Balancing Performance and Visual Appeal in Dynamic Graph Labeling

Nils Henrik Seitz* (Mat.Nr. 218205308)

Faculty of Computer Science
University of Rostock

Abstract

To tackle the NP-hard problem of graph labeling, a greedy approach was used to generate label positions. Conflict Detection is done fastly with simple bounding boxes, so that valid labels can be stored quick and efficient in proper data structures like maps and quadtrees. Animations are used to ease changes of label positions or visibility on the eye. All these techniques combined make for a fluid and visually appealing experience, that does justice to dynamic labeling.

Um das NP-Harte Problem des Graph Labeling in dynamischen Graphen anzugehen, wird ein gieriger Ansatz genutzt, um Label Position zu generieren. Mit simplen Bounding Boxes wird schnelle Konfliktermittlung betrieben, sodass valide Positionen durch geeignete Datenstrukturen wie Maps und Quadtrees effizient und zügig gespeichert werden. Animationen runden Änderungen der Labelpositionen oder -sichtbarkeit visuell ab. Diese Maßnahmen führen im Gesamtbild zu einem flüssigen, visuell ansprechenden Ergebnis, das der Natur von dynamischem Labeling gerecht wird.

Keywords: automatic label placement, dynamic labeling, graph labeling, interactive labeling

Bemerkung: Dieses Projekt wurde in Kooperation mit Nico Trebbin erstellt. Der Bericht wurde als eigenständiges Werk von dem Autor angefertigt. Bilddateien haben gegebenenfalls einen gemeinsamen Ursprung als Material für die Präsentation im Rahmen des Projektes.

Viele der Kernbegriffe des Projektes werden in diesem Bericht im Original verwendet, um Konsistenz mit der Codebasis und den Präsentationen des Projektes zu erhalten.

*ns464@uni-rostock.de

1. Einleitung

Graphen sind wohl die mit am meisten genutzte Datenstruktur der Informatik. Durch sie lassen sich viele verschiedene Sachverhalte modellieren. Mit `iGraph.js` existiert als praktische Grundlage ein Tool zur interaktiven und dynamischen Visualisierung von Graphen von Tominski et al.[8]

Das Problem ist, dass ohne die Namen der Knoten und Kanten dem Graph recht wenig Bedeutung abgewonnen werden kann, es fehlt Kontext. Der Kontext kann durch ein Labeling des Graphen klarer werden. Dazu sind allerdings zwei Dinge zentral:

Erstens sollte das Labeling visuell ansprechend sein, das heißt, es muss intuitiv klar sein, welches Label zu welchem Knoten oder zu welcher Kante gehört und es sollten keine Überdeckungen entstehen.

Zweitens sollte das Labeling performant sein, da selbst die schönste Anordnung von Labels nicht nützlich ist in einem dynamischen und interaktiven Umfeld, wenn sie zu lange zum Errechnen braucht. Den bestmöglichen Kompromiss aus Labeling Qualität und Performance zu finden, ist die Kernaufgabe dieses Projektes.

Dazu haben wir eine Labeling Pipeline implementiert:

1. Koordinaten der Knoten/Kanten updaten (von `iGraph.js`)
2. Potentielle Labelpositionen generieren
3. Konfliktermittlung
4. Tatsächliche Labelposition festlegen und speichern
5. Visualisierung

Natürlich ist diese nicht ganz so statisch und sukzessiv wie hier vereinfacht beschrieben. Zum Beispiel finden Schritt 2 und 3 verschränkt statt, also werden nicht erst alle potentiellen Positionen generiert und dann nach Konflikten gesucht. Stattdessen finden sie abwechselnd statt.

Dies spiegelt den gierigen Ansatz wider, der dieser Implementation zu Grunde liegt. Dadurch soll die Kernaufgabe des Projektes umgesetzt werden, also schnell möglichst gute Labelpositionen zu finden. Dies schien die sinnvollste Option in Anbetracht der Tatsache, dass das Labeling Problem NP-Hard ist.[4]

Es sei abschließend gesagt, dass diese Pipeline und die verwendeten Techniken und Methoden nicht ausschließlich für Graph Labeling genutzt werden müssen. Beispielsweise wäre auch eine Nutzung für Radare von etwa Schiffen oder Flugzeugen vorstellbar, bei denen andere Schiffe/Flugzeuge mit Namen versehen werden sollen. Formal genommen ist das ja ein Graph ohne Kanten - sollte nur darstellen, dass die Techniken und Algorithmen sehr generisch sind.

Als hauptsächliche theoretische Grundlage diente zur Spezifikation der Pipeline sowie der Labeling Verfahren und Konfliktermittlung eine Arbeit von Luboschik et al.[3], in der das Vorgehen beschrieben wird.

2. Labeling

Labeling bezeichnet den Prozess, ein Objekt sichtbar mit seinem Namen oder anderen Informationen zu versehen. Somit gilt es für einen Graphen, seine

Knoten und seine Kanten zu labeln.

In erster Linie werden hier die verschiedenen Verfahren erklärt, wie Labelpositionen abhängig von ihrem Objekt und seiner Position erzeugt werden. Oft wird im folgenden über "potentiellen"Labelpositionen geschrieben - der Grund dafür ist, dass, sobald mehrere Objekte gelabelt werden müssen, sich deren potentielle Labelpositionen gegenseitig überdecken können. Dieses Problem wird in Conflict Detection und Bounds aufgelöst. Somit wird sich hier auf die Verfahren für ein individuelles Objekt konzentriert.

2.1. Knotenlabeling

Beim Labeling der Knoten sind als geometrische Repräsentation ein Kreis, das heißt, x, y-Koordinaten und ein Radius, sowie der entsprechende Name des Knotens durch die Daten von `iGraph.js` gegeben.

Nun ist es Aufgabe des Labelings, den Text des Labels erkennbar und visuell ansprechend in Nähe des dazugehörigen Knotens zu platzieren. Die Breite und die Höhe eines Labels werden von der ausgewählten Schriftart (siehe Konfiguration) beeinflusst. Aktuell ist die Höhe auf eine Zeile begrenzt in der Implementation. Die Breite eines Labels hängt zusätzlich von dem entsprechenden Namen bzw. der Anzahl der Buchstaben ab.

So ergeben sich die Höhe und Breite des Labels (sowie der Text selbst) und es muss eine Position in x, y-Koordinaten in Abhängigkeit von dem entsprechenden Knoten gefunden werden.

Die Reihenfolge des Labelings ergibt sich aus einer einmaligen Sortierung. Diese ist absteigend und sortiert nach der Größe der Radien der Knoten. Ein größerer Radius steht für eine höhere Relevanz, denn er bedeutet mehr Nachbarn und daher sind diese Knoten vom größerem Interesse.

Zur Generierung der Position eines potentiellen Labels werden nacheinander verschiedene Verfahren verwendet, um möglichst effizient Labelpositionen zu generieren mit möglichst wenig Überdeckung der einzelnen potentiellen Positionen und so, dass niemals der dazugehörige Knoten überdeckt wird:

2.1.1. 4-Position-Model

Das 4-Position-Model positioniert die potentiellen Labels möglichst nahe an dem korrespondieren Knoten in den Himmelsrichtungen Nord-Ost, Nord-West, Süd-Ost und Süd-West. Die erste Position ist Nord-Ost, also rechts-oben, und von dort an wird entgegen x, y-Koordinaten ein weiteres Label generiert (wenn nötig).

Dieses und das nachfolgende Verfahren sind durch die gängigen Standards der Kartografie inspiriert. Auch dort werden eben beschriebene Positionen generiert, allerdings in leicht veränderter Reihenfolge.[1]

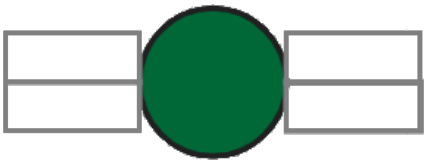

Abbildung 1: Labelpositionen im 4-Position-Model von ausgehenden vom dazugehörigen Knoten

2.1.2. 8-Position-Model

Im 8-Position-Model werden die potentiellen Labelpositionen in den Hauptthimmelsrichtungen Ost, Nord, West, Süd generiert. Startpunkt ist Osten, also rechts, und wieder wird entgegen des Uhrzeigersinns ggf. ein weiteres Label generiert.

Weiterhin werden die Nord-/Süd-Position im Abstand einer Labelhöhe und die Ost-/West-Position im Abstand einer Labelbreite vom dazugehörigen Knoten positioniert. Dies wird gemacht, um Überdeckung mit den zuvor im 4-Position-Model generierten Positionen zu vermeiden, da, wenn zuvor das 4-Position-Model schon keine Position finden konnte, das 8-Position-Model ohne den Abstand dann wahrscheinlich ebenfalls keine Position finden würde.

Das führt dazu, dass vor allem die Ost-/West-Position eher weit entfernt von dem korrespondieren Knoten sind (da Labeltexte bzw. Namen in der Regel eher breit als hoch sind). Die Kartografie verwendet diese Positionen nicht.[1]

In der Implementation wurde dieses Problem dadurch behoben, dass eine "Hilfslinie", als visueller Indikator, das Label mit seinem entsprechenden Knoten verbindet, sodass die Zugehörigkeit schneller ersichtlich wird. Diese Technik wird im Spiral-Model nochmals angewandt, da hier dasselbe Problem besteht.

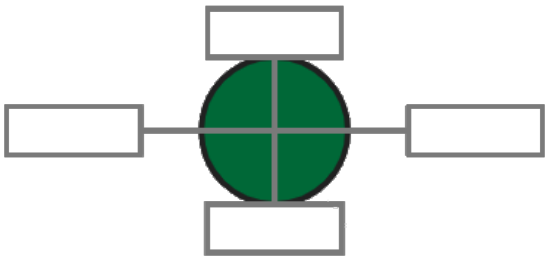

Abbildung 2: Labelpositionen im 8-Position-Model von ausgehenden vom dazugehörigen Knoten

2.1.3. Slider-Model

Das Slider-Model ist ein aufwendigeres Verfahren als die vorhergehenden beiden. Die Idee hier ist, die Eckpunkte des 4-Position-Model zu nehmen und dann Labelpositionen dazwischen zu generieren, in dem man das potentielle Label stückweise zum nächsten Eckpunkt verschiebt. Der Inkrement-Wert zum Verschieben ist konstant (siehe Magic Constants).

Wie auch beim 4-Position-Model in die erste Position die Nord-Ost-Ecke und von dort aus wird parallel zur y-Achse die Position verändert bis die Süd-Ost-Ecke erreicht wird. Da angekommen wird dann parallel zur x-Achse in Richtung der Süd-West-Ecke verschoben, usw.

Das Verschieben erfolgt hier offensichtlich im Uhrzeigersinn, um dem Trend der vorherigen beiden Verfahren entgegenzuwirken.

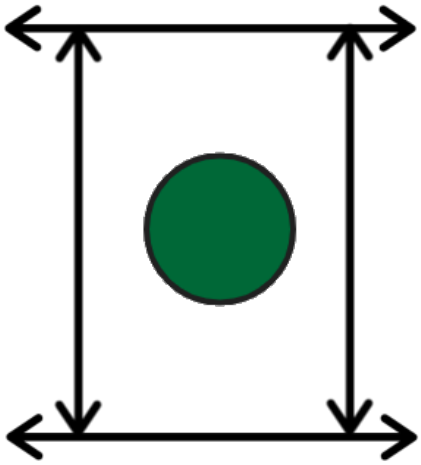

Abbildung 3: Labelpositionen im Slider-Model von ausgehenden vom dazugehörigen Knoten

2.1.4. Spiral-Model

Dieses Verfahren funktioniert gänzlich anders bisher genannten, da diese versuchen, Labelpositionen *kartesisch* zu finden, also durch Verschiebungen bezüglich der x, y-Koordinaten ausgehend vom Mittelpunkt des entsprechenden Knotens. Das Spiral-Model versucht, Labelpositionen *polar* zu finden:

$$s(m) = \begin{pmatrix} d \cdot \cos(2\pi\sqrt{\frac{m}{m_{max}}} \cdot c) \\ \sin(2\pi\sqrt{\frac{m}{m_{max}}} \cdot c) \end{pmatrix} \cdot \sqrt{\frac{m}{m_{max}}} \cdot r, \quad m \in \{1, \ldots, m_{max}\}$$

Das heißt, potentielle Labelposition werden hier spiralförmig generiert, also in ansteigendem Abstand vom dazugehörigen Knoten und in fairer Verteilung hinsichtlich der Richtung der Labels, da die Spirale auch den Winkel fortwährend inkrementiert. Auch ist die Idee wieder, den Trend von zuvor zu durchbrechen und auf eine unkonventionellere Art Labelposition zu suchen in Regionen, die zuvor noch nicht beachtet wurden.

