Vol.2019-5 No.4-3
2020/02/09

マトロイドに基づく QuickCheck テスティングフレームワークの開発

荒 井 大 和[†1]　鈴 木 大 郎[†1]

最適化問題は多くの場面で解くことを要求される重要な問題の 1 つで、この問題を解くプログラムを実装することは多い。そのようなプログラムの正当性の検査において、最適化問題の最適性を形式的に示す手法は有用である。また最適化問題はしばしばグラフを用いて解かれるので、そのような解法を実装したプログラムのテストにはランダムなグラフデータの生成器が必要になる。本研究では重み付きマトロイド上の最適化問題に適用可能な解の最適性検査プログラムと様々なデータ構造で表されたグラフが生成できるグラフ生成器を備えた QuickCheck のテスティングフレームワークを開発する。本稿は次のように構成される。まず本稿を読むための事前知識を述べる。次に本研究で開発したフレームワークの概要を述べ、以降の節で本フレームワークで提供する関数の設計と実装について述べる。最後に、本フレームワークで提供する関数の性能評価と、今後の課題について述べる。

1. はじめに

最適化問題は多くの場面で解くことを要求される重要な問題の一つであるため、最適化問題を解く多くのプログラムが実装されている。実装したプログラムの正当性を検査する方法として最も多く利用されているのは、プログラムのテストである。テストの手法として様々な方法が提案され、利用されているが、その中に性質によるテストと呼ばれる手法がある。QuickCheck [1] はこの手法に基づくテストツールである。

QuickCheck は Haskell で実装されたテストツールであり、現在は C++や Java などにも移植されている。Haskell の QuickCheck ではプログラムのソースコード内にプログラムが満たすべき性質を形式化して記述できる。また、データ型に応じたランダムなテストデータを生成して、記述した性質をテストできる。しかし性質の形式化をおこなう際に、多くの補助関数が必要になる場合がある。またテストをおこなうときに完全にランダムなテストデータを生成してしまうと、非常に効率の悪いテストになってしまう場合がある。効率の良いテストをおこなう為には、形式化された性質に基づく、適切なテストデータ生成器を考える必要がある [2]。これらの QuickCheck 使用者への負担は、あらかじめ問題の種類に応じた、性質の形式化を補助する関数とテストデータの生成器を用意することで緩和できると考えられる。

性質の形式化を補助する関数は、問題の持つ共通の性質を見つけることで設計できると考えられる。例えば最適化問題にはマトロイド [8] により定式化可能なものがある。マトロイドは離散最適化で使われる論理の一つで、マトロイドにより定式化された最適化問題は単純な貪欲法により、その最適解が求まる。マトロイドを Haskell のプログラムで形式化できれば、マトロイドで定式化可能な最適化問題についても同様のことが可能になる。また、形式化された最適化問題も貪欲法により最適解の導出が可能である。ある解が対象の問題の最適解かどうか（以降、解の最適性と呼ぶ）を示す性質の形式化を補助する関数の設計は、貪欲法による最適解の導出の過程が参考にできる。

本研究では、組み合わせ最適化問題に対するテスティングフレームワークの開発を目的とする。このフレームワークが提供するものは以下の 2 つである。

- QuickCheck による解の最適性を検査する性質の形式化を補助する関数
- 最適化問題を解くプログラムで頻繁に扱われるデータ構造に応じたテストデータ生成器

本研究では以下の二つを実装したテスティングフレームワークの開発を目標とする。

- マトロイドで定式化可能な最適化問題について、解の最適性を示す性質の形式化を補助する関数
- 最適化問題を解くプログラムで頻繁に用いられるグラフデータを生成する、ランダムグラフ生成器

2. 準　　備

この節では本稿を読むのに必要な知識を述べる。2.1 節では QuickCheck による性質の

†1 会津大学
University of Aizu

形式化と形式化された性質のテスト、テストデータの生成器などについて述べる。2.2 節ではマトロイドの定義と、マトロイドによる最適化問題の定式化について述べる。

2.1 QuickCheck

QuickCheck は Haskell プログラムのテストのためのコンビネータライブラリである。QuickCheck では、プログラムが満たすべき性質を Haskell のプログラムとして形式化し、その性質をランダムなテストデータを用いてテストできる。テストが失敗するとそのときのテストデータが出力されるため、性質に対する反例を見つけるのに便利である。また QuickCheck では、性質のテストに用いるデータの生成器も記述できる。これらの機能は `QuickCheck` モジュール内で実装されている。ここでは `QuickCheck` の実装については触れず、性質の形式化と検査、生成器の定義についてのみ説明する。実装の詳細は文献 [3] を参照のこと。

2.1.1 性質の形式化と検査

まず性質の形式化について述べる。単純な性質は、結果が `Bool` 型の値になる関数として定義できる。例えば、「整数のリストを 2 回反転させたものは元のリストと等しい」というリスト反転関数 `reverse` が満たすべき性質は以下のように書ける。

```
prop_RevRev :: [Int]→Bool
prop_RevRev xs = reverse (reverse xs) == xs
```

記述した性質をテストするときは、関数 `quickCheck` を以下のように呼び出す。

```
quickCheck prop_RevRev
```

QuickCheck は、ランダムに生成されたテストデータを用いて、記述した性質が成り立つことを検査する。デフォルトでは 100 個のテストデータが生成される。上の例では型シグネチャで指定した `[Int]` 型の値が 100 個生成され、テストデータとして用いられる。テストが成功すると、以下のような結果が返る。

```
+++ OK, passed 100 tests.
```

一方で、テストが失敗すると QuickCheck は性質に対する反例を返す。例えば誤ったリスト反転関数 `reverse'` と、この関数が満たすべき性質を以下のように与える。

```
reverse' [] = []
reverse' (x : xs) = xs ++ [x]
prop_RevRev' :: [Int]→Bool
prop_RevRev' xs = reverse' (reverse' xs) == xs
```

この性質のテストは以下のような結果を返す。

```
quickCheck prop_Rev
*** Failed! Falsifiable (after 5 tests and 4 shrinks):
[0,0,1]
```

