RCOEM
Shri Ramdeobaba College of
Engineering and Management, Nagpur

Shri Ramdeobaba College of Engineering and Management

Compiler Design Lab (CSP327)
Mini project Report
Session 2024-2025

Course Coordinator: Prof. V. Bongirwar

By:

Name	Roll No.
Aboli Patne	01
Shamika Aney	08
Akshay Padia	23

Date: 24th April, 2025

GitHub Repository:
Link

Problem Definition

Background

Optimization is a crucial phase in compiler design, aiming to enhance the performance of the generated code. Among various techniques, Loop Invariant Computation is significant for improving runtime efficiency. It identifies computations within loops that remain unchanged across iterations and safely moves them outside the loop. This reduces redundant operations and boosts execution speed.

The Problem

Loops often contain computations that do not change with each iteration but are still redundantly executed, leading to inefficiencies. The challenge is to detect such loop-invariant statements and move them outside the loop without altering the program's behavior. This involves precise data flow analysis and understanding of control structures. The goal is to implement this optimization and visualize the transformation using graphical tools to clearly demonstrate the improvement.

Solution Description

The proposed solution aims to optimize loop performance in programs through loop-invariant computation using data flow analysis techniques. Our approach involves constructing a Program Flow Graph (PFG) from the given Three-Address Code (TAC), identifying leaders to define basic blocks, and computing GEN and KILL sets for each block. Using these, we perform iterative IN/OUT analysis to determine which expressions are loop-invariant and can be safely moved outside the loop. The system automates the detection of loop-invariant computations, thus enhancing the overall efficiency of the code while preserving its semantic correctness.

Objective

The objective of this implementation is to:

1. To identify loop-invariant computations within loops using data flow analysis.
2. To construct a Program Flow Graph (PFG) from the given Three-Address Code.
3. To compute GEN and KILL sets for each basic block in the control flow.
4. To apply IN/OUT analysis iteratively until convergence for accurate data flow tracking.
5. To determine the loop-invariant statements.
6. To visualize the process through graphical representation for better understanding.

Code Explanation with Input/Output

1. Finding Leaders

Leader statements are identified by the following rules:
Rule 1: The first instruction is considered the leader.
Rule 2: The target of conditional or unconditional goto is a leader.
Rule 3: The instruction that immediately follows a conditional goto is a leader.

```python
def find_leaders(statements):
    leaders = {1}

    for i, stmt in enumerate(statements, start=1):
        if "goto" in stmt:
            target = int(stmt.split("goto")[1].strip())
            leaders.add(target)

            if "if" in stmt.lower():
                if i + 1 <= len(statements):
```

```
                leaders.add(i + 1)

    return sorted(leaders)
```

2. Finding Blocks

A basic block consists of a leader and ends on the instruction just before the next leader, but not including the next leader. Any statement not placed in block can never be executed & may now be removed if desired.

```
def form_basic_blocks(statements, leaders):
    sorted_leaders = sorted(leaders)
    basic_blocks = []
    block = []
    current_leader_index = 0

    for i, stmt in enumerate(statements, start=1):
        if i in leaders:
            if block:
                basic_blocks.append(block)
            block = []
        block.append((i, stmt))

    if block:
        basic_blocks.append(block)

    stmt_to_block = {}
    for block_idx, block in enumerate(basic_blocks):
        for stmt in block:
            stmt_to_block[stmt[0]] = f"B{block_idx + 1}"

    return basic_blocks, stmt_to_block
```

3. Finding Dominators

Node ‘d’ dominates node ‘n’ if all paths from the entry node to ‘n’ go through ‘d’ (d dom n)

- Every node dominates itself.
- The initial node dominates all nodes in G.
- The entry node of a loop dominates all nodes in the loop.

```python
def find_dominators(basic_blocks, flow_graph):
    nodes = {f"B{i+1}" for i in range(len(basic_blocks))}
    dominators = {n: set(nodes) for n in nodes}
    dominators["B1"] = {"B1"}
    changed = True
    while changed:
        changed = False
        for node in nodes - {"B1"}:
            preds = [p for p in flow_graph if node in flow_graph[p]]
            new_doms = set(nodes)
            for pred in preds:
                new_doms.intersection_update(dominators[pred])
            new_doms.add(node)
            if dominators[node] != new_doms:
                dominators[node] = new_doms
                changed = True
    return dominators
```

