Indian Journal of Science and Technology, Vol 9(38), DOI: 10.17485/ijst/2016/v9i38/102163, October 2016
ISSN (Print) : 0974-6846
ISSN (Online) : 0974-5645

A New Approach for Optimization of Program Dependence Graph using Finite Automata

Shanthi Makka* and B. B. Sagar

BITs-Ranchi (Noida Campus), Noida - 201301, Uttar Pradesh, India; shanthi_makka@yahoo.com, shanthi.makka@gmail.com, drbbsagar@gmail.com

Abstract

The Objective is to optimize the time in identification of similar code segment in a program, which is represented using PDG. The method adapted is minimization of finite automata, the states, which are having similar transitions, can be combined into a single state, through which we can remove the duplicate code in program. Methodology used in current study is detection of isomorphic sub graphs in a graph where the program segment has been represented using graph, this approach is very lengthy because finding of sub graphs and identifying isomorphicity between sub graphs. Findings are the demonstrated through an example in figure 1 to figure 4. The efficiency of suggested idea has been demonstrated in analysis part. This approach can be used in compiler optimization phase because it connects computationally related parts of a program; PDG is non-linear data structure in which the transformations can be performed uniformly for both data and control dependences.

Keywords: DFA, Finite Automata, Isomorphic Graphs, Minimization of Finite Automata, Optimized PDG, PDG

1. Introduction

A Graph can be defined as a non-linear data structure, which allows random access to data and there are many algorithms[6] exist to do manipulations on data stored in a Graph. In the field of Computer Science, the term "graph transformation" is wholly employed to evidences the ceremonious skeleton for metamorphosis of the Class Graph, which exemplifies the high level structure of the source code[7]. There are three different types of graph data structures such as Control Flow Graph, Data Flow Graph and PDG essential worn for analysis part, and their transformation algorithms occasionally confide on additional data structure is called as Abstract Syntax Trees (AST) for the factual transformations. But, in this paper we have demonstrated a PDG can also be used for transformations. The fundamental supremacy of graph transformations in accustomed is that they tender a ceremonial and mathematical schema that concedes for disparate perfunctory contingent upon properties, inclusive of the potentiality subject to a given transformation is amend. Flow analysis is a method of analysis of data and control flows of a source program. In object-oriented programming languages[10], preconditions can be verified by the data flow in source program i.e., that the extracted method returns solely one result and the control flow in a program arbitrates whether the method can be extracted at all or not i.e., it has to verify that there must be a single entry and single exit precondition point.

Finite State Automata has great impact in the filed of computer science software as well as in hardware applications. A Finite State Machine (FSM) allows us to have great hypothetical approach in solving various problems in Computer Science and Information technology and Automata[13] also enables application to run at efficient or maximum speed. The increasing computational power of computers does not depend only on the increase CPU frequency but also on other invented technologies. So, the finite automata implementations must consider all these technologies.

A Deterministic Finite Automata can be formally defined as 5 tuples, $M=(Q, \Sigma, \delta, q_0, F)$ where Q is Finite set of states, Σ is Finite set of input alphabet, δ is a Transition

*Author for correspondence

function: QXΣ→Q, which reflects entire behavior of a system, q_0 is a start or initial state from where the processing ahs be done and F is Finite set of Final States, which determines the acceptance of the string.

2. Program Dependence Graph (PDG)

The Program Dependence Graph (PDG) enables both data flow and control flow dependences in a segment of a program. The PDG also represents execution order like DFG. The conventional Program Dependence Graph (PDG)[1] is a directed graph where the vertices denotes computing and control operations in a program and also few vertices are employed as entry nodes (represents entry into a function or a procedure) and exit nodes (indicates going out from a function or procedure). The dependency between segments in a program can be represented by drawing edge between two nodes, which represents two different segments of a program. There are two major constituents, one is that the edges have been differentiated as data and control dependences and secondly we use flag for control dependence edges. If there is a control dependency from a node 'C' to node 'D', the segment denoted by node 'D' will get executed after the segment denoted by node 'C'. In a similar way if there is a data dependency edge from node 'X' to node 'Y', then the segment represented by 'X' assigns some value to a variable, which will be used at the segment represented by 'Y'. The main application of PDG is in program slicing[1,2], extraction of all consecutive or not consecutive statements in to a method, which can affect the value of the variable at given location. In a PDG, the instructions are placed in the vertices of a graph and a control dependence edge decides the execution sequence of statements in a source code and also determines how many times the target code get executed. A data dependence edge indicates the value of some data initialized or updated is positively used by target code or not.

