**Name: Bhavesh Kewalramani**
**Roll No.: A-25**

# Practical No. 9

## Theory

### Basic Block

The basic block is a set of statements. The basic blocks do not have any in and out branches except entry and exit. It means the flow of control enters at the beginning and will leave at the end without any halt. The set of instructions of basic block executes in sequence.
Here, the first task is to partition a set of three-address code into the basic block. The new basic block always starts from the first instruction and keep adding instructions until a jump or a label is met. If no jumps or labels are found, the control will flow in sequence from one instruction to another.
The algorithm for the construction of basic blocks is given below:
**Algorithm:** Partitioning three-address code into basic blocks.
**Input**: The input for the basic blocks will be a sequence of three-address code.
**Output**: The output is a list of basic blocks with each three address statements in exactly one block.
**METHOD:** First, we will identify the leaders in the intermediate code. There are some rules for finding leaders, which are given below:

1. The first instruction in the intermediate code will always be a leader.
2. The instructions that target a conditional or unconditional jump statement are termed as a leader.
3. Any instructions that are just after a conditional or unconditional jump statement will be a leader.

Each leader's basic block will have all the instructions from the leader itself until the instruction, which is just before the starting of the next leader.
**Example:**
Consider the following source code for a 10 x 10 matrix to an identity matrix.

```
for i from 1 to 10 do

     for j from 1 to 10 do

       a [ i, j ] = 0.0;

 for i from 1 to 10 do

       a [ i,i ] = 1.0;
```

The three address code for the above source program is given below:

```
1) i = 1
```

```
2) j = 1

3) t1 = 10 * i

4) t2 = t1 + j

5) t3 = 8 * t2

6) t4 = t3 - 88

7) a[t4] = 0.0

8) j = j + 1

9) if j <= 10 goto (3)

10) i = i + 1

11) if i <= 10 goto (2)

12) i = 1

13) t5 = i - 1

14) t6 = 88 * t5

15) a[t6] = 1.0

16) i = i + 1

17) if i <= 10 goto (13)
```

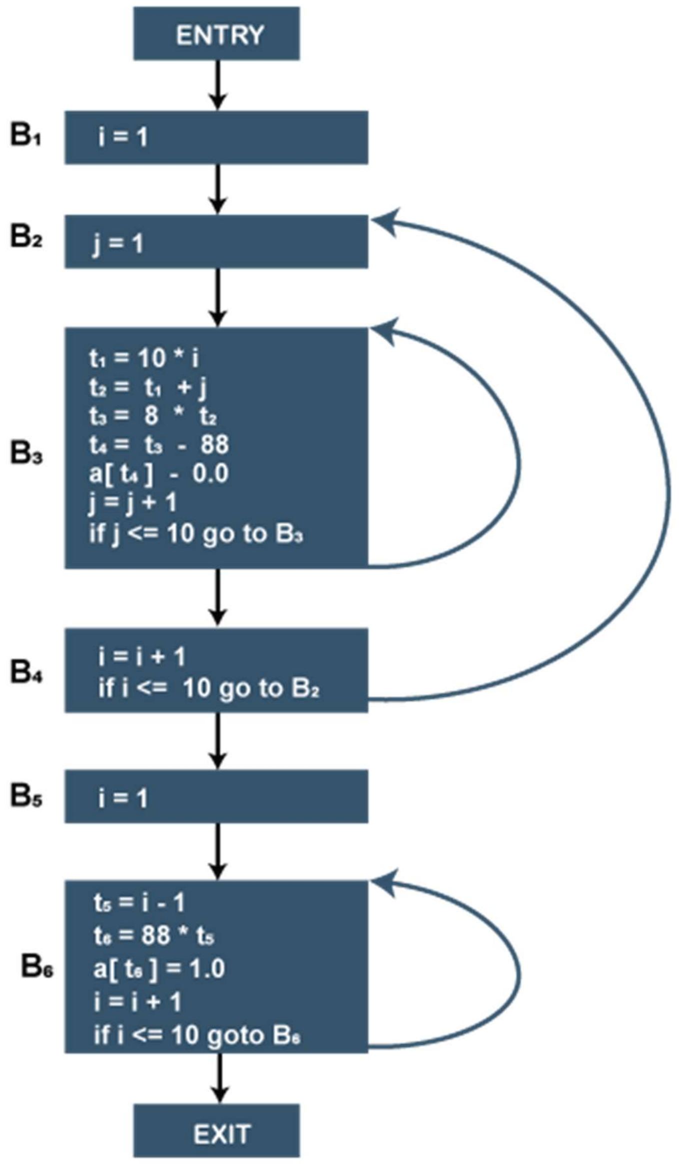
- According to the given algorithm, instruction 1 is a leader.
- Instruction 2 is also a leader because this instruction is the target for instruction 11.
- Instruction 3 is also a leader because this instruction is the target for instruction 9.
- Instruction 10 is also a leader because it immediately follows the conditional goto statement.
- Similar to step 4, instruction 12 is also a leader.
- Instruction 13 is also a leader because this instruction is the target for instruction 17.