これは、`[0,0,1]` がテストで見つかった、この性質の反例であることを表す。

2.1.2 生 成 器

QuickCheck によるテストで用いられるテストデータは、生成器によって生成される。QuickCheck では Haskell の基本的なデータ型に対する完全にランダムなデータの生成器 `arbitrary` があらかじめ用意されている。しかし完全にランダムなデータではなく何らかの規則に従ったテストデータや、ユーザが定義したデータ型に対するテストデータを用いる場合は、ユーザーが生成器を定義する必要がある。

生成器は型構成子 `Gen` からなるモナドとして定義されている。したがって、Haskell のモナド演算子が利用できる。生成器 `do { x ← s; e }` は、集合 $\{e|\ x \in s\}$ とみなせる。以下に、Classen と Hughes による生成器の定義の例を紹介する [1]。

任意の整数 x より大きい整数の集合 $\{y|\ y \leq x\}$ は、$\{x + abs\ n \mid n \in \mathbb{Z}\}$ と表せるため、その生成器は以下のように定義できる。

```
atLeast :: Int→Gen [Int]
atLeast x = do n ← arbitraray
               return (x + abs n)
```

2.2 マトロイド

マトロイドは離散最適化の分野に用いられる数学的構造の一つである。組み合わせ最適化問題の多くがマトロイドで定式化できる。マトロイドが重要である主な理由は、マトロイド上の最適化問題に単純な貪欲法が適用できることである。本研究ではこの性質に着目し、解が最適解かどうかを検査するアルゴリズムを設計する。このアルゴリズムについては 4.1 節で述べる。マトロイドに関する性質は多数存在するが、本節ではマトロイドによる最適化問題定式化のために必要な最低限の知識のみ述べる。詳細は文献 [8] を参照のこと。

2.2.1 定 義

マトロイドとは、有限集合 E とその部分集合族 $\mathcal{F} \subseteq 2^E$ について、以下の条件を満たす順序対 $M = (E, \mathcal{F})$ のことである。

(1) $\emptyset \in \mathcal{F}$ である
(2) $X \subseteq Y \in \mathcal{F}$ ならば、$X \in \mathcal{F}$ である
(3) $X, Y \in \mathcal{F}$ かつ $|X| > |Y|$ ならば、$Y \cup \{x\} \in \mathcal{F}$ となる $x \in X \backslash Y$ が存在する

GREEDY($M = (E, \mathcal{F}, c)$)

1: $A = \emptyset$
2: **for** $c(x)$ の降順で各 $x \in E$ を参照 **do**
3: **if** $A \cup \{x\} \in \mathcal{F}$ **then**
4: $A = A \cup \{x\}$
5: **return** A

図 1 重み付きマトロイド上の最適化問題に対する貪欲アルゴリズム

$\mathcal{F}$ は E の独立集合族と呼ばれ、$\mathcal{F}$ の要素は独立集合と呼ばれる。$F \in \mathcal{F}$ について、$F \cup \{x\} \in \mathcal{F}$ であるとき、x を F の拡張という。F が拡張をもたないとき、F を極大であるという。また $X \subseteq E$ について、X の部分集合のうち極大な独立集合を X の基という。本稿では特に断りがない限り E の基を単に基と記述する。

2.2.2 重み付きマトロイドと貪欲法

マトロイド $M = (E, \mathcal{F})$ に対して、関数 $c : E \to \mathbb{R}$ が与えられるとき、$M = (E, \mathcal{F}, c)$ を重み付きマトロイドと呼び、c をコスト関数と呼ぶ。任意の $F \in \mathcal{F}$ について、コスト関数 c を拡張した関数

$$\hat{c}(F) = \sum_{x \in F} c(x)$$

と定義する。このとき、重み付きマトロイドを用いて $\hat{c}(F)$ を最大にする F を求める問題は最適化問題としてみなせる。以下では重み付きマトロイドによって定式化可能な最適化問題を重み付きマトロイド上の最適化問題と呼ぶ。また重み付きマトロイド上の最適化問題の最適解を最適部分集合と呼ぶ。$c : E \to \mathbb{R}_+$ ($\mathbb{R}_+$ は正の実数の集合) であるとき、最適部分集合 F は常に基である [10]。本稿ではコスト関数 c の値域はつねに $\mathbb{R}_+$ であるものとする。

重み付きマトロイドの最適部分集合は単純な貪欲法で求められるので、重み付きマトロイド上の最適化問題には単純な貪欲法が適用できる。重み付きマトロイド上の最適化問題を解くアルゴリズム GREEDY [4] の疑似コードを図 1 に示す。GREEDY は任意の重み付きマトロイド $M = (E, \mathcal{F}, c)$ を入力とし、最適部分集合 A を返す。c の値域は $\mathbb{R}_+$ なので、A は必ず基になる。

このアルゴリズムは、各要素 $x \in E$ を重みの降順で調べ、$A \cup \{x\}$ が独立ならば直ちに集合 A に加えることから貪欲アルゴリズムとみなせる。

2.2.3 最小全域木問題の定式化

最小全域木問題に対する、重み付きマトロイドによる定式化の例を示す [4]。最小全域木問題とは、連結無向グラフ $G = (V, \mathcal{E})$ と各辺の重みを返す関数 $w : \mathcal{E} \to \mathbb{R}_+$ が与えられたとき、全頂点を連結する辺集合で重みの総和が最小のものを求める問題である。この問題は重み付きマトロイド $M_G = (E_G, \mathcal{F}_G, c)$ を用いて、以下のように定式化できる。

- $E_G = \mathcal{E}$
- $\mathcal{F}_G = \{F \subseteq \mathcal{E} :$ グラフ (V,F) は森である $\}$
- $c(x) = w_0 - w(x)$
  (ただし w_0 はどの辺の重みよりも大きい定数)

