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 2** — 29. Oktober 2003

Thema : Leader Election (Aufgaben mit Lösungshinweisen)

Ausgabetermin: 29. Oktober 2003

Abgabe: 10. November 2003

Achtung: in der kommenden Woche ist so gut wie jeder vom Lehrstuhl auf einer Konferenz. Demzufolge ist die Abgabe des Zettels den Montag die Woche drauf.

Aufgabe 1 (Time Slice (3 Punkte)) Geben Sie den Code für den *TimeSlice*-Algorithmus an (Aufgabe 3.10).

Solution: The time slice algorithm for leader election in a uni-directional, synchronous ring of known size is given informally at page 35 of [**?**]. It's idea is rather simple and exploits the fact that the total order has a known[1] potential minimum element, namely 0. The algorithms proceeds in phases, where each phase is exclusively reserved to circulate a particular id once around the ring. In phase 0, it's the turn of process with id 0, if it exists, to initiate the round trip. After n rounds, it would notice that the token arrives back, and it can announce itself as leader. If process 0 does not exist, process with id 1 has its chance, etc.

[1] As opposed to some minimum unknown element which always exists in a finite, total order given by the UID.

```
messages : {ping}

// --- state --
u = uid: UID
phase = 0  : int
const  N  : int     // known size of the network
round = 0 : int         // counter for round
status = unknown : {unknown, leader, nonleader};

// message generation

send the current value of send to i+1                // ring- topology

// transition function

send := null;                                       // clear channel
v = incoming message;                               // can be ⊥
case (round, phase)
     (0,uid)  ->  if   status = unknown               // I'll have one try
                  then send := ping;                  // if no one else
                  round := round +1;                  // has been elected yet
     (N,_) ->
           case (v,phase) of
              (ping,uid)  ->  status := leader
              (⊥,_)       ->  status := non-leader    // I had my chance
              (ping,_)    ->  raise error              // must not happen
           round := 0;                                 // reset round  and
           phase := phase + 1;                         // enter next phase
     (_)  ->
           case (v) of
               ping   -> status := non-leader
               ⊥ -> skip
           round := round +1
           send  := v
```

Remember the synchronous execution model. Initually, the system starts with all channels empty in a state, given by the initial values of all variables. In our case, we start in round 0 and in phase 0. After that, the regular behavior of the rounds starts with round 1.[2]

$$c_0 m_1 c_1 m_2 c_2 \ldots$$

Since we have no lossy channels currently, we do not need two separate states for the buffer, one of the original content and one where some of the messages are lost. The message generation is rather simple. As usual it just "pumps" the content of specficied variables into the channel. In our case, it's the variable `send`.[3]

Remarks: the classical stumbling stone: one forgets to stop the phases, i.e., one forgets that a process must recognize when it becomes a *non-leader!* The above code sends just one `ping`. One could also use the unique identifier. This would simplify some of the cases.

Aufgabe 2 (Verbesserter *OptFloodMax* (4 Punkte)) Consider the "further optimized" version of *OptFloodMax* described in Section 4.1.3, which prevents processes from sending *max-uid*-information to processes from which they have previously received such messages.

[2]Round 0 in the model is not a "full" round as the message generation and consumption is missing.

[3]The code given could be slightly optimized. It "misses a beat" initially, since by the initial states, no message is generated, while one could probably start with `ping` right away. The same stutter seems to occur when a phase is completed; also then, there is a round with no messages. This means the variable `round` is not in sync with the actual rounds. Probably this could be made nicer

1. Give the code for this algorithm.
2. Prove the correctness of your algorithm by relating it to *OptFloodMax*, using the same sort of simulation strategy used in the proof of correctness for *OptFloodMax* (i.e., in the proof of Theorem 4.2).

