Karlsruhe Institute of Technology

Visualisierung von Argumentkarten - Inkrementelle Layouts

Diplomarbeit
von

Stefan Altmayer, Lukas Barth, Eric Braun,
David Hopf, Frederike Neuber

An der Fakultät für Informatik
Institute of Theoretical Computer Science

Betreuende Mitarbeiter: Ignaz Rutter

Bearbeitungszeit: 1st October 2014 – 31th March 2015

Inhaltsverzeichnis

1. Content Chapters

The content chapters of your thesis should of course be renamed. How many chapters you need to write depends on your thesis and cannot be said in general.

1.1. Einführung

In dieser Arbeit beschäftigen wir uns mit der Frage, wie Argumentkarten inkrementell gelayoutet werden können. Argumentkarten sind dabei abstrakt betrachtet gelayoutete Graphen, bei denen Knoten nicht punktförmig, sondern durch Rechtecke repräsentiert werden. Argumentkarten tragen darüber hinaus noch einiges an Semantik, so können beispielsweise Kanten entweder Unterstützung oder Angriffe von Argumenten untereinander darstellen, oder die Knoten sind beschriftet und sogar nummeriert. Für unsere Ansätze werden wir diese Semantik nicht benutzen.

Das Problem des inkrementellen Layouts besteht darin, eine bereits existierende Argumentkarte zu verändern und für diese veränderte Argumentkarte ein neues Layout zu finden. Das neue Layout soll dabei in einer Weise gefunden werden, die die "'mentale Karte"' erhält, dass also ein menschlicher Betrachter, der die alte Karte kannte, sich in der neuen Karte schnell zurecht findet. Dies erreichen wir, indem wir die neue Karte so layouten, dass sie der alten, bekannten Karte ähnlich ist.

Wir haben uns außerdem dafür entschieden, als Veränderungen der Karte ausschließlich das Hinzufügen eines einzelnen Knotens zu betrachten. Die Entfernung einer kleinen Anzahl von Knoten ist vermutlich gut durch einfaches Weglassen der Knoten umsetzbar, und daher algorithmisch nicht interessant. Die meisten anderen Operationen, wie zum Beispiel das Verschieben von Knoten, können dann als eine Aufeinanderfolge von Entfernungs- und Einfügeoperationen dargestellt werden.

1.1.1. Das Problem

Das Problem stellt sich aus Sicht eines Informatikers dar wie folgt. Gegeben sind:

1. ein Graph $G = (V, E)$
2. je ein Rechteck pro Knoten, oder genauer eine Funktion $b : V \to R^2$, die jedem Knoten seine Höhe und Breite zuordnet,
3. eine (überlappungsfreie) geometrische Einbettung von G, also eine Funktion $\omega : V \to R^2$, die jedem Knoten einen Punkt in der Ebene zuweist,

4. ein einzufügender neuer Knoten $a \in R^2$ mit einer Menge M von Knoten, zu denen er adjazent ist
5. eine Funktion $\psi : M \to R^2$, die zu jedem Knoten $m \in M$ adjazent zu a den Vektor zuordnet, der angibt, wie a relativ zu m platziert werden soll.

Gesucht ist dann eine neue geometrische Einbettung ω' von $(V \cup \{a\}, E \cup \{(a, m) \mid m \in M\}$, die "'ähnlich"' ist zu ω.

Ähnlichkeit

Bei der Definition der zu optimierenden Ähnlichkeit bieten sich mehrere Ansätze an: Zunächst die *absolute Ähnlichkeit*, bei der schlicht die Summe der Entfernungen aller Positionen von ihren Ursprungspositionen minimiert wird, also

$$\min_{\omega'} \sum_{v \in V} |\omega(v) - \omega'(v)| \tag{1.1}$$

Diese Definition erschien uns jedoch zu starr. Insbesondere sind wir der Meinung, dass der Erhalt der "'mentalen Karte"' hauptsächlich davon abhängig ist, wie die Positionen der Knoten zueinander sind. Ein in unseren Augen besserer Ansatz könnte also lauten:

$$\min_{\omega'} \sum_{(v,w) \in V \times V} |(\omega(v) - \omega(w)) - (\omega'(v) - \omega'(w))| \tag{1.2}$$