このとき $\mathcal{F}_G$ の基 A は、それぞれ $|V| - 1$ 本の辺を持つ全域木に対応し、任意の基 A に対して

$$\begin{aligned}\hat{c}(A) &= \sum_{e \in A} c(e) \\ &= \sum_{e \in A} (w_0 - w(e)) \\ &= (|V| - 1)w_0 - \sum_{e \in A} w(e) \\ &= (|V| - 1)w_0 - w(A)\end{aligned}$$

であるため、$\hat{c}(A)$ が最大であるとき、$w(A)$ が最小となる。したがって、与えられた重み付きマトロイドの最適部分集合を求めるアルゴリズム GREEDY によって、最小全域木を解くことができる。特に、この問題に対する GREEDY は、Kruskal 法 [9] の疑似コードそのものである [4]。

3. フレームワーク

この節では本研究で開発したテスティングフレームワークの概要を述べる。本研究のテスティングフレームワークは QuickCheck のプログラムテストで動作する解の検査プログラムと、重み付きグラフのテストデータ生成器を提供する ⋆2。これらの設計と実装の詳細

⋆2 本研究のテスティングフレームワークは従来のもののようにテスト環境を構築するものではない。

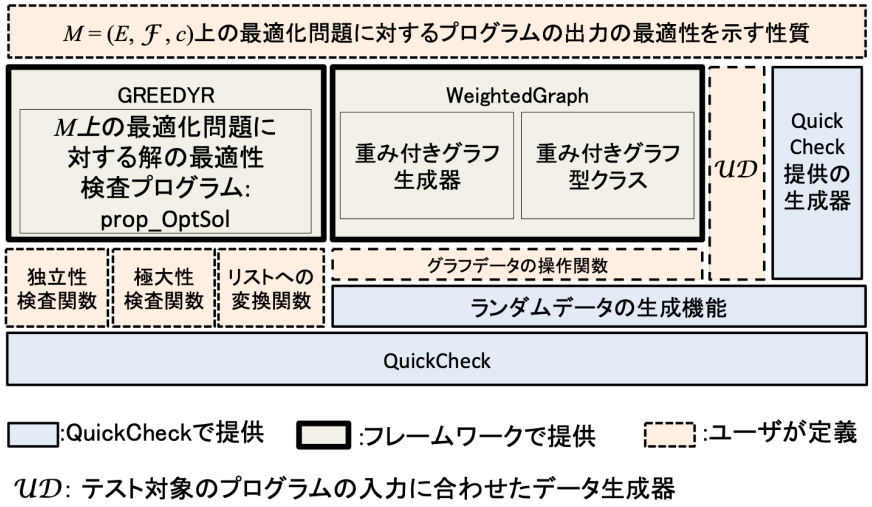

図 2　フレームワークの構成図

はのちの節で述べる。

3.1 構　　成

本研究のテスティングフレームワークの構成を述べる。フレームワークの構成を図 2 に示す。図 2 ではフレームワークが提供する要素を太い枠線で表し、ユーザーが定義するものを点線の枠で囲んで表す。テスティングフレームワークは 2 つのモジュール `GREEDYR` と `WeightedGraph` から構成される。これらのモジュールはユーザーが定義する QuickCheck の性質で利用できる関数を提供する。`GREEDYR` が提供する `prop_OptSol` は重み付きマトロイド $M = (E, \mathcal{F}, c)$ 上の最適化問題に対する解の最適性を検査するプログラムで、QuickCheck の性質として定義されている。`prop_OptSol` では集合はその要素を 1 対 1 の関係で含むリストで実装される。ただしリストの要素の順序に意味はないものとして実装されている。

`prop_OptSol` は、解の最適性検査のために以下の関数を用いる。

- ある解 $A \subseteq E$ について $F \in \mathcal{F}$ を検査する独立性検査関数
- A が基であるかを検査する極大性検査関数
- テストデータまたは検査対象のプログラムの出力をリストへ変換する関数

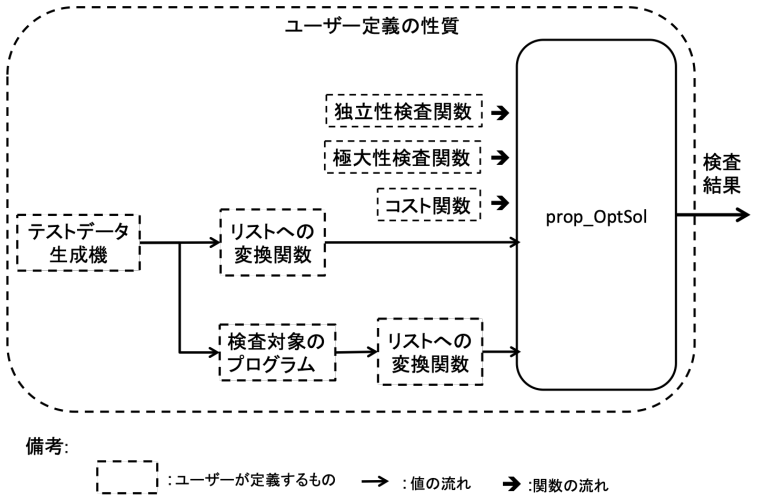

図 3　フレームワークの実行

これらの関数はユーザーが用意する必要がある。独立性検査関数と極大性検査関数は、解の最適性検査のために必要となる関数で用いられる。またリストへの変換関数はプログラムの入出力がリストとは限らず、またリストであっても適切なリストではない場合があるため必要となる。`prop_OptSol` 実装の詳細は 4.2 節を参照のこと。

`WeightedGraph` モジュールは重み付きグラフ生成器 `graphGen` と重み付きグラフ型クラス `WeightedGraph` から構成される。`graphGen` はグラフデータの操作関数を用いてグラフデータを生成する。グラフデータの操作関数は重み付きグラフ型クラスのメソッドであり、ユーザーは重み付きグラフ型クラスのインスタンスとして、グラフデータの操作関数を与える必要がある。実装の詳細は 5.2 節を参照のこと。

