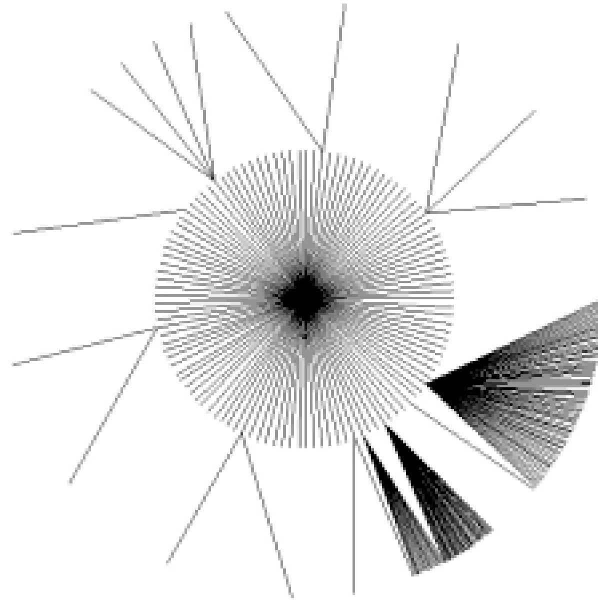
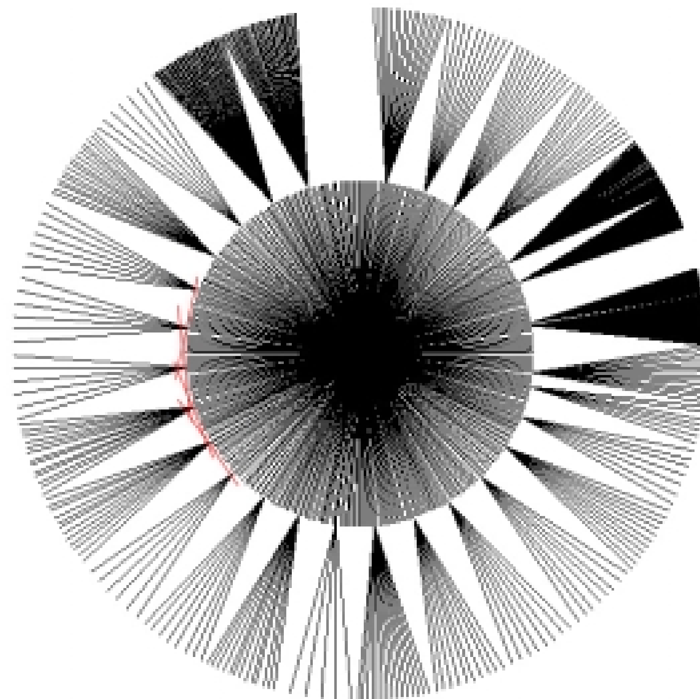
Radial Tree Graph Drawing Algorithm for Representing Large Hierarchies

Greg Book & Neeta Keshary

University of Connecticut
December 2001



Graph of career.uconn.edu on January 2, 2002 using the radial tree graph drawing method.



Graph of nytimes.com with tangent lines drawn in red

1
Description of Problem and Solution

We wanted to represent the directory structure of websites in a graph format and we chose a radial tree graph. Our graph would appear as if you were looking down onto a tree with the branches radiating from the center. An important consideration would be that the branches of the tree do not overlap. We solved this problem by creating limits for each parent node in which their children nodes could reside on the graph.

The radial tree graph also solves the problem of drawing a tree so that nodes are evenly distributed. A binary tree that is drawn linearly, so that the root is on one end, and nodes of the same level line up, will grow crowded very quickly. This is because trees tend to grow more nodes per level than they do levels. A radial tree will spread the larger number of nodes over a larger area as the levels increase. We use the terms *level* and *depth* interchangeably.

In combination with a fisheye magnification of the graph, this graph format can be a useful tool for viewing large hierarchies.

2
Graph Drawing Algorithm

The graph is drawn to fit our screen area in our web explorer example. While it is possible to draw the graph into any size space, not all of it has to be displayed on the screen at the same time. It may be useful to only display part of it, in effect magnifying a portion of the graph. Assuming that drawing space and screen space are considered the same, we describe how to draw the graph using variables called *ScreenHeight* and *ScreenWidth*.

2.1
Description of Algorithm

To implement the graph without leaves overlapping, we chose to have limits for each directory in which the children of that directory would lie. All nodes on the graph lie in concentric circles that are focused in the center of the screen. We begin by laying out the position of these circles.

First, the center of the screen must be located. The center is found by dividing the *ScreenHeight* and *ScreenWidth* in half and the root node of the graph is placed at this point. The smaller of the *ScreenHeight/2* and *ScreenWidth/2* is used to determine how large the graph will be. It looks better to keep the entire graph on the screen, so the graph size will be based on the smaller of the two *ScreenHeight* and *ScreenWidth* values calculated before.

