```
void  cc_simplify_expensive(vec<Lit>& out_learnt) {
// Simplify conflict clause stored in out_learnt.
// Precondition: seen[v] == 1 if v or ~v occurs in out_learnt.

    intSet levels = find_levels(out_learnt);
    // 'levels' is the set of levels that occur in the conflict clause.
    // It is used to exit early in litRedundant() if the algorithm is
    // visiting literals at levels that cannot be removed later.

    out_learnt.copyTo(analyze_toclear);

    int i, j = 1;
    for (i = 1; i < out_learnt.size(); i++) {
        if (litRedundant(out_learnt[i], levels) == false) {
            out_learnt[j] = out_learnt[i];
            j ++; } }
    out_learnt.shrink(i - j);
    clean_seen(0); }

bool  litRedundant(Lit q0, intSet levels) {
// Check if 'q0' can be removed from conflict clause.
// Precondition: seen[v] == 1 if v or ~v occurs in out_learnt or does not
// occur, but can be removed if it is added.  Postcondition: same.

    if (reason[var(q0)] == NULL)
        return false;

    analyze_stack.clear();
    analyze_stack.push(var(q0));
    int top = analyze_toclear.size();

    while (analyze_stack.size() > 0) {
        Clause& c = *reason[analyze_stack.last()];
        analyze_stack.pop();
        for (int i = 1; i < c.size(); i++) {
            Lit qi  = c[i];
            Var v  = var(qi);
            if (!seen[v] && level[v] > 0) {
                if (reason[v] != NULL && levels.in(level[v])) {
                    analyze_stack.push(v);
                    analyze_toclear.push(qi);
                    seen[v] = 1; }
                else {
                    clean_seen(top);
                    return false; } } } }
    return true; }

void  clean_seen(int top) {
    while (analyze_toclear.size() > top) {
        Var v = var(analyze_toclear.last());
        analyze_toclear.pop();
        seen[v] = 0; } }
```