Die Parameter der Gleichung von Luboschik et al.[3] beeinflussen die Form und Ausrichtung der Spirale:

- d: Orientierung der Spirale (im/gegen den Uhrzeigersinn), ($d \in \{-1, 1\}$)
- c: Krümmung bzw. Anzahl der Rotation der Spirale, ($c \in \mathbb{N}$)

- m_{max}: Maximalanzahl der Punkte in der Spirale, ($m_{max} \in \mathbb{N}$)
- m: Der jeweilige m-te Punkt der Spirale

Ergänzend zu den vorherigen Verfahren ist es sinnvoll, auch das Spiral-Model zu nutzen, da hier schnell Abstand vom dazugehörigen Knoten gewonnen werden kann. Dies ist wichtig, da die Verfahren zuvor vorrangig in der Nähe des Knotens versuchen, eine Position zu finden. Das Spiral-Model wird aber nur genutzt, wenn die Verfahren dabei erfolglos geblieben sind.

Wie auch beim 8-Position-Model wird, ob des erweiterten Abstands zum korrespondieren Knoten, eine Hilfslinie genutzt, um die Verbindung für gefundene Labelpositionen und ihre Knoten deutlich zu machen.

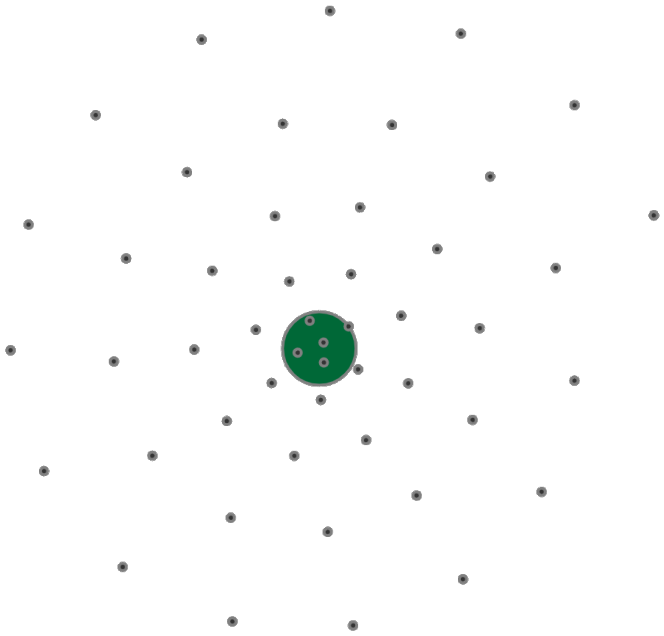

Abbildung 4: Labelpositionen im Spiral-Model von ausgehenden vom dazugehörigen Knoten. Parameter im Beispiel: $m_{max} = 50, d = 1, c = 6, r = 500$

2.2. Kantenlabeling

Im Gegensatz zum Labeling der Knoten werden beim Kantenlabeling einige Einsparungen vorgenommen. Der Grund dafür liegt zum ersten in der Natur von Graphen selbst, da diese für n Knoten bis zu n^2 Kanten haben können. Bei einem großen und dichten Graphen wären das schlichtweg zu viele Labelpositionen, wenn man so aufwendige Verfahren wie bei den Knoten verwendet.

Ein weiterer Grund ist speziell für `iGraph.js`: Die Daten, die für Kanten zur Verfügung stehen, sind in der Regel nur Integer-Werte. Da diese Werte ohne Kontext relativ nichtssagend sind, werden Kanten stattdessen mit den Namen

der durch sie verbundenen Knoten benannt. Somit gibt die Kante Auskunft über den Nachbar, selbst wenn dieser selbst nicht zu sehen ist, weil er z. B. außerhalb des Bildschirms ist oder keine Labelposition für diesen Knoten gefunden werden konnte.

Bei den Kanten wird auf eine Vorsortierung, wie es bei Knoten, verzichtet, vor allem aus Effizienzgründen, aber auch aus fehlender Notwendigkeit, da Kanten tendenziell ergänzend zum Knotenlabeling sind.

Kanten werden mit einer kleineren Schriftgröße gelabelt und erst, wenn das Knotenlabeling abgeschlossen ist und dann noch freie Kapazität vorhanden ist, das heißt, die festgelegte Maximalanzahl an gezeigten Labels noch nicht überschritten wurden (siehe Konfiguration) und der Zoom Mode nicht aktiv ist.

Das Kantenlabeling verzichtet zur Vereinfachung der Conflict Detection und Bounds) auf den gängigen Standard der Kartografie, die Labels parallel zu der Kante auszurichten. Aus den oben genannten Gründen ist das Kantenlabeling aber sowieso nebensächlich, weswegen diese Entscheidung für die Performance und gegen die Ästhetik in diesem Fall gerechtfertigt ist.

2.2.1. Midpoint

Der Startpunkt für eine potenzielle Labelposition auf Kanten ist grundsätzlich der Mittelpunkt der Kante, das heißt, das Label befindet sich in gleichmäßigem Abstand zu seinen Knoten und der Mittelpunkt des Labels liegt auf dem der Kante. Die Ausrichtung ist, wie auch bei den Knoten, parallel zur x- bzw. y-Achse.

2.2.2. Interpolation

Weitere potenzielle Labelpositionen werden entlang der Kante interpoliert bzw. das Label wird entlang der Kante in gleichmäßigem Abstand verschoben. Ob dieses Verfahren verwendet werden soll und wenn ja, wie viele Positionen auf der Kante interpoliert werden sollen, sind als Magic Constants in `consts.js` definiert. Falls der Mittelpunkt der Kante selbst wieder ein Ergebnis der Interpolation ist, wird er übersprungen, da er zuvor bereits erfolglos als Labelposition getestet wurde und nur dann die Interpolation überhaupt angewandt wird.

3. Conflict Detection und Bounds

Bevor Labels generiert werden, sollte sichergestellt sein, dass die Knoten und Kanten überhaupt sichtbar sind, um unnötige Berechnung zu ersparen. Wenn ein potentielles Label zu einem tatsächlichen Label wird, sollte sichergestellt sein, dass es sich nicht mit anderen tatsächlichen Labels oder Knoten überdeckt. Um diese Probleme zu lösen, werden Bounding Boxes genutzt:

3.1. Bounding Boxes

Eine Bounding Box ist ein Rechteck um die minimalen und maximalen Ausdehnungen eines Objekts in den x, y-Koordinaten, das somit Grenzen angibt, in dem das Objekt auf jeden Fall vollständig liegt.

Bounding Boxes wurden genutzt um die Bounds, also die Bildschirmgrenze, zu implementieren. Weiterhin fanden zu Anwendung als Repräsentation der Knoten und der Labels, sodass mit simplen Vergleichen eine Überdeckung ausgeschlossen werden kann. Eine weitere Nutzung findet sich im Zoom Mode zum Ersparen vieler unnötiger Berechnungen.

3.1.1. Bounds Checking

Die Bounds werden hier durch eine große Bounding Box repräsentiert. Für jeden Knoten und für jede Kante muss in jeder Iteration des Render-Loops kontrollieren werden, ob sie komplett sichtbar sind, das heißt, sich komplett inbounds befinden. Außerdem muss dies für generierte Labels bzw. ihre Positionen kontrolliert werden. Das ist dann der Fall, wenn für das Objekt folgendes gilt:

Sei a das Objekt und b die Bounds. a, b sind Bounding Boxes:

$$in = a_{xmin} > b_{xmin} \wedge a_{ymin} > b_{ymin} \wedge a_{xmax} < b_{xmax} \wedge a_{ymax} < b_{ymax}$$

Für die Knoten bleibt die Sortierung nach der Größe der Radien auch nach dem Filtern via Bounds Check erhalten, sodass der wichtigste, derzeit sichtbare Knoten als erstes versucht wird zu labeln. Um aus dem Versuch, als potenziellen Labelpositionen, tatsächliche Labelpositionen zu machen, muss überprüft werden, dass diese sich nicht mit Knoten schneiden (oder später mit schon gesetzten, also tatsächlichen, Labelpositionen).

3.1.2. Label Conflicts

Da ein Greedy-Ansatz genutzt wurde, wird, sobald eine potenzielle Labelposition konfliktfrei (cf) ist, diese auch sofort genutzt - auf die Gefahr hin, dass dies Labelpositionen von anderen Objekten blockiert und gegebenenfalls ein besseres Labeling nicht erreicht wird.

Für eine generierte Position, wie in Labeling beschrieben, wird nun zur Konfliktermittlung eine Bounding Box erzeugt, also das Label ëingerahmt". Für diese Bounding Box wird jetzt kontrolliert, ob sie sich mit keiner Bounding Box von allen sichtbaren Knoten und keiner Bounding Box von gegebenenfalls schon erzeugten Labels überdeckt. Ähnlich wie von Luboschik et al.[3] beschrieben: Für alle Vergleich muss gelten: Die Bounding Box des potenziellen Labels ist total links von, rechts von, über oder unter der liegen mit der sie verglichen wird (also des Knotens oder tatsächlichen Labels):

Sei a die potenzielle Bounding Box und B die Menge der Bounding Boxes von Knoten und tatsächlichen Labels:

$$cf \leftrightarrow \forall b \in B : a_{xmax} < b_{xmin} \vee a_{xmin} > b_{xmax} \vee a_{ymax} < b_{ymin} \vee a_{ymin} > b_{ymax}$$

So können sukzessive potenzielle Labelpositionen überprüft und, falls konfliktfrei, zu tatsächlichen Labels umgewandelt werden, die dann im weiteren Verlauf selbst Berücksichtigung bei der Konfliktermittlung finden.

3.2. Quadtree

Da sich während des Prozesses der Labelgeneration und Konfliktermittlung die Positionen der Knoten nicht verändern, wäre es sehr ineffizient, immer wieder alle Knoten (und deren Labels, sofern generiert) abzufragen.

Da die Aufteilung der Knoten im Raum gleich bleibt (innerhalb einer Iteration des Render-Loops), ist es sinnvoller, diese einmalig zu Beginn in einem Quadtree zu sortieren bzw. zu strukturieren.

Im Vergleich zu der naiven Methode mit einer Komplexität von $O(n^2)$ (da n-mal Knoten mit $n-1$ anderen Knoten verglichen werden) befindet man sich mit dem Quadtree in der Komplexitätsklasse $O(n \log n)$[1] [2].

Das Vorgehen des Quadtrees ist simpel: Initial startet man mit dem gesamten Bildschirm und allen Knoten. Nun ermittelt man den Mittelwert der x-Werte der Knoten und dasselbe analog für die y-Werte. Diese beiden Mittelwerte teilen nun als Geraden den Bildschirm in vier nicht zwangsläufig gleich große Teilbereiche.

Diese Teilbereiche haben annähernd die gleiche Anzahl an Knoten, im Idealfall $\frac{n}{4}$ der Knoten des Gesamtbereiches. Nun wird für diese Teilbereiche rekursiv das gleiche Verfahren angewandt wie eben für den gesamten Bildschirm beschrieben. Rekursionsabbruch ist, wenn die Teilbereiche weniger als zwei Knoten enthalten oder die maximale Rekursionstiefe erreicht ist (siehe Magic Constants).

Ein Sonderfall stellen Knoten dar, durch die die Trenngerade der Bereiche verläuft (da es sich hier nicht um Punkte handelt). Diese müssen dann in beiden Teilbereichen, auf jeder Seite der Trenngerade, als enthalten gezählt werden.

