社団法人 電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.
SS96-54 (1997-01)

プログラム依存グラフの部分的更新法

○高田 智規, 佐藤 慎一†, 井上 克郎

大阪大学 基礎工学部 情報工学科
† NTT データ通信株式会社 研究開発本部

〒560 大阪府豊中市待兼山町 1-3 情報工学科 井上研究室
Tel 06-850-6571, Fax 06-850-6574
E-mail: t-takada@ics.es.osaka-u.ac.jp

あらまし　プログラム依存グラフ（Program Dependence Graph, PDG）は, プログラム内の文間の依存関係を表す有向グラフである. この PDG の辺をたどることにより, ある文に関連する文の集合（プログラムスライス, スライス）を抽出することができる. PDG やスライスはデバッグ・保守など様々な用途に用いられる.

一般に, PDG の計算には時間がかかるが, これまではプログラムが変更されるたびに PDG 全体を再計算していた. 本研究では, プログラムが変更されたときに, PDG のプログラムの変更箇所に対応する部分だけを更新するアルゴリズムを提案する. これにより, 再計算の時間が軽減され, デバッグ等を効率的に行うことが期待される.

キーワード　プログラム依存グラフ, 更新, プログラムスライス

Incremental Update Method of Program Dependence Graph

○ Tomonori Takada, Shin'ichi Sato†, Katsuro Inoue

Faculity of Engineering Science, Osaka University
† NTT DATA Communications Systems Corporation

Inoue Lab, Department of Information and Computer Science, 1-3, Machikaneyama, Toyonaka, Osaka 560, Japan
Tel: +81-6-850-6571, Fax +81-6-850-6574
E-mail: t-takada@ics.es.osaka-u.ac.jp

Abstract

Program Dependence Graph(PDG) is a direction graph which shows dependencies between statements of a program. A set of reachable nodes(statements) from a node v is called Slice, which shows all related statements to v. PDG and Slice are used various purpose such as debugging and maintenance.
Computing PDG requires fairly long time, however, many systems recompute all of PDG when a source program is modified, even in the case that the modification is very minor and limited one. We propose a method of updating PDG efficiently with minor program modification.

key words　**Program Dependence Graph, PDG, Program Slice, Data Dependence**

1 まえがき

プログラム依存グラフ（Program Dependence Graph, PDG）[5] は，プログラムの各文における変数間の依存関係を表す有向グラフである．PDG の各節点はプログラム中の各文・条件式を表し，有向辺は依存関係（データ依存・制御依存）を表す．PDG の有向辺をたどることにより，ある文における変数と依存関係のある文の集合（プログラムスライス [2]）を抽出することができる．

これまでに，プログラムスライスを抽出しデバッグを効率良く進めるためのシステム [4] を開発した．このシステムでは，プログラムに変更を加えるたびに PDG 全体を再計算しており，それに多くの時間を費していた．しかし，PDG のうちプログラムに変更のあった箇所に関する部分だけを更新することができれば，PDG の再計算に要する時間が短縮され，作業効率の大幅な向上が期待できる．

プログラム内の部分的な変更がその他の文に与える影響を調べるアルゴリズムは既に提案されている [1, 6]．しかし，[1] のアルゴリズムは PDG の利用を考えていない，関数間にまたがる文の依存関係について考慮されていない，などの問題があり，[6] のアルゴリズムでは保持する情報が非常に多くなるという問題がある．

そこで，プログラムの部分的変更が行われた時に効率良く PDG を再計算する手法を提案し，実際に既存のシステムに実装しその有効性を確かめた．

2 諸定義

2.1 入力言語

本研究では Pascal 風言語を入力言語とする．この言語には文として条件文（if 文），代入文，繰返し文（while 文），入力文（readln 文），出力文（writeln 文），手続き呼出し文，複合文（begin-end）がある．変数の型としてはスカラ型のみでポインタ型はない．プログラムは，大域変数宣言，手続き（関数）定義，メインプログラムからなり，ブロック構造はない．手続き内では内部で宣言された局所変数と仮引数変数および大域変数のみが参照可能で，他の手続き内の局所変数は参照できない．手続きは，自己再帰的および相互再帰的に定義可能であり，その引数は，値渡しで扱われる．