So there are six basic blocks for the above code, which are given below:
**B1 for statement 1**
**B2 for statement 2**
**B3 for statement 3-9**
**B4 for statement 10-11**
**B5 for statement 12**
**B6 for statement 13-17.**

**Flow Graph**

It is a directed graph. After partitioning an intermediate code into basic blocks, the flow of control among basic blocks is represented by a flow graph. An edge can flow from one block X to another block Y in such a case when the Y block's first instruction immediately follows the X block's last instruction. The following ways will describe the edge:

- There is a conditional or unconditional jump from the end of X to the starting of Y.
- Y immediately follows X in the original order of the three-address code, and X does not end in an unconditional jump.



Flow graph for the 10 x 10 matrix to an identity matrix.

- Block B1 is the entry point for the flow graph because B1 contains starting instruction.
- B2 is the only successor of B1 because B1 doesn't end with unconditional jumps, and the B2 block's leader immediately follows the B1 block's leader.
- B3 block has two successors. One is a block B3 itself because the first instruction of the B3 block is the target for the conditional jump in the last instruction of block B3. Another successor is block B4 due to conditional jump at the end of B3 block.
- B6 block is the exit point of the flow graph.

## LOOPS IN FLOW GRAPH

A graph representation of three-address statements, called a flow graph, is useful for understanding code-generation algorithms, even if the graph is not explicitly constructed by a code-generation algorithm. Nodes in the flow graph represent computations, and the edges represent the flow of control.

**Dominators:**

In a flow graph, a node d dominates node n, if every path from initial node of the flow graph to n goes through d. This will be denoted by d dom n. Every initial node dominates all the remaining nodes in the flow graph and the entry of a loop dominates all nodes in the loop. Similarly every node dominates itself.

Example:
*In the flow graph below,

*Initial node,node1 dominates every node. *node 2 dominates itself

*node 3 dominates all but 1 and 2. *node 4 dominates all but 1,2 and 3.
*node 5 and 6 dominates only themselves,since flow of control can skip around either by goin through the other.

*node 7 dominates 7,8 ,9 and 10. *node 8 dominates 8,9 and 10.
*node 9 and 10 dominates only themselves.

