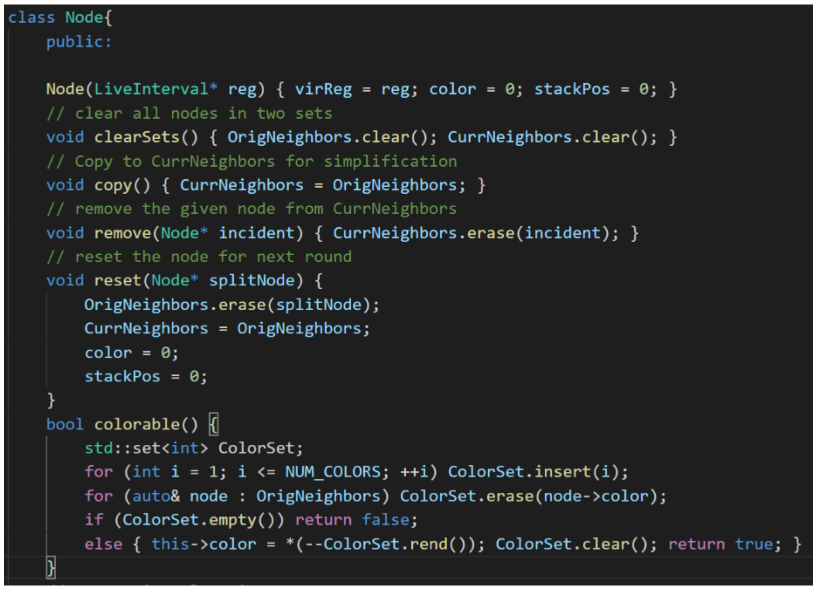
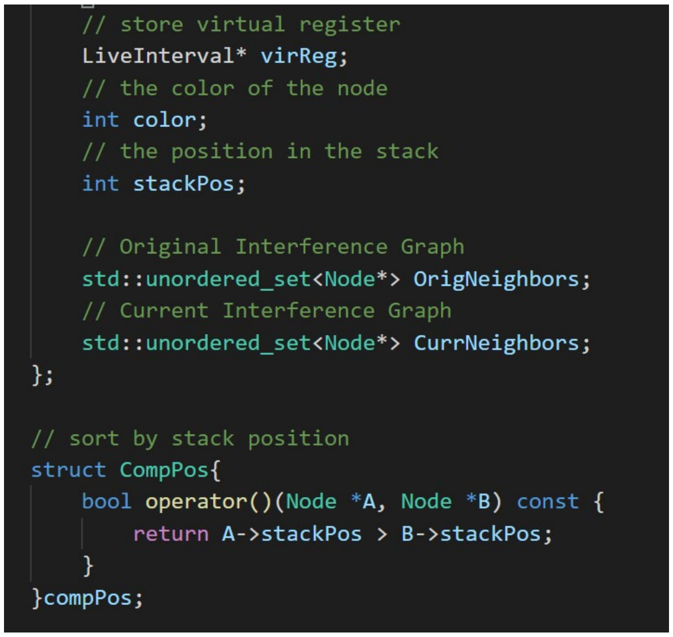
CS502-PA6 Report Meng-Chieh Lin

First, I use the following class to build the structure for each node. With the member functions, I can get all I need in the process. The colorable function is to check if the node is colorable.