4. Detecting Loop

- A loop is a cycle that satisfies two conditions, it has a single entry node and it is strongly connected.
- A loop exists in a program if and only if there are back edges in the program flow graph.
- To detect the back edge in the program flow graph, dominators must be found.
- The back edge is the edge in the program flow graph (PFG) where the head node dominates the tail node.

```python
def detect_loops(flow_graph, dominators):
    natural_loops = []
    for node, successors in flow_graph.items():
        for succ in successors:
            if succ in dominators[node]:
                natural_loops.append((node, succ))
    return natural_loops
```

5. Constructing Program Flow Graph (PFG)

Analyze the last statement of all the basic blocks.The flow can be directed to the next basic block or it may jump to another basic block if there is a conditional or unconditional goto statement.

```
def construct_flow_graph(basic_blocks, stmt_to_block):
    flow_graph = {f"B{i + 1}": set() for i in range(len(basic_blocks))}

    for i, block in enumerate(basic_blocks):
        block_name = f"B{i + 1}"
        last_stmt_num, last_stmt = block[-1]

        if "goto" in last_stmt:
            target_stmt = int(last_stmt.split("goto")[1].strip())
            if target_stmt in stmt_to_block:
                flow_graph[block_name].add(stmt_to_block[target_stmt])

        if "if" in last_stmt and "goto" in last_stmt:
            target_stmt = int(last_stmt.split("goto")[1].strip())
            if target_stmt in stmt_to_block:
                flow_graph[block_name].add(stmt_to_block[target_stmt])

            if i + 1 < len(basic_blocks):
                flow_graph[block_name].add(f"B{i + 2}")

        elif "goto" not in last_stmt and i + 1 < len(basic_blocks):
            flow_graph[block_name].add(f"B{i + 2}")

    return {k: sorted(list(v)) for k, v in flow_graph.items()}
```

```
def visualize_flow_graph(flow_graph):
    G = nx.DiGraph()
    for node, successors in flow_graph.items():
        for succ in successors:
            G.add_edge(node, succ)
```

```python
    plt.figure(figsize=(8, 6))
    pos = nx.spring_layout(G, seed=42)
    nx.draw(G, pos, with_labels=True, node_color="lightblue",
edge_color="black", node_size=2000, font_size=12, font_weight="bold")
    plt.title("Program Flow Graph")
    plt.show()
```

6. Computing GEN and KILL

GEN (B): Set of all definitions generated in Block B.
KILL (B): Set of all definitions outside Block B that define the same variables which are defined in Block B.

```python
def compute_gen_kill(basic_blocks):
    gen = {}
    kill = {}
    defined_vars = {}

    for block_idx, block in enumerate(basic_blocks):
        block_name = f"B{block_idx + 1}"
        gen[block_name] = set()
        kill[block_name] = set()

        for stmt_num, stmt in block:
            if "=" in stmt and "goto" not in stmt:
                var_name = stmt.split("=")[0].strip()
                gen[block_name].add(stmt_num)

                if var_name in defined_vars:
                    defined_vars[var_name].append((block_name, stmt_num))
                else:
                    defined_vars[var_name] = [(block_name, stmt_num)]

    for block_name in gen:
        kill[block_name] = set()
        for var, defs in defined_vars.items():
            defs_in_other_blocks = [stmt_num for b, stmt_num in defs if b
!= block_name]
            if any(stmt_num in gen[block_name] for _, stmt_num in defs):
                kill[block_name].update(defs_in_other_blocks)

    return gen, kill
```

