グラフアルゴリズムの構成的定義と変換に関する研究

指導教官　武市 正人　教授

1999年1月29日

篠埜 功

目 次

第1章　序論

1.1　研究の背景

プログラムの信頼性、生産性の向上のためにアルゴリズムを構成的に記述し、正しさの証明、プログラム変換による効率改善、仕様からのプログラム導出などを行う手法に関する研究が以前から行われている。リスト、木の上のアルゴリズムに関しては多くの研究がなされており [1]、最近ではグラフアルゴリズムに関しても構成的記述が試みられている [2, 3, 4]。しかし、グラフは、リスト、木などのように自然に構成的に定義することが難しく、そのため、グラフアルゴリズムの構成的記述も難しい。グラフアルゴリズムの構成的記述は現在のところ、深さ優先探索のみを対象としたもの [2, 5] や、有向無閉路グラフのみを対象としたもの [3] など、適用範囲を限定して行われている。[5] においては深さ優先探索に関する融合変換 (fusion) が行われているが、融合規則 (fusion law) が提示されていない。[4] においては深さ優先探索をする関数に関する融合規則が提案されているが、非常に複雑であり、規則の適用が難しく、実用上の問題点がある。また、グラフ上のアルゴリズムを仕様から導出するということは行われていない。

1.2　研究の目的

グラフの構成的取り扱いのしやすさを改善、向上させることを研究の目的とする。

1.3　本研究で行ったこと

本論文では、グラフの一般的探索 [6] を行う関数を構成的に定義する。これは深さ優先探索、幅優先探索をその特別の場合として含むものであり、適用範囲が広いものである。この一般的探索関数は Hylomorphism という形に変換することができ [7, 8]、それにより、Hylo fusion と呼ばれる融合変換やその他いろいろな変換を行うことができる。また、グラフの一般的探索関数の関数型言語 Haskell による 1 つの効率のよい実現法を示す。

また、グラフアルゴリズムに関しても仕様からのアルゴリズムの導出が可能であることを最短路問題を例として取り上げて示す。

1.4 本論文の構成

グラフの構成的定義 (第 2 章)

自然数、リスト、木を構成的に定義することにより、構成的定義とはいかなるものであるかを概観したのち、どのようにすればグラフを構成的にとらえることができるかを述べる。

グラフの探索に関するアルゴリズムの記述 (第 3 章)

構成的に定義されたグラフ上でグラフの探索に関するアルゴリズムを構成的に記述する。

グラフアルゴリズムの変換 (第 4 章)

一般的探索関数を Hylomorphism と呼ばれる形で記述し、いくつかの例について Hylo fusion と呼ばれる融合変換を行う。

グラフアルゴリズムの導出 (第 5 章)

グラフアルゴリズムの仕様からの導出が可能であることを最短路問題を例として取り上げて示す。

結論 (第 6 章)

本研究のまとめを行い、今後の課題について述べる。

第2章　グラフの構成的定義

自然数、リスト、木などに関するアルゴリズムを記述するときに自然数、リスト、木などを構成的に定義すると、アルゴリズムも自然に構成的に定義できる。そうすることにより、プログラム変換、プログラムの正しさの証明、仕様からのプログラムの導出などを行いやすくなり、プログラムの信頼性、生産性が向上する。この章では、自然数、リスト、木を構成的に定義することにより、構成的定義とはいかなるものであるかを概観したのち、どのようにすればグラフを構成的にとらえることができるかを述べる。

2.1　自然数の構成的定義

自然数は、零であるかまたは自然数に 1 を加えたものとして定義できる。

$$Nat \ ::= \ Zero \mid Succ \ Nat$$

自然数 Nat の上の多くの関数は構成的に定義することができ、例えば、2 つの自然数の和を求める関数 $plus$ は

$$\begin{aligned} plus \ m \ Zero &= m \\ plus \ m \ (Succ \ n) &= Succ \ (plus \ m \ n) \end{aligned}$$

と定義することができ、また、2 つの自然数の積を求める関数 $mult$ は関数 $plus$ を用いることにより、

$$\begin{aligned} mult \ m \ Zero &= Zero \\ mult \ m \ (Succ \ n) &= plus \ m \ (mult \ m \ n) \end{aligned}$$

と定義することができる。ここで、関数の引数はカリー化 (currying)[9, 10] を行っている。以下では特に断ることなくカリー化を行うこととする。和や積を求める関数と同様にして、自然数上の階乗関数、冪乗関数なども構成的に定義することができる。関数の構成的定義 (definition by structural recursion)[1] とは、自然数の上の場合では

$$\begin{aligned} f \ Zero &= c \\ f \ (Succ \ n) &= h \ (f \ n) \end{aligned}$$

のような形をしているものをいい、データの構成子をあるもので置き換えるようになっているものをいう。自然数の場合では、上で定義した関数 f は引数に自然数が与えられるとその中の $Zero$ を c で置き換え、$Succ$ を h で置き換えたものを結果として返す。このような関数 f を

$$f = (\![c, h]\!)$$

のように記述することにすると、自然数上で構成的に定義できる関数はすべて $(\![\ ,\ ]\!)$ で記述できる。例えば、自然数上の和、積、冪乗を求める関数は

$$\begin{aligned} plus\ m &= (\![m, Succ]\!) \\ mult\ m &= (\![Zero, plus\ m]\!) \\ expn\ m &= (\![1, mult\ m]\!) \end{aligned}$$

のように記述できる。$(\![c, h]\!)$ は自然数に関する準同型写像 (homomorphism) であり、カテゴリー理論におけるカタモルフィズム (catamorphism) になっている [1]。この性質により、理論的取り扱いがしやすくなる。

2.2 リストの構成的定義

リストは、Nil であるかまたはリストの先頭にある型 α のある要素を付け加えたものとして定義できる。

$$Listr\ \alpha\ ::=\ Nil \mid Cons\ (\alpha, Listr\ \alpha)$$

リスト上の関数も自然数の場合と同様にして容易に定義することができる。例えば、

$$map\ f\ [a_1, a_2, \ldots, a_n] = [f\ a_1, f\ a_2, \ldots, f\ a_n]$$

のようにリストの各要素にある関数 f を適用する関数 map は、

$$\begin{aligned} map\ f\ Nil &= Nil \\ map\ f\ (Cons\ (a, x)) &= Cons\ (f\ a,\ map\ f\ x) \end{aligned}$$

のように構成的に定義できる。リスト上の構成的な関数は

$$\begin{aligned} f\ Nil &= c \\ f\ (Cons\ (a, x)) &= h\ (a,\ f\ x) \end{aligned}$$

という形をしており、このような関数 f を

$$f = (\![c, h]\!)$$

と記述する。自然数の場合と同様に、$(\![c, h]\!)$ は、引数にリストが与えられるとその中の Nil を c で、$Cons$ を h で置き換えたものを結果として返す。リスト上で構成的に定義できる関数はすべて $(\![\ ,\ ]\!)$ で記述できる。例えば、上で定義した map は

$$\begin{aligned} map\ f &= (\![Nil, h]\!) \\ &\text{where } h\ (a, x) = Cons\ (f\ a,\ x) \end{aligned}$$

のように定義できる。また、リストのすべての要素の和、積を求める関数は、

$$\begin{aligned} sum &= (\![0, plus]\!) \\ product &= (\![1, mult]\!) \end{aligned}$$

のように定義できる。$(\![c, h]\!)$ はリストに関する準同型写像 (homomorphism) であり、カテゴリー理論におけるカタモルフィズム (catamorphism) になっている [1]。

2.3 木の構成的定義

この節では、二分木 (binary trees) を構成的に定義してみる。二分木は、ある型 α の要素を持つ葉であるかまたは 2 つの木を子として持つものとして定義できる。

$$Tree\ \alpha\ ::=\ Tip\ \alpha \mid Bin\ (Tree\ \alpha,\ Tree\ \alpha)$$

二分木上の関数も自然数、リストの場合と同様にして容易に定義することができる。例えば、木に含まれている葉の個数を求める関数 $size$ は、

$$\begin{aligned} size\ (Tip\ a) &= 1 \\ size\ (Bin\ (x, y)) &= size\ x\ + size\ y \end{aligned}$$

のように構成的に定義できる。二分木の上の構成的な関数は

$$\begin{aligned} f\ (Tip\ a) &= g\ a \\ f\ (Bin\ (x, y)) &= h\ (f\ x,\ f\ y) \end{aligned}$$

という形をしており、このような関数 f を

$$f = (\![g, h]\!)$$

と記述する。自然数、リストの場合と同様に、$(\![g, h]\!)$ は、引数に二分木が与えられるとその中の Tip を g で、Bin を h で置き換えたものを結果として返す。二分木上で構成的に定義できる関数はすべて $(\![\ ,\ ]\!)$ で記述できる。例えば、上で定義した $size$ は

$$\begin{aligned} size\ &= (\![one, plus]\!) \\ &\text{where } one\ a = 1 \end{aligned}$$

のように定義できる。また、木の深さを求める関数は、

$$\begin{aligned} depth \ &= (\![zero,\ (Succ \ \circ \ bmax)]\!) \\ &\text{where } zero\ a = 0 \end{aligned}$$

のように定義できる。ここで、$bmax$ は自然数の対を引数としてとり、大きい方を結果として返す関数である。$(\![g, h]\!)$ は二分木に関する準同型写像 (homomorphism) であり、カテゴリー理論におけるカタモルフィズム (catamorphism) になっている [1]。

2.4 グラフの構成的定義

以上で自然数、リスト、木の構成的定義およびその上の構成的な関数について述べた。自然数、リスト、木は自然に構成的に定義することができ、また構成的な関数の記述も容易である。しかしグラフは、リスト、木などのように自然に構成的に定義することが難しく、そのため、構成的な関数の記述も難しい。また、グラフの探索では、探索の順序に任意性があり、構成的なグラフの探索の記述には工夫を要する。以下に、現在までに提案されている主なグラフの構成的定義を述べる。

2.4.1 構成的定義 1

グラフを以下のように構成的に定義する。

$$\begin{aligned} Graph \quad ::= \quad & Empty \\ | \quad & Edge \\ | \quad & Vert_{m,n} \\ | \quad & Swap_{m,n} \\ | \quad & Graph \parallel Graph \\ | \quad & Graph \ {}^{0}_{9} \ Graph \end{aligned}$$

このグラフの定義は有向無閉路グラフのみを対象とするものであり、グラフを 6 つの構成子 $Empty,\ Edge,\ Vert_{m,n},\ Swap_{m,n},\ \parallel,\ {}^{0}_{9}$ から構成する [3]。それぞれの構成子の意味は以下の通りである。

- $Empty$ は何もない空のグラフを表す。
- $Edge$ は図 2.1 のような一本の枝を表す。
- $Vert_{m,n}$ は入ってくる枝が m 本、出ていく枝が n 本ある一つの頂点を表す。例えば、$Vert_{3,2}$ は図 2.2 のようなグラフを表す。

図 2.1: $Edge$

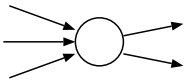


図 2.2: $Vert_{3,2}$

- $Swap_{m,n}$ は入力の最初の m 本が出力の最後の m 本になり、入力の最後の n 本が出力の最初の n 本になっているものを表す。例えば、$Swap_{3,2}$ は図 2.3 のようなグラフを表す。
- $x \parallel y$ はグラフ x とグラフ y を上下に並列に並べたグラフを表す。例えば、$Vert_{1,2} \parallel Vert_{2,1}$ は図 2.4 のようなグラフを表す。
- $x \begin{smallmatrix} 0 \\ 9 \end{smallmatrix} y$ はグラフ x から出ている枝とグラフ y へ入る枝を順に結び付けたものを表す。例えば、$Vert_{0,1} \begin{smallmatrix} 0 \\ 9 \end{smallmatrix} Vert_{1,0}$ は図 2.5 のようなグラフを表す。

以上のように、いくつかのグラフを組み合わせてグラフを構成していくので、頂点に接続していない枝があり、それが構成子 $\begin{smallmatrix} 0 \\ 9 \end{smallmatrix}$ によって結び付くようになっている。これらの 6 つの構成子を用いることにより、任意の有向無閉路グラフを表すことができる。この定義では、1 つのグラフの表現が複数存在し、例えば、図 2.6 のようなグラフは、

$$
\begin{aligned}
&(3 \times Vert_{0,2}) \begin{smallmatrix} 0 \\ 9 \end{smallmatrix} \\
&(Edge \parallel ((2 \times Swap_{1,1}) \begin{smallmatrix} 0 \\ 9 \end{smallmatrix} (Edge \parallel Swap_{1,1} \parallel Edge)) \parallel Edge) \begin{smallmatrix} 0 \\ 9 \end{smallmatrix} \\
&(2 \times Vert_{3,0})
\end{aligned}
$$

のように表現することもできるし、

$$
\begin{aligned}
&(3 \times Vert_{0,2}) \begin{smallmatrix} 0 \\ 9 \end{smallmatrix} \\
&(Edge \parallel (2 \times Swap_{1,1}) \parallel Edge) \begin{smallmatrix} 0 \\ 9 \end{smallmatrix} \\
&((2 \times Edge) \parallel Swap_{1,1} \parallel (2 \times Edge)) \begin{smallmatrix} 0 \\ 9 \end{smallmatrix} \\
&(2 \times Vert_{3,0})
\end{aligned}
$$

