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 3** — 12. November 2003

Thema : **Spannbaum (Aufgaben mit Lösungshinweisen)**

Ausgabetermin: 12. November 2003

Abgabe: 17. November 2003

Aufgabe 1 (Bellman-Ford (4 Punkte)) Bearbeiten Sie Aufgabe 4.14 aus [1], d.h., geben sie den Code für Bellman-Ford an und beweisen Sie die Korrektheit. Benutzen und Beweisen Sie dabei die Invariante aus der Vorlesung (Seite 62 unten).

Aufgabe 2 (*SynchGHS* (4 Punkte)) Geben sie den Code für den Algorithmus von Gallager, Hamblet, and Spira (*SyncGHS*) an. (Aufgabe 4.15)

Solution: The synchronous version of the algorithm is described in [1, Section 4.4, p. 66]; the original algorithm is asynchronous. The original citation is [**?**].

As a recap concerning the synchronous model and its executions: we start in some initial state (that's round 0, so to say) and from then on, the rounds are *message assignment* followed by a *state assigment.* The message assigments are produced by the message generation function of the processes; afterwards (in one round), the process reads the vector of incoming messages along all its neighbors and produces a state change.

Now to the algorithm, as far as I remember. Given a connected, undirected, weighted process graph, the goal is to determine a *minimum spanning tree.* The underlying idea of the algorithm is the same as known from corresponding sequential MST-algorithms. Starting from an empty subgraph, the algorithm adds one edge after the other, maintaining a subset of the/a spanning tree, until the "spanning forest" connects the full graph. Thus, the growing forest partions the graph at each state in connected components,[1] the single trees. Starting with the empty subgraph, i.e., n single-node components, the inductive step adds one edge combining exactly two components. Joining two tree components mainains acyclity, and to preserve also the *minimality*-requirement, the trick, given one components, is to pick a *minimal* edge which connects the component with another.[2] The choice of component in each step distinguishes the main to sequential variants of the algorithm (Prim's and Kruskal's version).

Here, things get more complex, since the algorithm is intended to run in parallel. This, especially, refers to mentioned main step joining trees. While, depending on the variant, the

[1] The trees are *undirected* as the graph.

[2] In the proof as presented in [**?**], edges with this property which can be added maintaining the invariant, are called *safe* edges.

sequential algorithm treats one component at a time, in the parallel version, at least one tries to let the trees grow in parallel. That this works requires that the two indendantly acting components take the same choices, and this is guaranteed if the weights are *unique.*[3]

Let's first consider how a tree grows larger. Globally, the tree must find out the edge connecting to another tree which is of minimal weight; this edge is unique by assumption. This cannot be locally determined, but all nodes in the tree must contribute and the minimum is the collected in a *convergecast* process. Also locally, a node must find out which of its graph neighbors belong to its own same tree. To be able to do so, each node must have knowledge to which clique it currently belongs. For this purpose, one node in the tree, the *leader,* is used to represent the tree (and furthermore is useful for distributing and collecting information.)

Once a component has determined the unique edge to add, the two trees are combined. This means, a new leader is determined and the new leader must be broadcast to all nodes.

The algorithm so far seem simpler than it is. For instance it's not case that only pairs of components merge. Component T_1 determines that it connects to T_2 and T_2, at the same time determines T_3. Second of all, even in the synchronous execution model, the same time does not immeadiately mean that, for instance, both parts come to the decision at the same time. Furthermore we have to be very careful, in this context, that at each round *at most one* message can be sent in each link. For instance, a node cannot respond to a search-message handed down from one's own leader and to a test-message from a neighbor tree in the same round. If we want to avoid buffering in the process —which we want— we need to sequentialize the phases appropriately.