7. Computing Predecessors

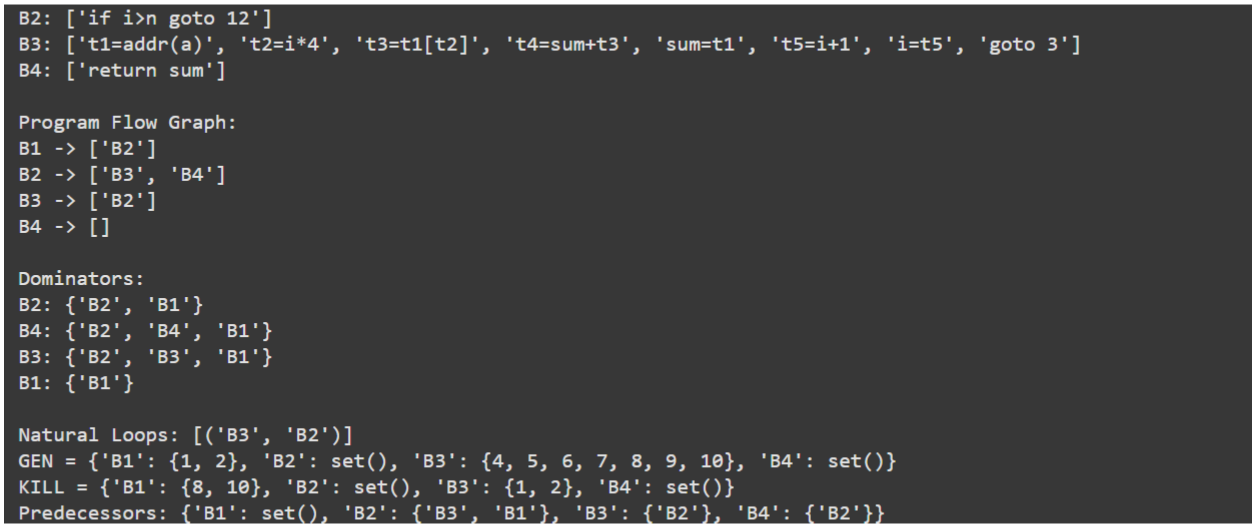
Predecessor (B): Set of all incoming edges in Block B.

```
def compute_predecessors(flow_graph):
    predecessors = {node: set() for node in flow_graph}

    for node, successors in flow_graph.items():
        for succ in successors:
            predecessors[succ].add(node)

    return predecessors
```

