Lab4.2 实验报告

姓名：陈奕衡 学号：PB20000024

实验要求

根据 gvn 算法，在已经实现的 cminus 编译器的基础上，实现进一步的中间代码优化

实验难点

- 论文中算法给出的伪代码过于简略，很多问题需要自己想办法解决（循环依赖，phi 指令的创建与维护，在常量传播下的 vpf 判断）
- 试验周期虽然很长，但是因为起步阶段给的提示比较少，导致写了很多无用代码（比如直接递归处理了 bin 和 phi 类型指令导致循环依赖问题，加入常量传播之后等价类会分裂导致需要在 intersect 外处理 phi 指令）
- 隐藏样例出问题很难去 debug，根本不知道为什么代码出了问题，尤其是像重载使用错误这种难以发现的问题。公开样例相比隐藏样例过于简单，反而未能直观体现出这类错误，导致想全部通过隐藏样例十分困难，也难以找到自己代码逻辑上的误区
- 我的 c++ 功底不好，因此阅读相关的 cpp 文件以正确调用函数也成为了一个相当难的点

实验设计

detectEquivalences() 函数

这个函数整体分为了三个部分，根据指令来算的话，整体的时间复杂度为 $O(n)$（n 为指令数，并且假设内部调用的每个函数都为常数时间复杂度，后续会在思考题中计算算法具体的总复杂度）。根据论文，该函数最多迭代 n 次收敛。

```
bool changed = false;
// initialize pout with top
partitions pout = {};
partitions Top;
Top.insert(GVN::createCongruenceClass());

//deal with function
if (func_->get_num_of_args()) {
    for(auto &arg : func_->get_args()) {
        auto argval = dynamic_cast<Value *>(arg);
        auto lval = ArgExpression::create(arg);
        auto newC = createCongruenceClass(next_value_number_++);
        newC->value_expr_ = lval;
        newC->leader_ = argval;
        newC->members_.insert(newC->leader_);
        pout.insert(newC);
    }
}

//deal with global_variable
```

```
for (auto &gv1 : m_->get_global_variable()) {
    auto gv = &gv1;
    auto gvVal = dynamic_cast<Value *>(gv);
    auto lval = GlobalExpression::create(gv);
    auto newC = createCongruenceClass(next_value_number_++);
    newC->value_expr_ = lval;
    newC->leader_ = gvVal;
    newC->members_.insert(newC->leader_);
    pout.insert(newC);
}

//deal with entry_bb's partition_out
int flag = 0;
for (auto &bb1 : func_->get_basic_blocks()) {
    auto bb = &bb1;
    if (flag == 0) {
        pin_.insert({bb, {}});
        for (auto &instr1 : bb->get_instructions()) {
            auto instr = &instr1;
            pout = transferFunction(instr, pout);
        }
        pout_.insert({bb, pout});
        flag = 1;
    }
    else pout_.insert({bb, Top});
}
```

首先就是构建 Top 分区，并且处理 entry_bb （bb 块里面的变量，函数参数，以及全局变量）的分区，由于 entry_bb 不存在前驱块，因此无需考虑 phi 指令，直接对每条指令使用 transferFunction() 即可。在此之后就得到了处理好的 entry_bb 分区，并且完成了之后所有分区的初始化。

```
// iterate until converge
int num = 0;
do {
    // see the pseudo code in documentation
    int flag = 0;
    changed = false;
    partitions ori_pout;
    for (auto &bb1 : func_->get_basic_blocks()) { // you might need to visit the
blocks in depth-first order
        auto bb = &bb1;
        // get PIN of bb by predecessor(s)
        // iterate through all instructions in the block
        // and the phi instruction in all the successors
        if (flag == 0) {
            flag++;
            continue;
        }

        partitions pin = Top;
        for (auto &instr1 : bb->get_instructions()) {
```

```
            auto instr = &instr1;
            if (instr->is_phi()) copyStatment(instr);
        }
        ori_pout = pout_[bb];

        for (auto &bb_pre : bb->get_pre_basic_blocks()) {
            pin = join(pin, pout_[bb_pre]);
        }
        if (pin_[bb] == pin && num != 0) continue;

        pin_[bb] = clone(pin);
        pout_[bb] = clone(pin_[bb]);

        for (auto &instr1 : bb->get_instructions()) {
            auto instr = &instr1;
            pout_[bb] = transferFunction(instr, pout_[bb]);
        // check changes in pout
        }
        if (!(pout_[bb] == ori_pout)) changed = true;
    }
    num++;
} while (changed);
```