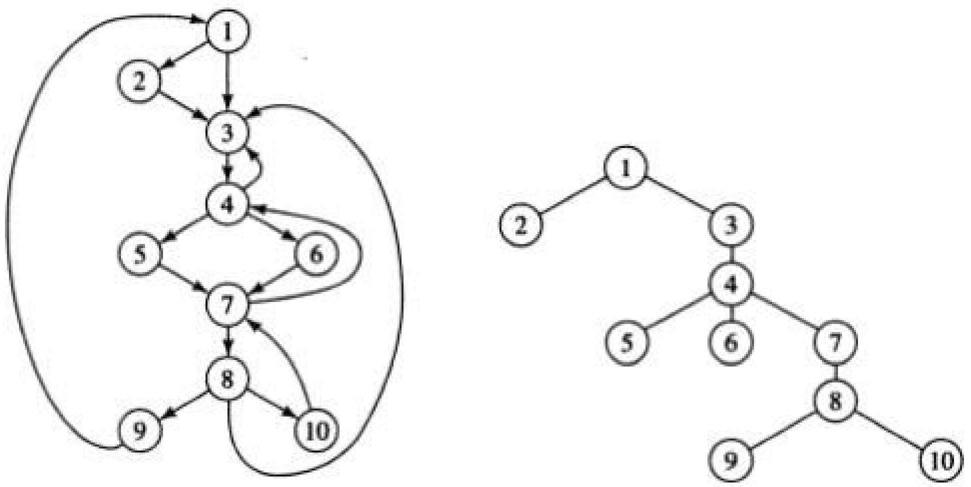


**Fig. 5.3(a) Flow graph (b) Dominator tree**

The way of presenting dominator information is in a tree, called the dominator tree, in which

- The initial node is the root.
- The parent of each other node is its immediate dominator.
- Each node d dominates only its descendents in the tree.

The existence of dominator tree follows from a property of dominators; each node has a unique immediate dominator in that is the last dominator of n on any path from the initial node to n. In terms of the dom relation, the immediate dominator m has the property is d=!n and d dom n, then d dom m.

***

D(1)={1}
D(2)={1,2}
D(3)={1,3}
D(4)={1,3,4}
D(5)={1,3,4,5}
D(6)={1,3,4,6}
D(7)={1,3,4,7}
D(8)={1,3,4,7,8}
D(9)={1,3,4,7,8,9}
D(10)={1,3,4,7,8,10}

**Natural Loops:**

One application of dominator information is in determining the loops of a flow graph suitable for improvement. There are two essential properties of loops:

- Ø A loop must have a single entrypoint, called the header. This entry point-dominates all nodes in the loop, or it would not be the sole entry to the loop.
- Ø There must be at least one way to iterate the loop(i.e.)at least one path back to the headerOne way to find all the loops in a flow graph is to search for edges in the flow graph whose heads dominate their tails. If a→b is an edge, b is the head and a is the tail. These types of

edges are called as back edges.

Example:

In the above graph,

7→4 4 DOM 7

10 →7 7 DOM 10
4→3
8→3
9 →1

The above edges will form loop in flow graph. Given a back edge n → d, we define the natural loop of the edge to be d plus the set of nodes that can reach n without going through d. Node d is the header of the loop.

**Algorithm: Constructing the natural loop of a back edge**.
Input: A flow graph G and a back edge n→d.

Output: The set loop consisting of all nodes in the natural loop n→d.

Method: Beginning with node n, we consider each node m*d that we know is in loop, to make sure that m's predecessors are also placed in loop. Each node in loop, except for d, is placed once on stack, so its predecessors will be examined. Note that because d is put in the loop initially, we never examine its predecessors, and thus find only those nodes that reach n without going through d.

Procedure insert(m);

if m is not in loop then begin loop := loop U {m};

push m onto stack end;
stack : = empty;

```
loop : = {d}; insert(n);
while stack is not empty do begin
pop m, the first element of stack, off stack;

        for each predecessor p of m do insert(p)
end
```

**Inner loops:**

If we use the natural loops as “the loops”, then we have the useful property that unless two loops have the same header, they are either disjointed or one is entirely contained in the other. Thus, neglecting loops with the same header for the moment, we have a natural notion of inner loop: one that contains no other loop.

When two natural loops have the same header, but neither is nested within the other, they are combined and treated as a single loop.

**Pre-Headers:**

Several transformations require us to move statements “before the header”. Therefore begin treatment of a loop L by creating a new block, called the preheader. The pre-header has only the header as successor, and all edges which formerly entered the header of L from outside L instead enter the pre-header. Edges from inside loop L to the header are not changed. Initially the pre-header is empty, but transformations on L may place statements in it.

