Dokumentation: Implementierung des Sugiyama-Algorithmus

Lukas Kohlhaas, Timur Sultanov, Nicolas Steinbach

1 Einleitung

Die Projektaufgabe bestand darin, den Sugiyama-Algorithmus in der Programmiersprache C++ unter Zuhilfenahme der Bibliothek LEDA umzusetzen. Das Team umfasste Lukas Kohlhaas (s4llkohl@uni-trier.de), Timur Sultanov (s4trsult@uni-trier.de) und Nicolas Steinbach (s4nsstei@uni-trier.de). Wir orientierten uns hauptsächlich an dem Vorlesungsskript 'Visualization of Graphs - Lecture 8: Hierarchical Layouts: Sugiyama Framework' von Philipp Kindermann. Der Algorithmus wurde in einzelne Teilschritte/Steps untergliedert, die auf die Team-Mitglieder aufgeteilt wurden.

- Nicolas: 'Cycle-Breaking' nach der Heuristik von Berger und Shor
- Timur: 'Leveling' und 'Edge Drawing'
- Lukas: 'Crossing Minimization' und 'Vertex Positioning'

2 Motivation und Absicht

Nicolas entwickelte die Software-Architektur des Projekts mit, wobei neben dem logischen Primärziel, den Sugiyama-Algorithmus fehlerfrei in Quellcode zu überführen und ausführbar zu machen, folgende Aspekte ebenso im Vordergrund standen:

a) Eine möglichst einfache API für den Sugiyama-Algorithmus erzeugen. Allerdings trotzdem die Möglichkeit erhalten, wichtige Konfigurationen vornehmen zu können.

b) Neue Schritte/Steps sollten ohne großen Aufwand ergänzt werden können. Ebenso sollten bereits existierende Schritte durch neue Implementierungen ersetzt werden können. Denn es gibt häufig mehrere Heuristiken/Algorithmen für den gleichen Step, wie etwa 'Cycle-Breaking' (nach [Berger, Shor '90] und nach [Eades, Lin, Smyth '93]) oder 'Crossing-Minimization' (mithilfe der Barycenter Heuristik [Sugiyama et al. '81] und Median Heuristik [Eades & Wormald '94])
Mehrere Beispiele für ergänzende Steps wären etwa das Löschen von

Hilfsknoten, die beim Step 'Leveling' hinzugefügt werden mussten, oder das Herstellen der ursprünglichen Kanten, welche im Step 'Cycle-Breaking' umgekehrt werden mussten. Diese könnte man nach Belieben dem Programm hinzufügen oder weglassen.

c) Simples Navigieren/Iterieren über die Ergebnisse des Sugiyama-Algorithmus sollte möglich sein:

   i. Ein User-Interface für die Navigation, welches man selbst konfigurieren und austauschen kann.

   ii. Das Iterieren über die Ergebnisse sollte vorwärts und rückwärts möglich sein.

   iii. Die Ergebnisse sollten einmal vorberechnet und dann gecachet werden, sodass sie auf Abruf verfügbar sind. Dadurch entstehen keine Verzögerungen bei der Navigation durch aufwendige Zwischenberechnungen - wie etwa die Crossing-Minimization.

3 Aufbau

Die Architektur wird durch das UML-Diagramm 'UML_Sugiyama.drawio' unter 'docs' dargestellt. Dabei benötigen einige Klassen und Assoziationen eine besondere Erklärung:

