平成30年度 学士論文

データ構造の関係性を 視認しやすくする 自動グラフレイアウト手法

東京工業大学 理学部 情報科学科
学籍番号 15_03230
小串 智希

指導教員

増原 英彦 教授

平成31年2月28日

概要

プログラミング環境を改善する研究の一つにライブプログラミング環境がある。ライブプログラミング環境は、プログラムを編集すると実行結果を即座にフィードバックする。これによってプログラマは実行結果の確認とプログラムの編集を同時に行うことができる。

ライブプログラミングにおけるデータ構造の関係を図示する研究に、岡らの Kanon [1] がある。Kanon はコードを実行したときに作られるデータ構造を用いて、プログラムの編集後即座にデータ同士の参照関係を図示する。

現在の Kanon によるデータ構造のレイアウトは必ずしもデータ構造をプログラマが視認しやすいような形にレイアウトできないため、本研究はオブジェクトの集合が与えられるとそれらの画面上での配置を決定する新しいレイアウト手法を提案した。これは、エッジの角度に働く力を導入した力学的手法によってデータ構造をグラフとしてレイアウトするアルゴリズムである。ライブプログラミングのために、入力はオブジェクト群とその参照関係のみを使用し、その他のプログラマからの指示は必要としない。また、配置はデータ構造の関係がプログラマにとって分かりやすい、つまりデータの参照関係とノードの配置に整合性があり且つグラフとして見やすいレイアウトを目指したものである。

本研究で提案したアルゴリズムは、既存のデータ構造レイアウトアルゴリズムと比べて、エッジの角度を自動決定するため、データ構造の種類に関する汎用性が向上した。また、識別子に依存することなくレイアウトすることができ、さらにデータ構造が入れ子になっている場合でも比較的視認しやすい形でレイアウトすることができた。

謝辞

　本研究を進めるにあたり、多くのアドバイスやご指導をしていただいた増原英彦教授、青谷知幸助教、ならびに研究室の皆様に心より感謝いたします。

目 次

第 1 章　導入 **8**

第 2 章　研究背景 **10**
2.1 ライブプログラミング 10
2.1.1 ライブプログラミングの例 10
2.2 ライブプログラミングの課題 12
2.3 ライブプログラミングでのデータ構造の可視化 12
2.3.1 データ構造を可視化するライブプログラミングの例 12
2.3.2 ライブプログラミングでのデータ構造レイアウトに求められる性質 13
2.4 グラフレイアウト手法 14
2.4.1 グラフ 14
2.4.2 グラフレイアウト手法 14
2.4.3 グラフレイアウト手法の例 15

第 3 章　課題・目的 **17**
3.1 ライブプログラミングでのデータ構造可視化の課題 17
3.1.1 既存のデータ構造レイアウト手法の問題点 17

第 4 章　提案 **21**
4.1 概要 21
4.2 提案手法 22
4.2.1 角度決定のアイデア 22
4.2.2 角度を指定した力学的手法によるグラフレアウトの提案 22

第 5 章　実装方法 **25**
5.1 全体の流れ 25
5.1.1 エッジの角度決定まで 25
5.1.2 力学的手法（Force-directed Method）によるノードの座標決定 36
5.2 例外的な処理 41
5.2.1 循環のあるグラフ構造について 41

5.2.2 配列について 44
5.2.3 オブジェクトの色分けについて 45

第 6 章 実験・提案手法の評価 **48**
6.1 グラフレイアウトについて 48
6.2 計算時間について 53

第 7 章 関連研究 **55**
7.1 ライブプログラミング環境 55
7.2 データ構造の可視化 56
7.3 グラフレイアウトアルゴリズム 56

第 8 章 まとめ・課題 **58**
8.1 まとめ 58
8.2 課題 58
8.2.1 計算時間の向上 58
8.2.2 ユーザー実験 58
8.2.3 相対角度の導入 59

図目次

2.1 YinYang 環境 [2] の図 11
2.2 Live Editor [3] で表示される図 11
2.3 Python Tutor [10] で表示される図 13
2.4 Kanon [1] で表示される図 13
2.5 Kamada-Kawai アルゴリズム [7] によって描画されたグラフの図 15
2.6 GraphViz [6] によって描画された階層グラフの図 16

3.1 リストのレイアウト結果 19
3.2 二分木のレイアウト結果 19
3.3 三分木のレイアウト結果 19
3.4 Tree のレイアウト 19
3.5 同じ Tree を、クラス名とフィールド名を変えてレイアウトしたもの 19
3.6 二分木をノード要素として持つリストのレイアウト結果 . . 20

4.1 概要図 21
4.2 エッジの角度の決定方法 22
4.3 $\frac{\pi}{3}$ の角度を持ったエッジに働く角度力 23

5.1 色無しのレイアウト 45
5.2 色有りのレイアウト。視認性が向上する 45

6.1 Kanon でのリストレイアウト 49
6.2 提案アルゴリズムでのリストレイアウト 49
6.3 Kanon での二分木レイアウト 49
6.4 提案アルゴリズムでの二分木レイアウト 49
6.5 Kanon での三分木レイアウト 50
6.6 提案アルゴリズムでの三分木レイアウト 50
6.7 Kanon での循環リストレイアウト 50
6.8 提案アルゴリズムでの循環リストレイアウト 50
6.9 Kanon での二分木を要素として持つリストのレイアウト . 51

6.10 提案アルゴリズムでの二分木を要素として持つリストのレイアウト 51
6.11 Kanon でのリストを要素として持つ二分木のレイアウト . 51
6.12 提案アルゴリズムでのリストを要素として持つ二分木のレイアウト 51
6.13 Kanon での二分木の配列を要素として持つリストのレイアウト 52
6.14 提案アルゴリズムでの二分木の配列を要素として持つリストのレイアウト 52
6.15 Kanon での環境図のレイアウト 52
6.16 提案アルゴリズムでの環境図のレイアウト 52
6.17 ノード数が多くなった二分木のレイアウト結果 53
6.18 ノード数と objectDraw の実行にかかった時間 54

表 目 次

6.1　ノード数と objectDraw の実行にかかった時間 54

第1章　導入

ソフトウェア開発において、開発作業を効率よく行うために様々な研究・開発がなされている。例えば、プログラムを自動生成するプログラミング合成の研究や、より効率的にコードを生成するためのコンパイラの研究などがある。また、プログラミング環境の改善の研究として統合開発環境（Intergrated Development Environment, IDE）などがある。

プログラミング環境の改善を目的とする研究の一つに、ライブプログラミング環境がある。ライブプログラミング環境は、プログラムを編集すると実行結果・実行の変化を即座にフィードバックする。これによってプログラマは実行結果の確認とプログラムの編集を同時に行うことができる。

既存のライブプログラミングには課題が多くあり、そのため実用的なプログラムの開発環境で使用されているものは多くない。例えば、実行に時間のかかるプログラムでフィードバックの高速性を保つのが難しいこと、データ構造の参照関係を図表現として十分に表現できるものが無いこと、などが挙げられる。

ライブプログラミングにおけるデータ構造の関係を図示する研究に、J.Guo らの Python Tutor [10] や岡らの Kanon [1] がある。これらはデータ構造を図表現として可視化するライブプログラミング環境である。

本研究では、ライブプログラミング環境でデータ構造をどのようにレイアウトすべきか、という点に注目する。

ライブプログラミング環境でのデータ構造レイアウトに求められる性質として、

- プログラマの余計な入力の手間がいらない
- データ構造の関係性がプログラマにとって分かりやすいように表示される
- データ構造の種類に関する汎用性が高い

の 3 つが考えられる。しかし、既存のデータ構造レイアウト手法ではこれらの性質を十分に満たせていない。

本研究では上記の問題点を解決するために、オブジェクトの集合が与えられるとそれらの画面上での配置を決定するレイアウト手法を提案する。

ライブプログラミングのために、入力はオブジェクト群とその参照関係のみを使用し、その他のプログラマからの指示は必要としない。また、配置はデータ構造の関係がプログラマにとって分かりやすい、つまりデータの参照関係とノードの配置に整合性があり且つグラフとして見やすいレイアウトを目指す。

第2章　研究背景

2.1　ライブプログラミング

通常のプログラミング環境ではプログラマは「コードを編集」→「コンパイル」→「実行し結果を確認」→「コードを編集」→……といった一連の実行の流れを繰り返す。この流れは「Edit-Compile-Run サイクル」と呼ばれている。

Edit-Complile-Run サイクルはコードを編集する行為、コンパイルする行為、実行する行為のそれぞれがフェーズとして大きく分断されているため、コードの編集から実行結果の確認までに待ち時間が発生してしまう問題点がある。

この問題点を解決するのがライブプログラミングである。ライブプログラミング環境はプログラムを編集すると即座にプログラムの実行結果や実行時の情報をプログラマに提示する。そのため、プログラマは常に実行結果が正しい挙動を見せているかを確認しながらプログラムを編集することができる。

2.1.1　ライブプログラミングの例

2.1.1.1　YingYang

YingYang [2] はライブプログラミングを可能にする言語、またはその環境のことである。

YinYang では Probing と Tracing という二つの機能を使い、式や変数の値を画面に表示する。

2.1.1.2　Live Editor

Live Editor [3] は JavaScript およびそのライブラリである Processing.js のライブプログラミング環境である。

Live Editor でプログラムを編集すると、描画に関する関数によって描かれる図が表示画面に描画される。

```
@sqrt(25.0);
5.015247
def sqrt(x : float) = {
  var y = x;
  var t = 4;
  while (t > 0) do {
    t = t - 1;
    prn(" t "++t ++ " y " ++ y);
    y = y / 2.0 + x / (2.0 * y)
  };
  @y;
  5.015247
}
```

```
t 3 y 25
t 2 y 13
t 1 y 7.461538
t 0 y 5.406027
```

図 2.1: YinYang 環境 [2] の図

Live Editor Example

```
1 fill(255, 0, 0);
2 rect(10, 10, 100, 100);
3
4 fill(0, 255, 0);
5 ellipse(150, 80, 40, 40);
```

図 2.2: Live Editor [3] で表示される図

2.2　ライブプログラミングの課題

既存のライブプログラミングにはいくつかの問題点が指摘されている。以下にそのうちの一部を紹介する。

一つ目は、実行に時間のかかるプログラムでは編集してから即座に結果をフィードバックできない点である。

二つ目は、実用的なプログラミングで使用されているものが少ない点である。一つ目の問題点より、ライブプログラミングによって高速に結果がフィードバックされるプログラムは限定的である。そのため、教育目的のライブプログラミング環境などはあるが業務に使われるようなライブプログラミング環境はほとんど存在しない。

三つ目は、データ構造の扱いが十分でない点である。Kanon が開発される以前のライブプログラミング環境には実行結果として画像を出力するものや、数値や文字列を出力するものなどが存在していたが、データ構造の参照関係を実行結果として表示するライブプログラミング環境は存在しなかった。

2.3　ライブプログラミングでのデータ構造の可視化

既存のプログラミング環境では、例えばデータ構造に対する操作を記述しているときに操作による変化を観察するためには、プログラマが操作の前後に出力文を挿入しなければならず、また操作による変化が正しいことを確認するためには、出力された文字列からデータの構造を想像した上で変化を読み取らなければならない。そのためプログラマの負担が大きい。

