```
//! s-sided stack of two features by walking core ids
//  NOTE: s is intended to 'carry' the growing meta pattern
//  starting=1: indicate that this is iteration 0, i.e. the original call to this function
int GSWalk::conflict_resolution (vector<int> core_ids, GSWalk* s, bool starting, int ceiling) {


    // sanity check: core
    if (core_ids.size()>0) {

        // Increase hops for sw
        if (starting) { s->hops+=hops; }

        // NESTING 0 VARS
        set<int> u12; // the incremental union set over both d12 and d21 and j (includes mutex and node conflict edges)
        map<int, int> stack_locations;
        //  ^^^  ^^^
        //  to   from


        // NESTING 1 VARS (WHILE LOOP THROUGH INSERTION CANDIDATES)
        map<int, map<int, GSWNode> > ninsert21;
        map<int, map<int, GSWEdge> > einsert21;
        map<int, map<int, GSWNode> > ninsert12;
        map<int, map<int, GSWEdge> > einsert12;
        map<int, int> c12_inc;


        // NESTING 2 VARS (FOR LOOP THROUGH CORE)
        set<int> d1;  // the set of edges going out of j in this
        set<int> d2;  // the set of edges going out of j in s
        set<int> i12; // the common edges for j (core and node conflict edges)
        set<int> c12_tmp;
        set<int> c12; // the common edges for j without the core (exactly the node conflict edges)
        set<int> d12; // the mutex edges for j (neither core nor node conflict edges)
        set<int> d21; //
        set<int> index_revisit; // index nodes to be revisited, due to out-of-bound indices in to-nodes


        sort(core_ids.begin(), core_ids.end());
        if (nodewalk.size() < core_ids.size()) { cerr << "ERROR! More core ids than nodes." << endl; exit(1); }
        int border=core_ids.back();

        // prepare basic structure: copy core nodes and connecting edges from this to s, if s is empty
        bool nodewalk_empty=0;
        if (s->nodewalk.size() == 0) {
            nodewalk_empty=1;
            if (!starting) {
                cerr << "Error! Empty nodewalk but not starting." << endl; exit(1);
            }
        }

        if (nodewalk_empty) {
            #ifdef DEBUG
            if (fm::die) {
                cout << "Initializing new SW (" << activating << ")." << endl;
            }
            #endif
```

```
    }

    for (vector<int>::iterator index = core_ids.begin(); index!=core_ids.end(); index++) {
        edgemap::iterator from = edgewalk.find(*index);
        if (from!=edgewalk.end()) {
            map<int, GSWEdge>& e1i = from->second;
            for (map<int, GSWEdge>::iterator to=e1i.begin(); to!=e1i.end(); to++) {
                if (to->first <= border) {
                    map<Tid, int> weightmap_a;
                    map<Tid, int> weightmap_i;
                    set<InputNodeLabel> inl;
                    set<InputEdgeLabel> iel;
                    GSWNode n = { inl };
                    GSWEdge e = { to->first, iel, weightmap_a, weightmap_i, 0, 0 };
                    if (nodewalk_empty) s->add_edge(from->first, e, n, 0, &core_ids, &u12);
                    if (nodewalk_empty || starting) stack_locations[to->first]=from->first;
                }
            }
        }
    }
    if (nodewalk_empty || starting) s->stack(this,stack_locations);
    if (nodewalk_empty) s->activating=activating;
    //cout << "init to " << s->activating << endl;


    bool did_ceiling=0;
    stack_locations.clear();

    // Do 'create candidate list and insert lowest edge' while still edges unexpanded for core ids.
    do {

        #ifdef DEBUG
        if (fm::die) {
            cout << "-CR begin-" << endl;
            cout << this ;
            cout << s ;
            cout << "core: '";
            each(core_ids) cout << core_ids[i] << " ";
            cout << "'" << endl;
        }
        #endif

        ninsert21.clear();
        einsert21.clear();
        ninsert12.clear();
        einsert12.clear();
        c12_inc.clear();
        index_revisit.clear();

        #ifdef DEBUG
        if (fm::die) {
            cerr << "U12: '";
            each_it(u12, set<int>::iterator) cerr << *it << " ";
            cerr << "'" <<endl;
        }
        #endif

        // Gather candidate edges for all core ids!