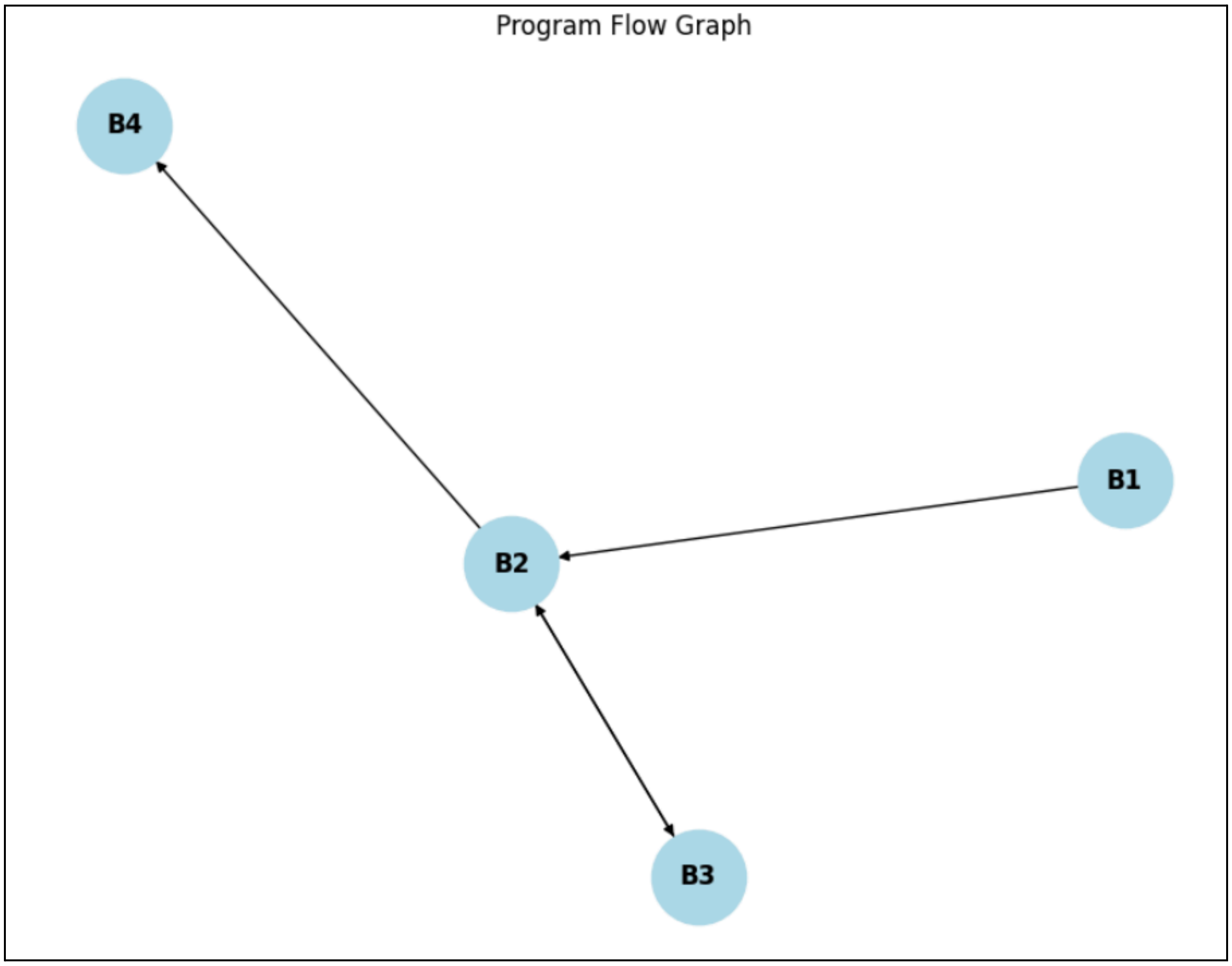
8. Input:

```
statements = [
"i = 0",
"sum = 0",
"t1 = 4",
"if i > n goto 11" ,
"t2 = i * t1",
"t3 = addr(a)",
"t4 = t3[t2]",
"sum = sum + t4",
"i = i + 1",
"goto 4",
"return sum "
]
```

Output:



```
B2: ['if i>n goto 12']
B3: ['t1=addr(a)', 't2=i*4', 't3=t1[t2]', 't4=sum+t3', 'sum=t1', 't5=i+1', 'i=t5', 'goto 3']
B4: ['return sum']

Program Flow Graph:
B1 -> ['B2']
B2 -> ['B3', 'B4']
B3 -> ['B2']
B4 -> []

Dominators:
B2: {'B2', 'B1'}
B4: {'B2', 'B4', 'B1'}
B3: {'B2', 'B3', 'B1'}
B1: {'B1'}

Natural Loops: [('B3', 'B2')]
GEN = {'B1': {1, 2}, 'B2': set(), 'B3': {4, 5, 6, 7, 8, 9, 10}, 'B4': set()}
KILL = {'B1': {8, 10}, 'B2': set(), 'B3': {1, 2}, 'B4': set()}
Predecessors: {'B1': set(), 'B2': {'B3', 'B1'}, 'B3': {'B2'}, 'B4': {'B2'}}
```



9. Computing IN and OUT

IN (B): Set of all definitions which are reaching the start of Block B.

IN (B) = U OUT (P) for predecessor P of B.