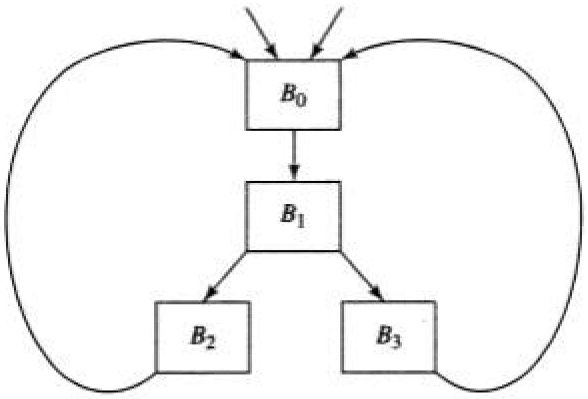


**Fig. 5.4 Two loops with the same header**

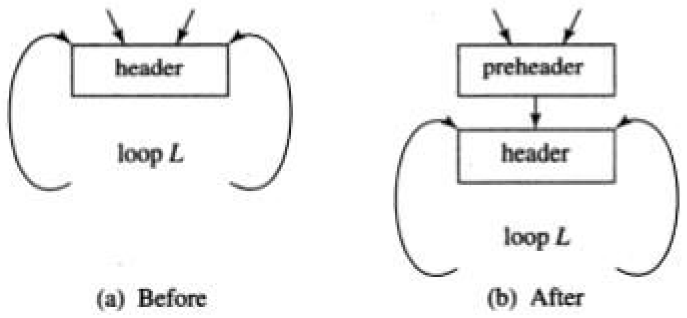


**Fig. 5.5 Introduction of the preheader**

**Reducible flow graphs:**

Reducible flow graphs are special flow graphs, for which several code optimization transformations are especially easy to perform, loops are unambiguously defined, dominators can be easily calculated, data flow analysis problems can also be solved efficiently. Exclusive use of structured flow-of-control statements such as if-then-else, while-do, continue, and break statements produces programs whose flow graphs are always reducible.

The most important properties of reducible flow graphs are that

1. There are no umps into the middle of loops from outside;
2. The only entry to a loop is through its header

**Definition:**

A flow graph G is reducible if and only if we can partition the edges into two disjoint groups, forward edges and back edges, with the following properties.

1. The forward edges from an acyclic graph in which every node can be reached from initial node of G.
2. The back edges consist only of edges where heads dominate theirs tails.

Example: The above flow graph is reducible. If we know the relation DOM for a flow graph, we can find and remove all the back edges. The remaining edges are forward edges. If the forward edges form an acyclic graph, then we can say the flow graph reducible. In the above example remove the five back edges 4→3, 7→4, 8→3, 9→1 and 10→7 whose heads dominate their tails, the remaining graph is acyclic.

**Practicals**

---

**Aim:**

Write a program to perform loop detection by finding leader, basic blocks and program flow graph &amp; natural loop.