We have considered an example of QUICK SORT, Constructed a PDG where Dotted arrows are used to represent data dependencies and Solid arrows are used for Control Dependencies. The Looping constructs and Recursive calls are represented through self-loops. There is a dotted line from recursive calls to main function, because whenever there is a recursive call in program, the control moves to main function on specified data inputs. The control dependency is already shown through solid line.

```
void QUICK_SORT(int input[20],int l, int h)
{
 int p,b,tp,f;
 if(l<h) {
  p = input[l];
  f = l;
  b = h;
  while(f<b) {
   while((input[f]<=p)&&(f<h))
   {f++; }
   while(input[b]>p)
   { b--; }
   if(f<b) {
    tp=input[f];
    input[f]=input[b];
    input[b]=tp;
 } }
 tp=input[l];
 input[l]=input[b];
 input[b]=tp;
 QUICK_SORT(input,l,b-1);
 QUICK_SORT(input,b+1,h);
 }}
```

→ Data dependences
→ Control dependences

The effective use of PDG is depends only on the existing data flow and nested blocks, but it never depend on the sequence of instructions occurred in a source program. If any two instructions are not having any control and data dependency edges, then we can swap those instructions

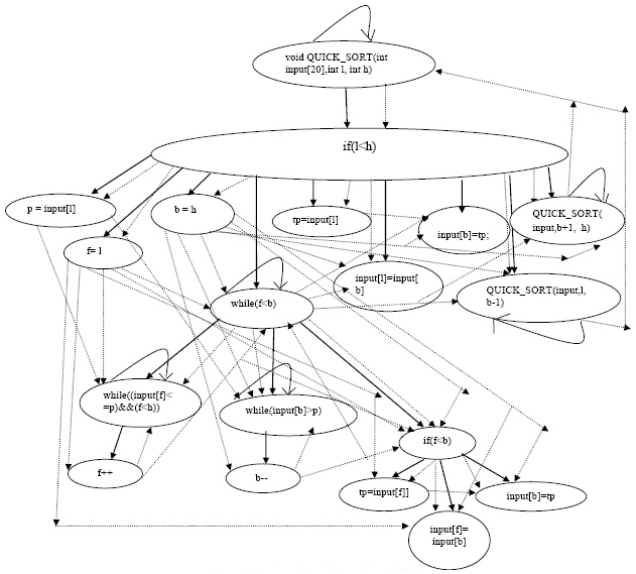

Figure 1. Program Dependence Graph (PDG).

without interfering the basic significance. Mostly it is used in the optimization phase of compilation.

3. Finite State Machines (FSM)

Ceremoniously the scope of FSM covers all the states and transitions, which a machine can accept while moving ahead with sequence of input symbols in input alphabet. Suppose you want to write a program to recognize the word "main" refer below program segment:

```
l:  Scanf("%c",&ch)
    while (char != "m") Scanf("%c",&ch)
    if (Scanf("%c",&ch)!= "a") goto l
    if (Scanf("%c",&ch)!= "i") goto l
    if (Scanf("%c",&ch)!= "n") goto l
    done
```

The Program can be explained as follows:

Initialization
Searching for "m"
Recognized "m", Searching for "a"
Recognized "ma", Searching for "i"
Recognized "mai", Searching for "n"
Recognized "main"

The demonstration of entire process in a graphical way as follows:

i. Every vertex represents a statement in a process
ii. Edges or arcs from one vertex to other shows the movement from one statement to another statement
iii. The Labels on the arcs denotes the to the input required to make a transition

The processing of input string using Finite Automata is as follows:

i. Start processing at initial state
ii. If the next input symbol in the input string matches with character on the arc then Automata moves the next state
iii. Repeat the process for all the characters in a input string

a. If there is no further move possible, then simply stop
b. If an automata reaches to final state after processing entire input string then accept

Basically, there are two variants of Finite Automata are there, one is DFA, a Deterministic Finite Automata: From every state for every input symbol the transition must be there and that transition must be unique. The second category is NDFA, Non Deterministic Finite Automata: Not compulsory to have transitions for every state on every input symbol, transition may not unique. Fundamentally NDFA can have multiple paths simultaneously where as DFA must have single path at the same time.