グラフデータ生成器は、テスト対象のプログラムの入力がグラフデータである場合に有用である。そうでない場合はユーザがプログラムのテストデータ生成器を定義する。

3.2 実　　行

フレームワークを用いて定義した性質の検査の実行について述べる。実行の様子を図 3 に示す。実行のためにユーザーが用意するものを図 3 では点線の枠で囲んで表す。テストデータ生成器はテストデータを生成する。テストデータにグラフデータを用いるときは、本研究で提供する `graphGen` が利用できる。生成されたテストデータはリストへの変換関数と

GREEDY_REVERSE($A, M = (E, \mathcal{F}, c)$)

1: **for** $c(x)$ の昇順で各 $x \in A$ を参照 **do**
2: A = A - {x}
3: $e = (E \backslash A)$ のうち $A \cup \{e\} \in \mathcal{F}$ を満たす重み最大の要素
4: **if** $c(e) \neq c(x)$ **then**
5: **return** $False$
6: **return** $True$

図 4 解の最適性検査アルゴリズムの疑似コード

検査対象のプログラムに渡される。リストへの変換関数に渡されたテストデータは適切なリストに変換され、prop_OptSol に渡される。検査対象のプログラムは渡されたテストデータに応じた解を出力し、この出力はリストへの変換関数を経由して prop_OptSol に渡される。prop_OptSol はこれらの入力と、ユーザーが与えた独立性検査関数、極大性検査関数、コスト関数を用いて検査対象のプログラムの出力が最適解であるかを検査し、その結果を出力する。

4. 解の最適性検査アルゴリズム

4.1 設　　計

あるプログラムの出力が $M = (E, \mathcal{F}, c)$ 上の最適化問題に対する最適解であるか検査するアルゴリズムの設計について述べる。GREEDY による部分集合の導出を参考に設計した最適性検査アルゴリズム GREEDY_REVERSE を図 4 に示す。このアルゴリズムは $M = (E, \mathcal{F}, c)$ と E の任意の基 A を受け取り、A が M に対する最適解であるかを判定する。GREEDY_REVERSE は GREEDY の解の導出を遡るように動作する。以下にその様子を述べる。

- GREEDY_REVERSE は A の要素をコストの昇順で参照ことで GREEDY で解の要素を加える順序とは逆順に A の要素を参照する。
- GEEDY は解の拡張となる要素を重みの大きい順に 1 つずつ加えるのに対し、GREEDY_REVERSE は A から要素を取り除き、その要素が A に対してその時点で一番重みの大きい拡張であったかを調べる。
- GREEDY は空集合から最適部分集合を導出するのに対し、GREEDY_REVERSE は A が空集合になるまで動作したときのみ A が最適部分集合であることを表す $True$ を返す。

4.1.1 証　　明

GREEDY_REVERSE は入力 $M = (E, \mathcal{F}, c)$ とその任意の基 A について、A が M の最適解であるかを正しく検査する。以下のその証明を述べる。

補題 1. $(E, \mathcal{F})$ をマトロイドとする。$X, Y \in \mathcal{F}$ かつ $|X| \geq |Y|$ ならば、$|X| = |Y| + |Z|$, $Y \cup Z \in \mathcal{F}$ を満たす $X \backslash Y$ の部分集合 Z が存在する。

証明: $|X| - |Y|$ に関する帰納法を用いる。$|X| - |Y| = 0$ のとき、$Z = \emptyset$ は上の条件を満たす。$|X| - |Y| > 0$ とする。マトロイドの条件 3 より、$Y \cup \{x\} \in \mathcal{F}$ を満たす $x \in X \backslash Y$ が存在する。帰納法の仮定より、$|X| = |Y \cup \{x\}| + |Z'|$ と $Y \cup \{x\} \cup Z' \in \mathcal{F}$ を満たす $Z' \subseteq X \backslash (Y \cup \{x\})$ が存在する。このとき、$Z = Z' \cup \{x\}$ は補題の条件を満たす $X \backslash Y$ の部分集合である。 □

以下の 2 つの補題では、GREEDY_REVERSE の実行での i 回目の繰り返し $(i \geq 1)$ で選ばれる x と e をそれぞれ x_i と e_i、i 回目の繰り返しの 2 行目で得られる A を A_i で表す。また、$A_0 = A$ とする。

補題 2. M を重み付きマトロイド、A を M の最適部分集合とする。GREEDY_REVERSE(A,M) が n 回目の繰り返しを実行するなら、$c(e_n) = c(x_n)$ が成り立つ。

証明: n 回目の繰り返しで得られる A_n と e_n は $|A| \geq A_n \cup \{e_n\}$ を満たすので、補題 1 より $|A| = |A_n \cup \{e_n\}| + |Z|$ と $A_n \cup \{e_n\} \cup Z \in \mathcal{F}$ を満たす $Z \subseteq A \backslash (A_n \cup \{e_n\})$ が存在する。$e_n \notin A_n$ なので $|Z| = n - 1$ だが、$Z \subseteq A \backslash (A_n \cup \{e_n\}) = \{x_i \mid 1 \leq i \leq n\} \backslash \{e_n\}$ なので、$x_k \notin Z$ $(1 \leq k \leq n)$ を満たす x_k が唯一つ存在する。GREEDY_REVERSE の定義より $c(x_k) \leq c(x_n) \leq c(e_n)$ だが、$\hat{c}(A) \geq \hat{c}(A_n \cup \{e_n\} \cup Z) = \hat{c}(A) - c(x_k) + c(e_n)$ なので、$c(x_k) = c(e_n)$。したがって、$c(e_n) = c(x_n)$。 □

補題 3. $M = (E, \mathcal{F}, c)$ を重み付きマトロイド、A を E の基 $(|A| = m)$、B を M の最適部分集合とする。GREEDY_REVERSE(A,M) が $True$ を出力するなら、$0 \leq i \leq m$ を満たす i について、以下の条件を満たす B の部分集合 B_i が存在する。

- $\hat{c}(A_i) \geq \hat{c}(B_i)$
- $|B \backslash B_i| = i$
- $\forall x \in B_i . \forall y \in B \backslash B_i . c(x) \geq c(y)$