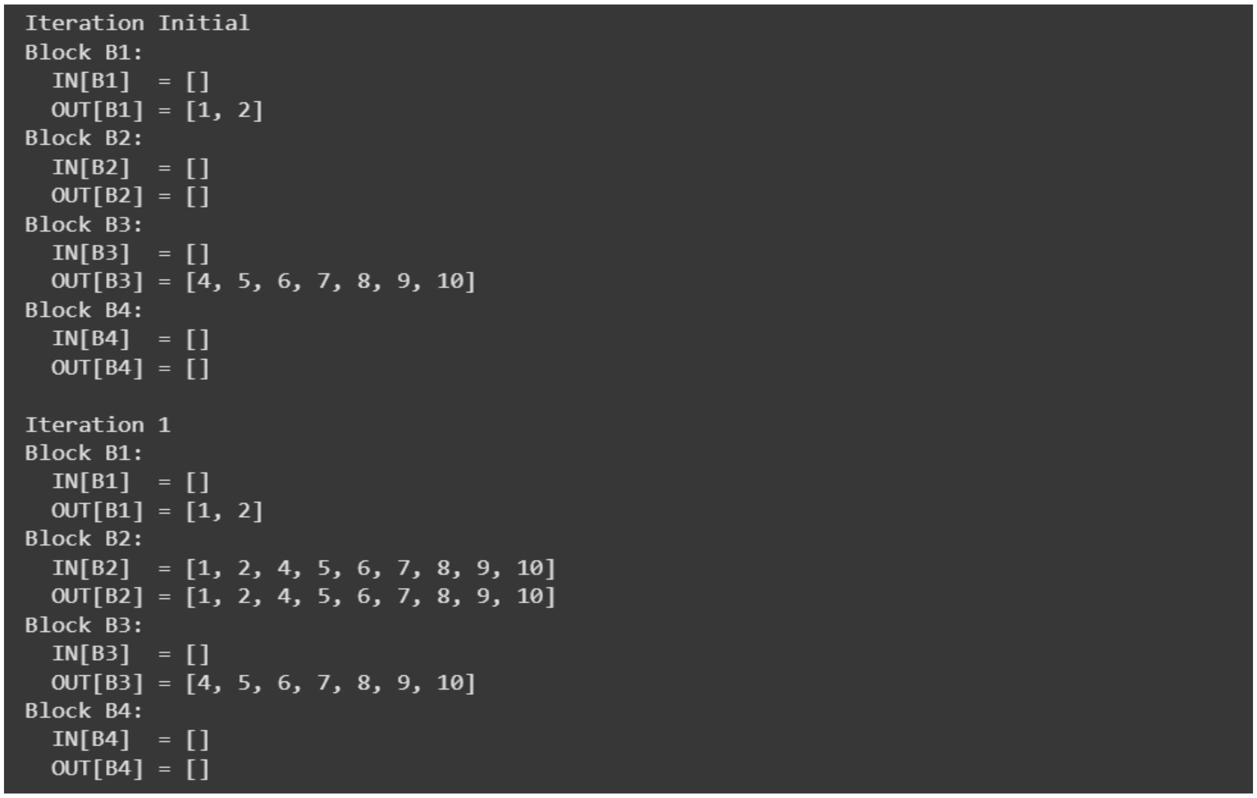
OUT (B): Set of all definitions reaching the end of Block B.

OUT (B) = IN (B)-KILL (B) U GEN (B)