进入迭代之后，没有必要对于 entry_bb 中的指令做进一步处理，直接跳过。之后对于每个基本块首先进行 phi 指令的处理，即进行 copyStatment() 以及 join()。在预先处理好 phi 指令之后，便可以开始对于其他指令在 transferfunc() 中进行处理了，最后还需要进行判断分区是否在迭代改变以停止迭代。这里还有一个需要注意的点，就是对于 pout 中的 copyStatment() 的维护，如果每次无论 pin 变不变都进行迭代，会导致上一次 copy 的 phi 指令被抹去，最终得到一个错误的等价类，因此对于 pin 不变而不继续进行对块中指令执行 transferfunc() 极为重要。

```
for (auto &bb1 : func_->get_basic_blocks()) {
    auto bb = &bb1;
    for (auto &cc : pout_[bb]) {
        if (cc->value_expr_ && cc->value_expr_->get_expr_type() ==
Expression::e_constant) {
            auto lcon = std::dynamic_pointer_cast<ConstantExpression>(cc-
>value_expr_);
            cc->leader_ = dynamic_cast<Constant *>(lcon->get_lval());
        }
    }
}
```

最后这段就是单纯的辅助进行常量传播，即将每一个等价类中 value_expr_ 是 ConstantExpression 的等价类的 leader_ 替换为相应的常数，以保证 replace_cc_members() 能正确删除指令。

介绍完整个算法的核心函数，之后就是各个功能函数的介绍。

transferFunction(Instruction *x, partitions pin) 函数

这个函数主要用来处理除 phi 指令以外的所有指令，同样分为三个部分。同上假设，这个函数时间复杂度为 $O(n)$（这里进行删除和插入都需要遍历）

首先是对于待处理操作数在原等价类的删除

```
if (x->is_void() || x->is_phi()) return pin;

partitions pout = clone(pin);
auto xval = dynamic_cast<Value *>(x);
// TODO: get different ValueExpr by Instruction::OpID, modify pout
std::set<Value *>::iterator it;
for (auto &cc : pout) {
    if (cc->index_ == 0 || !cc->members_.size()) {
        pout.erase(cc);
        if (!pout.size()) break;
    }
    if ((it = cc->members_.find(xval)) != cc->members_.end()) {
        cc->members_.erase(it);
    }
}
```

这里会跳过 void 类型的指令（就是没有返回值的指令）和已经处理好的 phi 指令。这里要将空等价类删去，避免后续函数使用 pout 时报段错误

之后是对于指令的分析：

```
auto ve = valueExpr(x, pout);
if (!ve) return pout;
shared_ptr<Expression> vpf;
vpf = valuePhiFunc(ve, pout);
```

最后是对于新生成的指令的插入：

```
int flag = 0;
for (auto &cc : pout) {
    if(vpf && (*(cc->value_expr_) == *(vpf))){
        cc->members_.insert(xval);
        cc->value_expr_ = vpf;
        flag = 1;
        break;
    }
    else if (*(cc->value_expr_) == *(ve)) {
        cc->members_.insert(xval);
        flag = 1;
        break;
    }
}
```

```
if (!flag) {
    auto newC = createCongruenceClass(next_value_number_++);
    if (vpf) newC->value_expr_ = vpf;
    else newC->value_expr_ = ve;

    newC->leader_ = xval;
    newC->members_.insert(newC->leader_);
    pout.insert(newC);
}

return pout;
```

