Visualisierung von Graphenalgorithmen

Diplomarbeit

von

Philipp Matthias Hahn

28. März 2002

Technischen Universität Darmstadt
Fachbereich Informatik
Fachgebiet Systemprogrammierung
Betreuer: Dr. Klaus Guntermann
Prüfer: Prof. Dr. Helmut Waldschmidt

Zusammenfassung

Graphenprobleme begegnen einem an vielen Stellen und Orten, für die es eine Vielzahl von verschiedenen Algorithmen und Verfaren gibt, um diese effizient zu lösen. Während des Studiums und auch danach steht man immer wieder vor dem Problem, die Funktionsweise der Algorithmen zu verstehen und diese zu implementieren. Wie schön wäre es da, wenn der Algorithmus bereits fertig vorliegen würde und man sich dessen Funktionsweise an einer geeigneten Animation klarmachen könnte.

Diese Diplomarbeit hat das Ziel, „Algorithmenanimationen“ aus dem Gebiet der „Graphentheorie“ näher zu beleuchten. Sie gliedert sich dabei in die folgenden vier Abschnitte:

Kapitel 1 beschreibt die Wünsche und Ziele dieser Arbeit. Das Kapitel enthält einige grundlegende Definitionen und gibt eine Einführung in die Visualisierung von Algorithmen.

Kapitel 2 nennt bereits existierende Systeme zur Visualisierung und vergleicht diese miteinander.

Kapitel 3 beschreibt die Probleme der vorhandenen Ansätze und versucht diese mit neuen Ideen zu lösen.

Kapitel 4 fasst die Ergebnisse der Arbeit zusammen und gibt einen Ausblick auf weitere Verbesserungen und Erweiterungsmöglichkeiten.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

Kapitel 1

Einführung

Die Klasse der Graphenalgorithmen umfaßt eine Vielzahl von Algorithmen unterschiedlicher Komplexität, nicht nur bezüglich ihrer Laufzeit, sondern auch bezüglich der Implementierung. Die *Visualisierung von Graphenalgorithmen* beschäftigt sich damit, den Ablauf eines Algorithmus darzustellen. Dies ist nicht zu verwechseln mit Plazierungsalgorithmen („Beautifier“, „Layouter“), deren Aufgabe es ist, einen Graphen „möglichst schön“ darzustellen, was auch immer man darunter verstehen will. (planar, wenig Überschneidungen, abstandskonform, ...)

In der Vorlesungen *Grundzüge der Informatik III* [53] und *Graphenalgorithmen* [54] von Professor Waldschmidt werden einige Verfahren aus dem Bereich der Graphentheorie betrachtet. Zunächst werden einige allgemeine Grundlagen geschaffen, bevor dann verschiedene Algorithmen entwickelt und vorgestellt werden. Diese Algorithmen werden in einer Pascal-ähnlichen Sprache notiert, was aber nicht direkt auf einem Rechner ausgeführt werden kann, da einige Vorgehensweisen in einem natürlichsprachlichen Satz anstelle von Befehlen formuliert werden. An Hand eines Beispieles wird dann der Ablauf exemplarisch durchgeführt, um die Funktionsweise zu verdeutlichen.

Die Animation von Graphenalgorithmen ist ein Teil dessen, was als „Softwarevisualisierung“ bezeichnet wird. Diese läßt sich nach [41] in drei Bereiche aufteilen, wie es in Abbildung 1.1 dargestellt ist. Programme als Implementierungen von Algorithmen sind eine Voraussetzung dafür, automatisch aus einem Ablauf Animationen zu erzeugen. Der Datenvisualisierung beruht auf der Erkenntnis, daß Algorithmen oft für sie typische Datenstrukturen verwenden und das durch deren Beobachtung Rückschlüsse auf die Programme bzw. sogar auf die Algorithmen möglich sind.

Das Nachvollziehen des Ablaufes zum genauen Verständnis wird durch mehrere Sachverhalte erschwert:

1. Umfangreichere Algorithmen erstreckten sich oft über mehrere Folienseiten und sind deshalb in ihrer Gesamtheit nur schwer überschaubar.

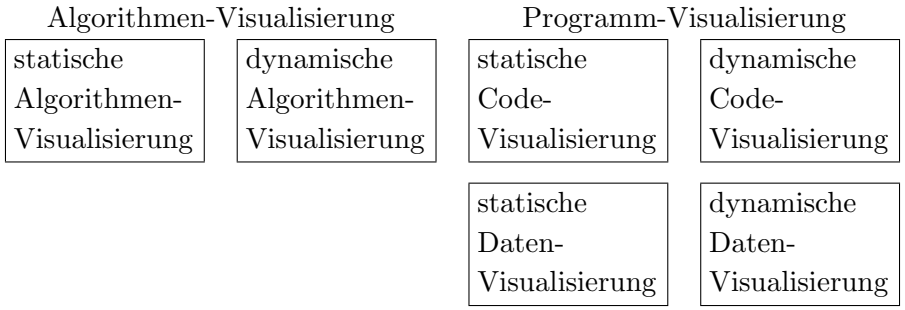

Abbildung 1.1: Teilgebiete der Software Visualisierung

2. Bei größeren Graphen geht sehr schnell die Übersicht darüber verloren, welcher Teil des Graphen bereits abgearbeitet ist und wo sich der Algorithmus im Moment gerade befindet.

3. Algorithmus, Beispielgraph und Trace sind aus Platzmangel nie gleichzeitig auf einer Folie sichtbar, so daß mehrfach zwischen diesen gewechselt werden muß.

Aus diesen und anderen Defiziten ergab sich der Wunsch, die Präsentation zu verbessern. Ausgangspunkt für diese Arbeit ist deshalb der Wunsch nach einem System, das folgende Kriterien berücksichtigt:

- Der Graph, der Programmtext und der Zustand des Algorithmus sollen parallel angezeigt werden können. Je nach Anwendungsfall kann es aber auch genügen, eine Teilmenge davon anzuzeigen.

- Die Visualisierung soll möglichst sowohl für die Erzeugung der Vorlesungsfolien als auch zum privaten Studium einsetzbar sein.

  Für eine Vorlesung ist es nötig, die Beispiele im Vorhinein vorzubereiten, so daß ein reibungsloser Ablauf garantiert ist. Trotzdem muß es das System zulassen, daß an geeigneter Stelle in den Ablauf eingegriffen werden kann, um Zwischenfragen zu klären oder Teile der Vorführung zu überspringen.

  Für ein Privatstudium ist es sinnvoll, selbst Änderungen an den Algorithmen, Datenstrukturen und Graphen vornehmen zu können. Nach Möglichkeit soll das System daher auch von den Studenten zum Eigenstudium auf Rechnern der Universität oder auf eigenen Rechnern eingesetzt werden können.

- Der Algorithmus soll als solcher erkennbar bleiben und nicht durch das Einfügen der Visualisierung überfrachtet und unkenntlich werden.

- Die Darstellung des Algorithmus muß es zulassen, daß einige Programmteile informell beschrieben werden können. Aus Platzgründen kann es sinnvoll sein, nicht auf Implementierungsdetails eingehen zu müssen.

- Die Adaptierung auf weitere Algorithmen soll möglichst einfach sein.

Es existieren bereits eine Vielzahl von unterschiedlichen Systemen, an denen man sofort erkennt, daß sich große Unterschiede im Unfang und bei der Unterstützung ergeben. Eine Suche im Internet führt sehr viele unterschiedliche Vorstellungen zum Thema „Algorithmenanimation“ zu Tage. Ein umfangreiche Aufstellung und Klassifizierung von verschiedenen Animationssysteme wurde zum Beispiel im Zuge einer Diplomarbeit [39] ausführlich erarbeitet.

Am unteren Ende der Skala befinden sich Applets, die genau einen Algorithmus für genau einen ganz speziellen Datenfall animieren. Der Quelltext des Algorithmus wird so gut wie nie angezeigt und der Inhalt der Hilfsdatenstrukturen bleibt verborgen. Die Eingriffsmöglichkeiten sind zu gering, als daß sie sich in einer Vorlesung verwenden lassen könnten.

Das obere Ende gibt es nicht. Stattdessen finden sich sehr viele verschiedenen Systeme, die das eine oder andere Themengebiet mehr oder minder vollständig abdecken. In dem meisten Fällen hat man hier aber das Problem, daß sich diese Systeme nur mit sehr viel Aufwand von Hand anpassen lassen, was sie für den Einsatz in der Lehre ungeeignet bzw. sogar vollständig unbrauchbar macht.

1.1 Animation von Algorithmen

Zur bessen Übersicht wird deshalb an dieser Stelle ein kurzer Überblick zum Erstellen einer Animation gegeben, bevor dann die einzelnen Ziele und Wünsche genannt werdem. Ziel ist es, den Veranstalter einer Lehrveranstaltung bzw. einen Teilnehmer dabei zu unterstützen, sich die Funktionsweise von Graphenalgorithem zu verdeutlichen. Dieser Prozess gliedert sich dabei in mehrere Schritte:

1.1.1 Eingabe des Algorithmus

Zunächst muß der Algorithmus in einer geeigneten Form vorhanden sein. In verschiedenen Vorlesungsunterlagen oder Büchern findet man oft Implementierungen, die sich nicht sofort auführen lassen. Aus Platz- und Übersichtsgründen werden oft einige Anweisungen oder gar ganze Teile ausgelassen bzw. eher informell beschrieben. Für die automatische Generierung einer Visualisierung ist aber eine lauffähige Version nötig. In den meisten Fällen benötigt man dafür verschiedene andere Datenstrukturen, von denen

auch ausführbare Versionen vorhanden sein müssen. Auch diese müssen gegebenenfalls zum besseren Verständnis dargestellt und mit animiert werden, wofür es hoffentlich schon entsprechende Algorithmenbibliotheken inklusive einer Auswahl von Präsentationen gibt.

1.1.2 Eingabe des Graphens

Als nächstes ist ein Graph anzugeben, für den exemplarisch der Ablauf betrachtet werden soll. So vielfältig die Algorithmen sind, so vielfältig sind auch die Anforderungen an den Graphen:

- Ist der Graph gerichtet oder ungerichtet?
- Welche Knoten- und Kantengewichte müssen angegeben werden?
- Muß der Graph als Adjazenzliste, Adjazenzmatrix oder in einer anderen Form vorliegen?
- Wird der Graph durch den Algorithmus verändert oder erweitert?
- Demonstriert der Graph die gewünschten Spezialfälle im Ablauf des Algorithmus?
- Lässt sich der Graph auch für andere Algorithmen verwenden oder adaptieren?
- Soll der Graph automatisch layoutet werden und/oder kann er manuell vom Benutzer plaziert werden?

In Abschnitt 2.4 werden einige Grafikformate vorgestellt, die dem Austausch von Graphen diesen. Für fast alle gibt es entsprechende Editoren, mit denen diese eingegeben und verändert werden können. Einige bieten über ein Plugin-Konzept die Möglichkeit, den Graphen auf Knopfdruck automatisch layouten zu lassen. Diese Möglichkeit ist besonders am Anfang nützlich, um für die Bearbeitung von Hand eine brauchbare Ausgangsbasis zu erhalten. Die Feinpositionierung sollte aber immer dem Benutzer möglich sein, da er schon durch eine geschickte Anordnung der Knoten die weitere Animation stark beeinflussen kann. Die so gewonnenen Positionsangaben aus der Eingabe können unter gewissen Umständen direkt bei der Ausgabe wiederverwendet werden, was den Gesamtaufwand der Erstellung sicherlich reduziert.

1.1.3 Erstellen der Animation

Dies ist der interessanteste und komplexeste Schritt. Aus dem Programmablauf heraus muß eine Visualisierung generiert werden. Wünschenswert wäre

hier, dies weitgehend automatisch zu erledigen. Man wird jedoch nicht darum herumkommen, den Benutzern hier große Freiheiten einzuräumen, da nur diese wissen, auf welche Besonderheiten sie besondern eingehen wollen und welche eher vernachlässigt werden können. Je nach Zielpublikum muß entschieden werden, **welche** Daten- und Hilfsstrukturen **wie** und **wie oft** dargestellt werden sollen.

Zunächst müssen die notwendigen Informationen aus dem Programmablauf gewonnen werden. Nach [13] lassen sich zwei Arten unterscheiden, auf die sich Visualisierungen spezifizieren lassen:

Ereignissgesteuert/Imperativ

Bei diesem Ansatz werden zusätzliche Anweisungen dem Programmtext hinzugefügt. Zunächst müssen relevante Ereignisse im Algorithmus identifiziert werden, die dargestellt werden sollen. In einem zweiten Schritt werden an den entsprechenden Stellen zusätzliche Anweisungen eingefügt, die dann entweder direkt der Darstellung dienen, oder die Darstellung über eine entsprechende Zwischenschicht veranlassen.

Dieser Ansatz bietet den höchsten Freiheitsgrad, da der Programmierer die volle Kontrolle über die Visualisierung besitzt. Es ist ihm überlassen, die entsprechende Anweisungen an den interessanten Stellen unterzubringen. Aktionen können beliebig einfach oder komplex sein und feingranular oder grobgranular eingesetzt werden. Als Ausgangspunkt kann jeder Algorithmus dienen, für den der Quellcode vorliegt.

Ein gravierender Nachteil ist allerdings, daß der Programmtext durch die Annotationen stark erweitert und mit algorithmusfremden Anweisungen überfrachtet wird. Zudem muß der Programmierer den Quellcode genaustens kennen, um an allen wichtigen Stellen entsprechende Aufrufe einfügen zu können.

Eine so erstellte Animation ist lediglich für diesen einen Algorithmus einsetzbar. Eventuelle Änderungen am Algorithmus müssen von Hand in der annotierten Version nachgeführt werden und Varianten des Algorithmus erfordern eine vollständig eigene Annotation.

Zustandsabhängig/Deklarativ

Dieser Ansatz kommt ohne eine Veränderung des Quelltextes aus. Stattdessen läuft der Algorithmus in einer speziell präparierten Umgebung ab, die es ermöglicht, alle Veränderungen in den Datenstrukturen zu beobachten und aufzuzeichnen. Jede Veränderung der Daten triggert dann automatische die Visualisierung, die automatisch aus dem aktuellen Zustand generiert wird. Dazu wird auf deklarative Art und Weise angegeben, **welche** Daten darzustellen sind. Das **wie** der Darstellung kann über bereits vorhandenen parametrisierbare Beschreibungen geschehen, die in speziellen Bibliotheken

abgelegt sind. In den meisten Fällen lassen sich weitere Visualisierungen hinzufügen, die dann auch in anderen Algorithmenvisualisierungen genutzt werden können.

Vorteilhaft an dieser Variante ist sicherlich, daß im Prinzip jede Implementierung eines Algorithmus sofort ohne Änderungen zur Visualisierung geeignet ist. Wichtigste Voraussetzung allerdings ist, daß es eine entsprechende Laufzeitumgebung gibt, die eine beobachtete Ausführung erlaubt.

Nicht zu vernachlässigen ist der Aufwand, der in die deklarative Beschreibung der Animation zu investieren ist. Zwar können aus den Veränderungen von Speicherstellen die entsprechenden Aktionen generiert werden, aber man bewegt sich auf einem sehr niedrigen Niveau. Beispielsweise wird das Vertauschen zweier Element in den meisten Fällen als zyklische Zuweisung dreier Variablen realisiert. Anstatt diese Zuweisungen einzeln anzuzeigen, bedarf es einiger Anstrengungen, dies als eine Aktion ansprechend zu animieren.

Im Ablauf lassen sich zwei unterschiedliche Modi verwirklichen:

Interaktiver Modus

Im interaktiven Betrieb kann der Benutzer während der Laufzeit die Animation anhalten, verändern oder zurücksetzen. Je nach Laufzeitumgebung kann es möglich sein, direkt die Daten oder sogar den Algorithmus selbst während der Laufzeit zu verändern. Dieses Modus ist besonders für Lernende interessant, da das ihnen die Möglichkeit gibt, dem Algorithmus direkt „über die Schulter“ zu schauen und an den Stellen anzuhalten, wo es Probleme gibt. Gerade wenn aus Platzmangel nicht alle Hilfsstrukturen angezeigt werden können, kann der Benutzer hier bei Bedarf alle Objekte von Hand inspizieren.

Batch Modus

Im Batch-Modus hingegen läuft der Algorithmus von alleine vollständig ab und protokolliert interessante Ereignisse und Veränderungen in einer Datei mit. Aus dieser werden dann erst anschließend die Informationen extrahiert und für die Animation verwendet.

Dieser zweite Ansatz hat den Vorteil, daß sich die Visualisierung vollständig vom Ablauf des Algorithmus trennen lässt. Dem Tool zur Visualisierung ist es schlussendlich egal, welcher Algorithmus in welcher Programmiersprache das Protokoll geschrieben hat, solange das Format stimmt. In den gesammelten Daten kann beliebig vorwärts und rückwärts navigiert werden, was den Rückgriff auf vorherige Daten erheblich vereinfacht. Die gleiche Funktionalität in einem interaktiven Betrieb müßte mit Laufzeiteinbußen und zusätzlichem Verwaltungsaufwand aufwendig ergänzt werden, während sie im Batch-Modus von sich aus bereits gegeben ist[10].

1.1.4 Ausgabe der Animation

Der interaktive Modus setzt eine Laufzeitumgebung voraus, die das Beobachten der Datenstrukturen online ermöglicht. Daneben existiert aber der Wunsch, auch andere Ausgabeformate zu unterstützen, etwa um Schnappschüsse für Printmedien zu erzeugen. So mancher Vortragende möchte sich nicht auf die Tücken einer Online-Vorführung einlassen, wenn eine gleichwertige Animation genausogut im Vorhinein in Ruhe vorbereitet werden kann.

Allerdings wird dadurch der Gestaltungsspielraum stark eingeschränkt: Während im Online-Betrieb kleine Animationen benutzt werden können, um die Aufmerksamkeit des Betrachters auf den interessanten Teil der Animation zu lenken, muß man sich sonst auf diskrete Schnappschüsse beschränken, die mehr oder minder weit auseinanderliegen.

Als Beispiel sei hier ein so einfacher Algorithmus wie der Bubble-Sort-Algorithmus genannt. In einer Online-Variante lassen die beiden zu vergleichenden bzw. zu vertauschenden Elemente jeweils kurz durch aufblinken hervorheben, bevor sie in einer flüssigen Animation vertauscht würden. In der Offline-Version müsste man sich mit der Hervorhebung vor dem Vergleich und einer Darstellung nach dem Vergleich begnügen.

Neben der zeitlichen Komponente lassen sich je nach Medium zusätzlich Farbe, Schattierung, Strichdicke und Linienmuster bei der Darstellung von Graphen verwenden. Daneben kann es sinnvoll sein, Hilfsdatenstrukturen wie Warteschlangen, Stacks und andere lokale Variablen anzuzeigen. Bei rekursiven Funktion ist insbesondere auch der Aufrufstapel interessant. Dieser muß nicht immer als Stack visualisiert werden, sondern kann zum Beispiel bei Graphenalgorithmen auch implizit als Kantenfolge hervorgehoben werden.

Die Ausgabe sollte nicht zu viel und nicht zu wenig Informationen enthalten. Bei überfrachten Darstellungen verliert der Betrachter leicht das wesentliche aus den Augen und kann schlecht die Zusammenhänge zwischen den einzelnen Teilen der Darstellung herstellen. Umgekehrt führen mangelnde Informationen dazu, daß der Betrachter schnell die Konzentration und die Lust verliert, da er die ihm fehlenden Daten erst mühsam von Hand suchen muß.

Die folgende Aufzählung enthält einige Objekte, die bei einer Visualisierung darzustellen sein könnten:

- Der Algorithmus als Programmtext
- Der Graph, wie er als Eingabe für den Algorithmus dient
- Weitere Hilfsgraphen bzw. vom Eingangsgraphen abgeleitete Graphen. (Beispielsweise ist bei vielen Flußalgorithmen neben dem Eingangsgraphen der Restgraph von großem Interesse, so daß dieser getrennt vom Ursprungsgraphen dargestellt werden sollte.)

