Московский

Энергетический Институт

(Технический Университет)

Институт \_\_\_\_АВТ

Кафедра ПМ

**ОТЧЕТ**

ПО КУРСОВОЙ РАБОТЕ

По дисциплине

«Параллельные Системы и Параллельные Вычисления»

Тема: Реализация алгоритмов Структурной и Вычислительной сложности программ для функционального языка программирования FPTL

Время выполнения проекта с 15.04.10 по 04.06.10г

Студенты Кузнецов А. И., Шаграев А. Г. А-13-05

1. **Основные определения с позиции теории графов**

Пусть ФС задана в виде системы функциональных уравнений:

Определим ориентированный граф с петлями следующим образом:

Множество вершины такого графа есть множество функциональных переменных схемы, а множество дуг – суть отношение непосредственной зависимости на множестве функциональных переменных. Этот граф – *граф непосредственной зависимости* функциональной схемы.

Сформулируем теперь основные определения в терминах графа непосредственной зависимости. Итак, пусть – граф непосредственной зависимости некоторой ФС.

**Определение 1**. Переменная зависит от переменной *непосредственно* тогда и только тогда, когда .

**Определение 2**. Классом транзитивности переменной будем называть множество вершин, достижимых из .

**Определение 3.** Переменная определена рекурсивно, если через эту вершину проходит хотя бы один цикл.

**Определение 4.** Определение является простым рекурсивным, если для всех , достижимых из , не достижима из . Формально: .

Введем дистанционную матрицу:

где величина равна длине кратчайшего пути из в , если таковой существует, и равна бесконечности, если такового не существует.

**Определение 5.** Определения и взаимно рекурсивны, если достижима из и наоборот, т. е., если .

**Определение 6.** Класс взаимной рекурсивности для - это множество вершин максимальной по вложению сильно связной компоненты орграфа , содержащей . Будем обозначать его .

Эквивалентность введенных определений определениям из лекций достаточно очевидна.

1. **Сложность функциональных схем**

Разумеется, возможно определить понятия вложенности и строгой вложенности определений функциональных переменных; впрочем, эти определения будут дословно повторять определения из лекций, поэтому не будем приводить их здесь.

Использование теории графов позволяет сравнивать по сложности не только функциональные переменные, но также и функциональные схемы.

Так, если и – графы непосредственных зависимостей двух функциональных схем, то, если вкладывается в , можно считать структурную сложность первой функциональной схемы не большей, чем структурную сложность второй функциональной схемы. Это определение необходимо уточнить ввиду того, что рассматриваются взвешенные орграфы.

Скажем, что граф непосредственных зависимостей вкладывается в граф непосредственных зависимостей , если существует функция , такая, что:

1. ,
2. ,

То есть, должно существовать взаимно однозначное отображение между и некоторым подмножеством , сохраняющее дуги графа .

Итак, если граф непосредственных зависимостей функциональной схемы вкладывается в указанном выше смысле в граф непосредственных зависимостей функциональной схемы , скажем, что структурная сложность функциональной схемы *не меньше*, чем структурная сложность функциональной схемы .

Определим теперь отношение строгого порядка.

Скажем, что граф непосредственных зависимостей вкладывается в граф непосредственных зависимостей строго, если существует функция , такая, что:

1. ,
2. ,

То есть, граф вкладывается в граф , но это не изоморфизм.

Таким образом, если граф непосредственных зависимостей функциональной схемы вкладывается строго в указанном выше смысле в граф непосредственных зависимостей функциональной схемы , скажем, что структурная сложность функциональной схемы (строго) меньше, чем структурная сложность функциональной схемы .

1. **Индексы и вектор-индексы структурной спектральной сложности графов в приложении к анализу структурной сложности функциональных программ**

Известно из лекций, что простые определения не сложнее простых рекурсивных, а последние – не сложнее взаимно рекурсивных.

При этом выше уже показано, что определение является взаимно-рекурсивным, если вершина принадлежит нетривиальной компоненте сильной связности; простым рекурсивным, если вершина не достижима ни из одной достижимой из нее вершины, за исключением, возможно, самой этой вершины; простым, если она сама определена не рекурсивно, и все вершины, достижимые из нее, определены также не рекурсивно.

Пусть с весовой функцией – граф непосредственных зависимостей ФС. Вектор

где – размер максимальной по вложению компоненты сильной связности, содержащей вершину , – количество рекурсивно определенных вершин, достижимых из , – количество вершин, достижимых из , будет характеризовать сложность определения функциональной переменной .

Такие вектора допускают простое лексикографическое упорядочивание.

