TreeView: un generador de árboles para ayuda a la docencia

H. Navarro[1]

[1]Centro de Computación Gráfica, Universidad Central de Venezuela. hector.navarro@ciens.ucv.ve

RESUMEN

Los árboles como estructura de datos son muy usados y estudiados en las ciencias de la computación. En muchas materias del pensum de estudian los árboles, su estructura, y algoritmos básicos sobre estos. Muchas veces, para ayudar a su comprensión es de mucha ayuda el uso de figuras demostrando el estado de los árboles. La creación de estos árboles en distintos programas de edición gráfica puede volverse una tarea que consume tiempo y propensa a errores. TreeView es una herramienta que genera un árbol en formato SVG a partir de una archivo de texto plano sencillo que describe el árbol. En este trabajo se describe el algoritmo usado para desplegar los árboles y se muestran algunas pruebas hechas para demostrar la utilidad de TreeView.

ABSTRACT

Trees as a data structure are very used and studied in computer science. Tree data structures, and its associated algorithms are studied in several subjects on the pensum. In order to completely understand these structures it is useful the use of figures showing different stated of the trees. The creation of these trees in different graphics editing programs can be a very time consuming, error prone task. TreeView is a tool for generating trees in SVG format from a plain text file that describes the tree. In this work the algorithm used to render trees is described, and some tests demonstrating the capabiblities of TreeView are shown.

Keywords: Animación de Algoritmos, Árboles, Despliegue de árboles

1. Introducción

En muchas asignaturas de carreras afines a Ciencias de la Computación en donde se estudian algoritmos es necesario estudiar árboles que son estructuras de datos jerárquicas básicas. Para la mejor comprensión de parte del estudiante de estas estructuras de datos y los algoritmos asociados a ellas, es importante observar representaciones gráficas de los árboles que se están estudiando.

En la Escuela de Computación de la UCV muchas materias tienen material de apoyo creado por profesores y preparadores, que pueden servir de ayuda al estudiante para comprender los temas dictados. Generalmente, cuando se desea dibujar un árbol para incluirlo en estos materiales de apoyo, este dibujo se hace a mano con diversas herramientas gráficas como PowerPoint, InkScape, etc. Construir árboles muy complejos o varios árboles para mostrar diversas etapas de un algoritmo suele ser una tarea tediosa, repetitiva y que quita mucho tiempo. Por estas razones se decidió hacer un algoritmo que tome como entrada una especificación de un árbol en formato de texto y genere una o varias imágenes que representen gráficamente el árbol.

Entre los árboles a ser considerados se tienen árboles binarios, árboles generales, heaps, conjuntos disjuntos, heaps de Fibonacci, QuadTrees, OcTrees, etc. Hay una diversidad de características a tomar en cuenta como el número de hijos que cada nodo puede tener, el número de raíces que existen (en el caso de bosques), permitir resaltar un camino en particular del árbol, o resaltar nodos particulares, etc.

El autor desea agradecer al Profesor Esmitt Ramírez del Centro de Computación Gráfica, por sus valiosos comentarios que ayudaron en el mejoramiento de la herramienta aquí propuesta.

2. Trabajos previos

Un árbol es un grafo dirigido acíclico, en donde existe un nodo especial llamado raíz, el cuál no tiene arcos salientes [CLRS09].

Para dibujar un árbol correctamente ciertas reglas estéticas que deben cumplirse han sido establecidas [WS79, RT81]:

1. El árbol impone una distancia en los nodos, ningún nodo debe estar mas cerca a la raíz que ninguno de sus ancestros.
2. Los nodos en un mismo nivel del árbol deben estar sobre una línea recta. Todas las líneas rectas correspondientes a cada nivel deben ser paralelas.
3. El orden relativo de los nodos en cualquier nivel debe ser el mismo que en el orden transversal del árbol.
4. Para un árbol binario, el hijo izquierdo debe estar posi-

cionado a la izquierda de su padre, y el hijo derecho a su derecha.
5. Un padre debe estar centrado respecto a sus hijos.
6. Cada subárbol de un árbol debe estar dibujado de la misma forma, independientemente de el lugar en el que está.

Reingold y Tilford [RT81] describen un algoritmo divide y conquista para árboles binarios llamado algoritmo RT. La idea básica es dibujar cada sub-árbol por separado y luego unirlos lo mas cerca posible. El mayor problema de este algoritmo es que cuando un nodo tiene únicamente un hijo izquierdo o derecho este es dibujado justo debajo de su padre, haciendo imposible determinar si el nodo es un hijo izquierdo o derecho. Este algoritmo únicamente soporta árboles binarios.