これより、ライブプログラミング環境でのデータ構造の関係の表示はプログラマにとって大きな意味を持つ。

2.3.1　データ構造を可視化するライブプログラミングの例

2.3.1.1　Python Tutor

Python Tutor [10] は Web ブラウザ上で動作する Python のためのプログラム可視化ツールである。記述したコードを 1 ステップごとにどのような内部状態になっているかを図表現で可視化する。Java、JavaScript、TypeScript、Ruby、C、C++など幅広い言語に対応しており、さらに Python 及び JavaScript にはライブプログラミング機能も備わっている。

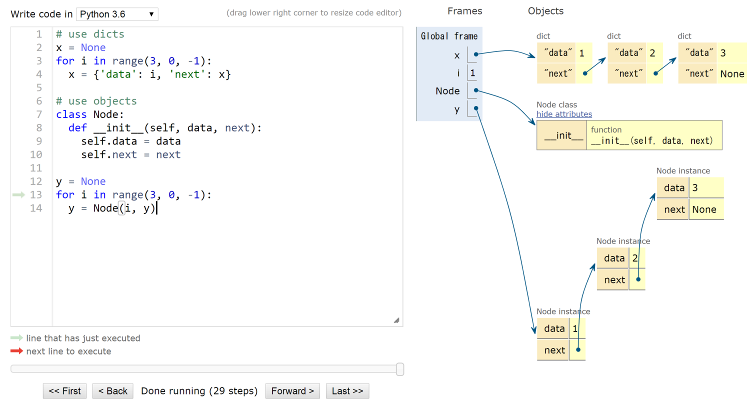

図 2.3: Python Tutor [10] で表示される図

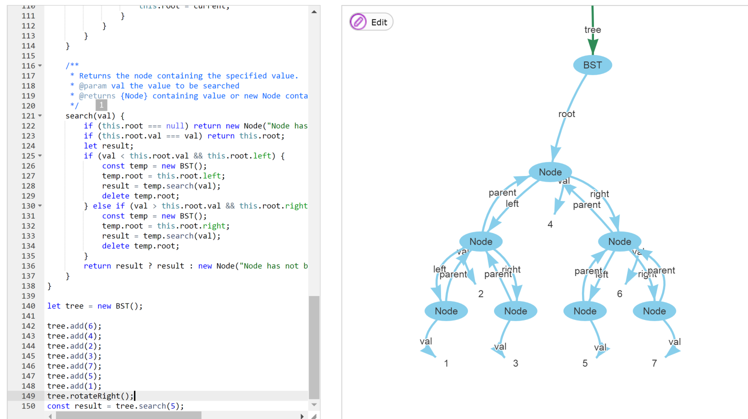

図 2.4: Kanon [1] で表示される図

2.3.1.2　Kanon

Kanon [1] は、データ構造を図表現として可視化するライブプログラミング環境である。JavaScript 言語を対象に、Khan Academy の Live Editor [3] を拡張して設計されている。

Kanon は、プログラムが編集される度にプログラムを実行し、その途中で生成されたオブジェクト及び参照関係を図表現として可視化する。

2.3.2　ライブプログラミングでのデータ構造レイアウトに求められる性質

プログラマがライブプログラミング環境でのデータ構造のレイアウトの性質に求めるものには以下のようなものが考えられる。

(1) プログラマの余計な入力の手間がいらない

(2) データ構造の関係性がプログラマにとって分かりやすいように表示される

(3) データ構造の種類に関する汎用性が高いこと

データ構造の種類や図の表示の仕方などをプログラマが一つ一つ指示するのはプログラマにとって負担である。よって（1）が考えられる。

また例えば、リスト構造を図示したときにオブジェクトを表すノードが一直線上に並んでいないと、プログラマはリストのどこにどの値が格納されているかが分かりづらくなる。ライブプログラミングで図を表示させる目的はプログラマを補助することなので、プログラマにとって分かりやすい図が表示されることはとても重要である。よって（2）が考えられる。

また、同様の理由により表示することのできるデータ構造の種類が少ないとプログラマの補助にならないので（3）が考えられる。

2.4　グラフレイアウト手法

2.4.1　グラフ

グラフとは、あるノードの集合と、そのノード間を結ぶエッジの集合のことである。

実際のアプリケーションでは、コンピューター・ネットワーク、ソフトウェア・プログラム構造、プロジェクト管理図など様々なものをモデル化するためにグラフがよく使用される。そのため、2 つのノードの最短パスを見つけるメソッドや最小コストのパスを見つけるメソッドなど、様々なグラフ理論の定理をアプリケーションに利用することができる。

2.4.2　グラフレイアウト手法

グラフを綺麗にレイアウトするためのアルゴリズムは多く研究・開発されており、様々なアプリケーションのニーズに対応している。

どのようなレイアウトが「優れている」と考えるかの基準はユーザーによって異なってくるが、全ての基準に共通する一つの目標として「ノード間同士の関係をどれだけ理解しやすいか」が挙げられる。

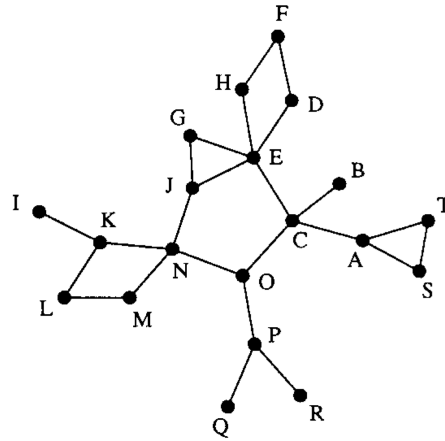

図 2.5: Kamada-Kawai アルゴリズム [7] によって描画されたグラフの図

2.4.3　グラフレイアウト手法の例

2.4.3.1　力学的手法を用いたレイアウト

力学的手法を用いたレイアウトは、ノードに仮想的な力を働かせ、エネルギーが最小になるようなノードの座標を見つけてレイアウトするアルゴリズムである。

Kamada-Kawai アルゴリズム [7] は力学的手法を用いたレイアウトアルゴリズムの一つであり、各ノードをばねで繋いだものと仮定してエネルギーが最小となるノード座標を見つけレイアウトする。

2.4.3.2　階層グラフレイアウト

階層グラフレイアウトは、ノードを各階層に割り振ってレイアウトするアルゴリズムである。Sugiyama フレームワーク [5] は階層グラフレイアウトアルゴリズムの一つであり、(1) 階層割当、(2) 交差削減、(3) 座標割当、の 3 つの手順を踏むことで階層グラフをレイアウトする。

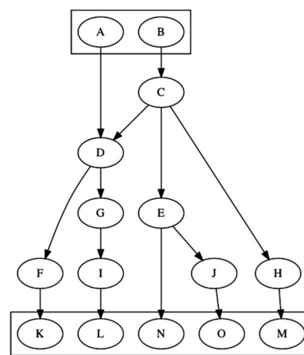

図 2.6: GraphViz [6] によって描画された階層グラフの図

第3章　課題・目的

3.1　ライブプログラミングでのデータ構造可視化の課題

2.3.2 節で述べたように、プログラマがライブプログラミング環境でのデータ構造のレイアウトに求める性質には以下のようなものがある。

(1) プログラマの余計な入力の手間がいらない

(2) データ構造の関係性がプログラマにとって分かりやすいように表示される

(3) データ構造の種類に関する汎用性が高いこと

しかし、既存のデータ構造レイアウト手法ではこれらの性質を十分に満たせていないと考える。

3.1.1　既存のデータ構造レイアウト手法の問題点

既存のデータ構造レイアウト手法には以下の 3 つが考えられる。

(1) 力学的手法を用いたグラフレイアウト

(2) 階層グラフレイアウト

(3) データ構造の種類ごとに異なるレイアウトを使用するもの

これらの手法にはそれぞれに問題点がある。

3.1.1.1　力学的手法レイアウト・階層レイアウトの問題点

力学的手法を用いたグラフレイアウトはデータ構造の種類に関する汎用性が高くなるが、エッジの情報、つまりフィールドの情報が無視されてしまう欠点がある。これは、レイアウトされたデータ構造の関係性の分かりにくさに繋がる。

階層グラフレイアウトは木構造などの特定の構造を性質が分かりやすい形にレイアウトすることができるが、データ構造の種類に関する汎用性が低いという欠点がある。

3.1.1.2　データ構造の種類ごとに異なるレイアウトを使用する場合の問題点

データ構造の種類ごとに異なるレイアウトを使用するツールに Kanon がある。

Kanon は、List や Tree などのデータ構造をプログラムの編集後即座に表示し、また編集前と編集後で構造のどの部分が変化したのかをプログラマーが確認しやすいような機能を搭載している。しかし Kanon では、データ構造がプログラマにとって視認しやすい形にレイアウトされる場合とそうでない場合がある。そこで、Kanon が持つレイアウトの問題点を以下の 3 つに分類する。

3.1.1.2.1　データ構造の種類に関する汎用性の欠如

現在、Kanon ではリスト構造と二分木構造のみに対応しており、この 2 種類では整合性をもってレイアウトできるが、それ以外のデータ構造では必ずしも視認しやすいような形にレイアウトできない。

例えば、三分木（Trie）などはプログラマが視認しやすい形にはレイアウトされない。

3.1.1.2.2　識別子に依存したレイアウトの決定

Kanon は、オブジェクトのクラス名やフィールド名が特定の文字列と一致するかどうかでデータ構造の種類を判断する。そのため、オブジェクト間の参照関係が同じであっても定義元のクラス名やフィールド名を変えると元のレイアウトが崩れてしまう場合がある。

