1 Graph Drawing: Bäume

1.1 Einführung

Graph Drawing beschäftigt sich mit Methoden zur geometrischen Darstellung von Graphen und Netzwerken. Solche Visualisierungen von Netzwerken und darin enthaltenen Daten sind ganz offensichtlich wichtig für das Verständnis und ergo weit verbreitet: Software Engineering(UML Diagramme), Datenbanken, Echtzeit Systeme, VLSI Design, Medizin/ Bioinformatik, etc., etc.. Dabei kann es oft mühsam sein, das Layout (sprich Knoten- und Kantenplazierung) „per Hand“ zu erstellen. Dies ist natürlich insbesondere dann der Fall, wenn sich die Graphen häufig ändern und/oder eine große Anzahl von Knoten und Kanten enthalten. In solchen Fällen sind Layout-Algorithmen sehr praktisch. Diese Algorithmen sollen einen Graphen klar und einfach in eine Ebene einbetten, so daß gewisse strukturelle Eigenschaften möglichst leicht erkannt werden können

Ästhetische Kriterien definieren dabei die Zielfunktion(en), nach denen die Graph Drawing Algorithmen optimieren sollen. Neben spezifischen Kriterien, die durch das jeweilige Anwendungsgebiet gegeben sind, wird oft nach folgenden allgemeinen Kriterien optimiert, da diese generell die Lesbarkeit erhöhen.

- **Kantenkreuzungen** Kantenkreuzungen sollten minimiert werden. Idealerweise wird eine planare Darstellung gefunden. Leider ist nicht jeder Graph planar.
- **Zeichenfläche** Auch die Fläche sollte minimiert werden, denn offensichtlich sind die Darstellungsmedien (Bildschirm, Ausdruck) beschränkt.
- **Kantenlängen** Minimierung der Summe aller Kantenlängen, oder der maximalen Länge einer Kante, oder der Varianz der Kantenlängen, oder ...
- **Kantenknicke** Analog ...
- **Symmetrie**

Offensichtlich sind die o.g. Optimierungskriterien einander entgegengesetzt. Daher sind Tradeoffs oft unvermeidbar. Auch hat sich herausgestellt, dass die meisten der zugrundeliegenden Probleme algorithmisch hart sind. Deswegen werden oftmals Heuristiken und approximative Strategien eingesetzt. Neben diesen ästhetischen Kriterien sollen manchmal auch noch weitere Nebenbedingungen erfüllt werden. Z.B. soll ein bestimmter Knoten eine bestimmt Position erhalten, oder eine Teilmenge der Knoten benachbart gezeichnet werden, Subgraphen könnten vordefinierte Formen haben, oder Pfade im Graphen vordefinierte Richtungen.

Im Folgenden wird ein einfacher Algorithmus vorgestellt, mit dem ein kompaktes hierarchisches Layout von einem gewurzelten Binärbaum erstellt werden

1.2 Teile und Herrsche

Der Divide & Conquer Ansatz ist im Graph Drawing weit verbreitet. Dabei wird der Graph zunächst in Subgraphen aufgeteilt, die Subgraphen werden rekursiv gezeichnet,

und zuletzt werden die Darstellungen der Subgraphen geeignet zusammengefügt. Am Besten eignet sich der Divide & Conquer Ansatz natürlich für Graphklassen die einfach rekursiv zerlegt werden können wie Series-Parallele Graphen, oder, wie nachfolgend behandelt, Bäume. Die Grundidee des ebenenweisen Bäumzeichnens durch Divide & Conquer beschreibt der nachfolgende Algorithmus. Dabei sei d_h (d_v) der gewünschte horzontale (vertikale) Mindestabstand.