Das ist eine Schwachstelle des Quadtree in Situation, in denen die Knoten sehr stark aufeinander sind, z. B. bei starkem Zoom Out oder wenn die Lens Funktion von `iGraph.js` verwendet wird.

Da die Knoten dann fast identische Positionen haben, gelingt keine Aufteilung der Knoten in Teilregionen und alle Knoten sind in nahezu allen Regionen. Das führt den logarithmischen Charakter des Baums ad absurdum.

Der Quadtree erkennt selbst anhand eines Grenzwertes (Magic Constants), ob er ineffizient ist. Eine Lösung dafür wird ist ein Zoom Mode, der nachfolgend erklärt wird. Sollte auch das nicht möglich sein, werden die Knoten in den ineffizienten Teilregionen (also die den Grenzwerte von x Elementen pro Teilregion überschreiten) selbst als ineffizient markiert und werden beim Labeling übersprungen (obwohl sie sichtbar sind). Die Chance, die freie Kapazität bis zur Maximalanzahl an gezeigten Labels zu nutzen, wird dann anderen, weniger relevanten (laut Vorsortierung), dafür effizienten Knoten gelassen.

In den meisten Fällen ist der Quadtree aber effizient und das Ergebnis sind viele kleine, unabhängige Teilbereiche, in denen nur sehr wenige Knoten enthalten sind. Wenn nun ein potenzielles Label überprüft werden soll, so wird geguckt, welche Teilregionen des Quadtrees es überdeckt.

[1] Voraussetzung: Der Baum ist gut balanciert

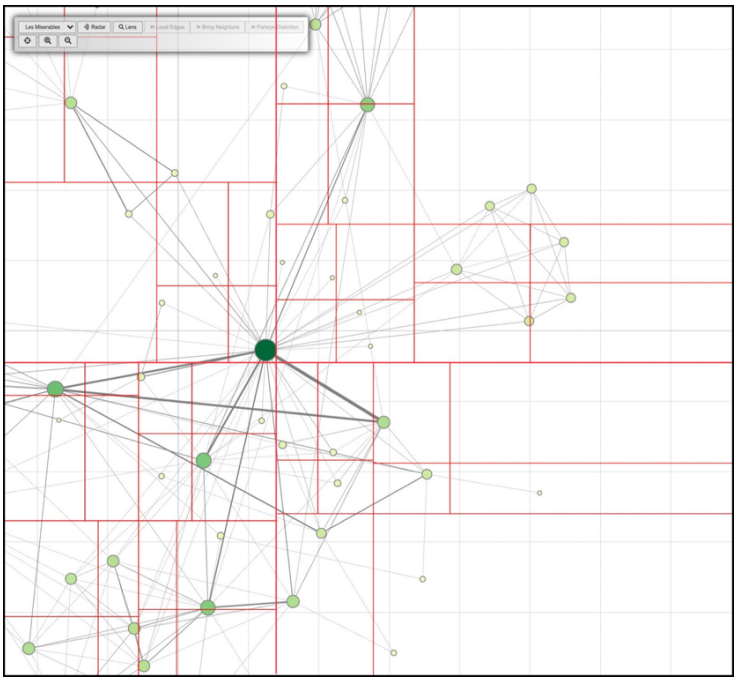

Abbildung 5: Aufteilung eines Quadtrees in seine Teilbereiche

Mit allen Knoten, die in diesen überdeckten Regionen enthalten sind, wird dann eine individuelle Konfliktermittlung durchgeführt. Das heißt, es werden nur Konflikte in der Nähe der potenziellen Labelposition direkt kontrolliert und der Rest wurde eliminiert durch den rekursiven Abstieg (mit Zugriffszeit $O(\log n)$) im Quadtree mit Grenzen, also der Bounding Box, des Labels.

Bleibt die potenzielle Labelposition (nach der Beschreibung aus Label Conflicts mit allen Elementen der überdecken Teilbereiche) konfliktfrei, so kann es zu einem tatsächlichen Label gemacht werden. Dieses wird als neues Element in die eben überprüften Teilregionen übernommen, sodass es zukünftig auch im Quadtree repräsentiert ist und bei der Konfliktermittlung berücksichtigt wird.

4. Adaptation/Optimierung

Mit dem Quadtree wurde schon eine erste Form der Optimierung in diesem Projekt vorgestellt. Diese ist allerdings sehr generisch und in der Computergrafik vielseitig eingesetzt.

Die Idee, aufwendige Berechnungen vorher zu erkennen und einzusparen bleibt auch bei nächsten Optimierungen, jedoch sind diese spezieller auf das Projekt bzw. Graph Labeling zugeschnitten und es geht hier um die Balance zwischen visuell ästhetischem und trotzdem performantem Labeling, also einen Trade-Off zwischen diesen beiden Faktoren. Ein Quadtree ist immer besser als naive, da besteht kein Trade-Off. Insofern sind die

Tendenziell gilt: Je mehr Element potentiell zu labeln sind und je dichter diese Elemente beieinander sind, desto aufwendiger wird das Labeling und sollte zu Gunsten der Performance vereinfacht werden.

4.1. Zoom Mode

Wenn die Elemente zu dicht beieinander oder sogar aufeinander bzw.überdeckend sind, kann es sind, dass der Quadtree ineffizient wird. Dies ist vor allem der Fall, wenn man weit rauszoomt. Dann ist der Graph quasi auf einem Punkt und um ihn herum ist viel ungenutzter Raum.

Ob der Zoom Mode nutzbar ist, wird kontrolliert, wenn der Quadtree sich als ineffizient deklariert. Hierfür werden von allen Knoten minimalen und maximalen x- und y Werte ermittelt und aus diesen eine große Bounding Box generiert, die dann den Graphen vollständig enthält.

Diese Bounding Box des Graphen wird mit den Bounds verglichen und es wird überprüft, ob oberhalb oder unterhalb der Graph-Bounding Box noch wenigstens eine Labelhöhe Platz ist oder ob sich links oder rechts davon noch wenigstens eine Labelbreite ungenutzter Raum befindet.

Findet sich an wenigstens einer Seite solch ungenutzter Raum, dann wird mittels Spiral Model versucht, eine Position in diesem Raum zu finden. Wird eine solche Position gefunden, ist sie in-bounds und hat keine Überdeckung mit der Bounding-Box des Graphen.

Spiral Model wird verwendet, da dies der einzige Algorithmus ist, der sich inkrementweise von seinem Ursprung (also dem korrespondieren Knoten) entfernt und man mit der Position des Labels außerhalb der großen Bounding Box des Graphen landen muss. Die wäre mit den anderen Algorithmen nicht zuverlässig möglich. Wie bei Spiral Model üblich werden die tatsächlichen Labels mit einer Hilfslinie zum Knoten verbunden.

Tatsächliche Labels werden hier nicht in den Quadtree einsortiert, da er im Zoom Mode nicht genutzt wird. Stattdessen werden sie sich direkt gemerkt. Der Performancegewinn hier ist, dass alle Knoten als eine Bounding Box dargestellt werden, und man somit im Prinzip nur Labelpositionen miteinander, aber nicht mit Knoten vergleichen muss.

In der Praxis ist die Anzahl der gelabelten Knoten konstant (siehe Magic Constants) und nur ein Bruchteil der zu sehenden Knoten, sodass die quadratische Komplexität hier noch nicht zu Performanceeinbußen führt.

Kantenlabeling findet im Zoom-Mode nicht statt, da die Kanten praktisch von den Knoten im Zoom überdeckt werden und hier auch Berechnungen eingespart werden können.

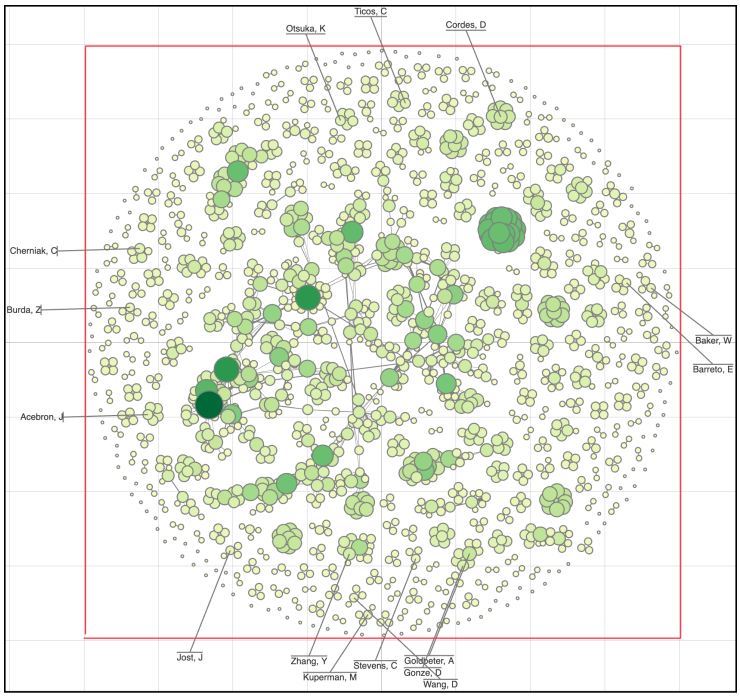

Abbildung 6: Zoom Mode aktiv. Außerdem zu sehen: Hilfslinie zwischen Knoten und Labels

4.2. Performance Modes

Performance Modes bilden ein Zusammenspiel aus der Anzahl der sichtbaren Knoten, also derjenigen, die in-bounds sind, und den genutzten Labeling Verfahren. Allgemein gesagt: Je mehr Knoten sichtbar sind, desto weniger Verfahren werden genutzt - und die genutzten Verfahren sind die einfacheren Verfahren.

Dies wird durch Grenzwerte realisiert (siehe Magic Constants). Die Grenzwerte (siehe Figure 7) wurden vor allem durch manuelles Testen ermittelt. Ziel war es das eine Iteration des Render-Loops möglichst unter 50 ms bleibt.

Minimum	Maximum	Verfahren
1	99	alle Verfahren
100	499	4-Position-Model, 8-Position-Model, Slider-Model
500	999	4-Position-Model, 8-Position-Model
500	∞	4-Position-Model

Abbildung 7: Intervalle wie standardmäßig im Projekt festgelegt (siehe Magic Constants) mit dazugehörigen Labeling Verfahren

Hier ist der Zweck, dass man bei sehr vielen sichtbaren Knoten durch das Labeling eine schnelle, grobe Orientierung bekommen kann. Danach kann man

via Zoom-In den gewünschten Bereich genauer explorieren, wofür dann auch ein aufwendiges und somit detailliertes Labeling wünschenswert wäre. Da dann auch weniger Knoten zu sehen sind, was zu weniger Vergleichen führt, ist hier genug Zeit für solche komplizierten Berechnung.

5. Visualisierung

Wenn ein potentielles Label nach Feststellen von Konfliktfreiheit zu einem tatsächlichen Label wird, so ist es für das Backend (wie in Conflict Detection und Bounds beschrieben) relevant, dieses Label von nun an in der Konfliktermittlung zu berücksichtigen. Für das Frontend gilt es dann, das Label an der gefundenen, freien Position zu zeichnen und animieren, sodass es für den Nutzer angenehmer anzuschauen ist.

5.1. Drawing

Wenn einem für eine Labelposition zugesichert werden kann, dass diese konfliktfrei ist, dann wird der Text bzw. Name des Labels ausgelesen und an eben diese Position in dem 2D-Canvas zu dem Graphen und ggf. anderen Labels hinzugefügt.

Ein Sonderfall stellen Label dar, die mit 8-Position-Model oder Spiral-Model generiert wurden, da hier eine Hilfslinie zur Kenntlichmachung der Zusammengehörigkeit von Knoten und Label zusätzlich gezeichnet wird.

Diese Labels enthalten daher weitere Informationen über die Linie und die Seite des Labels, zu der diese Linie verbindet. Die Seite ist immer entgegengesetzt zur Richtung, in der das Label gefunden wurde, also wenn das Label unterhalb des Knoten (Süd-Position) ist, verbindet die Hilfslinie Knoten und Oberseite des Labels. Die Hilfslinie endet immer in der Mitte der Seite. Nur die Seite, mit der die Linie verbindet, wird dann zur weiteren Verdeutlichung selbst auch als Linie gezeichnet.