3.1.1.2.3　入れ子構造の非対応

あるデータ構造のノードが別のデータ構造のノードを要素として持つと、単体で描画されるときに持つ視認しやすい形が失われてしまう。

例えば、二分木をノード要素として持つリストを描画すると、リストノードの一直線に並ぶという性質が失われてしまう。

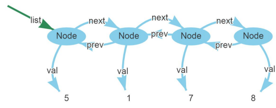

図 3.1: リストのレイアウト結果

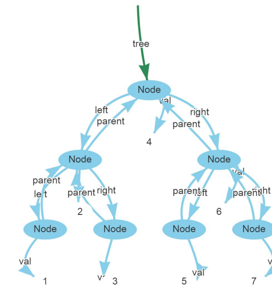

図 3.2: 二分木のレイアウト結果

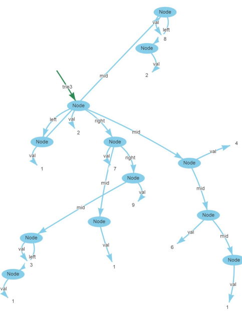

図 3.3: 三分木のレイアウト結果

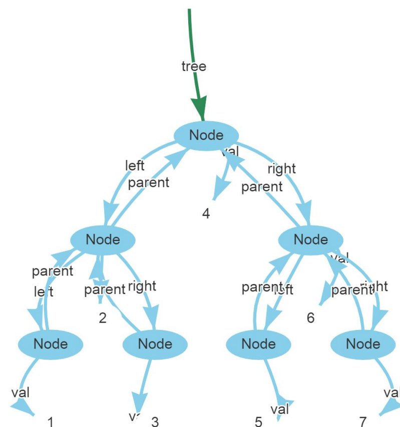

図 3.4: Tree のレイアウト

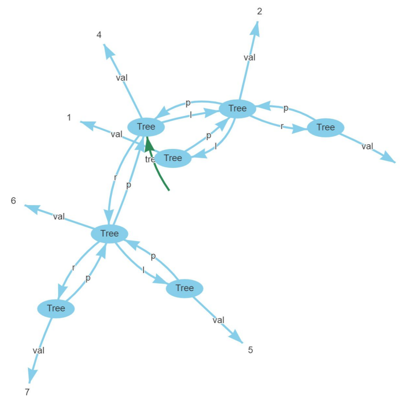

図 3.5: 同じ Tree を、クラス名とフィールド名を変えてレイアウトしたもの

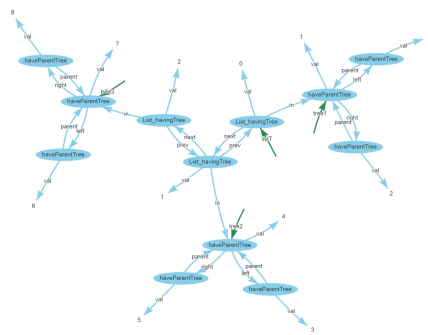

図 3.6: 二分木をノード要素として持つリストのレイアウト結果

第4章　提案

我々は、エッジの角度に働く力を導入した力学的手法によるデータ構造レイアウトアルゴリズムを提案する。

このアルゴリズムは、オブジェクトの集合が与えられるとそれらの画面上での配置を決定するものである。ライブプログラミングで使用するため、入力はオブジェクト群とその参照関係のみとし、その他のプログラマからの指示は必要としない。またこのアルゴリズムは、データ構造の関係がプログラマにとって分かりやすい、つまりデータの参照関係とノードの配置に整合性があり、且つグラフとして見やすいレイアウトを目指す。

4.1　概要

各オブジェクトの参照関係からグラフエッジの角度を決定する。決められたエッジの角度を元に、力学的手法（Force-directed Method）により各ノードの座標を求める。

力学的手法の詳細については 4.2.1 節、エッジの角度の決め方については 4.2.2 節で説明する。

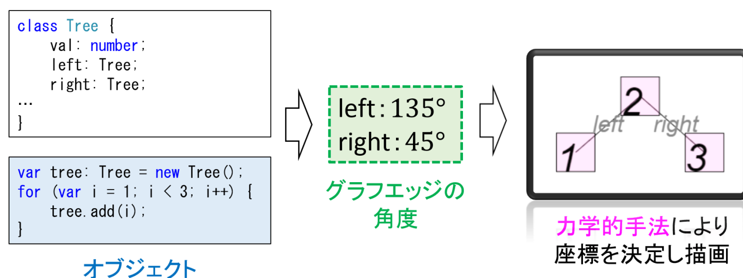

図 4.1: 概要図

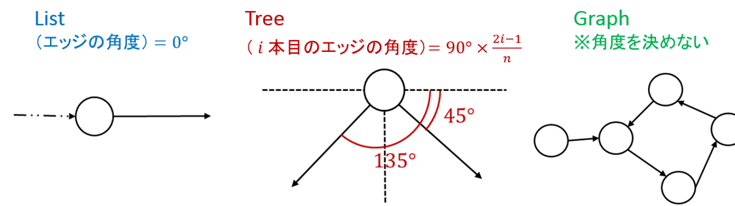

図 4.2: エッジの角度の決定方法

4.2　提案手法

4.2.1　角度決定のアイデア

プログラマの余計な入力の手間を省くため、エッジの角度をオブジェクトの参照関係のみから自動決定する方法を提案する。

まずはオブジェクト群を、クラス毎に分ける。

次に各クラスを、オブジェクトの参照関係から 3 つのグループに分類する。一つ目は一本木（List）構造、二つ目は n 分木（Tree）構造、三つ目は閉路のあるグラフ構造である。それぞれのグループによって、エッジの角度の決め方を変える。

一本木構造では、各オブジェクトから次のオブジェクトへ伸びているフィールドの数は 1 つのはずなので、これらのエッジの角度を全て 0° にする。

n 分木構造では、各オブジェクトから次のオブジェクトへ最大 n 本のフィールドが伸びているはずなので、i 本目のエッジの角度を $90° \times \frac{2i-1}{n}$ とする。

閉路のあるグラフ構造では、角度を決めないようにし、力学的手法を用いる際に角度力を働かせないようにする。

また、異なるオブジェクト集合を結ぶエッジの角度については、n 分木構造と同じように i 本目を $90° \times \frac{2i-1}{n}$ とする。ここでの n は、そのオブジェクトから異なるクラスのオブジェクトはいくつ参照されているかを表す。

4.2.2　角度を指定した力学的手法によるグラフレアウトの提案

上記のアルゴリズムを実現させるため、我々は角度を指定した力学的手法によるグラフレイアウトを提案する。

力学モデルの一つである Fruchterman-Reingold アルゴリズム [8] は各ノードに引力と斥力を働かせてノード座標を動かしていくが、我々はここに**角度力**を加えたモデルを提案する。

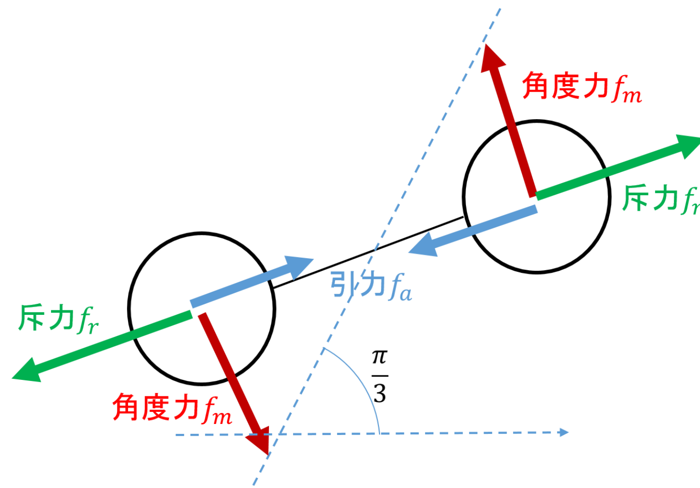

図 4.3: $\frac{\pi}{3}$ の角度を持ったエッジに働く角度力

4.2.2.1　角度力

グラフ $G = (V, E)$ の辺集合 E の要素 $e \in E$ は 2 つの端点の集合 (v_1, v_2) の形で表されるが、我々はエッジに端点を結んだ線分の偏角の情報を追加する。角度力とは、このエッジの偏角に依存して働く力である。具体的には、角度情報の与えられたエッジ (v_1, v_2, θ) について、ベクトル $\vec{v_2} - \vec{v_1}$ と x 軸のなす角が ϕ だったとき、v_1 と v_2 には $\theta - \phi$ の 2 乗に比例する力のベクトルが $\vec{v_2} - \vec{v_1}$ と垂直の方向に働く。この 2 つの力ベクトルを角度力と呼ぶ。また、(v_1, v_2, θ) の組み合わせのことを角度付きエッジと呼ぶ。

4.2.2.2　角度を指定した力学的手法によるグラフレイアウト

各ノードには斥力 f_r、引力 f_a、角度力 f_m の 3 つを働かせる。斥力は自分以外の全てのノード、引力は隣接ノードからのみ受ける。

斥力、引力は以下の式で定義する。

$$f_r = -\frac{k^8}{d^3}$$

$$f_a = \frac{d^2}{k}$$

k は以下の式で定義する。

$$k = c\frac{|V|}{d_{max}}$$

d はノード間の距離、c は定数、$|V|$ はノードの個数、d_{max} とはグラフのノード同士の最短経路長の最大値である。これは Floyd-Warshall 法などで求めることができる。

計算した斥力・引力・角度力を元に頂点を動かす。このときに重要なこととして、動かす変位の大きさは温度パラメータ t 以下にすることである。

全てのノードを動かしたら温度パラメータを下げ、また斥力・引力・角度力を計算する。これを温度パラメータが 0 を下回るまで繰り返す。

第5章　実装方法

本章では、角度を指定した力学的手法によるグラフレイアウトアルゴリズムの実装及び、エッジの角度の自動決定アルゴリズムの実装の方法、並びに工夫した点を述べる。提案アルゴリズムのソースコードは 14 行の HTML と約 1100 行の TypeScript で構成されている。

5.1　全体の流れ

関数 objectDraw にレイアウトしたいデータ構造のオブジェクトと、具体的な値を表示させたいフィールド名を入力として渡すと自動でグラフをレイアウトする。

objectDraw にオブジェクトを渡すと、グラフレイアウトに必要なフィールドを全て収集し、自動で角度の割り振りを行う。エッジに割り振られた角度を元に角度付きエッジリストを生成し、力学的手法（Force-directed Method）によって各ノードの座標を決定しレイアウトする。

5.1.1　エッジの角度決定まで

グラフをレイアウトする際には、

(1) 角度付きエッジリスト

(2) 各エッジのフィールド名

(3) ノードの値として表示させる number（or string, boolean）

(4) 各ノードのクラス名

が必要になる。これらは allObjectNumbering と allNumberInNode の 2 つの関数によって与えられる。

ソースコード 5.1: グラフレイアウトに必要な変数

```
1 function objectDraw(obj: Object, ...pname: string[]) {
2     ...
```

```
3     var GRAPHTEXT: number[] = new Array();  //角度付きエッ
          ジリスト
4     var ARRAYFIELD: string[] = new Array();   //各エッジ
          のフィールド名
5     var DOTNUMBER: number = allObjectNumbering(obj, 0,
          DRAW_CIRCLE, GRAPHTEXT, ARRAYFIELD) + 1;
6     var EDGENUMBER: number = Object.keys(GRAPHTEXT).length
           / 3;
7     var ARRAYTEXT: number[] = new Array();  //ノードの値
          として表示させるnumber(or string, boolean)
8     var ARRAYCLS: string[] = new Array();    //各ノードの
          クラス名
9     allNumberInNode(obj, ARRAYTEXT, ARRAYCLS, ...pname);
10    ...
11 }
```

5.1.1.1 allObjectNumbering

関数 allObjectNumbering では内部で objectNumbering という別の関数を使用する。

関数 objectNumbering では、オブジェクト obj、現在のノード番号 num、角度付きエッジリストを書きこんでいく配列 text、各エッジのフィールド名を書きこんでいく配列 text2 が入力として与えられ、ノードの個数を返り値として返す。

まずはじめに objectNumbering はオブジェクトグラフに循環があるかどうかを判定する。オブジェクトグラフに循環が無かった場合、objectNumbering は与えられたオブジェクトのフィールドから、そのオブジェクトと同じ型のフィールドを辿っていき、各オブジェクトに番号を振っていきながらエッジリストを text に追加していく。

