5.4 LOOPS IN FLOW GRAPHS

A Loop is a sequence of statements that is executed repeatedly as long as a particular condition is satisfied.

In Compiler Design, loops are represented using Flow Graphs. Identifying loops is important because loops may execute many times, so optimizing code inside a loop can significantly improve program performance.

Example of a Loop

Consider the following program:

```
i = 1;


while (i <= 5)
{
   sum = sum + i;
   i = i + 1;
}
```

The corresponding intermediate code may be:

1. i = 1
2. if i > 5 goto 6
3. sum = sum + i
4. i = i + 1
5. goto 2
6. end

Here, statements 2 to 5 are repeatedly executed until the condition becomes false.

5.4.1 Back Edge

A Back Edge is an edge in a flow graph that goes from a block back to an earlier block in the control flow.

Example:

B2

B3

Exit

B4

B4 → B2 is the Back Edge

Importance:

Back edges are useful for identifying loops in a flow graph.

5.4.2 Natural Loop

A Natural Loop is a loop formed around a back edge where the target of the back edge is a suitable entry point called the Loop Header.

For a back edge:

\[n \rightarrow d \]

where:

- n = Source block.
- d = Destination block or loop header.

The natural loop consists of:

1. The header d.
2. The source block n.
3. All nodes that can reach n without going through d.

Example:

B1

B2

Loop Header

B3

B4

B4 → B2 (Back Edge)

Natural Loop: B2 → B3 → B4 → B2

Here:

- B4 → B2 is the back edge.
- B2 is the loop header.

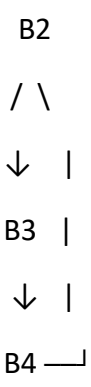
- B2, B3 and B4 form the natural loop.

5.4.3 Loop Header

The Loop Header is the block where the loop begins.

It is the destination of the back edge.

Example:



Here:

- B4 → B2 is the back edge.
- B2 is the loop header.

The loop header usually contains the condition that controls whether the loop continues.

5.4.4 Properties of Loops in Flow Graphs

A loop generally has the following properties:

1. Entry Point

The point where control enters the loop.

2. Loop Body

The statements that are repeatedly executed.

3. Exit Condition

The condition that determines when the loop terminates.

4. Back Edge

The edge that transfers control back to the loop header.

5. Exit Edge

The edge through which control leaves the loop.

5.4.5 Loop Detection

The compiler can analyze a flow graph to identify loops.

General Process:

Flow Graph

Find Back Edges

Identify Loop Headers

Find Nodes Belonging to Each Loop

Construct Natural Loops

Apply Loop Optimizations

5.4.6 Types of Loops

There are three common types of loops.

A. Single Loop

A loop containing one loop body.

Example:

B1

↓

B2 ←—┐

↓ |

B3 —┘

Here, B2 and B3 form a single loop.

B. Nested Loops

A loop that contains another loop inside it is called a Nested Loop.

Example:

```
for(i = 0; i < 5; i++)
{
   for(j = 0; j < 5; j++)
   {
      // statements
   }
```

}

Flow graph conceptually:

Outer Loop

|

↓

Inner Loop

|

↓

Inner Loop Repeats

|

↓

Outer Loop Repeats

The inner loop executes completely for each iteration of the outer loop.

C. Multiple / Sequential Loops

Two or more loops may occur one after another.

Example:

B1

↓

B2 ←┐

↓ |

B3 ─┘

↓

B4

↓

B5 ←┐

↓ |

B6 ─┘

Here, there are two separate loops that execute one after another.

5.4.7 Importance of Loops in Compiler Optimization

Loops are very important because their statements may execute many times.

Example:

```
for(i = 0; i < 1000000; i++)
{
   x = a + b;
}
```

If a + b does not change inside the loop, repeatedly calculating it is unnecessary.

The compiler can move the calculation outside the loop.

Before Optimization:

For each iteration:

```
   x = a + b
```

After Optimization:

```
t = a + b
```

For each iteration:

```
   x = t
```

This is an example of Loop-Invariant Code Motion.

Advantages of Loop Optimization

Loop optimization can:

1. Reduce execution time.
2. Reduce the number of instructions executed repeatedly.
3. Reduce loop-control overhead.
4. Improve processor utilization.
5. Improve overall program performance.