这里发现通过 valuePhiFunc() 找到的 vpf 最终与其等价类中的 ve 一致，因此决定使用 ve 统一代替 value_phi_ 的功效，以便减少变量的维护。

valueExpr(Instruction *instr, partitions pin) 函数

这个函数是根据不同指令生成相应的 value_expr，这里大概分成了几种类型的指令介绍。由于遍历了 partition 找指令，因此时间复杂度也为 $O(n)$

binary 指令

此类指令的 expr 对应如下：

```
// arithmetic expression
class BinaryExpression : public Expression {
  public:
    static std::shared_ptr<BinaryExpression> create(Instruction::OpID op,
                                                    std::shared_ptr<Expression>
lhs,
                                                    std::shared_ptr<Expression>
rhs) {
        return std::make_shared<BinaryExpression>(op, lhs, rhs);
    }
    virtual std::string print() {
        return "(" + Instruction::get_instr_op_name(op_) + " " + lhs_->print() + "
" + rhs_->print() + ")";
    }

    bool equiv(const BinaryExpression *other) const {
        if (op_ == other->op_ and *lhs_ == *other->lhs_ and *rhs_ == *other->rhs_)
            return true;
        else
            return false;
    }

    Instruction::OpID get_instr_type() const { return op_; }
    gvn_expr_t get_lexpr_type() const { return lhs_->get_expr_type(); }
    gvn_expr_t get_rexpr_type() const { return rhs_->get_expr_type(); }
    std::shared_ptr<Expression> get_lval() const { return lhs_; }
    std::shared_ptr<Expression> get_rval() const { return rhs_; }
```

```
    BinaryExpression(Instruction::OpID op, std::shared_ptr<Expression> lhs,
std::shared_ptr<Expression> rhs)
        : Expression(e_bin), op_(op), lhs_(lhs), rhs_(rhs) {}

  private:
    Instruction::OpID op_;
    std::shared_ptr<Expression> lhs_, rhs_;
};
```

这种表达式对应于 instruction.h 中的 isBinary() 中的所有指令。其构造过程如下：

```
shared_ptr<Expression> lval, rval;

auto conLval = dynamic_cast<Constant *>(instr->get_operand(0));
if (conLval) {
    lval = ConstantExpression::create(conLval, 0);
} else {
    auto InsLval = dynamic_cast<Instruction *>(instr->get_operand(0));
    for (auto &cc : pin) {
        int flag = 0;
        for (auto mem : cc->members_) {
            if (mem == instr->get_operand(0)) {
                if (cc->value_expr_->get_expr_type() == Expression::e_constant &&
!InsLval->is_phi()) {
                    auto lcon = std::dynamic_pointer_cast<ConstantExpression>(cc-
>value_expr_);
                    lval = cc->value_expr_;
                    conLval = lcon->get_lval();
                }
                else if (cc->value_expr_->get_expr_type() == Expression::e_arg ||
                         cc->value_expr_->get_expr_type() ==
Expression::e_global) {
                    lval = cc->value_expr_;
                } else {
                    lval = VarExpression::create(mem, cc->value_expr_);
                }

                flag = 1;
                break;
            }
        }
        if (flag) break;
    }
}

...

if (conLval && conRval) {
    auto conval = folder_->compute(instr, conLval, conRval);
    auto conexp = ConstantExpression::create(conval, 0);
    return conexp;
```

```
}

if (!lval || !rval) return {};

auto exp = BinaryExpression::create(instr->get_instr_type(), lval, rval);

return exp;
```

这里只展示了 lval 的构造，其大致思路就是：对于常量，给 lval 分配一个 ConstantExpression；对于变量，需要寻找到其相应的 value_expr_ 之后根据不同表达式作相应处理。这里的 VarExpression 用来表示除了函数参数和全局变量以外的所有变量，是用来解决循环依赖问题和通过比较不同变量 value_expr_ 来实现取等的办法。最后，还要根据常量来进行常量传播。

这里可以实现如下的优化：

- 优化前

```
%op1 = call i32 @input()
%op2 = %op1 + 1
%op3 = %op1 + 1
%op4 = %op2 + 1
%op5 = %op3 + 1
%op6 = 3 + 1
%op7 = %op6 + 1
%op8 = %op7 + %op6
%op9 = %op1 + %op6
call void @output(i32 %op4)
call void @output(i32 %op5)
call void @output(i32 %op7)
call void @output(i32 %op8)
call void @output(i32 %op9)
```