4. Optimization of PDG using FA

We proposed an approach for Optimization of PDG using FA: The concept of minimization of Finite Automata has been used to identify similar or identical transitions, can be merged further through which we can have minimum number of states which fulfills the same task.

4.1 Minimization Algorithm [14] of Finite Automata

i. Identify and remove the unreachable states. These states are the states with no incoming transition, but only outgoing transition.
ii. Draw two transition tables T_1 and T_2 where T_1 contains all rows which contain states from Q-F and T_2 contains all the states from set F.
iii. All trap states are indistinguishable. So we remove all the trap states except for the one with the lowest index and replace the trap states reference with the only trap state left.
iv. Find the similar rows from T_1 such that the states after transition on a given input are same for those states. From the set of similar rows remove all the rows from the table except the one with the smallest index and make corresponding changes to the table.
v. Repeat the above step till all redundant rows have been eliminated.
vi. Repeat step iv and v for table T_2
vii. Now combine the tables to get the minimized DFA.

4.2 Demonstration of our approach with an example

Let us consider an example of finding Greatest Common Divisor (GCD) of two numbers.

```
void function() {int x ,y;
scanf("%d",&x);
scanf("%d",&y);
while ( x !=y) {
if (x> y)
x = x - y;
else y= y -x;
}printf("%d",x); }
```

Renaming of Statements for simplification: Rename all the statements in above program as A,B,C,D, E, F, G and H for void function() , scanf("%d",&x), scanf("%d",&y), **while (** x !=y),**if** (x> y), x = x – y, y= y –x, printf("%d",x) respectively. Redraw the PDG, by mentioning variable dependency on the arc of data dependency line.

Now, we can apply the concept of minimization of finite automata

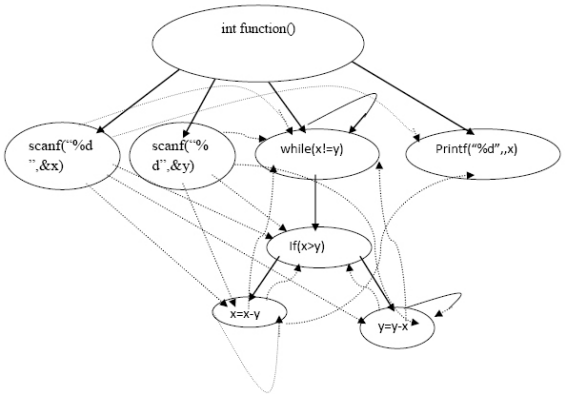

Figure 2. PDG for Greatest Common Divisor.

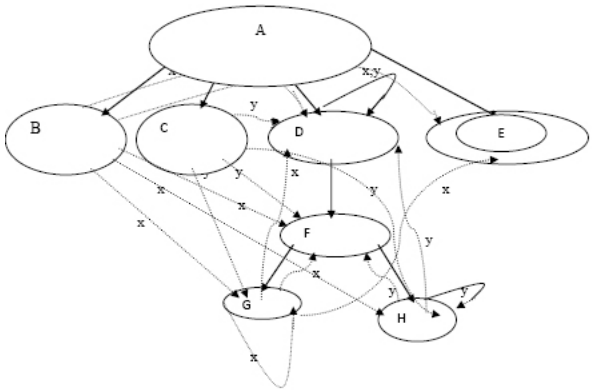

Figure 3. PDG for Greatest Common Divisor along with renaming of nodes and labeling.

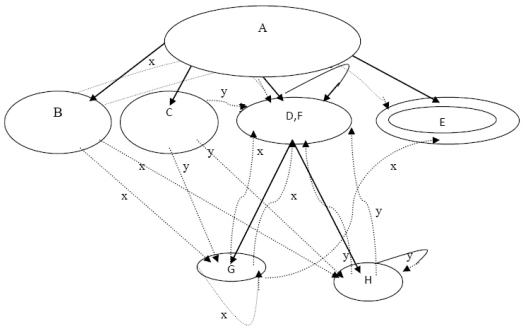

Figure 4. Optimized PDG for Greatest Common Divisor.