(Aufgabe 4.4)

Solution: *OptFloodMax* is an improved version of the simple *FloodMax* algorithm for leader election in a general, synchronous network. The trivial flooding algorithm is rather brute force. It just floods the network with the maximum value encountered so far. Since the diameter of the network is known, each process just stops after pumping its knowledge about the maximum into the network enough times — keeping a record of the maximum encountered so far.

The improved version of the algorithm (*OptFloodMax*) [**?**, p. 54] uses the following simple observation: only news should be propagated since it makes no sense in repeating oneself in sending out the same values over and over again. In other words, the locally known maximum is sent only in a round, when the incoming information *updates* this value.

Part 1 of the exercise, where one adds another bit of optimization, is rather trivial. One can use the code of the optimized flooding algorithm almost completely; only the set of channels where to send the outgoing messages is redefined: it makes no sense to send the newly learn maximum back to those processes from which one has learnt it. Note that in one round, the might be more than just one incoming channel from which a process receives the new optimum.

```
Process(i)
    messages : {ping}

    // --- state --
    id       = uid  : UID
    max-uid  = id   : UID           // initually, I'm the largest, as far as I know
    new-info = true : boolean
    round = 0 : int                 // counter for round
    status = unknown : {unknown, leader, non-leader};

    // message generation

    if   rounds < diam and new-info = true
    then send max-uid to all {j ∈ out-nbrs | incoming uid from j ≠ max(u) }

    // transition function

    round := round + 1;
    let u be the set of uid's arriving from in-nbr

    if max(u) > max-id
    then new-info := true else new-info:= false
    max-uid := max(max-uid, u);          // state change only if new-info = true
    if     rounds = diam
    then   if max-id = id
           then status := leader
           else status := non-leader
```

For part (2), we have to give some proof, basically adapting the proof for *OptFloodMax*. The assertional proof uses two auxiliary assertions, where the second one uses the *simulation* technique. Basically this allows to

1.

2.

Aufgabe 3 (*SyncBFSMax* (6 Punkte)) Consider the optimized version of SynchBFS described in Section 4.2.2, which prevents processes from sending search messages to processes from which they have previously received such messages.

1. Give code for this algorithm.
2. Prove the correctness of your algorithm by relating it to SynchBFS, using the same sort of simulation strategy used in the proof of correctness for OptFloodMax (i.e., in the proof of Theorem 4.2).

(Aufgabe 4.6)

Solution: This algorithm is an improvement of the synchronous breadth-first search algorithm (*SynchBFS*). Let's first recapitulate the basic algorithm (cf. [**?**, Section 4.2]. The algorithm assumes a strongly connected process graph with an explicit root node, which is intended as the root of the breadth-first search tree.

The algorithm works conceptually similar to the ordinary, sequential breadth-first traversal of a graph, i.e., it uses markings to distinguish visited and not-yet-visited nodes. [4] Initially, the root node is marked, the rest of the graph unmarked. The code for the basic algorithm is shown in Figure **??**

The algorithms is stupid insofar, that a freshly discovered nodes indiscrimitately sends search messages to all its neighbors. Better is to send messages in a more directed manner "outwards", away from the root node. More precisely, there is no sense in sending messages back to "discoveres"[5] The reason is that a seach message can originate only from a process which is marked (and, in the previous round, just discovered). A marked process, however, does not react to any message. In the terminology of Section 2.1, the state of a marked process with `just-discovered = false` is a *halting state.*

The optimization leads to the code of Figure **??**. The only difference is in the message generating function: the process does not use the static `out-nbgr`, i.e., the list of outgoing neighbors to send the exploration messages, but only those from its neighbors from which, in the previous round,[6] there had been no incoming messages.

Conceptually, there are *three possible* stati, which are represented in the optimized version as three colors: white for not yet touched, grey for just discovered, and back for finished i.e., for discoverd but not right now.

For the proof, it was required to do a *simulation* proof connecting the optimized version with the original one. As inspiration, one could have a look at the way, the optimized flooding algorithm was related to the plain one. In the way presented in [**?**] so far, the simulation proof method can be seen as a special form of assertional invariant proof. What make the situation slightly dubious here is that the algorithms are *nondeterministic!* At least if we take the BFS tree into account! Anyway, here we go.