証明: $m-i$ に関する帰納法による。証明には、マトロイドの条件 1,2、補題 1、および、すべての E の基の濃度は等しいこと [10] を用いる。 □

定理 1. $M=(E,\mathcal{F},c)$ を重み付きマトロイド、A を E の基とする。A は M の最適部分集合であるとき、かつそのときに限り、GREEDY_REVERSE(A,M) の出力は $True$ になる。

証明: A が M の最適部分集合であるならば、GREEDY_REVERSE(A,M) の出力が $True$ になることは、補題 2 より明らか。

次に、A が E の基で、GREEDY_REVERSE(A,M) が $True$ を出力すると仮定する。B を M の最適部分集合とすると、補題 3 より $\hat{c}(A) \geq \hat{c}(B)$ となることがわかる。したがって、A は M の最適部分集合である。 □

4.1.2 適 用 例

GREEY_REVERSE による最小全域木問題に対する解の最適性の検査の例を以下に述べる。ただし以下では $G=(V,\mathcal{E})$ について、任意の二頂点 $s,t \in V$ の間の無向辺を (s,t) と表す。

図 5 で示したグラフ $G=(V,\mathcal{E})$ に対して、グラフ $T_1=(V,\mathcal{E}_1)$、$T_2=(V,\mathcal{E}_2)$ が最小全域木であるかどうかをそれぞれ検査する。ただし G の各辺に対する重み関数 $w: V\times V \to \mathbb{R}_+$ が与えられているものとし、$w0 \geq 5$ の定数とする。

2.2.3 節で示したように最小全域木問題は重み付きマトロイド $M_G=(E_G,\mathcal{F}_G)$ で定式化できる。この場合 A が E_G の基であるとは、グラフ (V,A) が木であることをいう。T_1、$T2$ は木であるため、GREEDY_REVERSE を適用するための事前条件を満たす。それぞれのグラフが最小全域木であるかの検査は以下の様におこなわれる。

GREEDY_REVERSE($\mathcal{E}_1, M_G$) の検査は次の様におこなわれる。$\mathcal{E}_1$ の要素をコスト関数 c について昇順に参照する。ここで参照されるのは (b,d) である。$E_G\backslash(\mathcal{E}_1\backslash\{(b,d)\})\cup\{e\} \in \mathcal{F}$ を満たし、$c(e)$ を最大にする辺を探す。このような e はこの場合 (a,c) と (b,d) の二通りが存在するが、どちらも検査対象の辺と同じコストであるため、この場合の検証は成功する。残りの辺 (c,d)、(a,b) についても同様の手順で検査がおこなわれ、それらは成功する。GREEDY_REVERSE は $True$ を返し、T_1 は G に対する最小全域木であることが示された。

GREEDY_REVERSE($\mathcal{E}_2, M_G$) の検査は次の様におこなわれる。$\mathcal{E}_1$ の要素をコスト関数 c について昇順に参照する。ここで参照されるのは (b,c) である。$E_G\backslash(\mathcal{E}_2\backslash\{(b,c)\})\cup\{e\} \in \mathcal{F}$ を満たし、$c(e)$ を最大にする辺を探す。そのような e はこの場合 (c,d) である。$c((c,d)) > c((b,c))$ なので GREEDY_REVERSE は $false$ を返す。以上からから G に対する最小全域木でないことが示された。

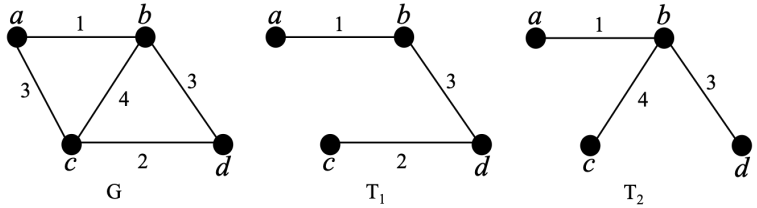

図 5 重み付き連結無向グラフ G とその全域木 T_1,T_2

4.2 実 装

4.1 節で述べた解の最適性の検査アルゴリズムの Haskell での実装について述べる。以下で述べる関数は本研究のフレームワークの `GREEYR` モジュールに格納される。

実装されるプログラムでは、集合の代わりにリストデータ構造を用いる。例えば任意のマトロイド $M=(E,\mathcal{F})$ について、集合 E を型変数 x を用いた $[x]$ 型のリストで表す。ただしリストの各要素は集合の各要素に対応する。リストがこのような構造を持つことで、集合の要素を昇順に参照し操作をおこなうアルゴリズムは、昇順にソートしたリストに対する再帰的な関数として実装できる。さらにリストに対する操作をおこなう関数はデフォルトで多く提供されているため、そのような関数の実装も容易になる。集合族 $\mathcal{F}$ を具体的に与える代わりに、$\{y\}\cup F \in \mathcal{F}$ $(y \in E)$ を調べる関数を用いる。本稿ではこれを独立性検査関数と呼ぶ。関数を用いることで $\mathcal{F}$ の要素を列挙する必要がなくなり、集合 F に要素 x を加えた集合が $\mathcal{F}$ の要素であるかどうかの判定をするだけで済むようになる。

最適性の検査プログラムは以下のような QuickCheck の性質として実装される。以下ではある最適化問題を重み付きマトロイドで定式化したものを $M=(E,\mathcal{F},c)$ とする。

```
prop_OptSol::(Ord x, Ord n) =>
([x]→Bool)→(x → n)→(x →[x]→Bool)→[x]→[x]→Bool
prop_OptSol p c f e a = p a && greedyR c f e a
```

型変数 x、n はそれぞれリストの要素の型とコストの値の型を表す。`prop_OptSol` の引数が表すものを以下に述べる。

- p は解 a が E の基であるかを検査する極大性検査関数である。
- c は x 型の値から重みを表す n 型の値を返すコスト関数である。
- f は $\{y\}\cup F \in \mathcal{F}$ を調べる独立性検査関数を表す。ただし集合はリストで実装する。