5.2. Animation

Die Animation finden immer dann statt, wenn sich der Sichtbarkeitszustand von Labels im Vergleich zur vorherigen Iteration des Render Loops ändert oder es nach wie vor sichtbar ist, aber die Position des Labels sich geändert hat. Das passiert, wenn in der neuen Iteration für ein anderes, wichtigeres Label diese Position gefunden wurde, dann im Nachhinein aber noch für das unwichtigere Label eine andere Position gefunden werden kann. Wäre dies nicht der Fall, würde sich der Sichtbarkeitszustand zu nicht sichtbar ändern und das unwichtigere Label würde ausgeblendet werden (siehe Figure 8).

Hierfür wird die Klasse aus Animatable.js aus iGraph.js verwendet, die dort auch schon die Animation der Knoten händelt. Somit werden Knoten und Labels durch dieselben Funktionen animiert.

	sichtbar_t	nicht sichtbar_t
sichtbar_{t-1}	animate	fade out
$\text{nicht sichtbar}_{t-1}$	fade in	–

Abbildung 8: Zustände von Labelsichtbarkeit vor $(t-1)$ und nach (t) dem Update der Positionen und die entsprechenden Animationsfunktionen

5.2.1. Fade In / Fade Out

Diese Funktion verändern lediglich die Alpha-Werte, also die Transparenz, der Labels und ggf. der dazugehörigen Linien. Für das Ausblenden werden sukzessive die Alpha-Werte dekrementiert, zum Einblenden werden sie sukzessive erhöht.

5.2.2. Animate

Da hier vor und nach dem Update das Label zu sehen ist, muss es verschoben werden. Dabei kann es passieren, dass es kurzzeitig zu Überdeckung des von Labels und Knoten kommt. In der finalen Position, nach der Animation, allerdings nicht mehr, da diese Position konfliktfrei ist.

Die Bewegungen bzw. die Animationen sind nicht rein linear, was das Labeling flüssiger, interaktiver und "lebendigerërscheinen lässt, auch wenn es zusätzliche Berechnungen erfordert. Diese können angestellt werden, da zuvor auf Effizienz geachtet wurde.

6. Konfiguration

Die Konfiguration des Labeling kann über zwei Wege erfolgen:

Der erste ist der User Mode und erfolgt über ein minimales GUI, das bereits durch `iGraph.js` vorhanden war. Neue Funktionalität wurden einfach in zusätzliche Buttons implementiert.

Der zweite Weg ist eine Art Author Mode. Es existiert ein File `consts.js`, in denen alle im Projekt genutzten Magic Constants in einem Objekt zentral definiert sind und somit auch dort abgeändert werden könnten. Die Standardeinstellungen sind allerdings getestet, auf `iGraph.js` zugeschnitten und haben sich über Monate etabliert. Bei Änderung könnten ungewollte Konsequenzen und Bugs auftreten.

6.1. GUI

Im Gegensatz zu Änderungen der Magic Constants ist das GUI sicher. Es beachtet festgelegte Minimal- und Maximalwerte und sorgt dafür, dass diese nicht unterschritten bzw. überschritten werden (sofern die Standardwerte in `consts.js` verwendet werden).

Die durch das GUI zu ändernden Parameter werden durch Buttons realisiert (siehe Figure 9) und lauten:

- Label anzeigen (Toggle)
- Anzahl angezeigte Labels erhöhen
- Anzahl angezeigte Labels reduzieren
- Schriftgröße erhöhen
- Schriftgröße reduzieren

Weiterhin gilt zu sagen, dass für die Anzahl der gezeigten Labels im Zoom Mode weniger Labels angezeigt werden als sonst. Um hier filigraner einstellen zu können, erfolgt das De-/Inkrementieren der Anzahl der gezeigten Labels in Einserschritten. Für den normalen Modus werden mehrere Labels auf einmal hinzugefügt bzw. gelöscht.

Um Berechnungen für große Graphen zu reduzieren sind geringe Labelanzahl und Schriftgröße eines der besten Werkzeuge, sollten allerdings vom User eingestellt werden können - daher ein GUI.

Abbildung 9: Simples GUI. Zweite Zeile ist verantwortlich für Labeling Einstellungen

6.2. Magic Constants

Ursprünglich zum vereinfachten internen Testen und zur Strukturierung des Codes gedacht, ist eine Sammlung der Magic Constants gegebenenfalls auch für externe Nutzer interessant. Viele Parameter, die über das GUI nicht verstellt werden können, sind hier konfigurierbar, beispielweise:

- Schriftart und -farbe
- Interpolationsschritt für Kanten (siehe Kantenlabeling)
- Ineffizienzgrenzen der Teilregionen des Quadtree (siehe Quadtree).
- Parameter der Spirale (siehe Spiral-Model)
- Grenzwerte für die Performance Modes (siehe Figure 7)

Änderung an den Werten können immer dazu führen, dass das Programm gar nicht funktioniert oder zumindest nicht wie erwartet. Für die Nutzung von gebräuchlichen Werten sollten allerdings maximal die Performance schlechter werden, Funktionalität aber erhalten bleiben.

Möglicherweise gibt es auch noch bessere Einstellungen hinsichtlich der Performance, die visuell gleichermaßen ansprechend sind wie die Standardeinstellungen.

7. Über das Projekt

7.1. Technische Details und Umsetzung

Das gesamte Projekt wurde in JavaScript umgesetzt, da die Basis `iGraph.js`, schon in JavaScript geschrieben war. Es ist maximal modular zur Basis, das heißt, bis auf eine zusätzliche Zeile im Update-Loop (und die GUI-Elemente) wurden am Quellcode der Basis keine Veränderungen vorgenommen.

Der Code Style ist ES6-konform. Es wurden Klassen genutzt, um die einzelnen, logischen Bereiche auch im Code entsprechend zu trennen. Manche Klassen, wie z. B. `BBox.js`, sind eher Namespaces für Funktionen mit entsprechendem Zuständigkeitsbereich. Daher sind diese Funktionen dann statisch.

Ausnutzung von sprachlichen Besonderheiten oder Datenstrukturen wurden in folgenden Bereichen vorgenommen:

7.1.1. Maps

Die Knoten und Kanten werden in Maps gespeichert. Man kann sie eindeutig durch eine ID zuordnen (die von `iGraph.js` selbst kommt, im Falle von Kanten) oder einfach durch den Namen des Labels (im Falle von Knoten). So erhält man eine schnelle Zugriffszeit von $O(1)$ für das Updaten der Werte. Bei den Knoten wird in der Map die Sortierung erhalten, sodass die Map im weiteren Verlauf einfach zum Array konvertiert werden kann.

7.1.2. Generator Functions

Generator Functions werden genutzt, um den Greedy Approach des Projektes zu realisieren. Die Elemente werden einzeln "yieldedünd weiterverarbeitet. Zum Beispiel kann so eine Funktion für alle potentiellen Labelpositionen angegeben werden, wenn sie aber als Generator Function gekennzeichnet ist, kann man Finden einer tatsächlicher Labelposition einfach returnen und die restlichen Positionen werden nicht berechnet.

7.2. Async Functions

Wo möglich werden asynchrone Funktionen benutzt. Zum Beispiel ist das Zeichen und Animieren der Labels unabhängigen vorherigen Schritt und kann somit asynchron erledigt werden, sobald die Labelposition konfliktfrei ist.

7.3. Bekannte Bugs

Beim Testen wurden die meisten Bugs behoben - ein selten auftretender Bug ist übrig geblieben: "Flying Labels".

Manchmal kann es vorkommen, dass beim Fade Out von Labels diese von ihrer Position "wegfliegenïn eine Ecke des Bildschirms. Dies fällt nur auf, wenn enorm viele Berechnungen stattfinden, also die Animationen langsamer sind.

Eine weitere ungünstige Konsequenz, die aus dem Fokus auf die Performance entstand: Wenn alle sichtbaren Knoten als ineffizient markiert werden (und somit nicht gelabelt werden) und kein Zoom Mode möglich ist, z. B. wenn man Radar und Lens Funktion von `iGraph.js` nutzt, dann wird gar nichts gelabelt.

Diese Situation sind aber praktisch unmöglich, man muss sie sehr gezielt erzwingen.

7.4. Verbesserungsvorschläge und Anregungen

Die Grundlage des Labeling wurde durch dieses Projekt erstellt. Auf der Seite der Performance wurden viele Techniken umgesetzt und Möglichkeit genutzt, um unnötige Berechnungen zu ersparen. Hinsicht der Visualisierung können aber vor allem noch Verbesserungen vorgenommen werden:

7.4.1. Verbessertes Kantenlabeling

Die Kantenlabels sollten ebenfalls nach kartografischen Standard gezeichnet werden.[1] Das heißt, parallel zur entsprechenden Kante. Dafür wären zum einen verbesserte Animationen nötig, um die Labels zu rotieren. Zum anderen eine verbesserte Konfliktermittlung, da dann nicht mehr garantiert werden kann, dass die Kantenlabels parallel zur x- und y-Achse sind. Möglicherweise wäre der Ansatz von Schwartges et al.[7] hierfür ein sinnvolles Werkzeug - und auch darüber hinaus für Kanten die keine Linie sind. Dies lässt sich dann gegebenenfalls mit dem nächsten Vorschlag verbinden.

Die Kanten werden generell sehr vernachlässigt behandelt, was aber auch der Natur der Daten geschuldet ist. Für aufwendiges Design und zusätzliche visuelle Kodierungen der Kanten sind viele Verbesserungsvorschläge durch Romat et al.[5] gegeben. Das Kantenlabeling muss hier eine Balance finden und die zusätzlichen Informationen ergänzen, darf sich aber nicht aufdrängen. Vielleicht ist es auch generell sinnvoller, vorrangig Knoten zu labeln und die Bedeutung der Kanten anders visuell zu enkodieren für ein intuitives Verständnis.

7.4.2. Verbesserte Repräsentation der Knoten als Kreis

Das würde die Labelqualität verbessern, da die Labels noch näher an den Knoten wären und dementsprechend auch weniger Platz an jedem Knoten verschwenden (die Ecken der Bounding Box, die nicht Teil des Knotens sind).

Auch hier wäre eine verbesserte Konfliktermittlung, da die Knoten dann als Kreis statt als Bounding Box repräsentiert würden. Zusammen mit dem ersten Verbesserungsvorschlag müsste man dann nicht Rechtecke parallel zur x- und y-Achse, sondern stattdessen beliebig orientierte Rechtecke und Kreise vergleichen, was wesentlich aufwendiger ist.

7.4.3. Konservativere Animationen

Ein wichtiger Teil ist durch Target Separation nach Ghani et al. [6] bereits getan. Die Struktur des Graphen und die Konfliktfreiheit der Labels ergeben dies. Zuweilen sind die Animationen allerdings sehr aktiv, was zu viel Unruhe im Labeling führt. Gegebenenfalls sollte man eine Toleranzzeit einführen, sodass sich Labels kurzzeitig überlappen dürfen, da viele Animationen durch minimale Überdeckungen während der Bewegungen entstehen. Eine solche Toleranzzeit würde das Gesamtbild etwas statischer machen und einfach bei den Magic Constants hinzugefügt werden.

Danksagung

Ich möchte meinem Projektpartner Nico Trebbin für die Zusammenarbeit danken. Weiterhin danke ich Prof. Christian Tominski für die Möglichkeit, auf seiner Codebasis aufzubauen und für Anregungen während der Meetings.

Literatur

[1] URL https://www.e-education.psu.edu/geog486/node/557.

[2] Russell A. Brown. Building a balanced k-d tree in o(kn log n) time. *ArXiv*, abs/1410.5420, 2014.

[3] Martin Luboschik, Heidrun Schumann, and Hilko Cords. Particle-based labeling: Fast point-feature labeling without obscuring other visual features. *IEEE Transactions on Visualization and Computer Graphics*, 14 (6):1237–1244, 2008. ISSN 1077-2626. doi:10.1109/tvcg.2008.152. URL https://doi.org/10.1109/TVCG.2008.152.

