Unit-5.

Machine Independent Optimizations.

The principle sources of Optimization.

A Compiler Optimization must preserve the semantics of Original program.

Causes of Redundancy.

→ There are many redundant Operations in a typical program

→ The programmer may find it to be more Convenient to recalculate the result, and leaving it to the Compiler to recognize only One Calculation.

→ Some times redundancy is a side effect of writing the program in high level.

→ Compiler eliminate the redundancies, The programs becomes efficient and easy to maintain.

Semantic - preserving Transformation.

There are number of ways by using in which Compiler can improve a program without changing its functions. Such Techniques are Common-subexpression elimination, copy propagation, dead code elimination and Constant folding.

```
(1)   i = m-1
(2)   j = n
(3)   t1 = 4*n
(4)   v = a[t1]
(5)   i = i+1
(6)   t2 = 4*i
(7)   t3 = a[t2]
(8)   if t3<v goto (5)
(9)   j = j-1
(10)  t4 = 4*j
(11)  t5 = a[t4]
(12)  if t5>v goto (9)
(13)  if i>=j goto (23)
(14)  t6 = 4*i
(15)  x = a[t6]
(16)  t7 = 4*i
(17)  t8 = 4*j
(18)  t9 = a[t8]
(19)  a[t7] = t9
(20)  t10 = 4*j
(21)  a[t10] = x
(22)  goto (5)
(23)  t11 = 4*i
(24)  x  = a[t11]
(25)  t12 = 4*i
(26)  t13 = 4*n
(27)  t14 = a[t13]
(28)  a[t12] = t14
(29)  t15 = 4*n
(30)  a[t15] = x
```

Figure 9.2: Three-address code for fragment in Fig. 9.1

```
t6 = 4*i
x = a[t6]
```

as shown in steps (14) and (15) of Fig. 9.2. Similarly, `a[j] = x` becomes

```
t10 = 4*j
a[t10] = x
```

in steps (20) and (21). Notice that every array access in the original program translates into a pair of steps, consisting of a multiplication and an array-subscripting operation. As a result, this short program fragment translates into a rather long sequence of three-address operations.

Figure 9.3 is the flow graph for the program in Fig. 9.2. Block B_1 is the entry node. All conditional and unconditional jumps to statements in Fig. 9.2 have been replaced in Fig. 9.3 by jumps to the block of which the statements are leaders, as in Section 8.4. In Fig. 9.3, there are three loops. Blocks B_2 and B_3 are loops by themselves. Blocks B_2, B_3, B_4, and B_5 together form a loop with B_2 the only entry point.

9.1.3 Semantics-Preserving Transformations

There are a number of ways in which a compiler can improve a program without changing the function it computes. Common-subexpression elimination, copy propagation, dead-code elimination, and constant folding are common examples of such function-preserving (or *semantics-preserving*) transformations: we shall consider each in turn.

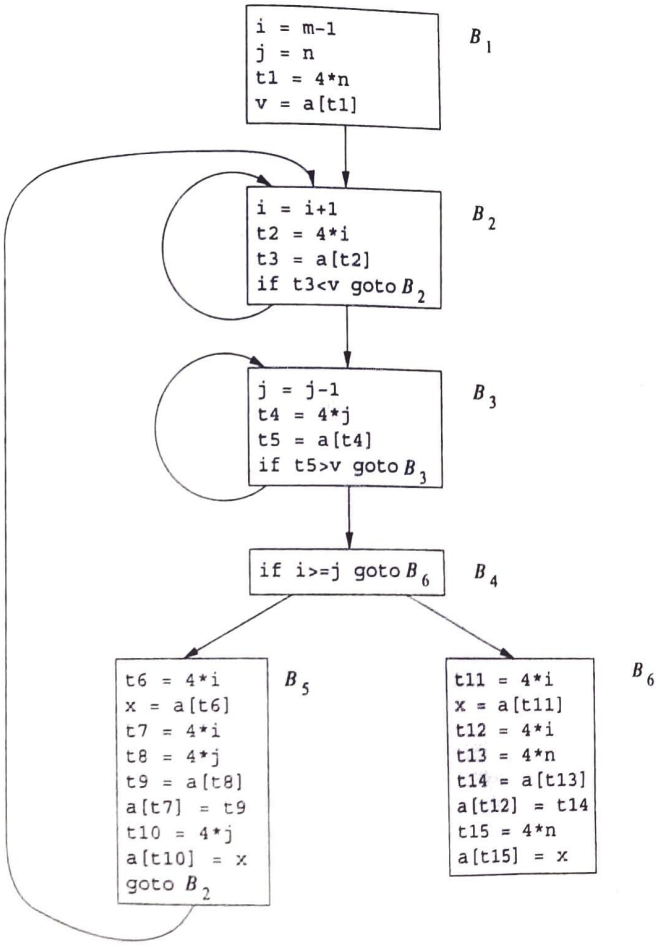
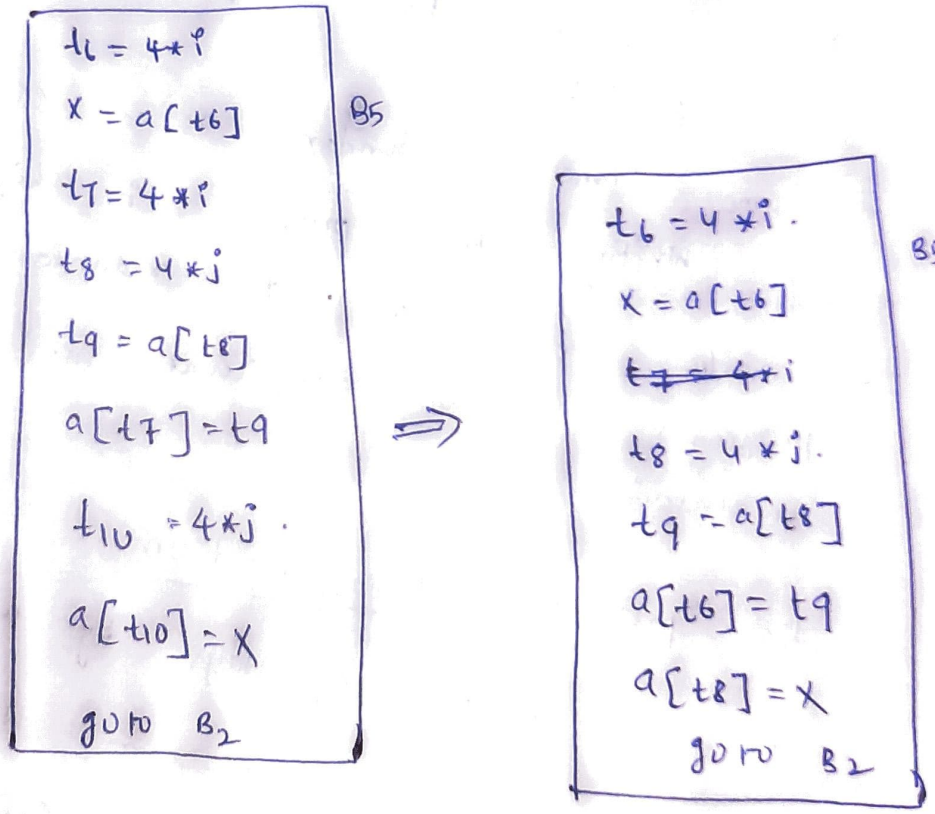