- 优化后

```
%op1 = call i32 @input()
%op2 = %op1 + 1
%op4 = %op2 + 1
%op9 = %op1 + 4
call void @output(i32 %op4)
call void @output(i32 %op4)
call void @output(i32 5)
call void @output(i32 9)
call void @output(i32 %op9)
```

cmp 指令

这类指令与 bin 指令的处理基本一致，只需要多添加一项 cmp_op 的判断即可

```
dynamic_cast<CmpInst *>(instr)->get_cmp_op()
```

call 指令

这里只需处理有返回值的 call，其结构如下：

```
class FuncExpression : public Expression {
  public:
    static std::shared_ptr<FuncExpression> create(Instruction *ins, Value *val,
std::vector<std::shared_ptr<Expression>> args, bool is_pure) { return
std::make_shared<FuncExpression>(ins, val, args, is_pure); }
    virtual std::string print() { return ins_->print(); }
    // we leverage the fact that constants in lightIR have unique addresses
    bool equiv(const FuncExpression *other) const {
      if (!is_pure_) {
        return ins_ == other->ins_;
      }

      if (!this->args_.size() && !other->args_.size()) {
        return val_ == other->val_;
      } else {
        if (this->args_.size() != other->args_.size()) {
          return false;
        }

        auto it1 = this->args_.cbegin();
        auto it2 = other->args_.cbegin();
        while (it1 != this->args_.cend() && it2 != other->args_.cend()) {
            if (!(*it1 == *it2)){
                return false;
            }
            it1++;
            it2++;
        }
        return val_ == other->val_;
      }
    }
    FuncExpression(Instruction *ins, Value *val,
std::vector<std::shared_ptr<Expression>> args, bool is_pure) : Expression(e_func),
ins_(ins), val_(val), args_(args), is_pure_(is_pure) {}

  private:
    Instruction *ins_;
    Value *val_;
    std::vector<std::shared_ptr<Expression>> args_;
    bool is_pure_;
};
```

这里添加了 is_pure_ 来储存函数是否为纯函数，之后还取了一个 Instruction 辅助虚函数的判断。其大体构造思路跟 bin 一致。

剩余的指令基本与 bin 的构造一致，因此不单独做介绍。值得注意的是对于 load 指令和 alloca 指令，只进行等价类创建，而不参与任何合并操作。对于类型转换类指令，同样要进行常量传播。

valuePhiFunc(shared_ptr ve, const partitions &pin) 函数

此函数的作用就是去寻找 bin 和 phi 之间的间接优化关系进行优化。整体算法思路与伪代码一致，需要特别注意的一个地方就是在计算表达式的时候需要考虑常量折叠。由于遍历了 partition 找指令，因此时间复杂度也为 $O(n)$。

```
for (auto &cc : pin) {
    if (cc->value_expr_ == lphi) {
        auto instr = dynamic_cast<Instruction *>(cc->leader_);

        if (!instr) return {};

        auto ins_instr = dynamic_cast<Instruction *>(instr->get_operand(0));

        if (ins_instr) {
            Constant *lphi_conval;
            if (lphi_lval->get_expr_type() == Expression::e_constant &&
                rphi_lval->get_expr_type() == Expression::e_constant) {
                auto lcon = std::dynamic_pointer_cast<ConstantExpression>
(lphi_lval);
                auto rcon = std::dynamic_pointer_cast<ConstantExpression>
(rphi_lval);

                auto bconLval = lcon->get_lval();
                auto bconRval = rcon->get_lval();
                lphi_conval = folder_->compute(ins_instr, bconLval, bconRval);
                lexp = ConstantExpression::create(lphi_conval, 0);
            } else {
                lexp = BinaryExpression::create(bve->get_instr_type(), lphi_lval,
rphi_lval);
            }
        }
        else return {};

        auto bb = dynamic_cast<BasicBlock *>(instr->get_operand(1));
        lpout = pout_[bb];

        break;
    }
}
```