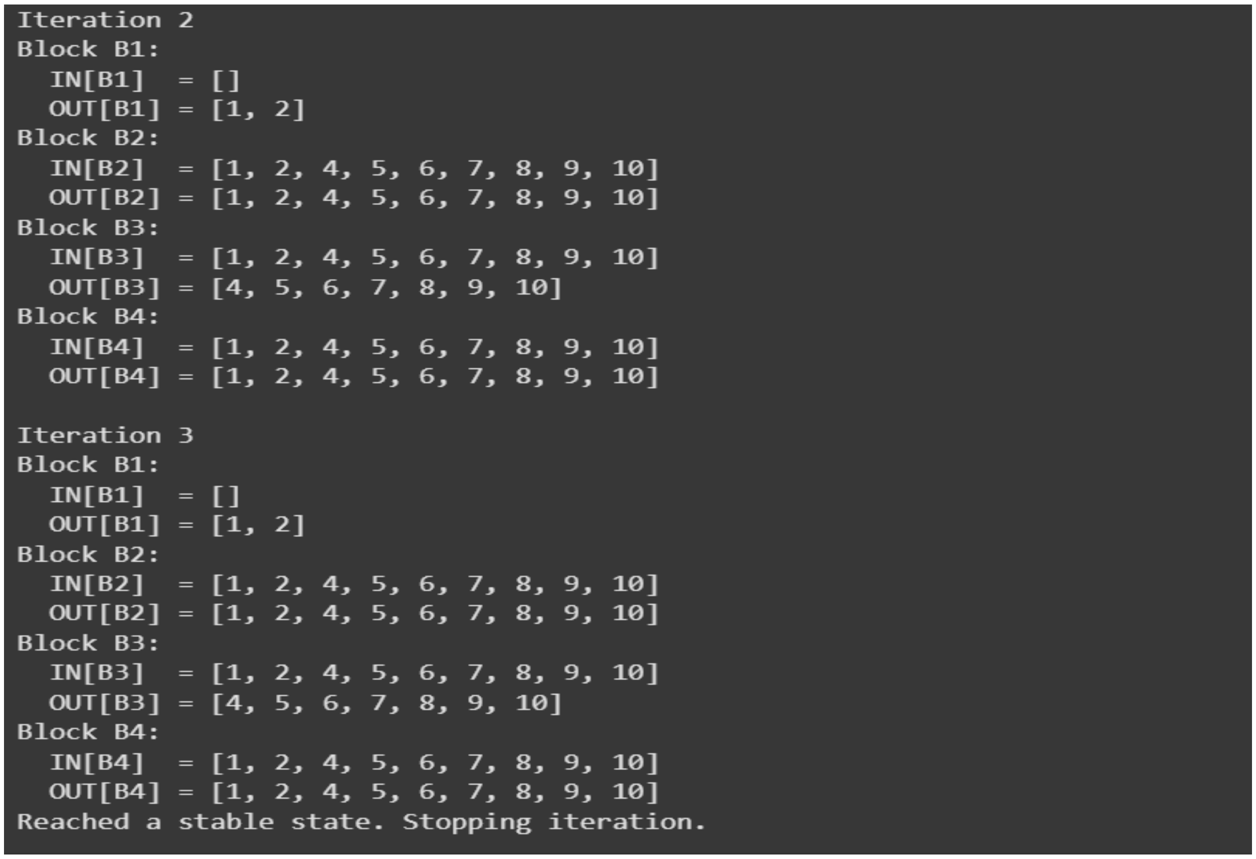
```
IN = {block: set() for block in gen}
OUT = {block: gen[block].copy() for block in gen}

def print_sets(iteration):
    print(f"\nIteration {iteration}")
    for block in sorted(gen.keys()):
        print(f"Block {block}:")
        print(f"  IN[{block}]  = {sorted(list(IN[block]))}")
        print(f"  OUT[{block}] = {sorted(list(OUT[block]))}")

print_sets("Initial")

iteration = 0
change = True

while change:
    iteration += 1
    change = False

    IN_prev = IN.copy()
    OUT_prev = OUT.copy()

    for block in sorted(gen.keys()):
        if predecessors[block]:
            IN[block] = set().union(*(OUT_prev[p] for p in
predecessors[block]))
        else:
            IN[block] = set()

        OUT[block] = gen[block].union(IN[block] - kill[block])

    print_sets(iteration)
```

```
if IN != IN_prev or OUT != OUT_prev:
    change = True
else:
    print("Reached a stable state. Stopping iteration.")
    break
```

