Christian-Albrechts-Universität zu Kiel
Institut für Informatik und Praktische Mathematik
Prof. Dr. W.-P. de Roever
Martin Steffen, Immo Grabe

Verteilte Algorithmen

Wintersemester 2003/04 — **Serie 10** — 20. Januar 2004

Thema : GHS (Aufgaben mit Lösungshinweisen)

Ausgabetermin: 20. Januar 2004

Abgabe: 26. Januar 2004

Aufgabe 1 (GHS (8 Punkte)) Lösen Sie Aufgabe 32 aus Kapitel 15. Halten Sie sich dabei an den informell beschriebenen Ablauf, und verwenden Sie speziell die angegebenen Nachrichtennamen aus [2, Abschnitt 15.5.5].

Solution: The algorithm is described [2, Section 15.5, p. 509]; the original reference is [1].

Let's recapitulate what I remember about the algorithm, especially what is different from the synchronous version. See also the explanation at the exercise about *SynchGHS*. The most important difference seems to be the merge and absorb operation. The synchronization roughly achieves that the levels are in sync, but we have to be careful to find a new unique leader. Perhaps we could do another leader election, but, as already done in the synchronous GHS, we can do smarter.

The core insight, already in the synchronous setting, that given an arbitrary number of level k component that collapse, there is exactly one MWOE *common* to two components.[1] This insight is used to distinguish the *merge* and the absorb operation.

[**To do:** *A couple of things are not yet good. Hardest problem is, that merging is not allowed, if the mwoe is not bidirectional. Another one is that also the leader must send a test. Minor ones is that no report-success is done.*] !?

```
// GHS (Gallager/Humblet/Spira) (Id: ghs.code,v 1.39 2004/01/29 16:47:14 softtech Exp )
Types:
   Level  = Nat;
   Weight = Nat;
   CID    =  Weight * Level + UID                // UID is used only for level = 0
Messages                        // just to get an overview
```

