual);
13                   encontrado = true;
14              }
15
16              ++it;
17      }
18
19      for (it = sub_arbol.begin_preorder(); it != sub_arbol.end_preorder(); ++it)
20          if ((*it).es_personaje())
21            personajes_levantados.insert((*it).obtener_personaje());
22
23      return personajes_levantados;
    }
```

2.5 Profundidad promedio de las hojas (calidad del árbol).

Para realizar esta operación, primero utilizamos un algoritmo recursivo para calcular la suma de las profundidades de todas las hojas, y después dividimos esta suma entre el número de hojas totales (el número de personajes).

El algoritmo para calcular la suma de las profundidades de las hojas tiene la siguiente cabecera:

```
1 void QuienEsQuien::suma_profundidades(const bintree<Pregunta>::node n,
2         int & prof_parcial, int & sum_prof)
```

Donde n es el nodo por el que vamos investigando, prof_parcial es la profundidad del nodo actual y sum_prof es la suma de las profundidades las hojas. El algoritmo funciona de la siguiente manera:

Nada más empezar el recorrido, sumamos 1 a la profundidad, ya que estamos investigando un nuevo nodo. A continuación, preguntamos si el nodo es un personaje. Si lo es, sumamos la profundidad parcial a la suma de las profundidades, sum_prof, pero restandole 1 para equilibrar el +1 que se hace en el nodo raíz.

Si no es un personaje, procesamos los nodos de la izquierda y la derecha (si alguno de ellos es nulo, no lo procesamos) llamando de nuevo al algoritmo recursivo. Una vez terminemos de procesar alguno de los nodos, hay que restar 1 a la profundidad parcial, ya que estamos subiendo al padre de nuevo.

Al finalizar el algoritmo, se habrán investigado todos los nodos y tendremos la suma de las profundidades de las hojas en la variable sum_prof.

Código fuente:

```
void QuienEsQuien::suma_profundidades(const bintree<Pregunta>::node n,
        int & prof_parcial, int & sum_prof)
{
   prof_parcial += 1;

   if ((*n).es_personaje())
        sum_prof+=prof_parcial-1;
        // -1 para equilibrar el +1 que se hace en el nodo raíz
   else{
        if (!n.left().null()){ // Procesamos la izquierda
             suma_profundidades(n.left(),prof_parcial,sum_prof);
             prof_parcial-=1; // subimos al padre
        }
        if (!n.right().null()){ // Procesamos la derecha
             suma_profundidades(n.right(),prof_parcial,sum_prof);
             prof_parcial-=1; // subimos al padre
        }
   }
}
```

Y el método que llama a este otro método y calcula y devuelve el promedio:

```
float QuienEsQuien::profundidad_promedio_hojas(){

   int sum_prof = 0;
   int sum_par = 0;
   suma_profundidades(arbol.root(),sum_par,sum_prof);

   return ((float)sum_prof/personajes.size());
}
```

3. Bibliografía.

- [1] “Abstracción y Estructuras de Datos en C++”, apuntes de clase del profesor Joaquín Fernández Valdivia, Departamento de Ciencias de la Computación, Universidad de Granada, Invierno 2017
- [2] C++ Reference: STL Containers, http://www.cplusplus.com/reference/stl/