Algorithmus 1.1 Ebenen-weises Baumlayout

func DrawBinBaum(Knoten v)

if v hat keine Kinder **then**
    Zeichne den Knoten.
else
    // Divide:
    DrawBinBaum(linker Teilbaum von v)
    DrawBinBaum(rechter Teilbaum von v)

    // Conquer:
    Verschiebe die beiden Teilbäume so, dass
    sie den Abstand $2 * d_h$ Einheiten haben. Zeichne
    die Wurzel v eine Einheit (d_v) über, und horizontal
    mittig zwischen den Teilbäumen. Wenn v nur einen
    rechten(linken) Teilbaum hat, zeichne v eine
    Einheit links(rechts) von der Wurzel des Teilbaums.
fi

Bei dem Algorithmus wird an jedem Knoten (ausgehend von der Wurzel v) rekursiv für beide Teilbäume das Layout berechnet, um daraus ein Gesamtlayout (Knoten und beide Teilbäume) zu ermitteln. Die Funktion *DrawBinBaum* liefert als Ergebnis das Layout eines Baumes T mit Wurzel v folgendermaßen: an jedem Knoten $\neq v$ von T ist die relative Verschiebung zu dessen Vorgänger (im Sinne der Postorder Traversierung) gespeichert.

Da die Verschiebe-Operation bei kompletten (Unter-)bäumen nicht effizient realisierbar wäre, werden die absoluten (x-)Positionen in einer 2. Phase berechnet:

- **Phase 1** Zunächst wird durch die *Postorder* Traversierung *DrawBinBaum* rekursiv der horizontale Abstand jedes Knotens bzw. dessen x-Position *relativ* zu seinem jeweiligen Elterknoten bestimmt. Dazu muss in jedem Schritt der Abstand der Teilbäume effizient berechnet werden. Dies geschieht durch *Konturlisten*.

- **Phase 2** In der 2.Phase werden die relativen x-Positionen durch eine *Preorder* Traversierung in absolute Positionen umgewandelt, indem die Verschiebungen der Knoten von der Wurzel aus akkumuliert werden. Gleichzeitig wird die Ebeneneinteilung der Knoten (d.h. ihre y-Position) durch Bestimmung der $Tiefe$ der Knoten im Baum festgelegt.

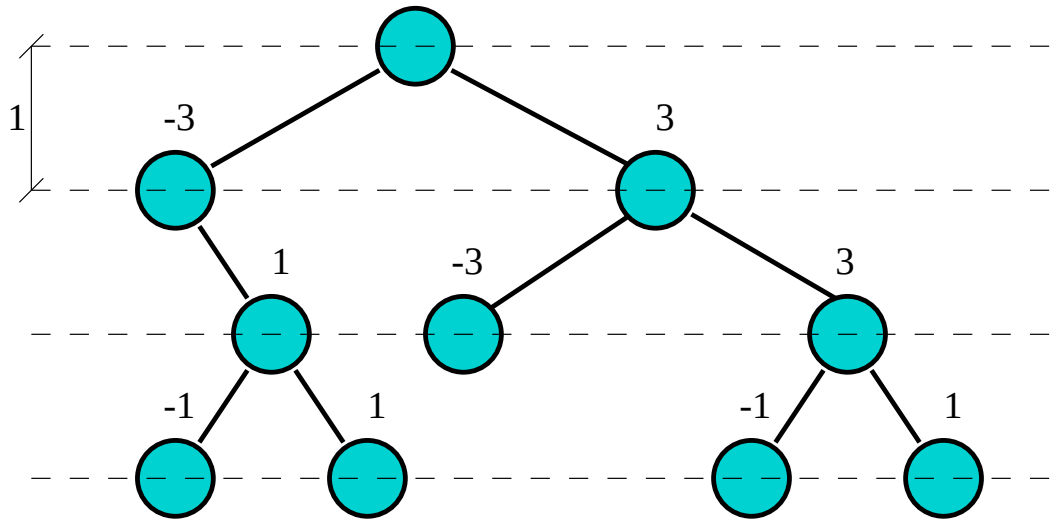

Abbildung 1: Beispiel für ein hierarchisches Layout eines Baumes

Die 2. Phase ist offensichtlich unproblematisch und in linearer Zeit durchführbar, im folgenden Abschnitt wird deshalb die 1.Phase genauer betrachtet.

1.3 Abstandsberechnung

Um die Postorder Traversierung effizient (d.h. in Linearzeit) durchführen zu können, werden *Konturlisten* verwendet.