またこのとき、同じ型のオブジェクトを辿っていく中で再び同じオブジェクトにたどり着いて無限ループに陥ってしまう危険性を回避するために、一度辿ったオブジェクトを記憶しておく配列 arrayObj を用意しておく。新しいオブジェクトにたどり着くたびにそのオブジェクトを arrayObj に書きこんでいく。

ソースコード 5.2: objectNumbering

```
1 //データ構造をグラフに描画するために必要なエッジリストを配
      列text に書きこんでいく、返り値はノードの個数
```

```
2 function objectNumbering(obj: Object, num: number,
    drawcircle: boolean, text: number[], text2: string[],
     fldArray: string[]): number {
3     var arrayObj: Object[] = new Array();
4     var arrayProp: string[] = new Array();
5     var unnecessaryProp: string[] = new Array();
6     identify_unnecessaryProperty(obj, unnecessaryProp);
7     enum_sameProperty(obj, arrayProp);
8     var newArray: string[] = arrayProp.filter(n =>
        unnecessaryProp.indexOf(n) == -1);
9     var ntree: number = Object.keys(newArray).length;
10
11    //補助関数
12    function objectNumbering_sub(obj: Object, num: number,
         arrayObj: Object[], fldArray: string[],
        unnecessaryProp: string[], ntree: number, text:
        number[], text2: string[]) {
13
14        if (isgraph && drawcircle == true) { //グラフ構造
            の場合
15          ...
16        } else { //木構造の場合
17            var cls: Function = obj.constructor;
18            var n: number = 0;
19            arrayObj[Object.keys(arrayObj).length] = obj;
20            var dotnum: number = num;
21
22            for (var prop in obj) {
23                if (obj[prop] instanceof cls) {
24                    if (!sameObject_inArray(obj[prop],
                        arrayObj)) {
25                        ...
26                    }
27                }
28            }
29
30            return dotnum;
31        }
32    }
33
34    var renum: number = objectNumbering_sub(obj, num,
        arrayObj, fldArray, unnecessaryProp, ntree, text,
        text2);
35    return renum;
36 }
```

角度付きエッジリストを書きこんでいく text には text[3∗i] に始点のノード番号を、text[3∗i+1] に終点のノード番号を、text[3∗i+2] にエッジの偏角を弧度法 ($-\pi \sim \pi$) で書きこむ。このとき、関数 enum_sameProperty と関数 identify_unnecessaryProp で予めレイアウトに必要なフィールド名とそうでないフィールド名を羅列している。

例えば、Tree クラスに left・right・parent の 3 つのフィールドがあった場合、parent フィールドは必ず left フィールドか right フィールドの逆向きのフィールドとなっているため角度を割り当てる必要がない。このとき、関数 enum_sameProperty では left・right・parent の 3 つのフィールドが羅列され、関数 identify_unnecessaryProp では parent フィールドが羅列される。

関数 identify_unnecessaryProp ではオブジェクトグラフ内のフィールドの中で、互いに逆方向を向き合っているフィールドの組を全て列挙し、その中の最頻値を不必要なフィールドとして返り値に渡す。

ソースコード 5.3: enum_sameProperty

```
1  //自身の型と同じであるプロパティ名を全て列挙し,配列に保存し
       ておく
2  function enum_sameProperty(obj: Object, arrayProp: string
       []) {
3      var arrayObj: Object[] = new Array();
4
5      //補助関数
6      function enum_sameProperty_sub(obj: Object, arrayProp:
            string[], arrayObj: Object[]) {
7          var cls: Function = obj.constructor;
8          arrayObj[Object.keys(arrayObj).length] = obj;
9
10         for (var prop in obj) {
11             if (obj[prop] instanceof cls) {
12                 if (!sameName_inArray(prop, arrayProp)) {
13                     arrayProp[Object.keys(arrayProp).
                           length] = prop;
14                 }
15                 if (!sameObject_inArray(obj[prop],
                       arrayObj)) {
16                     enum_sameProperty_sub(obj[prop],
                           arrayProp, arrayObj);
17                 }
18             }
19         }
20     }
```

```
21
22     enum_sameProperty_sub(obj, arrayProp, arrayObj);
23 }
```

ソースコード 5.4: identify_unnecessaryProperty

```
1 //オブジェクト内のプロパティの中から、常に他のプロパティと逆
      方向になっているプロパティの名前を返す
2 function identify_unnecessaryProperty(obj: Object,
      unnecessaryProp: string[]) {
3     var arrayObj: Object[] = new Array();
4     var arrayProp: string[] = new Array();
5
6     identify_unnecessaryProperty_sub(obj, arrayObj,
          arrayProp);
7     identify_unnecessaryPropertySet_sub(arrayProp,
          unnecessaryProp);
8
9     //補助関数、不要なプロパティ名の集合を求める
10    function identify_unnecessaryPropertySet_sub(arrayProp
          : string[], unnecessaryProp: string[]) {
11        if (Object.keys(arrayProp).length != 0) {
12            var str: string = mode_inArray(arrayProp);  //
                  ここで最頻値を発見する
13            unnecessaryProp[Object.keys(unnecessaryProp).
                  length] = str;
14            ...
15
16            identify_unnecessaryPropertySet_sub(arrayProp,
                  unnecessaryProp);
17        }
18    }
19
20    //補助関数
21    function identify_unnecessaryProperty_sub(obj: Object,
           arrayObj: Object[], arrayProp: string[]) {
22        ...
23    }
24
25    //補助関数２
26    function string_in_setOfString(str: string, str1:
          string, str2: string): boolean {
27        return str == str1 || str == str2;
28    }
29 }
```

---

enum_sameProperty で羅列した要素数から identify_unnecessaryProperty で羅列した要素数を引いた数（:=ntree とする）によって各フィールド名に割り当てる角度を決定する。このときの角度の決め方は、第 4.2.2 節で述べた「角度決定のアイデア」に従う。ntree = 1 の場合は角度を全て 0 とし、ntree≥2 の場合は、i 本目のエッジの角度を $\frac{\pi}{2} \times \frac{2i-1}{n}$ とする。

また、辿ったフィールドが identify_unnecessaryProperty で羅列したフィールド名だった場合、そのフィールドの逆向きフィールドに割り当てられた角度を 180° 反転させた角度をエッジに割り当てる。

ソースコード 5.5: objectNumbering_sub

```
1 function objectNumbering_sub {
2     ...
3     if (ntree == 1) {
4         text[Object.keys(text).length] = 0 + flag * Math.
              PI;
5     } else {
6         text[Object.keys(text).length] = Math.PI * ((ntree
               - fldArray.indexOf(property)) * 2 - 1) / (
              ntree * 2) - flag * Math.PI;
7     }
8     ...
9 }
```

ここまでが関数 objectNumbering の仕様であるが、objectNumbering は入力で与えられたオブジェクトと同じ型のオブジェクトしか辿っていかない。そこで、自身と異なる型も全て辿っていくのが関数 allObjectNumbering である。

allObjectNumbering では厳密には「プリミティブ型（number・string・boolean・symbol・undefined・null）と関数型（function）」以外の全てのオブジェクトを辿っていく。その際、事前に辿るフィールド名を全て関数 enum_DataStructureProperty で列挙しておく。

ソースコード 5.6: enum_DataStructureProperty

```
1 //自身の型とは異なるデータ構造のプロパティ名を全て列挙し、配
      列に保存しておく
2 function enum_DataStructureProperty(obj: Object, arrayProp
      : string[]) {
3     var cls: Function = obj.constructor;
4     var arrayObj: Object[] = new Array();
```

```
5
6     //補助関数
7     function enum_DataStructureProperty_sub(obj: Object,
          arrayProp: string[], arrayObj: Object[], cls:
          Function) {
8         arrayObj[Object.keys(arrayObj).length] = obj;
9
10        for (var prop in obj) {
11            if (!(obj[prop] instanceof cls) && typeof obj
                  [prop] != "number" && typeof obj[prop] !=
                   "string" && typeof obj[prop] != "boolean
                  "
12                && typeof obj[prop] != "symbol" && typeof
                       obj[prop] != "undefined" && typeof
                      obj[prop] != "function" && obj[prop]
                      != null) {
13                if (!sameName_inArray(prop, arrayProp)) {
14                    arrayProp[Object.keys(arrayProp).
                          length] = prop;
15                }
16            }
17        ...
18        }
19    }
20
21    enum_DataStructureProperty_sub(obj, arrayProp,
          arrayObj, cls);
22 }
```

ここで挙げたフィールド名を辿っていく。これらのフィールドの角度は 4.2.2 節で述べたように、n 分木のフィールドと同じように角度を決定する。また、各ノードの番号は objectNumbering に割り当てさせる。

ソースコード 5.7: allObjectNumbering

```
1 //全てのデータ構造をグラフに描画するために必要なエッジリスト
      トを配列text に書きこんでいく、返り値はノードの個数
2 function allObjectNumbering(obj: Object, num: number,
      drawcircle: boolean, text: number[], text2: string
      []): number {
3
4     function allObjectNumbering_sub(obj: Object, num:
          number, drawcircle: boolean, text: number[], text2
          : string[], clsArray: Function[], fldArray: string
          [][]): number {
```

```
5        if (obj instanceof Array) { //オブジェクトが配列の
             場合
6            ...
7        } else {
8            var cls: Function = obj.constructor;
9
10           if (!sameT_InArray<Function>(cls, clsArray)) {
11             ...
12           } else {
13               var arrayProp: string[] = new Array();
14               enum_DataStructureProperty(obj, arrayProp
                     );
15               var numberProp: number = Object.keys(
                     arrayProp).length;
16               var dotnum: number = objectNumbering(obj,
                     num, drawcircle, text, text2, fldArray
                     [clsArray.indexOf(cls)]);
17
18               //補助関数、各ノードの中に入っている別のデー
                     タ構造のエッジリストを配列
                     text に書きこんでいく
19               function edgeObjectNumbering(obj: Object,
                     num: number, arrayObj: Object[], text:
                      number[], text2: string[], clsArray,
                     fldArray): number {
20                   var cls: Function = obj.constructor;
21                   arrayObj[Object.keys(arrayObj).length]
                          = obj;
22                   var Objectnum: number = num;
23
24                   for (var prop in obj) {
25                       for (var i = 0; i < numberProp; i
                             ++) {
26                           if (prop == arrayProp[i] && !(
                                 obj[prop] instanceof cls))
                                 {
27                               text[Object.keys(text).
                                     length] = Objectnum;
28                               text[Object.keys(text).
                                     length] = dotnum + 1;
29                               text[Object.keys(text).
                                     length] = Math.PI / (
                                     numberProp * 2) * ((
                                     numberProp - i) * 2 -
                                     1);
```

```
30                              text2[Object.keys(text2).
                                    length] = prop;
31                              dotnum =
                                    allObjectNumbering_sub(
                                    obj[prop], dotnum + 1,
                                     drawcircle, text,
                                    text2, clsArray,
                                    fldArray);
32                          }
33                      }
34                  }
35
36                  for (var prop in obj) {
37                      if (obj[prop] instanceof cls) {
38                          if (!sameObject_inArray(obj[
                                prop], arrayObj)) {
39                              Objectnum =
                                    edgeObjectNumbering(obj
                                    [prop], Objectnum + 1,
                                     arrayObj, text, text2,
                                     clsArray, fldArray);
40                          }
41                      }
42                  }
43
44                  return Objectnum;
45              }
46
47              var arrayObj: Object[] = new Array();
48              edgeObjectNumbering(obj, num, arrayObj,
                    text, text2, clsArray, fldArray);
49
50              return dotnum;
51          }
52      }
53  }
54
55  var clsArray: Function[] = new Array();
56  var fldArray: string[][] = new Array();
57
58  return allObjectNumbering_sub(obj, num, drawcircle,
        text, text2, clsArray, fldArray);
59 }
```