> **Assertion (1):** For both algorithms, the following holds: If $\texttt{status}_i$ `= grey` and $j \in \texttt{in}_i$, the $\texttt{status}_j$ `= black`, for all rounds r.

[4] The presentation of the sequental algorithm in for instance [**?**] uses three different marks: white, grey, and black. Those are also visible in the algorithm here:

[5] Let's call the set of discovers of a node those processes that sends messages to the process which lead to the step from the unmkarked to marked status.

[6] In the model, message generation by convention is the *first thing* to do in a round.

```
Process(i:id)
    messages : search of id

    // --- state --

    parent = ⊥ : process
    leader : process = i  // initially, each one is his own leader
    mark   = if  (i = i0)
             then marked
             else unmarked      : {marked, unmarked}

    just-discovered = if   (i = i0) then true else false : bool

    in = ∅ : set of ids

    // message generation

    if   just-discovered
    then search(i)  to out-nbrs
    else skip

    // transition function

    in               = incoming search messages;
    just-discovered  := false;
    case mark of
      unmarked -> just-discovered := true;      // triggers further  messages
                  mark            := marked;
                  parent          := pick(in);  // choose a parent
      marked   -> skip
```

Figure 1: Breadth first search

```
Process(i:id)
    messages : search of id

    // --- state --

    parent = ⊥ : process
    mark   = if    (i = i_0)  // i_0 = root
             then  grey
             else  white        : {white, grey, black}

    in = ∅ : set of ids

    // message generation

    if   mark = grey
    then send search(i) to (out-nbrs \ in)   // optimization
    else skip

    // transition function

    in               = incoming search messages;
    case (mark,in) of
         (grey, _)   ->
              mark   := black
         (white, ∅) -> skip      // no input, no reaction
         (white, _)   ->
              mark   := grey                // triggers messages
              parent := pick(in);           // choose a parent
         (black,_)    -> skip
```

Figure 2: Breadth first search (optimized)

Proof: By induction on the number of rounds and for both versions of the program, the argument is the same. In the initial state, i.e., for $r = 0$, the assertion holds vacuously, as only the root node is grey and for the root the set of incoming ids is empty.

Now, for the inductive step consider the following: from the transition function for the message generation at the beginning of round $r + 1$, only *grey* nodes send messages to their neighbors; both versions of the BFS differ in this step, but agree in no non-grey nodes ever send messages. Now the grey nodes in round $r + 1$ are, again by looking at the code, those which had beed white before and which receive at least *one* message in $r + 1$. Converting the perspective from sender to receiver, in the transition function in round $r + 1$, only messages originating from grey nodes (at round r) are stored in `in`. Additionally, all grey nodes of round r are painted black in round $r + 1$, from which the assertion follows. □

We can use this observation for the simulation.

Assertion: for any round r, the values of `mark` and `parent` correspond[7]

Proof: Procceed by induction on the number of rounds r. For $r = 0$, i.e., for the initial states, the two algorithms are obviously in relation.

Now, assume that up to completion of r the assumption holds. The next thing to happen is the message generation, and afterwards the state transition of round $r + 1$.

For message generation, the difference is that a grey nodes of the original algorithm sends to *all* its outgoing neighbors, while the optimized version only to those not in `in`.

So all nodes not the at the receiving end of a communication with a grey node will still be identical at the end of round $r + 1$. So let j be such a neighbor of a grey node i. As mentioned, the only case where the algorithms differs is when $j \in \texttt{in}_i$, so let us concentrate on processes j this case. By Assertion (1), j is black.[8] Now the transition of round $r + 1$ for the black node j does nothing, preserving the invariant. □

[7] for the parent pointer, one has to assume that the nondeterministic choice picts the same.

[8] The values so far are still meant at the end of round r.