[4] Dale A. Schoenefeld Oleg V. Verner, Roger L. Wainwright. Placing text labels on maps and diagrams using genetic algorithms with masking. 1997. doi:10.1287/ijoc.9.3.266.

[5] Hugo Romat, Caroline Appert, Benjamin Bach, Nathalie Henry-Riche, and Emmanuel Pietriga. Animated edge textures in node-link diagrams: A design space and initial evaluation. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, page 1–13, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356206. doi:10.1145/3173574.3173761. URL https://doi.org/10.1145/3173574.3173761.

[6] J. S. Yi S. Ghani, N. Elmqvist. Perception of animated node-link diagrams for dynamic graphs. 2012. URL https://doi.org/10.1111/j.1467-8659.2012.03113.x.

[7] Nadine Schwartges, Alexander Wolff, and Jan-Henrik Haunert. Labeling streets in interactive maps using embedded labels. In *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '14, page 517–520, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450331319. doi:10.1145/2666310.2666494. URL https://doi.org/10.1145/2666310.2666494.

[8] C. Tominski, J. Abello, F. van Ham, and H. Schumann. Fisheye tree views and lenses for graph visualization. In *Tenth International Conference on Information Visualisation (IV'06)*, pages 17–24, 2006. doi:10.1109/IV.2006.54.

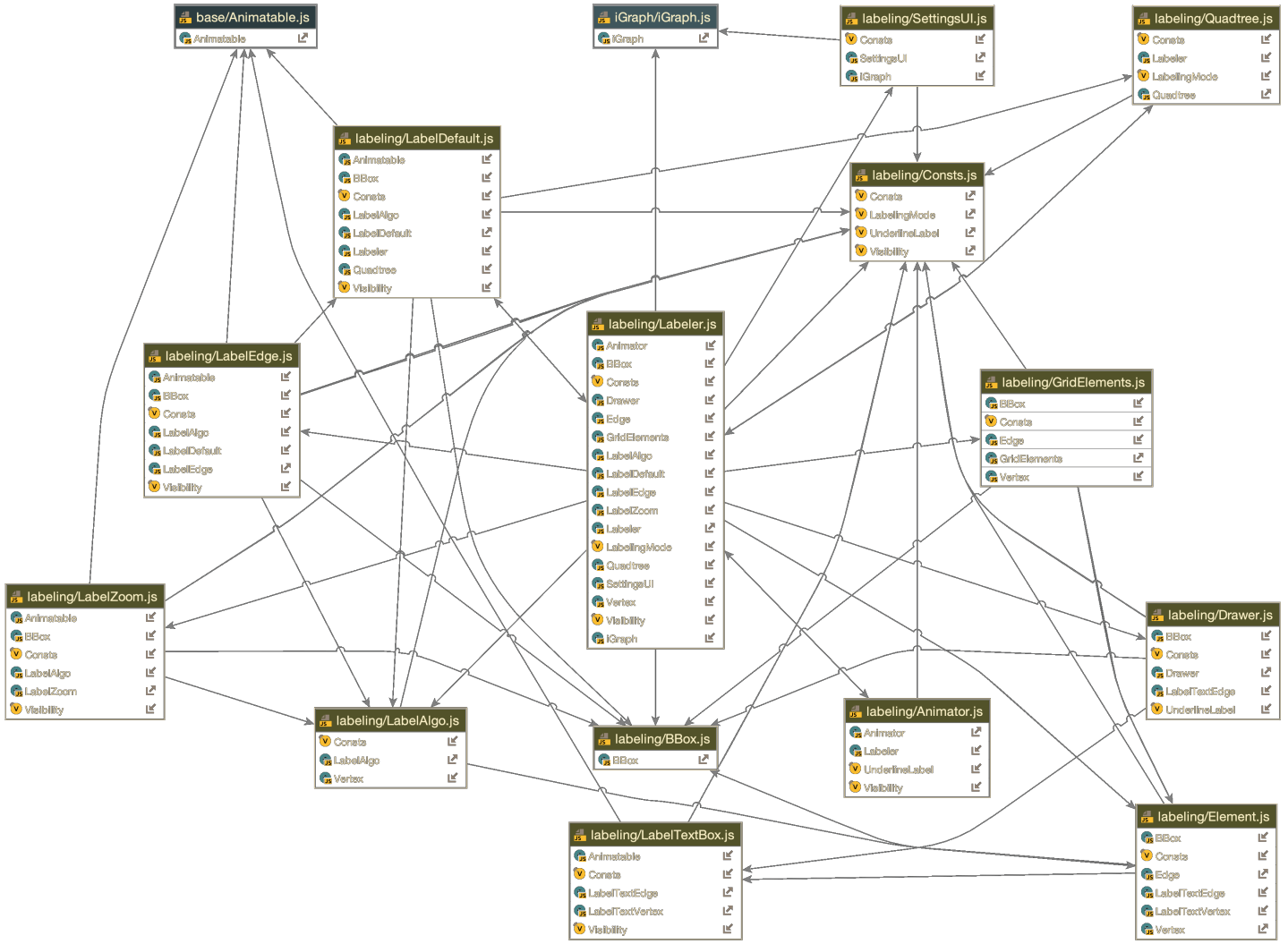

Abbildung 10: Class Diagram

Class: Animatable

Animatable()

new Animatable()

FILE OF iGraph. WE DID NOT WRITE THIS FILE.

Used Animator and LabelTextBox. From iGraph original.

Source: base/Animatable.js, line 9

Classes

Animatable
FILE OF iGraph. WE DID NOT WRITE THIS FILE.

Class: Animator

(abstract) Animator()

(abstract) new Animator()

Rather a namespace than a class for all functions responsible for tasks regarding transparency/opaqueness like fading in and out labels and its corresponding "visual-aid" lines.

Source: labeling/Animator.js, line 19

Methods

animate(time, label_line, labeler) → {boolean}

Animates the current label. Position or opacity of the label will be modified during this process.

Parameters:

Name	Type	Description
time	number	The current timestamp of the animation frame.
label_line	Array.<(Element|Line)>	An array of elements and lines. The lines are optional and will be created in the animation process.
labeler	Labeler	The labeler that has labeled the elements.

Source: labeling/Animator.js, line 33

Returns:

True, if the labels need further updates.

Type
boolean

(private) faded_in(label_line, time, labeler) → {boolean}

Fade the label in thus increase the label's opacity

Parameters:

Name	Type	Description
label_line	Array.<(Element|Line)>	An array of elements and lines. The lines are optional and will be created in the animation process.
time	number	The current timestamp of the animation frame.
labeler	Labeler	The labeler that has labeled the elements.

Source: labeling/Animator.js, line 142

Returns:

True, if the label needs further updates.

Type
boolean

(private) faded_out(label_line, time, labeler) → {boolean}

Fade the label out thus decrease the label's opacity.

Parameters:

Name	Type	Description
label_line	Array.<(Element|Line)>	An array of elements and lines. The lines are optional and will be created in the animation process.
time	number	The current timestamp of the animation frame.
labeler	Labeler	The labeler that has labeled the elements.

Source: labeling/Animator.js, line 116

Returns:
True, if the label needs further updates.

Type
boolean

(private) movement(label_line, time, labeler) → {boolean}

Move the label to the target position. Due to the animation-process the position will be partially increased.

Parameters:

Name	Type	Description
label_line	Array.<(Element|Line)>	An array of elements and lines. The lines are optional and will be created in the animation process.
time	number	The current timestamp of the animation frame.
labeler	Labeler	The labeler that has labeled the elements.

Source: labeling/Animator.js, line 170

Returns:
True, if the label needs further updates.

Type
boolean

(private) specify_line(label_line)

Creates a line from the label to the center of the node.

Parameters:

Name	Type	Description

Name	Type	Description
label_line	Array.<(Vertex|Line)>	An array of elements and lines. The lines are optional. Only done with Vertex in eight_pos and sample_algorithm.

Source: labeling/Animator.js, line 54

Class: BBox

(abstract) BBox()

(abstract) new BBox()

The functions used here convert to/from a Bounding Box and do required comparisons in 2D internally. It's rather a namespace for all functions used regarding basic geometry, thus they are made static, since this "class" has no behavior modifying internal state.

Source: labeling/BBox.js, line 19

Methods

canvas(width, height) → {BoundingBox}

Generates a Bounding Box matching the screen width and height for further calculations.

Parameters:

Name	Type	Description
width	number	Canvas width.
height	number	Canvas height.

Source: labeling/BBox.js, line 128

Returns:

Bounding Box representing screen's bounds.

Type
BoundingBox

center(bbox) → {Point}

Calculates the Center Point of a given Bounding Box, used for Edge Midpoints and to position Labels centered inside their corresponding Bounding Box.

Parameters:

Name	Type	Description
bbox	BoundingBox	A Bounding Box (of an Edge or a LabelTextBox).

Source: labeling/BBox.js, line 117

Returns:

Center Point of the Bounding Box (as [x,y]).

Type
Point

edge(p1, p2) → {BoundingBox}

Generates a Bounding Box for Edges, so that the Edge is a Diagonal of that Bounding Box.

Parameters:

Name	Type	Description
p1	Point	Start Point of an Edge, i.e. Center Point of a Vertex.
p2	Point	End Point of an Edge, i.e. Center Point of another Vertex.

Source: labeling/BBox.js, line 155

Returns:

Bounding Box exactly around the Edge.

Type
BoundingBox

element_border(visible_vertices) → {BoundingBox}

Generates a Bounding Box around all elements plus a small buffer around is. Only used when LabelingMode is ZOOM_LABELING.

Parameters:

Name	Type	Description
visible_vertices	Array.<Vertex>	All Vertices that a currently in bounds/visible.

Source: labeling/BBox.js, line 86

Returns:

A Bounding Box surrounding all visible Vertices.

Type
BoundingBox

has_no_conflict(bbox, prefilter) → {boolean}

Only used in slider_model because there, the individual, possible label positions have a lot of overlap themselves, resulting in lots of redundant checks inside the Quadtree. Instead, prefilter all elements, that overlap with the entire "Slider Region" (all possible positions for Slider) from the Quadtree only once before.

Parameters:

Name	Type	Description
bbox	BoundingBox	A possible Label Position to test for conflict.
prefilter	Array.<(Vertex|LabelTextBox)>	A collection of all Labels and Vertices in the corresponding "Slider Region".

Source: labeling/BBox.js, line 52

Returns:

True, if no overlap arises with any element of the prefilter.

Type

boolean

has_overlap(a, b) → {boolean}

Checks, if two given Bounding Boxes overlap.

Parameters:

Name	Type	Description
a	BoundingBox	Bounding Box of Vertex, Edge, Label or Bounds.
b	BoundingBox	Another Bounding Box of Vertex, Edge, Label or Bounds.

Source: labeling/BBox.js, line 31

Returns:

True, if given Bounding Boxes overlap.

Type
boolean

in_bounds(bounds, elem_bbox) → {boolean}

Determines, if a given element is *fully* visible and thus calculations must be done with respect to that element.

Parameters:

Name	Type	Description
bounds	BoundingBox	A specific Bounding Box matching the width and height of the screen, e.g. [0, 0, 1920, 1080].
elem_bbox	BoundingBox	Bounding Box of the element.

Source: labeling/BBox.js, line 70

Returns:

True, if the element is fully visible.

Type
boolean

vertex(x, y, r) → {BoundingBox}

Generates a Bounding Box for Vertices, centered on the Vertex' x,y-Coordinates and side length of the Vertex' diameter.

Parameters:

Name	Type	Description
x	number	x-value of a given Vertex.
y	number	y-value of a given Vertex.
r	number	Radius of a given Vertex.

Source: labeling/BBox.js, line 142

Returns:

Square Bounding Box exactly around the Vertex.

Type

BoundingBox

Class: Drawer

(abstract) Drawer()

(abstract) new Drawer()

Rather a namespace than a class, for all functions regarding drawing Labels and Lines on the canvas.

Source: labeling/Drawer.js, line 14

Methods

draw_label(label_line, labeler)

Draws the labels on the canvas.