Definition 1 *Die* linke Kontur *eines binären Baumes T mit Höhe h ist die Knotenfolge v_0,...,v_h, wobei v_i der linkeste Knoten der Tiefe i in T ist. Analog ist die* rechte Kontur *definiert.*

Um nun einen Knoten v in der Postorder Phase zu bearbeiten, werden die linke Konturliste des rechten Teilbaums und die rechte Konturliste des linken Teilbaums bis zum Ende der kürzeren Liste durchlaufen. Dabei werden die x-Positionen der Knoten in den Konturlisten als kumulative Verschiebung bezüglich v berechnet. Der minimale Abstand der Teilbäume ergibt sich nun durch den minimalen Abstand zweier Knoten gleicher Tiefe in den Konturlisten.

Nun muss noch die Kontur des an v gewurzelten Teilbaums T' aus den Konturen des linken (T'') und rechten Teilbaums (T''') konstruiert werden:

- **Falls T' und T'' die gleiche Höhe haben** ist die linke Kontur von T' gleich der linken Kontur von T'' plus v, und die rechte Kontur von T' gleich der rechten Kontur von T''' plus v.

- **Falls T''' höher als T'' ist**, ist die rechte Kontur von T' gleich der rechten Kontur von T''' (plus v). Die linke Kontur von T' ist gleich der linken Kontur von T'' (plus v, und dem Teil der linken Kontur von T''', um den T''' höher ist als T''.

- **Der Fall, dass T'' höher als T''' ist**, wird analog zum vorhergehenden Fall behandelt.

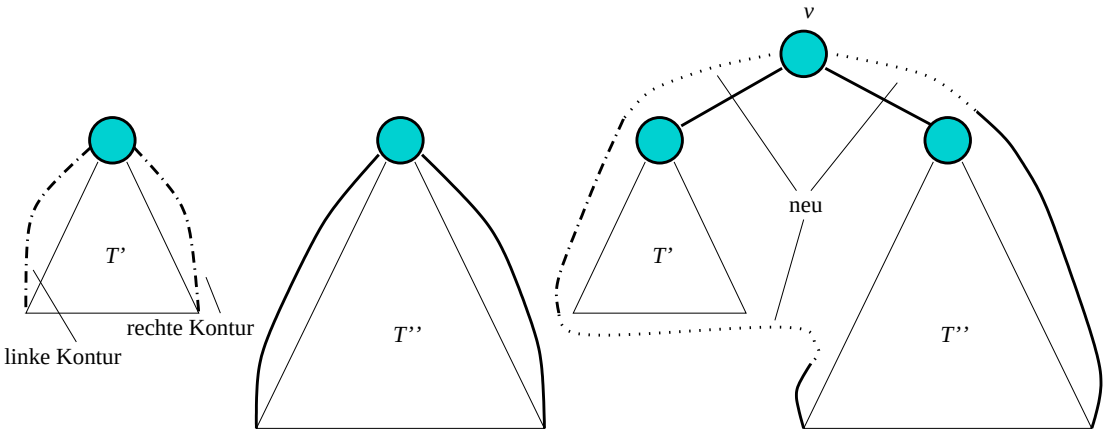

Abbildung 2: Zwei Teilbäume zusammenfügen

Um die beschriebenen Operationen effizient zu implementieren, bieten sich verkettete Listen an. Zeiger/Referenzen auf diese Konturlisten sollten dann zweckmässigerweise von der aufgerufenen Funktion zurückgegeben werden. Damit ist die Laufzeit an jedem Knoten proportional zur Höhe des kleineren Teilbaums. Die Gesamtlaufzeit wird durch

$$\sum_{v \in T}(1 + min\{h''(v), h'''(v)\}) = n + \sum_{v \in T} min\{h''(v), h'''(v)\} = O(n)$$

beschrieben, wobei $h''(v)$ ($h'''(v)$) die Höhe des linken (rechten) Teilbaums an v angeben. Ein kurzes Argument, warum $\sum_{v \in T} min\{h''(v), h'''(v)\} = n$ ist, wird in der Übungsstunde präsentiert, oder aber in [1].

Der oben beschriebene Algorithmus wurde der Übersichtlichkeit wegen nur auf binären Bäumen definiert. Eine Verallgemeinerung auf allgemeine, gewurzelte Bäume sollte aber nicht zu schwer sein.

Literatur

[1] Battista G., Eades P., Tamassia R., Tollis I.: *Graph Drawing*, Prentice Hall, 1999.