- Hilfsstrukturen wie Warteschlangen, Stacks, Mengen, lokale Variablen
- Tracetabelle der „wichtigen“ Variablen

1.2 Kategorien von Algorithmen

Graphenalgorithmen treten in zwei Ausprägungen auf, die im folgenden unterschieden werden sollen.

1.2.1 Statische Graphenalgorithmen

In diese Kategorie fallen Algorithmen, die auf einem gegebenen Graphen arbeiten und Informationen wie „kürzeste Wege“, „mehrfachen Zusammenhang“ oder „Planarität“ berechnen. Für diese Klassen von Algorithmen ist das Problem der Darstellung darauf zu reduzieren, daß das Layout einmalig festgelegt wird und der Algorithmus den Graphen danach nicht ändert.

1.2.2 Dynamische Graphenalgorithmen

In die zweite Klasse fallen Algorithmen, die eine Graphenstruktur transformieren und als Ergebnis ein oder mehrere Graphen ausgeben, die sich nicht als Subgraphen des Ausgangsgraphen darstellen lassen. Eines der Hauptprobleme, die sich für diese Klasse ergibt, ist die sich ändernde Plazierung von Kanten und Knoten. Für jeden darzustellenden Graphen sind dann alle Knoten- und Kantenpositionen festzulegen. Es ist deshalb sicherlich wünschenswert, einen Großteil des Layouts automatisch zu erledigen und nur im Einzelfall selber Hand anlegen zu müssen.

In diese Gruppe gehören Algorithmen zur Konstruktion von „minimalen Suchbäumen“ bzw. „Änderungen in Suchbäumen“. Da diese Datenstrukturen oftmals als Hilfsstrukturen in anderen Algorithmen vorkommen, kann schlecht auf sie verzichtet werden.

1.3 Graphenrepräsentation

Graphen können auf unterschiedliche Art und Weise repräsentiert werden. Adjazenzlisten oder Adjazentmatrizen sind die klassischen Formen, aber bereits bei diesen gibt es viele Variationen. Problematisch ist die in der Hinsicht, daß den Algorithmen der Graph meist in einer ganz speziellen Form übergeben werden muß. In der einfachsten Form genügt es, alle Kanten und Knoten aufzuzählen und diese in Form von Listen als Parameter mitzugeben oder global zur Verfügung zustellen.

Dabei kommen aber schnell weitere Fragen auf, die im vorhinein zu klären sind:

- Werden ungerichtete und gerichtete Kanten unterstützt?
- Können ungerichtete und gerichtete Kanten gleichzeitig genutzt werden?
- Werden Mehrfachkanten und hierarchische Graphen[1] unterstützt?
- Wie werden Kanten und Knoten eindeutig identifiziert? Welche Namenskonventionen sind einzuhalten?
- Wie werden Kanten- und Knotengewichte spezifiziert? Wie können andere Attributierungen gemacht werden?
- Lassen sich Graphen verändern und erweitern?

Abschnitt 2.3 beschreibt einige der geläufigen Formate zum Austausch von Graphen. Eine Umgebung zur Graphenvisualisierung sollte die wichtigsten davon einlesen und verarbeiten können, um auf bereits existierende Beispiele und Programme zurückgreifen zu können.

1.4 Ziel dieser Arbeit

Ziel dieser Arbeit soll es sein, verschiedene bereits existierende Ansätze miteinander zu vergleichen, deren Stärken und Schwächen aufzuzeigen, um daraus Verbesserungen abzuleiten. Weiter soll untersucht werden, wie sich solche Animationen mit Standardtools erzeugen und in andere Dokumente integrieren lassen. Soweit möglich, soll ein Prototyp entwickelt werden, um daran die Machbarkeit der Ansätze zu demonstrieren.

[1] Graphen, die anderen Graphen als Knoten haben

Kapitel 2

Existierende Arbeiten

In zahlreichen Arbeiten wurden bereits viele verschiedene Aspekte der Softwarevisualisierung angegangen. Eine Suche in der gängigen Literatur und im Internet fördert eine Fülle von Artikeln und Produkten zu Tage, die unmöglich alle gesichtet und untersucht werden können[41, 48, 39]. Einige wenige sollen im folgenden erwähnt werden, um diese auf die Verwendbarkeit für die Visualisierung von Graphenalgorithmen zu untersuchen. Dies geschieht einerseits im Hinblick darauf, nicht alle Räder erneut erfinden zu müssen, andererseits um bereits vorhandene Komponenten und Ideen auf ihre Weiterverwendbarkeit und Einschränkungen zu untersuchen.

2.1 Animationssysteme

Animationssysteme sind in der Regel recht allgemein gehalten und nicht auf die Animation von Graphenalgorithmen spezialisiert. Einerseits bietet diese Universalität vielfältige Möglichkeiten zur Animation, andererseits führt der Mangel an ansprechend vorgefertigten Animationen dazu, daß Präsentationen auf sehr niedrigem Niveau ohne spezielle Unterstützung für häufig vorkommende Datenstrukturen erstellt werden müssen.

2.1.1 Animal

Algorithm Animation (Animal) [44] entstand an der *Universität Siegen* und ist ein System zum Erstellen von Animationen. Zugrunde liegt ein Dateiformat, daß die Beschreibung von geometrischen Objekten und einigen komplexeren Datenstrukturen wie Felder und Listen erlaubt. Für diese können neben der Position weitere Darstellungsmerkmale wie Farben und Linienarten angegeben werden, die sich skriptgesteuert verändern lassen, wodurch sich einfache bis komplexe Animationen erzeugen lassen.

Abgespielt werden können die Animationen durch ein *Java*-Programm, das gleichzeitig auch zum Erstellen und Bearbeiten derselben dient. Die ge-

samte Animation wird von Hand erstellt und kann nicht automatisch aus dem Ablauf eines Programms gewonnen werden. Deshalb müssen der Programmtext und Hervorhebungen darin ebenfalls von Hand in die Beschreibung mit aufgenommen werden.

Animal bietet sich zur Darstellung von Animationen an, verfügt selbst aber über keine Möglichkeit, diese automatisch aus der Beobachtung eines Algorithmenablaufs zu erstellen.

2.1.2 JAnim

Algorithm Animation Using Java (JAnim) [3] entstand als Diplomarbeit an der *Universität Osnabrück* und bietet ein steuerbares Skriptsystem, mit dem Animationen in verschiedenen Geschwindigkeiten vorwärts und rückwärts abgespielt werden können. Außerdem bietet es die Möglichkeit, Breakpoints auf verschiedene Klassen und Objekte zu setzen.

Leider wird das System nicht mehr weiterentwickelt und ist inzwischen auch nicht mehr über das Internet zu beziehen.

2.1.3 JAWAA

Java and Web based Algorithm Animation (JAWAA) [40] ist von der *Duke University in North Carolina* und bietet eine einfache Skriptsprache zur Visualisierung von Datenstrukturen. Neben Stacks und Queues wird auch direkt die Darstellung von Graphen unterstützt. Das *Java*-Applet ist selbstablaufend und kann lediglich angehalten werden.

Das Programm arbeitet lediglich ein Skript ab, das die Animation vollständig beschreiben muß. Verzögerungen sind direkt im Skript angegeben und können zur Verlangsamung oder Beschleunigung zur Laufzeit interaktiv skaliert werden.

Die Sprache unterstützt neben einfachen geometrischen Objekten und Texten auch Bäume, Graphen, Warteschlangen und Stacks, die hinzugefügt, gelöscht und bewegt werden können. Jedem Objekt oder einen Gruppe von Objekten können verschiedene Parameter wie Vorder- und Hintergrundfarbe zugewiesen werden.

2.1.4 Samba/Polka/XTango

Polka [49] von *Georgia Tech* ist ein offline System und kann lediglich Skripte abarbeiten, die zuvor erstellt wurden. Es handelt sich um eine Reimplementierung des älteren *XTango*, das sehr stark auf *UNIX* und dessen *X11*-Bedienoberfläche zugeschnitten war. *Samba* ist das dazu passende interaktive Frontend dafür, wovon es auch ein in *Java* geschriebenes Applet namens *JSamba* gibt. Das Gesamtsystem basiert darauf, daß die in *C* bzw.

C++ geschriebenen Algorithmen um zusätzliche Befehle erweitert werden, um wichtige Ereignisse an die Darstellungsebene zu melden.

Samba ist nicht auf Graphen spezialisiert, sondern arbeitet lediglich mit geometrischen Objekten wie Dreiecken, Rechtecken, Linien, Polygonen und Texten. Diese können bewegt und mit Darstellungsattributen versehen werden, um von Hand den Ablauf zu animieren. Gegenüber *JAWAA* bietet es den Vorteil, mehrere Sichten auf mehrere Fenster verteilen zu können, auch wenn jedes für sich vollständig von Hand beschrieben werden muß. Der Ablauf kann im interaktiven Betrieb angehalten werden und die Darstellung über einen Schieberegler verlangsamt bzw. beschleunigt werden.

2.1.5 GraphTool

GraphTool [33] von der *University of California in Irvine* ist ein Programm, das der Eingabe und Manipulation von Graphen dient. Programme können das in *C* programmierte GraphTool dazu nutzen, dynamische Veränderungen in Graphen zu visualisieren und mit dem Benutzer zu interagieren. Dazu startet *GraphTool* das Programm in einem eigenen Prozeß und stellt diesem über die Standardeingabe die Graphenstruktur sowie interaktive Ereignisse zur Verfügung, während die Standardausgabe des Programms den Graphen manipuliert und der Interaktion mit dem Benutzer dient.

Da *GraphTool* auf Graphen spezialisiert ist, können auch nur diese dargestellt werden. Eine gleichzeitige Darstellung des Programmtextes oder anderer Datenstrukturen ist nicht vorgesehen, könnten aber „von Hand“ aus Linien, Rechtecken und Texten zusammengesetzt werden.

2.1.6 Goblin

Von der *Universität Augsburg* wurde die *Library for Graph Matching and Network Programming Problems* (GOBLIN) geschrieben, die eine komplett in *C++* und *Tcl/Tk* geschrieben Algorithmensammlung mit Visualisierungsmöglichkeit umfasst. Neben den grundlegenden Strukturen für die Verwaltung von Graphen bietet es auch erweiterte Strukturen für Perfekte Graphen [1, 2, 54, 20], wie zum Beispiel Dreiecksgraphen, Vergleichbarkeitsgraphen und Splitgraphen. Mitgeliefert werden bereits fertige Algorithmen für das Kürzeste-Wege-Problem, Minimal-aufspannende-Bäume, Zusammenhangskomponenten, Flußprobleme sowie einige verschiedene andere Graphenprobleme. Mit dem eingebauten Editor lassen sich Graphen eingeben und verändern und er dient gleichzeitig dazu, Schnappschüsse des Ablaufs darzustellen. Diese werden durch Annotationen in den Algorithmen erstellt und in einer LISP-ähnlichen Syntax in einer Datei gespeichert.

Die Bibliothek ist bereits sehr umfangreich und enthält etliche Algorithmen, aber diese enthalten viele zusätzliche Anweisungen zur Konsistenzkontrolle und Visualisierung. Algorithmen laufen immer vollständig ab, so

daß nur eine Visualisierung der Schnappschüsse nach dem Programmende möglich ist. Bei der Anzeige ist jeweil immer nur eine Datenstruktur zu sehen, so daß die Anzeige ständig bei Veränderungen der Hilfsdatenstrukturen zwischen diesen und dem eigentlichen Graphen umschaltet, was äußerst störend wirkt. Während der Visualisierung wird der Programmtext auch nicht mit angezeigt, so daß nie eindeutig klar wird, an welcher Stelle sich der Algorithmus befindet.

2.1.7 Zeus

Zeus entstand am *Digital's System Research Center* und ist in *Modula-3* geschrieben. Es ist allgemein gehalten und kann dazu genutzt werden, die verschiedensten Algorithmen zu animieren, unter anderem auch solche, die auf Graphen arbeiten. Dazu werden einige grundlegende Bibliotheken mitgeliefert, die solch grundlegende Dinge wie abstrakte Klassen für Graphen, Kanten und Knoten enthalten. Auf diesen aufbauend arbeiten dann die Algorithmen, die interessante Ereignisse durch Aufrufe von entsprechenden Methoden signalisieren. Allerdings ist der Programmierer selbst dafür verantwortlich, die entsprechenden Aufrufe von Hand einfließen zu lassen, ein automatischer Aufruf aufgrund von Veränderungen in den Daten selbst ist nicht vorgesehen.

Zeus selbst bietet bereits eine Möglichkeit dazu, während des Ablaufs einen Programmtext parallel zur Ausführung anzuzeigen. Dabei handelt es sich jedoch nicht um den eigentlichen Programmtext des Algorithmus, sondern jede beliebige Textdatei kann dazu verwendet werden. In diese werden spezielle Tags eingebettet, die dann als Referenzmarken bei der Anzeige dienen, die die Anzeige mit der Ausführung zu synchronisieren.

2.1.8 DDD

Der *Data Display Debugger* (DDD) [56] ist ein grafisches Frontend für die Kommandozeilendebugger *gdb*, *pydb*, *jdb* und andere. Dadurch können *C*-, *C++*-, *Python*-, *Java*- und *Perl*-Programme mit einer einheitlichen Bedienoberfläche untersucht werden. Debugger bieten viele Fähigkeiten, die eng mit der Animation von Datenstrukturen verwandt sind. Sie gestatten das Beobachten von Variablen, ermöglichen das Anhalten der Ausführung bei bestimmten Konditionen und zeigen Parallel zur Ausführung den Quelltext des Programms an.

DDD zeichnet sich gegenüber anderen Debuggern, die lediglich Unterbrechungspunkte, Einzelschrittmodus und textuelle Ausgbe von Variablen unterstützen, durch seine grafische Anzeige aus. In dieser können Datenstrukturen strukturiert angezeigt werden, was insbesondere auch bei verschachtelten und rekursiven Objekten der Fall ist. Diese können teilweise auch automatisch angeordnet werden und werden bei jeder Programmun-

terbrechung halbautomatisch aktualisiert. Die Einstellungen zur Darstellung können persistent auf der Festplatte abgespeichert werden, so daß diese reaktiviert werden können. Abbildung 2.1 zeigt einen Bildschirmschappschuss einer solchen Sitzung, in der ein einfaches *C*-Programm mit einer verketteten Liste untersucht wird.

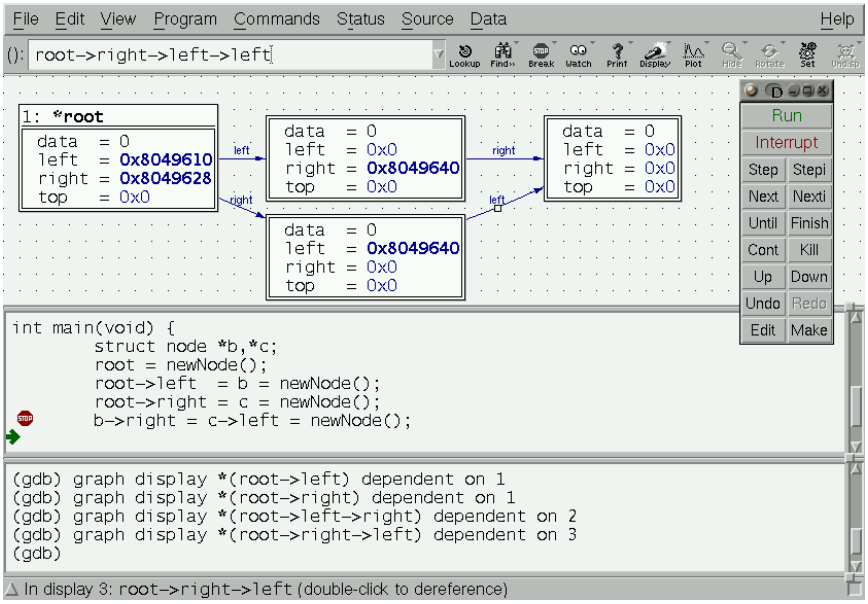

Abbildung 2.1: Data Display Debugger

Die Unterstützung der einzelnen Programmiersprachen ist jeweils nur so gut, wie die des zugrundeliegenden Debuggers. So funktioniert zum Beispiel die Anzeige von verschränkten Strukturen nicht mit dem *Python*-Debugger. Die Darstellung komplexerer Strukturen wird schnell sehr unübersichtlich, da *DDD* immer alle Werte einer Struktur anzeigt. Dynamisch erzeugte Objekte müssen von Hand explizit angezeigt und plaziert werden.

2.2 Algorithmensammlungen

Graphen sind sehr grundlegende Strukturen und werden deshalb von vielen Programmen benutzt, um weiterführende Probleme zu lösen. Deshalb existieren bereits einige Programmbibliotheken, die viele Graphenalgorithmen mit ihren zugehörigen Datenstrukturen enthalten. Im Folgenden werden einige Bibliotheken aufgeführt und auf ihrer Verwendbarkeit in eigenen Visualisierungen untersucht.

2.2.1 LEDA

Die *Library of Efficient Data Types and Algorithms* (LEDA) [36] ist eine Bibliothek von verschiedennen Graphenalgorithmen, die unter der Federführung von Prof. Melhorn an der *Universität Saarbrücken* entstanden ist. *LEDA* ist in *C++* mit Templates implementiert und war für den akademischen Bereich lange Zeit frei verfügbar, jedoch sind inzwischen für die Verwendung Lizenzgebühren fällig. Ziel der Implementierung ist es, eine Vielzahl von Algorithmen zur Verfügung zu stellen, um eine lauffähige Version zu haben und diese miteinander vergleichen zu können.

Für einige Algorithmen existieren bereits animierte Versionen, um deren Wirkungsweise und Programmablauf zu verdeutlichen. Diese fügen sehr umfangreiche Änderungen in den Quelltext ein, so daß die eigentliche Struktur des Algorithmus vollkommen verloren geht. Dargestellt werden dann auch lediglich die Veränderungen auf den Datenstrukturen, eine gleichzeitige Darstellung des Programmtextes ist nicht einfach möglich. In einem weiteren Projekt [37] wird derzeit versucht, die Visualisierungen durch Verwendung der *C++*-Templatemechanismen so weit wie möglich aus den Programmtexten zu verbannen.

2.2.2 Graphlet: GTL/GraphScript/GML

Die *Graph Template Library* (GTL) [5] der *Universität Passau* stellt eine *C++* Bibliothek zur Verfügung, die sich stark an die STL[1] anlehnt. Sie stellt als Grundlage einige Klassen zur Darstellung von Graphen und darüber hinaus auch einige Algorithmen bereit, darunter unter anderem Algorithmen zur Bestimmung des maximalen Flusses und des zweifachen Zusammenhangs. Für die Nutzung mit *Java* werden bereits Wrapper-Klassen angeboten, die auf die native *C++* Implementierung zurückgreifen. Im Gegensatz zu *LEDA* ist *GTL* für den universitären Einsatz frei von Lizenzgebühren verfügbar.

GraphScript erweitert die *GTL* um einen in *Tcl/Tk* programmierten Editor, der zur Eingabe und Manipulation von Graphen verwendet werden kann. Verschiedene Plazierungsalgorithmen sind in den Editor integriert, um Graphen zu layouten. Graphen können in der *Graph Modelling Language (GML)* gespeichert und geladen werden, wodurch ein Austausch mit anderen Programmen möglich ist. Wie bei *LEDA* auch ist eine gleichzeitige Anzeige des Quelltextes nicht realisiert.