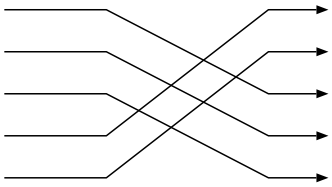


図 2.3: $Swap_{3,2}$

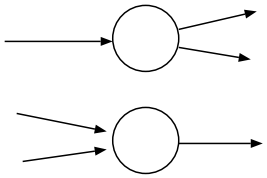


図 2.4: *beside*

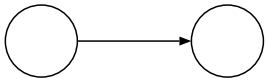


図 2.5: *before*

のように表現することもできる。ここで、$m \times x$ は、m 個の x が $\|$ を間にはさんで並んでいるものを表す。例えば、$3 \times Vert_{0,2}$ は

$$Vert_{0,2} \parallel Vert_{0,2} \parallel Vert_{0,2}$$

を表す。

このグラフ上の構成的な関数は一般に

$$\begin{aligned} f\ Empty &= a \\ f\ Edge &= b \\ f\ Vert_{m,n} &= v_{m,n} \\ f\ Swap_{m,n} &= s_{m,n} \\ f\ (x \| y) &= f\ x \oplus f\ y \\ f\ (x \,{}^{0}_{9}\, y) &= f\ x \otimes f\ y \end{aligned}$$

という形をしており、このような関数 f を

$$(\!|a,\ b,\ v,\ s,\ \oplus,\ \otimes|\!)$$

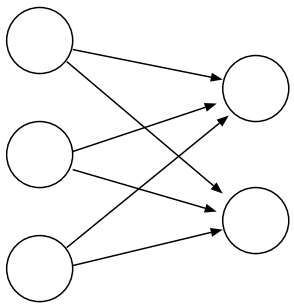


図 2.6: 有向無閉路グラフの例 1

と記述する。自然数、リスト、木の場合と同様に、$(\!|a,\ b,\ v,\ s,\ \oplus,\ \otimes|\!)$ は、引数にグラフが与えられるとその中の $Empty$ を a で、$Edge$ を b で、$Vert$ を v で、$Swap$ を s で、$\|$ を $\oplus$ で、$^0_9$ を $\otimes$ で置き換えたものを結果として返す。グラフ上で構成的に定義できる関数はすべて $(\!|\ ,\ ,\ ,\ ,\ ,\ |\!)$ で記述できる。例えば、すべての枝の向きを逆にする関数は

$$(\!|Empty, Edge, v, s, \|, \otimes|\!)$$
$$\text{where } v_{m,n} = Vert_{n,m},\quad s_{m,n} = Swap_{n,m},\quad t \otimes u = u^0_9 t,$$

のように定義できる。また、最短路を求める関数 sp は次のように定義できる。

$$sp = (\!|a, b, v, s, \oplus, \otimes|\!)$$

where

- a は 0×0 行列
- b は値 0 をもつ 1×1 行列
- $v_{m,n}$ は要素がすべて 1 の $m \times n$ の行列
- $s_{m,n}$ は $(m+n) \times (n+m)$ 行列で $\begin{pmatrix} A & B \\ C & D \end{pmatrix}$ の形をしており、A と D はそれぞれ m 行 n 列と n 行 m 列で要素はすべて ∞、B と C はそれぞれ m 行 m 列と n 行 n 列で対角成分はすべて 0 でそれ以外はすべて ∞。
- t と u がそれぞれ m 行 n 列と p 行 q 列の行列だったとすると、$t \oplus u$ は $(m+p)$ 行 $(n+q)$ 列の行列 $\begin{pmatrix} t & \infty \\ \infty & u \end{pmatrix}$ である。
- t と u がそれぞれ m 行 p 列と n 行 q 列の行列だったとすると、$t \otimes u$ は、半環 $(\min, +)$ における、t と u の行列積である。すなわち、

$$(t \otimes u)_{i,j} = \min_{1 \le k \le n} (t_{i,k} + u_{k,j}) \qquad \text{for } 1 \le i \le m,\ 1 \le j \le p$$

ただし、∞ は $+$ の零元であり、$\min$ の単位元である。結果として m 行 n 列の行列が返ってきたとすると、その行列の (i, j) 成分が、始点に頂点が接続していない枝のうち i 番目のものから終点に頂点が接続していない枝のうち j 番目のものへの最短路中の頂点数を表している。例えば、図 2.7 は

$$Vert_{1,2} \,{}^0_9\, ((Vert_{1,1} \,{}^0_9\, Vert_{1,1}) \,\|\, Vert_{1,1}) \,{}^0_9\, Vert_{2,1}$$

と表すことができ、これを sp の引数に与えると、

$$\begin{pmatrix} 1 & 1 \end{pmatrix} \otimes ((\begin{pmatrix} 1 \end{pmatrix} \otimes \begin{pmatrix} 1 \end{pmatrix}) \oplus \begin{pmatrix} 1 \end{pmatrix}) \otimes \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

となり、結果は $\begin{pmatrix} 3 \end{pmatrix}$ となる。これは、図 2.7 の左端の枝から右端の枝への最短路中の頂点数が 3 であることを表している。

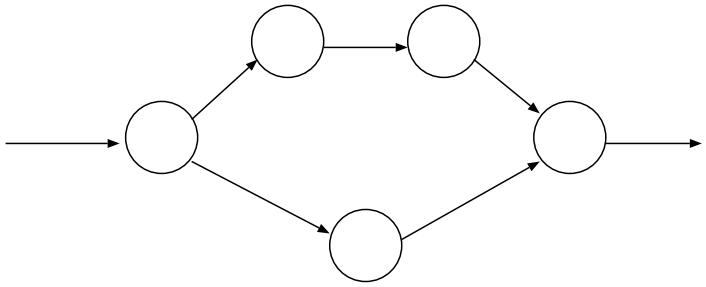


図 2.7: 有向無閉路グラフの例 2

2.4.2 構成的定義 2

構成的定義 1 は有向無閉路グラフのみを対象としているので、適用範囲が狭く、実用性に乏しい。以下で、任意の有向グラフを表現し得る構成的な定義を述べる。

グラフは、何もない *Empty* というグラフであるか、またはグラフに一つの頂点とそれに付随する枝 (*Context*) とを付け加えたものとして定義できる [4]。

$$\begin{aligned} Graph &::= Empty \mid Context\ \&\ Graph \\ Context &::= ([Vertex],\ Vertex,\ [Vertex]) \end{aligned}$$

この定義では、グラフはコンテクスト (*Context*) を最小単位として構成される。コンテクストはグラフ中の 1 つの頂点に関する情報を表しており、

- 第 1 要素はその頂点の predecessor(その頂点を終点とする枝の始点) のリスト、
- 第 2 要素はその頂点の名前、
- 第 3 要素はその頂点の successor(その頂点を始点とする枝の終点) のリスト

を表している。このコンテクストを順番に組み上げていくことにより、グラフを構成していく。例えば、図 2.8 のグラフは、次のように徐々に構成していくことができる。まず、

$$Empty$$

という、なにもない空のグラフから始める。これに、どれか一つの頂点を加える。例えば、d を加えるとすると、

$$([\,],\ d,\ [\,])\ \&\ Empty$$

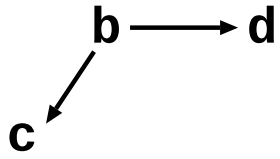
d

となる。対応するグラフも下に並べて書いてある。以下で頂点を一つずつ加えていくが、それぞれの段階の式は、1 つのグラフを表しており、それは図 2.8 のグラフの部分グラフでもある。それぞれの段階の式の predecessor と successor のリスト中には、その式の表している部分グラフ中にない頂点は含まれていない。c を加えると

$$([\,],\ c\ ,[\,])\ \&\ ([\,],\ d,\ [\,])\ \&\ \mathit{Empty}$$

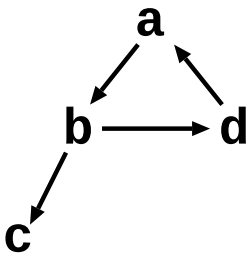
となり、b を加えると

$$([\,],\ b,\ [c,\ d])\ \&\ ([\,],\ c,\ [\,])\ \&\ ([\,],\ d,\ [\,])\ \&\ \mathit{Empty}$$



となり、a を加えると

$$([d],\ a,\ [b])\ \&\ ([\,],\ b,\ [c,\ d])\ \&\ ([\,],\ c,\ [\,])\ \&\ ([\,],\ d,\ [\,])\ \&\ \mathit{Empty}$$



となり、これで図 2.8 のグラフを表すことができた。このグラフの定義には次のような性質がある。

- この定義で任意の有向グラフを表すことができる [4]。
- グラフを組み上げる順番だけグラフの表現が存在する。つまり、頂点数が n のグラフの表現は $n!$ 通り存在する。

2.5 グラフの一般的定義

一般にはグラフは構成的には定義せず、例えば、頂点集合と枝集合とその間の関係によって定義する。具体的には、

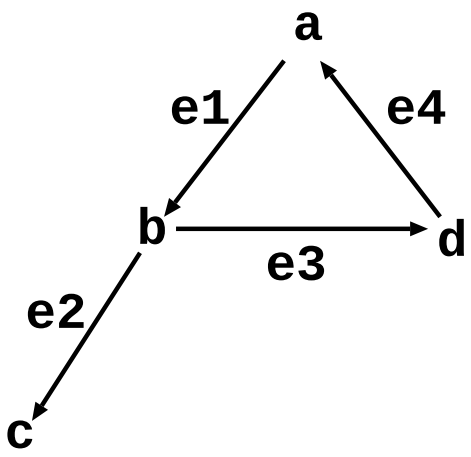


図 2.8: 有向グラフの例 1

- 枝の集合を A
- 頂点の集合を V
- 各枝に対して、その終点を対応させる関数を ∂^+、その始点を対応させる関数を ∂^-

とし、$(V, A, \partial^+, \partial^-)$ の 4 つ組がグラフであると定義する。例えば、図 2.8 のグラフの場合は、

$$\begin{aligned} V &= \{a, b, c, d\} \\ A &= \{e1, e2, e3, e4\} \\ \partial^+ &= \{(e1, b), (e2, c), (e3, d), (e4, a)\} \\ \partial^- &= \{(e1, a), (e2, b), (e3, b), (e4, d)\} \end{aligned}$$

とすればよい。数学的にグラフを取り扱う際にはこのままの表現で問題ないが、計算機でグラフを扱う際には効率をよくするために、行列表現やリスト表現がよく用いられる。

2.5.1 行列によるグラフの表現

行列によるグラフの表現法としては次のようなものがある [11]。

- 接続行列 (incidence matrix) による表現法
  各頂点を行のインデックス、各枝を列のインデックスに対応させて (逆でも良い)、その間の関係を行列の各成分におく。その関係としては 3 種類考え、

その頂点がその枝の始点であって終点でないとき 1、終点であって始点でないとき −1、その他のとき 0 とする (値は何でもよい)。この表現法では、自己閉路である枝が含まれている場合には、その枝がどの頂点に接続しているかということについての情報を入れることができない (関係を 4 種類に増やせば可能)。
(例) 図 2.8 のグラフは接続行列を使うと表 2.1 のように表すことができる。

	e1	e2	e3	e4
a	1	0	0	-1
b	-1	1	1	0
c	0	-1	0	0
d	0	0	-1	1

表 2.1: 図 2.8 のグラフの接続行列による表現

- 隣接行列 (adjacency matrix) による表現法
  各頂点を行のインデックス、列のインデックスに対応させ、u 行 v 列の成分を、u から v への枝があれば 1、そうでなければ 0 とする。この表現法では、同じ始点と同じ終点をもつ並列枝に関する情報を入れることができない。(値として 2 以上を使ってよければ可能。)
  (例) 図 2.8 のグラフは隣接行列を使うと表 2.2 のように表すことができる。

	a	b	c	d
a	0	1	0	0
b	0	0	1	1
c	0	0	0	0
d	1	0	0	0

表 2.2: 図 2.8 のグラフの隣接行列による表現

2.5.2 リストによるグラフの表現

計算機上でグラフの問題を効率的に解くアルゴリズムを設計する際には、行列表現をそのまま 2 次元配列として保持すると、例えばある頂点の successor(その頂点を始点とする枝の終点) の集合を求めたりするときに O(頂点数) の手間がかかるので、枝数が少ない場合には無駄が多くなる。そのような場合にはリスト表現が用いられることもある [11]。そのときに、頂点と枝の関係を表すものと、頂点と頂点の関係を表す隣接リスト (adjacency list) による表現とがある。隣接リストによ

るグラフの実現を Pascal で記述すると、以下のようになる。まず隣接リストは、頂点の型を NODE とすると、

```
type LIST = ^CELL;
     CELL = record
              nodeName: NODE;
              next: LIST
     end;
```

のように表すことができる。これを用いて

```
headers: array[NODE] of LIST
```

のようにグラフの頂点を添字とする隣接リストの配列 headers を作成し、配列 headers の各エントリに、各頂点の successor の集合を表している隣接リストを入れることによりグラフを表現することができる。