```
class Node{
    public:

    Node(LiveInterval* reg) { virReg = reg; color = 0; stackPos = 0; }
    // clear all nodes in two sets
    void clearSets() { OrigNeighbors.clear(); CurrNeighbors.clear(); }
    // Copy to CurrNeighbors for simplification
    void copy() { CurrNeighbors = OrigNeighbors; }
    // remove the given node from CurrNeighbors
    void remove(Node* incident) { CurrNeighbors.erase(incident); }
    // reset the node for next round
    void reset(Node* splitNode) {
        OrigNeighbors.erase(splitNode);
        CurrNeighbors = OrigNeighbors;
        color = 0;
        stackPos = 0;
    }
    bool colorable() {
        std::set<int> ColorSet;
        for (int i = 1; i <= NUM_COLORS; ++i) ColorSet.insert(i);
        for (auto& node : OrigNeighbors) ColorSet.erase(node->color);
        if (ColorSet.empty()) return false;
        else { this->color = *(--ColorSet.rend()); ColorSet.clear(); return true; }
    }
```

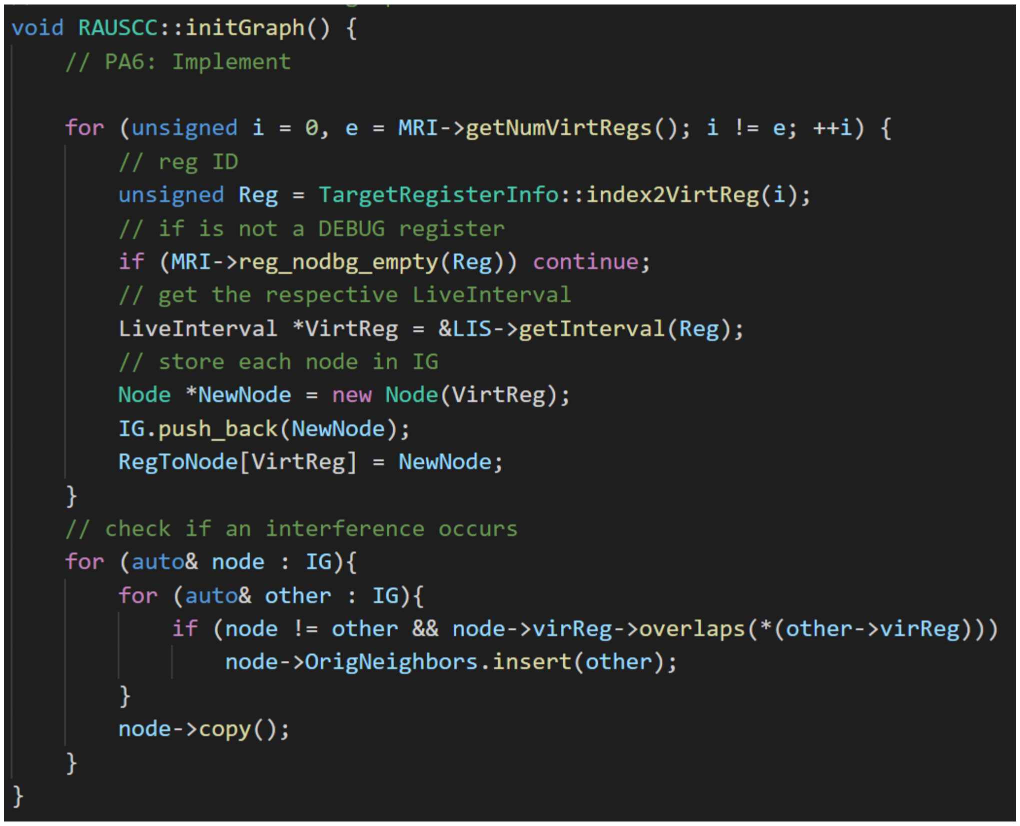


```
    // store virtual register
    LiveInterval* virReg;
    // the color of the node
    int color;
    // the position in the stack
    int stackPos;

    // Original Interference Graph
    std::unordered_set<Node*> OrigNeighbors;
    // Current Interference Graph
    std::unordered_set<Node*> CurrNeighbors;
};

// sort by stack position
struct CompPos{
    bool operator()(Node *A, Node *B) const {
        return A->stackPos > B->stackPos;
    }
}compPos;
```

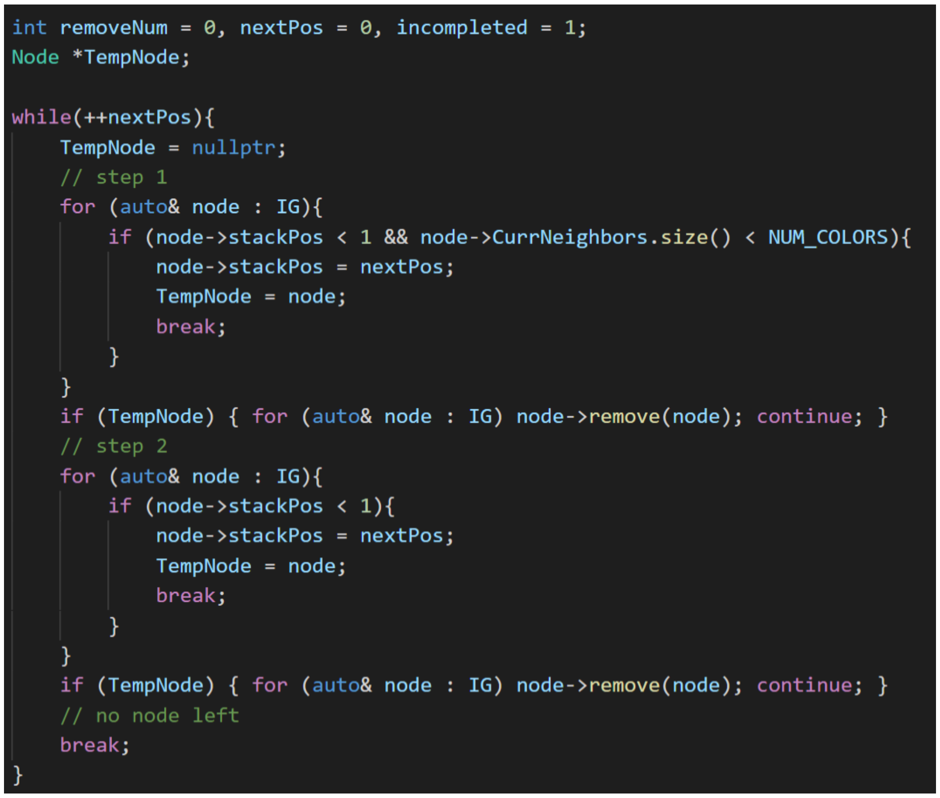
Second, I build the interference graph by storing them in a vector. To map each virtual register with the node contains it, I use unordered map to achieve that. Then I use two nested for loops to build edges between nodes if they are interfering each other.