2.2.3 Boost: GGCL/BGL

Die *Generic Graph Component Library* (GGCL) [47] ist eine *C++* Bibliothek, die ursprünglich von der *University of Notre Dame* stammt. Inzwi-

[1] Standard Template Library

schen ist sie jedoch unter dem Namen *Boost Graph Library* im *C++ Boost* Projekt aufgegangen, das eine Vielzahl von verschiedenen Algorithmen und Datenstrukturen als *C++* Template-Bibliotheken zur Verfügung stellt. Diese Bibliothek implementiert zwar auch etliche Graphenalgorithmen, klammert aber den Bereich der Visualisierung vollständig aus.

Durch die extensive Nutzung von Templates und Adaptoren ist es möglich, die in *BGL* implementierten Algorithmen auf Objekte von Altanwendungen anzuwenden, ohne daß diese speziell für die Nutzung der *BGL* angepasst werden müssen. Die Allgemeinheit der Implementierung ist jedoch für das Studieren der Algorithmen weniger geeignet, da sie von den wesentlichen Ideen der Algorithmen zu sehr ablenken.

2.2.4 CGAL

Die *Computational Geometry-Algorithms Library* (CGAL) [16] war eine für den universitären Berech frei verfügbare Sammlung von Algorithmen. Inzwischen ist jedoch eine Registrierung erforderlich, auch wenn diese für den akademischen Bereich weiterhin frei erhältlich ist. Hauptgebiet ist die Verarbeitung von 2D- und 3D-Vektordaten, weniger die allgemeine Verarbeitung von Graphen als abstrakte Strukturen. Die Bibliothek ist in *C++* implementiert und nutzt dessen Template-Mechanismus für anwendungsspezifische Anpassungen.

2.2.5 GFC

Von *IBM* stammen die *Graph Foundation Classes for Java* (GFC) [8], die ein *Java*-Framework für die Verarbeitung und Darstellung von Graphen bilden.

Auf der untersten Ebene stehen die Basis-Klassen *GFC* zur Beschreibung von mehrdimensionale Graphen und Hypergraphen zur Verfügung. Durch die Benutzung zweier Dictionaries ist es möglich, Nutzdaten und Daten der Algorithmen jedem Objekt zuzuordnen, ohne daß dazu eigene Klassen abzuleiten wären.

Darauf aufbauend arbeitet das *Graph Drawing Framework* (GDF), welches zur Visualisierung von Kanten und Knoten dient. Mit verschiedenen Plugins kann die Darstellung individuell angepasst werden.

Das *Graph Layout Framework* (GLF) schließlich stellt verschiedenen Layout-Algorithmen bereit, um Kanten und Knoten automatisch anordnen zu lassen.

Die drei Packete stellen lediglich ein allgemeines Framework zur Berarbeitung und Darstellung von Graphen bereit, zur Zeit existieren keine Implementierungen von Algorithmen, die diese Frameworks verwenden.

2.3 Graphenrepräsentation

Es gibt sehr unterschiedliche Formate, um Graphenstrukturen zwischen Programmen auszutauschen. Zwar existieren einige Bemühungen, sich auf ein ein standardisiertes Format zu einigen, aber viele der zuvor genannten Algorithmensammlungen verwenden noch ihre eigenen Formate, die im folgenden exemplarisch beschrieben werden. Durch die Verwendung bereits existierender Formate kann man auf bereits vorhanderen Programmschnittstellen und Editoren zurückgreifen und erspart sich das Programmieren eigener Routinen zum Import und Export.

2.3.1 GML

Die *Graph Modeling Language* (GML) [27] wird von *Graphlet* benutzt und kann statische Graphen speichern. Dieses Format ist textbasiert und beinhaltet neben der Struktur des Graphen zusätzliche Annotationen, die von verschiedenen Programmen bearbeitet werden können. Maximal können drei Dimensionen zur Plazierung der Graphen in den Koordinatenangaben benutzt werden und dynamische Veränderungen werden vom Format nicht weiter unterstützt.

Da dieses Format vor allem von verschiedenen andere Programme genutzen wird, um untereinander Graphen auszutauschen, führt es eine interessante Klassifizierung der Nutzerdaten in zwei Kategorien ein: „Sichere“ Daten bleiben selbst bei Änderung des Graphens erhalten, während „unsichere“ Daten bei Veränderungen durch andere Programme, die diese Daten nicht verarbeiten können, entfernt werden.

2.3.2 XGMML

Die *eXtensible Graph Markup and Modeling Language* (XGMML) [42] ist eine 1:1-Übersetzung der *GML* in *XML* und bietet daher auch alle Möglichkeiten von *GML*. Theoretisch läßt sich das Format durch die Benutzung von *XM* Namensräumen beliebig erweitern, um fehlende Möglichkeiten nachträglich zu ergänzen, allerdings ist unklar, wie andere Programme mit solchen Sprachdialekten umgehen.

2.3.3 GraphXML

Graph XML [26] kann im Gegensatz zu den übrigen Formaten auch dynamische Graphen beschreiben, wenn auch stark eingeschränkt. Das Format baut zwar auf *XML* auf, ist aber nicht modular und beschränkt sich lediglich auf die Positionierung von Graphen im zwei- bzw. dreidimensionalen Raum. Den grafischen Objekten „Knoten“ und „Kanten“ können Position, Größe, Linien- und Füllfarben bzw. -Muster zugeordnet werden. Zusätzlich können

beliebige Nutzerdaten in die Struktur integriert bzw. extern referenziert werden. Dynamische Veränderungen werden durch das Ersetzen und Entfernen von Teilhierarchien des Graphen erreicht. Die Beschreibung von kontinuierlichen Bewegungen und Veränderungen ist damit nur sehr arbeitsintensiv in Handarbeit möglich.

2.3.4 GXL

Die *Graph eXchange Language* (GXL) [29] ist eine *XML*-Basierte Sprache zum Beschreiben statischer Graphen. Neben den Standardtypen „Knoten“ und „Kanten“ existiert noch die „Relation“, die im Gegensatz zur binären Kante eine mehrdimensionale Verknüpfung von Objekten darstellt. Unterstützt werden auch Hypergraphen und verschachtele Graphen. Jedem Objekt können Attribute zugeordnet werden, durch die der Graph annotiert werden kann.

2.3.5 GraphML

Die *Graph Markup Language* (GraphML) [6] ist eine Initiative von *Graph-Drawing.org* mit dem Ziel, ein einheitliches Format für den Austausch von Graphen zu definieren. *GraphML* basiert auf *GraphXML* und *GXL* und ist ein weiteres *XML*-basierendes Format. Es ist darauf ausgelegt, Graphen zwischen verschiedenen Anwendungen austauschen zu können, ohne daß durch Konvertierungsprozesse Informationen verloren gehen. Unterstützt werden neben gemischt gerichteten und ungerichteten Graphen auch Hypergraphen[2] und verschachtele Graphen[3].

Die Sprachdefinition klammert Visualisierung und Dynamik gezielt aus. *GraphML* sieht diese als eine modulare Erweiterungen der Basisdefinition vor. Als Beispielanwendung wird bereits ein einfacher Editor als Konzeptbeweis mitgeliefert.

2.4 Eingabe/Ausgabe

Zur Eingabe der Graphen werden Editoren benötigt, um die eben vorgestellten Formate zu erzeugen. Zwar handelt es sich in den meisten Fällen um Textdateien, die sich prinzipiell mit jedem Texteditor bearbeiten lassen, aber diese Art und Weise ist sehr unkomfortabel und fehleranfällig. Da solche Editoren die Graphen sowieso schon grafisch anzeigen, bietet es sich an, diese auch zur Ausgabe und zur Anzeigen von Zwischenschritten während dem Programmablauf einzusetzen. Für die Weiterverwendung der Darstel-

[2] Kanten können mehr als zwei Knoten miteinander verbinden

[3] Hierarchische Graphen enthalten als Knoten wiederum Graphen

lung in anderen Programmen muß eine Möglichkeit bestehen, die Daten entsprechend zu exportieren.

2.4.1 Texteingabe

Für kleine und einfache Beispiele ist sicherlich noch die Eingabe des Graphen mit Hilfe eines Texteditors von Hand möglich, aber bereits die Plazierung der Knoten und Kanten gestaltet sich ohne direkte Visualisierung schwierig. Selbst mit der Formatbeschreibung durch *DTD*s bzw. *XML-Schema*[17] und der Verwendung geeigneter *XML*-Editoren ist die Eingabe sehr fehleranfällig und mühsam. Da Textformate im Gegensatz zu Binärformaten leicher zu erzeugen und zu bearbeiten sind, sollte als Austauschformat auf jeden Fall ein Textformat gewählt werden.

2.4.2 XFIG/jFIG

In der UNIX-Welt ist das in *C* programmierte vektororientierte Zeichenprogramm *XFIG* [32] bzw. dessen *Java*-Implementierung *jFIG* [25] weit verbreitet. Da insbesondere zum Zeitpunkt dieser Arbeit die bisherigen Lösungen auf dieses Programm zurückgegriffen haben, soll es hier erwähnt werden.

Das größte Problem bei einer Nutzung stellt sich folgendermaßen dar: *XFIG* selbst ist ein reines Zeichenprogramm und bietet daher keine direkte Unterstützung zum Zeichnen von Graphen. Knoten müssten als Kreisen und Rechtecken dargestellt werden, während Kanten als einfache Linien zwischen diesen verwandt werden. Erst durch eine geeignete Nachbearbeitung müssten aus der geometrischen Nähe der Linienendpunkte zu den Knoten die entsprechende *XML*-Struktur aufgebaut werden. Eine Attributierung von Knoten und Kanten wäre nur über ähnliche Umwege machbar und scheint sehr fehleranfällig zu sein. Insbesondere bei Verschiebungen müssten inzidente Kanten von Hand nachgeführt werden, da sich die von *XFIG* gebotenen automatischen Mittel dazu nicht in allen Fällen nutzen lassen.

Das verwendete *XML*-Format ist zwar für die Weiterverarbeitung in andere Formate sehr gut geeignet, als Endanwenderformat aber nur sehr eingeschränkt. Demhingegen ist das von *XFIG* verwendete Format weit verbreitet und eignet sich deshalb als eines der Darstellungsformate. Insbesondere bietet dieses die Möglichkeit, die erzeugten Grafiken direkt weiterzuverarbeiten, bevor sie dann über *transfig* bzw. *fig2dev* in ein abschließendes Format umgewandelt werden. Durch die Formatvielfalt von *fig2dev* erhält man so sehr einfach eine Vielfalt weiterer Ausgabeformate, die eine Integration in bereits bestehende Programme enorm vereinfacht. Die bereits vorhandene *Java*-Implementierung von *jFIG* erleichtert weiter das Einbinden in ein in *Java* geschriebenes Visualisierungssystem, weshalb sich das *XFIG*-Format als Ausgabeformat stark anbietet.

2.4.3 SVG

Scalable Vector Graphics (SVG) [34] ist ein vom W3C vorgeschlagenes Format zur Darstellung von Zeichnungen und Bildern im zweidimensionalen Raum. Es ist kein Format zum Austausch von Graphenstrukturen, sondern vielmehr ein Format zum Anzeigen und Darstellen von geometrischen Objekten. Neben den primitiven Grundformen werden aufwendige Filterungen und Animationen unterstützt.

Scalable Vector Graphics ist ein relativ junges Format für die Übertragung von vektoriellen Grafiken im Internet, das inzwischen eine offizielle Empfehlung des W3C ist. Zur Zeit existieren wenige brachbare Implementierungen des Formats: Von *Adobe* gibt es den *SVG Viewer* für verschiedene Rechnerplattformen, der sich als Plugin in die gängigsten WWW-Browser integrieren läßt. *Mozilla* enthält erste Ansätze einer Implementierung, so daß (X)HTML[4], MathML[5] und *SVG* sich nahtlos in einem *XML*-Dokument nutzen lassen. Von der *Apache Foundation* gibt es das *Batik*-Projekt [24], das vollständig in *Java* implementiert ist und momentan alle statischen Funktionen von SVG unterstützt. *Batik* wird unter anderem auch in *Formatting Objects* (FOP) [51] genutzt, um aus einer *XML*-Spezifikation PDF-Dateien zu generieren, was die Aufbereitung als Folien und gedruckte Skripte ermöglicht. Weiterhin existieren verschiedene andere Implementierungen in *C* und *Java*, die das leichte Einbinden in eigene Programme und das Erzeugen von Rasterbildern im GIF-, PNG-, TIFF- und JPEG-Format ermöglichen.

Neben der Möglichkeit zur Beschreiben von statischen Bildern bietet *SVG* auch die Möglichkeit der Animation und Interaktion mit dem Betrachter. Letzteres wird aber zum Zeitpunkt dieser Arbeit nur in der Implementierung von Adobe unterstützt, in *Batik* gibt es erste Ansätze dazu. Aus Mangel an vollständigen Viewern sollte sie Ausgabe von SVG deshalb zur Zeit noch auf statische Darstellungen beschränkt werden.

2.4.4 PDF

Das Portable Document Format eignet sich als Format für statische Bilder, da es sich als Standard für den Austausch von druckbaren Dokumenten im Internet etabliert hat. Da jedoch die beiden anderen Exportformate XFIG und *SVG* jeweils eine Konvertierungsmöglichkeit nach PDF anbieten, kann auf eine direkte Ausgabe von PDF verzichtet. Dynamik innerhalb von PDF ist nur auf aufwendigen Umwegen erreichbar und scheidet deshalb für solche Zwecke aus.

[4] *XML*-Variante der Hyper-Text Markup Language
[5] *XML*-Sprache zur Beschreibung mathematischer Zusammenhänge

2.4.5 Eigener Editor/Viewer

Aus der bereits oben erwähnten *Java*-Implementierung *jFIG* läßt sich auch ein eigener Editor entwickeln, der bessen dem geforderten Profil enspricht. Aus dem *OpenJGraph* [31] Projekt gibt es bereits einen einfachen Editor zur Eingabe und Manipulation von Graphen, der um die fehlene Funktionalität zur Annotation von Knoten und Kanten erweitert werden kann. Viele andere Projekte [18, 12, 46] haben ihre eigenen Editoren, die sehr stark in das Gesamtprogramm eingebunden sind und nicht ohne weiteres weiterverwenden lassen.

2.5 Graphenvisualisierung

Es existieren bereits einige Ansätze, die sich speziell mit der Visualisierung von Graphen beschäftigen. Deshalb war der Ausgangspunkt für diese Arbeit zunächst einmal, die bereits vorhandenen Lösungen auf ihre Verwirklichung und Nutzbarkeit zu untersuchen.

2.5.1 Y-Files

Y-Files [55] von der *Universität Tübingen* ist ein in *Java* geschriebenes Applet, daß ein paar ausgewählte Graphenalgorithmen animiert. Die Algorithmen sind in *Java* geschrieben und greifen auf vorgefertigte Datenstrukturen zurück, um die Funktionalität zu implementieren. Durch Annotationen im Programm werden die Visualisierungen von Hand getriggert. Zusätzlich wird die Möglichkeit geboten, während des Programmablaufs einen Programmtext anzeigen zu lassen. Dabei muß es sich aber nicht um den Quelltext des ausgeführten Programms handeln, vielmehr wird in den meisten Fällen ein Text angezeigt werden, der zwar schön aussieht, aber nicht notwendigerweise konsistent mit dem eigentlichen Programm ist.

Bereits enthalten ist ein Editor zum Eingeben und Verändern von Graphen, die als serialisierte *Java*-Objekte persistent gemacht werden können. Mehrere Layout-Algorithmen stehen zur Verfügung, um auf Knopfdruck erzeugte Zufallsgraphen schnell ansprechend zu layouten.

Die Animationen beschränken sich auf die Darstellung des Graphens und des „Programmtextes“. Kanten werden bei Traversierung durch eine kurze Animationen hervorgehoben und danach dem Zustand nach entsprechend eingefärbt. Hilfestellungen und Anmerkungen können als Kommentar in den Graphen eingeblendet werden, allerdings können diese nicht deaktiviert werden.

2.5.2 VEGA

Visualization environment for geometric algorithms (VEGA) [28] ist an der *Universität Freiburg* entstanden und ist auf die Visualisierung von Geometrie-Algorithmen spezialisiert. VEGA ist als Client/Server-System realisiert: Während auf dem Clienten ein *Java*-Programm abläuft, das für die Darstellung und Benutzerführung verantwortlich ist, läuft der eigentliche in *C++* programmierte Algorithmus auf dem Server ab. Neben einer Vielzahl von Algorithmen aus *LEDA* bzw. *CGAL* können auch eigene Algorithmen vom Server-Betreiber eingespielt werden.

2.5.3 Pavane

Das *Declarative Program Visualization System* (Pavane) [43] ist ein deklaratives Visualisierungssystem für Programme von der *Washington University in St. Louis*. Über entsprechende Felddeklarationen läßt sich der Zustand eines Programmes überwachen, so daß beim Aufruf einer Visualisierungsfunktion die grafische Darstellung aktualisiert wird. Dies passiert jedoch nicht automatisch, sondern muß durch bestimmte Aufrufe aus dem Programm heraus von Hand getriggert werden.

Das System ist nicht auf Graphenalgorithmen spezialisiert und arbeitet deshalb nur auf den von *C* bekannten Datenstrukturen. Mit einer eigenen deklarativen Sprache werden dann aus diesen ggf. verschachtelten Strukturen die zugehörigen Visualisierungen generiert.

2.5.4 Gato

Die *Graph Animation Toolbox* (GATO) [46] von der *Universität zu Köln* kommt den geforderten Zielen am nächsten: Dabei handelt es sich um eine Kombination von *Python* und *Tk*, die das Laden von Algorithmen und Graphen erlaubt, die parallel in mehreren Fenstern agezeigt werden. Durch einen sogenannten Prolog ist es möglich, Voreinstellungen und Visualisierungsstrategien auszuwählen. Abbildung 2.2 zeigt beispielsweise den Schnappschuß einer Tiefensuche. Links im Bild wird der Quelltext angezeigt, in dem jederzeit Unterbrechungspunkte gesetzt werden können. Rechts daneben wird der Graph in einem eigenen Fenster angezeigt, in dem farblich markiert angezeigt wird, ob ein Knoten unbesucht, erstmalig besucht oder abgearbeitet ist.

Kritisieren an der Realisierung kann man, daß die Konfigurationsmöglichkeiten noch unzureichend sind: So möchte man die Knotengrößen bei der Darstellung je nach Graph unterschiedlich groß wählen können oder weitere Angaben zu Knoten und Kanten hinzufügen. Die Verwendung von mehreren Fenstern ist ungeschickt, da diese jedesmal erst aufwendig plaziert werden

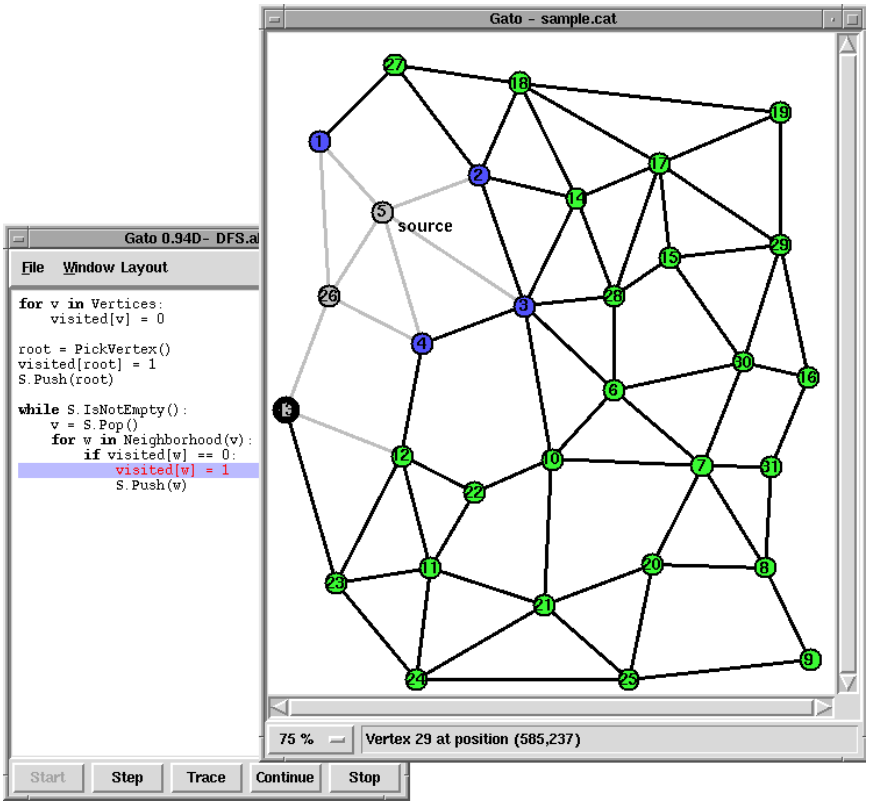

Abbildung 2.2: Graph Animation Toolbox GATO

müssen bevor man alle Informationen gleichzeitig sehen kann. Das Anzeigen weiterer Datenstrukturen wie Warteschlangen und Stacks ist nicht möglich.