5.1.1.2　allNumberInNode

関数 allNumberInNode は、関数 objectDraw の引数として与えられた文字列と一致するフィールドの値を収集する関数である。

関数 allNumberInNode では内部で numberInNode という別の関数を使用する。

関数 numberInNode では objectNumbering と同じオブジェクトの辿り方をしながら与えられた文字列 pname と一致するフィールドの値を配列に書きこんでいく。複数のフィールドの値を表示できるように、可変長引数を使用している。また、値の存在しないノードに対しては null を配列に書きこむ。

ソースコード 5.8: numberInNode

```
1  //データ構造の
       pname という名前のプロパティ内部の数字を配列 text に書きこんでいく

2  function numberInNode(obj: Object, text: number[], text2:
        string[], ...pname: string[]) {
3      var arrayObj: Object[] = new Array();
4
5      //補助関数
6      function numberInNode_sub(obj: Object, arrayObj:
           Object[], text: number[], text2: string[], ...
           pname: string[]) {
7          var cls: Function = obj.constructor;
8          var clsname: string = getName(obj);
9
10         arrayObj[Object.keys(arrayObj).length] = obj;
11
12         var flag: number = 0;
13         for (var prop in obj) {
14             if (sameName_inArray(prop, pname)) { //
                   pname と一致したフィールドの値を配列に書きこむ
15                 text[Object.keys(text).length] = obj[prop
                       ];
16                 text2[Object.keys(text2).length] = clsname
                       ;
17                 flag = 1;
18             }
19         }
20         if (flag == 0) { //値の存在しないノードには
               null を書きこむ
21             text[Object.keys(text).length] = null;
```

```
22            text2[Object.keys(text2).length] = clsname;
23        }
24        ...
25    }
26
27    numberInNode_sub(obj, arrayObj, text, text2, ...pname
        );
28 }
```

numberInNode も objectNumbring と同様、同じ型のオブジェクトのみを辿っていくので、全てのオブジェクトを辿っていく allNumberInNode を用意する。辿っていくフィールド名は、関数 enum_DataStructureProperty で列挙したフィールド名である。

ソースコード 5.9: allNumberInNode

```
1 //全てのデータ構造の
    pname という名前のプロパティ内部の数字を配列 text に書きこんでいく

2 function allNumberInNode(obj: Object, text: number[],
    text2: string[], ...pname: string[]) {
3     if (obj instanceof Array) { //オブジェクトが配列の場合
4         ...
5     } else {
6         numberInNode(obj, text, text2, ...pname);
7
8         var arrayProp: string[] = new Array();
9         enum_DataStructureProperty(obj, arrayProp);
10
11        var numberProp: number = Object.keys(arrayProp).
            length;
12
13        //補助関数
14        function allNumberInNode_sub(obj: Object, arrayObj
            : Object[], text: number[], text2: string[],
            ...pname: string[]) {
15            var cls: Function = obj.constructor;
16            arrayObj[Object.keys(arrayObj).length] = obj;
17
18            for (var prop in obj) {
19                for (var i = 0; i < numberProp; i++) {
20                    if (prop == arrayProp[i] && !(obj[
                        prop] instanceof cls)) {
21                        allNumberInNode(obj[prop], text,
                            text2, ...pname);
```

```
22                      }
23                  }
24              }
25
26              for (var prop in obj) {
27                  if (obj[prop] instanceof cls) {
28                      if (!sameObject_inArray(obj[prop],
                            arrayObj)) {
29                          allNumberInNode_sub(obj[prop],
                                arrayObj, text, text2, ...
                                pname);
30                      }
31                  }
32              }
33          }
34
35          var arrayObj: Object[] = new Array();
36          allNumberInNode_sub(obj, arrayObj, text, text2,
                ...pname);
37      }
38  }
```

5.1.2 力学的手法（Force-directed Method）によるノードの座標決定

関数 allObjectNumbering・allNumberInNode により得られたグラフ情報を元に、Fruchterman-Reingold アルゴリズム [8] を改良した力学的手法によってノードの座標を計算していく。

5.1.2.1 ノード・エッジ・グラフの用意

グラフのノードとエッジを表すクラスを定義する。ノードを表すクラスには、座標を表すフィールド以外に速度を表すフィールドを用意しておく。また、ノードの各座標の初期値はランダムに設定しておく。

ソースコード 5.10: Dot・Edge・Graph の用意

```
1      //点のクラス
2      class Dot_G {
3          x: number;
4          y: number;
5          dx: number; //速度のx 成分
```

```
6           dy: number; //速度のy 成分
7           fax: number; //引力のx 成分
8           fay: number; //引力のy 成分
9           frx: number; //斥力のx 成分
10          fry: number; //斥力のy 成分
11          fmx: number; //モーメントのx 成分
12          fmy: number; //モーメントのy 成分
13          nodenum: number | string; //点をノードと見なした時
                の中身の変数
14          nodecls: string; //点をノードと見なした時のクラス
                名
15      }
16
17      //辺のクラス
18      class Edge_G {
19          dot1: Dot_G;
20          dot2: Dot_G;
21          angle: number; //辺の角度
22          fieldname: string; //エッジのフィールド名を表す
23      }
24
25      //グラフのクラス
26      class Graph_G {
27          dot_number: number; //ノードの数
28          edge_number: number; //エッジの数
29          edges: Edge_G[];
30          dots: Dot_G[];
31      }
32
33      for (var i = 0; i < DOTNUMBER; i++) { //各ノードの座
            標をランダムに設定する
34              dots[i].init(Math.floor(Math.random() * canvas
                    .width), Math.floor(Math.random() * canvas
                    .height), ARRAYTEXT[i], ARRAYCLS[i]);
35      }
```

5.1.2.2　引力・斥力・角度力の計算

角度付きエッジリストを元にそれぞれのノードに働く引力・斥力・角度力を計算していく。

引力は隣接ノードからのみ力を受ける。引力 f_a の大きさは $f_a = \frac{d^2}{k}$（d はノード間のユークリッド距離）で定義される。また、$k = c\frac{|V|}{d_{max}}$ と定義する。c は定数、$|V|$ はノードの全個数、d_{max} はグラフのノード同士の最

短経路長の最大値である。

これに対して斥力 f_r は他の全てのノードから力を受け、その大きさは $f_r = \frac{k^8}{d^3}$ で定義される。

元の Fruchterman-Reingold アルゴリズムでは斥力は $\frac{k^2}{d}$ であり、$k = c\sqrt{\frac{area}{|V|}}$（area は描画領域の面積）と定義されているが、角度力を導入するにあたって一部係数や式を変更している。これらの係数や式は経験的に得られたものである。

また角度力については、与えられたエッジの端点 v_1 と v_2 に対してベクトル $\overrightarrow{v_2} - \overrightarrow{v_1}$ と x 軸のなす角 ϕ を計算し、エッジリストに決められた角度 θ に対して $\theta - \phi$ の 2 乗に比例する力を v_1 と v_2 に働かせる。このとき、角度力のベクトルは $\overrightarrow{v_2} - \overrightarrow{v_1}$ と垂直の方向に働くようにする。

計算した各力は Dot_G の速度フィールドに書きこまれる。

ソースコード 5.11: focus_calculate

```
1   //2 点間の引力を計算
2   function f_a(r: number, K: number): number {
3       return r * r / K;
4   }
5
6   //2点間の斥力を計算
7   function f_r(r: number, K: number): number {
8       return Math.pow(K, Knum) / Math.pow(r, rnum);
9   }
10
11  //各点の引力･斥力を計算し､Dot[]に代入していく
12  function focus_calculate(dots: Dot_G[]) {
13
14      //各点の斥力を計算
15      for (var i = 0; i < DOTNUMBER; i++) {
16          for (var j = 0; j < DOTNUMBER; j++) {
17              if (j != i) {
18                  var dx: number = dots[i].x - dots[j].
                        x;
19                  var dy: number = dots[i].y - dots[j].
                        y;
20                  var delta = Math.sqrt(dx * dx + dy *
                        dy);
21                  if (delta != 0) {
22                      var d: number = f_r(delta, K) /
                            delta;
23                      dots[i].frx += dx * d;
24                      dots[i].fry += dy * d;
```

```
25                      }
26                  }
27              }
28          }
29
30          //各点の引力を計算
31          for (var i = 0; i < EDGENUMBER; i++) {
32              var dx: number = edges[i].dot1.x - edges[i].
                    dot2.x;
33              var dy: number = edges[i].dot1.y - edges[i].
                    dot2.y;
34              var delta: number = Math.sqrt(dx * dx + dy *
                    dy);
35              if (delta != 0) {
36                  var d: number = f_a(delta, K) / delta;
37                  var ddx: number = dx * d;
38                  var ddy: number = dy * d;
39                  edges[i].dot1.fax += -ddx;
40                  edges[i].dot2.fax += +ddx;
41                  edges[i].dot1.fay += -ddy;
42                  edges[i].dot2.fay += +ddy;
43              }
44          }
45
46          //各点の角度に基づいて働く力を計算
47          for (var i = 0; i < EDGENUMBER; i++) {
48              if (GRAPHTEXT[3 * i + 2] != 9973) {
49                  var dx: number = edges[i].dot2.x - edges[i
                        ].dot1.x;
50                  var dy: number = edges[i].dot2.y - edges[i
                        ].dot1.y;
51                  var delta: number = Math.sqrt(dx * dx + dy
                         * dy);
52                  var rad: number = Math.atan2(dy, dx);
53                  if (delta != 0) {
54                      var d: number = rad - GRAPHTEXT[3 * i
                             + 2];
55                      var ddx: number;
56                      var ddy: number;
57                      var ex: number = KRAD * dy / delta;
                            //角度に関する力の基本ベクトル(元の
                            ベクトルを負の方向に 90度回転)
58                      var ey: number = - KRAD * dx / delta;
                             //角度に関する力の基本ベクトル(元
                            のベクトルを負の方向に 90度回転)
```

```
59                  if (Math.abs(d) <= Math.PI) {
60                      ddx = d * Math.abs(d) * ex;
61                      ddy = d * Math.abs(d) * ey;
62                  } else {
63                      var dd: number = d + 2 * Math.PI;
64                      if (d < 0) {
65                          ddx = dd * Math.abs(dd) * ex;
66                          ddy = dd * Math.abs(dd) * ey;
67                      } else {
68                          ddx = -dd * Math.abs(dd) * ex;
69                          ddy = -dd * Math.abs(dd) * ey;
70                      }
71                  }
72                  edges[i].dot1.fmx += -ddx;
73                  edges[i].dot1.fmy += -ddy;
74                  edges[i].dot2.fmx += ddx;
75                  edges[i].dot2.fmy += ddy;
76              }
77          }
78      }
79  }
```