**Program:**

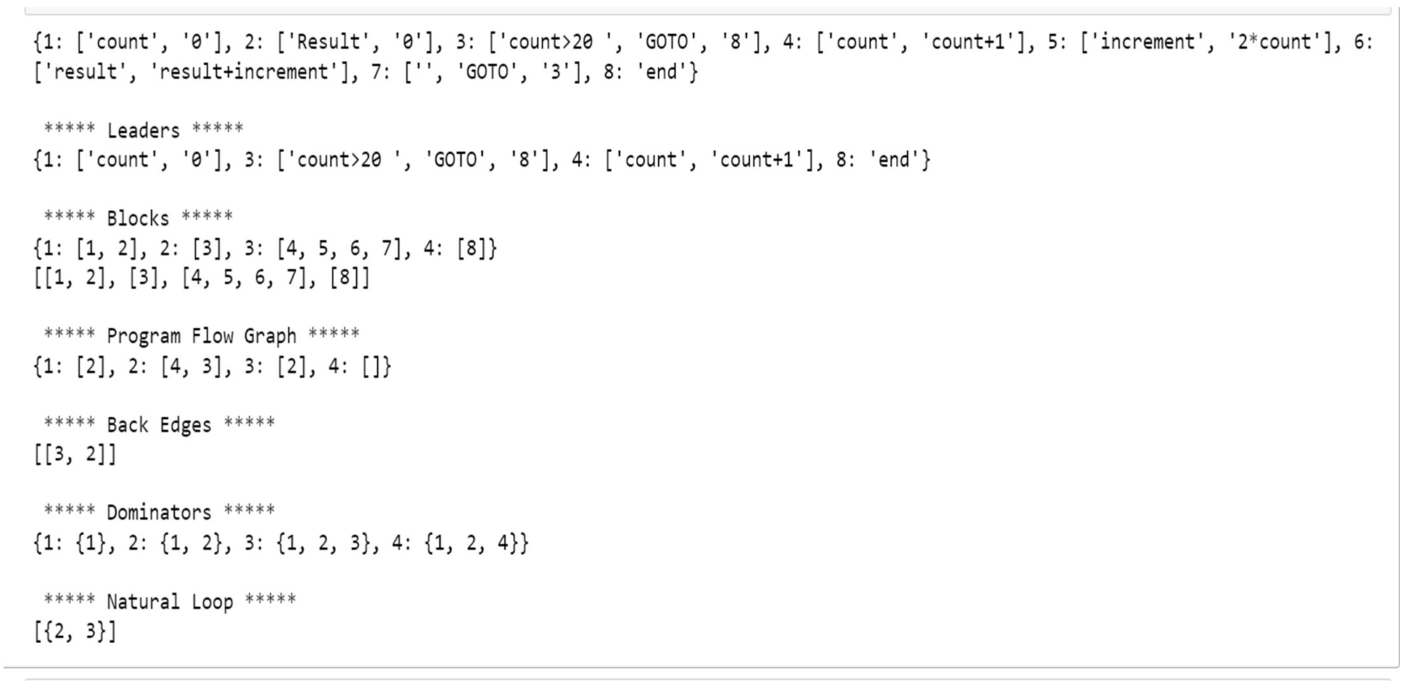
```
from typing import List
from collections import deque

def printpath(path: List[int]) -> None:

    size = len(path)
    for i in range(size):
        print(path[i], end = " ")

    print()

def isNotVisited(x: int, path: List[int]) -> int:

    size = len(path)
    for i in range(size):
        if (path[i] == x):
            return 0

    return 1

def findpaths(g: List[List[int]], src: int,dst: int, v: int):


    q = deque()
    c=[]
    path = []
    path.append(src)
    q.append(path.copy())

    while q:
        path = q.popleft()
        last = path[len(path) - 1]

        if (last == dst):
            c.append(set(path))
        for i in range(len(g[last])):
            if (isNotVisited(g[last][i], path)):
                newpath = path.copy()
                newpath.append(g[last][i])
                q.append(newpath)
    return c


def naturalLoop(g: List[List[int]], src: int,dst: int, v: int, edg: int):


    q = deque()
    c=[]
    path = []
    path.append(src)
    q.append(path.copy())

    while q:
        path = q.popleft()
        last = path[len(path) - 1]

        if (last == dst) and (edg not in path):
            c.append(set(path))

        for i in range(len(g[last])):
            if (isNotVisited(g[last][i], path)):
                newpath = path.copy()
                newpath.append(g[last][i])
                q.append(newpath)
    return c

a=["count=0","Result=0","If count>20 GOTO
8","count=count+1","increment=2*count","result=result+increment","GOTO 3","end"]

k=1
dict1={}
for i in a:
    lhs=[]
    if "=" in i:
        i,j=i.split("=")
        lhs.append(i)
        lhs.append(j)
        dict1.update({k:lhs})
        k+=1
    elif "GOTO" in i:
        lhs=[]
```

```
            i=i.replace("If"," ")
            i=i.strip()
            i,j=i.split("GOTO")
            lhs.append(i)
            lhs.append("GOTO")
            j=j.strip()
            lhs.append(j)
            dict1.update({k:lhs})
            k+=1
        else:
            dict1.update({k:i})
            k+=1
    print(dict1)

    pfg1={}

    for k,v in dict1.items():
        if "GOTO" in v:
            pfg1.update({k:v})

    # print(pfg1)

    leader={}
    last=k
    m=1
    for k,v in dict1.items():
        if k==1:
            leader.update({k:v})
            m+=1
        elif "GOTO" in v:
            leader.update({int(v[2]):dict1[int(v[2])]})
            m+=1
        elif "GOTO" in dict1[k-1] and dict1[k-1][0]!="":
            leader.update({k:v})