L1={{E},{A},{B,C,D,F,G,H}}
L2={{E},{A},{B,G},{C,H},{D,F}}
L3={{E},{A},{B},{C},{H},{D,F},{G}}
L4={{E},{A},{B},{C},{H},{D,F},{G}}
Stop when previous and current set is same (i.e. L3=L4)

4.3 Algorithm

Construct PDG for the given segment of a Program. Renaming of nodes with labels like A,B,C,…. Etc. Reconstruct the PDG. Apply minimization approach to identify similar transitions. Merge identical transitions in to single node. You can see final Optimized PDG.

4.4 Applications of PDG

4.4.1 Program Slicing using PDG

A program slicing is an immediate application of PDG can be defined as a sequence of steps, where the value of a variable get changed at particular segment of a program. Slicing cannot be used directly for "extract method" refactoring[8] mentioned in this paper, which is to clipping of a remnant of continuous statements. Let us consider below example:

```
int sum,n,min;
sum=0;
min= ∞
for (int i = 0; i < n; i++)
{
sum+ = a[i];
if(min>a[i])
min= a[i];}
printf("Summation= ", sum);
printf("minimum value= ", min);
```

After the slicing for finding sum is as follows:

```
 int summation(int[] a) {
int sum= 0;
for (int i = 0; i < n; i++) {
sum+= a[i];
}return sum;}
```

After the slicing for finding smallest element is as follows:

```
int smallest(int[] a)
{int min= ∞;
for (int i = 0; i < n; i++)
{if(min>a[i])
min=a[i];
}return min;
}
```

By extracting both the methods in above slices, the original code can be turned into:

```
 int sum= summation(a);
int p = smallest(a);
printf("Summation= ", sum);
printf("minimum value= ", min);
```

If the PDG of the code is known, slicing is nothing but a reverse the edges of PDG[3] and collects all the vertices, which are reachable from the vertex containing the final targeted variable. Contemplate that, Slicing extracts not only consecutive statements, but also when slicing using two different variables, some statements may be common in both slices. In above example 'for' loop got dualized. A slicing can be used to return multiple results through methods and it is also benefited in analysis and implementation parts which were discussed in detailed in previous papers[3-5].

4.4.2 Identification of Similar Sub Graphs

Now a day's all most in all software systems[1] similar or duplicated code is common, all though programmers knows that cut, copy and paste is a bad practice but still it is used by everyone. The identification of similar code can be done by finding similar sub graphs in a directed graph. That kind of similar sub graphs are called as isomorphic graphs. This approach is used on PDG not only to determine morphological configuration of programs but also the data flow within the segments of program. This approach can be implemented within the non-polynomial amount of time. The main draw back with this approach is software maintenance is very difficult, but it is very easy and cheap during the software development (i.e. the modifications done in the primitive code should also implied in replicated code and errors might have been duplicated in the duplicated code). Two graphs are said to be isomorphic if they have similar number of vertices are connected in a same way. That means the path existing between every two vertices should be similar in both isomorphic graphs.

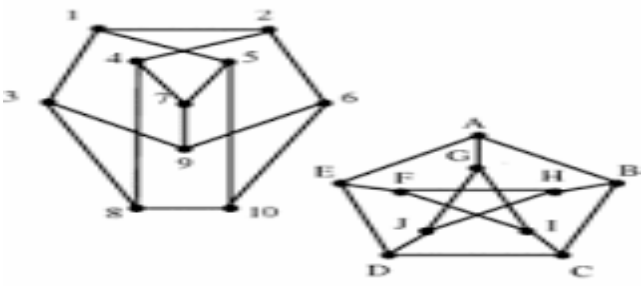

Figure 5. Isomorphic graphs.

5. Analysis

Our suggested approach begins with the constructions of PDG for the given segment of a program, which postulates O (n^2) time where n is number of operational statements. Then we optimize PDG using the concept of minimization of a Finite Automata, which takes O (mc log m) time, where m is the number of states and c is the size of an input alphabet. But number of statements in a PDG is equal to the number of states of a finite automata i.e., m=n. So by substituting 'n' in place of 'm' the resultant time turns in to

O (n*c log n). So the total time required to implement our approach is **T (n)= O (nc log n)+ O(n 2).** We have also plotted 3 D surface graph for this analysis part by taking c, n, and T (n) as three dimensions.