El algoritmo BKW desarrollado por Bruggemann-Klein y Wood [BKW89] evita este problema desplazando hacia la izquierda o derecha el nodo para hacer notar el tipo de subárbol.

El algoritmo RT fue modificado por Luo para soportar árboles generales y etiquetas [Luo93]. El resultado es el algoritmo MRT el cual permite dibujar árboles con una cantidad variable de nodos hijos, y la colocación de etiquetas en diversas partes del árbol para resaltar información de interés en la figura resultante.

Animación de algoritmos se refiere a la creación de imágenes asociadas a los estados intermedios de un algoritmo con el fin de comprender mejor el comportamiento de estos algoritmos [FK02]. Estas técnicas han sido ampliamente usadas en la enseñanza de técnicas de programación y estructuras de datos [FK02].

3. Formato de entrada de árboles

El programa implementado provee clases para generar un archivo con extensión `.svg` que puede ser editado usando programas de gráficos vectoriales como Inkscape. La entrada para este programa puede ser un árbol generado en memoria o se puede cargar un árbol desde un archivo. Esta sección describe el formato que deben tener los archivos a cargar.

Cada archivo contiene un sólo árbol, y la relación entre distintos niveles está marcada por la identación. Cada línea del archivo puede contener un nodo del árbol. El número de espacios en blanco (identación) antes del valor del nodo, indica el nivel en el que está el nodo. De esta forma, un árbol como el que se observa en la Figura 1 se describe como:

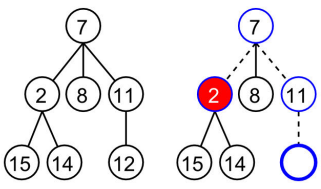

Figura 1: *Árbol sencillo con 7 nodos. El lado izquierdo muestra el árbol sin ningún atributo, del lado derecho se modifican algunos atributos del árbol*

```
7
 2
  15
  14
 8
 11
  12
```

Cada nodo puede tener asociado ciertos atributos como un identificador (`id`), color de texto (`textcolor`), texto en negritas (`bold`), color de relleno (`fill`), o una marca para indicar que no tienen ninguna etiqueta asociada (`nt`). Estos atributos aparecen justo al lado de la etiqueta entre paréntesis. Por ejemplo: `7(id=8 textcolor=0x00ff00 bold)` indica que el nodo con etiqueta 7 tiene identificador 1, color de texto `0x00ff00` (rojo), y aparece resaltado (en negritas). Los identificadores sirven para poder hacer referencias posteriores al nodo. Por ejemplo, al finalizar la definición del árbol es posible definir un camino que será resaltado en el árbol. Para definir este camino se deben indicar los dos nodos entre los cuales está definido el camino. Por ejemplo, supongamos el árbol de la Figura 1, y supongamos que el nodo 2 tiene las propiedades `(id=1 fill=0xff0000 textcolor=0xffffff)` y el nodo 12 las propiedades `(id=2 bold nt)`. Podemos ahora definir un camino entre estos dos nodos usando sus identificadores (1 y 2, no confundir con las etiquetas 2 y 12). Este camino se define así:

```
path 1 2 (dashed 0x0000ff)
```

Lo cual quiere decir que se desea dibujar todo el camino entre los nodos 1 y 2 usando líneas punteadas (`dashed`), y usando el color azul (`0x0000ff`) para dibujar los nodos. Es importante resaltar que todos los nodos involucrados en el camino van a tener estas propiedades (líneas punteadas y nodos de color azul). Esto es, los nodos 2, 7, 11 y 12. El nodo 12 no tiene ninguna etiqueta debido a que tiene la propiedad `nt`.

Es posible tener nodos invisibles, los cuales serán explicados en detalle mas adelante. Un nodo invisible ocupa cierto espacio en el dibujo (el espacio que ocuparía el nodo si estuviera visible), pero el dibujo del nodo no se genera. Para nodos invisibles en lugar de escribir la etiqueta correspondiente al nodo se escribe un guión "–".

Los archivos de entrada pueden tener comentarios los cuales comienzan con el caracter "#" y terminan hasta el final de la línea.

Por la sencillez del formato de archivo de entrada, el análisis sintáctico del archivo de definición de árboles es muy simple. El archivo se lee línea a línea, eliminando los comentarios y llevando el control de identación para saber a que nivel pertenece el nodo actual. Una vez leída la etiqueta del nodo, se leen los atributos (si existen). El algoritmo implementado es un Parser LL muy sencillo y útil para gramáticas simples como esta.