2.2 到達定義

文 s における変数 x（の値）の定義が文 t に到達するとは，文 s が変数 x を定義し，かつ，s から t に至る実行パス $s, u_1, u_2, \ldots, u_k, t$ 中に変数 x を再定義しないような実行パス $u_1, u_2, \ldots, u_k$ が存在する場合を言う．

文 n における変数 v の定義が文 s に到達する場合，文 s には到達定義 $\langle n, v \rangle$ が存在すると言う．到達定義集合とは，全ての変数の到達定義を表した集合である．

2.3 依存関係

プログラム中の文間の依存関係として以下の二種類を考える．

1. **データ依存関係**
   (Data Dependence, DD)
   変数 v を参照している文 t において到達定義集合中に $\langle s, v \rangle$ が存在するとき，文 s から文 t に対して変数 v に関するデータ依存関係があるという．

2. **制御依存関係**
   (Control Dependence, CD)
   文 s が条件文または繰り返し文であり，文 s の条件判定の結果によって文 t を実行するか否か決まる時，文 s から文 t への制御依存関係があるという．

2.4 プログラム依存グラフ

プログラム依存グラフ（Program Dependence Graph, PDG）とは，図1のような，プログラム内の各命令間に存在する依存関係を有向グラフで表したものである．本研究では，[3] のアルゴリズムを用いて求めたPDG を対象とする．

プログラム内の文または条件式（条件文・繰り返し文の条件判定部分）に対応した節点（文節点）及び，手続き境界をこえる依存関係の解析などに用いる特殊節点がある．

辺は前述の依存関係を表し，データ依存関係を表す辺はデータ依存辺といい，節点（文）s から節点 t への変数 v に関するデータ依存辺を，$s \xrightarrow{v} t$ と表す．制御依存関係を表す辺は制御依存辺といい，節点 s から節点 t への制御依存辺を $s \dashrightarrow t$ と表す．

依存関係を表す辺の他に，フロー辺と呼ばれる実行順序を表す辺がある[1]．文 s から文 t に（他の文の実行無しに）直接制御が移る可能性がある時，節点 s から節点 t へのフロー辺 $s \rightarrow t$ と表す．パス $s, \ldots . t$ が存在するとは，節点 s からフロー辺を順方向にたどり節点 t へ到達するような経路が存在することをいう．

PDG に対して，以下を定義する．

$source(c, v)$
: 節点 c から変数 v に関するデータ依存辺を遡った（有向辺を逆方向にたどった）節点の集合を表す

$target(c, v)$
: 節点 c から変数 v に関するデータ依存辺を辿った（有向辺を順方向にたどった）節点の集合を表す

$prev(c)$
: 節点 c からフロー辺を遡った節点の集合を表す

[1] フロー辺を PDG に含めない場合が多いが，ここではこのグラフを PDG と呼ぶ

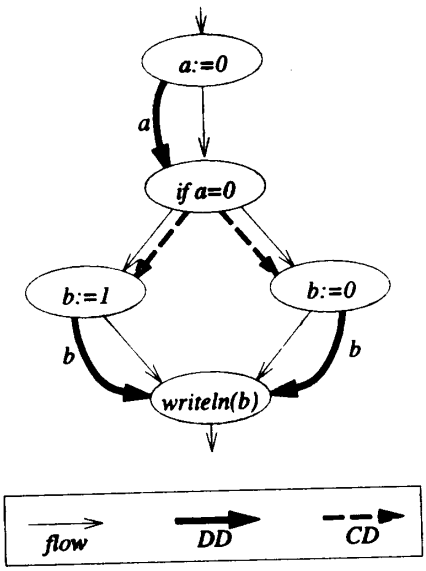

図 1: PDG の例

$next(c)$
: 節点 c からフロー辺を辿った節点の集合を表す

DD データ依存辺の集合

CD 制御依存辺の集合

$FLOW$ フロー辺の集合

2.5 前定義節点

節点 r における変数 w の定義が節点 s の入口に到達している場合，節点 r を，節点 s の変数 w に関する前定義節点と呼ぶ．また，前定義節点の集合を前定義節点集合と呼ぶ．