Aber auch hier geht in die Optimierung noch die relative Position von zwei Knoten ein, die vielleicht an ganz unterschiedlichen Enden der Argumentkarte stehen, und bei denen eine Abweichung von der gewünschten relativen Position weniger stört als bei zwei Knoten, die dicht beieinander, und somit vermutlich auch in einem engeren semantischen Zusammenhang stehen. Dies könnte man nun über Gewichtungen versuchen zu lösen, wir haben uns aber entschieden, stattdessen nur die relativen Positionen von adjazenten Knoten zu betrachten:

$$\min_{\omega'} \sum_{(v,w) \in E} |(\omega(v) - \omega(w)) - (\omega'(v) - \omega'(w))| \tag{1.3}$$

Zusätzlich muss noch die Position des neu einzufügenden Knotens berücksichtigt werden. Wäre nicht angegeben, wie der neue Knoten relativ zu seinen Nachbarn positioniert werden soll, so würde er vermutlich einfach außerhalb der bisher gelayouteten Argumentkarte platziert, da so der Term 1.3 durch die alten Positionen optimiert wird, ohne dass eine Überlappung entstehen würde. Wir fügen also hinzu:

$$\min_{\omega'} \sum_{m \in M} |(\psi(m)) - (\omega'(a) - \omega'(m))| \tag{1.4}$$

Verschiebung versus Skalierung

Wenn wir für einen Moment davon ausgehen, dass das neue Layout nicht überlappungsfrei sein muss, so werden (eine geeignete Wahl von ψ vorausgesetzt) die Summen 1.4 und 1.3 optimiert (zu Null) dadurch, dass alle alten Knoten auf ihren Plätzen bleiben und der neue Knoten am gewünschten Punkt (mit alten Knoten überlappend) eingefügt wird. Ausgehend von diesem Layout sehen wir zwei grundlegend verschiedene Möglichkeiten, den notwendigen Platz für ein überlappungsfreies Layout zu schaffen: Zunächst können natürlich einfach einzelne Knoten verschoben werden. Dies "'verzerrt"' die relativen Positionen von Knoten und erhöht die Summen 1.4 und 1.3. Andererseits ist es aber auch möglich, einfach alle

Kantenlängen mit einem gewissen Faktor zu multiplizieren, was dadurch, dass das Layout insgesamt größer wird, äquivalent dazu ist, die Knotengrößen kleiner zu skalieren. Diese zweite Möglichkeit erhöht ebenfalls beide Summen, wir halten dies allerdings eigentlich für falsch: Für den Betrachter ändert sich nicht die relative Lage der Knoten zueinander, die Knoten werden nur kleiner. Wir führen daher einen Skalierungsfaktor c ein. Natürlich muss c so gewählt werden, dass er möglichst nah an 1 gelegen ist. Insgesamt ergibt sich somit als Optimierungsfunktion:

$$\begin{aligned}\min_{\omega'} \quad & \alpha \sum_{(v,w)\in E} |c(\omega(v)-\omega(w)) - (\omega'(v)-\omega'(w))| \\ & +\beta \sum_{m\in M} |(c\psi(m)) - (\omega'(a)-\omega'(m))| \\ & +\gamma(|1-c|) \end{aligned} \tag{1.5}$$

Zeilen 1 und 2 entsprechen dabei den Summen 1.4 und 1.3 unter Berücksichtigung von c. Die Koeffizienten α, β und γ dienen dazu, die einzelnen Teile der Optimierungsfunktion gegeneinander zu gewichten.

1.2. Heuristische Ansätze

TODO Heuristik-Blah

1.2.1. Spring Embedder

TODO allgemeines über Spring Embedder.[**?**]

Wenn wir davon ausgehen, dass im gelayouteten Graphen nur die Kantenlängen, und nicht die Richtung der Kanten, entscheidend ist, ist der Term 1.3 trivial in einen Spring Embedder umzusetzen. Dies wäre insbesondere geeignet, falls der Graph eine hohe Steifigkeit aufweist, da in diesem Fall die Kantenlängen allein das Layout des Graphen festlegen.[1]

Leider sind Argumentkarten vergleichsweise dünn besetzte Graphen, und daher ist eine geringe Steifigkeit anzunehmen. Um hier zu dem von Term 1.3 gewünschten Ergebnis zu kommen, wäre es vermutlich notwendig, zusätzlich orbitale Federn um die Knoten herum zu legen, die die Winkel der ausgehenden Kanten relativ zueinander an das gewünschte Ergebnis anzunähern. Siehe Zeichnung TODO. Solche kreisförmigen Federn sind bereits beispielsweise erfolgreich für Lombardi-Zeichnungen mit optimaler Winkelauflösung verwendet worden[**?**], wobei hier die Kräfte den kompletten Knoten rotiert haben. Kreisförmige Federn zischen Kantenpaaren konnten wir in der Literatur bisher keine finden. TODO nochmal überprüfen!