2.6 構成的定義 1、構成的定義 2 の比較

構成的定義 2 では任意の有向グラフを表すことができるが、構成的定義 1 では有向無閉路グラフのみしか表すことができない。よって、表現しうるグラフの範囲の広さという点については、構成的定義 2 の方が優れている。しかし、構成的定義 1 はあるカテゴリー (category) について initial algebra[1] になっており [3]、構成的定義 2 は initial algebra にはなっていないので、理論的取り扱いという面においては構成的表現法 1 の方が優れている。次章以降では任意の有向グラフを対象としたグラフの探索を取り扱うので、次章以降では構成的定義 1 は用いず、構成的定義 2 を用いることとする。

2.7 隣接行列による表現と構成的定義 2 との比較

あるグラフを表現している隣接行列の部分行列をとってくると、それはもとのグラフの部分グラフになっている。よって、構成的定義 2 と隣接行列による表現との間には単純な対応関係があり、図 2.8 の例では次のようになる。

- $([\,],\ d,\ [\,])$ & $Empty$ は

	d
d	0

に対応し、

- $([\,],\ c,\ [\,])$ & $([\,],\ d,\ [\,])$ & $Empty$ は

	c	d
c	0	0
d	0	0

に対応し、

- $([\,],\ b,\ [c,\ d]) \ \&\ ([\,],\ c,\ [\,]) \ \&\ ([\,],\ d,\ [\,]) \ \&\ Empty$ は

	b	c	d
b	0	1	1
c	0	0	0
d	0	0	0

に対応し、

- $([d],\ a,\ [b]) \ \&\ ([\,],\ b,\ [c,\ d]) \ \&\ ([\,],\ c,\ [\,]) \ \&\ ([\,],\ d,\ [\,]) \ \&\ Empty$ は

	a	b	c	d
a	0	1	0	0
b	0	0	1	1
c	0	0	0	0
d	1	0	0	0

に対応する。

よって、隣接行列による表現は構成的であると見ることもできる。

第3章　グラフの探索に関するアルゴリズムの記述

グラフのアルゴリズムで最も基本的なのは、すべての頂点や辺に何らかの処理を加えて回ることである。そのためには、それぞれの頂点や辺を 2 回以上調べることなく、しかも必ず 1 回は調べる組織的な手順が必要になる。このように、グラフ全体を組織的に調べることをグラフの探索 (search) という [6]。

3.1　単純な探索 (頂点を順番に調べる)

探索の最も簡単な方法は、頂点を順番に調べることであるが、これは、グラフのすべての枝の向きを逆にしたり、ある頂点がグラフ中に存在しているかどうかを調べたり、辺の重みの総和を求めたりするような、グラフの構造に関係しない単純な渡り歩きで行える演算のみで求まるものにしか適用できない。この単純な探索を行う関数 *ufold* は、次のように定義できる [4]。

$$
\begin{array}{lcl}
ufold & :: & (Context \to \alpha \to \alpha) \to \alpha \to Graph \to \alpha \\
ufold\ f\ u\ Empty & = & u \\
ufold\ f\ u\ (c\ \&\ g) & = & f\ c\ (ufold\ f\ u\ g)
\end{array}
$$

例えば、グラフの枝の向きをすべて逆にするような関数は

$$
\begin{array}{lcl}
grev & :: & Graph \to Graph \\
grev & = & ufold\ (\lambda(p,v,s)r\ .\ (s,v,p)\ \&\ r)\ Empty
\end{array}
$$

のように定義することができ、また、ある頂点 u がグラフ中に含まれているかどうかを調べる関数は

$$
\begin{array}{lcl}
gmember & :: & Vertex \to Graph \to Bool \\
gmember\ u & = & ufold\ (\lambda(p,v,s)r\ .\ u == v\ or\ r)\ False
\end{array}
$$

のように定義することができる。

3.2 アクティブパターンマッチ

前節で述べた単純な探索を行うにはグラフが構成された順番にそってグラフを探索すればよいので自然に構成的に探索関数を記述できるが、枝の接続状態に応じて探索の順序が変化するような探索関数を構成的に記述するには、グラフから任意の頂点を 1 つ取り除くことができるようにする必要がある。このためにアクティブパターンマッチと呼ばれる技法を用いることが提案されている [4, 12]。グラフを引数にとる際に、アクティブパターンマッチによって、ある指定された頂点 v 及びその頂点に付随する枝と、それらを除いたグラフとに分割するということを行う。以下では、$c \ \& \ g$ のように記述されているときは普通のパターンマッチを表し、$(p, v, s) \ \underline{\&} \ g$ のように $\underline{\&}$ を用いて記述されているときはアクティブパターンマッチを表すものとする。

3.3 深さ優先探索

グラフの探索法のうち、最も応用範囲が広いのは深さ優先探索 (depth first search) である。深さ優先探索の具体的な訪問順序は次の通りである。

- 最初に、訪問する頂点を一つ選ぶ。
- ある頂点の次に訪問する頂点は、その頂点の successor のうちでまだ訪問していない頂点のうちの一つとする。(successor が一つもない場合、または訪問していない successor が一つもない場合には、訪問していない successor があるところまで戻ってそのうちの 1 つを訪問する。)

深さ優先探索をしながらある計算を行うことによって、トポロジカルソーティング (topological sorting) をしたり、連結成分 (connected component) に分解したり、強連結成分 (strongly connected component) に分解したり、2 重連結成分 (biconnected component) に分解したりすることができる [13, 2]。

深さ優先探索関数の構成的定義はいくつか提案されているが [2, 4]、以下のように定義することもできる。

$$
\begin{aligned}
&\mathit{depthfold} \ :: \ (\mathit{Context} \to \alpha \to \alpha \to \alpha) \to \alpha \to [\mathit{Vertex}] \to \mathit{Graph} \to \alpha \\
&\mathit{depthfold} \ f \ u \ vs = \mathit{fst} \ \circ \ \mathit{depthfold}' \ f \ u \ vs
\end{aligned}
$$

それぞれの引数の意味は次の通りである。

- 第 1 引数 (f のところ) には、計算をする関数が与えられる。
- 第 2 引数 (u のところ) に、初期値のような役割をする値が与えられる。

- 第 3 引数 (vs のところ) には、頂点のリストが与えられる。探索はこのリストの第 1 要素から始まる。第 1 要素に与えられた頂点から到達可能な頂点をすべて訪問し終わったら、このリストの第 2 要素を見る。もし第 2 要素が訪問されていなければそれを始点として残りのグラフを深さ優先で訪問する。訪問されていれば第 3 要素へうつり、同じようなことを繰り返し、このリストがなくなったら終了する。(この、第 3 引数に与えられるリストのことを深さ優先探索のイニシャルオーダー (initial order) と呼ぶことにする。)
- 第 4 引数 (g のところ) にはグラフが与えられる。

depthfold′ は結果と残ったグラフの対を返す。

$$
\begin{aligned}
&\mathit{depthfold'} \;::\; (\mathit{Context} \to \alpha \to \alpha \to \alpha) \\
&\qquad\qquad \to \alpha \\
&\qquad\qquad \to [\mathit{Vertex}] \\
&\qquad\qquad \to \mathit{Graph} \\
&\qquad\qquad \to (\alpha, \mathit{Graph}) \\
&\mathit{depthfold'}\ f\ u\ [\,]\ g = (u, g) \\
&\mathit{depthfold'}\ f\ u\ (v : vs)\ g = \\
&\quad \text{if } (\mathit{gmember}\ v\ g) = \text{let } ((p,n,s)\ \&\ g') = \mathit{apm}\ v\ g \\
&\qquad\qquad\qquad\qquad\qquad (r1, g1) = \mathit{depthfold'}\ f\ u\ s\ g' \\
&\qquad\qquad\qquad\qquad\qquad (r2, g2) = \mathit{depthfold'}\ f\ u\ vs\ g1 \\
&\qquad\qquad\qquad\qquad \text{in } (f\ (p,n,s)\ r1\ r2,\ g2) \\
&\quad \text{else } \mathit{depthfold'}\ f\ u\ vs\ g
\end{aligned}
$$

ここで、*apm* はアクティブパターンマッチを行った結果をグラフとして返す関数であるとする。例として、グラフの深さ優先森 (depth first forest)[2] を求める関数 *dfs* を定義してみると次のようになる。

$$
\begin{aligned}
&\text{data } \mathit{Tree}\ a = \mathit{Node}\ a\ [\mathit{Tree}\ a] \\
&\mathit{dfs} :: [\mathit{Vertex}] \to \mathit{Graph} \\
&\mathit{dfs} = \mathit{depthfold}\ f\ [\,] \\
&\quad \text{where } f\ (p,n,s)\ r1\ r2 = \mathit{Node}\ n\ r1\ :\ r2
\end{aligned}
$$

3.4 幅優先探索

深さ優先探索は応用範囲の広い探索法であるが、すべての場合に深さ優先探索が適しているというわけではない。深さ優先探索と対照的な探索法として幅優先探索 (breadth first search) がある。幅優先探索では、最初の頂点を訪問した後、この頂点から枝 1 本で到達可能な頂点を順番に訪問する。これが終わったら、これらの頂点のいずれかから枝 1 本で到達可能な頂点を調べる。このとき、一度訪問していれば訪問しない。以下同様に繰り返す。

幅優先探索をする関数 *breadthfold* は次のように定義できる。

$$
\begin{aligned}
&\mathit{breadth} \;::\; (\mathit{Context} \to \alpha \to \alpha \to \alpha) \to \alpha \to [\mathit{Vertex}] \to \mathit{Graph} \to \alpha \\
&\mathit{breadthfold}\ f\ u\ vs = fst\ \circ\ \mathit{breadthfold}'\ f\ u\ vs \\
&\mathit{breadthfold}'\ f\ u\ [\,]\ g = (u, g) \\
&\mathit{breadthfold}'\ f\ u\ (v : vs)\ g = \\
&\quad \text{if } (\mathit{gmember}\ v\ g) = \text{let } ((p, n, s)\ \&\ g') = \mathit{apm}\ v\ g \\
&\qquad\qquad\qquad\qquad (r1, g1) = \mathit{breadthfold}'\ f\ u\ vs\ g' \\
&\qquad\qquad\qquad\qquad (r2, g2) = \mathit{breadthfold}'\ f\ u\ s\ g1 \\
&\qquad\qquad\qquad \text{in } (f\ (p, n, s)\ r1\ r2,\ g2) \\
&\quad \text{else } \mathit{breadthfold}'\ f\ u\ vs\ g
\end{aligned}
$$

例として、グラフの幅優先森を求める関数 *bfs* を定義してみると次のようになる。

$$
\begin{aligned}
&\mathit{dfs} :: [\mathit{Vertex}] \to \mathit{Graph} \\
&\mathit{bfs} = \mathit{breadthfold}\ f\ [\,] \\
&\quad \text{where } f\ (p, n, s)\ r1\ r2 = \mathit{Node}\ n\ r2\ :\ r1
\end{aligned}
$$

3.5 一般的探索

前節までで、単純な探索関数、深さ優先探索関数、幅優先探索関数を個別に定義したが、この節ではこれらをその特別の場合として含む、一般的探索関数を構成的に定義する。

深さ優先探索は、次に訪問し得る頂点 (これを前線と呼ぶ [6]) のうち、出発点からの深さ最大の頂点を訪問する探索法である。逆に、幅優先探索は、次に訪問し得る頂点のうち出発点からの深さ最小の頂点を訪問する探索法である。次に訪問する頂点を選ぶ基準をこれらとは違ったものにすると、別の種類の探索アルゴリズムが得られる。これを一般的な探索アルゴリズムと呼ぶ [6]。一般的な探索では、次に訪問し得る頂点について何らかの評価基準を設け、その値が最大 (または最小) のものを選んで訪問する。例えば、最短路問題の解法の 1 つのダイクストラ法 [14] は出発点からの道の重みの和が最小の頂点を選ぶ。また、最小木問題の解法の 1 つの Prim のアルゴリズムでは、枝の重みが最小のものを選ぶ。深さ優先探索や幅優先探索は、一般的な探索アルゴリズムにおいて評価基準を出発点からの深さにしたものに相当する。