- e は E をリストで表したものである。
- a は任意の $[x]$ 型のリストで、$M = (E, \mathcal{F}, c)$ で定式化した最適化問題を解くプログラムの出力を渡す。

それぞれの引数の型は prop_OptSol の型シグネチャで与えた通りである。

prop_OptSol による解の最適性の検査は次のようにおこなわれる。prop_OptSol が上で述べた通りの引数を受け取ったとする。まず、&&の左辺の $p\ a$ は検査対象のプログラムの出力 a が E の基であるかを判定し Bool 型の値を返す。これは 4.1 節で述べた GREEDY_REVERSE を適用するための事前条件を表す。&&の右辺の関数 greedyR は GREEDY_REVERSE を Haskell で実装したものである。$p\ a$ が True を返すとき、greedyR による解の最適性の検査がおこなわれる。$p\ a$ が False であるとき、Haskell は遅延評価なので性質 prop_OptSol は greedyR を実行せずただちに False となる。

適　用　例

prop_OptSol を利用するときは、マトロイドによる最適化問題の定式化と引数で渡す関数やリストデータを用意する必要がある (3.2 節参照)。以下に最小全域木問題を解くプログラムについて、prop_OptSol を用いた解の最適性を表す性質の形式化の例を述べる。

プログラムで扱うグラフのデータ構造を以下のように定義する。

```
newtype Edge = Ed (Char,Char,Int)
newtype Graph = Gr [Edge]
```

頂点は Char 型の値で表し、辺の重みは Int 型の値で表す。Edge は辺を表す型で、その値 Ed (a, b, w) は頂点 a,b を繋ぐ無向辺と、辺の重みが w であることを表す。Graph 型はグラフを表す型で、その値 Gr $[x_1, x_2, .., x_n]$ は x_1 から x_n の n 本の辺からなる無向グラフであることを表す。このデータ構造で表された連結無向グラフについて、その最小全域木を計算するプログラムを実装し、正しさをテストする。ここでは最小全域木を計算するプログラム kruskal::Graph→Graph を文献 [11] より引用する。このプログラムの出力の正しさを表す性質 prop_OptMST は以下のように定義できる。ただし、ここでは辺に割り当てらえる最大の重みを maxW で与えるものとする。

```
prop_OptMST::Graph → Bool
prop_OptMST (Gr g) = prop_OptSol isTree c isForest g g'
  where
    (Gr g') = kruskal (Gr g)
    c (Ed (_,_,w')) = maxW - w'
```

Gr g' はテストデータ Gr g に対する kruskal の出力を表す。このとき Gr g を prop_OptSol の引数 e、Gr g' を prop_OptSol の引数 a とみなすために、型構成子を除いた [Edge] として扱う。この場合は型構成子を除くのみでリストへ変換できるが、検査対象のプログラムの入力に複雑なデータ構造を用いると、prop_OptSol の引数に与えるためにはリストへの変換関数が必要になる場合がある。c は辺の重みを返す関数を表す。isTree::[Edge]→Bool は prop_OptSol の引数の p に、isForest::Edge →[Edge]→Bool は prop_OptSol の引数の f に対応する関数をそれぞれ Haskell で実装したものである。

5. 重み付きグラフ生成器

5.1 設　　計

重み付きグラフ生成器は、テストで用いる際の利便性を考慮して以下のように設計されている。

- 任意のデータ構造で表された重み付きグラフが生成可能である。
- 生成されるグラフの頂点数と辺に割り当てられる重み値の下限と上限が指定できる。
- グラフの特徴を指定できる。

ここでグラフの特徴とは連結や非連結、有向や無向といったグラフやグラフの辺が持つ性質のことを示す。

グラフの特徴を生成器に与えることで無駄なグラフデータの生成を防ぐことができる。例えば最小全域木問題について、この問題を解くプログラムの出力が問題の最適解であることを示す性質をテストする場合を考える。この性質のテストに必要なテストケースは連結グラフである。ここで生成器に連結グラフだけを生成する術がない場合、性質にグラフの連結性を表す事前条件記述してそのようなテストデータを排除する必要がある。すると別途でグラフの連結性を判断するプログラムが必要になり、テストの効率も悪くなるという問題がある。グラフ生成器を連結グラフのみを生成するように制限できればこの問題は解決する。

5.2 実　　装

5.1 で述べたグラフ生成器の Haskell での実装を述べる。本研究ではグラフ生成器が任意のデータ構造で表されたグラフを生成できるようにするために、Haskell の型クラスシステムを用いて重み付きグラフ型クラスを定義する。グラフ生成器は重み付きグラフ型クラスのメソッドを用いて実装され ている。本研究で実装した重み付きグラフ型クラスと重み付きグラフ生成器は WeightedGraph モジュール内に記述される。

値	指定した場合
REFLEXIVE	自己ループを持つ場合がある
BIPARTITE	二部グラフのみ生成する
ACYCLIC	閉路無しのグラフのみ生成する
CONNECTED	連結グラフのみ生成する
DIRECTED	有向グラフのみを生成する
MULTIPLE	グラフは多重辺を持つ場合がある
COMPLETE	完全グラフのみを生成する
TREE	木となるグラフのみを生成する

表 1 グラフの特徴の指定による生成器のふるまい

5.2.1 型クラス

重み付きグラフ型クラスの定義は以下の通り。

```
class (Eq v,Ord v,Enum v,Rondom w,Num w,Arbitrary w)
 =>WeightedGraph g e v w | g → e,v → e,e → v,e → w,e → g where
                \* methods *\
```