        for (int index = 0; index<core_ids.size(); index++) {
```

```
            int j = core_ids[index];

            //s->nodewalk[j].stack(nodewalk[j]);

            d1.clear();
            d2.clear();
            edgemap::iterator e1 = edgewalk.find(j);
            edgemap::iterator e2 = s->edgewalk.find(j);
            if ( e1!=edgewalk.end() ) {
                // remember tos of this->j
                for (map<int, GSWEdge>::iterator it = e1->second.begin(); it!=e1->second.end();
                it++) d1.insert(it->first);
            }
            if ( e2!=s->edgewalk.end() ) {
                // remember tos of s->j
                for (map<int, GSWEdge>::iterator it = e2->second.begin(); it!=e2->second.end();
                it++) d2.insert(it->first);
            }
            #ifdef DEBUG
            if (fm::die) {
                cout << "j: " << j << ", W1: " << e1->second.size() << ", W2: " << e2->second.size()
                << endl;
            }
            #endif
            // calculate intersection and diff sets of tos
            i12.clear(); set_intersection(d1.begin(), d1.end(), d2.begin(), d2.end(),
            std::inserter(i12, i12.end()));                  // intersection (symmetric)
            c12_tmp.clear(); set_difference(i12.begin(), i12.end(), core_ids.begin(),
            core_ids.end(), std::inserter(c12_tmp, c12_tmp.end()));
            c12.clear(); set_difference(c12_tmp.begin(), c12_tmp.end(), u12.begin(), u12.end(),
            std::inserter(c12, c12.end()));        // intersection \ core_ids (symmetric)

            // REMOVE MORE IDS FROM C12 HERE?????
            each_it(c12, set<int>::iterator) {
                if (*it<core_ids.back()) c12.erase(*it);
            }

            d12.clear(); set_difference(d1.begin(), d1.end(), i12.begin(), i12.end(),
            std::inserter(d12, d12.end()));                  // mutex set
            d21.clear(); set_difference(d2.begin(), d2.end(), i12.begin(), i12.end(),
            std::inserter(d21, d21.end()));

            // intersection \ core
            each_it(c12, set<int>::iterator) {
                #ifdef DEBUG
                if (fm::die) {
                    map<int, GSWEdge>& w2j = e2->second;
                    cout << "C12: " << j << "->" << w2j.find(*it)->first;
                    cout << " < ";
                    set<InputEdgeLabel>& labs = w2j.find(*it)->second.labs;
                    for (set<InputEdgeLabel>::iterator it2=labs.begin(); it2!=labs.end(); it2++) {
                        cout << *it2 << " ";
                    }
                    cout << ">" << endl;
                }
                #endif
                if (ceiling==0 || *it<ceiling) c12_inc[*it]=j;
            }
```

```
                // single edges
                each_it(d21, set<int>::iterator) {
                    if (ceiling==0 || *it<ceiling) {
                        if (*it <= nodewalk.size()) { // only insert in-bound edges
                            #ifdef DEBUG
                            if (fm::die) {
                                map<int, GSWEdge>& w2j = e2->second;
                                cout << "D21: " << j << "->" << w2j.find(*it)->first;
                                cout << " < ";
                                set<InputEdgeLabel>& labs = w2j.find(*it)->second.labs;
                                for (set<InputEdgeLabel>::iterator it2=labs.begin(); it2!=labs.end();
                                it2++) {
                                    cout << *it2 << " ";
                                }
                                cout << ">" << endl;
                            }
                            #endif
                            map<Tid, int> weightmap_a;
                            map<Tid, int> weightmap_i;
                            set<InputNodeLabel> inl;
                            set<InputEdgeLabel> iel;
                            GSWNode n = { inl };
                            GSWEdge e = { *it, iel, weightmap_a, weightmap_i, 0, 0 };
                            ninsert21[*it][j]=n;
                            einsert21[*it][j]=e;
                        }
                        else { // remember index for next round
                            #ifdef DEBUG
                            if (fm::die) cout << "Node-to-revisit: " << j << endl;
                            #endif
                            index_revisit.insert(j);
                        }
                    }
                }

                // nothing inserted, so no recalculation of d12 necessary

                // single edges
                each_it(d12, set<int>::iterator) {
                    if (ceiling==0 || *it<ceiling) {
                        if (*it <= s->nodewalk.size()) {
                            #ifdef DEBUG
                            if (fm::die) {
                                map<int, GSWEdge>& w1j = e1->second;
                                cout << "D12: " << j << "->" << w1j.find(*it)->first; // needs no check
                                for end() by def of d12
                                cout << " < ";
                                set<InputEdgeLabel>& labs = w1j.find(*it)->second.labs;
                                for (set<InputEdgeLabel>::iterator it2=labs.begin(); it2!=labs.end();
                                it2++) {
                                    cout << *it2 << " ";
                                }
                                cout << ">" << endl;
                            }
                            #endif
                            map<Tid, int> weightmap_a;
                            map<Tid, int> weightmap_i;
                            set<InputNodeLabel> inl;
```

```
                    set<InputEdgeLabel> iel;

                    GSWNode n = { inl };
                    GSWEdge e = { *it, iel, weightmap_a, weightmap_i, 0, 0 };

                    ninsert12[*it][j]=n;
                    einsert12[*it][j]=e;
                }
                else {
                    #ifdef DEBUG
                    if (fm::die) cout << "Node-to-revisit: " << j << endl;
                    #endif
                    index_revisit.insert(j);
                }
            }
        }
    } // end for core ids


    map<int, map<int, GSWEdge> >::iterator it21 = einsert21.begin();
    map<int, map<int, GSWEdge> >::iterator it12 = einsert12.begin();
    map<int,int>::iterator itc=c12_inc.begin();

    // Must recognize 'bags'
    bool do_ceiling=0;
    int next_to=0;
    if (u12.size()) next_to= maxi((*(--u12.end()))+1, core_ids.back()+1);
    else next_to=core_ids.back()+1;
    int c=0;

    // Insert lowest conflict edge!