探索の様子を図示すると図 3.1 のようになる。黒く塗りつぶした頂点が訪問済みの頂点を表し、斜線のついた頂点が次に訪問する頂点の候補 (前線) を表している。

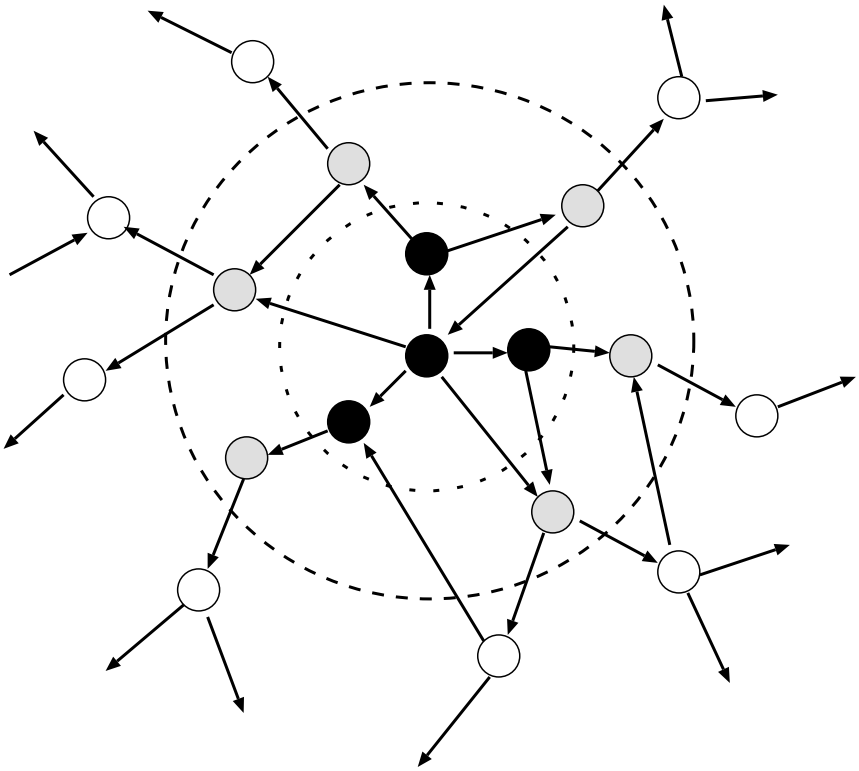


図 3.1: グラフの探索

一般的探索関数 $explore$ を以下のように定義する。

$$
\begin{aligned}
&explore :: ((FrontElemet, \alpha) \to \alpha) \\
&\qquad \to \alpha \\
&\qquad \to ((Vertex, Value, [Vertex]) \to PriorityQueue \\
&\qquad\quad \to PriorityQueue) \\
&\qquad \to (PriorityQueue, Graph) \\
&\qquad \to \alpha \\
&explore\ acc\ r\ \uplus\ (fs, Empty) = r \\
&explore\ acc\ r\ \uplus\ ((pv, v, d) \lhd fs,\ (p, v, s)\ \underline{\&}\ g) \\
&\quad = acc\ ((pv, v, d),\ explore\ acc\ r\ \uplus\ ((v, d, s) \uplus fs,\ g))
\end{aligned}
$$

第2式の $(p, v, s)\ \underline{\&}\ g$ は、引数にとるグラフを、頂点 v(及びそれに付随する枝) とそれを除いたグラフに分割することを表している。このようなパターンマッチをアクティブパターンマッチ [4] と呼ぶ。このように定義することにより、関数が呼ばれるたびにグラフの頂点が1つずつ減少していくが、これは訪問済みの頂点に訪問済みのマークをつけることに対応している。第一引数の acc には $(FrontElement, \alpha) \to \alpha$ の型の関数、第二引数の r にはグラフが $Empty$ の場合の型 α の値を指定する。$FrontElement$ 型は前線の要素の型であり、前線の要素は、(その頂点にくる1つ前の頂点、前線中の頂点自身、評価値) の3つ組で表す。

$$FrontElement = (Vertex, Vertex, Value)$$

前線は $FrontElement$ 型の要素を持つリストで表し、評価値の小さい (または大きい) 順に並べ、前線中の頂点には重複がないようにする。前線に新たに頂点を $\uplus$ によって加える際には、順番を保ち、かつ頂点の重複がないように加える。$(v, d, s)\ \uplus\ fs$ は頂点 v の successor のリスト s のそれぞれの評価値を計算し、前線 fs に加えることを表す。前線を保持する fs は優先順位つき待ち行列 (priority queue) であればよく、その型を $PriorityQueue$ とする。$(pv, v, d) \lhd fs$ は前線を、前線の先頭要素 (pv, v, d) とそれを取り除いた前線 fs に分けることを表す。前線とグラフは対で表現したが、これは、前線とグラフの両方が関数 $explore$ の操作対象になっているということを表している。なお、ここで定義した $explore$ は、前線の初期値リスト中の頂点から到達不可能な頂点がある場合にはマッチする場合がなくなってしまうが、分かり易くするため、また、後で述べる融合変換ができるようにするためにこのような定義を用いる。

例として、グラフの頂点数を求める関数を $explore$ を用いて記述すると次のようになる。

$$
\begin{aligned}
nodeNumber &:: Graph \to Integer \\
nodeNumber\ g &= explore\ (\lambda(x, y)\,.\,1 + y)\ 0\ \uplus\ (fs, g)
\end{aligned}
$$

ここで、$\uplus$、fs は適当に定めればよい。

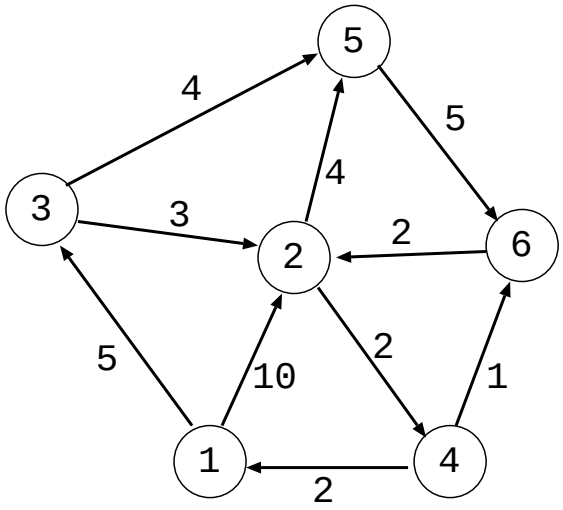


図 3.2: 有向グラフの例

3.5.1 グラフの一般的探索の例

最短路問題

グラフの 2 点間の最短距離を求めるアルゴリズムの 1 つにダイクストラ法 [14] があるが、これを一般的探索関数 $explore$ を用いて記述すると次のようになる。

$$
\begin{aligned}
&dijkstra :: Vertex \to Graph \to [(Vertex, Value)] \\
&dijkstra\ v\ g = explore\ acc\ r\ \uplus\ (fs, g) \\
&\quad \text{where} \\
&\qquad acc\ ((pv, v, d),\ r) = (v, d)\ :\ r \\
&\qquad r = [\,] \\
&\qquad (v, d, s) \uplus fs = [(v, v', d + dis\ v\ v') | v' \leftarrow s] \bowtie fs \\
&\qquad fs = [(v, v', dis\ v\ v') | v' \leftarrow s] \\
&\qquad fs = toPriorityQueue\ [(\_, v, 0)]
\end{aligned}
$$

ここで、$\bowtie$ は、新たな前線の候補リストを現在の前線に頂点に重複がないように加える関数であり、$toPriorityQueue$ はリストを優先順位つき待ち行列にする関数であるとする。また、$dis\ v\ v'$ は、枝 vv' の重みを表す。これによって (頂点, その頂点への最短距離) の対のリストが結果として得られる。例えば、図 3.2 のグラフ (これを ex とする) において頂点 1 から他の頂点への最短距離を求めるには, $dijkstra\ 1\ ex$ を計算すればよく、結果は、

$$[(1, 0), (3, 5), (2, 8), (5, 9), (4, 10), (6, 11)]$$

となる。

最小木問題

無向グラフの最小木 (minimum spanning tree) を求めるアルゴリズムの 1 つに Prim のアルゴリズム [15] がある。これを一般的探索関数 $explore$ を用いて記述すると次のようになる。

$$
\begin{aligned}
&prim :: Graph \to Tree\ \ Vertex \\
&prim\ ((p, v, s)\ \&\ g) = explore\ acc\ r\ \uplus\ (fs, g)\ (Node\ v\ [\,]) \\
&\quad \text{where} \\
&\qquad acc\ ((pv, v, d), r) = r\ \circ\ addv\ (pv, v) \\
&\qquad r = id \\
&\qquad (v, d, s) \uplus fs = [(v, v', dis\ v\ v') | v' \leftarrow s] \bowtie fs \\
&\qquad fs = toPriorityQueue\ [(v, v', dis\ v\ v') | v' \leftarrow s]
\end{aligned}
$$

ここで、$\bowtie$ は、新たな前線の候補リストを現在の前線に頂点に重複がないように加える関数であり、$toPriorityQueue$ はリストを優先順位つき待ち行列にする関数であるとする。また、$addv$ は

$$
\begin{aligned}
&addv :: (Vertex, Vertex) \to Tree\ \ Vertex \to Tree\ \ Vertex \\
&addv\ (pv, v)\ (Node\ v'\ cs) = \text{if}\ (pv == v')\ \text{then}\ (Node\ v'\ ([Node\ v\ [\,]]\ +\!\!+\ cs)) \\
&\qquad\qquad\qquad\qquad\qquad \text{else}\ Node\ v'\ [addv\ (pv, v)\ c | c \leftarrow cs]
\end{aligned}
$$

という関数であり、第 1 引数に与えられる枝 (pv, v) の終点 v を第 2 引数に与えられる木 r 中の頂点 pv の子として付け加えることを意味している。ただし、次章で行う融合変換が行えるようにするために、次に定義する深さ優先探索木を返す関数 dfs の定義中に現れる $addv$ とは少し異なる定義となっている。また、無向グラフは両方向に重みが同じ枝がある有向グラフで表現されているものとする。

深さ優先探索

深さ優先探索は一般的探索において出発点からの深さを前線の評価基準として用いることにより表すことができる。例えば、深さ優先探索木を返す関数 dfs は、

$$
\begin{aligned}
&dfs :: [Vertex] \to Graph \to Tree\ \ Vertex \\
&dfs\ vs\ g = dfs'\ (fs, g)\ (Node\ \_\ [\,]) \\
&\quad \text{where} \\
&\qquad fs = toPriorityQueue\ [(\_, v, 0) | v \leftarrow vs] \\
&\qquad dfs'\ (fs, Empty)\ r = r \\
&\qquad dfs'\ ((pv, v, d) \lhd fs,\ (p, v, s)\ \underline{\&}\ g)\ r = dfs'\ ((v, d, s) \uplus fs, g)\ (addv\ (pv, v)\ r) \\
&\qquad (v, d, s) \uplus fs = [(v, v', d-1) | v' \leftarrow s] \bowtie fs
\end{aligned}
$$

のように表すことができる。この関数 dfs は、仮想的な頂点 $\_$ とその頂点から vs 中の頂点への枝をグラフ g に加え、頂点 $\_$ を出発点として深さ優先探索を行い、仮想

的な頂点 _ を根とする探索木を結果として返すことを表している。$addv$ は

$$
\begin{aligned}
&addv :: (Vertex, Vertex) \to Tree\ Vertex \to Tree\ Vertex \\
&addv\ (pv, v)\ (Node\ v'\ cs) = \text{if } (pv == v')\ \text{then } (Node\ v'\ (cs\ +\!\!+\ [Node\ v\ [\,]])) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{else } Node\ v'\ [addv\ (pv, v)\ c | c \leftarrow cs]
\end{aligned}
$$

という関数であり、第 1 引数に与えられる枝 (pv, v) の終点 v を第 2 引数に与えられる木 r 中の頂点 pv の子として付け加えることを意味している。関数 dfs は 3.5 で定義した関数 $explore$ を用いて記述すると

$$
\begin{aligned}
&dfs\ vs\ g = explore\ acc\ r\ \uplus\ (fs, g)\ (Node\ \_\ [\,]) \\
&\quad \text{where} \\
&\qquad fs = toPriorityQueue\ [(\_, v, 0) | v \leftarrow vs] \\
&\qquad (v, d, s) \uplus fs = [(v, v', d-1) | v' \leftarrow s] \bowtie fs \\
&\qquad r = id \\
&\qquad acc\ ((pv, v, d), r) = r\ \circ\ addv\ (pv, v)
\end{aligned}
$$

となる。

トポロジカルソーティング (深さ優先探索による)

トポロジカルソーティングは深さ優先探索により行うことができるので、一般的探索関数を用いて表現することができる。トポロジカルソーティングとは、有向グラフが与えられたときに、頂点をある順序で 1 列に並べることであり、その列中の右にある頂点から左にある頂点へは与えられた有向グラフ中において枝が存在しないという条件を満たすような列のうちの 1 つを結果として返すというものである。これを行うには、まず、与えられたグラフの深さ優先探索木を求め、次にそれを後順 (postorder) で渡り、それを逆順にすればよい [2]。トポロジカルソーティングを行う関数 $topsort$ は、次のように記述できる。

$$
\begin{aligned}
&topsort :: Graph \to [Vertex] \\
&topsort\ g = (tail\ \circ\ reverse\ \circ\ postorder)\ (dfs\ (nodes\ g)\ g)
\end{aligned}
$$

この関数 $topsort$ はグラフを引数としてとり、そのグラフのトポロジカルソーティングの 1 つをリストとして返す。ここで、$tail$ はリストを引数としてとり、そのリストの先頭要素を除いた残りのリストを返す関数であり、これにより仮想的な頂点 _ が取り除かれる。$nodes$ は、引数にとったグラフ中のすべての頂点をリストにして返す関数である。後順で渡る関数 $postorder$ は、

$$
\begin{aligned}
postorder\ \ &:: \ \ Tree\ \alpha \to [\alpha] \\
postorder\ (Node\ a\ ts)\ \ &= \ \ concat\ (map\ postorder\ ts)\ +\!\!+\ [a]
\end{aligned}
$$

のように定義でき、これは木を引数としてとり、それを後順で渡った結果をリストとして返す。リストを逆順にする関数 $reverse$ の定義は、

$$\begin{array}{lcl} reverse & :: & [\alpha] \to [\alpha] \\ reverse\ [\,] & = & [\,] \\ reverse\ (x : xs) & = & reverse\ xs\ +\!\!+\ [x] \end{array}$$