4. Método

Los árboles se almacenan en nodos, cada uno de los cuales maneja ciertas propiedades como color de borde, color de relleno, etiqueta, identificador y algunas propiedades especiales como un valor lógico de marca y un valor lógico de invisibilidad los cuales serán explicados posteriormente. Cada nodo tiene además un apuntador a su padre y a todos sus hijos. Se proveen métodos para desplegar el árbol en un archivo con formato SVG (Scalable Vector Graphics).

```
struct Nodo{
 vector<Nodo*> hijos;
 Nodo * padre;
 bool invisible;
 bool marca;
 int x,y;
 // Otros atributos de despliegue como color
 // forma, etc
};
```

Listado 1: Estructura de datos para nodos

El nodo raíz del árbol está contenido en un objeto de clase `Tree` el cual tiene un apuntador al nodo raíz, además de tener información específica para el despliegue, como la caja delimitadora del árbol, el número de niveles que contiene y otros. Esta clase contiene métodos para buscar y resaltar caminos en el árbol.

4.1. Despliegue de árboles

Para desplegar árboles se supone que ya se conoce la posición de cada nodo del árbol. De esta manera, el árbol simplemente despliega recursivamente cada uno de los hijos del nodo actual y finalmente despliega el nodo. En el Listado 2 se aprecia el algoritmo que realiza esta operación. Se recorre cada hijo del nodo actual, y aquellos que no son invisibles serán desplegados recursivamente. Para esto, primero se dibuja la línea que conecta al nodo actual con el hijo, y luego se dibuja recursivamente al hijo. Una vez que todos los hijos han sido desplegados se dibuja un círculo en que representa al nodo actual.

```
Nodo.desplegar()
{
 para cada h en hijos hacer
 {
  si(no h.invisible)
  {
   SVG.dibujar_linea(x+radio, centroy,
            hijo.x+radio, hijo.y+radio);
   h.desplegar();
  }
 }
 SVG.dibujar_circulo(x+radio, y+radio, radio,
                     propiedades_del_nodo);
}
```

Listado 2: Despligue recursivo del árbol

Es importante destacar en este punto que existen dos opciones para dibujar las líneas que conectan a un nodo con sus hijos. La Figura 2 muestra estas opciones. En la primera las líneas salen del centro del nodo, mientras que en la segunda las líneas salen del punto mas bajo del nodo. La diferencia en implementación es simplemente la forma cómo se calcula la variable `centroy`. Para hacer que las líneas salgan del centro del nodo, se hace `centroy = y + radio`, mientras que para que las líneas vayan al punto más bajo del nodo se hace `centroy = y + 2*radio`.

4.2. Cálculo de las posiciones de los nodos

Como vimos en la sección anterior, el despliegue del árbol es sumamente sencillo una vez que las posiciones de los nodos han sido calculadas. Para el cálculo de estas posiciones

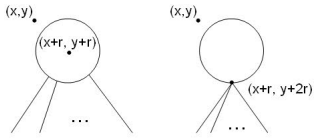

Figura 2: *Formas de conectar un nodo con sus hijos. En la izquierda las líneas salen del centro del nodo, en la derecha las líneas salen del punto mas bajo de éste*

se debe tomar en cuenta parámetros cómo la distancia entre cada nivel del árbol, distancia entre nodos hermanos, etc.

Una primera aproximación a este problema puede hacerse de manera recursiva, viendo que para calcular la posición de un nodo es necesario ver la posición de cada uno de sus hijos. Cuando sus hijos están ubicados podemos colocar al nodo centrado respecto a ellos. Ahora bien, para colocar a cada hijo es importante considerar que estos no pueden solaparse. Inicialmente puede tomarse la caja delimitadora de cada árbol para estar seguros de que no hay solapamiento, aunque esto puede traer muchos espacios vacíos como se observa en la parte izquierda de la Figura 3. La alternativa que se implementó toma en cuenta la caja delimitadora de cada nivel del árbol. De esta forma es posible conseguir un mejor ajuste de los subárboles, con menos espacios vacíos.

