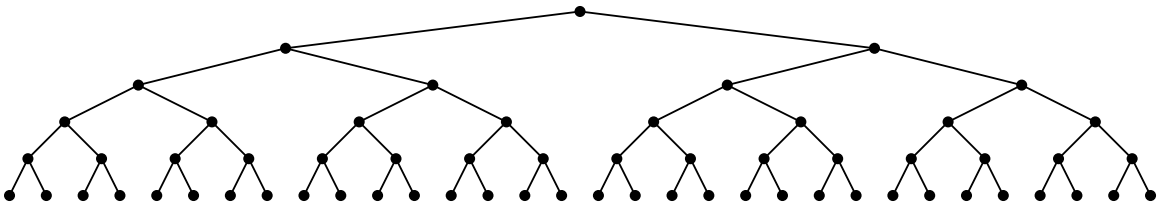
Tree Drawing

Vincent La

March 7, 2018

Binary Tree Drawing

Trees with at most two child nodes are used widely in computer science. The simplicity of their structure easily lends to mathematical analysis about algorithms operating on binary trees. Given the vast utility of this tree, many have tried to define algorithms which draw binary trees.



N-Ary Tree Drawing

Code

```
// This file contains only the tree drawing algorithm itself

#include "tree.h"

void TreeNode::calculate_xy(const unsigned int depth, const float offset, const
    /** Calculate the x, y coordinates of each node via a preorder traversal
     */

    if (depth == 0) this->calculate_displacement();

    this->x = offset + (displacement * options.x_sep);
    this->y = depth * options.y_sep;

    if (left()) left()->calculate_xy(depth + 1, this->x, options);
    if (right()) right()->calculate_xy(depth + 1, this->x, options);
}
```

```
void TreeBase::calculate_displacement() {
    /** Calculate the displacement of each node via a postorder traversal */
    this->merge_subtrees(0);
}

float TreeBase::distance_between(TreeBase* left, TreeBase* right) {
    /* Return the minimum horizontal distance needed between two subtrees
     * such that they can be placed 2 units apart horizontally
     *
     * Precondition: All nodes except the top nodes have correct displacements s
     */
    auto left_side = left->right_contour(),
            right_side = right->left_contour();
    std::map<int, float> left_cd = TreeNode::cumulative_displacement(left_side),
            right_cd = TreeNode::cumulative_displacement(right_side);

    float left_dist, right_dist, current_dist = 0;

    for (int i = 0; i < (int)std::max(left_cd.size(), right_cd.size()); i++) {
        left_dist = 0;
        right_dist = 0;
        if (left_cd.find(i) != left_cd.end()) left_dist = left_cd[i];
        if (right_cd.find(i) != right_cd.end()) right_dist = right_cd[i];

        if (abs(right_dist - left_dist) > current_dist) current_dist = abs(right
    }

    if (( (left != right) && (left->displacement == 0) ) || (right->displacement
    return current_dist;
}

std::map<int, float> TreeBase::cumulative_displacement(
        const std::vector<TreeBase*>& contour) {
    // Return the cumulative displacement taken up by a contour at each depth

    std::map<int, float> cd;
    if (!contour.empty()) cd[0] = contour[0]->displacement;
    for (size_t i = 1; i < contour.size(); i++)
        cd[i] = cd[i - 1] + contour[i]->displacement;

    return cd;
}

void TreeNode::merge_subtrees(float displacement) {
```

```
    /** "Merge" the subtrees of this node such that they have a horizontal separ
     *  by setting an appropriate displacement for this node
     *
     *  displacement: Default displacement value if this is a leaf node
     *  - Should be -1 for left nodes and 1 for right nodes
     */

    // Set displacements for edge cases
    if (!this->left() && !this->right()) {
        // Base Case: This is a leaf node (i.e. no children)
        this->displacement = displacement;
        return;
    }
    else if (!this->left() || !this->right()) {
        // Edge Case: One child is NULL
        if (this->left()) this->left()->displacement = 1;
        else this->right()->displacement = -1;
    }

    // Postorder traversal
    if (this->left()) this->left()->merge_subtrees(-1);
    if (this->right()) this->right()->merge_subtrees(1);

    // Merge subtrees (if they exist)
    if (this->left() && this->right()) {
        // Because by default, this node has displacement zero,
        // it will be centered over its children
        float subtree_separation = (this->distance_between(this->left(), this->r
        this->left()->displacement = -subtree_separation;
        this->right()->displacement = subtree_separation;
    }
}

std::vector<TreeBase*> TreeBase::left_contour() {
    /** Return a list with the left contour of a vertex */
    std::vector<TreeBase*> node_list;
    this->left_contour(0, node_list);
    return node_list;
}

void TreeBase::left_contour(int depth, std::vector<TreeBase*>& node_list) {
    if (node_list.size() <= depth) node_list.push_back(this);
    if (this->left()) this->left()->left_contour(depth + 1, node_list);
    if (this->right()) this->right()->left_contour(depth + 1, node_list);
}
```

```
std::vector<TreeBase*> TreeBase::right_contour() {
    /** Return a list with the right contour of a vertex */
    std::vector<TreeBase*> node_list;
    this->right_contour(0, node_list);
    return node_list;
}

void TreeBase::right_contour(int depth, std::vector<TreeBase*>& node_list) {
    if (node_list.size() <= depth) node_list.push_back(this);
    if (this->right()) this->right()->right_contour(depth + 1, node_list);
    if (this->left()) this->left()->right_contour(depth + 1, node_list);
}
```