Die Sprache *Python* bietet sowohl Vorteile als auch Nachteile: Die Sprache ist noch relativ wenig verbreitet und durch ihren minimalistischen Ansatz nicht jedermans Sache. Als Interpretersprache hat sie mit Geschwindigkeitseinbußen zu kämpfen, was aber auch Vorteile bieten kann, insofern als das Programm auch zur Laufzeit verändert werden kann und so zum weiteren Selbststudium anreizt. *Python*-Interpreter gibt es für fast alle Plattformen (Apple-Macintosh, Unix, Windows,...), insbesondere auch eine Variante, die in *Java* implementiert ist. Dieses bietet neben der oben schon erwähnten Anbindung über *Tk* die Möglichkeit, das Programm als Applet in jedem javafähigen Browser einzusetzen.

Wie auch *Java* sind große Teile der Sprache selbst in *Python* implementiert. So läßt sich aus der Sprache selbst heraus auf den Parser und das Laufzeitsystem zugreifen, was ein sehr tiefes Eingreifen in den Ablauf der Programme erlaubt. Über die Debugging-Schnittstelle lassen sich weitere Informationen während der Ausführung gewinnen.

2.5.5 Leonardo

Leonardo [12] von der *University of Roma* ist eine komplette Programmierumgebung, bestehend aus Entwicklungsumgebung, *C*-Compiler und einer virtuellen Maschine, mit der sich Algorithmen visualisieren lassen. Realisiert wird der deklarativen Ansatz, indem das übersetzte *C*-Programm innerhalb der virtuelle Maschine ausgeführt wird, die das Beobachten aller Datenstrukturen ermöglicht. Über die spezielle Sprache *Alpha* lassen sich Visualisierungen definieren, die automatisch bei Änderungen von Speicherstellen getriggert werden.

Die Software ist momentan leider nur für den Apple-Macintosh verfügbar und erst die nächste Version 4.0 wird auch unter anderen Betriebssystemen laufen. Die zur Zeit verfügbare Beta-Version ist vom Umfang her noch zu sehr unausgereift, um damit effektiv arbeiten zu können oder den Einsatz beurteilen zu können.

Abbildung 2.3 gibt einen Überblick über den Aufbau von *Leonardo*, die aus einem Artikel [11] der Autoren stammt und die Idee hinter der Visualisierung abstrakter Datentypen beschreibt. Bei den in der Literatur und im Handbuch angegeben Beispielen fallen ein paar kleinere Defizite auf.

Die Visualisierung beruht auf der **Veränderung** von Speicherstellen. Ein Vergleich, der den Speicher nur **liest**, kann so nicht direkt angezeigt werden. Um dennoch solche Nur-Lese-Zugriffe zu visualisieren, müssen künstlich Schreiboperationen eingeschoben werden.

Ein weiteres Problem tritt auf, wenn nicht alle Schreibzugriffe direkt zum Neuzeichnen der Anzeige führen sollen. Um zum Beispiel das Vertauschen zweier Variablen unter Zuhilfenahme einer Hilfsvariablen als eine atomare Aktion darzustellen, muss die automatische Aktualisierung vor der ersten Zuweisung deaktiviert und nach der letzten wieder aktiviert werden.

Da sich *Leonardo* nur auf *C* als Sprache beschränkt, läßt die Lesbarkeit der Programme schnell zu wünschen übrig, sobald diese eine gewisse Größe überschreiten. *C* ist in vieler Hinsicht nach nur ein besserer Hochsprachen-Assembler und nicht unbedingt für eine elegante Programmierweise bekannt.

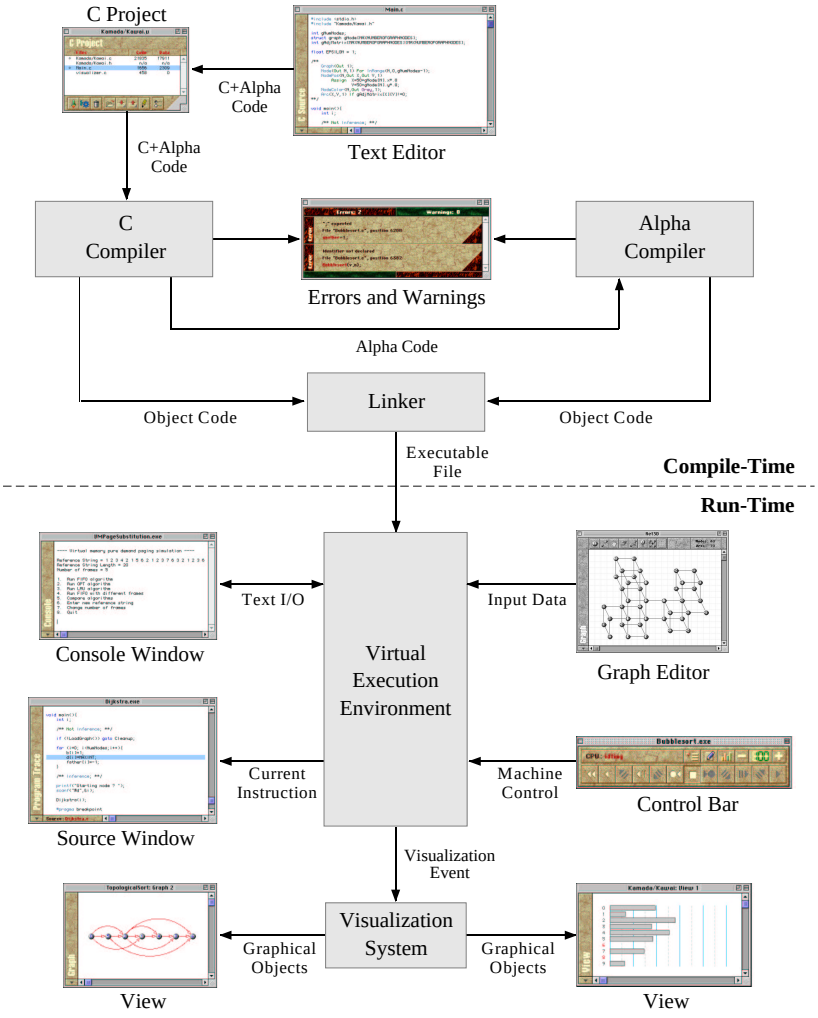

Abbildung 2.3: Strukturübersicht Leonardo

Kapitel 3

Graphalgorithm Animation Tool

Dieses Kapitel beschreibt zunächst verschiedene Strategien, die im Zuge der Arbeit näher untersucht wurden. Im zweiten Teil wird ein eigener auf *Python* basierender Ansatz vorgestellt, der prototypenhaft implementiert wurde. Den Abschluß bildet eine Zusammenfassung und ein Ausblick auf mögliche Erweiterungen dieses Ansatzes.

3.1 Ansätze

Bereits zu Beginn der Arbeit gab es verschiedene Ansätze und Ideen, wie eine Visualisierung zu verwirklichen sein könnte.

3.1.1 Aspektorientierte Programmierung

AspectJ [21] bietet eine Erweiterung zu *Java*, um aspektorientiert programmieren zu können. Diese Methodik erlaubt es, verschiedene Belange eines Programmes zentral an einer Stelle zusammenzufassen, anstatt sie über viele Klassen und Methoden des Programmes zu verteilen. Ein erweiterter Compiler sorgt während der Übersetzung dafür, daß die Aspekte mit dem eigentlichen Ablauf verwoben werden. Hintergedanke daran ist, die Visualisierung des Algorithmus als einen eigenen Aspekt aufzufassen und so den algorithmischen Teil des Programms von der Visualisierung trennen zu können.

Um überhaupt mit *AspectJ* aspektorientiert programmieren zu können, müssen jedoch einige Voraussetzungen geschaffen sein, die den bereits aufgestellten Forderungen entgegenlaufen:

- Der Algorithmus muß in *Java* formuliert werden, da *AspectJ* speziell nur für die Programmiersprache *Java* gemacht ist. Die Verwendung

von *Java* ist in gewisser Hinsicht ein Vorteil, da man sich die Plattformunabhängigkeit zunutze machen kann, um die Visualisierungen sowohl in der Vorlesung als auch privat im Eigenstudium das gleiche System nutzen zu können. Allerdings werden durch die Benutzung der Programmiersprache *Java* einige Formulierungen von Algorithmen recht umständlich und erschweren unnötig das Verständniss.

- Aspektorientierung setzt zumindestens bei *AspectJ* auf einer objektorientierte Fassung der Algorithmen auf, da nur Methodenaufrufe um Aspekte erweitert werden können. Da *Java* noch keine Templates[4] unterstützt, erfordert dies eine Vielzahl von Typ-Umwandlungen, die den Programmcode unnötig aufblähen.

- Um die Visualisierung zu ändern, muß jedesmal der Programmtext des Aspektes geändert und der Algorithmus samt der Animation neu übersetzt werden. Insbesondere bei der Erstellung von Animationen sind solche langen Umlaufzeiten oft störend und hemmen den Fortgang der Arbeit.

- Für jeden Algorithmus muß eine eigene Visualisierung geschrieben werden. Da Aspekte sehr nah an den Methodenaufrufen arbeiten, lassen sich einmal erstellte Visualisierungen nur sehr schwer auf andere Algorithmen der selben Klasse umstellen.

Neben *AspectJ* gibt es noch einige weitere Projekte wie *Subject-oriented programming* (SOP) [38] und *Hyperspaces* [50] von *IBM*, die eine Unterstützung für *C++* und *Java* bieten. Prinzipiell haben diese mit den selben Problemen zu kämpfen und werden deshalb an dieser Stelle nicht ausführlicher betrachtet.

3.1.2 Preprocessing

Vom Literate Programming her ist der Ansatz bekannt, den Quelltext der Programme mit in die Dokumentation aufzunehmen und diesen nur zu übersetzungszwecke daraus zu extrahieren. Durch einen Changefile-Mechanismus [22, 23] ließen sich die Visualisierungsaufrufe in den Quelltext integrieren und so vom eigentlichen Algorithmus trennen.

Das Problem bei diesem Ansatz ist, daß der dargestellte Algorithmus sich vom ausgeführten Algorithmus unterscheidet. Das Laufzeitsystem müsste zum Anzeigen eine Synchonisation der beiden Versionen durchführen, damit der Benutzer nur die für ihn relevaten Zeilen ohne die Visualisierungsanweisungen sieht.

Auch bietet dieser Ansatz dahingehend keine Unterstützung für eine Automatisierung der Darstellung, daß einmalig erstellte Animationen in anderen Algorithmen weiterverwendet werden könnten. Für jeden Algorithmus

müsste man wieder bei Null anfangen und den Programmcode mit Anotationen zur Visualisierung und Ablaufsteuerung versehen.

Alternativ zum Verweben des Algorithmus mit der Animation könnte man auch gleich die Anweisungen zur Visualisierung mit in das Programm schreiben, wie es zum Beispiel *LEDA* in einigen Beispielen tut. Diese Zeilen sollte man dann aber entspechend kennzeichnen, um diese Bereiche bei Bedarf (insbesondere zur Anzeige) auszublenden.

Die *Opt-Out*-Variante hat gegenüber dem *Opt-In*-Verweben den Vorteil, daß die Visualisierung und der eigentliche Algorithmus leichter konsistent gehalten werden können. Allerdings lässt sich so pro Algorithmus auf diese Weise jeweils nur eine Visualisierung integrieren. Sollen mehrere unterschiedliche Visualisierungen notiert werd, so muß der algorithmische Teil des Programms entweder mehrfach kopiert werden, was für spätere Änderungen unschön ist, oder es müssen entsprechende Markierungen gesetzt werden, um die verschiedenen Visualisierungen voneinander zu trennen. Letzteres erfordert aber die Unterstützung durch spezielle Editoren, um mit den unterschiedlichen Programmteilen effizient und übersichtlich arbeiten zu können.

3.1.3 Virtuelle Maschine

Die Systeme *Gato*[46], *Leonardo*[12] und *Pavane*[43] beruhen mehr oder minder alle auf dem Ansatz, den Algorithmus (Controller) in einer kontrollierten Umgebung ablaufen zu lassen. Dabei greifen sie auf Datenstrukturen (Model) zu, die vom Laufzeitsystem beobachtet werden. Dieses triggert nach festgelegten Regeln bei Änderungen in den Daten dann automatisch die zugeordnete Visualisierungen (Views). Abbildung 3.1 stellt diesen Zusammenhang des *Model-View-Controller* Konzepts bildlich dar.

Diese Vorgehensweise bietet den Vorteil, daß Algorithmen fast ohne jegliche Änderungen sofort ausführbar sind. Lediglich die benutzten Datenstrukturen wie Listen, Stacks, Warteschlangen und Graphen müssen gegebenenfalls umgestellt und durch entsprechende animierte Versionen ersetzt werden. Viele Sprachen bieten zwar von sich aus bereits einfache Strukturen wie Listen oder Stacks an, aber gerade in den komplexeren Algorithmen werden Hilfsdatentypen benötigt, die nicht im Standardlieferumfang der Programmiersprachen enthalten sind. Als Beispiele seien hier stellvertretend „Graphen“ und „Mengen“ genannt. Es existieren zwar auch hier verschiedene Bestrebungen[15], solche gängigen Standardstrukturen zur Verfügung zu stellen, aber dadruch, daß sie sehr allgemeingehalten sind, sind sie entsprechend aufwendig und für viele Zwecke zu umfangreich. Von daher ist es ratsam, eine Bibliothek mit solchen Hilfstypen anzulegen und sich direkt dabei Gedanken zu machen, wie diese darzustellen sind.

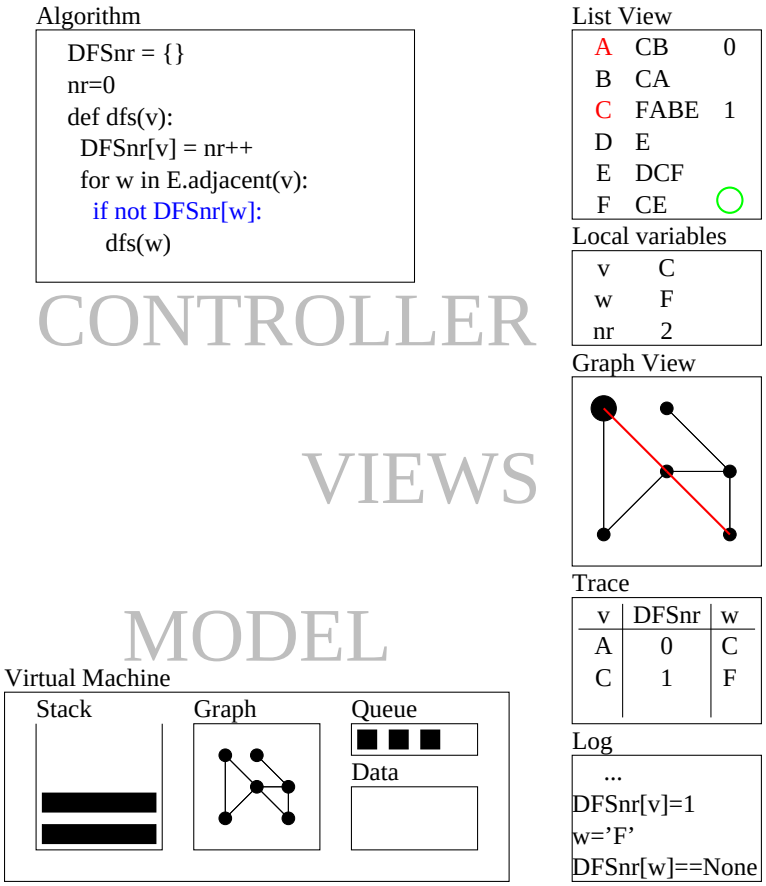

Abbildung 3.1: Model-View-Controller

3.1.4 Interpreter

Ausgehend von den vorherigen Überlegungen ergab sich die Idee, als Grundlage eine interpretierte Sprache zu verwenden. Diese bieten sozusagen eine virtuelle Maschine gratis und ermöglichen es zudem, zu Laufzeit die Umgebung selbst zu inspizieren und zu verwenden.

Idealerweise sollte eine solche Sprache folgende Kriterien erfüllen:

- Die Sprache soll bereits bekannt oder zumindestens leicht erlernbar sein. Da Graphenalgorithmen ein breites Publikum ansprechen und die Software zur Visualisierung insbesondere auch für Grundstudiumsvorlesungen eingesetzt werden soll, ist dieser Punkte nicht zu vernachlässigen.

- Die Sprache soll möglichst einfach sein, damit sie leicht zu verstehen, zu erlernen und zu verwenden ist. Vorallem soll sie intuitiv zu verstehen

sein, um nicht unnötig von den eigentlichen Algorithmen abzulenken, um die es hauptsächlich geht.

- Die Sprache soll mächtig genug sein, um die Algorithmen möglichst nah an die aus den Vorlesungsskripten bekannten Beschreibungen anlehnen zu können. Da diese in einer Pascal-ähnlichen Syntax notiert sind, kommen nur prozedurale Sprachen in betracht.

- Die Sprache soll auf möglichst allen wichtigen Rechnerarchitekturen lauffähig sein, damit sich die Animationen überall erstellen und ausführen lassen. Dies ist insbesondere dann wichtig, wenn das System auch für das Eigenstudium von Algorithmen verwendet werden können soll.

- Die Sprache soll performant genug sein, um die Algorithmen direkt auch für größere Anwendungen verwenden zu können. Dieser Punkt ist eher nebensächlich, da in den meisten Fällen eher kleinere Beispiele für Demonstrationszwecke gewählt werden. Dennoch kann es interessant sein, das Verhalten bei grösseren Eingabefällen zu untersuchen, weswegen der Punkt hier aufgeführt wird.

- Die Sprache soll möglichst typisiert sein, um zusätzlich Typsicherheit zu gewinnen. Weiterhin können diese Informationen indirekt dazu verwandt werden, automatisch passende grafische Repräsentationen zu erzeugen.