2.6 プログラムの変更

プログラムの変更として，以下の 3 種類を考える．

- 文の削除
- 文の挿入
- 文の変更

これら3種類の変更のうち，文の削除と挿入によるPDGの更新アルゴリズムを以下に述べる．文の変更によるアルゴリズムは削除と挿入を組み合わせて実現できる．

プログラムの変更は，PDGの文節点一つを対象とする．

3 アルゴリズムの概要

プログラムの変更による制御依存辺の変化は，変更前の制御依存辺を調べることにより容易に調べることができる．そこで，以降はデータ依存辺の変化について述べる．

[6]では，プログラムの変更に伴う到達定義集合の変化をフロー辺を辿り，後の節点に伝えていくことにより，変更の影響によって変化した依存関係を再計算していた．しかし，この方法では各節点ごとに到達定義集合を保存しておかなければならず，無駄であった．

PDGのデータ依存辺は解析時に到達定義集合を基にして引かれている．よって，変更前のPDG中のデータ依存辺を見れば，到達定義集合の一部を知ることができる．例えば，ある節点 s において $r \xrightarrow{v} s$ があったとき，変更前の到達定義集合の中に $\langle r, v\rangle$ が存在していたことになる．また，$* \xrightarrow{v} s$ となるデータ依存辺全てを調べることにより，到達定義集合中の v に関する到達定義は復元することが可能である．

そこで，プログラムが変更されたときに，変更前のPDG中のデータ依存辺を基に依存関係の変化を調べ，PDGの更新を行う．

4 関数内解析アルゴリズム

4.1 前定義節点の発見

アルゴリズム FINDPREDEF
input: 節点 s，変数 v
output: 前定義節点集合 $PreDef(s,v)$

1. $PreDef(s,v) \leftarrow \phi$
2. $c \in prev(s)$ なる各 c に対して2a.以下を順に行う．
   (a) c において v が定義されていれば，$PreDef(s,v) \leftarrow PreDef(s,v) \cup c$
   (b) c において v が参照されていれば，$PreDef(s,v) \leftarrow PreDef(s,v) \cup source(c,v)$
   (c) c において v が定義も参照もされていなければ，$c \leftarrow prev(c)$ として2a.以下を順に行う．

4.2 削除

アルゴリズム DELETEVERTEXSINGLE
input: 削除する節点 s
output: 更新されたPDG

1. s で定義されている各変数 v 全てについて以下を順に行う．
   (a) $DD \leftarrow DD \cup \{ r \xrightarrow{v} t \mid r \in PreDef(s,v),\ t \in target(s,v)\}$
   (b) $FLOW \leftarrow FLOW \cup \{ p \rightarrow n \mid p \in prev(s),\ n \in next(s)\}$
2. 節点 s から出て行くあるいは節点 s に入ってくる全ての辺を削除し，節点 s 自体を削除する．

4.3 挿入

アルゴリズム INSERTVERTEXSINGLE
input: 挿入する節点 s
output: 更新されたPDG
但し，$prev(s), next(s)$ は既に定義してあるものとする．

1. $FLOW \leftarrow FLOW \cup \{ p \rightarrow s \mid p \in prev(s)\} \cup \{ s \rightarrow n \mid n \in next(s)\} - \{ p \rightarrow n \mid p \in prev(s),\ n \in next(s)\}$
2. s で定義している各変数 v 全てについて，以下を順に行う．