[1] The argument goes like this: we are given ar partitioning of the nodes in connected components, which bild the trees of the forest. Each of the cliques picks one MWOE, i.e., one minimum edge which connects to neighboring clique. It can easily be seen that, considering the cliques as nodes in a "supergraph" with the MWOEs as edges, this yields a (directed) graph with *exactly one cycle*. If a component has an incoming and an outgoing edge and an outgoing edge, then the weight of the outgoing edge smaller or equal then the weight of the incoming edge by the minumality of the choice and the the weights in the undirected graph are "symmetric".. Since furthermore the weights of the edges are unique, the order-relation is strict, if the two mentioned edges do not belong to the same unordered edge of the underlying graph. Since, as said above, there is exactly one cycle in the graph, the cycle must be of length two. In other words: exactly two components choose each other by and MWOE-link, and the two links are inverses.

```
  initiate of CID              // broadcast in component from leader
  test     of CID              // probe some neighbor potentially outside
  report   of ..               // convergecast in component
  accept, reject of unit       // answer to the test
  changeroot of ...            // leader to MWOE-node
  connect   of ...                                  // contact the other component
  probe_finished : bool = false

Signature
  input:   receive(m)_{i,j}; // input comm. with channel process
           wakeup():          // trigger from outside
  output:  start,report_m
  internal: send(m)_{i,j}

States:

  i:      Id                                        // id of the process under consideration
  uid:    UID                                       // unique identifier

  level: Nat = 0;                                   // from level k -> k+1: reducing the
                                                    // number of components. A level-k component
                                                    // as ≥ 2^k nodes.
  cid: CID = uid;                                   // at the very beginning (level = 0), we use uid
  mwoe:                                             // local information concerning MWOE
  core:    Edge of Id x Id  = ⊥ //
  leader : Id = i     // when merging, larger one of core edge will be taken
  send[j], receive[j]: fifo queues for each j∈ nbrs, initially empty

  branch : set of edges   = ∅ ⊆ nbrs // part of mst tree fragment
  rejected : set of edges = ∅ ⊆ nbrs
  basic    : set of edges =  nbrs

Transitions:
  send(m)_{i,j}:             // asynch. network model
     precondition: m is first in send
     effect        remove first element of send[j]

  receive_{j,i}(m):
      effect:      add m to receive[j]


 // ------------------------------------------------------------------------------------------
   t_initiate_and_probe_out
     precondition: receive[j] = initiate(cid')@ r' ∨ leader = i
     effect:
         if    leader ≠ i
         then  parent := j                          // remember where to converge-cast later
               receive[j] := r';                    // remove it from the queue
               cid := cid' (= (cw',l'))             // store the component id (including the level)
         else  skip                                 // the leader never receives an initiate message,
                                                    // and its parent remains ⊥, the cid
                                                    // is provided either by initialization or
                                                    // at the end of the previous level.
         add initiate(cid) to send[branch\parent]   // relay (or spawn, if leader) the broadcast
                                                    // along the sons of the component tree
         add test(cid)     to send[basic];          // probe potential neighbors to
                                                     // determine their component.
                                                     // those from rejected need not
                                                     // be tested.
                                                     // The component id, i.e., the pair
                                                     // of the core weight and the level
                                                     // is sent to the neighbor for a
                                                     // comparison.
                                                     // should be optimized
 // ------------------------------------------------------------------------------------------
   t_probeanswer_or_delay
```

```
    precondition: receive[j] = test(cid') @ r'      // we receive a probe message
                                                    // potentially from a foreign group.
                                                    // cid may (initially) be of the form
                                                    // cid' = uid' for some user id, or
                                                    // cid' = (cw',level')
    effect:
      if     cid' = cid
      then   add  reject() to send[j]                // call-back: I'm in the same component
      else                                           // different component id's: cid ≠ cid'
           if level ≥ l
      then    add  accept() to send[j]               // call-back: I'm in a different component
      else                                           // undecided yet: wait until level reaches l
              pending[j] := delay(cid')              // inform j, when level of cid' reached
// ------------------------------------------------------------------------------------------
  t_probeanswer_delayed:
    precondition: pending[j] = delay(cid') ∧    // request for j still unanswered
                  cid ≥_l cid'                  // but we've caught up, level-wise
    effect:                                           // so we give the belated answer as above
      if    cid = cid'
      then  add reject() to send[j]
      else  add accept() to send[j]
      fi;
      pending[j] := ⊥                           // flush


// ------------------------------------------------------------------------------------------
  t_probe_finished:                                          // get the answers back
    precondition:
       head of receive[j] =  m_j, for all j in basic
                                                    // all acks are back

    effect
       let rejected-ids = id's with reject answer and
           accepted-ids = id's with accept answer
       in  basic    := basic \ rejected-ids;
           rejected := rejected + rejected_ids;
       mwoe := edge(i,j,w) = min accepted-ids        // found a local candidate
                                                    // if accepted-ids = ∅,
                                                    // min is undefined
       probed_finished := true

// ------------------------------------------------------------------------
   t_convergecast:                          // we can collect information
                                            // as soon as we have received all our
                                            // own probes and if all of our sons
                                            // have send their opinion about
                                            // the MWOE. contains the leader/root
                                            // as special case
     precondition:
       have-probed = true ∧
       head of receive[j] = m_j, for all j in branch \ parent
     effect:
        let messages be the set of all heads receive[j], j in branch\parent
        mwoe := min(mwoe, messages)         // build overal minimun; non-strict
        if   leader ≠ i
        then  add report(mwoe) to send[parent]
              remove head for all receive[j] // clear input buffers
        else                                // if we are leader, we have no
                                            // parent. We can therefore decide
                                            // and initiate the next phase.
                                            // the leader now knows the mwoe
                                            // but this information must be
                                            // handed down to the relevant process,
                                            // which is the one in the tree, which
                                            // is mentioned in the mwoe
                                            // this is done via a broadcast along the
```

```
                                              // tree branches
          leader := ⊥;                      // reset, I'm (probably) no longer leader
          parent := ⊥                       // and new parents will be handed out soon
          add changeroot(mwoe) to send[branches];

// ---------------------------------------------------------------------------

   t_broadcast_mwoe:                                  // if a process receives a changeroot-message
                                                      // it must compare whether it's him or not. If not
                                                      // he can forget about it. If it's him, it initiates
                                                      // the connect protocol.
     precondition:
       receive[j] = changeroot(mwoe) = changeroot(edge(in, out, w))
     effect:
       if    i = in                                    // if the node is the connector
       then                                            // time to connect to the partner out there
            add connect(cid.level,uid) to send[out]
            branch := branch + (in,out);
            basic  := basic  - (in,out)
       else                                           // if not, pass it on down
            add changeroot(mwoe) to send[branches\parent]

// ---------------------------------------------------------------------------------------

   t_merge_or_absorb                                 // if a component receives a connect-message
                                                      // it means it is contacted by another one,
                                                      // who has chosen the edge as _his_ mwoe.
      precondition:
        receive[j] = connect(level',uid') @ r'
      effect:
         receive[j] := r';                           // remove it
         if
            level' = cid.level                       // identical levels =>
         then                                        // MERGE!
            if    (uid > uid')
            then                                     // I'm the new leader
                  leader := uid;
                  level := level + 1;
                  parent :=  ⊥;           // the old parents are irrelevant
                  branch := branch + edge(i,j)  //
                  cid    := (weight(i,j), level)
                  add initiate (cid)
            else                                     // Nope, I'm not the new leader, my
                 skip                                // partner j is. I just do nothing,
                                                     // as my machinery will (or has
                                                     // already has) send a connect to the opposite
                                                     // number and this is how j learns about
                                                     // his leadership. I just have to passively
                                                     // wait for the his initiate message, which, for
                                                     // me, opens the next stage
            fi
          else                                       // level' < level : ABSORBE
                                                     // the sender j has to be ``absorbed''
                                                     // i.e., incorporated into our mst
                                                     // the level is not advanced, but the
                                                     // other component has to be brought up-to date
                                                     // this is done using the initiate message
              branch := branch + edge(i,j);
              add initiate(cid) to send[j]                                    //

Tasks:  { ...}
```