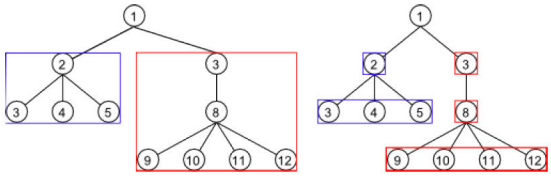

Figura 3: *En la izquierda se toman en cuenta las cajas delimitadoras para que los hijos no se solapan, originando muchos espacios vacíos. En la derecha cada nivel del árbol tiene su propia caja delimitadora, por lo que se consigue un mejor ajuste entre los dos subárboles, logrando menos espacios vacíos*

Ahora el problema de calcular la posición de un nodo se reduce a saber cómo unir los nodos hijos, teniendo en cuenta los límites en X de cada nivel. Esto es, para cada nivel del árbol, cuánto espacio ocupa en las coordenadas X (las coordenadas Y no interesan ya que son constantes para cada nivel). Para la unión de dos árboles se debe tener una estructura de datos auxiliar que indica los límites de cada nivel del árbol. Un límite es simplente un par de enteros x_0, x_1 que indican las coordenadas X mínima y máxima de ese nivel. Entonces, para hacer la unión entre dos árboles consideramos primero el caso cuando alguno de los dos árboles es vacío (en cuyo caso el resultado es el otro árbol). Si ningún ábol es vacío hay que ver cuál tiene altura mínima ya que esa es la cantidad de niveles que deben considerarse en la unión. El algoritmo pseudo formal se describe en el listado 3:

```
union(vector<Limite> r, vector<Limite> n,
      xt, maxl)
{
 s1 = r.size();
 s2 = n.size();
 maxl = 0;
 si(s1==0)
```

```
{
 r = n;
 xt = 0;
 retornar;
}
xt = -999999;
mins = min(s1, s2);
para i=0 hasta mins hacer
{
 xtl = r[i].12 - n[i].11 + minDist;
 si(xtl > xt)
  xt = xtl;
}

para i=0 hasta mins hacer
 r[i].12 = n[i].12 + xt;

si (s2>i)
 mientras i<s2 hacer
 {
  r.insertar(Limite(n[i].11+xt,
                    n[i].12+xt));
  i = i + 1;
 }

minLim1 = 999999;

para j=0 hasta i hacer
 si (r[j].11 < 0 y r[j].11 < minlim1)
  minlim1 = r[j].11;

si(minlim1 < 0)
 para j=0 hasta i hacer
  r[j].trasladar(-minlim1);
}
```

Listado 3: Unión entre dos árboles

Inicialmente se verifica si uno de los árboles es vacío, ya que la unión de ambos árboles generará al otro árbol. En caso de que ninguno esté vacío se procede a determinar la altura mínima entre ellos, la cuál será usada como base para la generación del nuevo árbol. Los niveles que tengan en común ambos árboles (altura mínima) son combinados tomando en cuenta siempre que el árbol de la derecha debe trasladarse tomando como posición base $X = 0$. En cada iteración se comparan los dos niveles de los árboles para determinar cuánto es lo mínimo que puede trasladarse el árbol de la derecha (Δx) para que los niveles no se solapen. Finalmente, los niveles restantes (niveles que no tienen en común los dos árboles) son trasladados según el Δx calculado anteriormente.

Como se explicó anteriormente la salida del programa es un árbol en formato SVG. Un archivo SVG es un XML en donde se especifican figuras geométricas (líneas, elipses, rectángulos, etc), con sus propiedades de dibujo incluyendo color, grosor, relleno. Se creó una sencilla clase para soportar la escritura de este formato de archivo, que aunque únicamente soporta unos pocos tipos de figuras y atributos, son suficientes para la clase de árboles que estamos dibujando. En caso de ser necesario, este archivo puede ser retocado posteriormente con diversos programas como Inkscape.

Existen muchas estructuras de datos cuya implementación no es únicamente un árbol sino un bosque constituído por varios árboles. Ejemplos de esto son los Heaps de Fibonacci [CLRS09], Heaps Binomiales [CLRS09], y conjuntos disjuntos [AUH83]. Este tipo de estructuras son soportadas a través de la definición de los árboles que componen al bosque en archivos separados, los cuáles son posteriormente unidos generando un único archivo SVG resultante.

Como se explicó anteriormente es posible tener nodos invisibles. Esto se hizo para dar soporte a árboles binarios, en donde un nodo que tenga subhijo izquierdo pero no tenga subhijo derecho es distinto a un nodo con subhijo derecho pero sin subhijo derecho. Esta distinción no existe en árboles que no sean binarios. La Figura 4 muestra esta situación. El árbol de la izquierda no tiene hijo derecho por lo que éste se representa con un nodo invisible que ocupa cierto espacio y obliga al nodo izquierdo a moverse a la izquierda. El árbol del centro tiene un hijo izquierdo invisible para hacer que el nodo derecho se mueva hacia la derecha. Finalmente, el árbol de la derecha no es binario por lo que aparece con un sólo hijo.