Output:



```
Iteration Initial
Block B1:
  IN[B1]  = []
  OUT[B1] = [1, 2]
Block B2:
  IN[B2]  = []
  OUT[B2] = []
Block B3:
  IN[B3]  = []
  OUT[B3] = [4, 5, 6, 7, 8, 9, 10]
Block B4:
  IN[B4]  = []
  OUT[B4] = []

Iteration 1
Block B1:
  IN[B1]  = []
  OUT[B1] = [1, 2]
Block B2:
  IN[B2]  = [1, 2, 4, 5, 6, 7, 8, 9, 10]
  OUT[B2] = [1, 2, 4, 5, 6, 7, 8, 9, 10]
Block B3:
  IN[B3]  = []
  OUT[B3] = [4, 5, 6, 7, 8, 9, 10]
Block B4:
  IN[B4]  = []
  OUT[B4] = []
```



```
Iteration 2
Block B1:
  IN[B1]  = []
  OUT[B1] = [1, 2]
Block B2:
  IN[B2]  = [1, 2, 4, 5, 6, 7, 8, 9, 10]
  OUT[B2] = [1, 2, 4, 5, 6, 7, 8, 9, 10]
Block B3:
  IN[B3]  = [1, 2, 4, 5, 6, 7, 8, 9, 10]
  OUT[B3] = [4, 5, 6, 7, 8, 9, 10]
Block B4:
  IN[B4]  = [1, 2, 4, 5, 6, 7, 8, 9, 10]
  OUT[B4] = [1, 2, 4, 5, 6, 7, 8, 9, 10]

Iteration 3
Block B1:
  IN[B1]  = []
  OUT[B1] = [1, 2]
Block B2:
  IN[B2]  = [1, 2, 4, 5, 6, 7, 8, 9, 10]
  OUT[B2] = [1, 2, 4, 5, 6, 7, 8, 9, 10]
Block B3:
  IN[B3]  = [1, 2, 4, 5, 6, 7, 8, 9, 10]
  OUT[B3] = [4, 5, 6, 7, 8, 9, 10]
Block B4:
  IN[B4]  = [1, 2, 4, 5, 6, 7, 8, 9, 10]
  OUT[B4] = [1, 2, 4, 5, 6, 7, 8, 9, 10]
Reached a stable state. Stopping iteration.
```

10. Calculating UD Chains

UD chains may be constructed once reaching definitions are computed

- Case 1: If use of a variable b in block B is not preceded by any definition of the variable b, then attach all definitions of b in IN[B] to the u-d chain of that use of b.

- Case 2: If use of a variable b in block B is preceded by any definition of b, then only that definition is on the u-d chain of the use of variable b.

```python
def calculate_ud_chains(statements, IN):
    ud_chains = {}

    for stmt_num, statement in statements.items():
        parts = statement.split("=")

        if len(parts) > 1:
            defined_var = parts[0].strip()
            used_vars = [var for var in parts[1].strip().split()
                         if var.isalpha() and var not in ("addr", "[",
"]")]

            ud_chains.setdefault(defined_var, []).append(stmt_num)

            for var in used_vars:
                ud_chains.setdefault(var, [])
                if var not in ud_chains:
                  ud_chains[var].extend([def_num for def_num in
IN[block_name] if statements[def_num].startswith(var + "=")])
                else: # Case 2
                  ud_chains[var].append(stmt_num)

        else:
            pass

    return ud_chains
```