The distance between levels in the graph is calculated by taking the smaller distance that was created before and dividing it by the number of levels in the graph. We call this distance *d*. This distance is used later on to calculate the location of nodes (figure 1).

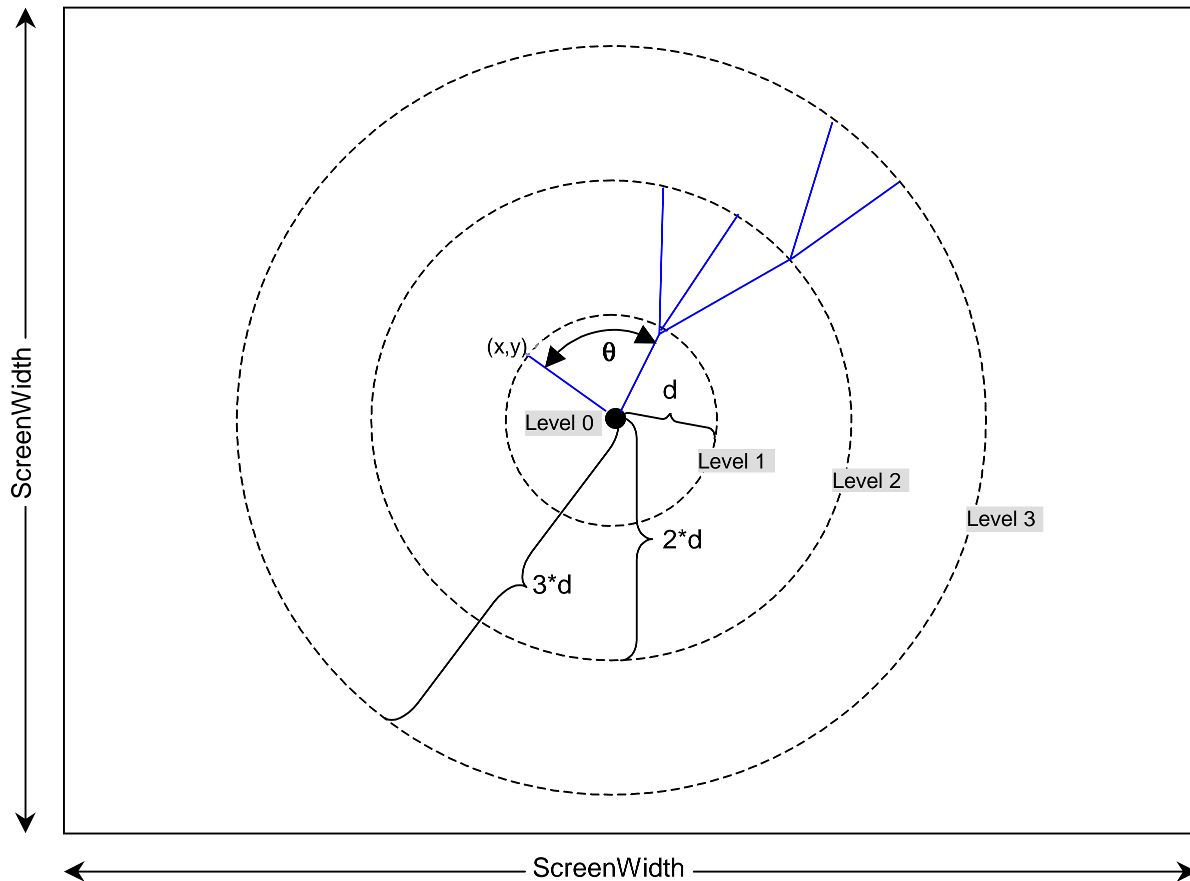


Figure 1 – General layout of radial graph

Once the location and size of the graph are determined, we want to begin placing nodes on the graph. We work out from the root of the graph; out from level 0. Begin by placing the root node at the center of the graph. This node is the parent for every other node that will be drawn in the next level.

Level 1 is a special case since all the nodes have the same parent. The nodes for level 1 can be distributed over all the 360° of the circle. We used pi measure to keep things consistent with what the sin and cos functions return in C++. We divide 2pi by the number of nodes at level 1. This creates an AngleSpace between the nodes on the circle. We then iterate through all the nodes at level 1 and calculate the position of the node using the equations:

$$node_x = (d*LevelNum) * \sin(NodeIndexWithinLevel*AngleSpace))$$
$$node_y = (d*LevelNum) * \cos(NodeIndexWithinLevel*AngleSpace))$$

If a node happens to be a directory, then the bisector limits and tangent limits must be calculated (figure 2). All the children of that directory must fall within certain limits on the circle on which they will be drawn to prevent overlapping. The tangent limit is drawn perpendicular to the directory. The points where it intersects the circle immediately outside the directory's level are taken as tangent limits for the directory. This is done so that the children of a directory do no overlap other nodes by being on an opposite side of the circle.