1.2.2. Seam Carving

Seam Carving ist eine verbreitete Methode in der Bildbearbeitung, die von S. Avidan und A. Shamir vorgestellt wurde.[**?**] Das Ziel von Seam Carving ist es, ein Bild zu verkleinern, indem Pixel weggelassen werden, ohne visuell den Eindruck einer verzerrung zu erwecken. Hierfür verwendet Seam Carving eine Technik, die die Autoren als "'content aware"' bezeichnen: Die wegzulassenden Pixel werden als durchgänginge Streifen, die sogenannten "'Seams"' gewählt. Bei der Auswahl der Seams wird darauf geachtet, dass diese möglichst wenig durch

[1] Dies ist nicht ganz korrekt: Es können mehrere unterschiedliche geometrische Einbettungen mit denselben Kantenlängen existieren, z.B. indem Dreiecke umgeklappt werden. Da ein Spring Embedder allerdings nur kleine, inkrementelle Änderungen der Einbettung vornimmt, ist dies hier nicht relevant.

Bereiche des Bildes verlaufen, in denen ein Weglassen störend wäre. Hierfür beschreiben die Autoren verschiedene Energiefunktionen auf den Bildern.

Seam Carving hat außerdem bereits Anwendung im Bereich des Graph Drawings gefunden, beispielsweise beim Layout von Word Clouds.[**?**] Das Problem war hier, nach einem initialen Layout, das die Knoten des Graphens nach bestimmten vorgegbenen Abständen voneinander mittels Multidimensional Scalings verteilt hat, den überflüssigen Platz zu entfernen. Hierbei sollten die relativen Positionen der Knoten möglichst wenig verzerrt werden. Die Autoren beschreiben hierfür die für das Seam Carving verwendete Energiefunktion als Summe der Quadrate der inversen Abstände von den Knoten.

Ein ähnlicher Ansatz ist auch für unser Problem denkbar. Wie in Absatz 1.1.1 beschrieben, ist es immer möglich, alle Kanten einfach so lange zu skalieren, bis der neue Knoten überlappungsfrei platziert werden kann. TODO Paragraph referenzieren? Das Problem besteht hier darin, dass das gesamte Layout hierbei beliebig groß werden kann, zur Betrachtung also das Bild insgesamt verkleinert werden muss, was einer Skalierung der Knotergrößen gleichkommt.

1.3. Ein Integer Linear Program

...

1.3.1. Implementation

1.3.1.1. User Interface

In diesem Kapitel wird die grafische Oberfläche, welche Teil des Prototypen ist, mit ihren Funktionalitäten vorgestellt. Außerdem wird die Bedeutung für einen eigenen Editor im Rahmen des Seminars erläutert.

Beschreibung

Die GUI ist eine Webanwendung, welche mit Javascript programmiert wurde. Hierzu wurden die Javascript-Bibliotheken Cytoscape.js[**?**] und Angular.js[**?**] benutzt. Cytoscape.js bietet bereits Funktionalitäten, um einen Graphen darzustellen und zu manipulieren. Angular.js wurde als Javascript Framework benutzt, um die Interaktion mit Html und Javascript zu erleichtern. Ansonsten wurde reines Javascript benutzt, um die weiteren Features zu implementieren.

Da diese Arbeit sich auf das inkrementelle Einfügen von Knoten fokussiert, wurde der Editor mit einer Funktion ausgestattet, welche es erlaubt, einen Knoten lediglich mit ein paar Mausklicks einzufügen und die Größe festlegen zu können. Es wird verhindert, dass ein neu eingefügter Knoten mit einem bestehenden Knoten überlappt, indem für den neuen Knoten genug Platz geschaffen wird. Der Algorithmus um Platz zu schaffen unterstützt die vier folgenden Modi:

- es wird Platz gemacht, indem alle Knoten in eine der vier orthogonalen Richtungen verschoben werden, damit unmittelbar links, rechts, oberhalb und unterhalb von dem neuen Knoten kein anderer Knoten sich befindet. Alle Knoten, die in die gleiche Richtung verschoben werden, werden um den gleichen Betrag verschoben. Dieser Betrag wird so bestimmt, dass alle Knoten einen minimalen Abstand von dem eingefügten Knoten einhalten.
- der erste Modus aber es wird nur links und rechts Platz gemacht.
- der erste Modus aber es wird nur oben und unten Platz gemacht.