The tree is undirected, by which we mean that each node in the tree has not only a parent pointer, as given by the (synchronous breadth-first) spanning tree algorithm, but also a a pointer to the sons, which is provided by the echoeing convergecast which determines the minimum

If a process i is non-leader and receives a search-message, then it must find out for all his non-tree neighbors whether he's belonging to the same component or not. Furthermore, it must pass-on the search-message further along the tree-component. The test-message is directly acknowledged, and this establishement takes thus takes 2 rounds. Of course a component does not send all its test-messages at the same time.

[3]One can easily prove that under this additional assumption, the MST is unique. Therefore there will not be "contradicting" choices when joining two cliques, which could lead to cycles. This is not to that two cliques connect each other at the same point in time in a bidirectional way.

```
// GHS (Gallager/Humblet/Spira), synchronous version

message alphabet M:
   test:         UID
   test_ack:    {inside, outside}
   search:      UID              // leader broadcast his id in the tree

states_i:
  uid       : UID                // as usual, unique identifier, ordered
  level     : Nat = 0            // counts number of ''mergings''
  treeedges :     = ∅ // ⊆ nbgrs = in-nbrs = out-nbrs
  leader    : bool   = true      // initially, every one is his own leader
  parent: Id         = ⊥
  sons  : Set of Id = ∅ // at the end of each level:
                                  // parent of all sons = process itself
                                  // and vice version: process is contained in
                                  // the sons of the parent (except leader)

msgs_i:

   // The following message generation uses disjoint edge sets; furthermore
   // it can be shown that in each round, at most one of the cases produces
   // a non-empty message, and this globally.
   send to_outside  to all j∈ nbrs \ (sons ∪ parent);
   send to_sons to all j∈ sons       // initially, only the leader does so

trans_i:
   if stage = exploring
     in      := incoming search(id) message;  // there can be at most one along the tree
     if in ≠⊥
     then
        leader := id;                          // now I know my leader = id of my root
        to_sons := search(id);                 // hand it down in the message phase
     fi
     k := k + 1;                              // we do some flooding
     if   k = f(level)                        // if f(level) is big enough, then to_sons = ⊥
     then stage := lookaround

  if  stage = lookaround
  then
     k:=0;                                     // reset the level-internal counter
     to_outside := test(leader);               // the partner can deduce my id
     stage = lookaround2
  fi
  if  stage = lookaround2
  then
     to_outside[j] = ⊥, for all j      // reset
     for each test(leader') from nbrs_j
         if    leader' ≠ leader
         then  to_outside[j] = inside;
         else  to_outside[j] = outside;     // in the sync. setting, one could remain silent
     stage := build_min;
     k     := 0                              // just reset again

  if stage = build_min                       // we pump the min upwards, towards the root = leader
  then
     if k = 0                                // only for k=0, information comes from outside
     then  min := min{weights of edges from where a ''outside'' has been sent};
     else  min := min{min, converge(min)};
     fi
     k := k+1;
     if   k = f(level)
     then k := 0                          // reset
          stage := add_edge;              // now each leader knows the minumum edge
                                          // but the edge is not known to the process
                                          // which it connects to the outside
```

```
fi

if stage  = add_edge                    // tell the new edge, or finish
then
    if leader and max = ⊥
    then                                // no more edge can be added => finito
         stage := finalize
    elseif leader and max ≠ ⊥
    then   to_sons := (i,j)
    else                               // non-leader
         to_sons := edge(i,j), where edge(i,j) is from in-nbrs
         if id = i
         then  new_edge := (i,j)
    fi
    k := k+1;
    if    k = f(level)
    then  stage = merge
else
fi
if stage  = merge                       // now: we need to add this information and
                                        // to merge the trees. Each clique may add
                                        // a different edge, this means, temporarily
                                        // the sub-structure is directed.
                                        //
                                        //
                                        //

// if stage = exploring

// leader searchs the component environment
//     send test(leader) to all j ∈ nbgr \ nbrs_t
```

Aufgabe 3 (Convergecast (4 Punkte)) Geben Sie den Code für den Convergecast Algorithmus an, der am Ende von Abschnitt 4.4 skizziert wird. (Aufgabe 4.20).

References

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