Тогда скажем, что определение не менее сложно, чем определение , если , где вектора сравниваются лексикографически, и сложнее, чем определение , если ,

Это определение соответствует указанным выше соображениям; так, если некоторая функциональная переменная определена взаимно рекурсивно, то первая компонента вектора для нее будет больше единицы. Если другая переменная определена не взаимно рекурсивно, то первая компонента вектора будет равна единице. Тогда первая переменная будет считаться определенной сложнее, чем вторая. При этом логично считать из двух взаимно рекурсивных определений более сложным то, что является взаимно рекурсивным для большего количества переменных. Аналогичные соображения действуют и в отношении остальных двух компонент вектора.

Возможно также построение индекса сложности для вершины; индексом будет являться величина

где – некоторые весовые коэффициенты. Например, если известно, что взаимно рекурсивные определения в пять раз сложнее простых рекурсивных и в 12 раз сложнее простых определений, то индекс может иметь вид

Если введены весовые коэффициенты, то будем говорить, что определение является не менее сложным, чем определение , если , и является более сложным, чем определение , если .

Возможно также и сравнение ФС по сложности на основе введенных характеристик. Если с весовой функцией – граф непосредственных зависимостей функциональной схемы , то индексом сложности функциональной схемы будет являться величина

Интересным является рассмотрение «нормированного» индекса сложности

Сравнение функциональных схем по такой величине оправдывается тем соображением (если его, конечно, принимать как истинное), что сложность меньшей системы, состоящей из более сложных компонент, больше сложности большей системы, состоящей из менее сложных компонент.

Вектор-индексом можно было бы называть вектор, составленный из значений для всех , упорядоченный по невозрастанию. В таком случае две ФС оказываются также сравнимы, коль скоро сравнимыми являются их вектор-индексы.

Также возможно ввести матричную характеристику функциональной схемы. – матрица, столбцы которой являются векторами для всех , при этом столбцы упорядочены по невозрастанию.

Таким образом введенные векторные и матричные характеристики функциональных схем так же, как и числовые, позволяют сравнивать функциональные схемы по структурной сложности.

При этом, естественно, остается открытым вопрос целесообразности использования введенных характеристик; для прояснения этого вопроса необходимо проведение отдельного исследования.

1. **Другие интегральные характеристики графов**

Цикломатический индекс МакКейба:

где *p* – количество компонент сильной связности.

Диаметр и радиус:

Средняя степень (под степенью здесь будем понимать количество входящих и исходящих дуг из вершины):

Также интегральной характеристикой будет являться размер максимальной сильно связной компоненты.

1. **Вычисление характеристик**

Для реализации функций, вычисляющих характеристики графов, понадобится два класса.

class BasicDigraph

{

protected:

vector<vector<size\_t> > edgeList; // список исходящих дуг

public:

BasicDigraph() {}

~BasicDigraph() {}

size\_t size() const;

bool operator () (size\_t v, size\_t u) const;

vector<size\_t> operator [] (size\_t v) const;

friend istream& operator >> (istream& input, BasicDigraph &G);

friend ostream& operator << (ostream& output, const BasicDigraph &G);

};

class AccessibilityDigraph : public BasicDigraph

{

protected:

// имена функциональных переменных, соответствующих вершинам

vector<string> mark;

public:

AccessibilityDigraph() {};

AccessibilityDigraph(const string& program);

void Read(const string &filename);

void Write(const string &filename) const;

string GetMark(size\_t v) const {return mark[v]; };

};

typedef BasicDigraph Digraph;

Первый из них является базовым классом для орграфов, второй реализует его уточнение: класс графа непосредственных зависимостей. Отличие второго класса заключается в реализации методов чтения и записи, также в нем хранятся имена функциональных переменных.

Все вычисляемые характеристики определены для любых ориентированных графов, поэтому все функции, вычисляющие структурную сложность, получают входной аргумент типа const BasicDigraph&.

Вычисление матрицы достижимостей и матрицы расстояний производятся алгоритмом Флойда.

vector<vector<bool> > AccessibilityTable( const BasicDigraph& G )

{

size\_t size = G.size();

vector<vector<bool> > ans( size, vector<bool>( size, false ) );

for (size\_t i = 0; i < size; ++i)

{

for (size\_t j = 0; j < size; ++j)

{

ans[i][j] = G(i,j);

}

}

for (size\_t k = 0; k < size; ++k)

{

for (size\_t i = 0; i < size; ++i)

{

for (size\_t j = 0; j < size; ++j)

{

ans[i][j] = ans[i][j] || ( ans[i][k] && ans[k][j] );

}

}

}

return ans;

}

vector<vector<size\_t> > DistanceMatrix(const BasicDigraph& G)

{

size\_t size = G.size();

vector<vector<size\_t> > ans(size, vector<size\_t> (size, size + 1));

for (size\_t i = 0; i < size; ++i)