5.1.2.3　各点の移動

計算した力を元に、各点の座標を移動させていく。このとき、動かす変位の大きさが温度パラメータ t をうわまらないようにする。

全てのノードの座標を動かしたら温度パラメータを少し下げ、再び各点に働く力を計算する。これを t が 0 を下回るまで繰り返していく。t が 0 を下回った時点で計算を終了し、計算結果を画面に表示する。

ソースコード 5.12: draw

```
1    //温度パラメータが0以下になるまで安定状態を探索する
2    while (true) {
3        draw();
4        if (t <= 0) break;
5    }
6
7   //fruchterman-
        Reingold法でエネルギーを最小化し、グラフを描画する
8   function draw() {
9
10      //各点に働く力を計算
```

```
11        focus_calculate(dots);
12
13        //各点の速度から､次の座標を計算する
14        for (var i = 0; i < DOTNUMBER; i++) {
15            var dx: number = dots[i].dx;
16            var dy: number = dots[i].dy;
17            var disp: number = Math.sqrt(dx * dx + dy *
                  dy);
18
19            if (disp != 0) {
20                var d: number = Math.min(disp, t) / disp;
21                dots[i].x += dx * d;
22                dots[i].y += dy * d;
23            }
24        }
25
26        //温度パラメータを下げていく、0を下回ったら終了
27        t -= dt;
28        if (t <= 0) stopCalculate();
29    }
30
31    //計算終了
32    function stopCalculate() {
33        ...
34    }
```

5.2　例外的な処理

5.2.1　循環のあるグラフ構造について

第 4 章でも述べた通り、循環のあるグラフ構造については例外的な処理を行う。具体的には、循環部分を発見したら循環部分を含むクラスのオブジェクト群の角度力を全て 0 にする。これにより、例えば循環リストなどは元の Fruchteman-Reingold アルゴリズムの通りにレイアウトされる。

循環部分の発見は、あるオブジェクトから特定のフィールドを辿っていったときに元のオブジェクトにたどり着くかどうかで判定する。この特定のフィールドとは、関数 enum_sameProperty で列挙したフィールドから関数 identify_unnecessaryProperty で列挙したフィールドを除いたフィールドを指す。

ソースコード 5.13: isCirculationOfOneProperty

```
1 //オブジェクトの特定のフィールドについて、そのフィールドを辿
    っていったときに循環があるかどうかを判定する
2 //循環があった場合には、循環部分のエッジの両端点のオブジェク
    トを配列arrayCircule に格納する
3 function isCirculationOfOneProperty(obj: Object,
    arrayCircle: Object[], fieldCircle: string[], ...
    property: string[]): boolean {
4     var cls: Function = obj.constructor;
5     var arrayObj: Object[] = new Array();
6
7     //補助関数
8     function isCirculationOfOneProperty_sub(obj: Object,
        cls: Function, arrayObj: Object[], arrayCircle:
        Object[], fieldCircle: string[], ...property:
        string[]): boolean {
9         arrayObj[Object.keys(arrayObj).length] = obj;
10        var bool: boolean = false;
11
12        for (var prop in obj) {
13            if (obj[prop] instanceof cls &&
                sameName_inArray(prop, property)) {
14                if (sameObject_inArray(obj[prop], arrayObj
                    )) {
15                    arrayCircle[Object.keys(arrayCircle).
                        length] = obj;
16                    arrayCircle[Object.keys(arrayCircle).
                        length] = obj[prop];
17                    fieldCircle[Object.keys(fieldCircle).
                        length] = prop;
18                    bool = true;
19                } else {
20                    var bool2: boolean =
                        isCirculationOfOneProperty_sub(obj[
                        prop], cls, arrayObj, arrayCircle,
                         fieldCircle, ...property);
21                    bool = bool || bool2;
22                }
23            }
24        }
25
26        return bool;
27    }
28
29    return isCirculationOfOneProperty_sub(obj, cls,
        arrayObj, arrayCircle, fieldCircle, ...property);
```

```
30 }
```

グラフ内に循環が発見された場合は、元のオブジェクトに戻ってきたエッジの端点を記憶しておき、関数 objectNumbering 内でこれらの端点の番号をエッジリストに追加する。またこのとき、エッジリストの角度部分には $[-\pi, \pi]$ の範囲外の適当な数字を割り当てておく。（実装内では 9973 を割り振っている。この数字に特に意味はない。）

ソースコード 5.14: objectNumbering での循環の扱い

```
1  function objectNumbering(...): number {
2      //オブジェクトが木構造かグラフ構造化を判断する
3      var arrayCircle: Object[] = new Array();
4      var fieldCircle: string[] = new Array();
5      var isgraph: boolean = isAnyCirculationOfOneProperty(
           obj, arrayCircle, fieldCircle, ...newArray);
6
7      function objectNumbering_sub(...) {
8
9          if (isgraph && drawcircle == true) { //グラフ構造
               の場合
10
11             var cls: Function = obj.constructor;
12             var n: number = 0;
13             arrayObj[Object.keys(arrayObj).length] = obj;
14             var dotnum: number = num;
15
16             //循環部分のノードの番号を配列
                   arrayCircle に記憶しておく
17             for (var i = 0; i < Object.keys(arrayCircle).
                   length; i++) {
18                 if (obj == arrayCircle[i]) {
19                     addedge_ofgraph[i] = num;
20                 }
21             }
22
23             for (var prop in obj) {
24                 if (obj[prop] instanceof cls) {
25                     if (!sameObject_inArray(obj[prop],
                           arrayObj)) {
26                         n += 1;
27                         text[Object.keys(text).length] =
                               num;
28                         text[Object.keys(text).length] =
                               dotnum + 1;
```

```
29                        text[Object.keys(text).length] =
                              9973;
30                        text2[Object.keys(text2).length] =
                               prop;
31
32                        dotnum = objectNumbering_sub(...);
33                    }
34                }
35            }
36
37            return dotnum;
38
39        } else { //木構造の場合
40            ...
41        }
42 }
```

5.2.2　配列について

objectDraw に与えられたオブジェクトが配列だった場合、allObjectNumbering と allNumberInNode の関数内で配列を表すようなクラスを新たに定義し、そのクラスから生成されたオブジェクトを元の関数に渡すことで処理している。

ソースコード 5.15: 配列 Array 型の対応

```
1  function allObjectNumbering(...): number {
2      function allObjectNumbering_sub(...): number {
3          if (obj instanceof Array) { //オブジェクトが配列
               Array 型であった場合
4              class objectArray {
5                  obj: Object;
6                  ARRAY: objectArray;
7
8                  constructor(obj: Object) {
9                      this.obj = obj;
10                     this.ARRAY = null;
11                 }
12                 add(obj: Object) {
13                     if (this.ARRAY == null) {
14                         var node: objectArray = new
                               objectArray(obj);
15                         this.ARRAY = node;
```

```
16                     } else {
17                         this.ARRAY.add(obj);
18                     }
19                 }
20             }
21             var oA: objectArray = new objectArray(obj[0]);
22             for (var i = 1; i < Object.keys(obj).length;
                   i++) {
23                 oA.add(obj[i]);
24             }
25             return allObjectNumbering_sub(...);
26         } else {
27             ...
28         }
29     }
30 }
```

5.2.3　オブジェクトの色分けについて

本アルゴリズムの特徴の一つとして、プログラマーの視認性を向上させるためにレイアウトされるグラフのノードに彩色を行っていることが挙げられる。基本的には、同じクラスから生成されたオブジェクトは同じ色に、異なるクラスから生成されたオブジェクトは異なる色になる。

厳密に述べると、発見されたオブジェクトのクラスに順番にピンク、黄緑、水色、黄色、オレンジ、グレーの色を割り当てていき、見つかったクラス数が 6 種類を超えた場合は再びピンクから割り当てていく。

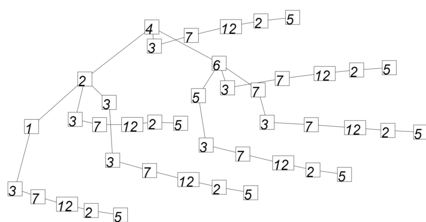

図 5.1: 色無しのレイアウト

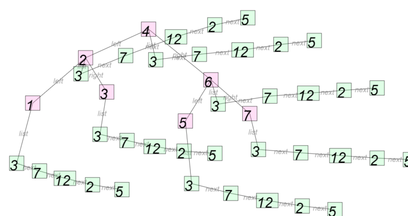

図 5.2: 色有りのレイアウト。視認性が向上する

色分けについては、クラス配列からそれぞれの色を表す string の配列

を作り、draw 関数に渡すことで実現した。

ソースコード 5.16: coloring

```
1 function objectDraw(obj: Object, ...pname: string[]) {
2
3     //色分け配列の用意
4     var arrayColor: string[] = new Array();
5     var dotcls: string[] = new Array();
6     coloring(ARRAYCLS, arrayColor, dotcls);
7
8    //クラスの種類の数からノードの色分けをする
9     function coloring(ARRAYCLS: string[], arrayColor:
         string[], dotcls: string[]) {
10
11         for (var i = 0; i < DOTNUMBER; i++) {
12             if (!sameT_InArray<string>(ARRAYCLS[i], dotcls
                 )) {
13                 dotcls[Object.keys(dotcls).length] =
                     ARRAYCLS[i];
14             }
15         }
16
17         var leg: number = Object.keys(dotcls).length;
18
19         for (var i = 0; i < leg; i++) {
20             switch (i % 8) {
21                 case 0: //ピンク
22                     arrayColor[i] = "rgba
                         (255,96,208,0.2)";
23                     break;
24                 case 1: //黄緑
25                     arrayColor[i] = "rgba
                         (96,255,128,0.2)";
26                     break;
27                 case 2: //水色
28                     arrayColor[i] = "rgba
                         (80,208,255,0.2)";
29                     break;
30                 case 3: //黄色
31                     arrayColor[i] = "rgba
                         (255,224,32,0.2)";
32                     break;
33                 case 4: //オレンジ
34                     arrayColor[i] = "rgba
```

```
                    (255,160,16,0.2)";
35                  break;
36              case 5: //グレー
37                  arrayColor[i] = "rgba
                    (224,224,224,0.2)";
38                  break;
39              default:
40                  arrayColor[i] = "rgba
                    (255,255,255,0.2)";
41          }
42      }
43  }
44
45  }
```

第6章　実験・提案手法の評価

本章では、提案したアルゴリズムが既存のライブプログラミングでのデータ構造可視化の課題を解決できたかどうかの評価を行う。またライブプログラミングで重要となる計算速度についての評価を行う。

6.1　グラフレイアウトについて

データ構造を構成するプログラムを16種類作成し、Kanonで同じように作成したプログラムのレイアウト結果と見比べて比較した。16種類のデータ構造は以下の通りである。