(a) 各 $t \in target(s,v)$ に対して，パス $s,...,t$ が存在すれば，以下を順に実行．

i. $DD \leftarrow DD \cup s \xrightarrow{v} t$

ii. t が s 以降に実行され，任意の $r \in PreDef(s,v)$ に対して v を定義しないようなパス $r,...,t$ が存在しなければ，$DD \leftarrow DD - \{ r \xrightarrow{v} t \mid r \in PreDef(s,v)\}$

3. s で参照している各変数 u について，以下を行う．

$DD \leftarrow DD \cup \{ r \xrightarrow{u} s \mid r \in PreDef(s,u)\}$

```
program proc(input,output);
var a: integer;

procedure inc;
begin
  a:=a+1
end;

begin
  readln(a);
  inc;
  writeln(a)
end.
```

図 2: プログラム proc

4.4 変更

変更アルゴリズムは削除・挿入アルゴリズムを組み合わせることで実現できる．

変更前に定義していた変数に対して，削除アルゴリズムの辺に関する部分のみを実行し，参照していた変数に関する辺を削除する．新たに定義・参照している変数に対して，挿入アルゴリズムの辺に関する部分のみを実行する．変更前後で共通して定義・参照している変数に対しては，操作は不要である．

5 関数間解析アルゴリズム

5.1 PDG

図 2のプログラム proc は手続き *inc* の中で大域変数 a を参照・定義している．このプログラムの PDG は図 3のようになる．大域変数の参照・定義のために手続きの入口に global-in, global-out と呼ばれる特殊節点がそれぞれ作られている．

global-in 節点には大域変数を定義している文からデータ依存辺を引き，手続き内で参照する大域変数は全て global-in 節点からデータ依存辺を引く．

global-out 節点には手続き内で大域変数を定義している文からデータ依存辺を引き，手続き外で大域変数を参照している文にデータ依存辺を引く．

関数の場合は，戻り値を表す特殊節点がある他は手続きの場合と同様である．

5.2 削除

アルゴリズム DELETEVERTEX
input: 削除する節点 s
output: 更新された PDG

削除節点 s で大域変数 u を参照し，大域変数 vを定義しているとする．

1. アルゴリズム DELETEVERTEXSINGLE を用いて s を削除する．

2. s の属する関数 f で u を参照している文が s だけなら，2a. 以下を実行．

(a) f中の u に関する global-in 節点 G_{in} を削除．

(b) $\forall a; DD \leftarrow DD - a \xrightarrow{u} G_{in}$

(c) 各 fの呼び出し文 *call*について，以下を順に行う．

i. $\forall a; DD \leftarrow DD - a \xrightarrow{u} call$

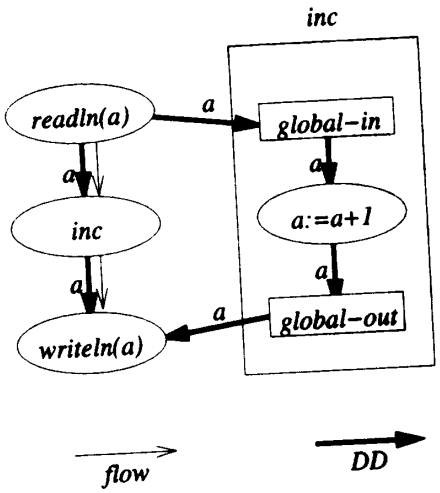

図 3: プログラム proc の PDG

ii. *call*の属する関数で u を参照する文が他になければ，2a. から繰り返す．

3. fで vを定義している文が s だけなら，以下を順に実行．

(a) f中の vに関する global-out 節点および関連する辺を削除．

(b) 各 fの呼び出し文 *call*について以下を実行．

$$DD \leftarrow DD \cup \{ r \xrightarrow{v} t \mid r \in PreDef(call, v), t \in target(call, v)\} - \{ call \xrightarrow{v} t \mid t \in target(call, v)\}$$