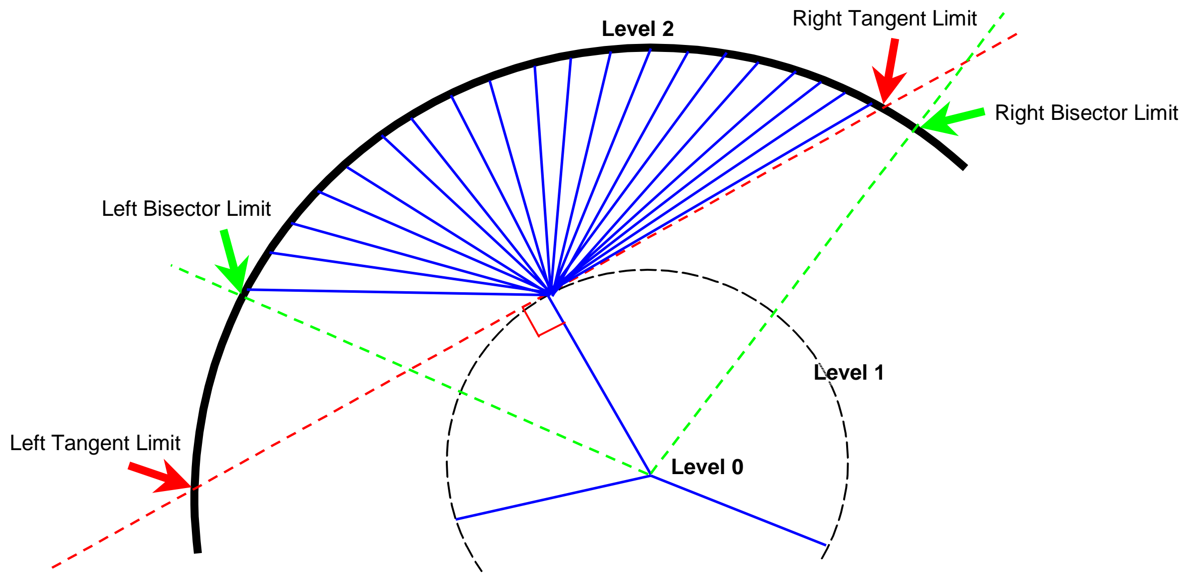


Figure 2 – Tangent and bisector limits for directories.

The bisector limits exist between directories. Between any two directories, a bisector limit will be calculated for both directories on either side of the limit. The bisector limit is needed so children do not overlap the children of an adjacent directory.

We use the following equation to calculate the tangent limits:

$$ArcAngle = 4 * \arcsin(R_{in}/R_{out})$$

This will give us the angle on the outer circle in which the tangent line will cross. Adding half of the ArcAngle to each side of the directory will give the tangent limits.

In Levels 2 and greater, we must know the location of a parent to draw its children. Since the parents have limits where their children can go, we must keep track of the children that are being drawn and make sure they fall within their parent's space. We must get a list of the parents that have children in the level we are at. With that list, we get a list of the children for each parent. Knowing the limits of each parent, we loop through the list of parents and then loop through all the children for that parent. We calculate the child's location relative to the parent's, adding in the offset of the limit angle.

After calculating the location, if there are any directories at the level, we must calculate the bisector and tangent limits for those directories. The method is the same as level 1 for levels 2 and higher.