Figure 9.3: Flow graph for the quicksort fragment

Global Common Sub Expressions

An Occurrence of E is called as Common Sub Expression if E was previously computed and values of variables are not changed since the previous computation. We can avoid recomputing of E, if we can use its previously computed value.

Following block has assignments to t7 and t10 for computing 4*i and 4*j. These steps have been eliminated in next figure. Which uses t6 instead of t7 and t8 instead of t10.



After Common subexpressions are eliminated, B5 still evaluates 4*i and 4*j, which are already evaluated in B2 and B3.

in B3, $t_4 = 4 * j$, so 3 statements

in B5

$t_8 = 4 * j$ // replaced by t_4.

$a[t_4] = a[t_8]$

$a[t_8] = X$

Becomes after replacing t_8, which is redundant.

$t_4 = a[t_4]$ } can be written as $a[t_4] = X$

$a[t_4] = X$

Similary in B5 B2, $t_2 = 4 * i$, so in B5 following statements

$t_6 = 4 * i$ // replaced by t_2

$X = a[t_6]$

becomes $X = a[t_2]$. and in B2 $t_3 = a[t_2]$

so $X = t_3 \rightarrow (i)$

Finally B5 becomes

$X = t_3$.

$a[t_2] = t_5$

$a[t_4] = X$

goto B_2

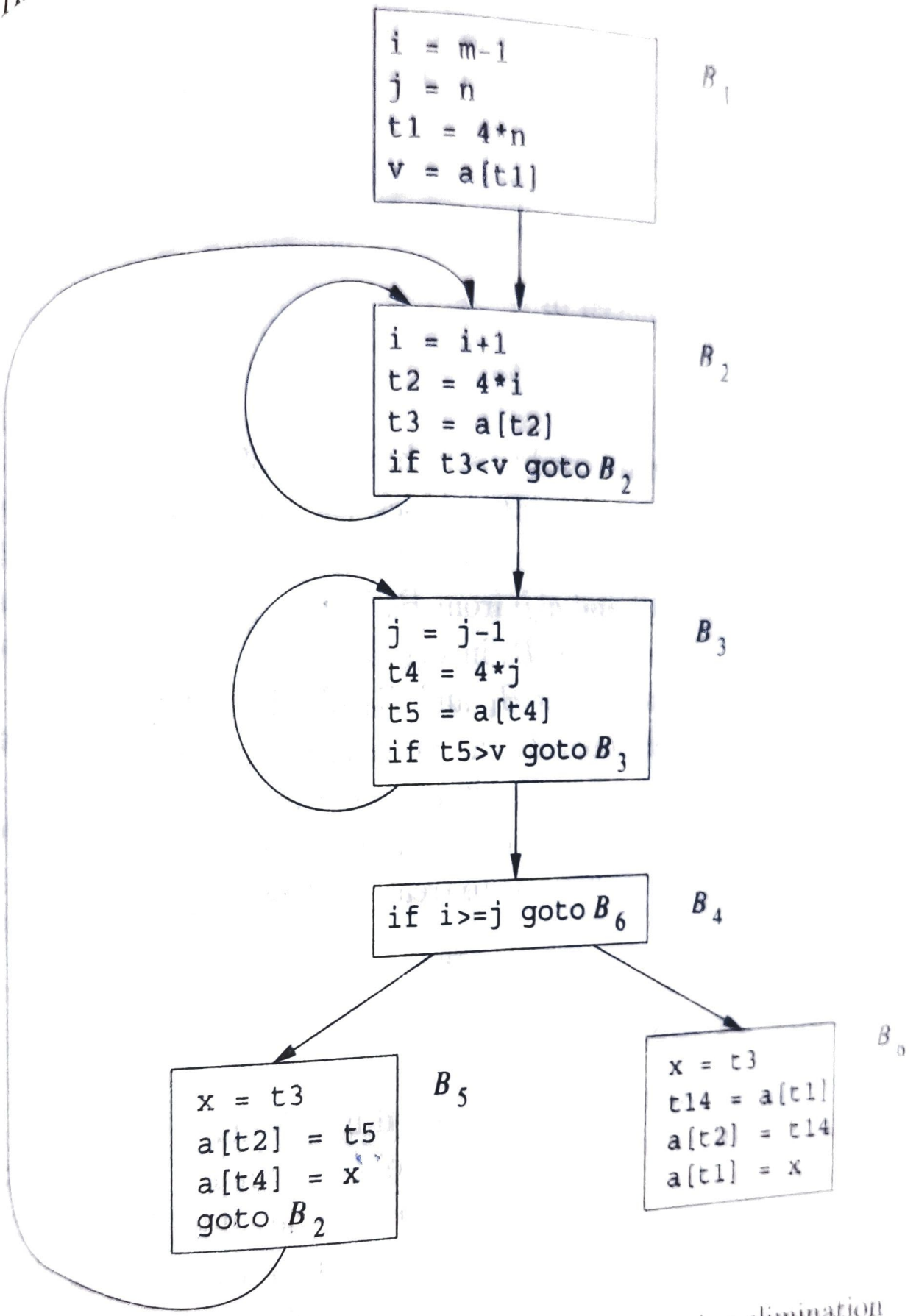


Figure 9.5: B_5 and B_6 after common-subexpression elimination

Copy Propagation.