(c) *call*の属する関数で vを定義する文が他に無ければ，3a. から繰り返す．

5.3 挿入

アルゴリズム INSERTVERTEX
input: 挿入する節点 s
output: 更新された PDG

挿入節点 s で変数 u を参照し，vを定義しているとする．

1. アルゴリズム INSERTVERTEXSINGLE を用いて s の挿入を行う．

2. s の属する関数 fで u を参照している文が無ければ，以下を順に実行．

(a) global-in 節点 G_{in}を作成する．

(b) $DD \leftarrow DD \cup G_{in} \xrightarrow{u} s$

(c) 各 fの呼び出し文 *call*について，以下を順に実行する．

i. $DD \leftarrow DD \cup \{ r \xrightarrow{v} call \mid r \in PreDef(call, u)\} \cup \{ r \xrightarrow{v} G_{in} \mid r \in PreDef(call, u)\}$

(d) *call*の属するの関数で u を参照している文が他に無ければ，2a. から繰り返す．

3. fで vを定義している文が無ければ，以下を順に実行．

(a) global-out 節点 G_{out}を作成する．

(b) $DD \leftarrow DD \cup s \xrightarrow{v} G_{out}$

(c) 各 fの呼び出し文 *call*について以下を順に実行する．

i. 各 $t \in target(s, v)$ に対して，パス $G_{out}, ...t$ が存在すれば以下を順に実行する．

A. $DD \leftarrow DD \cup G_{out} \xrightarrow{v} t$

B. 任意の $r \in PreDef(call, v)$ に対して，vを定義しないようなパス $r, ...t$ が存在しなければ $DD \leftarrow DD - \{ r \xrightarrow{v} t \mid r \in PreDef(call, v)\}$

(d) *call*の属する関数で vを定義していなかったら 3a. から繰り返す．

5.4 変更

関数間変更アルゴリズムは，同様に，関数内変更アルゴリズムを行った後，特殊節点に関する処理を行えば良い．

6 計算量

PDG の計算・更新にかかわる要素を表 1 に示す．

P	手続きの総数
G	大域変数の総数
L	手続きの局所変数の最大値
S_i	手続き呼び出しの総数
S_t	文の総数
V	PDG の節点の総数
E	PDG の有向辺の総数

表 1: PDG 計算・更新にかかわる要素

6.1 PDG 計算

集合演算にその要素数に比例する計算量が必要だとして，PDG 全体を計算するのに必要な時間計算量は $O(P \cdot S_t \cdot (G+L))$ である [3].

6.2 本手法

前定義節点の発見

アルゴリズム FINDPREDEF の時間計算量は，$O((V+E)\cdot(G+L))$ となる．

削除

アルゴリズム DELETEVERTEXSINGLE の時間計算量は，$O((V+E)\cdot(G+L))$ となる．

その内訳は

FINDPREDEF	$O((V+E)\cdot(G+L))$
データ依存辺追加	$O(E)$
フロー辺追加	$O(E)$
辺削除	$O(E)$
節点削除	$O(V)$

である．

同様に，アルゴリズム DELTEVERTEX の時間計算量は $O(S_i \cdot (V+E)\cdot(G+L))$ となる．

ステップごとに示すと，

1.	$O((V+E)\cdot(G+L))$
2.	$O(V+S_i\cdot E)$
2a.	$O(V)$
2b.	$O(E)$
2c.	2a. .. 2b.$+E\cdot S_i$
3.	$O(S_i\cdot(V+E)\cdot(G+L))$
3a.	$O(V+E)$
3b.	$O(S_i\cdot(V+E)\cdot(G+L))$