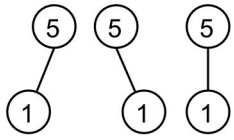

Figura 4: *El árbol de la izquierda tiene un hijo izquierdo pero no hijo derecho (invisible). El árbol del centro tiene hijo derecho pero su hijo izquierdo es invisible. El árbol de la derecha no necesita nodos invisibles ya que no es binario*

5. Interfaz Web

Con el fin de facilitar el uso de TreeView se creó una pequeña interfaz web que permite crear interactivamente un árbol. A través de esta interfaz el usuario escribe en un control de texto la descripción del árbol, con el formato que se explicó en la sección 3. Esta interfaz web toma periódicamente la descripción del árbol y la envía al servidor web vía AJAX. El servidor web procesa la descripción, e intenta generar una imagen asociada a la descripción del árbol. Muchas veces esta imagen no podrá ser completada debido a errores de sintaxis. En caso de que la imagen pueda generarse correctamente, esta es enviada de vuelta al cliente y es mostrada. De esta forma, el usuario puede hacer cambios en tiempo real sobre el árbol y ver los resultados de inmediato. La Figura 5 muestra la interfaz web. Puede observarse en la parte superior el control de texto para introducir la descripción del árbol, y en la parte inferior el árbol obtenido. Esta imagen del árbol se modifica cada vez que se cambia la descripción del árbol. El botón "Download" permite descargar el archivo de la imagen en formato SVG.

6. Pruebas y resultados

En esta sección se presentan algunas pruebas hechas mostrando la potencialidad de la herramienta, y se discuten los resultados.

Se crearon diversos árboles que surgen naturalmente al aplicar diversos algoritmos sobre distintas estructuras de datos. En la primera prueba que se realizó se hicieron varias inserciones en un árbol binario de búsqueda con el fin de resaltar la forma en que se van insertando los nodos. La Figura 7 muestra el resultado correspondiente. Se destacan nodos

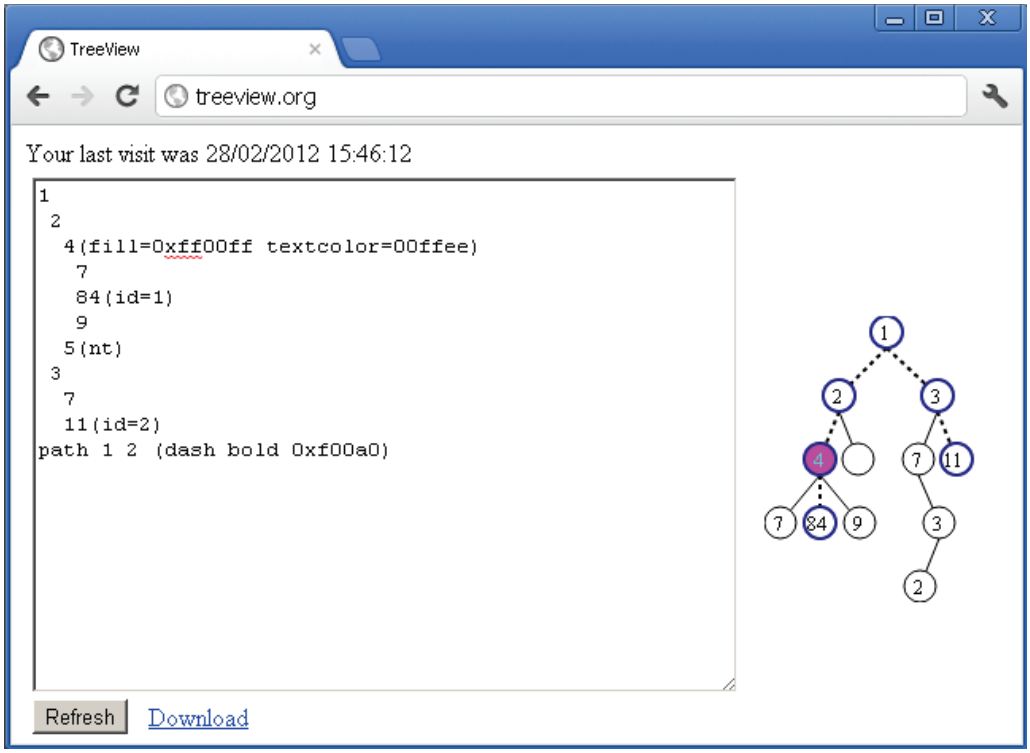

Figura 5: *Interfaz web para TreeView*

del árbol pintados con una brocha gruesa para resaltarlos. En este caso los nodos resaltados corresponden a el nodo insertado en cada paso.