のように定義できる。この $reverse\ xs$ の計算量は、リスト xs の長さを n とすると $\mathrm{O}(n^2)$ だが、$\mathrm{O}(n)$ の定義も可能である (定義は省略する)。トポロジカルソーティングを行う関数 $topsort$ は、一般的探索関数 $explore$ を用いて記述すると、

$$\begin{array}{l} topsort :: Graph \to [Vertex] \\ topsort\ g = (tail\ \circ\ reverse\ \circ\ postorder) \\ \qquad\qquad (explore\ acc\ r\ \uplus\ (fs, g)\ (Node\ \_\ [\,])) \\ \quad \text{where} \\ \qquad acc\ ((pv, v, d),\ r) = r\ \circ\ addv\ (pv, v) \\ \qquad r = id \\ \qquad (v, d, s) \uplus fs = [(v, v', d-1) | v' \leftarrow s] \bowtie fs \\ \qquad fs = toPriorityQueue\ [(\_, v, 0) | v \leftarrow nodes\ g] \end{array}$$

のようになる。

3.5.2 一般的探索関数の実現

この節では、一般的探索関数 $explore$ の関数型言語 Haskell による効率のよい 1 つの実現法を示す。関数

$$explore\ acc\ r\ \uplus\ (fs, g)$$

は、グラフ g の頂点数を N、枝数を M とし、$acc\,(a, r)$ の平均計算量を $f(N, M)$ とすると、

$$\mathrm{O}(M \log(N) + N f(N, M))$$

のコストで実現可能である。例えば、ダイクストラ法では、$f(N, M) = \mathrm{O}(1)$ であるので $\mathrm{O}(M \log(N) + N)$ となる。これは、手続き型言語においてヒープを用いてダイクストラ法を実現したときの計算量と次数が同じである。

上で述べたコストで実現するためには、まず、グラフの効率的なアクティブパターンマッチが必要である。そのために、グラフを書き換えのできない、隣接リストの配列として表現する。

```
type Graph = Array Vertex [(Vertex, Dist)]
```

配列の値は、各頂点を始点とする枝の終点 (successor) とその枝の重みの対である。アクティブパターンマッチで要求されるのは、ある頂点の successor のリストと、その頂点とそれに付随する枝を除いた結果、残ったグラフである。そのために、各頂点を添字とする書き換え可能な配列を 1 つ用意し、各アクティブパターンマッチで取り除かれた頂点に訪問済みの印をつけるようにする。さらに、state transformer [16] の技法を用いて、この配列を状態として保持し、次々と受け渡しをするようにする。一般的探索を行う関数を explore とし、この定義の主な部分を以下に示す。

```
explore acc r merge (fs, g) =
    runST (mkEmpty (bounds g) >>= \m ->
           exploreaux acc r merge fs (m,g))

exploreaux acc r merge [] (m,g) = return r
exploreaux acc r merge ((pv,v,d):fs) (m,g) =
    include m v >>= \_ ->
    suc v (m,g) >>= \s ->
    exploreaux acc r merge (merge (v,d,s) fs) (m,g) >>= \x ->
    return (acc (pv,v,d) x)
```

ここでは、分かりやすくするために、前線をリストで管理したが、これを例えばヒープにすれば、上で述べたコストになる。

前線をヒープにした場合の一般的探索関数を hexplore とすると、定義の主な部分は以下のようになる。

```
hexplore acc r  merge fs g =
  runST (mkEmpty (bounds g) >>= \m ->
         hexploreaux acc r merge (foldr insert E fs) (m,g))

hexploreaux acc r merge E (m,g) = return r
hexploreaux acc r merge heap (m,g) =
  let ((pv,v,d),heap') = deletemin heap in
    contains m v >>= \visited ->
    if visited then
      hexploreaux acc r merge heap' (m,g)
    else
      include m v >>= \_ ->
      suc v (m,g) >>= \s ->
      hexploreaux acc r merge (merge (v,d,s) heap') (m,g)
                  >>= \x ->
      return (acc (pv,v,d) x)
```

ここで、`E`は要素がない空のヒープを表しており、`deletemin`は、引数としてヒープをとり、(評価値が最小の要素, それを除いた残りのヒープ) の対を結果として返す関数を表す。また、`merge (v,d,s) heap'`の部分では、sの数だけヒープへの要素の挿入を行い、挿入の際には頂点の重複を許すものとする。そのため、ヒープから評価値が最小の要素をとりだすときに、それ以前に同じ頂点がとりだされているかどうかを調べる必要がある。よって、`hexplore`の計算量は、`acc a r`の平均計算量を$f(N, M)$とすると、$\mathrm{O}(M \log(M) + Nf(N, M))$となる。並列枝 (parallel branches) がない場合にはMの最大値はN^2であるので、`hexplore`の計算量は、$\mathrm{O}(M \log(N) + Nf(N, M))$となる。以下では、計算量を求める際には、並列枝はないものとして求めることとする。一例としてダイクストラ法の計算量を求めてみると、$f(N, M) = \mathrm{O}(1)$であるので$\mathrm{O}(M \log(N) + N)$となる。これは、手続き型言語でヒープを用いてダイクストラ法を実現したときの計算量と次数が同じである。

第4章　グラフアルゴリズムの変換

関数型言語の最適化の技法の1つに、融合変換(fusion)[1]と呼ばれるものがある。小さな関数を組み合わせてプログラムを書いて大きなプログラムを作る場合、合成関数が現れるが、合成関数の間では最終結果には現れない中間のデータの受渡しが行われる。これは効率がよくないので、中間データの受渡しをなくしたい。このためには、2つの関数の合成関数を1つの関数にする必要があり、これは融合変換と呼ばれる。融合変換を行うことにより、時間計算量(time complexity)、領域計算量(space complexity)が小さくなる。時間計算量が大幅に小さくなることはあまりないが、領域計算量が大幅に小さくなることは多い。これまでには、グラフの探索関数に関して融合変換を行う際には、それぞれの探索関数について融合規則を見つけ出すということが行われていたが、本論文では、探索関数をHylomorphism[7, 8]と呼ばれる形に記述しなおしてから変換を行う。これにより、Hylo fusion[7, 8]と呼ばれる融合変換や、その他いろいろな変換を行うことができる。この章では、Hylomorphismの定義を述べ、前章で定義したグラフの一般的探索関数をHylomorphismで記述したのち、Hylo fusionと呼ばれる融合変換をいくつかの例について行う。

4.1　Hylomorphismの定義

3つ組からなるHylomorphismsは以下のように定義される[7, 8]。

定義 1 (Hylomorphism)
2つの射(morphism)

$$\phi : G\,A \to A, \psi : B \to F\,B$$

と自然変換(natural transformation)

$$\eta : F \dot{\to} G$$

が与えられたとき、Hylomorphism $[\![\phi, \eta, \psi]\!]_{G,F}$ は次の等式を満たす最小不動点として定義される。

$$f = \phi \circ (\eta \circ F\,f) \circ \psi$$

$[\![\phi, \eta, \psi]\!]_{G,F}$ の添字の関手(functor)G, F は、自明なときは省略する。　□

Hylomorphism の記述力は高く、ほとんどの再帰的な関数は Hylomorphism で記述することができ、また、Hylomorphism に関する変換規則はいくつか知られており、それを利用することにより、さまざまな変換を行うことができる [7, 8]。前章で定義した一般的探索関数 *explore* を Hylomorphism の形で記述すると、次のようになる。

$$
\begin{aligned}
&explore\ acc\ r \uplus = [\![r \triangledown acc,\ id,\ \psi]\!] \\
&\quad \text{where} \\
&\quad\ \psi\ (fs, Empty) = (1,\ ()) \\
&\quad\ \psi\ ((pv, v, d) \triangleleft fs,\ (p, v, s)\ \underline{\&}\ g) = (2,\ ((pv, v, d),\ ((v, d, s) \uplus fs,\ g)))
\end{aligned}
$$

ここで用いた記法の意味については、[7, 8] に述べられている。Hylomorphism は、Hylomorphism の性質により、catamorphism と anamorphism の合成関数に変換することができ、一般的探索関数は、

$$[\![r \triangledown acc,\ id,\ \psi]\!] = (\!| r \triangledown acc |\!) \circ [\!(\psi)\!]$$

のように catamorphism $(\!| r \triangledown acc |\!)$ と anamorphism $[\!(\psi)\!]$ の合成関数で表すことができる。これは、$[\!(\psi)\!]$ によって探索木を表すリストが生成され、そのリストが $(\!| r \triangledown acc |\!)$ によって消費されるということを表している。$(\!| r \triangledown acc |\!)$, $[\!(\psi)\!]$ を具体的に書き下すと、以下のようになる。

$$(\!| r \triangledown acc |\!) = foldr\ acc\ r$$

$$[\!(\psi)\!] = gentree \uplus$$

ここで、$\uplus$ は ψ の定義中に現れていた $\uplus$ であり、$foldr$ は

$$
\begin{aligned}
&foldr && :: && (\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta \\
&foldr\ f\ a\ [\,] && = && a \\
&foldr\ f\ a\ (x : xs) && = && f\ x\ (foldr\ f\ a\ xs)
\end{aligned}
$$

という関数であり、*gentree* は、

$$
\begin{aligned}
&gentree :: ((Vertex, Value, [Vertex]) \to PriorityQueue \\
&\qquad\qquad \to PriorityQueue) \\
&\qquad\qquad \to (PriorityQueue, Graph) \\
&\qquad\qquad \to [FrontElement] \\
&gentree\ \uplus\ (fs, Empty) = [\,] \\
&gentree\ \uplus\ ((pv, v, d) \triangleleft fs,\ (p, v, s)\ \underline{\&}\ g) \\
&\quad = (pv, v, d)\ :\ gentree\ \uplus\ ((v, d, s) \uplus fs,\ g)
\end{aligned}
$$

という関数である。Hylomorphism $[\![r \triangledown acc,\ id,\ \psi]\!]$ はこの 2 つの関数の合成関数が融合されたものになっている。

4.2 一般的探索関数に関する融合変換

この節では、Hylo fusion の融合規則を述べたのち、その適用例をいくつか示す。
Hylo fusion の融合規則は、以下の 2 つがある [7]。

左融合規則

$$f \circ \phi = \phi' \circ G\ f$$

ならば

$$f \circ [\![\phi, \eta, \psi]\!]_{G,F} = [\![\phi', \eta, \psi]\!]_{G,F}$$

右融合規則

$$\psi \circ g = F\ g \circ \psi'$$

ならば

$$[\![\phi, \eta, \psi]\!]_{G,F} \circ g = [\![\phi, \eta, \psi']\!]_{G,F}$$

4.2.1 融合変換の例 1—*eccentricity*

ある頂点 v から他の頂点までの距離のうちで最も大きなものを求める関数 *eccentricity* は以下のように定義できる。

$$\begin{aligned} &eccentricity && :: && Vertex \to Graph \to Dist \\ &eccentricity\ v\ g && = && (maximum \circ (map\ snd))\ (dijkstra\ v\ g) \end{aligned}$$

3.5.1 において一般的探索関数 *explore* を用いて記述されている *dijkstra* を Hylomorphism で記述しなおすと

$$\begin{aligned} &dijkstra :: Vertex \to Graph \to [(Vertex, Value)] \\ &dijkstra\ v\ g = [\![r \triangledown acc, id, \psi]\!]\ (fs, g) \\ &\quad \text{where} \\ &\quad\quad r = [\,] \\ &\quad\quad acc\ ((pv, v, d),\ r) = (v, d)\ :\ r \\ &\quad\quad \psi\ (fs, Empty) = (1, ()) \\ &\quad\quad \psi\ ((pv, v, d) \triangleleft fs,\ (p, v, s)\ \underline{\&}\ g) = (2,\ ((pv, v, d),\ ((v, d, s) \uplus fs,\ g))) \\ &\quad\quad (v, d, s) \uplus fs = [(v, v', d + dis\ v\ v') | v' \leftarrow s] \bowtie fs \\ &\quad\quad fs = toPriorityQueue\ [(\_, v, 0)] \end{aligned}$$

となるので、

$$eccentricity\ v\ g = (maximum \circ (map\ snd) \circ [\![r \triangledown acc, id, \psi]\!])\ (fs, g)$$

となる。左融合規則より、

$$(maximum \circ (map\ snd) \circ (r \triangledown acc))\ x = ((f_1 \triangledown f_2) \circ F\ (maximum \circ (map\ snd)))\ x$$

を満たす f_1, f_2 が存在すれば、融合変換が行える。以下で x について場合分けをすることにより、f_1, f_2 を導出する。

- $x = (1, ())$ の場合

$$\begin{aligned} \text{LHS} &= (maximum\ \circ\ (map\ snd))\ [\,] \\ &= maximum\ [\,] \\ &= -\infty \\ \text{RHS} &= f_1 \end{aligned}$$

  よって、$f_1 = -\infty$

- $x = (2, ((pv, v, d), xs))$ の場合

$$\begin{aligned} \text{LHS} &= (maximum\ \circ\ (map\ snd))\ (acc\ ((pv, v, d), xs)) \\ &= (maximum\ \circ\ (map\ snd))\ ((v, d) : xs) \\ &= maximum\ (d : map\ snd\ xs) \\ &= max\ d\ (maximum\ (map\ snd\ xs)) \\ \text{RHS} &= f_2\ ((pv, v, d), maximum\ (map\ snd\ xs)) \end{aligned}$$