Block B5 can be further improved by eliminating x, using two transformations. One concerns assignment of the form u = v called as Copy statement.

The idea behind copy propagation is to use v for u, whenever possible copy statement u=v

For ex, in B5 assignment x=t3 is a copy.

Copy propagation yields following code.

```
x = t3.
a[t2] = t5
a[t4] = t3
goto B2
```

Dead code elimination

A variable is live, if its value can be used, otherwise it is dead.

A dead code or unused code, is the statements which computed the values that can never be used

For ex :- Consider the following statement

```
if (debug)
print.
....
```

If debug is set FALSE, The print statement is dead, because it cannot be reached. We can eliminate both test and print statements from the Object code.

One Advantage of copy propagation is it turns copy statement to Dead code. For eny copy propagation followed by dead code elimination removes x in B5, and generate following code.

```
a[t2] = t5
a[t4] = t3
goto B2.
```

Code Motion.

Loops are very important place for optimization. Programs spends bulk of their time in Executing inner loops. If we decrease number of statements in inner loop, and even we increase number of statements out of loop The running time of a program may be improved.

Code motion is a technique which reduces the number of statements in a loop. If an expression yields same result as allways in each iteration known as loop-in variant- computation, and evaluates the expression before loop. By code Motion such expression can be evaluated out of loops only once

Ex:- Consider The statement

```
While ( i<= limit-2)   // Stmt does not change limit.
```

Code motion result in following code.

```
t = limit-2
while ( i<= t)
```

Now limit-2 is performed only once. previously there were n+1 calculations if loop executes n times.

Induction Variables and Reduction in Strength

A variable x is said to be induction variable, if there is a positive or negative constant c, such that each time x is assigned its value is increased by c. For ex. i and t_2 are induction variables in the loop in B2

The transformation of replacing an expensive operation, such as multiplication by a cheaper one such as Addition is known as Strength reduction.

For ex, in B2

$$i = i + 1$$

$$t_2 = 4 * i$$

Can be reduced to

$$t_2 = t_2 + 4.$$

Constant Folding.

If the value of an expression is always constant, using that constant instead of expression is known as Constant folding. This occurs at Compile time.

Introduction to Data Flow Analysis.

→ Data flow analysis refers to a body of techniques that derive info about flow of data along program execution paths.

→ The Data-Flow Abstraction.

→ The execution of a program can be viewed as a transformations of program state, which consists of values of all variables.

→ Each Execution of an intermediate-code statement transforms input state to a new output state.

→ The i/p state is associated with program point before the statement and output state is associated with program point After statement.

→ The sequence of program points (points) through a flow graph will be created as program executes.

→ Within Each basic block, The program point after a statement is same as program point before the next statement.

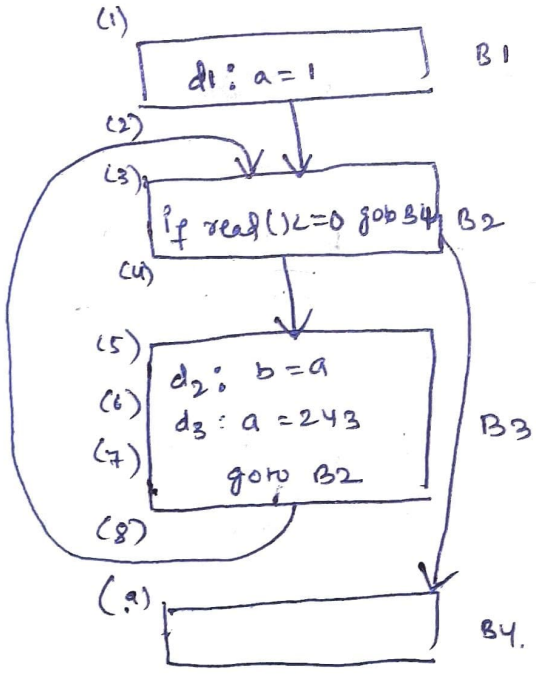
→ if There is an edge from Block B1 to B2 Then program point after the last statement in B1 may be immediately followed by ~~before~~ program point

Before The first statement of B2.

The execution path (simply path) from point P1 to Pn is a sequence of points P1, P2 .. Pn. such that.

1. Pi is the point immediately preceding a statement and Pi+1 is the point immediately following some statement.
2. if Pi is end of some block, then Pi+1 is beginning of Successor block.

Following figure shows number of execution paths.



→ The path NOT entering the loop in B3 has the shortest path consisting of Program points 1, 2, 3, 4, 9

→ The path existing one ~~instruction~~ iteration of the loop has the points 1, 2, 3, 4, 5, 6, 7, 8, 3, 4, 9

The Data Flow Analysis Schema.

In Data Flow analysis, we associate each program point a data flow value, which represents sets of all program states. The set of all possible data flow values is the domain for this application. Data flow values are denoted before and after each statement IN[S] and OUT[S]. The data flow problem finds solutions to a set of constraints on the IN[S]'s and OUT[S]'s. There are two sets of constraints Transfer functions and control flow constraints.

Transfer function.

The Data flow values before and after a statement are constrained by semantics of statement.

→ if a variable a has value v, before executing b = a, Then both b and a will have value v after the statement. This relationship b/w data flow values before and after the Assignment statement are known as transfer function.