Parameters:

Name	Type	Description
label_line	Array.<(Element|Line)>	An array of elements and lines. The lines are optional.
labeler	Labeler	The labeler that has labeled the elements.

Source: labeling/Drawer.js, line 24

(private) draw_line(label_with_lines, labeler)

Draw a line from the label to the node (if given)

Parameters:

Name	Type	Description
label_with_lines	Array.<(Element|Line)>	An array of elements and lines. The lines are optional.
labeler	Labeler	The labeler that has labeled the elements.

Source: labeling/Drawer.js, line 52

Class: Edge

Edge(**link, labeltext, labelclass**)

new Edge(link, labeltext, labelclass)

Representation of an Edge/Link in iGraph, modeling its state and behavior.

Parameters:

Name	Type	Description
link	Object	The original object from NodeLink, which gets processed when creating GridElements.
labeltext	string	The name/text of the corresponding element, that is used for labeling later.
labelclass	LabelTextBox	Specific Class (LabelTextEdge) for Edges, storing and calculating all relevant Parameter, like Font Size, Label Width and Height, etc.

Source: labeling/Element.js, line 121

Classes

Edge

Methods

(generator) interpolate() → {Point}

Interpolates additional possible label position along the edge. Since Mid-Point gets tested first always, it's ignored here. Works any amount of Interpolations >= 2 (set in Consts), but better for even numbers.

Source: labeling/Element.js, line 167

Yields:

Yields evenly spaced points along the edge.

Type
Point

update_xy(link)

Updates the Start, Mid and End Point of the Edge as well as the Bbox.

Parameters:

Name	Type	Description
link	Object	The original object from NodeLink, which gets processed when creating GridElements.

Source: labeling/Element.js, line 150

Class: Element

(abstract) Element(labeltext, labelclass)

(abstract) new Element(labeltext, labelclass)

Real Abstract Class, that inherits to the specific Grid Elements, i.e. Vertex and Edge.

Parameters:

Name	Type	Description
labeltext	string	The name/text of the corresponding element, that is used for labeling later.
labelclass	LabelTextBox	Subclasses use different classes to represent their labels, since Vertices and Edges have e.g. different Font Sizes. It is as if we pass the specific constructor as an argument.

Source: labeling/Element.js, line 14

Methods

label() → {string}

Just a "shortcut" or alias to save some screen space.

Source: labeling/Element.js, line 43

Returns:

The name of the label.

Type
string

(abstract) update_xy()

Abstract Method. Is used for both Subclasses to update the Center/Mid-Point.

Source: labeling/Element.js, line 53

Class: Graph

Graph()

new Graph()

FILE OF iGraph. WE DID NOT WRITE THIS FILE.

Used to create NodeLink and its Dots/Links. From iGraph original.

Source: base/Graph.js, line 10

Classes

Graph

FILE OF iGraph. WE DID NOT WRITE THIS FILE.

Documentation generated by JSDoc 3.6.10 on Wed Feb 02 2022 18:13:43 GMT+0000 (Coordinated Universal Time)

Class: GridElements

GridElements**(node_link)**

new GridElements(node_link)

Collects each individual Vertex and each Edge in a Map. Furthermore, keeps track of the Screen Bounds, Amount Labels and some other characteristics of the graph. Regarding representation, this is our equivalent to NodeLink.

Parameters:

Name	Type	Description
node_link	NodeLink	The original graph representation.

Source: labeling/GridElements.js, line 15

Classes

GridElements

Methods

(private) parse_edges(links)

Makes an Entry in the Edge Map for each Link and creates the corresponding Edge.

Parameters:

Name	Type	Description
links	Array.<Object>	All original "edge" objects from NodeLink.

Source: labeling/GridElements.js, line 108

(private) parse_vertices(dots)

Makes an Entry in the Vertex Map for each Dot and creates the corresponding Vertex.

Parameters:

Name	Type	Description
dots	Array.<Object>	All original "vertex" objects from NodeLink.

Source: labeling/GridElements.js, line 97

unlabel_vertices()

Sets every Vertex to "not labeled".

Source: labeling/GridElements.js, line 50

(private) update_edges(links)

Gets the corresponding Edge from the Edge Map via Index and calls its update function, handing over the individual edge as argument to be used inside that update function.

Parameters:

Name	Type	Description
links		

Source: labeling/GridElements.js, line 132

(private) update_vertices()

Gets the corresponding Vertex from the Vertex Map via Labelname and calls its update function, handing over the new x,y-Coords as argument.

Source: labeling/GridElements.js, line 121

(generator) visible_edges(node_link) → {Edge}

After updating the Edges, check, which Edges are still in bounds are return those. Yield here, since Edges are not required for Conflict Detection, and thus can be done "just in time" (opposed to Vertices).

Parameters:

Name	Type	Description
node_link	NodeLink	The original graph representation.

Source: labeling/GridElements.js, line 82

Yields:

A *visible* edges. i.e. in bounds.

Type
Edge

visible_vertices(node_link, canvas) → {Array.<Vertex>}

After updating the bounds and x,y-Coordinates of the Vertices, check, which Vertices are still in bounds are return those. Return here, since all vertices are required for Conflict Detection (opposed to Edges, were we yield).

Parameters:

Name	Type	Description
node_link	NodeLink	The original graph representation.
canvas	Object	Canvas Object from iGraph.

Source: labeling/GridElements.js, line 64

Returns:

Updated List of all *visible* vertices.

Type
Array.<Vertex>

Class: LabelAlgo

LabelAlgo()

new LabelAlgo()

LabelAlgo is rather a namespace than a class and implements different algorithms (from the paper below). These algorithms convert the given args to Bounding Boxes, that could be a valid label(-box) position, which is checked afterwards.
https://innovis.cpsc.ucalgary.ca/innovis/uploads/Courses/InformationVisualizationDetails2009/Luboschik2008.pdf

Source: labeling/LabelAlgo.js, line 12

Methods

(generator, static) eight_pos(x, y, width, height, r) → {BoundingBox}

Calculates the label Bounding Boxes using the eight-pos model.

Parameters:

Name	Type	Default	Description
x	number		x-coordinate of the node's center
y	number		y-coordinate of the node's center
width	number		the width of the label
height	number		the height of the label
r	number	0	the radius ot the node

Source: labeling/LabelAlgo.js, line 57

Yields:

the label bbox

Type
BoundingBox

(generator, static) four_pos(x, y, width, height, r) → {BoundingBox}

Calculates the label Bounding Boxes using the four-pos model.

Parameters:

Name	Type	Default	Description
x	number		x-coordinate of the node's center.
y	number		y-coordinate of the node's center.
width	number		The width of the label.
height	number		The height of the label.l
r	number	0	The radius ot the node.

Source: labeling/LabelAlgo.js, line 39

Yields:

The label bbox.

Type
BoundingBox

(static) identity(x, y, width, height, ..._) → {BoundingBox}

Calculates the label Bounding Boxes using identity.

Parameters:

Name	Type	Attributes	Description
x	number		x-coordinate of the node's center.
y	number		y-coordinate of the node's center.
width	number		The width of the label.
height	number		The height of the label.
_	any	<repeatable>	Anonymous, to match the structure of the other functions.

Source: labeling/LabelAlgo.js, line 24

Returns:

The label bbox.

Type
BoundingBox

(generator, static) slider_model(x, y, width, height, r, step) → {BoundingBox}

Calculates the label Bounding Boxes using the slider-pos model.

Parameters:

Name	Type	Default	Description
x	number		x-coordinate of the node's center.
y	number		y-coordinate of the node's center.
width	number		The width of the label.
height	number		The height of the label.l
r	number	0	The radius ot the node.
step	number		The step size.

Source: labeling/LabelAlgo.js, line 98

Yields:

The label bbox.

Type
BoundingBox

(generator, static) spiral(x, y, mmax, d, c, r, m) → {Point}

For a given args and a Point in 2D, this calculates new points spiraling away from the center.

Parameters:

Name	Type	Default	Description
x	number		x-coordinate of the node's center.
y	number		y-coordinate of the node's center.
mmax	number		Maximum Iteration Number, i.e. Amount of calculated spiral points.
d	number		"Orientation" of the spiral. -1 is clockwise, 1 counterclockwise.
c	number		Curvature of the spiral. The higher the value, the more curve.
r	number		Radius of the Spiral.
m	number	1	Starting iteration number.

Source: labeling/LabelAlgo.js, line 150

Yields:

The m-th Point of the spiral.

Type
Point

label_point_on_bbox(v, bbox) → {Point}

Calculates the vertex's corresponding point on the canvas' bbox.

Parameters:

Name	Type	Description
v	Vertex	A vertex.
bbox	BoundingBox	The canvas' bbox.

Source: labeling/LabelAlgo.js, line 164

Returns:

The corresponding point of the vertex on the bbox.

Type
Point

(private) sample_function(x, y, m, mmax, d, c, r) → {Point}

One Iteration Step of the entire Algorithm.

Parameters:

Name	Type	Description
x	number	x-coordinate of the node's center.
y	number	y-coordinate of the node's center.
m	number	Starting iteration number.

Name	Type	Description
mmax	number	Maximum Iteration Number, i.e. Amount of calculated spiral points.
d	number	"Orientation" of the spiral. -1 is clockwise, 1 counterclockwise.
c	number	Curvature of the spiral. The higher the value, the more curve.
r	number	Radius of the Spiral.

Source: labeling/LabelAlgo.js, line 213

Returns:

The m-th Point of the spiral.

Type

Point

Class: LabelDefault

LabelDefault()

new LabelDefault()

Rather a namespace than a class, containing all functions (statically) that are used for Default Labeling. Thus, every function gets passed labeler as reference as last argument, because it's necessary to access some attributes of it. Its refactored this way to reduce clutter inside Labeler.

Source: labeling/LabelDefault.js, line 23

Methods

(async, generator, static) make_labels(vertices, labeler) → {LabeledVertex}

The labeling process of the vertices. Tries to find a valid label for each vertex (until Maximum is hit), with different procedures. Updates all necessary attributes during this, so that the yielded vertex is "ready to be processed".

Parameters:

Name	Type	Description
vertices	Generator.<Vertex>	The visible vertices to test for valid labels.
labeler	Labeler	A Reference of Labeler (main).

Source: labeling/LabelDefault.js, line 39

Yields:

the label-able vertex, still needs to be drawn and animated.

Type
LabeledVertex

get_filter(bbox, labeler) → {Array.<(Vertex|LabelTextBox)>}

Filter contains all elements of the regions of the quadtree, that the given BoundingBox overlaps. These are possible conflicts and need to be checked.

Parameters:

Name	Type	Description
bbox	BoundingBox	Bounding Box of a potential Label.
labeler	Labeler	A Reference of Labeler (main).

Source: labeling/LabelDefault.js, line 90

Returns:

Vertices and Already set Labels that are in regions overlapped by the given

Bounding Box.

Type
Array.<(Vertex|LabelTextBox)>

(private) is_valid(v, fn, labeler) → {boolean}

Creates valid label box positions with no overlaps to other objects except connection lines between nodes or labels, Uses different procedures, depending on the given function fn.

Parameters:

Name	Type	Description
v	Vertex	The vertex that is tried to be labeled.
fn	function	The labeling function.
labeler	Labeler	A Reference of Labeler (main).

Source: labeling/LabelDefault.js, line 113

Returns:

True, if a valid label position was found.

Type
boolean

(private) slider_prefilter(v, labeler) → {Array.<(Vertex|LabelTextBox)>}

Since the possible Labels of slider_model have a lot of overlap, to avoid descending the Quadtree everytime (which is costly), only to get nearly identical possible conflicts, which would be done with get_filter: Do one check for the entire slider area and check these elements. If there aren't a lot of elements to begin with, this is faster.

Parameters:

Name	Type	Description
v	Vertex	The Vertex slider_model is used on.
labeler	Labeler	A Reference of Labeler (main).

Source: labeling/LabelDefault.js, line 170

Returns:

Vertices and Already set Labels that are in regions overlapped by the given Slider Area.

Type
Array.<(Vertex|LabelTextBox)>

Class: LabelEdge

LabelEdge()

new LabelEdge()

Rather a namespace than a class, containing all functions (statically) that are used for Edge Labeling. Thus, every function gets passed labeler as reference as last argument, because it's necessary to access some attributes of it. Its refactored this way to reduce clutter inside Labeler.