这里对 getVN 函数做出了一点改动，如下：

```
shared_ptr<Expression> GVN::getVN(const partitions &pout, shared_ptr<Expression>
ve) {
    // TODO: return what?
    for (auto it = pout.begin(); it != pout.end(); it++) {
        if ((*it)->value_expr_ and *(*it)->value_expr_ == *ve) {
            if ((*it)->value_expr_->get_expr_type() == Expression::e_constant) {
                return (*it)->value_expr_;
            }

            auto var = VarExpression::create((*it)->leader_, (*it)->value_expr_);
            return var;
        }
    }
    return {};
}
```

对于常量返回相应表达式即可，但是对于变量则需要返回其 VarExpression 以保证能够正确构造 phi 函数

copyStatment(Instruction *instr) 函数

这个函数是在 join 函数之前处理 phi 指令的，即将 phi 指令当成赋值语句放在左右两边的前驱块的对应等价类中。这个步骤分为两个步骤，删除和复制。由于遍历了 partition 找指令，因此时间复杂度也为 $O(n)$

```
auto instrval = dynamic_cast<Value *>(instr);

auto lval = instr->get_operand(0);
auto bb_prel = dynamic_cast<BasicBlock *>(instr->get_operand(1));

for (auto &cc : pout_[bb_prel]) {
    if (cc->index_ == 0) break;
    std::set<Value *>::iterator it;
    if((it = cc->members_.find(instrval)) != cc->members_.end()) {
        cc->members_.erase(it);
        if(!cc->members_.size()) {
            pout_[bb_prel].erase(cc);
            break;
        }
    }
}
```

这里展现了对于上一次 copy 或 join 生成的 phi 的结果的删除步骤。不去删除的话，就会在一个分区中，该 phi 指令同时处在两个不同的等价类中，导致后续过程错误。

```
int lflag = 0;
auto conLval = dynamic_cast<Constant *>(lval);
for (auto &cc : pout_[bb_prel]) {
    if (cc->index_ == 0) break;
    if (cc->value_expr_->get_expr_type() == Expression::e_valf) {
```

```
            auto lcon = std::dynamic_pointer_cast<ConstantfExpression>(cc-
>value_expr_);
            auto conval = lcon->get_lval();
            if (conLval == conval && lcon->get_bb() == bb_prel) {
                cc->members_.insert(instrval);
                lflag = 1;
                break;
            }
        }
        for (auto mem : cc->members_) {
            if (mem == lval) {
                cc->members_.insert(instrval);
                lflag = 1;
                break;
            }
        }
        if (lflag) break;
    }

    if (conLval && !lflag) {
        auto lexp = ConstantfExpression::create(conLval, 0, bb_prel);
        auto newC = createCongruenceClass(next_value_number_++);
        newC->value_expr_ = lexp;
        newC->leader_ = instrval;
        newC->members_.insert(instrval);
        pout_[bb_prel].insert(newC);
        for (auto &cc : pout_[bb_prel]) {
            if (cc->index_ == 0) pout_[bb_prel].erase(cc);
        }
    }
```

这里是进行复制的步骤，考虑到插入等价类或者新建等价类两种情况。对于这里的 phi，使用了一种特殊的表达式，以防止错误的合并。

这里所说的错误合并如下：

```
labelx:
    %op1 = phi i32 [ 2, %label_entry ], [ %op5, %labely ]
labely:
    %op5 = %op1 + 1
    %op10 = phi i32 [ 2, %label? ], [ %op31, %labelz ]
labelz:
    %op31 = %op10 + 1
```

如果使用普通的常数折叠，会将 %op1 和 %op10 错误的分在同一类，之后由于变量表达式 VarExpression 会根据 expr_判断相不相等，因此就会错误分类。所以，这里设计了 ConstantfExpression 用来存储 phi 的 bb，以保证两个 phi 函数在第一轮分类时不会被误判为相等。

join(const partitions &P1, const partitions &P2) 函数

这里我是直接照搬了伪代码：

```
partitions p = {};
if (!P1.size()) return p;
else if (!P2.size()) return p;

for (auto &cc : P1) {
    if (cc->index_ == 0) return clone(P2);
}
for (auto &cc : P2) {
    if (cc->index_ == 0) return clone(P1);
}

if (P1 == P2) return clone(P1);

for (auto &cc1 : P1) {
    for (auto &cc2 : P2) {
        auto newCC = GVN::intersect(cc1, cc2);
        if (newCC) p.insert(newCC);
    }
}

return p;
```