Transfer functions are of two types.

1. Forward flow problem.

info propagate forward along the execution path. Transfer fn of a statement s is denoted by f_s, takes data flow value before the statement and produces new data flow value after the statement, as follow.

$$OUT[s] = f_s(IN[s])$$

2. Backward - flow problem → Transfer function f_s, converts a data flow value after the statement to a new data flow value before the statement. as follows

$$IN[s] = f_s(OUT[s]).$$

Control flow Constraints.

Within a Block control flow is simple

If a Block B consists of statements $S_1, S_2; S_n$,

Then control flow value out of S_i is same as control flow value into S_{i+1}. That is

$$IN[S_{i+1}] = OUT[S_i]$$

Reaching Definitions.

Reaching Definitions is one of the most Common data flow schemas.

A definition d, reaches a point P, if there is a path from d to P, such that d is not killed along the path.

A definition will be killed for a variable x, if there is any other definition of x along the path. When a definition of d, reaches a point P, x is used at P.

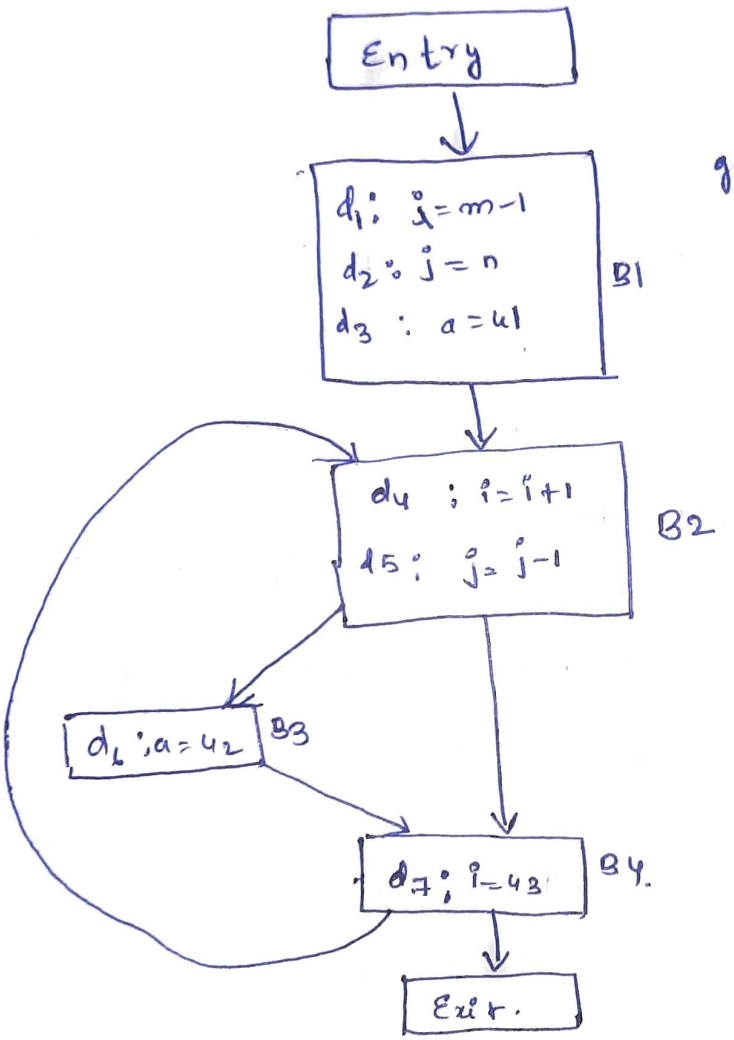
Example :- Following is a flow graph with seven definitions. In B_2, all definition in B1, reaches ~~B2~~ begining of B2.

→ The definition d5 : j = j-1 in block B2 also reaches the begining of B2, Because no other definition of j can be found on the loop leading Back to B2.

This definition kills definition $d_2: j = n$, and prevents it from reaching B3 and B4.

→ The statement $d_4: i = i+1$, does not reach the beginning of B2 in loop, Because i is redifined by $d_7: i = 43$

→ The definition $d_6: a = 4i$ also reaches beginning of B2. But kill definition d_3 in B1



gen $B_1 = \{d_1, d_2, d_3\}$

kill $_{B_1} = \{d_4, d_5, d_6, d_7\}$

gen $B_2 = \{d_4, d_5\}$

kill $_{B_2} = \{d_1, d_2, d_7\}$

gen. $B_3 = \{d_6\}$

kill $B_3 = \{d_3\}$

gen $B_4 = \{d_7\}$

kill $B_4 = \{d_1, d_4\}$

Flow-Graph reaching Definitions.

Foundations of Data-Flow Analysis.

A Data Flow Analysis Framework $(D, V, \wedge, F)$ Consists of

1. Direction of data Flow D, which is either Forwards or backwards.
2. A semilattice, which includes a domain values V and a meet Operator, $\wedge$.
3. Transfer Function F, V to V

Semilattices.

Semilattice is a set V and a binary meet Operator $\wedge$, such that for all x, y, z in V

1. $x \wedge x = x$ (meet is idempotent).
2. $x \wedge y = y \wedge x$ (meet is Commutative).
3. $x \wedge (y \wedge z) = (x \wedge y) \wedge z$ (meet is associative)

Semi lattice has a Top element T, such that.

for all x in V.