Source: labeling/LabelEdge.js, line 25

Methods

(async, generator, static) make_labels(edges, labeler) → {LabeledEdge}

The labeling process of the edges. First checking for Midpoint label, and then doing interpolation along the edge, if defined in Consts. Updates all necessary attributes during this, so that the yielded edge is "ready to be processed".

Parameters:

Name	Type	Description
edges	Generator.<Edge>	The visible edges to test for valid labels.
labeler	Labeler	A Reference of Labeler (main).

Source: labeling/LabelEdge.js, line 41

Yields:

The label-able edge, still needs to be drawn and animated.

Type
LabeledEdge

(private) valid_interpolation(e, labeler) → {boolean}

The labeling process of the edges. First checking for Midpoint label, and then doing interpolation along the edge, if defined in Consts. Updates all necessary attributes during this, so that the yielded edge is "ready to be processed".

Parameters:

Name	Type	Description
e	Edge	Edge to test for valid labels.
labeler	Labeler	A Reference of Labeler (main).

Source: labeling/LabelEdge.js, line 116

Returns:

True, if valid interpolated label positon was found.

Type
boolean

(private) valid_midpoint(e, bbox, labeler) → {boolean}

Checks, if there is a valid label position on midpoint of an edge, and sets it ups, if yes.

Parameters:

Name	Type	Description
e	Edge	Edge to test for valid labels.
bbox	BoundingBox	Bounding box of midpoint label.
labeler	Labeler	A Reference of Labeler (main).

Source: labeling/LabelEdge.js, line 84

Returns:

True, if midpoint position is valid.

Type
boolean

Class: LabelTextBox

(abstract) LabelTextBox(**labeltext, world_coordinates, font_size**)

(abstract) new LabelTextBox(labeltext, world_coordinates, font_size)

Real Abstract Class, that inherits to the specific LabelBoxes, i.e. LabelTextVertex and LabelTextEdge.

Parameters:

Name	Type	Description
labeltext	string	The label's name/text.
world_coordinates	Point	An array of the nodes center coordinates in world coordinates. See Viewport unproject.
font_size	number	The font's size, as set in SettingsUI.

Source: labeling/LabelTextBox.js, line 13

Methods

(private) calc_fontwidth(fontsize) → {TextMetrics}

Calculates the width of the label box.

Parameters:

Name	Type	Description
fontsize	number	The font's size.

Source: labeling/LabelTextBox.js, line 104

Returns:

An array of the boxes' metadata including the label's width.

Type
TextMetrics

calc_width_height(fontsize)

Calculates the height of the label box.

Parameters:

Name	Type	Description
fontsize	number	The font's size.

Source: labeling/LabelTextBox.js, line 63

fade_in(coords)

Fade the label in, thus increase the opacity.

Parameters:

Name	Type	Description
coords	Point	The coordinates of the used vertex's center in world coordinates.

Source: labeling/LabelTextBox.js, line 81

fade_out()

Fade the label out thus decrease the opacity.

Source: labeling/LabelTextBox.js, line 92

width() → {number}

Just a "shortcut" or alias to save some screen space.

Source: labeling/LabelTextBox.js, line 54

Returns:

The width of a Label, depending on Length of String and Font Size.

Type
number

world_position(world_coordinates)

Resets the world_position with a new Animatable.

Parameters:

Name	Type	Description
world_coordinates	Point	2D-Coordinates.

Source: labeling/LabelTextBox.js, line 44

Class: LabelTextEdge

LabelTextEdge(**label, world_coordinates, font_size**)

new LabelTextEdge(label, world_coordinates, font_size)

Class representing the label of an edge.

Parameters:

Name	Type	Description
label	string	The label's name/text, coming from the corresponding Edge.
world_coordinates	Point	Default arg: [0,0] gets passed to super constructor.
font_size	number	The font's size, default arg defined via Consts.

Source: labeling/LabelTextBox.js, line 140

Classes

LabelTextEdge

Documentation generated by JSDoc 3.6.10 on Wed Feb 02 2022 18:13:43 GMT+0000 (Coordinated Universal Time)

Class: LabelTextVertex

LabelTextVertex(label, world_coordinates, font_size)

new LabelTextVertex(label, world_coordinates, font_size)

Class representing the label of a vertex.

Parameters:

Name	Type	Description
label	string	The label's name/text, coming from the corresponding Vertex.
world_coordinates	Point	Default arg: [0,0] gets passed to super constructor.
font_size	number	The font's size, default arg defined via Consts.

Source: labeling/LabelTextBox.js, line 116

Classes

LabelTextVertex

Documentation generated by JSDoc 3.6.10 on Wed Feb 02 2022 18:13:43 GMT+0000 (Coordinated Universal Time)

Class: LabelZoom

LabelZoom()

new LabelZoom()

Rather a namespace than a class, containing all functions (statically) that are used for Zoom Labeling. Thus, every function gets passed labeler as reference as last argument, because it's necessary to access some attributes of it. Its refactored this way to reduce clutter inside Labeler.

Source: labeling/LabelZoom.js, line 17

Methods

(async, generator, static) make_labels(elems, vertices, labeler) → {LabeledVertex}

The labeling process of the vertices, when zoom out and Graph is cramped. Tries to find a valid label for each vertex (until Maximum is hit), with different procedures. Updates all necessary attributes during this, so that the yielded vertex is "ready to be processed".

Parameters:

Name	Type	Description
elems	Array. <BoundingBox>	An array that initially consists only the graph's bounding box.
vertices	Generator. <Vertex>	The vertex to label.
labeler	Labeler	A Reference of Labeler (main).

Source: labeling/LabelZoom.js, line 35

Yields:

The label-able vertex with a connection line between label and node, still needs to be drawn and animated.

Type
LabeledVertex

(private) is_valid(v, elems, labeler) → {boolean}

Creates valid label box positions with no overlaps to other objects except connection lines between nodes and labels. Used for labeling cramped graphs and when zoom out, i.e. space around is available.

Parameters:

Name	Type	Description
v	Vertex	The vertex that is tried to be labeled.
elems	Array. <BoundingBox>	An array that consists of set labels' bounding boxes and the graph's bounding box.

Name	Type	Description
labeler	Labeler	A Reference of Labeler (main).

Source: labeling/LabelZoom.js, line 86

Returns:

True, if a valid label position was found.

Type
boolean

Class: Labeler

Labeler**(main)**

new Labeler(main)

Main Class, bundling all functionality implemented.

Parameters:

Name	Type	Description
main	iGraph	The iGraph system.

Source: labeling/Labeler.js, line 26

Classes

Labeler

Methods

add_label(element)

Adds the label box to the region-vertex-map of Quadtree. Loosely said, this is the "backend" operation of Label adding.

Parameters:

Name	Type	Description
element	Element	A Grid Element (e.g. vertices or edges).

Source: labeling/Labeler.js, line 94

(async, private) animate(time, label_lines) → {Promise.<boolean>}

Animation process of the labeler. Animates the graph elements.

Parameters:

Name	Type	Description
time	number	The current timestamp of the animation frame.
label_lines	Array.<(Element|Line)>	An array of elements and lines. The lines are optional.

Source: labeling/Labeler.js, line 131

Returns:

True, if the label needs further updates.

Type
Promise.<boolean>

(async, private) draw(label_box)

Draws the labels at the animated label position.

Parameters:

Name	Type	Description
label_box	Array.<(Element|Line)>	An array of elements and lines. The lines are optional.

Source: labeling/Labeler.js, line 141

(private) has_zoom_place(prefilter) → {boolean}

Checks, if there is enough space around the graph to place labels, i.e. when zoomed out.

Parameters:

Name	Type	Description
prefilter	BoundingBox	A Bounding Box around the entire graph with some buffer space.

Source: labeling/Labeler.js, line 302

Returns:

True, if there is space to place labels around the Graph.

Type
boolean

(private) make_edges(nodelink) → {Generator.<LabeledEdge>}

Does involve every calculation and check from updating the edge position, then generating possible labels until finding a valid one. These will be yielded by the Generator.

Parameters:

Name	Type	Description
nodelink	NodeLink	The original graph data from iGraph.

Source: labeling/Labeler.js, line 188

Returns:

A Generator Function producing visible Edges.

Type
Generator.<LabeledEdge>

(private) make_vertices(nodelink, context) → {Generator.<LabeledVertex>}

Does involve every calculation and check from updating the vertex position, then generating possible labels until finding a valid one. These will be yielded by the Generator.

Parameters:

Name	Type	Description
nodelink	NodeLink	The original graph data from iGraph.
context	Object	The context from iGraph.

Source: labeling/Labeler.js, line 153

Returns:

A Generator Function producing visible Vertices.

Type
Generator.<LabeledVertex>

(async) run(main, time) → {Promise.<boolean>}

Creates and labels the graph elements (e.g. nodes or edges).

Parameters:

Name	Type	Description
main	iGraph	The iGraph system.
time	number	The current timestamp of the animation frame.

Source: labeling/Labeler.js, line 58

Returns:

Due to animations, returns true if at least one label needs further updates.

Type
Promise.<boolean>

(private) set_labeling_mode(vertex_amount)

Depending on the visible vertex amount and the thresholds defined in Consts, an adequate Labeling Mode is chosen. The higher the Quality, the more labeling procedures are used.

Parameters:

Name	Type	Description
vertex_amount	number	The amount of visible vertices.

Source: labeling/Labeler.js, line 279

(private) update_context(context)

Set the current canvas context (from iGraph).

Parameters:

Name	Type	Description
context	Object	The 2d-canvas' context.

Source: labeling/Labeler.js, line 112

(private) update_quadtree(vertices)

Resets Quadtree Region Map and Inefficiency (also for Vertices) and then creates a new Quadtree.

Parameters:

Name	Type	Description
vertices	Array. <Vertex>	The visible vertices the quadtree is constructed upon.

Source: labeling/Labeler.js, line 198

Class: NodeLink

NodeLink()

new NodeLink()

FILE OF iGraph. WE DID NOT WRITE THIS FILE.

The original data like x,y-Coordinates, Size of Radius, Label names, etc. is all parsed from this "Graph Representation" of iGraph.

Source: iGraph/NodeLink.js, line 16

Classes

NodeLink
FILE OF iGraph. WE DID NOT WRITE THIS FILE.

Documentation generated by JSDoc 3.6.10 on Wed Feb 02 2022 18:13:43 GMT+0000 (Coordinated Universal Time)

Class: Quadtree

Quadtree(vertices, bounds, max_depth, key)

new Quadtree(vertices, bounds, max_depth, key)

A specific implementation of a k-D tree, to minimize check for conflicts. The idea is to divide the 2D-space into distinct regions so that elements are uniformly distributed (which means regions are not equally big). Then checks only need to be done inside a region for each Element.

Parameters:

Name	Type	Description
vertices	Array.<Vertex>	All vertices visible inside the bounds.
bounds	BoundingBox	Screen bounds - for root, it's the entire screen, for everything else.
max_depth	number	Gets initialized as set in Consts and then decremented by one per recursion layer.
key	string	Used as key for the static region_vertex_map to "name" each subregion. Get initialized as empty string in root, and from there on adds number per recursion layer: 0 - top_left, 1 - top_right, 2 - bottom_left, 3 - bottom_right.

Source: labeling/Quadtree.js, line 15

Classes

Quadtree

Methods

get_label_regions(bbox) → {Array.<string>}

Given a Bounding Box of a potential Label, descend the Quadtree via DFS (starting from Root Node with key "") and find all the subregions, this Bounding box overlaps.

Parameters:

Name	Type	Description
bbox	BoundingBox	A potential Labels Bounding Box.

Source: labeling/Quadtree.js, line 99

Returns:

The keys, that are string, of all subregions of the quadtree that are overlapped by the Bounding Box.

Type

Array.<string>

(private) recurse(max_depth)

Used in constructor. Makes a new Quadtree Node for each subregion with its corresponding bounds. Decrements Depth by 1 and appends to the key/makes key for the new (sub-)region.