一行目の定義は型クラスのインスタンスとなる型変数の制約を並べたタプルである。二行目の | の左辺の WeightedGraph がこのクラスの名前で、その次に並ぶ g, e, v, w それぞれ、グラフ、辺、頂点、重みを表す型変数である。また右辺は型変数同士の依存関係を表す。WeightedGraph はインスタンスとなる型の間に依存関係を導入し、それぞれの型の依存関係からグラフのデータ構造を表す。例えば $g \to e$ はグラフを表す型から辺を表す型が定まるという依存関係を表す。またこの関係は関数的である。例えば関係 $g \to e$ について、グラフの型 g がもつ辺の型 e は一意に決まる。ここでは $g \to e$ とは逆向きの依存関係 $e \to g$ も定義されている。どちらの依存関係も関数的であるため、このとき g と e はインスタンスの型に対して 1 対 1 の関係となる [7]。

5.2.2 生成器

生成器に指定可能なグラフの特徴と、特徴を指定した場合の生成器のふるまいを表 1 に示す。これらの特徴は以下のデータ型宣言で定義する。

```
data GRAPHTYPE = REFLEXIVE|BIPARTITE|ACYCLIC|
     CONNECTED|DIRECTED|MULTIPLE|COMPLETE|TREE
```

これらのグラフの特徴は [GRAPHTYPE] 型のリストで生成器に渡して指定する。モジュール内では [GRAPHTYPE] 型は型シノニムを使って

頂点数	1000	5000	10000	20000	40000	80000	100000
生成の時間 (sec)	0.052	1.92	10.1	53.5	346	1712	2812

表 2 graphGen が頂点数ごとの単純無向グラフを 1 つ生成するのにかかる時間

```
type ENABLES = [GRAPHTPYE]
```

と定義する。グラフの特徴の間で矛盾が生じる場合、生成器は暗黙の内に矛盾を取り除くか、エラーを返す。例えば、[REFLEXIVE,BIPARTITE] を生成器に渡した場合、自己ループ辺を持つグラフは二部グラフの定義と矛盾するため生成器はエラーを返す。一方 [COMPLETE] を渡した場合、連結グラフである必要があるので暗黙のうちに CONNECTED を指定する。また、TREE は指定すると ACYCLIC、CONNECTED を暗黙のうちに指定し、DIRECTED を指定から外す。

実装するグラフ生成器では、頂点数と重みの上限と下限をそれぞれ指定できる。これらは型シノニムを用いて

```
type VNumRange = (Int,Int)
type WRange w= (w, w)
```

と定義する。これらの型の値は、実装したグラフ生成器内では QuickCheck が提供する choose::(a,a)->Gen a に渡される。choose は指定された値のペアの間からランダムに値を選択する生成器である。例えば (1,5)::(Int,Int) を与えると 1 から 5 の間の Int の値がランダムに生成される。

実装されたグラフ生成器 graphGen の型を以下に示す。

```
graphGen::WeightedGraph g e v w =>
          WRange w →VNumRange→ENABLES→Gen g
```

graphGen は WeightedGraph クラスのメソッドを用いて実装されるため、グラフを表す型は WeightedGraph クラスのインスタンスである必要がある。w は辺の重みを表す型変数でなので WRange w は重み値のペアを表す型となる。

graphGen の型シグネチャを見ると、引数からは g と w の型しか定まらないことがわかる。しかし、引数では与えられない e と v の型も WeightedGraph のインスタンスになっているため、型の依存関係による型推論から定まる。そのため、graphGen は引数からグラフの生成に必要となるすべての型が定まる。

6. まとめ

本研究の成果として、重み付きマトロイドで定式化可能な最適化問題を解くプログラ

頂点数	200	400	600	800	1000
グラフ生成の時間 (sec)	0.003	0.01	0.02	0.04	0.06
テスト時間 (sec)	0.175	1.71	5.34	13.7	27.4

表 3 1 つのグラフデータの生成とそれを用いた prop_MST の 1 回の検査時間

ムのテスティングフレームワークを Haskell で実装できた。また、本研究では最小全域木問題、タスクスケジューリング問題、割当問題についてそれぞれの問題を解くプログラムのテストに本研究のフレームワークが適用できた。この節では本研究で実装したフレームワークについて、まず実装した prop_OptSol と graphGen の性能の評価を述べる。次にこのフレームワークの評価を踏まえて今後の課題を述べる。

6.1 性能評価

まず本研究で prop_OptSol が適用できることを確認した問題について、それぞれの評価を述べる。prop_OptSol の計算時間については 6.2 節で詳しく述べる。ただし以下ではそれぞれの問題は重み付きマトロイド $M = (E, \mathcal{F}, c)$ で定式化されているものとする。またそれぞれの問題に対するプログラムのテストのために prop_OptSol に与えた極大性検査関数と独立性検査関数は効率を突き詰めて実装されたものではなく、著者が確認のために実装したものである。

最小全域木問題では、検査対象の出力が基でないときに $\mathcal{O}(|E|^3)$、基であるとき $\mathcal{O}(|E|^5)$ となった。タスクスケジューリングでは、検査対象のプログラムの出力が基でないとき $\mathcal{O}(|E|\mathbf{log}\ |E|)$、基であるときは $\mathcal{O}(|E|^3\mathbf{log}\ |E|)$ となった。割当問題では、検査対象のプログラムの出力が基でないとき $\mathcal{O}(|E|^2)$、基であるときは $\mathcal{O}(|E|^5)$ となった。

次に graphGen によるグラフ生成にかかる時間の計測結果を述べる。計測用いるデータ型と WeightedGraph クラスのインスタンスを図 6 に示す。指定した頂点数の単純無向グラフ（graphGen の第 3 引数に空リストを渡した場合）の生成にかかる時間の測定結果を表 2 に示す。

graphGen は 1000 個頂点を持つグラフは約 0.05 秒、10000 個の頂点を持つグラフは約 10 秒で生成された。QuickCheck によるテストでは 100 のテストデータを生成するため、その場合テストデータの生成時間は約 100 倍になる。したがって graphGen によるグラフの生成には、頂点数が 100 個のグラフは約 5 秒、頂点数が 10000 個のグラフは約 17 分を要すると推測される。

