5.1 Introduction

Code optimization is an optional phase in which the compiler tries to improve the intermediate code so as to get better performance in terms of execution time or storage space. So, when do we optimize the code?

a. Whether code optimization improve codes substantially in comparison to the complexity of the optimization process?

b. Does it preserve the meaning of the source program?

c. Whether the gain is significant w.r.to the execution time or space?

Example: The complexity of algorithm is not or cannot be improved by the code optimization i.e. Bubble sort algorithm cannot be converted to the Quick sort with the help of code optimization.

5.2 Sources of Code Optimization

One question can be raised like this: how do we achieve the code optimization?

→ The optimization can be better achieved by **efficient use of Registers** and **addressing formats** during object code generation.

→ **Loop Optimization**: The optimization should focus on the innermost loop to improve the code as this part of the program is most often executed [90 - 10 Rule].

→ **Code Motion**: It deals with the movement of the code from one place to (generally inside the loop) to another place (generally outside the loop).

Example:

```
for( i = 0; i <= 1000; i++){
    PI = 22.0/7.0;
    a = PI * r[i] * r[i];
    print(a);
}
```

To achieve this, first we need to:
(a) Identify the loop invariant statements.
(b) Move the loop invariant code to a place before the beginning of the loop.

→ **Common Sub-expression:**

Example: Consider the following expressions

```
X = A + B + C,
Y = A + B + D and
Z = (A + B) * E.
```

We can represent the expression in quadruple like this:

$T_1 = A + B$
$T_2 = T_1 + C$
$X = T_2$
$T_3 = A + B$
$T_4 = T_3 + D$

```
Y     = T4
T5    = A + B
T6    = T5 * E
Z     = T6
```

To achieve this, we need to:
(a) Identify the common sub-expressions from the source code.
(b) Evaluate it as a temporary variable.
(c) Use the temporary variable instead of the common sub- expressions.

→ Elimination of Induction Variable:

A set of variables are said to be induction variable if a change in one place in source code induces change in other place.

```
for( i = 0; i <= 1000; i++){
    x = i * i;
    y = i / 2.0;
    z = x * 3;
}
```

To achieve this, we need to:
(a) Identify the both dependent and independent variables from the source code.
(b) Eliminate the dependent induction variables by replacing them with the independent one.

→ Reduction of Strength:

Some computations are stronger i.e. takes more time or space than other similar computations.

Example: '*' operator is stronger than '+'.

'^' operator is stronger than '*'.
'*' operator is stronger than '<<'.

So,

```
pow(X, 2)  ⇒ X * X
2 * A      ⇒ A + A
```

```
Str1 = "ABC"
Str2 = "XYZ"
```

To find the length of Str_1 and Str_2 together, we can write

```
(a) strlen(Str1) + strlen(Str2)
(b) strlen(strcat(Str1, Str2))
```

Looking at (a) and (b), we need to compare the strength of the functions `strlen` and `strcat` and the less strength function should be used to get optimized code.

→ Constant Folding:

Replacing a variable with a constant whenever possible.

Example:

If we have an expression `X = A + B` and `A = 5` then we can get optimized code by replacing the value of `A` by its constant value.

5.3 Loop Identification

Now, How do we optimize the loop in a source code?

Example: Some statements in Intermediate representation (Quadruple).

```
 1:  T1         = 0
 2:  T2         = 0
 3:  I          = 1
 4:  T1         = 4 * I
 5:  T2         = A - 4
 6:  T3         = B - 4
 7:  T4         = T1 + T2
 8:  T5         = T2 + T3
 9:  T6         = T4 + T5
10:  Y          = T6
11:  I          = I + 1
12:  I          < 20 (4)
13:  K          = 0
14:  K          = K + 4
15:  K          <= 20 (14)
16:  .
     .
     .
```

First we need to identify the Leaders of the Basic Block (BB) from which execution starts. Leaders helps to construct the Basic Blocks (BBs) and using the BBs we can create the flow graph. Flow graphs determines which blocks are in loop in the source program.

A. Leader Identification:

Step – 1. Mark the first statement as a Leader.

Step – 2. Mark the statement that follows a conditional jump as a Leader.

Step – 3. The target of any branch statement is a Leader.

B. Construction of Basic Block

A Basic Block (BB) is set of quadruples (or intermediate codes) of which all the quadruples are executed once when the control enters the block. The program control enters to the BB only through the leader and once entered all the statements are executed. So,

1. Construct the initial BB by including the first Leader and all the statements up to the next leader but not including the next Leader.
2. For each Leader, construct a BB by including all the statements up to the next leader as EOF but not including the next Leader itself.

C. Construction of Flow Graph

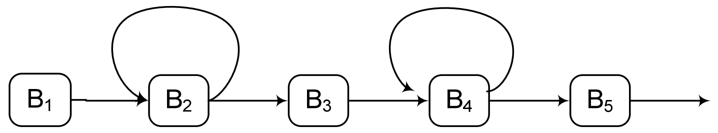
→ Create the nodes/vertices from the BBs i.e. nodes are BBs.