- die Knoten werden um den gleichen Betrag nach außen verschoben, damit kein anderer Knoten mit dem Neuen überlappt

In jedem Modus wird ein Mindestabstand zwischen dem neu eingefügten Knoten und allen anderen eingehalten. Somit kommt es zu keinen Überlappungen der Knoten, die zu nah beieinander sind. Abhängig vom Modus der gerade aktiv ist zeigt die GUI ein entsprechendes Overlay um anzuzeigen, wo überall Platz gemacht wird. Nachdem ein Knoten mit einem der vier Modi eingefügt wurde, kann dessen Größe festgelegt werden. Anschließend hat der Benutzer die Möglichkeit über einen Button das Layout des Graphen anzupassen. Somit wird der Platz, der vorher gemacht wurde, wieder minimiert, wobei versucht wird, gewisse Layoutbedingungen nicht zu verletzen (die wird näher in FIXME beschrieben). Des Weiteren zeigt jeder Knoten einen Titel und eine Beschreibung an. Somit können Informationen von Thesen und Argumenten in den Knoten angezeigt werden. Dies ist eine grundlegende Anforderung an eine Argumentkarte.

Außerdem können Knoten horizontal sowie vertikal auf einer Linie angeordnet werden. Dies hilft dem Benutzer, den Graphen nach seinen Bedürfnissen zu gestalten. Hier können verschiedene Funktionen genutzt werden, um neue Position der Knoten zu berechnen. In dem Prototypen wird der Mittelwert der x- oder y- Koordinate berechnet und als neue Position der auszurichtenden Knoten benutzt. Eine weitere Funktion könnte die Knoten an dem zuletzt markierten Knoten ausrichten oder an einem Knoten ausrichten, den man explizit auswählt. Da es sich nur um einen Prototypen handelt wurden diese Funktionen jedoch nicht implementiert.

Die letzte wichtige Funktion ist das Importieren von *.graphml* Dateien. Somit können Argumentkarten aus dem Argunet-Editor importiert werden. Dies erlaubt ein schnelles Testen der Algorithmen. Es werden Position und Größe der Knoten, Kanten zwischen den Knoten und die Beschriftung der Knoten importiert. Falls man ein orthogonales Kantenrouting implementiert hätte, könnte man hier auch noch die Position der Knicke auf den Kanten importieren, falls diese in der *.graphml*-Datei vorhanden sind.

Bedeutung für das Seminar

Die Anforderung der Philosophiestudenten, war, dass der Graph sich nur wenig ändern sollte, wenn ein Knoten eingefügt wird. Die verschiedenen Modi dienen dazu, dass der Benutzer entscheiden kann, wieviel sich im Graphen ändert. Hier muss angemerkt werden, dass nur der erste Modus garantieren kann, dass sich orthogonale Kanten nicht überlappen würden. Da diese Arbeit sich jedoch ausschließlich auf die Position der Knoten bezieht und die Kanten ausklammert, wird dieses Problem nicht weiter diskutiert.

Der anschließende Layoutalgorithmus versucht dann so weit es geht das ursprüngliche Layout wieder herzustellen. Wie ähnlich der Graph nach dem Einfügen des Knoten aussieht, hängt stark von der Position und der Größe des neuen Knoten ab. Die zusätzliche Ausrichtfunktion dient dazu, dass der Benutzer in das Layout eingreifen kann, und noch mehr mitbestimmen kann wie das endgültige Layout aussieht.

Diese beiden Funktionen sind sinnvoll, da der Benutzer, welcher eine Argumentkarte erstellt, meistens bereits weiß, wie die Knoten angeordnet werden sollen. Insbesondere weiß der Benutzer auch wie ein neu eingefügter Knoten positioniert werden soll. Der Benutzer ist jedoch nicht darauf angewiesen diese Layoutfunktionen zu benutzen. Ohne eine Anpassung vom Benutzer versucht der ILP-Algorithmus eine optimale (und somit in den meisten Fällen eine für den Benutzer wünschenswerte) Lösung zu finden. Die GUI bietet dem Benutzer Funktionen um schnell einen Graphen inkremetell aufzubauen und zugleich hat er die Freiheit selbst mit zu entscheiden, wie er den Graphen gestaltet.

Anhang

A. Appendix Section 1

ein Bild

Abbildung A.1.: A figure