Parameters:

Name	Type	Description
max_depth	number	still remaining recursion steps.

Source: labeling/Quadtree.js, line 173

(private) split_x(vertices) → {Array.<Array.<Vertex>>}

The X-Split is done second and thus splits each of the two halves from the Y-Split in two halves again, i.e. quarters.

Parameters:

Name	Type	Description
vertices	Array.<Vertex>	Either the top or bottom half of vertices after Y-Split.

Source: labeling/Quadtree.js, line 154

Returns:

Left Vertices and Right Vertices of either the top or bottom half.

Type

Array.<Array.<Vertex>>

(private) split_y(vertices) → {Array.<Array.<Vertex>>}

The Y-Split is done first and thus splits in two halves.

Parameters:

Name	Type	Description
vertices	Array.<Vertex>	All vertices of the region.

Source: labeling/Quadtree.js, line 133

Returns:

Top Vertices and Bottom Vertices after splitting up.

Type

Array.<Array.<Vertex>>

Class: SettingsUI

SettingsUI(main)

new SettingsUI(main)

Keeps track of label font_size and amount of labels shown while changing these values. Therefore, it sets a new object responsible for this task upon creation inside the "main" class.

Parameters:

Name	Type	Description
main	iGraph	class which this project is built on top of.

Source: labeling/SettingsUI.js, line 10

Classes

SettingsUI

Members

settings

A Reference to the Object that manages Font Size and Labels shown in iGraph.

Properties:

Name	Type	Description
active	boolean	Toggle Labels shown (on/off).
label_amount	number	Reads in from Consts.
font_size	number	Reads in from Consts.
changed_font_size	boolean	Keeps track, if font size was changed compared to last update.
inc_label_amount	function	Increases Amount of Labels shown (checks against a pre-defined maximum (Consts)).
dec_label_amount	function	Decreases Amount of Labels shown (checks against a pre-defined minimum (0)).
inc_font_size	function	Increases Font Size (checks against a pre-defined maximum (Consts)).
dec_font_size	function	Decreases Font Size (checks against a pre-defined minimum (Consts)).
is_max_label	function	Returns boolean (check, if Maximum Amount of Labels are shown).
is_min_label	function	Returns boolean (check, if Minimum Amount of Labels are shown).
is_max_font_size	function	Returns boolean (check, if Font Size is Maximum).

Name	Type	Description
is_min_font_size	function	Returns boolean (check, if Font Size is Minimum).

Source: labeling/SettingsUI.js, line 76

Methods

font_size_edge() → {number}

Source: labeling/SettingsUI.js, line 93

Returns:

Font Size of Edges.

Type
number

font_size_vertex() → {number}

Recent Fontsize multiplied with a Ratio Constant set in Consts. (Default: 2).

Source: labeling/SettingsUI.js, line 85

Returns:

Font Size of Vertices.

Type
number

label_amount_default() → {number}

Source: labeling/SettingsUI.js, line 101

Returns:

Recent Amount of Labels shown in Default Labeling.

Type
number

label_amount_zoom() → {number}

Recent Amount of Labels shown divided by the LABEL_DELTA defined in Consts, since in Zoom Labeling de-/increments of 1 are done and thus, this compensates.

Source: labeling/SettingsUI.js, line 112

Returns:

Recent Amount of Labels shown in Zoom Labeling.

Type
number

Class: Vertex

Vertex(dot, labeltext, labelclass)

new Vertex(dot, labeltext, labelclass)

Representation of a Vertex/Node in iGraph, modeling its state and behavior.

Parameters:

Name	Type	Description
dot	Object	The original object from NodeLink, which gets processed when creating GridElements.
labeltext	string	The name/text of the corresponding element, that is used for labeling later.
labelclass	LabelTextBox	Specific Class (LabelTextVertex) for Vertices, storing and calculating all relevant Parameter, like Font Size, Label Width and Height, etc.

Source: labeling/Element.js, line 61

Classes

Vertex

Methods

label_info() → {Array.<number>}

Just a "shortcut" or alias to save some screen space.

Source: labeling/Element.js, line 89

Returns:

It's used in combination with the unpack-operator (...) to give args to functions.

Type
Array.<number>

update_xy(x, y)

Updates the Center Point of the Vertex as well as the Bbox.

Parameters:

Name	Type	Description
x	number	New x-value of the Vertex.
y	number	New y-value of the Vertex.

Source: labeling/Element.js, line 110

xyr() → {Array.<number>}

Just a "shortcut" or alias to save some screen space.

Source: labeling/Element.js, line 100

Returns:

It's used in combination with the unpack-operator (...) to give args to functions.

Type

Array.<number>

Class: Viewport

Viewport()

new Viewport()

FILE OF iGraph. WE DID NOT WRITE THIS FILE.

Used for Animatable, and thus Animator. From iGraph original.

Source: base/Viewport.js, line 11

Classes

Viewport
FILE OF iGraph. WE DID NOT WRITE THIS FILE.

Documentation generated by JSDoc 3.6.10 on Wed Feb 02 2022 18:13:43 GMT+0000 (Coordinated Universal Time)

Global

Members

(constant) Consts

Consts is a Map to store all Magic Constants with a descriptive name and make them reusable throughout the project.

Properties:

Name	Type	Description
MAX_VERTEX	70	In Default Labeling Modes: Default Maximum Amount of Vertices shown.
ZOOM_MAX_VERTEX	15	In Zoom Labeling Mode: Default Maximum Amount of Vertices shown.
MAX_LABEL	100	Maximum Amount overall, Vertices and Edges combined.
EDGE_SEPARATOR	" | "	String to separate the labels of the two corresponding vertices to make the Edge Label.
INC_SLIDER	10	Increment value, in which slider_model changes x-/y-values.
DEFAULT_SAMPLE	Array. <number>	Args used for sample_algorithm when LabelingMode is HIGH_QUALITY.
ZOOM_SAMPLE	Array. <number>	Args used for sample_algorithm when LabelingMode is ZOOM_PERFORMANCE.
INEFFICIENT_THRESHOLD	4	Any region containing more than the specified amount of vertices makes the Quadtree inefficient.
INEFFICIENT_BUFFER	1	Value gets added/subtracted to/from Threshold to avoid fast switching between Labeling Modes.
MAX_DEPTH	5	Maximum Depth of Recursion for Quadtree to avoid Stack Overflows.
TEXT_ALIGN	"center"	Text Align (for HTML).
TEXT_BASELINE	"middle"	Text Baseline (for HTML).
FONT	"Helvetica Neue"	Font used (for HTML).
FONT_COLOR	"rgb(52, 52, 52)"	Font Color used (for HTML).
LINE_COLOR	"rgb(122, 122, 122)"	Line Color used (for HTML).

Name	Type	Description
FONT_SIZE	14	Default Edge Font Size, Vertex is doubled. Can be changed for Labeling with SettingsUI.
MIN_FONT_SIZE	8	Default Minimum Edge Font Size. SettingsUI respects that minimum while changing Font Size.
MAX_FONT_SIZE	34	Default Maximum Edge Font Size. SettingsUI respects that maximum while changing Font Size.
OPACITY_FULL	1	No Transparency, fully faded in.
HIGH_QUALITY_THRESHOLD	100	If less Vertices than this amount, LabelingMode is set to HIGH_QUALITY.
QUALITY_THRESHOLD	500	If less Vertices than this amount (but more than HIGH_QUALITY_THRESHOLD), LabelingMode is set to QUALITY.
PERFORMANCE_THRESHOLD	1000	If more Vertices than this amount, LabelingMode is set to HIGH_PERFORMANCE. If less (but more than QUALITY_THRESHOLD), LabelingMode is set to PERFORMANCE.
EPSILON	0.000000001	Typical Epsilon to avoid unexpected behavior with floating point numbers.
INTERPOLATION_STEPS	2	Amount of Interpolation Points for Edge Labeling. Should be an even number. Midpoint is always done, so a Value of e.g. 2 means: 1/4 and 3/4 of the Edge.
ZOOM_BUFFER_FACTOR	1.25	Used when LabelingMode is set to ZOOM_PERFORMANCE, to scale the buffer zone "around" the Graph (useful to prevent Labels from stick to close to the graph).
LABEL_DELTA	5	When changing Amount of Labels via SettingsUI, this value is used as de-/increment. Should be divisor of MAX_VERTEX and MAX_LABEL.
FONT_SIZE_DELTA	2	When changing Font Size via SettingsUI, this value is used as de-/increment.
FONT_SIZE_RATIO	2	The scaling Factor used for Vertex Font Size (with Edge Font Size as "baseline").

Source: labeling/Consts.js, line 91

(constant) LabelingMode

LabelingMode is an Enum to change the way Labeling is done. Everything - except ZOOM_PERFORMANCE, as it works totally different - counts as "Default Labeling".

Properties:

Name	Type	Description
ZOOM_PERFORMANCE	0	If there is enough space around the Graph (and Quadtree is inefficient), this mode is used. Places Labels around the Graph with special "zoom" functions in Labeler.
HIGH_PERFORMANCE	1	Set via specific Thresholds in Consts. Uses only four_pos to label.
PERFORMANCE	2	Set via specific Thresholds in Consts. Uses only four_pos and eight_pos to label.
QUALITY	3	Set via specific Thresholds in Consts. Uses four_pos, eight_pos and slider_model to label.
HIGH_QUALITY	4	Set via specific Thresholds in Consts. Uses four_pos, eight_pos, slider_model and sample_algorithm to label.

Source: labeling/Consts.js, line 163

(constant) UnderlineLabel

UnderlineLabel is an Enum to determine the position of the "visual-aid-line" connecting Vertex and Label in eight_pos and sample_algorithm relative to the label.

Properties:

Name	Type	Description
TOP	0	Line connects to top of the label.
BOTTOM	1	Line connects to bottom of the label.
LEFT	2	Line connects to left of the label.
RIGHT	3	Line connects to right of the label.

Source: labeling/Consts.js, line 206

(constant) Visibility

Visibility is an Enum to determine the Animator behavior after the most recent updates.

Properties:

Name	Type	Description
FADED_IN	0	Visibility State for Labels were not visible before, but are visible now.
FADED_OUT	1	Visibility State for Labels were visible before, but are not visible now.
VISIBLE	2	Visibility State for Labels were visible before and are still visible now.

Source: labeling/Consts.js, line 186

Type Definitions

BoundingBox

[x_min, y_min,x_max, y_max]

Type:

- Array.<number>

Source: labeling/BBox.js, line 6

LabeledEdge

[e] (to match the array format of vertices and use the same functions for Drawer and Animator

Type:

- Array.<Edge>

Source: labeling/LabelEdge.js, line 10

LabeledVertex

[v] or [v,[v.x,v.y]]

Type:

- Array.<(Vertex|Point)>

Source: labeling/LabelDefault.js, line 10

Line

[x_node,y_node,x_bbox,y_bbox,UnderlineLabel]

Type:

- Array.<(number|UnderlineLabel)>

Source: labeling/Animator.js, line 7

Point

[x,y]

Type:

- Array.<number>

Source: labeling/BBox.js, line 1

Class: iGraph

iGraph()

new iGraph()

FILE OF iGraph. WE DID NOT WRITE THIS FILE.

The original Program our Labeling Project is written on top on. It contains all "Graph Representation" in NodeLink.

Source: iGraph/iGraph.js, line 15

Classes

iGraph
FILE OF iGraph. WE DID NOT WRITE THIS FILE.

Methods

data(graph)

After NodeLink is created, we immediately parse and process all information from the Graph and create Labeler.

Parameters:

Name	Type	Description
graph	Graph	Contains Information about Nodes and Links.

Source: iGraph/iGraph.js, line 35

update(time)

The usual update loop of iGraph with the only addition, that after the Grid and NodeLink is drawn, Labeler does its async run function, that might still update after finishing calculations, even though for iGraphs own needUpdate variable, there was no need to update.

Parameters:

Name	Type	Description
time	number	The current timestamp of the animation frame.

Source: iGraph/iGraph.js, line 352

Home

Documentation generated by JSDoc 3.6.10 on Wed Feb 02 2022 18:13:43 GMT+0000 (Coordinated Universal Time)