Leider ist keine Programmiersprache bekannt, die alle Eigenschaften erfüllt. *Java* ist vielen Studenten bereits in den Anfangssemestern bekannt und auch entsprechend mächtig, scheidet aber wegen den unleserlichen Typenumwandlungen zur Zeit aus. (Siehe Abschnitt 3.1.1 für weitere Gründe) *JavaScript* wird zwar von vielen Browsern unterstützt, aber der Sprachumfang ist zu sehr auf die Verarbeitung von WWW-Seiten beschränkt. *Lisp* ist vielen Informatikern ebenfalls bekannt, kommt aber wegen des funktionalen Programmierstils nicht in Frage. *Ruby*[35] ist eine vor wenigen Jahren entstandenen Sprache, die sich an *Perl* und *Python* anlehnt und vorallem den objektorientierten Gedanken auf die Spitze treibt Allerdings ist die Syntax sehr gewöhnungsbedürftig und die Sprache hat sich nicht sehr weit verbreitet. Der direkte Konkurrent dazu ist *Python*[52], das inzwischen in der Version 2.2 vorliegt. Testweise wurden einige Algorithmen unter dem Gesichtspunkt implementiert, möglichst nah an den Vorlagen aus den Vorlesungen zu bleiben. Als Vorteile für *Python* ergaben sich dabei folgende Punkte:

- *Python* ist eine interpretierte Sprache. Dies ermöglicht es, während der Programmabarbeitung den Quelltext noch zu ändern, um zum Beispiel Alternativen auszuprobieren. Die Verwendung einer interpretierten Sprache hat den Vorteil, daß das Laufzeitsystem direkt für

die Gewinnung von Aufrufen und Veränderungen verwendet werden kann. Für übersetzte Sprachen müsste der Prozessor, auf dem das Programm läuft, selber Möglichkeiten bieten, Breakpoints zu setzen bzw. lesende- und schreibende Speicherzugriffe abzufangen. Alternativ dazu wäre ein virtueller Prozessor nötig, wie ihn beispielsweise *Leonardo* realisiert. Anderenfalls müssten alle verwendeten Datentypen so modifiziert werden, daß sie interessante Ereignisse selber protokollieren, was einen umfangreichen Eingriff in den Programmquelltext darstellt.

- *Python* ist minimalistisch. *Python* selbst besteht aus sehr wenigen, dafür aber relativ mächtigen Strukturen. Einerseits ist die Sprache dadurch leicht zu erlernen, andererseits ermöglicht dies es, den Algorithmus relativ abstrakt zu formulieren.

- *Python* ist plattformübergreifend, das heißt, daß das System an keine feste Architektur gebunden ist und dadurch von vielen Personen benutzt werden kann. Dies befreit einen vor allem von unterschiedlichen Compileraufrufen und Bibiotheksversionen, mit denen man bei übersetzten Sprachen zu kämpfen hat. Für *Python* gibt es Anbindungen an verschiedene andere Sprachen, falls man doch auf die Implementierung von zeitkritischen Routinen in einer anderen Sprache angewiesen ist. Für die grafische Ausgabe kann unter anderem *Tk*, *Qt*, *Gtk* sowie das *Java-AWT* verwendet werden.

- Durch eine in *Java* implementierte Version *Jython* [30] ist die Integration mit einem Laufzeitsystem, welches in *Java* implementiert sein könnte, problemlos möglich. In dieser Kombination ist der Einsatz über das Internet denkbar, wodurch man es den Benutzern ersparen kann, *Python* und andere benötigte Software auf dem lokalen Rechner zu installieren.

Allerdings sind auch einige Nachteile zu nennen:

- *Python* ist eine untypisierte Sprache, so daß keine Typüberprüfung vor der Ausführung des Programms stattfindet. Dadurch fehlt dem Betrachter auch die Information, um welchen Datentyp es sich bei den benutzen Variablen handelt. Diese Informationen müssen auf andere Art und Weise mitgeteilt werden. Allerdings erspart man sich durch die Untypisiertheit Typumwandlungen, die die Lesbarkeit des Quelltextes reduzieren.

- *Python* hat von Version zu Version einige gravierende Änderungen durchgemacht. Viele Skripte, die für Version 1.5 geschrieben wurden, müssen an einigen Stellen an neuere Versionen angepasst werden. Inzwischen existieren 3 verschiedene Versionen nebeneinander: 1.5.2, 2.1.2 und 2.2.

- Rekursionen sind bis *Python* 2.1 nur umständlich programmierbar, da die Selbstreferenzierung nur außerhalb von Objekten und Funktionen zugelassen ist. Erst in Version 2.2 ist es möglich, ohne den Umweg über eine Objektklasse Rekursionen zu programmieren. Für Version 2.1 wurde inzwischen auch eine Möglichkeit geschaffen, dieses Feature aus Version 2.2 zu importieren.

- *Python* ist zwar wesentlich aufgeräumter als *Perl*, aber an einigen Stellen bleiben doch Wünsche offen. In *Python* wird die Objektorientiertheit dadurch realisiert, daß bei Methodendeklarationen innerhalb einer Klasse als erster Parameter die Objektinstanz übergeben wird, die typischerweise mit "self" bezeichnet wird. Dadurch muß aber jede Referenz auf eine Instanzenvariable mit genau diesem Prefix versehen werden, was den Quelltext unnötig verkompliziert. Listen, Tupel (unveränderliche Listen), assoziative Felder, Zeichenketten und Zahlen sind in *Python* keine Objekte, sondern interne Typen mit Objektcharakter. Zwar lassen sich diese Typen alle durch eigenen Klassen emulieren und ersetzen, aber dies erhöht unnötig den Programmieraufwand und bereitet bei der dynamischen Instanziierung zur Laufzeit unnötig Probleme.

3.2 Beispiele

Anhand einiger Beispiele aus den Vorlesungen *Grundzüge der Informatik III* und *Graphenalgorithmen* sollen im Folgenden verschiedene Varianten einer Implementierung beschrieben werden. Ziel dabei ist es herauszubekommen, welche Strukturen in den Algorithmen identifiziert werden können und wie diese visualisiert werden können. Einige Algorithmen werden beispielhaft in *Python* implementiert, um daran die Stärken und Schwächen der Sprache deutlich zu machen.

3.2.1 Minimal Aufspannende Bäume nach Kruskal

Der Kruskal-Algorithmus zum Bestimmen minimal aufspannender Bäume verwendet zwei Datenstrukturen, nämlich eine Partition Z der Knotenmenge V und eine sortierte Liste Q aller Kanten E:

Listing 3.1: Kruskal Algorithmus

```
def kruskal(V,E,fE):
    "Find minimum spanning tree"
    Tree = [ ]
    Q = E.sortedBy(fE)
    Z = UF([ (v,) for v in V ])
    while len(Z) > 1:
```

```
        e = (v1,v2) = Q.pop(0)
8       V1 = Z.find(v1)
        V2 = Z.find(v2)
10      if V1 != V2:
            Z.union(V1,V2)
12          Tree.append(e)
    return Tree
```

3.2.2 Menge der Knotenmengen

Die Menge Z stellt eine Partition der Menge der Knoten dar. Zu Beginn sind diese Mengen einelementig und der Algorithmus läuft solange, bis nur noch eine einzige Menge übrig ist, die dann alle Knoten enthält. Diese schrittweise Vereinigung der Knotenmengen läßt sich auf zwei Arten darstellen:

- Die Zugehörigkeit der Knoten zu einer der Knotenmengen wird in einem eigenen Bereich dargestellt. In einem Bereich ist der Ausgrangsgraph plaziert, während in einem anderen Bereich die Knoten nach ihrer Zugehörigkeit geordnet sind. Die Visualisierung der Vereinigungsoperation ist hier einfach, da die Mengen beliebig plaziert werdne können, so daß die beiden beteiligten Mengen $V1$ und $V2$ direkt nebeneinander angeordnet werden könnten.

  Die räumliche Trennung verschiedener Repräsentaten eines Knotens erschwert jedoch das Verständnis des Ablaufes, da die Aufmerksamkeit des Betrachters zwischen den verschiedenen Repräsentaten wechseln muß und sich der Benutzer jedesmal neu auf dem Ausgabemedium orientieren muß. Im interaktiven Betrieb besteht immerhin die Möglichkeit, daß immer alle Repräsentaten eines Knotens hervorgehoben werden, wenn eine dieser ausgewählt wird. Dadurch hat der Benutzer die Möglichkeit, sich ggf. schnell einen Überblick darüber zu verschaffen, welche Repräsentaten ein und dasselbe Objekt repräsentieren. Trotzdem kann es sinnvoll sein, diese Mengen in einem getrennten Bereich darzustellen, nämlich dann, wenn eine besondere Implementierung für die Mengenoperationen gewählt wird, wie es zum Beispiel im „Round Robin“ Algorithmus der Fall ist. Dort werden die Knoten in einer Partition verwaltet und für jede dieser Partitionen die Kanten in einem Prioritätenwarteschlange mit besonderen Eigenschaften. Eine gemeinsame Darstellung dieser beiden Strukturen in einer Anzeige wäre zu überfrachtet, weshalb sich in diesem Fall mehrere Darstellungen anbieten, um besser auf die Besonderheiten der benutzten Stukturen eingehen zu können.

- Alternativ kann die bereits vorhandene Darstellung des Ursprungsgraphen direkt für die Visualisierung genutzt werden. Dazu werden die

Knoten der beiden Mengen $V1$ und $V2$ entsprechend hervorgehoben und anschließend vereinigt. Dies ermöglicht sofort die bereits zusammenhängenden Komponenten zu erkennen. Nachteilig wirkt sich aus, daß diese Lösung nicht beliebig weit skaliert, denn die Menge der unterscheidbaren Farben ist begrenzt.

Ein weiterer Nachteil besteht darin, daß dann das Merkmal „Farbe“ nicht anderweitig zum Hervorheben des Algorithmenablaufs benutzt werden kann. Dort müsste dann auf „Blinken“ bzw. das Ändern der Linienstärke oder Schraffur zurückgegriffen werden.

- Aus [7] stammt die Idee, die Darstellung in 4 Ebenen aufzuteilen. Abbildung 3.2 stellt dieses Konzept bildlich dar.

  An oberster Stelle werden die Knoten des Graphens gezeichnet. Für diese können Farbe, Schraffuren, Linientyp, etc. beliebig gewählt werden. Weiterhin ist es möglich, in den Knoten oder an ihnen Texte anzubringen. Die Bezeichnung der Knoten ist insbesondere wichtig, um Bezüge zu anderen Daten herzustellen, die an anderer Stelle dargestellt werden.

  Auf der nächst niedrigeren Ebene werden die Kanten gezeichnet. Auch bei diesen sind verschiedene Darstellungsattribute möglich. Durch entsprechendes Einfärben von Kantenzügen kann zum Beispiel implizit der Aufrufstack bei rekursiven Traversierungsalgorithmen dargestellt werden.

  Auf der darunterliegenden Ebene werden sogenannte „Highlights“ verwaltet. Ursprünglich sind diese nur für Knoten vorgesehen, aber dieses Konzept läßt sich auch auf die Kanten erweitern. Diese Hervorhebungen werden oft dazu verwendet, die Aufmerksamkeit kurzzeitig auf bestimmte Knoten und Kanten zu lenken, weil diese gerade im Algorithmus bearbeitet werden.

  Die unterste Ebene schließlich dient dazu, beliebige Polygone unterhalb des Graphens anzuzeigen. Diese können entweder implizit aus den Koordinaten der Knoten und Kanten gebildet werden oder über vollständig eigene Koordinaten spezifiziert werden. Ob und inwieweit letzteres bei einer automatischen Generierung von Animationen nützlich ist, ist allerdings noch zu klären. Diese Hervorhebung bietet sich an, um Partitionen von verschiedenen Knoten und Kanten darzustellen, wie es zum Beispiel im *Dijkstra-Algorithmus* zur Bestimmung der kürzesten Wege oder im Algorithmus zur effizienten Bestimmung von *Zusammenhangskomponenten* vorkommt.

  Abbildung 3.3 zeigt einen Schnappschuß aus dem Ablauf des *Kruskal-Algorithmus* zur Bestimmung eines minimal aufspannenden Baumes.

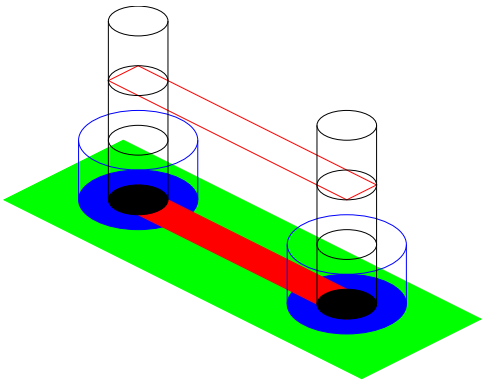

Abbildung 3.2: 4-Ebenen Konzept

Die Kanten (A, B), (F, E) und (E, D) wurden bereits für den Baum ausgewählt, wodurch die Knotenmengen $\{A, B\}$ bzw. $\{D, E, F\}$ gebildet wurden. Die Kante (D, F) wurde verworfen, da die beiden Endknoten bereis in derselben Menge waren. Die Kante (B, C) wird im aktuellen Schritt gerade bearbeitet und ist deswegen hervorgehoben.

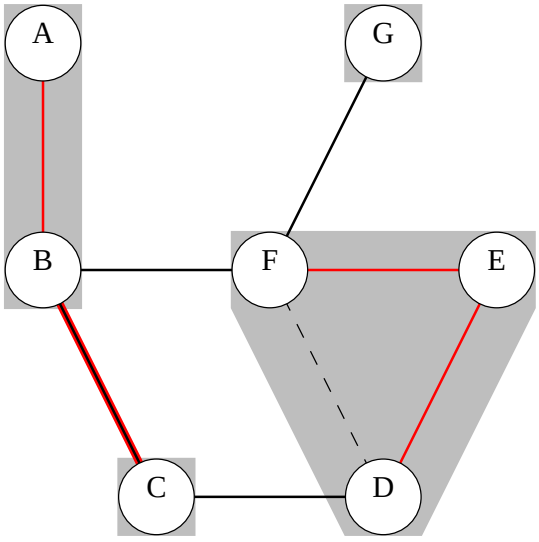

Abbildung 3.3: Beispiel für Kruskal

3.2.3 Liste aller Kanten

Die Kanten werden sortiert nach Kantengewichten in einer Liste Q gehalten, die schrittweise abgearbeitet werden. Eine Kante wird entweder ausgewählt und die beiden benachbarten Knoten vereinigt, oder die Kante wird verworfen und nicht weiter betrachtet.

Die Visualisierung der Kanten gestaltet sich schwieriger als die der Knoten:

- Bei einer vom Ursprungsgraphen getrennten Darstellung der Kanten in einem zweiten Bereich ist die Assoziation mehrerer Repräsentanten ein und der selben Kante miteinander sehr schwierig. Farbkodierungen scheiden meist schon auf Grund der hohen Anzahl von Kanten aus, ebenso andere Merkmale wie Strichstärke und Linienart. Beschriftungen helfen nur wenig, da sie keine schnelle Zuordnung zusammengehöriger Kanten zulassen. Wenn überhaupt sollte eine weitere Darstellung möglichst die Ausrichtung der Kanten aus dem Ursprungsgraphen beibehalten, um den Wiedererkennungswert nicht noch weiter zu reduzieren. Dies führt allerdings dazu, daß sich der Platzbedraf mit jeder solchen zusätzlichen Darstellungen gleich verdoppelt und damit meist aus Platzmangel ausscheidet. Skalierungen und Ausschnitte helfen dabei wenig, unter Umständen kann man jedoch auf die Darstellung der Knoten als Endpunkte der Kanten verzichten.

- Die Kanten werden nach aufsteigenden Kantengewichten abgearbeitet. Da nacheinander abgearbeitete Kanten räumlich weit auseinander liegen können, springt der Algorithmus ggf. wild umher, was für den Betrachter nur schwer nachzuvollziehen ist. Eine sortierte Darstellung müßte in einem eigenen Bereich erfolgen, was zu den eben schon genannten Platzproblemen führt. Für den interaktiven Betrieb besteht die Möglichkeit, die Darstellung erst auf Verlangen einzublenden bzw. den Urspungsgraphen so umzuordnen, daß die Sortierung der Kanten ersichtlich wird. Allerdings sollte man darauf achten, daß durch diese Vorsortierung nicht eine unzulängliche Vereinfachung bei der Ausführung der Algorithmen suggeriert wird, etwa dadurch, daß eine Sortierung sehr kostspielig ist und erst aufwendig durch den Algorithmus erzielt werden muß.

- Bei anderen Algorithmen kann die Reihenfolge für die Visualisierung günstiger ausfallen. Im Falle einer *Breitensuche* oder bei der Bestimmung des *Dreifachen-Zusammenhangs* werden erst alle Kanten eines Knotens der Reihe nach abgearbeitet, bevor andere Kanten in Betracht gezogen werden. Die Reihenfolge kann hier durch eine Zahlenangabe direkt neben oder auf den Kanten angegeben werden.

3.2.4 Rekursion ala Tiefensuche

Das folgende Programm implementiert eine allgemeingehaltene Tiefensuche in *Python*. Da bis zur Version 2.1 Rekursionen nur über globale Funktionen oder Methoden von Objekten realisiert werden konnten, benutzt diese Variante letztere Möglichkeit. Dieses biete sich hier an, da neben der eigentlichen rekursiven Funktion auch ein Zähler *count*, ein Verweis auf den Vorgängerknoten *parent* und drei Funktionen benutzt werden, die beim ersten Besuch eines Knotens ($firstVisit$), nach dem Besuch eines Nachbarknotens ($afterReturn$) bzw. beim Auftreten eines Zykluses ($atCycle$) aufgerufen werden.

Listing 3.2: DFS Traversierung

```
2   def visit ( self ,v):
        "Helper function"
4       self .DFSnr[v] = self.count
        self .count += 1
6       self . firstVisit ( v )
        for w in self .E.adjacent( v ):
8           if not self .DFSnr.has_key( w ):
                self .T.append( (v,w) )
10              self .parent[w] = v
                self . visit ( w )
12              self .afterReturn(v)
            elif (w != self .parent[v]) and (self .DFSnr[v] > self.DFSnr[w]):
14              self .R.append( (v,w) )
                self .atCycle( v )
```

Dabei ergibt sich das Problem, wie globale Parameter wie die DFS-Nummer $DFSnr$ und der aktuelle Zähler *count* zu speichern sind. Die Implementierung folgt hier dem *Visitor-Pattern*[19] und ist als eine eigene Klasse implementiert. Alternativ dazu könnten global gültige Variablen benutzt werden, wodurch sich aber ein Benennungsproblem gibt. Insbesondere bei der Bestimmung des *Dreifachen Zusammenhangs* führt dies zu Problemen, da dort mehrere Traversierungen des Graphens erforderlich sind.

Beim rekursiven Aufruf ergibt sich das Probelm, die Aufrufreihenfolge implizit oder explizit darzustellen. Neben der Darstellung der Aufrufparameter ist eventuell auch die Darstellung lokaler Variablen sinnvoll. Diese Informationen können sehr viel Platz benötigen, weswegen hier besonders hohe Flexibilität gefordert ist. Es erscheint sinnvoll, zum Beispiel nur den aktuellen Stackframe anzuzeigen und alle übergeordneten zusammenzuklappen bzw. ganz auszublenden. Jedoch sollte man einem interaktiven Benutzer die Möglichkeit bieten, die Informationen auf Wunsch dennoch vollständig darzustellen.

Neben der Darstellung des Aufrufstacks kann bei einigen Algorithmen die Aufrufreihenfolge implizit als ein Weg im Graphen verstanden werden, die sich durch Färbung oder Verdickung der Kanten deutlich machen läßt. Dabei lassen sich jedoch schlecht eventuelle Parameter darstellen, so daß man in den meisten Fällen darauf verzichten muß. In einer interaktiven Umgebung kann man dem Betrachter aber die Möglichkeit bieten, im Aufrufstack nach oben zu navigieren, um dort den Zustand und den Wert lokaler Variablen einzusehen.

3.2.5 Abgeleitete Graphen

Einige Algorithmen erzeugen oder arbeiten auf abgeleiteten Graphen. Dabei handelt es sich um Kopien der Ausgangsgraphen, die durch den Algorithmus im Laufe der Zeit verändert und angepasst werden, um eine Aussage über die Ursprungsgraphen zu treffen.