$$T \wedge x = x.$$

semi lattice may have bottom element denoted by $\perp$, such that

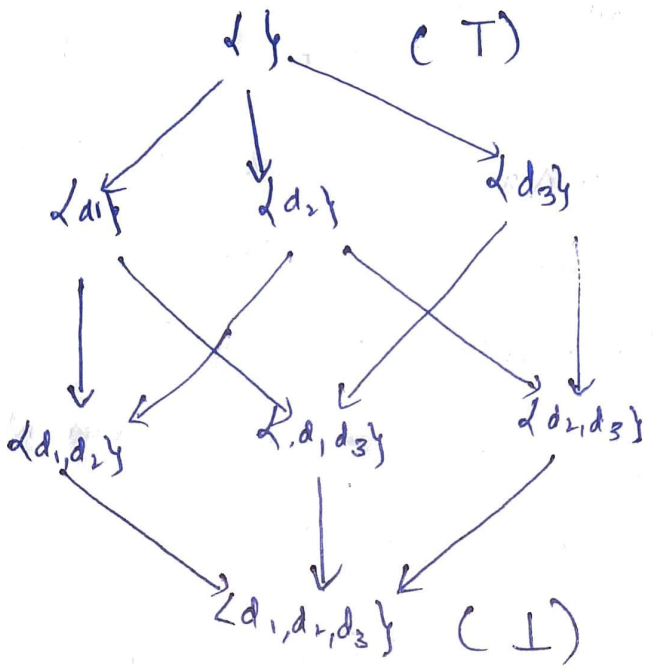
for all x in V,

$$\perp \wedge x = \perp$$

Lattice Diagrams.

We can Draw a Domain V, as a lattice diagram, which is a graph whose nodes are elements of V, and edges are directed downward, from x to y, if $y \leq x$.

Following figure shows Lattice Diagram for 3 definitions d_1, d_2, d_3, since $\leq$ is a $\supseteq$, an edge is directed from subset to its superset



Constant Propagation

Constant propagation or Constant folding replaces expressions that evaluate the same constant every time they are executed, by constant.

Constant propagation framework is as follows.

1. it has unbounded set of possible data values even for a fixed flow graph.
2. It is not distributive.

Constant propagation is a forward data flow problem.

Data-Flow values for the Constant-propagation framework.

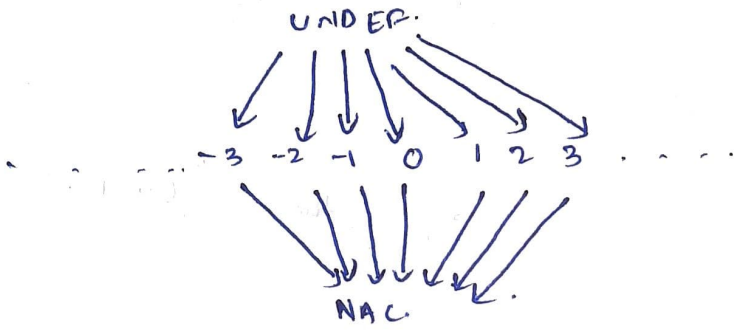
The set of data flow values in a product lattice consists of the following.

1. All Constants for the type of variable.
2. The Value NAC (Not a Constant). A Value will be NAC, if it is not a constant. A variable may have been assigned a value, or derived other variable which is not Constant
3. The Value UNDEF, stands for undefined A Variable is UNDEF if nothing can be asserted.

→ NAC and UNDEF are not same. They are Opposites. NAC says we have so many values for a variable, so it is not a constant.

→ UNDEF says we have seen so little values, so we can not say any thing about constant.

→ Following figure shows semilattice for a typical integer-valued variable. Top element is UNDEF, and bottom is NAC. The constant values are unordered, They are less than UNDEF and greater than NAC.



The meet of two values is their greatest lower Bound

$$UNDEF \wedge V = V$$

$$NAC \wedge V = NAC.$$

For any Constant

$$C \wedge C = C$$

given two distinct constant.

$$C_1 \wedge C_2 = NAC.$$

Transfer function for Constant Propagation framework.

→ Transfer functions set F accepts a map of variables to values in constant lattice and returns one these such map.

→ F contains a identity function, which takes a map as input and returns same map as O/P.

→ F contains a Transfer function, given any input map, returns m_0, where $m_0(V) = UNDEF$. for all variables V.

In general f_s be the transfer function for statement S, and let m and m' represents data flow values such that $m' = f_s(m)$

1. if f_s is not an assignment stmt, Then f_s is simply identity function.

2. if f_s is an assignment statement to variable x, Then $m'(v) = m(v)$, for all variables $v \neq x$, $m'(x)$ can be defined as

a) if RHS of the statement is constant $m'(x) = C$.

b) if RHS is of the form $y+z$ then

$$m'(x) = \begin{cases} m(y) + m(z) & \text{if } m(y), m(z) \text{ are constants.} \\ NAC & \text{if either } m(y) \text{ or } m(z) \text{ is NAC.} \\ UNDEF & \text{Otherwise.} \end{cases}$$

(3) if RHS is any other expression

then $m'(x) = NAC$.

Monotonicity of Constant-Propagation Framework

Constant propagation Frame work is monotone.

For $z = y + z$, fs for 2(b) is tabulated below.

The first and second columns represents possible values of y and z. The last represent O/p value of x.

The values are ordered Greatest to lowest in each column. The function is monotone because for each value of y value of x does not get bigger as the value of z is smaller. For exy, In case y has constant value C1, value of z varies from UNDEF to C2 to NAC The value of X varies from UNDEF to C1+C2 to NAC.

m(y)	m(z)	m'(x)
UNDEF	UNDEF	UNDEF.
	C2	UNDEF.
	NAC.	NAC.
C1	UNDEF.	UNDEF.
	C2	C1+C2
	NAC	NAC.
NAC	UNDEF.	NAC
	C2	NAC
	NAC.	NAC.

Partial Redundancy Elimination.

We can minimize the number of expressions evaluations for ex", in a flow graph, There is an expression x+y is evaluated number of times. We can reduce the evaluation by keeping the result of x+y in one temporary variable, and using this along execution path whenever needed.

Redundancy in program exists in several forms. It may exist in the form of Common subexpressions, where several expressions produces same value. Redundancy may exist in the form of loop-invariant ~~variable~~ expression, which evaluates same value in each iteration Redundancy can also be partial, if it exists on some path but not on all paths of execution. Common subexpression and loop-invariant expressions are examples of partial Redundancy.

The Sources of Redundancy.