{

for (size\_t j = 0; j < G[i].size(); ++j)

{

ans[i][G[i][j]] = 1;

}

}

for (size\_t k = 0; k < size; ++k)

{

for (size\_t i = 0; i < size; ++i)

{

for (size\_t j = 0; j < size; ++j)

{

ans[i][j] = min( ans[i][j], ans[i][k] + ans[k][j] );

}

}

}

return ans;

}

Диаметр, радиус и т.д. вычисляются просто по определению, например:

size\_t Radius(const BasicDigraph &G)

{

vector<vector<size\_t> > dist = DistanceMatrix(G);

size\_t minmax = G.size();

for (size\_t i = 0; i < G.size(); ++i)

{

size\_t max = 0;

for (size\_t j = 0; j < G.size(); ++j)

{

if ( ( dist[i][j] < G.size() ) && ( dist[i][j] > max ) )

{

max = dist[i][j];

}

}

if (max < minmax)

{

minmax = max;

}

}

return minmax;

}

Выделение компонент сильной связности реализуется классическим алгоритмом, см., например, <http://e-maxx.ru/algo/export_strong_connected_components>

1. **Вычислительная сложность**

Будем вычислять сложность следующим образом.

Пусть известны текущие сложности всех определенных в программе функций, а также встроенных функций. При этом возможно вычислить сложность любой определенной через них функции по формулам

Рассмотрим также тернарную операцию условной композиции с условием, верным с вероятностью p.

Тогда с вероятностью p будет выполнено b, с вероятностью (1 – p) будет выполнено c. Стало быть,

Теперь будем по введенным правилам итеративно пересчитывать сложности определенных в программе функций.

В функцию вычисления сложности будем передавать вместо имен функций их текущие сложности. Тогда она получит описание функции в обратной польской нотации: в качестве операций будут выступать операции композиции, в качестве аргументов – текущие сложности. Если приходит константа, то ее сложность считаем равной нулю.

Операции и данные будем хранить в двух разных стеках. Проходя строку описания функции, будем заполнять их:

vector<double> dataStack;

vector<string> opStack;

size\_t pos = 2;

while ( ( pos < data.size() ) && ( data[pos] != ";" ) )

{

if ( IsOp( data[pos] ) )

{

opStack.push\_back(data[pos]);

}

else

{

if ( complexity.find( data[pos] ) != complexity.end() )

{

dataStack.push\_back( complexity.find( data[pos] )->second );

}

else

{

dataStack.push\_back(0);

}

}

++pos;

}

Таким образом, если встретилась операция композиции, она заносится в стек операций, если встретилась функция (либо пользовательская, либо встроенная), то ее текущая сложность помещается на стек данных, иначе на стек кладется ноль (встретилась константа).

Далее происходит вычисление сложности.

while (dataStack.size() > 1)

{

string op = opStack[opStack.size() - 1];

opStack.pop\_back();

double curRes = 0;

double a1, a2, a3;

a1 = dataStack[dataStack.size() - 1];

a2 = dataStack[dataStack.size() - 2];

if (dataStack.size() > 2)

a3 = dataStack[dataStack.size() - 3];

if ((op == "\*") || (op == "+"))

{

curRes = std::max(a1,a2);

dataStack.pop\_back();

dataStack.pop\_back();

}

else if (op == ".")

{

curRes = a1 + a2;

dataStack.pop\_back();

dataStack.pop\_back();

}

else if (op == "->")

{

curRes = std::max(a1, a2);

dataStack.pop\_back();

dataStack.pop\_back();

}

else if (op == "->,")

{

// вероятности здесь. a3 - если условие выполнено

curRes = 0.4\*std::max(a1, a2) + 0.6\*std::max(a1, a3);

dataStack.pop\_back();

dataStack.pop\_back();

dataStack.pop\_back();

}

else

{

// если это не композиция, то это встроенная функция

// сложность встроенной функции - это сложность вычисления

// аргумента + сложность вычисления функции

curRes = a1 + complexity[op];

dataStack.pop\_back();

}

// кладем вычисленную сложность на вершину стека данных

dataStack.push\_back(curRes);

}

Пока стек данных содержит более одного элемента, нужно вычислять сложность. Сложность вычисляется по указанным выше формулам. Вероятность выполнения условия в операции условной композиции вводится пользователем вручную.

**Литература:**

1. Макконнелл С. Совершенный код. Мастер-класс / Пер. с англ. – М.: Издательство «Русская Редакция»; СПб.: Питер, 2007. – 896 стр.: ил.
2. D. Darcy, C. Kemerer, S. Slaughter, J. Tomayko. The structural complexity of software: an experimental test. http://www.pitt.edu/~ckemerer/CK%20research%20papers/TSE0024-0205R1-FinalPaper.doc