            m+=1

    sk=sorted(leader.keys())
    leader1={}
    for k in sk:
        leader1.update({k:leader[k]})
    leader=leader1.copy()
    print("\n ***** Leaders *****")
    print(leader)

    sk.append(last+1)
    # print(sk)
```

```
blocks={}
m=1
for k in range(len(sk)-1):
    b=[]
    for j in range(sk[k],sk[k+1]):
        b.append(j)
    blocks.update({m:b})
    m+=1

print("\n ***** Blocks ***** ")
print(blocks)

pfg={}

sk11=list(blocks.values())
print(sk11)

for k,v in blocks.items():
    c=[]
    for i in v:
        if i==1:
            c.append(k+1)
        if "GOTO" in dict1[i]:
            ind=0
            for j in range(len(sk11)):
                if int(dict1[i][2]) in sk11[j]:
                    ind=j+1
            if ind not in c:
                c.append(ind)
            if dict1[i][0]!="" and k+1 not in c:
                c.append(k+1)

    pfg.update({k:c})

print("\n ***** Program Flow Graph *****")
print(pfg)

backedges=[]
for k,v in pfg.items():
    for i in v:
        if k>i:
            backedges.append([k,i])

print("\n ***** Back Edges *****")
print(backedges)
```

```
paths=[]
for k,v in pfg.items():
    for i in v:
        paths.append([k,i])

g = [[] for _ in range(len(list(blocks.keys()))+1)]

for i in paths:
    g[i[0]].append(i[1])

skeys=list(blocks.keys())

dominator={}
for j in range(0,len(skeys)):
    c=findpaths(g,1,skeys[j],len(skeys))
    u = set.intersection(*c)
    dominator.update({skeys[j]:u})

print("\n ***** Dominators *****")
print(dominator)


naturalloop=[]
for j in range(len(backedges)):
    d=[]
    u=set()
    for k in range(len(skeys)):
        temp1=set()
        temp1.add(backedges[j][1])
        c=naturalLoop(g,skeys[k],backedges[j][0],len(skeys),backedges[j][1])
        u = u.union(*c)
    u = u.union(temp1)
    naturalloop.append(u)

print("\n ***** Natural Loop *****")
print(naturalloop)
```

**Output:**



```
{1: ['count', '0'], 2: ['Result', '0'], 3: ['count>20 ', 'GOTO', '8'], 4: ['count', 'count+1'], 5: ['increment', '2*count'], 6:
['result', 'result+increment'], 7: ['', 'GOTO', '3'], 8: 'end'}

 ***** Leaders *****
{1: ['count', '0'], 3: ['count>20 ', 'GOTO', '8'], 4: ['count', 'count+1'], 8: 'end'}

 ***** Blocks *****
{1: [1, 2], 2: [3], 3: [4, 5, 6, 7], 4: [8]}
[[1, 2], [3], [4, 5, 6, 7], [8]]

 ***** Program Flow Graph *****
{1: [2], 2: [4, 3], 3: [2], 4: []}

 ***** Back Edges *****
[[3, 2]]

 ***** Dominators *****
{1: {1}, 2: {1, 2}, 3: {1, 2, 3}, 4: {1, 2, 4}}

 ***** Natural Loop *****
[{2, 3}]
```