initiate This internal transition is responsbible for getting the whole cycle starting. It sends a *broadcast* within the clique of objects and triggers also the test-messages probing the border of the true. Furthermore, it sets for each tree node except the root, the parent

"pointer" to prepare for the convergecast. The undirected tree is given by the *branch* pointers; when the broadcast is done, and during the convergecast, the parent-pointers are the directed spanning tree with the leader as root.

The information broadcast by this message is the component identifier, which is needed to determine the MWOE. In first approximation, everything which identifies the component can be used, for instance the unique identifier of the root. To assist in the synchronization and merge/absorb procedure, however, one uses another piece of information, namely (the weight of) one *internal* edge. Since weights are globally unique[2] this can be used to identifier a component. Additionally, it is useful in to pair with this identifier also the current *level* of the component.

There is a special case, however, and this is level 0. At this stage, the compents consist all of single node, and this there are no component-internal node. We use the user id of the root (which is the single-node, anyhow) as identification. Note, however, that the initiate message is never used with this particular initial component identity, since at level 0, each component has no tree-edges. Therefore the only place where are the *test-message* probing the environment of the single-node component.

This internal transition is not only triggered by some received initiate-message, but also for the (new) *root* at the beginning of a new level of a component; especially it is enabled at the very beginning at round 0, where each node is a root and the set of branches is empty, by initialization. Therefore, no initialization is sent, but a test-message towards all direct neighbors.

probe-answer (`t_probeanswer_or_delay`) The action is part of the query, sent at the rim of a component towards the neighboring one. Each member of a (potentially) neighboring component, and triggered in the course of its initialization phase, has sent us a test-probe along one of its non-branch edges, and wishes now an answer whether we belong to his component. We can compare the sent component id for this purpose; we have to keep in mind here, that the component id in most cases is a pair of an internal edge "identifier" —the unique weight, to be precise— and the level, but at level 0, the node-identity is sent. Also the component id of us might be either such a pair or an process identity (in effect, the self identity in this case).[3] Anyway, since component identifiers are unique, equality of the sent identity with the own immedately allows the answer, that this edge cannot be the MWOE of the requester, since he's in the same component as we (and componts only merge, they never split).[4]

In case the cid's are different, situation is less clear, since we might not yet be aware of some new identity, which is distributed in the process of the merge/absorb procedure which collapses two components. We can, however, use the transmitted level-information to see whether we are lagging behind. If we are at least as advanced as the sender, we can assure him that we are in different components (asserted by an accept-message).[5]

[2]Per *undirected* edge, of course.

[3]Note that the identity $(\_, 0)$ (I guess) is not possible. A component reaches level 1 by merging, which means counting up.

[4]Note that if the sent cid is a user id, then the comparison must yield a false.

[5]Note that the levels are advanced not before the a new leader is determined and the leader is the first one counts up; this starts the whole cycle again and, as said before, the new level is propagated across the next-level

The only case where we cannot give an immediate answer is when `cid` *not* = `cid'` and our level is < than the one received. In this case we just have to wait until we have reached the same level at which point the answer is determined. In delaying an answer to some probing partner, we must cope with the fact that we might have more than one answer pending for a while, but of course at most one along each neighboring links, as the requester blocks for the answer. This late answer is given in the transition `t_probeanswer_delayed`.