a) Die Klasse 'sugiyama' dient als eine Fassade (gemäß dem Design-Pattern 'Facade'), da es dem User eine übersichtliche API zum gesamten Subsystem liefert, wie zuvor in (2.a) gefordert:
Man muss dem 'sugiyama'-Objekt bei der Initialisierung nur das aktuelle Graph-Window übergeben. Anschließend kann man ihm mithilfe der Methode 'add(step)' die gewünschten Steps in der Reihenfolge hinzufügen, in der sie dann auch tatsächlich ausgeführt werden sollen, ähnlich dem Design-Pattern 'Strategy' mit mehreren Strategien. Zusätzlich sollte man das 'step_viewer'-Objekt festlegen, siehe Methode 'setStepViewer(step_viewer)', um mühelos über die berechneten Ergebnisse der Teilschritte navigieren zu können. Das 'step_viewer'-Objekt selbst muss dabei mit einer Implementierung des Interfaces 'step_user_interface' über die Methode 'setUI(step_user_interface)' konfiguriert werden. Die Ergebnisse der Steps lassen sich durch den Aufruf der Methode 'executeAllSteps()' berechnen. Anschließend kann man mithilfe der Methode 'viewAllSteps()' die Navigation über Jene starten.

b) Ein 'sugiyama'-Objekt wird automatisch mit einem Objekt der Klasse 'graph_update_tracker' initialisiert. Der 'graph_update_tracker' dient als Datenstruktur und ermöglicht es, Daten über alle Step-Objekte hinweg zu kommunizieren - wie etwa Knoten, die im Vergleich zum Original-Graphen neu hinzugefügt oder entfernt wurden.

c) Das Interface (genauer die Abstrakte Klasse) 'step' kennzeichnet explizit die einzelnen Teilschritte. Der Programmierer muss dabei ausschließlich die abstrakte Methode 'run()' implementieren. Die Klasse kann frei von Graphik- und Iterations-Logik implementiert werden und nur den tatsächlichen Algorithmus enthalten. Denn die Template-Methode 'execute' ruft intern die 'run()'-Methode auf und erstellt - unabhängig davon - für jede 'step'-Klasse einen Snapshot über die private Methode 'saveResult()'. Dieser Snapshot, bestehend aus Graph und Positionen, wird in einem 'positionable_graph'-Objekt gespeichert, welches mit dem jeweiligen Step assoziiert ist und damit nur einmal berechnet werden muss. Zwar benötigt dies mehr Speicherplatz - denn jedes Mal muss ein eigener Graph erzeugt werden und dessen Positionen berechnet werden - allerdings überwiegt meiner Meinung nach die bessere Performance diesen Nachteil, siehe (2.c.iii). Das 'step'-Interface bietet insgesamt den einfachsten Punkt, um den Algorithmus anzupassen oder zu erweitern, ohne existierenden Source-Code abändern zu müssen, siehe (2.b).

d) Jedes 'step'-Objekt hat Zugriff auf das aktuelle 'sugiyama'-Objekt - sodass Informationen, die global für alle Teilschritte interessant sein könnten (wie etwa der 'graph_update_tracker') direkt verfügbar sind.

e) Das Interface 'step_user_interface' hat nur eine abstrakte Methode 'nextStep()', welche nur eine Integer zurückgibt, sodass das User-Interface extrem leicht ohne künstliche Abhängigkeiten und Beschränkungen erweitert bzw. ausgetauscht werden kann (siehe (2.c.i)).

f) Der 'step_viewer' bietet dem User mit der Methode 'view()' die Möglichkeit, je nach Eingabe auf dem User-Interface sich einen Schritt vorwärts oder auch rückwärts zu bewegen, siehe (2.c.ii). Anschliessend zeigt es den mit dem aktuellen Step assoziierten Graphen und dessen Positionen mithilfe des aktuellen 'positionable_graph'-Objekts. Sollte der 'step_viewer' unabhängig von der Fassade 'sugiyama' verwendet werden, so muss man die gewünschten Steps manuell über die Methode 'setSteps(list<step>)' setzen.