前面的空集判断是为了配合我的 detectEquivalences() 函数。该函数的时间复杂度为 $O(n^2)$。

intersect(std::shared_ptr Ci, std::shared_ptr Cj) 函数

这个函数主要用来处理交集和生成 phi 函数，其构造分为三个部分。该函数的时间复杂度为 $O(v)$，v 是一个等价类中的变量数。

```
auto newC = createCongruenceClass(next_value_number_++);
std::set_intersection(Ci->members_.cbegin(), Ci->members_.cend(),
                      Cj->members_.cbegin(), Cj->members_.cend(),
                      inserter(newC->members_, newC->members_.begin()));
```

这里使用了 set_intersection 函数取交集，之后就是对于交集的相应处理：

```
if (Ci->leader_ == Cj->leader_) {
    newC->index_ = Ci->index_;
    newC->value_expr_ = Ci->value_expr_;
    newC->leader_ = Ci->leader_;
    newC->value_phi_ = Ci->value_phi_;
    return newC;
}
```

leader_ 相等时，代表了两个等价类相等（leader_ 会和 value_expr_ 绑定），这样的话直接使用新的 value_expr_ 即可。观察到有两个前驱块时，左边的前驱块会比右边的前驱块先遍历，因此左边的值一定是最新值。

```
auto instr = dynamic_cast<Instruction *>(mem);
if (instr && instr->is_phi()) {
    shared_ptr<Expression> lval, rval, lexp, rexp;

    auto conLval = dynamic_cast<Constant *>(instr->get_operand(0));
    if (conLval) {
        lval = ConstantExpression::create(conLval, 0);
        lexp = lval;
    } else {
        auto InsLval = instr->get_operand(0);
        for (auto mem : Ci->members_) {
            if (mem == InsLval) {
                if (Ci->value_expr_->get_expr_type() == Expression::e_constant ||
                    Ci->value_expr_->get_expr_type() == Expression::e_arg ||
                    Ci->value_expr_->get_expr_type() == Expression::e_global) {
                    lval = Ci->value_expr_;
                } else {
                    lval = VarExpression::create(mem, Ci->value_expr_);
                }
                lexp = Ci->value_expr_;
                break;
            }
        }
    }

    ...

    if (lval && rval) {
        shared_ptr<Expression> exp;
        if (*lval == *rval) exp = lexp;
        else exp = PhiExpression::create(lval, rval);

        newC->value_expr_ = exp;
        newC->leader_ = mem;
        return newC;
    }
}
```

这里是对于 phi 函数的构建，可以看出只有左右都有值的时候，正确 phi 函数才会被创建。这里还有一个特判，是关于 phi 指令左右两边相等的情况（phi(1, 1) = 1）。

重载函数

判断分区相等

这个只需要判断分区中所有等价类一致即可

```
bool operator==(const GVN::partitions &p1, const GVN::partitions &p2) {
    // TODO: how to compare partitions?
    if (!p1.size() && !p2.size()) return true;

    if (!p1.size() || !p2.size()) return false;

    if (p1.size() == p2.size()) {
        auto it1 = p1.cbegin();
        auto it2 = p2.cbegin();
        while (it1 != p1.cend() && it2 != p2.cend()) {
            if (!(**it1 == **it2)) return false;
            it1++;
            it2++;
        }
        return true;
    }
    return false;
}
```

判断等价类相等

这个函数无需判断等价类中的全部元素，只需核心判断 value_expr_ 和 member_ 相等即可

```
bool CongruenceClass::operator==(const CongruenceClass &other) const {
    // TODO: which fields need to be compared?
    if (this->value_expr_ == other.value_expr_ && this->members_.size() ==
other.members_.size()) {
        auto it1 = this->members_.cbegin();
        auto it2 = other.members_.cbegin();
        while (it1 != this->members_.cend() && it2 != other.members_.cend()) {
            if (!(*it1 == *it2)) return false;
            it1++;
            it2++;
        }
        return true;
    }

    return false;
}
```

