構成的手法によるグラフアルゴリズムの導出

Constructive Approach to Deriving Graph Algorithms

篠埜 功　胡 振江　武市 正人
Isao SASANO　Zhenjiang HU　Masato TAKEICHI

東京大学大学院工学系研究科情報工学専攻
Department of Information Engineering, University of Tokyo
({sasano,hu,takeichi}@ipl.t.u-tokyo.ac.jp)

概 要

効率のよいアルゴリズムの設計を人間の思考で行うことは大変な労力を要し、その正しさの証明は別途行う必要がある。それに対し、単純なプログラムに正しさを保ったプログラム変換を施すことにより、効率が良くかつ正しいプログラムを得るという手法がある。リスト、木を扱う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、グラフアルゴリズムの設計に関してはあまり行われていない。本論文では、構成的に表現されたグラフ上でのプログラムの変換規則を提示し、それを用いて最短経路問題を解く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単純な仕様から導出する。

1 導　入

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]、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 [2]。その理由としてはグラフはリストや木のように自由で再帰的な構造 (free recursive data structure) をしていないために変換を行いにくいということが挙げられる。しかし、グラフは非常に多くの分野に現れるものであり、効率のよいグラフアルゴリズムを設計することは重要なことである。

本論文では、構成的に定義されたグラフ上でのプログラムの変換規則を提示し、それを用いて最短経路問題を解く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単純な仕様から導出し、これがダイクストラ法 [3]と同等であることを示す。本論文の記法は関数型言語 Haskell に近いものである。

2 グラフの表現法

グラフの表現法としては、行列表現や隣接リストによる表現などが一般的だが、次のように構成的に表現することもできる [4]。

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

グラフは、空のグラフ $Empty$ であるか、またはグラフに頂点 1 つ (とその頂点に接続している枝) を構成子 & によって付け加えたものである。 (v は頂点名を表し、 p は頂点 v を終点とする枝の始点のリスト、 s は頂点 v を始点とする枝の終点のリストを表

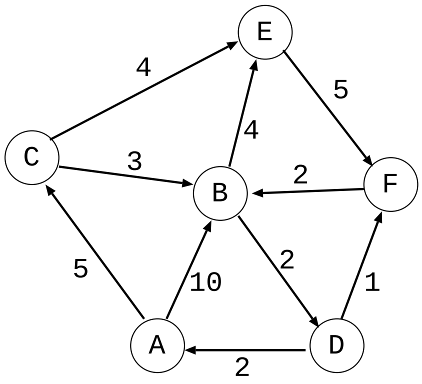

図 1　有向グラフの例

す。) 例えば、図 1のグラフは

$([D], A, [B, C])$ & $([C, F], B, [D, E])$ &
$([\,], C, [E])$ & $([\,], D, [F])$ &
$([\,], E, [F])$ & $([\,], F, [\,])$ & $Empty$

のように表現できる。ただし & は右結合的であるとし、枝の長さは枝を引数にとる関数 $dist$ で表すことにする。グラフをこのように表現することにより、グラフ上のアルゴリズムを帰納的に表現できるようになる。

3　帰納的な関数

グラフ上の関数として次のような帰納的な形のものを考える。

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

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り、第 1 引数の頂点 v が先頭にきているように第 2 引数のグラフを分割している。これはアクティブパターンマッチ [4]と呼ばれ、これを $\underline{\&}$ で表す。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

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

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と呼ばれる変換規則により融合することができる。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。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

$$k\ \phi_1 = \psi_1 \quad (1)$$

$$k\ (\phi_2\ (p, v, s)\ zs) = \psi_2\ (p, v, s)\ (k * zs) \quad (2)$$

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。

4　最短経路問題

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 [3]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。

4.1　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、

$$sp\ v\ g\ t = min\ (distance\ *\ (paths\ v\ g\ t))$$

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。

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

ただし、 $dist(a,b)$ は枝 (a,b) の長さを表す。

4.2 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。3 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- abstraction すると、

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

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、

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

のようになる。以下で、 (1), (2) を満たす ψ_1, ψ_2 を導きだす。まず、 ψ_1 を導出すると、

$k\ \phi_1 = \lambda t.\ \infty$

となる。よって、

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

となる。次に、 ψ_2 を導出すると、

$k\ (\phi_2\ (p,v,s)\ zs) =$
$\quad \lambda t.\ (\text{if } v == t \text{ then } 0$
$\quad\quad \text{else } min\ ((\lambda(x,y).\ x + y\ t)*$
$\quad\quad\quad zip\ [dist(v,v')|v' \leftarrow s]\ k * zs))$

となる。よって、

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

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

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

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。

4.3 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 [5]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

$sp\_i\ v\ Empty\ t = [\infty]$
$sp\_i\ v\ ((p,v,s)\underline{\&}g)\ t = \text{if } v == t \text{ then } [0]$
$\quad \text{else } fold\ smaller\_i\ [\infty]$
$\quad\quad [dist(v,v') : (dist(v,v')+) * sp\_i\ v'\ g\ t|v' \leftarrow s]$

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。

$smaller\_i\ [\,]\ y = [\,]$
$smaller\_i\ x\ [\,] = [\,]$
$smaller\_i\ (a:x)\ (b:y) =$
$\quad \text{if } a < b \text{ then } a : smaller\_i\ x\ (b:y)$
$\quad \text{else if } b < a \text{ then } b : smaller\_i\ (a:x)\ y$
$\quad \text{else } a : smaller\_i\ x\ y$

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アイゼイション [6]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_im の定義は以下のようにすればよい。

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

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。

4.4 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 2のようになる。

例として図 1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 $\underline{\&}$ [4]、 $mark$、 $visited$ である。これはダイクストラ法 [3]と同等である。

$$
\begin{array}{l}
sp\ v\ g\ t = last\ (sp\_im\ v\ g\ t\ [\,]) \\
sp\_im\ v\ Empty\ t = [\infty] \\
sp\_im\ v\ ((p,v,s)\underline{\&}g)\ t = \\
\quad \text{if } v == t \text{ then } [0] \\
\quad \text{else if } visited(v) \text{ then } [\infty] \\
\quad \text{else } \{mark(v); \\
\qquad\quad fold\ smaller\_i\ [\infty]\ [dist(v,v') : \\
\qquad\qquad (dist(v,v')+) * sp\_im\ v'\ g\ t | v' \leftarrow s]\} \\
smaller\_i\ [\,]\ y = [\,] \\
smaller\_i\ x\ [\,] = [\,] \\
smaller\_i\ (a:x)\ (b:y) = \\
\quad \text{if } a < b \text{ then } a : smaller\_i\ x\ (b:y) \\
\quad \text{else if } b < a \text{ then } b : smaller\_i\ (a:x)\ y \\
\quad \text{else } a : smaller\_i\ x\ y
\end{array}
$$

図 2 最終的に得られたプログラム

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、メモアイゼイション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。

5 結　論

本論文では、グラフを扱うプログラムのうちの一部のものについての変換規則を示し、それを用いて単純なプログラムから効率的なプログラムを導出し得ることを示した。具体例として最短経路問題をとりあげ、ダイクストラ法を導出した。

参考文献

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

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

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

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

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

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