4 Weitere Kommentare zum Projekt

4.1 Lernprozess (LEDA, C++, Make)

Einen Großteil der verfügbaren Projektzeit habe ich damit verbracht, die LEDA-Bibliothek kennen zu lernen, um diese als Grundlage für das Projekt nutzen zu können. Dabei waren Kenntnisse in C++ unerlässlich, da sowohl die LEDA-Bibliothek selbst darin implementiert wurde, als auch das Projekt darin entwickelt werden sollte. Den ersten Eindrücken nach ist C++ eine hochkomplexe Sprache und hat mich als Studenten mit seinen Konstrukten wie beispielsweise Header-Dateien, Pointers und Referenzen oder dem komplexen Build-Prozess (bestehend aus Kompilieren und Linken) weit mehr gefordert als die Sprachen Java oder Python, sodass alleine das Erlernen der Grundlagen von C++ bereits

einige Wochen Zeit gekostet hat. Da das Integrieren in einer IDE weitere Probleme mit sich führte und letztendlich nicht fehlerfrei funktioniert hatte, entschloss ich mich meinen Teil des Projekts über einen einfachen Editor umzusetzen, weshalb der Build-Prozess auch manuell vorgenommen werden musste. Letztendlich entschied ich mich dazu - wie das LEDA-Projekt auch - das Build-Tool 'Make' und dessen 'Makefiles' dafür zu verwenden.

4.2 Struktur und Erweiterbarkeit

Wir entschieden uns für eine Untergliederung des Projekts in die 3 Submodule bzw. Subprojekte 'architecture', 'implementation' und 'demo', da sich so der Aufbau des Projekts am besten wiederspiegeln lässt:

a) Das Projekt 'architecure' ist ausschließlich abhängig von der LEDA-Bibliothek und muss nur selten angepasst werden, weshalb sie die stabilste Komponente darstellt.

b) Das Projekt 'implementation' enthält unterschiedliche Implementierungen für die abstrakte Klasse 'step' und das Interface 'step_user_interface'.Sollten weitere konkrete Klassen benötigt werden, so kann man diese dem Projekt einfach hinzufügen, indem man neue Header-Dateien unter dem entsprechenden Ordner in 'incl', sowie zugehörige Implementierungen unter dem entsprechenden Ordner in 'src' hinzufügt. Des Weiteren muss man natürlich die Makefile ergänzen und den Build-Prozess neu anstoßen. Alternativ kann auch eine eigene, unabhängige (statische) Bibliothek erzeugt und dem Linker bei den Executables von 'demo' hinzufügt werden.

c) In dem Projekt 'demo' kann man beliebige Anwendungsfälle implementieren und übersetzen. Der Build-Prozess ist bereits so aufgesetzt, dass die Architektur aus (a) sowie die Implementierungen aus (b) zur Verfügung stehen, und man unabhängig davon die Demos hier schreiben, übersetzen und ausführen kann.

Neben der Ordnerstruktur, findet sich die Subprojekt-Struktur ebenfalls in den Makefiles und Git wieder.

4.3 Kommunikation und Versionskontrolle

Zur Versionskontrolle verwendeten wir das bekannte System 'Git' und stellten uns gegenseitig die Fortschritte auf 'GitHub' unter `https://github.com/Steini1998` zur Verfügung. Dabei ist jedes der 3 Submodule 'architecture', 'implementation' und 'demo' ein eigenständiges Git-Projekt 'Sugiyama_Architecture', 'Sugiyama_Implementations' und 'Sugiyama_Demos' und ebenfalls verfügbar auf der gleichen GitHub-Seite. Sie werden über das Parent-Projekt 'Sugiyama' zusammengeführt.