挿入

アルゴリズム INSERTVERTEXSINGLE の時間計算量は，$O((V+E)\cdot(G+L))$ となる．

アルゴリズム INSERTVERTEX の時間計算量は $O(S_i\cdot(V+E)\cdot(G+L)+S_i\cdot(V\cdot E+(V+E)(G+L)))$ となる．

変更

関数内変更アルゴリズムの時間計算量は，$O((V+E)\cdot(G+L))$ となる．

関数間変更アルゴリズムの時間計算量は $O(S_i\cdot(V+E)\cdot(G+L)+P\cdot(V\cdot E+(V+E)(G+L)))$ となる．

6.3 考察

6.1節～6.2節に PDG の再計算および更新アルゴリズムの時間計算量を示した．

本アルゴリズムの時間計算量が最悪となるのは PDG が完全グラフであり，特殊節点に関する操作を全ての関数で行う時であり，この時はこれまで通り PDG 全体を計

表 2: 実行時間 (単位は秒. 1/100 単位)

	100 行	450 行	470 行
全体の計算	0.24	6.56	23.28
本手法	0.01	0.01	0.04

算した方が効率が良い. しかし, 通常のプログラムでこのようになることは無い.

一つの関数の大きさはプログラム全体の大きさとは独立で, ある定数以下であると仮定すると FINDPREDEF は O(1) で計算できる.

特殊節点に関する処理は, ある関数内で定義・参照していない大域変数を新たに挿入する, またはある関数内で定義・参照している大域変数を全て削除した時に行うが, 一般にこのような変更はあまり行われない. これにより, 特殊節点に関する操作は $O(1)$ 回繰り返されるだけであると仮定できる.

以上の仮定のもとでは, 削除・挿入・変更全て $O(V+E)$ で計算できる. さらに, PDG 全体を計算するためにはプログラム全体の構文解析などが必要であるが, 本アルゴリズムでは部分的な構文解析だけで済む.

7 実装

本アルゴリズムを [4] のシステムに組み込んだ. そのうち, 削除アルゴリズムの実行時間を表 2 に示す (SPARCstation 20, Memory 64MB).

これにより本アルゴリズムによる PDG の変更がプログラムを変更した後に全体を再計算することに比べ, 有効であると言える.

8 あとがき

プログラムの部分的変更が行なわれた際に, PDG を部分的に変更する手法を提案した. また, 既存のシステムに組み込むことによりその有効性を確認した. 本手法は, 今回実際に組み入れたシステムに限らず, PDG を利用しその一部が頻繁に変更される可能性のあるシステム使用時の作業効率を向上させることが期待される. 本研究では Pascal 風言語を入力言語として扱ったが, PDG およびフロー辺を用いて解析を行っていれば, どのような言語にも適用可能である.

参考文献

[1] Barbara G. Ryder and Marvin C. Pall, "Incremental Data-Flow Analysis Algorithm", *ACM Transactions on Programming Languages and Systems*, vol. 10, 1988, pp. 1–50.

[2] Mark Weiser, "Program Slicing", In *Proceedings of the Fifth International Conference on Software Engineering*, pp. 439–449, San Diego, CA, September 1981.

[3] 植田 良一, 練 林, 井上 克郎, 鳥居 宏次, "再帰を含むプログラムのスライス計算法", 電子情報通信学会論文誌, vol. J78-D-I, January 1995, pp. 11–22.

[4] 佐藤 慎一, 飯田 元, 井上 克郎, "プログラムの依存関係解析に基づくデバッグ支援システ ムの試作", 情報処理学会論文誌, vol. 37, April 1996, pp. 536–545.

[5] Susan Horwitz and Thomas Reps, "The Use of Program Dependence Graphs in Software Engineering", In *Proceedings of the 14th International Conference on Software Engineering*, pp. 392–411, Dallas, Texas, 1992. Association for Computing Machinery.

[6] 鍛冶 武志, "プログラムの部分的変更にともなう依存解析グラフの更新手法", 大阪大学基礎工学部情報工学科 特別研究報告, March 1996.