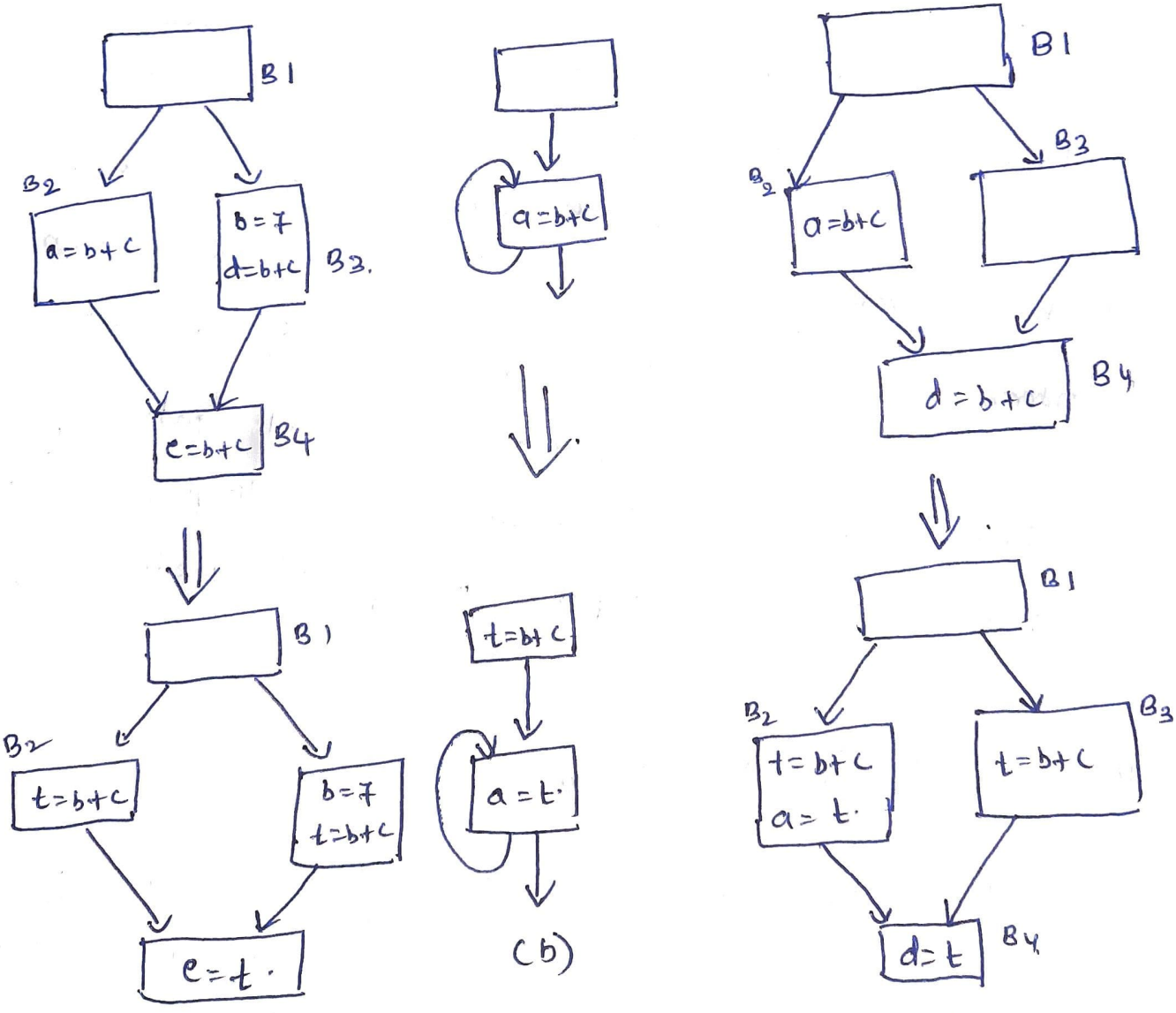
Following figure shows Three forms of redundancy : Common subExpressions, loop invariant expressions and Partially ~~redundt~~ redundant expressions. Figure also shows

the code before and after each Optimization.

Loo

Global Common Sub Expression.

In the below Figure(a) b+c computed in Block B4, which is redundant because b+c is already evaluated in B2 and B4. We can Optimize the code by storing the result of b+c in Block 2 and 3, in a variable t, and then assigning. Value of t to variable e in Block B4. instead of re evaluating the expression.



Loop - Invariant Expressions.

Figure (b). shows an example of loop-invariant expression. The expression b+c is loop invariant assuming b and c are not redifined within loop.

We can Optimize the program by replacing all re-exiations in a loop by a single calculation outside the loop.

We assign the Computation to a temporary variable say t, and then replace the expression in loop by t. This is known as code motion.

Partially - Redundant Expressions.

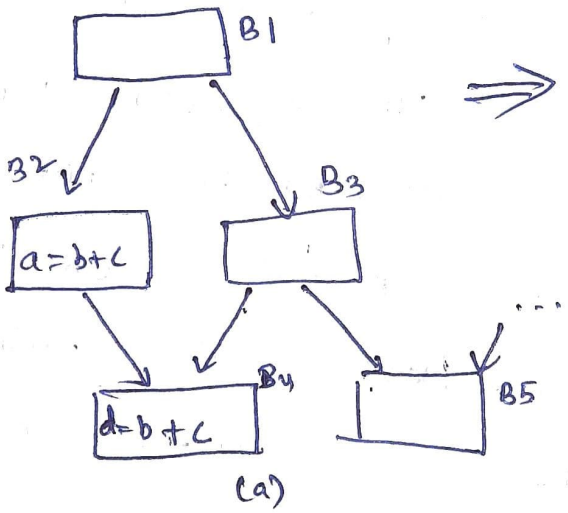
An Ex. of partially Redundant expression is shown in figure (c). The expression b+c is redundant on path B1→B2→B4, but not on path B1→B3→B4. This partial Redundancy on second path can be eliminated by placing b+c ~~on~~ in ~~second~~ B3. All the results of b+c are written in t, and Calculation of b+c is replaced by t. Partial redundancy can be eliminated by new expression Compdations.