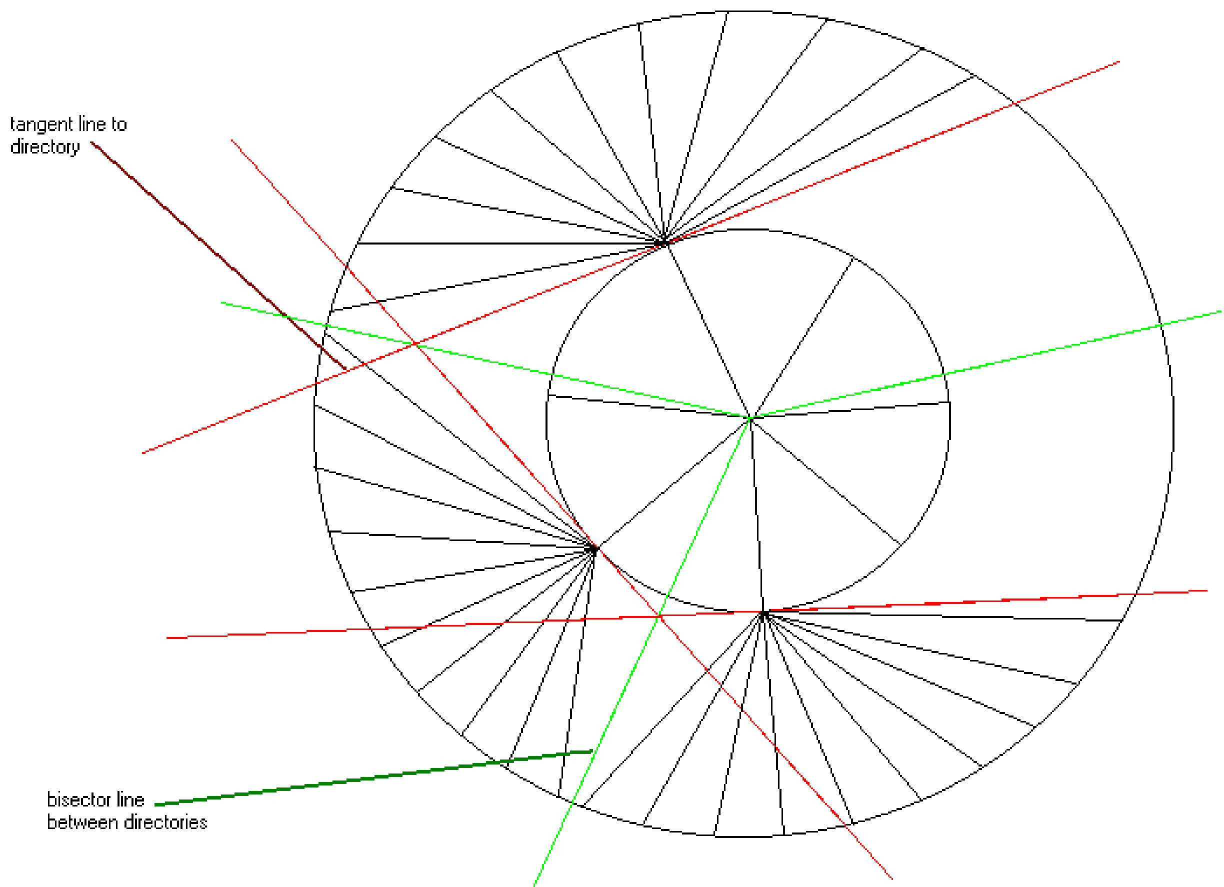


Figure 3 – Detail for bisector limits

2.2
Psuedocode for the graph drawing algorithm

Loop through each level in the data structure

Switch level

```
case 0:
    Find the center of the drawing area, to center the graph
    Set RootNode = CenterX, CenterY

case 1:
    AngleSpace = (2Π radians/NumNodesAtThisLevel)
    Loop through all nodes at this level
        Calculate x,y positions:
        If (Node.type == Parent)
            Calculate bisector limits and tangent limits for the node
    End loop
```

case 2:

Nodes in levels two and higher must be grouped according to their parent.
Loop through all nodes at this level and get a list of the parent nodes
And get the number of children for each parent
Calculate the AngleSpace for each parent:

AngleSpace = (leftLimit – rightLimit)/NumNodesForThisParent

Foreach parent

Loop through all nodes for that parent

Calculate x,y position for the child node
If (childnode.type == Directory)

Calculate bisector and tangent limits

End loop

End foreach

End switch

2.3 Example Data Structure for File Tree

This data structure was created with the help of Matt Costa and Freddie Pitts, also from University of Connecticut.

Listing 1 – data structure definition

```
typedef unsigned int uint;
typedef unsigned char uchar;

// node types.
#define INT_FILE  0     // file.
#define DIRECTORY 2     // directory.

struct node
{
      uchar type;              // one of defined types (above).
      char* name;              // self explanatory (s.e.)
      uint depth;              // depth from root ( depth at root = 0 ).
      node** children;         // pointer to array of pointers to children
      node** links;            // pointer to array of pointers to links
      node* parent;            // s.e.
      node* next;              // next node at depth
      node* prev;              // prev node at depth

      float x;                 // x coordinate.
      float y;                 // y coordinate.
      float degsToPrevDir;     // degrees to previous directory.
      float degsToNextDir;     // degrees to next directory.
      float angle;
      float rightBisectorLimit;
      float leftBisectorLimit;
      float rightTangentLimit;
```

```
        float leftTangentLimit;
        float rightLimit;
        float leftLimit;

};

struct nodeTable
{
        uint level;                 // level of this table.
        uint numNodes;              // number of nodes at this level.
        node* firstNode;            // pointer to first node at this level.
        node* lastNode;             // pointer to the last node at this level.
        nodeTable* next;            // pointer to table for level+1.
};
```

2.3
Traversal of the Data Structure

We chose to create a data structure that represented the tree as a combination of tables and trees. The tree doesn’t exist as a series of pointers in the normal sense of a tree, it exists as a linked list, which makes traversal of the tree easier when drawing graphs. Each node has its own structure, containing the information about the location of the node. We also created a linked list of node tables. Each table contains a pointer to the beginning of a linked list containing the nodes for that level.