La Figura 6 muestra un árbol dibujado con distintas configuraciones de separación vertical y horizontal. En (a) la separación horizontal es 5 y la vertical 30 unidades, dando lugar a un árbol con un radio aspecto vertical. En (b) la separación vertical se reduce a 20, obteniendo un árbol con radio aspecto más cercano a 1 (medidas similares en X y Y.). En (c) la medida vertical se reduce aun más obteniendo un árbol más achatado, con algunas líneas que se ocultan detrás de los nodos (por ejemplo la línea que une al nodo 8 con el nodo 12). En (d) la separación vertical se fija en 20 y la separación horizontal se aumenta a 10. En (e) la separación horizontal se aumenta más aun hasta 20, obteniendo un árbol con nodos muy separados.

La Figura 8 muestra dos inserciones sobre un árbol AVL en donde es necesario realizar rotaciones. Gracias a la habilidad de resaltar nodos con brochas gruesas es posible destacar los puntos en donde hay nodos que participan en el desbalanceo del árbol, originando rotaciones.

La Figura 9 muestra varias etapas de la inserción de nodos en un Heap. Se destacan nodos en distintos colores para resaltar los nodos que se están comparando en cada iteración.

La Figura 10 muestra un Heap de Fibonacci [CLRS09] el cuál no es un árbol sino un bosque. TreeView permite la creaci ’on de bosques aunque no conecta los distintos árboles del bosque. La imagen de la derecha muestra el resultado final retocado con Inkscape.

La Figura 11 muestra un ejemplo de un conjunto disjunto

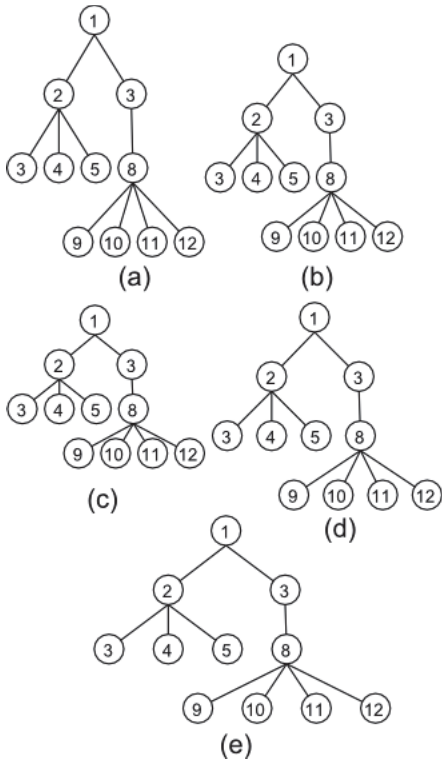

Figura 6: *Un mismo árbol dibujado con distintas separaciones horizontales y verticales. En (a) las separación horizontal es de* 5*, y vertical* 30*. En (b) las separaciones son* 5 *y* 20*. En (c)* 5 *y* 10*. En (d)* 10 *y* 20 *y en (e)* 20 *y* 20*.*