6. Related Work

Construction of isomorphic sub graphs to identify duplicated code (clone) in a program[11] was already discussed by R. Komondoor and S. Horowitz, but their procedure has certain limitations like they have to visit every node exactly once during sub graph construction and their approach cannot analyze big programs because of PDG generating infrastructure. In[1,12] they mentioned a procedure to identify similar code in a program is similar to identification of differences between programs based on PDG, i.e. once if we can identify differences between modules of programs and from the remaining segments clone identification makes easier. Few papers also demonstrated how PDG can be used for complier optimization phase. In[13], they have implemented Deterministic Finite Automata on Parallel Computers.

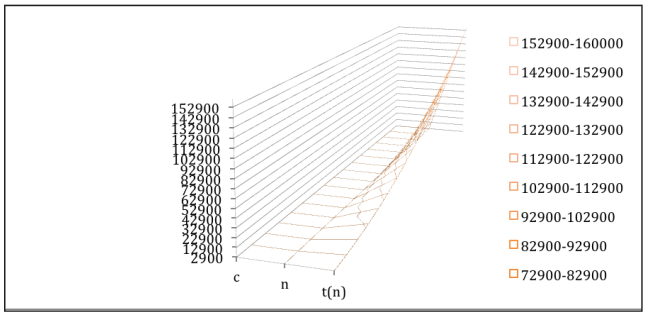

Figure 6. Graphical representation of performance analysis.

7. Conclusion

An efficient way or structure to represent a program, called Program Dependence Graph has been discussed and also demonstrated how to construct a PDG for segment of a program with an example. In this paper we have presented an approach for optimization of PDG using the concept of minimization finite automata i.e., we have also demonstrated an approach in an algorithmic form and a framework to identify similar code through similar sub graphs using program dependence graphs. Program slicing concept also demonstrated with an appropriate example. Different ways to model or represent a computer program also discussed in this paper, i.e. a class graph, abstract syntax tree, control flow graph along with an appropriate examples. An approach to identify clone or duplicated code in the program demonstrated with an algorithm.

8. Future Work

We have discussed optimization of PDG using minimization of Finite Automata, this approach can be extended in identification of identical code. We have also demonstrated how PDG can be used for refactoring[9] for clone detection, slicing and extract method with few examples. This work can be extended to make efficient algorithm to identify independent modules in a programs. After that how these modules can be executed on or according to the topology suggested. This work can further bring in to the parallelization scenario.

9. References

1. Horwitz SB, Reps TW. The use of program dependence graphs in software engineering**.** In Proceedings of the Fourteenth International Conference on Software Engineering. 1992.
2. Horwitz SB, Reps TW, Binkley D. Interprocedural slicing using dependence graphs. ACM Transactions on Programming Languages and Systems. 1990; 12(1).
3. Komondoor R. Automated Duplicated-Code Detection and Procedure Extraction, PhD at the University of Wisconsin: Madison, 2003.
4. Ettinger R. Refactoring via Program Slicing and Sliding, PhD at the Oxford University Computing Laboratory: Programming Tools Group, 2006.
5. Zanardini D. The Semantics of Abstract Program Slicing, Proc. of the 8th IEEE Int Working Conf on Source Code Analysis and Manipulation. 2008. p. 89–98.
6. Cormen TH. Introduction to Algorithms. 2nd edn. MIT Press, 1998.
7. Mens T. On the Use of Graph Transformations for Model Refactoring, Generative and Transformational Techniques in Software Engineering. 2006; 219–57.
8. Martin Fowler. Refactoring Home Page. Available from: http://www.refactoring.com.
9. Mens T, Tourwe T. A Survey of Software Refactoring. IEEE Transactions on Software Engineering. 2004; 30(2):126–39.
10. Verbaere M, Ettinger R, de Moor O. Jun GL: A Scripting Language for Refactoring, 28th International Conference on Software Engineering. 2006. p. 172–81.
11. Komondoor R, Horwitz S. Using slicing to identify duplication in source code. In Eighth International Static Analysis Symposium (SAS), 2001.
12. Horwitz S, Prins J, Reps T. Integrating noninterfering versions of programs. ACM Transactions on Programming Languages and Systems. 1989; 11(3).
13. Holub J, Stekr S. Implementation of Deterministic Finite Automata on Parallel Computers. This research has been partially supported by the Ministry of Education, Youth and Sports under research program MSM 6840770014 and the Czech Science Foundation as project 201/06 (2009): 1039.
14. Hopcroft JE, Motwani R, Ullman JD. Introduction to automata theory, languages, and computation. ACM SIGACT News. 2001; 32(1):60–5.