We traverse the graph by starting at the first node table, and going to the first node. The first level should be level zero, and should contain only one node. We follow the pointer to the next node table, and iterate through the nodes at that level. This is repeated for all levels until the end of the last node table is reached.

3
Code Listing for Graph Drawing Algorithm

```
int CalculateNormalXY() {

  RedisplayCounter++;

  nodeTable* NodeTable;

  node* PreviousDirectory;
  node* CurrentNode;
  node* TempCurrentNode;
  node* LastNode;
  node* FirstNode;
  node* FirstDirectory;
  node* TempParent;
  node* RootNode;

  float RootX, RootY;
  int level;
  int NumLevels;
  int d;
  int NodeIndex;
```

```
int ParentsSoFar;
int FoundParent;
int FirstDirectoryFound;
int HadPreviousDirectory;
double AngleSpace;
double NodeSpace;
double LastDirectoryAngle;
double DegsToPrevDirectory, DegsToNextDirectory;
double Arc;
double ArcAngle;
double RemainingDegrees;
double DegsToFirstDirectory;
double leftlimit, rightlimit;
double LeftTanLimit, RightTanLimit;
double RightBisLimit, LeftBisLimit;
double LeftLimit;

struct ParentStructure
{
  node* Children[MAXCHILDREN];
  int NumChildren;
  node* ParentPtr;
} Parents[MAXPARENTS];

int ParentIndex, ChildIndex;
int ChildrenSoFar;
int NumNodes;
int NumChildren;
int NumParents;

// end declaring variables //

RootX = FocusX = ((X2 - X1)/2) + X1;
RootY = FocusY = ((Y2 - Y1)/2) + Y1;
ScreenHeight = (Y2 - Y1);
ScreenWidth = (X2 - X1);

NumLevels = data1->GetDepth();

if (NumLevels != 0) {
  d = GetLevelHeight(ScreenWidth, ScreenHeight, NumLevels);
}

for (level=0; level <= NumLevels; level++)
{
  if (level == 0) {
    NodeTable = data1->GetNodeTable(level);
    RootNode = NodeTable->firstNode;
    RootNode->x = RootX;
    RootNode->y = RootY;
  }
  if (level == 1)
  {
    NodeTable = data1->GetNodeTable(level);
    CurrentNode = NodeTable->firstNode;
    FirstNode = NodeTable->firstNode;
    LastNode = NodeTable->lastNode;
    NumNodes = GetNumNodes(1,level);

    AngleSpace = (double)(6.28318530718)/(double)NumNodes;

    FirstDirectoryFound = 0;  // if a directory has been found yet
    LastDirectoryAngle = 0.0;    // angle of the last directory
    HadPreviousDirectory = 0;    // if have had a previous directory
    DegsToPrevDirectory = 0.0; // initialize degrees to previous for the first directory
    DegsToNextDirectory = 0.0; // initialize degrees to next

    NodeIndex = 0;
    // loop through all nodes at this level
    while (CurrentNode)
    {
```

```
        if ((CurrentNode->type == 0) || (CurrentNode->type == 2)) {
          // compute x,y positions
          CurrentNode->x = RootX + ((d*level) * cos(AngleSpace*NodeIndex));
          CurrentNode->y = RootY + ((d*level) * sin(AngleSpace*NodeIndex));
          CurrentNode->angle = AngleSpace * NodeIndex;


          if (CurrentNode->type == 2)
          {
            // if no first directory has been found yet, then one has been found, so set it to
the first directory
            if (!FirstDirectoryFound)
            {
              FirstDirectory = CurrentNode;
              FirstDirectoryFound = 1;
            }

            // get degrees to previous directory
            DegsToPrevDirectory = CurrentNode->angle - LastDirectoryAngle;

            // calculate right bisector limit
            RightBisLimit = CurrentNode->angle - DegsToPrevDirectory/2;
            CurrentNode->rightBisectorLimit = RightBisLimit;

            // if its not the first directory, set the NextDegrees for the previous directory to
the previous degrees
            // of the current directory
            if (HadPreviousDirectory) {
              // set degrees to next directory for the previous directory
              PreviousDirectory->degsToNextDir = DegsToPrevDirectory;

              // set left bisector limit
              PreviousDirectory->leftBisectorLimit = CurrentNode->rightBisectorLimit;

            }

            // calculate ARC
            ArcAngle = ( (double)d * (double)level ) / ( (double)d * ( (double)level + 1.0 ) );
            Arc = 4*(asin( ArcAngle ));

            // calculate left tangent limit
            LeftTanLimit = CurrentNode->angle + (Arc/2);