Can All Redundancy be eliminated.

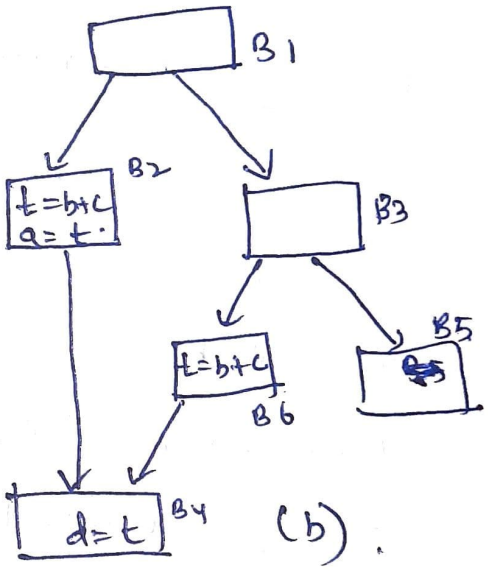
The Redundancy can not be eliminated Completely ~~until~~ unless new blocks are added to flow graphs.

Ex:- In below flow graph (a), The expression b+c is redundantly in B4, along the path B1→B2→B4. To eliminate partial Redundancy We Can not move Computation ~~to~~ b+c on B3, which creates extra Computation b+c, when path B1→B3→B5 is taken.

The Solution is to is to insert Computation of b+c only along edge B3 to B4. This can be done by placing a new Block B6, and making flow control to go through B6 from B3 Before Reaching B4.



(a)



(b).

Critical edge of a flow graph, is any edge leading from a node with more than one Successor and to a node With more than one predecessor. The edge from B3-to-B4, is critical Because B3 has TWO Successors, and B4 has two Predecessors.

Lazy Code Motion

Values of expressions which are redundant are held in registers. If these values are computed as late as possible minimizes its life time. The duration between the value is defined and the time it is used. This minimizes the usage of Registers.

The Optimization of eliminating partial Redundancy with goal of delaying Computations as much as possible is known as "Lazy Code motion"

Full Redundancy.

An expr e in Block B is fully Redundant if along all paths reaching B, e has been evaluated and Operands of e have not been redefined. For ex, Let S be set of Blocks, each contains e, Then it is known as Full Redundancy.

Partial Redundancy.

The expression e is defined in only few Blocks. Then it is partial Redundancy. Lazy code motion attempts to render in e in all Blocks to make it fully Redendent.

Loops in Flow Graph.

loops are important because programs spends most of the time in executing loops, and optimizations to improve performance of loops.

Loops affect the running time of program analysis. If the program does not have loops we can identify data flow analysis by simply making one pass through the program. Various concepts are used in finding iterative data flow analysis like dominators, depth-first-Ordering, back edges, graph depth and reducibility.

1. Dominators.

A node d in flow graph dominates node n, written as d dom n, if every path from entry node to n goes through d. Every node dominates itself.

Ex:- Consider the following flow graph, with entry node 1. The entry node dominates every node. Node 2 dominates only itself, since control can reach any other node along path 1-3.

→ Node 3 dominate all but 1 and 2.

→ Node 4 dominates all but 1, 2 and 3

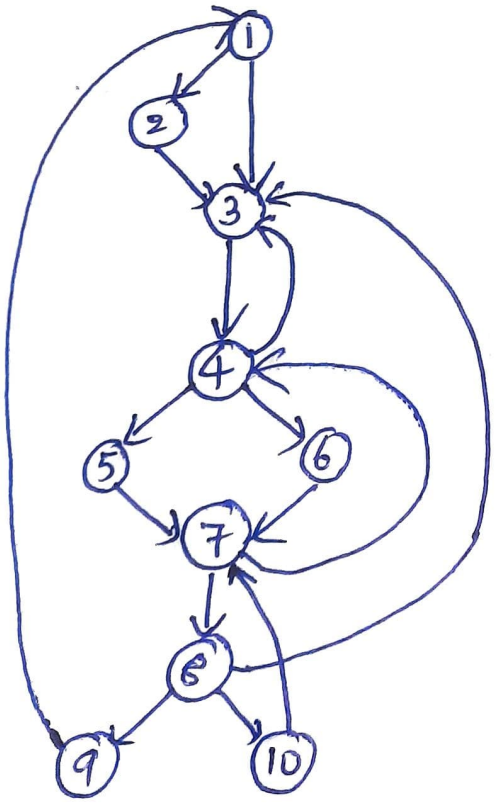
→ Node 5 and 6 dominate itself. Bcoz Because control can skip around either by going through Other.

→ Node 7 dominates 7, 8, 9 and 10.

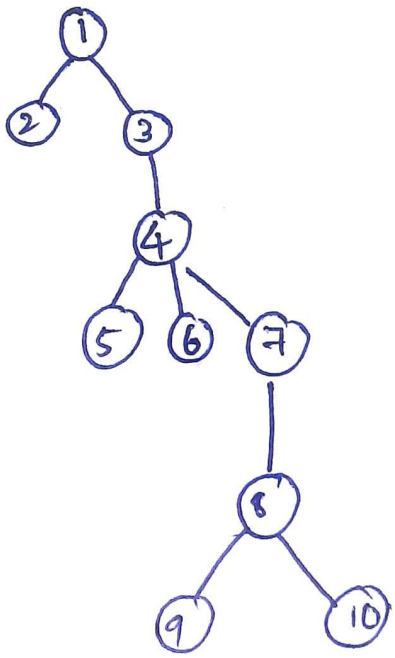
→ Node 8 dominates 8, 9, and 10.

→ Node 9 and 10 dominate only themselves.

Dominator info can be represented by Dominator tree, in which the entry node is root, each node dominates only its descendants. in the Tree. Below Fig (a) show flow graph, and fig (b) is dominator Tree.



(a). Flow Graph.



(b). Dominator Tree.

Depth First Ordering.

Depth first search of a graph visits all the nodes in the graph once, by starting at entry node and visiting the nodes as far away from entry node as quickly as possible.