Listing 3.3: Ford-Fulkerson Algorithmus

```
def fordFulkerson(V,E,fCap,q,s):
    """fordFulkerson(V,E,fCap,q,s) -> number
    Find maximum flow from Q to S in graph (V,E) with
    capacity function fCap(E).
    """
    rest = {}
    for e in E: rest[e] = fCap(e)
    maxFlow = 0
    while 1:
        R = Directed.Graph(V, [e for e in rest.keys() if rest[e] > 0])
        path = find(R, q, s)
        if not path: break
        r = min( [rest[e] for e in path] )
        for (v,w) in path:
            rest [ (v,w) ] = rest [ (v,w) ] - r
            rest [ (w,v) ] = rest.get( (w,v), 0) + r
        maxFlow += r
    return maxFlow
```

Beispielsweise arbeiten viele Algorithmen zur Bestimmung des maximalen Flusses auf einem Restgraphen, der zunächst die Knoten und Kanten des Ursprungsgraphen enthält. Durch das Aufbringen von Flüssen ändern sich über mehrere Schritte hinweg die noch zur Verfügung stehende Restkapazitäten. Zusätzlich kommen neue Kanten in Rückrichtung hinzu oder vorhandene Kanten werden aus dem Graphen entfernt.

Im Falle des oben angegebenen *Ford-Fulkerson-Algorithmus* sind also pro Kante die 3 Angaben *Maximale Kapazität*, *Momentaner Fluß* und *Restkapazität* angebracht. Wenn im Ausgangsgraphen bereits eine Kante in umge-

kehrter Richtung existiert, sind dann maximal 6 Angaben (2× Kapazität, 2× Fluß, je 1× Restkapazität in jede der beiden Richtungen) unterzubringen.

Alternativ bietet sich die Möglichkeit, neben dem Ursprungsgraphen mit den Kapazitäten und Flüssen den Restgraph gesondert darzustellen. Von Vorteil daran ist, daß man hier alle Kanten weglassen oder dimmen kann, die für den Algorithmus nicht mehr in Frage kommen. (Bezogen auf das Beispiel von oben wären das solche Kanten mit Restkapazität 0.) Allerdings benötigen mehrere Darstellungen des Graphens zusätzlichen Platz, der unter Umständen anders sinnvoller genutzt werden kann. Zudem ist es für den Betrachter schwieriger, die beiden Graphen wieder zur Deckung zu bringen, um die Zusammenhänge zwischen den Graphen erkennen zu können. Von daher sollten Kopien bzw. abgeleitete Graphen zumindestens mit der gleichen Anordnung von Knoten und Kanten untereinander oder nebeneinander angeordnet sein, um den Wiedererkennungsprozeß zu erleichtern. Markierungen des Weges oder Hervorhebungen von Knoten bzw. Kanten sollten auch in allen Darstellungen gleichzeitig erfolgen, damit diese als Orientierungshilfen eingesetzt werden können. Programmtechnisch ist dies dadurch zu erreichen, daß die Koordinaten von Knoten relativ zu einem Ursprungspunkt des jeweils zugeordneten Graphen interpretiert werden.

3.2.6 Dynamische Strukturen

Verstärkt wirken sich diese Probleme noch bei der Verwendung dynamischer Strukturen aus. Darunter sind solche Strukturen wie Listen und Bäume zu verstehen, die während der Laufzeit des Algorithmuses dynamisch erzeugt und verändert werden. Solche Veränderungen wie das Einfügen von neuen Elementen oder das Entfernen von alten Elementen müssen dem Betrachter so dargestellt werden, daß diesem sofort einsichtig ist, welches Element mit der Ausführung des nächsten Befehls entfernt bzw. eingefügt wird. Insbesondere beim Entfernen muß das betroffenen Element bereits vor der Ausführung des nächsten Befehls hervorgehoben werden, so daß der Benutzer ggf. die Ausführung des Programms anhalten oder sogar noch abändern kann. Dabei treten neue Konflikte auf, da ggf. die Visualisierung des letzten Befehls sich mit der Darstellung des nächsten Befehls überlappen.

Es erscheint daher sinnvoll, die Ausführung eines Befehles in zwei Phasen zu unterteilen: Die erste Phase dient dazu, die Argumente und Objekte des als nächsten auszuführenden Befehls zu markieren, während die zweite Phase die Ausführung und das Ergebnis visualisiert. Das Markieren der Argumente kann weitgehend automatisch geschehen, so daß in den meisten Fällen nur die eigentliche Ausführung entsprechend dargestellt werden braucht. Für sehr einfache Befehle, deren Ausführung sofort ersichtlich ist, soll es möglich sein, eine oder beide der Phasen zu überspringen, um die Animation nicht in die Länge zu ziehen.

Ein weiteres Problem tritt dadurch auf, daß das Plazieren solcher ver-

änderlichen Graphen zusätzlichen Aufwand erfordert. Insbesondere das Hinzufügen neuer Knoten bereitet Probleme, da diesen Objekten zunächst eine Position zugeordnet werden muß. Mit der Ergänzung weiterer Kanten kann es später sinnvoll sein, Knoten an eine andere Stelle zu verschieben, um zum Beispiel die Zahl von Überschneidungen zu minimieren. Das Bewegen von Knoten hat für den Betrachter den Nachteil, daß er sich in der neuen Plazierung erst wieder neu orientieren muß, von daher sollten einmal plazierte Element möglichst an ihrer Stelle belassen werden. Ausschließen sollte man diese Möglichkeit aber grundsätzlich nicht, da sich dies durchaus auch nützlich verwenden läßt, etwa zum Verdeutlichen der Komponenten bei der Bestimmung von Zusammenhangskomponenten. Auf jeden Fall sollte eine solche Verschiebung in einer interaktiven Umgeung durch eine kontinuierliche Animation erfolgen. Ist dies nicht möglich, so besteht ggf. die Möglichkeit, die Verschiebung durch das Einzeichnen von Pfeilen in einem Zwischenschritt darzustellen.

Da für den Ablauf der hier betrachteten Algorithmen die grafische Plazierung uninteressant ist, bietet es sich an, die Anordnung der Objekte dem Benutzer zu überlassen. Im dynamischen Fall, bei dem Knoten hinzukommen und verschwinden, wählt man dazu einen Zustand des Graphens, bei dem möglichst alle Knoten vorhanden sind und ordnet für diesen alle Objekte an. Die so gewonnenen Positionsangaben werden dann zum Beispiel in einer Datei hinterlegt, von wo aus sie dann bei der Visualisierung abgerufen werden können.

Manche Strukturen lassen sich sehr leicht plazieren, wie es etwa bei Wurzelbäumen der Fall ist. In den *GFC* [8] wurde dazu das *Graph Layout Framework* eingeführt. Dieses ermöglicht es, einem Graphen einen Layout-Algorithmus zuzuordnen, der bei Bedarf aufgerufen wird, wenn die Bildschirmposition der Kanten und Knoten benötigt wird.

3.3 Anforderungen an das Animationssystem

Aus den vorhergehenden Untersuchungen und Beispielen ergeben sich einige Forderungen an das Animationssystem, die auf jeden Fall unterstützt werden sollten.

3.3.1 Datenstrukturen

In den meisten Fällen benötigen Algorithmen weitere Datenstrukturen. Im folgenden werden die wichtigsten für sie typischen Operationen aufgeführt, für die Animationen vorhanden sein sollten.

List Listen treten in den meisten Fällen in einer der beiden Ausprägungen *Verkettete Liste* und *Feldbasierte Liste* auf.

	Linked List	Array
Insert single	X	X
Delete single	X	X
Insert multiple	X	
Delete multiple	X	
Swap two		X
Merge w. other	X	X

Queue Warteschlangen sind in vielen Fällen *Listen*, die nur eingeschränkt Operationen zulassen. Daneben existieren aber noch eine Reihe weiterer Implementierungen wie *Binäre Heaps* und *Fibonacci Heaps*, deren Operationen wesentlich aufwendiger zu animieren sind.

	List	BinHeap	FibHeap
Enqueue	X	X	X
Dequeue	X	X	X
Merge w. other	X	X	X

Stack Stapel sind relativ einfach, da sie nur die Operationen *Push* und *Pop* zulassen. Der Aufrufstapel zur Programmausführung ist ein Spezialfall eines Stacks, der implizit immer vorhanden ist. Insbesondere bei der Visualisierung von rekursiven Funktionsaufrufen ist dessen Darstellung sinnvoll, um gleichzeitig damit Übergabeparameter und lokale Variablen anzeigen zu können. Um Platz zu sparen bietet er sich dabei an, teile dieser Anzeige zusammenzuklappen oder auszublenden.

Set Einzelne Mengen sind einfach zu realisieren, da diesen nur die Methode *Add*, *Remove*, *Merge* und *Find* benötigen. Durch Verschachtelungen entstehen aber komplexere Strukturen, die eine gesonderte Visualisierung erfordern. Als Beispiel seien hier *Partitionen* mit den Operationen *Merge*, *Split* und *Find* genannt.

	Set	Partition
Insert element	X	
Remove element	X	
Merge w. other		X
Split		X
Find	X	X

Graph Graphen lassen sich auf unterschiedlichste Art und Weise notieren und darstellen, *Adjazenzlisten* und *Adjazenzmatrizen* sind nur zwei Möglichkeiten. Allgemein besteht ein Graph aus *Knoten* und *Kanten*, an denen zusätzliche Markierungen und Gewichte angebracht werden können. Kanten und Knoten lassen sich *entfernen* oder *hinzufügen*, Kantengewichte nur *ändern*. In *Bäumen* sind die Kante oft implizit duch einen Verweis auf den Vaterknoten realisiert, weswegen das

Ändern von Endpunkten einer Kante zusätzlich unterstützt werden sollte.

3.3.2 Effekte

Diese Veränderungen der Datenstrukturen müssen nun für den Betrachter anschaulich visualisiert werden. Dazu müssen interessante Bereiche und Objekte hervorgehoben werden. Hauptsächlich für den interaktiven Betrieb, aber auch für die Generierung von Schnappschüssen erscheinen folgende Möglichkeiten zur Visualisierung sinnvoll:

- **Fokussieren/Hervorheben**
  Einzelne Objekte oder Gruppen von Objekten können einen Fokus erhalten oder verlieren, wie es in Abbildung 3.2 dargestellt ist. Der Wechsel des Fokuses von einem Objekt auf ein anderes kann im interaktiven Betrieb durch Überblenden anstatt durch einen apprupten Wechsel erfolgen, damit der Betrachter den Wechsel besser wahrnimmt. Reichen Farben und Muster alleine nicht aus, so können zusätzlich Pfeile bei der Anzeige verwendet werden.

- **Blenden/Dimmen**
  Objekte können ein- bzw. ausgeblendet werden. Das Ausblenden kann sich auch nur auf das Dimmen beschränken, um das Objekt weiterhin in Hintergrund halten zu können. Dies ist besonders nützlich, um bereits verarbeitete Objekte in den Hintergrund zu verschieben, ohne ihre Umgebung als Kontext für andere Aktivitäten zu verlieren.

  Das vollständige Entfernen vom Bildschirm hat für den Betrachter den folgenden Nachteil: Eine Kante kann zum Beispiel bis zum Zeitpunkt ihres Entfernens eine wichtige Funktion erfüllt haben. Wird diese dann vollständig entfernt, so fehlt dem Benutzer evtl. ab diesem Zeitpunkt ein wichtiger Anhaltspunkt, wie der Algorithmus zu den voherigen Zuständen gekommen ist.

- **Bewegen**
  Bei manchen Algorithmen kann es sinnvoll sein, die dargestellten Objekte zu bewegen. Diese Möglichkeit sollte aber nur sehr vorsichtig eingesetzt werden, da jede Umordnung des Dargestellten eine Neuorientierung des Betrachters erfordert. Zumindestens im interaktiven Betrieb sollten die Übergänge fließend erfolgen und Sprünge vermieden werden. Im nicht-interaktiven Betrieb kann es hilfreich sein, wenigstens einen Zwischenschritt zu erzeugen, bei dem ein Pfeil die Bewegungsrichtung anzeigt.

3.3.3 Ablaufsteuerung

Bei dynamischen Veränderungen und deren Visualisierung ergibt sich wie an anderer Stelle auch das Problem, in wie weit die Animation selbständig ablaufen soll und in wie weit der Entwickler und Betrachter eingreifen können soll. Es erscheint sinnvoll, sowohl interaktive Steuerelemente (drücken einer Taste oder eines Knopfes) und automatische Abläufe zu ermöglichen, die allerdings vom Benutzer in der Ablaufgeschwindigkeit verändert und angehalten werden können.

In vielen Algorithmen finden sich besonders interessante Stellen, während die übrigen Befehle eher trivial und weniger interessant sind. Anstatt jeden Befehl einzeln darzustellen, sollten Befehle zu Ausführungseinheiten zusammengefasst werden, die dann gemeinsam visualisiert bzw. sogar übergangen werden können. Wie solche Blöcke beispielsweise spezifiziert werden können, wird unter anderem im nächsten Abschnitt beschrieben.

Abbildung 3.4 zeigt einige Navigationelemente, die sinnvollerweise folgende Funktionen erfüllen:

Reset Setzt die Visualisierung zurück zum Anfang, so als ob sie gerade geladen wurde.

Block Zurück Macht alle Befehle bis zu einer vorher gesetzten Marke rückgängig.

Zurück Macht den letzten Befehl rückgängig.

Anhalten Hält die Animation augenblicklich an.

Weiter Fährt mit dem nächsten Befehl fort.

Vorlauf Überspringt die momentane Animation und bleibt vor der Ausführung des nächsten Befehls stehen.

Block Vorlauf Überspringt alle Animationen bis zum Beginn des nächsten Blocks.

Ende Führt den Algorithmus bis zum Ende aus und zeigt das endgültige Ergebnis ohne Zwischenschritte an.

Abbildung 3.4: Ablaufsteuerung

Bei den Vorlauf- und Rücklauf-Operationen stellt sich jedesmal die Frage, ob diese jedesmal animiert werden sollen oder einfach übersprungen werden. Als Zwischenlösung können diese weichen Übergänge zwischen zwei Zuständen auch stark beschleunigt wiedergegeben werden, um den Betrachter nicht zu langweilen. Allgemein sollte deshalb eine Möglichkeit vorhanden sein, die Ablaufgeschwindigkeit mit einem Multiplikator versehen zu können, der alle Verzögerungen entsprechend skaliert bzw. sogar vollständig deaktiviert.

Ein zentrales Problem bei der Animation besteht darin, Unterbrechungen im Programmfluß geeignet zu spezifizieren. Unterbrechungen dienen dazu, „uninteressante“ Programmabläufe zu überspringen, um an „interessanten“ Punkten den Zustand zu visualisieren. Von verschiedenen *Debuggern* ist das Setzen von *Breakpoints* im Programm bekannt, die den Programmablauf vor oder nach der Ausführung eines markierten Befehls unterbrechen. Diese Art der Unterbrechung ist auch bei der Visualisierung von Graphenalgorithmen sinnvoll, allerdings wünscht man sich gerade dabei einen hohen Freiheitsgrad und vielfältige Möglichkeiten, solche „interessanten Ereignisse“ zu spezifizieren. So interessiert nicht jede Ausführung eines Befehls, sondern nur bestimmte Iterationen sind interessant. Fortgeschrittene Debugger für Programme bieten dazu die Möglichkeit, auf verschiedene Zähler zurückzugreifen oder Breakpoints nur bei bestimmten Konditionen auszuführen. Moderne Prozessoren bieten dafür spezielle Unterstützung an, um diese Tests effektiv durchführen zu können.

Bei der Angabe solcher Unterbrechungen im Algorithmenablauf möchte man ähnliche Methoden benutzen, um den Ablauf automatisch anhalten zu können. In den seltensten Fällen erscheint es sinnvoll, diese Konditionen in absoluten Zahlen der From „beim dritten Durchlauf“ anzugeben, da diese nur für genau diesen Algorithmus mit genau dieser einen Eingabe funktionieren. Man wird eine Animation in den meisten Fällen zwar für einen speziellen Graphen erstellen, aber eine kleine Änderung des Graphens kann schon dazu führen, daß viele bereits angegebenen Unterbrechungen überarbeitet werden müssen, um hinzugekommene oder weggefallene Schleifendurchläufe zu kompensieren.

Vielmehr möchte man solche Bedingungen in der Regel auf einer höheren Ebene angeben, etwa in der From „nachdem dies und jenes eingetreten ist, anhalten“. Das Erstellen einer solchen Beschreibung bieten den Vorteil, daß sich einmal erstellte Unterbrechungen durchaus auch für andere Datenfälle verwenden lassen. Verglichen mit der Primitivvariante ist es jedoch wensentlich aufwendiger und komplizierter, solche komplexen Unterbrechungen zu beschreiben.

3.4 Graph Animation Tool

Ausgangsbasis ist zunächst der sogenannte Ablauftrace, der alle Veränderungen des Programmzustands enthält. Diese zeitlich linear geordneten Ereignisse werden in einem ersten Schritt um zusätzliche Informationen angereichert. Aus dem Programmtext werden Informationen wie Zeilennummern, Programmblöcke und Programmkonstrukte extrahiert und miteinander verwoben. Daraus ergibt sich ein erweiterter Trace, in dem jede Zuweisung einem Programmabschnitt zugeordnet werden kann. Diese Abschnitte lassen sich in einen dynamischer Ablaufbaum verwandeln, der neben dem zeitlichen Ablauf auch Verschachtelungen und Strukturinformationen enthält.

In diesem Baum können nun Ereignisse darüber spezifiziert werden, daß der Weg von der Wurzel bis zu dem zugehörigen Knoten beschrieben wird, der das Ereignis repräsentiert. Zur Beschreibung solcher Wege läßt sich *XPath* [9] einsetzen, welches vielfältige Möglichkeiten dazu bietet. *XPath* ist ein wichtiger Bestandteil von *XML* und dient dazu, in der Baumstruktur des *DOM*[1] zu navigieren. Beispielsweise beschreibt der Ausdruck //dfs[@n>2] alle dfs-Knoten, deren Attribut n größer 2 ist. Die Wahl von *XPath* hat unter anderem die Vorteile, daß bereits etliche Implementierungen davon in den unterschiedlichsten Strachen existieren und diese Ausdrücke auch von anderen Programmen weiterverarbeitet werden können.

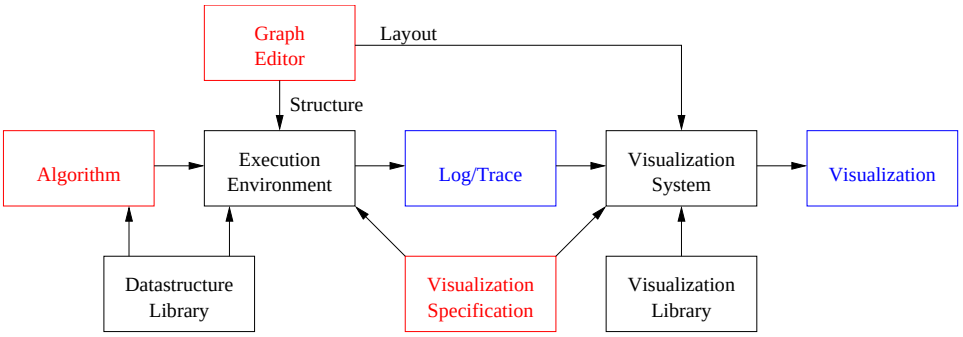

Abbildung 3.5: Datenfluß

Abbildung 3.5 beschreibt beispielhaft, wie ein Animationssystem strukturiert sein könnte. Die schwarz dargestellten Module sind das eigentliche System, rot markiert sind die Eingaben des Benutzers, während die Ausgaben blau umrandet sind. Die folgenden Abschitte gehen im einzelnen auf die verschiedenen Module ein.

[1] Document Object Model

3.4.1 Eingabe des Algorithmus

Als erstes muß der Algorithmus implementiert werden. Dazu sollte eine Programmbibliothek bereitstehen, die möglichst alle benötigten Datenstrukturen bereits fertig implementiert und getestet zur Verfügung stellt.