```cpp
void RAUSCC::initGraph() {
    // PA6: Implement

    for (unsigned i = 0, e = MRI->getNumVirtRegs(); i != e; ++i) {
        // reg ID
        unsigned Reg = TargetRegisterInfo::index2VirtReg(i);
        // if is not a DEBUG register
        if (MRI->reg_nodbg_empty(Reg)) continue;
        // get the respective LiveInterval
        LiveInterval *VirtReg = &LIS->getInterval(Reg);
        // store each node in IG
        Node *NewNode = new Node(VirtReg);
        IG.push_back(NewNode);
        RegToNode[VirtReg] = NewNode;
    }
    // check if an interference occurs
    for (auto& node : IG){
        for (auto& other : IG){
            if (node != other && node->virReg->overlaps(*(other->virReg)))
                node->OrigNeighbors.insert(other);
        }
        node->copy();
    }
}
```

Last, the following part is the way I implement the algorithm. For more details, please check my code.



```cpp
int removeNum = 0, nextPos = 0, incompleted = 1;
Node *TempNode;

while(++nextPos){
    TempNode = nullptr;
    // step 1
    for (auto& node : IG){
        if (node->stackPos < 1 && node->CurrNeighbors.size() < NUM_COLORS){
            node->stackPos = nextPos;
            TempNode = node;
            break;
        }
    }
    if (TempNode) { for (auto& node : IG) node->remove(node); continue; }
    // step 2
    for (auto& node : IG){
        if (node->stackPos < 1){
            node->stackPos = nextPos;
            TempNode = node;
            break;
        }
    }
    if (TempNode) { for (auto& node : IG) node->remove(node); continue; }
    // no node left
    break;
}
```