5 Einzelheiten zu den einzelnen Steps

5.1 Step 2: Leveling

In diesem Schritt hat man aufgrund des vorherigen Schrittes einen azyklischen Graphen gegeben. Ziel ist es nun, die Knoten in Schichten zuzuweisen. Dabei möchte man eine Abbildung $y : V \to \{1, \ldots, n\}$ definieren, sodass für jede Kante $uv \in E$ gilt: $y(u) < y(v)$. Hierfür setzt man für alle Knoten q, die keinen eingehende Kanten haben $y(q) = 0$. Ausgehend von diesen Quellknoten setzt man für die anderen Knoten v mit eingehenden Kanten:

$$y(v) = \max\{y(u) \mid uv \in E\} + 1.$$

Was man hier praktisch berechnet, ist also die Länge des längsten Pfades von einem Quellknoten zu einem nicht-Quellknoten v plus 1. Dies wird in der Funktion 'longest_path' mithilfe einer Queue in linearer Zeit implementiert.
Aufgrund der azyklischen Eigenschaft des Graphen, wodurch jeder Knoten maximal einmal in die Queue eingefügt wird und auch jede Kante genau einmal betrachtet wird, und da sämtliche Operationen wie zum Beispiel 'Q.pop()' und das Überprüfen der Bedingung $level[u] < level[v] + 1$ konstante Zeit benötigen, summieren sich die Kosten auf insgesamt $\mathcal{O}(|V| + |E|)$ pro Funktionsaufruf.
Hat man die Schichtzuweisung abgeschlossen, so werden nun sogenannte 'dummy_nodes' eingefügt, die später entfernt werden. Diese sollen zwei Knoten, bei denen mindestens eine Schicht dazwischen liegt, mit neuen Kanten verbinden. Die ursprüngliche Kante wird hierbei entfernt und später wieder hinzugefügt. Dafür wird ein Tracker benutzt, in welchem man die entfernten Kanten und hinzugefügten 'dummy_nodes' abspeichert, womit einem späteren Schritt durch den Tracker die Informationen übergeben werden können zur Wiederherstellung der ursprünglichen Knoten- und Kantenmenge.

Zum Schluss soll es also keine Kanten mehr geben, die über mehrere Schichten laufen. In der entsprechenden Funktion 'dummy_nodes' wird über alle Kanten iteriert und überprüft, ob der Schichtenunterschied der inzidenten Knoten größer ist als eins. Ist dies der Fall, dann wird die Kante entfernt und entsprechend viele Dummy-Knoten hinzugefügt, die mit neuen Kanten mit den inzidenten Knoten verbunden werden.

Zum Schluss dieses Schrittes werden den Knoten Positionen zugeordnet, sodass diese in Graphwin in ihren entsprechenden Schichten visualisiert werden. Die Funktion 'determine_positions' iteriert hier simpel über alle Knoten des Graphen und setzt die y-Koordinate auf $50 \cdot (level[v] + 1)$. Die x-Koordinate wird auch auf ein Vielfaches von 50 gesetzt.

Dabei hilft die Funktion 'position_empty', um herauszufinden, ob die Position, an der man den Knoten setzen will, nicht schon belegt ist. Es sollen sich ja keine Knoten überlappen in Graphwin, wodurch man manche Knoten dann nicht mehr sehen könnte. Ist an der Position schon ein Knoten, dann wird die

x-Koordinate solange um 50 erhöht, bis man an einer noch leeren Position angekommen ist.

Im Worst-Case braucht die Funktion 'determine_positions' eine Laufzeit von $\mathcal{O}(|V|^2)$, da in 'determine_positions' über alle Knoten iteriert wird und man in dieser Schleife dann im Worst-Case in der Funktion 'position_empty' nochmals eine Schleife hat, die insgesamt $|V|$-mal ausgeführt wird. Es ist aber möglich, die Laufzeit auf $\mathcal{O}(|V|)$ zu reduzieren. Man könnte zum Beispiel freie und belegte Positionen in getrennte Listen abspeichern, die man dann bei der Platzierung eines neuen Knotens aktualisiert.