Listing 3.4: DFS-Durchlauf

```
DFSnr = {}
nr = 0
def dfs(v):
  global E, DFSnr, nr
  DFSnr[v] = nr
  nr += 1
  for w in E.adjacent(v):
    if not DFSnr.has_key(w):
      dfs(w)
```

Diese zu Listing 3.2 alternative Beispielimplementierung einer Tiefensuche benutzt den speziellen Aufruf E.adjacent(v), der aus der Graphen-Bibliothek stammt und die zu v benachbarten Knoten als Liste zurückliefert. Anstatt ein indiziertes Feld zu benutzen, werden die DFS-Nummern hier in einem assoziativen Array gespeichert. Vorteil dieser Lösung ist, daß jedes „unveränderliche Objekt“ als Knoten verwendet werden kann, seien es Zahlen, Zeichenkette oder echte Objekte im Sinne von *Python*. Um ein assoziatives Feld verwenden zu können, müssen die Objekte lediglich „hashbar“ und miteinander vergleichbar sein. Insbesondere lassen sich so beliebige Objekte verwenden, um zum Beispiel Informationen, die zu den Knoten selbst gehörigen, innerhalb des Knoten-Objekts abzuspeichern, anstatt sie in externen Strukturen unterzubringen.

Von dieser Möglichkeit, die DFS-Nummern in einem eigenen Attribut der Knoten abzuspeichern, wird hier allerdings nicht gebrauch gemacht. Und zwar aus den folgenden beiden Gründen:

- Sie funktioniert nur auf Objekten, was die Verwendung von Zahlen oder Zeichenketten als Knoten ausschließt, da es sich dabei nicht um *Python*-Objekte handelt, sondern um *Python*-Primitive.

- Der Name des Attributes wird bei der Implementierung des Traversierungsalgorithmus festgelegt. *Python* bietet zwar auch eine Möglichkeit, den Namen des Attributs erst zur Laufzeit festzulegen, worauf aber hier verzichtet wurde, um die Implementierung des Algorithmus an dieser Stelle nicht unnötig zu verkomplizieren. Soll der Graph merhfach mit der gleichen Implementierung traversiert werden, so müssen zuvor die alten Werte gesichert und zurückgesetzt werden. In der oben implementierten Variante genügt es dagegen, das Feld zu kopieren bzw. zu löschen.

Große Laufzeitunterschiede ergeben sich bei der Verwendung von assoziativen Arrays gegenüber indizierten Feldern nicht, was auf der internen Implementierung von Feldern in *Python* begründet ist.

3.4.2 Eingabe des Graphen

Die Eingabe des Graphen kann auf unterschiedliche Art und Weise erfolgen. In Abschnitt 2.3 wurden verschiedene Formate zur Graphenrepräsentation beschrieben und in Abschnitt 2.4 verschiedene Editoren zu deren Eingabe vorgestellt. Für dieses Beispiel wird der Graph direkt in *Python* aus einer Liste der Kanten generiert.

Listing 3.5: Eingangsgraph

```
from Graph.Helpers import graphFromEdges
from Graph import Undirected
G = graphFromEdges([
  ('A','C'),('A','B'),('B','C'),('C','F'),('C','E'),('D','E'),('E','F')
  ], Undirected)
V,E = G[:2]
```

Die letzte Zeile macht die Knoten und Kanten des Graphens G unter den Namen V und E zugänglich. Als dritte Komponente des Graphens liefert der Aufruf von graphFromEdges eine Gewichtsfunktion zurück, die hier lediglich verworfen wird. Diese Funktion definiert sich durch eine optionale Angabe bei den Kanten, die hier aber vergelassen ist, da sie für dieses Beispiel nicht benötigt wird.

3.4.3 Ausführung des Programms

Danach kann kann das Programm ausgeführt werden. Dazu wird eine spezielle Umgebung geschaffen, in der das Programmfragment ausgeführt wird.

Listing 3.6: Algorithmusausführung

```
from Tracer import Wrapper
env={'E':Wrapper(E), 'DFSnr':Wrapper(DFSnr), 'nr':Wrapper(nr)}
execfile ("dfs.py", env)
exec "dfs('A')\n" in env
```

In *Python* wird nur zwischen lokaler und globaler Sichtbarkeit unterschieden. Dazu verwaltet *Python* alle Variablen in zwei assoziativen Feldern, die Namen auf Objekte abbilden. Zur Ausführung wird ein neuer Namensraum *env* für die globalen Variablen vorbereitet, um darüber dem Algorithmus alle Objekte mitzugeben. Die interessanten Variablen werden durch einen "Wrapper" gekapselt, der alle Zugriffe auf diese Variablen registriert und in ein "Log" geschreibt.

Zusätzlich zur Beobachtung der interessanten Objekte läßt sich der Programmablauf genauer aufzeichnen. Dazu existiert der "Tracer", der alle Funktionsaufrufe, Rückgabewerte und Ausnahmebedingungen sowie die ausgeführten Programmzeilen aufzeichnet. Dieser wird durch das folgendes Codefragment 3.7 eingerichtet:

Listing 3.7: Programmüberwachung

```
import Tracer,sys
...
sys.settrace(Tracer.Tracer())
exec ...
sys.settrace(None)
```

Das so gewonnene "Log" enthält neben den Methoden- und Funktionsaufrufen auch die Zeilennummern aus dem Programm, die später als Referenzpunkt in einer Visualisierung verwenden können. Der Tracer ist eine Klasse, die vom *Python*-Interpreter während der Ausführung aufgerufen wird, um verschiedene Ereignisse mitzuteilen:

- **Methoden- bzw. Funktionsaufrufe**
  Neben dem Namen der Funktion werden zusätzlich alle Argumente übergeben. Über *Python*-eigene Sprachmittel lassen sich diese inspizieren und protokollieren.

- **Funktionsrücksprung**
  Beim Verlassen einer Funktion oder Methode wird der Rückgabewert festgehalten.

- **Ausnahmebedingung**
  Tritt während der Ausführung eine Ausnahmebedingung auf, so wird das Auftreten statt eines Rückgabewerts aufgezeichnet.

- **Zeilenausführung**
  Vor der Ausführung einer Programmzeile, wird diese in das "Log" geschrieben. Stehen mehrere Befehle in einer Zeile, so wir diese Zeile auch mehrfach gemeldet. Deshalb sollte wie in *Python* üblich pro Programmzeile lediglich eine Anweisung stehen.

Durch das Ausführen entsteht das folgende "Log" 3.8:

Listing 3.8: Unlimitierter Trace

```
dfs.py : 2     dfs (v='A')
dfs.py : 2 def dfs(v):
dfs.py : 3         global E, DFSnr, nr
dfs.py : 4         DFSnr[v] = nr
Tracer.py : 202       __setitem__ ( self={}, index='A', value=0)
Tracer.py : 202     def __setitem__(self, index, value):
```

Zunächst einige Erklärungen zu den Einträgen dieses Auszugs:

1. Die Funktion dfs wird mit dem Parameter v='A' aufgerufen.
2. Der Interpreter führt an dieser Stelle die Funktion dfs(v) aus, wie sie zuvor definiert wurde.
3. Die Variablen E, DFSnr, nr wird aus dem globalen Kontext verwendet.
4. Der DFSnr[v] des Knotens v wird der Wert nr zugewiesen.
5. Der Wrapper für die Variable DFSnr wird aufgerufen, um den Wert für den Eintrag 'A' auf '0' zu setzen
6. Ab dieser Stelle führt der Interpreter den Wrapper aus, der auch vollständig mitprotokolliert wird.

Dieses "Log" enthält zwar nun viele benötigte Informationen, ist aber für eine automatische Weiterverarbeitung eher weniger geeignet. Der vollständige Trace umfasst über 500 Zeilen, da jeder Aufruf, jede Änderung und jede Abfrage mitgeschrieben wird. Unter anderem enthält es auch solche Aufrufe, die in Hilfsklassen wie dem "Wrapper" stattfinden.

An dieser Stelle gilt es deshalb, diese Informationsflut einzuschränken, wozu sich der "Tracer" eines einfachen Tricks bedient. Dafür muß man jedoch einen tieferen Blick in die Funktionsweise des *Python*-Debuggers werfen. Führt der Interpreter eine Funktion aus und ist eine Tracefunktion aktiviert, so werden dieser Tracefunktion einige Informationen über den Aufruf mitgeteilt. Diese Funktion kann nun über ihren Rückgabewert entscheiden, ob und mit welcher Funktion alle weiteren Aufrufe verfolgt werden sollen. In der Implementierung von oben gab die Funktion einfach immer eine Referenz auf sich selbst zurück, wodurch alle rekursiven Aufrufe vollständig mitprotokolliert wurden.

Um nun an dieser Stelle die nötige Flexibilität zu haben, bedient sich der Tracer eines zweiten Tricks: Jedem Objekt (und damit auch jeder Funktion bzw. Methode) können neben einem Kommentar zur Dokumentation auch weiter Attribute zugeordnet werden. Über einen solchen Kommentar wird nun gesteuert, ob die Ausführung weiter rekursiv weiterverfolgt werden soll, oder ob stattdessen eine besondere Meldung im "Log" vermerkt werden soll.

Des weiteren werden die Informationen umsortiert und hierarchisch in einem sogenannten "Execution Tree" angeordnet. In diesem Ablaufbaum ist dann zum Beispiel erkennbar, welche Funktionsblöcke aufgrund welcher Aufrufen ausgeführt wurden oder welche Schleifen wie oft durchlaufen wurden.

Listing 3.9: Hierarchischer Trace

```
<line file="dfs.py" nr="2">def dfs(v):</line>
<line file="dfs.py" nr="3">global E, DFSnr, nr</line>
```

```
<line file="dfs.py" nr="4">DFSnr[v] = nr</line>
<call name="__setitem__">
  <arg name="self" id="135162596"><instance type="Wrapper"/></arg>
  <arg name="index" id="134779304"><string>A</string></arg>
  <arg name="value" id="135225164"><number>0</number></arg>
  <return id="1074392460"><none/></return>
</call>
<line file="dfs.py" nr="5">nr += 1</line>
```

3.4.4 Erzeugen der Ausgabe

Aus diesem Baum lassen sich jetzt die benötigten Informationen mit Hilfe von *XPath* und *XSLT* extrahieren. Über geeignete Templates läßt sich der Baum in jede beliebige andere Darstellung transformieren, zum Beispiel in eine Trace-Tabelle oder in Methodenaufrufe zur Online-Visualisierung. Das *XSLT*-Fragment aus Listing 3.10 extrahiert zum Beispiel aus dem Logfile eine *HTML Tabelle* mit den Zuweisungen für die Variable $DFSnr$.

Listing 3.10: Informations Extraktion

```
<xsl:template match="//call[@name='__setitem__' and
    arg[@name='self' and @id=$DFSnr]]">
    <html:tr>
        <html:th>
            <xsl:value-of select="arg[@name='index']/string"/>
        </html:th>
        <html:th>
            <xsl:value-of select="arg[@name='value']/number"/>
        </html:th>
    </html:tr>
</xsl:template>
```

Eine andere Möglichkeit der Ausgabe besteht darin, auf diese Weise *AnimalScript*[45] zu erzeugen, die Animationssprache von *Animal*. Diese Variante hat den Vorteil, daß man ein bereits vorhandenes System für die Ausgabe nutzen kann und man sich die aufwendige Implementierung einer eigenen Ausgabe sparen kann. Diese Vorgehensweise wurde an dieser Stelle jedoch nicht weiter verfolgt.

3.4.5 Zusammenfassung

Die soweit erzielten Ergebniss lassen sich mit den Animationssystemen aus Abschnitt 2.1 und den Programmen zur Graphenvisualisierung aus Abschnitt 2.5 vergleichen. Dabei ist folgendes festzustellen:

- Über speziell präparierte Datenstrukturen ist es möglich, interessante Ereignisse aus dem Ablauf des Algorithmus zu extrahieren. Dies

ist vergleichbar mit der Vorgehensweise von *Gato*. Ein Problem dieser Vorgehensweise besteht darin, daß Zuweisungen zu lokalen Variablen, die erst innerhalb einer Routine erfolgen, so nicht gekapselt werden können oder gekapselte Objekt sogar durch ungekapselte Objekte ersetzt werden.

- Über eine gesonderte Sprache wird spezifiziert, welche Objekte und Ereignisse visualisiert werden sollen. Im Gegensatz zu der speziellen Visualisierungssprache *Alpha* von *Leonardo* wird hier *XSLT* verwendet. Diese ist nicht auf die Spezifikation von Animationen spezialisiert, weshalb es zunächst einer umfangreichen Bibliothek bedarf, um damit in vertretbarer Zeit die erwünschten Darstellungen zu erhalten.

- Das aus einigen Animationssystemen bekannte Problem, Ereignisse nur bei Veränderungen von Speicherstellen erzeugen zu können, wird durch die Funktionweise von *Python* teilweise umgangen. Das Vergleichen und Abfragen von Variablen führt in *Python* intern zu Methodenaufrufen, die über den Wrapper abgefangen und protokolliert werden können. Allerdings können auf diese Weise nicht alle Ereignisse protokolliert werden, da der *Python*-Debugger zum Beispiel keine Aufrufe von internen Funktionen meldet.

3.5 Erweiterter XML-Trace

Für die Generierung einer Animation reichen die so gewonnenen Informationen in den meisten Fällen noch nicht aus.

- Lediglich Zugriffe auf zuvor gekapselte Objekte werden aufgezeichnet. Wenn lokale Variablen ebenfalls protokolliert werden sollen, müssen diese durch im Programmtext sichtbare Anweisungen initialisiert werden, was unter Umständen die Lesbarkeit herabsetzt.

- Die aufgezeichneten Quelltextzeilen des ausgeführten Programms sind nur schlech verwertbar. Für eine genauere Analyse müssen diese erst syntaktisch analysiert werden und in Beziehung zum umliegenden Programmcode gesetzt werden.

Aus letzterem ergab sich die Idee, den dynamischen Programmablaufbaum anstatt der Programmzeilen aufzuzeichnen. Dadurch wird es möglich, Schleifen und Abfragen als solche zu erkennen und diese Informationen gezielt zur Animation zu verwenden.

Das folgende Beispiel soll die Idee hinter dieses Konzept verdeutlichen. Listing 3.11 enthält ein kurzes Programm, daß die Summe der ersten 4 natürlichen Zahlen berechnet.

Listing 3.11: Summation von Zahlen: Python

```
sum = 0
for i in [1 2 3 4]:
  sum += i
```

Ein standard *XML*-Trace von der Ausführung ist in Listing 3.12 zu finden. Interessiert man sich zum Beispiel für die Summation im Falle $i = 3$, so läßt sich dies nur schwierig als *XPath*-Ausdruck formulieren, etwa als /line[@nr='2']/following::var[@name='i' and self='3']/following::node().

Listing 3.12: Summation von Zahlen: LOG

```
<line file="sum.py" nr="1">sum = 0</line>
<var name="sum">0</var>
<line file="sum.py" nr="2">for i in [1 2 3 4]:</line>
<var name="i">1</var>
<line file="sum.py" nr="3">  sum += i</line>
<var name="sum">1</var>
<line file="sum.py" nr="2">for i in [1 2 3 4]:</line>
<var name="i">2</var>
<line file="sum.py" nr="3">  sum += i</line>
<var name="sum">3</var>
<line file="sum.py" nr="2">for i in [1 2 3 4]:</line>
<var name="i">3</var>
<line file="sum.py" nr="3">  sum += i</line>
<var name="sum">6</var>
<line file="sum.py" nr="2">for i in [1 2 3 4]:</line>
<var name="i">4</var>
<line file="sum.py" nr="3">  sum += i</line>
<var name="sum">10</var>
```

Durch eine Integration der syntaktischen Elemente, wie sie in Listing 3.13 dargestellt ist, läßt sich der dritte Schleifendurchlauf etwa als /for/loop[3]/body schreiben.

Listing 3.13: Summation von Zahlen: XML

```
<var name="sum">0</var>
<for name="i">
 <loop>
  <head>
   <var name="i">1</var>
  </head>
  <body>
   <var name="sum">1</var>
  </body>
 </loop>
 <loop>
```

```
12   <head>
      <var name="i">2</var>
14   </head>
     <body>
16    <var name="sum">3</var>
     </body>
18  </loop>
    <loop>
20   <head>
      <var name="i">3</var>
22   </head>
     <body>
24    <var name="sum">6</var>
     </body>
26  </loop>
    <loop>
28   <head>
      <var name="i">4</var>
30   </head>
     <body>
32    <var name="sum">10</var>
     <body>
34  </loop>
</for>
```

Damit sich ein solches erweitertes Protokoll natlos mit den bereits gewonnenen Informationen aus dem "Tracer" und "Wrapper' kombinieren läßt, ist eine Übersetzung des Programms in eine *XML*-Struktur notwendig. An dieser Stelle kommen einige Eigenschaften von *Python* hilfreich zu tragen:

1. Große Teile von *Python* sind selbst wieder in *Python* geschrieben. Ähnlich wie *Java* bietet *Python* auch die Möglichkeit, zur Ausführungszeit auf die Laufzeitumgebung selbst zurückzugreifen. Insbesondere ist es in *Python* möglich, den spracheigenen Parser für die Programmanalyse zu benutzen. Auf diese Weise kann darauf verzichtet werden, selbst einen Parser schreiben zu müssen.

2. Die Sprache *Python* besteht aus nur wenigen Programmkonstrukte, die sich relativ gut in eine *XML*-Struktur überführen lassen. In den folgenden Abschnitten werden einige Vorschläge dazu gemacht, wie eine solche Abbildung aussehen könnte.

3.5.1 IF

if-Bedingungen bestehen minimal aus einem Test und einem Programmblock, können aber auch mehrere Tests enthalten.

Listing 3.14: Python Syntax: IF

```
if EXPRESSION: SUITE
elif EXPRESSION: SUITE
else: SUITE
```

Listing 3.15: XML Syntax: IF

```
<!ELEMENT if (test+,(suite|else)?)>
```

3.5.2 WHILE

while-Schleifen bestehen aus einem Eingangstest und dem Schleifenkörper. Optional kann ein weiterer Programmblock angegeben werden, der nur ausgeführt wird, wenn die Schleife nicht vorzeitig durch ein break verlassen wird.

Listing 3.16: Python Syntax: WHILE

```
while EXPRESSION: SUITE
else: SUITE
```

Listing 3.17: XML Syntax: WHILE

```
<!ELEMENT while (test,(suite,test)*,else?)>
```

3.5.3 FOR

for-Schleifen iterieren über alle Werte einer Liste. Neben der Liste und den lokalen Variablen muß der Schleifenkörper angegeben werden. Wie bei den while-Schleifen auch, kann optional ein weiterer Programmblock angegeben werden, der nur nach dem Erreichen des Endes der Liste ausgeführt wird, wenn die Schleife nicht vorzeitig durch ein break beendet wird.

Listing 3.18: Python Syntax: FOR

```
for TARGET_LIST in EXPRESSION_LIST: SUITE
else: SUITE
```

Listing 3.19: XML Syntax: FOR

```
<!ELEMENT for (loop,else?)>
<!ELEMENT loop (head,body)>
<!ELEMENT head (var+)>
<!ELEMENT body (#ANY)>
```

3.5.4 TRY

Zur Ausnahmebehandlung dienen try−except−finally-Blöcke, wobei mehrere except-Blöcke angegeben werden können und finally optional ist.

Listing 3.20: Python Syntax: TRY

```
try: SUITE
except EXCEPTION: SUITE
finally : SUITE
```

Listing 3.21: XML Syntax: TRY

```
<!ELEMENT try (suite,except?,finally?)>
```

3.5.5 Zuweisungen

Zuweisungen können neben einfachen Variablen auch Tupel beinhalten, was das Vertauschen mehrerer Werte innerhalb einer Anweisung erlaubt. Zur einfacheren Verwendung kann es sinnvoll sein, die alten Werte der Variablen vor der Zuweisung ebenfalls zu protokollieren, was aber momentan mit dem folgenden Fragment nicht vorgesehen ist.