probe evaluation Now our perspective is back on the side of the probe-sender: we have sent, and still as part of the initialization phase, along all out potential partners the probing message, to form a "local opinion" about the MWOE. Later we will combine it with information from our sons (if any) in the convergecast towards the root of our current tree[6] Anyway, that the convergecast rolls back in an orderly manner, we just collect our own neighborhood information: when all queried neighbors —they correspond to the list we have remembered in the *basic*-list— have answerd, we adapt the `mwoe`-component. From all the ones which have answered with an accept, we store the edge with the minimum weight as our local canditate. Additionally we appropriately adapt the `basic` and `rejected` bookkeeping of our graph neighbors and prepare ourself for the convergecast setting the `probe_finished` flag.

convergecast Having completed the local neighborhood survey (using the basic neighborghs), the convergecast may begin, echoing back the results to the root for evaluation and further distribution. The direction is given by the parent pointers of each node, which at this moment exactly represent a directed version of the component mst, with all arrows pointed towards the root. During the convergecast, iteratively the minimum is built, such the component-global mwoe is sifted out finally at the root. The covergecast use the report-message, which carries the mwoe edge, including the weight information used in the minimum construction.

inform connector and connect Once the mwoe is determined by the root of the component, we must make use of this edge to really contract the neighboring component. Since we know the connecting node, we just broadcast the mwoe to all nodes in the tree, where everyone ignores it (but passes it on) except the connecting process. This time we need not remember any parents. Now finally, the component via its connector can do the basic indictive step, *enlarging* the component by connecting it to the neighbor using the determined mwoe. To do so it sends the connector uid plus the component level.

merge or absorb Now we switch perspective to the receiver of the connect request. There are two different situations now.[7]

component as part of the initialization broadcast. The fact that we, the process being probed, have the same level or a higher level one than the one sending the query, means that we cannot be in the same component since in one component the identifiers are sent in separate "waves". And especially the probe message is sent *before* the MWOE of the component is determined.

[6] The root corresponds to our current leader, but it seems we don't need a state veriable for that. We might as well use a boolean flag.

[7] Remember the graph-theoretical property, that, under the assumptions given: adding for each group of components a mwoe, gives exactly one length-2-cycle. This is the new core edge and determines the new leader. But actually, this is not so important.

Aufgabe 2 (GHS-Ablauf (3 Punkte)) Lösen Sie Aufgabe 34 aus Kapitel 15 (Seite 528), d.h. beschreiben Sie einen Ablauf des *GHS*be dem eine reject-Nachricht als Antwort auf Test zum einem Zeitpunkt zurückkommt, bei der "Fragesteller" diese Kante als `branch`, also zum MST gehörig klassifiziert hat. Argumentieren Sie, daß das in Ordnung geht.

Solution: First it is clear, that the test-message is sent along *non-branch*-edges, only, which is done in the course of the initiation-broadcast. This means, between the sending of the test and the reception of the reject-answer, the process must have changed added the value to its `branch`-set. Adding a branch is the core step which joins two component (by absorbing or merging). Since the process i, along this edge (i, j) sends only one test-message and "blocks" afterward (on this edge) it will not itself finish his "local survey." and this compent it belongs to will not be able to finish its search for a MWOE, which could result in adding this edge to the branches.

The only way, therefore, that the branch is added is that caused by another component. Especially, the other component can connect to the current component by choosing (j, i), i.e., the edge (j, i) in the reverse direction, as bridging MWOE.

More concretely, a following scenario is possible. Assume processes i and j as members in two (currently) separate components C_i and C_j, with level $l_i > l_j$. The order is important, since we want i to send a test-message, and the opposite number j to delay the answer; therefore the level known at j must be strictly smaller. Anyway, i sends the test to j (transition `t_initiate_and_probeout`) at which point the outgoing edge (i, j) is not part of the branches.[8] "At the same time", j is in the same situation (but at a lower level) and conversly sends his test-message the opposite direction to i, who answers immediately with an accept.

Assume then that component C_j is able to complete the search for a mwoe and it determines (j, i) as bridge to C_i. After receiving the mwoe-broadcast inside C_j, the node j adds (j, i) to his branches and sends the connect-message to i. The level-situation at this point is still unchanged, which means that C_j is *absorbed!* Anyway, when i receives the connect-message in transition `t_merge_or_absorb`, it adds the (i, j) to his branches in turn, [9] and by comparing the level decide that it is engaged in an absorbed. The level is not counted, of course, no new leader is chosen, simply C_j is swallowed. Process i triggers this by sending j an initiate-message, which especially contains the level of C_i which strictly higher than the one of C_j. The initiate-message percolates through C_j, and especially raises the level of j. This finally unlocks `t_probeanswer_delayed`, which sends back a reject, since both are in the same component in the meantime.