思考题

1. 请简要分析你的算法复杂度

根据论文，假设有 n 个等价类，每个等价类有 v 个变量，程序中一共有 j 个汇合点。这样子根据上述分析，时间复杂度最高的就是 join 函数，所以每次迭代的最高复杂度就是 $O(n^2v^2j)$。总共的最大迭代次数为 n，因此总时间复杂度为 $O(n^3v^2j)$，这里跟论文分析的有出入（论文中最终为 $O(n^3vj)$），是因为我没有建立起 value_expr_ 和 phi 表达式的映射，导致了多一次的遍历，从而有更高的复杂度。

2. `std::shared_ptr`如果存在环形引用，则无法正确释放内存，你的 Expression 类是否存在 circular reference?

我的 Expression 类不会存在环形引用，因为每个 expression 内的嵌套只有 ConstantExpression 和 VarExpression，而这两个表达式本身不可能形成环形引用。

3. 尽管本次实验已经写了很多代码，但是在算法上和工程上仍然可以对 GVN 进行改进，请简述你的 GVN 实现可以改进的地方

- 首先肯定就是冗余的 bb 块了，就像是例子中 labely 的 bb 块是可以完全优化掉的。

```
labelx:
    ...
    br labely
labely:
    br labelz
labelz:
    ...
```

- 其次是对于异常处理程序的调用，代码里面是会每个数组都开一个相应的 bb 块，但是鉴于这里会直接跳出，因此所有数组异常处理只需跳进去同一个块即可。进一步，代码完全相同的某些 bb 块也可以进行合并，这种合并可能会招致前驱块大于两个，因此要进一步改进 join() 函数并且考虑会对循环结构产生的意料之外的错误。

```
labelx:
    call void @neg_idx_except()
    ret void
labely:
    ...
labelz:
    call void @neg_idx_except()
    ret void
```

- 之后还有相应的 load 和 store 指令的进一步优化。比如当两个 load 指令的目标地址相同并且中间不包括对该地址进行任何 store 操作时，这两个 load 指令可以合并。

```
labelx:
    %op37 = load i32, i32* %op36
    ...(no store of %op36)
    %op99 = load i32, i32* %op36
```

- 以及涉及 phi 的常量传播，例子如下：

```
labelx:
    %op17 = phi i32 [ 2800, %label8 ], [ 2786, %label31 ]
```

```
    %op18 = add %op17, 10
labely:
    %op19 = phi i32 [ 2810, %label8 ], [ 2796, %label31 ]
    %op20 = call i32 @input()
    %op21 = add %op33, %op34
```

可以优化为：

```
labelx:
    %op17 = phi i32 [ 2800, %label8 ], [ 2786, %label31 ]
    %op18 = phi i32 [ 2810, %label8 ], [ 2796, %label31 ]
labely:
    %op19 = call i32 @input()
    %op20 = add %op18, %op34
```

- 还可以将访问块的顺序改为深度优先遍历，减少可能发生的潜在错误
- 建立起 value_expr_ 和 phi 表达式的映射，降低算法时间复杂度

实验总结

收获到了一个新的算法，很多 c++ 使用技巧，以及一颗为 debug 日夜不眠的心。当然，在此基础上能够更加熟练运用流图，进一步理解了 ssa, phi 指令以及常量传播在其中的作用。最后，最大的收获莫过于一个可以处理部分代码优化的中间代码优化器以及几乎实现整个实验的成就感。

实验反馈（可选 不会评分）

真心希望可以多添加一点公开样例来暴露自身代码问题，或者是对于运算符重载或者指针运用（c++ 基础知识）这方面多一些引导。写了大概一个月，更多时候不是在思考代码逻辑上的问题或者缺失考虑的情况（跟编译相关的知识），而是在想这个重载用的对不对，那个指针写的对不对，到最后的一个没能解决的 bug 依旧是这方面的问题，但是真的不知道在哪里出了问题。

我也很感谢这次实验的经历，让我了解到了更多的关于 c++ 实践的相关知识（编译原理这门课的实验是第二个需要用到大量 c++ 知识的），助教对于类封装的代码风格也让我受益匪浅。