- 単方向リスト
- 双方向リスト
- 単方向循環リスト
- 双方向循環リスト
- 単方向二分木
- 二分探索木
- 赤黒木（バランス木）
- 三分木（トライ木）
- 二分探索木を要素として持つリスト
- 二分探索木を要素として2つ持つリスト
- 二分探索木の配列を要素として持つリスト
- バランス木を要素として持つリスト
- リストを要素として持つ二分木
- 二分探索木の配列を要素として持つ二分探索木

- 二分探索木の配列を持つオブジェクト
- 環境

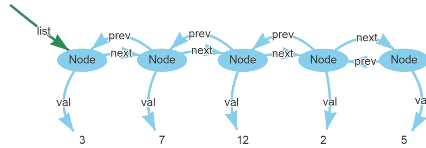

図 6.1: Kanon でのリストレイアウト

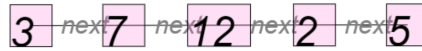

図 6.2: 提案アルゴリズムでのリストレイアウト

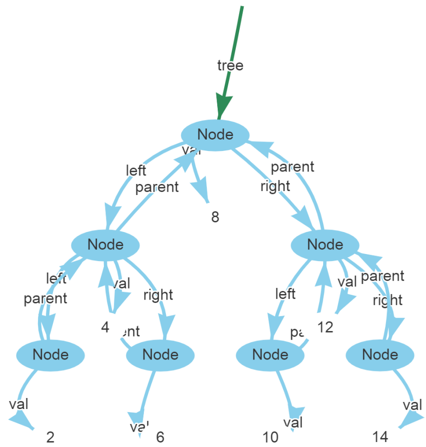

図 6.3: Kanon での二分木レイアウト

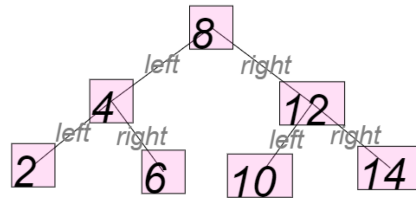

図 6.4: 提案アルゴリズムでの二分木レイアウト

本アルゴリズムでは与えられたデータ構造を

(1) 一本木構造

(2) n 分木構造（$n \geq 2$）

(3) 循環のあるグラフ構造

のいずれかに分類してレイアウトするので、「データ構造の種類に関する汎用性の欠如」の問題点は解決している。そのため、Kanon では視認しやすい形にレイアウトされなかった三本木（Trie）なども視認しやすい形でレイアウトされる。

また、レイアウトはクラス名やフィールド名に依存しないのでこれらの名前を変えてもレイアウトは元の性質を失わない。よって、「識別子に依

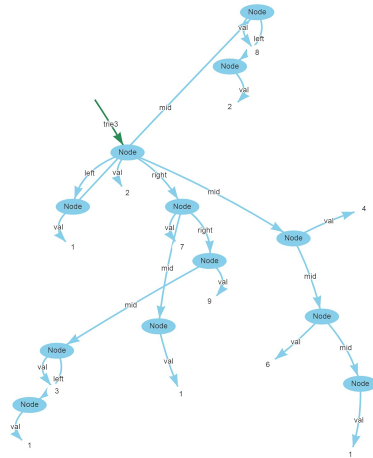

図 6.5: Kanon での三分木レイアウト

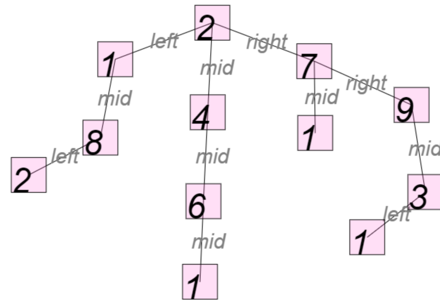

図 6.6: 提案アルゴリズムでの三分木レイアウト

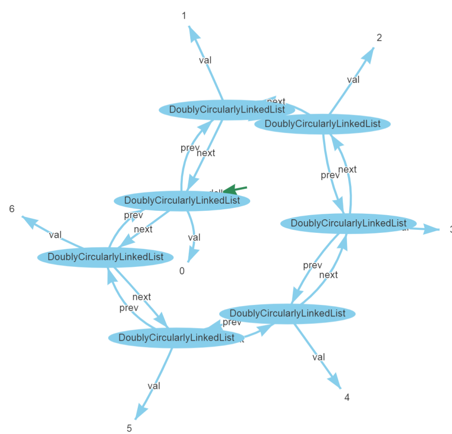

図 6.7: Kanon での循環リストレイアウト

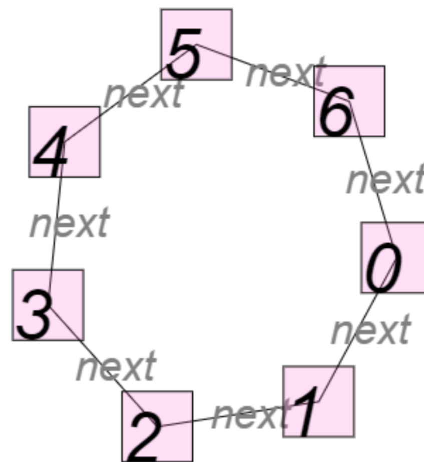

図 6.8: 提案アルゴリズムでの循環リストレイアウト

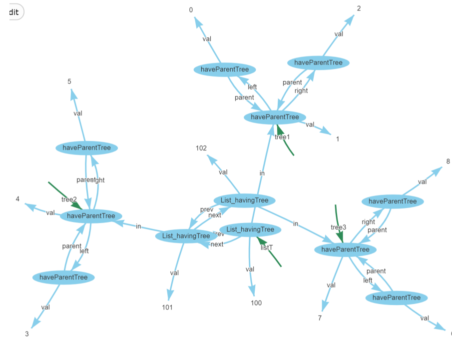

図 6.9: Kanon での二分木を要素として持つリストのレイアウト

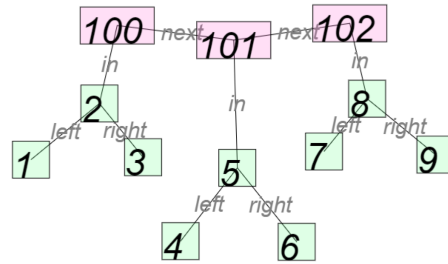

図 6.10: 提案アルゴリズムでの二分木を要素として持つリストのレイアウト

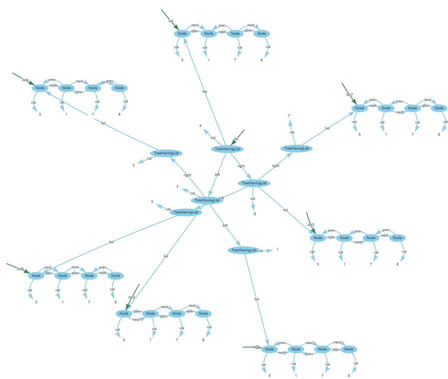

図 6.11: Kanon でのリストを要素として持つ二分木のレイアウト

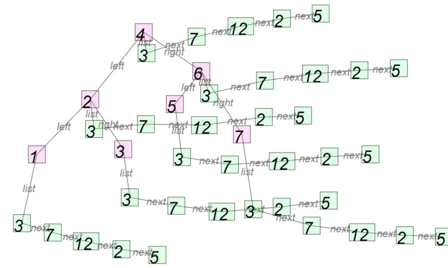

図 6.12: 提案アルゴリズムでのリストを要素として持つ二分木のレイアウト

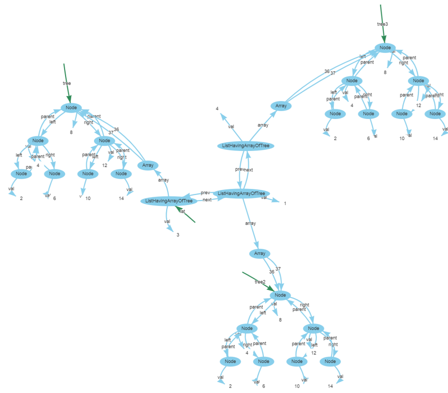

図 6.13: Kanon での二分木の配列を要素として持つリストのレイアウト

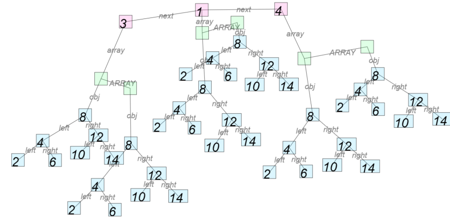

図 6.14: 提案アルゴリズムでの二分木の配列を要素として持つリストのレイアウト

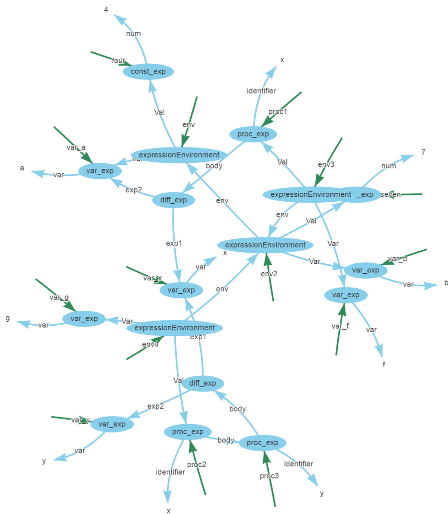

図 6.15: Kanon での環境図のレイアウト

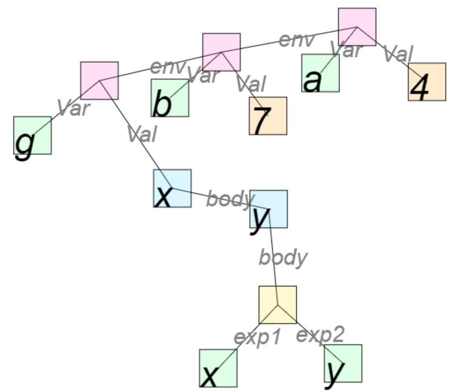

図 6.16: 提案アルゴリズムでの環境図のレイアウト

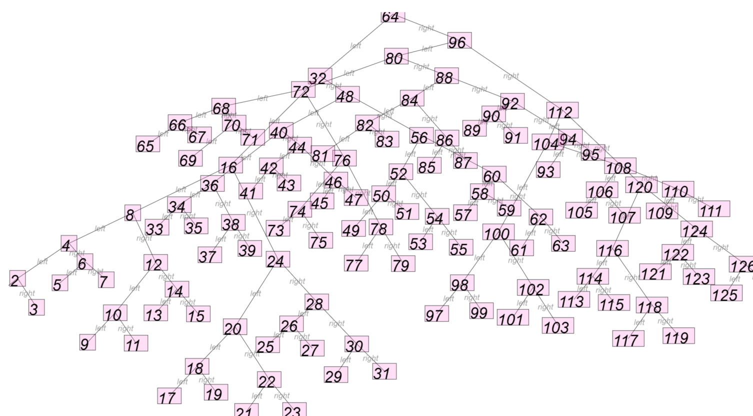

図 6.17: ノード数が多くなった二分木のレイアウト結果

存したレイアウトの決定」の問題点を解決している。さらに、allObjectNumbering や allNumberInNode でプリミティブ型と関数型以外の全てのオブジェクトを辿るようにしているので、「入れ子構造の非対応」も解決している。