For the algorithm, that's ok. Since the edge we discussed is rejected, it won't contribute to any MWOE to the outside. If C_i (or now the C_i-part of the conbined component) had already had gotten some accept-messages, that's ok, since it's level and its root has not changed.

[8] The sets `branch` and `basic` are disjoint.

[9] This makes the subgraph determined by the `branch`-variables symmetric/undirected again.

```
// GHS (Gallager/Humblet/Spira) (Id: ghs2.code,v 1.1 2004/02/10 06:14:53 Steffen Exp )
// another variant (kudos Immo)


Transitions:
//------------------------------------------------------------------------
  send(m)_{i,j}                                        // ''standard out''
    precondition:
      m is first on send(j)
    effect:
      remove first element of send(j)
//------------------------------------------------------------------------
  initiate_search
    precondition:
      status = intitiating
    effect:
      reported := ∅;
      mwoe     := ∞;
      ∀(cid',k) ∈ pending_test
        if         cid'.level = cid.level
        then if    cid'.core  = cid.core
             then
                   rejected := rejected ∪ (basic ∩ k)
                   basic    := basic \ k;
                   add reject to send k
             else
                   add accept to send(k);
           pending_test := pending_test \ (cid',k);
      branch := branch ∪ k, ∀ (k) ∈ pending_connect;
      pending_connect := ∅;
      if   length(basic) > 0
      then status := searching
           add test(cid) to send(first(basic))
      else
           status := reporting
      add initiate(cid) to send(k), ∀ k ∈ branch \ parent
//------------------------------------------------------------------------
  receive initiate(cid')_{j,i}
    effect:
      parent := j;
      cid    := cid';
      status := initiating
//------------------------------------------------------------------------
  receive changeroot(mwoe')_{j,i}
    effect:
      status := connecting;
      mwoe   := mwoe';
      add changeroot(mwoe) to send(k), ∀ k ∈ branch \ parent
//------------------------------------------------------------------------
  receive connect(cid')_{j,i}
    effect:
      if   cid'.level < cid.level
      then branch := branch ∪ j;
           basic  := basic \ j;
           if   status = seaching ∨ status =  reporting
           then add initiate(cid) to send(j)
      else if
                cid'.level = cid.level ∧
                (status = connecting ∨ status = connected) ∧
                mwoe = (i,j)
           then if   i > j
                then parent := null
                     status := initiating
                     cid    := (mwoe,cid.level + 1)
           else
                insert (cid',j) to pending connect
```

```
//-------------------------------------------------------------------------
  connect
    precondition:
      status = connecting
      ∃i∈ nbrs: (i,j) = mwoe
    effect:
      if   (cid',j)∈ pending_connect ∧ i > j
      then
           parent := null
           cid    := (mwoe, cid.level+1);
           status := initiating
      else
           branch := branch ∪ j;
           basic  := basic \ j;
           status := connecting
           add connect to send(j)

//-------------------------------------------------------------------------
  receive (test(cid'))_{i,j}
    effect:
      if      cid'.level < cid.level
              add accept to send(j)
      else if cid'.level = cid.level
           if   cid'.core  = cid.core
           then
                rejected := rejected ∪ (basic ∩ j);
                basic    := basic \ j;
                add reject to send(j);
           else
                add accept to send(j)
      else
           insert(cid',j) to pending_test
//-------------------------------------------------------------------------
  receive (reject)_{j,i}
    effect:
      rejected := rejected ∪ (basic ∩ j);
      if   length(basic) > 0
      then add test(cid) to send(first(basic))
      else status := reporting
//-------------------------------------------------------------------------
  receive (accept)_{j,i}
    effect:
      mwoe := (i,j);
      status := reporting;
//-------------------------------------------------------------------------
  receive report(mwoe'))_{j,i}
    effect:
      if   weight(mwoe') < weight(mwoe)
      then mwoe := mwoe';
      reported  := reported ∪ j
//-------------------------------------------------------------------------
  report
    precondition:
      status = reporting;
      branch = reported;
      parent ≠ null
    effect:
      status := reported;
      add report(mwoe) to send(parent)
//-------------------------------------------------------------------------
  report
    precondition
      status = reporting;
      branch = reported;
```

```
      parent = null
    effect
      if   (mwoe < ∞)
      then status := connecting
           add changeroot(mwoe) to send(j), forall j∈branch
      else status := done
//------------------------------------------------------------------------
```

□

References

[1] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and Systems*, 5(1):66–77, 1983.

[2] Nancy Lynch. *Distributed Algorithms.* Kaufmann Publishers, 1996.