  よって、$f_2\ ((\_, \_, d), r) = max\ d\ r$

以上より、

$$maximum\ \circ\ (map\ snd)\ \circ\ [\![r \triangledown acc, id, \psi]\!] = [\![f_1 \triangledown f_2, id, \psi]\!]$$

となり、関数 $maximum\ \circ\ (map\ snd)$ と関数 $[\![r \triangledown acc, id, \psi]\!]$ の合成が融合された単一の関数

$$[\![f_1 \triangledown f_2, id, \psi]\!]$$

による表現が得られる。

4.2.2 融合変換の例 2—*travel*

Euclid 的巡回セールスマン問題の近似解法の 1 つに、最小木法とよばれるものがある [6]。これは、まず、グラフの最小木を求め、次にその木を前順 (preorder) で渡るという方法である。この関数を *travel* とすると、以下のように定義できる。

$$\begin{aligned} travel\ g &= preorder\ (prim\ g) \end{aligned} \qquad travel :: Graph \to [Vertex]$$

ここで、$preorder$ は

$$\begin{array}{lcl} preorder & :: & Tree\ \alpha \to [\alpha] \\ preorder\ (Node\ a\ ts) & = & [a]\ +\!\!+\ concat\ (map\ preorder\ ts) \end{array}$$

のように定義される関数であり、木を引数としてとり、それを前順で渡った結果をリストとして返す。3.5.1 で定義した関数 $prim$ を Hylomorphism で記述しなおすと、

$$\begin{array}{l} prim :: Graph \to Tree\ \ Vertex \\ prim\ ((p,v,s)\ \&\ g) = [\![r \triangledown acc, id, \psi]\!]\ (fs, g)\ (Node\ v\ [\,]) \\ \quad \text{where} \\ \quad\quad r = id \\ \quad\quad acc\ ((pv,v,d),\ r) = r\ \circ\ addv\ (pv,v) \\ \quad\quad \psi\ (fs, Empty) = (1, ()) \\ \quad\quad \psi\ ((pv,v,d) \triangleleft fs,\ (p,v,s)\ \underline{\&}\ g) = (2,\ ((pv,v,d),\ ((v,d,s) \uplus fs,\ g))) \\ \quad\quad (v,d,s) \uplus fs = [(v,v',d + dis\ v\ v')|v' \leftarrow s] \bowtie fs \\ \quad\quad fs = toPriorityQueue\ [(v,v',dis\ v\ v')|v' \leftarrow s] \end{array}$$

となる。よって、$g = (p,v,s)\ \&\ g'$ とおくと、

$$travel\ ((p,v,s)\ \&\ g') = ((preorder\ \circ)\ \circ\ [\![r \triangledown acc, id, \psi]\!])\ (fs, g')\ (Node\ v\ [\,])$$

となる。左融合規則より、

$$((preorder\ \circ)\ \circ\ (r \triangledown acc))\ x = ((r' \triangledown acc')\ \circ\ F\ (preorder\ \circ))\ x \qquad (4.1)$$

を満たす r', acc' が存在すれば、融合変換が行える。以下で x について場合分けをすることにより、r', acc' を導出する。

- $x = (1, ())$ の場合

$$\begin{array}{rcl} \text{LHS} & = & preorder\ \circ\ id \\ & = & preorder \\ \text{RHS} & = & r' \end{array}$$

  よって、$r' = preorder$

- $x = (2, ((pv,v,d), y))$ の場合

$$\begin{array}{rcl} \text{LHS} & = & preorder\ \circ\ (acc\ ((pv,v,d), y)) \\ & = & preorder\ \circ\ y\ \circ\ addv\ (pv,v) \\ \text{RHS} & = & acc'\ ((pv,v,d),\ preorder\ \circ\ y) \end{array}$$

  よって、$acc'\ ((pv,v,d), r) = r\ \circ\ addv\ (pv,v)$ となる。これは、acc の定義と同じなので、$acc' = acc$ である。

以上より、

$$travel\ ((p,v,s)\ \&\ g) = [\![preorder \triangledown acc, id, \psi]\!]\ (fs, g)\ (Node\ v\ [\,])$$

となり、関数 $(preorder \circ)$ と $[\![r \triangledown acc, id, \psi]\!]$ が融合された単一の関数

$$[\![preorder \triangledown acc, id, \psi]\!]$$

による表現が得られる。これは表面的には融合されているが、効率の改善される融合はなされていない。このような場合は、さらに Accumulation Fusion と呼ばれる融合変換行う [17]。変換過程は省略し、結果のみを以下に示す。

$$\begin{aligned}
&travel\ ((p,v,s)\ \&\ g) = [\![id \triangledown acc', id, \psi]\!]\ (fs, g)\ [v] \\
&\quad \text{where} \\
&\qquad acc'\ ((pv,v,d),r) = r\ \circ\ addv'\ (pv,v) \\
&\qquad addv'\ (pv,v)\ (x:xs) = \text{if } pv == x \text{ then } x:v:xs \\
&\qquad\qquad\qquad\qquad\qquad \text{else } x : (addv'\ (pv,v)\ xs)
\end{aligned}$$

4.2.3 融合変換の例 3—*topsort*

前章で定義した関数 *topsort* は一般的探索関数とある関数との合成関数になっているので、融合変換の対象になっている。前章で記述した関数 *topsort* を Hylomorphism で記述しなおすと、

$$\begin{aligned}
&topsort\ g = (tail\ \circ\ reverse\ \circ\ postorder)\ ([\![r \triangledown acc, id, \psi]\!]\ (fs, g)\ (Node\ _\ [\,])) \\
&\quad \text{where} \\
&\qquad r = id \\
&\qquad acc\ ((pv,v,d),\ r) = r\ \circ\ addv\ (pv,v) \\
&\qquad \psi\ (fs, Empty) = (1, ()) \\
&\qquad \psi\ ((pv,v,d) \lhd fs,\ (p,v,s)\ \underline{\&}\ g) = (2,\ ((pv,v,d),\ ((v,d,s) \uplus fs,\ g))) \\
&\qquad (v,d,s) \uplus fs = [(v,v',d-1) | v' \leftarrow s] \bowtie fs \\
&\qquad fs = toPriorityQueue\ [(_,v,0) | v \leftarrow nodes\ g]
\end{aligned}$$

となる。左融合規則より、

$$\begin{aligned}
&((tail\ \circ\ reverse\ \circ\ postorder\ \circ)\ \circ\ (r \triangledown acc))\ x \\
&= ((r' \triangledown acc') \circ F\ ((tail\ \circ\ reverse\ \circ\ postorder\ \circ) \circ))\ x
\end{aligned}$$

を満たす r', acc' が存在すれば、融合変換が行える。以下で x について場合分けをすることにより、r', acc' を導出する。

- $x = (1, ())$ の場合

$$\begin{aligned} \text{LHS} &= tail \circ reverse \circ postorder \circ id \\ &= tail \circ reverse \circ postorder \\ \text{RHS} &= r' \end{aligned}$$

よって、$r' = tail \circ reverse \circ postorder$

- $x = (2, ((pv, v, d), y))$ の場合

$$\begin{aligned} \text{LHS} &= tail \circ reverse \circ postorder \circ (acc\ ((pv, v, d), y)) \\ &= tail \circ reverse \circ postorder \circ y \circ addv\ (pv, v) \\ \text{RHS} &= acc'\ ((pv, v, d),\ tail \circ reverse \circ postorder \circ y) \end{aligned}$$

よって、$acc'\ ((pv, v, d), r) = r \circ addv\ (pv, v)$ となる。これは、acc の定義と同じなので、$acc' = acc$ である。

以上より、

$$topsort\ g = [\![tail \circ reverse \circ postorder \triangledown acc, id, \psi]\!]\ (fs, g)\ (Node\ \_\ [\,])$$

となり、関数 $(preorder \circ)$ と $[\![r \triangledown acc, id, \psi]\!]$ が融合された単一の関数

$$[\![tail \circ reverse \circ postorder \triangledown acc, id, \psi]\!]$$

による表現が得られる。これは表面的には融合されているが、効率の改善される融合はなされていない。このような場合は、さらに Accumulation Fusion と呼ばれる融合変換行う [17]。変換過程は省略し、結果のみを以下に示す。

$$\begin{aligned} &topsort\ g = [\![tail \triangledown acc', id, \psi]\!]\ (fs, g)\ [\_] \\ &\quad \text{where} \\ &\qquad acc'\ ((pv, v, d), r) = r \circ addv'\ (pv, v) \\ &\qquad addv'\ (pv, v)\ (x : xs) = \text{if } pv == x \text{ then } x : v : xs \\ &\qquad\qquad\qquad\qquad\qquad\qquad \text{else } x : (addv'\ (pv, v)\ xs) \end{aligned}$$

4.2.4 融合変換の例 4—*scc*

グラフを強連結成分 (strongly connected component) に分解する関数 *scc* は、

$$\begin{aligned} scc &:: Graph \to Tree\ Vertex \\ scc\ g &= dfs\ (reverse\ (postOrd\ g))\ (transposeG\ g) \end{aligned}$$

と定義できる [2]。$transposeG$ は

$$\begin{array}{lcl} transposeG & :: & Graph \to Graph \\ transposeG\ Empty & = & Empty \\ transposeG\ ((p,v,s)\ \&\ g) & = & (s,v,p)\ \&\ g \end{array}$$

と定義される関数であり、グラフを引数にとり、そのグラフの枝の向きをすべて逆にしたグラフを結果として返す。$postOrd$ は

$$\begin{array}{lcl} postOrd & :: & Graph \to [Vertex] \\ postOrd\ g & = & init\ (postorder\ (dfs\ (nodes\ g)\ g)) \end{array}$$

と定義される関数であり、グラフを引数にとり、そのグラフの深さ優先探索木を後順 (post order) で渡った結果をリストにして返す。$init$ はリストを引数にとり、そのリストの最後の要素を除いたリストを返す関数であり、これにより仮想的な頂点 _ が取り除かれる。$scc\ g$ の結果として仮想的な頂点 _ を根とする深さ優先探索木が返ってくるが、頂点 _ のそれぞれの子が強連結成分になっている。前節において関数 dfs は Hylomorphism によって記述されており、それを用いるとこの関数 scc は

$$\begin{array}{l} scc\ g = (\llbracket r \triangledown acc, id, \psi \rrbracket\ \circ\ (id \times transposeG))\ (fs, g)\ (Node\ \_\ [\,]) \\ \quad \text{where} \\ \quad\quad fs = toPriorityQueue\ [(\_, v, 0) | v \leftarrow reverse\ (postOrd\ g)] \end{array}$$

となる。ここで、$\times$ は、

$$(f \times g)\ (x,\ y) = (f\ x,\ f\ y)$$

と定義される演算子である。右融合規則より、

$$(\psi\ \circ\ (id \times transposeG))\ x = (F\ (id \times transposeG)\ \circ\ \psi')\ x$$

を満たす ψ' が存在すれば、融合変換が行える。以下で x について場合分けをすることにより、ψ' を導出する。

- $x = (fs, Empty)$ の場合

$$\begin{array}{rcl} \text{LHS} & = & \psi\ (fs, Empty) \\ & = & (1, ()) \\ \text{RHS} & = & F\ (id \times transposeG)\ (\psi'\ (fs, Empty)) \end{array}$$

よって、$\psi'\ (fs, Empty) = (1, ())$

- $x = ((pv, v, d) \triangleleft fs,\ (p, v, s)\ \underline{\&}\ g)$ の場合

$$\begin{aligned}
\text{LHS} &= \psi\ ((pv, v, d) \triangleleft fs,\ (s, v, p)\ \underline{\&}\ (transposeG\ g)) \\
&= (2, ((pv, v, d),\ ((v, d, p) \uplus fs,\ transposeG\ g))) \\
\text{RHS} &= F\ (id \times transposeG)\ (\psi'\ ((pv, v, d) \triangleleft fs,\ (p, v, s)\ \&\ g))
\end{aligned}$$

よって、

$$\psi'\ ((pv, v, d) \triangleleft fs,\ (p, v, s)\ \underline{\&}\ g) = (2, ((pv, v, d),\ ((v, d, p) \uplus fs,\ g)))$$

以上より、

$$\begin{aligned}
&scc\ g = [\![r \triangledown acc, id, \psi']\!]\ (fs, g)\ (Node\ _\ [\,]) \\
&\quad \text{where} \\
&\qquad r = id \\
&\qquad acc\ ((pv, v, d),\ r) = r\ \circ\ addv\ (pv, v) \\
&\qquad \psi'\ (fs, Empty) = (1, ()) \\
&\qquad \psi'\ ((pv, v, d) \triangleleft fs,\ (p, v, s)\ \underline{\&}\ g) = (2,\ ((pv, v, d),\ ((v, d, p) \uplus fs,\ g))) \\
&\qquad (v, d, s) \uplus fs = [(v, v', d - 1) | v' \leftarrow s] \bowtie fs
\end{aligned}$$