    if (itc != c12_inc.end()) {
        if ( (it21 == einsert21.end()) || (itc->first < it21->first) ) {
            if ( (it12 == einsert12.end()) || (itc->first < it12->first) ) {
                 if (itc->first>next_to) {
                     do_ceiling=1; c=itc->first;
                     #ifdef DEBUG
                     cout << "1) NEXT TO < " << itc->first << endl;
                     #endif
                 }
                 else {
                     // AM: remember itc->first (to) and itc->second (from)
                     stack_locations[itc->first]=itc->second;
                     u12.insert(itc->first);
                 }
            }
        }
        if (u12.size()) next_to= maxi((*(--u12.end()))+1, core_ids.back()+1);
    }


    // Decide which edge to insert, then do it!
    if ( it21 != einsert21.end() || it12 != einsert12.end() ) {
        bool insertion_done = 0;
        if (it21 != einsert21.end()) {
            // equal: direction should be 0 (siblingwalk dominance), so we merge from left to
            right and from top to down (in this order)
            if ( (it12 == einsert12.end()) || (it21->first < it12->first) || (it21->first ==
            it12->first) ) {
```

```
                    if (it21->second.size()>1) { cerr << "Error! More than one edge to the same node
                    (21)." << endl; exit(1); }
                    // to node is out of range: re-insert index for next round
                    if (it21->first > nodewalk.size()) {
                        cerr << "Error! 21: to-node '" << it21->first << "' is out of bound." <<
                        endl; exit(1);
                    }
                    else {
                        if (it21->first>next_to) { // Enter ceiling mode to resolve bag
                            #ifdef DEBUG
                            cout << "2) NEXT TO < " << it21->first << endl;
                            #endif
                            do_ceiling=1;
                            c=it21->first;
                        }
                        else {
                            add_edge(
                                it21->second.begin()->first,  // begin() jumps to first of always
                                only one element.
                                it21->second.begin()->second,
                                ninsert21[it21->first][it21->second.begin()->first],
                                1,
                                &core_ids,
                                &u12
                            );
                            // AM: remember it21->first (to) and it21->second.begin()->first (from)
                            for stacking
                            // stack_locations[it21->first]=it21->second.begin()->first; DO NOT
                            REMEMBER!!! ONLY DIRECTION THIS->S AND CONFLICTS
                            u12.insert(it21->first);
                        }
                    }
                    insertion_done = 1;
                }
            }
            if (it12 != einsert12.end()) {
                if ( (it21 == einsert21.end()) ||  (it12->first < it21->first) ) {
                    if (it12->second.size()>1) { cerr << "Error! More than one edge to the same node
                    (12)." << endl; exit(1); }
                    if (it12->first > s->nodewalk.size()) {
                        cerr << "Error! 12: to-node '" << it12->first << "' is out of bound." <<
                        endl; exit(1);
                    }
                    else {
                        if (it12->first>next_to) { // Enter ceiling mode to resolve bag
                            #ifdef DEBUG
                            cout << "3) NEXT TO < " << it12->first << endl;
                            #endif
                            do_ceiling=1;
                            c=it12->first;
                        }
                        else {
                            s->add_edge(
                                it12->second.begin()->first,
                                it12->second.begin()->second,
                                ninsert12[it12->first][it12->second.begin()->first],
                                1,
                                &core_ids,
                                &u12
```

```
                            );
                            // AM: remember it12->first (to) and it12->second.begin()->first (from)
                            for stacking