            CurrentNode->leftTangentLimit = LeftTanLimit;

            // calculate right tangent limit
            RightTanLimit = CurrentNode->angle - (Arc/2);
            CurrentNode->rightTangentLimit = RightTanLimit;

            // get right and left limits
            CurrentNode->leftLimit = GetLeftLimit(CurrentNode->leftBisectorLimit, CurrentNode-
>leftTangentLimit);
            CurrentNode->rightLimit = GetRightLimit(CurrentNode->rightBisectorLimit, CurrentNode-
>rightTangentLimit);

            LastDirectoryAngle = CurrentNode->angle;
            PreviousDirectory = CurrentNode;

            // set that we've had a previous directory
            HadPreviousDirectory = 1;
          }
          NodeIndex++;
        }
        TempCurrentNode = CurrentNode;
        CurrentNode = CurrentNode->next;
      }
      if (FirstDirectoryFound) {
        // add remaining degrees to first directory
        RemainingDegrees = (6.28318530718) - PreviousDirectory->angle;

        DegsToFirstDirectory = RemainingDegrees + FirstDirectory->angle;
```

```
        RightBisLimit = FirstDirectory->angle - DegsToFirstDirectory/2;
        FirstDirectory->rightBisectorLimit = RightBisLimit;

        if (FirstDirectory->rightBisectorLimit < 0) {
          LeftLimit = FirstDirectory->rightBisectorLimit + (6.28318530718);
        }
        else {
          LeftLimit = FirstDirectory->rightBisectorLimit;
        }
        PreviousDirectory->leftBisectorLimit = LeftLimit + (6.28318530718);

        FirstDirectory->leftLimit = GetLeftLimit(FirstDirectory->leftBisectorLimit,
FirstDirectory->leftTangentLimit);
        FirstDirectory->rightLimit = GetRightLimit(FirstDirectory->rightBisectorLimit,
FirstDirectory->rightTangentLimit);

        PreviousDirectory->leftLimit = GetLeftLimit(PreviousDirectory->leftBisectorLimit,
PreviousDirectory->leftTangentLimit);
        PreviousDirectory->rightLimit = GetRightLimit(PreviousDirectory->rightBisectorLimit,
PreviousDirectory->rightTangentLimit);
      }

    }
    if (level >= 2) {
      // level is not equal to one... so do all the other levels

      // loop through each node in this level, put the nodes into groups by its parent
      // get list of node's parents for this level
      // loop through all nodes all this level
      FirstDirectoryFound = 0;  // if a directory has been found yet
      LastDirectoryAngle = 0.0;    // angle of the last directory
      HadPreviousDirectory = 0;    // if have had a previous directory

      ParentsSoFar = 0;
      NodeTable = data1->GetNodeTable(level);
      CurrentNode = NodeTable->firstNode;
      LastNode = NodeTable->lastNode;
      // loop through all nodes at this level to get list of parents
      while (CurrentNode)
      {
        if (((CurrentNode->type == 0) || (CurrentNode->type == 2)) && (CurrentNode->parent)) {
          TempParent = CurrentNode->parent;
          // loop through Parents array
          FoundParent = 0;
          for (ParentIndex=0; ParentIndex <= ParentsSoFar; ParentIndex++) {
            if (TempParent == Parents[ParentIndex].ParentPtr) {
              FoundParent = 1;
              break;
            }
          }
          if (!FoundParent) {
            // add the parent to the array
            Parents[ParentsSoFar].ParentPtr = TempParent;
            ParentsSoFar++;
          }
          Parents[ParentsSoFar].ParentPtr = TempParent;
        }
        CurrentNode = CurrentNode->next;
      }
      NumParents = ParentsSoFar - 1;

      for (ParentIndex=0; ParentIndex <= NumParents; ParentIndex++) {

        ChildrenSoFar = 0;

        CurrentNode = NodeTable->firstNode;
        while (CurrentNode)
        {
          if ((CurrentNode->type == 0) || (CurrentNode->type == 2)) {
            TempParent = CurrentNode->parent;
```

```
            if (Parents[ParentIndex].ParentPtr == TempParent) {
              ChildrenSoFar++;
              Parents[ParentIndex].Children[ChildrenSoFar] = CurrentNode;
              Parents[ParentIndex].NumChildren = ChildrenSoFar;
            }
          }
          CurrentNode = CurrentNode->next;
        }

      }

      for (ParentIndex=0; ParentIndex <= NumParents; ParentIndex++) {

        NumChildren = Parents[ParentIndex].NumChildren;