となり、関数 $[\![r \triangledown acc, id, \psi]\!]$ と $(id \times transposeG)$ の合成が融合された単一の関数 $[\![acc \triangledown r, id, \psi']\!]$ による表現が得られる。

第5章　グラフアルゴリズムの導出

アルゴリズムの設計は現在人間の思考によって行われているが、これは大変な労力を要することであり、プログラムの正しさの保証も容易にはできない。そこで、まずプログラムを分かりやすく記述し、それに正しさを保ったプログラム変換を施していくという手法によりアルゴリズムの設計を行うことが提案されてきている。この手法を用いると、正しくかつ効率のよいプログラムを得ることができ、さらに、どのようにして効率的なアルゴリズムが設計されたのかが明確に分かる。これまでにリストや木の上のアルゴリズムの設計にプログラム変換を用いるという研究は数多くなされてきているが [1, 18]、グラフ上のアルゴリズムに関してはあまり行われてきていない [3]。この章では、2.4.2 で定義されたグラフ上で最短路問題の単純な仕様を記述し、プログラム変換によって効率的なアルゴリズムの関数プログラミングによる表現を導出し、これがダイクストラ法 [14] と同等であることを示す。なお、この章の内容の主要部分は [19] に書かれている。

5.1　帰納的関数

グラフ上の関数として、構成的ではないが、次のような帰納的な形のものを考える。

$$\begin{aligned} f\ v\ Empty &= \phi_1 \\ f\ v\ ((p,v,s)\ \underline{\&}\ g) &= \phi_2\ (p,v,s)\ [f\ v'\ g | v' \leftarrow s] \end{aligned}$$

ここで、第 1 引数の v は第 2 引数にとるグラフの分割の仕方を指定するためのものである。$\underline{\&}$ はアクティブパターンマッチ [4] を表す。この関数 $f\ v$ は ϕ_1, ϕ_2 によって決まるので、$(\!|\phi_1, \phi_2|\!)_v$ と記述することにする。

グラフ上の多くの関数は、この形に表すことができる。例えば、頂点 v から頂点 t へのすべての (ループのない) 経路をリストにして返す関数 $paths$ は

$$\begin{aligned} &paths && :: && Vertex \to Graph \to Vertex \to [[(Vertex, Vertex)]] \\ &paths\ v\ Empty\ t && = && [\,] \\ &paths\ v\ ((p,v,s)\ \underline{\&}\ g)\ t && = && \text{if } v == t \text{ then } [[\,]] \\ & && && \text{else } concat[((v,v') :) * (paths\ v'\ g\ t) | v' \leftarrow s] \end{aligned}$$

のように表すことができる (経路は枝のリストで表現し、始点が a、終点が b の枝を (a, b) のように表すものとする。また、$*$ は関数をリストのすべての要素に適用する map 関数を表す)。

さらに、この形に表された関数と別の関数との合成関数は融合変換 (fusion) により融合することができる。関数 $(\![\phi_1, \phi_2]\!)_v$ とある関数 k との合成関数

$$k \circ (\![\phi_1, \phi_2]\!)_v$$

を融合することを考えると、以下の融合規則が成り立つことが分かる。

融合規則

$$k\ \phi_1 \quad = \quad \psi_1 \tag{5.1}$$

$$k\ (\phi_2\ (p, v, s)\ zs) \quad = \quad \psi_2\ (p, v, s)\ (k * zs) \tag{5.2}$$

ならば

$$k \circ (\![\phi_1, \phi_2]\!)_v = (\![\psi_1, \psi_2]\!)_v$$

この融合規則を用いて変換をする例を次の節でとりあげる。

5.2 最短路問題

この節では例として、有向グラフの、2 点間の最短距離を求める問題を考える。枝の長さはすべて正とする。この問題を解く効率のよいアルゴリズムとしてダイクストラ法 [14] が知られている。しかし、ダイクストラ法がどのように導き出されたのかは明らかではない。そこで、単純に記述されたプログラムからダイクストラ法と同等なアルゴリズムが導出できることを以下で示す。

5.2.1 問題の単純な記述

まず、最短経路問題を単純に記述する。グラフ g の、頂点 v から頂点 t への最短距離を求めるには、頂点 v から頂点 t への経路をすべて求め、次にそれぞれの経路の長さを求め、最後にそのうちのもっとも小さな値をとればよい。経路の長さを求める関数を $distance$、リストの要素の最小値をとる関数を min とすると、

$$\begin{aligned} sp &:: Vertex \to Graph \to Vertex \to Dist \\ sp\ v\ g\ t &= min\ (distance\ *\ (paths\ v\ g\ t)) \end{aligned}$$

が頂点 v から頂点 t への最短距離となる。($paths$ はループのない経路のみを出力するが、ループのある経路は最短ではないので、問題はない。) $distance$ の定義は、次のようにする。

$$\begin{aligned} distance &:: [(Vertex, Vertex)] \to Dist \\ distance\ [\,] &= 0 \\ distance\ ((a, b) : x) &= dist(a, b) + distance\ x \end{aligned}$$

ただし、$dist(a,b)$ は枝 (a,b) の長さを表す。

5.2.2 融合変換による中間データの受渡しの除去

次に、$min \circ (distance\ *)$ と $paths$ を融合し、中間データの受渡しをなくす。前節で述べた融合規則と対応させるために $paths$ の第3引数を λ-抽象 (λ-abstraction) すると、

$$\begin{aligned} paths\ v\ Empty &= \lambda t.\ [\,] \\ paths\ v\ ((p,v,s)\ \underline{\&}\ g) &= \lambda t.\ (\text{if } v == t \text{ then } [[\,]] \\ &\qquad \text{else } concat\ (h_1\ t\ *\ (zip\ [(v,v')|v' \leftarrow s] \\ &\qquad\qquad [paths\ v'\ g\ |v' \leftarrow s]))) \end{aligned}$$

のようになる。ただし、

$$h_1\ t = \lambda(x,y).\ (x :) * (y\ t)$$

とした。よって、融合規則との対応は、

$$\begin{aligned} k &= min \circ (distance\ *)\ \circ \\ \phi_1 &= \lambda t.\ [\,] \\ \phi_2\ (p,v,s)\ zs &= \lambda t.\ (\text{if } v == t \text{ then } [[\,]] \\ &\qquad \text{else } concat\ (h_1\ t\ *\ (zip\ [(v,v')|v' \leftarrow s]\ zs))) \end{aligned}$$

のようになる。以下で、(5.1), (5.2) を満たす ψ_1, ψ_2 を導きだす。まず、ψ_1 を導出する。

$$\begin{aligned} & k\ \phi_1 \\ = \quad & \{k, \phi_1\} \\ & min \circ (distance\ *) \circ \lambda t.\ [\,] \\ = \quad & \{(min \circ (distance\ *)) \text{ を中に入れて}\} \\ & \lambda t.\ ((min \circ (distance\ *))\ [\,]) \\ = \quad & \{\circ, *\} \\ & \lambda t.\ (min\ [\,]) \\ = \quad & \{min\ [\,] = \infty \text{ とすると}\} \\ & \lambda t.\ \infty \end{aligned}$$

となる。よって、

$$\psi_1 = \lambda t.\ \infty$$

となる。次に、ψ_2 を導出する。

$$
\begin{aligned}
& k\ (\phi_2\ (p, v, s)\ zs) \\
= \quad & \{k, \phi_2\} \\
& min \circ (distance\ *) \circ \lambda t.\ (\text{if } v == t \text{ then } [[\,]] \\
& \qquad \text{else } concat\ (h_1\ t\ *\ (zip\ [(v, v') | v' \leftarrow s]\ zs))) \\
= \quad & \{(min \circ (distance\ *)) \text{ を中に入れて}\} \\
& \lambda t.\ (\text{if } v == t \text{ then } (min \circ (distance\ *))\ [[\,]] \\
& \quad \text{else } (min \circ (distance\ *))\ (concat\ (h_1\ t * (zip\ [(v, v') | v' \leftarrow s]\ zs)))) \\
= \quad & \{\circ\} \\
& \lambda t.\ (\text{if } v == t \text{ then } min\ (distance\ *\ [[\,]]) \\
& \qquad \text{else } min\ (distance\ *\ (concat\ (h_1\ t\ *\ (zip\ [(v, v') | v' \leftarrow s]\ zs))))) \\
= \quad & \{(f*) \circ concat = concat \circ ((f*)*) \text{ より}\} \\
& \lambda t.\ (\text{if } v == t \text{ then } min\ (distance\ *\ [[\,]]) \\
& \quad \text{else } min\ (concat\ ((distance\ *) * (h_1\ t * (zip\ [(v, v') | v' \leftarrow s]\ zs))))) \\
= \quad & \{*\} \\
& \lambda t.\ (\text{if } v == t \text{ then } min\ [distance\ [\,]] \\
& \quad \text{else } min\ (concat\ (((distance\ *) \circ h_1\ t) * (zip\ [(v, v') | v' \leftarrow s]\ zs)))) \\
= \quad & \{distance\} \\
& \lambda t.\ (\text{if } v == t \text{ then } min\ [0] \\
& \quad \text{else } min\ (concat\ (((distance\ *) \circ h_1\ t) * (zip\ [(v, v') | v' \leftarrow s]\ zs)))) \\
= \quad & \{min[0] = 0,\ h_1\ t = \lambda(x, y).(x :) * (y\ t)\} \\
& \lambda t.\ (\text{if } v == t \text{ then } 0 \\
& \qquad \text{else } min\ (concat\ (((distance\ *) \circ\ \lambda(x, y).(x :) * (y\ t)) * \\
& \qquad\qquad (zip\ [(v, v') | v' \leftarrow s]\ zs)))) \\
= \quad & \{(distance\ *) \text{ を中に入れて}\} \\
& \lambda t.\ (\text{if } v == t \text{ then } 0 \\
& \qquad \text{else } min\ (concat\ ((\lambda(x, y).distance * ((x :) * y\ t)) * \\
& \qquad\qquad (zip\ [(v, v') | v' \leftarrow s]\ zs))))
\end{aligned}
$$

$$
\begin{aligned}
= \quad & \{distance * ((x :) * y\ t) = (dist\ x\ +) * (distance * y\ t)\} \\
& \lambda t.\ (\text{if } v == t \text{ then } 0 \\
& \quad \text{else } min\ (concat\ ((\lambda(x,y).(dist\ x\ +) * (distance * y\ t)) * \\
& \qquad (zip\ [(v,v')|v' \leftarrow s]\ zs)))) \\
= \quad & \{min \circ concat = min \circ (min*)\} \\
& \lambda t.\ (\text{if } v == t \text{ then } 0 \\
& \quad \text{else } min\ (min * ((\lambda(x,y).(dist\ x\ +) * (distance * y\ t)) * \\
& \qquad (zip\ [(v,v')|v' \leftarrow s]\ zs)))) \\
= \quad & \{*\} \\
& \lambda t.\ (\text{if } v == t \text{ then } 0 \\
& \quad \text{else } min\ ((min\ \circ\ (\lambda(x,y).(dist\ x\ +) * (distance * y\ t))) * \\
& \qquad zip\ [(v,v')|v' \leftarrow s]\ zs)) \\
= \quad & \{min \text{ を中に入れて }\} \\
& \lambda t.\ (\text{if } v == t \text{ then } 0 \\
& \quad \text{else } min\ ((\lambda(x,y).min((dist\ x\ +) * (distance * y\ t))) * \\
& \qquad zip\ [(v,v')|v' \leftarrow s]\ zs)) \\
= \quad & \{min((dist\ x\ +) * (distance * y\ t)) = dist\ x\ +\ min\ (distance * y\ t)\} \\
& \lambda t.\ (\text{if } v == t \text{ then } 0 \\
& \quad \text{else } min\ ((\lambda(x,y).dist\ x + min(distance * y\ t)) * \\
& \qquad zip\ [(v,v')|v' \leftarrow s]\ zs)) \\
= \quad & \{min, dist, distance, t \text{ を中に入れて }\} \\
& \lambda t.\ (\text{if } v == t \text{ then } 0 \\
& \quad \text{else } min\ ((\lambda(x,y).x + y) * \\
& \qquad zip\ [dist(v,v')|v' \leftarrow s]\ [min(distance * (z\ t))|z \leftarrow zs])) \\
= \quad & \{t \text{ を外にだして }\} \\
& \lambda t.\ (\text{if } v == t \text{ then } 0 \\
& \quad \text{else } min\ ((\lambda(x,y).x + y\ t) * \\
& \qquad zip\ [dist(v,v')|v' \leftarrow s]\ [min \circ (distance*) \circ z|z \leftarrow zs])) \\
= \quad & \{min \circ (distance*) \circ \text{ を外にだして }\} \\
& \lambda t.\ (\text{if } v == t \text{ then } 0
\end{aligned}
$$

$$
\begin{aligned}
&\text{else } min\ ((\lambda(x,y).x+y\ t)* \\
&\qquad zip\ [dist(v,v')|v' \leftarrow s]\ (min \circ (distance*)\circ) * zs)) \\
= &\quad \{k = min \circ (distance*)\circ\} \\
&\lambda t.\ (\text{if } v == t \text{ then } 0 \\
&\quad \text{else } min\ ((\lambda(x,y).x+y\ t)* \\
&\qquad zip\ [dist(v,v')|v' \leftarrow s]\ k * zs))
\end{aligned}
$$