→ Establish an edge $B_i \rightarrow B_j$ *iff* B_i immediately precedes B_j in order of execution.

(i) Establish $B_i \rightarrow B_j$ *iff* B_j immediately follows B_i.

(ii) Establish $B_i \rightarrow B_j$ if the last statement of B_i jumps to the Leader of B_j.

Example:

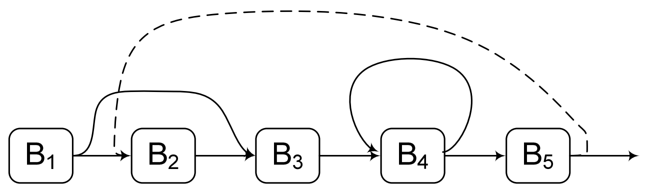


Now, we define a loop: a set of BBs are said to be in a loop iff:

(i) Strongly Connected. [i.e. Exists a path between any two nodes within the set only.]

(ii) There is only one entry and exit point.

Example:



This is not a loop because entry points are in B_2 and B_3.

5.4 Construction of DAG (Directed Acyclic Graph)

A DAG is constructed to have an equivalent graph for a Basic Block (BB), preferably for a loop. To construct a DAG, Let we have the quadruples of three types:

(a) A = B op C	i.e. `x = y + z`
(b) A = op B	i.e. `x = -y`
(c) A = B	i.e. `x = y`

Again, Let *node(x)* returns node *'n'* if formed otherwise return *false*.

<u>Step – 1:</u> If *node(B)* is not created.

→ Create a leaf node.

→ Label it with the name *'B'*.

Let *'n_1'* be the node formed or created.

In case of program statement type of (a), If *node (C)* is not created then,

→ Create a node.

→ Label it with the name *'C'*.

→ Let *'n_2'* be the node formed or created.

<u>Step – 2:</u>

1. In case of program statement type of (a): If there is no intermediate node labeled with *'op'* having left child *'n_1'* and right child *'n_2'*, then

→ Create a node.

→ Label it with *'op'*.

→ Set *'n_1'* as left child, *'n_2'* as right child.

Let *'n_3'* is the intermediate node formal or created.

2. In case of program statement type of (b): If there is no intermediate labeled with *'op'* and having only child *'n_1'*.

→ Create an intermediate node.

→ Label it with *'op'*.

→ Let *'n_1'* is the only child.

Let *'n_3'* be the intermediate node found or created on (1) and (2).

3. In case of program statement type of (c): $n_3 = n_1$.

<u>Step – 3:</u>

Add *'A'* to be the list of attached identifier for node *'n_3'*. If node(A) is found and *'n'* is the node.

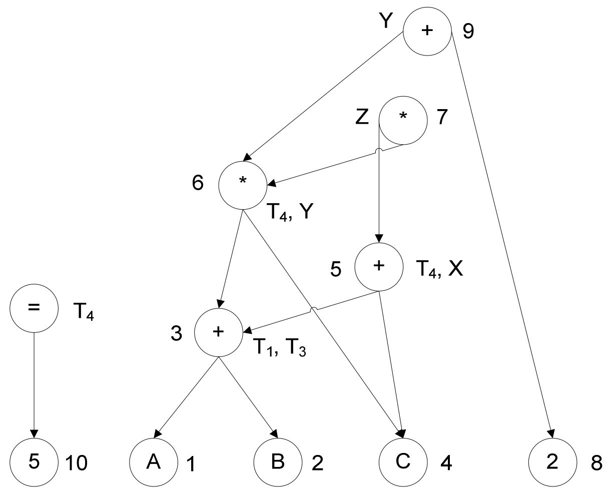
→ Remove *'A'* from the list of attached identifier of node *'n'*.

<u>Step – 4:</u>

Repeat *Step – 1, 2 & 3* for all the statements in the BBs.

Example:

```
T₁  = A + B
T₂  = T₁ + C
X   = T₂
T₃  = A + B
T₄  = T₃ * C
Y   = T₄
Z   = X * Y
Y   = Y + 2
T₄  = 5.
```



5.5 Use of DAG

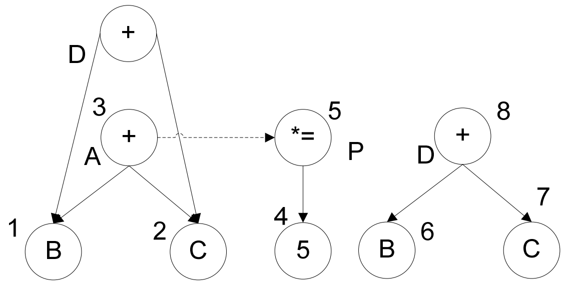
(a) To identify the common sub-expressions.
Example: {T_1, T_3}, {T_1, X}, {T_4, Y} etc.

(b) It shows the data items used in the BBs which may help in code motion.

(c) It shows the computations which are activate after the control exit from the particular BB.

Example: Extended DAG

```
A   = B + C
*P  = 5
D   = B + C
```



+
D
3
+
A
5
*=
P
8
+
D
7
1
B
2
C
4
5
B
6
C