しかし、提案アルゴリズムはノード数が多くなるとエッジの交差が増えることが確認された。元々、力学的手法によるグラフレイアウトアルゴリズムはノードの座標が局所解に陥ってしまうケースがあるという問題点を抱えている。提案アルゴリズムにおいてもノード数が増えると局所解に陥りやすくなってしまい、結果としてエッジの交差が多く発生してしまう（図 6.5）。

6.2　計算時間について

計算速度の測定には TypeScript の performance 関数を用いた。objectDraw 関数が実行される前から測定を始め、完全に実行が終わった段階で測定を終了する。この測定を 10 回繰り返し、初めの 1 回を除いた 9 回の平均値を測定値とする。実装ではノードにかかる力の計算の反復回数を 8000 回と設定した。計算速度の測定には Intel Core i7(2.5 GHz), メモリ 8GB の計算機を使用した。

objectDraw には平衡二分木 rbt(Red-Black-Tree) を渡して測定する。ノード数は $2^i - 1 (1 \leq i \leq 7)$ とする。

結果は表 6.1 のようになった。

ライブプログラミングでは即座に実行結果をフィードバックしたいので 1 秒という時間が主な目安となってくるが、ノード数が 30 を超えたあたりから計算に 1 秒以上かかるようになった。

表 6.1: ノード数と objectDraw の実行にかかった時間

ノード数	タイム [ms]
$2^1 - 1$	8.244
$2^2 - 1$	37.099
$2^3 - 1$	93.088
$2^4 - 1$	382.988
$2^5 - 1$	1109.533
$2^6 - 1$	5864.555
$2^7 - 1$	23933.822

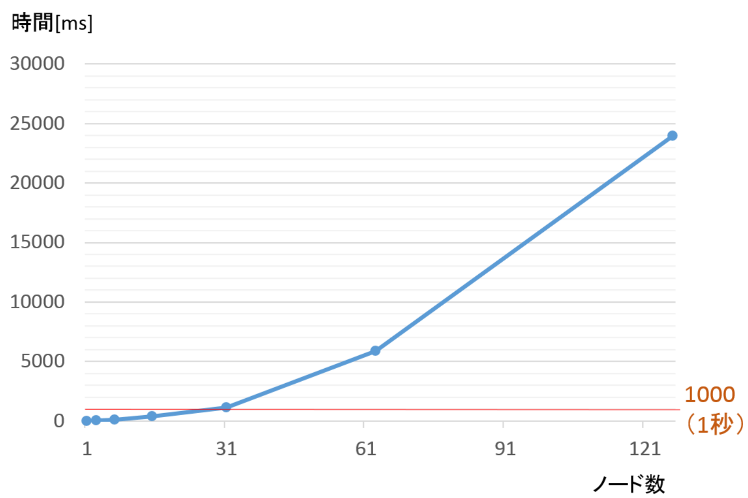

図 6.18: ノード数と objectDraw の実行にかかった時間

第7章　関連研究

7.1　ライブプログラミング環境

本研究で提案したオブジェクトグラフ自動レイアウトアルゴリズムはデータ構造のためのライブプログラミング環境でのレイアウトアルゴリズムである。第2章で述べたKanon以外にも、プログラミングの効率の向上を目指して様々なライブプログラミング環境が開発されている。

Kanon [1] は、データ構造を図表現として可視化するライブプログラミング環境である。JavaScript言語を対象に、Khan AcademyのLive Editorを拡張して設計されている。Kanonは、プログラムが編集される度にプログラムを実行し、その途中で生成されたオブジェクト及び参照関係を図表現として可視化する。本研究でのデータ構造のグラフ的表現にはKanonの影響を強く受けている。Kanonはデータ構造の可視化に特化したライブプログラミング環境であるが、第3章で述べたように一部のデータ構造が上手くレイアウトされない。

YinYang [2] はライブプログラミング環境で動作するプログラミング言語、またはその環境のことである。ProbingとTracingという2つの機能を用いてプログラマをサポートする。Probingはプログラム中の式の値を表示する機能であり、表示したい式の直前に@を挿入することでその式の値を表示する。また、Tracingは通常のプログラミングのprint文のような機能であり、prn関数の引数として渡された式を評価して値を画面に表示する。YinYangにはデータ構造を図表現として表示する機能はない。

Live Editor [3] はJavaScriptおよびそのライブラリであるProcessing.jsのライブプログラミング環境である。Live Editorでプログラムを編集すると、描画に関する関数によって描かれる図が表示画面に描画される。また、数値をドラッグアンドドロップで変更する機能などによってプログラマを支援している。しかし、Live Editorも他のライブプログラミング環境と同様に、データ構造を図表現として表示する機能は搭載していない。

7.2　データ構造の可視化

複雑な構造を持ったデータをソースコードから読み取って、頭の中で理解するのはプログラマにとって負担のかかる作業である。そのため、データ構造を可視化する研究が多くされている。

ghc-vis [9] は Haskell の対話環境で用いられるデータ構造可視化ツールである。Haskell のプログラムのデータ構造を図表現として表示する。また、Haskell には lazy evaluation や sharing などの特徴があり、ghc-vis を使うことでこれらの挙動を図表現として見ることができる。しかし、ghc-vis のレイアウトアルゴリズムでは「同じフィールドを同じ方向に揃える」という処理は行われない。

Python Tutor [10] は Web ブラウザ上で動作する Python のためのプログラム可視化ツールである。記述したコードを 1 ステップごとにどのような内部状態になっているかを図表現で可視化する。また、Java、JavaScript、TypeScript、Ruby、C、C++など幅広い言語に対応している。しかし、Python Tutor のレイアウトアルゴリズムでも「同じフィールドを同じ方向に揃える」という処理は行われない。

7.3　グラフレイアウトアルゴリズム

グラフを綺麗にレイアウトするためのアルゴリズムは多く研究・開発されており、様々なアプリケーションのニーズに対応している。

Tilford-Reingold アルゴリズム [4] は木構造のレイアウトアルゴリズムである。木のエッジがその他のエッジと重ならない、同じ深さにあるノードが同じ水平のライン上にある、木が出来る限り狭く描かれる、親のノードがその子のノードの中央に描かれる、サブツリーがどこに存在していても同じ構造のサブツリーが描かれる、といった 5 つの条件が守られるようにレイアウトされる。非常にバランスよく木をレイアウトすることができるが、データ構造をレイアウトしようとしたときに木構造以外の構造に応用しにくい、また入れ子構造のデータに対して入れ子の形を無視してレイアウトしてしまうという欠点がある。

Sugiyama フレームワーク [5] は階層型レイアウトアルゴリズムのひとつであり、GraphViz というグラフレイアウトソフトウェアにも搭載されているアルゴリズムである。（1）階層割当、（2）交差削減、（3）座標割当、の 3 つの手順を踏むことで階層グラフをレイアウトする。しかし、データ構造のレイアウト時には「フィールドの情報が無視されてしまう」という問題点がある。

Fruchterman-Reingold アルゴリズム [8] は力学モデルアルゴリズムのひとつであり、各ノードにかかる引力と斥力を計算しながらノードを動か

していき座標を求める。また、温度パラメータと呼ばれるパラメータによって各ノードの変位の最大値が制限されているのも特徴のひとつである。本研究で用いた力学的手法は Fruchterman-Reingold アルゴリズムの影響を非常に強く受けている。しかし、データ構造のレイアウトに使用すると Sugiyama フレームワークと同じように「フィールドの情報が無視されてしまう」ことが問題点となる。

第8章　まとめ・課題

8.1　まとめ

本研究では、データ構造の関係性を視認しやすくする自動グラフレイアウト手法の提案を行った。またそのために、新たに角度力を追加した力学的手法の提案を行った。そして、それらのアルゴリズムの設計・実装を行った。

結果、「データ構造の種類に関する汎用性の欠如」「識別子に依存したレイアウトの決定」「入れ子構造の非対応」の 3 つの問題点が解決された。またどのような場合に上手くレイアウトされないかを考察した。

8.2　課題

本節では、本研究の今後の課題を述べる。

8.2.1　計算時間の向上

第 6 章で述べたとおり、本研究で提案したアルゴリズムは計算時間が $\mathcal{O}(|V|^2)$ となる。ノード数が 60 を超えたあたりから計算時間が 1 秒以上かかるようになってしまう。これは「編集すると即座に結果を表示する」というライブプログラミングの目標と反しており、計算時間の向上は必須の課題であると言える。

8.2.2　ユーザー実験

本研究の目標は「ライブプログラミング環境でデータ構造をプログラマに理解しやすい形でレイアウトさせる」というものであるため、本研究の目的が達成されたかの可否は個々のプログラマの主観に強く依存する。そのため、どれだけ研究目的が達成されたかの指標とするために今後ユーザー実験を行う予定である。具体的には、提案したアルゴリズムをユーザーに使用してもらい、レイアウトされた結果を見てデータ構造が理解しやすいかどうかのアンケートをとる、という形でユーザー実験を行う。

8.2.3　相対角度の導入

　現在の実装では、全てのエッジの角度を画面内の絶対角度として自動決定している。これを、画面内の絶対角度ではなく他のエッジとどれだけ角度が開いているかによって端点に働く角度力を変えるアルゴリズムを考えている。このアルゴリズムを相対角度によるアルゴリズムと便宜上呼ぶことにする。相対角度によるアルゴリズムを導入することで、「データ構造の種類に関する汎用性の欠如」を完全に克服できると考えている。

参考文献

[1] Oka, Akio, Hidehiko Masuhara, and Tomoyuki Aotani. ”Live, synchronized, and mental map preserving visualization for data structure programming.” Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software(Onward! 2018). ACM, 2018.

[2] Sean McDirmid. Usable Live Programming. In Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, Onward! 2013, pages 53–62, New York, NY, USA, 2013. ACM.

[3] John Resig. Redefining the Introduction to Computer Science. http://ejohn.org/blog/introducing-khan-cs/. Accessed 2016-12-23.

[4] Reingold, Edward M., and John S. Tilford. ”Tidier drawings of trees.” IEEE Transactions on software Engineering 2 (1981): 223-228.

[5] K. Sugiyama, S. Tagawa and M. Toda, “Methods for visual understanding of hierarchical system structures,” IEEE Transactions on Systems, Man, and Cybernetics, 11, pp. 109–125, 1981.

[6] Ellson, John, et al. ”Graphviz—open source graph drawing tools.” International Symposium on Graph Drawing. Springer, Berlin, Heidelberg, 2001.

[7] Kamada, Tomihisa, and Satoru Kawai. ”An algorithm for drawing general undirected graphs.” Information processing letters 31.1 (1989): 7-15.

[8] Fruchterman, Thomas MJ, and Edward M. Reingold. ”Graph drawing by force directed placement.” Software: Practice and experience 21.11 (1991): 1129-1164.

[9] Felsing, Dennis. Visualization of lazy evaluation and sharing. Diss. Bachelor ' s thesis, Karlsruhe Institute of Technology, Germany, 2012.

[10] Philip J. Guo, "Online python tutor: embeddable web-based program visualization for cs education." Proceeding of the 44th ACM technical symposium on Computer science education. ACM, 2013.