        leftlimit = Parents[ParentIndex].ParentPtr->leftLimit;

        rightlimit = Parents[ParentIndex].ParentPtr->rightLimit;

        AngleSpace = (leftlimit - rightlimit)/NumChildren;

        for (ChildIndex = 1; ChildIndex <= NumChildren; ChildIndex++) {

          CurrentNode = Parents[ParentIndex].Children[ChildIndex];

          if ((CurrentNode->type == 0) || (CurrentNode->type == 2)) {
            // calculate the x,y positions of the children
            CurrentNode->x = RootX + ((d*level) * (cos(AngleSpace*ChildIndex + rightlimit)));
            CurrentNode->y = RootY + ((d*level) * (sin(AngleSpace*ChildIndex + rightlimit)));
            CurrentNode->angle = (AngleSpace * ChildIndex) + rightlimit;

            // if is directory...
            if (CurrentNode->type == 2) {

              // if no first directory has been found yet, then one has been found, so set it to
the first directory
              if (!FirstDirectoryFound)
              {
                FirstDirectory = CurrentNode;
                FirstDirectoryFound = 1;
              }

              // get degrees to previous directory
              DegsToPrevDirectory = CurrentNode->angle - LastDirectoryAngle;

              // calculate right bisector limit
              RightBisLimit = CurrentNode->angle - DegsToPrevDirectory/2;
              CurrentNode->rightBisectorLimit = RightBisLimit;

              // if its not the first directory, set the NextDegrees for the previous directory
to the previous degrees
              // of the current directory
              if (HadPreviousDirectory) {
                // set degrees to next directory for the previous directory
                PreviousDirectory->degsToNextDir = DegsToPrevDirectory;

                // set left bisector limit
                PreviousDirectory->leftBisectorLimit = CurrentNode->rightBisectorLimit;

              }

              // calculate ARC
              ArcAngle = ( (double)d * (double)level ) / ( (double)d * ( (double)level + 1.0 ) );
              Arc = 4*(asin( ArcAngle ));

              // calculate left tangent limit
              LeftTanLimit = CurrentNode->angle + (Arc/2);
              CurrentNode->leftTangentLimit = LeftTanLimit;

              // calculate right tangent limit
              RightTanLimit = CurrentNode->angle - (Arc/2);
```

```
              CurrentNode->rightTangentLimit = RightTanLimit;


              // get right and left limits
              CurrentNode->leftLimit = GetLeftLimit(CurrentNode->leftBisectorLimit, CurrentNode-
>leftTangentLimit);
              CurrentNode->rightLimit = GetRightLimit(CurrentNode->rightBisectorLimit,
CurrentNode->rightTangentLimit);

              LastDirectoryAngle = CurrentNode->angle;
              PreviousDirectory = CurrentNode;

              // set that we've had a previous directory
              HadPreviousDirectory = 1;
            }
          }
          TempCurrentNode = CurrentNode;
          CurrentNode = CurrentNode->next;
          NodeIndex++;
        }
        if (FirstDirectoryFound) {
          // add remaining degrees to first directory
          RemainingDegrees = (6.28318530718) - PreviousDirectory->angle;

          DegsToFirstDirectory = RemainingDegrees + FirstDirectory->angle;

          RightBisLimit = FirstDirectory->angle - DegsToFirstDirectory/2;

          FirstDirectory->rightBisectorLimit = RightBisLimit;

          if (FirstDirectory->rightBisectorLimit < 0) {
            LeftLimit = FirstDirectory->rightBisectorLimit + (6.28318530718);
          }
          else {
            LeftLimit = FirstDirectory->rightBisectorLimit;
          }
          PreviousDirectory->leftBisectorLimit = LeftLimit + (6.28318530718);

          FirstDirectory->leftLimit = GetLeftLimit(FirstDirectory->leftBisectorLimit,
FirstDirectory->leftTangentLimit);
          FirstDirectory->rightLimit = GetRightLimit(FirstDirectory->rightBisectorLimit,
FirstDirectory->rightTangentLimit);

          PreviousDirectory->leftLimit = GetLeftLimit(PreviousDirectory->leftBisectorLimit,
PreviousDirectory->leftTangentLimit);
          PreviousDirectory->rightLimit = GetRightLimit(PreviousDirectory->rightBisectorLimit,
PreviousDirectory->rightTangentLimit);
        }
      }
    }
  }

}
```