となる。よって、

$$
\begin{aligned}
\psi_2\ (p,v,s)\ xs\ &=\ \lambda t.\ (\text{if } v == t \text{ then } 0 \\
&\qquad \text{else } min((\lambda(x,y).\ x+y\ t) *\ zip\ [dist(v,v')|v' \leftarrow s]\ xs))
\end{aligned}
$$

となる。以上より、$sp\ v = (\![\psi_1, \psi_2]\!)_v$ なので

$$
\begin{aligned}
sp\ v\ Empty\ t\ &=\ \infty \\
sp\ v\ ((p,v,s)\ \underline{\&}\ g)\ t\ &=\ \text{if } v == t \text{ then } 0 \\
&\qquad \text{else } min[dist(v,v') + sp\ v'\ g\ t|v' \leftarrow s]
\end{aligned}
$$

となり、これで中間データの受渡しをなくすことができた。

5.2.3 無駄な呼び出しの回避

前節で、いくつかの再帰関数の合成関数が一つの再帰関数に融合され、中間データの受渡しが取り除かれた。ここからさらに効率を上げるために、無駄な呼び出しをしないようにすることを考える。一般に、一つの再帰的な関数の無駄な呼び出しは次の 2 つに分けられる。

(1) 最終的な結果を得るのに不必要な呼び出しをする。

(2) 同じ呼び出しを 2 回以上する。

まず、(1) については、呼び出す必要のあるものだけ呼び出すようにすると、不必要な呼びだしを省くことができる。これは遅延評価と呼ばれるが、前節でえられたままの形では有効に遅延評価を行えるような形になっていない。そこで、improving value[20] と呼ばれる技法を用いることにする。improving value というのは、結果として得られる値が必ずある値以上であるというその値を徐々に増加させていくというものである。ここでは improving value を要素が単調増加の順にならんでいるリストで表現することにし、リストの最後の要素が求める値となるようにする。結果をこのようなリストにして出力する関数を sp_i とすると、sp の定義は

$$sp\ v\ g\ t = last\ (sp\_i\ v\ g\ t)$$

のようになる。sp_i の定義は

$$
\begin{array}{lcl}
sp\_i & :: & Vertex \to Graph \to Vertex \to [Dist] \\
sp\_i\ v\ Empty\ t & = & [\infty] \\
sp\_i\ v\ ((p,v,s)\ \&\ g)\ t & = & \text{if } v == t \text{ then } [0] \\
 & & \text{else } fold\ smaller\_i\ [\infty] \\
 & & \qquad [dist(v,v') : (dist(v,v')+) * sp\_i\ v'\ g\ t | v' \leftarrow s]
\end{array}
$$

のようにできる。(*fold* はたたみ込み演算子である。) ここで、$smaller\_i$ は 2 つの improving value の値の小さい方を improving value として返す関数で、実際には、2 つのリストの要素を重複要素は 1 つにしながら 1 つのリストに小さい順にならべればよい。よって、$smaller\_i$ の定義は以下のようになる。

$$
\begin{array}{lcl}
smaller\_i & :: & [Dist] \to [Dist] \to [Dist] \\
smaller\_i\ [\,]\ y & = & [\,] \\
smaller\_i\ x\ [\,] & = & [\,] \\
smaller\_i\ (a:x)\ (b:y) & = & \text{if } a < b \text{ then } a : smaller\_i\ x\ (b:y) \\
 & & \text{else if } b < a \text{ then } b : smaller\_i\ (a:x)\ y \\
 & & \text{else } a : smaller\_i\ x\ y
\end{array}
$$

この新たに定義された sp を遅延評価することにより、不必要な呼び出しをすることなく結果を得ることができる。

以上のように improving value を用いて遅延評価を行っても、効率を悪くする要因 (2) の、同じ頂点を 2 回展開する可能性は残っているので、それを省くことを考える。全く同じ展開を省くという技法はメモ化 (memoisation)[21, 22] と呼ばれる。この場合には全く同じ展開は現れないが、前に 1 度展開した頂点を展開しようとする場合を考えてみる。すると、2 回目以降に展開しようとした場合には、improving value の性質と遅延評価により、1 回目に展開したものより短い経路である可能性はないので、1 回目に展開したものはその頂点までの最短経路で展開してきたものになっている。そして、最短経路の部分経路は最短経路であることを考慮すると、2 回目以降の展開は削除してよいことが分かる。2 回目以降の展開を削除するために、展開した頂点を保持しておくことにする。sp_i を、展開した頂点を保持することによって効率化した関数を sp_im とすると、sp の定義は次のようになる。

$$sp\ v\ g\ t = last\ (sp\_im\ v\ g\ t)$$

```
sp v g t = last (sp_im v g t)
sp_im v Empty t = [∞]
sp_im v ((p, v, s) & g) t =
    if v == t then [0]
    else if visited(v) then [∞]
    else {mark(v);
          fold smaller_i [∞] [dist(v, v') :
              (dist(v, v')+) * sp_im v' g t|v' ← s]}
smaller_i [ ] y = [ ]
smaller_i x [ ] = [ ]
smaller_i (a : x) (b : y) =
    if a < b then a : smaller_i x (b : y)
    else if b < a then b : smaller_i (a : x) y
    else a : smaller_i x y
```

図 5.1: 最終的に得られたプログラム

sp_im の定義は以下のようにすればよい。

$$
\begin{array}{lcl}
sp\_im & :: & Vertex \to Graph \to Vertex \to [Dist] \\
sp\_im\ v\ Empty\ t & = & [\infty] \\
sp\_im\ v\ ((p, v, s)\ \&\ g)\ t & = & \text{if } v == t \text{ then } [0] \\
 & & \text{else if } visited(v) \text{ then } [\infty] \\
 & & \text{else } \{mark(v); \\
 & & \quad fold\ smaller\_i\ [\infty]\ [dist(v, v') : \\
 & & \quad\ (dist(v, v')+) * sp\_im\ v'\ g\ t|v' \leftarrow s]\}
\end{array}
$$

ここで、$visited(v)$ は頂点 v を展開したかどうかを調べる関数であり、$mark(v)$ は頂点 v を展開した頂点として保持するという意味である。

5.2.4 最終結果とダイクストラ法との関係

最終的に得られたプログラムは図 5.1 のようになる。

例として図 3.2 のグラフを g とし、頂点 A から頂点 F への最短経路の長さを求めてみる。これは

$$sp\ A\ g\ F = last\ (sp\_im\ A\ g\ F)$$

により求められる。$sp\_im\ A\ g\ F$ を展開していくと

$$[5, 8, 9, 10, 11, 11]$$

のようになるので、11 が求める値となる。

得られたアルゴリズムは距離の小さなものから順に sp_im を展開していくようになっており、頂点数を N とすると、展開は最大で N 回行われる。1 回の展開で行われる計算は主に数の大小比較と $*$ であり、あとは、O(1) でできるグラフの分割 &[4]、$mark$、$visited$ である。手続き型言語でヒープを用いてダイクストラ法を実現した場合、計算量は、頂点数を N、頂点数と枝数のうち大きい方を M とすると $\mathrm{O}(M\log(N))$ である。ここで最終的に得られたプログラムの計算量は $\mathrm{O}(M\log(N))$ よりも大きいが、手順的にはダイクストラ法 [14] と同じものが得られている。また、得られたアルゴリズムは 2 点間の最短距離を求めるものだが、展開するときにはその展開した頂点への最短距離も求まっている。図 3.2 の例では、$sp\_im\ A\ g\ F\ [\,]$ の結果のリストの最後の値を除いたリスト $[5, 8, 9, 10, 11]$ のそれぞれの値が、C,B,E,D, F への最短距離になっている。よって、展開した頂点を保持する際に値も保持するようにすれば、展開した頂点への最短距離も求まることになる。よって、この点においても得られたアルゴリズムはダイクストラ法と同等である。

ダイクストラ法はどのように考え出されたのか不明であったが、単純なプログラムからプログラム変換、improving value、遅延評価、メモ化を用いて導出することにより、ダイクストラ法の設計法の一つが示された。

第6章　結論

本論文では、グラフの探索一般を記述し得る一般的探索関数を構成的に定義し、それを Hylomorphism という形に変換することにより、Hylo fusion という融合変換を行うことができることを示した。これまでの研究では、融合変換を行うための融合規則をそれぞれの探索関数について見つけ出すということが行われていたが、本研究では、まず、一般的探索関数でグラフの探索に関するアルゴリズムを記述し、それを Hylomorphism に変換し、Hylomorphism についてすでに知られている融合規則を適用するということを行った。また、一般的探索関数の関数型言語 Haskell による 1 つの効率のよい実現法を示した。さらに、最短路問題を解く効率的なアルゴリズムを単純な仕様から導出し、グラフアルゴリズムに関してもアルゴリズムの導出が可能であることを示した。以下に今後の課題について述べる。

- 本論文で一般的探索関数に関して行った融合変換は前線の部分にはかかわらない単純な変換であり、頂点の訪問の順序は融合変換によっては変換しない。前線の部分にもかかわってくるような変換を行うことができれば、融合変換の効果、適用範囲が大きくなる可能性がある。
- 第 5 章で行ったダイクストラ法の導出と一般的探索関数によるダイクストラ法の記述との関係を明らかにする。
- 本論文で扱った以外のグラフアルゴリズム、例えば最大流問題 (network flow problem, maximum flow problem, ネットワークフロー問題)、グラフの平面性の判定、グラフの同形性の判定、グラフのマッチングなどの問題についてアルゴリズムの導出、融合変換などが行えるかどうかを検討する。
- 仕様からのアルゴリズムの導出ということに関しては、現在のところ、最短路問題のみしかできていないが、他の問題についても考察する。
- グラフの構成的定義の方法は本論文で述べたもの以外でもあると思われるので、それについて考察する。

謝辞

本研究は様々な人の助力の上に成立しました。
指導教官の武市正人教授は, 研究の指針を示して下さり、数々の有効な助言をして下さいました。
胡振江講師は, 何度も議論をして下さり、さまざまな質問にも答えて下さり、また、数々の有効な助言をして下さいました。
岩崎英哉助教授, 教育用計算機センターの田中哲朗助教授、尾上能之助手、武市研究室の大学院生の皆さんにもいろいろとお世話になりました。
以上の方々に心から感謝します。

参考文献

[1] Richard Bird and Oege de Moor. *Algebra of Programming.* Prentice Hall, 1996.

[2] David J. King and John Launchbury. Structuring depth-first search algorithms in haskell. *Conference record of POPL '95: the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 344–354, 1995.

[3] Jeremy Gibbons. An initial-algebra approach to directed acyclic graphs. *LNCS 947: Mathematics of Program Construction*, pages 122–138, 1995.

[4] Martin Erwig. Functional programming with graphs. *2nd ACM SIGPLAN International Conference on Functional Programming(ICFP '97)*, pages 52–65, 1997.

[5] John Launchbury. Graph algorithms with a functional flavour. *Fitst International Spring School on Acvanced Functional Programming, LNCS 925*, pages 308–331, 1995.

[6] 石畑 清. **岩波講座ソフトウェア科学** *3* **アルゴリズムとデータ構造**. 岩波書店, 1989.

[7] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Deriving structural hylomorphisms from recursive definitions. *ACM SIGPLAN International Conference on Functional Programming(ICFP'96)*, pages 73–82, 1996.

[8] A. Takano and E. Meijer. Shortcut deforestation in calculational form. *Conference on Functional Programming Languages and Computer Architecture*, pages 306–313, 1995.

[9] 武市 正人. **岩波講座ソフトウェア科学** *4* **プログラミング言語**. 岩波書店, 1994.

[10] Richard Bird and Philip Wadler. **関数プログラミング**, *Functional Programming.* 近代科学社, 1991. 武市 正人 訳.

[11] 高橋 磐郎, 藤重 悟. **岩波講座情報科学** *17* **離散数学**. 岩波書店, 1981.

[12] Martin Erwig. Active patterns. *8th Int. Workshop on Implementation of Functional Languages, LNCS 1268*, pages 21–40, 1996.

[13] John Hopcroft and Robert Tarjan. Efficient algorithms for graph manipulation. *Communications of the Association for Computing Machinery*, 16(6):372–378, 1973.

[14] E.W.Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[15] R.C.Prim. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36:1389–1401, 1957.

[16] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. *Conference recored of the twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 71–84, 1993.

[17] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Calculating accumulations. *New Generation Computing*, 17(2), 1999. to appear.

[18] A. Pettorossi and M. Proietti. Rules and strategies for program transformation. In *IFIP TC2/WG2.1 State-of-the-Art Report*, pages 263–303. LNCS 755, 1993.

[19] 篠埜 功, 胡 振江, 武市 正人. 構成的手法によるグラフアルゴリズムの導出. **日本ソフトウェア科学会第 *15* 回大会論文集**, pages 269–272, 1998.

[20] F.Warren Burton. Encapsulating non-determinacy in an abstract data type with deterninate semantics. *Journal of Functional Programming*, 1(1):3–20, 1991.

[21] D.Michie. Memo functions and machine learning. *Nature*, 218(5136):19–22, April 1968.

[22] Nobuo Yamashita. Built-in memoisation mechanism for functional programs. Master's thesis, University of Tokyo, 1995.