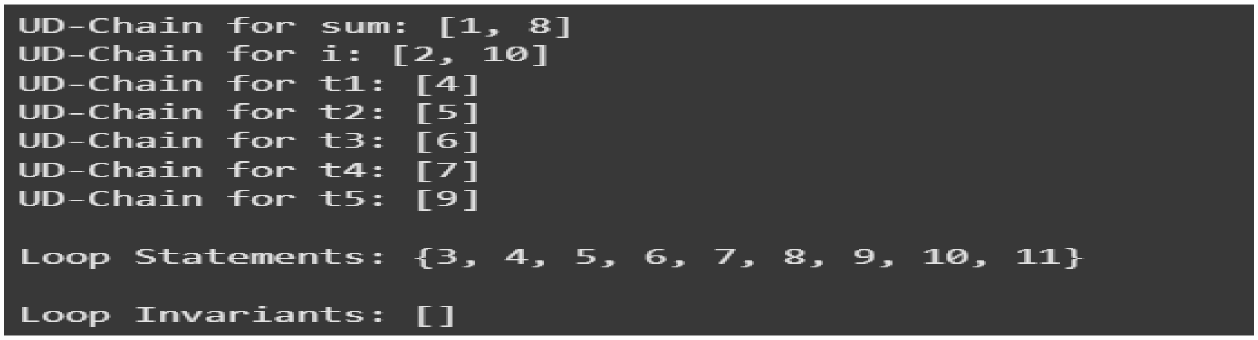
```
def generate_mappings(statements, basic_blocks):
    stmt_map = {i + 1: stmt for i, stmt in enumerate(statements)}

    stmt_to_block = {}
    for block_idx, block in enumerate(basic_blocks, start=1):
        block_name = f"B{block_idx}"
        for stmt_num, _ in block:
            stmt_to_block[stmt_num] = block_name

    return stmt_map, stmt_to_block

stmt_map, stmt_to_block = generate_mappings(statements, basic_blocks)

ud_chains = calculate_ud_chains(stmt_map, IN)

for var, chain in ud_chains.items():
    print(f"UD-Chain for {var}: {chain}")
```

11. Detecting Loop Invariants

- Find the UD-Chain of Operands on the R.H.S of all statements in the blocks contained in the loop
- If the UD-Chain of all Operands contains statement outside the loop then it is loop invariant. If the statement is loop invariant then it can be moved out of the loop to the pre-header if the conditions are satisfied (stated later)

```python
def get_loop_statements(loops, stmt_to_block):
    loop_statements = set()
    for loop in loops:
        header, back_edge = loop
        for stmt, block in stmt_to_block.items():
            if block == back_edge or block == header:
                loop_statements.add(stmt)
    return loop_statements

def identify_loop_invariants(ud_chains, loop_statements):
    invariants = []
    for var, chain in ud_chains.items():
        if all(stmt not in loop_statements for stmt in chain):
            invariants.append(var)
    return invariants
loop_statements = get_loop_statements(loops, stmt_to_block)
invariants = identify_loop_invariants(ud_chains, loop_statements)
print("\nLoop Statements:", loop_statements)
print("\nLoop Invariants:", invariants)
```

Output:



```
UD-Chain for sum: [1, 8]
UD-Chain for i: [2, 10]
UD-Chain for t1: [4]
UD-Chain for t2: [5]
UD-Chain for t3: [6]
UD-Chain for t4: [7]
UD-Chain for t5: [9]

Loop Statements: {3, 4, 5, 6, 7, 8, 9, 10, 11}

Loop Invariants: []
```

Key Observations

1. **Systematic Basic Block and CFG Generation**
   The project accurately identifies leaders and forms basic blocks, followed by the construction of a Control Flow Graph (CFG) using `networkx`. This graph captures all possible control paths, laying the groundwork for advanced code analysis and optimization.

2. **Use-Definition (UD) Chain Analysis**
   A robust UD chain computation maps each variable to its use and definition points. This analysis is fundamental for understanding data flow and is a key step toward enabling optimizations like dead code elimination and constant propagation.

3. **Loop Invariant Detection for Optimization**
   The project identifies loop-invariant computations by combining UD chain data with loop structure analysis. This insight allows for performance enhancement through loop-invariant code motion, a classic compiler optimization technique.

Significance of Loop Invariant Computation

- **Improves Execution Efficiency:** By moving loop-invariant computations outside the loop, the number of redundant operations is reduced, leading to faster execution.

- **Enables Compiler Optimizations:** It is a key component of compiler optimizations like loop-invariant code motion, which helps streamline code during the optimization phase.

- **Reduces Computational Overhead:** Avoids recalculating values that remain unchanged during loop iterations, thereby saving CPU cycles.

- **Facilitates Better Resource Utilization:** By minimizing unnecessary operations inside loops, it helps in better utilization of memory and processing power.