5.2 Step 3: Crossing Reduction

Der Kern der Kreuzungsminimierung ist die Methode count_edge_crossings, welche die Anzahl aller Kantenkreuzungen im Graphen zählt. Da Graphen unabhängig der Knoten- und Kantenpositionen definiert sind, besitzt die LEDA Bibliothek keine Methode die dies automatisch berechnet. Entsprechend wurden dafür neue Methoden implementiert, die zuerst aus den Positionen aus dem zugehörigen Node Array alle Knoten als Punkte und alle Kanten als Segmente neu definiert. Segmente haben für diesen Schritt den Vorteil, dass LEDA bereits eine Methode zur Verfügung stellt, die zurück gibt ob sich zwei Segmente kreuzen. Da eine Segmentkreuzung auch dann besteht, wenn zwei Kanten sich am Ursprung/Ziel berühren, schließen wir diese Kreuzungen sofort aus, um eine korrekte Zählung zu gewährleisten. count_edge_crossings iteriert einmal über alle Kanten des Graphen und innerhalb dieser Schleife noch mal über alle Kanten des Graphen. Dabei nimmt sie für jedes Paar unterschiedlicher Kanten des Graphen die Koordinaten der Ursprünge und Die der Ziele aus dem Node Array der Positionen und generiert daraus zwei Segmente. Diese werden an eine Methode segments_intersect weiter gegeben, die oben genanten 'Berührungsfall' ausschließt und danach mit der von LEDA gegebenen Methode einen bool-Wert zurück gibt. Abhängig von diesem Wert wird die Anzahl der Kreuzungen inkrementiert. Zuletzt wird dieser Wert noch halbiert, da die Menge der geprüften Kantenpaare eine geordnete Menge ist und so jedes Kantenpaar doppelt überprüft und gezählt wird. Die Methode count_edge_crossings hat eine Laufzeit von $\mathcal{O}(n^2)$ und wir brauchen sie nach jeder Änderung am Positionen-Array. Für die Median- und Barycenter-Ansätze passiert das zwar nur einmal am Ende, bei Greedy allerdings für jede Tauschoperation. Entsprechend ist die Verwendung des Greedy Ansatzes für sehr große Graphen nicht zu empfehlen.

Die Methode greedy_crossing_reduction implementiert eine heuristische Strategie zur Reduktion von Kantenkreuzungen, indem die Positionen der Knoten in einzelnen Schichten des Graphen iterativ optimiert werden. Der Prozess beginnt mit der Festlegung einer Startkoordinate $yCoord = 50$, welche die erste Schicht des Graphen repräsentiert. Für jede Schicht werden alle Knoten mit der entsprechenden y-Koordinate in einer Liste gesammelt. Sind in der aktuellen Schicht keine Knoten mehr vorhanden, wird die Schleife abgebrochen. Die

gesammelten Knoten werden anschließend nach ihren x-Koordinaten aufsteigend sortiert, um die Reihenfolge für mögliche Tauschoperationen festzulegen. Für jedes benachbarte Knotenpaar u und w, das einen Abstand von exakt 50 Einheiten in der x-Koordinate aufweist, wird geprüft, ob ein Tausch der Positionen die Anzahl der Kantenkreuzungen reduziert. Hierbei kommt die Methode swap_and_check zum Einsatz, welche zunächst die Anzahl der aktuellen Kantenkreuzungen mit der Methode count_edge_crossings berechnet. Anschließend tauscht die Methode die Positionen der beiden Knoten temporär und berechnet die Kantenkreuzungen erneut. Falls die Anzahl der Kreuzungen nicht reduziert wird, werden die Positionen der Knoten zurückgetauscht; andernfalls verbleiben sie in der neuen Anordnung. Damit besitzt swap_and_check eine Laufzeit von $\mathcal{O}(n^2)$, da sie durch zwei Aufrufe von count_edge_crossings dominiert wird. Nach Abschluss der Überprüfung einer Schicht wird die y-Koordinate um 50 Einheiten erhöht, und der Prozess wird für die nächste Schicht wiederholt. Der Algorithmus endet, wenn keine weiteren Knoten in einer Schicht vorhanden sind. Da die Methode für jede Schicht potenziell alle benachbarten Knotenpaare überprüft und für jedes Paar swap_and_check aufruft, ergibt sich die Laufzeit von greedy_crossing_reduction als $\mathcal{O}(k \cdot m^2)$, wobei k die Anzahl der Schichten und m die Anzahl der Kanten im Graphen ist. Dies macht die Methode insbesondere für sehr große Graphen mit vielen Knoten und Kanten ineffizient.