4
Example Graphs

Figure 3

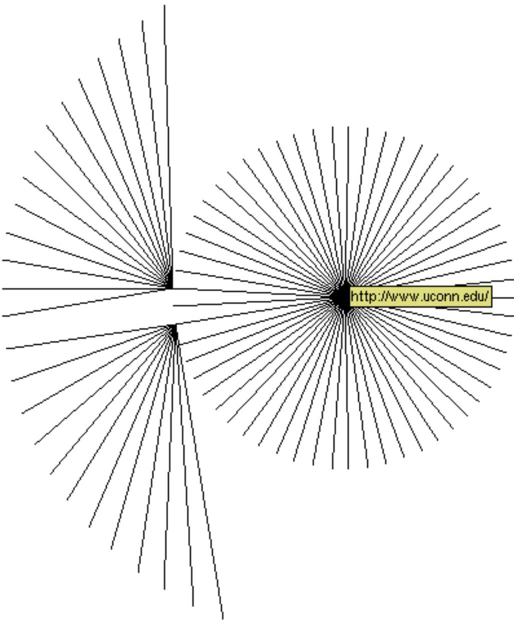


Figure 4

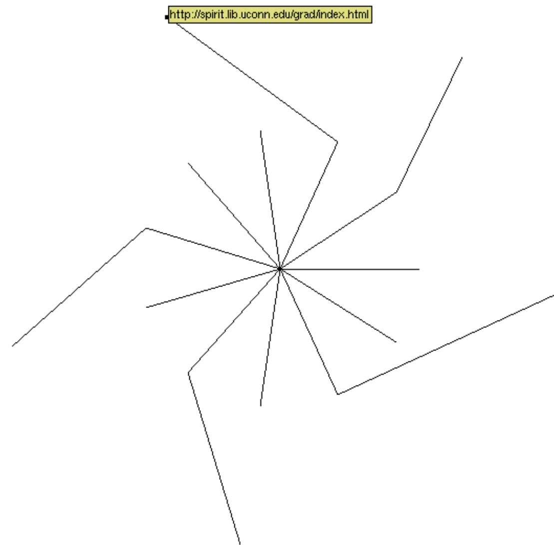


Figure 5

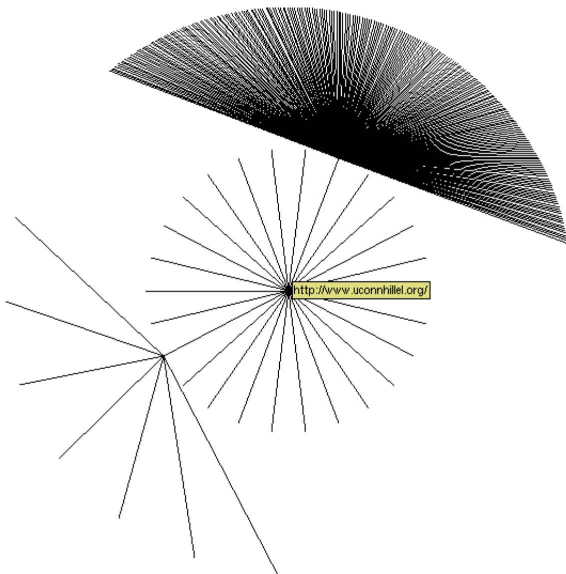


Figure 6

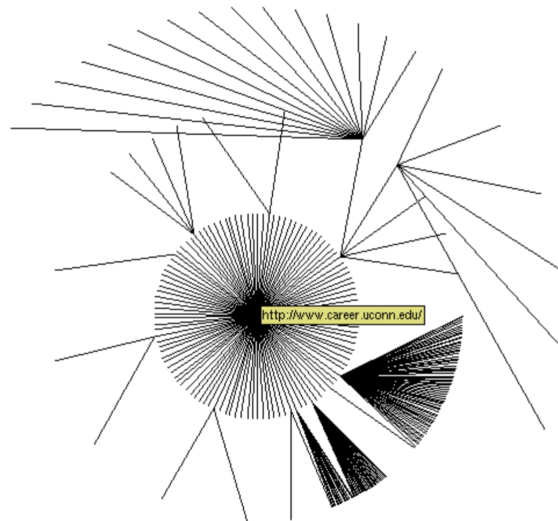


Figure 3 – http://www.uconn.edu drawn to 2 levels. Notice both the tangent limits and bisector limits effecting the placement of the children of the two directories. It's helpful to distribute the directories evenly in a graph, or they may bunch up in a particular area of the graph, this is shown in Figure 6.
Figure 4 – http://spirit.lib.uconn.edu. Notice the pinwheel shape because each directory on the graph contains only one file.
Figure 5 – http://www.uconnhillel.org. Tangent limits are very well defined.
Figure 6 – http://www.career.uconn.edu. Well defined bisector limits in the lower left of the graph. Some overlap occurred in this graph. We could attribute it to rounding errors, but it is most likely a calculation error in our code.

Works Cited

- Costa, Matt & Pitts, Freddie; CSE269 Project Presentation; October 2001
- Perkins, Chris; Mathematical Knowledge; May 2001