                            stack_locations[it12->first]=it12->second.begin()->first;
                            u12.insert(it12->first);
                        }
                    }
                    insertion_done = 1;
                }
            }

            if (!insertion_done) { cerr << "Error! No insertion done. " << einsert21.size() << " "
            << einsert12.size() << endl; exit(1); }
        }

        // bag handling
        if (do_ceiling) {
            #ifdef DEBUG
            if (fm::die) {
                cout << endl << endl << endl << "STARTING CEILING MODE next_to: '" << next_to << "'
                c: '" << c << "'" << endl << endl << endl;
            }
            #endif
            conflict_resolution(vector<int> (u12.begin(),u12.end()), s, 0, c);
            for (int i=next_to; i<c; i++) {
                u12.insert(i);
            }
            #ifdef DEBUG
            if (fm::die) {
                cout << endl << endl << endl << "ENDING CEILING MODE next_to: '" << next_to << "' c:
                '" << c << "'" << endl << endl << endl;
            }
            #endif
            do_ceiling=0;
        }


        #ifdef DEBUG
        else {
            if (fm::die) {
                 cout << "Finished CR." << endl;
            }
        }
        #endif

        each_it(edgewalk, edgemap::iterator) {
            for(map<int, GSWEdge>::iterator it2=it->second.begin(); it2!=it->second.end(); it2++) {
                if (it2->first >= nodewalk.size()) {
                    cout << "Error! Nodewalk contains not enough nodes. Index: " << it2->first << ",
                    size: " <<  nodewalk.size() << endl;
                    cout << this ;
                }
            }
        }
        each_it(s->edgewalk, edgemap::iterator) {
            for(map<int, GSWEdge>::iterator it2=it->second.begin(); it2!=it->second.end(); it2++) {
                if (it2->first >= s->nodewalk.size()) {
                    cout << "Error! S-Nodewalk contains not enough nodes. Index: " << it2->first <<
                    ", size; " << s->nodewalk.size() << endl;
```

```
                    cout << s ;
                }
            }
        }


    } while (einsert21.size() || einsert12.size() || c12_inc.size()); // Finished all edges for core
    ids
    #ifdef DEBUG
    if (fm::die) {
         cout << "Left while loop·" << endl;
    }
    #endif


    #ifdef DEBUG
    if (fm::die) {
        //cout << this ;
        //cout << s ;
        cout << "-stack-" << endl;
    }
    #endif

    each_it(edgewalk, edgemap::iterator) {
        for(map<int, GSWEdge>::iterator it2=it->second.begin(); it2!=it->second.end(); it2++) {
            if (it2->first >= nodewalk.size()) {
                cout << "Error! Nodewalk contains not enough nodes. Index: " << it2->first << ",
                size: " <<  nodewalk.size() << endl;
                cout << this ;
                exit(1);
            }
        }
    }
    each_it(s->edgewalk, edgemap::iterator) {
        for(map<int, GSWEdge>::iterator it2=it->second.begin(); it2!=it->second.end(); it2++) {
            if (it2->first >= s->nodewalk.size()) {
                cout << "Error! S-Nodewalk contains not enough nodes. Index: " << it2->first << ",
                size; " << s->nodewalk.size() << endl;
                cout << s ;
                exit(1);
            }
        }
    }

    // stack labels and activities to s
    // all core id nodes
    // all edges leaving core id nodes
    // this includes edges inside the core, as well as edges leaving the core
    s->stack(this, stack_locations);


    #ifdef DEBUG
    if (fm::die) {
        cout << this ;
        cout << s ;
        cout << "-CR end-" << endl;
    }
    #endif
```

```
        // calculate one step core ids
        #ifdef DEBUG
        if (fm::die) if (index_revisit.size()) cout << "Revisiting '";
        #endif
        each_it(index_revisit, set<int>::iterator) {
            #ifdef DEBUG
            if (fm::die) cout << *it  << " ";
            #endif
            u12.insert(*it);
        }
        #ifdef DEBUG
        if (fm::die) if (index_revisit.size()) cout << "'." << endl;
        #endif

        if (u12.size()) conflict_resolution(vector<int> (u12.begin(),u12.end()), s, 0, ceiling);

        if (!ceiling) each_it(s->nodewalk, nodevector::iterator) {
            if (it->labs.size() == 0) {
                cerr << "Error! S-Labels left to fill." << endl; exit(1);
            }
        }

        return 0;
    }
    return 1;
}
```