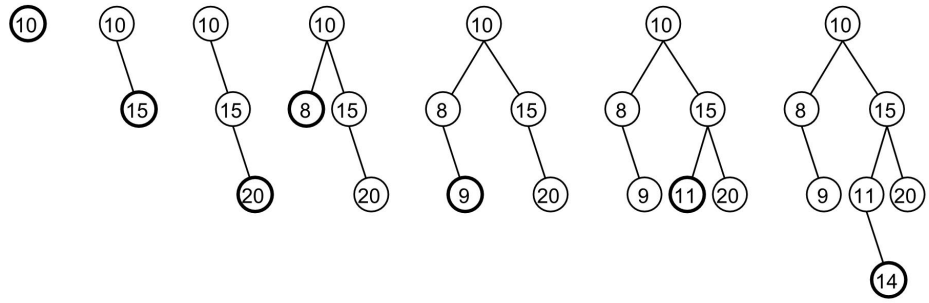

Figura 7: *Resultado de insertar varios nodos en un árbol binario de búsqueda*

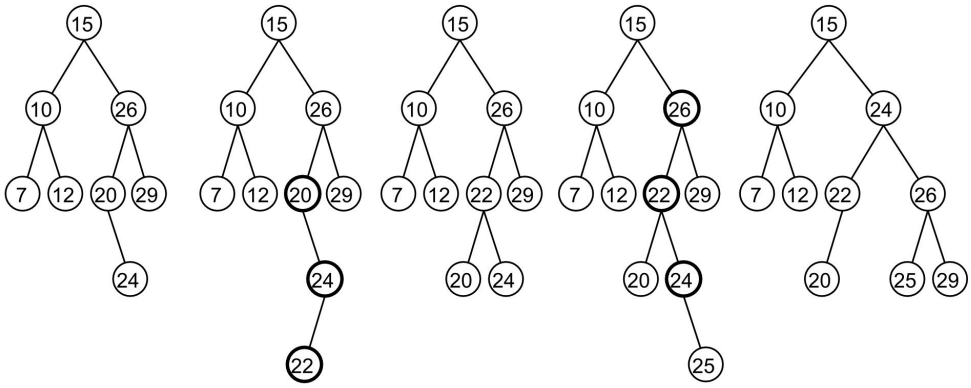

Figura 8: *Resultado de insertar nodos en un árbol AVL incluyendo rotaciones*

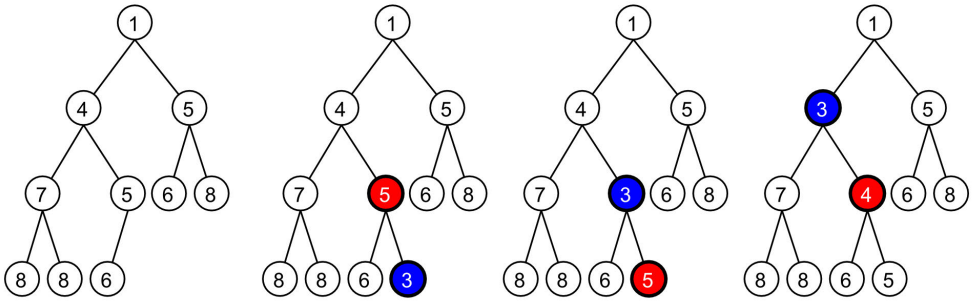

Figura 9: *Varias etapas en la inserción de un nodo en un Heap. El nodo azul es el que se insertó y se compara contra su padre (rojo) hasta que ya no pueda subir más*

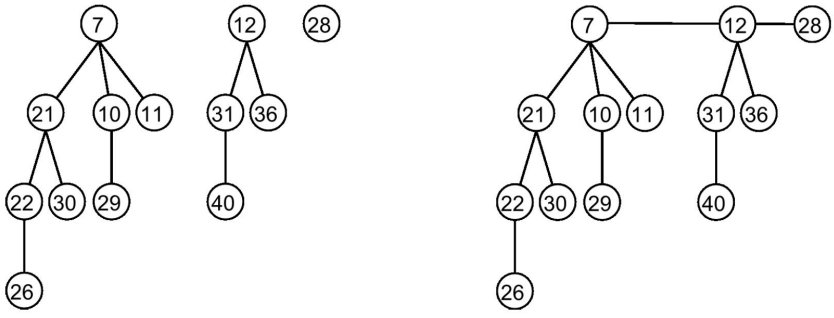

Figura 10: *Un Heap de Fibonacci es en realidad un bosque compuesto por múltiples árboles. La Figura de la izquierda muestra el bosque generado por TreeView, y la figura de la derecha es el resultado final retocado con Inkscape. Se agregaron enlaces entre los bosques que no son actualmente soportados por TreeView*

el cual se representa como un bosque. Los árboles de éste bosque pueden tener cualquier cantidad de hijos.

La Figura 12 muestra un árbol binario de búsqueda y se destaca un camino en el árbol correspondiente a los nodos que serían visitados si se desea encontrar el elemento 66.