Listing 3.22: XML Syntax: ASSIGN

```
<!ELEMENT var (#PCDATA)>
<!ATTLIST var name CDATA #REQUIRED>
```

3.5.6 Tests

Tests sind nicht auf boolsche Werte beschränkt, sondern können ähnlich wie in *C* beliebige Werte umfassen. Vergleiche sind nicht nur auf zwei Argumente beschränkt, sondern können kettenförmig fortgesetzt werden. Neben den typischen Vergleichen wie „kleinergleich“, „größer“, „ungleich“, ... können auch Elemente auf das Enthaltensein in einer Liste überprüft werden. Zudem können Objekte ihre eigenen Vergleichsoperatoren implementieren, wodurch beim Entwurf einer geeineten Abbildung hier besondere Vorsicht geboten ist. Das folgende Fragment ist daher nur als Platzhalter zu verstehen.

Listing 3.23: XML Syntax: TEST

```
<!ELEMENT test (#ANY)>
<!ATTLIST test result (0|1) #REQUIRED>
```

3.5.7 Funktionsaufrufe

Funktionsaufrufe werden wie bisher gehandhabt. Neben dem Namen der Funktion und den Parametern ist besnders vom Interesse, ob eine Funktion einen Wert zurückgelifert hat, und wenn ja, welchen. Alternativ dazu kann eine Ausnahme bei der Ausführung auftreten, die anstatt eines Rückgabewerts auftreten kann.

Listing 3.24: XML Syntax: CALL

```
<!ELEMENT call (arg*,(return|exception)?)>
<!ATTLIST call name CDATA #REQUIRED>
```

3.6 Fazit

Die in Abschnitt 3.5 angesprochenen Abbildungen konnte nicht mehr verwirklicht werden. Das Extrahieren des Syntaxbaums wurde prototypisch implementiert und für sehr kleine Beispiele getestet. Die Transformation dieser Daten in eine *XML*-Struktur und deren anschließende Verwendung erfordert allerdings noch sehr viel Arbeit und wurde nicht mehr angegangen.

Kapitel 4

Zusammenfassung und Ausblick

Die Visualisierung von Graphenalgorithmen umfasst einen sehr großen und umfangreichen Themenbereich. Im Laufe dieser Diplomarbeit wurde klar, daß innerhalb von einem halben Jahr nicht das erbracht werden kann, was andere Arbeiten zu diesem Thema über Jahrzehnte hinweg geleistet haben.

Das Einarbeiten in die unterschiedlichen Techniken (AOP, *XML*, *XSLT*, *Python*) und die Sichtung der verschiednenen, bereits existierenden Ansätze hat sehr viel Zeit verschlungen. Dabei traten immer wieder kleine Probleme auf, die häufig auch auf fehlerhafte Implementierungen in der genutzten Software zurückgeführt werden konnten. In einigen Fällen, in denen der Quellcode dazu vorlag, wurden Fehler gesucht, behoben und den Autoren mitgeteilt. In einem anderen Fall war es sogar notwendig, den vorliegenden *Java*-Bytecode mit einem Disassembler zurückzuverwandeln, um die Funktionsweise des Programms zu verstehen.

Am Ende bleibt ein eher getrübter Nachgeschmack zurück, wie wenig man einer einsetzbaren Lösung nähergekommen ist. Aus Zeitgründen beschränkt sich diese Arbeit deshalb eher auf den ersten Teil der Visualisierung, nämlich darauf, wie aus dem Ablauf eines Algorithmuses die benötigten Informationen extrahiert werden können. Die weiteren Punkte Animation und Darstellungen, wie sie in Abbildung 3.5 dargestellt sind, werde nur am Rande angegangen.

Im Gegensatz zu vielen anderen Arbeiten, in denen Algorithmen durch implizite Instrumentierungen des Quelltextes erzeugt werden, schließt sich diese Diplomarbeit der expliziten Vorgehensweise an, wie sie zum Beispiel von *Gato* und *Leonardo* verfolgt wird. Durch eine spezielle Laufzeitumgebung werden dynamisch während der Ausführung interessante Ereignisse detektiert und in eine Visualisierung umgesetzt. Im Umterschied zu *Leonardo*, das zur Zeit am weitesten fortgeschrittene System dieser Art, wird anstelle von *C* die Programmiersprache *Python* verwendet. Diese bietet als interpre-

tierte Sprache ähnlich wie *Java* spracheigene Mittel, um den Programmablauf zu überwachen und die benutzten Objekte zu inspizieren. Gleichzeitig bietet sie umfangreiche Möglichkeiten, die aus den verschiedenen Vorlesungen bekannten Algorithmen in einer ähnlich ansprechenden Form umzusetzen.

Wie sich am Ende herausstellte, reichen diese einfachen, spracheigenen Mittel jedoch nicht aus und führen nicht weit genug. In einem nächsten Schritt wäre deshalb zu untersuchen, in wie weit man die erkannten Defizite durch eine genauere Untersuchung der Algorithmen auf sprachlicher Ebene ausräumen kann.

Andere Programmiersprachen außer *Python* sind ebenfalls noch näher zu untersuchen. Die Untypisiertheit der Sprache ermöglicht es zwar, auf Typumwandlungen zu verzichten, was im Gegensatz zu der in Abschnitt 3.1.1 vorgestellten *Java*-Variante wesentlich lesbareren Code ermöglicht, aber genau diese Typinformationen können auch wertvolle Informationen liefern:

- Der Betrachter des Algorithmus kann daran erkennen, wie welche Datenstrukturen ineinander verschachtelt sind und welche Informationen darin gespeichert sind.
- Das Animationssystem benötigt diesen Informationen auf jeden Fall, um für die Datenstrukturen geeignete Darstellungen auswählen zu können.

Ein weiterer Punkt, der bis jetzt nur unzureichend betrachtet wurde, ist, wie die gewonnenen Informationen genau darzustellen sind. Kurz angesprochen wurde die Verwendung von *Animal* als Darstellungssystem, aber gerade der interaktive Betrieb erfordert weitergehende Möglichkeiten. Verschiedene andere Projekte bieten unterschiedlich flexible Komponenten an, aber die meisten davon befinden sich selber noch mitten in der Entwicklung und sind noch nicht ausgereift genug.

Anhang A

Beispiele

Im Rahmen dieser Diplomarbeit wurden mehrere Algorithmen in Python implementiert. Diese stehen als Ausgangsbasis für weitere Arbeiten zur Verfügung. Im einzelnen handelt es sich dabei um Algorithmen aus folgenden Kategorien:

A.1 Traversierung von Graphen

- Tiefensuche DFS
- Breitensuche BFS

A.2 Union-Find Problem

A.2.1 Listen basierende Verfahren

- Implementierung mit Python-Listen
- Vector mit Mengenzugehörigkeit
- Zusätzlich verkettete Listen für jede Menge

A.2.2 Baum basierende Verfahren

- Unbeschränkter Wurzelbaum
- Zusätzliche Verwendung des Rangs
- Zusätzliche Pfadverkürzung

A.3 Prioritäts Warteschlangen

- Unsortiertes Feld mit Pythen-Arrays
- Binärer Heap. Verwendet ein Feld zur Speicherung und ein weiteres Felf mit zustätzlichen Positionsangaben zum schnellen Auffinden
- Fibonacci Heap
- Verschmelzbare Linkshalde
- Linkshalde mit verzögertem Verschmelzen

A.4 Kürzeste Wege Problem

- Bellman-Ford
- Moore
- Dijkstra

A.5 Minimal aufspannende Bäume

- Boruvka
- Kruskal
- Jarnik/Prim
- Round-Robin

A.6 Flußprobleme

- Ford-Fulkerson

A.7 Sortierung

- Bubble-Sort
- Insert-Sort
- Merge-Sort
- Heap-Sort
- Quick-Sort

Anhang B

Danksagung

An dieser Stelle möchte ich mich bei all denen bedanken, sie mich während meines Studiums und vor allem während der Erstellung dieser Arbeit unterstützt haben.

- Bei Dr. Klaus Guntermann für die Betreuung der Diplomarbeit und die langanhaltende Zusammenarbeit, nicht zu vergessen das Seminar zum Thema Software-Entwicklungswerkzeuge.
- Bei Prof. Helmut Waldschmidt für die Vorlesungen auf dem Gebiet der Graphenalgorithmen, über die ich zu diesem Thema gekommen bin.
- Bei Martin Girschick für die vielen wertvollen Anregungen und Gespräche.
- Bei Linus Tovalds für das brauchbare Betriebssystem.
- Bei Dr. Ulrik Schröder, Dr. Henning Pagnia, Dr. Felix Gärtner, Falk Fraikin, Thomas Leonhardt, Christoph Müller, Jörg Treschau und all den anderen für die gute Zusammenarbeit bei den Programmierwettkämpfen.
- Bei meiner Mutter und meinem Vater für alles bis jetzt.

Anhang C

Erklärung zur Diplomarbeit

gemäß § 19 Abs. 6 DPO/AT

Hiermit versichere ich, die vorliegende Diplomarbeit ohne Hilfe Dritter mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfbehörde vorgelegen.

Darmstadt, den 28. März 2002

Philipp Matthias Hahn

Literaturverzeichnis

[1] C. Berge. Färbung von graphen, deren sämtliche bzw. ungerade kreise starr sind. *Wissenschaftliche Zeitschrift*, pages 114–115, 1961. Mathematisch-Naturwissenschaftliche Reihe.

[2] C. Berge. Perfect graphs. In D. Fulkerson, editor, *Studies in graph theory, Part I*, volume 11 of *MAA Studies in Mathematics*, pages 1–22, Washington, 1975. Mathematical Association of America.

[3] Birgit Bischof. JAnim — algorithm animation using java. Website, OSWEGO State University of New York, 1999. `http://www.cs.oswego.edu/~birgit/diplom/Anim.html`.

[4] Gilad Bracha et al. Adding generics to the java programming language. Public review draft specification, SUN Community Process, 2001. `http://developer.java.sun.com/developer/earlyAccess/adding_generics/`.

[5] F.J. Brandenburg et al. Graph template library/graphlet. Website, University of Passau, 1999. `http://infosun.fmi.uni-passau.de/Graphlet/`.

[6] Ulrik Brandes, Markus Eiglsperger, Ivan Herman, Michael Himsolt, Scott Marshall, and Sascha Meinert. Graph markup language. Website, Graph Drawing Organization, 2000. `http://www.graphdrawing.org/graphml/`.

[7] Marc H. Brown and Marc Najork. ZEUS — algorithm animation at SRC. Website, Compaq System Research Center, 1993. `http://www.research.compaq.com/SRC/zeus/home.html`.

[8] Christian Lenz Cesar. Graph foundation classes for java. Website, IBM, 1999. `http://www.alphaworks.ibm.com/tech/gfc`.

[9] James Clark et al. Xml path language (xpath) version 1.0. W3c recommendation, W3C, 1999. `http://www.w3.org/TR/xpath`.

[10] P. Crescenzi and C. H. Papadimitriou. Reversible simulation of space-bounded computations. *Theoretical Computer Science*, 143:159–165, 1995.

[11] C. Demetrescu and I. Finocchi. A technique for generating graphical abstractions of program data structures. In *Proceedings of the 3rd International Conference on Visual Information Systems (Visual'99)*, number 1614 in LNCS, pages 785–792. Springer, Amsterdam, The Netherlands, June 1999. `http://www.dis.uniroma1.it/pub/demetres/papers/visual99.ps.gz`.

[12] Camil Demetrescu and Irene Finocchi. Leonardo — a c programming environment for reversible execution and software visualization. Website, University of Rome La Sapienza", 1999. `http://www.dis.uniroma1.it/~demetres/Leonardo/`.

[13] Camil Demetrscu, Irene Finocchi, and John T. Stasko. Specifying algorithm visualizations: Interesting events or state mapping? *ALCOM-FT Technical Reports*, October 2001. `http://www.brics.dk/cgi-alcomft/db?state=reports\&id=ALCOMFT-TR-01-177`.

[14] Stephan Diehl and Andreas Kerren, editors. *Tagungsband GI-Workshop: Softwarevisualisierung SV 2000*, Technischer Bericht A01/00. FR 6.2 Informatik, Universität des Saarlandes, 2000. `http://rw4.cs.uni-sb.de/sv2000/proceed/`.

[15] Peter Donald, Craig McClanahan, Rodney Waldhoff, and James Strachan. Jakarta commons. Website, Apache Software Foundation, 2001. `http://jakarta.apache.org/commons/collections.html`.

[16] Andreas Fabri et al. *Computational Geometry-Algorithms Library.* `http://www.cgal.org/`.

[17] David C. Fallside et al. Xml schema. W3c recommendation, W3C, May 2001. `http://www.w3.org/TR/xmlschema-0/`.

[18] Christian Fremuth-Paeger. goblin — a graph object library for network programming problems. Website, University of Augsburg, 2002. `http://www.math.uni-augsburg.de/opt/goblin.html`.

[19] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. Design patterns: abstraction and reuse of object-oriented design. In Oscar M. Nierstrasz, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 707, pages 406–431, Berlin, Heidelberg, New York, Tokyo, 1993. Springer-Verlag.

[20] M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs.* Academic Press, New York, 1980.

[21] Bill Griswold, Erik Hilsdale, Jim Hugunin, Mik Kersten, Gregor Kiczales, and Jeff Palm. Aspectj-oriented programming (aop) for java. Website, XEROX Corporation, 1998. `http://aspectj.org/servlets/AJSite`.

[22] Klaus Guntermann. *tie — merge or apply WEB change files*. TH Darmstadt. UNIX manuale page.

[23] Klaus Guntermann and Wolfgang Rülling. Another approach to multiple changefiles. *TUGboat*, 7(3):134–134, October 1986.

[24] Vincent Hardy and Thierry Kormann. Batik. Website, Apache Software Foundation, 2000. `http://xml.apache.org/batik/`.

[25] Norman Hendrich. The java diagram editor. Website, University of Hamburg, 1997. `http://tech-www.informatik.uni-hamburg.de/applets/jfig/`.

[26] I. Herman and M. S. Marshall. Graph xml — an xml-based graph description format. In K. Ryall, editor, *Symposium on Graph Drawing GD'00*. Springer Verlag, 2000. `http://www.cwi.nl/InfoVisu/GraphXML/`.

[27] Michael Himsolt. Graph modeling language. Website, University of Passau, 1996. `http://www.infosun.fmi.uni-passau.de/Graphlet/GML/`.

[28] Hipke. Visualization environment for geometric algorithms. Website, University of Freiburg, 1998. `http://www.informatik.uni-freiburg.de/~hipke/Vega/`.

[29] Ric Holt, Andy Schürr, Susan Elliott Sim, and Andreas Winter. Graph exchange language, 2000. `http://www.gupro.de/GXL/`.

[30] Jim Hugunin, Barry Warsaw, Finn Bock, and Guido van Rossum others. *jython*, 1997. `http://www.jython.org/`.

[31] Jesus M. Salvo Jr. OpenJGraph — java graph and graph drawing project. `http://openjgraph.sourceforge.net/`.

[32] Paul King, Brian V. Smith, and Supoj Sutanthavibul. *XFig Drawing Program for X Window System*. `http://www.xfig.org/`.

[33] V. J. Leung, M. B. Dillencourt, and A. L. Bliss. GraphTool: A tool for interactive design and manipulation of graphs and graph algorithms. In N. Dean and G. E. Shannon, editors, *Computational Support for Discrete Mathematics*, pages 269–278. American Mathematical Society, 1994. `http://www.ics.uci.edu/~vitus/gtool/`.

[34] Chris Lilley et al. Scalable vector graphics. W3c recommendation, W3C, September 2001. http://www.w3.org/TR/SVG/.

[35] Yukihiro Matsumoto et al. *Ruby: Programmers' Best Friend*. http://www.ruby-lang.org/.

[36] Kurt Melhorn et al. Library of efficient data types and algorithms. Website, Algorithmic Solutions, 1998. http://www.algorithmic-solutions.com/as_html/products/leda/products_leda.html.

[37] Stefan Näher and Oliver Zlotowski. Ein system zur visualisierung von algorithmen und datenstrukturen mit LEDA. In Diehl and Kerren [14]. http://rw4.cs.uni-sb.de/sv2000/proceed/abstract.ps.gz.

[38] Harold Ossher. Subject-oriented programming. Website, IBM Research, 1998. http://www.research.ibm.com/sop/.

[39] Frank Peters. Visualisierung von algorithmen — systematik und bestandsaufnahme. Diplomarbeit, Carl von Ossietzky Universität Oldenburg, Oldenburg, December 2001. http://www-cg-hci.informatik.uni-oldenburg.de/~da/peters/Kalvin/Darbeit/DA-Peters.doc.

[40] W. Pierson and S. H. Rodger. Web-based Animation of Data Structures Using JAWAA. *Twenty-ninth SIGCSE Technical Symposium on Computer Science Education*, pages 267–271, 1998. http://www.cs.duke.edu/csed/jawaa/JAWAA.html.

[41] B. A. Price, R. M. Baecker, and I. S. Small. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, 1993.

[42] John Punin and Mukkai Krishnamoorthy. extensible graph markup and modeling language. Website, Rensselare Polytechnic Institute, 1999. http://www.cs.rpi.edu/~puninj/XGMML/.

[43] G.-C. Roman and K. C. Cox. A declarative program visualization system. Website, Washington University in St. Louis, 1989. http://www.cs.wustl.edu/mobilab/DeclarativeVis.html.

[44] Guido Rößling. Animation tool animal. Website, University of Siegen, 2000. http://www.informatik.uni-siegen.de/~roesslin/Animal/.

[45] Guido Rößling and Bernd Freisleben. AnimalScript: an extensible scripting language for algorithm animation. In *Proceedings of the thirty second SIGCSE technical symposium on Computer Science Education*, pages 70–74. ACM Press, 2001. http://www.informatik.uni-siegen.de/~roesslin/publications/SIGCSE2001.pdf.

[46] Alexander Schliep and Winfried Hochstättler. Graph animation toolbox. Website, University of Cologne, 1998. `http://www.zpr.uni-koeln.de/~gato/`.

[47] Jeremy Siek. Generic graph component library. Website, University of Notre Dame, 2000. `http://www.boost.org/libs/graph/doc/table_of_contents.html`.

[48] www.softwarevisualisierung.de. `http://www.softwarevisualisierung.de/`.

[49] John T. Stasko. Samba/Polka/XTango. Website, Georgia Institute of Technology, 1990. `http://www.cc.gatech.edu/gvu/softviz/algoanim/algoanim.html`.

[50] P. Tarr, H. Ossher, W. Harrison, and S.M. Sutton. Multi-dimensional separation of concerns: Software engineering using hyperspaces. Website, IBM Research, 1998. `http://www.research.ibm.com/hyperspace/`.

[51] James Tauber et al. Formatting objects. Website, Apache Software Foundation, 1999. `http://xml.apache.org/fop/`.

[52] Guido van Rossum. *The Python Language*, 1991. `http://www.python.org/`.

[53] Helmut Waldschmidt. Grundzüge der informatik III. Vorlesung, TH Darmstadt, 1993.

[54] Helmut Waldschmidt. Graphenalgorithmen. Vorlesung, TU Darmstadt, 1999. `http://www-sp.iti.informatik.tu-darmstadt.de/lehre/graphenalgorithmen/`.

[55] Roland Wiese, Markus Eiglsperger, and Peter Schabert. yFiles. Website, University of Tübingen, 1999. `http://www-pr.informatik.uni-tuebingen.de/yfiles/`.

[56] Andreas Zeller. Datenstrukturen visualisieren und animieren mit DDD. In Diehl and Kerren [14]. `http://rw4.cs.uni-sb.de/sv2000/proceed/sv.ps.gz`.