Die Methode calculate_barycenter berechnet den baryzentrischen Mittelpunkt der Nachbarknoten eines gegebenen Knotens v im Graphen. Sie summiert zunächst die x-Koordinaten aller Zielknoten der ausgehenden Kanten von v und speichert diese Summe in der Variable barycenter_x. Gleichzeitig zählt sie die Anzahl der betrachteten Kanten, gespeichert in der Variablen degree. Sollte v keine ausgehenden Kanten haben, bleibt degree gleich null, und der baryzentrische Mittelpunkt wird nicht verändert. Existieren jedoch ausgehende Kanten, wird die Summe der x-Koordinaten durch die Anzahl der betrachteten Knoten geteilt, um den Durchschnitt zu berechnen. Die Methode gibt den berechneten baryzentrischen Wert als double zurück. Da der Aufwand für die Iteration über alle ausgehenden Kanten eines Knotens linear in der Anzahl der Kanten ist, besitzt calculate_barycenter eine Laufzeit von $\mathcal{O}(d)$, wobei d den Grad des Knotens v bezeichnet. Die Methode barycenter_crossing_reduction nutzt den Barycenter-Ansatz zur Minimierung von Kantenkreuzungen in einem Graphen. Sie beginnt mit der Erstellung eines temporären Arrays y_coords, das die y-Koordinaten aller Knoten speichert. Anschließend wird die maximale y-Koordinate im Graphen ermittelt, um die höchste Schicht zu bestimmen. Die Methode iteriert danach über alle Schichten des Graphen in absteigender Reihenfolge, beginnend mit der höchsten Schicht. Für jede Schicht werden die Knoten gesammelt, deren y-Koordinate mit der aktuellen Schicht übereinstimmt. Diese Knoten werden basierend auf ihrem Barycenter-Wert sortiert, welcher für jeden Knoten mit der Methode calculate_barycenter berechnet wird. Nach der Sortierung werden die Knoten in der Schicht neu positioniert: Jeder Knoten erhält eine neue x-Koordinate, die auf dem berechneten Barycenter basiert. Um Überlappungen zu vermeiden, wird sichergestellt, dass die neue x-Koordinate mindestens 50 Einhei-

ten größer ist als die x-Koordinate des vorherigen Knotens. Die y-Koordinate der Knoten bleibt dabei unverändert. Die Methode barycenter_crossing_reduction iteriert über alle Schichten des Graphen, sortiert die Knoten einer Schicht nach ihrem Barycenter-Wert und berechnet anschließend neue Positionen. Im Best-Case, bei gleichmäßig verteilten Knoten mit wenigen Kanten, beträgt die Laufzeit $\mathcal{O}(N \cdot \log(N))$, da das Sortieren dominiert. Im Worst-Case, wenn der Graph sehr dicht ist, erhöht sich die Laufzeit auf $\mathcal{O}(N^2)$, da calculate_barycenter für alle Knotenpaare iterieren muss.