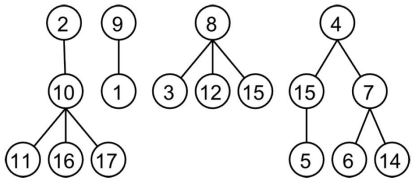

Figura 11: *Ejemplo de conjunto disjunto, representado por un bosque*

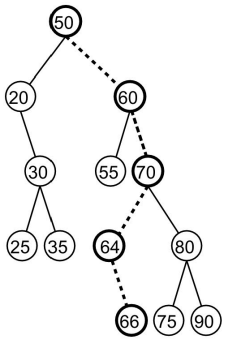

Figura 12: *En esta figura se destaca un camino dentro del árbol. En este caso el árbol corresponde a los nodos que serían visitados si se busca el elemento* 66 *en el árbol.*

Otra estructura de datos muy usada en el área de Computación Gráfica son los QuadTrees que permiten descomponer jerárquicamente una imágen [FB74]. La Figura 13 muestra un QuadTree de 3 niveles. Las etiquetas de los nodos no son de interés de éste árbol, por lo que fueron removidas.

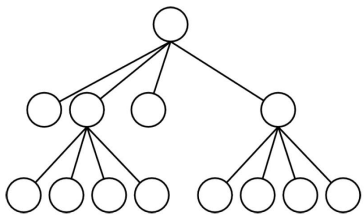

Figura 13: *QuadTree con* 3 *niveles.*

7. Conclusiones y trabajos futuros

El algoritmo desarrollado permite la creación de árboles genéricos y árboles binarios de una forma sencilla, que pueden ser retocados posteriormente con aplicaciones simples o usados directamente. Los diversos atributos soportados proveen una gran flexibilidad lo que hace sencilla la construcción de árboles complejos con unas pocas líneas. El formato de archivo provisto es fácil de generar y entender, pudiéndose crear árboles sencillos a mano, o pudiendo desde cualquier herramienta ya existente generar fácilmente estos archivos con el fin de visualizar árboles.

La herramienta ha probado ser útil para el despliegue de árboles. Además de las diversas pruebas mostradas aquí, también fue utilizada para la creación de todas las figuras de árboles de las notas de docencia de la materia Técnicas Avanzadas de Programación dictada en la Universidad Central de Venezuela.

La posibilidad de mostrar diversos caminos entre un par de nodos es sumamente útil para explicar diversos tipos de algoritmos asociados a árboles.

Es posible estimar automáticamente la separación vertical y horizontal entre nodos para quitarle esta carga al usuario. El usuario podría tener siempre la potestad de modificar los valores sugeridos por el sistema.

En un futuro se planea agregar soporte para colocar etiquetas en sitios específicos, por ejemplo asociadas a arcos. También es posible tener arcos de retorno que apunten de un nodo hacia alguno de sus antecesores con el fin de soportar árboles de ejecución.

El despliegue de árboles con muchos hijos (quadtrees y octrees) puede mejorarse en un futuro para que los árboles sean visualmente más agradables.

El análisis sintáctico del archivo de entrada puede mejorarse para hacerlo más robusto, implementando un parser recursivo descendente que considere manejo de errores. El formato de archivos debe actualizarse para soportar las nuevas características propuestas como manejo de etiquetas.

La interfaz web puede ser mejorada para que sea más atractiva, incluyendo ejemplos, y la posibilidad de almacenar y reutilizar árboles creados previamente. También sería útil hacer resaltado de sintaxis en el área de texto para ayudar en la definición de los árboles.

Referencias

[AUH83] AHO A., ULLMAN J., HOPCROFT J.: *Data Structures and Algorithms*, 1 ed. Addison Wesley, 1983.

[BKW89] BRÜGGEMANN-KLEIN A., WOOD D.: Drawing trees nicely with tex. *T E X: Applications, Uses, Methods 2* (1989), 185–206.

[CLRS09] CORMEN T., LEISERSON C., RIVEST R., STEIN C.: *Introduction to Algorithms*, 3 ed. The MIT Press, 2009.

[FB74] FINKEL R. A., BENTLEY J. L.: Quad trees: A data structure for retrieval on composite keys. *Acta Inf. 4* (1974), 1–9.

[FK02] FLEISCHER R., KUCERA L.: Algorithm animation for teaching. In *Software Visualization*, Diehl S., (Ed.), vol. 2269 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2002, pp. 640–642.

[Luo93] LUO T.: *TreeDraw: A Tree-Drawing System*. PhD thesis, University of Waterloo, 1993.

[RT81] REINGOLD E. M., TILFORD J. S.: Tidier drawing of trees. *IEEE Trans. Software Eng* (1981).

[WS79] WETHERELL C., SHANNON A.: Tidy drawings of trees. *IEEE Trans. Software Eng. 5*, 5 (1979), 514–520.