次に prop_OptSol と graphGen を用いたプログラムテストにかかる時間の計測結果を述べる。計測のために具体的な問題として最小全域木問題を扱う。

プログラムの出力が最小全域木であることを示す性質は 4.2 節で述べた prop_OptMST を用いる。graphGen で指定した頂点数の連結無向グラフを 1 つ生成し、そのデータに対する prop_OptMST の 1 回の検査が終了するまでの時間を計測する。グラフの頂点数に応じたこの計測の結果を表 3 に prop_OptMST に示す。graphGen は 1000 個の頂点を持つ連結無向グラフを約 0.06 秒で生成できるのに対し、prop_OptMST の検査は約 27 秒かかった。この場合 QuickCheck で prop_OptMST を 100 回テストするのに 46 分程度かかると推測される。

実装された現在の graphGen と prop_OptSol はデータサイズの小さいテストでは充分な速度で動作する。そこで小さなデータサイズのテストデータでも、ある程度の信頼性を得らるテスト方法を紹介する。これはサイズの小さいテストデータを用いてテストの試行回数を増やす手段である。これは「ほとんどのバグは小さな判例を持つ」ことから、テストデータがとりうる範囲 (以下スコープと呼ぶ) を小さな範囲に限定し、その範囲内で取りうるすべてのテストデータをテストすればバグが見つかるだろうという小スコープ仮説 [6] の考え方に基づく。

小スコープ仮説に基づく prop_OptMST のテストは次のように考えられる。テストに用いるグラフデータを頂点を区別しない連結グラフ（以下、連結非標識グラフと呼ぶ）として考える。Cadogan の公式 [5] から頂点数別のユニークなグラフの総数が求まる。例えば頂点数が 9 個の連結非標識グラフの総数は 261080 通りある。prop_OptMST に対して 9 個の頂点をもつグラフデータによるテストから高い信頼性を得るには 261080 回試行すればよい。この場合 prop_OptMST のテストは約 32 秒で終わる。

ただし上記で述べたテスト方法には問題がある。生成器が生成するデータは重複する場合がある。つまりスコープの範囲から取りうるデータの総数と同じ回数テストを試行しても、スコープの範囲内のすべてのデータについてテストしたとは限らない。テストの信頼性をあげるのであれば、テストデータの総数より多い回数テストの試行するべきである。本稿の筆者は現時点で QuickCheck の生成器が生成するテストデータから重複を取り除く手段があるかはわからない。

6.2 今後の課題

研究の過程で気づいた課題点を以下に述べる。

- prop_OptSol は評価時間に改善の余地がある。独立性検査関数 f の計算量を $\mathcal{O}(f(E))$ で表すとすると、GREEDY_REVERSE の計算量は $\mathcal{O}\ (f(E) \times |E|\mathbf{log}|E|)$ となる。一方このアルゴリズムを実装した greedyR では、リスト構造では要素の参照に $\mathcal{O}(E)$ か

```
newtype Edge = Eg (Char,Char,Int)
newtype Graph = Gr [Edge]
instance WeightedGraph Graph Edge Char Int where
        \ * methods * \
```

図 6 グラフ生成の測定に用いるデータ型

かるので、$\mathcal{O}\left(|E|^2\mathbf{log}|E| \times f(E)\right)$ となった。`greedyR` はリストデータを扱う関数として実装されているが、代わりに Haskell の `Array` などの配列構造を用いればアルゴリズム本来の計算量に近づけられる。

- `prop_OptSol` を利用するためには、最適化問題をマトロイドで定式化する必要がある。また、`prop_OptSol` の引数に応じた関数を用意しなければならない。最適化問題がマトロイドで定式化可能であるかを判断するアルゴリズムや、定式化を補助する性質などが見つかれば、本研究のフレームワークがさらに利用しやすくなる。
- さらに広い範囲での最適問題に対して解の最適性を示す手法が開発できれば、フレームワークの適用範囲の拡大が期待できる。これにはマトロイドの双対性やマトロイド交差などの公理などが利用できるのではないかと推察される。
- グラフ生成器の生成データが重複しないようにできれば小スコープ仮説に基づくテストができる。QuickCheck の生成器が重複したデータを生成しないようにする手段があるのであれば、本研究のグラフ生成器にも適用するべきである。
- 実装されたグラフ生成器の実装の見直しにより、データの生成時間の高速化が期待できる。グラフ生成器がグラフ生成にかかる時間は QuickCheck の仕様に起因するものではない。また実装されたグラフ生成器は効率を突き詰めて実装したものではないため、実装の見直しによる高速化は充分見込める。

参 考 文 献

1) Koen Claessen and John Hughes. *QuickCheck: An Automatic Testing Tool for Haskell*, 2003.
2) Koen Claessen and John Hughes. The fun of programming edited by jeremy gibbons and oege de moor, palgrave macmillan, 2003, isbn 1-4039-0772-2 (hb), 0-333-99285-7 (sb). *J. Funct. Program.*, 14(5):21–22, September 2004.
3) Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. *SIGPLAN Not.*, 46(4):53–64, May 2011.
4) ThomasH. Cormen, CharlesEric Leiserson, RonaldL. Rivest, Clifford Stein, 哲夫 浅野, 和生 岩野, 博司 梅尾, 雅史 山下, and 幸一 和田. アルゴリズムイントロダクション *pp365-366.* 世界標準 MIT 教科書. 近代科学社, 第 3 版, 総合版 edition, 2013.
5) Frank harary and EdgarM. Palmer. *A survey of Graphical Enumeration Problems.* North-Holland, 1973.
6) Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2006.
7) MarkP. Jones. Type classes with functional dependencies. In *Proceedings of the 9th European Symposium on Programming Languages and Systems*, ESOP '00, page 230–244, Berlin, Heidelberg, 2000. Springer-Verlag.
8) Bernhard Korte and Jens Vygen. *Matroids*, pages 321–353. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
9) J.B. Kruskal. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. In *Proceedings of the American Mathematical Society, 7*, 1956.
10) ChristosH. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, Inc., USA, 1982.
11) Fethi Rabhi and Guy Lapalme. *Algorithms; A Functional Programming Approach.* Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 1999.