Die Methode calculate_median_x berechnet den Median der x-Koordinaten der Nachbarknoten eines gegebenen Knotens v im Graphen. Dazu sammelt sie die x-Koordinaten aller Zielknoten der ausgehenden Kanten von v in einer Liste, sortiert diese und ermittelt den Median. Ist die Anzahl der Nachbarn ungerade, wird der mittlere Wert ausgegeben; bei einer geraden Anzahl wird der Durchschnitt der beiden mittleren Werte berechnet. Die Laufzeit von calculate_median_x beträgt $\mathcal{O}(d \cdot \log(d))$, wobei d der Grad des Knotens ist, da die Methode die x-Koordinaten sortiert. Die Methode median_crossing_reduction reduziert Kantenkreuzungen durch iterative Neuordnung der Knoten in Schichten basierend auf dem Median der x-Koordinaten ihrer Nachbarknoten. Ähnlich wie bei der Methode barycenter_crossing_reduction werden die Knoten für jede Schicht gesammelt, basierend auf ihrem Median sortiert und neu positioniert, wobei sichergestellt wird, dass keine Überlappungen entstehen. Die Methode median_crossing_reduction weist im Best-Case eine Laufzeit von $\mathcal{O}(N \cdot \log(N))$ auf, da die Sortierung der Knoten innerhalb der Schichten dominiert und nur wenige Kanten vorhanden sind. Im Worst-Case, wenn der Graph sehr dicht ist, steigt die Laufzeit auf $\mathcal{O}(N^2)$, da für jeden Knoten die Berechnung des Medians alle Nachbarknotenpaare betrachten muss. Die tatsächliche Laufzeit hängt somit stark von der Struktur des Graphen ab, insbesondere von der Anzahl der Kanten und der Verteilung der Knoten in den Schichten. Die Methode ist ähnlich effizient wie barycenter_crossing_reduction und eignet sich besser für sparsame Graphen oder Szenarien, in denen Mediane besser die Struktur der Knotenverteilung berücksichtigen.

Die Methode cross_reduction kombiniert die zuvor beschriebenen Ansätze zur Kreuzungsreduktion und wendet sie auf den gegebenen Graphen G an. Zuerst wird für jede der drei Ansätze (Greedy, Barycenter und Median) die Position der Knoten in separaten Arrays kopiert und die jeweilige Reduktionsmethode aufgerufen. Danach wird die Anzahl der Kantenkreuzungen für jede der Positionen berechnet. Die Methode vergleicht dann die Anzahl der Kreuzungen und wählt die Knotenposition mit der geringsten Anzahl an Kreuzungen aus. Diese wird schließlich dem ursprünglichen Knotenpositions-Array zugewiesen.

5.3 Step 4: Vertex Positioning

Die Methode straighten_edges sortiert die Knoten eines Graphen, um eine gleichmäßige Verteilung der Knoten entlang der x-Achse zu erreichen, während

die y-Koordinaten beibehalten werden. Zunächst wird für jede Schicht (bestimmt durch die y-Koordinate der Knoten) eine Liste der Knoten in dieser Schicht erstellt. Diese Knoten werden nach ihren x-Koordinaten sortiert, und anschließend wird jeder Knoten entlang der x-Achse so positioniert, dass die Knoten in jeder Schicht gleichmäßig mit einem Abstand von 50 Einheiten verteilt sind. Nachdem alle Knoten einer Schicht neu positioniert wurden, wird die Methode zur nächsten Schicht fortgesetzt, bis keine Knoten mehr in der aktuellen Schicht vorhanden sind.

5.4 Step 5: Drawing Edges

In diesem Schritt sollen nun Änderungen an den Kantenausrichtungen und der Knoten- und Kantenmenge rückgängig gemacht werden. Hierfür benutzen wir den Tracker, der all die Informationen über eben diese Änderungen abgespeichert hat.
Zum Anfang werden alle Dummyknoten und somit auch die dazu inzidenten Kanten gelöscht in der Funktion removeDummys.
Als Zweites werden alle zuvor gelöschten Kanten wieder hinzugefügt in addremovedEdges. Zuletzt sollen die Kanten, die zuvor umgedreht wurden zur Eliminierung von Kreisen, wieder richtig gedreht werden. Da die LEDA-Library keine Funktion dafür bietet, wird die falsch gedrehte Kante erst gelöscht, um sie dann mit den angepassten Start- und Endknoten wieder einzufügen.