The pre Order Traversal visits a node Then its children from left to right. Post Order Traversal visits the children then the node in left to right order.

Depth first ordering is reverse of post Order Traversal. In Depth first ordering we visit a node, and then its Children from right to left.

Ex:- One possible depth first presentation of flow graph is shown below. The depth first traversal of flow graph is given by

1 → 3 → 4 → 6 → 7 → 8 → 10 → 8 → 9 → 8 → 7 → 6 → 4 → 5 → 4 → 3 → 1 → 2

The pre order Traversal is

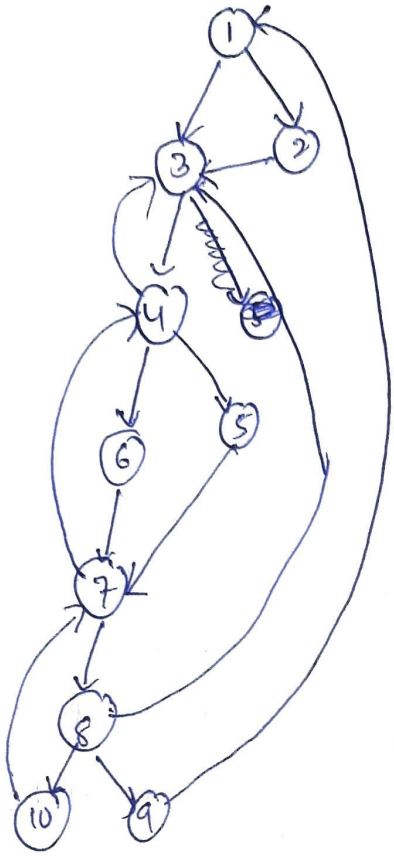
1, 3, 4, 6, 7, 8, 10, 9, 5, 2.

post order Traversal is

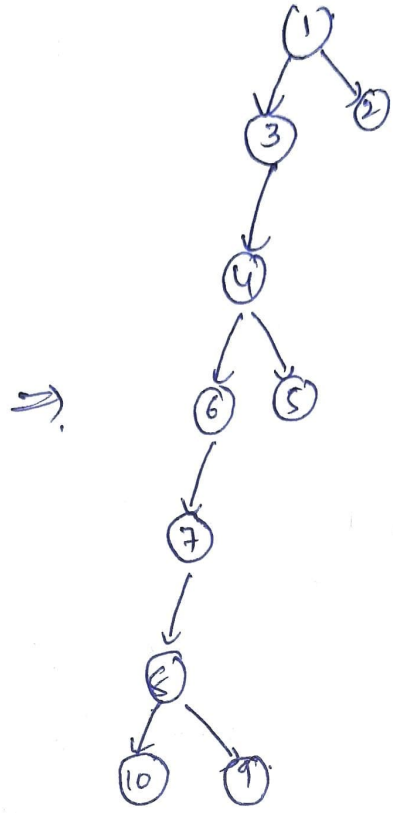
10, 9, 8, 7, 6, 5, 4, 3, 2, 1

Depth first order is reverse of post-order sequence, given by.

1, 2, 3, 4, 5, 6, 7, 8, 9, 10.



Flow Graph.



Depth first order representation.

Edges in Depth-first Spanning Tree (DFST)

The Edges of DFST, for a flow graph are of 3 types.

1. advancing edges. → go from a node m to its proper descendants. like 1→2, 1→3, 3→4, 4→5, 4→6, 6→7, 7→8, 8→10, 8→9.

2. Retreating edges. → The edges which goes from m to its ancestors. Ex:- 4→3, 7→4, 10→7, 8→3, and 9→1.

3. Cross Edges. The edge $m \to n$, in which neither m nor n is an ancestor of other. For ex. edges $2 \to 3$ and $5 \to 7$.

Depth of a Flow Graph.

Depth of a flow graph is largest number of retreating edges on any cycle free path. For ex, in previous Figure, The longest path of retreating edges is

$10 \to 7 \to 4 \to 3$, and depth is 3.

Natural loops.

Loops can be specified in source program in many ways, like for loops, While loops, and loops can also be defined by using labels and gotos. Statements. A natural loop is defined by TWO essential properties.

1. it must have single-entry node, called as header This node dominates all nodes in loops;
2. There must be back edge that enters The loop. Otherwise it is not possible for flow of control to return to header, i.e; There is no loop.

Ex:- In previous flow graph There are five back edges $10 \to 7$, $7 \to 4$, $4 \to 3$, $8 \to 3$ and $9 \to 1$

→ Back edge $10 \to 7$ has ~~loop~~ natural loop of {7,8,10}.

→ Back edge $7 \to 4$ has natural loop of {4,5,6,7,8,10}

Therefore it contains the loop of $10 \to 7$

{7,8,10} is inner loop of {4,5,6,7,8,10}

→ The edges 4→3, 8→3 has same header node 3, and same set of nodes {3,4,5,6,7,8,10}

These two loops can be combined as one.

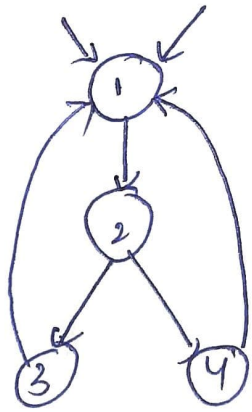
This loop contains two smaller loops earlier discussed.

→ Finally edge 9→1, has its natural loop, the entire flow graph and hence outer most loop.

The four loops are nested within one another.

When two natural loops have the same header, and neither is properly contained within one another, these can be combined and treated as one loop.

For ex:- in below loop, the back edges 3→1 and 4→1, and the natural loops are {1,2,3} and {1,2,4}, we can combine them into single loop {1,2,3,4}



Two loops with same header.

If loop contains another back edge $2 \rightarrow 1$, the loop would be $\{1,2\}$, which is properly contained within $\{1,2,3,4\}$, so it can not be combined with natural loops, but treated as nested loop.